On the Computational Power of Permutations in Sorting and Searching Problems

Gianni Franceschini

September 15, 2005

Addr: Dipartimento di Informatica, Via F.Buonarroti 2, 56127 Pisa, Italy.
E-mail: francesc@di.unipi.it. Homepage: http://www.di.unipi.it/~francesc.
Tel: +39 050 2212796. Fax: +39 0502212726.
Thesis supervisor:

Prof. Roberto Grossi
Abstract

In this work we face the issues of space efficient computations in sorting and searching problems. During the years, the concept of space efficiency has been subject to many different interpretations. We are interested in the most restrictive among the interpretations. The elements in the input set are drawn from a totally ordered and possibly infinite universe $U$. The total number of locations that any computation can use is equal to the number of elements. At any given moment of its execution, the computation is allowed to retain a constant number of auxiliary numerical values at most. Finally, the elements are completely abstract: any two elements can only be compared or exchanged in their locations. In this setting, the number of comparisons and the number of exchanges performed during a computation are the natural measures of complexity. We will adopt also other measures of complexity since we extend these requirements to computational models other than the RAM model and having multi-level memory hierarchies, namely the External-Memory model and Cache-Oblivious model.

From a completely abstract point of view, what we are going to deal with are mere permutations of the input set $S \subset U$. Any algorithm for our setting is a function $f$ from and to the set of permutations of $S$. The function can be calculated retaining at most a constant number of numerical values of $O(\log |S|)$ bits each at the same time. The computation of the function for any sequence in input can be decomposed into a sequence of simple steps of two kinds, exchanges and comparisons. They are functions from and to the set of permutations of $S$ too but in the second case the function is just the identity. Hence, any comparison needs to have side effects on the set of numerical values that we allow to be maintained by $f$. Since the set of numerical values simultaneously retainable by $f$ has cardinality $O(1)$, most of the effects of comparison steps seem to have to be spent too locally in the computation. If that were the only way to maintain information in this permutation-based model of computation, a larger quantity of comparisons would seem to be needed to solve a problem, with respect to a model allowed to use a linear number of numerical values and auxiliary locations to permute efficiently the input elements.

As we will see in this work, that is not the case since the permutations themselves can be used to retain auxiliary information without disturbing the evolution of the computation and paying only a constant slowdown factor in the complexity bounds. We will find optimal solutions for some well-known open problems related to sorting
and searching.

We will show how a set of elements drawn from a generic universe can be sorted using the asymptotically optimal number of comparisons, data moves and auxiliary space. After that, we will prove that the same result can be accomplished respecting the additional constraint of stability in case the elements in input form a multiset, that is preserving the original relative order of the occurrences of the same element in the input sequence. Then, we will depart for a little while from the classical RAM model and concentrate our efforts on the Cache-Oblivious model. In this setting we will prove the existence of a sorting algorithm that is work, cache and space optimal simultaneously. After the Cache-Oblivious model, we will move to a natural extension of the RAM model well suited to study space optimality when the input elements are drawn from multidimensional domains. In this setting we will show the first sorting algorithm for vectors and records achieving the asymptotic optimality for scalar comparisons, moves and space. We will conclude our efforts in the field of sorting-related problems with the generalized merging problem. In this problem $s$ sorted sequences have to be fused into one. Space, comparison and move optimal solutions were known so far only for the case $s = O(1)$ (though some ideas for the case $s = O(\text{polylog} n)$ can be found in the previous work on the subject, as we will see). We will show how this can be achieved for any value of $s$.

After the sorting-related problems, we will focus on searching-related problems. We will start with the space optimal searching of a set of $k$-dimensional vectors. If the input elements are arranged in lexicographically sorted order, we know from an important previous result that upper and lower bounds for the time complexity match. Surprisingly enough, the matching bounds are not equal to the natural bound of $O(\log n + k)$ that can be seen as a generalization of the well-known optimal bound of $O(\log n)$ comparisons for searching a set of unidimensional elements laid out in sorted order. We will show that, for any set of vectors, there exists a permutation of them that can be searched in $O(\log n + k)$ time and hence, that the lexicographical order is not the optimal arrangement for space optimal searching of vectors. After that result, we will go back to unidimensional domains and face a dynamic problem: the implicit dictionary problem. In this problem a dynamic set of elements has to be maintained permuted (without any other resource other than the locations for the permutation itself and $O(1)$ locations for auxiliary numerical values) so that space optimal search, insertion and deletion operations are supported with optimal time bounds. First, we will consider the External-Memory model and show that an optimal solution for the implicit dictionary problem exists under a reasonable condition over one of the parameters of the model. Moreover, we will show how the same solution can be adapted for the RAM model in order to improve the best bounds known for the problem disproving a long-standing conjecture. After the External-Memory model, we will focus on the RAM model again. We will improve significantly the bounds for the search operation in the worst case and for the update operations in amortized sense. Then, we will keep moving in the direction of amortized analysis, but considering once again the more general Cache-Oblivious
model. The result of these efforts will be the first implicit dictionary for the Cache-Oblivious model (and hence also for the RAM and External-Memory models) with optimal worst case bound for the search operation and optimal amortized bounds for the update operations. Subsequently, we will disprove a very old conjecture about implicit dictionaries. We will show that there exist implicit dictionaries supporting searches in polylogarithmic time and updates with a constant number of data moves in the worst case. Finally, we will manage to solve definitely the implicit dictionary problem for all the three models considered, giving a new structure achieving the worst case optimal bounds also for insertion and deletion operations. With this last result we will also answer to a very old question left open since the introduction of the first implicit data structure, the well-known Williams’ heap [Williams, 1964]. We can finally give a positive answer: there exists a structure having the same characteristics of Williams’ Heap (though much more complicated) that is optimally searchable.
“Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer.”

— Donald E. Knuth,
*The Art of Computer Programming, Vol. 3: Sorting and Searching*
Acknowledgments

Thanks to everyone who sustained (in every meaning of the word) me during these last three years of my Ph.D. education. I am particularly grateful to my parents for their unconditional support. I am also very grateful to the Algorithms and Data Structures Research Group in the Department of Computer Science at the University of Pisa for the absolute freedom I have been allowed in my research efforts. I would also like to thank the referees of this thesis, Professor Jyrki Katajainen and Professor Rudolf Fleischer, for their helpful comments.

Finally, I am immensely grateful to my supervisor Professor Roberto Grossi for his teachings, advice and support...and for bailing me out of trouble more than once :-) .
Contents

1 Introduction ............................................. 21
  1.1 Computing with Permutations ......................... 21
  1.2 Models .............................................. 22
  1.3 Problems ............................................ 25
  1.4 Contributions and Organization of this Work ........ 28

I Models, Problems & Tools ................................ 33

2 Computational Models .................................... 35
  2.1 A Lot of Possibilities ............................... 35
  2.2 The In-Place RAM Model for Vectors ................. 37
  2.3 Hierarchical Memory Models ......................... 39
  2.4 The External-Memory Model ......................... 40
  2.5 The Cache-Oblivious Model ......................... 41

3 The Problems ............................................ 45
  3.1 Sorting Problems ................................. 45
    3.1.1 Sorting with minimum data movements .......... 46
      3.1.1.1 Comparison, move and space optimal sorting . 47
      3.1.1.2 How difficult is to achieve the stability? ... 49
    3.1.2 Sorting in the Cache-Oblivious model .......... 51
      3.1.2.1 Funnelsort .................................. 51
      3.1.2.2 Distribution sort ............................ 51
      3.1.2.3 Proximity Mergesort ......................... 52
    3.1.3 Sorting in a multidimensional domain .......... 53
      3.1.3.1 Space optimality and multidimensional domains ... 53
      3.1.3.2 Previous sub-optimal solutions ............... 54
      3.1.3.3 The solution and its implications .......... 55
    3.1.4 Generalized merging problem and adaptive sorting ... 56
      3.1.4.1 The generalized (multiway) merging problem ... 56
      3.1.4.2 Sub-optimal solutions ....................... 56
      3.1.4.3 The problem is difficult even for $s = O(1)$ ... 57
3.1.4.4 Links with adaptive sorting ...................... 58
3.1.4.5 Sorting by merging or merging by sorting? .... 58

3.2 Searching Problems ...................................... 59
3.2.1 Searching in a multidimensional domain .......... 59
3.2.1.1 Simple description, difficult solution .......... 60
3.2.1.2 No sorting? Better searching! ................. 61
3.2.2 Implicit dictionaries ................................... 61
3.2.2.1 Implicit dictionary and their dynamics .......... 62
3.2.2.2 Previous work on implicit dictionaries .......... 63
3.2.2.3 Optimal implicit dictionaries over unbounded universes do exist. ......................... 65

4 Tools ......................................................... 69
4.1 Tools ....................................................... 69
4.2 Bit Stealing .............................................. 70
4.3 Internal Buffering ........................................ 72
4.3.1 Full-power internal buffering ..................... 72
4.3.2 Moderate and adaptive internal buffering ........ 73
4.3.3 Internal buffering and implicit data structures .... 74
4.4 Sorted Order Maintaining Algorithms .................. 75
4.4.1 A natural problem .................................... 75
4.4.2 The basic solution .................................... 75
4.4.3 Maintaining the invariants ......................... 76
4.5 Searching a Set of Vectors ............................. 77
4.5.1 Hirschberg's linear scan ............................. 77
4.5.2 Manber and Myers' searching algorithm .......... 78

II Sorting ....................................................... 81

5 Checkmate in O(n) Moves .................................... 85
5.1 The Problem ............................................... 85
5.1.1 The result ............................................. 86
5.1.2 The algorithm in a nutshell ......................... 87
5.2 Sorting with an Additional Memory .................... 88
5.2.1 Buffer memory ....................................... 89
5.2.2 Structure of the segment memory .................. 89
5.2.3 Structure of the frame memory ..................... 90
5.2.4 Relationship between the frame and segments .... 91
5.2.5 Structure of the pointer memory ................... 92
5.2.6 Inserting elements in the structure ............... 93
5.2.7 Extracting in sorted order — frame level ...... 95
5.2.8 Extracting in sorted order — segment level ....... 96
5.2.9 Rebalancing at the segment level ........................................... 99
5.2.10 Halving a segment ................................................................. 101
5.2.11 Rebalancing at the frame level ............................................. 102
5.2.12 Summary .................................................................................. 107
5.3 In-Place Sorting ........................................................................... 107
  5.3.1 Building a pointer memory ..................................................... 107
  5.3.2 Partition-based sorting .......................................................... 109
  5.3.3 Handling short blocks ............................................................. 114
  5.3.4 An alternative solution ......................................................... 114
5.4 Conclusion ................................................................................... 114

6 Stability is Very Difficult to Achieve ................................................... 117
  6.1 Introduction .................................................................................. 117
  6.2 The Algorithm in Brief .............................................................. 118
    6.2.1 What’s new? ........................................................................... 119
    6.2.2 Stealing bits ......................................................................... 119
    6.2.3 Extracting distinct elements .................................................. 120
    6.2.4 Sorting in presence of many distinct elements ....................... 121
    6.2.5 Sorting in presence of few distinct elements ......................... 121
  6.3 Stealing Bits ................................................................................. 122
  6.4 Extracting a Set of Distinct Elements ........................................... 123
    6.4.1 Main cycle of the buffer extraction ........................................ 124
      6.4.1.1 First phase: collecting some placeholders ....................... 124
      6.4.1.2 Second phase: collecting the distinct elements ............... 125
      6.4.1.3 Third phase: collecting the placeholders back ............... 125
    6.4.2 Managing a growing set of distinct elements compactly .......... 126
      6.4.2.1 Abstract problem .......................................................... 126
      6.4.2.2 The structure ............................................................... 129
  6.5 Sorting with Many Distinct Elements .......................................... 134
    6.5.1 Sorting $b$ elements with $b$ distinct placeholders .................. 134
      6.5.1.1 The structure .................................................................. 135
      6.5.1.2 Traversing the structure .................................................. 138
    6.5.2 The fragmented multi-way merging ....................................... 139
  6.6 Sorting with Few Distinct Elements ............................................ 142
    6.6.1 Sorting with two kinds of internal buffer .............................. 142
      6.6.1.1 First phase .................................................................... 143
      6.6.1.2 Second phase .................................................................. 143
      6.6.1.3 Third phase .................................................................... 145
    6.6.2 Sorting $CB$ ......................................................................... 147
  6.7 Conclusion ................................................................................... 148
7 The Value of Proximity
   7.1 Introduction ........................................... 150
   7.1.1 Memory hierarchies and the Cache-Oblivious model ...... 150
   7.1.2 The result in this chapter .......................... 151
   7.1.3 Organization of the chapter .......................... 152
   7.2 The Sorting Problem & the Cache-Oblivious Model ........... 152
   7.2.1 Funnelsort ........................................... 153
   7.2.2 Distribution sort .................................... 154
   7.3 Proximity Mergesort .................................... 156
   7.3.1 The intuition ....................................... 156
   7.3.2 The algorithm ....................................... 158
       7.3.2.1 The strategy ................................... 159
       7.3.2.2 Distribution of segments ....................... 160
   7.3.3 Complexity analysis ................................ 161
   7.4 Optimal In-Place Sorting ................................ 162
   7.4.1 Encoding the integers ............................... 163
   7.4.2 Creating “virtual” working areas .................... 165
   7.5 Conclusions and Open Problems .......................... 167

8 What if the Universe is Multidimensional? .................. 169
   8.1 Introduction ........................................... 170
   8.1.1 The model .......................................... 170
   8.1.2 Fundamental problems in multidimensional domains ...... 171
   8.1.3 Implications of our result .......................... 173
   8.1.4 Chapter organization ................................ 174
   8.2 Comparison to Recent Work ................................ 174
   8.2.1 Stealing bits from vectors ........................... 174
   8.2.2 Internal buffering with vectors ..................... 175
   8.2.3 Sorting few vectors .................................. 175
   8.3 High-Level Description .................................. 176
   8.3.1 Heavy bits .......................................... 177
   8.3.2 Buffering and session sorting ........................ 178
   8.3.3 Sorting each \( \mathcal{M}_i \) individually ............... 180
   8.3.4 Known tools. ........................................ 180
   8.4 Heavy Bits .............................................. 181
   8.5 Buffering and Session Sorting ............................ 183
   8.5.1 Right-bounded permutations .......................... 184
   8.5.2 Session sorting ...................................... 186
   8.6 Sorting Each Block Individually .......................... 188
   8.6.1 Pivots and buckets in \( \mathcal{M}_B \) ...................... 189
   8.6.2 Routing among the pivots ............................ 190
   8.6.3 Maintaining the structure ........................... 192
   8.6.4 Sorting the buckets ................................. 193
9 Sorting by Merging or Merging by Sorting?  
9.1 Introduction .................................................. 196
  9.1.1 The problem .............................................. 196
  9.1.2 Previous work ........................................... 197
    9.1.2.1 Previous efforts .................................... 197
    9.1.2.2 Adaptive sorting .................................... 198
  9.1.3 Less constrained variants of the problem ............... 198
  9.1.4 Our theoretical contribution ........................... 199
  9.1.5 Organization of the chapter ............................ 200
9.2 Assumptions, Tools, Obstacles and Basic Ideas ............ 201
  9.2.1 Assumptions ............................................. 201
  9.2.2 Tools .................................................. 201
  9.2.3 Some of the obstacles posed by the problem .......... 202
  9.2.4 Basic ideas ............................................ 203
9.3 Merging by Sorting ........................................... 204
  9.3.1 What if $\log^2 n \leq s \leq n^\epsilon$? .................... 204
  9.3.2 What if $s < \log^2 n$? ................................. 206
    9.3.2.1 Breaking the subsequences .......................... 206
    9.3.3 Layout of the memory and invariants .................. 207
      9.3.3.1 Merging phase .................................... 208
  9.3.4 What if $s > n^\epsilon$? ............................... 210
9.4 Conclusions .................................................. 210

III Searching .................................................... 213

10 No Sorting? Better Searching! ............................... 217
  10.1 Introduction ............................................. 218
    10.1.1 When the sorted order is not ideal .................. 218
    10.1.2 The result ........................................... 219
  10.2 Overview .................................................. 222
    10.2.1 First step: efficient bit stealing with $k$-dimensional elements .... 223
    10.2.2 Second step: a more parsimonious search algorithm ........ 224
  10.3 Permuting for Encoding Bits of Information .............. 225
    10.3.1 The ditch: A basic tool for encoding bits ............. 226
    10.3.2 Encoding large ditches ............................... 228
    10.3.3 Preprocessing array $\mathcal{F}$ ....................... 229
  10.4 Searching with Few Bits of Information .................. 230
    10.4.1 Restricting the range for searching .................... 231
    10.4.2 Searching within a twin interval ....................... 232
    10.4.3 Searching in the bit probe model ....................... 235
10.5 Conclusions ................................................................. 235

11 The “Implicitus” B-Tree .................................................. 237
  11.1 Introduction ............................................................ 237
    11.1.1 The problem .................................................... 238
    11.1.2 The implicit B-Tree ......................................... 239
    11.1.3 Basic assumptions and organization of the chapter .... 241
  11.2 High-Level Structure of the Implicit B-tree ..................... 242
    11.2.1 Basic characteristics ....................................... 242
    11.2.2 Routing the search .......................................... 243
    11.2.3 Inserting an element ....................................... 245
    11.2.4 Deleting an element ........................................ 247
  11.3 Embedding the B-tree Nodes into the Compactor Lists ......... 249
    11.3.1 Encoding information in the nodes ....................... 249
    11.3.2 Packing the nodes in the lists ........................... 250
    11.3.3 Resizing the objects in the compactor lists ............ 251
  11.4 Low-Level Memory Organization of the Compactor Lists ..... 253
    11.4.1 Memory layout .............................................. 254
    11.4.2 Relocation in the $k$-area ................................. 254
    11.4.3 Relocation in the $tk$-area ............................... 256
  11.5 Refining the Solution ............................................. 258
  11.6 Conclusions .......................................................... 262

12 Exponential Implicit Tree .............................................. 263
  12.1 Introduction ........................................................ 263
  12.2 Overview and Roadmap ............................................. 265
  12.3 A Semi-Dynamic Implicit Dictionary ............................. 267
    12.3.1 Actual and virtual chunks ................................ 267
    12.3.2 Routing and searching elements ......................... 268
    12.3.3 Layout in memory .......................................... 268
    12.3.4 Updating the Layout ....................................... 270
    12.3.5 Handling insertions ........................................ 271
      12.3.5.1 Increasing the number of virtual chunks ............ 273
      12.3.5.2 Increasing the number of actual chunks ............ 273
      12.3.5.3 Splitting ................................................ 275
  12.4 Fully-Dynamic Implicit Dictionary Handling Deletions ....... 275
    12.4.1 Extension of the main structure ........................... 275
      12.4.1.1 Modifications for routing and searching .......... 276
      12.4.1.2 Modifications for the insertion .................... 277
    12.4.2 Handling deletions .......................................... 277
      12.4.2.1 Decreasing the number of virtual chunks .......... 278
      12.4.2.2 Decreasing the number of actual chunks ............ 279
      12.4.2.3 Merging ................................................ 279
12.4.2.4 Sharing ............................................. 279
12.5 Amortized Analysis and Conclusions .......................... 280

13 Cache-Obliviousness vs. Implicitness ....................... 285

13.1 Introduction ............................................. 286
  13.1.1 Optimal implicit dictionary in the RAM model .......... 286
  13.1.2 Introducing the issue of cache-obliviousness in implicit dictionaries ............................................. 288
13.2 Overview of the Flat Implicit Tree .......................... 289
13.3 Bottom Layer: Buckets as Implicit Dynamic Forest .......... 293
  13.3.1 Bucket organization .................................... 293
  13.3.2 Memory management of the trees ....................... 295
    13.3.2.1 Node area ........................................... 296
    13.3.2.2 Memory and spare areas ............................ 297
  13.3.3 Internal structure of the trees ....................... 298
    13.3.3.1 Intermediate nodes ................................ 298
    13.3.3.2 Leaves .............................................. 298
    13.3.3.3 Maniples and spare elements ....................... 301
  13.3.4 Supporting the operations on the buckets ............... 302
    13.3.4.1 Searching an element in a bucket ................. 302
    13.3.4.2 Inserting an element into a bucket ................ 302
    13.3.4.3 Deleting an element from a bucket ................ 305
  13.3.5 Analysis of the costs .................................. 305
13.4 Top Layer: The Super-Root ................................ 307
  13.4.1 Actual chunks and virtual chunks ..................... 307
  13.4.2 Density functionalities ............................... 307
  13.4.3 Rebalancing/redistribution of chunks ................ 309
    13.4.3.1 Redistribution within a segment .................. 309
    13.4.3.2 Redistribution of chunks for an internal node .... 310
  13.4.4 Analysis .............................................. 312
13.5 Top Layer: Cache-Obliviousness ............................ 313
  13.5.1 Building the VEB-permutation ........................ 314
  13.5.2 Searching the VEB-permutation ........................ 315
  13.5.3 Maintaining the VEB-permutation ...................... 317
  13.5.4 Analysis .............................................. 318
13.6 Rebuilding and Final Analysis ................................ 319
13.7 Conclusions ............................................. 319

14 Implicit Dictionaries with \(O(1)\) Modifications per Update 321

14.1 Introduction ............................................. 322
  14.1.1 The open question and our contribution ............... 322
  14.1.2 Organization of the chapter .......................... 322
14.2 Techniques and Limitations of Known Implicit Dictionaries .... 323
14.2.1 Storing and carrying information ........................................ 323
14.2.2 Two main problems ......................................................... 323
14.2.3 Structure replication ....................................................... 324
14.2.4 The free sets ................................................................. 325
14.3 Amortized Updates ........................................................... 326
  14.3.1 The aggregating area ..................................................... 327
    14.3.1.1 The invariants for \( \mathcal{A} \) ......................... 327
    14.3.1.2 Searching in \( \mathcal{A} \) .............................. 331
    14.3.1.3 Maintaining the invariants: insertions ................ 331
    14.3.1.4 Maintaining the invariants: deletions .............. 335
  14.3.2 The chunk area ......................................................... 336
14.4 Worst Case Updates ......................................................... 337
  14.4.1 Free sets and lists of pseudo chunks ......................... 338
    14.4.1.1 The invariants for \( \mathcal{A} \) ..................... 338
    14.4.1.2 Searching in \( \mathcal{A} \) .......................... 340
    14.4.1.3 Maintaining the invariants for \( \mathcal{A} \) .......... 341
  14.4.2 Controlling the size of free sets and organizing \( \mathcal{C} \) .... 342

15 In the Worst-Case Scenario ................................................. 347
  15.1 Introduction ............................................................. 347
  15.2 Preliminary Algorithmic Tools ........................................ 350
  15.3 Districts of Chunks ..................................................... 352
    15.3.1 Portions .......................................................... 352
    15.3.2 Districts .......................................................... 352
    15.3.3 How to search an element in the districts ............ 354
    15.3.4 How to update the districts .................................. 356
  15.4 Indirection with Dynamic Buckets ..................................... 358
    15.4.1 The structure of the buckets ................................. 358
    15.4.2 Memory layout .................................................... 360
      15.4.2.1 Filling area .............................................. 360
      15.4.2.2 Leaf area ............................................... 361
      15.4.2.3 Maniple area ............................................ 363
  15.5 Synchronization between Layer \( \mathcal{D} \) and Layer \( \mathcal{B} \) ............ 363
    15.5.1 The problem .................................................... 363
    15.5.2 The solution .................................................... 364
    15.5.3 Priority queues ................................................ 365
  15.6 Conclusion .............................................................. 366

16 Conclusions ................................................................. 369

Bibliography ............................................................................. 372
List of Figures

4.1 An example of Hirschberg’s linear scan with $qpjoknlm$ as searching element. Dark grey cells are mismatches while light grey ones are matches. .................................................. 78

5.1 The heap-like structure with $t = 3$. ................................. 96
5.2 Abstract figure of the structure used to grow the sample set. .... 103
5.3 Two iterations of the partition-based sorting. ..................... 112

7.1 The van Emde Boas layout of a complete binary tree. .............. 155
7.2 Example of the permutation in Lemma 7.1 when $X = 6$ and $Y = 3$. 158
7.3 Sub-sequences and sub-zones. ........................................ 165
7.4 Encoding areas $E$, $E'$ and active/inactive areas pairs $(A_j, I_j)$ for $j = 0,1,2$. .................................................. 167

8.1 A generic instance of $GVSP\{m, p, h\}$. Each of the $n$ vector locations in $\mathcal{V}$ contains one vector. Each of the $O(1)$ auxiliary locations contains one integer of $O(\log n)$ bits. ................................. 178

9.1 The memory layout. .................................................. 208

10.1 A ditch (light gray) encoding all 0s; hence, no two elements in twin positions are swapped. ................................. 228
10.2 The array $\mathcal{A}$ after the preprocessing process we described in Section 10.3.3. ................................. 231

11.1 Low-level memory organization in three areas. .................. 253
11.2 An example of relocation of maniple $x$ being resized as maniple $x'$ (with one more element) in the $k$-area. (a) The head of list $i$ contains part of $x$, and the next-to-head contains the rest of $x$ and part of $y$ (if any). (b) The next-to-head is swapped with the last allocation unit in zone $A$. (c–d) After the addition of a new element in location $c$ to obtain $x'$, maniple $x$ disappears from list $i$ and the next-to-head of list $i$ becomes the new head containing part of $y$ only; the new head of list $i + 1$ becomes $x'$. ................................. 255
12.1 The set $act(v)$ of actual chunks (top row) and the set $vir(v)$ of virtual chunks (bottom row) in a node $v$, along with their encoded pointers (the bullets) to the children of $v$.

12.2 The zones for the memory allocation of nodes.

12.3 Processing a chain Figure 12.1.

12.4 A partition of the actual chunks in segment $z$ into original chunks (left) and patch chunks (right), shown in the top row. Their associated virtual chunks are shown in the bottom row.

13.1 An actual chunk $c$ with its associated bucket $b_c$ of elements, where $c$ is the root of $b_c$. The virtual chunks $c_1, c_2, c_3$ are associated with $c$, and their corresponding buckets are $b_1, b_2, b_3$. Note that the elements in $c, c_1, c_2, c_3$ will be stored in the root area of Figure 13.2, while the elements in $b_c, b_1, b_2, b_3$ will be suitably stored in the node area, manipulate area and spare area shown in Figure 13.2.

13.2 Memory layout of the array of $n$ locations storing the flat implicit tree.

13.3 Compactor zones $s$ and $s + k$.

13.4 A leaf $v$, its associated manipulate $z$ with the spare elements.

13.5 Procedure $\text{Find}$ to search $x$ in a VEB-permutation of $2^h - 1$ elements stored.

15.1 The districts in layer $\mathcal{D}$.

15.2 Compactor zones and sub-zones with broken items highlighted.
Chapter 1

Introduction

1.1 Computing with Permutations

In this work we face the issues of space efficient computations in sorting and searching problems. During the years, the concept of space efficiency has been subject to many different interpretations that will be reviewed later in this work. We are interested in the most restrictive among the interpretations (note that these restrictions give strength to our results, since we are going to prove optimal upper bounds for some computational problems). Let us explain intuitively in what those restrictions consist.

The elements in the input set are drawn from a totally ordered and possibly infinite universe \( \mathcal{U} \). Let \( n \) be the cardinality of the input set. One location of memory contains a single element and the total number of locations that any computation can use is \( n \). At any given moment of its execution, the computation is allowed to retain a constant number of auxiliary numerical value representable with \( O(\log n) \) bits at most (that requirement rules out algorithms defined by non-tail recursion, for instance). Finally, the elements are completely abstract: any two elements can only be compared or exchanged in their locations. Hence, the number of comparisons and the number of exchanges performed during a computation are the natural measures of complexity in this restrictive setting (though we will adopt also other measures of complexity when we extend these requirements to computational models having multi-level memory hierarchies).

From a completely abstract point of view, what we are going to deal with are mere permutations of the input set \( \mathcal{S} \subset \mathcal{U} \). Any algorithm for our setting is a function \( f \) from and to the set of permutations of \( \mathcal{S} \) (the input set is supposed to be arranged in some initial order that may even be important for the problem to be solved, as in the cases where the stability of the algorithm is an objective). The function can be calculated retaining at most a constant number of numerical values of \( O(\log |\mathcal{S}|) \) bits each at the same time. The computation of the function for any sequence in input can be decomposed into a sequence of simple steps of two kinds,
exchanges and comparisons. They are functions from and to the set of permutations of $S$ too but in the second case the function is just the identity. Hence, in order not to be completely useless to the computation, any comparison step needs to have side effects on the set of numerical values that we allow to be maintained by $f$. That last fact leads us to a simple but central observation. Since the set of numerical values simultaneously retainable by $f$ has cardinality $O(1)$, most of the effects of comparison steps seem to have to be spent locally in the computation (that is, the numerical value stored by a comparison is used shortly after in the computation). If that were the only way to maintain information in this permutation-based model of computation, a larger quantity of comparisons would seem to be needed to solve a problem, with respect to a model allowed to retain a linear number of numerical values and auxiliary locations to permute efficiently the input elements (intuitively, since the results of most of the comparisons can be retained for a very short time, these same comparisons are bound to be repeated later in the computation). As we will see, that is not the case since the permutations themselves can be used to retain auxiliary information without disturbing the evolution of the computation and paying only a constant slowdown factor in the complexity bounds.

In the rest of this work we will describe our algorithms allowing the use of a constant number of free locations where the elements can be moved (instead of using only exchanges and no free locations). We do this only for the sake of simplicity, since any algorithm using data moves and a constant number of free locations can be easily transformed into one using only exchanges and no free locations, paying only a constant slowdown factor in the complexity bounds (we will return on this matter in Chapter 2).

1.2 Models

In this section we give a short and intuitive introduction about the models of computation we are going to consider in the subsequent chapters. We will return on this issue in Chapter 2, where we give detailed descriptions of the models we will deal with.

Hypotheses about the universe. Let us introduce an aspect that will be common to all the models we are going to describe, that is the characteristics of the set from which the input elements are drawn. We assume that the elements arrive from a totally ordered and possible infinite universe (hence, do not allow any hypothesis about the input elements to be done, as in the comparison model). In this setting the elements are considered atomic and it is not possible to exploit hypotheses about their internal structure. In particular, hashing, changes of domain and element duplication are not allowed. Essentially they can be only compared and atomically exchanged or moved. Therefore, we assume we are given a total order $(\mathcal{U}, \leq)$ where the universe $\mathcal{U}$ is a generic, possibly infinite, set of elements.
The RAM model. The classical Random Access Machine model popularized in [Aho, Hopcroft, and Ullman, 1976] has a single, unlimited level of memory. Each location of the memory can be accessed in constant time. An algorithm operating in this model is space optimal if it uses only a constant number of auxiliary locations of memory besides the ones containing the input elements drawn from the universe $\mathcal{U}$. In this intuitive definition it is assumed that the algorithm does not use any extra auxiliary memory from more or less hidden sources. Intuitively, the space occupied by algorithm at any time in the computation has to be constant. For example, generic recursive definitions are not admitted because of the hidden recursion stack that, in the general case, does not occupy a constant amount of memory.

Since we are going to face problems in more general settings where the input elements have non-constant length and cannot be compared or moved in constant time, we extend the RAM model appropriately. We will have to deal with input elements belonging to $\mathcal{U}^k$. Basically, the extended RAM model has two kind of locations, the vector locations devoted to contain only input elements (that are indeed vectors) and auxiliary locations used to store numerical values. Each component of a vector can be accessed in constant time and a single vector movement has a cost proportional to the dimension, that is $k$. Standing the constraint that vector locations contain only input elements, an algorithm operating in this extended RAM model is space optimal if it uses a constant number of locations besides the ones used to store the input elements (the constraint about the space occupied by the algorithm we gave for the case of unidimensional elements applies in this case too).

In the RAM model and its variant for multidimensional elements, the complexity of an algorithm will be evaluated with the three natural metrics: the number of comparisons required, the number of moves performed and the number of auxiliary locations used besides the ones strictly necessary for the input elements.

The External-Memory model. Almost any modern computing systems take advantage of multi-level memory hierarchies. The highest level of the hierarchy contains a great amount of memory with slow access time (and with low costs). On the other end of the hierarchy, in the lowest level, a small quantity of a more expensive type of memory with fast access time is used. The processing unit can address the whole highest level, but it can only access the locations in the lowest one. During a generic computation, the data flows up and down in this hierarchy, following the requests of the running algorithm(s). The data transfers between two adjacent levels involve blocks of contiguous locations to allow the amortization of the costs of inter-level communications. In order for that kind of strategy to succeed the computation has to maintain some form of data locality.

Complex memory hierarchies contributed to increase the distance between the practical and theoretical computing. An algorithm with an optimal complexity in the RAM model can be very easily outperformed in a real computation by sub-optimal algorithms exploiting a more effective data locality. In order to overcome
the inability of the RAM model to capture these new aspects of computing, a number of theoretical models have been proposed.

The External-Memory model [Aggarwal and Vitter, 1988] has a simple two-level memory hierarchy: a limited and fast internal memory of size $M$ and an unlimited but slow external memory.

The processing unit can address the external memory but can process only the data that reside in internal memory. An algorithm operating in this model must directly manage the transfers of data between the two levels. The transfers involve the elements contained in blocks of $B$ contiguous locations each.

The natural complexity measure for an algorithm operating in the External-Memory model is the number of block transfers it incurs, commonly called the the I/O complexity. Unlike the RAM model, we are not going to consider problems involving elements drawn from multidimensional domains.

An algorithm operating in this model is space optimal if it uses $n$ contiguous locations in external memory for the input elements and $O(1)$ extra auxiliary locations to store numerical values. Since the only constraint about the number of locations employed for the input set concerns the external memory, during a computation we can have the input set spread among a total of $n + M$ locations, counting also the $M$ location of the internal memory.

It could seem strange that while defining a space optimal algorithm we allow more than $n + O(1)$ space for the input set ($n$ input locations in external memory and $M$ in internal memory). A more restrictive definition is certainly possible and we can require that a total of $n$ input locations are used throughout the computation ($n - M$ of which in external memory). Even though the solutions for all problems we can think of in the less restrictive model seem to be easily adaptable to the stricter one, the stricter model seems to be pointless, especially when the memory hierarchy has more than two levels. We will talk about this after the introduction of the Cache-Oblivious model.

The Cache-Oblivious model. Introduced in [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Prokop, 1999], the Cache-Oblivious model can be seen has combination of RAM model and External-Memory model. That model is composed of two main parts: the Ideal-Cache model and the cache-oblivious algorithms.

The Ideal-Cache model shares some aspects with the External-Memory model. It consists of two levels of memory. The cache level, containing $M$ locations and partitioned into blocks (or cache lines) of $B$ contiguous locations each, and the main memory level which is unbounded. As in the External-Memory model, the processing unit can address the locations of the main memory but it can process only the data residing in cache. Moreover, the cache is fully associative, meaning that each block coming from main memory can be stored anywhere in the cache.

Unlike the External-Memory model, any algorithm operating in the Cache-Oblivious model has no control over the flow of blocks between the levels of memory.
Whenever there is a cache miss, that is the block $b$ containing the data necessary to the computation is not in cache, an optimal off-line replacing strategy is used to substitute a residing block with $b$. Hence, the model is supposed to know in advance the future memory requests of the computation. That aspect, together with the full associativity assumption, can bring some doubts about the strength of the connection between the theoretical model and the practical computing systems. However, in the seminal paper [Frig, Leiserson, Prokop, and Ramachandran, 1999] it has been shown that algorithms satisfying a reasonable regularity condition on the number of cache misses in the Ideal-Cache model, have a small constant-factor overhead for that measure when used in a weaker cache model with the realistic LRU replacement strategy. In the same paper, an expected-time, favorable argumentation is provided for the full-associativity assumption.

An algorithm is cache-oblivious if it has no memory-hierarchy-specific parameterization. In the particular case of the Ideal-Cache model, that means that the algorithm cannot be defined in terms of the hierarchy parameters $B$ and $M$. Hence, an algorithm for the Cache-Oblivious model looks like an algorithm for the RAM model. Unlike the External-Memory model, in the Cache-Oblivious model there are two measures for the complexity of an algorithm. The work complexity $W$ is the standard complexity in the RAM model (evaluating the number of comparisons, arithmetic operations, data moves...). The cache complexity $Q$ is the total number of cache misses incurred during the computation.

Since the cache complexity analysis holds for any value of $B$ and $M$, it holds for any level of a more general multi-level memory hierarchy. The definition of space optimality in this model follows the ones we saw for the External-Memory model: an algorithm is space optimal if it uses a constant number of main memory locations besides the contiguous ones necessary to contain the input elements. As we observed in the case of the External-Memory model, we could adopt a definition for space optimality imposing a limited use of the locations of the other (lower) levels of the memory hierarchy. However the stricter definition seems to be pointless because of the very foundational hypothesis of all the hierarchical memory models: the only level that can contain the whole input set is the highest one (also the one with the slowest memory, in practice) and the sizes of all the other levels should be negligible if compared with the size of the highest one. If that is not true then there is no need to have a hierarchical memory model in the first place. Moreover, in the particular case of the Cache-Oblivious model, only the highest level of the hierarchy is managed directly by the algorithm and a waste of the locations of the other levels seems to be unavoidable.

### 1.3 Problems

In this section we give a brief highlight of the computational problems that will be faced in the subsequent chapters. The history, the previous work and detailed
description of any single problem will be given in Chapter 3 and in the subsequent
chapters relative to the particular problem itself.

**Sorting in a linear number of moves.** As we said before, the efficiency of an
algorithm in the RAM model can be measured with the three natural metrics of
comparison complexity, move complexity and space complexity. It has been gener-
ally conjectured, for many years, that a sorting algorithm matching simultaneously
the asymptotic lower bounds on all above computational resources does not exist.
This is the first problem we will face in this work. We want to give an answer to the
following question. Does there exist a sorting algorithm that is comparison, move
and space optimal in the RAM model? That is, does there exist an algorithm
that can sort \( n \) input elements performing \( O(n \log n) \) comparisons, \( O(n) \) moves and
using a constant number of auxiliary locations?

**Space optimal sorting in the Cache-Oblivious model.** When dealing with
sorting in the Cache-Oblivious model, it is usually assumed the presence of a tall
cache (i.e. \( M = \Omega(B^2) \)). Under this assumption, the cache complexity of any sorting
algorithm in the Cache-Oblivious model is \( \Omega \left( \frac{n}{B} \log_M n \right) \). The work complexity can
be directly derived from the hypotheses on the universe \( \mathcal{U} \) and is \( \Omega(n \log n) \) as in
the RAM model.

The only two work and cache optimal sorting algorithms known so far for the
Cache-Oblivious model are the Funnel-sort (and its variant, the Lazy Funnel-sort
in [Brodal and Fagerberg, 2002a]) and the exponential Distribution sort, both intro-
duced in the seminal paper [Frigo, Leiserson, Prokop, and Ramachandran, 1999].

Unfortunately, these two algorithms are nowhere near to achieve space optimality
along with work and cache optimality. Furthermore, space optimality seems to be
challenging in the Cache-Oblivious model because of the apparently antithetical
requirements needed to obtain the maximum space saving and the data locality.
For these reasons, the techniques originally developed for this problem in the RAM
model and External-Memory model appear to be of difficult application in the cache-
oblivious scenario. For all these reasons it is an important objective to settle down
the question of whether there exists or not a sorting algorithm work, cache and space
optimal in the Cache-Oblivious model.

**Space optimal sorting in a multidimensional domain.** The existing sorting
algorithms in a multidimensional domain seem to need a large quantity of auxiliary
information in order to achieve optimal complexity. To be fully convinced of this
observation it is sufficient to think about the longest common prefix information
(or equivalent) used in every efficient sorting algorithm both in RAM model and
External-Memory model, like the ones in [Bentley and Sedgewick, 1997] and [Arg\-n, Ferragina, Grossi, and Vitter, 1997] just to cite one well-known example for any of
these models. If space optimality is an objective, that large quantity of apparently
unavoidable auxiliary information becomes an obvious problem. Another problem strictly related to the space optimality is given by the fact that in absence of a linear number of auxiliary input locations even the task of permuting the input elements seems to require more work. That becomes a problem when the cost of moving an input element is not a constant, as in the case of vectors or records for instance. All those problems contribute to make the problem of space optimal sorting in a multidimensional domain very challenging, as we will see.

**Generalized merging problem.** In the balanced merging problem, we are given \( s \leq n \) sorted sequences of \( n/s \) elements each drawn from \( \mathcal{U} \) and they have to be fused into a single sorted sequence. In the unbalanced merging problem the total number of input elements is still \( n \) but the \( s \) sorted subsequences to be fused can differ in their lengths. As we will see in Chapters 3 and 9 the research focusing on the existence of a comparison, move and space optimal solution for the merging problem has been fervent since the sixties, bringing results for the special case where \( s = O(1) \) (binary merging). In spite of all the research efforts, at the best of our knowledge, so far, even the existence of a comparison, move and space optimal solution for the merging problem covering any range of values better than \( s = O(1) \) is unknown (though some ideas suitable for the case \( s = O(\text{polylog} n) \) are in [Katajainen and Pasanen, 1999], as we will see in Chapter 9). For those reasons the problem is certainly difficult and given the parametric nature of it, some natural questions arise. How far can we push the parameter \( s \)? Is there a comparison, move and space optimal solution for the merging problem that is independent from any particular value of \( s \)? How heavily marked is the boundary between merging and sorting problems?

**Searching in a multidimensional domain.** Let us consider the following fundamental problem. We are given a static set of \( n \) \( k \)-dimensional vectors. Assuming to be in the RAM model, the objective is to search for another vector among them using only \( O(1) \) auxiliary information. The obvious lower bound for this problem is \( \Omega(\log n + k) \) component comparisons and arithmetic operations. When the dimension of the vectors is a constant, we are in luck since the permutation we are searching for is the sorted one and the universally known binary search will do a fine job in this case. If the dimension \( k \) of the vector cannot be considered a constant then the solution is not so obvious. As we will see, the research focusing on this problem has started in the seventies. Not many results have been produced for this fundamental problem, as a witness of its intrinsic complexity we point out the last very important result in the field [Andersson, Hagerup, Hästad, and Petersson, 1994, Andersson, Hästad, and Petersson, 1995b, Andersson, Hagerup, Hästad, and Petersson, 2001]. In this series of works the authors prove that the complexity of the problem is \( \omega(\log n + k) \) when we restrict to the sole sorted permutation. In other words the natural lower bound \( \Omega(\log n + k) \), that finely works for the case of sorted sequences of unidimensional elements, is not tight in case the elements in the set
arranged in sorted order have a non-constant dimension. This fact leaves us with the following open question: given a set of \( n \) \( k \)-dimensional elements, does there exist a permutation (necessarily different from the sorted one) of the set allowing to search it in \( O(\log n + k) \) time?

The implicit dictionary problem. A data structure containing \( n \) elements drawn from \( U \) is *implicit* if it occupies only \( n + O(1) \) contiguous locations and if each operation defined over the structure is space optimal. Hence, at any given moment of its life, the structure appears as a simple permutation of the elements plus a constant number of numerical values (the number of elements, for instance).

While the precise definition of implicit data structure was introduced in the late seventies [Munro and Suwanda, 1979], the first actual “exemplar” of this fascinating kind of structures dates back to the sixties. We are referring to the well-known Williams’ heap [Williams, 1964], the very elegant priority queue supporting extractions of the maximum element and insertions in \( O(\log n) \) time in the RAM model (interestingly, the first comparison and space optimal sorting algorithm, the Heapsort, is based on the first implicit data structure, the heap).

As we will see, with the seminal paper [Munro and Suwanda, 1979] the research on implicit data structures turned immediately toward the dictionary problem. Because of its general importance, it is convenient to formulate the implicit dictionary problem in a form parametric in the model of computation:

*For a given model of computation \( \mathcal{M} \) and a universe \( U \), is there a space optimal update algorithm \( U \) to maintain a dynamic set \( S \subseteq U \) such that the following three requirements are met. (i) Only \( |S| \) locations plus \( O(1) \) values of auxiliary information of \( O(\log |S|) \) bits each are used. (ii) \( S \) can be searched optimally (with respect to model \( \mathcal{M} \)). (iii) \( U \) is space optimal and cost optimal (with respect to model \( \mathcal{M} \)).*

In light of the fact that a hypothetical optimal implicit dictionary can be seen as a more powerful heap, in the particular case of the RAM model the implicit dictionary problem can be perfectly synthesized in a single question that has been around since the introduction of Williams’ heap in the sixties:

*Is there a searchable heap?*

### 1.4 Contributions and Organization of this Work

This work is divided into three main parts.

In the first part we will describe the models of computation we will consider, the open problems we will face and known tools we will use in our solutions.
Chapter 2. We give a precise description of the models of computation we will consider in our research. All the models will be slightly modified in order to incorporate directly the space optimality constraint that is one of the main themes of this work.

Chapter 3. We will give a detailed presentation of the problems we are going to face together with their history, that in most of the cases is quite long, and the previous related work.

Chapter 4. We will describe some known tools that we will use throughout this work in our solutions for the stated problems. Most of these techniques will be used in a large variety of ways. Because of this, for any technique we give there the main ideas common to all the variants we will use in the rest of this work. We will give the peculiar details of any particular variant when we make use of it.

The second part of the work will be devoted to the description of our results to the sorting-related problems that we have briefly described previously in this chapter. Every chapter in this part is based on a published paper with the exception of Chapter 9 that is based on a papers currently submitted for publication.

Chapter 5. The presentation in this chapter is based on the paper [Franceschini and Geffert, 2003] (FOCS 2003) and [Franceschini and Geffert, 2005] (Journal of the ACM) that solved the long-standing open problem of whether there exists a sorting algorithm that matches the asymptotic lower bounds on all computational resources simultaneously. Therefore, we shall exhibit the first comparison, moves and space optimal sorting algorithm. Moreover we will be able to give a precise analysis down to the constant factors. Our algorithm operates in-place, with at most $2n \log n + o(n \log n)$ element comparisons and $(13+\varepsilon)n$ element moves in the worst case, for each $n \geq 1$. Here $\varepsilon > 0$ denotes an arbitrarily small, but fixed, real constant. The number of auxiliary arithmetic operations with indices is bounded by $O(n \log n)$. We can slightly reduce the number of moves, to $(12+\varepsilon)n$, in a modified version that uses $6n \log n + o(n \log n)$ comparisons.

Chapter 6. The presentation in this chapter is based on the papers [Franceschini, 2005a] (STACS 2005) and [Franceschini, 2005c]. We settle the remaining open question after the introduction of the algorithm in [Franceschini and Geffert, 2003] (Chapter 5). The algorithm in this paper is unstable meaning, when it is applied to a multiset, the initial order of the occurrences of the same element is not maintained after the termination. Stability is a very difficult property to maintain when space optimality is an objective. As we will see the algorithm in this chapter is much different and much more complicated than the unstable one in [Franceschini and Geffert, 2003].
Chapter 7. The presentation in this chapter is based on the paper [Franceschini, 2004] (SODA 2004). In this chapter we shall give a twofold contribution. We present the first sorting algorithm that is work optimal, cache optimal and space optimal in the Cache-Oblivious model, that is having a work complexity $O(n \log n)$, a cache complexity $O\left(\frac{n}{B} \log M n\right)$ and using $O(1)$ auxiliary locations of memory. Furthermore, we introduce a new approach to the sorting problem previously unknown both in the Cache-Oblivious model and RAM model.

Chapter 8. The presentation in this chapter is based on the paper [Franceschini and Grossi, 2005a] (ICALP 2005). In this chapter, we have to deal with input elements drawn from a multidimensional domain. We show that the large quantity of auxiliary information (e.g. longest common prefixes information) usually needed in this setting and the non-constant cost for moving elements do not prevent from having a space, comparison and move optimal sorting algorithm. We present the first space optimal algorithm for lexicographically sorting $n$ elements in $O(nk + n \log n)$ time.

Chapter 9. The presentation in this chapter is based on the paper [Franceschini, 2005b] currently submitted for publication. We present the first comparison, move and space optimal algorithm for the generalized merging problem that works for any value of parameter $s$. This closes the problem and is a huge step forward since, as we will see in detail later in this work, the best comparison, move and space optimal solutions so far known for this problem worked only in case $s = O(1)$ (though some ideas suitable for the case $s = O(\text{polylog } n)$ are in [Katajainen and Pasanen, 1999], as we will see in Chapter 9). This result has also some implications for the field of adaptive sorting.

Finally, in the third and last part of this work we will present the searching-related results. Every chapter in this part is based on a published paper with the exception of Chapter 14 which is based on a paper currently submitted for publication.

Chapter 10. The presentation in this chapter is based on the paper [Franceschini and Grossi, 2004b] (FOCS 2004). We close the problem of searching (more precisely, the nearest-neighbour query problem) a (multi)set of elements drawn from a multidimensional domain without using auxiliary information. We close it perhaps in the most unexpected way since we show that the sorted permutation of the elements is not the one allowing an optimal search algorithm. More precisely, we show that, for any set $\mathcal{U}$ and for any set $S \subseteq \mathcal{U}^k$ of $n$ elements there exists a permutation $\mathcal{P}(S)$ different from the lexicographically sorted one that can be searched using $O(k + \log n)$ component comparisons and $O(1)$ values of auxiliary information. That allows us to conclude that the complexity of space optimal searching is $\Theta(k + \log n)$. 
Chapter 11. The presentation in Chapter 11 is based on the papers [Franceschini, Grossi, Munro, and Pagli, 2002] (FOCS 2002) and [Franceschini, Grossi, Munro, and Pagli, 2004] (Journal of Computer and System Sciences 68). We face the implicit dictionary problem in the External-Memory model and we prove that, under the condition $B = \Omega(\log n)$ there exists an I/O optimal implicit dictionary, the Implicit B-Tree. More precisely, in the External-Memory model, under the assumption $B = \Omega(\log n)$ the Implicit B-Tree can be searched and updated with $O(\log_B n)$ block transfers in the worst case and can report $r$ consecutive elements with $O(\log_B n + r/B)$ block transfers also in the worst case (both bounds are optimal). Furthermore, in the RAM model, the Implicit B-Tree can be searched and updated in $O\left(\frac{\log^2 n}{\log \log n}\right)$ time. That disproved the long-standing conjecture in [Munro, 1986].

Chapter 12. The presentation in this chapter is based on the paper [Franceschini and Grossi, 2003a] (SODA 2003). We focus our attention on the RAM model, in the effort of improving the bounds for implicit dictionaries. We introduce the Exponential Implicit Tree that lower the search time bound to $O(\log n \log \log n)$ in the worst case and the update time to $O(\log n \log \log n)$ in amortized sense. The Exponential Implicit Tree is the first, fully dynamic (i.e. supporting both insertions and deletions) implicit dictionary dealing with the amortized analysis. That is a difficult task since, as we will see, the particular setting of the implicit dictionary rules out some well-known data structuring techniques specifically suited for amortized analysis.

Chapter 13. The presentation in this chapter is based on the paper [Franceschini and Grossi, 2003b] (ICALP 2003) and [Franceschini and Grossi, 2005b] (Theory of Computing Systems). We move our aims to the Cache-Oblivious model and we introduce the Flat Implicit Tree, the first implicit dictionary having optimal search in the worst case and optimal update operations in amortized sense. More precisely, the Flat Implicit Tree can be searched with a worst case work complexity $O(\log n)$ and a worst case cache complexity $O(\log_B n)$ and can be updated with the same optimal bounds but in amortized sense. Since the Flat Implicit Tree has the same work complexity and cache complexity if used, respectively, in the RAM model and External-Memory model, we obtain also the first optimal implicit dictionary for the former model and the first optimal dictionary not depending on any constraint on $B$ for the latter one.

Chapter 14. The presentation in this chapter is based on the paper [Franceschini and Munro, 2006] (SODA 2006). The most interesting lower bound on implicit dictionaries is the 1988 result [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988]. The authors prove a tradeoff between the search time and the update time in implicit dictionaries: if the update cost, number of comparisons and exchanges, for an implicit dictionary is $O(1)$ then the search cost must be $\Omega(n^\epsilon)$, for some constant $\epsilon > 0$. In their conclusion, the authors
left open the question of whether a tradeoff of that kind would hold if only the modifications (i.e. data moves) performed during any update were considered. They conjectured that this would be in fact the case and that any implicit dictionary performing only $O(1)$ exchanges per update should very quickly get disorganized, thus ending up requiring $\Omega(n^c)$ comparisons per search. In chapter 14, we answer to this long-standing open question by disproving the conjecture.

Chapter 15. The presentation in this chapter is based on the paper [Franceschini and Grossi, 2003c] (WADS 2003). We will be finally able to close the implicit dictionary problem introducing a structure with optimal update bounds in the worst case. As for the Flat Implicit Tree, we will solve the problem in the Cache-Oblivious model thereby achieving the result also for the RAM and External-Memory models.
Part I
Models, Problems & Tools
Chapter 2

Computational Models

Abstract

In this chapter, we describe the computational models we are going to deal with in the rest of this work.


- The widely studied External-Memory model (see [Aggarwal and Vitter, 1988]) has a simple two-level memory hierarchy: a limited and fast internal memory and an unlimited slow external memory. The internal and external memory are considered associated with, respectively, the main memory and the disk of actual computers.

- The Cache-Oblivious model [Frigg, Leiserson, Prokop, and Ramachandran, 1999] can be seen as a “successor” of the RAM model, a successor that incorporates a lot of the new architectural aspects which characterize the real world computing systems, like multiple levels of memory, hidden automatic strategies to exchange data between two consecutive levels etc.

On top of these models we pose some additional constraints. This is in order to formalize and capture, for any of these models, the essence of the intuitive idea of operating over a mere permutation of atomic elements being allowed to use nothing but a constant-sized set of additional space resources.

2.1 A Lot of Possibilities

Space concerns. The classical Random Access Machine [Aho, Hopcroft, and Ullman, 1976] has a single level of memory composed by an unlimited number of locations. Each location can be accessed and modified in constant time. During the
years, many algorithms dealing with space efficiency in this model have been proposed. However, the concept of space efficiency has been subject to many different interpretations.

Many authors considered in-place algorithms, that is using a constant number of additional locations besides the ones strictly necessary to store the input elements (just two examples, perhaps the most famous and old example [Williams, 1964] and a newer one we know well about [Franceschini, 2005a]). Some other authors considered the more weak definition in in-situ algorithms where it is allowed the use of a number of additional locations that is of the order of the logarithm of the size of the input set (see [Laflin and Brebner, 1970], [Munro, Raman, and Salowe, 1990], ...).

Also the internal resources (local variables, pointers, recursion stack...) "embedded" in the algorithm itself have to be explicitly considered. Even without the (rather ridiculous) attempt to swept the problem under the rug allowing the use of internal resources at will, apparently marginal aspect of the definition of the model can have an important impact on the computational power. For example, some authors allow recursive definitions of the algorithms, for example [Munro and Raman, 1991].

Let us consider a generic simple binary divide et impera approach to the solution of a given problem of size \( n \). If the divide step does not split the problem into two sub-problems with immediately known or easily calculable sizes then a constant number of values of \( O(\log n) \) bits each might have to be stored for each recursive call (the merge step might need that values later). Therefore a total of \( \omega(\log n) \) bits of auxiliary information is used and \( \omega(1) \) additional locations of memory have to be available. If the algorithm can be defined recursively that fact is completely hidden, against the intended spirit of the in-place setting. This weaker definition of space efficiency can be seen as standing in the middle between the stronger in-place definition and the in-situ definition: a number of additional locations that is of the order of the logarithm of the size of the input set can be used but they have to be used only to deal with the possible recursive definition of the algorithm.

**Universe concerns.** Another aspect that contribute to variate the models for space efficiency (and also the models where space efficiency is not an objective) concerns the characteristics of the set from which the input elements are drawn. Many authors assume that the input elements arrive from a totally ordered and possible infinite universe (or, simply, they do not allow any hypothesis about the input elements to be done, as in the comparison model). In this setting the elements are considered atomic and it is not possible to exploit hypotheses about their internal structure. Essentially they can be only compared and atomically moved.

In particular settings like multidimensional universes (for example in the case of vectors or records...), the possibility of moving only portions of elements may be considered. This is an immediate advantages when the number of moves is a
concern, since in the case of multidimensional universes the cost of a single move cannot be considered constant but has to be bounded with the dimension of the universe. For example, in the case of vectors with $k$ scalar components each, two vectors that have a large prefix in common can be exchanged in $O(k - lcp)$ time instead of $O(k)$ time (where $lcp$ denote the length of the longest common prefix between the two vectors).

Finally, many papers deal with cases in which hypotheses over the input universe can be exploited, for example when the universe is the set of integer numbers (e.g. integer sorting algorithms [Han, 2001] even if space optimality is difficult to obtain), or more generally when the internal structure of the input elements is known.

This kind of hypothesis allows, for example, the use of duplication of elements and therefore the possibility to operate with the subset of distinct elements instead of the whole input set. Another typical advantage brought by hypotheses over the input universe is the possibility of changing the domain, that is finding a (maybe space efficient) function to reduce the input set to another set easier to be treated.

As we will see in the next section, we are interested in the most restrictive of the settings:

- the input elements are atomic,
- they can only be moved or compared and
- only a constant quantity of any auxiliary resources can be used, ruling out the possibility of recursion-hidden resources.

## 2.2 The In-Place RAM Model for Vectors

We want to study the computational power of common models when very basic operations are allowed and permutations of the input set are the only object and result of any step of a computation. Since we are going to study sorting and searching problems both in unidimensional and multidimensional universe settings, we give a generic definition involving $k$-dimensional vectors, where $k \geq 1$.

We are given a total order $(\mathcal{U}, \leq)$ where the universe $\mathcal{U}$ is a generic, possibly infinite, set of elements. We are given an input set $\mathcal{V} \subseteq \mathcal{U}^k$ ($k \geq 1$) of $n$ vectors stored into $n$ vectorial locations, one vector of $\mathcal{V}$ per location. Two kinds of operations on the vector locations are permitted:

- Atomic exchange any two vectors in $O(k)$ time.
- Access the $i$th scalar component of any two vectors for comparison purposes in $O(1)$ time.

In the literature, when the choice is to deal with an input set with assumptions like these, it is said that the input elements are drawn from an unbounded universe.
We are also given a constant number of auxiliary locations, each location storing one integer of $O(\log n)$ bits. We employ the standard repertoire of instructions on the auxiliary locations, with $O(1)$ time per operation. A feasible computation accesses vector locations and auxiliary locations according to the following rules:

- It is allowed to perform pairwise comparisons and atomic exchanges in the vectorial locations, and no other operations are permitted.
- It is allowed to use the full computational power of the RAM model on the auxiliary locations so as to decide which comparisons and atomic exchanges should be performed on the vectorial locations.

Note that exchanging two vectors uses just two auxiliary locations to specify the two vector locations involved. Finally, all the algorithms we will propose for this model will not suffer of the hidden auxiliary information problem we discussed in the previous section. Even when, for the sake of description, we give an algorithm with a recursive definition, we will show either how the recursion stack uses only $O(\log n)$ bits (fitting in a constant number of auxiliary locations) or how it can be eliminated entirely.

It is worth to point out that an alternative definition with a generic move operation instead of the exchange operation is possible. Following this other way we would have to admit also a constant number of free vectorial locations to allow the inversion of the relative order of any two vectors. The two approaches are completely equivalent but we believe that having to deal with a set of $n$ elements that can only be exchanged better captures the essence of space optimal algorithms.

On the other hand, the definition with the move operation and with a constant number of free vectorial locations is preferable for the sake of description. Therefore, in the following chapters, we will be a little loose and we will talk indifferently of move or exchange operations. It is easy to see that any computation described using moves and a constant number of free vectorial locations can be translated into one using only the $n$ vectorial locations and exchange operations (the price to pay is an increased constant factor for the number of moves). We can just choose a constant number of distinct vectors and make them act as placeholders simulating the constant number of free vectorial locations needed by the move-based computation (of course the extreme case in which not even $\omega(1)$ distinct elements are present is obvious, whatever the sorting or searching-related problem we are facing is). The current location of any placeholder can be tracked using one of the $O(1)$ auxiliary integer locations that we allow in both flavors of the model. As a matter of fact this is a trivial application of the internal buffering technique (see Chapter 4) that is a fundamental instrument in the development of the vast majority of space optimal algorithms.

In the third part of this work, we will face some dynamic searching problems. Also in this case, an exchange-based model is better in pointing out the fact that,
amazingly, the dynamic searching problem is solved by a simple permutation of the
dynamic set of elements and a function that takes the current permutation and gives
back a new one (possibly larger or shorter by one element) in the target bounds. The
model can be dynamized simply by allowing the adding of the new \( (n+1) \)st vectorial
location or the removing of the \( n \)th one. The growth of the size (number of bits) of
any auxiliary location follows the common transdichotomous model [Fredman and
Willard, 1993]. However, once again for the sake of description, we are going to use
the move-based variant also in the third part. We are allowed to do so by the same
reasons we have just given for the static use of the model.

2.3 Hierarchical Memory Models

One of the main characteristics of “real world” computing systems is the use of
multi-level memory hierarchies. Typically, the highest level of the hierarchy contains
a great amount of memory with slow access time (and with low costs). On the
contrary, for the lowest level a small quantity of more expensive type of memory
with fast access time is used. Usually, the processing unit can address the whole
highest level, but it can only access the locations in the lowest one. During a
generic computation, the data flows up and down in this hierarchy, following the
requests of the running algorithm(s). Because of hardware limitations and in order
to amortize the cost of the inter-level flow of information, the data transfers between
two adjacent levels always involve blocks of contiguous locations. Clearly, that kind
of strategy is successful only if the computation maintains some form of data locality.

A typical example of memory hierarchy of a modern computer is composed by
registers, look-ahead buffers, level one and level two cache (even level three in high-
end computers), main memory and disk. Each level has its own block size, making
hard the optimal tuning of algorithms. Moreover the vast majority of these levels
have a strategy for the replacement of blocks (used when the level is full) that is
unknown to the algorithms or, at least, that cannot be directly controlled.

Complex memory hierarchies contributed to increase the distance between the
practical and theoretical computing. An algorithm that has an optimal complexity
in the classic RAM model can be potentially outperformed in a real computation
by sub-optimal algorithms which exploit a better data locality. The RAM model
is unable to capture the aspects of memory access patterns and so a number of
theoretical models have been proposed in order to cope with these new issues of
modern computing.

In [Aggarwal, Alpern, Chandra, and Snir, 1987a] a first step toward this direction
was done with the introduction of the Hierarchical Memory model. Like in RAM,
this model has an unlimited number of locations. It aims at modeling a multi-level
hierarchy with the use of a function \( f(i) \) that gives the access cost for the \( i \)th location.
The authors mainly study the case in which \( f(i) = \lceil \log i \rceil \). For the sorting problem
in particular, they prove an upper bound \( O(N \log N \log \log N) \) and the matching
lower bound. Other cost functions are examined and in [Aggarwal, Chandra, and Snir, 1987b] the model is extended with block transfers. But it is with the widely studied External-Memory model, popularized by Aggarwal and Vitter [Aggarwal and Vitter, 1988], that the main stream of research in this new field begun.

2.4 The External-Memory Model

In the External-Memory model [Aggarwal and Vitter, 1988] a simple two-level memory hierarchy is proposed:

- A limited and fast internal memory of size $M$.
- An unlimited slow external memory.

The processing unit can address the external memory but can process only the data that reside in internal memory. An algorithm operating in this model must directly manage the transfers of data between the two levels. Those transfers involve the elements contained in blocks of $B$ contiguous locations each. Therefore, a natural complexity measure for an algorithm operating in the External-Memory model is the number of block transfers it incurs. This measure is called the I/O complexity and can be justified if the internal and external memory are associated with, respectively, the main memory and with the disk of a computer. The role of disk accesses is dominant in the total running time of any "reasonable" algorithm.

External-Memory model and space-optimality. Let us introduce the simple variation of the External-Memory model that we use to capture completely the setting in which only very basic operations, block transfers in this case, are allowed and permutations of the input set are the only object and result of any step of a computation.

Again, we have a total order $(\mathcal{U}, \leq)$ where the universe $\mathcal{U}$ is a generic, possibly infinite, set of elements. Since we are not going to study any problem involving vectors in hierarchical memory models, we are given an input set $\mathcal{I} \subseteq \mathcal{U}$ of $n$ elements stored into $n$ contiguous input locations in the external memory, one element of $\mathcal{I}$ per location. The internal memory is also composed by $M$ input locations. We are also given a constant number of auxiliary locations, each location storing one integer of $O(\log n)$ bits. Let us suppose for simplicity that the auxiliary locations are in internal memory.

Because of the basic operation of block transfer, at any given moment during a computation the input set $\mathcal{I}$ is stored into the $M+n$ input locations in internal and external memory. As in the case of RAM, the complete generality of the universe $\mathcal{U}$ does not allow any duplication or bit manipulation of any kind. The input elements can be only moved, as an effect of block transfers between internal and external memory, or compared (as long as they reside in internal memory). For the case of
a dynamic input set, the same simple considerations we did for the in-place RAM model for vectors hold.

It could seem strange that in a setting of space-optimality we allow more than \( n + O(1) \) space for the input set (as we use \( n \) input locations in external memory and \( M \) in internal memory). It is certainly possible to give a more restrictive definition and to require that a total of \( n \) input locations are used throughout the computation (\( n - M \) of which in external memory). Moreover, the solutions for all problems we can think of in the less restrictive model seem to be easily adaptable to the stricter one using a simple instance of the buffering technique (a little bit like we did in Section 2.2, when we proved the equivalence between move-based and exchange-base models).

However the stricter model seems to be pointless because of the very foundational hypothesis of all the hierarchical memory models: the only level that can contain the whole input set \( \mathcal{I} \) is the highest one (also the one with the slowest memory, in practice) and the sizes of all the other levels are negligible if compared with the size of the highest level. If that is not true then there is no need to have a hierarchical memory model in the first place. Therefore, in the general case of a multilevel hierarchical memory model, the space-optimality involves only the highest (and slowest and largest) level of the hierarchy. If \( M_1, M_2, \ldots, M_{l-1} \) are the \( l - 1 \) lowest levels of a \( l \)-level memory model then the total number of input locations used by a space-optimal algorithm is \( n + \sum_{i=1}^{l-1} M_i \).

## 2.5 The Cache-Oblivious Model

A recent, though widely studied, model of computation is the Cache-Oblivious model, introduced by Frigo et al. [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Prokop, 1999]. That model is composed of two main parts:

- the ideal-cache model and
- the cache-oblivious algorithms (and data structures).

### The ideal-cache model.

The ideal-cache model consists of two levels of memory, the cache level that contains \( M \) locations and is partitioned into blocks (or cache lines) of \( B \) contiguous locations each, and the main memory level that can be arbitrarily large. The processing unit can address the locations of the main memory but it can process only the data residing in cache.

Unlike the External-Memory model, when the block \( b \) that contains the data necessary to the computation is not in cache (that event is called a cache miss), an optimal off-line replacing strategy (i.e., the model is supposed to know in advance the future memory requests of the computation) is used to substitute a residing block with \( b \). Therefore, an algorithm operating in the ideal-cache model cannot directly manage the transfers of blocks between levels. Furthermore, the cache is fully
associative, that is each block coming from main memory can be stored anywhere in the cache.

**Cache-oblivious algorithms.** An algorithm is *cache-oblivious* if it has no memory-hierarchy-specific parameterization. In particular, if that algorithm operates in the ideal-cache model, it cannot be defined in terms of the hierarchy parameters $B$ and $M$. Hence, an algorithm for the Cache-Oblivious model looks like an algorithm for the RAM model. Moreover, the definition of in-placeness for the RAM model naturally extends to the cache-oblivious one: an algorithm operates in-place if it uses $O(1)$ extra locations of main memory besides the contiguous ones containing the input elements. Unlike most of the results for the External-Memory model, for the algorithms and data structures in Cache-Oblivious model there are always two measures for the complexity of an algorithm:

- The *work complexity* $W$, that is the standard complexity in the RAM model (evaluating the number of comparisons, arithmetic operations, data moves...).

- The *cache complexity* $Q$, that is the total number of cache misses incurred during the computation.

**The Cache-Oblivious model in theory and practice.** Besides the great theoretical interests in finding algorithms that, essentially, are able to conduct an optimal computation without knowing those parameters the concept of optimality is defined with, someone can have practical concerns. In particular, the ideal-cache model might seem unrealistic because of the optimal off-line replacement strategy.

However, Frigo et al. [Frigo, Leiserson, Prokop, and Ramachandran, 1999] have shown that algorithms satisfying a reasonable regularity condition on the number of cache misses in the ideal-cache model, have a small constant-factor overhead for that measure when used in a weaker cache model with the realistic LRU replacement strategy. For the full-associativity assumption an expected-time, favorable argumentation is provided in [Frigo, Leiserson, Prokop, and Ramachandran, 1999]. Furthermore, since the cache complexity analysis holds for any value of $B$ and $M$, it holds for any level of a more general multi-level memory hierarchy.

Therefore, the Cache-Oblivious model can be seen as a “successor” of the RAM model, a successor that incorporates a lot of the new architectural aspects which characterize the real world computing systems. Despite of its recent introduction, the Cache-Oblivious model is widely studied and there are many types of (mainly theoretical) results:

• Priority queues [Arge, Bender, Demaine, Holland-Minkley, and Munro, 2002, Brodal and Fagerberg, 2002b].

• Graph algorithms [Arge, Bender, Demaine, Holland-Minkley, and Munro, 2002].

• Optimal dictionaries, with amortized and worst case bounds, partially-persistent, space consuming and implicit [Bender, Demaine, and Farach-Colton, 2000, Bender, Duan, Iacono, and Wu, 2002d, Brodal, Fagerberg, and Jacob, 2002, Bender, Cole, and Raman, 2002b, Franceschini and Grossi, 2003b,c].

• Static tree layout [Bender, Demaine, and Farach-Colton, 2002c, Alstrup, Bender, Demaine, Farach-Colton, Munro, Rauhe, and Thorup, 2002].

• Dynamic sets scannings [Bender, Cole, Demaine, and Farach-Colton, 2002a].

• Meaningful lower bounds [Brodal and Fagerberg, 2003, Bender, Brodal, Fagerberg, Ge, He, Hu, Iacono, and Lopez-Ortiz, 2003].

Cache-Oblivious model and space-optimality. In the case of the Cache-Oblivious model, the simple variation that we introduce in order to capture completely the setting in which the permutations of the input set are the only actor in any step of a computation, can be viewed as a fusion of the two we introduced for the RAM model (Section 2.2) and for the External-Memory model (Section 2.4).

• The total order $(\mathcal{U}, \leq)$ is again our universe. Since we are not going to study any vector problem in the Cache-Oblivious model, the input set $\mathcal{I}$ is a subset of $\mathcal{U}$ (or we can also say that $k = 1$).

• Initially the $n$ elements in $\mathcal{I}$ are stored in $n$ contiguous input locations in the main memory. An algorithm is allowed to perform only comparisons and exchanges over the input elements. The cache is composed of input locations too and, during a computation, the $n$ input elements will be scattered in throughout all the $n + M$ input locations.

• An algorithm is also allowed of a constant number of auxiliary locations with $O(\log n)$ bits each. For simplicity we will suppose the $O(1)$ auxiliary locations constantly residing in cache (that would be also the behavior of the optimal off-line replacement strategy in the ideal-cache model, since it is plausible that the auxiliary locations are accessed very often).

For the case of a dynamic input set, the model supports the adding of the new $(n + 1)$st input location or the removing of the $n$th one. The growth of the size (number of bits) of any auxiliary location follows the common transdichotomous model [Fredman and Willard, 1993].
Data locality and space optimality. As we said before, during a computation in a hierarchical memory model, the data flows up and down in the hierarchy. To amortize the cost of the inter-level flow of information, the data transfers between two adjacent levels need to involve blocks of contiguous locations. Therefore, any algorithm aiming to be efficient in a hierarchical memory model has to maintain some form of data locality, meaning that it has to concentrate the computation over a limited amount of close locations for a relatively long period of time.

This characteristic seems to clash with the space-optimality requirement. This can be easily recognized examining any of the advanced space-optimal result in the RAM model like for example any in-place merging algorithm (see [Kronrod, 1969], [Trabb Pardo, 1977], [Salowe and Steiger, 1987], [Huang and Langston, 1988b]...). The tendency to jump all over the input sequence is clearly damning evidence against a good data locality. In settings as the External-Memory model, where the sizes of block and internal memory is known, we are allowed to tune the amount of data locality of algorithms and data structures (as in the case of the B-Tree, for example), and hence the task of having both data locality and space optimality seems to be a little easier. That would not seem to be the case for the Cache-Oblivious model.
Chapter 3

The Problems

Abstract

In this chapter we introduce the problems we are going to solve in Part II and III of this thesis. We also review the motivations and the previous work concerning any of those problems and we briefly state our findings.

First, we introduce the sorting problems that will be faced in Part II.

- Sorting with minimum data movements.
- Sorting in the Cache-Oblivious model.
- Sorting in a multidimensional domain.
- Generalized (multiway) merging and adaptive sorting.

Then, we introduce the static and dynamic searching problems that will be faced in Part III.

- Searching in a multidimensional domain.
- Implicit dictionary problem.

3.1 Sorting Problems

From the very beginnings of computer science, sorting is one of the most fundamental problems, of great practical and theoretical importance. Virtually in every field of computer science there are problems that have the sorting of a set of objects as a primary step toward solution. (For early history of sorting, see Section 5.5 in [Knuth, 1973]). However, the reader does not have to trust us on this as it is also the opinion of of a very renown Computer Scientist. From the third volume Sorting and Searching of The Art of Computer Programming we can get the clear opinion of the author in this matter:
“Sorting algorithms make an interesting case study of how to attack computer programming problems in general.”


Unfortunately, in the very same volume (but only until the second edition) Donald Knuth seems to come forward with a less positive opinion about how effective the pair (human, computer) can be in sorting a set of objects:

“Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer.”


In this section we describe some sorting-related problems, review their history and the previous work done and briefly stating our findings. We are going to solve them in Part II of this thesis.

3.1.1 Sorting with minimum data movements

As we saw in Chapter 2, in the derivatives of the Random Access Machine model it is natural to measure the efficiency of a sorting algorithm (as for any other algorithm) with three metrics: the number of comparisons it requires, the number of element moves it performs and the number of auxiliary memory locations it uses, besides the ones strictly necessary for the input elements. It is well-known that a comparison-based algorithm must perform, in the worst case, at least the following number of comparisons to sort an input sequence consisting of \( n \) elements:

\[
[\log n!] \geq n \log n - n \log e \approx n \log n - 1.443n
\]

By [Munro and Raman, 1996a], we know that a corresponding lower bound for element movements is

\[
[3/2n]
\]

Concerning upper bounds for the number of comparisons, already the plain version of mergesort gets closely to the optimum, with at most \( n \lfloor \log n \rfloor - n + 1 \) comparisons. However, this algorithm needs also \( n \) auxiliary locations for storing \( n \) elements, and therefore it is not space optimal. That is, it does not work with only a constant auxiliary storage, besides the data stored in the input sequence. Even in a practical context, in-place algorithms play an important role, because they maximize the size of data that can be processed in the main memory without an access, during the computation, to a secondary storage device.
3.1.1.1 Comparison, move and space optimal sorting

Comparison and space optimal sorting. The rich history of comparison and space optimal sorting algorithms, using \(O(n \log n)\) comparisons and, at the same time, \(O(1)\) auxiliary storage, begins with a binary-search version of insertsort. This algorithm uses less than \(\log n! + n\) comparisons, only a single input location for putting elements aside, and only \(O(1)\) index variables, of \(\log n\) bits each, for pointing to the input sequence. Unfortunately, the algorithm performs \(\Omega(n^2)\) element moves, which makes it unacceptably slow, as \(n\) increases.

The Heapsort [Floyd, 1964, Williams, 1964] was the first space optimal sorting algorithm with a total running time bounded by \(O(n \log n)\) in the worst case. More precisely, it uses less than \(2n \log n\) comparisons with the same \(O(1)\) auxiliary space requirements as insertsort, but only \(n \log n + O(n)\) moves, if the moves are organized a little bit carefully.

Since then, many cloned versions of Heapsort have been developed; the two most important ones are bottom-up Heapsort [Wegener, 1993] and a \(\log^*\)-variant [Carlsson, 1992]. Both these variants use not only the same number of moves as the standard Heapsort, but even exactly the same sequence of element moves for each input, if the input does not contain duplicated elements. (See also the procedure “shiftdown” in [Schaffer and Sedgewick, 1993] and the papers [Fleischer, 1994, Bollobás, Fenner, and Frieze, 1996]). However, they differ in the number of comparisons. Though bottom-up variant uses only \(3/2n \log n + O(n)\) comparisons, its upper bound for the average case is even more important; with \(n \log n + O(n)\) comparisons, it is one of the most efficient in-place sorting algorithms. The \(\log^*\)-variant is slightly less efficient in an average, but it guarantees less than \(n \log n + n \log^* n\) comparisons in the worst case. For a more detailed analysis, see also [Li and Vitányi, 1993, Schaffer and Sedgewick, 1993].

Then in-place variants of a \(l\)-way mergesort came to the scene [Katajainen, Pasanen, and Teuhola, 1996, Reinhardt, 1992], with at most \(n \log n + O(n)\) comparisons, \(O(1)\) auxiliary storage, and \(\epsilon n \log n + O(n)\) moves. Instead of merging only 2 blocks, \(l\) sorted blocks are merged together at the same time. Here \(l\) denotes an arbitrarily large, but fixed, integer constant, and \(\epsilon > 0\) an arbitrarily small, but fixed, real constant. Except for the first extracted element in each \(l\)-tuple of blocks, the smallest element is found with \(\log l\) comparisons, if \(l\) is a power of two, since the \(l\) currently leftmost elements of the respective blocks are organized into a selection tree. Though \(\log l\) is more than one comparison required in the standard 2-way merging, the number of merging sweeps across the sequence comes down to \(\lceil \log n / \log l \rceil\), so the number of comparisons is almost unchanged. As an additional bonus, the number of element moves is reduced if, instead of elements, only pointers to elements are swapped in the selection tree.

By the use of some other tricks, the algorithm is made in-place and the size of auxiliary storage is reduced to \(O(1)\). The early implementation of this algorithm, having so promising upper bounds, turned out to be unacceptably slow. It was
observed that operations with indices representing the current state of the selection tree became a bottleneck of the program. Fortunately, the state of a selection tree with a constant number of leaves can be represented implicitly, without swapping indices. This indicates that even by summing comparisons and moves we do not get the whole truth, the arithmetic operations with indices are also important.

The $l$-way variant has been generalized to a $(\log n/\log\log n)$-way in-place mergesort [Katajainen and Pasanen, 1999]. This algorithm uses $n\log n + O(n\log\log n)$ comparisons, $O(1)$ auxiliary storage, and only $O(n\log n/\log\log n)$ element moves. Since $l$ is no longer a constant here, the information about the selection tree is compressed, among others, into bits of $(\log n)$-bit index variables by complicated bitwise operations, which increases, among others, the number of arithmetic operations. As the authors declare, the algorithm is mainly of theoretical interest. The theoretical contribution of that algorithm is important, since it is the first member of the comparison and space optimal family breaking the bound $\Omega(n\log n)$ for the number of moves.

**Move and space optimal sorting.** The move and space optimal family of algorithms, sorting with $O(n)$ element moves and $O(1)$ auxiliary space, is not so numerous. The first algorithm of this type is *selectsort*, which is a natural counterpart of *insertsort*. Carefully implemented, it sorts with at most $2n - 1$ moves, a single input location for putting one element aside, and $O(1)$ indices. Unfortunately, it performs also $\Omega(n^2)$ comparisons.

As shown in [Munro and Raman, 1996a], $O(n^2)$ comparisons and $O(1)$ indices suffice for reduction of the number of moves to the lower bound $\lfloor 3/2n \rfloor$.

Another improvement is a *generalized Heapsort* [Munro and Raman, 1992]: It is based on a heap in which internal nodes have $\lfloor n^{1/\ln l} \rfloor$ children, for a fixed integer $l$. The corresponding heap tree is thus of constant height, which results in an algorithm sorting with $O(n)$ moves, $O(1)$ space, and $O(n^{1+\varepsilon})$ comparisons.

**Comparison and move optimal sorting.** Finally, consider the comparison and move optimal family, sorting with $O(n\log n)$ comparisons and $O(n)$ element moves. The first member is a so-called *tablesort* [Knuth, 1973, Munro and Raman, 1992]. We use any algorithm with $O(n\log n)$ comparisons but, instead of elements, we move only indices pointing to the elements. When each element’s final position has been determined, we transport all elements to their destinations in linear time. However, this algorithm requires $\Omega(n)$ auxiliary indices.

The storage requirements have been reduced to $O(n^\varepsilon)$ by a variant of *samplesort* [Munro and Raman, 1992]. The same result can also be obtained by the in-place variant of the $l$-way mergesort [Katajainen, Pasanen, and Teuhola, 1996, Katajainen and Pasanen, 1999], mentioned above, if $l = \lceil n^\varepsilon \rceil$. This reduces the number of merging sweeps down to a constant, which results in $O(n\log n)$ comparisons and $O(n)$ element moves. Such modification is no longer in-place, it uses $O(n^\varepsilon)$ auxiliary
3.1. SORTING PROBLEMS

indices to represent a selection tree.

The PileSort [Katajainen and Vitale, 2003] requires \( n \log n + 0.59n + O(1) \) comparisons, \( 2.5n + O(1) \) moves, and \( 4n + O(w) \) extra bits to sort a sequence of \( n \) elements, where \( w \) denotes the length of the machine word.

**Comparison, move and space optimality at the same time?** So far, there was no known algorithm sorting, in the worst case, with \( O(n \log n) \) comparisons, \( O(n) \) moves and \( O(1) \) auxiliary storage (and \( O(n \log n) \) arithmetic operations).

This ultimate goal has only been achieved in the average case [Munro and Raman, 1992]. In the worst case, the algorithm uses \( \Omega(n^2) \) comparisons but, for a randomly chosen permutation of input elements, the probability of this worst case scenario is negligible.

It was generally conjectured, for many years, that an algorithm matching simultaneously the asymptotic lower bounds on all above computational resources does not exist. For example, in [Raman, 1991], it was proved that the algorithm with \( O(n^{1+\varepsilon}) \) comparisons using generalized heaps is optimal among a certain restricted family of space optimal sorting algorithms performing \( O(n) \) moves. It was hoped that, by generalizing from a restricted computational model to all comparison-based algorithms, we could get a higher trade-off among comparisons, moves, and auxiliary space.

The result we will present in Chapter 5, contradicts the above conjectures and closes a long-standing open problem. The presentation in Chapter 5 is based on the papers [Franceschini and Geffert, 2003] (FOCS 2003) and [Franceschini and Geffert, 2005] (Journal of the ACM, to appear).

*We shall exhibit the* first comparison, moves and space optimal sorting algorithm. *Moreover we will be able to give a precise analysis down to the constant factors. Our algorithm operates in-place, with at most* \( 2n \log n + o(n \log n) \) *element comparisons and* \((13+\varepsilon)n \) *element moves in the worst case, for each* \( n \geq 1 \). *Here* \( \varepsilon > 0 \) *denotes an arbitrarily small, but fixed, real constant. The number of auxiliary arithmetic operations with indices is bounded by* \( O(n \log n) \). *We can slightly reduce the number of moves, to* \((12+\varepsilon)n \), *in a modified version that uses* \( 6n \log n + o(n \log n) \) *comparisons.*

### 3.1.1.2 How difficult is to achieve the stability?

In the general case of input sequences with repeated elements, an important requirement for a sorting algorithm is to be *stable*: the relative order of equal elements in the final sorted sequence is the same found in the original one.

As we said before, the *Heapsort* [Williams, 1964] is the first space optimal sorting algorithm performing \( O(n \log n) \) comparisons in the worst case. However, this algorithm is *highly unstable* and the number of element moves performed in the worst case is \( O(n \log n) \). The existence of a sorting algorithm that is *stable, comparison and*
space optimal was proven by Trabb Pardo [Trabb Pardo, 1977], with the introduction of the first stable in-place merging algorithm. Concerning the partition-based approach, the ordinary Quicksort [Hoare, 1962] is only space optimal. An unstable, comparison and space optimal sorting can be derived using an in-place selection algorithm like [Lai and Wood, 1988]. Other examples are in [Durian, 1986, Bing-Chao and Knuth, 1986, Wegner, 1987]. A stable, comparison and space optimal version of the Quicksort has been proposed in [Katajainen and Pasanen, 1994].

Concerning the sorting algorithms with optimal number of data moves, the classical selection sort operates in-place, and performs $O(n)$ moves and $O(n^2)$ comparisons in the worst case but it is not stable. The improvement in [Munro and Raman, 1992] with a generalization of the Heapsort performing $O(n^{1+\varepsilon})$ comparisons, maintains the instability. Finally, a stable algorithm with these same bounds was presented in [Munro and Raman, 1996b].

If the space optimality is given up, the address-table sort [Knuth, 1973] can be easily modified to achieve the stability.

As we have seen, the algorithm in [Katajainen and Pasanen, 1999] is the first sorting requiring $o(n \log n)$ moves in the worst case while guaranteeing in-placeness and $O(n \log n)$ comparisons. Unfortunately the technique of internal buffering (we will introduce this technique in Chapter 4) that is fundamental to their result, makes their algorithm unstable. Our comparison, move and space optimal algorithm in [Franceschini and Geffert, 2003, 2005] (see Chapter 5) is highly unstable for the same reason.

As a matter of fact, the internal buffering technique (see Chapter 4) is, at the same time, the best friend and worst foe of any advanced result in space optimality. Introduced in [Kronrod, 1969], this technique has been at the base of virtually every result in the field, for more than three decades. The basic idea is simple: some of the input elements are used as placeholders in order to simulate a working area and permute the other elements with less moves. It is easy to understand how disruptive the internal buffering is when the stability of the algorithm is an objective: if the placeholders are not distinct, the original order of identical placeholders can be lost using the simulated working area. As witness of the clash between stability and internal buffering technique, we can cite the huge difference in complexity between the first in-place linear time merging algorithm, due to Kronrod [Kronrod, 1969], and the first stable one by Pardo [Trabb Pardo, 1977]. Therefore, there seems to be the need for new powerful techniques for space optimality.

The result we will present in Chapter 6 closes the problem. The presentation in Chapter 6 is based on the papers [Franceschini, 2005a] (STACS 2005) and [Franceschini, 2005c].

We will present the first comparison, move and space optimal algorithm able to sort a multiset stably. It uses a completely new approach to the problem that allow the use of an internal buffering technique that is adaptive to the number of distinct elements present in the input sequence.
3.1.2 Sorting in the Cache-Oblivious model

In [Prokop, 1999] it has been proved that, assuming the presence of a tall cache \( (M = \Omega(B^2)) \), the cache complexity of any sorting algorithm in the Cache-Oblivious model is

\[
\Omega\left(\frac{n}{B} \log_M n\right)
\]

The work complexity, that is the number of operations, is \( \Omega(n \log n) \) and that lower bound comes directly from the comparison model. These lower bounds are matched by two sorting algorithms presented in the seminal paper [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Prokop, 1999]: Funnelsort and Distribution sort.

In [Frigo, Leiserson, Prokop, and Ramachandran, 1999], both the sorting algorithms relied on the so-called tall cache assumption i.e. \( M = \Omega(B^2) \). Subsequently, Brodal and Fagerberg [Brodal and Fagerberg, 2003] proved that the tall cache assumption (or its less stringent version \( M = \Omega(B^{1+\epsilon}) \) seen in [Brodal and Fagerberg, 2002a]) is necessary for the cache optimality in sorting. So, from now on we also assume that \( M = \Omega(B^2) \).

3.1.2.1 Funnelsort

The Funnelsort was successively simplified by Brodal and Fagerberg [Brodal and Fagerberg, 2002a] with a variant called Lazy Funnelsort. The Funnelsort divides the input sequence into about \( n^{1/3} \) contiguous sub-sequences of about \( n^{2/3} \) elements each. Then it recursively sorts these sub-sequences and finally produces the sorted sequence using an \( n^{1/3} \)-merger. Essentially, a \( t \)-merger is a complete binary tree with \( t \) leaves (one for each sorted input sequence), with a buffer for each edge and laid out in memory following the recursive van Emde Boas layout (see [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Prokop, 1999] and Chapter 7) by which also the size of buffers is determined. When invoked, a \( t \)-merger outputs the first \( t^3 \) elements of the sorted sequence obtained by merging the \( t \) sorted input sequences.

This particular layout is also a basic tool adopted in all the optimal dynamic dictionaries so far developed for the Cache-Oblivious model, as in [Prokop, 1999, Bender, Demaine, and Farach-Colton, 2000, Bender, Duan, Iacono, and Wu, 2002d, Brodal, Fagerberg, and Jacob, 2002, Bender, Cole, and Raman, 2002b, Franceschini and Grossi, 2003b,c]. In [Vinther, 2003], the author proved that, for the case of sorting, the layout is not important but the buffer sizes. Vinther also gave a variant of Funnelsort that uses only \( o(n) \) extra space.

3.1.2.2 Distribution sort

The Distribution sort in [Frigo, Leiserson, Prokop, and Ramachandran, 1999] is very peculiar. It is not a "pure" distribution sort, like the Quicksort for example, because the divide phase of its divide et impera approach does not necessarily produce
**disjoint** subproblems (where two subproblems \( P' \) and \( P'' \) are disjoint if for any two elements \( p' \) and \( p'' \) belonging respectively to the input set of \( P' \) and \( P'' \), we have that \( p' \leq p'' \)).

The algorithm performs the following steps.

(i) It divides the input sequence into about \( n^{1/2} \) contiguous sub-sequences of the same size and recursively sorts these sub-sequences.

(ii) It distributes the elements in \( q \leq \sqrt{n} \) buckets \( B_1, \ldots, B_q \) such that \( |B_i| \leq 2\sqrt{n} \) and \( \max\{x : x \in B_i\} \leq \min\{x : x \in B_{i+1}\} \), for \( i = 1, 2, \ldots, q - 1 \).

(iii) It recursively sorts each bucket.

(iv) It put the elements back into the memory zone where they were initially contained.

The way the second step is performed is crucial to obtain the optimal cache complexity. Unlike other kinds of sorting based on distribution, the phase in which the pivots (or sample set) are chosen and the distribution phase are not separated. In [Frigo, Leiserson, Prokop, and Ramachandran, 1999] a *bucket splitting* technique is used: the buckets have a variable size upper bounded by \( 2\sqrt{n} \); when a bucket reaches that threshold, its medium element is selected (that will be a new pivot) and the bucket is partitioned in two new buckets of size \( \sqrt{n} \). The bucket splitting technique alone it is not sufficient to achieve a good cache complexity. The order in which the sub-sequences are processed is critical. For example, if we process the sub-sequences from the leftmost to the rightmost one, distributing the elements of each sub-sequence along all the existing buckets in one single run, we incur in \( O(n) \) cache misses instead of \( O(n/B) \), as required by the cache optimality. As we will see, in our result we do not make use of the bucket splitting technique but we use that particular processing order introduced in [Frigo, Leiserson, Prokop, and Ramachandran, 1999] for disposing some particular groups of elements.

### 3.1.2.3 Proximity Mergesort

The only two optimal and cache-oblivious sorting algorithms known so far were the original Funnelsort (and its variant, the Lazy Funnelsort) and Distribution sort, if we exclude the algorithm directly derivable from from the priority queue in [Brodal and Fagerberg, 2002b] (the other optimal priority queue of Arge et al. [Arge, Bender, Demaine, Holland-Minkley, and Munro, 2002] relies on an optimal sorting algorithm itself).

Moreover, space optimality seems to be challenging in the Cache-Oblivious model because of the apparently antithetical requirements needed to obtain the maximum space saving and the data locality. For these reasons, the techniques originally
developed for this problem in the RAM and External Memory models appear to be of difficult application in the cache-oblivious scenario.

In Chapter 7 we will close the problem of space optimal sorting in the Cache-Oblivious model. The presentation in Chapter 7 is based on the paper [Franceschini, 2004] (SODA 2004).

We shall present the Proximity Mergesort, the first space optimal sorting algorithm for the Cache-Oblivious model with a work complexity $O(n \log n)$ and a cache complexity $O \left(\frac{n}{B} \log_M n\right)$. Furthermore, with our new algorithm we introduce a completely new approach to the sorting problem for this model of computation.

3.1.3 Sorting in a multidimensional domain

We are interested in studying the computational complexity of the classical problem of comparison-based sorting by considering the case in which the elements are of non-constant length, denoted $k$. We are interested in evaluating asymptotically the influence of $k$ in terms of the optimality of the bounds required for lexicographical sorting.

3.1.3.1 Space optimality and multidimensional domains

We aim at minimizing simultaneously the time and space bounds under the assumption that the elements are vectors $x \in \mathcal{U}^k$ of $k$ scalar components over a totally ordered, possibly unbounded set $\mathcal{U}$. We adopt the in-place RAM model for vectors we introduced in Section 2.2. The model easily extends to $k$-field records in $\mathcal{U}_1 \times \mathcal{U}_2 \times \cdots \times \mathcal{U}_k$.

As we have seen, the in-place RAM model for vectors naturally extends the comparison model to elements of non-constant length. Although some of the previous work uses the model in which chunks of the vectors can also be exchanged, we prefer to impose the requirement that vectors are indivisible and should be entirely exchanged. We obtain optimal bounds with this strict model without requiring the extra power of the other model. This model is useful for studying, in an abstract way, the complexity of in-place sorting and searching for a variety of elements of length $k$: fixed-length strings\(^1\), $k$-field records, $k$-dimensional points, $k$-digit numbers, etc.

In this model, even the fundamental problems represent a real challenge. One significant example is how to perform in-place searching on a set $\mathcal{V}$ of $n$ vectors. The problem has been introduced in [Hirschberg, 1978], with upper bounds of $O(k + \ldots)$.

\(^1\)For example, we can fit the model to strings in the C language, where the $n$ vectorial locations are declared as char $\mathcal{V}[n \times k]$ by assigning entries $\mathcal{V}[(i - 1)k], \ldots, \mathcal{V}[ik - 1]$ to the $i$th vectorial location, $1 \leq i \leq n$, and using $O(1)$ local int variables as auxiliary locations. Note that no string terminator is needed in $\mathcal{V}$; also, using an array of pointers char *$p[n]$ referring to the beginnings of vectors in $\mathcal{V}$ (i.e., $p[j] = \mathcal{V} + jk$ for $0 \leq j < n$) would violate the model by requiring $O(n)$ auxiliary locations.
time and $O(k \log n)$ time and, until 2004, the best result in the field has been [Andersson, Hagerup, Håstad, and Petersson, 1994, 2001, Andersson, Håstad, and Petersson, 1995b] where with very sophisticated techniques, the authors were able to prove the following tight bound

$$
\Theta \left( \frac{k \log \log n}{\log \log(4 + \frac{k \log \log n}{\log n})} + k + \log n \right)
$$

whenever a lexicographical sorted sequence is assumed. As we will see later, in [Franceschini and Grossi, 2004b] we have been able to prove that the optimal bound for the problem is $\Theta(k + \log n)$.

Space optimal sorting is an even more intriguing example in the difficult scenario introduced by the assumption of a multidimensional domain. Any space optimal sorting algorithm for unidimensional domain can be turned into an $O(nk \log n)$-time space optimal algorithm for vectors, loosing optimality in this way. The lower bound of $\Omega(nk + n \log n)$ time easily derives from decision trees [Knuth, 1973].

### 3.1.3.2 Previous sub-optimal solutions

If the number of comparisons is to be minimized, the best up-to-date result for space optimal sorting is $n \log n + O(nk \log^* n)$ scalar comparisons and $n \log n + O(nk)$ vector exchanges by Munro and Raman [Munro and Raman, 1991]. Since each vector exchange takes $O(k)$ time, the time complexity sums up to $O(nk^2 + nk \log n)$. The number of comparisons can be reduced to $n \log n + O(nk)$ using Ultimate Heapsort [Katajainen, 1998]. This observation was made in [Katajainen, Pasanen, Rosaz, and Teuhola, 1999].

For the same reason, the multikey Quicksort analyzed in [Bentley and Sedgewick, 1997] yields a non-optimal algorithm of cost $O(nk \log n)$ when adapted to run in the in-place model for vectors, since it requires $O(n \log n)$ vector exchanges. The original version of the algorithm takes $O(nk + n \log n)$ time since it can exploit $O(n)$ auxiliary locations to store the pointers to the vectors. Sorting exchanges the pointers rather than the vectors, following the address table sorting suggested in [Knuth, 1973] at page 74. When records should be physically rearranged in storage so that their elements are in order, Knuth writes that address table sorting is preferable to avoid moving the bulky records around. The reason is apparent; each move of a record is expensive while moving a pointer to the record is more efficient. The in-place RAM model for vectors fits this intuition formally by introducing a cost of $O(k)$ for exchanging vectors. However, it forbids the use of an array of $O(n)$ auxiliary pointers as done in address table sorting. In other words, exchanging elements is only doable by exchanging their bulky records rather than their pointers. (We identify the records with their elements for the sake of discussion.)

As can be seen, the cost of exchanging and comparing vectors is the major bottleneck towards optimality. The in-place RAM model for vectors is centered
around this feature and requires algorithms operating with \(O(n)\) vector moves. As we have seen, in [Franceschini and Geffert, 2003, 2005] (Chapter 5) an optimal in-place algorithm for unidimensional domains with \(O(n)\) data moves is devised and subsequent results [Franceschini, 2004, 2005a] (Chapters 7 and 6) have shown how to achieve cache-obliveness or stableness for in-place sorting. However, the \(O(k)\)-time cost of each vector comparison makes these methods non-optimal in our setting.

3.1.3.3 The solution and its implications

The above discussion highlights the fact that the known algorithms are unable to simultaneously achieve time optimality and space optimality for sorting vectors. In Chapter 8 we will solve this problem. The presentation in Chapter 8 is based on the paper [Franceschini and Grossi, 2005a].

We shall present the first in-place optimal sorting algorithm for vectors (or more generic multidimensional domains). Let \(k\) be the dimension of the input vectors. Our algorithm sorts any input set of \(n\) elements in \(O(nk + n \log n)\) time in the in-place RAM model for vectors.

An implication of our result is that we resolve the following apparent anomaly between searching and sorting. As previously mentioned, the results for efficient in-place searching in [Andersson, Hagerup, Håstad, and Petersson, 1994, 2001, Andersson, Håstad, and Petersson, 1995b, Franceschini and Grossi, 2004b, Hirschberg, 1978, 1980, Kosaraju, 1979] stem from the fact that the vectors are already given in lexicographic order. Before our algorithm, we could not preprocess an arbitrary arrangement of vectors to this end, since no optimal in-place algorithm was known for the lexicographic sorting problem itself. While we could prove the existence of certain permutations of the vectors leading to efficient in-place searching, we were not yet able to construct them with optimal space and time bounds. With the result in [Franceschini and Grossi, 2005a] (Chapter 8) we can provide optimal preprocessing for efficient space optimal searching when the vectors are initially arranged in any arbitrary order, with a preprocessing cost of \(O(nk + n \log n)\) time.

Asymptotically speaking, another implication is that sorting bulky records can be done optimally in place by exchanging them directly without using the \(O(n)\) auxiliary locations required by Knuth’s address table sorting. We remark that our bounds hold for internal sorting. Interestingly, the results in [Arge, Ferragina, Grossi, and Vitter, 1997] show that breaking up long strings into smaller substrings gives more computational power when sorting in external memory. In our internal-memory setting, however, breaking vectors does not give more power for space optimal sorting by comparisons. We can trivially derive a lower bound of \(\Omega(nk + n \log n)\) for the model where we can break the vectors into chunks to be exchanged. Since our bound of \(O(nk + n \log n)\) also holds for the latter model, we easily derive its optimality.
3.1.4 Generalized merging problem and adaptive sorting

3.1.4.1 The generalized (multiway) merging problem

As we know, in the sorting problem we are given a set of $n$ elements from a given total order and they have to be disposed in the ordered permutation induced by the order relation. We will deal with the following, strictly related problem:

**Generalized Merging Problems.** In the balanced merging problem, we are given $s \leq n$ sorted sequences of $n/s$ elements each drawn from $U$ and they have to be fused into a single sorted sequence. In the unbalanced merging problem the total number of input elements is still $n$ but the $s$ sorted subsequences to be fused can differ in their lengths.

Therefore, algorithms solving the merging problems are supposed to exploit the pre-sortedness of the input elements in order to arrive to the final sorted sequence with less computational effort.

Seeing the parametric definition of the merging problems natural questions arise: How far can we push the parameter $s$? **Is there a comparison, move and space optimal solution for the merging problems that is independent from any particular value of $s$?** How heavily marked is the boundary between merging and sorting problems? Let us enter deeply into this matter seeing what happens if we drop one of the optimality constraints.

3.1.4.2 Sub-optimal solutions

If we give up the optimality on the number of moves, a space optimal and comparison optimal solution that is independent from the value of $s$ follows immediately by the existence of a comparison, move and space optimal solution for the case $s = 2$. As we will see later, the research around the existence of an algorithm that could fuse two sequences of $m$ elements each, using $O(m)$ comparisons, $O(m)$ moves and $O(1)$ auxiliary locations has been active and successful since the late sixties [Kronrod, 1969]. Given that fact, a space optimal and comparison optimal solution for the merging problem that is independent from the value of $s$ is a simple variation of the plain Mergesort. It employs any full-optimal merging algorithm for $s = 2$ and starts the execution merging couples of sorted sequences instead of couples of elements. That brings us the wanted bounds of $O(n \log s)$ comparisons and $O(1)$ auxiliary locations at a cost of a sub-optimal $O(n \log s)$ number of moves.

If we overlook the optimality on the number of comparisons, we can just use the algorithm for the sorting problem in [Franceschini and Geffert, 2003, 2005] (see Chapter 5). With that algorithm, we can just ignore the sorted sequences, and sort the whole sequence using the optimal $O(n)$ moves and $O(1)$ auxiliary locations at the cost of a sub-optimal $O(n \log n)$ number of comparisons.
3.1. SORTING PROBLEMS

Finally, if the space optimality is to be sacrificed, a linear-space solution that is independent from the value of $s$ and performs $O(n \log s)$ comparisons and $O(n)$ moves can be obtained using a dictionary that is searchable in $O(\log m)$ comparisons (when it contains $O(m)$ elements) and updateable in $O(1)$ moves, either amortized or in the worst case (e.g. [Levcopoulos and Overmars, 1988b, Andersson and Lai, 1991, Fleischer, 1993]...). Any such dictionary can be simply used as a priority queue containing, at any time during the process, the smallest $s$ elements among the ones still in the sorted sequences in input.

3.1.4.3 The problem is difficult even for $s = O(1)$

As we said before, the research focusing on the existence of a comparison, move and space optimal solution for the merging problems has been fervent since the sixties, bringing results for the special case where $s = O(1)$. In spite of all the research efforts, at the best of our knowledge, so far, even the existence of a comparison, move and space optimal solution for the merging problem covering any range of values better than $s = O(1)$ was unknown (if we exclude the range $n^\epsilon \leq s \leq n$, for any real constant $\epsilon < 1$, obviously obtainable with a plain application of the sorting algorithm in [Franceschini and Geffert, 2003, 2005] (see Chapter 5). Some ideas suitable for the case $s = O(\text{polylog})$ are in [Katajainen and Pasanen, 1999], as we will see in Chapter 9.

As a further witness of the intrinsic difficulty of finding an optimal solution for the merging problem, even for the limited range with $s = 2$, we will quickly review the main results for this particular case. The first solution was proposed in [Kronrod, 1969], in that seminal paper fundamental tools for space-optimality, like the internal buffering technique (see Chapter 4), were introduced. Unfortunately, the two-way merging algorithm of Kronrod contained an insidious error compromising the correctness of the algorithm in the general case of input with repeated elements.

After Kronrod, Horvath [Horvath, 1978] devised a stable (i.e. the initial relative order of equal elements is maintained after the process) merging algorithm assuming the possibility of element modifications. Subsequently, Trabb Pardo [Trabb Pardo, 1977] removed this requirement.

The error in Kronrod’s work went undisclosed until [Salowe and Steiger, 1987], when a simpler way to stable merging was devised. In [Huang and Langston, 1988a] an unstable modification of Kronrod’s algorithm is given but unfortunately also that solution contains an error. Later on, the same authors gave a stable algorithm in [Huang and Langston, 1988b].

When $s = 2$, the lower bound for the number of comparisons in case of sequences of two different lengths $m < n$ is $\Omega(m \log(n/m))$. Symvonis achieved that lower bound in [Symvonis, 1995] and subsequently in [Geffert, Katajainen, and Pasanen, 2000] both stable and unstable algorithms with the same asymptotic bound but better constant factors were given.
3.1.4.4 Links with adaptive sorting

As we will see, a corollary of our main statement represent an important novelty for the adaptive sorting problem (see [Estivill-Castro and Wood, 1992] for a survey on the subject). In the adaptive sorting problem the initial order of the input elements is quantified with a pre-sortedness measure. The complexity measures for the adaptive sorting algorithms are expressed in function of the chosen pre-sortedness measure. With the development of this field, many pre-sortedness measures have been introduced together with a concept of optimality for any such measure. The measure $\text{Runs}$ is defined as the number of ascending contiguous sub-sequences of the input sequence. Using any space optimal merging algorithm as the one in [Salowe and Steiger, 1987], it is possible to achieve adaptive sorting algorithms that are space optimal and $\text{Runs}$ optimal but sub-optimal with respect to the number of moves. Similar results can be obtained with other measures but, at the best of our knowledge, so far, for any pre-sortedness measure, no full-optimal adaptive sorting algorithms were known.

3.1.4.5 Sorting by merging or merging by sorting?

In the presentation in Chapter 9, that is based on [Franceschini, 2005b], we will prove the following:

There exists an algorithm $A$ with the following property: For any $1 \leq s \leq n$ and for any set of $s$ sorted sequences containing a total of $n$ elements (drawn from $U$), $A$ computes the whole sorted sequence with $O(n \log s)$ comparisons, $O(n)$ moves and $O(1)$ auxiliary cells of memory.

Any solution for the balanced merging problem performs $\Omega(n)$ data moves. In the balanced merging problem the sorted subsequences are assumed to be of equal length. Hence, we have an full optimal solution for the balanced merging problem. So far it was unknown if any such algorithm existed, even for any restricted ranges of values for $s$ better than the plain $s = O(1)$.

Moreover, our main statement has an important corollary involving the field of the adaptive sorting algorithms. Since we can easily adapt our algorithm to sort $s$ sequences of different sizes $t_1, \ldots, t_s$ ($\Sigma t_i = n$) with the same bounds we stated above, we also prove that:

There exists a pre-sortedness measure, that is $\text{Runs}$, and a comparison, move and space optimal adaptive sorting algorithm for that measure. In other words, there exists an algorithm $A$ such that, for any sequence $X$ of $n$ elements, $A$ sorts $X$ with $O(n \log \text{Runs}(X))$ comparisons, $O(n)$ moves and $O(1)$ auxiliary locations of memory.
At the best of our knowledge, so far, for any pre-sortededness measure, no full-optimal adaptive sorting algorithms were known (see [Estivill-Castro and Wood, ACM Comp. Surveys, 24, Section 4]).

From a more intuitive point of view, we prove that it is possible to pass from merging to sorting in a seamless fashion, without losing the optimality with respect to any of the three main complexity measures. In light of this fact, we could say that merging really is not the mere subroutine with limited power that the well known sorting by merging approach has led us to believe. Instead, as we will see, a powerful sorting is a basic subroutine of our full-optimal merging algorithm.

It seems to be necessary to rethink the hierarchical relation between sorting and merging, as the title of Chapter 9 suggests. Until now, only one side of this relation has been known. The simple two-way merging algorithms have been proven to be a nice and effective brick to build simple sorting algorithms following the sorting by merging approach.

Now the situation seems to be inverted, the complicated comparison, move and space optimal sorting algorithm in [Franceschini and Geffert, 2003, 2005] (see Chapter 5) can be seen as a brick to build a very powerful merging algorithm following a new merging by sorting approach.

3.2 Searching Problems

In this section we describe some searching-related problems (both static and dynamic), review their history and the previous work done and briefly stating our findings. We are going to solve them in Part III of this thesis.

3.2.1 Searching in a multidimensional domain

From the third volume Sorting and Searching of The Art of Computer Programming we can have the clear opinion of the author about the importance of sorting in the searching problem:

"Imagine how hard it would be to use a dictionary if its words were not alphabetized!"

— Donald E. Knuth,


Sorting \( n \) elements in an array provides a basic data organization for optimal searching, with \( \Theta(\log n) \) comparisons in the worst case. This fact is corroborated by our common sense and everyday practice; sorting and searching are strictly related companions in designing and analyzing many algorithms for the comparison model. Does this viewpoint completely cover the intrinsic complexity of this fundamental problem?
3.2.1.1 Simple description, difficult solution

We are given \( n \) vectors from \( \mathcal{U}^k \). The vectors can be maintained in any permutation. Following the in-place RAM model for vectors, the \( i \)th vector in the permutation can be selected in constant time. Given any such vector, the \( j \)th component in it can be accessed in constant time as well.

We are interested in the fundamental problem of establishing whether a given vector appears in the input set of vectors without any preprocessing other than permuting the vectors in some way (no explicit index allowed). Setting \( k = 1 \) we obtain the classical searching problem for unidimensional domain. After sorting the set, we can run binary searching with optimal cost \( \Theta(\log n) \).

For arbitrary values of \( k \), previous work focused on elements sorted under the lexicographic order. The problem was introduced by Hirschberg [Hirschberg, 1978], with upper bounds of \( O(k + n) \) and \( O(k \log n) \) in the worst case. The former is obtained by scanning the sorted sequence while the latter is a simple binary search. The worst-case lower bound of \( \Omega(k + \log n) \) follows quite easily. The logarithmic term in \( n \) comes from the decision tree for searching in a set of \( n \) elements, while the linear term in \( k \) comes from the need of reading all the \( k \) components in the search element. The first nontrivial upper bound was

\[
O(k \log n / \log k)
\]

by Hirschberg [Hirschberg, 1980]. Kosaraju [Kosaraju, 1979] later improved it to

\[
O(k \sqrt{\log n} + \log n)
\]

With the sophisticated techniques in [Andersson, Hagerup, Håstad, and Petersson, 1994, Andersson, Håstad, and Petersson, 1995b, Andersson, Hagerup, Håstad, and Petersson, 2001] the authors proved that searching a lexicographically sorted sequence of vectors requires

\[
\Theta\left(\frac{k \log \log n}{\log \log (4 + \frac{k \log \log n}{\log n}) + k + \log n}\right)
\]

component comparisons in the worst case. Note that the bound is \( \Theta(\log n) \) when \( k = 1 \), which is a well known fact in algorithmics, as stated above.

The information theoretical lower bound of \( \Omega(\log n + k) \) can be achieved with other data structures keeping the vectors sorted and exploiting additional information in extra space (e.g., the longest common prefix information in suffix arrays by Manber and Myers [Manber and Myers, 1993] applied or the extra fields in ternary search trees by Bentley and Sedgewick [Bentley and Sedgewick, 1997]). The result in [Andersson, Hagerup, Håstad, and Petersson, 1994, Andersson, Håstad, and Petersson, 1995b, Andersson, Hagerup, Håstad, and Petersson, 2001] deals with the sequence of vectors alone, without any additional preprocessing and information (e.g., no pointers or integers) apart from \( O(1) \) values.
3.2.1.2 No sorting? Better searching!

But, as we will see, the lexicographically sorted sequence is the fundamental starting point for our result in Chapter 10. The presentation in Chapter 10 is based on the paper [Franceschini and Grossi, 2004b] (FOCS 2004). We will prove the following statement:

For any set $\mathcal{U}$ and for any set $S \subseteq \mathcal{U}^k$ of $n$ elements there exists a permutation $\mathcal{P}(S)$ different from the lexicographically sorted one that can be searched using $O(k + \log n)$ component comparisons and $O(1)$ values of auxiliary information. Therefore, the complexity of space optimal searching is $\Theta(k + \log n)$.

As a result, we provide a (hopefully!) unexpected insight on the relation between sorting and searching. We show that sorted arrays are not the best data organization suitable for searching $k$-dimensional elements. In this sense, sorting is an optimal placement of elements for searching only when $k = O(1)$.

Additionally, we can compute the rank of a search element external to the sequence, with a cost of $\Theta(k + \log n)$, and we can identify its predecessor or successor within the same bounds. We can also list the elements in the sequence belonging to a given input interval $[a, b]$ for two elements $a \leq b$. We attain an output-sensitive cost of $\Theta(k + \log n + \#\text{retrieved})$, where $\#\text{retrieved}$ denotes the number of elements in the input set that belong to the interval $[a, b]$. Since the latter bounds cannot be achieved with the sorted sequence alone, this strengthens the fact that our new permutation is more powerful than the lexicographically sorted one. As a side remark, we can obtain a sorted array from our permutation and vice versa in $O(nk)$ time; hence, our permutation is efficiently reversible.

Interestingly, our bounds hold also for the bit probe model [Elias and Flower, 1975, Yao, 1984]. In this model, the elements are distinct binary strings and the complexity accounts for the number of bits probed. We will show how to permute the input set so that membership requires $\Theta(k)$ bit probes in the worst case, which is optimal. Note that our bound gives an alternative perspective to Yao’s result [Yao, 1984] on achieving an optimal search for elements that can be permuted in an array without using extra space (i.e., for storing the name of a suitable hashing function). It also relates to the issues on extra space studied for perfect hashing by Fredman, Komlős and Szemerédi [Fredman, Komlós, and Szemerédi, 1984], non-oblivious hashing by Fiat, Naor, Schmidt and Siegel [Fiat, Naor, Schmidt, and Siegel, 1988b, 1992], and implicit probe search by Fiat and Naor [Fiat and Naor, 1989, 1993].

3.2.2 Implicit dictionaries

Almost every introductory textbook on algorithms and data structures presents, among others, two ways of organize $n$ distinct elements $a_1, a_2, \ldots, a_n$ into a sequence. The elements can be sorted so that binary search takes $O(\log n)$ time
(in [Knuth, 1973] Mauchly (1946) is credited for inventing this organization of the elements). Also, the elements can be permuted in heap order as shown by Williams and Floyd (1964), thus permitting to identify the current maximum element in constant time and supporting insertions and deletions of individual elements in $O(\log n)$ time [Williams, 1964, Floyd, 1964]. Indeed, sorted sequences and heaps are well-known examples of implicit data structures.

In the dictionary problem a set of $n$ distinct elements $a_1, a_2, \ldots, a_n \in \mathcal{U}$ is maintained over a total order, in which the only operations allowed on the elements are moves (or exchanges) and comparisons using the standard RAM model of computation [Aho, Hopcroft, and Ullman, 1976]. The dictionary supports the operations of searching, inserting and deleting an arbitrary element $x$. Besides membership, searching may also involve finding the predecessor or the successor of $x$, or reporting all the elements ranging in an interval $[x, x']$ where $x' \geq x$.

Heap-ordered sequences have the drawback of requiring $O(n)$ time for searching, while inserting or deleting an element in the middle part of sorted sequence may take $O(n)$ time. A long-standing question is whether there exists a permutation of the elements in the array of $n$ locations combining the best qualities of sorted arrays and heaps, so that each operation requires $O(\log n)$ time.

### 3.2.2.1 Implicit dictionary and their dynamics

Both sorted sequences and heap-ordered sequences store a suitable permutation of the $n$ elements, encoding an implicit tree by a partial order fixed a priori on the positions of the elements. No other "structural information" is required other than the elements in the permutation perceived as growing or shrinking with $n$. Along the same lines, we can see an implicit data structure for the dictionary problem as a sequence that is extendible to the right, one position at a time and is composed of $n$ distinct elements $a_1, a_2, \ldots, a_n$ suitably permuted, for $n \geq 1$. Following the in-place RAM model for vector ($k = 1$ in that case), the sequence is stored into $n$ adjacent vector locations, or input locations as we will call them from now on (since $k = 1$), with only $O(1)$ auxiliary locations used to store $O(\log n)$ bits of information.

The memory segment can be enlarged or shortened to the right in constant time, one location at a time. Inserting a new element $a_{n+1}$ extends the segment by one input location storing $a_{n+1}$, and shuffles $a_1, a_2, \ldots, a_n, a_{n+1}$ accordingly to an update algorithm. Deleting element $a_i$ yields a sequence of $n - 1$ elements producing a new permutation of $a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$.

That sort of encoding of structure by permutations of $a_1, a_2, \ldots, a_n$ is admissible by a simple information-theoretical argument showing that the number of permutations is much larger than the number of trees as $n!$ is much larger than, say, $C_n = \binom{2n}{n}/(n+1)$, the Catalan number of binary trees with $n$ nodes (our implicitly encoded data structures).
3.2. SEARCHING PROBLEMS

Nonetheless, so far no implicit data structure for the dictionary problem taking $O(\log n)$ time per operation was known. This fact may seem rather surprising considering that sorted sequences and heaps are long-lived examples of implicit dictionaries. As we shall see, the problem is algorithmically challenging and extending the implicit structure of sorted sequences and heaps is far from being an easy task. So far, in order to support the full repertoire of insert, delete and search operations in $O(\log n)$ time, the only alternative was implementing dictionaries as dynamic linked data structures such as AVL trees [Adel’son-Vel’skii and Landis, 1962] and other balanced data structures.

The implicit dictionary problem can be stated in the following abstract way.

**Implicit dictionary problem.** For a given model of computation $\mathcal{M}$ and a universe $\mathcal{U}$, is there a space optimal update algorithm $U$ to maintain a dynamic set $S \subseteq \mathcal{U}$ such that:

- Only $|S|$ locations plus $O(1)$ values of auxiliary information of $O(\log |S|)$ bits each are used.
- $S$ can be searched optimally (with respect to model $\mathcal{M}$).
- $U$ is space optimal and cost optimal (with respect to model $\mathcal{M}$).

### 3.2.2.2 Previous work on implicit dictionaries

Munro and Suwanda [Munro and Suwanda, 1980] examined the general case of ordered elements belonging to an unbounded universe (as in the models we introduced in Chapter 2) in the RAM model, where the $\Omega(\log n)$ lower bound on search time derives from the comparison model [Knuth, 1973]. They were the first to introduce the notion of implicit data structures inspired by the heap of Williams and Floyd, mentioning a previous (unpublished) result by Bentley et al. [Bentley, Detig, Guibas, and Saxe, 1978] supporting only insertions and searches. While the term “implicit” originated in [Munro and Suwanda, 1980], it has also been the subject of papers taking a somewhat different point of view, including a long lists of results in perfect hashing [Fredman, Komlós, and Szemerédi, 1984, Fiat, Naor, Schmidt, and Siegel, 1992], bounded-universe dictionaries [Brodnik and Munro, 1999, Pagh, 2002], and cache-oblivious data structures [Brodal, Fagerberg, and Jacob, 2002]. These results were obtained for less stringent models different from the model adopted by Munro and Suwanda and in following papers.

Yao [Yao, 1984] examined the special case of elements belonging to a bounded universe $U$, so that each element can be interpreted as an integer ranging in $0 \ldots |U| - 1$ and occupying a word of size $w = \Omega(\log |U|)$ bits. He proved that, independently of how the $n$ elements are permuted inside a segment of $n$ locations, searching requires $\Omega(\log n)$ time for sufficiently large $U$. However, he showed that encoding some information in one extra location of space (e.g., the name of a hash function) gives
more computational power and makes constant-time membership search possible for sufficiently large $U$. Since then, the two-level scheme by Fredman, Komlós and Szemerédi [Fredman, Komlós, and Szemerédi, 1984] and the many related papers provided a burst of interest in the design and the analysis of efficient algorithms for perfect hashing in constant-time search with $n + o(1)$ locations of space. Recent improvements in this direction are described in Fich and Miltersen [Fich and Miltersen, 1995] and Raman, Raman and Rao [Raman, Raman, and Rao, 2002], encoding the elements in at most $n$ locations of space by avoiding to store explicitly a permutation the elements as required in the lower bound of Yao. We remark that these techniques are not viable in our case, since they do not support all the dictionaries primitives (e.g., range searching) and if the input elements are drawn from an unbounded universe (see Chapter 2), they are atomic and can only be compared or exchanged (or moved).

Several papers faced the problem of designing an implicit data structure for the dictionary problem over an unbounded universe. Munro and Suwanda gave a lower bound of $\Omega(\sqrt{n})$ time per operation on implicit data structures based on a priori partial orders, such as the sorted array and the heap. Their bipartental heap matches that lower bound, giving $O(\sqrt{n})$ time per operation. They also showed how to beat the lower bound by using a partial order that is not fixed a priori and that is based on rotated lists, achieving $O(n^{1/3} \log n)$ time per operation.

In [Frederickson, 1983] the author presents a collection of data structures recursively using rotated lists, requiring just $O(1)$ RAM registers to operate dynamically. Search time is $O(\log n)$ while insertions and deletions are supported in time

$$O \left( n^{2/(\log n)} \log^{3/2} n \right) = o \left( n^\epsilon \right)$$

for any fixed value of $\epsilon > 0$.

Exploiting the properties of an in-place merge that is $O(n)$-time searchable at any time of its execution, Munro and Poblete [Munro and Poblete, 1987] provided an implicit data structure with $O(\log^2 n)$ search time supporting only insertions in $O(\log n)$ time.

In the mid 80s, Munro [Munro, 1986] achieved the first poly-logarithmic bounds holding simultaneously for searches and updates. Going through the crucial idea of encoding $O(\log n)$ bits by a permutation of $O(\log n)$ elements, he attained $O(\log^2 n)$ time by a variant of AVL trees [Adelson-Vel’skii and Landis, 1962], with $O(\log n)$ height and $O(\log n)$ accessed elements per level to encode pointers. The paper speculated that $\Theta(\log^2 n)$ may be optimal.

Borodin et al. [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988] and Radhakrishnan and Raman [Radhakrishnan and Raman, 2001] gave an interesting tradeoff between data moves in performing an update and the number of comparisons necessary for a search. Their lower bound does not, however, rule out the $O(\log n)$ behavior for the problem. They motivate the study of implicit data structures as an important topic in characterizing the set of permutations that are
searchable and updatable in logarithmic time.

3.2.2.3 Optimal implicit dictionaries over unbounded universes do exist.

Implicit B-Tree. In Chapter 11 we will describe the Implicit B-Tree. The presentation in Chapter 11 is based on the papers [Franceschini, Grossi, Munro, and Pagli, 2002] (FOCS 2002) and [Franceschini, Grossi, Munro, and Pagli, 2004] (Journal of Computer and System Sciences 68). We are going to prove the following statement.

There exists an implicit dictionary, the Implicit B-Tree, with the following bounds:

- In the External-Memory model, under the assumption $B = \Omega(\log n)$ the Implicit B-Tree can be searched and updated with $O(\log B n)$ block transfers in the worst case and can report $r$ consecutive elements with $O(\log B n + r/B)$ block transfers also in the worst case (both bounds are optimal).

- In the RAM model, the Implicit B-Tree can be searched and updated in $O\left(\frac{\log^2 n}{\log \log n}\right)$ time. That disproved the conjecture in [Munro, 1986].

Exponential Implicit Tree. In Chapter 12 we will concentrate on improving the bounds in the RAM model. We will describe the Exponential Implicit Tree. The presentation in Chapter 12 is based on the paper [Franceschini and Grossi, 2003a] (SODA 2003). We are going to prove the following statement.

There exists an implicit dictionary, the Exponential Implicit Tree, for the RAM model that is searchable in $O(\log n \log \log n)$ time in the worst case and is updatable in $O(\log n \log \log n)$ time in amortized sense.

Flat Implicit Tree. In Chapter 13 we will close the Implicit Dictionary Problem in the RAM and Cache-Oblivious model, but with update bounds optimal in amortized sense. The presentation in 13 is based on the papers [Franceschini and Grossi, 2003b] (ICALP 2003) and [Franceschini and Grossi, 2005b]. We are going to prove the following statement.
There exists an implicit dictionary, the Flat Implicit Tree, with the following optimal bounds:

- In the RAM model, the Flat Implicit Tree can be searched in $O(\log n)$ time in the worst case and can be updated in $O(\log n)$ time in amortized sense.

- In the Cache-Oblivious model, the Flat Implicit Tree can be searched with a work complexity $O(\log n)$ and a cache complexity $O(\log_B n)$ both in the worst case. Moreover, the Flat Implicit Tree can be updated with the same work and cache complexity but in amortize sense.

**Fast updates and polylogarithmic search.** Subsequently, in Chapter 14 we will disprove a very old conjecture about implicit dictionaries. The presentation in Chapter 14 is based on the paper [Franceschini and Munro, 2006] (SODA 2006). The most interesting lower bound on implicit dictionaries is the 1988 result [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988]. The authors prove a tradeoff between the search time and the update time in implicit dictionaries: if the update cost, number of comparisons and exchanges, for an implicit dictionary is $O(1)$ then the search cost must be $\Omega(n^\epsilon)$, for some constant $\epsilon > 0$. In their conclusion, the authors left open the question of whether a tradeoff of that kind would hold if only the modifications (i.e. data moves) performed during any update were considered. They conjectured that this would be in fact the case and that any implicit dictionary performing only $O(1)$ exchanges per update should very quickly get disorganized, thus ending up requiring $\Omega(n^\epsilon)$ comparisons per search. In Chapter 14, we answer to this long-standing open question by disproving the conjecture. First, we prove the existence of an implicit dictionary supporting searches with $O(\log n)$ comparisons in the worst case and updates with $O(1)$ exchanges and $O(\log n)$ comparisons in amortized sense. Then, we concentrate on worst case updates and prove the existence of an implicit dictionary that can be updated with a constant number of exchanges in the worst case while still needing only a polylogarithmic number of comparisons per search in the worst case.

**Worst case optimality.** Finally, in Chapter 15 we close definitely the Implicit Dictionary Problem for the RAM and Cache-Oblivious models (as well as for the External-Memory model) proving worst case bounds for both search and update operations. The presentation in 15 is based on the paper [Franceschini and Grossi, 2003c] (WADS 2003). Therefore, we will finally be able to say the last words for the Implicit Dictionary Problem:

There exists an implicit dictionary that can be optimally searched and updated in the RAM, Cache-Oblivious and External-Memory models.
The implicit dictionary of Chapter 15 is extremely complex and any practical implementations would be useless, due to the large hidden constant factors in the complexity bounds.
Chapter 4

Tools

Abstract

In this chapter we introduce some useful techniques and algorithms that we will use throughout this work. Most of these techniques can be used in a large variety of ways. Therefore, for any technique, we give here the main ideas common to all the variants we will have in the rest of this work. We will give the peculiar details of any particular variant when we make use of it. We describe the following techniques:

Block exchanging: how to exchange in-place and linear time, two adjacent blocks of elements, possibly of different sizes.

Small integer packing: how to store efficiently groups of integer of $o(\log n)$ bits each.

Bit stealing: how to encode the value of a bit in the relative order of a couple of distinct elements.

Internal buffering: how to use some elements of the input set as placeholders in order to permute the other ones efficiently.

Sorted order maintaining algorithms: how to maintain in sorted order and stored in $O(m)$ locations (and efficiently, of course) a totally ordered dynamic set of $m$ objects.

Hirschberg’s linear scan: how to search a sequence of $n$ $k$-dimensional vectors space optimally and with $O(k + n)$ comparisons.

Manber and Myers’ searching algorithm: how to search in a sequence of $n$ $k$-dimensional vectors in $O(k + \log n)$ time using $O(n)$ auxiliary locations.

4.1 Tools

In this chapter we introduce some classic techniques for space efficient algorithms and data structures and some well-known algorithms that are not usually associated
with this particular field but that we will prove to be fundamental tools in solving the problems introduced in Chapter 3.

Both the techniques and the algorithms will be used in a variety of different ways throughout this work. Given that large set of variants, it would not make any sense if, for any of the techniques or algorithms in this chapter, we tried to give a general purpose description that fits all the needs, because his would probably wind up with significantly more intricate solutions in the following chapters. Instead, for any technique, we give here only the main ideas common to all the variants we will have in the rest of this work. We will give the peculiar details of any particular variant when we make use of it. Let us start with two simple techniques.

Block reversing and exchanging. Let us consider a block \( X = x_1 \ldots x_t \) of \( t \) consecutive elements. We can obtain the reverse \( X^R = x_t \ldots x_1 \) in linear time and in-place simply exchanging \( x_1 \) with \( x_t \), \( x_2 \) with \( x_{t-1} \) and so forth. Two adjacent blocks \( X \) and \( Y \), possibly of different sizes, can be exchanged in-place and linear time with three block reversals, since \( YX = (X^R Y^R)^R \). If the two blocks are of the same size, they do not need to be adjacent: \( |X| \) simple exchanges (\( x_1 \) with \( y_1 \), \( x_2 \) with \( y_2 \) and so forth) would do the job in this particular case.

Small integers packing. From the requirement of space optimality we can use only a constant number of auxiliary locations (see Chapter 2) of \( O(\log n) \) bits each. However, we can still use a number \( f = \omega(1) \) of integers provided that they are small, that is if we have that \( uf = O(\log n) \), where \( u \) is the number of bits each small integer need in order to be represented. They can be packed into a single location and they can be modified and read in constant time just like if they were stored in \( f \) auxiliary locations (that can be done with a straightforward use of division by two, i.e. shift, operators). This technique has been used in many papers, for example in [Carlsson, Munro, and Poblete, 1988] just to stay on topic.

4.2 Bit Stealing

The idea. When we say that, in case of space efficient algorithms or data structures a permutation carries information we really mean it in the classic sense of the concept. Using the simple, but extremely powerful, technique of the bit stealing, bits of information can be encoded using a set of distinct elements (from a totally ordered universe) and, more importantly, they can be decoded selectively and with little slow-down. That is not the case of encoding information using, for example, an algorithm ranking all the possible permutations of the set of elements; this way we could encode the maximum quantity of information possible but it would not be easy to recover it selectively.

With the bit stealing technique the value of a bit is encoded in the relative order of a pair of distinct elements:
4.2. BIT STEALING

- the value 0 can be encoded having the pair stored in increasing order and,
- the value 1 can be encoded with the decreasing order of the two elements.

More complex uses. If we can dispose of many pairs of distinct elements, we can organize them in any complex way, for example in words of $O(\log n)$ bits each. From an algorithmic point of view, the encoded bits and words can be used pretty much as they were normal bits and words in auxiliary locations of memory. On the other hand, from the point of view of the complexity analysis, they have to be used with different cost models. For example, if we use words of $t$ contiguous encoded bits (i.e. pairs of distinct elements), we have the following costs:

- **Reading** the value in a word takes $t$ comparisons.
- **Changing** the value in a word takes $t$ comparisons and $O(t)$ exchanges in the worst case.
- **Any sequence of $2^t$ increments by one** starting from zero costs $O(t2^t)$ comparisons but only $O(2^t)$ exchanges in the worst case (and $O(1)$ exchanges in amortized sense, see [Cormen, Leiserson, Rivest, and Stein, 2001]).

Even if the costs of using stolen bits can be high, this technique is the foundation of almost every advanced results in the field of space optimal algorithms and data structures. As a witness of that we will point out the large difference in efficiency between the update operations of the last implicit dictionary not using the bit stealing and the first one using this technique: insertion and deletion cost a worst case time

$$O \left( 2^{\sqrt{2 \log n}} \log^3 n \right)$$

in [Frederickson, 1983] but only $O(\log^2 n)$ in [Munro, 1986].

It has to be noted that it is not necessary that the two distinct elements of a pair encoding one bit of information reside in adjacent locations nor that the words of an “encoding memory” reside all in a contiguous zone of locations. For example, in [Franceschini and Geffert, 2003] (see Chapter 5) the two elements of any pair used to steal a bit are one in the leftmost part of the memory and the other one in the rightmost part, always separated by $O(n)$ locations. Another meaningful example of that is the implicit dictionary in [Franceschini and Grossi, 2003b] where elements are organized in “chunks” of $O(\log n)$ distinct elements each that encode a variety of satellite information; during the lifetime of the structure, the chunks move around the memory carrying with them the encoded information.
Two difficult settings. One question that will have to be answered each time
the bit stealing technique is used is how the set of pairs of distinct elements can be
found. As we will see this will be problematic in two cases.

- When the input set is not necessarily composed by distinct elements (hence
  it is a multiset). Obviously in that case we cannot just take any adjacent
  pair because its elements may be equal. The method we will use to solve this
  problem is different from the recursive one used in [Munro and Raman, 1992].

- When the elements in the input set belong to a multidimensional domain and
  the total order to be exploited is the lexicographic one. In this case another
  problem adds to the previous one: in order to establish which is the larger
  between the two elements of a pair we may have to do $O(k)$ comparisons. As
  we will see, we cannot afford to pay such a cost and we will have to introduce
  a method lowering the decoding cost of a single stolen bit to $O\left(\frac{k}{\text{polylog}(n)}\right)$ (see
  Chapter 8).

4.3 Internal Buffering

The technique of the internal buffering was introduced in the seminal paper [Kro-
nrod, 1969], where the first space optimal, linear time, two-way merging was pre-
\newpage

\textbf{Example 4.1} Let us suppose that we have a set $S$ of $n$ distinct elements and let
us suppose that we are allowed to use an auxiliary integer memory $M$ of $n$ locations
of $\log n$ bits each. The elements in $S$ are considered atomic (as usual) and the
locations of $M$ cannot be used to store them but only to contain integers, pointers
etc. Finally, let us suppose that we are furnished with an algorithm $A$ such that it
can optimally sort any sequence of $m$ elements given

- an auxiliary integer memory $I$ of $m$ locations

- and an auxiliary input memory $E$ of $m$ locations (where, unlike $I$, the input
  elements can be stored).

The task is to use $A$ and $M$ to sort $S$ efficiently without using any other resource
(that is, the elements in $S$ can only be exchanged). How do we proceed?

We suddenly remember that an in-place linear time selection algorithm was intro-
duced in [Lai and Wood, 1988]. With such a powerful result by our side we
proceed with the following steps (let us suppose for the sake of description that $n$
is a power of 2):
(i) Using the in-place linear selection algorithm in [Lai and Wood, 1988], we select the element \( p \) with rank \( \frac{n}{2} \) in \( S \).

(ii) We partition in-place \( S \) around \( p \) (this is trivial since the elements in \( S \) are distinct). Let \( S' \) and \( S'' \) be the two resulting sets with \( |S'| = \frac{n}{2} - 1 \).

(iii) We sort \( S' \) using \( A \) and the elements in \( S'' \) to simulate the auxiliary input memory \( E \) required by the given sorting algorithm. This can be done very easily: when \( A \) wants to move the \( i \)th (in the order left after the partitioning) element from \( S' \) to location \( j \) of \( E \) we just exchange the \( i \)th element of \( S' \) with the \( j \)th element of \( S'' \).

(iv) After step (iii) we have \( S' \) sorted. Then we sort \( S'' \) recursively and we are done.

The algorithm is tail-recursive, hence we do not have to solve the stack-related problems we mentioned in Chapter 2. Since the sub-problems scale like the powers of 2, the optimality of the algorithm follows easily from the optimality of \( A \).

The set \( S'' \) with the placeholders is the internal buffer and its members are the buffer elements whereas the elements in \( S' \) are the active elements.

We could have used a merge-based approach that is dual to the distribution-based approach in Example 4.1. With the merge-based approach we begin directly with step (iii) and after the recursive step (iv), we merge the \( O(\log n) \) sorted sets from right to left (i.e. starting from the smallest sorted set) using an in-place, linear time, two-way merging algorithm.

It has to be noted that in this simple example we do not have to be able to distinguish between active and buffer elements at any moment during the execution of the algorithm. In the advanced applications of the internal buffering technique the need to distinguish between the two kind of elements becomes insurmountable. In fact, the differences between the two approaches are in the way the active elements can be distinguished from the buffer elements. In the distribution-based we can tell if an element is a buffer or an active one simply comparing it with \( p \) (the element used in the distribution). In the merging-based approach there is not a single way to do so. It all depends on the task that has to be executed with the help of the internal buffer. Usually, we can distinguish between the two kinds of elements encoding information about the distribution of active and buffer elements. The merge-based approach might seem more complicated but it becomes precious when comparisons cost too much, like when the input set is drawn from a multidimensional domain (see Chapter 8).

### 4.3.2 Moderate and adaptive internal buffering

It is worth to point out that the recursive approach used in Example 4.1 for the subdivision between active and buffer elements is very powerful but it cannot be
used for every problem. For example, the *generalized merging problem* (see Chapters 3 and 9) cannot be solved exploiting an internal buffer of size $O(n)$ like the one produced with the recursive approach. This is because the initial order of the buffer elements is completely disrupted after the use of the internal buffer. Therefore, the pre-sortedness of the input set that must be exploited in the generalized merging problem is irremediably lost.

Usually, a more moderate buffering approach is in order. The set $S$ of input elements is divided into two subsets $S'$ and $B$, where the latter is the buffer set and has cardinality $o(S)$. Then, $S'$ is conquered efficiently with the aid of $B$ that can be subsequently conquered with a sub-optimal method (its size has to be chosen with this objective in mind). Finally, a last merging step conquers the whole set $A$ easily.

Obviously the smaller the internal buffer must be, the harder the solution becomes. This is just the case when the stability of the algorithm is a concern, as in [Franceschini, 2005a] (Chapter 6). Let us recall that a sorting algorithm (or any other algorithm permuting a multiset) is stable if in the final sequence of input elements the relative order of the equal elements is the same they were before the execution of the algorithm. It is easy to understand the clash between the internal buffering technique and the stability constraint:

- On one hand, if we use an internal buffer that has a fixed size and does not depend on the number of the equal elements in the input sequence, we may have equal elements within the buffer. After the use of the internal buffer their relative order would be irreparably lost.

- On the other hand, even if succeeded in the very hard task of extracting one of the largest internal buffer containing only distinct elements, we would have to solve the problem using a buffer that could be very small.

In [Franceschini, 2005a] (see Chapter 6) that second, “adaptive” approach is used.

### 4.3.3 Internal buffering and implicit data structures

The internal buffering technique has never been used in the implicit data structures. The reason would seem obvious: the buffer elements are pushed around continuously during the execution of the algorithm and, for the computation, they are virtually indistinguishable from one another. This is terrible in an online setting like the one of a dictionary where, at any given moment, any element in the structure has to be searchable. In [Franceschini and Grossi, 2003c] (see Chapter 15) this belief is proven wrong and a sophisticated internal buffering technique is used to prove that worst case optimal upper bounds are possible for the implicit dictionary problem.
4.4 Sorted Order Maintaining Algorithms

4.4.1 A natural problem

We will have to face the following basic abstract problem many times in this work.

**Problem 4.1** We have a semi-dynamic set $S$ drawn from a totally ordered, possible infinite universe.

- $S$ is subject to repeated insertions.
- $S$ is laid out in a possibly unbounded sequence of slots $s_1s_2s_3\ldots$ that can contain one object of $S$ each.
- The objects in $S$ can be compared in $O(1)$ time and can be moved to a free slot in $z = \Omega(1)$ time.

We want to maintain the set $S$ (that is subject to insertions of new objects) so that

(i) $S$ is searchable in $O(\log|S|)$ time,

(ii) for any two $d', d'' \in S$ stored respectively in the slots $s_i$ and $s_j$, we have that $d' \leq d''$ iff $i \leq j$ and

(iii) if $s_r$ is the rightmost occupied slot then $r = O(|S|)$.

This problem was solved for the first in [Itai, Konheim, and Rodeh, 1981]. In this paper the authors give an algorithm that, for any sequence of $n$ insertions, solves the abstract problem in $O(n\log^2 n)$ time (i.e. each insertion costs $O(\log^2 n)$ time in amortized sense). Subsequently, a solution for a relaxed problem without condition (i) with the same aggregate cost of the one in [Itai, Konheim, and Rodeh, 1981] but with a worst case cost of $O(z \log^2 n)$ time for each insertion is presented in [Willard, 1992]. Recently an alternative solution has been proposed in [Bender, Demaine, and Farach-Colton, 2000]. They are all based on the same fundamental idea.

4.4.2 The basic solution

We will briefly show the solution inductively. Let us consider the set $S$ at a generic moment of its existence under the maintenance of the algorithm. Let us give the inductive hypotheses holding at that precise moment.

Let $n = |S|$, and $m$ be the size of the portion of the sequence of slots devoted by the algorithm to contain the objects. Let $m' \leq m$ be the index of the rightmost slot occupied by an object. By induction, let us assume that $m = O(n)$ and $m$ is a power of 2.
The sub-sequence of $m$ contiguous slots is seen as (implicitly) divided into frames of size $\log m$: the first frame is the sub-sequence of slots $s_1 s_2 \ldots s_{\log m - 1} s_{\log m}$, the second one is $s_{\log m + 1} s_{\log m + 2} \ldots s_{2 \log m - 1} s_{2 \log m}$ and so forth. The first slot of any frame is occupied by an object belonging to the frame (that is also the smallest object contained in the frame, since, by inductive hypothesis, all the objects are in sorted order at this moment).

The frames are associated with the leaves of a binary tree of height $h = O(\log \frac{m}{\log m}) = O(\log n)$. Each internal node $v$ of the tree is associated with a contiguous sub-sequence of slots denoted by $s(v)$ and defined in the following way.

- If an internal node is a parent of two leaves then its children are associated with two consecutive frames.

- If two internal nodes $v', v''$ are siblings then $s(v')$ and $s(v'')$ are consecutive sub-sequences and the sub-sequence $s(u)$ associated with their parent $u$ is the concatenation of $s(v')$ and $s(v'')$.

Each node $v$ of the tree is associated with its density $d(v)$ defined as the ratio between the number of objects contained in $s(v)$ and the size (in slots) of $s(v)$. Each node has a density threshold that is a function of its depth in the tree (the root is a depth 0 the leaves at $h$). Let $\tau_0 < 1$ be a real constant. The density threshold $\tau_k$ associated with any node with depth $k$ is

$$\tau_k = \tau_0 + k \frac{1 - \tau_0}{h}.$$ 

### 4.4.3 Maintaining the invariants

Holding those hypotheses, an object can be searched in $O(\log |S|)$ time in two simple steps. First, a binary search is run over the objects in the first slots of the frames. Then, the search ends with a linear scan into the frame identified in the first step.

For the maintaining of the invariants we have to use the density thresholds. Let suppose a new object $o'$ is inserted in the structure. We first execute a binary search for its frame in the sequence and insert $o'$ in it (in sorted order). If the frame $o'$ belongs to is full (that is, it contains $\log m$ objects after the insertion of $o'$), the threshold $t_h = 1$ for the densities of the leaves is not respected anymore and we have to redistribute the objects to fix the problem. This is done in two steps. First, we find the lowest ancestor (of the frame) $v$ respecting its density threshold. Then, we redistribute evenly the objects in $s(v)$. This restores the density constraints for all the descendants of $v$.

Using this simple scheme of density-based redistribution, the total cost of $n$ insertions is $O(n \log^2 n)$. This is very simple to prove. By hypothesis, we know that $d(v) < \tau_i$, where $i$ is the depth of $v$. After the redistribution the objects are evenly scattered in $s(v)$ and therefore, for any descendant $u$ of $v$ we have that

$$d(u) < \tau_i.$$
Hence, before one of the two children of \( v \) surpasses its threshold again, there have to be at least
\[
|s(u)| (\tau_{i+1} - \tau_i)
\]
insertions in \( s(u) \). Since we payed \( O(z |s(v)|) \) to redistribute the objects in \( s(v) \), we have an amortized cost per-level
\[
O \left( \frac{z}{1 - \tau_0} \right) = O( zh ) = O(z \log m).
\]
Since the levels of the tree are \( h \), we obtain the wanted amortized bound. As a last note, we point out that if during the redistribution phase, not even the root of the tree respect its density threshold then the space devoted to the structure \( (m) \) is doubled and the tree gets one more level and another root node.

This rebalancing scheme will be used in a variety of ways in this work. Of course we will have to use it either opportune modified and “implicitized” or with the help of opportune auxiliary techniques (e.g. the internal buffering and bit stealing to say the more obvious names) in order to respect the space optimality requirement. We will use also the scheme in [Bender, Demaine, and Farach-Colton, 2000] that basically add to the previous scheme another set of density thresholds in order to lower bound the density of any node. That allows to solve a problem that is more general than Problem 4.1. The new problem also considers the possibility of deletion operations and requires that the total cost of a generic sequence of insertion \textit{and} deletion operations is bounded.

We will use the worst case algorithm in [Willard, 1982, 1992] as well. This algorithm is very powerful but yet simple and elegant. This is important since we will use it mainly as a black-box similarly to what it was done in [Dietz and Sleator, 1987], without the need to modify the extremely sophisticated analysis in [Willard, 1982, 1992].

4.5 Searching a Set of Vectors

4.5.1 Hirschberg’s linear scan

The first work dealing with the problem of \textit{space optimal vector searching} is [Hirschberg, 1978]. In this paper a simple search algorithm was introduced. It has a linear complexity \( O(n + k) \) but it is space optimal and turned out to be a very useful tool in many subsequent works. Due to its linear bound its involvement in other results occurs mainly when a small subset, typically \( O(\log n) \), of the input set of vectors has to be searched easily and space optimally.

With this algorithm the vectors need to be in lexicographical sorted order. Since any component of any vector is supposed to be accessible in \( O(1) \) time, the sorted sequence of \( n \) vectors can be seen as matrix with \( k \) rows and \( n \) columns. The algorithm moves from a cell of the matrix into another starting from \( (1, 1) \) (the first component of the first vector):
Figure 4.1: An example of Hirschberg’s linear scan with *qpjoknlm* as searching element. Dark grey cells are mismatches while light grey ones are matches.

1. If the *current cell* of the matrix *does not match* with the *current component* of the search element, then proceed with the *next matrix cell to the right*.

2. Otherwise proceed with the *next matrix cell below*.

3. If the bottom of the matrix is reached *with a match*, verify if the *current column and the search element* are equal.

Figure 4.5.1 is an example of Hirschberg’s linear scan when the element to be searched is *qpjoknlm*.

### 4.5.2 Manber and Myers’ searching algorithm

Manber and Myers’ searching algorithm was introduced with the *suffix array* [Manber and Myers, 1993], but it can be applied also to a lexicographically sorted sequence \( x_1, x_2, \ldots, x_{n-1}, x_n \) of vectors of length \( k \).

The problem with the plain binary search in this setting is that at any step of the search we have to compare the searching vector and the current “pivot” in the sequence starting from the first component. That is because we cannot make any hypothesis about the common prefix between the previous pivot and the current one and therefore we cannot exploit the component comparisons we did in the previous step of the search.

The basic idea of Manber and Myers’ searching algorithm is to pre-compute some information about the sequence of vectors so that the binary search can go through without too much rescanning of the pivots. Anyone knows how the binary search operates. Initially we have the whole interval of elements \([x_1, x_n]\) and we examine the median \( x_M \) element of the interval. If the searching element is less than or equal
to $x_M$ the search continues recursively into the interval $[x_1, x_M]$, otherwise we recur into $[x_{M+1}, x_1]$.

The idea of Manber and Myers is to precompute the longest common prefixes $lcp(x_L, x_M)$ and $lcp(x_M, x_R)$ for any of the $O(t)$ intervals $[L \ldots M \ldots R]$ (where $M$ is the middle point) induced by the binary search process. Let $[L \ldots M \ldots R]$ be the current interval during the binary search of a vector $x$. During the search the algorithm maintains the two values $lcp_R = lcp(x_R, x)$ and $lcp_L = lcp(x, x_L))$. For the outcome of the comparison of $x$ with $x_M$ there are two groups of three cases each depending on whether $\max(lcp_L, lcp_R) = lcp(x_L, x)$ or $\max(lcp_L, lcp_R) = lcp(x_R, x)$.

They are symmetric, so let’s assume $\max(lcp_L, lcp_R) = lcp(x_L, x)$ and let us denote it with $m$:

1. if $m < lcp(x_L, x_M)$ then $x > x_M$ and the search proceeds with $[x_M, x_R]$.
2. if $m = lcp(x_L, x_M)$, $x_M$ and $x$ are compared starting from component $m + 1$ and the next interval to proceed with is decided when the first mismatch is found (if any).
3. if $m > lcp(x_L, x_M)$, then $x < x_M$ and the search proceeds with $[x_L, x_M]$.

That kind of algorithm may seem inherently static because, apparently, all the information that guides the searches may have to be totally re-computed if a new element has to be inserted into the sorted sequence. Actually that is not the case and the ideas of Manber and Myers’ algorithm can be applied to obtain dynamic searching algorithms (dictionaries).
Part II

Sorting
“Sorting algorithms make an interesting case study of how to attack computer programming problems in general.”

— Donald E. Knuth,
The Art of Computer Programming, Vol. 3: Sorting and Searching
Chapter 5

Checkmate in $O(n)$ Moves

Abstract

In this chapter, we present an in-place algorithm for sorting a sequence of size $n$ that performs, in the worst case, at most $O(n \log n)$ element comparisons and $O(n)$ element moves.

The presentation in this chapter is based on the paper [Franceschini and Geffert, 2003] (FOCS 2003) that solved a long-standing open problem, stated explicitly, e.g., in [Munro and Raman, 1992], of whether there exists a sorting algorithm that matches the asymptotic lower bounds on all computational resources simultaneously.

The algorithm we introduce in this chapter is unstable (i.e., the initial relative order of equal elements is lost after the execution). Maintaining optimality and stability in space optimal settings has been recognized as a very difficult task. In Chapter 6, we will propose a stable algorithm using a completely different approach which is much more complex. Nevertheless, in the case of the unstable algorithm we will be able to provide a precise analysis down to the constant factors.

5.1 The Problem

As we saw in Chapter 2, in the derivatives of the Random Access Machine model it is natural to measure the efficiency of a sorting algorithm (as for any other algorithm) with three metrics: the number of comparisons it requires, the number of element moves it performs and the number of auxiliary locations it uses, besides the ones strictly necessary for the input elements. It is well-known that a comparison-based algorithm must perform, in the worst case, at least the following number of comparisons to sort an input sequence consisting of $n$ elements:

\[
\lceil \log n! \rceil \geq n \log n - n \log e \approx n \log n - 1.443n
\]
By [Munro and Raman, 1996a], we know that the corresponding lower bound for element movements is \( \frac{3}{2n} \). A natural question concerns the existence of a sorting algorithm that is optimal with respect to the three main complexity measures at the same time. So far no sorting with this unique characteristic was known. Nevertheless, given the importance of the sorting problem much effort has been devoted to this topic with the following algorithms as the state of the art in their respective categories.

**Comparison and space optimal sorting.** In ’99 a \((\log n / \log \log n)\)-way in-place mergesort was introduced in [Katajainen and Pasanen, 1999]. This algorithm uses \( n \log n + O(n \log \log n) \) comparisons, \( O(1) \) auxiliary storage, and only \( O(n \log n / \log \log n) \) element moves. It uses a compressed selection tree to conduct the multiway merging process. The information about the selection tree is compressed into bits of \((\log n)\)-bit index variables by complicated bitwise operations, which increases, among others, the number of arithmetic operations. Therefore, the algorithm is mainly of theoretical interest; it is the first member of the comparison and space optimal family breaking the bound \( \Omega(n \log n) \) for the number of moves.

**Move and space optimal sorting.** The generalized Heapsort [Munro and Raman, 1992] is the leader in here. It is based on a heap in which internal nodes have \( \lceil n^{1/l} \rceil \) children, for a fixed integer \( l \). The corresponding heap tree is thus of constant height, which results in an algorithm sorting with \( O(n) \) moves, \( O(1) \) space, and \( O(n^{1+\varepsilon}) \) comparisons.

**Comparison and move optimal sorting.** A variant of samplesort in [Munro and Raman, 1992] sorts with \( O(n \log n) \) comparisons, \( O(n) \) moves and storage requirements of \( O(n^\varepsilon) \) auxiliary locations.

The resource bounds for some previous sorting algorithms can be seen in table 5.1.

### 5.1.1 The result

It was generally conjectured, for many years, that an algorithm matching simultaneously the asymptotic lower bounds on all above computational resources does not exist. For example, in [Raman, 1991], it was proved that the algorithm with \( O(n^{1+\varepsilon}) \) comparisons using generalized heaps is optimal among a certain restricted family of space optimal sorting algorithms performing \( O(n) \) moves. It was hoped that, by generalizing from a restricted computational model to all comparison-based algorithms, we could get a higher trade-off among comparisons, moves, and auxiliary space.

The result we will present in this chapter contradicts the above conjecture and closes a long-standing open problem. This presentation is based on the papers
5.1. THE PROBLEM

<table>
<thead>
<tr>
<th></th>
<th>Compar.</th>
<th>Moves</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lower bounds</strong></td>
<td>$[\log n!]$</td>
<td>$[3/2n]$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Comparisons-Space</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertsort</td>
<td>$\log n! + n$</td>
<td>$\Omega(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$2n \log n$</td>
<td>$n \log n + \ldots$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$\frac{\log n}{\log \log n}$-Mergesort</td>
<td>$n \log n + \ldots$</td>
<td>$O\left(n^{\frac{\log n}{\log \log n}}\right)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Moves-Space</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectsort</td>
<td>$\Omega(n^2)$</td>
<td>$2n - 1$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$\ell$-Heapsort</td>
<td>$O\left(n^{1+\frac{\ell}{\ell-1}}\right)$</td>
<td>$O(\ln n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Comparisons-Moves</strong></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tablesort</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>$\Omega(n)$</td>
</tr>
<tr>
<td>$n^{1/\ell}$-way mergesort</td>
<td>$n \log n + \ldots$</td>
<td>$O(\ln n)$</td>
<td>$O(n^{1/\ell})$</td>
</tr>
</tbody>
</table>

Table 5.1: Resource bounds for some known sorting algorithms.


The algorithm we introduce in this chapter is *unstable*, that is the initial relative order of equal elements is lost after the execution. Maintaining optimality and stability in space optimal settings has been recognized as a very difficult task especially when the internal buffering technique has to be used to achieve the optimality (see Chapter 4). In Chapter 6, we will propose a stable algorithm using a completely different approach which is much more complex. Nevertheless, in the case of this unstable algorithm we will be able to provide a precise analysis down to the constant factors. For this reason we will have to be really precise in the definition of the algorithm.

### 5.1.2 The algorithm in a nutshell

Using an evenly distributed sample $a_1, \ldots, a_f$ of size $\Theta(n/(\log n)^4)$, we split the elements into segments $\sigma_0, \sigma_1, \ldots, \sigma_f$, of length $\Theta((\log n)^4)$ each, so that elements in $\sigma_k$ satisfy $a_k \leq a \leq a_{k+1}$. The sorted sequence is obtained by forming $\sigma_0', a_1, \sigma_1', \ldots, a_f, \sigma_f'$, where $\sigma_k'$ denotes $\sigma_k$ in sorted order.

To sort $\sigma_k$, we use a modified Heapsort, with internal nodes having $\Theta((\log n)^{4/5})$ sons, which results in a constant number of moves per each element extracted from the heap.

Since an evenly distributed sample is hard to find, we have to grow it dynamically with a method that resembles the way the B-Trees are maintained balanced. When some $\sigma_k$ becomes too large, we halve it into two segments of equal length, and insert the median in the sample. To minimize moves required for insertions in the sample,
it is sparsely distributed in a block of size $\Theta(n/(\log n)^3)$, not losing advantage of a quick binary search.

When a local "crowd" of elements appear somewhere, it is eliminated by redistributing the sample more evenly in its surroundings. Of course that should not happen too often in order to amortize the cost of the redistribution. To avoid the corresponding segment movement, only pointers (which we do not have, of course; we are going to have to encode them somewhere) connecting $a_k$'s with $\sigma_k$'s are moved, the segments stay motionless in a separate workspace (to be simulated with the internal buffering technique).

However, we do not have a buffer of size $3n$, required for the sample and the segments, nor $P \approx \Theta(n/(\log n)^2)$ bits, for pointers. The bits are "created" at the very beginning by a modified Heapsort, collecting the smallest and the largest $P$ elements to blocks $\Pi_L$ and $\Pi_R$, which leaves a block $\mathcal{A}'$ in between. Then the $j$th bit can be encoded by swapping the $j$th element in $\Pi_L$ with the $j$th element in $\Pi_R$.

To simulate a buffer for sorting the block $\mathcal{A}'$ of length $n'$, we select the element $b_5^5$ of rank $[n'/4]$ and partition $\mathcal{A}'$ into blocks $A_<$ and $B_>$, using $b_5^5$ as a pivot. Then sort $A_<$, using $B_>$ as an empty buffer. (We can test if a given location contains a buffer element, by a single comparison with $b_5^5$. Before an active element is moved, one buffer element escapes to the current location of the hole). After sorting $A_<$ we iterate, focusing on $B_>$ as a new block $\mathcal{A}'$. Since in this case, we do not care about stability, we are allowed to use full-power internal buffering, see Chapter 4. After $O(\log n)$ iterations, we are done.

5.2 Sorting with an Additional Memory

Before presenting the space optimal algorithm, we shall concentrate on a simpler task. We are going to sort a given contiguous block $A$, consisting of $m$ elements, using only $O(m \log m)$ comparisons and $O(m)$ element moves. As some additional resources, we are given a buffer memory, of size at least $3m - 1$, that can be used as a temporary workspace, and a pointer memory, capable of containing at least $\lfloor 4m/(\log m)^2 \rfloor$ bits. After we have defined and precisely analyzed an algorithm for this simpler case, we will show how to remove these auxiliary resources without increasing too much the constant factors in the complexity bounds.

To let the elements move, we also have a hole, that is, one location, the content of which can be modified without destroying any element. An assignment $a_j := a_i$ transports not only one element from the location $i$ to $j$, but also the hole from $j$ to $i$. At the very beginning, the hole is in a single extra location, besides the given input sequence. This technique is useful when the number of moves is to be minimized down to the constant factor.
5.2.1 Buffer memory

The buffer memory forms a separate contiguous block $B$, initially consisting of at least $3m - 1$ buffer elements. Like in the distribution-based approach in Chapter 4, all buffer elements are greater than or equal to a given buffer separator $b^\circ$, placed in an extra location, while all elements in $A$ are strictly smaller than $b^\circ$.

During the computation, the elements of $A$ and $B$ are mixed up. However, by a single comparison with $b^\circ$, we can test whether any given location contains a buffer element, or an active element, a subject of sorting, placed originally in $A$ (see Chapter 4).

The buffer memory $B$ consists of two parts. First, there is a low level segment memory, a sequence of segments allocated dynamically from the right end of $B$ and growing to the left, as the computation demands. All allocated segments are of the same fixed length. Second, there is a fixed high level frame memory, placed at the left end of $B$.

5.2.2 Structure of the segment memory

All segments are of a fixed length $s$, where

$$
s = \begin{cases} 
[(\log m)^4] & \text{so that } s \text{ is odd.} \\
[(\log m)^4] + 1 & 
\end{cases}
$$

During the computation, the number of active segments never exceeds $s_w$, defined by

$$
s_w = \left\lfloor \frac{2m}{s} \right\rfloor \leq \frac{2m}{(\log m)^4},
$$

and hence the size of workspace reserved for the segment memory is bounded by

$$
S = s_w s \leq 2m.
$$

Here we assume that $m$ is “sufficiently large,” such that $s \leq m$, and hence $s_w \geq 2$. We shall later discuss how to handle a block $A$ that is “short.”

Initially, all segments are free, containing buffer elements only. The algorithm keeps the starting position of the last segment that has been allocated in a global index variable $\bar{s}$. Initially, $\bar{s}$ points to the right end of the buffer memory $B$. To allocate a new segment, the procedure simply performs the operation $\bar{s} := \bar{s} - s$, and returns the new value of $\bar{s}$ as the starting position of the new segment. Immediately after allocation, some $\lfloor s/2 \rfloor$ active elements (smaller than $b^\circ$) are transported to the first $\lfloor s/2 \rfloor$ positions of the new segment. The corresponding buffer elements are saved in the locations released by the active elements. From this point forward, the segment becomes active.
In general, the structure of an active segment is

\[ c_1 \ldots c_h b_{h+1} \ldots b_s, \]

where \( c_1 \ldots c_h \) are active elements stored in the segment, while \( b_{h+1} \ldots b_s \) are some buffer elements. The value of \( h \) is kept between \( \lfloor s/2 \rfloor \) and \( s-1 \), so that at least one half (roughly) of elements in each active segment is active, and still there is a room for storing one more active element. Neither \( c_1 \ldots c_h \) nor \( b_{h+1} \ldots b_s \) are sorted. In addition, the algorithm does not keep any information about the boundary \( h \) separating active and buffer elements, if the segment is not being manipulated at the present moment. However, since all active elements are strictly smaller than \( b^2 \) and all buffer elements are greater than or equal to \( b^s \), we can quickly determine the number of active elements in any given segment, using a binary search with \( b^2 \) over the \( s \) locations of the segment, which costs only \( 1 + [\log s] \leq O(\log \log m) \) comparisons, by (5.1).

### 5.2.3 Structure of the frame memory

The frame memory, placed at the left end of \( B \), consists of \( r_\# \) so-called frame blocks, each of length \( r \), where

\[
    r = 1 + \left\lfloor \log \frac{2^m}{s} \right\rfloor \leq 2 + \log(2^m) = \log m,
\]

\[
    r_\# = 2^{r-1} = 2^{\left\lfloor \log \frac{2^m}{s} \right\rfloor} \leq 2 \cdot 2m/s \leq \frac{4m}{(\log m)^3}, \tag{5.4}
\]

using (5.1) and \( m \geq 4 \). That is, the frame memory is of total length

\[
    R = r_\# r \leq \frac{4m}{(\log m)^3}. \tag{5.5}
\]

Using (5.3) and \( m \geq 4 \), we get that the total space requirements for the segment and frame memories do not exceed the size of the buffer \( B \), since

\[
    R + S \leq \frac{4m}{(\log m)^3} + 2m \leq 3m - 1.
\]

A frame block is either free, containing buffer elements only, or it is active, containing some active elements followed by some buffer elements. Initially, all frame blocks are free. During the computation, active frame blocks are concentrated in a contiguous left part of the frame, followed by some free frame blocks in the right part. However, there are some important differences from the segment memory structure:
The active elements, forming a left part of a frame block, are in sorted order. So are the active frame blocks, forming a left part of the frame memory. More precisely, let $a_1, a_2, \ldots, a_f$ denote the sequence of all active elements stored in the frame memory, obtained by reading active elements from left to right, ignoring buffer elements and frame block boundaries. Then $a_1, a_2, \ldots, a_f$ is a sorted sequence of elements. Consequently, a sub-sequence of these, stored in the first (leftmost) positions of active frame blocks, denoted here by $a_{i_1}, a_{i_2}, \ldots, a_{i_g}$, must also be sorted. Here $f$ denotes the total number of active elements in the frame, while $g$ the number of active frame blocks, at the given moment. Similarly, $a_{i_j}a_{i_{j+1}}a_{i_{j+2}}\ldots a_{i_{j+1}-1}$, the sequence of active elements stored in the $j$th frame block, is also sorted.

The number of active elements in an active frame block can range between 1 and $r-1$. That is, we keep room for potential storing of one more active element in each active frame block, but we do not care about a sparse distribution of active elements in the frame. The only restriction follows from the fact that there are no free blocks in between some active blocks.

### 5.2.4 Relationship between the frame and segments

Each active element in the frame memory, i.e., each of the elements $a_1, a_2, \ldots, a_f$, has an associated segment $\sigma_1, \sigma_2, \ldots, \sigma_f$ in the segment memory. The segment $\sigma_k$, for $k \text{ ranging } 1 \text{ and } f$, contains some active elements satisfying $a_k \leq a \leq a_{k+1}$, taken from $A$ and stored in the structure so far. The active elements satisfying $a_f \leq a$ are stored in $\sigma_f$, similarly, those satisfying $a \leq a_1$ are stored in a special segment $\sigma_0$. Note that the segment $\sigma_0$ has no “parent” in the sequence $a_1, a_2, \ldots, a_f$, that is, no frame element to be associated with. Chronologically, $\sigma_0$ is the first active segment that has been allocated. If $f = 0$, i.e., no active elements have been stored in the frame yet, all active elements are transported from $A$ to $\sigma_0$.

Note also that (in order to keep the number of active elements in active segments balanced) we do allow some elements equal to $a_k$ be stored both in $\sigma_{k-1}$ and in $\sigma_k$. In general, we may even have $a_k = a_{k+1} = \ldots = a_{k'},$ for some $k < k'$. Then elements equal to $a_k$ may be found in any of the segments $\sigma_{k-1}, \sigma_{k}, \ldots, \sigma_{k'}$. However, the algorithm tries to store each “new” active element $a$, coming from $A$, in the leftmost segment that can be used at the moment, i.e., it searches for $k$ satisfying $a_k < a \leq a_{k+1}$.

Recall that we also maintain the invariant that each active segment contains at least $\lceil s/2 \rceil$ active elements. Thus, if the frame contains $f$ active elements at the given moment, namely, $a_1, a_2, \ldots, a_f$, for some $f \geq 1$, the total number of active elements, stored both in the frame and the segments $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_f$, is at least

$$f + (f+1) \cdot \left\lfloor \frac{s}{2} \right\rfloor.$$
Now, using the fact that \( s \) is odd, by (5.1), we get that this number is at least

\[
f + (f+1) \cdot \left(\frac{f}{2} - \frac{1}{2}\right) = (f+1) \cdot \frac{f}{2} + \left(\frac{f}{2} - \frac{1}{2}\right) \geq (f+1) \cdot \frac{f}{2}.
\]

However, the total number of all active elements is exactly equal to \( m \), which gives \( m \geq (f+1) \cdot \frac{f}{2} \), and hence also \( f+1 \leq 2m/s \). But \( f+1 \), the number of active segments, is an integer number, which gives that \( f+1 \leq \lfloor 2m/s \rfloor \). Therefore, using (5.2) and (5.4),

\[
f+1 \leq \left\lfloor \frac{2m}{r} \right\rfloor = s_s, \tag{5.6}
\]

\[
f \leq \left\lfloor \frac{2m}{r} \right\rfloor \leq 2^\left\lceil \log(2^{\frac{m}{r}}) \right\rceil = r_s.
\]

(The argument has used the assumption that \( f \geq 1 \). However, (5.6) is trivial for \( f = 0 \), since \( r_s \geq s_s \geq 2 \), if \( m \) is sufficiently large).

As a consequence, we get that \( f+1 \), the number of active segments, does not exceed \( s_s \), the capacity of the segment memory. Second, \( f \), the number of active elements in the frame, will never exceed \( r_s \), the total number of blocks in the frame, and hence there is enough room to store all active frame elements, even if each active frame block contained only a single element of the sequence \( a_1, a_2, \ldots, a_f \).

### 5.2.5 Structure of the pointer memory

The relative order of active frame elements in the sequence \( a_1, a_2, \ldots, a_f \) does not correspond to the chronological order, in which the segments \( \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_f \) are allocated in the segment memory. Therefore, with each element position in the frame, we associate a pointer to the starting position of corresponding segment. More precisely, if the frame is viewed as a single contiguous zone of elements \( x_1 \ldots x_R \) (ignoring boundaries between the frame blocks), then the corresponding zone of pointers is \( \pi_1 \ldots \pi_R \). If, for some \( \ell \), the element \( x_\ell \) is a buffer element, then \( \pi_\ell = 0 \), which represents a NIL pointer. Conversely, if \( x_\ell \) is an active element belonging to the sequence \( a_1, a_2, \ldots, a_f \), then the value of \( \pi_\ell \) represents the starting position of the segment associated with \( x_\ell \). (The pointer \( \pi_0 \) to the segment \( \sigma_0 \), having no “parent” in the frame, is stored separately, in a global index variable).

Since there are at most \( s_s \) segments, all of equal length, a pointer to a segment can be represented by an integer value ranging between 0 and \( s_s = \lfloor 2m/s \rfloor \leq m/2 \), using (5.2). Thus, a single pointer can be represented by a block of \( p \) bits, where

\[
p = 1 + \lfloor \log s_s \rfloor \leq \log m. \tag{5.7}
\]
The number of pointers is clearly equal to $R$, the total size of the frame. Therefore,

$$p_\ast = R.$$  

Thus, the pointer memory can be viewed as a contiguous sequence consisting of $p_\ast$ bit blocks, of $p$ bits each, and hence, by (5.5), its total length is at most

$$P = p_\ast p = R \cdot p \leq \frac{4m}{(\log m)^2}, \quad (5.8)$$

using also the fact that $P$ must be an integer number.

Since an in-place algorithm can store only a limited amount of information in index variables, the pointer memory is actually simulated by two separate contiguous blocks $\Pi_L$ and $\Pi_R$, each containing at least $\lfloor 4m/(\log m)^2 \rfloor$ elements. Initially, $\Pi_L$ and $\Pi_R$ are sorted, and the largest (rightmost) element in $\Pi_L$ is strictly smaller than the smallest (leftmost) element in $\Pi_R$. This allows us to encode the value of the $j$th bit, for any $j$ ranging between 1 and $\lfloor 4m/(\log m)^2 \rfloor$, by swapping the $j$th element of $\Pi_L$ with the $j$th element of $\Pi_R$.

Testing the value of the $j$th bit is thus equivalent to comparing the relative order of the corresponding elements in $\Pi_L$ and $\Pi_R$, which costs only a single comparison. Setting a single bit value requires a single comparison and, optionally, a single swap of two elements, i.e., 3 element moves. The initial distribution of elements in $\Pi_L$ and $\Pi_R$ represents all $\lfloor 4m/(\log m)^2 \rfloor$ bits cleared to zero.

### 5.2.6 Inserting elements in the structure

The procedure sorting the block $A$ works in two phases. In the first phase, the procedure takes, one after another, all $m$ active elements from $A$ and inserts them in the structure described above. The procedure also exchanges some buffer elements from $B$, and keeps the structure “balanced.” In the second phase, all active elements are transported back to $A$, this time in sorted order.

For each active element $a$ in $A$, we find a segment, among $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_f$, where this element should go.

1. By the use of a binary search with the given element $a$ over $a_{i_2}, \ldots, a_{i_g}$, that is, over the leftmost locations in the active frame blocks, find the “proper” frame block for the element $a$, i.e., the index $j$ satisfying $a_{i_j} < a \leq a_{i_{j+1}}$. Note that the element $a_{i_t}$ is excluded from the range of the binary search. If $a \leq a_{i_2}$, the binary search will return $j = 1$, i.e., the first frame block. Similarly, for $a_{i_g} < a$, the binary search returns $j = g$, i.e., the last frame block. If $g < 2$, we can go directly to the first (and only) active frame block without using any binary search, that is, $j := 1$.

2. By the use of a binary search with the given element $a$ over the $r$ locations in the $j$th active frame block, find the “proper” active frame element for the
element \( a \), i.e., the index \( k \) satisfying \( a_k < a \leq a_{k+1} \). Note that, since \( a_{i_j} < a \leq a_{i_{j+1}} \), the elements \( a_k \) and \( a_{k+1} \) are between \( a_{i_j} \) and \( a_{i_{j+1}} \) in the sequence \( a_1, a_2, \ldots, a_f \) of all frame elements, not excluding the possibility that \( a_{i_j} = a_k \), and/or \( a_{k+1} = a_{i_{j+1}} \). Recall that the \( j \)th active frame block begins with the active elements \( a_{i_j} a_{i_{j+1}} a_{i_{j+2}} \ldots a_{i_{j+1}-1} \), followed by some buffer elements, to fill up the room, so that the length of the block is exactly equal to \( r \). These buffer elements are not sorted, however, they are all greater than or equal to \( b^2 \), the smallest buffer element. On the other hand, the element \( a \), being active, is strictly smaller than \( b^2 \). This allows us to use the binary search with the given \( a \) in the standard way, which returns the index \( k \) satisfying \( a_k < a \leq a_{k+1} \). For \( a_{i_{j+1}-1} < a \), the binary search returns correctly \( k = i_{j+1} - 1 \). If \( j = 1 \), that is, if we are in the first frame block, the binary search may end up with \( k = 0 \), indicating that \( a \leq a_1 = a_i \).

3. Let the active frame element \( a_k \), satisfying \( a_k < a \leq a_{k+1} \), be placed in a position \( \ell \) of the frame memory, that is, \( a_k = x_\ell \). (For \( k = 0 \), we take \( \ell := 0 \)). Then read the information from \( \pi_\ell \) in the pointer memory and compute the starting position of the segment \( \sigma_k \). This segment contains elements ranging between \( a_k \) and \( a_{k+1} \). If \( k = 0 \), i.e., the element \( a \) should go to \( \sigma_0 \), the starting position of the segment is obtained from a separate global index variable.

4. By the use of a binary search with the buffer separator \( b^2 \) over the \( s \) locations in the current segment, find the boundary \( h \) dividing the segment into two parts, namely, \( c_1 \ldots c_h \), the active elements stored in the segment, and \( b_{h+1} \ldots b_s \), some buffer elements, filling up the room.

5. Save the buffer element \( b_{h+1} \) aside, to the current location of the hole, and, after that, store the given element \( a \) in the segment. If \( h+1 < s \), we are ready to insert the next element from \( A \). However, if \( h+1 = s \), the current segment cannot absorb any more elements. Therefore, if the segment has become full, we call a procedure “rebalancing” the structure before trying to store the next element. This procedure will be described later, in Section 5.2.9.

The above process is repeated until all \( m \) active elements have been inserted in the structure.

Initially, the procedure allocates the segment \( \sigma_0 \), and stores the first \( s-1 \) active elements directly in \( \sigma_0 \), without travelling via the frame. The number of moves for these elements is the same as in the standard case, i.e., two moves per each inserted element.

Let us now determine the standard cost of inserting a single element.

**Lemma 5.1** If we exclude the costs of rebalancing, the insertion of \( m \) elements into the structure requires \( 2m \log m + O(m \log \log m) \) comparisons and \( 2m \) moves.
Proof: The binary search looking for a proper frame block inspects a range consisting of \(g-1 < r\) elements, and hence it performs at most \(1 + \lfloor \log r \rfloor \leq \log m\) comparisons, by (5.4). The second binary search, looking for a proper active element within the given frame block, inspects a range of \(r\) elements, performing at most \(1 + \lfloor \log r \rfloor \leq O(\log \log m)\) comparisons, using (5.4). Reading the value encoded in the pointer \(\pi_t\) requires \(p \leq \log m\) element comparisons, by (5.7). The binary search with \(b^\circ\) over the \(s\) locations in the current segment uses \(1 + \lfloor \log s \rfloor \leq O(\log \log m)\) comparisons, by (5.1). Finally, saving one buffer element and transporting the element \(a\) to the current segment can be performed with 2 element moves. However, these costs do not include rebalancing. Since \(m\) elements are inserted this way, we proved the thesis.\(\square\)

5.2.7 Extracting in sorted order — frame level

In the second phase, the active elements are transported back to \(A\), in sorted order. Let \(f_m\) denote the maximal value of \(f\), corresponding to the number of active elements in the frame at the moment when the last active element has been stored in the structure. Thus, the frame memory contains the sorted sequence of active elements \(a_1, a_2, \ldots, a_{f_m}\), intertwined with some buffer elements, so the total size of the frame is \(R\), consisting of elements \(x_1 \ldots x_R\). Then we have active elements in the segments \(\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{f_m}\), with \(\sigma_k\) containing active elements that satisfy \(a_k \leq a \leq a_{k+1}\). Thus, to produce the sorted order of all active elements, it is sufficient to move, back to \(A\), the sequence \(\sigma'_0, a_1, \sigma'_1, a_2, \sigma'_2, \ldots, a_{f_m}, \sigma'_{f_m}\), where \(\sigma'_k\) denotes the block of sorted active elements contained in \(\sigma_k\).

The procedure begins with moving the block \(\sigma'_k\) to \(A\). (We shall return to the problem of sorting a given segment \(\sigma_k\) below, in Section 5.2.8.) Then, in a loop iterated for \(\ell = 1, \ldots, R\), check whether \(x_\ell\) is an active element. This requires only a single comparison, comparing \(x_\ell\) with \(b^\circ\). If \(x_\ell\) is a buffer element, it is skipped, we can go to the next element in the frame.

If \(x_\ell\) is an active element, i.e., \(x_\ell = a_k\), for some \(k\), the following steps are executed:

1. The procedure saves the leftmost buffer element, not moved yet from the output block \(A\), in the current location of the hole and, after that, moves \(x_\ell = a_k\) to \(A\). (The first free position in \(A\), i.e., the position of the leftmost buffer element, is kept in a separate global index variable, and incremented each time a new active element is transported back to \(A\).

2. Then we read the value encoded in the pointer \(\pi_\ell\) and compute the starting position of the segment \(\sigma_k\).

3. After that, we move all active elements contained in \(\sigma_k\) to \(A\), in sorted order, by the procedure presented in Section 5.2.8.

Before showing how the segment \(\sigma_k\) can be sorted, let us derive computational costs of the whole procedure above, not including the cost of sorting \(\sigma_k\).
Lemma 5.2 If we exclude the costs of sorting the segments, the extraction of the elements in sorted order requires \( O(m/(\log m)^3) \) comparisons and \( O(m/(\log m)^4) \) moves.

Proof: Testing whether \( x_\ell \) is an active element, for \( \ell = 1, \ldots, R \), requires \( R \leq O(m/(\log m)^3) \) comparisons, by (5.5). Transporting \( x_\ell = a_k \) to \( A \) requires only \( 2f_m \) element moves in total, since only active elements are moved. This gives \( 2f_m \leq 2r_* \leq O(m/(\log m)^4) \) element moves, by (5.6) and (5.4). Reading the values of \( f_m \) pointers, of length \( p \) bits each, can be done with \( f_m p \leq r_* p \leq O(m/(\log m)^3) \) comparisons, using (5.6), (5.4), and (5.7). Summing up, we have the thesis. \( \square \)

5.2.8 Extracting in sorted order—segment level

Now we can describe the routine extracting, in sorted order, all active elements contained in the given segment \( \sigma_k \). Let \( h_k \) denote the number of active elements in \( \sigma_k \). Clearly, using (5.1) we know that

\[
h_k \leq s \leq \lfloor (\log m)^4 \rfloor + 1.
\]

Initially, the routine determines the value of \( h_k \) by the use of a binary search with \( b^2 \) over the \( s \) locations of the segment. This costs \( 1 + \lfloor \log s \rfloor \leq O(\log \log m) \) comparisons.

After that, the routine uses a generalized version of Heapsort, which in turn uses a modified heap-like structure, with

\[
t = \lfloor (\log m)^4/5 \rfloor
\]

root nodes (instead of a single root node), and with internal nodes having \( t \) sons (instead of two sons, see figure 5.1 for an example with \( t = 3 \)). More precisely, we organize \( c_1 \ldots c_{h_k} \), the active elements contained in the segment, into the implicit structure with the following properties:

**Shape of the heap.** First, the father of the node \( c_e \) is the node \( c_{e'} \), where \( e' = \lfloor (e-1)/t \rfloor \), provided that \( e' \geq 1 \). If \( e' < 1 \), then \( c_e \) is one of the root nodes.
This implies that the heap has \( t \) roots, and that the sons of \( c_e \) are the nodes \( c_{te+1}, c_{te+2}, \ldots, c_{te+t} \). If, for some \( e \) and \( d < t \), we have \( te + d = h_k \), the corresponding node \( c_e \) has only \( d \) sons, instead of \( t \). A leaf is a node \( c_e \) without any sons, that is, with \( te \geq h_k \).

**Height of the heap.** Second, the heap does not have more than five levels, since, by travelling to a root from \( c_{h_k} \), we get

\[
\begin{align*}
h^{(1)} &= \left\lfloor \frac{h_k - 1}{t} \right\rfloor < \frac{h_k}{t}, \\
h^{(2)} &= \left\lfloor \frac{h_k}{t} \right\rfloor < \frac{h_k}{t^2}, \\
h^{(3)} &= \left\lfloor \frac{h_k}{t^2} \right\rfloor < \frac{h_k}{t^3}, \\
h^{(4)} &= \left\lfloor \frac{h_k}{t^3} \right\rfloor < \frac{h_k}{t^4}, \\
h^{(5)} &= \left\lfloor \frac{h_k}{t^4} \right\rfloor \leq \frac{h_k}{t^5} - \frac{1}{t} < \frac{h_k}{t^5} - \frac{1}{t} \leq \frac{s}{t^5} = \frac{s}{t^5} - \frac{1}{t^5}.
\end{align*}
\]

If we had \( 1 \leq h^{(5)} \), then \( 1 < s/t^5 - 1/t^5 \), and hence also \( t^5 < s - 1 \). Now, using \( t = \left\lceil (\log m)^{4/5} \right\rceil \) and \( s \leq \left\lceil (\log m)^{4/5} \right\rceil + 1 \), by (5.1), we would obtain \( \left\lceil (\log m)^{4/5} \right\rceil < \left\lceil (\log m)^{4/5} \right\rceil \), which is a contradiction. To see this, note that, for each real \( x > 0 \), \( \left\lceil x^{4/5} \right\rceil \geq x^4 \). But \( \left\lceil x^{4/5} \right\rceil \) is an integer number, and hence \( \left\lceil x^{4/5} \right\rceil \geq \left\lceil x^4 \right\rceil \).

**Heap property.** Third, if a node contains an active element, then this element is not greater than any of its sons. Note that we do not care about sons of a node containing a buffer element. (Initially, there are no buffer elements in the heap. However, when some active elements have been extracted, buffer elements will fill up the holes).

This heap property is established in the standard way: For \( e = \left\lfloor (h_k - 1)/t \right\rfloor, \ldots, 1 \), establish this property in the positions \( e, \ldots, h_k \). This only requires to determine whether \( c_e \) is not greater than the smallest of its sons and, if necessary, swap the smallest son with \( c_e \). Processing a single node this way costs \( t \) comparisons and 3 element moves. After that, the heap property is re-established for the son just swapped in the same way. This may activate a further walk, up to some leaf.

Taking into account that there are \( h^{(1)} \) nodes with paths of lengths 1, 2, 3, or 4 (starting from the given node and ending in a leaf), \( h^{(2)} \) nodes with paths of lengths 2, 3, or 4, \( h^{(3)} \) nodes with paths of lengths 3 or 4, and \( h^{(4)} \) nodes with paths of length 4, we get that building the heap costs \( t \sum_{i=1}^{4} h^{(i)} < 2h_k \) comparisons and \( 3 \sum_{i=1}^{4} h^{(i)} < 6h_k/(\log m)^{4/5} \) moves.
After building the heap, the routine transports, \( h_k \) times, the smallest element from the heap to the output block \( A \). Here the moves are organized as follows.

1. Save the leftmost buffer element, not moved yet from \( A \), in the current location of the hole.

2. Find the smallest element, placed in one of the \( t \) roots, and move this element to \( A \).

3. Find the smallest element among the \( t \) sons of this root, and move this element to the node corresponding to its father.

Iterating this process at most five times, we end up with a hole in some leaf. Now, we are done. The hole in the leaf will be filled up by a buffer element in the future, as a side effect. (Usually, in the next iteration, extracting the next smallest element from the heap).

Thus, unlike in the standard version of Heapsort, the size of the heap does not shrink but, rather, some new buffer elements are inserted into the heap structure, filling up the leaf holes. These buffer elements are then handled by the extracting routine in the standard way, as ordinary active elements. Since these elements may travel down, from the leaf level closer to the root level, a node containing a buffer element may have a son containing a smaller buffer element. This will do no harm, however, since each buffer element is strictly greater than any active element, because of the buffer separator \( b^2 \). Thus, no buffer element can be extracted from the heap as the smallest element in the first \( h_k \) iterations, when the routine terminates.

**Lemma 5.3** Sorting all segments does not require more than \( O(m(\log m)^{4/5}) \) comparisons or \( 6m + O(m/(\log m)^{1/5}) \) moves.

**Proof:** Deriving the costs of the above routine is straightforward. The routine repeats \( h_k \) iterations, performing each time at most \( 5(t-1) \leq 5(\log m)^{4/5} \) comparisons and 6 moves, since the heap has at most five levels. This gives \( h_k 5(\log m)^{4/5} \) comparisons and \( h_k 6 \) moves.

Now we can sum the costs of sorting the segment \( \sigma_k \). Determining the value of \( h_k \) costs \( O(\log \log m) \) comparisons. Building the heap costs at most \( 2h_k \) comparisons and \( 6h_k/(\log m)^{4/5} \) moves. Extracting active elements in sorted order costs \( h_k 5(\log m)^{4/5} \) comparisons and \( h_k 6 \) moves. Summing up, we get \( h_k O((\log m)^{4/5}) \) comparisons and \( h_k (6/(\log m)^{4/5} + 6) \) moves.

To obtain the total cost of sorting all segments \( \sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{f_m} \), we use the fact that \( \sum_{k=0}^{f_m} h_k \leq m \), since the number of active elements stored in the segments is bounded by the total number of active elements. Therefore, the sum over all segments results in the bound of the thesis. \( \square \)
Alternatively, we could use the heap structure with parameter $t = \lfloor \log m \rfloor$. This results in a heap with four levels, instead of five (since $[x]^4 \geq [x^4]$, for each real $x > 0$). This reduces the leading factor for the number of moves from $6m$ to $5m$. The price we pay is increasing the number of comparisons, from $o(m \log m)$ to $4m \log m + O(m)$. The detailed argument is very similar to the proof for $t = \lceil (\log m)^{4/5} \rceil$.

### 5.2.9 Rebalancing at the segment level

This procedure is activated by the routine of Section 5.2.6, inserting a new active element in the structure, when, for some $k$, the segment $\sigma_k$ has become full, having absorbed $s$ active elements.

At the moment of activation, some global index variable is pointing to the starting position of $\sigma_k$. The procedure also remembers $\ell$, the position of the associated active element $a_k = x_\ell$ in the frame memory, as well as $j$, the position of the frame block containing the element $a_k$. We shall call this block the current frame block. (If $\sigma_k = \sigma_0$, i.e., $k = 0$, there is no associated element in the frame. Then $\ell = 0$, but we still have the current frame block, namely, $j = 1$). The above indices were computed when the latest active element was inserted in the structure.

1. By the use of a binary search with the buffer separator $b^2$ over the $r$ locations in the current frame block, find $\ell'$, the position of the leftmost buffer element in this block. We shall denote this element by $b^\circ$. Recall that we maintain the invariant that each active frame block has a room for one more active element, and therefore it does contain at least one buffer element.

2. Find a median in the segment $\sigma_k$, i.e., an element $a^\circ$ of rank $|s/2|+1$. Without loss of efficiency, the selection procedure will position $a^\circ$ at the end of $\sigma_k$.

3. The median $a^\circ$ is inserted in the current frame block, one position to the right of $a_k$. The active elements lying in between $a_k$ and $b^\circ$, that is, occupying locations $x_{\ell+1} \ldots x_{\ell'-1}$ in the frame memory, are shifted one position to the right. At the same time, $b^\circ$ is saved from $x_{\ell'}$ to the location released by $a^\circ$ at the end of the segment $\sigma_k$. (As a special case, if $a_k$ is the rightmost active element in the current frame block, only $b^\circ$ and $a^\circ$ are swapped. The same holds when $\sigma_0$ is rebalanced for the first time, with $\ell = 0$ and $\ell' = 1$). Since $a^\circ$ has been picked from $\sigma_k$, it satisfies $a_k \leq a^\circ \leq a_{k+1}$, and hence the sequence of active elements stored in the frame memory remains sorted.

4. After shifting the active elements in the locations $x_{\ell+1} \ldots x_{\ell'-1}$ one position to the right, we have to shift the corresponding pointers $\pi_{\ell+1} \ldots \pi_{\ell'-1}$ as well, so the active elements remain connected with their segments. To move an integer pointer value from $\pi_e$ to $\pi_{e+1}$, we only have to read the value encoded in $\pi_e$ and, at the same time, clear $\pi_e$, and then to encode this value in $\pi_{e+1}$. Such transport of a pointer costs $O(p)$ comparisons and moves.
5. We need to connect a new active element in the frame with a new segment. This concerns the element $a^\circ$, now placed in $x_{t+1}$. Thus, we allocate a new segment $\sigma^\circ$ and encode its starting position in the pointer $\pi_{t+1}$.

6. The full segment $\sigma_k$ is halved, that is, we place some $|s/2|$ active elements greater than or equal to $a^\circ$ into the left part of $\sigma^\circ$ and collect the remaining $|s/2|$ active elements, smaller than or equal to $a^\circ$, in the left part of the original segment $\sigma_k$. Since many elements may be equal to $a^\circ$, we distribute such elements both to $\sigma_k$ and $\sigma^\circ$, so that their active parts are of equal lengths. This also requires to save $|s/2|$ buffer elements, placed originally in $\sigma^\circ$, to the locations released in $\sigma_k$. (We shall give more details below, in Section 5.2.10). The outcome of halving is that the active elements in $\sigma_k$ are split into two segments $\sigma_k$ and $\sigma^\circ$, satisfying $a_k \leq a \leq a^\circ$ and $a^\circ \leq a \leq a_{k+1}$, respectively.

7. If there is still a room for storing one more active element in the current frame block, the structure has been rebalanced. We are done, ready to take the next element from $A$. However, if this block has become full, because of $a^\circ$, the program control jumps to a routine rebalancing the frame level, described later, in Section 5.2.11.

Let us now derive the computational costs.

**Lemma 5.4** The total cost of keeping the segment level balanced is $O(m)$ comparisons and $(3+\varepsilon)m$ moves, where $\varepsilon > 0$ is an arbitrarily small, but fixed, real constant.

**Proof:** The binary search, determining the position of the leftmost buffer element in the current frame block, inspects a range of $r$ elements, performing $1 + \lfloor \log r \rfloor \leq O(\log \log m)$ comparisons, by (5.4). Finding a median, in a segment of length $s$, requires only $O(s) \leq O((\log m)^4)$ comparisons and $\varepsilon s \leq \varepsilon (2 + (\log m)^4)$ element moves, where $\varepsilon > 0$ is an arbitrarily small, but fixed, real constant, by [Geffert and Kollár, 2001] and (5.1). Rearranging the elements $a^\circ, b^\circ$, and $x_{t+1} \ldots x_{\rho-1}$ in their locations can be done with at most $r + 2 \leq O(\log m)$ moves, by (5.4). Shifting the pointers $\pi_{t+1} \ldots \pi_{\rho-1}$ one position to the right costs $O(rp) \leq O((\log m)^2)$ comparisons, by (5.4) and (5.7), together with the same number of moves. Encoding the starting position of a new segment in the pointer $\pi_{t+1}$ requires $O(p) \leq O(\log m)$ element moves, by (5.7). Halving the active elements in $\sigma_k$ into two segments $\sigma_k$ and $\sigma^\circ$ requires only $O(s) \leq O((\log m)^4)$ comparisons and $3/2s \leq 3/2(2 + (\log m)^4)$ moves, using Lemma 5.5, displayed in Section 5.2.10 below, and (5.1).

By summing the bounds above, we get that a single activation of the procedure rebalancing a segment performs $O((\log m)^4)$ comparisons and $(3/2+\varepsilon)(\log m)^4$ moves. Taking into account that each activation increases the number of active segments, that we start with one segment, namely, $\sigma_0$, and that we end up with $f_{m+1}$ segments, we see that the number of activations is bounded by $f_m$. This value is bounded by $f_m \leq s_m \leq 2m/(\log m)^4$, using (5.6) and (5.2). □
5.2.10 Halving a segment

Here we describe a simple procedure for halving, needed in Section 5.2.9 above. We are given a segment \( \sigma_k \) of size \( s \), and a median \( a^\circ \), that is, an element of rank \( \lfloor s/2 \rfloor + 1 \), put aside. We want to place some \( \lfloor s/2 \rfloor \) active elements greater than or equal to \( a^\circ \) into the left part of another given segment \( \sigma^\circ \), of size \( s \) again, and collect the remaining \( \lfloor s/2 \rfloor \) elements smaller than or equal to \( a^\circ \) in the left part of \( \sigma_k \). The first \( \lfloor s/2 \rfloor \) buffer elements of \( \sigma^\circ \) must be saved. We have two phases.

**First phase: counting.** In the first phase, with \( s-1 \) comparisons and no moves, we count \( c' \), the number of elements strictly smaller than \( a^\circ \), in \( \sigma_k \). This gives us \( c = \lfloor s/2 \rfloor - c' \), the number of elements equal to \( a^\circ \) that should remain in \( \sigma_k \). This number will be required in the second phase, when each element \( a \) of \( \sigma_k \) is compared with \( a^\circ \) twice, using \( "a < a^\circ" \) and \( "a > a^\circ" \). The elements strictly smaller than \( a^\circ \) and the first \( c \) elements detected to be equal to \( a^\circ \) will be considered “small,” while the remaining equal elements and those strictly greater than \( a^\circ \) will be “large.” Each time an element \( a = a^\circ \) is detected, the counter \( c \) will be decreased by one, until it gets to zero. From then on, any “new” element \( a \) will be considered “small” if and only if \( a < a^\circ \), and “large” otherwise.

**Second phase: moving.** In the second phase, the configurations of the segments are

\[
\sigma_k = A_1 U B_1 b^\circ \quad \text{and} \quad \sigma^\circ = A_2 B_2,
\]

where \( A_1 \) and \( A_2 \) denote, respectively, the active elements of \( \sigma_k \) found to be “small” or “large,” collected so far, \( B_1 \) the buffer elements moved from \( \sigma^\circ \) to \( \sigma_k \), \( B_2 \) the elements of \( \sigma^\circ \) not moved yet, \( U \) the elements of \( \sigma_k \) not examined yet, and \( b^\circ \) a single buffer element, filling up the room. \( A_2 \) and \( B_1 \) are of equal length, not exceeding \( \lceil s/2 \rceil \). Initially, \( \sigma_k = U b^\circ \), \( \sigma^\circ = B_2 \), with \( A_1, A_2, \) and \( B_1 \) empty. The procedure also remembers the current position of the hole. (After the first iteration, the hole is always in the leftmost location of \( B_1 \)).

The second phase proceeds in a loop, as follows.

1. Using at most two comparisons, the rightmost element \( a \) of \( U \) is determined to be “small” or “large.” If \( a \) is large, we save the leftmost element from \( B_2 \) in the current location of the hole and fill up the new hole in \( B_2 \) by \( a \). Thus, \( A_2 \) and \( B_1 \) have been extended, while \( U \) and \( B_2 \) have been reduced.

2. If \( a \) is small, we scan \( U \) from left to right until we find the first element \( a' \) that is large. All elements on the left of \( a' \) become a part of \( A_1 \), without being moved. Since \( a \) is a small element placed on the right of the position \( \lfloor s/2 \rfloor \), \( a' \) must be found before we reach the position \( \lfloor s/2 \rfloor + 1 \), or else we would have more than \( \lfloor s/2 \rfloor \) small elements, which is a contradiction. Now we save the
leftmost element from $B_2$ to the hole, fill up the hole in $B_2$ by $a'$, and move $a$ to the place released by $a'$. Then all necessary boundaries are updated.

This is repeated until we have transported exactly $\lfloor s/2 \rfloor$ active large elements from $\sigma_k$ to $\sigma'$. As a consequence, the remaining $\lfloor s/2 \rfloor$ active elements of $\sigma_k$, placed on the left of $B_1$, must be all small, since the rank of $a'$ is $\lfloor s/2 \rfloor + 1$ and $s$ is odd, by (5.1).

**Lemma 5.5** Given a median $a'$, a segment of size $s$ can be halved with at most $O(s)$ comparisons and $3/2s$ moves.

**Proof:** We have used at most $3s$ comparisons in total, and at most three moves per each large element moved from $\sigma_k$ to $\sigma'$.

\[\square\]

### 5.2.11 Rebalancing at the frame level

This routine is activated by the procedure of Section 5.2.9, rebalancing a segment, when it finds out that, for some $j$, the $j$th frame block has become full, having absorbed $r$ active elements. As a side effect, the routine may increase the number of active blocks in the frame. The routine is based on a new variant of the well-known data structure (see [Itai, Konheim, and Rodeh, 1981, Willard, 1982]), used to maintain a set of elements in sorted order in a contiguous zone of memory (see Chapter 4).

For the purpose of keeping the frame memory balanced, the frame consisting of $r_\ast$ frame blocks is viewed, implicitly, as a complete binary tree with $r_\ast = 2^{r-1}$ leaves, and hence of (edge) height $r-1$. We introduce the following numbering of levels: $i = 0$ for the leaves, 1 for their fathers, and so on, ending by $i = r-1$ for the root. Each node of the tree is associated with a contiguous subzone of the frame blocks, and with a path leading to this node from the root, as follows.

- The $j$th leaf, for any $j$ ranging between 1 and $2^{r-1}$, is associated with the $j$th frame block, i.e., with a subzone consisting of $1 = 2^0$ frame blocks, starting from the block position $j$.

The corresponding path from the root to this leaf is represented by the number $\overline{j} = j-1$. It is easy to see that by reading the binary representation of $\overline{j}$ from left to right (with leading zeros so that its length is $r-1$) we get the branching sequence along this path; 0 is interpreted as branching to the left, while 1 as branching to the right.

- Given a node $v$ at a level $i$, associated with a path number $\overline{j}$ and with a subzone of length $2^i$ blocks, starting from a block position $j$, the father $v'$ of this node is associated with the path number $\overline{j'} = \lfloor \overline{j}/2 \rfloor$, and with the subzone of length $2^{i+1}$, starting from the block position $j' = j$, if $\overline{j}$ is even ($v$ is a left son of $v'$), but from $j' = j - 2^i$, if $\overline{j}$ is odd (right son).
Thus, the subzone for the father is obtained by concatenating the two subzones for its sons, while its path number by cutting off the last bit in the path number for any of its sons.

During the computation, the number of active elements in some local area of the frame may become too large. The purpose of rebalancing a subzone, associated with a node \( v \) at a level \( i \), for \( i > 0 \), is to eliminate such local densities and redistribute active elements more evenly. Since the density threshold of any descendant of \( v \) is greater than that associated with \( v \), all the violations of the density constraints under the subtree rooted at \( v \) will be removed. More precisely, after rebalancing the subzone, the following two conditions will hold:

\[
\text{The number of active elements, in any frame block belonging to the subzone associated with the given node } v \text{ at the level } i, \text{ will not exceed the threshold } \tau_i = r - i \tag{5.9}
\]

\[
\text{The frame memory will not contain any free blocks (without active elements) in between some active blocks.} \tag{5.10}
\]

Note that, if a node \( v \) at a level \( i > 0 \) is an ancestor of the \( j \)th leaf, the condition (5.9) ensures that the \( j \)th frame block is not full any longer. Neither is any other block within the subzone. Such redistribution of active elements is possible only if \( \alpha(v) \), the total number of active elements in the subzone associated with \( v \), is bounded by \( \alpha(v) \leq \tau_i 2^i \). We say that the node \( v \) overflows, if the following holds

\[
\alpha(v) > \tau_i 2^i.
\]

Note also that the distance between the leftmost active frame block and the rightmost one in the frame level is \( \Theta(wr) \), where \( w \) is the number of active elements...
contained in the frame level and $r$ is the length of a frame block. That is because the number of active elements stored in an active frame block is lower bounded only by 1 (condition (5.9)) and by the fact that there are no frame block among the active blocks (condition (5.10)).

The condition (5.10) is required also because of the procedure presented in Section 5.2.6, transporting active elements from the block $A$ to the structure. Recall that this procedure uses a binary search over the leftmost locations in the active frame blocks, and hence these blocks must form a contiguous zone.

Now we can describe the routine rebalancing the frame.

1. Starting from the father of the frame block that is full, climb up and find the lowest ancestor $v$ that does not overflow, with $\alpha(v) \leq \pi_i 2^i$. The formulas for $j$ and $\tilde{j}$, presented above, give us a simple tool for computing the boundaries of the associated subzones, along the path climbing towards the root. To compute the value of $\alpha(v)$, for the given ancestor $v$ at the given level $i$, simply scan all $2^i$ frame blocks forming the associated subzone and sum up the numbers of active elements in these blocks, using a binary search with the buffer separator $b^\mathfrak{z}$ over the $r$ locations in each block.

2. Move the $\alpha(v)$ active elements in the associated subzone of $v$ to the last $\alpha(v)$ locations. That is, processing all $2^i r$ locations in the subzone from the right to left, collect all elements smaller than $b^\mathfrak{z}$ to the right end. Before moving an active element from $x_e$ to $x_{e'}$, for some $e < e'$, the buffer element in the target position $x_{e'}$ is saved to the current location of the hole. Then move the associated pointer in the corresponding positions of the pointer memory, from $\pi_e$ to $\pi_{e'}$, by reading and clearing the bit value encoded in $\pi_e$ and encoding this value in $\pi_{e'}$.

3. Redistribute the $\alpha(v)$ active elements back, this time more evenly in the $2^i$ frame blocks of the subzone, moving also the pointers in the corresponding positions, as follows. Let $\alpha_n = \lfloor \alpha(v)/2^i \rfloor$ and $\alpha_m = \alpha(v) \mod 2^i$. Then put $\alpha_n + 1$ active elements in each of the first $\alpha_m$ blocks, and $\alpha_n$ active elements in each of the remaining $2^i - \alpha_m$ blocks. In each block, the active elements are concentrated in its left part.

4. As a side effect of redistribution, the size of the active part in the frame memory may have been increased. This requires to update the value of $g$, the number of active frame blocks, kept in a separate global index variable. Let $g'$ be the block position of the rightmost frame block in the subzone of $v$. Then let $g := \max \{g, g'\}$.

It should be pointed out that, for each leaf, the desired ancestor $v$ without overflow does exist. Using (5.6), (5.4), and $\tau_i = r - i$, for the level $i = r - 1$, that is, for $v$ being the root node, we get
\[ \alpha(v) = f \leq r_\alpha = 1 \cdot 2^{r-1} = \tau_{r-1}2^{r-1}, \]
and hence at least the root node does not overflow. Therefore, in the first step, the loop climbing up towards the root must halt correctly.

Further, the redistribution of active elements, presented in the third step, is correct, since

\[(\alpha_d + 1)\alpha_m + \alpha_d(2^i - \alpha_m) = \alpha(v).\]

**Lemma 5.6** The redistribution procedure satisfies conditions (5.9) and (5.10).

*Proof:* It is easy to see that the redistribution satisfies the condition (5.9) above, using the fact that the node \( v \) does not overflow, and hence \( \alpha(v) \leq \tau_i2^i \). There are two cases to consider: For \( \alpha(v) \leq \tau_i2^i - 1 \), we have

\[ \alpha_d + 1 \leq \left( \frac{\tau_i2^i - 1}{2^i} \right) + 1 \leq (\tau_i - 1) + 1 = \tau_i, \]

since \( \tau_i \) is an integer. If \( \alpha(v) = \tau_i2^i \), we get \( \alpha_m = 0 \), and hence all \( 2^i \) blocks are “remaining,” with only \( \alpha_d \) active elements in each. But here

\[ \alpha_d = \left\lfloor \frac{\tau_i2^i}{2^i} \right\rfloor = \lfloor \tau_i \rfloor = \tau_i. \]

It is also easy to see that the redistribution satisfies the condition (5.10). Since \( v \) is the lowest ancestor that does not overflow, along the path from the full frame block towards the root, it must have a son that does overflow, with at least \( \tau_i-12^{i-1} \) active elements in its subzone. (As a special case, for \( i = 1 \), we get \( \tau_02^0 = (r-0)r1 = r \) active elements in the \( j \)th frame block that is full). The subzone of the son is a part of the subzone associated with \( v \), and hence

\[ \alpha(v) \geq \tau_i-12^{i-1} \geq 2 \cdot 2^{i-1}, \]

using the fact that \( i-1 \leq r-2 \). But then

\[ \alpha_d = \left\lfloor \frac{\alpha(v)}{2^i} \right\rfloor \geq 1. \]

This implies that each frame block in the subzone associated with \( v \) contains at least one active element after redistribution, and hence the zone of active frame blocks will remain contiguous. \( \square \)

Now, we have to analyze the costs of maintaining the frame level balanced.

**Theorem 5.1** The total cost of keeping the frame memory balanced is \( O(m/\log m) \) comparisons, together with the same number of moves.
Proof: Consider now the cost of a single activation of the above routine, rebalancing a subzone for a node $v$ at a level $i$. Looking for the lowest ancestor without overflow requires to count the numbers of active elements in the associated subzones along a path climbing up from a father of a leaf, for levels $e = 1, \ldots, i$. In the $e$th level, $2^e$ blocks are examined, by a binary search over the $r$ locations of the block. By (5.4), this has the following bound for the comparisons:

$$
\sum_{e=1}^{i} 2^e \cdot (1 + \lfloor \log r \rfloor) \leq 2^i O(\log \log m).
$$

The cost of the second step, collecting $\alpha(v)$ active elements to the right end, is $2^i r$ comparisons (one comparison with $b^3$ for each location in the subzone), plus $\alpha(v)2i+1$ moves (two moves per each collected element). However, with each collected element, the corresponding pointer must also be transported, which gives additional $\alpha(v)O(p)$ comparisons and moves. Using $\alpha(v) \leq \tau_i 2^i \leq r 2^i$, together with (5.4) and (5.7), the number of comparisons and moves performed in the second step can be bounded by

$$
2^i O(rp) \leq 2^i O((\log m)^2).
$$

The same computational resources are sufficient in the third step, redistributing the same number of active elements back, but more evenly, together with their pointers. Again, this gives $\alpha(v)O(p)$ comparisons and moves, which can be bounded by $2^i O((\log m)^2)$.

Finally, the fourth step does not require any element comparisons or moves, it just updates one index variable, in $O(1)$ time.

Summing up, the cost of a single activation is $2^i O((\log m)^2)$ comparisons and moves, for each node $v$ at the fixed level $i > 0$. To get the total cost, we must take into account how frequently such rebalancing is activated.

When a rebalancing is activated, $v$ must have a son with at least $\tau_{i-1} 2^{i-1}$ active elements, since $v$ is the lowest ancestor that does not overflow, along some path climbing up. Now, trace back the history of computation, to the moment when the entire subzone associated with $v$ was a subject of redistribution for the last time. This way we get a node $v'$, either an ancestor of $v$ or $v$ itself, at a level $i' \geq i$, with the associated subzone containing the entire subzone for $v$. After the redistribution for $v'$, both sons of $v$ contained at most $\tau_{i'} 2^{i-1} \leq \tau_{i} 2^{i-1}$ active elements. Thus, in the meantime, the number of active elements in one of the sons of $v$ has been increased by at least $\tau_{i-1} 2^{i-1} - \tau_{i} 2^{i-1} = 2^{i-1}$.

Since other redistributions, taking place between the moments of rebalancing $v'$ and $v$, could not “import” any active elements to the subzone of $v$ from any other parts of the frame, the $2^{i-1}$ additional active elements must have been inserted here. (See the procedure of Section 5.2.9, third step). Thus, there have to be at least $2^{i-1}$ insertions in the associated subzone between any two redistributions for $v$. Note that, for the fixed level $i$, subzones associated with different nodes $v$ do not
overlap. Thus, we can charge the cost of each activation, for the given node \( v \), to the \( 2^{i-1} \) insertions preceding this activation in the given subzone, without charging the same insertion more than once. This gives the following bound for comparisons and moves, per a single insertion of an active element in the frame memory.

\[
2^i \frac{O((\log m)^2)}{2^{i-1}} \leq O((\log m)^2).
\]

Since, in the whole computation, there were only \( f_m \leq r_+ \leq O(m/(\log m)^4) \) insertions, by (5.6) and (5.4), we get the cost \( O(m/(\log m)^2) \) comparisons and moves, for rebalancing of all nodes at the fixed level \( i \). By summing over all levels, using \( i \leq r-1 \leq \log m \), by (5.4), we get the total cost.

5.2.12 Summary

By summing the bounds presented in Lemmas 5.1–5.4 and Theorem 5.1 above, we get:

**Theorem 5.2** In order to sort the given block \( A \) of size \( m \) we need \( 2m \log m + O(m(\log m)^{4/5}) \) comparisons and \( (1+\varepsilon)m \) moves, where \( \varepsilon > 0 \) is an arbitrarily small, but fixed, real constant, provided we can use additional buffer and pointer memories, of respective sizes \( 3m-1 \) and \( [4m/(\log m)^2] \).

The algorithm presented above assumes that \( m \) is "sufficiently large," so that \( s \), defined by (5.1), satisfies \( s \leq m \). This presupposition holds for each \( m > 2^{16} = 65536 \). Shorter blocks are handled in a different way, by the procedure described later, in Section 5.3.3. The bounds presented by Theorem 5.2 for the number of comparisons and moves will remain valid.

5.3 In-Place Sorting

Now we can present an in-place algorithm sorting the given sequence \( A \) consisting of \( n \) elements. If \( n \leq 2^{16} \), the sequence is sorted directly, by the procedure of Section 5.3.3, described later. In the general case, for \( n > 2^{16} \), the task of the main program is to provide sufficiently large pointer and buffer memories for the procedure presented in Section 5.2.

5.3.1 Building a pointer memory

The size of the largest block ever sorted by the procedure of Section 5.2 will not exceed \( m = n/4 \). Using (5.8) and the fact that the function \( 4x/(\log x)^2 \) is monotonically increasing for \( x \geq 8 \), we see that the size of the pointer memory can be bounded by
$$P = \left\lfloor \frac{n}{\left(\log_2 \frac{n}{4}\right)^2} \right\rfloor.$$  

This will suffice for all sorted blocks.

The pointer memory is built by collecting two contiguous blocks $\Pi_L$ and $\Pi_R$. The block $\Pi_L$, placed at the left end of $\mathcal{A}$, will contain the smallest $P$ elements of the sequence $\mathcal{A}$, while $\Pi_R$, placed at the right end, the largest $P$ elements.

The block $\Pi_R$ is created first, by the use of the Heapsort with $t$ root nodes and internal nodes having $t$ sons. The detailed topology of edges connecting nodes in this kind of heap has been presented in Section 5.2.8, devoted to extracting sorted elements at the segment level.

However, there are some substantial differences from the generalized Heapsort of Section 5.2.8. This time the branching degree is $t = \lceil \log n \rceil$. Therefore, the height of the heap is

$$q \leq 1 + \lceil \log_t n \rceil \leq O \left( \frac{\log n}{\log \log n} \right).$$  

Here we keep large elements at the root level, instead of small elements. That is, no node contains an element smaller than any of its sons. Unlike in Section 5.2.8, no buffer elements are used here to fill up the holes, the heap structure shrinks in the standard way, when the largest element is extracted.

The initial building of the heap structure is standard, and agrees with the heap building in Section 5.2.8. By $t \geq \log n$, it is easy to see that, for a heap with $n$ elements, branching degree equal to $t$, and $q$ levels, the number of comparison of the heap initialization can be bounded by

$$t \sum_{i=1}^{q-1} \frac{n}{t^i} < \frac{nt}{t-1} \leq O(n)$$

whereas the number of moves is

$$3 \sum_{i=1}^{q-1} \frac{n}{t^i} < \frac{3n}{t-1} \leq O \left( \frac{n}{\log n} \right).$$

After building the heap, the routine extracts, $P$ times, the largest element from the heap in the standard way. That is, when the largest element is extracted, it replaces the element in the rightmost leaf, which in turn is inserted into the “proper” position along the so-called special path, starting from the position of the largest root (just being extracted) and branching always to the largest son.

The costs of the above routine are straightforward. The trajectory of the special path can be localized with $q(t-1)$ comparisons, and the new position for the element in the rightmost leaf can be found by a binary search along this trajectory with $1 + \lfloor \log q \rfloor$ comparisons. Summing up, an extraction of the largest element can be
done with \( q(t-1) + (1 + \log q) \) comparisons, together with \( q + 2 \) moves. Using
\( t \leq O(\log n) \) and \( q \leq O(\log n / \log \log n) \), we get, per a single extraction, at most
\( O((\log n)^2 / \log \log n) \) comparisons, together with \( O(\log n / \log \log n) \) moves.

If we let the above procedure run till the end, it would sort the entire sequence \( \mathcal{A} \)
in time \( O(n(\log n)^2 / \log \log n) \). However, the execution is aborted as soon as the
largest \( P \) elements are collected. Since \( P \leq O(n / (\log n)^2) \), the cost of building the
heap becomes dominant, and hence the block \( \Pi_R \) is created with \( O(n) \) comparisons
and \( O(n / \log n) \) moves.

After \( \Pi_R \), the block \( \Pi_L \) is created in the same way, with the same computational
needs of comparisons and moves. Instead of large elements here we collect the
smallest \( P \) elements. In addition, since \( \Pi_L \) should be created at the left end of \( \mathcal{A} \),
all indices are manipulated in a mirrorlike way, seeing the first position to the left
of \( \Pi_R \) as the beginning of the sequence.

From the above discussion, we have that:

**Lemma 5.7** Building the pointer memory requires \( O(n) \) comparisons and \( O(n / \log n) \)
moves.

Now the configuration of the sequence \( \mathcal{A} \) has changed to \( \Pi_L \mathcal{A}' \Pi_R \), where \( \mathcal{A}' \) de-
notes the remaining elements, to be sorted. Before proceeding further, the algorithm
verifies, with a single comparison, whether the largest (rightmost) element in \( \Pi_L \) is
strictly smaller than the smallest (leftmost) element in \( \Pi_R \).

If this is not the case, all elements in \( \mathcal{A}' \) must be equal to these two elements.
Therefore, the algorithm terminates, the entire sequence \( \mathcal{A} \) has already been sorted.

Conversely, if \( \Pi_L \) and \( \Pi_R \) pass the test above, they can be used to imitate a
pointer memory consisting of \( P \) bits.

### 5.3.2 Partition-based sorting

When the blocks \( \Pi_L \) and \( \Pi_R \) have been created, the zone \( \mathcal{A}' \) is kept in the form
\( \mathcal{A}_s \mathcal{A}_u \), where \( \mathcal{A}_s \) and \( \mathcal{A}_u \) represent the sorted and unsorted parts of \( \mathcal{A}' \), respectively.
Each element in \( \mathcal{A}_s \) is strictly smaller than the smallest element of \( \mathcal{A}_u \). The routine
described here is a partition-based loop. In the course of the \( i \)th iteration, the
length of \( \mathcal{A}_u \) is \( n_i \), with \( n_i < n_{i-1} \). Initially, for \( i = 0 \), \( \mathcal{A}_s \) is empty, \( \mathcal{A}_u = \mathcal{A}' \), and
\( n_0 = n - 2P < n \). The loop proceeds as follows (see Figure 5.3).

1. Find \( b^\leq \), an element of rank \([n_i / 4]\) in \( \mathcal{A}_u \). The selection procedure places this
element at the right end of \( \mathcal{A}_u \), so the configuration of \( \mathcal{A}' \) changes to \( \mathcal{A}_s \mathcal{A}'_u b^\leq \).
   Here \( \mathcal{A}'_u \) denotes a mix of elements in \( \mathcal{A}_u \), of length \( n_i - 1 \).

2. \( \mathcal{A}'_u \) is partitioned into two blocks \( A_< \) and \( B_\geq \) consisting, respectively, of el-
   ements strictly smaller than \( b^\leq \) and of those greater than or equal to \( b^\leq \).
The configuration of the sequence thus changes to \( \mathcal{A}_s A_< B_\geq b^\leq \). The respective
lengths of \( A_< \) and \( B_\geq \) will be denoted here by \( n_{i,<} \) and \( n_{i,\geq} \). Note that,
even for a large block \( A_u \), we may obtain a very short block \( A_\zeta \), since many elements may be equal to \( b^\zeta \). In fact, the block \( A_\zeta \) may even be empty, of length \( n_{i,\zeta} = 0 \).

3. Sort the block \( A_\zeta \) by the procedure described in Section 5.2, using some initial segments of \( \Pi_L \) and \( \Pi_R \) as a pointer memory and of \( B_\zeta \) as a buffer memory, with \( b^\zeta \) as a buffer separator.

This is possible, since \( b^\zeta \) has been selected as an element of rank \( \lceil n_i/4 \rceil \), and hence

\[
 n_{i,\zeta} \leq \left\lfloor \frac{n_i}{4} \right\rfloor - 1 < \frac{n_i}{4},
\]

with \( n_{i,\zeta} + n_{i,\geq} + 1 = n_i \). But the required size of buffer is only

\[
3n_{i,\zeta} - 1 \leq \frac{4n_i}{3} - 1 = n_i - 1 - \frac{n_i}{4} \leq n_i - 1 - n_{i,\zeta} = n_{i,\geq}.
\]

Therefore, the block \( B_\zeta \) of length \( n_{i,\geq} \) is sufficiently long. Similarly, the required number of bits for pointers is

\[
\left\lfloor \frac{4n_{i,\zeta}}{(\log n_{i,\zeta})^2} \right\rfloor \leq \left\lfloor \frac{n}{(\log(n/4))^2} \right\rfloor = P,
\]

and hence the pointer memory is also sufficiently large. (If \( n_{i,\zeta} \leq 2^{16} \), \( A_\zeta \) is sorted as a short block).

4. Restore the sorted order in \( \Pi_L \) and \( \Pi_R \), by clearing all bits of the pointer memory to zero. Among others, this is required because the procedure of Section 5.2 will also be used in subsequent iterations, when it assumes that all bits are initially cleared.

5. After sorting \( A_\zeta \), the configuration of \( A' \) is \( A_u A_{<,\zeta} B'_\zeta b^\zeta \), where \( A_{<,\zeta} \) denotes the sorted version of the block \( A_\zeta \) and \( B'_\zeta \) a mixed up version of \( B_\zeta \). Now put the first element in \( B'_\zeta \) aside and move \( b^\zeta \) to the first position after \( A_{<,\zeta} \). After that, collect all elements smaller than or equal to \( b^\zeta \) to the left part of \( B'_\zeta \), processing also the element put aside. Since \( B_\zeta \) did not contain elements strictly smaller than \( b^\zeta \), this actually partitions \( B'_\zeta \) into two blocks \( A_- \) and \( B_> \) consisting, respectively, of elements equal to \( b^\zeta \) and of those strictly greater than \( b^\zeta \), of respective lengths \( n_{i,-} \) and \( n_{i,>} \). Clearly, \( n_{i,-} + n_{i,>} = n_{i,\geq} \). The configuration has changed to \( A_u A_{<,\zeta} b^\zeta A_- B_> \).
6. Observe that $A_\langle A_\leq b^\leq A_\leq$ and $B_\geq$ can be viewed as “new” variants of blocks $A_\leq$ and $A_\geq$. Thus, we can start a new iteration, with $B_\geq$ as a new block $A_\geq$, of length $n_{i+1} = n_{i\geq}$. The above process is iterated until the length of unsorted part drops to $2^{16}$, or below. This residue is then sorted as a short block, without using a buffer or pointers, which will be described later, in Section 5.3.3.

Now we can derive computational costs.

**Theorem 5.3** The given sequence, consisting of $n$ elements, can be sorted in-place by performing at most $2n \log n + o(n \log n)$ comparisons and $(13+\varepsilon)n$ element moves, where $\varepsilon > 0$ denotes an arbitrarily small, but fixed, real constant. The number of auxiliary arithmetic operations with indices is bounded by $O(n \log n)$.

**Proof:** First, recall that $b^\leq$ has been selected as an element of rank $[n_i/4]$, and hence

$$n_{i+1} = n_{i\geq} \leq n_i - \left\lfloor \frac{n_i}{4} \right\rfloor \leq \frac{3}{4} n_i.
$$

Taking into account that $n_0 \leq n$, we get $n_i \leq (3/4)^i n$, for each $i \geq 0$. This gives that

$$\sum_{i=0}^{I-1} n_i \leq 4n,$$

$$I \leq O(\log n),$$

where $I$ denotes the number of iterations. Second, it is easy to see that

$$\sum_{i=0}^{I-1} (n_i + n_i) + n_I \leq n$$

since, in different iterations, the final locations occupied by $A_\leq$, $b^\leq$, and $A_\geq$, do not overlap. Here $n_I$ denotes the length of the residual short block.

Let us now present the costs for the $i$th iteration. Selection of $b^\leq$, an element of the given rank in a block of length $n_i$, costs $O(n_i)$ comparisons and $\varepsilon n_i$ moves, by [Geffert and Kollár, 2001]. Partitioning of $A'_{\leq}$ into blocks $A_\leq$ and $B_\leq$ can be done with $n_i$ comparisons and $2n_{i\geq} + 1$ moves, since the length of $A'_{\geq}$ is $n_i - 1$, and the number of collected elements, strictly smaller than $b^\leq$, is $n_i$. The cost of sorting the block $A_\leq$ is bounded by

$$2n_{i\leq} \log n_{i\leq} + O \left( n_{i\leq} (\log n_{i\leq})^{\frac{1}{2}} \right) \leq 2n_{i\leq} \log n + O \left( n_{i\leq} (\log n)^{\frac{1}{2}} \right)$$

comparisons and $(11+\varepsilon)n_{i\leq}$ moves, by Theorem 5.2. Sorting of the block $A_\geq$ is followed by restoring the sorted order in $\Pi_L$ and $\Pi_R$, by clearing all bits, which costs $O(P) \leq O(n/(\log n)^2)$ comparisons, together with the same number of moves.
Figure 5.3: Two iterations of the partition-based sorting.
Positioning \(b^\perp\) to the right of \(A_{\leq s}\) requires only 2 element moves. Finally, the \(i\)th iteration is concluded by partitioning \(B'_{\geq}\) into blocks \(A_\leq\) and \(B_\geq\), with at most \(n_{i,\geq} \leq n_i\) comparisons and \(2n_{i,-} + 1\) moves, since the length of \(B'_{\geq}\) is \(n_{i,\geq}\), and the number of collected elements equal to \(b^\perp\) is \(n_{i,-}\). The cost of sorting the residual short block does not exceed the bounds for the standard case. We have the following bound for the number of comparisons:

\[
2n_T \log n_T + 6.25n_T \leq 2n_T \log n + O \left( n_T \left( \log n \right)^{\frac{4}{5}} \right)
\]

and the next one for the number of moves (see Section 5.3.3):

\[
9.75n_T \leq (11 + \varepsilon)n_T.
\]

Now we can sum the above costs over all iterations, using (5.11) and (5.12). For the number of comparisons, this gives

\[
C(n) \leq \sum_{i=0}^{T-1} n_i O(1) + \sum_{i=0}^{T-1} n_{i,\leq} \left( 2 \log n + O \left( \left( \log n \right)^{4/5} \right) \right) + \sum_{i=0}^{T-1} O \left( \frac{n}{\log n} \right)
\]

\[
+ n_T \left( 2 \log n + O \left( \left( \log n \right)^{4/5} \right) \right)
\]

\[
\leq O(n) \left( \sum_{i=0}^{T-1} (n_{i,\leq} + 1 + n_{i,-}) + n_T \right) \cdot \left( 2 \log n + O \left( \left( \log n \right)^{4/5} \right) \right) + O \left( \frac{n}{\log n} \right)
\]

\[
\leq O(n) + n \left( 2 \log n + O \left( \left( \log n \right)^{4/5} \right) \right) + O \left( \frac{n}{\log n} \right)
\]

\[
\leq 2n \log n + O \left( n \left( \log n \right)^{4/5} \right).
\]

For the number of moves, we get

\[
M(n) \leq \sum_{i=0}^{T-1} \varepsilon n_i \left( 13 + \varepsilon \right) n_{i,\leq} + \sum_{i=0}^{T-1} 2n_{i,-} + \sum_{i=0}^{T-1} O \left( \frac{n}{\log n} \right)
\]

\[
+ (11 + \varepsilon)n_T
\]

\[
\leq \varepsilon n \left( \sum_{i=0}^{T-1} (n_{i,\leq} + 1 + n_{i,-}) + n_T \right) \cdot (13 + \varepsilon) + O \left( \frac{n}{\log n} \right)
\]

\[
\leq \varepsilon n + n \cdot (13 + \varepsilon) + O \left( \frac{n}{\log n} \right)
\]

\[
\leq (13 + \varepsilon)n,
\]

where \(\varepsilon > 0\) is an arbitrarily small, but fixed, real constant. The above analysis did not include the costs of the initial building of pointer memory. However, by Lemma 5.7, this can be done with only \(O(n)\) comparisons and \(O(n/\log n)\) moves, and hence the bounds displayed above represent the total computational costs of the algorithm. \(\square\)
5.3.3 Handling short blocks

The algorithm presented above needs a procedure capable of sorting blocks of small lengths, namely, with $m \leq 2^{16} = 65536$. This is required, among others, to sort blocks $A_k$ that are short. We could sweep the problem under the rug by saying that “short” blocks can, “somehow,” be sorted with $O(1)$ comparisons and moves, since they are of constant lengths. However, the upper bounds presented by Theorem 5.2 in Section 5.2.12 require some more details, especially for $(1+\varepsilon)m$, the number of moves. Last but not least, these lengths are important in practice.

One of the possible simple solutions is to use our version of Heapsort, with 5 roots and internal nodes having 5 sons. Using the analysis presented in Section 5.3.1, devoted to building a pointer memory, for $t = 5$, $m \leq 2^{16}$, and hence for at most $q \leq 1 + |\log m| \leq 7$ levels, one can easily verify that we shall never use more than $2m \log m + 6.25m$ comparisons or $9.75m$ moves.

5.3.4 An alternative solution

As pointed out at the end of Section 5.2.8, devoted to extracting sorted elements from segments, we could use a heap structure with four levels, instead of five, in a segment. This slightly reduces the number of moves, but increases the number of comparisons. The detailed argument parallels the proof of Theorem 5.3.

**Corollary 5.1** The given sequence, consisting of $n$ elements, can be sorted in-place by performing at most $6n \log n + o(n \log n)$ comparisons and $(12+\varepsilon)n$ element moves, where $\varepsilon > 0$ denotes an arbitrarily small, but fixed, real constant.

5.4 Conclusion

We have described the first space optimal sorting algorithm performing $O(n \log n)$ comparisons and $O(n)$ element moves in the worst case, which closes a long-standing open problem. This presentation is based on the papers [Franceschini and Geffert, 2003] (FOCS 2003) and [Franceschini and Geffert, 2005] (Journal of the ACM, to appear).

However, the algorithms presented in Theorem 5.3 and Corollary 5.1 do not sort stably, since the order of buffer elements may change. If some elements used in buffers are equal, their original order cannot be recovered. Moreover it seems impossible to stabilize this algorithm or even think to a stable algorithm using an approach correlated to this. That is because of the “full-power” scheme used for the internal buffering (see Chapter 4). $O(n)$ buffer elements are used in the algorithm and there seems to be no way to keep track of their initial order without using stolen bits and therefore breaking the constraint for the number of moves performed.

This leaves us with a fascinating question:
Does there exist an algorithm operating in-place and performing, in the worst case, at most $O(n \log n)$ comparisons, $O(n)$ moves, $O(n \log n)$ arithmetic operations, and, at the same time, sorting elements stably, so that the relative order of equal elements is preserved?
Chapter 6

Stability is Very Difficult to Achieve

Abstract

In this chapter we settle a long-standing open question, namely whether it is possible to sort a sequence of $n$ elements stably (i.e. preserving the original relative order of the equal elements), using $O(1)$ auxiliary space and performing $O(n \log n)$ comparisons and $O(n)$ data moves.

This problem was stated explicitly in [Munro and Raman, 1992] where an in-place but unstable sorting algorithm that performs $O(n)$ data moves and $O(n^{1+\epsilon})$ comparisons was given. Subsequently in [Munro and Raman, 1996b] the authors presented a stable algorithm with these same bounds. Recently, an unstable sorting algorithm that matches the asymptotic lower bounds on all computational resources was presented in [Franceschini and Geffert, 2003] (see Chapter 5).

The presentation in this chapter is based on the papers [Franceschini, 2005a] (STACS 2005) and [Franceschini, 2005c].

6.1 Introduction

In the general case of input sequences with repeated elements, a theoretically and practically important requirement for a sorting algorithm is to be stable: the relative order of equal elements in the final sorted sequence is the same found in the original one.

The Heapsort [Williams, 1964] is the first space and comparison optimal sorting algorithm. However, this algorithm is highly unstable and the number of element moves performed in the worst case is $O(n \log n)$. Trabb Pardo [Trabb Pardo, 1977] proved the existence of a sorting algorithm that is stable, comparison and space optimal introducing the first stable in-place merging algorithm. If we consider the
partition-based approach, the ordinary Quicksort [Hoare, 1962] is only space optimal. An unstable, comparison and space optimal sorting can be derived using an in-place selection algorithm like [Lai and Wood, 1988]. Other examples are in [Durian, 1986, Bing-Chao and Knuth, 1986, Wegner, 1987]. A stable, comparison and space optimal version of the Quicksort has been proposed in [Katajainen and Pasanen, 1994]. If we give priority to the number of data moves, the classical selection sort operates in-place, and performs $O(n)$ moves and $O(n^2)$ comparisons in the worst case but it is not stable. The improvement in [Munro and Raman, 1992] with a generalization of Heapsort performing $O(n^{1+\epsilon})$ comparisons, maintains the instability. A stable algorithm with these same bounds was presented in [Munro and Raman, 1996b].

If we relax the space optimality constraint, the address-table sort [Knuth, 1973] can be easily modified to achieve the stability.

The algorithm described in [Katajainen and Pasanen, 1999] is the first sorting requiring $o(n \log n)$ moves in the worst case while guaranteeing in-placeness and $O(n \log n)$ comparisons. However, the technique of the internal buffering (see Chapter 4) makes that algorithm unstable. Our comparison, move and space optimal algorithm described in [Franceschini and Geffert, 2003] (see Chapter 5) is highly unstable for the same reason.

In this chapter we settle the long-standing open question explicitly stated by Munro and Raman in [Munro and Raman, 1992], namely whether it is possible to sort a sequence of $n$ elements stably, using $O(1)$ auxiliary space, performing $O(n \log n)$ comparisons and $O(n)$ data moves. So far, the best known algorithm for stable in-place sorting with $O(n)$ moves was the one presented by Munro and Raman in [Munro and Raman, 1996b], performing $O(n^{1+\epsilon})$ comparisons in the worst case. The result we will give in this chapter closes the problem. The presentation is based on the papers [Franceschini, 2005a] (STACS 2005) and [Franceschini, 2005c].

### 6.2 The Algorithm in Brief

Two basic techniques are very common when space-efficiency of algorithms and data structures in the comparison model is the objective. The first one is the bit stealing [Munro, 1986] (see Chapter 4): a bit of information is encoded in the relative order of a pair of distinct input elements.

The second technique is the internal buffering [Kronrod, 1969] (see Chapter 4), in which some of the input elements are used as placeholders in order to simulate a working area and permute the other elements at less cost. The internal buffering is one of the most powerful techniques for space efficient algorithms and data structures. However, it is easy to understand how disruptive the internal buffering is when the stability of the algorithm is an objective. If the placeholders are not distinct, the original order of identical placeholders can be lost using the simulated
working area. As a witness of the clash between stability and internal buffering technique, we can cite the difference in complexity between the first in-place linear time merging algorithm, due to Kronrod [Kronrod, 1969], and the first stable one by Trabb Pardo [Trabb Pardo, 1977].

6.2.1 What’s new?

The crucial difference with other space and comparison optimal but unstable sorting algorithms performing a near-optimal or optimal number of moves, like [Katajainen and Pasanen, 1999] or [Franceschini and Geffert, 2003], is in how the internal buffering technique is used. In those algorithms a large internal buffer with \( O(n) \) elements has to be found. In order to have such a large buffer, the internal buffering process is iterated \( O(\log n) \) times, halving the size of the current sub-problem, until the current it becomes so small that it can be treated without internal buffering.

The problem in this process is that it ignores completely the characteristics of the input sequence, namely, the number of distinct elements. If the sequence has a very limited number of distinct elements, the conventional approach for internal buffering cannot do anything good. On the other hand, it is probable that this extreme characteristic of the sequence can be exploited in some unconventional bottom-up way.

Our algorithm follows an approach that can be well synthesized as “adaptive”. Using some new sophisticated techniques, we introduce a sorting method that adapts to the number \( d \) of distinct elements of the input sequence. In particular, for what concerns the harder case where the sequence has only a small number of distinct elements, that method allows us to sort a sequence using two kinds of internal buffers:

- An internal buffer with only \( O(d) \) distinct elements is needed to sort the whole sequence stably and within our resource bounds. That requires an efficient algorithm to extract a set of \( O(d) \) distinct elements from the input sequence.

- An internal buffer with \( O(n) \), not necessarily distinct, elements. The original order of this large buffer can be completely recovered after it has been used. As we will see, this normally unachievable task (as in [Katajainen and Pasanen, 1999] and [Franceschini and Geffert, 2003]) is a direct consequence of the small number of distinct elements in the sequence.

Our strategy for the development of a stable sorting algorithm matching the asymptotic lower bounds on all the computational resources can be synthesized in four major points.

6.2.2 Stealing bits

In (Section 6.3), we show how to extract from the input sequence \( \Theta \left( \frac{n}{\log n} \right) \) pairs of two distinct elements each stably and within our computational bounds. They will
be used for encoding purposes with the basic technique of bit-stealing. We make use of the stable in-place selection and the stable in-place partitioning algorithms described in [Katajainen and Pasanen, 1994, 1992]. We obtain a “slow” auxiliary encoding memory that we will use in the rest of the chapter to sort the remaining $m = O(n)$ elements laid down in sequence $A$. Finally, after $A$ is sorted, the $O \left( \frac{n}{\log n} \right)$ elements devoted to information encoding will be sorted using the normal in-place stable Mergesort.

### 6.2.3 Extracting distinct elements

In (Section 6.4) we show how to extract as much distinct elements as we can from the $m = O(n)$ elements in the sequence $A$ left to be sorted at the end Section 6.3. We start gathering the elements with rank less than or equal to $s = O \left( \frac{n}{\log^3 n} \right)$ from the other ones. Let the resulting sequence be $A' A''$. This can be done stably and within our target bounds using once again the stable in-place selection and partitioning algorithms proposed in [Katajainen and Pasanen, 1994, 1992]. Let $d$ be the number of distinct elements among the ones in $A''$ (that is the elements with rank greater than $m - s = \Omega \left( \frac{n}{\log^3 n} \right)$).

Then, we show how to extract $b = O \left( \min \left( d, \frac{n}{\log^3 n} \right) \right)$ distinct elements from $A''$, stably and within our computational bounds, by exploiting the elements in $A'$. Basically, we grow a data structure (encoding its auxiliary data in the auxiliary encoding memory built in the previous point) at the left end of the space, where all the elements of $A'$ initially reside. The structure is built while we scan $A''$ from left to right for distinct elements (their distinctness being evaluated using the structure built so far). The structure is basically a semi-dynamic dictionary optimally searchable but with a slower insertion time. When an element of $A''$ qualifies for the structure, it is exchanged with the first (leftmost) available element in $A'$ (that is not belonging to the structure) and the structure is updated. At the end of the scan we have to extract the elements in $A''$ that have been used as placeholders (as in the internal buffering technique) when a new element was inserted in the structure. For the sake of stability, we have to extract those scattered elements maintaining the original order that occurrences of the same kind and had before the construction of the structure. In the end we will have $b$ distinct elements in a subsequence $B$ at the right end of the space (the set of distinct element grows in left end of the space and it is finally moved when it is complete). We will be left with the problem of sorting the subsequence $C$ with the remaining elements from $A''$ by exploiting the buffer of distinct elements $B$ (and of course the set of encoded bits stolen in the Section 6.3. After $C$ is sorted, the $b = O \left( \min \left( d, \frac{n}{\log^3 n} \right) \right)$ buffer elements in $B$ can be sorted using the normal in-place Mergesort (we do not even need the stable one, since $B$ contains distinct elements).
6.2.4 Sorting in presence of many distinct elements

In (Section 6.5), we show how to proceed when the subsequence $A^*$ from the previous point has “many” distinct elements, meaning $\Omega \left( \frac{n}{\log^2 n} \right)$ distinct elements.

We divide our sorting problem in $O(\log^2 n)$ sub-problems of size $O \left( \frac{n}{\log^2 n} \right)$ and show how to solve those sub-problems assuming the availability of a sufficient number of distinct elements to be used as placeholders, that is in case $b = \Theta \left( \frac{n}{\log^2 n} \right)$. Those buffer elements are the one in $B$ obtained from Section 6.4.

Basically, we introduce a structure that can sort a set of $O(b)$ elements in $O(n \log n)$ comparisons and $O(n)$ moves stably by exploiting $B$ and the auxiliary encoded memory respectively from the Section 6.4 and 6.3. As in the buffer extraction procedure of Section 6.4, this structure may be seen as a semi-dynamic dictionary but in that case we have almost completely opposite targets. When we extract distinct elements, we want a structure totally compact with efficient search but with possibly slow insertion. When we sort $C$ in presence of many distinct elements, we want a structure that is not compact, as it exploits the set of distinct buffer elements, but that has an efficient insertion, in particular for what concerns the complexity bound on the number of moves (we need $O(1)$ amortized moves for each insertion).

After we used the elements in $B$ and the structure to build $O(\log^3 n)$ runs of sorted elements out of the original sequence $C$ left to be sorted from Section 6.4, we have to merge these runs stably and within our computational bounds. To this purpose, we introduce a multi-way stable merging technique requiring a very limited amount of placeholders to deliver the final sorted sequence of $O(n)$ elements.

6.2.5 Sorting in presence of few distinct elements

In (Section 6.6), we show how to deal with the hardest case, namely when, after the extraction of the buffer elements in the second point, we are left to sort a sequence $C$ with “few” distinct elements, meaning a sequence with $b = o \left( \frac{n}{\log^2 n} \right)$ distinct elements. First, we partition the sequence into three zones $C'YC''$ around a pivot element (the occurrences of the same kind of the pivot will be in zone $Y$). That is done because we are going to need an efficient way to distinguish between the sequence of active elements $V$ (that will be impersonated first by $C'$ and then by $C''$) and two types of buffer elements, a set (in $B$) of few but distinct buffer elements and another set $G$ of many but not necessarily distinct elements (that will be impersonated first by $C''$ and then by $C'$).

Then, we show how to exploit the scarcity of distinct elements in the sequence $V$ grouping the identical elements laying in sub-zones of size $O(b \log^2 n)$ and how to acquire and encode in the auxiliary encoding memory from Section 6.3 a linked structure traversing the groups of clustered equal elements in $V$ in sorted order.

Finally, we show how to use the groups and the encoded linked structure to permute first $C'$ using $YC''$ as working zone and then $C''$ using $C'Y$ without disrupting
the (sorted) order of $C'$.

6.3 Stealing Bits

As we mentioned in the introduction, with the bit-stealing technique (see [Munro, 1986]) the value of a bit is encoded in the relative order of two distinct elements (e.g. the increasing order for 0 and the decreasing order for 1). In this section we show how to collect $O(n/\log n)$ pairs of distinct elements, stably and within our computational bounds.

The rank of an element $x_i$ in a sequence $S = x_1 \ldots x_t$ is the cardinality of the multiset

$$\{x_j \in S \mid x_j < x_i \text{ or } (x_j = x_i \text{ and } j \leq i)\}. $$

The rank of an element $x$ in a set $S$ is similarly defined. Let $r = \lceil n/\log n \rceil$ and let $\pi'$ and $\pi''$ be, respectively, the element with rank $r$ and the element with rank $n-r+1$ in the input sequence. We want to partition stably and in-place the input sequence in five zones $J'P'A''P''J''$ such that, for each $j' \in J'$, $p' \in P'$, $a \in A$, $p'' \in P''$ and $j'' \in J''$, we have that

$$j' < p' = \pi' < a < p'' = \pi'' < j''.$$

That can be done in $O(n)$ comparisons and $O(n)$ moves using the stable in-place selection and the stable in-place partitioning of Katajainen and Pasanen [Katajainen and Pasanen, 1994, 1992].

Zones $J'$ and $J''$ can be sorted stably and in-place in $O(n)$ time simply using the stable in-place mergesort (e.g. [Salowe and Steiger, 1987]). If there are no elements in $A$, we are done since the input sequence is already sorted. Otherwise we are left with the unsorted subsequence $A$ and with a set $M$ of $r = O(n/\log n)$ pairs of distinct elements, that is

$$M = \{(Q'[1], Q''[1]), (Q'[2], Q''[2]), \ldots, (Q'[r], Q''[r])\},$$

where $Q' = J'P'$ and $Q'' = P''J''$.

The starting addresses of $Q'$ and $Q''$ can be maintained in two locations of auxiliary memory (we can use $O(1)$ auxiliary locations) and so, for any $i$, we can retrieve the addresses of the elements of the $i$-th pair in $O(1)$ operations. Therefore, we can view $M$ as a sequence of encoding words of $t$ bits each, for any $t$. Those encoding words can be used pretty much as if they were normal ones. We have to pay attention to the costs of using encoding bits or encoding words, though: reading an encoding word of $t$ bits takes $t$ comparisons, changing it costs $t$ comparisons and $O(t)$ data moves in the worst case or $O(1)$ moves amortized if we perform a sufficiently long sequence of increments by one (see [Cormen, Leiserson, Rivest, and Stein, 2001], the binary counter analysis). Moreover, we could have chosen the ranks of $\pi'$ and $\pi''$ as
cr and $n - cr + 1$ for any constant $c$, so that the number of encoded bits would be $cr$ without changing the asymptotic bounds of the algorithm.

Therefore, if $m$ is the size of $A$, we can make the following assumption:

**Assumption 6.1** We can use an auxiliary encoding memory $\mathcal{M}$ consisting of $O\left(\frac{r}{\log m}\right)$ words of $\lfloor \log m \rfloor$ encoding bits each and with the following cost model. For any word $w$, for any $q \leq \lfloor \log m \rfloor$ and for any group $g$ of $q$ bits of $w$:

- retrieving the values of all the bits in $g$ requires $q$ comparisons;
- changing the values of all the bits in $g$ requires $\Theta(q)$ moves.

Hence, if we are able to solve the following problem over the sequence $A$, we are able to solve the original problem.

**Problem 6.1** Under Assumption 6.1, sort the sequence $A$ of $m$ ($< n$) elements stably, using $O(1)$ locations of auxiliary memory, performing $O(m \log m)$ comparisons and $O(m)$ data moves.

In the following sections, we will use the auxiliary encoding memory $\mathcal{M}$ as normal auxiliary memory for numeric values (of course each use of $\mathcal{M}$ must be correctly accounted in the complexity analysis). We will declare explicitly any new auxiliary data (indices, pointers...) stored in $\mathcal{M}$.

### 6.4 Extracting a Set of Distinct Elements

In this section we show how to go from sequence $A$ to $J'' P''' CB$ such that:

**Property 6.1**

(i) For any elements $j \in J''$, $p \in P'''$ and $q \in CB$, we have that $j < p < q$.

(ii) $J''$ is in sorted order.

(iii) The element with rank $\lceil r/\log^2 m \rceil + 1$ in $A$ is in $P'''$ together with all the other occurrences of its kind.

(iv) Let $d$ be the number of distinct elements in $CB$, $B$ contains

$$b = \min(d, \lfloor |CB| / \log^3 m \rfloor) = O(\lfloor r/\log^2 m \rfloor)$$

distinct elements.

(v) Any two equal elements in $J'' P''' CB$ are in the same relative order they were in $A$. 
After we show how to obtain the new sequence $J''' P^n C B$ satisfying Property 6.1 within our target bounds, we will be left with the problem of sorting $C B$. The elements in $B$ will be used in Sections 6.5 and 6.6 as in the technique of internal buffering [Kronrod, 1969]. Basically, some of the elements are used as placeholders to simulate a working area in which the other elements can be permuted efficiently. If the placeholders are not distinct, the stability of the algorithm becomes a difficult task since the original order of identical placeholders can be lost using the simulated working area. The elements in $B$ are distinct so we do not have to worry, as long as we can sort $O(|C|)$ elements with $o(|C|)$ placeholders (Section 6.5). However, as we will see in Section 6.6, if $|B|$ is too small, we have to deal with bigger buffers whose internal order has to be preserved entirely, not only the relative order of equal elements.

6.4.1 Main cycle of the buffer extraction

We first present the main cycle of the algorithm for the creation of $J''' P^n C B$. The main cycle depends on a complex structure that we will introduce later in Section 6.4.2.

Before we start, let us recall the basic technique for space-efficient block exchange. From a block $X = x_1 \ldots x_t$ of $t$ consecutive elements we can obtain the reverse $X^R = x_t \ldots x_1$ in linear time and in-place simply exchanging $x_1$ with $x_t$, $x_2$ with $x_{t-1}$ and so forth. Two consecutive blocks $X$ and $Y$, possibly of different sizes, can be exchanged in-place and linear time with three block reversals, since $Y X = (X^R Y^R)^R$.

The main cycle of the buffer extraction procedure has three phases.

6.4.1.1 First phase: collecting some placeholders

In the first phase we extract some elements, possibly non-distinct, that will help in the process of collecting the set of distinct elements that will reside in $B$.

First, we select the element of rank $[r / \log^2 m] + 1$ in $A$. Then, we partition $A$ according to that element.

We obtain a new sequence $A' P''' A''$ that clearly satisfies points (i) and (iii) in Property 6.1, with $J''' = A'$ and $C B = A''$. The selection and the partitioning can be done in-place and stably using once again the linear time algorithms proposed by Katajainen and Pasanen [Katajainen and Pasanen, 1994, 1992]. Therefore, point (v) in Property 6.1 is also satisfied. If $A''$ is void, we sort $A'$ using the in-place stable Mergesort and we are clearly done. Otherwise, we leave $A'$ as it is and we proceed with the second phase.
6.4.1.2 Second phase: collecting the distinct elements

Throughout this phase, we will continue to denote with $A$ the evolving sequence of $m$ elements. We have that $A = A'P^m A''$ right after the first phase. Let us denote with $h$ the index of the rightmost location of $P^m$ (that is $A[h] = P^m[|P^m|]$).

We maintain two indices $i$ and $i'$ initially set, respectively, to 1 and $m$. The following steps are repeated until $i > \lfloor |A''|/\log^3 m \rfloor$ or $i' = h$:


2. If $A[i']$ is not in $A[1 \ldots i - 1]$, exchange $A[i']$ and $A[i]$, **PROCESS($A[1 \ldots i]$) and increase $i$ by one.

3. Decrease by one $i'$.

At the end of this second phase, we have collected $b = \min(d, \lfloor |A''|/\log^3 m \rfloor)$ distinct elements in $A[1 \ldots b]$ ($d$ is the number of distinct elements in $A''$). How the procedures **SEARCH** and **PROCESS** work will be explained in Section 6.4.2.

6.4.1.3 Third phase: collecting the placeholders back

After the second phase, the first $b$ elements residing in $A'P^m$ at the end of the first phase are scattered in the subsequence $A[h + 1 \ldots m]$. Therefore, point (v) in Property 6.1 is no longer satisfied by the current sequence $A$. We have to collect them back.

First, we partition the subsequence $A[h + 1 \ldots m]$ according to $A[h]$. Let $CA''$ be the resulting sequence, where for any $a \in A''$ and $c \in C$, we have that $a \leq |h| < c$. We use once again the linear time, stable partitioning algorithm described in [Katajainen and Pasanen, 1992].

Then, we reverse $A''$, recovering the original order holding before the second phase, we sort it using the stable in-place Mergesort and we exchange it with $A[1 \ldots b]$. After that, the resulting sequence $A$ respects all the points in Property 6.1.

**Lemma 6.1** Under Assumption 6.1, the buffer extraction algorithm operates in-place, Property 6.1 holds for the resulting sequence, and the comparisons and moves performed are, respectively, $O(mX_s + X_p + m)$ and $O(Y_p + m)$, where

- $X_s$ upper bounds the number of comparisons of each invocation of **SEARCH** in Step 1,

- $X_p$ and $Y_p$ are, respectively, the total number of comparisons and moves performed by the $b$ invocations of **PROCESS** in Step 2.
Proof: First phase. We apply the stable, in-place, linear time selection and partitioning algorithm proposed in [Katajainen and Pasanen, 1994, 1992]. If we have to sort already the elements in \( A' \) (because nothing has to be done for \( A'' \)), we can use the normal stable, in-place Mergesort still staying within out bounds since \( |A'| = O(m/\log^3 m) \).

Second phase. The cycle is iterated \( O(m) \) times, hence the total cost of the invocations of SEARCH is \( O(mX) \) comparisons. In one iteration, Step 2 can be executed \( O(r/\log^2 m) \) times and, excluding the costs of PROCESS, its complexity is \( O(1) \) and therefore the total cost of Step 2 is \( o(r) = o(m) \) comparisons and moves. Step 3 contributes with another \( O(m) \) arithmetic operations.

Third phase. It consists simply in one application of the partitioning algorithm in [Katajainen and Pasanen, 1992], a constant number of applications of block reversing and exchanging and the final application of the stable, in-place Mergesort to the first \( b = O(m/\log^3 m) \) elements in \( A \).

\[ \square \]

6.4.2 Managing a growing set of distinct elements compactly

In this section we describe the structure we use to perform efficiently the operations SEARCH and PROCESS in Steps 1 and 2 of the second phase of the buffer extraction algorithm.

The structure has two levels:

- the \textit{routing level}, that directs the searches, and
- the \textit{collection level}, that contains the majority of the elements in the structure.

First we give the solution to an abstract problem. Then we describe the structure and, in particular, how to reduce the managing of the routing level to an instance of the abstract problem. The abstract problem (and its solution) will come in handy again in Section 6.5 when we describe another two-level structure but with different target bounds and characteristics than the one in this section.

6.4.2.1 Abstract problem.

We want to solve the following abstract problem.

Problem 6.2 We are given two disjoint sets: \( \mathcal{R} \) with routing elements and \( \mathcal{F} \) with filler elements. The following hypotheses hold:

(i) Routing and filler elements belong to the same, totally ordered, possibly infinite universe.

(ii) At any time we are presented with a new routing element to be included in \( \mathcal{R} \).

(iii) At most \( \rho < m \) elements will be included in \( \mathcal{R} \).
(iv) At the beginning $|\mathcal{R}| = 1$, the unique routing element is in the first location followed by the fillers.

(v) At any time $\frac{|\mathcal{F}|}{|\mathcal{R}|} > \log \rho$. The possible growth of $F$ is not a concern, new filler elements will eventually be added after the current last element.

(vi) We can use an auxiliary memory $M$ of $O(\rho)$ words of $\log \rho$ bits each to store auxiliary data.

The task is to manage the growth of $\mathcal{R}$ so that the following properties hold:

(a) At any time $\mathcal{R}$ and $\mathcal{F}$ are stored in a zone $Z$ of $|\mathcal{R}| + |\mathcal{F}|$ contiguous memory locations.

(b) At any time $\mathcal{R}$ can be searched with $O(\log |\mathcal{R}|)$ comparisons and a constant number of accesses to $M$.

(c) When $\mathcal{R}$ is complete, the total number of comparisons, moves and accesses to $M$ performed is $O(|\mathcal{R}| \log^2 \rho)$.

**How to solve Problem 6.2.** We maintain in $Z$ a sequence $S$ of contiguous segments containing $\sigma = O(\log \rho)$ elements each. Let $S = S_1S_2\ldots S_{i-1}S_i$ be the sequence of segments at a given time.

A segment can contain both routing and filler elements. A segment with $i$ routing elements has the following structure:

$$a_1a_2\ldots a_{i-1}a_if_1f_2\ldots f_{i-i},$$

where any $a_j$ is a routing element and any $f_j$ is a filler element. There is at least one routing element for each segment.

For any segment we have to be able to discern between its routing and filler elements. Standing on the sole hypotheses of Problem 6.2, we must assume that it is not possible to classify any element in $\mathcal{R} \cup F$ as routing or filler simply by inspection. Therefore any segment $S_i$ is associated with an integer counting the number of routing elements in it. The buffer separator technique used in [Franceschini and Geffert, 2003] cannot be used for this purpose because, as we will see, in at least one instance of the abstract Problem 6.2 there is no relation between the filler elements and the routing elements of a segment.

As we will see shortly, the number of segments will always be a power of 2. The routing elements are maintained in sorted order throughout the whole $S$, that is the routing elements of every segment are in sorted order and, for any two routing elements $a', a''$ such that $a' \in S_i, a'' \in S_j$ and $i < j$, we have that $a' \leq a''$.

A routing element can be easily searched in $O(\log |\mathcal{R}|)$ comparisons and a constant number of accesses to $M$: first, do a binary search over the first elements of
the segments (that are all routing elements); then read in $M$ the number of routing elements of the only segment selected with the previous step and search in it. Therefore, property (b) of Problem 6.2 holds.

While inserting new routing elements, the invariants on $S$ can be maintained with a variation of the well-known, density based algorithm described in [Itai, Konheim, and Rodeh, 1981]. The nodes of an implicit binary tree are associated with subsequences of $S$. The root of the tree is associated with the whole sequence $S$. The left child of the root is associated with $S_1S_2\ldots S_{t/2-1}S_{t/2}$, the right child with $S_{t/2+1}S_{t/2+2}\ldots S_{t-1}S_t$ and so forth (remember the number of segments will always be a power of 2). Therefore, there is a leaf for each segment. A node $v$ has two attributes: the level $l(v)$ (the leaves are at level 0, the root is at level $\log t$) and the number $d(v)$ of routing elements contained in the subsequence associated with $v$. Each level has a threshold: level $i$ has threshold $\tau_i = \sigma - i$. The tree has $2t - 1 \leq 2|R| \leq 2\rho$ nodes and therefore all the attributes of the nodes of the tree can be stored in $M$ (actually, they can also be calculated at rebalancing time, without storing or encoding anything but we do not need to do this).

When a new routing element $a$ is inserted in the proper segment

$$S' = a_1a_2\ldots a_{i-1}a_ia_ig_2\ldots g_{\sigma-i}$$

containing $i < \sigma$ routing elements, the first filler element $g_1$ is moved after the current last element of $Z$ and the process ends with the segment

$$S' = a_1a_2\ldots a_jaa_{j+1}a_{i-1}a_ig_2\ldots g_{\sigma-i},$$

for some $j$.

Otherwise, if $S'$ is full, we find the lowest ancestor $v$ of the leaf associated with $S'$ such that $d(v) \leq 2^{l(v)} \cdot \tau_l(v)$ and redistribute the routing elements evenly in the subsequence associated with $v$. That costs $O(2^{l(v)} \cdot \tau_l(v))$. If not even the root of the tree satisfies the condition, then the number of segments is doubled (so it is always a power of 2), a new implicit tree with a larger root is used and the redistribution can be performed. By hypothesis (v) in Problem 6.2 we know that, for any $|R|$, there are sufficient filler elements to create the new $t$ segments. Therefore, property (a) of Problem 6.2 holds.

Finally, with a simple analysis, similar to the one in [Itai, Konheim, and Rodeh, 1981], it can be proved that property (c) of Problem 6.2 holds. After the redistribution of the elements in the subsequence associated with $v$, for each descendant $u$ of $v$ we have that $d(u) \leq 2^{l(v)} \cdot \tau_l(v)$, in particular that holds for the children of $v$. Before the rebalancing, there was a child $u'$ of $v$ such that $d(u') > 2^{l(u')} \cdot \tau_l(u')$.

Therefore, before $v$ needs to be rebalanced again there will have to be at least

$$2^{l(u')} (\tau_l(u') - \tau_l(v))$$

insertions in the subsequence associated to any child $u''$ of $v$. 
Since the rebalancing of $v$ cost $O(2^{l(v)} \cdot \tau_{l(v)})$, we have that the amortized cost relatively to level $l(v)$ of the insertion that triggered the rebalancing is

$$O \left( \frac{2^{l(v)} \cdot \tau_{l(v)}}{2^{l(v')} \cdot \left( \tau_{l(v')} - \tau_{l(v)} \right)} \right) = O(\tau_{l(v)}).$$

Since there are $\log t$ levels, the complete amortized cost is $O(\sigma \log t) = O(\log \rho \log t) = O(\log^2 \rho)$. Therefore we can conclude that Problem 6.2 is solved.

### 6.4.2.2 The structure

In order to manage the growth of the set of the buffer elements in the second phase of the buffer extraction algorithm, we have to solve the following problem.

**Problem 6.3** Under Assumption 6.1, we have to handle the growth of a set $B$ of at most $\lfloor r/\log^2 m \rfloor = O(m/\log^3 m)$ distinct elements so that the following properties hold:

(a) At any time, $B$ is stored in $|B|$ contiguous memory locations.

(b) At any time, $B$ can be searched with $O(\log m)$ comparisons.

(c) When $B$ is complete, the total number of comparisons and moves performed is $O(|B| \log^3 m)$.

Let us give our solution for Problem 6.3. Let $B$ be the (growing) zone of the memory in which $B$ will be maintained. The auxiliary encoding memory $M$ has $O(\frac{r}{\log m})$ words of $\log m$ bits each. Since $|B| \leq \lfloor r/\log^2 m \rfloor$, it is easy to associate with every $x \in B$ a constant number $h$ of words of auxiliary data. We can allocate in $M$ an array $P$ of $\lfloor r/\log^2 m \rfloor$ entries of $h$ words each and maintain the following invariant:

\begin{equation}
\text{The element in position } i\text{-th in } B \text{ has its auxiliary data stored in the } i\text{-th entry of } P. \tag{6.1}
\end{equation}

For the sake of description, we will skip that kind of details in the algorithm and implicitly assume that an element is always moved together with its encoded $O(1)$ words of auxiliary data (of course we will consider the cost model for $M$ in the analysis).

At any time, $B$ is divided into two contiguous zones $R$ and $H$ ($B = RH$). The elements of the routing level are in $R$ (together with some elements of the collection level that will act as fillers, as we will see).
Buckets. Each routing element \( a \) is associated with a set of elements \( \beta(a) \) in the collection level that we will call a bucket. Let \( a' \) and \( a'' \) be two consecutive (in the sorted order) routing elements: for each \( x \in \beta(a') \) we have that \( a' < x < a'' \). For what concerns the number of elements in a bucket, we have that

\[
4 \lceil \log m \rceil \leq |\beta(a)| \leq 8 \lceil \log m \rceil.
\]

Let us focus on a single bucket \( \beta \) and let \( \text{even}_\beta \) and \( \text{odd}_\beta \) be, respectively, the set of the elements of \( \beta \) with even and odd rank. The elements in \( \text{odd}_\beta \) are stored in sorted order in a contiguous zone of memory in \( H \) while the elements in \( \text{even}_\beta \) are stored in \( R \) and may be scattered. They will play a role similar to the one of buffer elements but more powerful because they will be searchable at any moment of the lifetime of the structure.

Pointers are used to keep track of the elements in \( \text{even}_\beta \): each \( d' \in \text{odd}_\beta \) has a pointer to its successor \( \text{succ}(d') \in \text{even}_\beta \) and \( \text{succ}(d') \) has a pointer to \( d' \) (obviously, if \( |\beta| \) is odd, the successor of the largest element in \( \text{odd}_\beta \) does not belong to the set \( \text{even}_\beta \)).

The routing element \( a \) associated with \( \beta \) has a pointer to the location of \( \text{odd}_\beta \) and \( \text{odd}_\beta \) has a pointer to \( a \). Assuming that we are able to maintain the layout we just introduced for the buckets, we can show a way to satisfy property (b) of Problem 6.3.

Lemma 6.2 A bucket can be searched with \( O(\log m) \) comparisons.

Proof: Searching for an element \( u \) in a bucket \( \beta \) is straightforward. First we search in \( \text{odd}_\beta \). If \( u \not\in \text{odd}_\beta \), let \( o \in \text{odd}_\beta \) be the predecessor of \( u \) in \( \text{odd}_\beta \). We access \( \text{succ}(o) \) using the pointer of \( o \). If \( \text{succ}(o) \) is not equal to \( u \), the search ends. In total we have to access \( O(1) \) words of auxiliary information in \( \mathcal{M} \). From Assumption 6.1, the thesis follows. \( \square \)

Sub-zones. \( H \) has to accommodate the set \( \text{odd}_\beta \) for any bucket \( \beta \). By the upper and lower bounds for the number of elements in a bucket, we know that for any \( \text{odd}_\beta \)

\[
2 \lceil \log m \rceil \leq |\text{odd}_\beta| \leq 4 \lceil \log m \rceil.
\]

All the \( \text{odd}_\beta \) of size \( i \) are maintained in a contiguous sub-zone \( H_{i-2 \lceil \log m \rceil + 1} \) of \( H \) (the sub-zone \( H_j \) contains sets of size \( j + 2 \lceil \log m \rceil - 1 \)). Therefore, there are \( z = 2 \lceil \log m \rceil + 1 \) sub-zones. We have that

\[
H = H_1 H_2 \ldots H_{z-1} H_z,
\]

that is the sub-zones are in increasing order by the size of sets they contain.
For any $H_i$, the first set $\text{odd}_\beta$ may be rotated, that is it may have its first $i + 2 \lfloor \log m \rfloor - 1 - j$ elements at the right end of $H_i$ and the last $j$ at the left end, for any $j$ called index of rotation of $H_i$.

Some sub-zones may be void. For any zone, we store in $\mathcal{M}$ its starting address in $H$ and its index of rotation. If the indices of rotation are known, the particular case of a routing element $a$ having a bucket $\beta$ with the set $\text{odd}_\beta$ in the first position of its sub-zone can be treated simply (e.g., with an extra flag for each routing element to recognize the particular case).

**Basic operations.** We are going to need two basic operations on the sub-zones: $\text{Slide\_by\_one}(i)$ and $\text{Move\_back}(o)$.

- With $\text{Slide\_by\_one}(i)$ all the sub-zones $H_j$ with $j \geq i$ are rotated by one position to the right (assuming that there is a free location at the right end of $H$). The execution of this operation is obvious.

- With $\text{Move\_back}(o)$ the set $o$ of size $4 \lfloor \log m \rfloor$ (the maximum size possible) is moved from $H_z$ to $H_1$.

  1. $o$ is exchanged with the second set in $H_z$.
  2. $o$ is exchanged with the portion of the first set in $H_z$ residing at the left end ($H_z$ can have rotation index > 0).
  3. For each $H_i$, $2 \leq i \leq z - 1$, $o$ is exchanged with the first $4 \lfloor \log m \rfloor$ elements of $H_i$.
  4. $o$ is exchanged with the portion (if any) of the first set of $H_1$ residing at the right end ($H_1$ can have rotation index > 0 too).

**Lemma 6.3** The operations $\text{Slide\_by\_one}(i)$ and $\text{Move\_back}(o)$ can be executed with $O(\log^3 m)$ moves and comparisons.

**Proof:** In $\text{Slide\_by\_one}(i)$ we have to access and modify $\Theta(z - i)$ pointers in $\mathcal{M}$ storing the rotation indices of the sub-zones involved. That requires $\Theta((z - i) \log m) = O(\log^2 m)$ moves and comparisons in the worst case. Moreover, we have to update the pointers of $O(z - i)$ buckets whose odd sets are stored in the $\Theta(z - i)$ sub-zones and were rotated before the execution of this operation. Therefore, $\text{Slide\_by\_one}(i)$ requires $O(\log^3 m)$ moves and comparisons in the worst case.

Since the minimum size of a set in $H$ ($2 \lfloor \log m \rfloor$) is of the same order of the maximum size ($4 \lfloor \log m \rfloor$), in $\text{Move\_back}(o)$ we have to access and modify $O(z)$ pointers stored in $\mathcal{M}$. Analogously to the previous case, we have to update the pointers of $\Theta(\log m)$ buckets whose odd sets are moved in this operation. Therefore, $\text{Move\_back}(o)$ requires $O(\log^3 m)$ moves and comparisons in the worst case.  

\qed
Maintaining the invariants for the collection level. Let us show how the invariants on $B$ introduced so far can be maintained when an element $u$ has to be inserted in a bucket $\beta$ associated with a routing element $a$ placed somewhere in zone $R$.

Let us suppose $|\beta| = p < 8[\log m]$, and let $i$ be the rank of $u$ in $\beta \cup \{u\}$. There are two phases: in the first phase we reorganize the space to make room for the new element; in the second phase we rearrange the elements of $\beta$, since the arrival of $u$ may change $odd_\beta$ and $even_\beta$ substantially.

- **Space reorganization.** If $p$ is odd then $even_\beta$ increases by one and $odd_\beta$ remains of the same size. We invoke SLIDE_BY_ONE(1) to free a location between $R$ and $H$ and put $u$ in that location temporarily.

Otherwise, if $p$ is even then $odd_\beta$ increases by one and $even_\beta$ remains of the same size.

1. We invoke SLIDE_BY_ONE($p/2 - 2[\log m] + 2$) in order to have a free location between $H_{p/2-2[\log m]+1}$ (the sub-zone that contained $odd_\beta$ before the insertion of $u$ in $\beta$) and $H_{p/2-2[\log m]+2}$ (the new sub-zone of $odd_\beta$ after the insertion) and put $u$ in the free location temporarily.

2. We exchange $odd_\beta$ with the last set in $H_{p/2-2[\log m]+1}$.

3. We exchange $odd_\beta$ with the portion (if any) of the first set of $H_{p/2-2[\log m]+1}$ residing at the right end of the sub-zone.

4. After $odd_\beta$ is joined with $u$, we exchange $odd_\beta \cup \{u\}$ with the portion of the first set in $H_{p/2-2[\log m]+2}$ residing at the left end of the new sub-zone.

- **Rearrangement.** Let $\beta' = \beta \cup \{u\}$, $\Delta_o = \{o \in odd_\beta \mid u < o\}$ and $\Delta_e = \{e \in even_\beta \mid u < e\}$. If the rank of $u$ in $\beta'$ is odd, we have that

$$odd_{\beta'} = \{u\} \cup (odd_\beta - \Delta_o) \cup \Delta_e \quad \text{and} \quad even_{\beta'} = (even_\beta - \Delta_e) \cup \Delta_o.$$ 

Similarly, if the rank of $u$ in $\beta'$ is even, we have that

$$odd_{\beta'} = (odd_\beta - \Delta_o) \cup \Delta_e \quad \text{and} \quad even_{\beta'} = \{u\} \cup (even_\beta - \Delta_e) \cup \Delta_o.$$ 

Given the definition of $odd_{\beta'}$ and $even_{\beta'}$, the rearrangement is a simple sequence of exchanges between an element in $H$ and an element in $R$. For each exchange we have to access one pointer.

Let us suppose $\beta$ is full (before the insertion of $u$). We have to split $\beta \cup \{u\}$ into two buckets $\beta', \beta''$ and the median element $a'$. $\beta'$ and $\beta''$ will contain, respectively, the $4[\log m]$ smallest and the $4[\log m]$ largest elements of $\beta \cup \{u\}$. $a'$ will be a new routing element and its bucket will be $\beta''$. 
• **Space reorganization.** First, we make room for a new routing element invoking \texttt{Slide\_by\_one}(1) and put \( u \) in the free location between \( R \) and \( H \). Second, we invoke \texttt{Move\_back}(odd\(_\beta\)) to move odd\(_\beta\) in \( H_1 \).

• **Rearrangement.** We have to reorganize the disposition of the elements in \( \beta \cup \{u\} \) for the splitting. However, we already know that the starting address of odd\(_\beta\) will be the same of odd\(_\beta\) after \texttt{Move\_back}(odd\(_\beta\)) and the starting address of odd\(_\gamma\) will be \( 2[\log m] \) locations further. If the rank of \( u \) in \( \beta \cup \{u\} \) is \( 4[\log m] + 1 \) then \( u \) is the new routing element to be inserted in \( R \) and we do not have to do any reorganization.

Otherwise, we exchange \( u \) with the element of rank \( 4[\log m] + 1 \) and then we proceed to rearrange the other elements in the same way we do when \( \beta \) is not full. Anyway, by the end of the rearrangement process, we have the new routing element \( a' \) in the location between \( R \) and \( H \), ready to be inserted in the routing level.

When we have to insert an element in a bucket that is full, we split the bucket and we are left with a new routing element to be inserted in the routing level.

**Lemma 6.4** Maintaining the invariants for the collection level costs \( O(\log^3 m) \) moves and comparisons in the worst case.

**Proof:** When \( |\beta| = p < 8[\log m] \), the cost of \texttt{Slide\_by\_one}(1) dominates the complexity of the space reorganization. For what concern the rearrangement, we have to access at most \( O(\log m) \) pointers in \( \mathcal{M} \). Therefore, by Lemma 6.3, when we have to insert an element in a bucket that is not full, we pay \( O(\log^3 m) \) moves and comparisons in the worst case.

When \( \beta \) is full, the cost of the execution of \texttt{Move\_back}(odd\(_\beta\)) dominates the complexity of the space reorganization in this case. For the rearrangement, we have to access at most \( O(\log m) \) pointers in \( \mathcal{M} \). Therefore, by Lemma 6.3, when we have to insert an element in a bucket that is full, we pay \( O(\log^3 m) \) moves and comparisons to split the bucket. \( \square \)

**The routing level.** With the organization of \( RH \) presented so far, we are able to satisfy all the hypotheses in Problem 6.2. As expected, \( \mathcal{R} \) contains the routing elements produced by splitting the buckets and \( \mathcal{F} \) contains the elements in \( \bigcup_{\beta} \text{even}_\beta \).

Hypotheses (i), (ii), (iii) and (iv) are obviously satisfied. For each routing element there is a bucket \( \beta \) with at least \( 4[\log m] \) elements and at least \( 2[\log m] \) of those elements belong to \( \mathcal{F} \), therefore Hypothesis (v) is satisfied. Concerning Hypothesis (vi), we can use the encoded memory \( \mathcal{M} \).

Therefore, the solution to the abstract Problem 6.2 can be used and we are able to manage the growth of the set of routing elements so that the following properties hold:
• At any time, all the routing elements and the ones in $\bigcup_{\beta} \text{even}_\beta$ are maintained compactly in zone $R$.

• At any time, the routing level can be searched with $O(\log m)$ comparisons and a constant number of accesses to $M$ (and hence $O(\log m)$ comparisons in total).

• In the routing level there is a slowdown factor $O(\log m)$ because we use the auxiliary encoded memory. However, we know that

$$|R| < \rho = O \left( \frac{|B|}{\log m} \right).$$

Therefore when the routing level is complete, the total number of comparisons and moves performed building it is $O(|B|\log^2 m) = O(m)$.

Joining those properties and Lemmas 6.2 and 6.4 we can conclude that Problem 6.3 is solved. Therefore, by Lemma 6.1, we have that:

**Theorem 6.1** Under Assumption 6.1, a sequence $J^m P^m CB$ satisfying Property 6.1 can be obtained from $A$, stably, using $O(1)$ locations of auxiliary memory, performing $O(m \log m)$ comparisons and $O(m)$ moves in the worst case.

### 6.5 Sorting with Many Distinct Elements

In this section we show how to sort the subsequence $CB$ of a sequence $J^m P^m CB$ satisfying Property 6.1 and with $b = |B| = |CB|/\log^3 m$. First, in Section 6.5.1, we show how to sort $b$ elements stably, using $O(1)$ auxiliary space, with $O(b \log m)$ comparisons and $O(b)$ moves, under Assumption 6.1 and using another $b$ distinct elements as placeholders (from $B$). Then, in Section 6.5.2, we show how to sort $CB$ using the technique in Section 6.5.1 and a multi-way stable merging technique requiring a very limited amount of placeholders.

#### 6.5.1 Sorting $b$ elements with $b$ distinct placeholders

For the sake of simplicity, let us suppose that 5 divides $b$, that is $b = 5b'$ for an integer $b'$. Let $D$ be the sequence of $b$ elements to be sorted and let $B$ be the sequence of $b$ placeholders. $D$ is divided into five contiguous subsequences of $b'$ elements each, $D = D_1D_2D_3D_4D_5$. Each $D_i$ is sorted using $B$ as we will describe shortly, and the final sorted sequence is obtained merging the subsequences in-place, stably and in linear time (e.g., using the merging algorithm described in [Salowe and Steiger, 1987]).

To sort each $D_i$, we use a structure with the same basic subdivision of the one in Section 6.4.2: a **routing level**, directing the searches, and a **collection level**, containing
the majority of the elements. After all the elements in $D_i$ are inserted, the structure is traversed to move the elements back in $D_i$ in stable sorted order.

### 6.5.1.1 The structure

First, we describe the logical organization of the collection level. Then, we show how to embed the collection level into the internal buffer $B$ and how to store in $M$ its auxiliary data. Finally, we show how to maintain the invariants and how, even in this case, the routing level can be seen as a particular instance of the abstract Problem 6.2.

**The collection level.** Each routing element $a$ is associated with a small balanced search tree $T(a)$ in the collection level. Let $a'$ and $a''$ be two consecutive (in the sorted order) routing elements: for each $x \in T(a')$ we have that $a' \leq x \leq a''$.

Let us consider a generic tree $T$. For any leaf $l$ of $T$ we have that

$$|\log m| \leq |l| \leq 2|\log m|. \quad (6.2)$$

On the other hand, for any internal node $u$ $T$ we have that

$$\left[\sqrt{\log m}\right] \leq |u| \leq 2\left[\sqrt{\log m}\right]. \quad (6.3)$$

$T$ has five levels and hence it contains at least $\log^3 m$ elements.

The elements in any leaf of $T$ are not in sorted order, they are in insertion order: a new element of a leaf is inserted in the last position regardless for its rank among the other elements. There is no auxiliary encoded data associated with any single element of a leaf. Instead, a *bit mask* of $2|\log m|$ bits is associated with the whole leaf, one bit for each possible position, with the expected meaning: the $i$-th bit is equal to one if and only if there is an element in position $i$.

The elements in any internal node $v$ are not in sorted order either. However, a small encoded pointer of $O(\log \log m)$ bits is associated with any of them. Those pointers are used to maintain a small linked list in which the elements of $v$ are maintained in sorted order. There is also a small encoded pointer to the head of the linked list and, of course, an encoded pointer of $O(\log m)$ bits to each child of the node.

**Embedding and encoding.** The internal buffer $B$ is divided into four contiguous zones $RC'C''W$, such that

$$|R| = \| - (2|\log m| + 1)$$

$$|C'| = |C''| = 2\|$$

$$|W| = 2|\log m| + 1$$
$R$, $C'$ and $C''$ will be devoted to the embedding of, respectively, routing elements, internal nodes and leaves of the trees in the collection level. $W$ will be used as working area.

$C''$ is divided into allocation units of $2 \lceil \log m \rceil$ positions (and elements) each. By the logical definition of the collection level, we know that for each leaf there is only a bit mask of $2 \lceil \log m \rceil$ bits. Therefore we allocate in $\mathcal{M}$ an array $I_{C''}$ of $2b'/2\lceil \log m \rceil$ entries with $2 \lceil \log m \rceil$ bits (two words) each.

Similarly, $C'$ is divided into allocation units of $2 \lceil \sqrt{\log m} \rceil$ positions (and elements) each. This time we allocate in $\mathcal{M}$ an array $I_{C'}$ of $2b'$ entries with three words each. It is a little more than we need but we can afford it (Assumption 6.1 and $b' < \lceil r/\log^2 m \rceil$).

Concerning $W$, we allocate in $\mathcal{M}$ an array $I_W$ of $|W|$ entries with three words each.

Finally, we do not allocate any auxiliary encoded data for zone $R$ right now, as it will be organized as a particular instance of abstract Problem 6.2.

As we will see, for both $C'$ and $C''$, the allocation units will be occupied from left to right and never released. Therefore, two simple counters are the whole auxiliary information we need in order to manage the allocation units. Finally, note that the sizes of $R$, $C'$ and $C''$ are beyond the need. Obviously, other choices are possible but they can only lower the constants in the computational bounds.

**Maintaining the invariants.** Let us suppose that an element $x$ has to be inserted in a tree $\mathcal{T}$ in the collection level. $x$ is routed toward a leaf $l$ as expected: the linked list of the current internal node is scanned to find the rightmost element less than or equal to $x$ and the process continues in the corresponding child. When $l$ is reached, the bit-mask is scanned and the leftmost position occupied by a placeholder is found (for that purpose a simple counter would do the job; as we will see, the bit-mask will be essential when the leaf is split and when the structure is visited). Finally, the corresponding bit is set to one and the placeholder and $x$ are exchanged.

If $l$ is full we have to split it in order to maintain invariant 6.2 on the size. We allocate the first free allocation unit in $C''$, (where the allocation process is a simple increment of an index, as we said) that will be the new leaf $l'$. Then we execute the following steps:

1. Find the rank $r(x)$ of $x$ in the sequence $lx$ (a simple scan of leaf $l$).

2. For $i = 2 \lceil \log m \rceil + 1$ to 1:
   
   (a) If $i = r(x)$ then exchange $x$ with $W[i]$ (a placeholder).

   (b) Otherwise, find the element $y$ with maximum rank (among the ones still in $l$) scanning $l$ and its bit mask, and set to zero the bit of $y$. Then, exchange $y$ with $W[i]$ (again, a placeholder).
3. Exchange the first $\lceil \log m \rceil$ elements in $l$ with $W[1 \ldots \lceil \log m \rceil]$, set the bit mask of $l$ to $1^{\lceil \log m \rceil}0^{\lfloor \log m \rfloor}$.

4. Exchange the first $\lceil \log m \rceil$ elements in $l'$ with $W[|W| - \lceil \log m \rceil + 1 \ldots |W|]$, set the bit mask of $l'$ to $1^{\lceil \log m \rceil}0^{\lfloor \log m \rfloor}$.

After that, we have to insert the element $x'$ of medium rank (that is still located in the $(\lceil \log m \rceil + 1)$-th position of $W$) in the parent of $l$, let it be $u$. If $u$ is not full, we simply follow the inner linked list of $u$ until the rightmost (in the list order that is also the sorted order) element $x''$ less than or equal to $x'$ is found. Then, we insert $x'$ after $x''$ in the list and set its child pointer to the starting address of $l'$.

Let us suppose $u$ is full, we have to split it in order to maintain invariant 6.3. We allocate a new unit $u'$ in $C'$, and execute the following steps assuming that every time an element is moved from a location of $C'$ to another in $W$, its auxiliary data (child pointer... ) stored in the encoded array $I_{C'}$ is moved to the corresponding position in $I_W$ too:

1. Follow the linked list of $u$ until the rightmost element $x''$ less than or equal to $x'$ is found. Let $r(x'')$ be the rank of $x''$ in the linked list. Exchange $x'$ with $W[r(x'') + 1]$.

2. Exchange the first $r(x'')$ elements in the inner list of $u$ with the first $r(x'')$ elements of $W$ (the first element in the list is exchanged with $W[1]$, the second one with $W[2]$ and so forth)

3. Exchange the last $2\lceil \sqrt{\log m} \rceil - r(x'')$ elements in the list of $u$ with the last $2\lceil \sqrt{\log m} \rceil - r(x'')$ of $W$.

4. Exchange $W[1 \ldots \lceil \sqrt{\log m} \rceil]$ with the first $\lceil \sqrt{\log m} \rceil$ elements of $u$ and initialize its list.

5. Exchange $W[|W| - \lceil \sqrt{\log m} \rceil + 1 \ldots |W|]$ with the first $\lceil \sqrt{\log m} \rceil$ elements of $u'$ and initialize its list.

After that, the element in position $\lceil \sqrt{\log m} \rceil + 1$ of $W$ is inserted in the parent of $u$ and the process is iterated. If even the root of the tree has to be split, its medium rank element is inserted in $R$.

Lemma 6.5 Under Assumption 6.1, the data structure can be built using $O(1)$ auxiliary space, $O(b \log m)$ comparisons and $O(b)$ moves.

Proof: $T$ has $O(1)$ levels and each internal node has $O(\sqrt{\log m})$ elements. Hence, the total number of comparisons we pay scanning the linked lists during the search for the position of $x$ is $O(\sqrt{\log m} \log \log m)$. Scanning the bit mask of $l$ costs $O(\log m)$ comparisons. Therefore, we pay $O(\log m)$ comparisons in order to find the position of $x$ in $T$. 
If \( l \) is not full, the insertion of \( x \) in it costs only \( O(1) \) moves, since we have to modify a bit of the bit mask and exchange \( x \) with the placeholder in \( l \).

If \( l \) is full, let us analyze the steps of the procedure for splitting a leaf. Step 1 is a simple scan and it takes \( O(\log m) \) comparisons. Step 2a is just a comparison of integer values. Step 2b takes a scan with \( O(\log m) \) comparisons and \( O(1) \) moves. Steps 2a and 2b are executed \( O(\log m) \) times and then the total cost of Step 2 is \( O(\log^2 m) \) comparisons and \( O(\log m) \) moves. Finally, Steps 3 and 4 are simple scans and exchanges and take \( O(\log m) \) comparisons and moves.

Then we have to insert \( x' \) in \( u \), the parent of \( l \). If \( u \) is not full we have to pay \( O(\sqrt{\log m} \log \log m) \) comparisons to follow its inner list and \( O(\log m) \) moves to update the pointers.

If \( u \) is full, let us analyze the steps of the procedure for splitting an internal node. We have to remember that every time an element of an internal node is moved its auxiliary encoded information in \( I_{C'} \) is moved too.

Step 1 scans the inner list of \( u \) and exchange one element of \( u \), that takes \( O(\sqrt{\log m} \log \log m + \log m) \) comparisons and \( O(\log m) \) moves. Steps 2 and 3 are just a series of exchanges of the remaining elements in \( u \) and then they take \( O(\sqrt{\log m} \log m) \) moves and comparisons (remember, also the auxiliary encoded data in \( I_{C'} \) is moved). Finally, Steps 4 and 5 do exchanges of \( O(\sqrt{\log m}) \) elements formerly contained in \( u \) and initialize two inner lists. Hence they cost \( O(\sqrt{\log m} \log m) \) moves and comparisons.

After the splitting of \( u \) the process is iterated at most for the \( O(1) \) ancestors of its. Given the above worst case costs and the fact that each leaf and each internal node has, respectively, \( O(\log m) \) and \( O(\sqrt{\log m}) \) elements, it is obvious to derive the amortized costs for inserting an element into \( T \). That is \( O(\log m) \) comparisons and \( O(1) \) moves in amortized sense.

If even the root of \( T \) has to be split, we insert a new routing element in \( R \). To organize \( R \) we use the solution to Problem 6.2 in Section 6.4.2.1. All the hypotheses in Problem 6.2 are satisfied. Obviously, \( R \) contains the elements produced by splitting a tree in the collection level and \( F \) the placeholders initially in \( R \). Hypothesis (i), (ii), (iii), (iv) are easily satisfied. For each routing element, there is a tree \( T \) with at least \( \log^3 m \) elements, therefore Hypothesis (v) is satisfied. We use the auxiliary encoded memory \( M \) to satisfy Hypothesis (vi).

Given the cost model in Assumption 6.1 and the solution to Problem 6.2 in Section 6.4.2.1, the thesis follows.

\[ \Box \]

### 6.5.1.2 Traversing the structure

After the construction of the structure, \( D_t \) contains placeholders. Traversing the structure is pretty standard. We maintain five pointers \( p_r, p_1, p_2, p_3, p_4 \) in auxiliary memory. \( p_r \) points to the rightmost visited routing element in \( R \) and \( p_1, p_2, p_3, p_4 \) point to the internal nodes in the current visiting path of the tree of the routing element pointed by \( p_r \). For each \( p_i \), we have to maintain a small pointer \( s_j \) to
the rightmost visited element in the internal linked list of the node pointed by \( p_j \). Actually, any pointed element (by \( p_r \) or by any \( s_j \)) is immediately exchanged with the leftmost placeholder in \( D_i \) and only its auxiliary encoded data is still accessible to guide the visit.

Each leaf \( l \) is visited in the following way:

1. Compute the number \( j \) of elements in \( l \) scanning its bit mask.

2. For \( i = 1 \) to \( j \):
   
   (a) Find the element \( x \) with minimum rank (among the ones still in \( l \)) scanning \( l \) and its bit mask,
   
   (b) Set its bit in the bit mask of \( l \) to zero.
   
   (c) Exchange \( x \) with the leftmost placeholder in \( D_i \).

By the cost model in Assumption 6.1, it is immediate to prove that the traversing phase ends with all the elements back in \( D_i \) in stable sorted order and that the whole traversing phase takes \( O(1) \) auxiliary locations, \( O(b \log m) \) comparisons and \( O(b) \) moves. Therefore, by Lemma 6.5 we can conclude that:

**Lemma 6.6** Under Assumption 6.1, \( b \) elements can be sorted stably, using \( O(1) \) auxiliary space and another set of \( b \) distinct elements as placeholders, with \( O(b \log m) \) comparisons and \( O(b) \) moves.

### 6.5.2 The fragmented multi-way merging

We know by hypothesis that the sequence \( J^m P^m CB \) satisfies Property 6.1 and \( b = |B| = |CB| / \log^3 m \). In Section 6.5.1, we showed how to sort \( b \) elements stably, using \( O(1) \) auxiliary space, with \( O(b \log m) \) comparisons and \( O(b) \) moves, under Assumption 6.1 and using \( B \) as an internal buffer.

Now, we show how to sort \( CB \) using the technique in Section 6.5.1 and a multi-way stable merging technique requiring a very limited amount of placeholders.

We want to solve the following problem.

**Problem 6.4** We have

- \( s \leq \frac{\log m}{\log \log m} \) sorted sequences \( E_1, \ldots, E_s \) of \( k \leq \frac{m}{s} \) elements each and

- a set \( \mathcal{U} \) of \( s(\lceil \log m \rceil)^2 \) distinct elements.

Under the Assumption 6.1, we want to sort the \( sk \) elements, stably, using \( O(1) \) auxiliary locations, with \( O(sk \log m) \) comparisons and \( O(sk) \) moves.
We name our solution to Problem 6.4 fragmented multi-way merging. Each sorted sequence is divided into \( \gamma = k/[\log m]^2 \) fragments of \( [\log m]^2 \) contiguous elements each (for simplicity, let us suppose \( [\log m]^2 \) divides \( k \)). Starting from the fragment with the largest element, we will denote the \( j \)-th fragment of the sequence \( E_i \) with \( F_i^j \).

The fragments of \( E_i \) are linked in a bidirectional list following the reverse sorted order of \( E_i \). The fragment with the largest element of a sequence is the head of the list. For each list we need \( 2k/[\log m]^2 \) words of \( [\log m] \) bits to store the pointers; for that, we use \( \mathcal{M} \) in the usual way.

One of the basic events in the process we are about to describe is the exchange of fragments (possibly belonging to two different sorted sequences). From now on we will assume that, when a fragment is moved, the pointers of its successor and its predecessor (if any) in its linked list are updated.

Let us denote the whole sequence of elements with \( P \) and with \( P_i \) the \( i \)-th fragment of \( P \) from the left end, for \( i = 1, \ldots, s \). The initial configuration is

\[
P = E_1E_2 \ldots E_{s-1}E_sU \quad \text{where} \quad E_i = F_i^iF_i^{i-1} \ldots F_i^2F_i^1
\]

and \( U \) contains the set \( U \) in some order. First, we exchange \( F_1^1 \) with \( F_1^i \), \( F_2^1 \) with \( F_2^{i-1} \) and so forth until the \( s \) heads are the first \( s \) fragments of \( P \) \((P_1 = F_1^1, P_2 = F_2^1 \ldots)\).

For \( i = 1, \ldots, s \) the fragment \( P_i \) is associated with a small integer \( p_i \) of \( O(\log \log m) \) bits containing the index (in \( P_i \)) of the first (from the right end) element of \( P_i \) not in \( U \). Two more indices num and last are maintained: num stores the current number of merged elements and last stores the address of the rightmost (in the whole \( P \)) fragment. All the integers \( p_i \) are stored in \( \mathcal{M} \) while num and last are in stored in two normal locations of memory. Initially all the small indices are set to \( [\log m]^2 \), num is set to 0 and last to \(|P| - |U| - [\log m]^2 + 1\).

Then the merging phase begins. The following steps are repeated until \( \text{num} = sk \):

1. The largest element among \( P_1[p_1], P_2[p_2], \ldots, P_s[p_s] \) is found (for the stability, in case of equal elements the one in the fragment of the sorted sequence with the largest index is chosen). Let it be \( P_i[p_i] \).

2. \( P_i[p_i] \) is exchanged with \( P[|P| - \text{num}] \), \( p_i \) is decreased by one and num is increased by one.

3. If \( P_i \) contains only elements of \( U \), that is \( p_i = 0 \), then let \( \nu \) be the address of the next fragment of \( E_i \). \( P_i \) is exchanged with the fragment starting at \( \nu \) and then the fragment starting at \( \nu \) is exchanged with the one starting at last. Finally, we set \( \text{last} = \text{last} - ([\log m])^2 \).

After the execution \( P = US \), where in \( U \) we have the elements of \( U \) in some order and in \( S \) we have the stably sorted sequence of the \( sk \) elements.
Let us prove that the wanted bounds hold. For any $1 \leq i \leq s$, Step 1 requires $O(\log \log n)$ comparisons to decode the small integer $p_i$ and one comparison between $P_i[p_i]$ and the current maximum. Since $s \leq \frac{\log m}{\log \log m}$, the total cost of each execution of Step 1 is $O(\log m)$.

In Step 2 we pay one exchange, one arithmetical increment and the cost of decreasing by one the selected $p_i$. This would be $O(\log \log m)$ moves in the worst case but only $O(1)$ moves in amortized sense with an analysis similar to the one for a binary counter (see [Cormen, Leiserson, Rivest, and Stein, 2001]).

Step 3 requires the access of a constant number of encoded pointers of $O(\log m)$ bits each, the exchange of a constant number of blocks of $[\log m]^2$ elements each (fragments) then the worst case bounds are $O(\log m)$ for the comparisons and $O(\log^2 m)$ for the moves. However, Step 3 is executed only when one of the head fragments $P_1, \ldots, P_s$, say $P_i$, is exhausted (i.e. it is full of buffer elements). Hence, the cost of the block exchanges of Step 3 is charged over the $[\log m]^2$ previous extractions from $P_i$ that did not lead to the execution of Step 3 and therefore $O(1)$ moves in amortized sense are performed in this step.

Since the total number of iteration is $O(\log^3 m)$ we have the wanted bounds and Problem 6.4 is solved.

**Sorting CB, finally.** With the fragmented multi-way merging and the technique of Section 6.5.1, we can finally sort the sequence $CB$ when $b = |B| = ||CB||/\log^3 m$.

1. $C$ is logically divided into $t = \lceil|C|/b\rceil$ subsequences $C_1 C_2 \ldots C_{t-1} C_t$ of $b$ elements each. Every $C_i$ is sorted using the technique in Section 6.5.1 with $B$ as internal buffer.

2. Since $b = ||CB||/\log^3 m$, we have that $t = O(\log^3 m)$. Therefore, the sorted runs $C'_1 C'_2 \ldots C'_{t-1} C'_t$ can be merged executing a constant number of iterations of the multi-way mergesort using the fragmented multi-way merging with $s = \frac{\log m}{\log \log m}$. (we have plenty of distinct buffer elements to use with the fragmented multi-way merging since $|B| = ||CB||/\log^3 m|$).

3. $B$ is sorted with the stable in-place mergesort (using [Salowe and Steiger, 1987]) and $C$ and $B$ are merged in-place stably (using [Salowe and Steiger, 1987], again).

Therefore, by Lemma 6.6 and by the solution to Problem 6.4, we can conclude that:

**Theorem 6.2** Under Assumption 6.1, the subsequence $CB$ of a sequence $J'''P'''CB$ satisfying Property 6.1 and $|B| = ||CB||/\log^3 m$, can be sorted stably, using $O(1)$ auxiliary locations, performing $O(m \log m)$ comparisons and $O(m)$ moves in the worst case.
6.6 Sorting with Few Distinct Elements

In this section we show how to sort the subsequence $CB$ of a sequence $J^m P^w CB$ satisfying Property 6.1 and with $b = |B| < \lfloor |CB|/\log^3 m \rfloor$, that is, when the number $d$ of distinct elements in $CB$ is less than $\lceil |CB|/\log^3 m \rceil$. First, we solve a general problem in Section 6.6.1. Then (Section 6.6.2), we show how the solution to the general problem can be used to sort $CB$.

6.6.1 Sorting with two kinds of internal buffer

We are interested in solving the following problem.

**Problem 6.5** We are given:

- A set $\mathcal{D}$ of $d' < \lfloor r/\log^2 m \rfloor$ elements.
- Two sequences $V$ and $G$ of $t \leq m$ elements each and $V$ has $d'' \leq d'$ distinct elements.
- An $O(1)$ time boolean function $\text{BEGINS\_TO\_V}(x)$ that, at any time, returns true if and only if $x$ belongs to the set of elements originally contained in $V$.

We want to go from sequence $VGD$ to sequence $V'GD'$ where $V'$ contains the elements in $V$ sorted stably and $D, D'$ contain the elements in $\mathcal{D}$ in any order. Under Assumption 6.1, we have to use $O(1)$ auxiliary locations and perform $O(t \log m)$ comparisons and $O(t)$ moves.

The abstract problem can be seen as the problem of sorting a sequence $V$ of elements with a limited number of distinct elements having by our side a constant time function helping to discern between the elements of $V$ and the other ones and two kinds of internal buffers:

- The first buffer is small, the order of its elements is not important and can be lost after the process. Moreover the number of elements in this buffer is greater than or equal to the number of distinct elements in $V$. That sequence would be $D$ with the elements of set $\mathcal{D}$. When we will use our solution for this abstract problem to sort the subsequence $CB$, the role of $D$ will be obviously played by the subsequence of distinct element $B$.

- The second buffer is as large as $V$ is but the original order of its elements is important and has to be maintained after $V$ is sorted. That large buffer would be $G$.

Our solution to Problem 6.5 has three phases.
6.6. SORTING WITH FEW DISTINCT ELEMENTS

6.6.1.1 First phase

$V$ is logically divided into $\frac{|V|}{d' \log m}$ contiguous blocks $V_1 V_2 \ldots$ of $d' [\log m]^2$ elements each. We want to sort any block $V_i$, stably, using $O(1)$ auxiliary locations, $O(|V_i| \log m)$ comparisons and $O(|V_i|)$ moves.

This can be accomplished the same way we sorted the sequence $C$ in Section 6.5.

1. Each sub-block of $d'$ contiguous elements of $V_i$ is sorted using the $d'$ elements of $D$ as placeholders (Section 6.5.1).

2. The $[\log m]^2$ sorted sub-blocks of $V_i$ are merged with a constant number of iterations of the multi-way mergesort using the fragmented multi-way merging (Section 6.5.2). We have to distinguish two cases, though.

   (a) If $|D| = d' \geq \frac{\log m}{\log \log m}^a$, we can apply the solution to Problem 6.4 we presented in Section 6.5.2.

   (b) If, on the other hand, $|D| = d' < \frac{\log m}{\log \log m}^a$ we may not have a sufficient number of distinct elements for the set $U$ in Problem 6.4. However, if $d' < \frac{\log m}{\log \log m}$ then

   \[ |V_i| = d' [\log m]^2 = polylog(m). \]

Hence, we can use the same fragmented multi-way merging technique in Section 6.5.2 but with fragments of size $O(\log \log m)$ instead of $O(\log^2 m)$. That reduces the size of the set $U$ from $O(s \log^2 m)$ to $O(s \log \log m)$. If we choose $s = \frac{\log m}{(\log \log m)^2}$, the number of iterations of the $s$-way mergesort needed to sort the whole block $V_i$ is still a constant but the size of $U$ is $O\left(\frac{\log m}{\log \log m}\right)$.

Therefore, the elements in $U$ do not have to be distinct anymore because we can maintain in a single word of (real) auxiliary memory the whole permutation to bring them back to their original order when the fragmented $s$-way merging process is done.

6.6.1.2 Second phase

After the first phase, each block $V_i$ of $V$ is sorted and divided into at most $d'' \leq d'$ runs of equal elements. Since $|V_i| = d' [\log m]^2$, the total number $t_r$ of runs in $V$ is less than or equal to $t/\log m$. For any run, let the first element be the head and the rest of the run be the tail.

1. Each block $V_i$ is divided into two sub-blocks $H_i$ and $V'_i$. $H_i$ contains the heads of all the runs of $V_i$ and $V'_i$ contains all the tails. Both $H_i$ and $V'_i$ are in sorted order. This subdivision can be accomplished in a linear number of moves with
at most $d'$ applications of the well-known in-place block exchanging technique (we recalled it in Section 6.4.1).

Let $i_r$ be the number of runs of $V_i$. Let $h_1, \ldots, h_{i_r}$ be the heads we have to collect, indexed from the leftmost to the rightmost in $V_i$.

Let $U_1, \ldots, U_{i_r}$ be the subsequences of $V_i$ that separate $h_1, \ldots, h_{i_r}$, that is

$$V_i = h_1 U_1 h_2 U_2 \ldots U_{i_r-1} h_{i_r} U_{i_r}.$$  

(some of them can be void). We collect $h_1, \ldots, h_{i_r}$ in a growing subsequence $H_i$ starting from the position of $h_1$. During the process, $H_i$ slides toward the right end of $V_i$. The process scans $V_i$ from left to right and therefore the positions of $U_1, \ldots, U_{i_r}$ and $h_1, \ldots, h_{i_r}$ are obtained "on the fly", during the scan.

Let $H_i = h_1$ and $j = 1$. The following steps are repeated until $j > i_r$:

(a) If $|H_i| \leq |U_j|$, do a block exchange between the two adjacent blocks $H_i$ and $U_j$.

(b) Otherwise, let $H_i = H'_i H''_i$ with $|H'_i| = |U_j|$. Exchange $H'_i$ with $U_j$ (obvious exchange of two non adjacent but equal sized blocks). After that $H_i := H''_i H'_i$.

(c) In both cases, now $H_i$ is adjacent to $h_{j+1}$; let $H_i = H_i h_{j+1}$ and increase $j$ by one.

Since the elements we are extracting (the heads of the runs of a single block $V_i$) are distinct, we do not care about their original order during the process. We simply sort them when they are finally collected at the right end of $V_i$. On the other hand, the order of the other elements of the runs of $V_i$ is maintained in the process.

2. Some information about runs and blocks is collected and stored in $M$.

- An array $I_H$ with $\frac{|V_i|}{d |\log m|^2}$ entries of two words each is stored in $M$. For any $i$, the first word of $I_H[i]$ contains $|H_i|$ and the second word contains the index of the first run of $V_i$ (the index is between 1 and $t_r$, from the leftmost run in $V$ to the rightmost).

- An array $I_R$ with $t_r$ entries of four words each is stored in $M$. For any $i$, the first word of $I_R[i]$ is initially set to $i$, the second one contains the address of the head of the $i$-th (in $V$) run, the third one contains the starting address of the tail of the $i$-th run and the fourth one contains the size of the $i$-th run.

- Finally, an array $I_{R^{-1}}$ with $t_r$ entries of two words each is stored in $M$. For any $i$, the first word of $I_{R^{-1}}[i]$ is initially set to $i$ and the second word of $I_{R^{-1}}[i]$ is initially set to 1.
All this information can be obtained within our target bounds simply by scanning $V$. In general, for any array $I$ of multi-word entries, we will denote the $p$-th word of the $i$-th entry with $I[i][p]$.

3. $I_{R-1}$ is sorted stably by head, that is, at any time of the sorting process, the sorting key for the two-word value in the $i$-th entry of $I_{R-1}$ is

$$V[I_R[I_{R-1}[i][1]][1]].$$

The sorting algorithm used is Mergesort with a linear time in-place stable merging (e.g. that described in [Salowe and Steiger, 1987]). During the execution of the algorithm, every time the two-word value in the $i$-th entry of $I_{R-1}$ is moved to the $j$-th entry, the corresponding entry in $I_R$ is updated, that is $I_R[I_{R-1}[j][1]][1]$ is set to $j$.

We remark that only the entries of the encoded array $I_{R-1}$ are moved (where any abstract move of an encoded value causes $O(\log m)$ actual moves of some elements contained in zones $Q'$ and $Q''$ of Section 6.3). In this process, none of the elements in $V$ are moved.

4. For $i = 2$ to $t_r$, let $I_{R-1}[i][2]$ be $I_{R-1}[i-1][2] + I_R[I_{R-1}[i][1]][4]$ (that is, if we had the elements in $V$ sorted stably into another sequence $V'$, $I_{R-1}[i][2]$ would be the starting address in $V'$ of the $i$-th run in the stable sorted order).

### 6.6.1.3 Third phase

After the second phase we are able to evaluate the function $\alpha_V : \{1, \ldots, t\} \rightarrow \{1, \ldots, t\}$ such that $\alpha_V(j)$ is the rank of the element $V[j]$ in the sequence $V$, performing $O(\log m)$ comparisons.

1. Let $V_i$ be the block of $V[j]$. We know where $H_i$ starts and ends, in fact

$$H_i = V[s_i \ldots s_i + I_H[i][1] - 1] \quad \text{where} \quad s_i = (i - 1)d'(\lceil \log m \rceil)^2 + 1.\]$$

Therefore, we can perform a binary search for $V[j]$ in $H_i$ and find the index $p_j$ in $V_i$ of the run to which $V[j]$ belongs.


3. Using the array $I_R$, we can find the position $k_j$ of $V[j]$ in its run. If $j = I_R[p_j'][2]$ then $V[j]$ is the head of its run. Otherwise, $V[j]$ belongs to the tail of its run. Let us define $k_j$ in the following way:

$$k_j = \begin{cases} 1 & \text{if } j = I_R[p_j'][2] \\ j - I_R[p_j'][3] + 2 & \text{Otherwise} \end{cases}$$
4. Finally, we have that $\alpha_V(j) = I_{R^{-1}}[I_R[p'_j][1][2] + k_j - 1$.

Using this algorithm and the given function \textsc{Belongs\_to\_V}(x) to discern between the elements originally contained in $V$ and the ones originally in $G$, it is possible to sort the elements in $V$ efficiently, using $G$ as internal buffer while preserving the original order of its elements.

The idea. Before the formal description of this last phase is given, a short outline is needed. The algorithm has two nested iterations.

- The outer iteration scans the elements of $V$ following the sorted order (we know the order of the runs from the previous phase, therefore the elements can be scanned in sorted order easily).

  During the scan, three kinds of elements can be found: heads of runs, elements belonging to the tails of their runs, and buffer elements from $G$ (as we will see, the inner iteration is responsible for the presence in these elements).

  - If a buffer element is found (recognized using the given function \textsc{Belongs\_to\_V}(x)), there is nothing to do: the element of $V$ previously stored in this position has already reached its final destination.

  - If a head is found, nothing can be done since the heads are the cornerstones of the algorithm used to find the rank of an element in $V$. As we will see, their treatment is delayed until the very end of the algorithm.

  - Finally, if an element $x$ of a tail is found, the inner iteration starts.

- The purpose of the inner iteration is to scan the cycle (of the permutation that disposes the elements of $V$ in sorted order) to which the element belongs. During the scan of the cycle of $x$ two kinds of elements can be found: heads of runs and tail elements (the first found is obviously $x$). Again, the heads are left in their position. On the other hand, any tail element $y$ is ranked (with $\alpha_V$), let its rank in $V$ be $r_y$, and is exchanged with the element in $G$ corresponding to its rank, that is $b_y = G[r_y]$. Then, there can be two cases:

  - If $V[r_y]$ is a head, it cannot be moved and then $b_y$ is left in the position in $V$ previously occupied by $y$ and treated in a special way.

  - If $V[r_y]$ is a tail element, we immediately exchange $V[r_y]$ with $b_y$, recovering the correct position for $b_y$. Therefore, the next element of the cycle is in the position previously occupied by $y$.

After the two nested iterations, a final simple iteration performs $t_r$ exchanges that bring the heads in their final positions.
The algorithm. Now we can give a precise description of the algorithm for the third phase of our solution to Problem 6.5.

The function \( \text{is\_head}(x) \) used in the algorithm returns true if \( V[x] \) is the head of its run. It can be calculated in the very same way the rank of an element in \( V \) is (with the exclusion of the fourth step).

1: FOR \( i = 1 \) to \( t_r \) and \( j = 2 \) to \( I_R[I_{R-1}[i][1]][4] \) DO  
2: \( \text{start} \leftarrow I_R[I_{R-1}[i][1]][3] + j - 1 \)  
3: IF \( \text{Belongs\_to\_V}(V[\text{start}]) \) THEN  
4: \( \text{next} \leftarrow \alpha_V(\text{start}) \)  
5: WHILE \( \text{next} \neq \text{start} \) DO  
6: Exchange \( V[\text{start}] \) and \( G[\text{next}] \)  
7: IF \( \text{is\_head}(\text{next}) \) THEN  
8: \( \text{next} \leftarrow \alpha_V(\text{next}) \)  
9: ELSE  
10: \( \text{next\_tmp} \leftarrow \text{next} \)  
11: \( \text{next} \leftarrow \alpha_V(\text{next}) \)  
12: Exchange \( V[\text{start}] \) and \( V[\text{next\_tmp}] \)  
13: FOR \( i = 1 \) to \( t_r \) DO  
14: \( \text{head} \leftarrow I_R[I_{R-1}[i][1]][2] \)  
15: Exchange \( V[\text{head}] \) and \( G[I_{R-1}[i][2]] \)  
16: Exchange \( G \) and \( V \)

6.6.2 Sorting \( CB \)

With the solution to the abstract Problem 6.5, we can finally sort the sequence \( CB \) when \( b = |B| < |CB|/\log^3 m |.\)

1. \( C \) is partitioned into three subsequences \( C'UC'' \), where \( U \) contains all the elements equal to the element \( c_m \) of rank \( |C|/2 \) in \( C \), and \( C', C'' \) contain all the elements of \( C \), respectively, less than and greater than \( c_m \). This partition can be easily obtained using the stable in-place selection and the stable in-place partitioning in [Katajainen and Pasanen, 1994, 1992].

2. To sort \( C' \) and \( C'' \) we can apply the solution to Problem 6.5.

   (a) We set \( V = C' \), \( G = (UC'')[1 \ldots |C'|] \), \( D = B \), \( \text{Belongs\_to\_V}(x) = (x < c_m) \) and sort \( C' \).

   (b) We set \( V = C'' \), \( G = (C'U)[1 \ldots |C''|] \), \( D = B \), \( \text{Belongs\_to\_V}(x) = (c_m < x) \) and sort \( C'' \).

(Obviously there can be extreme situations in which \( C' \) or \( C'' \) are void)
3. $B$ is finally sorted with the normal Mergesort using a linear time in-place stable merging (e.g. that described in [Salowe and Steiger, 1987]) and the two sequences are merged (once again with the algorithm given in [Salowe and Steiger, 1987]).

Therefore, we can conclude that:

**Theorem 6.3** Under Assumption 6.1, the subsequence $CB$ of a sequence $J''P''CB$ satisfying Property 6.1 and $|B| < \lceil |CB| / \log^3 m \rceil$, can be sorted, stably, using $O(1)$ locations of auxiliary memory, performing $O(m \log m)$ comparisons and $O(m)$ moves in the worst case.

### 6.7 Conclusion

By Theorems 6.1, 6.2 and 6.3 we can conclude that Problem 6.1 is solved and state the main result of this chapter.

**Theorem 6.4** Any sequence of $n$ elements can be sorted stably, using $O(1)$ auxiliary locations of memory, performing $O(n \log n)$ comparisons and $O(n)$ moves in the worst case.

This settles a long-standing open question explicitly stated by Munro and Raman in [Munro and Raman, 1992]. Before the introduction of this algorithm, the best known solution for stable in-place sorting with $O(n)$ moves was the one presented in [Munro and Raman, 1996b], performing $O(n^{1+\epsilon})$ comparisons in the worst case.
Chapter 7

The Value of Proximity

Abstract

In the Cache-Oblivious model, the underlying memory hierarchy has the following characteristics:

- Two levels, cache level with $M$ locations and the main memory level, unbounded in size. Both levels are divided into blocks of $B$ contiguous locations each.
- The content of any block of main memory can be transferred in any block of the cache (the cache is fully associative).
- An off-line optimal block-replacement algorithm is assumed to manage the block transfers between the two levels.

An algorithm is cache-oblivious if it is not conscious of any parameter of the memory hierarchy ($M$, the size of cache memory, and $B$, the size of the minimal contiguous block of information that can be transferred between the cache and the main memory). Hence, it cannot directly exploit these parameters to reach the optimality.

In the Cache-Oblivious model the complexity is measured with two criteria: the work complexity, which is the standard complexity in the RAM model, and the cache complexity, which is the total number of block transfers (cache misses) incurred during the computation.

In this chapter we shall give a twofold contribution. We will present the first sorting algorithm that is work optimal, cache optimal and space optimal in the Cache-Oblivious model. Furthermore, we introduce a new approach to the sorting problem previously unknown both in the Cache-Oblivious model and RAM model.

The presentation in this chapter is based on the paper [Franceschini, 2004] (SODA 2004).
7.1 Introduction

In the “real world” computing systems the use of multi-level memory hierarchies is a common practice. Typically, the highest level of the hierarchy contains a great amount of low-cost memory with slow access time. On the other end of the hierarchy, there is the lowest level with a small quantity of more expensive type of memory with fast access time.

Usually, the processing unit can address the whole highest level, but it can access only the locations in the lowest one. Therefore, during a generic computation, the data flows continuously between the levels of the hierarchy, in the attempt of the system to satisfy the requests of the running program(s). In order to amortize the cost of the inter-level flow of information, the data transfers between two adjacent levels always involve blocks of contiguous locations, therefore data locality is an important characteristic of algorithms operating in those computing environments.

Complex memory hierarchies contributed to increase the distance between the practical and theoretical computing. An algorithm that has an optimal complexity in the classic RAM model can be potentially outperformed in a real computation by sub-optimal algorithms which exploit a better data locality.

7.1.1 Memory hierarchies and the Cache-Oblivious model

Various attempts were made in the past in order to find computational models capturing the characteristics of “real world” memory hierarchies.

Other memory hierarchy models. Aggarwal et al. [Aggarwal, Alpern, Chandra, and Snir, 1987a] introduced the Hierarchical Memory model. Like in the RAM model, this model has an unlimited number of locations. It aims at modeling a multi-level hierarchy with the use of a function \( f(i) \) that gives the access cost for the \( i \)th location. The authors mainly study the case in which \( f(i) = \lceil \log i \rceil \).

The widely studied External-Memory model, popularized in [Aggarwal and Vitter, 1988], has a simple two-level memory hierarchy: a limited and fast internal memory of size \( M \) and an unlimited slow external memory. The processing unit can address the external memory but can process only the data that reside in internal memory. An algorithm operating in this model must directly manage the transfers of data between the two levels. Those transfers involve blocks of \( B \) elements. The complexity of an algorithm in this model is the number of block transfers it incurs. It aims to model the very common hierarchy formed by the internal memory and the hard disk.

The Cache-Oblivious model. A recent, though widely studied, model of computation is the Cache-Oblivious model, introduced by Frigo et al. [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Prokop, 1999]. For the sake of “argument local-
ity”, we give a very short review of the Cache-Oblivious model. A fully detailed description is in Chapter 2.

In the Cache-Oblivious model, the underlying memory hierarchy has the following characteristics:

- There are two levels: cache level composed by \( M \) locations and the main memory level of unbounded size. Both levels are divided into blocks of \( B \) contiguous locations each.

- The content of any block of main memory can be transferred in any block of the cache (the cache is fully associative).

- An off-line optimal block-replacement algorithm manages the block transfers between the two levels.

An algorithm is cache-oblivious if it is not conscious of any parameter of the memory hierarchy (\( M \), the size of cache memory, and \( B \), the size of the minimal contiguous block of information that can be transferred between the cache and the main memory). Hence, it cannot directly exploit these parameters to reach the optimality.

In the Cache-Oblivious model the complexity is measured with two criteria: the work complexity, which is the standard complexity in the RAM model, and the cache complexity, which is the total number of block transfers (cache misses) incurred during the computation. Since the descriptions of algorithms for the Cache-Oblivious model look exactly like the ones for the RAM model, we may see the Cache-Oblivious model as a “successor” of the RAM model incorporating a lot of the new architectural aspects which characterize the real world computing systems.

There could be some concerns about some assumptions made in the model being too unrealistic. However, Frigo et al. [Frigo, Leiserson, Prokop, and Ramachandran, 1999] have shown that algorithms satisfying a reasonable regularity condition on the number of cache misses in the ideal-cache model, have a small constant-factor overhead for that measure when used in a weaker cache model with the realistic LRU replacement strategy. For the full associativity assumption Frigo et al. provide an expected-time, favorable argumentation. Furthermore, since the cache complexity analysis holds for any value of \( B \) and \( M \), it holds for any level of a more general multi-level memory hierarchy.

### 7.1.2 The result in this chapter

The algorithm we will present in this chapter has a twofold contribution:

- We present the first in-place optimal sorting algorithm for the Cache-Oblivious model. A work optimal, dictionary based in-place sorting algorithm can be derived from the optimal implicit dictionaries in [Franceschini and Grossi,
2003b,c]. That straightforward algorithm can incur $O(N \log_B N)$ cache misses. However, a simple variation of the well-known in-place binary mergesort based on in-place merging can lower the cache complexity bound to $O\left(\frac{N}{B} \log N\right)$ block transfers.

- Furthermore, with our new algorithm we introduce a new optimal approach to the sorting problem for this model of computation. The only two other optimal approaches known so far were the Funnelsort and the Distribution sort (obviously this is not the well known two-way distribution sort for the RAM model) both presented in the seminal paper [Frisco, Leiserson, Prokop, and Ramachandran, 1999]. We will review those algorithms in a subsequent section.

The presentation in this chapter is based on the paper [Franceschini, 2004] (SODA 2004).

### 7.1.3 Organization of the chapter

In Section 7.2 we review the previous cache optimal sorting algorithms operating in the Cache-Oblivious model: the Funnelsort and the Distribution sort. In Section 7.3 we present a first version of Proximity Mergesort that is work and cache optimal but that uses $O(N)$ auxiliary space, in order to highlight the extreme differences between the new approach and the two previous ones. At the end of the same section we give the work and cache complexity analysis for the space sub-optimal version of the Proximity Mergesort. In Section 7.4 we show how to reduce the auxiliary space to $O(1)$ locations, reaching also the space optimality. We will point out the variations in the complexity analysis.

### 7.2 The Sorting Problem & the Cache-Oblivious Model

In the seminal paper [Frisco, Leiserson, Prokop, and Ramachandran, 1999] it was proven that the cache complexity of any sorting algorithm in the Cache-Oblivious model is $\Omega\left(\frac{N}{B} \log_M N\right)$ and the work complexity is $\Omega(N \log N)$. The latter lower bound comes directly from the comparison model. For what concerns the lower bound for the cache complexity, it can be simply derived from

- the $\Omega\left(\frac{N}{B} \log_M \frac{N}{B}\right)$ lower bound for the block transfers for the sorting problem in the External-Memory model [Aggarwal and Vitter, 1988] and

- the so-called tall cache assumption, that is, the imposed constraint that $M = \Omega(B^2)$. 
As it is obvious, the $\frac{N}{B} \log_M N$ cache misses could not be called a lower bound since it depends on the tall cache assumption. One could wonder if that would actually be the lower bound when no relations between the parameters $M$ and $B$ were imposed, beside the obvious one requiring that $M > B$. In [Brodal and Fagerberg, 2003] there is some evidence that without the tall cache assumption, or its weaker form $M = \Omega(B^{1+\epsilon})$, there is no cache-oblivious sorting algorithm with a cache complexity $O\left(\frac{N}{B} \log_M N\right)$. From now on, we will assume that the cache is tall, that is $M = \Omega(B^2)$, and we will have $\Omega\left(\frac{N}{B} \log_M N\right)$ as the target for the cache complexity of our new sorting algorithm.

So far, the lower bounds for work and cache complexity were matched by the two sorting algorithms presented in the seminal works [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Prokop, 1999]: Funnelsort and Distribution sort. A variant of the Funnelsort, the Lazy Funnelsort was given in [Brodal and Fagerberg, 2002a]. A sorting algorithm can also be derived from the priority queue in [Brodal and Fagerberg, 2002b], while the one in [Arge, Bender, Demaine, Holland-Minkley, and Munro, 2002] depends on a sorting algorithm itself. Moreover, a classical property like in-placeness seems to be challenging in the Cache-Oblivious model because of the apparently antithetical requirements needed to obtain the maximum space saving and the data locality. For these reasons, the techniques originally developed for this problem in the RAM and External-Memory models appear to be of difficult application in our scenario.

### 7.2.1 Funnelsort

The Funnelsort [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Brodal and Fagerberg, 2002a] can be seen as a version of the classic two-way mergesort where each single two-way merging step could be interrupted a number of time before the end in order to create locality of memory accesses.

1. The input sequence is divided into about $N^{1/3}$ contiguous sub-sequences of about $N^{2/3}$ elements each.

2. Each sub-sequence is sorted recursively and the final sorted sequence is produced using an $N^{1/3}$-merger.

Essentially, a $t$-merger is a complete binary tree with $t$ leaves (one for each sorted input sequence), with a buffer for each edge and laid out in memory following the van Emde Boas layout and by which also the size of the buffers is determined. When invoked, a $t$-merger outputs the first $t^3$ elements of the sorted sequence obtained by merging the $t$ sorted input sequences.

The van Emde Boas layout is also a basic tool adopted in all the optimal dynamic dictionaries so far developed for the Cache-Oblivious model, see [Prokop, 1999, Bender, Demaine, and Farach-Colton, 2000, Bender, Duan, Iacono, and Wu, 2002d,
Brodal, Fagerberg, and Jacob, 2002, Bender, Cole, and Raman, 2002b, Franceschini and Grossi, 2003b,c] (for the last two papers look also in Chapters 13 and 15). In [Vinther, 2003], the author proved that, for the case of sorting, the layout is not important but the buffer sizes. Vinther also gave a variant of Funnelsort that uses only $o(n)$ extra space.

To explain the van Emde Boas layout let us consider a complete binary tree $T$ of height $h = O(\log t)$ ($t$ is the number of elements in the tree). The layout is defined recursively as follow:

- Divide $T$ at level $\lfloor h/2 \rfloor$. We obtain a top recursive subtree $T_0$ and $p \leq 2^{\lfloor h/2 \rfloor}$ bottom recursive subtrees $T_1, T_2, \ldots, T_p$ having height at most $\lfloor h/2 \rfloor$.

- The van Emde Boas layout for $T$ is $vEB(T_0)vEB(T_1)\ldots vEB(T_p)$, where each $vEB(T_i)$ is the van Emde Boas layout of $T_i$ and is stored in a contiguous zone of memory.

In Figure 7.2.1 there is a pictorial example of the van Emde Boas layout. As we said, a $t$-merger can be seen as a complete binary tree $T$ with $t$ leaves where each edge is a buffer. The sizes of the buffers are decided in a recursive fashion as for the van Emde Boas layout.

- Divide $T$ at level $\lfloor h/2 \rfloor$. We obtain a top recursive subtree $T_0$ and $p \leq 2^{\lfloor h/2 \rfloor}$ bottom recursive subtrees $T_1, T_2, \ldots, T_p$ as in the van Emde Boas layout. The buffers $b_1, b_2, \ldots, b_p$ on the edges connecting the leaves of $T_0$ with the roots of $T_1, \ldots, T_p$ have size $O\left(t^2\right)$.

- The $t$-merger laid out as $vEB(T_0)b_1 \ldots b_pvEB(T_1)\ldots vEB(T_p)$ in a contiguous zone of memory (where each $vEB(T_i)$ is the van Emde Boas layout of $T_i$).

A $t$-merger is applied with a call of the following procedure on the root of the tree.

**FILL**(v)
1: while v’s output buffer is not full do
2: if left input buffer is empty then
3: FILL(left(v))
4: if right input buffer is empty then
5: FILL(right(v))
6: perform one merge step

A node is marked as “done” when its input buffers are empty and, recursively, all the nodes of the sub-trees rooted at its children are marked.

### 7.2.2 Distribution sort

The Distribution sort performs the following steps.
1. It divides the input sequence into about $N^{1/2}$ contiguous sub-sequences of the same size and recursively sorts these sub-sequences.

2. It distributes the elements in $q \leq \sqrt{N}$ buckets $B_1, \ldots, B_q$ such that $|B_i| \leq 2\sqrt{N}$ and $\max\{x : x \in B_i\} \leq \min\{x : x \in B_{i+1}\}$, for $i = 1, 2, \ldots, q - 1$.

3. It recursively sorts each bucket.

4. It copies the elements back to the input sequence.

The way the second step is performed is crucial to obtain the optimal cache complexity. Unlike other kinds of sorting based on distribution, the phase in which the pivots (or sample set) are chosen and the distribution phase are not separated. In [Frigo, Leiserson, Prokop, and Ramachandran, 1999] a bucket splitting technique is used: the buckets have a variable size upper bounded by $2\sqrt{N}$; when a bucket reaches that threshold then its median element is selected (that will be a new pivot) and the bucket is partitioned in two new buckets of size $\sqrt{N}$. The bucket splitting technique alone it is not sufficient to achieve a good cache complexity.

The order in which the sub-sequences are processed is critical. For example, if we process the sub-sequences from the leftmost to the rightmost one, distributing the elements of each sub-sequence along all the existing buckets in one single run, we incur in $O(N)$ cache misses instead of $O(N/B)$, as required by the cache optimality. As we will see, in our result we do not need to make use of the bucket splitting
technique but we use that particular processing order introduced in [Frigo, Leiserson, Prokop, and Ramachandran, 1999] for disposing some particular groups of elements.

We do not need to split the buckets because we are able to find a perfectly balanced sample set to distribute with. Dealing only with sub-problems of the exact same size is one of the characteristic that differentiates the Proximity Mergesort from the Distribution sort and the Funnelsort. That characteristic is of great help when the space optimality is an objective since, for any recursive application of the algorithm, we have to remember only few values.

7.3 Proximity Mergesort

In this section we present a first version of Proximity Mergesort that is work and cache optimal but that uses $O(N)$ auxiliary space, in order to highlight the extreme differences between it and the two previous approaches to work and cache optimal sorting in the Cache-Oblivious model. At the end of the section we give the work and cache complexity analysis for the space sub-optimal version of the Proximity Mergesort.

7.3.1 The intuition

Exponential divide and conquer approaches. Let us give the main ideas behind our algorithm. There is a common scheme that seems to be very effective for the optimality in the Cache-Oblivious model: a divide and conquer approach that refines the size of the subproblems in an "exponential" way. For example, the van Emde Boas layout is applied recursively to $\Theta(\sqrt{N})$ subtrees of size $\Theta(\sqrt{N})$ each, where $N$ is the number of elements in the whole tree.

In the Distribution sort the recursive calls are performed on sub-sequences of size $\Theta(\sqrt{N})$ each, and the distribution step is performed using a set of $\Theta(\sqrt{N})$ pivots. Again, in the Funnelsort the recursive calls generate $\Theta(N^{1/3})$ sorted sequences that need to be merged. In the Funnelsort these $\Theta(N^{1/3})$ sorted sequences are independent: given any element of a certain sequence, we cannot infer its rank inside another sequence, let alone the whole input set. For this reason, the final merge step in Funnelsort needs the aid of a very powerful and quite complicated structure like the $k$-merger.

Driving the elements in the proximity of their final position. Our approach is based on a different idea, namely, that the sorted sequences involved in the final merge step are not independent. The permutation of elements on which is performed the final merge step has a proximity property that guarantees that any element in this permutation is not too far from its final destination. With this property, we need to perform a final merge step that is greatly simplified with respect to that in Funnelsort.
7.3. PROXIMITY MERGESORT

We now describe the observation at the heart of the proximity property. A particular case of the following lemma was used by Kronrod [Kronrod, 1969, Salowe and Steiger, 1987] in the first known in-place linear time merging algorithm.

**Lemma 7.1** Let us consider a sequence of $N$ elements divided into $N/X$ sorted sub-sequences of $X$ elements each. Each sub-sequence is further divided into $X/Y$ segments of $Y$ elements each. Let us focus on the new sequence obtained by permuting the segments of $Y$ elements each, so that the segments are stably sorted according to their smallest element. We have that no element is more than $((N/X) - 1)(Y - 1)$ positions beyond its final position in the fully sorted sequence.

**Proof:** Let us consider the initial arrangement of the elements. The sequence is logically divided into sorted sub-sequences indexed from 1 to $N/X$.

We assign a unique *type* to each sequence based on its index, namely, the $t$-th sequence has *type* $t$, where $1 \leq t \leq N/X$. We say that an element has *type* $t$ if it belongs to the $t$-th sorted sub-sequence. We assign *type* $t$ to the segments of the $t$ sorted sub-sequences in the same manner, since each segment contains elements of the same *type* $t$ by construction. We refer to the first element of each segment, which is also the minimum in it, as the *header* of the segment.

Let us now consider the sequence obtained after permuting the segments so that their headers are in stable sorted order. Let us focus on a generic element $u$ belonging to a segment, say $S'$ of type $t'$, and let $r_u$ be the number of elements greater than $u$ but residing before $u$ in the permutation we are examining. If we can bound $r_u$ from above with $((N/X) - 1)(Y - 1)$, the thesis holds. Specifically, we count the maximum number of elements contributing to $r_u$. Let us go by type. By the stability of the permutation of the segments following the headers, we know that the elements of type $t'$ have maintained the relative order they had before the segments were permuted and so they cannot contribute to $r_u$.

Let $x$ be the header of the segment $S'$ containing $u$. Let us evaluate the contribution to $r_u$ for the elements of type $t'' \neq t'$. Consider all segments of type $t''$: we are going to prove that at most one of them, say $S''$, can contain elements contributing to $r_u$. Precisely, $S''$ is the segment of type $t''$ having the largest header less than or equal to $x$. Let $y \leq x$ be the header of $S''$ and $z$ be one of such contributing elements in $S''$. Segment $S''$ is laid out before $S'$ by construction but $z > u$ by the definition of $r_u$. Note that there can be at most $Y - 1$ such elements $z$ in $S''$.

For any other segment of type $t''$, the elements in it cannot contribute to $r_u$. By the fact that we are dealing with sorted sub-sequences, for any such sequence of type $t''$, there are two possibilities for its segments $S'' \neq S'$:

(a) $S''$ contains all elements that are less than or equal to $y \leq x$ (i.e., $S''$ is laid out before $S''$); they do not contribute to $r_u$ by transitivity since $x \leq u$. 


Figure 7.2: Example of the permutation in Lemma 7.1 when $X = 6$ and $Y = 3$.

(b) $S'''$ contains all elements that are greater than or equal to $z > u \geq x$ (i.e., $S'''$ is laid out after $S''$); they do not contribute because the header of $S'''$ is strictly larger than $x$ by transitivity and so $S'''$ is laid out after $S''$.

Summing up, the total contribution to $r_u$ for the elements of type $t'' \neq t'$ is at most $Y - 1$ (a subset of the elements in $S''$). Since there are $\frac{N}{X} - 1$ different types other than $t'$, we obtain the wanted upper bound for $r_u$. \hfill \square

Starting from this particular permutation (an example of which can be seen in Figure 7.3.1), we can obtain the sorted sequence simply by merging each segment, from the leftmost to the rightmost one, with the elements contained in the previous $(N/X) - 1$ segments. However we cannot follow precisely that approach since the final merging process is still too costly both for the work and cache complexity. As we will see, we can pay only $O(N)$ time and $O(N/B)$ block transfers for the final merging step of any recursive call operating over $N$ elements. Therefore, we refine this property as described next.

### 7.3.2 The algorithm

The input sequence $A$ is conceptually divided into contiguous portions:

- The *complete segments*, each of size $S = \log N$ with the exception of the rightmost one, the *short segment*. The minimum element of a generic segment $s$ is the *header* of it; let us denote the set of headers with $\gamma$.

- The sequence is further divided into *complete sub-sequences* containing $\sqrt{N}$ complete segments, excluding the rightmost one called the *short sub-sequence*. Let $R$ be the number of complete sub-sequences; clearly $R \leq \sqrt{N}/S$.

---

Proximity Mergesort$(A, N)$
1. Calculate $S$, $R$ and the starting position of the short sub-sequence.

2. Recursively sort the complete sub-sequences $A_1, \ldots, A_R$.

3. Move the headers of the segments in a contiguous zone of memory and sort them with the binary mergesort.

4. Find a set $\sigma$ of $R$ equally spaced elements in the sorted set of headers and move the elements of $\sigma$ in a contiguous zone of memory.

5. Recover the original order of the remaining $R(\sqrt{N} - 1)$ headers in $\gamma - \sigma$ and move them back in their respective sub-sequences.

6. Recover the original order of the headers in $\sigma$.

7. Move the set of segments corresponding to the headers in $\sigma$ in a contiguous area of memory and sort with the binary mergesort the headers in $\sigma$. During the sorting process, each time a header is moved move also the whole corresponding segment.

8. Allocate a new contiguous sequence $Z$ of memory divided into $R$ sub-zones $Z_1, \ldots, Z_R$ of size $S\sqrt{N}$ (the same of a complete sub-sequence). Copy the segments corresponding to $\sigma$ in the last $S - 1$ positions of each sub-zone, following the order of $\sigma$.

9. Distribute the segments corresponding to $\gamma - \sigma$ in these sub-zones according to the order of set $\sigma$. At the end of the distribution, move back to $A_1, \ldots, A_R$ the headers in $\sigma$ and the elements in $Z_1, \ldots, Z_R$.

10. Recursively sort the complete sub-sequences $A_1, \ldots, A_R$.

11. For each $i = 1, \ldots, R-1$, use a binary merging algorithm to merge the elements in zones $A_i$ and $A_{i+1}$.

12. Sort the elements in the short sub-sequence and in the short segment with the binary mergesort. Merge that sorted sequence of $< S(\sqrt{N} + 1)$ elements and the one obtained in step 11.

---

### 7.3.2.1 The strategy

We recursively sort each complete sub-sequence. However, instead of sorting the complete segments on their headers, we find a perfectly balanced sample set of $R$ equidistant headers and redistribute the complete segments using that set. Thus we obtain $R$ new complete sub-sequences $A_1, \ldots, A_R$ such that:
For each \( i = 1, \ldots, R - 1 \), the header of the segment with maximum
header in \( A_i \) is less than or equal to the header of the segment with
minimum header in \( A_{i+1} \).

With the second recursive sorting of the sub-sequences we achieve a permutation
that has the same proximity property of that in the observation done above. Namely:

\[
\text{No element is more than } D = (R - 1)(S - 1) \text{ positions beyond its }
f\text{inal position in the full-sorted sequence.}
\]

Since \( R \leq \sqrt{N}/S \), we have that \( D \leq \sqrt{N} \). Moreover, that permutation is
composed of \( R \) sorted sub-sequences of \( S\sqrt{N} \) elements, instead of \( N/S \) sub-sequences
with only \( S \) elements each, as in the observation. Hence, the “left to right” binary
merging process in step 11 ends with the sorted sequence of elements (short segment
and short sub-sequence excluded). Note that our values for the size of a segment
\((S)\) and for the size of a complete sub-sequence \((S\sqrt{N})\) are quite “large” for the
problem and other choices are possible (for example \( \sqrt{SN} \) for the size of a complete
sub-sequence).

### 7.3.2.2 Distribution of segments

Before we proceed any further, we should give a comment about step 9. If we
perform this step processing the sub-sequences from the leftmost to the rightmost
one, distributing the segments of each sub-sequence along all the existing sub-zones
in one single run, we incur in a number of cache misses that is

\[
O \left( R^2 \right) = O \left( \frac{N}{\log^2 N} \right)
\]

and this term can be \( \omega(N/B) \) when \( B = \omega(\log^2 N) \). As we will see, each non
recursive step in the Proximity Mergesort is required to have a cache complexity
\( O(N/B) \) in order to obtain the cache optimality.

As already anticipated in Section 8.1, we use the special processing order intro-
duced by Frigo et al. [Frigo, Leiserson, Prokop, and Ramachandran, 1999] with the
following procedure.

**Distribute** \((i, j, m)\)

1: IF \( m = 1 \) THEN
2: COPY ELEMENTS \((i, j)\)
3: ELSE
4: DISTRIBUTED \((i, j, m/2)\)
5: DISTRIBUTED \((i + m/2, j, m/2)\)
6: DISTRIBUTED \((i, j + m/2, m/2)\)
7: \textbf{Distribute}(i + m/2, j + m/2, m/2)

Unlike the Distribution sort in [Frigo, Leiserson, Prokop, and Ramachandran, 1999], we do not need the splitting technique to find our sample set: \( \sigma \) is composed of \( R \) perfectly equidistant elements in the sorted set of headers \( \gamma \). Furthermore, in the base case \( (m = 1) \), we do not copy single elements but whole segments. Therefore, in our particular case, the invocation \textit{CopyElements}(i, j) scans the remaining segments of the complete sub-sequence \( i \) (starting from the leftmost one) and moves them in the sub-zone \( j \) until a segment with the header greater than the sample of \( j \) is found. In summary, step 9 of our algorithm is realized invoking \textit{Distribute}(1, 1, R).

The following Lemma can be inferred from the cache complexity analysis that Frigo et al. and Prokop [Frigo, Leiserson, Prokop, and Ramachandran, 1999, Prokop, 1999] give for the Distribution sort.

\textbf{Lemma 7.2} The invocation \textit{Distribute}(1, 1, m) incurs \( O(B + m^2/B + d/B) \) cache misses, where \( d \) is the total number of elements distributed.

\subsection{Complexity analysis}

Let us start the work and cache complexity analysis estimating the cost of each non-recursive step of the Proximity Mergesort.

\textbf{Lemma 7.3} The non-recursive steps of the Proximity Mergesort have a total work complexity \( O(n) \) and a total cache complexity \( O(N/B) \).

\textit{Proof}: Step 1 is trivial. For step 3, the scanning and the copy of headers can be done in \( O(N/S) \) time and \( O(N/(SB)) = O(N/B) \) block transfers. The sorting part is done with the normal stable binary mergesort. This algorithm is cache-oblivious and requires a number of cache misses that is

\[ O \left( \frac{N}{BS} \log N \right) = O \left( \frac{N}{B} \right) \]

if \( N > cM \) (as we shall see, \( N \leq cM \) will be the base case condition for the cache complexity recurrence). Since the set \( \sigma \) is sorted, step 4 takes \( O(R) \) time and incurs \( O(N/B) \) cache misses. In step 5 and 6 we use again the binary mergesort; we can perform these two steps in time \( O(N) \) and with a number of cache misses

\[ O \left( \frac{N}{BS} \log N \right) = O \left( \frac{N}{B} \right) \]

whenever \( N > cM \). Step 7 can be performed with a simple variation of the binary mergesort at a cost of \( O(N/B) \) cache misses as in steps 3, 5 and 6. Steps 8 and 11 have the same asymptotic complexity of a scanning. Since the short segment and the short sub-sequence contain less than \( S(\sqrt{N}+1) \) elements, the binary mergesort invocation in step 12 costs
\[ O \left( \frac{S(\sqrt{N} + 1)}{B} \log N \right) = O \left( \frac{N}{B} \right), \]
if \( N > cM \). The other part of this step has the same asymptotic complexity as that of scanning.

Step 9 is realized invoking \textit{Distribute}(1, 1, R). We distribute whole segments instead of single elements but this can be seen as a particular case. Hence, by Lemma 7.2, step 9 requires a number of cache misses

\[ O \left( B + \frac{R^2}{B} + \frac{N}{B} \right) = O \left( \frac{N}{B} \right). \]

\[ \square \]

\textbf{Theorem 7.1} Proximity Mergesort has a work complexity \( O(N \log N) \) and a cache complexity \( O((N/B) \log_M N) \), if used to sort \( N \) elements.

\textit{Proof}: The recurrence for the work complexity is

\[ W(N) = 2R \cdot W(S\sqrt{N}) + O(N), \]
and the recurrence for the cache complexity is

\[ Q(N) \leq \begin{cases} O(N/B) & N \leq cM \\ 2R \cdot Q(S\sqrt{N}) + O(N/B) & \text{otherwise} \end{cases} \]

\[ \square \]

\section{7.4 Optimal In-Place Sorting}

In this section we describe how to reduce the quantity of work space used by Proximity Mergesort to \( O(1) \) auxiliary locations.

\textbf{Stability.} A sorting algorithm is \textit{stable} if it preserves the relative ordering of different occurrences of the same element. The property of stability is important when satellite data are carried around with the element being sorted. For example, a stable sorting algorithm is a necessary sub-procedure in the Radix sort, a linear time sorting algorithm that operates over sequences of integers and enforces their positional encoding.

With the transformation of the Proximity Mergesort into a space optimal we lose the \textit{stability} of our algorithm, property that in a space sub-optimal setting can be easily achieved pairing, for any \( 1 \leq i \leq N \), the element in position \( i \) in the original input sequence with a single integer of auxiliary data: the value \( i \) itself. As we will see (Section 7.5), the stability property seems to collide irreparably with the techniques we use to obtain a space-efficient algorithm.
Sources of space sub-optimality. We have to deal with a lot of circumstances in which the Proximity Mergesort exploits the auxiliary space.

- Auxiliary space is used to store indices, pointers, counters or, in general, integer of $O(\log N)$ bits.

- During the whole computation we use auxiliary space to permute efficiently various groups of elements.

- We have to deal with all the auxiliary space implicitly employed by recursion (recursion stack), both in the main algorithm and in the invocation \textit{Distribute}(1, 1, R) used to perform step 9.

7.4.1 Encoding the integers

Stealing bits in the Cache-Oblivious model. To eliminate the need for auxiliary locations devoted to store integers, we will use once again the classic technique of \textit{bit stealing} (see Chapter 4), being a widely used tool for achieving minimum space usage both in algorithms and data structures. We will give just a quick reminder before starting to use this technique. We can encode a single bit using the relative order of two distinct elements $x, y, x < y$: if $x$ precedes $y$ they encode a 0, otherwise they encode a 1. An integer of $\log N$ bits can be encoded using $\log N$ pairs of distinct elements, and, if the positions of these pairs are implicitly known, we can use this encoded integer as a normal one. The drawback is that a single manipulation of such encoded integers uses $O(\log N)$ operations and $O \left( \frac{\log N}{B} \right)$ cache misses (this is true only for particular positioning of the pairs, for example if the pairs are stored in a constant number of sequences of contiguous locations).

Estimating the quantity of bits to be stolen. Let us estimate how many bits Proximity Mergesort needs.

For steps 5 and 6, we need $\log N$ bits for each header, so that its original position after step 2 can be recorded. For step 9 we need a constant number of integers of $\log N$ bits for each sub-zone and each complete sub-sequence (\textit{Distribute} needs to know things like the number of remaining segments in a complete sub-sequence). So, we need a total of

$$a \frac{N}{S} \log N$$

bits,

for a properly chosen constant $a$. It is worth noting that we do not have to maintain the integers associated with sub-zones, when we perform the recursive calls in step 10.
Collecting the pairs of elements to steal bits. All the $P = a \frac{N}{S} \log N$ pairs of distinct elements are collected in the following way.

(i) With an in-place selection algorithm we select the $P$th and the $(N - P + 1)$st (in sorted order) elements of the input sequence. The optimality of the selection algorithm is not strictly required, because for this task we can employ $O(N \log N)$ operations and incur in $O((N/B) \log M N)$ cache misses. However, the optimal in-place selection algorithm of Lai and Wood [Lai and Wood, 1988] can be slightly modified to achieve the optimal cache complexity.

(ii) Then, we partition in-place the input sequence so that we end with the $P$ smallest elements in the $P$ leftmost positions and the $P$ largest elements in the $P$ rightmost positions. This task can be easily done in-place and cache-obliviously with $O(N)$ operations and $O(N/B)$ cache misses (we can lose the stability).

(iii) Finally, we sort the two sub-sequences of $P$ elements with the in-place unstable binary mergesort.

This last step raises a problem. Since $S = \log N$, we have that $P = O(N)$ and so the cache complexity of this step is $O((N/B) \log N)$. To solve this problem is sufficient to set

$$S = \log^2 N.$$

This new value of parameter $S$ does not change the asymptotic order of the solution of the recurrence for the cache complexity in Theorem 7.1 and reduces the cache complexity of step (iii) to $O(N/B)$. We use an in-place unstable mergesort based on a classical linear time in-place merging algorithm like the one of Kronrod [Kronrod, 1969, Salowe and Steiger, 1987]. This algorithm can be easily modified to be cache optimal.

The reduced problem. Let $E$ and $E'$ be the zones containing the $P$ smallest elements and the $P$ largest elements, respectively. If the $P$th and the $(N - P + 1)$st elements selected in step (i) are equal then at the end of step (iii) the input sequence is sorted and we are done.

Otherwise only the sub-sequence, say $A'$, of elements between $E$ and $E'$ must be sorted. Moreover, we are sure that the $i$th element of $E$ is distinct from the $i$th element of $E'$, for each $i = 1, \ldots, P$. So we have the needed $P$ pairs of distinct elements. We can encode any quantity $t$ of bits with $2t$ elements stored in two contiguous portions of $t$ locations, one portion in $E$ and the other in $E'$. For simplicity, from now on we will write “$x$ contiguous encoding pairs” even if these $x$ pairs are stored in two contiguous areas. When also $A'$ is sorted, it will be easy to recover the sorted order for $E$ and $E'$ with a straightforward variant of linear scanning.
Changes in the complexity analysis. Encoding the integers increases the work complexity and the cache complexity of steps 3, 5 and 6 in the main algorithm. We have to employ $R\sqrt{N}\log N$, $R(\sqrt{N} - 1)\log N$ and $R\log N$ contiguous encoding pairs with steps 3, 5 and 6, respectively. This is done to encode the position of each header in the sequence of elements obtained after step 2. Since a manipulation or a simple access of an encoded integer require the scanning of $O(\log N)$ contiguous locations, the complexity of step 3 increases from $O(N)$ to $O(N\log N)$ operations and from $O\left(\frac{N}{B^S}\log N\right)$ to $O\left(\frac{N\log N}{B^S}\log N\right)$ cache misses. Fortunately, with the new choice of $S$ the complexity for this step is still $O(N)$ operations and $O(N/B)$ cache misses. Analogous observations can be done for steps 5 and 6.

The use of encoded integers affects other parts of Proximity Mergesort. For example Distribute (used to perform step 9) is modified to use the encoded integers that store the state of complete sub-sequences and sub-zones (i.e., the counter of the number of segment contained). Also the local variables (i.e., $i, j, m$) need a particular treatment in order to remove all the auxiliary space used by recursion.

With these variations we obtain an algorithm that does not explicitly store any auxiliary integer used in the computation and that has the same complexity of the original Proximity Mergesort.

7.4.2 Creating “virtual” working areas.

In order to eliminate the need of working areas to permute efficiently the elements, we use the ingenious technique of the internal buffering (see Chapter 4) whose credit is due to Kronrod [Kronrod, 1969, Salowe and Steiger, 1987]. Instead of moving an element to an empty location, thus performing a destructive occupying, we interchange this “active” element, (an element is “active” when we are performing part of our computation on it) with an “inactive” one. Clearly, in order to complete our computation, an inactive element should become active sooner or later. We now show how to use this simple idea for our purpose.
After the creation of the encoding areas \(E\) and \(E'\), we have to sort only the area \(A'\) between \(E\) and \(E'\). Let \(N'\) be the size of \(A'\) and let \(dN''\) be the size of the largest working area used by Proximity Mergesort when invoked to sort a sequence of \(N''\) elements. We divide the sequence \(A'\) into two parts: the active area with \(\frac{N'}{d+1}\) active elements, and the inactive area with \(\frac{N'}{d+1}\) inactive elements. We use the inactive area to allocate the various working areas that the Proximity Mergesort requires for sorting the active area.

We need a method to distinguish the active elements from the inactive ones during the computation. By inspecting the algorithm, it is clear that the working areas are filled and emptied with a scanning order and so the task of distinguishing the two types of elements is quite easy. For example, in step 9 each complete subsequence is emptied from left to right and each sub-zone is filled from left to right (see Figure 7.3). So the simple counter of the segments that are stored in this subsequence or this sub-zone is sufficient to know where the inactive (and the active) elements are.

When the active area is sorted, we divide the \(\frac{N'}{d+1}\) elements of the former inactive area into a new inactive area of \(N'\frac{d}{(d+1)^2}\) elements and a new active area of \(N'\frac{d}{(d+1)^2}\) elements. Then we apply the Proximity Mergesort to the new active area (see Figure 7.4). This process is iterated until the areas have a constant number of elements. Let us denote the pair of active/inactive areas of the \(j\)th iteration with \((A_j, I_j)\). Let \(l\) be the number of iterations of this process.

Let \(s(i)\) be the size of the problem at iteration \(i\), \(i = 0, \ldots, l - 1\). The cache complexity for the iterated sorting process is:

\[
Q_{iter} = \sum_{i : s(i) > B} Q_{PM}(s(i)) + \sum_{i : s(i) \leq B} Q_{PM}(s(i)).
\]

where \(Q_{PM}\) is the cache complexity of the Proximity Mergesort. Since \(s(i) = N'\frac{d}{(d+1)^2}\) and \(Q_{PM}(s(i)) = O((s(i)/B) \log_M s(i))\), we have that the contribution of the first sum in \(Q_{iter}\) is \(O((N'/B) \log_M N')\). The second sum in \(Q_{iter}\) contributes to the cache complexity with \(O(1)\) cache misses, because all the elements sorted in the iterations involved in this sum are stored in a constant number of contiguous blocks. Therefore the iterated sorting process of \(A'\) can take \(O((N'/B) \log_M N')\) cache misses. Analogous calculations can be done for the work complexity.

When the iterated sorting process ends, we use an optimal in-place unstable merging algorithm to obtain the final sorted sequence. First, we merge \(A_l\) with \(A_{l-1}\) obtaining the sequence \(B_1\). Then, we merge \(B_1\) with \(A_{l-2}\) obtaining \(B_2\) and so on. At the end of this iterated merging process the elements in \(A'\) are sorted. The linear time in-place merging algorithm of Kronrod [Kronrod, 1969, Salowe and Steiger, 1987] can be easily modified to be cache optimal. With calculations similar to the ones for the iterated sorting process, we can conclude that the iterated merging process uses \(O(N)\) operations and incurs in \(O(N/B)\) cache misses.
Figure 7.4: Encoding areas $E$, $E'$ and active/inactive areas pairs $(A_j, I_j)$ for $j = 0, 1, 2$

7.5 Conclusions and Open Problems

We have described the first space, work and cache optimal sorting algorithm for the Cache-Oblivious model. Furthermore, the first version of Proximity Mergesort that uses $O(N)$ auxiliary space, introduces a new approach to the sorting problem in the Cache-Oblivious model.

The space optimal version of our algorithm is not stable. The main difficulty towards the stability relies on the technique used to simulate the working areas. When the inactive elements are moved back in their inactive areas, their original order is lost and cannot be recovered if there are equal inactive elements.

The simulation of working areas with this technique underlies all the existent in-place merging algorithms. Since all these solutions are quite complicated and since the in-place sorting problem in the Cache-Oblivious model is clearly harder than the space optimal merging problem, it is an interesting open problem how to achieve the stability.
Chapter 8

What if the Universe is Multidimensional?

Abstract

In this chapter we study the problem of determining the time and space complexity of optimal comparison-based sorting when the elements belong to a multidimensional domain. Vectors, strings, records, multi-dimensional points, and multi-digit numbers are notable examples for which it is unrealistic to assume a constant-time comparison cost. Consequently, elements are modeled as vectors of length $k$, so that comparing and moving any such element requires $O(k)$ time. Moreover, accessing one of the element’s components takes $O(1)$ time (no hashing or bit manipulation is allowed). The resulting model, the In-Place RAM Model for Vectors (see Chapter 2), naturally extends the comparison model to elements of non-constant length, $k$, and represents a challenge on how to give optimal asymptotic bounds simultaneously holding in terms of time and space.

One example of problems that stimulated algorithmic work in this direction is how to perform in-place searching on a set of $n$ elements, namely, requiring just $O(1)$ auxiliary data locations besides the storage for the input. When elements are in lexicographic (or alphabetic) order in [Andersson, Hagerup, Håstad, and Petersson, 1994, Andersson, Håstad, and Petersson, 1995b, Andersson, Hagerup, Håstad, and Petersson, 2001] it was shown that the complexity of in-place searching is worse than that of searching with $O(n)$ auxiliary data locations (e.g. [Manber and Myers, 1993]). Using permutations other than those resulting from sorting is a way to reach optimality: in [Franceschini and Grossi, 2004b] (see Chapter 10) it was shown that for any set of $n$ elements in lexicographic order, there exists a permutation of them allowing for $O(n + \log n)$ search time using $O(1)$ auxiliary locations.

In this chapter we consider space optimal sorting, which is another significant example. Any optimal in-place sorting algorithm for constant-sized elements can be turned into an $O(nk \log n)$-time in-place algorithm for $n$ elements, losing time optimality in this way. If the number of comparison is to
be minimized, the best up-to-date result is $n \log n + O(nk \log^* n)$ comparisons and $n \log n + O(nk)$ element movements in [Munro and Raman, 1991]. Since each element movement takes $O(k)$ time, the time complexity sums up to $O(nk^2 + nk \log n)$. For the same reason, the multikey Quicksort analyzed in [Bentley and Sedgewick, 1997] yields a non-optimal cost of $O(nk \log n)$, since it requires $O(n \log n)$ moves. As can be seen, optimality is known for in-place searching but it is unknown for in-place sorting in this case.

In this chapter, we give a positive answer to this foundational problem by presenting the first algorithm for lexicographically sorting $n$ elements in $O(nk + n \log n)$ time using $O(1)$ data locations, which is simultaneously optimal in time and space. As previously mentioned, the results for efficient in-place searching assume to start with elements that are already in lexicographic order. Before, we were unable to preprocess an arbitrary arrangement of the elements to this end, since no optimal in-place algorithm was known for the lexicographic sorting problem itself. We could prove the existence of certain permutations of the elements leading to efficient in-place searching; yet, we could not construct them with optimal time and space bounds. Now, we are able to provide optimal in-place preprocessing for in-place searching when the elements are initially arranged in any arbitrary order. Another effect is that sorting bulky records can be performed in place optimally by direct exchanges, thus avoiding the use of $O(n)$ auxiliary memory for storing the pointers to the records as required by Knuth’s address table sorting [Knuth, 1973].

The presentation in this chapter is based on the paper [Franceschini and Grossi, 2005a].

### 8.1 Introduction

We study the computational complexity of the classical problem of comparison-based sorting by considering the case in which the input elements are drawn from a multidimensional domain. We are interested in evaluating asymptotically the influence of the dimension $k$ in terms of the optimality of the bounds required for sorting. We aim at minimizing simultaneously the time and space bounds under the assumption that the elements are vectors $x \in \mathcal{U}^k$ of $k$ scalar components over a totally ordered, possibly unbounded set $\mathcal{U}$.

The presentation in this chapter is based on the paper [Franceschini and Grossi, 2005a].

#### 8.1.1 The model

Denoting the $i$th scalar component of vector $x$ by $x(i)$ for $1 \leq i \leq k$, we indicate the vector’s chunks by $x(i,j)$, which are the contiguous portions of $x$ consisting of
8.1. INTRODUCTION

$x(i), x(i + 1), \ldots, x(j)$, where $1 \leq i \leq j \leq k$. The lexicographic (or alphabetic) order, $x \leq y$, is defined in terms of the scalar components: either $x(1) < y(1)$ or recursively $x(2,k) \leq y(2,k)$ for $x(1) = y(1)$.

The model we are going to deal with is the In-Place RAM Model for Vectors (see Chapter 2). This model naturally extends the comparison model to elements of non-constant length. Although some of the previous work uses the model in which chunks of the vectors can also be exchanged, we prefer to impose the requirement that vectors are indivisible and should be entirely exchanged. We obtain optimal bounds with this strict model without requiring the extra power of the other model. One example of results fulfilling the strict model is [Franceschini and Grossi, 2004b] (see Chapter 10).

This model is useful for studying, in an abstract way, the complexity of in-place sorting and searching for a variety of objects of length $k$: fixed-length strings, $k$-field records, $k$-dimensional points, $k$-digit numbers, etc. In case of fixed length strings, for example, we can fit the model to strings in the C language, where the $n$ vectorial locations are declared as `char V[n x k]` by assigning entries $V[(i - 1)k]$, $\ldots, V[ik - 1]$ to the $i$th vectorial location, $1 \leq i \leq n$, and using $O(1)$ local `int` variables as auxiliary locations. Note that no string terminator is needed in $V$; also, using an array of pointers `char *p[n]` referring to the beginnings of vectors in $V$ (i.e., $p[j] = V + jk$ for $0 \leq j < n$) would violate the model by requiring $O(n)$ auxiliary locations.

The In-Place RAM Model for Vectors easily extends to $k$-field records in $U_1 \times U_2 \times \cdots \times U_k$, but we prefer to keep the notation simple. We can define the lexicographic order $\leq$ consistently with relations $<_i$ and $=_i$ on $U_i$ for each field of the records, $1 \leq i \leq k$, without any significant change in our bounds. Note that the model allows for variable-length fields in the fixed-length records.

8.1.2 Fundamental problems in multidimensional domains

One significant example of the augmented complexity of fundamental problems when extended to a multidimensional domain is how to perform in-place searching on a set $V$ of $n$ vectors.

The problem has been introduced in [Hirschberg, 1978], with upper bounds of $O(k + n)$ time and $O(k \log n)$ time. The former is obtained by a full scan of $V$ while the latter is a simple binary search on $V$. The lower bound of $\Omega(k + \log n)$ follows quite easily. The logarithmic term in $n$ comes from the decision tree for searching in a set of $n$ vectors, while the linear term in $k$ comes from the need of reading all the $k$ symbols in the search vector.

The first nontrivial upper bound is

$$O \left( \frac{k \log n}{\log k} \right)$$

appeared in [Hirschberg, 1980], and later improved by Kosaraju [Kosaraju, 1979] to
\( O\left(k\sqrt{\log n} + \log n\right). \)

With sophisticated techniques for proving upper and lower bounds on the complexity of searching \( \mathcal{V} \) in lexicographic order it has been proved in [Andersson, Hagerup, Håstad, and Petersson, 1994, 2001, Andersson, Håstad, and Petersson, 1995b] that searching a lexicographically sorted set of vectors without pre-processing requires

\[
\Theta\left(\frac{k\log\log n}{\log\log(4 + \frac{k\log\log n}{\log n})} + k + \log n\right)
\]

This bound is worse than \( \Theta(k + \log n) \), obtained by searching \( \mathcal{V} \) plus \( O(n) \) auxiliary locations (e.g. in [Manber and Myers, 1993]).

Using permutations other than those resulting from sorting is a way to reach optimality: in [Franceschini and Grossi, 2004b] (see Chapter 10) it has been shown that for any set \( \mathcal{V} \) of \( n \) vectors, there exists a permutation of them allowing for \( O(k + \log n) \) search time using \( O(1) \) auxiliary locations.

Space optimal sorting is an even more intriguing example in this scenario. Any optimal in-place sorting algorithm for constant-sized elements can be turned into an \( O(nk\log n) \)-time in-place algorithm for vectors, loosing optimality in this way. The lower bound of

\( \Omega(nk + n\log n) \)

easily derives from decision trees [Knuth, 1973]. If the number of comparisons is to be minimized, the best up-to-date result for space optimal sorting performs

\[
\begin{align*}
\text{scalar comparisons} & \quad n\log n + O(nk\log^* n) \\
\text{vector exchanges} & \quad n\log n + O(nk)
\end{align*}
\]

appeared in [Munro and Raman, 1991]. Since each vector exchange takes \( O(k) \) time, the time complexity sums up to \( O(nk^2 + nk\log n) \).

The multikey Quicksort analyzed in [Bentley and Sedgewick, 1997] yields a non-optimal algorithm of cost \( O(nk\log n) \) when adapted to run in the in-place model for vectors, since it requires \( O(n\log n) \) vector exchanges. The original version of the algorithm takes \( O(nk + n\log n) \) time since it can exploit \( O(n) \) auxiliary locations to store the pointers to the vectors. Sorting exchanges the pointers rather than the vectors, following the address table sorting suggested in [Knuth, 1973] at page 74.

When records should be physically rearranged in storage so that their elements are in order, Knuth writes that address table sorting is preferable to avoid moving the bulky records around. The reason is apparent: each move of a record is expensive while moving a pointer to the record is more efficient. The in-place model for vectors
fits this intuition formally by introducing a cost of \( O(k) \) for exchanging vectors. However, it forbids the use of an array of \( O(n) \) auxiliary pointers as done in address table sorting. In other words, exchanging elements is only doable by exchanging their bulky records rather than their pointers. (We identify the records with their elements for the sake of discussion.)

As can be seen, the cost of exchanging and comparing vectors is the major bottleneck towards optimality. The in-place RAM model for vectors is centered around this feature and requires algorithms operating with \( O(n) \) vector moves. Recently, in [Franceschini and Geffert, 2003, 2005] (see Chapter 5) have devised a comparison and space optimal algorithm for constant-sized elements with \( O(n) \) moves. Subsequent results [Franceschini, 2004, 2005a] (see Chapters 7 and 6) have shown how to achieve cache-obliviousness or stableness for in-place sorting. However, the \( O(k) \)-time cost of each vector comparison makes these methods non-optimal in the multidimensional setting. The bit encoding for vectors presented in [Franceschini and Grossi, 2004b] (see Chapter 10) cannot help either, as it assumes that vectors are initially sorted while this is actually the major goal in this chapter.

### 8.1.3 Implications of our result

The above discussion highlights the fact that the known algorithms, to the best of our knowledge, are unable to simultaneously achieve time optimality and space optimality for sorting vectors in place. Put it into simple words, optimality is known for in-place searching while it is unknown for in-place sorting in this case. Our main result is that of obtaining the first optimal bounds for sorting an arbitrary set of \( n \) vectors in place, taking \( O(nk+n \log n) \) time and using \( O(1) \) auxiliary locations.

An implication of our result is that we resolve the apparent anomaly arising from in-place searching and sorting. As previously mentioned, the results for efficient in-place searching in [Andersson, Hagerup, Håstad, and Petersson, 1994, 2001, Andersson, Håstad, and Petersson, 1995b, Franceschini and Grossi, 2004b, Hirschberg, 1978, 1980, Kosaraju, 1979] stem from the fact that the vectors are already given in lexicographic order. Before the algorithm we will present in this chapter, we could not preprocess an arbitrary arrangement of vectors to this end, since no optimal in-place algorithm was known for the lexicographic sorting problem itself. While we could prove the existence of certain permutations of the vectors leading to efficient in-place searching, we were not yet able to construct them with optimal space and time bounds. Now, we can provide optimal in-place preprocessing for efficient in-place searching when the vectors are initially arranged in any arbitrary order, with a preprocessing cost of \( O(nk+n \log n) \).

Asymptotically speaking, another implication is that sorting bulky records can be done optimally in place by exchanging them directly without using the \( O(n) \) auxiliary locations required by Knuth’s address table sorting. We remark that our bounds hold for internal sorting; it is an open problem to establish the complexity of external sorting for a set of vectors even if the algorithms are not in-place [Arge,
Ferragina, Grossi, and Vitter, 1997].

Interestingly, the results of Arge, Ferragina, Grossi and Vitter [Arge, Ferragina, Grossi, and Vitter, 1997] show that breaking up long strings into smaller substrings gives more computational power when sorting in external memory. In our internal-memory setting, however, breaking vectors does not give more power for in-place sorting by comparisons. We can trivially derive a lower bound of $\Omega(nk + n \log n)$ for the model where we can break the vectors into chunks to be exchanged. Since our bound of $O(nk + n \log n)$ also holds for the latter model, we easily derive its optimality.

8.1.4 Chapter organization

This chapter is organized as follows. We provide a high-level description of our sorting algorithm in Section 8.3. We then show how to steal and encode “heavy”, or slow, bits by using $o(n)$ vectors in Section 8.4. We thus reduce the original in-place sorting problem to that of sorting $n - o(n)$ vectors with the help of $o(n)$ heavy bits. We then further reduce the latter to the problem of sorting blocks of $o(n)$ vectors each, using a further block of $o(n)$ vectors for internal buffering (see Section 8.5). Finally, we show how to sort a given block in Section 8.6.

8.2 Comparison to Recent Work

As we have seen, some recent advances for in-place sorting [Franceschini, 2004, 2005a, Franceschini and Geffert, 2003] have been presented in the literature. The same holds for in-place searching [Franceschini and Grossi, 2004b]. There are several major problems concerned with the algorithms described in [Franceschini, 2004, 2005a, Franceschini and Geffert, 2003, Franceschini and Grossi, 2004b] that make their extension to a multidimensional setting difficult. We discuss them in some detail.

8.2.1 Stealing bits from vectors

The first problem concerns the technique for determining which pairs of distinct elements should encode bits. If applied to our setting (see Section 8.4), the distributive technique in [Franceschini, 2005a, Franceschini and Geffert, 2003] cannot guarantee that the longest common prefix, lcp from now on, of the selected pairs of vectors is within a range of $O(1 + k/\log n)$ consecutive values as we will do instead. This inevitably gives $O(k)$ decoding time for a single bit, yielding a non-optimal complexity.

The bit encoding technique for strings presented in [Franceschini and Grossi, 2004b] cannot help either, since it assumes that the vectors are initially sorted while this is actually the major goal in this chapter. As we will see in Section 8.4, even if
we perform a preliminary sorting of zones $\mathcal{L}$ and $\mathcal{R}$, the technique in [Franceschini and Grossi, 2004b] cannot provide $O(n / \log^2 n)$ encoded bits with $O(1 + k / \log n)$ decoding time each.

### 8.2.2 Internal buffering with vectors

The second problem is about the internal buffering technique for simulating a temporary area (see Section 8.5). Usually a sorting algorithm employing this technique will maintain, almost throughout the entire execution, two sets $\mathcal{M}'$ and $\mathcal{M}_B$ respectively of sorting and placeholder elements, using the latter to sort the former.

In [Franceschini and Geffert, 2003] (see Chapter 5), both $\mathcal{M}'$ and $\mathcal{M}_B$ can change during the execution: $\omega(1)$ different sets $\mathcal{M}_B$ are used. In our case $\mathcal{M}'$ is the only changing while $\mathcal{M}_B$ remains the same all the time and, due to its limited cardinality, can be sorted in-place by Lemma 8.1, but we are allowed to apply this lemma only a constant number of times! This is not possible in [Franceschini and Geffert, 2003]. Another difficulty is that we have to devise an alternative way to distinguish between normal elements and placeholders. In [Franceschini and Geffert, 2003] a buffer separator is maintained, i.e., an element $y$ such that $x < y \leq z$ for any $x \in \mathcal{M}'$ and $z \in \mathcal{M}_B$. When the algorithm has to find out if a particular element is a normal one or a placeholder, it simply compares the element with the buffer separator $y$.

The problem with the buffer separator solution when transposed in the vector setting is clear: $O(k)$ scalar comparisons have to be done in the general case. Due to the extensive use of comparisons with buffer separators in [Franceschini and Geffert, 2003], this technique would generate a total number of $O(nk \log n)$ scalar comparison. Instead, our solution relies on the internal structure of $\mathcal{M}_B$ (see Section 8.6) and entangles the computation even stronger in the use of encoded bits, so we need to use heavy bits with $O(1 + k / \log n)$ decoding time each.

### 8.2.3 Sorting few vectors

The third problem concerns the distribution technique used to sort a set of normal vectors $S$ using a set of placeholders $B$. From an high level view, the technique grows a large set of buckets of polylogarithmic size using a dynamic routing level that can route any element in $S$ to its bucket in $O(\log n)$ time. After $S$ has been distributed, any member of the resulting set of buckets is sorted independently. Any adaptation of the multi-way Heapsort used to sort each bucket in [Franceschini and Geffert, 2003] to our setting would be based on two possible choices. The first one avoids completely the use of encoded information and relies only on the rigid structure of the multi-way heap; unfortunately, at the best of our knowledge there is no way to do so without incurring in high scalar comparison costs due to the frequent rescanning of components of vectors already examined.
The second possible choice would be trying to "enrich" the heap with encoded information that essentially keep track of the prefixes of vectors in order to avoid rescanning. Unfortunately, also this solution seems to be unpracticable given the high costs of the use of encoded bits in the vector setting.

Our approach is based on the recursive application of the session sorting scheme described in Section 8.5. The scheme can be seen as a mix of sorting by merging and by distribution at the same time (see Theorem 8.1). It is a non-trivial generalization of a scheme originally presented for two-way in-place merging [Franceschini, 2004, Kronrod, 1969]. We use the session scheme at two different places of our sorting algorithm, thus relating two apparently distinct phases.

8.3 High-Level Description

We present our in-place sorting algorithm for vectors in a top-down fashion. We describe how to reduce the original problem to a sequence of simpler sorting problems to be solved. In our description, we identify the $n$ input vectors in $\mathcal{V}$ with their vectorial locations. At the beginning of the computation, $\mathcal{V}[i]$ represents the $i$th vectorial location, for $1 \leq i \leq n$, and contains the $i$th input vector. At the end of the computation, $\mathcal{V}[i]$ contains the vector of rank $i$ after the sorting (ties for equal vectors are broken arbitrarily). During the intermediate steps, we solve several instances of a general vector sorting problem, denoted

$$GVSP\{m, p, h\}$$

(see Figure 8.1). Given $n$ vectors in $\mathcal{V}$, we refer to $GVSP\{m, p, h\}$ as the problem of sorting a subset of $m$ contiguous vectors in $\mathcal{V}$, using

- $O(1)$ auxiliary locations,
- $p$ vectors as placeholders (see Chapter 4) initially residing in a contiguous sub-sequence of $p$ locations in $\mathcal{V}$,
- $h$ heavy bits suitably encoded by $h$ pairs of vectors taken from two contiguous sub-sequences, each consisting of $h$ locations in $\mathcal{V}$,

under the requirement that $m + p + 2h \leq n$ and that the four sub-sequences of $h$, $p$, $m$, and $h$ vector locations, respectively, are pairwise disjoint as shown in Figure 8.1. We will define the role of placeholders and heavy bits in the rest of the section.

The general notation of $GVSP\{m, p, h\}$ is useful for expressing the various sorting instances that we get by reducing our initial problem, $GVSP\{n, 0, 0\}$, to simpler problems (with suitable values of $m$, $p$ and $h$). Some basic instances occur just a constant number of times in the reduction and are easy to solve.

**Lemma 8.1** Any instance of $GVSP\{O(n/\log n), 0, 0\}$ can be solved in $O(nk)$ time.
Proof: We employ the in-place mergesort of [Salowe and Steiger, 1987] and pay a slowdown of $O(k)$ in the time complexity, since we run it on $O(n/\log n)$ vectors, each of length $k$. The cost is $O(k \times (n/\log n) \log (n/\log n)) = O(nk)$ time. 

We now present the high-level structure of our reduction. In the following, for any two vectors $x$ and $y$, we denote the length of their longest common prefix by $lcp(x, y) = \max(\{0\} \cup \{1 \leq \ell \leq k : x(1, \ell) = y(1, \ell)\})$.

### 8.3.1 Heavy bits

To begin with, we reduce an instance of $GVSP\{n, 0, 0\}$ to a suitable instance of 

$$GVSP\{n - o(n), 0, O(n/\log^2 n)\}$$

plus a constant number of instances of $GVSP\{O(n/\log n), 0, 0\}$. We partition in-place the sequence $V$ into contiguous sub-sequences $L$, $M$, and $R$, such that for each $x \in L$, $y \in M$ and $z \in R$, we have $x \leq y \leq z$. Moreover, the number of vectors in $L$ equals that of $R$, namely, $|L| = |R| = O(n/\log n)$. Assuming that $\max L \neq \min R$ (otherwise sorting is trivial), we consider the pairs

$$P = \{\langle L[i], R[i]\rangle \mid 1 \leq i \leq |L|\}.$$

Note that for every pair $\langle x, y \rangle \in P$, vectors $x$ and $y$ are distinct ($x < y$) and their first mismatching scalar component is at position $lcp(x, y) + 1$. Based on this observation we identify a subset $H$ of the pairs in $P$,

- $|H| = \Omega\left(\frac{n}{\log^2 n}\right)$, and

- there exists an interval $[l, r] \subseteq [1, k]$ of size $\max \left\{1, \frac{k}{\log n}\right\}$, such that $lcp(x, y) + 1 \in [l, r]$ for every pair $\langle x, y \rangle \in H$.

In this way we can use the vectors in $H$ for implicitly representing $O(n/\log^2 n)$ bits, called heavy bits, so that

- decoding one heavy bit requires $O(1 + k/\log n)$ time,

- encoding it takes $O(k)$ time.

Let us see why do we need these bits. When designing an optimal in-place algorithm, the constraint on using just $O(1)$ auxiliary locations, namely, $O(\log n)$ extra bits of information, is rather stringent. Fortunately, permutations of the elements encode themselves further bits of informations. Potentially, we can have $\log h!$ bits by permuting $h$ distinct elements. As we seen in Chapter 4, bit stealing [Munro, 1986] is a basic technique for implicitly encoding up to $h$ bits of information by pairwise permuting $h$ pairs of elements. In its original design, the technique encodes a bit
with each distinct pair of elements \( x \) and \( y \), such that \( x < y \). The bit is of value 0 if \( x \) occurs before \( y \) in the permutation; it’s of value 1 if \( x \) occurs after \( y \) in the permutation. The main drawback of this technique in our setting is that we need \( O(k) \) time for encoding and decoding one bit since \( x \) and \( y \) are vectors. As we shall see, we will require an amortized number of \( O(1) \) encoded bits and \( O(\log n) \) decoded bits per vector, so that we have to decrease the cost of decoding to \( O(1 + k/\log n) \) to stay within the claimed bounds.

At this stage, sorting \( \mathcal{V} \) reduces to sorting \( \mathcal{M} \) as an instance of \( GVSP\{n - o(n), 0, O(n / \log^2 n)\} \). After that, it also reduces to sorting \( \mathcal{L} \) and \( \mathcal{R} \) as instances of \( GVSP\{O(n / \log n), 0, 0\} \), easily solvable by Lemma 8.1.

### 8.3.2 Buffering and session sorting

We solve an instance of

\[
GVSP\{n - o(n), 0, O(n / \log^2 n)\}
\]

(i.e., sorting \( \mathcal{M} \)) by reduction to

- \( O(\log n) \) instances of \( GVSP\{O(n / \log n), O(n / \log n), O(n / \log^2 n)\} \),
- plus a constant number of instances of \( GVSP\{O(n / \log n), 0, 0\} \) (solvable by Lemma 8.1).

We logically divide \( \mathcal{M} \) into contiguous sub-sequences \( \mathcal{M}_1, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s \), where

\[
|\mathcal{M}_2| = \cdots = |\mathcal{M}_{s-1}| \leq |\mathcal{M}_s| = O\left(\frac{n}{\log n}\right)
\]

and \( s = O(\log n) \). Moreover, \( |\mathcal{M}_1| = O(n / \log n) \) has a sufficiently large multiplicative constant, so that \( \mathcal{M}_1 \) can host enough vectors playing the role of placeholders.
With reference to Figure 8.1, we sort the \( m = O(n/\log n) \) vectors in each individual \( \mathcal{M}_i, \ i \neq 1 \), using the \( p = O(n/\log n) \) placeholders in \( \mathcal{M}_1 \) and the \( h = O(n/\log^2 n) \) heavy bits encoded by the pairs in \( H \subseteq \mathcal{L} \times \mathcal{R} \).

Having just \( n \) vector locations, we cannot rely on a temporary area of vector locations for efficiently permuting the vectors with a few moves. We therefore exploit the virtual form of temporary area using the internal buffering technique of Kronrod [Kronrod, 1969] (see Chapter 4). We designate the vectors in \( \mathcal{M}_1 \) as placeholders for "free memory" since we do not care to sort them at this stage. Hence, they can be scrambled up without interplaying with the sorting process that is running on a given \( \mathcal{M}_i, \ i \neq 1 \). When we need to move a vector of \( \mathcal{M}_i \) to the temporary area, we simulate this fact by exchanging the vector with a suitable placeholder of \( \mathcal{M}_1 \). At the same time, we should guarantee that this exchange is somehow reversible, allowing us to put the placeholders back to the "free memory" in \( \mathcal{M}_1 \) without screwing up the sorting obtained for \( \mathcal{M}_i, \ i \neq 1 \).

Assuming to have obtained each of \( \mathcal{M}_2, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s \) in lexicographic order, we still have to merge them using the heavy bits in \( H \) and the placeholders in \( \mathcal{M}_1 \). It turns out that this task is non-trivial to be performed. Just to have a rough idea, let's imagine to run the 2-way in-place mergesort for \( O(\log s) = O(\log \log n) \) passes on them. This would definitely give a non-optimal time cost for the vectors since the number of vector exchanges would be \( \omega(n) \), loosing optimality in this way. We introduce a useful variant of the technique in [Franceschini, 2004] (see Chapter 7) that is a generalization of the technique seen in [Kronrod, 1969], thus obtaining what we call session sorting. Let us assume that the vectors are distinct (we shall disregard this assumption in Section 8.5).

The main goal of session sorting is that of rearranging all the vectors in \( \mathcal{M}_2, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s \), so that they are not too far from their final destination. If any such vector has rank \( r \) among all the other vectors in \( \mathcal{M}_2, \mathcal{M}_3, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s \), and occupies a position \( g > r \) after session sorting, we guarantee that \( g - r \leq |\mathcal{M}_i| \). (Note that we do not claim anything regarding the case \( g \leq r \).) Using this strong property, we show that the sequence of 2-way in-place operations for merging \( \mathcal{M}_i \) and \( \mathcal{M}_{i+1} \) for \( i = 2, 3, \ldots, s - 1 \) (in this order) yields the sorted sequence. (We remark that this is not generally true if we do not apply session sorting.) As a result, the entire sequence \( \mathcal{M}_2, \mathcal{M}_3, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s \) is in lexicographic order with a linear number of moves.

What remains to do is sorting \( \mathcal{L}, \mathcal{M}_1, \) and \( \mathcal{R} \) individually as instances of \( GVSP\{O(n/\log n), 0, 0\} \) by Lemma 8.1. Merging them in-place with the rest of sorted vectors is a standard task giving \( \mathcal{V} \) in sorted order. Hence, we have to show how to solve an instance of

\[
GVSP\{O(n/\log n), O(n/\log n), O(n/\log^2 n)\},
\]

which corresponds to sorting a given \( \mathcal{M}_i, \ i \neq 1 \), using the placeholders initially hosted in \( \mathcal{M}_1 \) and the heavy bits encoded by the pairs in \( H \).
8.3.3 Sorting each \( M_i \) individually

We describe this stage in general terms. For a given \( i \neq 1 \), let \( M' = M_i \) and \( M_B = M_1 \), for the instance of \( GVSP \{ |M'|, |M_B|, |H| \} \) that we are going to solve with the heavy bits in \( H \) (see Figure 8.1). Using \( M_B \) as a “free memory” area, we simulate the sorting of the \( m' = |M'| \) vectors by inserting them into a suitable structure that is incrementally built inside \( M_B \). Each insertion of a vector \( x \in M' \) into the internal structure of \( M_B \) exchanges \( x \) with a placeholder. After each such exchange we permute some of the vectors inside \( M_B \), so as to dynamically maintain a set of \( O(m' / \log^2 m') \) pivot vectors in the internal structure. The pivots have buckets associated inside \( M_B \) for the purpose of distributing the non-pivot vectors inserted up to that point, like in distribution sort. Each bucket contains \( \Theta(\log^2 m') \) vectors that are kept unsorted to minimize the number of vector exchanges needed to maintain the internal structure of \( M_B \).

The pivots inside \( M_B \) are kept searchable by a suitable blend of the techniques in [Franceschini and Grossi, 2004b; Itai, Konheim, and Rodeh, 1981; Manber and Myers, 1993] (see Chapters 4 and 10), requiring to decode \( O(\log n) \) heavy bits per inserted vector (which is fine since decoding takes \( O(1 + k / \log n) \) time). In particular, we logically divide each vector \( x \) into a concatenation of \( O(\log m') = O(\log n) \) equally sized chunks. We only store the \( lcp \) information for the chunks considered as “meta-characters,” thus obtaining an approximation of the \( lcp \) information for the vectors. After that the distribution completes by inserting all the vectors of \( M' \) into the internal structure of \( M_B \), we sort the buckets individually by using a constant number of recursive iterations of session sorting whose parameters are suitably adapted to the buckets’ size. The base case of the recursion consists in solving

\[
GVSP \{ O(\sqrt{\log m'}), O(\sqrt{\log m'}), 0 \},
\]

for which we design an optimal ad-hoc algorithm. After completing the individual sorting of the buckets, which still reside in \( M_B \), we exchange them with the placeholders that were temporarily moved to \( M' \). We place back the sorted buckets and their pivots to \( M' \) according to their relative order, which means that the \( m' \) vectors in \( M' \) are in lexicographic order.

8.3.4 Known tools.

We use few optimal algorithmic tools for atomic elements: in-place stable mergesort and in-place merge [Salowe and Steiger, 1987]; in-place selection for order statistics [Lai and Wood, 1988]. We apply these algorithms to vectors in a straightforward way by paying a slowdown of \( O(k) \) per elementary step in their time complexity. We also use Hirschberg’s linear scanning method [Hirschberg, 1978] for searching a set of \( n \) vectors of length \( k \), with the simple bound of \( O(k + n) \) time (see Chapter 4). We go through the convention that the last lowercase letters—\( \ldots, x, y, w, z \)—represent
vectors and the middle ones—..., i, j, k, l...—are auxiliary indices or parameters. We omit the ceilings and the floors for the non-integer values.

8.4 Heavy Bits

We describe how to reduce the problem of sorting in-place $n$ vectors—an instance of $GVSP\{n, 0, 0\}$—to an instance of

$$GVSP\{n - o(n), 0, O(n/\log^2 n)\}$$

plus a constant number of instances of

$$GVSP\{O(n/\log n), 0, 0\}.$$  

(The notation for $GVSP\{m, p, h\}$ is defined in Section 8.3 and illustrated in Figure 8.1.) This stage is characterized by a variant of the bit stealing technique (see [Munro, 1986] or Chapter 4), which is a crucial task when designing an optimal in-place algorithm. We describe how to reduce the decoding cost for a bit to $O(1 + k/\log n)$ time while preserving its encoding cost to $O(k)$. As previously mentioned, we will require an amortized number of $O(1)$ encoded bits and $O(\log n)$ decoded bits per vector for staying within the claimed bounds.

We begin by preprocessing the sequence of vectors in $\mathcal{V}$ that are contained in $\mathcal{V}$, so as to arrange in place the least $p$ vectors in lexicographic order at the beginning of $\mathcal{V}$ and the greatest $p$ vectors at the end of $\mathcal{V}$, for a value of $p = O(n/\log n)$ with a sufficiently large hidden multiplicative constant. For this, we partition in-place the sequence $\mathcal{V}$ into contiguous sub-sequences $\mathcal{L}$, $\mathcal{M}$, and $\mathcal{R}$, such that for each $x \in \mathcal{L}$, $y \in \mathcal{M}$ and $z \in \mathcal{R}$, we have $x \leq y \leq z$. Moreover, the number of vectors in $\mathcal{L}$ and $\mathcal{R}$ equals $p$, namely,

$$|\mathcal{L}| = |\mathcal{R}| = p = O\left(\frac{n}{\log n}\right).$$

We obtain the above partition by performing order statistics in-place [Lai and Wood, 1988] so as to identify the $p$th and $(n - p + 1)$st elements of $\mathcal{V}$ in $O(nk)$ time. Let $w_L$ and $w_R$ be these elements, respectively. We then scan $\mathcal{V}$ and partition its vectors into the three unsorted sub-sequences $\mathcal{L}$, $\mathcal{M}$ and $\mathcal{R}$. We first put vectors $x < w_L$ into $\mathcal{L}$ and vectors $z > w_R$ into $\mathcal{R}$ by exchanges. If there are less than $p$ vectors in $\mathcal{L}$, there should be multiple occurrences of $w_L$. An analogous observation holds for $\mathcal{R}$ and $w_R$. With a further scan, we can exchange a sufficiently large number of occurrences of $w_L$ and/or $w_R$, so that $\mathcal{L}$ and $\mathcal{R}$ contain $p$ vectors each. The remaining vectors constitute $\mathcal{M}$. In the rest of the chapter we assume that $w_L \neq w_R$; otherwise, $\mathcal{M}$ is made up of all equal vectors and sorting is trivially solved by applying Lemma 8.1 to $\mathcal{L}$ and $\mathcal{R}$.

Let’s consider the set of pairs of vectors thus obtained,
\[ P = \{ (\mathcal{L}[i], \mathcal{R}[i]) : 1 \leq i \leq p \} \subseteq \mathcal{L} \times \mathcal{R}. \]

Let us conceptually divide each of these vectors into chunks of \( k/\ell = O(1 + k/\log n) \) scalar components, where

\[ \ell = \min\{k, \log n\}. \]

We index these chunks from 1 to \( \ell \), in the order of their appearance inside the vector. We assign an integer label \( j \) to each pair \( (\mathcal{L}[i], \mathcal{R}[i]) \), where \( 1 \leq j \leq \ell \) and \( 1 \leq i \leq p \). Since \( \mathcal{L}[i] < \mathcal{R}[i] \) by construction, we defined label \( j \) to be the index of the chunk containing the first mismatching position for \( \mathcal{L}[i] \) and \( \mathcal{R}[i] \); that is, it satisfies

\[ (j - 1) \frac{k}{\ell} \leq \text{lcp}(\mathcal{L}[i], \mathcal{R}[i]) < j \frac{k}{\ell}. \]

By the pigeon principle, there must exist a value of \( j \) for which at least \( p/\ell = \Omega(n/\log^2 n) \) pairs in \( P \) are labelled \( j \). We can identify that value by running at most \( \ell \) in-place scans of \( \mathcal{L} \) and \( \mathcal{R} \), with an overall cost of \( O(\ell \times pk) = O(nk) \) time. With a further scan of \( \mathcal{L} \) and \( \mathcal{R} \), we single out \( h = \Theta\left( \frac{p}{\ell} \right) = \Theta\left( \frac{n}{\log^2 n} \right) \) pairs in \( P \) that have label \( j \), moving them in-place at the beginning of \( \mathcal{L} \) and \( \mathcal{R} \), respectively. Consequently, we identify these vectors in the first \( h \) locations in \( \mathcal{L} \) and \( \mathcal{R} \) by a set of pairs, denoted by \( H \):

- \( H \subseteq P \) and \( |H| = h = \Theta\left( \frac{n}{\log^2 n} \right) \);
- \( H = \{ (\mathcal{L}[i], \mathcal{R}[i]) : 1 \leq i \leq h \} \) after the preprocessing;
- there exists \( j \in [1, \ell] \) such that \( (j - 1) \frac{k}{\ell} \leq \text{lcp}(x, y) < j \frac{k}{\ell} \) for every pair \( (x, y) \in H \).

As a result of the above process we are now able to steal bits in \( H \) using the knowledge of \( j \) as follows.

**Encoding.** For \( 1 \leq i \leq h \), we encode the \( i \)th bit of value 1 by exchanging \( \mathcal{L}[i] \) and \( \mathcal{R}[i] \) in \( O(k) \) time; namely, \( \mathcal{L}[i] \) occupies now position \( i \) inside \( \mathcal{R} \) and \( \mathcal{R}[i] \) does it inside \( \mathcal{L} \). If the bit is 0, we leave them at their position (no exchange).
8.5. BUFFERING AND SESSION SORTING

Decoding. In order to decode the $i$th bit, we only compare their $j$th chunk to find their mismatching position in the interval of components $[(j-1)k/\ell+1, jk/\ell]$. In this way, we can establish whether or not the two vectors have been exchanged during encoding (and so we decode either 0 or 1). Decoding performs at most $k/\ell$ component comparisons and thus takes $O(1 + k/\log n)$ time.

The non-constant cost of bit stealing motivates our choice of referring to these bits as heavy.

Lemma 8.2 We can encode $h = \Theta(n/\log^2 n)$ heavy bits by the pairwise permutation of vectors in $H \subseteq \mathcal{L} \times \mathcal{R}$. Encoding one bit requires $O(k)$ time while decoding it requires $O(1+k/\log n)$ time. Preprocessing requires $O(nk)$ time using $O(1)$ auxiliary locations.

We keep $\mathcal{L}$ and $\mathcal{R}$ unsorted for encoding bits until the end of the algorithm. At that point, we can in-place sort $\mathcal{L}$ and $\mathcal{R}$ by Lemma 8.1, in $O(nk)$ time. Consequently we are left with the problem of sorting $\mathcal{M}$.

Lemma 8.3 There exists an $O(nk)$-time reduction from GVSP\{\(n, 0, 0\)\} to GVSP\{\(n-o(n), 0, O(n/\log^2 n)\)\}, using $O(1)$ auxiliary locations.

8.5 Buffering and Session Sorting

In this section, we detail how to sort the vectors in $\mathcal{M}$, which is an instance of GVSP\{\(n-o(n), 0, O(n/\log^2 n)\)\}. We reduce that instance to

- a number of $O(\log n)$ instances of GVSP\{\(O(n/\log n), O(n/\log n), O(n/\log^2 n)\)\},
- plus a constant number of instances of GVSP\{\(O(n/\log n), 0, 0\)\} that are easy to solve by Lemma 8.1.

We logically divide $\mathcal{M}$ into contiguous sub-sequences $\mathcal{M}_1, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s$, called blocks, where

$$|\mathcal{M}_2| = \cdots = |\mathcal{M}_{s-1}| \leq |\mathcal{M}_s| = O(n/\log n)$$

$$s = O(\log n)$$

In the following, we assume without loss of generality that $|\mathcal{M}_s| = |\mathcal{M}_{s-1}|$ (if not, we treat $\mathcal{M}_s$ differently, applying Lemma 8.1 to it). We remark that only a constant number of blocks can be sorted with the bounds of Lemma 8.1. Hence we should proceed otherwise. As previously mentioned, having just $n$ vector locations, we cannot rely on a temporary area of vector locations for efficiently permuting the vectors with a few moves. Hence, we designate the $O(n/\log n)$ vectors in $\mathcal{M}_1$, for
a sufficiently large multiplicative constant, to act as placeholders in the internal buffering technique (see [Kromrod, 1969] and Chapter 4).

We have thereby reduced our task to the problem of sorting each $\mathcal{M}_i$, $i \neq 1$, using $\mathcal{M}_1$ as “temporary area” and the $h = O(n / \log^2 n)$ heavy bits available from the pairs of vectors in $H \subseteq \mathcal{L} \times \mathcal{R}$. In this way we obtain $O(\log n)$ instances of $GVSP\{O(n / \log n), O(n / \log n), O(n / \log^2 n)\}$, as claimed above.

However, we are still missing a crucial part of the reduction performed at this stage, namely, how to obtain all the vectors in $\mathcal{M}_2, \mathcal{M}_3, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s$ in lexicographic order. Although each $\mathcal{M}_i$ can be individually arranged in that order by the reduction, the mere juxtaposition of the $\mathcal{M}_i$s, $i \neq 1$, is not necessarily in lexicographic order. We cannot apply in-place merging to the blocks as this would cause $\omega(n)$ vector exchanges, loosing optimality. We present a combinatorial approach to circumvent this difficulty.

We begin with a naive scheme that uses the 2-way in-place merging, denoted \textsc{in-place-merge}, and that, clearly, does not yield a linear number of moves (not even an optimal number of comparisons). The sequence of operations is

\[
\begin{align*}
\text{IN-PLACE-MERGE}(\mathcal{M}_2, \mathcal{M}_3) \\
\text{IN-PLACE-MERGE}(\mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4) \\
\ldots \\
\text{IN-PLACE-MERGE}(\mathcal{M}_2, \mathcal{M}_3, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s)
\end{align*}
\]

We will show that the naive scheme can be simulated by an alternative scheme performing a preprocessing of the vectors in $\mathcal{M}_2, \mathcal{M}_3, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s$. It obtains a right-bounded permutation, since it rearranges the vectors so that each vector cannot occupy a position beyond a bounded distance to the right of its final position in the sorted sequence.

As we will prove, the net effect of the right-bounded permutation is that we can simulate the inefficient scheme above by the following scheme with a linear number of exchanges:

\[
\begin{align*}
\text{IN-PLACE-MERGE}(\mathcal{M}_2, \mathcal{M}_3) \\
\text{IN-PLACE-MERGE}(\mathcal{M}_3, \mathcal{M}_4) \\
\ldots \\
\text{IN-PLACE-MERGE}(\mathcal{M}_{s-1}, \mathcal{M}_s)
\end{align*}
\]

We describe this permutation in general terms as it is of independent interest. A variant of this technique was presented in [Franceschini, 2004] (see Chapter 7).

### 8.5.1 Right-bounded permutations

We are given three positive integers $m, p, q$, such that $q$ divides $p$ and $p$ divides $m$, satisfying

\[
\left(\frac{m}{p} - 1\right) \times (q - 1) \leq p \quad (8.1)
\]
Given a sequence $\mathcal{B}$ of $m$ vectors, we logically divide it into $m/q$ sub-blocks of $q$ vectors each, denoted by

$$\mathcal{I}_1, \ldots, \mathcal{I}_{m/q}.$$ 

The sub-blocks are grouped into blocks of $p/q$ sub-blocks each, thus logically dividing $\mathcal{B}$ into $m/p$ blocks of $p$ vectors each, denoted by

$$\mathcal{B}_1, \ldots, \mathcal{B}_{m/p}.$$ 

A right-bounded permutation is the arrangement of the vectors in $\mathcal{B}$ resulting from steps P1–P2, with steps P3–P4 yielding the sequence in lexicographic order:

P1. For $j = 1, \ldots, m/p$, sort each block $\mathcal{B}_j$ individually.

P2. Sort stably the $m/q$ sub-blocks $\mathcal{I}_1, \ldots, \mathcal{I}_{m/q}$ according to their first vector (i.e., comparisons are driven by the minimum vector in each sub-block, and the rest of the vectors are considered as “satellite data”).

P3. For $j = 1, \ldots, m/p$, sort each block $\mathcal{B}_j$ individually (note that the content of the blocks changed!).

P4. For $j = 1, \ldots, m/p - 1$, merge the vectors contained in blocks $\mathcal{B}_j$ and $\mathcal{B}_{j+1}$.

**Lemma 8.4** For each vector $\mathcal{B}[i]$, $1 \leq i \leq m$, let $g_i$ be the number of vectors $\mathcal{B}[j] > \mathcal{B}[i]$ such that $1 \leq j < i$ right after steps P1–P2. Then

$$g_i \leq \left( \frac{m}{p} - 1 \right) \times (q - 1). \quad (8.2)$$

**Proof:** Let’s consider the arrangement of the vectors in $\mathcal{B}$ right after steps P1–P2. In order to prove equation (8.2), we need to consider the intermediate arrangement of the vectors in $\mathcal{B}$ after step P1 and before step P2. Recall that we logically divide $\mathcal{B}$ into blocks and sub-blocks, indexing the blocks from 1 to $m/p$.

We assign a unique type to each block based on its index, namely, block $\mathcal{B}_i$ is assigned type $t$, where $1 \leq t \leq m/p$, since it is the $t$th block in $\mathcal{B}$. For the intermediate arrangement above, we say that a vector has type $t$ if it belongs to $\mathcal{B}_i$ (recall that $\mathcal{B}_i$ is sorted). We can assign type $t$ to the sub-blocks of each $\mathcal{B}_i$ in the same manner, since each sub-block contains vectors of the same type $t$ by construction. Hence the type of a sub-block is well defined. We refer to the first vector of each sub-block, which is also the minimum in it, as the header of the sub-block.

Let us now resume the arrangement of the vectors in $\mathcal{B}$ right after steps P1–P2. Consider a generic vector $\mathcal{B}[i]$ belonging to a sub-block, say $\mathcal{I}'$ of type $t'$, and let $g_i$ be defined as above. We give an upper bound to $g_i$ so as equation (8.2) holds. Specifically, we count the maximum number of vectors contributing to $g_i$. Let’s
discuss them by their type. By the stability of the sorting process in step P2, we
know that the vectors of type \( t' \) have maintained the relative order they had in the
intermediate arrangement (after step P1 and before step P2) and so they cannot
contribute to \( g_i \).

Let \( x \) be the header of the sub-block \( \mathcal{I}' \) containing \( \mathcal{B}[i] \). Let's evaluate the
contribution to \( g_i \) for the vectors of type \( t'' \neq t' \). Consider all sub-blocks of type \( t'' \):
we claim that at most one of them, say \( \mathcal{I}'' \), can contain vectors contributing to \( g_i \).
Precisely, \( \mathcal{I}'' \) is the sub-block of type \( t'' \) having the largest header less than or equal
to \( x \). Let \( y \leq x \) be the header of \( \mathcal{I}'' \) and \( z \) be one of such contributing vectors in
\( \mathcal{I}'' \). Sub-block \( \mathcal{I}'' \) is laid out before \( \mathcal{I}' \) by construction but \( z > \mathcal{B}[i] \) by definition
of \( g_i \). Note that there can be at most \( q - 1 \) such vectors \( z \) in \( \mathcal{I}'' \).

For any other sub-block of type \( t'' \), we show that its vectors cannot contribute
to \( g_i \). Since the block of type \( t'' \) is sorted after step P1, there are two possibilities
for its sub-blocks \( \mathcal{I}'' \neq \mathcal{I}'' \): (a) \( \mathcal{I}'' \) contains all vectors that are less than or
equal to \( y \leq x \) (i.e., \( \mathcal{I}'' \) is laid out before \( \mathcal{I}'' \)); they do not contribute to \( g_i \) by
transitivity since \( x \leq \mathcal{B}[i] \). (b) \( \mathcal{I}'' \) contains all vectors that are greater than or
equal to \( z > \mathcal{B}[i] \geq x \) (i.e., \( \mathcal{I}'' \) is laid out after \( \mathcal{I}'' \)); they do not contribute because
the header of \( \mathcal{I}'' \) is strictly larger than \( x \) by transitivity and so \( \mathcal{I}'' \) is laid out after
\( \mathcal{I}' \). Summing up, the total contribution to \( g_i \) for the vectors of type \( t'' \neq t' \) is at
most \( q - 1 \) (a subset of the vectors in \( \mathcal{I}'' \)). Since there are \( \frac{m}{p} - 1 \) different types
other than \( t' \), we obtain the upper bound for equation (8.2). \( \square \)

**Theorem 8.1** After steps P1-P4, the sequence \( \mathcal{B} \) is sorted.

**Proof**: We proceed by induction on the length of prefixes of blocks in \( \mathcal{B} \). The base
case is obvious, as we know that \( \mathcal{B}_1 \) is sorted by step P3. Let us assume that the \( j \)
prefix of blocks \( \mathcal{B}_1 \mathcal{B}_2 \cdots \mathcal{B}_j \) is sorted by induction, for \( j \geq 1 \). After step P3, the
upper bound in equation (8.2) still holds for any vector \( v \) in block \( \mathcal{B}_{j+1} \) (modulo the
inner permutation due to the sorting of \( \mathcal{B}_{j+1} \)). Indeed, the number of vectors \( z > v \)
that are laid out before \( v \) cannot increase; those inside \( \mathcal{B}_{j+1} \) disappear after sorting
it and so the upper bound in equation (8.2) is anyway valid. By equation (8.1), we
derive that \( p \), the size of each block, is larger than the upper bound of equation (8.2).
As a result, the number of vectors \( z > v \) that belong to the \( j \)th prefix of blocks cannot
exceed \( p \). Hence, they should be contained in the last locations of block \( \mathcal{B}_j \) since
\( p = |\mathcal{B}_j| \) and \( \mathcal{B}_1 \mathcal{B}_2 \cdots \mathcal{B}_j \) is sorted by induction. This allows us to conclude that
after merging \( \mathcal{B}_j \) and \( \mathcal{B}_{j+1} \), the \((j + 1)\)st prefix of blocks \( \mathcal{B}_1 \mathcal{B}_2 \cdots \mathcal{B}_{j+1} \) is sorted,
thus proving the statement of the theorem. \( \square \)

### 8.5.2 Session sorting

We apply the steps stated in Theorem 8.1 to sorting the vectors in \( \mathcal{M} \) into sessions.
Hence, we choose \( \mathcal{M}_1 \) of size \( O(n/\log n) \) for the placeholders. We then fix
\[ q = \log^3 n \]

\[ p = q \frac{n}{\log n} = \Theta \left( \frac{n}{\log n} \right) \]

and we pick \( m \) as the largest multiple of \( p \) such that \( m \leq |\mathcal{M}| - |\mathcal{M}_1| \). These values satisfy equation (8.1). We therefore obtain the logical division of \( \mathcal{M} \) into blocks \( \mathcal{M}_1, \ldots, \mathcal{M}_{s-1}, \mathcal{M}_s \), as expected.

**Lemma 8.5** There exists an \( O(nk) \)-time reduction from \( GVSP \{n-o(n), 0, O(n/\log^2 n)\} \) to a number of \( O(n) \) instances of \( GVSP \{O(n/\log n), O(n/\log n), O(n/\log^2 n)\} \), using \( O(1) \) auxiliary locations.

**Proof:** We have to prove how to apply steps P1–P4 to \( \mathcal{M}_2, \ldots, \mathcal{M}_s \) (assuming w.l.o.g. that \( |\mathcal{M}_s| = |\mathcal{M}_{s-1}| \)).

In steps P1 and P3, we solve \( m/p = O(\log n) \) instances of

\[ GVSP \{O(n/\log n), O(n/\log n), O(n/\log^2 n)\}, \]

as claimed at the beginning of this section. We give some more details for them in Section 8.6.

In step P2, we have just \( m/q = O(n/\log^3 n) \) vectors to sort, which are the minimum in each sub-block. We refer to them as headers and to the rest of the vectors as satellite data (with \( q-1 \) vectors each). We associate a unique implicit index in the range from 1 to \( m/q \) with the satellite data in each sub-block. We employ the heavy bits in \( H \) so as to form a sequence of \( m/q \) integers \( h_1, h_2, \ldots, h_{m/q} \) of \( \log n \) bits each, employed to encode a permutation of these indexes. Note that we have direct access to any \( h_j, 1 \leq j \leq m/q \), in \( O(k \log n) \) time for encoding it and \( O(k + \log n) \) time for decoding it by Lemma 8.2.

At the beginning of step P2, we set \( h_j = j \) and exchange the \( j \)th placeholder in \( \mathcal{M}_1 \), for \( 1 \leq j \leq m/q \). We then apply the in-place stable mergesort on the headers thus collected in \( \mathcal{M}_1 \). Each comparison cost is \( O(k) \) time while each exchange requires \( O(k \log n) \) time. Indeed, when exchanging two headers inside \( \mathcal{M}_1 \), say at position \( j' \) and \( j'' \), we have also to swap the values of \( h_{j'} \) and \( h_{j''} \), involving their decoding and encoding in \( H \). Note that the satellite data is not exchanged but \( h_{j'} \) and \( h_{j''} \) are correctly updated to maintain the association of the headers with their satellite data in the sub-blocks. At the end of the mergesort, we exchange the \( j \)th header in \( \mathcal{M}_1 \) with the placeholder temporarily hosted in the \( h_j \)th sub-block. The total cost is \( O((m/q) \log (m/q) \times (k \log n)) = O((n/\log^3 n) \log n \times (k \log n)) = o(nk) \).

We now have to permute the sub-blocks according to the values of \( h_1, \ldots, h_{m/q} \) encoded in \( H \). Specifically, the \( h_j \)th sub-block must occupy the \( j \)th position among the sub-blocks to reflect the stable sorting of their headers. We employ an additional sequence of integers \( r_1, r_2, \ldots, r_{m/q} \) encoded in \( H \), initializing \( r_i = j \) if and only if
\( h_j = i \). We proceed incrementally for \( j = 1, 2, \ldots, m/q - 1 \) (in this order), preserving the invariant that we have correctly placed the first \( j-1 \) sub-blocks, with \( h_1, \ldots, h_{m/q} \) and \( r_1, r_2, \ldots, r_{m/q} \) suitably updated to reflect the fact that one permutation is the inverse of the other (in particular, \( h_j = r_j = j' \) for \( 1 \leq j' < j \), so the invariant is meaningful for the rest of the indexes). Note that some of the sub-blocks may have exchanged in order to place the first \( j-1 \) sub-blocks. Hence, when we refer to the \( j \)th and the \( h_j \)th sub-blocks, they are taken from the current arrangement of sub-blocks. If \( j = h_j \), the current sub-block is already correctly placed and the invariant is trivially preserved. Otherwise, we exchange the \( j \)th and the \( h_j \)th sub-blocks by pairwise exchanging their \( i \)th vectors for \( i = 1, 2, \ldots, q \). In order to preserve the invariant, we simultaneously swap the values of \( h_j \) and \( r_{h_j} \) and the values of \( r_j \) and \( r_{h_j} \), respectively, re-encoding them in \( H \). Since the exchange of sub-blocks requires the pairwise exchange of \( q \) vectors plus the encoding and decoding of \( O(1) \) values among \( h_1, \ldots, h_{m/q} \) and \( r_1, r_2, \ldots, r_{m/q} \), the cost is \( O(qk + k \log n) \). When \( j = m/q - 1 \), the last two sub-blocks are placed correctly and we have performed a total of \( O(m/q) \) such exchanges. The final cost is \( O(m/q \times (qk + k \log n)) = O(n/\log^3 n \times k \log^3 n) = O(nk) \). Hence, the total cost of step P2 is \( O(nk) \).

Finally, in step P4, we use the in-place merging with comparison cost \( O(k) \). As a result, we obtain a total cost of \( O(m/p \times pk) = O(nk) \) for step P4 (and \( \mathcal{M} \) is sorted).

\[ \square \]

### 8.6 Sorting Each Block Individually

We have to solve an instance of \( \text{GVSP} \{O(n/\log n), O(n/\log n), O(n/\log^2 n)\} \) (see Figure 8.1). We reformulate it equivalently as \( \text{GVSP} \{m', O(m'), O(m'/\log m')\} \), where \( m' = O(n/\log n) \) vectors in \( \mathcal{M}' \) should be sorted using a sufficiently large number \( O(m') \) of placeholders in \( \mathcal{M}_B \). We will need to encode \( O(1) \) sequences of integers of \( O(\log m') = O(\log n) \) heavy bits each in \( H \subseteq \mathcal{L} \times \mathcal{R} \), totaling \( O(m'/\log m') \) heavy bits.

We sort \( \mathcal{M}' \) by repeatedly inserting its vectors into an internal structure maintained inside \( \mathcal{M}_B \) to mimic a distribution sort. The sorting process examines one vector at a time, routing it to a suitable bucket inside \( \mathcal{M}_B \). Each insertion exchanges the current vector in \( \mathcal{M}' \) with a suitable placeholder in the bucket. After the process terminates, we sort the individual buckets in \( \mathcal{M}_B \). Having collected all the necessary information for producing the sorted sequence of buckets, we exchange them back to \( \mathcal{M}' \) in sorted order.

To begin with, we describe the internal organization of \( \mathcal{M}_B \), which we logically divide into two parts described next, the distribution area for the pivots and the bucket area. The size of both areas can be computed ahead of time.
8.6.1 Pivots and buckets in $\mathcal{M}_B$

The internal structure of $\mathcal{M}_B$ consists of $O(m'/\log^2 m')$ pivots and their associated buckets. We store pivots in the distribution area of $\mathcal{M}_B$ while we reserve a different area in $\mathcal{M}_B$ for the buckets. The association among pivots and buckets is that typical of distribution sorting. Each bucket is associated with a unique pivot, which is smaller than or equal to any other vector in the bucket. For any two consecutive pivots, say $x$ and $y$ with $x \leq y$, we have $x \leq x' \leq y$ for any $x'$ belonging to the bucket of $x$. However, the sequential layout of pivots does not necessarily reflect the sequential layout of the buckets. For this reason, we maintain the pivot-bucket association through a pointer encoded with $O(\log m')$ heavy bits in $H$.

The distribution area in $\mathcal{M}_B$ is logically divided into segments of $\log m'$ vector locations each, initially holding the placeholders. If a segment accommodates only placeholders, it should be considered as “empty.” Otherwise, if the segment contains at least one vector originated from $\mathcal{M}'$ (after an exchange), we say that it is allocated. We refer to these vectors in the allocated segments as pivots, since their purpose is routing the other vectors of $\mathcal{M}'$ towards their destination buckets in $\mathcal{M}_B$. At any given time, all the pivots in a segment are in sorted order and occupy the leftmost locations in it while the rest of the locations contain placeholders. To distinguish among them, it suffices to encode the number of pivots in the segment with $O(\log \log m')$ heavy bits in $H$. It is worth noting that the amortized number of heavy bit flips to maintain this number encoded in $H$ is a constant (see the binary counter in [Cormen, Leiserson, Rivest, and Stein, 2001]). In general, if $f$ is the number of allocated segments, we reserve a sufficiently large number of consecutive pairs in $H$ to encode these $O(f \log \log m')$ heavy bits. Also, the allocated segments occupy the first $f \times \log m'$ vector locations of the distribution area in $\mathcal{M}_B$. Scanning them in left-to-right order yields the sorted sequence of pivots herein contained.

The bucket area is logically divided into buckets of $2 \log^2 m'$ vector locations each, initially holding the placeholders. Analogously to segments, we define empty and allocated buckets (but we only use the term of pivots for referring to the vectors in the segments); the vectors occupy the leftmost positions of each bucket, with their number encoded in $H$ (with a constant amortized number of heavy bit flips). We also say that a bucket is full when only vectors from $\mathcal{M}'$ are contained in its locations. We preserve the invariant that the minimum number of vectors in a bucket is $\log^2 m'$ (while the maximum is clearly $2 \log^2 m'$). However, the vectors inside buckets are kept unsorted because it would cost too many vector exchanges to maintain them sorted. If $f$ is the number of allocated buckets, they occupy the last $f \times 2\log^2 m'$ vector locations of the bucket area in $\mathcal{M}_B$. Note the sequence of buckets in the area is unsorted, so that we need to maintain their association with the sorted pivots in the distribution area, as previously mentioned.
8.6.2 Routing among the pivots

When routing a vector among the buckets, we need to search its predecessor among the pivots, which are in sorted order. A binary search would give a slowdown factor of \( O(k) \), and so we adapt some of the techniques in [Franceschini and Grossi, 2004b, Manber and Myers, 1993] (see Chapters 10 and 4) for performing an efficient binary search with just \( O(\log n) \) heavy bits decoded. We refer to the minimum pivot in each segment as the \emph{leader} of that segment. Note that we have \( O(m'/\log^2 m') \) leaders and \( O(\log m') \) heavy bits per leaders to encode in \( H \).

If we are able to find the predecessor, \( z \), among the leaders, we can use the linear scanning method in [Hirschberg, 1978] to search among the other pivots in \( z \)'s segment to identify the predecessor and its associated bucket through the pivot-bucket pointer encoded in \( H \).

The ideal approach to find the predecessor of the searching element among the leaders would be that of using the search of Manber and Myers [Manber and Myers, 1993] (see Chapter 4). Originally designed for an array of suffixes, it finely works also for arbitrary elements of unbounded length in lexicographic order. However, this searching algorithm and its variations devised thereafter access too many bits. In [Franceschini and Grossi, 2004b] (see Chapter 10) a variant that is less demanding in the number of bits to be decoded has been proposed.

Let \([L, R]\) be the current interval reached by the searching process, we denote the middle point by \( M \). The algorithm maintains the \emph{lcp} value between the search element \( x \) and the elements \( x_L, x_R \) at positions \( L \) and \( R \) in the sorted sequence, respectively. We know by induction that \( x_L \leq x \leq x_R \). We want to infer the outcome of the comparison of \( x \) versus \( x_M \) (the element in position \( M \)) while preserving the invariant on searching into a smaller interval, either \([L \ldots M]\) or \([M \ldots R]\). Let us suppose that \( \text{lcp}(x_L, x) \geq \text{lcp}(x, x_R) \) (the other case is symmetric), and let \( m = \text{lcp}(x_L, x) \) be the number of initial symbols in \( x \) that have been successfully matched so far.

1. Case \( m < \text{lcp}(x_L, x_M) \). Set \( L = M \).
2. Case \( m = \text{lcp}(x_L, x_M) \). Compute \( \text{lcp}(x, x_M) \) by comparing \( x \) and \( x_M \) from position \( m + 1 \) on. Set \( m = \text{lcp}(x, x_M) \) thus computed. Access symbols in positions \( m + 1 \) (if any) of \( x \) and \( x_M \). If \( x[m + 1] > x_M[m + 1] \), set \( L = M \); else, set \( R = M \).
3. Case \( m > \text{lcp}(x_L, x_M) \). Set \( R = M \).

In order to maintain its invariant, the algorithm requires to maintain the values of \( \text{lcp}(x_L, x) \) and \( \text{lcp}(x, x_R) \) after dealing with each of the three cases. The first two cases are not a problem. In the first case, \( \text{lcp}(x_L, x) = \text{lcp}(x, x_M) \), which is also the current value of \( m \). In the second case, \( \text{lcp}(x_L, x) \) or \( \text{lcp}(x, x_R) \) has been just computed as well since it becomes equal to \( m \). However, in the third case, the algorithm needs \( \text{lcp}(x_L, x_M) \) since it becomes the new value of \( \text{lcp}(x, x_R) \). This
requires reading $O(\log s)$ bits somewhere. Unfortunately, it would give a sub-optimal search in our case.

We make the Manber and Myers’ algorithm more parsimonious by introducing two crucial modifications:

- We encode the required $lcp$ information in unary using a ternary string. For a non-negative integer $\ell \leq s$ (i.e., an $lcp$ value), we define its “unary-ternary” representation as $a_0, a_1, \ldots, a_s$, where

$$a_0 = \cdots = a_{\ell - 1} = 0$$
$$a_\ell = 1$$
$$a_{\ell + 1} = \cdots = a_s = 2$$

Each such digit can be encoded by two bits. The clear advantage is that comparing an integer $g$ to $\ell$, where $0 \leq g \leq s$, just requires decoding an individual digit, $a_g$. The price to pay is that we require now $O(s)$ bits to represent an $lcp$ value, instead of $O(\log s)$ bits.

- We change the invariant maintained by the algorithm on interval $[L \ldots R]$. Since it requires the value of $lcp(x_L, x_M)$ to handle the third case, we have to decode the value of $lcp(x_L, x_M)$ from its “unary-ternary” representation. This is equivalent to identifying the digit $a_{lcp(x_L, x_M)} = 1$ by accessing $\Omega(\log s)$ bits from that representation. Instead, we make a simple but effective observation. We keep only $m$, which is the $lcp$ value between the search element $x$ and the element in $\{x_L, x_R\}$ that maximizes that $lcp$, say $x_L$. In other words, the value of $lcp(x, x_R)$ in the third case above is not actually functional to the search as it is smaller than $m$.

In summary, we only need the “unary-ternary” representation $a_0, a_1, \ldots, a_s$ of $lcp(x_L, x_M)$, without explicitly computing the actual value of $lcp(x_L, x_M)$. We rephrase the three cases of the original search algorithm by examining $a_m$ according to our modifications.

1. Case $a_m = 0$: Inferring that $m < lcp(x_L, x_M)$, set $L = M$ and do not change $m$ (since $x_L$ still maximizes the $lcp$ value).

2. Case $a_m = 1$: Inferring that $m = lcp(x_L, x_M)$, proceed as in the original search algorithm. The new value of $m = lcp(x, x_M)$ indicates that $x_M$ is the best match (which becomes either $x_L$ or $x_R$ in the next search step, as previously mentioned).

3. Case $a_m = 2$: Inferring that $m > lcp(x_L, x_M)$, set $R = M$ and do not change $m$ (since $x_L$ still maximizes the $lcp$ value).
As it should be clear, we do not need the actual value of \( \text{lcp}(x_L, x_M) \), but only the outcome of its comparison with \( m \) (i.e., \( a_m \), which are two bits). Moreover, when \( m \neq \text{lcp}(x_L, x_M) \), the value of \( m \) is unchanged and \( x_L \) still maximizes the \( \text{lcp} \) value. Therefore, we can avoid to decode the entire value of \( \text{lcp}(x_L, x_M) \).

**Lemma 8.6** We can modify Manber and Myers’ algorithm for \( r \) elements of length \( s \) so as to access \( O(1) \) bits of \( \text{lcp} \) values per step during the binary search. We represent each \( \text{lcp} \) value by \( O(s) \) bits to work, for a total of \( O(rs) \) bits storing the \( \text{lcp} \) information. The resulting algorithm requires a total of \( O(\log r) \) bits of the latter kind, and \( O(s + \log r) \) character comparisons.

**Proof:** As previously mentioned, we store the \( \text{lcp} \) information needed by the algorithm in “unary-ternary”, thus requiring no more than \( O(s) \) bits per \( \text{lcp} \) value. Since the number of the \( \text{lcp} \) values is bounded by the number of elements, this yields a total of \( O(rs) \) bits. Our modification of the algorithm accesses just \( O(1) \) of these bits for each step. Since there are \( O(\log r) \) searching steps, this gives a total of \( O(\log r) \) accessed bits. Note that the number of character comparisons remains unchanged, \( O(s + \log r) \), as in the original algorithm. \( \square \)

In order to exploit the modified version of Manber and Myers’ algorithm, we conceptually divide each of the leaders into \( O(\log m') \) equally long chunks (as in Section 8.4), so that we can equivalently consider them as made up of \( O(\log m') \) macro-components each. By Lemma 8.6, we have to decode only \( O(1) \) heavy bits for each step of the modified search algorithm. Therefore the following holds:

**Lemma 8.7** A vector can be routed toward its bucket in \( O(k + \log m') \) time.

### 8.6.3 Maintaining the structure

Given a vector \( x \) in \( \mathcal{M}' \), we can quickly identify its bucket as we showed in Section 8.6.2. After decoding the number of vectors in the bucket to compute the position of the leftmost placeholder in the bucket, we exchange the latter with \( x \) and update that number (the amortized cost is like the binary counter in [Cormen, Leiserson, Rivest, and Stein, 2001]). If the bucket is full, we first find the median in the set given by the \( 2^{\log \frac{1}{2} m'} \) vectors in it plus \( x \). We split the bucket into two, exchange half of the vectors with the placeholders of a new bucket in the bucket area, and associate the latter to the median, which is promoted to become a pivot in the same segment as the previous pivot. If the segment is full, we apply the widely-used basic rebalancing technique introduced in [Itai, Konheim, and Rodeh, 1981] (see Chapter 4) and used, for example, in [Willard, 1982, Bender, Demaine, and Farach-Colton, 2000, Brodal, Fagerberg, and Jacob, 2002] or [Franceschini and Grossi, 2003b, Franceschini and Geffert, 2003, Franceschini, 2005a] (see Chapters 13, 5 and 6). The technique is applied to the distribution area with the necessary information encoded in the heavy bits provided by \( H \). The size of the buckets is sufficient
to amortize the cost of the partial rebuilding for the segments and their information encoded in $H$. Since we have already treated extensively this kind of technique, we will omit the description (see Chapter 4).

Lemma 8.8 The structure can be built in $O(km' + m' \log m')$ time.

8.6.4 Sorting the buckets

At the end of the distribution, we put back each pivot to its associated bucket. Note that the relative order among the buckets can be inferred by the pivot-bucket pointers encoded in the heavy bits (as it occurred with $h_1, \ldots, h_{m'/q}$ in Section 8.5.2). We exchange the buckets back to $\mathcal{M}'$ following this order. After that, recall that we still have to sort each bucket individually. We apply session sorting of Section 8.5 but in a recursive fashion. This means that sorting in steps P1 and P3 is performed by a recursive call to sorting sessions. Note that the recursion depth is a constant since we choose $q = r^{1/4}$, $p = r^{3/4}$, so that equation (8.1) is satisfied, where $r$ is the current number of vectors to sort. Hence, one recursion step scales from $r$ to $r^{3/4}$. After a constant number of recursion steps, we reach the base case for $r = O(\sqrt{\log m'})$, which we can manage directly with an ad-hoc algorithm described below. As a result, we optimally reduce the problem of sorting $O(\log^2 m')$ vectors to that of sorting $O(\sqrt{\log m'})$ vectors, for which we can provide an optimal solution. Hence sorting $O(\log^2 m')$ vectors requires $O(k \cdot \log^2 m' + \log^3 m')$ time, which is $O(k + \log n)$ per vector.

It remains to show how to sort $O(\sqrt{\log m'})$ vectors as an instance of

$$GVSP\{O(\sqrt{\log m'}), O(\sqrt{\log m'}), 0\}$$

(see Figure 8.1). We first rank the vectors by linking them in a sorted list without moving the vectors (we mimic the insertion sort in a list without moving vectors). The list pointers of $O(\log \log m')$ bits each, however, are not encoded with heavy bits in this case. Since we sort one bucket at a time and have $O(\sqrt{\log m'})$ such pointers, we can keep the $O(\sqrt{\log m'} \cdot \log \log m') = o(\log n)$ bits for all the pointers in one auxiliary location. We can access any such pointer in constant time, and we can append a new pointer to them within the same complexity by using RAM operations. We apply Hirschberg's linear scanning to add a new vector to the list and mimic insertion sort. Hence, the cost per vector is $O(k + \sqrt{\log m'})$. After setting up the linked list that reflects the sorted order, we permute the vectors using the temporary buffer of $O(\sqrt{\log m'})$ placeholders. The time complexity to sort $O(\sqrt{\log m'})$ vectors is bounded by $O(k\sqrt{\log n} + \log n)$.

Lemma 8.9 A bucket of $O(\log^2 m')$ vectors can be sorted in $O(\log^2 m' \times (k + \log m'))$ time.

By Lemmas 8.8 and 8.9, we can conclude that:
**Lemma 8.10** An instance of \( GVSP \{O(n/\log n), O(n/\log n), O(n/\log^2 n)\} \) can be solved in \( O(n + nk/\log n) \) time using \( O(1) \) auxiliary locations.

### 8.7 Conclusions

We are now able to state the main result of this chapter:

**Theorem 8.2** An arbitrary set of \( n \) vectors of length \( k \) can be sorted in-place optimally, taking \( O(nk + n \log n) \) time and using \( O(1) \) auxiliary locations.

**Proof:** By Lemma 8.10 and Lemma 8.5, we can solve an instance of \( GVSP \{n - o(n), 0, O(n/\log^2 n)\} \) in \( O(\log n \times (n + nk/\log n) + nk) = O(nk + n \log n) \) time. Within this bound, we can solve \( GVSP \{n, 0, 0\} \) by Lemma 8.3. \( \square \)
Chapter 9

Sorting by Merging or Merging by Sorting?

Abstract

We are given a total order \((\mathcal{U}, \leq)\), with a possibly infinite universe \(\mathcal{U}\). The only operations allowed on input elements are comparisons and moves to empty locations of memory (see Chapter 2).

In this chapter we prove the existence of an algorithm that, for any set of \(s \leq n\) sorted sequences containing a total of \(n\) elements (drawn from \(\mathcal{U}\)) computes the whole sorted sequence with the following resource bounds:

- \(O(n \log s)\) comparisons,
- \(O(n)\) data moves and
- \(O(1)\) auxiliary locations of memory besides the ones strictly necessary for the \(n\) input elements.

This is a huge improvement that settles down the fundamental problem of generalized (or multiway) merging. The best known algorithms with these same bounds are not independent from the values of parameter \(s\) and are strictly limited to the particular case \(s = O(1)\).

From a more intuitive point of view, we prove that it is possible to pass from merging to sorting in a seamless fashion, without losing the optimality with respect to any of the three main complexity measures. Any merging algorithm that has such a complete independence from the values of parameter \(s\) of the merging problem seems to be “more powerful” than any sorting algorithm. As the title of this chapter suggests, in light of our main statement, it seems necessary to rethink the hierarchical relation between sorting and merging that has ever seen the merging as mere subproblem of sorting.

Our main statement has an implication in the field of adaptive sorting algorithms and improves [Franceschini and Geffert, FOCS 2003], showing that it is possible to exploit some form of pre-sortedness to lower the number of comparisons while still maintaining the optimality for space and data moves.
More precisely, let us denote with $\text{Opt}_M(X)$ the cost for sorting a sequence $X$ with an algorithm that is optimal with respect to a pre-sortedness measure $M$. To the best of our knowledge, so far, for any pre-sortedness measure $M$, no full-optimal adaptive sorting algorithms were known (see [Estivill-Castro and Wood, 1992], page 472). The best that could be obtained were algorithms sorting a sequence $X$ using $O(1)$ space, $O(\text{Opt}_M(X))$ comparisons and $O(\text{Opt}_M(X))$ moves. Hence, the move complexity seemed bound to be a function of $M(X)$ (as for the comparison complexity). We prove that there exists a pre-sortedness measure for which that is false: the pre-sortedness measure $\text{Runs}$, defined as the number of ascending contiguous subsequences in a sequence. That follows directly from our main statement, since $\text{Opt}_M(X) = O(|X|\log \text{Runs}(X))$.

The presentation in this chapter is based on the paper [Franceschini, 2005b].

9.1 Introduction

As many famous Computer Scientists have recognized (e.g. [Knuth, 1973]), sorting and merging have always been fundamental problems in computation. In this chapter we prove a (we believe) surprising property that blurs the boundary between these two problems and seems to force to rethink the usual hierarchical relation that sees merging as a mere subproblem of sorting, a simple tool to build an efficient and simple sorting algorithm.

Moreover, as a corollary of our main statement we will prove the existence of a previously unknown kind of adaptive sorting algorithms. In particular, we will prove that there exists a pre-sortedness measure $M$ such that any sequence of elements can be sorted $M$-optimally, move optimally and space optimally simultaneously.

9.1.1 The problem

We are given a total order $(\mathcal{U}, \leq)$, with a possibly infinite universe $\mathcal{U}$. The input element are drawn from $\mathcal{U}$ and they are considered atomic (no bit manipulation, hashing etc...). The only operations allowed on input elements are comparisons between two elements and moves to empty cells of memory. Hence, the complexity of an algorithm will be measured by the usual three metrics: the number of comparisons, the number of moves and the number of cells of memory used, besides the ones necessary to store the input elements. For any such metric, we will refer to the optimality of an algorithm with the usual meaning of asymptotic optimality up to a constant factor. We will call an algorithm full-optimal if it is optimal with respect to the three metrics simultaneously.
In the sorting problem, we are given a set of \( n \) elements from \( U \) and they have to be disposed in the ordered permutation induced by the relation \( \leq \). We will deal with the following, related problems:

**Problem 9.1 (Multiway Merging Problems)** In the balanced merging problem, we are given \( s \leq n \) sorted sequences of \( n/s \) elements each drawn from \( U \) and they have to be fused into a single sorted sequence. In the unbalanced merging problem the total number of input elements is still \( n \) but the \( s \) sorted subsequences to be fused can differ in their lengths.

An algorithm solving the two merging problems is supposed to exploit the presortedness of the input elements in order to arrive to the final sorted sequence with less computational effort.

**A natural question.** Seeing the parametric definition of the merging problems, natural questions arise: How far can we push the parameter \( s \)? How much marked is the boundary between merging and sorting problems? Is there a full-optimal solution for the merging problems for any value of \( s \)?

### 9.1.2 Previous work

The research focusing on the existence of a full-optimal solution for the merging problem has been fervent since the sixties, bringing results for the special case where \( s = O(1) \). In spite of all the research efforts, at the best of our knowledge, so far, even the existence of a full-optimal solution for the merging problem covering any range of values better than \( s = O(1) \) was unknown (if we exclude the range \( n^\varepsilon \leq s \leq n \), for any real constant \( \varepsilon < 1 \), obviously obtainable with a plain application of the sorting algorithm in [Franceschini and Geffert, 2003]).

### 9.1.2.1 Previous efforts

As a further witness of the intrinsic difficulty of finding a full-optimal solution for the merging problem, even for the limited range with \( s = O(1) \), we will quickly review the main results for this case.

The first solution was proposed in [Kronrod, 1969], in that seminal paper fundamental tools for space-optimality, like the internal buffering technique, were introduced. Unfortunately, the two-way merging algorithm of Kronrod contained an insidious error compromising the correctness of the algorithm in the general case of input with repeated elements.

After Kronrod, Horvath [Horvath, 1978] devised a stable (i.e. the initial relative order of equal elements is maintained after the process) merging algorithm assuming the possibility of element modifications. Subsequently, Trabb Pardo [Trabb Pardo, 1977] removed this requirement.
The error in Kronrod's work went undisclosed until [Salowe and Steiger, 1987], when a simpler way to stable merging was devised. In [Huang and Langston, 1988a] an unstable modification of Kronrod algorithm is given but unfortunately also that solution had a similar problem. Later on, the same authors gave a stable algorithm in [Huang and Langston, 1988b].

When \( s = 2 \), the lower bound for the number of comparisons in case of sequences of two different lengths \( m < n \) is \( O(m \log(n/m)) \). Symvonis achieved that lower bound in [Symvonis, 1995] and subsequently Geffert, Katajainen and Pasanen [Geffert, Katajainen, and Pasanen, 2000] gave both stable and unstable algorithms with the same asymptotical bound but better constant factors.

### 9.1.2.2 Adaptive sorting

As we will see, a corollary of our main statement represents an important novelty for the adaptive sorting problem (see [Estivill-Castro and Wood, 1992] for a survey on the subject). In the adaptive sorting problem the initial order of the input elements is quantified with a pre-sortedness measure. The complexity measures for the adaptive sorting algorithms are expressed in function of the chosen pre-sortedness measure.

With the development of this field, many pre-sortedness measures have been introduced together with a concept of optimality for any such measure. The measure Runs is defined as the number of ascending contiguous sub-sequences of the input sequence. Using any in-place merging algorithm as the one in [Salowe and Steiger, 1987], it is possible to achieve adaptive sorting algorithms that are space-optimal and Runs-optimal but sub-optimal with respect to the number of moves.

Similar results can be obtained with other measures but, at the best of our knowledge, so far, for any pre-sortedness measure, no full-optimal adaptive sorting algorithms were known.

### 9.1.3 Less constrained variants of the problem

Let us enter deeply into this matter seeing what happens if we drop the full-optimality constraint and we are allowed, in turn, to use a sub-optimal number of moves, a sub-optimal number of comparisons or a sub-optimal number of auxiliary memory locations.

- If we give up the optimality on the number of moves, a space-optimal and comparison-optimal solution that is independent from the value of \( s \) follows immediately by the existence of a full-optimal solution for the case \( s = 2 \). As we will see later, the research around the existence of an algorithm that could fuse two sequences of \( m \) elements each, using \( O(m) \) comparisons, \( O(m) \) moves and \( O(1) \) auxiliary locations has been active and successful since the late sixties [Kronrod, 1969]. Given that fact, a space-optimal and comparison-optimal solution for the merging problem that is independent from the value
of $s$ is a simple variation of the plain Mergesort. It employs any full-optimal merging algorithm for $s = 2$ and starts the execution merging couples of sorted sequences instead of couples of elements. That brings us the wanted bounds of $O(n \log s)$ comparisons and $O(1)$ auxiliary memory locations at a cost of a sub-optimal $O(n \log s)$ number of moves.

- If we overlook the optimality on the number of comparisons, we can just use the full-optimal algorithm for the sorting problem in [Franceschini and Geffert, 2003] (see Chapter 5). With that algorithm, we can just ignore the sorted sequences, and sort the whole sequence using the optimal $O(n)$ moves and $O(1)$ auxiliary locations at the cost of a sub-optimal $O(n \log n)$ number of comparisons.

- If the space-optimality is to be sacrificed, a linear-space solution that is independent from the value of $s$ and performs $O(n \log s)$ comparisons and $O(n)$ moves can be obtained using a dictionary that is searchable in $O(\log m)$ comparisons (when it contains $O(m)$ elements) and updatable in $O(1)$ moves, either amortized or in the worst case (e.g. [Levcopoulos and Overmars, 1988b, Andersson and Lai, 1991, Fleischer, 1993]...). Any such dictionary can be simply used as a priority queue containing, at any time during the process, the smallest $s$ elements among the ones still in the sorted sequences in input.

### 9.1.4 Our theoretical contribution

In this chapter, we prove the following theorem:

**Theorem**  There exists an algorithm $A$ with the following property: For any $1 \leq s \leq n$ and for any set of $s$ sorted sequences containing a total of $n$ elements (drawn from $\mathcal{U}$), $A$ computes the whole sorted sequence with $O(n \log s)$ comparisons, $O(n)$ moves and $O(1)$ auxiliary cells of memory.

Obviously any solution for the balanced merging problem performs $\Omega(n)$ data moves. In the balanced merging problem the sorted subsequences are assumed to be of equal length. Hence, it is straightforward to prove that any solution for that problem performs $\Omega(n \log s)$ comparisons. Therefore, by the previous theorem, the following holds:

**Corollary (Full-optimal merging)**  There exists a full-optimal solution for the balanced merging problem for any value of $s$. That is, there exists performing $O(n \log s)$ comparisons, $O(n)$ moves and using $O(1)$ auxiliary cells of memory.

Concerning the unbalanced merging problem, if the lengths of the sorted subse-
quences are not considered in the complexity measures, that is if we aim to give complexity bounds that depend only on parameters $n$ and $s$ (as the definition of the problem implies), then the algorithm of Theorem 9.1 is a full-optimal solution for that problem too. Instead, if we are interested in evaluating the complexity of the unbalanced merging problem taking into account also the lengths of the sorted subsequences, let them be $n_1, n_2 \ldots n_s$, then simple approaches using less comparisons come immediately in mind. For example, a simple merging strategy performing at any time a binary merging operation between the two shortest runs at that time, would use $O\left(\sum_{i=1}^{s} n_i \log(n/n_i)\right)$ comparisons. Unfortunately, an approach like this seems to require a number of data moves of the same order.

Our main statement has a consequence involving the field of adaptive sorting algorithms. We will improve the result in [Franceschini and Geffert, 2003], showing that it is possible to exploit some form of pre-sortedness to lower the number of comparisons while still maintaining the optimality for space and data moves. More precisely, let us denote with $\text{Opt}_M(X)$ the cost for sorting a sequence $X$ with an algorithm that is optimal with respect to a pre-sortedness measure $M$. To the best of our knowledge, so far, for any pre-sortedness measure $M$, no full-optimal adaptive sorting algorithms were known (see [Estivill-Castro and Wood, 1992], page 472). The best that could be obtained were algorithms sorting a sequence $X$ using $O(1)$ space, $O(\text{Opt}_M(X))$ comparisons and $O(\text{Opt}_M(X))$ moves. Hence, the move complexity seemed bound to be a function of $M(X)$ (as for the comparison complexity). We prove that there exists a pre-sortedness measure for which that is false.

**Corollary 9.1 (Full-optimal adaptive sorting)** There exists a pre-sortedness measure $M$ and an algorithm $A_M$ such that, for any sequence $X$ of $n$ elements, $A_M$ sorts $X$ with $O(\text{Opt}_M(X))$ comparisons, $O(n)$ moves and $O(1)$ auxiliary cells of memory.

Consider the pre-sortedness measure $\text{Runs}$, defined as the number of ascending contiguous subsequences in a sequence. We know that $\text{Opt}_M(X) = O(|X| \log \text{Runs}(X))$, therefore Corollary 9.1 follows from Theorem 9.1.

Finally, let us explain our contribution from a more intuitive point of view. In this chapter we prove that it is possible to pass from merging to sorting in a seamless fashion, without losing the optimality with respect to any of the three main complexity measures. In light of this fact, we could say that merging is not a mere subroutine with limited power as in the well known sorting by merging approach. Instead, as we will see, a sorting algorithm is a basic subroutine for our full-optimal merging algorithm.

### 9.1.5 Organization of the chapter.

In Section 9.2, we will first give some assumptions for the sake of the presentation. Then, we will introduce some known tools that will be used in the solution. Finally,
we will give a brief description of some of the obstacles posed by the problem and the basic, high level ideas to overcome them. In Section 9.3, we will describe our merging by sorting algorithm.

9.2 Assumptions, Tools, Obstacles and Basic Ideas

9.2.1 Assumptions

In order to make this presentation more readable, we do not stress on formal details (as we said, we usually avoid the use of ceilings and floors).

Moreover, let us assume that the \( n \) elements in input are distinct. Considering the case of repeated elements when the stability of the algorithm is not a main concern, would fill the presentation with technicalities without adding anything valuable from a theoretical point of view. On the other hand, the problem of the stability seems to be very insidious and some of the techniques used in our solution are particularly prone to destabilize the input during the merging process.

9.2.2 Tools

Let us recall briefly some powerful tools we will use in our algorithm. For some of them a detailed description can be found in Chapter 4.

The first tool we use is the internal buffering technique [Kronrod, 1969]. Essentially, some of the elements are used as placeholders to simulate a working area in which the other input elements can be permuted efficiently. Usually, a set \( \mathcal{A} \) of input elements is divided into two subsets \( \mathcal{A}' \) and \( \mathcal{B} \), where the latter is the buffer set and has cardinality \( o(\mathcal{A}) \). Then, \( \mathcal{A}' \) is conquered efficiently with the aid of \( \mathcal{B} \) that can be subsequently conquered with a sub-optimal method. Finally, a last merging step conquers the whole set \( \mathcal{A} \) easily. Other approaches for the internal buffering technique can be found in Chapter 4.

The second tool we use is the bit stealing technique [Munro, 1986]. This technique is very popular and very simple: the value of a bit is encoded in the relative order of two distinct elements (e.g. the increasing order for 0). Stolen bits can be used pretty much as the normal ones. The important difference is that the costs of their use fall directly over the comparison count (e.g. reading a stolen word costs \( O(\log n) \) comparisons in the worst case) and the move count (e.g. modifying a stolen word could costs \( O(\log n) \) moves in the worst case or \( O(1) \) in amortized sense if the word is used as a binary counter [Cormen, Leiserson, Rivest, and Stein, 2001]). The elements that back up the stolen bits and the other elements are divided and conquered with the same simple process used for the internal buffer.

The third tool we use is an in-place linear time two-way merging, any algorithm among the ones we briefly reviewed in Section 9.1 would do the job. As it can be
easily imagined this algorithm will be mainly used for the final merging step in the simple divide and conquer schemes for the internal buffering and stolen bits.

The fourth tool is the full-optimal sorting in [Franceschini and Geffert, 2003] (see Chapter 5). Following our new approach of "merging by sorting" we will use this algorithm to solve proper sub-problems of the main one.

The fifth tool is the well-known, basic technique for space-efficient block exchange. From a block \( X = x_1 \ldots x_t \) of \( t \) consecutive elements we can obtain the reverse \( X^R = x_t \ldots x_1 \) in linear time and in-place simply exchanging \( x_1 \) with \( x_t \), \( x_2 \) with \( x_{t-1} \) and so forth. Two consecutive blocks \( X \) and \( Y \), possibly of different sizes, can be exchanged in-place and linear time with three block reversals, since \( YX = (X^R Y^R)^R \).

### 9.2.3 Some of the obstacles posed by the problem

At first sight, we could think that the main technique used in [Franceschini and Geffert, 2003, 2005] (see Chapter 5) could be used also with this problem. In that paper, the authors essentially used a dictionary that is searchable in \( O(\log n) \) comparisons and updatable in \( O(1) \) data moves in amortized sense. There are two major obstacles.

- First, encoding the auxiliary data used by the dictionary using stolen bits so that the decoding does not penalize the search.

- Second, embedding the dictionary into a large pool of buffer elements in order to achieve the space-optimality while maintaining the update in \( O(1) \) data moves.

Using these techniques to overcome the space inefficiency of the third sub-optimal approach we mentioned in Section 9.1 might seem to be the right way.

Unfortunately, that approach requires that any element inserted in the dictionary is coupled with the index of the sorted sequence it belonged to. That is unavoidable: when the minimum element \( x \) is removed from the dictionary, the new inserted element has to be the successor of \( x \) in the original sorted sequence. In the general case there is no way to predict which sorted subsequence the extracted element came from or in what position of the dictionary the new element will be inserted. Therefore, any element passing through the dictionary would have to be charged with \( O(\log s) \) moves for the encoding of the index of its subsequence of origin.

There is another aspect of the solution in [Franceschini and Geffert, 2003, 2005] (see Chapter 5) that is not suitable for a direct use in the merging problem. Both the set of pairs of distinct elements for the bit stealing and the set of buffer elements are collected using selection and partitioning algorithms. Those algorithms disrupt the the \( s \) sorted sequences in input, thereby nullifying any effort to exploit the pre-sortedness of the elements. As we will see, we are forced to use a smaller amount
of buffer elements \((O(n/polylog(n)))\) elements against the \(O(n)\) elements used in [Franceschini and Geffert, 2003]) in order not to compromise the pre-sortedness of the input. In the full version of this paper we will adapt the solution for the case of repeated elements. In that case we will have to use \(O(n^\epsilon)\) stolen bits against the \(O(n/\log n)\) used in [Franceschini and Geffert, 2003].

### 9.2.4 Basic ideas

The starting intuition is that there has to be a different way to accomplish the task for anyone of three peculiar cases:

(i) when \(s\) is \(O(n^\epsilon)\) and \(\Omega(polylog(n))\);

(ii) when \(s\) is very small, that is \(s = O(polylog(n))\);

(iii) when \(s\) is very large, that is \(s = \Omega(n^\epsilon)\).

The following intuitions will guide us to a full-optimal solution for any of the above sub-problems.

- The way we will solve the first case is just what the approach merging by sorting is all about: breaking the merging problem into sub-problems that can be easily solved with sorting algorithms. The merging problem will be divided into

  (i) one sorting sub-problem of size \(O(n/s)\) but with macro-elements of size \(s\),

  (ii) \(\frac{n}{s}\) sorting sub-problems of size \(s^2\) each and

  (iii) \(\frac{n}{s^2}\) binary merging sub-problems again of size \(s^2\).

The details will be given in Section 9.3.1.

- In the second case, it would be reasonable to think that the solution could be found extending the usual approach of two-way merging to the case of \(polylog(n)\)-way merging. The major obstacle to overcome consists in the fact that when \(s = \omega(1)\) there does not seem to be a way to use less than \(O(n)\) buffer elements. As we mentioned in Section 9.1, that is what prevents the sorting algorithm in [Katajainen and Pasanen, 1999] from being also a full-optimal solution for the merging problem when \(s = O(polylog(n))\).

As we will see, a way to solve the problem in this second range of values of \(s\) consists in breaking the sorted subsequences in pieces as if they were linked lists. That kind of technique could have also been used to make the algorithm in [Katajainen and Pasanen, 1999] stable.
The third case can be seen as the simple base case of the merging by sorting approach. The natural intuition is that $s$ is so large that the solution for the problem has to be more similar to a sorting algorithm than a classical merging algorithm in which, at any step, the currently smallest element among the sorted subsequences is selected.

As a matter of fact, since we are interested in asymptotic optimality up to a constant factor, for any fixed constant $\epsilon$ this case can be solved simply applying the full-optimal sorting algorithm in [Franceschini and Geffert, 2003].

9.3 Merging by Sorting

Let $R$ be the input sequence and $R_1, R_2, R_3 \ldots R_{s-2}, R_{s-1}, R_s$ be the $s$ sorted sequences of $n/s$ elements each. We will distinguish among three main ranges of values for the parameter $s$ and the solutions for these three cases will be given in the next sections.

For the sake of simplicity, in any of these sub-sections we will first describe the solution assuming $s$ sorted sequences of the same length. After that, we will point out in the proof the necessary changes for sorted sequences of generic lengths. As we will see, these changes are very simple for the second range of values of $s$ and almost null for the first and last ones.

9.3.1 What if $\log^2 n \leq s \leq n^\epsilon$?

As we will see soon, we are going to need some “simulated resources”. We need $O(\frac{n}{\log n}) = O(\frac{n}{\log n})$ stolen bits. We can collect the $O(\frac{n}{\log n})$ pairs of two distinct elements each simply taking the first $O(\log n)$ sorted subsequences $R_1, R_2, \ldots$, $O(\frac{n}{\log n})$ buffer elements can be collected in the same way.

Let $t$ be the index of the first remaining sorted subsequence. Since $s - t = \Theta(s)$, for the sake of simplicity we are going to pretend that, after collecting stolen bits and buffer elements, we are left with two new objects. These are the sequences $B$ and $E$ containing the wanted buffer elements and pairs of encoding elements, respectively. All the objects so far introduced are initially laid out in memory in the following way: $EBR_1R_2R_3 \ldots R_{s-2}R_{s-1}R_s$.

Let us consider each sorted subsequence as logically divided into $n/s^2$ blocks of $s$ contiguous elements each. We have four main phases.

(i) We sort all the blocks according to their first elements (since any block is already sorted internally, its first element is also its smallest one). We can use the mergesort with the in-place linear time two-way merging algorithm we chose in Section 9.2. However, in order to stay within our resource bounds, we must sort only the set of the first elements of the blocks. Hence, we collect this set exchanging each one of its members with one buffer element in $B$. Finally,
9.3. MERGING BY SORTING

to maintain the connection between the set of the first elements of the blocks and the remaining parts of the blocks, we associate a pair of back and forward encoded pointers (of $O(\log n)$ stolen bits each) to each pair \textit{[first element, remaining elements of the block]}. This encoded information can be easily maintained up to date during the execution of mergesort and subsequently used to bring the blocks in sorted order in only $O\left(\frac{n}{s}\right)$ “block moves”.

(ii) Let $B_1 B_2 \ldots B_{2^s-1} B_2$ be the sorted sequence of the blocks after the first step. Let us logically form $\frac{n}{s}$ groups $G_1, G_2, \ldots, G_{\frac{n}{s}}$ of $s$ contiguous blocks each. In this phase we sort each group $G_i$ using the algorithm in [Franceschini and Geffert, 2003].

(iii) Let $G'_1 G'_2 \ldots G'_{\frac{n}{s}}$ be the sequence of sorted groups we obtained after the second phase. In this phase, we apply the in-place two-way merging algorithm we chose in Section 9.2 in a left-to-right “chained” fashion starting with the two first sorted groups $G'_1, G'_2$ then with $G'_2, G'_3, G'_3, G'_4$ and so forth until the pair $G'_{\frac{n}{s}-1}, G'_{\frac{n}{s}}$ is merged.

(iv) We sort the buffer elements and the elements used to steal bits. Then we merge that sorted sequence with the one obtained after the execution of the third phase.

\textbf{Lemma 9.1} For any $\log^2 n \leq s \leq n^\varepsilon$ and for any set of $s$ sorted sequences containing a total of $n$ elements (drawn from $U$), we can compute the whole sorted sequence with $O(n \log s)$ comparisons, $O(n)$ moves and $O(1)$ auxiliary cells of memory.

\textit{Proof sketch:} In the first phase, we use the normal in-place binary mergesort over the set of the first elements of the blocks. That alone would cost $O\left(\frac{n}{s}\right) = o(n)$ comparisons and moves. However, at any basic step of the binary mergesort we have to decode and re-encode a constant number of pointers with $O(\log n)$ stolen bits each, for a total cost of $O\left(\frac{n}{s} \log n\right)$ moves and comparisons. That is $O(n)$ by the hypothesis over the values of $s$. The final permutation of the blocks costs $O(n)$ moves and $O\left(\frac{n}{s} \log n\right) = o(n)$ comparisons. In the second phase $O\left(\frac{n}{s}\right)$ groups of $s^2$ elements each are internally sorted using the full-optimal sorting algorithm in [Franceschini and Geffert, 2003]. For each group, $O(s^2 \log s)$ comparisons and $O(s^2)$ moves are spent, for a total of $O(n \log s)$ comparisons and $O(n)$ moves. The third phase exploits a combinatorial property that is a generalization of the one introduced in [Kronrod, 1969]. Basically, after the second phase, any element in the sequence $G'_1 G'_2 \ldots G'_{\frac{n}{s}}$ may be at most one group (excluding its own one) above its final position in the corresponding sorting sequence. Therefore, the “chained”, left-to-right application of the binary in-place merging yields the sorted sequence in $O(n)$ moves and comparisons. Finally, the fourth phase easily conquers the sub-problem for the elements used to simulate the resources.
If the \( s \) sorted subsequences have generic lengths, we have to add a simple preprocessing phase executed before the four phases we described. This additional phase is needed in order to ensure the initial assumption about the presence of \( n / s \) sorted blocks of \( s \) contiguous elements each. The additional phase is a scan of the sequence in input. Starting from the left end of the sequence we consider \( s \) contiguous elements at a time. If they are in sorted order we go to the next \( s \), otherwise we sort them. Since the input sequence is composed by \( s \) sorted subsequences, it is straightforward to prove that the preprocessing phase performs \( O(n + s^2 \log s) \) comparisons and moves. The constant \( \epsilon \) can be chosen so that the cost of the preprocessing phase is \( O(n) \). After the preprocessing phase the algorithm continues with the remaining four phases unchanged. \( \Box \)

9.3.2 What if \( s < \log^2 n \)?

We are going to show how to solve the problem just for the case \( s \leq \frac{\log n}{\log \log n} \). When \( \frac{\log n}{\log \log n} < s < \log^2 n \), we can solve the problem simply applying iteratively the solution for the case \( s = \frac{\log n}{\log \log n} \) as if we were sorting the sequence by merge sort (a \( \frac{\log n}{\log \log n} \)-way merge sort algorithm). Let \( g = \frac{\log n}{\log \log n} \). The \( g \)-way merge sort would scan the \( n \) elements \( O(\log s / \log g) \) times, performing a total of \( O((\log s / \log g)n \log g)) = O(n \log s) \) comparisons and \( O((\log s / \log g)n) = O(n) \) moves.

9.3.2.1 Breaking the subsequences

The major obstacle to overcome in this case consists in the fact that when \( s = \omega(1) \) there does not seem to be a way to use less than \( O(n) \) buffer elements. Some ideas suitable for this case are also in [Katajainen and Pasanen, 1999]. As we mentioned in Section 9.1, that is what prevents the sorting algorithm in [Katajainen and Pasanen, 1999] from being also a full-optimal solution for the merging problem when \( s = O(polylog(n)) \). We solve this problem breaking the sorted sequences into linked lists.

Let us divide any sorted subsequence \( R_i \) into \( p \) contiguous blocks of size \( \log^2 n \). The \( j \)th block of \( R_i \) will be denoted by \( R_j^i \). Block \( R_1^i \) contains the smallest \( \log^2 n \) elements of \( R_i \), block \( R_2^i \) the second smallest \( \log^2 n \) elements and so forth. Initially the blocks of \( R_i \) are laid out contiguously and in sorted order, (i.e. \( R_i = R_1^i R_2^i \ldots R_p^i \)).

The blocked subsequences can be naturally seen as \( s \) doubly-linked lists of \( p \) macro-elements each (the blocks). In the following we will freely refer to a generic \( R_i \) as a list or as a sorted subsequence. As we will see, the introduction of those simple lists will be of great help. By allowing the possibility of merging \( s \) linked lists of sorted elements instead of \( s \) unbreakable sorted sequences, the need for buffer elements drops from \( O(s \times n / s) = O(n) \) units to \( O(s \times \text{block-size}) = O(s \log^2 n) \) only.
9.3. MERGING BY SORTING

As in the previous case, some \( o(n) \) elements will be devoted to placeholdering or encoding duties. However, this time the numbers are slightly different, especially for what concern the buffer elements. The quantities of stolen bits and buffer elements we have to collect depend on the lists:

- Since the blocks are part of doubly-linked lists that will be scattered in the \( n \) locations of memory, they are going to need \( \text{succ} \) and \( \text{pred} \) pointers of \( O(\log n) \) bits each.

- We are going to iteratively extract the smallest elements of lists. We will need \( \log^2 n \) buffer elements for any list so that the extraction can be reduced to an exchange with a buffer element.

Therefore, we need \( O\left(\frac{n}{\log n}\right) \) stolen bits and \( s \log^2 n \) buffer elements. The corresponding elements can be collected in the same way we did in Section 9.3.1. As we already did in that section, we are going to pretend for the sake of presentation that the number of sorted subsequences to be processed is still \( s \) and that we are left with two new objects: the sequences \( B \) and \( E \) containing the wanted buffer elements and the encoded bits. All the objects so far introduced are initially laid out in memory in the following way: \( EB R_1 R_2 R_3 \ldots R_{s-2} R_{s-1} R_s \).

9.3.3 Layout of the memory and invariants

Now we will describe the layout of the memory right before the merging phase begins and the invariants defined over the layout that will be maintained during the computation. During the merging phase, the \( n \) cells of memory are partitioned into five zones. We will list them from the leftmost to the rightmost one (see Fig. 9.1).

- The encoding zone \( \mathcal{E} \). It contains the elements for stealing bits. This zone is static. During the merging phase there will be a lot of activity here, due to the continuous execution of encoding/decoding-related comparisons and moves. However, the boundaries of that zone will never change.

- The sorted zone \( \mathcal{I} \). At any time during the merging phase, it contains the \( |\mathcal{I}| \) smallest elements (with the exception of buffer elements and the ones for stealing bits, of course) in sorted order.

- The buffer zone \( \mathcal{B} \). At any time of the merging phase, it contains a subset of the collected buffer elements. Initially \( \mathcal{B} \) contains all the buffer elements but the extractions from the lists will make its size shrink or enlarge during the merging phase. Moreover, that zone will move toward the right end of the memory during the whole merging phase, because of the "pressure" by the ever-growing zone \( \mathcal{I} \). At the end of the computation \( \mathcal{B} \) will have regained all the buffer elements and will end up at the right end of the memory.
Figure 9.1: The memory layout.

- The block zone $\mathcal{B}$. At any time of the merging phase and for any list $R_i$, it will contain the currently remaining blocks of $R_i$ with the exception of the first (the one with the smallest elements). Since the merging process will continuously take away batches of elements from the tops of the lists, the meaning of “currently remaining blocks” of a list should be clear. The left boundary of $\mathcal{B}$ will move toward right. At the end of the computation, this zone will be empty, given that the objective of the merging phase is to have all the elements of the lists transferred into $\mathcal{J}$ in sorted order.

- The leading zone $\mathcal{L}$. This zone has fixed boundaries and comprises $s \log^2 n$ locations. At any time during the merging phase, the remaining elements of the first block of any list $R_i$ are stored contiguously, in sorted order and padded with a sufficient number of buffer elements (i.e. if $l$ is the number of the remaining elements of the first block of list $R_i$ then the elements are laid out as $f_1f_2\ldots f_{\log^2 n-1} r_1r_2\ldots n$, where the $r_i$s are the element of $R_i$). The leading zone will be at the center of the merging process since it is in that zone that the $s$-way choices will be made.

**Lemma 9.2** The costs of bringing the zones in their initial state before the merging phase are within our target bounds.

**Proof sketch:** Initially, the elements are distributed in memory in the following way

$$EBR_1^1R_1^2\ldots R_{i-1}^p-1R_1^pR_2^1R_2^2\ldots R_{i-1}^p-1R_{i-1}^pR_{i-1}^1R_{i-1}^2\ldots R_{i-1}^p-1R_{i-1}^pR_{i-1}^p$$

Basically, we have to do $s$ block exchanges to move $R_1^1, R_2^1, \ldots, R_{i-1}^1, R_i^1$ into the last $s \log^2 n$ locations (that is into the leading zone $\mathcal{L}$). The total cost is $O(s \log^2 n)$ moves for the block exchanges plus $O(s \log n)$ moves for updating the succ and pred pointers of any block involved in the exchange. \hfill $\Box$

### 9.3.3.1 Merging phase

We have the problem of how to keep track of the position of the smallest element of any list. We are treating the case $s \leq \frac{\log n}{\log \log n}$ and therefore we cannot count on
9.3. MERGING BY SORTING

a lower bound for $s$. That excludes completely any solution involving encoding by bit stealing the value of any such pointer (the decoding would cost more than the wanted $O(\log s)$ comparisons). Since we want to maintain all these elements trapped into the leading zone $\mathcal{L}$, each pointer to one of them needs only $O(\log|\mathcal{L}|) = O(\log(s\log^2 n)) = O(\log \log n)$ bits. Since $s \leq \frac{\log n}{\log \log n}$, we can maintain a small balanced tree of such small pointers into a constant number of auxiliary cells of memory (which we are allowed to use). For any $1 \leq i \leq s$, together with the smallest pointer to the smallest element belonging to the list $R_i$, we are going to maintain also a small integer with the value of the number of buffer elements in the block of $R_i$ currently contained in $\mathcal{L}$ ($O(\log \log n)$ bits are needed in that case too). A similar approach has been used also in [Katajainen and Pasanen, 1999] and is the combination of two classic basic techniques: integer packing and merging by selection tree.

The tree will be used to guide the iterative selection of the currently smallest element. In the merging phase the following steps will be executed until all the elements of the lists end up in sorted order in zone $\mathcal{I}$.

1. Find the smallest element among the ones contained into the leading zone $\mathcal{L}$.

2. Exchange this element with the first (leftmost) element of the buffer zone $\mathcal{B}$.
   (This implicitly enlarge and shrink by one position the sorted zone $\mathcal{I}$ and the buffer zone $\mathcal{B}$, respectively)

3. If the block in $\mathcal{L}$ corresponding to the just exchanged element, now contains only buffer elements, we load into $\mathcal{L}$ the next block in its list.

**Lemma 9.3** For any $s < \log^2 n$ and for any set of $s$ sorted sequences containing a total of $n$ elements (drawn from $U$), we can compute the whole sorted sequence with $O(n \log s)$ comparisons, $O(n)$ moves and $O(1)$ auxiliary cells of memory.

**Proof sketch:** The costs of bringing the zones in their initial state before the merging phase are within our target bounds (Lemma 9.2).

Now we have to consider the costs of the merging phase. In step 1 we use the small tree to find the smallest element in $\mathcal{L}$. The small tree has $s$ nodes, its pointers are completely contained into a constant number of auxiliary locations and is fully balanced. Therefore searching and updating the tree costs $O(\log s)$ comparisons.

Step 2 is a mere exchange of elements (we can maintain the starting location of the five zones into as many auxiliary locations). Finally, step 3 consists in an access to the small tree, two block exchanges (we first exchange the block $b$ in $\mathcal{L}$ with the next one in its list, then we exchange again the block $b$, now in $\mathcal{B}$, with the first block in $\mathcal{R}$, thus enlarging and shrinking of $\log^2 n$ positions the sizes, respectively, of $\mathcal{B}$ and $\mathcal{R}$) and the update of $O(\log n)$ stolen bits (linked lists informations). Since this operation can be charged on $\Omega(\log^2 n)$ steps gone without block transfers, the amortized costs of step 3 is $O(1)$ comparisons and moves for any moved element.
The changes for the case of sorted subsequences of generic lengths are minimal and straightforward. That is because the algorithm is already capable to manage the fact that during the evolution of the merging process the remainders of the sorted subsequences are going to differ in length. Therefore, starting from an initial input sequence with sorted subsequences of generic length is just a special case of the common situations managed during the merging process.

9.3.4 What if \( s > n^\epsilon \)?

For any fixed real constant \( \epsilon < 1 \), if \( s > n^\epsilon \) then the problem can be easily solved by applying the sorting algorithm in [Franceschini and Geffert, 2003] to the whole input sequence \( R \), completely ignoring its pre-sortedness. We would like to point out that this can be seen as a base case of the merging by sorting approach in which the sorting subroutine can solve the main problem of merging by itself. Similarly, for the dual companion of the sorting by merging approach, the particular case in which there are only a constant number of sorted sub-sequences in the input sequence can be solved directly by the merging subroutine.

**Lemma 9.4** For any \( n^\epsilon < s \leq n \) and for any set of \( s \) sorted sequences containing a total of \( n \) elements (drawn from \( U \)), we can compute the whole sorted sequence with \( O(n \log s) \) comparisons, \( O(n) \) moves and \( O(1) \) auxiliary cells of memory.

**Proof:** In [Franceschini and Geffert, 2003], it has been proven that there exists a full-optimal solution for the sorting problem using \( O(n \log n) \) comparisons, \( O(n) \) moves and \( O(1) \) auxiliary cells. If \( s > n^\epsilon \), for a fixed real constant \( \epsilon \), that solution is also a full-optimal solution for the balanced merging problem. Since we care about the parameter \( s \) only, the case of sorted subsequences with generic lengths has the same solution.

9.4 Conclusions and Open Problems

In light of Lemmas 9.4, 9.1 and 9.3 we are now allowed to state our main result.

**Theorem 9.1** There exists an algorithm \( A \) with the following property: For any \( 1 \leq s \leq n \) and for any set of \( s \) sorted sequences containing a total of \( n \) elements (drawn from \( U \)), \( A \) computes the whole sorted sequence with \( O(n \log s) \) comparisons, \( O(n) \) moves and \( O(1) \) auxiliary cells of memory.

By Theorem 9.1, the following holds:

**Corollary 9.2 (Full-optimal merging)** There exists a full-optimal solution for the balanced merging problem for any value of \( s \). That is, there exists performing \( O(n \log s) \) comparisons, \( O(n) \) moves and using \( O(1) \) auxiliary cells of memory.
We obtain also the following result in the field of adaptive sorting algorithms:

**Corollary 9.3 (Full-optimal adaptive sorting)** There exists a pre-sortedness measure, that is Runs, and a full-optimal adaptive sorting algorithm for that measure. In other words, there exists an algorithm $A$ such that, for any sequence $X$ of $n$ elements, $A$ sorts $X$ with $O(n \log \text{Runs}(X))$ comparisons, $O(n)$ moves and $O(1)$ auxiliary locations of memory.

As for the open problems, the one concerning the stability of the algorithm seems to be the more challenging. The history of the Computer Science branch of research focusing on space-efficient algorithms can certainly teach us that the difficulties added by the stability constraint can be rather overwhelming. To understand that, it is sufficient to make a quick comparison between the first in-place merging [Kronrod, 1969] and the first stable one [Trabb Pardo, 1977]. However, a full-optimal stable sorting algorithm has been discovered recently [Franceschini, 2005a] (see Chapter 6). Since a full-optimal sorting algorithm is an angular stone in the merging by sorting approach we introduced, maybe a full-optimal stable merging algorithm is not that far in the future.
Part III

Searching
“Imagine how hard it would be to use a dictionary if its words were not alphabetized!”

— Donald E. Knuth,
Chapter 10

No Sorting? Better Searching! (…then why don’t just slave off Part II?)

Abstract

Given \( n \) elements drawn from a total order, if a comparison can be computed in constant time then the best organization (in the RAM model) in order to search among the elements is maintaining them in sorted order. In that case searching requires \( \Theta(\log n) \) comparisons in the worst case, which is optimal.

In this chapter, we demonstrate that this basic fact in data structures does not hold for the general case of multi-dimensional elements, whose comparison cost is proportional to their length. In two papers [Andersson, Hagerup, Hästad, and Petersson, 1994, Andersson, Hästad, and Petersson, 1995b] and the full version in [Andersson, Hagerup, Hästad, and Petersson, 2001], Andersson et al. study the complexity of searching a sorted array of \( n \) elements, each of length \( k \), arranged in lexicographic (or alphabetic) order for an arbitrary, possibly unbounded, ordered alphabet. They give sophisticated arguments for proving a tight bound in the worst case for this basic data organization, up to a constant factor, obtaining

\[
\Theta\left(\frac{k \log \log n}{\log \log (4 + \frac{k \log \log n}{\log n})} + k + \log n\right)
\]

component comparisons (or probes). Note that the bound is \( \Theta(\log n) \) when \( k = 1 \), which is the case that is well known in algorithmics.

We describe a novel permutation scheme of the \( n \) elements that is different from the sorted order, and sorting is just the starting point for describing our preprocessing. When elements are stored according to this “unsorted” order in the array, the complexity of searching drops to

\[
\Theta\left(k + \log n\right)
\]

component comparisons (or probes) in the worst case, which is optimal among all possible permutations of the \( n \) elements in the array, up to a constant factor. Again, the bound is \( \Theta(\log n) \) when \( k = 1 \).
Jointly with the aforementioned result of Andersson et al., our finding provably shows that keeping $k$-dimensional elements sorted in an array is not the best data organization for searching. This fact was not observable before by just considering $k = O(1)$ as sorting is an optimal organization in this case. More implications of our result are commented in the introduction of the chapter and in Chapter 3.

The presentation in this chapter is based on the paper [Franceschini and Grossi, 2004b] (FOCS 2004).

### 10.1 Introduction

#### 10.1.1 When the sorted order is not ideal

"Imagine how hard it would be to use a dictionary if its words were not alphabetized!" As noted at the beginning of Knuth's book [Knuth, 1973], sorting $n$ elements in an array provides a basic data organization for optimal searching, with $\Theta(\log n)$ time and comparisons in the worst case. This fact is corroborated by our common sense and everyday practice; sorting and searching are strictly related companions in designing and analyzing many algorithms for the comparison model. Does this viewpoint completely over the intrinsic complexity of this basic problem? We show in this chapter that the sorted order is not the best general way to permute multi-dimensional elements for optimal searching. This kind of result is completely new in the RAM model, where in general there is no need to maintain data locality to achieve optimal bounds.

**Known cases.** In other models where the need for data locality is pressing, things are different. For example, in the External Memory model (see Chapter 2) an optimal searching algorithm require $O(\log_B n)$ block transfers, where $B$ is the size of the block of adjacent elements that can be transferred in main memory in one time. In this case the sorted order is not the optimal layout for searching and this can be intuitively understood looking at the binary search process (that is strongly linked with the sorted order). This process may require $O(\log n)$ block transfers and hence is not optimal in the External Memory model.

However, the binary search process is somewhat near to the approach that has to be followed in this model. Instead of a binary search we can use a $B$-ary search. Let us be a little loose with details for the sake of example: in the first step we compare the searching element with the set $S$ of $O(B)$ elements of rank $n/B, 2n/B, 3n/B, \ldots$; then we recur in one of the $O(B)$ disjoint sets of $O(n/B)$ elements each induced by $S$. Of course an approach like this can still require $\omega(\log_B n)$ block transfers if we have to jump too much in memory in order to recover the elements in $S$ of each recursive step. Changing the permutation of the elements in memory is the solution for this
problem: first, store \( S \) in \( O(B) \) contiguous locations; then, process recursively \( O(B) \) disjoint sets of \( O(n/B) \) elements each induced by \( S \).

In the Cache-Oblivious model (see Chapter 2) the need for data locality is even greater than in the case of External Memory model, since the parameters of the memory hierarchy, \( B \) in particular, are unknown to the algorithms. However, static searching (asymptotical) optimality can be obtained by permuting the elements following the van Emde Boas recursive layout (see Chapter 7).

**The case of the RAM model.** As we have just seen, in other models the reason why a sorted sequence does not allow optimal searching is in the need to maintain data locality. In the RAM model that is not the motivation. The sub-optimality of the sorted permutation appears when two requirements meet:

- *Space optimality*, that is using only \( O(1) \) auxiliary locations to store values representable with \( O(\log n) \) bits.

- *Optimality in a multidimensional domain* and, in particular, when the elements to be permuted cannot be compared in \( O(1) \) time (e.g., vectors).

### 10.1.2 The result

**The setting.** For a complete description of the model see Chapter 2. We are given \( n \) elements, where each element is a sequence of \( k \) symbols drawn from an arbitrary, possibly unbounded, ordered alphabet. The elements can be maintained in any permutation, and the \( i \)th element in the permutation can be selected in constant time. Given any such element, the \( j \)th symbol in it can be accessed in constant time as well.

Conceptually, it is more useful to think of the permutation as a \( k \times n \) matrix \( \mathcal{A} \) in which the \( i \)th column contains the \( i \)th element, say \( x \); hence, \( x \) itself can be considered as an array \( x \equiv x[1\ldots k] \). The elements underlie the lexicographic (alphabetic) order, namely, for any two elements \( x, y \) in \( \mathcal{A} \), we have \( x \leq y \) if and only if either \( x = y \) or there exists \( j < k \) such that \( x[1\ldots j] = y[1\ldots j] \) and \( x[j+1] < y[j+1] \). In this model, we can only compare the individual symbols of the elements (e.g., no hashing or bit manipulation involving these symbols). We measure the time complexity by accounting interchangeably for the number of comparisons or component probes, as they are linearly proportional to each other in our case. Hence, comparing any two elements \( x \) and \( y \) requires \( O(k) \) time.

**History of the problem.** We study the fundamental problem of establishing whether a given element of length \( k \) appears in \( \mathcal{A} \) as one of its stored elements. Setting \( k = 1 \) we obtain the classical searching problem for an array \( \mathcal{A} \). After sorting it, we can run binary searching with optimal cost \( \Theta(\log n) \). For arbitrary values of \( k \), previous work focused on elements sorted under the lexicographic order.
The problem was introduced by Hirschberg [Hirschberg, 1978], with upper bounds of \( O(k + n) \) and \( O(k \log n) \) in the worst case. The former is obtained by scanning \( \mathcal{A} \) while the latter is a simple binary search on \( \mathcal{A} \). The worst-case lower bound of \( \Omega(k + \log n) \) follows quite easily. The logarithmic term in \( n \) comes from the decision tree for searching in a set of \( n \) elements, while the linear term in \( k \) comes from the need of reading all the \( k \) symbols in the search element.

The first nontrivial upper bound was

\[
O \left( \frac{k \log n}{\log k} \right)
\]


\[
\Theta \left( \frac{k \log \log n}{\log \log (4 + \frac{k \log \log n}{\log n})} + k + \log n \right)
\]

cOMPONENT comparisons in the worst case. Note that the bound is \( \Theta(\log n) \) when \( k = 1 \), which is a well known fact in algorithmics.

Using the same model as that adopted in previous work, we describe a novel permutation of the \( n \) elements in \( \mathcal{A} \) that is different from the sorted order (used only as the starting point for defining our preprocessing). When elements are stored according to this “unsorted” order in the array, we show that the worst-case complexity of searching in the array is

\[
\Theta (k + \log n)
\]

component comparisons, which is asymptotically optimal among all possible permutations of the \( n \) elements in the array including those given by sorting. Also this bound is \( \Theta(\log n) \) when \( k = 1 \). More precisely, we are going to prove the following statement:

For any set \( \mathcal{U} \) and for any set \( S \subseteq \mathcal{U}^k \) of \( n \) elements there exists a permutation \( \mathcal{P}(S) \) different from the lexicographically sorted one that can be searched using \( O(k + \log n) \) component comparisons and \( O(1) \) values of auxiliary information. Therefore, the complexity of space optimal searching is \( \Theta(k + \log n) \).

As a result, we provide an unexpected insight on the relation between sorting and searching. We show that sorted arrays are not the best data organization suitable for searching \( k \)-dimensional elements. In this sense, sorting is an optimal placement of elements for searching an array only when \( k = O(1) \).
Other features of the new permutation. Additionally, we can compute the rank of a search element among those stored in $\mathcal{A}$, with a cost of $\Theta(k + \log n)$, and we can identify its predecessor or successor within the same bounds. We can also list the elements in $\mathcal{A}$ belonging to a given input interval $[a, b]$ for two elements $a \leq b$. We attain an output-sensitive cost of $\Theta(k + \log n + \#\text{retrieved})$, where $\#\text{retrieved}$ denotes the number of elements in $\mathcal{A}$ that belong to the interval $[a, b]$. Since the latter bounds cannot be achieved with the sorted array alone, this strengthens the fact that our data organization is more powerful than putting the elements in sorted order. As a side remark, we can obtain a sorted array from our permutation and vice versa in $O(nk)$ time; hence, our permutation is efficiently reversible and efficiently and space optimally computable in light of the result in Chapter 8.

Interestingly, our bounds hold also for the dictionary in the bit probe model [Elias and Flower, 1975, Yao, 1984]. In this model, the elements are distinct binary strings and the complexity accounts for the number of bits probed in the array $\mathcal{A}$ storing them. We can store these elements permuted in $nk$ bits, so that membership query requires $\Theta(k)$ bit probes in the worst case, which is optimal. Note that our bound gives an alternative perspective to Yao’s result [Yao, 1984] on achieving an optimal search for elements that can be permuted in an array without using extra space (i.e., for storing the name of a suitable hashing function). It also relates to the issues on extra space studied for perfect hashing by Fredman, Komlós and Szemerédi [Fredman, Komlós, and Szemerédi, 1984], non-oblivious hashing by Fiat, Naor, Schmidt and Siegel [Fiat, Naor, Schmidt, and Siegel, 1988b, 1992], and implicit probe search by Fiat et al. [Fiat and Naor, 1989, 1993].

What if the requirements are given up? If the space optimality is given up then the bounds above can be achieved with other data structures keeping the elements sorted and exploiting additional information in extra space (e.g., the longest common prefix information in suffix arrays by Manber et al. [Manber and Myers, 1993] applied to $\mathcal{A}$ or the extra fields in ternary search trees by Bentley et al. [Bentley and Sedgewick, 1997]). The result of Andersson et al. deals with an array alone, without any additional preprocessing and information (e.g., no pointers or integers) apart from $O(1)$ values, namely, the address of the array and its size $n \times k$. In this sense, we obtain the first optimal implicit organization for $k$-dimensional elements, since sorted arrays are optimal just for $k = O(1)$. To see the connection, let’s recall that Munro et al. [Munro and Suwanda, 1980] define an implicit data structure as a permutation of the elements plus just $O(1)$ locations.

We can give up the requirement of having to deal with elements that are not comparable in $O(1)$ time as in the problem of arranging $n$ records with $k$ fields into a $k \times n$ array so that searches can be performed quickly for any given field value. Searches under this model can be supported in $O(\log n)$ time by Fiat et al. [Fiat, Munro, Naor, Schäffer, Schmidt, and Siegel, 1991], where the “$O$” includes a factor of $k \log k$. This method cannot be extended to solve our problem.
Chapter organization. The presentation in this chapter is based on the paper [Franceschini and Grossi, 2004b] (FOCS 2004) and is organized as follows. In Section 10.2, we give an overview of the new permutation. In Section 10.3, we show how to encode bits with elements of unbounded length and, hence, how to implicitly represent the extra information for optimal searching. In Section 10.4, we describe how to exploit the encoding by devising a search algorithm that probes few bits during its execution.

10.2 Overview

Main issues. In our setting we are required to consider only space optimal search algorithms together with an input domain with elements with \( k \) components, where \( k \) cannot be considered a constant and where each component plays a role in determining the relative order between a pair of elements (i.e. two elements cannot be compared in \( O(1) \) time in the general case). Because of this we will face two main issues.

The first issue concerns the encoding of information in the permutation of the input elements. The plain use of the technique of bit stealing, that is storing a bit in the relative order of two distinct elements would not lead us anywhere near optimality:

- It would imply the use of \( O(k) \) component probes for any bit decoded since we have to find a mismatch in order to distinguish the elements and decode the encoded bit.

- Using encoded bits at that price is completely useless in our problem, unless one can devise a way to search in \( O(k + \log n) \) probes while decoding only a constant number of bits. Obviously, that would not make any sense because we are already allowed to use \( O(\log n) \) bits explicitly stored into \( O(1) \) auxiliary locations of memory and hence there would not be any reason to encode \( O(1) \) bits.

The second issue concerns the need to search in possibly large subsets of keys using few encoded bits. Even if we were capable to overcome the first issue and find an efficient way to collect, encode and decode a large amount of bits, they would still be \( O(n) \) (if we were really lucky). Many algorithms for searching in a set of strings or vectors were devised in the past. Maybe the most famous and flexible is the one Manber and Myers introduced for the suffix array, a data structure for the full text indexing problem. A search algorithm like the one in [Manber and Myers, 1993] (see Chapter 4 for a description of it) uses too much auxiliary resources. In particular:
• It uses too many bits. As it can be seen from the overview in Chapter 4, the number of bits used is \(O(n \log k)\), since at least a longest common prefix length value has to be stored for any element.

• It uses the bits too often. At any step of the search the longest common prefix length of a pair of elements in \(S\) has to be retrieved. This is not very much of a problem when auxiliary space is allowed but it becomes a big one when we have to rely only on encoded space and obviously we cannot exploit any parallelism in order to decode an encoded word.

10.2.1 First step: efficient bit stealing with \(k\)-dimensional elements

As we saw in Chapter 4, two distinct elements do not have necessarily to be in adjacent positions to be used to steal a bit. If their positions can be retrieved in constant time and without comparisons there is still a gain. This is very important in the multidimensional setting because we can exploit properties of the lexicographically sorted sequence \(S\) of the input elements (that we assumed to be given to us) that otherwise would go wasted. Let us denote with \(lcp(x, y)\) the length of the longest common prefix of \(x\) and \(y\). Consider the following two approaches to the problem:

• In general, for any index \(i\), the quantity \(lcp(S[i], S[i+1])\) cannot be bounded in function of \(i\). That is for any two adjacent elements in the lexicographically sorted sequence, their first mismatch cannot be found without an amount of comparisons depending on \(k\). Since the knowledge of the location of a mismatch is fundamental to bit stealing, that approach is not the ideal one.

• Let us consider \(d_1 = lcp(S[1], S[n])\), that is the longest common prefix of the first and last element in the sorted sequence. They are used to steal a first bit. Let us continue with \(d_2 = lcp(S[2], S[n-1])\). Since \(S\) is lexicographically sorted, we know that \(d_2 \geq d_1\). Otherwise we would have either \(lcp(S[1], S[n-1]) < d_1\) or \(lcp(S[1], S[2]) < d_1\) and hence one of the two elements in positions 2 and \(i-1\) would be either greater than \(S[n]\) or less than \(S[1]\), against the hypothesis of \(S\) being lexicographically sorted. That pair of elements is used to steal a second bit.

We can continue in that way with the pairs \(S[3], S[n-2]\), \(S[4], S[n-3]\) and so forth. The important thing to note is that in the end we will have individuated at most \(k\) different longest common prefix lengths \(d_1, d_2 \ldots\). Moreover, the symmetric pairs having longest common prefix of length \(d_i\) are “contiguous” (with the obvious extended meaning of the case) in \(S\). Hence, we have to deal with at most \(k\) pairs of symmetric intervals whose symmetric pairs share the same longest common prefix.
Therefore, using the second approach above,

(a) if we were able to encode somewhere the $O(k(\log k + \log n))$ bits of the information describing the symmetric intervals and

(b) if we were able to redirect any search to only one pair of symmetrical intervals,

we would be able to reduce the original problem into one in which we can exploit a linear number of stolen bits that can be decoded in $O(1)$ time each. As we will see in Section 10.3, premises (a) and (b) can be guaranteed using the symmetric intervals approach recursively on a limited amount of elements ($O(\log n)$).

### 10.2.2 Second step: a more parsimonious search algorithm

As we said the problems with Manber and Myers’ searching algorithm are two.

- The first problem is that it uses too many bits of auxiliary information. Since it has to store at least one $lcp$ value for each element in the sequence the total number of bits used for a sequence of $m$ elements is $O(m \log k)$. As we saw previously, after the reduction of the original searching problem into the one of searching a pair of symmetric interval, we can dispose only of a linear number of stolen bits. As we will see this problem can be easily solved by static bucketing with the help of Hirschberg’s scan (see Chapter 4). Essentially, given a sequence $I$ of $m$ elements in sorted order (one of the intervals of a pair of symmetric intervals), we apply the searching algorithm only to the sub-sequence composed by the $b$-spaced elements, where $b = O(\text{polylog} n)$, that is the sub-sequence $I_b = I[1]I[b + 2]I[2b + 3]…$ so that we have a sufficient number of stolen bits. Each contiguous sub-sequence of $b = O(\text{polylog} n)$ elements placed between two $b$-spaced elements, is organized in a static $O(\log m)$-ary complete tree that can be easily searched with a constant number of executions of the mentioned Hirschberg’s scan.

- The second problem is in how much often the auxiliary bits of information are used. At any step of the binary search used by the algorithm, $O(1)$ values of $O(\log k)$ bits each are used. Being bounded to use stolen bits, that would translate into a searching time with a slowdown factor $O(\log k)$. To solve this problem we store only an approximation of the usual information linked to the longest common prefixes between the extremities of the intervals induced by the binary search and the median elements of these intervals (see Chapter 4 for the details on Manber and Myers’ searching algorithm). We divide each element in $O(\log n)$ parts of $O(k/\log n)$ contiguous components each. The longest common prefix information we maintain will be concerning those macro-components of $O(k/\log n)$ basic components each, that is the length of the longest common prefix of macro-components of two elements. Each one of
those values requires $O(\log \log n)$ stolen bits if encoded in binary, but instead we will encode it in a tabular form requiring $O(\log n)$ slots of two stolen bits each:

- the $i$th slot encode a 0 if the longest common prefix ends in a macro-components with index $j > i$;
- the $i$th slot encode a 1 if the longest common prefix ends in the $i$th macro-component;
- the $i$th slot encode a 2 if the longest common prefix ends in a macro-components with index $j < i$.

That particular way to store the approximate information is fundamental: since we may have to access the longest common prefix information at any step of the binary search, if we used the usual, binary way to store it, we would have to pay $O(\log \log n)$ comparisons for any step of the binary search leading to a sub-optimal searching algorithm.

10.3 Permuting for Encoding Bits of Information

In this section we provide a novel tool for encoding and decoding bits with elements of unbounded length $k$. What is easily done in the previous chapters or, for example, in the implicit data organizations presented in the previous literature [Munro, 1986], namely, using a pair of distinct elements to encode the value of a bit (the bit stealing technique see Chapter 4), is non-trivial in our case. There are two reasons for that:

- First, the elements are not necessarily distinct under our model (unlike in [Munro, 1986]).
- Second, decoding just one bit requires an $O(k)$-time comparison of two elements when done naively.

Starting from the sorted sequence of $r$ elements in non-decreasing order, we show how to encode bits according to a new scheme, which we call a ditch. We obtain two incarnations: small ditches, obtained by permuting $r = O(\log n)$ elements, and large ditches by permuting $r = O(n)$ elements.

We recall that the $n$ input elements are maintained in $\mathcal{A}$, which is conceptually a $k \times n$ matrix to be permuted column-wise, and initially sorted in non-decreasing order. Here, the columns are the $n$ elements, where $a_j$ is the element in the $j$th column of $\mathcal{A}$. The $i$th symbol $a_j[i]$ in that element is in entry $\mathcal{A}[i, j]$. Hence, the $i$th row in the matrix contains the $i$th component of all the elements. For any two elements $x$ and $y$, we introduce the notation

$$lcp(x, y) = \max (\{0\} \cup \{1 \leq \ell \leq k : x[1\ldots\ell] = y[1\ldots\ell]\})$$
to denote the length of their longest common prefix.

We first describe the ditch in Section 10.3.1 and some properties common to small and large ditches. As we will see, in order to be efficiently used to encode information, large ditches need to be described by some amount of auxiliary information (of course far less than the amount of information a large ditch can encode, otherwise there will be no advantage in using large ditches). In Section 10.3.2 we will show how to encode that information. We describe how to preprocess $A$ by encoding a large ditch in it.

### 10.3.1 The ditch: A basic tool for encoding bits

We start out from a sequence of $r$ elements, $x_1, \ldots, x_r$, initially in sorted order. We say that $i$ and $j$ are twin positions (for $1 \leq i < j \leq r$) if and only if

$$i + j = r + 1$$

that is, the number of elements up to position $i$ equals that of elements from position $j$ to the end. Let’s “dig” into the sequence by comparing incrementally the elements at twin positions. We trace this process with an integer $d$, called digging depth, according to the procedure below:

**Digging ($x_1, \ldots, x_r$):**

1: $d \leftarrow 1, i \leftarrow 1, j \leftarrow r$
2: WHILE $i < j$ DO
3: WHILE $d \leq k$ AND $x_i[d] = x_j[d]$ DO
4: \hspace{1cm} $d \leftarrow d + 1$
5: \hspace{1cm} $i \leftarrow i + 1, j \leftarrow j - 1$

The pseudocode determines a ditch, illustrated by the example in Figure 10.1, which we characterize formally by its useful properties. The relevant one is that of encoding bits so that they can be quickly decoded. We first note an important property for Digging.

**Lemma 10.1** The inner loop of procedure Digging computes $d = \text{lcp}(x_i, x_j) + 1$ for twin positions $i$ and $j$. The total cost of Digging for $r$ sorted elements of length $k$ is $O(k + r)$ time.

**Proof:** Since $\text{lcp}(x_{i'}, x_{j'}) \leq \text{lcp}(x_i, x_j)$ for $i' < i < j < j'$, the digging depth for $i$ and $j$ is surely at least $d$, before the execution of the inner loop. After that loop, we consequently have $d = \text{lcp}(x_i, x_j) + 1$. The monotonicity of the digging depths accounts for the total cost of digging. Hence, the incremental cost of comparing elements in twin positions $i = i' + 1$ and $j = j' - 1$, after having done it with $i'$ and $j'$, is proportional to

$$\text{lcp}(x_i, x_j) - \text{lcp}(x_{i'}, x_{j'}) + O(1).$$
The total cost is a telescopic sum that evaluates to the maximum \( \text{lcp} \) value, \( O(k) \), plus the number \( r \) of elements in the sequence. This gives the final bound on the cost of digging. 

Since the elements involved in digging are lexicographically sorted, we can easily verify that the following holds.

**Lemma 10.2** For any pair of twin positions \( i \) and \( j \) with \( i < j \), if the digging depth satisfies \( d \leq k \), then \( x_i[d] < x_j[d] \).

We exploit Lemma 10.2 for encoding and decoding bits. We adopt a simple rule for encoding a bit with a pair of twin positions \( i \) and \( j \): swap elements \( x_i \) and \( x_j \) to encode 1, so that \( x_j \) is in position \( i \) and \( x_i \) is in position \( j \); otherwise, leave \( x_i \) and \( x_j \) at their own positions \( i \) and \( j \), respectively, thus encoding 0.

Decoding is simple and takes constant time for any given pair of twin positions \( i \) and \( j \), when their digging depth \( d \) is known. Namely, let \( z_i \) and \( z_j \) be the elements in these positions (note that either \( z_i = x_i \) and \( z_j = x_j \), or \( z_i = x_j \) and \( z_j = x_i \)). We decode the \( i \)th bit by simply comparing \( z_i[d] \) and \( z_j[d] \): that bit is 1 if and only if \( z_i[d] > z_j[d] \), as a consequence of Lemma 10.2.

Of course, as long as we know what pair of components can be probed for difference the process of stealing bit from multidimensional elements is no more difficult than the analogous for the case of elements with constant length. The problems arise when we have to decode a batch of stolen bits and, of course, we do not know the digging depth for any pair of distinct elements.

We can extend the property above to \( b \) contiguous pairs of twin positions simply inverting the digging process. We say that two pairs \( i, j \) and \( i', j' \) are contiguous (where \( i' < i < j < j' \)) if \( i = i' + 1 \) and \( j = j' - 1 \). If we do not know their digging depths, we cannot apply directly the constant-time decoding of individual bits described above. We can circumvent this drawback as stated below.

**Lemma 10.3** We can decode the \( b \) bits encoded by \( b \) consecutive pairs of twin positions in \( O(b + k) \) time.

**Proof:** Decoding all the \( b \) bits is like digging. What changes in decoding is that the elements are pairwise permuted in twin positions. Interestingly the inner components of the ditch do not change. So the matched components in the inner while loop at lines 3–4 of procedure DIGGING are likewise matched in decoding. As a result, decoding computes on the fly all the digging depths \( d \) of the first \( b \) twin positions. A mismatch causing a loop exit might find that the elements in the current twin positions are swapped for encoding purposes. Since we know \( d \) at this point, we can decode the current bit by looking at the mismatch according to Lemma 10.2. The analysis for the time complexity of decoding goes along the same lines as that of Lemma 10.1.
Figure 10.1: A ditch (light gray) encoding all 0s; hence, no two elements in twin positions are swapped.

We remark that our use of ditches will be twofold. For a small ditch, with \( r = O(\log n) \), we will scan all the \( r \) elements. For a large ditch, with \( r = O(n) \), we need to encode the positional information of the ditch's components to avoid a full scan of the \( r \) elements.

### 10.3.2 Encoding large ditches

In this section, we focus on large ditches where the \( r = O(n) \) sorted elements are to be permuted suitably. Consequently, we have to encode the information representing a large ditch itself, and define the structure deriving from the digging process. Pictorially, the ditch is the concatenation of two specular stairs, one descending and one ascending, as shown in Figure 10.1.

The components implicitly removed from a row during the digging (the elements in the white locations in Figure 10.1), are equal to the homologous components of the median element. Let's fix any row \( d \) such that \( d \) is one of the digging depths computed by procedure Digging. There can be further components (not shown in the figure) outside the ditch in row \( d \) that are equal to the median's; however, they can only extend beyond one side of the ditch rather than both sides (otherwise the ditch would be larger).

As noted in the proof of Lemma 10.1, the digging depth \( d \) is monotonically non-decreasing. Thus we can split the sequence of digging depths into twin intervals, as shown in Figure 10.1. They collect all the twin positions giving rise to the same digging depth \( d \). These positions are contiguous and form two twin intervals \( \mathcal{T}', \mathcal{T}'' \), with \( |\mathcal{T}'| = |\mathcal{T}''| \), whose digging depth is \( d \). Using the observation above, we can scan the twin intervals from left to right and number them for reference purposes. Note that we cannot have more than \( 2k \) twin intervals.

Using the features above, we represent a ditch by recording somewhere the po-
sitional information of each pair of twin intervals in it, say $\mathcal{T}', \mathcal{T}'':$

- The leftmost and rightmost position in $\mathcal{T}'$, in $O(\log n)$ bits. We can infer that of $\mathcal{T}''$ from $\mathcal{T}'$ as they are specular with respect to the median position in the ditch.

- Their digging depth $d$, in $O(\log k)$ bits. However, as we will see, we do not need to remember $d$. The search procedure that we will define has two main phases and the first one is devoted to find the only twin interval in which the search should continue with the second phase. Since we can compute the value of one longest common prefix staying within our target bound, the digging depth of the twin interval in which the search will continue can be recovered in $O(k)$ time by computing $d = \text{lcp}(x_{i'}, x_{i''}) + 1$ on the fly, for any two elements with $i' \in \mathcal{T}'$ and $i'' \in \mathcal{T}''$.

We have to encode just $O(\log n)$ bits per pair of twin intervals. Since there are $O(k)$ of them in a ditch, we need to encode a total of $O(k \log n)$ bits to represent a ditch implicitly. In Section 10.3.3 we will see how these bits are encoded.

We have seen so far how to encode bits in small and large ditches. As we will see, the permutation for space and comparison optimal search over a multidimensional domain is composed by:

- $O(k)$ small ditches of $O(\log n)$ elements each.

- One large ditch with $O(n)$ elements.

The ditch is not yet able to route the search optimally towards a twin interval. Another goal of the ditch is indeed reducing the problem of searching an element in $\mathcal{A}$ to that of searching among the elements of one selected twin interval. For this, we will need to single out the leftmost element in $\mathcal{T}$, for each twin interval $\mathcal{T}$ from left to right, but we will see more about that in Section 10.3.3, where we give the details of the preprocessing of the elements.

It is worth noting that searching inside a twin interval is easier than the original search in $\mathcal{A}$ as we can rely on the implicitly encoded bits in the interval. Indeed, each pair of twin intervals $\mathcal{T}', \mathcal{T}''$ encodes $b$ bits, where $b = |\mathcal{T}'| = |\mathcal{T}''|$. Hence, we can assign $b/2$ bits to each twin interval; say, the bits encoded by odd positions to $\mathcal{T}'$ and those encoded by even positions to $\mathcal{T}''$. These bits will be used with a variation of the Manber and Myers’ search requiring less bits of auxiliary information than the original one. We defer the details on how to exploit this to Section 10.4.

### 10.3.3 Preprocessing array $\mathcal{A}$

We now have a better picture of how to permute the elements by preprocessing the input array $\mathcal{A}$. We recall that it is a two-dimensional $k \times n$ matrix of elements
initially sorted in lexicographic order. We divide \( \mathcal{A} \) into four zones with the (exotic) name of zones A, B, C and D.

Zones A and B contain overall \( O(k \log n) \) distinct elements from \( \mathcal{A} \) to form the large ditch (in zones C and D).

- If the number of distinct elements in \( \mathcal{A} \) is smaller than that, we are in a special case which we can easily handle using the algorithm given in [Andersson, Hagerup, Håstad, and Petersson, 2001], for example.

- Otherwise, starting from the sorted sequence of these distinct elements:

  1. We divide them into \( k \) blocks of \( \Theta(\log n) \) elements each.
  2. We move the leftmost element of each block, in left-to-right order, to zone A.
  3. We thus form a small directory of \( O(k) \) elements for identifying a block in zone B, which hosts the rest of these elements.

The \( i \)th block implicitly encodes \( O(\log n) \) bits with a small ditch. The encoded integers are the positional information for the \( i \)th pair of twin intervals in the large ditch (stored in zones C and D). We use Lemma 10.3 for the small ditch in the block, with \( b = O(\log n) \). Hence, decoding the information for a pair of twin intervals requires \( O(k + \log n) \) time by accessing zone B.

The purpose of zone A is to make zone B searchable. (Alternatively, we can employ techniques from [Andersson, Hagerup, Håstad, and Petersson, 2001] to combine zones A and B.)

Zones C and D contain the rest of the elements and properly form the large ditch. We move the leftmost element from each twin interval of the ditch to zone C (note that the elements in this zone are in lexicographic order). The remaining elements remain in zone D (they are still sorted at this stage but will be permuted later on). The \( O(k) \) elements in zone C constitute a small search directory to locate a twin interval in zone D. The result of the preprocessing described in this section is pictorially synthesized in Figure 10.2.

We refer the reader to Section 10.4 for the preprocessing within a twin interval and the adopted searching method.

### 10.4 Searching with Few Bits of Information

We now can give a description of our search, which uses Hirschberg’s search [Hirschberg, 1978] (see Chapter 4) as a basic routine. The latter takes \( O(k + r) \) time for \( r \) elements and we refer to it as scanning search. If the \( r \) elements encode bits with our digging methods, it’s not difficult to combine our on-the-fly decoding based on digging and the scanning search, attaining \( O(k + r) \) total time. This is an immediate corollary to Lemma 10.3.
10.4. SEARCHING WITH FEW BITS OF INFORMATION

Figure 10.2: The array \( T \) after the preprocessing process we described in Section 10.3.

### 10.4.1 Restricting the range for searching

Let us show how the search for an element in the input set can be reduced to a search into one of the twin intervals defined in Section 10.3.

1. Given a search element \( x \), we first check whether \( x \) occurs in zones A or B. For this:

   (a) We apply the scanning search to the \( r = O(k) \) elements that form the directory in zone A, in \( O(k) \) time.

   (b) We then identify a block in zone B with \( r = \Theta(\log n) \) elements.

   (c) We apply here the (combined) scanning search in \( O(k + \log n) \) time.

2. If we do not find the element, we search it in the large ditch represented by zones C and D.

   (a) We check if the search element equals the median element in the ditch in \( O(k) \) time.

   (b) If not, we proceed to search in the directory of zone C. Again, we run the scanning search on the \( r = O(k) \) elements in that zone, in \( O(k) \) time.

3. If the element is not in zone C, we end up in one position, say, that of the \( i \)th element in zone C.

   (a) We identify the unique twin interval containing \( x \) in zone D (if any; if \( x \) occurs more than once, we take its leftmost occurrence). Indeed, we can infer that we have to access the \( i \)th twin interval, say \( T_i \).

   (b) We go to zone B and decode the positional information of \( T_i \) by scanning \( b = O(\log n) \) elements in the corresponding block of that zone, in \( O(k + \log n) \) time by Lemma 10.3.
(c) Afterward we compute the digging depth of $\mathcal{T}$ in $O(k)$ time. We recall that this depth, say $d$, can be computed as $d = \text{lcp}(y, z) + 1$ for any chosen pair of elements $y$ in $\mathcal{T}$ and $z$ in $\mathcal{T}$'s twin interval.

As a result we have reduced, in $O(k + \log n)$ time, the problem of searching element $x$ in the array $\mathcal{A}$ to the problem of searching $x$ from a twin interval $\mathcal{T}$ of zone D. We therefore focus on such interval $\mathcal{T}$ for preprocessing and searching by exploiting the fact that we can encode implicitly up to $b/2$ bits in it, where $b = |\mathcal{T}|$.

**Lemma 10.4** We can permute the elements in the input array $\mathcal{A}$ to form a ditch so that searching in $\mathcal{A}$ reduces, in $O(k + \log n)$ time, to searching in one of the twin intervals in the ditch.

### 10.4.2 Searching within a twin interval

We describe how to preprocess and search a twin interval $\mathcal{T}$, assuming that we can encode up to $b/2$ bits by permuting elements, where $b = |\mathcal{T}|$. If $|\mathcal{T}| = O(\log n)$, we run the (combined) scanning search in $O(k + \log n)$ and we are done. We assume hereafter that $|\mathcal{T}| = \Omega(\log n)$.

We would like to use the search of Manber and Myers [Manber and Myers, 1993], which we refer to as MM-search. Originally appeared in the field of full text indexing with the suffix array structure, it can be used also for arbitrary elements from a multidimensional domain in lexicographic order. However, MM-search and its variations devised thereafter access too many bits. We propose a variant that is tailored for our twin intervals in that it is less demanding in the number of bits to be decoded.

We first review in a nutshell how MM-search works for a set of $r$ elements of length $s$ each. It performs a variation of the classical binary search according to three cases. For the current search interval, $[L \ldots R]$, we denote the middle point by $M$. The invariant maintains the $\text{lcp}$ value between the search element $x$ and the elements $x_L, x_R$ at positions $L$ and $R$, respectively. We know by induction that $x_L \leq x \leq x_R$. We want to infer the outcome of the comparison of $x$ versus $x_M$ (the element in position $M$) while preserving the invariant on searching into a smaller interval, either $[L \ldots M]$ or $[M \ldots R]$. For the sake of discussion, suppose that $\text{lcp}(x_L, x) \geq \text{lcp}(x, x_R)$, and let $m = \text{lcp}(x_L, x)$ be the number of initial symbols in $x$ that have been successfully matched so far.

1. Case $m < \text{lcp}(x_L, x_M)$. Set $L = M$.

2. Case $m = \text{lcp}(x_L, x_M)$. Compute $\text{lcp}(x, x_M)$ by comparing $x$ and $x_M$ from position $m + 1$ on. Set $m = \text{lcp}(x, x_M)$ thus computed. Access symbols in positions $m + 1$ (if any) of $x$ and $x_M$. If $x[m + 1] > x_M[m + 1]$, set $L = M$; else, set $R = M$. 
3. Case $m > \text{lcp}(x_L, x_M)$. Set $R = M$.

In order to maintain its invariant, MM-search requires to maintain the values of $\text{lcp}(x_L, x)$ and $\text{lcp}(x, x_R)$ after dealing with each of the three cases. The first two cases are not a problem. In the first case, $\text{lcp}(x_L, x) = \text{lcp}(x, x_M)$, which is also the current value of $m$. In the second case, $\text{lcp}(x_L, x)$ or $\text{lcp}(x, x_R)$ has been just computed as well since it becomes equal to $m$. However, in the third case, MM-search needs $\text{lcp}(x_L, x_M)$ since it becomes the new value of $\text{lcp}(x, x_R)$. This requires reading $O(\log s)$ bits somewhere would make our algorithm sub-optimal because of the fact that we are bound to use the stolen bits, requiring $O(1)$ comparisons in order to decode each one of them.

We make MM-search less demanding by introducing two crucial modifications:

- We encode the required $\text{lcp}$ information in unary using a ternary string. For a non-negative integer $\ell \leq s$ (i.e., an $\text{lcp}$ value), we define its “unary-ternary” representation as $a_0, a_1, \ldots, a_s$, where $a_0 = \cdots = a_{s-1} = 0$, $a_s = 1$, and $a_{s+1} = \cdots = a_{\ell} = 2$. Each such digit can be encoded by two bits. The clear advantage is that comparing an integer $g$ to $\ell$, where $0 \leq g \leq s$, just requires decoding an individual digit, $a_g$. The price to pay is that we require now $O(s)$ bits to represent an $\text{lcp}$ value, instead of $O(\log s)$ bits.

- We change the invariant maintained by the MM-search on interval $[L \ldots R]$. Since it requires the value of $\text{lcp}(x_L, x_M)$ to handle the third case, we have to decode the value of $\text{lcp}(x_L, x_M)$ from its “unary-ternary” representation. This is equivalent to identifying the digit $a_{\text{lcp}(x_L, x_M)} = 1$ by accessing $\Omega(\log s)$ bits from that representation. Instead, we make a simple but effective observation. We keep only $m$, which is the $\text{lcp}$ value between the search element $x$ and the element in $\{x_L, x_R\}$ that maximizes that $\text{lcp}$, say $x_L$. In other words, the value of $\text{lcp}(x, x_R)$ in the third case above is not actually functional to the search as it is smaller than $m$.

Summarizing, we store an approximation of the original longest common prefix information needed by Manber and Myers’ algorithm. We only need the “unary-ternary” representation $a_0, a_1, \ldots, a_s$ of $\text{lcp}(x_L, x_M)$, without explicitly computing the actual value of $\text{lcp}(x_L, x_M)$. We can rephrase the three cases of MM-search by examining $a_m$ according to our modifications.

1. Case $a_m = 0$: Inferring that $m < \text{lcp}(x_L, x_M)$, set $L = M$ and do not change $m$ (since $x_L$ still maximizes the $\text{lcp}$ value).

2. Case $a_m = 1$: Inferring that $m = \text{lcp}(x_L, x_M)$, proceed as in the original MM-search. The new value of $m = \text{lcp}(x, x_M)$ indicates that $x_M$ is the best match (which becomes either $x_L$ or $x_R$ in the next search step, as previously mentioned).
3. Case \( a_m = 2 \): Inferring that \( m > \text{lcp}(x_L, x_M) \), set \( R = M \) and do not change \( m \) (since \( x_L \) still maximizes the \( \text{lcp} \) value).

It should be noticed that we do not need the actual value of \( \text{lcp}(x_L, x_M) \), but only the outcome of its comparison with \( m \) (i.e., \( a_m \), which are two bits). Moreover, when \( m \neq \text{lcp}(x_L, x_M) \), the value of \( m \) is unchanged and \( x_L \) still maximizes the \( \text{lcp} \) value. Therefore, we can avoid to decode the entire value of \( \text{lcp}(x_L, x_M) \).

**Lemma 10.5** We can modify MM-search for \( r \) elements of length \( s \) so as to access \( O(1) \) bits of \( \text{lcp} \) values per step during the binary search. We represent each \( \text{lcp} \) value by \( O(s) \) bits to work, for a total of \( O(rs) \) bits storing the \( \text{lcp} \) information. The resulting MM-search requires a total of \( O(\log r) \) bits of the latter kind, and \( O(s + \log r) \) component comparisons.

**Proof:** As previously mentioned, we store the \( \text{lcp} \) information needed by MM-search in “unary-ternary”, thus requiring no more than \( O(s) \) bits per \( \text{lcp} \) value. Since the number of the \( \text{lcp} \) values is bounded by the number of elements, this yields a total of \( O(rs) \) bits. Our modification of MM-search accesses just \( O(1) \) of these bits for each step. Since there are \( O(\log r) \) searching steps, this gives a total of \( O(\log r) \) accessed bits. Note that the number of component comparisons remains unchanged, \( O(s + \log r) \), as in the original MM-search. \( \square \)

We are now ready to describe the preprocessing of a twin interval \( \mathcal{T} \) of length \( b = |\mathcal{T}| \).

1. We first partition \( \mathcal{T} \) into blocks of size \( \Theta(\log b) \).

2. We take the leftmost element in each block (from left to right) as the *leading* element of the block.

So, we have overall \( O(b/\log b) \) leading elements in \( \mathcal{T} \). Finding an element inside a block can be done with the (combined) scanning search, in \( O(k + \log n) \) time (since \( b \leq n \)).

In order to identify the suitable block of \( \mathcal{T} \), we first search the leading elements. We consider each leading element, of length \( k \), as composed by \( \log b \) macro-components. Each macro-component is made up of \( \Theta(1 + k/\log b) \)-sized piece of the original element. So the comparison of any two macro-components requires \( O(1 + k/\log b) \) time. We encode the \( s = O(\log b) \) bits needed by the modified MM-search in Lemma 10.5 for each leading element, in its corresponding block of \( \mathcal{T} \).

We recall that we measure the cost by accounting interchangeably for the number of comparisons and component probes, as they are linearly proportional to each other in our case.
Theorem 10.1 Given a sequence of \( n \) elements of length \( k \) each, in lexicographic order, we can preprocess them by permuting in \( O(nk) \) time, so that each search has a cost of \( \Theta(k + \log n) \), which is optimal.

Proof: We have shown so far how to reduce the search in \( \mathcal{A} \) to the search in a twin interval, in \( O(k + \log n) \) time. We search among the leading elements of the twin interval. By Lemma 10.5, since \( r = O(b/\log b) \) and \( s = \log b \), we can perform a search with \( O(\log b) \) macro-component comparisons, decoding \( O(\log b) \) bits. This gives a total of \( O(k + \log b) \) time as each macro-component comparison requires \( O(1 + k/\log b) \) time. We then search inside a block in \( O(k + \log b) \) time. We get the final bound since \( b \leq n \).

10.4.3 Searching in the bit probe model

As a corollary of Theorem 10.1, we obtain an optimal search bound in the bit probe model [Elias and Flower, 1975, Yao, 1984]. In this model, the elements are distinct binary strings and the complexity accounts for the number of bits probed in the array \( \mathcal{A} \) storing them. Array \( \mathcal{A} \) is therefore a \( k \times n \) binary matrix, where we store the elements permuted, so that membership query requires \( \Theta(k) \) bit probes in the worst case, which is optimal. The occupied space is that of the elements, namely, \( nk \) bits.

Corollary 10.1 In the bit probe model, \( n \) elements of \( k \) bits each can be permuted in \( \mathcal{A} \), so that they occupy just \( nk \) bits and searching an element probes \( O(k) \) bits in the worst case, which is optimal.

Proof: If \( k > 1/2 \log n \), we use Theorem 10.1 adapted to the bit probe model. Otherwise, we access one of the all \( O(\sqrt{n}) \) possible answers that we encode in the permutation itself. This requires decoding \( O(1) \) bits, and so accessing \( O(1) \) elements.

10.5 Conclusions

In this chapter, we have shown that a sorted array is not the best data organization for space and comparison optimal search when the input elements are drawn from a multidimensional domain. Our result sheds a further light on the relation between searching and sorting from a theoretical point of view. We have described a new organization of \( k \)-dimensional elements that is based upon a suitable permutation of them, and that allows us to search optimally in \( O(k + \log n) \) time. These elements in sorted order cannot be optimally searched as a consequence of the results in [Andersson, Hagerup, Håstad, and Petersson, 2001] and in [Franceschini and Grossi, 2004b] (FOCS 2004, on which the presentation in this chapter is based). Nevertheless, we still like to consult dictionaries of alphabetized words!
Chapter 11

The “Implicitous” B-Tree

Abstract

An implicit data structure for the dictionary problem maintains \( n \) data values in the first \( n \) locations of an array in such a way that it efficiently supports the operations insert, delete and search. No information other than that in \( O(1) \) auxiliary locations and in the input data is to be retained. The only operations allowed on the data values (other than reads and writes) are comparisons. In this chapter we describe the Implicit B-tree, a data structure for the External-Memory model (see Chapter 2) supporting the three operations in \( O(\log_B n) \) block transfers like in regular “ubiquitous” B-trees (see [Comer, 1979]), from which the awesome joke in the title derives), under the realistic assumption that a block stores \( B = \Omega(\log n) \) elements. Moreover, reporting \( r \) consecutive elements in sorted order has a cost of \( O(\log_B n + r/B) \) block transfers. En route a number of space efficient techniques for handling segments of a large array in a memory hierarchy are developed. Being implicit, the proposed data structure occupies exactly \( [n/B] \) blocks of memory after each update, where \( n \) is the number of elements after each update and \( B \) is the number of elements contained in a memory block. In main memory, the time complexity of the operations is \( O(\log^2 n / \log \log n) \), disproving a conjecture of the mid 1980’s (see [Munro, 1986]).

The presentation in this Chapter is based on the papers [Franceschini, Grossi, Munro, and Pagli, 2002] (FOCS 2002) and [Franceschini, Grossi, Munro, and Pagli, 2004] (Journal of Computer and System Sciences 68).

11.1 Introduction

Data storage capacity is increasing rapidly, and doubles every 18 months according to Moore’s law. Processor speed is increasing even faster, so one can deploy extra computing power to squeeze more data on storage devices [Witten, Moffat, and Bell, 1999]. Minimizing storage costs for large processes is as important as ever, indeed an
absolutely crucial issue is to minimize memory use in a manner such that retrievals from the "next level" in the memory hierarchy are minimized.

In this chapter, we consider a fundamental problem of data organization in a fundamental way. In particular we re-examine the problem of designing data structures for dynamic search that achieve the best space saving possible, that is, they occupy $O(1)$ memory locations other than those strictly needed for the distinct elements and the problem size. We want to insert, delete and search elements in a dictionary, with all the auxiliary information (including small integers and internal pointers) being implicitly encoded by a suitable permutation of the elements.

In other words, if we take a snapshot of the memory, we just see the $n$ distinct elements permuted in $n$ memory locations, assuming that each element occupies a single location for the sake of presentation. After each update, a new permutation is obtained to fully encode the resulting implicit data structure at no extra memory cost. This class of data structures are known in the literature as implicit [Munro and Suwanda, 1980] (see Chapter 3), and Williams' heap [Williams, 1964] is undoubtedly the best known example. There are many advantages to this organization of the elements:

- Compressing the sequence of permuted elements automatically compresses also their associated data structure. Compression of database records is an important consideration [Chen, 2002].

- Transmitting over a channel or dumping for backup the sequence of permuted elements provides indirectly the service also for their associated data structure at no extra cost. This property can be useful to download indexed data in mobile and personal computing.

- There is no replication in the data structure, with low storage costs and no need to maintain consistency among multiple copies of the elements. Scanning the elements is very efficient.

Implicit data structures have been studied to investigate the power of pointers. While the term originated in studying precisely the problem we now investigate [Munro and Suwanda, 1980], it has also been the subject of papers taking a somewhat different point of view, including the seminal paper of Yao [Yao, 1984], a long list of results on the related topic of perfect hashing [Fredman, Komlós, and Szemerédi, 1984] and the dictionary problem over a bounded universe [Brodinik and Munro, 1999, Pagh, 2002, Raman, Raman, and Rao, 2002].

11.1.1 The problem

We are given a set of $n$ elements from an ordered, but unbounded universe. These elements are considered to be atomic, and we may only perform comparisons and moves (see Chapter 2). Under such a model, it is immediately obvious that $\log_2 n$
comparisons are necessary and sufficient to perform a search. In that sense the sorted array is the most basic “nontrivial implicit data structure”. It handles searches optimally and uses no auxiliary storage for pointers etc., but it is static.

The heap of Williams [Williams, 1964] is the archetypal implicit data structure. It supports a priority queue (operations insert and extract maximum) in the first \( n \) locations of an array in \( O(n \log n) \) time. Furthermore, no other “structural information” is required, other than the ability of the array to be perceived as growing or shrinking with \( n \).

The first work on an implicit data structure for the dictionary problem, and as noted the term, dates from the late 1970’s ([Munro and Suwanda, 1980], publication is far from instantaneous) with an \( O(n^{1/3} \log n) \) bound. Frederickson [Frederickson, 1983] generalized an aspect of this approach to achieve a \( o(n^e) \) bound. Munro [Munro, 1986] improved this to \( O((\log^2 n) \) with a method based on pointer encoding and AVL trees [Adelson-Vel’skii and Landis, 1962]. Indeed the paper speculates that \( \Theta((\log^2 n) \) may be optimal (we will disprove that long-standing conjecture with a variant of the structure we are going to describe in this chapter).

Borodin et al. [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988] and Radhakrishnan and Raman [Radhakrishnan and Raman, 2001] gave an interesting tradeoff between data moves in performing an update and the number of comparisons necessary for a search. The bound does not, however, rule out \( O(\log n) \) behavior for the problem.

Closely related to our problem is that of arranging \( n \) \( k \)-field records into an \( n \) by \( k \) array so that searches can be performed quickly given any field value. The static two field problem bears some superficial similarity to the dynamic single element dictionary problem. Nevertheless, searches under this model can be supported in \( O(\log n) \) time [Fiat, Munro, Naor, Schäffer, Schmidt, and Siegel, 1991], where the “\( O \)” does include a factor depending on \( k \).

\subsection{The implicit B-Tree}

\textbf{The theoretical result.} In this chapter, we describe the implicit B-tree in the context of a memory hierarchy. Our main result is showing that the worst-case cost of searching and updating in the External-Memory model (see Chapter 2) is

\[ O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n \right) \text{ I/Os} \]

where each I/O operation is block transfers of size \( B \).

Moreover, reporting \( r \) consecutive elements in sorted order is output sensitive, namely, it takes \( O(r/B) \) block transfers plus the search cost, which is \( O\left(\left\lceil (\log n)/B \right\rceil \log_B n \right) \) in our case. These poly-logarithmic costs reduce to \( O(\log_B n) \) like in regular B-trees, under the realistic assumption that a block stores \( B = \Omega(\log n) \) elements. Being implicit, our data structure occupies exactly \( \lceil n/B \rceil \) blocks of memory after each update.
An important byproduct of implicit B-trees is that they disprove a long-standing conjecture in implicit data structures for the dictionary problem [Munro, 1986], and improve our basic understanding of a theory of data ordering. It was conjectured that $\Omega(\log^2 n)$ time is needed for supporting the operations in implicit dictionaries in main memory. Fixing $B = \Theta(\log n)$, we can use implicit B-trees in main memory for attaining $O(\log^2 n / \log \log n)$ time.

The presentation in this chapter is based on the papers [Franceschini, Grossi, Munro, and Pagli, 2002] (FOCS 2002) and [Franceschini, Grossi, Munro, and Pagli, 2004] (Journal of Computer and System Sciences 68).

What about practical applications? While we do not claim the method, as stated, is easily implementable, preliminary experimental results show that the data structure could be useful in practice [Bello and Fantoni, 2003]. Compared to the implementation of regular B-trees, such as those in [Sedgewick, 1992] and in Berkley DB by Sleepycat [SleepyCat Software, 2003], the constant hidden in the big-Oh notation of the time bounds for implicit B-trees is certainly larger, as expected. However, these implementations occupy six times more space than that of implicit B-trees, and this can give a payoff in performance in several cases.

For example, `mmap` is a commonly used low-level routine of the file system to access memory mapped files faster than standard I/O routines, and it has a limitation of 4Gb or similar, in common operating systems (e.g., [The Open Group, 2003]). It comes out that implicit B-trees can handle up to 4Gb of input data alone with `mmap` while the other data structures can handle roughly up to 4Gb of both input data and auxiliary data, which translates in nearly up to 700Mb of input data alone due to their waste of space. As a result, for input data size ranging from 700Mb to 4Gb, they can be less efficient than implicit B-trees, even though their hidden constant in the time complexity is smaller.

Developments after the Implicit B-tree. As we will see in the rest of this thesis, after the Implicit B-tree (see [Franceschini, Grossi, Munro, and Pagli, 2002] and [Franceschini, Grossi, Munro, and Pagli, 2004]), there have been some developments in the field. However the Implicit B-Tree is still ahead of them in some aspects.

The first data structure is the exponential implicit tree [Franceschini and Grossi, 2003a] (SODA 2003, see Chapter 12) achieving $O(\log n \log \log n)$ time for searching and $O(\log n \log \log n)$ amortized time for updating. It exploits in-place algorithms for amortizing the update bounds and introduces different kinds of chunks of $O(\log n)$ contiguous elements to delay the expensive reorganizations. Differently from implicit B-trees, it uses an exponential tree of $O(\log \log n)$ height (except very few cases). This organization yields significantly worse bounds in memory hierarchy than implicit B-trees.

The second data structure, the Flat Implicit Tree [Franceschini and Grossi, 2003b] (ICALP 2003, see Chapter 13) is cache-oblivious and obtains the bounds
of $O(\log_B n)$ block transfers, where the cache-obliviousness of the model lies in the fact that the block transfer size $B$ is unknown to the algorithms operating in the model [Frigo, Leiserson, Prokop, and Ramachandran, 1999]. The top layer uses a van Emde Boas permutation [Frigo, Leiserson, Prokop, and Ramachandran, 1999] of the elements as a directory, and the bottom layer introduces techniques of memory layout for attaining cache-obliviousness. Compared to implicit B-trees, the cost of reporting $r$ consecutive elements in sorted order is not output sensitive in that cache-oblivious data structure as it may take $\Omega(r)$ block transfers. In general, it is an open problem to devise a cache-oblivious B-tree with output-sensitive cost [Bender, Demaine, and Farach-Colton, 2000] while allowing (amortized) bounds of $O(\log_B n)$ for any $B$, even with explicit data structures. The implicit B-tree is able to attain output-sensitive and worst-case bounds for $B = \Omega(n)$ as it is aware of the block size $B$.

The third structure [Franceschini and Grossi, 2003c] (WADS 2003, see Chapter 15) achieves the same bounds of the Flat Implicit Tree but in the worst case, closing the long-standing problem of the existence of an optimal implicit dictionary. With respect to the Implicit B-Tree, that last structure is much more complex and is not output sensitive.

### 11.1.3 Basic assumptions and organization of the chapter

We now recall the main features of the implicit model for dictionaries, in which no other “structural information” is explicitly stored other than the elements themselves that are suitably permuted for encoding indirectly the needed structural information.

An implicit data structure for the dictionary problem is a sequence of $n$ memory locations that is extendible to the right, one location at a time, and that stores $n$ distinct elements $a_1, a_2, \ldots, a_n$ suitably permuted, for any $n > 1$, where each element $a_i$ occupies a distinct location of the array. The array is allocated in a contiguous segment of $n$ adjacent memory locations, with just $O(1)$ further occupied or wasted locations. All what is known is the starting position of the memory segment hosting the array, as the rest of the information is implicitly encoded by the permutation of $a_1, a_2, \ldots, a_n$.

We adopt the basic technique of bit stealing (see Chapter 4) to encode the structural information by implicitly representing a pointer or an integer of $b = \log N$ bits by using $2b$ distinct elements, say, $y_1, z_1, y_2, z_2, \ldots, y_b, z_b$ (chosen from $a_1, a_2, \ldots, a_n$) that are pairwise permuted with the following rule: if the $i$th bit is 0, then $\min\{y_i, z_i\}$ precedes $\max\{y_i, z_i\}$; else, the bit is 1 and $\max\{y_i, z_i\}$ precedes $\min\{y_i, z_i\}$. As we saw, the immediate drawback of this technique is that the mere access of, say, a pointer takes $b$ comparisons and an update of its value requires $b$ comparisons and $O(b)$ exchanges of input elements.

In our algorithms for implicit B-trees, our search performs only read accesses to the elements. When inserting or deleting, we need to move elements around to re-encode the pointers or the integers affected by the updating. In general, inserting
a new element \( a_{n+1} \) extends the array by one location storing \( a_{n+1} \), and shuffles \( a_1, a_2, \ldots, a_n, a_{n+1} \) accordingly to encode the resulting data structure in \( n + 1 \) locations. Deleting element \( a_i \) produces an array of \( n - 1 \) locations containing a new permutation of \( a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \).

In the following, we assume without loss of generality that \( n \), the number of elements, is bounded above by \( N \), such that \( \log N = \Theta(\log n) \). This assumption is not a limitation, as shown by Frederickson [Frederickson, 1984b], but it is useful to fix the bit-length of the pointers. In fact, we can keep \( O(\log \log n) \) distinct data structures of size \( 2^c, c \leq i \leq \lceil \log \log n \rceil \) (where \( c \) is a constant), so that the total cost is dominated by the cost of the biggest of such data structures. So, our results apply regardless of changes in \( n \).

The chapter is organized as follows. In Section 11.2, we give a high level definition of our implicit B-tree and the supported dictionary operations. In Section 11.3, we embed the structure in a collection of lists, in order to limit the waste of space due to the variable size of the tree nodes. We show how to completely remove any additional space besides that of the elements in Section 11.4, where the low level memory organization is given in some detail. Finally, in Section 11.5, we discuss the refinements to obtain our final bounds.

## 11.2 High-Level Structure of the Implicit B-tree

### 11.2.1 Basic characteristics

The implicit B-tree has a high-level structure similar to that of regular B-trees; however, the internal organization of the nodes differs and indeed the node size grows with \( N \). This organization depends on two parameters,

\[
k = \Theta(\log N) = O(\log n)
\]

\[
t = \Theta(B^c)
\]

(for a positive real constant \( c < 1/2 \)). The nodes contain \( \Theta(tk) \) elements and have \( \Theta(t) \) children (as we need \( k \) elements to encode each pointer), thus resulting in a height of \( O(\log_t n) = O(\log_B n) \). Hence, we need to read \( \Theta(\log n) \) elements (with \( \Theta(\log n / B) \) block transfers) per node to traverse the tree from the root to a leaf, for a total of \( O(\lfloor \log n \rfloor \log_B n) \) block transfers.

Specifically, each leaf contains between \( tk \) and \( 2tk + k - 1 \) elements grouped in chunks of \( k \) elements each, except the last chunk in the leaf, which can have fewer than \( k \). The number of elements in a leaf can increase or decrease by 1 after an update.

Each internal node contains between \( tk \) and \( 2tk \) elements also grouped in chunks of \( k \) elements each, and the number of elements increases or decreases by \( k \). As usual, the root \( R \) can be smaller, namely, it stores between \( k \) and \( 2tk \) elements in
grouped chunks. Other internal nodes have between $t + 1$ and $2t + 1$ children, and we use a chunk of $k$ elements to separate their pointers instead of single elements. More specifically, if a node has $i + 1$ children $c_1, c_2, \ldots, c_{i+1}$, then it stores $i$ chunks $b_1, \ldots, b_i$ of $k$ elements each.

The elements in $b_j$ are all greater than those in $b_{j'}$ with $1 \leq j' < j$, and all less than those in $b_{j''}$ with $j < j'' \leq i$. The elements in $b_j$ are also greater than those in the subtree rooted at $c_j$ and less than those in the subtree rooted at $c_{j+1}$. The pointers to $c_1, \ldots, c_i$ and any other auxiliary information of the nodes are implicitly encoded by a pairwise permutation of the elements as mentioned in the introduction and detailed in Section 11.3. At this stage of description, we assume that we have a primitive to encode and decode a pointer or a small integer of $O(k)$ bits at the cost of scanning $O(k)$ elements.

If we take a snapshot of the node, we can observe that the elements are not sequentially stored in the node in the order induced by the chunks $b_1, \ldots, b_i$. Inside each chunk $b_j$ we permute pairwise the elements to encode the pointers for children $c_j$ and $c_{j+1}$ and some other information introduced later on. Furthermore, we also permute chunks; as we shall see, this is crucial to handle efficiently the node with our algorithms, especially when splitting or merging. More precisely, the $t$ smallest chunks $b_1, \ldots, b_t$ are permuted in the first $tk$ positions of the node, and the other $r = i - t$ are permuted in the remaining positions of the node. Unlike the permutation of the elements, the permutation of the chunks does not encode any bits of information but it is the mere byproduct of splitting and merging without relocation.

It is worth noting that chunks in all the nodes represent a partition of the elements into intervals. At any time, the elements in a chunk are either all greater than or all less than those in any other chunk. In other words, to merge any two consecutive chunks, it suffices to concatenate them.

### 11.2.2 Routing the search

**Directories for permuted internal nodes.** In order to efficiently route the elements during a traversal of the B-tree, we maintain at most two directories per node. The first directory occupies the first $2t$ positions of the node. Let $b_{\sigma(1)}, \ldots, b_{\sigma(t)}$ denote the chunks $b_1, \ldots, b_t$ shuffled according to the permutation $\sigma$, so that $\sigma(j) = \ell$ indicates that chunk $b_\ell$ has position $j$ in the permutation $\sigma$. The directory for these chunks is made up of the smallest and the largest element of $b_{\sigma(1)}, \ldots, b_{\sigma(t)}$, in this order.

As a result, each chunk $b_{\sigma(j)}$ has two elements moved to the directory. The directory for the permutation of the trailing $r$ chunks is analogous and occupies $2r$ positions starting at $tk + 1$. At any time, the permutation of the pairs of elements in the directories reflects that of the chunks in the nodes. The first part of the node contains the first $t$ chunks and their directory, and the second part contains the trailing $r$ chunks and their directory. This organization will be helpful when a node is split.
Routing. Routing a search for element $x$ in a node (other than the root) requires a sequential scan of the $O(t)$ elements in the directories to identify a pair of elements $x_L$ and $x_R$ such that $x_L \leq x \leq x_R$ and no other element $x'$ in the directories satisfies either $x_L < x' \leq x$ or $x \leq x' < x_R$. We have two cases:

- $x_L$ and $x_R$ belong to the same chunk $b_{\sigma(j)}$ (i.e., $x_L$ occupies an odd numbered position, $2\sigma(j) - 1$, in the directory). We route the element $x$ inside the node, indeed inside $b_{\sigma(j)}$.

- $x_L$ and $x_R$ belong to different chunks $b_{\sigma(j_L)}$ and $b_{\sigma(j_R)}$, respectively (i.e., $x_L$ occupies an even numbered position $2\sigma(j_L)$ in the directory, but $x_R$ an odd numbered position $2\sigma(j_R) - 1$). These chunks would be consecutive if not permuted according to $\sigma$, and can be located in constant time by the positions of elements $x_L$ and $x_R$ in the directories. Note that $x_L$ and $x_R$ are not necessarily consecutive in the directories, but $\sigma(j_L) + 1 = \sigma(j_R)$. So they both encode the pointer to child $c_{\sigma(j_R)}$, and routing to $x$ proceeds recursively to that child.

Our encoding scheme for the auxiliary information in the nodes is based on the pairwise permutation, and includes also the elements in the directories. This is not much of a problem, as the pairwise permutation of the elements for encoding bits and the permutation $\sigma$ in a directory do not interfere each other. To see why, it suffices to consider first the smallest and then the largest in each pair of elements in a directory. Hence, we can handle the two cases above with minor modifications.

**Lemma 11.1** Following the search path for an element in a permuted internal node can be performed with

- $O(t + k) = O(B^e + \log n)$ comparisons

- $O\left(\frac{t + k}{B}\right) = O\left(\frac{\log n}{B}\right)$ block transfers.

**Searching for an element.** The search operation for an element $x$ is rather standard. We start out from the root $R$ and, if $x \not\in R$, we route $x$ in each traversed node until we find $x$ inside a chunk or we reach the leaf, whose elements are scanned to find the element.

**Lemma 11.2** Searching for an element in the implicit B-tree takes

- $O\left(\frac{(t + k) \log n}{\log t}\right) = O((B^e + \log n) \log_B n)$ comparisons

- $O\left(\left\lceil\frac{\log n}{B}\right\rceil \log_B n\right)$ block transfers.
11.2. HIGH-LEVEL STRUCTURE OF THE IMPLICIT B-TREE

Proof: The number of visited nodes is $O(\log n / \log t) = O(\log_B n)$ and, for each node, we pay $O(t + k)$ comparisons and a cost of $O((t + k)/B) = O(\lfloor \log n / B \rfloor)$ block transfers to route the element by Lemma 11.1, and of $O(k/B) = O(\lfloor \log n / B \rfloor)$ block transfers to decode the pointer to one of its children, as $t = \Theta(B^r)$ and $k = O(\log n)$. \qed

11.2.3 Inserting an element

Let us now examine the insert operation for a new element $x$. We run the search for $x$ as described above. Once we find the position of $x$ inside chunk $b_j$ of a node, say $u$, we make room for $x$ in $b_j$ according to two cases:

- If $u$ is a leaf, we shift by one position to the right the elements in $b_j$ following $x$, as well as the elements in the following chunks $b_{j+1}, b_{j+2}, \ldots$, so that we have one position free in $u$ to store $x$. The size of the leaf $u$ increases by 1.

- If $u$ is an internal node, we shift by one position to the right the elements in $b_j$ following $x$. The rightmost shifted element replaces the rightmost element of $b_j$ in the directory. The latter element is then inserted in the leftmost leaf of the subtree rooted at $c_{j+1}$, since the chunks of the B-tree are disjoint intervals as previously noted. The leftmost leaf is reached by a downward traversal of the subtree.

In both cases, we end up in a leaf whose number of elements increases by 1. No internal node has yet changed size. However, after the insertion, if the leaf involved becomes of size $2tk + k$, we have to split into two leaves storing $t$ chunks each, respectively, $b_1, \ldots, b_t$ and $b_{t+2}, \ldots, b_{2t+1}$. Then, we have to insert the median chunk $b_{t+1}$ into the leaf’s parent (which we know because it is on the root-to-leaf path of the insertion, see below), updating two pointers in the parent to the two leaves just created. We encode pointers to the nodes in the root-to-leaf path into the reserved area during the descending phase of the insertion. In case of rebalancing, when we need to access the path in reverse order, we simply decode the pointers encoded in the reserved area.

Rebalancing. At this point we may trigger a sequence of split operations on the internal nodes along the root-to-leaf path traversed during the insertion. Here is how we handle it.

Let us consider the general case of an internal node $u$ of maximal size $2tk$, in which we have to insert the median chunk $b$ originating from the split of its child, say $c_j$, into two nodes $c_j'$ and $c_j''$. Assume without loss of generality that $j \leq t$, so that we have to update the first half of node $u$ and its directory.

1. We first identify the position $i$ of the chunk $b_i$ in the permutation (with $\sigma(i) = t$) containing the largest pair of elements in the directory of $u$. 


2. We move $b_t$ in a temporary area (see below for details) and store $b$ in its place, putting the smallest and the largest element of $b$ in the two directory positions left free by the homologous elements in $b_t$.

3. We then access the $O(k)$ elements in $u$ encoding the two child pointers associated with $b_t$, and permute these elements pairwise to encode the pointers to $c'_j$ and $c''_j$ associated with $b$.

4. We then locate the (at most two) neighbor chunks of $b$ in $u$ by using the directory, and re-encode one of their pointers to one of the new children $c'_j$ and $c''_j$, respectively.

5. We finally record in the auxiliary information for the left half $u'$ of $u$ the fact that it is now an independent node with $tk$ elements, and do the same in the right half $u''$ of $u$. Note that we do not need to move their elements further.

The temporary area we use in step 2 may be implemented in various ways. For example, we can maintain the elements in the structure partitioned into two contiguous groups of $n - k$ and $k$ elements, respectively, and use the $k$ elements to simulate a temporary area with the internal buffering technique. Alternatively, we can simply exchange $b$ and $b_t$ (that can be done in-place trivially).

As a result, we now have to insert $b_t$ in $u$'s parent, and replace the parent's pointer to $u$ with the pointers to $u'$ and $u''$. Hence the splitting process in the internal nodes caused by an insert operation does not involve all the elements of each such node, but only those contained in the directories and in $O(1)$ chunks.

**Lemma 11.3** Splitting a permuted internal node (of maximal size $2tk = O(B^e \log n)$) can be performed with

$$O \left( \frac{t + k}{B} \right) = O \left( \frac{\log n}{B} \right)$$

block transfers by relocating only $O(\log n)$ elements in $O(1)$ chunks.

**Proof:** Accessing the directory of the node requires $O(t/B)$ block transfers. Replacing the elements in the selected chunk and re-encoding $O(1)$ pointers requires to access $O(k)$ elements with $O(t/B)$ block transfers. In summary, $O(1)$ chunks are involved and only their $O(k)$ elements are relocated. \(\square\)

In the above process, when we reach an internal node with fewer than $2t$ chunks or when we create a new root, we perform an update by reading all the $O(tk)$ elements in it. We must also relocate the node because its number of elements increases by $k$. This operation requires

$$O \left( \frac{tk}{B} \right) = O \left( \frac{\log n}{B^{1-e}} \right) = O(\log_B n)$$
block transfers and a (costly) node relocation. Fortunately, our B-tree organization guarantees that such an expensive operation is needed at most twice, namely, in the leaf originating the split and in the node in which the splitting process terminates.

**Lemma 11.4** Inserting an element in the implicit B-tree takes

- \(O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right) \) block transfers and

- the cost of resizing \(O(1)\) nodes.

**Proof:** Finding the position of the element in a suitable chunk requires a number of block transfers that is \(O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)\) by Lemma 11.2. After that, a suitable leaf must host one more element, and this costs \(O(tk/B) = O(\log n / B^{1-c}) = O(\log_B n)\) block transfers plus the resizing of the leaf. In the worst case, the leaf and its \(O(\log_B n)\) ancestor nodes have to be split, each split operation costing \(O((t+k)/B) = O\left(\left\lceil \frac{\log n}{B} \right\rceil \right)\) by Lemma 11.3. Finally, an internal node has to be resized like the leaf. The total cost is \(O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)\) plus the resizing of at most two nodes. \(\square\)

### 11.2.4 Deleting an element

As for the delete operation with an element \(x\), we identify the chunk \(b_j\) of a node \(u\), such that \(x\) is in \(b_j\), by running the search previously described. We then take the smallest element in the successor chunk of \(b_j\) in the leftmost leaf descending from the child \(c_{j+1}\) of \(u\), since the chunks of the B-tree are disjoint intervals as previously noted. That element replaces the rightmost element of \(b_j\) in the directory of \(u\), since the latter element reuses the space left free by \(x\) in \(b_j\) (whose auxiliary information has to be re-encoded).

We deal with \(O(t)\) elements in the directory, \(O(k)\) elements in the chunk \(b_j\) of \(u\), and \(O(tk')\) elements in the leftmost leaf mentioned above. Here, we have a similar situation to that of insertion. We end up deleting an element in a leaf, which can initiate a merge operation or the sharing of a chunk of a node with one of its siblings. Merging can be treated similarly to splitting, and sharing is essentially reducible to the problem of increasing the size of a node by \(k\) and of decreasing the size of one its adjacent siblings by \(k\). As for merging, we can merge two nodes of size \(tk\) by recording both the fact they are now a single node and a reference in one to the other. We can borrow a chunk from a node’s parent in a manner analogous to splitting.

**Lemma 11.5** Merging two permuted internal nodes (of minimal size \(tk\)) can be performed with

\[O\left(\frac{t+k}{B}\right) = O\left(\frac{\log n}{B}\right)\]

block transfers by relocating only \(O(\log n)\) elements in \(O(1)\) chunks.
Proof: Analogous to that of Lemma 11.3. \hfill \square

Again, the expensive operations are at the leaf originating the merge and at the node in which the merging process stops (because it contains more than $t$ chunks or it can borrow a chunk from one of its adjacent siblings with the sharing operation). This takes $O(tk/B) = O(\log_B n)$ block transfers and at most three node relocations because of the change of their size.

**Lemma 11.6** Deleting an element in the implicit B-tree takes

- $O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right)$ block transfers and
- the cost of resizing $O(1)$ nodes.

Proof: Finding the position of the element in a suitable chunk requires a number of block transfers that is $O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right)$ by Lemma 11.2. After that, a suitable leaf has to host one less element, with $O(tk/B) = O(\log n / B^{1-c}) = O(\log_B n)$ block transfers and the resizing of the leaf. In the worst case, the leaf and its $O(\log_B n)$ ancestor nodes have to be merged, each merge requiring $O((t + k)/B) = O \left( \left\lceil \frac{\log n}{B} \right\rceil \right)$ by Lemma 11.3. Finally, an internal node has to be resized because it does not propagate the merge or it applies sharing to borrow a chunk from one of its siblings. The total cost is $O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right)$ plus the resizing of $O(1)$ nodes. \hfill \square

At this level of detail, we want to maintain the implicit B-tree as a permutation of its stored elements occupying a fixed and compact portion of $n$ memory locations (each capable of storing exactly one element). So an update can be seen as a transition from a permutation of $n$ elements to a permutation of $n + 1$ or $n - 1$ elements. The main computational difficulty in maintaining this interesting property is that nodes can change their sizes after an update (searching is not a problem in this sense). To overcome this problem, we must recompact some nodes by shifting their elements and possibly relocating them by exchanging positions in the permutation.

Our organization guarantees that the expensive relocation mentioned above is necessary for $O(1)$ nodes (a leaf and possibly one ancestor node and its sibling). The rest of the nodes in a root-to-leaf path are involved in split or merge operations that are computationally cheaper, and do not need relocation as they locally permute only $O(k)$ elements (see Lemmas 11.3 and 11.5). The importance of this fact is clear; in the worst case, an update operation requires a constant number of transformations of the first type, while the number of splits or merges depends on the height $O(\log_B n)$ of the tree.

In the rest of the chapter, we proceed to accommodate the elements of the nodes in memory, according to a two-level scheme. In the first level (Section 11.3), we pack together the nodes of identical size, embedding them in compactor lists to eliminate most of the memory waste. In this context, we preserve the invariant that lists are
made up of fixed-size allocation units, and each node is stored in $O(1)$ allocation units and must encode also the auxiliary information for the lists.

We use an abstract view of the memory manager, which we describe later on in the second level (Section 11.4). Here, we assign contiguous memory to the allocation units of the lists and dynamically maintain a further packing of the initial unit (head) of each list, as the heads are the only allocation units partially filled.

## 11.3 Embedding the B-tree Nodes into the Compactor Lists

In this section, we show how to pack the variable-size nodes into a suitable collection of compactor lists of fixed-size allocation units, which will be stored compactly as discussed in Section 11.4 to relocate memory areas easily and without memory waste.

### 11.3.1 Encoding information in the nodes

Before describing the compactor lists in detail, we must fix the information implicitly encoded in the nodes. That is, given a B-tree node $u$ storing $ik$ elements, where $t \leq i \leq 2t$, we can read the information stored in $u$ in two ways:

- **Explicit**: $u$ is a sequence of $ik$ elements, in which the first $2t$ elements form the directory, followed by what remains of $t$ chunks $b_1, \ldots, b_t$. Then, if $i > t$, we have the second directory with $2r$ elements followed by what remains of $r$ chunks $b_{t+1}, \ldots, b_i$, where $r = t - i$. Recall that these chunks and the corresponding pairs of elements in the directories are shuffled according to a permutation $\sigma$ induced by the update operations (see Section 11.2). To recover chunk $b_{\sigma(j)}$ in position $j$ of the permutation, we have also to take the pairs of elements in positions $2j - 1$ and $2j$ inside the directory.

- **Implicit**: $u$ can encode up to $ik/2$ bits of auxiliary information partitioned into $i$ segments of $k/2$ bits each. Each segment is encoded by the pairwise exchange of elements, so it needs $k$ elements. The value of $k/2$ is enough to encode an initial reserved area that we will use shortly to encode the auxiliary information for the compactor lists, and two pointers to the children of $u$. It is crucial to keep the segments shuffled by the same permutation $\sigma$ of the chunks described in the previous point.

The two ways of reading the elements in $u$ are not aligned because the directories in $u$ cause different displacements. But we can retrieve a chunk and its associated auxiliary information by reading the directory and accessing a suitable segment. Indeed the information encoded for the $j$th chunk in the permutation is encoded in the $j$th segment. As a result, the whole structure of the B-tree node is efficiently encoded by its elements alone.
One important consideration is that of the information in the reserved area of segments. As we shall see, we must physically move the nodes from one part of the memory to another. In performing this task, we must take care to re-encode (permute) the information in the reserved area after reorganizing a node.

11.3.2 Packing the nodes in the lists

Keeping the above structure in mind, we can classify nodes according to their size. We pack those having identical size $ik$ into a list labeled $i$ to reflect that each of them contains $i$ chunks of $k$ elements. The total number of lists thus created is $t+1$.

The layout of the leaves. Since the leaves have size ranging from $tk$ to $2tk+k−1$, we treat them differently. We store each leaf as a

- **leading part** being of size a multiple of $k$,

- plus at most two **maniples** each of size at least $k/2$, indicated respectively as **primary** and **secondary**.

This minimal length property allows us to encode logical pointers. The size of the leading part changes by $k$ and that of the maniples by 1, as we preserve the following invariants:

1. If the primary maniple exists, then it contains between $k/2$ and $k−1$ elements. These elements are all greater than those in the leading part.

2. If the secondary maniple exists, then:

   (a) the primary maniple also exists;
   (b) the secondary maniple must contain exactly $k/2$ elements;
   (c) these elements are greater than those in the primary maniple and in the leading part.

3. The rest of the elements (i.e., those in the leading part) must be a multiple of $k$.

Note that we can preserve the invariants under the insertion and deletion of a single element by scanning all the elements in the leaf. It implies that the leading part contains $ik$ elements with $t−1 ≤ i ≤ 2t$. As previously mentioned, the maniples have size ranging from $k/2$ to $k−1$. This motivates the creation of another collection of $k/2$ lists for the maniples, and an additional list labeled $t−1$ to store the leading parts of size $(t−1)k$. 
11.3. EMBEDDING THE B-TREE NODES INTO THE COMPACTOR LISTS

**Encoding the compactor lists.** We refer to the two collections of lists above as *compactor lists*, as our allocation scheme hinges on their efficient manipulation. Specifically, we have \( t + 2 \) lists for the B-tree nodes, numbered from \( t - 1 \) to \( 2t \), where list \( i \) packs the B-tree nodes and the leaves’ leading parts of size \( ik \). We call them \( tk \)-lists to reflect that each list is made up of doubly linked *allocation units* of size \( tk \).

We allocate the B-tree nodes to lists according to the size of the nodes. If a node is of size \( tk \), it is stored in a single allocation unit. If its size is \( 2tk \), it occupies two allocation units. The remaining nodes contain each a number of elements that is not a multiple of the size \( tk \) of an allocation unit. Hence we pack the elements of each such node in at most three allocations units, where the first and the last units can be shared with other nodes. Each allocation unit contains the elements of at most two B-tree nodes. For the sake of presentation, we will discuss only B-tree nodes as the leading parts of leaves are handled identically in the lists.

We need a further property to encode the pointers in the list. Since the size of a unit is a multiple of \( k \), the unit must start with a segment of a node and the auxiliary information of the unit is encoded in the reserved area of that segment. In particular, we encode the two pointers that link the allocation unit in its list, and the displacement of the beginning (if any) of the B-tree node inside that unit (the first element in a node is not necessarily the first element in the unit).

We also have \( k/2 \) lists for the maniples, numbered from \( k/2 \) to \( k - 1 \), where list \( i \) packs the maniples of size \( i \). We call them \( k \)-lists to reflect that each list is made up of doubly linked *allocation units* of size \( k \). They are handled in a manner analogous to the \( tk \)-lists. Note that, since each manipulate has size at least \( k/2 \), each allocation unit can store the elements of at most two maniples and encode two pointers for its list.

We point out the crucial property that only \( O(1) \) pointers refer to an element or a node in any allocation unit. This follows from the fact that each allocation unit stores the elements of at most two B-tree nodes or maniples. It will be useful when relocating the units from one area to another of the memory in Section 11.4.

### 11.3.3 Resizing the objects in the compactor lists

**Resizing the maniples.** We now discuss how to resize a manipulate after inserting or deleting a element from its leaf. As previously mentioned, we can maintain the invariants on the maniples of the leaf by a linear scan of its elements with \( O(tk/B) = O(\log_B n) \) block transfers. Because of this, we may have to change the size of the maniples and of the leading part of the leaf. As for the leading part, we can increase or decrease its size by \( k \), as outlined in the next paragraph.

As the size of a manipulate changes, we simply move it from one \( k \)-list to another. For example, a manipulate can move from list \( i \) to list \( i + 1 \) in case of an insertion without violating the size invariant, or vice versa in case of a deletion. If the list is \( i = k - 1 \), an insertion removes the manipulate from that list, and the \( k \) elements
(including the newly inserted one) are added to the leading part. If \( i = k/2 \) and the maniple is primary, the deletion of an element may cause the redistribution of the elements with the secondary maniple (or its creation, if it does not exist, by taking the elements from its leading part).

There are few such cases to handle and all of them require moving one or two maniples from their \( k \)-lists. Consequently, we discuss how to move a maniple \( m \) from list \( i \) to list \( j \).

1. We first exchange \( m \) with the maniple at the beginning of list \( i \) to preserve the packing of the maniples in the list.

2. We then change the size of \( m \) from \( i \) to \( j \) as required by the operation at hand.

3. Next we move \( m \) from the beginning of list \( i \) to its proper destination at the beginning of list \( j \).

In other words, we reduce the problem of resizing a maniple to the problem of moving the head (i.e., allocation unit at the beginning) of two \( k \)-lists.

**Lemma 11.7** Resizing a maniple in the \( k \)-lists requires

- the relocation of \( O(1) \) maniples and
- \( O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right) \) block transfers.

*Proof:* We relocate two maniples in the same list, and move the head from one list to another. After that, we need to identify the \( O(1) \) leaves containing the relocated maniples by choosing one element per relocated maniple and searching that element in the data structure (Lemma 11.2). \( \square \)

**Resizing the B-tree nodes.** We resize and relocate the leading part of a leaf after preserving the invariants on its maniples, and do the same with an internal node after a sequence of split or merge operations. Sharing elements with a sibling node can be seen analogously, as we increase the size by \( k \) in a node and decrease the size by \( k \) in the sibling node. In these cases, we move a node from list \( i \) to list \( i + 1 \) (if a chunk of \( k \) elements is added) or from list \( i + 1 \) to list \( i \) (if a chunk is deleted). Handling them is analogous to that of \( k \)-lists, except that we move nodes around instead of maniples, and allocation units are of size \( tk \).

A different situation arises when the size of an internal node changes because of a split (Lemma 11.3) or a merge (Lemma 11.5). Although we logically change lists (from list \( t \) to list \( 2t \), or vice versa), we need not relocate the nodes and search in the data structure. It suffices to re-encode \( O(1) \) pointers in the \( tk \)-lists because we keep the directories in the internal nodes.
Lemma 11.8 Resizing a node in the tk-lists has a cost given by

- the relocations of $O(1)$ nodes and
- $O \left( \left\lfloor \frac{\log n}{B} \right\rfloor \log_B n \right)$ block transfers.

The cost of resizing reduces to $O \left( \left\lfloor \frac{\log n}{B} \right\rfloor \right)$ when splitting or merging the node (no relocation needed).

Proof: We relocate two nodes in the same list, and move the head from one list to another. After that, we need to identify the $O(1)$ nodes whose encoded pointers refer to the relocated nodes. Again, we choose one element per relocated node and search that element in the data structure (Lemma 11.2). When splitting or merging, we only have to re-encode $O(1)$ pointers by accessing their elements in $O(k/B)$ block transfers.

We can now store the compactor lists by allocating contiguous memory to each allocation unit. As a result, we obtain an almost implicit data structure in that the heads of the lists are the only allocation units that are partially filled. In order to achieve a fully implicit data structure in $n$ memory locations, we have to handle the floating heads of the compactor lists as described in Section 11.4.

11.4 Low-Level Memory Organization of the Compactor Lists

We are left with the problem of storing the compactor lists in adjacent memory locations with no waste of space. As previously noted, the allocation units are full except the heads of the lists. We have to deal with these floating heads by packing them in a suitable area of the memory in a dynamic setting.
11.4.1 Memory layout

Given \( n \) adjacent memory locations in which we have to store the B-tree, we lay out the compactor lists according to the following low-level memory organization in three consecutive main areas (see Figure 11.1):

**tk-area** contains all the \( tk \)-lists in a total space that is a multiple of \( k \). The area is divided into two zones. Zone A stores all the allocation units that are not the heads in their lists. Hence, its size increases and decreases by \( tk \). Zone H is floating and packs together the elements stored in the heads of the lists. Its size increases and decreases by \( k \).

**k-area** contains all the \( k \)-lists. Again, here zone A contains all the allocation units that are not the heads in their lists, and its size increases and decreases by \( k \). Zone H is floating and packs together the elements stored in the heads of the lists. Its size increases and decreases by 1.

**root area** contains the root \( R \) of the B-tree which is treated separately from the other B-tree nodes. Its elements are stored at the end of the allocated memory area shown in Figure 11.1.

We also have to encode some additional pointers:

- The \( t + 2 \) pointers to the allocation units that are heads of the \( tk \)-lists (there are \( t + 2 \) of them) and a further \( t + 2 \) pointers to the next-to-head allocation units.

- The analogous pointers to the head and next-to-head units in the \( k \)-lists (there are \( k/2 \) of them), which adds up to further \( k \) pointers.

Globally, if our data structure stores \( \Omega((t + k)k) \) elements, we can encode these pointers and spread them in the reserved areas of the first segments in the \( tk \)-area. Then each of them can be retrieved by scanning \( O(k) \) elements. (In the border case of \( O((t + k)k) \) elements, we implement each operation on our data structure by a sequential scan of these few elements; see Section 11.5 for reducing their number.)

We now have to show how to manage the compactor lists according to the above memory organization. By Lemma 11.7 and 11.8, we can focus on moving elements from one head to another.

11.4.2 Relocation in the \( k \)-area

Suppose we want to move the maniple \( x \) packed at the beginning of list \( i \) to the beginning of list \( i + 1 \) because there is a new element added to \( x \) for obtaining \( x' \) with \( i + 1 \) elements (the other operations are treated analogously). An example is shown in Figure 11.2.
Figure 11.2: An example of relocation of maniple \( x \) being resized as maniple \( x' \) (with one more element) in the \( k \)-area. (a) The head of list \( i \) contains part of \( x \), and the next-to-head contains the rest of \( x \) and part of \( y \) (if any). (b) The next-to-head is swapped with the last allocation unit in zone \( A \). (c–d) After the addition of a new element in location \( c \) to obtain \( x' \), maniple \( x \) disappears from list \( i \) and the next-to-head of list \( i \) becomes the new head containing part of \( y \) only; the new head of list \( i + 1 \) becomes \( x' \).

1. We begin by shifting all the \( O(tk) \) elements in root \( R \), one position to the right, to make room for one more element in zone \( H \).

2. Let \( c \) be this free memory location (which we fill with the extra element), and \( y \) be the maniple that follows \( x \) in list \( i \), such that the last elements in \( x \) and the first elements in \( y \) share the same allocation unit (\( y \) may not exist). Since \( x \) may occupy at most the first two allocation units of list \( i \), we know that the first unit is in zone \( H \) of the \( k \)-area and the second unit, if any, is in zone \( A \) (Figure 11.2.a). We swap the latter unit with the last unit in zone \( A \), at the cost of moving \( O(k) \) elements and searching the data structure to identify the \( O(1) \) leaves containing the involved maniples (Figure 11.2.b).

3. At this point, we have the head of list \( i \) in zone \( H \) and the next-to-head at the end of zone \( A \), immediately adjacent to zone \( H \). Scanning the \( O(k^2) \) elements in the last unit of zone \( A \) and in the whole zone \( H \), we can make \( y \) be the beginning of list \( i \), and \( x' \) be the beginning of list \( i + 1 \) (Figure 11.2.c). (We will describe in Section 11.5 how to scan fewer elements.)

4. If \( x' \) contains \( k \) elements or less, we (logically) shorten the end of zone \( A \) by \( k \) positions and increase zone \( H \) by \( k \) positions to its left (this is the case illustrated in Figure 11.2.c).
Otherwise, \( x' \) contains more than \( k \) elements: we take the last \( k \) elements in \( x' \) and form an allocation unit in list \( i + 1 \) to be placed at the end of zone A, and keep the rest of the elements in \( x' \) in the head of list \( i + 1 \) in zone H (see Figure 11.2.d).

5. We need to update \( O(k) \) pointers to the heads and next-to-heads (of \( k \)-lists) that are encoded in the \( tk \)-area, because the change of size of one head causes the shift of the others.

**Lemma 11.9** Relocating a maniple in the k-area has an I/O complexity that is

\[
O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n + k^2 / B \right) = O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n + \frac{\log^2 n}{B} \right) .
\]

*Proof:* The total cost of the operation is given by scanning and/or moving \( O(k^2) \) elements (in the whole zone H and in at most two allocation units of zone A), updating \( O(k) \) pointers by scanning \( O(k^2) \) elements (the pointers to the heads and the next-to-heads), and performing \( O(1) \) searches (Lemma 11.2) in the data structure. \( \square \)

### 11.4.3 Relocation in the \( tk \)-area

Suppose we want to move a node at the beginning of list \( i \) to list \( i + 1 \). This is caused by the fact that we wish to add a chunk of \( k \) elements to an internal node or to the leading part of a leaf. In other words, a chunk of size \( k \) in the \( k \)-area has to be moved to increase the size of a node in the \( tk \)-area. From the discussion above on the relocation in the \( k \)-area, this chunk occupies the last allocation unit in zone A of the \( k \)-area. Therefore we proceed in the following way:

1. We exchange it with the first allocation unit in zone A, so that it immediately follows zone H of the \( tk \)-area.

2. We logically increase the latter area by \( k \) positions to its right and decrease the \( k \)-area by \( k \) positions to its left.

3. We now have all relevant elements in zone H of the \( tk \)-area, proceeding in a fashion similar to what done in the \( k \)-area.

**Lemma 11.10** Relocating a node in the \( tk \)-area has an I/O complexity that is

\[
O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n + t^2 k / B \right) = O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right) .
\]
Proof: The total cost of the operation is given by scanning and/or moving $O(t^2k)$ elements (in the whole zone $H$ and in at most two allocation units of zone $A$), updating $O(t)$ pointers by scanning $O(tk)$ elements (the pointers to the heads and the next-to-heads), and performing $O(1)$ searches (Lemma 11.2) in the data structure.

Notable exceptions are the merge and split operations in the internal nodes. Here, we logically merge or split two allocation units of $tk$ elements each, and note that this does not require physically moving the heads of lists $tk$ and $2tk$. The cost is just that of scanning $O(t+k)$ elements, even though the total size of the segments is $\Theta(tk)$, as anticipated in Lemma 11.3, Lemma 11.5 and Lemma 11.8.

As a general rule in our allocation scheme, a pointer from an internal node to one of its children uses a level of indirection if the child is a head of one of the $tk$-lists. Similarly, the pointer in the leading part of a leaf uses indirection if the manipulate is head of one of the $k$-lists. In order to check this rule, it suffices to see if the pointer refers to a reserved area inside a segment by a simple mathematical computation. This rule requires minor modifications in our algorithms and does not change the complexity of the algorithms.

Lemma 11.11 An implicit B-tree for $n$ elements stored in $n$ memory locations can be maintained so that

- insertions and deletions take a worst case number of block transfers that is

$$O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n + \frac{\log^2 n}{B}\right).$$

- Searching an element takes a worst case number of block transfers that is

$$O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n \right).$$

- The only operations performed on the elements are comparisons and moves; and furthermore no additional space is required for the data structure. That is, the $n$ memory locations store both the elements and the (implicit) data structure.

Proof: The cost of searching derives from Lemma 11.2. For the insertion, each operation requires $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n \right)$ block transfers plus the cost of resizing $O(1)$ nodes by Lemma 11.4. Resizing the node not involved in a split or the leaf hosting one more element and its maniples takes $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n \right)$ block transfers plus the relocation of $O(1)$ nodes and maniples by Lemma 11.7 and 11.8. Relocating $O(1)$ nodes and maniples takes
\[ O \left( \left[ \frac{\log n}{B} \right] \log_B n + \frac{t^2 k}{B} + \frac{k^2}{B} \right) \]

block transfers by Lemma 11.9 and 11.10. Finally, it takes \( O \left( \left[ \frac{\log n}{B} \right] \right) \) block transfers to split a node (with at most \( O(\log_B n) \) such nodes) by Lemma 11.8. Hence, the total cost per insertion is

\[ O \left( \left[ \frac{\log n}{B} \right] \log_B n + \frac{t^2 k}{B} + \frac{k^2}{B} \right) = O \left( \left[ \frac{\log n}{B} \right] \log_B n + \frac{\log^2 n}{B} \right), \]

since \( k = O(\log n) \) and \( t = \Theta(B^\epsilon) \), for a positive real constant \( \epsilon < 1/2 \). The analysis for a deletion is similar by replacing Lemma 11.4 by Lemma 11.6 in the analysis for an insertion. \( \Box \)

### 11.5 Refining the Solution

In this section, we refine the data structure in order to get better bounds by lowering the \( \frac{\log^2 n}{B} \) additive term in Lemma 11.11. We focus on the case \( B^{1-\epsilon} < \log_B n \) in the rest of this section, as that term is bounded by \( O(\log_B n) \) when \( B^{1-\epsilon} \geq \log_B n \).

We must avoid accessing \( \Theta(k^2) \) elements when relocating the maniples in the \( k \)-area (see Lemma 11.9). We deal with two situations:

1. We have to access the \( \Theta(k) \) pointers to the heads and to the next-to-heads of \( k \)-lists. Recall that these pointers are encoded by \( \Theta(k^2) \) elements in the \( tk \)-area, namely, in the reserved area of the segments in the first nodes of zone A.

2. We have to shift elements in zone H of the \( k \)-area, when resizing a maniple stored in one of the \( k \)-list heads. Recall that all heads of \( k \)-lists are packed together in zone H and contain \( \Theta(k^2) \) elements in total.

In the following, when we refer to zone H, we mean inside the \( k \)-area.

**Accessing the head-pointers of the \( k \)-lists.** Solving point 1 is fairly easy. We use a further level of indirection by encoding the \( \Theta(k) \) pointers to the heads inside zone H and by putting a single pointer at the beginning of the \( tk \)-area to reflect this displacement.

Note that each pointer encoded in zone H is tagged with the number \( j \) of its \( k \)-list. If the list is empty, we can avoid to store the pointer. Letting \( g \leq k/2 \) be the number of nonempty lists, at the beginning of the \( tk \)-area we keep the \( g \) pointers to the next-to-heads as they are not changed by a relocation of elements inside zone H. The number of encoded pointers at the beginning of the \( tk \)-area drops from \( \Theta(t+k) \)
11.5. Refining the Solution

259
to $\Theta(t + g)$, requiring thus only $\Theta((t + g)k)$ elements to set up our data structure, which is always guaranteed by the definition of $g$.

The smallest number of elements in this case is for $g = O(1)$ and an individual node satisfies the above requirement since it stores $\Theta(tk)$ elements. A larger value of $g$ means that we have $\Omega(gk)$ additional elements, and so we may encode the $g$ pointers. On the other hand, if we have $o(tk)$ elements, we can run brute-force updates and searches by scanning all the elements.

**Shifting the heads of the $k$-lists.** As for point 2, we extend the definition of heads in $k$-lists, so that nonempty lists have heads containing $\Theta(k)$ elements (e.g., we can unite the previously defined heads with their next-to-heads and make next-to-heads point to the successors of the resulting heads).

The main idea is to fragment each head of a $k$-list in zone $H$ into allocation units of size $\ell = \Theta(\log B n)$, where $k$ is a multiple of $\ell$. In a certain sense, we wish to represent the heads in zone $H$ as $\ell$-lists, so that we transform the $k/2$ heads of the $k$-lists into $k/2$ lists whose allocation units have size $\ell$ inside zone $H$. The association rule is immediate, namely, the whole $\ell$-list $j$ stores the elements in the head of $k$-list $j$ (so, a pointer to the latter head and the pointer to the former list are logically the same). The full allocation units of these $\ell$-lists are stored at the beginning of zone $H$. The heads of the $\ell$-lists are stored in a compact way at the end of zone $H$.

Note that the latter heads contain $O(k\ell)$ elements, which we can fully scan with the following I/O complexity:

$$O\left(\frac{k\ell}{B}\right) = O\left(\left\lceil \frac{\log n}{B} \right\rceil \log B n\right).$$

We encode the pointers to the heads and to the next-to-heads of the $g$ nonempty $\ell$-lists in the first $O(g)$ allocation units of size $\ell$ in zone $H$, as each such pointer requires $O(\ell)$ bits. We also tag each pointer with the number $j$ of its $\ell$-list, encoding $j$ by a pairwise permutation of the elements. Note that, to have direct access to one of the $g$ pointers to next-to-heads in point 1, we can scan the pointers to the $g$ nonempty $\ell$-lists in the first $O(g\ell)$ elements of zone $H$ and find its rank $h$. Then, we simply decode the $h$th pointer to next-to-heads in point 1. At any time, the order of the pointers to the $\ell$-lists in zone $H$ reflects that of the $g$ pointers in point 1.

**Lemma 11.12** Relocating a maniple in the $k$-area, as stated in Lemma 11.9, can be performed with a number of block transfers that is

$$O\left(\left\lceil \frac{\log n}{B} \right\rceil \log B n\right).$$

**Proof:** Let’s first see how to recover the head of $k$-list $j$.

We identify the head of $\ell$-list $j$ by scanning all heads of size $O(\ell)$ in the last part of zone $H$, examining $O(k\ell)$ adjacent elements with
\[ O \left( \frac{k \ell}{B} \right) = O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right) \]

block transfers.

We traverse the whole \( \ell \)-list \( j \) in zone \( H \) with further

\[ O \left( \left\lceil \frac{\ell}{B} \cdot \frac{k}{\ell} \right\rceil \right) = O \left( \frac{k}{B} + \frac{k}{\ell} \right) = O \left( \frac{\log n}{B} + \log B \right) = O \left( \log_B n \right) \]

block transfers, since \( B^{1-\omega} < \log_B n \).

Now, when relocating a maniple, we have to handle two heads, which can recovered and recompiled as described above, with

\[ O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right) \]

block transfers, plus the cost of \( O(1) \) searches (Lemma 11.2).

We also need to update the pointers to the heads of the \( k \)-lists, encoded in the first \( O(g \ell) = O(k \ell) \) elements of zone \( H \). The total cost is

\[ O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right) \cdot \]

When we have to resize one of the maniple, we can proceed similarly with a cost of

\[ O \left( \frac{k}{\ell} + \frac{k \ell}{B} \right) = O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right), \]

as we can move single units of size \( O(\ell) \). Note that it is not necessary to maintain the elements in sorted order. \( \square \)

We have described a single data structure for implicit B-trees. In general, we adopt the scheme by Frederickson [Frederickson, 1984b], keeping \( O(\log \log n) \) distinct data structures of size \( 2^c, c \leq i \leq \lfloor \log \log n \rfloor \) (where \( c \) is a constant), so that the total cost is dominated by the cost of the largest of such data structures. Insertions are always performed on the largest structure; a deletion is performed by moving an element from the largest structure into the one where the element to be deleted is found. We obtain our main result.

**Theorem 11.1** An implicit B-tree for \( n \) elements can be maintained in \( n \) memory locations so as to support

- searching, inserting and deleting with a worst case number of block transfers that is
\[ O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right), \]

- reporting \( r \) consecutive elements in sorted order with a worst case number of block transfers that is

\[ O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n + r/B \right). \]

Proof: If \( B^{1-\epsilon} \geq \log_B n \), we can use Lemma 11.11. Otherwise, it suffices to replace Lemma 11.9 by Lemma 11.12 in the proof of Lemma 11.11, noting that the cost of \( O(\left\lceil \frac{\log n}{B} \right\rceil \log_B n) \) to retrieve at most two maniples in a leaf does not change the overall complexity. In order to report \( r \) consecutive elements in sorted order and belonging to a range \([x, y]\), let’s suppose to have performed a search of \( x \) and \( y \) to identify two paths in the implicit B-tree. We then scan all the heads of size \( O(\ell) \) in the last part of zone \( H \), examining \( O(k\ell) \) adjacent elements with

\[ O \left( \frac{k\ell}{B} \right) = O \left( \left\lceil \frac{\log n}{B} \right\rceil \log_B n \right) \]

block transfers to report those contained in the interval \([x, y]\). Then, it suffices to report all the elements traversed in the nodes delimited by the two paths. The procedure is essentially that of regular B-trees, with three main observations.

(i) We do not have to scan the elements in the heads of size \( O(\ell) \), because they are processed as mentioned above.

(ii) The \( O(k/B + k/\ell) = O(\log_B n) \) cost of recovering the elements in the maniples of a leaf that are not in (1), is dominated by the cost \( O(tk/B) = O(\log_B n) \) of accessing the rest of the elements in the leaf; indeed, \( k/B + k/\ell = O(tk/B) \) if and only if \( B/t < \ell \), which is the condition \( B^{1-\epsilon} < \log_B n \) that we satisfy.

(iii) The \( O(k) \) elements needed to decode a pointer either contribute to the cost \( O(\left\lceil \frac{\log n}{B} \right\rceil \log_B n) \) or they are part of the output, and so the cost of decoding that pointer is absorbed by the output-sensitive cost.

To output the elements in order, it suffices to make an on-line merge of the elements belonging to \([x, y]\) and residing in the heads of size \( O(\ell) \) in the last part of zone \( H \) with the elements retrieved with the traversal of the implicit B-tree. \( \square \)

Under the realistic assumption that \( B = \Omega(\log n) \), the term \( O(\left\lceil \frac{\log n}{B} \right\rceil \log_B n) \) in Theorem 11.1 reduces to \( O(\log_B n) \) like in regular B-trees. Note that previous B-tree-like data structures occupied \( n + \Omega(n) \) memory locations with the above bounds, while we occupy exactly \( n \) locations (the minimum possible with no waste of space) in \( \lceil n/B \rceil \) blocks of memory.
Corollary 11.1 When $B = \Omega(\log n)$, the implicit B-tree takes $O(\log_B n)$ block transfers per operation and reporting $r$ consecutive elements in sorted order requires $O(\log_B n + r/B)$ block transfers. The memory occupancy is optimal, i.e., $[n/B]$ blocks of memory.

In main memory, we can fix $B = \Theta(\log n)$ and multiply the number of block transfers by $O(B)$ to get the total (CPU) running time.

Theorem 11.2 Fixing $B = \Theta(\log n)$, an implicit B-tree for $n$ elements stored in $n$ locations of main memory supports insertions, deletions and searches with a worst case complexity

$$O(\log^2 n / \log \log n).$$

11.6 Conclusions

We have introduced a data structure, the implicit B-tree, to provide new bounds for searching and updating an implicit dictionary, which is entirely encoded by a suitable permutation of its elements. In hierarchical memory, these operations require $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n\right)$ block transfers in the worst case, and reporting $r$ consecutive elements in sorted order takes $O\left(\left\lceil \frac{\log n}{B} \right\rceil \log_B n + r/B\right)$ block transfers. Under the realistic assumption that $B = \Omega(\log n)$, the above bounds reduce to $O(\log_B n)$ like in regular B-trees.

In main memory, we can implement the dictionary operations in $O(\log^2 n / \log \log n)$ time, improving a long-standing bound of $O(\log^2 n)$ for the problem (see [Munro, 1986]).
Chapter 12

Exponential Implicit Tree

Abstract

In this chapter we describe an implicit dictionary supporting searches in $O(\log n \log \log n)$ time in the worst case, and insertions and deletions in $O(\log n \log \log n)$ amortized time. As we saw in the previous chapter the problem of the implicit dictionary is very old and the best known bound in the RAM model before the Implicit B-Tree was $O(\log^2 n)$ in the worst case [Munro, 1986]. With the advent of the Implicit B-Tree (see Chapter 11, [Franceschini, Grossi, Munro, and Pagli, 2002] and [Franceschini, Grossi, Munro, and Pagli, 2004]) the best known bound became $O(\log^2 n / \log \log n)$ still in the worst case.

Since there seems to be a compulsory $O(\log n)$ slowdown factor for any level of an implicit tree and in order to beat the long-standing bound in [Munro, 1986], the basic strategy adopted with the Implicit B-tree was to lower the height of the tree to $O(\log n / \log \log n) = o(\log n)$. With the Exponential Implicit Tree we move forward in this strategy using an exponential tree with $O(\log \log n)$ levels (of course the difficult part in all that is not how to figure out this basic strategy but how to make things work with such increasingly flattened trees). With the Exponential Implicit Tree the problem of amortized analysis in fully dynamic implicit dictionaries (i.e. dictionaries supporting both insertions and deletions) is faced for the first time.

The presentation in this chapter is based on the paper [Franceschini and Grossi, 2003a] (SODA 2003).

12.1 Introduction

In Chapter 11 (see [Franceschini, Grossi, Munro, and Pagli, 2002] and [Franceschini, Grossi, Munro, and Pagli, 2004]), we presented the Implicit B-Tree, an implicit data structure for the External-Memory model (see Chapter 2) that requires $O(\log_B n)$ block transfers per operation in the worst case like B-trees, assuming that the block
size is $B = \Omega(\log n)$. If we use the Implicit B-Tree in the RAM model (see Chapter 2) fixing $B = \Theta(\log n)$, it provides a bound of $O(\log^2 n / \log \log n)$ in the worst case for search, insert and delete operations. This was the first surprising improvement of a long-standing bound of $O(\log^2 n)$ in [Munro, 1986]. That reopened the issue of finding an optimal implicit dictionary for the RAM model, that is an implicit dictionary requiring $O(\log n)$ time for searching and updating.

In this chapter, we provide a further step towards this direction, and describe an implicit data structure supporting searches in $O(\log n \log \log n)$ time in the worst case, and insertions and deletions in $O(\log n \log \log n)$ amortized time. A lesson learned from [Franceschini, Grossi, Munro, and Pagli, 2002] is that of using a B-tree of large fan-out with height of $o(\log n)$ while allocating variable-size nodes in compact memory lists of fixed-size allocation units. Since there seems to be a compulsory $O(\log n)$ slowdown factor for any level of an implicit tree and in order to beat the long-standing bound in [Munro, 1986], we follow the basic strategy adopted with the Implicit B-tree that is the one aiming to lower the height of the tree to $o(\log n)$. With the Exponential Implicit Tree we move forward in this strategy using an exponential tree with $O(\log \log n)$ levels (of course the difficult part in all that is not how to figure out this basic strategy but how to make things work with such increasingly flattened trees).

The problem of amortized analysis in fully dynamic implicit dictionary (i.e. supporting both insertions and deletions) is faced for the first time here. It should be noted that several data structuring techniques requiring the duplication of elements cannot work in the implicit model.

- Overmars' rebuilding technique [Overmars, 1983] maintains a copy of the data structure.

- Arge's buffering technique [Arge, 1995] should keep a copy of the element to be deleted or a pointer to it.

To make things more complicated, we have that during the amortization only in-place algorithms can be used as otherwise the data structures would not require $O(1)$ auxiliary locations to operate but also a non-constant sized temporary area, which makes no sense in the Implicit Model. As a result, our deletion algorithm is almost symmetrical to the insertion.

Unlike the Implicit B-Tree, when used in external memory, the Exponential Implicit Tree is designed to exploit the characteristics of the RAM model and the bounds in [Franceschini, Grossi, Munro, and Pagli, 2002] and [Franceschini, Grossi, Munro, and Pagli, 2004] for the External-Memory model are not improved. Nevertheless, the Exponential Implicit B-Tree represents a significant improvement over the worst-case bounds in main memory of $O(\log^2 n)$ time in [Munro, 1986] and of $O(\log^2 n / \log \log n)$ time in [Franceschini, Grossi, Munro, and Pagli, 2002, 2004].
12.2 Overview and Roadmap

Preliminaries. We assume without loss of generality that the number $n$ of elements is upper bounded by $N$, such that $\log N = \Theta(\log n)$. This assumption is not a limitation, as crucially shown by Frederickson [Frederickson, 1984b], but it is useful to fix the bit-length of the pointers.

In fact, we can keep $O(\log \log n)$ distinct data structures of size $2^c$, $c \leq i \leq \lceil \log \log n \rceil$ (where $c$ is a constant), so that the total cost is dominated by the cost of the biggest of such data structures. So, our results apply regardless of changes in $n$. Strictly speaking, we do not need to store the value of $n$, but we can use a variable encoding (like $\delta$-codes) with $O(\log n)$ bits implicitly represented with the first elements of the data structure.

As always, we adopt the basic technique of bit stealing (see Chapter 4) to implicitly represent the structural information by encoding a pointer or an integer of $d = \log N$ bits by using $2d$ distinct elements $x_1, y_1, x_2, y_2, \ldots, x_d, y_d$ that are pairwise permuted with the well-known simple rule: (relative) increasing order for a 0, decreasing order for a 1.

Exponential Implicit Tree: invariants. At any time, the elements in our data structure are grouped into chunks of $k = \Theta(\log N) = O(\log n)$ elements each. Each chunk contains (pairwise permuted) elements from a certain interval of values, and chunks are pairwise disjoint as intervals. This allows us to define a total order on any set of the chunks.

Our data structure hinges on a known tree organization of height $O(\log \log n)$, defined by a recursive partition of elements into equal sets (see [Fredman and Willard, 1993, Andersson, Hagerup, Nilsson, and Raman, 1998]. Given $n$ elements, we organize them into chunks as described above.

- We select $\Theta(\sqrt{n})$ chunks as in distribution sorting, and distribute the remaining chunks in buckets. The selected chunks are stored in the root, and the buckets are recursively stored in $\Theta(\sqrt{n})$ children of the root.

- The $i$th bucket is of size $\Theta(\sqrt{n})$ and is stored in the $i$th children of the root. We stop the recursion as soon as the bucket size is $O(1)$ chunks. The basic buckets are stored in the leaves of the tree, all having the same level (distance from the root, which has level 0). The level of the leaves is the height $h$ of our tree.

Formally speaking, let $b$ be any constant power of 2, and $\beta_i$ denote $b^{2^{h-i}}$ for $0 \leq i \leq h$. Our tree satisfies the following properties:

(i) The root contains $r$ chunks such that $1 \leq r \leq 4\beta_0$.

(ii) An arbitrary node at level $i \geq 1$ containing $r$ chunks satisfies $\beta_i \leq r \leq 4\beta_i$.
(iii) An arbitrary leaf contains \( r \) chunks plus at most \( k - 1 \) spare elements, where \( b \leq r \leq 4b \). Leaves are the only nodes whose number of elements can change by 1.

(iv) The height is \( h \leq \alpha \), where \( \alpha \) is the minimum positive integer such that \( b^{2\alpha} = \sqrt{N} \).

Looking at the properties (i)–(iii), we can figure out the tree as a B-tree of doubly logarithmic height \( h = O(\log \log n) \), whose nodes are of variable size depending on their level. The chunks are the basic items encoding the pointers to children in each node.

**Exponential Implicit Tree: dynamics.** Updating relies on splitting, merging and sharing of nodes. However, as we shall see, restructuring the tree is a non-trivial task since the nodes changing size must be relocated in an expensive way. We opt for amortization in this case to distribute the costs during the lifetime of the data structure.

A crucial component of our solution is the memory layout of the nodes to support the amortized updates. In particular, we keep some invariants on the chunks (and their encoded pointers to the children) in the nodes so that, given an internal node \( u \), we classify its chunks as being actual and virtual (leaves are only made of actual chunks). The actual chunks physically reside inside the memory area allocated to \( u \), while the virtual chunks are interleaved with the latter chunks, but stored independently in another special area of the memory.

From a logical point of view, node \( u \) is the merge of the two kinds of chunks, and so a simple variant of the binary search inside \( u \) is sufficient to route the elements to its children. We introduce virtual chunks to accumulate enough potential in the amortized insertions, so as to be able to absorb the cost of reallocating \( u \) to a larger memory area. We need therefore to store virtual chunks in a common area shared by all nodes so as not to change frequently the number of actual chunks during the insertions.

We further classify actual chunks into original and patch to accumulate operations for the amortized deletions so that we postpone the reallocation of \( u \) in a smaller memory area. The invariants on all these chunks guarantee a correct balance, so that amortization can work in \( O(\log n) \) time per level.

**Chapter organization.** This chapter is organized as follows. In Section 12.3, we first focus on an semi-dynamic implicit dictionary that supports only insertions and searches, and introduce the formal definition of actual and virtual chunks. In Section 12.4, we then consider the full case of an implicit dictionary supporting insertions, deletions and searches, introducing original and patch chunks and their invariant. In Section 12.5, we provide the amortized analysis of our data structure.
12.3 A Semi-Dynamic Implicit Dictionary

In this section, we describe a first version of our implicit dictionary, which is able to support searches in $O(\log n \log \log n)$ time and insertions in $O(\log n \log \log n)$ amortized time. We will describe in Section 12.4 how to deal also with deletions.

12.3.1 Actual and virtual chunks

Chunks are at the heart of our data structure as they can implicitly encode bits of information by a suitable permutation of their elements. We need to classify chunks according to their physical location in memory.

- The chunks that are stored in the memory area allocated to a node $v$ are called *actual*. The set of actual chunks of a node $v$ is denoted by $act(v)$. This set can be retrieved directly by accessing $v$.

- We also employ chunks called *virtual* that are still associated with node $v$, but are stored in a separate memory area (note that we do not need virtual chunks in the leaves). The set of the virtual chunks of a node $v$ is denoted by $vir(v)$. This set is represented by a doubly linked list of virtual chunks, in which the pointer to the head of the list is encoded at the beginning of node $v$, and the pointers to the adjacent chunks of $c' \in vir(v)$ are encoded in $c'$ itself.

Actual chunks and virtual chunks are related. We will maintain the following invariants for any node $v$:

(i) each chunk $c \in act(v)$ has associated at most one chunk $c' \in vir(v)$, with a pointer to $c'$ being encoded in $c$, and

(ii) if $c'$ exists, it is the successor of $c$ in node $v$ (hence, $c'$ cannot be the successor of another virtual chunk in $v$).

As a result, the chunks in $act(v)$ and $vir(v)$ are interleaved and encode the pointers to the children of $v$ (see Figure 12.1).

We preserve an invariant on the number of actual chunks and virtual chunks. Recall from Section 12.2 that the actual chunks satisfy the conditions $1 \leq |act(v)| \leq 4\beta_0$ for $i = 0$ (the root) and $\beta_i \leq |act(v)| \leq 4\beta_i$ otherwise. We will maintain two other invariants:

(iii) $|act(v)|$ is always a multiple of $\sqrt{\beta_i}$.

(iv) $\sqrt{\beta_i} \leq |vir(v)| \leq 4\sqrt{\beta_i}$ for $i \geq 1$.

In the rest of the chapter, given an internal node $v$, we define the size of $v$ as the number $|act(v)|$ of its actual chunks (intuitively, changing their number causes a relocation).
12.3.2 Routing and searching elements

In order to efficiently route the elements during a traversal of the tree, we perform a binary search in each traversed node \( v \). Specifically, we perform first the search on the set \( \text{act}(v) \), which is available inside \( v \). Once a chunk \( c \in \text{act}(v) \) is identified by the search, we access its associated chunk \( c' \in \text{vir}(v) \) (if any) in \( O(\log n) \) time and check also \( c' \). As one can see, there is no need to scan the list representing \( \text{vir}(v) \). We anticipate that the layout of node \( v \) in memory can store \( \text{act}(v) \) in \( O(1) \) contiguous memory areas.

**Lemma 12.1** Routing an element in a node requires \( O(\log n) \) time.

Searching is standard, with the observation that we must pay \( O(k) = O(\log n) \) time to decode a pointer from a node \( v \) to one of its children. The cost of searching is \( O(\log n \log \log n) \) in total as the height is \( h = O(\log \log n) \).

**Lemma 12.2** Searching an element in the semi-dynamic implicit dictionary requires \( O(\log n \log \log n) \) time.

12.3.3 Layout in memory

We now detail how the nodes and the virtual chunks are allocated in memory. We divide the memory in \( O(h) \) zones:

- Zone \( P \) has the only purpose to store \( O(\log \log n) \) integers and pointers of \( O(\log n) \) bits each as bookkeeping information related to the several levels in the data structure.

- Zone \( Z_i \) storing the nodes at level \( i \), for \( 0 \leq i \leq h \).

- Zone \( V \) storing all the virtual chunks at the several levels.

The zones in the memory layout appear in the order given by

\[ P, Z_1, Z_2, \ldots, Z_{h-1}, V, Z_h, Z_0. \]

and are illustrated in Figure 12.2.
**Internal nodes’ zones.** Let’s focus on any zone $Z_i$, for $1 \leq i \leq h - 1$. It stores the internal nodes at level $i$ in compacted lists. We encode the memory address of zone $Z_i$ and the value of $\beta_i$ in zone $P$. Knowing that the number of actual chunks in each node is a multiple of $\sqrt{\beta_i}$ ranging from $\beta_i$ to $4\beta_i$, we can see that the size of a node can have at most $3\sqrt{\beta_i} + 1$ different values at level $i$. We therefore keep so many compactor lists for their storage in memory, where doubly linked list $j (0 \leq j \leq 3\sqrt{\beta_i})$ packs all the nodes having size $j\sqrt{\beta_i} + \beta_i$ into allocation units of $\beta_i$ chunks each (a node can be stored in more than one allocation unit). The allocation units are all full, except the head allocation unit, which can be partially filled. In order to preserve the implicitness of the data structure, we divide $Z_i$ in two parts, denoted $A_i$ and $H_i$, so that

- $A_i$ contains the allocation units that are not heads of lists and
- $H_i$ contains their heads in a *compacted form* (concatenation of all the elements in the heads in a single sequence).

Note that $H_i$ contains at most $(3\sqrt{\beta_i} + 1)\beta_i$ chunks. We refer the reader to the compactor lists described in Chapter 11 and [Franceschini, Grossi, Munro, and Pagli, 2002, 2004] for further details, especially on how to encode the pointers of the compactor lists inside their allocation units. We manage the heads in $H_i$ differently from what done in Chapter 11 and [Franceschini, Grossi, Munro, and Pagli, 2002, 2004], taking $O(\beta_k + \sqrt{\beta_i} \log n \log \log n)$ time.

- Part $H_i$ is a collection of $O(\sqrt{\beta_i})$ heads, each containing a multiple of $\sqrt{\beta_i}$ actual chunks.
- Each such head is kept fragmented into $O(\sqrt{\beta_i})$ pieces of $\sqrt{\beta_i}$ linked actual chunks.
- The relocation takes $O(\beta_i k)$ time plus $O(\sqrt{\beta_i} \log n \log \log n)$ time for redirecting $O(\sqrt{\beta_i})$ pointers.
**Remaining zones.** We are left to examine the organizations for three other zones, $V$ for the virtual chunks, $Z_h$ for the leaves and $Z_0$ for the root.

- Next in the memory layout, we have zone $V$ for the virtual chunks. This is a simple sequence of allocation units, each containing exactly one chunk. Its starting position is encoded in zone $P$.

- Following $V$, we have zone $Z_h$ for the leaves of the data structure. Here, the organization of part $A_h$ is identical to that of the other parts $A_i$ in the previous levels, except that the allocation unit of the lists is exactly one chunk of $k$ elements and the number of these lists is $O(k)$ (since the number of elements of a leaf can increase or decrease by 1). As a result, $H_h$ contains $O(k)$ heads, each with $O(k)$ elements. We store these $O(k^2)$ elements in $H_h$ as smaller lists having smaller allocation unit of $\ell = \Theta(\log k)$, where $k$ is a multiple of $\ell$. Handling these smaller list is like in [Franceschini, Grossi, Munro, and Pagli, 2002, 2004] (see Chapter 11), and so we do not detail them any further. We need to encode $O(k)$ pointers of only $O(\log k)$ bits internally in part $H_h$.

- Finally, we have zone $Z_0$ that contains only the root. We handle the whole $Z_0$ (i.e., the root) with a cyclic representation, in which we break it into its single elements. We need $O(1)$ integers and pointers of $O(\log n)$ bits, which are encoded inside zone $P$ for this purpose.

### 12.3.4 Updating the Layout

During the lifetime of the data structure, we need to keep zone $P$ filled with $O(\log N \log \log N) = O(\log n \log \log n)$ arbitrary elements for bookkeeping. They are completely scanned each time at a cost of $O(\log n \log \log n)$ time. If one of them is deleted, we replace it with another element that is removed from the rest of the data structure. So, we can focus on how to update the remaining zones (described in Section 12.3.3) when the size of a node changes.

As for zone $Z_i$, with $1 \leq i \leq h - 1$, the reorganization comes in two fashions.

**Sliding.** We need to translate the whole zone $Z_i$ by no more than $\beta_j k$ (where $j < i$) positions either to the left or to the right, provided that the elements near to the leftmost or rightmost element of $Z_i$ and waiting to be transported to the opposite side of $Z_i$ are ready.

This can be accomplished in $O(\sqrt{\beta_j k}) = O(\beta_j k)$ time by rotating part $A_i$ and sliding the whole $H_i$. We take the first $\sqrt{\beta_j / \beta_i}$ allocation units in $A_i$ and append them to the end of $A_i$. We then completely slide by $\beta_j k$ positions the $O(\sqrt{\beta_i} k) = O(\beta_j)$ chunks in $H_i$. We have also to redirect the pointers to the $\beta_j / \beta_i$ allocation units of $A_i$ that are relocated, in $O(\sqrt{\beta_j / \beta_i} \log n \log \log n)$ time.
Lemma 12.3 Sliding zone $Z_i$ by $O(\beta_i k)$ positions either to the left or to the right (where $j < i$) takes $O\left(\beta_i k + \sqrt{\beta_i / \beta_j} \log n \log \log n\right)$ time in the worst case.

Resizing. We need to move a node from a list to another list internally to $Z_i$ because its size changes by $\sqrt{\beta_i}$ chunks. In this case, we need to exchange the locations of $O(1)$ allocation units in $A_i$, so as to work on the heads of the involved lists. We also need to rotate some of the heads in $H_i$ to reflect the fact that we move a node from the beginning of a list to the beginning of another list. The cost is still $O(\beta_i k)$ time, as each head requires $O(\sqrt{\beta_i})$ time for the rotation, and there are $O(\sqrt{\beta_i})$ heads in $H_i$. The movement of the heads is expensive as we need to identify the parents of the nodes relocated to update their pointers to the relocated children. This requires $O(1)$ searches of cost $O(\log n \log \log n)$ by Lemma 12.2. The total cost is $O(\beta_i k + \log n \log \log n)$ time.

A notable exception is represented by the split and merge operations. A split causes a node of size $4\beta_i$ to be divided into two nodes of size $2\beta_i$ each, and a merge causes two nodes of size $\beta_i$ each to be united in a single node of size $2\beta_i$. The nodes involved in these operations have size (before and after the operations) equal to $\beta_i$, $2\beta_i$, or $4\beta_i$. Being a multiple of the size $\beta_i$ of the allocation units, we do not need to resize $H_i$, but it suffices to perform $O(1)$ searches and change $O(1)$ pointers because we can always keep the heads of these three lists nonempty. (We point out that resizing $H_i$ by $\beta_i$ positions, instead of $\sqrt{\beta_i}$, is too costly to amortize.) In this way, the cost is $O(\beta_i k + \log n \log \log n)$ time.

Lemma 12.4 The cost of resizing a node at level $i$ is $O\left(\beta_i k + \sqrt{\beta_i} \log n \log \log n\right)$ time in the worst case.

Handling the remaining zones. Zone $V$ is easy to handle as its allocation units are single chunks. In zone $Z_h$, we manage $A_h$ analogously to the other $A_i$'s, and $H_h$ in $O(\log n \log \log n)$ time analogously to what is done in [Franceschini, Grossi, Munro, and Pagli, 2002, 2004] (see Chapter 11). Finally, the root in $Z_0$ is rotated one element at time, so its cost of relocation is directly proportional to the number of shifted positions.

12.3.5 Handling insertions

Inserting a new element $x$ follows a downward traversal of the tree like in the search, eventually finding the chunk of a node $u$ in which the element must be inserted. Then we proceed with the following process.

1. We insert $x$ in the chunk and, if $u$ is not a leaf, we remove the largest element in the chunk and insert it in the leftmost (or rightmost, depending on the case) leaf descending from the right pointer encoded in the chunk.
2. In order to make room for one more element in the leaf, which is allocated in zone $Z_h$ (Section 12.3.3), we rotate the root in zone $Z_0$ by one position to the right.

3. We also increase the number of elements of the leaf by 1, and therefore we need to exchange it with the leaf in the head of its compactor list and move it to the head of the target list, updating the layout as described in Section 12.3.4.

4. If the number of elements in the leaf is still less than $4bk + k$, we can conclude the operation at the cost of $O(\log n \log \log n)$ time.

5. Otherwise, if the number of elements becomes $4bk + k$, we have to split the leaf and some of its ancestor nodes.

The splitting of the leaf whose number of elements has increased can trigger a reconstruction process in several contiguous levels of the tree, possibly reaching the root.

The process of restructuring the tree is quite expensive and a possible way to deal with it in the amortized sense is to delay the change of size of the nodes that are ancestors of the leaf. Virtual chunks are introduced for this purpose, and we accumulate enough of them before splitting an internal node (we recall that we do not need virtual chunks in the leaves).

The chain of a node. Before describing this process in some detail below, we need to define the following notion. We say that a given subset $C$ of the chunks $act(v) \cup vir(v)$ associated with a node $v$ is a chain starting at actual chunk $c$ in node $v$ if the following conditions hold:

(i) $C$ is the union of a set of $q$ actual chunks $\{c, c_1, \ldots, c_{q-1}\} \subseteq act(v)$ and of a set of $q - 1$ virtual chunks $\{c_1', \ldots, c_{q-1}'\} \subseteq vir(v)$, for an integer $q > 0$;

(ii) In node $v$, chunk $c_1'$ is the successor of $c$ and the predecessor of $c_1$; chunk $c_j'$ is the successor of $c_{j-1}$ and the predecessor of $c_j$, for $2 \leq j \leq q - 1$.

(iii) Chunk $c_{q-1}$ has no virtual chunk associated with it (i.e., its successor is another actual chunk, if it exists in $v$).

We refer to the number $q$ of actual chunks in $C$ as the length of $C$, denoted by $|C|$. An example of chain is illustrated in Figure 12.1.

We can now focus on the reorganization of an internal node $v$ at level $i$, where $1 \leq i \leq h - 1$, by exploiting the invariant that it satisfies on the number of virtual chunks (see Section 12.3.1). Let $c$ be the chunk that must be inserted into $v$ and has been originated from the split of its child $u$ into two nodes $u'$ and $u''$. We take three possible actions described next.
12.3.5.1 Increasing the number of virtual chunks

This case holds when the number of virtual chunks \( |vir(v)| < 4\sqrt{\beta_i} \) is not maximal. We have room for at least one more virtual chunk, and we do not change the number \( |act(v)| \) of actual chunks. We process with the following actions.

1. We find the actual chunk \( p \) that is the predecessor of \( c \) in node \( v \) by a simple binary search (if \( p \) does not exist, we take the successor with minor modifications).

2. Let us consider the chain \( C = p c'_1 c_1 \ldots c'_q c_q \) starting at \( p \) (by definition of chain, \( c_q \) has no virtual chunk associated). We find the position of \( c \) in \( C \) (either before or after \( c'_1 \)) and obtain a sorted sequence \( S \), which is the sequence of chunks in \( C \) with \( c \) added in its suitable position.

3. At this point, we create a new chain \( C' \) from \( S \) to replace \( C \) with \( C' \) in the memory layout. Namely, the chunks in odd positions become actual and those in even positions become virtual (see Figure 12.3 for an example when \( c \) precedes \( c'_1 \)). The crux of this operation relies in the property that \( |C| = |C'| \), that is, the number of actual chunks in chains \( C \) and \( C' \) does not change (although we change the sets \( act(v) \) and \( vir(v) \) and increase \( |vir(v)| \) by 1).

After this process, we have the invariant on virtual chunks defined in Section 12.3.1 preserved again. Since the total number of chunks involved in chains \( C \) and \( C' \) is at most twice the number of virtual chunks, we have:

**Lemma 12.5** The cost for increasing the number of virtual chunks in a node at level \( i \) is \( O(\sqrt{\beta_i} k) \) time.

12.3.5.2 Increasing the number of actual chunks

This case applies when the number of virtual chunks is maximal \( |vir(v)| = 4\sqrt{\beta_i} \), while the number \( r \leq 4\beta_i - \sqrt{\beta_i} \) of actual chunks is not maximal in node \( v \). We proceed with the following process.
1. We suspend the insertion of $c$ in node $v$.

2. We increase the size of $v$ by $\sqrt{\beta_i}$ actual chunks, which are chosen from the virtual chunks available in $v$.

   (a) We first select $\sqrt{\beta_i}$ virtual chunks in $v$, which are all in zone $V$ as described in Section 12.3.3.

   (b) We then recompact zone $V$ by filling the $\sqrt{\beta_i}$ slots left empty by the selected virtual chunks with other chunks in $V$ (the first $\sqrt{\beta_i}$ remaining ones in it). As a result, we shrink zone $V$ by $\sqrt{\beta_i}$ chunks to the left.

   (c) Now, zone $Z_{h-1}$ preceding zone $V$ has room for sliding $\sqrt{\beta_i} k$ positions to its right as described in Section 12.3.4. We apply an identical sliding to the other zones $Z_{h-2}, Z_{h-3}, \ldots, Z_{i+1}$ in this order.

   (d) We end up with additional room to the right of zone $Z_i$, which has its part $H_i$ enlarged to host the $\sqrt{\beta_i}$ selected chunks. They are turned into actual chunks in node $v$. We then reorganize part $H_i$ as described in Section 12.3.4.

3. After restructuring, node $v$ contains $r + \sqrt{\beta_i}$ actual chunks and $3 \sqrt{\beta_i}$ virtual chunks. Now we can resume the insertion of $c$ in $v$ with the operation described in the previous paragraph, increasing the number of virtual chunks by 1.

For the sake of presentation, we described steps 2b, 2c and 2d as if we move a hole of $\sqrt{\beta_i} k$ free locations up to the zone of the node whose number of actual chunks is increased. That process can be easily modified so that the $\sqrt{\beta_i} k$ elements belonging to the virtual chunks that are going to become actual are moved instead of the hole.

It has to be noted that the relocation of $\sqrt{\beta_i}$ chunks in zone $V$ requires to perform so many searches to locate the nodes having them associated as virtual chunks. We also need to perform further $O(1)$ searches to handle the $O(1)$ relocated nodes in zone $Z_i$.

In order to analyze the cost of this case, we first observe that recompacting zone $V$ requires $O\left(\sqrt{\beta_i} \log n \log \log n\right)$ time as the cost is dominated by the searches (Lemma 12.2). The sliding of each zone $Z_j$, $j > i$, takes $O\left(\beta_i k + \sqrt{\beta_i} / \beta_j \log n \log \log n\right)$ time by Lemma 12.3 (see Section 12.3.4) and is repeated for at most $h = O(\log \log n)$ zones. The internal reorganization of zone $Z_i$ takes $O\left(\beta_i k + \sqrt{\beta_i} \log n \log \log n\right)$ time by Lemma 12.4. So the total cost is $O\left(\beta_i k + \sqrt{\beta_i} \log n (\log \log n)^2\right)$.

**Lemma 12.6** The cost for increasing the number of actual chunks in a node at level $i$ is $O\left(\beta_i k + \sqrt{\beta_i} \log n (\log \log n)^2\right)$ time.
12.3.5.3 Splitting

The last case is when both the actual chunks and the virtual chunks are maximal in number (i.e., \(|act(v)| = 4\beta_i\) and \(|vir(v)| = 4\sqrt{\beta_i}\)). We insert chunk \(c\) in node \(v\), and we take the median chunk \(m\) in the set \(act(v) \cup vir(v) \cup \{c\}\), and propagate it upward to \(v\)'s parent. We then split \(v\) as two nodes \(v'\) and \(v''\), each node with \(2\beta_i\) actual chunks and \(2\sqrt{\beta_i}\) virtual chunks. In general, the actual chunks and the virtual chunks are not so evenly partitioned. Without getting into detail, we redistribute them evenly by in-place merging the chunks in \(act(v) \cup vir(v) \cup \{c\} - \{m\}\). The total cost is \(O(\beta_i k)\) time.

Lemma 12.7 The cost of splitting a node at level \(i\) is \(O(\beta_i k)\) time.

12.4 Fully-Dynamic Implicit Dictionary Handling Deletions

In this section, we describe how to deal with deletions in the data structure presented in Section 12.3. The virtual chunks can mostly help us in handling deletions. Unfortunately, this is not the case when merging two nodes with the chunk \(c\) removed from their parent, and \(c\) is actual with no associated virtual chunk. In this case, we cannot find easily a replacement for that chunk in the parent node, and we must completely reorganize it or change its size (we recall that the invariants are defined in terms of the size of the internal nodes, i.e., their number of actual chunks). This enforces an update of the layout, as described in Section 12.3.4, that we cannot amortize. We introduce patch chunks as replacements when this kind of situation arises.

12.4.1 Extension of the main structure

Given an internal node \(v\), we divide its actual chunks in a set \(seg(v)\) with the following characteristics:
• The members of $\text{seg}(v)$ are segments. Each segment is composed by $\sqrt{\beta_k}$ adjacent chunks of $v$ (recall that the size of $v$ is multiple of $\sqrt{\beta_k}$, i.e., $|\text{act}(v)| = \sqrt{\beta_k} |\text{seg}(v)|$).

• Each segment $z \in \text{seg}(v)$ is further divided into two parts, the leftmost one storing the actual chunks called original and the rightmost one storing the actual chunks called patch (see Figure 12.4). We denote by $\text{orig}(v)$ the set of original chunks in all segments of $v$, and by $\text{pat}(v)$ the set of all patch chunks in those segments. Moreover, we encode in the first chunk of $z$ the last position in $z$ occupied by an original chunk.

• The sets $\text{orig}(v)$ and $\text{pat}(v)$ have their members in sorted order if we take the segments in a left-to-right scan of $v$.

We keep the following invariant on each internal node $v$ and each segment $z \in \text{seg}(v)$ to make binary search possible on the elements stored in $v$:

$$|\text{pat}(v)| < |\text{seg}(v)| + \sqrt{\beta_k}. \quad (12.1)$$

\text{The first chunk of } z \text{ is in } \text{orig}(v) \text{ and the last chunk of } z \text{ is in } \text{pat}(v). \quad (12.2)$$

Since we keep $\text{orig}(v)$ and $\text{pat}(v)$ sorted independently of each other, we associate virtual chunks with each of them separately. In particular, $\text{orig}(v)$ and its virtual chunks satisfy properties (i)-(ii) of the invariant described in Section 12.3.1, as well as do $\text{pat}(v)$ and its virtual chunks. Note that $\text{orig}(v) \cup \text{pat}(v)$ and the union of their virtual chunks do not necessarily satisfy that invariant. When needed, we can run an in-place merge and a redistribution of the chunks to satisfy the invariant globally in the node $v$ at level $i$, at a cost of $O(\beta_k)$ time. We can also define a notion of a chain analogous to that given in Section 12.3.5, applied to either $\text{orig}(v)$ and its virtual chunks or to $\text{pat}(v)$ and its virtual chunks.

### 12.4.1.1 Modifications for routing and searching

In order to route elements in the new organization of node $v$, we perform first a binary search on $\text{orig}(v)$ and its virtual chunks, and then another binary search on $\text{pat}(v)$ and its virtual chunks. As chunks are disjoint, either we end up inside a chunk and we stop the search, or we have to access one child of $v$. That child can be identified by comparing the outcome of the two binary searches above. The total cost of routing stated in Lemma 12.1 remains unchanged. We therefore have:

**Lemma 12.8** Searching an element in the implicit dictionary takes $O(\log n \log \log n)$ worst-case time.
12.4.1.2 Modifications for the insertion

The algorithm for inserting an element in the new organization does not change much. We end up in a leaf as described in Section 12.3.5. At this point, if the leaf splits, we still have three possible actions on its ancestor nodes (see Section 12.3.5).

(a) If the total number of virtual chunks is not maximal, we find a chain either in \( \text{orig}(v) \) or in \( \text{pat}(v) \), and proceed analogously by adding one more virtual chunk.

(b) If the total number of virtual chunks is maximal but the number of actual chunks is not maximal, we proceed as before by adding \( \sqrt{N} \) virtual chunks. Moreover, we turn all the actual chunks in the node into original and distribute them in the segments of the node, except the last chunk in each segment (which remain patch to reflect the invariant in Section 12.4.1).

(c) Splitting does not vary but, again, we turn all the actual chunks in the node into original and distribute them in the segments of the node, except the last chunk in each segment.

It is worth noting that Lemmas 12.5, 12.6, and 12.7 still hold in the new organization of our data structure.

12.4.2 Handling deletions

Deleting an element \( x \) follows a downward traversal of the tree like in the search, eventually finding the chunk of a node \( u \) in which the element must be deleted. We remove \( x \) from the chunk and, if \( u \) is not a leaf, we insert in the chunk the smallest element in the leftmost (or rightmost, depending on the case) leaf descending from the right pointer encoded in the chunk (and delete that element from the leaf). As a result, we decrease the number of elements in a leaf by one and an upward reorganization starts.

1. As the leaf is allocated in zone \( Z_k \) (Section 12.3.3), we rotate the root in zone \( Z_0 \) by one position to the left.

2. We need to exchange the leaf with the node in the head of its list and move then to the head of the target list, updating the layout as described in Section 12.3.4.

3. If the number of elements in the leaf is at least \( bk \), we can conclude the operation at the cost of \( O(\log n \log \log n) \) time.

4. Otherwise, if the number of elements becomes strictly less than \( bk \), we have to merge the leaf with a sibling having \( bk \) elements or share it with a sibling having at least \( (b + 1)k \) elements.
After step 4, we are in the situation in which a chunk \( c \) of an internal node \( v \) must be moved to two children \( u' \) and \( u'' \) so as to form a single child \( u \) of \( v \). We take four possible actions to remove \( c \) from \( v \), as described below.

### 12.4.2.1 Decreasing the number of virtual chunks

This case holds when the number of virtual chunks \( |\text{vir}(v)| > \sqrt{\beta_k} \) is not minimal. Hence, we can remove virtual chunks without changing the number \( |\text{act}(v)| \) of actual chunks. There are more problems than in the case of the insertion in order to maintain the invariant for actual and virtual chunks, since the merging nodes below may be needing the deletion of an actual chunk without any virtual associated. We proceed in the following way.

1. If \( c \) is virtual, we can safely remove it from \( v \).

2. If \( c \) is actual (original or patch) and has a virtual chunk \( c' \) associated, we can remove \( c \) and replace it with \( c' \), which becomes actual in \( v \).

3. We have a problem when \( c \) is actual but has no virtual chunk associated. We have two cases.
   
   (a) i. If \( c \) is patch, we remove it and replace with any virtual chunk \( c' \) of \( v \).
   
   ii. Then, we insert \( c' \) in its ordered position in \( \text{pat}(v) \).
   
   iii. Finally, we reorganize the whole set \( \text{pat}(v) \). Since it contains at most \( |\text{pat}(v)| < |\text{seg}(v)| + \sqrt{\beta_k} = O(\sqrt{\beta_k}) \) chunks, we can (in-place) redistribute them in the segments of \( v \) to account for the removal of \( c \) and the insertion of \( c' \) in the set.
   
   (b) i. If \( c \) is original, we take the segment \( z \) containing \( c \), and slide \( c \) to the end of the original chunks in \( z \), so that \( z \) contains first all the original chunks except \( c \), then \( c \), and finally all the patch chunks.

   ii. At this point, we remove \( c \) from \( z \) and replace it with any virtual chunk \( c' \) of \( v \).

   iii. We then turn \( c' \) into patch, and reorganize the whole set \( \text{pat}(v) \) as described in case 3a.

Note that, in case 3b, we increase the number of patch chunks in \( v \) by 1. If invariant (12.1) is violated in this way, we reorganize the whole node \( v \), merging (in-place) the original and the patch chunks and their virtual chunks, and redistributing the chunks in the segments of \( v \), so that we preserve the number of virtual chunks and actual chunks, and we only leave one patch chunk in the last position of each segment to maintain the invariants (12.1) and (12.2) in Section 12.4.1. This reorganization takes \( O(\beta_k) \) time.

**Lemma 12.9** The cost for decreasing the number of virtual chunks in a node at level \( i \) is \( O(\sqrt{\beta_k}) \) time. If invariant (12.1) is violated, the cost is \( O(\beta_k) \) time.
12.4.2.2 Decreasing the number of actual chunks

This case applies when the number of virtual chunks is minimal \(|\text{vir}(v)| = \sqrt{\beta_i}\), while the number \(r \geq \beta_i + \sqrt{\beta_i}\) of actual chunks is not minimal in node \(v\). We proceed as follows.

1. We suspend the deletion of \(c\) in node \(v\), and we decrease the size of \(v\) by \(\sqrt{\beta_i}\) actual chunks, which become virtual chunks in \(v\). Making \(\sqrt{\beta_i}\) actual chunks virtual, requires a reorganization of the memory layout symmetrical to what discussed in Section 12.3.5.2.

2. After restructuring, node \(v\) contains \(r - \sqrt{\beta_i}\) actual chunks and \(2\sqrt{\beta_i}\) virtual chunks. Now we can resume the deletion of \(c\) in \(v\) with the operation described in Section 12.4.2.1, decreasing the number of virtual chunks by 1.

Lemma 12.10 The cost for decreasing the number of actual chunks in a node at level \(i\) is \(O\left(\beta_i k + \sqrt{\beta_i} \log n (\log \log n)^2\right)\) time.

12.4.2.3 Merging

This case holds when both the actual chunks and the virtual chunks are minimal in number (i.e., \(|\text{act}(v)| = \beta_i\) and \(|\text{vir}(v)| = \sqrt{\beta_i}\)), and there is a sibling \(v'\) of \(v\) that satisfies the same conditions on its chunks. Let \(c'\) be the chunk in the common parent that separates \(v\) from \(v'\). We recursively remove \(c'\) from the parent, and merge (in-place) \(c'\) and all the chunks (actual and virtual) in \(v'\) and in \(v\), except \(c\). We create a single node from \(v\) and \(v'\), and redistribute the sorted chunks in the segments of the new node, so that we assign \(2\sqrt{\beta_i}\) virtual chunks and \(2\beta_i\) actual chunks, leaving one patch chunk in the last position of each segment to maintain the invariants (12.1) and (12.2) in Section 12.4.1.

Lemma 12.11 The cost of merging two sibling nodes at level \(i\) is \(O(\beta_i k)\) time.

12.4.2.4 Sharing

The last case applies whenever none of the previous three cases holds. Namely, both the actual chunks and the virtual chunks in \(v\) are minimal in number (i.e., \(|\text{act}(v)| = \beta_i\) and \(|\text{vir}(v)| = \sqrt{\beta_i}\)), but any of its sibling \(v'\) (say to the right) has more than \(\beta_i + \sqrt{\beta_i}\) chunks. Let \(c\) be the chunk in the common parent that separates \(v\) from \(v'\). We proceed in the following way.

1. We remove \(c\) from \(v\) performing the same operations described in the first case (decreasing the number of virtual chunks). After that, \(v\) has \(\sqrt{\beta_i} - 1\) virtual chunks.
2. We then identify the minimal chunk $c'$ (actual or virtual) in $v'$, and remove it from $v'$ according to one of the first two cases previously discussed (decreasing the number of virtual chunks, or decreasing the number of actual chunks).

3. We replace $c$ with $c'$ in the common parent of $v$ and $v'$ (note that it is a mere substitution, no need to reorganize the parent).

4. We then insert $c'$ in $v$ as described in the first case (increasing the number of virtual chunks) of Section 12.3.5.

As a result of the above process, $v'$ has one less chunk, the common parent of $v$ and $v'$ does not change the number of its actual (original and patch) chunks and their virtual chunks, satisfying the invariants (12.1) and (12.2) of Section 12.4.1. Finally, $v$ has $\sqrt{\beta_i}$ virtual chunks and $\beta_i$ actual chunks. The cost of sharing depends on how $c'$ is removed from $v'$:

**Lemma 12.12** Sharing two sibling nodes at level $i$ has the same time complexity of either decreasing the number of virtual chunks (Lemma 12.9) or decreasing the number of actual chunks (Lemma 12.10).

Being more explicit, the time bound implied by Lemma 12.12 is

1. $O\left(\sqrt{\beta_i}k\right)$ when $v$ borrows a chunk from $v'$ by decreasing the number of virtual chunks in $v'$ under invariant (12.1);

2. $O\left(\beta_i k\right)$ when $v$ borrows a chunk from $v'$ by decreasing the number of virtual chunks in $v'$ under invariant (12.2);

3. $O\left(\beta_i k + \sqrt{\beta_i} \log n (\log \log n)^2\right)$ when $v$ borrows a chunk from $v'$ by decreasing the number of actual chunks in $v'$.

### 12.5 Amortized Analysis and Conclusions

In this section, we provide the amortized analysis of the update operations supported by our data structure $T$. We use a potential function $\Phi(T) = \sum_{v \in T} \Phi(v)$ defined as the sum of the potential on each node $v$:

$$\Phi(v) = \begin{cases} 
\mathcal{A}(v) + \mathcal{Y}(v) + \mathcal{P}(v) & v \text{ node at level } i \geq 1 \\
\mathcal{L}(v) & v \text{ leaf}
\end{cases}$$

Let $\gamma = k \log n \log \log n$ be the “potential” of a chunk of $k$ elements. In the formula for $\Phi(v)$, we denote by $\mathcal{A}(v)$ the potential provided by the actual chunks, having value $c_1 \beta_i \gamma$ for a constant $c_1$ when the actual chunks are minimal or maximal, and having value 0 when they are half on the way ($|\text{act}(v)| = 2\beta_i$):
\[ \mathcal{A}(v) = \begin{cases} \ c_1 (-|act(v)| + 2\beta_i) \gamma & |act(v)| \leq 2\beta_i \\ \ c_1 \left( \frac{1}{2} |act(v)| - \beta_i \right) \gamma & |act(v)| > 2\beta_i \end{cases} \]

We denote by \( \mathcal{V}_i(v) \) the potential provided by the virtual chunks, having value \( c_2 \beta_i \gamma \) for a constant \( c_2 \) when the virtual chunks are minimal or maximal, and having value 0 when they are half on the way (\( |vir(v)| = 2\sqrt{\beta_i} \)):

\[ \mathcal{V}_i(v) = \begin{cases} \ c_2 (\sqrt{\beta_i} |vir(v)| + 2\beta_i) \gamma & |vir(v)| \leq 2\sqrt{\beta_i} \\ \ c_2 \left( \frac{1}{2} \sqrt{\beta_i} |vir(v)| - \beta_i \right) \gamma & |vir(v)| > 2\sqrt{\beta_i} \end{cases} \]

We denote by \( \mathcal{P}_i(v) \) the potential for the patch chunks having value 0 when they are minimal (\( |pat(v)| = |seg(v)| \)) and value \( c_3 \beta_i \gamma \) for a constant \( c_3 \) when they are maximal:

\[ \mathcal{P}_i(v) = c_3 \sqrt{\beta_i} \gamma (|pat(v)| - |seg(v)|) \]

Finally, we have the potential \( \mathcal{L}(v) \) for each leaf \( v \), having value \( c_4 \gamma \) for a constant \( c_4 \) when the number of elements are minimal or maximal, and having value 0 when they are half on the way (i.e., \( 2bk \) elements):

\[ \mathcal{L}(v) = \begin{cases} \ c_4 \left( \frac{1}{bk} |v| + 2 \right) \gamma & |v| \leq 2bk \\ \ c_4 \left( \frac{1}{2bk + 1} |v| - \frac{2bk}{2bk + 1} \right) \gamma & |v| > 2bk \end{cases} \]

**Theorem 12.1** Searching an element in the implicit data structure requires a worst case time \( O(\log n \log \log n) \). Inserting or deleting an element takes \( O(\log n \log \log n) \) amortized time.

**Proof:**

Let us consider an arbitrary sequence of \( n \) insert and delete operations. Taking the first \( s \leq n \) operations into account, we can observe that \( \Phi(T_s) \geq 0 \) by definition of \( \Phi \), where \( T_s \) denotes the data structure resulting from applying the \( s \) operations to the initial empty data structure \( T_0 \). What we have to prove is that the amortized cost of the \( s \)th operation is

\[ \hat{t}_s = t_s + \Phi(T_s) - \Phi(T_{s-1}) = O(\log n \log \log n), \]

where \( t_s \) denote the actual cost of the \( s \)th operation.

We do not discuss all possible cases here, but we choose one of them and present its analysis. (The rest of the cases can be treated analogously.)
Suppose that the \( s \)th operation inserts an element into \( T_{s-1} \) causing a sequence of splits from a leaf \( l \) to a child of node \( v \) (say, at level \( i \)), where we stop splitting as we can increase the number of actual chunks of \( v \) (see Section 12.3.5). In the following, we denote by \( v_j, i < j < h \), the intermediate nodes along the path from \( v \) to \( l \), where each \( v_j \) is split into \( v'_j \) and \( v''_j \).

In order to evaluate the difference in potential, \( \Phi(T_s) - \Phi(T_{s-1}) \), produced by the \( s \)th operation in this case, we observe that the leaf \( l \), with maximal number of elements, is split into \( l' \) and \( l'' \), each with \( 2bk \) elements. This gives a negative contribution of

\[
\Phi(l) - (\Phi(l') + \Phi(l'')) = c_4 \gamma = O \left( \log^2 n \log \log n \right),
\]

since \( \Phi(l') = \Phi(l'') = 0 \).

For the intermediate nodes \( v_j \), we have an analogous situation, where \( |act(v_j)| = 4\beta_j \) and \( |vir(v_j)| = 4\sqrt{\beta_j} \), giving a negative contribution of

\[
\Phi(v) - \left( \Phi(v'_j) + \Phi(v''_j) \right) = (c_1 + c_2)\beta_j \gamma + \mathcal{P}_j(v_j),
\]

where \( \Phi(v'_j) = \Phi(v''_j) = 0 \) and \( \mathcal{P}_j(v_j) \geq 0 \).

Finally, for node \( v \) in \( T_{s-1} \), we have \( |vir(v)| = 4\sqrt{\beta_i} \) and \( |act(v)| = r \leq 4\beta_i - \sqrt{\beta_i} \) since we can increase the number of actual chunks by hypothesis. Assuming without loss of generality that \( r \geq 2\beta_i \), it follows that

\[
\Phi(v) = c_1 \left( \frac{1}{2}r - \beta_i \right) \gamma + c_2 \beta_i \gamma + \mathcal{P}_i(v).
\]

After the \( s \)th operation, \( v \) is modified and we denote the resulting node by \( v' \) in \( T_s \). Note that \( |act(v')| = r + \sqrt{\beta_i} \) and \( |vir(v')| = 3\sqrt{\beta_i} \). Hence,

\[
\Phi(v') = c_1 \left( \frac{1}{2}(r + \sqrt{\beta_i}) - \beta_i \right) \gamma + c_2 \left( \frac{3}{2} \beta_i - \beta_i \right) \gamma - \mathcal{P}_i(v').
\]

Since \( \mathcal{P}_i(v) = \mathcal{P}_i(v') \), we get a negative contribution of

\[
\Phi(v) - \Phi(v') = \frac{1}{2} \gamma \left( c_2 \beta_i - c_1 \sqrt{\beta_i} \right) = \Omega \left( \beta_i \log^2 n \log \log n \right).
\]

The changes in the potential described above are the only ones performed when transforming \( T_{s-1} \) into \( T_s \). We can bound the total change in terms of the sum of the negative contribution by

\[
\Phi(T_{s-1}) - \Phi(T_s) > \Phi(v) - \Phi(v').
\]

It remains to evaluate \( t_s \) by summing the costs in Lemma 12.2, Lemma 12.6 and Lemma 12.7, obtaining

\[
t_s = O \left( \beta_i kn + \sqrt{\beta_i} \log n (\log \log n)^2 \right),
\]
so that we can conclude that

\[
\hat{t}_s = t_s - (\Phi(T_{s-1}) - \Phi(T_s)) < t_s - (\Phi(v) - \Phi(v')) = O(\log n \log \log n).
\]

by choosing suitably \(c_1\) and \(c_2\).

\[\square\]

**Conclusions.** In this chapter we presented an implicit dictionary for the RAM model (see Chapter 2) that is searchable in \(O(\log n \log \log n)\) time in the worst case and is updatable in \(O(\log n \log \log n)\) time in amortized sense. The presentation in this chapter is based on the paper [Franceschini and Grossi, 2003a] (SODA 2003) and at the time of the publication this result was a rather big improvement in performance with respect to the AVL Implicit Tree [Munro, 1986] \(O(\log^2 n)\) for searching and updating) and the Implicit B-Tree [Franceschini, Grossi, Munro, and Pagli, 2002, 2004] \(O(\log^2 n / \log \log n)\) for searching and updating, see Chapter 11). Besides the bounds, the important aspects of this result are the successful use of amortized analysis in the implicit dictionary problem and the fact that the basic strategy of reducing the levels of the tree while still paying a full slowdown of \(O(\log n)\) for each level, instead of trying to pay less decoding costs when passing from a level to the next one in a taller tree, is a rewarding strategy. As we will see in the following chapters, the former will we the right strategy to finally solve the implicit dictionary problem.
Chapter 13

Cache-Obliviousness vs. Implicitness (i.e. Locality vs. Compactness)

Abstract

A sequence of $n$ distinct elements can be sorted for logarithmic searching or can be organized as a heap for logarithmic updating, but it is unclear how to attain logarithmic time for both searching and updating. This natural question dates back to the heap of Williams in the sixties and relates to the fundamental issue whether additional space behind the elements gives more computational power in dictionaries and how data ordering helps. Implicit data structures have been introduced in the eighties with this goal. One finding in this chapter is that of closing the above long-standing issue with the Flat Implicit Tree, which is the first optimal data structure obtaining $O(\log n)$ time in the worst case for search and $O(\log n)$ time in amortized sense for insertions and deletions in an array of $n$ locations. Another finding is motivated by the antithetic features of implicitness and cache-obliviousness in data structures. The flat implicit tree combines their best qualities achieving $O(\log n)$ comparisons and $O(\log_B n)$ block transfers for any block size $B$, where the value of $B$ and the capacity of the memory level are unknown to the algorithms operating in the model (the bounds for the updates are in amortized sense). As a result, we avoid space wasting by just using the plain array of $n$ locations for the elements and, simultaneously, we provide optimal data access at any level of the memory hierarchy.

The presentation in this chapter is based on the paper [Franceschini and Grossi, 2003b] (ICALP 2003).
13.1 Introduction

A sequence of \( n \) distinct elements can be sorted for logarithmic searching or can be organized as a heap for logarithmic updating, but it is unclear how to attain logarithmic time for both searching and updating. This natural question dates back to the heap of Williams in the sixties and relates to the fundamental issue whether additional space behind the elements gives more computational power in dictionaries and how data ordering helps. Implicit data structures have been introduced in the eighties with this goal. In this chapter we are going to face again the implicit dictionary problem. As we have seen, before the introduction of the Implicit B-Tree (see [Franceschini, Grossi, Munro, and Pagli, 2002], [Franceschini, Grossi, Munro, and Pagli, 2004] and Chapter 11) the best upper bound known for this problem has been the one in [Munro, 1986] (for the history of the research in this field before [Munro, 1986] see Chapters 3 and 11). The Implicit B-Tree has also the merit of being the first implicit dictionary for a multi-level memory model (in this particular case the External-Memory model, see Chapter 2). After the Implicit B-Tree, the research turned again toward the RAM model and the fundamental computational problem of the existence of an implicit dictionary featuring search an update in \( O(\log n) \) time. The Exponential Implicit Tree (see [Franceschini and Grossi, 2003a] and Chapter 12) comes close to this objective being searchable in \( O(\log n \log \log n) \) in the worst case and updatable in \( O(\log n \log \log n) \) time in amortized sense.

In this chapter we will have two major findings. The first finding is that of closing the above long-standing issue with the Flat Implicit Tree, which is the first optimal data structure obtaining \( O(\log n) \) time for search and update in an array of \( n \) locations.

The second finding is motivated by the antithetic features of implicitness and cache-obliviousness in data structures. The Flat Implicit Tree combines their best qualities achieving \( O(\log n) \) comparisons and \( O(\log_B n) \) block transfers for any block size \( B \), where the value of \( B \) and the capacity of the memory level are unknown to the algorithms operating in the model. As a result, we avoid space wasting by just using the plain array of \( n \) locations for the elements and, simultaneously, we provide optimal data access at any level of the memory hierarchy.

13.1.1 Optimal implicit dictionary in the RAM model

The implicit dictionary problem is intimately related to the fundamental question whether using extra space gives more computational power than using exactly \( n \) locations, and how data ordering can help in this task. The ultimate goal is in the flavor of the theoretical result of Yao [Yao, 1984], but in a dynamic setting and with an unbounded universe (see Chapters 2 and 3):
**Implicit dictionary problem.** For a given model of computation \( \mathcal{M} \) and a universe \( \mathcal{U} \), is there a space optimal update algorithm \( U \) to maintain a dynamic set \( S \subseteq \mathcal{U} \) such that:

- Only \( |S| \) locations plus \( O(1) \) values of auxiliary information of \( O(\log |S|) \) bits each are used.
- \( S \) can be searched optimally (with respect to model \( \mathcal{M} \)).
- \( U \) is space optimal and cost optimal (with respect to model \( \mathcal{M} \)).

In this chapter, we give a positive answer to the long-standing question above with respect to two models: the RAM model and the Cache-Oblivious model (see our next finding). We describe an implicit data structure, called Flat Implicit Tree (that is F.I.T., please appreciate the wonderful joke), which, in the RAM model, requires \( O(\log n) \) time for searching and \( O(\log n) \) amortized time for updating, with just \( O(1) \) auxiliary locations.

An obvious feature of our data structure, being implicit, is that it achieves the best space saving possible within optimal time bounds. In this context self-adjusting search trees [Sleator and Tarjan, 1985] and random search trees [Martínez and Roura, 1998] are often mentioned for their ability, among others, to avoid balancing information for saving space while requiring \( O(\log n) \) amortized or expected time. However, they still need \( n + \Omega(n) \) locations of memory (for the pointers).

As in [Franceschini and Grossi, 2003a] (see Chapter 12), we face the issue of amortization in implicit data structures. It should be noted that several data structuring techniques requiring the duplication of elements cannot work in the implicit model.

- Overmars’ rebuilding technique [Overmars, 1983] maintains a copy of the data structure.
- Arge’s buffering technique [Arge, 1995] should keep a copy of the element to be deleted or a pointer to it.

Moreover, during the amortization, we can use only in-place algorithms as otherwise the data structures would not require \( O(1) \) extra locations to operate but also a non-constant temporary area, which makes no sense in the implicit model. As a result, our deletion algorithm is almost symmetrical to the insertion.

The reader may notice a greater level of details than in other works on data structures. This is probably due to the fact that not only we have to describe the algorithms for the supported operations, but we have to carefully handle the memory layout of the elements in the array.

Our solution hinges on a hybrid approach using the partial order fixed \textit{a priori} in sorted arrays to handle a large segment of quasi-sorted data (the super-root), and the flexible encoding of [Munro, 1986] (see Chapter 4) to handle a large collection
of relatively small buckets of permuted elements (the intermediate nodes and the leaves). We obtain a constant depth with $O(\log n)$ accessed elements per traversed node, which is the problem left open in the Exponential Implicit Tree [Franceschini and Grossi, 2003a] (see Chapter 12). For this, we introduce new ideas to maintain a flat data structure with just three levels of nodes: a super-root, a level of large intermediate nodes, and a level of leaves augmented with spare elements. We also use variant of techniques previously introduced in [Itai, Konheim, and Rodeh, 1981, Bender, Demaine, and Farach-Colton, 2000] (see Chapter 4).

13.1.2 Introducing the issue of cache-obliviousness in implicit dictionaries

The dictionary problem can be faced from another perspective, namely, that of cache-oblivious B-trees [Bender, Demaine, and Farach-Colton, 2000] and other dictionaries in the cache-oblivious (memory-hierarchy) model [Frigo, Leiserson, Prokop, and Ramachandran, 1999]. In this ideal model there is a hierarchy with two levels of memory, where one level is small and fast and the other level is large but slow. Data transfers between the two levels occur in blocks of $B$ data items; however, the value of $B$ and the capacity of the fast memory are unknown to the algorithms operating in the model.

In [Frigo, Leiserson, Prokop, and Ramachandran, 1999], the model is shown to simulate any realistic model with two or more memory levels with just a small constant-factor slowdown. As in [Bender, Demaine, and Farach-Colton, 2000], we do not make any assumption on the capacity of the memory levels. It is demonstrated there how to perform the dictionary operations in $O(\log_B n)$ block transfers for searching and $O(1 + r/B)$ block transfers for scanning $r$ contiguous elements like regular B-trees, while updates take $O(\log_B n + \log^2 n/B)$ amortized block transfers. If scans are not required, the updates take $O(\log_B n)$ amortized block transfers.

Other results in this direction have obtained simplifications [Bender, Duan, Iacono, and Wu, 2002d, Brodal, Fagerberg, and Jacob, 2002] and worst-case bounds for the updates [Bender, Cole, and Raman, 2002b], and have avoided the use of pointers [Brodal, Fagerberg, and Jacob, 2002] by keeping all elements stored into an array of size $n + \Theta(n)$. More references to cache-oblivious algorithms and data structures are reported in the survey [Demaine, 2002].

The motivation for our second result arises from the observation that the solutions proposed so far to the dictionary problem are somehow antithetic in the implicit model and in the Cache-Oblivious model.

- Implicit dictionaries entirely encode the information by permuting the elements in $n$ contiguous memory locations with no additional location, apparently loosing data locality. Due to the dynamic maintenance of the permutation without wasting memory locations, the algorithms give rise to irregular access patterns jumping from one memory location to another.
• Cache-oblivious dictionaries carefully handle irregular access patterns to data for getting locality at the price of wasting $\Theta(n)$ memory locations. In order to preserve dynamically the locality of elements, these are suitably interleaved with empty locations whose purpose consists in delaying more expensive redistributions that have a small amortized cost in this way.

In the context of these studies it is an interesting theoretical question to see whether the appealing features of the two models can be blend together by avoiding their contrasting drawbacks mentioned above. We investigate the possibility of designing an optimal dictionary that is simultaneously cache-oblivious and implicit, with a major focus on searching and updating. We show that the Flat Implicit Tree requires $O(\log n)$ comparisons and $O(\log_B n)$ block transfers for searching, and $O(\log n)$ comparisons and $O(\log_B n)$ amortized block transfers for updating, while using just $n$ memory locations. Not only we avoid space wasting, but we provide optimal data access at any level of the memory hierarchy.

A part of the flat implicit tree is inspired by the van Emde Boas layout of [Prokop, 1999] modified in a non-trivial way with further implicit techniques. Compared to previous work, the data structure we are going to introduce in this chapter is complementary to that in [Franceschini, Grossi, Munro, and Pagli, 2002] (see Chapter 11), as in that case the bounds hold in the worst case and use the cache-aware model; moreover, scanning $r$ elements takes $O(\log_B n + r/B)$ block transfers.

It is worth noting that the pointerless data structure in [Brodal, Fagerberg, and Jacob, 2002] is cache-oblivious but, as noted by its authors, it is not properly implicit as it occupies $(1 + \epsilon)n$ locations for any $\epsilon > 0$. As a final remark, our data structure does not support efficient scanning, but this is also an open problem in cache-oblivious data structures alone when the bounds are $O(\log_B n)$ as in our case.

**Chapter organization.** This chapter is organized as follows. Section 13.2 gives an overview of the flat implicit tree organized in two layers, with Section 13.3 describing the bottom layer, Section 13.4 giving a description of the top layer, while Section 13.5 discusses how to make the top layer cache-oblivious. We put all together in Section 13.6 for our final bounds.

### 13.2 Overview of the Flat Implicit Tree

We first give some preliminaries. We encode data by a pairwise permutation of elements as the bit stealing technique in [Munro, 1986] (see Chapter 4). To encode a pointer or an integer of $b$ bits by using $2b$ distinct elements $x_1, y_1, x_2, y_2, \ldots, x_b, y_b$, we permute them in pairs $x_i, y_i$ by the known rule: if the $i$th bit is 0, then $\min\{x_i, y_i\}$ precedes $\max\{x_i, y_i\}$; else, the bit is 1 and $\max\{x_i, y_i\}$ precedes $\min\{x_i, y_i\}$. Although $x_1, y_1, x_2, y_2, \ldots, x_b, y_b$ are permuted, they are quasi sorted. Any algorithm (e.g., binary search) that requires them sorted still works with minor modifications.
Figure 13.1: An actual chunk $c$ with its associated bucket $b_c$ of elements, where $c$ is the root of $b_c$. The virtual chunks $c_1, c_2, c_3$ are associated with $c$, and their corresponding buckets are $b_1, b_2, b_3$. Note that the elements in $c, c_1, c_2, c_3$ will be stored in the root area of Figure 13.2, while the elements in $b_c, b_1, b_2, b_3$ will be suitably stored in the node area, manipule area and spare area shown in Figure 13.2.

Chunks and spare elements. From a high-level point of view, the flat implicit tree is a suitable collection of chunks [Munro, 1986, Franceschini, Grossi, Munro, and Pagli, 2002, Franceschini and Grossi, 2003a] and spare elements. Each chunk contains $k$ elements that are pairwise permuted for encoding a constant number of integers and pointers, each of $b = O(\log n)$ bits. The elements in any chunk belong to a certain interval of values, and the chunks are pairwise disjoint when considered as intervals. This allows us to define a total order on any set of the chunks, so that we can write $c_1 < c_2$ for any two chunks meaning that the elements in chunk $c_1$ are all smaller than those in chunk $c_2$.

Not all the elements are stored in the chunks. A number of $O(\log n)$ elements are kept in a preamble $\mathcal{P}$ to encode some bookkeeping information. The remaining ones are the spare elements, which are kept together in a contiguous segment of memory without a particular organization.

The epochs. Differently from previous work on implicit data structures, we keep the invariant that the chunk size is

$$k = \Theta(\log n'),$$

where $n'$ is a power of two satisfying $n'/4 < n < n'$. The constant hidden in the notation for $k$ is sufficiently large to encode all the structural information for the flat implicit tree; as it should be clear in the rest of the chapter, the amount of this information is independent from $n$, hence, a constant.

Using $n'$ for fixing $k$ is important as it avoids to change $k$ each time that $n$ changes. We can avoid to keep the value of $n$ and $n'$ explicitly, as a variable-
length encoding, such as the δ-code [Witten, Moffat, and Bell, 1999], can represent
them asymptotically in the preamble $\mathcal{P}$. The algorithms for maintaining the data
structure are parametric in $n'$ and $k$. We resize $k$ only when either

$$n = n'/4 \quad \text{or} \quad n = n'$$

This event marks the beginning of a new epoch. Hence, the lifetime of the data
structure can be divided into epochs. At the beginning of each epoch, we start the
process of rebuilding our implicit dictionary to guarantee that $n'/4 < n < n'$, which
is described in Section 13.6.

The two-layer organization. After each rebuilding, the $n$ distinct elements are
organized in two layers as shown in Figure 13.1:

- The top layer is the super-root which contains

  $$a = \Theta(n/k^2)$$

  chunks, called actual chunks, where $a$ is always a power of two. Also, it
  contains a number of virtual chunks stored in no particular order. As described
  in Section 13.4, we have $O(1)$ virtual chunks associated with each actual chunk.

  Specifically, for an actual chunk $c$, we require that $c$ and its associated virtual
  chunks are consecutive in the total order defined over all the chunks in the
top layer. Cache-obliviousness in this layer makes use of the static scheme of
[Prokop, 1999] but with the additional requirement of employing in-place
algorithms to achieve implicitness as described Section 13.5.

- The bottom layer is a dynamic implicit forest, where each tree implements a
  bucket of $\Theta(k^2)$ elements organized into chunks, called bucket chunks, plus the
  spare elements being handled differently. Each bucket tree is organized into
  $O(1)$ levels. The root is either an actual chunk or a virtual chunk in the top
  layer. The descending elements of the root are in the bottom layer.

  Specifically, the root has a single child, called intermediate node, containing
  $\Theta(\sqrt{k})$ bucket chunks and $\Theta(\sqrt{k})$ leaves as children. Each such leaf has
  associated a manage of $\Theta(k)$ elements with further $\Theta(\sqrt{k})$ spare elements.
The buckets are pairwise disjoint when considered as intervals. (The latter
property is useful for routing elements inside a bucket.) More details are in
Section 13.3.

The actual chunks are in a sorted array of the top layer. While searching an
element in this two-layer organization we can exploit their order to access the asso-
ciated virtual chunks. As a result, we can quickly identify the bucket of elements
that might contain a given element. Searching follows this idea by performing first
a binary search on the actual chunks (one element per chunk is enough). Once an actual chunk \( c \) is identified, we can retrieve its associated \( O(1) \) virtual chunks (if they exist) and, among their \( O(1) \) buckets, we can identify the bucket in which to perform the search. We aim at decoding integers and pointers from a total of \( O(\log n) \) elements, so that the total cost is \( O(\log n) \) time.

As customary to implicit data structures, we provide a view of the memory layout of the flat implicit tree as this heavily affects the complexity of the operations. We have \( O(1) \) contiguous areas shown in Figure 13.2:

- The preamble \( \mathcal{P} \) of \( O(k) \) elements encoding \( O(1) \) pointers and integers for bookkeeping purposes.
- The root area for the top layer storing first the elements in the actual chunks in a suitable order and, then, the virtual chunks in no particular order. Recall that these chunks are the roots of the bucket trees. The area may grow or shrink by \( k \) positions to its right.
- The area for the bottom layer is divided into
  - node area, for the intermediate nodes and the leaves,
  - manipulate area and spare area, for the bucket trees.

The whole area for the bottom layer may grow or shrink by \( k \) positions to its left and by one position to its right.

Compared to the compactor lists in [Franceschini, Grossi, Munro, and Pagli, 2002, Munro, 1986], our memory management introduces the compactor zones for handling the above areas as each compactor zone stores objects of the same size in a contiguous segment of memory, which is crucial to achieve our bounds. Although both compactor lists and zones succeed in storing “objects” of variable size without memory waste, a compactor zone is more suitable for cache-obliveness. On the other hand, the compactor zones offer update bounds that are inherently amortized, as opposed to the compactor lists that are very suitable for worst case bounds. As we will see in Chapter 15, we will need both these techniques to achieve optimal worst case bounds.

The bounds for the resulting data structure are stated in the following theorem.
Theorem 13.1: There exists a dynamic data structure storing $n$ distinct elements that is both implicit and cache-oblivious and that supports the following operations using just $O(1)$ auxiliary locations:

- Searching with a cost of $O(\log n)$ time and of $O(\log_B n)$ block transfers per operation;
- Inserting and deleting with an amortized cost of $O(\log n)$ time and of $O(\log_B n)$ block transfers.

We prove Theorem 13.1 in the rest of the chapter. In what follows, the bounds are intended to hold in the worst case, unless we explicitly mention them as amortized bounds. We define the size of the several entities (nodes, areas, zones, etc.) as the number of elements contained in each of them.

### 13.3 Bottom Layer: Buckets as Implicit Dynamic Forest

We describe the bottom layer storing the buckets in the form of an implicit dynamic forest. Each bucket is a tree whose root is either an actual or a virtual chunk in the top layer. We need to insert, to delete and to search an element in a bucket, as required by the dictionary problem. In case of bucket overflows (too many elements) and bucket underflows (too few elements), we need splitting the bucket or merging it with a neighbor bucket, while preserving implicitness and cache-obliviousness.

#### 13.3.1 Bucket organization

**Invariants for the buckets.** As previously mentioned, each bucket is a tree made up of bucket chunks and spare elements, except for the root, which is either an actual or a virtual chunk. The leaves vary by a chunk at a time and contain from $k\sqrt{k}$ to $4k\sqrt{k}$ elements (i.e., from $\sqrt{k}$ to $4\sqrt{k}$ bucket chunks). Each intermediate node is the only child of its root and its size varies by a chunk at a time. The pointer to the $i$th child leaf is encoded by $O(\log n')$ elements in the $i$th chunk of the intermediate node.

To bound the number of chunks (hence, of children) of an intermediate node $u$, we follow the approach of weight-balanced trees (e.g., [Arge and Vitter, 1996]) defining the weight $w(u)$ as the number of elements in the leaves that are children of $u$. We keep on $u$ the invariant

$$4k^2 \leq w(u) \leq 16k^2.$$

As a result, the number of chunks in $u$ ranges from its minimum weight divided by the largest number of elements in a leaf to its maximum weight divided by the smallest number of elements in a leaf, that is, from $\sqrt{k}$ to $16\sqrt{k}$ chunks.
In addition, each leaf has associated a maniple varying by $\sqrt{k}$ elements at a time, thus containing from $k$ to $5k$ elements that are greater than those in the leaf. We call a group of $\sqrt{k}$ contiguous elements a mini-chunk, so that $\sqrt{k}$ mini-chunks give rise to a single chunk (also mini-chunks are disjoint when considered as intervals). Hence, each maniple varies by a mini-chunk at a time.

In order to manage the insertion and deletion of individual elements, we associate the $\Theta(\sqrt{k})$ spare elements with the last $\sqrt{k}$ mini-chunks in each maniple. From 1 to 5 spare elements are associated with each mini-chunk, and they are contained in the interval represented by the mini-chunk. However, we cannot encode in $z$ the pointers to the $\Theta(\sqrt{k})$ spare elements in $z$. We encode them in the first $\Theta(\sqrt{k})$ chunks of the corresponding leaf $v$, which is sufficiently large to allow this encoding with its elements.

A sketch of how invariants are maintained. We keep the above invariants on the number of elements during the updates. We show later that inserting an element is tantamount to incrementing by 1 the number of spare elements in a maniple $z$. If a mini-chunk in $z$ has associated 6 spare elements, we suitably redistribute the elements in $z$. If $z$ has associated a total of $5\sqrt{k} + 1$ spare elements, we extract $\sqrt{k}$ spare elements and transform them into a mini-chunk that is added to $z$. However, if $z$ has its maximum capacity of $5k$ elements, we extract the smallest chunk in $z$ inserting it as the largest chunk in its corresponding leaf $v$.

If $v$ contains $4k\sqrt{k}$ elements, we go on splitting both $v$ and $z$, creating two leaves with their associated maniples and spare elements. All satisfy the invariants on their number of elements, which is roughly half on the way between the minimum and maximum allowed. The split may cause a chunk $c$ to be inserted in the intermediate node $u$, parent of $v$.

If $u$ has maximum weight $w(u) = 16k^2$, we need to split $u$ into two nodes of roughly the same weight. We create two buckets from the current bucket and the chunk $c'$ resulting from the split of $u$ becomes the root of the new bucket with the largest elements. Finally, we add $c'$ to the root area as actual or virtual chunk.

Deleting an element is analogous, except that we merge instead of splitting $u$ (although, after a merge, we may split once); as for $v$, we need merging and borrowing with an individual element.

We now detail the ideas sketched above in the rest of the section. In Section 13.3.2, we describe how to allocate the elements in memory and, in Section 13.3.3, we focus on the internal organization of the trees. In Section 13.3.4, we show how to implement the search and update operations on the buckets using the techniques introduced in Sections 13.3.2–13.3.3. Finally, in Section 13.3.5, we provide the analysis of our algorithms.
13.3.2 Memory management of the trees

We postpone the layout of the roots to Sections 13.4 and 13.5. Here we discuss the layout of the intermediate nodes, the leaves, the maniples, and the spare elements in the bottom layer.

**Overview of the layouts.** We store the spare elements in no particular order in the spare area. Using *compactor zones* (or, simply, *zones*) we accommodate the intermediate nodes and the leaves in the node area and the maniples in the maniple area. The proposed primitives are invoked when a node or a maniple changes size and so its elements must be suitably relocated in memory. What we describe next for the nodes holds also for the maniples.

We pack together the nodes of *identical size* $s$, embedding them in a suitable zone devoted to nodes of size $s$ and called *zone* $s$. When a node changes size, it also changes zone. Each node in zone $s$ occupies a contiguous segment of $s$ memory locations, except possibly the node at the beginning of each zone. In this case, we maintain the property that the node is stored in two segments of $s_1$ and $s_2$ locations, respectively, where $s_1 + s_2 = s$. The last $s_2$ elements of the nodes are at the beginning of zone $s$ and the first $s_1$ elements of the node are at the end of zone $s$. We call *broken* this node and any other node in the zone *unbroken*. Hence, all nodes are unbroken except possibly one node per zone (see Figure 13.3). The (encoded) pointer to a broken node contains extra $O(\log k)$ bits to encode that value of $s_1$.

The zones share common techniques, such as the following primitive for rotating a zone $s$. Suppose we have $m$ elements to the right of zone $s$, and let $X$ be the memory segment hosting the whole zone $s$ along with the $m$ elements to its right. A *rotation* is an in-place primitive that incrementally moves the $m$ elements to the beginning of $X$ and the first $m$ elements in zone $s$ to the end of $X$ (or vice versa) *without* scanning the whole $X$. For $i = 1, 2, \ldots, m$, it exchanges the $i$th element of zone $s$ with the $i$th among the $m$ elements. At the end, the broken node (if any) in zone $s$ may become unbroken and at most one unbroken node may become broken.

We search one element for each of the latter nodes re-encoding the $O(1)$ pointers to it that are identified through the search. We also re-encode the starting position of zone $s$. The cost of a rotation is $O(k + m)$ time and $O(\lceil (k + m)/B \rceil)$ block transfers,
plus the cost of $O(1)$ searches. We now give more details on the zones.

### 13.3.2.1 Node area.

The node area contains $15 \sqrt{k} + 1$ compactor zones in increasing order of index $s$. Each zone $s$ is adjacent to zone $s - k$ (to its left) and to zone $s + k$ (to its right), since the size of the nodes is a multiple of $k$. The starting positions of all the zones in this area are encoded in the first $\sqrt{k}$ chunks of the area itself. We support the following basic primitives in this area:

- **ExtractNode($w$)** extracts node $w$ placing it between the node area and the manipule area. Let $s$ be the size of $w$.

  1. We exchange $w$ with the rightmost unbroken node in zone $s$ (note that $w$ may be the broken node).

  2. Now, we have $w$ followed by the initial portion of the broken node (if any) in zone $s$ and by zone $s + k$ (if any). We exchange these $O(k \sqrt{k})$ elements by the block exchange technique (see Chapter 4) in $O(k \sqrt{k})$ time and $O([k \sqrt{k} / B])$ block transfers, with the initial portion of the broken node in zone $s$ followed by $w$ and by zone $s + k$.

  3. We also need to perform $O(1)$ searches to re-encode $O(1)$ pointers in their parents. As a result, we shorten zone $s$ by $s$ positions to the right and have $w$ between zones $s$ and $s + k$.

  4. For $s' = s + k, s + 2k, \ldots, 16k \sqrt{k}$ (i.e., incrementing $s'$ by $k$), we move $w$ from the left to the right of zone $s'$ by a rotation and update the new starting positions of zone $s'$.

At the end, we have $w$ to the right of zone $16k \sqrt{k}$, that is, between the node area and the manipule area.

- **InsertNode($w$)** inserts node $w$ into its suitable compactor zone and is similar to **ExtractNode** (with the same cost), where $w$ is between the node area and the manipule area.

**Lemma 13.1** The cost of **ExtractNode** and **InsertNode** in the node area is $O(k^2)$ time and $O(\sqrt{k} [k \sqrt{k} / B])$ block transfers, plus the cost of $O(\sqrt{k})$ searches.

- **TransferChunk($c$)** transfers chunk $c$ from an area to another. Either $c$ is between the node area and the manipule area and we want it laying between the root area and the node area, or vice versa. We proceed like in **ExtractNode** and **InsertNode**. However, we do not insert or delete $c$ inside any zone, but we simply rotate each zone $s'$ to move $c$ from one side to another of zone $s'$.

The cost of the primitive takes into account the fact that $c$ has size $k$.

**Lemma 13.2** The cost of **TransferChunk** in the node area is $O(k \sqrt{k})$ time and $O(\sqrt{k} [k / B])$ block transfers, plus the cost of $O(\sqrt{k})$ searches.
13.3.2.2 Maniple and spare areas

**Maniple area.** It contains $4\sqrt{k} + 1$ compactor zones in increasing order of index $s$. Each zone $s$ contains the maniple of identical size $s$. Its neighbors are zone $s - \sqrt{k}$ (to its left) and zone $s + \sqrt{k}$ (to its right), since the size of the maniples is a multiple of $\sqrt{k}$. The starting points of all the zones are encoded in the first $\sqrt{k}$ chunks of the node area. We support the following primitives that we will employ to implement the update operations in Section 13.3.4:

- **ExtractManiple**($z$) extracts maniple $z$ and place it either between the node area and the maniple area, or between the maniple area and the spare area. Its implementation is analogous to **ExtractNode**, taking into account the size $O(k)$ of the maniples.

- **InsertManiple**($z$) inserts $z$ into its zone analogously to **InsertNode**, where $z$ is either between the node area and the maniple area, or between the maniple area and the spare area.

- **TransferElements**($m$) transfers $m$ contiguous elements from the left of the maniple area to its right, or vice versa, where $m \leq k$. It is like **TransferChunk**, except that each zone rotates by $m$ positions.

**Lemma 13.3** The cost of **ExtractManiple**, **InsertManiple** and **TransferElements** in the maniple area is $O(k\sqrt{k})$ time and $O(\sqrt{k} \lceil k/B \rceil)$ block transfers, plus the cost of $O(\sqrt{k})$ searches.

**Spare area.** Here there are no compactor zones, but the spare elements are stored contiguously without any specific order. So, the basic primitives are simple to describe.

- One primitive inserts a new spare element to the right of the spare area and extends the right border of the area to include the new spare element.

- Another primitive extracts a spare element leaving a hole inside the spare area that is filled with the rightmost spare element in the area, thus shortening the right border of the spare area by one position.

In all cases, we search the element to update its pointer encoded in a suitable chunk of a leaf.

**Lemma 13.4** Extracting or inserting a spare element in the spare area has a cost of $O(k)$ time and $O([k/B])$ block transfers plus the cost of $O(1)$ searches.

We may also want to collect $m$ spare elements between the maniple and the spare area. In that case, we can see this operation as a sequence of $m$ extractions according to Lemma 13.4, giving a total cost of $O(km)$ time and $O(m \lceil k/B \rceil)$ block transfers plus the cost of $O(m)$ searches.
13.3.3 Internal structure of the trees

The internal structure of the bucket trees allows for quick searching and updating even though their size is $\Theta(k^2)$. We assume that the algorithms described in Section 13.3.2 provide the necessary memory support for what we discuss below. Hence, the analysis of the costs in this section does not account for the costs of the memory management given in Section 13.3.2.

13.3.3.1 Intermediate nodes

The simplest organization is that of the intermediate nodes. Given an intermediate node $u$, containing $t = \Theta(\sqrt{k})$ chunks $c_1, \ldots, c_t$, we keep a directory in the first $2t$ positions of $u$ containing the smallest and the greatest element of $c_1, \ldots, c_t$, respectively, in this order. If the elements in a chunk $c_j$ change, it is a simple task to update $c_j$ and the directory. Routing a search for element $x$ in $u$ examines some of the $O(t)$ elements in the directories to identify a pair of elements $x_1$ and $x_2$ such that $x_1 \leq x \leq x_2$. These two elements belong to at most two consecutive chunks $c_j$ and $c_{j+1}$ that are accessed by simple offsetting. We also support the following primitives for an intermediate node $u$:

- $\textsc{BuildDirectory}(u)$ when the elements of $u$ are in sorted order. For $j = 1, 2, \ldots, t$, we shift the leftmost and rightmost element in each chunk $c_j$.

- $\textsc{InsertChunk}(c, u)$ assumes that $c$ is contiguous to $u$ in memory. We perform a block exchange of $c$ without its leftmost and rightmost element (fixing $m = k - 2$), so that it reaches its position in $u$, say after chunk $c_j$. Then, we shift the two aforementioned elements of $c$ to their positions in the directory of $u$, just after those of $c_j$.

- $\textsc{ExtractChunk}(c, u)$ is analogous to $\textsc{InsertChunk}$.

Lemma 13.5 Given an intermediate node $u$, routing an element in $u$ using its directory requires $O(k)$ time and $O([k/B])$ block transfers. $\textsc{BuildDirectory}$ takes $O(k^2)$ time and $O(\sqrt{k} [k \sqrt{k} / B])$ block transfers. $\textsc{InsertChunk}$ and $\textsc{ExtractChunk}$ require $O(k \sqrt{k})$ time and $O([k \sqrt{k} / B])$ block transfers.

13.3.3.2 Leaves

The leaves undergo a more involved organization because they delegate the insertion and deletion of individual elements to their associated maniples (see Figure 13.4). In particular, when inserting an element into a leaf $v$, we start a sequence of internal rotations that quickly make room for the new element and kick out the greatest element in $v$.

Analogously, when removing an element from $v$, we want the opposite situation, namely, remove the deleted element and add to $v$ an element that is greater than
any other element in $v$. Reducing the problem of inserting or deleting an element in $v$ to that of handling the operation suitably in its maniple $z$ is the rationale for the rotational machinery described next.

We give a two-step description, in which we first describe how to organize the elements in the leaf $v$ for working in main memory, and then how to make this organization optimal in the Cache-Oblivious model that is with a cache complexity of $O(\log_B n')$ block transfers.

**First step: work optimal organization.** For the sake of discussion, let's assume that we have to add one more element $x$ to a chunk $c$ in $v$. What we have to do is inserting $x$ into $c$ by shifting its elements while extracting the maximum element in $c$, which we insert into the chunk just to the right of $c$. Hence, for any chunk $c'$ to the right of $c$, we insert into $c'$ the minimum element (extracted as the greatest from the chunk to the left of $c'$) while extracting the maximum element (being inserted as the smallest into the chunk to the right of $c'$). When we reach the right end of $v$, we have extracted the maximum element in $v$. While we can shift the elements in $c$, we cannot afford the $O(k)$ cost of shifting all the elements in the remaining chunks $c'$, as they can be $\Theta(\sqrt{k})$ in number.

Consequently, we organize each chunk $c'$ in $v$ as follows, denoting by $x_1$ the minimum element to be inserted into $c'$ and by $x_2$ the maximum element to be extracted from $c'$ in the generic iteration step (the symmetric operation of deleting $x_1$ and inserting $x_2$ is analogous):

1. The elements $a_1, a_2, \ldots, a_k$ are kept rotated by an offset $r$, occurring as $a_{r+1} \cdots a_k \cdot a_1 \cdots a_r$ in $c'$.

2. Inserting $x_1$ while extracting $x_2 = a_k$ gives $a_{r+1} \cdots a_{k-1} x_1 a_1 \cdots a_r = a_{r+2}' \cdots a_k' \cdot a_1' \cdots a_r' a_{r+1}'$.

3. Either $1 \leq r \leq \sqrt{k}$ or $k - \sqrt{k} + 1 \leq r \leq k$ (i.e., elements $a_{\sqrt{k}+1} \cdots a_{k-\sqrt{k}}$ are not rotated).
4. \(a_{\sqrt{k}+1}, \ldots, a_{k-\sqrt{k}}\) encode the \(O(1)\) pointers and integers for the structural information in \(c'\).

Preserving the above organization crucially relies on keeping condition 3 valid. When the latter condition is violated in a chunk, we reset its rotation with a block exchange.

**Second step: achieve cache optimality.** Right now, the internal organization of the leaves is not yet suitable for cache-obliviousness because the sequence of rotations previously mentioned access at least one element per traversed chunk, which is a problem for small values of the (unknown) block size \(B\). We observe that the accessed elements in this task are always kept in the first or the last mini-chunk of each chunk \(c'\), namely, \(a_1, \ldots, a_{\sqrt{k}}\) and \(a_{k-\sqrt{k}+1}, \ldots, a_k\) by condition 3.

To obtain cache optimality, we form a directory at the beginning of leaf \(v\) with these two mini-chunks for each chunk. We obtain a larger directory than that of intermediate nodes, still the number of elements in the directory being \(O(k)\). As a result, the rotation of each chunk \(c'\) will always involve its elements occurring inside that directory. Routing a search element \(x\) inside \(v\) uses the directory in a way similar to that described for the intermediate nodes.

We also support the following primitives for a leaf \(v\):

- **BuildDirectory**\((v)\) when the elements of \(u\) are in sorted order. For each chunk \(c\) in \(v\), we apply block exchanges to the leftmost and the rightmost mini-chunk in \(c\).

- **InsertChunk**\((c,v)\) assumes that \(c\) is contiguous to \(v\) in memory.

  1. We perform a block exchange of \(c\) without its leftmost and rightmost mini-chunk (fixing \(m = k - 2\sqrt{k}\)), so that it reaches its position in \(v\), say after chunk \(c'\).

  2. We exchange the two aforementioned mini-chunks of \(c\) to their positions in the directory of \(v\), just after those of \(c'\). Note that the rotation for \(c\) has offset 0.

- **ExtractChunk**\((c,v)\) is analogous to **InsertChunk**.

**Lemma 13.6** Given a leaf \(v\), routing an element in \(v\) using its directory requires \(O(k)\) time and \(O(|k/B|)\) block transfers. **BuildDirectory** costs \(O(k^2)\) time and \(O(\sqrt{k} \lfloor k \sqrt{k}/B \rfloor)\) block transfers. **InsertChunk** and **ExtractChunk** require \(O(k \sqrt{k})\) time and \(O(\lfloor k \sqrt{k}/B \rfloor)\) block transfers.

We postpone the cost of how to maintain the invariant for conditions 1–4 to Section 13.3.3.5.
13.3. Bottom Layer: Buckets as Implicit Dynamic Forest

13.3.3.3 Maniples and spare elements

Given a leaf \( v \), its associated maniple \( z \) is divided into mini-chunks. The elements in \( z \) are all greater than those in \( v \), and they are kept in sorted order among the mini-chunks. The last \( \sqrt{k} \) mini-chunks in \( z \) have the spare elements associated, namely, from 1 to 5 spare elements per mini-chunk. These elements are contained in the interval represented by the corresponding mini-chunk.

The property above is crucial for searching in the maniple and decoding only \( O(1) \) pointers to the spare elements. These pointers are encoded in the first \( 5\sqrt{k} \) chunks of \( v \), specifically, in the elements of \( v \) mentioned in condition 4 on the invariant for the leaves. The association is static and, independently of how many spare elements a mini-chunk has, we encode 5 pointers in \( v \) for its spare elements (some of them can be nil, encoded as 0). When searching in \( z \), we decode at most 5 of these pointers, namely, those corresponding to the mini-chunk in which our search ends.

We also support the following primitives:

- **InsertElement**\((x, z)\): assumes that there are less than \( 5\sqrt{k} \) spare elements in total in \( z \) and that they are contained in the interval represented by their mini-chunk.

1. Since \( z \) is in sorted order, we identify the mini-chunk \( c \) of \( z \) that encloses \( x \).
2. If \( c \) has less than 5 spare elements, we insert \( x \) into \( c \) incrementing its spare elements by one, encoding its pointer in the proper chunk of \( v \).
3. Otherwise, we extract the greatest element \( y \) of \( c \) and identify the closest mini-chunk \( \acute{c} \) to the right of \( c \) (or to its left) that has less than 5 spare elements.
   
   We shift by one position to the right (or to the left) all the elements between \( c \) and \( \acute{c} \) to make room for \( y \) in sorted order. Now, \( \acute{c} \) has one extra element to be inserted into it, incrementing its spare elements by one as described above.

- **ExtractElement**\((x, z)\): Analogous to InsertElement.

As we shall see, we periodically scan all mini-chunks and their spare elements choosing as new spare elements the medians of each mini-chunk. This preserves the crucial property that the spare elements are inside the mini-chunks (seen as intervals) for any sequence of InsertElement and ExtractElement operations.

**Lemma 13.7** Given a maniple \( z \) and its leaf \( v \), routing an element in \( z \) requires \( O(k) \) time and \( O([k/B]) \) block transfers. InsertElement and ExtractElement take \( O(k) \) time and \( O([k/B]) \) block transfers.
13.3.4 Supporting the operations on the buckets

We now discuss how to implement the search, insert, and delete operations on the buckets.

13.3.4.1 Searching an element in a bucket

Searching element $x$ after visiting the root of the bucket in the top layer consists of routing $x$ inside the intermediate node $u$, the only child of the root, as stated in Lemma 13.5.

1. If we find the element as a member of a chunk in $u$, we are done.

2. Otherwise, we identify a chunk $c$ in $u$, reaching the leaf $v$ whose pointer is encoded in $c$.

3. We route $x$ also inside $v$ according to Lemma 13.6. If $x$ is a member of $v$, we are done.

4. Otherwise, either $x$ is not stored in the dictionary, or it is stored in its associated minipe $z$. We search in $z$ according to Lemma 13.7 and identify a mini-chunk $c'$ in $z$. If $x$ is a member of $c'$ or equals one of its $O(1)$ spare elements, we are done.

5. Otherwise, we can infer that the search of $x$ is unsuccessful.

**Lemma 13.8** Given the root of a bucket, searching $x$ in the bucket takes $O(k)$ time and $O([k/B])$ block transfers.

13.3.4.2 Inserting an element into a bucket

We now discuss the insertion of $x$ into a bucket. We describe our insertion in two phases, a descending phase (Steps 1–3) and an ascending phase (Steps 4–5).

1. We begin by searching $x$ as stated in Lemma 13.8. If $x$ belongs to the chunk that is the root, we add $x$ and shift at most $k$ elements to extract the maximum element in that chunk, obtaining the new element $x$ to be inserted into the intermediate node $u$, the only child of the bucket root.

   In general, inserting $x$ into a chunk of $u$ goes along the same lines. The maximum of the chunk is extracted to make room for the new element. We set $x$ to this maximum element, which should be inserted into the suitable child leaf $v$ of $u$.  

2. We insert \( x \) into a leaf \( v \) with maniple \( z \). We run the algorithm described in Section 13.3.3.2 for inserting \( x \) into \( v \) while extracting its maximum element by means a sequence of internal rotations. Let \( x \) be the extracted element, which should be inserted into \( z \).

3. We insert \( x \) into maniple \( z \) using the following substeps:

   (a) If \( \sqrt{k} \) insertions and deletions have been performed on \( z \), we reset the counting for \( z \) and choose its new spare elements as the medians of each mini-chunk (see Section 13.3.3).

   (b) If the number of spare elements in \( z \) is less than \( 5\sqrt{k} \), we perform \textsc{InsertElement}(\( x, z \)), increasing by one the number of spare elements associated with \( z \) and \textsc{Exit}.

   (c) If \( z \) has the maximum number of spare elements, \( 5\sqrt{k} \), we move the smallest \( \sqrt{k} \) spare elements, say \( s_1, s_2, \ldots, s_{\sqrt{k}} \), in \( z \) at the beginning of the spare area, shortening that area.

   Then, we proceed by induction for \( i = 1, 2, \ldots, \sqrt{k} \), where the smallest \( i - 1 \) elements in \( z \) have been exchanged with \( s_1, \ldots, s_{i-1} \).

   i. We find the rank \( r_i \) of \( s_i \) among the current elements of \( z \).

   ii. We then shift the first \( r_i \) elements in \( z \) by one position to the left to make room for \( s_i \), with the smallest element "ejected" from \( z \) and stored in the location left free by \( s_i \).

   With an analogous method, we can redistribute the remaining spare elements in \( z \), so that they are the medians of their corresponding mini-chunks.

   (d) i. We run \textsc{ExtractManiple}(\( z \)) to bring \( z \) between the node area and the maniple area.

   ii. We apply \textsc{TransferElements}(\( \sqrt{k} \)) to move near to \( z \) the \( \sqrt{k} \) extracted elements, which form a mini-chunk.

   iii. We move this mini-chunk at the beginning of \( z \) (as it is the smallest mini-chunk) by block exchange (see Chapter 4).

   iv. We extend \( z \) with this mini-chunk.

   v. If \( z \) has still no more than \( 5k \) elements, we run \textsc{InsertManiple}(\( z \)) to put \( z \) back to its proper compactor zone and jump to Step 3a.

4. At this point, \( z \) has \( 5k + \sqrt{k} \) elements and stays between the node area and the maniple area.

   (a) We run \textsc{ExtractNode}(\( v \)) for the leaf \( v \) having \( z \) associated, putting \( v \) between the node area and the maniple area, just to the left of \( z \).
(b) If \( v \) has less than \( 4k\sqrt{k} \) elements, we separate the first chunk \( c \) of \( z \) from the rest of it, thus shortening \( z \) by a chunk and creating a chunk that must be inserted into the leaf \( v \) with \text{INSERTCHUNK}(c,v). We then execute \text{INSERTNODE}(v) and \text{INSERTMANILE}(z) to their compactor zones, and jump to Step 3a.

(c) If \( v \) has \( 4k\sqrt{k} \) elements, we split \( v \) into two leaves \( v_1 \) and \( v_2 \), their maniples \( z_1 \) and \( z_2 \) and their spare elements from those in \( z \) and from the ones associated with \( v \).

i. We move the spare elements of \( z \) at the beginning of the spare area and apply \text{TRANSFERELEMENTS} to bring them to the right of \( z \) between the node area and the maniple area.

ii. We perform an in-place merge of the elements in \( z \) and the sorted sequence of the spare elements by repeated insertions. As a result, we have a sorted sequence of elements by scanning \( v \) and the sequence formed with the fusion of \( z \) with the spare elements.

iii. We divide the sequence in six parts:

\[
v_1, z_1, s_1, c, v_2, z_2, s_2,
\]

where \( c \) is the median chunk. Leaf \( v_1 \) contains \( 2k\sqrt{k} \) elements and has associated the maniple \( z_1 \) of \( 2k \) elements and \( 2\sqrt{k} \) spare elements in \( s_1 \). Leaf \( v_2 \) contains \( 2k\sqrt{k} \) elements and has associated the maniple \( z_2 \) of \( 2k + \sqrt{k} \) elements and \( 2\sqrt{k} \) spare elements in \( s_2 \). We perform some block exchanges in these \( O(k\sqrt{k}) \) elements obtaining the following sequence:

\[
v_1, v_2, c, z_1, z_2, s_1, s_2.
\]

iv. We apply \text{BUILDDIRECTORY} to \( v_1 \) and \( v_2 \), and then reinsert \( v_1 \) and \( v_2 \) into the node area with \text{INSERTNODE}, and \( z_2 \) and \( z_1 \) into the maniple area with \text{INSERTMANILE}.

We are left with the elements in \( c, s_1, s_2 \) between the node area and the maniple area. We encode in \( c \) a pointer to the position of \( v_2 \).

v. We apply \text{TRANSFERELEMENTS} moving the elements in \( s_1, s_2 \) to the positions between the maniple area and the spare area. We extend the spare area by \(|s_1| + |s_2|\) positions to the left.

Since the elements in \( s_1 \) and \( s_2 \) are not spare elements, we redistribute these spare elements among the elements of their maniples, in a way similar to that presented in Step 3c.

5. At this point, chunk \( c \) stays between the node area and the maniple area.

(a) We run \text{EXTRACTNODE}(u) for the parent \( u \) of \( v \), so that \( u \) is near to \( c \) between the node area and the maniple area. We add \( c \) to \( u \) with \text{INSERTCHUNK}(c,u).
(b) If $u$ has weight $w(u) \leq 16k\sqrt{k}$ elements, we run $\text{InsertNode}(u)$ back to its compactor zone and jump to Step 3a.

(c) Otherwise, $u$ has weight $w(u) > 16k\sqrt{k}$.

   i. We merge its directory with the rest of its elements.

   ii. Now the elements in $u$ are sorted, and we split them into $u_1$, $c'$ and $u_2$ in this order so that $w(u_1)$ and $w(u_2)$ are approximately $8k\sqrt{k}$, and $c'$ is the median chunk.

   iii. We apply $\text{BuildDirectory}$ to $u_1$ and $u_2$, and insert $u_1$ and $u_2$ into the node area using $\text{InsertNode}$.

   iv. We encode in $c'$ a pointer to the position of $u_2$ and move $c'$ using $\text{TransferChunk}$ so that it reaches the position between the root area and the node area (it will become part of the root area).

We have thus created two buckets, and $c'$ is the root of the new bucket containing $u_2$. We jump to Step 3a.

13.3.4.3 Deleting an element from a bucket

The delete operation for an element $x$ admits an implementation that is symmetrical to that of the insertion. Consequently, we do not give here the details. We only mention here that, once that $x$ is identified in the bucket tree, we delete it from its chunk adding a new maximum element from a child.

The deletion in a manipulate $z$ starts out an ascending phase that uses merges in place of splits. When a merge is not doable for a leaf $v$, we borrow a single element from a neighbor (leaf with its manipulate and associated spare elements). In all the other cases, we merge $v$ and, if necessary, its parent $u$. The latter may require a split as second operation to preserve the invariant on its weight $w(u)$. Moreover, the insertions and the deletions interact in that they change the rotation of the chunks in the leaves and the position of the spare elements inside the mini-chunks of the manipiles.

13.3.5 Analysis of the costs

We provide a detailed analysis for the amortized cost of the insertions. In the proof, we closely follow the algorithmic scheme presented in Section 13.3.4 to show that the cost of each step is $O(k + \log n')$ time and $O(k/B + \log_B n')$ block transfers.

**Theorem 13.2** In the bottom layer, each bucket contains $\Theta(k^2)$ elements at any time.

- Searching a bucket takes $O(k)$ time and $O([k/B])$ block transfers.

- The amortized cost of updating a bucket is $O(k)$ time and $O([k/B])$ block transfers per insert/delete operation plus the cost of $O(1)$ searches.
• Any newly created bucket will merge or split after further $\Omega(k^2)$ update operations inside that bucket.

• At any time, only $O(1)$ auxiliary locations are required to operate dynamically.

Proof: The complexity of Step 1 is within our target bounds by Lemma 13.8 and by the fact the $O(1)$ chunks are accessed.

Let us evaluate the amortized complexity of Step 2. The rotations are all performed in the first $O(k)$ elements. What is more expensive is the reset of the rotation in the chunks of the leaf that otherwise would violate conditions 1–4. Resetting each chunk costs $O(k)$ time and $O([k/B])$ block transfers and there can be $O(\sqrt{k})$ of them. Since for any two consecutive resettings there are $\Omega(\sqrt{k})$ insertions and deletions involving leaf $v$, the amortized cost of a resetting is $O(k)$ time and $O([k/B])$ block transfers.

Let us evaluate the amortized complexity of Step 3. Step 3a requires $O(k\sqrt{k})$ time and $O(\sqrt{k}[k/B])$ block transfers because of decoding the pointers to the $\Theta(\sqrt{k})$ spare elements. Step 3b is worst-case by Lemma 13.7. Step 3c has the same cost as that of Step 3a, plus the cost of $O(\sqrt{k})$ searches by Lemma 13.4. Step 3d costs as Step 3c by Lemma 13.3 and the by the cost of a block exchange procedure. In summary, the total cost of Step 3 is $O(k\sqrt{k})$ time and $O(\sqrt{k}[k/B])$ block transfers plus the cost of $O(\sqrt{k})$ searches. Since Steps 3a–3d occur every $\Omega(\sqrt{k})$ insertions and deletions involving $v$, the amortized cost is $O(k)$ and $O([k/B])$ block transfers plus the cost of $O(1)$ searches.

Step 4 occurs every $\Omega(k)$ insertions and deletions involving $z$, whose leaf is $v$, except for Step 4c, which occurs every $\Omega(k\sqrt{k})$ insertions and deletions involving $v$ and $z$. The cost of Step 4a is given by Lemma 13.1, that of Step 4b by Lemma 13.3 and Lemma 13.5. This gives an amortized cost of $O(k)$ time and $O([k/B])$ block transfers plus the cost of $O(1)$ searches. The cost of Step 4c is detailed as follows: that of Step 4(c)i is given by Lemma 13.3 and Lemma 13.4; Step 4(c)ii is $O(\sqrt{k}[k/B])$ for the $O(\sqrt{k})$ repeated insertions; Step 4(c)iii is $O(k[\sqrt{k}/B])$ for the block exchanges; Step 4(c)iv by Lemma 13.1, Lemma 13.3 and Lemma 13.5; Step 4(c)iv by Lemma 13.3. The total cost of Step 4c is $O(k^2)$ time and $O(\sqrt{k}[k\sqrt{k}/B])$ block transfers plus the cost of $O(\sqrt{k})$ searches, giving an amortized cost of $O(k)$ time and $O([k/B])$ block transfers plus the cost of $O(1)$ searches, as expected.

Step 5 occurs every $\Omega(k\sqrt{k})$ insertions and deletions involving leaf $v$, whose parent is the intermediate node $u$. The cost of Step 5a is given by Lemma 13.1 and Lemma 13.5; that of Step 5b by Lemma 13.1; finally, that of Step 5c by $O(k^2)$ time and $O(\sqrt{k}[k/B])$ block transfers for the merge done with $O(\sqrt{k})$ repeated insertions, and by Lemma 13.1, Lemma 13.4 and Lemma 13.5. This concludes the analysis of the insertions.

A detailed analysis of the deletions can done analogously to that of the insertions and we leave the details to the interested reader. □
13.4 Top Layer: The Super-Root

The top layer contains the super-root of the flat implicit tree and collects all the actual chunks and the virtual chunks. We remark that these chunks are the roots of the buckets in the bottom layer discussed in Section 13.3. The memory layout in the super-root area is simple:

- first, all the actual chunks in sorted order
- then, all the virtual chunks in no particular order.

We describe in this section how to handle efficiently the super-root in main memory. We discuss how to make it cache-oblivious in Section 13.5.

13.4.1 Actual chunks and virtual chunks

The $a$ actual chunks are stored in sorted order in the first $ak$ positions of the super-root area, where

- $a = O \left( \frac{n'}{k^3} \right)$
- $a$ is always a power of two.

Each actual chunk has associated at most $\alpha = O(1)$ virtual chunks, which are the nearest in the order of the (actual and virtual) chunks. They are kept in a linked list starting from the actual chunk. The rest of the area contains the virtual chunks in no particular order as the linked lists allow their retrieval.

The super-root area resizes by $k$ positions to the right at a time to make room for one more or less chunk after bucket splitting or merging (see Section 13.3.3 and Theorem 13.2). The number $a$ of actual chunks changes only when rebuilding (see Section 13.6) or when performing a full redistribution of actual and virtual chunks.

Since the actual chunks are kept sorted, we can route an element to its (actual or virtual) chunk in the top layer in $O(\log n')$ time.

**Lemma 13.9** Routing an element in the super-root takes $O(\log n')$ time to identify the correct bucket to access.

13.4.2 Density functionalities

We partition the actual chunks into $a/\hat{k}$ segments of $\hat{k}$ adjacent chunks each, where $\hat{k}$ is the power of two such that $\hat{k} \leq k < 2\hat{k}$, obtaining a sorted array $R$ of $O(a/\hat{k}) = O(n'/k^3)$ segments. The segments in $R$ are the leaves of an implicit complete binary tree of height

$$h = O \left( \log \frac{n'}{k^3} \right),$$
and an internal node $u$ represents a portion of consecutive segments in $R$. If $u$ is at depth $s$, then its portion in $R$ contains $2^{h-s}k$ actual chunks. There is a static mapping between the internal nodes and the segments of $R$ (except the last). In other words, each segment is associated with a leaf and an internal node; the last segment is associated with the rightmost leaf. In the following, when we refer to the nodes and the leaves of $R$ we mean those in the implicit tree defined above.

The number of actual chunks and that of virtual chunks are related each other by applying the notion of density [Bender, Demaine, and Farach-Colton, 2000, Brodal, Fagerberg, and Jacob, 2002, Itai, Konheim, and Rodeh, 1981] to $R$. In our case, we have the further requirements of avoiding to produce empty slots and of distributing virtual chunks among the actual chunks without violating their order (i.e., we cannot simply move one virtual chunk from a position to another of $R$). That allows us to maintain virtual chunks almost uniformly distributed in $R$, with at most $\alpha$ virtual per actual chunk.

Let $\text{act}(u)$ denote the actual chunks descending from $u$. As previously mentioned, if $u$ has depth $s \leq h$, then

$$|\text{act}(u)| = 2^{h-s}k.$$ 

Let $\text{vir}(u)$ denote the virtual chunks associated with the actual chunks in $\text{act}(u)$, where

$$|\text{vir}(u)| \leq \alpha |\text{act}(u)|.$$ 

We encode the value of $|\text{vir}(u)|$ in the segment of $R$ associated with $u$. We define the density of $u$ as

$$d(u) = \frac{|\text{vir}(u)|}{\alpha |\text{act}(u)|}.$$ 

The density of all nodes is constrained according to their depth and to some positive integer constants $\alpha$ and positive real constants $\rho_0$ and $\tau_0$. In order to amortize the cost of updating, we require that the constants satisfy the relations

$$2\alpha \rho_0 + 1 < \alpha \tau_0$$

$$\alpha \geq 2$$

$$0 < \rho_0 < \tau_0 < 1$$

(e.g., $\alpha = 2$, $\rho_0 = 0.1$, and $\tau_0 = 0.9$). We also need two further constants $\rho_\infty$ and $\tau_\infty$, such that

$$0 < \rho_\infty < \rho_0 < \tau_0 < \tau_\infty < 1.$$ 

Let’s define the density thresholds:
\[
\rho_s = \rho_0 - \frac{s}{h}(\rho_0 - \rho_\infty) \quad \text{and} \quad \tau_s = \tau_0 + \frac{s}{h}(\tau_\infty - \tau_0)
\]

(note that \(\rho_h = \rho_\infty\) and \(\tau_h = \tau_\infty\)). The density of a node \(u\) at depth \(s\) must satisfy

\[\rho_s \leq d(u) \leq \tau_s.\]

We say that a leaf \(u\) overflows if \(d(u) > \tau_h\) and underflows if \(d(u) < \rho_h\). An internal node \(u\) at depth \(s\) overflows (resp., underflows) if \(d(u) > \tau_s\) (resp., \(d(u) < \rho_s\)) and recursively at least one of its children either overflows or underflows [Bender, Demaine, and Farach-Colton, 2000, Brodal, Fagerberg, and Jacob, 2002, Itai, Konheim, and Rodeh, 1981].

### 13.4.3 Rebalancing/redistribution of chunks

The association of virtual chunks with actual chunks is dynamically maintained when bucket splitting creates new chunks and bucket merging removes chunks (see Section 13.3.3 and Theorem 13.2). As long as we can keep at most \(\alpha\) virtual chunks associated with each actual chunk, we do not need to redistribute chunks. However, when removing an actual chunk that has no virtual chunk associated with it, or when creating a new chunk from a chunk in a maximal list of size \(\alpha\), we have to redistribute the chunks by reclassifying them as actual and virtual, while preserving their order.

As it should be clear, the classification has an impact on the allocation in main memory as detailed at the beginning of Section 13.4.

The redistribution of actual and virtual chunks should preserve the order between chunks. In a node \(u\) involved in the redistribution, we change the sets of chunks \(\text{act}(u)\) and \(\text{vir}(u)\) without changing the set of chunks contained in the segment of \(R\) corresponding to \(u\), and without changing the size \(|\text{act}(u)|\) and \(|\text{vir}(u)|\) (hence, the density of \(u\)).

#### 13.4.3.1 Redistribution within a segment

The redistribution of chunks inside the segments (leaves) of \(R\) can operate sequentially on the whole segment in \(O(k^2)\) time. For example, let's take two consecutive actual chunks \(a_i\) and \(a_{i+1}\) in a segment of \(R\), with associated lists \(L_i\) and \(L_{i+1}\) of virtual chunks, each list containing at most \(\alpha\) entries.

Suppose that, after a bucket splitting, \(L_i\) becomes of size \(\alpha + 1\). The rightmost virtual chunk in \(L_i\), say \(c\), replaces \(a_{i+1}\). Specifically, \(c\) becomes actual occupying the positions of \(a_{i+1}\) in \(R\), and \(a_{i+1}\) becomes virtual occupying the positions of \(c\). Moreover, \(a_{i+1}\) is prepended to list \(L_{i+1}\), which becomes associated with \(c\). If also \(L_{i+1}\) becomes of size \(\alpha + 1\), we go on iteratively with the next actual chunk \(a_{i+2}\) until we reach the end of the segment.
Alternatively, we can consider $a_{i-1}$ and $a_i$ and proceed analogously towards the beginning of $R$. Handling the removal of an actual chunk after bucket merging can be treated in the same fashion.

13.4.3.2 Redistribution of chunks for an internal node

When the density of a segment reaches either its maximum or its minimum value, we run the redistribution of chunks in an internal node $u$ of $R$ as described next. We work under the hypothesis that $u$ is the deepest ancestor of the segment overflowing or underflowing, such that at least one of the children of $u$ overflows or underflows whereas $u$ does not, that is, its density is

$$\rho_s \leq d(u) \leq \tau_s,$$

where $s$ is the depth of $u$.

We now show how to redistribute the actual and virtual chunks in the portion of $R$ corresponding to $u$, denoting by $u[i]$ the $i$th actual chunk in that portion, for $1 \leq i \leq |act(u)| = 2^{h-s} \mathring{k}$.

Although we change the status of some actual and virtual chunks, thus changing the chunks in sets $act(u)$ and $vir(u)$, we preserve their size $|act(u)|$ and $|vir(u)|$ (hence, the density of $u$). The redistribution for $u$ is in-place to preserve implicitness at any time and runs in two phases.

**First phase.** In the first phase, we scan the actual chunks of $u$ in decreasing order. We employ a linked list $L$, which is initially empty, and it incrementally contains virtual chunks. For $i = 2^{h-s} \mathring{k}, \ldots, 2, 1$, we execute the following steps:

1. If the actual chunk in $u[i]$ has virtual chunks associated with it, we append them to the end of $L$ in decreasing order (no chunk relocation is needed, just re-encoding of pointers).

2. The chunk at the beginning of $L$ replaces the actual chunk in $u[i]$, which is appended to the end of $L$ (here we switch an actual chunk with a virtual chunk).

At the end of the first phase, list $L$ contains the $|vir(u)|$ smallest chunks in $u$, while $u$ stores the $|act(u)|$ largest chunks in its portion of $R$.

**Second phase.** In the second phase, we scan all the chunks in $u$ and distribute $|vir(u)|/|act(u)|$ virtual chunks per actual chunk. We employ the list $L$ computed in the first phase, executing the following steps for $i = 1, 2, \ldots, 2^{h-s} \mathring{k}$:

1. The chunk at the beginning of $L$ replaces the chunk in $u[i]$, which is appended to the end of $L$. 
2. The next $|\text{vir}(u)|/|\text{act}(u)|$ chunks are removed from $L$ to create the linked list of virtual chunks associated with $u[i]$.

At the end of the second phase the virtual chunks are uniformly distributed among the actual chunks by preserving their order. The density $d(u)$ of $u$ does not change, whereas the density of any descendant $v$ of $u$ is $d(v) = d(u)$. Hence, it satisfies

$$\rho_t \leq d(v) \leq \tau_t,$$

where $t > s$ is the depth of $v$, since $\rho_t < \rho_s < \tau_s < \tau_t$.

**Redistribution in the root.** A special case of redistribution is in the root $u$ of $R$. The two-phase algorithm is quite similar to what described so far, except that the number $a$ of actual chunks in $R$ doubles (root overflow) or halves (root underflow) in the second phase. Note that the size of the super-root area in the memory layout does not change. Consequently we run a mere reorganization inside the super-root area.

- When the root $u$ overflows, we have $d(u) > \tau_0$ and so

$$|\text{vir}(u)| > \alpha \tau_0 |\text{act}(u)|.$$

In the second phase, we transform $a = |\text{act}(u)|$ virtual chunks in $L$ into actual chunks, so that $R$ contains $2a$ actual chunks. After the distribution, the new density for $u$ is

$$d(u) > \frac{\alpha \tau_0 - 1}{2\alpha} > \rho_0,$$

since we fix our constants satisfying $2\alpha \rho_0 + 1 < \alpha \tau_0$. The new density also verifies $d(u) < \tau_0$, since we double the number of actual chunks while decreasing the number of virtual chunks.

- When $u$ underflows, we have $d(u) < \rho_0$ and so

$$|\text{vir}(u)| < \alpha \rho_0 |\text{act}(u)|.$$

In the second phase, we transform $(1/2)a = (1/2)|\text{act}(u)|$ actual chunk in $R$ into virtual chunks in $L$, so that $R$ remains with $(1/2)a$ actual chunks. After the distribution, the new density for $u$ is

$$d(u) < \frac{2\alpha \rho_0 + 1}{\alpha} < \tau_0$$

as $2\alpha \rho_0 + 1 < \alpha \tau_0$. The new density also verifies $d(u) > \rho_0$, since we halve the number of actual chunks while increasing the number of virtual chunks.
In both cases, we re-encode the new values of $|\text{vir}(u)|$ in the suitable chunks of $R$. Since the height $h$ of the implicit tree changes, we encode this information in the preamble $\mathcal{P}$ of the memory layout. This helps us to define the new values of $\rho_s$ and $\tau_s$ for the several levels $s$ on the fly.

Since the density of the resulting root $u$ of $R$ satisfies

$$\rho_0 < d(u) < \tau_0$$

after the redistribution, all the descendants $v$ of $u$ have density $d(v)$ satisfying

$$\rho_s < d(v) < \tau_s,$$

where $s$ is the depth of $v$, since $\rho_0 < d(v) < \tau_0$ and $\rho_s < \rho_0 < \tau_0 < \tau_s$.

13.4.4 Analysis

The analysis of the amortized cost for an internal node $v$ on depth $s+1$ is a variation of that given in [Bender, Demaine, and Farach-Colton, 2000]. The cost for rebalancing $v$ is $O(\|(\text{act}(u)| + |\text{vir}(u)|) \cdot k)$, where $u$ is the parent of $v$ (and has depth $s$). In order for $v$ to overflow again, we need at least additional $\alpha|\text{act}(v)|((\tau_{s+1} - \tau_s)\Theta(k^2)$ insertions in the buckets associated with the actual and virtual chunks in the portion of $R$ corresponding to $v$ (see Theorem 13.2), and thus the amortized cost per operation on $v$ is

$$\frac{(\|\text{act}(u)| + |\text{vir}(u)|)\Omega(k)}{\alpha|\text{act}(v)|((\tau_{s+1} - \tau_s)\Theta(k^2)} = \frac{(\|\text{act}(u)| + |\text{vir}(u)|)h}{\alpha|\text{act}(v)|((\tau_\infty - \tau_0)\Theta(k)} \leq \frac{(\alpha + 1)|\text{act}(u)|h}{\alpha|\text{act}(v)|((\tau_\infty - \tau_0)\Theta(k)} = O(h/k) = O(1),$$

noting that $|\text{act}(u)| = 2|\text{act}(v)|$. A similar analysis holds for deletions. In order for $v$ to underflow again, we need at least additional $\alpha|\text{act}(v)|((\rho_{s+1} - \rho_s)\Theta(k^2)$ deletions in the buckets associated with the actual and virtual chunks in the portion of $R$ corresponding to $v$ (see Theorem 13.2), and thus the amortized cost per operation on $v$ is

$$\frac{(\|\text{act}(u)| + |\text{vir}(u)|)\Omega(k)}{\alpha|\text{act}(v)|((\rho_{s+1} - \rho_s)\Theta(k^2)} = \frac{(\|\text{act}(u)| + |\text{vir}(u)|)h}{\alpha|\text{act}(v)|((\rho_0 - \rho_\infty)\Theta(k)} \leq \frac{(\alpha + 1)|\text{act}(u)|h}{\alpha|\text{act}(v)|((\rho_0 - \rho_\infty)\Theta(k)} = O(h/k) = O(1),$$

Since there are $h$ levels, each update operation has an amortized cost of $O(h^2/k) = O(\log n^h)$ for the whole maintenance of $R$ and the super-root. We assume that the root $u$ of $R$ satisfies $\rho_0 < d(u) < \tau_0$, which is guaranteed initially and, later, by each redistribution of chunks in the root as previously discussed.
Theorem 13.3 In the top layer, routing an element to its bucket takes $O(\log n')$ time. Inserting or deleting a chunk in any position of the super-root has an amortized cost of $O(h^2/k) = O(\log n')$ time, where $h = O(\log n')$. At any time, only $O(1)$ auxiliary locations are required to operate dynamically.

13.5 Top Layer: Cache-Obliviousness

We show how to modify the organization described in Section 13.4 so that it has optimal bounds for the cache complexity as well. We recall that the top layer collects all the actual chunks and the virtual chunks, which are the roots of the buckets in the bottom layer discussed in Section 13.3.

The memory layout in the root area is simple; first, all the $a$ actual chunks in sorted order, where $a = 2^h = \Theta(n' / \log^2 n')$ is always a power of two; then, all the virtual chunks in no particular order. Each actual chunk has at most $\alpha = O(1)$ virtual associated with it, which are the nearest in the order of the (actual and virtual) chunks. They are kept in a linked list starting from the actual chunk. The root area resizes by $k$ positions to the right at a time to handle one more or one less chunk after bucket splitting or merging (see Section 13.3.3 and Theorem 13.2). The number $a$ of actual chunks changes only when rebuilding or when performing a full redistribution of actual and virtual chunks.

Overview of the new layout. We have only to discuss how to make cache-oblivious the access to the actual chunks since the virtual chunks are not much of a problem. In the following, we assume that the rightmost actual chunk is treated separately and we are left with $a - 1 = 2^h - 1$ actual chunks. The main idea is to build an internal directory for the root similarly to what is done for internal nodes $u$ in Section 13.3.3. We refer to actual chunks both when they contain $k$ elements or when they have only $k - 2$ elements since their smallest and greatest element are in the directory. It is worth noting that the directory is permuted while the actual chunks are kept in increasing order.

Moreover, the elements in the directory are located between the actual chunks and the virtual chunks. Conceptually, we treat each pair of elements in the directory as a single interval. When searching an element $x$, we compare it to each interval by exploiting the fact that the intervals are disjoint: either $x$ is inside the interval, or it is to the left or the right of the interval. If we have cache-oblivious access to the directory, we can access an actual chunk and its associated virtual chunks in $O(k)$ comparisons and $O(\lfloor k/B \rfloor)$ block transfers.

We focus therefore on how to permute the directory assuming to have just $2^h - 1$ elements (rather than $2^h - 1$ pairs of elements encoding disjoint intervals) for the sake of discussion.
13.5.1 Building the VEB-permutation

We define the van Emde Boas permutation (shortly, VEB-permutation) of $2^h - 1$ elements following Prokop's recursive scheme for van Emde Boas trees [Prokop, 1999]. Suppose to have a complete binary tree with $h \geq 1$ levels and $2^h - 1$ nodes, where $h = 1$ indicates that the tree has just one node. The tree stores the sorted sequence of $2^h - 1$ elements in symmetric order. Since we do not keep this tree anywhere, we permute the elements recursively according to the tree structure. In what follows, let $A$ denote the memory segment hosting these $2^h - 1$ elements with the scheme of the VEB-permutation and let VEB-tree indicate the complete binary tree mentioned above.

- If $h = 1$, we simply store the element associated with the unique node in the VEB-tree.
- Otherwise, let $h_T = \lceil h/2 \rceil$ and $h_B = h - h_T$.
  - We recursively store the $2^{h_T} - 1$ entries in the top tree of height $h_T$ in the first $2^{h_T} - 1$ locations of $A$.
  - Then, for $i = 0, 1, \ldots, 2^{h_T} - 1$, we recursively store the $2^{h_B} - 1$ entries of the bottom tree number $i$ (from left to right) in the $i$th portion of $A$ (i.e., starting from $A[2^{h_T} + i \cdot (2^{h_B} - 1)]$).

We now describe how to build a VEB-permutation in-place in

$$O(2^h h) = O(a \log a) = O(n'/k)$$

comparisons and block transfers. (We can achieve a better bound but this does not improve the final bounds.)

1. We first run Heapsort on the sequence of $2^h - 1$ entries.

2. We then apply the recursive scheme mentioned above. The base case for $h = 1$ is easy to handle, so let’s suppose $h > 1$ and compute $h_T$ and $h_B$.

   (a) For $j = 1, \ldots, 2^{h_T} - 1$, we swap the element, say $t_j$, in position $j$ with the element in position $j \cdot 2^{h_B}$. Now, the elements associated with the top tree are in order in the first $2^{h_T} - 1$ positions of $A$.

   (b) We now sort the elements in the rest of $A$ (i.e., $A[2^{h_T} \ldots 2^{h} - 1]$). To sort the elements in $A[2^{h_T} \ldots 2^{h} - 1]$, we exploit the fact they are almost sorted, apart from the elements $t_1, \ldots, t_{2^{h_T}}$ (the latter are in relative order). We perform some block exchanges. We start from $t_{2^{h_T}}$, which is exchanged so that it meets $t_{2^{h_T} - 1}$. In general, for $j < 2^{h_T}$, we have exchanged and collected $t_{j+1}, \ldots, t_{2^{h_T}}$ to the right of $t_j$, and we want to exchange the former elements near the latter. We execute a block exchange to move $t_{j+1}, \ldots, t_{2^{h_T}}$ as close as possible to $t_j$; then, we move $t_j$ towards them, thus obtaining $t_j, t_{j+1}, \ldots, t_{2^{h_T}}$ collected together, as expected.
(c) We recursively apply our construction to the elements in \(A[1, \ldots, 2^h - 1]\) (the top tree) and, for \(i = 0, 1, \ldots, 2^h - 1\), to the elements in \(A[2^h + i \cdot (2^h - 1) \ldots 2^h + (i + 1) \cdot (2^h - 1) - 1]\) (the bottom subtree number \(i\)).

**Lemma 13.10** A VEB-permutation for \(2^h - 1\) elements can be built in \(O(2^h h) = O(a \log a) = O(n'/k)\) comparisons and block transfers, where \(a = 2^h = O(n'/k^2)\).

**Proof:** The cost of this construction is asymptotically bounded from above by the solution to the recurrence

\[
C(2^h - 1) = C(2^h - 1) + 2^h C(2^h - 1) + O(2^h).
\]

Note that putting \(t_1, \ldots, t_{2^h}\) in their suitable order (w.r.t. the other elements) gives the sorting and costs only \(O(2^h)\) as we scan \(A\) from the right to the left. For a suitable constant \(d\), we have that

\[
C(2^h - 1) \leq d2^h h
\]

by substitution on the right hand side of the recurrence.

We remark that the construction of the VEB-permutation is not fully in-place as it uses the recursion. We therefore handle directly the non-tail recursion by using a stack storing only the pairs of values \(h_T, h_B\) thus found at each recursive level. Since \(h_T\) and \(h_B\) can be encoded in \(O(\log h)\) bits and we keep \(O(\log h)\) of them during the recursion, the total number of bits for the full stack is \(O(\log^2 h) = o(\log n')\). We can handle this stack in \(O(1)\) locations in constant time per operation, using simple arithmetic operations for push and pop. We do not violate the assumptions of the RAM uniform cost. Using this “implicit” stack in a location and additional \(O(1)\) location, we are able to build in-place the VEB-permutation in \(O(n'/k)\) comparisons and block transfers. \(\square\)

### 13.5.2 Searching the VEB-permutation

We now get to the main point, namely, how to search an element \(x\) in the VEB-permutation for \(2^h - 1\) elements. In [Prokop, 1999] it is shown that traversing the path from the root to a node takes

\[
O(\log_B(2^h - 1)) = O(\log_B n')
\]

block transfers. We show how to do it without the extra information required in [Brodal, Fagerberg, and Jacob, 2002], which is not permitted in the model for implicit data structures.

We use procedure \(\text{FIND}(x, h)\) in Figure 13.5 to achieve our goal. Before invoking it, we know that the segment \(M = A[i \ldots i + 2^h - 2]\) of \(2^h - 1\) elements corresponds to a subtree \(S\) of the VEB-tree. We also know that \(A[i]\) is the element in the root
\textbf{\textsc{Find}}(x, h):
1: \textbf{if} \( h = 1 \) \textbf{then}
2: \hspace{0.5cm} \textbf{if} \( x \leq A[i] \) \textbf{then}
3: \hspace{1.5cm} \text{bfs} \leftarrow \text{bfs} \times 2
4: \hspace{0.5cm} \textbf{else}
5: \hspace{1.5cm} \text{bfs} \leftarrow \text{bfs} \times 2 + 1
6: \hspace{0.5cm} \textbf{else}
7: \hspace{2cm} h_T \leftarrow \lceil h/2 \rceil, h_B \leftarrow h - h_T
8: \hspace{1.5cm} \textsc{Find}(x, h_T)
9: \hspace{2cm} \text{rank} \leftarrow \text{bfs} \mod 2^{h_T}
10: \hspace{1.5cm} i \leftarrow i + (2^h - 1) \times \text{rank} + 1
11: \hspace{1.5cm} \textsc{Find}(x, h_B)
12: \hspace{2cm} \text{bfsroot} \leftarrow \text{bfs}/2^{h_B+1}
13: \hspace{2cm} \text{rankroot} \leftarrow \text{bfsroot} \mod 2^{h_T}
14: \hspace{1.5cm} i \leftarrow i + (2^h - 1) \times (2^{h_T} - 1 - \text{rankroot})

Figure 13.5: Procedure \textsc{Find} to search \( x \) in a VEB-permutation of \( 2^h - 1 \) elements stored.

of \( S \), and that the breadth-first number of \( A[i] \) in the VEB-tree is \text{bfs}. (Initially, \( S \)
\( S \), and that the breadth-first number of \( A[i] \) in the VEB-tree is \text{bfs}. (Initially, \( S \)
\( \text{is the VEB-tree, } M = A \), and so \( i = 1 \) and \( \text{bfs} = 1 \).

When \textsc{Find}(x, h) completes, it has traversed the path from the root of \( S \) to one
leaf \( v \) of \( S \) and \( i \) has reached the position of the last element in \( M \). The routing
of element \( x \) must go on either to the left or to the right of \( v \) in the rest of the
VEB-tree, and we crucially know the \text{bfs} of the next node to visit. In any case,
\text{bfs} \mod 2^{h} gives the rank of \( x \) among the elements in \( M \). Identifying the position \( j \)
such \( A[j] = x \) (if any) is a minor modification.

We now show how to keep the invariant on \textsc{Find}(x, h) by induction on \( h \) (see
Figure 13.5). If \( h = 1 \), this is immediate. Let's take the case of \( h > 1 \). We
compute the number of levels \( h_T \) and \( h_B \) for the top and bottom subtrees of \( S \),
respectively called \( S_T \) and \( S_B \). We want to identify their corresponding segments
\( M_T \) and \( M_B \) in \( A \). First, note that \( M_T = A[i \ldots i + 2^{h_T} - 2] \). So, we can invoke
directly \textsc{Find}(x, h_T) to route \( x \) in \( S_T \). By induction, \( \text{rank} = \text{bfs} \mod 2^{h_T} \) tells us the
number of \( S_B \) (starting from 0 and going from left to right in \( S \)). Since each bottom
subtree has size \( 2^{h_B} - 1 \), we can infer that \( M_B \) starts at position \( i + (2^{h_B} - 1) \cdot \text{rank} + 1 \)
of \( A \). So we update \( i \) to this new value. We can invoke \textsc{Find}(x, h_B) to route \( x \) in \( S_B \).
By induction, we correctly compute \text{bfs} in the VEB-tree for the next node below \( S_B \)
that is also the next node below \( S \). In order to preserve the invariant, we need to
update \( i \) so that it is the last position of \( M \) in \( A \). We have to compute \text{rank} again,
since we cannot keep this value due to the implicitness. We therefore, compute the
breadth-first number of the root of \( S_B \), which is the current value of \text{bfs} divided by
\( 2^{h_B+1} \) (the exponent \( h_B + 1 \) comes from the fact that \text{bfs} refers to one level below
the leaves of $S_B$). We then take the modulo of the resulting breadth-first number in rankroot, as we did for rank (indeed lines 9 and 13 compute the same value but at different times in the recursion; so we cannot keep the values of the variables rank in the recursive levels to obtain an in-place algorithm). We finally increment $i$ noting that we have to jump over the elements of $(2^{br} - 1 - \text{rankroot})$ bottom subtrees of size $2^{bn} - 1$. As a result, we keep the invariant for $S$.

Finally, we observe that we traversed a path from the root of $S$ to a leaf of it. The only accessed elements in $A$ are at line 2 updating bfs, so that the next element to be compared with $x$ is either the left child or the right child of $A[y]$ but not both.

**Lemma 13.11** A VEB-permutation for $2^h - 1$ elements can be searched in $O(h) = O(\log n')$ comparisons and $O(h/\log B) = O(\log_B n')$ block transfers.

**Proof:** The cost of searching is asymptotically upper bounded by the solution to the recurrence

$$C(2^h - 1) = C(2^{br} - 1) + C(2^{bn} - 1) + O(1).$$

The crucial observation is that FIND traverses a downward path from the root to an internal node, whose length is asymptotically bounded by $C(2^h - 1)$. For suitable constant $d_1$ and $d_2$, we have that

$$C(2^h - 1) \leq d_1 h - d_2$$

by substitution on the right hand side of the recurrence. Hence, a VEB-permutation for $2^h - 1$ elements can be searched in $O(h)$ comparisons and $O(h/\log B)$ block transfers. \hfill $\Box$

### 13.5.3 Maintaining the VEB-permutation

We have three main cases in the process of maintaining the VEB-directory.

**Normal redistribution.** When redistributing the actual and virtual chunks like in Section 13.4.3, we are allowed to pay a search in the VEB-directory for each chunk involved in the redistribution with its two smallest and greatest elements, replacing them when the corresponding chunk is exchanged.

**Halving the number of the actual chunks.** When halving the number of actual chunks, we must move the $a/2$ largest chunks to the right of the VEB-directory (thus halving the VEB-directory), adding them to the list of virtual chunks as mentioned in Section 13.4.3.

1. We first run Heapsort on the VEB-directory to get it in decreasing order.
2. We then merge the $a/2$ largest chunks with the left half of the sorted VEB-directory.

3. We then apply a block exchange by $k \cdot a/2$ positions to the left of the VEB-directory and rebuild the VEB-directory as described in Section 13.5.1.

4. Finally, we transform these $a/2$ largest chunks into virtual chunks and complete the redistribution described in Section 13.4.3 (we replace elements in the VEB-directory when redistributing).

**Doubling the number of actual chunks.** When doubling the number of actual chunks, we have the opposite situation, namely, the $a$ smallest virtual chunks to the right of the VEB-directory send their smallest and greatest element in the VEB-directory, and their $k - 2$ remaining elements become actual chunks to the left of the VEB-directory. We process these virtual chunks one by one in order, assuming that each chunk $c$ of them is adjacent to the VEB-directory (a simple exchange of two virtual chunks can achieve this as we have the virtual chunks linked in order).

1. We take the largest element in $c$ and move it adjacent to the smallest element.

2. We then exchange incrementally the other $k - 2$ elements with the first $k - 2$ elements currently in the VEB-directory (note that, because of this, the VEB-directory is scrambled).

At the end, we transform these $a$ virtual chunks into actual chunks and rebuild the VEB-directory from scratch. We then complete the redistribution as described in Section 13.4.3.

**Lemma 13.12** A VEB-permutation for $2^h - 1$ elements can be updated when halving or doubling the number of actual chunks in no more than $O(a \log a + ak)$ time and block transfers.

### 13.5.4 Analysis

The analysis of the amortized cost for an internal node $v$ on depth $s + 1$ is a simple extension of that given in Section 13.4.4, in which the cost in terms of block transfers is $(|\text{act}(u)| + |\text{vir}(u)|) O([k/B])$, giving rise to

$$\frac{(|\text{act}(u)| + |\text{vir}(u)|) h O([k/B])}{\alpha |\text{act}(v)|(\tau_\infty - \tau_0) \Theta(k^3)} = O\left(\left[\frac{k}{B}\right]^h \frac{h}{k^3}\right).$$

When $v$ is the root, however, we must replace the cost of $(|\text{act}(u)| + |\text{vir}(u)|) O([k/B])$ by $O(a[k/B])$ plus the cost of rebuilding and maintaining the VEB-permutation as stated in Lemma 13.10 and 13.12, which is bounded by
\[ O(a \log a + ak) = O(n'/k). \]

As a result, noting that \(|act(v)| = \Theta(n'/k^2)\) when \(v\) is the root, the amortized cost for the root is
\[
\frac{O(n/k)}{a\Theta(n/k^2)(\tau_\infty - \tau_0)\Theta(k^2)} = O(h/k) = O(1).
\]

Since there are \(h\) levels, of which one is for the root, we obtain a total amortized cost of \(O(h^2/k^2 |k/B|) = O(\lfloor k/B \rfloor)\) block transfers.

**Theorem 13.4** In the top layer, routing an element to its bucket takes \(O(\log n')\) comparisons and \(O(\log n')\) block transfers. Inserting or deleting a chunk in any position of the root area has an amortized cost of \(O(k)\) comparisons and \(O(\lfloor k/B \rfloor)\) block transfers plus the cost of \(O(1)\) searches.

### 13.6 Rebuilding and Final Analysis

In our algorithms we assumed that \(n'/4 < n < n'\) at any time, where \(n'\) is a power of two. That value of \(n'\) is important to fix the parameter \(k\) discussed in Section 13.2. Note that the time complexity of our algorithms is parametric in \(k\) and that we fixed \(k = \Theta(\log n')\) to get our claimed bounds.

To preserve the invariant on \(n'/4 < n < n'\) when \(n = n'\), we double \(n'\), update the value of \(k\) and rebuild by a sequence of \(O(n)\) insertions, with the only difference that we fix \(k = \Theta(\log n')\) even if we may have re-inserted less than \(n'/4\) elements during the rebuilding (this is important to avoid triggering a sequence of recursive rebuilding).

Analogously, when \(n = n'/4\) we halve \(n'\), update the value of \(k\), and rebuild. In both cases, the event marks the beginning of a new epoch, with \(n = n'/2\) for the new value of \(n'\) after rebuilding. Hence our invariant is maintained with \(n\) half on the way between \(n'/4\) and \(n'\).

The amortized cost of rebuilding is given by the total cost of \(O(n')\) insertions divided by the number of insertions and deletions performed in an epoch, which is \(\Omega(n')\). As a result, the amortized cost of the rebuilding is the cost of \(O(1)\) insertions (here it should be clear why we do not start a nested sequence of rebuilding operations, since we keep \(k = \Theta(\log n')\) unchanged for all the rebuilding). Apart from the rebuilding, the cost of search, insert and delete is that of Theorems 13.2–13.4. From these costs, our main result, Theorem 13.1, follows.

### 13.7 Conclusions

In this chapter we presented the Flat Implicit Tree, which avoids space wasting by just using the plain array of \(n\) locations for the elements and, simultaneously, provides optimal data access at any level of the memory hierarchy.
The Flat Implicit Tree is the first optimal data structure obtaining $O(\log n)$ time for search and update in an array of $n$ locations since the invention of the heaps in the sixties.

The Flat Implicit Tree is also cache-oblivious achieving $O(\log n)$ comparisons and $O(\log_B n)$ block transfers for any block size $B$, where the value of $B$ and the capacity of the memory levels are unknown to the algorithms operating in the model.
Chapter 14

Implicit Dictionaries with $O(1)$ Modifications per Update and Fast Search

Abstract

The most interesting lower bound for the implicit dictionary problem is the 1988 result of Borodin, Fich, Meyer auf der Heide, Upfal and Wigderson [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988]. The authors prove a tradeoff between the search time and the update time in implicit dictionaries: if the update cost, number of comparisons and exchanges, for an implicit dictionary is $O(1)$ then the search cost must be $\Omega(n^\epsilon)$, for some constant $\epsilon > 0$. In their conclusion, the authors left open the question of whether a tradeoff of that kind would hold if only the modifications (i.e. exchanges) performed during any update were considered. They conjectured that this would be in fact the case and that any implicit dictionary performing only $O(1)$ exchanges per update should very quickly get disorganized, thus ending up requiring $\Omega(n^\epsilon)$ comparisons per search.

In this chapter, we answer to this long-standing open question by disproving the conjecture. First, we prove the existence of an implicit dictionary $S$ supporting searches with $O(\log |S|)$ comparisons in the worst case and updates with $O(1)$ exchanges and $O(\log |S|)$ comparisons in amortized sense. Then, we concentrate on worst case updates and prove the existence of an implicit dictionary that can be updated with a constant number of exchanges in the worst case while still needing only a polylogarithmic number of comparisons per search in the worst case.

The presentation in this chapter is based on the paper [Franceschini and Munro, 2006] (SODA 2006).
14.1 Introduction

14.1.1 The open question and our contribution

A well-known, classical result for the Implicit Model is described in [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988]. In that 1988 paper Borodin, Fich, Meyer auf der Heide, Upfal and Wigderson proved a tradeoff between the search time and the update time in implicit dictionaries. They essentially proved that if the update cost, that is the total number of comparisons and exchanges, for an implicit dictionary is $O(1)$ then the search cost must be $\Omega(n^\epsilon)$, for some constant $\epsilon > 0$ (an analogous tradeoff for generic dictionaries has recently been proved in [Radhakrishnan and Raman, 2001]).

In their conclusions, the authors left open the question of whether a tradeoff of that kind would hold if only the modifications performed during an update were considered. They conjectured that this would be in fact the case and that any implicit dictionary performing $O(1)$ exchanges per update should very quickly get disorganized, thus requiring $\Omega(n^\epsilon)$ comparisons per search.

In this chapter we answer to the long-standing open question left in [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988] and disprove the conjecture advanced by the authors. First, we prove the existence of an implicit dictionary $S$ supporting searches with $O(\log |S|)$ comparisons in the worst case and updates with $O(1)$ exchanges and $O(\log |S|)$ comparisons in amortized sense. Then, we concentrate on worst case updates and prove the existence of an implicit dictionary which can be updated with a constant number of exchanges in the worst case while still needing only a polylogarithmic number of comparisons per search. Any previously known implicit dictionary with polylogarithmic search requires $\Omega(\log |S|)$ modifications for the updates.

14.1.2 Organization of the chapter

The presentation in this chapter is based on the paper [Franceschini and Munro, 2006] (SODA 2006). In Section 14.2, we start take a look to the previous work on the implicit dictionary problem from the point of view of the exchanges in the update operations. In particular, we focus on the known implicit dictionaries with polylogarithmic search and update time. Then, we show how some of the powerful techniques that are at the very core of these advanced implicit structures are also the major obstacles in the path for solving the open problem left in [Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988]. Finally, we give a glimpse of the ideas to overcome these obstacles. Our two new implicit dictionaries with amortized and worst case updates are described in Section 14.3 and in Section 14.4 respectively.
14.2 Techniques and Limitations of Known Implicit Dictionaries

14.2.1 Storing and carrying information

Implicit dictionaries can be partitioned into two sets according to the contribution in [Munro, 1986]. The implicit dictionaries before that paper did not have polylogarithmic complexity for search and update simultaneously. The best result within that “first generation” of contributions is due to Frederickson [Frederickson, 1983]. In that paper the author presented a collection of dictionaries with the best one supporting searches in $O(\log n)$ comparisons and updates in $O(n^{\sqrt{\log n/\log \log n}}) = o(n^\epsilon)$ time, for any fixed value of $\epsilon > 0$.

All the implicit dictionaries before the one in [Munro, 1986] have a common characteristic: they do not carry enough auxiliary information. That statement may be confusing at first because of the very definition of the Implicit Model which allows the algorithms to store a number of auxiliary bits of information that is $O(\log |S|)$. But “carrying information” does not necessarily mean “storing information”.

Let us suppose that the input set $S$ is permuted in increasing order. $S$ can be searched optimally with $O(\log |S|)$ comparisons. If we perturb the sorted order of $S$ allowing pairs of adjacent elements to be in any order, we can still search the set with $O(\log |S|)$ comparisons using a simple modification of the binary searching algorithm. The difference between these two organizations is that the second one can carry $|S|/2$ bits of information. Using the bit-stealing technique (see Chapter 4), the value of the $i$th bit is 0 or 1 if the $i$th pair of adjacent elements is in increasing or decreasing order, respectively. Any bit can be accessed with a comparison and modified with an exchange. The implicit dictionary in [Munro, 1986] uses this technique as well as the other implicit dictionaries that followed it.

14.2.2 Two main problems

A common way to use auxiliary memory in data structures consists in associating with any element in the structure a constant number of values (e.g. the children pointers in a binary search tree) stored in cells of memory contiguous to the location of the element. An element and its auxiliary values are seen as a whole. A similar way to use stolen bits in implicit dictionaries is maintaining groups of adjacent elements (according to $\leq$) and treating them as basic units. These groups are called chunks and their size is $a \log |S|$, for an integer constant $a$. Hence, the chunks are disjoint intervals of elements from $(U, \leq)$. Each one of them encodes a constant number $(a/2)$ of values of $\log |S|$ bits each. Two problems immediately arise.

(i) In the dictionary problem the cardinality of $S$ is not fixed.
(ii) Elements are inserted or removed from $S$ one at a time and we have to create chunks out of unaggregated elements and vice versa.

If these problems were solved, a simple way to organize chunks for searches and updates would be by treating them as nodes of a balanced search tree where each chunk encodes the pointers to its children (and any other needed attribute). That simple organization introduces an $O(\log |S|)$ multiplicative slowdown factor on the search time (in the worst case, we have to decode a pointer of $\log |S|$ bits for any of the levels of the tree). More sophisticated organizations are necessary to match the bounds of generic dictionaries ([Franceschini, Grossi, Munro, and Pagli, 2002, Franceschini and Grossi, 2003a,b,c, Franceschini, Grossi, Munro, and Pagli, 2004, Franceschini and Grossi, 2005b]).

### 14.2.3 Structure replication

The first problem can be solved by *structure replication*. At any time the permutation of $S$ is composed by $r = \lceil \log \log |S| \rceil + 1$ zones $Z_1Z_2 \ldots Z_{r-1}Z_r$. For any $1 \leq i \leq r-1$,

$$|Z_i| = 2^2^i, \text{ and } |Z_r| = |S| - \sum_{i=1}^{r-1} |Z_i|.$$ 

Any zone $Z_i$ has a fixed chunk size

$$k_i = a \log |Z_i|.$$ 

Let us assume that the second problem is solved and we have a “parametric” implicit dictionary $\mathcal{D}_k(S)$ that, given a fixed chunk size $k$,

- can be searched in $O(k \log^d |S|)$ comparisons
- and updated in $O(k \log^d |S|)$ comparisons and exchanges, for a constant $d$.

Given that, we can have an implicit dictionary $\mathcal{D}(S)$ with $O \left( \log^d |S| \right)$ complexity in four steps.

(i) We organize each $Z_i$ as a parametric dictionary $\mathcal{D}_{k_i}(Z_i)$.

(ii) We insert elements only in $Z_r$.

(iii) We treat the deletion of an element $t$ from $Z_i, i < r$ as a deletion of an element $t'$ from $Z_r$ plus the deletion of $t$ from $Z_i$ and the insertion of $t'$ in $Z_i$.

(iv) Searching the whole structure consists in $r$ searches, one for each zone.
From now on, a fixed chunk size $k$ can be assumed and we are left with the following situation. We have a set of chunks organized for polylogarithmic searches, insertions and deletions of chunks. We have to manage the insertions and deletions of elements falling between two consecutive chunks (chunks are disjoint interval of elements and the meaning of expressions such as “consecutive chunks”, “predecessor chunk” and “successor chunk” is well defined). Insertions and deletions of elements falling within a chunk can be reduced to the former case with $O(1)$ exchanges. For instance, the insertion of $t$ into a chunk $c$ can be reduced to

(i) the exchange of $t$ with the largest (smallest) $t' \in c$ and

(ii) the insertion of $t'$ among the elements falling between the successor (predecessor) of $c$.

14.2.4 The free sets

Let us call free sets the disjoint intervals of elements falling between two consecutive chunks. For any pair of consecutive chunks $c, c'$ there is one free set $F$ between them. Either $c$ or $c'$ encodes a pointer to a position where some elements of $F$ are laid. As we said, the free sets, unlike the chunks, can vary one element at a time. Their role is that of “factories” for the creation (and destruction) of chunks. The role of the chunks is to route the searches for most of the way, down to a single free set.

Managing the free sets is a central problem in implicit dictionaries. Until now, it has been solved in two ways and both of them follow the same basic principle: free sets of the same size have to be close one another.

**Compactor lists.** The first approach is based on the compactor lists (see [Munro, 1986]). The free sets are partitioned into linked lists whose auxiliary information (e.g. succ and pred pointers) is encoded using the elements in the lists. There is one list for any possible size of a free set and free sets of the same size are kept into the same list. We will not give the details of this technique. The central aspect is that every time a free set $F$ is updated, $\Omega(|F|)$ exchanges are needed (the free set changes list).

**Compactor zones.** The second approach uses compactor zones (see [Franceschini and Grossi, 2003b]). Free sets of equal size are kept into contiguous areas by rotating the extremities of such areas whenever an update occurs. The approach is simpler than the previous one but the complexity of updates is the same. There are $O(k)$ additional exchanges in these two approaches. When a free set is updated and moved, the pointer encoded into one of its neighbor chunks is updated to maintain the searchability.

We will overcome the limitations of known implicit dictionaries starting from a simple idea. Let us suppose that we have some disjoint sets of contiguous elements
and that there is a fixed structure indexing the sets in polylogarithmic time (i.e. the indexing structure can route an element to the set to which it can possibly belong).

For any two sets \( \mathcal{G}, \mathcal{G}' \), if we want to maintain a "connection" between them, we can use pointers.

Another way to do it is by exchanging the positions of two of their elements \( e \in \mathcal{G}, e' \in \mathcal{G}' \). If we want to go from \( \mathcal{G}' \) to \( \mathcal{G} \), it is sufficient to search for \( e \), exploiting the fixed indexing structure and the disjointness of the sets. Even though some elements are not laid down contiguously with the other elements of their sets, they can still be reached. To establish if \( e \) belongs to \( \mathcal{G}' \),

- we first verify if \( \hat{e} \in \mathcal{G}' - \{ e' \} \)
- and then we look for \( \hat{e} \) in \( \mathcal{G} \) (which can be reached by searching for \( e \)).

The two approaches have different costs in the Implicit Model. Using the connection between \( \mathcal{G} \) and \( \mathcal{G}' \), costs \( O(k) \) comparisons in the first case (decoding a pointer) and a polylogarithmic number of comparisons in the second case (using the indexing structure).

On the other hand, the cost of breaking the connection between \( \mathcal{G} \) and \( \mathcal{G}' \) and having a new one between \( \mathcal{G}' \) and another set \( \mathcal{G}'' \) is \( O(k) \) exchanges in the first case (encoding a pointer) but only two exchanges in the second case (we exchange back \( e \) and \( e' \) and then exchange \( e' \) with some \( e'' \in \mathcal{G}'' \)).

### 14.3 Amortized Updates

In this section we describe an implicit dictionary supporting searches with \( O(\log |\mathcal{S}|) \) comparisons in the worst case and updates with \( O(1) \) exchanges and \( O(\log |\mathcal{S}|) \) comparisons in amortized sense. Using the structure replication technique (Section 14.2) we can focus on the simpler problem where \( |\mathcal{S}| \) is upper bounded by a fixed \( N \). We denote with \( n \) and \( k \) the cardinality of \( \mathcal{S} \) and the chunk size \( a \log N \), respectively.

The permutation is divided into two main contiguous areas: the chunk area \( \mathcal{C} \) and the aggregating area \( \mathcal{A} \). \( \mathcal{C} \) contains all the chunks of the dictionary. They are not the only occupants though. Any free set \( \mathcal{F} \) has its elements partitioned between \( \mathcal{C} \) and \( \mathcal{A} \).

The elements of \( \mathcal{F} \) placed in \( \mathcal{C} \) are the dummy elements and are organized into \( O(1) \) dummy chunks. Dummy chunks have size \( k \) but they are not meant to route the searches. They are a by-product of the growth of the free sets useful to simplify the organization of the chunks in an efficient index. In \( \mathcal{C} \), the dummy chunks outnumber the chunks by a constant multiplicative factor.

\( \mathcal{C} \) can be shrunk or enlarged by \( k \) positions at its right end. The central part of the structure is \( \mathcal{A} \). The remaining elements of the free sets are in \( \mathcal{A} \). As the name suggests, \( \mathcal{A} \) is given or taken single elements (every update operation goes through \( \mathcal{A} \) and gives or takes aggregates of \( k \) elements each (chunks and dummy chunks).
to or from $C$. $A$ can enlarge or shrink by $k$ positions at its left end and by one position at its right end.

As we noticed in Section 14.2, insertions and deletions of elements within a chunk can be reduced to updates within a free set with $O(1)$ additional exchanges. Therefore, we consider only the latter case.

### 14.3.1 The aggregating area

We assume that $C$ can route a search toward a free set in $O(\log N)$ time. As we will see in Section 14.3.2, once we have the aggregating capabilities of $A$, the organization of $C$ is fairly simple. $A$ is divided into four zones. From left to right, they are

- the padding zone $P$,
- the hub zone $H$,
- the loose zone $L$,
- the filler zone $F$.

#### 14.3.1.1 The invariants for $A$

The elements of a free set $F$ are partitioned into $C$ and $A$. The elements in $C$ are aggregated into a number of dummy chunks. The subset of dummy elements of $F$ is denoted by $\text{Dummy}(F)$ and we have that

$$d'k \leq |\text{Dummy}(F)| \leq d''k,$$

for two constant values $d', d''$. The dummy chunks of $F$ are linked in a bidirectional list whose pointers are encoded into the dummy chunks themselves.

The remaining elements of $F$ are the aggregating elements, $\text{Aggr}(F)$. They are partitioned between $H$ and $L$. The commuting elements of $F$, $\text{Comm}(F)$, are in $H$ and their number is fixed:

$$|\text{Comm}(F)| = 40k.$$

The loose elements of $F$, $\text{Loose}(F)$, are scattered in $L$. We have that

$$0 < |\text{Loose}(F)| \leq 2k.$$
The loose zone. The loose zone \( \mathcal{L} \) is divided into \( \lfloor |\mathcal{L}| / 2k \rfloor \) spots of 2k contiguous positions each except for the last one, the incomplete spot, \( \text{spot}_t \). The position of \( \text{spot}_t \) is one of the \( O(1) \) values of \( \log N \) bits allowed by the Implicit Model.

Any spot is associated with one free set. We denote by \( \text{spot} (\mathcal{F}') \) the spot of a free set \( \mathcal{F}' \). Not every free set is associated with a spot. The free sets without a spot are linked in a bidirectional list, the no-spot list. In the general case, the elements in a spot are uncorrelated with the free set associated with the spot and are uncorrelated with one another. \( \text{spot}_t \) may not be associated with a free set. Let \( s = |\text{spot}_t| \).

(i) If \( s \geq k \) (\( s \leq \frac{1}{2}k \)) then \( \text{spot}_t \) is (is not) associated with a free set.

(ii) If \( \frac{1}{2}k < s < k \) and the last time this relation did not hold we had \( s = k \) (\( s = \frac{1}{2}k \)) then \( \text{spot}_t \) is (is not) associated with a free set.

The hub zone. \( \mathcal{H} \) is divided into hubs of 40k contiguous positions each. Each free set \( \mathcal{F}' \) is associated with a hub, denoted by \( \text{hub} (\mathcal{F}') \). The commuting elements of a free set \( \mathcal{F} \) are not completely stored in \( \text{hub} (\mathcal{F}') \). Some of the elements in \( \text{Comm} (\mathcal{F}') \) may be contained into the hubs of other free sets (and vice versa). The distribution of \( \text{Comm} (\mathcal{F}') \) of a free set \( \mathcal{F}' \) in \( \mathcal{H} \) depends on two factors:

(i) The locations in \( \mathcal{L} \) of the elements in \( \text{Loose} (\mathcal{F}') \).

(ii) The free sets to which the loose elements contained in \( \text{spot} (\mathcal{F}') \) belong. (if \( \mathcal{F}' \) is associated with a spot).

Let us focus on a “specimen” free set \( \mathcal{F} \). To describe the invariants on the distribution of \( \text{Aggr} (\mathcal{F}) \), we introduce a finer partitioning of it. This second partitioning complements the first one discriminating between commuting elements \( \text{Comm} (\mathcal{F}) \) and loose elements \( \text{Loose} (\mathcal{F}) \). \( \text{Aggr} (\mathcal{F}) \) is divided into four disjoint sets:

- the set of triplets, denoted by \( \text{Enc} (\mathcal{F}) \) \text{Triplet} (\mathcal{F}) ;
- the set of twins, denoted by \( \text{Twin} (\mathcal{F}) \);
- the encoding elements, denoted by \( \text{Enc} (\mathcal{F}) \);
- the bogus elements, denoted by \( \text{Bogus} (\mathcal{F}) \).

The first three of these subsets have fixed cardinalities:

\[
|\text{Triplet} (\mathcal{F})| = 9k \quad (14.1)
|\text{Twin} (\mathcal{F})| = 6k \quad (14.2)
|\text{Enc} (\mathcal{F})| = 25k \quad (14.3)
\]
On the other hand, \( \textit{Bogus}(\mathcal{F}) \) has a variable cardinality that is linked with the one of \( \textit{Loose}(\mathcal{F}) \) (they do not share any element, though):

\[
|\textit{Bogus}(\mathcal{F})| = |\textit{Loose}(\mathcal{F})|.
\]

(14.4)

\( \textit{Triplet}(\mathcal{F}) \) is partitioned into \( 3k \) disjoint sets of three elements each, the \textit{triplet sets}.

- The minimum of a triplet set \( \mathcal{T} \) is the \textit{guide triplet}, denoted by \( \alpha(\mathcal{T}) \).
- The median is the \textit{connection triplet}, denoted by \( \beta(\mathcal{T}) \).
- The maximum of \( \mathcal{T} \) is the \textit{loose triplet}, denoted by \( \gamma(\mathcal{T}) \).

In its turn, \( \textit{Twin}(\mathcal{F}) \) is partitioned into \( 3k \) disjoint sets of two elements each, the \textit{twin sets}. The elements of a twin set \( \mathcal{W} \) have similar naming: we have a \textit{guide twin}, \( \alpha(\mathcal{W}) \), and a \textit{connection twin}, denoted by \( \beta(\mathcal{W}) \).

We are now ready to give the final invariants concerning \( \textit{Aggr}(\mathcal{F}) \) and \( \textit{hub}(\mathcal{F}) \). Let us assume that \( \mathcal{F} \) has a spot in \( \mathcal{L} \).

The first \( 9k \) positions of \( \textit{hub}(\mathcal{F}) \) are devoted to the triplet sets. They form the \textit{triplet block}, denoted by \( \textit{hubtriplet}(\mathcal{F}) \), and are divided into \( 3k \) homes of three contiguous positions each. Each triplet set has a home in \( \textit{hubtriplet}(\mathcal{F}) \) and, at any time, at least one of its three elements is stored in its home.

The \( 6k \) positions of \( \textit{hub}(\mathcal{F}) \) after the triplet block, are for the twin sets. They form the \textit{twin block}, denoted by \( \textit{hubtwin}(\mathcal{F}) \) and, as for the triplet block, they are divided into \( 3k \) homes of two contiguous positions each. Each twin set has a home in \( \textit{hubtwin}(\mathcal{F}) \) and at least one of its two elements is stored in its home.

Triplet and twin sets can be in three states: \textit{available}, \textit{employed} and \textit{invalid}. As we will see, changes of state happen during the updates. The positions of the elements of triplet and twin sets are affected by their states. Let \( \mathcal{T} \) be a triplet set and let \( i \) be the position of the home of \( \mathcal{T} \) in \( \textit{hubtriplet}(\mathcal{F}) \).

- \( \mathcal{T} \) is \textit{available}. Its elements are all stored in its home: \( \textit{hubtriplet}(\mathcal{F})[i] = \alpha(\mathcal{T}) \), \( \textit{hubtriplet}(\mathcal{F})[i+1] = \beta(\mathcal{T}) \) and \( \textit{hubtriplet}(\mathcal{F})[i+2] = \gamma(\mathcal{T}) \).

- \( \mathcal{T} \) is \textit{employed}. We have that \( \gamma(\mathcal{T}) \in \textit{Loose}(F) \). Let \( \mathcal{F}' \) be the free set such that \( \gamma(\mathcal{T}) \in \textit{spot}(\mathcal{F}') \).
  - If \( \mathcal{F}' \) does not exist, that is if \( \gamma(\mathcal{T}) \) is located in \( \textit{spot}_t \) then \( \textit{hubtriplet}(\mathcal{F})[i+1] = \beta(\mathcal{T}) \).
  - Otherwise, there exists a twin set \( \mathcal{W}' \in \textit{Twin}(\mathcal{F}') \) such that \( \textit{hubtriplet}(\mathcal{F})[i+1] = \beta(\mathcal{W}') \) and \( \textit{hubtwin}(\mathcal{F}')[i'+1] = \beta(\mathcal{T}) \), where \( i' \) is the position of the home of \( \mathcal{W}' \).

In both cases, there exists an element \( b \in \textit{Bogus}(F) \) such that \( \textit{hubtriplet}(\mathcal{F})[i+2] = b \).
• \( T \) is invalid. Its elements are all stored in its home.

At any time, \( \mathcal{F} \) has at least one available triple set. For any \( l \in \mathcal{F} \), we have that \( l \in \text{Loose}(\mathcal{F}) \) if and only if there exists an employed triplet set \( T_i \) of \( \mathcal{F} \) such that \( \gamma(T_i) = l \).

Let us consider the twin sets. Let us focus on a twin set \( \mathcal{W} \) whose home in \( \text{hubtwin}(\mathcal{F}) \) starts from position \( i \).

• \( \mathcal{W} \) is available. We have that \( \text{hubtwin}(\mathcal{W})[i] = \alpha(\mathcal{W}) \) and \( \text{hubtwin}(\mathcal{W})[i + 1] = \beta(\mathcal{W}) \).

• \( \mathcal{W} \) is employed. There exists a free set \( \mathcal{F}' \) with an employed triplet set \( T'' \) such that \( \gamma(T'') \in \text{spot}(\mathcal{F}) \), \( \text{hubtwin}(\mathcal{W})[i + 1] = \beta(T'') \) and \( \text{hubtriplet}(\mathcal{F}'')[i'' + 1] = \beta(\mathcal{W}) \), where \( i'' \) is the position of the home of \( T'' \).

• \( \mathcal{W} \) is invalid. Its elements are all stored in its home.

The invariant on the number of available twin sets is a little different from the case of triplet sets:

(i) if \( \mathcal{F} \) has a spot in \( \mathcal{L} \) then it must have at least one available twin set;

(ii) if \( \mathcal{F} \) does not have a spot, it must have at least \( k \) available twin sets.

The difference between available and invalid triplet (twin) sets is in the following property:

\[
\text{Let } \mathcal{V} \text{ be the set of triplets (twins) belonging to available or employed triplet (twin) sets. Let } Q \text{ be the sequence of the elements in } \mathcal{V} \text{ in sorted order. For any triplet (twin) set } \mathcal{U} \text{ in } \mathcal{V}, \text{ the elements of } \mathcal{U} \text{ are consecutive in } Q.\tag{14.5}
\]

The elements in \( \text{Enc}(\mathcal{F}) \) are always stored in \( \text{hub}(\mathcal{F}) \) where they occupy the last \( 25k \) positions, the encoding block \( \text{hubenc}(\mathcal{F}) \). The last \( k \) encoding elements maintain \( O(1) \) encoded pointers of \( \log N \) bits each: a pointer to the list of dummy chunks of \( \mathcal{F} \) and a pointer to \( \text{spot}(\mathcal{F}) \) or a couple of pointers to link into the no-spot list, if \( \mathcal{F} \) were not associated with any spot in \( \mathcal{L} \). The states of triplet and twin sets of \( \mathcal{F} \) are in two simple bitmasks encoded in the first \( 24k \) elements of \( \text{hubenc}(\mathcal{F}) \) (four elements for any triplet and twin set).

The padding and filler zones. \( \mathcal{P} \) and \( \mathcal{F} \) are not strictly necessary to the structure but their use results in simpler update algorithms. A global invariant is defined for the whole permutation: for any \( e \in \mathcal{E} \cup \mathcal{H} \cup \mathcal{L} \) and \( f \in \mathcal{P} \cup \mathcal{F} \) we have that \( e < f \). The total size of the two zones is fixed:
\[ |\mathcal{P} \cup \mathcal{F}| = p \cdot (d''k + 40k + 2k), \]  
(14.6)

for a constant \( p \) (\( d''k + 40k + 2k \) is the maximum size of a free set). The size of \( \mathcal{P} \) varies (and because of that the size of \( \mathcal{F} \) varies too), we have that

\[ k \leq |\mathcal{P}| \leq 80k. \]  
(14.7)

### 14.3.1.2 Searching in \( \mathcal{A} \)

Let us assume that the free sets and \( \mathcal{A} \) respect the invariants in Section 14.3.1.1. We also assumed that \( \mathcal{C} \) is capable of routing a search toward a free set in \( O(\log N) \) comparisons. There are four main steps to search an element \( u \) in a free set \( \mathcal{F} \):

1. We search for \( u \) among the elements in \( \text{Dummy}(\mathcal{F}) \) and \( \text{hub}(\mathcal{F}) \) simply by scanning.

2. We scan \( \text{hubtriplet}(\mathcal{F})[3i - 2] \), for \( 1 \leq i \leq 3k \) and find the position \( j \) of the largest element in an employed triplet set and less than \( u \). Using \( \mathcal{C} \), we search for the free set \( \mathcal{F}' \) containing \( \text{hubtriplet}(\mathcal{F})[3j - 1] \). Then, we scan \( \text{hubbw}(\mathcal{F'})[2i] \), for \( 1 \leq i \leq 3k \) looking for \( u \). Finally, we look for \( u \) in \( \text{spot}(\mathcal{F'}) \).

3. We scan \( \text{hubbw}(\mathcal{F})[2i-1] \), for \( 1 \leq i \leq 3k \) and find the position \( j \) of the largest element in an employed twin set and less than \( u \). Using \( \mathcal{C} \), we search for the free set \( \mathcal{F''} \) containing \( \text{hubbw}(\mathcal{F})[2j] \). Then, we scan \( \text{hubtriplet}(\mathcal{F''})[3i - 1] \), for \( 1 \leq i \leq 3k \) looking for \( u \).

4. If the incomplete spot is not associated with any free set, we look for \( u \) into it.

**Lemma 14.1** If \( \mathcal{C} \) can route a search toward a free set in \( O(\log N) \) comparisons, then any free set can be searched in \( O(\log N) \) comparisons.

### 14.3.1.3 Maintaining the invariants: insertions

Let us assume that an element \( u \) has been routed by \( \mathcal{C} \) toward a free set \( \mathcal{F} \) for insertion. At first \( u \) is in the (new) position after \( \mathcal{F} \). We have four phases.
First phase: making the connections.

1. We first exchange \( u \) with \( \mathcal{F}[1] \). Then, using \( \text{hubenc}(\mathcal{F}) \), we find an available triplet set \( \mathcal{T} \) and its home’s position \( i \). We exchange \( l = \text{hubtriplet}(\mathcal{F})[i + 2] \) with \( u \). After that, \( u \in \text{Bogus}(\mathcal{F}) \) and \( l \in \text{Loose}(\mathcal{F}) \).

2. If \( \text{spot}_t \) is associated with a free set \( \mathcal{F}' \), (whose hub can be found searching for one of the elements in the incomplete spot)

   (a) we find an available twin set of \( \mathcal{F}' \mathcal{W} \) and its home’s position \( i' \);

   (b) we exchange \( \text{hubtriplet}(\mathcal{F})[i + 1] \) and \( \text{hubtwin}(\mathcal{F})[i' + 1] \) (the connection triplet and twin of \( \mathcal{T} \) and \( \mathcal{W} \), respectively);

   (c) we mark \( \mathcal{T} \) and \( \mathcal{W} \) as employed. Note that the position of \( \text{spot}_t \) is one of \( O(1) \) values of \( \log N \) bits each allowed by the Implicit Model.

   (d) Before this phase ends, we check if \( \text{spot}_t \) now contains \( 2k \) elements. In that case we just have to update the retained value \( (O(1) \text{ cost}) \) with the position of the new, and empty, \( \text{spot}_t \).

3. If \( \text{spot}_t \) (where \( l \) resides now) is not associated with any free set and its size is \( < k \), we mark \( \mathcal{T} \) as employed and the phase ends.

4. If \( \text{spot}_t \) has not a free set and now its size has become \( k \), we associate it with a free set \( \mathcal{F}'' \) and “connect” \( \mathcal{F}'' \) with the free sets whose elements are in \( \text{spot}_t \). \( \mathcal{F}'' \) is the first free set of the the no-spot list. It is removed from the no-spot list and a pointer to the \( \text{spot}_t \) is encoded in \( \text{hubenc}(\mathcal{F}'') \). After that, for any \( l' \in \text{spot}(\mathcal{F}'') \), we do the following.

   (a) We find its free set, let it be \( \mathcal{F}''' \), its triplet set, let is be \( \mathcal{T}''' \) and its home’s position \( i''' \).

   (b) We find an available twin set \( \mathcal{W}'' \) of \( \mathcal{F}''' \), let \( i'' \) be its home’s position (by the invariant on the available twin sets, there are at least \( k \) of them, since \( \mathcal{F}'' \) did not have a spot).

   (c) We exchange \( \text{hubtriplet}(\mathcal{F}''')[i''' + 1] \) and \( \text{hubtwin}(\mathcal{F}'') [i'' + 1] \) and we mark \( \mathcal{T}''' \) and \( \mathcal{W}'' \) as employed.

Second phase: resetting triplet and twin blocks. In the first phase some triplet and twin sets have been employed. That might cause violations of the invariants on the number of available triplet and twin sets. The second phase corrects this problem executing a reset operation for any violating free set.

Let \( \mathcal{F}' \) a free set left with no available triplet sets after the first phase. Let \( A \) be the portion of \( \mathcal{F} \) going from the first to the \((9k + |\text{Bogus}(\mathcal{F}')|)\)th position. Let \( B \) be the portion going from the \((9k + |\text{Bogus}(\mathcal{F}')| + 1)\)th to the \((18k + 2|\text{Bogus}(\mathcal{F}')| + 1)\)th position. We have three main steps.
1. We exchange the elements in $\text{Triplet}(\mathcal{F}) \cup \text{Bogus}(\mathcal{F})$ with the ones in $A$. That requires $O(|\text{Loose}(\mathcal{F})|) = O(k)$ searches to find the connection and loose triplets of employed triplet sets.

2. We sort $\text{Triplet}(\mathcal{F}) \cup \text{Bogus}(\mathcal{F})$ by executing $9k + |\text{Bogus}(\mathcal{F})|$ scans of $A$. In the $j$th scan,

   (a) we look for the smallest $e \in \text{Triplet}(\mathcal{F}) \cup \text{Bogus}(\mathcal{F})$ still in $A$;

   (b) once we find its location, we exchange it with the $j$th element of $B$.

Any filler element is greater than any element of $\mathcal{F}$ and hence it is easy to discern among triplet, bogus and filler elements during any scan of $A$ (we find the smallest element in $B$, exchange it with $B[9k + |\text{Bogus}(\mathcal{F})| + 1]$ and use it to discern during the scans).

3. We have to exchange the elements of $\text{Triplet}(\mathcal{F}) \cup \text{Bogus}(\mathcal{F})$ back. Let $t$ be an index initially set to 1.

   (a) For each $i = 1 \ldots 3k$, we decode the $i$th entry of the bitmask in $\text{hube}(\mathcal{F})$ containing the states of the triplet sets of $\mathcal{F}$.

   (b) If the encoded value is “available”, we exchange $B[t], B[t+1], B[t+2]$ with $\text{hubtriplet}(\mathcal{F})[i], \text{hubtriplet}(\mathcal{F})[i+1], \text{hubtriplet}(\mathcal{F})[i+2]$, respectively, and set $t = t + 3$.

   (c) If the value of the $i$th entry is “invalid”, we do the same we do for the previous case but in the end we set the $i$th entry to “available”.

   (d) If the value of the $i$th entry is “employed”:

      i. We exchange $B[t]$ with $\text{hubtriplet}(\mathcal{F})[i]$, and $B[t+3]$ with $\text{hubtriplet}(\mathcal{F})[i+2]$ (hence $B[t+3] \in \text{Bogus}(\mathcal{F})$).

      ii. We use $e = \text{hubtriplet}(\mathcal{F})[i+1]$ (which is not a filler element) to find its free set $\mathcal{F}''$.

      iii. We set $t = t + 4$, exchange $B[i+2]$ with one of the filler elements in $\text{spot}(\mathcal{F}'')$ and exchange $B[i+1]$ with the filler element contained in the position $h$ of $\text{hutwin}(\mathcal{F}'')$ such that the twin set corresponding to $h - 1$ is employed and $\text{hutwin}(\mathcal{F}'')[h - 1]$ is the largest element in $\{\text{hutwin}(\mathcal{F}'')[2t - 1]|1 \leq t \leq 3k\}$ less than $e$.

After the reset operation, $\mathcal{F}$ has no invalid triplet set. The reset operation for a free set violating the invariant on the number of available twin sets follows the same lines.
Third phase: creating a new dummy chunk. After the first phase we may have that \(|\text{Loose}(\mathcal{F})| = 2k + 1\). To bring \(\mathcal{F}\) back to normal we want to create a new dummy chunk out of \(k\) of its bogus elements.

1. We decode the bitmask in \(\text{hubenc}(\mathcal{F})\) looking for employed triplet sets. Let \(\mathcal{T}\) be one of them and let \(i\) be its home’s position.

2. We exchange \(\text{hubtriplet}(\mathcal{F})[i + 2] \in \text{Bogus}(\mathcal{F})\) with the last element \(l\) of \(\text{spot}_i\) (or, if there is no \(\text{spot}_i\), the last element of the rightmost complete spot, that becomes the new \(\text{spot}_i\)).

3. Let \(\mathcal{F}'\) and \(\mathcal{F}''\) be the free sets of \(\text{hubtriplet}(\mathcal{F})[i + 1]\) and \(l\), respectively. Let \(i'\) and \(i''\) be the home positions of the twin set of \(\text{hubtriplet}(\mathcal{F})[i + 1]\) and of the triplet set of \(l\), respectively.

   (a) We exchange \(\gamma(\mathcal{T})\) (which is in \(\text{spot}(\mathcal{F}')\)) with \(l\).

   (b) We exchange \(\text{hubtriplet}(\mathcal{F})[i + 1]\) with \(\text{hubtwin}(\mathcal{F}')[i' + 1]\) (now \(\mathcal{T}\) is at home).

   (c) We exchange \(\text{hubtwin}(\mathcal{F}')[i' + 1]\) with \(\text{hubtriplet}(\mathcal{F}'')[i'' + 1]\) and mark \(\mathcal{T}\) as “available”.

4. After the above steps have been applied to \(k\) employed triplet sets, a new dummy chunk \(d\) is placed between \(\mathcal{L}\) and \(\mathcal{F}\). It is worth noting that \(\text{spot}_i\) has lost \(k\) elements in the process. Now its size may be < \(1/2k\) and we may have to apply to \(\text{spot}_i\) a “dissociating” procedure perfectly symmetrical to the “associating” procedure we described at the end of the first phase.

5. To bring \(d\) in \(\mathcal{C}\), we exchange it with the first \(k\) elements in \(\mathcal{P}\). Then, we add \(d\) to the linked list containing \(\text{Dummy}(\mathcal{F})\).

6. If \(|\mathcal{P}| \geq k\), the phase ends. Otherwise, we recover the invariant of \(\mathcal{P}\) in three steps:

   (a) We exchange \(\text{spot}_i\) with the leftmost \(40k\) elements of \(\mathcal{F}\). These are two contiguous blocks and can be exchanged easily: first reverse \(\text{spot}_i\), then reverse \(\mathcal{F}[1 \ldots 40k]\) and finally reverse \((\text{spot}_i)^R(\mathcal{F}[1 \ldots 40k])^R\).

   (b) We exchange \(\mathcal{L}[1 \ldots 40k]\) with the filler elements exchanged in the first step. \(\mathcal{L}[1 \ldots 40k]\) consists of 20 spots and, for each free set \(\mathcal{F}\) associated with one of them, we re-encode the pointer in \(\text{hubenc}(\mathcal{F})\).

   (c) We exchange the first hub \(h\) in \(\mathcal{H}\) with the \(40k\) filler elements exchanged in the previous steps and we update the pointer to \(h\) encoded by one of the neighbor chunks of \(h\)’s free set. (this is one of the rare cases when a hub is moved).
Fourth phase: creating a new chunk. If the free set $\mathcal{F}$ has become too large, its elements have to be partitioned into two free sets $\mathcal{F}', \mathcal{F}''$ and a chunk $c$ so that for any $f' \in \mathcal{F}'$, $f'' \in \mathcal{F}''$ and $e \in c$, we have that $f' < e < f''$. A pointer to $\mathcal{F}''$ (i.e. to the location of its hub) is encoded into $c$ and $c$ is moved into $\mathcal{C}$. The two new free sets are created so that all the “variable parts” with invariants with lower and upper bounds (e.g. dummy chunks, loose elements...) have their size halfway. On the other hand, there are no invalid triplet and twin sets in the two new free sets.

The techniques for this phase are the same described in the previous three phases. For instance, the first step consists in exchanging all the elements of $\mathcal{F}$ with the ones in the first $|\mathcal{F}|$ positions of $\mathcal{F}$ and then sort them into the next $|\mathcal{F}|$ positions of $\mathcal{F}$ by repeated scans (we do the same in the second phase with the reset operation). Therefore, we skip the details of this phase.

14.3.1.4 Maintaining the invariants: deletions

Let us assume that $v$ has to be removed from the free set $\mathcal{F}$. There are four phases in this process. The last three are symmetrical to the last three for the insertion. (e.g. the third phase is about eliminating a dummy chunk instead of creating one and the fourth phase is about merging two free sets instead of splitting one). Therefore we do not describe them in this short presentation. The first phase has more significant differences. Let us give some details for it.

Let us suppose $v \in \text{Twin}(\mathcal{F})$. Let $\mathcal{W}$ be the twin set of $v$ and let $j$ be its home’s position in $\text{hubtwin}(\mathcal{F})$. If $\mathcal{W}$ is employed, we have a first step to keep the searchability and a second step with the actual deletion.

1. We look into the bitmask in $\text{hubenc}(\mathcal{F})$ for an available twin set: let $\mathcal{W}'$ be one of them and let $j'$ be its home’s position. Let $\mathcal{F}'$ be the free set where $\text{hubtwin}(\mathcal{F})[j + 1]$ belongs and let $\mathcal{T}'$ be the triplet set such that $\beta(\mathcal{T}') = \text{hubtwin}(\mathcal{F})[j + 1]$ with its home’s position $i'$.

   (a) We exchange $\text{hubtwin}(\mathcal{F})[j + 1]$ and $\text{hubtriplet}(\mathcal{F}')[i' + 1]$.

   (b) We exchange $\text{hubtriplet}(\mathcal{F}')[i' + 1]$ and $\text{hubtwin}(\mathcal{F})[j' + 1]$ and mark $\mathcal{W}'$ as “employed”.

Now both the elements of $\mathcal{W}$ are at home again and the searchability of $\gamma(\mathcal{T}')$ is guaranteed.

2. We look into the bitmask in $\text{hubenc}(\mathcal{F})$ for an employed triplet set. Let it be $\mathcal{T}$ and let $i$ be its home’s position in $\text{hubtriplet}(\mathcal{F})$.

   (a) We exchange $\text{hubtriplet}(\mathcal{F})[i + 2]$ with $v = \text{hubtwin}(\mathcal{F})[j + 1]$ and mark $\mathcal{W}'$ as “invalid”.

   (b) We exchange $v = \text{hubtriplet}(\mathcal{F})[i + 2]$ with the last element $l$ of $\text{spot}_1$.
(c) Let $\mathcal{F}''$ and $\mathcal{F}'''$ be the free sets of $\text{hubtriplet}(\mathcal{F})[i+1]$ and $l$, respectively.
Let $i''$ and $i'''$ be the home positions of the twin set of $\text{hubtriplet}(\mathcal{F})[i+1]$ and of the triplet set of $l$, respectively.

i. We exchange $\gamma(\mathcal{T})$ (which is in $\text{spot}(\mathcal{F}'')$) with $l = \text{hubtriplet}(\mathcal{F})[i+2]$.

ii. We exchange $\text{hubtriplet}(\mathcal{F})[i+1]$ with $\text{hubtwin}(\mathcal{F}'')[i''+1]$ (now all the elements of $\mathcal{T}$ are at home) and mark $\mathcal{T}$ as “available”.

iii. We exchange $\text{hubtwin}(\mathcal{F}'')[i''+1]$ with $\text{hubtriplet}(\mathcal{F}''')[i'''+1]$ to guarantee the searchability of $l$.

3. Finally, $v$, which now is between $\mathcal{L}$ and $\mathcal{F}$, is exchanged with the last element in $\mathcal{F}$ and is deleted.

The cases where $v \in \text{Triplet}(\mathcal{F})$ and $v \notin \text{Triplet}(\mathcal{F}) \cup \text{Twin}(\mathcal{F})$ are shorter because there is no need for the first step.

**Lemma 14.2** If $\mathcal{C}$ can route a search toward a free set in $O(\log N)$ comparisons, updating $\mathcal{A}$ costs $O(\log N)$ comparisons and $O(1)$ exchanges in amortized sense.

### 14.3.2 The chunk area

Once the problem of creating chunks efficiently has been solved in $\mathcal{A}$, $\mathcal{C}$ can be organized with techniques widely used outside the Implicit Model. We want to solve the following problem.

We have a set of $m$ units, $O(m)$ space and we want to organize the units in a dictionary supporting:

(i) worst case searches in $O(\log m)$ unit comparisons and $O(1)$ accesses to auxiliary data; 

(ii) amortized updates in $O(\log m)$ unit comparisons, $O(1)$ unit moves and $O(1)$ accesses to auxiliary data.

This problem can be solved using the well-known padded array [Itai, Konheim, and Rodeh, 1981, Willard, 1982, Bender, Demaine, and Farach-Colton, 2000] in a two-level scheme:

- upper level with a padded array containing $O(m/\log^h m)$ units;

- lower level with small balanced trees with logarithmic branching factor and $\Theta(\log^h m)$ units each.
The solution to this problem can be used as a black-box for organizing $C$. The ratio between dummy chunks and chunks in $C$ can be any chosen constant. Therefore, we can simulate an auxiliary space linear in the number of chunks with the dummy chunks playing the role of “empty locations”. When a chunk $c$ needs to be moved into an “empty location”, it is exchanged with the corresponding dummy chunk $d$. To keep $d$ searchable we need to update $O(1)$ encoded pointers of the list of dummy chunks of the free set of $d$.

That said, it is easy to see how the abstract problem’s costs translate into our costs. Given the disjointness of the chunks:

- The number of comparisons is of the same order of the number of unit comparisons.
- Any unit move translates into $O(\log N)$ element exchanges.
- Any access to auxiliary data translates into $O(\log N)$ comparisons or exchanges in our case.

Therefore, we have the following lemma:

**Lemma 14.3** $C$ can be organized so that searches cost $O(\log N)$ comparisons in the worst case and insertions and deletions of chunks costs $O(\log N)$ comparisons and exchanges in amortized sense.

Since any free set can split or merge only after it has been subject to $\Omega(\log N)$ update operations since the last splitting or merging, by Lemmas 14.1, 14.2 and 14.3 we can conclude that:

**Theorem 14.1** There exists an implicit dictionary supporting searches with $O(\log |S|)$ comparisons in the worst case and updates with $O(1)$ exchanges and $O(\log |S|)$ comparisons in amortized sense.

### 14.4 Worst Case Updates

In this section we prove that there exists an implicit dictionary that can be searched in $O(\text{polylog}|S|)$ time and updated in $O(1)$ exchanges and $O(\text{polylog}|S|)$ comparisons in the worst case. To keep the description as simple as possible, we do not try to minimize the exponents of the polylogarithmic bounds. Thanks again to the structure replication technique (Section 14.2) we focus on the problem where $|S| \leq N$; \(n\) and \(k\) still denote the cardinality of $S$ and the chunk size $a \log N$, respectively.

The permutation is once again divided into $C$ and $\mathcal{A}$. The areas have the same roles they had in Section 14.3: $C$ routes the searches, and does not change very often; $\mathcal{A}$ changes at any update and creates (and destroys) the chunks. Once we have the aggregating capabilities of $\mathcal{A}$, organizing $C$ becomes a fairly standard task. Unlike the previous structure, the free sets are completely contained in $\mathcal{A}$ and $C$ contains only chunks.
14.4.1 Free sets and lists of pseudo chunks

Let us assume that the chunk area is capable of routing the search for an element toward a free set in $O(\log^2 N)$ comparisons.

14.4.1.1 The invariants for $A$

**Partitioning of a free set.** A free set $F$ is partitioned into three subsets.

- The set of *loose elements*, denoted by $\text{Loose}(F)$, has a fixed number of elements (unlike the previous structure): $|\text{Loose}(F)| = 2k$.
- The set of *linking elements*, denoted by $\text{Link}(F)$, has a fixed cardinality too: $|\text{Link}(F)| = 6k + 12$.
- The set of *steady elements*, denoted by $\text{Steady}(F)$, contains the rest of the elements of $F$ and we have that $\min(F), \max(F) \in \text{Steady}(F)$.

**Pseudo chunks.** $A$ contains *pseudo chunks* of $k$ elements each except for the last one, the *incomplete pseudo chunk*. Pseudo chunks are not disjoint intervals. There are three types of pseudo chunks.

- The *steady chunks* have only steady elements from the same free set.
- The *linking chunks* may contain steady elements of one free set and linking elements, possibly from more than one free set.
- The *I/O chunks* may contain steady elements of one free set and loose elements of many free sets; an I/O chunk without steady elements is *free*.

**Lists of pseudo chunks.** Any free set $F$ is associated with a bidirectional linked list of pseudo chunks denoted by $\text{list}(F)$. The pointers of the list are encoded in the chunks themselves. As for the hub in the previous structure, $\text{list}(F)$ contains most of the elements of $F$ but not all of them and it may contain some elements from other free sets.

- The first six pseudo chunks of $\text{list}(F)$ are linking chunks and they contain only linking elements (no steady elements for them). They are denoted by $ch_1^M(F)$, $ch_2^M(F)$, $ch_3^M(F)$, $ch_4^M(F)$, $ch_5^M(F)$, $ch_6^M(F)$.
- The seventh pseudo chunk of $\text{list}(F)$, denoted by $ch_7^M(F)$, is a linking chunk too but only its first twelve elements are linking. The remaining elements belong to $\text{Steady}(F)$ and, in particular, $\min(F), \max(F) \in ch_7^M(F)$.
- The eighth pseudo chunk of $\text{list}(F)$ may be an I/O chunk containing at least one steady element; if that is the case, this chunk is denoted by $ch_8^I(F)$. 
• The subsequent chunks of $\text{list}(\mathcal{F})$ are all steady chunks with the possible exception of the last one.

• The last pseudo chunk of $\text{list}(\mathcal{F})$ may be an I/O chunk with at least one steady element and is denoted by $\text{ch}^o(\mathcal{F})$.

The pseudo chunks of $\text{list}(\mathcal{F})$ are connected with successor and predecessor pointers encoded into themselves. There is an exception to that rule, though. $\text{ch}^l(\mathcal{F})$'s predecessor ($\text{ch}^u(\mathcal{F})$) and successor (the first steady chunk in $\text{list}(\mathcal{F})$) pointers and the two corresponding back pointers to $\text{ch}^l(\mathcal{F})$ are only partially encoded, in the general case. Any of these four pointers respects the following condition:

$$
\text{Let } s \text{ be the number of steady elements in } \text{ch}^l(\mathcal{F}), \text{ only the first } \min(s, \log N) \text{ bits are encoded.} \quad (14.9)
$$

To maintain the connection between the chunks preceding and following $\text{ch}^l(\mathcal{F})$, pointers between $\text{ch}^u(\mathcal{F})$ and the first steady chunk of $\text{list}(\mathcal{F})$ are encoded in them.

**Distribution of loose, linking and steady elements.** Any steady element in $\text{list}(\mathcal{F})$ belongs to $\text{Steady}(\mathcal{F})$ and any element in $\text{Steady}(\mathcal{F})$ is contained in $\text{list}(\mathcal{F})$. That is not true for the other kinds of elements. Let us discuss the distribution of $\text{Loose}(\mathcal{F})$ and $\text{Link}(\mathcal{F})$ inside and outside $\text{list}(\mathcal{F})$ and the presence in that list of loose and link elements not in $\mathcal{F}$.

Let us focus on the first six chunks of a list. In order to be able to search among the loose elements, any I/O chunk in $\mathcal{F}$ has to be managed by one list. A list can manage at most four I/O chunks in $\mathcal{F}$ and anyone of them is managed with the help of one of the first four chunks of the list.

The following conditions completely characterize the distribution of linking elements into the first six linking chunks of a list. Let us consider a generic free set $\mathcal{F}$ with its list $\text{list}(\mathcal{F})$ and suppose that $\text{ch}^u_1(\mathcal{F})$ is used to manage an I/O chunk $c$. We have that:

(i) For any loose element $l$ such that $l \in c$ and $l \in \text{Loose}(\mathcal{F})$, there exist two linking elements $t \in \text{Link}(\mathcal{F})$ and $t' \in \text{Link}(\mathcal{F}')$ such that $t' \in \text{ch}^u_2(\mathcal{F})$ and $t \in \text{ch}^u_1(\mathcal{F})$.

(ii) The number of linking elements in $\text{ch}^u_1(\mathcal{F})$ that do not belong to $\text{Link}(\mathcal{F})$ is equal to the number of loose elements not in $\text{Loose}(\mathcal{F})$ that are contained in the I/O chunk $c$ managed by $\text{list}(\mathcal{F})$ with the help of $\text{ch}^u_1(\mathcal{F})$.

(iii) The number of linking elements in $\text{ch}^u_1(\mathcal{F}) \cup \text{ch}^u_2(\mathcal{F})$ that do not belong to $\text{Link}(\mathcal{F})$ is equal to the number of loose elements in $\text{Loose}(\mathcal{F})$ that are contained in I/O chunks not managed by $\text{list}(\mathcal{F})$. 
A pointer to the managed chunk \( c \) is encoded in the helper chunk \( ch_i^u (F) \) of \( \text{list}(F) \). There is an exception to that rule, though. The I/O chunk \( ch^o (F') \) of any free set \( F' \) is directly linked to \( \text{list}(F') \) with an encoded pointer. If \( \text{list}(F) \) manages \( ch^o (F') \) with \( ch_i^u (F) \) then there is only a partially encoded pointer in \( ch_i^u (F) \). The following condition holds:

\[
\text{Let } l \text{ be the number of loose elements in } ch^o (F), \text{ only the first } \min(l, \log N) \text{ bits are encoded in } ch_i^u (F). \tag{14.10}
\]

Let us focus on the seventh linking chunk of \( \text{list}(F) \), \( ch_i^u (F) \) and its twelve linking elements. Any list that is managing less than four I/O chunks is in the free manager list \( L_M \). The free manager list is not built by encoding auxiliary information (i.e. the pointers). The bidirectional links of \( L_M \) are implemented by exchanging pairs of linking elements.

Let \( F' \) and \( F'' \) be two free sets whose lists are adjacent in \( L_M \), with \( \text{list}(F') \) being the predecessor \( \text{list}(F'') \). We have that \( ch_i^u (F') [2] \in \text{Link}(F'') \) and \( ch_i^u (F'') [1] \in \text{Link}(F') \). \tag{14.11}

Therefore, for any free set \( F' \) whose list is in \( L_M \), a pair of linking elements in \( \text{list}(F') \) is used: the first one is for the predecessor of \( \text{list}(F') \) in \( L_M \) and the second one is for the successor. Following a link in \( L_M \) costs \( O(\log^2 N) \) comparisons (by the assumption on \( \mathcal{E} \)).

Any list that is managing a free I/O chunks is in the free I/O chunk list \( L_{i/o} \). Links in \( L_{i/o} \) are implemented in the same way the ones in \( L_M \) are. In this case, for any free set \( F' \) whose list is in \( L_{i/o} \), the third and fourth linking elements in \( ch_i^u (F') \) are used.

The fifth and sixth linking elements of \( ch_i^u (F') \) of any free set \( F' \) will be used to implement the links of a set of lists whose structure and purpose will be described in Section 14.4.2. The last six linking elements of chunks \( ch_i^u (\ast) \) are used to make connections between pairs of lists. Let us suppose that \( F \) has the I/O chunk \( ch^o (F) \) in its list.

Let \( F' \) be the free set whose list is managing \( ch^o (F) \) and \( ch_i^u (F') \) be the linking chunk of \( \text{list}(F) \) helping to manage \( ch^o (F) \). We have \( ch_i^u (F') [6 + i] \in \text{Link}(F) \) and \( ch_i^u (F) [11] \in \text{Link}(F') \). \tag{14.12}

The case where \( \text{list}(F) \) is managing \( ch^i (F) \) is analogous.

14.4.1.2 Searching in \( \mathcal{A} \)

Let us assume that the free sets and \( \mathcal{A} \) respect the invariants in Section 14.4.1.1 and that we have to search for an elements \( u \) into a free set \( F \).
1. We scan $\mathit{list}(\mathcal{F})$ looking for $u$. The I/O chunk $\mathit{ch}^1(\mathcal{F})$ (if any) is reached searching the free set whose list is managing it. To this purpose we use linking element $\mathit{ch}^n(\mathcal{F})$ [12].

2. We scan again the linking chunks of $\mathit{list}(\mathcal{F})$, $\mathit{ch}^1_1(\mathcal{F})$, $\mathit{ch}^1_2(\mathcal{F})$, $\mathit{ch}^2_1(\mathcal{F})$, $\mathit{ch}^2_2(\mathcal{F})$, $\mathit{ch}^3(\mathcal{F})$, $\mathit{ch}^8(\mathcal{F})$ and $\mathit{ch}^u(\mathcal{F})$, this time looking for elements $v \notin \mathcal{F}$ (to verify that we can use $\text{min}(\mathcal{F})$ and $\text{max}(\mathcal{F})$, residing in $\mathit{ch}^u(\mathcal{F})$, since free sets are disjoint). For any such $v$:

(a) We reach its free set $\mathcal{F}_v$.

(b) We scan $\mathit{ch}^1(\mathcal{F}_v)$, $\mathit{ch}^2(\mathcal{F}_v)$, $\mathit{ch}^3(\mathcal{F}_v)$, $\mathit{ch}^4(\mathcal{F}_v)$, $\mathit{ch}^5(\mathcal{F}_v)$ and $\mathit{ch}^6(\mathcal{F}_v)$ looking for $u$.

(c) If $v$ is contained in $\mathit{ch}^8(\mathcal{F})$ or $\mathit{ch}^2(\mathcal{F})$, we look for $u$ in the I/O chunks $\mathit{list}(\mathcal{F}_v)$ is managing.

**Lemma 14.4** If $\mathcal{C}$ can route a search toward a free set in $O(\log^2 N)$ comparisons, then any free set $\mathcal{F}$ can be searched in $O(\log^3 N + |\mathcal{F}|)$ comparisons.

### 14.4.1.3 Maintaining the invariants for $\mathcal{A}$

Let us show how to insert an element $v$ in a free set $\mathcal{F}$. $v$ resides in the last position of the permutation at first. Creating or removing connections based on exchanging elements (e.g. in $L_{I/O}$ and $L_M$ or the connections of pairs of lists where one list manages one of the I/O chunks of the other) can be done in $O(1)$ moves and $O(\log^3 N)$ comparisons in the obvious way and we will skip the details. We have four steps:

1. If $\mathit{list}(\mathcal{F})$ has not $\mathit{ch}^1(\mathcal{F})$, we look at the first list $\mathit{list}(\mathcal{F}')$ in $L_{I/O}$. We choose one of the free I/O chunks $\mathit{list}(\mathcal{F}')$ is managing and create the connection between $\mathit{list}(\mathcal{F}')$ and $\mathit{list}(\mathcal{F})$. If $\mathit{list}(\mathcal{F}')$ is managing more than one free I/O chunk, we leave it in $L_{I/O}$, otherwise we remove it.

2. We exchange $v$ with the leftmost loose element of $\mathit{ch}^1(\mathcal{F})$, let it be $l$. As a result of this exchange $l$ is not searchable anymore and $v \in \text{Steady}(\mathcal{F})$.

3. Let $\mathcal{F}_l$ be the free set of $l$. Let $\mathcal{F}'$ be the free set whose list is managing $\mathit{ch}^1(\mathcal{F})$ and $\mathit{ch}^4(\mathcal{F}')$ be the relative helper linking chunk in $\mathit{list}(\mathcal{F}')$. Finally, let $\mathcal{F}''$ be the free set whose list is managing the incomplete pseudo chunk (which is an I/O chunk with no steady elements) and $\mathit{ch}^8(\mathcal{F}'')$ be the relative helper linking chunk.

(a) We look for an element $t \in \mathcal{F}_l$ residing in $\mathit{ch}^4(\mathcal{F}')$ and for an element $t' \in \mathcal{F}'$ residing in $\mathit{ch}^8(\mathcal{F}_l) \cup \mathit{ch}^8(\mathcal{F}_l)$ and we exchange them.
(b) We choose an element $t'' \in \mathcal{F}''$ residing in $ch_j^{\mathcal{M}}(\mathcal{F}'')$ and we exchange it with $t$.

4. Let $s$ be the number of steady elements in $ch^i(\mathcal{F})$. Let $p_H$, $p$ and $p_I$ be the positions of $ch^u(\mathcal{F})$, the first steady chunk $c$ of list($\mathcal{F}$) and $ch^l(\mathcal{F})$ in the permutation, respectively.

(a) If $s \leq \log N$ we need to encode the $s$th bits of $p$ and $p_H$ in $ch^l(\mathcal{F})$ and the $s$th bit of $p_I$ in $ch^u(\mathcal{F})$ and in $c$.

(b) If $s = k$, list$(\mathcal{F})$ has no $ch^l(\mathcal{F})$ anymore since that formerly I/O chunk has become the new first steady chunk of list$(\mathcal{F})$. $\mathcal{F}'$’s list is not the manager of $ch^l(\mathcal{F})$ anymore and we have only to update the number of I/O chunks list$(\mathcal{F}')$ is ready to manage (two encoded bits in $ch^u(\mathcal{F}')$) and possibly insert list$(\mathcal{F}')$ in $L_m$.

The deletion of $v$ from $\mathcal{F}$ is symmetrical to the insertion. A peculiar aspect to be noticed is that deletions always happen in the I/O chunk $ch^o(\mathcal{F})$. If $v$ is not the rightmost steady element of $ch^o(\mathcal{F})$ $e$, we just exchange them and $e$ changes type from steady to the type of $v$.

We have also to consider that we need to have a loose element in the last position of the permutation at any time (to be exchanged with the departing element $v$). We can guarantee that condition simply by executing a process that exchanges the free I/O chunk managed by the first list in $L_{v,0}$ with the rightmost complete pseudo chunk in $\mathcal{A}$ in case the latter is not a free I/O chunk. The exchange of the chosen pseudo chunks is executed in parallel with the flow of normal update operations and in an incremental fashion (with $O(1)$ element exchanges at a time).

**Lemma 14.5** If $\mathcal{C}$ can route a search toward a free set in $O(\log^2 N)$ comparisons, updating a free set $\mathcal{F}$ in $\mathcal{A}$ costs $O(\log^3 N + |\mathcal{F}|)$ comparisons and $O(1)$ exchanges in the worst case.

### 14.4.2 Controlling the size of free sets and organizing $\mathcal{C}$

$\mathcal{C}$ is organized as a 2-3 tree with chunks as basic units: each node of the tree can contain one or two chunks (and have two or three children). That scheme guarantees the wanted routing complexity, $O(\log^2 N)$ comparisons, but it requires $O(\log^2 N)$ exchanges for any chunk inserted or removed.

To solve this problem, we control the size of free sets using a global strategy that allows splitting or merging one carefully chosen free set once every $\Omega(\log^2 N)$ updates of $\mathcal{A}$. The $O(\log^2 N)$ exchanges needed to insert or remove a chunk from $\mathcal{C}$ are executed incrementally over the subsequent $\Omega(\log^2 N)$ updates of $\mathcal{A}$ (that do not affect $\mathcal{C}$).
14.4. WORST CASE UPDATES

The idea at the core of the technique has been discovered independently in [Dietz and Sleator, 1987] and [Levcopoulos and Overmars, 1988a]. Intuitively, it can be seen as round-based pebble game.

- There are at most $m$ rounds and in each round $r$ pebbles are distributed in some piles.
- At the end of each round the largest pile is split into two piles of the same size.

In [Dietz and Sleator, 1987] and [Levcopoulos and Overmars, 1988a] it has been proved that the piles can contain $O(r \log m)$ pebbles, at any time and for any possible distribution scheme. This result is used in [Dietz and Sleator, 1987] and [Levcopoulos and Overmars, 1988a] to control the size of buckets subject to insertions while the deletions are treated by global rebuilding.

In the Implicit Model deletions have to be treated directly and therefore we follow the approach in [Dietz and Raman, 1994]. In that paper the authors treat both insertions and deletions symmetrically. They describe their approach under the assumption that the number of elements in the structure is maintained between $m/2$ and $2m$ for an initial value $m$ (they have rounds of fixed size $\Theta(\log m)$). When the number of elements goes outside the bounds, they use global rebuilding. In our case, we can exploit the structure replication technique (which is not suitable for the problem considered in [Dietz and Raman, 1994], the finger search problem).

Following [Dietz and Raman, 1994], we define

- the fullness of a free set $\mathcal{F}$ as $\phi(\mathcal{F}) = |\mathcal{F}| / \log^3 N$;
- the criticality of a free set $\mathcal{F}$ as $\rho(\mathcal{F}) = \max(0.7 - \phi(\mathcal{F}), \phi(\mathcal{F}) - 1.8, 0)$.

The size of the round is $r = b \log^2 N$, for a sufficiently small constant $b$. Once every $r$ updates in $\mathcal{A}$ we focus on the free set $\mathcal{F}$ with the largest $\rho(\mathcal{F}) > 0$. We have to be able to treat the following three cases.

1. If $\phi(\mathcal{F}) > 1.8$, we have to split $\mathcal{F}$ into two free sets $\mathcal{F}', \mathcal{F}''$ with roughly the same number of elements and a chunk $c$ such that $f' < f < f''$, for any $f' \in \mathcal{F}'$, $f'' \in \mathcal{F}'$ and $f \in c$. Then, $c$ has to be inserted in $\mathcal{C}$ and the $O(\log^2 N)$ exchanges needed to update the 2-3 tree will be executed incrementally over the next $r$ updates of the next round.

2. If $\phi(\mathcal{F}) < 0.7$ and one of the free sets adjacent to $\mathcal{F}$ has fullness $\geq 1$, we have to move elements into $\mathcal{F}$ until they have the same size.

3. If $\phi(\mathcal{F}) < 0.7$ and the previous case is not possible, we have to remove the chunk $c$ predecessor of $\mathcal{F}$ from $\mathcal{C}$ executing the $O(\log^2 N)$ exchanges needed to update $\mathcal{C}$ incrementally over the next $r$ updates of the next round. Then, $c$, $\mathcal{F}$ and the free set $\mathcal{F}'$ predecessor of $c$ have to be merged into a new free set.
Assuming that we are able to treat these three cases, then

$$\rho(\mathcal{F}) \leq 0.2$$

(see [Dietz and Raman, 1994]) and therefore

$$|\mathcal{F}| = \Theta(\log^3 N),$$

for any free set \( \mathcal{F} \). Since the elements of a free set are maintained completely unsorted, the first case is the most difficult. We have to treat it in two phases.

1. We extract chunk \( c \) from \( \mathcal{F} \) using \( O(\text{polylog}N) \) comparisons and \( O(k) \) moves, without partitioning the other elements according to \( c \) and executing the process incrementally over the next \( k \) updates of \( \mathcal{A} \) in the next round.

The rationale behind the first phase is that we have to insert \( c \) in \( \mathcal{C} \) within the next \( r \) updates in \( \mathcal{A} \) and therefore we cannot partition \( \mathcal{F} \) according to \( c \) while we are extracting it (\(|\mathcal{F}| = \Theta(\log^3 N) \) and we would not have \( c \) ready before the end of the round).

Since we are disregarding the exponent of the polylogarithmic bounds for the comparisons, the simplest way to accomplish the task of the first phase is the following:

(a) We find the median \( m \) by \(|\mathcal{F}|/2 \) repeated scans of \( \mathcal{F} \).

(b) We gather the remaining \( k - 1 \) elements adjacent (in the sorted order) to \( m \) with another \( k - 1 \) scans.

2. While \( c \) is being inserted in \( \mathcal{C} \) in time for the end of the round, the remaining elements are partitioned according to \( c \) into \( \mathcal{F}' \) and \( \mathcal{F}'' \) incrementally over the next \( q \log^3 N \) updates occurring in \( \mathcal{F} \), for an opportune constant \( q \).

Hence, the process of partitioning the remaining \( \Theta(\log^3 N) = \Theta(|\mathcal{F}|) \) elements of \( \mathcal{F} \) is done incrementally but locally. That can be done safely, because we know that none of the two new free sets \( \mathcal{F}' \) and \( \mathcal{F}'' \) that will come out from this partitioning process will have criticality greater than zero before that \( \Omega(\log^3 N) \) new updates occur in them.

In order to be able to select efficiently the free set with largest criticality, we maintain the lists of the free sets into \( O(\log^3 N) \) linked lists implemented in the same way \( L_{i_0} \) and \( L_m \) are (Section 14.4.1.1). We have one linked list \( L_i \) for any possible size \( i \) of a free set. The \( i \)th chunk in \( \mathcal{C} \) is the \textit{header} of \( L_i \). The first list \( \text{list}(\mathcal{F}) \) in \( L_i \) has exchanged one of its linking elements in \( ch^n(\mathcal{F}) \) with one of the elements of the header of \( L_i \). When a free set \( \mathcal{F} \) is updated, it is scanned to calculate its cardinality and \( \text{list}(\mathcal{F}) \) is detached from the current list and is attached to \( L_{|\mathcal{F}|} \).

\textbf{Theorem 14.2} There exists an implicit dictionary supporting...
• searches with $O(\text{polylog}|S|)$ comparisons,

• updates with $O(1)$ exchanges and $O(\text{polylog}|S|)$ comparisons

in the worst case.
Chapter 15

In the Worst-Case Scenario...

Abstract

In this chapter we close the open issue on implicit dictionaries left open in Chapter 13 and dating back to the sixties (see Chapters 3 and 11 for the history of the problem). A sequence of \( n \) elements can be sorted so that searching takes \( O(\log n) \) time. Alternatively, it can be organized as a heap so that inserting and deleting elements take \( O(\log n) \) time. We show that these bounds can be simultaneously achieved in the worst case for searching and updating by suitably maintaining a permutation of the \( n \) elements in the sequence. The resulting data structure is called implicit and, at any moment of its existence, it uses just \( O(1) \) auxiliary locations beside the \( n \) locations for the input elements.

The data structure is also cache optimal in the Cache-Oblivious model, attaining \( O(\log_B n) \) block transfers in the worst case for any (unknown) value of the block size \( B \), without wasting more than \( O(1) \) auxiliary locations at the highest (and slowest and largest) level of the memory hierarchy (see Chapter 2 for a precise definition of the Cache-Oblivious model).

The presentation in this chapter is based on the paper [Franceschini and Grossi, 2003c] (WADS 2003).

15.1 Introduction

In this chapter we consider again the classical dictionary problem in which a set of \( n \) distinct elements \( a_1, a_2, \ldots, a_n \) is maintained over a total order, where the only operations allowed on the elements are moves (or exchanges) and comparisons using the standard RAM model (see Chapter 2) of computation [Aho, Hopcroft, and Ullman, 1976]. The dictionary supports the operations of searching, inserting and deleting an arbitrary element \( x \).

Implicit dictionaries solve the problem by maintaining a plain permutation of \( a_1, a_2, \ldots, a_n \) to encode the data structures [Munro and Suwanda, 1980]. When
employed in this context, heaps [Williams, 1964] have the drawback of requiring $O(n)$ time for searching, while inserting or deleting an element in the middle part of sorted sequences may take $O(n)$ time [Knuth, 1973]. A long-standing question is whether there exists an organization of the elements in a sequence of $n$ locations combining the best qualities of sorted sequences and heaps, so that each operation requires $O(\log n)$ time. Previous work since the sixties did not achieve polylog time in both searching and updating.

Let us briefly recap the milestones since the first implicit dictionary achieving polylog time in both searching and updating (for a detailed history of the problem see Chapters 3 and 11).

**The Implicit AVL Tree.** The first milestone toward the direction of an optimal implicit dictionary is the implicit AVL tree in [Munro, 1986], showing for the first time that polylog time is possible, namely $O(\log^2 n)$, by encoding bits in chunks of $O(\log n)$ permuted elements. It was conjectured a bound of $\Theta(\log^2 n)$ because $\Theta(\log n)$ pointers of $\Theta(\log n)$ bits are decoded/encoded in the worst case to execute an operation in the implicit AVL tree.

**The Implicit B-Tree.** The second milestone is the implicit B-tree that supports all the operations in $O(\log^2 n / \log n \log n)$ time in the worst case [Franceschini, Grossi, Munro, and Pagli, 2002] (see Chapter 11). Notwithstanding the small improvement in main memory, this recent result disproved the conjecture of the eighties, making viable the possibility of getting a bound of $O(\log n)$. The implicit B-tree uses nodes of relatively large fan-out that are augmented with a permuted directory to support fast searching inside each node. For a known block size $B = \Omega(\log n)$, it supports the operations in $O(\log_B n)$ block transfers like regular B-trees, while scanning $r$ contiguous elements requires $O(\log_B n + r/B)$ block transfers.

**The Exponential Implicit Tree.** The Exponential Implicit Tree [Franceschini and Grossi, 2003a] (see Chapter 12) comes close to the objective being searchable in $O(\log n \log \log n)$ in the worst case and updatable in $O(\log n \log \log n)$ time in amortized sense. The issue of amortized analysis is introduced for the first time in the implicit dictionary problem and an exponential tree of height $O(\log \log n)$ is used, exploiting in-place algorithms to amortize the bounds and introducing different kinds of chunks of $O(\log n)$ contiguous elements to delay the expensive reorganizations of the updates.

**The Flat Implicit Tree.** The Flat Implicit Tree [Franceschini and Grossi, 2003b] (see Chapter 13) represents the third milestone. It is the first implicit data structure with optimal $O(\log n)$ time for searching and $O(\log n)$ amortized time for updating. The structure obtains $O(\log n)$ amortized time with a two-layer tree of constant height (except very few cases), adapting the redistribution technique of [Bender,
Demaine, and Farach-Colton, 2000, Itai, Konheim, and Rodeh, 1981] to the implicit model. It also attains cache optimality in the Cache-Oblivious model (see Chapter 2) supporting search in $O(\log_B n)$ cache faults and update in $O(\log_B n)$ cache faults in amortized sense.

The top layer uses a van Emde Boas permutation [Frigo, Leiserson, Prokop, and Ramachandran, 1999] of the elements as a directory, and the bottom layer introduces compactor zones to attain cache-obliviousness. Compared to implicit B-trees, the update bounds are amortized and scanning is not optimal. On the other hand, achieving an optimal scanning is still an open problem in explicit cache-oblivious dictionaries even with amortized update bounds of $O(\log_B n)$. The implicit B-tree attains this goal with worst-case bounds as it is aware of the block size $B$.

**Worst case optimal bounds, finally.** In this chapter we focus on the worst-case complexity of implicit dictionaries. The best worst case bound is that of $O(\log^2 n / \log \log n)$ with the implicit B-trees. For explicit cache-oblivious data structures, the best space occupancy in [Brodal, Fagerberg, and Jacob, 2002] is $(1 + \epsilon)n$ locations for any $\epsilon > 0$ with an $O(1 + r/B)$ scanning cost for $r$ elements, but the update bounds are amortized, whereas the worst-case result in [Bender, Cole, and Raman, 2002b] uses more space.

Here, we propose a new scheme for implicit data structures that takes $O(\log n)$ time and $O(\log_B n)$ block transfers in the worst case for any unknown $B$, as in the Cache-Oblivious model. The structure uses just $O(1)$ auxiliary locations at the highest (and slowest and largest) level of the multilevel memory hierarchy (the only level able to contain the whole input, see Chapter 2).

This closes the problem of determining a permutation of the elements in a sequence, so that both searching and updating are logarithmic in the worst case as explicit dictionaries.

We introduce new techniques to design our data structures. We use some spare elements and some chunks, called filling chunks, to allocate nodes of the tree in an implicit way. When we actually need a chunk, we replace the filling chunk with the routing chunk, and relocate the filling chunk. This can be seen as an extension of the internal buffering technique [Kronrod, 1969] (see Chapter 4). The internal buffering technique has never been used in the implicit data structures. The reason would seem obvious: the buffer elements are pushed around continuously during the execution of the algorithm and, for the computation, they are virtually indistinguishable from one another. This is terrible in an online setting like the one of a dictionary where, at any given moment any element in the structure has to be searchable. Here this belief is proven wrong and a sophisticated internal buffering technique is used to prove that worst case optimal upper bounds are possible for the implicit dictionary problem.

We also design a bottom layer that can be updated very quickly. We reuse techniques from previous work, but we apply them in a novel way since we have
to perform the memory management of the elements in the sequence. We have to carefully orchestrate data movement as we cannot leave empty slots in any part of the sequence.

The chapter is organized as follows. In Section 15.2, we review some basic techniques that we apply to implicit data structures. We then describe our main data structure in two parts, in Sections 15.3–15.4, putting them together in Section 15.5 for the sketch of the final analysis of the supported operations.

15.2 Preliminary Algorithmic Tools

Stolen bits and chunks of elements. As always, we will use the bit stealing technique (see [Munro, 1986] and Chapter 4). Adjacent elements in the sequence are grouped together into chunks, where each chunk contains $O(k)$ (pairwise permuted) elements encoding a constant number of integers and pointers, each of $b = O(\log n)$ bits. The elements in any chunk belong to a certain interval of values, and the chunks are pairwise disjoint when considered as intervals, thus yielding a total order on any set of chunks.

We introduce some terminology on the chunks to clarify their different use. We have two kind of chunks:

- **Routing** chunks that help us in routing the search of individual elements.

- **Filling** chunks that provide a certain flexibility in filling the entries of the sequence in that we can keep them in no particular order.

Filling chunks have a similar role to the one of buffer elements in the internal buffering technique that we have been using throughout this thesis. They will be used as placeholders in order not to move too much the routing chunks when an update is in order. As we will see, since we are in an online setting and the structure has to support the search query for any element at any time, accessing the filling chunk is done via the routing chunks and encoded pointers (which are a necessity since the filling chunks are scattered in memory, because of their own role in the structure).

The number of elements in a chunk is fixed to be either $k$ or $k - \alpha$ for a certain constant $\alpha > 1$, which is clear from the context. We also use a set of spare elements that can be individually relocated and referenced for a finer level of flexibility in filling the memory, associating $O(1)$ spare elements to some chunks. When considered as intervals, the chunks include the spare elements although the latter physically reside elsewhere in the sequence.

Known techniques. Our algorithms employ some powerful tools to achieve their worst-case and cache-oblivious bounds.
One tool is Willard’s algorithm [Willard, 1992] and its use in Dietz-Sleator lists [Dietz and Sleator, 1987]. Suppose we have a sequence Z of N slots (for a fixed N) storing a dynamic set S of n ≤ N objects, drawn from a totally ordered universe. At any time, for every pair of object s₁, s₂ ∈ S, if s₁ < s₂ then the slot storing s₁ precedes that storing s₂. The data structure proposed by Willard in [Willard, 1992] achieves this goal using a number of $O(\log^2 N)$ arithmetical operations, comparisons and moves, in the worst case, for the insertion or the deletion of an individual object in Z.

In our use of Willard’s scheme, the routing chunks play the role of the full slots while the filling chunks that of the empty slots. It is possible to insert a new routing chunk (thus replacing a filling chunk that goes elsewhere) and delete a routing chunk (putting in its place a filling chunk taken somewhere). These operations have to maintain the invariant of Willard’s scheme according to the total order of the routing chunks stored in the slots. Since the slots are of size $O(k)$ in our case, the bounds of Willard’s scheme have to multiplied by a factor of $O(k)$ time or $O(k/B)$ block transfers to insert or delete a routing chunk.

Another useful tool is the van Emde Boas (VEB) layout of Prokop [Prokop, 1999, Frigo, Leiserson, Prokop, and Ramachandran, 1999] (see Chapters 7 and 13). Given a complete binary search tree T with $n = 2^i - 1$ nodes, the VEB layout of T allows for searching with $O(\log_B n)$ block transfers in a cache-oblivious fashion. Brodal et al. [Brodal, Fagerberg, and Jacob, 2002] describe how to avoid pointers in the VEB layout, still using extra memory. In [Franceschini and Grossi, 2005b] (see Chapter 13) it is showed how to make the VEB layout implicit in the form of a VEB permutation of the n elements.

The last tool is for memory management of nodes of variable size with compactor lists [Franceschini, Grossi, Munro, and Pagli, 2002, 2004] (see Chapter 11) and compactor zones [Franceschini and Grossi, 2003b] (see Chapter 13). In the design of implicit data structures, nodes are sets of permuted elements that should be maintained as continuously as possible. For this, nodes of the same size are kept together in a segment of contiguous locations, the compactor zone, or in a linked list of fixed size allocation units, the compactor list.

Their use make possible to avoid the creation of empty locations during the operations since the nodes of the same size are collected together. However, when a node changes size, we have to relocate the node from one compactor zone (or list) to another. Since we want to achieve worst-case bounds, we will use those two techniques in two different cases.

- The compactor lists will be used to host nodes of size $\Theta(\sqrt{\log n})$, since they are efficient with small size nodes.

- The compactor zones will be used to host nodes of size $\Theta(\log n)$, since they can be incrementally maintained still permitting searching.
For larger nodes, we use a different approach described in Section 15.4.

15.3 Districts of Chunks

15.3.1 Portions

The sequence of \( n \) elements is partitioned into \( O(\log \log n) \) portions as in [Frederickson, 1983], where the \( p \)th portion stores \( 2^p \) elements, except the last portion, which can store less elements than expected. The last portion is the only one growing. When the last portion reach its maximum size, it is frozen and a new last portion will possibly start with the next operations on the structure.

In order to maintain the previous property, whenever we have to insert an element in the structure, we insert it into the last portion. The case of the deletion is slightly different. We delete the element from the portion it belongs. If that portion is not the last one then we insert in it an element deleted from the last portion (any element in the last portion would do). Searching is applied to each portion. By the exponentially-growing sizes of the portions, we have that if each portion of size \( m \) can be searched in \( O(\log m) \) time and \( O(\log_B m) \) block transfers then searching the whole sequence of \( n \) elements sums up to \( O(\log n) \) time and \( O(\log_B n) \) block transfers, which is the final cost of the supported operations.

In the rest of the chapter we focus on the last portion \( A \) of the sequence, assuming without loss of generality that \( A \) is a sequence of \( n \) elements, where \( N = 2^{2p} \) is the maximum size of \( A \) for some given integer \( p > 0 \) and \( n \leq N \) is sufficiently large to let us fix

\[
k = \Theta(\log N) = \Theta(\log n).
\]

(The implicit model assumes that \( A \) occupies just \( n + O(1) \) locations and that it can be extended to the right one location at a time up to \( n = N \) locations.) This condition is guaranteed using Frederickson’s partitioning.

15.3.2 Districts

The first \( O(\log N) \) elements of \( A \) form a preamble encoding some bookkeeping information for \( A \). We partition the rest of \( A \) into two parts:

- The layer \( \mathcal{D} \) of the districts.
- The layer \( \mathcal{B} \) of the buckets (yes, we have quite an imagination for names).

We defer the discussion of layer \( \mathcal{B} \) to Section 15.4. Here, we focus on the districts in layer \( \mathcal{D} \) in which we use chunks of size \( k - \alpha \) for a certain constant \( \alpha > 1 \).

We partition the initial part of layer \( \mathcal{D} \) into a number (at most logarithmic) of consecutive districts \( D_0, D_1, \ldots \), so that each \( D_i \) contains \( 2^i \) chunks and \( \Theta(2^i) \) spare
elements according to the invariants and properties we give next. We denote the 
zone of $\mathcal{D}$ to the right of the districts by $F$ (see Figure 15.1).

(i) **Chunks and spare elements.** The chunks in layer $\mathcal{D}$ are routing chunks 
and filling chunks, each with

$$\alpha = \Theta(1)$$

spare elements associated. The routing chunks occur *only in the districts*
$D_0, D_1, \ldots$, while the filling chunks can occur anywhere in $\mathcal{D}$ (i.e., both in
$D_0, D_1, \ldots$ and in $F$).

(ii) **Order of routing chunks.** The total left-to-right sequence of routing chunks 
among all districts in $\mathcal{D}$ is in sorted order, while the filling chunks are not in 
any particular order. Given any two routing chunks (as close as possible), the 
sequence of filling chunks between them can be arbitrarily long. As we will 
see, this is a direct consequence of using the algorithm in [Willard, 1982, 1992] 
to manage the insertions and deletions of new routing chunks in $\mathcal{D}$.

(iii) **Filling chunks.** With each routing chunk $c$, there are $\Theta(1)$ filling chunks
associated. Their number can range between two suitable constants, so that 
the overall number of filling chunks in $F$ is at least $2^{i+1}$, where $i+1$ is the 
current number of districts in $\mathcal{D}$. That is because, whenever the rightmost 
district, the one of size $2^i$ becomes full, we have to be ready to start a new 
district of size $2^{i+1}$, initially composed only by filling chunks.

The filling chunks associated with $c$ are *the nearest to $c$ in the total order of
the chunks*, and the pointers to them are encoded in $c$.

(iv) **Directories within the districts.** The first $\Theta(2^i)$ positions of each dis-

ctrict $D_j$ are initially occupied by some of the spare elements associated with
the filling chunks in $\mathcal{D}$. We require that, at any time, the number of these
positions is a multiple of the chunk size. The routing elements (that is, ele-
ments extracted from a routing chunk) that will be placed in these positions
(replacing some spare elements) will form a directory for quickly searching the

![Diagram](image-url)
routing chunks in \( D_j \). To achieve the cache optimality, the directory will be a VEB permutation (see [Franceschini and Grossi, 2003b] and Chapter 13) of the routing elements.

(v) **The preamble.** The routing chunks in \( D_j \) are to the right of their directory, and the first chunk \( c \) immediately after the directory is always routing. We maintain the smallest element of \( c \) as a spare element that is stored in the preamble of \( A \). In this way, we can discover in which district to search by first reading \( O(\log n) \) adjacent spare elements in that preamble.

The set \( \mathcal{W} \) of the first chunks of the districts will be searched in a simple way: first, the directory in the preamble is scanned to identify the unique chunk \( c \in \mathcal{W} \) that could possibly contain the searching element; then this \( c \) is scanned.

(vi) **Spare elements.** The rest of the spare elements are in \( F \), at the beginning (a multiple of the chunk size) and at the end (any number of them). We incrementally move the spare elements from the end of \( F \) to the beginning of \( F \) (or vice versa), when adding (or removing) routing chunks in \( D_i \), the rightmost district in \( \mathcal{D} \).

When the number of routing chunks in \( \mathcal{D} \) is sufficiently large, the elements at the beginning of \( F \) are already organized to create \( D_{i+1} \), thus shortening \( F \) and preparing for \( D_{i+2} \) (if any). An analogous situation occurs when \( D_i \) has no more routing chunks, and \( D_{i-1} \) becomes the rightmost district.

It is worth to point out the two main differences between districts and portions.

- A district is twice as large as its predecessor while the size of a portion is the square of the size of its predecessor. Therefore we cannot search in the districts the way we search in the portions, that is doing a search in each portion. That brings us to the second difference.

- Any routing element in a district \( D_j \) is greater (less) than any routing element in \( D_{j-1} \) (\( D_{j+1} \)) while there no relation whatsoever between the elements of any pair of portions. That allow us to collect a small directory (see invariant (v)) to direct any search into only one district.

**15.3.3 How to search an element in the districts**

The organization mentioned in points (i)–(vi) is not yet suitable for searching (with the exclusion of the set \( \mathcal{W} \) of the first chunks of the districts, see invariant (v)). As mentioned in point (v), we can identify efficiently in which district, say \( D_j \), we must go on searching. Once we identify the correct routing chunk in \( D_j \), it is a simple task to examine its associated filling chunks (see invariant (iii)). Consequently, it suffices to show how to search an element in \( D_j \), so that a routing chunk can be identified in \( O(\log n) \) time and with \( O(\log_B n) \) block transfers.
15.3. DISTRICTS OF CHUNKS

The directory and the implicit tree associated with the chunks. With this goal in mind, we set up the directory of \( D_j \) following the VEB permutation mentioned in Section 15.2. Since this scenario is well exploited and studied in Chapter 13 (see [Franceschini and Grossi, 2003b]), we do not describe the one-to-one mapping between the VEB permutation in the directory and the \( 2^j - 1 \) nodes of a complete binary tree.

The complete binary tree we are referring to is the one associated to the \( 2^j - 1 \) chunks, both routing and filling, in \( D_j \) (we leave out the first chunk of \( D_j \) since the set \( \mathcal{W} \) of the first chunks of the districts can be searched simply because of the role of its members in the creation of the preamble, see invariant (v)).

- The root of the tree is associated with the \( 2^{j-1} \)th chunk of \( D_j \).
- The left child of the root is associated with the \( 2^{j-2} \)th chunk of \( D_j \).
- The right child is associated with the \( 2^{j-1} + 2^{j-2} \)th one.
- ...so forth, recursively...

In turn, the nodes of the tree are in one-to-one correspondence with the \( 2^j - 1 \) chunks in \( D_j \).

How to make the implicit tree searchable. Although the tree is not yet a search tree (because not all the chunks in \( D_i \) are routing chunks), we can activate the search path in it for each routing chunk in \( D_i \). In the beginning, the directory is made up of spare elements from filling chunks and no search path is active.

Given a routing chunk \( c \), let \( u \) be the corresponding node in the complete binary tree encoded by the VEB permutation. Since \( c \) contains \( \Theta(\log n) \) elements and the chunks are disjoint as intervals, we can exchange the smallest elements in \( c \) with the spare elements found in the upward path from \( u \) (we will need \( O(\log |D_i|) \) elements for this task, hence the elements in \( c \) are sufficient to “connect” it, even directly to the root).

The general property we maintain is that the exchanged elements of \( c \) must guide the search towards \( u \) when searching elements that fall inside \( c \) as an interval. In other words, the paths activated for all the routing chunks form a search tree. The nodes along these paths contain elements taken from the routing chunks, while the rest of the elements in the directory are spare elements from the filling chunks.

The routing chunks host temporarily the spare elements that they exchanged in the directory. The spare elements hosted inside the routing chunk \( c \) can still be retrieved from the pointers encoded in their filling chunks. Vice versa, the elements in \( c \) that are currently in the directory stay along some of the nodes in the upward path from \( u \), and they will be eventually retrieved during a search for an element whose position is within the interval represented by \( c \). Therefore, retrieving the
routing elements in the directory of any district costs \( O(\log n) \) time and \( O(\log_B n) \) block transfers.

With this organization of the directory, searching is a standard task with the VEB permutation as each node has now a routing element when needed. What can be observed is that we actually exchange elements in pairs to encode a flag bit indicating whether \( u \) has associated spare elements or elements from a routing chunk. The rest of the searching in the VEB permutation is unchanged.

**Lemma 15.1** Any element \( x \) can be searched in \( \mathcal{D} \) in \( O(k + \log n) \) time and with \( O(k/B + \log_B n) \) block transfers, identifying the (routing or filling) chunk that contains \( x \) (as an interval).

15.3.4 How to update the districts

In order to maintain the invariant for \( \mathcal{D} \), we apply Willard’s algorithm (see Section 15.2) to any single district (without its directory) as a black-box. Therefore the number of routing chunks within each district \( D_i \) is dictated by this algorithm. With the valuable help of Willard’s algorithm, we will show how to maintain the following invariant for the whole sequence of districts. Let \( D_i \) be the rightmost district:

\[
\text{Each of } D_0, D_1, \ldots, D_{i-1} \text{ has the maximum number of routing chunks according to Willard’s algorithm, and the rest are filling chunks. Instead, } D_i \text{ does not host the maximal number of routing chunks allowed by Willard’s algorithm.} \tag{15.1}
\]

**Willard's algorithm for a single district.** The structural information needed by Willard’s algorithm to manage any district \( D_j \) can be easily encoded and maintained in this very district. The algorithm needs \( O(m \log m) \) bits of auxiliary information, where \( m \) is the number of objects maintained in sorted order. Since in our case the objects maintained by Willard’s algorithm are chunks of \( k = \Theta(\log n) \) distinct elements each, the number of distinct pairs in \( D_j \) is sufficient to encode by bit stealing all the auxiliary information needed by the algorithm (of course we will have to watch out for the inevitable slowdown in the analysis).

Let us focus on the rightmost district \( D_i \) and how to maintain its organization of the elements when the routing chunks are added or removed.

The first routing chunk \( c \) inserted in \( D_j \) has to be to the immediate right of the directory and is not involved in it because it will be searched using the small directory in the preamble (see invariant \( (v) \)). To this purpose we exchange the smallest element in \( c \) with the leftmost spare element in the preamble.

Let us examine the general case of an insertion or deletion of a routing chunk in \( D_i \) occurring at any moment of its existence. Willard’s scheme preserves the distribution of routing chunks (objects) among the filling chunks (free slots) in \( O(\log^2 n) \) steps (since we are dealing with objects that are chunks, those steps do not have \( O(1) \)
cost). In each step, it relocates a routing chunk $c$ from one position to another in $D_i$ by exchanging it with a filling chunk $c'$. This step requires exchanging the elements of the two chunks incrementally, then performing a search for any element of $c'$ in order to locate and re-encode the incoming pointer to $c'$ (we point out that the filling chunks are connected with the structure with encoded pointers).

**How to search in a district.** However this alone does not guarantee searchability as we need to update the VEB permutation of $D_i$. We therefore divide the step in further $O(\log n)$ substeps that essentially remove $c$ and its search path in the directory and re-insert it into another position, along with its new search path. Specifically, in each substep:

1. We retrieve one of the $O(\log n)$ elements of $c$ that are in the directory and put it back in $c$ by exchanging it with the corresponding spare element temporarily hosted in $c$ (note that each spare element requires a search).

2. Next, we exchange $c$ with $c'$, and propagate the same exchange in the VEB permutation of the directory.

3. We then run further $O(\log n)$ substeps to trace the path for the new position of $c$ and exchange its elements so that it is now searchable in the directory.

During the substeps, $c$ is the only chunk not searchable in $D_i$. But we can encode a pointer to it in the preamble, so that searching treats $c$ as a special case. When the task for $c$ is completed, Willard’s scheme takes another routing chunk, which becomes the new special case.

In summary, each of the $O(\log^2 n)$ steps in Willard’s scheme can be divided into further $O(\log n)$ substeps, each costing $O(k + \log n) = O(\log n)$ time and $O(k/B + \log_B n) = O(\log_B n)$ block transfers. It is crucial noting that after each substep, we can run the search as stated in Lemma 15.1 plus the special case for the current $c$.

**How to maintain invariant 15.1.** When inserting and deleting routing chunks in a district $D_j$, for $j < i$, we perform the same steps as in $D_i$. However we must preserve the property that the number of routing chunks is maximal. Let us suppose we have to insert a routing chunk $c$ in $D_j$. We first use Willard’s algorithm to extract the largest routing chunk in $D_j$, let it be $c'$ and to insert $c$ in $D_j$ (we may have that $c = c'$). Then, we extract the largest routing chunk in $D_{j+1}$, let it be $c''$ and insert $c'$ in this district. As it should be clear, we go on that way until we reach the rightmost district $D_i$ that will be the one growing the number of routing chunks. The deletion of a routing chunk from $D_j$ is symmetrical.

Since there are $O(\log n)$ districts, we have an extra logarithmic factor in the number of substeps for the districts in the entire layer $\mathcal{D}$. 
Theorem 15.1 Layer \( \mathcal{D} \) can be maintained under insertion and deletion of single routing chunks and filling chunks by performing no more than \( O(\text{polylog}(n)) \) incremental substeps, each requiring, in the worst case,

- \( O(\log n) \) time
- and \( O(\log_B n) \) block transfers.

After executing each single substep, searching an element for identifying its chunk takes \( O(\log n) \) time and \( O(\log_B n) \) block transfers for any (unknown) value of \( B \).

15.4 Indirection with Dynamic Buckets

The layer \( \mathcal{B} \) of the sequence \( A \) introduced in Section 15.3 is populated with buckets containing from \( \Omega(k^{d-1}) \) to \( O(k^d) \) elements, for a constant \( d \geq 5 \). Each bucket is a balanced tree of constant height. A tree is maintained balanced by split and merge operations applied to the nodes. Unlike regular B-trees, the condition that causes a rebalancing for a node is defined with a parameter that depends on the whole size of the subtree rooted at the node (e.g., see the weight-balanced B-trees [Arge and Vitter, 1996]). We now give a high level description of the buckets assuming that the size of each chunk is \( k \) and that we can rely on a suitable memory layout of the nodes. We postpone the discussion of the layout to Section 15.4.2, which is crucial for both implicitness and cache-obliviousness.

15.4.1 The structure of the buckets

A bucket has a constant number \( d \) of levels. Each bucket is associated with either a routing chunk or a filling chunk of layer \( \mathcal{D} \), and all the elements in the bucket are greater than those in that chunk.

Leaves. A leaf of a bucket contains from \( k \) to \( 16k \) elements. Moreover, a leaf has associated a maniple that contains from \( \sqrt{k} \) to \( 5\sqrt{k} \) elements and a number of filling chunks that ranges from \( r \) to \( 4r \) for a suitable constant \( r \). The exact value of \( r \) concerns the memorization of the internal nodes of the buckets, as clarified in Section 15.4.2. The filling chunks of a leaf \( l \) are maintained in increasing sorted order in a linked list, say \( f_1, \ldots, f_s \). Letting \( m \) be the maniple associated with \( l \), we have that

- \( f_j \) is the predecessor of \( f_{j+1} \) for \( 1 \leq j < s \), and
- for each choice of elements \( x \in f_s, x' \in l \) and \( x'' \in m \), we have \( x < x' < x'' \).

As we shall see, each leaf \( l \), its maniple \( m \) and its filling chunks are maintained in a constant number of zones of contiguous memory. Hence, searching in these objects requires a total of \( O(k + \log n) \) time and \( O(k/B + \log_B n) \) block transfers in the worst case.
15.4. INDIRECT WITH DYNAMIC BUCKETS

Internal nodes. An internal node contains routing chunks and filling chunks, and
the pointer to the jth child is encoded by $O(\log n)$ elements in the jth chunk, which
must be routing. Following an approach similar to that in [Arge and Vitter, 1996],
we define the weight $w(v)$ of an internal node $v$ at level $i$ (here, the leaves are at
level 1) as the number of elements in the leaves descending from $v$. For any internal
node $v$ at level $i$, we maintain the following invariant for the weight

$$4^ik^i \leq w(v) \leq 4^{i+1}k^i.$$  

For this reason the number of routing chunks of an internal node can range from $k$

to $16k$.

For the root of a bucket, we only require the upper bound on its weight, since
the bucket size can be $\Omega(k^{d-1})$ and the number of chunks in the root can be $O(1)$.

Since the number of elements in an internal node is $O(k^2) = O(\log^2 n)$, in order
to pay $O(\log_B n)$ block transfers when searching and updating an internal node $v$,
we maintain a directory of $\Theta(k)$ elements in $v$, analogously to what done in [Frances-
chini, Grossi, Munro, and Pagli, 2002, 2004] (see Chapter 11). Thus the chunks of $v$
are not maintained in sorted order, but their order can be retrieved by scanning
the directory in $v$. In this way, any operation on $v$ involves only $O(1)$ chunks and
portions of $\Theta(k)$ contiguous elements each.

Handling insertions and deletions. If we ignore the memory management, the
insertion or the deletion of an element in a bucket is a relatively standard task.
If $x$ is the element to be inserted into chunk $c$, the root of a bucket, we place $x$ in
its position inside $c$, shifting at most $k$ elements to extract the maximum element
in that chunk. We obtain the new element $x$ to be inserted into the node whose
pointer is encoded in $c$. In general, inserting $x$ into a chunk of an internal node $u$
go in the same lines.

When we reach a leaf $l$, we perform a constant number of shifts and extractions
of the maximum element in its filling chunks $f_1, \ldots, f_s$ and in $l$ itself. We end up
inserting an element into the maniple $m$ of $l$. After that we may have to do a little
work in order to maintain the bucket invariants.

1. If the size of $m$ exceeds the maximum allowed, we extract the $\sqrt{k}$ smallest
   elements from $m$ and insert them into $l$.

2. If the size of $l$ is less than $16k$, we are done.

3. On the contrary, if the size of $l$ also exceeds the maximum allowed but the
   number of its filling chunks is still less than $4r$, we extract the smallest chunk
   of $l$ and create a new filling chunk $f_{s+1}$.

4. Instead, if the number of filling chunks is the maximum allowed, $4r$, we “split”
   the whole group made up of the leaf $l$, its maniple $z$ and its filling chunks.
That is to say, we partition all the elements so that we have two new groups of the same kind, each group member satisfying all the invariants with their values half on the way between the maximum and the minimum allowed. We also generate a median (routing) chunk that have to be inserted in the parent of \( l \), encoding a pointer in that chunk to the new leaf.

5. We then examine all the ancestors of \( l \), except the root, splitting every ancestor that exceeds its maximum allowed weight, obtaining two nodes of roughly the same weight.

Deleting an element is analogous, except that we merge two internal nodes, although we may split once after a merge when the resulting node is too big. For the leaves we need merging and borrowing with an individual element. Merging and splitting the root of a bucket fall inside the control of a mechanism for the synchronization between layer \( \mathcal{D} \) and layer \( \mathcal{B} \), described in Section 15.5.

15.4.2 Memory layout

We now discuss how to store the buckets in a contiguous portion of memory, which is divided into three areas.

- The filling area stores all filling chunks of layer \( \mathcal{B} \) and the routing chunks of the internal nodes of the buckets.

- The leaf area stores all the leaves of the buckets using a new variation of the technique of compactor zones [Franceschini and Grossi, 2003b] (see Chapter 13) that is suitable for de-amortization.

- The manipulate area stores all the maniples using a set of compactor lists.

15.4.2.1 Filling area

We use the filling chunks to allocate the internal nodes. We need here to make some clear remarks on what we mean by “allocate.” Suppose we want to allocate an empty node \( v \) with \( 16k \) chunks. We take a segment of \( 16k \) filling chunks that are contiguous and devote them to \( v \). Since each filling chunk can be placed everywhere in the memory, when we need to insert a routing chunk \( c \) into \( v \), we can replace the leftmost available filling chunk in \( v \) with \( c \), moving that filling chunk elsewhere at the cost of searching one of its elements and of re-encoding the pointer to it, with \( O(\log n) \) time and \( O(k/B) \) block transfers.

Keeping the above remark in mind, we logically divide the filling zone into segments of \( 16k \) filling chunks each, since we can have a maximum of \( 16k \) routing chunks for an internal node. A segment is considered “free memory” if it contains only filling chunks. An internal node \( v \) with \( t \) routing chunks is stored in a segment
with the first \( t \) routing chunks permuted (as we still need the internal directory to search in \( v \)) and the remaining \( 16k-t \) filling chunks. When a routing chunk needs to be inserted into an internal node \( v \) whose weight is not maximal, we put the chunk in place of a filling chunk in the segment assigned to \( v \). The replaced filling chunk will find a new place in

- either the segment of the child \( u \) of \( v \), if \( u \) is the result of the splitting of an internal node,

- or between the filling area and the leaf area, if \( u \) is the result of the splitting of a leaf (in that case the filling area increases by one chunk).

The deletion of a routing chunk in \( v \) is analogous. We replace the chunk with a filling chunk that arises either from the two merged children of \( v \), if these children are internal nodes, or from the last position of the filling area, if these children are leaves (and the filling area decreases by one chunk). Thus, using the directory for the routing as described above, we are able to insert or delete a chunk in an internal node in \( O(\log n) \) time and \( O(k/B) \) block transfers.

When we have to split an internal node \( v \) in two nodes \( v', v'' \), we allocate a new segment \( a \) for \( v'' \) while re-using the segment of \( v \) for \( v' \), and exchange incrementally the routing chunks in the segment of \( v' \) with filling chunks of \( a \), the segment for \( v'' \). We exchange a constant number of chunks at each step, and these \( s = O(k) \) steps are spread through the subsequent \( s \) operations operating through \( v \). Note that, during this transition, \( v \) is considered not split but only partitioned in two segments instead of one. The execution of a merge is analogous. The invariants defined on the buckets guarantee that we can terminate an incremental transfer before that a further split or merge occurs.

The management of segments is through a simple linked list of free segments. The constant \( r \) that bounds the minimum number of filling chunks associated with a leaf can be easily chosen so that we can guarantee that there exists a sufficient number of filling chunks in layer \( B \) for all internal nodes.

### 15.4.2.2 Leaf area

The size of the leaves ranges from \( k \) to \( 16k \) elements, and vary by \( \sqrt{k} \) elements at a time. Using the technique of the compactor zones, we maintain \( 15\sqrt{k}+1 \) zones of contiguous memory, one for each possible size. Each zone is indexed by the size of the leaves it contains. The zones are in order by this index, so that zone \( s \) precedes zone \( s + \sqrt{k} \), for each \( s = k, k + \sqrt{k}, k + 2\sqrt{k}, \ldots, 16k - \sqrt{k} \).

When we have to add \( \sqrt{k} \) elements to a leaf \( l \) of size \( s \), we would like to extract \( l \) out of all compactor zones, moving \( l \) near to the \( \sqrt{k} \) elements to be added by rotating each traversed zone by \( s \) elements. As a result, all the leaves are in a contiguous portion of memory except for a single leaf that can be "broken" in two pieces because of the rotation. This scheme is simple and powerful but too costly.

We achieve our worst-case bounds with a two-step modification of this scheme.
First step: lazy rotations. We want to execute the rotations occurring in the compactor zones in a "lazy" way. To this purpose we can exploit the fact that, for each leaf $l$:

1. $\Omega(\sqrt{k})$ update operations occur in its maniple between two consecutive variations of $\sqrt{k}$ in the size of $l$;

2. $\Omega(k)$ update operations occur in its maniple between two consecutive variations of $k$ in the size of $l$ (due to the creation/destruction of a filling chunk);

3. $\Omega(k)$ update operations occur in its filling chunks and its maniple between two consecutive splits or merges of $l$.

Consequently, we have a sufficient number of operations to perform incrementally the updates involving a leaf $l$. The basic idea is to execute a constant number of rotations from zone to zone in a single operation.

Second step: commuting sub-zones. The second step introduces two commuting sub-zones between any two compactor zones. These two sub-zones work like the compactor zones but contain blocks of elements in transit between zones (see Figure 15.2). For any pair of sub-zones, the first sub-zone contains the blocks of $k + \sqrt{k}$ elements that have to be inserted in or deleted from a leaf. The second sub-zone contains

- chunks that have to be inserted or deleted in a leaf;

- all the chunks of the leaves to be split or merged.

For example, when a leaf reaches its maximum number of elements, it is transformed into a linked list of $O(1)$ chunks going to the second sub-zone near zone $16k$. At this point, we incrementally move these chunks until we reach the sub-zone near zone $8k$; we split the list into two parts and put them as two new leaves of size $8k$. Note that the leaf is still searchable while traversing the zones.
15.4.2.3 Maniple area

The maniple area is handled with compactor lists [Franceschini, Grossi, Munro, and Pagli, 2002]. However, we use allocation units of size $\sqrt{k}$, and so the structural information for them must be encoded in the leaves associated with the maniples. Each time we need a structural information (e.g., next allocation unit in a list), we perform a search to locate the corresponding leaf. There are $O(\sqrt{k})$ heads of size at most $\sqrt{k}$, so the whole head area occupies $O(k)$ positions and can be scanned each time.

**Theorem 15.2** Searching, inserting and deleting an element in a bucket of layer $\mathcal{D}$ takes $O(\log n)$ time and $O(\log B n)$ block transfers in the worst case for any (unknown) value of $B$.

15.5 Synchronization between Layer $\mathcal{D}$ and Layer $\mathcal{B}$

15.5.1 The problem

We combine the two layers described in Sections 15.3–15.4 by using a simple variation of a technique developed in [Dietz and Sleator, 1987]. Basically, every other $\Omega(\text{polylog}(n))$ operations in layer $\mathcal{B}$, we eventually split the largest bucket and we merge the smallest bucket. This causes the insertion and the deletion of a routing chunk in layer $\mathcal{D}$. By setting up the suitable multiplicative constants, we provide a time slot that is sufficient to complete the algorithms operating in layer $\mathcal{D}$.

More in detail. If we let the buckets in layer $\mathcal{B}$ freely to be split or merged whenever the pattern of insertions and deletions requires it, we will have an unconstrained flux of insertions and deletions of routing chunks in layer $\mathcal{D}$. This is obviously bad because, as we saw early in this chapter, we need to execute the update algorithm based on Willard’s scheme [Willard, 1992] in a lazy way, that is we need to spread each execution of the algorithm (each execution corresponding to an insertion or deletion of routing chunk in $\mathcal{D}$) over $s = O(\text{polylog}(n))$ sub-steps using $O(\log n)$ time and $O(\log_B n)$ block transfers each. Those $s$ substeps have to be executed together with the next $s$ update operations that will be requested to the structure after the insertion of the routing chunk in $\mathcal{D}$ that triggered the Willard’s algorithm. It easy to understand how it is necessary that no one of those $s$ update operations could cause the insertion of a routing chunk in $\mathcal{D}$ in its turn, because this would interfere with the concurrent, “lazy” execution of Willard’s algorithm. It is also easy to see how the simple constraint that any bucket in $\mathcal{B}$ splits or merges after $\Omega(\log^d n)$ update operations ending in it is not sufficient for our purpose. We need a stronger constraint involving the whole set of buckets simultaneously.
15.5.2 The solution

The following theorem proved in [Dietz and Sleator, 1987] comes to help us:

**Theorem 15.3 ([Dietz and Sleator, 1987])** Let \( x_1, \ldots, x_m \) be \( m \) real valued variables, all initially zero. Repeatedly perform the following procedure:

1. Find an index \( i, 1 \leq i \leq m, \) such that \( x_i = \max_j \{x_j\} \). Set \( x_j \) to zero.
2. Pick \( m \) non-negative reals \( a_1, \ldots, a_m \) such that \( \sum_{i=1}^{m} a_i = 1 \).
3. For \( i = 1, \ldots, m, \) set \( x_i \) to \( x_i + a_i \).

No \( x_i \) will ever exceed \( H_{m-1} - 1 \), where \( H_t = \sum_{i=1}^{t} i^{-1} \), the \( t \)-th harmonic number.

As we said, in order to correctly update layer \( \mathcal{D} \) we need that between any two consecutive updates of layer \( \mathcal{D} \) (insertions or deletions of routing chunks) there is a sequence of \( s = \Omega(\text{polylog}(n)) \) update operations that terminate without trying to modify \( \mathcal{D} \). To obtain this we change the way the buckets are grown and split or merged. We fix the upper and lower bounds for the size of the buckets in advance but we do not immediately merge or split a bucket violating them instead let the things go forward for a little while and after that we take the two buckets that deviate the most from the upper and lower bounds and adjust them.

More precisely, let \( s = O(\log^d n) \), we fix \( d = t \) (\( d \) is the maximum number of levels that a bucket should have, see Section 15.4.1) and operate in the following way throughout the entire life of the structure.

**Procedure 15.1**

1. We execute the next \( s \) update operations without splitting or merging any bucket in \( \mathcal{B} \).
2. We find the bucket with the largest number of elements. If it violates the upper bound for its weight (see Section 15.4.1), we split it (eventually inserting a routing chunk in layer \( \mathcal{D} \)).
3. We find the bucket with the smallest number of elements. If it violates the lower bounds for its weight (see Section 15.4.1), we merge it (eventually extracting a routing chunk from layer \( \mathcal{D} \)).

As we said, we let things go for a little while (\( s \) update operations) in layer \( \mathcal{B} \) without doing anything about the possible violations of the upper and lower bounds on the number of elements in a bucket. From an operative point of view, with the organization for the buckets we gave in Section 15.4 we are perfectly able to manage the growth of buckets with a constrained, \( \Theta(k) \), branching factor and any height. From the point of view of the complexity analysis, we have to be able to
estimate the heights of the buckets created with this new approach. Obviously, we are particularly concerned about the upper bound of the heights since we cannot afford non-constant heights.

Fortunately Theorem 15.3 can help us: the maximum number of elements in a bucket is \( O(sH_n) = O(s \log n) = O(\log^{t+1} n) \). Since the buckets have branching factor \( O(k) \), their heights are still upper bounded by a constant.

Therefore we have the following:

**Lemma 15.2** If we build layer \( \mathcal{B} \) using Procedure 15.1, we will obtain buckets whose heights are still upper bounded by a constant.

### 15.5.3 Priority queues

For what concerns the execution of steps 2 and 3, we need to encode two implicit priority queues in order to execute those steps efficiently. Let us consider the priority queue for the maximum weight.

**The lists.** We want to maintain \( O(\log(n)) \) linked lists \( L_1, L_2 \ldots \) where each unit in a list contains a back-pointer to a corresponding bucket. There is one list for each possible weight that a bucket can have and, obviously, buckets with the same weight \( i \) have a unit (with its back-pointer) in the same list \( L_i \).

The lists are linked in their turn in a linked list \( \mathcal{L} \) where they appear in ascending order with respect to the weight they are associated with. Each unit in \( \mathcal{L} \) contains a pointer to the list \( L' \) it represents and a pointer to the last unit of \( L' \). Of course, there can be empty lists but they are always represented in \( \mathcal{L} \). Each unit in any list \( L' \) contains also a pointer to the unit in \( \mathcal{L} \) representing \( L' \).

**The bit tree.** Finally, we associate a complete tree \( T \) with \( \mathcal{L} \). Each node of \( T \) has \( r = O(\log n) \) pointers and a bit-mask of \( O(\log n) \) bits. The units in \( \mathcal{L} \) are grouped in sequences of \( r \) consecutive (in \( \mathcal{L} \)) units. In \( T \) there is a leaf for each group of lists:

- the \( i \)th pointer of the leaf points to the unit in \( \mathcal{L} \) representing the \( i \)th list of the group and
- the \( i \)th bit of the bit-mask is set to 1 if and only if the \( i \)th list of the group is not void.

The leaves of \( T \) are grouped in a similar way and they are associated with internal nodes one level higher in \( T \). The only difference is in the definition of the bit-mask: the \( i \)th bit of the bit-mask of an internal node \( v \) is set to 1 if and only if there is a bit set to 1 in the bit-mask of the \( i \)th child of \( v \). The definition of \( T \) goes on like this until the root is reached. It is obvious that \( T \) has \( O(1) \) levels. Let us see how the lists are used.
**Extractions and insertions.** When we have to extract the bucket with the maximum weight from the lists, we use the tree $T$ in the following way.

1. We start from the root of $T$ and find the rightmost $1$ in the bit-mask of the root, let it be the $j$th bit.

2. Then we proceed recursively in the same way with the $j$th child of the root. At the end of this process we have found the rightmost, non-void list $L_i$ (the one with the largest index $i$).

3. We extract the first unit of $L_i$ that points to the bucket with maximum weight.

4. If $L_i$ had only one unit, we have to update the bit mask of the leaf of $T$ pointing to the unit of $L_i$ in $\mathcal{L}$. That can trigger a series of symmetric updates until the root is reached.

   When a new element is inserted, the bucket receiving it removes itself from $L_i$ and insert itself in $L_{i+1}$ in the obvious way, accessing and modifying $O(1)$ pointers. If $L_i$ had only one unit, an update of $T$ analogous to the one we have seen in the extraction procedure is triggered. The same happens in case $L_{i+1}$ was void.

   The lists and the tree $T$ are conceptually composed of $O(n/\text{polylog}(n))$ values representable with $O(\log(n))$ bits each. Therefore they can be entirely encoded by bit stealing in the obvious way. Moreover, it is easy to see how both the extraction and the insertion accesses and modify $O(1)$ values of $O(\log n)$ bits each. Therefore the operations on the priority queue are performed in $O(\log n)$ time and with $O(\log n/B) = O(\log_B n)$ block transfers.

   Therefore we can conclude that:

**Lemma 15.3** Steps 2 and 3 can be executed in $O(\log n)$ time and with $O(\log_B n)$ blocks transfers in the worst case.

By Lemmas 15.2 and 15.3 we have that:

**Theorem 15.4** Searching, inserting and deleting an element in a bucket of layer $B$ built with Procedure 15.1 takes $O(\log n)$ time and $O(\log_B n)$ block transfers in the worst case for any (unknown) value of $B$.

### 15.6 Conclusion

By Theorems 15.1 and 15.4 we can now state the main result of this chapter. This result answers a very old, fundamental question dating back to the sixties when the first implicit data structure, the heap of Williams (see [Williams, 1964]), was introduced. Is there a searchable heap?
Theorem 15.5 There exists an implicit dictionary, the Searchable Heap, with the following optimal bounds:

- In the RAM model, the dictionary can be searched and updated in $O(\log n)$ time in the worst case.

- In the Cache-Oblivious model, the dictionary can be searched and updated with a work complexity $O(\log n)$ and a cache complexity $O(\log_B n)$ both in the worst case.
Chapter 16

Conclusions

The main objective of this work was to investigate the computational power of permutations or, more precisely, of models having the permutations of the input set as unique exploitable resource to solve a problem. In order to do so, we focused on some well-known open problems in the field of space efficient computations.

We started concentrating on sorting-related problems and considering the classical RAM model at first. We showed how a set of elements drawn from a generic universe can be sorted using the asymptotically optimal number of comparisons, data moves and auxiliary space (see Chapter 5, [Franceschini and Geffert, 2003, 2005]). A result of that kind has been believed to be very unlikely for a long time.

After that, we proved that the same result can be accomplished respecting the additional constraint of stability in case the elements in input form a multiset, that is preserving the original relative order of the occurrences of the equal elements in the input sequence (see Chapter 6, [Franceschini, 2005a,c]). It is well known that achieving the stability is a very difficult objective in settings where space optimality is a goal without losing the optimality with respect to the other measures of complexity (any in-place stable merging algorithm can be taken as a witness of that).

Then, we departed for a little while from the classical RAM model and concentrated our efforts on the Cache-Oblivious model. In that recent model data locality seems to be the key toward optimal solution for any problem. At first, it would seem that locality and space optimality could not be achieved simultaneously very easily, since some of the powerful techniques used in the past for space optimality seem to cause a lot of unpredictable separation among groups of elements that should be maintained adjacent when data locality is a concern. In this setting we proved the existence of a sorting algorithm that is work, cache and space optimal simultaneously (see Chapter 7, [Franceschini, 2004]).

After the Cache-Oblivious model, we went back to the RAM model or, more precisely, a natural extension of it well suited to study space optimality when the input elements are drawn from multidimensional domains (e.g. sets of vectors). In this setting we showed the first sorting algorithm for vectors and records achieving
the asymptotical optimality for scalar comparisons, moves and space.

Finally, we concluded our efforts in the field of sorting-related problems with the generalized merging problem. In this problem $s$ sorted sequences have to be fused into one. Space, comparison and move optimal solutions were known only for the case $s = O(1)$. We showed how this can be achieved for any value of $s$ (see Chapter 9, [Franceschini, 2005b]).

After the sorting-related problems, we focused on searching-related problems. We started with the space optimal searching of a set of $k$-dimensional vectors. If the input elements are arranged in lexicographically sorted order, we know from an important previous paper that upper and lower bounds for the time complexity match. Surprisingly enough, the matching bounds are not equal to the natural bound of $O(\log n + k)$ that can be seen as a generalization of the well-known optimal bound of $O(\log n)$ comparisons for searching a set of unidimensional elements laid out in sorted order. We showed that, for any set of vectors, there exists a permutation of them that can be searched in $O(\log n + k)$ time (see Chapter 10, [Franceschini and Grossi, 2004a]). Hence, we proved, somewhat surprisingly, that the lexicographical order is not the optimal arrangement for space optimal searching of vectors.

After the static multidimensional searching problem, we went back to unidimensional domains and faced a dynamic problem: the implicit dictionary problem. In this problem a dynamic set of elements has to be maintained permuted (without any other resource other than the locations for the permutation itself and $O(1)$ locations for auxiliary numerical values) so that space optimal search, insertion and deletion operations are supported with optimal time bounds. First, we posed ourselves into the External-Memory model and showed that an optimal solution for the implicit dictionary problem exists under a reasonable condition over one of the parameters of the model (see Chapter 11, [Franceschini, Grossi, Munro, and Pagli, 2002, 2004]). Moreover, we showed how the same solution can be adapted for the RAM model in order to improve the best bounds known for the problem disproving a long-standing conjecture.

After the External-Memory model, we focused on the RAM model again. We improved significantly the bound for the search operation in the worst case and for the update operations in amortized sense (see Chapter 12, [Franceschini and Grossi, 2003a]). In order to do so we had to develop new techniques capable to eliminate the intrinsic clashes between the amortized and implicit settings.

Then, we kept moving in the direction of amortized analysis, but considering once again the more general Cache-Oblivious model. The result of these efforts was the first implicit dictionary for the Cache-Oblivious model (and hence also for the RAM and External-Memory model) with optimal worst case bound for the search operation and optimal amortized bounds for the update operations (see Chapter 13, [Franceschini and Grossi, 2003b]).

Subsequently, we disproved a very old conjecture advanced in the 1988 result
[Borodin, Fich, Meyer auf der Heide, Upfal, and Wigderson, 1988] about implicit dictionaries. We showed that there exist implicit dictionaries supporting searches in polylogarithmic time and updates with a constant number of data moves in the worst case (see Chapter 14, [Franceschini and Munro, 2006]).

Finally, we managed to solve definitely the implicit dictionary problem for all the three models considered, giving a new structure achieving the worst case optimal bounds also for insertion and deletion operations (see Chapter 15, [Franceschini and Grossi, 2003c]). With this last result we also answered to a very old question left open since the introduction of the first implicit data structure, the well-known Williams’ heap [Williams, 1964]. We have been able to give a positive answer: there exists a structure having the same characteristics of Williams’ Heap (though much more complicated) that is optimally searchable.
Bibliography


Lars Arge and Jeffrey Scott Vitter. Optimal dynamic interval management in external memory (extended abstract). In *37th Annual Symposium on Foundations
of Computer Science, pages 560–569, Burlington, Vermont, 14–16 October 1996. IEEE.


Gianni Franceschini. Sorting Stably, In-Place, with $O(n \log n)$ Comparisons and $O(n)$ Moves. In *Proceedings of the 22nd Symposium on Theoretical Aspects of Computer Science (STACS)*, 2005a.

Gianni Franceschini. Sorting by merging or merging by sorting? Submitted for publication, 2005b.

Gianni Franceschini. Sorting Stably, In-Place, with $O(n \log n)$ Comparisons and $O(n)$ Moves. Submitted to journal, 2005c.


Gianni Franceschini and J. Ian Munro. Implicit dictionaries with \(O(1)\) modifica-

Gianni Franceschini, Roberto Grossi, J. Ian Munro, and Linda Pagli. Implicit B-

Gianni Franceschini, Roberto Grossi, J. Ian Munro, and Linda Pagli. Implicit B-
trees: A new data structure for the dictionary problem. Journal of Computer and


G. N. Frederickson. Recursively rotated orders and implicit data structures: a lower

Greg N. Frederickson. Implicit data structures for the weighted dictionary problem.
In 22nd Annual Symposium on Foundations of Computer Science, pages 133–139,
Nashville, Tennessee, 28–30 October 1981. IEEE.

Greg N. Frederickson. Implicit data structures for the dictionary problem. Journal

Greg N. Frederickson. Self-organizing heuristics for implicit data structures. SIAM
Journal on Computing, 13(2):277–291, 1984b. ISSN 0097-5397 (print), 1095-7111
(electronic).

Greg N. Frederickson. Implicit data structures for weighted elements. Information

Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic

Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table
with \(O(1)\) worst case access time. J. ACM, 31(3):538–544, 1984. ISSN 0004-5411.

M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblative algo-
rithms. In IEEE, editor, 40th Annual Symposium on Foundations of Computer
Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1999. IEEE Computer
Society Press. ISBN 0-7695-0409-4 (softbound), 0-7803-5955-0 (casebound), 0-
7695-0411-6 (microfiche).


J. Ian Munro and Hendra Suwanda. Implicit data structures (preliminary draft). In ACM, editor, *Conference record of the eleventh annual ACM Symposium on


M. H. Overmars. The design of dynamic data structures. Lecture Notes in Computer Science, 156, 1983.


