# Data Structures: Sequence Problems, Range Queries, and Fault Tolerance

Allan Grønlund Jørgensen

PhD Dissertation


Department of Computer Science
Aarhus University
Denmark

# Data Structures: Sequence Problems, Range Queries and Fault Tolerance

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Allan Grønlund Jørgensen
July 26, 2010

# Abstract

The focus of this dissertation is on algorithms, in particular data structures that give provably efficient solutions for sequence analysis problems, range queries, and fault tolerant computing. The work presented in this dissertation is divided into three parts.

In Part I we consider algorithms for a range of sequence analysis problems that have risen from applications in pattern matching, bioinformatics, and data mining. On a high level, each problem is defined by a function and some constraints and the job at hand is to locate subsequences that score high with this function and are not invalidated by the constraints. Many variants and similar problems have been proposed leading to several different approaches and algorithms. We consider problems where the function is the sum of the elements in the sequence and the constraints only bound the length of the subsequences considered. We give optimal algorithms for several variants of the problem based on a simple idea and classic algorithms and data structures.

In Part II we consider range query data structures. This a category of problems where the task is to preprocess an input sequence using as little time and space as possible such that one can efficiently compute a certain function on the elements in a given query subsequence. There are many types of functions that has been considered in connection with input from different sources. The input could be ip-data sorted by ip-address, real estate prices sorted by zip code, advertising cost sorted by time etc. We consider data structures for two classic statistics functions, namely median and mode.

Finally, Part III investigates fault tolerant algorithms and data structures. This deals with the trend of avoiding elaborate error checking and correction circuitry that would impose non-negligible costs in terms of hardware performance and money in the design of todays high speed memory technologies. Hardware, power failures, and environmental conditions such as cosmic rays and alpha particles can all alter the memory in unpredictable ways. In applications where large memory capacities are needed at low cost, it makes sense to assume that the algorithms themselves are in charge for dealing with memory faults. We investigate searching, sorting and counting algorithms and data structures that provably returns sensible information in spite of memory corruptions.

# Acknowledgements

My four years of ph.d. study is at an end. It is time to thank all the people who has helped me through the years. First, I would like to thank my advisor Gerth S. Brodal for all the time and help. Furthermore, I would like to thank Madalgo center leader Lars Arge for the opportunity. Additionally, I would like to thank my co-authors Gerth S. Brodal, Thomas Mølhave, Rolf Fagerberg, Irene Finocchi, Giuseppe F. Italiano, Fabrizio Grandoni, Gabriel Moruz, Maarten Löffler, Jeff Phillips, Mark Greve, Kasper D. Larsen and Jakob Truelsen.

*Allan Grønlund Jørgensen,*
*Århus, July 26, 2010.*

# Contents

**Abstract** **v**

**Acknowledgements** **vii**

**1 Introduction** **1**
1.1 Selecting and Reporting Sums from Sequences . . . . . 2
1.1.1 Reporting the $k$ largest sums . . . . . 4
1.1.2 Selecting the $k$'th largest sum . . . . . 5
1.2 Range Queries . . . . . 7
1.2.1 Range Median . . . . . 7
1.2.2 Range Mode . . . . . 11
1.3 Fault Tolerance . . . . . 14
1.3.1 Priority Queues Resilient to Memory Faults . . . . . 16
1.3.2 Searching in the Presence of Memory Corruptions . . . . 17
1.3.3 Fault Tolerant External Memory Algorithms . . . . . 19
1.3.4 Counting in a Hostile Environment . . . . . 20

**I Sums** **23**

**2 The $k$ Maximal Sums Problem** **25**
2.1 Basic Idea and Algorithm . . . . . 26
2.2 Binary Heaps . . . . . 28
2.3 Partial Persistence and $H^j_{\text{suf}}$ Construction . . . . . 28
2.4 Fredericksons Heap Selection Algorithm . . . . . 30
2.5 Combining the Ideas . . . . . 30
2.6 Extension to Higher Dimensions . . . . . 31
2.7 Space Reduction . . . . . 32

**3 Selecting Sums in Arrays** **35**
3.1 The Length Constrained $k$ Maximal Sums Problem . . . . . 36
3.1.1 A Linear Time Algorithm . . . . . 36
3.2 Sum Selection Problem . . . . . 39
3.2.1 The Bheap . . . . . 39
3.2.2 An $O(n \log(k/n))$ Time Algorithm . . . . . 41
3.2.3 Selection in Weight Balanced Trees . . . . . 44
3.2.4 Lower Bound . . . . . 46

3.3 Length Constrained Sum Selection . . . . . 47

**II Range Queries** **49**

**4 Data Structures for Range Median Queries** **51**
4.1 Simple Range Selection Data Structure . . . . . 52
4.1.1 Basic Structure . . . . . 52
4.1.2 Getting Linear Space . . . . . 53
4.2 Improving Query Time . . . . . 54
4.2.1 Structure . . . . . 54
4.2.2 Range Selection Query . . . . . 55
4.2.3 Getting Linear Space . . . . . 58
4.2.4 Construction in $O(n \log n)$ Time . . . . . 60
4.3 Dynamic Range Selection . . . . . 61
4.3.1 Structure . . . . . 61
4.3.2 Range Selection Query . . . . . 62
4.3.3 Updates . . . . . 63
4.4 Lower Bound for Dynamic Data Structures . . . . . 64
4.5 Main Open Problems . . . . . 64

**5 Cell Probe Lower Bounds and Approximations for Range Mode** **65**
5.1 Cell Probe Lower Bound for Range Mode . . . . . 67
5.2 Range $k$-Frequency . . . . . 70
5.3 3-Approximate Range Mode . . . . . 72
5.4 $(1 + \varepsilon)$-Approximate Range Mode . . . . . 73

**III Fault Tolerant Algorithms** **77**

**6 Priority Queue Resilient to Memory Faults** **79**
6.1 Fault Tolerant Priority Queue . . . . . 80
6.1.1 Structure . . . . . 80
6.1.2 Push and Pull Primitives . . . . . 81
6.1.3 Insert and Deletemin . . . . . 82
6.2 Analysis . . . . . 83
6.2.1 Correctness . . . . . 83
6.2.2 Complexity . . . . . 84
6.3 Lower bound . . . . . 86

**7 Optimal Resilient Dictionaries** **89**
7.1 Optimal Randomized Static Dictionary . . . . . 90
7.2 Optimal Static Dictionary . . . . . 91
7.3 Dynamic Dictionary . . . . . 96

**8 Fault Tolerant External Memory Algorithms** **101**
8.1 Lower Bound for Dictionaries . . . 103
8.2 Randomized Static Dictionary . . . 104
8.3 Optimal Deterministic Static Dictionary . . . 106
8.4 Dynamic Dictionaries . . . 109
8.4.1 Structure . . . 109
8.4.2 Operations . . . 110
8.5 Sorting . . . 111
8.5.1 Multi-way Merging . . . 111
8.5.2 Sorting . . . 112
8.6 Priority Queue . . . 112
8.6.1 Operations on $\mathcal{L}$ . . . 113
8.6.2 Operations on Internal Buffers . . . 117

**9 Counting in Unreliable Memory** **119**
9.1 Lower Bounds and Tradeoffs . . . 120
9.2 Data Structures . . . 121
9.2.1 Replicating Bits . . . 121
9.2.2 Round-Robin Counting . . . 122
9.2.3 Counting by Scanning Bits . . . 123
9.2.4 Using Randomization to Obtain Fast Increments . . . 125
9.3 Open Problems . . . 127

**Bibliography** **129**

# Chapter 1

## Introduction

The focus of this dissertation is on algorithms, in particular data structures that give provably efficient solutions for sequence analysis problems, range queries, and fault tolerant computing. It is based on eight papers, seven are already published and the last paper is in the submission stage. The papers are naturally divided into three groups which defines the structure of the dissertation.

Part I considers sequence analysis problems and is based on the papers

**[D2]** *A Linear Time Algorithm for the k Maximum Sums Problem* with Gerth Stølting Brodal. Proceedings of the 32nd International Symposium on Mathematical Foundations of Computer Science, 2007.

**[D3]** *Selecting Sums In Arrays* with Gerth Stølting Brodal. Proceedings of the 19th International Symposium on Algorithms and Computation, 2009.

Compared to the published version of [D3] we have added Section 3.2.3 and Section 3.2.1.

Part II investigates range query data structures and is formed by the papers

**[D4]** *Data Structures for Range Median Queries* with Gerth Stølting Brodal. Proceedings of the 20th International Symposium on Algorithms and Computation, 2009.

**[D7]** *Approximating the Mode and Determining Labels with Fixed Frequency* with Mark Greve, Kasper Dalgaard Larsen and Jakob Truelsen. Manuscript.

Most of Section 4.3 and Section 4.2.4 has been added when compared to [D4]. The added part was in the appendix of the paper when it was submitted. The paper [D4] has been merged with [47] for a journal submission and parts added in this dissertation is also included in this merged paper.

Finally, Part III considers algorithms and data structures that are tolerant to memory faults and it is based on the papers

**[D8]** *Priority Queues Resilient To Memory Faults* with Gabriel Moruz and Thomas Mølhave. Proceedings of the 10th Workshop on Algorithms and Data Structures, 2007.

**[D1]** *Optimal Resilient Dictionaries* with Gerth Stølting Brodal, Rolf Fagerberg, Gabriel Moruz, and Thomas Mølhave. Technical Report DAIMI PB-585, 2007.

**[D5]** *Fault Tolerant External Memory Algorithms* with Gerth Stølting Brodal and Thomas Mølhave. Proceedings of the 11th Workshop on Algorithms and Data Structures, 2009.

**[D6]** *Counting in the Presence of Memory Faults* with Gerth Stølting Brodal, Gabriel Moruz, and Thomas Mølhave. Proceedings of the 20th International Symposium on Algorithms and Computation, 2009.

The paper [D1] is a technical report which was merged with a paper by Irene Finocchi, Fabrizio Grandoni and Giuseppe F. Italiano into the paper [C1] presented at the 15th Annual European Symposium on Algorithms in 2007. Compared to our paper [D5] we have added Section 8.6.

In this chapter we provide a thorough overview and description of the problems that we have considered in this dissertation. We describe the origin of each problem and the results we achieve. We also give a description of the ideas and techniques we used and how we apply them and to obtain the result. This description is informal and we try to give as much insight as we can without getting caught in technical issues. These can of course be found later in the dissertation where all the details are included.

## 1.1 Selecting and Reporting Sums from Sequences

In Part I we consider sequence analysis problems that have risen from applications in pattern matching, bioinformatics, and data mining. On a high level, each problem is defined by a function and some constraints and the job at hand is to locate subsequences that score high with this function and are not invalidated by the constraints. Many variants and similar problems have been proposed leading to several different approaches and algorithms. We consider problems where the function is the sum of the elements in the sequence and the constraints bound the length of the subsequences considered.

In an array, $A[1,\ldots,n]$, of numbers each subarray, $A[i,\ldots,j]$ for $1 \leq i \leq j \leq n$, of $A$ defines a sum, $\sum_{t=i}^{j} A[t]$. There are $\binom{n}{2} + n$ different subarrays each inducing a sum. The task at hand is to locate a subarray $A[i,\ldots,j]$ of $A$ maximizing the sum of its elements, $\sum_{t=i}^{j} A[t]$. The problem is known as the *maximal sum* problem and was formulated in a pattern matching context by Ulf Grenander, and has been considered by applications in Data Mining [43] and Bioinformatics [3].

Bentley describes the problem and an elegant optimal linear time algorithm credited to Jay Kadane in [18]. The algorithm is based on the following insight: The largest sum in $A[1,\ldots,t+1]$ is either the largest sum in $A[1,\ldots,t]$ or it is the largest sum that ends at index $t+1$. The algorithm scans the input array from left to right and maintains the best solution, $\max_{1\leq i\leq j\leq x} \sum_{t=i}^{j} A[t]$,

| Problem | Time Bound |
|---|---|
| $k$ Max Sums | $O(n+k)$ |
| Length Constrained $k$ Max Sums | $O(n+k)$ |
| Sum Selection | $\Theta(n\log(k/n))$ |
| Length Constrained Sum Selection | $O(n\log(k/n))$ |

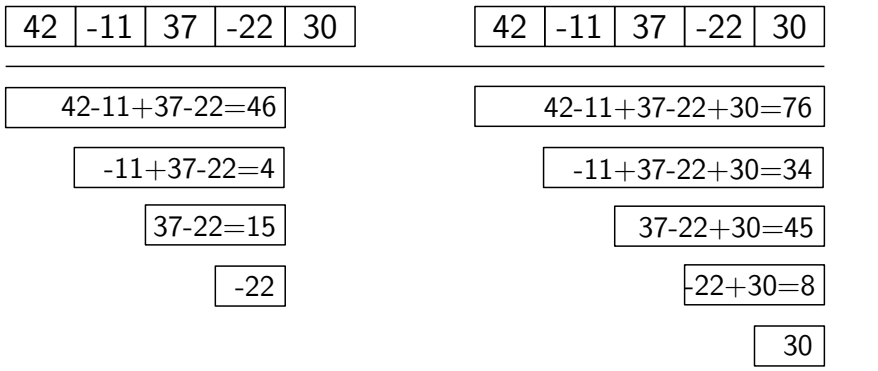
Table 1.1: Results we achieve for selecting and reporting sums in sequences.

and the best suffix solution, $\max_{1\leq i\leq x}\sum_{t=i}^{x}A[t]$, in the part of the input array, $A[1,\ldots,x]$, scanned so far. Both values are updated in constant time in each step yielding a linear time algorithm using $O(1)$ additional space.

The maximal sum problem has been considered in higher dimensions as well. In two dimensions the input is an $m\times n$ matrix of numbers, and the task is to locate the connected submatrix storing the largest aggregate. This problem can be solved by a reduction to $\binom{m}{2}+m$ one-dimensional instances of size $n$, but the resulting algorithm is not optimal [87, 88]. As far as we know the fastest known algorithm uses $O(m^2n\sqrt{\log\log m/\log m})$ time [87]. Interestingly, the best known lower bound is the trivial bound of $\Omega(mn)$ (the time needed to read input) leaving a significant gap. The two-dimensional version was the first problem studied, introduced as a method for maximum likelihood estimations of patterns in digitized images [18].

A natural extension of the maximum sum problem is to compute the $k$ largest sums for $1\leq k\leq\binom{n}{2}+n$. The subarrays are allowed to overlap and the sums that are output are not required to be sorted. This extended problem was introduced in [12] and further investigated in [13, 14, 17, 29, 67]. We refer to this problem as the *$k$ maximal sums* problem. Another generalization of the maximal sum problem is to restrict the length of the subarrays considered. This generalization is considered in [35, 55, 69] mainly motivated by applications in Bioinformatics such as finding tandem repeats [93], locating GC-rich regions [49], and constructing low complexity filters for sequence database search [5]. This naturally leads us to the *length constrained $k$ maximal sums* problem. Given the input array $A$ and integers $l$ and $u$, return the $k$ largest sums consisting of at least $l$ and at most $u$ numbers (length of subarray between $l$ and $u$). Of course, the $k$ maximal sums problem is the special case of this problem where $l=1$ and $u=n$. This problem is considered in [68]. In [67, 68] the authors generalize the maximal sum problem to *sum selection*, e.g. return a $k$'th largest sum. In [68] the authors consider the *length constrained sum selection* problem, e.g. return a $k$'th largest sum among all sums of length at least $l$ and at most $u$.

**Our Results:** In [D2, D3] we give algorithms for all of these problems based on the same techniques. We have shown the results we achieve in Table 1.1. Our algorithms are based on the following idea: Assume that we know the order of all the sums ending at index $t$ and that we want to compute the order of all the sums that end at index $t+1$. If we ignore the sum that consists only of $A[t+1]$ then the sums ending at index $t+1$ are ordered exactly like the




$$L_4 = \{-22, 4, 15, 46\} \qquad L_5 = \{8, 34, 30, 45, 76\}$$
$$L_4 \cup \{0\} = \{-22, 4, 0, 15, 46\} \qquad = \{x + 30 \mid x \in L_4 \cup \{0\}\}$$

Figure 1.1: Illustration of the relation between sums ending at succeeding indices. The input array is shown twice at the top. On the left side we have illustrated all sums ending at index 4, and on the right side we have shown all sums ending at index 5. $L_i$ is the sorted version of the sums ending at index $i$. Notice how $L_5$ can be constructed from $L_4$.

sums ending at index $t$, since the only difference is that we add $A[t+1]$ to all sums. Thus, to order the sums ending at index $t+1$ we only need to compute how a single sum relates to the sums ending at index $t$. This is illustrated in Figure 1.1.

This also allows for an efficient representation of the $O(n^2)$ sums in the array since the sums ending at index $t+1$ are the same as the sums ending at index $t$ with a fixed constant added to each, plus one more sum. This is a perfect place to use persistent data structures [33]. In short, a partially persistent data structure allows the newest version of the data structure to be updated (which yields a new version), while supporting queries to the new version and all previous versions of the data structure. In our papers we use the node copying technique to achieve partial persistence [33].

### 1.1.1 Reporting the $k$ largest sums

In [D2], we use this idea and represent the $O(n^2)$ sums using $O(n)$ space in partially persistent heap-ordered binary trees. We use a very simple heap-ordered binary tree which sole feature is that it supports insertions in amortized constant time. For $t = 1, \ldots, n$ we construct a representation for all sums ending at index $t$ from the representation of the sums ending at index $t-1$ in constant time[1]. After this construction, we extract the $k$ largest sums from the $n$ partially persistent heap-ordered binary trees using Fredericksons heap selection algorithm [42] that outputs the $k$ largest elements in a heap-ordered binary tree in $O(k)$ time. As a result we get an optimal $O(n+k)$ time algorithm for the

[1] we add a constant to all numbers in our data structure by writing it in the root of the tree and pushing it downwards during traversal

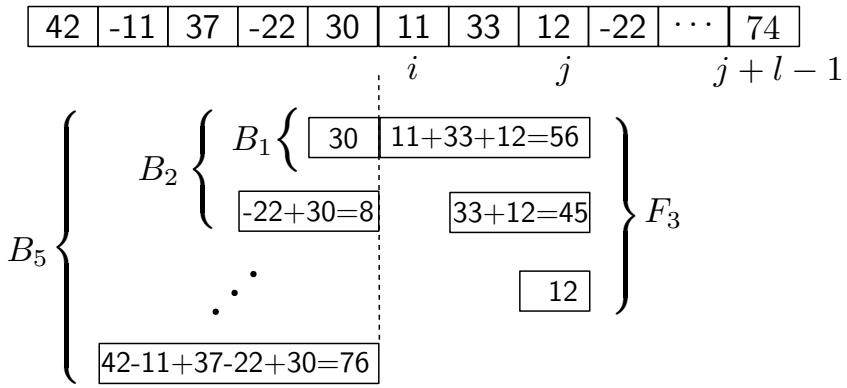
$k$ maximal sums problem that uses $O(n+k)$ additional space. By considering $k$ input elements from the input array at a time, we reduce the additional space usage of our algorithm to $O(k)$, which is needed to store the output. The resulting algorithm becomes a natural generalization of Kadanes algorithm. We also show how to use our algorithm to solve the $d$ dimensional version of the problem where the input sides have size $n_1, \ldots, n_d$ in $O(n_1 \prod_{i=2}^{d} {n_i}^2 + k)$ time using $O(\prod_{i=1}^{d-1} n_i + k)$ additional space. These algorithms are, however, not optimal for all $k$.

If we use a heap-ordered binary tree data structure that also supports deletions efficiently our approach immediately gives an $O(n \log(u-l) + k)$ time algorithm for the length constrained $k$ maximal sums problem: In the $t$'th step of the construction described above we insert the sum of length $l$ ending at index $t$ and delete the sum of length $u+1$ ending at index $t$. After the construction we extract the $k$ largest sums with Fredericksons heap selection algorithm. In [D3] we give an $O(n+k)$ time algorithm for the length constrained $k$ maximal sums problem using the same ideas. The only difference compared to the algorithm described above is how we insert all sums of length between $l$ and $u$ in partially persistent heap-ordered binary trees without performing deletions. We divide the input array into slabs of size $u-l+1$ and for each slab use the same construction as above, and the same construction applied *backwards*: If we have a representation of all sums ending at index $t$ that starts after index $s+1$, we can get a representation of all sums ending at index $t$ starting after index $s$ by adding the single sum $A[s, \ldots, t]$. This idea allows us to *delete* the sum of length $u+1$ in each step of the algorithm by considering the previous version of the representation of these sums. We have illustrated the idea in Figure 1.2. Again, we use partially persistent heap-ordered binary trees to represent the sets of sums and Fredericksons heap selection algorithm to extract the $k$ largest sums.

### 1.1.2 Selecting the $k$'th largest sum

We also consider the sum selection problem in [D3]. With our basic approach we can almost immediately get an $O(n \log n)$ time algorithm for the sum selection problem as follows. Replace the heap-ordered binary trees with weight balanced search trees [10, 72]. Similarly to above, for $t = 1, \ldots, n$, we construct a weight balanced tree storing the sums ending at index $t$ from the data structure for $t-1$ in $O(\log n)$ time. Then, we input the $n$ trees to the sorted column matrix selection algorithm of Frederickson and Johnson [42] that selects the $k$'th largest element from a set of $n$ sorted arrays in $O(n \log(k/n)$ time. Adapting Frederickson and Johnsons algorithm to work on the weight balanced trees we construct instead of sorted arrays is a technical exercise and it is described in Chapter 3.

We reduce the time for the sum selection problem to $O(n \log(k/n))$ by carefully extracting $O(k)$ sums to perform the final selection on. We consider the input array in consecutive slabs of $k/n$ elements. From this division, we divide the sums into two overall sets, sums where the start and end index are in the same slab, and the rest. We represent the $O(k)$ sums that start and end in




$$S_{j+l-1} = \{c_{j+1,j+l-1+x} \mid x \in F_3\} \cup \{c_{a,j+l-1} + x \mid x \in B_{u-l-2}\}$$

Figure 1.2: Organizing the sums of length between $l$ and $u$. The dashed line depicts the separation of two slabs. $B_a$ is the *a shortest* sums that end at index $i-1$ and the sets are incrementally constructed by scanning backwards. $L_a$ are the $a$ shortest sums that end at index $i+a-1$ constructed as in Figure 1.1. The constants $c_{a,b} = \sum_{t=a}^{b} A[t]$ are the sum of the elements between index $a$ and $b$, for any a,b. $S_{j+l-1}$ are the $u-l+1$ sums of length between $l$ and $u$ that end at index $j+l-1$, and as it is shown, this is the union of two sets. For the following index $(j+l)$ we use two different constants and use the previous version of the $B$ set and the following version of the $F$ set.

the same slab in partially persistent weight balanced trees and these sums are constructed as earlier, e.g. for every slab we start with the empty tree, and for the following $k/n$ indices we construct the sums ending there by inserting an element into this partially persistent weight balanced tree. This takes $O(\log(k/n))$ time per element since there is at most $k/n$ elements in each tree considered. This takes care of all sums that start and end inside the same slab. For the remaining sums, each ending in a different slab in which it starts, we use a partially persistent heap-ordered tree data structure where each node stores a sorted array of $k/n$ elements (we get a partial order of sorted arrays of size $k/n$). This data structure we denote a block heap and it supports the insertion of $k/n$ elements in $O(k/n)$ amortized time. As several times before, the sums in this set ending at index $i$ are constructed from the sums ending at index $i-1$. Consider an index $i$ contained inside a slab. The sums ending at index $i$ that does not start in this slab is the sums ending at index $i-1$ that does not start in this slab with $A[i]$ added to each sum. So getting these from the previous set is simple. Whenever we finish a slab we add the $k/n$ sums from the weight balanced tree we created for the last index of the slab to the set by inserting them into the block heap data structure. In the end we extract the $O(k)$ only sums that can be the $k$'th largest sum from the block heaps using Fredericksons heap selection algorithm. These sums are located in $O(n)$ arrays. Combined with the $O(k)$ sums stored in the weight balanced trees we have all sums that could be the $k$'th largest sum. Next we use Fredericksons and Johnsons sorted

column matrix selection algorithm that we have adapted and retrieve the $k$'th largest sum.

We also prove a matching $\Omega(n \log(k/n))$ lower bound on the number of comparisons between linear combinations of the input elements any algorithm needs to solve the problem by a reduction from the cartesian sum selection problem [42]. Finally, we combine our two algorithms described above and obtain an $O(n \log(k/n))$ time algorithm for the length constrained sum selection problem.

## 1.2 Range Queries

There is a large group of problems where the input is an *unordered* array and the task is to preprocess this array using as little time and space as possible such that one can efficiently compute a certain function on the elements in a given query interval (subrange of the input array). These problems are denoted range query problems, and natural candidates for such function are

- Sum: This problem is easily solved with $O(n)$ preprocessing time and space and constant query time by computing prefix sums, since each query can be answered by subtracting two prefix sums.
- Semigroup operator: This function is harder to compute than the sum, since subtraction is not available. However, there exists a very efficient solution. Using $O(n)$ processing time and space, each query can be answered in $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function [96]. A matching lower bound is known [97].
- Maximum, Minimum: For these particular semigroup operators, the problem can be solved slightly more efficiently: $O(n)$ preprocessing time and space is sufficient to allow constant time queries, see e.g. [44].
- Rank: The problem of finding the number of elements smaller than a query element within a query range. This is a natural problem appearing frequently in data bases and it has been studied extensively. A linear space data structure supporting queries in $O(\log n / \log \log n)$ time is presented in [57], and a matching lower bound in the cell probe model for any data structure using $O(n \log^{O(1)} n)$ space is shown in [75, 76].

Several of these problems have been extended to higher dimensions and have also been considered in dynamic scenarios. In the following we denote the input array by $A$ and we let $A[i, j] = A[i, \ldots, j]$ denote the subarray of $A$ from index $i$ to index $j$, both included. In [D4] we consider the range median problem and in [D7] we consider the range mode problem. The model of computation is the RAM model with a word size of $w \geq \log n$ bits.

### 1.2.1 Range Median

The median of a set $S$ of size $n$ is an element in $S$ that is larger than $\lfloor \frac{n-1}{2} \rfloor$ other elements from $S$ and smaller than $\lceil \frac{n-1}{2} \rceil$ other elements from $S$. In the

| | Query Time | Space | Batched Problem |
|---|---|---|---|
| [62] | $O(\log n)$ | $O(n\frac{\log^2 n}{\log\log n})$ | - |
| [79] | $O(1)$ | $O(n^2\frac{\log^2\log n}{\log^2(n)})$ | - |
| [50] | - | - | $O(n\log k + k\log(k)\log n)$ |
| [47] | $O(\log n)$ | $O(n)$ | $O(n\log k + k\log n)$ |
| Our Results | $O(\frac{\log n}{\log\log n})$ | $O(n)$ | $O(n\log k + k\frac{\log n}{\log\log n})$ |

Table 1.2: Overview of the most important results on range median/selection queries.

range median problem the input array $A$ must be preprocessed into a data structure that given indices $i$ and $j$, $1 \leq i \leq j \leq n$, returns an index $i'$, $i \leq i' \leq j$, such that $A[i']$ is the median of the elements in the subarray $A[i, j]$. In addition to being a natural extension of the median problem, the problem has applications in practice, namely obtaining a *typical* element in a given time series out of a given time interval [50]. The range median problem is considered in [45, 47, 62, 78, 79].

Consider analyzing logs of internet advertisements run by for instance Google for various companies. Every click on a sponsored link by an internet user records a timestamp and the price paid by the advertiser for this click. Advertisers usually runs several add campaigns over different time intervals and wants to compare the prices paid with the overall market cost in their campaign periods. A typical comparison is with the median price paid by all advertisers in this time interval. This boils down to a range median query for each interval in an array storing the price of each click sorted by the corresponding timestamps. With this and other applications in mind Muthukrisnan and Har-Peled defined and considered the batched range median problem in [50]. The input to the batched range median problem is an array of size $n$ and a set of $k$ queries, $(i_1, j_1), \ldots, (i_k, j_k)$, and the task is to output the answer to these $k$ queries. We have shown the most important results for range median queries in Table 1.2. Range median queries are naturally generalized to *range selection*, given indices $i, j$ and $s$, return the index of the $s$'th smallest element in $A[i, j]$. In the following we consider these queries instead since they are more general and fits more naturally with the data structures we develop.

**Our Results** Our main result is a linear space data structure that answers range selection queries in $O(\log n/\log\log n)$ time. Our data structure also supports range rank queries in $O(\log n/\log\log n)$ if we can perform a predecessor search on the elements stored in $A$ in the same time bound, e.g. for instance if the elements stored in $A$ are integers.

The data structure is based on the following observation, used in [47]. Suppose we partition the elements in $A$ into two arrays: one storing the $n/2$ smallest values, and one storing the $n/2$ largest values, the elements ordered in each array by their position in $A$. Denote these two arrays $A_{small}$ and $A_{large}$ respectively. Assume we are searching for the element of rank $s$ in $A[i, j]$. We compute how

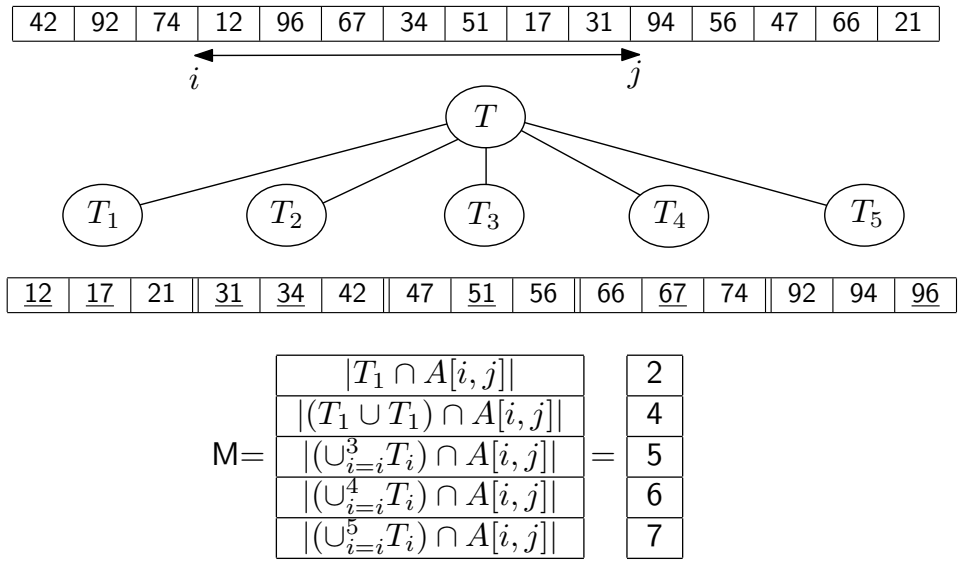



Figure 1.3: An example of the prefix counts we consider. In this example the fanout is 5 and the elements in $A[i, j]$ are underlined in the partition of the elements into 5 subtrees. $M$ is the list of prefix counts for the query $A[i, j]$, e.g the $\ell$'th row stores the number of element from $A[i, j]$ that are contained in the first $\ell$ subtrees. If a query is searching for the fourth smallest element in $A[i, j]$ it is contained in $T_2$ as indicated by the prefix counts.

many elements from $A[i, j]$ there are in $A_{small}$ which is the rank difference between $i - 1$ and $j$ in $A_{small}$ when elements are represented by their position in $A$. Denote this count $c$. If $c \geq s$ then the element of rank $s$ in $A[i, j]$ is the element of rank $s$ in $A_{small} \cap A[i, j]$. Otherwise, it is the element of rank $s - c$ in $A.high \cap A[i, j]$. Thus we have to solve the same problem in an array of half the size. This idea leads naturally to a binary search tree similar in structure to a range tree.

In [D4] we generalize this idea and develop a data structure that uses $O(n)$ words of space and supports queries in $O(\log n / \log \log n)$ time. The idea is to use a tree with branching factor $f = \lceil \log^{\varepsilon} n \rceil$ for some $0 < \varepsilon < 1$, instead of a binary tree. This reduces the depth of the tree to $O(\log n / \log \log n)$.

A query is answered by descending the tree, starting from the root, until we reach the leaf that stores the element of the given rank $s$ in $A[i, j]$. There are two different cases for computing the child to continue the descend: First, an attempt is made to identify the correct child in constant time, by computing, for every $l \in \{1, \ldots, f\}$, an approximation of how many elements of $A[i, j]$ that are contained in the first $l$ subtrees (called *prefix count* in the following). Note that the element of rank $s$ is contained in the leftmost subtree of $T_v$ whose prefix count is at least $s$. Figure 1.3 shows a small example of such prefix counts. To compute all approximate prefix counts in constant time, the approximation only computes $g = O(\log n / f)$ bits of each prefix count. This approximation reduces the branching decision to an interval $[\ell_1, \ell_2]$, and this interval is found

| 103200345 |
|---|
| 109002233 |
| 109122324 |
| 109367210 |
| 109666010 |
| 112111099 |

(a)

| 103 | 200 | 345 |
|---|---|---|
| 109 | 002 | 233 |
| 109 | 122 | 324 |
| 109 | 367 | 210 |
| 109 | 666 | 010 |
| 112 | 111 | 099 |

(b)

Figure 1.4: (a) A set prefix sums (M) as in Figure 1.3 - the numbers are this large for sake of exposition. (b) A division of the prefix sums into three sections (columns). Each section stores the same number of digits from each prefix count and can be stored in one machine word. Our algorithm (approximately) computes such a section in constant time. If a query searches for the element of rank 104400234 we can determine in constant time that the second subtree contains this element just by looking at the first section. However, if a query searches for the element of rank 109221123 we cannot determine in which subtree to descend from the information in the first section. We can only conclude that it is between the second and the sixth subtree.

in constant time using word level parallelism. If the interval contains only a constant number of indices, the branching decision in this node is made in constant time, by exactly computing the prefix count for these indices. If this is not the case the correct child is computed using a binary search (where each step, computes the exact prefix count for a given subtree), which takes $O(\log f) = O(\log \log n)$ time. An example of the approximation we compute is shown in Figure 1.4. The important property that we show is that every time we perform a binary search, the number of relevant bits in all prefix counts that remains to be considered is reduced by $g$ (in the example from Figure 1.4 this means that in the following nodes the first section contains only zeroes and is ignored). Therefore, this second case can only occur $O(\log n/g) = O(f)$ times, and the total time for the search is $O(f \log \log n + \log n/\log \log n) = O(\log n/\log \log n)$.

We use the same idea to design a dynamic data structure for the range median problem. Basically, we combine the standard dynamization techniques of using a weight-balanced tree, where associated data structures are rebuilt from scratch when a rotation occurs, with our static data structure. This way we get an $O(n \log n/\log \log n)$ space data structure that supports queries and updates in $O((\log n/\log \log n)^2)$ time, worst case and amortized respectively. We also prove a lower bound of $\Omega(\log n/\log \log n)$ on the query time for data structures that support updates in $O(\log^{O(1)} n)$ time by a reduction from the marked ancestor problem [4].

Our data structure uses $O(n)$ words of space and supports range selection queries in $O(\log n/\log \log n)$ time and it can be constructed in $O(n \log n)$ time. Combining this with the algorithm for the batched problem in [47], we get an algorithm for the batched range median problem with a running time of $O(n \log k + k \log n/\log \log n)$.

| Problem | Query Time | Space |
|---|---|---|
| Range Mode | $\Omega(\frac{\log n}{\log(Sw/n)})$ | $S$ |
| 3-Approximate Range Mode | $O(1)$ | $O(n)$ |
| $(1+\varepsilon)$-Approximate Range Mode | $O(\log 1/\varepsilon)$ | $O(n/\varepsilon)$ |
| Range 1-Frequency | $O(\log^2 \log n)$ | $O(n \log n)$ |
| Range $k$-Frequency | $O(\log n/ \log \log n)$ | $O(n)$ |
| Range $k$-Frequency | $\Omega(\frac{\log n}{\log(Sw/n)})$ | $S$ |

Table 1.3: Results we achieved for range mode and related problems. For range $k$-frequency the upper bound works for any $k$ while the lower bound holds for any constant $k \geq 2$.

### 1.2.2 Range Mode

The frequency of a label $l$ in a multiset $S$ of labels, is the number of occurrences of $l$ in $S$. The mode of $S$ is the most frequent label in $S$. In case of ties, any of the most frequent labels in $S$ can be designated the mode.

The range mode problem has been considered in [62, 78, 79]. We let $M_{i,j}$ denote the mode in the subarray $A[i, j]$, and let $F_{i,j}$ denote the frequency of $M_{i,j}$ in $A[i, j]$. The approximation variant of the range mode problem, we denote the $c$-approximate range mode problem. In a $c$-approximate range mode query in $A[i, j]$ we are searching for a label that has a frequency that is at most a factor $c$ from $F_{i,j}$. We also define the range $k$-frequency problem. In a range $k$-frequency query in $A[i, j]$ we want to determine whether there is a label occurring precisely $k$ times in $A[i, j]$.

There are two results on range mode queries neither very satisfying. For any constant $\varepsilon \leq \frac{1}{2}$, there is a data structure that uses $O(n^{2-2\varepsilon})$ space that supports range mode queries in $O(n^{\varepsilon})$ time [78], and for constant query time the best space bound achieved is $O(n^2 \log \log n/ \log^2 n)$ [79]. Given the rather large bounds for the range mode problem, the approximate variant of the problem was considered in [22].

**Our Results** The results on range mode and related problems we achieve in our paper [D7] are shown in Table 1.3. Our main result is a cell probe lower bound for range mode data structures. We prove that any data structure that uses $S$ space needs $\Omega(\log n/ \log \frac{Sw}{n})$ time for a range mode query. In particular this means that nay data structure that supports queries in constant time needs $\Omega(n^{1+O(1)})$ space and data structures that uses $O(n \log^{O(1)} n)$ space needs $\Omega(\log n/ \log \log n)$ time for a range mode query. We prove our lower bound with the technique developed by Pătraşcu and Thorup in [76, 77], by reducing the lopsided set disjointness problem from a communication complexity to a set of approximately $n/ \log n$ parallel queries on a range mode data structure.

We also consider the range $k$-frequency problem and using reductions we show that for any constant $k > 1$ the problem is equivalent to 2D range stabbing. In the restricted case where $k = 1$, the problem corresponds to determining whether there is a unique label in a subarray. We show, somewhat surprisingly,

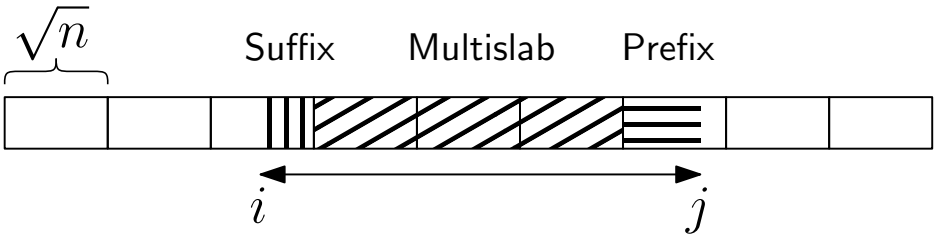



Figure 1.5: Idea for our 3-Approximate Range Mode data structure.

that for near-linear space data structures, determining whether there is a label occurring exactly twice (or $k > 1$ times) in a subarray, is exponentially harder than determining if there is a label occurring exactly once. Specifically, we reduce range 1-frequency to four-sided 3D orthogonal range emptiness, which can be solved with $O(\log^2 \log n)$ query time and $O(n \log n)$ space by a slight modification of the data structure presented in [1].

We give improved upper bounds for the $c$-approximate range mode problem. First we present a data structure for the 3-approximate range mode problem that uses $O(n)$ space and supports queries in constant time.

Finally, we use our 3-approximate range mode data structure, to develop a data structure for $(1 + \varepsilon)$-approximate range mode. This data structure uses $O(\frac{n}{\varepsilon})$ space and answers queries in $O(\log \frac{1}{\varepsilon})$ time. This removes the dependency on $n$ in the query time compared to the previously best data structure, while matching the space bound. Thus, we have a linear space data structure with constant query time for the $c$-approximate range mode problem for any constant $c > 1$. We note that we get the same bound if we build on the 4-approximate range mode data structure from [22].

Our data structure for 3-approximate range mode it is based on the following simple idea. Assume that we have partitioned a query into three parts, then the frequency of the mode in the entire range is at most three times the maximum individual frequency found in one of the parts. We do a recursive subdivision based on this insight. The input array of size $n$ is partitioned into $\sqrt{n}$ slabs of equal size and we store the mode for every consecutive sequence of slabs (multi-slab), and for each slab we additionally store the mode of all prefixes and suffixes of the slab. This way we can divide all queries that cover a slab into three, a suffix of one slab, a multi-slab, and a prefix of one slab, see Figure 1.5. Queries contained in one slab are handled recursively. We employ a lowest common ancestor (LCA) data structure [44] on top of this tree. We use the LCA data structure to locate the highest node where the subdivision divides the query, and in this node we determine the maximum in the three parts that constitute the query in constant time. Finally we reduce the space to $O(n)$ by using simple bit packing techniques.

We use our 3-approximate range mode data structure to develop a data structure for $(1 + \varepsilon)$-approximate range mode. This data structure is based on the following simple insight also used in [22]. To approximate the answer to any query starting at index one, we only need to know the indices, $t \geq 1$

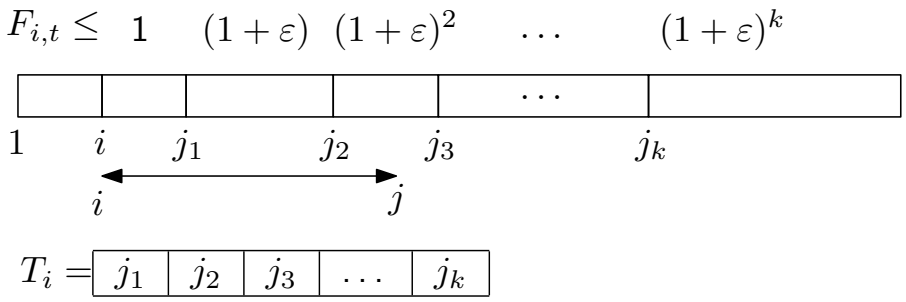



Figure 1.6: Idea for $(1+\varepsilon)$-approximate range mode data structure. The figure shows the consecutive ranges of $A$ where the frequency of the mode in $A[i,t]$ for $t \geq 1$ is between $(1+\varepsilon)^{k-1}$ and $(1+\varepsilon)^k$ and the indices $j_k$ separating them for $k = 1, \ldots, \log_{(1+\varepsilon)}(n)$. These indices defines the sorted array $T_i$. An $(1+\varepsilon)$ approximation of $F_{i.j}$ can be computed by a predecessor search for $j$ in $T_i$. In the example query, this search yields $j_2$, and we know that the frequency of $A[j_2]$ in $A[i,j]$ is at least $(1+\varepsilon)$ and that $F_{i,j}$ is at most $(1+\varepsilon)^2$.

where $F_{1,t}$ becomes $(1+\varepsilon)^k$ for $k = 0, 1, \ldots$, e.g. it has increased by a factor of $(1+\varepsilon)$. This gives a sorted list of $O(\log_{(1+\varepsilon)}(n)) = O(\frac{1}{\varepsilon}\log n)$ indices where the approximation changes. An approximate range mode query in $A[1,j]$ is then answered by doing a predecessor search for $j$ in this list, and returning the associated element. For a simple example see Figure 1.6. We can repeat this construction for all $n$ possible starting indices which requires $O((n/\varepsilon)\log n)$ space. These lists are, however, very similar, and by exploiting this one can reduce the space needed to store the information contained in these lists to $O(n/\varepsilon)$. In [22] the authors employ partially persistent search trees to represent these lists in $O(n/\varepsilon)$ space while supporting predecessor searches in any version. This gives a query bound of $O(\log\log_{(1+\varepsilon)} n)$ time. Instead of using persistence we use various tables to store the $O(n/\varepsilon)$ needed values and we employ succinct rank and select data structures [56] to efficiently extract the information we need from these tables. We end up with a data structure that can access any entry in any of the $n$ lists defined above in constant time. To avoid searching a list of size $O(\log_{(1+\varepsilon)} n)$ in a query, we initially use our 3-approximate range mode data structure to reduce the size of the interval of indices we need to consider to $O(\log_{(1+\varepsilon)} 3) = O(\frac{1}{\varepsilon})$ which we then binary search.

In conclusion, our data structure for the $(1+\varepsilon)$ approximate range mode problem uses $O(\frac{n}{\varepsilon})$ space and answers queries in $O(\log\frac{1}{\varepsilon})$ time. This removes the dependency on $n$ in the query time compared to the previously best data structure, while matching the space bound. Thus, we have a linear space data structure with constant query time for the $c$-approximate range mode problem for any constant $c > 1$.

## 1.3 Fault Tolerance

The final part originates from a tendency to avoid sophisticated error checking and correction circuitry that would impose non-negligible costs in hardware performance and price in the design of todays high speed memory technologies. Contemporary memory devices such as SRAM and DRAM can be unreliable due to a number of factors, such as hardware errors, power failures, radiation, and cosmic rays that can temporarily affect the memory behavior resulting in unpredictable, random, independent failures known as soft memory errors. In applications where large memory capacities are needed at low cost, it makes sense to assume that the algorithms themselves are in charge for dealing with memory faults. Since the amount of cosmic rays increases dramatically with altitude, soft memory errors are of special concern in fields like avionics or space research.

Memory devices continually become smaller, work at higher frequencies and lower voltages, and in general have increased circuit complexity [30]. Unfortunately, these improvements come at the cost of reliability [89,90] and soft memory error rates are expected to rise for both DRAM and SRAM memories [89].

An unreliable memory can cause problems in most software ranging from the harmless to the very serious, such as breaking cryptographic protocols [20, 94], taking control of a Java Virtual Machine [48] or breaking smart-cards and other security processors [6, 7, 85]. Corrupted content in memory cells can greatly affect many standard algorithms. For instance, in a typical binary search in a sorted array, a single corruption encountered in the early stages of the search can cause the search path to end $\Omega(N)$ locations away from its correct position, and a single corrupted value can induce as much as $\Theta(n^2)$ inversions in the output of a standard implementation of mergesort [39]. Replication of data can help in dealing with corruptions, but is not always feasible, since the time and space overheads of storing and fetching replicated values can be significant.

Memory corruptions have been addressed in various ways, both at the hardware and software level. At the hardware level, the soft memory errors can be handled by means of error detection mechanisms such as parity checking, redundancy or Hamming codes. Unfortunately, implementing these mechanisms incur penalties with respect to performance, size and money. Therefore, memories using these technologies are rarely found in large scale computing clusters or ordinary workstations. On the software level, a series of low-level techniques have been proposed for dealing with the soft memory errors, many of them coping with corrupted instructions. Examples include algorithm based fault tolerance [54], assertions [84], control flow checking [98], or procedure duplication [82].

A multitude of algorithms that deal with unreliable information in various ways were developed during the last decades. Aumann and Bender [11] introduced *fault tolerant pointer-based data structures*. In their model, error detection is done upon access, *i.e.* accessing a faulty pointer yields an error message. Obviously, this is not always the case in practice, since a pointer might get corrupted to a valid value and thus an error is not reported. Furthermore, their algorithms allow a certain amount of the data structure to be lost upon corrup-

tions, and this is not accepted in many practical applications. The *liar model* considers algorithms in a comparison model where the result of a comparison is unreliable. Work in this model include fundamental problems such as sorting and searching [21,65,83]. A standard technique used in the design of algorithms in the liar model is query replication, which is not of much help when memory cells, and not comparisons, are unreliable. Kutten and Peleg [63,64] introduced the concept of *fault local mending* in the context of distributed networks. A problem is fault locally mendable if there exists a correction algorithm whose running time depends only on the (unknown) number of faults. Some other works studying network fault tolerance include [32, 46, 52, 53, 60, 66, 74].

Finocchi and Italiano [39] introduced the *faulty-memory random access machine*, based on the traditional RAM model. In this model, memory corruptions can occur at any time and at any place in memory during the execution of an algorithm, and corrupted memory cells cannot be distinguished from uncorrupted cells. In the faulty-memory RAM, it is assumed that there is an adaptive adversary, that chooses how, where, and when corruptions occur. The model is parameterized by an upper bound, $\delta$, on the number of corruptions the adversary can perform during the lifetime of an algorithm, and $\alpha \leq \delta$ denotes the actual number of corruptions that takes place. Note that $\alpha$ is not known by the algorithm. Motivated by the fact that registers in the processor are considered incorruptible, $O(1)$ safe memory locations are provided. Moreover, it is assumed that reading a word from memory is an atomic operation. In randomized computation, as defined in [39], the adversary does not see the random bits used by an algorithm. An algorithm is *resilient* if it works correctly on the set of uncorrupted cells in the input. For instance, a resilient sorting algorithm outputs all uncorrupted elements in sorted order while corrupted elements can appear at arbitrary positions in the output. We say that a sequence of elements is *faithfully ordered* if all the uncorrupted elements in the sequence appear in sorted order.

Several important results has been achieved in the faulty-memory RAM. In the original paper, Finocchi and Italiano [39] proved lower bounds and gave (non-optimal) resilient algorithms for sorting and searching. Algorithms matching the lower bounds for sorting and searching(expected time) were presented in [36]. An optimal resilient sorting algorithm takes $\Theta(n \log n + \delta^2)$ time, whereas optimal searching is performed in $\Theta(\log n + \delta)$ time. Furthermore, in [38] a resilient deterministic search tree that performs searches and updates in $O(\log n + \delta^2)$ time amortized was developed. Finally, in [80] it was shown that resilient sorting algorithms are of practical interest.

The cost of resilient algorithms can be expressed in two slightly different ways. For example, a resilient sorting algorithm using $O(N \log N + \delta^2)$ time can also be described as a sorting algorithm that uses $O(N \log N)$ time while tolerating $O(\sqrt{N \log N})$ corruptions. While the former version is more general, the latter emphasizes the fact that one can use a resilient sorting algorithm for "free" asymptotically as long as the number of corruptions is $O(\sqrt{N \log N})$.

**Reliable Values** Perhaps the most important concept used in the design of resilient algorithms, in particular resilient data structures, is *reliable values* (or variables). A reliable value is stored solely in unreliable memory and can be retrieved and updated reliably. The idea is fairly simple. To store a value $v$ as a reliable value, $v$ is written in $2\delta+1$ consecutive memory cells in the unreliable memory. The point is that only $\delta$ corruptions can occur and thus at least $\delta+1$, more than half, of the cells will always contain $v$. Here we note that if we use $\delta$ cells or less, they may be all be corrupted and we cannot guarantee that $v$ can be retrieved. To extract $v$ all, we find the most frequent element in the $2\delta+1$ cells and we know that this value appears in more than half of cells. This leads is to the *majority problem* considered in [23]. Here, the input is a sequence of $k$ elements and we are promised that at least $\lceil \frac{k+1}{2} \rceil$ of the elements are equal. Let $x$ denote this uniquely determined value. The goal is to recover $x$ from the $k$ items efficiently. In the comparison model there is a simple algorithm by Boyer and Moore [23] that extracts $x$ while using $O(1)$ additional space and $O(k)$ time. The algorithm keeps two variables in memory: a majority candidate, $a$, and a counter, $c$, initially equal to zero. The sequence of elements is scanned once and in each step one of the following three operations is performed. If $c$ is zero the current element is made the new candidate stored in $a$ and $c$ is set to one. If the current element is equal to $a$, $c$ is incremented by one, and if they differ $c$ is decremented by one. When the scan has completed, $a$ is the majority element of the sequence. If the sequence does not have a majority elements the output of the algorithm is undefined (it returns some arbitrary element).

We can extract a reliable variable $v$ by running the majority algorithm while keeping the candidate element and the counter in the $O(1)$ safe memory cells. It follows that any value can be maintained reliably in $O(\delta)$ space and it can be accessed and updated in $O(\delta)$ time. This means that we can make any algorithm or data structure resilient with a multiplicative cost of $O(\delta)$ in time and space. We call these naive algorithms.

### 1.3.1 Priority Queues Resilient to Memory Faults

In [D8] we give a resilient priority queue that supports *insert* and *deletemin* in $O(\log n + \delta)$ amortized time. The deletemin operation of a resilient priority queue must either return the smallest uncorrupted element in the queue or a corrupted one. In particular this means that inserting $n$ elements into the priority queue and calling deletemin $n$ times must be a resilient sorting algorithm. Our priority queue was the first data structure that achieved a time bound of $O(\log n + \delta)$ for queries and updates.

The organization of our data structure is inspired by the cache oblivious priority queue in [9], and we make extensive use of the resilient binary merging algorithm from [40]. We store the elements in levels of increasing size, each level storing an *up* buffer and a *down* buffer. We maintain a natural invariant in our data structure stating that uncorrupted elements in the down buffer at level $i$ are smaller than the uncorrupted elements in the up and down buffer at level $i+1$. We use the resilient merging algorithm to efficiently move the elements between the levels, such that large elements are pushed *upwards* into the larger lists and

small are pulled *downwards* into the small lists without breaking the invariant. We use a simple unsorted insertion buffer of size $\Theta(\log n + \delta)$. Whenever this list is full we merge it with the smallest level where the up and down buffer are of size $\Theta(\log^2 n + \delta^2)$. If this level is filled elements are recursively pushed to the second level, which is twice as large, and so on. In a deletemin we consider all the elements in the insertion buffer and the smallest $O(\log n + \delta)$ elements from the first level. If this deletion makes the lowest level to small we fetch elements from the next level, and this continues recursively.

The hardest part is to ensure that we maintain the relationship between the elements in lists of different levels in spite of corruptions when we merge sorted lists and redistribute the elements. This is a somewhat technical issue that we will not consider here. The main point is that the extra cost we incur for this as well as the extra time we need for resilient merging compared to standard merging is dwarfed because of rather large sizes we have chosen for our buffers.

We also prove a lower bound stating that any data structure that does not use $\delta$ time for an insert followed by a deletemin cannot be a resilient priority queue. This lower bound is only valid under the assumption that the data structure does not store any elements in safe memory between operations. Basically, the lower bound proves that if we insert an element and it is not compared to at least $\delta$ other elements from the data structure, then an adversary can force the deletemin operation to return a wrong answer. Combining this with the lower bound of $\Omega(\log n)$ from the comparison model we get a matching lower bound of $\Omega(\log n + \delta)$ comparisons.

### 1.3.2 Searching in the Presence of Memory Corruptions

In [D1] we consider comparison based search problems in the faulty memory RAM. A resilient searching algorithm [36] has the following semantics when searching for an element $e$ in a set $S$. If there is an uncorrupted copy of $e$ in $S$ the query must return yes, if zero elements in $S$ are equal to $e$ it must return no. Otherwise, the result is undefined, e.g. an algorithm cannot give a wrong answer.

In [D1] we give an optimal resilient algorithm for searching a sorted input array of $n$ elements. This algorithm uses $O(\log n + \delta)$ time, matching the lower bound proved in [40]. Our algorithm is based on the classic binary search algorithm which we extend and achieve the following three things. First, we ensure that we only advance one level in the binary search into the wrong part of the array for each corrupted element we encounter. This we achieve making each step verify both the left and the right boundary of the subarray not yet ruled out by our binary search. Secondly, we design a verification procedure that determines whether the search has been misled. This is invoked whenever a binary search would normally end, that is, when the algorithm has located two consecutive elements that compare differently to the search element. This algorithm scans the neighborhoods of the these two elements in parallel. An element to the left agrees if it is smaller than the search element and disagrees if it is larger. The point is that if we find an agreeing and then a disagreeing element to the left, then at least one of them is corrupted. If the number of

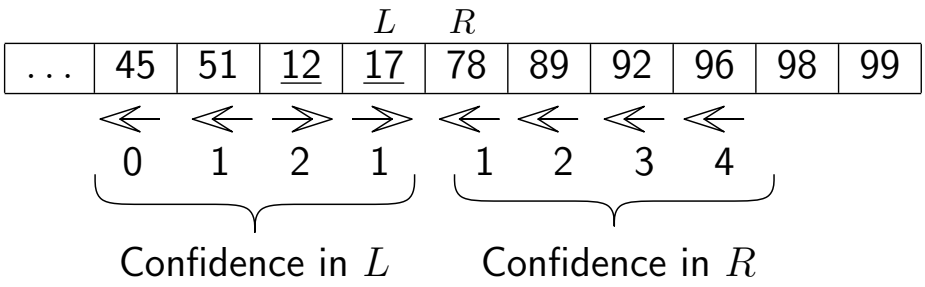



Figure 1.7: Verifying the result of a binary search. $L$ and $R$ are the left and right boundary computed by a binary search, and the underlined elements are corrupted. Each arrow indicate in what direction the search should continue if you query that element. Two arrows pointing in opposite directions means that at least one of the two corresponding elements are corrupted. The figure also shows that if the confidence in $L$ or $R$ reaches zero and we have scanned $t$ elements in either direction then we have discovered $t/2$ corrupted elements at least.

disagreeing elements becomes equal to the number of agreeing elements on one of the sides the procedure *fails*. In this case we backtrack the search. If the difference becomes larger than the number of remaining corruptions possible on both sides, the procedure succeeds. In this case we know that there is at least one uncorrupted agreeing element on each side and we can safely end the search. The important thing about this procedure is that the running time is linear in the number of corrupted elements it reads or the algorithm ends. The algorithm is illustrated in Figure 1.7. Finally, we ensure that we only read each corrupted element once by ensuring that we only read each element once. This we achieve by considering the input as $\delta + 1$ disjoint sequences, where the $i$'th sequence consists of every $\delta + 1$'th element from the input[2]. Initially, we search the first sequence and whenever we determine that we have been misled by a corruption (failed verification) we skip to the following sequence. These three properties ensure that the combined time we need for verifying and backtracking our search is $O(\delta)$, and we get an algorithm with a running time of $O(\log n + \delta)$.

With our optimal binary search algorithm in hand we design a dynamic data structure that uses linear space and supports updates and queries in $O(\log n+\delta)$ time, amortized and worst case respectively. Our data structure is based on the binary search trees in [24] that maintain the elements in a tree that is almost completely balanced, e.g. only the $O(1)$ last levels are not complete, and standard bucketing techniques.

We divide the elements in sorted order into simple bucket data structures each storing $\Theta(\delta \log n)$ elements. We represent each bucket by its largest element and these elements are stored naively in the binary search tree data structure from [24], e.g. the entire tree is stored as a reliable variable by repli-

[2]This is not completely true since we also reserve elements to be used only by the verification algorithm, but almost

cating both elements and structural information of the tree $2\delta + 1$ times. The tree is stored in breadth-first order. In a query we run our resilient binary search algorithm on the last complete level of the tree and do a naive tree search on the subtree this search identifies. This yields a simple bucket data structure which is also searched naively. Since the bucket structures are so large and we allow them to vary by a constant factor, updates to the top tree are very few and the amortized cost of these is small.

### 1.3.3 Fault Tolerant External Memory Algorithms

In [D5] we consider resilient algorithms in the external memory model (IO model). Memory corruptions are of particular concern for applications dealing with massive amounts of data since such applications typically run for a very long time, and are thus more likely to encounter memory cells containing corrupted data. However, algorithms designed in the RAM model assume that an infinite amount of memory cells are available. This is not true for typical computers where internal memory is limited and elements are transferred between the memory and a much larger, but dramatically slower, hard drive in large consecutive blocks. This means that it is important to design algorithms with a high degree of locality in their memory access pattern, that is, algorithms where data accessed close in time is also stored close in memory. This situation is modeled in the *I/O model* of computation [2]. In this model a disk of unlimited size and a memory of size $M$ are available. Elements are transferred between disk and memory in blocks of size $B$ and computation is performed on elements in memory only. The complexity measure is the number of block transfers (*I/O*s) performed.

For the I/O model, a comprehensive list of results have been achieved. It is shown in [2] that sorting $N$ elements requires $\Theta(N/B \log_{M/B}(N/B))$ I/Os. See recent surveys [8, 92] for an overview of other results. In the I/O model, a comparison based dictionary with optimal queries can be achieved with a B-tree [15], which supports queries and updates in $O(\log_B N)$ I/Os.

Current resilient algorithms do not scale past the internal memory of a computer and thus, it is currently not possible to work with large sets of data I/O-efficiently while maintaining resiliency to memory corruptions. Since both models become increasingly interesting as the amount of data increases, it is natural to consider whether it is possible to achieve resilient algorithms that use the disk optimally. Very recently, this was also proposed as an interesting direction of research by Finocchi *et al.* [37, 40].

We extend the faulty memory RAM to an external memory equivalent by allowing the adversary to corrupt elements on disk as well as elements in memory. We give IO efficient versions of all algorithms and data structures considered in the resilient memory model. These algorithms can tolerate orders of magnitude more corruptions compared to their internal memory counterparts without affecting the asymptotic IO complexity for realistic values of $N, B$ and $\delta$.

We prove an $\Omega(\frac{1}{\varepsilon} \log_B n + \delta/B^{1-\varepsilon})$ lower bound on the IO complexity for resilient comparison based search algorithms. This bound is achieved by considering a worst case adversary model. In this adversary model every block of

elements fetched from disk is first passed to the adversary who then decides what elements to corrupt before the algorithm sees them. We let the same adversary decide where the search element resides among the elements that have not yet been ruled out by previous IOs. The adversary uses the following strategy. The search algorithm performs an IO and fetches $B$ elements from disk. The adversary artificially divides the sorted set of input elements into $B^\varepsilon$ slabs of equal size and finds the slab containing the fewest elements from this last IO. The adversary decides that search element lies in this slab and corrupts all the elements in the slab from the IO. This game is then played recursively on this slab. After a bit of analysis of this strategy we end up with a lower bound of $\Omega(\frac{1}{\varepsilon}\log_B n + \delta/B^{1-\varepsilon})$ IOs.

We design a linear space data structure with a matching query bound by combining the B-tree with our resilient binary searching algorithm from [D1] described above. Given a constant $1/\log B < c \leq 1$, we make a tree with fanout $B^c$ and replicate each guiding element $B^{1-c}$ times. This gives a tradeoff between the height of the tree and the number of corruptions needed to misguide a search. Similar to the division into $n/\delta$ sequences in our binary search algorithm, we store $\Theta(\delta/B^{1-c})$ copies of the tree, such that the tree is stored as a reliable value, and whenever a query discovers $\Theta(B^{1-c})$ corruptions we switch to the following tree. We use a verification procedure similar to the procedure we designed for our optimal binary search algorithm and get a data structure that supports queries using $O(\frac{1}{c}\log_B N + \frac{\delta}{B^{1-c}})$ IOs and $O(\log n + \delta)$ time.

We also consider IO efficient resilient sorting algorithms. Under the assumption that $\delta < M^\gamma$ for $\gamma < 1$, we can construct a multi-way merging algorithm based on the resilient binary merging algorithm. This is basically a binary tree where each node reliably stores a running instance of the resilient merging algorithm from [40]. With this multi-way merging algorithm we can perform the standard IO efficient version of merge-sort, and we get a sorting algorithm that uses $O(\frac{N}{B}\log_B \frac{N}{B})$ IOs. The multi-way merging algorithm is also the main component in the IO efficient priority queue we design which supports insert and deletemin in optimal $O(\frac{1}{B}\log_{M/B} N/B)$ IOs and $O(\log n + \delta)$ time amortized.

### 1.3.4 Counting in a Hostile Environment

Finally, in our efforts to understand the possibilities and limits of the model we consider counter data structures that are stored exclusively in the faulty memory in [D6]. Such data structures makes sense when $\omega(1)$ counters are needed.

There are some simple lower bounds that set the rules for what we can achieve with a counter data structure stored solely in corruptible memory. For instance, if a data structure uses $\delta$ space or less then the adversary can corrupt the entire thing. Secondly, any deterministic query algorithm that uses $\delta$ time or less could end up only reading data controlled by the adversary and cannot guarantee anything. Finally, if an increment takes $t \leq \delta$ time, the adversary can simulate $\delta/t \geq 1$ increments without detection. This lower bound gives a tradeoff between the update time and the accuracy of the counter data structure, e.g. any data structure that supports increments in $t$ time can only guarantee

| Time ($n$ increments) | Query time | Additive error $\gamma$ | Space |
|---|---|---|---|
| $O(\delta n)$ | $O(\delta)$ | 0 | $O(\delta)$ |
| $O(nt \log(\delta/t) + \alpha \log(\alpha/t))$ | $O(\delta)$ | $\alpha/t$ | $O(\delta)$ |
| $O(n + \alpha \log \alpha)$ | $O(\delta)$ | $\alpha \log \delta$ | $O(\delta)$ |
| $O(n)$ | $O(\delta^2)$ | $O(\alpha^2)$ | $O(\delta)$ |
| $O(n + \alpha\sqrt{\delta})$ | $O(\delta)$ | $\alpha$ | $O(\delta)$ |
| Expected $O(n)$ | $O(\delta)$ | $\alpha$ | $O(\delta)$ |

Table 1.4: Overview of the different trade-offs we obtain for counter data structures stored solely in corruptible memory. The first data structure is the naive way of counting.

that a query returns a value that is at most $\delta/t$ away from the number of increments performed. Formally, let $C$ be the number of times a counter has been incremented, we say that a counter data structure has additive error $\gamma$ if it returns a value $v \in [C - \gamma, C + \gamma]$. First we note that it is trivial to store a counter exactly in faulty-memory using $O(\delta)$ space and $O(\delta)$ time for increments and queries by storing the value of the counter as a reliable value, e.g. the naive approach. This only touches one part of the trade-off. In our paper [D6] we give a range of data structures giving different trade-offs between the update time and the additive error when the update time is $o(\delta)$.

**Our Results** We have shown the various trade-offs we obtain in Table 1.4.

The first row shows the bounds of the naive approach. The second and third row in the table are the result of replicating each bit of the number of the increments depending on the significance of the bit. The fourth data structure is a simple round robin scheme that obliviously stores $\Theta(\delta)$ counters that are incremented in round robin fashion. The interesting part here is that we actually get a non-trivial bound on the error, albeit the query time is quite large. The final two data structures uses one cell to store one increment and keeps the additive error down to one per corruption. Whenever $\Theta(\delta)$ increments have been performed we record this in a reliable value and reset the data structure. The hard part is to find a cell that does not yet store an increment or determine that $\Omega(\delta)$ increments have been performed since the last reset. In the deterministic case we get a penalty of $O(\alpha\sqrt{\delta})$ time over $n$ increments to actually do this. When we employ randomization to handle both problems we get a data structure with amortized constant increment time in expectation.

# Part I

# Sums

# Chapter 2

## The $k$ Maximal Sums Problem

In the $k$ maximal sums problem the input is an array $A$ of size $n$ and the task is to compute the $k$ largest sub-vectors for $1 \leq k \leq \binom{n}{2} + n$. The sub-vectors are allowed to overlap, and the output is $k$ triples of the form $(i, j, sum)$ where $sum = \sum_{s=i}^{j} A[s]$. The problem in [12] as an extension of the maximal sum problem. The solution for $k = 1$ does not seem to be extendable in any simple manner to obtain a linear algorithm for any $k$. Therefore, different solutions to this extended problem have emerged over the past few years. These results are summarized in Table 2.1.

A lower bound for the $k$ maximal sums problem is $\Omega(n+k)$, since an adversary can force any algorithm to look at each of the $n$ input elements and the output size is $\Omega(k)$.

**Our Results** In this chapter we close the gap between upper and lower bounds for the $k$ maximal sums problem. We design an algorithm computing the $k$ sub-vectors with the largest sums in an array of size $n$ in $O(n + k)$ time. We also describe algorithms solving the problem extended to any dimension. We begin by solving the two-dimensional problem where we obtain an $O(m^2 \cdot n + k)$ time algorithm for an $m \times n$ input matrix with $m \leq n$. This improves the previous best result [29], which was an $O(m^2 \cdot n + k \log k)$ time algorithm. This solution is then generalized to solve the $d$ dimensional problem in $O(n^{2d-1} + k)$ time, assuming for simplicity that all sides of the $d$-dimensional input matrix are equally long. Furthermore we describe how to minimize the additional space usage of our algorithms. The additional space usage of the one dimensional algorithm is reduced from $O(n + k)$ to $O(k)$. The input array is considered to be read only. The additional space usage for the algorithm solving the two-dimensional problem is reduced from $O(m^2 \cdot n + k)$ to $O(n + k)$ and for the general algorithm solving the $d$ dimensional problem the space is reduced from $O(n^{2(d-1)} + k)$ to $O(n^{d-1} + k)$.

Our main contribution is the first algorithm solving the $k$ maximal sums problem using $O(n + k)$ time and $O(k)$ space. The result is achieved by gen-

[1]The $k$ maximal sums problem can also be solved in $O(n + k)$ time by a reduction to Eppstein's solution for the $k$ shortest paths problem [34] which also makes essential use of Fredericksons heap selection algorithm. This reduction was observed independently by Hsiao-Fei Liu and Kun-Mao Chao [27].

| Paper | Time complexity |
|---|---|
| Bae & Takaoka [12] | $O(n \cdot k)$ |
| Bengtson & Chen [17] | $O(\min\{k + n\log^2 n, n\sqrt{k}\})$ |
| Bae & Takaoka [13] | $O(n \log k + k^2)$ |
| Bae & Takaoka [14] | $O((n + k) \log k)$ |
| Lie & Lin [68] | $O(n \log n + k)$ expected |
| Cheng *et al.* [29] | $O(n + k \log k)$ |
| Liu & Chao [27][1] | $O(n + k)$ |
| Our Contribution | $O(n + k)$ |

Table 2.1: Overview of results for the $k$ maximal sums problem

erating a binary heap that implicitly contains the $\binom{n}{2} + n$ sums in $O(n)$ time. The $k$ largest sums from the heap are then selected in $O(n + k)$ time using the heap selection algorithm of Frederickson [41]. The heap is built using partial persistence [33]. The space is reduced by only processing $k$ elements at a time. The resulting algorithm can be viewed as a natural extension of Kadane's linear time algorithm for solving the maximum sum problem introduced earlier.

**Outline** The remainder of this Chapter is structured as follows: In Section 2.1 the overall structure of our solution is explained. Descriptions and details regarding the algorithms and data structures used to achieve the result are presented in Sections 2.2, 2.3 and 2.4. In Section 2.5 we combine the different algorithms and data structures completing our algorithm. This is followed by Section 2.6 where we show how to use our algorithm to solve the problem in $d$ dimensions. Finally in Section 2.7 we explain how to reduce the additional space usage of the algorithms without penalizing the asymptotic time bounds.

## 2.1 Basic Idea and Algorithm

The term heap denotes a max-heap-ordered binary tree. The basic idea of our algorithm is to build a heap storing the sums of all $\binom{n}{2} + n$ sub-vectors and then use Fredericksons binary heap selection algorithm to find the $k$ largest elements in the heap.

In the following we describe how to construct a heap that implicitly stores all the $\binom{n}{2} + n$ sums in $O(n)$ time. The triples induced by the $\binom{n}{2} + n$ sums in the input array are grouped by their end index. The *suffix set* of triples corresponding to all sub-vectors ending at position $j$ we denote $Q^j_{\text{suf}}$, and this is the set $\{(i, j, sum) \mid 1 \leq i \leq j \ \wedge sum = \sum_{s=i}^{j} A[s]\}$. The $Q^j_{\text{suf}}$ sets can be incrementally defined as follows:

$$Q^j_{\text{suf}} \quad = \quad \{(j, j, A[j])\} \cup \{(i, j, s + A[j]) \mid (i, j - 1, s) \in Q^{j-1}_{\text{suf}}\}. \qquad (2.1)$$

As stated in equation (2.1) the suffix set $Q^j_{\text{suf}}$ consists of all suffix sums in $Q^{j-1}_{\text{suf}}$ with the element $A[j]$ added as well as the single element suffix sum $A[j]$.

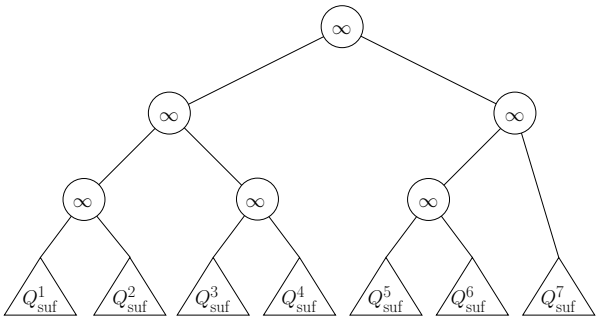



Figure 2.1: Example of a complete heap $H$ constructed on top of the $H^j_{\text{suf}}$ heaps. The input size is 7.

Using this definition, the set of triples corresponding to all $\binom{n}{2} + n$ sums in the input array is the union of the $n$ disjoint $Q^j_{\text{suf}}$ sets. We represent the $Q^j_{\text{suf}}$ sets as heaps and denote them $H^j_{\text{suf}}$. Assuming that for each suffix set $Q^j_{\text{suf}}$, a heap $H^j_{\text{suf}}$ representing it has been built, we can construct a heap $H$ containing all possible triples by constructing a complete binary heap on top of these heaps. The keys for the $n-1$ top elements is set to $\infty$ (see Figure 2.1). To find the $k$ largest elements, we extract the $n-1+k$ largest elements in $H$ using the binary heap selection algorithm of Frederickson [41] and discard the $n-1$ elements equal to $\infty$.

Since the suffix sets contain $\Theta(n^2)$ elements the time and space required is still $\Theta(n^2)$ if they are represented explicitly. We obtain a linear time construction of the heap by constructing an implicit representation of a heap that contains all the sums. We make essential use of a heap data structure to represent the $Q^j_{\text{suf}}$ sets that supports insertions in amortized constant time.

Priority queues represented as heap ordered binary trees supporting insertions in amortized constant time already exist. One such data structure is the self-adjusting binary heaps of Tarjan and Sleator described in [86] called Skew Heaps. The Skew heap is a data structure reminiscent of Leftist heaps [31, 61]. Even though the Skew heap would suffice for our algorithm, it is able to do much more than we require. Therefore, we design a simpler heap which we will name *Iheap.* The essential properties of the Iheap are that it is represented as a heap-ordered binary tree and that insertions are supported in amortized constant time.

We build $H^{j+1}_{\text{suf}}$ from $H^j_{\text{suf}}$ in $O(1)$ time amortized without destroying $H^j_{\text{suf}}$ by using the partial persistence technique of [33] on the Iheap. This basically means that the $H^j_{\text{suf}}$ heaps become different versions of the same Iheap. To make our Iheap partially persistent we use the node copying technique [33]. The cost of applying this technique is linear in the number of changes in an update. Since only the insertion procedure is used on the Iheap, the extra cost of using partial persistence is the time for copying amortized $O(1)$ nodes per insert operation. The overhead of traversing a previous version of the data structure is $O(1)$ per data/pointer access.

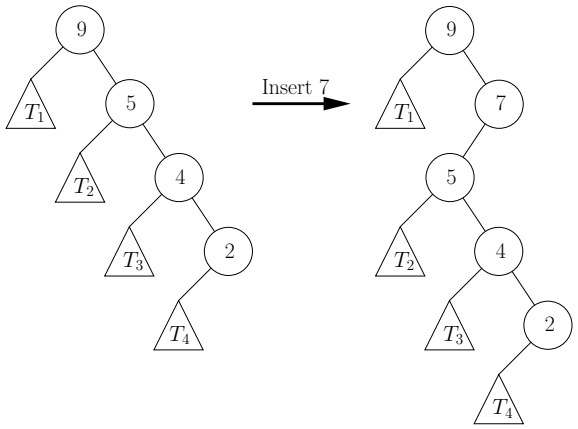



Figure 2.2: An example of an insertion in the Iheap. The element 7 is compared to 2,4 and 5 in that order, and these elements are then removed from the rightmost path.

## 2.2 Binary Heaps

The main data structure of our algorithm is a heap supporting constant time insertions in the amortized sense. The heap is not required to support operations like deletions of the minimum or an arbitrary element. All we do is insert elements and traverse the structure top down when we apply Fredericksons heap selection algorithm. We design a simple binary heap data structure Iheap by reusing the idea behind the Skew heap and perform all insertions along the rightmost path of the tree starting from the rightmost leaf.

A new element is inserted into the Iheap by placing it in the first position on the rightmost path where it satisfies the heap order. This is performed by traversing the rightmost path bottom up until a larger element is found or the root is passed. The element is then inserted as a right child of the larger element found (or as the new root). The element it is replacing as a right child (or as root) becomes the left child of the inserted element. An insertion in an Iheap is illustrated in Figure 2.2. If $O(\ell)$ time is used to perform an insertion operation because $\ell$ elements are traversed, the rightmost path of the heap becomes $\ell - 1$ elements shorter. Using a potential function on the length of the rightmost path of the tree, we get amortized constant time insertion for the Iheap. Each element is passed on the rightmost path only once, since it is then placed on the left-hand side of element passing it, and never returns to the rightmost path.

**Lemma 2.1** *The Iheap supports insertion in amortized constant time.*

## 2.3 Partial Persistence and $H^j_{\mathbf{suf}}$ Construction

As mentioned in Section 2.1 the $H^j_{\text{suf}}$ heaps are built based on equation (2.1) using the partial persistence technique of [33] on an Iheap.

Data structures are usually *ephemeral*, meaning that an update to the data structure destroys the old version, leaving only the new version available for

use. An update changes a pointer or a field value in a node. Persistent data structures allow access to any version old or new. Partially persistent data structures allow updates to the newest version, whereas fully persistent data structures allow updates to any version. With the partial persistence technique known as node copying, linked ephemeral data structures, with the restriction that for any node the number of other nodes pointing to it is $O(1)$, can be made partially persistent [33]. The Iheap is a binary tree and therefore trivially satisfies the above condition. The amortized cost of using the node copying technique is bounded by the cost of copying and storing $O(1)$ nodes from the ephemeral structure per update.

The basic idea of applying node copying to the Iheap is the following (see [33] for further details). Each persistent node contains one version of each information field in an original node, but it is able to contain several versions of each pointer (link to other node) differentiated by time stamps (version numbers). However, there are only a constant number of versions of any pointer, why each partially persistent Iheap node only uses constant space. Accessing relatives of a node in a given version is performed by finding the pointer associated with the correct time stamp. This is performed in constant time making the access time in the partially persistent Iheap asymptotically equal to the access time in an ephemeral Iheap.

According to equation (2.1), the set $Q^{j+1}_{\text{suf}}$ can be constructed from $Q^j_{\text{suf}}$ by adding $A[i+1]$ to all elements in $Q^j_{\text{suf}}$ and then inserting an element representing $A[i+1]$. To avoid adding $A[i+1]$ to each element in $Q^j_{\text{suf}}$, we represent each $Q^j_{\text{suf}}$ set as a pair $\langle \delta_j, H^j_{\text{suf}} \rangle$, where $H^j_{\text{suf}}$ is a version of a partial persistent Iheap containing all sums of $Q^j_{\text{suf}}$ and $\delta_j$ is a value that must be added to all elements. With this representation a constant can be added to all elements in a heap implicitly by setting the corresponding $\delta$. Similar to the way the $Q^j_{\text{suf}}$ sets were defined by equation (2.1) we get the following incremental construction of the pair $\langle \delta_{j+1}, H^{j+1}_{\text{suf}} \rangle$:

$$\langle \delta_0, H^0_{\text{suf}} \rangle = \langle 0, \emptyset \rangle \ , \tag{2.2}$$

$$\langle \delta_{j+1}, H^{j+1}_{\text{suf}} \rangle = \langle \delta_j + A[i+1], H^j_{\text{suf}} \cup \{-\delta_j\} \rangle \ . \tag{2.3}$$

Let $\langle \delta_j, H^j_{\text{suf}} \rangle$ be the latest pair built. To construct $\langle \delta_{j+1}, H^{j+1}_{\text{suf}} \rangle$ from this pair, an element with $-\delta_j$ as key is inserted into $H^j_{\text{suf}}$. We insert this value, since $\delta_j$ has not been added to any element in $H^j_{\text{suf}}$ explicitly, and because the sum $A[i+1]$ that the new element are to represent must be added to all elements in $H^j_{\text{suf}}$ to obtain $H^{j+1}_{\text{suf}}$. Since we apply partial persistence on the heap, $H^j_{\text{suf}}$ is still intact after the insertion, and a new version of the Iheap with the inserted element included has been constructed. $H^{j+1}_{\text{suf}}$ is this new version and $\delta_{j+1}$ is set to $\delta_j + A[i+1]$. Therefore, the newly inserted element represents the sum $-\delta_j + \delta_j + A[i+1] = A[i+1]$. This ends the construction of the new pair $\langle \delta_{j+1}, H^{j+1}_{\text{suf}} \rangle$. Since all sums from $H^j_{\text{suf}}$ get $A[i+1]$ added because of the increase of $\delta_{j+1}$ compared to $\delta_j$ and the new element represents $A[i+1]$ we conclude that $\langle \delta_{j+1}, H^{j+1}_{\text{suf}} \rangle$ represents the set $Q^{j+1}_{\text{suf}}$. The time needed for constructing $H^{j+1}_{\text{suf}}$ is the time for inserting an element into a partial persistent

Iheap. Since the size of an Iheap node is $O(1)$, by Lemma 2.1 and the node copying technique, this is amortized constant time

**Lemma 2.2** *The time for constructing the n pairs* $\langle \delta_j, H^j_{suf} \rangle$ *is* $O(n)$.

## 2.4 Fredericksons Heap Selection Algorithm

The last algorithm used by our algorithm is the heap selection algorithm of Frederickson, which extracts the $k$ largest[2] elements in a heap in $O(k)$ time. Input to this algorithm is an infinite heap ordered binary tree. The infinite part is used to remove special cases concerning the leafs of the tree, and is implemented by implicitly appending nodes with keys of $-\infty$ to the leafs of a finite tree. The algorithm starts at the root, and otherwise only explores a node if the parent already has been explored.

The main part of the algorithm is a method for locating an element, $e$, such that $k$ is larger than at least $k$ elements and at most $ck$ elements, for some constant $c$. After this element is found the input heap is traversed and all elements larger than $e$ are extracted. Standard selection [19] is then used to obtain the $k$ largest elements from the $O(k)$ extracted elements. We refer to [41] for the details.

**Theorem 2.1 ( [41])** *The k largest elements in a heap can be found in* $O(k)$ *time.*

## 2.5 Combining the Ideas

The heap constructed by our algorithm is actually a graph because the $H^j_{\text{suf}}$ heaps are different versions of the same partially persistent Iheap. Also, the roots of the $H^j_{\text{suf}}$ heaps include additive constants $\delta_j$ to be added to all of their descendants. However, if we focus on any one version, it will form an Iheap. This Iheap we can construct explicitly in a top down traversal starting from the root of this version, by incrementally expanding it as the partial persistent nodes are encountered during the traversal. Since the size of a partially persistent Iheap node is $O(1)$, the explicit representation of an Iheap node in a given version can be constructed in constant time.

However, the entire partially persistent Iheap does not need to be expanded into explicit heaps, only the parts actually visited by the selection algorithm. Therefore, we adjust the heap selection algorithm to build the visited parts of the heap explicitly during the traversal. This means that before any node in a $H^j_{\text{suf}}$ heap is visited by the selection algorithm, it is built explicitly, and the newly built node is visited instead. We remark that the two children of an explicitly constructed Iheap node, can be nodes from the partially persistent Iheap.

The additive constants associated with the roots of the $H^j_{\text{suf}}$ are also moved to the expanding heaps, and they are propagated downwards whenever they are

[2] Actually Frederickson [41] considers min-heaps.

encountered. An additive constant is pushed downwards from node $v$ by adding the value to the sum stored in $v$, removing it from $v$, and instead inserting it into the children of $v$. Since nodes are visited top down by Fredericksons selection algorithm, it is possible to propagate the additive constants downwards in this manner while building the visited parts of the partially persistent Iheap. Therefore, when a node is visited by Fredericksons algorithm, the key it contains is equal to the actual sum it represents.

**Lemma 2.3** *Explicitly constructing $t$ connected nodes in any fixed version of a partially persistent Iheap while propagating additive values downwards can be done in $O(t)$ time.*

**Theorem 2.2** *The algorithm described in Section 2.1 is an $O(n+k)$ time algorithm for the $k$ maximal sums problem.*

*Proof.* Constructing the pairs $\langle \delta_j, H^j_{\text{suf}} \rangle$ for $i = 1, \ldots, n$ takes $O(n)$ time by Lemma 2.2. Building a complete heap on top of these $n$ pairs, see Figure 2.1, takes $O(n)$ time. By Lemma 2.3 and Theorem 2.1 the result follows. □

## 2.6 Extension to Higher Dimensions

In this section we use the optimal algorithm for the one-dimensional $k$ maximum sums problem to design algorithms solving the problem in $d$ dimensions for any natural number $d$. We start by designing an algorithm for the $k$ maximal sums problem in two dimensions, which is then extended to an algorithm solving the problem in $d$ dimensions for any $d$.

**Theorem 2.3** *There exists an algorithm for the two-dimensional $k$ maximal sums problem, where the input is an $m \times n$ matrix, using $O(m^2 \cdot n + k)$ time and space with $m \leq n$.*

*Proof.* Without loss of generality assume that $m$ is the number of rows and $n$ the number of columns. This algorithm uses the reduction to the one-dimensional case mentioned in the introduction by constructing $\binom{m}{2} + m$ one-dimensional problems. For all $i, j$ with $1 \leq i \leq j \leq m$ we take the sub-matrix consisting of the rows from $i$ to $j$ and sum each column into a single entrance of an array. The array containing the rows from $i$ to $j$ can be constructed in $O(n)$ time from the array containing the rows from $i$ to $j-1$. Therefore, we for each $i = 1, \ldots, m$ construct the arrays containing rows from $i$ to $j$ for $j = i, \ldots, m$ in this order.

For each one-dimensional instance we construct the $n$ heaps $H^j_{\text{suf}}$. These heaps are then merged into one big heap by adding nodes with $\infty$ keys, by the same construction used in the one-dimensional algorithm, and use the heap selection algorithm to extract the result. This gives $(\binom{m}{2} + m) \cdot (n-1) + \binom{m}{2} + m - 1$ extra values equal to $\infty$.

It takes $O(n)$ time to build the $H^j_{\text{suf}}$ heaps for each of the $\binom{m}{2} + m$ one-dimensional instances and $O(m^2 \cdot n + k)$ time to do the final selection. □

The above algorithm is naturally extended to an algorithm for the $d$-dimensional $k$ maximum sums problem, for any constant $d$. The input is a $d$-dimensional matrix $A$ of size $n_1 \times n_2 \times \cdots \times n_d$.

**Theorem 2.4** *There exists an algorithm solving the $d$-dimensional $k$ maximal sums problem using $O(n_1 \cdot \prod_{i=2}^{d} {n_i}^2)$ time and space.*

*Proof.* The dimension reduction works for any dimension $d$, *i.e.* we can reduce an $d$-dimensional instance to $\binom{n_d}{2} + n_d$ instances of dimension $d-1$. We iteratively use this dimension reduction, reducing the problem to one-dimensional instances. Let $A^{i,j}$ be the $d-1$-dimensional matrix, with size $n_1 \times n_2 \times \cdots \times n_{d-1}$ and $A^{i,j}[i_1] \cdots [i_{d-1}] = \sum_{s=i}^{j} A[i_1] \cdots [i_{d-1}][s]$.

We obtain the following incremental construction of a $d-1$-dimensional instance in the dimension reduction, $A^{i,j} = A^{i,j-1} + A^{j,j}$. Therefore, we can build each of the $\binom{n_d}{2} + n_d$ instances of dimension $d-1$ by adding $\prod_{i=1}^{d-1} n_i$ values to the previous instance. The time for constructing all these instances is bounded by:

$$\begin{aligned} T(1) &= 1 \\ T(d) &= \left(\binom{n_d}{2} + n_d\right) \cdot \left(T(d-1) + \prod_{i=1}^{d-1} n_i\right) , \end{aligned}$$

which solves to $O(n_1 \cdot \prod_{i=2}^{d} {n_i}^2)$ for $n_i \geq 2$ and $i = 1, \ldots, d$. This adds up to $\prod_{i=2}^{d}(\binom{n_i}{2} + n_i) = O(\prod_{i=2}^{d} {n_i}^2)$ one-dimensional instances in total. For each one-dimensional instance the $n_1$ heaps, $H^j_{\text{suf}}$, are constructed. All heaps are assembled into one complete heap using $n_1 \cdot \prod_{i=2}^{d} \left(\binom{n_i}{2} + n_i\right) - 1$ infinity keys ($\infty$) and heap selection is used to find the $k$ largest sums. □

## 2.7 Space Reduction

In this section we explain how to reduce the space usage of our linear time algorithm from $O(n+k)$ to $O(k)$. This bound is optimal in the sense that at least $k$ values must be stored as output.

**Theorem 2.5** *There exists an algorithm solving the $k$ maximal sums problem using $O(n+k)$ time and $O(k)$ space.*

*Proof.* The original algorithm uses $O(n+k)$ space. Therefore, we only need to consider the case where $k \leq n$. Instead of building all $n$ heaps at once, only $k$ heaps are built at a time. We start by building the $k$ first heaps, $H^1_{\text{suf}}$ ,$\ldots$,$H^k_{\text{suf}}$, and find the $k$ largest sums from these heaps using heap selection as in the original algorithm. These elements are then inserted into an *applicant set*. Then all the heaps except the last one are deleted. This is because the last heap is needed to build the next $k$ heaps. Remember the incremental construction of $H^{j+1}_{\text{suf}}$ from $H^j_{\text{suf}}$ defined in equation (2.3) based on a partial persistent Iheap.

We then build the next $k$ heaps and find the $k$ largest elements as before. These elements are merged with the applicant set and the $k$ smallest are deleted

using selection [19]. This is repeated until all $H^j_{\text{suf}}$ heaps have been processed. The space usage of the last heap grows by $O(k)$ in each iteration, ruining the space bound if it is reused. To remedy this, we after each iteration find the $k$ largest elements in the last heap and build a new Iheap with these elements using repeated insertion. The old heap is then discarded. Only the $k$ largest elements in the last heap can be of interest for the suffix sums not yet constructed, thus the algorithm remains correct.

At any time during this algorithm we store an applicant set with $k$ elements and $k$ heaps which in total contains $O(k)$ elements. The time bound remains the same since there are $O(\frac{n}{k})$ iterations each performed in $O(k)$ time. □

In the case where $k = 1$, it is worth noticing the resemblance between the algorithm just described and the optimal algorithm of Jay Kadane described in the introduction. At all times we remember the best sub-vector seen so far. This is the single element residing in the applicant set. In each iteration we scan one entrance more of the input array and find the best suffix of the currently scanned part of the input array. Because of the rebuilding only two suffixes are constructed in each iteration and only the best suffix is kept for the next iteration. We then update the best sub-vector seen so far by updating the applicant set. In these terms with $k = 1$ our algorithm and the algorithm of Kadane are the same and for $k > 1$ our algorithm can be seen as a natural extension of it.

The original algorithm solving the two-dimensional version of the problem requires $O(m^2 \cdot n + k)$ space. Using the same ideas as above, we design an algorithm for the two-dimensional $k$ maximal sums problem using $O(m^2 \cdot n + k)$ time and $O(n + k)$ space.

**Theorem 2.6** *There exists an algorithm for the $k$ maximal sums problem in two dimensions using $O(m^2 \cdot n + k)$ time where $m \leq n$ and $O(n + k)$ additional space.*

*Proof.* Using $O(n)$ space for a single array we iterate through all $\binom{m}{2} + m$ one-dimensional instances in the standard reduction creating each new instance from the last one in $O(n)$ time. We only store in memory $\lceil \frac{k}{n} \rceil$ instances at a time.

We start by finding the $k$ largest sub-vectors from the first $\lceil \frac{k}{n} \rceil$ instances by concatenating them into a single one-dimensional instance separated by $-\infty$ values and use our one-dimensional algorithm. No returned sum will contain values from different instances because that would imply that the sum also included a $-\infty$ value. The $k$ largest sums are saved in the applicant set. We then repeatedly find the $k$ largest from the next $\lceil \frac{k}{n} \rceil$ instances in the same way and update the applicant set in $O(k)$ time using selection. When all instances have been processed the applicant set is returned.

If $k \leq n$ we only consider one instance in each iteration. The $k$ largest sums from this instance is found and the applicant set is updated. This can all be done in $O(n + k) = O(n)$ time using the linear algorithm for the one-dimensional problem and standard selection. There are $\binom{m}{2} + m$ iterations resulting in an $O(m^2 \cdot n) = O(m^2 \cdot n + k)$ time algorithm.

If $k > n$ each iteration considers $\lceil \frac{k}{n} \rceil \geq 2$ instances. These instances are concatenated using $\lceil \frac{k}{n} \rceil - 1$ extra space for the $\infty$ values. The $k$ largest sums from these instances are found from the concatenated instance using the linear one-dimensional algorithm in $O((\lceil \frac{k}{n} \rceil \cdot n) + k) = O(k)$ time. The number of iterations is $(\binom{m}{2} + m)/\lceil \frac{k}{n} \rceil \leq (\binom{m}{2} + m) \cdot \frac{n}{k}$, leading to an $O(m^2 \cdot n + k)$ time algorithm.

For both cases the additional space usage is at most $O(n + k)$ at any point during the iteration since only the applicant set, $\lceil \frac{k}{n} \rceil$ instances, and $\lceil \frac{k}{n} \rceil - 1$ dummy values are stored in memory at any one time. □ The above

algorithm is extended naturally to solve the problem for $d$ dimensional inputs of size $n_1 \times n_2 \times \cdots \times n_d$.

**Theorem 2.7** *There exists an algorithm solving the d-dimensional k maximal sums problem using $O(n_1 \cdot \prod_{i=2}^{d} {n_i}^2)$ time and $O(\prod_{i=1}^{d-1} n_i + k)$ additional space.*

*Proof.* As in Theorem 2.4, we apply the dimension reduction repeatedly, using $d - 1$ vectors of dimension $1, 2, \ldots, d - 1$ respectively, to iteratively construct each of the $\prod_{i=2}^{d}(\binom{n_i}{2} + n_i) = O(\prod_{i=2}^{d} {n_i}^2)$ one-dimensional instances. Every time a $d-1$-dimensional instance is created we recursively solve it. Again only $\lceil \frac{k}{n_1} \rceil$ one-dimensional instances and the applicant set is kept in memory at any one time and the algorithm proceeds as in the two-dimensional case. The space required for the arrays is $\sum_{i=1}^{d-1} \prod_{j=1}^{i} n_j = O(\prod_{i=1}^{d-1} n_i)$ with $n_i \geq 2$ for all $i$. □

# Chapter 3

## Selecting Sums in Arrays

A generalization of the maximal sum problem is to restrict the length of the subarrays considered. In [55] Huang describes an $O(n)$ time algorithm locating the largest sum of length at least $l$, while in [69] an $O(n)$ time algorithm locating the largest sum of length at most $u$ is described. The algorithms can be combined into at linear time algorithm finding the largest sum of length at least $l$ and at most $u$ [69]. In [35] it is shown how to solve the problem in $O(n)$ time when the input elements are given online one by one.

The *length constrained k maximal sums* problem is defined as follows. Given an array $A$ of length $n$, find the $k$ largest sums consisting of at least $l$ and at most $u$ numbers. Lin and Lee solved the problem using a randomized algorithm with an expected running time of $O(n \log(u-l) + k)$ [68]. Their algorithm is based on a randomized algorithm that selects the $k$th largest length constrained sum from an array in $O(n \log(u-l))$ expected time. The authors state as an open problem whether this is optimal. Furthermore, in [67] Lin and Lee describe a deterministic $O(n \log n)$ time algorithm that selects the $k$th largest sum in an array of size $n$. They propose as an open problem whether this bound is tight. This problem is known as the *sum selection* problem.

**Our Contribution** In this chapter we settle the time complexity for the sum selection problem and the length constrained $k$ maximal sums problem. First, we describe an optimal $O(n+k)$ time algorithm for the length constrained $k$ maximal sums problem in Section 3.1. This algorithm is an extension of our optimal algorithm solving the $k$ maximal sums problem from [D2]. Secondly, we prove a time bound of $\Theta(n \log(k/n))$ for the sum selection problem in Section 3.2. An $O(n \log(k/n))$ time algorithm that selects the $k$th largest sum is described in Section 3.2.2, and in Section 3.2.4 we prove a matching lower bound using a reduction from the cartesian sum problem [42]. Finally, in Section 3.3 we combine the ideas from the two algorithms we have designed and obtain an $O(n \log(k/n))$ time algorithm that selects the $k$th largest sum among all sums consisting of at least $l$ and at most $u$ numbers. This bound is always as good as the previous randomized bound of $O(n \log(u-l))$ by Lin and Lee [68], since there are $\sum_{t=l}^{u}(n-t+1) \leq n(u-l+1)$ subarrays of length between $l$ and $u$ in an array of size $n$ and thus $k/n \leq u-l+1$. The results are summarized in Table 3.1.

| Problem | Previous Work | Our Contribution |
|---|---|---|
| Length Constrained $k$ Maximal Sums | $O(n \log(u-l)+k)$ exp. [68] | $O(n+k)$ |
| Sum Selection | $O(n \log n)$ [67] | $\Theta(n \log(k/n))$ |
| Length Constrained Sum Selection | $O(n \log(u-l))$ exp. [68] | $O(n \log(k/n))$ |

Table 3.1: Overview of results on length constrained $k$ maximal sums and sum selection.

## 3.1 The Length Constrained $k$ Maximal Sums Problem

In this section we present an optimal $O(n+k)$ time algorithm that reports the $k$ largest sums of an array $A$ of length $n$ with the restriction that each sum is an aggregate of at least $l$ and at most $u$ numbers. We reuse the idea from the $k$ maximal sums algorithm in [D2], and construct a heap[1] that implicitly represents all $\sum_{t=l}^{u} n-t+1 = O(n(u-l))$ valid sums from the input array using only $O(n)$ time and space. The $k$ largest sums are then retrieved from the heap using Fredericksons heap selection algorithm [41] that extracts the $k$ largest elements from a heap in $O(k)$ time. We note that the $k$ maximal sums algorithm from [D2] can be altered to use a heap supporting deletions to obtain an $O(n \log(u-l)+k)$ algorithm solving the problem without randomization. The difference between our new $O(n+k)$ time algorithm and the algorithm solving the $k$ maximal sums problem [D2] is in the way the sums are grouped in heaps. This change enables us to solve the problem without deleting elements from a heap. In the following we assume that $l < u$. If $l = u$ the problem can be solved in $O(n)$ time using a linear time selection algorithm [19].

### 3.1.1 A Linear Time Algorithm

For each array index $j$, for $j = 1, \ldots, n-l+1$, we build data structures representing all sums of length between $l$ and $u$ ending at index $j+l-1$. This is achieved by constructing all sums ending at $A[j]$ with length between 1 and $u-l+1$, and then adding the sum of the $l-1$ elements, $A[j+1, \ldots, j+l-1]$, following $A[j]$ in the input array to each sum. To construct these data structures efficiently, the input array is divided into *slabs* of $w = u-l$ consecutive elements, and the sums are grouped in disjoint sets, $\hat{Q}_j$ and $\bar{Q}_j$ for $j = 1, \ldots, n$, depending on the slab boundaries.

Let $a$ be the first index in the slab containing index $j$, i.e. $a = 1 + \left\lfloor \frac{j-1}{w} \right\rfloor w$. The set $\hat{Q}_j$ contains all sums of length between $l$ and $u$ ending at index $j+l-1$ that start in the slab containing index $j$ and is defined as follows:

$$\hat{Q}_j \quad = \quad \{(i, j+l-1, sum) \mid a \leq i \leq j,\ sum = c + \textstyle\sum_{t=i}^{j} A[t]\}\ ,$$

[1] For simplicity of exposition, by heap we denote a heap ordered binary tree where the largest element is placed at the root.

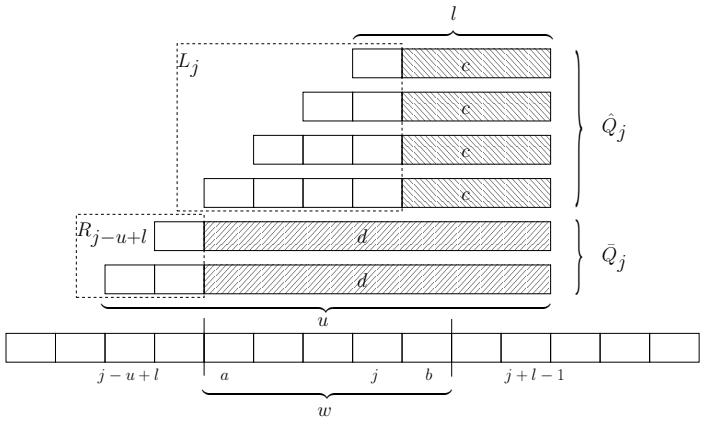



Figure 3.1: Overview of the sets, $l = 4, u = 9, c = \sum_{t=j+1}^{j+l-1} A[t]$ and $d = \sum_{t=a}^{j+l-1} A[t]$. A new slab begins at index $a$ and ends at index $b$.

where $c = \sum_{t=j+1}^{j+l-1} A[t]$ is the sum of $l-1$ numbers in $A[j+1, \ldots, j+l-1]$. The set $\bar{Q}_j$ contains the $(u-l+1)-(j-a+1) = u-l-j+a$ valid sums ending at index $j+l-1$ that start to the left of index $a$, thus:

$$\bar{Q}_j = \{(i, j+l-1, sum) \mid j-u+l \leq i < a,\ sum = d + \textstyle\sum_{t=i}^{a-1} A[t]\}\ ,$$

where $d = \sum_{t=a}^{j+l-1} A[t]$ is the result of summing the $j-a+l$ numbers in $A[a, \ldots, j+l-1]$. The sets are illustrated in Figure 3.1.

By construction, the sets $\hat{Q}_j$ and $\bar{Q}_j$ are disjoint and their union is the $u-l+1$ sums of length between $l$ and $u$ ending at index $j+l-1$.

With the sets of sums defined we continue with the representation of these. The sets $\hat{Q}_j$ and $\bar{Q}_j$ are represented by pairs $\langle \hat{\delta}_j, \hat{H}_j \rangle$ and $\langle \bar{\delta}_j, \bar{H}_j \rangle$ where $\hat{H}_j$ and $\bar{H}_j$ are partially persistent heaps and $\hat{\delta}_j$ and $\bar{\delta}_j$ are constants that must be added to all elements in $\hat{H}_j$ and $\bar{H}_j$ respectively to obtain the correct sums. For the heaps we use the Iheap from [D2] which supports insertions in amortized constant time. Partial persistence is implemented using the node copying technique [33].

We construct representations of two sequences of sets, $L_j$ and $R_j$ for $j = 1, \ldots, n$, that depend on the slab boundaries. Consider the slab $A[a, \ldots, j, \ldots, b]$ containing index $j$. The set $L_j$ contains the $j-a+1$ sums ending at $A[j]$ that start between index $a$ and $j$. The set $R_j$ contains the $b-j+1$ sums ending at $A[b]$ starting between index $j$ and $b$, see Figure 3.1.

Each set $L_j$ is represented as a pair $\langle \delta_j^L, H_j^L \rangle$ where $\delta_j^L$ is an additive constant as above and $H_j^L$ is a partially persistent Iheap. The pairs are incrementally constructed while scanning the input array from left to right as follows:

$$\begin{aligned} \langle \delta_a^L, H_a^L \rangle &= \langle A[a], \{0\} \rangle \wedge \\ \langle \delta_j^L, H_j^L \rangle &= \langle \delta_{j-1}^L + A[j], H_{j-1}^L \cup \{-\delta_{j-1}^L\} \rangle\ . \end{aligned}$$

These are also the construction equations used in [D2]. Constructing a representation of $L_a$ is simple, and creating a representation for $L_j$ can be done efficiently given a representation of $L_{j-1}$. The representation of $L_j$ is constructed

by implicitly adding $A[j]$ to all elements from $L_{j-1}$ by setting $\delta_j^L = \delta_{j-1}^L + A[j]$ and inserting an element to represent the sum $A[j]$. Since $\delta_{j-1}^L + A[j]$ needs to be added to all elements in the representation of $L_j$, an element with $-\delta_{j-1}^L$ as key is inserted into $H_{j-1}^L$, yielding $H_j^L$ ending the construction. Partial persistence ensures that the Iheap $H_{j-1}^L$ used to represent $L_{j-1}$ is not destroyed. By the above description and the cost of applying the node copying technique [33] the amortized time needed to construct a pair is $O(1)$.

The $R_j$ sets are represented by partially persistent Iheaps $H_j^R$, and these representations are built by scanning the input array from right to left. We get the following incremental construction equations:

$$\begin{aligned} H_b^R &= \{A[b]\} \wedge \\ H_j^R &= H_{j+1}^R \cup \{\textstyle\sum_{t=j}^{b} A[t]\} \ . \end{aligned}$$

Similar to the $\langle \delta_j^L, H_j^L \rangle$ pairs, constructing a partial persistent Iheap $H_j^R$ also takes $O(1)$ time amortized. Therefore, the time needed to build the representation of the $2n$ sets $L_j$ and $R_j$ for $j = 1, \ldots, n$ is $O(n)$.

We represent the sets $\hat{Q}_j$ and $\bar{Q}_j$ using the representations of the sets $L_j$ and $R_{j-u+l}$. Figure 3.1 illustrates the correspondence between $\hat{Q}_j$ and $L_j$ and $\bar{Q}_j$ and $R_{j-u+l}$. Consider any index $j \in \{1, \ldots, n-l+1\}$, and let $A[a, \ldots, j, \ldots, b]$ be the current slab containing $j$. The set $\hat{Q}_j$ contains the sums of length between $l$ and $u$ that start in the current slab and end at index $j+l-1$. The set $L_j$ contains the $j-a+1$ sums that start in the current slab and end at $A[j]$. Therefore, adding the sum of the $l-1$ numbers in $A[j+1, \ldots, j+l-1]$ to each element in $L_j$ gives $\hat{Q}_j$ and thus:

$$\hat{Q}_j = \langle c + \delta_j^L, H_j^L \rangle \ ,$$

where $c = \sum_{t=j+1}^{j+l-1} A[t]$.

Similarly, the set $\bar{Q}_j$ contains the $u-l+1-(j-a+1) = u-l-j+a$ sums of length between $l$ and $u$ ending at $A[j+l-1]$ starting in the previous slab. The set $R_{j-u+l}$ contains the $u-l-j-a$ shortest sums ending at the last index in the previous slab. Therefore, adding the sum of the $j+l-a$ numbers in $A[a, \ldots, j+l-1]$ to each element in $R_{j-u+l}$ gives $\bar{Q}_j$ and thus:

$$\bar{Q}_j = \langle d, H_{j-u+l}^R \rangle \ ,$$

where $d = \sum_{t=a}^{j+l-1} A[t]$.

**Lemma 3.1** *Constructing the $2(n-l+1)$ pairs that represent $\hat{Q}_j$ and $\bar{Q}_j$ for $j = 1, \ldots, n-l+1$ takes $O(n)$ time.*

*Proof.* Constructing all $\langle \delta_j^L, H_j^L \rangle$ pairs and all $H_j^R$ partial persistent Iheaps takes $O(n)$ time, and calculating sums $c$ and $d$ takes constant time using a prefix array. Constructing the prefix array takes $O(n)$ time. Therefore, constructing $\hat{Q}_j$ and $\bar{Q}_j$ for $j = 1, \ldots, n-l+1$ takes $O(n)$ time. □ After constructing the $2(n-l+1)$ pairs, they are assembled into one large heap using $2(n-l+1)-1$

dummy $\infty$ keys as in [D2]. The largest $2(n-l+1)-1+k$ elements are then extracted from the assembled heap in $O(n+k)$ time using Fredericksons heap selection algorithm. The implicit sums given by adding $\delta$ values are explicitly computed while Fredericksons algorithm explores the final heap top down in the way described in [D2]. The $2(n-l+1)-1$ dummy elements are discarded.

**Theorem 3.1** *The algorithm described reports the $k$ largest sums with length between $l$ and $u$ in an array of length $n$ in $O(n+k)$ time.*

## 3.2 Sum Selection Problem

In this section we prove an $\Theta(n\log(k/n))$ time bound for the sum selection problem by designing an $O(n\log(k/n))$ time algorithm that selects the $k$th largest sum in an array of size $n$ and by proving a matching lower bound.

The idea of the algorithm is to reduce the problem to selection in a collection of sorted arrays and weight balanced search trees [10, 72]. The trees and the sorted arrays are constructed using the ideas from Section 3.1 and [D2]. Selecting the $k$th largest element from a set of trees and sorted arrays is done using an essential part of the sorted column matrix selection algorithm of Frederickson and Johnson [42]. The part of Frederickson and Johnsons algorithm that we use is an iterative procedure named *Reduce*. In a round of the Reduce algorithm each array, $A$, is represented by the $1+\lfloor\alpha|A|\rfloor$ largest element stored in the array, and a constant fraction of the elements in each array may be eliminated. This can be approximated in weight balanced search trees and the complexity analysis from [42] remains valid.

The lower bound is proved using a reduction from the $X+Y$ cartesian sum selection problem [42].

We note that if $k \leq n$ then the $k$ maximal sums algorithm from [D2] can be used to solve the problem optimally in $O(n)$ time.

To construct the sorted arrays efficiently, we use a heap data structure, that is a generalization of the Iheap, which we name *Bheap*. The Bheap is a heap ordered binary tree where each node of the tree contains a sorted array of size $B$. By heap order, we mean that all elements in a child of a node $v$ must be smaller than the smallest element stored in $v$. Sorted arrays of $B$ elements are required to be inserted in $O(B)$ time amortized. Our Bheap implementation is based on ideas from the functional random access lists in [73] and simple bubble up/down procedures based on merging sorted arrays.

### 3.2.1 The Bheap

To construct the sorted arrays efficiently, we use a heap data structure, that is a generalization of the Iheap, which we name *Bheap*. The Bheap is a heap ordered binary tree where each node of the tree contains a sorted array of size $B$. By heap order, we mean that all elements in a child of a node $v$ must be smaller than the smallest element stored in $v$. Sorted arrays of $B$ elements are required to be inserted in $O(B)$ time amortized. Our Bheap implementation is

based on ideas from the functional random access lists in [73] and simple bubble up/down procedures based on merging sorted arrays.

**Merging Complete Heaps** The main operation used by the Bheap is an algorithm for merging two complete heaps $H_1, H_2$ each containing $(2^i-1)$ nodes, using a node $v$ storing $B$ sorted elements. The operation works as follows. $H_1, H_2$ are set as the children of $v$. Let the roots of $H_1, H_2$ be $R_1, R_2$ and let $s_1, s_2$ be the smallest element of $R_1, R_2$ respectively. Assume without loss of generality, that $s_1 \leq s_2$. The algorithm merges the elements in $R_1, R_2$, and $v$ into one sorted sequence $S$. The largest $B$ elements are put in $v$, the next $B$ elements are put into $R_1$, and the remaining $B$ are put into $R_2$.

Since $s_1 \leq s_2$, there are at least $2B$ elements at least as large as $s_1$ in $R_1 \cup R_2$, and therefore the smallest element from $S$ put in $R_1$ is greater than or equal to $s_1$. This means that the binary tree rooted at $R_1$, is still heap ordered. The root clearly contains the largest elements, thus only the heap order of the subtree rooted at $R_2$ needs to be restored which is done recursively.

**Lemma 3.2** *Merging two complete heaps of size $2^i-1$ using a block of $B$ sorted elements takes $O(iB)$ time.*

*Proof.* The algorithm merges 3 sorted arrays of size $B$ at most $i$ times □

**Structure of the Bheap** The data structure is a tree, where the rightmost path is a spine. The root is given rank 0, and any other node on the spine has rank equal to 1 plus rank of its parent. Let $v_i$ denote the node on the spine with rank $i$. If $v_i$ has a left child, then it is a complete heap with $2^i - 1$ nodes or $2^{i+1} - 1$ nodes. The latter can occur if $v_{i+1}$, the right child of $v_i$. also has a left child of size $2^{i+1} - 1$. Remark that the root only has a left child if $v_1$ has a left child.

Inserting a sorted array $b$ of size $B$ is done as follows. A new node $v$ containing $b$ is created. If the Bheap does not contain any nodes, $v$ becomes the root. If the Bheap only contains one element, $v$ is set as right child of the root. A bubble-up process starting from $v$ is then initiated. The elements stored in $v$ are merged with the elements stored in $v$'s parent, $p(v)$. The largest elements are put in $p(v)$ and the remaining in $v$. The process is then repeated in $p(v)$ and stops when it has reached the root. If the last two nodes on the spine both have left children and they contain the same number of elements, the spine is extended by one by inserting $v$ at the end. The bubble-up procedure is then invoked on $v$. If there are two nodes $v_i, v_{i+1}$ that both have left children storing $2^{i+1} - 1$ nodes, these two complete heaps are merged into a complete heap $h$ of size $2^{i+2} - 1$ using $v$. The resulting complete heap is then inserted as a left child of $v_{i+2}$ if this node does not already have a left child. If $v_{i+2}$ does have a left child it is set as the left child of $v_{i+1}$. The bubble-up procedure is then invoked on the root of $h$. Otherwise, $v$ is inserted as a left child of $v_1$ if $v_1$ does not have a left child, and as a left child of the root otherwise. Again the bubble-up procedure is invoked on $v$ to restore heap order.

**Lemma 3.3** *Inserting $B$ sorted elements into the Bheap takes $O(B)$ time amortized*

*Proof.* Two complete heaps of size $2^i - 1$ are merged every $2^{i+1} - 1$ insertions in $O(iB)$ time. The bubble-up procedure performed after the merging also takes $O(iB)$ time. The spine is extended by one to length $i + 1$ one insertion before two heaps of size $2^{i+1} - 1$ are merged for the first time, and it takes $O(iB)$ time. By summing over all insertions the amortized cost becomes $O(B)$. □

### 3.2.2 An $O(n \log(k/n))$ Time Algorithm

In this section we reduce the sum selection problem to selection in a set of trees and sorted arrays. We use the weight balanced B-trees of Arge and Vitter [10] with degree $B = O(1)$. Similar to the grouping of sums in Section 3.1, each index $j$, for $j = 1, \ldots, n$, is associated with data structures representing all possible sums ending at $A[j]$. The set representing all sums ending at index $j$ is defined as follows:

$$Q_j = \left\{(i, j, sum) \mid 1 \leq i \leq j,\ sum = \textstyle\sum_{t=i}^{j} A[t]\right\} .$$

The input array is divided into slabs of size $w = \lceil k/n \rceil$, and the set $Q_j$ is represented by two disjoint sets $WB_j$ and $BH_j$ that depend on the slab boundaries. The set $WB_j$ contains the sums ending at index $j$ beginning in the current slab, and $BH_j$ contains the sums ending at index $j$ not beginning in the current slab. Let $a = 1 + \left\lfloor \frac{j-1}{w} \right\rfloor w$, i.e. the first index in the slab containing index $j$, then:

$$\begin{aligned} WB_j &= \left\{(i, j, sum) \mid a \leq i \leq j,\ sum = \textstyle\sum_{t=i}^{j} A[t]\right\} \wedge \\ BH_j &= \left\{(i, j, sum) \mid 1 \leq i < a,\ sum = c + \textstyle\sum_{t=i}^{a-1} A[t]\right\} , \end{aligned}$$

where $c = \sum_{t=a}^{j} A[t]$ is the sum of the $j - a + 1$ numbers in $A[a \ldots j]$. The sets $WB_j$ and $BH_j$ are disjoint, and $WB_j \cup BH_j = Q_j$ by construction. The sets are illustrated in Figure 3.2.

The set $WB_j$ is represented as a pair $\langle \tau_j, T_j \rangle$ where $T_j$ is a partial persistent weight balanced B-tree and $\tau_j$ is an additive constant that must be added to all elements in $T_j$ to obtain the correct sums. The set $BH_j$ is represented as a pair $\langle \delta_j, H_j \rangle$ where $\delta_j$ is an additive constant and $H_j$ is a partial persistent Bheap with $B = w$.

The pairs $\langle \tau_j, T_j \rangle$ are constructed as follows. If $j$ is the first index of a slab, i.e. $j = 1 + tw$ for some natural number $t$, then:

$$\langle \tau_j, T_j \rangle = \langle A[j], \{0\} \rangle .$$

This is the start of a new slab, and a new partial persistent weight balanced B-tree representing $A[j]$, the first element in the slab, is created. If $j$ is not the first index in a slab then:

$$\langle \tau_j, T_j \rangle = \langle \tau_{j-1} + A[j], T_{j-1} \cup \{-\tau_{j-1}\} \rangle ,$$

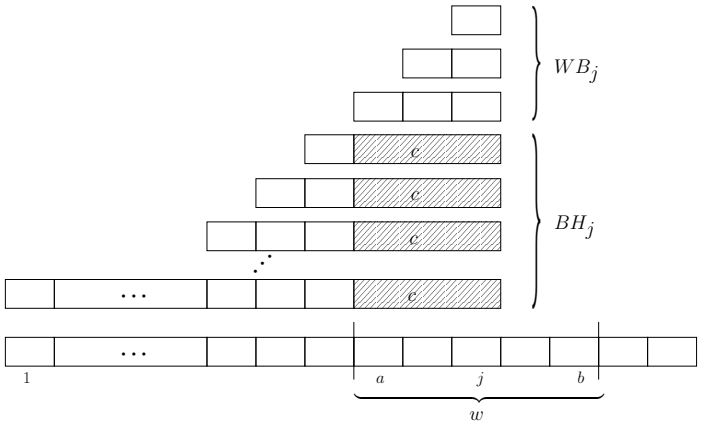



Figure 3.2: Overview of the sets. Slab size $w = 5$, and $A[a, \ldots, b]$ is the slab containing index $j$.

i.e. we change the additive constant and insert $-\tau_{j-1}$ into the weight balanced tree $T_{j-1}$. These construction equations are identical to the construction equations from Section 3.1, and partial persistence ensures that $T_{j-1}$ is not destroyed by constructing $T_j$.

For the $\langle \delta_j, H_j \rangle$ pairs representing the sets $\hat{Q}_j$, we observe that if $j \leq w$ then $BH_j = \emptyset$, thus:

$$\langle \delta_j, H_j \rangle = \langle 0, \emptyset \rangle \ .$$

If $j > w$ and $j$ is not the first index in a slab, then adding $A[j]$ to all elements from the previous set yields the new set, thus:

$$\langle \delta_j, H_j \rangle = \langle \delta_{j-1} + A[j], H_{j-1} \rangle \quad .$$

If $j$ is the first index of a slab, i.e. $j = 1 + tw$ for some integer $t \geq 1$, all $w$ sums represented in $\langle \tau_{j-1}, T_{j-1} \rangle$ are inserted into a sorted array $S$ and each sum explicitly calculated. This sorted array then contains all sums starting in the previous slab ending at index $j-1$. For each element in $S$ the additive constant $\delta_{j-1}$ is subtracted and $S$ is inserted into the Bheap $H_{j-1}$. The construction equation becomes:

$$\langle \delta_j, H_j \rangle \quad = \quad \langle \delta_{j-1} + A[j], H_{j-1} \cup S \rangle \quad ,$$

where

$$S = \left\{ (i, j, s - \delta_{j-1}) \mid j - w \leq i < j, s = \textstyle\sum_{t=i}^{j-1} A[t] \right\} \ .$$

Again, partial persistence ensures that the previous version of the Bheap, $H_{j-1}$, is not destroyed.

**Lemma 3.4** *Constructing the pairs $\langle \delta_j, H_j \rangle$ and $\langle \tau_j, T_j \rangle$ for $j = 1, \ldots, n$ takes $O(n \log(k/n))$ time.*

*Proof.* The Bheap and the weight balanced B-trees have constant in and out-degree. Therefore, partial persistence can be implemented for both using the node copying technique [33].

For the Bheap, amortized $O(1)$ pointers and arrays are changed per insertion. The extra cost for applying the node copying technique is $O(B) = O(w)$ time amortized per insert operation. Constructing the sorted array $S$ from a weight balanced B-tree takes $O(w)$ time. An insert in a Bheap is only performed every $w$th step, and calculating additive constants in each step takes constant time. Therefore, the time used for constructing all $\langle \delta_j, H_j \rangle$ pairs is $O(n + \frac{n}{w}w) = O(n)$.

Each insert in a weight balanced B-tree is performed on a tree containing at most $w$ elements using $O(\log w)$ time. Therefore, the extra cost of using the node copying technique is $O(\log w)$ time amortized per insert operation. Calculating an additive constant $\tau_j$ takes constant time, thus, constructing all $\langle \tau_j, T_j \rangle$ pairs takes $O(n \log(k/n))$ time. □

After the $n$ pairs, $\langle \delta_i, H_i \rangle$, storing Bheaps are constructed, they are assembled into one large heap in the same way as in Section 3.1. That is, we construct a complete heap on top of the pairs using $n-1$ dummy nodes storing the same array of $w$ dummy $\infty$ elements. We then use Fredericksons heap selection algorithm in the same way as in Section 3.1 where the representative for each node is the smallest element in the sorted array stored in it. Using Fredericksons heap selection algorithm the $2n-1$ nodes with the maximal smallest element and their $2n$ children are extracted. This takes $O(n)$ time and the nodes extracted from the Bheap gives $3n$ sorted arrays by discarding the $n-1$ dummy nodes.

**Lemma 3.5** *The $3n$ nodes found as described above contain the $k$ largest sums contained in the $n$ pairs $\langle \delta_i, H_i \rangle$.*

*Proof.* The $4n-1$ nodes found by the heap selection algorithm forms a connected subtree $T$ of the heap rooted at the root of the heap. Any element $e$ stored in a node $v_e \notin T$ is smaller than all elements stored in any internal node $v \in T$ since, by heap order, $e$ is smaller than the smallest element in the leaf of $T$ that is on the path from $v_e$ to the root. The smallest element in a leaf is smaller than the smallest element in any internal node since the leaf was not picked by the heap selection algorithm. There are $2n-1$ internal nodes in $T$ and $n$ of these does not store dummy elements. Therefore, for each element not residing in $T$ there at least $nw = n\lceil \frac{k}{n} \rceil \geq k$ larger elements in the $3n$ found nodes. □

These $3n$ sorted arrays of size $w$ and the $n$ pairs $\langle \tau_i, T_i \rangle$ storing weight balanced B-trees of size at most $w$ contain at most $4nw = 4n\lceil \frac{k}{n} \rceil \leq 4(k+n)$ sums. The $3n$ arrays and the $n$ weight balanced B-trees are given as input to the adapted sorted column matrix selection algorithm, which extracts the $k$th largest element from these in $O(n \log(k/n))$ time. The fact that the weight balanced B-trees are partially persistent versions of the same tree and contain additive constants is handled by expanding the trees and computing the sums explicitly during the top down traversals performed by the selection algorithm as in Section 3.1 and [D2].

**Theorem 3.2** *The algorithm described selects the $k$th largest sum in an array of size $n$ in $O(n \log(k/n))$ time.*

### 3.2.3 Selection in Weight Balanced Trees

Frederickson and Johnson [42] designed an optimal algorithm for selection in a matrix with sorted columns i.e. an optimal algorithm for selection in a set of sorted arrays. The algorithm is based on two procedures, *CUT* and *REDUCE*, but for the purpose of our algorithm, only the REDUCE algorithm is needed. However, the REDUCE algorithm also need to be applied to trees, not only sorted arrays.

The input to the REDUCE algorithm is $n$ sorted arrays. The $i$th array has size $n_i$ and $N = \sum_{i=1}^{n} n_i \leq 9k/2$. In one iteration of the REDUCE algorithm where $k > N/2$, the $1 + \lfloor \alpha n \rfloor$'th smallest element is found for each array, and in some of the arrays all elements up to and including the $1 + \lfloor \alpha n \rfloor$'th element are discarded. This is all done in constant time per sorted array. The case where $k \leq N/2$ is similar.

The same thing can be done approximately in a weight balanced tree. We use the leaf oriented weight balanced $B$-trees of Arge and Vitter [10], where all leaves are at the same level. There are two weight parameters $b, k \geq 8$.

The weight $w(v)$, of a node $v$ is the number of elements in the leafs of the subtree rooted at $v$. In a weight balanced $B$-tree the following conditions hold:

- All leaves are on the same level (0) and each leaf $u$ has weight $\frac{1}{4}k \leq w(u) \leq k$
- An internal node $v$ on $l$ has, has weight $w(v) \leq b^l k$.
- Except for the root, an internal node $v$ at level $l$ has weight $\frac{1}{4}b^l k \leq w(v)$.
- The root has more than one child

Each node of the tree stores the smallest and the largest element stored in each its children.

In a weight balanced b-tree, an element with rank $\beta n$, where $\alpha_1 = \alpha/c \leq \beta \leq \alpha = \alpha_2$ for some constant $c$, can be located by finding the subtree with the $\beta n$ smallest elements. This subtree is found by traversing the leftmost path in the tree. Eventually the weight of the leftmost child, $v_a$, of the current node, $v$, at level $l$ becomes smaller than $\alpha_2 n$ and the traversal stops. However, $w(v) \geq \alpha_2 n$ since we did not stop at $v$'s parent, and the weight of the root is $n > \alpha_2 n$. Therefore, $w(v_a) \leq \alpha_2 n < w(v)$.

The tree rooted at $v_a$ is only valid if $\alpha_1 n \leq w(v_a) \leq \alpha_2 n$ is satisfied. Therefore, if we set $\alpha_1 = \frac{\alpha_2}{4b}$ then

$$\alpha_1 n = \alpha_2 n/4b \leq b^l k/4b = \frac{1}{4} b^{l-1} k \leq w(v_a) \ ,$$

and $w(v_a)$ is in the correct interval.

If $\alpha_2$ is chosen as a small constant, this takes constant time, since only a constant number of steps is required before $\frac{1}{4}b^l k \leq \alpha_2 n$ when the root has two children and the weight constraints are valid. Discarding all elements with rank at most $\beta n$ is done by discarding the subtree found above, and then apply rebalancing to maintain the tree weight balanced.

Usually when a node, $v$, at level, $l$, needs rebalancing after deleting an element, the weight, $w(v)$, is only one smaller than what is required, i.e. $w(v) = \frac{1}{4}b^l k - 1$. However, in this case an entire subtree is deleted, and the weight $w(v)$ of the node may be as much as $b^{l-1}k$ to small, i.e. $\frac{1}{4}b^l k - b^{l-1}k \leq w(v)$ is the only guarantee on $w(v)$ after the cut of a subtree.

If the weight constraint is violated, i.e. $w(v) < \frac{1}{4}b^l k$, $v$ is fused with its leftmost sibling $s$. Since $s$ satisfies the weight constraint, $\frac{1}{4}b^l k \leq w(s)$. By adding up $w(v)$ and $w(s)$ we get:

$$w(s) + w(v) \leq 2 \cdot \frac{1}{4}b^l k - b^{l-1}k = b^{l-1}k\left(\frac{1}{2}b - 1\right) \geq \frac{1}{4}b^l k\ ,$$

for $b \geq 4$, and the fused node satisfies the lower weight constraint. If the merged node is heavier than $b^l k$ after the merge it is split as usual. The path to the root is then traversed and nodes are fused and split similarly if they are unbalanced.

The REDUCE algorithm uses the linear time weighted selection algorithm of [59] to find a partition $(Q_1, x_i, Q_2)$ of a set $Q$ consistent with the weighted selection of an $M$th weighted element. Let the $j$th element in $Q$ be denoted $q_j$ and have weight $w_j$. The weighed selection algorithm partitions $Q$ into three disjoint sets $(Q_1, Q_2, \{q_i\})$ such that $\sum_{j\in Q_1} w_j < M \leq w_i + \sum_{j\in Q_1} w_j$, where $j \in Q_1$ implies $q_j \leq q_i$ and $j' \in Q_2$ implies $q_i \leq q_{j'}$.

The REDUCE procedure is used iteratively as in [42]. The $n$ weight balanced b-trees $T_1, \ldots, T_n$ has size $n_1, \ldots, n_n$ respectively. Instead of an element with rank $1 + \lfloor \alpha n_i \rfloor$ where $\alpha \leq 1 - \sqrt{1/2 + 1/N}$ is a small constant, an element with rank $\beta_i n_i$ is used where $\alpha_1 \leq \beta_i \leq \alpha_2$ for tree $T_i$ for $i = 1, \ldots, n$ with $\alpha_1 = \frac{\alpha}{4b}$ and $\alpha_2 = \alpha$. When only $O(n)$ elements remain, the $k$th largest is found using linear selection.

Therefore, the analysis of the REDUCE procedure which is presented in Lemma 3 in [42] is done again. Following this analysis, we focus on the case where $k > N/2$. Let $\beta_i n$ be the actual rank of the element $x_i$ used in tree $T_i$ where $\alpha_1 n_i \leq \beta_i n_i \leq \alpha_2 n_i$. First we analyze how many elements the algorithm discards in each round. Let $(Q_1, x_i, Q_2)$ be the weighted selection for the $\alpha_2 N$'th weighed element, where $x_i \in T_i$. From all trees $T_j \in Q_1 \cup \{T_i\}$ all elements ranked smaller as well as the $\beta_j$'th ranked element are discarded, and we get the following bound on the number of elements discarded in an iteration:

$$\beta_i n_i + \sum_{j\in Q_1} \beta_j n_j \geq \alpha_1 \left(n_i + \sum_{j\in Q_1} n_j\right) \geq \alpha_1\alpha_2 N$$

since $n_i + \sum_{j\in Q_1} n_j \geq \alpha_2 N$ by weighed selection and $\beta_j \geq \alpha_1$ for all $j$. Therefore, each round discards a constant fraction of the remaining elements.

The largest element removed by the cut is $x_i$. Therefore, if the number of elements at least as large as $x_i$ after the cut is at least $N/2 > k$, then the $k$th largest element remains. No elements from trees in $Q_2$ are cut. For each tree $T_j \in Q_2$ all elements larger than the $\beta_j n_j$'th element are at least as large as $x_i$ by weighed selection. Clearly this is also true for the remaining elements in $T_i$.

We get the following bound on the number of elements left larger than $x_i$

$$(1-\beta_i)n_i + \sum_{j\in Q_2}(1-\beta_j)n_j \geq (1-\alpha_2)\left(n_i + \sum_{j\in Q_2} n_j\right) \geq (1-\alpha_2)^2 N$$

The last inequality follows from the fact that

$$n_i + \sum_{j\in Q_2} n_j = N - \sum_{j\in Q_1} n_j \geq N - \alpha_2 N = (1-\alpha_2)N$$

since $\sum_{j\in Q_1} n_j < \alpha_2 N$ by weighed selection. Because $(1-\alpha_2)^2 N - 1 \geq N/2$ for $\alpha_2 \leq 1 - \sqrt{1/2 + 1/N}$ we conclude that the $k$th largest element is not deleted in an iteration of REDUCE.

**Theorem 3.3** *Selecting the $k$th largest element from a set of $n \leq k$ weight balanced b-trees and sorted arrays containing $O(k)$ elements in total takes $\Theta(n + n\log k/n)$ time.*

*Proof.* The REDUCE algorithm is run iteratively as in [42] and stops when only $O(n)$ elements remain. In each iteration the trees are handled as described above and the sorted arrays as in [42]. The time bound follows from the above analysis, the fact that $\alpha_1$ is only a constant smaller than $\alpha_2 = \alpha$, and [42]. □

### 3.2.4 Lower Bound

In this section we prove a matching lower bound of $\Omega(n\log(k/n))$ time for the sum selection problem via a reduction from the $X + Y$ cartesian sum selection problem. The lower bound model counts the number of comparions between linear combinations of the input elements.

In the $X + Y$ cartesian sum selection problem as defined in [42], the input is two unsorted arrays $X$ and $Y$ and an integer $k$, and the task is to select the $k$th largest element in the cartesian sum $\{x + y \mid x \in X,\ y \in Y\}$.

Given an instance of the $X + Y$ cartesian sum selection problem, $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_m\}$, and $k$, construct the following array $A$ :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $x_1 - x_2$ | $x_2 - x_3$ | $\cdots$ | $x_{n-1} - x_n$ | $x_n + \infty + y_1$ | $y_2 - y_1$ | $\cdots$ | $y_m - y_{m-1}$ |

where $\infty$ is a number larger than $(n+m)\cdot\max\{|x| \mid x \in X\} \cup \{|y| \mid y \in Y\}$. The sums in $A$ have the following properties:

- A sum ranging from $i$ to $j$ where $i \leq n \leq j$ represents the sum $(\sum_{t=i}^{n-1} A[t]) + x_n + \infty + y_1 + (\sum_{t=n+1}^{j} A[t]) = x_i + y_{j-n+1} + \infty$.
- A sum including $A[n] = x_n + \infty + y_1$ is larger than any sum that does not

There are more sums in the sum selection instance than there are in the $X + Y$ cartesian sum instance since any sum not containing $A[n]$ does not correspond to an element in the cartesian sum. However, the $k$th largest sum

does contain $A[n]$ and corresponds to the $k$'th largest sum in the cartesian sum instance. Therefore, any algorithm that selects the $k$th largest sum in an array can be used to select the $k$th largest element in the cartesian sum.

The lower bound for selecting the $k$th largest element in the cartesian sum $(X+Y)$ is $\Omega(m+p\log(k/p))$ comparisons where $|X|=n, |Y|=m$ with $n \leq m$ and $p=\min\{k,m\}$ [42]. In the reduction the size of the array $A$ is $n+m-1$, which is $\Theta(n+m)=\Theta(m)$, and it can be built in $O(m)$ time.

**Theorem 3.4** *Any algorithm which makes comparison between linear combinations of the input elements only requires $\Omega(n\log(k/n))$ comparisons to select the $k$th largest sum in an array of size $n$.*

## 3.3 Length Constrained Sum Selection

In this section we sketch how to select the $k$th largest sum consisting of at least $l$ and at most $u$ numbers from an array of size $n$ in $O(n\log(k/n))$ time. The algorithm combines the ideas from Section 3.1 and Section 3.2. Similar to Section 3.2 the algorithm works by reducing the problem to selection in a collection of weight balanced search trees and sorted arrays. It should be noted that a deterministic algorithm with running time $O(n\log(u-l))$ can be achieved by using weight balanced B-trees instead of Iheaps in the algorithm from Section 3.1, and using these as input to the adapted sorted column matrix selection algorithm instead of the heap selection algorithm.

To achieve $O(n\log(k/n))$ time, we constrain the lengths of the sums considered and divide the input array into slabs of size $u-l$ as in Section 3.1. Subsequently, we efficiently construct representations of the sets $\hat{Q}_j$ and $\bar{Q}_j$ defined in Section 3.1 using weight balanced trees and Bheaps by subdividing each slab into sub-slabs of size $\lceil \frac{k}{n} \rceil$ as in Section 3.2, recall $k/n \leq u-l+1$. Weight balanced B-trees are used to represent sums residing inside a sub-slab, and Bheaps are used to represent sums covering multiple sub-slabs. The sums are illustrated in Figure 3.3. The Bheaps and the weight balanced B-trees are constructed efficiently as in Section 3.2 using partial persistence.

After the representations of the sets $\hat{Q}_j$ and $\bar{Q}_j$ are constructed, the algorithm continues as in Section 3.2. The sorted arrays storing the $k$ largest sums stored in the Bheaps are extracted using Fredericksons heap selection algorithm. The sorted arrays and the weight balanced B-trees are then given as input to the adapted sorted column matrix selection algorithm that selects the $k$th largest sum.

**Theorem 3.5** *The $k$th largest sum of length between $l$ and $u$ in an array of size $n$ can be selected in $O(n\log(k/n))$ time.*

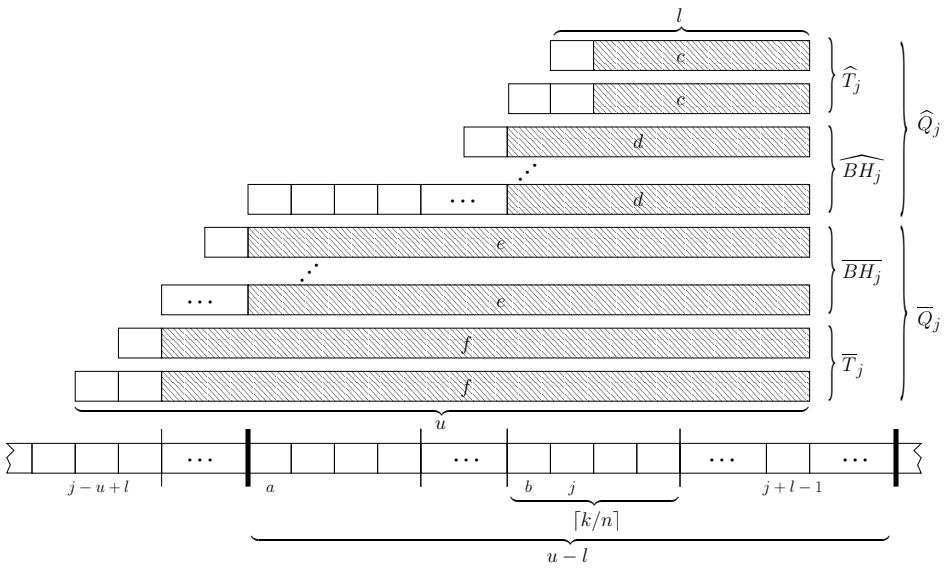



Figure 3.3: Combining ideas - The sums associated with index $j$. A new slab of length $u-l$ starts at index $a$ and a new subslab of length $\lceil k/n \rceil = 4$ starts at index $b$. $c = \sum_{t=j+1}^{j+l-1} A[t]$ , $d = \sum_{t=b}^{j+l-1} A[t]$ , $e = \sum_{t=a}^{j+l-1} A[t]$ and $f = \sum_{t=x}^{j+l-1} A[t]$ where $x$ is the first index in the subslab following the subslab containing index $j-u+l$. The set $\hat{Q}_j$ is split into $\hat{T}_j$, represented by a weight balanced tree, and $\widehat{BH_j}$, represented by a Bheap. The set $\bar{Q}_j$ is split similarly.

# Part II

# Range Queries

# Chapter 4

## Data Structures for Range Median Queries

The median of a set $S$ of size $n$ is an element in $S$ that is larger than $\lfloor \frac{n-1}{2} \rfloor$ other elements from $S$ and smaller than $\lceil \frac{n-1}{2} \rceil$ other elements from $S$. In the range median problem one must preprocess an input array $A$ of size $n$ into a data structure that given indices $i$ and $j$, $1 \leq i \leq j \leq n$, a query must return an index $i'$, $i \leq i' \leq j$, such that $A[i']$ is the median of the elements in the subarray $A[i,j] = [A[i], A[i+1], \ldots, A[j]]$. This range median problem is considered in [45,47,62,78,79]. In the batched case, the input is an array of size $n$ and a set of $k$ queries, $(i_1, j_1), \ldots, (i_k, j_k)$, and the output is the answer to these $k$ queries [50]. Range median queries are naturally generalized to *range selection*, given indices $i, j$ and $s$, return the index of the $s$'th smallest element in $A[i,j]$. A related problem is *range dominance* (or *range rank*) queries, given indices $i, j$ and a value $e$, return the number of elements from $A[i,j]$ that are less than or equal to $e$ (dominated by $e$). This corresponds to 3-sided range counting queries for a set of points.

Previously, the best linear space data structure supported range selection queries in $O(\log n)$ time [45, 47]. In the dynamic case the only known data structure uses $O(n \log n)$ space and supports updates and queries in $O(\log^2 n)$ time [47]. For dominance queries, linear space data structures supporting queries in $O(\log n / \log\log n)$ time is known, as well as a matching lower bound [57, 75, 76]. In the dynamic case [71] describes an $O(n)$ space data structure that supports dominance queries in $O((\log n / \log\log n)^2)$ time and updates in $O(\log^{9/2} n / (\log\log n)^2)$ time. A query lower bound of $\Omega((\log n / \log\log n)^2)$ for data structures with $O(\log^{O(1)} n)$ update time is proved in [75].

**Our Results** In this chapter we use the RAM model of computation with word-size $\Theta(\log n)$. We design a static linear space data structure that supports both range selection and range rank queries in $O(\log n / \log\log n)$ time. This is the best known for range median data structures using $O(n \log^{O(1)} n)$ space, and for range dominance queries this is optimal. Our dynamic data structure uses $O(n \log n / \log\log n)$ space and supports queries and updates in $O((\log n / \log\log n)^2)$ time. For dominance queries this query time is optimal. We prove an $\Omega(\log n / \log\log n)$ time lower bound on range median queries for data structures that can be updated in $O(\log^{O(1)} n)$ time using a reduction from the marked ancestor problem [4], leaving a significant gap to

the achieved upper bound. With our static data structure we improve the $O(n \log k + k \log n)$ time bound for the batched range median problem achieved in [47] to $O(n \log k + k \log n / \log \log n)$ time. If $k > \sqrt{n}$ we construct our static data structure in $O(n \log n) = O(n \log k)$ time and perform $k$ queries. This takes $O(n \log n + k \log n / \log \log n) = O(n \log k + k \log n / \log \log n)$ time. If $k < \sqrt{n}$ then $O(n \log k)$ time is already achieved by [47, 50].

## 4.1 Simple Range Selection Data Structure

In this section we describe the data structure of Gfeller and Sanders [47], which uses linear space and supports queries in $O(\log n)$ time. First, we describe a data structure that uses $O(n \log n)$ space and supports queries in $O(\log n)$ time. Then the space is reduced to $O(n)$ using standard techniques. The main idea is the following. Sort the input elements and place them in the leaves of a binary search tree. Consider a search for the $s$'th smallest element in $A[i, j]$. If the left subtree of the root contains $s$ or more elements from $A[i, j]$ then it contains the $s$'th smallest element from $A[i, j]$. If not, it is in the right subtree. We augment each node of the tree with prefix sums such that the number of elements from $A[1, j]$ contained in the left subtree can be determined for any $j$, and we use fractional cascading [28] to avoid a search for the needed prefix sums in each node.

### 4.1.1 Basic Structure

Let $A = [y_1, \ldots, y_n]$ be the input array. We sort $A$ and build a complete binary search tree $T$ that stores the $n$ elements in the leaves in sorted order. We introduce the following notation. For a node $v$ in $T$, let $T_v$ denote the subtree rooted at $v$, and $|T_v|$ the number of leaves in $T_v$. The *$x$-predecessor* of an index ($x$-coordinate) $i$ in $T_v$ is the largest index $i'$ such that $i' \leq i$ and $y_{i'} \in T_v$. If no such index exists the $x$-predecessor of $i$ is zero. The *$x$-rank* of an index $i$ in $T_v$ is the number of elements from $A[1, i]$ contained in $T_v$. An $x$-rank is essentially a prefix sum. If we know the $x$-rank of $i-1$ and $j$ in $T_v$, we know the number of elements from $A[i, j]$ in $T_v$ since this is the difference between the two. Notice that in $T_v$, the $x$-rank of $j$ and the $x$-rank of the $x$-predecessor of $j$ are equal.

Each node $v$ of $T$ stores two indices for each element $y_i \in T_v$ in an array $A_v$ of size $|T_v|$. Let $y_i \in T_v$ and $r_i$ be the $x$-rank of $i$ in $T_v$. The $r_i$'th pair of indices stored in $A_v$ is the $x$-rank of $i$ in the left subtree, and the $x$-rank of $i$ in the right subtree of $v$. These are fractional cascading indices, meaning that the $x$-rank of $i$ in the left (right) subtree is the position of the indices stored for the $x$-predecessor of $i$ in the left (right) subtree. The structure is illustrated in Figure 4.1.

The arrays are constructed by scanning $A$, starting with $y_1$, and *inserting* the elements, $y_i$, in increasing $i$ order into $T$. For each element $y_i$, the search path to $y_i$ is traversed, and in each visited node $v$ the pair of indices for $y_i$ are appended to $A_v$. The data structure can be built in $O(n \log n)$ time and uses $O(n \log n)$ words of space.

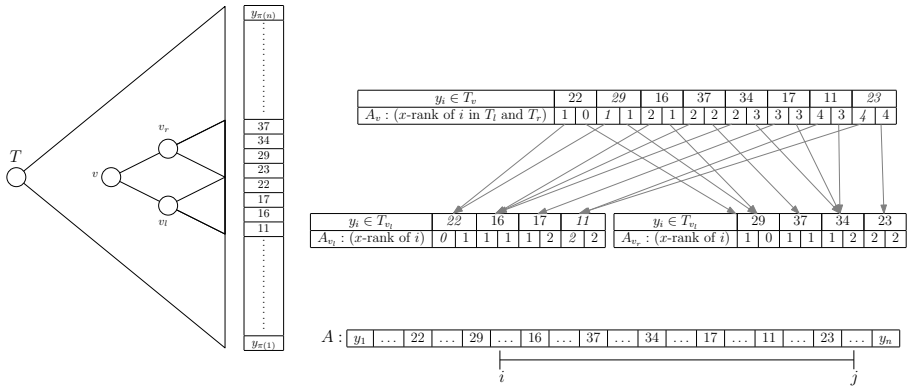



Figure 4.1: Range Selection Data Structure. The position of all elements between 11 and 37 are shown in $A$. The arrows are added to demonstrate how the $x$-ranks work as fractional cascading indices. The search path and $x$-ranks used by a query with indices i,j and s=3 are shown in italic. In $A_v$ the query uses the $x$-ranks defined by the elements 23 and 29, since these are define the $x$-predecessor of $i-1$ and $j$ in $T_v$ respectively. Since there are $3 = 4-1$ elements from $A[i,j]$ in $T_{v_l}$ the search continues in $T_{v_l}$. In $A_{v_l}$ the $x$-ranks stored for the (potentially new) $x$-predecessors are used. These are the $x$-ranks defined by the elements 11 and 16, since the positions of these in $A$ are the $x$-predecessors of $i-1$ and $j$ in $T_{v_l}$ respectively. The positions of these in $A_{v_l}$ are given by the $x$-ranks (fractional cascading indexes) defined by the elements 23 and 29 in $A_v$.

**Range Selection Query** A range selection query is given two indices $i$ and $j$, and an integer $s$, where $1 \leq i \leq j \leq n$, and must return the $s$'th smallest element in $A[i,j]$. In a node $v$ of $T$ the search is guided using the $x$-ranks stored for the $x$-predecessor of $i-1$ and $j$ in $T_v$. In the root this is the $i-1$'th and $j$'th pair stored in the root's array. By subtracting the $x$-ranks for the left subtree we learn how many elements, $s'$, from $A[i,j]$ the left subtree of $v$ contains. If $s \leq s'$ the search continues in the left subtree. Otherwise, we set $s = s - s'$ and continue the search in the right subtree. Notice that each step learns the $x$-ranks of $i-1$ and $j$ in the children nodes, which are needed to lookup the indices stored for the $x$-predecessors of $i-1$ and $j$ in the subsequent step. A query takes $O(\log n)$ time since each step takes constant time. Given indices $i$ and $j$ in a range median query we return the $s = \lfloor \frac{j-i}{2} \rfloor + 1$'th smallest element in $A[i,j]$. Given indices $i,j$ and a value $e$ in a range rank query, we do a predecessor search for $e$ in $T$: in each step where the search continues to the right child, the number of elements from $A[i,j]$ in the left subtree is computed as above, and these are added up. When a leaf is reached this sum is the rank of $e$ in $A[i,j]$.

### 4.1.2 Getting Linear Space

We reduce the space usage of the data structure to $O(n)$ by replacing the arrays stored in each node by simple rank and select data structures [56] as follows. In each node $v$, the array $A_v$ is partitioned into chunks of size $\log n$. The

last entry of each chunk, i.e. every $\log n$'th entry of $A_v$, is stored as before. For the remaining entries of a chunk, one bit is stored, indicating whether the corresponding element resides in the left or right subtree. These bits are packed in order into one word, which we denote a direction word. This reduces the space to $O(n)$ bits per level of $T$. Even though $v$ no longer stores an $x$-rank for each element in $T_v$, a needed $x$-rank is easily computed from the stored chunks in constant time. Let $r_j$ be the $x$-rank of $j$ in $T_v$, and let $j' = \lfloor j/\log n \rfloor$. The indices stored in the $j'-1$'th chunk yields the $x$-rank, $r_\lambda$, in the left subtree of $y_\lambda \in T_v$. The first $r_j - j' \log n$ bits in the direction word from the $j'$'th chunk determines how many elements from $A[\lambda+1, j]$ that reside in the left subtree. The sum of these is the $x$-rank of $j$ in the left subtree. The latter is computed using complete tabulation. The extra table needed for this uses $O(n)$ additional space.

## 4.2 Improving Query Time

In this section we generalize the data structure from Section 4.1 and obtain a linear space data structure that supports range selection and range counting queries in $O(\log n/\log\log n)$ time. First, we describe a data structure that supports queries in $O(\log n/\log\log n)$ time but uses slightly more than $O(n)$ space. Then we reduce the space to $O(n)$ by generalizing the ideas from Section 4.1.2.

### 4.2.1 Structure

The data structure is a balanced search tree $T$ storing the $n$ elements from $A = [y_1, \ldots, y_n]$ in the leaves in sorted order. The fan-out of $T$ is $f = \lceil \log^\varepsilon n \rceil$ for some constant $0 < \varepsilon < 1$. The height of $T$ is $O(\log n/\log f) = O(\log n/\log\log n)$. Each node $v \in T$ contains $f \cdot |T_v|$ prefix sums: For each element, $y_i \in T_v$, and for each child index, $1 \leq \ell \leq f$, $v$ stores the number of elements from $A[1, i]$ that reside in the first $\ell$ subtrees of $T_v$. We denote by $t^i_\ell$ such a prefix sum. These prefix sums are stored in $|T_v|$ bit-matrices, one matrix $M_i$ for each $y_i \in T_v$. The $\ell$'th row of bits in $M_i$ is the number $t^i_\ell$. The rows form a non-decreasing sequence of numbers by construction. The matrices are stored consecutively in an array $A_v$ as above, i.e. $M_i$ is stored before $M_j$ if $i < j$, and the $x$-rank of $i$ in $T_v$ defines the position of $M_i$ in $A_v$. If $y_j \notin T_v$ then $v$ does not store a matrix $M_j$, but it is still well defined and equal to the matrix $M_{j'}$, that is stored in $v$, where $j'$ is the $x$-predecessor of $j$ in $T_v$. Each matrix is stored in two different ways. In the first copy each row is stored in one word. In the second copy each matrix is divided into sections of $g = \lfloor \log n/f \rfloor$ columns. The first section contains the first $g$ bits of each of the $f$ rows, and these are stored in one word. This is the $g$ most significant bits of each prefix sum stored in the matrix. The second section contains the last three bits of the first section and then the following $g-3$ bits, and so on. The reason for this overlap of three bits will become clear later. We think of each section as an $f \times g$ bit matrix. For technical reasons, we ensure that the first column of each matrix only contain zero entries by prepending a column of zeroes to all

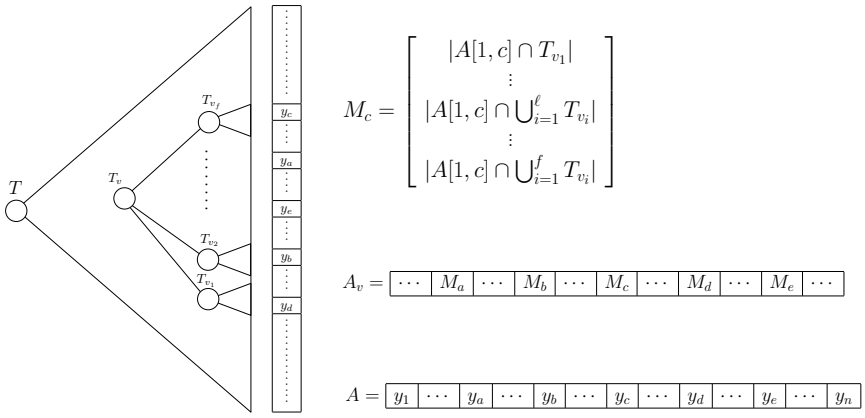



Figure 4.2: A graphic overview of the data structure. As showed in the tree, $y_d < y_b < y_e < y_a < y_c$ and they are all contained in $T_v$. A concrete example of a matrix is shown in Figure 4.3.

matrices before the division into sections. An overview of the data structure is shown in Figure 4.2.

### 4.2.2 Range Selection Query

A query is given indices $i, j$ and $s$ and locates the $s$'th smallest element in $A[i, j]$. In each node we consider the matrix $M' = M_j - M_{i-1}$ (row-wise subtraction). The $\ell$'th row of $M'$ is $t_\ell^j - t_\ell^{i-1}$, i.e. the number of elements from $A[i, j]$ contained in the first $\ell$ subtrees. We compute the smallest $\ell$ such that the $\ell$'th row in $M'$ stores a number greater than or equal to $s$, and this defines the subtree containing the $s$'th smallest element in $T_v$. In the following pages we describe how to compute $\ell$ without explicitly constructing the entire matrix $M'$.

The intuitive idea to guide a query in a given node, $v$, is as follows. Let $K = |T_v \cap A[i, j]|$ be the number of elements from $A[i, j]$ contained in $T_v$. We consider the section from $M'$ containing the $\lceil \log K \rceil$'th least significant bit of each row. All the bits stored in $M'$ before this section are zero and thus not important. Using word-level parallelism we find an interval $[\ell_1, \ell_2] \subseteq [1, f]$, where the $g$ bits of $M'$ match the corresponding $g$ bits of $s$ and the following row. These indices define the subtrees of $T_v$ that can contain the $s$'th smallest element in $T_v$. We then try to determine which of these subtrees contain the $s$'th smallest element. First, we consider the children of $v$ defined by the endpoints of the interval, $\ell_1$ and $\ell_2$. If neither of these contain the $s$'th smallest element in $A[i, j]$, we know that the subtree of $T_v$ containing the $s$'th smallest element stores approximately a factor of $2^g$ elements from $A[i, j]$ fewer than $T_v$, since the $g$ most significant bits of the **prefix** sum of the row corresponding to this subtree are the same as the bits in the preceding row. Stated differently, the number of elements in this subtree does not influence the $g$ most important bits of the prefix sum, and thus it must be *small*. In this case we determine $\ell$ in $O(\log \log n)$ time using a standard binary search. The

point is that this can only occur $O(\log n/g)$ times, and the total cost of these searches is $O(\log n \log\log n/f) = O(\log^{1-\varepsilon} n \log\log n) = o(\log n/\log\log n)$. In the remaining nodes we use constant time.

There are several technical issues that must be worked out. The most important is that we cannot actually produce the needed section of $M'$ in constant time. Instead, we compute an approximation where the number stored in the $g$ bits of each row of the section is at most one too large when compared to the $g$ bits of that row in $M'$. The details are as follows.

In a node $v \in T$ the search is guided using $M_{p_i}$ and $M_{p_j}$ where $p_i$ is the $x$-predecessor of $i-1$ in $T_v$ and $p_j$ is the $x$-predecessor of $j$ in $T_v$. For clarity we use $M_{i-1}$ and $M_j$ for the description. A query maintains an index $c$, initially one, defining which section of the bit-matrices that is currently in use i.e. $c$ defines the section of $M'$ containing the $\lceil \log K \rceil$'th least significant bit. We maintain the following invariant regarding the $c$'th section of $M'$ in the remaining subtree: in $M'$, all bits before the $c$'th section are zero, i.e. the important bits of $M'$ are stored in the $c$'th section or to the right of it. For technical reasons, we ensure that the most important bit of the $c$'th section of $M'$ is zero. This is true before the query starts since the first bit in each row of each stored matrix is zero.

**Computing an approximation of $M'$** We compute the approximation of the $c$'th section of $M'$ from the $c$'th section of $M_j$ and $M_i$. This approximation we denote $w^{i,j}$ and think of it as an $f \times g$ bit-matrix. Basically, the word containing the $c$'th section of bits from $M_{i-1}$ is subtracted from the corresponding word in $M_j$. However, subtracting the $c$'th section of $g$ bits of $t_\ell^{i-1}$ from the corresponding $g$ bits of $t_\ell^j$ does not encompass a potential cascading carry from the lower order bits when comparing the result with the matching $g$ bits of $t_\ell^j - t_\ell^{i-1}$, the $\ell$'th row of $M'$. This means that in the $c$'th section, the $\ell$'th row of $M_{i-1}$ could be larger than $\ell$'th row of $M_j$. To ensure that each pair of rows is subtracted independently in the computation of $w^{i,j}$, we prepend an extra one bit to each row of $M_j$ and an extra zero bit to each row of $M_i$ to deal with cascading carries. Then we subtract the $c$ section of $M_{i-1}$ from the $c$'th section of $M_j$, and obtain $w^{i,j}$. After the subtraction we ignore the value of the most significant bit of each row in $w^{i,j}$ (it is masked out). After this computation, each row in $w^{i,j}$ contain a number that either matches the corresponding $g$ bits of $M'$, or a number that is one larger. Since the most important bit of the $c$'th section of $M'$ is zero, we know that the computation does not *overflow*. If all bits in $w^{i,j}$ are zero the algorithm never needs to consider the current section again, and it is skipped in the remaining subtree by increasing $c$ by one, without breaking the invariant, and $w^{i,j}$ is recomputed.

An example of this computation is showed in Figure 4.3.

**Searching $w^{i,j}$** Let $s_b = s_1, \ldots, s_g$ be the $g$ bits of $s$ defined by the $c$'th section, initially the $g$ most important bits of $s$. If we had actually computed the $c$'th section of $M'$ then only rows matching $s_b$ and the first row containing an even larger number can define the subtree containing the $s$'th smallest element. However, since the rows can contain numbers that are one *to large*, we also

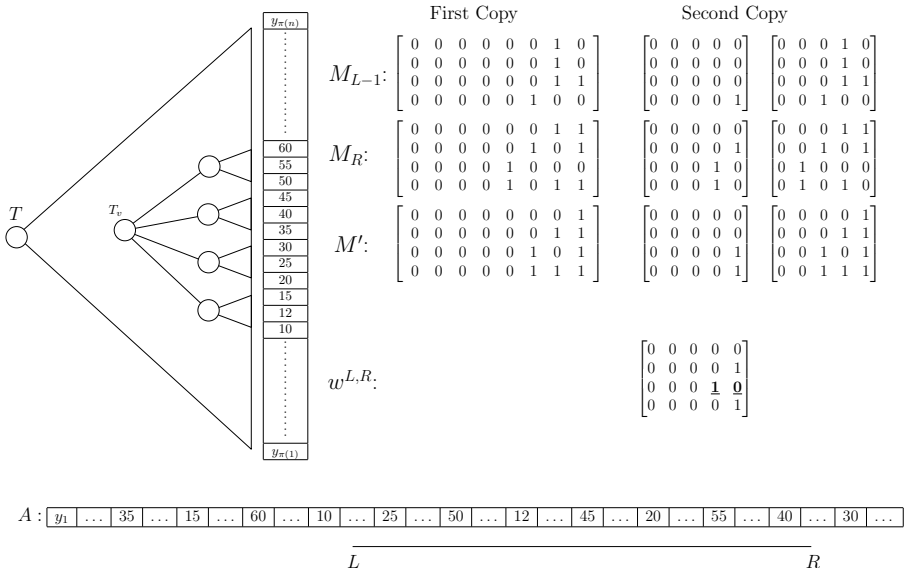



Figure 4.3: A concrete example of the matrices used. In this example, $f = 4$ and $g = 5$. The figure shows the matrices $M_L, M_R$ and $M'$, how they appear when they are divided into sections. The figure also shows the $f \times g$ matrix $w^{L,R}$ a query produces. Notice that the third row of $w^{L,R}$ stores a number one larger than the corresponding bits in matching section of $M'$.

consider all rows matching $s_b + 1$, and the first row storing a number even larger. Therefore, the algorithm locates the first row of $w^{i,j}$ storing a number greater than or equal to $s_b$ and the first row greater than $s_b + 1$. The indices of these rows we denote $\ell_1$ and $\ell_2$, and the subtree containing the $s$'th smallest element corresponds to at row between $\ell_1$ and $\ell_2$. Subsequently, it is checked whether the $\ell_1$'th or $\ell_2$'th subtree contains the $s$'th smallest element in $T_v$ using the first copy of the matrices (where the rows are stored separately). If this is not the case, then the index of the correct subtree is between $\ell_1 + 1$ and $\ell_2 - 1$, and it is determined by a binary search. The binary search uses the first copy of the matrices. In the $c$'th section of $M'$, the $g$ bits from the $\ell_1 + 1$'th row represents at number that is at least $s_b - 1$, and the $\ell_2 - 1$'th row a number that is at most $s_b + 1$. Therefore, the difference between the numbers stored in row $\ell_1 - 1$ and $\ell_2 - 1$ in $M'$ is at most two. This means that in the remaining subtree, the $c$'th section of bits from $M'$ ($t^j_\ell - t^{i-1}_\ell$ for $1 \leq \ell \leq f$) is a number between zero and two. Since the following section stores the last three bits of the current section, the algorithm safely skips the current section in the remaining subtree, by increasing $c$ by one, without violating the invariant. We need two bits to express a number between zero and two, and the third bit ensures that the most significant bit of the $c$'th section of $M'$ is zero. After the subtree, $T_\ell$, containing the $s$'th smallest element is located $s$ is updated as before, $s = s - (t^j_{\ell-1} - t^{i-1}_{\ell-1})$. Let $r_{i-1} = t^{i-1}_\ell - t^{i-1}_{\ell-1}$, be the $x$-rank of $i - 1$ in $T_\ell$, and $r_j = t^j_\ell - t^j_{\ell-1}$, the $x$-rank of $j$ in $T_\ell$. In the subsequent node the algorithm uses the $r_{i-1}$'th and the $r_j$'th stored matrix to guide the search. This corresponds to the matrix

stored for the $x$-predecessor of $i-1$ and the $x$-predecessor of $j$ in $T_\ell$ (fractional cascading).

In the next paragraph we explain how to determine $\ell_1$ and $\ell_2$ in constant time. Thus, if the search continues in the $\ell_1$'th or $\ell_2$'th subtree, the algorithm used $O(1)$ time in the node. Otherwise, a binary search is performed, which takes $O(\log f)$ time, but in the remaining subtree an additional section is skipped. An additional section may be skipped at most $\lceil 1+\log n/(g-3)\rceil = O(f)$ times. When the search is guided using the last section there will not be any problems with cascading carries. This means that the search continues in the subtree corresponding to the first row of $w^{i,j}$ where the number stored is at least as large as $s_b$, and a binary search is never performed in this case. We conclude that a query takes $O(\log n/\log\log n + f\log f) = O(\log n/\log\log n)$ time.

Given $i, j$ and $e$ in a rank query we use a linear space predecessor data structure (van Emde Boas tree [91]) that in $O(\log\log n)$ time yields the predecessor $e_p$ of $e$ in the sorted order of $A$. Then, the path from $e_p$ to the root in $T$ is traversed, and during this walk the number of elements from $A[i,j]$ in subtrees hanging of to the left are added up using the first copy of the bit matrices. The data structures uses $O(nf\log n/\log\log n) = O(n\log^{1+\varepsilon} n/\log\log n)$ space.

**Determining $\ell_1$ and $\ell_2$** The remaining issue is to compute $\ell_1$ and $\ell_2$. A query maintains a search word, $s_w$, that contains $f$ independent blocks of the $g$ bits from $s$ that corresponds to the $c$'th section. Initially, this is the $g$ most important bits of $s$. To compute $s_w$ we store a table that maps each $g$-bit number to a word that contains $f$ copies of these $g$ bits. After updating $s$ we update $s_w$ using a bit-mask and a table look-up. A query knows $w^{i,j} = v_1^1,\ldots,v_g^1,\ldots,v_1^d,\ldots,v_g^d$ and $s_w$ which is $s_b = s_1,\ldots,s_g$ concatenated $f$ times. The $g$-bit block $v_1^\ell,\ldots,v_g^\ell$ from $w^{i,j}$ we denote $w_\ell^{i,j}$ and the $\ell$'th block of $s_1,\ldots,s_g$ from $s_w$ we denote $s_w^\ell$. We only describe how to find $\ell_1$, $\ell_2$ can be found similarly. Remember that $\ell_1$ is the index of the first row in $w^{i,j}$ that stores a number greater than or equal to $s_b$. We make room for an extra bit in each block and make it the most significant. We set the extra bit of each $w_\ell^{i,j}$ to one and the extra bit of each $s_w^\ell$ to zero. This ensures that $w_\ell^{i,j}$ is larger than $s_w^\ell$, for all $\ell$, when both are considered $g+1$ bit numbers. $s_w$ is subtracted from $w^{i,j}$ and because of the extra bit, this operation subtracts $s_w^\ell$ from $w_\ell^{i,j}$, for $1\le\ell\le f$, independently of the other blocks. Then, all but the most significant (fake) bit of each block are masked out. The first one-bit in this word reveals the index $\ell$ of the first block where $w_\ell^{i,j}$ is at least as large as $s_w^\ell$. This bit is found using complete tabulation.

### 4.2.3 Getting Linear Space

In this section we reduce the space usage of our data structure to $O(n)$ words. The previous data structure stores a matrix for each element on each level of the tree, and every matrix uses $O(f\log n)$ bits of space. Instead we only store a matrix for every $t = \lceil f\log n\rceil$'th element. In each node, the sequence of matrices

is divided into chunks of size $t$ and only the last matrix of each chunk is explicitly stored. For each of the remaining elements in a chunk, $\lceil \log f \rceil$ bits are used to describe in which subtree it resides. The description for $d = \lfloor \log n / \lceil \log f \rceil \rfloor$ elements are stored in one word, which we denote a direction word. Prefix sums are stored after each direction word summing up all previous directions words in the chunk, i.e. storing how many elements that was inserted in the first $\ell$ subtrees for $\ell = 1, \ldots, f$. Since each chunk stores the direction of $t$ elements, at most $\lceil f \log t / \log n \rceil = O(1)$ words are needed to store these $f$ prefix sums. We denote it a prefix word. The data structure uses $O(n)$ words of space.

**Range Selection Query** The query works similarly to above. The main difference is that we do not use the matrices $M_{i-1}$ and $M_j$ to compute $w^{i,j}$ since they are not necessarily stored. Instead, we use two matrices that are stored which are *close* to $M_{i-1}$ and $M_j$. The direction and update words enables us to exactly compute any row of $M_j$ and $M_{i-1}$ in constant time. Therefore, the main difference compared to the previous data structure, is that the potential difference between $w^{i,j}$, that we compute, and the $c$'th section of $M'$ is marginally larger, and for this reason the overlap between blocks is increased to four.

In a node $v \in T$ a query is guided as follows. Let $r_i$ be the $x$-rank of $i-1$ and $r_j$ the $x$-rank of $j$ in $T_v$. Let $i' = \lfloor r_i/t \rfloor$ and $j' = \lfloor r_j/t \rfloor$. The matrices stored in the $i'$'th and $j'$'th chunk respectively are used to guide the search. These matrices we denote $M_a$ and $M_b$.

Since $v$ stores a matrix for every $t$'th element in $T_v$, we have $t_\ell^R - t_\ell^b \leq t$ for any $1 \leq \ell \leq f$. Now consider the matrix $\bar{M} = M_b - M_a$, the analog to $M'$ for $a, b$. Then the $\ell$'th row of $\bar{M}$ is at most $t$ smaller than or larger than the $\ell$'th row of $M'$. If we add the difference between the $\ell$'th row of $\bar{M}$ and $M'$ to the $\ell$'th of $\bar{M}$ and ignore cascading carries then only the least $\lceil \log t \rceil = \lceil (1+\varepsilon) \log \log n \rceil$ significant bits change. Stated differently, unless we use the last section, the number stored in the $\ell$'th row of the $c$'th section of $\bar{M}$ is at most one from the corresponding number stored in $M'$.

We can obtain the value of any row in $M_R$ as follows. For each $\ell$ between 1 and $f$, we compute how many of the first $r_R - R't$ elements represented in the $R'+1$'th chunk that are contained in the first $\ell$ children from the direction and prefix words. These are the elements considered in $M^R$ but not in $M^b$. Formally, the $p = \lfloor (r_R - R't)/d \rfloor$'th prefix word stores how many of the first $pd$ elements from the chunk reside in the first $\ell$ children for $1 \leq \ell \leq f$. Using complete tabulation on the following direction word, we obtain a word storing how many of the following $r_R - R't - pd$ elements from the chunk reside in the first $\ell$ children for all $\ell$, $1 \leq \ell \leq f$. Adding this to the $p$'th prefix word yields the difference between the $\ell$'th row of $M_R$ and $M_b$, for all $1 \leq \ell \leq f$. The difference between $M_a$ and $M_{L-1}$ can be computed similarly. Thus, any row of $M_R$ and $M_{L-1}$, and the last section of $M_R$ and $M_{L-1}$ can be computed in constant time.

If the last section is used it is computed exactly in constant time and the search is guided as above. Otherwise, we compute the difference between each row in the $c$'th section of $M_a$ and $M_b$, yielding $w^{a,b}$. As above, the computation

of $w^{a,b}$ does not consider cascading carries from lower order bits and for this reason the $\ell$'th row of $w^{a,b}$ may be one too large when compared to the same bits in $\bar{M}$. Furthermore, the number stored in the $\ell$'th row of $\bar{M}$ could be one larger or one smaller than the corresponding bits of $M'$.

We locate the first row of $w^{a,b}$ that is at least $s_b - 1$ and the first row greater than $s_b + 2$ as above, and the subtree we are searching for is defined by a row between these two, and if it is none of these, a binary search is used to determine it. In this case, by the same arguments as earlier, each row in the $c$'th section of $M'$ in the remaining subtree, represents a number between zero and six. Since we have an overlap of four bits between sections, we safely move to the next section after every binary search.

Range rank queries are supported similarly to above.

**Lemma 4.1** *The data structure described uses linear space and supports range selection and range rank queries in $O(\log n / \log\log n)$ time.*

### 4.2.4 Construction in $O(n \log n)$ Time

In this section we describe how to construct the linear space data structure from the previous section in $O(n \log n)$ time. We sort the input elements, and build a tree with fan-out $f$ on top of them. Then we construct the nodes of the tree level by level starting with the leafs.

In each node $v$, we scan the elements in $T_v$ in chunks of size $t$ ordered by their positions in $A$, and write the direction and prefix words. After processing each chunk, we construct its corresponding matrix. We use an array $C$ of length $f$, there is one entry in $C$ for each child of $v$. We scan the elements by their position in $A$, and if $y_i$ is contained in the $\ell$'th subtree then we add one to $C[\ell]$, and append $\ell$ to the description word using $\lceil \log f \rceil$ bits. After $\lfloor \log n / \lceil \log f \rceil \rfloor$ steps we build a prefix word by making an array $D$ such that $D[i] = \sum_{\ell=1}^{i} C[i]$ and store it in $O(1)$ words. After $t$ steps we store $D$ as a matrix, i.e., the $\ell$'th row of the matrix stores the number $D[\ell]$. We make the second copy of the matrix by repeatedly extracting the needed bits from the first copy. For the next chunk we reset $C$ and repeat. When we build the next matrix we add the numbers stored in $D$ to the previously built matrix, and get the first copy. The second copy is computed as before.

Merging the $f$ lists of elements from the children takes $O(|T_v| \log f)$ time. Adding a number to a direction word takes constant time. Constructing a prefix word takes $O(f)$ time and we do that for every $O(\log(n)/\log f)$'th element. Constructing the matrices takes $O(f)$ time for the first copy and $O(f^2)$ time for the second, and we do this for every $O(f \log n)$'th element. We conclude that we use $O(n \log f) = O(n \log\log n)$ time per level of the tree, and there are $O(\log n / \log\log n)$ levels.

**Theorem 4.1** *The data structure described uses linear space, supports range selection and range rank queries in $O(\log n / \log\log n)$ time, and it can be constructed in $O(n \log n)$ time.*

## 4.3 Dynamic Range Selection

In this section we show how our data structure can be made dynamic. Our dynamic data structure uses $O(n \log n / \log\log n)$ space and supports queries and updates in $O((\log n / \log\log n)^2)$ time, worst case and amortized respectively.

Our data structure maintains a set of points, $S = \{(x_i, y_i)\}$, under insertions and deletions. A query is given values $x_l, x_r$ and an integer $s$ and returns the point with the $s$'th smallest $y$ value among the points in $S$ with $x$-value between $x_l$ and $x_r$. We store the points from $S$ in a weight-balanced search tree [10, 72], ordered by $y$-coordinate. In each node of the tree we maintain the bit-matrices, defined in the static structure, dynamically using a weight-balanced search tree over the points in the subtree, ordered by $x$-coordinate. The main issue is efficient generation of the needed sections of the bit-matrices used by queries. The quality of the approximation is worse than in the static data structure, and we increase the overlap between sections to $O(\log\log n)$. Otherwise, a search works as in the static data structure.

### 4.3.1 Structure

The data structure is a weight-balanced B-tree $T$ with $B = \lceil \log^\varepsilon n \rceil$, where $0 < \varepsilon < \frac{1}{2}$, containing the $n$ points in $S$ in the leaves, ordered by their $y$-coordinates. Each internal node $v \in T$ stores a *ranking tree* $R_v$. This is also a weight-balanced B-tree with $B = \lceil \log^\varepsilon n \rceil$, containing the points stored in $T_v$, ordered by their $x$-coordinates. Each leaf in a ranking tree stores $\Theta(B^2)$ elements. Since the data structure depends on $n$, it is rebuilt every $\Theta(n)$ updates. Let $h = O(\log_B n)$ be the maximal height the trees can get and $f = O(B)$ the maximal fan-out of a node (until the next rebuild).

Let $v$ be a node in $T$ and denote the $a \leq f$ subtrees of $v$ by $T_1, \ldots, T_a$. The ranking tree $R_v$ stored in $v$ is structured as follows. Let $u$ be a node in $R_v$ and denote its $b \leq f$ subtrees by $R_1, \ldots, R_b$. The node $u$ stores $a$ bit-matrices $M_1^u, \ldots, M_a^u$. In the matrix $M_q^u$ the $p$'th row stores the number of elements from $\bigcup_{1 \leq i \leq q} R_i$ that are contained in $\bigcup_{1 \leq i \leq p} T_i$. Additionally, $u$ also stores up to $B^2$ updates, each describing from which subtree of $v$ the update came, from which subtree of $u \in R_v$ the update came, and whether it was an insert or a delete. These updates are stored in $O(B^2 \log B / \log n) = O(1)$ words.

As in the static case, each matrix is stored in two ways. In the first copy each row is stored in one word. For the second copy, each matrix is divided into sections of $g = \lfloor \log n / f \rfloor$ bits, and each section is stored in one word. These sections have $\lceil \log(2h + 2) \rceil + 1 = O(\log\log n)$ bits of overlap.

Finally, a linear space dynamic predecessor data structure [16] containing the $|T_v|$ elements in $R_v$, ordered by $x$, is also stored.

If we ignore the updates stored in update blocks, each matrix $M_j$, as defined in the static data structure, corresponds to the row-wise sum of at most $h$ matrices from $R_v$. Consider the path from the root of $R_v$ to the leaf storing the point in $T_v$ with maximal $x$ coordinate smaller than or equal to $j$, i.e. the predecessor of $j$ in $T_v$ when the points are ordered by their $x$ coordinates. Starting with the zero matrix we add up matrices as follows. If the path continues in the

$\ell$'th subtree at the node $u$, then we add the $\ell - 1$'th matrix stored at $u$ ($M^u_{\ell-1}$) to the sum. Summing up, the $p$'th row of the computed matrix is the number of points, $(x, y) \in T_v$ where $x \leq j$, contained in the first $p$ subtrees of $T_v$, and this is exactly the definition of $M_j$ in the static data structure.

### 4.3.2 Range Selection Query

Given values $L, R$ and an index $s$ in a query, we perform a topdown search in $T$. As in the static data structure, we maintain an index $c$ that defines which section of the matrices is currently in use and approximate the $c$'th section of the matrix $M'$ (as defined in the static data structure).

In a node $v \in T$, we compute an approximation, $w^{L,R}$, of the $c$'th section of $M'$ from the associated ranking tree as follows. First, we locate the leafs of the ranking tree containing the points with maximal $x$-coordinates less than $L - 1$ and $R$ using the dynamic predecessor data structure. Then we traverse the paths from these two leafs in parallel until the paths merge, which happens at the latest at the root. We call them the left and right path.

Initially, we set $w^{L,R}$ to the zero matrix. Assume that we reach node $u_L$ from its $p_L$'th child on the left path and the node $u_R$ from its $p_R$'th child on the right path. Then we subtract the $c$'th section from the $p_L - 1$'th matrix stored in $u_L$ from the $c$'th section of the $p_R - 1$'th matrix stored in $u_R$. The subtraction of sections is performed as in the static data structure. We add the result to $w^{L,R}$. If the paths are at the same node we stop, otherwise, we continue up the tree.

Since we do not consider cascading carries each subtraction might produce a section containing rows that are one to large. Similarly, we add a section to $w^{L,R}$ in each step, which ignores cascading carries from the lower order bits, giving numbers that could be one to small. Furthermore, we ignore up to $B^2$ updates in each step.

Now consider the difference between $w^{L,R}$ and the $c$'th section of $M'$. First of all we have ignored up to $B^2$ updates in two nodes on each level of $T$, which means that in the matrices we consider the combined number that is stored in the $\ell$'th row may differ by up to $2hB^2$ compared to $M'$. As in the static data structure, unless we are using the last section, this only affects the number stored in each row by one.

Furthermore, in the computation of $w^{L,R}$ we do $h$ subtractions and $h$ additions of sections and each of these operations ignores the lower order bits. Combining this with the ignored updates, we get that each row of $w^{L,R}$ is at most $h$ smaller or larger than the corresponding value in $M'$.

Let $s_b$ be the number defined by the $c$'th section of $g$ bits from $s$. The query locates the first row that is at least $s_b - h$ and the first row greater than $s_b + h$. Then the algorithm checks whether either of these corresponds to the subtree containing the answer. If not, the $c$'th section of the remaining matrices store a number between zero and $2h$, and for this reason the overlap between sections is set to $\lceil \log(2h + 1) \rceil + 1 = O(\log \log n)$ bits. In this case we use a binary search that in each step traverses $R_v$ as above (path up from leaves containing the maximal $x$-coordinate less than $L - 1$ and $R$) to compute the needed rows

of $M'$ exactly using the first copy of the stored matrices and the information stored in the update words. Extracting information from update words is done by complete tabulation on each of the $O(1)$ update words.

If we are using the last section, there will not be problems with the lower order bits in the additions and subtractions of sections. We add the changes stored in the update words on the paths and obtain the last section of $M'$ exactly, and a binary search is never performed in this case.

As before, the current section is skipped in the remaining subtree in this case.

### 4.3.3 Updates

An element $e$ is inserted into the data structure as follows. First, $e$ is added to $T$. If a node in $T$ splits, two new structures are built from the existing one, and the parent node is rebuilt. Then, we traverse path from $e$ to the root. In each node $v \in T$ on this path, $e$ is inserted into the ranking tree $R_v$, and the dynamic predecessor data structure. If a node splits in $R_v$, two new nodes are constructed from the old one, and the parent rebuilt. After $R_v$ has been updated, $e$ is inserted in each node in $R_v$ on the path from $e$ to the root, by appending a description of the update to the update words. When $B^2$ updates have been appended to a node in a ranking tree, all bit-matrices in this node are recomputed and the update words are cleared. Deletes are handled similarly, except that updates to the base tree and ranking trees are handled using global rebuilding.

**Theorem 4.2** *The data structure uses $O(n \log n / \log\log n)$ space and queries and updates are supported in $O((\log n / \log\log n)^2)$ time worst case and amortized respectively.*

*Proof.* The height of $T$ is $O(\log n / \log B) = O(\log n / \log\log n)$ and in each node on a path we traverse a ranking tree of height $O(\log n / \log B) = O(\log n / \log\log n)$. We do a binary search $O(\log n / B)$ times and each binary search takes $O(\log B \log n / \log\log n) = O(\log n)$ time. The combined time for all binary searches is $O(\log^2 n / B) = o((\log n / \log\log n)^2)$ time.

Updating the base tree $T$ (splitting nodes) takes $O(B)$ time amortized per update. Updating a ranking also takes $O(B)$ time amortized per update and $O(\log n / \log B)$ ranking trees are changed in each update. Furthermore, an update is appended to a node on each level of a ranking tree, on each level of $T$. This means that $O((\log n / \log\log n)^2)$ update words are changed. The matrices stored in a node in a ranking tree can be recomputed in $O(f^2) = O(B^2)$ time. This amounts to $O(1)$ time amortized per update added to the node. Each element defines a node in a ranking tree on each level of $T$. A node in a ranking tree stores at most $f$ bit-matrices, each storing $f$ numbers, and up to $B^2$ updates, each using $O(\log B)$ bits of space. Thus, a node in the ranking tree uses $O(f^2 + B^2 \log B / \log n) = O(B^2)$ words of space, and each ranking tree, $R_v$, contains $O(|T_v|/B^2)$ nodes. □

## 4.4 Lower Bound for Dynamic Data Structures

In this section we describe a reduction from the marked ancestor problem to a dynamic range median data structure. In the marked ancestor problem the input is a complete tree of degree $b$ and height $h$. An update marks or unmarks a node of the tree, initially all nodes are unmarked. A query is provided a leaf $v$ of the tree and must return whether there exist a marked ancestor of $v$. Let $t_q$ and $t_u$ be the query and update time for a marked ancestor data structure. Alstrup et al. proved the following lower bound trade-off for the problem, $t_q = \Omega(\frac{\log n}{\log(t_u w \log n)})$ [4], where $w$ is the word size.

**Reduction** Let $T$ denote a marked ancestor tree of height $h$ and degree $b$. For each node $v$ in $T$ we associate two pairs of elements, which we denote *start-mark* and *end-mark*. We translate $T$ into an array of size $4|T|$ by a recursive traversal of $T$, where we for each node $v$ outputs its start-mark, then recursively visit each of $v$'s children, and then output $v$'s end-mark. Start-marks are used to mark a node, and end-marks ensure that markings only influences the answer for queries in the marked subtree. When a node $v$ is unmarked, start-mark=end-mark=(0,1) and when $v$ is marked, start-mark is set to (1,1) and end-mark to (0,0).

**Query** A marked ancestor query for a leaf $v$ is answered by returning yes if and only if the range median from the subarray ranging from the beginning of the array to the start-mark element associated with $v$ is one. If zero nodes are marked, the array is on the form $[0, 1, \ldots, 0, 1]$. Since the median in any range that can be considered by a query is zero, any marked ancestor query returns no. If $v$ or one of its ancestors is marked there will be more ones than zeros in the range for $v$, and the query answers yes. A node $u$ that is not an ancestor of $v$ has its start-mark and end-mark placed either before $v$'s marks or after $v$'s marks, and independently of whether $u$ is marked or not, it contributes and equal number of zeroes and ones to $v$'s query range. Since the reduction requires an overhead of $O(1)$ for both queries and updates we get the following lower bound.

**Theorem 4.3** *Any data structure that supports updates in $O(\log^{O(1)} n)$ time uses $\Omega(\log n / \log\log n)$ time to support a range median query.*

## 4.5 Main Open Problems

There are two main open problems. First, what is the lower bound on the query time for range selection queries in static $O(n \log^{O(1)} n)$ space data structures? We can prove that any $O(n \log^{O(1)} n)$ space data structure needs $\Theta(\log n / \log\log n)$ time for three-sided range median queries by a reduction from two dimensional rectangle-stabbing [76]. Furthermore, there is a gap between the upper and lower bounds for the batched range median problem for $k = \Omega(n^{1+\varepsilon})$, and the lower bound [50] is only valid in the comparison model.

# Chapter 5

## Cell Probe Lower Bounds and Approximations for Range Mode

In this chapter we consider the range mode problem, the range $k$-frequency problem, and the $c$-approximate range mode problem. The frequency of a label $l$ in a multiset $S$ of labels, is the number of occurrences of $l$ in $S$. The mode of $S$ is the most frequent label in $S$. In case of ties, any of the most frequent labels in $S$ can be designated the mode.

For all the problems we consider the input is an array $A$ of length $n$ containing labels. For simplicity we assume that each label is an integer between one and $n$. In the range mode problem, we must preprocess $A$ into a data structure that given indices $i$ and $j$, $1 \leq i \leq j \leq n$, returns the mode, $M_{i,j}$, in the subarray $A[i,j] = A[i], A[i+1], \ldots, A[j]$. We let $F_{i,j}$ denote the frequency of $M_{i,j}$ in $A[i,j]$. In the $c$-approximate range mode problem, a query is given indices $i$ and $j$, $1 \leq i \leq j \leq n$, and returns a label that has a frequency of at least $F_{i,j}/c$. In the range $k$-frequency problem, a query is given indices $i$ and $j$, $1 \leq i \leq j \leq n$, and returns whether there is a label occurring precisely $k$ times in $A[i,j]$.

For the upper bounds we consider the unit cost RAM, and let $w = \Theta(\log n)$ denote the word size. For lower bounds we consider the cell probe model of Yao [95]. In this model of computation a random access memory is divided into cells of $w$ bits. The complexity of an algorithm is the number of memory cells the algorithm accesses and all other computations are free.

**Previous Results.** The first data structure for the range mode problem achieving constant query time was developed in [62]. This data structure uses $O(n^2 \log\log n/\log n)$ space. This was subsequently improved to $O(n^2/\log n)$ space in [78] and finally to $O(n^2 \log\log n/\log^2 n)$ in [79]. For non-constant query time, the first developed data structure uses $O(n^{2-2\varepsilon})$ space and answers queries in $O(n^{\varepsilon}\log n)$ time, where $0 < \varepsilon \leq \frac{1}{2}$ is a query-space tradeoff constant [62]. The query time was later improved to $O(n^{\varepsilon})$ without changing the space bound [78].

Given the rather large bounds for the range mode problem, the approximate variant of the problem was considered in [22]. With constant query time, they solve 2-approximate range mode with $O(n \log n)$ space, 3-approximate range

mode with $O(n \log\log n)$ space, and 4-approximate range mode with linear space. For $(1+\varepsilon)$-approximate range mode, they describe a data structure that uses $O(\frac{n}{\varepsilon})$ space and answers queries in $O(\log\log_{(1+\varepsilon)} n) = O(\log\log n + \log\frac{1}{\varepsilon})$ time. This data structure gives a linear space solution with $O(\log\log n)$ query time for $c$-approximate range mode when $c$ is constant.

There are no non-trivial lower bounds for the any of the problems we consider.

**Our Results** In this chapter we show the first lower bounds for range mode data structures and range $k$-frequency data structures and provide new upper bounds for the $c$-approximate range mode problem and the range $k$-frequency problem.

In Section 5.1 we prove our lower bound for range mode data structures. Specifically, we prove that any data structure that uses $S$ cells and supports range mode queries must have a query time of $\Omega(\frac{\log n}{\log(Sw/n)})$. This means that any data structure that uses $O(n \log^{O(1)} n)$ space needs $\Omega(\log n/\log\log n)$ time to answer a range mode query. Similarly, any data structure that supports range mode queries in constant time needs $\Omega(n^{1+\Omega(1)})$ space. We suspect that the actual lower bound for near-linear space data structures for the range mode problem is significantly larger. However a fundamental obstacle in the cell probe model is to prove lower bounds for static data structures that are higher than the number of bits needed to describe the query. The highest known lower bounds are achieved by the techniques in [76, 77] that uses reductions from problems in communication complexity. We use this technique to obtain our lower bound and our bound matches the highest lower bound achieved with this technique.

Actually our construction proves the same lower bound for queries on the form, is there an element with frequency at least (or precisely) $k$ in $A[i,j]$, where $k$ is given at query time. In the scenario where $k$ is fixed for all queries it is trivial to give a linear space data structure with constant query time for determining whether there is an element with frequency at least $k$. In Section 5.2 we consider the case of determining whether there is an element with frequency exactly $k$, which we denote the range $k$-frequency problem. To the best of our knowledge, we are the first to consider this problem. We show that 2D rectangle stabbing reduces to range $k$-frequency for any constant $k > 1$. This reduction proves that any data structure that uses $S$ space, needs $\Omega(\log n/\log(Sw/n))$ time for a query [75, 76], for any constant $k > 1$. Secondly, we reduce range $k$-frequency to 2D rectangle stabbing. This reduction works for any $k$. This immediately gives a data structure for range $k$-frequency that uses linear space, and answers queries in optimal $O(\log n/\log\log n)$ time [57] (we note that 2D rectangle stabbing reduces to 2D range counting). In the restricted case where $k = 1$, this problem corresponds to determining whether there is a unique label in a subarray. The reduction from 2D rectangle stabbing only applies for $k > 1$. We show, somewhat surprisingly, that determining whether there is a label occurring exactly twice (or $k > 1$ times) in a subarray, is exponentially harder than determining if there is a label occurring exactly once. Specifically,

we reduce range 1-frequency to four-sided 3D orthogonal range emptiness, which can be solved with $O(\log^2 \log n)$ query time and $O(n \log n)$ space by a slight modification of the data structure presented in [1].

In Section 5.3 we present a simple data structure for the 3-approximate range mode problem. The data structure uses linear space and answers queries in constant time. In Section 5.4 we develop a data structure for $(1 + \varepsilon)$-approximate range mode. This data structure uses $O(\frac{n}{\varepsilon})$ space and answers queries in $O(\log \frac{1}{\varepsilon})$ time.

## 5.1 Cell Probe Lower Bound for Range Mode

In this section we show a query lower bound of $\Omega(\log n / \log \log n)$ for any range mode data structure that uses $O(n \log^{O(1)} n)$ space for an input array of size $n$.

We prove the lower bound for the slightly different problem of determining the frequency of the mode. Since the frequency of an element can be determined in any range in $O(\log \log n)$ time (two predecessor searches) the lower bound for range mode follows.

**Communication Complexity and Lower Bounds** In communication complexity we have two players Alice and Bob. Alice receives as input a bit string $x$ and Bob a bit string $y$. Given some predefined function, $f$, the goal for Alice and Bob is to compute $f(x, y)$ while communicating as few bits as possible.

Lower bounds on the communication complexity of various functions have been turned into lower bounds for static data structure problems in the cell probe model. The idea is as follows [70]: Assume we are given a static data structure problem and consider the function $f(q, D)$ that is defined as the answer to a query $q$ on an input set $D$ for this problem. If we have a data structure for the problem that uses $S$ memory cells and supports queries in time $t$ we get a communication protocol for $f$ where Alice sends $t \log S$ bits and Bob sends $tw$ bits. In this protocol Alice receives $q$ and Bob receives $D$. Bob constructs the data structure on $D$ and Alice simulates the query algorithm. In each step Alice sends $\log S$ bits specifying the memory cell of the data structure she needs and Bob replies with the $w$ bits of this cell. Finally, Alice outputs $f(q, D)$. Thus, a communication lower bound for $f$ gives a lower bound tradeoff between $S$ and $t$.

This construction can only be used to distinguish between polynomial and exponential space data structures. Since range mode queries are trivially solvable in constant time with $O(n^2)$ space, we need a different technique to obtain lower bounds for near-linear space data structures. Pătraşcu and Thorup [76,77] developed a technique for distinguishing between near linear and polynomial space by considering reductions from communication complexity problems to $k$ parallel data structure queries. The main insight is that Alice can simulate all $k$ queries in parallel and only send $\log \binom{S}{k} = O(k \log \frac{S}{k})$ bits to define the $k$ cells she need in each step. For the right values of $k$ this is significantly less than $k \log S$ bits which Alice would need in each step for performing the queries

sequentially.

**Lopsided Set Disjointness (LSD)** In LSD Alice and Bob receive subsets $S$ and $T$ of a universe $U$. The goal for Alice and Bob is to compute whether $S \cap T \neq \emptyset$. LSD is parameterized with the size $|S| = N$ of Alice's set and the fraction between the size of the universe and $N$, which is denoted $B$, e.g. $|U| = NB$. Notice that the size of Bob's set is arbitrary and could be as large as $NB$. We use $[X]$ to denote the set $\{1, 2, \ldots, X\}$. There are other versions of LSD where the input to Alice has more structure. For our purpose we need Blocked-LSD. For this problem the universe is considered as the cartesian product of $[N]$ and $[B]$, e.g. $U = [N] \times [B]$ and Alice receives a set $S$ such that $\forall j \in [N]$ there exists a unique $b_j \in [B]$ such that $(j, b_j) \in S$, e.g. $S$ is of the form $\{(1, b_1), (2, b_2), \ldots, (N, b_N) \mid b_i \in [B]\}$. The following lower bound applies for this problem [76].

**Theorem 5.1** *Fix $\delta > 0$. In a bounded-error protocol for Blocked-LSD, either Alice sends $\Omega(N \log B)$ bits or Bob sends $\Omega(NB^{1-\delta})$ bits.*

**Blocked-LSD reduces to $N/k$ parallel range mode queries** Given $n$, we describe a reduction from Blocked-LSD with a universe of size $n$ ($n = NB$) to $N/k$ parallel range mode queries on an input array $A$ of size $\Theta(n)$. The size of $A$ may not be exactly $n$ but this will not affect our result. The parameters $k$ and $B$ are fixed later in the construction. From a high level perspective we construct an array of permutations of $[kB]$. A query consists of a suffix of one permutation, a number of complete permutations, and a prefix of another permutation. They are chosen such that the suffix determines a subset of Bob's set and the prefix a subset of Alice's set. The frequency of the mode is equal to two plus the number of complete permutations spanned by the query if and only if the two sets intersect.

Bob will store a range mode data structure and Alice will simulate the query algorithm. First we describe the array $A$ that Bob constructs when he receives his input. Let $T \subseteq [N] \times [B]$ be this set. The array Bob constructs consists of two parts which are described separately. We let $\cdot$ denote concatenation of lists. We also use this operator on sets and in this case we treat the set as a list by placing the elements in lexicographic order. Bob partitions $[N]$ into $N/k$ consecutive chunks of $k$ elements, e.g. the $i$'th chunk is $\{(i-1)k+1, \ldots, ik\}$ for $i = 1, \ldots, N/k$. With the $i$'th chunk Bob associates the subset $L_i$ of $T$ with first coordinate in that chunk, e.g. $L_i \subseteq \{(i-1)k+t \mid t = 1, \ldots, k\} \times [B]$. Each $L_i$ is mapped to a permutation of $[kB]$.

We define the mapping $f : (x, y) \to (x - 1 \bmod k)B + y$ and let the permutation be $([kB] \setminus f(L_i)) \cdot f(L_i)$. The first part of $A$ is the concatenation of the permutations defined for each chunk $L_i$ ordered by $i$, e.g. $([kB] \setminus f(L_1)) \cdot f(L_1) \cdots ([kB] \setminus f(L_{N/k})) \cdot f(L_{N/k})$. The second part of $A$ consists of $B^k$ permutations of $[kB]$. There is one permutation for each way of picking a set of the form $\{(1, b_1), \ldots, (k, b_k) \mid b_i \in [B]\}$. Let $R_1, \ldots, R_{B^k}$ denote the $B^k$ sets on this form ordered lexicographically. The second part of the array becomes $f(R_1) \cdot ([kB] \setminus f(R_1)) \cdots f(R_{B^k}) \cdot ([kB] \setminus f(R_{B^k}))$.

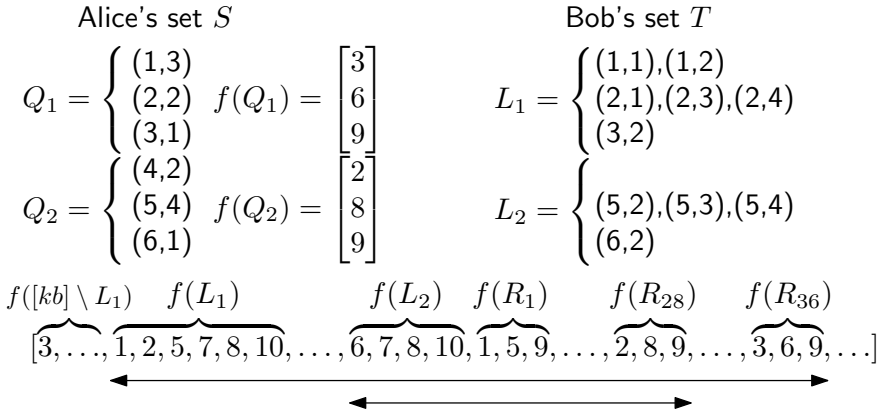



Figure 5.1: An example of the array Bob creates and the queries Alice simulates. In this example $N = 6, B = 4$ and $k = 3$.

We now show how Alice and Bob can determine whether $S \cap T \neq \emptyset$ from this array. Bob constructs a range mode data structure for $A$ and sends $|L_i|$ for $i = 1, \ldots, N/k$ to Alice. Alice then simulates the query algorithm on the range mode data structure for $N/k$ queries in parallel. The $i$'th query determines whether the $k$ elements $Q_i = \{((i-1)k+1, b_{(i-1)k+1}), \ldots, (ik, b_{ik})\}$ from $S$ have an empty intersection with $T$ (actually $L_i$) as follows.

Alice determines the end index of $f(Q_i)$ in the second part of $A$. We note that $f(Q_i)$ always exists in the second part of $A$ by construction and Alice can determine the position without any communication with Bob. Alice also determines the start index of $f(L_i)$ in the first part of $A$ from the sizes she initially received from Bob. The $i$'th query computes the frequency $R_i$ of the mode between these two indices. Let $p$ be the number of permutations of $[kB]$ stored between the end of $f(L_i)$ and the beginning of $f(Q_i)$ in $A$, then $F_i - p = 2$ if and only if $Q_i \cap T \neq \emptyset$, and $F_i - p = 1$ otherwise. Since each permutation of $[kB]$ contributes one to $F_i$, $F_i - p$ is equal to two if and only if at least one of the elements from $Q_i$ is in $L_i$ meaning that $S \cap T \neq \emptyset$. We conclude that Blocked-LSD reduces to $N/k$ range mode queries in an array of size $NB + B^k kB$.

To obtain a lower bound for range mode data structures we consider the parameters $k$ and $B$ and follow the idea from [76]. Let $S$ be the size of Bob's range mode data structure and let $t$ be the query time. In our protocol for Blocked-LSD Alice sends $t \log \binom{S}{N/k} = O(t\frac{N}{k} \log \frac{Sk}{N})$ bits and Bob sends $twN/k + N/k \log(kB)$ bits. By Theorem 5.1, either Alice sends $\Omega(N \log B)$ bits or Bob sends $\Omega(NB^{1-\delta})$. Fix $\delta = \frac{1}{2}$. Since $N/k \log(kB) = o(N\sqrt{B})$ we obtain that either $t\frac{N}{k} \log(\frac{Sk}{N}) = \Omega(N \log B)$ or $twN/k = \Omega(N\sqrt{B})$. We constrain $B$ such that $B \geq w^2$ and $\log B \geq \frac{1}{2} \log(\frac{Sk}{N}) \Rightarrow B \geq \frac{Sk}{n}$ and obtain $t = \Omega(k)$. Since $|A| = NB + B^k kB$ and we require $|A| = \Theta(n)$, we set $k = \Theta(\log_B n)$. To maximize $k$ we choose $B = \max\{w^2, \frac{Sk}{n}\}$. We obtain that $t = \Omega(k) = \Omega(\log N / \log \frac{Swk}{n}) = \Omega(\log n / \log \frac{Sw}{n})$ since $w > k$.

**Theorem 5.2** *Any data structure that uses $S$ space needs $\Omega(\frac{\log n}{\log(\frac{Sw}{n})})$ time for a range mode query.*

## 5.2 Range $k$-Frequency

In this section, we consider the *range $k$-frequency* problem and its connection to classic geometric data structure problems. We show that the range $k$-frequency problem is equivalent to 2D rectangle stabbing for any fixed constant $k > 1$, and that for $k = 1$ the problem reduces to four-sided 3D orthogonal range emptiness.

In the 2D rectangle stabbing problem the input is $n$ axis-parallel rectangles. A query is given a point, $(x, y)$, and must return whether this point is contained[1] in at least one of the $n$ rectangles in the input. A query lower bound of $\Omega(\log n/\log\log n)$ for data structures using $O(n\log^{O(1)} n)$ space is proved in [76], and a linear space static data structure with this query time can be found in [57].

In four-sided 3D orthogonal range emptiness, we are given a set $P$ of $n$ points in 3D, and must preprocess $P$ into a data structure, such that given an open-ended four-sided rectangle $R = [-\infty, x] \times [y_1, y_2] \times [z, \infty]$, the data structure returns whether $R$ contains a point $p \in P$. Currently, the best solution for this problem uses $O(n \log n)$ space and supports queries in $O(\log^2 \log n)$ time [1].

For simplicity, we assume that each coordinate is a unique integer between one and $2n$ (rank space).

**Theorem 5.3** *The range $k$-frequency problem reduces to 2D rectangle stabbing.*

*Proof.* Let $A$ be the input to the range $k$-frequency problem. We translate the ranges of $A$ where there is a label with frequency $k$ into $O(n)$ rectangles as follows. Fix a label $x \in A$, and let $s_x \geq k$ denote the number of occurrences of $x$ in $A$. If $s_x < k$ then $x$ is irrelevant and we discard it. Otherwise, let $i_1 < i_2 < \ldots < i_s$ be the positions of $x$ in $A$, and let $i_0 = 0$ and $i_{s+1} = n + 1$. Consider the ranges of $A$ where $x$ has frequency $k$. These are the subarrays, $A[a, b]$, where there exists an integer $\ell$ such that $i_\ell < a \leq i_{\ell+1}$ and $i_{\ell+k} \leq b < i_{\ell+k+1}$ for $0 \leq \ell \leq s_x - k$. This defines $s_x - k + 1$ two dimensional rectangles, $[i_\ell + 1, i_{\ell+1}] \times [i_{\ell+k}, i_{\ell+k+1} - 1]$ for $\ell = 0, \ldots, s_x - k$, such that $x$ has frequency $k$ in $A[i, j]$ if and only if the point $(i, j)$ stabs one of the $s_x - k + 1$ rectangles defined by $x$. By translating the ranges of $A$ where a label has frequency $k$ into the corresponding rectangles for all distinct labels in $A$, we get a 2D rectangle stabbing instance with $O(n)$ rectangles. □

This means that we get a data structure for the range $k$-frequency problem that uses $O(n)$ space and supports queries in $O(\log n/\log\log n)$ time.

**Theorem 5.4** *For $k = 1$, the range $k$-frequency problem reduces to four-sided orthogonal range emptiness queries in 3D.*

*Proof.* For each distinct label $x \in A$, we map the ranges of $A$ where $x$ has frequency one (it is unique in the range) to a 3D point. Let $i_1 < i_2 < \ldots < i_s$ be the positions of $x$ in $A$, and let $i_0 = 0$ and $i_{s+1} = n+1$. The label $x$ has frequency one in $A[a, b]$ if there exist an integer $\ell$ such that $i_{\ell-1} < a \leq i_\ell \leq b < i_{\ell+1}$.

[1] points on the border of a rectangle are contained in the rectangle

We define $s$ points, $P_x = \{(i_{\ell-1}+1, i_\ell, i_{\ell+1}-1) \mid 1 \leq \ell \leq s\}$. The label $x$ has frequency one in the range $A[a,b]$ if and only if the four-sided orthogonal range query $[-\infty,a] \times [a,b] \times [b,\infty]$ contains a point from $P_x$ (we say that $x$ is inside range $[x_1,x_2]$ if $x_1 \leq x \leq x_2$). Therefore, we let $P = \bigcup_{x \in A} P_x$ and get a four-sided 3D orthogonal range emptiness instance with $O(n)$ points. □

Thus, we get a data structure for the range 1-frequency problem that uses $O(n \log n)$ space and supports queries in $O(\log^2 \log n)$ time.

**Theorem 5.5** *Let $k$ be a constant greater than one. The 2D rectangle stabbing problem reduces to the range $k$-frequency problem.*

*Proof.* We show the reduction for $k = 2$ and then generalize this construction to any constant value $k > 2$.

Let $R_1, \ldots, R_n$ be the input to the rectangle stabbing problem. We construct a range 2-frequency instance with $n$ distinct labels each of which is duplicated exactly 6 times. Let $R_\ell$ be the rectangle $[x_{\ell_0}, x_{\ell_1}] \times [y_{\ell_0}, y_{\ell_1}]$. For each rectangle, $R_\ell$, we add the pairs $(x_{\ell_0}, \ell)$, $(x_{\ell_1}, \ell)$ and $(x_{\ell_1}, \ell)$ to a list $X$. Similarly, we add the pairs $(y_{\ell_0}, \ell)$, $(y_{\ell_1}, \ell)$, and $(y_{\ell_1}, \ell)$ to a list $Y$. We sort $X$ in descending order and $Y$ in ascending order by their first coordinates. Since we assumed all coordinates are unique, the only ties are amongst pairs originating from the same rectangle, here we break the ties arbitrarily. The concatenation of $X$ and $Y$ is the range 2-frequency instance and we denote it $A$, i.e. the second component of each pair are the actual entries in $A$, and the first component of each pair is ignored.

We translate a 2D rectangle stabbing query, $(x,y)$, into a query for the range 2-frequency instance as follows. Let $p_x$ be the smallest index where the first coordinate of $X[p_x]$ is $x$, and let $q_y$ be the largest index where the first coordinate of $Y[p_y]$ is $y$. If $A[p_x] = A[p_x+1]$, two consecutive entries in $A$ are defined by the right endpoint of the same rectangle, we set $i_x = p_x + 2$ (we move $i_x$ to the right of the two entries), otherwise we set $i_x = p_x$. Similarly for the $y$ coordinates, if $A[|X|+q_y] = A[|X|+q_y-1]$ we set $j_y = q_y - 2$ (move $j_y$ left of the two entries), otherwise we set $j_y = q_y$. Finally we translate $(x,y)$ to the range 2-frequency query $[i_x, |X| + j_y]$ on $A$, see Figure 5.2. Notice that in the range 2-frequency queries that can be considered in the reduction, the frequency of a label is either one, two, three, four or six. The frequency of label $\ell$ in $A[i_x, |X|]$ is one if $x_{\ell_0} \leq x \leq x_{\ell_1}$, three if $x > x_{\ell_1}$ and zero otherwise. Similar, the frequency of $\ell$ in $A[|X|+1, |X|+j_y]$ is one if $y_{\ell_0} \leq y \leq y_{\ell_1}$, three if $y > y_{\ell_1}$ and zero otherwise. We conclude that the point $(x,y)$ stabs rectangle $R_\ell$ if and only if the label $\ell$ has frequency two in $A[i_x, |X| + j_y]$. Since $x, y \in \{1, \ldots, 2n\}$, we can store a table with the translations from $x$ to $i_x$ and $y$ to $j_y$. Thus, we can translate 2D rectangle stabbing queries to range 2-frequency queries in constant time.

For $k > 2$ we place $k-2$ copies of each label between $X$ and $Y$ and translate the queries accordingly. □

We conclude that for data structures using $O(n \log^{O(1)} n)$ space, the range $k$-frequency problem is exponentially harder for $k > 1$ than for $k = 1$.

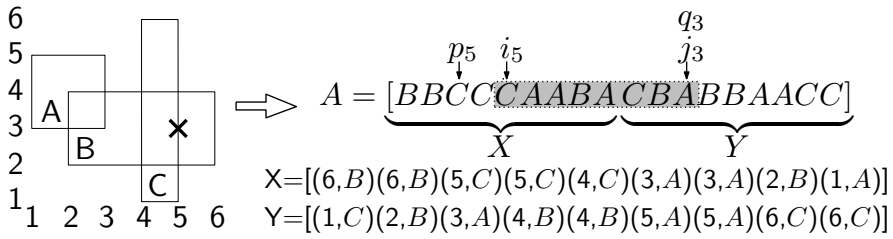



Figure 5.2: Reduction from 2D rectangle stabbing to range 2-frequency. The $\times$ marks a stabbing query, $(5, 3)$. This query is mapped to the range 2-frequency query $[i_5, |X| + j_3]$ in $A$, which is highlighted. Notice that $i_5 = p_5 + 2$ since $A[p_5] = A[p_5 + 1]$.

## 5.3 3-Approximate Range Mode

In this section, we construct a data structure that given a range $[i, j]$ computes a 3-approximation of $F_{i,j}$.

We use the following observation from [22]. If we can cover $A[i, j]$ with three disjoint subintervals $A[i, x], A[x + 1, y]$ and $A[y + 1, j]$ for which we know $F_{i,x}, F_{x+1,y}$ and $F_{y+1,j}$, then

$$\tfrac{1}{3}F_{i,j} \leq \max\{F_{i,x}, F_{x+1,y}, F_{y+1,j}\} \leq F_{i,j}.$$

First, we describe a data structure that uses $O(n \log\log n)$ space, and then we show how to reduce the space to $O(n)$. The data structure consists of a tree $T$ of polynomial fanout where the $i$'th leaf stores $A[i]$, for $i = 1, \ldots, n$. For a node $v$ let $T_v$ denote the subtree rooted at $v$ and let $|T_v|$ denote the number of leaves in $T_v$. The fanout of node $v$ is $f_v = \lceil\sqrt{|T_v|}\rceil$. The height of $T$ is $\Theta(\log\log n)$. Along with $T$, we store a lowest common ancestor (LCA) data structure, which given indices $i$ and $j$, finds the LCA of the leaves corresponding to $i$ and $j$ in $T$ in constant time [51].

For every node $v \in T$, let $R_v = A[a, b]$ denote the consecutive range of entries stored in the leaves of $T_v$. The children $c_1, \ldots, c_{f_v}$ of $v$ partition $R_v$ into $f_v$ disjoint subranges $R_{c_1} = A[a_{c_1}, b_{c_1}]$, $\ldots$, $R_{c_{f_v}} = A[a_{c_{f_v}}, b_{c_{f_v}}]$ each of size $O(\sqrt{|T_v|})$. For every pair of children $c_r$ and $c_s$ where $r < s - 1$, we store $F_{a_{c_{r+1}}, b_{c_{s-1}}}$. Furthermore, for every child range $R_{c_i}$ we store $F_{a_{c_i},k}$ and $F_{k,b_{c_i}}$ for every prefix and suffix range of $R_{c_i}$ respectively. To compute a 3-approximation of $F_{i,j}$, we find the LCA of $i$ and $j$. This is the node $v$ in $T$ for which $i$ and $j$ lie in different child subtrees, say $T_{c_x}$ and $T_{c_y}$ with ranges $R_{c_x} = [a_{c_x}, b_{c_x}]$ and $R_{c_y} = [a_{c_y}, b_{c_y}]$. We then lookup the frequency $F_{a_{c_{x+1}}, b_{c_{y-1}}}$ stored for the pair of children $c_x$ and $c_y$, as well as the suffix frequency $F_{i,b_{c_x}}$ stored for the range $A[i, b_{c_x}]$ and the prefix frequency $F_{a_{c_y},j}$ stored for $A[a_{c_y}, j]$, and return the max of these.

Each node $v \in T$ uses $O(|T_v|)$ space for the frequencies stored for each of the $O(|T_v|)$ pairs of children, and $O(|T_v|)$ for all the prefix and suffix range frequencies. Since each node $v$ uses $O(|T_v|)$ space and the LCA data structure uses $O(n)$ space, our data structure uses $O(n \log\log n)$ space. A query makes

one LCA query and computes the max of three numbers which takes constant time.

We just need one observation to bring the space down to $O(n)$. Consider a node $v \in T$. The largest possible frequency that can be stored for any pair of children of $v$, or for any prefix or suffix range of a child of $v$ is $|T_v|$, and each such frequency can be represented by $b = 1 + \lfloor \log |T_v| \rfloor$ bits. We divide the frequencies stored in $v$ into chunks of size $\lfloor \frac{\log n}{b} \rfloor$ and pack each of them in one word. This reduces the total space usage of the nodes on level $i$ to $O(n/2^i)$. We conclude that the data structure uses $O(n)$ space and supports queries in constant time.

**Theorem 5.6** *There exists a data structure for the* 3*-approximate range mode problem that uses $O(n)$ space and supports queries in constant time.*

## 5.4 $(1+\varepsilon)$-Approximate Range Mode

In this section, we describe a data structure using $O(\frac{n}{\varepsilon})$ space that given a range $[i,j]$, computes a $(1+\varepsilon)$-approximation of $F_{i,j}$ in $O(\log \frac{1}{\varepsilon})$ time. We use that $\frac{1}{\log(1+\varepsilon)} = O(\frac{1}{\varepsilon})$ for any $0 < \varepsilon \leq 1$.

Our data structure consists of two parts. The first part solves all queries $[i,j]$ where $F_{i,j} \leq \lceil \frac{1}{\varepsilon} \rceil$, and the latter solves the remaining. The first data structure also decides whether $F_{i,j} \leq \lceil \frac{1}{\varepsilon} \rceil$.

**Small Frequencies** For $i = 1, \ldots, n$ we store a table, $Q_i$, of length $\lceil \frac{1}{\varepsilon} \rceil$, where the value in $Q_i[k]$ is the largest integer $j \geq i$ such that $F_{i,j} = k$. To answer a query $[i,j]$ we do a successor search for $j$ in $Q_i$. If $j$ does not have a successor in $Q_i$ then $F_{i,j} > \lceil \frac{1}{\varepsilon} \rceil$, and we query the second data structure. Otherwise, let $s$ be the index of the successor of $j$ in $Q_i$, then $F_{i,j} = s$. The data structure uses $O(\frac{n}{\varepsilon})$ space and supports queries in $O(\log \frac{1}{\varepsilon})$ time.

**Large Frequencies** For every index $1 \leq i \leq n$, define a list $T_i$ of length $t = \lceil \log_{1+\varepsilon}(\varepsilon n) \rceil$, with the following invariant: For all $j$, if $T_i[k-1] < j \leq T_i[k]$ then $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$. The following assignment of values to the lists $T_i$ satisfies this invariant:

Let $m(i,k)$ be the largest integer $j \geq i$ such that $F_{i,j} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1$. For $T_1$ we set $T_1[k] = m(1,k)$ for all $k = 1, \ldots, t$. For the remaining $T_i$ we set

$$T_i[k] = \begin{cases} T_{i-1}[k] & \text{if } F_{i,T_{i-1}[k]} \geq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil + 1 \\ m(i,k) & \text{otherwise} \end{cases}$$

The $n$ lists are sorted by construction. For $T_1$, it is true since $m(i,k)$ is increasing in $k$. For $T_i$, it follows that $F_{i,T_i[k]} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1 < F_{i,T_i[k+1]}$, and thus $T_i[k] < T_i[k+1]$ for any $k$.

Let $s$ be the index of the successor of $j$ in $T_i$. We know that $F_{i,T_i[s]} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{s+1} \rceil - 1$, $F_{i,T_i[s-1]} \geq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{s-1} \rceil + 1$ and $T_i[s-1] < j \leq T_i[s]$. It follows that

$$\lceil \tfrac{1}{\varepsilon}(1+\varepsilon)^{s-1} \rceil + 1 \leq F_{i,j} \leq \lceil \tfrac{1}{\varepsilon}(1+\varepsilon)^{s+1} \rceil - 1 \tag{5.1}$$

and that $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^s \rceil$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$.

The second important property of the $n$ lists, is that they only store $O(\frac{n}{\varepsilon})$ different indices, which allows for a space-efficient representation. If $T_{i-1}[k] \neq T_i[k]$ then the following $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1 - \lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil - 1 \geq \lfloor (1+\varepsilon)^k \rfloor - 3$ entries, $T_{i+a}[k]$ for $a = 1, \ldots, \lfloor (1+\varepsilon)^k \rfloor - 3$, are not changed, hence we store the same index at least $\max\{1, \lfloor (1+\varepsilon)^k \rfloor - 2\}$ times. Therefore, the number of changes to the $n$ lists, starting with $T_1$, is bounded by

$$\sum_{k=1}^{t} \frac{n}{\max\{1, \lfloor (1+\varepsilon)^k \rfloor - 2\}} = O(\tfrac{n}{\varepsilon}) \ .$$

This was observed in [22], where similar lists are maintained in a partially persistent search tree [33]. This data structure uses $O(\frac{n}{\varepsilon})$ space and supports queries in $O(\log \log_{1+\varepsilon} n)$ time.

We maintain these lists without persistence such that we can access any entry in any list $T_i$ in constant time. Let $I = \{1, 1+t, \ldots, 1+\lfloor (n-1)/t \rfloor t\}$. For every $\ell \in I$ we store $T_\ell$ explicitly as an array $S_\ell$. Secondly, for $\ell \in I$ and $k = 1, \ldots, \lceil \log_{1+\varepsilon} t \rceil$ we define a bit vector $B_{\ell,k}$ of length $t$ and a change list $C_{\ell,k}$, where

$$B_{\ell,k}[a] = \begin{cases} 0 & \text{if } T_{\ell+a-1}[k] = T_{\ell+a}[k] \\ 1 & \text{otherwise} \end{cases}$$

Given a bit vector $L$, define $\mathrm{sel}(L, b)$ as the index of the $b$'th one in $L$. We set

$$C_{\ell,k}[a] = T_{\ell+\mathrm{sel}(B_{\ell,k},a)}[k] \ .$$

Finally, for every $\ell \in I$ and for $k = 1+\lceil \log_{1+\varepsilon} t \rceil, \ldots, t$ we store $D_\ell[k]$ which is the smallest integer $z > \ell$ such that $T_z[k] \neq T_\ell[k]$. We also store $E_\ell[k] = T_{D_\ell[k]}[k]$. We store each bit vector in a rank and select data structure [56] that uses $O(\frac{n}{w})$ space for a bit vector of length $n$, and supports $rank(i)$ in constant time. A $\mathrm{rank}(i)$ query returns the number of ones in the first $i$ bits of the input.

Each change list, $C_{l,k}$ and every $D_\ell$ and $E_\ell$ list is stored as an array. The bit vectors indicate at which indices the contents of the first $\lceil \log_{1+\varepsilon} t \rceil$ entries of $T_\ell, \ldots, T_{\ell+t-1}$ change, and the change lists store what the entries change to. The $D_\ell$ and $E_\ell$ arrays do the same thing for the last $t - \lceil \log_{1+\varepsilon} t \rceil$ entries, exploiting that these entries change at most once in an interval of length $t$.

Observe that the arrays, $C_{\ell,k}, D_\ell[k]$ and $E_\ell[k]$, and the bit vectors, $B_{\ell,k}$ allow us to retrieve the contents of any entry, $T_i[k]$ for any $i, k$, in constant time as follows. Let $\ell = \lfloor i/t \rfloor t$. If $k > \lceil \log_{1+\varepsilon} t \rceil$ we check if $D_\ell[k] \leq i$, and if so we return $E_\ell[k]$, otherwise we return $S_\ell[k]$. If $k \leq \lceil \log_{1+\varepsilon} t \rceil$, we determine $r = \mathrm{rank}(i-\ell)$ in $B_{\ell,k}$ using the rank and select data structure. We then return $C_{\ell,k}[r]$ unless $r = 0$ in which case we return $S_\ell[k]$.

We argue that this correctly returns $T_i[k]$. In the case where $k > \lceil \log_{1+\varepsilon} t \rceil$, comparing $D_\ell[k]$ to $i$ indicates whether $T_i[k]$ is different from $T_\ell[k]$. Since $T_z[k]$ for $z = \ell, \ldots, i$ can only change once, $T_i[k] = E_\ell[k]$ in this case. Otherwise, $S_\ell[k] = T_\ell[k] = T_i[k]$. If $k \leq \lceil \log_{1+\varepsilon} t \rceil$, the rank $r$ of $i - \ell$ in $B_{\ell,k}$, is the number of changes that has occurred in the $k$'th entry from list $T_\ell$ to $T_i$. Since $C_{\ell,k}[r]$

stores the value of the $k$'th entry after the $r$'th change, $C_{\ell,k}[r] = T_i[k]$, unless $r = 0$ in which case $T_i[k] = S_\ell[k]$.

The space used by the data structure is $O(\frac{n}{\varepsilon})$. We store $3\lceil\frac{n}{t}\rceil$ arrays, $S_\ell$, $D_\ell$ and $E_\ell$ for $\ell \in I$, each using $t$ space, in total $O(n)$. The total size of the change lists, $C_{\ell,k}$, is bounded by the number of changes across the $T_i$ lists, which is $O(\frac{n}{\varepsilon})$ by the arguments above. Finally, the rank and select data structures, $B_{\ell,k}$, each occupy $O(\frac{t}{w}) = O(\frac{t}{\log n})$ words, and we store a total of $\lceil\frac{n}{t}\rceil\lceil\log_{1+\varepsilon} t\rceil$ such structures, thus the total space used by these is bounded by

$$\begin{aligned} & O\left(\frac{t}{\log n}\frac{n}{t}\log_{1+\varepsilon} t\right) = O\left(n\frac{\log_{1+\varepsilon} t}{\log n}\right) = O\left(\frac{n}{\varepsilon}\frac{\log t}{\log n}\right) \\ = \; & O\left(\frac{n}{\varepsilon}\frac{\log\frac{\log(\varepsilon n)}{\varepsilon}}{\log n}\right) = O\left(\frac{n}{\varepsilon}\frac{\log(n\log(\varepsilon n))}{\log n}\right) = O\left(\frac{n}{\varepsilon}\right) . \end{aligned}$$

In the last line we used that if $\lceil\frac{1}{\varepsilon}\rceil \geq n$ then we only store the small frequency data structure. We conclude that our data structures uses $O\left(\frac{n}{\varepsilon}\right)$ space.

To answer a query $[i, j]$, we first compute a 3-approximation of $F_{i,j}$ in constant time using the data structure from Section 5.3. Thus, we find $f_{i,j}$ satisfying $f_{i,j} \leq F_{i,j} \leq 3f_{i,j}$. Choose $k$ such that $\lceil\frac{1}{\varepsilon}(1+\varepsilon)^k\rceil + 1 \leq f_{i,j} \leq \lceil\frac{1}{\varepsilon}(1+\varepsilon)^{k+1}\rceil - 1$ then the successor of $j$ in $T_i$ must be in one of the entries, $T_i[k], \ldots, T_i[k + O(\log_{1+\varepsilon} 3)]$. As stated earlier, the values of $T_i$ are sorted in increasing order, and we find the successor of $j$ using a binary search on an interval of length $O(\log_{1+\varepsilon} 3)$. Since each access to $T_i$ takes constant time, we use $O(\log\log_{1+\varepsilon} 3) = O(\log\frac{1}{\varepsilon})$ time.

**Theorem 5.7** *There exists a data structure for $(1+\varepsilon)$-approximate range mode that uses $O(\frac{n}{\varepsilon})$ space and supports queries in $O(\log\frac{1}{\varepsilon})$ time.*

The careful reader may have noticed that our data structure returns a frequency, and not a label that occurs approximately $F_{i,j}$ times. We can augment our data structure to return a label instead as follows.

We set $\varepsilon' = \sqrt{(1+\varepsilon)} - 1$, and construct our data structure from above. The *small frequency* data structure is augmented such that it stores the label $M_{i,Q_i[k]}$ along with $Q_i[k]$, and returns this in a query. The *large frequency* data structure is augmented such that for every update of $T_i[k]$ we store the label that caused the update. Formally, let $a > 0$ be the first index such that $T_{i+a}[k] \neq T_i[k]$. Next to $T_i[k]$ we store the label $L_i[k] = A[i+a-1]$. In a query, $[i, j]$, let $s$ be the index of the successor of $j$ in $T_i$ computed as above. If $s > 1$ we return the label $L_i[s-1]$, and if $s = 1$ we return $M_{i,Q_i[\lceil 1/\varepsilon'\rceil]}$, which is stored in the small frequency data structure.

In the case where $s = 1$ we know that $\lceil\frac{1}{\varepsilon'}\rceil \leq F_{i,j} \leq \lceil\frac{1}{\varepsilon'}(1+\varepsilon')^2\rceil - 1 = \lceil\frac{1}{\varepsilon'}(1+\varepsilon)\rceil - 1$ and we know that the frequency of $M_{i,Q_i[\lceil 1/\varepsilon'\rceil]}$ in $A[i, j]$ is at least $\lceil\frac{1}{\varepsilon'}\rceil$. We conclude that the frequency of $M_{i,Q_i[\lceil 1/\varepsilon'\rceil]}$ in $A[i, j]$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$.

In the case where $s > 1$, we know that $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{s-1}\rceil + 1 \leq F_{i,j} \leq \lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{s+1}\rceil - 1$ by equation (5.1), and that the frequency, $f_L$, of the label $L_i[s-1]$ in $A[i, j]$ is at least $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{s-1}\rceil + 1$. This means that

$F_{i,j} \leq \frac{1}{\varepsilon'}(1+\varepsilon')^{s+1} \leq (1+\varepsilon')^2 f_L = (1+\varepsilon) f_L$, and we conclude that $f_L$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$.

The space needed for this data structure is $O(\frac{n}{\varepsilon'}) = O(\frac{n(\sqrt{1+\varepsilon}+1)}{\varepsilon}) = O(\frac{n}{\varepsilon})$, and a query takes $O(\log \frac{1}{\varepsilon'}) = O(\log \frac{1}{\varepsilon} + \log(\sqrt{1+\varepsilon}+1)) = O(\log \frac{1}{\varepsilon})$ time.

# Part III

# Fault Tolerant Algorithms

# Chapter 6

## Priority Queue Resilient to Memory Faults

In this chapter we design and analyze a priority queues in the faulty-memory RAM model.

**Our Results** We design a data structure that uses $O(n)$ space for storing $n$ elements and performs both INSERT and DELETEMIN in $O(\log n + \delta)$ time amortized. Our priority queue matches the bounds for an optimal comparison based priority queue in the RAM model while tolerating $O(\log n)$ corruptions. It is a significant improvement over using the resilient search tree in [38] as a priority queue, since it uses $O(\log n+\delta^2)$ time amortized per operation and thus only tolerates $O(\sqrt{\log n})$ corruptions to preserve the $O(\log n)$ bound per operation. Our priority queue is the first resilient data structure allowing $O(\log n)$ corruptions, while still matching optimal bounds in the RAM model. Our priority queue does not store elements in reliable memory between operations, only structural information like pointers and indices. We prove that any comparison based resilient priority queue behaving this way requires worst case $\Omega(\log n+\delta)$ time for either INSERT or DELETEMIN.

The resilient priority queue is based on the cache-oblivious priority queue by Arge *et al.* [9]. The main idea is to gather elements in large sorted groups of increasing size, such that expensive updates do not occur too often. The smaller groups contain the smaller elements, so they can be retrieved faster by DELETEMIN operations. We extensively use the resilient merging algorithm in [36] to move elements among the groups. Due to the large sizes of the groups, the extra work required to deal with corruptions in the merging algorithm becomes insignificant compared to the actual work done.

**Outline** The remainder of the paper is structured as follows. We give a detailed description of the resilient priority queue in Section 6.1, while in Section 6.2 we prove its correctness and complexity bounds. Finally, in Section 6.3 we prove matching lower bounds for resilient priority queues.

**Preliminaries** Given two sequences $X$ and $Y$, we let $XY$ denote the *concatenation* of $X$ and $Y$. A sequence $X$ is *faithfully ordered* if its uncorrupted keys appear in non-decreasing order.

**Definition 6.1** *A resilient priority queue maintains a set of elements under the operations* Insert *and* Deletemin*. An* Insert *adds an element and a* Deletemin *deletes and returns the minimum uncorrupted element or a corrupted one.*

We note that our definition of a resilient priority queue is consistent with the resilient sorting algorithms introduced in [39]. Given a sequence of $n$ elements, inserting all of them into a resilient priority queue followed by $n$ Deletemin operations yields a faithfully ordered sequence.

## 6.1 Fault Tolerant Priority Queue

In this section we introduce the resilient priority queue. It resembles the cache-oblivious priority queue by Arge *et al.* [9]. The elements are stored in faithfully ordered lists and are moved using two fundamental primitives, Push and Pull, based on faithful merging. We describe the structure of the priority queue in Section 6.1.1 and then introduce the Push and Pull primitives in Section 6.1.2. Finally, in Section 6.1.3, we describe the Insert and Deletemin operations.

### 6.1.1 Structure

The resilient priority queue consists of an insertion buffer $I$ together with a number of layers $L_0, \ldots, L_k$, with $k = O(\log n)$. Each layer $L_i$ contains an up-buffer $U_i$ and a down-buffer $D_i$, represented as arrays. Intuitively, the up-buffers contain large elements that are on their way to the upper layers in the priority queue, whereas the down-buffers contain small elements, on their way to lower layers. The buffers in the priority queue are stored as a doubly linked list $U_0, D_0, \ldots, U_k, D_k$, see Figure 6.1. For each up and down buffer we reliably store the pointers to their adjacent buffers in the linked list and their size. In the reliable memory we store pointers to $I$, $U_0$ and $D_0$, together with $|I|$. Since the position of the first element in $U_0$ and $D_0$ is not always the first memory cell of the corresponding buffer, we also store the index of the first element in these buffers in reliable memory. The insertion buffer $I$ contains up to $b = \delta + \log n + 1$ elements. For layer $L_i$ we define the threshold $s_i$ by $s_0 = 2 \cdot (\delta^2 + \log^2 n)$ and $s_i = 2s_{i-1} = 2^{i+1} \cdot (\delta^2 + \log^2 n)$, where $n$ is the number of elements in the priority queue. We use these thresholds to decide whether an up buffer contains too many elements or whether a down buffer has too few. For the sake of simplicity, the up and down buffers are grown and shrunk as needed during the execution such that they don't use any extra space.

To structure the priority queue, we maintain the following invariants for the up and down buffers.

- *Order invariants:*
  1. All buffers are faithfully ordered.
  2. $D_iD_{i+1}$ and $D_iU_{i+1}$ are faithfully ordered, for $0 \leq i < k$.

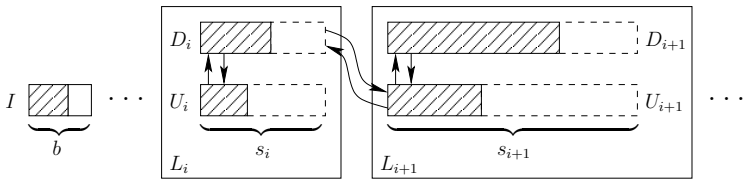



Figure 6.1: The structure of the priority queue. The buffers are stored in a doubly linked list using reliably stored pointers. Additionally, the size of each buffer is stored reliably.

- *Size invariants:*
  3. $s_i/2 \leq |D_i| \leq s_i$, for $0 \leq i < k$.
  4. $|U_i| \leq s_i/2$, for $0 \leq i < k$.

By maintaining all the up and down buffers faithfully ordered, it is possible to move elements between neighboring layers efficiently, using faithful merging. By invariant 2, all uncorrupted elements in $D_i$ are smaller than all uncorrupted elements in both $D_{i+1}$ and $U_{i+1}$. This ensures that small elements belong to the lower layers of the priority queue. We note that there is no assumed relationship between the elements in the up and down buffers in the same layer. Finally, the size invariants allow the sizes of the buffers to vary within a large range. This way, $\Omega(s_i)$ INSERT or DELETEMIN operations occur between two operations on the same buffer in $L_i$, yielding the desired amortized bounds.

Since the $s_i$ values depend on $n$, whenever the size of the priority queue increases or decreases by $\Theta(n)$, we perform a global rebuilding. This rebuilding is done by collecting all elements, sorting them with an optimal resilient sorting algorithm [36], and redistributing the output into the down buffers of all the layers starting with $L_0$. After the global rebuilding, the up buffers are empty and the down buffers full, except possibly the last down buffer.

### 6.1.2 Push and Pull Primitives

We now introduce the two fundamental primitives used by the priority queue. The PUSH primitive is invoked when an up buffer contains too many elements, breaking invariant 4. It "pushes" elements upwards, repairing the size invariants locally. The PULL operation is invoked when a down buffer contains too few elements, breaking invariant 3. It fills this down buffer by "pulling" elements from the layer above, again locally repairing the size invariants. Both operations faithfully merge consecutive buffers in the priority queue and redistribute the resulting sequence among the participating buffers. After merging, we deallocate the old buffers and allocate new arrays for the new buffers.

**Push** The PUSH primitive is invoked when an up buffer $U_i$ breaks invariant 4, *i.e.* when it contains more than $s_i/2$ elements. In this case we merge $U_i$, $D_i$ and $U_{i+1}$ into a sequence $M$ using the resilient merging algorithm in [36]. We then distribute the elements in $M$ by placing the first $|D_i| - \delta$ elements in a new

buffer $D'_i$, and the remaining $|U_{i+1}| + |U_i| + \delta$ elements in a new buffer $U'_{i+1}$. After the merge, we create an empty buffer, $U'_i$, and deallocate the old buffers. If $U'_{i+1}$ contains too many elements, breaking invariant 4, the PUSH primitive is invoked on $U'_{i+1}$. When $L_i$ is the last layer, we fill $D'_i$ with the first elements of $M$ and create a new layer $L_{i+1}$ placing the remaining elements of $M$ into $D'_{i+1}$ instead of $U'_{i+1}$. Since $|D'_i|$ is smaller than $|D_i|$, it could violate invariant 3. This situation is handled by using the PULL operation and is described after introducing PULL.

Unlike the priority queue in [9], the PUSH operation decreases the size of a down buffer. This is required to preserve invariant 2, in spite of corruptions. After a PUSH call, $D'_i$ can contain elements from $U_i \cup U_{i+1}$. Since there is no assumed relationship between elements in $U_i \cup U_{i+1}$ and those in $D_{i+1} \cup U_{i+2}$, we need to ensure that each element in $D'_i$ originating from $U_i \cup U_{i+1}$ is faithfully smaller than the elements in $D_{i+1} \cup U_{i+2}$. Assume the size of $D_i$ is preserved, *i.e.* $|D'_i| = |D_i|$. Consider a corruption that alters an element in $D_i$ to some large value before the PUSH. This corrupted value could be placed in $U'_{i+1}$ and, since $|D'_i| = |D_i|$, an element from $U_i \cup U_{i+1}$ must be placed in $D'_i$. This new element in $D'_i$ potentially violates invariant 2.

**Pull** The PULL operation is called on a down buffer $D_i$ when it contains less than $s_i/2$ elements, breaking invariant 3. In this case, the buffers $D_i$, $U_{i+1}$, and $D_{i+1}$ are merged into a sequence $M$ using the resilient merging algorithm in [36]. The first $s_i$ elements from $M$ are written to a new buffer $D'_i$, and the next $|D_{i+1}| - (s_i - |D_i|) - \delta$ elements are written to $D'_{i+1}$. The remaining elements of $M$ are written to $U'_{i+1}$. A PULL is invoked on $D'_{i+1}$, if it is too small.

Similar to the PUSH operation, the extra $\delta$ elements lost by $D_{i+1}$ ensure that the order invariants hold in spite of possible corruptions. That is, a corruption of an element in $D_i \cup D_{i+1}$ to a very large value may cause an element from $U_{i+1}$ to take the place of the corrupted element in $D'_{i+1}$ and this element is possibly larger than some uncorrupted element in $D_{i+2} \cup U_{i+2}$.

After the merge, $U'_{i+1}$ contains $\delta$ more elements than $U_{i+1}$ had before the merge, and thus it is possible that it has too many elements, breaking invariant 4. We handle this situation as follows. Consider a maximal series of subsequent PULL invocations on down buffers $D_i, D_{i+1}, \ldots, D_j$, $0 \leq i < j < k$. After the first PULL call on $D_i$ and before the call on $D_{i+1}$ we store a pointer to $D_i$ in the reliable memory. After all the PULL calls we investigate all the affected up buffers, by simply following the pointers between the buffers starting from $D_i$, and invoke the PUSH primitive wherever necessary. The case when PUSH operations cause down buffers to underflow is handled similarly.

### 6.1.3 Insert and Deletemin

An element is inserted in the priority queue by simply appending it to the insertion buffer $I$. If $I$ gets full, its elements are added to $U_0$ by first faithfully sorting $I$ and then faithfully merging $I$ and $U_0$. If $U_0$ breaks invariant 4, we invoke the PUSH primitive. If $L_0$ is the only layer of the priority queue and $D_0$

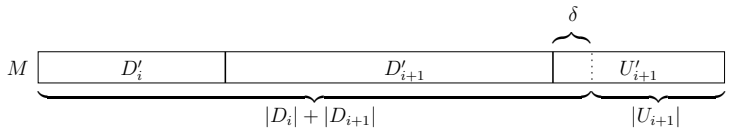



Figure 6.2: The distribution of $M$ into buffers.

violates the size constraint, we faithfully merge the elements in $I$ with $D_0$ instead.

To delete the minimum element in the priority queue, we first find the minimum of the first $\delta + 1$ values in $D_0$, the minimum of the first $\delta + 1$ values in $U_0$, and the minimum element in $I$. We then take the minimum of these three elements, delete it from the appropriate buffer and return it. After deleting the minimum, we right-shift all the elements in the affected buffer from the beginning up to the position of the minimum. This way we ensure that elements in any buffer are stored consecutively. If $D_0$ underflows, we invoke the Pull primitive on $D_0$, unless $L_0$ is the only layer in the priority queue. If $U_0$ or $D_0$ contains $\Theta(\log n + \delta)$ empty cells, we create a new buffer and copy the elements from the old buffer to the new one.

## 6.2 Analysis

In this section we analyze the resilient priority queue. We prove the correctness in Section 6.2.1 and analyze the time and space complexity in Section 6.2.2.

### 6.2.1 Correctness

To prove correctness of the resilient priority queue, we show that the DeleteMin operation returns the minimum uncorrupted value or a corrupted value. We first prove that the order invariants are maintained by the Pull and Push operations.

**Lemma 6.1** *The* Pull *and* Push *primitives preserve the order invariants.*

*Proof.* Recall that in a Pull invocation on buffer $D_i$, the buffers $D_i$, $U_{i+1}$, and $D_{i+1}$ are faithfully merged into a sequence $M$. The elements in $M$ are then distributed into three new buffers $D'_i$, $U'_{i+1}$, and $D'_{i+1}$, see Figure 6.2. To argue that the order invariants are satisfied we need to show that the elements of the down buffer on layer $L_j$, for $0 \leq j < k$, are faithfully smaller than the elements of the buffers on layer $L_{j+1}$, where $k$ is the index of the last layer. The invariants hold trivially for unaffected buffers. The faithful merge guarantees that $D'_iD'_{i+1}$ as well as $D'_iU'_{i+1}$ are faithfully ordered, and thus the individual buffers are also faithfully ordered. Since invariant 2 holds for the original buffers all uncorrupted elements in $D_{i+1}$ and $U_{i+1}$ are larger than the uncorrupted elements in $D_i$, guaranteeing that $D_{i-1}D'_i$ is faithfully ordered. Finally, we now show that $D_{i+1}D_{i+2}$ and $D_{i+1}U_{i+2}$ are faithfully ordered.

Let $m$ be the minimum uncorrupted element in $D_{i+2} \cup U_{i+2}$. We need to show that all uncorrupted elements in $D'_{i+1}$ are smaller than $m$. If no uncorrupted element from $U_{i+1}$ is placed in $D'_{i+1}$, the invariant holds by the order invariants before the operation. Otherwise, assume that an uncorrupted element $y \in U_{i+1}$ is moved to $D'_{i+1}$. Since $|U'_{i+1}| = |U_{i+1}| + \delta$ and $y$ is moved to $D'_{i+1}$, at least $\delta + 1$ elements originating from $D_i \cup D_{i+1}$ are contained in $U'_{i+1}$. Since there can be at most $\delta$ corruptions, there exists at least one uncorrupted element, $x$, among these. By faithful merging, all uncorrupted elements in $D'_{i+1}$ are smaller than $x$, which means that $y \leq x$. Since $x$ originates from $D_i \cup D_{i+1}$, it is smaller than $m$. We obtain $y \leq m$.

A similar argument proves correctness of the PUSH operation. We conclude that both order invariants are preserved by PULL and PUSH operations.

□

Having proved that the order invariants are maintained at all times, we now prove the correctness of the resilient priority queue.

**Lemma 6.2** *The* DELETEMIN *operation returns the minimum uncorrupted value in the priority queue or a corrupted value.*

*Proof.* We recall that the DELETEMIN operation computes the minimum of the first $\delta + 1$ elements of $U_0$ and $D_0$. It compares these values with the minimum of $I$, found in a scan, and returns the smallest of these elements. Since $U_0$ and $D_0$ are faithfully ordered, the minimum of their first $\delta + 1$ elements is either the minimum uncorrupted value in these buffers, or a corrupted value even smaller. Furthermore, according to the order invariants, all the values in layers $L_1, \ldots, L_k$ are faithfully larger than the minimum in $D_0$. Therefore, the element reported by DELETEMIN is the minimum uncorrupted value or a corrupted value. □

### 6.2.2 Complexity

In this section we show that our resilient priority queue uses $O(n)$ space and that INSERT and DELETEMIN take $O(\log n + \delta)$ amortized time. We first prove that the PULL and PUSH primitives restore the size invariants.

**Lemma 6.3** *If a size invariant is broken for a buffer in $L_0$, invoking* PULL *or* PUSH *on that buffer restores the invariants. Furthermore, during this operation* PULL *and* PUSH *are invoked on the same buffer at most once. No other invariants are broken before or after this operation.*

*Proof.* Assume that PUSH is invoked on $U_0$, and that it is called iteratively up to some layer $L_l$. By construction of PUSH, the size invariants for all the up buffers now hold. Since a PUSH steals $\delta$ elements from the down buffers, the layers $L_0, \ldots, L_l$ are traversed again and PULL is invoked on these as needed. The last of these PULL operations might proceed past layer $L_l$. Similarly, a PULL may cause an up buffer to overflow. However, since the cascading PUSH

operations left $|U_i| = 0$ for $i \leq l$, any new PUSH are invoked on up buffers only on layer $L_{l+1}$ or higher, thus PUSH is invoked on each buffer at most once. A similar argument works for the PULL operation.

□

**Lemma 6.4** *The resilient priority queue uses $O(n + \delta)$ space to store $n$ elements.*

*Proof.* The insertion buffer always uses $O(\log n + \delta)$ space. We prove that the remaining layers use $O(n)$ space. For each layer we use $O(\delta)$ space for storing structural information reliably. In all layers, except the last one, the down buffer contains $\Omega(\delta^2)$ elements by invariant 3. This means that for each of these layers the elements stored in the down buffer dominate the space complexity. The structural information of the last layer requires additional $O(\delta)$ space. □

The space complexity of the priority queue can be reduced to $O(n)$ without affecting the time complexity, by storing the structural information of $L_0$ in safe memory, and by doubling or halving the insertion buffer during the lifetime of the algorithm such that it always uses $O(|I|)$ space.

**Lemma 6.5** *Each* INSERT *and* DELETEMIN *takes $O(\log n + \delta)$ amortized time.*

*Proof.* We define the potential function:

$$\Phi = \sum_{i=1}^{k} (c_1 \cdot (\log n - i) \cdot |U_i| + c_2 \cdot i \cdot |D_i|)$$

We use $\Phi$ to analyze the amortized cost of a PUSH operation. In a PUSH operation on $U_i$, buffers $U_i$, $D_i$, and $U_{i+1}$ are merged. The elements are then distributed into new buffers $U_i'$, $D_i'$, and $U_{i+1}'$, such that $|U_i'| = 0$, $|D_i'| = |D_i| - \delta$, and $|U_{i+1}'| = |U_{i+1}| + |U_i| + \delta$. This gives the following change in potential $\Delta\Phi$:

$$\begin{aligned} \Delta\Phi &= -|U_i| \cdot c_1 \cdot (\log n - i) - \delta \cdot c_2 \cdot i + (|U_i| + \delta) \cdot c_1(\log n - (i+1)) \\ &= -c_1 \cdot |U_i| + \delta(-c_2 \cdot i + c_1 \cdot \log n - c_1 \cdot i - c_1) \,. \end{aligned}$$

Since the PUSH is invoked on $U_i$, invariant 4 is not valid for $U_i$ and therefore $|U_i| \geq \frac{s_i}{2} = 2^i (\log^2 n + \delta^2)$. Thus:

$$\Delta\Phi \leq -c_1 \cdot |U_i| + c_1 \cdot \delta \cdot \log n \leq -c_1 \cdot 2^i \cdot (\log^2 n + \delta^2) + c_1 \cdot \delta \cdot \log n \leq -c_1 \cdot c' \cdot |U_i| \,, \tag{6.1}$$

for some constant $c' > 0$.

Since faithfully merging two sequences of size $n$ takes $O(n + \delta^2)$ time [36], the time used for a PUSH on $U_i$ is upper bounded by $c_m \cdot (|U_i| + |D_i| + |U_{i+1}| + \delta^2)$, where $c_m$ depends on the resilient merge. This includes the time required for retrieving reliably stored variables. Adding the time and the change in potential we are able to get the amortized cost less than zero by tweaking $c_1$ based on equation (6.1). This is because $|U_i|$ is $\Omega(\delta^2)$ and at most a constant fraction smaller than the participants in the merge.

A similar analysis works for the PULL primitive. We now calculate the amortized cost of INSERT and DELETEMIN. We ignore any PUSH or PULL operations since their amortized costs are negative. The amortized time for inserting an element in $I$, sorting $I$, and merging it with $U_0$ is $O(\log n + \delta)$ per operation. The change in potential when adding elements to $L_0$ is $O(\log n)$ per element. The time needed to find the smallest element in a DELETEMIN is $O(\log n + \delta)$, and the change in potential when an element is deleted from $L_0$ is negative.

The cost of global rebuilding is dominated by the cost of sorting, which is $O(n \log n + \delta^2)$. There are $\Theta(n)$ operations between each rebuild, which leads to $O(\log n + \delta)$ time per operation, since $\delta \leq n$, and this concludes the proof. □

**Theorem 6.1** *The resilient priority queue takes $O(n)$ space and uses amortized $O(\log n + \delta)$ time per operation.*

## 6.3 Lower bound

In this section we prove that any resilient priority queue takes $\Omega(\log n + \delta)$ time for either INSERT or DELETEMIN in the comparison model, under the assumption that no elements are stored in reliable memory between operations. This implies optimality of our resilient priority queue under these assumptions. We note that the reliable memory may contain any structural information, e.g. pointers, sizes, indices.

**Theorem 6.2** *A resilient priority queue containing $n$ elements, with $n > \delta$, uses $\Omega(\log n + \delta)$ comparisons to perform an* INSERT *followed by* DELETEMIN.

*Proof.* Consider a priority queue $Q$ with $n$ elements, with $n > \delta$, that uses less than $\delta$ comparisons for an INSERT followed by a DELETEMIN. Also, $Q$ does not store elements in reliable memory between operations. Assume that no corruptions have occurred so far. Without loss of generality we assume that all the elements in $Q$ are distinct. We prove there exists a series of corruptions $C$, $|C| \leq \delta$, such that the result of an INSERT of an element $e$ followed by a DELETEMIN returns the same element regardless of the choice of $e$.

Let $k < \delta$ be the number of comparisons performed by $Q$ during the two operations. We force the result of each comparison to be the same regardless of $e$ by suitable corruptions. In all the comparisons involving $e$, we ensure that $e$ is the smallest. We do so by corrupting the value which $e$ is compared against if necessary, by adding some positive constant $c \geq e$ to the other value. If two elements different than $e$ are compared, we make sure the outcome is the same as if no corruptions had happened. If one of them was corrupted, adding $c$ to the other one reestablishes their previous ordering. If both of them were corrupted by adding $c$, their ordering is unchanged and no corruptions are needed. Forcing any comparison to give the desired outcome requires at most one corruption, and therefore $|C| \leq k < \delta$.

We now consider the value $e'$ returned by DELETEMIN on $Q$. If $e = e'$ then we choose $e$ to be larger than some element $x \in Q$ not affected by a corruption in $C$. Such a value exists because the size of the priority queue is larger than $\delta$. Since $e = e' > x$, $Q$ returned an uncorrupted element that was not the minimum uncorrupted element in $Q$. If $e \neq e'$ we choose $e$ to be smaller than any element in $Q$. With such a choice of $e$, no corruptions are required and the value returned by $Q$ was not corrupted, but still larger than $e$. This proves $Q$ is not resilient.

Adding the classical $\Omega(\log n)$ bound for priority queues in the comparison model the result follows. □

# Chapter 7

## Optimal Resilient Dictionaries

In this chapter we investigate comparison based search algorithm in the faulty memory RAM. A resilient searching algorithm must return a positive answer if there exists an uncorrupted element in the input equal to the search key. If there is no element, corrupted or uncorrupted, matching the search key, the algorithm must return a negative answer. If there is a corrupted value equal to the search key, the answer can be both positive or negative.

**Our Results** We propose two optimal resilient static dictionaries, a randomized one and a deterministic one, as well as a dynamic dictionary.

Randomized static dictionary: We introduce a resilient randomized static dictionary that support searches in $O(\log n + \delta)$ time, matching the bounds for randomized searching in [36]. We note however that our dictionary is somewhat simpler and uses only $O(\log \delta)$ worst case random bits, whereas the algorithm in [36] uses expected $O(\log \delta \cdot \log n)$ random bits. On the downside, our dictionary assumes that the corruptions are performed by a non-adaptive adversary, i.e. an adversary that does not perform corruptions based on the behavior of the algorithm. Given the motivation of the model, i.e. corruptions performed by cosmic rays or alpha particles, the assumption is reasonable.

Deterministic static dictionary: We give the first optimal resilient static deterministic dictionary. It supports searches in a sorted array in $O(\log n + \delta)$ time in the worst case, matching the lower bounds from [39]. Unlike its randomized counterpart, the deterministic dictionary does not make any assumptions regarding the way in which corruptions are performed. Previously, the best deterministic dictionary, supported searches in $O(\log n + \delta^{1+\varepsilon})$ time [36].

Dynamic dictionary: We introduce a deterministic dynamic dictionary that significantly improves over the resilient search trees by Finocchi *et al.* [38]. It supports searches in $O(\log n + \delta)$ in the worst case, and insertions and deletions in $O(\log n + \delta)$ time amortized. Also, it supports range queries in $O(\log n + \delta + k)$ time, where $k$ is the output size.

## 7.1 Optimal Randomized Static Dictionary

In this section we introduce a simple randomized resilient search algorithm. It searches for a given element in a sorted array using worst case $O(\log \delta)$ random bits and expected time $O(\log n+\delta)$, assuming that corruptions are performed by a non-adaptive adversary. The running time matches the algorithm by Finocchi et al. [36], which, however, uses expected $O(\log n \cdot \log \delta)$ random bits. The main idea of our algorithm is to implicitly divide the sorted input array in $2\delta$ disjoint sorted sequences $S_0, \ldots, S_{2\delta-1}$, each of size at most $\lceil n/2\delta \rceil$. The $j$'th element of $S_i$, $S_i[j]$, is the element at position $\text{pos}_i(j) = 2\delta j + i$ in the input array. Intuitively, this divides the input array into $\lceil n/2\delta \rceil$ consecutive *blocks* of size $2\delta$, where $S_i[j]$ is the $i$'th element of the $j$'th block. Note that, since $2\delta$ disjoint sequences are defined from the input array and at most $\delta$ corruptions are possible, at least half of the sorted sequences $S_0, \ldots, S_{2\delta-1}$ do not contain any corrupted elements.

The algorithm generates a random number $k \in \{0, \ldots, 2\delta-1\}$ and performs an iterative binary search on $S_k$. We store in safe memory $k$, the search key $e$, and the left and right indices, $l$ and $r$, used by the binary search. The binary search terminates when $l$ and $r$ are adjacent in $S_k$, and therefore $2\delta$ elements apart in the input array, since $\text{pos}_k(r) - \text{pos}_k(l) = 2\delta$ when $r = l + 1$. If the binary search was not misled by corruptions, then the location of $e$ is between $\text{pos}_k(l)$ and $\text{pos}_k(r)$ in the input array. To check whether the search was misled, we perform the following verification procedure. Consider the neighborhoods $N_l$ and $N_r$, containing the $2\delta + 1$ elements in the input array situated to the left of $\text{pos}_k(l)$ and to the right of $\text{pos}_k(r)$ respectively. We compute the number $s_l = |\{z \in N_l \mid z \leq e\}|$ of elements in $N_l$ that are smaller than $e$ in $O(\delta)$ time by scanning $N_l$. Similarly, we compute the number $s_r$ of elements in $N_r$ that are larger than $e$. If $s_l \geq \delta + 1$ and $s_r \geq \delta + 1$, and the search key is not encountered in $N_l$ or $N_r$, we decide whether it lies in the array or not by scanning the $2\delta - 1$ elements between $\text{pos}_k(l)$ and $\text{pos}_k(r)$. If $s_l$ or $s_r$ is smaller than $\delta + 1$, a corruption has misguided the search. In this case, a new $k$ is randomly selected and the binary search is restarted.

**Theorem 7.1** *The randomized dictionary supports searches in $O(\log n + \delta)$ expected time and uses $O(\log \delta)$ expected random bits.*

*Proof.* We first prove the correctness of the algorithm. Assume that $s_l \geq \delta + 1$ and $e \notin N_l$. Since only $\delta$ corruptions are possible, there exists an uncorrupted element in $N_l$ strictly smaller than $e$. Because the input array is sorted, no uncorrupted elements to the left of $\text{pos}_k(l)$ in the input array are equal to $e$. By a similar argument, if $s_r \geq \delta + 1$ and $e \notin N_r$, then no uncorrupted elements to the right of $\text{pos}_k(r)$ in the input array are equal to $e$. If no corrupted elements are encountered during the binary search, all the uncorrupted elements of $N_l$ are smaller than $e$, and therefore $s_l \geq \delta + 1$. Similarly, we have $s_r \geq \delta + 1$, and the algorithm terminates after scanning the elements between $l$ and $r$.

We now analyze the running time. Each iteration generates a random number $k \in \{0, \ldots, 2\delta - 1\}$, using $O(\log \delta)$ random bits. The sorted sequences

induced by different $k$'s are disjoint, thus at most $\delta$ of them may contain corruptions. Since there are $2\delta$ sorted sequences, the probability of selecting a value $k$ that leads to a corruption-free sequence is at least $1/2$, and therefore the expected number of iterations is at most two. Each iteration uses $O(\log n)$ time for the binary search and $O(\delta)$ time for the verification. We conclude that a search uses expected $O(\log \delta)$ random bits and $O(\log n + \delta)$ expected time. □

We note that for each iteration an adaptive adversary can learn about the subsequence $S_k$ on which we perform the binary search by investigating the elements accessed. Subsequently a single corruption suffices to force the search path to end far enough from its correct position such that the verification fails. In this situation, the algorithm performs $O(\delta)$ iterations and therefore $O(\delta(\log n + \delta))$ time regardless of the random choices of subsequences on which to perform the binary search.

We obtain a worst case bound of $O(\log \delta)$ random bits by using a standard derandomization technique. In the $i$'th iteration we perform the binary search on sequence $S_{h(i)}$, for $h(i) = (r_0 + ir_1 + i^2 r_2 + i^3 r_3) \bmod k$, where $k$ is a prime number with $2\delta \leq k < 4\delta$, and $r_i$ are chosen uniformly at random in $\{0, \ldots, k-1\}$. By construction $h(i)$ is a 4-wise independent hash function [58], which suffices to obtain an expected constant number of iterations for our algorithm [81].

## 7.2 Optimal Static Dictionary

In this section we close the gap between lower and upper bounds for deterministic resilient searching algorithms. We present a resilient algorithm that searches for an element in a sorted array in $O(\log n + \delta)$ time in the worst case, which is optimal [39]. It is an improvement of the previously published best deterministic dictionary, which supports searches in $O(\log n + \delta^{1+\varepsilon})$ time [36]. We reuse the idea presented in the design of the randomized algorithm and define disjoint sorted sequences to be used by a binary search algorithm. Similarly to the randomized algorithm, we design a verification procedure to check the result of the binary search. We design the adapted binary search and the verification procedure such that we are guaranteed to advance only one level in the binary search for each corrupted element misleading the search. We count the number of detected corruptions and adjust our algorithm accordingly to ensure that no element is used more than once, excepting a final scan performed only once on two adjacent blocks. The total time used for verification is $O(\delta)$.

We divide the input array into implicit blocks. Each block consists of $5\delta + 1$ consecutive elements of the input and is structured in three segments: the *left verification segment*, $LV$, consists of the first $2\delta$ elements, the next $\delta + 1$ elements form the *query segment*, $Q$, and the *right verification segment*, $RV$, consists of the last $2\delta$ elements of the block, see Figure 7.1. The left and right verification segments, $LV$ and $RV$, are used only by the verification procedure. The elements in the query segment are used to define the sorted sequences $S_0, \ldots, S_\delta$, similarly to the randomized dictionary previously introduced. The $j$'th element of sequence $S_i$, $S_i[j]$, is the $i$'th element of the query segment of

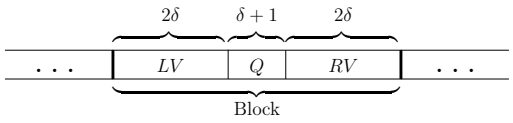



Figure 7.1: The structure of a block. The left and right verification segments, $LV$ and $RV$, contain $2\delta$ elements each, and the query segment $Q$ contains $\delta+1$ elements.

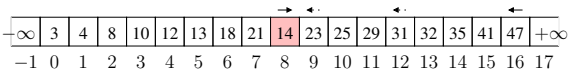



Figure 7.2: Example of binary search on a sequence $S_k$, for the search key 21. The arrows show the direction of the search. The emphasized element is corrupted.

the $j$'th block, and is located at position $\text{pos}_i(j) = (5\delta+1)j + 2\delta + i$ in the input array.

We store a value $k \in \{0, \ldots, \delta\}$ in safe memory identifying the sequence $S_k$ on which we currently perform the binary search. Also, $k$ identifies the number of corruptions detected. Whenever we detect a corruption, we change the sequence on which we perform the search by incrementing $k$. Since there are $\delta + 1$ disjoint sequences, there exists at least one sequence without any corruptions.

**Binary search** The binary search is performed on the elements of $S_k$. Similarly to the randomized algorithm, we store in safe memory the search key, $e$, and the left and right sequence indices, $l$ and $r$, used by the binary search. Initially, $l = -1$ is the position of an implicit $-\infty$ element. Similarly, $r$ is the position of an implicit $\infty$ to the right of the last element. Since each element in $S_k$ belongs to a distinct block, $l$ and $r$ also identify two blocks, $B_l$ and $B_r$.

Each step in the binary search compares the search key $e$ against the element at position $i = \lfloor (l+r)/2 \rfloor$ in $S_k$. Assume without loss of generality that this element is smaller than $e$. We set $l$ to $i$ and decrement $r$ by one. We then compare $e$ with $S_k[r]$. If this element is larger than $e$, the search continues. Otherwise, if no corruptions have occurred, the position of the search element is in block $B_r$ or $B_{r+1}$ in the input array. When two adjacent elements are identified as in the case just described, or when $l$ and $r$ become adjacent, we invoke a verification procedure on the corresponding blocks. The pseudo-code description of the binary search is given in Algorithm 1, and a working example is shown in Figure 7.2.

The verification procedure determines whether the two adjacent blocks, denoted $B_i$ and $B_{i+1}$, are correctly identified. If the verification succeeds, the binary search is completed, and all the elements in the two corresponding adjacent blocks, $B_i$ and $B_{i+1}$ are scanned. The search returns true if $e$ is found during the scan, and false otherwise. If the verification fails, the search may

**Algorithm 1**: Pseudo-code for the binary search procedure.

```
l ← −1
r ←last-block+1
while r − l > 1 do
    i ← ⌈(l+r)/2⌉
    if rep_k(block(i)) < e then
        l ← i
        r ← r − 1
        if rep_k(block(r)) < e then
            if verify(r,r+1) is successful then
                return success
            else
                Backtrack
    else if rep_k(block(i)) > e then
        Similar to previous case.
    else
        return success
if verify(l,r) is successful then
    return success
else
    Backtrack
```

have been misled by corruptions and we backtrack it two steps. To facilitate backtracking, we store two word-sized bit-vectors, $d$ and $f$ in safe memory. The $i$'th bit of $d$ indicates the direction of the search and the $i$'th bit of $f$ indicates whether there was a rounding in computing the middle element in the $i$'th step of the binary search respectively. We can easily compute the values of $l$ and $r$ in the previous step of the binary search by retrieving the relevant bits of $d$ and $f$. If the verification fails, it detects at least one corruption and therefore $k$ is incremented, thus the search continues on a different sequence $S_k$.

**Verification phase** Verification is performed on two adjacent blocks, $B_i$ and $B_{i+1}$. It either determines that $e$ lies in $B_i$ or $B_{i+1}$ or detects corruptions. The verification is an iterative algorithm maintaining a value which expresses the confidence that the search key resides in $B_i$ or $B_{i+1}$. We compute the *left confidence*, $c_l$, which is a value that quantifies the confidence that $e$ is in $B_i$ or to the right of it. Intuitively, an element in $LV_i$ smaller than $e$ is consistent with the thesis that $e$ is in $B_i$ or to the right of it. However an element in $LV_i$ larger than $e$ is inconsistent. Similarly, we compute the *right confidence*, $c_r$, to express the confidence that $e$ is in $B_{i+1}$ or to the left of it.

We compute $c_l$ by scanning a sub-interval of the left verification segment, $LV_i$, of $B_i$. Similarly, the right confidence is computed by scanning the right verification segment, $RV_{i+1}$, of $B_{i+1}$. Initially, we set $c_l = 1$ and $c_r = 1$. We scan $LV_i$ from right to left starting at the element at index $v_l = 2\delta - 2k$ in $LV_i$. Intuitively, by the choice of $v_l$ we ensure that no element in $LV_i$ is accessed

**Algorithm 2**: Pseudo-code for the verification procedure.

```
input : k: Number of errors identified so far
        δ: maximum number of errors
        l:index of the left block
        r:index of the right block
LV ← index of first element in LV_l
RV ← index of first element in RV_r
i_l ← LV + 2δ − 2k
i_r ← RV + 2k
c_r, c_l ← 1
while 0 < min(c_l, c_r) < δ − k + 1 do
    if A[i_l] < e then
        c_l ← c_l + 1
    else
        c_l ← c_l − 1
        k ← k + 1
    if A[i_r] > e then
        c_r ← c_r + 1
    else
        c_r ← c_r − 1
        k ← k + 1
    i_l ← i_l − 1
    i_r ← i_r + 1
if min(c_l, c_r) = 0 then
    return failure
else
    Scan left and right block and return result
```

more than once. Similarly, we scan $RV_{i+1}$ from left to right beginning with the element at position $v_r = 2k$. In an iteration we compare $LV_i[v_l]$ and $RV_{i+1}[v_r]$ against $e$. If $LV_i[v_l] \leq e$, $c_l$ is increased by one, otherwise it is decreased by one and $k$ is increased by one. Similarly, if $RV_{i+1}[v_r] \geq e$, $c_r$ is increased; otherwise, we decrease $c_r$ and increase $k$. The verification procedure stops when $\min(c_r, c_l)$ equals $\delta - k + 1$ or 0. The verification succeeds in the former case, and fails in the latter. The pseudo-code for the verification procedure is introduced in Algorithm 2, and a working example is shown in Figure 7.3.

**Theorem 7.2** *The resilient algorithm searches for an element in a sorted array in $O(\log n + \delta)$ time.*

*Proof.* We first prove that when $c_l$ or $c_r$ decrease during verification, a corruption has been detected. We increase $c_l$ when an element smaller than $e$ is encountered in $LV_i$, and decrease it otherwise. Intuitively, $c_l$ can been seen as the size of a stack $S$. When we encounter an element smaller than $e$, we treat it as if it was pushed, and as if a pop occurred otherwise. Initially, the element $g$ from the query segment of $B_i$ used by the binary search is pushed in $S$. Since $g$ was used to define the left boundary in the binary search, $g < e$ at that time.

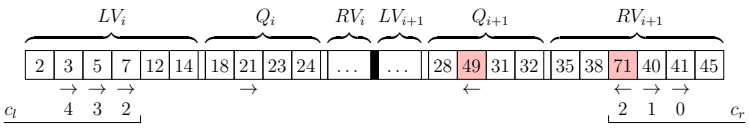



Figure 7.3: A verification step for $\delta = 3$, with $k = 1$ initially. The search key is 45. The verification algorithms stops with $c_r = 0$, reporting failure. The emphasized elements are corrupted.

Each time an element $LV_i[v] < e$ is popped from the stack, it is *matched* with the current element $LV_i[v_l]$. Since $LV_i[v] < e < LV_i[v_l]$ and $v_l < v$, at least one of $LV_i[v_l]$ and $LV_i[v]$ is corrupted, and therefore each match corresponds to detecting at least one corruption. It follows that if $2t-1$ elements are scanned on either side during a failed verification, then at least $t$ corruptions are detected.

We now argue that no single corrupted cell is counted twice. A corruption is detected if and only if two elements are matched during verification. Thus it suffices to argue that no element participates in more than one matching. We first analyze corruptions occurring in the left and right verification segments. Since the verification starts at index $2(\delta - k)$ in the left verification segment and $k$ is increased when a corruption is detected, no element is accessed twice, and therefore not matched twice either. A similar argument holds for the right verification segment. Each failed verification increments $k$, thus no element from a query segment is read more than once. In each step of the binary search both the left and the right indices are updated. Whenever we backtrack the binary search, the last two updates of $l$ and $r$ are reverted. Therefore, if the same block is used in a subsequent verification, a new element from the query segment is read, and this new element is the one initially on the stack. We conclude that elements in the query segments, which are initially placed on the stack, are never matched twice either.

To argue correctness we prove that if a verification is successful, and $e$ is not found in the scan of the two blocks, then no uncorrupted element equal to $e$ exists in the input. If a verification succeeds and $e$ is not found in either block, then $c_l \geq \delta - k + 1$. Since only $\delta - k$ more corruptions are possible, there is at least one uncorrupted element in $LV_i$ smaller than $e$ and thus there can be no uncorrupted elements equal to $e$ to the left of $B_i$ in the input array. By a similar argument, if $c_r \geq \delta - k + 1$, then all uncorrupted elements to the right of $B_{i+1}$ in the input array are larger than $e$.

We now analyze the running time. We charge each backtracking of the binary search to the verification procedure that triggered it. Therefore, the total time of the algorithm is $O(\log n)$ plus the time required by verifications. To bound the time used for all verification steps we use the fact that if $O(f)$ time is used for a verification step, then $\Omega(f)$ corruptions are detected or the algorithm ends. At most $O(\delta)$ time is used in the last verification for scanning the two blocks. □

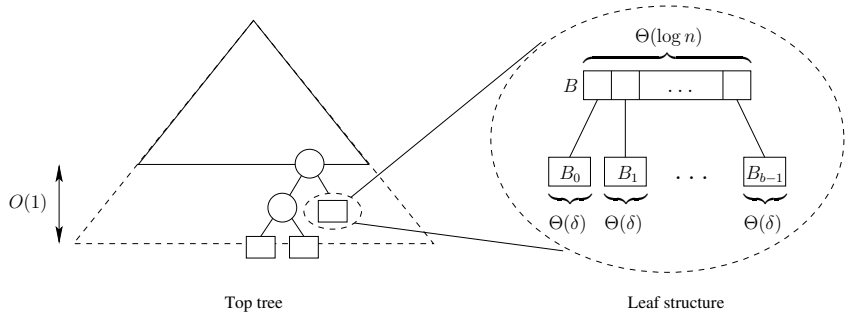



Figure 7.4: The structure of the dynamic dictionary.

## 7.3 Dynamic Dictionary

In this section we describe a linear space resilient deterministic dynamic dictionary supporting searches in optimal $O(\log n + \delta)$ worst case time and range queries in optimal $O(\log n + \delta + k)$ worst case time, where $k$ is the size of the output. The amortized update cost is $O(\log n + \delta)$.

**Structure** The sorted sequence of elements is partitioned into a sequence of *leaf structures*, each storing $\Theta(\delta \log n)$ elements. For each leaf structure we select a guiding element, and we place these $O(n/(\delta \log n))$ guiding elements in the leaves of a reliably stored binary search tree. Each guiding element is chosen such that it is larger than all uncorrupted elements in the corresponding leaf structure.

For this reliable *top tree* $T$, we use the (non-resilient) binary search tree in [24], which consists of $h = \log |T| + O(1)$ levels when containing $|T|$ elements. In the full version [25] it is shown that the tree can be maintained such that the first $h - 2$ levels are complete. We lay the tree in memory in left-to-right breadth first order, as specified in [24]. It uses linear space, and an update costs amortized $O(\log^2 |T|)$ time. A global rebuilding is performed when $|T|$ changes by a constant factor.

All the elements and pointers in the top tree are stored reliably, using replication. Since a reliable value takes $O(\delta)$ space, $O(\delta|T|)$ space is used for the entire structure. The time used for storing and retrieving a reliable value is $O(\delta)$, and therefore the additional work required to handle the reliably stored values increases the amortized update cost to $O(\delta \log^2 |T|)$ time.

The leaf structure consists of a top bucket $B$ and $b$ buckets, $B_0, \ldots, B_{b-1}$, where $\log n \leq b \leq 4 \log n$. Each bucket $B_i$ contains between $\delta$ and $6\delta$ input elements, stored consecutively in an array of size $6\delta$, and uncorrupted elements in $B_i$ are smaller than uncorrupted elements in $B_{i+1}$. For each bucket $B_i$, the top bucket $B$ associates a guiding element larger than all elements in $B_i$, a pointer to $B_i$, and the size of $B_i$, all stored reliably. Since storing a value reliably uses $O(\delta)$ space, the total space used by the top bucket is $O(\delta \log n)$. The guiding elements of $B$ are stored as a sorted array to enable fast searches using the deterministic resilient search algorithm from Section 7.2.

**Lemma 7.1** *The dynamic dictionary uses $O(n)$ space to store $n$ elements.*

*Proof.* Since a leaf structure stores $\Theta(\delta \log n)$ input elements, the top tree contains $O(n/(\delta \log n))$ nodes, using $O(\delta|T|) = O(\delta n/(\delta \log n)) = o(n)$ space. Each of the $O(n/(\delta \log n))$ leaf structures uses $O(\delta \log n)$ space and therefore the total space used for leaf structures is $O(n)$. □

**Searching** The search operation consists of two steps. It first locates a leaf in the top tree $T$, and then searches the corresponding leaf structure. Let $h$ denote the height of $T$. If $h \leq 3$, we perform a standard tree search from the root of $T$ using the reliably stored guiding elements and pointers. Otherwise, we locate two internal nodes, $v_1$ and $v_2$, with guiding elements $g_1$ and $g_2$, such that $g_1 < e \leq g_2$, where $e$ is the search key. Since $h - 2$ is the last complete level of $T$, level $\ell = h - 3$ is complete and contains only internal nodes. The breadth first layout of $T$ ensures that elements of level $\ell$ are stored consecutively in memory. The search operation locates $v_1$ and $v_2$ using the deterministic resilient search algorithm from Section 7.2 on the array defined by level $\ell$. The search only considers the $2\delta + 1$ cells in each node containing guiding elements and ignores memory used for auxiliary information, e.g. sizes and pointers. Although they are stored using replication, the guiding elements are considered as $2\delta + 1$ regular elements in the search. Since the space used by the auxiliary information is the same for all nodes, these gaps in the memory layout of level $\ell$ are easily excluded from the search. We modify the resilient searching algorithm previously introduced such that it reports two consecutive blocks with the property that if the search key is in the structure, it is contained in one of them. The reported two blocks, each of size $5\delta + 1$, span $O(1)$ nodes of level $\ell$ and the guiding elements of these are queried reliably to locate $v_1$ and $v_2$. The appropriate leaf can be in either of the subtrees rooted at $v_1$ and $v_2$, and we perform a standard tree search in both using the reliably stored guiding elements and pointers. Searching for an element in a leaf structure is performed by using the resilient search algorithm from Section 7.2 on the top bucket, $B$, similar to the way $v_1$ and $v_2$ were found in $T$. The corresponding reliably stored pointer is then followed to a bucket $B_i$, which is scanned.

Range queries can be performed by scanning the level $\ell$, starting at $v$, and reporting relevant elements in the leaves below it.

**Lemma 7.2** *The search operation of the dynamic dictionary uses $O(\log n + \delta)$ worst case time. A range query reporting $k$ elements is performed in worst case $O(\log n + \delta + k)$ time.*

*Proof.* The initial search in the top tree takes $O(\log n + \delta)$ worst case time by Theorem 7.2. Traversing the $O(1)$ levels to a leaf takes time $O(\delta)$. Searching in the top bucket of the leaf structures uses $O(\log \log n + \delta)$ time, again using Theorem 7.2. The final scan of a bucket takes time $O(\delta)$.

In a range query, the elements reported in any leaf completely contained in the query range pay for the $O(\delta \log n)$ time used for going through the bottom

part of the top tree and scanning the top bucket. The search pays for the rightmost traversed leaf. □

**Updates** Efficiently updating the structure is performed using standard bucketing techniques. To insert an element into the dictionary, we first perform a search to locate the appropriate bucket $B_i$ in a leaf structure, and then the element is appended to $B_i$ and the size of $B_i$ in the top bucket is updated. When the size of $B_i$ increases to $6\delta$, we split it into two buckets, $B_s$ and $B_g$, of almost equal sizes. We compute a guiding element that splits $B_i$ in $O(\delta^2)$ time by repeatedly scanning $B_i$ and extracting the minimum element. The element $m$ returned by the last iteration is kept in safe memory. In each iteration, we select a new $m$ which is the minimum element in $B_i$ larger than the current $m$. Since at most $\delta$ corruptions can occur, $B_i$ contains at least $2\delta$ uncorrupted elements smaller than $m$ and $2\delta$ uncorrupted elements larger, after $|B_i|/2 = 3\delta$ iterations. The elements from $B_i$ smaller than $m$ are stored in $B_s$, and the remaining ones are stored in $B_g$. The guiding element for $B_s$ is $m$, while $B_g$ preserves the guiding element of $B_i$. The new split element is reliably inserted in the top bucket using an insertion sort step, by scanning and shifting the elements in $B$ from right to left, and placing the new element at its appropriate position. Similarly, when the size of the top bucket becomes $4 \log n$, it is split in two new leaf structures. The first leaf structure consists of the first $2 \log n$ bottom buckets, and the second leaf structure contains the rest. The second leaf structure is associated with the original guiding element, and the guiding element of the new leaf structure is the last guiding element in its top bucket. This new guiding element is inserted into the top tree.

Deletions are handled similarly by first searching for the element and then removing it from the appropriate bucket. When an element is deleted from a bucket, we ensure that the elements in the affected bucket are stored consecutively by swapping the deleted element with the last element. If the affected bucket holds fewer than $\delta$ elements after the deletion, it is merged with a neighboring bucket. If the resulting bucket contains more than $6\delta$ elements, it is split as described above. If the top bucket contains less than $\log n$ guiding elements, it is merged with a neighboring leaf structure which is found using a search. Following this, the original leaf is deleted from the top tree.

**Lemma 7.3** *The insert and delete operations of the dynamic dictionary take $O(\log n + \delta)$ amortized time each.*

*Proof.* An update in the top tree takes $O(\delta \log^2 n)$ time and requires $\Omega(\delta \log n)$ updates in the leaf structures. Thus each update costs amortized $O(\log n)$ time for operations in the top tree. Splitting and merging a bucket of a leaf structure takes time $O(\delta \log n)$ for updates to the top bucket and $O(\delta^2)$ time for computing a split element for a bucket. A bucket is split or merged every $\Omega(\delta)$ operations resulting in an amortized update cost of $O(\log n + \delta)$. Appending or removing a single element to a bucket takes worst case time $O(\delta)$ for updating the size. Adding the $O(\log n + \delta)$ cost of the initial search concludes the proof. □

**Theorem 7.3** *The resilient dynamic dictionary structure uses $O(n)$ space while supporting searches in $O(\log n + \delta)$ time worst case with an amortized update cost of $O(\log n + \delta)$. Range queries with an output size of $k$ is performed in worst case $O(\log n + \delta + k)$ time.*

# Chapter 8

## Fault Tolerant External Memory Algorithms

In this chapter we conduct the first study of external memory algorithms and data structures in the presence of an unreliable internal and external memory.

**Our Contribution** The work in this chapter combines the faulty memory RAM and the external memory model in the natural way. The model has three levels of memory: a disk, an internal memory of size $M$, and $O(1)$ CPU registers. All computation takes place on elements placed in the registers. The content of any cell on disk or in internal memory can be corrupted at any time, but at most $\delta$ corruptions can occur. Moving elements between memory and registers takes constant time and transferring a chunk of $B$ consecutive elements between disk and memory costs one *I/O*. Transfers between the different levels are atomic, no data can be corrupted while it is being copied. Correctness of an algorithm is proved with the assumption that an adaptive adversary may perform corruptions during execution. For randomized algorithms we assume that the random bits are hidden from the adversary. In two natural variants of our model it is assumed that corruptions take place only on disk, or only in memory.

We present I/O-efficient solutions to all problems that, to the best of our knowledge, have previously been considered in the faulty memory RAM. It is not clear that resilient algorithms can be optimal both in time and in I/O-complexity. Most techniques for designing I/O-efficient algorithms naturally try to arrange data on disk such that few blocks need to be read in order to extract the information needed, whereas resilient algorithms try to put little emphasis on individual, potentially corrupted, memory cells.

It is known that any resilient comparison based search algorithm must examine $\Omega(\log N + \delta)$ memory cells [40]. Combining this with the well-known $\Omega(\log_B N)$ I/O lower bound on external memory comparison based searching, we get a simple lower bound of $\Omega\left(\log_B N + \frac{\delta}{B}\right)$ I/Os, and $\Omega(\log N + \delta)$ time. In Section 8.1 we prove a stronger lower bound of $\Omega\left(\frac{1}{\varepsilon}\log_B N + \frac{\delta}{B^{1-\varepsilon}}\right)$ I/Os for a search, for all $\log_B N \leq \delta \leq B \log N$ and $\varepsilon$ given by the equation $\delta = \frac{B^{1-\varepsilon}}{\varepsilon}\log_B N$. In the case where $\delta = \Theta(\frac{B}{\log B}\log_{\log B} N)$, setting $\varepsilon = \frac{\log\log B}{\log B}$ gives a lower bound of $\Omega(\log_{\log B} N + \frac{\delta}{B}\log B)$ which is $\omega(\log_B N + \frac{\delta}{B})$. We

| | I/O Complexity | Assumptions | I/O Tolerance (max $\delta$) | Time Tolerance (max $\delta$) |
|---|---|---|---|---|
| Deterministic Dictionary | $O\left(\frac{1}{\varepsilon}\log_B N + \frac{\delta}{B^{1-\varepsilon}}\right)$ | $\frac{1}{\log B} < \varepsilon < 1$ | $O(B^{1-\varepsilon}\log_B N)$ | $O(\log N)$ |
| Randomized Dictionary | $O(\log_B N + \frac{\delta}{B})$ | Memory Safe | $O(B\log_B N)$ | $O(\log N)$ |
| Priority Queue | $O(\frac{1}{1-\varepsilon}\frac{1}{B}\log_{M/B}(\frac{N}{M}))$ | $\delta \leq M^\varepsilon$, $\varepsilon < 1$ | $O(M^\varepsilon)$ | $O(\log N)$ |
| Sorting | $O(\frac{1}{1-\varepsilon}\mathrm{Sort}(N))$ | $\delta \leq M^\varepsilon, \varepsilon < 1$ | $O(M^\varepsilon)$ | $O(\sqrt{N\log N})$ |

Table 8.1: The first column shows the I/O upper bounds presented in our paper with the assumptions shown in the second column. The third and fourth column shows how many corruptions the algorithms can tolerate while still matching the optimal algorithms in the I/O and comparison model respectively. Note that the restriction imposed by the time bounds are orders of magnitude stronger than the ones imposed by the I/O bounds for realistic values of $M$, $B$ and $N$.

come to the interesting conclusion that no deterministic resilient dictionary can obtain an I/O bound of $O(\log_B N + \frac{\delta}{B})$ without some assumptions on $\delta$. The lower bound is valid for randomized algorithms as long as the internal memory is unreliable. For deterministic algorithms, the lower bound also holds if the internal memory is reliable and corruptions only occur on disk.

In Section 8.2 we construct a resilient dictionary supporting searches using expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time for any $\delta$ if corruptions occur exclusively on disk. Thus, we have an interesting separation between the I/O complexity of resilient randomized and resilient deterministic searching algorithms. This also proves that it is important whether it is the disk or the internal memory that is unreliable.

In Section 8.3 we present an optimal resilient static dictionary supporting queries in $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time when $\log N \leq \delta \leq B\log N$ and $\frac{1}{\log B} \leq c \leq 1$. Queries use $O(\log_B N + \frac{\delta}{B})$ I/Os and $O(\log n + \delta)$ time for $\delta \leq \log N$ and $\delta > B\log N$. Additionally, we construct randomized and deterministic dynamic dictionaries with optimal query bounds using our static dictionaries.

Finally, in Section 8.5 we describe a resilient multi-way merging algorithm. We use this algorithm to design an optimal resilient sorting algorithm using $O(\frac{1}{1-\varepsilon}\mathrm{Sort}(N))$ I/Os and $O(N\log N + \alpha\delta)$ time under the assumption that $\delta \leq M^\varepsilon$, for $0 \leq \varepsilon < 1$. The multi-way merging algorithm is also used to design a resilient priority queue for the case $\delta \leq M^\varepsilon$, where $0 \leq \varepsilon < 1$. Our priority queue supports Insert and DeleteMin in optimal $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ I/Os amortized, matching the bounds for non-resilient external memory priority queues. The amortized time bound for both operations is $O(\log N + \delta)$ matching the time bounds of the optimal resilient priority queue of [D8].

Table 8.1 shows an overview of the upper bounds in this paper. The two last columns in the table shows how many corruptions our algorithms can tolerate while still achieving optimal bounds in the I/O model and comparison model

respectively. Note that the bounds on $\delta$ required to get optimal time are orders of magnitude smaller than the bounds required to get optimal I/O performance for realistic values of $N$, $M$ and $B$. We conclude that it is possible, under realistic assumptions, to get resilient algorithms that are optimal in both the I/O-model and the comparison model without restricting $\delta$ more than what was required to obtain optimal time bounds in the faulty memory RAM.

## 8.1 Lower Bound for Dictionaries

Any resilient searching algorithm must examine $\Omega(\log N + \delta)$ memory cells in the comparison model [40]. The $\Omega(\log N)$ term follows from the comparison model lower bound for searching. It is well-known that comparison-based searching in the I/O model requires expected $\Omega(\log_B N)$ I/Os. Since any resilient searching algorithm must read at least $\Omega(\delta)$ elements to ensure at least some non-corrupted information is the basis for the output, we get the following trivial lower bound.

**Lemma 8.1** *For any comparison-based randomized resilient dictionary the average-case expected search cost is* $\Omega\left(\log_B N + \frac{\delta}{B}\right)$ *I/Os.*

In this section we prove a stronger lower bound on the worst-case number of I/Os required for any deterministic resilient static dictionary in the comparison model. We do not make any assumptions on the data structure used by the dictionary, nor on the space it uses. Additionally, we do not bound the amount of computation time used in a query and we assume that the total order of all elements stored in the dictionary are known by the algorithm initially. During the search for an element $e$, an algorithm gains information by performing block I/Os, each I/O reading $B$ elements from disk. Before a block of $B$ elements is read into memory the adversary can corrupt the elements in the block. The adversary is allowed to corrupt up to $\delta$ elements during the query operation, but does not have to reveal when it chooses to do so. Also, the adversary adaptively decides what the rank of the search element has among the $N$ dictionary elements. Of course, the rank must be consistent with the previous uncorrupted elements read by the algorithm.

**Theorem 8.1** *Given $N$ and $\delta$, any deterministic resilient static dictionary requires worst-case* $\Omega\left(\frac{1}{\varepsilon}\log_B N\right)$ *I/Os for a search, for all $\varepsilon$ where* $\frac{1}{\log B} \leq \varepsilon \leq 1$ *and* $\delta \geq \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$.

*Proof.* We design an adversary that uses corruptions to control how much information any correct query algorithm gains from each block transfer.

Let $\varepsilon$ be a constant such that $\frac{1}{\log B} \leq \varepsilon \leq 1$. The strategy of the adversary is as follows. For each I/O, the adversary narrows the *candidate interval* where $e$ can be contained in by a factor $B^\varepsilon$. Initially, the candidate interval consists of all $N$ elements. For each I/O, the adversary implicitly divides the sorted set of elements in the candidate interval into $B^\varepsilon$ slabs of equal size. Since the search algorithm only reads $B$ elements in an I/O, there must be at least one slab containing at most $B^{1-\varepsilon}$ of these elements. The adversary corrupts these

elements, such that they do not reveal any information, and decides that the search element resides in this slab. The remaining elements transferred are not corrupted and are automatically consistent with the interval chosen for $e$. The game is then played recursively on the elements of the selected slab, until all elements in the final candidate interval have been examined.

For each I/O, the candidate interval decreases by a factor $B^\varepsilon$. The algorithm has no information regarding elements in the slab except for the corrupted elements from the I/Os performed so far. After $k$ I/Os the candidate interval has size $\frac{N}{(B^\varepsilon)^k}$ and the adversary has introduced at most $kB^{1-\varepsilon}$ corruptions. The game continues as long as there is at least one uncorrupted element among the elements remaining in the candidate interval, which the adversary can choose as the search element. All corrupted elements may reside in the current candidate interval, and the game ends when the size of the candidate interval, $\frac{N}{(B^\varepsilon)^k}$, becomes smaller than or equal to the total number of introduced corruptions, $kB^{1-\varepsilon}$. It follows that at least $\Omega\left(\log_{B^\varepsilon} \frac{N}{B^{1-\varepsilon}}\right) = \Omega\left(\frac{1}{\varepsilon}\log_B N\right)$ I/Os are required. The adversary introduces at most $B^{1-\varepsilon}$ corruptions in each step. If $\varepsilon$ satisfies $\frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N \leq \delta$, then the adversary can play the game for at least $\frac{1}{\varepsilon}\log_B N$ rounds and the theorem follows. □

For deterministic algorithms it does not matter whether elements can be corrupted on disk or in internal memory. Since the adversary is adaptive it knows which block of elements an algorithm will read into internal memory next, and may choose to corrupt the elements on disk just before they are loaded into memory, or corrupt the elements in internal memory just after they have been written there. In randomized algorithms where the adversary does not know the algorithm's random choices it cannot determine which block of elements will be fetched from disk before the transfer has started. Therefore, the adversary can follow the strategy above only if it can corrupt elements in internal memory.

By setting $\delta = \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$ in Theorem 8.1, we get the following corollary.

**Corollary 8.1** *Any deterministic resilient static dictionary requires worst-case* $\Omega(\frac{1}{\varepsilon}\log_B N) = \Omega(\frac{\delta}{B^{1-\varepsilon}})$ *I/Os for a search, where* $\delta \in [\log_B N, B\log N]$*, and* $\varepsilon$ *given by* $\delta = \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$*.*

The trivial I/O lower bound for a resilient searching algorithm is $\Omega\left(\log_B N + \frac{\delta}{B}\right)$. Setting $\varepsilon = \frac{\log\log B}{\log B}$ in Theorem 8.1 shows that this is not optimal.

**Corollary 8.2** *For* $\delta = \frac{B}{\log B}\log_{\log B} N$ *any deterministic resilient static dictionary requires worst-case* $\Omega(\frac{\log B}{\log\log B}(\log_B N + \frac{\delta}{B}))$ *I/Os for a search.*

## 8.2 Randomized Static Dictionary

In this section we describe a simple I/O-efficient randomized static dictionary, that is resilient to corruptions on the disk. Corruptions in memory are not allowed, thus the adversarial lower bound in Theorem 8.1 does not apply. The

dictionary supports queries using expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time. The algorithm is similar to the randomized binary search algorithm in [40]. Remember that, if only elements on disk can be corrupted, the lower bound from Theorem 8.1 also holds for deterministic algorithms. This means that deterministic and randomized algorithms are separated by the result in this section.

The idea is to store the $N$ elements in the dictionary in sorted order in an array $S$ and to build $2\delta$ B-trees [15], denoted $T_1, \ldots, T_{2\delta}$, of size $\lfloor \frac{N}{2\delta} \rfloor$. The $i$'th B-tree $T_i$ stores the $2\delta j + i$'th element in $S$ for $j = 0, \ldots, \lfloor \frac{N}{2\delta} \rfloor - 1$. Each node in each tree is represented by a faithfully ordered array of $\Theta(B)$ search keys. The nodes of the B-tree are laid out in left to right breadth first order, to avoid storing pointers, i.e. the $c$'th child of the node at index $k$ has index $Bk + c - (B - 1)$.

The search for an element $e$ proceeds as follows. A random number $r_1 \in \{1, \ldots, 2\delta\}$ is generated, and the root block of $T_{r_1}$ is fetched into the internal memory. In this block, a binary search is performed among the search keys resulting in an index, $i$, of the child where the search should continue. A new random number $r_2 \in \{1, \ldots, 2\delta\}$ is generated, and the $i$'th child of the root in tree $T_{r_2}$ is fetched and the algorithm proceeds iteratively as above. The search terminates when a leaf is reached and two keys $S[2\delta j+i]$ and $S[2\delta(j+1)+i]$ have been identified such that $S[2\delta j + i] \leq e < S[2\delta(j + 1) + i]$. If the binary search was not mislead by corruptions of elements, then $e$ is located in the subarray $S[2\delta j + i, \ldots, 2\delta(j + 1) + i]$. To check whether the search was mislead, the following *verification procedure* is performed. Consider the neighborhoods $L = S[2\delta(j-1)+i-1, \ldots, 2\delta j+i-1]$ and $R = S[2\delta(j+1)+i+1, \ldots, 2\delta(j+2)+i+1]$, containing the $2\delta + 1$ elements in $S$ situated to the left of $S[2\delta j + i]$ and to the right of $S[2\delta(j + 1) + i]$ respectively. The number $s_L = |\{z \in L \mid z \leq e\}|$ of elements in $L$ that are smaller than $e$ is computed by scanning $L$. Similarly, the number $s_R$ of elements in $R$ that are larger than $e$ is computed. If $s_L \geq \delta + 1$ and $s_R \geq \delta + 1$, and the search key is not encountered in $L$ or $R$, we decide whether it is contained in the dictionary or not by scanning the subarray $S[2\delta j, \ldots, 2\delta(j + 1)]$. If $s_L$ or $s_R$ is smaller than $\delta + 1$, at least one corruption has misguided the search. In this case, the search algorithm is restarted.

**Theorem 8.2** *The data structure described is a linear space randomized dictionary supporting searches in expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time assuming that memory cells are incorruptible and block transfers are atomic.*

*Proof.* The proof roughly follows the proof of [36]. First, we prove correctness of the algorithm. Assume that $s_L \geq \delta + 1$ and $e \notin L$. Since only $\delta$ corruptions are possible, there exists at least one uncorrupted element in $L$ smaller than $e$. Because $S$ is sorted, no uncorrupted elements to the left of $S[2\delta j]$ in $S$ can be equal to $e$. By a similar argument, if $s_R \geq \delta+1$ and $e \notin R$, then no uncorrupted elements to the right of $S[2\delta(j + 1)]$ in $S$ are equal to $e$. If no corruptions are encountered during the B-tree search, all the uncorrupted elements of $L$ are less than or equal to $e$, and therefore $s_L \geq \delta + 1$. Similarly, we have $s_R \geq \delta + 1$, and the algorithm terminates after scanning $S[2\delta j, \ldots, 2\delta(j + 1)]$.

In each step, the algorithm chooses a random B-tree among the $2\delta$ B-trees, and loads the next node from the randomly chosen B-tree to guide the search. The adversary cannot know which B-tree that is used in any iteration, since he does not know the random bits and block reads are atomic. Let $\beta_i$ be the number of nodes containing corruptions on level $i$ in each of the B-trees. Then, the probability that the node used in iteration $i$ contains corruptions is at most $\frac{\beta_i}{2\delta}$ and thus the probability that the algorithm does not use any blocks containing corrupted elements is $\prod_{i=1}^{\log_B N}\left(1-\frac{\beta_i}{2\delta}\right)$ which is at least $\frac{1}{2}$ [C1]. It follows that the expected number of restarts caused by misguided searches due to faults is at most 2. □

If memory cells were corruptible the atomic transfer assumption would be of little use. The adversary could simply corrupt the elements in the internal memory after the block transfer completes, decreasing the benefit of the randomization.

## 8.3 Optimal Deterministic Static Dictionary

In this section we present a linear space deterministic resilient static dictionary. Let $c$ be a constant such that $\frac{1}{\log B} \leq c \leq 1$. The dictionary supports queries in $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time. In Section 8.1 we proved a lower bound on the I/O complexity of resilient dictionaries, and by choosing $c$ in the above bound to minimize the expression for $\alpha = \delta$, this bound matches the lower bound. Thus, this dictionary is optimal.

Our data structure is based on the B-tree and the resilient binary search algorithm from Chapter 7. In a standard B-tree search one corrupted element can misguide the algorithm, forcing at least one I/O in the *wrong* part of the tree. To circumvent this problem, each guiding element in each internal node is determined by taking majority of $B^{1-c}$ copies. This gives a trade-off between the number of corruptions required to misguide a search, and the fan-out of the tree, which becomes $B^c$. Additionally, each node stores $2\delta + 1$ copies of the minimum and maximum element contained in the subtree, such that the search algorithm can reliably check whether it is on the correct path in each step. We ensure that the query algorithm avoids reading the same corrupted element twice by ensuring that any element is read at most once. The exact layout of the tree and the details of the search operation are as follows.

*Structure:* Let $S$ be the set of elements contained in the dictionary and let $N$ denote the size of $S$. The dictionary is a $B^c$-ary search tree $T$ built on $\frac{N}{\delta}$ leaves. The elements of $S$ are distributed to the leaves in faithful order such that each leaf contains $\delta$ elements. Each leaf is represented by a *guiding element* which is smaller than the smallest uncorrupted element in the leaf and larger than the largest uncorrupted element in the preceding leaf. The top tree is built using these guiding elements. The tree is stored in a breadth-first left-to-right layout on disk, such that no pointers are required.

Each internal node $u$ in $T$ stores three types of elements; guiding elements, minimum elements, and maximum elements, stored consecutively on disk. The

guiding elements are stored in $\lceil (2\delta+1)/B^{1-c} \rceil$ identical blocks. Each block contains $B^{1-c}$ copies of each of the $B^c$ guiding elements in sorted order such that the first $B^{1-c}$ elements are copies of the smallest guiding element. This means that each guiding element is stored $2\delta+1$ times and can be retrieved reliably. The minimum elements are $2\delta+1$ copies of the guiding element for the leftmost leaf in the subtree defined by $u$, stored consecutively in $\lceil \frac{2\delta+1}{B} \rceil$ blocks. Similarly the maximum elements are $2\delta+1$ copies of the guiding element for the leaf following the rightmost leaf in the subtree defined by $u$, stored consecutively in $\lceil \frac{2\delta+1}{B} \rceil$ blocks. Additionally, minimum and maximum elements are stored with each leaf. The minimum are $4\delta$ copies of the guiding element representing the leaf, stored consecutively in $\frac{4\delta}{B}$ blocks, and the maximum elements are $4\delta$ copies of the guiding element representing the subsequent leaf, stored consecutively in $\frac{4\delta}{B}$ blocks. These are used to verify that the algorithm found the only leaf that may store an uncorrupted element matching the search element.

*Query:* A query operation for an element $q$, uses an index $k$ that indicates how many chunks of $B^{1-c}$ elements the algorithm has discarded during the search, initially $k=0$. Intuitively, a chunk is discarded if the algorithm detects that $\Omega(B^{1-c})$ of its elements are corrupted. The query operation traverses the tree top-down, storing in safe memory the index $k$, and $O(1)$ extra variables required to traverse the tree using the knowledge of its layout on disk. In an internal node $u$, the algorithm starts by checking whether $u$ is on the correct path in the tree using the copies of the minimum and maximum elements stored in $u$. This is done by scanning $B^{1-c}$ of the $2\delta+1$ copies of the minimum element starting with the $kB^{1-c}$'th copy, counting how many of these that are larger than $q$. If $B^{1-c}/2$ or more copies of the minimum element are larger than $q$ the block is discarded by incrementing $k$ and the search is restarted (backtracked) at node $v$, where $v=u$ if $u$ is root of the tree and the parent of $u$ otherwise. The maximum elements are checked similarly. If the algorithm backtracks, $k$ is increased ensuring that the same element is never read more than once.

If the checks succeed the $k$'th block storing copies of the $B^c$ guiding elements of $u$ is scanned from left to right. The majority value of each of the $B^{1-c}$ copies of each guiding element is extracted in sorted order using the majority algorithm [23] and compared to $q$, until a retrieved guiding element larger than $q$ is found or the entire block is read. The traversal then continues to the corresponding child. If any invocation of the majority algorithm fails to select a value, or two fetched guiding elements are out of order, the block is discarded as above by increasing $k$ and backtracking the search to the parent node.

Upon reaching a leaf, the algorithm verifies whether the search found the correct leaf. This is achieved by running a variant of the verification procedure designed for the same purpose in Chapter 7. Counters $c_l$ and $c_r$, which are initially 1, are stored in safe memory. Then the copies of the minimum and maximum element are scanned in chunks of $B^{1-c}$ elements, starting from the $2kB^{1-c}$'th element. If the majority of elements in a chunk of $B^{1-c}$ copies of the minimum element are smaller than the search element, $c_l$ is increased by 1. Otherwise, $c_l$ is decreased and $k$ increased by one. The copies of maximum elements are treated similarly. Note that every decrement of $c_l$ or $c_r$ signals that at least $\frac{B^{1-c}}{2}$ corruptions have been found. Thus, $c_l$ represents the number

of chunks scanned that has not yet been contradicted, where the majority of copies indicates that the search element is in the current leaf or in leafs to the right. Similar for $c_r$. If $\min\{c_l, c_r\}$ reaches 0, we backtrack to the parent of the leaf as above. If $\min\{c_l, c_r\}\frac{B^{1-c}}{2}$ gets larger than $\delta - k(\frac{B^{1-c}}{2}) + 1$ the verification succeeds. The algorithm finishes by scanning the $\delta$ elements stored in the leaf, returning whether it finds $q$ or not.

**Lemma 8.2** *The data structure is a linear space resilient dictionary supporting queries in* $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ *I/Os, for any* $1/\log B \leq c \leq 1$.

*Proof.* A value is only erroneously retrieved when at least $\frac{B^{1-c}}{2}$ of the copies used to determine it are corrupted. If $k$ is incremented because of a failed check in an internal node $u$, then at least one value in the parent of $u$ was erroneously retrieved or at least $\frac{B^{1-c}}{2}$ copies of the minimum or maximum value read at $u$ were corrupted. If $k$ is increased during a verification of a leaf, the majority of elements in one chunk of $B^{1-c}$ copies of the minimum (maximum) element was larger (smaller) than the search element and in another the majority was smaller (larger) than the search element. Therefore, $k$ is only increased when $\frac{B^{1-c}}{2}$ corruptions are detected. Since $k$ increases before any backtracking is performed, the algorithm never reads the same element twice, proving that all corruptions counted are distinct.

The algorithm finishes when $\min\{c_l, c_r\}\frac{B^{1-c}}{2} \geq \delta - k\frac{B}{2} + 1$ during the verification of a leaf. Since the adversary has at most $\delta - k\frac{B}{2}$ corruptions left, in at least one chunk of $B^{1-c}$ copies of the minimum element read during the verification, more than half of the elements are uncorrupted. Since the majority of copies in this block are smaller than the search element, no uncorrupted elements matching the search key can be in leafs to the left. Similar for the blocks containing copies of the maximum, proving that the correct leaf is found.

To bound the I/O complexity, we count how many nodes of the tree the algorithm visits, that are **not** on the correct root to leaf path. If a search is guided in the wrong direction (away from the correct root to leaf path), the majority of $B^{1-c}$ copies of a guiding element in the relevant block are corrupted. For each additional step performed by the algorithm after a *wrong* turn, either the minimum or the maximum chunk scanned must contain $\frac{B^{1-c}}{2}$ corruptions.

In the verification step, each time a minimum and a maximum block is scanned either $k$ or $\min\{c_l, c_r\}$ is increased. Therefore, if $2t-1$ I/Os were performed by a failed verification $k$ increased by $t$, meaning that $\frac{tB^{1-c}}{2}$ corruptions were detected. We conclude that the algorithm uses $O(\log_{B^c} N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}) = O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ I/Os. □

To obtain optimal time bounds for the dictionary, we use the resilient binary search algorithm from Chapter 7 on each block, instead of scanning it. If more than $\frac{B^{1-c}}{2}$ corruptions are discovered during the search of a block, it is discarded as above. Otherwise, $\frac{B^{1-c}}{2}$ supporting elements are found on both sides of an element, and the algorithm continues to the corresponding child as before. This reduces the time used per node to $O(\log B + B^{1-c})$. Verification takes $O(\delta)$ time in total.

**Lemma 8.3** *For any* $\frac{1}{\log B} \leq c \leq 1$*, queries use* $O((B^{1-c} + \log B)(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}}) + \delta)$ *time.*

**Corollary 8.3** *If* $\delta > B \log N$*, queries use* $O(\frac{\delta}{B})$ *I/Os and* $O(\delta)$ *time.*

*Proof.* Follows from Lemma 8.2 and 8.3 by setting $c = \frac{1}{\log B}$ in Lemma 8.2 and 8.3, i.e. $T$ is a binary tree. □

**Corollary 8.4** *If* $\delta < \log N$*, queries use* $O(\log_B N)$ *I/Os and* $O(\log N)$ *time.*

*Proof.* Follows from Lemma 8.2 and 8.3 by setting $c = 1 - \frac{\log\log B}{\log B}$ i.e. $T$ has degree $\frac{B}{\log B}$. □

**Corollary 8.5** *If* $\log N \leq \delta \leq B \log N$ *for any* $\frac{1}{\log B} \leq c \leq 1$*, queries use* $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ *I/Os and* $O(\log N + \delta)$ *time.*

*Proof.* Follows from Lemma 8.2 and 8.3 by selecting $c \in [\frac{1}{\log B}, 1 - \frac{\log\log B}{\log B}]$ such that $\frac{1}{c}\log_B N = \frac{\delta}{B^{1-c}}$. □

## 8.4 Dynamic Dictionaries

In this section we present a dynamic I/O-efficient resilient dictionary based on the techniques in used for the dynamic dictionary in Chapter 7 and the static dictionary presented in Section 8.3. The dynamic dictionary supports queries and updates in $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ I/Os and $O(\log N + \delta)$ time, worst-case and amortized respectively for any fixed constant $c$ in the range $\frac{1}{\log B} \leq c \leq 1$.

### 8.4.1 Structure

The data structure consists of a search tree, $T$, constructed on a set of $\Theta(N/(\delta \log^3 N))$ leaf structures, each containing $\Theta(\delta \log^3 N)$ elements.

The top-level search tree $T$ is based on the binary trees of Brodal et al. [24], with all elements and pointers replicated $2\delta + 1$ times. We maintain $T$ such that all levels of the tree, except possibly the last $O(1)$ levels, are complete [24]. Additionally, we maintain an auxiliary static dictionary $D_T$, described in Section 8.3, containing the replicated guiding elements stored in the lowest level of $T$ that is guaranteed to be complete.

A leaf structure contains $\Theta(\delta \log^3 N)$ elements distributed, in sorted order, into $\Theta(\log^3 N)$ buffers of size $\Theta(\delta)$. The buffers are the leaves of a three level search tree $L_T$ with fan-out $\Theta(\log N)$. Similar to $T$ the components of $L_T$ are stored reliably. Finally, the $2\delta + 1$ copies of each of the $\Theta(\log N)$ elements in each internal node of $L_T$ are stored in a static dictionary (Section 8.3). Note that the elements in the buffers stored in the leaves of $L_T$, however, are *not* replicated.

### 8.4.2 Operations

A query operation for an element $q$ initially queries the dictionary $D_T$. The constant sized subtrees of $T$ rooted at the nodes returned by this query are traversed by reliably determining the replicated pointers and guiding elements. This traversal identifies a leaf structure. The corresponding tree $L_T$ is traversed starting from the root, using the static dictionaries stored in the internal nodes of $L_T$ to guide the search, and retrieving the replicated pointers to traverse the tree reliably. Upon reaching a leaf in $L_T$, its associated buffer is scanned and the query returns true if $q$ is found, otherwise it returns false.

To delete an element $e$ from the dictionary it is first removed from the buffer that contains it. If the buffer becomes too small it is merged with a neighboring buffer, and the representative of the merged buffer becomes the largest of the two guiding elements associated with the respective buffers before the merge. The smaller guiding element is then removed from all ancestors of the merged leafs in $L_T$ by completely rebuilding them. If the leaf structure $L$ now contains too few elements it is merged with a neighboring leaf structure in $T$ and rebuild. In this case the top tree $T$ is also updated using the algorithm of [24] and the reliably stored elements and structural information. This update may alter the set of elements stored in the lowest complete level of $T$. Each new element in the lowest complete level overwrites the element it replaced in the static dictionary $D_T$. An insert is handled similarly.

**Theorem 8.3** *The data structure described is a deterministic dynamic resilient dictionary supporting searches and updates in* $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ *I/Os and* $O(\log N + \delta)$ *time, worst-case and amortized respectively.*

*Proof.* The top tree $T$ uses $O(|T|\delta/B) = O(N/B)$ blocks and the size of each leaf structure is dominated by the elements stored in the leaf buffers.

In a query, one static dictionary storing $O(\frac{N}{\log^3 N})$ elements and three static dictionaries storing $O(\delta \log N)$ elements are searched, $O(1)$ replicated values are retrieved reliably, and a buffer storing $O(\delta)$ elements is scanned.

Completely rebuilding a static dictionary from a sorted set of $N$ replicated elements takes $O(\delta N/B)$ I/Os and $O(\delta N)$ time. Therefore, rebuilding three nodes in $L_T$ during an update takes $O(\delta \log N)$ time and $O(\frac{\delta}{B}\log N)$ I/Os. Since the size of the buffers in the leaves can vary by a constant factor, this is only needed every $\Theta(\delta)$ updates, meaning that the amortized cost of updating $L_T$ is $O(\frac{\log N}{B})$ I/Os and $O(\log N)$ time. Rebuilding an entire leaf structure from a sorted set of buffers takes $O(\frac{\delta}{B}\log^3 N)$ I/Os and $O(\delta \log^3 N)$ time and this only happens every $\Theta(\delta \log^3 N)$ updates. Thus, the amortized cost of updating a leaf structure is $O(\frac{1}{B})$ I/Os and $O(1)$ time.

The update time for the top tree $T$ as presented in [24] is $O(\log^2 N)$. Since all components of $T$ are stored reliably, updating $T$ takes $O(\delta \log^2 N)$ time and $O(\lceil \frac{\delta}{B} \rceil \log^2 N)$ I/Os. When $T$ is updated, all elements that have been replaced at the lowest guaranteed complete level of $T$ are overwritten by the value replacing them in this level in the auxiliary static dictionary $D_T$. In the static dictionary from Section 8.3, an element can exist replicated $O(1)$ times on each level of the tree. Therefore, it takes $O(\delta \frac{1}{c}\log_B N)$ time and $O(\lceil \frac{\delta}{B^{1-c}} \rceil \frac{1}{c}\log_B N)$

I/Os to update such a value, and amortized $O(\log^2 N)$ elements are changed in each update of $T$. Again, since the size of the leaf structures can vary within a constant fraction, updates to $T$ are only needed every $\Theta(\delta \log^3 N)$ update, thus the amortized update cost for $T$ is $O(1)$ I/Os and $O(1)$ time. □

## 8.5 Sorting

In this section we present a resilient multi-way merging algorithm and use it to design a resilient I/O-efficient sorting algorithm. It is also used in the next section to design a resilient I/O-efficient priority queue. First we show how to merge $\gamma$ faithfully ordered lists of total size $x$ when $\gamma \leq \min\{\frac{M}{B}, \frac{M}{\delta}\}$.

### 8.5.1 Multi-way Merging

Initially, the algorithm constructs a perfectly balanced binary tree, $T$, in memory on top of the $\gamma$ buffers being merged. Each edge of the binary tree is equipped with a buffer of size $5\delta + 1$. Each internal node $u \in T$ stores the state of a running instance of the *PurifyingMerge* resilient binary merging algorithm from [36]. In each round $O(\delta)$ elements from both input buffers are read and the next $\delta$ elements in the faithful order are output. If corrupted elements are found, these are moved to a fail buffer and the round is restarted. The algorithm merges elements from the buffers on $u$'s left child edge and right child edge into the buffer of $u$'s parent edge. The states and sizes of all buffers are stored as reliable variables. The entire tree including all buffers and state variables are stored in internal memory, along with one block from each of the $\gamma$ input streams and one block for the output stream of the root. Instead of storing a fail buffer for each instance of *PurifyingMerge*, a global shared fail buffer $F$ is stored containing all detected corrupted elements.

Let $b_l$ and $b_r$ be the buffers on the edges to the left and right child respectively and let $b$ denote the buffer on the edge from $u$ to its parent. If $u$ is the root, $b$ is the output buffer. The elements are merged using the *fill* operation, which operates on $u$, as follows. First, it checks whether $b_l$ and $b_r$ contain at least $4\delta + 1$ elements, and if not they are filled recursively. Then, the stored instance of the *PurifyingMerge* algorithm is resumed by running a round of the algorithm outputting the next $\delta$ elements to its output stream. The multi-way merging algorithm is initiated by invoking *fill* on the root of $T$ which runs until all elements have been output. Then, the elements moved to $F$ during the *fill* are merged into the output using *NaiveSort* and *UnbalancedMerge* as in [40].

**Lemma 8.4** *Merging $\gamma = \min\{\frac{M}{B}, \frac{M}{\delta}\}$ buffers of total size $x \geq M$ uses $O(x/B)$ I/Os and $O(x \log \gamma + \alpha\delta)$ time.*

*Proof.* The correctness follows from Lemma 1 in [40]. The size of $T$ is $O(\gamma(\delta + B)) = O(\min\{\frac{M}{B}, \frac{M}{\delta}\}(\delta + B)) = O(M)$. We use $\gamma$ I/Os to load the first block in each leaf of $T$ and $O(x/B)$ I/Os for reading the entire input and writing the output. The final merge with $F$ takes $O(x/B)$ I/Os. Since $T$ fits completely in memory we perform no other I/Os.

Merging two buffers of total size $n$ using *PurifyingMerge* takes $O(n + \alpha\delta)$ time where $\alpha$ is the number of detected corruptions in the input buffers. Since detected corruptions are moved to the global fail buffer each corruption is only charged once. Each element passes through $\log \gamma$ nodes of $T$ and the final merge using *NaiveSort* and *UnbalancedMerge* takes $O(x + \alpha\delta)$ time. □

### 8.5.2 Sorting

Assuming $\delta \leq M^\varepsilon$ for $0 \leq \varepsilon < 1$, we can use the multi-way merging algorithm to implement the standard external memory $M^{1-\varepsilon}$-way merge sort from [2] matching the optimal external memory sorting bound for constant $\varepsilon$.

**Theorem 8.4** *Our resilient sorting algorithm uses $O(\frac{1}{1-\varepsilon}Sort(N))$ I/Os and $O(N \log N + \alpha\delta)$ time assuming $\delta \leq M^\varepsilon$.*

## 8.6 Priority Queue

In this section we describe a comparison-based resilient priority queue which is optimal with respect to both time and I/O performance assuming that $\delta \leq M^\varepsilon$. An optimal I/O-efficient priority queue uses $\Theta(1/B \log_{M/B}(N/M))$ I/Os amortized per operation [2]. An $\Omega(\log N + \delta)$ time lower bound for comparison-based resilient priority queues was proved in Chapter 6.

Our priority queue is based on an amortized version of the worst-case optimal external memory priority queue of [26] and uses our new multi-way merging algorithm to move elements between disk and internal memory.

The priority queue consists of a part on disk, denoted $\mathcal{L}$, and three structures, $D, I$ and $F$, in internal memory. We maintain that $D$ stores the smallest elements in the priority queue and that $I$ stores newly inserted elements. We maintain that $D$ has more than $\delta + 1$ elements and that both $I$ and $D$ have at most $M + \delta + 1$ elements, the former ensures that there is at least one uncorrupted element in $D$. Finally, the buffer $F$, of size $2\delta$, stores possibly corrupted elements.

The structure on disk, $\mathcal{L}$ is a linked list of levels. Each level consists of a number of faithfully ordered sequences, represented by linked lists of blocks of size $\max\{B, \delta\}$, stored in a linked list. Let $N_\ell$ denote the number of buffers at level $\ell$, and let $\mathcal{L}_\ell^i$, $0 \leq i \leq N_\ell$ denote the $i$'th buffer of level $\ell$. The elements at level $\ell$ is $\mathcal{L}_\ell = \mathcal{L}_\ell^0 \cup \cdots \cup \mathcal{L}_\ell^{N_i}$. We define the parameter $\gamma = \min(M/B, M/\delta)$ and maintain that $N_\ell < \gamma$ and that $|\mathcal{L}_\ell| < \gamma^{\ell+1}M$. Let $h$ denote the index of the highest level, the first level is level 0. The pointers in the linked lists as well as buffer offsets and sizes are stored as reliable variables. We describe two operations, PULL and PUSH on $\mathcal{L}$ that extracts and adds $M$ elements to $\mathcal{L}$ respectively.

We will need the following result on selection in the faulty memory RAM: We say that element $x$ in a buffer $X$ has *faithful-rank* $r$ if there is at least $r - \alpha - \delta$ uncorrupted elements in $X$ smaller than $x$ and at most $|X| - r + \alpha + \delta$ uncorrupted elements of $X$ larger than $x$.

**Lemma 8.5** *Given an integer $k$ and a faithfully ordered sequence $S$, $|S| < M$, an element with a faithful-rank $k$ in $S$ can be selected in $O((\alpha+1)\delta)$ time. The element returned has index $i \in \{k, \ldots, k+\alpha\}$ in $S$.*

*Proof.* The algorithm maintains an index $k'$, initialized to $k$ in safe memory. Initially $S[k']$ i copied into a safe memory variable. The algorithm then checks whether the majority of the $2\delta+1$ elements immediately to the left of $S[k']$ are smaller than $S[k']$, and whether the majority of the $2\delta+1$ immediately to the right are larger than $S[k']$. If this is the case $S[k']$ is returned. Otherwise, the algorithm increments the value of $k'$ by one and restarts. The complexity of the algorithm depends on the number of iterations. An iteration fails to select an element if the element at $S[k']$ is corrupted to a value larger than $\delta+1$ of the $2\delta+1$ preceding elements or a value smaller than the $\delta+1$ of the $2\delta+1$ succeeding elements. If $S[k']$ is uncorrupted the algorithm always terminates and returns $S[k']$ and thus at most $\alpha$ iterations are performed. These arguments also prove correctness of the algorithm. □

### 8.6.1 Operations on $\mathcal{L}$

This section describes the basic operations used to manipulate $\mathcal{L}$. Let *merger*$(\ell, x)$ denote an invocation of the multi-way merging algorithm that extracts $x$ elements from level $\ell$. Let $T$ be the binary tree with $N_\ell$ leaves used in the merging algorithm.

If $x < |\mathcal{L}_\ell|$ we do not output all the elements of $\mathcal{L}_\ell$. In that case we run the merger without fetching new elements from the leaves of $T$ once $x$ elements have been output. This empties all the buffers of $T$. The extra elements are gathered into a buffer $E$ of size at most $6M$. We prepend $E$ to a buffer of level $\ell$ making sure this buffer remains faithfully ordered. If there is a buffer, $\mathcal{L}_\ell^j$ with $|\mathcal{L}_\ell^j| < 6\delta$ we simply append this buffer to $E$ and sort the result using the resilient sorting algorithm. If no such buffer exist we use the selection algorithm from Lemma 8.5 with $k = 2\delta+2$ for all $N_\ell$ buffers of level $\ell$ in turn - remembering the maximum element returned so far in the reliable memory. The algorithm from Lemma 8.5 works by repeatedly trying different elements and checking whether or not it is consistently ordered with the $2\delta+1$ elements to its left and the $2\delta+1$ elements to its right. If not it tries the neighboring elements and continues in this fashion. We modify the algorithm such that a candidate element that doesn't work are moved to our global fail buffer $F$ and the front of the list is compacted. Since each round and the compaction take $O(\delta)$ time, the entire operation takes $O(\alpha'\delta)$ time where $\alpha'$ is the number of corrupted elements we sample. By moving corrupted elements to $F$, we ensure that we never spend $\delta$ time in the selection algorithm for the same corruption more than once.

If the maximum element is from buffer $\mathcal{L}_\ell^j$ we take the first $6\delta$ elements of $\mathcal{L}_\ell^j$ and merge them with $E$, using the complete binary merging algorithm [36] and prepend the resulting buffer to $\mathcal{L}_\ell^j$. Finally, the extracted elements are returned.

**Lemma 8.6** *An invocation of* merger$(\ell,x)$ *for $x \geq M$ returns the $x$ faithfully smallest elements from $\mathcal{L}_\ell$ using $O(x/B)$ I/Os and $O(x \log N_\ell + \alpha\delta)$ time and leaves all buffers of $\mathcal{L}_\ell$ faithfully ordered.*

*Proof.* The I/O-complexity follows from Lemma 8.4. Extracting the $x$ elements from the merger and emptying the remaining elements into $E$ uses $O(x \log N_\ell + \alpha\delta)$ time by Lemma 8.4. The $O(N_\ell)$ invocations of the selection algorithm and performing compactions use $O(N_\ell\delta + \alpha'\delta) = O(M + \alpha'\delta) = O(x + \alpha'\delta)$ time where $\alpha'$ is the number of corrupted elements chosen as selection candidates.

We empty the merger into a buffer $E$ and locate a buffer $m$ in which $E$ added to. Finding this list involves using selection $N_\ell$ times and merging the first $6\delta$ elements of the buffer with the elements of $E$, for a total cost of $O(N_\ell\delta + \delta\alpha + M) = O(M + \delta\alpha) = O(x + \delta\alpha)$.

We must also prove that the faithful ordering of the buffer $\mathcal{L}_\ell^j$ in which we insert the elements of $E$ is maintained. Recall that $\mathcal{L}_\ell^j$ was the buffer in which the selection algorithm from Lemma 8.5 returned the largest element $r$ when invoked with $k = 2\delta + 2$. Let $y$ be the maximum uncorrupted element of $E$, we now prove that $y \leq r$.

Let $\mathcal{L}_\ell^p$ be the buffer from which $y$ originated and let $c$ be the element returned by the selection algorithm for $\mathcal{L}_\ell^p$. Since $c$ has faithful-rank $2\delta+2$ there must be at least one uncorrupted element in $\mathcal{L}_\ell^p$ smaller than $c$, this element is also smaller than $y$ and thus $y \leq c$. Since $r$ was the maximum among all the selections we know that $r \geq c$ and thus we conclude that $y \leq c \leq r$. □

### Pushing elements to $\mathcal{L}$

The Push operation inserts a buffer, $A$, of size $M$ into $\mathcal{L}$. It does so by simply adding it as a new list of level $i = 0$ and incrementing $N_i$ by one. If $N_i = \gamma$ all elements of level $i$ are merged into a new list, $A' = \mathit{merger}(i, |\mathcal{L}_i|)$ which is then recursively inserted into level $i + 1$, level $i$ is now empty.

**Lemma 8.7** *The* Push *operation adds $M$ elements to $\mathcal{L}$ and maintains that $N_\ell < \gamma$ and $|\mathcal{L}_\ell| < \gamma^{\ell+1}M$ for $0 \leq \ell \leq h$ and that all buffers in $\mathcal{L}$ remain faithfully ordered. For each visited level, $\ell$, it uses $O(|\mathcal{L}_\ell| \log \gamma + \alpha\delta + N_\ell\delta)$ time and $O(|\mathcal{L}_\ell|/B)$ I/Os while pushing $|\mathcal{L}_\ell|$ elements to the next level.*

*Proof.* By construction, the number of buffers in each level is maintained by Push. We now prove that $|\mathcal{L}_\ell| < \gamma^{\ell+1}$ for all levels $\ell$ after the operation.

This claim is proved by induction. Initially, all lists on level 0 has size $M$, since they are inserted when the $I$ buffer is full and we push $M$ elements into $\mathcal{L}$. Now assume that the claim holds for level $i$. Push is invoked on level $i$, only when $N_i = \gamma$. Since all buffers have size $\gamma^i$ and there are $\gamma$ of them, the size of $\mathit{merger}(i, |\mathcal{L}_i|)$ is at most $|\mathcal{L}_i| = \gamma^{i+1}$ and thus, the list inserted on level $i + 1$ is not to large. Elements moved to the fail buffer only makes levels smaller. The correctness of the multi-way merger proves that the buffers of $\mathcal{L}$ remain faithfully ordered. □

**Pulling elements from $\mathcal{L}$**

The PULL operation iterates from level $i = 0$ to level $h$ and maintains a faithfully ordered buffer $S$ of size $M+\delta$ which contains elements that are faithfully smaller than elements in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$. Furthermore, an index $\ell \leq i$ is maintained with the property that $|\mathcal{L}_j|$ is unchanged for $j \leq i, j \neq \ell$, and $\mathcal{L}_\ell$ contains $M+\delta$ fewer elements than before the PULL.

Initially $S = \mathit{merger}(0, M+\delta)$, $i = 1$ and $\ell = 0$. Each iteration proceeds as follows. First the $M+\delta$ faithfully smallest elements from level $i$ are extracted by invoking $A = \mathit{merger}(i, M+\delta)$. The complete binary merging algorithm from [36] is then invoked on $A$ and $S$ to produce a buffer $C$ of size $2(M+\delta)$. We now split $C$ into two halves $C_1 = C[1, \ldots, M+\delta]$ and $C_2 = C[M+\delta+1, \ldots, 2(M+\delta)]$. We set $S = C_1$. We place $C_2$ in level $\ell$ or level $i$. To do this we perform the same operation we used to put $E$ back into $\mathcal{L}$ in the multi-way merging algorithm. The only difference is that we look at all buffers on level $i$ and on level $\ell$. If $C_2$ is placed on level $\ell$ we set $\ell = i$ for the next iteration. When all levels have been visited, the first $M$ elements of $A$ are merged with $D$. The remaining elements are inserted into $I$.

**Lemma 8.8** *The* PULL *returns $M+\delta$ elements in faithful order and the $M$ first are the faithfully smallest elements of $\mathcal{L}$. All buffers in $\mathcal{L}$ remain sorted. Ignoring the $O(\alpha)$ elements moved to the fail buffer, the size of all but one level remains the same, and the last level contains $M+\delta$ fewer elements. A* PULL *uses $O(h\frac{M}{B})$ I/Os and $O(hM \log N_\ell + \alpha\delta)$ time.*

*Proof.* We first prove, that the $M$ smallest uncorrupted elements of $\mathcal{L}$ are extracted, we then prove that buffers in $\mathcal{L}$ remain sorted, and finally we prove the time and I/O bounds.

1. To prove (1) we basically need to prove that the following invariant holds: After performing an iteration of the PULL algorithm at level $i$ the $M-\delta-\alpha$ smallest uncorrupted elements of $A$ are smaller than any uncorrupted element in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_i$. Since $|A| = M+\delta$, this invariant implies that the $M$ first elements of $A$ are smaller than any uncorrupted elements in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_i$.

   Before proving the invariant we need to prove that the smallest $M+\delta-\alpha$ uncorrupted elements of $C$ are in $C_1$ which become the new $S$. Let $X$ denote the set containing the $M+\delta-\alpha$ smallest uncorrupted elements in $C$ and let $m = \max\{x \in X\}$. Since $C$ is faithfully ordered, all elements of $X$ appear faithfully ordered in $C$ and there are no uncorrupted elements larger than $m$ in positions before $m$ by definition of $X$. Thus, the only elements not in $X$ that can be stored in $C$ before $m$ are corrupted elements. There are at most $\alpha$ of these and thus, the index of $m$ is smaller than $M+\delta-\alpha+\alpha = M+\delta$, implying that all of $X$ is in $C_1 = C[1, \ldots, M+\delta]$.

   We prove that the invariant holds by induction. It holds in the base case by Lemma 8.6. For the general case, assume that we have just completed the iteration at level $i$ and let $\alpha'$ denote the number of corruptions before

the most recent iteration, and $\alpha \geq \alpha'$ the number of corruption after the iteration. Let $X'$ be the $M + \delta - \alpha$ smallest elements after the iteration and $X$ the $M+\delta-\alpha'$ smallest elements before. We need to prove that any element in $X'$ is smaller than any uncorrupted element in $\mathcal{L}_0, \ldots, \mathcal{L}_i$ under the assumption that any element from $X$ is smaller than any uncorrupted element in $\mathcal{L}_0, \ldots, \mathcal{L}_{i-1}$. Since all the elements of $X'$ are uncorrupted, it is enough to prove that the largest element, $m \in X'$ is smaller than all uncorrupted elements in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_i$.

We split the proof in two cases.

- $m \in X' - X$: In this case $m$ originates from $A \subset \mathcal{L}_i$. By the correctness of the merger, $m$ is smaller than everything remaining in $\mathcal{L}_i$. $m$ is also, by definition of $X'$, smaller than everything in $A - X'$. We need to prove that $m$ is also smaller than everything in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$. Assume there is an element $x \in X$ larger than $m$. If so, we have that $m \leq x \leq y$ for any uncorrupted element $y \in \mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$, by the induction hypothesis. We now prove that it is impossible for $m$ to be larger than all elements of $X$, we do this by contradiction and assume that $m$ is bigger than all $x \in X$. Thus, for $m$ to be the $M + \delta - \alpha'$ largest uncorrupted in $C_1$ it must be smaller than at least $|X| - |X'| + 1 = \alpha' - \alpha + 1$ elements from $X$. But since $m$ is uncorrupted and larger than all the elements of $X$ which are uncorrupted, each of the elements from $X$ that are not in $X'$ must be corrupted instead. Thus $\alpha' - \alpha + 1$ elements needs to have been corrupted in $S$ or corrupted during the merge, which is impossible by definition of $\alpha$ and $\alpha'$.
- $m \in X \cap X'$: By the induction hypothesis, $x$ is smaller than all uncorrupted elements of $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$. It remains to be shown that $m$ is also smaller than all elements of $\mathcal{L}_i$. Since $m$ was from $X$, and thus *not* from $A$, we know that the number of elements in $S$, but not in $X'$, is $|S - (X' \cap S)| \geq |S| - |X'| + 1 = \alpha' + 1$ and by definition of $\alpha'$, there is at least one uncorrupted element of $S$ not in $X'$ and the smallest of these elements are smaller than $x$ by definition of $X'$, but larger than all remaining uncorrupted elements in $S - S \cup X'$ and $\mathcal{L}_i$ by Lemma 8.6.

2. We need to prove that all lists remain sorted after PULL. The only change to the lists happen when elements are removed from the lists by the merger, and when the elements left in the merger are re-inserted in level $\ell$ or level $i$. Removing elements from a buffer does not change the order of the remaining elements. The second part was proved in Lemma 8.6.

3. The complexity of the PULL operation is upper bounded by the invocations of $merger(\ell, M + \delta)$ for $0 \leq \ell \leq h$.

□

### 8.6.2 Operations on Internal Buffers

The internal data structures, $D$ and $I$, are implemented using the optimal internal memory resilient dynamic dictionaries presented in Chapter 7. Recall that these dictionaries maintain $n$ elements under updates and searches in amortized $O(\log n + \delta)$ time per operation and use $O(n)$ space.

To insert $x$ into the priority queue, we simply insert it into $I$. If $I$ grows to size $M$ we push $M$ elements into $\mathcal{L}$. We do this by using the resilient binary merging algorithm to merge $I$ and $D$ into a new buffer $E$. We now re-insert the first $|D| - \delta$ elements of $E$ into a $D$, the next $\delta$ are put in $I$ and the remaining $M$ elements are pushed to $\mathcal{L}$ using the PUSH operation. If $|D| < \delta + 1$ now it is filled by a PULL operation.

A DELETEMIN finds the minimum element in $F$, in the first $\delta + 1$ elements of $D$ and of the first $\min(\delta + 1, |I|)$ elements of $I$ and returns the minimum of these three, the element is deleted from its buffer. To find the $\delta + 1$ smallest elements from $D$ and $I$ respectively, we need to extract the $\delta + 1$ minimum elements from the resilient dictionaries in Chapter 7. This dictionary stores the elements in buffers of size $\Theta(\delta)$ and one can move between successor and predecessor buffers in $O(\delta)$ time. If a DELETEMIN causes $D$ to become smaller than $\delta+1$, we *pull* $M+\delta$ from $\mathcal{L}$ and fill $D$ with the $M$ first elements. The last $\delta$ elements are put into $I$. If this causes $I$ to become larger than $M$ we empty $I$ into $\mathcal{L}$ as above.

Since $h$, the number of levels of $\mathcal{L}$, might not decrease even if DELETEMIN is invoked many times, we use global rebuilding and rebuild the entire data structure every $\Theta(N)$ operations.

**Theorem 8.5** *Our data structure is a linear space resilient priority queue supporting operations* INSERT *and* DELETEMIN *in amortized* $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ *I/Os and* $O(\log N + \delta)$ *time, assuming* $\delta \leq M^{\varepsilon}$ *for* $0 \leq \varepsilon < 1$.

*Proof.* To bound the cost of DELETEMIN and INSERT we bound the number of levels in $\mathcal{L}$. Since a level $\mathcal{L}_\ell$ is pushed to higher levels only if it is full and since we use global rebuilding, the highest level in the priority queue is $h = O(\log_\gamma(N/M))$. Both PUSH and PULL use $O(1/B)$ I/Os and $O(\log \gamma)$ time per element they touch on each level. Therefore, the total time, including the operations on $I$ and $D$ is $O(h \log \gamma + \log M + \delta) = O(\log N + \delta)$ per element. Furthermore, we use a total of $O(\alpha\delta)$ time to cope with corrupted elements. Thus, the total time used to perform $N$ operations is $O(N((log_\gamma(N/M))(\log \gamma) + \log M + \delta) + \alpha\delta) = O(N \log N + \delta N + \alpha\delta)$, or amortized $O(\log N + \alpha\delta/N + \delta) = O(\log N + \delta)$ per operation.

The total I/O cost per element is $O(\frac{1}{B}h) = O(\frac{1}{B}\log_\gamma(N/M))$. Thus, the I/O complexity for $N$ operations is $O(N(\frac{1}{B})\log_\gamma(N/M)) = O(\frac{1}{1-\varepsilon}\text{Sort}(N))$, or $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ amortized per element. □

# Chapter 9

## Counting in Unreliable Memory

In this chapter we investigate the fundamental problem of counting in faulty memory. Keeping many reliable counters in the faulty memory is easily done by replicating the value of each counter $\Theta(\delta)$ times and paying $\Theta(\delta)$ time every time a counter is queried or incremented. We decrease the expensive increment cost to $o(\delta)$ and present upper and lower bound tradeoffs decreasing the increment time at the cost of the accuracy of the counters.

**Definition 9.1** *A resilient counter with* additive error $\gamma$ *is a data structure with an* increment *operation and a* query *operation. The query operation returns an integer between $v - \gamma$ and $v + \gamma$ where $v$ is the number of increment operations preceding the query.*

We investigate upper and lower bound tradeoffs between the time needed for $n$ increase operations and the additive error of the counter. We only consider data structures where no information is stored in safe memory between operations, therefore the counters are stored completely in unreliable memory. Our results are summarized in Table 9.1.

**Our results** In Section 9.1 we prove that any resilient counter with non-trivial additive error must use $\Omega(\delta)$ space, and that a deterministic query operation requires $\Omega(\delta)$ time. Furthermore, we prove a lower bound tradeoff between the increment time and the additive error, stating that if an increment operation takes $t \leq \delta$ time, the additive error is at least $\lfloor \delta/t \rfloor$ in the worst case, i.e. (increment time) $\times$ (additive error) $\geq \delta$. The lower bounds suggest that an optimal resilient counting data structure is characterized by an $O(\delta)$ space bound, $O(t)$ increment time, $O(\alpha/t)$ additive error and $O(\delta)$ query time.

In Section 9.2.1 and 9.2.3 we provide deterministic data structures where both the increment time and the additive error depend on $\alpha$. The first result in Section 9.2.1 provides a tradeoff between the increment time and the additive error that does not blow up the space used by the data structure nor the query time. Given any $t \geq 1$ the data structure has additive error $\alpha/t$ and supports $n$ increments in $O(nt \log(\delta/t) + \alpha \log(\alpha/t))$ time. A small change to this data structure gives a data structure with additive error $\alpha \log \delta$ that supports $n$ increments in $O(n + \alpha \log \alpha)$ time. In Section 9.2.3 we describe a data structure

| Time ($n$ increments) | Query time | Additive error | Space | Section |
|---|---|---|---|---|
| $O(\delta n)$ | $O(\delta)$ | 0 | $O(\delta)$ | - |
| $O(nt\log(\delta/t)+\alpha\log(\alpha/t))$ | $O(\delta)$ | $\alpha/t$ | $O(\delta)$ | 9.2.1 |
| $O(n+\alpha\log\alpha)$ | $O(\delta)$ | $\alpha\log\delta$ | $O(\delta)$ | 9.2.1 |
| $O(n)$ | $O(\delta^2)$ | $O(\alpha^2)$ | $O(\delta)$ | 9.2.2 |
| $O(n+\alpha\sqrt{\delta})$ | $O(\delta)$ | $\alpha$ | $O(\delta)$ | 9.2.3 |
| Expected $O(n)$ | $O(\delta)$ | $\alpha$ | $O(\delta)$ | 9.2.4 |

Table 9.1: Overview of our upper bounds.

with additive error $\alpha$ that supports $n$ increments in $O(n+\alpha\sqrt{\delta})$ time. This is optimal for $n=\Omega(\alpha\sqrt{\delta})$.

In Section 9.2.2 we describe a deterministic data structure where the time used by an increment is independent of the number of possible corruptions. The data structure supports increments in $O(1)$ time in the worst case. The additive error of the data structure is $O(\alpha^2)$ and queries are supported in $O(\delta^2)$ time.

Finally, in Section 9.2.4 we present a randomized data structure with additive error $\alpha$, that supports $n$ increments in $O(n)$ time in expectation and supports queries in $O(\delta)$ time in the worst case. This is optimal up to constant factors.

The additive error of any of our resilient counters can be reduced by a factor of $t$ by using $t$ counters. Each increment operation increments all $t$ counters and the query operation returns the sum of all $t$ counters divided by $t$. However, this produces a new tradeoff by increasing the increment time and space by a factor of $t$. Similarly, any of our resilient counters can be used to create a new counter that supports both decrement and increment operations with the same additive error. This is achieved by using two counters; one to count the number of increment operations and one to count the number of decrement operations.

## 9.1 Lower Bounds and Tradeoffs

We present some simple lower bounds on space and time for resilient counters.
*Space.* Any resilient counter data structure with non-trivial additive error must use more than $\delta$ space. If the data structure uses $\delta$ space or less, the adversary can corrupt the entire structure and force a query operation to return any arbitrary value.
*Deterministic Query.* Any deterministic algorithm uses at least $\delta$ probes in the worst case for a query. If a query algorithm reads at most $\delta$ memory cells the adversary can simulate any value by corrupting $\delta$ cells. This means that the adversary can completely control the value returned by a query, making it impossible to get a non-trivial bound on the additive error.
*Deterministic Increment.* If an increment takes $k$ time the adversary can roll back the changes to the data structure done by the last $\lfloor\delta/k\rfloor$ increments, or do the changes to the data structure corresponding to $\lfloor\delta/k\rfloor$ increments. Thus, the counter has additive error at least $\lfloor\delta/k\rfloor$ in the worst case.

## 9.2 Data Structures

### 9.2.1 Replicating Bits

In this section we describe a data structure that is parameterized with an integer $t$, $1 \leq t \leq \delta$. The data structure uses $O(\delta)$ space and has additive error $\lfloor \alpha/t \rfloor$. The time used for $n$ increments is $O(nt \log(\delta/t) + \alpha \log(\alpha/t))$, and queries take $O(\delta)$ time.

*Structure.* The data structure maintains the bits of the binary representation of the counter value separately, each bit replicated depending on its significance as follows. For $i = 0, \ldots, \lfloor \log(\delta/t) \rfloor$ the $i$'th least significant bit is replicated $t2^{i+1}$ times in $t2^{i+1}$ different memory cells. The value of the remaining $w - \lfloor \log(\delta/t) \rfloor$ most significant bits are stored in a reliable variable $v$. The memory cells are stored in one array of size $O(\delta)$.

*Increment.* Increments are implemented as binary addition, where we consider the $i$'th bit to be one if at least $t2^i$ of the $t2^{i+1}$ copies of it are non-zero. The $i$'th bit is set by writing the value of the bit in all of the $t2^{i+1}$ copies.

*Query.* The query algorithm reliably retrieves the value of the $w - \lfloor \log(\delta/t) \rfloor$ bits stored in $v$. For the lower order bits, we add $2^i$ to the sum, for $i = 0, \ldots, \lfloor \log(\delta/t) \rfloor$, if at least $t2^i$ of the $t2^{i+1}$ copies of the $i$'th least significant bit are non-zero.

*Additive Error.* Since the value of the $i$'th bit is given by the majority value of $t2^{i+1}$ copies, the adversary must use $t2^i$ corruptions to alter the $i$'th bit. Changing the $i$'th bit changes the value stored in the data structure by $2^i$, yielding an additive error of $\lfloor \alpha/t \rfloor$.

*Complexity.* If no corruptions occur, we update the $i$'bit of the counter every $2^i$ increments, taking $O(t2^i)$ time. Similarly, we update $v$ after $\Theta(\delta/t)$ increments in $O(\delta)$ time. Therefore, if we ignore corruptions, the time used for $n$ increments is $O(nt \log(\delta/t))$.

The only way corruptions can influence the running time of increment operations is by changing the value of a bit. Assume the adversary corrupts the $i$'th bit, using $t2^i$ corruptions. After a number of increments a cascading carry affects this (corrupted) bit and the increment operation writes the $t2^{i+1}$ copies of the $i+1$'th bit. We charge the work needed to move the $t2^i$ corrupted bits to the corruptions that caused them. These corrupted bits can be charged in $\log(\delta/t) - i$ such cascading carries. However, when $k$ increments have been performed, where $kt > \alpha$, the time used by the increments alone is $O(kt \log \delta/t)$ dwarfing the time needed to deal with corruptions. Otherwise, the number stored in the data structure is at most $k + \alpha/t \leq 2\alpha/t$. Thus, the most significant bit written in an increment operation is the $\lceil \log(\alpha/t) \rceil$ least significant bit. We conclude that the extra time needed to deal with corruptions is $O(\alpha \log(\alpha/t))$.

**Theorem 9.1** *The counter structure uses $O(\delta)$ space and has additive error $\lfloor \alpha/t \rfloor$. The time used for $n$ increments is $O(nt \log(\delta/t) + \alpha \log(\alpha/t))$ and queries take $O(\delta)$ time.*

**Trading off Additive Error for Increment Time** We can reduce the time for $n$ increments to $O(n + \alpha \log \alpha)$ by storing the $\lfloor \log \log \delta \rfloor$ least significant bits in the same memory cell. For $i = \lfloor \log \log \delta \rfloor + 1, \ldots, \log \delta$ the $i$'th least significant bit is replicated in $2^{i+1}/\lfloor \log \delta \rfloor$ memory cells. The remaining bits are stored in a reliable value $v$ as before. One corruption can change the $\lfloor \log \log \delta \rfloor$ least significant bits causing an additive error of at most $\lfloor \log \delta \rfloor$, and $2^i/\lfloor \log \delta \rfloor$ corruptions are needed to corrupt the $i$'th bit. The increment and the query are basically the same.

**Corollary 9.1** *The counter structure uses $O(\delta)$ space and has additive error $\alpha \log \delta$. The time used for $n$ increments is $O(n + \alpha \log \alpha)$ and queries use $O(\delta)$ time.*

### 9.2.2 Round-Robin Counting

In this section we describe a data structure that uses $O(\delta)$ space and has $O(\alpha^2)$ additive error. Increments are supported in constant time, and queries use $O(\delta^2)$ time.

*Structure.* The data structure consists of an array $A$ of $k = 2\delta + 3$ integers $C_1, \ldots, C_k$ used as counters, and a round-robin index $i$. The structure is initialized by setting all counters to zero and $i$ to one. We denote by *corrupted counter* a counter that has been changed directly by the adversary.

*Increment.* If $i$ is not in the range $1, \ldots, k$, it has been corrupted and we reset it to one. Next, we increment first $C_i$ and then $i$. If $i$ becomes $k + 1$ we set it to one. Note that $i$ could have been corrupted to a value in $1, \ldots, k$, but we do not check if this happened.

Let $v_j$ be the number of times the increment algorithm has incremented $C_j$, and let $v = \sum_{j=1}^{k} v_j$ denote the correct value of the counter. If no corruption has taken place, then $C_1 = \cdots = C_r = d + 1$ and $C_{r+1} = \cdots = C_k = d$, where $d = \lfloor v/k \rfloor$ and $r = v \mod k$. Furthermore, if no counter has been corrupted, $v = \sum_{j=1}^{k} C_j$, regardless of corruptions of the round robin index $i$.

*Query.* Let $\alpha_i$ be the number of times $i$ has been corrupted. The key observation for the query algorithm is that for any two uncorrupted counters, $C_a$ and $C_b$, we have $|v_a - v_b| \leq \alpha_i + 1$, which means that $|v/k - v_a| \leq \alpha_i + 1$.

First, we compute a value $m$ larger than or equal to at least one uncorrupted counter, and smaller than or equal to at least one uncorrupted counter. Since the difference between two uncorrupted counters is at most $\alpha_i + 1$, $m \in \{\frac{v}{k} - \alpha_i - 1, \frac{v}{k} + \alpha_i + 1\}$. After computing $m$, simply returning $mk$ yields an additive error of $O((\alpha + 1)k) = O((\alpha + 1)\delta)$. To improve the additive error we locate $O(\alpha)$ counters which are too far from $m$ and ignore them.

We store $m$ in safe memory and compute it as in [C1] as follows. Initially, we set $m$ to $-\infty$. The $k$ counters are scanned $\lceil k/2 \rceil$ times. In each iteration we update $m$ to the minimum counter larger than the current $m$. Since $k = 2\delta + 3$, after $\lceil k/2 \rceil$ iterations there exist two uncorrupted counters, such that one is smaller and one is larger than $m$.

Next, we find a bound, $x$, on the number of the counters that are too far away from $m$ as follows. Initially, we set $x$ to one. Then, the number of

counters $c$ outside the range $\{m-x,\ldots,m+x\}$ is counted in a scan. If $c \geq x$ we increment $x$ and recompute $c$. This process ends when $x$ becomes larger than $c$. Finally, we scan the $k$ counters maintaining a sum, initially zero, in safe memory. If a counter stores a value in the range $\{m-x, m+x\}$ we add it to the sum. If a counter is outside the range, it is far from $m$, and we add $m$ to the sum. Finally, we return the computed sum.

*Additive Error.* Let $\alpha_c$ be the number of times a counter was corrupted by the adversary. By definition, $\alpha_i + \alpha_c = \alpha \leq \delta$. First we recall that for any two uncorrupted counters, $C_a$ and $C_b$, we have $|v_b - v_a| \leq 1 + \alpha_i$, and that the value of $m$ is in the range $\{\frac{v}{k} - \alpha_i - 1, \frac{v}{k} + \alpha_i + 1\}$. Therefore, if $x \geq \alpha_i + 1$ in the above algorithm, then $c$, the number of counters that are not in the range $\{m-x, m+x\}$, is at most $\alpha_c$, the number of counter corruptions. At most $\alpha_c$ corrupted counters can be counted by $c$, and we conclude that when the algorithm terminates, then $x \leq \alpha_i + \alpha_c + 1$.

Let $S$ be the set of counters not counted by $c$, i.e. all counters in the range $\{m-x, m+x\}$. All uncorrupted counters in $S$ are unchanged and do not contribute to the error. Let $C_j$ be a corrupted counter in $S$. By definition of $m$ and $x$ we know that $|v_j - C_j| \leq |v_j - m| + |m - C_j| \leq \alpha_i + 1 + x \leq 2\alpha + 1$. Therefore, each corrupted counter in $S$ can affect the additive error by $O(\alpha)$. We add $m$ to the result for all counters outside the range $\{m-x, m+x\}$. By definition of $m$, the value for uncorrupted counters not in $S$ differs from $m$ by at most $\alpha_i + 1$. Similarly, for any corrupted counter $C_j$ not in $S$ the difference between $m$ and $v_j$ is at most $\alpha_i + 1$. There are at most $x = O(\alpha)$ counters not in $S$, and at most $\alpha_c$ corrupted counters in $S$, leading to an additive error of $O(\alpha^2)$.

*Complexity.* The increment operation uses $O(1)$ time to update a counter and the round robin index. The query time is given by the time used to compute $m$ and $x$, that is $O(\delta^2)$.

**Theorem 9.2** *The counter data structure described uses $O(\delta)$ space and has an additive error of $O(\alpha^2)$. Increments are supported in $O(1)$ time and queries in $O(\delta^2)$ time.*

### 9.2.3 Counting by Scanning Bits

We describe a counter data structure that uses $O(\delta)$ space with additive error $\alpha$. It performs $n$ increments in $O(n + \alpha\sqrt{\delta})$ time, and answers queries in $O(\delta)$ time. First, we describe a simpler data structure with an additive error of $\alpha$ that supports $n$ increments in $O(n + \alpha\delta)$ time. Subsequently, we reduce the cost for $n$ increments to $O(n + \alpha\sqrt{\delta})$.

*Structure.* The data structure stores an array $A$ of $\delta$ memory cells, a reliable variable $v$, and a round-robin index $i$. Each cell of $A$ is used to store a single bit. We initialize all values in $A$ to zero, $v$ to zero, and $i$ to one.

*Increment.* If $A[i] = 0$ we set $A[i] = 1$ and set $i = 1 + (i+1 \mod \delta)$. Otherwise, we count the number of non-zero entries in $A$. We add this number plus one (for the current increment) to $v$ and set all entries in $A$ to zero.

*Query.* We count the number $v'$ of non-zero entries in $A$, retrieve $v$, and return $v + v'$.

*Additive Error.* Every time we add a value, $k$, to the reliable value $v$ in an increment we have seen $k-1$ non-zero entries in $A$. The only way a cell in $A$ can be non-zero is if it was set to one by an earlier increment operation, or the adversary corrupted it. Conversely, a cell is set to zero either after updating the reliable value or by a corruption. Thus, the number returned by a query differs by at most $\alpha$ from the actual number of increments performed.

*Complexity.* If no corruptions occur, the increment operation takes $O(1)$ amortized time, since setting a value in $A$ to one takes $O(1)$ time and updating $v$ takes $O(\delta)$ time and occurs every $\delta+1$ increments. Every corruption to the round robin index $i$ or an element of $A$ can force us to scan $A$ and reliably add a value to $v$, and this takes $O(|A|+\delta) = O(\delta)$ time. Therefore, $n$ increments take $O(n+\alpha\delta)$ time.

**Improving Increment Time by Packing** We improve the time used for $n$ increments to $O(n+\alpha\sqrt{\delta})$ by packing elements in $A$ to an auxiliary array. In addition to the reliable value $v$ and the array $A$ of size $\delta$, we store an array $P$ of size $\delta$, which is logically divided into $\Theta(\sqrt{\delta})$ blocks of $\sqrt{\delta}$ consecutive memory cells.

*Increment.* First, we test if $i$ is in the range $\{1,\ldots,\delta\}$. If not then $i$ has been corrupted and we set it to one. Then, we test whether $A[i]=0$ and if so, we set $A[i]=1$ and increment $i$. If $i$ becomes $\delta+1$ we set $i$ to one. However, unlike the simpler data structure, if $A[i]\neq 0$, a *packing phase* is initiated. In the packing phase we scan $A$ from left to right starting from $A[1]$ until we encounter a zero, or the end of $A$ is reached. During the scan we count the amount, $c$, of non-zero entries read and set all these entries to zero. After the scan $i$ is set to one. Then, we set $c$ entries in $P$ to one as follows. Let $d_j$ be the index in $P$ of the first element in the $j$'th logical block. We scan $P$ from $d_1$. If we see an entry storing a zero, we set it to one, and decrement $c$. If we see something else we go to the start of the following logical block and continue. We stop the packing phase when $c$ reaches zero or a non-zero element, or the boundary of the last block is found. If $c>0$ after the packing phase, we count the amount of non-zero elements in $A$ and $P$ in a scan and set all entries to zero. This count summed with $c$ is added to $v$.

*Query.* The query operation returns the sum of $v$ and the number of ones in $A$ and $P$.

*Additive Error.* Similarly to the simpler data structure, each corruption can only change the value of the data structure by one. It follows that the additive error is $\alpha$.

*Complexity.* We analyze the time used between two consecutive updates of $v$ and this time-frame we denote a round. The array $A$ consists of a number of sections of non-zero elements separated by zeros. Note that the packing phase removes at least one section. If no corruptions occur, increments can only extend sections. A corruption, of a cell in $A$ or of the index $i$, may extend a section, connect two sections, create a section or split an existing section in two. The same things can happen in an increment following a corruption of the index $i$. Thus, the number of sections created during a round is bounded by

one plus the number of corruptions, and a section is moved only once in $P$.

Moving $t$ non-zero entries from $A$ to $P$ in a packing phase takes $O(t+\sqrt{\delta})$ time, and the clean ending the round takes $O(\delta)$ time. Let $c_p$ be the number of increments and $\alpha_p$ be the number of corruptions in the $p$'th round. Since the packing phase is called at most $\alpha_p + 1$ times, the time used in the $p$'th round is $O(c_p + \alpha_p\sqrt{\delta} + \delta)$. We show that the $O(\delta)$ time used for the clean can be payed for by the $c_p$ increments and the $\alpha_p$ corruptions, by charging $O(1)$ per increment and $O(\sqrt{\delta})$ per corruption.

If we copy elements to the $i$'th logical block in $P$ in a packing phase and encounter a non-zero entry before filling all the $\sqrt{\delta}$ cells, at least one cell in the block is corrupted. Furthermore, we never put elements in the $i$'th block again unless a new corruption occur, setting a zero in the first entry of the block. This means that the only block that is changed by a packing phase that is not completely filled or has a cell that has been corrupted since the last time it was updated, is the last block considered in the phase.

When an increment performs a clean, ending the round, the first block of all logical blocks contained a non-zero entry during the packing phase. We categorize the $\sqrt{\delta}$ logical blocks as *filled* blocks, *corrupted* blocks, and *last* blocks. A filled block is a logical block which a packing phase has filled with $\sqrt{\delta}$ non-zero entries, a corrupted block contains a cell that has been corrupted during the round and which is not filled, and a last block is a block that does not contain a corrupted cell, but was not completely filled during the packing phase that put a one in the first entry of the block.

There are at most $\alpha_p + 1$ packing phases in a round, thus at most $\alpha_p + 1$ last blocks, and at most $\alpha_p$ corrupted blocks. If there are $f$ filled blocks then we have performed at least $f\sqrt{\delta} - \alpha_p$ increments in the round. This means that there are $\sqrt{\delta} - f$ other blocks (corrupted, last) and since there are $O(\alpha_p)$ blocks that are not filled, $\sqrt{\delta} - f = O(\alpha_p)$. We have charged each increment $\Theta(1)$, which means that the increments have payed at least $f\sqrt{\delta} - \alpha_p$. It remains to charge $\delta - (f\sqrt{\delta} - \alpha_p) = \sqrt{\delta}(\sqrt{\delta} - f) + \alpha_p$ to the $\alpha_p$ corruptions. Since $\sqrt{\delta} - f = O(\alpha_p)$, we have charged enough if each corruption pays $\Theta(\sqrt{\delta})$. We conclude that $n$ increments take $O(n + \alpha\sqrt{\delta})$ time.

**Theorem 9.3** *The counter data structure uses $O(\delta)$ space and has additive error $\alpha$. The time used for $n$ increments is $O(n+\alpha\sqrt{\delta})$ and queries are answered in $O(\delta)$ time.*

### 9.2.4 Using Randomization to Obtain Fast Increments

In this section we describe a randomized data structure that uses $O(\delta)$ space and has additive error $\alpha$. The expected time used for $n$ increments is $O(n)$, and queries are supported in $O(\delta)$ time in the worst case. The data structure is similar to the data structures in Section 9.2.3 but randomization is used to find an empty cell fast.

*Structure.* The data structure stores an array $A$ of size $k = 3\delta$ and a resilient variable $v$. Initially, $v$ and all entries in $A$ are set to zero.

*Increment.* We pick a random index $r \in \{1, \ldots, k\}$ and *probe* $A[r]$. If $A[r] = 0$, we set $A[r] = 1$ and return. Otherwise, the probe *failed* and we do one of

two things: with probability $\frac{k-1}{k}$ we restart the increment operation and with probability $\frac{1}{k}$ we *clean* the array. The clean operation counts the number of non-zero entries in $A$ and adds this plus one (the current increment) to the reliable value $v$, then it sets all entries in $A$ to zero.

*Query.* The query operation is the same as the one in Section 9.2.3, it simply counts the number of non-zero entries in $A$ and returns the sum of this number and $v$.

*Additive Error.* As in Section 9.2.3 the additive error is $\alpha$ since each unreliable array entry contributes at most one to the result.

*Complexity.* The query operation simply scans $A$ and retrieves $v$ in $O(\delta)$ time. The expected time analysis of the increment operation is more involved. The sequence of $n$ increments is logically divided into $\lceil n/t \rceil$ rounds of $t = \lceil \delta/2 \rceil$ increments. We prove that the expected cost of each round is $O(t)$, and then the bounds follow from linearity of expectation. We split each full round in two parts, the first part consists of the increments performed before the first clean in the round, and the remaining increments are the second part. If a round does not do a clean, we additionally charge for repeatedly doing failed probes until a clean would be performed. When the first part starts, the state of the array $A$ could be anything. When the second part starts, the array stores only zero values. We divide the cost of the $t$ increments into three.

The cost of successful probes, the cost of failed probes and the cost of doing cleans. The cost of the successful probes is $O(t)$. The cost of failed probes, is divided into two, a cost for the failed probes in the first part and a cost for the failed probes in the second part. The first part ends when the first clean is performed. We charge the first failed probe in each increment to the increment itself. The remaining number of failed probes is upper bounded by the number of times we restart the increment operation before we clean, and a clean is performed with probability $\frac{k-1}{k}$. Thus, the probability of doing exactly $f$ additional failed probes is $(\frac{k-1}{k})^f \frac{1}{k}$. This means that the expected cost of failed probes in the first part is bounded by $t + \sum_{f=0}^{\infty} f(\frac{k-1}{k})^f(\frac{1}{k}) = O(t)$. In the second part we place at most $t$ ones in $A$ and the adversary can at most introduce $\delta$ non-zero entries. Therefore, during each increment in the second part, half of the entries in $A$ contains a zero. This means that for each increment in the second part we expect to do one failed probe implying that the expected cost of failed probes in the second part is linear in the number of increments. Each round makes one clean in the first part, and for each increment in the second part, the probability of doing a clean is at most $\frac{1}{2}\sum_{f=1}^{\infty} \frac{1}{2^f}(\frac{k-1}{k})^{f-1}\frac{1}{k} \leq 2/k$. Thus, the expected cost for doing cleans in the second part is $O(1)$ per increment, we conclude that the expected cost of a full round is $O(t)$.

Only the last round remains. If this is the first round, it has no first part, and by the analysis above the cost of this round is linear in the number of increments. If the last round is not the first round, the expected cost is $O(\delta)$ even if zero increments has been performed. We charge this cost to the second to last round.

**Theorem 9.4** *The counter data structure described uses $O(\delta)$ space and has additive error $\alpha$. The expected time used for $n$ increments is $O(n)$, and queries*

*use $O(\delta)$ time.*

## 9.3 Open Problems

The main open problem is whether there exists a data structure that given any $t \geq 1$ has additive error $O(\alpha/t)$, supports increments in $O(t)$ time and queries in $O(\delta)$ time. One resilient counter needs $\Omega(\delta)$ space. It would be interesting to see if one can store $k$ counters using $o(k\delta)$ space with each counter having a non-trivial bound on the additive error. Most of the counters presented require $\Theta(\delta)$ space for a reliable variable which seems hard to share among several counters. It may be interesting to see if one can use the safe memory to store some state to achieve this and possibly circumventing the lower bound tradeoff between increments and additive error.

Most resilient algorithms use the majority algorithm from [23] to implement reliable variables. The majority algorithm is defined for promise problems where we know that a majority element exists in the input. It would be interesting to solve the more general problem of finding the *mode* (most frequently occurring element). An $o(\delta^2)$ algorithm for this problem could help improve the query time of the counter from Section 9.2.2. We have considered counters that can increment and decrement by one. A natural extension of this is to consider general counters where an increment operation is parameterized by a value $\Delta$ to be added to the counter. Similar lower bounds apply: if $\Delta$ is unbounded and the update time is $o(\delta)$ then adversary is able to simulate an addition of an arbitrary variable. We therefore need to find a different error metric for general counters or alternatively investigate if there are efficient algorithms when $\Delta < \delta$.

# Bibliography

## Papers included in this thesis


[D1] G. S. Brodal, R. Fagerberg, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. Technical report, DAIMI PB-585, Department of Computer Science, Aarhus University, 2007.

[D2] G. S. Brodal and A. G. Jørgensen. A linear time algorithm for the $k$ maximal sums problem. In *Proceedings of the 32nd International Symposium Mathematical Foundations of Computer Science*, pages 442–453, 2007.

[D3] G. S. Brodal and A. G. Jørgensen. Selecting sums in arrays. In *Proceedings of the 19th International Symposium on Algorithms and Computation*, pages 100–111, 2008.

[D4] G. S. Brodal and A. G. Jørgensen. Data structures for range median queries. In *Proceedings of the 20th International Symposium on Algorithms and Computation*, pages 822–832, 2009.

[D5] G. S. Brodal, A. G. Jørgensen, and T. Mølhave. Fault tolerant external memory algorithms. In *Proc. of the 11th International Symposium on Algorithms and Data Structures*, pages 411–422, 2009.

[D6] G. S. Brodal, A. G. Jørgensen, G. Moruz, and T. Mølhave. Counting in the presence of memory faults. In *Proc. of the 20th International Symposium on Algorithms and Computation*, pages 842–851, 2009.

[D7] M. Greve, A. G. Jørgensen, K. D. Larsen, and J. Truelsen. Cell probe lower bounds and approximations for range mode. Submitted.

[D8] A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. In *Proc. of the 10th International Workshop on Algorithms and Data Structures*, pages 127–138, 2007.

## Other papers co-authered by the author

[C1] G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. In *Proceedings of the 15th Annual European Symposium on Algorithms*, pages 347–358, 2007.

## Other references

[1] P. Afshani. On dominance reporting in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, pages 41–51, 2008.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[3] L. Allison. Longest biased interval and longest non-negative sum interval. *Bioinformatics*, 19(10):1294–1295, 2003.

[4] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 534–543, Washington, DC, USA, 1998. IEEE Computer Society.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[6] R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In *Proc. of the 2nd Usenix Workshop on Electronic Commerce*, pages 1–11, 1996.

[7] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*, pages 125–136, 1997.

[8] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[9] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. of the 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[10] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[11] Y. Aumann and M. A. Bender. Fault tolerant data structures. In *Proc. of the 37th Annual Symposium on Foundations of Computer Science*, pages 580–589, 1996.

[12] S. E. Bae and T. Takaoka. Algorithms for the problem of $k$ maximum sums and a vlsi algorithm for the $k$ maximum subarrays problem. In *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 247–253. IEEE Computer Society, 2004.

[13] S. E. Bae and T. Takaoka. Improved algorithms for the $k$-maximum subarray problem for small $k$. In *Proceedings of the 11th Annual International Conference on Computing and Combinatorics*, pages 621–631, 2005.

[14] S. E. Bae and T. Takaoka. Improved algorithms for the $k$-maximum subarray problem. *Comput. J.*, 49(3):358–374, 2006.

[15] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[16] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38 – 72, 2002.

[17] F. Bengtsson and J. Chen. Efficient algorithms for $k$ maximum sums. In *Proc. of the 15th International Symposium on Algorithms and Computation*, pages 137–148, 2004.

[18] J. Bentley. Programming pearls: algorithm design techniques. *Commun. ACM*, 27(9):865–873, 1984.

[19] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[20] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *Eurocrypt*, pages 37–51, 1997.

[21] R. S. Borgstrom and S. R. Kosaraju. Comparison-based search in the presence of errors. In *Proc. of the 25th Annual ACM symposium on Theory of Computing*, pages 130–136, 1993.

[22] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *Proceedings of the 22nd Symposium on Theoretical Aspects of Computer Science*, pages 377–388, 2005.

[23] R. S. Boyer and J. S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[24] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[25] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via trees of small height. Technical Report ALCOMFT-TR-02-53, ALCOM-FT, May 2002.

[26] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. of the 6th Scandinavian Workshop on Algorithm Theory*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer Verlag, Berlin, 1998.

[27] K.-M. Chao and H.-F. Liu. Personal communication, February 2007.

[28] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[29] C.-H. Cheng, K.-Y. Chen, W.-C. Tien, and K.-M. Chao. Improved algorithms for the $k$ maximum-sums problems. *Theoretical Computer Science*, 362(1-3):162–170, 2006.

[30] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE micro*, 23(4):14–19, 2003.

[31] C. A. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Dept. of Computer Science, Stanford University, 1972.

[32] K. Diks and A. Pelc. Optimal adaptive broadcasting with a bounded fraction of faulty nodes (extended abstract). In *Proc. of the 5th Annual European Symposium on Algorithms*, pages 118–129, 1997.

[33] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[34] D. Eppstein. Finding the $k$ shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1999.

[35] T.-H. Fan, S. Lee, H.-I. Lu, T.-S. Tsou, T.-C. Wang, and A. Yao. An optimal algorithm for maximum-sum segment and its application in bioinformatics. In *Proceedings of the 8th International Conference on Implementation and Application of Automata*, pages 46–66, 2003.

[36] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming*, volume 4051 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2006.

[37] I. Finocchi, F. Grandoni, and G. F. Italiano. Designing reliable algorithms in unreliable memories. *Computer Science Review*, 1(2):77–87, 2007.

[38] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. In *Proc. of the 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 547–553, 2007.

[39] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *Proc. of the 36th Annual ACM Symposium on Theory of Computing*, pages 101–110, 2004.

[40] I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, 2008.

[41] G. N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.

[42] G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in x+y and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982.

[43] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining with optimized two-dimensional association rules. *ACM Trans. Database Syst.*, 26(2):179–213, 2001.

[44] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th annual ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[45] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the 16th String Processing and Information Retrieval Symposium*, pages 1–6, 2009.

[46] L. Gasieniec and A. Pelc. Broadcasting with a bounded fraction of faulty nodes. *Journal of Parallel and Distributed Computing*, 42(1):11–20, 1997.

[47] B. Gfeller and P. Sanders. Towards optimal range medians. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, pages 475–486, 2009.

[48] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

[49] S. Hannenhalli and S. Levy. Promoter prediction in the human genome. *Bioinformatics*, 17:S90–96, 2001.

[50] S. Har-Peled and S. Muthukrishnan. Range medians. In *Proceedings of the 16th Annual European Symposium on Algorithms*, pages 503–514, 2008.

[51] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[52] J. Hastad and T. Leighton. Fast computation using faulty hypercubes. In *Proc. of the 21st Annual ACM Symposium on Theory of Computing*, pages 251–263, 1989.

[53] J. Hastad, T. Leighton, and M. Newman. Reconfiguring a hypercube in the presence of faults. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing*, pages 274–284, 1987.

[54] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33:518–528, 1984.

[55] X. Huang. An algorithm for identifying regions of a dna sequence that satisfy a content requirement. *Computer Applications in the Biosciences*, 10(3):219–225, 1994.

[56] G. J. Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988.

[57] J. JáJá, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proceedings of the 15th International Symposium on Algorithms and Computation*, pages 558–568, 2004.

[58] A. Joffe. On a set of almost deterministic $k$-independent random variables. *Annals of Probability*, 2(1):161–162, 1974.

[59] D. B. Johnson and T. Mizoguchi. Selecting the kth element in $X + Y$ and $X_1 + X_2 + \cdots + X_m$. *SIAM J. Comput.*, 7(2):147–153, 1978.

[60] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. D. Thomborson, and A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors (extended abstract). In *Proc. of the 31st Annual Symposium on Foundations of Computer Science*, pages 285–296, 1990.

[61] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., 1998.

[62] D. Krizanc, P. Morin, and M. H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

[63] S. Kutten and D. Peleg. Fault-local distributed mending. *Journal of Algorithms*, 30(1):144–165, 1999.

[64] S. Kutten and D. Peleg. Tight fault locality. *SIAM Journal on Computing*, 30(1):247–268, 2000.

[65] K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Transactions on Computers*, 40(9):1081–1084, 1991.

[66] F. T. Leighton and B. M. Maggs. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proc. of the 30th Annual Symposium on Foundations of Computer Science*, pages 384–389, 1989.

[67] T.-C. Lin and D. T. Lee. Efficient algorithms for the sum selection problem and $k$ maximum sums problem. In *Proceedings of the 17th International Symposium on Algorithms and Computations*, pages 460–473, 2006.

[68] T.-C. Lin and D. T. Lee. Randomized algorithm for the sum selection problem. *Theor. Comput. Sci.*, 377(1-3):151–156, 2007.

[69] Y.-L. Lin, T. Jiang, and K.-M. Chao. Efficient algorithms for locating the length-constrained heaviest segments, with applications to biomolecular sequence analysis. In *Proc. of the 27th International Symposium of Mathematical Foundations of Computer Science*, pages 459–470, 2002.

[70] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *J. Comput. Syst. Sci.*, 57(1):37–49, 1998.

[71] Y. Nekrich. Orthogonal range searching in linear and almost-linear space. In *Proceedings of the 10th International Workshop on Algorithms and Data Structures*, pages 15–26, 2007.

[72] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the 4th Annual ACM symposium on Theory of computing*, pages 137–142, 1972.

[73] C. Okasaki. Purely functional random-access lists. In *Functional Programming Languages and Computer Architecture*, pages 86–95, 1995.

[74] S. Park and B. Bose. All-to-all broadcasting in faulty hypercubes. *IEEE Transactions on Computers*, 46(7):749–755, 1997.

[75] M. Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proceedings of the 39th ACM Symposium on Theory of Computing*, pages 40–46, 2007.

[76] M. Pătraşcu. (Data) STRUCTURES. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 434–443, 2008.

[77] M. Patrascu and M. Thorup. Higher lower bounds for near-neighbor and further rich problems. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 646–654, 2006.

[78] H. Petersen. Improved bounds for range mode and range median queries. In *Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science*, pages 418–423, 2008.

[79] H. Petersen and S. Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2008.

[80] U. F. Petrillo, I. Finocchi, and G. F. Italiano. The price of resiliency: a case study on sorting with memory faults. In *Proc. of the 14th Annual European Symposium on Algorithms*, pages 768–779, 2006.

[81] S. Pettie and V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 713–722, 2002.

[82] D. K. Pradhan. *Fault-tolerant computer system design.* Prentice-Hall, Inc., 1996.

[83] B. Ravikumar. A fault-tolerant merge sorting algorithm. In *Proc. of the 8th Annual International Conference on Computing and Combinatorics*, pages 440–447, 2002.

[84] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In *Proc. of the 26th Annual International Symposium on Fault-Tolerant Computing*, pages 394–403, 1996.

[85] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In *Proc. of the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 2–12, 2002.

[86] D. D. Sleator and R. E. Tarjan. Self adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.

[87] T. Takaoka. Efficient algorithms for the maximum subarray problem by distance matrix multiplication. *Electr. Notes Theor. Comput. Sci.*, 61, 2002.

[88] H. Tamaki and T. Tokuyama. Algorithms for the maximum subarray problem based on matrix multiplication. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 446–452, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[89] Tezzaron Semiconductor. Soft errors in electronic memory - a white paper. http://www.tezzaron.com/about/papers/papers.html, 2004.

[90] A. J. van de Goor. *Testing Semiconductor Memories: Theory and Practice.* ComTex Publishing, Gouda, The Netherlands, 1998. ISBN 90-804276-1-6.

[91] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[92] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science*, 2(4):305–474, 2008.

[93] R. Y. Walder, M. R. Garrett, A. M. McClain, G. E. Beck, T. M. Brennan, N. A. Kramer, A. B. Kanis, A. L. Mark, J. P. Rapp, and V. C. Sheffield. Short tandem repeat polymorphic markers for the rat genome from marker-selected libraries. *Mammalian Genome*, 9(12):1013–1021, December 1998.

[94] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 421–430, 2001.

[95] A. C.-C. Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981.

[96] A. C.-C. Yao. Space-time tradeoff for answering range queries (extended abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 128–136, 1982.

[97] A. C.-C. Yao. On the Complexity of Maintaining Partial Sums. *SIAM J. Comput.*, 14(2):277–288, 1985.

[98] S. S. Yau and F.-C. Chen. An approach to concurrent control flow checking. *IEEE Transactions on Software Engineering*, SE-6(2):126–137, 1980.