RANGE MEDIAN ALGORITHMS

MASTER'S THESIS

DAVID KJÆR— 20050713

ADVISOR: GERTH STØLTING BRODAL

AUGUST 31ST 2010

DEPARTMENT OF COMPUTER SCIENCE
AARHUS UNIVERSITY

Abstract

The range median problem is that of preprocessing a list or array such that the median of any arbitrary subarray can be found rapidly. The challenge is to use as little space as possible in the process.

This Thesis is divided into two parts. The first part contain a general survey on structures solving the range median problem. Included in the survey are solutions that solve the range selection problem and thereby the range median problem by inclusion. As a supplement to these algorithms a brief summary of the state of popular range problems is included in the survey. The majority of the time is spent describing the solutions given in [BGJS10] written by Gfeller, Sanders, Brodal and Jørgensen. The range problems will be considered in both a dynamic and a static setting.

The second part describes implementations of some of the data structures given in [BGJS10] as well as the outcome of experiments performed on these.

Acknowledgements

I would like to thank my advisor Gerth for his invaluable insights and advice during this project.
I would also like to thank my parents and my office mate Kasper for putting up with me for these six months.

Contents

List of Algorithms

Chapter 1

Introduction

The world is continuously becoming flooded with information. Aside from the information we ourselves are generating and storing like text documents, images and spreadsheets that is shared and indexed, huge amounts of data are collected without human intervention. This includes sensory input from the security system of a nuclear power plant or images taken by the Hubble Telescope. Answering queries efficiently on large amount of data is probably the most important challenge computer scientists face in the coming years.

In this thesis we will direct our focus to a specific class of data, that is a list of elements where the position of an element in the list is significant. For instance, a list periodic measurements are ordered by time, possibly even appended a time stamp, and the data is of no use if the list is reordered. Notice, that in real life this list could be infinite. We will abstract away form this, however, and assume that we are given a finite list of some length n. Furthermore, we will impose a total ordering on the elements stored in the list. This does not always make sense. How would one order images for instance?

Having large amounts of data is in itself not very interesting. Ideally, we would like to be able to answer interesting queries about the data. Given that elements can be totally ordered interesting queries of course include: the minimum and maximum element, the most frequent element, also called the mode, or simply the r'th largest element in the list. Efficient linear time algorithms exist for all of these problems. In some cases we are really not interested in answers to queries involving the entire list. An example could be a web marketer wanting to analyze peoples Internet surfing habits. For the sake of this example imagine he is in possession of a list of measurements describing the IP-traffic on a web server each minute for a year. There are several values of interest here. For instance, the minimum and maximum value of simultaneous visitors. In fact the median value is probably the most interesting value here since it would give an approximation of the average while simultaneously leveraging the impact of statistical outliers caused by downtime or bot attacks. However, knowing the median of the entire list does not yield a lot of valuable information. Especially, it does not give any information about peoples' habits. In order to collect valuable information the web marketer would have to

sample the median in different time spans giving him an overview of the actual behavior of users. But that poses a problem. We know that finding the median of the list is easy, [BFP+72]. We do not know whether or not sampling arbitrary ranges is easy. Bear in mind, that the measurements are naturally sorted by time and that the different samples could contain overlapping time spans.

The example above calls for an efficient solution of what is called a range problem. The word range here refers to an contiguous sublist within the input list. Informally speaking we can define range problems in the following way:

Definition 1 *Given a list of values* $\boldsymbol{A}$ *of length* n *calculate a function* $f(j,i)$ *where* $1 \leq i \leq j \leq ns$. *Formulating the function* f *is the range problem.*

Since any range can be translated such that its endpoints correspond to indexes between 1 and n the above definition makes sense. Throughout this thesis I will refer to the f as a data structure and an actual call to f as a query.

There are exactly $\binom{n}{2}$ different ranges that we could query. Hence, assuming there is plenty of time to preprocess the list calculating a table of all possible ranges in advance would render us capable of answering any range query constant time. Using $O(n^2)$ words of space this approach very quickly becomes infeasible. The other extreme is to copy the sublist specified by the range endpoints into a new array and running the most time efficient algorithm. This uses only a constant times the memory already used by the list. Unfortunately, the query time now is at least linear.

Surprisingly, to the best of my knowledge the range median problem was first considered ub 2005 by Krizanc, Morin and Smid, [KMS05]. Since then, only smaller improvements has been done and the solutions proposed have been very complicated. Quite recently Gfeller, Brodal, Sanders and Jørgensen published a paper with solid improvements of the previous algorithms, [BGJS10].

This thesis will survey the solutions known for the range median problems. However, we will especially focus on the solutions presented in [BGJS10]. As a conclusion I will present the implementations I have done on some of the data structures and the result of experiments performed on these implementations.

Here follows a short introduction to the individual chapters:

- Chapter 2 will briefly sum up the previous work done on the more popular range problems, as well as give a more in depth characterization of the previous work on the range median problem.
- Chapter 3 will present the algorithms given in [BGJS10].
  - For the static version of the problem, that is the input list will not change, an $O(k \log n + n \log k)$ using $O(n \log n)$ space solution will be described in section 3.1.1. This algorithm is so simple that the above upper bounds holds for the Pointer Machine model [1]. Section 3.1.2 will present an

[1] Very restrictive model only allowing memory to be accessed by pointers

improvement of the above data structure using only $O(n)$ space. Section 3.1.3 will take a different approach to solving the static version of the range median problem an will arrive at a data structure achieving $O(\log n / \log\log n)$ query time using only linear space and $O(n \log n)$ preprocessing time.

- Section 3.2 will introduce the dynamic version of the range median problem and a very efficient solution obtaining $O((\log n / \log\log n)^2)$ time worst case for queries. The updates can be performed subject to the same upper bound but this time amortized. The data structures uses $O(n(\log n / \log\log n)^2)$ words of space.
- Section 3.3 Will describe a space efficient constant time data structure. That is, the data structure uses $O(n^{3/2})$ words of space worst case achieving constant query time on the average. Unfortunately, it does not seem possible to generalize this to a deterministic worst case constant time data structure having the above space bound.

- Chapter 4 contains descriptions of four implementations of data structures solving the range median problem. In addition a series of experiments will be performed and discussed in an attempt to measure the efficiency of these implementations in terms of space and time.
- Chapter 5 will contain a brief summary, and a discussion of future work.

The code referenced in Chapter 5 is enclosed on a compact disc.

Chapter 2

Range Problems

In this section I'll give a quick summary of the history and current bounds for several different range problems. The main emphasis will be put on the range median problem.

2.1 Group Operators

This is a relatively simple set of range problems with very efficient solutions in both the static and dynamic cases. Examples of query problems belonging to this class are the range sum, range product and similar simple mathematical queries. This class of range problems sees the input array as a collection of elements from some set S. The query operator $\circ$ and S form a mathematical group $(\mathsf{S}, \circ)$. That is, there exists a zero-element 0_{S} and for all elements $s \in \mathsf{S}$ there exists a unique element s^{-1} such that $s \circ s^{-1} = 0_{\mathsf{S}}$. The static case is now immediately solvable in constant time using only linear space. For a specific group $(\mathsf{S}, \circ)$ and input array $\mathbf{A}$ we simply calculate a table **tab** of size $|\mathbf{A}|$ defined by

$$\mathbf{tab}[k] = \begin{cases} 0_{\mathsf{S}} & \text{if } k = 0 \\ \mathbf{tab}[k-1] \circ \mathbf{A}[k] & \text{if } 1 \leq k \leq n \end{cases}$$

The array **tab** is often referred to as a prefix array. Solving a query is now done by lookups in **tab**. More precisely the query $\mathbf{Q}(i, j)$ is equal to $\mathbf{tab}[j] \circ \mathbf{tab}[i-1]^{-1}$.

The dynamic case is significantly harder. In fact, Patrascu and Demaine proved a lower bound saying that comparions based data structure requires $\Omega(\log |\mathbf{A}|)$ time for both queries and updates in the cell probe model, [PD06].

This is, in fact, a tight bound. A way to achieve the upper bound is to construct a binary search tree, ordered by index, on top of the input array $\mathbf{A}$. Each internal node of this tree stores the collected sum, or what ever group operator is in use, of its children. Updates to element k of the input array is handled by searching for the leaf of corresponding to element k and adding the difference between the old value and the new value to all nodes on the search path. The following method calculates the sum of $\mathbf{A}[1, k]$ for some $1 \leq k \leq n$: Search for the leaf k. Backtrack through the root-to-leaf path while maintaining a sum of the values stored in the internal nodes.

This sum is initialized to the value stored in $\mathbf{A}[k]$. While visiting an internal node, you add the sum stored in this node. If the search navigated though the left subtree, however, the value stored in the right subtree is subtracted from the sum.

The query is now answered by applying the above method to j and i and subtract the second result from the first. The height of the tree is $O(\log |\mathbf{A}|)$. Hence, the number of nodes visited during an update or query is at most this value.

2.2 Semigroup Operators

Range problems for semigroup operators are harder to solve than group operators, since an inverse group operation is not available. However, the problem can still be solved very efficiently. Using $O(n)$ preprocessing time allows any query to be answered in time $O(\alpha(n))$ in the static case. The idea is to construct a directed acyclic graph where each node stores a preprocessed value. The values are such that any range query can be answered by summing the values of $O(\alpha(n))$ nodes.

Actually this is a consequence of a more general result proved in [Yao82]. This states that using $O(cn)$ preprocessing time a query can be solved in time $O(\alpha_c(n))$, where α_c refers to the $(c/2)$-th function of the primitive recursive hierarchy. A matching lower bound is shown in [Yao85].

Minimum and maximum are special semigroup operators in this context, since a comparison based data structure allowing for constant time queries using only $O(n)$ preprocessing time and space exists for the static case. Algorithms obtaining this bound does so by reducing the range maximum and range minimum problems to the lowest common ancestor problem. A more recent relatively simple construction can be found in [BFC04].

2.3 Mode

The range mode problem is the problem of finding an element in a range that occurs at least as often as any other. This is neither a group nor a semigroup problem. It is, like the range median problem, a relatively new subject of research, first considered by Krizanc, Morin and Smit, [KMS05]. It is still a rather unexplored problem and one would expect large improvements in the future compared to the current solutions. The original article solves the range mode problem in query time $O(n^\epsilon \log n)$ using $O(n^{2-2\epsilon})$ space, where $0 < \epsilon < 1$. For obtaining constant query time the authors constructed a solution using $O(n^2 \log\log n / \log n)$ space. These bounds were later improved by [PG09] to $O(n^\epsilon)$ query time using $O(n^{2-2\epsilon})$ space, and $O(n^2 \log\log n / \log^2 n)$ space with constant query time, respectively.

For a value array of size n, Greve, Jørgensen, Larsen and Truelsen very recently showed a lower bound space/time tradeoff in the cell probe model, [GJLT10]. The

result states that a data structure using S memory cells, with word size w spends at least $\Omega(\frac{\log n}{log(Sw/n)})$ time solving a query. The authors proved this bound by a complicated construction involving reduction to the problem of retrieving the frequency of the mode, and then using a general result in communication complexity. Notice that this leaves a significant gap to the best known upper bounds stated above.

2.4 Range Median

The range median problem was first considered by Krizanc, Morin and Smit, [KMS05]. The same article that gave birth to the range mode problem.

The range median problem will be the main focus of this thesis. Many times, however, it will be discussed in context of its generalization the range selection problem. The more general notion of selection covers the process of obtaining the element of rank k from a collection. A range selection query does exactly that on a range. The underlying collection could be a tree or a list for instance. In this thesis I will limit myself to discussing the range median and range selection problems on lists even though the original article present solutions on a tree as well. In fact, for the most part I will assume that the list is given as an array facilitating random access for constant time retrieval of any element.

The selection problem is obviously related to the sorting problem. We know that sorting a list of size n in the comparison model requires $O(n \log n)$ comparisons. Since it is possible to sort a list by doing n range selection queries on the range $[1,\ n]$, one for each rank, the combined number of comparisons of preprocessing and performing the n queries most be above $\Omega(n \log n)$. The range median problem is subject to the same lower bound. The reduction is as follows.

Given an input array **A**, "the sorting instance", of size n we will construct an array **B** of size $4n$ such that n range median queries determines the complete order of **A**. The array contains three sections. The first section comprises n elements which are all set to ∞. The second section also of size n consists of the elements in **A**, whereas the last section consists of $2n$ elements all set to $-\infty$. The sections are inserted in this order. Now the range $[1,\ 2n]$ consists of n $-\infty$ values and the original elements of **A**. The median of this range is the minimum element of **A**. Similarly, the element of rank two is the range median of $[1,\ 2n+2]$. In general, the element of rank k in **A** can be found by doing a range median query on $[1,\ 2n+2(k-1)]$. It is evident, that n queries for incrementing ranks would yield a sorted version of **A**. We used $O(n)$ additional time and space constructing **B**. Hence, the comparisons used for preprocessing **B** and performing n range queries on this would be $\Omega(n \log n)$.

It should be noted that this does not show a lower bound of $\Omega(\log n)$ comparisons per range median/selection query. In fact, using $O(n \log n)$ comparisons finding the permutation of **A** we could find the median of any range in constant time by a simple lookup.

In this section I will give a brief introduction to some of the historic solutions proposed for the range median problem by various authors. Since the dynamic version is quite recent all solutions in this section will solve the static online or offline problem. I will give the solutions in chronological order which in turn gives an increasingly better time/space product for the tradeoffs and decreasing space usage for the constant time solutions.

For the remainder of this chapter I will present quite a large number of data structures. These structures represent the historic contributions that has been made to the range median problem.

The first structure is by Krizanc, Morin and Smid. It works only in the static case and was published in the very same article that first defined the range median problem. It does not work for arbitrary ranks within the range though. The data structure uses $O(n \log n^2 / \log \log n)$ space achieving logarithmic query time.

The second structure is from the same article. It does not work for arbitrary ranks but is based on the standard range tree data structure. The data structures answers queries in $O(n^\epsilon)$ for any $\epsilon > 0$ using only linear space.

The authors actually presents a third data structure achieving linear time using $O(n^2 \log \log n / \log n)$. This data structure is not described in this Thesis merely noted.

The following two structures are attributed to Holger Petersen. He focuses on creating constant time solutions and decreasing the space usage.

The first structure achieves constant time using $O(n^2 \log^k n / \log n)$ [1]. His second approach uses $O(n^2 \log \log^2 n / \log^2 n)$. Both of these structures are based on techniques from succinct data structures and are packed with details. I will abstract away from some of these in my description below.

The final subsection is devoted to a interval tree based structure my Har-Peled and Muthukrishnan. This obtains an impressive $O(n \log k + k \log k \log n)$ using only $O(n \log n)$ space for k queries given as batch. On the downside this structure is very complex and to achieve the worst case bound mentioned the queries have to be known in advance. If this is not the case the query time becomes amortized.

Krizanc, Morin and Smid

The solution given in [KMS05] is the first stated for the static range median problem on a list. In fact the article presents two different solutions which I will describe in turn.

[1] $\log^{(1)} n = \log n, \log^{(k)} n = \log \log^{(k-1)} n$

Range Counting Data Structure

The input array **A** is divided into b blocks – or slabs – each of size n/b where n are the number of elements contained in the input array. The constant b should be at least two. Each of the b blocks are then preprocessed, in a way that will be described shortly, such that any range median query can be answered using lookup tables and a sequence of searches on a special *persistent* binary tree.

The data structure stores $2n/b + 1$ elements for each of the $\binom{b}{2}$ pairs of blocks in an auxilliary table. These represent the n/b elements on either side of the median of the elements contained in the *multislab* between and including these blocks. The $2n/b + 1$ elements are kept as augmented binary search trees which is a structure that will be defined shortly. We call these slab lists. Additionally, to each block we associate partially persistent augmented binary search trees. These two trees contain the elements inserted in increasing and decreasing order of index, respectively. The division is then repeated, now on block level, ending when each block has constant size.

Now to answer a query the data structure identifies the blocks containing the range endpoints and performs a clever selection procedure on the union of three components: the slab list stored for these two blocks and a search tree from each block. Now we have an overview of the structure let us have a closer look at the details.

The data structure uses the following crucial lemma:

Lemma 2 *Let A, B and C be multisets with $|A| = |B| = k$. The median of $A \cup B \cup C$ is either in A or C or is within rank k of the median in B, i.e. has rank within $[\lfloor |B|/2 \rfloor - k, \lceil |B|/2 \rceil + k]$*

Proof It is clear that the rank of the median of $A \cup B \cup C$ is equal to $\lfloor B/2 \rfloor + k$, since the size of A and C was k.

Let b_1 be the element of rank $\lfloor |B|/2 \rfloor - k$ in B. The rank of b_1 can at most increase by $2k$ in the union and therefore it will have rank smaller than or equal to the final median. Let b_2 be the element of rank $\lceil |B|/2 \rceil + k$ in B. b_2 will at least retain this rank as it can only increase. Hence, the median of the union is bounded by b_1 and b_2.

Let x be an element in B which is smaller than b_1. The rank of c $A \cup B \cup C$ must be strictly smaller than $\lceil |B|/2 \rceil + k$. Analogously any element with rank higher than b_2 must have strictly greater rank in the final union than that of the median. Therefore the median must lie between b_1 and b_2. □

The *augmented binary search tree* is a binary search tree where each internal node is decorated with the size of its subtree – or node count. For this application we assume that elements are stored in both leafs and internal nodes. For any node the rank of the element it stores is just the node count of the left child plus one. Note that, using the method of finding the rank of a node just described the tree forms a binary search tree on rank. The authors define the following lemma giving

a method to find an element of arbitrary rank within the union of three augmented binary search trees $T_a,\ T_a,\ T_b$.

Lemma 3 *Giving three augmented search trees,$T_a,\ T_a,\ T_b$, the element of any giving rank within the tree trees can be found in time* $O(h(T_a) + h(T_b) + h(T_c))$*, where the h-function represents the height of the tree.*

This implies that if balanced binary search trees are used as base for the augmented tree the median can be found in worst case $O(\log n)$ time. The basic idea of the proof is to order the trees by the value of their roots, and then search using the node count to eliminate a subtree of either the maximum or the minimum tree. The details can be found in the appendix to the original article.

As mentioned the data structure uses a partially persistent version of the augmented binary search tree. Unfortunately it is not possible to use the conventional methods for making binary search tree persistent with constant space overhead. According to the authors this will not work on an augmented binary search tree. Hence, this data structure utilizes the simpler method of path-copying, [DSST89]. This means that a tree that has been subject to k updates requires $n \log k$ space.

We will denote the two partially persistent augmented binary search trees stored for the k'th block $T_{f_{B_k}}$ and $T_{r_{B_k}}$. The tree $T_{f_{B_k}}$ contains the elements of B_k inserted in increasing order by index whereas $T_{r_{B_k}}$ contain the same elements inserted in decreasing order.

Now consider a range median query $\mathbf{Q}(i, j)$. The blocks containing the left and right endpoints of the range are identified by index $i' = \lceil i/(n/b) \rceil$ and $j' = \lceil j/(n/b) \rceil$, respectively. It is important that $i' \neq j$. If this is not the case we simply recurse on $B_{i'}$ until we find a level where it is.

A tree containing all elements from $B_{i'}$ that are inside the query range corresponds to version $(i \ div\ b) \cdot (n/b) - i\%b + 1$ of $T_{r_{B_i}}$. Analogously, the tree containing all elements from $B_{j'}$ inside the query range is found using version $j\%b$ of $T_{f_{B_{j'}}}$. Next we obtain the $2n/b$ elements stored for the slab of blocks corresponding to $B_{i'+1}$ and $B_{j'-1}$. Note that if $i' + 1 = j$ this slab list is actually empty. However, this makes no difference to the applicability of lemmas. From Lemma 3 we know that these trees combined contain all the possible candidates for the median we are after. Therefore deploying Lemma 3 on the three augmented binary search trees with rank $\lceil n/2 \rceil$ would yield the median of range $[i,\ j]$.

Let us analyse the query time of this data structure. The height of the three search trees is $O(\log (n/b))$. The search for a level where $i' \neq j'$ takes at most $O(\log_b n) = O(\log n)$ time. Hence, the overall query time becomes $O(\log n)$

The space complexity of this solution is a bit more involved. On an arbitrary level the persistent trees stored for each block contribute $O((n/b) \log n)$ words to the memory consumption. The slab lists contribute $O(n/b)$ for each pair of blocks.

Summed over all blocks and pairs of blocks, respectively, the total amount of memory used by this level becomes

$$O(b^2 n/b + b(n/b)\log n) = O(n(b + \log n))$$

The overall space consumption is therefore described by the following recursion:

$$T(n) = O(n(b + \log n)) + bT(n/b) = O(n(b + \log n)\log_b n)$$

For instance, if b is chosen to be $\log n$, the space usage becomes $O(n \log n \log n / \log\log n)$ which is equal to $O(n \log n^2 / \log\log n)$. This is in fact asymptotic optimal choice of b.

Krizanc, Morin and Smid did not analyze the preprocessing time of their data structure, or at least it was not included in the final article. However, it is at least nb^2 because of the $\binom{b}{2}$ slabs that are handled.

Range Tree Based Data Structure

Before describing this approach I will state another technical lemma, **[?]** Lemma 3. The proof is rather involved and can be found in the original article.

Lemma 4 *Let $A_1, \ldots, A_k$ be sorted arrays of size $O(n)$. There exists an $O(k \log n)$ algorithm which finds the element of rank i in $A_1 \cup A_2 \cup \cdots \cup A_k$*

The construction is a complete b - ary range tree build on top of the input array **A**. Thus, elements are stored in the leafs. An internal node stores a sorted list of the elements contained in its subtree.

Consider a range median query $\mathbf{Q}(i, j)$. The data structure starts by tracing two leaf-to-root paths. Consider the path from the root to the leaf containing the element of index i . The children immediately to the right of this path all store sorted lists of elements from the range $[i,\ n]$. In fact, these form a complete partitioning, albeit permuted, of this range. An analogous observation is true for the path from the root to the leaf containing the j'th element in **A**. However, in this case we choose to consider the children to the left of the path. The element lists of these completely partition $[1,\ j]$.

The root-to-leaf paths will inevitable have a common subpath, perhaps only consisting of the root. Denote the node where the paths split v. Now from node v the data structure collects the lists contained in children to the right of the path from v to the i 'th leaf and elements contained in children to the left of the path from v to the j 'th leaf. These lists completely partition the range $[i,\ j]$. There are at most $b \log_b n$ of such lists. The height of the range tree is $O(\log_b n)$ and each element stored by exactly one ancestor on each level of the tree. Hence, the overall space consumption of the range tree is $n \log_b n$. The algorithm uses this tree to search for a specific index in **A**. Now using Lemma 4 it is possible to calculate the median of the node lists in time $O(b \log_b n \log n) = O(b\frac{\log n^2}{\log b})$. The space of the data structure is dominated by

the range tree, which as mentioned earlier uses $O(n \log_b n)$ words of space.
Note that, by choosing b correctly we can assure that there for any $\epsilon > 0$ exists a linear space variant of the above data structure that solves a range median query in time n^ϵ.

Constant Time Solution

In addition to the trade offs given above, the authors propose a data structure which ensures constant query time, and uses slightly less than quadratic space. I will not present this structure since it is very complicated. However, the space complexity of the solution is $O(n^2 \log\log n / \log n)$.

Petersen

Holger Petersen has made two contributions to this problem. Both have presented solutions to the constant query time version of the range median and range mode problems. The second paper was coauthored with Grabowski.

$O(n^2 \log^{(k)} n / \log n)$ Space Solution

In the first article Petersen presents a layered data structure for the constant time query version of the problem, [Pet08]. The data structure is basically a clever way to store all answers by storing them succinctly in tables. The actual way of storing pointers in this structure is a bit complicated. For the technical details I refer to the article.

In the following I will sketch the data structure, which has space complexity $O(n^2 \log^k n / \log n)$, where $k \geq 1$ is the number of layers. The notation $\log^{(k)} n$ refers to the iterated logarithm. Continuing the layering would entail a solution using $O(n^2 \log^* n / \log n)$ space which is better, but with the downside of increasing query time to $O(\log^{(*)} n)$.

Let $k \geq 2$. For $k = 1$ the naive method of storing the median for all pairs of elements solves problem within the space bound stated above.

The data structure consists of k layers. There are three different cases of preprocessing depending on which level is being processed. The level 1 structure is fairly similar to that of [KMS05]. The input array **A** is divided into blocks of size $b = \log n$. For each pair of blocks the structure store pointers to the $4b$ elements that are possible results for queries with starting and ending point in these two blocks, respectively. That is b elements from each block and the $2b$ middle elements in the multislab defined by these two blocks.

For level $2 \leq \ell \leq k - 1$ the data structure basically repeats the procedure for level 1. The size of a block on level ℓ is $b_\ell = \log^{(\ell)} n$. For any pair of blocks the data structure stores pointers to the $4b_\ell$ elements that are potential results for queries beginning and ending in these blocks. Notice that the candidates for a pair of blocks

are a subset of the ones found for the two blocks of level $\ell - 1$ containing them. Therefore, pointers are stored relative to the pair of level $\ell - 1$ parent blocks themselves storing pointers. There are $4 \log^{(\ell-1)} n$ pointers for each pair of blocks in level $\ell - 1$, and therefore $\log^{(\ell)} n$ size pointers is sufficient on level ℓ. Finally at level k a pointer of size $\log^{(k)} n$ is stored for all pairs of indexes. For a pair of indexes (i, j), the pointer identifies the element in the level $k - 1$ structure representing the median of range $[i, j]$ relative to the data stored for the pair of blocks containing elements i and j on level $k - 1$.

Consider a query $\mathbf{Q}(i, j)$. Put simply the data structure follows pointers originating from the element stored for (i, j) on level k. When the search hits level 1 a pointer to an actual element of $\mathbf{A}$ is found and this element is returned. Unfortunately, it is not that simple to perform a query, due to the succinct nature of the data structure. A lot of machinery has to be in place in order to squeeze the space usage down to the wanted. Consequently, retrieving the pointers require some bit fiddling and will not be shown here. Instead see ([Pet08], Theorem 3).

I will now analyze the space usage of this data structure. To sum up the space requirement of level 1 is $O((n/b)^2 \log n^2)$ bits which is $O(n^2/\log n)$ words assuming a $O(\log n)$ word size. The overall space usage of level ℓ for $2 \leq \ell \leq k - 1$ is $O((n/\log^{(\ell)} n)^2 \log^{(\ell)} n^2)$. Since there are $k - 2$ of these levels The level k data structure requires $O(n^2 \log^{(k)} n)$ space in bits – or $O(n^2 \log^{(k)} n/\log n)$ space in words. Summed over all levels this gives:

$$O(n^2/\log n + (k-2)n^2 \log n + n^2 \log^{(k)} n/\log n) = O(n^2 \log^{(k)} n/\log n)$$

Notice that this is measured in words.

$O(n^2 \log\log^2 n/\log^2 n)$ space solution

This solution was published in collaboration with Grabowski, [PG09]. Again I will skip some details for brevity. The new data structure is essentially a way to fit the data structure from the former section into two levels instead of k.

The input array $\mathbf{A}$ is split into super blocks of $b = \log^2 n$ elements. For each pair of super blocks B_i and B_j a pointer to the $4b$ elements that are possible medians of ranges starting and ending in these super blocks are stored in a sorted table. The space required to store these elements for all pairs of super blocks is $O((n/b)^2 b) = O(n^2/\log n)$ bits which is equal to $O(n^2/\log^2 n)$ words if stored compactly. Note that the address space of the table for any pair of super blocks is $O(\log\log n)$.

Now at level 2 super blocks are split into blocks of length $\Theta(\log n/\log\log n)$. The key point in achieving the improved space complexity of this data structure is handling candidate medians within super blocks differently from those contained in slabs between super blocks. Note, that the space bound permits that answers for ranges either contained within a single super block or that has endpoints in neighboring blocks can be stored in a lookup table. There are $O(n^2/\log n^2)$ such ranges,

and for each the address of the median can be stored in a single word.

Hence, we only consider ranges with a nonempty middle super block slab from now on. Now for each pair of level 2 blocks not in either the same or neighboring super blocks the data structure stores a list of $O(\log n / \log\log n)$ pointers. These point to the addresses of median candidates of the middle super block slab between the super blocks containing these level 2 blocks that was gathered at level 1. There are exactly $O(n^2 \log\log^2 n / \log^2 n)$ such pairs of level 2 blocks.

Now the authors form a table $c\,(i, h, j, k)$ that for super block i containing level 2 block h, and super block j containing level 2 block k stores the addresses of elements that are possible medians for the range $[(i-1)s + hb,\ (j-1)s + (k-1)b]$. Since there are $\Theta(\log n / \log\log n)$ such addresses of length $O(\log\log n)$ the list of addresses will fit into one word. Additionally addresses to the original elements within a block are stored unsorted in a single word. This is possible since each address has length $O(\log\log n)$ and there are exactly $O(\log n \log\log n)$ such addresses to be stored.

A query $\mathbf{Q}(i, j)$ can now be answered by finding the super blocks and level 2 blocks containing the elements of index i and j , respectively. If the super blocks are equal or neighbors we simply lookup the result in the precomputed table. In the other case the data structure masks out the addresses of the prefix ending immediately before i of the address word of the level 2 block containing it. Analogously, for the block containing j the suffix just after j is masked out. Concatenating these words with the entry in c that is defined by the four computed values produces a bit string of constant length in words. Addresses for all possible median candidates are stored in this section. The result of the range median query is then extracted from this string using a lookup table.

Har-Peled and Muthukrishnan

Har-Peled and Muthukrishnan propose a solution to the offline static version in the article, [HPM08]. Their definition of the static offline range median problem is as follows

Definition 5 *Given n unsorted elements in an input array* $\boldsymbol{A}$ *and k intervals (ranges), determine the median in each of these k possibly overlapping, intervals and return these as a batch.*

A data structure that solves the offline problem using $O(n \log n)$ space and a collected query time of $O(n \log k + k \log k \log n)$ both worst case was then presented. If the intervals are given in an online fashion the query bound becomes amortized. The data structure is very hard to describe. Hence, I will leave out some technical lemmas and details.

The authors present the following lemma. The proof can be found in Appendix A of [HPM08].

Lemma 6 *Given ℓ sorted arrays of length at most n there exists an algorithm returning the median of the union of the arrays in time* $O(\ell \log n/\ell)$.

Notice the similarity to Lemma 4 that was stated in a previous Section.

Essentially, the data structure is an interval tree where each node keeps a sorted array representing the elements stored in the leafs contained in its subtree.

Denote the query intervals delivered to the data structure $I_1, I_2, \ldots, I_k$. The input array is split into $2k-1$ atomic intervals such that no endpoint of $I_1, I_2, \ldots, I_k$ is contained within any of the atomic intervals. Such a division can be obtained by splitting the input array on the interval endpoints in a sweep line fashion. These atomic intervals are sorted and inserted into a balanced binary search tree. Notice since the intervals are assumed known at the time of preprocessing we can sort the intervals without causing trouble for future queries. The height of the resulting binary tree is $O(\log k)$.

Now the median of an arbitrary query interval I_d can be found by searching for the endpoints of I_d and collecting the atomic intervals between the two resulting leafs. Using the algorithm from Lemma 6 on these atomic intervals median of I_d is obtained. Since the time bound of the batched version included the preprocessing time also the sorting of the intervals stored at internal nodes count. Since the overall cost of sorting is $O(n \log n)$ the promised time bound is exceeded by this approach for $k = o(n)$.

The actual data structure is using the overall structure as described above but the sorting of the input intervals is handled more cleverly. The authors introduce a concept they call u-sorting. An array $\mathbf{A}$ is u-sorted if there exists a sub array $\mathbf{A}_\ell$ of some size, depending on u by some constant, subject to the following constraints:

- For any element a in $\mathbf{A}_\ell$ it is true that if $a' \in \mathbf{A}$, $a' \leq a$ exists a' is placed after a in $\mathbf{A}$. That is a is placed on the index corresponding to its rank in $\mathbf{A}$.

- Any two elements of $\mathbf{A}_\ell$ are placed at most n/u elements apart in $\mathbf{A}$.

It is easy to u-sort an arbitrary array. A way to do it in the optimal number of comparisons is simply to pick the median and scan through the array dividing it into an lower and an upper half. Then place the median in the middle and call this procedure recursively on the lower and upper half. Repeated for u levels this produces a u-sorted array. Actually, if continued for n levels, this is exactly the same method used for the "partition" sub procedure of the popular quick sort. The number of comparisons used for the above procedure is $O(n \log u)$.

In the orginal article the authors demonstrate how to merge two u-sorted arrays into a new u-sorted array in linear time. The machinery that makes the fast algorithm possible is an extension of, [HPM08] Theorem 4.

Lemma 7 *Given ℓ u-sorted arrays $A_1, A_2, \ldots, A_l$ with total size n and an integer $1 \leq t \leq n$, there is an algorithm that returns ℓ sub arrays $B_1, B_2, \ldots, B_\ell$ and a number t' such that:*

- *The t''th ranked element in $B_1 \cup B_2 \cup \ldots \cup B_l$ is the t'th ranked element in $A_1 \cup A_2 \cup \ldots \cup A_\ell$.*
- *The running time of the algorithm is $O(\ell \log n/\ell)$*
- $\sum_{i=1}^{\ell} |B_i| = O(\ell \cdot (n/u))$

The data structure basically uses the same procedure as the slow algorithm I initially described but with the modification of applying u-sorting instead of complete sorting to the atomic intervals. Particularly, the atomic intervals are u-sorted in the leafs of the tree and are combined in internal notes. All this preprocessing takes $O(n \log u)$ time. Calculating the median of one of the query intervals is done by running the algorithm of Lemma 7 on the $m = \log k$ disjoint subarrays obtained by searching for the interval endpoints and collecting the lists of internal nodes covering the entire range. As the integer t we use r. Since the accumulated size of the resulting arrays is at most $O(m(n/u))$ selecting the element of rank t' from the union of resulting arrays is done by merging the arrays and using linear selection.

The number of comparisons required for solving all k queries is: $O(kmn/u + km \log n)$. If u is chosen to be k^2 this reduces to: $O(n + k \log k \log n)$. Since the preprocessing time was $O(n \log u) = O(n \log k)$ the overall number of comparisons required for solving a batch of k range median queries becomes:$O(n \log k + k \log k \log n)$.

Notice that this data structure only works when the queries are known. The authors demonstrate how to convert it into a data structure solving the problem for known k and even unknown k. In doing this, however, the time bound becomes amortized. Even so this was still the best tradeoff known for the static case at the time.

Chapter 3

Efficient Solutions to the Range Median Problem

In this section I will describe the data structures presented by Brodahl, Gfeller, Grønlund and Sanders in [BGJS10]. The article presents the best known time/space trade offs and are at some points approaching optimality in terms of upper bounds. Furthermore the authors are, to my knowledge, the first to consider the dynamic version of this problem.

Section 3.1 will contain three different solutions to the static range media problem. As I progress through the three solutions the time/space product gradually improves to the point where the query time is sublogarithmic and the space usage is linear.

Section 3.2 defines the dynamic range median and range selection problems. Then it describes a solution that solves the ladder, and the first by inclusion. This solution is basically a clever customization of one of the static solutions. It is surprisingly fast, however leaving somewhat of a gap to the current lower bound. The final Section 3.3 in this chapter is dedicated to a constant time solution solving the range median problem exclusively. The article proposes a very neat little solution that answer a range median query in constant time on the average using $O(n^{3/2})$ words of space in the worst case. Not only is it more space efficient, but it is also way simpler than any costant time solution presented in chapter 2. Contrary to the other solutions we have seen all structures in this chapter, except the constant time variant, works for general ranks in the range $[1,\ j - i + 1]$. Hence, it would really be more proper to call them range selection data structures than range median data structures.

3.1 Static RMP

I will start by describing the authors' solution to the static, both online and offline, version of the range median problem. The article gives three different solutions to

the static problem.

1. A $O(k \log n + n \log k)$ time algorithm using $O(n \log n)$ space in the RAM or pointer machine mode. This algorithm is optimal in the pointer machine model.
2. A space efficient variant of the first algorithm using linear space and obtaining the same running time in the RAM model however.
3. A very complicated, time efficient and compressed data structure achieving $O(\log n / \log \log n)$ query time with linear space.

In the next three sections I will describe these three data structures in detail. The main idea of the constructions is to translate a range selection query into a series of range counting queries. Later in this thesis, I will implement and experiment with variants of the first two structures.

$O(n \log n + n \log k)$ Pointer Machine Data Structure

The first data structure for the static range median problem that I will describe in this section supports k range median queries in $O(k \log n + n \log k)$ time using $O(k \log n)$ space. The k queries can be given in an online fashion or as a batch. We only demand k to be known as a way of analyzing the running time. It is neither the fastest nor the most space efficient algorithm presented in the original paper, but it is justified by the fact that it solves the problem optimally, for $k = O(n)$, in the Pointer Machine model.

The data structure is much simpler than the ones of Krizanc et al, Petersen and Har-Peled and Muthukrishnan. Even more importantly it achieves its running time and space bound without using any knowledge about the queries beforehand. Furthermore, a simple $O(n \log n)$ complete preprocessing phase allows for an arbitrary query to be solved in time $O(\log n)$. From a bird's perspective the data structure is a balanced binary tree, with adjacent structures in the form of arrays or binary search trees. The crucial part in achieving the query time bound is that preprocessing of one branch is delayed until a query demands it. I will now proceed to discussing the internals of the tree.

The nodes of the base tree contains an array of elements and either two or no children. The children will be denoted lower and upper, respectively. During initialization of the data structure the root of this tree is created. The array of elements it contains is a possibly slightly altered copy of the array[1] given as input to the data structure. The children of the node are constructed as queries demand them. I will now describe the approach taken by a single query. This is done by the means of

[1] I will throughout this paper state that elements are given as arrays contrary to, for instance, linked lists. This is, for the most, true but in some cases where the specific model of computation do not allow random access one should substitute every reference to arrays by the concept of linked lists.

a recursive function. In reality it traces out a path in the tree, creating child nodes when needed. I will be a bit cavalier about the description though, not distinguishing between the input array of the node and the actual node. Hence, when i refer to the **lower** part of an array it is actually assumed that I proceed to the lower child of the node having storing this array.

Recall the "partition" procedure of quick sort that we briefly visited in the discussion of the Har-Peled and Muthukrishnan construction. The procedure extracts the median from an array of elements in linear time and splits the array into an upper and lower half. We assume the median is to be appended to the lower half since it is defined to be the element of rank $\lfloor \frac{n}{2} \rfloor$[2]. The median considered here is the actual median of the input array – not the range median we have discussed until now. Notice, utilizing a simple scanning procedure ensures that the upper and lower half are still sorted with respect to their index in the original array. Let the **A** be the input array. Each element is augmented by its index in **A**. Both as a way to break ties, but also because it is crucial to the range counting taking place later. For the rest of this section the name element refers to the corresponding value/index pair. We will denote the lower array resulting from partitioning the array by the median **A.lower**, and let **A.upper** be the corresponding upper half.

As mentioned earlier this data structure solves the range selection problem which, in turn, also means the range median problem. For the remainder of this section the term query will refer to a range selection query $\mathbf{Q}([i,\ j],\ r)$ where $[i,\ j]$ is the query range and $1 \leq r \leq j - i + 1$ is the rank of the element wanted.

We will now take a closer look at how the query process works. Consider a query $\mathbf{Q}([i,\ j],\ r)$. The range $[i,\ j]$ is of course completely contained in **A**. The "partition"-procedure distributes some of the elements from $[i,\ j]$ to **A.lower** and some to **A.upper**. Assume now that the element of r'th rank in the range $[i,\ j]$ was distributed to **A.lower**. This means that the number of elements originally in the $[j,\ j]$ contained in **A.lower** is at least r. Since we did not sort the elements in **A.lower** by value we still can not return the correct element. However, we can discard the elements in **A.upper** from the rest of the search. Realizing this we let the data structure proceed recursively on **A.lower** with query $\mathbf{Q}([i,\ j],\ r)$.

In the other case, where the element of rank r in the range is contained in **A.upper**, less than r elements of the range is contained in **A.lower**. Assume that m_l elements of the range $[i,\ j]$ is contained in A.lower. This means that $m_u = j - i - m_l + 1$ elements was distributed to A.upper. Since we have discarded the m_l smallest element of the query range, the wanted element in A.upper now has rank $r - m_l$ in A.upper and the data structures proceeds recursively on A.upper with query $\mathbf{Q}([i,\ j],\ r - m_l)$.

[2]We can assume that all elements are distinct. If not, we could just postfix these with their index in **A**.

The remaining problem now is how to count the elements of $[i,\ j]$ in **A.lower**. Searching for the index of j in A.lower returns the number of elements that have index smaller than or equal to j. By subtracting the count obtained from a similar search for search for $i-1$, we get the actual count of elements in **A.lower** that originated from $[i,\ j]$ within **A**. Recall that the elements of **A.lower** are sorted by their original position in **A**. Thus, a binary search for an arbitrary index between 1 and n returning the position of the predecessor of the element with that index-parameter in **A.lower** can be performed in $O(\log n)$ worst case time.

Algorithm 1 Range selection algorithm with binary search.

```
1:  procedure RangeSelectionQuery(A, i , j , r )
2:      if |A| = 1 then return A[0]
3:      end if
4:      if A.lower = ∅ then                ▷ Split the array into lower and upper array
5:          median ← linearSelection(A, ⌊n/2⌋)
6:          for all elm ∈ A do
7:              if elm ≤ median then
8:                  A.lower.append(elm)
9:              else
10:                 A.upper.append(elm)
11:             end if
12:         end for
13:     end if
14:     m_l ← A.lower.search(j) − A.lower.search(i − 1)   ▷ Search with respect to
    index field. For i ≤ 0 return 0.
15:     if m_l ≥ r then
16:         return RangeSelectionQuery(A.lower, i, j, r)
17:     else
18:         return RangeSelectionQuery(A.upper, i, j, r − m_l)
19:     end if
20: end procedure
```

The query procedure is depicted in pseudo code above. We still have a problem however. The recursion tree has height $\log n$, and on each level we perform two binary searches. This means that we use $\sum_{i=0}^{\log n} \log 2^i$ comparisons on range counting which sums to $O(\log^2 n)$. Since we perform k such queries, the compound cost becomes $O(k \log^2 n)$.

Looking for an alternative way of doing range counting we notice that the series of binary searches performed for a query are all performed on the same values, $i-1$ and j. Since the input array **A** never changes neither do the results of the binary searches. Furthermore, the result of a range count is the same whether we query for the actual range parameters, or their predecessors in the current array of the recur-

sion. This suggests that one of the log-factors can be removed by the technique of fractional cascading – and that is indeed the case. Actually, a quite simple version of the technique will suffice. For an arbitrary element in **A** we must be able to find the largest element smaller than it with respect to index, its predecessor, in both **A.lower** and **A.upper**. Now, what was formerly a series of binary searches is performed by doing a single binary search for $i - 1$ and j, or simply indexing the array, on the top level and following the fractional cascading pointers.

Setting the fractional cascading pointers is quite simple. For an arbitrary element e the fractional cascading pointers are set in the following way. After processing e on line 6 in the pseudo code above, the largest element in **A.lower** smaller than e with respect to the index field, is the last element currently in **A.lower**. The pointer to **A.upper** is set analogously. This means that exactly one of the two pointers will point to e on the level below, see figure 3.1. If **A.lower** and **A.upper** are arrays, which is probably true if we are not in the Pointer Machine model, two simple integers representing the indexes of e's predecessors on the level below will work as fractional cascading pointers. If **A.lower** and **A.upper** are for instance binary search trees we need actual pointers.

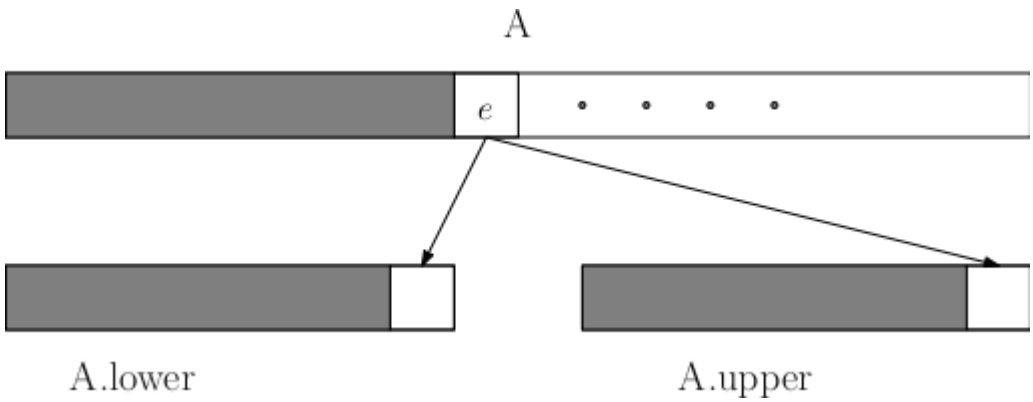

Figure 3.1: *Setting fractional cascading pointers for an element* e

To actually perform the range counting in the presence of fractional cascading pointers is simple if we are using indexes as pointers as described above. It is done by a simple subtraction of the two pointers pointing into **A.lower**. If on the other hand **A.lower** and **A.upper** are stored as binary search trees we have to augment the search tree nodes with its rank in a preorder traversal of the tree. Notice that, for the RAM version of the data structure, we can actually omit the index field of the element, since it is no longer needed for the binary search. This property is only true if we have actual random access though.

Analysis

Introducing fractional cascading pointers helped salvage the query part of the asymptotic running time, since each query now uses $O(\log n)$ operations for a single binary

search and $O(\log n)$ operations following fractional cascading pointers.

It still remains to bound the preprocessing part that takes place while answering the k queries. We will call a node empty if its associated lower child has not been created yet. The worst case number of operations used for preprocessing is attained when queries meet an empty node on the lowest level possible counting from the root. Assuming queries are picked by an adversary so as to maximize the cost of preprocessing a level d node can be the lowest empty node for $O(2^d)$ queries. At that time all its cousins on the same level must be filled. If we assume visiting a preprocessed node is free, the running time of a query meeting the first empty node at level d is $O(n/2^d)$ which is linear in the size of the associated array of this node. This follows from the fact that the data structure uses a linear selection algorithm on line 5 and a simple scanning algorithm to split the array. Furthermore, the running time of preprocessing levels $d+1$, $d+2, \ldots, \log n$ is dominated by the time it takes to preprocess level d. Assuming the adversary wishes to meet an unfilled node as early as possible The lowest level of the tree that he can delay filling completely is level $\lfloor \log k \rfloor$. Note that level $\lceil \log k \rceil$ could be left partially filled to cover the remainder. Filling a level completely costs $O(n)$ operations since every element is processed exactly once. Combined the worst case number of operations performed while preprocessing becomes $O(n \log k)$. The overall running time of answering k queries thus becomes $O(n \log k + k \log n)$ as wanted. The space bound follows through a similar adversary argument since new space is only allocated when a query needs it.

This data structure works both for known and unknown k since it makes no assumptions about, and no choices based upon, the queries it receives. The bounds hold both for the online and the offline case.
For $k = \omega(n/\log n)$ the data structure is an improvement over that of Har-Peled and Muthukrishnan. Note that complete preprocessing, by simply splitting the nodes recursively independent of queries, yields a data structure capable of answering queries in time $O(\log n)$ using $n \log n$ space.

Getting Linear Space

The above data structure had the advantage that it was very simple, and that it was close to optimal for the very restrictive Pointer Machine model. The downside however was that it needed quite a bit of space when k became large.

In this section I will describe a data structure that retains the fast query time but reduces the space usage to linear in terms of machine words. Hence, this data structure is only working in the well known RAM model and its variants. We assume that the word size w is such that we can address elements within A using a constant number of words. That is, $w = O(\log n)$.

The new improved data structure is actually only a slightly rebuild version of the old. The main insight is that the associated arrays and fractional cascading pointers

can be replaced by a space efficient binary rank data structure, in a way that only uses n bits, or n/w words, on each level. We did not use the associated arrays for anything else but range counting anyway. Having these structures we can omit the associated arrays in the internal nodes and store elements in the leafs to be accessed by a query. We can also omit the augmented index field entirely.

The layout of the data structure is essentially the same as the Pointer Machine Solution. Each node has two children, and an associated structure, **lowerbits** which is a binary rank/select structure, [OS07]. We will not use binary selection for anything here, since we only need to count the bits set in an arbitrary prefix of a binary string. The associated structure is build by a function **make_rank**(b) where b is a binary string. The associated structure can be queried by a function **rank**(t) that returns the number of 1-bits in the t'th prefix of b. I will describe one possible implementation of such a space efficient rank structure that uses $n + o(n)$ space, where n is the length of b, and answers the rank query in constant time later.

Remember that the only information we needed to branch the search in the fractional cascading algorithm was the number of elements in the query range present in the lower array. A simple way to store the required information would be to store n bits for each associated structure where a bit t is set if the element with index t in **A** is present in the lower array of the split. Unfortunately, such a strategy uses n bits for each internal node and would therefore use too much space. Instead the data structure takes a different course, still letting the binary rank count the number of elements in **A.lower** smaller than a given index but doing so in a dense manner where we are sure that at least half the bits of the binary range structure are set. The way to do this is to construct a binary string as long as the input array of this node, and checking bit t in this string if the t'th element in the input array was distributed to the lower half during partitioning. The d'th level of the tree now contain 2^d binary strings of combined length n. If stored compactly this uses $O(n \log n)$ when summed over all levels.

Doing the range counting this way, however, makes queries a bit more intricate to perform. The reason is that the query range has to be transformed in order to fit the contracted bit array. Consider a range selection query $\mathbf{Q}([i,\ j],\ r)$ at an arbitrary node N of the tree. Let $rankl = N.\mathbf{lowerbits.rank}(i-1)$ and $rankr = N.\mathbf{lowerbits.rank}(j)$ be the number of elements appended to **N.lower** with index less than or equal to $i-1$ and j, respectively. If $r \leq rankr - rankl$ we know that the element of rank r is located in the lower subtree. For the next level of the recursion the query rank remains to be r but the query range changes. The new query range becomes $rankl + 1$ and $rankr$. By definition this range contain all elements in the range $[i,\ j]$ of the associated array of **N**.

If $r > rankr - rankl$ the element of the required rank must be found in **N.upper**. In this case we have to change both the query range and the query rank. The new query rank becomes $r_n = r - (rankr - rankl)$ as it did in the Pointer Machine data structure. How do we translate the query range in this case? We discard exactly

rankl elements smaller than i by proceeding with **N.lower**. Hence, the left endpoint should be shifted *rankl* towards the beginning of **N.upper**. The same is true for j and *rankr*. Hence, the new query range becomes $\mathbf{Q}([i - rankl,\ j - rankr],\ r - (rankr - rankl))$. Pseudocode describing the query and preprocessing procedure is depicted in Algorithm 2.

Algorithm 2 Linear space RAM data structure with logarithmic query time

```
1: procedure RangeSelectionQuery(A, i , j , r )
2:     if |A| = 1 then return A[0]
3:     end if
4:     if A.lower = ∅ then                ▷ Split the array into lower and upper array
5:         median ← linearSelection(A, ⌊n/2⌋)
6:         idx ← 0
7:         b ← zeroes(|A|)                ▷ A binary string containing of all zeroes
8:         for all elm ∈ A.arr do
9:             if elm ≤ median then
10:                b.set(idx)
11:                A.lower.append(elm)
12:            else
13:                A.upper.append(elm)
14:            end if
15:            idx ← idx + 1
16:        end for
17:        A.lowbits ← make_rank(b)
18:        deallocate A        ▷ Remove the value array associated to this node since
   there is now no need for it.
19:    end if
20:    rankl ← A.lowbits.rank(i − 1)
21:    rankr ← A.lowbits.rank(j)
22:    m_l ← rankr − rankl
23:    if m_l ≥ r then
24:        return RangeSelectionQuery(A.lower, rankl + 1, rankr, r)
25:    else
26:        return RangeSelectionQuery(A.upper, i − rankl, j − rankr, r − m_l)
27:    end if
28: end procedure
```

Notice that the input array of a node is removed since it is no longer needed. Elements are stored in leafs though making it possible to answer queries. Recall from the previous section the input arrays of level d of the tree contain n elements combined. Since these input arrays have been replaced by space efficient binary trees using a number of bits linear in the number of elements, we use n/w words on each level. Since there are $O(\log n)$ levels in the tree the data structure uses linear space when all levels are filled completely. The logic of answering queries are the

same, though now the fractional cascading pointers have been replaced by rank counting queries that can be achieved in constant time for each level. Therefore the running time of a query is not affected by the change.

Even the data structure we just described can be improved on. In the next Section I will describe a RAM data structure solving a range selection query achieving $O(\log n / \log\log n)$ query time still using linear space. First, however I will describe a Binary Rank/Selection Data Structure supporting the prefix rank counting in constant time using only a linear number of bits.

Verbatim – A Binary Rank/Selection Data Structure

I will now, as promised, describe a binary rank/selection data structure. The data structure takes a binary string containing n bits, and supports the following operations:

- **rank**(i) – Counts the bits set in the i'th prefix of the binary string.
- **select**(i) – Returns the index of the ith bit set in b.

The structure Verbatim, [OS07], is a binary rank/selection structure supporting the **rank**-operation in constant time, while using only $n + o(n)$ space in bits. Since we do not really need the **select**-operation, I will only describe the implementation of the **rank**-operation for the Verbatim data structure.

Recall that we defined $w = \lceil \log n \rceil$ to be the word size of the RAM. The preprocessing phase of this data structures the binary string into superblocks of size w^2, which are in turn divided into blocks of size w.

A table tsb of size n/w^2 holds the prefix sum bits set in super blocks. I.e. $tsb[1] = 0$ and $tsb[i] = tsb[i-1] +$ "# of bits set in previous super block" for any $i \in [2, \lfloor n/w^2 \rfloor]$. This is essentially the same as storing the value of **rank** for each w^2 sized chunk of the binary string. Additionally, the data structure maintains a 2-dimensional table tb that stores the prefix sums of bits set within the at most w blocks contained of a super block. The prefix sums for the blocks stored are relative to the start of their parent super block. I.e. $tb[i][0] = 0$ and $tb[i][j] = tb[i][j-1] +$ "# of bits set in block j-1 within super block i" for $1 \leq j \leq w$, $1 \leq i \leq n/w^2$. This ensures that the prefix sums of tb can be stored using $\log\log n$ bits. Notice, that both tables are easily calculated by doing only one scan through the binary string. Pseudo code showing this scan is presented in algorithm 3.

The space usage, measured in bits, of the two tables become: $O(nw/w^2) = O(n/\log n)$ for tsb and $O(n \log\log n / w)$. In total, the two tables contribute with $o(n)$ bits to the space usage of this algorithm.

Besides storing the two tables Verbatim stores the original n-bit bitstring using in total $n + smallOhn$ space.

The **rank**-operation is implemented as a lookup in the two tables. In addition a constant **POPCNT**-operation most be supported. The **POPCNT**-operation does, given a binary string at most a constant number of words long, return the number

Algorithm 3 Preprocessing phase of the Verbatim binary rank/select data structure data structure

```
procedure MAKEVERBATIM(b)
    sblockidx ← 1
    sblockbitcnt ← 0
    tsb[0] ← 0
    while sblockidx ≤ ⌈n/w²⌉ do
        tb[sblockidx][0] ← 0
        nword ← Get the next w bits from b
        blockidx ← 1
        while blockidx ≤ ⌈n/w⌉ do
            blockbitcnt ← 0
            for all bit ∈ nword do
                if bit = 1 then
                    blockbitcnt ← blockbitcnt + 1
                end if
            end for
            sblockbitcnt ← sblockbitcnt + blockbitcnt
            tb[sblockbitidx][blockbitidx] ← tb[sblockbitidx][blockbitidx − 1] +
blockbitcnt
        end while
        tsb[sblockidx] ← tsb[sblockidx − 1] + sblockbitcnt
    end while
end procedure
```

of bits set in the string in constant time. If not implemented directly by the CPU, It can be tabulated using $O(\sqrt{n}\log^2 n)$ bits.

Given a rank query **rank**(i) the super block containing the i'th bit has index $sb_i = \lceil i/w^2 \rceil$. The block, within super block sb_i containing i can now be calculated by the modulo operator: $bl_i = \lceil i/w \rceil \mod w^2$. Having these two values we can lookup $tsb[sb_i]$ and $tb[sb_i][bl_i]$. This yields the rank of the string ending in the block just preceding the bl_i'th block. Calling **POPCNT** on the i'th prefix of the bl_i'th block gives the remaining part rank. The table lookups are clearly done in constant time. As is the call to **POPCNT**-operation as discussed above.

The Verbatim data structure is actually not the best binary rank/select data structure. Others has been produced which, by empirical studies have been proven faster. They all, however, bear a striking resemblance to Verbatim but split the string in different ways. An example of such a data structure can be found in [OS07].

Achieving Sublogarithmic Query Time and Linear Space

The data structures presented in the former two sections have been relatively simple. Their simplicity is mostly due to the "partitioning"-phase of quick sort, and

the fact that range counting is, in essence, a simple problem that can be solved efficiently. The downside of having these two components in a data structure is that no matter how one chooses the pivotal element in the "partition"-function the optimal height of the the root-to-leaf path followed while answering a query is $O(\log n)$. Since, the range median, and thereby the range selection problem is no easier than sorting and the lower bound for sorting in the RAM model is $\Omega(\log \log n)$ it leaves a somewhat significant gap to the trivial lower bound. Some of this gap, however, is bridged by the data structure that will be described next.

In this section I will describe a data structure that as of this moment, gives the best tradeoff between query time and space usage. The data structure permit queries to be answered in sublogarithmic time while maintaining the linear space bound in words of the former two static solutions. The cost, however, is that the solution much more complicated.

The main idea of this data structure is a different way to do range counting, using advance RAM-operations permitting it to be done in constant time, while increasing the branching factor of the tree to $\lceil \log^{\epsilon} n \rceil$ for some $0 < \epsilon < 1$. Since the tree has height $\log_{\log^{\epsilon} n} n = O(\log n / \log \log n)$ a sublogarithmic solution is achieved. Squeezing the space down to linear is, unfortunately, quite an ordeal complicating queries a bit.

Overall Structure

Contrary to the "partition"-based structures this one is built bottom-up. The input array **A** is sorted while still keeping track of an elements original position in **A**. A balanced search tree **T** with branching factor $\lceil \log^{\epsilon} n \rceil$ is built on top of the sorted array **A**.The leafs of **T** is simply the elements in **A**. The real magic of this structure happens in the internal nodes. For the rest of this section I will refer to the set of leafs contained in the subtree rooted at an internal node $v \in \mathbf{T}$ as $\mathbf{T_v}$. Each internal node v of **T** stores the following memory.

- $f = \log^{\epsilon} n$, pointers some possibly empty to the children of $\mathbf{T_v}$.
- An array, $\mathbf{A_v}$, consisting of $|\mathbf{T_v}|$ matrices, each containing f rows of bits. For a specific $y_i \in \mathbf{T_v}$ the the ℓ'th row of corresponding matrix M_i stores the number of elements of $A[1, i]$, where i is the original position of y_i in **A**, contained in the first ℓ subtrees of v. Note, that the layout of the array ensures that the position of M_i is equal to the number of elements from T_v contained in $A[1, i]$. Each row of a matrix is stored in its own word.
- A copy of the matrices above. The matrices are stored such that the $g = \lfloor \log n / f \rfloor = O(\log^{1-\epsilon} n)$ first columns of the f rows, for the raminder of this description denoted a section, are stored in a single word. The second section consists of the last 3 columns of the previous section followed by the first $g - 3$ columns not yet processed. This pattern continues until all columns have

been handled. We can think of each section as a $f \times g$ matrix.In order to handle come carry problems the first row of each section has a columns of zeroes prepended before being divided into sections. For each matrix we will number the sections from 1 to f/g, from the most significant end. Notice, that a portion of the sections starting from 1 could contain all zeroes.

The purpose of these fields, and especially how they each participate in answering queries will be discussed in the next Section.

For each internal node v and arbitrary $1 \leq i \leq |T_v|$ the rows of matrix M_i of A_v form a nondecreasing sequence of integers. This property is due to the fact that any leaf contained in one of the first $1 \leq j \leq f$ will be counted in row j, and all following rows of M_i. Furthermore, each row in the array $\mathbf{A_v}$, that is the integers stored in a fixed row of all matrices forms a nondecreasing sequence since they are in effect prefix counts. In the next Section I will describe how this construction permits us to answer a range selection query.

Handling queries

Consider a range selection query $\mathbf{Q}([i,\ j],\ r)$ where $1 \leq i,\ j \leq n$ and $r \leq j - i + 1$. The data structure, basically, just traces out the path to the leaf holding the element of r 'th rank in the range $[i,\ j]$. In each internal node the branching decision depends on a prefix range counting in the number of leafs contained in the subtrees of its children. The main difficulty is making the branching decision in an arbitrary internal node v in constant time. This is, of course, a direct consequence of increasing the branching factor of the tree.

Intuitively it seems simple. For any internal node v we have stored the prefix count of leafs contained in subtrees. In particular, we have two matrices M_j and M_{i-1}, corresponding to elements a_j and a_{i-1}, respectively. Since some elements are discarded as we descend down the tree a_j and a_{i-1} might not be present in $\mathbf{T_v}$. Therefore, while searching down the tree M_j is actually represented by $M_{\mathrm{d}(j)}$ where $\mathrm{d}(j)$ is the largest index such that $a_{\mathrm{d}(j)} \in \mathbf{T_v}$ and $\mathrm{d}(j) \leq j$. M_{i-1} is changed to $M_{\mathrm{d}(i-1)}$ in a similar way.

Consider a matrix M' defined by $M_{\mathrm{d}(j)} - M_{\mathrm{d}(i-1)}$. The ℓ'th row of M' holds the number of leafs in the first ℓ subtrees of v contained in $A[i, j]$. Consider the minimal row r_ℓ of M' where $r \leq r_\ell$. The element of rank r in $A[i, j]$ will be contained in the subtree rooted at the ℓ'th child of v, and the search will continue by that node. Unfortunately, since each row of the matrices was stored in a word of its own, just computing M' from the matrices in $\mathbf{A_v}$ would require $\mathrm{O}(f)$ RAM operations. The rest of this Section will be devoted to one thing. Explaining how we can make the branching decision described above without explicitly computing M'.

The main insight is that it is possible to obtain a range $[\ell_1,\ \ell_2] \in [1,\ f]$ of indexes from M' in constant time. The range specifies a list of rows from M' which have their

most significant sections in common with r . Since the remaining less significant sections are not included all rows in the list could in fact be smaller than r . Therefore we include the row immediately after the range. In summary, the children specified by the range $[\ell_1,\ \ell_2]$ are all candidates for the next node in the search, and none outside the range can be candidates. Consider choosing ℓ from $[\ell_1,\ \ell_2]$ such that ℓ is the first row larger than r . If $\ell_2 - \ell_1 \leq 2$ we can determine ℓ in constant time. For both endpoints compare the word $M^{\ell_i}_{\mathrm{d}(j)} - M^{\ell_i}_{\mathrm{d}(i-1)}$ with r . This operation only costs a single RAM-operation. If the range is larger we first test the endpoints ℓ_1 and ℓ_2. If either the ℓ_1'th row is smaller than r or both the ℓ_2'th and the row just before it are larger than r we have to search for ℓ using a binary search. Such a search uses $\mathrm{O}(\log\log^e n) = \mathrm{O}(\log\log n)$ operations. If we were forced to do a binary search on all levels the query time would become $\mathrm{O}(f \log\log n)$. Later we will show that no more than $\mathrm{O}(\log n/g)$ binary searches are performed.

Recall that we stored the matrices of A_v, for an arbitrary internal node v, in an alternative way consisting of a number of $f \times g$ matrices called sections. These sections are each stored in a single word and we can therefore subtract them using just one RAM operation.

The query process maintains a value c, initially 1, holding the number of the current section of the bit matrices in use. We maintain an invariant ensuring that the c'th section indeed contains the most significant bit of the matrix M'. That is, all sections to the left of the c'th section contain nothing but zeroes. Let r_c be the c'th section of r . Next it will be explained how to maintain c and r as the search progresses. Let v be an internal node encountered during the search. We'll let K denote the number of elements from $A[i,j]$ contained in $\mathbf{T_v}$. We will denote by S_i and S_j the c'th section of the matrices represented by i and j, respectively. All sections preceeding S_i and S_j in these matrices will contain all zeroes. By the invariant S_i and S_j contain the most significant bits of the two words. The goal is to obtain a word W_{ij} approximating the c'th section of M'. The obvious approach would be to subtract S_i from S_j. Unfortunately, we have to be a bit more clever. Depending on how the individual rows are distinguished within the section subtraction of lower order rows could cause a cascading query to inflict on the later rows. To remedy this every row in S_j is prepended by a 1 while every row in S_i is prepended by a zero. These actions are achieved through bitmasks. This will stop the carry from cascading through the word and ensure mononicity of W_{ij}. The first bit of each row of W_{ij} is then masked out before the process continues.

The word W_{ij} is still not entirely equal to the c'th section of M'. The reason is that a cascading query stemming from the least significant bits is not counted. The consequence is that a row in W_{ij} can become one larger than the corresponding row in the c'th section of M'.

As described earlier we calculate $[\ell_1,\ \ell_2]$ now using W_{ij}. In order to factor in the missing carry, we must include elements that match the first g bits in r added by one, as well as the first row immediately following these. It is clear that no rows

that would have been candidates are discarded by the approximation. The search for the first child containing at least r children is performed by first checking the interval endpoints ℓ_1 and ℓ_2. If none of these options represent the correct choice for the next node in the search, we have to resort to binary search among the elements in $[\ell_1 + 1,\ \ell_2 - 1]$. This correctly identifies the subtree in which the queried element resides. There are still, however, some issues about the handoff of responsible to the child that must be addressed.

The purpose of the active section index c is now explained. Since the $(\ell_1 + 1)$'th row of W_{ij} is at most 1 smaller than the "optimal" row in M', whereas the $(\ell_2 - 1)$'th row is at most 1 too large, the difference between them is at most 2.

Hence, after guiding the search to a child v', all rows in the c'th section of M', that is $M_{i-1} - M_{j-1}$, calculated for v' will represent a number between 0 and 2. This is the reason for the 3 bit overlap between sections, that seemed a bit misplaced when we described the structure. By the above discussion the first three bits of any row in the $(c + 1)$'th section of $M'_{v'}$ will start by a zero, which satisfies the invariant we maintain, and a number between 0 and 2 is encoded in the next two bits. All bits in section 1 through c will contain nothing but zeroes. The bits stored in section c will remain zero for the rest of the search. We can therefore safely increment c, shifting the active section towards the lesser significant bit positions of r and M'.

We need a way to determine the new prefix count matrices $M_{i'}$ and $M_{j'}$ that will form the basis of the approximation in the next level of the search. At the top level of the search $\mathbf{A_v}$ contains n matrices and M_{i-1} and M_{j-1} are simply found by indexing into this array. For the remaining levels we deploy fractional cascading. We set f pointers for each matrix. The query rank r needs to be translated to fit the size of $\mathbf{T'_v}$. We will translate it in the same way as we did in the $O(\log n)$-time data structures described in the previous Sections. The number of elements from $A[1, i-1]$ and $A[1, j]$ contained in $\mathbf{T_{v'}}$ is stored in the f'th rows $M^f_{i'}$ and $M^f_{j'}$ of the two new element matrices, and the new query rank therefore becomes $r - (M^f_{j'} - M^f_{i'})$.

I will now show how the range $[\ell_1,\ \ell_2]$ can be calculated in constant time. First we need to define some tables that are assumed created during the preprocessing phase:

- A table **rep** that for each possible g-bit string, s, stores a string consisting of s repeated f times, where each occurrence of the string is preprended a 0 bit. We can think of this as a single word.
- A table **lessbit** that given a word storing binary string returns the index of the least significant 1-bit.

Remember that each row of the matrix W_{ij} starts with a 0 bit by construction. Let r_p be the element stored for for the c'th section of r in table **rep**. By using a bitmask containing ones on every g'th position we exchange the 0 bit in front of every row

in W_{ij} by a 1 bit. The idea now is to subtract the word r_p from W_{ij}, hereby creating a new word W_{diff}. The 1 bits masked into W_{ij} serve two purposes:

1. They ensure that carries will not cascade from one row to another,
2. If row ℓ is less than r_p the inserted 1 will be converted to a 0-bit by the carry.

By the remarks above, the value of ℓ_1 can be read from the entry corresponding to W_{diff} in **lessbit**. The maximum element of the range, ℓ_2, can be found in a similar way.

Since c can be incremented at most $\lceil 1 + \log n/(g-3) \rceil = \mathrm{O}(f)$ times, no more than $\mathrm{O}(f)$ binary searches will ever be performed. Each of these costs $\mathrm{O}(\log f) = \mathrm{O}(\log\log n)$ time. In the cases where a binary search was not necessary, that is the next subtree is represented by either ℓ_1 or ℓ_2, the branching decisions is made in constant time.

Hence, the amount of time spent guiding the search down the tree amounts to:

$$\begin{aligned}\mathrm{O}(f \log\log n + \log n/\log\log n) &= \mathrm{O}(\log^{\epsilon} n \log\log n + \log n/\log\log n)\\ &= \mathrm{O}(\log n/\log\log n)\end{aligned}$$

The remaining operations performed in an internal node take constant time. We can therefore conclude that the data structure answers a range selection query in time $\mathrm{O}(\log n/\log\log n)$.

The space used by the data structure just described is, unfortunately, a bit above linear. Each element of **A** is represented on every level of the tree by a matrix taking up f words of space. Since there are $\log n/\log\log n$ levels in the tree the complete space usage of the data structure becomes

$$\mathrm{O}(n \log^{\epsilon}(n) \log(n)/\log\log n) = \mathrm{O}(n \log^{1+\epsilon}(n)/\log\log n)$$

In the next Section I will sketch how the layout of the matrices can be compressed enough to make the whole data structure fit within $\mathrm{O}(n)$ words.

Matrix compression

In this Section I will briefly sketch the way the data structure described above is compressed to fit in a linear number of words. Instead of getting bogged down with the details one can keep the mental image of the previous Section, since the new structure is basically just thais crammed into chunks of memory, and skip this Section.

The technique used is standard and a concept that is widely spread in both data structures and dynamic algorithms. The idea is to insert ”check points“ within the memory, and then compress the memory between them in such a way that any memory cell can be restored in constant time from one or more check points.

We'll define $t = \lceil f \log n \rceil$ and call it the chunk size of this data structure. For any internal node v the array $\mathbf{A_v}$ is split into chunks of size t such that only the last matrix is stored completely. For the remaining elements in $\mathbf{A}$ $\lceil \log f \rceil$ bits are stored, each describing the index of the subtree this element belongs to. The descriptions are stored compactly meaning $d = \lfloor \log n / \lceil f \rceil \rfloor$ of them are packed into one word called a direction word. After each direction word a prefix count meaning the same as in the previous Section, is stored. That is, the count of elements for each subtree stored wihthin the preceding chunk. Since there are t elements in a chunk storing the prefix sum can be accomplished using $f \lceil \log t \rceil / \log n = \mathrm{O}(1)$ words. On an arbitrary level of the tree the n elements are split into n/t chunks each using $\mathrm{O}(f + t/d + 1)$ space. Summed over all levels this gives a space usage of $\mathrm{O}(n)$ words.

The query procedure is somewhat more involved than in the first version of the data structure. The reason, of course, is that we do not necessarily have M_i and $M_s tR$ available. Recall that those where key for obtaining the approximation word W_{ij}. Given a query $\mathbf{Q}([i,\ j],\ r)$, and assuming the search is currently at an internal node v, we denote by r_i and r_j the number of elements from $\mathbf{T_v}$ contained in $A[1, L-1]$ and $A[1, R]$, respectively. We use the matrices stored for the $\lfloor r_i/t \rfloor$'th and $\lfloor r_j/t \rfloor$'th chunks as replacements for M_i and M_j. Denote these M_a and M_b. Since each chunk comprises t elements, any row in M_b is at most t smaller than the corresponding row in M_j. The same is true for M_a and M_i. The value of any row in $\bar{M} = M_b - M_a$ can therefore differ by at most t from its corresponding row in M'. But this means that at most the $\log t$ least significant bits can be wrong, essentially meaning only the first section is affected. The remaining, more significant sections is indirectly affected by a possibly missing carry though. The c'th section of $\bar{M}$ can therefore be both one smaller or one larger than the corresponding section of M'.

Using matrix $\bar{M}$ and the direction words and prefixes stored in chunks immediately after M_a and M_b, we can calculate a similar approximation. The main impact is that the overlap between sections in the is increased to 4 bits instead of 3.

Let $L' = \lfloor r_i/t \rfloor$ and similarly $R' = \lfloor r_j/t \rfloor$. Aside from creating the matrices the query process is similar to that of the previous data structure. Specifically, an active section counter c is kept and initialized to 1 before the query. An invariant is maintained saying that all more significant bits than those of the c'th sections are zero and the first bit of the c'th section will also be zero. Recall that the matrix M_b is associated with the R''th chunk.

I will now describe how to recreate a row of M_j using M_b and the prefix and direction words stored in the $(j'+1)$'th chunk. M_i is recovered in a similar way using M_a and the $(i'+1)$'th chunk. This recovered version is only used for the binary search search, or if c identifies the last section. In any other case the c'th section of M_b and M_j are the same except for the carry problem. The objective is to calculate for each ℓ between 1 and f the number of elements from $A[1, R]$ that is contained within the first ℓ children of the current node v. We already know this value for

$A[1, tR']$ which is stored in the ℓ'th row of M_b. The last portion is equal to the number of elements among the first $(r_j - t \cdot j')$ contained in the $(R'+1)$'th chunk. The direction word with index $p = \lfloor (r_j - R't)/d \rfloor$ stores how many of the first pd elements from the chunk are stored in the first ℓ subtrees. This only leaves the elements in the direction word containing r_j.

We start by masking out the last bits corresponding to the addresses directing elements larger than r_j.A table of size at most n is created that for each valid direction word, including those where a prefix or postfix has been masked out, stores the prefix count of the incidences for the f children. These prefix counts can each be stored in $\log t$ bits and all f therefore fit in 1 word.

A lookup in this table gives the prefix count for element l among the remaining $r_j - j't - pd$ elements that was not yet accounted for. We have therefore completely restored the ℓ'th row of M_j. If c currently points to the last section of r the matrices M_i and M_j are calculated completely, the first $c-1$ most significant sections will contain zeroes, using the above solution and the search is guided exactly as in the previous data structure. In the other case we calculate the word W_{ij} using the c'th section of the two matrices M_a and M_b. The range $[\ell_1,\ \ell_2]$ containing the candidates for containing the r 'th element in $\mathbf{A}[i, j]$ is calculated in the same way as described above.

Like earlier a cascading query from the lower bits of the rows in M_a and M_b could cause a row in the approximation matrix to become 1 too large compared to $\bar{M}$. Therefore when determining $[\ell_1,\ \ell_2]$ we need to include rows matching both r_c, the c'th section of r , $r_c - 1$, $r_c + 1$ and $r_c + 2$. Moreover, we must include the first row immediately after the maximum. The additional slack of 1 to each side comes from using $\bar{M}$ as the basis for the approximation and was accounted for above. Consequently, the value of the $(\ell_2 - 1)$'th row of the c'th of M' is at most $r_c + 3$, whereas the value of the $(\ell_1 + 1)$'th row of the same section at least becomes $r_c - 3$. Hence, the difference between rows $(\ell_2 - 1)$ and $(\ell_1 + 1)$ is 6. Consequently, to correctly distinguish each of these rows in the remaining subtree we need 3 bits. The extra bit is set to zero as above.

In summary, by introducing chunks and small changes to the query phase we succeeded in achieving linear space. As for the query time, both the fractional cascading between chunks and advancement of c active section is assumed. The rest of the modification is basically supported by table lookups which can be done in constant time. Since the branching factor of the tree remains the same so does the height. We can therefore conclude that in spite of the modification we retain a $O(\log n / \log \log n)$ query time as promised.

3.2 Dynamic Range Median Algorithm

In this section I will describe the dynamic data structure for the range selection problem described in [BGJS10]. This data structure is, literally, a dynamic version of the first $O(\log n / \log\log n)$ solution I described in the previous sections. Its main core is a B-tree. The internal nodes of this B-tree are storing matrices similar to the prefix matrices that was an integral part of the static solution.

For this dynamic setting we do not have an input array, or list, that we can refer to as we did in previous chapters. The definition of the dynamic range selection problem that I will use is the following: A set of points (x, y),of size n, that supports the following three functions:

- A query function $\mathbf{Q}([i,\ j],\ r)$ which returns the r 'th y-value among the set of points having x-values within the range $[i,\ j]$.
- An insert function $insert(x, y)$ that inserts the point (x, y) or changes the value of y for the x'th value if already present.
- And finally a $delete(x, y)$-function deleting the point (x, y) if present.

The data structure supports range selection queries in $O((\log / \log\log n)^2)$ worst case time. Insert – and delete operations can be done within the same time bound, albeit amortized. The space usage in words is $O(n \log n / \log\log n)$.

Structure

As mentioned earlier the core of structure is a weighted B-tree with $B = \log^{\epsilon} n$ ordered on the y-values of the points. Points are stored in leafs and an internal node has at most B children. Since the layout of the algorithm depends on n, B will change as updates are coming in. More specifically the data structure will be rebuild for every $\Theta(n)$ updates. In the process B is changed to reflect the new size of the point set.
Let $c(v)$ denote the number of children of a node within a B-tree. A weighted B-tree is a data structure designed for IO-efficient search structures and is specifically designed for data structures that needs associated structures. That is, the weighted B-tree minimizes the splits and fuses caused by updates to the main tree. In this data structure each internal node v keeps two associated structures.

- A *ranking tree* $\mathbf{R_v}$. This is a B-tree storing matrices, in some sense, of all elements contained in the subtree of v, $\mathbf{T_v}$. The leafs $\mathbf{R_v}$ each store B^2 elements. The maximum height of a ranking tree is $h = \log_B n$. We will have more to say about the structure of ranking trees in the next section.
- A linear space predecessor structure that stores all elements in $\mathbf{R_v}$ stored by x-value. This structure is used for accessing the leaf in $\mathbf{R_v}$ containing the predecessor of query index in $\mathbf{T_v}$.

For an arbitrary internal node v of the main tree the *ranking tree* $\mathbf{R_v}$ stores the elements of the corresponding $\mathbf{T_v}$ in the leafs. An internal node u of $\mathbf{R_v}$ contains an array of at most f prefix matrices. For each of the $\mathrm{c}(v) \leq f$ children of v a node u of $\mathbf{R_v}$ stores a single matrix. The p'th row of the ℓ'th matrix, $1 \leq \ell \leq \mathrm{c}(v)$ and $1 \leq p \leq \mathrm{c}(u)$ contains the number of leafs within the first p subtrees of u that is a copy of a leaf in one of the first ℓ subtrees of v. The rows of an arbitrary matrix are each stored in one word as with the case with the static structure. Furthermore, as we did in the static solution an alternative copy of each matrix is stored along with the original. This copy stores the rows in sections of size $g = \lfloor \log n / f \rfloor$ bits each consuming one word of memory. However, we need $\Theta(\log \log n)$ bits of overlap between the rows in successive sections this time. Why this is necessary will be explained later.

Additionally, each node of $\mathbf{R_v}$ stores the description of up to B^2 updates. First of all an update description contained within the update buffer of a node u of $\mathbf{R_v}$ states whether it was an insertion or a deletion. Moreover, both the subtree of v and the subtree of u, from which the update originated is stored as part of the update description.

Range Selection Query

I will now describe how to perform a range selection query on the structure described in the previous section.

A query $\mathbf{Q}([i,\ j],\ r)$ is naturally interpreted as finding the r'th y-value among the points which x-value falls within the interval $[i,\ j]$.

The main idea is to reconstruct the matrices M_i and M_j or at least an approximation of them. Th These will be used for branching between the at most B children of an internal node v of the main tree. The interpretation of these matrices are analogous to the ones stored in an internal node of the static structure. The reconstruction is done using the matrices encountered along a search path in a ranking tree. The subtree of the main tree containing the r 'th point then corresponds to the smallest row of $M' = M_j - M_i$ having a larger value than r .

The search ends in the leaf containing the queried element.

Assume that the search through the main tree is currently located in a node v. The predecessors of $i-1$ and j in $\mathbf{R_v}$ is found in constant time using the linear space predecessor structure. These correspond to the values obtained by $\mathrm{d}(i)$ and $\mathrm{d}(j)$ that we implemented using fractional cascading. The predecessor structure gives two pointers to the leafs storing the predecessors as well.

Now, the procedure traces the paths from the two leafs towards the root of $\mathbf{R_v}$. Steps are taken simultaneously, that is both paths ascend one level for each step until they merge in a node u. Commonly speaking node u is the lowest common ancestor of the two leafs. Assume the two paths arrived at node u from subtrees u_i and u_j, respectively. The subtrees to the left of u_i contain x-values in the range $[-\infty,\ i-1]$,

whereas the subtrees to the right of u_j contain values in the range $[j+1,\ \infty]$. On the other hand, we are certain that leafs in subtrees strictly between u_i and u_j correspond to points with x-value in the range $[i+1,\ j-1]$. Now let M'_u correspond to the row wise difference between the matrices immediately before the matrix of u_j and that immediately after u_i. The ℓ'th of M'_u counts the number of points stored in subtrees 1 through ℓ of v having x-values within some middle portion of the range. The left end of this middle portion correspond to the leftmost leaf of the subtree rooted at $u_i + 1$, and the rightmost interval correspond to the rightmost leaf in the subtree rooted at $u_j - 1$. The part of the interval not yet counted corresponds to leafs within the subtrees $u_s t R$ and u_i. The remaining leafs of the interval is included into M' by retracing the path from u to the leafs and adding the following to M'. For a node of the leftmost path we add the rowvise difference between the rightmost matrix and the matrix belonging to the subtree immediately to the right of the path. In the rightmost path we add the matrix belonging to the subtree immediately to the left of the path. When the search hit the leafs the the B^2 elements of the leafs are accounted for by a simple scan. As an alternative M' can be calculated bottom up by doing the calculations as we trace out the paths.

However, the subtractions and addition of full matrices are obviously too expensive. Instead an approximation section, W^u_{ij}, of the active section in M' is calculated by adding and subtracting corresponding sections of the matrices along the two paths – instead of full matrices. Figure 3.2 presents a graphical representation of the process. As we know from the static version adding sections only require standard RAM-operations on single words. Thus, the cost of calculating the approximation section is at most $\mathrm{O}(h) = \mathrm{O}(\log n / \log\log n)$.

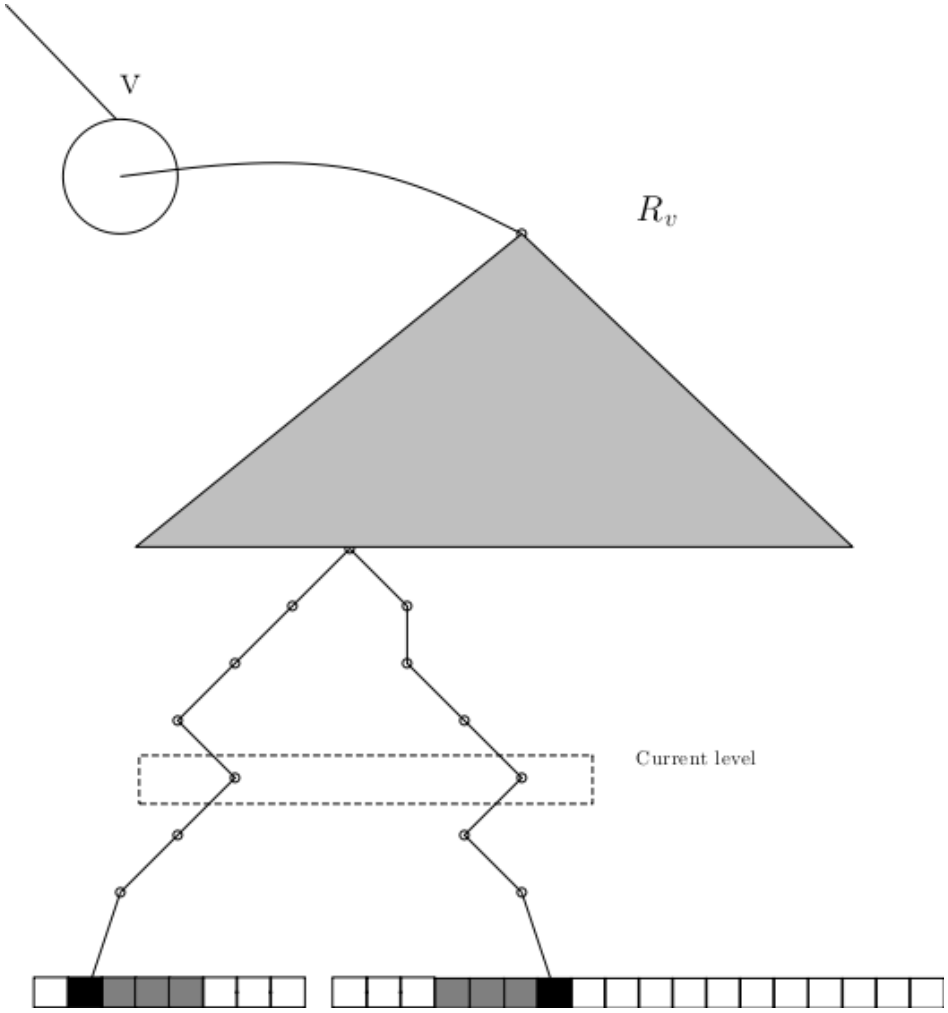

Figure 3.2: *A graphical representation of the calculation of W_{ij}^u. The gray squares represent the roots that has already been counted, whereas the black squares are the leafs holding the predecessors of i and stR in R_v.*

The data structures keeps an active section counter c initially set to one, which is incremented each time the search advances one level.

Since we make at most h subtractions of sections at most h cascading carries that would have been counted in the corresponding section of M' is not counted. Similarly, we make at most h additions while calculating W_{ij}^u. The c'th section W_{ij}^u is therefore at most $\lceil h \rceil$ too large and accordingly at least $\lceil h \rceil$ too small compared to the c'th section of M'. Additionally, the update buffer of u containing at most B^2 postponed updates could change the ℓ'th row by $\log B^2 = \mathrm{O}(\log \log n)$. This will only affect the last section directly, but a cascading query could cause the c'th section to change by one to either side. After calculating the range $[\ell_1, \ell_2]$ in exactly the same way as for the static data structure, the maximum difference among the $\ell_1 + 1$'th row

and $\ell_2 - 1$'th rows of M' becomes $2h + 2$. If we add a zero bit needed by the invariant, the total overlap between sections becomes $\lceil \log(2h + 2) + 1 \rceil = O(\log \log n)$.

Next we need a way to actually get the rows of M' needed for the binary search. Since we have no need for all $c(v)$ rows – and calculating them would be rather expensive – we only ever calculate a single row of M' at a time. We will employ the following procedure to retrieve an arbitrary row of M'. It is basically a matter of employing the process of calculating W_{ij}^u. However, we use the first copy of the matrices all the way in order to calculate precisely. In addition, the update buffer of the node u must be taken into account now. Correcting for the missing updates is really simple when working on only one row. Any missing insert of a point that falls within the query range, that was placed in a child with index smaller than or equal to the row in question causes an increase by one. An analogous delete would cause the target row to decrease by one. Unfortunately, it is too expensive for us to scan though the update buffer. Instead, a table is created, from which we can extract the wanted information using the constant number of words storing the update buffer as index.

Finding the smallest prefix larger than r in M' is done in much the same way as we did in the static case. We start by checking the interval end points ℓ_1 and ℓ_2 and if none of these work, we perform a binary search on the rows of M' to find the correct row. The binary search reconstructs a row of M' from the matrix in each search step as described above. The cost of performing a binary search is therefore

$$\begin{aligned} O(\log f \log_B n) &= O(\log \log (n) \cdot \log n / \log \log n) \\ &= O(\log n) \end{aligned}$$

Since a row is associated with a child of v, we can continue the search at this node. The active section counter c can at most be incremented $\lceil \log n / g \rceil$ times making the worst case accumulated cost of binary searching during a query $o((\log n / \log \log n)^2)$.

Now, we will recap and show that this solution actually answers range selection queries in the time required.

The height of the main tree is $O(\log_B n) = O(\log n / \log \log n)$. A range selection query consisted of a root to leaf search in the B-tree while performing a series of operations in internal nodes along the path. Calculating the approximation section W_{ij}^u takes time $O(h) = O(\log_B n) = O(\log n / \log \log n)$. The time spent processing this word, that is finding the range $[\ell_1, \ell_2]$ is $O(1)$.

Finding the next subtree was accomplished by trying the interval endpoints ℓ_1 and ℓ_2 and, if necessary, perform a binary search retracing the path from the leafs to their least common ancestor at most $\log n$ times. We saw above that this at most takes $o((\log n / \log \log n)^2)$ time accumulated over the whole range selection query. Hence, the time querying the ranking tree is bounded by the time spent calculating

W_{ij}^u. The worst case time of a range selection query thus becomes

$$\begin{aligned} \mathrm{O}(\log_B(n)\log n/\log\log n) &= \mathrm{O}((\log n/\log\log n)(\log n/\log\log n)) \\ &= \mathrm{O}((\log n/\log\log n)^2) \end{aligned}$$

We can hereby conclude that this solution correctly answers a range selection query and that it does it in worst case time $\mathrm{O}((\log n/\log\log n)^2)$.

Updates

The essence of a dynamic data structure is having fast updates. The reason is that "dynamization" always adds an extra cost on queries compared to online static solutions, and if update time is very bad and we are likely to make many queries and only few updates we could be better off just completely rebuilding a static solution every time an update is issued. This is not the case for this data structure however. First of all, the building time of the static structure is at best $\mathrm{O}(n\log n)$, since sorting is involved. Moreover, the dynamic solution supports both queries and updates in the same theoretical time bound, namely $\mathrm{O}((\log n/\log\log n)^2)$.

As I mentioned in the introduction the entire structure is rebuilt for every $n/2$ updates. Rebuilding the entire tree can be done in time $\mathrm{O}(n(\log n/\log\log n)^2)$, building the tree and ranking structures bottom up, adding an amortized cost of $\mathrm{O}((\log n/\log\log n)^2)$ to each update.

Insert

In order to insert a point (x,y) in the dynamic data structure a search for y is performed in the main B-tree, and the point is inserted into a new leaf. Inserting a new leaf could cause its parent to overflow. In this case, the parent is split into two new nodes. For each internal node v on the path from the root to the new leaf, x is inserted into both the ranking tree $\mathbf{R_v}$ and the predecessor structure. This could cause a number, at most h, of the nodes in $\mathbf{R_v}$ to split. A split in $\mathbf{R_v}$ is handled by introducing a new node and rebuilding the parent.

We still have to address the problem of dividing the associated structures, ranking tree and predecessor structure, when an internal node splits. We will do this by keeping 2 additional structures of each type in each internal node of the main tree. One that will be distributed to the left node of the split, and one for the right node. This is possible since it is known exactly when the node splits, and which children will be distributed to the left and right nodes, respectively. While updating the ranking trees or predecessor structure a similar update is issued in either the left or right structure depending on from which child the update came. The same technique is used to split the matrices stored by internal nodes of the ranking tree. This technique only introduces a constant space and insert time overhead.

After inserting x in $\mathbf{R_v}$ the entire path from the root to the leaf containing x is visited and a description of the update is appended to the update buffer. Once the

update buffer of a node is completely full, that is, for every B^2 updates the matrices of this node are rebuilt.

The internal node of a ranking tree stores at most f matrices, each having at most f rows. Since prefix sums are very easy to calculate in constant time per row, we can recalculate all matrices in time $O(f^2)$. Since $B = f$ this means that each of the updates added to the update buffer is handled in $O(1)$ time amortized.

The search for the correct leaf in the main B-tree can be done in $O(\log_B n)$ time. For each internal node on the path an insert in a $O(n)$-sized B-tree, the ranking tree, is performed. What about the cost of splitting a node? Since the children of the original nodes are divided between the two new nodes we have to change the pointers of these nodes costing $O(B)$ RAM-operations per node split. Sharing the ranking tree and predecessor structure cost only constant time since we employed the gradual global rebuild as described above. Furthermore, the amortized number of splits caused by a update on B-tree is known to be two. This result also applies to weighted B-trees. Hence the amortized cost of inserting in the main tree becomes $O(B)$. By the same arguments the insert of a nw leaf in a ranking tree costs $O(\log_B n)$ for searching and $O(B)$ for splitting. Note, that appending an update to the update buffers of internal nodes is assumed to be done in constant time. The overall time spent inserting the new point into a single rankning tree thus becomes $O(B + \log_B n)$. Since we perform an insert in a ranking tree for each internal encountered on the path from the root to the new leaf the total update time becomes

$$\begin{aligned} O(B + \frac{\log n}{\log\log n}(1 + B + \log n/\log\log n)) &= O(B + \frac{B\log n}{\log\log n} + (\log n/\log\log n)^2) \\ &= O(\frac{\log^{1+\epsilon} n}{\log\log n} + (\log n/\log\log n)^2) \\ &= O((\log n/\log\log n)^2) \end{aligned}$$

We can therefore conclude that the time used by an insert operation is as predicted.

Delete

A delete takes a point (x, y) value and deletes this point from the structure if present. Contrary to inserts the delete update does not change the main tree forcing fuses/shares along a leaf-to-node path. In stead nodes are marked as deleted and removed when the structure is rebuilt every $n/2$'th update.

Like the insert operation a search for the leaf holding the y-value of the point in question is performed. If the search did not succeed in finding a node we simple break out of the operation. In the other case the found leaf is marked as deleted but not removed from the tree. If the marked leaf was the last non marked child of its parent, the parent is marked. This marking process continues up the tree. After we are done marking nodes in the main tree, a similar marking process is performed on the associated structures of internal nodes. This is done by searching for the point

in the ranking tree adding delete updates to the update words and marking nodes along the way. As mentioned earlier the nodes marked due to deletes are removed when the global rebuild is performed. This happens every $n/2$'th update.

Searching for the leaf containing the point in question still takes time $O(\log n / \log\log n)$. Marking a node in the ranking tree for an internal node v also takes time $O(\log n / \log\log n)$. If the marked elements x-value is present in the predecessor structure of v we have to delete it from here too. For any reasonable implementation of this structure the cost of this is at most $O(\log\log n)$ and is thus dominated by searching through $\mathbf{R_v}$.

We can therefore conclude that the delete operation is accomplished in amortized time $O((\log n / \log\log n)^2)$.

3.3 Constant Time Static Algorithm

We have now seen data structures solving RMP in both a static - and dynamic - setting. As was mentioned quite a few times these structures were in fact very versatile. Not only did they support range median queries they were in fact able to solve queries for any rank in a range. For that reasong these data structures were dubbed range selection data structures.

The data structure that will be described next is a genuine range median structure. It uses properties only pertaining to the median of a range and can not answer arbitrary range selection queries. This structure shares a lot of ideas with the constant time structure of Petersen and Grabowski presented in Section 2.4. However this data structure only has one level and exploits some basic properties of the median. The structure supports queries in constant time on the average using only $O(n^{3/2})$ words of space and $O(n^{3/2})$preprocessing time. By using some clever trick the space and preprocessing time can be reduced by a $\log n$-factor on the average.

Constant Time Structure

Imagine that our input array $\mathbf{A}$ is divided into three parts:

- A part $\mathbf{M}$ consisting of the middle $2a$, where $a = O(\sqrt{n})$, elements in $\mathbf{A}$ [3], ie $\mathbf{A}[a+1, n-a]$.
- A part $\mathbf{F}$ consisting of the first a elements. This is equal to the range $\mathbf{A}[1, a]$.
- A part $\mathbf{E}$ consisting of all the remaining elements not yet covered. That is, all elements in the range $\mathbf{A}[n-a+1, n]$.

Furthermore assume that for any query $\mathbf{Q}(i, j)$ it is true that $i \in \mathbf{F}$ and $j \in \mathbf{E}$.

[3] Rounding issues are skipped here for brevity.

The median of $\mathbf{M}$ is the element of rank $\lfloor n/2 \rfloor - a$ in $\mathbf{A}$. This is obvious since $\mathbf{M}$ contains $n - 2a$ elements. We know from Lemma 2 on page 8 that the median of the union of $\mathbf{F}$, $\mathbf{E}$ and $\mathbf{M}$, is either contained in $\mathbf{F}$ or $\mathbf{E}$ or consists of a shift of rank of at most a from the median in $\mathbf{M}$. This specifically means that if not contained in $\mathbf{F}$ or $\mathbf{E}$ the median is within the range $[\lfloor n/2 \rfloor - 2a,\ \lfloor n/2 \rfloor]$ of $\mathbf{M}$ both inclusive. It should be noted, that no matter the specific choice of i and j this property holds for the median. The data structure stores these $2a + 1$ elements in a sorted array $\mathbf{M_c}$.

If we, for a minute, assume that all elements in $\mathbf{F}$ and $\mathbf{E}$ is either smaller than $\mathbf{M_c}[1]$ or larger than $\mathbf{M_c}[2a + 1]$. Furthermore, let s_l and s_r be the number of elements from $A[i, a]$ and $A[n - a + 1, j]$, respectively, smaller than $\mathbf{M_c}[1]$. These quantities are assumed precalculated during preprocessing.

The number of elements larger than $\mathbf{M_c}[2a + 1]$ must then be equal to the remaining number of elements due to the assumption we made on $\mathbf{F}$ and $\mathbf{E}$. We'll denote this quantity $b = j - i + 1 + 2a - n - s_r - s_l$. The median of the union is now located in $\mathbf{M_c}$ and it has rank $a + 1 + \left\lfloor \frac{b - s_l - s_r}{2} \right\rfloor$.

Unfortunately, saying that all the middle elements of the range is contained in $\mathbf{M_c}$ is a bold assumption. In an arbitrary input there can of course be elements in $\mathbf{F}$ or $\mathbf{E}$ that is larger than $\mathbf{M_c}[1]$ and smaller than $\mathbf{M_c}[2a + 1]$. The data structure store these elements explicitly in a sorted list X, along with its index. I will now analyze the expected size of $\mathbf{X}$.

We will determine an upper bound on the probability that an arbitrary element x from $\mathbf{F}$ or $\mathbf{E}$ intersects the array $\mathbf{M_c}$. That is, $\mathbf{M_c}[1] \leq x \leq \mathbf{M_c}[2a + 1]$. Since the element x is drawn randomly the probability of it having rank $1 \leq t \leq n$ in $\mathbf{A}$ is $\frac{1}{n}$. There are $2a + 1$ illegal ranks, that would cause the element x to intersect $\mathbf{M_c}$. Thus, the probability we were looking for can be stated as follows:

$$\begin{aligned} P_r[\mathbf{M_c}[1] \leq x \wedge x \leq \mathbf{M_c}[2a + 1]] &= \frac{2a + 1}{n} \\ &= \frac{2a + 1}{\Theta(a^2)} \\ &= \mathrm{O}(1/a) \end{aligned}$$

If we assume that the elements of $\mathbf{E}$ and $\mathbf{F}$ are added independently one by one, the expected size of the set $\mathbf{X}$ becomes:

$$\mathrm{E}(|\mathbf{X}|) = \sum_{x \in \mathbf{F} \cup \mathbf{E}} P_r[\mathbf{M_c}[1] \leq x \wedge x \leq \mathbf{M_c}[2a + 1]] = 2a\,\mathrm{O}(1/a) = \mathrm{O}(1)$$

For a general range query $\mathbf{X}$ is scanned in constant time, and the elements inside the range $[i,\ j]$ are appended to a list $\mathbf{X'}$. Note that our value b, calculated like above, has now become too big. By subtracting $|\mathbf{X'}|$ from b we moved the calculated median $\frac{\mathbf{X'}}{2}$ to the left before taking the actual elements of $\mathbf{X'}$ into account. This means the median of $A[i, j]$ is equal to the element of rank $a + 1 + \left\lfloor \frac{b - s}{2} \right\rfloor + \frac{\mathbf{X'}}{2}$

in $\mathbf{M_c}$. Alternatively, we notice that adding the elements of $\mathbf{X'}$ to $\mathbf{M_c}$ can at most shift the median $\frac{|\mathbf{X'}|}{2}$ to either side. Consequently, the median must either be an element from $\mathbf{X}$ or within the subarray $\mathbf{M_c}[a+1+\left\lfloor\frac{b-s}{2}\right\rfloor, a+1+\left\lfloor\frac{b-s}{2}\right\rfloor+|X'|]$. This means that the median of $A[i,j]$ can be found by doing selection after rank $\frac{|\mathbf{X'}|}{2}$ within the set $\mathbf{M_c}[a+1+\left\lfloor\frac{b-s}{2}\right\rfloor, a+1+\left\lfloor\frac{b-s}{2}\right\rfloor+|\mathbf{X'}|] \cup \mathbf{X'}$. The way we will go about this is to simply merge the two constant sized arrays. Elements from $\mathbf{X'}$, that is smaller than $\mathbf{M_c}[a+1+\left\lfloor\frac{b-s}{2}\right\rfloor]$ or larger than $\mathbf{M_c}[a+1+\left\lfloor\frac{b-s}{2}\right\rfloor+|\mathbf{X'}|]$ are not added to the merge since we do not have enough information to conclude anything about their rank within $\mathbf{M_c} \cup \mathbf{X'}$. They still move the median either left or right however, and therefore they are counted. Assume that we, while doing the merge find t_l elements from $\mathbf{X'}$ smaller than $\mathbf{M_c}[a+1+\left\lfloor\frac{b-s}{2}\right\rfloor]$ and t_r elements larger than $\mathbf{M_c}[a+1+\left\lfloor\frac{b-s}{2}\right\rfloor+|\mathbf{X'}|]$. The median of $\mathbf{A}[i,j]$ is now the element of rank $\frac{|\mathbf{X'}|}{2}+\left\lfloor\frac{t_r-t_l}{2}\right\rfloor$ within the merged elements.

The merging is done in time $O(|X'|)$, that is, constant time on the average.

The above method assumed that i and j were located within the first a and last a elements, respectively. That is, of course, not necessarily the case. We can still use the above idea though, but it has to be generalized a bit. First of all i and j could be closer together than $\Theta(n-\sqrt{n})$ elements. Morever, the two extremes of the range will probably not be evenly distributed around the middle of $\mathbf{A}$ that is contained in different halfs. What we need is a covering scheme that will take the above solution and generalize it to all ranges in $\mathbf{A}$ ensuring all queries of all lengths can be answered.

First of all, notice that the size of $\mathbf{A}$, which is always denoted n, is merely a constant here. The calculations – of b, s_l and $\mathbf{M_c}$ – used in the above method works for any fraction of n, and we can therefore split – or trisect – an arbitrary contiguous subarray of $\mathbf{A}$. In order to elaborate on this idea we introduce categories of divisions with different values of a, that is lengths of start end ending pieces of the trisection. These are intended to answer queries with different widths of query ranges – with larger categories handling wide queries. For each category divisions are distributed over the input array $\mathbf{A}$, such that we are able to answer queries for any range of width corresponding to the category. I will now describe the covering scheme proposed by the authors.

Covering Scheme

I will start by describing the structure of the categories used in this covering scheme. later I will argue why it can be stored using only $O(n^{3/2})$ words of memory, and demonstrate how it can be calculated in time $O(n^{3/2}\log n)$.

A category in this covering scheme is identified by the width of queries it sup-

ports. More specifically category d is capable of answering all possible queries that has width between d^2 and $d^2 + 2d$. Since $d^2 + 2d + 1 = (d^2 + 1)$ the ranges of query lengths supported by successive categories d and $d + 1$ are contiguous. Obviously $d \in [1,\ \sqrt{n}]$ meaning this layout of categories handles all query widths possible.

For an arbitrary category $\lfloor n/d - d + 1 \rfloor$ subarrays are trisected into first, middle and last part and for each of these the 2^d median candidates are stored. So are the values s_l and s_r. Each subarray has length $2i + (i^2 - i) + 2i$. The trisection consists of a starting and ending part each of length $2i$, and a middle part of length $i^2 - 1$. The layout is such that the middle part C of the k'th subarray starts at index $dk + 1$ for $k \in [0,\ n/d - d]$. This actually means that the first and second subarray of index 0 and 1, respectively, starts outside **A**. Equivalently, the last subarray can end outside **A**. This can be handled by adding dummy values and we will not bother with this in this description.

I owe to describe how the categories and subarrays actually come into play while answering a query. Given an arbitrary query $\mathbf{Q}([i,\ j],\ r)$ the the category $d = \lfloor \sqrt{j - i + 1} \rfloor$, contains the longest subarray for which the middle part will be contained within the query range. We now have to identify the subarray, of category d, having i and j within **F** and **E**, respectively. Since the start of successive subarrays shift by d elements i is contained within exactly two different "left parts", those of the $\lfloor i/d \rfloor$'th and $\lfloor i/d \rfloor + 1$'th subarray. Now if the query width of the query range is within the range $[d^2,\ d^2 + 1]$, the subarray of index $k = \lfloor i/d \rfloor$ is used and in any other case the one with index $k + 1$. It is worth noticing that, since we round down in our calculation, i is always located in the last half of **F**. The reason for this is that a query range of length d^2 or $d^2 + 1$ must have its left endpoint within the second half of **F** while the right endpoint is within the first half of **E**. We can therefore safely choose the k'th subarray and run our median finding algorithm as described earlier. If the query range has width in the range $[d^2 + 2,\ d^2 + 2d]$, j is definitely contained within the last d elements of the k'th subarray, or equivalently within the last $2d$ elements of the $(k + 1)$'th subarray.

I will now analyze the space usage of this particular covering scheme. The category d contains $\mathrm{O}(n/d)$ subarrays, each storing $\mathrm{O}(d)$ median candidates, in their M_c lists, and $\mathrm{O}(d)$ words of auxiliary information. Summing over all $\mathrm{O}(\sqrt{n})$ categories this gives a space consumption of $\mathrm{O}(n^{3/2})$.

The question is now, how should we actually go about implementing this? I will now demonstrate how to construct all subarrays of the category d in $\mathrm{O}(n \log d + n)$ time. The overall running time, taken over all categories, is then dominated by the last category taking time $\mathrm{O}(n \log \sqrt{n}) = \mathrm{O}(n \log n)$.

We will start the preprocessing of the d'th category, for $1 \leq d \leq \sqrt{n}$ by creating an empty balanced binary search tree. This tree will hold the elements in the middle section **M** of the current subarray. As a way to start the process the first $d^2 - d$ elements of **A** are added to the binary search tree. The $2d + 1$ elements around the median of **M** is found by doing a standard range query on the binary search tree in time $\log(d^2 - d) + d$ and stored in an array accessible through the index of the subarray. The remaining $4d$ elements in the sections **F** and **E** are then compared with

the median candidates by a simple scan, during which the auxiliary information s_l, s_r and $\mathbf{X}$ is collected. The elements corresponding to the first d of $\mathbf{M}$ is then deleted from the tree, and the first d of E is inserted. Now the binary search tree contain the elements of the center part of the next subarray. This process continues until no more subarrays need processing.

There are n/d subarrays in each category, and we use $O(d + \log d)$ time for each, disregarding the time used inserting and deleting elements in the search tree. This gives a total running time of $O(n)$ solely for processing the subarrays. In fact, the time spent in each category is actually dominated by inserting in and deleting from the binary search tree. Each element is inserted once and deleted once from a tree of size $O(\log(d^2 - d))$, giving a total time of $O(n \log(d^2 - d)) = O(n \log d)$.

Chapter 4

Implementation

In this chapter I will discuss implementations I made of some of the static algorithms presented in Section 3.1. The implementations will be accompanied by plots demonstrating that these implementations actually follow the expected theoretic bounds both in terms of query time and space. In the final Section I will compare the time and space usage of the different solutions choosing the solution best suited for everyday use.

I have implemented the following algorithms:

- A very naive solution simply copying the desired range of the input array and sorting it to find the element of wanted rank. I will actually not perform experiments on this solution. It is simply too slow. Instead it has been used for testing the correctness of the other solutions.
- A variant of the pointer machine efficient solution using binary search. Notice, in this implementation is actually not mode for the pointer machine.
- The actual pointer machine solution using fractional cascading. Again I use random access within the data structure making it a RAM data structure. I made two different implementations of this structure that will be treated separately.
- The space efficient solution using a binary rank tree.

The implementation was done in C++ using templates ensuring easy customization of the data structures. All implementations have been tested against the naive implementation, that definitely works, and are currently working as expected. The testing procedure consisted picking various random input arrays of increasing sizes. A naive and the structure under test was then created using the chosen input array. The process the fired random queries at the two data structures and checked that they gave consistent answers.

In the first section I will state the shared interface of these data structures and describe the behavior of some shared utility functions.

The second section will contain a brief introduction of the measures that will be used to determine the of the implementations. Furthermore, I will account for the experiments that were conducted in order to collect these measures.

In the third section I will present each of the implementations. Each subsection, except the naive solution, will be concluded by a small presentation of the results gathered by the experiments. Notice that I will defer all comparisons between the structure to the last section of this chapter.

4.1 Interface

I will now describe the common interface of the four implementations.

I will start by giving a brief overview of the machinery involved - and shared - by all the data structures.

The foundation of each structure is a binary tree. Nodes are created with an array as parameter. The tree is only linked from above, meaning there are no parent or sibling-pointers. The binary tree is created by initializing a root with the input array **A**, whereby a single node with two empty children pointers is created. When the first query is made to the structure **A** is split into an upper and lower half and the root creates its left and right children by making its child pointers identify two new nodes created with the lower and upper arrays, respectively. In my implementation the left child corresponds to the **lower** array that was mentioned in Section 3.1 whereas the right child represents the upper array. The leafs of the tree are defined to be nodes that has an input array of size 1.

The tree is built as queries arrive meaning that the splitting procedure performed on the root initially is repeated on internal nodes that have empty child pointers. As an alternative, a function that builds the entire tree splitting all nodes doing a preorder traversal on the tree as it is being created. The implementation of the algorithms does not include padding of **A** to make n a perfect power of 2. Hence, the tree is not necessarily perfectly balanced.

The way I implemented the four different versions was by plugging three different nodes into the above tree. A tree node supports the following operations:

- **splitNode** This operation splits the input array of its node object, taking the median of the input array as an argument and creates all auxiliary branching information.

- **find** This operation takes two indexes between 1 and n and a pointer to a piece of memory holding the current query parameters. The two indexes should define a range and the method returns the number of elements from the associated array of the node that falls within this range. Furthermore, this method is responsible for resetting i and j stored in the associated memory.

- **getLower** and **getHigher** accesses the children of a node. For a leaf they should simply return *null*.
- Utility methods like **getArray** and **getSize** that are there for technical reasons. Their function is exactly as would be expected by their respective names.

The actual range selection structure stores a pointer to the root as its way to access the tree. Initially, this pointer is set to *null*, and it is only initialized as part of a query or a explicit call to the preprocessing operation. The range selection interface is defined as follows:

- **RangeSelectionQuery** Given i , j and r find the element of rank r in $[i,\ j]$.
- **Preprocess** Build the entire tree at once.

The implementation of **RangeSelectionQuery** is the same for all tree implementation. The operation is implemented recursively. To accomplish this I have introduced a piece of memory which is passed with a query down the tree, storing the current versions of the query parameters. When a **find**-operation is performed the four values associated with this memory are changed. These represent the current i and j , depending on whether the query is guided to its left or right child. I.e. calculating the new query parameters is the responsibility of the node, not the **RangeSelectionQuery**-function. Notice that this is necessary to have a shared **RangeSelectionQuery**-operation between the different version, since the query parameters are changed in different ways according to how range counting is performed. The memory is only allocated once, not for every query, and is managed by a resource pool singleton.

A subprocedure of splitting a node during a query or a preprocess operations is calling the global selection function, which is implemented as a function object and passed as a template paramenter, to retrieve the median of the associated array of the current node. These implementations use a linear median algorithm which I will describe next. However, choosing a different method might yield better time measurements. Then the calculated median is used as an argument for **splitNode** which completes the task. This sequence of actions perform exactly what is depicted in algorithm 2, line 4 through 10, in Section 3.1.

I will now describe the linear selection algorithm that is used for finding medians. During the course of a query it is at most called $O(\log n)$ times.

Linear Selection Algorithm

The algorithm I have implemented is truly classical. It is the original algorithm by Plum et al, [BFP$^+$72], from 1972, that pioneered the search for an efficient method to solve the selection problem. Its complexity is linear in the worst case, but the

constants have later been shown to be very large. I will just briefly sum up the highlights of this algorithm.

The input is assumed to be an array $\mathbf{A}$ of length n and a number r where $1 \leq r \leq n$. The algorithm proceeds recursively, by discarding a certain portion of the array that cannot contain the element of rank r.

This portion is found in the following way. Divide the array into $\lceil n/5 \rceil$ subarrays of length five[1]. Since these subarrays have constant size, we find the median of each in constant time by sorting the arrays. The medians of these small arrays are collected in yet another array. The algorithm is then called recursively using the array of medians and $\lceil n/10 \rceil$. Note, that the result of the above call is the median of the medians. We can think of this as an approximation of the actual median of the array. Now the algorithm starts to resemble the tree data structures of Section 3.1. The input array $\mathbf{A}$ is divided into two arrays, $\mathbf{A_s}$ and A_l. These contain the elements from $\mathbf{A}$ smaller and larger than the median of medians, respectively. If $|\mathbf{A_s}| \leq r$, the algorithm is called recursively with $\mathbf{A_s}$. In the other case, recursion is called using $\mathbf{a_l}$ and $r - |\mathbf{A_s}|$ as arguments.

The recursion is stopped when the size of the input array is smaller than or equal to some constant. Historically, and in order to make the analysis work this constant was chosen to be 140.

Assume for a minute that we order the sorted subarrays of length five by the value of their median. The median of medians, $\mathbf{m}$, is then contained in the $\lceil n/10 \rceil$'th subarray, that is exactly in the middle. The median of medians is now larger than all medians of subarrays in the first half and smaller than all medians in the second half. We will now bound the size of $\mathbf{A_s}$ and $\mathbf{A_l}$.

We definitely know that all elements in the first half which is among the first 3 elements of their subarray are smaller than $\mathbf{m}$. Analogously, all elements in the second half that are among the last 3 elements of their subarray must be larger than $\mathbf{m}$. That is, $3n/10$ of the elements are distributed to A_l and a comparable amount, possible minus a constant, is distributed to A_l. We can not make any assumptions about neither the last two elements of each subarray in the first part neither the first 2 elements of subarrays in the second part. Nevertheless, the information gained ensures that A_s and A_l can at most contain $\lfloor 7n/10 \rfloor$ elements each. If we factor in the recursive call we made to obtain $\mathbf{m}$, the final recurrence relation becomes:

$$\mathrm{T}(n) \leq \begin{cases} \mathrm{T}(\lceil n/5 \rceil) + \mathrm{T}(7n/10) + \mathrm{O}(n) & \text{if } n > 140 \\ \mathrm{O}(1) & \text{else} \end{cases}$$

The above recursion can be proven to solve to $\mathrm{O}(n)$ by induction. I will not do this here.

I ended up doing a rather straightforward implementation of this algorithm. The recursive function is implemented by a C++ function object[2], with a private recursive utility function. The input arrays for this algorithm is a C++ STL vector, and

[1] The array can be padded to avoid special cases.

[2] A function object is a class where the only implemented function is the parenthesis-operator

two C++ vectors are used to represent the partition arrays A_s and A_l. Partitioning is done in one single scan over $\mathbf{A}$, and elements are added to the end of the partitioning vectors taking a total of $O(n)$ operations in total.

The above solution assumes that all elements are different in order to work. My implementations handles this issue during the scan of $\mathbf{A}$, by adding an element equal to $\mathbf{m}$ to A_s if and only if $|\mathbf{A_s}| \leq |\mathbf{A}/2|$. In the opposite case the element is added to A_l. This ensures that the recursion will end for input arrays that contain all equal elements, and the recursion function describing the worst case time still applies.

4.2 Measures and Experiments

In this section I will list the measures upon which we will conduct our analysis of the implementations. Furthermore, I will describe experiments designed for collecting these measures under different use scenarios of the data structures. These experiments will be done on each implementation, and in the end of this chapter I will compare the four main implementation on the basis of these results.

The two main measures that I will focus on are time spent which serves as an abstraction for simple instructions on the RAM, and the space usage. The tool used for measuring the time spent is low level hardware timers on Linux. The space measure is collected by overwritten *new* and *delete* operators increasing and decreasing a global variable, respectively. The variable is stored as a singleton, which means that one, and only, one can be instantiated. The singleton is updated by public methods and the cost of this would possibly make an impact on the time measurement. Therefore, the overwritten *new* and *delete* operators are turned on by compile flags and this should not be done while measuring time. Furthermore, the global singleton variable is only linked in when space measurements are collected.

I will conduct the following experiments to gather measures on the different implementations:

1. I The first experiment consists of posing a lot of random queries to structures with different sizes input arrays. That is, both the query range $[i,\ j]$ and the query rank r are chosen independently but in a way such that $i \leq j$. The input arrays are composed of randomly picked numbers. The sizes of the input array will be in the range $[2^{10},\ 2^{22}]$. The experiment is performed on both clean data structures taking the cost of preprocessing into account – and on fully preprocessed structures.

2. The space measure is analyzed through an experiment that calls the preprocessing function 10 times on clean data structures while measuring the space

"operator()". An invocation of such a function is then done by forming a temporary object only to call the parenthesis-operator on it. This is comparable to the concept of functors known from functional programming languages.

usage through the modified *new* and *delete* operations. The output of the experiment is the average of these 10 runs.

Technical Specifications

The experiments have been run on a Lenovo laptop equipped with a 2.6 GHz Celeron M CPU, 3 GB RAM and a Linux Fedora 13 distribution (Linux kernel 2.33.8). The machine was booted in kernel run level 1 in order to minimize the noise caused by a running GUI. The experiments were deployed as Python scripts calling an executable version of my code with various configuration options.

I am now done describing the parts of the implementation that is shared between all the tree based data structures. In the next 4 sections I will describe the implementation of the four different versions one by one. I will focus on the details that distinguishes the four versions.

4.3 Naive Implementation

This implementation is probably the most naive way you can think of solving the range median/selection problem. It is, however simple, still a data structure following the above interface. Hence, it is constructed given an input array **A**. When processing a query, it does not change the tree at all. Instead it copies the range specified by the query directly from **A** into a new temporary STL vector and uses standard the STL sort operation to sort this. The element of selected rank is then extracted directly from this temporary vector and returned.

The space usage of this datastructure is $O(n)$. The query time is $O(n \log n)$ worst case. It should also be noted that this data structure, does not ”learn“. I.e. ten queries using the same range and rank will take an equal amount of operations.

4.4 Binary Search Implementation

The binary search version also support the described interface. I will describe the way the node for this specific tree based structure has been implemented as well as the way a query is handed off to a child node.

Notice that we, in this implementation, search for the elements actual position within **A** all the way down the tree. Therefore the elements stored in the associated structures of the nodes are augmented with its original index in **A**. A comparator operator has been implemented for this element/index pair, such that this change has no impact on neither the **RangeSelectionQuery**-function nor auxiliary selection function.

The **splitNode**-operation is implemented just as would be expected. That is, there are no special considerations to be taken here. Notice, that a simple left to right scan always produce upper and lower vectors sorted by index. Hence, the

conditions for doing a binary search are met. The binary search is implemented in a public operation **PerformBinarySearch** that takes an index and returns the number of elements within the associated array having at most this index in **A**. The implementation is a standard discrete binary search.

The **find** operation then calls the **PerformBinarySearch** twice. Once with the index immediately before the left endpoint of the range given as input and once with the right endpoint. The difference between these are then returned. No change is done to the memory holding the current i and j , since we have access to the original index of all elements.

To sum up, the most significant features of this implementation are:

1. Elements are augmented with their original index in **A**.
2. Binary search is used to implement the find operation, and we perform exactly two on each level of the tree.
3. The query parameters, sent with the recursion through the tree, never change after they are handed to the node in the top of the recursion.

As we discussed in Section 3.1 the theoretical time spent answering k queries in this implementation is $O(k \log n^2 + n \log k)$. The space used is $O(k \log n)$ for $k \leq O(n)$ and $O(n \log n)$ for a fully preprocessed structure.

Results of Experiments

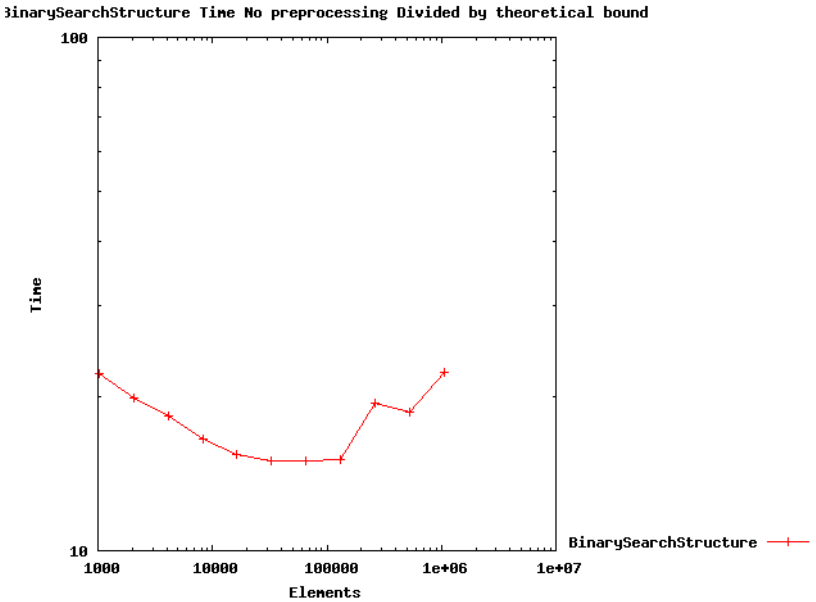

Figure 4.1: Results of experiment 1 performed on a clean binary search structure. Divided by the theoretic time bound which is $10000 \log n^2 + n \log 10000$.

Figure 4.1 shows the time measurements of 10000 random queries on a clean binary search based structure. The plot looks a bit strange since it starts of by a steep downhill slope and then suddenly takes an uphill slope. I think this is due to the random nature of the queries and the relatively small constant that this query time analysis seems to hide. Anyways it does not stray to far from a constant line. It is therefore safe to assume that the query time is in fact $O(k \log n + n \log k)$.

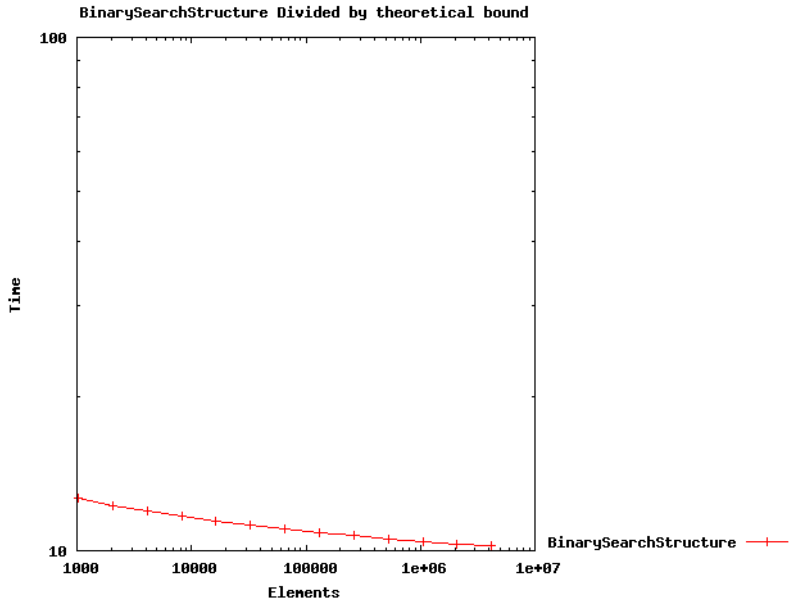

Figure 4.2: *Space usage of a fully preprocessed binary search structure.*

Figure 4.2 shows the results of experiment 2 on the binary search solution. It does not quite start out resembling a constant line, but appears to converge towards the end.

4.5 Fractional Cascading Structure

I will now describe the implementation of the tree structure where elements are linked with their predecessors down the tree. From here I will refer to this as the fractional cascading structure. Even though I stated in Section 3.1 that this structure is optimal in the pointer machine model my specific implementation is intended for the RAM model. I.e. the associated arrays of the nodes are just ordinary vectors and elements can be accessed using fractional cascading. Knowledge about the original index within the original array is not needed in this data structure. Therefore, contrary to the implementation described in the previous section the input vector $\mathbf{A}$ is given to the root without modifications.

I will start by describing the node that is plugged into the tree template in order to construct the range selection data structure. Aside from the associated input array this node stores fractional cascading pointers for each node. The storage

method used for these pointers is a point of variation. This is accomplished through a builder pattern interface defining getter and setter methods. The setter should receive an index representing a position in the input array and two integers indexing elements within the lower and upper arrays. After invocation of the method fractional cascading pointers are set such that the element representing by the first index references the positions on the level below designated by the two indexes. The getter method takes an index as input and returns the fractional cascading pointers originating from the element identified by the index as a pair of integers.

An implementation of this interface will be called a pointer storing strategy from now on.

Note that the type of pointer storing strategy used is passed to the node as a template parameter. Hence, it is known at compile time.

I have implemented two different concrete pointer storing strategies. The first type stores two C++ vectors which I will denote **leftFracPtrs** and **rightFracPtrs**. The integer stored in the ℓ'th position of **leftFracPtrs** contains the index of its predecessor, with respect to original index in **A**, in the associated array of the left child. The ℓ'th integer in **leftFracPtrs** represents the same, but for the associated array of the right child.

The second pointer storing strategy I implemented tries to benefit from storing elements that will most likely be used together close together. Note that both fractional cascading pointers of the current i and j are of interest to us since they are used to reset the memory passed on to one of the children of the node we are currently visiting. Therefore, whenever accessing one we will get the other. This storage method places the right and left fractional cascading pointers of each element in a C++ STL pair. The rationale behind this pointer storage strategy is that both pairs of pointers for the elements of interest are needed in every application of **find**. Hence, we allegedly save some random access just returning them as pairs, instead of indexing into two different vectors not necessarily adjacent in memory. We will compare these two methods of storage later.

The **splitNode** operation constructs and fills out the fractional cascading pointer structure. While processing an element during the splitting process the setter method of the pointer storage strategt is called with the index of the current element and the current sizes of the upper and lower arrays after the element has been inserted. Note that this means that exactly one of the pointers will point to the element itself on the level below.

The **find**-operation is implemented by taking the calling the getter method on the pointer storing strategy with i and j , respectively. Note that only the first element of the resulting pairs are of interest for calculating the range count. By definition these values are exactly equal to the number of elements inserted prior to i and j . Hence, by subtracting the first from the ladder, we get the range count of elements in the lower array of this node. The four query parameters kept in auxiliary memory are set to reflect the pairs returned by the pointer storing structure.

Since each element in the fractional cascading vectors is a 32 bit, or one word, integer, this implementation uses 2 words of space for each element on each level. Furthermore, we still store the associated element arrays, also costing 1 word per element on each level. If we sum over all levels the total space becomes $O(3n \log n)$. Notice that compared to the binary search solution this is one extra word per element on every level.

The time for k queries, however, becomes $O(k \log n + n \log k)$, since we only use constant time on each **find**-operation and constant additional time setting up the fractional cascading pointers.

Results of Experiments

We would expect the fractional cascading solutions to use a sizable amount of space. That is, for a fully preprocessed data structure I would expect the constant hidden in the analysis in Section 3.1 to be quite large. Since, the vector-implementation and the STL-pair implementation store the same amount of integer pointers I would not expect there to be a large difference between the space measured by the experiments. Perhaps, the effect of not allocating memory, as is done by C++ vectors in order to ensure constant time insertion, could cause the pair-strategy to win by a smidge.

The query time of this solution is expected to approach the theoretical time bound. Aside from that, the constant of the "bigOh"-notation should be relatively small. After all, the preprocessing phase is mostly inserting elements into different vectors.

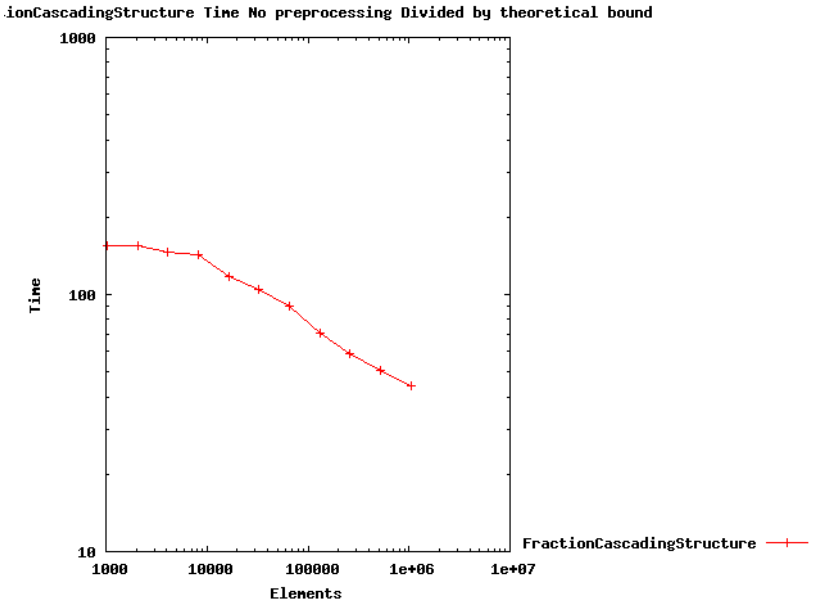

Figure 4.3: *Results of runnning experiment 1 on a clean fractional cascading structure using the vector pointer strategy. Divided by the theoretic time bound which is* $10000 \log n + n \log 10000$.

A plot of the values obtained from running experiment 1 on the fractional cascading structure using the vector pointer strategy is shown in Figure 4.3. The values have been divided by the expected theoretical running time for $k = 10000$ queries. The connected points seem to converge to a constant line towards the end. The downhill climb observed in the beginning, could be explained with large amount of preprocessing done by the first few queries. The probability of hitting a node that has not yet been preprocessed is simply to small. As n grows the depth of the tree becomes the dominant factor as preprocessing becomes expensive even on lower levels. The value of the points are somewhere between 100 and 200 the constants hidden by the ”bigOh“ notation are a bit large. Most likely the cost of recursion and random access have a large part to play here. Just as expected values seem to decrease as n grows and these small constants lose impact.

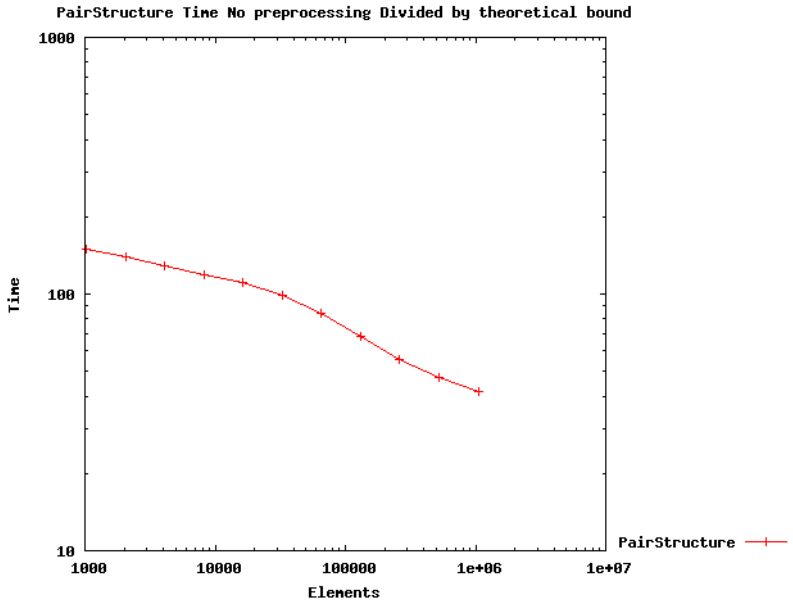

Figure 4.4: *Results of experiment 1 on a clean fractional cascading structure using the pair pointer strategy. Divided by the theoretic time bound which is* $10000 \log n + n \log 10000$.

Figure 4.4 shows the result of experiment 1 on the fractional cascading structure using the pair pointer strategy. The values are divided by the theoretical time bound for $k = 10000$ queries. As was the case with the vector pointer strategy the points seem to converge to a constant line towards the end. Aside from that we see the same steep decline for smaller n. Note that the actual values here are a bit smaller, hinting at a smaller constant.

The space usage of the two strategies should be relatively similar. The only thing that could sway the result in favor of the pair pointer strategy is the possible excess capacity in C++ vectors. Even though we never use it it is allocated and counts towards the space usage.

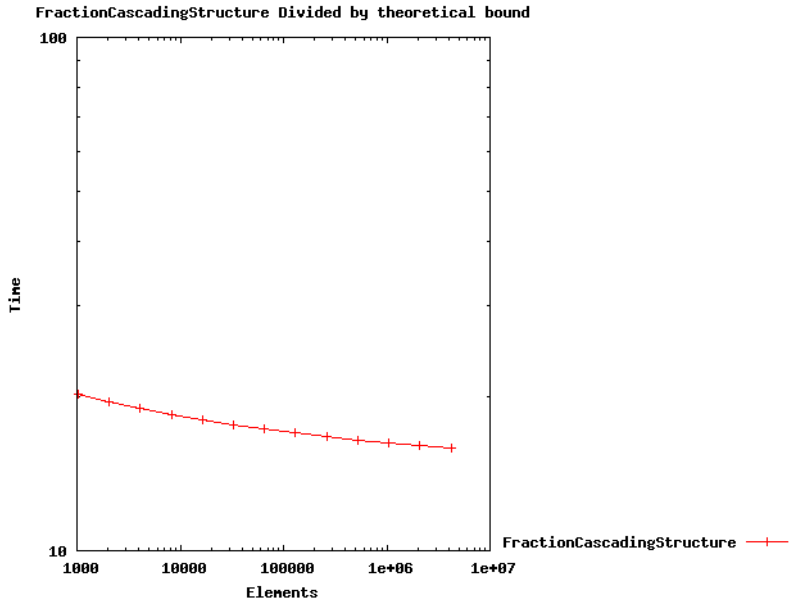

Figure 4.5: *Space usage of the factional cascading structure using vector strategy for storing pointers. Divided by theoretical space bound which is* $n \log n$ *for the fully preprocessed structure.*

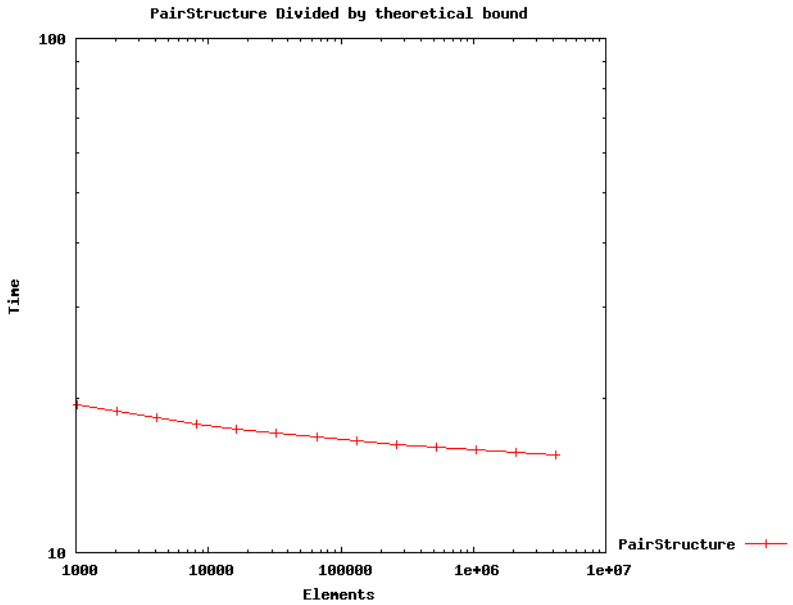

Figure 4.6: *Space usage of the factional cascading structure using pair strategy for storing pointers. Divided by theoretical space bound which is* $n \log n$ *for the fully preprocessed structure.*

The plots depicted in Figure 4.5 and Figure 4.6 shows the amount of space for different values of n divided by $n \log n$. The points of both plots tend to converge to a constant line.

From the plots we have seen in this section we can conclude that the measures gathered from experiments are consistent with the analysis we performed in Section 3.1. Furthermore, this is true for both pointer strategies.

I will compare the two pointer strategies more thoroughly in the concluding section of this chapter.

4.6 Binary Rank Tree Implementation

I will now describe my implementation of the data structure using a linear time space binary rank data structure for performing range counting queries. The node I implemented to complete this data structure has additional parameters that I will briefly account for.

The word size **w**, does not depend on the actual size of **A** but can be chosen and passed to the data structure as it is declared. This is accomplished by a template argument. Furthermore, the node in itself has a query parameter specifying which binary rank data structure to use. Note, that these things can only be set on compile time – not on runtime. In this way the implementation should avoid some of the hidden performance costs of polymorphism.

The binary rank structure used for this first take on an implementation is the one described in Section 3.1 adherent to the description of the linear space data structure. It is called **Verbatim** and supports rank queries in constant time using only $n + o(n)$ bits of space where n is the size of the bit string. The **rank**-operation involved a call to a **POPCNT**-function, that is either implemented as a machine instruction on the CPU or by means of a table. My implementation of **Verbatim** uses the *__builtin_popcnt* operation of the GCC tool chain, that uses a combination of a compact table and bit operations.
The arrays that store prefix sums for blocks and super blocks are actual arrays of integers each using one word of space. Hence, I have chosen the word size to be 32 bits, which is larger than $\log n$ for all real world applications. It is certainly enough for what I had capacity to test. An immediate space optimization of this structure would of course be to change the memory layout into a more compact bit string, and retrieve a certain portion using bit operations and masks instead of indexing.

The **splitNode**-operation creates the lower and upper vectors in the usual way. In the process, it fills out a n'-bit binary string, where n' is the size of this nodes associated array, by setting the ℓ'th bit if the ℓ'th element is distributed to the lower array. An instance of the binary rank structure is the created using the bit string, and the node is given a pointer to this structure. It is possible for other nodes to obtain a copy of this pointer. When the node has been split the two child nodes are created and given their associated arrays. The final thing done by the pseudo code corresponding to this operation, algorithm 2 lines 4 through 18, is to delete the associated array of this note. It is actually not possible to delete a vector as long as you are holding a reference to it, nor is there a method that clears all its memory. The

problem is that a C++ vector for optimization purposes retains its allocated capacity even though you delete the elements stored in the vector. The way out in my implementation, was to simply replace it with a vector of size zero. A vector is never created with more memory than needed, which is why this approach is an efficient way to clear the space used by the associated array of this node.

The **find**-operation implemented for this node uses the binary rank structure of its left node. The return value is found by subtracting the value $rank_j$ from $rank_i$. These integers are the result of calling the rank operation with j and $i-1$, respectively. The two new i , j pairs are set as is depicted in algorithm 2, line 26.

Even though this implementation forces the user to choose a constant word size on compile time, the wordsize is still $\log n$ for all intends and purposes. Therefore, the arguments for the space usage of the algorithm we made in Section 3.1 still apply. The only place that it should have an impact is on the constant hidden by the ”bigOh“ notation.

As we saw in Section 3.1 each level uses $O(n \log n)$ words of memory summing to a total of $O(n)$. The **rank**-operation is still implemented by only two array lookups and a single call to a **POPCNT**-operation. All of which is done in constant time. This means k queries are answered in time $O(k \log n + n \log k)$, theoretically that is.

Results of Experiments

I will now present the results of running the experiments on this implementation. My expectations are that both the space and query time measures will be consistent with the theoretical analysis performed in Section 3.1.

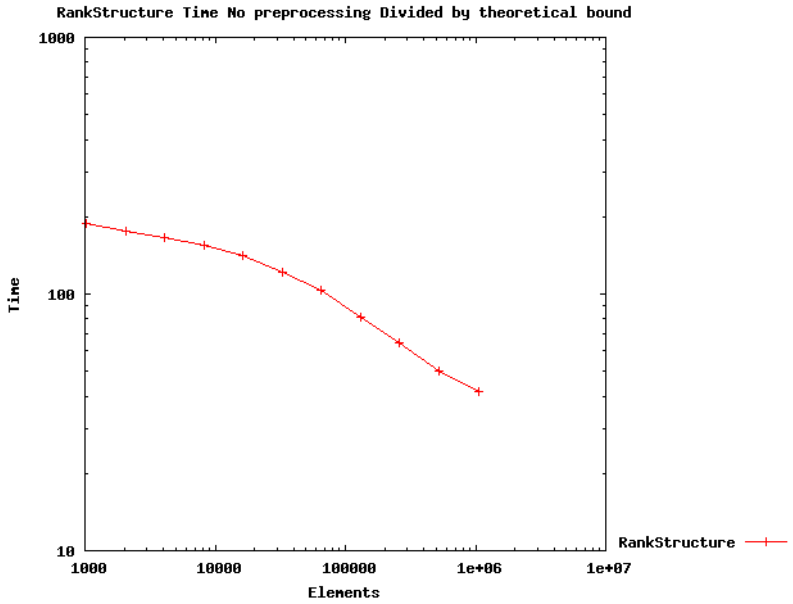

Figure 4.7: *Time of 10000 random queries to the binary rank tree structure. Divided by theoretical space bound which is* $10000 \log n + n \log 10000$ *for the fully preprocessed structure.*

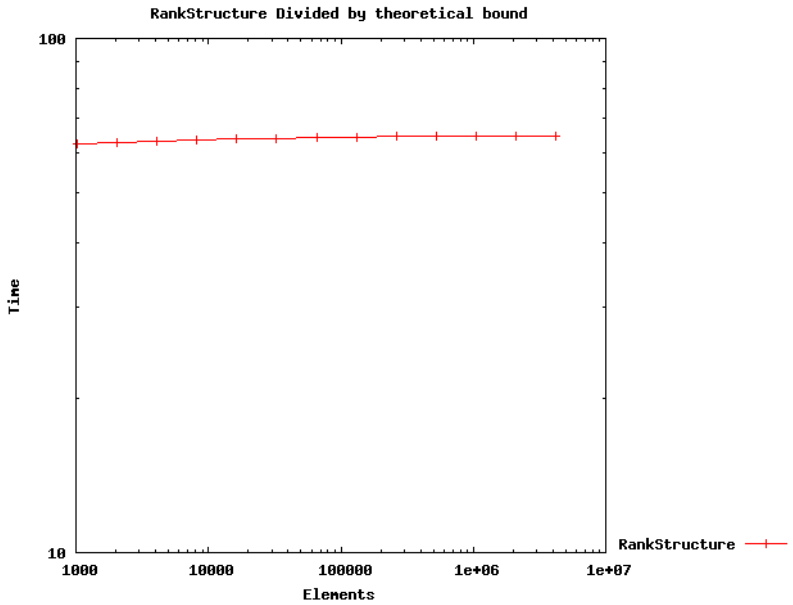

Figure 4.8: *Space usage of a fully preprocessed binary rank tree structure. Divided by theoretical space usage which is n.*

The plot shown in Figure 4.7 and Figure 4.8 shows the outcome of running experiment 1 and experiment 2 on the implementation. As we can see both sets of points seem to converge to a constant. Notice, however, that the constant hidden in

the analysis of the space analysis turned out to be rather large.

4.7 Comparison of Implementations

In the previous three sections it was shown that the implementations more or less performed as would be expected with regards to the theoretical analysis. In this section I will present further plots and compare the implementations on the basis of these. I will start by summing up my expectations to how the comparison will turn out.

Most likely the fractional cascading structure will turn out to have the best query time. I base this on the fact that the number of operations performed during a query is relatively low compared to the binary rank data structure. The binary search data structure is likely to be a lot slower than the others with respect to query time simply because of the excess of operations spent on the range counting.

With regards to space I would predict the binary range structure to win by a great distance. We saw in the previous section that the space used by this structure was in fact linear. Simultaneously, the space used by the fractional cascading and range structures were shown to converge to $O(n \log n)$.

I will now present a plot showing the actual time spent solving 10000 queries, that is running experiment 1. Notice that all four implementations are plotted together for easier immediate comparison.

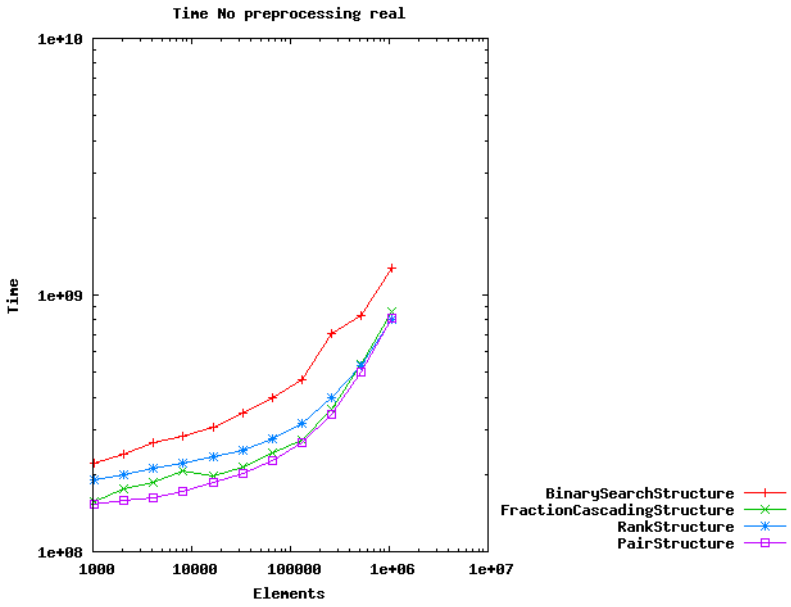

Figure 4.9: *Real times measured for experiment 1.*

The plot depicted in Figure 4.9 does seem to cooperate the prediction made earlier. The binary search structure is way slower than the others. The two flavors of fractional cascading structures are almost equal with a slight advantage to the

pair pointer strategy. This difference could possibly be explained by the random nature of the queries and input structure. What surprises me a bit is that the binary rank tree structure appears to be gaining on them as the number of elements grows. Actually they all three seem to meet in the last point.

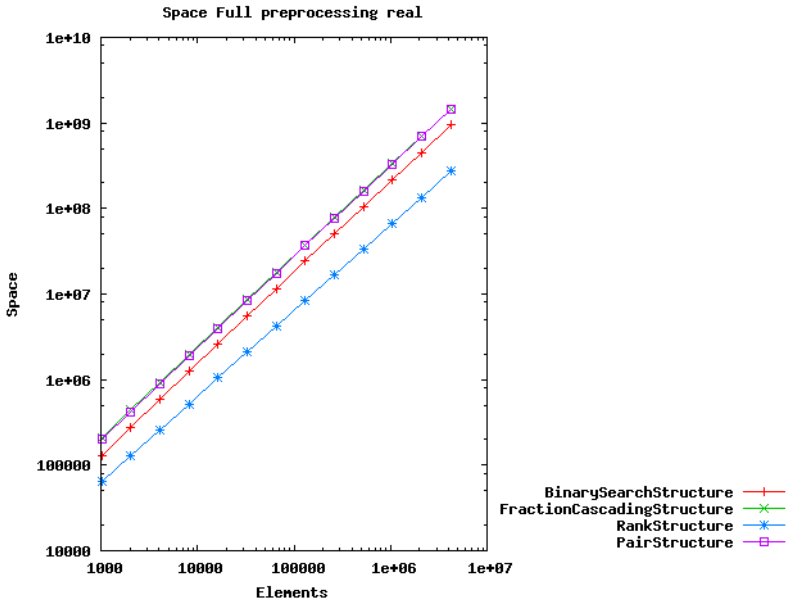

Figure 4.10: *Space measured for fully preprocessed data structures.*

The plot in Figure 4.10 shows the space used by the four data structures, when they are fully preprocessed. Notice that the actual values in the input array are of absolutely no importance here. Actually randomness plays no part at all.

All normalized space plots we examined in the previous sections had a strange downhill slope. Here we see why. Apparently, the space usage of the fractional cascading structures is completely dominated by constants. Aside from that the results immediately read of the plots is that the binary rank tree structures is by far the most space efficient. The Furthermore, the two fractional cascading structures use virtually the same amount of space. The binary search structure is naturally a bit more space efficient than the two fractional cascading structures since it only stores two integers per element on each level of the tree whereas the other store three.

If I had to choose between the four implementation, I would use the binary rank tree structure. Even though we would lose a bit on query time, the space efficiency gained fully makes up for it.

Chapter 5

Conclusion

The purpose of this thesis was to investigate the current state of the range median problem. I will now briefly summarize what was covered in the process. The thesis started out looking at some classic simple range problems. Specifically the current optimal bounds for these. For range queries involving group and semi group optimal solutions were known. Especially, this was the case for the first semi interesting problem we encountered, namely the range min/max problem. Similar in nature to the these problems neither the range median nor the more general range selection problem can be solved using conventional range problem techniques. In Section 2.4 we got acquainted with the historical solutions proposed in the very short history of the range median and range selection problems. This brought us to the current state of the art solutions. This included a data structure that solved that was optimal in the very restrictive pointer machine mode and a static RAM data structure achieving sublogarithmic query time with linear space. Furthermore, a rather complicated solution achieving constant query time for average input was presented. First and foremost however, we were equipped with a set problems well suited for practical implementation. The very simple tree based structures of Section 3.1. Chapter 5 was devoted to these implementations and the experiment performed on these.

The speed and the space usage of the implementations presented in this thesis clearly proves that the range median problem actually can be solved. Even for realistic input sizes. Combined with a very fast binary rank tree the linear space tree solution is actually both very space and time efficient. I highly doubt the $O(\log n / \log \log n)$ would be able to outperform the solutions that we experimented with in Chapter 5. I suspect that the constants hidden in both the time and space bound will overshadow the small theoretical advantage over for instance the binary rank structure.

It is clear that a lot of work remains in this area. Both in terms of lowering upper bounds and raising lower bounds. The next natural step would be to consider the problems on higher dimensions. That is finding the median of an totally ordered set of points within a ball in hyperspace. A next step would be to consider other models such as the IO-model. Furthermore, the relationship between the range selection and the range median problem could be interesting to investigate. The current so-

lutions seem to solve the range selection problem just as well as the range median problem even though it is more general. On the contrary, it is evident from this thesis that introducing the range selection problem into the equation greatly simplified the solutions for the range median problem.

Bibliography

[BFC04] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.

[BFP+72] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Linear time bounds for median computations. In *STOC*, pages 119–124. ACM, 1972.

[BGJS10] Gerth Stølting Brodal, Beat Gfeller, Allan Grønlund Jørgensen, and Peter Sanders. Towards optimal range median. *Theoretical Computer Science, Special issue of ICALP'09*, 2010.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[GJLT10] Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. *ICALP*, 2010.

[HPM08] Sariel Har-Peled and S. Muthukrishnan. Range medians. In *ESA*, pages 503–514, 2008.

[KMS05] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

[OS07] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*. SIAM, 2007.

[PD06] Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

[Pet08] Holger Petersen. Improved bounds for range mode and range median queries. In Viliam Geffert, Juhani Karhumäki, Alberto Bertoni, Bart Preneel, Pavol Návrat, and Mária Bieliková, editors, *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 418–423. Springer, 2008.

[PG09] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2009.

[Yao82] Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In *STOC*, pages 128–136. ACM, 1982.

[Yao85] Andrew Chi-Chih Yao. On the complexity of maintaining partial sums. *SIAM J. Comput.*, 14(2):277–288, 1985.