Data structures for orthogonal intersection searching and other problems

Ph.D. thesis

by
Christian Worm Mortensen

May 26, 2006

Preface

The work presented in this thesis has been performed while I was enrolled as a ph.d. student at the IT University of Copenhagen in the so-called 4-year program. I was enrolled in February 2002. In summer 2003 my ph.d. advisors Stephen Alstrup and Theis Rauhe founded a company called Octoshape and my advisors became Lars Birkedal, Rasmus Pagh and Anna Östlin. I received my masters degree in January 2004. From February 2004 to July 2004 I was visiting Max-Planck-Institut für Informatik in Saarbrücken. From August 2004 to August 2005 I was on full time leave from the IT University working in Octoshape. From August 2005 to December 2005 I worked part time in Octoshape and part time at the IT University to finish the present thesis.

I wish to thank my original ph.d. advisors Stephen Alstrup and Theis Rauhe for introducing me to the area of data structures from a research perspective, for encouraging me to apply for a ph.d. grant, for taking me as their ph.d. student and for giving me inspiration to a large part of my work. I will like to thank Lars Birkedal, Rasmus Pagh and Anna Östlin for taking me as their student and for much inspiration and help. I wish to thank Arne Andersson for reading and commenting on a preliminary version of this thesis in summer 2004 which allowed me to take a one-year leave from the IT University.

I will like to thank Philip Bille and Inge Li Görtz for many useful discussions during my work. Also, I wish to thank my office mates including Andrzej Wąsowski and Noah Torp-Smith and the members of the former Theory department at the IT University to make my life on the IT University much more pleasant.

I wish to thank Gerth Stølting Brodal for suggesting me to stay at the Max-Planck-Institut in my half year stay abroad. I also wish to thank the many people making my life there much more enjoyable including Ingmar Weber and Dörthe Schwarts. I also wish to thank Ingmar Weber and Holger Bast for our work on a joint paper outside the scope of this thesis.

I wish to thank my coauthors Joseph JaJa, Qingmin Shi, Seth Pettie, Rasmus Pagh and Gerth Stølting Brodal for our joint work.

I wish to thank Rasmus Pagh for reading parts of this thesis and giving usefull suggestions.

Finally, I wish to thank the thesis committee members, Henrik Reif Andersen, Lars Arge and Mark de Berg for evaluating the present thesis.

Abstract

This thesis presents new data structures for several orthogonal intersection searching problems. It also shows a lower bound for implicit priority queues supporting the decrease-key operation.

The orthogonal intersection searching problems are all studied from an upper-bound perspective and in the RAM model of computation. The 2-dimensional dynamic orthogonal range reporting problem is to maintain, under insertions and deletions, a set of points in the plane such that given a query rectangle with sides parallel to the coordinate axes, the set of points inside the rectangle can be reported. The problem naturally generalizes to any dimension larger than or equal to one. Similarly, the 2-dimensional dynamic segment reporting problem is to maintain, again under insertions and deletions, a set of horizontal line segments in the plane such that given a vertical segment the set of horizontal segments intersecting it, can be reported. Both problems can also be considered in the static case where updates are not allowed but where the objects (points or segments) are given in advance for preprocessing.

In Chapter 2, we give a solution to the 2-dimensional dynamic orthogonal range reporting problem and to the 2-dimensional dynamic orthogonal segment intersection reporting problem. Both these solutions support updates in time $O(\log n)$ and queries in time $O(\log n + k)$ where n is the number of objects (points or vertical line segments) and k is the size of the output. The structures use space $O(n \log n / \log\log n)$. By reduction to the dictionary problem the time usage of the structures is in some sence optimal if coordinates of objects can only be accessed by a comparison operation.

In Chapter 3, the orthogonal range reporting structure of Chapter 2 is improved to allow updates in time $O(\log^{\delta} n)$ for some (not any) constant $\delta < 1$ and queries in time $O(\log n/\log\log n + k)$. Here we need to assume a slightly different way of specifying coordinates to bypass the $\Omega(\log n)$ lower bound that exists if we can only compare coordinates. The space usage of the structure is $O(n \log^{\delta} n)$. The time usage of the structure is in some sense optimal in the cell probe model of computation by a lower bound of Alstrup et. al.. The structure is extended to arbitrary constant dimension $d \geq 2$ where it has update time $O(\log^{d+\delta-2} n)$, query time $O((\log n/\log\log n)^{d-1} + k)$ and space usage $O(n \log^{d+\delta-2} n)$.

In Chapter 4, we consider the one-dimensional variant of the dynamic orthogonal range reporting problem. Here we assume each coordinate is an integer which fits into a computer word of w bits. We show a whole range of tradeoffs between update time and query time where one of them is that we for any constant $\epsilon > 0$ can support updates in amortized time $O(w^{\epsilon})$ with high probability and queries in worst case constant time. We also give a structure for dynamic approximate range counting in one dimension. All the structures of Chapter 4 use linear space.

In Chapter 5, we consider two problems. The first is the static version of dominance reporting in any constant dimension $d \geq 3$. This problem is a variant of orthogonal range reporting where the kind of queries that can be asked are restricted. We show that for this problem it is possible to answer queries in time $O((\log n/\log\log n)^{d-2} + k)$ using space only $O(n(\log n/\log\log n)^{d-3})$. The next problem we consider is a static variant of range counting in any constant dimension $d \geq 2$ where the goal is not to report the points inside the query but to count the number of points.

We show that for this problem it is possible to answer queries in time $O((\log n / \log\log n)^{d-1})$ using space $O(n(\log n/\log\log n)^{d-2})$.

In Chapter 6, we consider a generalized variant of orthogonal range reporting called *colored* range reporting. Here we assume that objects (points or horizontal line segments) have a color and the objective is not to report objects but to report the different colors of the objects. We give a structure for the dynamic 1-dimensional version of this problem where the n points are supposed to be in an array of size n. The structure uses linear space and supports updates in time $O(\log\log n)$ with high probability and queries in worst case time $O(\log\log n + k)$. We use this result together with partial persistence to obtain improved upper bounds for the 2-dimensional static colored variants of both orthogonal range reporting and orthogonal segment reporting.

In Chapter 7, we give lower bounds for an implicit heap supporting the decrease-key operation. Our major result is a lower bound stating that unless delete is allowed to take very long time, then the amortized cost of decrease-key must be $\Omega(\log^* n)$. The lower bound is obtained by reducing the decrease-key operation to the *Absent Minded Usher's Problem* which we also introduce in this thesis.

Contents

Chapter 1

Introduction

This thesis consists of the present introduction in Chapter 1 together with revised versions of the following 6 papers appearing as Chapter 2 to 7. The chapters 2 to 6 contain upper bounds for different orthogonal intersection searching problems. Chapter 7 contains lower bounds for implicit heaps supporting the decrease-key operation.

Chapter 2 Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. Appeared in Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA'03) [79].

Chapter 3 Fully-dynamic orthogonal range reporting on RAM. To appear in SIAM Journal of Computing (SICOMP).

Chapter 4 Dynamic range reporting in one dimension. Joint work with Gerth Stølting Brodal and Rasmus Pagh. An improved version of this paper coauthored with Mihai Pătraşcu appeared in Proceeding of the 37th ACM Symposium on Theory of Computing (STOC'05) [81].

Chapter 5 Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting. Joint work with Joseph JaJa and Qingmin Shi. Appeared in Proceedings of The 15th International Symposium on Algorithms and Computation, (ISAAC'2004) [64].

Chapter 6 New upper bounds for some colored orthogonal intersection reporting problems. Published as a technical report [80].

Chapter 7 Implicit Priority Queues with Decrease-Key. Joint work with Seth Pettie. Appeared in Proceeding of the 19th Workshop on Algorithms and Data Structures (WADS'2005) [82]. One part of the original version of this paper [82] contained a bug and this part is not included in the version presented here.

Section 4.3 about *Data stream perfect hashing* should *not* be included in the judgment of the present thesis and is only included for completeness.

Chapter 2 contains material which is somewhat overlapping with Chapter 3. In particular, Lemma 2.1 can be obtained by plugging Lemma 2.2 into Theorem 3.1. For this reason the reader may want to skip the proof of Lemma 2.1. Further, the range reporting structure of Chapter 2 is strictly improved in Chapter 3. Chapter 2 is included for the following reasons. 1) It continues in a natural way an existing line of papers (see Section 1.5.3) 2) It contains a structure for segment reporting not included in Chapter 3 and 3) It is a kind of warm-up to what we do in Chapter 3.

Chapter outline In Section 1.1 we define what an intersection searching problem is and define the particular intersection searching problems we study in Chapter 2 to 6. In Section 1.2 we discuss output sensitiveness which is an import issue in some intersection searching problems. In section 1.3 we describe different computational models and in Section 1.4 we describe different *ordering variants*. In Section 1.5 we survey results on intersection searching and present the main results of Chapter 2 to Chapter 6. In Section 1.6 we present the main results of Chapter 7. Finally Section 1.7 has a conclusion.

1.1 Intersection searching

Let S be a set of points in the plane. The 2-dimensional *orthogonal range counting* (or in this thesis just *range counting*) problem is for a given query rectangle q with sides parallel to the coordinate axes to find $|S \cap q|$. In the 2-dimensional *orthogonal range reporting* (or in this thesis just *range reporting*) problem we want instead to find the set $S \cap q$. The obvious application of these two problems is in databases. Consider a database of persons where each person has an associated age and salary. We can represent each record of this database as a point in the plane. A range reporting query can then ask for all persons with an age between 30 and 40 years and a salary between 2000€ and 2500€ and a range counting query can ask how many such persons there is. See Willard [111] for additional database applications of range reporting.

The range counting and reporting problems are two examples of *intersection searching* problems. In general, intersection searching is for a given set of geometric objects S and a given query object q to say "something" about the objects from S intersecting q. We make this precise in the next paragraph. A myriad of different intersection searching problems has been studied in the literature, see the extensive survey by Agarwal and Erickson [1]. As one example, which most likely has not been studied, S may be a set of balls in high dimensional space each having an assigned integer. Given a query line the objective could be to find the ball which intersects the query line such that the assigned integer is minimized.

Formal definitions We now formally define what an intersection searching problem is and we define most of the intersection searching problems studied in this thesis.

An intersection searching problem is defined by a triple $(Q, D, (G, +))$. Here Q is the possible set of queries, D is the possible set of objects and $(G, +)$ is a commutative semigroup with neutral element. Given 1) a finite set $S \subseteq D$ where each object $e \in S$ has a *weight* $w(e) \in G$ and 2) a query $q \in Q$ we must then find the semigroup sum $\sum_{e \in S \mid e \cap q \neq \emptyset} w(e)$. Since G has a neutral element this sum is always defined. In this chapter we will use n for the number of objects in S.

We will now describe some possible definitions of $(G, +)$. In the *counting* variant G is the set of non-negative integers, the semigroup sum is addition on integers and each element $e \in S$ has weight $w(e) = 1$. That is, the goal is to count how many objects from S intersect q. In the *emptiness* variant $G = \{\text{true}, \text{false}\}$, $+$ is logical "or", and "false" is neutral element. Each element $e \in S$ has weight $w(e) = \text{true}$. That is, a query must check if S contains any element intersecting q. In the *reporting* variant G is the set of possible subsets of D, the semigroup sum is set union and each element $e \in S$ has weight $w(e) = \{e\}$. That is, the goal is to find (or report) the objects from S which intersect q. The *colored*[1] *reporting* variant is as the non-colored reporting variant except that each element $e \in S$ has a color $\mathit{color}(e)$ and $w(e) = \{\mathit{color}(e)\}$. That is, the goal in a query is to find the colors represented among the objects from S which

[1]Some papers use *generalized* or *categorial* instead of colored

intersect q. In the *semigroup* variant we make no particular restriction on $(G,+)$. That is, the solution should work for any semigroup (when we study upper bounds).

We will now describe some possible definitions of Q and D. In the d-dimensional *orthogonal range* (or in this thesis just *range*) variant D is the set of d-dimensional points and Q is the set of d-dimensional rectangles (or ranges) with sides parallel to the coordinates axes. For $d = 2$ we will describe a query by the quadruple (x_1, x_2, y_2, y_2) with $x_1 \leq x_2$ and $y_1 \leq y_2$ representing the rectangle with the two corners (x_1, y_1) and (x_2, y_2). A d-dimensional point $q = (q_1, \ldots, q_d)$ is said to *dominate* a d-dimensional point $p = (p_1, \ldots, p_d)$ if $p_1 \leq q_1, \ldots p_d \leq q_d$. Inspired by this, in the d-dimensional *dominance* variant, D is the set of d-dimensional points and Q is the set of infinite rectangles which for some point q can be written as $\{p \mid q \text{ dominates } p\}$ [2]. We will describe a query by the point q so the points we consider during a query are the ones dominated by q. In the *3-sided* variant, D is the set of 2-dimensional points and Q can be described by a triple (x_1, x_2, y_1) representing the rectangle $\{(x, y) \mid x_1 \leq x \leq x_2 \wedge y \leq y_1\}$. It follows, that the 3-sided variant is more general than the 2-dimensional dominance variant but less general than the 2-dimensional range variant. In the *orthogonal segment intersection* variant (or in this thesis just *segment* variant), D is the set of horizontal line segments in the plane and Q is the set of vertical line segments in the plane. We will describe an element from Q by a triple (x, y_1, y_2) where $y_1 \leq y_2$ representing the vertical segment between (x, y_1) and (x, y_2). The d-dimensional *orthogonal point enclosure* (or in this thesis just *point enclosure*) variant is as the range variant except that Q and D are interchanged. That is, D is the set of d-dimensional rectangles with sides parallel to the coordinate axes and Q is the set of d-dimensional points.

We distinguish between the static and dynamic variant of intersection searching. In the *static* variant S is given once and for all for preprocessing. After the preprocessing, we must answer queries from Q in an online fashion. In the *dynamic* (sometimes called *fully-dynamic*) variant, S is initially empty. We are then given an online sequence of insertions to S, deletions from S and queries from Q which we must answer.

We have described five restrictions on $(G,+)$, five restrictions on D and Q and the dynamic and static variant. This gives $5 \cdot 5 \cdot 2 = 50$ variants where some are special cases of others. Most[3] intersection searching problems studied in this thesis are among the problems described here. Further, most of the problems can be studied in different dimensions, in different computational models (Section 1.3) and in different ordering variants (Section 1.4).

1.2 Output sensitiveness

We define the *query time* of an intersection searching structure as the time it takes to answer a query. In the reporting (resp. colored reporting) variants of intersection searching the query time is normally stated as a function $f(n, k)$ where n is the number of objects in the structure and k is the size of the output (elements from S resp. colors of elements). We call $f(n, 1)$ the *search time* of the structure. If f is depend on k we say the algorithm is *output sensitive*. Since it is usually assumed that the query algorithm produces its output object by object we usually have $f(n, k) \geq k$ and since k can be as large as n it usually does not make sense to study reporting algorithms which are not output sensitive.

Often f has the form $f(n, k) = g(n) + O(k)$ for some function g and if not otherwise noted we will assume this is the case. In the literature it is not uncommon to have $f(n, k) = g(n) + O(k \log \log n)$ [26, 6, 58, 21]. In [26, 6] the $\log \log n$ factor comes from the desire of saving space: Points are kept in a universe with reduced size and have to be converted to points in

[2]Some papers including Chapter 5 use instead p dominates q

[3]The exception being the approximate range counting structure of Chapter 5.

the full universe before they are reported. We will discuss these structures in greater detail in Section 1.5.11.

Sometimes it is possible to exploit in a non-trivial way the fact, that if the output is large then we are also allowed to spend additional time to find the points to report. This fact was used by Chazelle in his filtering search [25] and this is also an important idea used in this thesis; in particular the construction mentioned in Section 1.5.1.

A consequence of this is also that the the search time of the reporting variant is typically smaller than the query time of other variants such as the counting variant and this is even justified by lower bounds (see Section 1.5.8 and Section 1.5.10). Intuitively the reason for this is the following. In the non-reporting variants, even if there are many points placed in an evil way, we have to find some aggregate information about the points in a short time when answering a query. In the reporting variant, we have long time to answer a query covering many points because of output sensitiveness.

We conclude this section with one technical remark. In the reporting variant, we may be in a situation where we do not want all objects of a query answer. Instead, we may want to stop the algorithm while it is producing the answer. How long time does the algorithm use if we stop it after it has produced $l \leq k$ objects? (the algorithm does not know in advance when we will stop it.) Most algorithms in the literature, including all the algorithms in this thesis, do not claim that this takes asymptomatically shorter time than letting the algorithm find all k objects and then throw the last $k - l$ objects away. But most algorithms which claim to spend time $f(n, k)$, including the algorithms in this thesis, can be modified to run in time $f(n, l)$ instead. As a special case, if we take $l = 1$ we have a solution for the emptiness problem.

1.3 Computational models

In the literature, intersection searching has been studied in many different computational models. We will briefly survey some of these models. Also, we will mention the implicit model used in Chapter 7 on implicit heaps. This model has not (yet) been studied in connection with range searching in dimensions larger than 1. All the new intersection searching algorithms in this thesis are considered in the RAM model of computation (Section 1.3.2).

1.3.1 Pointer machine model

A data structure in the *pointer machine* model consists of a set of *nodes*. Each node has a constant number of *fields* and a field may either contain a pointer to a node or some other item such as a point. There is a constant number of *access pointers* which point to nodes of the structure and which are available to operations. There exists many variants of the pointer machine model. These variants mainly differ in exactly which kind of information can be stored in the fields and how this information can be manipulated. However, all the variants share the following central property: It is not possible to perform any pointer arithmetic so in particular arrays are not possible. When implementing intersection searching on a pointer machine it is assumed, if not otherwise noted, that coordinates of objects can only be accessed by a unit-cost comparison operation. That is, the comparison order variant as defined in Section 1.4.1 is assumed.

1.3.2 RAM model

The pointer machine has some nice theoretical properties but it has not much to do with real hardware. The *Random Access Machine* model, or just *RAM*, was designed to capture the main properties of a typical computer. Historically, this model has also been defined in various ways, see e.g. Hagerup [59]. We will only consider the word-RAM model as defined in [59]. In this model a *word* is an integer of w bits where w is a parameter of the model called the *word size*. The memory is an infinite array of *cells* where each cell can contain a word. The instruction set includes indirect addressing and operations such as addition, subtraction, bitwise shifting, bitwise and, bitwise or and bitwise xor, multiplication and division. All operations are assumed to have *unit cost*, that is to take equally long (constant) time. The space usage of an algorithm is the largest index of a cell accessed.

We will now describe a standard trick called *word parallelism* which we will use again and again. In general the idea in word parallelism is to pack several elements into a single word and then handle these elements in parallel. As one of the simplest examples of this, we can represent a subset of $[1 \ldots w]$ as a bitvector in a single word. Union and intersection of such subsets can then be performed in constant time by the bitwise or and the bitwise and instructions respectively. However, more often than not, we do not have such instructions in the instructions set which can perform exactly the kind of manipulations we need. In this case, we often use the following trick which really exploits the power of the RAM model. Suppose we are designing a structure D with N elements and that inside it we need several instances of a structure with at most n elements where n is much smaller than N. Using some kind of global rebuilding (see Overmars [87]) we can with D build an array with $O(N)$ elements which we will call a *global lookup table*. In each instance of the structure with at most n elements, we may be able to pack multiple elements into a single value in the range $[1 \ldots N]$. Looking up this value in the global lookup table basically gives us our own private instruction doing almost whatever we like to do.

1.3.3 Cell probe model

The cell probe model is stronger than the RAM model and is useful for proving lower bounds. Essentially, the cell problem model is identical to the RAM model except that in the cell probe model each operation can remember the content of all cells probed during its execution and perform any computation on these for free; the only thing that takes time is accessing the memory.

1.3.4 Arithmetic model

Like the cell probe model, the arithmetic model is used for proving lower bounds. In this model, we have an array of cells $A[1 \ldots m]$ where each cell can contain an element from G and where m is the space usage. To answer a query, the algorithm must select $S \subseteq [1 \ldots m]$ and for each $i \in S$ select a positive integer $w(i)$ such that the answer to the query is the semigroup sum $\sum_{i \in S} w(i) \cdot A[i]$. The query time is then $|S|$. If the structure supports updates the algorithm is in one operation allowed to multiply the content of a cell with a given non-negative integer or to set one cell to the semigroup sum of two other cells. Further, on insertions we assume $A[1]$ is set to the semigroup value for the new point. We do not account for any other time or space used by the algorithm: we can assume the algorithm has an oracle at its disposal which can answer any question in zero time.

1.3.5 I/O model

The main criticism of the RAM model is the unit cost assumption. One argument against this assumption is, that in reality all memory accesses do not take equally long time because of the existence of caches and virtual memory. A model which tries to capture this fact is the I/O model of Aggarwal and Vitter [3]. This model is parametrized over over three parameters B,

M and N where $B \leq M \leq N$. In the I/O model, there exists two kinds of memory: primary (fast) memory and secondary (slow) memory. Both primary and secondary memory consists of an array of blocks. Each block can contain B records where a record is similar to a field in the pointer machine model. The input is assumed to take up N/B blocks which are initially placed in secondary memory. The primary memory consists of M/B blocks and it is possible to transfer a block between the secondary and the primary memory using one I/O. The only way to access the secondary memory is through these transfers. Normally, the time usage of an algorithm is measured as the number of I/Os used and all computation in the primary memory is considered free. In this sense, the I/O model has similarities with the cell probe model. As in the RAM model, the space usage (in blocks) is defined as the highest index of a block accessed in secondary memory.

When studying intersection reporting in the I/O model of computation, we like the query time to have the form $f(N, k) = g(N) + O(k/B)$ when k points are reported (f and g also depend on M and B). That is, we are only supposed to make $O(1/B)$ I/Os for each element reported.

1.3.6 Implicit model

In the implicit model, a data structure with n elements must consists of a permutation of the elements placed in the first n entries of an array and nothing else. If the structure is dynamic, the algorithm, is allowed to permute the elements on updates. Different restrictions can be placed on the operations such as how much storage they can use during the execution of an operation etc.

1.4 Ordering variants

In all the intersection searching algorithms defined in Section 1.1 each element of D and Q can be described by a constant number of points which in turn can be described by one coordinate for each dimension. We now describe three different ways to order coordinates each called an *ordering variant*. We then discuss how these ordering variants relate on the RAM model of computation.

1.4.1 Definitions

In the *comparison order* variant coordinates can only be examined by a unit-cost comparison operation. A typical way to phrase this is to say that coordinates are elements in $\mathbb{R}$. When using comparison order in the RAM model we assume the description of a coordinate fits into a single word but that the value of such a word only makes sense to the special comparison operation. The comparison order variant is probably the most studied variant in the literature. We observe, that since a structure for intersection searching can also be used as a dictionary, there is a trivial lower bound on $\Omega(\log n)$ on the query time in this variant.

In the *m-order* variant (non-standard terminology) coordinates are assumed to have values in $[1 \ldots m]$. Here we assume $n \leq m$ and in the RAM model we further assume $m \leq 2^w$ where w is the word size. The *word order variant* (non-standard terminology) is an abbreviation for the 2^w-order variant.

The *list order* variant is suggested in Chapter 3 of the present thesis. In this variant, we assume, that each coordinate used on a given axis is represented by an element in a doubly linked list. The order the elements have in this list determines the order of the corresponding coordinates.

Note, that for a $d > 1$ dimensional intersection searching problem, it is possible to use different ordering variants on different axes. This is heavily used in Chapter 3.

We now introduce some non-standard terminology which we will only use in the present chapter. As mentioned, in this thesis, the objects of S can be described by a constant number of points. If, in the m-order variant or the list order variant, each possible value for a given axis is used by at least one such point, that is, there is a surjective mapping from used points to the axis, we will say the axis is *surjective*. Further, if for any p and q describing different elements from S we have that p and q do not share a coordinate for a given axis, that is, there is an injective mapping from points to the axis, we say that axis is *injective*. We will say an axis is *bijective* if it is both surjective and injective. A problem is said to be in *rank space* if all axes are in the m-order variant and are bijective (m can be different for different axes). The term rank space is standard.

1.4.2 Relation between ordering variants on RAM

In this section we discuss the different ordering models and how they relate on the RAM model of computation. For this discussion we first need to consider the predecessor problem and the problem of maintaining order in a list.

The predecessor problem

A fundamental problem on the RAM is the predecessor problem. Given a set S of words and a query word q the problem is to identify the largest word $p \in S$ such that $p \leq q$ where p and q are interpreted as integers. In the equivalent successor problem we must instead find the smallest word $p \in S$ such that $p \geq q$. This problem has been intensively studied both in the static variant where S is given once and for all for preprocessing and in the dynamic variant where insertions to S, deletions from S and queries are given as an online sequence.

Theorem 1.1. *(Beame and Fich [15]) If the space usage is bounded by $n^{O(1)}$ the query time for the static predecessor problem is* $\Omega(\min(\sqrt{\log n/\log\log n}, \log w/\log\log w))$

It should be noted that Theorem 1.1 only shows that for each sufficiently large w there *exists* an n such that the theorem holds. For example, the theorem does not hold if $n = \Omega(2^w)$ as the problem becomes trivial in this case. The lower bound of the theorem actually holds in the cell probe model of computation. The proof of [15] only holds for deterministic query algorithms but the bound also holds for randomized Monte Carlo queries [97]. Beame and Fich also showed matching upper bounds using super linear space. Andersson and Thorup transformed these upper bounds into bounds for the dynamic predecessor problem with linear space usage.

Theorem 1.2. *(Andersson and Thorup [11]) There exists a deterministic data structure for the dynamic predecessor problem using space $O(n)$ such that updates and queries take worst-case time $O(\min(\sqrt{\log n/\log\log n}, \log\log n \log w/\log\log w)$.*

There exists a dynamic structure which uses time $O(\log w)$ for both updates and queries. For linear space usage, however, the structure needs hashing and thus randomization.

Theorem 1.3. *([105, 39, 107]) There exists a randomized data structure to the predecessor problem which uses space $O(n)$ such that updates and queries can be performed in time $O(\log w)$ with high probability.*

We will call the structure of Theorem 1.3 a *VEB*.

Maintaining order in a list

The problem of maintaining order in a list is as follows. We must maintain a doubly linked list under insertions and deletions of elements such that given two query elements we can determine which element is first in the list.

Theorem 1.4. *(Dietz and Sleator [37]) There exists a data structure for maintaining order in a list where every operation take worst case constant time.*

Discussion

In this section we discuss the different ordering variants on the RAM model of computation. The conclusion of the discussion is that on the RAM the list order variant is the most flexible variant in dimensions larger than or equal to two because it also gives good upper bounds for the other variants. The converse is not always true.

We first consider the static case. The list order variant is here identical to the m-order variant for an appropriate m. Further, an algorithm for the comparison or general m-order variant is also an algorithm for the list order variant with the same performance (if m is sufficiently big). Conversely, an algorithm for the list order variant can be converted to one for the comparison order variant if we add a term of $O(\log n)$ to the query time. Note that since a structure for the comparison order variant usually can be used as a dictionary $\log n$ is also a lower bound for the query time. By keeping a static predecessor structure for each axis, an algorithm for the list order variant can be converted to one for the m-order variant by adding the time of a predecessor query to the query time. Further, by the simple reduction of Section 1.5.5, such a term is required even in the maybe simplest intersection searching problem in two dimensions: The dominance reporting problem. In 1 dimension the situation is different: Here it is possible to get constant query time and linear space usage [5] for the range reporting problem in the word order model which according to Theorem 1.1 is not possible for the predecessor problem. The dynamic version of the problem is the topic of Chapter 4.

We next consider the dynamic case. In this case the list order variant combines the best of the other variants if seen from the perspective of the user of the data structure: Like in the comparison order variant it is always possible to insert a new coordinate between two existing coordinates when performing an insertion. This is not always possible in the m-order variant. On the other hand, the list order variant does not have a trivial lower bound of $\Omega(\log n)$ like the comparison order variant has. It follows from Theorem 1.4 that a structure for the comparison order variant can be turned into a structure for the list order variant with the same performance. Conversely, a structure for the list order variant can be turned into a structure for the comparison order variant if we add a term of $O(\log n)$ to the update and query time since such a term allows us to maintain and use a balanced search with the elements of a given axis. Like in the static case we can convert a structure for the list order variant to a structure for the m-order variant by maintaining a predecessor structure with universe size m. Actually, it is also possible to convert a structure for the m-order variant to the list order variant. The trick is to assign a number to each element in the list and then renumber list elements on insertions. The leads to a large overhead on the update time since we will normally have to renumber $\Omega(\log n)$ elements on each insertion. However, the trick is used in Chapter 3 and a similar trick is also used in [86].

1.5 Intersection searching

In this section we survey results on intersection searching focusing on techniques related to the ones used in this thesis. Along the way, we present the results of Chapter 2 to Chapter 6 and sketch how these results are obtained. We also sketch some possible ways to obtain new results.

1.5.1 From few to many points

Before we start the survey we present a construction introduced in this thesis. Chapter 2, 3 and to some extend Chapter 6 is based on this construction so it is very central to this thesis. The construction is described as Theorem 3.1 in Chapter 3 and also in a slightly less general form in Section 2.3 in Chapter 2.

Suppose we have a dynamic range reporting structure with an x-axis in the n-order variant and an injective y-axis in the u order variant where u may be much smaller than n. The construction then removes the injectivity of the y-axis by increasing the update and search time with a factor $\log \log n$. The construction is essentially a modified VEB of Theorem 1.3. Note that the original structure can contain only u points while the constructed structure can contain $u \cdot n$ points, hence the the name of this subsection.

1.5.2 The range tree

The historically first way to create a range reporting structure with poly-logarithmic search time and space usage $O(n \log^{O(1)} n)$ is the *range tree* [19, 16][4]. Suppose we have n points in the comparison order variant and that no two points share the same x or y coordinate. In a range tree, we span a complete binary tree T over one axis, say, the x-axis. We keep in each internal node $v \in T$ a list $v.L$ with the points stored in nodes descendant to v sorted according to their y-coordinate. Each point is stored one time on each level of T so the space usage becomes $O(n \log n)$. Suppose we are given a query rectangle. The x-coordinates of the rectangle uniquely determines a subset M of the nodes of T such that 1) a leaf of T is a descendant to a node in M if and only if it is within the query rectangle in the x-direction 2) no nodes in M have an ancestor relationship 3) $|M| = O(\log n)$. The answer to the query can be found by binary search in the lists of of the nodes of M so the search time of the algorithm becomes $O(\log^2 n)$.

Willard [108] observed that the range tree could be improved with *downpointers*. Consider the list $v.L$ in an internal node v of T. We augment each element of $e \in v.L$ with two pointers: One for each child of v which goes to the successor of e in the list of that child. We then only need to perform binary search in the the list of the root of T, the other positions we need to identify in lists of the nodes of M can be found by following the downpointers. The search time of the algorithm is reduced to $O(\log n)$. When not using downpointers Willard and Lueker observed that one could make T a BB(α) tree and get a dynamic version with update and query time $O(\log^2 n)$.

Chazelle and Guibas [30] generalized downpointers as well as other ideas into a data structuring technique called fractional cascading. Mehlhorn and Näher [75] made fractional cascading dynamic with amortized time bounds on updates. They used this to develop a data structure for the 2-dimensional range reporting problem in the comparison order variant with update and query time $O(\log n \log \log n)$ on a pointer machine again making T a BB(α) tree and using time $O(\log \log n)$ on each level T. They obtained the same bounds for the segment intersection problem. Dietz and Raman [36] removed amortization from the results of [75] on a RAM. As a part of this they used the trees of Willard and Lueker [113] instead of BB(α) trees for balancing T.

[4] See Chapter 3 for further bibliographic notes of the range tree

Imai and Asano [62] have developed a structure for the 2-dimensional range reporting problem in the comparison order variant with update and search time amortized $O(\log n)$ using space $O(n \log n)$. However, they only allowed updates in the form of either insertions or deletions; the same structure could not support both. They obtained the same bounds for the segment intersection variant.

While a standard range tree has degree 2 this thesis also uses range trees with degree $d > 2$. In each node we then need a range reporting structure for up to n points where the x-axis has length d. A query can be answered by performing a query in at most two such structures on each of the $O(\log n / \log d)$ levels of T.

1.5.3 Chapter 2: Dynamic segment intersection and range reporting in the comparison order variant

Chapter 2 continues the line of research outlined in Section 1.5.2 by bringing the update and search time down to worst case $O(\log n)$ for both range reporting and segment reporting. The binary degree of T is increased to $u = \log^{\epsilon} n$ for a constant $\epsilon > 0$ while we still use time $O(\log \log n)$ on each level of T. What we essentially need in order for this to work for range reporting is to have a range reporting structure for n points where the x-axis is in the u-order variant and the y-axis is injective and in the list order variant. We get this as follows. We use a q-heap of Fredman and Willard [52] to make a constant-time range reporting structure supporting u points. We plug this structure into the construction of Section 1.5.1 (with n in the construction set to $n/\log^{O(1)} n$) and interchanging the two axes) getting a structure for $n/\log^{O(1)} n$ points where the x-axis is in the u-order variant and the y-axis is in the $n/\log^{O(1)} n$-order variant. By grouping into blocks we turn the y-axis into the list order variant getting the structure we need. The range reporting result of Chapter 2 is improved in Chapter 3; we will discuss this in Section 1.5.7. Finally, in order to balance T we cannot use BB(α) trees because we want a structure with worst case time performance and we cannot use the structure of [113] because they have binary degree. Instead, we use WBB trees [109, 35, 14] to balance T.

1.5.4 Extending to higher dimensions

Consider the intersection searching structure supporting range queries where the elements of D are points (we do not specify any restrictions on the semigroup $(G, +)$). Suppose we have a structure for the d'-dimensional range problem for some constant $d' \geq 1$. The range tree then gives us a way to get a structure for the d-dimensional range problem for any constant $d \geq d'$ as follows. If $d = d'$ we already have a structure so suppose $d > d'$. We sort all points of S according to, say, their first coordinate and then we span a binary tree T over the points. Inside each node of T we recursively store the points descendant to T in a $d-1$-dimensional structure where we ignore the first coordinate of points. If the structure with dimension d' is dynamic the structure with dimension d can also be made dynamic by making T a BB[α] tree. The space, query and possibly update time of the d-dimensional structure becomes a factor $O(\log^{d-d'} n)$ larger than that of the d'-dimensional structure. If we do not use downpointers or a similar technique, we may increase the time usage further.

The method described works well for the reporting variant, the emptiness variant and the counting variant of our problems. It also works for the colored variant but we should mention that the performance is not very good since each color may be reported up to $\log n$ times for each recursive structure. That is, we get a factor $O(\log^{d-d'} n)$ on the query time, not just on the search time.

We now describe variants of the method just described. One variant is to use range trees with *slack parameter* [74, 103]. Here the idea is to store secondary structures only on levels of

T divisible by k where k is a parameter. In a query in T, we may then need to ask in up to $O((\log n)2^k/k)$ recursive structures so the decrease in update time gives a dramatic increase in query time. However, it also reduces the space usage.

Chazelle [28] and Alstrup, Brodal and Rauhe [6] used the following method which in a static setting gives an increase with a factor of $O(\log n/\log\log n)$ in the query time and $O(\log^{1+\epsilon} n)$ in the space usage for each dimension and for any constant $\epsilon > 0$. We call this method the *all-children-pair* method (non-standard terminology). The idea is to give T degree $\log^{\epsilon/2} n$. For each node $v \in T$ we then for each pair of children store a structure with dimension $d-1$ containing the points descendant to that node. This method is dynamized in Chapter 3 where we use a WBB tree. The increase in the update time is a factor $O(\log^{1+\epsilon} n)$ for each dimension.

This thesis proposes a third method to extend into higher dimensions which we will call the *universe extending* method. The method was introduced in [78]. A variant of the method has been independently described in the I/O-model of computation [56]. Unlike the other methods above it is not a black-box transformation, since it does not increase the dimension of an arbitrary structure. Suppose we have a structure for the d-dimensional range problem where the first $d' < d$ axes are in the bijective list order variant and the last $d - d'$ axes are in the $\log^\epsilon n$-order variant for some constant $\epsilon > 0$. Then we can take the $d' + 1$th coordinate and convert it to be in the list order variant by adding a factor $O(\log n/\log\log n)$ to the search time, the query time and the space usage, everything is in the worst case sense. Again, the idea is to give T degree $\log^\epsilon n$ and make it a WBB tree. So how do we get the structure with $d - d'$ extra axes in $\log^\epsilon n$-order variant to begin with? This varies. However, often it is possible to get the extra axes for free. For example, this is done in Chapter 5.

1.5.5 3-sided range reporting

Range reporting structures for 3-sided queries are well studied. McCreight [72] made the *priority search tree* which is an optimal dynamic (and optimal static) structure for 3-sided queries in the comparison order variant for the pointer machine. The structure has update and search time $O(\log n)$ and space usage $O(n)$. The structure consists of the contained points arranged in a clever way in a binary tree.

As mentioned, the implicit model of computation has not yet been studied in connection with range searching in dimensions larger than one. However, a variant of the priority search tree seems like a natural thing to study in this model. One might get inspiration from some of the recent work on implicit dictionaries (see e.g. [45]).

In the word order variant on the RAM, Willard [112] reduced the search and update time of priority search trees to $O(\log n/\log\log n)$ using q-heaps [52] by increasing the degree of the priority search tree to $O(\log^\epsilon n)$ a constant $\epsilon > 0$. In Chapter 3, we reduce the update time further to $O(\log^\delta n)$ for some (not any) constant $\delta < 1$ in the list order variant (see Section 1.5.7 for more details). The result of Willard is optimal in terms of time per operation: Let U be the update and S be the search time of a structure supporting dominance emptiness queries in 2 dimensions. For word size poly-logarithmic in n Alstrup, Husfeldt and Rauhe [8] showed that $S = \Omega(\log n/\log(U \log n))$ in the cell probe model of computation. Nothing in this lower bound, however, prevents a constant update time preserving the same query time.

The static version of the 3-sided range reporting problem is also well studied. It turns out that the upper bounds in this case naturally has the x-axis in the bijective list order variant and the y-axis in the comparison order variant. Fries et. al. [53] has given an upper bound with search time $O(\log\log n)$ and linear space usage $O(n)$ for the pointer machine. Earlier, Gabow, Bentley and Tarjan [54] showed that on the RAM the problem can be solved with search time $O(1)$ and linear space usage. If the x-axis is in the m-order variant for $m \geq n$ we get a solution by combining this structure with a predecessor structure for the set $[1 \dots m]$. Conversely, a simple reduction shows that the static version of the weaker dominance problem can be used to answer predecessor queries. Let P be a static set of words for which we want to find predecessor and assume w.l.o.g. that $-\infty, \infty \in P$. Let P' be the set of pairs $(p_1, p_2) \in P \times P$ such that p_1 is the predecessor of p_2 in P. For a given query q there will be exactly one point $(p_1, p_2) \in P'$ such that $p_1 \leq q$ and $q \leq p_2$ and p_1 and p_2 will be the predecessor and successor of q respectively. Further, the point (p_1, p_2) can be found by a dominance reporting query in P'. We conclude that on the RAM, the static version of the 3-sided reporting problem in the m-order variant is equivalent to the predecessor problem on the set $[1 \dots m]$ if $m \geq n$.

1.5.6 From 3-sided to 4-sided

Suppose we have n 2-dimensional points and that we have a structure which can answer 3-sided queries. We now describe a general way to make a structure supporting 4-sided queries from this. The idea has also been used by Chazelle [25] and Overmars [88]. We span a binary tree T over the y-axis. Each internal node of T covers an interval on the y-axis. Let P_v be the points in the leafs descendant to a node $v \in T$. We then keep in v two secondary structures supporting 3-sided queries containing the points of P_v. Given a query (x_1, x_2, y_1) the structure $v.T$ can report the points $(x, y) \in P_v$ such that $x_1 \leq x \leq x_2$ and $y \leq y_1$ and the structure $v.B$ can report the points such that $x_1 \leq x \leq x_2$ and $y \geq y_1$. Given a 4-sided query (x_1, x_2, y_1, y_2) we can answer the query by finding the nearest common ancestor (NCA) $v \in T$ of the leafs with y_1 and y_2 and then perform a 3-sided query (x_1, x_2, y_1) in $v'.B$ for the left child v' of v and the 3-sided query (x_1, x_2, y_2) in $v''.T$ for the right child v'' of v. The query time of this structure becomes the same as the query time for the 3-sided structure. The space usage is increased with a factor $O(\log n)$ because each point is stored in $O(\log n)$ secondary structures. Since a structure answering 3-sided queries typically uses linear space in both the dynamic and the static case a 4-sided structure based on this approach will use space $O(n \log n)$.

Like the range tree, we can also give this tree degree $d > 2$ and this was in fact done in [25]. In this case we need to store in each node a 4-sided range reporting structure for up to n points where the y-axis has length d. In a query, we then also need to query this structure in the NCA node.

1.5.7 Chapter 3: Dynamic range reporting in the list order variant

While the structure of Chapter 2 is optimal in terms of time per operation in the comparison order variant, it is not the case in list order or m-order variant. In Chapter 3, the orthogonal range reporting structure of Chapter 2 is improved to allow updates in time $O(\log^\delta n)$ for some (not any) constant $\delta < 1$, have search time $O(\log n / \log \log n)$ and space usage $O(n \log^\delta n)$ in the list order variant. As described in Section 1.5.5 this is optimal in terms of time per operation even if only dominance queries are supported. The segment intersection problem is not considered in Chapter 3.

By using the all-children-pair method of Section 1.5.4 the result is extended to any constant dimension $d \geq 2$ giving update time $O(\log^{d+\delta-2} n)$, search time $O((\log n / \log \log n)^{d-1})$ and space usage $O(n \log^{d+\delta-2} n)$. Also, a structure supporting 3-sided queries is given. This structure has update time $O(\log^\delta n)$, query time $O(\log n / \log \log n)$ and uses space $O(n)$ reducing the update time of Willard [112].

The structure supporting 3-sided queries is made as follows. As Willard [112] we make a priority search tree with degree $\log^\epsilon n$ for a constant $n > 0$. However, the tree can only contain approximately $u = 2^{\log^\gamma n}$ points for a constant $\gamma > 0$. Using word parallelism we can give the structure constant update time while the query time is kept on $O(\log u / \log \log n)$. Next, we build a 3-sided structure for n points by making a priority search tree with the structures just described.

The structure supporting 4-sided queries is made as follows. First, we take the structure supporting 3-sided queries for u points and extend it to support 4-sided queries using a construction similar to the one of Section 1.5.6. Again this structure has constant update time and the query time is $O(\log u)$. We then create a range tree with degree u in a similar way to the way we made a high-degree range tree in Chapter 2. The difference is that this time we use the structure just mentioned to plug into the construction of Section 1.5.1 instead of the structure based on q-heaps. In a range tree structure, we have to proceed all the way down the tree to answer a query and this gives a query time which is slightly too high. We fix this as follows. On a few levels of T we place, in the style of Section 1.5.6, the 3-sided structure we have developed. When we reach such a structure we do not need to proceed further down the tree; instead we can find the rest of the points needed to be reported from that subtree by making a single query in the 3-sided structure.

1.5.8 4-sided range reporting on the pointer machine

The space needed for static range reporting on a pointer machine is well understood. Chazelle[25] shows that for the pointer machine the 2-dimensional range reporting problem can be solved with query time $O(\log n)$ and space usage $O(n \log n / \log \log n)$. The idea is to use the construction of Section 1.5.6 giving the structure degree $\log^{\epsilon} n$ and using a priority search tree to answer 3-sided queries. As described in Section 1.5.6 we then in each internal node T need to have a 4-sided range reporting structure where the x axis has length $\log^{\epsilon} n$. This structure is created as follows. For each element on the x-axis we store, in a list, the points with that x-coordinate sorted according to the y-axis. We keep a single structure for the 1-dimensional point enclosure reporting problem and for all pairs (y_1, y_2), where y_1 and y_2 are neighbors in one of the lists, we store the line segment between y_1 and y_2 in this structure. For a given point y we can then find the predecessors of y in all the lists by making a single query in this structure and the total search time can be made $O(\log n)$. The space usage becomes $O(n \log n / \log \log n)$. The structure can be extended into higher dimensions $d > 2$ using some of the techniques of Section 1.5.4. Chazelle [27] remarks that using range trees with slack parameter this structure can be extended to any constant dimension $d > 2$ with space usage $O(n(\log n / \log \log n)^{d-1})$ and query time $O(\log^{d-1+\epsilon} n)$ for any constant $\epsilon > 0$. More interestingly, Chazelle in the same paper shows that the space usage of all these structures are optimal: Any pointer machine range reporting structure with poly-logarithmic search time (and constant time per point reported) must use space $\Omega(n(\log n / \log \log n)^{d-1})$. So for $d = 2$ the problem is completely understood. However, like for the RAM, it is still not known what the search time should be for higher dimensions. Is it possible to get search time $O(\log n)$ for any constant dimension?

1.5.9 The I/O model

Arge, Samoladas and Vitter [13] described a dynamic structure supporting 3-sided queries in time $O(\log_B n + k/B)$ when k points are reported, supporting updates in time $O(\log_B n)$ and using space $O(n/B)$ disk blocks. This structure is optimal in terms of time per operation in the comparison order variant. The solution is based on a priority search tree with degree B^{ϵ} for a constant $\epsilon > 0$ and is very similar to the structure used by Willard [112] for the RAM. They also describe a dynamic structure supporting 4-sided queries in time $O(\log_B n + k/B)$ when k points are reported, supporting updates in time $O((\log_B n) \log(n/B) / \log \log_B n)$, and using space $O((n/B) \log(n/B) / \log \log_B n)$ disk blocks. This structure is very similar to the structure of Chazelle [25] as described in Section 1.5.8. The tree is given degree $O(\log_B n)$ instead of $\log^\epsilon n$ and they use an I/O efficient structure to handle the 1-dimensional point enclosure problem.

As mentioned in Chapter 3, the results of that chapter may very well be modified to improve the result of [13], at least for realistic values of N and B. This may even lead to structures usable in practice. One of the main ingredients in such an improvement would be to adapt the construction of Section 1.5.1 to the I/O model. We now sketch a way to do this. For $B = 2$ we get the same construction as in Section 1.5.1 so the sketch will in fact be a generalization. The sketch requires detailed knowledge of the construction of Section 1.5.1. The construction of Section 1.5.1 relies on the fact that every time we need to follow a pointer upwards in the structure when reporting points, we know that in the end we will report a point which will pay for following the pointer. In the generalization, there will be at least $B/2$ additional points which can be reported in the end every time we follow a pointer. This is obtained in the following way. For the rest of this paragraph consider a fixed level in the recursion, a fixed y-coordinate and the rectangle of the structure that corresponds to a fixed bottom structure and a point in the top structure. Like in the construction of Section 1.5.1 all points for the area are inserted in the bottom structure. The construction of Section 1.5.1 supports inserting 2 points into the bottom structure without going further down in the recursion. We increase this to B points (to be placed on one disk block). When the considered area contains at most $B/2$ points these points are also kept in the top structure (the structure has to support multiple points stored at the same coordinates). When we get point $B/2 + 1$ we replace an arbitrary point of the top structure with a pointer like in the construction of Section 1.5.1. For each additional point we get (up to B points) we remove an arbitrary point (different from the pointer) from the top structure. This concludes the sketch.

Finally, the static 3-sided and 4-sided range reporting problems have been studied in the *cache oblivious* model of computation. This model is as the I/O model except that B is unknown to the algorithm. Instead it is assumed that the system automatically swaps disk blocks in and out of memory as necessary and the time usage is the number of I/Os the system needs to make. In this model, Arge et. al. [12] gave a static structure for 3-sided queries with space usage $O(N \log N)$ and query time $O(\log_B N + k/B)$ when k points are reported. They used this structure to get a structure for 4-sided queries with the same query time but with space usage $O(N \log^2 N/ \log\log N)$. The paper also contains results for static cache-oblivious range counting and semigroup variants.

1.5.10 The semigroup variant

This thesis contains no results for the semigroup variant. However, since the range variant of the semigroup variant is well understood in terms of upper and lower bounds, and since these bounds looks similar to the bounds for the range reporting problem, we will survey some of the results here.

Fredman [47] showed that in the range variant in the arithmetic model n operations in the form of updates and queries take time $\Omega(n \log^d n)$ in d dimensions. This lower bound assumes that the algorithm works independently of the semigroup. Using a structure similar to a range tree this is also an upper bound. Chazelle [28] considered the dominance variant also in the arithmetic model and showed that for the static case with space usage m a query takes time $\Omega((\log n/ \log(2m/n))^{d-1})$. This is complemented with upper bounds tight for $m = \Omega(n(\log n)^{d-1+\epsilon})$ for any constant $\epsilon > 0$. So for small values of m, which are maybe the most interesting, the problem is still not completely understood. The upper bound is essentially obtained by using a variant of a range tree with degree $\log^\epsilon n$ using the all-children-pair method of Section 1.5.4. Chazelle also considered the semi-dynamic problem where insertions but not deletions are allowed. In this variant he shows that n inserts and queries take time $\Omega(n(\log n/\log\log n)^d)$. We finally remark, that unlike the lower bounds of Fredman, the lower bounds of Chazelle do not require the structure to work for any semigroup: The lower bounds of Chazelle works for any semigroup which fulfills some reasonable constraints. Note that it is not possible to show a non-trivial lower bound holding for any semigroup. As an example, consider the trivial semigroup consisting of only a neutral element.

1.5.11 Saving space in the static case

In Section 1.5.2 and 1.5.6 we have described two different structures for the 4-sided range reporting problem using space $O(n \log n)$. Chazelle [26] described how the space usage of static range reporting can be reduced and he obtained the following two results. The first uses space $O(n \log^\epsilon n)$ for any constant $\epsilon > 0$ and has query time $O(\log n + k)$ when k points are reported. The second uses space space usage $O(n \log\log n)$ and has query time $O(\log n + k \log\log n)$ when k points are reported.

We now partly describe how Chazelle obtained his structure. As base of the structure we use a range tree T of Section 1.5.2 with binary degree. The lists in each node of T is represented as an array with one bit for each point. This bit indicates whether this point is in the left or the right subtree of the node. The total space used by these bits is $O(n)$ since for each point we use one bit for each level of T. The effect of a downpointer is obtained as follows. In each list, we group the list into blocks with $\log n$ bits. In each block we store how many 1 (resp 0) bits there is in the blocks to the left of the block. The number of 1 (resp 0) bits to the left of a given bit can then be found by adding the corresponding number for the block with the number of 1 (resp 0) bits in a word of at most $\log n$ bits. The later value can be found in constant time using a global lookup table. This sum is exactly what we need in order to find the position a point has in the bit array of its children so we get the effect of downpointers. Before continuing we remark that the structure so far uses space $O(n)$ and also can be used to answer range count queries in time $O(\log n)$. We return to range counting in Section 1.5.14.

One problem remains. When reporting points during a query we have to *identify* which point a given bit corresponds to. We will not in detail describe how this is done. The main idea is to span a tree over the levels of T much in the same way as we do in Chapter 4. If the tree has constant degree we get the result with space usage $O(n \log\log n)$ and if the tree has constant height we get the result with space usage $O(n \log^\epsilon n)$.

Chazelle was only interested in the comparison order variant. In the list order variant we can reduce the query time of the structure just mentioned using the the technique of Section 1.5.6. The following description is very sketchy. We will make a range tree in the same way as Chazelle and use the same structure to identify which point corresponds to a given bit. In addition to this we will make 3-sided structures in the style of Section 1.5.6. The 3-sided structure we need in each node of the tree should be as follows. It should contain $m < n$ points. The x-axis is in the n-order variant and the y-axis in the comparison order variant. The structure should use space $O(m \log\log n)$ bits. The structure is given a function which given the rank of a point on the x-axis gives the precise coordinate in either constant time (for the $O(n \log^\epsilon n)$ space solution) or in time $O(\log\log n)$ (for the $O(n \log\log n)$ space solution). It is not hard to make such a structure by grouping points into blocks with $O(\log n)$ points on the x-axis and then for each block to insert the point with smallest y-coordinate in the 3-sided structure of [54]. In each block we have only $O(\log n)$ points and these can easily be handled. The problem that remains is that for a given x-coordinate we need to identify the rank of it among the points contained. To do this we maintain a predecessor structure on the blocks which allows us to identify the block the predecessor corresponds to. Inside the block, we then do a binary search among the

	Update time	Query time	Space usage
[26]		$O(\log n + k)$	$O(n \log^\epsilon n)$
[26]		$O(\log n + k \log\log 4n/k)$	$O(n \log\log n)$
[6]		$O(\log\log n + k)$	$O(n \log^\epsilon n)$
[6]		$O((\log\log n)^2 + k)$	$O(n \log\log n)$
[36, 75]	$O(\log n \log\log n)$	$O(\log n \log\log n + k)$	$O(n \log n)$
[67]	$O(\log n)$	$O(n^\epsilon)$	O(n)
Chapter 3	$O(\log^\delta n)$	$O(\log n/\log\log n + k)$	$O(n \log^\delta n)$
[86]	$O(\log^2 n)$	$O(\log n + k)$	$O(n \log^\epsilon n)$
[86]	$O(\log^2 n)$	$O(\log n + k \log\log n)$	$O(n \log\log n)$
Conjectured	$O(\log^\gamma n)$	$O(\log n/\log\log n + k)$	$O(n \log^\epsilon n)$
Conjectured	$O(\log n)$	$O(\log n/\log\log n + k \log\log n)$	$O(n \log\log n)$

Figure 1.1: Upper bounds for the 2-dimensional range reporting problem on the RAM. $\epsilon > 0$ is *any* constant, $0 < \delta < 1$ is a specific constant and $\gamma < 1$ is a constant depending on ϵ.

$O(\log n)$ elements for the predecessor. For each step of the binary search we have to identify the coordinates of a point. The search time for the $O(n \log\log n)$ space solution thus becomes $O(\log^2 \log n)$ and for the $O(n \log^\epsilon n)$ space solution it becomes $O(\log\log n)$.

The results of the previous paragraph are not new. Exactly the same tradeoffs are obtained by Alstrup et. al. [6] by using an elegant but completely different construction. It is interesting that both the construction in the previous paragraph and the construction in [6] obtains the strange-looking $O(\log^2 \log n)$ search time. In [6] the same construction is used to obtain improved bounds for the static range semigroup variant of intersection searching. Further, it is directly dynamizeable which we will discuss in Section 1.5.12.

1.5.12 Saving space in the dynamic case

In figure 1.1 we have given an overview of different data structures for range reporting reporting in two dimensions on the RAM including the static structures from the preceding section.

The structure of [6] basically builds a 4-sided range reporting structure with small space usage from a given 4-sided range reporting structure with high space usage and from a given 3-sided structure. It creates many such structures most of them with few points and then saves these structures in a reduced universe to save space. The structure can be dynamized and this is done by Nekrich [86]. As mentioned in Section 1.4.2 he converts a structure for the m-order variant to one for the comparison order variant by renumbering coordinates. We now sketch how to obtain the bounds in the two last lines of Figure 1.1. Since this is only a sketch and since I have not made a full proof, I only state the bounds as conjectures. As Nekrich we will dynamize the structure of [6]. By using the list order variant structures of Chapter 3 as given structures we do not need to renumber coordinates as in [86].[5] When we proceed down the recursion in the construction we need something similar to downpointers/fractional cascading. The structure of Theorem 3.15 can be used for this purpose. While the structure in [6] continues the recursion until there is only a constant number of points left we will stop a little earlier. When we stop, we will store the points in the given 4-sided structure. We stop at the point where the space we use on this is equal to the space used by the rest of the structure except for a constant factor. This completes our sketch.

[5] Actually, the idea of the list order variant originally came from a desire of dynamizing [6]

1.5.13 Chapter 4: Dynamic one-dimensional range reporting in the word order variant

In Chapter 4 we consider one-dimensional range reporting in the word order variant. Let $1/\log w \leq \gamma \leq 1$ be a parameter. We show how to maintain a data structure for a dynamic set of keys $K \subseteq [0..2^w - 1]$ such that with high probability keys can be inserted and deleted in time $O(w^\gamma/\gamma)$ and such that we given i and j in worst case time $O(1/\gamma)$ can find a key from $K \cap [i..j]$ or determine that no such key exists. The space usage of the structure is linear

In one end of the spectrum (constant γ) we have constant query time and an update time which is exponentially smaller than the update time needed if we have to answer the related predecessor query. In the other end of the spectrum ($\gamma = 1/\log w$) the structure has the same performance as the classical VEB of Theorem 1.3.

The chapter is based on a dynamization of a method by Alstrup, Brodal and Rauhe [5] which provides a static structure for the same problem with constant query time and construction time $O(n\sqrt{w})$. The basic idea of both [5] and Chapter 4 is to build a trie over the universe $[0 \dots 2^w - 1]$ augmented with extra information. In Chapter 4 we design a dictionary for paths on this trie. A clever hashing scheme ensures the dictionary uses linear space in the number of paths in it. Further, we show that this dictionary is sufficient to maintain the trie and the needed extra information.

In a subsequent paper [81] based on Chapter 4 the structure is improved giving a new set of tradeoffs. For constant query time the update time is still $O(w^\epsilon)$ for any constant $\epsilon > 0$ while for update time $O(\log w)$ the query time is $O(\log\log w)$.

Unlike [81], Chapter 4 dynamizes a static structure given in [5] for dynamic approximate range counting. For example, for any constant $\epsilon > 0$ and any two words the structure can in constant time determine the number of points between the words within a factor $(1 - \epsilon)$. This can be used for a heuristic for range reporting in high dimensions as outlined in [5] and in Chapter 4.

1.5.14 Chapter 5: Static range counting and dominance reporting

Chazelle and Edelsbrunner [29] proposed two linear-space algorithms for the static 3-dimensional dominance reporting problem for a pointer machine. The first has query time $O(\log n + k \log n)$ when k points are reported and the second has query time $O(\log^2 n + k)$. On the RAM, Makris and Tsakalidis [71] shows how to solve the problem in the U-order variant in linear space and with query time $O((\log\log U)^2 \log\log\log U + k \log\log U)$ when k points are reported. They also describe a variant of the structure of [29] which works by performing 3-sided queries. Using the structure of [53] to answer these queries they obtain a structure for the pointer machine with space usage $O(n)$ and search time $O(\log n \log\log n)$. Using the 3-sided structure of [54] the search time is reduced to $O(\log n)$ on a RAM.

In Chapter 5 we reduce the search time further down to $O(\log n/\log\log n)$ on the RAM keeping the linear space usage. Further, we provide a slightly generalized result which allows us to extend the structure to higher dimensions using the universe extending method of Section 1.5.4. This gives s a solution to the d-dimensional version of the problem with search time $O((\log n/\log\log n)^{d-2})$ and space usage $O(n(\log n/\log\log n)^{d-3})$ for $d \geq 3$.

On a RAM, Chazelle [26] described how to solve the range counting problem in 2-dimensions using space $O(n)$ and query time $O(\log n)$ and in Section 1.5.11 we also sketched how this result was obtained. In Chapter 5 we reduce the query time of this to $O(\log n/\log\log n)$ by increasing the degree of the tree to $O(\log^\epsilon n)$ for a constant $\epsilon > 0$. Again, we provide a slightly generalized result allowing us to extend these structures into higher dimensions $d > 2$

using the universe extending method of Section 1.5.4. We obtain a structure with query time $O((\log n/\log\log n)^{d-1})$ using space $O(n(\log n/\log\log n)^{d-2})$ for $d \geq 2$.

An external version of Chazelle's scheme was proposed by Govindarajan, Arge, and Agarwal [56] which achieves linear space and $O(\log_B n)$ I/Os for handling range counting queries in $\mathbb{R}^2$. They also extended their algorithm to handle higher dimensional range counting queries in a similar way to Chapter 5, introducing a factor of $O(\log_B n)$ to both the space and the query complexity for each additional dimension. The results of[56] were published some time before the paper in Chapter 5 but the results were discovered independently.

1.5.15 Chapter 6: Colored range reporting

In this section we discuss colored intersection searching.

The first paper published about colored intersection searching was by Janardan and Lopez [65] and they considered a pointer machine. They first described a solution to the static 1-dimensional colored range reporting problem with space usage $O(n)$ and query time $O(\log n)$. We now give a rough description of this structure. We span a complete binary tree T over the points in sorted order. For each node $v \in T$ we create a list $v.R$ (resp. $v.L$) which contains, in order, the elements below the right (resp. left) child w of v which do not have any predecessor (resp. successor) with the same color below w. A query (x_1, x_2) can then be answered by finding the NCA of x_1 and x_2 in T and scan the lists $v.L$ and $v.R$. The search time becomes $O(\log n)$ and the space usage $O(n \log n)$. The space usage is reduced in the following way. If a point is stored in a list $v.L$ (resp. $v.R$) for more than one node $v \in T$ we will only store it in the list for the highest such node in T. During a query, we then not only have to search in the lists $v.R$ and $v.L$ for a single node v but also in all its ancestors. Fractional cascading is used to keep the search time on $O(\log n)$ while the space usage is reduced to $O(n)$.

The paper [65] also contained solutions to static segment and range variants of colored reporting. The results of colored segment intersection reporting is obtained by reducing the problem to the non-colored variant of segment intersection and they obtain search time $O(\log n)$ and space usage $O(n \log n)$ or search time $O(\log^2 n)$ and space usage $O(n)$ (see Figure 6.1 for a summary of results for colored segment reporting). Since a structure (colored or not) which can answer segment reporting queries can also answer 3-sided queries they can apply the technique of Section 1.5.6 and get a structure supporting 4-sided queries by adding a factor $O(\log n)$ to the space usage (see Figure 6.1 for a summary of results for colored range reporting reporting).

Gupta, Janardan and Smid [57] also considered the pointer machine. They reduced 1-dimensional colored range reporting to 3-sided non-colored reporting using the following observation. Consider a query (x_1, x_2). Let p be a point with color c and let p' be the predecessor of p among the points with color c. Then p is the point in the interval from x_1 to x_2 with color c closest to x_1 if and only if $x_1 \leq p \leq x_2$ and $p' < x_1$ and thus p can be found by a 3-sided query. Using a priority search tree to handle the 3-sided queries they thus get a dynamic version of the 1-dimensional result of [65] with the same search time ($O(\log n)$) and space usage ($O(n)$) and with update time $O(\log n)$. The paper contains a lot of other results and they also use partial persistence on dynamic structures to obtain some of them using a sweepline. By using this technique on the 1-dimensional structure just mentioned, it is possible to obtain a structure for segment reporting with space usage $O(n \log n)$ and search time $O(\log n)$ matching the performance of one of the structures from [65] (this is not described in [57]).

Muthukrishnan [85] observed that on a RAM it is possible to use the constant-time structure of [54] instead of the priority search tree and in this way he obtained a static structure for 1-dimensional colored range reporting with linear space usage and constant search time.

Agarwal, Govindarajan and Muthukrishnan [2] described a static structure for 2-dimensional colored range reporting in the U-order variant on the RAM. The space usage is $O(n \log^2 U)$ while the search time is $O(\log \log U)$. By making the structure for the n-order variant and combining it with a predecessor structure we can immediately reduce the space usage to $O(n \log^2 n)$ matching the space usage of [65]. The idea of their structure is the following. They take the 1-dimensional structure of [65] in the version using $O(n \log n)$ space. They observe that this structure can be made partly dynamic such that new points can be inserted with $O(\log n)$ modifications to the structure (one on each level of T). From this they make a partially persistent version of the structure supporting n insertions and using space $O(n \log n)$ and with search time $O(\log \log U)$. Using a sweepline this gives a structure for colored 3-sided reporting with search time $O(\log \log U)$ and space usage $O(n \log U)$. Finally the technique of Section 1.5.6 is used to give a structure supporting general 4-sided queries.

In Chapter 6 a structure for the dynamic 1-dimensional colored range reporting problem in the bijective n-order variant on the RAM is given. The structure reduces the update and search time of the pointer machine structure of [57] from $O(\log n)$ to $O(\log \log n)$. The result is obtained as follows. By using a slight variant of [65] we are able to reduce the problem to 3-sided range reporting where the x-axis is in the bijective n-order variant and the y-axis is in the $\log n$-order variant (the height of T). Plugging a variant of a priority-search tree of [112] into the construction of Section 1.5.1 we get can get such a structure and obtain the claimed bounds. Using the techniques already outlined in the previous paragraphs the result can be transformed into a static structure for colored segment reporting with space usage $O(n \log \log n)$ and search time $O(\log^2 \log n)$ and into a structure for static range reporting with space usage $O(n \log n \log \log n)$ and search time $O(\log^2 \log n)$. By using a naive data structure instead of the structure based on Section 1.5.1 other tradeoffs are given in Chapter 6 (see Figure 6.1 and Figure 6.2). Building on a preliminary version of Chapter 6 Shi and JaJa improved this and got a structure with search time $O(\log n)$ and space usage $O(n)$ for segment reporting and search time $O(\log n)$ and space usage $O(n \log n)$ for range reporting. Their structures run on a pointer machine and the improvement is based on using 1-dimensional point enclosure structure in the same way as it is described in Section 1.5.8.

1.6 Chapter 7: Implicit Priority Queues with Decrease-key

As the only chapter of this thesis, Chapter 7 is completely unrelated with intersection searching and thus has essentially nothing to do with the rest of the thesis. For this reason we will only briefly cover this chapter here. While the other results of this thesis shows upper bounds, Chapter 7 shows a lower bound. The problem considered is a heap in the implicit model of computation (see Section 1.3.6). The most famous structure in this model is probably Williams's binary heap [114], which supports the priority queue operations *insert* and *delete-min* in logarithmic time. What we consider is implicit heaps supporting the *decrease-key* operation like the Fibonacci heap [50]. Our main result is that if an implicit heap uses *zero extra space* and supports decrease-key in $o(\log^* n)$ amortized time, then the amortized cost of insert/delete-min must be $\Omega(n^{1/\log^{(k)} n})$ for any constant k. The original version of this paper [82] claimed a matching upper bound giving an implicit heap supporting insert and delete-min in logarithmic time and decrease-key in $O(\log^* n)$ time. However, there was a bug in the analysis so the upper bound is not included here. Our lower bound relies crucially on the fact that the heap has no extra storage. Given just one word of extra storage our lower bound breaks completely. Our lower bound works by reducing the decrease-key operation to a simple problem which we have called the *Absent Minded Usher's Problem.* Imagine an usher seating indistinguishable theater patrons one-by-one in a large theater. The usher is equipped with two operations: he can *probe* a seat to see if it is occupied or not and he can *move* a given patron to a given unoccupied seat. (Moving patrons after seating them is perfectly acceptable.) The catch is this: before seating each new patron we wipe clean the usher's memory. That is, he must proceed without any knowledge of which seats are occupied or the number of patrons already seated. We give short proofs showing that any deterministic ushering algorithm must seat m patrons with $\Omega(m \log^* m)$ probes and moves, and that this bound is asymptotically tight.

1.7 Conclusion

We have developed several new data structures for intersection searching and shown a lower bound for implicit heaps. I will conclude this introduction by stating which contributions of this thesis I find most interesting.

First, I will mention the construction of Section 1.5.1 which allows us to make a 4-sided structure which can contain many points from one which can contain few points. As already mentioned, this construction is the base of Chapter 2, Chapter 3 and also plays a role in Chapter 6. Further, as outlined in Section 1.5.9 the construction may also lead to new bounds in the I/O model of computation.

Second, I will mention the definition and tight analysis of the Absent Minded Usher's Problem in Chapter 7. This problem is interesting because it is simple to state and also has a rather simple analysis leading to a tight bound of $\Theta(\log^* n)$. Further a tight bound of $\Theta(\log^* n)$ is rather unusual.

Third, the construction leading to the colored 1-dimensional structure of Chapter 6 is interesting though as mentioned it is somewhat similar to an existing construction [65].

Chapter 2

Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time

Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time

Christian Worm Mortensen *

Abstract

We consider the two dimensional fully-dynamic orthogonal range reporting problem and the two dimensional fully-dynamic orthogonal line segment intersection reporting problem in the comparison model. We show that if n is the number of stored elements, then these problems can be solved in worst case time $\Theta(\log n)$ plus time proportional to the size of the output pr. operation.

2.1 Introduction

The two dimensional fully-dynamic orthogonal range reporting problem (henceforth the range reporting problem) is to maintain a finite set $R_r \subset \mathbb{R}^2$ of points under insertions and deletions such that for any query rectangle q the subset of R_r contained in q can be reported.

The two dimensional fully-dynamic orthogonal line segment intersection reporting problem (henceforth the segment reporting problem) is to maintain a finite set R_s of horizontal line segments in $\mathbb{R}^2$ under insertions and deletions such that for any vertical line segment q the set of line segments in R_s which intersect q can be reported.

Let n be the number of maintained elements and k be the number of elements reported by a given query. In this paper we develop deterministic data structures for the range and segment reporting problems such that updates can be performed in worst case time $O(\log n)$ and queries can be answered in worst case time $O(\log n+k)$. The space usage of our structures is $O(n \log n/ \log\log n)$. The model of computation is a unit-cost RAM with a word size logarithmic in n.

2.1.1 Previous results

Orthogonal range searching including range and segment reporting has been extensively studied during the last 30 years. For surveys see Agarwal and Erickson [1] and Chiang and Tamassia [32]. For books with earlier results see Mehlhorn [74] and Preparata and Shamos [92].

Besides the applications in computational geometry, range reporting has applications in databases. Consider a database of persons where each person has an associated age and weight. We can represent each record of this database as a point in $\mathbb{R}^2$. A range reporting query can then ask for all persons between 40 and 50 years which have a weight between 60 and 80 kg. Willard [111] gives additional applications in databases.

We have summarized various data structures for the range and segment reporting problems in table 1. Most data structures for the range and segment reporting problems, including

*IT University of Copenhagen. E-mail:cworm@it-c.dk

Update Time	Query Time	Space usage	Remarks	Source
$O(\log^2 n/\log\log n)$	$O(\log^{2+\epsilon} n/\log\log n + k)$	$O(n\log n/\log\log n)$	R	[102]
$O(\log n)$	$O(\log^2 n + k)$	$O(n)$	S	[31]
$O(\log n)$	$O(\log n + k)$	$O(n\log n)$	RSI	[62]
$O(\log n\log\log n)$	$O(\log n\log\log n + k)$	$O(n\log n)$	RS	[36, 75]
$O(\log n)$	$O(\log n + k)$	$O(n\log n/\log\log n)$	RS	New

Remarks: R: Range reporting, S: Segment reporting, I: Only insertions or deletions.

Table 2.1: *Various data structures for the range and segment reporting problems.*

the structures of this paper, are based on variants of range and segment trees [19, 70, 16, 20, 108]. Range and segment trees support updates in time $O(\log^2 n)$ and queries in time $O(\log^2 n + k)$. In the static case without updates, Willard [108] described how the query time could be reduced to $O(\log n + k)$ by the use of so-called downpointers. Chazelle and Guibas [30] generalized downpointers as well as other techniques into a data structuring technique which they called fractional cascading. Mehlhorn and Näher [75] made fractional cascading dynamic with amortized time bounds on updates. They used this to develop data structures for the range and segment reporting problems with update time $O(\log n\log\log n)$ and query time $O(\log n\log\log n + k)$. Dietz and Raman [36] described general methods for deamortization and they used this to remove amortization from dynamic fractional cascading. Combining this with techniques of Willard and Lueker [113] they removed amortization from the range and segment reporting structures in [75].

2.1.2 Outline of our method and paper

The data structures of this paper build directly on the data structures of [36] and the work which led to them as described in section 2.1.1. We do, however, not consider fractional cascading in general, since an approach similar to the simpler downpointer technique is sufficient for our purpose.

After the preliminaries in section 2.2, we in section 2.3 describe what could be called a $1+\epsilon$ dimensional dynamic range reporting structure for a finite universe (ie. a grid line). The structure is somewhat similar to the data structure of van Emde Boas et al. [105] but also uses techniques of Fredman and Willard [52]. In section 2.3 we also sketch how the described structure can be used to create data structures for the segment and range reporting problems on a grid with update time $O(\log n)$ and query time $O(\log n + k)$. Thus the main purpose of section 2.4, 2.5 and 2.6 is to show how to handle the situation where we are not on a grid.

In section 2.4 we use techniques similar to the ones used by Dietz and Sleator [37] and Dietz and Raman [36] to extend the data structure of section 2.3 to work on a linked list instead of a grid line. In section 2.5 we review WBB trees (Weight Balanced B-trees). These trees have also been described by Dietz [35] and Arge and Vitter [14]. Following a general suggestion in [14] we use these trees as a simple replacement for the techniques in [113]. In section 2.6 we combine the structures of section 2.4 and 2.5 into our final structures.

The paper concludes with acknowledgements in section 2.7.

2.2 Preliminaries

For $x_1, x_2 \in \mathbb{R}$ we define $[x_1 \ldots x_2] = \{x \in \mathbb{R} \mid x_1 \le x \le x_2\}$ and for an arbitrary finite set M we define $|M|$ as the cardinality of M. If A is an array indexed by a set M, we for $m \in M$

define $A[m]$ as the element of A at index m. For integers $i \geq 0$ and $d > 0$ we define $i \bmod d$ and $i \operatorname{div} d$ in the standard way to be the unique integers such that $0 \leq i \bmod d < d$ and $i = d(i \operatorname{div} d) + i \bmod d$.

If M is a set of totally ordered elements and $m, m' \in M$, we write $m < m'$ as an abbreviation for $m \neq m'$ and $m \leq m'$. In the rest of this paragraph we assume M is finite. We define the rank of an element $m \in M$ as $|\{m' \in M \,|\, m' < m\}|$. We define $\min M$ ($\max M$) as the minimal (maximal) element of M or the special symbol $\perp$ if M is empty. Similarly, if each element $m \in M$ has an associated value $m.v$ from a total order, we define $\min_v M$ ($\max_v M$) as $\perp$ if M is empty and else as the element $m \in M$ such that for all $m' \in M$ we have $m.v \leq m'.v$ ($m.v \geq m'.v$).

We order the elements of a linked list L such that if $e, e' \in L^*$ and e is before in e' in L or if $e = e'$ then $e \leq e'$. Further, when inserting an element e' into a non-empty linked list L, we assume we are given the element $e \in L$ which should be the predecessor of e' after the insertion†.

If T is a tree we consider the leafs of T to be the elements of T, but for convenience, we will write $e \in T$ if e is a node in T. We order the elements of T such that if $e, e' \in T$ are leafs and e is to the left of e' or if $e = e'$ then $e \leq e'$.

In the rest of this paper, all given time and space bounds are deterministic and worst case. In general, we let k be the number of elements reported by a given query. We assume N is a fixed value and that we can use preprocessing time $O(N)$ for building tables of size $O(N)$. As a corollary we assume we can compute simple functions from the integers in $[0 \ldots O(N)]$ to the integers in $[0 \ldots O(N)]$ in constant time. With these assumptions we in theorem 2.9 and 2.10 below give data structures for the range and segment reporting problems with a space usage of $O(N \log N / \log\log N)$ such that $O(N)$ insertions can be performed and such that updates can be performed in time $O(\log N)$ and queries can be answered in time $O(\log N + k)$. Our results from section 2.1 can easily be obtained from these data structures by using the global rebuilding techniques of Overmars [87].

We assume $\epsilon > 0$ is a sufficiently small constant. We define the set C of colors to be the integers in $[0 \ldots O(\log^\epsilon N)]$ and we represent sets $C' \subseteq C$ as bit vectors in a single word.

2.3 A data structure for a colored grid line

In this section we describe a data structure G_n containing elements $e \in G_n$. We require n to be at most $O(N)$. Each element $e \in G_n$ has an associated position $e.\texttt{pos}$ and color $e.\texttt{col}$. We require that $e.\texttt{pos}$ is an integer in $[0 \ldots n-1]$ and that $e.\texttt{col} \in C$. Two different elements in G_n are not allowed to have both the same position and color implying $|G_n| \leq n|C|$. For $c \in C$ and $0 \leq i < n$ we define $\texttt{pred}(G_n, i, c) = \max_{\texttt{pos}}\{e \in G_n \,|\, e.\texttt{col} = c,\, e.\texttt{pos} \leq i\}$ and $\texttt{succ}(G_n, i, c) = \min_{\texttt{pos}}\{e \in G_n \,|\, e.\texttt{col} = c,\, i \leq e.\texttt{pos}\}$. Further, for $0 \leq i < n$, $0 \leq j < n$ and $C' \subseteq C$ we define $\texttt{report}(G_n, i, j, C') = \{e \in G_n \,|\, i \leq e.\texttt{pos} \leq j, e.\texttt{col} \in C'\}$. In this section we proof the following lemma:

Lemma 2.1. *We can maintain G_n using space $O(n \log\log n \log^\epsilon N)$ such that we in time $O(\log\log n)$ can a) insert and delete elements and b) answer* `pred` *and* `succ` *queries*‡. *Also, we can c) answer* `report` *queries in time $O(\log\log n + k)$.*

Lemma 2.1 almost immediately gives data structures for the range and segment reporting problems on an n times n grid with update time $O(\log n)$ and query time $O(\log n + k)$. We now

*We will not distinguish between a data structure and the set of elements it contains

†We will not distinguish between an element in a data structure and a pointer to it

‡We will not distinguish between a query and its answer.

sketch how to obtain these structures. We do not refer to this sketch later in the paper. For the range reporting problem, we create a tree X with degree $O(\log^\epsilon N)$ and n leafs numbered from 0 to $n-1$ from left to right. We color the children of each internal node of X with a unique color from C. In each internal node v of X we keep a secondary structure $v.\mathtt{L}$ with type G_n from lemma 2.1. Suppose e is a point on the grid at position $(e.\mathtt{x}, e.\mathtt{y})$ implying $0 \le e.\mathtt{x} < n$ and $0 \le e.\mathtt{y} < n$. Further suppose, that l is the leaf with number $e.\mathtt{x}$ and that v' is a non-root ancestor of l (or l itself) with color c and parent v. We then save e in $v.\mathtt{L}$ with color c and position $e.\mathtt{y}$. An update to the structure just described only requires an update to one secondary structure on each level of X. Similarly, the points in a given query rectangle can be reported by performing a single `report` query in at most two secondary structures on each level of X. Since X has height $O(\log n/\log\log n)$ the claimed time bounds follows. The segment reporting problem can be solved in a similar way by assigning colors to pairs of children instead of to children. We need to reduce the degree of X to $O(\log^{\epsilon/2} n)$ to have sufficient colors to do this.

The rest of this section is devoted to the proof of lemma 2.1. We create a data structure of van Emde Boas et al. [105] (henceforth a VEB) for each color in C. Each element $e \in G_n$ is inserted at position $e.\mathtt{pos}$ in the VEB for color $e.\mathtt{col}$. Using these VEBs we link all elements with the same color together in order according to their positions. This will not use too much space and allows us to perform insertions, deletions as well as `pred` and `succ` queries in time $O(\log\log n)$ pr. operation.

To answer a `report` query using these VEBs, we need to search separately in each VEB using time $\Omega(\log\log n \log^\epsilon N)$ in the worst case even if we do not find any elements to report. To answer `report` queries fast, we therefore in addition to the VEBs maintain a data structure S_n which we develop below. The data structure S_n contains triples $(e,i,c) \in S_n$ where e is an arbitrary pointer, i is an integer in $[0\ldots n-1]$, and $c \in C$ is the color of the triple. We maintain S_n such that $(e, e.\mathtt{pos}, e.\mathtt{col}) \in S_n$ iff $e \in G_n$. We support a special $\mathtt{reportany}(S_n, i, j, C')$ query in S_n: For each color $c \in C'$ for which there exists a triple $(e,t,c) \in S_n$ such that $i \le t \le j$, the query $\mathtt{reportany}(S_n, i, j, C')$ reports (e,c) from exactly one such triple. A $\mathtt{report}(G_n, i, j, C')$ query can then be answered by first performing a $\mathtt{reportany}(S_n, i, j, C')$ query. For each pair (e,c) reported by this query, we can follow the pointers maintained by the VEB for color c to report the rest of the elements.

We now describe the structure S_n. To avoid tedious details, we assume n has the form $n = 2^{2^m}$ for some integer $m \ge 0$. We observe that if $n > 2$ has this form, then $\sqrt{n}$ has same form. The structure S_n is somewhat similar to a VEB, and we define it inductively on n. If $S_n = \emptyset$ the recursion stops. Else, we keep an array $S_n.\mathtt{min}$ ($S_n.\mathtt{max}$) indexed by C. For each $c \in C$ we store the triple $(e,j,c) \in S_n$ with minimal (maximal) value of j in $S_n.\mathtt{min}[c]$ ($S_n.\mathtt{max}[c]$) if any. We also keep an array $S_n.\mathtt{bottom}$ indexed by the integers in $[0\ldots\sqrt{n}-1]$ where in each entry we store a recursive structure with type $S_{\sqrt{n}}$. We store each triple $(e,i,c) \in S_n \setminus (S_n.\mathtt{min} \cup S_n.\mathtt{max})$ as $(e, i \bmod \sqrt{n}, c)$ in $S_n.\mathtt{bottom}[i \operatorname{div} \sqrt{n}]$. Finally, we keep a single recursive structure $S_n.\mathtt{top}$ also with type $S_{\sqrt{n}}$. If for $c \in C$ the structure $S_n.\mathtt{bottom}[i]$ contains exactly one triple $(e,j,c) \in S_n$ with color c, we store (e,i,c) in $S_n.\mathtt{top}$. We note that in this case e is stored in both $S_n.\mathtt{bottom}$ and in $S_n.\mathtt{top}$. If $S_n.\mathtt{bottom}[i]$ contains more than one triple with color c we store (e',i,c) in $S_n.\mathtt{top}$ where e' is a pointer to the recursive structure in $S_n.\mathtt{bottom}[i]$.

Inserting a triple (e,j,c) in S_n: If S_n contains at most one triple with color c, we just update $S_n.\mathtt{min}[c]$ and $S_n.\mathtt{max}[c]$ and we are done. Else, we first check if (e,j,c) should go into $S_n.\mathtt{min}[c]$ ($S_n.\mathtt{max}[c]$) and if this is the case we interchange (e,j,c) and the triple in $S_n.\mathtt{min}[c]$ ($S_n.\mathtt{max}[c]$). Let T be the structure in $S_n.\mathtt{bottom}[j \operatorname{div} \sqrt{n}]$. We then insert $(e, j \bmod \sqrt{n}, c)$ in T. Let m be the number of elements in T with color c after this insertion. If $m = 1$, we insert $(e, j \operatorname{div} \sqrt{n}, c)$ in $S_n.\mathtt{top}$. If $m = 2$, there is a triple $(e', j \operatorname{div} \sqrt{n}, c)$ in $S_n.\mathtt{top}$, and we replace (see next paragraph) this triple with the triple $(e'', j \operatorname{div} \sqrt{n}, c)$ where e'' is a pointer to T. We observe that when we update $S_n.\texttt{top}$ then T has at most two elements with color c and the update in T is performed by just accessing $T.\texttt{min}$ and $T.\texttt{max}$. It follows that we do non-constant work in at most one recursive structure.

Replacing a triple $(e, j, c) \in S_n$ with the triple (e', j, c): If $(e, j, c) = S_n.\texttt{min}[c]$ ($(e, j, c) = S_n.\texttt{max}[c]$) we just update $S_n.\texttt{min}[c]$ ($S_n.\texttt{max}[c]$) and we are done. Else, let T be the structure in $S_n.\texttt{bottom}[j \operatorname{div} \sqrt{n}]$. First, we in T replace $(e, j \bmod \sqrt{n}, c)$ with $(e', j \bmod \sqrt{n}, c)$. Next, if T has exactly one triple with color c, we replace $(e, j \operatorname{div} \sqrt{n}, c)$ with $(e', j \operatorname{div} \sqrt{n}, c)$ in $S_n.\texttt{top}$. As with insertions it follows, that we only do non-constant work in at most one recursive structure.

Deleting a triple (e, j, c) from S_n: If S_n has at most two triples with color c, we just update $S_n.\texttt{min}[c]$ and $S_n.\texttt{max}[c]$ and we are done. Suppose therefore that S_n contains at least three triples with color c. We split into three cases: Case 1: In this case $(e, j, c) = S_n.\texttt{min}[c]$. Let $(e', t, c) = S_n.\texttt{top}.\texttt{min}[c]$, $(e'', l, c) = S_n.\texttt{bottom}[t].\texttt{min}[c]$ and $i = l + t\sqrt{n}$. Then (e'', i, c) is the triple in $S_n \setminus (S_n.\texttt{min} \cup S_n.\texttt{max})$ with color c which has the minimal value of i. Instead of deleting (e, j, c) from S_n we delete (e'', i, c) and afterwards we set $S_n.\texttt{min}[c]$ to (e'', i, c). Case 2: In this case $(e, j, c) = S_n.\texttt{max}[c]$ and this case is handled in a symmetric way to case 1. Case 3: In this case (e, j, c) is not equal to $S_n.\texttt{min}[c]$ or $S_n.\texttt{max}[c]$. The case is handled in way similar to insertions: Let T be the structure in $S_n.\texttt{bottom}[j \operatorname{div} \sqrt{n}]$. We then delete $(e, j \bmod \sqrt{n}, c)$ from T. Let m be the number of elements in T with color c after this deletion. Suppose $m = 1$ and let (e', i, c) be the triple with color c in T. Then there is a triple $(e'', j \operatorname{div} \sqrt{n}, c)$ in $S_n.\texttt{top}$ where e'' is a pointer to T and we replace this triple with the triple $(e', j \operatorname{div} \sqrt{n}, c)$. If $m = 0$ we delete $(e, j \operatorname{div} \sqrt{n}, c)$ from $S_n.\texttt{top}$. Again we observe that we only do non-constant work in at most one recursive structure.

What remains is to describe how to answer a `reportany` query. Answering this query intuitively consists of two parts: In the first part we inductively on n search down through S_n to find triples to report. In the second part, we follow zero or more pointers from the found triples up through S_n to get triples to report which were inserted by the user on the top level of the induction on n. Below we will first describe how to answer a $\texttt{reportany}_{\text{down}}$ query which corresponds to the first part. After this, we describe how to answer a `reportany` query by first performing a $\texttt{reportany}_{\text{down}}$ query and then do work corresponding to the second part.

In order to support $\texttt{reportany}_{\text{down}}$ queries fast, we in addition to $S_n.\texttt{min}$ ($S_n.\texttt{max}$) maintain an array $S_n.\texttt{min}'$ ($S_n.\texttt{max}'$) with the type of the following lemma such that if $S_n.\texttt{min}[c] = (e, j, c)$ ($S_n.\texttt{max}[c] = (e, j, c)$) then $S_n.\texttt{min}'[c] = j$ ($S_n.\texttt{max}'[c] = j$). The lemma is an easy corollary to Fredman and Willard [52]:

Lemma 2.2. *We can maintain an array A indexed by C where each entry in A is an integer in $[0..O(N)]$ such that updates to A can be performed in constant time and such that we given i and j can calculate the set $\texttt{report}(A, i, j) = \{c \in C \mid i \leq A[c] \leq j\}$ in constant time. The space usage of A is $O(|C|)$ and A can be initialized in constant time.*

Proof. We insert the elements of A in the q-heap data structure of [52]. The q-heap can contain at most $O(\log^{\epsilon} N)$ elements for some $\epsilon > 0$, requires a global lookup table of size $O(N)$ which can be constructed in time $O(N)$, and allows updates as well as predecessor and successor queries to be performed in constant time. Using a q-heap and a global lookup table with $O(N)$ entries we can in a single word maintain a table B indexed by the elements in C such that $B[c]$ is the rank of $A[c]$ among the elements in A. Given a query $\texttt{report}(A, i, j)$ we can use the q-heap to find the rank of i and j among the elements in A. The answer to the query only depends on these ranks and on B and thus the answer can be found in constant time using a global lookup table with $O(N)$ entries.□ □

Answering a $\texttt{reportany}_{\text{down}}(S_n, i, j, C')$ query: If $i > j$ or $S_n = \emptyset$ we report nothing. Else, we perform a $\texttt{report}(S_n.\texttt{min}', i, j)$ query and we let C'' be the answer to the query. We then for each color $c \in C' \cap C''$ report the triple $S_n.\texttt{min}[c]$. After this, we set C' equal to $C' \setminus C''$. Next, we report from $S_n.\texttt{max}'$ in a similar way. If $i = 0$ or $j = n-1$ there can be no additional triples to report and we are done. Else, if $i \operatorname{div} \sqrt{n} = j \operatorname{div} \sqrt{n}$ we perform a $\texttt{reportany}_{\text{down}}($ $S_n.\texttt{bottom}[i \operatorname{div} \sqrt{n}], i \bmod \sqrt{n}, j \bmod \sqrt{n}, C')$ query. Finally if $i \operatorname{div} \sqrt{n} \neq j \operatorname{div} \sqrt{n}$ we split the query into the three queries $\texttt{reportany}_{\text{down}}(S_n.\texttt{bottom}[i \operatorname{div} \sqrt{n}], i \bmod \sqrt{n}, \sqrt{n} - 1, C')$, $\texttt{reportany}_{\text{down}}(S_n.\texttt{top}, i \operatorname{div} \sqrt{n} + 1, j \operatorname{div} \sqrt{n} - 1, C')$ and $\texttt{reportany}_{\text{down}}(S_n.\texttt{bottom}[j \operatorname{div} \sqrt{n}], 0, j \bmod \sqrt{n}, C')$ using call-by-reference semantics for the C' parameter. We notice, that the first and the last of these three queries are answered by just looking at the min, min′, min and max′ fields in the recursive structure.

Answering a $\texttt{reportany}(S_n, i, j, C')$ query: We first perform a $\texttt{reportany}_{\text{down}}(S_n, i, j, C')$ query. Let M be the answer to this query. We then iteratively for each triple $(e, t, c) \in M$ where e points to some recursive structure T in our induction on n, replace $(e, t, c) \in M$ with $T.\texttt{min}[c]$. For later use we note, that we could equally well use $T.\texttt{max}[c]$ and that $T.\texttt{max}[c] \neq T.\texttt{min}[c]$. When no more replacements can be done, the set of (e, c) for which there exists t such that $(e, t, c) \in M$ is the answer to the query.

The space usage of S_n is bounded by $O(n \log\log n \log^\epsilon N)$ as required. To see this observe, that the recursion depth is $O(\log\log n)$ and that we use space $O(n \log^\epsilon N)$ in each level of the recursion.

The running time of the update operations in S_n is $O(\log\log n)$. This follows from the observation that when an update operation operates in more than one recursive structure, it does non-constant work in at most one of them. A similar argument shows that the running time of the $\texttt{reportany}_{\text{down}}$ query in S_n is $O(\log\log n + k)$.

We finally argue, that it takes time $O(\log\log n + k)$ to answer the report query in G_n. We first observe, that the reportany query in S_n may need to follow up to $O(\log\log n)$ pointers for each triple reported by the $\texttt{reportany}_{\text{down}}$ query and might therefore use time $O(\log\log n \log^\epsilon N)$ to report $O(\log^\epsilon N)$ triples. The central observation is, that each time we follow a pointer to a substructure T when answering the reportany query in S_n, there is as noted earlier at least two different triples in T we could report and which will be reported by the report query in G_n. This ensures that we stay within the claimed time bound for the report query in G_n concluding our proof of lemma 2.1.

2.4 A data structure for a colored linked list

In this section we use lemma 2.1 to develop the linked list data structure L_n of theorem 2.3 below. We do this using techniques similar to the ones used in [37, 36].

As in section 2.3 we require n is at most $O(N)$ and we further require $|L_n| \leq n$. Each element $e \in L_n$ has an associated set of colors $e.\texttt{Col}$ and we require $e.\texttt{Col} \subseteq C$ and $|e.\texttt{Col}| = O(1)$. We say $e \in L_n$ has color c if $c \in e.\texttt{Col}$. We define $\texttt{pred}(L_n, e_2, c) = \max\{e_1 \in L_n \mid c \in e_1.\texttt{Col}, e_1 \leq e_2\}$, $\texttt{succ}(L_n, e_1, c) = \min\{e_2 \in L_n \mid c \in e_2.\texttt{Col}, e_1 \leq e_2\}$ and $\texttt{report}(L_n, e_1, e_2, C') = \{e \in L_n \mid C' \cap e.\texttt{Col} \neq \emptyset, e_1 \leq e \leq e_2\}$. In this section we show:

Theorem 2.3. *We can maintain L_n using space $O(n)$ such that we in time $O(\log\log N)$ can a) insert a new element without color, b) change the colors of an element and c) perform* pred *and* succ *queries. Finally we can d) perform* report *queries in time $O(\log\log N + k)$.*

We group the elements of L_n into blocks of $\Theta(\log^{3+\epsilon} N)$ elements unless $|L_n| < O(\log^{3+\epsilon} N)$ in which case we make a single block with $|L_n|$ elements. We order the blocks such that if b and b' are blocks, $e \in b$, $e' \in b'$ and $e \leq e'$ in L_n then $b \leq b'$. In each block b we keep a standard balanced binary tree $b.\texttt{tree}$. We use the elements of b as the leafs of $b.\texttt{tree}$ such that if $e, e' \in b$ and $e \leq e'$ in L_n, then $e \leq e'$ in $b.\texttt{tree}$. For each color $c \in C$ we link the leafs with color c

together in a leaf list. We say that an internal node has color $c \in C$ if it has a descendant leaf with color c. Further we say that a block b has color c if the root of b.`tree` has color c. We keep an array b.`allleafs` with $O(\log^{3+\epsilon} N)$ entries such that for each leaf $e \in b$.`tree` there is an entry in b.`allleafs` with a pointer to e. Also, we keep a list b.`freelist` of indices unused in b.`allleafs`. In each internal node $e \in b$.`tree`, we keep an array e.`leafs` indexed by C such that $e.\texttt{leafs}[c] \neq 0$ iff e has color c and such that if e has color c then $b.\texttt{allleafs}[e.\texttt{leafs}[c]]$ is a pointer to a leaf with color c descendant to e. We note, that each of the saved indices is $O(\log\log N)$ bits and thus e.`leafs` can be stored in a single word. It follows that a block uses space linear in the number of elements stored in it. It also follows, that if $e \in b$.`tree` is an internal node, then we can find e.`leafs` in constant time by looking at the children of e. We observe, that by looking at e.`leafs` we can in constant time find the set of colors represented among the leafs descendant to e.

We now need the following theorem showed by Willard [110] by a deamortization the structure of Itai et al. [63]:

Theorem 2.4. *Suppose B is a linked list such that $|B| \leq m$. Further suppose each element $b \in B$ has an integer position $b.\texttt{pos} \leq O(m)$ such that for $b, b' \in B$ we have:*

$$b < b' \implies b.\texttt{pos} < b'.\texttt{pos} \tag{2.1}$$

Then we can maintain B using space $O(m)$ such that insertions can be performed in time $O(\log^2 m)$ by repositioning $O(\log^2 m)$ elements. Further (2.1) can be maintained during the repositioning.

We insert the blocks of L_n into the linked list B of theorem 2.4 selecting $m = n/\log^{3+\epsilon} N$ such that if $b \leq b'$ are blocks then $b \leq b'$ in B. The theorem then assigns a position b.`pos` to each block b. If $n = \Omega(\log^{3+\epsilon} N)$ we keep a single structure G with type $G_{\Theta(n/\log^{3+\epsilon} N)}$ from lemma 2.1 and observe that this uses space $O(n)$. If n is not $\Omega(\log^{3+\epsilon} N)$ then all elements of L_n fit into a single block and we do not keep this structure because it may use more than linear space. In the rest of this section we assume for brevity that $n = \Omega(\log^{3+\epsilon} N)$. For each block b and for each color c of b, we then insert b in G with color c and position b.`pos`.

Answering a $\texttt{pred}(L_n, e, c)$ ($\texttt{succ}(L_n, e, c)$) query: This can be done by first searching bottom up in b.`tree` for a predecessor (successor) of e with color c in the block b in which e is located. This can be done in time $O(\log\log N)$ since each node in b.`tree` has information about which colors are represented among its descendant leafs. If no element is found by this search a $\texttt{pred}(G, b.\texttt{pos}+1, c)$ ($\texttt{succ}(G, b.\texttt{pos}-1, c)$) query is performed to find the block which contains the predecessor (successor) and this is followed by a local search in the found block using time $O(\log\log N)$. It follows that we use total time $O(\log\log N)$.

Answering a $\texttt{report}(L_n, e_1, e_2, C')$ query: This can be done in a somewhat similar way. We will describe a procedure which reports each $e \in L_n$ for which $e_1 \leq e \leq e_2$ one time for each color e has in C'. Since e has at most a constant number of colors we can easily modify the procedure such that e is reported only once. Let b_1 (b_2) be the block e_1 (e_2) is located in. We first perform a local search bottom up in b_1.`tree` and b_2.`tree` to find the elements from b_1 and b_2 which should be reported. This can be done in time $O(\log\log N + k)$ using the `leafs` arrays, the `allleafs` arrays and the leaf lists. Finally, if $b_1 \neq b_2$ we perform the query $\texttt{report}(G, b_1.\texttt{pos}+1, b_2.\texttt{pos}-1, C')$ to find the blocks which contain elements that should be reported. Let b be a reported block inserted in G with color c and let e be the root node of b.`tree`. Then, using the leaf lists starting in $e.\texttt{leafs}[c]$ we can get the elements from b which should be reported in constant time pr. element. It follows that we use total time $O(\log\log N + k)$.

Changing the colors of an element in L_n: Let b be the block the element is located in. We first update b.`tree` bottom up using time $O(\log\log N)$. Also, we update the leaf lists which can also be done in time $O(\log\log N)$ because each node has information about which colors are represented among its descendant leafs and because each node can have at most a constant number of colors. Let C' (C'') be the set of colors b has before (after) the update. For each $c \in C'' \setminus C'$ we insert b with color c and position $b.\texttt{pos}$ in G and for each $c \in C' \setminus C''$ we remove b with color c and position $b.\texttt{pos}$ from G. Since $C'' \setminus C'$ and $C' \setminus C''$ have constant size it follows that the update can be performed in time $O(\log\log N)$.

Before we describe how to insert a new uncolored element in L_n we show corollary 2.6 below. The corollary builds on the following theorem proofed by Dietz and Sleator [37, theorem 5] where $H_n = \sum_{i=1}^{n} 1/i$ and where we observe $H_n = \Theta(\log n)$:

Theorem 2.5. *Suppose $x_1, \dots, x_n$ are variables which are initially zero. Iterate the following two steps: 1) Add a non-negative value to each x_i such that $\sum_{i=1}^{n} x_i$ increases by at most one. 2) Set a largest x_i to zero. Then for all i, $1 \le i \le n$ we always have $x_i \le 1 + H_{n-1}$.*

Corollary 2.6. *Suppose $c \ge 1$ and that $\mathcal{C}$ is a collection of at most n sets which initially contains one empty set. Iterate the following two steps: 1) Add a total of c elements to the sets of $\mathcal{C}$ and 2) If $M \in \mathcal{C}$ is a largest set in $\mathcal{C}$ and $|M| \ge 5c(1 + H_{n-1})$ then split M into two sets of size at least $\lfloor 2c(1 + H_{n-1}) \rfloor$ and replace these with M in $\mathcal{C}$. Then for all $M \in \mathcal{C}$ we always have $|M| \le 6c(1 + H_{n-1})$.*

Proof. Consider the beginning of an arbitrary iteration. Assume $\mathcal{C} = \{M_1, \dots, M_m\}$ and define $M_{m+1}, \dots, M_n = \emptyset$. Define for $1 \le i \le n$:

$$x_i = \max(0, |M_i|/c - 5(1 + H_{n-1})) \tag{2.2}$$

Let $\mathcal{V} = \{x_1, \dots, x_n\}$ be the set containing the n variables defined in (2.2).

It is easy to show by induction on the iterations that the variables in $\mathcal{V}$ will behave as described in theorem 2.5. The lemma then for $1 \le i \le n$ gives $x_i \le 1 + H_{n+1}$ implying $|M_i| \le 6c(1 + H_{n-1})$ as desired.□ □

We now describe how to insert a new uncolored element e in L_n. First, we find the block b where e should be saved using time $O(\log\log N)$. We then remove an index i from $b.\texttt{freelist}$ and insert a pointer to e in $b.\texttt{allleafs}[i]$ using constant time. Finally, we insert e in $b.\texttt{tree}$ using time $O(\log\log N)$. Now set $c = \Theta(\log^{2+\epsilon} N)$. To prevent the blocks from becoming too big we do the following: When a block contains $\lceil 4c(1+H_{n-1}) \rceil$ elements, we set a mark between two neighbor elements in the block such that the number of elements on each side of the mark is at least $\lfloor 2c(1 + H_{n-1}) \rfloor$. Further, for every cth insertion in L_n we take a largest block b and if b has at least $5c(1 + H_{n-1})$ elements, we split b into two blocks b' and b'' at the marked position. From this it follows, that b and b' will both have size at least $\lfloor 2c(1 + H_{n-1}) \rfloor$ and corollary 2.6 thus ensures that no block will contain more than $6c(1 + H_{n-1}) = O(\log^{3+\epsilon} N)$ elements as claimed earlier. It now follows that b' and b'' will have size at most $\lceil 4c(1 + H_{n-1}) \rceil$ when b is split. It follows, that from the time we set a mark in a block to the time the block splits, at least $\Omega(c(1+H_{n-1})) = \Omega(\log^{3+\epsilon} N)$ insertions has to made in the block. During these insertions we build up new arrays $b'.\texttt{allleafs}$, $b'.\texttt{freelist}$, $b''.\texttt{allleafs}$ and $b''.\texttt{freelist}$ to be used by b' and b'' after the split.

To be able to find a largest block to split we keep the blocks in a binary heap using the block sizes as keys. Since there is $O(\log^{3+\epsilon} N)$ different block sizes all heap operations run in time $O(\log\log N)$.

In connection with the split theorem 2.4 requires us to change the position of $O(\log^2 N)$ blocks in order to assign a position to the new block created by the split. If b is a block which has its position changed from $\texttt{pos}$ to $\texttt{pos}'$ we for each color c of b have to remove b with position `pos` and color c from G and we have to insert b with position `pos'` and color c in G. Since there is only $O(\log^{\epsilon} N)$ colors it follows, that we need to perform at most $O(\log^{2+\epsilon} N)$ updates in G. We distribute these updates by performing $O(1)$ updates in G in each of the following $O(\log^{2+\epsilon} N)$ insertions of uncolored elements in L_n. Since we only split a block for every $O(\log^{2+\epsilon} N)$ insertion in L_n, we can finish this work before the next time a block is split. Finally, we need to make adjustments to the way queries are answered because a block may be partially moved in G. But since (2.1) can be maintained during the change of positions, we only need to make adjustments in the case where we ask a query from a block which is currently partially moved. Adjusting the way we answer queries to also work in this case is straight forward. We conclude that insertion in L_n can be performed in time $O(\log \log N)$ as required.

Finally, it follows from our arguments, that the space usage of L_n is $O(n)$ so this finishes our proof of theorem 2.3.

2.5 WBB trees

In this section we review WBB trees (Weight Balanced B-trees) which we use in section 2.6. Our description is very similar to the one in [14].

A WBB tree T with branching parameter $d \geq 6$ is a multi-branching search-tree containing elements used as leafs in T. Each element must have a key from a total order, and we assume that two different elements cannot have the same key. All leafs in T must have the same depth (and height 0). If $v \in T$ is an internal node, we let $l(v)$ be the set of leafs descendant to v. Further, if v has m ordered children $v_1, \ldots, v_m$ we in v keep $m+1$ keys $x_0 < x_1 < \ldots < x_m$ to guide the search. If $1 \leq i \leq m$ and x is a key in a node descendant to v_i then we require $x_{i-1} < x \leq x_i$ We define v'_{x_i} (v''_{x_i}) as the node with ordered children $v_1, \ldots v_i$ ($v_{i+1}, \ldots, v_n$) and keys $x_0, \ldots, x_i$ ($x_i, \ldots, x_n$). We observe that the keys x_0 and x_n are not necessary to support normal search operations in v, and we actually do not need to maintain them. But their formal existence turns out to be useful when we describe our solution to the segment reporting problem.

If the root node of T is not a leaf we require it to have at least two children. For all non-root internal nodes $v \in T$ at height h and all internal nodes $v' \in T$ at height h', we require $d^h < |l(v)|$ and $|l(v')| < 4d^{h'}$. It follows that internal nodes have degree less than $4d$ and all non-root internal nodes have degree greater than $d/4$. From this it follows that if T has n elements then the height of T is $\Theta(\log n / \log d)$.

We now sketch how to insert an element e in T. We will not support deletion of elements. If $|T| = 0$ we just make e the only node of T and we are done. If $|T| = 1$ we create a root node having e and the existing node of T as children and we done. Suppose finally $|T| \geq 2$. We then insert e as a new leaf in T at the appropriate position. Suppose first, that the insertion makes an internal node v at height h fulfill $|l(v)| = 3d^h$. We then *mark* the key x in v which maximizes $\min\{|l(v'_x)|, |l(v''_x)|\}$. We can think of this mark as a warning that v will later split at this key. Note that the insertion of e may cause the marking of a key in any subset of ancestors to e. Suppose next, that the insertion makes an internal node v at height h fulfill $|l(v)| = 4d^h$. It follows from lemma 2.7 below that one (and thus exactly one) key x kept to guide the search in v is marked and we split v into v'_x and v''_x. This split creates a new child to the parent of v. If v has no parent a new root node is created. As with marking the insertion of e may cause any subset of ancestors of e to split. We have:

Lemma 2.7. *If we iteratively insert elements into an initially empty WBB tree with branching parameter d, then it remains a WBB tree, and when we split a node v at height h into v'_x and v''_x we have $d^h < |l(v'_x)| < 3d^h$ and $d^h < |l(v''_x)| < 3d^h$.*

Proof. When we mark the key x in v we have $\min\{|l(v'_x)|, |l(v''_x)|\} \geq \lfloor (3d^h - 4d^{h-1})/2 \rfloor > d^h$ (because $d \geq 6$). This inequality is preserved during the following d^h insertions in $l(v)$ and the lemma follows since we always have $|l(v)| = |l(v'_x)| + |l(v''_x)|$ and since $|l(v)| = 4d^h$ when v is split.□ □

The following lemma summarizes some of the description and analysis above and states central properties of WBB trees:

Lemma 2.8. *Let v be a node at height h in a WBB tree T with branching parameter d and n elements. Then:*

1. *The height of T is $\Theta(\log n/\log d)$.*
2. $|l(v)| = O(d^h)$.
3. *At most one key in v is marked.*
4. *When a key in v is marked, then $\Omega(d^h)$ insertions has to be performed in $l(v)$ before v can split.*
5. *When v is split it is split at the marked key.*

In WBB trees, we can in each internal node v have secondary structures containing all elements in $l(v)$ and at the same time have worst case performance on updates: When a key in v is marked, we can start to build up new secondary structures for the two nodes v will eventually split into. Property 4 in lemma 2.8 ensures, that we have sufficient time to do this and property 5 of the lemma ensures that we know which leafs to put into which of the two new secondary structures.

2.6 The final data structures

We now combine the colored linked list data structure of theorem 2.3 with the WBB tree of section 2.5 to obtain our final data structures: In section 2.6.1 we develop a data structure R_r for the range reporting problem and in section 2.6.2 we sketch how a data structure R_s for the segment reporting problem can be constructed in a similar way.

2.6.1 The range reporting problem

Let R_r be a data structure for the range reporting problem where an element $e \in R_r$ represents the point $(e.\mathtt{x}, e.\mathtt{y})$. We assume for brevity that $e, e' \in R_r$ and $e \neq e'$ implies $e.\mathtt{x} \neq e'.\mathtt{x}$ and $e.\mathtt{y} \neq e'.\mathtt{y}$. For $x_1, x_2, y_1, y_2 \in \mathbb{R}$ define $\mathtt{report}(R_r, x_1, x_2, y_1, y_2) = \{e \in R_r \mid x_1 \leq e.\mathtt{x} \leq x_2, y_1 \leq e.\mathtt{y} \leq y_2\}$. In this section we show:

Theorem 2.9. *We can maintain R_r using space $O(N \log N/\log\log N)$ such that $O(N)$ insertions can be performed in R_r and such that a) insertions and deletions can be performed in time $O(\log N)$ and b)* `report` *queries can be answered in time $O(\log N + k)$.*

For $v \in \{\mathtt{x}, \mathtt{y}\}$ and $M \subseteq R_r$ we define $\mathtt{pred}_v(M, i) = \max_v\{e \in M \mid e.v \leq i\}$ and $\mathtt{succ}_v(M, i) = \min_v\{e \in M \mid i \leq e.v\}$. We first describe a simplified version of R_r which does not directly support insertions or deletions. We create a WBB tree X with branching parameter $\Theta(\log^\epsilon N)$ and thus height $\Theta(\log N/\log\log N)$ using the elements $e \in R_r$ as leafs such that $e.\mathtt{x} \leq e'.\mathtt{x}$ implies $e \leq e'$ in X. In each internal node $v \in X$ we keep a standard balanced binary search tree $v.\mathtt{tree}$ where each internal node has either zero or two children. We use the children of

v as leafs of $v.\texttt{tree}$ and we save the keys kept to guide the search in v (except for the largest and smallest) in the internal nodes of $v.\texttt{tree}$ in the natural way. We observe that the height of $v.\texttt{tree}$ is $O(\log\log N)$ and that we can maintain $v.\texttt{tree}$ such that the rank of a leaf in $v.\texttt{tree}$ among the leafs in $v.\texttt{tree}$ can be found in time $O(\log\log N)$. We color each of vs children with a unique color from C. We keep an array in v mapping the rank of a leaf in $v.\texttt{tree}$ to the color the leaf has when viewed as a child of v. We observe that this array uses $O(\log^\epsilon N \log\log N)$ bits and thus can fit into a single word. In v, we also store a colored linked list data structure $v.\texttt{L}$ with type L_n of theorem 2.3 where n is set to the maximum number of leafs v can span given its height in X. We insert in $v.\texttt{L}$ all elements in $l(v)$ such that if $e, e' \in v.\texttt{L}$ and $e.\texttt{y} \leq e'.\texttt{y}$ then $e \leq e'$ in $v.\texttt{L}$. We give each element $e \in v.\texttt{L}$ the color of the unique child v' of v such that $e \in l(v')$. Finally, we maintain a standard balanced binary search tree Y containing all elements in R_r where we for $e \in R_r$ use $e.\texttt{y}$ as key. We note, that each element $e \in R_r$ is stored in $O(\log N/\log\log N)$ colored linked lists located at the nodes of X. It follows that the space usage of R_r is $O(N \log N/\log\log N)$.

Suppose we are given a query $\texttt{report}(R_r, x_1, x_2, y_1, y_2)$. Using $O(\log N)$ time in X we can find the leafs $l_1, l_2 \in X$ such that $l_1 = \texttt{succ}_x(R_r, x_1)$ and $l_2 = \texttt{pred}_x(R_r, x_2)$. If $l_1 = \perp$ or $l_2 = \perp$ there is no elements to report and we are done. Define $V \subseteq X$ to be the set of internal nodes in X which is an ancestor of l_1 or l_2. We note $|V| = O(\log N/\log\log N)$ and that the root of X is contained in V. Each node $v \in V$ has a unique and possible empty maximal subset T_v of children such that $v' \in T_v$ and $e \in l(v')$ implies $x_1 \leq e.\texttt{x} \leq x_2$. For $v \in V$ let C_v be the set of colors of the nodes in T_v. The set T_v can be found in time $O(\log\log N)$. The answer to the query is then $\cup_{v\in V}\texttt{report}(v.\texttt{L}, \texttt{succ}_y(v.\texttt{L}, y_1), \texttt{pred}_y(v.\texttt{L}, y_2), C_v)$. Observe that $\texttt{succ}_y(v.\texttt{L}, y_1)$ or $\texttt{pred}_y(v.\texttt{L}, y_2)$ may be $\perp$ for some vs and these vs do not contribute with elements to the union.

We now describe how we top-down in X can find the elements $\texttt{succ}_y(v.\texttt{L}, y_1) \in R_r$ for each node $v \in V$ using total time $O(\log N)$. Since $\texttt{pred}_y(v.\texttt{L}, y_2) \in R_r$ can be found in a similar way it follows from theorem 2.3, that we will get a total query time of $O(\log N + k)$ as desired. If $v \in X$ is the root node then $\texttt{succ}_y(v.\texttt{L}, y_1)$ can be found in time $O(\log N)$ by searching in Y. Now, inductively suppose that $v' \in V$ has color c, is a child of $v \in V$ and that we know $\texttt{succ}_y(v.\texttt{L}, y_1) \in R_r$. We then observe, that $\texttt{succ}_y(v'.\texttt{L}, y_1)$ is equal to $\texttt{succ}(v.\texttt{L}, \texttt{succ}_y(v.\texttt{L}, y_1), c)$. Since this query by theorem 2.3 can be evaluated in time $O(\log\log N)$ the total time usage is $O(\log N + k)$ as required.

We now describe how to insert elements in R_r and how R_r should be modified to support insertions. Finding the lists and positions where the element should be inserted can be done top-down in X in a way similar to the way queries are answered. The tricky part is to handle splitting of nodes in X, and we need to make the data structure somewhat more complicated in order to support this. As soon as a node $v' \in X$ is marked with a key x (in the terminology of section 2.5), we start to prepare the split of v'. Assume for brevity, that v' is not the root of X and has parent v. We then select two colors c' and c'' such that no child of v has color c' or c'' and we color v' with c' and c'' such that v' has a total of three colors. We create two empty colored linked lists $v'.\texttt{L}'$ and $v'.\texttt{L}''$ with the same type as $v'.\texttt{L}$. During the following insertions in $v'.\texttt{L}$ we walk through $v'.\texttt{L}$ from one end to the other passing $O(1)$ elements at each insertion in $v'.\texttt{L}$. When we pass an element $e \in v'.\texttt{L}$ we insert e at the end of $v'.\texttt{L}'$ if $e.\texttt{x} \leq x$ and at the end of $v'.\texttt{L}''$ if $e.\texttt{x} > x$. When we insert e in $v'.\texttt{L}'$ ($v'.\texttt{L}''$) we change the color of e to c' (c'') in the lists in v containing e (there may be two lists if v has a marked key). Because of property 4 of lemma 2.8 we can adjust things such that when v' is split the lists $v'.\texttt{L}'$ and $v'.\texttt{L}''$ are finished, and the split can be performed in time $O(\log\log N)$ by splitting $v'.\texttt{tree}$. The two nodes v' is split into is given the colors c' and c'' respectively. The way we answer queries must be slightly modified to handle the described changes: First, let $v' \in X$ be a non-root internal node we visit when answering a $\texttt{report}$ query in R_r and let $v \in X$ be the parent of v'. Earlier we needed to make one $\texttt{succ}$ ($\texttt{pred}$) query in $v'.\texttt{L}$ for the single color of v. Now, we need to make a $\texttt{succ}$ ($\texttt{pred}$) in $v'.\texttt{L}$ for each of the at most three colors of v', and as answer we select the element which is closest to our query point in $v.\texttt{L}$. Second, in the case we insert an element e in $v.\texttt{L}$ at a position before the one we are currently inserting in $v.\texttt{L}'$ and $v.\texttt{L}''$, we must also insert e in the correct of $v.\texttt{L}'$ and $v.\texttt{L}''$. Finally we note, that we still only use $O(\log\log N)$ time when visiting a node in X. It follows that answering a $\texttt{report}$ query in R_r still takes time $O(\log N + k)$ and that inserting an element in R_r takes time $O(\log N)$.

We now describe how to handle deletions. We delete an element $e \in R_r$ by removing the color it has in each of the $O(\log N/\log\log N)$ lists it is located in. This will use a total time of $O(\log N)$. Also, we modify the insert operation so that when it inserts an element in $v.\texttt{L}'$ or $v.\texttt{L}''$ as described above, it does not give an uncolored element a color. This concludes our proof of theorem 2.9.

2.6.2 The segment reporting problem

Let R_s be the data structure for the segment reporting problem where an element $e \in R_s$ represents the line segment from $(e.\texttt{x}_1, e.\texttt{y})$ to $(e.\texttt{x}_2, e.\texttt{y})$. We assume for brevity that for $e \in R_s$ we have $e.\texttt{x}_1 < e.\texttt{x}_2$ and that no two segments in R_s share endpoints. We define $\texttt{report}(R_s, x, y_1, y_2) = \{e \in R_s \,|\, e.\texttt{x}_1 \le x \le e.\texttt{x}_2, y_1 \le e.\texttt{y} \le y_2\}$ In this section we give a proof sketch for the following theorem:

Theorem 2.10. *We can maintain R_s using space $O(N\log N/\log\log N)$ such that $O(N)$ insertions can be performed in R_s and such that a) insertions and deletions can be performed in time $O(\log N)$ and b)* $\texttt{report}$ *queries can be answered in time $O(\log N + k)$.*

We create a WBB tree X with branching parameter $\Theta(\log^{\epsilon/2} N)$ and thus height $\Theta(\log N/\log\log N)$. We use each element in R_s two times as a leaf in X namely one time representing its left and one time representing its right endpoint. Further, if $e, e' \in X$ are leafs and the endpoint represented by e is to the left of the endpoint represented by e', we require $e < e'$ in X. As in the range reporting problem, we keep in each internal node $v \in X$ a tree $v.\texttt{tree}$.

We color the children of each internal node $v \in X$ with a unique color from C. We keep a list $v.\texttt{L}$ with the type of theorem 2.3. We insert in $v.\texttt{L}$ all elements in $l(v)$ such if $e, e' \in v.\texttt{L}$ and $e.\texttt{y} \le e'.\texttt{y}$ then $e \le e'$ in $v.\texttt{L}$. If there is two identical line segments among the leafs descendant to v, then only one is inserted in $v.\texttt{L}$. We color an element $e \in v.\texttt{L}$ with the color of a child v' of v if $e \in l(v')$. We observe that each element in $v.\texttt{L}$ is given at most two colors by this. Finally, we keep a standard binary search tree Y which contains all elements in R_s using the y-values as keys.

For each node $v \in X$ we color each pair of keys stored to guide the search in v with a unique color from C not used as the color of a child of v. Since v only has $O(\log^{\epsilon/2})$ children we have sufficient colors to do this. Consider a line segment $e \in v.\texttt{L}$ and let x_1 and x_2 be the keys stored to guide the search in v such that $[x_1 \ldots x_2] \subseteq [e.\texttt{x}_1 \ldots e.\texttt{x}_2]$ has maximal length. Then we in $v.\texttt{L}$ color e with the color representing the pair (x_1, x_2).

Now suppose we are given a query $\texttt{report}(R_s, x, y_1, y_2)$. We then first search for y_1 and y_2 in Y and then we proceed down through the path $V \subseteq X$ which goes from the root of X to the leaf determined by x in way similar to when we answer queries for the range reporting problem. In each internal node $v \in V$ such that $v' \in V$ is a child of v, we report all line segments $e \in v.\texttt{L}$ such that $y_1 \le e.\texttt{y} \le y_2$ and such that $e.\texttt{col}$ contains a color representing an interval containing the keys of all elements in $l(v')$.

The time and space usage for R_s is identical to the time and space usage for R_r. Further, the described structure can be extended to support updates so this completes our proof sketch for theorem 2.10.

2.7 Acknowledgments

I would like to thank my supervisors Stephen Alstrup and Theis Rauhe for introducing me to the range reporting problem, and for giving many useful comments to earlier drafts of this paper.

Chapter 3

Fully-dynamic orthogonal range reporting on RAM

Fully-dynamic orthogonal range reporting on RAM *

Christian Worm Mortensen†

Abstract

We show that there exists a constant $\omega < 1$ such that the fully-dynamic d-dimensional orthogonal range reporting problem for any constant $d \geq 2$ can be solved in time $O(\log^{\omega+d-2} n)$ for updates and time $O((\log n/\log\log n)^{d-1} + r)$ for queries. Here n is the number of points stored and r is the number of points reported. The space usage is $O(n \log^{\omega+d-2} n)$. For $d = 2$ our results are optimal in terms of time per operation and this is the main contribution of this paper. Also for $d = 2$, we give a new improved fully-dynamic structure supporting 3-sided queries. The model of computation is a unit cost RAM. We order the coordinates of points using *list order* as defined in the paper.

Key words. range searching, data structures, orthogonal

AMS subject classifications. 68P05, 68P10, 68P15

3.1 Introduction

Consider a database R of persons where each person has attributes such as an age, a weight and a height. One of the most fundamental type of queries in the database is to ask for the set of persons in R who have an age between, say, 30 and 50 years, a weight between 80 and 90 kg and a height between 160 and 170 cm. Data structures supporting this kind of queries are called orthogonal range reporting structures and have been widely studied during the last 30 years. For surveys see Agarwal and Erickson [1] and Chiang and Tamassia [32]. In this paper we give new upper bounds for the fully-dynamic variant of this problem where we in an online setting allow queries to be intermixed with updates which can insert a new person or delete an existing person. In case each person has exactly two attributes our upper bound is optimal in terms of time per operation by a lower bound of Alstrup, Husfeldt and Rauhe [8].

3.1.1 Problem definition

In the d-dimensional fully-dynamic orthogonal range reporting problem (henceforth the dynamic d-dimensional range reporting problem or just the d-dimensional range reporting problem) we must maintain a set R of at most $O(n)$ d-dimensional points under insertions and deletions. For each point $(x_1, \ldots, x_d) \in R$ we must have $x_i \in L_i$ for $1 \leq i \leq d$ where L_i is an ordered set. Given a query $(x'_1, x''_1, \ldots, x'_d, x''_d) \in L_1 \times L_1 \times \cdots \times L_d \times L_d$ we must report the set

*This paper is based on [79] and [78]. The result of [79] for the orthogonal segment intersection problem is not covered in this paper. Also, we use another method for extending into higher dimensions than in [78].

†IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark (cworm@itu.dk)

$\{(x_1, \ldots, x_d) \in R \mid x'_1 \leq x_1 \leq x''_1 \wedge \ldots \wedge x'_d \leq x_d \leq x''_d\}$. We assume that if $(x_1, \ldots, x_d) \in R$ and $(y_1, \ldots, y_d) \in R$ are two different points then $x_1 \neq y_1 \wedge \ldots \wedge y_d \neq y_d$. Our model of computation is a unit cost RAM with word size at least $\log n$ bits. We define the *update time* to be the time it takes to insert a point into or delete a point from R. We only consider solutions to the problem where the time to answer a query has the form $Q + O(r)$ where r is the number of reported points and Q is independent of r, and we say such a solution has *query time* Q. Further, we call the maximum of the update time and the query time the *time per operation*.

We get different problems depending on what we select as L_i. Traditionally, one is only allowed to compare the elements of L_i using a unit cost comparison operation. This is often specified as $L_i = \mathbb{R}$ and makes good sense on a pointer machine. We call this the *comparison order* variant of our problem. Another possibility is to take L_i to be the set of non-negative integers smaller than m (we assume here $m \leq 2^w$ where w is the word size). We define this as the *m-order* variant. We define the *word-order* variant to be the m-order variant with $m = 2^w$. All these variants have their shortcomings. In the comparison order variant, there is a trivial lower bound of $\Omega(\log n)$ on the time per operation because this is a lower bound for the dictionary problem on $\mathbb{R}$. In the m-order variant, it is not always possible to create a new attribute between two existing attributes when performing an insertion. For these reasons, we introduce in this paper what we define as the *list order* variant. For any linked list L we order the elements of L such that if $e, e' \in L$ then $e < e'$ if $e \neq e'$ and e appears before e' in L. In the list order variant, we then let L_i be a linked list and we require each element of L_i to be the coordinate of exactly one point in R. When a point is inserted in (resp. deleted from) R a new element is then inserted in (resp. deleted from) each L_i for $1 \leq i \leq d$. Using a result by Dietz and Sleator [37] (cited as Theorem 3.8 in Section 3.5.3) a structure for the comparison order variant can be turned into a structure for the list order variant with the same performance. Conversely, a structure for the list order variant can be turned into a structure for the comparison order variant if we add a term of $O(\log n)$ to the update and query time since such a term allows us to maintain a balanced search tree with the elements of each L_i. Finally, by the use of Theorem 3.6 cited in Section 3.5.1, a structure for the list order variant can be turned into a structure for the word order variant if we add a term of $O(\sqrt{\log n / \log \log n})$ to the update and query time.

3.1.2 Our results

We show (Theorem 3.32) that there exists a constant $\omega < 1$ such that for any constant $d \geq 2$ the d-dimensional range reporting problem in the list order variant can be solved with update time $O(\log^{\omega+d-2} n)$, query time $O((\log n / \log \log n)^{d-1})$ and space $O(n \log^{\omega+d-2} n)$. The time bounds are worst case and the solution is deterministic. In the comparison order variant, the discussions of Section 3.1.1 imply that the same result hold for $d \geq 3$ and that for $d = 2$ the time per operation becomes $\Theta(\log n)$.

For $d = 2$ we also consider queries of the restricted form $(x'_1, x''_1, x'_2, x''_2) \in L_1 \times L_1 \times L_2 \times L_2$ where x'_2 is the minimal element of L_2. Such queries are called *3-sided* whereas queries where x'_2 can take any value in L_2 are called *4-sided*. We show (Theorem 3.24) that the 2-dimensional range reporting problem in the list order variant with 3-sided queries only, can be solved with update time $O(\log^\omega n)$, query time $O(\log n / \log \log n)$ and space $O(n)$. Also here, the time bounds are worst case and the solution is deterministic.

The data structure of this paper are too complicated and the constants involved to large to be practical useful. However, some of the ideas of the paper may still have practical implications (see Section 3.9 on open problems).

3.1.3 Relation to other work

The static 2-dimensional range reporting problem in the comparison order variant can be solved with query time $O(\log^2 n)$ and space $O(n \log n)$ using *range trees*. The range tree data structure was independently discovered by Bentley, Lee, Lueker, Shamos, Willard and Wong [19, 16, 17, 69, 70, 106, 108]. It is commonly recognized that Bentley and Shamos were the chronologically first authors to publish their results [19, 16]. In a dynamic setting, their data structure was known to have an $O(\log^3 n)$ query time and an $O(\log^2 n)$ update time using some version of the Logarithmic update method—sometimes called the static-to-dynamic transformation [18, 87, 89]. It was observed by Lueker and Willard (first independently and then in a joint paper [70, 106, 113]) that the query time of this dynamic method for range trees could be reduced to $O(\log^2 n)$ time (with no sacrifice in update time) using the BB(α) dynamic method. Willard [108] observed how to extend range trees with "downpointers" so that the query time could be further reduced to $O(\log n)$ time in the static case. The downpointer discovery by Willard [108] should not be confused with the BB(α) dynamic method (that was jointly co-authored by Lueker and Willard [70, 106, 113]). Because the two discoveries of the BB(α) dynamic method and the downpointer were made nearly simultaneously, several articles reviewing the prior literature have often inadvertently confused them.

Chazelle and Guibas [30] generalized downpointers as well as other ideas into a data structuring technique called fractional cascading. Mehlhorn and Näher [75] made fractional cascading dynamic with amortized time bounds on updates. They used this to develop a data structure for the 2-dimensional range reporting problem in the comparison order variant with update and query time $O(\log n \log\log n)$ on a pointer machine. Dietz and Raman [36] removed amortization from the results of [75] on a RAM. Using range trees the structure from [75] can be extended to $d > 2$ dimensions giving update and query time $O(\log^{d-1} n \log\log n)$ and usage space $O(n \log^{d-1} n)$. Until now, this was the fastest known solution in terms of time per operation (even in the n-order variant). Range trees with slack parameter [74, 103] can be used to get a structure with poly-logarithmic update and query time and with space usage $O(n(\log n/\log\log n)^{d-1})$. Until now, no solution with space usage $o(n(\log n/\log\log n)^{d-1})$ and poly-logarithmic time per operation was known (see end of paragraph for very recent work). For $d = 2$ a solution with linear space usage $O(n)$, update time $O(\log n)$ and query time $O(n^\epsilon)$ for a constant $\epsilon > 0$ is known [67]. Also, for $d = 2$, if only insertions or deletions (but not both) are supported Imai and Asaon [62] described a solution using space $O(n \log n)$ and time $\Theta(\log n)$ per operation. Very recently, Nekrich [86] dynamized a structure by Alstrup, Brodal and Rauhe [6] and obtained a structure for the d-dimensional dynamic range reporting problem with update time $O(\log^d n)$, query time $O(\log^{d-1} n)$ and space usage $O(n \log^{d-2+\epsilon})$ for any constant $\epsilon > 0$.

Let U be the update and Q be the query time of an orthogonal range reporting structure. For word size poly-logarithmic in n Alstrup, Husfeldt and Rauhe [8] showed that $Q = \Omega(\log n/\log(U \log n))$. This lower bound holds in the cell probe model of computation (which is stronger than the RAM model), for the amortized cost per operation, for 3-sided queries, and for the n-order variant. It follows that with poly-logarithmic update time, the query time becomes $\Omega(\log n/\log\log n)$. Together with our new upper bound, this lower bound gives that the time per operation for the 4-sided 2-dimensional range reporting problem in the list order variant and the m-order variant is $\Theta(\log n/\log\log n)$ for $m \geq n$. As already mentioned our results also imply a bound of $\Theta(\log n)$ in the comparison order variant. The time per operation for the 2-dimensional range reporting problem in all the order variants of Section 3.1.1 is thus now completely understood and this is the major contribution of this paper.

McCreight [72] developed the priority search tree which solves the 2-dimensional range reporting problem in the comparison order variant with 3-sided queries only on a pointer machine. The update and query time of the priority search tree is $O(\log n)$ and it uses space $O(n)$. Willard [112] modified the priority search tree and obtained a structure using time $O(\log n/\log\log n)$ for updates and queries in the word order variant on a RAM. In this paper, we reduce the update time for this problem further without increasing the query time or the space usage. Also, we generalize the result to the list order variant. As Willard, we use a modified version of the priority search tree.

A long standing open problem mentioned by McCreight [72] is whether it is harder to make a fully-dynamic structure supporting 4-sided than 3-sided queries. This paper together with the lower bound of [8] partly answers this question by showing that on the RAM the time per operation needed for the two problems is the same in all the order variants described in Section 3.1.1. Whether the space usage for structures supporting 4-sided queries can be reduced to linear as for 3-sided queries is still open. This is even the case in the static case where the best known structures with poly-logarithmic query time and constant time for each reported point uses space $O(n \log^{\epsilon} n)$. The query time is $O(\log n)$ for the comparison order variant [26] and $O(\log\log n)$ for the n-order variant [6].

On the pointer machine in the comparison order variant, Chazelle [27] has shown that any static or dynamic structure for the 2-dimensional range reporting problem with poly-logarithmic query time, must use space $\Omega(n(\log n/\log\log n)^{d-1})$. For $d = 2$ this bound is matched by a static structure with query time $O(\log n)$ and space usage $O(n\log n/\log\log n)$ [25]. For $d > 2$ the best known upper bound with matching space usage has query time $O(\log^{d-1+\epsilon} n)$ for any constant $\epsilon > 0$ also in the static case [27].

In the I/O model of computation with block size B and in 2 dimensions, a structure for 3-sided queries using space $O(n/B)$ disk blocks supporting updates in time $O(\log_B n)$ and queries in time $O(\log_B n + r/B)$ when r points are reported is described by Arge, Samoladas and Vitter [13]. This is optimal in terms of time per operation in the comparison order variant. For 4-sided queries they describe a solution supporting updates in time $O((\log_B n)\log(n/B)/\log\log_B n)$, queries in time $O(\log_B n + r/B)$ when r points are reported. The structure uses space $O((n/B)$ $\log(n/B)/\log\log_B n)$ disk blocks.

3.1.4 Outline of paper

After this introduction we continue with preliminaries in Section 3.2. In Section 3.3 we give various definitions related to two-dimensional range reporting. Also, we give an outline of how the results of this paper are obtained. In Section 3.4 we give the probably most central construction of this paper and this construction may be of independent interest. In Section 3.5 we review various wellknown data structures which we use in this paper. Also, we develop some new results on top of these structures. In Section 3.6 we develop our solution to the 2-dimensional range reporting problem with 3-sided queries only. In Section 3.7 we develop our solution to the 4-sided 2-dimensional range reporting problem. In Section 3.8 we extend the solution of Section 3.7 to higher dimensions. The paper concludes with open problems in Section 3.9 and acknowledgments in Section 3.10.

3.2 Preliminaries

For the rest of this paper we define $[i \dots j] = \{k \in \mathbb{Z} | i \le k \le j\}$ and we let log denote the base-two logarithm. We will measure space in bits. We will use $\log^i \log n$ as an abbreviation for $(\log\log n)^i$. All time bounds are worst case and all structures are deterministic if not otherwise noted.

A linked list is a list where the elements are doubly linked with pointers. We make no assumptions about how the elements are laid out in memory. If the list supports insertions, we can insert a new element in it if we have a pointer to the existing element the new element should be inserted before or after. If the list supports deletions we can delete an element if we have a pointer to it. We assume the elements of a linked list and the children of a node in a tree are ordered from left to right.

We will not distinguish between a data structure and the set of elements it contains. As a consequence, $|G|$ is the number of elements in the data structure G and $e \in G$ means that e is an element in G.

A function f is said to be *subadditive* if for integers $m, n \geq 0$ we have $f(m) + f(n) \leq f(m+n) + O(1)$. Suppose for each $n \geq 0$ we have a data structure G_n which uses space $f(n)$ bits and which can contain at most n elements. It follows (by induction on k) that if f is subadditive, then storing the elements of G_n in k different structures $G_{n(1)}, \ldots, G_{n(k)}$ uses space at most $f(n) + O(k)$ bits where $n = \sum_{i=1}^{k} n(i)$.

We will assume that the number of points which can be contained in the range reporting structures we ultimately design is bounded by $O(N)$. We note that N was called n in the introduction and that we have assumed a word size of at least $\log N$ bits. All the data structures we create in this paper are a part of one of our ultimate structures and can contain at most $O(N)$ elements. When constructing such a data structure, we often state that we can compute some function f from $[0 \ldots O(N)]$ to $[0 \ldots O(N)]$ in constant time using a *global lookup table*. What we mean by this is the following. As a part of the ultimate structure the data structure is a part of, we keep a table of size at most $O(N \log N)$ bits which is constructable in time $O(N)$. The function f can then be computed by performing a constant number of lookups in this table. We are allowed to assume the existence of such tables by using global rebuilding of Overmars [87]. For brevity, we will omit the details in constructing and looking up in these tables.

3.3 Definitions and outline of constructions

The constructions of this paper involve many different kinds of range reporting structures. In Section 3.3.1 we will define notation which captures almost all the kinds of 2-dimensional structures we look at. We will use this notation through the rest of the paper. Next, in Section 3.3.2 we will give an outline of the constructions that lead to the results of Section 3.1.2.

3.3.1 Definitions

We now define what it means for a range reporting structure R to have type $\mathrm{R}^\tau(t_\mathrm{x} : n_\mathrm{x}, t_\mathrm{y} : n_\mathrm{y})$ where n_x and n_y are integers, $\tau \in \{3, 4\}$ and $t_\mathrm{x}, t_\mathrm{y} \in \{\mathrm{d}, \mathrm{s}, \mathrm{s}'\}$. R must have an *x-axis* $R.L_\mathrm{x}$ and a *y-axis* $R.L_\mathrm{y}$. The rest of this section is parametrized over a parameter $z \in \{\mathrm{x}, \mathrm{y}\}$. Each sentence of this section that contains a z should then be read twice: first with $z = \mathrm{x}$ and then with $z = \mathrm{y}$. So we can state which axes R must have as follows. R must have a *z-axis* $R.L_z$. R contains elements $e \in R$ which are also called points. A point $e \in R$ has a z-coordinate $e.z \in R.L_z$.

If $t_z = \mathrm{d}$ we say the z-axis is *dynamic* and in this case, the z-axis is a linked list with $O(n_z)$ elements. Also, when $t_z = \mathrm{d}$ we require that each element of $R.L_z$ is the z-coordinate of exactly one point. If $\tau = 4$ the structure supports 4-sided queries and if $\tau = 3$ the structure support 3-sided queries as defined in Section 3.1.2. We will make this more precise in a moment. As an example, a structure with type $\mathrm{R}^4(\mathrm{d} : n, \mathrm{d} : n)$ is a 4-sided 2 dimensional range reporting structure in the list order variant and the structure can contain $O(n)$ points.

If $t_z = \mathrm{s}$ or $t_z = \mathrm{s}'$ we say the z-axis is *static* and in this case, $R.L_z$ is the set $[0 \ldots \Theta(n_z)]$ (which is not required to be explicitly stored). If $t_z = \mathrm{s}$ there can be at most one point in R with a given z-coordinate; if $t_z = \mathrm{s}'$ we have no such restriction. We require that no two points share the same x- and y-coordinate. More precisely we require that $e, e' \in M$ and $e.\mathrm{x} = e.\mathrm{y}$ and $e'.\mathrm{x} = e'.\mathrm{y}$ implies $e = e'$. As an example, a structure with type $\mathrm{R}^4(\mathrm{s}\!:\!n, \mathrm{s}\!:\!n)$ can contain at most $O(n)$ points and no two points can share an x- or y-coordinate. On the other hand, a structure with type $\mathrm{R}^4(\mathrm{s}'\!:\!n, \mathrm{s}'\!:\!n)$ can contain $O(n^2)$ points.

We support updates in the form of inserting a new point into R and deleting an existing point from R. From our requirements above it follows that if $t_z = \mathrm{d}$ we will then also have to insert an element in and delete an element from $R.L_z$ respectively. For $x_1, x_2 \in R.L_\mathrm{x}$ and $y_1, y_2 \in R.L_\mathrm{y}$ we support a query (x_1, x_2, y_1, y_2) which must report the set $\{e \in R | x_1 \le e.\mathrm{x} \le x_2 \wedge y_1 \le e.\mathrm{y} \le y_2\}$. We say that the points in this set are in the *query region*. If $\tau = 3$ we require that y_1 in such a query can be assumed to be the minimal element of $R.L_\mathrm{y}$. An implication of our definitions is that for $\tau = 4$ structures of type $\mathrm{R}^\tau(\mathrm{s}\!:\!n, \mathrm{d}\!:\!n)$ and $\mathrm{R}^\tau(\mathrm{d}\!:\!n, \mathrm{s}\!:\!n)$ are identical modulo renaming of axes while this is not the case for $\tau = 3$.

Suppose R can contain at most n points for n small compared to N. In case R has static axes, we will sometimes allow updates to be performed in subconstant time per update by packing several updates into a single word. In this case, the user must to each point $e \in R$ assign an unique identifier or just *id* $e.\mathrm{id}$ of size $O(\log n)$ bits. For each point e inserted into R the user must then in the words describing the update provide the triple $(e.\mathrm{x}, e.\mathrm{y}, e.\mathrm{id})$. For each point $e \in R$ deleted from R, the user must, again in the words describing the update, provide the tuple $(e.\mathrm{x}, e.\mathrm{y})$. Finally, in connection with queries the id of the reported points are given to the user (using constant time per point reported).

We say R has *performance* (U, Q, S, t) if R supports $O(t)$ updates in time $O(U)$, has query time $O(Q)$ and has uses space $O(S)$ bits. S is only allowed to depend on the type of R and not on the number of points stored in R. It follows that if R can contain n points we must have $S = \Omega(n \log n)$. We write (U, Q, S) as an abbreviation for $(U, Q, S, 1)$.

3.3.2 Outline of constructions

We now have the terminology to give an outline of the constructions leading to the results of Section 3.1.2. Our construction consists of a number of steps which we will refer to during the paper. First, we give on outline of how we prove Theorem 3.24. This theorem gives our structure with type $\mathrm{R}^3(\mathrm{d}\!:\!N, \mathrm{d}\!:\!N)$ proving the second result in Section 3.1.2. All the steps O1 to O4 are performed in Section 3.6.

- O1. First, we create a range reporting structure for the case we have only few points. For sufficiently small u compared to N (we have $u = 2^{\log^\epsilon N}$ for a constant $\epsilon > 0$) the structure has type $\mathrm{R}^3(\mathrm{s}\!:\!u, \mathrm{s}\!:\!u)$, has subconstant update time and small query time. The result of this step is in Lemma 3.20.
- O2. We then give the structure of step O1 a dynamic y-axis transforming it into a structure with type $\mathrm{R}^3(\mathrm{s} : u, \mathrm{d} : u)$. What we loose is the subconstant update time; this structure has only constant update time. The query time is unchanged. The technique used is to assign a label of size $O(u)$ to each element on the y-axis and then to relabel elements on insertions. The result of this step is in Lemma 3.22.
- O3. By creating a fixed-shape tree (actually, a variant of a priority search tree) with a structure of step O2 in each internal node, we get a structure with type $\mathrm{R}^3(\mathrm{s}\!:\!n, \mathrm{d}\!:\!n)$ for any $n \ge u$. The result of this step is in Lemma 3.23.
- O4. We then create our final structure with type $\mathrm{R}^3(\mathrm{d} : n, \mathrm{d} : n)$. This is done by grouping points on the x-axis into blocks with a poly-logarithmic number of points in each block.

The blocks are then numbered respecting the order of the blocks, and then they are inserted into the structure of step O3 using these numbers. The result of this step is in Theorem 3.24.

Next, we give an outline of how we prove Theorem 3.32. This theorem gives a structure with type $\mathrm{R}^4(\mathrm{d}\!:\!N, \mathrm{d}\!:\!N)$ and a similar structure for higher dimensions. This is the first result from Section 3.1.2. Step O5 to O10 are performed in Section 3.7 while step O11 is performed in Section 3.8.

- O5. We extend the construction in step O1 to give a structure with type $\mathrm{R}^4(\mathrm{s}'\!:\!u, \mathrm{s}\!:\!u)$. As in step O1 this structure has subconstant update time and small query time. The result of this step is in Lemma 3.25.
- O6. We use the same construction as in step O2, to get a dynamic x-axis. More precisely, we get a structure with type $\mathrm{R}^4(\mathrm{d}\!:\!u, \mathrm{s}'\!:\!u)$. The result of this step is in Lemma 3.26.
- O7. By maintaining a predecessor structure, we transform this into a structure with a static but long x-axis. The structure we get has type $\mathrm{R}^4(\mathrm{s}'\!:\!n, \mathrm{s}\!:\!u)$ for any n, $u \le n \le N$. The result of this step is in Lemma 3.27.
- O8. We then use the construction of Section 3.4 to give the structure of step O7 type $\mathrm{R}^4(\mathrm{s}'\!:\!n, \mathrm{s}'\!:\!u)$ almost without reducing its query time or increasing its update time. The result of this step is in Lemma 3.28.
- O9. We then give this structure a dynamic x-axis creating a structure R with type $\mathrm{R}^4(\mathrm{d}\!:\!n, \mathrm{s}'\!:\!u/\log^3 n)$. This is done by grouping points into blocks of size $O(u)$ on the x-axis. In each block, we keep a structure of step O6. As in step O4 we maintain a numbering of the blocks and we insert each block into the structure of step O8 using the numbers as x-coordinate. The result of this step is in Lemma 3.29.
- O10. We then construct our structure with type $\mathrm{R}^4(\mathrm{d}\!:\!n, \mathrm{d}\!:\!n)$. This is done by constructing a variant of a range tree using the structure of step O9 in some nodes and the structure of step O4 in some nodes of the tree. The result of this step is in Theorem 3.31.
- O11. Finally, we extend the structure of step O10 to higher dimensions. The result of this step is in Theorem 3.32.

3.4 From few to many points

In this section, we give the construction used in step O8 in Section 3.3.2. As mentioned earlier, this is probably the most central construction of this paper. The construction allows us to create a 4-sided range reporting structure which can contain many points from one that can contain few points. The construction has applications besides the one in this paper [80, 79] and for this reason, we will give a slightly more general form of the construction than we need here.

We now define what it means for a data structure G to have type $(n, Y, \mathcal{Y})$ where n is an integer and $\mathcal{Y}$ is a set of subsets of Y. G contains elements $e \in G$ where $e.\mathrm{x}$, $0 \le e.\mathrm{x} < n$ is the *position* of e and $e.\mathrm{y} \in Y$ is the *height* of e. Two different elements in G are not allowed to have both the same position and height implying $|G| \le n|Y|$. We are allowed to update G by inserting a new element or by deleting an existing element. Further, for $0 \le i, j < n$ and $q \in \mathcal{Y}$ we can ask the query $\texttt{report}(G, i, j, q)$ which must report the set $\{e \in G \,|\, i \le e.\mathrm{x} \le j \wedge e.\mathrm{y} \in q\}$. As in the rest of the paper, the query time is said to be $O(Q)$ if reporting this set takes time $O(Q)$ plus constant time for each reported point. We then show:

Theorem 3.1. *Suppose we have a data structure G' with type $(n, Y, \mathcal{Y})$ and the restriction:*

$$e, e' \in G' \wedge e.\mathrm{y} = e'.\mathrm{y} \implies e = e' \tag{3.1}$$

Suppose further G' uses space $O(S)$ bits, has query time $O(Q)$, and update time $O(U)$. Then we can make a structure G with type $(n, Y, \mathcal{Y})$ without the restriction (3.1) which uses space $O(Sn \log\log n)$ bits, has query time $O(Q \log\log n)$ and update time $O(U \log\log n)$.

Observe that since G' can contain $|Y|$ elements we must have $S = \Omega(|Y| \log n)$. In this paper, we only use the following corollary to the theorem:

Lemma 3.2. *Suppose there exists a structure with type $\mathrm{R}^4(\mathrm{s}'{:}\,n, \mathrm{s}{:}\,u)$ and performance (U, Q, S). Then there exists a structure with type $\mathrm{R}^4(\mathrm{s}'{:}\,n, \mathrm{s}'{:}\,u)$ and performance $(U \log\log n, Q \log\log n, Sn \log\log n)$.*

Proof. The lemma is just another way of phrasing Theorem 3.1 if we select $Y = [0 \dots O(u)]$ and $\mathcal{Y} = \{[y_1 \dots y_2] \mid 0 \le y_1, y_2 \le O(u)\}$. □

The rest of this section is devoted to the proof of Theorem 3.1. We need the following theorem which is shown by van Emde Boas et al. [105, 104].

Theorem 3.3. *Let w be the word size in bits. There exists a data structure called a VEB which can maintain a collection of $n \le U \le 2^w$ elements having keys in $[0 \dots U]$ which uses space $O(U \log U)$ bits and supports updates as well as predecessor and successor queries in time $O(\log\log U)$.*

We create a VEB of Theorem 3.3 for each element $y \in Y$. Each element $e \in G$ is inserted at position $e.\mathrm{x}$ in the VEB for height $e.\mathrm{y}$. Using these VEBs we link all elements with the same height together in order according to their positions. This will use space $O(n|Y| \log n) = O(Sn)$ and insertions and deletions can be performed in time $O(\log\log n)$.

We maintain a data structure S_n which we develop in the following. The data structure S_n contains triples $(e, x, y) \in S_n$ where e is an arbitrary pointer, x is an integer in $[0 \dots n-1]$, and $y \in Y$ is the height of the triple. We maintain S_n such that $(e, e.\mathrm{x}, e.\mathrm{y}) \in S_n$ iff $e \in G$. We support a special $\texttt{reportsub}(S_n, i, j, q)$ query which has the following properties:

1. $\texttt{reportsub}(S_n, i, j, q) \subseteq \texttt{report}(G, i, j, q)^*$.
2. If for given y there exists an element $e \in \texttt{report}(G, i, j, q)$ with $i \le e.\mathrm{x} \le j$ and $e.\mathrm{y} = y$ then $\texttt{reportsub}(S_n, i, j, q)$ contains at least one such e.
3. S_n has query time $O(Q)$.

A $\texttt{report}(G, i, j, q)$ query can then be answered as follows. First we perform a $\texttt{reportsub}(S_n, i, j, q)$ query. Property 1 and 3 ensures that we will not use too much time on this. For each height $y \in Y$ for which there exists an element $e \in \texttt{reportsub}(S_n, i, j, q)$ with height y we follow the pointers from e maintained by the VEB for height y to report the rest of the elements with this height. Property 2 ensures that this will report all elements.

We now describe the structure S_n. To avoid tedious details, we assume n has the form $n = 2^{2^m}$ for an integer $m \ge 0$. We observe that if $n > 2$ has this form, then $\sqrt{n}$ has same form. The structure S_n is somewhat similar to a VEB, and we define it inductively on n. If $S_n = \emptyset$ the recursion stops. Else, we keep an array $S_n.\texttt{min}$ (resp. $S_n.\texttt{max}$) indexed by Y. For each $y \in Y$ we store the triple $(e, x, y) \in S_n$ with minimal (resp. maximal) value of x in $S_n.\texttt{min}[y]$ (resp. $S_n.\texttt{max}[y]$) if any. We also keep an array $S_n.\texttt{bottom}$ indexed by $[0 \dots \sqrt{n} - 1]$ where in each entry we store a recursive structure with type $S_{\sqrt{n}}$. We store each triple $(e, i, y) \in S_n \setminus (S_n.\texttt{min} \cup S_n.\texttt{max})$ as $(e, i \bmod \sqrt{n}, y)$ in $S_n.\texttt{bottom}[i \operatorname{div} \sqrt{n}]^\dagger$. Finally, we keep a single recursive structure $S_n.\texttt{top}$ also with type $S_{\sqrt{n}}$. If for $y \in Y$ the structure $S_n.\texttt{bottom}[i]$ contains

*In Section 3.4 we will not distinguish between a query and its answer

†For integers i and d we define $i \bmod d$ and $i \operatorname{div} d$ to be the unique integers such that $0 \le i \bmod d < d$ and $i = d(i \operatorname{div} d) + i \bmod d$.

exactly one triple $(e, x, y) \in S_n$ with height y, we store (e, i, y) in $S_n.\mathtt{top}$. We note that in this case e is stored in both $S_n.\mathtt{bottom}$ and in $S_n.\mathtt{top}$. If $S_n.\mathtt{bottom}[i]$ contains more than one triple with height y we store (e', i, y) in $S_n.\mathtt{top}$ where e' is a pointer to the recursive structure in $S_n.\mathtt{bottom}[i]$.

Inserting a triple (e, x, y) in S_n: If S_n contains at most one triple with height y, we just update $S_n.\mathtt{min}[y]$ and $S_n.\mathtt{max}[y]$ and we are done. Else, we first check if (e, x, y) should go into $S_n.\mathtt{min}[y]$ (resp. $S_n.\mathtt{max}[y]$) and if this is the case we interchange (e, x, y) and the triple in $S_n.\mathtt{min}[y]$ (resp. $S_n.\mathtt{max}[y]$). Let T be the structure in $S_n.\mathtt{bottom}[x \operatorname{div} \sqrt{n}]$. We then insert $(e, x \bmod \sqrt{n}, y)$ in T. Let m be the number of triples in T with height y after this insertion. If $m = 1$, we insert $(e, x \operatorname{div} \sqrt{n}, y)$ in $S_n.\mathtt{top}$. If $m = 2$, there is a triple $(e', x \operatorname{div} \sqrt{n}, y)$ in $S_n.\mathtt{top}$, and we replace (see next paragraph) this triple with the triple $(e'', x \operatorname{div} \sqrt{n}, y)$ where e'' is a pointer to T. We observe that when we update $S_n.\mathtt{top}$ then T has at most two triples with height y and the update in T is performed by just accessing $T.\mathtt{min}$ and $T.\mathtt{max}$. It follows that we only perform a non-constant number of updates in at most one recursive structure.

Replacing a triple $(e, x, y) \in S_n$ with the triple (e', x, y): If $(e, x, y) = S_n.\mathtt{min}[y]$ (resp. $(e, x, y) = S_n.\mathtt{max}[y]$) we just update $S_n.\mathtt{min}[y]$ (resp. $S_n.\mathtt{max}[y]$) and we are done. Else, let T be the structure in $S_n.\mathtt{bottom}[x \operatorname{div} \sqrt{n}]$. First, we in T replace $(e, x \bmod \sqrt{n}, y)$ with $(e', x \bmod \sqrt{n}, y)$. Next, if T has exactly one triple with height y, we replace $(e, x \operatorname{div} \sqrt{n}, y)$ with $(e', x \operatorname{div} \sqrt{n}, y)$ in $S_n.\mathtt{top}$. As with insertions we note that we only need to perform a non-constant number of updates in at most one recursive structure.

Deleting a triple (e, x, y) from S_n: If S_n has at most two triples with height y, we just update $S_n.\mathtt{min}[y]$ and $S_n.\mathtt{max}[y]$ and we are done. Suppose therefore that S_n contains at least three triples with height y. We split into three cases: Case 1: In this case $(e, x, y) = S_n.\mathtt{min}[y]$. Let $(e', t, y) = S_n.\mathtt{top}.\mathtt{min}[y]$, $(e'', l, y) = S_n.\mathtt{bottom}[t].\mathtt{min}[y]$ and $i = l + t\sqrt{n}$. Then (e'', i, y) is the triple in $S_n \setminus (S_n.\mathtt{min} \cup S_n.\mathtt{max})$ with height y which has the minimal value of i. Instead of deleting (e, x, y) from S_n we delete (e'', i, y) and afterwards we set $S_n.\mathtt{min}[y]$ to (e'', i, y). Case 2: In this case $(e, x, y) = S_n.\mathtt{max}[y]$ and this case is handled in a symmetric way to case 1. Case 3: In this case (e, x, y) is not equal to $S_n.\mathtt{min}[y]$ or $S_n.\mathtt{max}[y]$. The case is handled in way similar to insertions: Let T be the structure in $S_n.\mathtt{bottom}[x \operatorname{div} \sqrt{n}]$. We then delete $(e, x \bmod \sqrt{n}, y)$ from T. Let m be the number of triples in T with height y after this deletion. Suppose $m = 1$ and let (e', i, y) be the triple with height y in T. Then there is a triple $(e'', x \operatorname{div} \sqrt{n}, y)$ in $S_n.\mathtt{top}$ where e'' is a pointer to T and we replace this triple with the triple $(e', x \operatorname{div} \sqrt{n}, y)$. If $m = 0$ we delete $(e, x \operatorname{div} \sqrt{n}, y)$ from $S_n.\mathtt{top}$. Again we observe that we only need to perform a non-constant number of updates in at most one recursive structure.

What remains is to describe how to answer a `reportsub` query. In order to do this, we in addition to $S_n.\mathtt{min}$ (resp. $S_n.\mathtt{max}$) maintain a structure $S_n.\mathtt{min}'$ (resp. $S_n.\mathtt{max}'$) with the type of the structure given to Theorem 3.1. We maintain $S_n.\mathtt{min}'$ (resp. $S_n.\mathtt{max}'$) such that $S_n.\mathtt{min}'$ (resp. $S_n.\mathtt{max}'$) contains an element e with position x, height y and a value $e.\mathrm{v} = e'$ iff $S_n.\mathtt{min}$ (resp. $S_n.\mathtt{max}$) contains the triple (e', x, y).

Answering a $\mathtt{reportsub}(S_n, i, j, q)$ query: If $i > j$ or $S_n = \emptyset$ we report nothing. Else, let M be the set of pairs $\{(e.\mathrm{v}, e.\mathrm{y}) \mid e \in \mathtt{report}(S_n.\mathtt{min}', i, j, q) \cup \mathtt{report}(S_n.\mathtt{max}', i, j, q)\}$. We then iterate the following as long as possible: Take a pair $(e, y) \in M$ where e points to a recursive structure T in our induction on n, and replace the pair with (e', y) and (e'', y) assuming $T.\mathtt{min}[y] = (e', x', y)$ and $T.\mathtt{max}[y] = (e'', x'', y)$. We observe that the time we spend on performing these replacements is proportional to $|M|$. After this, report the elements $\{e \mid \exists y.(e, y) \in M\}$. If $i = 0$ or $j = n - 1$ we stop. Else, if $i \operatorname{div} \sqrt{n} = j \operatorname{div} \sqrt{n}$ we perform a $\mathtt{reportsub}(S_n.\mathtt{bottom}[i \operatorname{div} \sqrt{n}], i \bmod \sqrt{n}, j \bmod \sqrt{n}, q)$ query. Finally if $i \operatorname{div} \sqrt{n} \neq j \operatorname{div} \sqrt{n}$ we split the query into the three queries $\mathtt{reportsub}(S_n.\mathtt{bottom}[i \operatorname{div} \sqrt{n}], i \bmod \sqrt{n}, \sqrt{n} - 1, q)$, $\mathtt{reportsub}(S_n.\mathtt{top}, i \operatorname{div} \sqrt{n} + 1, j \operatorname{div} \sqrt{n} - 1, q)$ and $\mathtt{reportsub}(S_n.\mathtt{bottom}[j \operatorname{div} \sqrt{n}], 0, j \bmod \sqrt{n}, q)$. We note that the first and the last of these three queries are answered by just looking at the `min`, `min'`, `min` and `max'` fields in the recursive structure. To analyze the space usage of S_n we need the following lemma:

Lemma 3.4. *Suppose p is defined by $p(0) = 2$ and $p(h) = 2 + (1 + 2^{2^{h-1}})p(h-1)$ for $h \geq 1$. Then $p(h) \leq (h+1)2^{2^h}$*

Proof. We show the lemma by induction on h. For $h = 0$ the lemma is trivially true. Suppose $h \geq 1$ and that the lemma is true for less h. Then we have $p(h) \leq 2 + (1 + 2^{2^{h-1}})h2^{2^{h-1}} = 2 + h2^{2^{h-1}} + h2^{2^h} \leq 2^{2^h} + h2^{2^h} = (h+1)2^{2^h}$. □

With p as in the lemma we observe that $p(\log\log n)$ is the number of `min`, `max`, `min'` and `max'` structures we need and the lemma then bounds the space usage of S_n to $O(Sn\log\log n)$ bits as desired.

The running time of the update operations in S_n is $O(U\log\log n)$. This follows from the observation that when an update operation operates in more than one recursive structure, it does a non-constant number of updates in at most one of these. A similar argument shows that the S_n has query time $O(Q\log\log n)$ concluding our proof of Theorem 3.1.

3.5 Lists, trees and predecessor

In Section 3.5.1 to 3.5.5 we review various well-known data structures for lists, trees and the predecessor problem. In Section 3.5.6 and Section 3.5.7 we develop some new results based on some of these well-known structures.

3.5.1 The predecessor problem

In this section we review various data structures for the predecessor problem. We have already cited one such structure, namely the VEB of Theorem 3.3. Combining the hashing scheme of [39] with Willard [107] we get the following structure.

Lemma 3.5. *If we allow randomization the space usage of Theorem 3.3 can be reduced to $O(n\log U)$ bits. The update time is then with probability $1 - n^{-c}$ for any desired constant $c > 0$.*

Building on Beame and Fich[15], Anderson and Thorup[11] have shown the following theorem.

Theorem 3.6. *There exists a data structure called a BFAT which can maintain a collection of n elements having keys in $[0 \ldots U]$. The structure uses space $O(n\log U)$ bits and supports updates as well as predecessor and successor queries in time $O(\min(\sqrt{\log n/\log\log n}, \log\log n\log\log U/\log\log\log U))$. U is called the universe size and we require $U = 2^{O(w)}$ where w is the word size in bits.*

We will take the time usage of Theorem 3.6 to be $O(\log^2\log U)$ (recall that we write $\log^i\log U$ for $(\log\log U)^i$). This is possible since we will always have $n \leq U$ and thus $\log\log n\log\log U/\log\log\log U \leq \log^2\log U$.

3.5.2 WBB trees

In this section we review WBB trees (Weight Balanced B-trees). Variants of these trees have been described by Willard [109], Dietz [35] and Arge and Vitter [14]. The description here is similar to the one in [14] where more details can be found.

Like an (a,b)-tree a WBB tree is a multibranching search tree where 1) elements are kept in the leaves 2) keys to guide the search are kept in internal nodes and 3) all leaves have the same depth (and height 0). We support updates to the tree in form of insertions (not deletions) of elements. We define the weight of an internal node as the number of elements in the leaves descendant to the node. The tree is parametrized over a *branching parameter* $d \geq 8$ and a *leaf parameter* $k \geq 1$. A leaf node must contain between k and $2k-1$ elements. When an insertion makes a leaf node contain $2k$ elements it is split into two leaves each with k elements. An internal node v at height h must have weight between kd^h and $4kd^h$ except if v is the root in which case we only require it to have weight at most $4kd^h$. When an insertion gives v weight exactly $3kd^h$ a key in v is *marked* such that there is at least kd^h elements descendant to v both to the left and to the right of the marked key. Such key exists because $(k3d^h - k4d^{h-1})/2 = kd^h(3-4/d)/2 \geq kd^h$. When v gets weight exactly $4kd^h$ it is split at the marked key. When a node w (leaf or internal) splits, a new child and a new key are inserted into the parent of w. If w is the root node a parent of w is first created as new root. The following lemma gives central properties for WBB trees:

Lemma 3.7. *Let T be a WBB tree with branching parameter d, leaf parameter k and n elements. Then:*

1. *A leaf contains between k and $2k-1$ elements.*
2. *Internal nodes have degree $O(d)$.*
3. *T has height at most $\min(\log n, O(\log n/\log d))$.*

Further, let $v \in T$ be an internal node at height $h \geq 1$. Then:

4. *At most one key in v is marked and when v split it is at the marked key.*
5. *There are at most $4kd^h$ and at least kd^{h-1} elements descendant to v.*
6. *Exactly kd^h insertions are performed in leaves descendant to v from the time where a key in v is marked to the time where v splits.*

As observed in [14], we can in WBB trees maintain secondary structures in each internal node v containing all elements in leaves descendant to v and at the same time have worst case time performance on updates: When a key in v is marked, we can start to build the secondary structures for the two nodes v will eventually split into. Each time an update is performed in a leaf descendant to v we move a constant number of elements to the two new secondary structures. Item 5 and 6 in Lemma 3.7 ensures that we have sufficient time to do this and item 4 of the lemma ensures that we know which elements to put into which of the two new secondary structures. We finally remark that we can assume there are pointers between an internal node and its children.

3.5.3 The list order problem

The following theorem is shown by Dietz and Sleator [37]. This theorem was mentioned in the introduction as the theorem allowing us to transform a structure for the list order variant into a structure for the comparison order variant.

Theorem 3.8. *There exists a linear sized data structure for a linked list supporting deletion and insertion of list elements in constant time such that we for given list elements e and e' in constant time can determine if $e < e'$.*

3.5.4 The online list labeling problem

We define the online list labeling problem as follows (other similar definitions are used in other papers, see eg. [116]). Let L be a linked list with at most n elements supporting insertions and deletions of elements. We must for each element $e \in L$ maintain an integer label e.label of size $O(n)$ such that for $e, e' \in L$:

$$e < e' \implies e.\text{label} < e'.\text{label} \tag{3.2}$$

An algorithm solving this problem is allowed to change the label of (relabel) elements when an element is inserted into or deleted from L. We require (3.2) to be maintained during this relabeling. If the algorithm relabels at most m elements on each insertion in or deletion from L we say that the algorithm has *relabeling cost* m. The following theorem is shown by Willard [110].

Theorem 3.9. *There exists an algorithm for the online list labeling problem with relabeling cost $O(\log^2 n)$, space usage $O(n \log n)$ bits, which uses time $O(\log^2 n)$ for insertions and deletions.*

3.5.5 List block balancing

The following theorem is proved by Dietz and Sleator [37, Theorem 5]. Recall that the Harmonic numbers $H_n = \sum_{i=1}^{n} 1/i$ satisfy $H_n = \Theta(\log n)$.

Theorem 3.10. *Suppose $x_1, \ldots, x_n$ are variables which are initially zero. Iterate the following two steps 1) Add a non-negative value to each x_i such that $\sum_{i=1}^{n} x_i$ increases by at most one. 2) Set a largest x_i to zero. Then for all i, $1 \leq i \leq n$ we always have $x_i \leq 1 + H_{n-1}$.*

The next lemma follows directly from this theorem.

Lemma 3.11. *Suppose $k \geq 1$ and that $\mathcal{C}$ is a collection of at most n sets and that initially $\mathcal{C} = \{\emptyset\}$. Iterate the following two steps. 1) Add at most k elements to the sets of $\mathcal{C}$ and 2) If $M \in \mathcal{C}$ is a largest set in $\mathcal{C}$ and $|M| \geq 5k(1 + H_{n-1})$ then split M into two sets each of size at least $2k(1 + H_{n-1})$ and let these replace M in $\mathcal{C}$. Then for all $M \in \mathcal{C}$ we always have $|M| \leq 6k(1 + H_{n-1})$.*

We now describe a general technique used in [37] which we will call *list block balancing* with *block size* $s \geq \log^3 n$. We use this technique as a part of the steps O4 and O9 in Section 3.3.2 where we group elements of an axis into numbered blocks and use this to convert a static axis to a dynamic axis.

Let L be an initially empty linked list in which at most $O(n)$ updates in the form of insertions and deletions can be performed. We want to maintain a grouping of the elements of L in blocks of size at most s respecting the order of the elements. Further, we want to maintain an integer label of size $O(n/s)$ for each block such that labels are always strictly increasing over the blocks.

Assume first that only insertions and not deletions can be performed in L. Define $k = s/(6(1+H_{n-1}))$ and note that $k = \Theta(s/\log n)$. When a block contains $\lceil 4k(1+H_{n-1}) \rceil$ elements we *mark* the middle position of the block. For every kth insertion in L we take (in constant time) a block with most elements and if it has $5k(1 + H_{n-1})$ elements or more we split it in two at the marked position. Lemma 3.11 gives that no block becomes larger than s. We use Theorem 3.9 to assign labels to the blocks. Each block split requires $O(\log^2 n)$ relabellings and we distribute this work over the k insertions until next split. It can be seen that we have to relabel a block for every $\Omega(k/\log^2 n) = \Omega(s/\log^3 n)$ insertion in L. We note that from the time a position in a block is marked to the block is split $\Omega(s)$ insertions are performed in the block. This allows us to move all existing elements of the block into the two new blocks and possibly insert them in some data structure maintained in each block. This includes maintaining pointers between a block and the elements is contains. We finally describe how to handle deletions of elements in L. We maintain a list L' which has L as a sublist. We never delete elements from L' but when we insert an element in L we also insert it in L' such that L remains a sublist of

L'. We then use the algorithm just described to maintain a block division of L' which induces the wanted block division on L (some blocks in L may become empty). The following lemma gives central properties for the list block balancing technique:

Lemma 3.12. *Let L be an initially empty linked list in which at most $O(n)$ updates in form of insertions and deletions can be performed. If we use list block balancing with block size $s \geq \log^3 n$ on L then*

1. *Each block has size at most s*
2. *Each block has a label of size $O(n/s)$. Further, $\Omega(s/\log^3 n)$ insertions are performed in L between each relabeling of a block.*
3. *At most one position in each block is marked and when a block is split it is at the marked position.*
4. *At least $\Omega(s)$ insertions are performed in a block from the time it is marked to it is split.*

3.5.6 A variant of the online list labeling problem

In this section we consider a variant of the online list labeling problem defined in Section 3.5.4 where the elements are relabeled in subconstant time per element. This is the basis for the steps O2 and O6 in Section 3.3.2 where we convert a static axis to a dynamic axis by relabeling coordinates. In the variant we consider, the user must to each element $e \in L$ assign a unique id of $O(\log n)$ bits when e is inserted into L, and this id cannot be changed as long as e remains in L. Further, we do not require the labels to be explicitly stored in the elements of L. Instead we require that we can calculate the label of a given element in constant time. A similar approach was used by Dietz and Sleator [37] in their proof of Theorem 3.8. Finally, when an element is inserted into or deleted from L, the user must be given an array of triples describing the relabeling taking place because of the insertion or deletion. A triple (i, l, l') in this array means that the element with id i and current label l is given label l'. For technical reasons we only allow a total of $O(n)$ updates in form of insertions and deletions to be performed in L.

When n is sufficiently small compared to N (that is, if we have a sufficiently large word size and can use a sufficiently large global lookup table), the following lemma gives a way to perform the relabeling caused by an insertion or deletion in constant time. In order to be able to do this, we pack the array describing the relabeling into a single word.

Lemma 3.13. *If $\log^4 n \leq \log^{1-\epsilon} N$ for a constant $0 < \epsilon < 1$ the described variant of the online list labeling problem can be solved in constant time per insertion in and deletion from L, a relabeling cost of $O(\log^3 n)$ and a space usage of $O(n \log n)$ bits.*

Proof. We use list block balancing with block size $\log^3 n$ on L. The label assigned to an element $e \in L$ in block b consists of the label of b multiplied with $\log^3 n$ plus the rank e has in b. This assignment of labels clearly fulfills (3.2) of Section 3.5.4 and because blocks have labels of size $O(n/\log^3 n)$ the labels assigned to elements in L have size $O(n)$. Further, we only need to relabel $O(1)$ blocks for each insertion in L and since blocks have size at most $\log^3 n$ we get a relabeling cost of $O(\log^3 n)$ as claimed.

For each block we maintain an array with the ids of the elements in the block in order. We store this array in a single word which is possible because it only takes up $O(\log^4 n)$ bits. These arrays can be used to find the label of a given element in constant time using a global lookup table. Also, using these arrays we can in constant time find the array of triples to return to the user because of an update. □

3.5.7 The colored predecessor problem

In this section we consider the colored predecessor problem which we define as follows. Let L be a linked list with $O(n)$ elements in which new elements can be inserted and from which existing elements can be deleted. Further, the user must assign a color to an element when it is inserted into L and also the user can change the color of an element in L. Given an element $e \in L$ and a color c, we must be able to find the predecessor of e in L with color c. Dietz and Raman [36, Lemma 4.2] have shown:

Theorem 3.14. *With two colors the colored predecessor problem can be solved in time $O(\log\log n)$ for updates and queries and space $O(n \log n)$ bits.*

Using an extension of the techniques in [36] we show the following theorem where we will only use the deterministic result. The theorem may be of independent interest which is why we also provide the randomized result.

Theorem 3.15. *The colored predecessor problem without restriction on the number of colors can be solved in space $O(n \log n)$ bits and in time $O(\log^2 \log n)$ for updates and queries. Alternatively, if we allow randomization, the time usage can be reduced to $O(\log\log n)$ and the update time is then with probability $1 - n^{-c}$ for any desired constant c.*

Proof. We first describe the randomized solution. The solution builds on ideas from the list block balancing technique. We first assume we can only update L by inserting a new element. Define $k = \log^2 n$. For each color c we divide the elements of L with color c into blocks with color c containing at most $6k(1+H_{n-1})$ elements respecting the order of the elements. We store the elements of each block in a balanced binary search tree. We mark the first element of each block b with color c except if b is the first block in L with color c. We insert all elements of L into the structure of Theorem 3.14 using marked and not-marked as the two colors. We use Theorem 3.9 to assign a label e.label to each marked element $e \in L$. We create a predecessor structure P_c of Lemma 3.5 for each color c and each marked element $e \in L$ with color c is inserted into P_c at position e.label.

Answering queries: Suppose we want to find the predecessor of $e \in L$ with color c. We first use time $O(\log\log n)$ to find the marked predecessor e' of e in L. Assuming first that e' exists, we use e'.label to perform a predecessor query in P_c which will identify a block b in time $O(\log\log n)$. If e' or b does not exists we take b to be the first block in L with color c. The block b will contain the element we are looking for (provided it exists) and this element can be then be found in time $O(\log\log n)$ using the binary search tree with the elements of b. We need to be able to compare two elements in L in constant time to do this, but this can be done using Theorem 3.8. We conclude that the total query time becomes $O(\log\log n)$.

Inserting an element e with color c in L: First, by performing a query as just described, we identify the strict predecessor e' of e with color c. We then insert e in the block containing e'. If e' does not exists we insert e in the first block with color c. So far, we have used $O(\log\log n)$ time. We observe that e' will never become the first element of a non-first block with color c so so far we do not need to remark elements in L because of the insertion. We now describe how to ensure that blocks do not become too big. Every time we have inserted k elements in L, we take (in constant time) a largest block and if it has at least $5k(1 + H_{n-1})$ elements we split it in $O(\log\log n)$ time into two blocks each with at least $2k(1 + H_{n-1})$ elements. Lemma 3.11 ensures that no block becomes too big. Splitting a block will mark a new element of L, and thus the structure of Theorem 3.9 requires us to change the label of $O(\log^2 n)$ marked elements. We distribute this work over the following $k = \log^2 n$ insertions in L performing $O(1)$ relabelings on each insertion. When relabeling a marked element with color c' we also update the structure

$P_{c'}$ to reflect the new label of the element. Lemma 3.5 then gives that we get an update time of $O(\log\log n)$ with probability $1 - n^{-c}$ for any desired constant $c > 0$.

We now support that the user can *dot* an element $e \in L$ in time $O(\log\log n)$. When answering queries we must then ignore the dotted elements. We modify the binary search tree in each block such that we can identify the predecessor of an element among the non-dotted elements. This can be done by writing in each node if the subtree rooted in that node contains a non-dotted element. Next, for each color c we modify the structure P_c such that we can perform a query among the blocks which contain at least one non-dotted element. Using these modified structures we can answer queries as before with one remark. When we have searched in a block b we may discover that the predecessor e' of the query is contained in the predecessor block b' of b which contain at least one non-dotted element. But we can find b' and then e' in time $O(\log\log n)$.

In order to make the data structure for L we do as in the the list block balancing technique. We maintain a list L' using the data structure just described such that the non-dotted elements of L' corresponds to the elements of L in the same order. We can then answer queries in L by asking in L'. When we insert an element in L we also insert a corresponding element in L'. When we delete an element from L we dot the corresponding element in L'. When we change the color of an element in L we dot the corresponding element in L' and insert a new corresponding element with the new color in L'. Using global rebuilding [87] we can ensure that no more than $O(n)$ updates are performed in L'.

Finally, for the deterministic result, we just use a BFAT of Theorem 3.6 as P_c instead of the predecessor structure of of Lemma 3.5. □

For readers familiar with dynamic fractional cascading [30, 75, 36] we note that we will use Theorem 3.15 in situations where dynamic fractional cascading has traditionally been used. The reason for this is that if we use dynamic fractional cascading the catalog graph will get high degree leading to bad performance in the known data structures for dynamic fractional cascading.

3.6 Structures supporting 3-sided queries

The main purpose of this section is to perform the steps O1 to O4 in Section 3.3.2 thus giving our final 3-sided range reporting structure. The data structure we develop will be based on the priority search tree of McCreight [72] and on extensions of the tree similar to the ones described by Willard [112] and Arge, Samoladas and Vitter [13]. This is combined with the use of buffers in the style of Brodal [22].

This section is organized as follows. In Section 3.6.1 we review priority search trees with the extensions of [112, 13]. In Section 3.6.2 we describe a dictionary for points and in Section 3.6.3 we describe how this dictionary can be transformed into a structure similar to the priority search tree of Section 3.6.1. In Section 3.6.4 we describe a way to implement the structure in Section 3.6.3 for few points giving the structure of step O1. Also, we give this structure a dynamic y-axis performing step O2. In Section 3.6.5 we look at 3-sided structures with many points first performing step O3 and finally step O4.

3.6.1 Priority search trees

The following theorem is essentially shown in [72]:

Theorem 3.16. *There exists a structure with type* $\mathrm{R}^3(\mathrm{d}:n, \mathrm{d}:n)$ *and performance* $(\log n, \log n, n\log n)$

The structure of the theorem is called a priority search tree. We will not show the theorem here. Instead, we will describe a way to make a structure R with type $\mathrm{R}^3(\mathrm{s}:n, \mathrm{s}:n)$ or $\mathrm{R}^3(\mathrm{s}:n, \mathrm{d}:n)$ and we will also refer to this structure as a priority search tree. The structure uses ideas from [72, 112, 13] where more details can be found. We defer the exact implementation details to later sections. We span a tree T with degree $d \geq 2$ and thus height $O(\log n/\log d)$ over $R.L_{\mathrm{x}}$. For each node $v \in T$ we maintain a set $v.P$ such that the following invariants are fulfilled.

j1. The sets $v.P$ for $v \in T$ are disjoint and their union is the points of R.
j2. If $v \in T$ is a non-root ancestor of $w \in T$ and $w.P$ is non-empty then $v.P$ is non-empty.
j3. If $v \in T$ is an ancestor of $w \in T$ and $p_w \in w.P$ and $p_v \in v.P$ then $p_v.\mathrm{y} \leq p_w.\mathrm{y}$.
j4. If $v \in T$ and $p \in v.P$ then v is an ancestor of the leaf with $p.\mathrm{x}$.

Observe that given the points of R and the values $|v.P|$ for all $v \in T$ there is at most one way to select $v.P$ for $v \in T$. We define for each internal node $v \in T$ the set $v.C$ as $\cup_{w \text{ child of } v} w.P$. Suppose we are given a query $(x_1, x_2, 0, y_2)$ where 0 is the first element in $R.L_{\mathrm{y}}$. Let $M \subseteq T$ be the set of internal nodes which are an ancestor to a leaf with x_1 or x_2 and note that $|M| = O(\log n/\log d)$. The query is answered as follows:

q1. For the root $v \in T$ report the points from $v.P$ inside the query region.
q2. Set $M' = M$.
q3. while M' is not empty:
    - q3.1. Remove a node v from M'.
    - q3.2. Report the points from $v.C$ which are inside the query region.
    - q3.3. Add to M' the internal nodes $u \in T \setminus M$ for which a point from $u.P$ was reported in item q3.2.

Lemma 3.17. *We have:*

1. *All points of R inside the query region are reported.*
2. *All the descendant leaves to the nodes added to M' in item q3.3 are between x_1 and x_2 on $R.L_{\mathrm{x}}$.*
3. *If item q1 and q3.2 can be handled in time $O(1+r)$ when r points are reported then the query time in R becomes $O(\log n/\log d)$.*

We now describe how to update R. We insert a point p in R by inserting it in $v.P$ for the root v of T. If this insertion breaks invariant j3 or if $v.P$ has becomes too big (according to some additional requirements we may have imposed), we take the point with the largest y-coordinate in $v.P$, remove it from $v.P$ and insert it recursively in $w.P$ for the child w of v such that invariant j4 is preserved. We delete a point by deleting it from the set $v.P$ it is located in. If this deletion breaks invariant j2 or if $v.P$ becomes too small (again, according to some additional requirements), we take the point p' from $v.C$ with minimal y-coordinate, add p' to $v.P$ and recursively delete p' from the set $w.P$ is is located in (w will be a child of v).

3.6.2 A dictionary for points

In this section we describe a dictionary for points. Besides supporting insertion of new points, it supports deletion and lookups of existing points given their x-coordinate. The dictionary uses ideas from [22]. In Section 3.6.2 we modify the dictionary to support 3-sided queries in the style of Section 3.6.1. We require updates to be performed such that two different points which are in the dictionary at the same time do not have the same x-coordinate. Further, it is not legal to perform a delete operation on a point which is not in the dictionary.

Our dictionary is parametrized over three parameters $d \geq 8$, g and r where $g \geq r$. We create a WBB tree T with branching parameter d and leaf parameter g. We keep pebbles in the nodes of T. Each pebble represents a point and there exists three kinds of pebbles: Light pebbles, insert pebbles and delete pebbles. Light pebbles are green while insert and delete pebbles are red. We think of light pebbles as representing contained points and of red pebbles as representing updates which are not fully completed. The internal nodes can only contain red pebbles and leaf nodes can only contain green pebbles. It is important to distinguish between pebbles and elements in the WBB tree—the former are used to describe which points the dictionary contains while the later are used to balance the tree and to maintain keys to guide the search in internal nodes. For $h \geq 1$ define:

$$
\begin{aligned}
m_h &= r((d-2)/(d-1) + \sum_{n=h}^{\infty} 1/d^h) \\
&= r(d^h - 2d^{h-1} + 1)/(d^h - d^{h-1})
\end{aligned}
$$

Because $h \geq 1$ we have $m_h \leq r$. The following procedure describes how to perform $r/3$ updates to the dictionary where pebbles describing the updates are added in item u3. In this section we do not add any pebbles in item u2 or item u4.1 but we allow it to be done to support our modifications in Section 3.6.3.

- u1. Set v to the root of T and h to the height of v.
- u2. Add at most r/d^h red pebbles to v.
- u3. Add at most $r/3$ red pebbles to v representing the updates.
- u4. while v is not a leaf:
  - u4.1. Add at most r/d^h red pebbles to v.
  - u4.2. Take m_h red pebbles from v (or as many as there is if there is fewer) and move them (one at a time) to the children of v as determined by their x-coordinate and the keys to guide the search in v.
  - u4.3. Set v to the child of v which has most red pebbles and set $h = h - 1$.

Note that in case there is more than m_h red pebbles in v in item u4.2 we do not specify which pebbles to take. When a pebble enters a node $v \in T$ we apply the following items exhaustively:

- a1. If v contains a delete pebble p and an insert pebble p' such that p deletes p' then remove both p and p'.
- a2. If v is a leaf and contains an insert pebble p then 1) convert p to a light pebble and 2) if there is no element with the same x-coordinate as p in the WBB tree insert such an element in v.
- a3. If v is a leaf and contains a delete pebble p then remove the light pebble in v that p deletes (which we know is there).

When a node v is split the red and green pebbles are distributed to the two new nodes as determined by their x-coordinate and the marked key in v (if v is an internal node) or by their x-coordinate and the elements in v (if v is a leaf). Note that we can perform a lookup in the dictionary by looking at the pebbles in the nodes between a leaf and the root of T. We have:

Lemma 3.18. *At item u1, a maximal set of siblings at height $h \geq 1$ contains at most $(d-1)m_h$ red pebbles.*

Proof. First observe that we can ignore splitting of nodes since they do not increase the number of pebbles in a maximal set of siblings. Further, there is no problem when a new root is created because it will contain no red pebbles. Consider the root node and assume it is not a leaf. Using

$d \geq 8$ it is easy to show $r/3+2r/d^h \leq m_h$ giving that all pebbles added to the root in item u2, u3 and u4.1 are removed again in item u4.2. Finally consider a maximal set S of non-root non-leaf siblings with height h. Assume by induction that the nodes in S contain at most $(d-1)m_h$ red pebbles. Assume further that they receive at most m_{h+1} pebbles from their common parent in item u4.2 and then receive additional at most r/d^h pebbles in item u4.1. The total number of red pebbles added to the nodes in S is then at most $m_{h+1} + r/d^h = m_h$. Let $s \leq dm_h$ be the total number of pebbles in the nodes of S after this and let v be the node in S with most red pebbles selected in item u4.3. There must be at least s/d pebbles in v which are then removed in item u4.2. After this the nodes in S contain at most $s(1-1/d) \leq dm_h(1-1/d) = (d-1)m_h$ red pebbles. □

The following lemma gives central properties for the maintained structure:

Lemma 3.19. *We have:*

1. *The execution of item u1 to u4 inserts at most r elements in T and this is done in a set of leaves which are siblings.*
2. *The space usage of T and its contained pebbles is linear in the number of insertions.*
3. *A leaf never contains more than $2g$ green pebbles.*
4. *A set of siblings never contains more than dr red pebbles.*

Proof. Item 1 and 2 are immediate. Item 3 follows from the fact that in each leaf there is an injective mapping from the green pebbles in the leaf to the elements of the WBB tree in the leaf. Finally, item 4 is true for leaf nodes since they contain no red pebbles. For internal nodes, item 4 is just a weak version of Lemma 3.18 because $m_h \leq r$. □

3.6.3 A structure supporting 3-sided queries

We now make a number of modifications to the structure of Section 3.6.2 in order to make it support 3-sided queries in the style of Section 3.6.1. Initially, we will ignore marking of keys and splitting of nodes in T. As in Section 3.6.2 two different points in the structure at the same time cannot have the same x-coordinate and in this section they cannot have the same y-coordinate either. We will still use insert and delete pebbles to represent updates which have not been fully completed. Also, the leaf nodes can still only contain light pebbles. However, we will now store both green and red pebbles in the internal nodes of T such that the green pebbles constitute a priority search tree of Section 3.6.1. We will keep the following invariant which corresponds to invariant j3.

i1. The y-coordinate of a green pebble in a node v may not be larger than the y-coordinate of any pebble below v in T.

For each internal node $v \in T$ we define a number $igreen(v)$ which we think of as the number of green pebbles we would like to be in v (there may be less but not more). If the root $v \in T$ is not a leaf we require

$$0 \leq igreen(v) \leq 2g \tag{3.3}$$

and for other internal nodes for $v \in T$ we require

$$g \leq igreen(v) \leq 2g \tag{3.4}$$

As described below, a green pebble may be deleted by a red pebble thus bringing the number of green pebbles in v down below $igreen(v)$. For this reason we introduce a new red pebble called a *poll* pebble which basically says that there should be a green pebble which is not there. Whenever a poll pebble is moved from a node v to one of its children w in step u4.2, we move a green pebble from w to v. This is made more precise below. We maintain the following invariant.

i2. Suppose v is an internal node and let v_o denote the number of poll and v_g the number of green pebbles in v. Then we require $igreen(v) = v_g + v_o$.

It may happen that we have no light pebble to move from w to v because all light pebbles in that part of the tree have either been deleted or moved up higher in three. For this reason we also introduce a new green pebble which we call a *heavy* pebble. A heavy pebble has no x-coordinate but has a y-coordinate larger than all non-heavy pebbles. According to invariant i1, a heavy pebble will never be above a light pebble in the tree; thus the names heavy and light pebble. When we consider splitting of nodes in T below, we relax invariant i1 such that this is not always true.

We now describe what happens when a red pebble enters a node $v \in T$ in item u4.2. We apply the following items exhaustively in addition to item a1 an a2 above (item a5 and a6 make item a3 superficial):

a4. Let p be an insert pebble in v and let p' be a green pebble in v with largest y-coordinate among the green pebbles in v. If p has smaller y-coordinate than p' then 1) convert p to a light pebble 2) if p' is a light pebble convert it to an insert pebble 3) if p' is a heavy pebble remove it.

a5. If v contains a delete pebble p and a light pebble p' such that p deletes p' then remove p and p' and add a poll pebble to v.

a6. If v is a leaf remove any poll pebble from v.

The following item describes how a poll pebble p in a node v is moved to a child of v in item u4.2. This does not follow from the description so far because poll pebbles have no x-coordinate.

m1. Let p' be the green pebble with the smallest y-coordinate among the green pebbles stored in the children of v if it exists. If p' exists, interchange p and p'. If p' does not exists, add a heavy pebble to v and remove p.

We note that the items a4, a5, a6 and m1 never increase the number of red pebbles in a node. Thus, the described pebble game is indeed just a variant of the game of Section 3.6.2 and all the lemmas of Section 3.6.2 still hold. Also it can be checked that invariant i1 and i2 are never broken.

The described structure can answer 3-sided queries as in Section 3.6.1 in the following sense. In item q1 and q3.2 we report the points represented by the insert and light pebbles in the node. However we do not report a point which is deleted by a delete pebble above it. We can avoid this as follows. First, we implement M' as a stack so that T is traversed in depth-first order. Next, while we traverse T we maintain the set of delete pebbles contained in the nodes which are ancestors to the node we are currently visiting. We then only report a point if it is not in this set. In order for the algorithm to work, we must maintain some variant of invariant j2. Essentially this is done by ensuring that all internal non-root nodes $v \in T$ contain at least one green pebble which is not deleted by a delete pebble above it. We handle this issue in detail in Section 3.6.4.

We now describe how to handle marking of keys and splitting of nodes in T and how to select $igreen(v)$ for internal nodes $v \in T$. If v does not have a marked key we select $igreen(v) = 0$ if v is the root and $igreen(v) = g$ otherwise. Assume for the rest of this paragraph v is an internal node with a marked key. Each time we pass though v in item u2 or item u4.1 we increase the value of $igreen(v)$ with $\min(2g - igreen(v), r/d^h)$ and we add the same amount of poll pebbles to v to preserve invariant i2. Since each pass through v adds at most r elements to T below v (item 1 of Lemma 3.19) it follows from property 6 of Lemma 3.7 that when v is split into v'

and v'', we have $igreen(v) = 2g$. We then set $igreen(v') = igreen(v'') = g$. In the following we modify our structure such that invariant i2 and a relaxed version of invariant i1 is preserved in v' and v''. We will say a child of v is a child of the left (resp. right) side of v if the the keys of the child are all smaller (resp. larger) than the marked key in v. Similarly, we will say a pebble is in the left (resp. right) side of v if it has an x-coordinate and if this coordinate is smaller (resp. larger) than the marked key in v. We will put each heavy pebble in the left or right side of v and this determines where the pebble should go when v splits. We will replace invariant i1 with invariant i1' and i1". These variants are weaker than invariant i1. Also, we will introduce a new invariant i3.

- i1'. The y-coordinates of the g green pebbles (or all of them if there is less than g green pebbles) in a non-root node w with smallest y-coordinates may not be larger than the y-coordinate of any pebble below w in T.
- i1". If an internal node v has a marked key, the y-coordinate of a green pebble in the left (resp. right) side of v must not be larger than the y-coordinate of any pebble below the left (resp. right) side of v.
- i3. If an internal node v has a marked key, there can be at most g green pebbles in each side of v.

Invariant i1' is sufficient to ensure we can use the same query algorithm as outlined above. Further, it can be checked, that invariant i1" and i3 allows us to preserve invariant i1' when we split a node. Inspired by invariant i3 we will say the left (resp. right) side of v is *full* if v has a marked key and the left (resp. right) side of v has g green pebbles.

We now modify the way we maintain our structure in order to preserve these modified invariants. First we modify the way we select p' in item m1 as follows. If a side of v is full, we select p' to be the green pebble with smallest y-coordinate among the green pebbles stored in the children of the non-full side of v. Further, if we add a heavy pebble in item m1 we do it to a non-full side. Next we modify the way we select p' in item a4 as follows. If p is on a full side of v we let p' be a green pebble in v with largest y-coordinate among the green pebbles in v on the same side of v as p. With these modifications it can be checked that the modified invariants are preserved.

3.6.4 Few points

The proof of the following lemma gives a way to implement the structure from Section 3.6.3 if there is few points. This is the structure from step O1 in Section 3.3.2.

Lemma 3.20. *Suppose $d \geq 8$ and that $1 \leq t \leq (\log^{1-\epsilon} N)/(d^2 \log^3 u)$ for a constant $0 < \epsilon < 1$. Then there exists a structure with type $\mathrm{R}^3(\mathrm{s}\!:\!u, \mathrm{s}\!:\!u)$ and performance $(1, 1+\log u/\log d, u \log u, t)$.*

Proof. As a basis for the structure we use the tree T from Section 3.6.3 with the same value of d as in the lemma here.

We store as a part of each insert and light pebble the point it represents including x- and y-coordinates using $O(\log u)$ bits for each pebble. Next, we store as a part of each delete pebble the coordinates of the point it deletes, again using $O(\log u)$ bits. Poll and heavy pebbles do not have any associated information so any pebble can be stored using $O(\log u)$ bits. Lemma 3.19 item 2 now gives that the space usage of T is linear in the number of insertions performed. Using global rebuilding of [87] we can get space usage $O(u \log u)$ bits and further Lemma 3.7 property 3 gives that we can keep the height of T on at most $\min(2 \log u, O(\log u/\log d))$ (we use $2 \log u$ instead of $\log u$ because we use global rebuilding on T).

In order to get constant update time, we modify the structure as follows. We maintain a set U of insert and delete pebbles. The user can then perform t updates by adding t pebbles to

U (using a global lookup table to convert them to the appropriate format). Each time the user does this we execute a constant number of steps in item u4. In item u3 we take all pebbles in U and insert them in the root of T. Since T has height $O(\log u)$ there is not too many pebbles in U if:

$$t \log u \leq r \tag{3.5}$$

We will select g and r such that the following properties are fulfilled.

k1. All pebbles in a maximal set of siblings of T can be stored in $O(\log^{1-\epsilon} N)$ bits in a single word.

k2. The maximal size of U plus the maximal number of red pebbles on a root path is larger than the minimal number of green pebbles in an internal node fulfilling invariant i1'.

Property k1 together with a global lookup table of size $2^{\log^{1-\epsilon} N}$ allows us to perform item u4.2, u4.3 and the corresponding updates in the involved nodes in constant time. Updating T will take constant time since we have just argued it consists of a constant number of steps each taking constant time. Note that in order for this to work, it is central that we explicitly store the x- and y-coordinates as a part of the insert, delete and light pebbles. Property k2 together with invariant i1' implies that any internal node v contains a green pebble p which is not deleted by a delete pebble above it and such that all green pebbles in nodes descendant to v have a y-coordinate not smaller than p (p may be a heavy pebble). This essentially ensures that invariant j2 is fulfilled. More precisely it ensures that the query algorithm outlined in Section 3.6.3 reports all points inside the query region. Also, using a global lookup table with size $2^{\log^{1-\epsilon} N}$ we can execute item q1 and q3.2 in time $O(1 + r)$ when r points are reported. Lemma 3.17 item 3 then gives that we get query time $O(\log u / \log d)$ in T.

We now discuss how to select g in order to fulfill property k1. Because $g \geq r$ Lemma 3.19 item 3 and 4 together with (3.3) and (3.4) implies that a set of siblings never contains more than $3dg$ pebbles. Since each pebble uses $O(\log u)$ bits property k1 is fulfilled if $dg \log u \leq \log^{1-\epsilon} N$. So we select:

$$g = (\log^{1-\epsilon} N)/(d \log u) \tag{3.6}$$

We then describe how to select r in order to fulfill property k2. Using Lemma 3.19 item 4 the number of green pebbles in an internal node is at least $g - dr$. If we also use the fact that T has height at most $3 \log u$ (it also has height at most $2 \log u$, but we use $3 \log u$ to support the modifications in the proof of Lemma 3.25) the number of red pebbles on a root path plus the number of pebbles in U is at most $r + 3dr \log u$. So we require $r + 3dr \log u < g - dr$ which is fulfilled if $4dr \log u \leq g$. So based on (3.6) we select:

$$r = (\log^{1-\epsilon} N)/(4d^2 \log^2 u) \tag{3.7}$$

Equation (3.5) together with (3.7) now gives our requirement for t. □

The reader may observe that a tree with fixed shape instead of the more complicated WBB tree appears to be sufficient in order to prove Lemma 3.20. The reason we use a WBB tree is the proof of Lemma 3.30.

In order to perform step O2 in section 3.3.2 we need to give the structure of Lemma 3.20 a dynamic y-axis. Since the same construction is needed in step O7 we prove the following general Lemma which can do this transformation.

Lemma 3.21. *Suppose* $\log^4 u \leq \log^{1-\epsilon} N$ *for a constant* $0 < \epsilon < 1$ *and there exists a structure with type* $\mathrm{R}^\tau(\mathrm{s}\!:\!u, \mathrm{s}\!:\!u)$ *(resp.* $\mathrm{R}^\tau(\mathrm{s}'\!:\!u, \mathrm{s}\!:\!u)$*) and performance* $(U, Q, S, \log^3 u)$*. Then there exists a structure with type* $\mathrm{R}^\tau(\mathrm{s}\!:\!u, \mathrm{d}\!:\!u)$ *(resp.* $\mathrm{R}^\tau(\mathrm{s}'\!:\!u, \mathrm{d}\!:\!u)$*) and performance* (U, Q, S).

Proof. We describe a solution in which at most $O(u)$ updates can be performed. This restriction can be removed by the use of global rebuilding [87]. Let R be the structure with type $\mathrm{R}^\tau(\mathrm{s}:u, \mathrm{d}:u)$ (resp. $\mathrm{R}^\tau(\mathrm{s}':u, \mathrm{d}:u)$) we want to design from the structure R' with type $\mathrm{R}^\tau(\mathrm{s}:u, \mathrm{s}:u)$ (resp. $\mathrm{R}^\tau(\mathrm{s}':u, \mathrm{s}:u)$). We assume each element $e \in R$ has an unique id $e.\mathrm{id}$ of size $O(u)$ which we must report when reporting an element from a query in R. We maintain a labeling of the elements of $R.L_\mathrm{y}$ using Lemma 3.13 getting the requirement $\log^4 u \le \log^{1-\epsilon} N$. If $e \in R$ then as id for $e.\mathrm{y} \in R.L_\mathrm{y}$ we use the pair $(e.\mathrm{id}, e.\mathrm{x})$. Lemma 3.13 then assigns a label $y.\mathrm{label}$ of size at most $O(u)$ to each $y \in R.L_\mathrm{y}$. For each element $y \in R.L_\mathrm{y}$ with id (i, x) we keep an element in R' with y-coordinate $y.\mathrm{label}$, x-coordinate x and id i. This can be done by using a global lookup table to convert the array of triples provided by Lemma 3.13 into updates to give to R'. A query $(x_1, x_2, y_1, y_2) \in R$ is then answered by performing the query $(x_1, x_2, y_1.\mathrm{label}, y_2.\mathrm{label})$ in R'. □

We remark that we cannot use the same technique to give the structure two dynamic axes. We finally get the structure of step O2 in Section 3.3.2:

Lemma 3.22. *Suppose $d \ge 8$ and that $\log^6 u \le (\log^{1-\epsilon} N)/d^2$ for a constant $0 < \epsilon < 1$. Then there exists a structure with type $\mathrm{R}^3(\mathrm{s}:u, \mathrm{d}:u)$ and performance $(1, 1 + \log u/\log d, u \log u)$.*

Proof. Selecting $t = \log^3 u$ in Lemma 3.20 gives a structure with type $\mathrm{R}^3(\mathrm{s}:u, \mathrm{s}:u)$ and performance $(1, \log u/\log d, u \log u, \log^3 u)$. Further we get the requirement that $\log^3 u \le (\log^{1-\epsilon} N)/(d^2 \log^3 u)$ or equivalently $\log^6 u \le (\log^{1-\epsilon} N)/d^2$. Finally, inserting the obtained structure into Lemma 3.21 gives the desired result since the requirement $\log^4 u \le \log^{1-\epsilon} N$ is fulfilled by our other requirement for u. □

3.6.5 Many points

The following lemma gives our structure from step O4.

Lemma 3.23. *For any sufficiently small constant $\epsilon > 0$ there exists a structure with type $\mathrm{R}^3(\mathrm{s}:n, \mathrm{d}:n)$ and performance $(1 + \log n/\log^{1/6-\epsilon} N, 1 + \log n/\log\log N, n \log n)$.*

Proof. Define $u = \min(n, 2^{\log^{1/6-\epsilon/2} N})$. We let R' be the structure we get from Lemma 3.22 if we set d in the lemma to $\log^\epsilon N$. Then the maximal u we can use in the lemma is equal to the u we have just defined and R' get type $\mathrm{R}^3(\mathrm{s}:u, \mathrm{d}:u)$ and performance $(1, 1+\log u/\log\log N, u \log u)$.

We assume for the rest of the proof that $n \ge u$ and thus $u = 2^{\log^{1/6-\epsilon/2} N}$ since otherwise R' is the structure we need to prove the theorem. We let the structure R with type $\mathrm{R}^3(\mathrm{s}:n, \mathrm{d}:n)$ we want to design be a priority search tree of Section 3.6.1. We give T degree u and require $|v.P| \le 1$ for all $v \in T$. In each internal node $v \in T$ we keep a structure $v.R$ with the same type as R'. Let $v \in T$ be an internal node and assume $p \in w.P$ for a child w of v. We then store a point $p' \in v.R$ corresponding to p with $p'.\mathrm{y} = p.\mathrm{y}$ and with $p'.\mathrm{x}$ set to the number of siblings w has to the left in T. Also, we link p and p' together with pointers. For each element $e \in R.L_\mathrm{y}$ there is a unique node $v \in T$ and a unique element $e' \in v.R.L_\mathrm{y}$ such that e and e' represent the same y-coordinate. We make a pointer from e to e' and color e with the color v. We maintain the colored predecessor structure of Theorem 3.15 on $R.L_\mathrm{y}$ using the assigned colors.

We answer queries as in Section 3.6.1 with the following remarks. Assume first a node $v \in M \cap M'$ is picked in item q3.1. In item q3.2 we then do as follows. For the (at most two) children $w \in M \cap M'$ of v we report the point in $w.P$ if it is in the query region. The remaining points to report can be found by a query in $v.R$. In order to find the y-coordinate to use in this query we make a query for the color of v in the structure of Theorem 3.15 maintained on $R.L_\mathrm{y}$. We conclude that we must use a total time of $O(\log u/\log\log N + \log^2\log n + r)$ to report r elements from v. This is also $O(\log u/\log\log N + r)$ since $u \leq n \leq N$. Assume next a node $v \in M' \setminus M$ is picked in item q3.1. Then Lemma 3.17 item 2 gives that we should report all the points in $v.C$ with sufficiently low y-coordinate. These points can be found in constant time per point by walking through $v.R.L_{\mathrm{y}}$ from the beginning stopping when the y-coordinate becomes too big. Since $|M| = O(\log n/\log u)$ we conclude that the query time in R becomes $O((\log u/\log\log N)\log n/\log u) = O(\log n/\log\log N)$.

Updates can also be performed as in Section 3.6.1 with remarks similar to the ones described for queries. For a total of $O(\log n/\log u)$ nodes $v \in T$ we need to update $v.R$ and to recolor an element of $R.L_{\mathrm{y}}$. Updating $v.R$ takes constant time while updating $R.L_{\mathrm{y}}$ takes time $O(\log^2 \log n)$. The total update time thus becomes $O(\log^2 \log n \log n/\log u) = O(\log^2 \log n \log n/\log^{1/6-\epsilon/2} N) = O(\log n/\log^{1/6-\epsilon} N)$ as claimed. $\square$

Finally the following theorem gives the structure of step O4 from Section 3.3.2. For $n = N$ this gives the second result from the introduction.

Theorem 3.24. *For any sufficiently small constant $\epsilon > 0$ there exists a structure with type* $\mathrm{R}^3(\mathrm{d}\!:\!n, \mathrm{d}\!:\!n)$ *and performance* $(\log\log n + \log n/\log^{1/6-\epsilon} N, \log\log n + \log n/\log\log N, n\log n)$.

Proof. We describe a solution in which at most $O(n)$ updates can be performed. This restriction can be removed by the use of global rebuilding [87]. Let R be the structure with type $\mathrm{R}^3(\mathrm{d}\!:\!n, \mathrm{d}\!:\!n)$ we want to design. We maintain a structure T of Lemma 3.23 with type $\mathrm{R}^3(\mathrm{s}\!:\!n/\log^3 n, \mathrm{d}\!:\!n/\log^3 n)$ and performance $(1 + \log n/\log^{1/6-\epsilon} N, 1 + \log n/\log\log N, n/\log^2 n)$. We use list block balancing with block size $\log^3 n$ on $R.L_{\mathrm{x}}$. We keep the points of a block b in a structure $b.R$ of Theorem 3.16. Let b be a block with assigned label x and let y be the first element of $b.R.L_{\mathrm{y}}$. We then insert b in T with x-coordinate x and y-coordinate y. This is possible because labels of blocks have size $O(n/\log^3 n)$. If an element $e \in R.L_{\mathrm{y}}$ represents the same y-coordinate as an element $e' \in T.L_{\mathrm{y}}$ (resp. $e'' \in b.R.L_{\mathrm{y}}$) we link e and e' (resp. e'') together with pointers.

We maintain the structure of Theorem 3.14 on $R.L_{\mathrm{y}}$ giving elements which have a pointer to an element in $T.L_{\mathrm{y}}$ one color and other elements another color. This allows us to identify the position in $T.L_{\mathrm{y}}$ of any element in $R.L_{\mathrm{y}}$ in time $O(\log\log n)$. For each block b we keep the elements of $b.R.L_{\mathrm{y}}$ in a balanced binary search tree and we maintain a structure of Theorem 3.8 on $R.L_{\mathrm{y}}$. This allows us to identify the position in $b.R.L_{\mathrm{y}}$ of any element in $R.L_{\mathrm{y}}$ also in time $O(\log\log n)$.

We need to relabel $O(1)$ blocks for each update in R so we need to perform $O(1)$ updates in T for each update in R. Since updating $b.R$ for a block b and identifying elements in and updating $b.R.L_{\mathrm{y}}$ and $T.L_{\mathrm{y}}$ takes time $O(\log\log n)$ the total update time becomes $O(\log\log n + \log n/\log^{1/6-\epsilon} N)$.

We now describe how to answer a query $(x_1, x_2, 0, y_2)$ where 0 is the minimal element of $R.L_{\mathrm{y}}$. Let b_1 be the block with x_1 and b_2 be the block with x_2. We then answer the query by first making a local query in $b_1.R$ and then a local query in $b_2.R$ using time $O(\log\log n)$ (if $b_1 = b_2$ we only need to perform one local query in total). The remaining points can be found by performing a query in T. For each reported block b we identify the points from b to report by walking through $b.R.L_{\mathrm{y}}$ from the beginning stopping when the y-coordinate becomes large than y_2 (as in the proof of Lemma 3.23). Identifying the elements to query from in $b_1.R.L_{\mathrm{y}}$, $b_2.R.L_{\mathrm{y}}$ and $T.L_{\mathrm{y}}$ requires time $O(\log\log n)$ so the total query time becomes $O(\log\log n + \log n/\log\log N)$. $\square$

3.7 Structures supporting 4-sided queries

The purpose of this section is to perform step O5 to O10 in Section 3.3.2 thus giving our final 4-sided range reporting structure for the 2-dimensional case. The section is organized as follows. In Section 3.7.1 we describe a simple way to make a structure supporting 4-sided queries from one supporting 3-sided queries. In Section 3.7.2 we use this to perform step O5, O6 and O7. We then apply the construction from Section 3.4 to perform step O8 and then we perform step O9. Finally, in Section 3.7.3 we perform step O10.

3.7.1 From 3-sided to 4-sided

In this section we describe a simple way to make a structure supporting 4-sided queries from one supporting 3-sided queries. Similar ideas have been used by Chazelle [25] and Overmars [88]. Suppose for any m, $m \le n$ we have a structure with type $\mathrm{R}^3(\mathrm{s}'\!:\!n, \mathrm{s}\!:\!m)$, and performance (U_m, Q_m, S_m). We will now describe how to make a structure R with type $\mathrm{R}^4(\mathrm{s}'\!:\!n, \mathrm{s}\!:\!n)$ and performance $(U_n \log n, Q_n, S_n \log n)$ provided U_m and Q_m are non-decreasing functions of m and provided S_m is a subadditive function of m. We span a complete binary tree T over $R.L_{\mathrm{y}}$. Each node $v \in T$ spans a subinterval I of $R.L_{\mathrm{y}}$ where $|I| \le n$. We store in v two secondary structures $v.R_1$ and $v.R_2$ both with type $\mathrm{R}^3(\mathrm{s}'\!:\!n, \mathrm{s}\!:\!|I|)$ and both containing all points of R with a y-coordinate in I. The structure $v.R_1$ (resp. $v.R_2$) should support queries of the form (x_1, x_2, y_1, y_2) where y_1 (resp. y_2) is the left most (resp. right most) element in I. Inserting (resp. deleting) a point p in (resp. from) R requires inserting (resp. deleting) p in (resp. from) two secondary structures in each of the $O(\log n)$ ancestors of the leaf of T with $p.\mathrm{y}$ using time $O(U \log n)$. Suppose we are given a query (x_1, x_2, y_1, y_2). Let $v \in T$ be the nearest common ancestor of y_1 and y_2 and let v_l be the left and v_r the right child of v. The query can then be answered by performing a single query in $v_l.R_2$ and a single query in $v_r.R_1$ giving a query time $O(Q)$.

3.7.2 At least one short axis

The following lemma gives the structure from step O5 of Section 3.3.2.

Lemma 3.25. *Suppose* $1 \le t \le (\log^{1-\epsilon} N)/\log^4 u$ *for a constant* $0 < \epsilon < 1$*. Then there exists a structure with type* $\mathrm{R}^4(\mathrm{s}'\!:\!u, \mathrm{s}\!:\!u)$ *and performance* $(1, 1 + \log u, u \log^2 u, t)$.

Proof. We will first how to construct a structure with type $\mathrm{R}^4(\mathrm{s} : u, \mathrm{s} : u)$ and the claimed performance. We will do this by combining the proof of Lemma 3.20 with the construction in Section 3.7.1. We keep all points in the dictionary of Section 3.6.2. To avoid confusion we call the tree T' instead of T. Further, we make a number of modifications and simplifications described in the following. First, we use the y- instead of the x-coordinates of points. Second, we let T' be a complete binary tree with fixed shape so nodes never split and keys are never marked. This implies that each leaf contains at most one green pebble. Third, we define $m_h = r/3$ for all h. Fourth, in our procedure for performing $r/3$ updates in T' we remove item u2 and u4.1. Lemma 3.18 can be shown to still hold with these modifications (a simpler proof carries through).

Like in Section 3.7.1 we have two secondary structures in each internal node of T'. Each time a pebble is moved from a node $v \in T'$ in item u4.2, a copy of the pebble is also given to the two secondary structures in v. For the secondary structures, we use structures similar to the one of Lemma 3.20 with $d = 8$. We select the same value for r in T' as in Lemma 3.20, namely the one given by (3.7) with $d = 8$. We can now answer queries as in done in Section 3.7.1. We note that when reporting points from the secondary structures in T' we must also take the delete pebbles of the relevant nodes of T' into account. But this is not a problem because the height of T' plus the height of any secondary structure is at most $3\log u$.

Like in the proof of Lemma 3.20 we will only add t pebbles at a time to the root of T' (where t will be determined in a moment). We note that during the time $r/3$ pebbles are inserted into the root of T' we must go through item u1 to u4 like in the proof of Lemma 3.20. Further, each time we execute item u4.2 in T' we must also execute the loop in item u1 to u4 in a constant number of secondary structures. Since the height of T' as well as the height of any secondary structure is at most $O(\log u)$ the following condition ensures that we do not add too many pebbles to the root of T' before we empty it again:

$$t\log^2 u \leq r$$

Together with (3.7) this gives our requirement for t.

At most $O(\log u)$ pebbles are created for each insertion in T' so using global rebuilding [87] we can get a space usage of $O(u\log^2 u)$ bits.

We give the structure type $\mathrm{R}^4(\mathrm{s}'\!:u, \mathrm{s}\!:u)$ using a standard trick: We insert each point e into the structure just described as a point e' with $e'.\mathrm{x} = c\cdot u\cdot e.\mathrm{x} + e.\mathrm{y}$ and $e'.\mathrm{y} = e.\mathrm{y}$ for a constant c. Clearly, each point inserted will have unique x and y-coordinates. However, it doubles the number of bits in each x-coordinate. It can be checked in our construction that this is not a problem. □

As a corollary we get the following lemma which gives the structure from step O6 of Section 3.3.2.

Lemma 3.26. *Suppose* $\log^7 u \leq \log^{1-\epsilon} N$ *for a constant* $0 < \epsilon < 1$. *Then there exists a structure with type* $\mathrm{R}^4(\mathrm{d}\!:u, \mathrm{s}'\!:u)$ *and performance* $(1, \log u, u\log^2 u)$.

Proof. Selecting $t = \log^3 u$ in Lemma 3.25 gives a structure with type $\mathrm{R}^4(\mathrm{s}'\!:u, \mathrm{s}\!:u)$ and performance $(1, \log u, u\log^2 u, \log^3 u)$. Further we get the restriction on u that $\log^3 u \leq (\log^{1-\epsilon} N)/\log^4 u$ or equivalently $\log^7 u \leq \log^{1-\epsilon} N$. Finally, inserting the obtained structure into Lemma 3.21 and interchanging the two axes gives the desired result since the requirement $\log^4 u \leq \log^{1-\epsilon} N$ is fulfilled by our other requirement for u. □

As a corollary, we get the structure from step O7 of Section 3.3.2.

Lemma 3.27. *Suppose* $\log^7 u \leq \log^{1-\epsilon} N$ *for a constant* $0 < \epsilon < 1$. *Then, for any* $n \leq N$ *there exists a structure with type* $\mathrm{R}^4(\mathrm{s}'\!:n, \mathrm{s}\!:u)$ *and performance* $(\log^2\log n, \log u + \log^2\log n, u(\log^2 u + \log n))$.

Proof. Let R' be the structure of Lemma 3.26 restricted to have type $\mathrm{R}^4(\mathrm{d}: u, \mathrm{s}: u)$. Let R be the structure with type $\mathrm{R}^4(\mathrm{s}'\!:n, \mathrm{s}\!:u)$ we want to design. We insert each element $e \in R$ in a BFAT of Theorem 3.6 with universe size $O((n+1)u)$ at position $u\cdot e.\mathrm{x} + e.\mathrm{y}$. We link all elements in R together according to this position. This list can be used as $R'.L_{\mathrm{x}}$. Further, at most u elements will be stored in the BFAT and this concludes the proof of the lemma. □

Plugging the structure of this lemma into Lemma 3.2 we get the structure of step O8 of Section 3.3.2.

Lemma 3.28. *Suppose* $\log^7 u \leq \log^{1-\epsilon} N$ *for a constant* $0 < \epsilon < 1$ *Then, for any* $n \leq N$ *there exists a structure with type* $\mathrm{R}^4(\mathrm{s}': n, \mathrm{s}': u)$ *and performance* $(\log^3\log n, \log u\log\log n + \log^3\log n, nu\log n\log\log n(\log^2 u + \log n))$.

The following lemma gives the structure from step O9 of Section 3.3.2.

Lemma 3.29. *For any $n \leq N$, and for any u where $\log^3 n \leq u$ and $\log^7 u \leq \log^{1-\epsilon} N$ for a constant $0 < \epsilon < 1$ there exists a structure with type* $\mathrm{R}^4(\mathrm{d}: n, \mathrm{s}': u/\log^3 n)$ *and performance* $(\log^3 \log n, (\log u + \log^2 \log n) \log\log n, n \log^2 u + n \log n)$.

Proof. We describe a solution in which at most $O(n)$ updates can be performed. This restriction can be removed by the use of global rebuilding [87]. Let T be the structure of Lemma 3.28 where we select u in the lemma as $u/\log^3 n$ and n in the lemma as n/u. The structure T then gets type $\mathrm{R}^4(\mathrm{s}' : n/u, \mathrm{s}' : u/\log^3 n)$ and performance $(\log^3 \log n, \log u \log\log n + \log^3 \log n, n \log n)$ (the space usage is actually smaller).

Let R be the structure with type $\mathrm{R}^4(\mathrm{d} : n, \mathrm{s}' : u/\log^3 n)$ we want to design. We use list block balancing with block size u on $R.L_\mathrm{x}$. If a block b with assigned label x contains one or more points with y-coordinate y we insert b in T at x-coordinate x and y-coordinate y. This is possible because labels have size $O(n/u)$. Further, we only need to relabel a block for every $O(u/\log^3 n)$ update of R. Since this is also the length of $R.L_\mathrm{y}$ we only need to perform $O(1)$ updates in T for each update in R. We maintain in each block b a structure $b.R$ of Lemma 3.26 with type $\mathrm{R}^4(\mathrm{d} : u, \mathrm{s}' : u)$ and performance $(1, \log u, u \log^2 u)$. All points in b are stored in $b.R$ and the total space used by these structures is $O(n \log^2 u + n \log n)$ bits

Suppose we are given a query (x_1, x_2, y_1, y_2). Let b_1 be the block with x_1 and b_2 be the block with x_2. We answer the query by first performing a local query in $b_1.R$ and then a local query in $b_2.R$ (if $b_1 = b_2$ we only need to perform one local query in total). The remaining points can be found by performing a query in T and then report relevant points from the reported blocks. This can be done in constant time per point if we for each block b and each y-coordinate $y \in R.L_\mathrm{y}$ maintain a linked list with the points in b with y-coordinate y.

The space usage of the structure is dominated by the space used in the $b.R$ structures for the different blocks b. The update time is dominated by the time used in T and this concludes the proof of the lemma. □

The following lemma gives the structure of Lemma 3.29 a somewhat dynamic y-axis.

Lemma 3.30. *The structure of Lemma 3.29 can be given a dynamic y-axis in the following sense. The y-axis is a linked list where each element has a unique id of size $O(u)$. The id of an element is fixed when the element is inserted in the y-axis and we do not allow elements to be deleted from the y-axis. We allow many points to share a y-coordinate. We must provide a function f which given (in a single word) an array with $O(\log^{1-\epsilon} N/\log u)$ ids, in constant time returns an array where the ids are sorted according to the order the elements with these ids have on the y-axis.*

Proof. Looking into the proofs of Lemma 3.27, Lemma 3.28 and Lemma 3.29 together with the construction of Section 3.4 we see that it is sufficient to show that we can get a dynamic y-axis in the described way in the structure of Lemma 3.26. From the proof of this lemma it follows that it is sufficient that we can get a dynamic x-axis in the described way in Lemma 3.25. When inserting a point into this structure we will use the id of the x-coordinate as x-coordinate. Looking into the proof of Lemma 3.25 we see that the x-coordinates of points are not used in the maintenance of the tree T'. They are, however used in the secondary structures which are the structures of Lemma 3.20. Here, they are used as x-coordinates of pebbles and as keys of the WBB tree. However, the function f will give us all the information we need about the order of these coordinates. Further, it will do it fast enough because the construction in Lemma 3.20 never handles more than $O(\log^{1-\epsilon} N/\log u)$ pebbles at the same. □

3.7.3 Two long axes

We are now ready to prove the following theorem which for $n = N$ shows the first result from the introduction for $d = 2$:

Theorem 3.31. *For any constant $\gamma > 0$ there exists a structure with type* $\mathrm{R}^4(\mathrm{d}:n, \mathrm{d}:n)$ *and performance* $(\log^{7/8+\gamma} N, \log N/\log\log N, \log^{1+7/8+\gamma} N)$.

The structure we construct in the proof of the theorem is a kind of range tree (see Section 3.1.3). Traditionally, range trees have degree 2. Our range tree are split in two parts. In the *top part*, nodes have degree u of Lemma 3.30 and we keep the points of an internal node in a structure of Lemma 3.30. In the *bottom part*, nodes have constant degree. The reason we need the bottom part is that the structure of Lemma 3.30 does not have a completely dynamic y-axis. The bottom part ensures that nodes in the top part have sufficiently many points below them to allow the table to compute the function f of Lemma 3.30 to be build. Finally, to keep the query time low we keep in some levels of the tree the points in the structure of Theorem 3.24.

Proof of theorem 3.31. Let R be the structure with type $\mathrm{R}^4(\mathrm{d}:n, \mathrm{d}:n)$ we want to design. We store the elements of R in a WBB tree T with leaf parameter 1 and for $e \in R$ we use $e.\mathrm{y}$ as key. We specify the degree of T in a moment. We let v_r denote the root of T. We first ignore splitting of nodes and marking of keys. We keep in each node $v \in T$ a linked list $v.L$ containing the points located in the leaves descendant to v sorted according to their x-coordinate. Let $v \in T$ be an internal node. We color the children of v such that no two different children of v have the same color. For each $e \in v.L$ there is exactly one child v' of v and one $e' \in v'.L$ such that e and e' represent the same point. We color e with the color of v' and we keep in e a pointer to e' (such a pointer is called a *downpointer* [108]). We keep the elements of $v.L$ in a colored predecessor structure of Theorem 3.15 using the assigned colors. The update and query time of this structure is $O(\log^2 \log |v.L|)$; we will take it to be $O(\log^2 \log N)$.

Suppose we are given a query (x_1, x_2, y_1, y_2) in R. We observe that $v_r.L$ contains all points of R sorted according to their x-coordinate. The coordinates x_1 and x_2 identifies two elements in $v_r.L$ corresponding to the query interval on the x-axis. Further, using the pointers of the nodes of T to their parents we can identify the simple path in T between v_r and the leaf identified by y_1. We proceed from v_r down along this path using the downpointers and the colored predecessor structures to maintain the query interval on $v.L$ for the nodes $v \in T$ we meet. Beginning in some node of T, we must report the elements in this interval that has a relevant set of colors. After this, we repeat the same process with y_2 instead y_1 and we are done.

Inserting a point p in R is performed in a way similar to the way a query is answered. $p.\mathrm{x}$ identifies a position in $v_r.L$ where the point should be inserted. Also, $p.\mathrm{y}$ identifies a simple path in T between the leaf with $p.\mathrm{y}$ and v_r. We proceed from v_r down along this path inserting p in $v.L$ for the nodes v we meet and using the downpointers and the colored predecessor structures to maintain the position to insert in.

Deleting a point p from R is done as follows. $p.\mathrm{x}$ determines the location of p in $v_r.L$. We then delete p from $v_r.L$ and use the downpointers to remove p from the other lists $v.L$ it is in. We do not delete p from T. Instead, we use global rebuilding [87] to ensure that at most half of the elements of T are deleted points.

Let $\epsilon > 0$ be a constant to be determined in a moment and suppose $v \in T$. If v has a height smaller than $\log^{1-\epsilon} N$ we say v is in the bottom part and otherwise we say v is in the top part of T. We will give v different degree parameter depending on whether it is in the bottom or top part. It is straight forward to modify WBB trees to support that. If a node w has a child v in the bottom part we say that w is a leaf of the top part and that v is a root of the bottom part.

Below we will analyze the time we need to use in the top and bottom part when performing an update or a query. Also, we will analyze how much space is used in the top and bottom part. Our results are summarized in the following table.

	Top part	Bottom part
Update time	$O(\log^{6/7+\epsilon/7} N \log^3 \log N)$	$O(\log^{1-\epsilon} N \log^2 \log N)$
Query time	$O(\log N / \log\log N)$	$O(\log^{1-\epsilon} N \log^2 \log N)$
Bits used	$O(n \log^{1+6/7+\epsilon/7} N)$	$O(n \log^{2+\epsilon} N)$

Ignoring $\log\log N$ factors the update time and space usage in the upper and lower part becomes identical if we select $\epsilon = 1/8$. The values here then become smaller than ones claimed in the lemma.

Suppose $v \in T$ is a node in the bottom part. We then give v degree parameter 8 which is the minimal degree parameter we allow for WBB trees. We note that according to Lemma 3.7 property 5 a subtree rooted at a root in the bottom part of T has $\Theta(8^{\log^{1-\epsilon} N})$ leaves and according to property 3 it has height $O(\log^{1-\epsilon} N)$. If we link the elements of $v.L$ with the same color together in order, we only need to spend time $O(\log^2 \log N)$ plus constant time for each point reported, when we visit v during a query. Also, when visiting v during an update, we need to spend time $O(\log^2 \log N)$. We conclude that we get update and query time $O(\log^{1-\epsilon} N \log^2 \log N)$ in the bottom part of T. Since each point of R is stored in exactly one list $v.L$ for each level of T the total space usage for the bottom part becomes $O(n \log^{2-\epsilon} N)$ bits.

Suppose now $v \in T$ is a node in the top part. We store the elements of $v.L$ in the structure of Lemma 3.30 using the colors as ids of y-coordinates. For this reason we give v degree parameter $u = 2^{\log^{(1-\epsilon)/7} N}$ and note that with this selection the structure of Lemma 3.30 with an x-axis of length m gets performance $(\log^3 \log N, \log u \log\log N, m \log N)$. The maximal length of a simple path from the root of T to a leaf of the top part is $O(\log N / \log u) = O(\log^{6/7+\epsilon/7} N)$. We keep in v a table $v.G$ with length $8^{\log^{1-\epsilon} N}$. Such a table is large enough to allow f needed by Lemma 3.30 to be computed in constant time. Also, it will not increase the space usage except for a constant factor. The update time in the top part becomes $O(\log^{6/7+\epsilon/7} N \log^3 \log N)$ and the space usage becomes $O((\log N / \log u) n \log N) = O(n \log^{1+6/7+\epsilon/7} N$ bits again because each point of R is stored in exactly one list $v.L$ for each level of T.

The query time in the top part of T is $O((\log N / \log u) \log u \log\log N) = O(\log N \log\log N)$ and this is an $O(\log^2 \log N)$ factor too high. We fix this problem as follows. For each node v in the top part of T which is on a level divisible by $\log N / ((\log^2 \log N) \log u)$, we keep two structures of Theorem 3.24. In these structures we save the points of $v.L$ in the style of Section 3.7.1. We then modify the way we answer queries as follows. When we in our proceeding down in T meet a node v in the top part of T on a level divisible by $\log N / ((\log^2 \log N) \log u)$ *and* when we have tried to report points from $w.L$ for the parent w of v we stop. We then perform a query in one of the structures of Theorem 3.24 kept in v. After this, we do not need to proceed further down in T. This modification reduces the query time in the top part of T to $O(\log N / \log\log N)$. Further, since each point of R is stored in at most $O(\log^2 \log N)$ structures of Lemma 3.24, the time and space usage is not increased except for constant factors.

We now describe how to handle marking of keys and splitting of nodes in T. Consider the situation where a key k in a node v with parent w is marked (to simplify the discussion we assume that w exists or equivalently that v is not the root). Suppose v will eventually split into itself and a new node v' which will become the right sibling of v. We then select a color c' for v' such that no other current or coming child of w has color c'. During the time v is marked we move the points from $v.L$ with keys larger than k to $v'.L$. When a point p is moved we also color p with c' in $w.L$. Using the remark following Lemma 3.7 things can be adjusted such that we are finished when v split such that v can be split in constant time. We have to adjust queries such that they can handle partly split nodes. The details are tedious but standard and are thus omitted. If v contains a structure of Theorem 3.24 the splitting of this structure can be handled in a way similar to the one just described. One problem remains. If v is in the top part of T we need to insert c' on the y-axis of the structure of Lemma 3.30 in w. This is done by updating $w.G$. Since w is in the top part of T it follows from Lemma 3.7 item 5 that $\Omega(2^{\log^{1-\epsilon} N})$ insertions of elements descendant to v will be performed before v splits so we have time to rebuild $w.G$. We rebuild parts of $w.G$ during these insertions as follows (stating that all of $w.G$ should be rebuild does not make sense because several children of w may be marked at the same time). When a key in v is marked there is a current set C of colors given to current or coming children of w. During the insertions of elements descendant to v before v splits we fill in all entries of $w.G$ which compare colors from $C \cup \{c'\}$. This will ensure that $w.G$ can always compare any set of colors of the children of w. In order to make queries work while v is splitting we actually need to update $w.G$ before we start building up $v'.L$, building up $v''.L$ and changing color of elements in $w.L$. □

3.8 Higher dimensions

We will say a structure for the d-dimensional range reporting problem in the list order variant has type R_n^d if it can contain n points. Further, we will say such a structure has performance (U, Q, S) if it has update time $O(U)$, query time $O(Q)$ and space usage $O(S)$ bits. As in Section 3.3.1 we allow U, Q and S to depend on d and n but not on the number of points stored. This section is devoted to the proof of the following theorem which for $n = N$ shows the first result from the introduction.

Theorem 3.32. *Let $d \geq 2$ and $\epsilon > 0$ be any constants and define $\omega = 7/8+\epsilon$. Then for any $n \leq N$ there exists a structure for R_n^d with performance* $(\log^{d-2+\omega} N, (\log N/\log\log N)^{d-1}, n\log^{d-1+\omega} N)$.

Our proof of Theorem 3.32 is based on Lemma 3.33 below which is a dynamization of a method proposed by Alstrup, Brodal and Rauhe [6]. They described how to extended the dimension of a range reporting structure in a static setting where updates were not allowed.

Lemma 3.33. *Suppose for any n we have a data structure with type R_n^d and performance (U_n, Q_n, S_n) where U_n and Q_n are non-decreasing functions of n and S_n is a subadditive function of n. Then for any n and any constant $\epsilon > 0$ we can make a structure for R_n^{d+1} with performance $((U_n + \log^2\log n)\log^{1+\epsilon} n, (Q_n + \log^2\log n)\log n/\log\log n, S_n \log^{1+\epsilon} n)$.*

Proof. Let R be the structure with type R_n^{d+1} we want to make. For the proof we fix an axis of R, say, the x-axis. We store the elements of R in a WBB tree T with degree parameter $\log^{\epsilon/2} n$ and leaf parameter 1. For $e \in R$ we use $e.\mathrm{x}$ as key where $e.\mathrm{x}$ is the x-coordinate of e. We first ignore deletion of points, splitting of nodes and marking of keys.

Lemma 3.7 item 3 gives that T gets height $O(\log n/\log\log n)$. Let $v \in T$ be an internal node with height h and t children. For each pair (i, j) where $1 \leq i \leq j \leq t$ we keep a secondary structure $v.R_{(i,j)}$ with the given type R_m^d where $m = 4\log^{h\epsilon/2} n$. Let $v_1, \ldots, v_t$ be the ordered children of v. We then in $v.R_{(i,j)}$ save all points which are in leaves descendant to v_k where $i \leq k \leq j$ and where we ignore the x-coordinates when we store the points. Lemma 3.7 item 5 gives that there is room for these points in $v.R_{(i,j)}$.

We give each secondary structure of T a unique color. For any axis $R.L_\mathrm{y}$ different from the x-axis let $v.R_{(i,j)}.L_\mathrm{y}$ be the same axis of the secondary structure $v.R_{(i,j)}$. We then maintain pointers between the elements of $v.R_{(i,j)}.L_\mathrm{y}$ and the corresponding elements of $R.L_\mathrm{y}$. Each element in $R.L_\mathrm{y}$ will contain a number of pointers and we use the BFAT of Theorem 3.6 (as a dictionary) to store these. Also, we maintain a structure of Theorem 3.15 on $R.L_{\mathrm{y}}$ where an element has the colors of the secondary structures it has pointers to (Theorem 3.15 can easily be generalized to allow an element to have any number of colors).

Assume we are given a query in R. By following pointers from nodes of T to their parents we can in linear time identify $O(\log n/\log\log n)$ secondary structures such that asking the query in these structures (ignoring the x-axis) gives the desired points. For each secondary structure the position on the axes in which we should perform the query can be found in time $O(\log^2 \log n)$ using the structure of Theorem 3.15 maintained on the axes of R. This shows that the query time becomes $O((Q_n + \log^2 \log n) \log n/\log\log n)$ as claimed (Q_n is a non-decreasing function of n).

Each point is stored in $O(\log^\epsilon n)$ secondary structures in $O(\log n/\log\log n)$ nodes of T. When inserting new points the updates to perform in the secondary structures can be found in a way similar to when answering queries and the update time becomes $O((U_n+\log^2 \log n) \log^{1+\epsilon} n/\log\log n)$ (U_n is a non-decreasing function of n). The space usage becomes $O(S_n \log^{1+\epsilon} n/\log\log n)$ bits (because S_n is a subadditive function of n).

Marking of keys and splitting of nodes can be handled as described after the remark of Lemma 3.7. The details are, like in the proof of Theorem 3.31, tedious but standard and are thus omitted. Finally, we delete a point by removing it from the axes of R and from the secondary structures in which it is located. We never delete elements from T. Instead, we use global rebuilding [87] to ensure that at most half of the elements of T are deleted points. □

Finally we have:

Proof of Theorem 3.32. If $d = 2$ Theorem 3.31 gives the desired result. Otherwise, we get the desired structure by applying Lemma 3.33 $d-2$ times to Theorem 3.31 where we set $n = N$. □

3.9 Open problems

We have given an optimal structure in terms of time per operation for the 2-dimensional range reporting on RAM. Also, we have given a new solution for the d-dimensional range reporting problem for any constant dimension $d \geq 3$. A very interesting open problem is to prove or disprove that our solution is optimal for any constant dimension $d \geq 3$. Showing a matching lower bound seems to be well beyond current lower-bound techniques.

The construction leading to our 2-dimensional range reporting structure (Theorem 3.31) is rather involved. Of course, it would be nice to have a simpler construction obtaining the same or a similar bound.

For query time $O(\log n/\log\log n)$, the update time of our two-dimensional range reporting structures is $O(\log^\omega n)$ for a constant $\omega < 1$. However, the lower bound of [8] does not prevent even constant update time in this case. An open problem is to close or narrow the gap between these two update times.

The basic technique in the data structures of this paper has been to pack many points together in a single word. This has allowed us to process points in subconstant time per point. This idea is not new and is very similar to what is done in the I/O model of computation [3] where elements (or points) are packed together on disk blocks. For this reason, data structures for the I/O model and for the RAM model of computation have often inspired each other and we have also used ideas from the I/O model in this paper. An open problem is to use ideas of this paper in the I/O model. In particular, it may be possible to reduce the update time of the structure of Arge et al. [13] for dynamic range reporting in the I/O-model of computation by using ideas from this paper. Such a solution may even be practical.

3.10 Acknowledgments

I am very grateful to my supervisors Stephen Alstrup and Theis Rauhe for introducing me to the range reporting problem, for bringing my attention to [22] and for helpful discussions in connection with this work. I would like to thank Dan Willard for clarifying the history of range trees. Finally, I would like to thank the anonymous referees for comments which have helped me to improve the presentation.

Chapter 4

On dynamic range reporting in one dimension

On dynamic range reporting in one dimension

Gerth Stølting Brodal
BRICS
Dept. of Computer Science
University of Aarhus
Email: gerth@brics.dk

Christian Worm Mortensen
IT University of Copenhagen
Email: cworm@itu.dk

Rasmus Pagh
IT University of Copenhagen
Email: pagh@itu.dk

Abstract

In this paper we study problems related to dynamic range reporting in one dimension on a RAM with word size w. Let $1/\log w \leq \gamma \leq 1$ be a parameter. We show how to maintain a data structure for a dynamic set of keys $K \subseteq [0..2^w - 1]$ such that with high probability keys can be inserted and deleted in time $O(w^\gamma/\gamma)$ and such that we given i and j in worst case time $O(1/\gamma)$ can find a key from $K \cap [i..j]$ or determine that no such key exists. For constant query time the update time is exponentially smaller than the update time needed if we have to answer the related predecessor query. The space usage of the structure is linear and it builds on a new optimal scheme for perfect hashing for a data stream which may be of independent interest. We conclude the paper with a linear-size data structure for dynamic approximate range counting.

4.1 Introduction

One of the most fundamental problems in the area of algorithms is the dynamic dictionary problem. This is the problem of maintaining a set K of keys under insertions and deletions while supporting a lookup query which should tell whether a given query key is in K. A widely studied generalization of the lookup query is the predecessor query which should find the largest key in K no larger than a given query key. In this paper we study the findany query which is more general than the lookup query but less general than the predecessor query. A findany query must find a key in K between two given query keys or determine that no such key exists. Algorithms that have traditionally used predecessor queries can sometimes be modified to use findany queries instead. In particular, this is the case for algorithms solving the one-dimensional range reporting problem which is to report all keys in K between two given query keys. In this paper we give a tradeoff for the update and query time for findany queries. In one end of the tradeoff we have a structure with the same performance as the classical predecessor structure of van Emde Boas et al. [105]. In the other end we have a structure with worst case constant query time and low update time with high probability. In contrast it is not possible to support predecessor queries in constant time unless exponentially higher update time is accepted. An important tool in our structure supporting findany queries is a new optimal scheme for perfect hashing for a data stream which may be of independent interest. Besides studying findany queries we also study approximate range counting (or just count) queries. This is the problem of approximating the number of keys in K between two given query keys. For any constant $\epsilon > 0$ we show how to approximate this number within a factor $1 - \epsilon$ in worst case constant time while keeping a low amortized update time with high probability.

The problems of answering findany and count queries have also been studied by Alstrup, Brodal and Rauhe [5]. They showed how to answer findany and count queries in a static setting where K is given once and for all. An implication of our results is a static structure for findany queries which has the same query time as the one in [5] but which can be constructed faster. We refer the reader to [5] for applications of the findany and count queries.

4.1.1 Model of computation

Our model of computation is a unit cost RAM and through out the paper we let w denote the word size. We assume the instruction set includes the standard computational operations on words (addition, subtraction, multiplication, shifting, bitwise and, bitwise or, and bitwise xor). Also we assume we have a constant time msb operation which can compute the position of the most significant bit set in a word. Fredman and Willard [51] have shown how the msb operation can be computed in constant time by the use of a constant number of precomputed words which only depend on w. We assume we have access to a sequence of truly random words. We measure space usage in words if not otherwise noted.

4.1.2 Main results

Define $U = [0..2^w - 1]$ where for integers $i \leq j$ we let $[i..j]$ denote the set of integers t for which $i \leq t \leq j$. We show how to maintain a dynamic set of keys $K \subseteq U$ under insertions and deletions while supporting findany and count queries defined as follows. A findany(i, j) query with $i, j \in U$ and $i \leq j$ must find a key in the set $K \cap [i..j]$ or determine that the set is empty. A count(ϵ, i, j) query with $i, j \in U$, $i \leq j$ and $0 < \epsilon < 1$ must find a number s' such that $(1 - \epsilon)s \leq s' \leq s$ where $s = |K \cap [i..j]|$. Let $n = |K|$ denote the number of keys. We then show the following.

Theorem 4.1. *For any constant $k \geq 2$ and $1/\log w \leq \gamma \leq 1$ the set K can be maintained using time amortized $O(w^\gamma/\gamma)$ for updates w.h.p.*, worst case time $O(1/\gamma)$ for* findany *queries and worst case time $O(1/\gamma + \log(1/\epsilon))$ for* count(ϵ, i, j) *queries provided $1/\log^k n < \epsilon$. The space usage is $O(n)$. If we only support* findany *queries the time bound for updates are worst case w.h.p..*

Our structure links the keys of K together in order. So we can answer a one-dimensional range reporting query, that is, report all keys in $K \cap [i..j]$ as follows. First we perform a findany(i, j) query and then we follow the links to find the remaining keys. This takes time $O(1/\gamma + k)$ where k is the number of reported keys. As a special case of Theorem 4.1 consider the case where $0 < \gamma \leq 1$ is constant. We then get update time $O(w^\gamma)$, constant time for findany queries and constant time for count(ϵ, i, j) queries provided $\epsilon > 0$ is also constant.

4.1.3 Relation to other results

The problems of Theorem 4.1 have been studied by Alstrup, Brodal and Rauhe [5] in a static setting. Here the n keys were assumed to be given once and for all. These keys were then preprocessed into a data structure supporting findany and count queries. We can make our dynamic result into a static result by inserting the n keys into an initially empty data structure of Theorem 4.1. We now discuss how this structure relates to the structure of [5] in case we select $0 < \gamma \leq 1$ to be constant. The structure will use space $O(n)$, support findany queries in time $O(1)$ and count(ϵ, i, j) queries in time $O(\log(1/\epsilon))$. This matches the performance of the

*We use w.h.p. as an abbreviation for with high probability. In a data structure with n elements we let high probability mean probability at least $1 - n^{-c}$ for any constant $c > 0$.

structure of [5] except that in [5] $\epsilon > 0$ can be arbitrary while we require $\epsilon > 1/\log^k n$ for a constant $k \geq 2$. On the contrary, our structure can be constructed in time $O(nw^\gamma)$ w.h.p. while the structure of [5] requires construction time expected $O(n\sqrt{w})$.

Miltersen et al. [77] also described a static data structure which could answer findany queries in constant time. This structure can be dynamized using update time $O(w)$. The space usage of the structure, however, is $O(nw)$ and it is not clear how to reduce this to linear.

4.1.4 Relation to the predecessor problem

The previously most efficient dynamic data structures for findany (and one-dimensional range reporting) queries actually supported the more general predecessor queries. The complexity of (static) predecessor queries is $\Omega(\min(\sqrt{\log n/\log\log n}, \log w/\log\log w))$ if $n^{O(1)}w$ bits of space is to be used [15]. The lower bound is nearly matched in the dynamic case, with queries and updates in time $O(\min(\sqrt{\log n/\log\log n}, \log\log n \log w/\log\log w))$, and linear space $O(nw)$ bits [11, 15].

If we require $o(\log w/\log\log w)$ query time, then by [15, Theorem 3.7] there is a set of size $n(w)$ (i.e., a function of w) such that the space usage of the data structure is $\Omega(n(w)2^{w^{1-\rho}})$, for any constant $\rho > 0$. In particular, inserting the $n(w)$ keys in this data structure must take amortized time $\Omega(2^{w^{1-\rho}})$ per key. (Otherwise we could reduce the space using a dictionary.) This is exponentially higher than our update time when supporting findany queries.

4.1.5 Orthogonal range reporting in high dimensions

As observed in [5] an important application of findany and count queries is the orthogonal range reporting problem in high dimensions δ. In a dynamic setting where points have integral coordinates this problem is to maintain a set $P \subseteq U^\delta$ of n points under insertions and deletions such that given $x_i, x'_i \in U$, $x_i \leq x'_i$ for $1 \leq i \leq \delta$ the points in $P \cap [x_1..x'_1] \times \cdots \times [x_\delta..x'_\delta]$ can be reported. For constant δ, the asymptotic fastest known structures for this problem [78, 79] have a time usage which make them impractical even when δ is just a small constant because a linear scan of the points would be faster. A heuristic optimization of the linear scan is to keep the points sorted for each dimension. For a given query we can then for each dimension make a count query to find a dimension with approximately fewest points in the query interval and then only scan the points which are within the query interval in the found dimension. The points to scan can be found by a one-dimensional range reporting query. Since the best solution for dynamic (exact) range counting uses time $O(\log n/\log\log n)$ for updates and queries [35] the previously fastest known solution for this heuristic uses update time $O(\delta \log n/\log\log n)$ and query time $O(\delta \log n/\log\log n + \delta k)$ where k is the number of points to scan. Using our structures with $t = 1/\gamma$ for a constant γ we get update time $O(\delta w^\gamma)$ and optimal query time $O(\delta + \delta k)$. The space usage is linear $O(\delta n)$.

4.1.6 Paper outline

We start with preliminaries in Section 4.2. In Section 4.3, 4.4 and 4.5 we describe how to answer findany queries as follows. In Section 4.3 we describe our new scheme for perfect hashing for a data stream. In Section 4.4 we use the scheme to make a dictionary for disjoint simple paths on a complete tree. This dictionary is used in Section 4.5 where we describe how to answer findany queries. Finally we in Section 4.6 discuss how to answer count queries deferring some material to appendix.

4.2 Preliminaries

In general we do not distinguish between a data structure and the set of elements it contains. We let log denote the base-two logarithm. We assume w.l.o.g. (without loss of generality) that w is a power of two.

In this paper a tree is always assumed to be rooted and the children of a node are assumed to be ordered from left to right. In the following, let T be a tree with root $r \in T$. The leaves of T are assumed to be numbered from 0 in the order we meet them in a depth first search where we visit the children of an internal node in order. The *depth* of a node $v \in T$ is the distance between v and r. The *height* of a leaf is 0 and the height of an internal node is 1 plus the maximum height of its children. The height of T is the height of r. We use $\mathrm{sub}_T(v)$ to denote the subtree of T rooted at v. If v' is an ancestor of v in T we let $[v \rightsquigarrow v']$ denote the path between v and v' and we let $]v \rightsquigarrow v'[$ denote the path strictly between v and v'.

We say T has degree d if each node in T has at most d children. If T has height h and degree d we say T is *complete* if T has exactly d^h leaves. If T is a tree with degree d we assign labels to the nodes of T as follows. The root has label 1. If a node v has label l then the ith child of v has label $l \cdot d + i - 1$. Note that in general the label of a leaf is different from its number.

If B is a binary tree and $v \in B$ is a node we let left(v), right(v) and parent(v) denote the left child, right child and parent of v respectively. We note that given the label of v we can find the label of these nodes in constant time.

We let a VEB be the data structure of van Emde Boas et al. [105] combined with hashing [39]. Such a data structure can maintain a set of n keys from U using space $O(n)$ such that keys can be inserted and deleted in time $O(\log w)$ w.h.p. and predecessor queries can be performed in worst case time $O(\log w)$.

We will use the q-heap data structure of Fredman and Willard [52]. A q-heap allows a set S of words where S has cardinality $O(\log^{1/4} n)$ to be stored such that elements can be inserted in and deleted from S in constant time and such that given a query integer q the largest element of S smaller than q can be identified in constant time. The q-heap requires a precomputed table with size $O(n)$ computable in $O(n)$ time and this table only depends on the word size and on n. Further, the q-heap requires an msb operation as discussed in Section 4.1.1. We will assume a q-heap allows S to have size $O(\log n)$. This assumption is valid because we can maintain a tree with constant height and degree $O(\log^{1/4} n)$ keeping a normal q-heap in each internal node and the elements of S in the leaves.

4.3 Data stream perfect hashing

In this section we show how to space-efficiently maintain a perfect hash function (i.e., a 1-1 function) for a set under an online sequence insertions and deletions of keys. What we maintain is a function in the sense that the value returned on a key does not change as long as the key remains in the set. A key that is removed and later reinserted may have its hash function value changed.

Let u be the size of the key space. Consider a sequence of operations, where each operation is either `insert`(x) or `delete`(x), where $x \in [0..u-1]$. If a sequence of sets $K_0, K_1, \ldots$ exists such that $K_0 = \emptyset$, $K_i \backslash K_{i-1} = \{x\}$ if the ith operation is `insert`(x), and $K_{i-1} \backslash K_i = \{x\}$ if the ith operation is `delete`(x), we say the sequence of operations is *valid*, and that $K_0, K_1, \ldots$ *corresponds* to the sequence of operations. We show the following.

Theorem 4.2. *Let n and u be positive integers such that $u = 2^{O(w)}$. Consider an online sequence of $n^{O(1)}$ valid insertions and deletions of elements in $[0..u-1]$, and let $K_0, K_1, \ldots$*

be the corresponding sequence of sets. Assuming $|K_i| \leq n$ for any i, we can maintain a data structure supporting the constant time query `eval`*(x) for $x \in [0..u-1]$, returning a value in $[1..m]$, where $m = n + o(n)$, such that:*

- *After update operation i,* `eval` *is 1-1 on K_i.*
- *If $x \in K_i \cap K_{i+1} \cap \cdots \cap K_j$ then* `eval`*(x) is constant between update operation i and j.*

After $o(n)$ preprocessing time, the processing time per update operation is constant w.h.p.[†] *and the space usage is $O(n \log\log u + w)$ bits w.h.p.*

The central property of the data structure of Theorem 4.2 is that it uses space sublinear in the space required for storing a set of size n (plus $O(1)$ machine words). It can be seen as perfect hashing for a data stream of updates, data stream data structures being characterized by using sublinear space. Previous construction algorithms for perfect hash functions have required all keys to be present in memory, using space $\Omega(nw)$ bits. In the full version of this paper we show the following theorem which establishes the optimality of Theorem 4.2.

Theorem 4.3. *If $u > n^{1+\epsilon}$ for a constant $\epsilon > 0$ then any data structure maintaining a perfect hash function under an online sequence of n insertions of keys from a universe of size u must use expected space $\Omega(n \log\log u)$ bits.*

We mention an independent application of Theorem 4.2. In a database we can maintain an index of a relation under insertions of tuples, using internal memory per tuple which is logarithmic in the length of the key for the tuple. If tuples have fixed length, they can be placed directly in the hash table, and need only be moved if the capacity of the hash table is exceeded.

4.3.1 The perfect hashing data structure

Our data structure has the following parts:

- A hash function $\rho : [0..u-1] \to \{0,1\}^v$, where $v = O(\log n)$, from the universal hash function family of [38].
- A hash function $\phi : \{0,1\}^v \to \{1,\ldots,r\}$, where $r = \lceil n/\log^2 n \rceil$, taken from Siegel's class of highly independent hash functions [101, Theorem 3].
- An array of hash functions $h_1,\ldots,h_r : \{0,1\}^v \to \{0,1\}^s$, where $s = \lceil (6+2c)\log\log u \rceil$, independently chosen from a universal hash function family. The constant c is specified below.
- A high performance dictionary [39] for a subset K'_i of the keys in K_i. The dictionary should have a capacity of $O(1 + n/\log u)$ keys (but might expand further). Along with the dictionary we store a linked list of length $O(1 + n/\log u)$, specifying certain vacant positions in the hash table.
- An array of dictionaries $D_1,\ldots,D_r$, where D_i is a dictionary that holds the key $h_i(\rho(k))$ for each key $k \in K_i \backslash K'_i$ where $\phi(\rho(k)) = i$. A unique value in $[0..j-1]$, where $j = (1+o(1))\log^2 n$, is associated with each key in D_i. A bit vector of j bits and an additional string of $\log n$ bits is used to keep track of which associated values are in use. We return to the exact choice of j and the implementation of the dictionaries later.

The hash function ρ is used to reduce the key length to v, in the sense that with high probability there will be no two keys in K_t having the same ρ-value, for any t. The constant in $v = O(\log n)$ depends on c in the high probability bound and the length of the sequence operations.

If operation t is `insert`(k), the new key k is included in K'_t if either

[†]In Section 4.3 we let high probability mean with probability at least $1 - n^{-c}$ where $c > 0$ can be chosen to any desired constant.

- There are j keys in D_i, where $i = \phi(\rho(k))$, or
- There exists a key $k' \in K_{t-1}$ where $\phi(\rho(k)) = \phi(\rho(k'))$ and $h_i(\rho(k)) = h_i(\rho(k'))$.

Otherwise k is associated with the key $h_i(\rho(k))$ in D_i. The idea is that all dictionaries in the construction assign to each of its keys a unique value within a subinterval of $[1..m]$. Each of the dictionaries $D_1, \ldots, D_r$ is responsible for an interval of size j, and the high performance dictionary is responsible for an interval of size $O(n/\log u) = o(n)$ (it may be assumed w.l.o.g. that $u \geq n$). Deletion of a key k is done in K'_i if $k \in K'_i$, and otherwise the associated key in the appropriate D_i is deleted.

To evaluate the perfect hash function on a key k we:

1. First see whether k is in the high performance dictionary. If so, we return the value associated with k.
2. Otherwise we compute $i = \phi(\rho(k))$ and look up the value Δ associated with the key $h_i(\rho(k))$ in D_i. Return $(i-1)j + \Delta$, i.e., position Δ within the ith interval.

Since $D_1, \ldots, D_r$ store keys and associated values of $O(\log\log n)$ bits, they can be efficiently implemented as constant depth search trees of degree $w^{\Omega(1)}$, where each internal node resides in a single machine word. This yields constant time for dictionary insertions and lookups, with a space usage of $O(\log^2 n \log\log u)$ bits for each dictionary. We do not go into details of the implementation as they are standard, but refer to [59] for explanation of the required word-level parallelism techniques.

What remains to describe is how the dictionaries keep track of vacant positions in the hash table in constant time per insertion and deletion. The high performance dictionary simply keeps a linked list of all vacant positions in its interval. Each of $D_1, \ldots, D_r$ maintain a bit vector indicating vacant positions, and additional $O(\log n)$ bits, each indicating the existence of any vacant position in an interval of size $O(\log n)$. This can again be done in constant time per operation, employing standard techniques.

Only $o(n)$ preprocessing is necessary for the data structure (essentially to build tables needed for the word-level parallelism). The major part of the data structure is initialized lazily.

4.3.2 Analysis

Since evaluation of all involved hash functions and lookup in the dictionaries takes constant time, evaluation of the perfect hash function is done in constant time. As we will see below, the high performance dictionary is empty with high probability unless $n/\log u > \sqrt{n}$. This means that it uses constant time per insertion with high probability. Hence, processing of a new key is done in constant time with high probability.

We now consider the space usage of our scheme. The function ρ can be represented in $O(w)$ bits. Siegel's highly independent hash function uses $o(n)$ bits of space. The hash functions $h_1, \ldots, h_r$ use $O(\log n + \log\log u)$ bits each, and $o(n \log\log u)$ bits in total. The main space bottleneck is the space for $D_1, \ldots, D_r$, which sums to $O(n \log\log u)$ (again w.l.o.g. assuming $n \leq u$).

Finally, we consider the space used by the high performance dictionary. We will show that it is $O(n)$ bits with high probability. This is done by showing that each of the following hold with high probability for all sets K_i in the sequence:

1. The function ρ is 1-1 on K_i.
2. There is no l such that $K_{i,l} = \{k \in K_i \mid \phi(\rho(k)) = l\}$ has more than j elements.
3. The set K'_i has $O(1 + n/\log u)$ elements.

That 1. holds with high probability is well known, see e.g. [39]. To show 2. we use the fact that, with high probability, Siegel's hash function is independent on every set of $n^{\Omega(1)}$ keys. We may thus employ Chernoff bounds for random variables with limited independence [96] to bound the probability that any l has $|K_{i,l}| > j$, conditioned on the fact that 1. holds. Specifically, we can use [95, Theorem 2.5.I.b] to argue that for any l, the probability that $|K_{i,l}| > j$ for $j = \lceil \log^2 n + \log^{5/3} n \rceil$ is $n^{-\omega(1)}$, which is negligible. On the assumption that 1. and 2. hold, we finally consider 3. We note that every key $k' \in K'_i$ is involved in a h_l-collision in $K_{i,l}$ for $l = \phi(\rho(k'))$, i.e., there exists $k'' \in K_{i,l} \backslash \{k'\}$ where $h_l(k') = h_l(k'')$. By universality, for any l the expected number of h_l-collisions in $K_{i,l}$ is $O(\log^4 n/(\log u)^{6+2c}) = O((\log u)^{-(2+2c)})$. Thus the probability of one or more collisions is $O((\log u)^{-(2+2c)})$. For $\log u \geq \sqrt{n}$ this means that there are no keys in K'_i with high probability. Specifically, c may be chosen as the sum of the constants in the exponents of the length of the operation sequence and the desired high probability bound. For the case $\log u < \sqrt{n}$ we note that the expected number of elements in K'_i is certainly $O(n/\log u)$. To see than this also holds with high probability, note that the event that one or more keys from $K_{i,l}$ end up in K' is independent among the l. Thus we can use Chernoff bounds to get that the deviation from the expectation is small with high probability.

4.4 A tree-path dictionary

Let T be a complete binary tree of height w. We will identify the nodes of T with their label. We now describe a dictionary D which can contain at most n simple paths from T. The paths must be disjoint and a path must go between a node and an ancestor of the node. The dictionary can be updated by inserting or deleting a path identified by its end nodes. It is the responsibility of the user of the dictionary to ensure that the paths which are in D at a particular time are disjoint. Given a node in T the dictionary can tell which path the node is on or that the node is not on any path. In this section we show:

Lemma 4.4. *For any parameter $1/\log w \leq \gamma \leq 1$ we can make the dictionary D with update time $O(w^\gamma/\gamma)$ w.h.p., query time worst case $O(1/\gamma)$ and space usage $O(n)$.*

If we are allowed to spend $O(\log w)$ time to precompute a word depending on γ and w, it is possible to interchange the update and the query time. But we always perform queries in connection with updates, so this will never be an advantage for us.

We set $d = w^\gamma$ and note that $2 \leq d \leq w$. The problem is then to make D with update time $O(d \log w/\log d)$, query time $O(\log w/\log d)$ and space usage $O(n)$. In the following we assume w.l.o.g. that d is a power of two. Let L be the complete tree with degree d and at least $w+2$ leaves which has lowest height. Let $v \in T$ be an arbitrary node of T and let e_v be the depth of v. We then assign a *layer* $\text{layer}(v)$ to v which is the height in L of the nearest common ancestor of the leaves in L with number e_v and $e_v + 1$. A path $[v \rightsquigarrow v']$ is then inserted in D in the following way:

I1. Write in v' that the path $[v \rightsquigarrow v']$ ends here.
I2. Set $u = v'$ and repeat:
    I2.1. Write in u the distance between u and v'
    I2.2. Set u to the nearest descendant of u on $[v \rightsquigarrow v']$ which has layer at least the same as u. Stop if no such node exists.

For the moment we ignore the space cost and assume we can write information in and read information from a node in constant time. Because of the way we assign layers to nodes the loop in item I2 is executed at most $O(d \log w/\log d)$ times (this is proportional to the height of L times its degree). Further, using word parallelism and the msb operation the whole operation can also be carried out in this time. Details will be given in the full version of the paper. A path is deleted in the same way as it is inserted except that instead of writing information in the nodes of T we remove the information that was written during the insertion of the path.

When given a query node $v \in T$ we identify the path v is on as follows:

Q1. Set $u = v$ and repeat:

  Q1.1. If there is a distance $h \geq 0$ in v check if the ancestor of v with distance h from v identifies a path containing v. If it does we have found the path and stop.

  Q1.2. Set v to the nearest ancestor of v which has a strictly higher layer than v. If no such node exists we are done and v is not on any path.

Because of the way we assign layers to nodes the loop in item Q1 is executed at most $O(\log w / \log d)$ times (this is proportional to the height of L). Again, we can also ensure that the operation can be carried out in this time.

We now argue for correctness. The argument works for any assignment of layers to the nodes of T. It is clear that if we report that a query node is on a path then this is indeed the case. So we must argue that we always find the path a given query node is on. So assume $[v \rightsquigarrow v'] \in D$ and we are given a query $u \in [v \rightsquigarrow v']$. Because of the way we perform insertions, any node in $[u \rightsquigarrow v']$ with maximal layer among the nodes in $[u \rightsquigarrow v']$ will have its distance to v' written. Further, in connection with queries we will visit exactly one such node. This concludes the argument for correctness.

We now discuss how we store information in the nodes of T. The information maintained in item I1 during insertions and deletions is stored using the hashing scheme of [39] with the label of the node as key. Since there is at most n paths in D this will use space $O(n)$. The distances maintained in item I2.1 during insertions and deletions are stored using the hashing scheme of Theorem 4.2 again using the labels of the nodes as keys.

We set n in the theorem to $O(nd \log w / \log d)$ which is proportional to the number of nodes that will contain a distance. If we in this scheme lookup a node in which there is currently no distance, nothing is guarantied about the result we get. But this is not a problem since our query algorithm will work no matter which distances are assigned to nodes not written during an insertion of a path currently in the dictionary.

The space usage will be $O(nd \log^2 w / \log d + w)$ bits w.h.p. (and this can be made worst case because the update time is w.h.p.). This space usage is $O(n)$ if $d / \log d < w / \log^2 w$ which we can assume w.l.o.g.. Finally, by using global rebuilding [87]. we ensure that no more than $O(nd \log w / \log d)$ insertions are performed in the dictionary.

4.5 The structure supporting findany queries

This section is devoted to the proof of the following lemma. The lemma proves that we can answer findany queries as claimed in Theorem 4.1. Let K be a dynamic set as in Section 4.1.2.

Lemma 4.5. *For any parameter $1/\log w \leq \gamma \leq 1$ the set K can be maintained using time $O(w^\gamma/\gamma)$ for updates w.h.p. and worst case time $O(1/\gamma)$ for* findany *queries. The space usage is $O(|K|)$.*

As in Section 4.4 we let T be a complete binary tree with height w and we identify the nodes of T with their label. We will not distinguish between an element $e \in U$ and the leaf of T with number e. It follows that we consider K a subset of leaves of T. We keep the elements of K in a VEB and we use this VEB to link the elements together in order. The set K induces a coloring on the nodes of T as follows. Leaves of T which are in K are black. Internal nodes which have a descendant black leaf in both their left and right subtree are black. An internal non-black node which is an ancestor of a black leaf is gray. Nodes which are not black or gray are white. W.l.o.g. we assume K always contains elements such that the root of T is black.

If $v \in T$ is a black or gray node we let $\mathrm{blacka}(v)$ denote the nearest black ancestor of v. If $v \in T$ is a black node we let $\mathrm{leafl}(v)$ (resp. $\mathrm{leafr}(v)$) be the left (resp. right) most black leaf in $\mathrm{sub}_T(v)$. If $v \in T$ is a black internal node we let $\mathrm{descl}(v)$ (resp. $\mathrm{descr}(v)$) denote the unique black node u in $\mathrm{sub}_T(\mathrm{left}(u))$ (resp. $\mathrm{sub}_T(\mathrm{right}(u))$) such that $\mathrm{blacka}(\mathrm{parent}(u)) = v$. We will explicitly maintain the set of black nodes and their descl and descr pointers using the hashing scheme of [39] with the label of the nodes as keys. In the next paragraph we describe how we compute blacka, leafl and leafr.

We first describe how to compute $\mathrm{blacka}(v)$ for a gray or black node $v \in T$. If v is black $\mathrm{blacka}(v) = v$ so assume v is gray. We maintain a dictionary D_a of Lemma 4.4. For all internal black nodes $u \in T$ we keep in D_a the paths $]\mathrm{descl}(u) \rightsquigarrow u[$ and $]\mathrm{descr}(u) \rightsquigarrow u[$. The node $\mathrm{blacka}(v)$ can then be found by a single lookup in D_a using time $O(1/\gamma)$. We next describe how to compute $\mathrm{leafl}(v)$ for a black node $v \in T$ ($\mathrm{leafr}(v)$ is computed in a symmetric way). We maintain a dictionary D_l of Lemma 4.4. Let $u \in T$ be any black leaf and let $w \in T$ be the black ancestor of u such that $\mathrm{leafl}(w) = u$ and either w is the root or $\mathrm{leafl}(\mathrm{blacka}(\mathrm{parent}(w))) \neq w$. We then keep in D_l the path $[u \rightsquigarrow w]$. The node $\mathrm{leafl}(v)$ can then be found by a single lookup in D_l using time $O(1/\gamma)$.

We now describe how to answer a $\mathrm{findany}(i, j)$ query. Let $v \in T$ be the nearest common ancestor of i and j. It is possible to find v in constant time using the msb operation. Let $v' = \mathrm{blacka}(v)$, $v'_l = \mathrm{descl}(v')$ and $v'_r = \mathrm{descr}(v')$. Any black leaf in T between i and j have either v'_l or v'_r as an ancestor. Further, neither v'_l nor v'_r is an ancestor of v. It follows, that if there is an element between i and j then one of $\mathrm{leafl}(v'_l)$, $\mathrm{leafl}(v'_r)$, $\mathrm{leafr}(v'_l)$ and $\mathrm{leafr}(v'_r)$ must be such an element so we check each of these leaves to see if it is in the interval $[i..j]$. The time to answer a query is $O(1)$ plus the time used by a constant number of lookups in the dictionary of Lemma 4.4. This give a total query time of $O(1/\gamma)$.

We now describe how to insert an element in K. Deletions are performed in a similar way. Let $v \in T$ be the new black leaf and let $w \in T$ be the nearest gray ancestor of v. There is no black nodes between $\mathrm{parent}(v)$ and w. Let v' be the black leaf closest to v which has w as an ancestor. There is no black leaves between v and v' so v' and w can be found using the VEB with the elements of K and the msb operation. We observe that that v and w are the only new black nodes and we color them black. Define $w' = \mathrm{blacka}(w)$. In the following we assume v is to the left of v' and that $w \in \mathrm{sub}_T(\mathrm{left}(w'))$. Symmetric descriptions apply in other cases. D_a contains a path $]\mathrm{descl}(w') \rightsquigarrow w'[$. We delete this path and insert the paths $]\mathrm{descl}(w') \rightsquigarrow w[$, $]v \rightsquigarrow w[$ and $]w \rightsquigarrow w'[$. Next, D_l contains a path $[v' \rightsquigarrow u]$ for some $u \in T$. We delete this path and insert the paths $[v \rightsquigarrow u]$ and $[v' \rightsquigarrow \mathrm{descl}(w')]$. Finally, we set $\mathrm{descl}(w) = v$, $\mathrm{descr}(w) = \mathrm{descl}(w')$ and then $\mathrm{descl}(w') = w$.

The time usage for updates is as follows ignoring constant time terms. First, we must use $O(\log w)$ time because we have to maintain and lookup in the VEBs associated with K. Next, we must perform a blacka query using time $O(1/\gamma)$. After this, we must perform a constant number of updates to the structure of Lemma 4.4 spending time $O(w^{\gamma}/\gamma)$ and this term dominates the time usage. Finally the space usage is linear so this concludes the proof of Lemma 4.5.

4.6 Supporting count queries

In this section we discuss the proof of the following lemma.

Lemma 4.6. *The set K can be maintained using time amortized $O(\log w)$ for updates w.h.p. and time worst case $O(\log(1/\epsilon))$ for* $\mathrm{count}(\epsilon, i, j)$ *queries provided $1/\log^k |K| < \epsilon$ for a constant*

$k \geq 2$ and either $i \in K$ or $j \in K$. The space usage is $O(|K|)$.

Theorem 4.1 is a corollary to Lemma 4.5 and Lemma 4.6: We maintain K in both the structure of Lemma 4.5 and the structure of Lemma 4.6. Suppose now we are given a $\text{count}(\epsilon, i, j)$ query where $i, j \in U$, $i \leq j$ and $1/\log^k |K| < \epsilon$. We then make a $\text{findany}(i, j)$ query in Lemma 4.5. If this gives an element $u \in K$ such that $i \leq u \leq j$ the answer to the query is $\text{count}(\epsilon, i, u-1) + \text{count}(\epsilon, u+1, j) + 1$. Otherwise, the answer is 0.

In this section we will prove lemma 4.7 below to which lemma 4.6 is a corollary. Assume every element $u \in K$ has a rational weight $1 \leq \text{weight}(u) \leq 3$. For $i, j \in U$ define:

$$s = \sum_{u \in K \cap [i..j]} \text{weight}(u) \tag{4.1}$$

If further $0 < \epsilon < 1$ we support a query $\text{weight}(\epsilon, i, j)$ which must return a number s' such that $(1-\epsilon)s \leq s' < s$. We have:

Lemma 4.7. *The set K can be maintained using time amortized $O(\log^3 n)$ for insertions and deletions, time $O(\log n)$ for changing an element weight and time $O(\log(1/\epsilon))$ for $\text{weight}(\epsilon, i, j)$ queries provided $i \in K$ or $j \in K$. Further, if in such a query $\epsilon < 1/s$ the predecessor of j in K is identified. The space usage is $O(n \log n)$.*

In appendix we prove lemma 4.6 as a corollary to lemma 4.7. The basic idea of the proof of this is to divide the keys into blocks with $\Theta(\log^{k+1} n)$ elements. The keys of each block are then kept in a balanced binary tree. Each tree is then inserted into lemma 4.7 with a weight approximately proportional to the number of elements in the tree.

Before proving lemma 4.7 we describe a special binary search tree B. B contains live and dead nodes. The user may update B by inserting a new element as a live leaf or by marking a live node as dead. In connection with updates B can chose to rebuild one or more subtrees $\text{sub}_B(v)$ using time $O(|\text{sub}_B(v)|)$. The rebuilding discards all dead nodes of $\text{sub}_B(v)$ but no changes occur in other parts of B. The following lemma can be extracted from Andersson and Lai [10]

Lemma 4.8. *Let n be a power of two and assume B contains between n and $n/2$ live nodes. Then there exists a structure for B such that:*

1. *Updates can be performed in time amortized $O(\log^2 n)$.*
2. *For each $v \in B$ at least half of the nodes in $\text{sub}_B(v)$ are live.*
3. *There exists a constant $\delta > 0$ such that all leaves of B have depth in $[\log n - \delta .. \log n + \delta]$.*

We will identify the nodes of B by their labels. Further, we will store the nodes of B in an array indexed by their labels and we note that this will only take up $O(n)$ space. We let B^t denote the complete subtree of B consisting of the highest $\log n - \delta$ levels. Observe that for each leaf in $v \in B^t$ we have $|\text{sub}_B(v)| = O(1)$.

We now return to the proof of lemma 4.7. For the rest of the paper we assume n is a fixed value which is a power of two and that K contains between n and $n/2$ elements. This assumption can be removed by the use of global rebuilding [87].

We keep the elements of K in the binary search tree B of lemma 4.8. When an element is removed from K we mark the corresponding element of B as dead. In each node $v \in B^t$ we keep a number $\text{aweight}(v)$ which is the sum of the weights of the live nodes in $\text{sub}_B(v)$. We have:

Lemma 4.9. *Assume a node $v \in B^t$ has depth d. Then* $\text{aweight}(v) = \Theta(n/2^d)$.

In each leaf of B^t we keep a q-heap where we insert the ancestors of v using their keys in B^t as keys. To maintain these q-heaps we need to increase the update time to amortized $O(\log^3 n)$ and the space usage to $O(n \log n)$. Suppose we are given a query weight(ϵ, u'_0, j) and assume w.l.o.g. that $u'_0 \in K$ is a leaf in B^t and $j \notin K$. Let s be as defined by (4.1) with $i = u'_0$. The following algorithm defines a sequence of nodes $u_0, \dots, u_p \in B^t$:

1. Set $p = 0$.
2. Repeat:
   - 2.1. Make a predecessor query for j in the q-heap of u'_p identifying an ancestor w of u'_p
   - 2.2. If w is a leaf in B^t:
     - 2.2.1. Set $u_p = w$
     - 2.2.2. Stop
   - 2.3. Set $u_p = \text{left}(w)$
   - 2.4. Set u'_{p+1} to the successor of w in B^t
   - 2.5. Set $p = p + 1$

In item 2.4 it is possible to calculate the (label of) the successor of u'_{p+1} in constant time. Further, since a successor to an internal node in a complete binary search tree is a leaf, the nodes $u'_0, \dots, u'_p$ are leaves in B^t and there is a q-heap to make a query in in item 2.1. We define the node sequence $v_1, \dots, v_q \in B^t$ to consists of the nodes right(v) for which $v \in [\text{parent}(u'_0) \rightsquigarrow u_0]$ and right$(v) \notin [u'_0 \rightsquigarrow u_0]$. Further, we order the nodes such that v_i has smaller depth than v_{i+1} for $1 \leq i < q$.

Lemma 4.10. *We have:*

1. *The nodes $\{u_i \,|\, 1 \leq i \leq p\} \cup \{v_i \,|\, 0 \leq i \leq q\}$ span exactly the nodes in B^t strictly between u'_0 and j. Further no node in this set is an ancestor to another node in the set.*
2. *The nodes in the sequence $u_0, \dots, u_p$ and the nodes in the sequence $v_0, \dots, v_q$ have strictly increasing depth in B^t.*
3. *Both the sequence $u_0, u_1, \dots, u_p$ and the sequence $v_0, v_1, \dots, v_q$ can be produced incrementally in constant time per node.*

We now describe how to answer queries. We take the nodes in $\{u_i \,|\, 1 \leq i \leq p\} \cup \{v_i \,|\, 1 \leq i \leq q\}$ in non-decreasing depth. For each taken node v we increment an initially one counter s' with aweight(v). When we are through all nodes we use constant time to count the remaining elements in B near j. Item 1 in Lemma 4.10 ensures that s' will end up with the value of s. Item 3 in the lemma ensures that each step in the algorithm takes constant time. Item 2 in the Lemma together with Lemma 4.9 ensures that the quantities we add are geometrically decreasing except for constant factors. So after $O(k)$ iterations there is at most a $1/2^k$ fraction of the live nodes between v and j which are not counted. Finally we note, that if $\epsilon < 1/s$ the tolerated additive error is less than one so in this case we indeed find the predecessor of j.

Appendix

In this appendix we prove lemma 4.6 as a corollary to lemma 4.7. We divide U into disjoint intervals (or blocks) B such that $U = \cup_{b \in B} b$. We define $l = 2 \log^{k+1} n$ and for $b \in B$ we require $l \leq |b \cap K| \leq 3l$. For $b \in B$ we let right$(b) \in U$ denote that largest element in b (this does not need to be in K). Further, for $v \in U$ we let $b(v) \in B$ denote the block such that $v \in b(v)$. We keep the elements $\{\text{right}(b) \,|\, b \in B\}$ in a VEB and in a structure T of Lemma 4.7. For right$(b) \in T$ we maintain weight(right(b)) such that $1 \leq (|b \cap K| - \log n)/l \leq \text{weight}(\text{right}(b)) \leq |b \cap K|/l$

implying $l \cdot \text{weight}(\text{right}(b)) \leq |b \cap K| \leq l \cdot \text{weight}(\text{right}(b)) + \log n$. Since $\text{weight}(\text{right}(b)) \geq 1$ this implies:

$$l \cdot \text{weight}(\text{right}(b)) \leq |b \cap K| \leq (l + \log n)\text{weight}(\text{right}(b)) \tag{4.2}$$

For $b \in B$ we keep a balanced search tree with the elements of $b \cap K$. We maintain the tree such that it has degree $O(\log^{\gamma} n)$ for a constant $0 < \gamma < 1$ and constant height. In each internal node v of the tree we keep the keys in v in a q-heap and we also in a single word keep the number of nodes in each of the children of v. Using table lookup it is possible to perform operations in the nodes of v in constant time. More details will be given in the full version of the paper.

Consider an insertion of an element $k \in U$. We first in time $O(\log w)$ locate the block $b \in B$ to which k belongs using the maintained VEB. We then insert the element in the search tree at b using constant time. In case we break the invariant for the weight of $\text{right}(b) \in T$ we select $\text{weight}(\text{right}(b)) = \max(1, (|i \cap K| - (\log n)/2)/l)$. If $|b \cap K|$ becomes too big we split b into two blocks with equally many elements from K. Deletions of elements are performed in a way similar to insertions except that if $|b \cap K|$ become too small b is merged with one of its neighbor blocks and the new block is then possibly split. It can be shown that the amortized cost of updates can be kept down on $O(\log w)$ w.h.p..

Assume now we are given a query $\text{count}(\epsilon, u, v)$ where $u \in K$ (the case $v \in K$ is handled similarly) and $1/\log^k n \leq \epsilon$ or equivalently $(\log n)/l \leq \epsilon/2$. Define $a = |K \cap [u..v]|$. An answer to the query is then an a' such that $(1-\epsilon)a' \leq a \leq a'$. Assume $\text{right}(b(u)) < v$ since otherwise we can find $a' = a$ in constant time by just looking in $b(u)$. Define $B_{uv} = \{b \,|\, \text{right}(b(u)) + 1 \leq \text{right}(b) \leq v - 1\}$ as the blocks strictly between $b(u)$ and $b(v)$. Further, define $s = \sum_{b \in B_{uv}} \text{weight}(\text{right}(b))$. We perform a $\text{aweight}(\epsilon/15, \text{right}(b(u)) + 1, v - 1)$ query in T and let s' the answer to the query. By Lemma 4.7 we have $(1 - \epsilon/15)s \leq s' \leq s$ implying $1/s' \leq 1/(s(1 - \epsilon/15))$.

Suppose first that the predecessor of v is found by the lemma. Then $b(v)$ is also found and we can in constant time find m as the cardinality of the set $\{t \in (b(u) \cup b(v)) \cap K \,|\, u \leq t \leq v\}$. We have $a = m + \sum_{b \in B_{uv}} |b \cap K|$. Equation (4.2) together with the definition of s then implies $m + ls \leq a \leq m + (l + \log n)s$ and since $s' \leq s$ this implies $m + ls' \leq a \leq m + (l + \log n)s$. We will use $a' = m + ls'$. It follows directly that $a' \leq a$. To show $(1-\epsilon)a \leq a'$ it is sufficient to show $(m + (l + \log n)s)/(m + ls') \leq 1/(1-\epsilon)$. To show this it is sufficient to show $(l + \log n)s/(ls') \leq 1/(1-\epsilon)$. Using $1/s' \leq 1/(s(1-\epsilon/15))$ it is sufficient to show $(l + \log n)s/(ls) \leq (1-\epsilon/15)/(1-\epsilon)$. Using $(\log n)/l \leq \epsilon/2$ it is sufficient to show $1 + \epsilon/2 \leq (1-\epsilon/15)/(1-\epsilon)$ and standard calculations show that this is true for all $0 < \epsilon < 1$.

Next suppose the predecessor of v is not found implying $\epsilon/15 \geq 1/s$. Since $6l$ is an upper bound on the number of elements from K in two blocks we know $\sum_{b \in B_{uv}} |b \cap K| \leq a \leq 6l + \sum_{b \in B_{uv}} |b \cap K|$. Equation (4.2) together with the definition of s then implies $ls \leq a \leq 6l + (l + \log n)s$ and since $s' \leq s$ this implies $ls' \leq a \leq 6l + (l + \log n)s$. We will use $a' = ls'$. It follows directly that $a' \leq a$. To show $(1-\epsilon)a \leq a'$ it is sufficient to show $(6l + (l + \log n)s)/(ls') \leq 1/(1-\epsilon)$. Using $1/s' \leq 1/(s(1-\epsilon/15))$ it is sufficient to show $(6l + (l + \log n)s)/(ls) \leq (1-\epsilon/15)/(1-\epsilon)$ or equivalently $6/s + 1 + (\log n)/l \leq (1-\epsilon/15)/(1-\epsilon)$. Using $1/s \leq \epsilon/15$ and $(\log n)/l \leq \epsilon/2$ it is sufficient to show $6\epsilon/15 + 1 + \epsilon/2 \leq (1-\epsilon/15)/(1-\epsilon)$ and standard calculations show that this is true for alle $0 < \epsilon < 1$.

Chapter 5

Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting

Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting

Joseph JaJa [*] Christian W. Mortensen [†] Qingmin Shi[*]

Abstract

We present linear-space sub-logarithmic algorithms for handling the *3-dimensional dominance reporting* and the *2-dimensional dominance counting* problems. Under the RAM model as described in [M. L. Fredman and D. E. Willard. "Surpassing the information theoretic bound with fusion trees", *Journal of Computer and System Sciences*, 47:424–436, 1993], our algorithms achieve $O(\log n/\log\log n + f)$ query time for the 3-dimensional dominance reporting problem, where f is the output size, and $O(\log n/\log\log n)$ query time for the 2-dimensional dominance counting problem. We extend these results to any constant dimension $d \geq 3$, achieving $O(n(\log n/\log\log n)^{d-3})$ space and $O((\log n/\log\log n)^{d-2} + f)$ query time for the reporting case and $O(n(\log n/\log\log n)^{d-2})$ space and $O((\log n/\log\log n)^{d-1})$ query time for the counting case.

5.1 Introduction

The d-dimensional dominance reporting (resp. counting) problem for a set S of d-dimensional points is to store S in a data structure such that given a query point q the points in S that dominate q can be reported (resp. counted) quickly. A point $p = (p_1, p_2, \ldots, p_d)$ dominates a point $q = (q_1, q_2, \ldots, q_d)$ if $p_i \geq q_i$ for all $i = 1, \ldots, d$. A number of geometric retrieval problems involving iso-oriented objects can be reduced to these problems (see for example [42]). For the rest of the introduction we let n denote the number of points in S, f denote the number of points reported by a dominance reporting query, and $\epsilon > 0$ be an arbitrary small constant. The results of this paper can be summarized by the following two theorems.

Theorem 5.1. *For any constant $d \geq 3$ there exists a data structure for the d-dimensional dominance reporting problem using $O(n(\log n/\log\log n)^{d-3})$ space such that queries can be answered in $O((\log n/\log\log n)^{d-2} + f)$ time.*

Theorem 5.2. *For any constant $d \geq 2$ there exists a data structure for the d-dimensional dominance counting problem using $O(n(\log n/\log\log n)^{d-2})$ space such that queries can be answered in $O((\log n/\log\log n)^{d-1})$ time.*

[*]Institute of Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA. Email: {joseph,qshi}@umiacs.umd.edu

[†]IT University of Copenhagen, Rued Langgaardsvej 7, 2300 København S, Denmark. Email: cworm@itu.dk. Part of this work was done while the author was visiting the Max-Planck-Institut für Informatik, Saarbrücken, as a Marie Curie doctoral fellow.

Note that a d-dimensional range counting query (in which each coordinate of a point to be reported is bounded from two sides instead of one as in dominance counting) can be handled by combining the results of a constant number (depending on d) of dominance counting queries of the same dimension. Hence the results in Theorem 5.2 are valid for range counting queries as well.

In this paper, we assume the RAM model as described by Fredman and Willard in [51], which allows the construction of q-heaps [52] (to be discussed in Section 5.2). In both Theorems 5.1 and 5.2 we assume coordinates of points are integer valued. But for $d \geq 4$ in Theorem 5.1 and $d \geq 3$ in Theorem 5.2 our results actually hold for real-valued coordinates.

Our success in proving the above theorems is based on a novel generalization of the concept of dimensions. This definition, given in Section 5.2, allows a k-dimensional point ($k \leq d$) in the standard sense to be appended with $d-k$ "special" coordinates, each of which can take only $\lceil \log^{\epsilon} n \rceil$ different values. Our approach is based on the fact that we can prove more general results for k-dimensional reporting (see Lemma 5.3) and counting queries (see Lemma 5.6) than what has been done before. That is, in addition to the constraints on the first k coordinates specified by a standard dominance query, our solutions satisfy the additional constraints on the $d-k$ "special" coordinates. These results will in turn lead to efficient extensions to higher dimensions.

5.1.1 Relation to Prior Work

In [29], Chazelle and Edelsbrunner proposed two linear-space algorithms for the 3-dimensional dominance reporting problem. The first achieves $O(\log n + f \log n)$ query time and the second achieves $O(\log^2 n + f)$ query time. These two algorithms were later improved by Makris and Tsakalidis [71] to yield $O((\log\log U)^2 \log\log\log U + f \log\log U)$ query time for a special case where coordinates of the points are integers from a bounded universe $[0, \ldots, U]$ and $O(\log n + f)$ query time for the general case. The previous best linear-space algorithm for the 2-dimensional dominance counting problem is due to Chazelle [26] and achieves $O(\log n)$ query time. An external version of Chazelle's scheme was proposed by Govindarajan, Arge, and Agarwal [56], which achieves linear space and $O(\log_B n)$ I/Os (B being the disk block size) for handling range counting queries in $\mathcal{R}^2$. They also extended their algorithm to handle higher dimensional range counting queries, introducing a factor of $O(\log_B n)$ to both the space and the query complexity for each additional dimension.

In [98], Shi and JaJa achieved $O(\log n / \log\log n + f)$ query time for the reporting case and $O(\log n / \log\log n)$ query time for the counting case, but at the expense of increasing the space cost in both cases by a factor of $\log^{\epsilon} n$. Like our solution, these solutions require coordinates of points to be integers.

It follows that, compared with previous solutions that require linear space and, for the reporting case, constant time per reported point, our results improve the query time by a factor of $\log\log n$. Further, the standard techniques for extending the structures into higher dimensions either require a $\log n$ factor on both the query time and the space usage for each dimension [17] or a $\log^{1+\epsilon} n$ factor on the space usage and a $\log n / \log\log n$ factor on the query time for each dimension [6] (while still only using constant time for each reported point). In this paper, we improve the cost of extensions to higher dimensions to a factor of $O(\log n / \log\log n)$ per dimension for both query time and space usage.

5.1.2 Paper Outline

We start with preliminaries in Section 5.2. In Section 5.3 we give a solution to our generalized 3-dimensional dominance reporting problem, thus proving Theorem 5.1 for $d = 3$. In Section 5.4

we give a solution to our generalized 2-dimensional dominance counting problem and prove Theorem 5.2 for $d = 2$. Finally, we extend our results to higher dimensions in Section 5.5.

5.2 Preliminaries

If an array A is indexed by a set M we will use the notation $A[m]$ for the element of A at index $m \in M$. For integers i and j we let $[i..j]$ denote the set of integers k for which $i \leq k \leq j$. If M is a set and $k \geq 0$ is an integer we let M^k denote the set $M \times \cdots \times M$ where M is repeated k times. When stating theorems, we define $i/0 = \infty$ when $i > 0$.

For the rest of this paper, we assume n is the number of points for the dominance reporting or counting structure we are ultimately designing. While developing the ultimate structures, we will construct structures with fewer than n points. We will assume we can use time $O(n)$ to precompute a constant number of tables with size $O(n)$ which only depend on the word size and on n.

We say a point $p = (p_1, \dots, p_d)$ has *dimension* (d', d, ϵ) if $p_{d'+1}, \dots p_d \in [1..\lceil \log^\epsilon n \rceil]$. Here we assume $1 \leq d' \leq d$ and $0 < \epsilon < 1$ are all constants. We refer to p_i, $1 \leq i \leq d$, as the i-coordinate of p. We say a set S has dimension (d', d, ϵ) if all the elements of S are (d', d, ϵ)-dimensional points. When creating a data structure containing a set S with dimension (d', d, ϵ), we assume without loss of generality that no two different points in S share any of their first d' coordinates. That is, if $p = (p_1, \dots, p_d) \in S$ and $r = (r_1, \dots, r_d) \in S$ and $p \neq r$ then $p_1 \neq r_1, \dots, p_{d'} \neq r_{d'}$. We define the *$i$-successor of an integer s* as the point $p = (p_1, \dots, p_d) \in S$ such that $p_i \geq s$ and p_i is minimized. We define the *i-rank of an integer s* as the number of points $(p_1, \dots, p_d) \in S$ for which $p_i \leq s$, and the *i-rank of a point* in S as the i-rank of its i-coordinate. We say the i-coordinate is in *rank space* if for all points $p = (p_1, \dots, p_d) \in S$ the i-rank of p_i is equal to p_i.

We will use the q-heap data structure of Fredman and Willard [52]. A q-heap allows a set S of words where S has cardinality $O(\log^{1/4} n)$ to be stored such that elements can be inserted in and deleted from S in constant time and such that given a query integer q the largest element of S smaller than q can be identified in constant time. The q-heap requires a precomputed table with size $O(n)$ computable in $O(n)$ time. The size of this table only depends on the word size and on n.

5.3 Three-dimensional Dominance Reporting

The purpose of this section is to design a data structure to solve the dominance reporting problem for a set with dimension $(3, d, \epsilon)$ for $d \geq 3$, in particular proving Theorem 5.1 for $d = 3$. We give this structure in Section 5.3.2. First, we give, in Section 5.3.1, a solution to a variant of the dominance reporting problem for a set with dimension $(2, d, \epsilon)$, where $d \geq 2$.

5.3.1 $(2, d, \epsilon)$-dimensional 3-sided Reporting

We define the *3-sided reporting* problem for a set S with dimension $(2, d, \epsilon)$ as a generalization of the dominance reporting problem as follows. A query must have the form $((q_1, q_1'), q_2, \dots, q_d)$ where $q_1 \leq q_1'$. The answer to such a query is the set of points $p = (p_1, \dots, p_d) \in S$ for which $q_1 \leq p_1 \leq q_1'$ and $p_2 \geq q_2, \dots, p_q \geq q_d$. We have:

Lemma 5.3. *Let $d \geq 2$ and $0 < \epsilon < 1/(d-2)$ be constants and let S be a $(2, d, \epsilon)$-dimensional point set of size $m \leq n$, where the 1-coordinate is in rank space. Then there exists a solution to the 3-sided reporting problem for S using $O(m)$ space such that queries reporting f points can be answered in $O(1 + f)$ time.*

For $d = 2$ this problem is well studied (see the survey by Alstrup et al. [7]). We show the lemma for any $d \geq 2$ by using an extension of a technique mentioned in [7]. Actually, we show Lemma 5.3 as a corollary to Lemma 5.4:

Lemma 5.4. *Under the assumptions of Lemma 5.3, we can create a structure using $O(m)$ space such that, given a query $((q_1, q_1'), q_3, \ldots q_d)$, we can, in constant time, either find the point $(p_1, \ldots, p_d) \in S$ with $q_1 \leq p_1 \leq q_1'$ and $p_3 \geq q_3, \ldots, p_d \geq q_d$ such that p_2 is maximized, or determine that no such point exists.*

We obtain Lemma 5.3 from Lemma 5.4 as follows. Suppose we are given a query $((q_1, q_1'), q_2, \ldots q_d)$ in Lemma 5.3. We then ask the query $((q_1, q_1'), q_3, \ldots q_d)$ in Lemma 5.4. If we do not get a point we stop, else let $p = (p_1, \ldots, p_d) \in S$ be the point we get. If $p_2 < q_2$ we stop. Otherwise, we report p and recursively ask the two queries $((q_1, p_1 - 1), q_3, \ldots, q_d)$ and $((p_1 + 1, q_1'), q_3, \ldots, q_d)$ in Lemma 5.4.

The rest of this section is devoted to the proof of Lemma 5.4. We assume m is a power of two and define $c = \lceil \log^\epsilon n \rceil$. We sort the points in increasing order by their 1-coordinates and group them into blocks each with $c^{d-2} \log m$ elements. We define for each block b a *1-range* $[b.l..b.r]$, where $b.l$ (resp. $b.r$) is the smallest (resp. largest) 1-coordinate of a point in b. We create a binary tree T built on the blocks b sorted in increasing order by $b.l$. We associate with each leaf v of T two arrays $v.\text{left}$ and $v.\text{right}$, both indexed by $[0..\log m] \times [1..c]^{d-2}$. Let u be the ancestor of v whose height is h (the height of a leaf is 0). Then $v.\text{left}[h, q_3, \ldots q_d]$ (resp. $v.\text{right}[h, q_3, \ldots q_d]$) stores the point $p = (p_1, \ldots, p_d)$ such that (i) p belongs to a block corresponding to a leaf between v and the leftmost (resp. rightmost) leaf node of the subtree rooted at u; (ii) $p_3 \geq q_3, \ldots p_d \geq q_d$; and (iii) p_2 is maximized, provided that such a point p exists. We note that, since each of the $O(m/(c^{d-2} \log m))$ leaves of T contains two arrays both with $O(c^{d-2} \log m)$ elements, the total space used so far is $O(m)$.

The data structure we just described has enabled us to answer a query $((q_1, q_1'), q_3, \ldots q_d)$ where $q_1 = b.l$ and $q_1' = b'.r$ for two blocks $b \neq b'$. We first locate, in constant time, the nearest common ancestor u of the two leaves corresponding to b and b'. This can be done in constant time using table lookup. Let h be the height of u. Then the point that satisfies the query (if any) must be in either $b.\text{right}[h-1, q_3, \ldots, q_d]$ or $b'.\text{left}[h-1, q_3, \ldots, q_d]$.

In order to be able to answer general queries where the range $[q_1, q_1']$ may partially intersect the 1-ranges of some blocks, we create an extra structure for each block b as follows. We build a constant-height tree $b.T$ with degree $\Theta(\log^\delta n)$, for a constant $\delta > 0$ to be determined later, over the points of b sorted in increasing order by their 1-coordinates. At each internal node $v \in b.T$ we keep for each pair (v_1, v_2) of its children, where v_1 is to the left of v_2, an array $v.R_{(v_1,v_2)}$ indexed by $[1..c]^{d-2}$. The entry $(q_3, \ldots, q_d)$ of this array identifies the point $p = (p_1, \ldots, p_d)$ in b such that (i) p is in a leaf below a child of v between v_1 and v_2; (ii) $p_3 \geq q_3, \ldots, p_d \geq q_d$; and (iii) p_2 is maximized, provided that such a point exists. Since within b a point can be uniquely identified using $O(\log \log n)$ bits, the total bit-cost of each internal node of $v.T$ is $O(c^{d-2} \log \log n \log^{2\delta} n)$. It follows that an internal node fits into a single word (and thus the total space usage will be linear), if $2\delta + (d-2)\epsilon < 1$, which can be satisfied by choosing $\delta = (1 - (d-2)\epsilon)/3$. Our assumption $\epsilon < 1/(d-2)$ ensures that δ will be positive.

We now describe how to answer a query of the form $((q_1, q_1'), q_3, \ldots, q_d)$ where $q_1, q_1' \in [b.l..b.r]$ for some block b. Let u be the nearest common ancestor of the leaves in $b.T$ corresponding to q_1 and q_1', and let Π_1 and Π_2 be respectively the paths from u to these two leaves. The answer to the query can be chosen from a constant number of "candidate points", each picked at a node on Π_1 or Π_2 from one of its associated arrays. For example, without loss of generality, consider the node u. Suppose its ith and jth children are respectively on paths Π_1 and Π_2, and assume $i < j - 1$. Then the candidate point contributed by u can be found in constant time at $u.R_{(v_1,v_2)}[q_3,\ldots,q_d]$, where v_1 and v_2 are respectively the $(i+1)$th and $(j-1)$th children of u.

Finally suppose we are given a query $((q_1,q_1'),q_3,\ldots,q_d)$ where $q_1 \in [b.l..b.r]$ and $q_1' \in [b'.l..b'.r]$ for two different blocks b and b', which can be easily identified in constant time. The output of the query is then one of the three points returned by the queries $((q_1,b.r),q_2,\ldots,q_d)$, $((b'.l,q_1'),q_2,\ldots,q_d)$, and $(b.r+1,b'.l-1,q_2,\ldots,q_d)$, the handling of which has already been described.

5.3.2 $(3,d,\epsilon)$-dimensional Dominance Reporting

In this section we show the following lemma:

Lemma 5.5. *Let $d \geq 3$ and $0 < \epsilon < \min(1/4, 1/(d-2))$ be constants and assume S is a $(3,d,\epsilon)$-dimensional point set of size $m \leq n$. Then there exists a solution to the dominance reporting problem for S using $O(m)$ space such that a query reporting f points can be answered in $O(\log m/\log\log n + f)$ time.*

We say a point $p=(p_1,\ldots,p_d)$ 2-*dominates* a point $q=(q_1,\ldots,q_d)$ if $p_i \geq q_i$, for $1 \leq i \leq 2$. We say a subset M of a point set P is 2-*maximal* if $p \in M$, $q \in P$ and $p \neq q$ implies that p does not 2-dominate q.

We build a search tree T with degree $c = \lceil \log^\epsilon n \rceil$ and height $O(\log m/\log\log n)$ over the points from S sorted in increasing order by their 3-coordinates. At each internal node $v \in T$ we store the keys to guide the search in T in a q-heap. For each internal or leaf node $v \in T$ we define $M(v)$ to be the largest 2-maximal set of the points stored in the leaves of the subtree rooted at v, excluding those in $M(v')$ for any ancestor v' of v. We keep each point $(p_1,\ldots,p_d) \in M(v)$ as the $(2,d-1,\epsilon)$-dimensional point $(p_1,p_3,p_4,\ldots,p_d)$ in the 3-sided reporting structure $D(v)$ of Lemma 5.3. For the moment we ignore the requirement in Lemma 5.3 that the 1-coordinate must be in rank space. At each internal node v, we also store another data structure $G(v)$ containing the points in $M(v')$ for all the children v' of v. A point $(p_1,\ldots,p_d)$ from the *ith* child of v (counted from the left) is stored in $G(v)$ as the $(2,d,\epsilon)$-dimensional point $(p_1,p_2,i,p_4,\ldots,p_d)$. $G(v)$ is also a 3-sided reporting structure of Lemma 5.3, again ignoring the rank space requirement.

Now suppose we are given a query $q=(q_1,\ldots,q_d)$. We first identify the path Π in T from the root to the leaf corresponding to the 3-successor of q_3. We can find Π in time $O(\log m/\log\log n)$ by using the q-heaps associated with the nodes of T. We then visit the root of T as described in the following. Actually, we will describe how to visit an arbitrary node $v \in T$. We first report the points from $M(v)$ which dominate q by performing a query $q' = ((q_1,q_1'),q_3,q_4,\ldots,q_d)$ in $D(v)$, where q_1' is the 1-coordinate of the 2-successor of q_2 in $M(v)$ (we will explain later how to find this 2-successor in constant time). The points returned by this query are exactly the points in $M(v)$ that dominate q. This is due to the fact that for any two points $r=(r_1,r_2,\ldots,r_d)$ and $s=(s_1,s_2,\ldots,s_d)$ in $M(v)$, $r_1 > s_1$ if and only if $r_2 < s_2$ (see [71] for more details). Next, suppose the kth child of v is on Π (we set $k=0$ if v is not on Π). We then perform the query $q'' = ((q_1,\infty),q_2,k+1,q_3,\ldots,q_d)$ in $G(v)$. If the answer to q'' contains a point $(p_1,p_2,i,p_4,\ldots,p_d)$, we recursively visit the ith child of v. If v has a child on Π we also recursively visit that child (such child exists if and only if v is an internal node on Π).

We now address the issue that the 1-coordinate of the points in $M(v)$ and $G(v)$ for a node v in T has to be in rank space in order for Lemma 5.3 to be applicable, and that the 2-successor of q_2 in $M(v)$ has to be identified in constant time. The first issue is resolved by replacing the 1-coordinate of each point in $M(v)$ (resp. $G(v)$) with its 1-rank with respect to $M(v)$ (resp. $G(v)$). Accordingly, before the query q' (resp. q'') is applied to $M(v)$ (resp. $G(v)$) we replace

q_1 with its 1-rank in $M(v)$ (resp. $G(v)$) (and, in the case of query q', replace q'_1 with the 1-rank of the 2-successor of q_2). Computing the 1-rank and the 2-successor of an integer in $M(v)$ and $G(v)$ for each node visited is exactly the *iterative search* problem defined in [30], which can be handled in $O(1)$ time if v is not the root of T and in $O(\log m/\log\log n)$ time if v is the root, using the fast fractional cascading technique of [99], which requires $O(m)$ space. (Notice that the standard fractional cascading technique described in [30] will not work here because the degree of T is not a constant.)

Note that a point in S is stored at most twice, once in $D(v)$ for a node v and once in $G(u)$, where u is the parent of v. The data structures of types D and G are all linear-space data structures. Therefore the overall space usage is $O(m)$. Further, it is easy to see that the number of nodes of T visited is $O(\log m/\log\log n + f)$ and that searching $D(v)$ and $G(v)$ at each such node v takes $O(1)$ time per reported point, hence the claimed querying complexity.

5.4 Two-dimensional Dominance Counting

The goal of this section is to design a data structure to solve the dominance counting problem for a set with dimension $(2, d, \epsilon)$ for $d \geq 2$, thus proving Theorem 5.2 for $d = 2$. We give this structure in Section 5.4.2. But first, we give, in Section 5.4.1, a solution with sub-linear space usage for a set with dimension $(1, d, \epsilon)$, for $d \geq 1$, where the 1-coordinate is in rank space.

5.4.1 $(1, d, \epsilon)$-dimensional Dominance Counting

This section is devoted to the proof of the following lemma:

Lemma 5.6. *Let $d \geq 1$ and $0 < \epsilon < 1/(d-1)$ be constants and assume S is an $(1, d, \epsilon)$-dimensional point set of size $m \leq n$, where the 1-coordinate is in rank space. Then there exists a solution to the dominance counting problem for S using $O(m \log\log n)$ bits of space such that queries can be answered in constant time.*

We assume m is a power of two and define $c = \lceil \log^{\epsilon} n \rceil$. We sort the points in increasing order by their 1-coordinates and group them into blocks each with $c^{d-1} \log m$ elements. Further, we partition each block into subblocks each with c^{d-1} elements. We label each block (resp. subblock) with the largest 1-coordinate of a point in that block (resp. subblock). For each block or subblock b we keep an array b.count indexed by $[1..c]^{d-1}$. For each block b we set $b.\text{count}[q_2, \ldots, q_d]$ to be the number of points in S dominating $(i+1, q_2, \ldots, q_d)$ where i is the label of b. For each subblock b' of a block b we set $b'.\text{count}[q_2, \ldots, q_d]$ to be the number of points in b dominating $(i+1, q_2, \ldots, q_d)$ where i is the label of b'. Finally, we encode the points of a subblock b' in $O(c^{d-1} \log c) = o(\log n)$ bits which we keep in a single word. This is possible since each of the c^{d-1} points in b' can be encoded in $O(\log c)$ bits.

Suppose we are given a query $q = (q_1, \ldots, q_d)$. We first identify, in constant time, the block (resp. subblock) b (resp. b') with the smallest label greater than or equal to q_1. We now describe how to find the number e of points in b' that dominate q. Notice that a query q with respect to a subblock can be described in $O(\log\log n)$ bits. To compute e in constant time, all we need is to append q to the description of b' and use the result to look up a global table, which requires $O(n)$ words since $O(c^{d-1} \log c \log\log n) = o(\log n)$. The answer to the query q is then $e + b.\text{count}(q_2, \ldots, q_d) + b'.\text{count}(q_2, \ldots, q_d)$.

We now analyze the space usage of the structure. Each array of the $m/(c^{d-1}$ $\log m)$ blocks contains c^{d-1} elements each of $\log m$ bits. It follows that the space used by these arrays is $O(m)$ bits. Each array of the m/c^{d-1} subblocks also contains c^{d-1} elements but each element is at most $c^{d-1} \log m$. It follows that each element can be represented by $O(\log\log n)$

bits so the total space used by the arrays associated with the subblocks is $O(m \log\log n)$ bits. Finally, each point of a subblock can be represented by $O(\log\log n)$ bits and it follows that the total space usage becomes $O(m \log\log n)$ as claimed.

5.4.2 $(2, d, \epsilon)$-dimensional Dominance Counting

In this section we show:

Lemma 5.7. *Let $d \geq 2$ and $0 < \epsilon < \min(1/4, 1/(d-1))$ be constants and let S be a $(2, d, \epsilon)$-dimensional point set of size $m \leq n$. Then there exists a solution to the dominance counting problem for S using $O(m)$ space such that queries can be answered in $O(\log m/\log\log n)$ time.*

We will prove the lemma under the assumption that the first coordinate is in rank space. Since the targeted query time is $O(\log m/\log\log n)$ time, we can easily remove this assumption by transforming the 1-coordinates of the points in S as well as the 1-coordinate of the query point into rank space by creating a search tree with degree $(\log^{\delta} n)$ on the 1-coordinates of the points, where $\delta < 1/4$, and storing the keys at each node of this tree in a q-heap.

We create a search tree T with degree $c = \lceil \log^{\epsilon} n \rceil$ and height $O(\log m/\log\log n)$ over the points from S sorted in increasing order by their 2-coordinates. At each internal node $v \in T$ we store the keys to guide the search in v in a q-heap. We define $G(v)$ to be the set of points stored in the subtree rooted at v. Let $p = (p_1, \ldots, p_d) \in G(v)$ be a point stored at a leaf descendant of the jth child of v, and let i be the 1-rank of p in $G(v)$. Then we store p as $(i, j, p_3, \ldots, p_d)$ in a structure $v.D$ of Lemma 5.6 with dimension $(1, d, \epsilon)$.

Now assume that we are given a query $q = (q_1, \ldots, q_d)$. We first identify the path Π in T from the root to the leaf storing the 2-successor of q_2. We can find Π in time $O(\log m/\log\log n)$ by using the q-heaps stored at the nodes of T. The answer to q is the sum of the answers to $\Theta(\log m/\log\log n)$ $(1, d, \epsilon)$-dimensional dominance counting queries, each applied to the structure $v.D$ for a node v on Π. For the root w, suppose the jth child of w is also on Π. Then the query applied to $w.D$ is $(q_1, j+1, q_3, \ldots, q_d)$. For the leaf that is on Π, we check the point stored there directly. Now consider a non-root internal node v on Π. Let u be its parent. (Note that u is also on Π.) Suppose the jth child of v is also on Π. Note that we already know the 1-rank $r_1(u)$ of q_1 in $G(u)$ ($r_1(w) = q_1$). The query applied to $v.D$ is $(r_1(v), j+1, q_3, \ldots, q_d)$, where $r_1(v)$ is the 1-rank of q_1 in $G(v)$. The value of $r_1(v)$ can be computed by performing a constant number of $(1, d, \epsilon)$-dimensional dominance queries in $u.D$. In fact, let $G_>(v)$ denote the union of $G(v')$ for all the right siblings v' of v, and let $G_\geq(v) = G(v) \cup G_>(v)$. Then $r_1(v)$ is the difference between the 1-rank of q_1 in $G_\geq(v)$ and the 1-rank of q_1 in $G_>(v)$. The 1-rank of q_1 in $G_\geq(v)$ is $|G_\geq(v)| - k_\geq(v) + 1$, where $k_\geq(v)$ is the number of points in $G_\geq(v)$ whose 1-coordinate is greater than or equal to q_1. $|G_\geq(v)|$ can be computed by performing the query $(0, j, q_3, \ldots, q_d)$ in $u.D$ and $k_\geq(v)$ can be computed by performing the query $(r_1(u), j, q_3, \ldots, q_d)$ in $u.D$. The 1-rank of q_1 in $G_>(v)$ can be computed similarly.

Since we only use constant time at each node of Π, the overall query time is $O(\log m/\log\log n)$. Furthermore, we store each of the m points of S in $O(\log m/\log\log n)$ structures, each of which uses $O(\log\log n)$ bits. Therefore the total space usage becomes $O(m)$ as claimed.

5.5 Higher Dimensional Dominance Reporting and Counting

In this section we give a general construction in Lemma 5.8 which can be used to obtain a structure for a $(d, d, 0)$-dimensional point set from a structure for a (d', d, ϵ)-dimensional point set, where $d' < d$. We then use this construction to prove Theorem 5.1 and 5.2 from

Lemma 5.5 and 5.7 respectively. A similar construction was given in [78] for orthogonal range reporting for a dynamic set of points.

Let $(G, +)$ be a semi-group and let S be a set with dimension (d', d, ϵ). Assume each point $p \in S$ has an associated semigroup element $g(p) \in G$. The (d', d, ϵ)-*dimensional dominance semigroup* problem is defined as follows. Given a (d', d, ϵ)-dimensional query point q, find the semigroup sum $\sum_{p \in S : p \text{ dominates } q} g(p)$. We will show:

Lemma 5.8. *Assume $2 \le d' \le d$ and $0 < \epsilon < 1/4$ are constants. Assume we have a data structure structure D' for a $(d'-1, d, \epsilon)$-dimensional dominance semigroup problem of size m' for any $m' \le n$. Then, we can derive a structure D for the (d', d, ϵ)-dimensional dominance semigroup problem of size $m \le n$. Further:*

1. *For every point in D we store $O(\log m / \log\log n)$ points in structures of type D'.*

2. *Given a query in D we can answer it by performing $O(\log m / \log\log n)$ queries in the structures of type D' and return the semigroup sum as the result.*

The space usage besides the space usage in item 1 is $O(n)$ and the queries to be performed in item 2 can be determined in constant time per query.

The dominance reporting problem can be seen as a special case of the dominance semigroup problem if we define the elements in G as point sets, $g(p) = \{p\}$, and select $+$ to be the union operator on sets. Theorem 5.1 then follows by applying Lemma 5.8 $d-3$ times to Lemma 5.5. Similarly, the dominance counting problem can be seen as a special case of the dominance semigroup problem, where the elements in G are non-negative integers, $g(p) = 1$, and $+$ is the integer addition. Theorem 5.2 then follows by applying Lemma 5.8 $d-2$ times to Lemma 5.7.

We now prove Lemma 5.8. Let S be a set of m (d', d, ϵ)-dimensional points. We build a tree T with degree $c = \lceil \log^\epsilon m \rceil$ over the points in S sorted by their d'-coordinates. At each internal node $v \in T$ we keep a dominance semigroup structure $v.D'$ with dimension $(d'-1, d, \epsilon)$ containing the points stored in the subtree rooted at v. A point $p = (p_1, \ldots, p_d)$ from the ith child of v is stored in $v.D'$ as $p' = (p_1, \ldots, p_{d'-1}, i, p_{d'+1}, \ldots, p_d)$ with $g(p') = g(p)$. Suppose now we are given a query $q = (q_1, \ldots, q_d)$ in D with dimension (d', d, ϵ). We first identify the path Π in T from the root to the leaf corresponding to the d'-successor of $q_{d'}$, which can be found in time $O(\log n / \log\log n)$ using the q-heaps in the nodes of T. For each internal node $v \in \Pi$, assume that the j_vth child of v is also on Π. The answer to the query in D is then the semigroup sum of the queries $(q_1, \ldots, q_{d'-1}, j_v + 1, q_{d'+1}, \ldots, q_d)$ in $v.D'$ for every $v \in \Pi$. This finishes the proof of Lemma 5.8.

Chapter 6

New upper bounds for some colored orthogonal intersection reporting problems

New upper bounds for some colored orthogonal intersection reporting problems

Christian Worm Mortensen
IT University of Copenhagen
E-mail: cworm@itu.dk

Abstract

We show how to dynamically maintain an array $A[1 \dots n]$ of colors such such that for given i and j the different colors in $A[i \dots j]$ can be reported in time $O(\log\log n + k)$ where k is the number of colors reported. The space usage is linear. Our solution is based on a new reduction to the range reporting problem on an n times $\log n$ grid. The previously best known solution used time $O(\log n + k)$ and linear space. Combining our new result with partial persistence we obtain new upper bounds for the colored static orthogonal segment intersection reporting problem and for the colored static orthogonal range reporting problem.

6.1 Introduction

Suppose D is a database of geometric objects. Given a query object q (normally not in D) the intersection reporting problem is to return the set of objects in D which intersect q. The intersection reporting problem has been intensively studied in the literature with different restrictions on D and q and in different computational models. See the survey by Agarwal and Erickson [1]. In this paper we study a variant of the intersection reporting problem introduced by Janardan and Lopez [65] which is called the *colored* (or *generalized* or *categorial*) intersection reporting problem. In this variant, each object in D has a color, and we must report the set of colors among the objects in D which are intersected by q. Besides [65] several other papers have considered this variant of the problem [57, 2, 21, 85]. In this paper, we give new upper bounds for the following colored intersection reporting problems.

Colored 1d range reporting. This problem is to maintain an array $A[1 \dots n]^\dagger$ of colors. The array can be updated by assigning a new color to $A[i]$ for an $i \in [1 \dots n]$. Given a query $(i, j) \in [1 \dots n]$ with $i \leq j$ we must report the colors among $A[i], A[i+1], \dots, A[j]$.

Colored segment reporting. We are given a set S of colored and horizontal line segments with endpoints in $[1 \dots n]^2$ which we can preprocess. Given a a query $(x, y_1, y_2) \in [1 \dots n]^3$ with $y_1 \leq y_2$ we must report the k colors represented among the segments from S which intersects the vertical line segment between (x, y_1) and (x, y_2).

Colored range reporting. We are given a set P of colored points in $[1 \dots n]^2$ which we can preprocess. Given a query $(x_1, x_2, y_1, y_2) \in [1 \dots n]^4$ with $x_1 \leq x_2$ and $y_1 \leq y_2$ we must report the k colors represented among the points from $P \cap ([x_1 \dots x_2] \times [y_1 \dots y_2])$.

---

†For integers $i \leq j$ we let $[i \dots j]$ denote the set of integers t for which $i \leq t \leq j$. For a set M we let AM denote an array A indexed by M and for $m \in M$ we let $A[m]$ denote the element of A at entry m.

The full name of the three problems are the colored dynamic 1-dimensional range reporting problem, the colored static orthogonal 2-dimensional segment intersection reporting problem and the colored static orthogonal 2-dimensional range reporting problem respectively. For the rest of this paper we will use the shorter names above. We now mention two applications of the three problems. Several additional applications can be found in [65, 2, 85]. Suppose in the colored range reporting problem, that each point represents a person with an age, a salary and type of employment. A query could then ask for the types of employment for which there is at least one person with an age between 30 and 40 years and a salary between 2000€ and 2500€. As the other application, Muthukrishnan [85] reduced several document retrieval problems to colored intersection reporting problems. One of the document retrieval problems he considered was the *document repeats problem* and our solution to the colored segment reporting problem implies a new solution for this problem with reduced space usage.

6.1.1 Models

The model of computation used in this paper is a unit-cost RAM with word size at least $\log n$ bits[‡]. We assume we have access to a sequence of truly random words.

Above, we have defined the colored segment reporting problem and colored range reporting problem on an n times n grid where n is the number of objects. Previous solutions to these two problems (see below) have either considered the case with coordinates in $\mathbb{R}^2$ (that is, coordinates are only accessible by a unit-cost comparison operation) or on a U times U grid for some $U \geq n$. Using a simple transformation based on binary search, our two structures can be converted into structures for $\mathbb{R}^2$ adding a term of $O(\log n)$ to the query time and a term of $O(n \log n)$ to the preprocessing time. Using another transformation based on the data structure of of van Emde Boas et. al. [105] we can instead get structures for a U times U grid if we add a term of $O(\log \log U)$ to the query time and a term of $O(n \log \log U)$ w.h.p.[§] to the preprocessing time. This can, in fact be improved even further with the use of more recent and complicated predecessor and sorting data structures. The transformations mentioned are standard and omitted.

6.1.2 Our results

Our solution to the colored 1d range reporting problem has update time $O(\log \log n)$ w.h.p., query time $O(\log \log n + k)$, space usage $O(n)$ and initialization time $O(n)$. Here k is the number of reported colors. The best previous solution to this problem was by Gupta, Janardan and Smid [57] and had update time $O(\log n)$, query time $O(\log n + k)$ and space usage $O(n)$. This solution, however, allowed A to be indexed by elements of $\mathbb{R}$.

Our solution to the colored segment reporting problem has query time $O(\log^2 \log n + k)$ and space usage $O(n \log \log n)$ or query time $O(\log n \log^2 \log n + k)$ and space usage $O(n)$. The preprocessing time is $O(n \log \log n)$ w.h.p. (w.h.p. because we use hashing [39]). Building on a preliminary version of this paper [80] Shi and JaJa [100] have improved the second result by reducing the query time to $O(\log n + k)$ while keeping the space usage on $O(n)$ and we will sketch the idea in this improvement. In [65] a solution to the colored segment reporting problem where segments were assumed to have endpoints in $\mathbb{R}^2$ was given. The solution had query time $O(\log n + k)$ and space usage $O(n \log n)$ or query time $O(\log^2 n + k)$ and space usage $O(n)$. Different solutions for n segments on an n times n grid are summarized in Table 6.1. The segment intersection reporting problem where all segments have different colors has been widely

[‡] All logarithms in this paper are base 2.

[§] We use w.h.p. as an abbreviation for with high probability. In a data structure with n elements we let high probability mean probability at least $1 - n^{-c}$ for any constant $c > 0$.

Query time	Space usage	Source
$O(\log n + k)$	$O(n \log n)$	[65]
$O(\log^2 n + k)$	$O(n)$	[65]
$O(\log^2 \log n + k)$	$O(n \log\log n)$	This paper
$O(\log n \log^2 \log n + k)$	$O(n)$	This paper
$O(\log n + k)$	$O(n)$	[100]

Table 6.1: Solutions to the colored segment reporting problem.

Query time	Space usage	Source
$O(\log\log n + k)$	$O(n \log^2 n)$	[2]
$O(\log^2 n + k)$	$O(n \log n)$	[65]
$O(\log^2 \log n + k)$	$O(n \log n \log\log n)$	This paper
$O(\log n \log^2 \log n + k)$	$O(n \log n)$	This paper
$O(\log n + k)$	$O(n \log n)$	[100]

Table 6.2: Solutions to the colored range reporting problem.

studied. Here the problem is to report the set of segments intersecting a given query segment. Chazelle [25] considered the case where segments have endpoints in $\mathbb{R}^2$ and gave an optimal structure with query time $O(\log n + k)$ and space usage $O(n)$. In Section 6.6.1 we remark, that the query time can be reduced to $O(\log^2 \log n + k)$ if endpoints lie on an n times n grid keeping the space usage on $O(n)$. Though we have not been able to find any references on this result it is not likely to be new.

For the colored range reporting problem we obtain a structure with query time $O(\log^2 \log n + k)$ and space usage $O(n \log n \log\log n)$ or query time $O(\log n \log^2 \log n + k)$ and space usage $O(n \log n)$. The preprocessing time is $O(n \log n \log\log n)$ w.h.p.. Building on a preliminary version of this paper [80] Shi and JaJa [100] have improved the second result by reducing the query time to $O(\log n + k)$ keeping the space usage on $O(n \log n)$ and we will also sketch the idea in this improvement.

In [65] a solution to the colored range reporting problem with points in $\mathbb{R}^2$ was given. The solution had query time $O(\log n + k)$ and space usage $O(n \log^2 n)$ or query time $O(\log^2 n + k)$ and space usage $O(n \log n)$. Assuming points lie on a U times U grid Agarwal, Govindarajan and Muthukrishnan [2] improved the first result by reducing the query time to $O(\log\log U)$ keeping the space usage on $O(n \log^2 U)$. Different solutions for n points on an n times n grid are summarized in Table 6.2. The colored range reporting problem where all points have different colors has also been widely studied. Here the problem is to report the set of points in a query rectangle. Chazelle [26] considered the case with points in $\mathbb{R}^2$ and gave a data structure with optimal query time $O(\log n + k)$ and a space usage $O(n \log^{\epsilon} n)$ for any constant $\epsilon > 0$. Alstrup, Brodal and Rauhe [6] considered the problem on an n times n grid and improved the query time to $O(\log\log n + k)$ keeping the same space usage. The colored range reporting problem where all points have the same color has also been studied. Here the problem can be seen as deciding if a given query rectangle is empty. Assuming points are in $\mathbb{R}^2$ Chazelle [26] has given an optimal solution to this problem with query time $O(\log n)$ and linear space usage $O(n)$.

The document repeats problem mentioned in the previous section is to preprocess a collection of text documents with total size n such that given a string q with length m and a number d, we can find the k documents which contain two occurrences of q with distance at most d. In [85] a solution with query time $O(m + k)$, space usage $O(n \log n)$ and preprocessing time $O(n \log n)$

was provided. By using our new structure for the segment reporting problem we can reduce the space usage to $O(n \log^2 \log n)$ while keeping the same query and preprocessing time.

6.1.3 Techniques and outline paper

We start with preliminaries in Section 6.2. This is followed by Section 6.3 which gives our structure for the colored 1d range reporting problem. Our structure is a modified version of the structure for the static variant of the problem considered in [65]. We reduce the colored 1d range reporting problem to the dynamic non-colored range reporting problem on an n times $\log n$ grid. No such reduction was made in [65]. But interestingly, the previous best known solution to the dynamic colored 1d range reporting problem [57] also worked by a reduction to the dynamic non-colored range reporting problem, but on an n times n grid.

In Section 6.4 we use a standard sweepline technique to transform our solution for the colored 1d range reporting problem into a solution for the colored segment reporting problem. We then transform this solution into a solution for the colored range reporting problem. In Section 6.5 we review and reformulate some results on partial persistence. Finally, in Section 6.6, we give several ways to solve the non-colored range reporting problem for an n times $\log n$ grid used by the recduction in Section 6.3. One of these is based on a strucure from [78, 79]. In Section 6.6 we also mention how the improvement of Shi and JaJa [100] is obtained.

6.2 Preliminaries

If T is a rooted tree and $v \in T$ is a node we let $height(v)$ denote the height of v in T (leafs have height 0). We let a VEB denote the data structure of van Emde Boas et. al. [105] combined with hashing [39] and the technique of Willard [107]. Such a structure makes it possible to maintain a set $S \subseteq [1 \dots n]$ using time $O(\log \log n)$ w.h.p. for inserting or deleting an element and time $O(\log \log n)$ for predecessor queries. The space usage is $O(|S|)$.

Suppose we have a data structure supporting update and query operations where each update may perform a number of writes and reads on memory cells and queries may only perform reads. Such a structure is said to be *partial persistent* if each update increments a timestamp and each query takes a timestamp of the version of the data structure in which the query should be performed.

The results of this paper builds on Lemma 6.1 below. We defer the proof of the lemma to Section 6.6. Suppose n is a power of two and that $B[1 \dots n]$ is an array of elements in $[0 \dots \log n]$ which are initially zero. Suppose further that for $i \in [1 \dots n]$ and $y \in [0 \dots \log n]$ we can update B by setting $B[i] = y$. Suppose also, that given a query (i, j, y) where $i, j \in [1 \dots n]$, $i \leq j$ and $y \in [1 \dots \log n]$ we can report the indices $e \in [i \dots j]$ for which $B[e] \geq y$.

Lemma 6.1. *Let k be the number of indices reported by a given query. Then:*

1. *We can maintain B with update time $O(\log \log n)$, query time $O(\log \log n + k)$ and space usage $O(n)$.*

Further let m be the number of updates performed. Then:

2. *We can make a partial persistent version of B with update time amortized $O(\log \log n)$ w.h.p., query time $O(\log^2 \log n + k)$ and space usage $O(m \log \log n)$.*

3. *We can make a partial persistent version of B with update time amortized $O(\log \log n)$ w.h.p., query time $O(\log n \log^2 \log n + k)$ and space usage $O(m)$.*

In all parts 1,2 and 3 a precomputed lookup table with $o(n)$ entries constructable in time $o(n)$ is needed.

6.3 A dynamic one-dimensional structure

In this section we describe how to maintain an array $A[1 \dots n]$ where each element in A has a color which is initially black. For given $i \in [1 \dots n]$ the array can be updated by assigning a color to $A[i]$. Further, for $i, j \in [1 \dots n]$ where $i \leq j$ we support a query (i, j) which returns the set of non-black colors among the elements $A[k]$ for which $i \leq k \leq j$. In this section we show:

Theorem 6.2. *The results for B in Lemma 6.1 also applies to A except that in part 1 the update time becomes w.h.p..*

Part 1 of this theorem gives our structure for the colored 1d range reporting problem.

We assume w.l.o.g. (without loss of generality) that n is a power of two. We span a complete binary tree T over the elements of A from left to right. We will not distinguish between an element of A, its index in A and the corresponding leaf in T. For every node $v \in T$ we let $span(v)$ be the set of leafs descendant to v. We define $left(v)$ ($right(v)$) as the left-most (right-most) element in $span(v)$.

Below we describe how to make a data structure supporting queries of the form $(left(v), i)$ for $v \in T$ and $i \in span(v)$. A data structure supporting queries of the form $(i, right(v))$ can be made in a symmetric way. General queries (i, j) can be answered using these data structures as follows. If $i = j$ the query is easy so assume $i < j$. We identify the nearest common ancestor v of $A[i]$ and $A[j]$ in constant time. Let u be the left and w be the right child of v. The colors to report can be found by the two queries $(i, right(u))$ and $(left(w), j)$ (the same color may be reported by both queries and we filter out duplicates).

What remains to describe is how to answer queries of the form $(left(v), i)$ for $i \in span(v)$. Again, if $left(v) = i$ the query is easy, so assume $left(v) < i$. For each internal node $v \in T$ we define (but do not explicitly store) the set $E(v)$ of non-black elements $e \in span(v)$ for which there is no element in $span(v)$ to the left of e with the same color as e. We observe that if $e \in E(v)$ then $e \in E(w)$ for the child w of v for which $e \in span(w)$ (because $span(w)$ is a subset of $span(v)$). It follows, that for each leaf $e \in T$ we can assign a y-coordinate $y(e)$ such that e is in $E(w)$ for exactly the ancestors w of e which is at distance at most $y(e)$ from e (if e is black we set $y(e) = 0$). The answer to the query is then the elements $e \in [left(v) \dots i]$ for which $y(e) \geq height(v)$. By maintaining an array B such that $B[i] = y(A[i])$ we can get these elements by performing a query in the structure provided by Lemma 6.1. We will now show that each update of A modifies the y-coordinate of at most a constant number of element of A and that these can be found in time $O(\log \log n)$ w.h.p.. It follows that this is sufficient in order to prove Theorem 6.2.

We first describe how to give a black element $e \in A$ a non-black color c. Let $v \in T$ be the nearest ancestor to e such that $span(v)$ contains an element different from e with color c and let e' be the leftmost such element. If v does not exists we set $y(e) = \log n$ and we are done. We note that we can find v (but not necessarily e') in time $O(\log \log n)$ if we for each color maintain a VEB containing the elements with that color. These VEBs are not used when answering queries and thus do not need to be made partial persistent. Further, it is these VEBs that make the update time w.h.p.. Let w be the child of v such that $e \in span(w)$. If w is the right child of v we set $y(e)$ to $height(w)$. If w is the left child of v we locate e' using the VEB we maintain for color c and then we first set $y(e)$ to $y(e')$ and next we set $y(e')$ to $height(w)$. We now argue for correctness by describing how the sets $E(u)$ change for $u \in T$ because of the update. Let $w' \neq w$ be the other child of v. Then $e' \in E(w')$ and further $e' \in E(v)$ before the update. We note that $span(w)$ contains no element with color c different from e and therefore e is inserted in $E(u)$ for the nodes $u \in T$ between e and w where $u \neq e$. If w is the right child of v then e is to the right of e' and thus no more changes occur. If w is the left child of v then e is to the left of e' and therefore e' is replaced by e in $E(u)$ for all ancestors u of v that contains e'.

We next describe how to color an element $e \in A$ with color c black. If e is black there is nothing to do so assume e is not black. Let v, w and e' be as before. If v does not exists or if w is the right child of v we set $y(e) = 0$. If w is the left child of v we locate e' using the VEB for color c and then we first set $y(e')$ to $y(e)$ and next we set $y(e)$ to 0. The argument of correctness is similar to before.

We finally note, that we can give a non-black element $e \in A$ a non-black color c by first coloring e black and then color e with c. As claimed it follows that changing the color of an element $e \in A$ changes the y-coordinate of at most a constant number of elements in A and these can be found in time $O(\log\log n)$ w.h.p..

6.4 Static two-dimensional structures

In this section we provide our structures for the colored segment reporting problem and for the colored range reporting problem by applying standard techniques to Theorem 6.2. We first show how to solve the colored segment reporting problem. Let S be the set of horizontal and colored segments. We assume w.l.o.g. that no segment in S is black and that all segments in S are disjoint. Let Y be the structure from Theorem 6.2 part 2 or 3 and let $X[1 \dots n]$ be an array. We then enumerate the numbers in $[1 \dots n]$ in increasing order. Let x be an enumerated number. For each left endpoint $(x, y) \in [1 \dots n]^2$ of a segment we set $Y[y]$ to the color of the segment. After this we record the current timestamp of Y in $X[x]$. Next, for each right endpoint $(x, y) \in [1 \dots n]^2$ of a segment we set $Y[y]$ to black. The answer to a query $(x, y_1, y_2) \in [1 \dots n]^3$ in S where $y_1 \le y_2$ can then be found by performing the query (y_1, y_2) in Y at timestamp $X[x]$. The time and space bounds from the introduction follows directly from Theorem 6.2.

Next, we consider the colored range reporting problem. Let P be the set of given colored points. W.l.o.g. assume that no two different points in P have the same coordinates. First assume we only need to support *3-sided queries*, that is queries of the restricted form $(1, x, y_1, y_2)$ for $x, y_1, y_2 \in [1 \dots n]$ and $y_1 \le y_2$. Our structure for the colored segment reporting problem can be used for this as follows. We store the point $(x, y) \in P$ as the horizontal segment between (x, y) and (n, y) in S. The answer to the query $(1, x, y_1, y_2)$ in P is then the same as the answer to the query (x, y_1, y_2) in S.

We now describe how to convert the structure supporting 3-sided queries just described to one supporting general queries. We span a complete binary tree T over the x-axis of the grid from left to right. We will not distinguish between a leaf of T and its coordinate on the x-axis. For any node $v \in T$ we let $P_v \subseteq P$ be the points of P which have an x-coordinate descendant to v. For each node $v \in T$ we store two structures supporting 3-sided queries. Given $x, y_1, y_2 \in [1 \dots n]$ where $y_1 \le y_2$ the first structure supports queries of the form $(1, x, y_1, y_2)$ among the points in P_v and the second supports queries of the form (x, n, y_1, y_2) among the points in P_v (the queries in the second structure are indeed 3-sided if we turn things around). Now suppose we are given a general query $(x_1, x_2, y_1, y_2) \in [1 \dots n]^4$ in P with $x_1 \le x_2$ and $y_1 \le y_2$. To answer the query we first locate the nearest common ancestor v of x_1 and x_2 in T. If v is a leaf then $x_1 = x_2$ and the answer to the query can be found by performing a query (x_1, n, y_1, y_2) among the points in P_v. Suppose v is not a leaf and has left child l and right child r. Then the answer to the query can be found by performing a query (x_1, n, y_1, y_2) among the points in P_l and a query $(1, x_2, y_1, y_2)$ among the points in P_r (the same color may be reported by both queries and we filter out duplicates).

As described the structure has a large space usage because each 3-sided structure we store must support queries on an n times n grid even if it contains much less than n points. To overcome this problem we use the technique mentioned in Section 6.1.1 which in case of m points allows us to reduce the grid size to m times m. This adds a term of $O(\log\log n)$ to the query time and a term of $O(n \log\log n)$ w.h.p. to the preprocessing time (coming from the usage of a VEB). But in our case this only changes the query and preprocessing time by constant factors. Since each point of P is stored in $O(\log n)$ structures supporting 3-sided queries it follows that we get an $O(\log n)$ factor on the space usage and preprocessing time as claimed.

6.5 Partial persistence

In this section we introduce a general way to make data structures partial persistent in the form of Lemma 6.3 below. The section can be seen as a reformulation of known results and techniques. We refer the reader to Driscoll et. al. [40], Dietz [34] and Dietz and Raman [36] for more details.

Let D be an arbitrary deterministic data structure. We can model D as a set of nodes $v \in D$ which we can think of as allocated by the `new` operator in C++ or Java. The data in a node $v \in D$ is contained in an array $\mathit{array}(v)[1 \ldots \mathit{size}(v)]$ where $\mathit{size}(v)$ is fixed on allocation of v. Each entry in $\mathit{array}(v)$ is a computer word and may contain a pointer to a node. In the RAM model $\mathit{size}(v)$ can be arbitrary whereas in the pointer machine model (which we do not consider) $\mathit{size}(v)$ must be constant. We assume D supports updates and queries as mentioned in the preliminaries and that queries always start in a fixed node of D. An update may in constant time per operation allocate a new node of any size or perform a read from or a write to an array in a node. A query may in constant time perform a read from an array in a node. We let n be an upper bound on the total number of writes to arrays and the total number of node allocations performed. On allocation of node v, the user must mark v as *big* or *small*. The user is only allowed to mark a node v as small if $\mathit{size}(v) = O(\log^c n)$ for a constant c.

Lemma 6.3. *Assume that for each small node $v \in D$ at most $\mathit{size}(v)$ pointers (the predecessor pointers) point to it. Then we can make D partial persistent using space $O(n)$. The time to update D is only increased by a constant factor but is made amortized and w.h.p.. The time to read an element in $\mathit{array}(v)$ in a query is $O(1)$ if v is small and $O(\log\log n)$ if v is big.*

Proof (sketch). Consider a node $v \in G$. We store in v both $\mathit{array}(v)$ and an additional array $\mathit{parray}(v)[1 \ldots \mathit{size}(v)]$ using hashing [39]. For each value w written to $\mathit{array}(v)[i]$ at time t we store w with key t in a predecessor structure in $\mathit{parray}(v)[i]$. A read of $\mathit{array}(v)[i]$ at a specific time can then be performed by a predecessor query in $\mathit{parray}(v)[i]$.

Assume v is big. If we use a VEB as predecessor structure we get the properties we want except that in connection with updates we have to use time $O(\log\log n)$ w.h.p. to perform a write to an array in a node. To reduce this to $O(1)$ w.h.p. we use a standard trick. We group the writes into blocks with $\log\log n$ elements and from each block we only insert the element with the smallest timestamp in the VEB. Since the timestamps of the inserted elements are always increasing we can handle insertions in a block in constant time.

Assume now v is small. We will then use the q*-heap of Fredman and Willard [112, 52] as predecessor structure. The q*-heap supports updates and predecessor queries among $O(\log^c n)$ elements in constant time per operation. To ensure that a predecessor structure will never contain too many elements we do as follows. For each element in a q*-heap in $\mathit{parray}(v)$ we place a pebble in v which can pay for a constant amount of time and space usage. When v contains $3\mathit{size}(v)$ pebbles we do as follows. We create a new copy v' of v with the elements of $\mathit{array}(v')$ initialized to $\mathit{array}(v)$ (and with $\mathit{parray}(v')$ initialized accordingly). Further, we take each of the at most $\mathit{size}(v)$ predecessor pointers of v and modify them to point to v' (we maintain in v the current set of predecessor pointers). $\mathit{size}(v)$ of the $3\mathit{size}(v)$ pebbles in v are used to pay for the work just described. $\mathit{size}(v)$ pebbles are used to place on v' and the remaining $size(v)$ pebbles are used to place on the small nodes in which we modify pointers to point to v' instead of v. It follows that the copying v to v' can be done amortized for free. □

6.6 Proof of Lemma 6.1

This section is devoted to the proof of Lemma 6.1. We prove each of the three parts of the lemma in different subsections.

6.6.1 Proof of Lemma 6.1 part 3

In this section we prove part 3 of Lemma 6.1. Alstrup et. al. [4] have showed:

Lemma 6.4. *A VEB can be modified such that only $O(1)$ memory cells are written on each update.*

From this we get:

Lemma 6.5. *A VEB can be made partial persistent with update time amortized $O(\log\log n)$ w.h.p., query time $O(\log^2\log n)$ and space usage $O(m)$ where m is the number of updates performed. Further, when we have made a query in a given version we can report the successors of the answer in that version in increasing order in constant time per element.*

Proof. We use the persistence technique of Lemma 6.3. We put the VEB of Lemma 6.4 into a single big node (we use a deterministic version of the VEB which do not use hashing). We then link the elements of the VEB together in order using small nodes which allows us to report the successors of a query answer in increasing order in constant time per element. □

Part 3 of Lemma 6.1 follows from Lemma 6.5: For each $y \in [1 \ldots \log n]$ we maintain a VEB of Lemma 6.5 with the indices e of B for which $B[e] = y$. Given a query we can just search in each of the at most $\log n$ relevant VEBs.

The improvement of Shi and JaJa [100] is obtained by removing a $O(\log^2\log n)$ factor from the query time as follows. We create a segment tree [41, 73] over $[1 \ldots n]$. For each pair of indices (e, e') such that e is the predecessor of e' in one of the $O(\log n)$ VEBs, we insert a segment from e to e' in the segment tree. Given a query (i, j, y) in B, a single $O(\log n)$ time query in the segment tree allows us to find the successor of i in each of the VEBs. Finally, we make the segment tree partial persistent with the technique of Section 6.5 using only small nodes.

As a corollary to Lemma 6.5 we get the following lemma announced in the introduction:

Lemma 6.6. *There exists a data structure for the colored segment reporting problem on an n times n grid where all colors are different with query time $O(\log^2\log n + k)$, space usage $O(n)$ and preprocessing time $O(n\log\log n)$ w.h.p.. Here k is the number of reported segments.*

Proof. We use the same algorithm as in Section 6.4 except that we use the partial persistent VEB of Lemma 6.5 instead of the structure from Theorem 6.2. This is possible because all colors are different. □

6.6.2 Proof of Lemma 6.1 part 1

In this section we give a proof sketch of part 1 of Lemma 6.1. First assume B contains at most $O(\log^c n)$ elements different from 0 for a constant c. Using the variant of priority search trees of Willard [112] we can maintain B using constant time for updates, linear space and time

$O(1+k)$ for queries where k is the number of reported elements. Plugging this structure with $c = 1$ into Theorem 3.1 we get a structure for B, without restriction on the number of elements different from 0 with update time $O(\log\log n)$, query time $O(\log\log n + k)$ and space usage $O(n \log n \log\log n)$. The space usage can be reduced to $O(n)$ using a standard trick: We group the elements of B into blocks with $O(\log^2 n)$ elements. For each block we insert a point with the largest y-coordinate into the structure just described and the points inside each block are kept in the variant of priority search tree of [112] with $c = 2$.

6.6.3 Proof of Lemma 6.1 part 2

In this section we give a proof sketch of part 2 of Lemma 6.1. The structure is obtained by applying Lemma 6.3 to the structure described in Section 6.6.2 except that we do not need the standard trick reducing the space usage. Looking into the structure Theorem 3.1 we see, that all nodes in the structure can be made small in the terminology of Lemma 6.3 with two exceptions: First, the structure uses (deterministic) VEBs to link elements together on the top level. But these VEBs are not used in connection with queries (only the links are used) and thus they do not need to be made partial persistent. Second, the structure contains a number of `bottom` arrays which need to be placed in big nodes. This is what increases the query time from $O(\log\log n + k)$ to $O(\log^2 \log n + k)$.

Chapter 7

Implicit Priority Queues with Decrease-Key

Implicit Priority Queues with Decrease-key

Christian W. Mortensen *   Seth Pettie †

Abstract

In this paper we show that any implicit priority queue with reasonable performance must use time amortized $\Omega(\log^* N)$ to perform a decreasekey operation. The decreasekey operation is reduced to a bit-probe-type game called the *Usher's Problem*, where one is asked to maintain a simple data structure without the aid of any auxiliary storage.

7.1 Introduction

An *implicit* data structure on N elements is one whose representation consists simply of an array $A[0..N-1]$, with one element stored in each array location. The most well known implicit structure is certainly Williams's binary heap [114], which supports the priority queue operations *insert* and *delete-min* in logarithmic time. Although the elements of Williams's heap are conceptually arranged in a fragment of an infinite binary tree, the tree edges are not explicitly represented. It is understood that the element at $A[i]$ is the parent of $A[2i+1]$ and $A[2i+2]$. The practical significance of implicit data structures is that they are, in certain circumstances, maximally space efficient. If the elements can be considered *atomic* then there is no better representation than a packed array.

In this paper we study the complexity of implicit priority queues that support the decrease-key operation. Our main result is that if an implicit priority queue uses *zero extra space* and supports decrease-key in $o(\log^* N)$ amortized time then the amortized cost of insert/delete-min must be $\Omega(N^{1/\log^{(k)} N})$, for any constant k. Our technique can also be extended to lower bound the worst-case complexity of decrease-keys. We show that if decreasekey takes worst-case time $o(\log \log N)$ then deletemin takes worst case-time $\Omega(N^{1/\log^{\epsilon} N})$ for any constant $\epsilon > 0$. Our lower bounds breaks if we have just one word of extra storage. We thus reinforce a theme in implicit data structures [115, 24, 44, 61], that it may take only a couple extra words of storage to alter the complexity of a problem.

In the original version of this paper [82] we claimed the following upper bounds. Given one extra word of storage we claimed there is an implicit priority queue matching the performance of Fibonacci heaps. That is, Delete-min would take logarithmic time and insert and decrease-key constant time, all amortized. Further, in case no extra storage was available we claimed we could handle decreasekey in amortized time $O(\log^* N)$ while keeping the same time usage for the other operations matching our lower bound for the amortized case. However, it has turned

*IT University of Copenhagen. Most of this work was done while the author was visiting the Max-Planck-Institut für Informatik, Saarbrücken, as a Marie Curie Doctoral Fellow. Email: cworm@itu.dk

†Max Planck Institut für Informatik. Supported by an Alexander von Humboldt Postdoctoral Fellowship. Email: pettie@mpi-sb.mpg.de

out that there was a bug in the analysis. The bug may be fixable; a thing we are working on. We are currently only sure we can handle decrease-key and inserts in amortized time $O(\log\log N)$ (with or without an extra word). The proof of this upper bound is not given here because (1) we still believe we may get an upper bound of $O(\log^* N)$ and (2) we currently have no good writeup of this upper bound.

For the lower bound, we reduce the decrease-key operation to the *Absent Minded Usher's Problem*, a game played in a simplified bit-probe model. Imagine an usher seating indistinguishable theater patrons one-by-one in a large theater. The usher is equipped with two operations: he can *probe* a seat to see if it is occupied or not and he can *move* a given patron to a given unoccupied seat. (Moving patrons after seating them is perfectly acceptable.) The catch is this: before seating each new patron we wipe clean the usher's memory. That is, he must proceed without any knowledge of which seats are occupied or the number of patrons already seated. We prove that any deterministic ushering algorithm must seat m patrons with $\Omega(m \log^* m)$ probes and moves, and that this bound is asymptotically tight.

Our lower bound proof attacks a subtle but fundamental difficulty in implicit data structuring, namely, orchestrating the movement of elements within the array, given little auxiliary storage. At present the ushering technique seems limited to proving small time-space tradeoffs, which is a valid criticism. However, we must point out that there are only a few techniques available for proving time-space lower bounds in succint data structures [33, 55, 76] and that none of them specifically address issues in *dynamic* data structuring. Furthermore, in many fundamental problems any non-trivial time-space tradeoff whatsoever would be welcome news. For instance, it is still an open problem whether an n-element subset of $\{1, \ldots, m\}$ can be stored in $\lceil \log \binom{m}{n} \rceil$ bits that allows membership queries in $O(1)$ time [55].

Organization. In Section 7.1.1 we define what an implicit priority queue is and what constitutes extra storage. Also, we further motivate the study of implicit priority queues by describing a possible interface between an implicit priority queue and an application. In Section 7.1.2 we summarize previous research. Section 7.2 is devoted to the usher's problem and in Section 7.3 we show our lower bounds on decreasekey.

7.1.1 Implicit Priority Queues

An implicit priority queue of size N consists of an array $A[0..N-1]$ (plus, possibly, some stored auxiliary information) where $A[0], \ldots, A[N-1]$ contain distinct elements (or *keys*) from an arbitrary total order. We also use A to denote the set of elements in the priority queue. The following operations are supported.

insert(κ)	: $A := A \cup \{\kappa\}$
deletemin()	: Return $\min A$ and set $A := A \backslash \{\min A\}$
decreasekey(i, κ)	: Set $A[i] := \min\{A[i], \kappa\}$

An operation decides what to do based on the auxiliary information and any comparisons it makes between elements. Before returning it is free to alter the auxiliary information and permute the contents of A, so long as its N elements lie in $A[0, \ldots, N-1]$. We assume that "N" is known to all operations.

The definition above sweeps under the rug a few issues that are crucial to an efficient and useful implementation. First, applications of priority queues store not only elements from a total order but *objects* associated with those elements. For example, Dijkstra's shortest path algorithm repeatedly asks for the *vertex* with minimum tentative distance; the tentative distance alone is useless. Any practical definition of a priority queue must take the application's objects into account. The second issue relates to the peculiar arguments to decreasekey. The application must tell decreasekey the *index* in A of the element to be decreased, which is necessarily a moving target since the data structure can permute the elements of A at will.

We propose a priority queue interface below that addresses these and other issues. Let us first sketch the normal (real world) interaction between an application and priority queue. To insert the object v the application passes the priority queue an identifier $id(v)$ of its choosing, together with $key(v)$, drawn from some total order. In return the priority queue gives the application a $pq\_id(v)$, also of its choosing, which is used to identify v in later calls to decreasekey. When v is removed from the queue, due to a deletemin, the application receives both $id(v)$ and $key(v)$. The application is expected to provide a comparator function for keys.

In our interface we give the data structure an extra degree of freedom, without placing any unreasonable demands on the governing application. Whereas the standard interface forces the data structure to assign *pq_id*s once and for all, we let the data structure update *pq_id*s as necessary. We also let the application maintain control of the keys. This is for two reasons, both concerning space. First, the application may not want to explicitly represent keys at all if they can be deduced in constant time, as may be the case in geometric applications. Second, the application can now use the same key in multiple data structures without producing a copy for each one. Below $\mathcal{Q}$ represents the contents of the data structure, which is initially empty. (Observe that this interface is modular. Neither the application nor the data structure needs to know any details about the other.)

The priority queue implements:

insert$(id(v))$	: Sets $\mathcal{Q} := \mathcal{Q} \cup \{id(v)\}$ and returns a $pq\_id(v)$
deletemin$()$	: Return, and remove from $\mathcal{Q}$, the $id(v)$ minimizing $key(v)$
decreasekey$(pq\_id(v))$	: A notification that $key(v)$ has been reduced

The application implements:

update$(id(v), x)$	: Set $pq\_id(v) := x$
compare$(id(v), id(w))$	: True iff $key(v) < key(w)$

Using this interface it is simple to implement an abstract implicit priority queue. The data structure would consist of an array of *id*s and we maintain, with appropriate update operations, that $pq\_id(v)$ indexes the position of $id(v)$ in A. For example, if we implemented a d-ary heap [66] with this interface every priority queue operation would end with at most $\log_d N$ calls to update, which is the maximum number of elements that may need to be permuted in A.

The only non-standard assumption made by our interface is that the application is capable of implementing the update function, that is, that it can locate where $pq\_id(v)$ is stored given $id(v)$, an identifier of *its* choosing. To our knowledge this is a valid assumption in every application of Fibonacci heaps [50].

Our interface should be contrasted with a more general solution formalized by Hagerup and Raman [60], in which *pq_id*s would be fixed once and for all. In their schema the application communicates with the data structure through an intermediary *quasidictionary*, which maps each *pq_id* (issued by the quasidictionary) to an identifier that can be updated by the data structure. Since saving space is one of the main motivations for studying implicit data structures we prefer our interface to one based on a quasidictionary.

7.1.2 Previous Work

Much of the work on implicit data structures has focused on the dictionary problem, in all its variations. The study of dynamic implicit dictionaries in the comparison model was initiated by Munro & Suwanda [84]. They gave a specific partial order (à la Williams's binary heap) that allowed inserts, deletes, and searches in $O(\sqrt{N})$ time, and showed, moreover, that with any partial order $\Omega(\sqrt{N})$ time is necessary for some operation. Munro [83] introduced a novel pointer encoding technique and showed that all dictionary operations could be performed in $O(\log^2 N)$

time, a bound that stood until just a few years ago. After a series of results Franceschini & Grossi [45] recently proved that all dictionary operations could be performed in worst-case logarithmic time with no extra storage. In other words, there is *no time-space tradeoff* at all for this dictionary problem. Franceschini & Grossi [46] also considered the static dictionary problem, where the keys consist of a vector of k characters. Their implicit representation allows for searches in optimal $O(k + \log N)$ time, which is somewhat surprising because Andersson et al. [9] already proved that optimal search is impossible if the keys are arranged in *sorted order*.

Yao [115] considered a static version of the dictionary problem where the keys are integers in the range $\{1, \ldots, m\}$. He proved that if m is sufficiently large relative to n then no implicit representation can support $o(\log n)$ time queries. His lower bound is sensitive in two respects: it requires *no* extra storage beyond the implicit representation and it requires that m be very large. In the same paper Yao proved that *one-probe* queries are possible if $m \leq 2n - 2$. Fiat et al. [44, 43] gave an implicit structure supporting constant time queries for any universe size, provided only $O(\log n + \log\log m)$ bits of extra storage. Zuckerman [117], improving a result of [44, 43], showed that $O(1)$ time queries are possible with zero extra storage, for m slightly superpolynomial in n. In the integer-key model we need to qualify the term "extra storage." An implicit representation uses $n\lceil \log m \rceil$ bits of storage, which is roughly $n \log n$ bits more than the information theoretic lower bound of $I = \lceil \log \binom{m}{n} \rceil$ bits. Some $I + o(I)$ space dictionaries were developed in [23, 90, 93, 94].

Implicit Priority Queues. Williams's binary heap [114] uses zero extra storage and supports inserts and deletemins in worst-case logarithmic time. It was generalized by Johnson [66] to a d-ary heap, which, for any fixed d, supports inserts and decreasekeys in $O(\log_d N)$ time and deletemins in $O(d \log_d N)$ time, all worst-case. Thus, the current best worst case upper bounds are far away from our worst case lower bound.

Carlsson et al.'s implicit binomial heap [24] supports inserts in constant time and deletemins in logarithmic time, both worst case. Their data structure uses $O(1)$ extra words of storage. Harvey and Zatloukal's postorder heap [61] also supports insert in $O(1)$ time; the time bound is amortized but they use zero extra storage. (Their data structure can be generalized to a postorder d-ary heap.)

General Priority Queues. The d-ary implicit heap has an undesirable tradeoff between decreasekeys and deletemins. In several applications of priority queues, like computing shortest paths, minimum spanning trees, and weighted matchings [50], the overall performance depends crucially on a fast decreasekey operation. Fredman and Tarjan's Fibonacci heap [50] demonstrates that all operations can be supported in optimal amortized time: constant for decreasekey, insert, and meld*, and logarithmic for deletemin. Aside from the space taken for *keys*, *ids*, and *pq_ids* (see Section 7.1.1), Fibonacci heaps require $4n$ pointers and $n(\log\log n + 2)$ bits. (Each node keeps a $(\log\log n + 1)$-bit rank, a mark bit, and references its parent, child, left sibling, and right sibling.) Kaplan and Tarjan [68] shaved the space requirements of Fibonacci heaps by n pointers and n bits. The Pairing heap [49] can be represented with $2n$ pointers and n bits. It handles decreasekey in sublograithmic time [91] but not in constant [48]. Fredman and Tarjan [50] claimed that a more complicated version of Fibonacci heaps uses just two pointers per node, though it was not described in [50]. We conclude with remarking, that it is possible to use two pointers per node as follows. We use one pointer in each node to link nodes together in *rings* with constant length. This gives us a constant number of unused pointers for each ring and we can use this to insert the ring in a traditional pointer based heap such as the Fibonacci

*Most applications of priority queues do not need a meld operation.

heap. We use the minimal element of the ring as key. Actually, this scheme allows us to reduce the space usage to one pointer per node except for an arbitrary small constant fraction of the nodes which need two pointers.

7.2 The Absent Minded Usher's Problem

In Section 7.3 we show that the decreasekey operation must be prepared to solve the Absent Minded Usher's Problem which is the topic of the present section.

Let A be an array of infinite length, where each location of A can contain a *patron* (indistinguishable from other patrons) or be empty. An *usher* is a program to insert a new patron into A. We are interested in the *usher process* which consists of invoking a given usher N times to insert N patrons into an initially empty array. The usher can in unit time move a patron to a new seat. Further, the usher can in unit time get a bit from an oracle. For example, the usher can ask the oracle if a given seat is occupied or ask for a random bit. The usher cannot get information in any other way. In particular, the usher cannot get more than one random bit in unit time and the usher does not know how many patrons it has already seated in A as a part of the usher process.

Theorem 7.1. *There exists an usher such that the user process takes time $O(N \log^* N)$. The usher only asks the oracle if a given seat is occupied or not.*

Proof. Divide up A into consecutive buffers (subarrays) $\{B_i\}$ where $|B_1| = 2$ and $|B_{i+1}| = |B_i|2^{|B_i|}$. Our algorithm maintains the invariant that all occupied positions of B_i occur before the unoccupied ones. The usher does as follows: seat the patron in $B_1[0]$, or if that is occupied, in $B_1[1]$. At this point B_1 may be fully occupied. In general, whenever B_i is fully occupied we perform a binary search to determine the minimum j s.t. $B_{i+1}[j]$ is unoccupied. We then move the contents of B_i to the positions $B_{i+1}[j], \ldots, B_{i+1}[j + |B_i| - 1]$. The cost for moving B_i to B_{i+1} is $\log(|B_{i+1}|/|B_i|) + |B_i| = 2|B_i|$. Since each patron appears in each buffer at most once, the amortized cost for N invocations of the usher is $2N \cdot \min\{i : |B_i| > N\} \leq 2N \log^* N$. □

The rest of this section is devoted to showing that this usher is asymptotically optimal by proving a matching lower bound in the form of Theorem 7.6 below. To simplify things we assume the patron to be inserted lies in the artificial position $A[-1]$. The usher is modeled as a binary decision tree. At each leaf is a list of pairs of the form (j_1, j_2), indicating that the patron at $A[j_1]$ should be moved to $A[j_2]$. Each leaf is called an *operation* and the cost of executing an operation o is its depth in its tree, denoted $d(o)$, plus the number of patron movements, denoted $m(o)$. On each invocation of the usher the oracle decides which path to follow in the tree, that is, which operation to execute.

Let us first sketch how our proof works. We imagine an infinite graph whose vertices are layed out in a grid (Figure 7.1). The x-axis corresponds to time (number of usher invocations) and the y-axis corresponds to the array A. The usher process can be modeled as N x-monotone paths through the grid, with each path representing where a particular patron was after each invocation of the usher. We assign each edge a cost, which represents in an amortized sense the time taken to bring that patron to its current position. By reasoning about the usher's decision tree we are able to derive a recurrence relation describing the costs of edges. The solution to this recurrence is then used to lower bound the complexity of the ushering process.

To be precise, define for a given usher process an infinite graph with vertex set $\{A_i[j] : i \geq 0, j \geq -1\}$, where $A_i[j]$ represents $A[j]$ *after* i usher invocations. There exists an edge $(A_{i-1}[j_1], A_i[j_2])$ exactly when $A_{i-1}[j_1]$ and $A_i[j_2]$ contain the same patron. Note that the graph is composed exclusively of paths. An edge is *leftover* if it is of the form $(A_{i-1}[j], A_i[j])$ and *fresh*

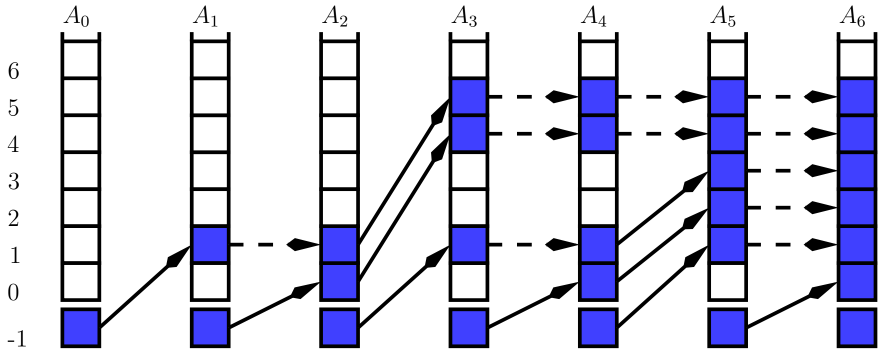

Figure 7.1: This (infinite) two dimensional grid depicts the flow of patrons over time for a given usher process. The ith patron to be inserted lies in position $A_{i-1}[-1]$. Solid edges are fresh, dashed ones leftover.

otherwise, i.e., fresh edges correspond to patron movements and leftover edges correspond to unmoved patrons. Let $\text{pred}(A_{i-1}[j_1], A_i[j_2])$ denote the edge $(A_{i-2}[j_3], A_{i-1}[j_1])$ if such an edge exists.

We now define a cost function c. If the edge (u, v) does not exist then $c(u, v) = 0$. Any edge of the form $(A_{i-1}[j], A_i[j])$ has cost $c(\text{pred}(A_{i-1}[j], A_i[j]))$: leftover edges inherit the cost of their predecessor. Let $C_i = \sum_{j_1,j_2} c(A_{i-1}[j_1], A_i[j_2])$ be the total cost of edges into A_i, and let $P_i = \sum_{j_1 \neq j_2} c(\text{pred}(A_{i-1}[j_1], A_i[j_2]))$ be the cost of the *predecessors* of the fresh edges into A_i. Let o_i be the operation performed at the ith usher invocation. (For example, in the ushering algorithm partially depicted in Figure 7.1, $o_2 = o_4 = o_6$ and o_1, o_3, and o_5 are distinct.) Each fresh edge e between A_{i-1} and A_i is assigned the same cost:

$$c(e) \stackrel{\text{def}}{=} \frac{P_i + d(o_i) + m(o_i)}{m(o_i)}$$

That is, the total cost assigned to these $m(o_i)$ fresh edges is their inherited cost, P_i, plus the actual cost of the operation o_i, that is, $d(o_i) + m(o_i)$. It follows from the definitions that C_i is exactly the cost of inserting the first i patrons. Let $T(m)$ be the *minimum* possible cost of a fresh edge associated with an operation performing m movements. We will show $T(m) = \Omega(\log^* m)$ and that this implies the amortized cost of N usher invocations is $\Omega(N \log^* N)$.

For the remainder of this section we consider the ith invocation of the usher. Suppose that the operation o_i moves patrons *from* locations $A_{i-1}[j_1, j_2, \ldots, j_{m(o_i)}]$. The patrons in these locations were placed there at various times in the past. Define $i_q < i$ as the index of the usher invocation that *last* moved a patron to $A[j_q]$.

Lemma 7.2. *Let p, q be indices between 1 and $m(o_i)$. If $i_p \neq i_q$ then $o_{i_p} \neq o_{i_q}$.*

Proof. Assume w.l.o.g. that $i_p < i_q$. The patron at $A_i[j_p]$ was placed there at usher invocation number i_p. However, if $o_{i_p} = o_{i_q}$ then usher invocation number i_q must also have placed some patron at $A[j_p]$, contradicting the definition of i_p since $i_p < i_q$. □

We categorize all operations in the usher's decision tree w.r.t. $m = m(o_i)$—recall that i is fixed in this section. An operation o is *shallow* if $d(o) < \lfloor \log m \rfloor / 2$ and *deep* otherwise. It is *thin* if $m(o) < \lfloor \sqrt{\log m} \rfloor$ and *fat* otherwise. (Our proof is not very sensitive to the definitions of thin and shallow.)

Lemma 7.3. $\left|\{q : o_{i_q} \textit{ is shallow and thin}\}\right| < \frac{1}{2}\sqrt{m \log m}$

Proof. By Lemma 7.2 all the *distinct* usher invocations in $\{i_1, \ldots, i_{m(o_i)}\}$ correspond to distinct operations. There are at most $2^{\log m/2-1} = \sqrt{m}/2$ different shallow operations and each thin operation contributes less than $\sqrt{\log m}$ moves. The lemma follows. $\square$

Lemma 7.4. *If (u, v) is a fresh edge, where $v \in A_k$ and o_k is deep and thin, then $c(u, v) \geq \frac{1}{2}\sqrt{\log m}$.*

Proof. We have $c(u, v) \geq \frac{d(o_k)+m(o_k)}{m(o_k)}$. Since $d(o_k) \geq \lfloor \log m \rfloor /2$ and $m(o_k) < \lfloor \sqrt{\log m} \rfloor$ this implies $c(u, v) \geq \frac{1}{2}\sqrt{\log m}$. $\square$

In summary, Lemma 7.3 implies that at least $m - \sqrt{m \log m}/2$ of the fresh edges between A_{i-1} and A_i can be traced back to earlier edges that are either in deep and thin operations or fat operations. The cost of edges in deep and thin operations is bounded by Lemma 7.4 and the cost of edges in fat operations is bounded inductively. Recall that $T(m)$ is the minimum cost of a fresh edge associated with an operation performing m moves.

Lemma 7.5. $T(m) \geq (\log^* m)/4$

Proof. Let e be a fresh edge into A_i with $m = m(o_i)$ and let $\beta = \min\{\frac{1}{2}\sqrt{\log m}, T(\sqrt{\log m}), T(\sqrt{\log m} + 1), \ldots\}$. Then:

$$c(e) = \frac{m(o_i) + P_i + d(o_i)}{m(o_i)} \geq \frac{m + (m - \frac{1}{2}\sqrt{m \log m}) \cdot \beta}{m}$$

Since the only property of e that we used in the above inequalities is $m(o_i) = m$, any lower bound on $c(e)$ implies the same lower bound on $T(m)$. We assume inductively that $T(r) \geq \frac{1}{4}\log^* r$, which holds for $r \leq 2^{16}$ since $T(r) \geq 1$ and $\log^* 2^{16} = 4$. For $m > 2^{16}$ we have:

$$\begin{aligned}
T(m) &\geq 1 + \left(1 - \sqrt{\log m/4m}\right) \cdot \beta \\
&\geq 1 + \left(1 - \sqrt{\log m/4m}\right) \cdot \frac{\log^*(\sqrt{\log m})}{4} \\
&> \left(1 - \sqrt{\log m/4m}\right)\left(\frac{\log^* m - 2}{4} + 1\right) > \frac{\log^* m}{4}
\end{aligned}$$

$\square$

Theorem 7.6. *For any usher the ushering process uses time $\Omega(N \log^* N)$. Some invocation of the user must use time $\Omega(\log N)$.*

Proof. The theorem basically follows from Lemma 7.5. After N usher invocations we perform an artificial operation o^*, with $d(o^*) = 1$ and $m(o^*) = N$, that moves the item in $A[j]$ (if it contains one) to $A[j+1]$. By definition of C_i we have $C_{N+1} = \sum_j c(A_N[j], A_{N+1}[j+1])$, which by Lemma 7.5 is at least $N \log^* N/4$. The operation o^* is most likely fictitious, though we can include it in the usher's decision tree by incrementing the depth of every other operation by one. Thus, the cost of the first N usher invocations, is $C_{N+1} - 2N - 1 = \Omega(N \log^* N)$.

The second part of the theorem is nearly trivial. Consider the operations whose depth and movement cost are both at most $\log N/2$. The number of array locations referenced by these operations is at most $\sqrt{N} \log N/2$. Thus N patrons cannot be inserted with these operations alone. $\square$

7.3 Lower bound for decreasekey

In this section we show a lower bound for the decreasekey operation. We assume that in each step the decreasekey operation can get a bit from an oracle. Note that if we allowed the same for delete then it would be trivial to make decreasekey and findmin in constant time and delete in logarithmic time. We let N be the number of elements in A and we assume all operations know N. We show the following theorem.

Theorem 7.7. *Let $k > 0$ be any constant. If decreasekey takes amortized time $o(\log^* N)$ then deletemin/insert takes amortized time $\Omega(N^{1/\log^{(k)} N})$.*

The rest of this section is devoted to the proof of following lemma to which Theorem 7.7 is a direct corollary. We let $\epsilon > 0$ be any constant and define $b = \log^*(N)$ and $m = \log^{(b-\epsilon b)}(N)$.

Lemma 7.8. *In any implicit priority queue we can always do at least one of the following.*

1. *For some $1 \le d \le N$, perform d decreasekeys that will use time $\Omega(db)$.*
2. *Perform at most $(b+1)m$ decreasekeys followed by $(b+1)m$ deletemins followed by $(b+1)m$ inserts that will use time $\Omega(N^{1/(2b^2m)})$.*

We let A_0 denote the initial content of the array which we assume is fixed throughout the proof. We let decreasekey(j) on A denote the operation that decreasekeys entry j of A to the minimum among the elements of A. We first execute *round 1* as follows. We select an index $0 \le i \le N-1$. We describe at the end of the proof exactly which index we select. We then perform d decreasekey(i) operations on A_0 obtaining an array A_1. We select d to be minimal such that $A_1[i]$ contains an element we have performed decreasekey on. Observe that the decreasekey(i) operation then solves the Absent Minded Usher's Problem with d insertions so Theorem 7.6 gives that this takes time $\Omega(d \log^* d)$. If $d \ge m$ this is $\Omega(d \log^*(m)) = \Omega(d\epsilon b) = \Omega(db)$ and Lemma 7.8 follows (situation 1). Otherwise, if $d < m$ and A_0 and A_1 differ in at least bm positions then Lemma 7.8 also follows because the at most m decreasekey(i) operations would have taken time $\Omega(bm)$ to execute (situation 1). So for the rest of the proof we can assume that after round 1, A_0 and A_1 differ in at most bm positions and $A_1[i]$ contains one of the m smallest elements of A_1. After round 1 we proceed with *round 2*.

In this paragraph we give some intuition for the rest of the proof. Let S be the at most bm elements (not indices) which are moved in round 1. After round 1, the fact that $A_1[i]$ is among the m smallest elements in A_1 is encoded by elements of S. In round 2 we will perform a decreasekey operation on some elements of S basically destroying the encoding of i. After this we will perform $(b+1)m$ deletemin operations which will need to identify $A_1[i]$ and we will argue that that we can select i such that this takes long time because the information about i has been destroyed. The major problem is that the decreasekey operations in round 2 may *reencode* the information about i we try to destroy based on which elements we decreasekey. Note that, in principle it is possible to encode $\Omega(\log n)$ bits in unit time†. So we must select elements from S to perform decreasekeys on such that (1) we destroy enough information about i and (2) the fact that we perform decreasekey operations on *those* elements does not give the priority queue much information about i. We obtain these properties by performing decreasekey operations on the elements from S which the $(b+1)m$ deletemin operations tries to read.

Inspired by the intuition just given, we will not look at the deletemin operation but instead at the *locatemins* operation. The locatemins operation on A must identify the smallest $(b+1)m$ keys of A (in no particular order). Clearly, the locatemins operation can be simulated by $(b+1)m$ deletemin operations followed by $(b+1)m$ insert operations. We model the locatemins

†To see this, assume the array is initially sorted. We can encode the integer ℓ by swapping $A[0]$ and $A[\ell]$; decoding is accomplished in $O(\log n)$ time by a binary search.

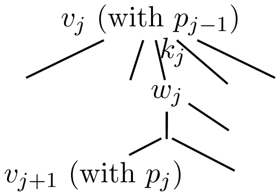

Figure 7.2: The path in T from v_j to v_{j+1} followed in round 2.

operation as a decision tree $T_{\min}$. Each internal node of $T_{\min}$ has two children and is labeled with a pair (j, j') with $0 \leq j, j' \leq N-1$. When we execute the tree on an array A, we go to the left in such a node if $A[j] < A[j']$ and otherwise we go to the right. Each leaf of $T_{\min}$ contains a set with the $(b+1)m$ indices of A which contain the smallest $(b+1)m$ elements. We model decreasekey(j) as the decision tree T^j_{dec}. Again, each internal node has two children. During an execution we assume an oracle decides which child to go to. Each leaf of T^j_{dec} contains a permutation of the elements of A describing how the array is modified. Formally, the trees $T_{\min}$ and T^j_{dec} are dependent on the number of elements in the array they are executed on since each operation gets that as a parameter. But we can ignore this issue because we only execute $T_{\min}$ and T^j_{dec} on arrays with N elements.

We will now define an infinite tree T without leafs which is independent of i. We will use T to select i and to model the possible executions of round 2. T will consist of two kind of nodes: *oracle* nodes and *decrease* nodes. First, we create a decrease node v as the root of T and define $A(v) = A_0$. We now describe how to give children to an arbitrary decrease node $v' \in T$. We will give each child of v' a label and going to a child with label j corresponds to performing a decreasekey(j) operation. Let π be the path we follow in $T_{\min}$ if we execute $T_{\min}$ on $A(v')$. Let $(i_1, i_2), (i_3, i_4), \ldots, (i_{q-1}, i_q)$ be the labels of the nodes in π in order. We then give v' a total of q children where we give the edge to the jth child of v' label i_j and make it a root of a copy of $T^{i_j}_{\text{dec}}$. It follows that the edge to two different children of v' can have the same label. The internal nodes of the copy of $T^{i_j}_{\text{dec}}$ become oracle nodes while the leafs become decrease nodes. Each such decrease node v'' will have a permutation p. We set $A(v'')$ to $A(v')$ permuted by p with the exception that we set $A(v'')[p(i_j)]$ to be minimal among the elements of $A(v'')$. We then recursively give v'' children in the way just described.

We will now describe how we execute round 2 (see Figure 7.2). Let I_1 be the at most bm indices where A_1 is different from A_0. Let v_1 be the root of T and set $j = 1$. We select the first child $w_j \in T$ of v_j with a label $k_j \in I_j$. Suppose first the following conditions are met:

c0. w_j exists.

c1. w_j has fewer than $N^{1/(2b^2m)}$ siblings to the left.

c2. v_j has depth in T smaller than $(b+1)mb$.

We then go to w_j and execute a decreasekey(k_j) operation. We follow the path from w_j to a decrease node v_{j+1} corresponding to the way the decreasekey(k_j) operation executes. Let p_j be the permutation in v_{j+1}. We define A_{j+1} to A_j permuted by p_j with the exception that we set $A_{j+1}[p_j(k_j)]$ to be minimal among the elements of A_{j+1}. Further, we define I_{j+1} to be $I_j \setminus \{k_j\}$ permuted by p_j. We then increment j by one and continue in the same way. Suppose next that one of c0, c1 or c2 is not met. Then we set $j_0 = j$, execute a locatemins unless c2 is not met, and we are done.

We observe that since $|I_{j+1}|$ is one smaller than $|I_j|$ and since $|I_1| \leq bm$ we will perform at most bm decreasekey operations in round 2 before condition c0 is not met. It follows, that if c2 is not met when we stop, then we would have spent time $(b+1)mb$ on performing at most $(b+1)m$ decreasekey operations (in round 1 and 2 together) and Lemma 7.8 (situation 1) follows. What remains is to see what happens if condition c2 is met when we stop.

Lemma 7.9. *For $1 \leq j \leq j_0$, $j' \notin I_j$ and $j'' \notin I_j$ we have*

$$A(v_j)[j'] < A(v_j)[j''] \iff A_j[j'] < A_j[j''] \tag{7.1}$$

Proof. We show the lemma by induction on j. By definitions the lemma is true for $j = 1$. For $1 < j \leq j_0$ observe that by construction of T we have $A(v_j)$ is equal to $A(v_{j-1})$ permuted by p_{j-1} except $A(v_j)[p_{j-1}(k_{j-1})]$ is minimal among the elements of $A(v_j)$. By the way we execute round 2, A_j is equal to A_{j-1} permuted by p_{j-1} except $A_j[p_{j-1}(k_{j-1})]$ is minimal among the elements of A_j. It immediately follows, that (7.1) is fulfilled for all j'' and $j' = p_{j-1}(k_{j-1})$. From the induction hypothesis and the fact that I_j is equal to $I_{j-1} \setminus \{k_{j-1}\}$ permuted by p_{j-1}, it then follows that (7.1) is fulfilled for all $j', j'' \in I_j$ for which j' and j'' are different from $p_{j-1}(k_{j-1})$ completing the proof. □

We will first argue, that if c2 is met and c1 is not met when we stop, then we will use time $\Omega(N^{1/(2b^2m)})$ on the final locatemins operation and Lemma 7.8 (situation 2) follows. Lemma 7.9 ensures, that as long as we do not read an entry from I_{j_0}, then we follow the same path in $T_{\min}$ if we execute it according to A_{j_0} as if we execute it according to $A(v_{j_0})$. So by construction of T, when we execute $T_{\min}$ on A_{j_0} we will perform at least one probe for each of the at least $N^{1/(2b^2m)}$ siblings to the left of w_{j_0}.

Finally, we will describe how to select i such that c0 is always met when we stop, concluding the proof of Lemma 7.8. Consider a node $v \in T$. Suppose we execute a locatemins operation on $A(v)$. This will identify $(b+1)m$ indices of $A(v)$. Further, by the permutations in the ancestors of v in T this will correspond to $(b+1)m$ position in A_0 where the elements were initially (they may have been decreasekeyed to the minimum in the meantime). Let I be the set of these indices for the decrease nodes v we can reach from the root of T by never visiting a node with more than $N^{1/(2b^2m)}$ siblings to the left and never visit a node with depth larger than $(b+1)mb$. We have $|I| \leq (b+1)mN^{(b+1)mb/(2b^2m)} < N$. We then select $1 \leq i \leq N$ such that $i \notin I$. Suppose now to obtain a contradiction that condition c0 is not met when we stop, that is that w_{j_0} does not exist. It then follows from Lemma 7.9 that if we executed a deletemin on A_{j_0} then we would get the same result as if we did it on $A(v_{j_0})$. It follows that this deletemin will not be able to identify the element which is on index i of A_1 which contradicts the correctness of deletemin.

7.3.1 Worst case lower bound for decreasekey

Let $\epsilon > 0$ be any constant.

Theorem 7.10. *If decreasekey takes worst case time $o(\log\log N)$ then deletemin takes worst case time $\Omega(N^{1/\log^{\epsilon} N})$.*

Proof. The proof of the theorem is identical to the proof of Theorem 7.7. except that we select $b = \log\log N$, $m = \log^{\epsilon/2} N$ and use the worst case lower bound of Theorem 7.6 instead of the amortized. □

Bibliography

[1] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives, 1999.

[2] Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. Range searching in categorical data: Colored range searching on grid. In *10th European Symposium on Algorithms*, pages 17–28, 2002.

[3] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[4] Stephen Alstrup, Gerth Brodal, Inge Li Görtz, and Theis Rauhe. Time and space efficient multi-method dispatching. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 2002.

[5] Stephen Alstrup, Gerth Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In ACM, editor, *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing: Hersonissos, Crete, Greece, July 6–8, 2001*, pages 476–482, New York, NY, USA, 2001. ACM Press.

[6] Stephen Alstrup, Gerth S. Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 198–207. IEEE Computer Society Press, 2000.

[7] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'2002 (Winnipeg, Canada, August 10-13, 2002)*, pages 258–264, New York, 2002. ACM SIGACT, ACM SIGARCH, ACM Press.

[8] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 534–543. IEEE Computer Society Press, 1998.

[9] Arne Andersson, Torben Hagerup, Johan Håstad, and Ola Petersson. Tight bounds for searching a sorted array of strings. *SIAM J. Comput.*, 30(5):1552–1578, 2000.

[10] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121, Bergen, Norway, 11–14 July 1990. Springer.

[11] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the thirty second annual ACM Symposium on Theory of Computing*, pages 335–342. ACM Press, 2000.

[12] Lars Arge, Gerth S. Brodal, Rolf Fagerberg, and Morten Laustsen. Cache-oblivious planar orthogonal range searching and counting. In *Symposium on Computational Geometry*, pages 160–169, 2005.

[13] Lars Arge, Vasilis Samoladas, and Jeffrey S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 346–357. ACM Press, 1999.

[14] Lars Arge and Jeffrey S. Vitter. Optimal external memory interval management. *SICOMP: SIAM Journal on Computing*, 32, 2003.

[15] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

[16] Jon L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, June 1979.

[17] Jon L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[18] Jon L. Bentley and J. B. Saxe. Decomposable searching problems. I. static-to-dynamic transformation. *ALGORITHMS: Journal of Algorithms*, 1, 1980.

[19] Jon L. Bentley and Michael I. Shamos. A problem in multivariate statistics: Algorithm, data structure, and applications. In *Proceedings of the Fifteenth Allerton Conference on Communication, Control, and Computing*, pages 193–201, September 1977.

[20] Jon L. Bentley and Derick Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29:571–577, July 1980.

[21] Panayiotis Bozanis, Nectarios Kitsios, Christos Makris, and Athanasios K. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Automata, Languages and Programming, 22nd International Colloquium*, volume 944 of *Lecture Notes in Computer Science*, pages 464–474, Szeged, Hungary, 1995. Springer-Verlag.

[22] Gerth S. Brodal. Predecessor queries in dynamic integer sets. In *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1200, pages 21–32. Springer, 1997.

[23] A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM J. Comput.*, 28(5):1627–1640, 1999.

[24] S. Carlsson, J. I. Munro, and P. V. Poblete. An implicit binomial queue with constant insertion time. In *Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Comput. Sci.*, pages 1–13. Springer, Berlin, 1988.

[25] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Computing*, 15(3):703–724, August 1986.

[26] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, June 1988.

[27] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, April 1990.

[28] Bernard Chazelle. Lower bounds for orthogonal range searching: II. the arithmetic model. *Journal of the ACM*, 37(3):439–463, July 1990.

[29] Bernard Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete Comput. Geom.*, 3:113–126, 1987.

[30] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[31] Siu W. Cheng and Ravi Janardan. Efficient dynamic algorithms for some geometric intersection problems. *Information Processing Letters*, 36(5):251–258, December 1990.

[32] Yi-Jen Chiang and Roberto Tamassia. Dynamic algorithms in computational geometry. Technical Report CS-91-24, Department of Computer Science, Brown University, March 1991.

[33] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 289–294, New York, January 7–9 2001. ACM Press.

[34] Paul. F. Dietz. Fully persistent arrays. In *Algorithms and Data Structures. Workshop (WADS '89)*, pages 67–74, Berlin - Heidelberg - New York, August 1989. Springer.

[35] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Algorithms and Data Structures, Workshop WADS '89*, pages 39–46. Springer, 1989.

[36] Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 78–88, 28–30 January 1991.

[37] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, May 1987.

[38] Martin Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science (STACS '96)*, volume 1046 of *Lecture Notes in Computer Science*, pages 569–580. Springer-Verlag, 1996.

[39] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 1990.

[40] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[41] Herbert Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical Report F59, Inst. Informationsverarb., Tech. Univ. Graz, 1980.

[42] Herbert Edelsbrunner and Mark H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14:124–127, 1982.

[43] Amos Fiat and Moni Naor. Implicit $O(1)$ probe search. *SIAM J. Comput.*, 22(1):1–10, 1993.

[44] Amos Fiat, Moni Naor, Jeanette P. Schmidt, and Alan Siegel. Nonoblivious hashing. *J. ACM*, 39(4):764–782, 1992.

[45] Gianni Franceschini and Roberto Grossi. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th Int'l Workshop on Algorithms and Data Structures (WADS)*, 2003.

[46] Gianni Franceschini and Roberto Grossi. No sorting? better searching! In *Proc. 45th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 491–498, 2004.

[47] Michael L. Fredman. A lower bound on the complexity of orthogonal range queries. *J. ACM*, 28(4):696–705, 1981.

[48] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999.

[49] Michael. L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[50] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[51] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, December 1993.

[52] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, June 1994. See also FOCS'90.

[53] Otfried Fries, Kurt Mehlhorn, Stefan Näher, and Athanasios K. Tsakalidis. A log log n data structure for three-sided range queries. *Inf. Process. Lett*, 25(4):269–273, 1987.

[54] Harold N. Gabow, Jon L. Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, New York, NY, USA, 1984. ACM Press.

[55] Anna Gál and Peter B. Miltersen. The cell probe complexity of succinct data structures. In *Proc. 30th Int'l Colloq. on Automata, Languages, and Programming*, pages 332–344, 2003.

[56] Sathish Govindarajan, Pankaj K. Agarwal, and Lars Arge. CRB-Tree: an efficient indexing scheme for range-aggregate queries. In *Proceedings of the 9th International Conference on Database Theory*, Siena, Italy, 2003.

[57] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *Journal of Algorithms*, 19(2):282–317, September 1995.

[58] Prosenjit Gupta, Ravi Janardan, Michiel H. M. Smid, and Bhaskar DasGupta. The rectangle enclosure and point-dominance problems revisited. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 162–171, New York, NY, USA, June 1995. ACM Press.

[59] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.

[60] Torben Hagerup and Rajeev Raman. An efficient quasidictionary. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 2002.

[61] N. J. A. Harvey and K. Zatloukal. The post-order heap. In *Proc. 3th Int'l Conf. on FUN with algorithms*, 2004.

[62] Hiroshi Imai and Takao Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8(1):1–18, March 1987.

[63] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *Automata, Languages and Programming, 8th Colloquium*, volume 115, pages 417–431. Springer, 13–17 July 1981.

[64] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *ISAAC*, pages 558–568, 2004.

[65] Ravi Janardan and M. Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry and Applications*, 3:39–69, 1993.

[66] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Info. Proc. Lett.*, 4(3):53–57, 1975.

[67] Kothuri V. Ravi Kanth and Ambuj K. Singh. Efficient dynamic range searching using data replication. *Information Processing Letters*, 68(2):97–105, 30 October 1998.

[68] Haim Kaplan and Robert E. Tarjan. New heap data structures. Technical Report TR-597-99, Computer Science Dept., Princeton University, March 1999.

[69] D. T. Lee and C. K. Wong. Quintary trees: A file structure for multidimensional database systems. *j-TODS*, 5(3):339–353, September 1980.

[70] George S. Lueker. A data structure for orthogonal range queries. In *19th Annual Symposium on Foundations of Computer Science*, pages 28–34. IEEE Computer Society Press, October 1978.

[71] Christos Makris and Athanasios Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6):277–283, June 1998.

[72] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[73] Erwin M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical Report CSL-80-9, Xerox Corp., 1980.

[74] Kurt Mehlhorn. *Data Structures and Algorithms: 3. Multidimensional Searching and Computational Geometry.* Springer, 1984.

[75] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[76] Peter B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. 16th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages ??–??, 2005.

[77] Peter B. Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *J. Comput. System Sci.*, 57(1):37–49, 1998. Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC '95).

[78] Christian W. Mortensen. Fully-dynamic orthogonal range reporting on RAM. Technical Report TR-22, IT University of Copenhagen, April 2003.

[79] Christian W. Mortensen. Fully-dynamic two dimensionl orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[80] Christian W. Mortensen. Generalized static orthogonal range searching in less space. Technical Report TR-33, IT University of Copenhagen, September 2003.

[81] Christian W. Mortensen, Rasmus Pagh, and Mihai Patrascu. On dynamic range reporting in one dimension. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 2005.

[82] Christian W. Mortensen and Seth Pettie. The complexity of implicit and space efficient priority queues. In *WADS*, pages 49–60, 2005.

[83] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986.

[84] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *J. Comput. Syst. Sci.*, 21(2):236–250, 1980.

[85] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 657–666, New York, January 6–8 2002. ACM Press.

[86] Yakov Nekrich. Space efficient dynamic orthogonal range reporting. In *SCG*, pages 306–313. ACM Press, 2005.

[87] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[88] Mark H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, June 1988.

[89] Mark H. Overmars and Jan van Leeuwen. Dynamic multi-dimentional data structures based on quad- and $K-D$ trees. *Acta Informatica*, 17:267–285, 1982.

[90] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.

[91] Seth Pettie. Towards a final analysis of pairing heaps. In *Proceedings 46th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 174–183, 2005.

[92] Franco P. Preparata and Michael I. Shamos. *Computational Geometry*. Springer, New York, 1985.

[93] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 233–242, 2002.

[94] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th Int'l Colloq. on Automata, Languages and Programming*, pages 357–368, 2003.

[95] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pages 331–340. ACM Press, 1993.

[96] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995.

[97] Pranab Sen and S. Venkatesh. Lower bounds for predecessor searching in the cell probe model. *CoRR*, cs.CC/0309033, 2003.

[98] Qingmin Shi and Joseph JaJa. Fast algorithms for 3-d dominance reporting and counting. Technical Report CS-TR-4437, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

[99] Qingmin Shi and Joseph JaJa. Fast fractional cascading and its applications. Technical Report CS-TR-4502, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

[100] Qingmin Shi and Joseph JaJa. Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. *Inf. Process. Lett.*, 95(3):382–388, 2005.

[101] Alan Siegel. On universal classes of extremely random constant time hash functions and their time-space tradeoff. Technical Report TR1995-684, New York University, 1995.

[102] Michiel Smid. Maintaining the minimal distance of a point set in less than linear time. *ALCOM: Algorithms Review, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM)*, 2, 1991.

[103] Michiel Smid. Range trees with slack parameter. *ALCOM: Algorithms Review, Newsletter of the ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM)*, 2, 1991.

[104] Peter van Emde Boas. Preserving order in a forest in less than logatirhmic time and linear space. *Infom. Process. Lett.*, 6:80–82, 1977.

[105] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[106] Dan E. Willard. *Predicate-Oriented Database Search Algorithms*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1978.

[107] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 24 August 1983.

[108] Dan E. Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing*, 14(1):232–253, February 1985. See also TR-22-78, Center for Research in Computing Technology, Harvard University, 1978.

[109] Dan E. Willard. Reduced memory space for multi-dimensional search trees (extended abstract). In *STACS*, pages 363–374, 1985.

[110] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, April 1992.

[111] Dan E. Willard. Application of range query theory to relational data base join and selection operations. *Journal of Computer and System Sciences*, 52(1):157–169, February 1996.

[112] Dan E. Willard. Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, June 2000.

[113] Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, July 1985.

[114] J. W. J. Williams. Algorithm 232 (heapsort). *Comm. ACM*, 7:347–348, 1964.

[115] A. C. C. Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981.

[116] Ju Zhang. Density control and on-line labeling probelms. Technical Report TR481, University of Rochester, Computer Science Department, December 1993. Ph.d. thesis.

[117] D. Zuckerman. *Computing Efficiently Using General Weak Random Sources.* Ph.D. Thesis, The University of California at Berkeley, August 1991.