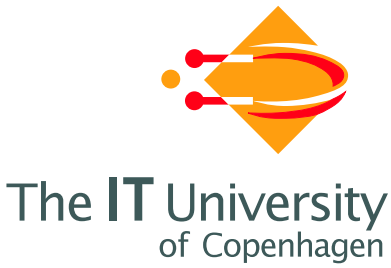

Fully-dynamic orthogonal range reporting on RAM

Preliminary version

Christian Worm Mortensen

IT University Technical Report Series

TR-2003-22

ISSN 1600–6100

April 2003

ISSN 1600–6100

ISBN 87-7949-030-1

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web www.it-c.dk

Fully-dynamic orthogonal range reporting on RAM

Preliminary version

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

April 5, 2003

Abstract

In a natural variant of the comparison model, we show that there exists a constant $\omega < 1$ such that the fully-dynamic d-dimensional orthogonal range reporting problem for $d \geq 2$ 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 $O(n \log^{\omega+d-2} n)$. In the standard comparison model the result holds for $d \geq 3$.

Contents

1 Introduction

The d-dimensional fully-dynamic orthogonal range reporting problem (henceforth the d-dimensional range reporting problem) is to maintain a finite set $R \subset \mathbb{R}^d$ of points under insertions and deletions such that for a given query $Q = [x_1 \ldots x_1'] \times \cdots \times [x_d \ldots x_d']$ the set $\{(x_1, \ldots, x_d) \in R \cap Q\}$ can be reported. In the rest of the introduction we let $n = |R|$ be the number of points in 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 a query time Q. Further, we call the the maximum of the update time and the query time the time per operation.

For any $d \geq 1$ it is easy to implement a dictionary for $\mathbb{R}$ using a solution for the d-dimensional range reporting problem and this gives a lower bound of $\Omega(\log n)$ on the time per operation. To avoid this lower bound, we will use a natural variant of the comparison model: For $1 \leq i \leq d$ define $P_i = \{x_i \in \mathbb{R} | \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_d.(x_1, \ldots, x_d) \in R\}$. We then for each i, $1 \leq i \leq d$ maintain a linked list L_i containing the elements of P_i in order. When specifying a point $(x_1, \ldots, x_d) \in \mathbb{R}^d$ as a part of an update or query, we assume we for each i, $1 \leq i \leq d$, are given a pointer to the list element in L_i with the predecessor of x_i in P_i.

In this variant of the comparison model, we show (theorem 7.1 and theorem 8.1) that there exists a constant $\omega < 1$ such that for any $d \geq 2$ the d-dimensional range reporting problem can be solved with update time $O(\log^{\omega+d-2} n)$, query time $O((\log n / \log\log n)^{d-1})$ and space usage $O(n \log^{\omega+d-2} n)$. The model of computation is a unit cost RAM with word size $\Omega(\log n)$ bits. The time bounds are worst case and the solution is deterministic. It follows, that for $d \geq 3$ our result also holds in the standard comparison model.

For $d = 2$ Alstrup, Husfeldt and Rauhe [2] gave a lower bound of $\Omega(\log n / \log\log n)$ per operation. It follows that the time per operation for the 2-dimensional range reporting problem is $\Theta(\log n / \log\log n)$ and thus now completely understood. Actually, the lower bound of [2] is stronger: 1) it holds on a grid, 2) it holds for the amortized cost per operation, 3) it holds in the cell probe model with a word size poly-logarithmic in n. One could suspect, that our solution may actually have an optimal time per operation for any constant dimension $d \geq 3$. Proving or disproving this is a very interesting open problem.

For $d = 2$ we also consider queries of the restricted form $[x_1 \ldots x_1'] \times [-\infty \ldots x_2']$. Such queries are called *3-sided* whereas general queries are called *4-sided*. We show (theorem 6.1) that the 2-dimensional range reporting problem with 3-sides queries only, can be solved with update time $O(\log^{\omega} n)$, query time $O(\log n / \log\log n)$ and space usage $O(n)$.

1.1 Previous results

Orthogonal range searching including range reporting has been extensively studied during the last 30 years. For surveys see Agarwal and Erickson [1] and Chiang and Tamassia [12]. For books with earlier results see Mehlhorn [17] and Preparata and Shamos [22].

Besides 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 who have a weight between 60 and 80 kg. Willard [26] gives additional applications in databases.

With range trees [8, 15, 7, 9, 24] the 2-dimensional range reporting problem can be solved in time $O(\log^2 n)$ for updates and queries and with a space usage of $O(n \log n)$. For $d = 2$ Mehlhorn and Näher [18] improved range trees and reduced the time usage to $O(\log n \log\log n)$ and this result

holds even on a pointer machine. Recently, Mortensen [19] reduced the time usage to $O(\log n)$ and the space usage to $O(n \log n / \log\log n)$ on a RAM. Using standard techniques this solution can be extended to $d \geq 2$ dimensions giving update and query time $O(\log^{d-1} n)$ and space usage $O(n \log^{d-1} n / \log\log n)$. Until now this was the fastest known solution (even on a grid) and further no solution with space usage $o(n \log^{d-1} n / \log\log n)$ and poly-logarithmic time per operation was known.

McCreight [16] showed that the 2-dimensional range reporting problem with 3-sided queries only can be solved in time $O(\log n)$ for updates and queries and with a space usage of $O(n)$. Willard [27] improved the time usage to $O(\log n / \log\log n)$. It follows that we improve the update time for this problem without increasing the space usage.

1.2 Outline of paper

After this introduction we continue with preliminaries in section 2. In section 3, which is also a preliminary-like section, we consider different problems related to linked lists. In section 4 we give definitions and lemmas related to the 2-dimensional range reporting problem. In section 5 we define and analyze a pebble game played on a rooted tree. This game is used in section 6 where we develop our solution for the 2-dimensional range reporting problem with 3-sided queries only. In section 7 we use this solution to develop our solution to the general 2-dimensional range reporting problem. In section 8 we extend the solution of section 7 to higher dimensions. Finally, the paper concludes with acknowledgments in section 9.

2 Preliminaries

2.1 Basic preliminaries

For the rest of this paper we define $[i \dots j] = \{k \in \mathbb{Z} | i \leq k \leq j\}$. We define $Pow(X) = \{Y | Y \subseteq X\}$ to be the set of subsets of X. Wet let log denote the base 2 logarithm.

We assume N is fixed (but not constant) throughout the paper and that it is an upper bound on the number of points in the range reporting structures we design. We assume a word size of $\Omega(\log N)$ bits.

We assume we can use preprocessing time $O(N)$ for building tables of $O(N)$ words. As a corollary we assume we can compute simple functions from $[0 \dots O(N)]$ to $[0 \dots O(N)]$ in constant time. Removing these assumptions can be done using the global rebuilding techniques of Overmars [20].

For $0 < \epsilon < 1$ we define $L^\epsilon = [0 \dots \Theta(\log^\epsilon N)]$. We represent subsets of L^ϵ as bit vectors in a single word. We assume $\delta > 0$ is a sufficiently small constant fixed throughout the paper. We define $\mathcal{L} = L^\delta$ to be the set of *layers*. Layers are used in section 8 to support dimensions larger than two.

We order the elements of a linked list L such that if e and e' are elements in L and e is before e' in L and $e \neq e'$ then $e < e'$. Further we write $e \leq e'$ iff $e = e'$ or $e < e'$.

In the rest of this paper we measure space usage in bits. Further, all time bounds are worst case and all structures are deterministic if not otherwise noted.

2.2 A note on elements in data structures

We often consider data structures G which contain elements. We will not distinguish between a data structure and the set of elements it contains. It follows that $|G|$ is the number of elements in G and that $e \in G$ means that e is an element in G. In general, we do not distinguish between elements and pointers to elements.

A data structure G often assigns a set of named fields to the elements it contains. If G assigns its element a field with name `fieldname` and $e \in G$ we use the notation $G[e].\texttt{fieldname}$ to refer to this field. We write $e.\texttt{fieldname}$ if G is clear from the context. Observe that G may not be redundant since e may be inserted in several data structures.

2.3 Some theorems

The following theorem is proved in [19] as a corollary to a lemma by Dietz and Sleator [14, theorem 5]. It is useful for developing dynamic data structures with worst case rather than amortized performance. Note that $H_n = \sum_{i=1}^{n} 1/i$ and that $H_n = \Theta(\log n)$.

Theorem 2.1. *Suppose $c \geq 1$ 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 $|M| \geq 5c(1 + H_{n-1})$ 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 $|M| \leq 6c(1 + H_{n-1})$.*

The following theorem is showed by van Emde Boas et al. [23]:

Theorem 2.2. *There exists a data structure called a VEB which can maintain a collection of $n \leq U$ elements having keys in $[0 \dots O(U)]$ which uses space $O(U \log U)$ and supports updates as well predecessor and successor queries in time $O(\log \log U)$.*

Building on Beame and Fich[6], Anderson and Thorup[3] have showed the following theorem (actually, they showed a stronger theorem):

Theorem 2.3. *There exists a data structure called a BFAT which can maintain a collection of $n \leq U$ elements having keys in $[0 \ldots O(U)]$ which uses space $O(n \log U)$ and supports updates as well as predecessor and successor queries in time $O(\log^2 \log U)$.*

The following theorem is showed by Dietz and Sleator [14]:

Theorem 2.4. *There exists a 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 $e < e'$.*

2.4 Subadditive functions

Suppose f is a real-valued function defined on an interval $[s \ldots \infty[\subset \mathbb{R}$. We say that f is *subadditive* if for $x, y \geq s$:

$$f(x) + f(y) \leq f(x + y) + O(1) \tag{1}$$

Suppose $f(n)$ is a subadditive function which measures the size of a data structure G with n elements. It then follows from (1) that if we instead of storing the elements in G store them in k disjoint and non-empty structures with the same type as G, then the space usage increases by at most an additive term of $O(k)$. The following lemma gives a sufficient condition for a function to be subadditive:

Lemma 2.5. *Suppose f is a double differentiable real-valued function defined on an interval $[s \ldots \infty[\subset \mathbb{R}$. If $f'(x) \geq 0$ and $f''(x) \geq 0$ then f is subadditive.*

Proof. For a fixed value $y \geq s$ define $g(x) = f(x)+f(y)$ and $h(x) = f(x+y)+f(s)$. Since $f'(x) \geq 0$ it follows that $g(s) = f(s)+f(y)$ is less than or equal to $h(s) = f(2s)+f(y)$. Further, since $f''(x) \geq 0$ it follows that for $x \geq s$ we have $g'(x) = f'(x)$ is less than or equal to $h'(x) = f'(x + y)$. It follows that $f(x) \leq h(x)$ showing the lemma. □

And then we get:

Lemma 2.6. *Let $\epsilon > 0$ and $\alpha > 0$ be any constants and define the function f on the interval $[3 \ldots \infty[$ by:*

$$f(x) = \alpha x \log^{\epsilon}(x)$$

Then f is subadditive.

Proof. This follows from lemma 2.5 and the fact that:

$$f''(x) = \frac{\alpha(\epsilon + \ln(x) - 1)\left(\frac{\ln(x)}{\ln(2)}\right)^{\epsilon-2}}{x^3 \ln^2 2}$$

where ln is the natural logarithm. □

3 Lists

In this section we consider two problems related to linked lists: In section 3.1 we consider the online list labeling problem and in section 3.2 the colored predecessor problem.

3.1 The online list labeling problem

In this section we consider the online list labeling problem which we define as follows (other similar definitions are used in other papers): Let L be a linked list with at most n elements in which new elements can be inserted and from which existing elements can be deleted. We must then for each element $e \in L$ maintain an integer label $e.\texttt{label}$ of size $O(n)$ such that for $e, e' \in L$ we have:

$$e < e' \implies e.\texttt{label} < e'.\texttt{label} \tag{2}$$

An algorithm solving this problem is allowed to change the label of (relabel) elements when an element is inserted in or deleted from L. We require (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 showed by Willard [25]:

Theorem 3.1. *There exists an algorithm for the online list labeling problem with relabeling cost $O(\log^2 n)$, space usage $O(n \log n)$, which uses time $O(\log^2 n)$ for insertions and deletions.*

We now consider a variant of the online list labeling problem. In this variant the user can assign an id (used below) of $O(\log n)$ bits to an element when it is inserted in L. Further, we do not require the labels to be explicitly maintained in the elements of L. Instead, we require that we can calculate the label of a given element in constant time. Finally, when an element is inserted in or delete from L, the user must be given an array of triples describing the relabeling taking place because of the insertion or deletion. A triple (h, t_o, t_n) in this array means the the element with id h and current label t_o is given label t_n. When n is small we can pack several such triples into a single word. This allows us to relabel elements in sub constant time per element:

Lemma 3.2. *If $\log^4 n = o(\log N)$ then the described variant of the online list labeling problem can be solved in constant time per insertion in or deletion from L, a relabeling cost of $O(\log^3 n)$ and a space usage of $O(n \log n)$.*

Proof. For simplicity we will not describe how to handle deletion of elements.

We first observe, that since the relabeling cost is $O(\log^3 n)$ the table describing the relabeling uses $O(\log^4 n)$ bits and thus fits into a single word.

Fix $c = \Theta(\log^2 n)$ in theorem 2.1. We group the elements of L into blocks respecting the order of the elements. For every cth insertion in L we take a block with maximal number of elements and if it has at least $5c(1 + H_n)$ elements we split it into two blocks with at least $\lceil 2c(1 + H_n) \rceil$ elements in each. Theorem 2.1 ensures that no block will contain more than $6c(1 + H_{n-1})$ elements.

We now describe how to maintain the blocks such that insertion of elements and splitting of blocks can be done in constant time. We keep the elements of a block b as leafs in a tree with height 2 such that all leafs have depth 2. The root of the tree represents b. If b has less than $\lceil 4c(1 + H_{n-1} \rceil$ elements the root has a single child w and all elements of b are children of w. When we insert the $\lceil 4c(1+H_{n-1} \rceil$th element in b we create a new child v of the root. During the following $O(c(1 + H_{n-1}))$ insertions in b we walk through $\lceil 2c(1 + H_n) \rceil$ elements of b starting at the smallest and we make the elements we meet children of v. We adjust things such that we are finished when

v contains $5c(1 + H_n)$ elements. When b splits we make a new root node of v and a new root node of w representing the two blocks b is split into.

We now describe how we assign labels to the elements of L. Using the natural order on blocks, we use theorem 3.1 to assign labels of size $O(n/\log^3 n)$ to the blocks. The label assigned to an element $e \in L$ in block b consists of the label of b multiplied with $\Theta(\log^3 n)$ plus the rank e has in b. This assignment of labels clearly fulfills (2) and gives labels with a label size of $O(n)$ as claimed. Further, since we only split a block for every cth insertion, we only need to relabel a constant number of blocks at each insertion in L. Since there is $O(\log^3 n)$ elements in each block the relabeling cost is $O(\log^3 n)$ as claimed.

In each block we keep an array containing the name and ids of the elements stored in the block sorted according to the order of the elements. We observe that this array fits into a single word. It follows that we can find the table describing the relabeling in constant time.

Finally, since each element has a pointer to a parent which has a pointer the block the element is contained in, we can also find the label assigned to an element in L in constant time. □

3.2 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 at most n elements in which new elements can be inserted and from which existing elements can be deleted. Further, the user can assign a color to an element when it is inserted and also change the color of an element. Given an element $e \in L$ and a color c, we must then be able to find the predecessor of e in L which has color c. With two colors this problem is known as the incremental union-split-find problem. Dietz and Raman [13, lemma 4.2] have showed:

Lemma 3.3. *With two colors the colored predecessor problem can be solved in time $O(\log\log n)$ and space $O(n \log n)$.*

Using an extension of the techniques in [13] we can show the following lemma where we will only use the deterministic result (except for the proof of lemma 3.5):

Lemma 3.4. *The colored predecessor problem without restriction on the number of colors can be solved in space $O(n \log n)$ and in expected time $O(\log\log n)$ or deterministic time $O(\log^2 \log n)$.*

Proof. We first describe the randomized solution.

For each color c, we divide the elements of L with color c into blocks with at most $O(\log^3 n)$ elements respecting the order of the elements in L. We keep the elements of a block b in a standard balanced binary search tree respecting the order of the elements. Further, if there is more than one block with elements with color c we mark the first element in b.

We insert all elements of L into the structure of lemma 3.3 using marked and not-marked as the two colors. We use theorem 3.1 to assign a label to the marked elements. We create a VEB for each color and each marked element $e \in L$ with color c and label i is inserted in the VEB for color c at position i. Note that the space usage of a VEB can be brought down to linear since we allow randomization.

Answering queries: Suppose we want to find the predecessor of $e \in L$ with color c. We first use $O(\log\log n)$ time to find the marked predecessor e' of e in L. Assuming for brevity that this element exists, we use the label of e' to perform a predecessor query in the VEB for color c. This will identify the block with the element we are looking for. The element can then be found in time $O(\log\log n)$ using the binary search tree in the block. We need to be able to compare two arbitrary elements in L in constant time to do this, but this can be done using theorem 2.4.

Inserting a new element in L: First, using the same techniques as when answering a query, we can easily insert the new element in the search tree of the appropriate block. The problem is to handle the case where blocks become too big. We do this the following way: Every time we have inserted $O(\log^2 n)$ elements in L, we take the largest block and split it into two blocks which differ in size by at most 1. This will mark a new element, and thus the structure of theorem 3.1 requires us to renumber $O(\log^2 n)$ marked elements. We distribute this work over the following $O(\log^2 n)$ insertions in L updating the VEBs for the different colors at the same time.

Changing the color of an element: First, we delete the element from the block it is located in. If the element is the only one in the block we remove the block. If the element is not the only one and if it is marked, we take the successor of the element in the block and mark this instead reusing the label of the old element. After this deletion in the block we just insert the element with the new color as described above.

Deleting an element: Instead of deleting an element we just mark it as deleted. We then use global rebuilding to ensure that the structure is never more than half filled up with deleted elements.

This concludes our description for the randomized case. The deterministic result can be obtained in the same way by using BFATs instead of VEBs. □

Consider the variant of the colored predecessor problem where each element in L can have an arbitrary number of colors and where we can add a color to an element, remove a color from an element and get the predecessor of an element with a given color. Further, we can delete an element if it has no colors. Let m be equal to n plus the number of pairs (e, c) where $e \in L$ has color c. We then have the following lemma and again we will only be using the deterministic result:

Lemma 3.5. *The described variant of the colored predecessor problem can be solved in space $O(m \log m)$ and in expected time $O(\log \log m)$ or deterministic time $O(\log^2 \log m)$.*

Proof. We introduce black as an additional color assuming no elements in L are black. We maintain a list L' using lemma 3.4 such that the sublist of L' consisting of the black elements is equal to L. Further, if e and e' are black elements in L' such that e is the black predecessor of e', then we keep between e and e' an element with color c for each color c that e has in L.

To answer a query we first make a predecessor query for the given color and then a predecessor query for black. Adding a color c to an element can be done by adding an element to L' with color c. Removing a color c from an element can be done by making a successor query for c and then remove the found element from L'. □

We now describe how lemma 3.5 can be seen as a variant of dynamic fractional cascading [18, 13] for readers familiar with that. The variant is as follows: The catalog graph must be directed and if there is an edge from a set M to a set M' then unlike normal (dynamic) fractional cascading M' must be a subset of M. Further, a node can have at most constant indegree and when walking around in the catalog graph we can only follow edges in their direction. On the other hand, unlike normal dynamic fractional cascading we have no restriction on the outdegree of a node. Lemma 3.5 can be used to implement this variant as follows: We assign each set a unique color and if there is an edge from a set M to a set M' and $e \in M'$ then we give the corresponding element $e \in M$ color c and link e and e' together with pointers.

4 Range reporting

4.1 Definitions

In this section we define what it means for a a range reporting structure R to have type $R^{\tau}_{t_x:n_x,t_y:n_y}$ where n_x and n_y are integers, $\tau \in \{3,4\}$ and $t_x, t_y \in \{d, s, s', s''\}$. To simplify the description we let z range over $\{x, y\}$ in the following two paragraphs.

R must have an *z-axis* $R.L_z$. If $t_z = d$ we say that the z-axis is *dynamic* and if $t_z = s$, $t_z = s'$ or $t_z = s''$ we say that the z-axis is *static*. If the z-axis is dynamic it is a linked list with $O(n_z)$ elements. If the z-axis is static it is a subset of $[0 \ldots \Theta(n_z)]$ and this subset is *not* required to be explicitly maintained.

R contains elements $e \in R$ which are also called points. A point $e \in R$ has a z-coordinate $e.\mathtt{z} \in R.L_z$ and a layer $e.\mathtt{layer} \in \mathcal{L}$. For given $z \in R.L_z$ define $M = \{e \in R | e.\mathtt{z} = z\}$. If $t_z = s$ or $t_z = d$ we require $|M| = 1$. If further $t_z = d$ and $e \in M$ then $e.\mathtt{z}$ should have a pointer to e. Let $w \in \{x, y\} \setminus \{z\}$ be the "opposite coordinate" of z. If $t_z = s'$ we require that $e, e' \in M$ and $e.\mathtt{layer} = e'.\mathtt{layer}$ implies $e = e'$. If $t_z = s''$ we require that $e, e' \in M$, $e.\mathtt{layer} = e'.\mathtt{layer}$ and $e.\mathtt{w} = e'.\mathtt{w}$ implies $e = e'$.

We support updates in the form of insertion of new elements in R and deletion of existing elements from R.

For $x_1, x_2 \in R.L_x$ and $y_1, y_2 \in R.L_y$ and $L \subseteq \mathcal{L}$ we support a query (x_1, x_2, y_1, y_2, L) which must report the set $\{e \in R | x_1 \leq e.\mathtt{x} \leq x_2 \wedge y_1 \leq e.\mathtt{y} \leq y_2 \wedge e.\mathtt{layer} \in L\}$. If $\tau = 3$ we require that y_1 in such a query can be assumed to be the minimal element of $R.L_y$.

We say R has query time Q if queries can be answered in time $Q + O(r)$ where r is the number of reported points. We say R has *performance* (U, Q, S, t) if it supports $O(t)$ updates in time $O(U)$, queries in time $O(Q)$ and has space usage $O(S)$. The space usage of R is only allowed to depend on the type of R and not on the number of points actually stored in R. We require t to be selected such that all updates can be described by $O(1)$ words and further the points themselves and not just pointers to the points must be given in this word. We write (U, Q, S) as an abbreviation for $(U, Q, S, 1)$.

Finally, we require that R can contain $\Omega(\log^{\epsilon} N)$ points for a sufficiently small constant $\epsilon > 0$. A consequence of this requirement is, that the asymptotic space usage of R is not influenced by the fact that we must store the layer for each point.

4.2 Some lemmas

The following lemma is proved in section 4.3 by reformulating the construction from [19, section 3]:

Lemma 4.1. *Suppose $u \leq n$ and there exists a structure with type $R^{\tau}_{s'':n,s':u/|\mathcal{L}|}$ and performance (U, Q, S). Then there exists a structure with type $R^{\tau}_{s'':n,s'':u/|\mathcal{L}|}$ and performance $(U \log\log n, Q \log\log n, Sn \log\log n)$.*

The following lemma can be used to get a structure to plug into lemma 4.1:

Lemma 4.2. *Suppose there exists a structure with type $R^{\tau}_{d:u,s':u/|\mathcal{L}|}$ and performance (U, Q, S). Then, for any n, $u \leq n$ there exists a structure with type $R^{\tau}_{s'':n,s':u/|\mathcal{L}|}$ and performance $(U + \log^2 \log n, Q + \log^2 \log n, S + u \log n)$.*

Proof. Let R be the given structure with type $R^{\tau}_{d:u,s':u/|\mathcal{L}|}$ and R' be the structure with type $R^{\tau}_{s'':n,s':u/|\mathcal{L}|}$ we want to design.

We insert each element $e \in R'$ in a BFAT at position $u|\mathcal{L}|e.\mathtt{x} + |\mathcal{L}|e.\mathtt{y} + e.\mathtt{layer}$. Using this BFAT we link all elements in R' together according to this position. This list can be used as $R.L_x$ and the lemma follows. □

Instead of using lemma 4.1 directly we will use the following corollary.

Lemma 4.3. *Suppose $u \leq n$ and there exists a structure with type $R^{\tau}_{d:u,s'':u/(|\mathcal{L}|\log^3 n)}$ and performance (U, Q, S). Then there exists a structure with type $R^{\tau}_{d:n,s'':u/(|\mathcal{L}|\log^3 n)}$ and performance $((U + \log^2 \log n) \log \log n, (Q + \log^2 \log n) \log \log n, (Sn/u) \log \log n + n \log n)$.*

Proof. Let R be the structure with type $R^{\tau}_{d:n,s'':u/(|\mathcal{L}|\log^3 n)}$ we want to design. For simplicity, we will not describe how to handle deletion of points.

First we take the given structure and restrict it to have type $R^{\tau}_{d:u/\log^3 n,s':u/(|\mathcal{L}|\log^3 n)}$. We plug this restricted structure into lemma 4.2 obtaining a structure with type $R^{\tau}_{s'':n/u,s':u/(|\mathcal{L}|\log^3 n)}$ and performance $(U+\log^2 \log n, Q+\log^2 \log n, S+u/\log^2 n)$. We then plug this structure into lemma 4.1 obtaining a structure T with type $R^{\tau}_{s'':n/u,s'':u/(|\mathcal{L}|\log^3 n)}$ and performance $((U+\log^2 \log n) \log \log n, (Q+\log^2 \log n) \log \log n, (S + u/\log^2 n)(n/u) \log \log n)$.

We group the elements of $R.L_x$ into blocks of size $O(u)$ using theorem 2.1: For every $\Theta(u/\log n)$ insertion in $R.L_x$ we take a largest blocks and split it in two if it has $\Omega(u)$ elements. We maintain a labeling of the blocks using theorem 3.1. If a block b with assigned label x contains a point with y-coordinate y and layer l we insert b in T at x-coordinate x, y-coordinate y and layer l.

Each time a block is split we need to relabel $O(\log^2 n)$ blocks and each relabeling requires $O(u/\log^3 n)$ updates in T. This gives a total of $O(u/\log n)$ updates for each block split and it follows that we only need $O(1)$ updates in T for each insertion in R.

Finally, we maintain in each block b a structure $b.R$ with the given type $R^{\tau}_{d:u,s'':u/(|\mathcal{L}|\log^3 n)}$. All points in b are stored in $b.R$ and we note that the total space used by these structures is $O(Sn/u + n \log n)$.

Suppose we are given a query (x_1, x_2, y_1, y_2, L). 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 $b_1.M$ and then a local query in $b_2.M$ (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, each y-coordinate $y \in R.L_y$ and each layer $l \in \mathcal{L}$ maintains the set of the points in b with y-coordinate y and layer l. □

Intuitively, lemma 4.3 says the following: Assume we have a structure with a dynamic but short x-axis and an essentially equally short y-axis. Then we can extended the x-axis to be arbitrary long paying only a small penalty in the update and query time. The following lemma can be used to get a structure to plug into lemma 4.3. Many variants of this lemma can be made.

Lemma 4.4. *Suppose $\log^4 u = o(\log N)$, $v \leq u$ and there exists a structure with type $R^{\tau}_{s:u,s'':v}$ ($R^{\tau}_{s':v,s:u}$) and performance $(U, Q, S, \log^3 u)$. Then there exists a structure with type $R^{\tau}_{d:u,s'':v}$ ($R^{\tau}_{s':v,d:u}$) and performance (U, Q, S).*

Proof. We show how to construct a structure R' with type $R^{\tau}_{d:u,s'':v}$ from a structure R with type $R^{\tau}_{s:u,s'':v}$. The other construction is similar.

We maintain a labeling of the elements of $R'.L_x$ using lemma 3.2. As id for $x \in R'.L_x$ we use the point x represents. For each element $x \in R'.L_x$ this assigns a label $x.\texttt{label}$ of size at most $O(u)$. We then insert each element $e \in R'$ in R at x-coordinate $e.\texttt{x}.\texttt{label}$, y-coordinate $e.\texttt{y}$ and layer $e.\texttt{layer}$. □

4.3 Proof of lemma 4.1

This section is devoted to the proof of lemma 4.1. We obtain the proof by reformulating the proof from [19, section 3].

We now define what it means for a data structure G to have type $(n, Y, \mathcal{Y})$ where n is an integer and $\mathcal{Y} \subseteq Pow(Y)$. G contains elements $e \in G$ where $e.\texttt{x}$, $0 \leq e.\texttt{x} < n$ is the *position* of e and $e.\texttt{y} \in Y$ is the *height* of e. Two different elements in G are not allowed to have both the same position and height implying $|G| \leq n|Y|$. We are allowed to update G by inserting a new element or by deleting an existing element. Further, for $0 \leq i, j < n$ and $q \in \mathcal{Y}$ we can ask the query $\texttt{report}(G, i, j, q)$ which must report the set $\{e \in G \,|\, i \leq e.\texttt{x} \leq j \wedge e.\texttt{y} \in q\}$. We say G has query time $O(Q)$ if this set can be found in time $O(Q + r)$ where r is the size of the set. In this section we show:

Lemma 4.5. *Suppose we have a data structure G' with type $(n, Y, \mathcal{Y})$ and the restriction:*

$$e, e' \in G' \wedge e.\texttt{y} = e'.\texttt{y} \implies e = e' \tag{3}$$

Suppose further G' uses space $O(S)$, 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) which uses space $O(Sn \log\log n)$, has query time $O(Q \log\log n)$ and update time $O(U \log\log n)$.

Lemma 4.1 is a corollary to lemma 4.5:

Proof of lemma 4.1. Let R' be the given structure with type $R^{\tau}_{s'':n,s':u/|\mathcal{L}|}$ and R be the structure with type $R^{\tau}_{s'':n,s'':u/|\mathcal{L}|}$ we want to design. Similarly, let G' be the structure which should be given to lemma 4.5 and G be the structure provided by lemma 4.5.

Select $Y = [0 \ldots O(u/|\mathcal{L}|)] \times \mathcal{L}$. We insert each point $e_r \in R$ as an element $e_g \in G$ selecting $e_g.\texttt{x} = e_r.\texttt{x}$ and $e_g.\texttt{y} = (e_r.\texttt{y}, e_r.\texttt{layer})$. Similarly, we insert the element $e_g \in G'$ as a point $e_r \in R'$. If $e_g.\texttt{y} = (y, l)$ we select $e_r.\texttt{x} = e_g.\texttt{x}$, $e_r.\texttt{y} = y$ and $e.\texttt{layer} = l$.

Select $\mathcal{Y} = \{[y_1 \ldots y_2] \subseteq Y\} \times Pow(\mathcal{L})$ where y_1 must be 0 if $\tau = 3$. Suppose we are given a query (x_1, x_2, y_1, y_2, L) in R. We transform this into a query $\texttt{report}(G, x_1, x_2, ([y_1 \ldots y_2], L))$. Similarly, suppose we are given a query $\texttt{report}(G', x_1, x_2, ([y_1 \ldots y_2], L))$. This query is transformed into the query (x_1, x_2, y_1, y_2, L) in R'. □

The rest of this section is devoted to the proof of lemma 4.5.

We create a VEB for each element $y \in Y$. Each element $e \in G$ is inserted at position $e.\texttt{x}$ in the VEB for height $e.\texttt{y}$. Using these VEBs we link all elements with the same height together in order according to their positions. This will not use too much space, and insertions and deletions can be performed in time $O(\log\log n)$ per operation.

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 $[0 \ldots n-1]$, and $y \in Y$ is the height of the triple. We maintain S_n such that $(e, e.\texttt{x}, e.\texttt{y}) \in S_n$ iff $e \in G$. We support a special $\texttt{reportsub}(S_n, i, j, q)$ 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 $i \leq e.\texttt{x} \leq j$ and $e.\texttt{y} = y$ then $\texttt{reportsub}(S_n, i, j, q)$ contains at least one such e.

We say S_n has query time $O(Q')$ if the set $\texttt{reportsub}(S_n, i, j, q)$ can be found in time $O(Q' + r)$ where r is the size of the set. We then also require the following property for S_n:

3. S_n has query time $O(Q)$.

A $\texttt{report}(G, i, j, q)$ query can then be answered in the following way: 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 $m \geq 0$. 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 $S_n.\texttt{min}$ ($S_n.\texttt{max}$) indexed by Y. For each $y \in Y$ we store the triple $(e, x, y) \in S_n$ with minimal (maximal) value of x in $S_n.\texttt{min}[y]$ ($S_n.\texttt{max}[y]$) if any. We also keep an array $S_n.\texttt{bottom}$ 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 $(e, i, y) \in S_n \setminus (S_n.\texttt{min} \cup S_n.\texttt{max})$ as $(e, i \bmod \sqrt{n}, y)$ in $S_n.\texttt{bottom}[i \operatorname{div} \sqrt{n}]$. 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 exactly one triple $(e, x, y) \in S_n$ with height y, we store (e, i, y) in $S_n.\texttt{top}$. We note that in this case e is stored in both $S_n.\texttt{bottom}$ and in $S_n.\texttt{top}$. If $S_n.\texttt{bottom}[i]$ contains more than one triple with height y we store (e', i, y) in $S_n.\texttt{top}$ where e' is a pointer to the recursive structure in $S_n.\texttt{bottom}[i]$.

Inserting a triple (e, x, y) in S_n: If S_n contains at most one triple with height y, we just update $S_n.\texttt{min}[y]$ and $S_n.\texttt{max}[y]$ and we are done. Else, we first check if (e, x, y) should go into $S_n.\texttt{min}[y]$ ($S_n.\texttt{max}[y]$) and if this is the case we interchange (e, x, y) and the triple in $S_n.\texttt{min}[y]$ ($S_n.\texttt{max}[y]$). Let T be the structure in $S_n.\texttt{bottom}[x \operatorname{div} \sqrt{n}]$. 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 $S_n.\texttt{top}$. If $m = 2$, there is a triple $(e', x \operatorname{div} \sqrt{n}, y)$ in $S_n.\texttt{top}$, 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 $S_n.\texttt{top}$ then T has at most two triples with height y and the update in T is performed by just accessing $T.\texttt{min}$ and $T.\texttt{max}$. 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 $(e, x, y) = S_n.\texttt{min}[y]$ ($(e, x, y) = S_n.\texttt{max}[y]$) we just update $S_n.\texttt{min}[y]$ ($S_n.\texttt{max}[y]$) and we are done. Else, let T be the structure in $S_n.\texttt{bottom}[x \operatorname{div} \sqrt{n}]$. 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 $S_n.\texttt{top}$. 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 $S_n.\texttt{min}[y]$ and $S_n.\texttt{max}[y]$ 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 $(e, x, y) = S_n.\texttt{min}[y]$. Let $(e', t, y) = S_n.\texttt{top}.\texttt{min}[y]$, $(e'', l, y) = S_n.\texttt{bottom}[t].\texttt{min}[y]$ and $i = l + t\sqrt{n}$. Then (e'', i, y) is the triple in $S_n \setminus (S_n.\texttt{min} \cup S_n.\texttt{max})$ 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 $S_n.\texttt{min}[y]$ to (e'', i, y). Case 2: In this case $(e, x, y) = S_n.\texttt{max}[y]$ and this case is handled in a symmetric way to case 1. Case 3: In this case

(e, x, y) is not equal to $S_n.\texttt{min}[y]$ or $S_n.\texttt{max}[y]$. The case is handled in way similar to insertions: Let T be the structure in $S_n.\texttt{bottom}[x \operatorname{div} \sqrt{n}]$. 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 $S_n.\texttt{top}$ 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 $S_n.\texttt{top}$. 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 $S_n.\texttt{min}$ ($S_n.\texttt{max}$) maintain a structure $S_n.\texttt{min}'$ ($S_n.\texttt{max}'$) with the type of the structure given to lemma 4.5. We maintain $S_n.\texttt{min}'$ ($S_n.\texttt{max}'$) such that $S_n.\texttt{min}'$ ($S_n.\texttt{max}'$) contains an element e with position x, height y and a value e' iff $S_n.\texttt{min}$ ($S_n.\texttt{max}$) contains the triple (e', x, y).

Answering a $\texttt{reportsub}(S_n, i, j, q)$ query: If $i > j$ or $S_n = \emptyset$ we report nothing. Else, let M be the values of the element reported by the two queries $\texttt{report}(S_n.\texttt{min}', i, j, q)$ and $\texttt{report}(S_n.\texttt{max}', i, j, q)$ (a point may be reported by both queries and in this case it should only appear one time in M). We then iteratively for each element $e \in M$ where e points to a recursive structure T in our induction on n, replace $e \in M$ with e' and e'' assuming $T.\texttt{min}[y] = (e', x', y')$ and $T.\texttt{max}[y] = (e'', x'', y'')$. We observe that the time we spend on performing these replacements is proportional to $|M|$. After this, we report the elements of 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 $\texttt{reportsub}(S_n.\texttt{bottom}[i \operatorname{div} \sqrt{n}], i \bmod \sqrt{n}, j \bmod \sqrt{n}, q)$ query. Finally if $i \operatorname{div} \sqrt{n} \neq j \operatorname{div} \sqrt{n}$ we split the query into the three queries $\texttt{reportsub}(S_n.\texttt{bottom}[i \operatorname{div} \sqrt{n}], i \bmod \sqrt{n}, \sqrt{n} - 1, q)$, $\texttt{reportsub}(S_n.\texttt{top}, i \operatorname{div} \sqrt{n}+1, j \operatorname{div} \sqrt{n}-1, q)$ and $\texttt{reportsub}(S_n.\texttt{bottom}[j \operatorname{div} \sqrt{n}], 0, j \bmod \sqrt{n}, q)$. We note, that the first and the last of these three queries are answered by just looking at the `min`, $\texttt{min}'$, `min` and $\texttt{max}'$ fields in the recursive structure.

To analyze the space usage of S_n we need the following lemma:

Lemma 4.6. *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`, $\texttt{min}'$ and $\texttt{max}'$ structures we need and the lemma then bounds the space usage of S_n to $O(Sn \log \log n)$ 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 lemma 4.5.

5 A pebble game

In this section we describe a combinatorial pebble game. The reader is recommended to read this section together with the proof of lemma 6.2 where the game is applied the first time.

5.1 A pebble game with red and green pebbles

The pebble game is played on a infinite rooted tree T without leafs and is parameterized over a parameter g. There exists 6 kind of pebbles: Insert, delete, push, poll, heavy and light pebbles. The first four of these are called red pebbles and the last two are called green pebbles. When the game starts, all nodes in T contain g heavy pebbles. In each round the player either:

b1. Add an insert or delete pebble to the root of T.

or select a node $v \in T$ and a red pebble p in v and apply one of the following items:

b2. If p is an insert or delete pebble then move p to a child of v.

b3. If p is a push pebble then take a light pebble p' from v and move p and p' to the same child of v.

b4. If p is a poll pebble and v' is a child of v with a light pebble p' then interchange p and p'. If no such child v' exists then convert p to a heavy pebble.

Next, the player can apply the following items any number of times in the nodes $v \in T$ involved in the applied item b1 to b4:

b5. Convert an insert pebble in v to a light pebble creating a push pebble in v.

b6. Remove a delete and a light pebble from v creating a poll pebble in v.

After this, the player must in each node $v \in T$ apply each of the following items in the given order where each item must be applied exhaustively:

b7. If v contains a push and a poll pebble remove both from v.

b8. If v contains a heavy and a poll pebble convert the poll pebble to a heavy pebble.

b9. If v contains a heavy and a push pebble remove both from v.

b10. If v contains a heavy and an insert or delete pebble p remove p.

Item 5 in the following lemma shows that there is always a light pebble p' to take in item b3:

Lemma 5.1. *Before each round we have:*

1. *No node contains both push and poll pebbles.*
2. *If a node contains a heavy pebble then it contains no red pebble.*
3. *If the player plays such that before each round each node contains at least one green pebble, then all nodes descendant to a node with a heavy pebble contain only heavy pebbles.*
4. *If g' is the number of green, o is the number of poll and u is the number of push pebbles in a node, then $g = g' + o - u$.*

5. All nodes contain at least as many light pebbles as push pebbles.

Proof. We show the lemma by induction on the game. For the base case we observe that all properties 1 to 5 are fulfilled when the game starts. For the inductive case we give a separate argument for each property:

1: This follows from b7 and from the fact that item b8 to b10 do not introduce push or poll pebbles.

2: This follows from b8 to b10.

3: Observe first, that the set of pebbles in a node v is only changed if v or vs parent (if any) contains a red pebble. It follows, from property 2 and the induction hypothesis, that the set of pebbles in a node which has a parent with a heavy pebble is never changed. Second observe, that the only item that can introduce a heavy pebble in a node v which does not already have a heavy pebble is item b4. This happens only if no child of v has a light pebble. Since all nodes by assumptions contains at least one green pebble it follows that all children of v contains a heavy pebble. It follows from property 2 that they contain no red pebbles and then they must consist of only heavy pebbles.

4: All items b1 to b10 are seen to preserve this.

5: We need to check the items b1 to b10 which adds a push pebble to or remove a light pebble from a node. b3: Here a push and a light pebble are moved from one node to another. b4: Here a light pebble may be removed from a node but a poll pebble is added which in item b7 will remove a push pebble if such one exists. b5: Here a push pebble is a added but a light pebble is also added. b6: Here a light pebble is removed, but the node contains a poll pebble and thus not any push pebbles. □

5.2 A strategy for the player

In this section we remove some of the freedom from the game in section 5.1 by describing when to apply which of the items b1 to b4. Each time one of the items b1 to b4 is applied, the remaining items b5 to b10 must still be applied as described in section 5.1. We require that all nodes in T have the same degree $d \geq 2$ and in addition to d and g, we parameterize the game over a parameter β. We then repeat the following items:

m1. Add (one at a time) β insert or delete pebbles to the root of T using item b1.

m2. Set v to the root of T.

m3. Take β red pebbles from v (or as many as there is if fewer) and move them (one at a time) down to the children of v using item b2 to b4.

m4. Set v to the child of v which has most red pebbles

m5. If there is more than $((d-1)/d)\beta$ red pebbles in v go to m3.

We have:

Lemma 5.2. *At the beginning of item m1 no set of siblings in T contains more than $(d-1)\beta$ red pebbles.*

Proof. We first observe, that at the start of item m1 the root contains no red pebbles, and it thus contains at most β red pebbles at the start of item m2. Now consider a maximal set S of siblings in T which contains at most $(d-1)\beta$ red pebbles and which receives at most β red pebbles from their

common parent in item m3. Let α be the number of red pebbles in the nodes of S after this and let v be the node in S with most red pebbles selected in item m4. It follows that $\alpha \leq d\beta$. Suppose that we do not go to item m3 in item m5. Then there cannot be more than $(d-1)\beta$ red pebbles among the nodes in S and we are done. Suppose next that we go to item m3 in item m5. Then there must be at least α/d red pebbles in v. After the push in item m3 in the following iteration, there will thus be at most $\alpha - \alpha/d = \alpha(1-1/d) \leq d\beta(1-1/d) = (d-1)\beta$ red pebbles among the nodes in S as required. □

As a simple corollary we get:

Lemma 5.3. *During the game no set of siblings in T contains more than $d\beta$ red pebbles.*

Lemma 5.4. *If $g > d\beta$ then all nodes descendant to a node with heavy pebbles contains only heavy pebbles.*

Proof. Lemma 5.3 gives that a node contains at most $d\beta$ red pebbles. Since $g > d\beta$ it follows from lemma 5.1 property 4 that all nodes contain at least one green pebble. The lemma then follows from lemma 5.1 property 3. □

Lemma 5.5. *During the game no set of siblings contains more than $d(g+2\beta)$ pebbles.*

Proof. Let g' be the number of green, o the number of poll and u the number of push pebbles in a set of siblings. Property 4 in lemma 5.1 then gives $dg = g' + o - u$. This implies $g' \leq dg + u$. Lemma 5.3 gives $u \leq d\beta$ and thus we have $g' \leq d(g+\beta)$. Finally an additional application of lemma 5.3 gives that the total number of pebbles is bounded by $d(g+2\beta)$ as desired. □

Observe that the worst case situation handled by lemma 5.5 is the case where a set of siblings is completely filled with push pebbles.

6 Structures supporting 3-sided queries

This main purpose of this section is to prove the following theorem which for $n = N$ shows the second result from the introduction.

Theorem 6.1. *For any constant $\rho > 0$ there exists a structure with type $R^3_{d:n,d:n}$ and performance $(\log^2 \log N + \log n / \log^{1/6-\delta/6-\rho} N, |\mathcal{L}| \log \log N + \log n / \log \log N, n \log n)$.*

Our proof of the theorem builds on priority search trees of McCreight [16], the extension of these trees as described by Willard [27], combined with the use of buffers in the style of Brodal [10].

6.1 Priority search trees

A priority search tree is a structure with type $R^3_{d:n,d:n}$ and performance $(\log n, \log n, n \log n)$ where all points must have the same layer. To illustrate the basic idea in priority search trees, we now describe how to design a structure R with type $R^3_{s:n,s:n}$ and performance $(\log n, \log n, n \log n)$ again requiring all points to have the same layer. The reader should refer to [16] for more details.

We span a complete binary tree T over the x-axis of R and we store the points of R in the nodes of T maintaining the following invariants:

i1. Each node in T contains at most one point.

i2. If $v \in T$ is an ancestor of $w \in T$ and w contains a point p_w then v contains a point p_v and further $p_v.\mathtt{y} < p_w.\mathtt{y}$.

i3. If $v \in T$ contains a point p_v then $p_v.\mathtt{x}$ is in a leaf descendant to v.

Observe that these invariants give a unique way to store the points of R in T.

Answering a query $(x_1, x_2, 0, y_2, \{l\})$ where l is the single layer: Let $M \subseteq T$ be the set of nodes which is an ancestor to a leaf with x_1 or x_2. Extend M with all children of nodes in M and observe that $|M| = O(\log n)$. We then first report all points from the nodes of M that should be reported. Let M' be the set of nodes which are not in M, has a parent from M with a reported point and has a descendant leaf between the leafs with x_1 and x_2. If M' is non-empty we recursively report the points from the nodes of M' in the same way. The reader should verify that this indeed reports the points that should be reported and that the query time is $O(\log n)$.

Inserting a point: This is done by inserting the point in the root of T so we describe how to insert a point p in an arbitrary node $v \in T$: If v does not contain a point we just store p in v. Else, let p' be the point in v. If $p'.\mathtt{y} < p.\mathtt{y}$ we recursively insert p in the child of v which has $p.\mathtt{x}$ in a descendant leaf. If $p.\mathtt{y} < p'.\mathtt{y}$ we store p in v instead of p' and recursively insert p' in the child of v which has $p'.\mathtt{x}$ in a descendant leaf.

Deleting a point: This is done by *polling* the node the point is stored in so we describe how to poll an arbitrary node $v \in T$: If v has no child with a point, we just remove the point from v. Else, let v' be the child of v with a point p' such that $p'.\mathtt{y}$ is minimized. We then store p' in v and recursively poll v'.

6.2 Extensions to priority search trees

In this section we describe three extensions to priority search trees which we will all apply below.

The first extension is to increase the degree of T as done in [27]. Updates and queries easily generalize to this though for the extension to be efficient we must be able to handle all the points in the children of a given node fast.

The second extension is to support $|\mathcal{L}|$ layers: We do this by conceptually keeping a separate priority search tree for each layer. Updates are then performed in the tree for the relevant layer. We store the conceptually separate trees as one tree allowing one point for each layer in each node. Assuming a relevant set of fast operations for all the points in a given node, this allows us to support layers efficiently.

The third extension is to drop invariant i1 and store multiple points (from each layer) in each node. A similar thing was done by Arge, Samoladas and Vitter [4]. Again, updates and queries can be adjusted to also handle this and again, assuming a relevant set of fast operations for all the points in a given node, we can handle this efficiently.

6.3 Small universe

The proof of the following lemma uses priority search trees with all extensions described in section 6.2 together with techniques of [10] to obtain a fast structure with static axises for a small universe.

Lemma 6.2. *For any constant $\rho > 0$ and for given d and u define $t = (\log^{1-\delta-\rho} N)/(d^2 \log^3 u)$. If $t \geq 1$ there exists a structure with type $R^3_{s':u/|\mathcal{L}|,s:u}$ and performance $(1, \log u/\log d, u \log u, t)$. The structure requires a global lookup table with $2^{O(\log^{1-\rho} N)}$ entries which can be computed in linear time.*

Proof. First assume there is only one layer. We span a complete binary tree T with degree d over the x-axis and we play the pebble game from section 5.2 on T. We let each light pebble represent a point. Together with the light pebbles, T is a priority search tree with the first and third extension described in section 6.2.

Updates are now basically performed by adding β insert and delete pebbles each representing a single update to the root of T and then execute the loop in item m3 to m5. To get a solution with worst case time behavior we modify this as follows: We only add t pebbles at a time to the root of T and each time we do this, we execute a constant number of the items in the loop in item m3 to m5. Since T has height $O(\log u/\log d)$, which is also $O(\log u)$, the following condition ensures that we do not add too many pebbles to the root of T before we empty it again:

$$t \log u \leq \beta \tag{4}$$

In item m3 the insert and delete pebbles goes to the child determined by the x-coordinate of the point they represent. When a delete pebble finds the light pebble it deletes, it applies item b6. When an insert pebble finds the place it should be (determined by its y-coordinate or by the appearance of a delete pebble deleting it) it converts itself to a light pebble by applying item b5.

To support $|\mathcal{L}|$ layer we do as in section 6.2: Each node of T is divided into $|\mathcal{L}|$ layers and the pebble game is played separately on each layer.

We store as a part of each insert and light pebble the point it represents including x and y coordinates. We observe that this takes $O(\log u)$ 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. Push and poll pebbles do not have any associated information so it follows that any pebble can be stored using $O(\log u)$ bits.

We store all the pebbles (from all layers) in a maximal set of siblings from T using $O(\log^{1-\rho} N)$ bits in a single word. Using the global lookup table this allows us to perform item m3 and the

corresponding updates in item b5 to b10 in constant time. Further, during queries, this allows us to handle all layers in parallel. 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. Lemma 5.5 gives that a set of siblings contains $O(d(g+2\beta)(\log^\delta N))$ pebbles and since each pebble uses $O(\log u)$ bits we can store the pebbles in the described way if:

$$d(g+2\beta)(\log^\delta N)\log u \leq \log^{1-\rho} N \tag{5}$$

Next, we require for each layer, that the number of red pebbles on a root-path is less than g. This ensures, that in each node and layer without a heavy pebble there is always a light pebble which is not deleted by a red pebble above it. Using lemma 5.3 this requirement is fulfilled if:

$$cd\beta \log u \leq g \tag{6}$$

for some constant $c \geq 1$. We select:

$$\begin{aligned} g &= (\log^{1-\delta-\rho} N)/(3d\log u) \\ \beta &= (\log^{1-\delta-\rho} N)/(3cd^2\log^2 u) \end{aligned} \tag{7}$$

and verify that with this selection (5) and (6) are satisfied. Further, (4) together with (7) gives our requirement for t.

Queries can now be answered in time $O(\log u/\log d)$ as in section 6.1 with one remark: We must ensure that we never report a point which is deleted by a delete pebble above it. This is done as follows: 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. There is always less than g such pebbles for each layer so this set can be described by $O(g\log u\log^\delta N) = O((\log^{1-\rho} N)/d)$ bits. Using this set and the global lookup table we can avoid reporting deleted points. □

In section 2 we made the general assumption that we can create lookup tables with $O(N)$ entries if they can be computed in linear time. The reason that we mention the lookup table explicitly in lemma 6.2 will be made clear in the proof of lemma 7.5. As a corollary to lemma 6.2 we get:

Lemma 6.3. *For any constant $\rho > 0$ and $u = O(2^{((\log^{1-\delta-\rho} N)/d^2)^{1/6}})$ there exists a structure with type $R^3_{s':u/|\mathcal{L}|,d:u}$ and performance $(1, \log u/\log d, u\log u)$.*

Proof. Select $t = \log^3 u$ in lemma 6.2. This gives a structure with type $R^3_{s':u/|\mathcal{L}|,s:u}$ and performance $(1, \log u/\log d, u\log u, \log^3 u)$. Further we get the restriction on u and d that $\log^3 u \leq (\log^{1-\delta-\rho} N)/(d^2\log^3 u)$. This can also be written as $\log^6 u \leq (\log^{1-\delta-\rho} N)/d^2$ which is obtained if $u = O(2^{((\log^{1-\delta-\rho} N)/d^2)^{1/6}})$. Finally, inserting the obtained structure into lemma 4.4 gives the desired result. □

6.4 Full universe

The following lemma shows how to build a structure for a full universe from a structure for a small universe:

Lemma 6.4. *Suppose $u \geq |\mathcal{L}|^2$ and we have a structure with type $R^3_{s':u/|\mathcal{L}|,d:u}$ and performance $(U, Q, u\log u)$. Then for $n \geq u$ we can make a structure with type $R^3_{s':n/|\mathcal{L}|,d:n}$ and performance $((U+\log^2\log n)\log n/\log u, (Q+\log^2\log n)\log n/\log u, n\log n)$.*

Proof. Let R be the structure with type $R^3_{s':n/|\mathcal{L}|,d:n}$ we want to design. We create a priority search tree with the first and second extension described in section 6.2 and we give T degree $\Theta(u/|\mathcal{L}|)$ and thus height $O(\log n/\log u)$. In each internal node $v \in T$ we keep a structure $v.R$ with type $R^3_{s':u/|\mathcal{L}|,d:u}$ containing the points stored in the children of v. We observe that $v.R.L_y$ is a linked list with the points stored in the children of v sorted according to the y-coordinate. For each layer, we link the elements of $v.R.L_y$ representing points with that layer together using the colored predecessor structure of lemma 3.4 with the layers as colors.

We color each node in T with a unique color and color each element $e \in R.L_y$ with the color of the unique node $v \in T$ such that there is an $e' \in v.R.L_y$ such that e and e' represent the same point. Further, we link e and e' together with pointers. We maintain the colored predecessor structure of lemma 3.4 on $R.L_y$ using the assigned colors.

Suppose we are given a query (x_1, x_2, y_1, y_2, L) where y_1 is the minimal element of $R.L_y$. Let $M \subseteq T$ be the set of nodes which is an ancestor to a leaf with x_1 or x_2 and note that $|M| = O(\log n/\log u)$. For each node $v \in M$ we locate the boundary of the query in $v.R.L_y$ by making a colored predecessor query from $y_2 \in R.L_y$ for the color of v. We then follow the pointer from the found element to the corresponding element in $v.R.L_y$. For each $v \in M$ we can then find the points stored in v and the children of v that should be reported by making a query in $v.R$. It follows that the total time spend so far is $O((Q + \log^2 \log n) \log n/\log u + r)$ where r is the number of reported points. Finally, the remaining points can be found in constant time per point: The idea is to walk through $v.R.L_y$ from the beginning for relevant nodes $v \in T$. We stop when we meet a point with a higher y-coordinate than y_2. This can be determined if we maintain the structure of theorem 2.4 on $R.L_y$.

Updates can be performed like in priority search trees so this concludes our proof of the lemma. □

We now immediately get the following structure for a full universe:

Lemma 6.5. *For any constant $\rho > 0$ there exists a structure with type $R^3_{s':n/|\mathcal{L}|,d:n}$ and performance $(\log^2 \log n(1 + \log n/\log^{1/6-\delta/6-\rho} N), \log n/\log\log N, n \log n)$.*

Proof. Select $d = \log^{3\rho} N$ in lemma 6.3. For $u = O(2^{(\log^{1-\delta-6\rho} N)^{1/6}})$ we then get a structure with type $R^3_{s':u/|\mathcal{L}|,d:u}$ and performance $(1, \log u/\log\log N, u \log u)$ proving the lemma for $n = O(u)$. For $n = \Omega(u)$ we plug the obtained structure into lemma 6.4. □

Finally we have:

Proof of theorem 6.1. Let R be the structure with type $R^3_{d:n,d:n}$ we want to design. For simplicity, we will not describe how to handle deletion of points.

We group the elements of $R.L_x$ into blocks with $O(\log^4 N)$ elements. For every $O(\log^3 N)$ insertion in $R.L_x$ we take a largest block and split it in two if it has $\Omega(\log^4 N)$ elements. Theorem 2.1 ensures that all blocks will have size $O(\log^4 N)$.

For each block and for each layer we keep a priority search tree [16] with the points in that block and layer. We maintain a labeling of the blocks using theorem 3.1. Finally, we maintain a single structure T from lemma 6.5 with type $R^3_{s':n/\log^4 N,d:n|\mathcal{L}|/\log^4 N}$ and performance $(\log^2 \log n(1+\log n/\log^{1/6-\delta/6-\rho} N), \log n/\log\log N, n \log n)$

Assume a block b has been assigned label m. For each layer $l \in \mathcal{L}$ we take the point stored in the root of the priority search tree stored in b for layer l and insert it in T at x-coordinate m and layer l.

Answering a query: Suppose first that the query does not cross a block boundary. Then the query can be answered in time $O(|\mathcal{L}| \log\log N)$ by performing a query in the priority search trees for each of the layers selected by query. Suppose next that the query crosses a block boundary. We then first perform a local query in the block at the beginning and in the block at the end of the query interval again spending time $O(|\mathcal{L}| \log\log n)$. The remaining points can be found by a query in T using time $O(\log n / \log\log N)$.

Inserting a new point: First we insert the point in a block using time $O(\log\log N)$ and if necessary we update T spending time $O(\log^2 \log n(1 + \log n / \log^{1/6-\delta/6-\rho} N))$.

Handling splits of blocks: The structure of theorem 3.1 requires us to renumber $O(\log^2 N)$ blocks on each split. This in turn requires us to perform $O(\log^{2+\delta} N)$ updates in T. We perform these updates during the $O(\log^3 N)$ insertions there is until we may have to split a block again. □

7 Structures supporting 4-sided queries

This section is devoted to the proof of the following theorem which for $n = N$ shows the first result from the introduction for $d = 2$:

Theorem 7.1. *Let $\epsilon > 0$ be any constant and define $\omega = 7/8 + \delta/8 + \epsilon$. Then there exists a structure with type $R^4_{d:n,d:n}$ and performance $(\log^\omega N, \log N/\log\log N, n\log^{1+\omega} N)$.*

7.1 From 3-sided to 4-sided

In this section we will 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 [11] and Overmars [21].

Assume we for any m, $m \leq n$, have a structure with type $R^3_{s'':n,s'':m}$, and performance (U, Q, S_m) and that S_m is a subadditive function of m. We will now describe how we can make a structure R with type $R^4_{s'':n,s'':n}$ and performance $(U\log n, Q, S_n \log n)$.

We span a complete binary tree T over $R.L_y$. Each node $v \in T$ spans a subinterval I of $R.L_y$ where $|I| \leq n$. We store in v two secondary structures $v.R_1$ and $v.R_2$ both with type $R^3_{s'':n,s'':|I|}$ and both containing all points in I. The structure $v.R_1$ ($v.R_2$) should support queries of the form (x_1, x_2, y_1, y_2, L) where y_1 (y_2) is the first (last) element in I.

Inserting (deleting) a point p in (from) R: This involves inserting (deleting) p in (from) two secondary structures in each of the $O(\log n)$ ancestors of the leaf with p using time $O(U \log n)$.

Answering a query (x_1, x_2, y_1, y_2, L): For a node $v \in T$ let v_1 and v_2 be the children of v such that all leafs below v_1 comes before the leafs below v_2 in $R.L_y$. Fix $v \in T$ to be the node which is an ancestor of both y_1 and y_2 such that neither v_1 nor v_2 is an ancestor of both y_1 and y_2. The query can then be answered by performing a single query in $v.R_1$ and a single query in $v.R_2$ giving a query time of $O(Q)$.

7.2 Small universe

The following lemma, which is similar to lemma 6.2 gives a structure for a small universe:

Lemma 7.2. *For any constant $\rho > 0$ and for given u define $t = (\log^{1-\delta-\rho} N)/\log^4 u$. If $t \geq 1$ there exists a structure with type $R^4_{s'':u,s:u}$ and performance $(1, \log u, u\log^2 u, t)$. The structure requires a global lookup table with $2^{O(\log^{1-\rho} N)}$ entries which can be computed in linear time.*

Proof. We show the lemma by combining the constructions in lemma 6.2 and in section 7.1.

Like in section 7.1 we span a complete binary tree T over the y-axis. On T we play the pebble game from section 5.2 with $g = 0$. As in the proof of lemma 6.2 the user performs $O(t)$ updates by adding $O(t)$ pebbles to the root of T in item m1 and again each added pebble must be an insert or delete pebble representing the update. In point m3 we move the pebbles down to the children determined by the y-coordinates of the points they represent. Unlike lemma 6.2 we never apply item b5 or b6 on the pebbles in T and it follows that T will only contain insert and delete pebbles. When a pebble comes down to a leaf of T we just let it disappear.

Like in section 7.1 we have two secondary structures in each internal node of T. Each time a pebble enters a node $v \in T$, a copy of the pebble is also given to the appropriate secondary structure in v. For the secondary structures, we use the structure of lemma 6.2 with $d = 2$. We select the same value of β for our game on T as in the game on the secondary structures, namely the value given by (7).

We can now answer queries as in done in section 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.

During the time β pebbles are inserted in the root of T we must go through item m3 to m5 like in the proof of lemma 6.2. Further, each time we execute item m3 in T we must also execute the loop in item m3 to m5 in a constant number of secondary structures. Since T as well as the trees in all the secondary structures have height $O(\log u)$ the following condition ensures that we do not add too many pebbles to the root of T before we empty it again:

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

Together with (7) this gives our requirement for t.

Finally, each point is stored in at most $O(\log u)$ pebbles in T and the secondary structures giving a total space usage of $O(u \log^2 u)$. □

As a corollary we get the following lemma which is similar to lemma 6.3:

Lemma 7.3. *For any constant $\rho > 0$ and $u = O(2^{(\log^{1-\delta-\rho} N)^{1/7}})$ there exists a structure with type $R^4_{d:u,s'':u}$ and performance $(1, \log u, u \log^2 u)$. The structure requires a global lookup table with $2^{O(\log^{1-\rho} N)}$ entries which can be computed in linear time.*

Proof. Select $t = \log^3 u$ in lemma 7.2 and exchange the two axises. This gives a structure with type $R^4_{s:u,s'':u}$ and performance $(1, \log u, u \log^2 u, \log^3 u)$. Further we get the restriction on u that $\log^3 u \leq (\log^{1-\delta-\rho} N)/\log^4 u$. This can also be written as $\log^7 u \leq \log^{1-\delta-\rho} N$ which is obtained if $u = O(2^{(\log^{1-\delta-\rho} N)^{1/7}})$. Finally, inserting the obtained structure into lemma 4.4 gives the desired result. □

As another corollary we get:

Lemma 7.4. *For any constant $\rho > 0$ and $u = O(2^{(\log^{1-\delta-\rho} N)^{1/7}})$ there exists a structure with type $R^4_{d:n,s'':u/(|\mathcal{L}| \log^3 n)}$ and performance $(\log^3 \log n, (\log u + \log^2 \log n) \log \log n, n \log \log n \log^2 u + n \log n)$. The structure requires a global lookup table with $2^{O(\log^{1-\rho} N)}$ entries which can be computed in linear time.*

Proof. This is obtained by plugging the structure of lemma 7.3 into lemma 4.3 □

The following lemma shows that we can get a somewhat dynamic y-axis in lemma 7.4. We need to look back into several proofs in order to prove this lemma:

Lemma 7.5. *The structure in lemma 7.4 can be given a dynamic y-axis where we allow any number of points with a given y-coordinate. In this structure, it takes time $2^{O(\log^{1-\rho} N)}$ to insert a new element on the y-axis. Further, the space usage is increased with an additive term of $2^{O(\log^{1-\rho} N)}$.*

Proof. First, we modify the construction in the proof of lemma 6.2 and 7.2 as follows: We assign each element on the x-axis a unique and arbitrary id. Further, instead of storing x-coordinates directly in the pebbles, we store the ids of the x-coordinates. This gives a dynamic x-axis in the following sense: We can insert a new element on the x-axis by assigning the new element an unused id and then recompute the global lookup table. This follows from the fact that the global lookup table is the only way by which coordinates of points are compared.

Looking into the proofs of lemma 7.3 and lemma 4.4 we see, that we in the same sense get a dynamic y-axis in lemma 7.3.

Finally, looking into the proofs of lemma 7.5, lemma 7.4, lemma 4.1 and lemma 4.3 we see, that we in the same sense get a dynamic y-axis in lemma 7.5. This is because all points given to the structure provided by lemma 7.5 are saved in instances of the structure given to the lemma. The key point is, that all instances of the structure given to the lemma can share the same lookup table. □

7.3 Full universe

Finally we have:

Proof of theorem 7.1. Let R be the structure with type $R^4_{d:n,d:n}$ we want to design. As in [19, section 6] we span a WBB tree (see the proof of theorem 7.1 for references) T over $R.L_y$. We specify the degree of T in a moment. See [19, section 5] or Arge and Vitter [5] for a description of WBB trees. We keep in each node $v \in T$ a linked list $v.L$ where we save all the points located in the leafs descendant to v sorted according to their x-coordinate. Let $v \in T$ be an internal node. We give each child of v a unique color. For each $e \in v.L$ there exists 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'. Further, we keep the elements of $v.L$ in a colored predecessor structure of lemma 3.4 using the assigned colors.

We now sketch how to answer a query (x_1, x_2, y_1, y_2, L) in R. Details can be found in [19, section 6]. Let v_r be the root of T and observe that $v_r.L$ contains all points in R sorted according to their x-coordinate. It follows that each of x_1 and x_2 identifies an element in $v_r.L$ corresponding to the query interval on the x-axis. We now proceed down T from v_r to the leaf determined by y_1. While we do this, we use the colored predecessor structures to maintain the query interval on the x-axis in $v.L$ for the nodes $v \in T$ we meet. Beginning in some node of T, we must report the elements inside this interval that has a relevant set of colors and layers. After this, we proceed from from v_r down to the leaf determined by y_2 in the same way and we are done.

We select a *division level* of T and all nodes of T below this level are said to be in the *lower* part of T and all nodes above are said to be in the *upper* part of T. Let $0 < \rho < 1$ be a constant to be determined in a moment. If $n = O(2^{\log^{1-\rho} N})$ we select the division level to be at the root of T such that the upper part of T is empty. Else, we select the division level such that each tree in the lower part of T has $\Theta(2^{\log^{1-\rho} N})$ elements.

We give each node v in the lower part of T degree $\Theta(1)$. We link all elements in $v.L$ with the same layer together using lemma 3.4 with layers as colors. Ignoring $\log\log N$ factors, this allows us to perform updates in the lower part of T in time $O(\log^{1-\rho} N)$. Queries can be answered in time $O(\log^{1-\rho+\delta} N)$ by performing at most a constant number of queries for each layer in each level of the lower part of T. Finally, the total space used in the lower part of T is $O(n \log^{2-\rho} N)$.

We give each node v in the upper part of T degree $\Theta(u/\log^{3+\delta} N)$ where $u = 2^{(\log^{1-\delta-\rho} N)^{1/7}}$. It follows that the height of the upper part of T becomes $O(\log N/\log u)$ which can be written as $O(\log^{1-(1-\delta-\rho)/7} N)$. We store the elements of $v.L$ in the structure of lemma 7.4 using the colors as y-coordinates. We observe that for each child v' of v, $v'.L$ contains $\Omega(2^{\log^{1-\rho} N})$ elements. It follows, that we can handle splits of nodes in the upper part of T using lemma 7.5 without extra time or space overhead. Ignoring $\log\log N$ factors, the total update time for the upper part of T is $O(\log N/\log u)$. The total space usage in the upper part of T ignoring $\log\log N$ factors is $O((\log N/\log u)(n\log^2 u + n\log N))$ and this can also be written as $O(n\log^{2-(1-\delta-\rho)/7} N)$.

The query time in the upper part of T is $O((\log N/\log u)\log u \log\log N)$. This can also be written as $O(\log N \log\log N)$ which is an $O(\log^2 \log N)$ factor too high. We fix this problem in the following way: For each node v in the upper part of T which is on a level divisible by $\log^2 \log N$, we keep two structures from theorem 6.1. In these structures we save the elements in $v.L$ in the style of section 7.1. We then modify the way we answer queries as follows: When we in our proceeding down in T meet a node v on a level divisible by $\log^2 \log N$ *and* when we should report points from $v.L$ we stop. We then perform a query in one of the structures from theorem 6.1 kept in v in the style of section 7.1. After this, we do not need to proceed further down in T. This modification cuts of a $O(\log^2 \log N)$ factor in the query time in the upper part of T and we conclude that we only need to use time $O(\log N/\log\log N)$ in answering queries. Further, since each point of R is inserted in at most $\log^2 \log N$ structures of theorem 6.1, the time and space usage is not increased by our modification except for constant factors.

Finally, we select the value of ρ that minimizes the update time. This is done when the update time in the lower and upper part of T is identical. Ignoring $\log\log N$ factors this is the case when $1-(1-\delta-\rho)/7 = 1-\rho$. This can also be written as $\rho = (1-\delta)/8$. Ignoring $\log\log N$ factors it follows that in both the upper and lower part of T the time usage becomes $O(\log^{7/8+\delta/8} N)$ and the space usage becomes $O(n\log^{1+7/8+\delta/8} N)$ which proves the lemma.

□

8 Higher dimensions

This section is devoted to the proof of the following theorem which shows the first result from the introduction for $d \geq 3$:

Theorem 8.1. *Let $d \geq 3$ and $\epsilon > 0$ be any constants and define $\omega = 7/8 + \epsilon$. Then there exists a solution for the d-dimensional range reporting problem with update time $O(\log^{d-2+\omega} N)$, query time $O((\log N/\log\log N)^{d-1})$ and space usage $O(N \log^{d-1+\omega} N)$ bits.*

The proof of the theorem is divided into two parts. First, we in section 8.1 describe a black box transformation transforming one structure to another. Second, we in section 8.2 apply this transformation $d-2$ times to the structure from theorem 7.1 in order to prove theorem 8.1 for a given d.

8.1 A black box transformation

We now define what it means for a data structure G to have type $(\mathcal{Q}, \epsilon)$. Each element $e \in G$ has a point $e.\mathtt{p}$ and a layer $e.\mathtt{layer} \in L^\epsilon$ (see section 2.1 for definition of L^ϵ). Observe that we use a slightly different definition of layers than in the preceding sections. G can be updated by inserting a new or deleting an existing element. $\mathcal{Q}$ must be a set of predicates defined on points. G must support a query $(q, L) \in \mathcal{Q} \times Pow(L^\epsilon)$ with answer $\{e \in G \,|\, q(e.\mathtt{p}) \wedge e.\mathtt{layer} \in L\}$. If G supports updates in time $O(U)$, queries in time $O(Q + r)$ where r is the number of reported elements and uses space $O(S(n))$ where $n = |G|$ we say G has *performance* $(U, Q, S(n))$.

Lemma 8.2. *Suppose there exists a structure with type $(\mathcal{Q}, \epsilon)$ and performance $(U, Q, S(n))$, that $U = \Omega(\log\log N)$, $Q = \Omega(\log\log N)$, and that S is subadditive. Suppose further f is a function from points to $\mathbb{R}$ which can be evaluated in constant time. Then there exists a structure with type $(\mathcal{Q}', \epsilon/2)$ where $\mathcal{Q}'$ is the set of predicates q' that can be written as $q'(p) = i \leq f(p) \leq j \wedge q(p)$ for $q \in \mathcal{Q}$ and $i, j \in \mathbb{R}$. Further the structure has performance $(U \log N/\log\log N, Q \log N/\log\log N, S(n) \log N/\log\log N)$.*

Proof. Let G' be the structure with type $(\mathcal{Q}', \epsilon/2)$ we want to design. We save each element $e \in G'$ in the leafs of a WBB tree (see eg. [19, section 5]) X with degree $\Theta(\log^{\epsilon/2} N)$ using $f(e.\mathtt{p})$ as key. We store the keys in each internal node $x \in X$ in a standard balanced binary search tree. Further, we assign each child of x a unique layer in $L^{\epsilon/2}$. We keep in x a secondary structure $x.G$ with type $(\mathcal{Q}, \epsilon)$. Let $e \in G'$ be an element which is in a leaf descendant to a child of x with layer l. We then store e in $x.G$ with $(x.G)[e].\mathtt{layer} = l + (G'[e].\mathtt{layer})\Theta(\log^{\epsilon/2} N)$.

Inserting (deleting) an element $e \in G'$ requires inserting (deleting) e in (from) $O(\log N/\log\log N)$ secondary structures in the nodes of X and this takes time $O(U \log N/\log\log N)$ as required. By assumption $U = \Omega(\log\log N)$ so this term also pays for the binary search we need to perform in the nodes of X.

Suppose we are given a query $(q', L') \in \mathcal{Q}' \times Pow(L^{\epsilon/2})$ where $q'(p) = i \leq f(p) \leq j \wedge q(p)$. The interval $\{r \in \mathbb{R} | i \leq r \leq j\}$ identifies a set $M \subseteq X \times Pow(L^{\epsilon/2})$ such that $|M| = O(\log N/\log\log N)$ and such that we get the elements in the answer by for each $(x, L) \in M$ to perform the query $(q, \{l_1 + l_2\Theta(\log^{\epsilon/2} N) \in L^\epsilon | l_1 \in L \wedge l_2 \in L'\})$ in $x.G$.

To support splitting of nodes in X we need to change the structure slightly. When a key r in node $x \in X$ with layer l and parent x' is marked in the terminology of [19, section 5], an additional secondary structure $x.G'$ is created and x is given an additional unique layer l' by x'. During the time r is marked we move the elements $e \in x.G$ for which $f(e.\mathtt{p}) > r$ from $x.G$ to $x.G'$ using

$(x.G')[e].\texttt{layer} = (x.G)[e].\texttt{layer}$. Define $l'' = l' + G'[e].\texttt{layer}\Theta(\log^{\epsilon/2} N)$. For each moved element e we set $(x'.G)[e].\texttt{layer} = l''$ if $e \in x'.G$ and $(x'.G')[e].\texttt{layer} = l''$ if $e \in x'.G'$ by removing and then reinserting e. □

8.2 A structure for higher dimensions

Finally we have:

Proof of theorem 8.1. Let R be the structure from theorem 7.1. With the terminology from section 8.1, R can be viewed as having type $(\mathcal{Q}, \delta)$ and performance $(\log^{\omega} N, \log N / \log\log N, n \log^{1+\omega} N)$. Here, $\mathcal{Q}$ contains all predicates q of the form $q(e) = x_1 \leq e.\mathtt{x} \leq x_2 \wedge y_1 \leq e.\mathtt{y} \leq y_2$ for $x_1, x_2 \in R.L_x$ and $y_1, y_2 \in R.L_y$.

Applying lemma 8.2 $d-2$ times to R almost proves the theorem. But only almost, because the first two coordinates need special treatment. This problem can be fixed by the use of lemma 3.5. Details will be given in the final version of the paper. □

9 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 [10] and for helpful discussions in connection with this work.

References

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

[2] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *39th Annual Symposium on Foundations of Computer Science: proceedings: November 8–11, 1998, Palo Alto, California*, pages 534–543. IEEE Computer Society Press, 1998.

[3] Arne A. 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: Portland, Oregon, May 21–23, [2000]*, pages 335–342. ACM Press, 2000.

[4] 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.

[5] Lars Arge and Jeffrey S. Vitter. Optimal dynamic interval management in external memory. In *37th Annual Symposium on Foundations of Computer Science: October 14–16, 1996*, pages 560–569. IEEE Computer Society Press, 1996.

[6] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing: Atlanta, Georgia, May 1–4, 1999*, pages 295–304. ACM Press, 1999.

[7] Jon L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–250, June 1978.

[8] 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.

[9] 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.

[10] Gerth S. Brodal. Predecessor queries in dynamic integer sets. In *Proc. 14th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1200, pages 21–32. Springer, 1997.

[11] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Computing*, 15(3):703–724, August 1986.

[12] Yi-Jen Chiang and Roberto Tamassia. Dynamic algorithms in computational geometry. Technical Report CS-91-24, Department of Computer Science, Brown University, March 1991.

[13] 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.

[14] 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.

[15] 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.

[16] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[17] Kurt Mehlhorn. *Data Structures and Algorithms: 3. Multidimensional Searching and Computational Geometry*. Springer, 1984.

[18] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[19] Christian W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[20] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[21] Mark H. Overmars. Efficient data structures for range searching on a grid. *Journal of Algorithms*, 9(2):254–275, June 1988.

[22] Franco P. Preparata and Michael I. Shamos. *Computational Geometry*. Springer, New York, 1985.

[23] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[24] 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.

[25] 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.

[26] 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.

[27] 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.