Optimal dynamic vertical ray shooting in rectilinear planar subdivisions.

Yoav Giyora *   Haim Kaplan †

Abstract

In this paper we consider the dynamic vertical ray shooting problem, that is the task of maintaining a dynamic set S of n non intersecting horizontal line segments in the plane subject to a query that reports the first segment in S intersecting a vertical ray from a query point. We develop a linear-size structure that supports queries, insertions and deletions in $O(\log n)$ worst-case time. Our structure works in the comparison model and uses a *RAM*.

1 Introduction

In this paper we consider data structures for the dynamic vertical ray shooting problem. In this problem we maintain a dynamic set S of n non intersecting horizontal line segments in the plane such that we can efficiently report the segment in S immediately above a given point. The vertical ray shooting problem is in fact a version of the *dynamic rectilinear planar point location* problem. In particular, given a subdivision of the plane by horizontal and vertical line segments, our data structure allows to find the rectangle containing a query point. The *dynamic rectilinear planar point location* problem is a special case of the general *dynamic planar point location* problem in which segments are not restricted to be horizontal or vertical. Obtaining a linear space data structure with logarithmic query and update time for *dynamic planar point location* is a central open question in algorithms and computational geometry. Although the restriction of segments to be horizontal is strong, an optimal algorithm for this special case was not known prior to our work.

We present a data structure in the *RAM* model of computation, that requires linear space and supports updates and queries in $O(\log n)$ worst-case time. Our data structure does not make any assumptions on the segments. That is, we manipulate the segments only by comparisons. In this sense our result is optimal, since by an easy reduction from sorting, at least one of the operations takes $\Omega(\log n)$ time.

Specifically, we present three data structures for the vertical ray shooting problem. The first two structures work in the pointer-machine model of computation. The first structure requires $O(n \log^{\epsilon} n)$ space, and supports queries in $O(\frac{1}{\epsilon} \log n)$ worst-case time, and updates in $O(\frac{1}{\epsilon} \log^{1+\epsilon} n)$ worst-case time. The second structure requires linear space, and supports queries in $O(\frac{1}{\epsilon} \log^{1+\epsilon} n)$ worst-case time and updates in $O(\frac{1}{\epsilon} \log n)$ worst-case time, where $\epsilon > 0$ is as small as we want. For the third structure we use the *RAM* model of computation, and achieve a linear size structure that supports both updates and queries in $O(\log n)$ worst-case time.

All our data structures use a segment tree with fan-out $O(\log^{\epsilon} n)$. In the first two structures we use dynamic fractional cascading, extended appropriately for our needs. To obtain logarithmic query and update time we generalize the Van Emde Boas structure [21]. This generalization allows to exploit word-level parallelism to speed up the fractional cascading query. A similar generalization has been made by Mortensen [17]. Finally, we reduce the space to linear, using a technique of Baumgarten *et al.* [4], in which we store all the segments in an *interval tree*, where only a carefully chosen subset of the segments is stored in a *segment tree*.

Our data structure extends, using standard techniques (multi-dimensional segment tree), to solve the *dynamic vertical ray shooting* problem in $\mathbb{R}^d$, for $d > 2$. In this problem, you maintain a dynamic set of hyperplanes orthogonal to the x_d-axis such that you can query for the first hyperplane intersecting a vertical ray from a given query point. We pay an overhead of a logarithmic factor in space, update time, and query time per dimension.

We denote an open horizontal segment from (x_s, y) to (x_e, y) by a triple (x_s, x_e, y).

Applications: Our data structure allows to obtain optimal (in the comparison model) solutions to the following two problems.

1) The three-dimensional layers-of-maxima problem: A point $p \in \mathbb{R}^3$ *dominates* another point $q \in \mathbb{R}^3$ if each coordinate of p is larger than the corresponding coordinate of q. Given a set S of n points in $\mathbb{R}^3$, the *maximum points* are those that are not dominated by any point in S. We define the maximum points in S to be *layer* 1 of S. We then delete the maximum

*Department of Computer Science, Tel-Aviv University, ISRAEL.

†Department of Computer Science, Tel-Aviv University, ISRAEL.

points from S and the maximum points among the remaining points are *layer* 2 of S. We continue to assign a layer to each point until S is empty. In the three-dimensional layers-of-maxima problem we want to efficiently compute the layer of each point.

Buchsbaum and Goodrich [5] present an algorithm to solve the three-dimensional layers-of-maxima problem. We use our data structure to implement their algorithm in linear space as follows. We sort the points by their *z-coordinate*. Then we sweep the space from $+\infty$ with an hyperplane parallel to the x-y plane. When the sweep plane reaches a point p, it assigns p to its layer. Let S_i consist of the projections on the sweep plane of the points that were already processed and assigned to layer i. Let M_i be the *maximum points* of S_i. Each set M_i form a staircase, where M_i *dominates* M_{i+1} in the plane. When we process a point p we find the staircase M_j which is immediately above p. The layer of p is $j+1$, and furthermore, p should be added to M_{j+1}, and points in M_{j+1} dominated by p should be removed. We implement this algorithm by maintaining the horizontal segments of all the staircases M_j in our ray shooting data structure H. We locate the staircase immediately above p by a ray shooting query from p. If we also maintain each M_j as a list, then we can easily find all the points in M_j to delete, as they are consecutive. To analyze the performance of this structure we note that for each point $p \in S$ we perform three operations on H, and on a list representing M_j. It follows that our implementation requires linear space and $O(n \log n)$ time. This answers the open problem of [5], as their implementation requires $O(n \log n / \log \log n)$ space.

2) The retroactive successor problem: Retroactive data structures were introduced by Demaine, Iacono, and Langerman [10]. A (fully) retroactive data structure allows to perform an update or a query at any given time. In the *retroactive successor* problem, a key with time stamp t can be inserted or deleted at time t. A query with a pair (t, k) should return the successor of k at time t.

The data structure of Demaine et al. [10] supports updates and queries in $O(\log^2 n)$ time. We use our data structure to obtain an optimal solution to the *retroactive successor* problem as follows. We represent each key of the retroactive structure with a segment in an optimal vertical ray shooting structure H. To insert a key y at time t_s, we insert the segment (t_s, ∞, y) to H. We implement a deletion of a key y at time t_e, by removing the segment (t_s, ∞, y) from H, and inserting the segment (t_s, t_e, y) instead. To return the successor of y at time t, we perform a query with a point (t, y). It follows that the *retroactive successor* problem can be implemented in linear space, to support updates and queries in $O(\log n)$ time, where n is the total number of updates performed on the retroactive structure.

Previous results: Mehlhorn and Näher [16] present a data structure for the *vertical ray shooting problem* that requires $O(n \log n)$ space, supports queries in $O(\log n \log \log n)$ worst-case time, and updates in $O(\log n \log \log n)$ amortized time. The best result we know with logarithmic query time is by Kaplan, Molad, and Tarjan [15], and requires $O(\log^2 n)$ time for updates.

Agarwal, Arge, and Yi [1] present an optimal solution to the *interval stabbing-max* problem. In this problem we maintain a dynamic set S of n intervals on the line, where each interval $s \in S$ has a weight $w(s)$. A query reports the interval in S with maximum weight containing a given point $q \in \mathbb{R}$. Assume without loss of generality that $w(s) > 0$ for all $s \in S$, and that we want to find the interval in S with minimum (rather than maximum) weight containing q. If we think of $w(s)$ as the *y-coordinate* of a segment s' in $\mathbb{R}^2$ that corresponds to the interval s, then the *interval stabbing-max* problem reduces to the vertical ray shooting problem, restricted to query points on the *x-axis*. Our implementation has the same performance as the implementation in [1], but we allow the query to be any point in $\mathbb{R}^2$ rather than only in $\mathbb{R}$.

Mortensen [17] presented a fully dynamic data structure for the two dimensional orthogonal range reporting problem and for the two dimensional line segment intersection reporting problem. The fully dynamic two dimensional line segment intersection reporting problem is the task of maintaining a set of horizontal line segments in $\mathbb{R}^2$ subject to a query that reports all the segments that intersect an vertical query line segment. The implementation presented in [17] is in the comparison model, and uses a RAM. It requires $O(n \log n / \log \log n)$ space, supports insertions and deletions in $O(\log n)$ worst-case time, and queries in $O(\log n + k)$ worst-case time, where k is the size of the output. The line segment intersection reporting problem can be viewed as the *decision* version of our problem. We borrow several techniques from Mortensen [17] in this paper.

There has been a lot of work on the more general *dynamic planar point location* problem. Baumgarten *et al.* [4] describe a data structure for the vertical ray shooting version of the general problem, and therefore also solves our problem. This solution requires linear space, supports queries in $O(\log n \log \log n)$ worst-case time, insertions in $O(\log n \log \log n)$ amortized time, and deletions in $O(\log^2 n)$ amortized time. Arge, Brodal, and Georgiadis [2] also describe a data structure for the general problem (which solves our problem). They provided two linear space implementations. Their first implementation works in the pointer machine model, supports queries in $O(\log n)$ worst-case time, insertions in $O(\log^{1+\epsilon} n)$ amortized time, and deletions in $O(\log^{2+\epsilon} n)$ amortized time. Their second implementation is randomized, requires a *RAM*, and supports queries in $O(\log n)$ time, insertions in $O(\log n \log^{1+\epsilon} \log n)$ amortized time, and deletions in $O(\log^2 n / \log\log n)$ amortized time.

Results in other papers, summarized in Table 1 below, assume that the segments of S form a particular subdivision of the plane. In these cases, a segment can be inserted to S only if it divides a single facet of the subdivision. If we try to solve the vertical ray shooting problem using one of these structures, by defining a subdivision based on the input set of segments, we are bound to fail, since an inserted segment can be contained in many facets. These previous works consider different families of planar subdivisions.

The structure of the rest of the paper: We present our data structure in four steps. Each step improves the previous one by using an additional technique. In the last step we obtain an optimal solution to the problem (i.e., it uses linear space, and supports all operations in logarithmic time). The first step presents two simple solutions to the vertical ray shooting problem in the pointer machine model. The running time in these solutions have a $\log^{\epsilon} n$ overhead factor over the optimal solution. The overhead of the first solution is in the update time, and the overhead of the second solution is in the query time. In the second step we use the unit cost *RAM* model (with a word size logarithmic in n), and show how to get rid of the $\log^{\epsilon} n$ overhead factor to achieve logarithmic bounds for all operations. In the third step we sketch a space saving technique. With this technique we improve the space requirement of our three implementations. In the full version of this paper [13] we add the forth step, in which we explain how to deamortize all our results. Before we present our implementations, we review several data structures. This review starts with a preliminary discussion about *segment trees* and *interval trees*, generalized to apply to a tree with a large fan-out rather than a binary tree. We continue by providing a detailed review for weight-balance B-trees and fractional cascading, since we use these structures throughout this paper.

2 Preliminaries

In this section we review the techniques that we use and improve, to achieve our results. In Section 2.1 we review *segment trees* and *interval trees*. These trees form the basic building blocks of our structures. In fact we use a version of these trees with large fan-out implemented as weight-balance B-trees. We review weight-balance B-trees in Section 2.2. We also formalize the splitting theorem (Theorem 2.1) for weight-balance B-trees, that we need and have not found stated explicitly. In Section 2.3 we review *Fractional Cascading*. We also add a new lemma (Lemma 2.1) that allows us to add and remove edges from a weight-balance B-tree T, while using Fractional Cascading with T as the underlying graph.

2.1 Segment trees and interval trees A segment tree T stores a set S of n intervals on the line. A standard segment tree is a balanced binary search tree over the $2n$ endpoints of S that are stored at the leaves of T. We use the same notation for both a leaf and the point that it contains. We associate an interval denoted by $range(v)$ with each node $v \in T$. If v is a leaf then $range(v)$ is the interval $[v, v]$. If v is an internal node, where w is the leftmost leaf in the subtree rooted by v, and z is the rightmost leaf in the subtree rooted by v, then $range(v) = [w, z]$. We associate with each node $v \in T$ a set $S(v)$ consisting of all segments $s \in S$ containing $range(v)$ but not containing $range(p(v))$. Each node $v \in T$ holds a secondary data structure, representing the set $S(v)$.

An interval tree is defined similarly. The difference is that in an interval tree an interval $s = (x_s, x_e)$ is in $S(v)$ if and only if v is the lowest common ancestor of x_s and x_e in T. It follows that a segment tree required $O(n \log n)$ space and an interval tree requires $O(n)$ space.

We use segment trees and interval trees which are not binary but each node has $O(d)$ children for some parameter d. The definition of $S(v)$ remains the same both for a segment tree and for an interval tree with a large fan-out.

For a point x on the line we denote by $P(x)$ the search path of x in T. The basic property of a segment tree is that the set of intervals that intersect a point x is the set $\{S(v) : v \in P(x)\}$. For an interval tree the set $\{S(v) : v \in P(x)\}$ contains (but not necessarily equals to) all intervals that intersect x. To locate exactly the intervals containing x we need to further search each secondary structure representing $S(v)$.

2.2 Weight-balanced B-trees We implement a segment tree and an interval tree as a weight-balance B-tree. Weight-balance B-trees were introduced by Arge and are mainly used to solve problems in the *external memory model* [3]. In this section we review a simpler version of weight-balance B-trees than the one introduced by Arge, and suggest a formal proof to their most important property.

Type		Space	Query	Insert	Delete
General	[4]	$O(n)$	$O(\log n \log\log n)$	$\bar{O}(\log n \log\log n)$	$\bar{O}(\log^2 n)$
General	[2]	$O(n)$	$O(\log n)$	$\bar{O}(\log^{1+\epsilon} n)$	$\bar{O}(\log^{2+\epsilon} n)$
General	[2]	$O(n)$	$O(\log n)$	$\bar{O}(\log n \log^{1+\epsilon} \log n)$	$\bar{O}(\log^2 n / \log\log n)$
Connected	[7]	$O(n)$	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$
Connected	[8]	$O(n \log n)$	$O(\log n)$	$\bar{O}(\log^3 n)$	$\bar{O}(\log^3 n)$
Monotone	[9]	$O(n \log n)$	$O(\log n)$	$\bar{O}(\log^2 n)$	$\bar{O}(\log^2 n)$
Monotone	[14]	$O(n \log n)$	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$
Convex	[19]	$O(N + n \log N)$	$O(\log n + \log N)$	$O(\log n \log N)$	$O(\log n \log N)$

Table 1: Previous results. We use $\bar{O}(*)$ to denote an amortized bound. N denotes the number of possible *y-coordinates* for edge endpoints in the subdivision.

For each node v we define following: $n(v)$ is the number of leaves in the subtree rooted by v, $h(v)$ is the height of v.

Definition 2.1. *A weight-balance B-tree T with branching parameter $d > 4$ is a search tree that keeps the items at its leaves, and has the following properties:*

1. *All leaves are at the same distance from the root (and their height is 0).*
2. *Each internal node $v \in T$ satisfies that $\frac{1}{2}d^{h(v)} < n(v) < 2d^{h(v)}$.*

The following properties immediately follow from this definition:

1. Each internal non root node has between $d/4$ and $4d$ children.
2. The root has between 2 and $4d$ children.
3. The height of T is $h = \Theta(\log n / \log d)$ (where n is the number of elements in the structure).

We keep a single point in each leaf, but note that Arge [3] defined weight-balance B-trees to have a leaf parameter k, which specifies the number of points stored in each leaf. An implementation of a weight-balance B-tree keeps in each node v, the numbers $n(v)$, $h(v)$, and a small search tree on $O(d)$ keys to direct the search. To find the leaf associated with an input key, we need to perform $h = O(\log n / \log d)$ search operations on these small search trees, which takes $O(\log n)$ time. When we insert a leaf into T, we may violate Definition 2.1 for some nodes $v \in T$. We handle such a violation by splitting v into two nodes. We implement deletions lazily by marking the relevant leaf ℓ as deleted. This *lazy-deletion* technique requires periodic rebuilding of the structure, so that the height of T remains $O(\log n / \log d)$ (n is the number of live elements in T). This rebuilding does not affect the amortized time bound of the operations. In the full version of this paper [13] we describe the implementation of each operation, and prove the following splitting theorem.

Theorem 2.1. *Let T be a weight-balance B-tree with branching parameter d, and let g be a function such that $g(n) \geq 1$ for all $n \geq 0$. Let $n(v) \cdot g(n)$ be the time spent when we split a node v, due to rebuilding of some secondary structures. Then the amortized time of insertion and deletion is $O(g(n) \cdot \log n / \log d)$.*

2.3 Dynamic fractional cascading Fractional Cascading (FC) [16] is a data structure to search a key in many ordered lists. Let U be an ordered set, and let $G = (V, E)$ be an undirected graph, which we call the *control graph*. Each vertex $v \in V$ is associated with a dynamic set $C(v) \subseteq U$, called the *catalogue* of v. By this definition, the size n of the input is proportional to $\sum_{v \in V} |C(v)| + |V| + |E|$.

Let $T' \subseteq V$ be a tree, and let $k \in U$ be a key. *FC* supports the following operation:

- *Find*(k, T'): For each $v \in T'$ return $y_v \in C(v)$, such that y_v is the successor of k in $C(v)$.

In the dynamic version of the problem we also support the following two operations:

- *Insert*(k, T'): For each $v \in T'$ insert k into $C(v)$.
- *Delete*(k, T'): For each $v \in T'$ delete k from $C(v)$.

The first implementation of dynamic *FC* was introduced by Mehlhorn and Naher [16]. Their implementation require that the degree of the underlying graph would be *locally bounded* by a constant: Let $R(e) = [l(e), r(e)]$ be a range associated with an edge $e \in E$ such that all *FC* operations with $e \in T'$ have their key k in $R(e)$. We say that G has *locally bounded degree* if there is a constant c such that for every vertex $v \in V$, and for every key $k \in U$, there are at most

c edges $e = (v, w)$ such that $k \in R(e)$. Our *control graphs* which are weight-balance B-trees with branching parameter d do not have locally bounded degree. Their *local degree* equals $O(d)$ where d is not a constant. To apply FC to a weight-balance B-tree, we use the implementation of Raman [20]. In the full version of this paper [13] we provide all the details regarding the implementation of Raman. The following theorem summarizes the properties of the FC data structure of Raman.

THEOREM 2.2. *We can implement FC in linear space such that each operation on a tree T' takes $O(\log n + |T'|(\log\log n + \log d))$ time. The time bound is worst-case for queries and amortized for insertions and deletions.*

To apply FC with a weight-balance B-tree T as the underlying graph we show [13] how to add and remove edges to the *control graph* G, as Raman [20] did not consider modifications to G. We also prove that if for each $v \in T$, $C(v)$ is proportional to $n(v)$, then the size of the data structure that FC holds in v is also proportional to $n(v)$. Our FC data structure then has the following property.

LEMMA 2.1. *Let the control graph T for the FC structure be a weight-balance B-tree with branching parameter d, such that for all $v \in G$, $|C(v)| \leq g(n) \cdot n(v)$ for some function g such that $g(n) \geq 1$ for $n \geq 0$. Then inserting or deleting an edge $e = (u, v)$ to or from T, respectively, takes $O(g(n)(n(u) + n(v))\frac{\log d + \log\log n}{d \log n})$ amortized time.*

3 Pointer machine implementations

In this section we present two simple solutions to the vertical ray shooting problem, both in the pointer machine model. Intuitively, we apply fractional cascading to a segment tree whose primary structure is a weight-balance B-tree T with branching parameter $O(\log^{\epsilon} n)$ (for some $\epsilon > 0$). The height of T is $O(\frac{1}{\epsilon}\log n/\log\log n)$, which allows us to achieve the desired bounds.

3.1 An implementation with optimal query time Our data structure is a segment tree T implemented as a weight-balance B-tree with fan-out $d = O(\log^{\epsilon} n)$. We store a segment $s = (x_s, x_e, y)$ in $S(v)$ for all the nodes $v \in T$ associated with the interval (x_s, x_e). Each segment is thus stored in at most $O(\log^{\epsilon} n)$ nodes at each level and a total of $O(\frac{1}{\epsilon}\log^{1+\epsilon} n/\log\log n)$ nodes overall. We use T as the *control graph* for a FC data structure, where for every node $v \in T$ we define the catalogue $C(v)$ as the set of segments $S(v)$ sorted by their *y-coordinate.*

Let $q = (q_x, q_y)$ be a query point. All the segments in S that intersect the vertical line $x = q_x$ are stored in the catalogues of the nodes on the search path $P(q_x)$ of q_x in T. Since we use FC it suffices to spend $O(\log\log n)$ time at each node v of $P(q_x)$, to locate the segment right above q in $S(v)$. So a query takes $O(\log n)$ time. In the full version of this paper [13] we bound the space used by this structure, and describe the implementation of each operation. The following theorem summarizes the properties of our data structure:

THEOREM 3.1. *The fully dynamic vertical ray shooting problem can be implemented in $O(\frac{1}{\epsilon} n \log^{1+\epsilon} n/\log\log n)$ space, to support queries in $O(\frac{1}{\epsilon}\log n)$ worst-case time, and updates in $O(\frac{1}{\epsilon}\log^{1+\epsilon} n)$ amortized time.*

3.2 An implementation with optimal update time Our data structure is a weight-balance B-tree T, similar to the structure of Section 3.1. We use T as the *control graph* for a FC structure, but we change the definition of the catalogues. We also store a secondary structure only at internal nodes of height at least two.

Let $s = (x_s, x_e, y)$ be a segment in S, let $P(x_s)$ and $P(x_e)$ be the search paths of x_s and x_e in T, and let $LCA(s)$ be the lowest common ancestor of x_s and x_e. Rather than storing a segment s in all nodes v such that $s \in S(v)$ we store s at the set of nodes on the suffixes of $P(x_s)$ and $P(x_e)$ from $LCA(s)$ to x_s and x_e respectively. We define $S'(v)$ to contain all segments (x_s, x_e, y) such that v is a descendant of $LCA(s)$ on $P(x_s)$, or v is a descendant of $LCA(s)$ on $P(x_e)$. The catalogue $C(v)$ now contains all segments of $S'(v)$ sorted by their y-coordinate.

Each internal node v of height at least two, has a secondary structure denoted by $M(v)$. The structure $M(v)$ is a small independent FC structure whose *control graph* is a star with a center v^*. The catalogue of v^*, $C(v^*)$, is identical to $C(v)$. For every pair of children (u, w) of v, we connect v^* to a leaf $\xi(u, w)$. The catalogue $C(\xi(u, w))$ contains all segments in $C(u) \bigcap C(w)$. That is, segments (x_s, x_e, y) such that x_s is a leaf descendant of u and x_e is a leaf descendant of w. In addition, for each child u of v, v^* has a leaf $\xi_r(u)$ whose catalogue $C(\xi_r(u))$ contains all segments (x_s, x_e, y) such that x_e is a leaf descendant of u and x_s is not a descendant of v. Similarly, v^* has a leaf $\xi_\ell(u)$ whose catalogue $C(\xi_\ell(u))$ contains all segments such that x_s is a leaf descendant of u and x_e is not a descendant of v. Note that $C(v^*)$ $(= C(v))$ is the union of all the catalogues of the leaves in $M(v)$. To gain some intuition note that insertion would be faster than for the data structure of Section 3.1 since a segment is contained only in $O(\frac{1}{\epsilon}\log n/\log\log n)$ catalogues. On the other hand, query is more expensive since for each node v on the query path we perform a FC query to $M(v)$ with a tree of size $O(\log^{2\epsilon} n)$.

We differ the description of the operations and the precise analysis to the full version of this paper. The following theorem summarizes the properties of our data structure.

THEOREM 3.2. *The fully dynamic vertical ray shooting problem can be implemented in* $O(\frac{1}{\epsilon} n \log n / \log\log n)$ *space, to support queries in* $O(\frac{1}{\epsilon} \log^{1+\epsilon} n)$ *worst-case time and updates in* $O(\frac{1}{\epsilon} \log n)$ *amortized time, for any* $\epsilon > 0$.

4 An implementation for the RAM model

This section provides an implementation for the vertical ray shooting problem that supports both queries and updates in logarithmic time, and requires $O(n \log n / \log\log n)$ space. To achieve this result, we use the unit cost RAM model, with a word size of w bits. Let W be the maximum number of objects in the structure. We assume that $\log W \leq w$. We allow arithmetic operations as well as bitwise boolean operations. For any integer $M \leq 2^w$ we denote by $[M]$ the set of integers in the interval $[0, M-1]$.

In the first part of this section, we introduce two data structures. The first structure is a Generalization of the Van Emde Boas structure [22] which we call $GVEB$. To implement the second structure, we use $GVEB$ to construct a Generalized *Union-Split-Find* structure which we call $GUSF$ (for a definition of Union-Split-Find see [11] Section 4.2, and [20] Section 5.2.3, and Section 4.2). These implementations are influenced by the work of Mortensen [17]. Mortensen ([17] Lemma 3.1), present a data structure which he denotes by S_n. Our $GVEB$ is similar to S_n, but is defined in a somewhat simpler way. $GVEB$ is designed to answer a different query than S_n. This is why we had to change some of the internals of S_n, and maintain different lookup tables. These modifications are made at the bottom level of the structure, so we have to describe it in detail to be able to describe our changes.

In the second part of this section, we present an implementation for the vertical ray shooting problem. This implementation, is similar to the one in Section 3.2. We hold a weight-balance B-tree T with branching parameter $d = \frac{1}{4} \log^{\frac{1}{8}} n$. Let $s = (x_s, x_e, y)$ be a segment of S, and let $P(x_s)$ and $P(x_e)$ be the search paths of x_s and x_e in T. We store s in all the nodes on $P(x_s) \bigcup P(x_e)$, where the secondary structure $M(v)$ of a node $v \in T$ is a $GUSF$ structure.

4.1 $GVEB$ - A generalized Van Emde Boas structure

Let $N, C \leq 2^w$ be two integers. A $GVEB$ structure with parameters (N, C) supports insertions and deletions of ordered pairs (k, c), where k is an integer in $[N]$, and c is an integer in $[\log^{\frac{1}{4}} C]$. We think of k as the key of the element, and of c as the color of the element. Let $C_q \subseteq [\log^{\frac{1}{4}} C]$ be some set of colors. $GVEB$ supports a $find(k, C_q)$ query that returns the successor of k with color $c \in C_q$. Our implementation is a variation of the recursive VEB structure [22], and supports all operations in $O(\log\log N)$ worst-case time. Instead of holding the minimum and maximum integers in the structure, each (recursive) structure keeps the minimum and maximum integers for each color $c \in C$. We use a *q-heap* data structure to manipulate these values in worst-case constant time. A *q-heap* [12] is a linear size data structure that supports insertions, deletions and successor queries in worst-case constant time. A *q-heap* with parameter M can accommodate up to $\log^{\frac{1}{4}} M$ elements of $[M]$, and requires a lookup table of size M. A *q-heap* also supports in constant time a $rank(k)$ query, that returns the number of elements smaller than k in the structure. We use q-heaps with parameter C.

In the full version of this paper [13] we provide all the details regarding this implementation. The following theorem summarizes the properties of our data structure.

THEOREM 4.1. *A* $GVEB$ *structure with parameters* (N, C) *requires* $O(N \log^{\frac{1}{4}} C)$ *space, can be initialized in* $O(N \log^{\frac{1}{4}} C)$ *time, requires a lookup table of size* C*, and supports find, insert and delete in* $O(\log\log N)$ *worst-case time.*

4.2 $GUSF$ - A generalized Union-Split-Find structure

The dynamic *union-split-find* structure holds a list of n elements subject to insertions and deletions, where some elements are marked. It supports a query with a key x that returns the marked successor of x in the list. Dietz and Raman implemented this structure in the RAM model ([11] Section 4.2, [20] Section 5.2.3). In their implementation all operations take $O(\log\log n)$ worst-case time. A $GUSF$ structure G with parameter C generalizes the dynamic *union-split-find* structure as follows. Each element x of G is associated with a subset $C(x) \subseteq [\log^{\frac{1}{4}} C]$ of colors. Let y and $prev$ be pointers to elements in the list, and let $C_q \subseteq [\log^{\frac{1}{4}} C]$ be a set of colors. $GUSF$ supports the following operations:

- $Find(y, C_q)$: Return the successor of y with color $c \in C_q$.
- $Add(y, prev)$: Inserts y immediately after $prev$, with $C(y) = \phi$.

- $Erase(y)$: Removes y from the structure. We assume that $C(y) = \phi$ when y is deleted.
- $Mark(y, c)$: Adds c to $C(y)$.
- $Unmark(y, c)$: Removes c from $C(y)$.

The implementation which we describe is similar to the implementation in Section 5.2.3 of [20]. The main difference is that we use the $GVEB$ structure of Section 4.1 rather than a VEB structure. Intuitively, the idea is to associate an integer value k with each element x of the list, where the order between those integers is equivalent to the order of the elements in the list. We assign integer values using the data structure of [23]. For each color $c \in C(x)$ we insert the pair (k, c) into a $GVEB$ structure of the previous section. Again, postponing details to the full version, the following theorem summarizes the properties of our data structure:

THEOREM 4.2. *A $GUSF$ with parameter C can be implemented in linear space, using a lookup table of size C, such that each operation takes $O(\log\log C + \log\log n)$ time, where the bound is amortized for insertions and deletions, and worst-case for all other operations.*

4.3 An implementation with optimal update and query time We implement a "compact" version of the segment tree structure of Section 2.1, by holding a $GUSF$ structure of Section 4.2 in each internal node. We maintain a weight-balance B-tree T with branching parameter $d = \frac{1}{4}\log^{1/8} n$. Each internal node $v \in T$ stores a $GUSF$ structure with parameter $N = O(n)$ (see Section 4.2), denoted by $M(v)$. The parameter N is set to n whenever we rebuild T. A segment (x_s, x_e, y) stored in $M(v)$ is mapped to a color that identifies the children of v such that x_s and x_e are their descendants. To achieve that we maintain at each node v the following three tables.

1. The table $U(v)$ maps each pair u, w of children of v, to a color $c(u, w) \in [\log^{\frac{1}{4}} n]$ (note that we allow u to equal w, so there is a color $c(u, u)$). This would be the color of every segment such that x_s is a descendant of u and x_e is a descendant of w. The table $U(v)$ also maps each child u of v to a color $c_\ell(u) \in [\log^{\frac{1}{4}} n]$ that corresponds to each segment such that x_s is a descendant of u and x_e is not a descendant of v. Similarly, $U(v)$ maps u to a color $c_r(u)$ that corresponds to each segment such that x_e is a descendant of u and x_s is not a descendant of v. We use this table to insert (delete) segments to (from) $M(v)$.
2. The table $Q(v)$ maps a child u of v to the following set of colors. For each pair (z, w) of children of v such that z is a left sibling of u and w is a right sibling of u, the colors $\{c(z, w), c_\ell(z), c_r(w)\}$ belong to this set. We use this table when we perform a query on $M(v)$.
3. The table $F(v)$ maps a child u of v to the set of colors containing $c_\ell(u), c_r(u)$ and $c(u, u)$. This set of colors also contains for each child $z \neq u$ of v the color $c(u, z)$. The role of this table is to replace the FC structure we used in the previous implementations.

We store a segment $s = (x_s, x_e, y)$ in the $M(v)$ structures of all the nodes v on $P(x_s) \bigcup P(x_e)$, where $P(x_s)$ and $P(x_e)$ are the search paths of x_s and x_e in T. For such a node v, we store y in $M(v)$ with a color $\overline{c}(s, v)$ that is defined as follows. If both $P(x_s)$ and $P(x_e)$ contain v then $\overline{c}(s, v) = c(v_s, v_e)$ where v_s is the child of v on $P(x_s)$ and v_e is the child of v on $P(x_e)$. If only $P(x_s)$ contains v then $\overline{c}(s, v) = c_\ell(u)$ where u is the child of v on $P(x_s)$. If only $P(x_e)$ contains v then $\overline{c}(s, v) = c_r(u)$ where u is the child of v on $P(x_e)$. Let r be the root of T. The structure $M(r)$ contains all the segments in our data structure. We also maintain a binary search tree T_0 over the *y-coordinates* of the segments stored in $M(r)$. We use T_0 to start the search in all the operations.

We now bound the space used by this structure. Each segment $s \in S$ is stored in $O(\log n / \log\log n)$ $GUSF$ structures. Each $GUSF$ structure requires space linear in the number of elements in it (see Theorem 4.2). To use the $GUSF$ structures with parameter $N = O(n)$ we also need a lookup table of size $O(n)$. We hold three additional tables, each of size $O(n)$. These tables are described below. So the overall space this structure requires is $O(n \log n / \log\log n)$.

We now describe the implementation of each operation.

Query: We perform a query with a point $q = (q_x, q_y)$ as follows. Let $(v_0, v_1, \ldots, v_k)$ be the search path for q_x in T, where v_0 is the root of T. Let y_i be the successor of q_y in $M(v_i)$. To find y_0 we use the search tree T_0. Then, we perform the following two steps for every $0 \leq i < k$. In the first step we find the segment s_i right above q in $M(v_i)$ by performing a $find(y_i, Q(v_{i+1}))$ on $M(v_i)$. In the second step we find y_{i+1} by performing a $find(y_i, F(v_{i+1}))$ on $M(v_i)$. Note that all the segments of $M(v_{i+1})$ are contained in $M(v_i)$, and their colors correspond to the set $F(v_{i+1})$. We return the segment with minimum *y-coordinate* in $\{s_i : 0 \leq i \leq k\}$. Correctness follows as we perform the query process described in Section 2.1.

We now bound the running time of a query. Searching in T_0 takes $O(\log n)$ time. Each query on a $M(v)$

structure takes $O(\log\log n)$ worst-case time (see Theorem 4.2). Since we perform $O(\log n/\log\log n)$ such queries, query takes $O(\log n)$ worst-case time. Next we define how to perform insertions and deletions.

Insert: We insert a segment $s = (x_s, x_e, y)$ in two phases. In the first phase we insert x_s and x_e into T. In the second phase we insert s into the $M(v)$ structures of nodes $v \in P(x_s) \bigcup P(x_e)$. We begin the second phase by finding the successors of y in the $M(v)$ structures of the nodes v on $P(x_s) \bigcup P(x_e)$ in $O(\log n)$ time, just like we did in query. For each node $v \in P(x_s) \bigcup P(x_e)$ we insert y to $M(v)$ with the color $\overline{c}(s, v)$. Each such insertion takes $O(\log\log n)$ time, so the second phase of the insertion takes $O(\log n)$ time.

We insert x_s and x_e to T using the regular implementation of insert into weight-balance B-trees. An insertion into a weight-balance B-tree may split nodes. We now describe how to carry out such splitting. Let v be a node that we need to split into v_ℓ and v_r, and let $p(v)$ be the parent of v in T. First, we delete v from T. Then, we update $Q(p(v))$, $F(p(v))$, and $U(p(v))$, and remove all the segments of $M(v)$ from $M(p(v))$, since their color in $M(p(v))$ may change. Then, we insert v_ℓ into T, so we have to update the tables representing $p(v)$ again, and create $M(v_\ell)$ and the three lookup tables associated with v_ℓ. Similarly, we insert v_r into T. Finally, we traverse the segments of $M(v)$, and insert each segment $s = (x_s, x_e, y)$ into $M(p(v))$. If either x_s or x_e is a descendant of v_ℓ we also insert s into $M(v_\ell)$, and similarly for $M(v_r)$.

We now explain how to update and create all the new lookup tables in constant time. When we delete v (or when we insert v_ℓ or v_r), the old lookup tables of $p(v)$, are no longer valid, and have to be replaced. The new lookup tables of $p(v)$ can be viewed as a function of the old tables and the index of v in the child list of $p(v)$. (The new lookup tables of v_ℓ and v_r are produced similarly.) To compute the new tables in constant time, we keep three super lookup tables: U^*, F^* and Q^* as follows.

U^* maps an old table $U(v)$ and an index $k \in [\log^{\frac{1}{8}} n]$, to a new table $U(v)$. The table $U(v)$ is a function $U : [\log^{\frac{1}{8}} n]^2 \to [\log^{\frac{1}{4}} n]$, since it maps a pair of children of v to a color. Therefore the number of entries in U^* is

$$\log^{\frac{1}{4}} n^{\log^{\frac{2}{8}} n} \cdot \log^{\frac{1}{8}} n << n\ .$$

Each entry of U^* contains of a table $U(v)$ of size $\log^{\frac{2}{8}} n$, so the overall space required by U^* is smaller than n.

Q^* maps an old table $Q(v)$ and an index k, to a new table $Q(v)$. The table $Q(v)$ is a function from $[\log^{\frac{1}{8}} n]$ to the power set of $[\log^{\frac{1}{8}} n]^2$, since it maps a child of v to a set of pairs of children of v. Therefore the number of entries in Q^* is

$$\left(2^{\log^{\frac{2}{8}} n}\right)^{(\log^{\frac{1}{8}} n)} \cdot \log^{\frac{1}{8}} n << n\ .$$

Each entry of Q^* contains of a table $Q(v)$. A table $Q(v)$ contains $\log^{\frac{1}{8}} n$ entries of size $\log^{\frac{2}{8}} n$ each, so the overall space required by Q^* is smaller than n.
The size of F^* can be bounded analogously. We rebuild U^*, Q^* and F^* whenever we rebuild T.

We now analyze the running time of insert. Other than the constant time operations described above, we perform a constant number of updates on $GUSF$ structures for every element in $M(v)$. Since $|M(v)| = O(n(v))$, these operation take $O(n(v)\log\log n)$ time, which dominates the running time of the split. By Theorem 2.1 insertion takes $O(\log n)$ amortized time.

Delete: Deleting a segment $s = (x_s, x_e, y)$ is an easier task, since we use the *lazy-deletion* technique (see Section 2.2). To delete s, we first remove s from the $M(v)$ structures of all the nodes on the search paths of x_s and x_e in T. Just like in insertion, these updates take $O(\log n)$ time. Then, we delete x_s and x_e from T (i.e., lazily marking these leaves as deleted) and we are done. Using the *lazy-deletion* technique requires periodical rebuilding of the entire structure (see Section 2.2). This rebuilding does not affect the amortized time bound of the operations. It follows that deletion also takes $O(\log n)$ amortized time.

The following theorem summarizes the properties of our data structure.

THEOREM 4.3. *The fully dynamic vertical ray shooting problem can be implemented in $O(n\log n\ /\log\log n)$ space, to support updates and queries in $O(\log n)$ time, where the bound is amortized for updates and worst-case for queries.*

5 How to use only linear space

In this section we show how to reduce the space required by the three structures we presented in this paper. The space saving technique, is influenced from the work of Baumgarten *et al.* [4], that present a linear space implementation for dynamic point location in general subdivisions. Intuitively, we save space by inserting the segments of S into a large fan-out *interval tree*, where only a fraction of the segments is also inserted into a *segment tree*. Here we show how to reduce the space of the data structure described in Section 3.1. The details related to reducing the space of the two other data structures are in [13].

We use as the primary structure a variation of the large fan-out interval tree defined in Section 2.1, implemented as a weight-balance B-tree (see Section

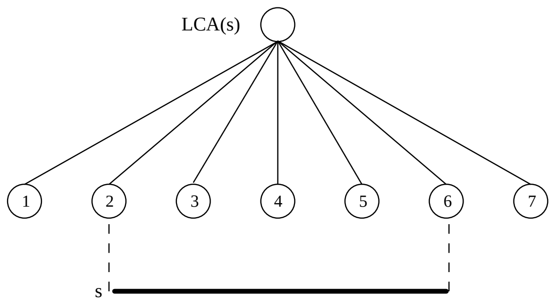

Figure 1: The segment s belongs to $I_\ell(2)$, $I_r(6)$, $I_m(3)$, $I_m(4)$ and $I_m(5)$

2.2). We denote this tree by T_I. The difference is that a node $v \in T_I$ does not maintain $S(v)$ in a secondary data structure, but segments from $S(p(v))$ as follows.

Consider a segment $s = (x_s, x_e, y) \in S$. Let v_s be the child of $LCA(x_s, x_e)$ such that x_s is a leaf descendant of v_s. Similarly, let v_e be the child of $LCA(x_s, x_e)$ such that x_e is a leaf descendant of v_e. We store s at all children of $LCA(x_s, x_e)$ that are between v_s and v_e (including v_s and v_e). For each node $v \in T$ we define $I(v)$ as the set of segments $s = (x_s, x_e, y)$ such that v is a child of $LCA(x_s, x_e)$ between v_s and v_e. We divide the set $I(v)$ into three subsets: $I_\ell(v)$, $I_m(v)$ and $I_r(v)$. The set $I_\ell(v)$ contains the segments $s = (x_s, x_e, y) \in I(v)$ such that $v = v_s$. The set $I_r(v)$ contains the segments such that $v = v_e$. The set $I_m(v)$ contains the segments $s \in I(v)$ such that s contains $range(v)$ (See Figure 1).

The main property of an *interval tree* still applies: Let $x \in \mathbb{R}$ be a point, and let $P(x)$ be the search path for x in T. Let $I_\ell(v, x)$ denote the set of segments of $I_\ell(v)$ whose left endpoints is smaller then x, and let $I_r(v, x)$ denote the set of segments of $I_r(v)$ whose right endpoints is larger then x. The set of all the segments that intersect the vertical line $X = x$ is $\{v \in P(x) : I_\ell(v, x) \cup I_r(v, x) \cup I_m(v)\}$.

We build on top of T_I three FC data structures; Π_ℓ, Π_m, and Π_r. The FC structure Π_m has in each node $v \in T_I$ a catalogue $C(v)$ containing the set of segments $I_m(v)$, sorted by their *y-coordinate*. The structures Π_ℓ and Π_r are defined analogously. For each $v \in T_I$, we divide the set $I_r(v)$ in Π_r into blocks each of size $\Theta(\frac{1}{\epsilon}\log^{1+\epsilon} n/\log\log n)$. For each block B_r of $I_r(v)$ we maintain the segment $win(B_r)$ of maximum right endpoint. Similarly, we divide the set $I_\ell(v)$ into blocks, and for each block B_ℓ of $I_\ell(v)$ we maintain the segment $win(B_\ell)$ of minimum left endpoint.

In addition to T_I we maintain two *segment trees*; T_r and T_ℓ as in Section 3.1. The structure of Section 3.1 supports a $Find^+(q)$ query that returns the segment right above q. We extend T_r and T_ℓ in a straightforward way to also support a $Find^-(q)$ query, that returns the segment right below q. For every node $v \in T_I$, and every block B_r of $I_r(v)$ we maintain $win(B_r)$ in T_r, unless B_r is the only block representing the set $I_r(v)$. Similarly, for every node $v \in T_I$, and every block B_ℓ of $I_\ell(v)$ we maintain $win(B_\ell)$ in T_ℓ, unless B_ℓ is the only block representing the set $I_\ell(v)$.

A block B_r of Π_r is implemented as a balanced binary search tree, whose leaves correspond to the set of *y-coordinates* of elements in B_r. In addition, each internal node u holds the maximum *x-coordinate*, $max(u)$, of a segment in its subtree. The blocks of Π_ℓ are implemented symmetrically. Next we bound the space used by this structure.

By Theorem 2.2 the space used by Π_m is $O(n\log^\epsilon n)$, since each segment in Π_m is stored in $O(\log^\epsilon n)$ catalogues. A segment $s_r \in \Pi_r$ is stored in a single catalogue and in a single block. It follows that Π_r requires linear space. The *segment tree* T_r keeps $O(n/(\frac{1}{\epsilon}\log^{1+\epsilon} n/\log\log n))$ segments of Π_r, and thus uses linear space (by Theorem 3.1). The space used by Π_ℓ and by T_ℓ is also linear. It follows that our structure uses $O(n\log^\epsilon n)$ space. To perform a query, we first query T_ℓ and T_r, which returns the segment right above q and the segment right below q amongst all the winners. We show that with these segments we can find the answer in T. In the full version of this paper [13] we provide all the details of how to perform each operation. The following theorem summarizes the properties of our data structure:

THEOREM 5.1. *There exists a data structure for the fully dynamic vertical ray shooting problem which requires $O(n\log^\epsilon n)$ space, supports queries in $O(\frac{1}{\epsilon}\log n)$ worst-case time, and updates in $O(\frac{1}{\epsilon}\log^{1+\epsilon} n)$ amortized time, for any $\epsilon > 0$.*

We reduce the space of the structure described in Section 3.2 to linear, using a similar technique. The main difference is that we divide the sets of segments represented at the leaves of each $M(v)$ structure into blocks. This result is summarized in the following theorem.

THEOREM 5.2. *There exists a data structure for the fully dynamic vertical ray shooting problem that requires linear space, supports queries in $O(\frac{1}{\epsilon}\log^{1+\epsilon} n)$ worst-case time, and updates in $O(\frac{1}{\epsilon}\log n)$ amortized time, for any $\epsilon > 0$.*

To reduce the query time in Theorem 5.2, we use the $M(v)$ structure described in Section 4.3, rather than the $M(v)$ structure described in Section 3.2. This way we finally achieve an optimal solution, as stated in the following theorem.

THEOREM 5.3. *There exists a data structure for fully dynamic vertical ray shooting problem that requires linear space, supports queries in $O(\log n)$ worst-case time, and updates in $O(\log n)$ amortized time.*

6 Open Problems

We present an optimal implementation to the vertical ray shooting problem, that works in the *RAM* model of computation. It is an open question whether a linear space implementation that supports all operations in logarithmic time exists for the pointer machine model.

Our implementation can be used for a set of segments that are not necessarily horizontal but have a constant number of different slopes (we can use a different data structure for each slope). It is an open question whether there exists an optimal solution for segments that have a non constant but still small number of different slopes.

Recent papers ([6], and [18]) present implementations to the static planar point location problem with query time $o(\log n)$, where the endpoints of all the segments belong to a $[u]$ by $[u]$ grid. It is an open question whether the (optimal) bounds presented in this paper can be improved, under the assumption that the endpoints of all the segments belong to a $[u]^2$ grid.

References

[1] P. Agarwal, L. Arge, and K. Yi. An optimal dynamic interval stabbing-max data structure? In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 803–812, 2005.

[2] L. Arge, G. S. Brodal, and L. Georgiadis. Improved dynamic planar point location. In *Proc. 47th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2006.

[3] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32:1488–1508, 2003.

[4] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *J. of Algorithm*, 17:342–380, 1994.

[5] A. L. Buchsbaum and M. T. Goodrich. Three-dimensional layes of maxima. *Algorithmica*, 39:275–286, 2004.

[6] Timothy M. Chan. Point location in $o(\log n)$ time, voronoi diagrams in $o(n \log n)$ time, and other transdichotomous results in computational geometry. In *Proc. 47th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2006.

[7] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM Journal on Computing*, 21:972–999, 1992.

[8] Y. J. chiang, F. P. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. *SIAM Journal on Computing*, 25:207–233, 1996.

[9] Y. J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *International Journal of Computational Geometry and Applications.*, 2(3):311–333, 1992.

[10] E. D. Demaine, J. Iacono, and S. Langerman. Retroactive data structures. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 274–283, 2004.

[11] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.

[12] M. L. Fredman and D. E. Willard. Transdichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994.

[13] Y. Giyora. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. Master's thesis, Dept. of Computer Science, Tel-Aviv University, ISRAEL, 2006.

[14] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. *SIAM Journal on Computing*, 28(2):612–636, 1998.

[15] H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic rectangular intersection with priorities. In *Proc. 35th Annual ACM Symposium on Theory of Computing (STOC)*, pages 639–648. ACM Press, 2003.

[16] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[17] C. W. Mortensen. Fully dynamic two dimensional range and line segment intersection reporting in logarithmic time. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 618–627, 2003.

[18] Mihai Patrascu. Planar point location in sublogarithmic time. In *Proc. 47th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2006.

[19] F. P. Preparata and R. Tamassia. Dynamic planar point location with optimal query time. *Theoretical Computer Science*, 74:94–114, 1990.

[20] R. Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time.* PhD thesis, Dept. of Computer Science, University of Rochester, 1992.

[21] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. *Information Processing Letters*, 6(3):80–82, 1977.

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

[23] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequencially ordered fine in good worst-case time. *Information and computation*, 97(2):150–174, 1992.