Dynamic Planar Point Location in Optimal Time

Yakov Nekrich*
Michigan Technological University
USA
yakov.nekrich@gmail.com

ABSTRACT

In this paper we describe a fully-dynamic data structure that supports point location queries in a connected planar subdivision with n edges. Our data structure uses $O(n)$ space, answers queries in $O(\log n)$ time, and supports updates in $O(\log n)$ time. Our solution is based on a data structure for vertical ray shooting queries that supports queries and updates in $O(\log n)$ time.

CCS CONCEPTS

• **Theory of computation → Sorting and searching**.

KEYWORDS

computational geometry, point location, dynamic data structures

ACM Reference Format:
Yakov Nekrich. 2021. Dynamic Planar Point Location in Optimal Time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC '21), June 21–25, 2021, Virtual, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3406325.3451100

1 INTRODUCTION

Planar point location is a fundamental problem with applications ranging from computer graphics to computer aided design. In this problem we store a connected planar subdivision in a data structure so that for any query point q the polygon that contains q can be found efficiently. The static data structure that uses $O(n)$ space, where n is the number of edges in the subdivision, and answers point location queries in optimal $O(\log n)$ time was presented by Sarnak and Tarjan [27]. In the dynamic scenario the subdivision can be updated by inserting or removing edges. The dynamic planar point location problem has been studied extensively over the last three decades [1, 3, 6, 8, 14, 15, 18, 19, 21, 26]. The existence of a dynamic data structure supporting queries and updates in $O(\log n)$ time is a major problem in geometric data structures [10, 11, 16, 28].

In the vertical ray shooting problem we keep a set of non-intersecting segments in a data structure, so that for an arbitrary query point q the first segment hit by a vertical downward ray from q can be found. Point location in a connected subdivision can be reduced to the vertical ray shooting problem: if we know the edge immediately below q, we can identify the polygon that contains q by keeping the segments in a union-split-find data structure. In this paper we present a data structure that answers dynamic vertical ray shooting queries in $O(\log n)$ time and supports segment insertions and deletions in $O(\log n)$ time. This result immediately implies the point location data structure that supports queries and updates in logarithmic time. Thus our result provides an answer to the long-standing open question [10, 11, 16, 28].

Main results on this problem are listed in Table 1. Optimal time for both updates and queries was previously achieved only for the special case of the orthogonal subdivision [19]. As will be explained later, the orthogonal case is arguably easier to handle. In all other scenarios the most efficient previously known data structures were described by Chan and Nekrich [8]. In order to achieve the optimal query time, the data structure from [8] requires an extra $O(\log^{\epsilon} n)$ factor in update cost. Other trade-offs presented in [8] are even closer to the optimal solution. However, as stated in [8], "Removing all the $\log\log n$ factors in the query and update time of our main result remains very challenging.".

Our Approach. The main challenge in the vertical ray shooting data structures is the lack of global order on the set of segments. Even a single insertion or deletion can significantly change the order of segments; see Fig. 1 on p. 4. The only exception is the special case of horizontal segments (corresponding to point location in an orthogonal subdivision). In this special case segments can be ordered by their y-coordinates. This is also the only case for which the $O(\log n)$-time solution was known previously [19].

Solutions of the general vertical ray shooting problem rely on the segment tree data structure or its variants. In these solutions every segment must be stored circa $O(\log n)$ times in different nodes of the segment tree. Since the global order of segments cannot be maintained, most previous methods with nearly-logarithmic query time require roughly $O(\log^2 n)$ time for either insertions or deletions. Two exceptions are the data structure of Cheng and Janardan[13] with $O(\log^2 n)$ query time and the result of Chan and Nekrich [8].

Chan and Nekrich [8] partially solve the issue of segment order by assigning categories[1] to segments. We can maintain the order among segments with the same category, even though they are stored in different nodes. On the other hand, segments stored in the same node can be assigned different categories. Therefore the list of segments stored in each node must be divided into multiple sublists that must be queried independently; this increases the time needed to answer a query. Different trade-offs between query and update time can be achieved with the approach of Chan and Nekrich [8],

*Research was supported in part by an MTU start-up grant D-97457.

STOC '21, June 21–25, 2021, Virtual, Italy

ACM ISBN 978-1-4503-8053-9/21/06…$15.00
https://doi.org/10.1145/3406325.3451100

[1] In [8] the authors used the term color. In this paper we speak of segment categories to avoid confusing notation.

Table 1: Previous and new results on dynamic planar point location. Entries marked M and C show results for monotone subdivisions and connected subdivisions respectively; O denotes an orthogonal subdivision and G denotes data structures for vertical ray shooting among non-intersecting segments. Entries marked † and ‡ require amortization and (Las Vegas) randomization respectively. In this table ε is an arbitrarily small positive constant.

Reference	Space	Query Time	Insertion Time	Deletion Time	
Bentley [6]	$n \log n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	G
Fries [18]	n	$\log^2 n$	$\log^4 n$	$\log^4 n$	C
Preparata–Tamassia [26]	n	$\log^2 n$	$\log^2 n$	$\log^2 n$	M
Chiang et al. [14]	$n \log n$	$\log n$	$\log^3 n$	$\log^3 n$	C
Chiang–Tamassia [15]	$n \log n$	$\log n$	$\log^2 n$	$\log^2 n$	M
Cheng–Janardan [13]	n	$\log^2 n$	$\log n$	$\log n$	G
Baumgarten et al. [3]	n	$\log n \log\log n$	$\log n \log\log n$	$\log^2 n$	G†
Goodrich–Tamassia [21]	n	$\log^2 n$	$\log n$	$\log n$	M
Arge et al. [1]	n	$\log n$	$\log^{1+\varepsilon} n$	$\log^{2+\varepsilon} n$	G†
Arge et al. [1]	n	$\log n$	$\log n(\log\log n)^{1+\varepsilon}$	$\log^2 n/\log\log n$	G†‡
Giyora–Kaplan [19]	n	$\log n$	$\log n$	$\log n$	O
Chan–Nekrich [8]	n	$\log n(\log\log n)^2$	$\log n \log\log n$	$\log n \log\log n$	G
Chan–Nekrich [8]	n	$\log n$	$\log^{1+\varepsilon} n$	$\log^{1+\varepsilon} n$	G
Chan–Nekrich [8]	n	$\log n$	$\log^{1+\varepsilon} n$	$\log n(\log\log n)^{1+\varepsilon}$	G
Chan–Nekrich [8]	n	$\log^{1+\varepsilon} n$	$\log n$	$\log n$	G
Chan–Nekrich [8]	n	$\log n \log\log n$	$\log n \log\log n$	$\log n \log\log n$	G‡
this paper	n	$\log n$	$\log n$	$\log n$	G

see Table 1. However, it seems that obtaining $O(\log n)$ time for both queries and updates is not possible with this approach.

In this paper we demonstrate that it is sufficient to maintain a partial order of segments in selected nodes of the segment tree. Our solution embeds *a macro-tree* with large node degree into the segment tree, as explained in Section 3. Segments stored in a node of the macro-tree are assigned categories as in [8]. The main idea of our approach is *re-ordering* the segments in each macro-tree node and attempting to insert the segments into the segment tree nodes. We can show that the majority of segments in a macro-tree node can be quickly inserted into the appropriate nodes of the segment tree. When a query is answered we can identify and process the "missed" segments (i.e., the segments that are not inserted into the segment tree) without increasing the query time. In order to obtain the final solution with optimal update and query time additional ideas are necessary. We modify the weighted telescoping search approach from [25] by limiting the height of the weighted search trees. This approach is then combined with other techniques, such as fractional cascading and segment categories.

We describe the macro-tree and the partial order of segments in the macro-tree nodes in Sections 3 and 4 respectively. We explain how segments are inserted from a macro-tree node into the nodes of the segment tree in Section 5. The remaining parts of our construction are presented in Sections 6 and Section D. In Section 2 we provide some preliminary definitions and sketch some previously designed techniques used in this paper.

Throughout this paper n will denote the total number of segments stored in a data structure. We will denote by $\mathtt{ray}(q, S)$ the answer to a vertical ray shooting query for a point q on a set of segments S. Our result is valid in the standard RAM model. We assume that random access to memory cells, coordinate-wise comparisons of points, and point–segment comparisons can be executed in constant time. The input can be real-valued. Some parts of our construction also require "non-standard" operations, such as finding the most significant bit. However, these operations are performed on small integers only and we can implement them using a look-up table of size $o(n)$. The look-up table can be initialized using additions and subtractions of small integers.

2 PRELIMINARIES

Segment Tree. Vertical ray shooting queries are answered using the segment tree data structure. In this paper we will employ the segment tree with node degree $\log^\varepsilon n$. Leaves of the segment tree T contain x-coordinates of segment endpoints; all leaves in T have the same depth. We associate a vertical slab $[X, X_{\mathtt{next}}) \times \mathbb{R}$ with every node ℓ, where X is the x-coordinate stored in ℓ and $X_{\mathtt{next}}$ is the x-coordinate stored in its right neighbor. The slab associated to an internal node is the union of slabs associated to its children. A segment s spans (the slab of) a node $u \in T$ if it intersects the

left and the right boundaries of the slab associated to a node u. We will say that a segment s is *relevant* for a node u (resp a node u is relevant for a segment s) if s does not span u but s spans at least one of its children. We keep every segment s in all nodes u of the segment tree that are relevant for s. Thus each segment is stored in at most two nodes u on every level of T. Hence every segment s must be stored in $O(h)$ nodes, where h is the height of the segment tree T.

In order to answer a vertical ray shooting query for a point q, we visit all nodes that contain q. All such nodes are on one root-to-leaf path of the segment tree. In every visited node we can find the first segment hit by a vertical ray from q using binary search in $O(\log n)$ time.

Union-Split-Find. Let A be an ordered set of elements and $B \subseteq A$. An ordered Union-Split-Find (USF) data structure finds, for any query element $q \in A$, the largest $e \in B$ such that $e \leq q$. A dynamic USF data structures supports insertions and deletions of elements into and from A (an inserted or deleted element can be also an element of B). We can support both updates and USF queries in $O(\log\log n)$ time where n is the size of A.

Finger Search. For an ordered set A, the predecessor of q in A is the largest element $x \in A$ such that $x \leq q$, $\mathrm{pred}(q, A) = \max\{\, x \in A \mid x \leq q \,\}$. If we are given a pointer to an element $e \in A$ and a query q, we can find $x = \mathrm{pred}(q, A)$ in $O(\log d)$ time, where d is the distance (i.e., the number of elements in A) between e and x, $d = |\{\, e' \in A \mid \min(x, e) \leq e' \leq \max(x, e) \,\}|$ is the number of elements in A between e and x [22]. We can modify the finger search and specify d as the search parameter: If we are given a pointer to an element $e \in A$, a query q, and an integer d, we can find $\mathrm{pred}(q, A)$ in $O(\log d)$ time provided that the distance between e and $\mathrm{pred}(q, A)$ is at most d; if the distance between q and pred exceeds d the finger search terminates after $O(\log d)$ time. We can also find the successor within the same time, where the successor of q in A is the smallest element $x \in A$ such that $x \geq q$, $\mathrm{succ}(q, A) = \min\{\, x \in A \mid x \geq q \,\}$.

Fractional Cascading. Suppose that we store an ordered list $C(u)$ in every node u of a tree T with node degree $\log^{\varepsilon} n$. We can search in all lists $C(u)$ along a root-to-leaf path in T using a fractional cascading technique. The dynamic variant [23] of the fractional cascading [12] supports searching in every $C(u)$ in time $O(\log\log n)$.

If we apply fractional cascading to lists of segments in a segment tree, then we can answer ray shooting queries in each node in $O(\log\log n)$ time. Thus the overall query time is $O(\log n)$. We refer to [1, 3, 8] for the description of fractional cascading adopted to the vertical ray shooting problem. However, fractional cascading cannot be used for updates. When a new segment is inserted into $O(h)$ different nodes, we must spend $O(h \log n)$ time.

Segment Categories. In order to speed-up the update and query time , Chan and Nekrich [7] introduced the concept of segment categorization (segment coloring).

Every segment stored in the data structure is assigned one or several colors that satisfy the following properties: for any category j, there is a vertical line l_j such that all segments with the same category j intersect l_j (ii) all segments that must be stored in a node u of the segment tree are assigned $O(\log^{\varepsilon} n)$ different categories (iii) each segment s is assigned $O(1)$ different categories (a segment can be assigned different categories in different nodes u_i where it must be stored).

Segment categorization significantly improves the update-query trade-offs. We classify all segments stored in a node u according to their categories; for each fixed j, we store all category-j segments from a node u in a separate list. Using fractional cascading, a new segment s can be inserted into all category-j lists in $O(\log n)$ time. Since a segment is assigned $O(1)$ categories, the total insertion time is logarithmic. However we must store $O(\log^{\varepsilon} n)$ independent lists in each node. This increases the query time by an $O(\log^{\varepsilon} n)$ factor.

3 BASE TREE AND MACRO-TREE

Our base tree has node degree $\log^{\varepsilon} n$. Leaves of T contain projections of segment endpoints onto the x-axis. We associate vertical slabs to nodes of T in the same way as in the segment tree. Additionally we embed the tree with node degree $2^{\varepsilon \log^{\delta} n}$, further called the macro-tree, into the base tree. Every node v of the macro-tree corresponds to a node $u \in T$, such that the height of u divides $\log^{\delta} n / \log\log n$. Constants ε and δ are chosen in such way that $\varepsilon \leq \delta/3$ and $\delta \leq 1/4$. The macro-tree is organized as a segment tree with large node degree [1, 8, 25]: we keep a segment s in every node v such that s spans at least one child of v, but s does not span v. If s intersects the left boundary of v, then we keep s in the list of left segments $\mathcal{L}(v)$ (in this case the right endpoint of s must be in the slab of v by definition of the segment tree). If s intersects the right boundary of v, we keep s in the set of right segments $\mathcal{R}(v)$. If s intersects neither the right nor the left boundary of v (i.e., s is entirely contained in the slab of v), we keep it in the set of middle segments $\mathcal{M}(v)$. Each segment is stored in at most one list $\mathcal{M}(v)$ and in $O(h)$ lists $\mathcal{L}(v_i)$ (resp. $\mathcal{R}(v_j)$) where $h = O(\log^{1-\delta} n)$ is the height of the macro-tree.

Segments in list $\mathcal{L}(v)$ are assigned categories using the method from [8], sketched in Section 2. Let $\mathcal{L}_j(v)$ denote the set of category-j segments in $\mathcal{L}(v)$. When a new segment is inserted, we can insert it into all appropriate lists $\mathcal{L}_{j_v}(v)$ in $O(\log n)$ time, as explained in [8]. Since the macro-tree has very large node degree, we cannot directly answer ray shooting queries on $\mathcal{L}_j(v)$. We could insert segments from $\mathcal{L}_j(v)$ into relevant nodes u of the base tree, but then the overall query time would grow by $O(\log^{\varepsilon} n)$ factor.

Our solution is based on *re-ordering segments* in $\mathcal{L}_j(v)$ that will be described in Section 4. Using this re-ordering we then insert segments from $\cup_j \mathcal{L}_j(v)$ into selected relevant nodes of the subtree T_v. An interesting feature of our method is that we do not have to maintain the total order of all segments in $\mathcal{L}(v)$. We also do not need to maintain the list of all relevant segments in a node u. We insert a segment into a node only if it can be done in $O(\log\log n)$ time. We show that it is possible to answer ray shooting queries even though only a partial order of segments in $\mathcal{L}(v)$ is known and lists of segments in the base tree nodes are incomplete. The insertion and query procedures are described in Section 5. Deletions and some technical details are deferred to Sections A and D. The set of right segments is processed in the same way.

Every middle segment $s \in \mathcal{M}(v)$ is inserted into lists $M(u)$ for all relevant nodes u of T, such that v is the lowest ancestor of u in the macro-tree. Segments in $M(u)$ are also assigned categories. A query is answered using a new variant of the weighted telescoping

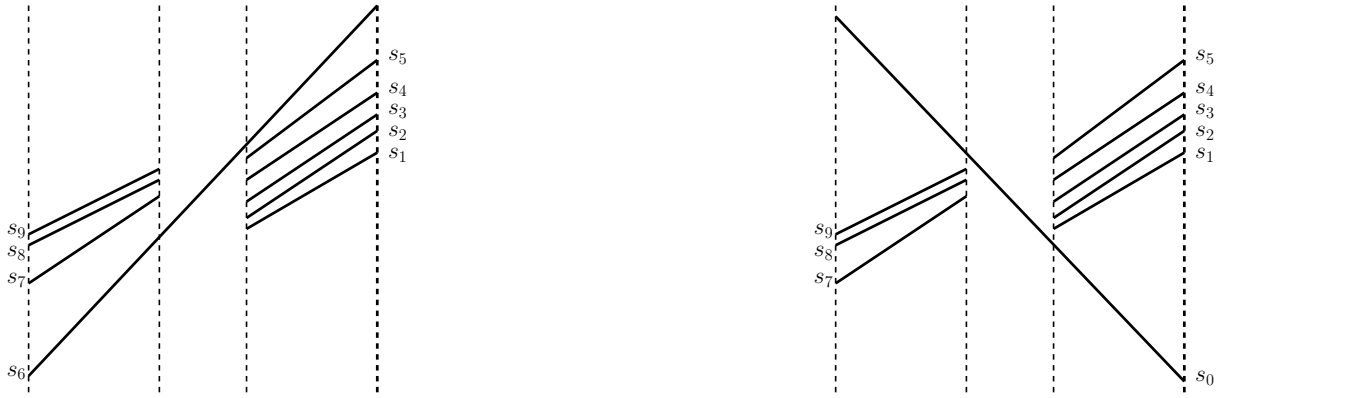

Figure 1: Example of a volatile segment order. A deletion and an insertion of one new segment in a multi-slab changes the order of segments from $s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_5 \prec s_6 \prec s_7 \prec s_8 \prec s_9$ (left) to $s_7 \prec s_8 \prec s_9 \prec s_0 \prec s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_5$ (right).

search[25] combined with the approach of [8]. Details can be found in Section 6.

4 ALMOST ORDERED LIST

The following problem plays an important role in our construction. Suppose that we are given a sequence of dynamic ordered lists $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_f$. Let $pred(e, X)$ denote the predecessor of an element e in a set X. We assume that for every new element e, we know $pred(e, \mathcal{L}_{j(e)})$ in some list $\mathcal{L}_{j(e)}$. We will say that a list $\mathcal{L} = \cup_{j=1}^{f} \mathcal{L}_j$ is *almost ordered* if we can represent $\mathcal{L}$ as a union of $f+1$ disjoint ordered lists, $\mathcal{L} = \mathcal{L}^g \cup (\cup_{j=1}^{f} \mathcal{L}'_j)$, so that (i) every list $\mathcal{L}'_j \subseteq \mathcal{L}_j$ is divided into groups $G_i(\mathcal{L}'_j)$ of $O(\log^2 n)$ elements and (ii) for each $G_i(\mathcal{L}'_j)$ we know its *range* $[e_l, e_f]$ in $\mathcal{L}^g$ satisfying $e_l < e < e_f$ iff $e \in G_i(\mathcal{L}'_j)$ for every $e \in \mathcal{L}'_j$. Elements in $\mathcal{L}^g$ are further called *good* elements and elements in all $\mathcal{L}'_j$ are further called *bad* elements. For a group $G_i(\mathcal{L}_j)$ we denote by $last(G_i)$ the largest element in $G_i(\mathcal{L}_j)$; $plast(G_i)$ denotes the largest element in the group $G_{i-1}(\mathcal{L}_j)$ that precedes $G_i(\mathcal{L}_j)$.

Lemma 4.1. *The union of f ordered lists $\mathcal{L} = \cup_{j=1}^{f} \mathcal{L}_j$ can be maintained as an almost-ordered list under insertions and deletions in time $O(f \cdot \log\log n)$ per update, so that (i) for every good element e and for each j, $1 \le j \le f$, we can find in $O(\log\log n)$ time the group G_i of $\mathcal{L}'_j$ satisfying $plast(G_i) < e \le last(G_i)$ and (ii) for every bad element e' we can find its group and its range in $O(\log\log n)$ time.*

Proof. We split each list $\mathcal{L}_j$ into groups of $\log^2 n$ elements. For an element $e \in \mathcal{L}_j$, we can find its group G_i and $plast(G_i)$ in $O(\log\log n)$ time using the dynamic USF data structure [23]. For all groups, a copy of the element $last(G_i)$ is stored in the list $\mathcal{L}^g$. See Fig. 2 for an example.

Suppose that a new element e is inserted into a list $\mathcal{L}_j$. For the group G_i that contains e, we find $plast(G_i)$ and its position in $\mathcal{L}^g$. Next we look for the predecessor of e in $\mathcal{L}^g$ among $\log^5 n$ elements[2] that follow $plast(G_i)$ in $\mathcal{L}^g$ using finger search. If $\text{pred}(e, \mathcal{L}^g)$ is found, we insert a copy of e into $\mathcal{L}^g$. Otherwise we insert a copy of e into $\mathcal{L}'_j$. This step takes $O(\log\log n)$ time. Since the position of e in $\mathcal{L}_j$ is known, we can find its position in $\mathcal{L}'_j$ using a USF data structure in $O(\log\log n)$ time.

Using standard techniques, we can guarantee that the size of each group $G_i(\mathcal{L}_j)$ is $\Theta(\log^2 n)$. When a group G_i is split into two groups, we find $last(G'_i)$ and $last(G''_i)$ and insert them into $\mathcal{L}^g$. The cost of inserting two new elements into $\mathcal{L}^g$ is $O(\log n)$; it can be distributed among $\Theta(\log^2 n)$ updates of G_i. Hence the total cost of an insertion is $O(\log\log n)$. When an element is deleted from a list $\mathcal{L}_j$ we delete its copy from $\mathcal{L}'_j$ or $\mathcal{L}^g$. The only exception is $last(G_i)$. When the last element e in a group G_i is removed and its predecessor in G_i is a bad element, we keep e in $\mathcal{L}_j$ and $\mathcal{L}^g$, but we mark it as deleted; if the predecessor e' of e is a good element, we remove e and set $last(G_i) = e'$. Since $\mathcal{L}'_j \subseteq \mathcal{L}_j$, $G_i(\mathcal{L}'_j) = O(\log^2 n)$ where $G_i(\mathcal{L}'_j) = G_i(\mathcal{L}_j) \cap \mathcal{L}'_j$. Thus each $\mathcal{L}'_j$ is divided into groups of $O(\log^2 n)$ elements.

We also maintain a separate USF data structure for good elements from $\mathcal{L}_j \setminus \mathcal{L}'_j$, $j = 1, \ldots, f$, and their positions in $\mathcal{L}^g$. For every $\mathcal{L}_j$, $1 \le j \le f$, and for any good element $e \in \mathcal{L}^g$ we can find the largest element $e' \in \mathcal{L}_j \setminus \mathcal{L}'_j$ such that $e' \le e$. When a new element is inserted into or deleted from $\mathcal{L}^g$, we update all such USF data structures in $O(f \log\log n)$ time. □

Thus the union of f ordered lists $\mathcal{L}_j$ is divided into a list of good elements $\mathcal{L}^g$ and f lists $\mathcal{L}'_j \subseteq \mathcal{L}_j$ of bad elements. For every good element e and for every $\mathcal{L}'_j$, we can quickly find the group $G_i(\mathcal{L}_j)$, such that $plast(G_i) < e \le last(G_i)$. The majority of elements in the re-ordered lists are good elements: for every group of $\Theta(\log^5 n)$ good elements there are $O(\log^{2+\varepsilon} n)$ bad elements. We will use the notation $plast(e) = plast(G)$ and $last(e) = last(G)$ where G is the group containing e.

5 LEFT SEGMENTS IN MACRO-TREE

Let $\mathcal{L}(v)$ denote the list of left segments stored in a macro-tree node v. All segments in $\mathcal{L}(v)$ are assigned categories using the method from [7], as explained in Section 2. Thus $\mathcal{L}(v)$ is a union of $O(\log^{\varepsilon} n)$ ordered lists $\mathcal{L}_1(v), \ldots, \mathcal{L}_f(v)$. We apply the re-ordering

[2] This choice of the power of log will be clear from the description in the following sections.

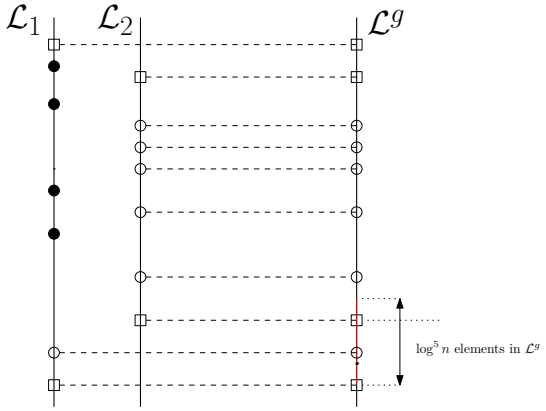

Figure 2: Example of list re-ordering. For simplicity, we assume in this example that each group G_i contains six elements. Only elements of lists $\mathcal{L}_1$ and $\mathcal{L}_2$ and their copies in $\mathcal{L}^g$ are shown. Last elements in each group are shown with squares, other good elements are shown with empty circles. The portion of $\mathcal{L}^g$ shown in red contains $\log^5 n$ elements. Only bad elements, depicted with filled circles, are kept in $\mathcal{L}'_i$ for $i = 1, 2$. Good elements are connected with their copies in $\mathcal{L}^g$ by dashed lines.

method from Lemma 4.1 to segments in $\mathcal{L}_1(v), \ldots, \mathcal{L}_f(v)$. Thus each segment is either a good segment stored in $\mathcal{L}^g(v)$ or a bad segment stored in some $\mathcal{L}'_j(v)$ for $1 \le j \le f$. Let T_v denote the subtree of T with height $\log^\delta n/\log\log n$ rooted in v. Nodes of T_v are nodes $u \in T$, such that v is the lowest macro-node ancestor of u. In order to support queries and updates, we store most segments from $\mathcal{L}(v)$ in the nodes of the subtree T_v. All segments in $\mathcal{L}(v)$ intersect the left vertical boundary ℓ_v of v. We will say in this section that a segment s_1 is larger than s_2 if p_1 is above p_2, where p_i, $i = 1, 2$, is the point where s_i intersects ℓ.

Our method is similar but not equivalent to the segment tree: not all segments that would be stored in a standard segment tree are kept in the nodes of T_v. We will say that a segment s is *relevant* for a node u (resp. a node u is relevant for s) if s spans at least one child of u, but s does not span u. We maintain two lists, $L(u)$ and $L_b(u)$, in a node $u \in T_v$. All segments from $\mathcal{L}^g(v)$ that are relevant for a node u are stored in $L(u)$. If s is a bad segment that is relevant for a node u, then (1) s can be stored in $L(u)$ or (2) s can be stored in one or several lists $L_b(u_j)$, where u_j is a child of u that is spanned by s, or (3) it is possible that s is stored in neither $L(u)$ nor in $L_b(u_j)$. When a new segment s_n is inserted, we attempt to find its position in $L(u)$ for each relevant node u, but we spend $O(\log\log n)$ time in each relevant node. If the proper position of s_n is found, we insert s_n into $L(u)$. In one special case, we also attempt to insert s_n into lists $L_b(u_i)$ for children u_i of a relevant node u. In this case, we will say that the node u is special. We will show that there is at most one special node $u \in T_v$ for each segment inserted into $\mathcal{L}(v)$. Although not all segments from $\mathcal{L}(v)$ are stored in all relevant nodes, we are able to process all "missed" segments when a query is answered.

For every set of bad segments in a group G_i of $\mathcal{L}'_j(v)$, we store a data structure that supports vertical ray shooting queries. All segments in G_i cross the same vertical line, i.e., the left vertical boundary of v. Since G_i contains a poly-logarithmic number of bad segments, both queries and updates can be supported in $O(\log\log n)$ time [13].

Insertions. Each list $L(u)$ and $L_b(u)$ is divided into blocks of $\Theta(\log^5 n)$ segments so that the lowest segment in every block is a good segment. Blocks can be maintained using standard techniques; a more detailed description is provided in Section A.

When a new segment is inserted into $\mathcal{L}(v)$, we distinguish between two cases. If the new segment is in $\mathcal{L}^g(v)$, we insert it into all $L(u)$ for all relevant nodes $u \in T_v$. Since all segments in $\mathcal{L}^g(u)$ are ordered, we can use fractional cascading and find the segment $s'(u) = \text{pred}(s, L(u))$ for all relevant nodes u in $O(\log\log n)$ time per node. We then insert s into $L(u)$ immediately after $s'(u)$ for all relevant u. If s is a bad segment, we identify the segment $plast(s)$ and find the segments $s_p(u) = pred(plast(s), L(u))$ for all relevant nodes u. Using fractional cascading all $s_p(u)$ can be found in $O(\log\log n)$ time per node. Finally we search among $\log^5 n$ segments that follow $s_p(u)$ for the segment $s'(u) = pred(s, L(u))$. If $s'(u)$ is found (i.e., if $s'(u)$ is among the $\log^5 n$ segments that follow $s_p(u)$), we insert s after $s'(u)$. If $s'(u)$ is not found (i.e., if $s'(u)$ is not among the $\log^5 n$ segments that follow $s_p(u)$), we proceed as follows.

We will say that a segment s has *span index* i_s in a node u if s spans the i_s leftmost children $u_1, \ldots, u_{i_s}$ of u. Let $L(u, j)$ denote the set of segments $s \in L(u)$ with span index j. We find $s(u, j) = \text{pred}(s_p(u), L(u, j))$ and search for the predecessor of s among $\log^5 n$ segments that follow $s(u, j)$ in $L(u, j)$. If the predecessor of s in $L(u, j)$ is found, we insert s into $L(u, j)$ and repeat the following procedure for $j = i_s, i_s - 1, \ldots, 1$: We find the segment $s_j(u) = \text{pred}(s_p(u), L_b(u_j))$, i.e., the largest segment $s_j(u) \le s_p(u)$ in $L_b(u_j)$. Next we search among $\log^5 n$ segments that follow $s_j(u)$ for the segment $s'_j(u) = pred(s, L_b(u_j))$. If $s'_j(u)$ is found, we insert s after $s'_j(u)$ into $L_b(u_j)$, decrement j, and repeat the same procedure for the new value of j. We stop when $j = 0$ or when $s'_j(u)$ is not found.

The following lemma bounds the number of lists $L_b(v)$ in T_v where a segment s is stored. A node v is special for segment s if s is stored in a list $L_b(v_i)$ for at least one child v_i of v.

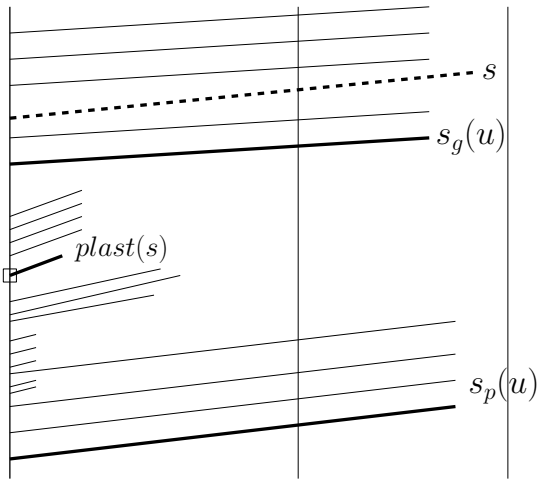

Figure 3: Implicit bad segment s in a list $L(u)$. The leftmost vertical line is the left boundary of v. Two other vertical lines depict the left and the right vertical boundaries of the node u. When we attempt to insert s into $L(u)$, there are at least $\log^5 n$ segments between $s_p(u)$ and s in $L(u)$ (only a part of segments stored in $\mathcal{L}^g(v)$ and $L(u)$ are shown).

Lemma 5.1. *For every segment $s \in \mathcal{L}(v)$ there is at most one special node $u \in T_v$.*

Proof. Let π_s denote the path from the root of the subtree T_v to the leaf of T_v that contains the right endpoint of a segment s. The i-signature of a segment s consists of span indices of the i highest nodes on π_s (listed in top-to-bottom order). If a segment s does not span any children of a node u, then the span index of s in u is 0. An i-signature uniquely specifies the path to a node u of depth i in T_v. Therefore all segments relevant for a node u, such that the depth of u in T_v is equal to r, have the same $(r-1)$-signature.

Suppose that a bad segment $s \in \mathcal{L}_j(v)$ is inserted. By a slight misuse of notation, we will say that a node u is relevant if it is relevant for the segment s. For a relevant node u on the path π_s we define $S(u)$ as the set of segments $s' \in L(u)$, such that $plast(s) < s' \le s$. All segments in $S(u)$, where u is a depth-r node, have the same $(r-1)$-signatures. Hence $S(w) \subset S(v)$ for any two relevant nodes w and v on π_s such that w is a descendant of v. As follows from the insertion procedure we insert a bad segment into $L(u)$ if and only if $S(u)$ contains at most $\log^5 n$ segments.

Consider a relevant node v such that $S(v)$ contains more than $\log^5 n$ segments. Suppose further that the span index of s in v is equal to l and $S(v)$ contains less than $\log^5 n$ segments with span index l.

Since $S(v) \subseteq S(w)$ for any ancestor w of v, all $S(w)$ contain more than $\log^5 n$ segments. Hence s is inserted into neither $L(v)$ nor $L(w)$ for any ancestor w of v. There are at most $\log^5 n$ segments in $S(v)$ that have the same r-signature as s. Hence for any descendant w' of v, $S(w')$ contains at most $\log^5 n$ segments. Hence s is inserted into $L(w')$ for all relevant descendants w' of v.

□

Thus a new segment s, inserted into $\mathcal{L}(v)$, is inserted into $O(\log^\delta n/\log\log n)$ lists $L(u)$ where each node u is on a path in T_v and $O(\log^\varepsilon n)$ lists $L_b(v_i)$ where all v_i are sibling nodes. Since we spend $O(\log\log n)$ time in each node, the total insertion time is $O(\log^\delta n)$.

As already mentioned, we divide the lists $L(u)$ and $L_b(u)$ into blocks of $\Theta(\log^5 n)$ consecutive segments and guarantee that the lowest segment in each block is a good segment. The maximum block size in all lists does not exceed $(1/4)\log^5 n$. Lists $L(u, j)$ are divided into blocks in the same way.

Queries. If a bad segment s is stored in $L(u)$ (or $L_b(u)$) we will say that s is a *bad explicit* segment in $L(u)$ (resp. $L_b(u)$). If s is not stored in $L(u)$ we will say that s is a *bad implicit* segment in u. We observe that not all relevant segments are stored in $L(u)$. Nevertheless vertical ray shooting queries can be answered correctly. The following facts outline the strategy that will be used to answer a query.

Lemma 5.2. *Suppose that a segment s, with span index $l \ge f$, is a bad implicit segment in $L(u)$ and a bad implicit segment in $L_b(u_f)$. Then there is at least one good segment $s_g(u) \in L(u)$ with span index $j \ge f$, such that $plast(s) < s_g(u) < s$.*

Proof. Suppose that s is a bad implicit segment in $L(u)$. Consider the time when we attempted to insert s into $L(u)$. Then there are more than $\log^5 n$ segments between $s_p(u) = pred(plast(s), L(u))$ and s in $L(u)$. Additionally one of the two following conditions is satisfied: (i) there are more than $\log^5 n$ segments with span index l between $s_p(u) = pred(plast(s), L(u))$ and s or (ii) there are more than $\log^5 n$ segments in between $s_f(u) = \text{pred}(s_p(u), L_b(u_f))$ and s in $L_b(u_f)$. In both cases one segment between $s_p(u)$ and s (resp. between $s_f(u)$ and s) is a good segment: there is at least one block B of $L(u, l)$ (resp. $L_b(u_f)$), such that all segments in B are larger than $plast(s)$ and smaller than s. The good segment $s_g(u)$ in B has span index $j \ge f$ and satisfies $plast(s) < s_g(u) < s$. Hence the statement of this lemma is true when a segment s is inserted. See Fig. 3.

When two blocks in $L_b(u_f)$ or $L(u, l)$ are merged, the resulting block does not contain any implicit segments. When two blocks in

$\mathcal{L}^g(v)$ are merged or split, the resulting block does not contain any bad segments. See Section A. Hence there is always a good segment between an implicit bad segment s and $plast(s)$ in $L_b(u_f)$. □

Lemma 5.3. *Suppose that $s = \mathtt{ray}(q, \mathcal{L}(v))$ is a bad implicit segment and let $s_g = \mathtt{ray}(q, \mathcal{L}^g(v))$. Then $plast(s) < s_g \le last(s)$ where $last(s)$ is the largest segment in the same group G_i as s and $plast(s)$ is the largest segment in the group preceding G_i.*

Proof. Suppose that q is contained in the slab of a node u and that $s = \mathtt{ray}(q, \mathcal{L}(v))$ is a bad implicit segment in u. Suppose also that q is contained in the slab of u_f for a child u_f of u. The segment s spans u_f. By Lemma 5.2, there is a good segment $s' \in L(u) \cup L_b(u_f)$ such that $s' > plast(s)$, $s' < s$, and s' spans u_f. If there is more than one such segment, let s' be the largest one. If $s_g \le plast(s)$, then $s_g < s' < s$. Since s' is below q, $s_g \neq \mathtt{ray}(q, \mathcal{L}^g(v))$. If $s_g \ge last(s)$ then $s_g > s$. Hence s_g is not below q and $s_g \neq \mathtt{ray}(q, \mathcal{L}^g(v))$. In both cases we obtain a contradiction. □

Now a query procedure can be implemented as follows. We visit all nodes $u \in T_v$ such that the vertical slab of u contains q and find $s_e(u) = \mathtt{ray}(q, L(u))$ in every visited node. We will show in Section D how vertical ray shooting queries on $L(u)$ can be supported in average time $O(\log\log n)$ per node. We re-visit all nodes $u \in T_v$ such that the vertical slab of u contains q and find $s_b(u) = \mathtt{ray}(q, L_b(u))$ in every visited node. This can be done using the same method as in Section D or using the standard fractional cascading. For each $s_e(u)$, we find the largest good segment $s_g(u) \le s_e(u)$; for each $s_b(u)$, we find the largest good segment $s'_g(u) \le s_b(u)$. Using a USF data structure, all $s_g(u)$ and $s'_g(u)$ can be found in $O(\log\log n)$ time per node. Let s^g be the largest segment among $s_g(u)$ and $s'_g(u)$, $u \in T_v$. Next for all j, $1 \le j \le \log^\varepsilon n$, we identify the group G_{i_j} of $\mathcal{L}_j(u)$ that contains s^g and answer the vertical ray shooting query on that group. Finally we select the largest segment among all segments $s_e(u)$, all segments $s_b(u)$, and all $s_j = \mathtt{ray}(q, G_{i_j})$ for $1 \le j \le \log^\varepsilon n$. If $s = \mathtt{ray}(q, \mathcal{L}(v))$ is a good segment or a bad explicit segment in some $L(u)$ or $L_b(u)$, then s is the largest segment among all $s_e(u)$ and $s_b(u)$. If s is a bad implicit segment in some $L(u)$ or $L_b(u)$, then s is contained in some group G_{i_j} by Lemma 5.3. Hence s is the largest of all s_j in the latter case.

Deletions. We need to address several technical issues in order to support deletions. We divide segments in $L(u)$ and $L_b(u)$, $u \in T_v$, into blocks and update the blocks regularly in order to maintain the property specified in Lemma 5.2. Additionally some good segments are kept in lists $L(u)$ after a deletion. In order to handle the segments that are marked as deleted, but still stored in $L(u)$ (resp. $L_b(u)$), we have to slightly modify our search procedure. Details are provided in Section A.

6 MIDDLE SEGMENTS IN A TREE

Every middle segment in the macro-tree must be inserted into $O(\log^\delta n)$ nodes of the base tree. We cannot afford to spend $O(\log n)$ time in each node where a new segment is inserted. Fortunately it is not necessary to order all segments stored in a node u. The solution is based on the combination of two techniques: segment categorization [8] support fast updates but require higher query time; we can modify this approach so that ray shooting queries are answered in selected nodes only. Weighted telescoping search [25] supports queries efficiently, but has higher update time. In order to obtain the optimal-time solution, we modify the weighted telescoping search approach by truncating the weighted search trees. The middle segments are stored in two independent data structures that are described below.

6.1 Truncated Weighted Trees

We keep all middle segments in the segment tree that supports weighted telescoping search [25]. A segment is stored in at most one node of the macro-tree as a middle segment. We keep each segment from $\mathcal{M}(v)$ in $O(\log^{\delta+\varepsilon} n)$ nodes of the base tree. If a segment s spans a node u of T_v, but s does not span the parent of u, then we store s in the list $M(u)$. All segments stored in a list $M(u)$ for each $u \in T$ are divided into *chunks*. We construct a separate weighted tree $\mathbb{T}_i(u)$ for each chunk. All segments in the list $M(u_R)$ of the root node u_R are kept in one chunk. Hence there is one weighted tree $\mathbb{T}(u_R)$. The chunks in a child u_k of a node u correspond to leaves in $\mathbb{T}_i(u)$: every leaf ℓ of $\mathbb{T}_i(u)$ corresponds to an interval of segments $[s_{\min}(\ell), s_{\max}(\ell)]$; if there are segments $s \in M(u_k)$, such that $s_{\min}(\ell) < s < s_{\max}(\ell)$, then there is a chunk of $M(u_k)$ for the leaf ℓ. We can assign weights to leaves of all trees in such way that the total weight of $\mathbb{T}(u_R)$ for the root node u_R is polynomial in n.

We modify this general approach of [25] so that the height of the weighted tree for each chunk is bounded by $O(\log^\delta n)$. Segments with weights less than $\Theta(\frac{1}{2^{\log^\delta n}})$-fraction of the total tree weight are lumped together in one leaf. We can insert a new segment into truncated segment tree in $O(\log^\delta n)$ time and remove a segment from a truncated tree in the same time. For a new segment s, we can identify all $O(\log^\delta n)$ nodes u where it should be stored in $M(u)$. We can also find all chunks where s must be stored in $O(\log n)$ time using the following procedure.

Let u_t denote the highest node such that s must be inserted into $M(u_t)$. We can find the chunk where s must be inserted in $O(\log n)$ time. Let T_μ denote the subtree rooted in a macro-node μ, such that T_μ contains u_t. Let π_r and π_l denote the path from u_t to the child of μ in the macro-tree that contains the right (left) endpoint of s. If s must be stored in $M(v)$ then v is a child of some node on π_r or π_l. We visit all nodes u on π_r, starting with u_t, and all children of these nodes where s must be inserted. When we visit a node u, we already know the chunk and the tree $\mathbb{T}_{j(u)}(u)$ where s must be inserted. We find the appropriate leaf ℓ_u where s must be stored. Then we continue in the children of u. In every child u_l of u where s must be inserted, we insert s into a leaf of $\mathbb{T}_{j(u_l)}(u_l)$, where $\mathbb{T}_{j(u_l)}(u_l)$ is the weighted tree of the chunk corresponding to ℓ_u in a child u_l of u. Nodes on the path π_l are processed in the same way. We visit $O(\log^{\delta+\varepsilon} n)$ nodes and spend $O(\log^\delta n)$ time in each node, except for u_t. The total time for an insertion is $O(\log n + \log^{2\delta+\varepsilon} n) = O(\log n)$.

When the total weight of a group leaf exceeds $\frac{2W}{2^{\log^\delta n}}$, we split the set of segments stored in this leaf into two parts of approximately equal weight. The splitting procedure takes $O(\log^3 n)$ time and

relies on the representation of segments, described in Section 6.2. This procedure is described in Section B.

6.2 Slow Searching among Middle Segments

We will also need the following simple modification of the method used in [7].

Let $M_k(u)$ denote the set of segments with category k in a node u; segment categories are defined in Section C. The set $AM_k(u) \supset M_k(u)$ is the superset of $M_k(u)$ augmented with segments from sets $M_k(w)$, where w is an ancestor of u. The set $AM_k(u)$ is used to support fractional cascading; we refer to [8] for a detailed description. All $AM_k(u)$ are divided into blocks of $\Theta(\log^2 n)$ segments. We maintain a set $AM'_k(u)$ that contains one selected segment from each block of $\Theta(\log^2 n)$ segments in $AM_k(u)$ and a set $AM'(u) = \cup_k AM'_k(u)$. All segments of $AM'(\cdot)$ are kept in a data structure supporting colored predecessor queries: given a segment $s \in AM'(\cdot)$ and a color k, we can find the largest $s \in AM'_k(\cdot)$ such that $s' \leq s$ in $O(\log\log n)$ time [24]. Since every segment s is assigned $O(1)$ different categories, each s can be inserted into $M_k(u)$ for all relevant nodes u in $O(\log n)$ time. The cost of maintaining $AM'(u)$ is $O(1)$ time per inserted or deleted segment. A ray shooting query is answered in two stages.

`Stage 1.` We visit all nodes that contain the query point q in the bottom-to-top order. For each visited node u we identify the smallest segment above q and the largest segment below q in $AM'(u)$.

`Stage 2.` Let u be a node that contains q and let the segment s_u be the segment immediately below q in $AM'(u)$. For every category k, we find the largest segment $s'_k \in M'_k(u)$, such that $s'_k \leq s_u$. Using the colored predecessor search data structure, we can find s'_k in $O(\log\log n)$ time. Finally we search for the largest segment $s_k \in AM_k(u)$ such that s_k is below q. By definition, s_k is among $\Theta(\log^2 n)$ segments that follow s'_k in $AM_k(u)$. Using finger search on $AM_k(u)$, s_k can be found in $O(\log\log n)$ time. Finally $s_o = \mathtt{ray}(q, AM(u))$ is the largest segment among all s_k. The total time to find s_o in each node u is $O(\log^{\varepsilon} n \log\log n)$.

The total cost of a query is $O(\log^{1+\varepsilon} n \log\log n)$. However, this modification has one interesting property: searches in nodes u during the second stage are executed independently of each other. In our method, that combines data structures from Sections 6.1 and 6.2, we will execute the second stage for only a small number of nodes.

6.3 Queries on Middle Segments in a Tree

A ray shooting query on middle segments can be answered as follows. We execute Stage 1 and find $\mathtt{ray}(q, AM'(u))$ for all nodes u that contain q. Then we run the weighted search starting at the root node. For every node u, we look for the segment s_u immediately below q in $AM(u)$ using the truncated weighted tree. If the search is finished and s_u is found, we move down to the child of u that contains q. Otherwise, we say that u is a *difficult* node. If u is a difficult node, we find s_u using the algorithm of Stage 2.

Let $u_1, u_2, \ldots, u_h$ denote the sequence of nodes that contain the query point q. Suppose that we search in a truncated weighted tree $\mathbb{T}_{u_i}$ when we visit the node u_i. Let $W(\mathbb{T}_{u_i})$ denote the total weight of the tree $\mathbb{T}_{u_i}$ and let $w(u_i, s)$ denote the weight of the leaf where the search in $\mathbb{T}_{u_i}$ is finished. We spend $O(\log \frac{W(\mathbb{T}_{u_{i+1}})}{w(u_i,s)} + \log\log n)$ time in each node u_i and $\frac{w(u_i,s)}{W(\mathbb{T}_{u_{i+1}})} = O(\log^4 n)$; see Section B. Hence, ignoring the time we spend in difficult nodes, the total time is $O(\log W(\mathbb{T}_{u_1}) + \log n) = O(\log n)$.

The number of difficult nodes is $O(\log n/\log^{\delta} n)$. Since we spend $O(\log^{\varepsilon} n \log\log n) = O(\log^{2\varepsilon} n)$ additional time in every difficult node, the total cost of finding s_u in all difficult nodes is bounded by $O(\log^{1+2\varepsilon-\delta} n) = o(\log n)$ time. The total cost of finding s_u in all nodes is $O(\log n)$.

7 PUTTING ALL PARTS TOGETHER

Using the data structures from Sections 5, 6, and D, we can answer a vertical ray shooting query on a dynamic set of segments. Let π_q denote the set of all nodes u that contain the query point q. First, we find $s_1(u) = \mathtt{ray}(q, L(u))$ for all nodes u in π_q. As shown in Section D, this step takes $O(\log n)$ time. Next, we examine all macro-nodes v that contain q. As explained in Section 5, we can answer a query $\mathtt{ray}(q, \mathcal{L}(v))$, provided we know $\mathtt{ray}(q, L(u))$ for all $u \in T_v$ such that u contains q. This step takes $O(\log^{\delta} n)$ time per node v or $O(\log n)$ time in total. Queries on right segments are answered in the same way. Finally, we answer queries $\mathtt{ray}(q, M(u))$ for all u that contain q. This step also takes $O(\log n)$ time as described in Section 6. Hence the total query time is $O(\log n)$.

Since each segment can be stored in $O(\log n)$ nodes, our data structure uses $O(n \log n)$ space. The space usage can be reduced to linear using the method of Chan and Nekrich [8, Section 3.2]. In our description we assumed for simplicity that the x–coordinates of segment endpoints are fixed. We can get rid of this assumption by implementing the base tree T as a weighted B-tree [2].

Theorem 7.1. *There exists an $O(n)$-space data structure that supports point location queries in $O(\log n)$ time and updates in $O(\log n)$ time.*

A DELETIONS ON LEFT MACRO-SEGMENTS

In this section we explain how the method from Section 5 is modified to support deletions. We also provide details of maintaining blocks in this section.

We divide each set $L(u)$ into blocks of $\Theta(\log^5 n)$ consecutive segments. A block can contain many good segments, but there is exactly one good segment in each block that is called the *anchor* segment. When an anchor segment s_a is deleted, we mark s_a as deleted and keep it in $L(u)$. For each block, we count the number b_u of updates and run the following background process: we identify the block B with the largest value of b_u; if the anchor segment in B is marked as deleted, we remove it from $L(u)$; next, we select the smallest segment s_B in B and make s_B a good segment. That is, we insert the segment s_B into $\mathcal{L}^g(v)$, and insert s_B into all appropriate nodes u. Finally we declare s_B to be the anchor segment. The new anchor segment is computed in $O(\log n)$ time. Additionally we *purge* the block B. The purging procedure finds all segments s, such that s is an implicit bad segment in the node u and $l_1 < s < l_2$ where l_1 and l_2 are the smallest and the largest segments in B. We turn every such segment s into a good segment; each s is inserted into $\mathcal{L}^g(v)$ and into lists $L(v)$ for all relevant nodes $v \in T_v$. We can find all such segments s in $O(\log n)$ time per segment. The

number of affected bad implicit segments is $O(\log^{2+\varepsilon} n)$: consider a bad segment s in $\mathcal{L}_j(v)$ such that s is relevant for u and $l_1 < s < l_2$. If $plast(s) > l_1$, then s is a bad explicit segment because there are $O(\log^5 n)$ segments between s and the predecessor of $plast(s)$ in $L(u)$. There is only one group G in each $\mathcal{L}_j(v)$, such that $plast(s) < l_1 < last(s)$. Since each group contains $O(\log^2 n)$ segments, there are $O(\log^{2+\varepsilon} n)$ bad implicit segments s, such that $l_1 < s < l_2$ and s is relevant for $L(u)$. The total time needed to purge a block is $O(\log^{3+\varepsilon} n)$.

By [17, Theorem 5], the anchor segment of a block is re-computed after $O(\log^{4+\varepsilon} n)$ updates of a block. We can choose the constants in such way, that the anchor segment is re-computed and a block is purged after less than $(\log^{4+\varepsilon} n)/4$ updates in a block. We also guarantee that the maximum block size is $(1/4)\log^5 n$ and the minimum block size is $(1/12)\log^5 n$ by splitting large blocks and merging a small block with its neighbor in a standard way. Lists $L_b(u)$ and lists $L(u,k)$ are maintained in the same way.

We also divide the global list $\mathcal{L}^g(v)$, that contains all good segments, into blocks of $\Theta(\log^3 n)$ consecutive elements. When a block of $\mathcal{L}^g(v)$ is updated $\log^2 n$ times, we erase the deleted segments from that block: if a segment $s_a \in \mathcal{L}_j(v)$ is marked as deleted, we identify the predecessor s_b of s_a in $\mathcal{L}_j(v)$, delete s_a, and insert s_b into $\mathcal{L}^g(v)$. Since each block intersects at most two groups of $\mathcal{L}'_j(v)$ for any fixed j, each block contains at most two anchor segments from $\mathcal{L}_j(v)$ for any fixed j. Hence each block contains at most $O(\log^\varepsilon n)$ anchor segments in total. We need $O(\log n)$ time to update one anchor segment and all anchor segments in a block of $\mathcal{L}^g(v)$ can be updated in $O(\log^{1+\varepsilon} n)$ time. By Theorem 5 in [17] each block is re-built after $O(\log^{2+\varepsilon} n)$ updates. We can choose constants in such a way that every block is updated after at most $(\log^3 n)/3$ segment insertions and deletions.

When a new segment s is inserted, we find the segment $plast(s)$ and the largest segment $s' \le plast(s)$ in $\mathcal{L}^g(v)$ that is not marked as deleted. Next we find the segment $s_p(u) = pred(s', L(u))$ in each relevant node $u \in T_v$. Then we proceed with insertion as described in Section 5. If $pred(s, L(u))$ is among $\log^5 n$ segments that follow $s_p(u)$ in $L(u)$, then we insert s into $L(u)$. If $pred(s, L(u))$ is among $\log^5 n$ segments that follow $s_p(u)$ in $L(u,l)$, where l is the span index of s, we insert it into lists $L_b(u_j)$ for some children u_j of u, $1 \le j \le l$. When a segment s is deleted, we remove it from all lists $L(u)$ and $L_b(u)$, such that s is stored in $L(u)$ (resp. in $L_b(u)$), but s is not an anchor segment in $L(u)$ ($L_b(u)$). If s is removed from all nodes $u \in T_v$, we also remove it from $\mathcal{L}^g(v)$.

A query is answered in almost the same way as described in Section 5. We visit all nodes $u \in T_v$, such that the slab of u contains the query point q. In every visited node we find $s_e(u) = \mathtt{ray}(q, L(u))$ and $s_b(u) = \mathtt{ray}(q, L_b(u))$. Both $s_e(u)$ and $s_b(u)$ can be good segments that are marked as deleted. Let $s_r(u)$ be the largest non-deleted segment in $L(u)$ such that $s_r(u) \le s_e(u)$ and let $s'_r(u)$ be the largest non-deleted segment in $L_b(u)$ such that $s'_r(u) \le s_b(u)$. We also find the largest good segment $s_g(u) \le s_r(u)$ in $L(u)$ and the largest good segment $s'_g(u) \le s'_r(u)$ in $L_b(u)$. We identify the largest segment s^g among all $s_g(u)$ and $s'_g(u)$, $u \in T_v$. For every j we find the group G_{i_j} of $\mathcal{L}'_j(v)$, such that $plast(G_{i_j}) < s^g \le last(G_{i_j})$ and answer the vertical ray shooting query on that group. Finally we select the largest segment among all segments $s_r(u)$, $s'_r(u)$ and $s_j = \mathtt{ray}(q, G_{i_j})$, $1 \le j \le \log^\varepsilon n$.

If the answer to a query is a good segment or a bad explicit segment, then the answer is the largest segment among all $s_r(u)$ and $s'_r(u)$. If the answer to a query is a bad implicit segment s, then this segment is contained in one of the groups G_{i_j}: Suppose that s spans a node u. As follows from the proof of Lemma 5.2, there are at least two blocks in $L(u)$ (resp. $L_b(u)$) such that all segments in these blocks are between $plast(s)$ and s. Hence one of the segments $s_r(u)$ and $s'_r(u)$ is larger than $plast(s)$; this segment is also smaller than $last(s)$. And at least one of $s_g(u)$ and $s'_g(u)$ is also between $plast(s)$ and $last(s)$. Hence s is in one of the groups G_{i_j}, such that $plast(G_{i_j}) < s^g \le last(G_{i_j})$.

B WEIGHTED TELESCOPING SEARCH ON $M(u)$

We explain how weighted search is implemented on lists $M(u)$, $u \in T$. Weighted search on $L(u)$ is implemented in a similar way. Let $M_j(u)$ denote the list of category-j segments stored in a node u. First, we explain how the weighted search trees $\mathbb{T}_i(u)$ can be constructed when all lists $M_j(u)$ are known. We construct the list $M(u) = \cup_j M_j(u)$ for each node u by merging the lists $M_j(u)$. Next we create augmented lists $AM(u)$, such that $M(u) \subseteq AM(u)$ and each segment from $AM(u) \setminus M(u)$ is a copy of a segment stored in $M(v)$ for some ancestor v of u. Lists $AM(u)$ satisfy the following properties: if a node v is a parent of a node w, then there are $O(\log^4 n)$ segments of $AM(v)$ between any two consecutive segments of $AM(v) \cap AM(w)$; second, every segment from $M(v)$ is stored in at most two lists $AM(u)$ where v is an ancestor of u. Augmented lists with these properties are used in fractional cascading; see e.g.,[1] or [8]. We refer to [8] for an explanation how $AM(u)$ can be constructed and maintained.

Next we assign weights to segments in all lists $AM(u)$ as follows. All segments stored in the leaf nodes are assigned weight 1. To compute the weight of a segment s in $AM(u)$ for an internal node u, we identify the preceding and the following down-bridges to a child u_i and compute the total weight of all segments between these down-bridges in u_i. The sum over all children u_i of u divided by $\log^2 n$ is the weight assigned to s. To be precise, let $DOWN(u, u_i) = AM(u) \cap AM(u_i)$ and let $W_i(s) = \sum \mathtt{weight}(s')$ where the sum is computed over all segments s' in $AM(u_i)$ satisfying $l_i < s' < h_i$, l_i is the largest segment in $DOWN(u, u_i)$ that is smaller than s, and h_i is the smallest segment in $DOWN(u, u_i)$ that is larger than s. Then $\mathtt{weight}(s) = (\sum_i W_i(s))/\log^4 n$ where the sum is over all children u_i of u.

When we know the weights of all segments in a tree, we construct the weighted trees $\mathbb{T}_i(u)$. For every chunk, starting with the chunk at the root node, we proceed as follows. First we compute the total weight W of all segments $s \in AM(u)$ in the chunk. Next we divide the chunk into subsets of consecutive segments C_i so that: (i) if a chunk C_i contains more than one segment, then the weight of C_i is at most $2W/d$; (ii) if the weight of a chunk C_i is less than W/d, then both C_{i-1} and C_{i+1} have weight at least $2W/d$. Here $d = 2^{\log^\delta n}$.

When subsets of a chunk and their weights are known, we create a weighted search tree, such that the weight of the i-th leaf is the weight of C_i. We can construct and maintain a tree in such a way

that the depth of the leaf corresponding to C_i is $O(\log \frac{W}{w(C_i)})$ where $w(C_i)$ is the total weight of all segments in the chunk C_i. If C_i contains more than one segment, then the i-th leaf is a pseudo-leaf.

When the leaves of $\mathbb{T}_k(u)$ in a node u are known, we create the trees in the children u_j of u. Leaves of each tree $\mathbb{T}_k(u)$ for $k = 1, 2, \ldots$ are visited in the left-to-right order. For each leaf C_i we find the successor $s_{\max}(C_i)$ of $\max(C_i)$ in $DOWN(u, u_j)$ and the predecessor $s_{\min}(C_i)$ of $\min(C_i)$ in $DOWN(u, u_j)$, where $\max(C_i)$ and $\min(C_i)$ denote the smallest and the largest segments in C_i (unless C_i is a pseudo-leaf, $\min(C_i) = \max(C_i)$). The set of segments in $M(u_j)$ between the copies of $s_{\min}(C_i)$ and $s_{\max}(C_i)$ is a chunk of $M(u_j)$. For each chunk, we identify the leaves and pseudo-leaves and construct the weighted search tree. We proceed in the same manner and construct chunks and weighted trees for all nodes of T.

Using truncated weighted telescoping search, we can answer ray shooting queries in all nodes u that contain the query point q. Suppose that we already know the leaf C_W of the weighted tree $\mathbb{T}_{i_w}(w)$ that contains the segment $s_a(w) = \mathtt{ray}(q, AM(w))$ in the parent w of u. We move to the chunk that is bounded by $s_{\min}(C_W)$ and $s_{\max}(C_W)$. Then we search in the weighted tree $\mathbb{T}_{i_u}(u)$ for the segment that is immediately below q or for the pseudo-leaf C_u that contains $s_a(u) = \mathtt{ray}(q, AM(u))$. The time we spend in a node u is $O(\log \frac{W(\mathbb{T}_{i_u}(u))}{w(u)})$. If the leaf C_u is not a pseudo-leaf, then we have found the segment $s' = \mathtt{ray}(q, AM(u))$. The segment $s'' = \mathtt{ray}(q, M(u))$ is the largest segment in $M(u)$ that is smaller than s'. We can find s'' in $O(\log \log n)$ time using a USF data structure.

Insertions and Deletions. Now we explain in more detail how weighted trees on chunks and updates of these trees are implemented. Let W denote the total weight of a chunk. Following [20], we define approximate weights of leaves. Let $\tau = \lceil \frac{W}{2d} \rceil$ where $d = 2^{\log^\delta n}$. Let w_i denote the weight of the i-th leaf ℓ_i. Then $w'_i = \lceil \frac{w_i}{d} \rceil$ is the approximate weight of ℓ_i. We maintain a weighted search tree for a chunk, so that the depth of ℓ_i is $O(\log(W'/w'_i))$ where $W' = \sum_i w'_i$ is the total approximate weight of all leaves using e.g., the method of [20].

Since $W' = \Theta(d)$ and $w'_i \geq 1$ for all i, every leaf has depth bounded by $O(d)$. Since $W' \cdot \tau \leq 2W$ and $w'_i \cdot \tau \geq w_i$, we have $\frac{W'}{w'_i} \leq \frac{2W}{w_i}$ and $\log(W'/w'_i) \leq \log(W/w_i) + O(1)$. Thus the depth of every leaf is bounded by $O(\min(d, \log(W/w_i)))$.

An insertion procedure for a new segment s, that is described in Section 6.1, identifies the leaf ℓ that stores $pred(s, M(u))$. If ℓ is a pseudo-leaf, we add s to ℓ. Otherwise, let ℓ' denote the leaf that is the right neighbor of ℓ. If ℓ' is pseudo-leaf, we add s to ℓ'. If both ℓ and ℓ' are pseudo-leaves, we insert a new leaf ℓ'' between ℓ and ℓ'. The weight of ℓ'' is set to 1.When a new leaf is inserted into $\mathbb{T}$, we also divide the chunk stored in a child of u.

Representation of Pseudo-Leaves. All segments in the pseudo-leaf can be classified as in Section 6.2. Let $\mathcal{G}_k(\ell)$ denote the list of segments with category k in the leaf ℓ. Since we know the order of all segments with a fixed category we know the smallest and the largest segment in $\mathcal{G}_k(\ell)$. We can represent $\mathcal{G}_k(\ell)$ as a sub-list of $AM_k(\ell)$. When a pseudo-leaf is split, we can divide it into two pseudo-leaves of almost equal weight by a binary search in $\mathcal{G}_k(\ell)$.

Splitting and Merging Trees $\mathbb{T}_k(u)$. When a new down-bridge $s \in AM(u) \cap AM(u_i)$ is inserted into some tree $\mathbb{T}_j(u)$, we split the corresponding tree $\mathbb{T}_k(u_i)$ into two trees $\mathbb{T}'_k(u_i)$ and $\mathbb{T}''_k(u_i)$. The cost of splitting two trees is bounded by their height; hence, the cost of splitting $\mathbb{T}_k(u_i)$ into $\mathbb{T}'_k(u_i)$ and $\mathbb{T}''_k(u_i)$ is bounded by $O(\log^\delta n)$.

C FRACTIONAL CASCADING AND SEGMENT CATEGORIES ON $M(u)$

We assign categories to segments of $M(u)$ using the following scheme. Each segment $s \in \mathcal{M}(v)$ is stored in three different structures: if s crosses the left vertical boundary of a node $u \in T_v$, then we keep s in the list $M^{(l)}(u)$ of the left middle segments. If s crosses the right vertical boundary of u, we keep s in the list $M^{(r)}(u)$ of the right middle segments. Finally if s crosses neither the left nor the right boundary of u, we keep it in the list $M^{(m)}(u)$ of the middle middle segments.

Consider a segment $s \in M^{(l)}(u)$. We identify the highest j such that s crosses the left vertical boundary of some node w and the height of w divides $\log^{j\varepsilon} n$. Next we find the highest k, such that s crosses the left vertical boundary of some node w' with height $k \cdot \log^{j\varepsilon} n$. The category of s is the pair (j, k). Since $0 \leq j < \lceil 1/\varepsilon \rceil$ and $1 \leq k < \log^\varepsilon n$, there are $O(\log^\varepsilon n)$ different categories. All segments in a category (j, k) cross the same vertical line; hence all segments in a list $M(u)$ with the same category (j, k) are elements of an ordered set. We can assign categories to segments in $M^{(r)}(u)$ in the same way.

Now consider a middle middle segment $s \in M^{(m)}(u)$. We assign a category l to s, where u_l is the leftmost sibling of u that is spanned by s. All segments are stored in a colored USF data structure of Mortensen [24]. We assign[3] a color r to a segment $s \in M^{(m)}_l(u_l)$ if u_r is the rightmost child of u spanned by s. In other words a segment $s \in M^{(m)}_l(u)$ with color r spans children $u_l, \ldots, u_r$ of s. For any $k > l$, $M^{(m)}_l(u_k) \subseteq M^{(m)}_l(u_l)$; $M^{(m)}_l(u_k)$ consists of segments in $M^{(m)}_l(u_l)$ with color $j \geq k$. Using the data structure of Mortensen [24], we can find, for any $s \in M^{(l)}(u)$, the largest segment $s' \in M^{(l)}(u)$ with color $j \geq k$ such that $s' \leq s$ (i.e., the largest segment $s' \leq s$ in $M^{(l)}(u)$ that spans u_k); insertions and deletions of segments into $M^{(l)}(u)$ are also supported. Insertions, deletions, and queries on a colored USF data structure take $O(\log \log n)$ time [24].

Consider a middle middle segment s in that spans children $u_l, \ldots, u_r$ of some node u. We can insert a new segment s into $M^{(m)}_l(u_l)$ in $O(\log n)$ time. Using a colored USF data structure [24], we can insert s into $M^{(m)}_l(u_j)$, for all right siblings u_j of u_l such that s spans u_j, in $O(\log \log n)$ time per node. Thus the total insertion time is $O(\log \log n \cdot \log^\varepsilon n + \log n) = O(\log n)$. When a segment is deleted, we delete it from every list $M^{(m)}_j(u)$ and update the corresponding USF data structure.

[3] We observe that colors are not synonymous with categories in this section.

Every list $M(u)$ is a union of $O(\log^\varepsilon n)$ lists $M_j^{(l)}(u)$ of middle left segments, $O(\log^\varepsilon n)$ lists $M^{(r)}(u)$ of middle right segments, and $O(\log^\varepsilon n)$ lists $M_j^{(m)}(u)$ of middle middle segments. Thus all segments in $M(u)$ can be assigned $O(\log^\varepsilon n)$ categories. To avoid tedious notation, we denote all segments in $M(u)$ with the same category by $M_k(u)$.

D DATA STRUCTURES FOR LISTS $L(u)$

The order of segments in a list $L(u)$ is already known. However each segment is inserted into up to $O(\log n/\log\log n)$ lists $L(u)$. We can therefore spend only $O(\log\log n)$ time in a node u when $L(u)$ is updated. Again we use two different data structures to represent segments in every $L(u)$. Our first approach is based on weighted telescoping search. All lists $L(u)$, $u \in T$, are divided into chunks. We assign weights to chunks as sketched in Section 6.1. We store a truncated weighted tree with node degree $\log^\varepsilon n$ and height $\log^{2\varepsilon} n$ for each chunk. Our second approach is based storing all segments in a data structure that answers queries in $O(\log^\varepsilon n)$ time per node using fractional cascading.

D.1 Truncated Weighted Search on $L(u)$

Recall that a span index of a segment s in a node u is the maximal j such that s spans the j-th child u_j of u. Since we store segments with different span indices in the nodes of the weighted tree, we would need $O(\log^{2\varepsilon} n)$ time to update a tree in $L(u)$ after a segment insertion or deletion. In order to reduce the update time, we combine two techniques: assigning small integer labels to segments and bufferization of updates. Both techniques were used separately in numerous data structures. Recently, Chan and Tsakalidis [9] combined both techniques to improve the update time of three-sided range searching. Our approach is similar, but we need to make some modifications because we apply bufferization to an unbalanced weighted tree.

We assign integer labels to all segments that are used to guide the search. We also assign labels to every inserted or deleted segment. Finally we assign labels to $\log^2 n$ largest and $\log^2 n$ smallest segments in every tree leaf. Each segment label is a positive integer bounded by $O(2^{2\log^{2\varepsilon} n})$; if $s_1 < s_2$, then the label $lab(s_1)$ of the segment s_1 is smaller than the label $lab(s_2)$ of the segment s_2. After each sequence of $2^{2\log^{2\varepsilon} n}$ updates, we re-build the labeling scheme from scratch.

We assign a unique integer id to every segment s that is assigned a label, so that $id(s) < 2^{\log^{2\varepsilon} n}$ for each segment s. We store a look-up table Tbl that contain pointers to all segments with labels, $Tbl[i]$ points to a segment s with $id(s) = i$. For every labeled segment s in $L(u)$ we also store $id(s)$. Unlike segment labels, segment ids are not monotonous: if $s_1 < s_2$, it is possible that $id(s_1) > id(s_2)$. When a segment is assigned an id, it does not change (except for the case when the labeling is re-built from scratch). When a segment s with $id(s) = x$ is moved to a leaf node ℓ (i.e., when the buffer in the parent of ℓ is flushed), we move the segment $s = Tbl[x]$ to ℓ.

We keep a buffer with labels and ids of $\log^{2\varepsilon} n$ segments in every node of the weighted tree. All updates are implemented using buffers. When a new segment s is inserted into a tree, we find its predecessor in the list of labeled segments in a chunk using a USF data structure. Then we assign a label $lab(s)$ and an id $id(s)$ to the new segment s; $lab(s)$ and $id(s)$ are inserted into the buffer of the root node. When a segment is deleted, we also assign it a label and an id (unless this segment already was assigned a label) and insert this information into the buffer of the root node. When a buffer of some node u contains $\log^{3\varepsilon} n$ segments, we flush the buffer, i.e., we move the segments from the buffer of u into buffers associated with the children of u. Since we store only the segment labels and ids in a buffer node, the buffer fits into one word. Hence we can flush the buffer in $O(\log^\varepsilon n)$ time using a look-up table. The total amortized cost of an update is $O(\frac{\log^\varepsilon n \cdot \log^{2\varepsilon} n}{\log^{3\varepsilon} n}) = O(1)$. The update cost can be de-amortized.

We also need to update some labels when new segments are inserted. The standard method of label maintenance [4, 17] requires that we change $O(\log^{4\varepsilon} n)$ labels after each insertion. We can update the labels in $O(1)$ time by storing multiple labels in the same word. Labels of all segments are stored in a list, in increasing order. Every element of a list stores the segment label and the segment id of some segment. We divide the list of labels into blocks, such that every block contains at most $r = \log^{4\varepsilon} n$ elements and at least $r/4$ elements. We employ a two-part labeling scheme: blocks are assigned monotonously increasing labels using the method from [4, 5, 17]. Each segment s is assigned a label $lab(s) = x \cdot r + y$, where x is the label of the block B that contains s and y is the number of segments $s' < s$ in the block B.

Each block fits into one word. When a new segment is inserted, we find the block B where it must be inserted. The y-components of up to r labels in B must be changed. We update these labels in $O(1)$ time after inserting a new block. When the number of segments in a block reaches r, we split the block into two equal-size blocks. In this case we change the block labels of $\log^{4\varepsilon} n$ different blocks and update all segment labels in these blocks. We can update the segment labels in $O(\log^{4\varepsilon} n)$ time, i.e., in $O(1)$ time per affected block.

One other issue related to dynamic labeling must be addressed: when a segment label is changed, it is possible that the "old" label of the same segment is stored somewhere in the tree. We update segment labels in the tree using the same buffering strategy as used for segment insertions/deletions. When the labels of segments in a block are changed, we insert the updated labels into the update buffer of the root node. For every segment we store its id, its new label, and its old label. Buffers with updated labels are flushed in the same way as buffers with inserted/deleted segments. When a buffer is flushed and some labels are moved from a node u to its child u_i, then the labels of segments in (the buffer of) u_i are updated accordingly.

In order to answer a query, we traverse a path in the tree and examine the contents of the buffers in each node on the path. Since the contents of all buffers on the path fit into one word, we can answer a query or identify the group leaf that contains the answer in $O(\ell \cdot \log\log n)$ time, where ℓ is the path length.

D.2 Slow Data Structure

Let $L(u, j)$ denote the list of all segments in $L(u)$ with span index j. Let $L'(u, j)$ denote sub-list that contains one segment from every block of $\log n$ segments in $L(u, j)$. Finally let $L'(u) = \cup_j L(u, j)$. We

maintain a standard fractional cascading data structure on $L'(u)$. Additionally we keep a USF data structure $V(u, j)$ for every j: for every segment $s \in L'(u)$ we can find the largest $s_j \in L(u, j) \cap L'(u)$ such that $s' < s$. $V(u, j)$ supports queries and updates in $O(\log \log n)$ time. We also maintain a data structure that supports finger searches on each $L(u, j)$.

A query is answered in two stages.

Stage 1. First, we identify the segment $p'(u)$ from $L'(u)$ directly below the query point q in every node u that contains q. Stage 1 can be executed in $O(\log n)$ time.

Stage 2. For each j, we find the largest $p_j(u) \in L(u, j) \cap L'(u)$ such that $p_j(u) \leq p(u)$. Next we search among $\log^2 n$ segments that follow $p_j(u)$ in $L(u, j)$ and and find $s(u, j) = \mathtt{ray}(q, L(u, j))$. The answer to $\mathtt{ray}(q, L(u))$ is the largest segment among all $s(u, j)$. Stage 2 takes $O(\log^{\varepsilon} n \log \log n)$ time per node.

Updates are supported as follows: when a new segment is inserted into $L(u, j)$ or a segment is deleted from $L(u, j)$, we update the USF data structure $V(u, j)$ and the data structure for finger search queries on $L(u, j)$. This can be done in $O(\log \log n)$ time. We maintain the block sizes of each $L(u, j)$ using standard techniques. When two blocks are merged or a block is split into two parts, we update $L'(u)$ and data structures $V(u, j)$ for all j. The total cost of splitting or merging blocks is $O(\log^{\varepsilon} n \log \log n)$. It can be distributed among $O(\log n)$ updates of $L(u, j)$.

D.3 Queries on $L(u)$

Queries on left segments are answered using the combination of two structures, described in Sections D.1 and D.2. We start by executing Stage 1 of the slow method from Section D.2. Then we search in all $L(u)$ using the truncated weighted trees, as described in Section D.1. Using the same analysis as in Section 6.3, this stage takes $O(\log n)$ time. If the search in a node u ends in a group leaf, we will say that a node u is difficult. If a node u is difficult, we execute Stage 2 from Section D.2 and find $s(u) = \mathtt{ray}(q, L(u))$. There are at most $O(\log^{1-2\varepsilon} n)$ difficult nodes and we spend $O(\log^{\varepsilon} n \log \log n)$ in each difficult node. Hence the total time spent in all difficult nodes is $O(\log n)$. The total time needed to find $\mathtt{ray}(q, L(u))$ in all visited nodes u is $O(\log n)$.

REFERENCES

[1] Lars Arge, Gerth Stølting Brodal, and Loukas Georgiadis. 2006. Improved dynamic planar point location. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*. 305–314. https://doi.org/10.1109/FOCS.2006.40

[2] Lars Arge and Jeffrey Scott Vitter. 2003. Optimal External Memory Interval Management. *SIAM J. Comput.* 32, 6 (2003), 1488–1508. https://doi.org/10.1137/S009753970240481X

[3] Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. 1994. Dynamic point location in general subdivisions. *J. Algorithms* 17, 3 (1994), 342–380. https://doi.org/10.1006/jagm.1994.1040

[4] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two Simplified Algorithms for Maintaining Order in a List. In *10th Annual European Symposium on Algorithms (ESA)*. 152–164. https://doi.org/10.1007/3-540-45749-6_17

[5] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. 2017. File Maintenance: When in Doubt, Change the Layout!. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1503–1522. https://doi.org/10.1137/1.9781611974782.98

[6] Jon Louis Bentley. 1977. Algorithms for Klee's rectangle problems. (1977). Unpublished manuscript, Department of Computer Science, Carnegie-Mellon University.

[7] Timothy M. Chan and Yakov Nekrich. 2015. Towards an Optimal Method for Dynamic Planar Point Location. In *Proc. 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 390–409. https://doi.org/10.1109/FOCS.2015.31

[8] Timothy M. Chan and Yakov Nekrich. 2018. Towards an Optimal Method for Dynamic Planar Point Location. *SIAM J. Comput.* 47, 6 (2018), 2337–2361. https://doi.org/10.1137/16M1066506

[9] Timothy M. Chan and Konstantinos Tsakalidis. 2018. Dynamic Planar Orthogonal Point Location in Sublogarithmic Time. In *34th International Symposium on Computational Geometry (SoCG 2018) (LIPIcs, Vol. 99)*, Bettina Speckmann and Csaba D. Tóth (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 25:1–25:15. https://doi.org/10.4230/LIPIcs.SoCG.2018.25

[10] Bernard Chazelle. 1991. Computational Geometry for the Gourmet: Old Fare and New Dishes. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*. 686–696. https://doi.org/10.1007/3-540-54233-7_174

[11] Bernard Chazelle. 1994. Computational geometry: A retrospective. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*. 75–94. https://doi.org/10.1145/195058.195110

[12] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 2 (1986), 133–162. https://doi.org/10.1007/BF01840440

[13] Siu-Wing Cheng and Ravi Janardan. 1992. New results on dynamic planar point location. *SIAM J. Comput.* 21, 5 (1992), 972–999. https://doi.org/10.1137/0221057

[14] Yi-Jen Chiang, Franco P. Preparata, and Roberto Tamassia. 1996. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. *SIAM J. Comput.* 25, 1 (1996), 207–233. https://doi.org/10.1137/S0097539792224516

[15] Yi-Jen Chiang and Roberto Tamassia. 1992. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Int. J. Comput. Geometry Appl.* 2, 3 (1992), 311–333. https://doi.org/10.1142/S0218195992000184

[16] Yi-Jen Chiang and Roberto Tamassia. 1992. Dynamic algorithms in computational geometry. *Proc. IEEE* 80, 9 (1992), 1412–1434.

[17] Paul F. Dietz and Daniel Dominic Sleator. 1987. Two Algorithms for Maintaining Order in a List. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. 365–372.

[18] Otfried Fries. 1990. *Suchen in dynamischen planaren Unterteilungen.* Ph.D. Thesis. Universität des Saarlandes.

[19] Yoav Giora and Haim Kaplan. 2009. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms* 5, 3 (2009), 28. https://doi.org/10.1145/1541885.1541889

[20] Mordecai J. Golin, John Iacono, Stefan Langerman, J. Ian Munro, and Yakov Nekrich. 2018. Dynamic Trees with Almost-Optimal Access Cost. In *26th Annual European Symposium on Algorithms (ESA) (LIPIcs, Vol. 112)*. 38:1–38:14. https://doi.org/10.4230/LIPIcs.ESA.2018.38

[21] Michael T. Goodrich and Roberto Tamassia. 1998. Dynamic trees and dynamic point location. *SIAM J. Comput.* 28, 2 (1998), 612–636. https://doi.org/10.1137/S0097539793254376

[22] Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. 1977. A new representation for linear lists. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*. 49–60. https://doi.org/10.1145/800105.803395

[23] Kurt Mehlhorn and Stefan Näher. 1990. Dynamic fractional cascading. *Algorithmica* 5, 2 (1990), 215–241. https://doi.org/10.1007/BF01840386

[24] Christian Worm Mortensen. 2006. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.* 35, 6 (2006), 1494–1525. https://doi.org/10.1137/S0097539703436722

[25] J. Ian Munro and Yakov Nekrich. 2019. Dynamic Planar Point Location in External Memory. In *35th International Symposium on Computational Geometry (SoCG)*. 52:1–52:15. https://doi.org/10.4230/LIPIcs.SoCG.2019.52

[26] Franco P. Preparata and Roberto Tamassia. 1989. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.* 18, 4 (1989), 811–830. https://doi.org/10.1137/0218056

[27] Neil Sarnak and Robert Endre Tarjan. 1986. Planar Point Location Using Persistent Search Trees. *Communincations of the ACM* 29, 7 (1986), 669–679. https://doi.org/10.1145/6138.6151

[28] Jack Snoeyink. 2004. Point location. In *Handbook of Discrete and Computational Geometry* (2nd ed.), Jacob E. Goodman and Joseph O'Rourke (Eds.). CRC Press LLC, Boca Raton, FL, Chapter 34, 767–787.