Towards an Optimal Method for Dynamic Planar Point Location

Timothy M. Chan* Yakov Nekrich†

August 12, 2015

Abstract

We describe a fully dynamic linear-space data structure for point location in connected planar subdivisions, or more generally vertical ray shooting among non-intersecting line segments, that supports queries in $O(\log n(\log\log n)^2)$ time and updates in $O(\log n \log\log n)$ time. This is the first data structure that achieves close to logarithmic query and update time simultaneously, ignoring $\log\log n$ factors. We further show how to reduce the query time to $O(\log n \log\log n)$ in the RAM model with randomization. Alternatively, the query time can be lowered to $O(\log n)$ if the update time is increased to $O(\log^{1+\varepsilon} n)$ for any constant $\varepsilon > 0$, or vice versa.

1 Introduction

In the *dynamic planar point location* problem, we want to maintain a connected planar polygonal subdivision with n edges, subject to insertions and deletions of edges, so that we can quickly return a label of the face containing a query point q.

More generally, we want to maintain a set of n non-intersecting line segments (not necessarily connected), subject to insertions and deletions of line segments, so that we can quickly find the segment immediately above a query point q, i.e., answer a *vertical ray shooting* query. Indeed, if the subdivision is connected, knowing the edge e immediately above q implies knowing the face containing q, by maintaining the list of edges surrounding each face in a concatenable queue structure (e.g., see [12]).

Dynamic planar point location is a natural generalization of one-dimensional dynamic predecessor search. It is among the most fundamental problems in geometric data structures, and also gives rise to one of the major remaining open questions in computational geometry, reiterated in several invited talks [9, 10] and surveys [14, 32]: is there a fully dynamic $O(n)$-space data structure for planar point location with $O(\log n)$ query and update time?

A variety of different techniques have been proposed, based on segment trees [4], interval trees and priority search trees [12], dynamic fractional cascading [1, 3], the trapezoid method [15], and primal/dual spanning trees [20]. As Chazelle put it in his STOC'94 invited talk abstract [10], "many solutions exist, but all suffer from extra logarithmic factors in one or several of the relevant resource measures (query, update time, and storage)." See the top part of Table 1 for a list of previous results. In every row, at least one of the time bounds is worse than $\log^2 n$. The sole exception is the most recent result by Arge et al. [1] from FOCS'06, whose update time beats $\log^2 n$ by merely a $\log\log n$ factor.

*Cheriton School of Computer Science, University of Waterloo (tmchan@uwaterloo.ca). Work supported by an NSERC grant.
†Cheriton School of Computer Science, University of Waterloo (yakov.nekrich@googlemail.com).

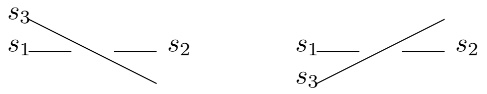

Figure 1: Not knowing the segment s_3 yet to be inserted, we cannot order s_1 and s_2 by y.

Omitted from the table are many more results [6, 19, 22, 27] for the special case of *orthogonal* subdivisions, i.e., vertical ray shooting among horizontal line segments. There, data structures with $O(\log n)$ query and update time are known (in the RAM model) [6, 19]. However, the orthogonal special case appears easier for multiple reasons; for example, in the general case, it is not possible to globally y-order all segments in advance (see Figure 1), and previously deleted segments may intersect current segments.

We obtain the new results shown in the bottom part of Table 1. In particular, we obtain the first result that truly breaks the $\log^2 n$ barrier and gets within $\log\log n$ factors of the desired $O(\log n)$ query and update time bound simultaneously.

Our basic data structure requires $O(\log n(\log\log n)^2)$ query time and $O(\log n\log\log n)$ update time. It can be described in two different ways—non-recursively in Section 2, or recursively in Section 4—the reader may choose to read either section, depending on his/her taste. Both versions are conceptually simple, and both sections require just a few pages. One $\log\log n$ factor in the query time unsurprisingly comes from dynamic fractional cascading (which uses van Emde Boas trees), but the other $\log\log n$ factor is where the novelty lies.

In the non-recursive version, we adopt a *multi-colored* variant of the standard segment tree, where different pieces of one segment may be assigned different colors and we only y-order pieces of the same color—this gets around the issue indicated by Figure 1. The $\log\log n$ factor arises from the number of colors, which are assigned using some form of binary searching over the $O(\log n)$ levels of the segment tree (see the definition of *canonical paths/subtrees* in Section 2.4). In the recursive version, we use instead a $\sqrt{n}$-way divide-and-conquer. The $\log\log n$ factor arises from the depth of the recursion. Readers familiar with the literature may compare and contrast our solution with other existing data structuring techniques which generate $\log\log n$ complexity, such as van Emde Boas trees [33] (which similarly have both a non-recursive and a recursive description) and "ball inheritance" [8].

The non-recursive version is further refined in Section 3, where we address remaining technical issues, such as reduction of the space bound to $O(n)$, query/update-time trade-offs (lowering the query time to $O(\log n)$ at the expense of increasing the update time to $O(\log^{1+\varepsilon} n)$, or vice versa), implementation in the pointer machine model, derandomization, and deamortization. The recursive version is further refined in Section 5, where we manage to shave off a $\log\log n$ factor from the $O(\log n(\log\log n)^2)$ query bound using RAM tricks, combined with an interesting randomized search technique [7]. (The non-recursive version does not naturally yield this improvement; on the other hand, the recursive version does not naturally yield some of the stated query/update trade-offs.)

2 Non-Recursive Approach

For simplicity, we assume that the endpoints of all segments have x-coordinates coming from a static set X of size $O(n)$; we will describe how to remove this assumption later in Section 3.1. We start by reviewing known relevant techniques in the next three subsections, before revealing our new method in Section 2.4.

Reference	Space	Query Time	Insertion Time	Deletion Time	
Bentley [4]	$n \log n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	G
Fries [18]	n	$\log^2 n$	$\log^4 n$	$\log^4 n$	C
Preparata–Tamassia [30]	n	$\log^2 n$	$\log^2 n$	$\log^2 n$	M
Chiang et al. [13]	$n \log n$	$\log n$	$\log^3 n$	$\log^3 n$	C
Chiang–Tamassia [15]	$n \log n$	$\log n$	$\log^2 n$	$\log^2 n$	M
Cheng–Janardan [12]	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 [20]	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$^{\dagger\ddagger *}$
Theorems 3.3 and 3.7	n	$\log n(\log\log n)^2$	$\log n \log\log n$	$\log n \log\log n$	G
Theorems 3.4 and 3.7	n	$\log n$	$\log^{1+\varepsilon} n$	$\log^{1+\varepsilon} n$	G
Theorem 3.8	n	$\log n$	$\log^{1+\varepsilon} n$	$\log n(\log\log n)^{1+\varepsilon}$	G*
Theorems 3.5 and 3.7	n	$\log^{1+\varepsilon} n$	$\log n$	$\log n$	G
Theorem 5.3	n	$\log n \log\log n$	$\log n \log\log n$	$\log n \log\log n$	G$^{\ddagger *}$

Table 1: Previous and new results on dynamic planar point location. Here, $\varepsilon > 0$ is an arbitrarily small constant. Entries marked M are for monotone subdivisions, C for connected subdivisions, and G most generally for vertical ray shooting among non-intersecting segments. Entries marked † and ‡ require amortization and (Las Vegas) randomization respectively. All results are in the pointer machine model, except the ones marked *, which are in the RAM model (assuming unit cost for standard operations on $(\log n)$-bit words, comparisons of input x-coordinates, and point–segment comparisons—the input may be real-valued).

2.1 Segment Tree

We first review the standard segment tree [4, 14, 16, 32].

We build a binary tree T of height $H = O(\log n)$, where each node corresponds to a vertical slab. The root slab is $(\min(X), \max(X)] \times \mathbb{R}$; each internal node's slab is the disjoint union of the slabs of the children; and each leaf slab has x-coordinates delimited by two consecutive elements of X.

Let S be the set of $O(n)$ input non-intersecting segments. A segment *spans* a slab if it intersects the slab but both endpoints are outside the slab. For each slab u in T, we define the list

$$L(u) = \{\text{all segments in } S \text{ that span } u \text{ but not } \mathit{par}(u)\},$$

where $\mathit{par}(u)$ denotes the parent of u.

The *segment tree* of S refers to the tree T together with the lists $L(u)$ at each node u, where each $L(u)$ is stored in *sorted y-order*, in a binary-searchable structure.

Queries. To answer a vertical ray shooting query for a point q using the segment tree, we can just find the $O(H)$ slabs in T containing q, find the successor of q (with respect to y-order) among the segments in $L(u)$ by binary search in $O(\log n)$ time for each such slab u, and return the lowest of the segments found. The overall query time is $O(H \log n) = O(\log^2 n)$.

Updates. To insert/delete a segment s, observe that s is in $O(H)$ lists of the segment tree. Namely, we find the two leaf slabs v_L and v_R containing the left and right endpoints of s (pretending that s has been

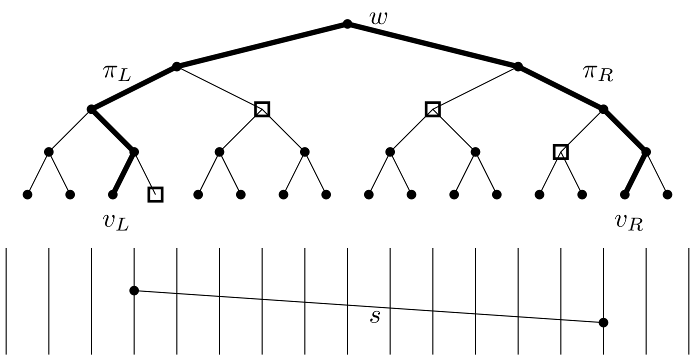

Figure 2: Two paths π_L and π_R in a segment tree. The segment s is in the lists $L(u)$ for the nodes u shown as white boxes.

extended slightly on both sides), take the lowest common ancestor w of v_L and v_R, and consider the path π_L from the left child of w to v_L and the path π_R from the right child of w to v_R; then s is in the lists $L(u)$ for right siblings u of the nodes along π_L (excluding its top node), and the left siblings u of the nodes along π_R (excluding its top node). See Figure 2. We can insert/delete s in $L(u)$ in $O(\log n)$ time for each affected node u. The overall update time is $O(H \log n) = O(\log^2 n)$.

2.2 Segment Tree with Dynamic Fractional Cascading

The query time can be improved to near-logarithmic by a dynamic form [22] of *fractional cascading* [11], which has been used as an ingredient in several previous methods for dynamic point location [1, 3]. We describe our variant of the technique, which uses randomization to simplify matters;[1] a derandomized variant will be described in Section 3.6.

We maintain an *augmented list* $L^+(u)$ for each node u, defined as follows. At the root u, we set $L^+(u) = L(u)$. Given $L^+(par(u))$, we first maintain lists $sample(L(u))$ and $sample(L^+(par(u)))$, where $sample(A)$ denotes a sublist of a list A, sorted in y-order, where each element of A is selected to be in the sublist independently with probability $\alpha = 1/\log^4 n$. Note that every two consecutive elements of $sample(A)$ are separated by $O((1/\alpha)\log n)$ elements of A w.h.p.[2] We set

$$L^+(u) = sample(L(u)) \cup sample(L^+(par(u))).$$

(In other words, we pass a fraction α of the elements of the parent list to each child's list.) Note that segments spanning $par(u)$ span u, and so inductively all segments in $L^+(u)$ span u and can indeed be y-ordered. (In contrast, if we were to pass elements instead from children lists to parent lists, the resulting lists would contain segments that are not y-comparable.) We assume that each list is stored in a structure supporting *finger search* [21].

We need a data structure to maintain a list A and a sublist $B \subseteq A$ that can find the predecessor/successor of any element in A among the elements in B. In our application, $A = L^+(u)$ and B is $sample(L(u))$ or $sample(L^+(par(u)))$. Mehlhorn and Näher [22] presented a *union-split-find* data structure (based on van Emde Boas trees) that can answer such a query in $O(\log\log n)$ time and support an update in $O(\log\log n)$

[1]We make the standard assumption that the update sequence is independent of the random choices made by the update algorithm. We do not assume that the update sequence is random, for otherwise the problem is easier [25].

[2]With high probability, i.e., with probability at least $1 - 1/n^d$ for an arbitrarily large constant d.

time given the location in A of the element to be inserted/deleted (which may require $O(\log n)$ time to find by an initial binary search).

Queries. To answer a vertical ray shooting query for a point q, we first find the successor of q in $L^+(u)$ for the leaf slab u containing q by binary search in $O(\log n)$ time. From this, we can deduce the successor of q in $\mathit{sample}(L(u))$ and the successor of q in $\mathit{sample}(L^+(\mathit{par}(u))$ in $O(\log\log n)$ time by querying a union-split-find structure. From these, we can deduce the successor of q in $L(u)$ and the successor of q in $L^+(\mathit{par}(u))$ by finger search in $O(\log((1/\alpha)\log n)) = O(\log\log n)$ time w.h.p. We then reset u to $\mathit{par}(u)$ and repeat. At the end, we have obtained the successor of q in $L(u)$ for all $O(H)$ slabs u containing q, and can return the lowest segment found. The overall query time is $O(\log n + H\log\log n) = O(\log n \log\log n)$.

Updates. Recall that the insertion/deletion of a segment s requires updates to the list $L(u)$ for $O(H)$ nodes u, costing $O(H\log n) = O(\log^2 n)$ time (an insertion to each $L(u)$ may require a binary search for s).

In addition, with probability α, an update to $L(u)$ may cause an update to the augmented list $L^+(u)$, which in turn may trigger updates to the augmented lists of the children of u, and so on. Let t_i denotes the expected number of updates to the augmented lists triggered by an update to $L^+(u)$ for a node u at level i. Then

$$t_i \leq 1 + \alpha \cdot 2t_{i+1}, \tag{1}$$

which yields $t_i = O(1)$. The $O(H)$ updates to the lists $L(u)$ thus trigger an expected $O(\alpha H)$ number of updates to all the augmented lists, costing $O(\alpha H \log n) = o(1)$ expected time. The overall expected update time therefore remains $O(\log^2 n)$.

The $\log^2 n$ barrier. Fractional cascading improves the query time but not the update time. The bottleneck in the insertion algorithm lies not in updating the augmented lists but in updating/searching the original lists $L(u)$ of the affected nodes u. Unfortunately, these searches for s in $L(u)$ cannot be sped up by union-split-find data structures, because the affected nodes u do not lie on two paths but are *siblings* of nodes along two paths. (If we were to pass elements from a node's list to a sibling's child's list, the resulting lists may contain segments that are not directly y-comparable.) Although it is possible to reduce the update time slightly by a $\log\log n$ factor at the expense of increasing the query time by increasing the fan-out of the tree, it is unclear how to get a more substantial improvement while staying within the segment tree framework.

2.3 1-Sided Case

There is one special case for which the $\log^2 n$ barrier can be easily overcome, and which will be used as a subroutine for our method later. The special case is when the line segments are *1-sided*, i.e., they all touch one of the walls, say, the right wall, of the root slab. Here, all the segments can obviously be y-ordered. (A solution to this special case is sometimes used as secondary structures in an *interval tree* [14, 16, 32], but we would like to stay within the segment tree framework.)

Specifically, for each slab u in T, define the list

$$L'(u) = \{\text{all segments in } S \text{ whose left endpoints are inside } u\}.$$

The resulting tree of lists, each stored in sorted y-order, is equivalent to a *2D range tree* [16] (if we transform each segment to a point (x, y) where x is the x-coordinate of the left endpoint and y is the y-coordinate at the right wall of the root slab). It is already known how to maintain a 2D range tree efficiently by dynamic fractional cascading, using union-split-find structures [22]. The situation is simpler than in Section 2.2 because the lists are already hierarchical, i.e., $L'(u) \subseteq L'(par(u))$. There is no need to augment these lists and pass a fraction of elements from one list to another, and it is simpler to proceed downward rather than upward during searches:

To insert/delete a segment s, we first update $L'(u)$ at the root u by binary search in $O(\log n)$ time. From the location of s in $L'(u)$, we can deduce the location of s in $L'(w)$ for the child w of u containing the left endpoint of s, and update $L'(w)$, in $O(\log\log n)$ time by operations on a union-split-find structure. We then reset u to w and repeat.

Observe that for each node u with left sibling v, the list $L(u)$ is exactly equal to $L'(v)$ in the 1-sided case; and for each node u which is itself a left child, the list $L(u)$ is equal to $\emptyset$. We conclude that in an insertion/deletion, we can update all the affected lists $L(u)$ in the segment tree for this special case in $O(\log n + H \log\log n) = O(\log n \log\log n)$ time.

2.4 New Method: A Multi-Colored Segment Tree

To overcome the $\log^2 n$ barrier in the general case, we introduce a relaxed, *multi-colored*[3] version of the segment tree. For each element s that appears in the list $L(u)$ of a node u, we assign s a color in u (note that the same segment s may be assigned different colors in different nodes). In other words, we partition $L(u)$ into multiple sublists

$$L_\chi(u) = \{s \in L(u) : s \text{ has color } \chi \text{ in } u\}.$$

In a multi-colored segment tree, instead of requiring $L(u)$ to be sorted in y-order, we only require $L_\chi(u)$ to be sorted in y-order for each color χ separately. (This provides more flexibility, since elements in $L(u)$ that are assigned different colors need not be y-compared.)

To speed up querying by fractional cascading, we maintain augmented lists $L^+_\chi(u)$ as before but for elements of each color χ separately; in other words, we define

$$L^+_\chi(u) = \mathit{sample}(L_\chi(u)) \cup \mathit{sample}(L^+_\chi(\mathit{par}(u))).$$

Queries. The query algorithm is now slowed down by a factor of K, the number of colors: we find the successor of the query point q in $L_\chi(u)$ for all slabs u containing q as before, using the augmented lists, in $O(\log n \log\log n)$ time w.h.p. for each color χ, and return the lowest segment found. The overall query time is $O(K \log n \log\log n)$ w.h.p.

The coloring scheme. We thus want to keep the number of colors K small to ensure good query time. The crux to our solution is a clever way to assign colors that achieves $K = O(\log\log n)$ and at the same time makes updates easier to do. To this end, we first make a few definitions. A *canonical path* is a path in T from a node at level $j2^i$ to a node at level $(j+1)2^i$ (or a leaf before that level) for some integers i and j with $i < \log H$. A *canonical subtree* is the subtree in T rooted at a node at level $j2^i$ and reaching all its descendants at level $(j+1)2^i$ (or leaves before that level) for some integers i and j. See Figure 3 (left). Note that each canonical path of length 2^i lies in a canonical subtree of height 2^i, and canonical subtrees of the same height are edge-disjoint. Furthermore, any path in T to a leaf can be decomposed into $O(\log H)$ canonical subpaths (with at most one subpath of each length 2^i).

[3]This is unrelated to existing literature on *colored range searching* which deals with colored input sets; in contrast, our challenge is in finding a good coloring.

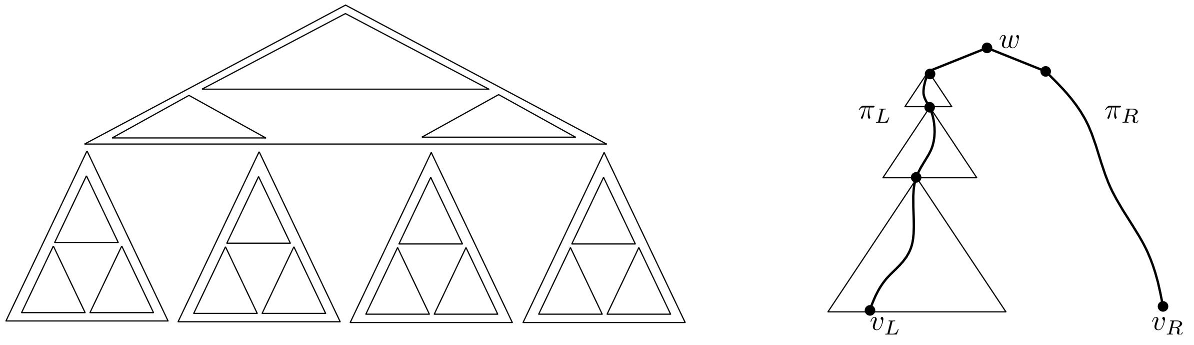

Figure 3: (Left) The canonical subtrees. (Right) Decomposing a path π_L into canonical subpaths.

For a given segment s, we find the two leaf slabs v_L and v_R containing the left and right endpoints of s (pretending that s has been extended slightly on both sides), take the lowest common ancestor w of v_L and v_R, and consider the path π_L from the left child of w to v_L and the path π_R from the right child of w to v_R. We decompose π_L into $O(\log H)$ canonical subpaths (see Figure 3(right)). For each such canonical subpath, say, of length 2^i, the segment s is in $L(u)$ for the right siblings u of the nodes along the subpath (excluding its top node); we assign s the color $(i, \text{“}L\text{”})$ in each such u. Similarly, we decompose π_R into $O(\log H)$ canonical subpaths. For each such canonical subpath, say, of height 2^i, the segment s is in $L(u)$ for the left siblings u of the nodes along the subpath (excluding its top node); we assign s the color $(i, \text{“}R\text{”})$ in each such u. As $i < \log H$, the number of possible colors is indeed $K = O(\log H) = O(\log\log n)$. The resulting query time is thus $O(\log n(\log\log n)^2)$ w.h.p.

Updates. Observe that anywhere inside a canonical subtree τ of height 2^i, the only segments that can be assigned the color $(i, \text{“}L\text{”})$ must have right endpoints outside the root slab of τ. In other words, when restricted to the lists $L_\chi(u)$ for the color $\chi = (i, \text{“}L\text{”})$, the tree τ is identical to a segment tree for 1-sided segments, as discussed in Section 2.3. A similar statement holds for the color $(i, \text{“}R\text{”})$.

Thus, when we insert/delete a segment, we can update the affected lists $L_\chi(u)$ at all siblings u of the nodes along a canonical subpath of length 2^i by the method in Section 2.3 in $O(\log n + 2^i \log\log n)$ time (since the height of the corresponding canonical subtree is 2^i). Doing this over all $O(\log H)$ canonical subpaths of π_L and π_R and summing over the $O(\log H)$ different possible i's, we obtain an overall update time of $O(\log H \log n + H \log\log n) = O(\log n \log\log n)$.

Finally, remember that the $O(H)$ updates to $L_\chi(u)$ may trigger updates to the augmented lists, but their expected cost is $o(1)$ as explained in Section 2.2.

Theorem 2.1 *There is a data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log n(\log\log n)^2)$ expected time and support insertions and deletions of segments in $O(\log n \log\log n)$ expected time, assuming that all x-coordinates are from a static set X of size $O(n)$.*

3 Refinements

3.1 Dynamizing X

To support insertions of new x-values to the set X, we can replace the static binary tree T with a *weight-balanced B-tree* [2] with constant-bounded degree. The amortized cost per insertion to X is known to be

$O(\log n)$, if the cost of splitting a node u takes time linear in the size n_v of the subtree at u (e.g., see [19, Theorem 2.1]). When a node u is split, we can recompute the lists $L_\chi(v)$ for the $O(1)$ affected nodes v from scratch, which can indeed be done in $O(n_u)$ time. In addition, we need to recompute the augmented lists $L_\chi^+(v)$ for all the descendants v of u from scratch; since the amortized cost for updating the augmented lists for the reinsertion of each segment is $O((\alpha H)\log n) = o(1)$, this step takes $o(n_u)$ time.

When n is increased or decreased by a factor of 2, we can rebuild the entire data structure from scratch.

3.2 Reducing Space

The space usage of our data structure is dominated by the total size of the lists $L(u)$ of the segment tree, which is $O(n\log n)$, since each segment is in $L(u)$ for logarithmically many nodes u, and the augmented lists have expected total size $o(n)$. In this subsection, we show how to reduce space to $O(n)$. In fact, we describe a general way to reduce the space requirement of any dynamic point location structure. The following space-reduction lemma is inspired by a recent work [26]. In a *decomposable search* problem [5], the answer to a query for a union of two subsets can be obtained from the answers to the queries for the subsets in constant time; for example, vertical ray shooting is a decomposable search problem.

Lemma 3.1 *Consider a decomposable search problem, where (i) there is an $S(n)$-space fully dynamic data structure with $Q(n)$ query time and $U(n)$ update time, and (ii) there is an $S_D(n)$-space deletion-only data structure with $Q_D(n)$ query time, $U_D(n)$ update time, and $P_D(n)$ preprocessing time. Then there is an $O(S(n/z) + S_D(n))$-space fully dynamic data structure with $O(Q(n/z) + Q_D(n)\log z)$ query time and $O(U(n/z) + U_D(n) + (P_D(n)/n)\log z)$ amortized update time for any given parameter z (assuming that $P_D(n)/n$ is nondecreasing).*

Proof: This follows from a variant of Bentley and Saxe's *logarithmic method* [5]. In the original version, a dynamic set is decomposed into $O(\log n)$ deletion-only subsets. Sizes of subsets increase geometrically, so that the smallest subset is of size $O(1)$ and every subset is $\Theta(f)$ times larger than the preceding subset, where f is a constant. In our construction, the smallest subset $\mathcal{C}_0$ is of size $\Theta(n/z)$ and every following subset $\mathcal{C}_i$ is $\Theta(f)$ times larger than $\mathcal{C}_{i-1}$. New points are always inserted into the subset $\mathcal{C}_0$. When a point is deleted, we remove it from its subset. We can maintain sizes of all subsets $\mathcal{C}_i$ using the method described in [5]; this incurs an amortized cost $O((P_D(n)/n)\log z)$ per update operation. Thus we decompose a dynamic set into $O(\log z)$ deletion-only data structures and a fully-dynamic data structure of size $O(n/z)$. □

Lemma 3.2 *If there is a deletion-only data structure for vertical ray shooting queries for n* horizontal *segments with $S_{\text{orth}}(n)$ space, $Q_{\text{orth}}(n)$ query time, $U_{\text{orth}}(n)$ update time, and $P_{\text{orth}}(n)$ preprocessing time, then there is a deletion-only data structure for vertical ray shooting queries for n arbitrary non-intersecting segments with $S_D(n) = S_{\text{orth}}(n) + O(n)$ space, $Q_D(n) = Q_{\text{orth}}(n) + O(\log n)$ query time, $U_D(n) = U_{\text{orth}}(n) + O(1)$ update time, and $P_D(n) = P_{\text{orth}}(n) + O(n\log n)$ preprocessing time.*

Proof: We preprocess the initial input in a static data structure, with $O(n\log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time [32]. We also compute a topological order of the segments, with the property that if segment s is below segment s' at some vertical line, s appears before s'. Such an ordering can be computed by a plane sweep in $O(n\log n)$ time [29]. We then map each segment s into a horizontal segment with y-coordinate equal to the rank of s in the topological order (while preserving the x-coordinates). We maintain these horizontal segments for vertical ray shooting in a given deletion-only data structure. To answer a query, we first answer the query in the initial static data structure; let p be the point on the returned segment s immediately above the query point (note that s could have been deleted). We then map p to rank space, i.e., reset its y-coordinate to the rank of s. Finally we answer the query for the mapped point p in the data structure for the horizontal segments and map the answer back. $\square$

Known results on dynamic vertical ray shooting among horizontal segments or dynamic orthogonal point location [6, 19] achieve $S_{\text{orth}}(n) = O(n)$, $Q_{\text{orth}}(n) = O(\log n)$, $U_{\text{orth}}(n) = O(\log n)$, and $P_{\text{orth}}(n) = O(n \log n)$. Theorem 2.1 achieves $S(n) = O(n \log n)$, $Q(n) = O(\log n (\log\log n)^2)$, and $U(n) = O(\log n \log\log n)$. Applying the two lemmas with $z = \log n$, we conclude:

Theorem 3.3 *There is an $O(n)$-expected-space data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log n(\log\log n)^2)$ expected time and support insertions and deletions of segments in $O(\log n \log\log n)$ expected amortized time.*

3.3 Trade-Off I

We can lower the query time at the expense of increasing the update time, by making a few changes to our method.

First, we increase the fan-out of the segment tree. Instead of a binary tree, we make T a degree-b tree with height $H = O(\log_b n)$ for a given parameter $b \leq \log n$. In Section 2.2, equation (1) now becomes $t_i \leq 1 + \alpha \cdot bt_{i+1}$, which still yields $t_i = O(1)$. The query time there reduces to $O(\log n + H \log\log n) = O(\log n \log_b \log n)$. Since each node now has $b - 1$ siblings, the update time increases by a factor of $O(b)$, however.

Second, in Section 2.3, for the special 1-sided case, we redefine

$$L'(u) = \{\text{all segments in } S \text{ whose left endpoints are inside } u \text{ or one of its left siblings}\}.$$

The resulting tree of lists corresponds to a b-ary variant of the 2D range tree. For each node u with immediate left sibling v, the list $L(u)$ is exactly $L'(v)$. In inserting/deleting a segment s, at each node u containing the left endpoint of s, there may be up to b children w for which s is in $L'(w)$. The update time there becomes $O(\log n + bH \log\log n) = O(b \log n \log\log n)$.

Third, for our multi-colored version of the segment tree in Section 2.4, we redefine a *canonical path* to be a path in T from a node at level jb^i to a node at level $(j + 1)b^i$, and a canonical subtree to a subtree in T rooted at a node at level jb^i and reaching descendants at level $(j + 1)b^i$, for some i, j with $i < \log_b H$. Canonical subtrees with the same height remain edge-disjoint, and every path to a leaf can be decomposed into $O(b \log_b H)$ canonical subpaths (with at most b subpaths of each length b^i). The number of colors is now $K = O(\log_b H) = O(\log_b \log n)$. The resulting query time becomes $O(\log n(\log_b \log n)^2)$.

In inserting/deleting a segment s to our multi-colored segment tree, the affected nodes include siblings u of the nodes along the $O(b \log_b H)$ canonical subpaths of π_L and π_R, and can be handled in $O(b \log_b H \log n + bH \log\log n) = O(b \log n \log\log n)$ time. In addition, the affected nodes include the $O(b)$ siblings of u between the top nodes of π_L and π_R, and we can assign s a new color *"M"* in each such u; these $O(b)$ updates take $O(b \log n)$ additional time. Updates to the augmented lists triggered require expected cost $O(\alpha bH \log n) = o(1)$.

To support insertions to X in Section 3.1, we can again use weight-balanced B-trees, with degree $\Theta(b)$. The cost per insertion to X is $O(b \log n)$, since the cost of splitting a node u is now $O(bn_u)$.

To reduce space to $O(n)$ in Section 3.2, we use a version of Lemma 3.1 with $O(Q(n/z)+Q_D(n)\log_b z)$ query time and $O(U(n/z)+U_D(n)+b(P_D(n)/n)\log_b z)$ update time, via a base-b variant of the logarithmic method (where we keep $O(\log_b z)$ deletion-only subsets). The query time remains $O(\log n(\log_b \log n)^2)$ and the expected update time remains $O(b \log n \log\log n)$. Setting $b = \log^{\varepsilon/2} n$ yields:

Theorem 3.4 *There is an $O(n)$-expected-space data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log n)$ expected time and support insertions and deletions of segments in $O(\log^{1+\varepsilon} n)$ expected amortized time for any constant $\varepsilon > 0$.*

3.4 Trade-Off II

Alternatively, we can lower the update time at the expense of increasing the query time, by the following changes.

First, we make T a degree-b tree with height $H = O(\log_b n)$ for a given parameter $b \le \log n$. For each node u, we decompose $L(u)$ into the following $O(b^2)$ sublists: if u is a k-th child, then for every $1 \le k_1 \le k \le k_2 \le b$ with $(k_1, k_2) \ne (1, b)$, define

$$L_{k_1,k_2}(u) = \{\text{all segments that span precisely the } k_1\text{-th to the } k_2\text{-th child of } par(u)\}.$$

Instead of requiring $L(u)$ to be sorted by y-order, we only require each $L_{k_1,k_2}(u)$ to be sorted by y-order. We maintain augmented lists for each (k_1, k_2) separately. The query time in Section 2.2 increases to $O(b^2(\log n + H \log\log n)) = O(b^2 \log n \log\log n)$. Note that $L_{k_1,k_2}(u)$ are identical lists for different siblings u from the k_1-th to the k_2-th child of $par(u)$; in updating $L_{k_1,k_2}(u)$, it suffices to update one copy of the list (although the augmented lists may be different, and a change to $L_{k_1,k_2}(u)$ may trigger changes to multiple augmented lists).

Second, in Section 2.3, we return to the original definition of $L'(u)$. If u is a k-th child, then $L_{k_1,b}(u)$ is exactly $L'(v)$ where v is the $(k_1 - 1)$-th child of $par(u)$ for each $1 < k_1 \le k$.

Third, for our multi-colored version of the segment tree in Section 2.4, we redefine a *canonical path* to be a path in T from a node at level $jb^{i+1} - j'b^i$ to a node at level jb^{i+1}, and a canonical subtree to a subtree in T rooted at a node at level $jb^{i+1} - j'b^i$ and reaching descendants at level jb^{i+1}, for some i, j, j' with $i < \log_b H$ and $1 \le j' \le b - 1$. Canonical subtrees with the same height remain edge-disjoint, and every path to a leaf can be decomposed into $O(\log_b H)$ canonical subpaths (with at most one subpath of each length $j'b^i$). For each canonical subpath of π_L, say, of length $j'b^i$, we use the color $(i, j, \text{“}L\text{”})$. We handle π_R similarly. The number of colors is now $K = O(b \log_b H) = O(b \log\log n)$. The resulting query time is $O(b^3 \log n \log\log n)$.

In inserting/deleting a segment to our multi-colored segment tree, the affected nodes include siblings u of the nodes along the $O(\log_b H)$ canonical subpaths of π_L and π_R, which can be handled in $O(\log_b H \log n + H \log\log n) = O(\log n \log_b \log n)$ time (remember that only one copy of identical lists for siblings needs to be updated). In addition, the affected nodes include the $O(b)$ siblings of u between the top nodes of π_L and π_R, and we can assign s a new color *“M”* in each such u; these $O(b)$ updates take $O(\log n)$ additional time (again, only one copy needs to be updated). Updates to the augmented lists triggered require expected cost $O(\alpha b H \log n) = o(1)$.

To support insertions to X in Section 3.1, we can again use weight-balanced B-trees, with degree $\Theta(b)$. The cost per insertion to X is $O(\log n)$, since the cost of splitting a node u is $O(n_u)$.

To reduce space to $O(n)$ in Section 3.2, we use a version of Lemma 3.1 with $O(Q(n/z)+bQ_D(n)\log_b z)$ query time and $O(U(n/z) + U_D(n) + (P_D(n)/n)\log_b z)$ update time, via another base-b variant of the logarithmic method (where we keep $O(b \log_b z)$ deletion-only subsets). The query time is $O(b^3 \log n \log\log n)$ and the amortized update time remains $O(\log n \log_b \log n)$. Setting $b = \log^{\varepsilon/4} n$ yields:

Theorem 3.5 *There is an $O(n)$-expected-space data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log^{1+\varepsilon} n)$ expected time and support insertions and deletions of segments in $O(\log n)$ expected amortized time for any constant $\varepsilon > 0$.*

3.5 Pointer Machine

Most parts of our algorithms in the preceding sections can be implemented in the pointer machine model. The only exception is the space-reduction part in Section 3.2, where we have invoked known dynamic (deletion-only) data structures for vertical ray shooting for horizontal segments [6, 19], which operates in the $(\log n)$-bit RAM model. In the pointer machine model, we can use instead the following alternative data structures, which are slightly less efficient but do not affect the final time bounds in Theorems 3.3, 3.4, and 3.5:

Lemma 3.6 *In the pointer machine model, there is a data structure for vertical ray shooting for horizontal segments with $S_{\text{orth}}(n) = O(n)$ space, $P_{\text{orth}}(n) = O(n \log n)$ preprocessing time, and*

(i) $Q_{\text{orth}}(n) = O(\log n \log\log n)$ *query,* $U_{\text{orth}}(n) = O(\log n \log\log n)$ *update time, or*

(ii) $Q_{\text{orth}}(n) = O(\log n)$ *query and* $U_{\text{orth}}(n) = O(\log^{1+\varepsilon} n)$ *update time, or*

(iii) $Q_{\text{orth}}(n) = O(\log^{1+\varepsilon} n)$ *query and* $U_{\text{orth}}(n) = O(\log n)$ *update time.*

Proof: (i) and (iii) were already proved by Giora and Kaplan [19] (it can be checked that their data structures can indeed be preprocessed in $O(n \log n)$ time).

(ii) was almost proved by Giora and Kaplan as well, except that they obtained $O(n \log^{\varepsilon} n)$ space. They used a space-reduction idea of Baumgarten et al. [3] based on dividing into blocks and keeping winners in each block, but we take the idea one step further and describe how to reduce space to linear. We assume familiarity with their paper [19, Section 6.1]. The base tree T is a tree with fan-out $b = \log^{\varepsilon} n$ and height $H = O(\log_b n)$. Each segment s is divided into three parts: take the lowest common ancestor w of the leaf slabs containing the two endpoints; the *left (resp. right) subsegment* is the portion of s inside the child slab of w containing the left (resp. right) endpoint of s; the *middle subsegment* refers to the remaining portion of s. Giora and Kaplan already obtained $O(n)$-space structures for vertical ray shooting among the left and right subsegments with $O(\log n)$ query time and $O(\log^{1+O(\varepsilon)} n)$ update time. We suggest a more space-efficient structure to handle the middle subsegments:

Let $M(w)$ denote the list of all the middle subsegments for the node w, in y-order. We divide $M(w)$ into $O(|M(w)|/t)$ blocks of size $O(t)$ by y-coordinate, with $t = \log^{2\varepsilon} n$. We keep a sublist $M'(w)$ constructed as follows. For each block B in $M(w)$ and each of the b child slabs w_i of w, we keep in $M'(w)$ the highest and the lowest segment from B inside the slab w_i. Then $M'(w)$ has size $O((|M(w)|/t) \cdot b) = O(|M(w)|/\log^{\varepsilon} n)$. We store only the middle subsegments of $M'(w)$ in Giora and Kaplan's data structure; this reduces their $O(n \log^{\varepsilon} n)$ space bound to $O(n)$. In addition, for each block B, we store a static data structure for vertical ray shooting with $O(t)$ space and $O(\log t)$ query time. The space for these structures is $O((|M(w)|/t) \cdot t) = O(|M(w)|)$ per node w, which sums to $O(n)$.

To answer a vertical ray shooting query among the middle subsegments for a point q, we first use Giora and Kaplan's structure to find the successor of q in $M'(u)$ for all the $O(H)$ slabs u containing q, in $O(\log n)$ overall time. For each such u, we can now identify the block B containing the true successor of q in $M(u)$ and finish the query in $O(\log t) = O(\log\log n)$ time by the static structure for B. The overall query time is $O(\log_b n \log\log n) = O(\log n)$.

For an update, we update Giora and Kaplan's structure in $O(\log^{1+O(\varepsilon)} n)$ time and rebuild the static data structure for one block in one $M'(w)$ list in additional $O(t \log t) = O(\log^{O(\varepsilon)} n)$ time. $\square$

3.6 Deamortization and Derandomization

Our only uses of amortization are from the weight-balanced B-tree in Section 3.1 and Lemma 3.2's variant of the logarithmic method in Section 3.2, but the first part can be deamortized (see [19]) and the second part can be deamortized (see [28]) by standard techniques.

Our only use of randomization comes from dynamic fractional cascading in Section 2.2. We can derandomize by incorporating Baumgarten et al.'s deterministic version of dynamic fractional cascading [3], but their scheme requires amortization. We describe a variant of deterministic dynamic fractional cascading, inspired by the method of Arge et al. [1], that guarantees good worst-case update time. We focus on the binary segment tree setting, but with straightforward modification, the technique can also be applied to the version with larger fan-out used in the trade-off results.

Every segment s is kept in $O(H)$ lists $L_\chi(u)$. To simplify the notation we will omit the subscripts and write $L(u)$ instead of $L_\chi(u)$ in the description below. Since each occurrence of s is stored in up to H lists $L^+(v)$, each segment can appear in up to H^2 structures in the worst case. Our variant keeps every element $s \in L(u)$ in a constant number of augmented lists $L^+(v)$ for descendants v of u. Hence the total number of instances of any segment s in all lists $L^+(w)$ and $L(u)$ is bounded by $O(H)$.

The second challenge is that we need $O(\log n)$ time to insert a new segment $s \in L(u)$ into an augmented list $L^+(w)$ for a descendant w of u. When a bridge segment $s \in L^+(w) \setminus L(w)$ is deleted, we also need $O(\log n)$ time because we have to insert another segment s' into $L^+(w)$. We employ delayed insertions and lazy deletions to solve this problem. Our method guarantees that there are $\Omega(\log^2 n)$ insertions into an ancestor of a node u for every insertion into $L^+(u)$. When a bridge segment s is deleted, it is marked as deleted but we keep a copy of s in the list $L^+(u)$. Segments marked as deleted will be called *phantom* segments, other segments will be called *real* segments. Let $\overline{L}(u)$ and $\overline{L}^+(u)$ denote the lists that contain all real and phantom segments; as before $L(u)$ and $L^+(u)$ are the lists of real segments. The number of phantom segments in any given list $\overline{L}^+(u)$ can be large; for instance, all segments in $\overline{L}^+(u) \setminus L(u)$ can be phantom segments. Unlike $L(u)$ segments in $\overline{L}^+(u)$ can intersect. We maintain two invariants that enable efficient searching in spite of intersections: (i) each real segment intersects at most H phantom segments and (ii) each phantom segment is intersected by at most $O(H \log^2 n)$ (real or phantom) segments. Segments in $\overline{L}^+(u)$ are ordered by y-coordinates of their endpoints, i.e., points where they intersect the right boundary (or left boundary) of u.

In Section 3.6.1 we describe how to define sets $\mathit{sample}(u)$ and how augmented lists $L^+(u) = \mathit{sample}(u) \cup \mathit{sample}(\mathit{par}(u))$ are maintained under insertions and deletions. In Section 3.6.2 we then show how invariant (ii) can be maintained and how to search in sets $\overline{L}^+(u)$ in spite of intersecting segments.

3.6.1 Structure

Each list $\overline{L}(u)$ is divided into groups of $O(\log^3 n)$ elements. Every group in $\overline{L}(u)$ except for the last one contains $\Omega(\log^3 n)$ elements. If $\overline{L}(u)$ consists of more than one group, we maintain a set $L'(u)$ that contains one representative segment from each group. $L'(u)$ is further divided into blocks that contain $\Theta(\log^2 n)$ consecutive segments from $L'(u)$. Every set $\mathit{subset}(j, L'(u))$ for $j = 0, 1, \ldots, H$ contains one element from every block (except for probably the last block in $L'(u)$). Finally we divide each $\mathit{subset}(j, L'(u))$ into chunks of size $\Theta(2^j \log n)$. We maintain sets $\mathit{subset}(t, j, L(u))$ for $t = 1, 2, \ldots, 2^j$ so that every $\mathit{subset}(t, j, L'(u))$ contains one segment for each chunk of $\mathit{subset}(j, L'(u))$ (except for, probably, the last chunk).

We say that u is a j-descendant of v if the distance from v to u equals j and v is an ancestor of u. We assign a unique index $\mathit{index}(v, u) \in [1, O(2^j)]$ to every j-descendant of v. Let $\mathit{sample}(u) =$

$\bigcup_w \mathit{subset}(j_w, \textit{\textbf{index}}(w,u), L'(w))$ where the union is over all ancestors w of u and j_w is the distance from w to u. Thus $\mathit{sample}(u)$ contains every $(2^{j_w} \cdot \mathrm{polylog}(n))$-th element from every ancestor list $L(w)$ (including also every $\mathrm{polylog}(n)$-th element from the list $L(u)$). We set $L^+(u) = \mathit{sample}(u) \cup \mathit{sample}(\mathit{par}(u))$. Sets $\mathit{sample}(u_1)$ and $\mathit{sample}(u_2)$ do not intersect for any two different nodes u_1 and u_2. There are $O(2\log^5 n)$ segments in $\mathit{subset}(t_1, j-1, L(u))$ between any two consecutive elements of $\mathit{subset}(t_2, j, L(u))$ for any t_1 and t_2. Hence there are $\mathrm{polylog}(n)$ segments from $\mathit{sample}(\mathit{par}(u))$ between any two elements of $\mathit{sample}(u)$ and there are $\mathrm{polylog}(n)$ elements of $L(\mathit{par}(u))$ between any two elements of $\mathit{sample}(\mathit{par}(u))$. By the same argument there are $\mathrm{polylog}(n)$ segments from $L^+(\mathit{par}(u))$ between any two elements of $L^+(u)$. Hence we can organize the search in lists $L^+(u)$ and $L(u)$ in the same way as in Section 2.2.

We will say that segments in $L'(u) \cap L(u)$ are *special* segments and segments in $L(u) \setminus L'(u)$ are *regular* segments. Only special segments can be potentially used as bridges. Deletions of special segments are handled in a different way: a deleted special segment s is kept in $L(u)$ until its group is rebuilt; deleted segments are called phantom segments. We maintain sizes of groups by running the following iterative process in the background. At the beginning of each iteration, we select the smallest group of size at most $2\log^3 n$ and merge it with one of its neighbor groups; the resulting group is split into two equal size groups if its size exceeds $7\log^3 n$. We also select the largest group of size at least $7\log^3 n$ and split it into two groups of equal size. When a group is re-built, we select its middle segment s_m as the special segment and insert it into $L'(u)$. The old special segment s_o is removed from $L'(u)$. If s_o is marked as deleted, we also remove s_o from $L(u)$; otherwise we keep s_o in $L(u)$ but handle it as a regular segment. If s_o was stored in some $\mathit{subset}(j, L'(u))$, we insert s_m into $\mathit{subset}(j, L'(u))$. If s_o was stored in $\mathit{subset}(t, j, L'(u))$, we insert s_m into $\mathit{subset}(t, j, L'(u))$ and into the list $\overline{L}^+(w)$ for some j-descendant w of u. An insertion into $\mathit{subset}(j, L'(u))$ or $\mathit{subset}(t, j, L'(u))$ takes $O(\log n)$ time. An insertion into $\overline{L}^+(w)$ takes $O(\log^2 n)$ time, as will be shown later in this section. We also take care that the special segment is neither the first nor the last segment in its group. Let m_j denote the number of segments deleted from the group G_j since the special segment in G_j was chosen. At the beginning of each iteration we find the group G_j with the highest value of m_j. We select the middle segment of G_j as the new special segment and remove the old special segment s_o from $L'(u)$ (if s_o was marked as deleted, we also remove it from $L(u)$).

We will show below that every iteration takes up to $O(\log^2 n)$ time. Its cost is distributed among $O(\log^2 n)$ following updates of $L(u)$. It can be shown that the size of any group in $L(u)$ is never less than $\log^3 n$ and never larger than $8\log^3 n$: let $d_i = \max(0, n_i - 7\log^3 n)$ where n_i is the number of segments in the i-th group. During each iteration we select the group G_j with the highest d_j and set $d_j = 0$. By Theorem 5 in [17], $d_i \le O(\log^2 n \cdot h_n)$ where h_n is the n-th harmonic number. We can choose constants so that $d_i \le \log^3 n/2$. Hence n_i never exceeds $7\frac{1}{2}\log^3 n$. By the same argument, $n_i \ge \frac{3}{2}\log^3 n$ and the number of segments deleted from a group G_i since its special segment was chosen is at most $\log^3 n/2$. We can maintain the sizes of blocks in $L'(u))$ in the same way.

Background processes that maintain sizes of chunks in $\mathit{subset}(j, L'(u))$ also use the same approach. For every j such that $0 \le j \le H$ and every $\mathit{subset}(j, L'(u))$, we select the largest chunk in $\mathit{subset}(j, L'(u))$ of size at least $7 \cdot 2^j \log n$ and split it into two chunks. We also select the smallest chunk of size at most $2 \cdot 2^j \log n$ and merge it with one of its neighbor chunks. When a chunk is split into two chunks, we must insert one new segment of $\mathit{subset}(j, L'(u))$ into $\mathit{subset}(t, j, L'(u))$ for $t = 1, \ldots, 2^j$; a segment inserted into $\mathit{subset}(t, j, L'(u))$ is also inserted into $\overline{L}^+(v)$ where v is a j-descendant of u with $\textit{\textbf{index}}(u, v) = t$. The cost of splitting a chunk is distributed among (at most) $2^j \log^3 n$ updates of $L(u)$ (or 2^j updates of $\mathit{subset}(j, L'(u))$). During every iteration of the background process, we spend $O(\log^2 n)$ time on splitting a chunk of $\mathit{subset}(j, L'(u))$ for $j = 0, \ldots, H$. We consider the chunk C of $\mathit{subset}(j, L'(u))$ that is currently being split and identify the next segment $s \in subset(j, L'(u))$ that must be inserted into some $subset(t, j.L'(u))$. We insert s into $subset(t, j, L'(u))$ in $O(\log n)$ time. Then, we also insert s into $\overline{L}^+(w)$ for a j-descendant w of u, such that *index*$(u, w) = t$. However, segments from different ancestors of w must be inserted into $\overline{L}^+(w)$ and insertion costs are distributed among updates; therefore we must order insertions into $\overline{L}^+(w)$ that originate in different ancestors. To this end, we use a queue Q_w to schedule insertions into $\overline{L}^+(w)$; an insertion of s via Q_w is guaranteed to be finished in $O(\log^2 n)$ time. When a segment s is inserted, we set $j = (j + 1) \bmod (H + 1)$ and spend $O(\log^2 n)$ time on $subset(j, L'(u))$ for the new value of j. The cost of merging chunks is distributed among updates of $L(u)$ in a similar way: when two chunks are merged, we remove one segment from every $subset(t, j, L'(u))$ and from a list $L^+(v)$ for every j-descendant v of u.

At any moment of time there are $O(1)$ elements of $L(u)$ that must be inserted into lists $\overline{L}^+(v)$ for some descendants v of u. Elements from different ancestors of v are inserted into $\overline{L}^+(v)$. In order to organize and order insertions from different ancestors, we associate a queue of segments Q_v with every node v. Q_v contains segments that are stored in ancestors of v and are scheduled to be inserted into $\overline{L}^+(v)$. All insertions of bridge elements into $\overline{L}^+(v)$ are executed via Q_v. We guarantee that every element $e \in L(u)$ that is added to Q_v will be inserted into $\overline{L}^+(v)$ during $O(\log^2 n)$ following updates of $L(u)$ and $O(\log^2 n)$ following updates of $L(v)$. Segments in Q_v are inserted into $\overline{L}^+(v)$ in the same order as they are inserted into Q_v. The queue processing procedure extracts the first segment from the queue and finds its position in $\overline{L}^+(v)$. At any moment of time, there are $O(1)$ queues that contain elements of $L(u)$. Every time when $L(u)$ is updated we spend $O(1)$ time on processing each queue Q_v that contains a segment from $L(u)$. We also spend $O(1)$ time on processing a queue Q_v every time when the list $L(v)$ is updated. Every element of Q_v is processed in $O(\log n)$ time. An element of $L(u)$ is inserted into Q_v at most once per $\Theta(\log^2 n)$ updates of $L(u)$. Insertions into Q_v can be also initiated by updates of $\overline{L}^+(v)$ that will be described in Section 3.6.2, but we execute at most H insertions into Q_v per $\Theta(\log^2 n)$ insertions into $\overline{L}^+(v)$. Hence the queue Q_v always contains $O(\log n)$ segments.

Our method satisfies the following conditions: (i) for $\log^2 n$ insertions into $L(u)$ there is at most one insertion into $L'(u)$ (ii) there is always at least one regular segment $s_3 \in L(u)$ between any two consecutive special segments s_1 and s_2 from $L'(u)$ (iii) each segment $s \in L(u)$ is stored in at most two lists $L^+(v)$ and $L^+(par(v))$ for some descendant v of u.

3.6.2 Updates and Queries

Updates. We insert one segment into $L^+(u) \setminus L(u)$ for $\log^2 n$ insertions into $L(w)$ for an ancestor w of u. Every real segment (including bridges) intersects at most H phantom bridge segments: there is always a real segment $s_b \in L(w)$ between any two bridge segments $s_1 \in L(w) \cap L^+(u)$ and $s_2 \in L(w) \cap L^+(u)$. Therefore a real segment s cannot intersect two bridge segments $s_1 \in L(w) \cap L^+(u)$ and $s_2 \in L(w) \cap L^+(u)$ for the same ancestor w of u. Since u has at most H ancestors, s intersects at most H phantom bridges.

Now we show how the number of segments that intersect a phantom segment can be bounded. Every real segment $s \in L^+(u)$ intersects at most H consecutive phantom bridges. We divide phantom bridges in the list $\overline{L}^+(u)$ into blocks, so that a block contains between H and $2H$ phantom segments. Using standard techniques, we can maintain phantom blocks under insertions and deletions by splitting and merging blocks. We associate *weight*(B) with every block B; *weight*(B) is equal to the maximal number of segments that intersect a phantom segment $s \in B$. When we initiate the process of insertion of some segment s into $\overline{L}^+(u)$, s is a real segment and can intersect at most H phantom segments. Hence an insertion of a segment s into

$\overline{L}^+(u)$ increments *weight*(B) by 1 for at most two blocks B. After every series of $\log^2 n$ insertions into $L(u)$ we identify a block B of maximal weight and replace all phantom segments in B with real segments. Insertions of new bridge segments are executed using a queue Q_u described in Section 3.6.1. The cost of replacing segments can be distributed among $\log^2 n$ following updates of $L(u)$ or one of its ancestors. It can be shown that the maximal weight of a block does not exceed $\log^3 n$. Hence every phantom segment is intersected by at most $\log^3 n$ real segments.

When a segment $s \in L(u)$ is deleted, we simply remove it from $L(u)$. If s is used as a bridge in some descendant v of u, we mark s as deleted and keep a copy of s in $\overline{L}(u)$ as described above. When $s \in L^+(u) \setminus L(u)$ is deleted, we mark s as deleted.

Queries. To find a segment directly above a query point q, we visit the leaf that contains the successor of q_x, where q_x is the x-coordinate of q, and all its ancestors. In each visited node we find the segment s_u that is directly above q in $L(u)$ as follows.

First, we search in $\overline{L}^+(u)$ for the lowest segment $s_1 \in \overline{L}^+(u)$ that is above q. The search procedure is terminated when s_1 is found or when the search fails. We say that a search fails if we find two segments s_1 and s_2 such that s_1 precedes s_2 in $\overline{L}^+(u)$, but s_1 is above q and s_2 is below q. We find the first segment $s_3 \in L(u) \cap \overline{L}^+(u)$ that follows s_1 in $\overline{L}^+(u)$. Since a phantom segment is intersected by at most $\log^3 n$ segments, we need to search $O(\log^3 n)$ segments in the neighborhood of s_3 in order to find the successor s_4 of q in $L(u) \cap \overline{L}^+(u)$. Since $\overline{L}^+(u)$ contains every $\log^2 n$-th segment of $L(u)$, we only need to search in $O(\log^2 n)$ neighborhood of s_4 in order to find s_u. We can find s_4 and s_u in $O(\log\log n)$ time by finger search. When s_4 is found, we identify the bridge segment s_5 that precedes s_4 in $\overline{L}(u)$; s_5 can be found in $O(\log\log n)$ time using a union-split-find data structure.

Consider the position of s_5 in $\overline{L}^+(par(u))$. Although s_5 can be a phantom bridge, s_5 is intersected by at most H other bridges. We set $u = par(u)$ and $s_1 = s_5$. Then we find the segment s_4 that is the successor of q in $L(u)$ and the new bridge s_5 that precedes s_4 in $\overline{L}(u)$ as described above. The segment s_4 is within $H\text{polylog}(n)$ segments in the neighborhood of s_1; hence s_4 can be found in $O(\log\log n)$ time. The segment s_5 is computed in $O(\log\log n)$ time using a union-split-find data structure. The same procedure is then repeated in all nodes u. The total query time is $O(\log n + H\log\log n)$.

Applying this technique to all our data structures implies:

Theorem 3.7 *We can modify the data structures in Theorems 3.3, 3.4, and 3.5 so that all bounds are deterministic worst-case bounds.*

3.7 Trade-Off III

We can decrease the deletion time of the first trade-off if standard $(\log n)$-bit word RAM operations are allowed and update procedures use randomization. Our starting point is the data structure described in Section 3.3. To simplify the notation, we omit the subscripts when they are clear from the context. Let v_L, v_R, π_L, and π_R be defined as in Section 2.1. All occurrences of a segment s in lists $L(\cdot)$ can be divided into three groups: (1) Occurrences of s stored in the children of w, where w is the lowest common ancestor of v_L and v_R. These occurrences are colored with a special color *"M"* and will be called middle occurrences. (2) Occurrences of s stored in the nodes $u \in \pi_R$ and their left siblings; these occurrences will be called left occurrences. (3) Occurrences of s stored in the nodes $u \in \pi_L$ and their right siblings; these occurrences will be called right occurrences. We will describe below how queries and updates on middle and left occurrences are supported. Right occurrences are symmetric to left occurrences. To answer a query q we find the segment immediately above q among middle, left, and right occurrences. The answer is the lowest of three segments.

Middle occurrences are kept in the same data structures as described in Section 3.3. Every segment s occurs $O(b)$ times as a middle segment and is stored in $O(b)$ structures $L(u)$. When a new segment is inserted, we insert it into $O(b)$ lists containing *"M"*-colored segments in $O(b \log n)$ time. When s is deleted, we can remove it from all *"M"*-lists and all fractional cascading structures in $O(b \log \log n)$ time.

Left occurrences are stored as follows. We associate a tree T_u to each node u of the global tree T. Leaves of T_u correspond to children u_i of u and each internal node has $\Theta(\log \log n)^{\varepsilon'})$ children for an arbitrary constant $\varepsilon' > 0$. We keep lists $V(v)$ in every node v of T_u. $V(v)$ contains all segments s such that s covers all leaf descendants of v but does not cover at least one leaf in the subtree of $par(v)$. Our data structure uses two level of fractional cascading. For every tree T_u we keep augmented lists $V^+(v)$. $V^+(v)$ are obtained from lists $V(\nu)$ in the same way as $L^+(u)$ are obtained from lists $L(u)$, $V^+(v) = sample'(V(v)) \cup sample'(V^+(par(v))$, but $sample'(A)$ contains every $(\log \log n)^3$-th element from the set A. For children u_i of u, we set $L'(u_i) = V^+(u_i)$. Using the same techniques as described in previous sections, we can maintain fractional cascading data structure on sets $L'(u)$ for all $u \in T$. We also keep the data structure $R(u)$ described by Arge et al. [1] in each tree T_u: for any two segments s_1 and s_2 from $\bigcup_{v \in T_u} V(v)$ and for any node $\nu \in T_u$, we can identify some segment $s' \in V(\nu)$ situated between s_1 and s_2 (or report that no such s' exists) in $O(\log \log \log n)$ time. Updates of $R(u)$ are supported in $O(\log \log n)$ time.

To answer a query, we visit all nodes $u \in \pi_R$ and find segments $s_l(u)$ and $s_u(u)$ that precede and succeed the query point q in $L'(u)$ for every $u \in \pi_r$. Let u_0 be a node on π_R and let u be the parent of u. We examine all internal nodes ν on the path from u_0 to the root of T_u. In each ν we find a segment s' between $s_l(u)$ and $s_u(u)$. If s' exists we search in the $(\log \log n)$-neighborhood of s' for segments that precede and follow q in $V^+(\nu)$. We update $s_l(u)$ and $s_u(u)$ accordingly and proceed in the parent of ν. The search in T_u examines $O(\log \log n / \log \log \log n)$ nodes. Finger search in $V(\nu)$ and a query on $R(u)$ take $O(\log \log \log n)$ time. Hence we spend $O(\log \log n)$ time in any T_u for $u \in \pi_R$ and the total query time is $O(\log n)$.

We keep lists $\tilde{L}(u) = \{ S \mid s \in L(u_i) \text{ for a child } u_i \text{ of } u \}$ in all internal nodes u and all colors except for *"M"*. When a new segment is inserted we find its position in $O(H)$ lists $\tilde{L}_i(u)$ as described in Section 3.3. For every node u, we identify children u_i of u covered by the new segment s and nodes $\nu \in T_u$ in which s must be stored. Since every $V(\nu)$ is a subset of $\tilde{L}(u)$, we can find the position of s in each $V(\nu)$ in $O(\log \log n)$ time using a union-split-find data structure. We also update the data structure $R(u)$. Since s is kept in $O((\log \log n)^{1+\varepsilon'})$ nodes of T_u, updates of union-split-find data structures and $R(u)$ take $O((\log \log n)^{2+2\varepsilon'})$ time. Auxiliary data structures in all nodes u are updated in $O(\log_b n (\log \log n)^{2+2\varepsilon'})$ time. We need $O(\log^{1+\varepsilon} n)$ time to find the position of s in all $\tilde{L}_i(u)$. Hence the total insertion time is $O(\log^{1+\varepsilon} n)$. Deletions are symmetric, but we do not have to find the position of s in lists $\tilde{L}_i(u)$. Hence the total deletion time is $O(\log_b n (\log \log n)^{2+2\varepsilon'})$. By setting $\varepsilon' = \varepsilon/2$, the deletion time is $O(\log n (\log \log n)^{1+\varepsilon})$.

We describe a derandomized variant of the data structure $R(u)$. We divide the segments of $L'(u)$ into blocks of size $\Theta(\log^2 n)$. Let m be the number of segments in $L'(u)$. Using the labeling scheme of Willard [34], we can assign positive integer labels $lab(G) \in [1, O(m/\log^2 n)]$ to every block G. Given two segments s_1 and s_2 from the same block G and a node $\nu \in T_u$, we can identify a segment $s \in V(\nu)$ or report that there is no such s. A data structure that supports such queries and updates on G in $O(1)$ time can be implemented using standard bit operations. Furthermore we keep a data structure *Rep*(ν) in every $\nu \in T_u$. *Rep*(ν) contains block labels of all segments in $V(\nu)$. For any two labels l_1 and l_2, *Rep*(ν) can find a segment s' from a block G' such that $l_1 \leq lab(G') \leq l_2$ or determine that no such $s' \in V(\nu)$ exists. We implement $Rep(\nu)$ using the one-dimensional reporting data structure of Mortensen et al. [24]. Their data structure uses dynamic dictionaries over a universe of size U. These dynamic dictionaries use linear space and rely on randomized update procedures. In our modified data structure we implement dictionaries using bit vectors. Hence the space usage of $Rep(\nu)$ is $O(m/\log^2 n)$ (to be compared with $O(|V(v)|)$ space used by the original data structure of Mortensen et al. [24]), but the update procedures are deterministic. A query on $R(u)$ is answered as follows. Given two segments $s_1 \in L(u)$ and $s_2 \in L(u)$ and a node $\nu \in T_u$, we can find their groups $G_1 \ni s_1$ and $G_2 \ni s_2$. We check if there is a segment $s' \in G_1 \cap V(\nu)$ such that s' is above s_1 or $s' \in G_2 \cap V(\nu)$ such that s' is below s_2. If there is no such s' we look for a segment s' such that the label of its block is in $[lab(G_1)+1, lab(G_2)-1]$.

Theorem 3.8 *There is an $O(n)$-space data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log n)$ worst-case time and support insertions and deletions of segments in $O(\log^{1+\varepsilon} n)$ and $O(\log n(\log\log n)^{1+\varepsilon})$ worst-case time respectively for any constant $\varepsilon > 0$ in the RAM model.*

4 An Alternative Recursive Approach

Let S be a set of at most N non-intersecting line segments, with x-coordinates from a static set X of size n, with $N \geq n$. We will describe how to support insertions to X later in Section 5.2.

The *y-successor* (resp. *y-predecessor*) of a point q in a set S refers to the segment in S immediately above (resp. below) q. An *elementary slab* refers to a vertical slab with x-coordinates defined by two consecutive values in X.

Let $sample_m(S)$ denote a random sample of S where each element is selected independently with probability $1/m$. Note that inside any elementary slab, two consecutive elements of $sample_m(S)$ are separated by $O((|S|/m)\log N)$ elements of S w.h.p.

To obtain a recursive solution, we need the right interface. The key is to define the following version of the query problem for some sufficiently large constant c:

$(*)$ Given a query point q and the y-successor s_0 of q in $sample_{n^c}(S)$, find the y-successor and y-predecessor of q in S.

At the beginning, $N = O(n)$ and $sample_{n^c}(S)$ has subconstant expected size, so s_0 is trivially known.

4.1 Data Structure

We divide the plane into $\sqrt{n}$ vertical slabs $\sigma_1, \ldots, \sigma_{\sqrt{n}}$, each with $\sqrt{n}$ x-coordinates from X. For each segment $s \in S$ that is not completely inside any σ_i, we divide s into three subsegments: the *left (resp. right) subsegment* refers to the portion of s in the slab containing the left (resp. right) endpoint, and the *middle subsegment* refers to the remaining portion of s.

1. Let L_i (resp. R_i) be the set of all left (resp. right) subsegments of S inside σ_i. We recursively build a data structure for each L_i and R_i.

2. Let M be the set of all middle subsegments of S. We recursively build a data structure for M, wich has $\sqrt{n}$ possible x-coordinates.

3. Let S_i be the subset of segments of S that are completely inside σ_i. We recursively build a data structure for each subset S_i, which has $\sqrt{n}$ possible x-coordinates.

4. For each elementary slab inside σ_i, we maintain *finger search trees* [21] for the y-ordered lists $\mathit{sample}_{n^{c/2}}(L_i)$, $\mathit{sample}_{n^{c/2}}(R_i)$, $\mathit{sample}_{n^{c/2}}(M)$, and $\mathit{sample}_{n^{c/2}}(S_i)$, and also van Emde Boas or *union-split-find* structures [22] to support $O(\log\log N)$-time predecessor/successor search in a list for query elements from another list. The total expected space over all $O(n)$ elementary slabs is $O((N/n^{c/2}) \cdot n) = o(N)$.

Notice that in each recursive subproblem in step 1, the input set is *1-sided*, i.e., the left or right endpoints of all segments have a common x-coordinate. In the case when S is 1-sided, then one of L_i and R_i is empty, M is 1-sided, and S_i is empty. When S is 1-sided, the segments can be globally y-ordered and we additionally keep van Emde Boas structures to support $O(\log\log N)$-time predecessor/successor search in each list L_i or R_i, and M, for query elements from S.

In the base case $n = 2$, we just maintain S in a finger search tree.

4.2 Query Algorithm

To solve the query problem $(*)$ for a given point q and segment s_0:

1. Suppose that q is in σ_i. We first find the y-successors of s_0 in $\mathit{sample}_{n^{c/2}}(L_i)$, $\mathit{sample}_{n^{c/2}}(R_i)$, $\mathit{sample}_{n^{c/2}}(M)$, and $\mathit{sample}_{n^{c/2}}(S_i)$ in the elementary slab containing q, using the union-split-find structures, which cost $O(\log\log N)$ time. Using these results as fingers, we then find the y-successors of q in $\mathit{sample}_{n^{c/2}}(L_i)$, $\mathit{sample}_{n^{c/2}}(R_i)$, $\mathit{sample}_{n^{c/2}}(M)$, and $\mathit{sample}_{n^{c/2}}(S_i)$, by finger search in $O(\log(n^c \log N)) = O(\log n + \log\log N)$ time w.h.p., since there are $O(n^c \log N)$ elements of S between q and s_0.

2. We recursively query the data structure for L_i and R_i.

3. We recursively query the data structure for M.

4. We recursively query the data structure for S_i, finally returning the lowest y-successor and highest y-predecessor of q found.

For the analysis, we first consider the case when S is 1-sided. Here, steps 2–4 require two recursive calls, since one of L_i and R_i is empty and S_i is empty. Thus, the expected query time satisfies the recurrence

$$Q_1(n) \;=\; 2\,Q_1(\sqrt{n}) + O(\log n + \log\log N), \tag{2}$$

with the base case $Q_1(2) = O(\log n + \log\log N)$. The total contribution of the $\log n$ term is $O(\log n \log\log n)$, whereas the total contribution of the $\log\log N$ term is $O(\log n \log\log N)$. Hence, $Q_1(n) = O(\log n \log\log N)$.

For the general case, step 2 takes $Q_1(\sqrt{n})$ expected time, and steps 3–4 require two recursive calls. Thus, the expected query time satisfies the recurrence

$$Q(n) \;=\; 2\,Q(\sqrt{n}) + Q_1(\sqrt{n}) + O(\log n + \log\log N) \;=\; 2\,Q(\sqrt{n}) + O(\log n \log\log N), \tag{3}$$

with the base case $Q(2) = O(\log n + \log\log N)$. Hence, $Q(n) = O(\log n \log\log n \log\log N) = O(\log n (\log\log N)^2)$.

4.3 Update Algorithm

To insert/delete a segment s in S:

1. We recursively insert/delete its left and right subsegments in L_i and R_i.
2. We recursively insert/delete its middle subsegment in M.
3. If s is completely inside σ_i, we recursively insert/delete s in S_i.
4. If s is in any of the lists $\mathit{sample}_{n^{c/2}}(L_i)$, $\mathit{sample}_{n^{c/2}}(R_i)$, $\mathit{sample}_{n^{c/2}}(M)$, and $\mathit{sample}_{n^{c/2}}(S_i)$, we update their associated data structures for all $O(n)$ elementary slabs in $O(n \log N)$ time.

For the analysis, we first consider the case when S is 1-sided. We assume that the position of s in in the global y-order of S is given, which can be found by an initial binary search in $O(\log N)$ time. We can then find the position of s in L_i or R_i, and M, in $O(\log \log N)$ time. Since one of L_i and R_i is empty and S_i is empty, steps 1–3 require two recursive calls. Step 4 is done with probability $O(1/n^{c/2})$ and so has expected cost $O((1/n^{c/2}) \cdot n \log N) = o(\log N)$ for a sufficiently large constant c. Thus, the expected update time, excluding the initial binary search, satisfies the recurrence

$$U_1(n) \;=\; 2\, U_1(\sqrt{n}) + O(\log \log N), \tag{4}$$

with the base case $U_1(2) = O(1)$. Hence, $U_1(n) = O(\log n \log \log N)$.

For the general case, step 1 takes $O(\log N) + 2U_1(\sqrt{n})$ expected time, steps 2–3 require only one recursive call, since s can contribute to M or S_i but not both, and step 4 has expected cost $O((1/n^{c/2}) \cdot n \log N) = o(\log N)$. Thus, the expected update time satisfies the recurrence

$$U(n) \;=\; U(\sqrt{n}) + 2\, U_1(\sqrt{n}) + O(\log N) \;=\; U(\sqrt{n}) + O(\log n \log \log N + \log N), \tag{5}$$

with the base case $U(2) = O(1)$. The total contribution of the $\log n \log \log N$ term is $O(\log n \log \log N)$, whereas the total contribution of the $\log N$ term is $O(\log N \log \log N)$. Hence, $U(n) = O(\log n \log \log N)$. We have therefore reproved Theorem 2.1.

5 Refinements of the Recursive Approach

5.1 Improving the Query Time

We now describe a modification of our method in Section 4 to remove one $\log \log N$ factor from the query time. The improvement comes from the 1-sided case. We first need a subroutine for a decision version of the 1-sided problem:

Lemma 5.1 *We can maintain a set Z of $O(N)$ non-intersecting 1-sided segments with n distinct x-coordinates, and a subset S of Z, so that given any two segments $s, s' \in Z$ and a value q_x, we can decide whether there exists a segment in S between s and s' at x-coordinate q_x in $Q_{\text{1-decis}}(n) = O(\log n + \log \log N)$ time in the RAM model. An update to S or Z takes $U_{\text{1-decis}}(n) = O(\log n + \log \log N)$ time, given the position of the segment being updated in the global y-order of Z.*

Proof: Suppose that all left endpoints have a common x-coordinate. Map each segment $s \in Z$ to a point (s_x, s_y) where s_x is the x-coordinate of the right endpoint and s_y is the y-coordinate of the left endpoint. Then the problem reduces to *3-sided orthogonal range emptiness*: deciding whether $[q_x, \infty) \times (s_y, s'_y)$ contains a point from the image of S.

This problem can be solved using a *dynamic priority search tree* (e.g., [12]), which has $O(\log N)$ query and update time, but we want better time bounds that depend on the number n of distinct x-coordinates. The problem can alternatively be solved using a *dynamic range tree* with *fractional cascading* [22], which achieves $O(\log n \log\log N)$ query and update time. To remove the extra $\log\log N$ factor, we can use a range tree with a larger fan-out w^{ε}, where $w = \log N$ is the word size—various such structures for orthogonal problems have been proposed by several researchers [6, 19, 23] using generalized union-split-find structures. These techniques improve the query and update time to $O((\log_w n + 1)\log\log N) = O(\log n + \log\log N)$ as desired on the RAM. $\square$

For our data structure, when S is 1-sided, we include the auxiliary structure from Lemma 5.1 for L_i or R_i, and M.

For the query algorithm, when S is 1-sided, we adapt a *randomized search* technique by Chan [7]: We imagine expanding the recursion for some constant c_0 number of levels, giving rise to $C = 2^{c_0}$ subproblems, each with $n^{1/C}$ x-coordinates, caused by steps 2 and 3. We randomly permute the order in which we perform these K subproblems (since these subproblems can be executed in any order). In step 2, we perform the recursive call only if there exists a segment of L_i or R_i between s and s', where s is the current lowest y-successor of q and s' is the current highest y-predecessor of q found so far—a condition testable by Lemma 5.1. In step 3, we proceed similarly. For the i-th subproblem considered ($i = 1, \ldots, C$), we thus need the recursive call only if its y-successor is the lowest of the y-successors among the first i subproblems, or its y-predecessor is the lowest of the y-predecessors among the first i subproblems—this holds with probability at most $2/i$. Hence, the expected number of recursive calls is at most $\sum_{i=1}^{C} 2/i = O(\log C)$. The recurrence (2) for the expected query time in the 1-sided case then improves to

$$\begin{aligned} Q_1(n) &= O(\log C)\, Q_1(n^{1/C}) + C\, Q_{\text{1-decis}}(n^{1/C}) + O(\log n + \log\log N) \\ &= O(\log C)\, Q_1(n^{1/C}) + O(C(\log n + \log\log N)), \end{aligned}$$

which solves to $Q_1(n) = O(\log n + \log^{\varepsilon} n \log\log N)$ for a sufficiently large constant C. The recurrence (3) in the general case is now

$$Q(n) \;=\; 2\,Q(\sqrt{n}) + O(\log n + \log^{\varepsilon} n \log\log N),$$

which solves to $Q(n) = O(\log n \log\log n + \log n \log\log N) = O(\log n \log\log N)$.

For the update time, the recurrence (4) for the 1-sided case is now

$$U_1(n) \;=\; 2\,U_1(\sqrt{n}) + 2\,U_{\text{1-decis}}(\sqrt{n}) + O(\log\log N) \;=\; 2\,U_1(\sqrt{n}) + O(\log n + \log\log N), \qquad (6)$$

which solves to $U_1(n) = O(\log n \log\log n + \log n \log\log N) = O(\log n \log\log N)$, exactly as before. So, for the general case, the same recurrence (5) still gives $U(n) = O(\log n \log\log N)$.

Theorem 5.2 *There is a data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log n \log\log n)$ expected time and support insertions and deletions of segments in $O(\log n \log\log n)$ expected time in the RAM model, assuming that all x-coordinates are from a static set X of size $O(n)$.*

5.2 Dynamizing X

Before discussing how to support insertions to X, we first discuss the preprocessing time for our data structure, which is trivially bounded by $O(N \cdot U(n)) = O(N \log n \log\log N)$.

In the 1-sided case, we can reduce the preprocessing time by a standard blocking technique [3]. Suppose that all left endpoints have a common x-coordinate. We divide the y-ordered list S of segments into $O(N/B)$ blocks of size $\Theta(B)$ with $B = \log^2 N$, and maintain a subset S' containing the segment with the rightmost right endpoint in each block. Then S' has size $O(N/B)$. We maintain our data structure for S', which has $O((N/B)\log n \log\log N) = o(N)$ preprocessing time. Each block can be kept in a naive data structure with $O(\log^{O(1)} B) = O(\log\log N)$ query and update time and linear preprocessing time. The total preprocessing time is $O(N)$, given the global y-order of S.

To answer a query, we answer the query for S' and then search in the blocks containing these answers and their neighboring blocks, with additional $O(\log\log N)$ time. (Technically, we also need to use s_0 to first find the y-successor of q in S' in $\mathit{sample}_{(n/B)^c}(S')$, but this is similar to step 1 of our query algorithm, requiring $O(\log n + \log\log N)$ time by finger search and union-split-find structures.)

For the general case, we can speed up the preprocessing algorithm by pre-computing a topological order of S, with the property that if s is below s' at some vertical line, s appears before s'; this takes $O(N\log N)$ time [29]. Afterwards, the preprocessing time is linear excluding the recursion to M and the S_i's. Each segment either contributes to M or an S_i, but not both. As the recursion has $O(\log\log n)$ levels, the total expected preprocessing time is $O(N\log N + N\log\log n) = O(N\log N)$.

Now, to support insertions of new x-values to the set X, we apply standard techniques (for example, found in weight-balanced B-trees [2]). We require that each value in X is the x-coordinate of $O(N/n)$ endpoints in S. When dividing into $\Theta(\sqrt{n})$ slabs, we maintain that each slab σ_i has $\Theta(N/\sqrt{n})$ endpoints. (Replacing $\sqrt{n}$ with $\Theta(\sqrt{n})$ in the recurrences does not affect their solutions.) In inserting a value to X, when this invariant is violated, we split σ_i into two slabs and rebuild the data structure for the portion of the segments inside the slabs, by our preceding preprocessing algorithm. This requires $O(N/\sqrt{n})$ time in the 1-sided case, and $O((N/\sqrt{n})\log N)$ time in the general case, but is done $O(\sqrt{n})$ times over N segment updates. Thus, the extra amortized cost per segment update is $O(1)$ for the 1-sided case, and $O(\log N)$ for the general case. Furthermore, each insertion to X causes rebuilding of structures associated with $O(1)$ elementary slabs, which requires $O((N/n^{c/2})\log N)$ time but is done $O(n)$ times over N segment updates. Thus, the extra amortized expected cost per segment update is $o(1)$. Adding an $O(1)$ term to (4) or (6) and an $O(\log N)$ term to (5) does not change the recurrences of the update time.

When n is increased or decreased by a factor of 2, we can rebuild the entire data structure from scratch. (All this can be deamortized with more effort.)

5.3 Other Issues

The space usage of the data structure in Theorem 5.2 can be immediately reduced to $O(n)$ by the general technique from Section 3.2. We conclude:

Theorem 5.3 *There is an $O(n)$-expected-space data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log n \log\log n)$ expected time and support insertions and deletions of segments in $O(\log n\log\log n)$ expected time in the RAM model.*

It is unclear how to implement the improved query algorithm in the pointer machine model, because $(\log N)$-bit word RAM operations are needed in Lemma 5.1 (and also in the known dynamic data structure for horizontal segments used for the space-reduction part in Section 3.2). It is also unclear how to derandomize the randomized search used in the improved query algorithm.

6 Final Remarks

Our method can be immediately generalized to handle non-intersecting x-monotone curve segments.

Removing all the $\log\log n$ factors in the query and update time of our main result remains very challenging. It is not clear how to remove them even in terms of decision tree complexity, i.e., counting only the number of comparisons of input x-coordinates and point–segment comparisons, and not the cost of other operations. In the comparison model, $\Omega(\log n)$ is a lower bound on the query cost, and by standard arguments in one dimension, $\Omega(\log n)$ is a lower bound on the update cost for any data structure with $o(n^{1-\varepsilon})$ query cost, at least for vertical ray shooting for general non-intersecting segments. However, we are not sure if $\Omega(\log n)$ is a lower bound on the update cost for point location in connected subdivisions.

Certain dynamic data structures for planar point location can be adapted to yield static data structures for three-dimensional point location in a convex subdivision [20, 31]. However, our method does not appear to have any implication to the three-dimensional problem, where finding a near-linear-space data structure with near-logarithmic query time remains open.

It is unclear how to solve the dynamic point location problem for subdivisions that are not connected, in particular, how to test whether two query points are in the same face when faces may have holes. The unresolved issue here (mentioned in [12]) concerns how to test whether two edges are part of the same face.

References

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

[2] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

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

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

[5] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[6] Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proceedings of the 19th Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 894–903, 2008.

[7] Timothy M. Chan. Geometric applications of a randomized optimization technique. *Discrete Comput. Geom.*, 22(4):547–567, 1999.

[8] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th Annual ACM Symposium on Computational Geometry*, pages 1–10, 2011.

[9] Bernard Chazelle. Computational geometry for the gourmet: Old fare and new dishes. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, pages 686–696, 1991.

[10] Bernard Chazelle. Computational geometry: A retrospective. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 75–94, 1994.

[11] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[12] Siu-Wing Cheng and Ravi Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21(5):972–999, 1992.

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

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

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

[16] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 3rd edition, 2008.

[17] Paul F. Dietz and Daniel Dominic Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, 1987.

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

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

[20] Michael T. Goodrich and Roberto Tamassia. Dynamic trees and dynamic point location. *SIAM J. Comput.*, 28(2):612–636, 1998.

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

[22] Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990.

[23] Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35(6):1494–1525, 2006.

[24] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pătraşcu. On dynamic range reporting in one dimension. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 104–111, 2005.

[25] Ketan Mulmuley. A fast planar partition algorithm, I. *J. Symb. Comput.*, 10(3/4):253–280, 1990.

[26] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Dynamic data structures for document collections and graphs. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems*, pages 277–289, 2015.

[27] Yakov Nekrich. Searching in dynamic catalogs on a tree. *CoRR*, abs/1007.3415, 2010.

[28] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inf. Process. Lett.*, 12(4):168–173, 1981.

[29] Larry Palazzi and Jack Snoeyink. Counting and reporting red/blue segment intersections. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, pages 530–540, 1993.

[30] Franco P. Preparata and Roberto Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18(4):811–830, 1989.

[31] Franco P. Preparata and Roberto Tamassia. Efficient point location in a convex spatial cell-complex. *SIAM J. Comput.*, 21(2):267–280, 1992.

[32] Jack Snoeyink. Point location. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 34, pages 767–787. CRC Press LLC, Boca Raton, FL, 2nd edition, 2004.

[33] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[34] Dan E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Inf. Comput.*, 97(2):150–204, 1992.