Improved Dynamic Planar Point Location
(Extended Abstract)

Lars Arge* Gerth Stølting Brodal Loukas Georgiadis*
Department of Computer Science, University of Aarhus, Denmark
E-mail: {large,gerth,loukas}@daimi.au.dk

Abstract

We develop the first linear-space data structures for dynamic planar point location in general subdivisions that achieve logarithmic query time and polylogarithmic update time.

1 Introduction

Point location is a fundamental problem in computational geometry, with a plethora of applications in areas such as geographical systems, graphics, and databases [10]. In the static case we are given a planar subdivision Π with n segments, i.e., a decomposition of the plane into polygonal regions induced by a straight-line planar graph. The goal is to preprocess Π into a data structure, so that we can efficiently answer queries of the form: Given a query point q, find the face of Π containing q. In the dynamic case, Π can be updated by inserting or deleting segments.

Both static and dynamic planar point location has been studied extensively. For the static case, several linear-space and $O(\log n)$-query structures have been designed [16, 24]. However, for the dynamic and insertion-only cases, no nontrivial $O(\log n)$-query structure is known, even using super-linear space. In this paper we develop linear-space and $O(\log n)$-query structures.

Previous results. Two linear-space structures are known for dynamic planar point location in general subdivisions; one by Cheng and Janardan [6] that supports queries in $O(\log^2 n)$ time and updates in $O(\log n)$ time, and one by Baumgarten, Jung and Mehlhorn [1] that supports queries and insertions in $O(\log n \log\log n)$ time and deletions in $O(\log^2 n)$ time. Both structures store the segments of Π in an interval tree [12] constructed on their projection on the x-axis, and use this structure to answer *vertical ray-shooting queries*: For a query point q, find the first segment of Π hit by the ray emanating from q in the $(+y)$-direction. After answering this query, the face containing q can be found in $O(\log n)$ time [21]. A number of super-linear space structures as well, as structures for restricted subdivisions, have also been developed. Refer to Table 1 and the survey in [8]. Note that $O(\log n)$ query time is achieved only in the case of monotone and connected subdivisions; in both cases the developed structures use $O(n \log n)$ space. For the insertion-only problem, the structure of Baumgarten et al. [1] achieves the best known bounds for general subdivisions; even for monotone subdivisions, no linear-space and $O(\log n)$-query structure is known. Refer to Table 2.

Our results. We develop the first linear-space data structures for dynamic planar point location in general subdivisions (or more specifically, for dynamic vertical ray-shooting), that achieve logarithmic query time (and $o(n)$ update time). Our results are summarized in Table 3. In the fully-dynamic case, we improve the best previous query bound of Baumgarten et al. [1] by a $\log\log n$ factor, while increasing the update time by a factor of $\log^{\varepsilon} n$. This structure is implementable in the pointer-machine model of computation. In the random-access machine (RAM) model, we improve the deletion bound by a $\log\log n$ factor, while only increasing the insertion bound by a $\log^{\varepsilon}\log n$ factor. In the incremental case, we obtain both $O(\log n)$-time queries and insertions in the RAM model. On a pointer machine, we obtain a structure with $O(\log^{1+\varepsilon} n)$ insertions and $O(\log n)$ queries, as well as a structure with $O(\log n)$ insertions and $O(\log n \log^* n)$ queries.

2 Achieving logarithmic query time

The point location structure of Baumgarten et al. [1] resembles an interval tree [12]. It consists of a binary

*Supported in part by US Army Research Office grant W911NF-04-1-0278 and an Ole Roemer Scholarship from the Danish National Science Research Council.

Table 1. Known structures; n is the number of segments in the subdivision and n' is the number of possible y-coordinates for edge endpoints; † indicates an amortized bound. All data structures are implementable on a pointer-machine.

Subdivision	Space	Query	Insert	Delete	Reference
Horizontal	$n \log n$	$\log n \cdot \log\log n$	$\dagger \log n \cdot \log\log n$	$\dagger \log n \cdot \log\log n$	[18]
Convex	$n' + n \log m$	$\log n + \log n'$	$\log n \cdot \log n'$	$\log n \cdot \log n'$	[23]
Monotone	$n \log n$	$\log n$	$\log^2 n$	$\log^2 n$	[9]
Monotone	n	$\log^2 n$	$\log n$	$\log n$	[14]
Connected	n	$\log^2 n$	$\log^4 n$	$\log^4 n$	[13]
Connected	$n \log n$	$\log n$	$\log^3 n$	$\log^3 n$	[7]
General	$n \log n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	[2]
General	n	$\log^2 n$	$\log n$	$\log n$	[6]
General	n	$\log n \cdot \log\log n$	$\dagger \log n \cdot \log\log n$	$\dagger \log^2 n$	[1]

Table 2. Known insertion-only structures; † indicates an amortized bound.

Subdivision	Model	Space	Query	Insert	Reference
Horizontal	Pointer Machine	$n \log n$	$\log n \cdot \log^* n$	$\dagger \log n \cdot \log^* n$	[15, 18]
Horizontal	RAM	$n \log n$	$\log n$	$\dagger \log n$	[15, 18]
Monotone	Pointer Machine	n	$\log n \cdot \log\log n$	$\dagger$ 1	[14]
General	Pointer Machine	n	$\log^2 n$	$\dagger \log^2 n$	[3, 24]
General	Pointer Machine	n	$\log n \cdot \log\log n$	$\dagger \log n \cdot \log\log n$	[1]

Table 3. New structures; † and ‡ indicates amortized and randomized bounds, repectively.

Subdivision	Model	Space	Query	Insert	Delete
General	Pointer Machine	n	$\log n$	$\dagger \log^{1+\varepsilon} n$	$\dagger \log^{2+\varepsilon} n$
General	RAM	n	$\ddagger \log n$	$\dagger \ddagger \log n \cdot \log^{1+\varepsilon} \log n$	$\dagger \ddagger \log^2 n / \log\log n$
General	Pointer Machine	n	$\log n$	$\dagger \log^{1+\varepsilon} n$	
General	Pointer Machine	n	$\log n \cdot \log^* n$	$\dagger \log n$	
General	RAM	n	$\log n$	$\dagger \log n$	

base tree $\mathcal{T}$ over the x-coordinates of the segments in the subdivision Π, with the actual segments stored in secondary structures attached to each internal node. To answer a query, the $O(\log n)$ secondary structures on a root-leaf path in $\mathcal{T}$ are searched. Using fractional cascading [5], Baumgarten et al. show how each secondary structure can be queried in $O(\log\log n)$ time, obtaining an $O(\log n \cdot \log\log n)$ total query bound.

Intuitively, the main idea in our structure is to increase the fan-out of the base tree to $\log^\varepsilon n$ such that its height is reduced to $O(\log n / \log\log n)$. This way we obtain an overall $O(\log n)$ query bound, provided that we can maintain the $O(\log\log n)$ secondary structure query bound. More precisely, our structure consists of a fan-out $\Theta(\log^\varepsilon n)$ base search tree structure $\mathcal{T}$ on the x-coordinates of the segments in Π; each leaf of $\mathcal{T}$ stores $\Theta(\log^\varepsilon n)$ x-coordinates. Here, $0 < \varepsilon < 1/2$ is some arbitrary fixed constant. For simplicity, we assume that all $O(n)$ possible endpoints are known in advance, so that we can keep the base tree fixed (i.e., we do not have to update it during insertions and deletions of segments). Also, to simplify our notation, we assume that each internal node has exactly $f = \log^\varepsilon n$ children and each leaf stores f endpoints. We can remove these assumptions using standard techniques; details will appear in the full paper.

With each node v in $\mathcal{T}$ we associate a range $\mathit{range}(v) = [\mathit{left}(v), \mathit{right}(v)]$ consisting of the range of x-coordinates stored in the subtree rooted in v. Furthermore, with each internal node v we associate $f+1$ vertical lines $\ell_j(v)$, $0 \le j \le f$, corresponding to the range boundaries of its children: $\ell_0(v) = \mathit{left}(v)$ and, for $1 \le j \le f$, $\ell_j(v) = \mathit{right}(v_j)$. We refer to the f ranges of the children of v as *slabs* of v and to the vertical lines $\ell_j(v)$ as *slab boundaries* of v. The f slabs of v define $\Theta(f^2)$ *slab intervals* $I(v,i,j)$, $1 \le i \le j \le f$, where $I(v,i,j)$ denotes the range between $\ell_{i-1}(v)$ and $\ell_j(v)$.

Three secondary structures are associated with each internal node v in $\mathcal{T}$: A left structure $\mathcal{L}(v)$, a right structure $\mathcal{R}(v)$, and a middle structure $\mathcal{M}(v)$. Each segment s in Π is stored in a left structure, a right

structure, and possibly a middle structure. Consider the highest node v such that the projection *proj*(s) of s on the x-axis intersects one of the vertical lines $\ell_k(v)$. Suppose the left endpoint of s is in *range*(v_i) and the right endpoint of s is in *range*(v_j), of children v_i and v_j of v. Then s is broken at $\ell_i(v) = \mathit{right}(v_i)$ and at $\ell_{j-1}(v) = \mathit{left}(v_j)$ into (at most) three pieces: a left piece s^- stored in $\mathcal{L}(v_i)$, a middle piece s^0 stored in $\mathcal{M}(v)$, and a right piece s^+ stored in $\mathcal{R}(v_i)$. Refer to Figure 1. Note that all segments in $\mathcal{L}(v)$ have their right endpoint on $\ell_f(v)$, and that all segments in $\mathcal{R}(v)$ have their left endpoint on $\ell_0(v)$; all segments stored in $\mathcal{M}(v)$ have both endpoints on vertical lines $\ell_j(v)$, $0 \le j \le f$.

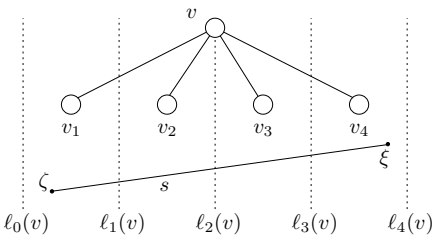

Figure 1. Segment s is broken into three pieces: The left piece s^-, from ζ to $\ell_1(v)$, is stored in $\mathcal{L}(v_1)$; the middle piece s^0, from $\ell_1(v)$ to $\ell_3(v)$, is stored in $\mathcal{M}(v)$; the right piece s^+, from $\ell_3(v)$ to ξ, is stored in $\mathcal{R}(v_4)$.

Now let Π^-, Π^0 and Π^+ denote the sets of left, middle and right segment pieces respectively, formed at all nodes of $\mathcal{T}$. To answer a query we answer it on each of the three sets individually and return the closest of the three segments we found. Intuitively, we answer the query on each of the sets as in the Baumgarten et al. structure [1] by querying secondary structures on one root-leaf path. In §4 and §5 we describe the structures for Π^0 and Π^+, respectively; the structure for Π^- is symmetric to the structure for Π^+. However, before doing so, we discuss fractional cascading on a segment tree in §3.

3 Segment Tree Fractional Cascading

In the structures for Π^0 and Π^+ described in §4 and §5 we will utilize fractional cascading, and especially fractional cascading used on various (high fan-out) segment tree structures, a number of times. Below we review fractional cascading and segment trees, and outline how we do fractional cascading on high fan-out segment trees.

3.1 Fractional cascading

Fractional cascading [5] (FC) is a technique for performing efficient *iterative search*. In the static case, iterative search takes place on a graph $G = (V, E)$, where each vertex has an associated catalog $C(v)$ with elements from a totally ordered universe. A query consists of an element e and a subgraph $G' = (V', E')$ of G, and the goal is to report the predecessor of e in $C(v)$, for each $v \in V'$. Using FC, query efficiency is accomplished by maintaining an *augmented* catalog $AC(v) \supseteq C(v)$ at each node $v \in V$; each augmented catalog $AC(v)$ contains *proper* elements $e \in C(v)$ and *non-proper* elements $e \in AC(v) \setminus C(v)$. The augmented catalogs of neighboring vertices in G have elements in common. These elements are linked together to form "bridges", which are used to alleviate the need for a full search in each $C(v)$.

In the dynamic case where elements can be inserted into or deleted from a catalog, Mehlhorn and Näher [18] showed how an FC technique can also be used to increase query efficiency. Their dynamic FC technique is based on a data structure for the dynamic version of the *interval union-split-find* (USF) problem: Maintain an ordered list of elements under the operations of inserting or deleting an element at a given location, and marking or un-marking a given element, such that the marked element closest to a given element can be found efficiently. All these operations can be performed in $O(\log \log n)$ time and $O(n)$ space on a pointer machine [17], which is optimal [19]. Imai and Asano [15] considered the case where only insertions and marking of elements is allowed. They achieved $O(1)$ time (amortized for markings and insertions) on a RAM and $O(\log^* n)$ time (amortized for markings and insertions) on a pointer machine.

3.2 Segment trees

Segment trees are $O(n \log n)$ space structures that can be used to solve the planar point location problem. A segment tree for the segments in the subdivision Π consists of a balanced binary search tree T on the x-coordinates of the segments, with the actual segments stored in lists $S(v)$ associated with each node v in T. As previously, we associate a range *range*(v) with v in a natural way. A segment s is stored in $S(v)$ iff *proj*(s) $\supseteq$ *range*(v) and *proj*(s) $\not\supseteq$ *range*(*parent*(v)). It follows that each segment is stored at $O(\log n)$ nodes.

A point location query q can be answered by querying the $O(\log n)$ lists $S(v)$ on a root-leaf path of T. Since the segments in a $S(v)$ can be totally ordered by their intersections with one of the boundaries of

range(v), a query on $S(v)$ can be answered in $O(\log n)$ time, resulting in $O(\log^2 n)$ total query time.

Although standard dynamic FC cannot be applied to the segment tree structure (since segments in different $S(v)$ lists are not totally ordered), Baumgarten et al. [1] showed how a modified version of dynamic FC can be used to improve the query bound. The main idea is to store an augmented segment list $AS(v)$ at each node v in T, consisting of $S(v)$ augmented with segments ("bridges") sampled from ancestors of v. The $AS(v)$ lists of all nodes of T are constructed top-down: At the root r, $AS(r) = S(r)$; then at node v (starting at $v = r$), $AS(v)$ is partitioned into blocks of size $\beta = \Theta(\log^2 n)$ and the median segment in each block is chosen as a *representative* and copied to the AS-lists of the children of v. Queries are then performed on the AS-lists of a root-leaf path in T from the leaf towards the root, while the augmented segments are used to alleviate the need for full search in each $AS(v)$. Using the USF results mentioned in §3.1, Baumgarten et al. [1] showed how this leads to an $O(\log n \cdot \log\log n)$ query bound.

In the use of FC on a segment tree, a technical complication arises from the fact that a single segment can appear in $\Theta(n)$ AS-lists. This means that when performing a deletion one cannot afford to remove a segment from all the lists it appears in. Baumgarten et al. [1] used the idea of *phantom segments* to deal with this problem: If a segment s is to be deleted, then it is removed from $AS(v)$ of the $O(\log n)$ nodes v where $s \in S(v)$. The remaining occurrences of s (bridges) are left as phantom segments. This in turn leads to complications since later inserted segments may intersect phantom segments. However, using the fact that sampled segments were medians of blocks, Baumgarten et al. [1] showed how such intersections can be handled.

3.3 High fan-out trees: No phantoms

As mentioned, in later sections we will utilize various implementations of high fan-out segment trees. Such a tree for the segments of Π consists of a fan-out $f = \Theta(\log^\varepsilon n)$ balanced search tree T on the x-coordinates of the segments, with the actual segments stored in a structure $S(v)$ associated with each node v in T. A segment s is stored in $S(v)$ iff *proj*(s) $\supseteq$ *range*(v_i) for one of more children v_i of v, and *proj*(s) $\not\supseteq$ *range*(v). Each segment s in $S(v)$ is shortened to the largest slab interval $I(v, i, j)$ it contains, that is, s is shortened such that it has endpoints on the slab boundaries $\ell_{i-1}(v)$ and $\ell_j(v)$. It follows that each segment is stored in at most two nodes on each level of T. Note that intuitively segments are assigned to nodes in the same way as in a high fan-out interval tree, except that only the middle segment pieces are stored in a node v; the rest of the segment (stored in a left and right structure in the interval tree case) is stored recursively in the subtrees rooted at children v_{i-1} and v_{j+1} of v.

As in the binary segment tree case, a point location query q on Π can be answered by answering q in the $S(v)$ structures on the $O(\log n/\log\log n)$ nodes on a root-leaf path of T. As Baumgarten et al. [1], we speed up queries using a dynamic FC method. That is, we sample segments from $S(v)$ and store them in augmented sets $AS(u)$ of descendants u of v (so that a segment $s \in AS(u) \setminus S(u)$ satisfies *proj*(s) $\supseteq$ *range*(u)). In later sections we will utilize several different implementations of $AS(v)$. However, in each case we will obtain a query bound of $O(\log\log n)$ for $AS(v)$, and thus a total query bound of $O(\log n)$. For each implementation, a key property of the sampling is that there is at most a poly-logarithmic number of segments in $AS(parent(v))$ spanning *range*(v) between two successive segments ("bridges") sampled from ancestors of v and stored in $AS(v)$ and $AS(parent(v))$. (Recall that all samples stored at v are totally ordered in *range*(v) so this property is well-defined.) Furthermore, we design our sampling so that we can avoid phantom segments, that is, so that the number of occurrences of a segment as a sample (a non-proper segment) is proportional to the number of real occurrences of the segment; this allows us to remove all occurrences of a segment when it is deleted. In summary, our segment tree sampling scheme fulfills the following key properties:

Property 3.1

(a) The number of segments in $AS(parent(v))$ spanning range(v) between two successive sample segments connecting $AS(v)$ and $AS(parent(v))$ is $O(\text{polylog}(n))$.

(b) For any segment s, the number of occurrences of s as a non-proper segment is proportional to the number of occurrences of s as a proper segment.

Below we sketch how we perform the sampling such that Property 3.1 is fulfilled. For simplicity, we first assume that $S(v)$ is simply implemented as a sorted list $L(v_i)$ for each child v_i of v, containing all segments in $S(v)$ that span *range*(v_i); note that such an implementation would use an excessive amount of space. Assume furthermore that for each node v we have constructed a list $\Delta(v)$ of sample segments from $L(v)$ such that the following are satisfied:

(i) For any two children v_i and v_j of v, for $i \neq j$, $\Delta(v_i) \cap \Delta(v_j) = \emptyset$.

(ii) The distance in $L(v)$ of any two successive samples from $\Delta(v)$ is $O(\text{polylog}(n))$.

Using the $L(v)$ and $\Delta(v)$ lists we can construct a set of augmented lists $AL(v) \supseteq L(v)$ fulfilling Property 3.1: For each node v in T we independently distribute the samples from $\Delta(v)$ to the AL-lists of the nodes in the subtree T_v rooted at v as follows. We partition $\Delta(v)$ into blocks of size $\Theta(\log n)$. For each block, we select *height*(v) (the height of T_v) samples and send each sample to a different level of T_v. Let $\Delta_i(v)$ be the list of samples sent to the level of depth i. Each level receives roughly the same number of samples, i.e., $\Theta(|\Delta(v)|/\log n)$. We distribute these samples among the f^i nodes of level i as follows: First, we partition $\Delta_i(v)$ into blocks of size $\Theta(f^i)$. Then, each node at level i receives one sample from each block of $\Delta_i(v)$. A sample s which was allocated to a node u is inserted in $AL(u)$ and $AL($*parent*$(u))$ creating a bridge between the two lists. It is relatively easy to see that this distribution scheme satisfies Property 3.1. Details will appear in the full paper.

What remains is to show how to perform the sampling without assuming that $S(v)$ is implemented with $L(v_i)$ lists, that is, to produce the $\Delta(v)$ samples such that assumptions (i) and (ii) are fulfilled. To do so, we assume that for each node v we maintain a list of the segments in each of the $O(f^2)$ slab intervals $I(v,i,j)$. A segment belongs to only one slab interval, so the total space required for these lists is linear and they can easily be maintained. We sample each $I(v,i,j)$ independently, by forming blocks of size $\Theta(f)$ and from each block we send a different sample to each $\Delta(v_k)$, $i \le k \le j$. This satisfies (i). It is easy to verify that (ii) also holds, since the range of v_k is crossed by $O(f^2)$ slab intervals of v. Thus, between any two consecutive samples in $\Delta(v_k)$, there are at most $O(f^2 \log n)$ segments in $L(v_k)$.

Finally, rather than augmenting $L(v)$ lists (which are too large to actually materialize) to obtain $AL(v)$ lists, in our application of the above sampling technique in §4 and §5 we augment the $S(v)$ structures directly to obtain augmented $AS(v)$ structures. In §4 the (main) $S(v)$ structure will be implemented using a persistent tree and in §5 using a segment tree; in both cases it is easy to insert the sampled segments to obtain $AS(v)$.

4 Middle segments

In this section we describe the structures $\mathcal{M}(v)$ associated with each node v in the base tree $\mathcal{T}$ of our point location structure, and how they can be used to efficiently answer queries on the segments Π^0.

4.1 Simple $O(n \log^\varepsilon n)$-space structure

Recall that $\mathcal{M}(v)$ stores a set of nonintersecting "middle" segments with endpoints located on the $f+1$ slab boundaries of v. One simple way of implementing $\mathcal{M}(v)$ is to maintain a sorted list (search tree) for each of the f slabs in v; each segment is then broken into at most f pieces corresponding to the slabs it spans, and stored in the relevant of these lists. This way a $\mathcal{M}(v)$ storing n segments uses $O(nf)$ space. However, queries on a $\mathcal{M}(v)$ structure can now easily be performed in $O(\log f + \log n) = O(\log n)$ time and updates in $O(f \log n)$ time.

We can speed up insertions using an approach similar to an approach used in [1]: Suppose we have already inserted s in slab i and now want to insert it in slab $i+1$. First we locate the two closest slab i segments, s_1 above s and s_2 below s, that also occur in slab $i+1$. By maintaining a pointer between the same occurrence of a segment in different slab lists, we then find s_1 and s_2 in the list for slab $i+1$; s must be inserted between these two segments, and we find the correct position by simultaneous searching downwards from s_1 and upwards from s_2. This can be done efficiently if the list is implemented using a finger search tree [4]. To actually find s_1 and s_2 in slab i we utilize a USF structure with query bound $O(\log\log n)$ [17]. Details will appear in the full paper. In the end we obtain an $O(f \log\log n + \log n)$ amortized insertion bound and $O(f \log n)$ amortized deletion bound. In the incremental case, we can use a simpler and faster method based on finger search trees for locating s_1 and s_2, resulting in an $O(f + \log n)$ amortized insertion time.

4.2 Linear-space structure

To obtain a linear space structure we will implement $\mathcal{M}(v)$ using two structures such that each of the middle segments in v is stored in one of the two structures: A *new segment structure* storing the $O(n/f)$ most recently inserted segments and an *old segment structure* storing the rest. The old segment structure will only support deletions, while the new segment structure supports both insertions and deletions. Queries are performed by simply querying the two structures separately; similarly, deletions are just performed on the relevant of the two structures. Insertions are performed on the new segment structure and after every $\Theta(n/f)$ updates the whole structure is rebuild such that all segments are stored in the old segment structure.

New segment structure. The new segment structure is simply the structure described in §4.1. Thus

it achieves $O(\log n)$ query bound and, since it stores $O(n/f)$ segments, uses $O(n)$ space. It can be updated in the time bounds stated at the end of §4.1.

Old segment structure. The old segment structure is basically the static point location data structure of Sarnak and Tarjan [24], which achieves $O(\log n)$ query time, using $O(n)$ space and $O(n \log n)$ preprocessing. Their structure is constructed by visiting the endpoints of the segments in increasing x-coordinate, inserting a segment s in a partially-persistent search tree when its left endpoint is visited and deleting it when its right endpoint is visited. To support deletions in the old segment structure we utilize a fully-persistent search tree [11] and modify the Sarnak-Tarjan method slightly so that if the current endpoint has a different x-coordinate from the last visited endpoint, then a new persistent version of the search tree is created; otherwise the update modifies the current version of the tree. Since we have $f + 1$ possible x-coordinates, the persistent tree only has $f + 1$ versions.

To delete a segment s from the old segment structure, we have to remove s from all versions of the persistent tree. To do so efficiently we utilize that the persistent structure has only $f + 1$ versions and that any version of a fully-persistent search tree can be updated efficiently. This way we can remove the at most f copies of a segment in $O(f)$ time amortized. Since this deletion increases the space by $O(f)$, we can afford to delete $O(n/f)$ segments. After $O(n/f)$ operations, we rebuild the structure. Details will appear in the full paper.

In summary, the old segment structure uses $O(n)$ space, supports queries in $O(\log n)$ time and deletions in $O(f)$ time amortized.

Rebuilding. We rebuild $\mathcal{M}(v)$ after $\Theta(n/f)$ updates so that all segments are stored in the old structure. We can do so in $O(n \log n)$ time, basically by running the Sarnak-Tarjan algorithm, which spends $O(\log n)$ time on each of the $O(n)$ segment endpoints. This leads to an additional $O(f \log n)$ amortized term in the update bound.

If we knew for each segment s the segment immediately above its left endpoint, the construction time could be reduced to $O(n)$ since we would then know the position in the persistent tree to insert s when we visit its left endpoint, in which case the actual insertion can be done in $O(1)$ time amortized. This would in turn reduce the additional update term to $O(f)$.

The segments above each left endpoint can in fact be found in $O(f + \log n)$ amortized time. To describe how, we let $\mathcal{P}'(v)$ denote the current set of segments in $\mathcal{M}(v)$ and $\mathcal{P}(v)$ the set of segments stored at $\mathcal{M}(v)$ the last time it was rebuild. Thus, $\mathcal{P}'(v) \setminus \mathcal{P}(v)$ is the set of segments added since the last rebuild (stored in the new structure). For each of the $O(n/f)$ segments in $\mathcal{P}'(v) \setminus \mathcal{P}(v)$, we simply perform a point location query for its left endpoint on both the new and the old segment structures to find the segment above the endpoint. Since each query takes $O(\log n)$ time, this is done in $O(n \log n/f)$ time in total, or $O(\log n)$ amortized per update. For each of the segments in $\mathcal{P}(v)$, the segment in $\mathcal{P}(v)$ above its left endpoint is already available in the persistent tree, so we only need to find the segment in $\mathcal{P}'(v) \setminus \mathcal{P}(v)$ above its left endpoint. We cannot afford to perform a point location query for each left endpoint, since that will take $O(n \log n)$ time. Instead we simply maintain a sorted list $L_i(v)$ for each slab boundary $\ell_i(v)$, $0 \leq i < f$, containing segments from $\mathcal{P}(v)$ with left endpoint on $\ell_i(v)$. To construct the new list for $\mathcal{P}'(v)$, while at the same time finding the segment in $\mathcal{P}'(v) \setminus \mathcal{P}(v)$ above each segment in $\mathcal{P}(v)$, we simply visit each slab boundary one at a time and merge $L_i(v)$ with the sorted list of segments in $\mathcal{P}'(v) \setminus \mathcal{P}(v)$ that span slab $i + 1$ (obtained from the new segment structure). Since the total size of all the $L_i(v)$ lists is n, and since each of the segments in $\mathcal{P}'(v) \setminus \mathcal{P}(v)$ is processed at most f times, the merges take $O(n + f \cdot n/f)$ time in total, or $O(f)$ amortized per update.

Lemma 4.1 *The middle structure storing n segments can be implemented on a pointer machine such that is uses $O(n)$ space and answers queries in $O(\log n)$ worst-case time. It supports insertions in $O(f \log\log n + \log n)$ and deletions in $O(f \log n)$ amortized time. In the insertion-only case, insertions are supported in $O(\log n)$ amortized time.*

4.3 Fractional cascading

To answer a point location query on Π^0 we traverse a root-leaf path in the base tree $\mathcal{T}$ and query $\mathcal{M}(v)$ in each visited node v. Thus, a direct use of the structure of §4.2 would result in an overall $O(\log^2 n/\log\log n)$ query bound.

Since the middle segments have a segment-tree-like allocation (the endpoints lie on an slab boundary of the nodes that stores the segment), we can reduce the search time of each $\mathcal{M}(v)$ to $O(\log\log n)$ using the FC method of §3.3; the amortized update bounds are not affected in this case. To make this method work we need to use fully-persistent finger search trees [11] in the old segment structure to facilitate fast local search; we also need to distinguish between two types of bridges to take care of the different representations

for $\mathcal{P}(v)$ and $\mathcal{P}'(v) \setminus \mathcal{P}(v)$. Details will appear in the full paper.

Lemma 4.2 *The set Π^0 of middle segments can be maintained in the middle structures such that point location queries can be performed in $O(\log n)$ worst-case time. Insertions can be performed in $O(\log^{\varepsilon} n \cdot \log\log n + \log n)$ amortized time and deletions in $O(\log^{1+\varepsilon} n)$ amortized time. In the insertion-only case, insertions can be performed in $O(\log n)$ amortized time. All bounds are valid on a pointer machine.*

5 Right segments

In this section we describe the right segment structures $\mathcal{R}(v)$ associated with each node v in the base tree $\mathcal{T}$ of our point location structure, and how they can be used to efficiently answer queries on the segments Π^+.

Recall that $\mathcal{R}(v)$ stores a set of nonintersecting "left" segments that all have left endpoint on the left boundary *left*(v) of the range associated with v; we let $N(v)$ denote this set of segments sorted by y-coordinate of their left endpoint. A point location query on Π^+ can be answered by traversing a root-leaf path in $\mathcal{T}$ while querying $\mathcal{R}(v)$ in each encountered node v. Cheng and Janardan [6] showed how $\mathcal{R}(v)$ can be implemented in the pointer machine model using a priority search tree such that a query can be answered in $O(\log n)$ time; using this structure directly would then result in a $O(\log^2 n/\log\log n)$ total query bound. Baumgarten et al. [1] used a FC method to reduce the $\mathcal{R}(v)$ search cost to $O(\log\log n)$, obtaining an $O(\log n \cdot \log\log n)$ overall query bound. We will use a similar method, modified to deal with the fact that our base tree has large fan-out; due to space constraint, we will skip many details in the description of the method.

5.1 $\mathcal{R}(v)$ structure

We implement the $\mathcal{R}(v)$ structure of node v by partitioning $N(v)$ into blocks of size $\beta = O(\log^2 n)$ and constructing a Cheng-Janardan priority search tree structure on each block. We also maintain two other sorted lists $AN(v)$ and $S(v)$ of segments defined as follows: In each block we identify the segment with the rightmost endpoint; we call this the *winner* of the block. Let $W(v)$ be the set of winners from $N(v)$. We store a copy of each segment $s \in W(v)$ in the list $AN(u)$ of each descendant u of v on the search path for the right endpoint of s. For each segment $s \in W(v) \cup AN(v)$ we then store the longest subsegment of s with right endpoint on a slab boundary of v in $S(v)$, that is, if the right endpoint of s is in the jth slab of v then the part of s spanning the first $j-1$ slabs is stored in $S(v)$. Refer to Figure 2. Note that the total size of all $AN(v)$ and $S(v)$ lists is $O(\frac{n}{\beta}\frac{\log n}{\log\log n}) = O(n/(\log n \log\log n))$. Finally, apart from the list itself, we also store the segments in $S(v)$ in a data structure $\mathcal{S}(v)$ described in §5.2 below.

To answer a point location query q, we use the $\mathcal{S}(v)$ structures to find, for each node u on the search path in $\mathcal{T}$ for q, the two segments in $S(u)$ immediately above and below q. In §5.2 we will sketch how the propagation of segments through the $AN(v)$ lists allows us to use FC to do so in $O(\log n)$ time. In the full version of this paper we show how we can then in $O(\log n)$ time identify a constant number of blocks of $N(u)$ to search, in each of the $O(\log n/\log\log n)$ nodes on the search path, in order to answer the query in $\mathcal{T}$. Since each such search requires only $O(\log\beta) = O(\log\log n)$ time, we answer the query in $O(\log n)$ time in total.

When updating Π^+ (inserting or deleting a segment), it is easy to find the relevant node v and update the Cheng-Janardan priority search tree structure in $O(\log n)$ time. In the full version of this paper, we show how such an update can at most result in the need for a constant number of updates on the $AN(u)$ and $S(u)$ lists in each of the $O(\log n/\log\log n)$ nodes u on a path from v to a leaf of $\mathcal{T}$. Using a number of ideas similar to the ones used in [1], we also show how all of these updates can be performed in $O(\log n)$ and $O(\log^2 n/\log\log n)$ amortized time for insertion and deletion, respectively. Finally, the $\mathcal{S}(u)$ structures for the nodes u on the path from v to a leaf also need to be updated; we describe how to do so in §5.2 below. We note that the update bounds of the $\mathcal{S}(u)$ structures determine the update bounds of our overall structure.

5.2 $\mathcal{S}(v)$ structure

What remains is to show how to implement the $\mathcal{S}(v)$ structures in each node v of $\mathcal{T}$ such that 1) the segments in $S(u)$ immediately above and below a query point q can be found in each node u on a root-leaf path in $O(\log n)$ time, and 2) a segment can be inserted in or deleted from the $\mathcal{S}(u)$ structure of each node u on a root-leaf path efficiently.

Since the segments in $S(v)$ have both endpoints on slab boundaries it is natural to use the middle segment structure $\mathcal{M}(v)$ described in §4 to implement $\mathcal{S}(v)$ (the structure can easily be modified to find both the segment above and below a query point); in fact, the simple $O(n\log^{\varepsilon} n)$ space structure of §4.1 suffices since the total size of all $S(v)$ lists is $o(n/\log n)$.

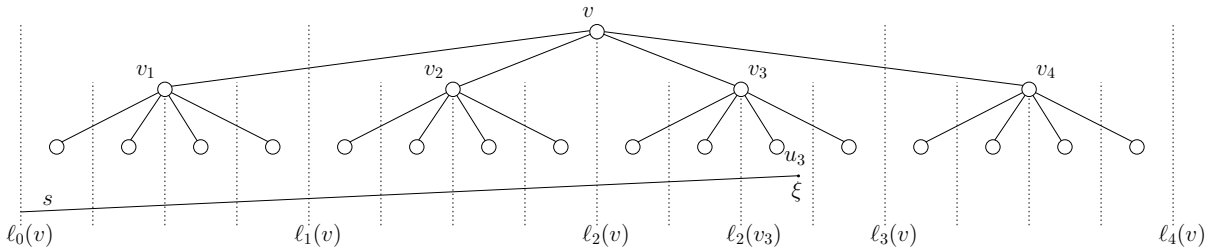

Figure 2. Segment s is a winner in $N(v)$. The part of s from $\ell_0(v)$ to $\ell_2(v)$ is stored in $S(v)$. The remaining part is copied to $AN(v_3)$. We continue this process for all nodes on the search path for ξ; at v_3, we store the part of s from $\ell_0(v_3)$ to $\ell_2(v_3)$ in $S(v_3)$, and send the piece after $\ell_2(v_3)$ to $AN(u_3)$.

Lemma 5.1 *A $\mathcal{S}(v)$ structure storing m segments can be implemented on a pointer machine such that it uses $O(m \log^{\varepsilon} m)$ space and answers queries in $O(\log f + \log m)$ worst-case time. Insertions can be performed in $O(f \log\log m + \log n)$ amortized time and deletions in $O(f \log m)$ amortized time. In the insertion-only case, insertions can be performed in $O(f + \log n)$ amortized time.*

In §5.2.3 we sketch how by using FC on this implementation of $\mathcal{S}(v)$ we can fulfill both 1) and 2) above. This leads to our best fully-dynamic pointer machine model point location structure. (The rest of the $\mathcal{R}(v)$ structure is also implementable on a pointer machine.) On the RAM however, we can utilize a segment tree idea, along with the fact that the left endpoints of all segments in $S(v)$ lie on the same vertical line, to obtain a better structure. We describe this implementation of $\mathcal{S}(v)$ in §5.2.2. Furthermore, in the insertion-only case we can develop simpler and more efficient structures both in the RAM and pointer machine models. We describe this structure in §5.2.1.

5.2.1 Improved insertion-only structure

To obtain the improved insertion-only structure we simply implement $\mathcal{S}(v)$ as a (binary) segment tree T. Since the segments in $S(v)$ only have $f+1$ distinct endpoint x-coordinates, the segment tree has height $O(\log f) = O(\log\log n)$ and uses $O(m \log f) = O(m \log\log n)$ space to store m segments.

To answer a query we search the segment lists of the $O(\log f)$ nodes on a root-leaf path in T. A straightforward implementation leads to an $O(\log f \log m)$ query bound; using FC and an efficient solution to the split-find with insertions problem [15], this can be improved to $O(\log m + \log f)$ in the RAM model and $O(\log m + \log f \log^* m)$ in the pointer machine model. Details will appear in the full paper.

To insert a segment s we need to insert s in the segment list of $O(\log f)$ nodes in T. A straightforward implementation leads to an $O(\log f \log m)$ bound; use of FC, as in [1], improves this to $O(\log m + \log f \log\log f)$. The reason the use of FC does not lead to an $O(\log m + \log f)$ insertion bound is that the segment tree gives rise to a FC instance on a directed graph with vertices of constant out-degree but in-degree $O(\log f)$: In a standard segment tree T the sequence of nodes where a given segment is stored consists of two subsequences; in one the heights are monotonically increasing and in the other they are monotonically decreasing. (In the structure for Π^+ we only have the second subsequence.) An insertion of a segment using FC takes place in two stages, one for each subsequence; in both stages we proceed bottom-up and move from the current node to the next node of the subsequence by locating appropriate bridges. In terms of FC, the problem is that while each node has a unique successor, it has $O(\log f)$ predecessors in all possible subsequences. To improve the insertion bound we obtain using FC, we change the way segments are assigned to nodes of T so that each node only has a constant number of possible predecessors and successors, that is, we obtain a FC graph with both constant out- and in-degree: Consider the decreasing subsequence. We start the insertion of a segment s at the appropriate leaf and move upwards. Let v be the current node, and let w be the node to the left of v on the same level as v. (These two nodes share a slab boundary.) The subsequence continues iff $\mathit{proj}(s) \supseteq \mathit{range}(w)$. In this case, if $\mathit{parent}(w)$ is not the parent of v and s spans $\mathit{range}(\mathit{parent}(w))$, we store s in $\mathit{parent}(w)$. Otherwise we store it in w. This way a segment is stored in at most two nodes of the same level, so the space use of the segment tree at most doubles; it is also easy to see that the query algorithm remains correct. But now the possible successors of v are w and $\mathit{parent}(w)$; conversely, the possible predecessors of w are v and the left child of v. This way the use of FC on the modified segment tree results in an $O(\log m + \log f)$ insertion bound. Details will appear in the full paper.

Lemma 5.2 *A $\mathcal{S}(v)$ structure storing m segments can be implemented such that it uses $O(m \log f)$ space and answers queries in $O(\log m + \log f)$ worst-case on a RAM and $O(\log m + \log f \log^* m)$ worst-case on a pointer machine. Insertions can be performed in $O(\log m + \log f)$ amortized time on a pointer machine.*

5.2.2 Improved RAM structure

As in the insertion-only structure, our improved RAM model $\mathcal{S}(v)$ structure is also based on a segment tree T. However, rather than a binary segment tree we use a high fan-out segment tree T. We also utilize two other main results, namely a range reporting and a list labeling result, which we describe below.

Dynamic 1d range reporting. In dynamic one-dimensional range reporting (RR) we maintain a set of integers Σ from a universe of size 2^w, under insertions and deletions, and support queries of the form *findany*(a, b). This query returns *any* element $x \in \Sigma$ such that $a \leq x \leq b$, or reports that $\Sigma \cap [a, b]$ is empty. Recently, Mortensen, Pagh and Pătraşcu [20] developed a linear-space data structure that achieves $O(\log \log w)$ time for *findany* and $O(\log w)$ for updates.

Segment labeling. We use a list labeling data structure to assign integer labels to segments in $S(v)$; using the linear-space structure of Willard [25] each of these labels has $O(n)$ size ($O(\log n)$ bits). Then we can use an array of size $O(n)$ for the reverse mapping of labels to segments; given a label we can find the corresponding segment, if it exists, in $O(1)$ time.

Each update in Willard's structure takes $O(\log^2 n)$ worst-case time and changes $O(\log^2 n)$ labels. This bound is not good enough for our purpose, so we do not label each segment individually. Instead, we amortize the cost of labeling by partitioning $S(v)$ into blocks of size $\gamma = \Theta(\log^3 n)$, and label the blocks. To facilitate search, we represent each block by its winner (defined as in §5.1). As we insert and delete segments in $S(v)$, a block representative may change with each update. The block labeling, however, remains the same for every interval of $\Theta(\log^3 n)$ operations. After that many updates, $O(\log^2 n)$ representatives receive a new label.

Structure. We use a segment tree T with fan-out $f' = \log^{\varepsilon} \log n$ to store the representatives for each block of $S(v)$. The height of T is $O(\log f / \log \log f)$ so we can spend $O(\log \log f) = O(\log \log \log n)$ time per level for a query. This bound matches the query time of the RR structure, since $w = O(\log n)$. Each node u of T stores segments that span at most f' slabs of v. We maintain the segments allocated to u using the simple structure of §4.1. Hence, for each slab $[\ell_{i-1}(u), \ell_i(u)]$ of u, $1 \leq i \leq f'$, we maintain a finger search tree which stores the representatives allocated to u that span $[\ell_{i-1}(u), \ell_i(u)]$; we also keep the labels of these representatives in an RR structure for $[\ell_{i-1}(u), \ell_i(u)]$. Finally, we apply the FC scheme of §3.3 on T, but using block size $\beta' = \Theta(\log^2 \log n)$.

To perform a query on $S(v)$, we start at the appropriate leaf of T and move towards the root. At each step we maintain the two best block representatives found so far, one immediately above and the other immediately below the query point q. Let a and b be the integers corresponding to those segments. In order to search the appropriate slab list of the parent node, we perform a *findany*(a, b) query on that list. Consider the two possible outcomes. First suppose *findany* returns a segment c of the parent list. Without loss of generality assume c is below q. Then the two new best representatives are in the interval $[a, c]$, which contains a number of representatives polynomial in β'. Hence the new best representatives can be found in $O(\log \log \log n)$ time, as desired, using the finger search tree. Next suppose that *findany* returns null. Clearly, in this case a and b remain the best representatives encountered so far. At the end of this search we are left with two block representatives a and b. To find the segments in $S(v)$ immediately above and below q it suffices to search the blocks of a and b; this search takes $O(\log \gamma) = O(\log \log n)$ time. Hence, excluding the time spend on the leaf of T, a query is answered in $O(\log \log n + \log \log \log n \frac{\log f}{\log \log f}) = O(\log \log n + \log f) = O(\log f)$ time.

Now we consider updates in $S(v)$. When the representative of a block changes, the new representative is inserted into the segment tree and the old is deleted. An insertion or deletion of a representative in $S(v)$ may modify $O(\log^{1+\varepsilon} f)$ segment lists of nodes in T. Given the location of a representative in $S(v)$ and its label, we can update T in $O(\log^{2+\varepsilon} f)$ amortized and randomized time. Details will appear in the full paper.

Lemma 5.3 *A $\mathcal{S}(v)$ structure storing m segments can be implemented on a RAM such that it uses $O(m \log^{1+\varepsilon} f)$ space and answers queries in $O(\log m + \log f)$ randomized time. Insertions can be performed in $O(\log m + \log^{2+\varepsilon} f)$ amortized and randomized time and deletions in $O(\log^{2+\varepsilon} f)$ amortized and randomized time.*

5.2.3 Fractional cascading

Recall that to answer a point location query q on Π^+ (the $\mathcal{R}(v)$ structures) efficiently, we need to query the

$\mathcal{S}(v)$ structures in the $O(\log n/\log\log n)$ nodes on a search path in $\mathcal{T}$. Regardless of which of our three implementations of $\mathcal{S}(v)$ we use, simply querying the $\mathcal{S}(v)$ structures individually will not give us the desired $O(\log n)$ bound. To obtain this bound we use FC on the $\mathcal{S}(v)$ structures (that is, segment trees): To be able to efficiently move from the root of the $\mathcal{S}(v_i)$ structure in v_i to the appropriate leaf of the $\mathcal{S}(v)$ structure in the parent v of v_i, we adapt the method of §3.3. We also speed up insertions using a FC method similar to the one used in §4.1. Details will appear in the full paper.

Lemma 5.4 *The insertion-only $\mathcal{S}(v)$ structures can be implemented so that their total size in $\mathcal{T}$ is $O(n)$. All the $\mathcal{S}(v)$ structures on a search path in $\mathcal{T}$ can be queried in t_{q} worst-case time and insertions can be performed in t_{ins} amortized time, where:*

- $t_{\mathrm{q}} = O(\log n)$ *and* $t_{\mathrm{ins}} = O(\log^{1+\varepsilon} n)$, *or*
- $t_{\mathrm{q}} = O(\log n \log^{*} n)$ *and* $t_{\mathrm{ins}} = O(\log n)$ *on a pointer machine;*
- $t_{\mathrm{q}} = O(\log n)$ *and* $t_{\mathrm{ins}} = O(\log n)$ *on a RAM.*

Lemma 5.5 *The $\mathcal{S}(v)$ structures can be implemented so that their total size in $\mathcal{T}$ is $O(n)$. All the $\mathcal{S}(v)$ structures on a search path in $\mathcal{T}$ can be queried in t_{q} worst-case time, and insertions and deletions are performed in t_{ins} and t_{del} amortized time, respectively, where:*

- $t_{\mathrm{q}} = O(\log n)$, $t_{\mathrm{ins}} = O(\log^{1+\varepsilon} n)$ *and* $t_{\mathrm{del}} = O(\log^{2+\varepsilon} n)$ *on a pointer machine;*
- $t_{\mathrm{q}} = O(\log n)$, $t_{\mathrm{ins}} = O(\log n \log^{1+\varepsilon}\log n)$ *and* $t_{\mathrm{del}} = O(\log^{2} n/\log\log n)$, *all randomized, on a RAM.*

Since the structure for Π^{+} is the bottleneck of our overall point location structure, the above lemmas imply the bounds shown in Table 3.

References

[1] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17(3):342–380, 1994.

[2] J. Bentley. A solution to Klee's rectangle problems. unpublished report, 1977.

[3] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[4] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[5] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structure technique. *Algorithmica*, 1(2):133–62, 1986.

[6] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM Journal on Computing*, 21(5), 1992.

[7] Y.-J. Chiang, F. P. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. *SIAM Journal on Computing*, 25(1):207–233, 1996.

[8] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. Technical Report CS-91-24, Brown University, March 1991.

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

[10] M. de Berg, M. van Krevald, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, second edition, 2000.

[11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[12] H. Edelsbrunner. Dynamic data structure for orthogonal intersection queries. Technical Report Technical Report F59, Inst. Informationsverarb. Tech. Univ. Graz, Graz, Austria, 1980.

[13] O. Fries. *Suchen in dynamischen planaren Unterteilungen*. PhD thesis, Universität des Saarlandes, 1990.

[14] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. *SIAM Journal on Computing*, 28(2):612–36, 1998.

[15] H. Imai and T. Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8(1):1–18, 1987.

[16] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[17] K. Mehlhorn. *Data structures and algorithms 1: sorting and searching*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[18] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–41, 1990.

[19] K. Mehlhorn, S. Näher, and H. Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(6):1093–1102, 1988.

[20] C. W. Mortensen, R. Pagh, and M. Pătraşcu. On dynamic range reporting in one dimension. In *Proc. 37th ACM Symp. on Theory of Computing*, pages 104–111, 2005.

[21] M. H. Overmars. Range searching in a set of line segments. In *Proc. 1st ACM Symp. on Computational Geometry*, pages 177–185, 1985.

[22] F. P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM Journal on Computing*, 18(4):811–830, 1989.

[23] F. P. Preparata and R. Tamassia. Dynamic planar point location with optimal query time. *Theoretical Computer Science*, 74(1):95–114, 1990.

[24] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

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