SERIE B — INFORMATIK

Dynamic Point Location in General Subdivisions

Hanna Baumgarten*
Hermann Jung**
Kurt Mehlhorn***

B 92–10
March 1992

Abstract

The *dynamic planar point location problem* is the task of maintaining a dynamic set S of n non-intersecting, except possibly at endpoints, line segments in the plane under the following operations:

- Locate(q: point): Report the segment immediately above q, i.e., the first segment intersected by an upward vertical ray starting at q;
- Insert(s: segment): Add segment s to the collection S of segments;
- Delete(s: segment): Remove segment s from the collection S of segments.

We present a solution which requires space $O(n)$, has query and insertion time $O(\log n \log\log n)$ and deletion time $O(\log^2 n)$. A query time below $O(\log^2 n)$ was previously only known for monotone subdivisions and horizontal segments and required non-linear space.

*Graduiertenkolleg Algorithmische Diskrete Mathematik, Institut für Informatik, Freie Universität Berlin, Arnimallee 2-6, W 1000 Berlin 33, Germany, This work is supported by the Deutsche Forschungsgemeinschaft under grant WE 1265/2-1,
**Fachbereich Informatik, Humboldt-Universität Berlin, PSF 1297, O 1086 Berlin, Germany,
***Max-Planck-Institut für Informatik and Fachbereich Informatik der Universität des Saarlandes, W 6600 Saarbrücken, Germany, This work is supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM)

Subdivision	**Space**	**Locate**	**Insert**	**Delete**	**Reference**
horizontal segments	$n \log n$	$\log n \log\log n$	$\log n \log\log n$	$\log n \log\log n$	Mehlhorn-Näher [?]
monotone	n	$\log^2 n$	$\log^2 n$	$\log^2 n$	Preparata-Tamassia [?]
monotone	$n \log n$	$\log n$	$\log^2 n$	$\log^2 n$	Chiang-Tamassia [?]
monotone	n	$\log^2 n$	$\log n$	$\log n$	Goodrich-Tamassia [?]
connected	n	$\log^2 n$	$\log^2 n$	-	Fries-Mehlhorn [?]
connected	n	$\log^2 n$	$\log^4 n$	$\log^4 n$	Fries [?]
general	$n \log n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	Bentley [?]
general	n	$\log^2 n$	$\log n$	$\log n$	Cheng-Janardan [?]
general	$n \log n$	$\log n \log\log n$	$\log n \log\log n$	$\log^2 n$	this paper, section 2
general	n	$\log n \log\log n$	$\log n \log\log n$	$\log^2 n$	this paper, section 3

Figure 1: Running Times of Dynamic Point Location Structures

1 Introduction

Planar point location is a fundamental geometric searching problem which has been extensively studied in recent years. In this paper we consider the problem of dynamically maintaining a set S of n non-intersecting (except possibly at endpoints) line segments in the plane under the following operations:

> Locate(q: point): Report the segment immediately above q, i.e., the first segment intersected by an upward vertical ray starting at q;
>
> Insert(s: segment): Add segment s to the collection S of segments;
>
> Delete(s: segment): Remove segment s from the collection S of segments.

Call a subdivision *connected* if the graph induced by the set of line segments S (the endpoints are the vertices, the segments are the edges) is connected. Connected subdivisions include the *monotone subdivisions* of [?] as a special case.

The *locate* operation in connected subdivisions is usually required to return the name of the region containing the query point (and not only the segment immediately above the query point) and some papers reserve the term dynamic planar point location problem for the searching problem in connected subdivisions. Overmars [?] has shown how to reduce the point location problem in connected subdivisions to the dynamic planar point location problem (as defined above) with only $O(\log n)$ additional cost per operation. On the other hand, there are applications of the dynamic point location problem, e.g. space sweep, where the connectedness assumption is unnatural.

Table 1 summarizes our results and compares them to previous work on deterministic point location structures. We achieve locate and insertion time $O(\log n \log\log n)$ and deletion time $O(\log^2 n)$, the bounds for insertions and deletions being amortized. The space bound is $O(n \log n)$ for our first solution (see section 2), and $O(n)$ for our second solution (see section 3). The second solution is a refinement of the first one. Previously, a query time below $O(\log^2 n)$ was only known for the special cases of monotone subdivisions [?] and horizontal line segments [?], respectively. In both cases non-linear space was needed. Our results leave open the question whether a solution with query and update time $O(\log n)$ is possible. Note however, that there are two randomized data structures ([?, ?]) which achieve expected logarithmic query and update time.

We now comment on the algorithmic concepts used in this paper. For the first solution we combine segment trees [?] and dynamic fractional cascading [?, ?], for the second solution we also

use interval trees [?, ?] and the interval–priority–search trees of [?].

Fractional cascading deals with the following situation: An undirected graph, in which a linear list is associated with each node, is given. The lists are all drawn from the same linearly ordered universe. A query selects a connected subgraph and locates the argument of the query in the linear list of every vertex of the subgraph. The fractional cascading technique augments the lists associated with the vertices by bridges. A bridge is an element which is stored in the lists of two adjacent vertices of the graph. The bridges between two lists divide the two lists into blocks and speed up the search as follows: Once a query element is located in the list and hence in a block of some vertex, the search in any neighbor list can be restricted to a block. Thus the time to locate an element in an augmented list (except in the first) is the time to find a closest bridge plus the time for the binary search in a block. In [?, ?] it was shown that block sizes can be kept bounded and that a closest bridge can be found in time $O(1)$; additional time $O(\log\log n)$ per vertex is needed to project the position of the query element in the augmented list of a vertex into its position in the original list.

In our first application of fractional cascading the underlying graph is a (segment-) tree for the set S of segments. A segment $s \in S$ is contained in the segment list $S(v)$ of a tree node v if s spans the range of v but does not span the range of v's parent. Fractional cascading as described above does not apply directly since there is no natural linear order on a set of non–intersecting line segments. Note that the "above"–order is only a partial order and that no linear extension of it is compatible with insertions and deletions. A solution to this problem was given by [?]. Bridges are created by copying elements in root to leaf direction; in this way the augmented segment list of every vertex is linearly ordered by the "above"–relation. As an unpleasant by–product the block size can only be controlled in the parent nodes but not in the children nodes. The way around this difficulty is to search bottom up.

Insertions and deletions create additional problems. For deletions the following problem arises. Fractional cascading copies elements from lists to neighboring lists to create bridges between lists. In this way an element may have copies in several lists. Suppose now that a segment with many copies is deleted from the collection S of segments. In standard dynamic fractional cascading ([?]), it is possible to leave the copies as *ghost elements* in the data structure. In the case of segments, the difficulty arises that ghost segments may intersect with segments inserted later. We allow intersections, but in a carefully controlled way (e.g., we guarantee that bridges never intersect and that no segment intersects more than one bridge), and show that slightly modified search techniques can cope with a small number of intersections.

An insertion of a segment s into a segment tree brings about the insertion of at most $2h$ (h = height of the segment tree) subsegments of s into the segment lists of nodes nearby the search path for the two endpoint of s (see Figure 3). An $O(\log^2 n)$ insertion time results if each subsegment is located by binary search. We show that previously inserted elements can guide the insertion process and that an amortized insertion time of $O(\log n \log\log n)$ can be obtained.

The second solution in addition uses interval trees [?, ?] and interval-priority-search trees [?]. The space requirement is reduced to $O(n)$, insertion, deletion and query time are unchanged, but the query algorithm becomes more complex. The reason is that the search algorithm in interval-priority-search trees is more complex than binary search and hence is more difficult to combine with fractional cascading. The details are given in section 3.

We assume some familiarity with interval and segment trees (cf. [?, Vol.1, sections VIII.5.1.1 and VIII.5.1.3]). Both kinds of trees store a set of non-intersecting line segments and are organized as augmented binary search trees. We use the following notation. For a node v of the search tree T, $key(v)$ denotes the x-coordinate which guides the search in node v, $L(v)$ denotes the vertical line defined by $x = key(v)$, $xrange(v)$ denotes the range of x-coordinates associated with v, $range(v)$ denotes the set of all points in $\mathbb{R}^2$ with x-coordinate in $xrange(v)$. For a segment s, $proj(s)$ denotes the projection of s to the x-axis. For a segment s and a node v, let $home(s) = v$ if $s \subseteq range(v)$ and s intersects $L(v)$. For nodes v and w, let $top(w) = v$ if w can be reached from v by going right once and then zero or more times to the left (see Figure 2). A segment s is called *elementary* if $proj(s) = xrange(v)$ for some node v of T.

In section 2 we discuss our first solution which is based on segment trees and fractional cascading. Section 3 is devoted to the combined interval-segment tree. Section 4 offers a short conclusion.

2 Segment Trees and Fractional Cascading

Let S be a set of pairwise non-intersecting (except at endpoints) non-vertical line segments and let T be a search tree for the x-coordinates of the endpoints of the segments in S. Let h be the height of T and let V be the set of nodes of T. We assume that $h = O(\log n)$. As usual, we associate with each node $v \in V$ a segment list as follows:

$$S(v) = \{s | s \in S, xrange(v) \subseteq proj(s) \text{ and } \neg(xrange(parent(v)) \subseteq proj(s))\}$$

A segment $s \in S$ contributes to the segment list $S(v)$ of at most $2h$ nodes, i.e., $N = \sum_{v \in V} |S(v)|$ is less than or equal to $2nh$. It is well known that segment trees achieve time $O(\log^2 n)$ for locate, insert and delete. In this section we adapt the fractional cascading method to our problem and improve the time bounds for locate and insert to $O(\log n \log\log n)$.

2.1 The Data Structure

The main idea is to use fractional cascading [**?**]. We associate an augmented segment list $AS(v) \supseteq S(v)$ with each node v of T. Every segment $s \in AS(v)$ satisfies $proj(s) \supseteq xrange(v)$ and is called *proper* if $s \in S(v)$ and *improper* otherwise. Two proper segments never intersect since S is a set of non-intersecting line segments. We maintain this property also for improper segments.

Invariant 1 *For all nodes v of T and distinct improper segments $s, s' \in AS(v)$ there is a segment $t \in S$ such that $xrange(v) \subseteq proj(t)$ and t separates s and s', i.e., t lies between s and s' in $range(v)$.*

Invariant 1 implies that no two improper segments in $AS(v)$ intersect and that a proper segment in $AS(v)$ intersects at most one improper segment in $AS(v)$. The lists $S(v)$, $AS(v) \setminus S(v)$, and $AS(v)$ are ordered according to the y-coordinates of the intersection of the segments with the line $L(v)$. In the case of a tie, the slope decides. Note that for lists $S(v)$ and $AS(v) \setminus S(v)$ this order coincides with the "above"-order for segments.

The list $AS(v)$ is partitioned in subsequences which we call *S-blocks* or simply *blocks*. For each block B we maintain a counter $pot_1(B)$, which counts the number of insertions into and deletions from B since the creation or last reorganization of B. A block B of $AS(v)$ is called *isolated* if it is the only block in $AS(v)$.

Invariant 2 *For all nodes v of T and all blocks B in $AS(v)$:*

(a) $pot_1(B) \leq a/4$,

(b) $|B| \leq 10a/4 + pot_1(B)$,

(c) if B is not isolated then $|B| \geq 5a/4 - pot_1(B)$,

where a is a parameter to be fixed later ($a = \Theta(\log^2 n)$).

Invariant 2 implies that $|B| \leq 11a/4$ always and that $|B| \geq a$ if B is not isolated. With each block B we associate a representative $rep(B)$ and store the representatives of non-isolated blocks of $AS(v)$ also in the lists $AS(w)$ for both children w of v. In this way, representatives serve as bridges (in the spirit of fractional cascading [**?**, **?**]) between neighboring lists and facilitate searching.

Invariant 3

(a) If B is not isolated in $AS(v)$ then $rep(B) \in AS(w)$ for both children w of v.

(b) For every improper segment $s \in AS(v)$ there is a non-isolated block B of $AS(parent(v))$ with $rep(B) = s$. The two occurrences of s are called a bridge and each occurrence of s is called a member of the bridge.

It is now easy to bound the total number of segments in T.

Lemma 1 *Let N be the number of elementary segments, i.e., $N = \sum_{v \in V} |S(v)| = O(nh)$. Then $\sum_{v \in V} |AS(v)| = O(N)$ and $\sum_{v \in V} |AS(v) \setminus S(v)| = O(N/a)$.*

Proof: Let $h(v)$ be the height of node v in T and let h be the height of the root. Obviously $|AS(root(T))| = |S(root(T)|$. Also the number of bridges between $AS(v)$ and $AS(w)$ for a child w of v is at most $|AS(v)|/a$ (by Invariant 3). We therefore have for $i < h$

$$\sum_{h(v)=i} |AS(v)| \leq \sum_{h(v)=i} |S(v)| + \frac{2}{a} \sum_{h(v)=i+1} |AS(v)|$$

and hence

$$\begin{aligned} \sum_{v \in V} |AS(v)| &= \sum_{i=0}^{h} \sum_{h(v)=i} |AS(v)| \\ &\leq \sum_{v \in V} |S(v)| + \sum_{i=0}^{h} \sum_{j=i+1}^{h} (2/a)^{j-i} \sum_{h(v)=j} |S(v)| \\ &< \sum_{v \in V} |S(v)| + \frac{4}{a} \sum_{v \in V} |S(v)| \end{aligned}$$

□

Blocks evolve over time. The representative $rep(B)$ of a block B is chosen by the following procedure.

procedure NEW_BRIDGE(B)

case 1: $|B \cap S(v)| \geq |B \cap (AS(v) \setminus S(v))|$, i.e., proper segments prevail. Choose as $rep(B)$ some segment s with $proj(s) = xrange(v)$ and the property that one half of the segments in $B \cap S(v)$ is strictly above s and one half is strictly below s.

case 2: $|(B \cap S(v))| < |B \cap (AS(v) \setminus S(v))|$, i.e., improper segments prevail. Choose as $rep(B)$ some segment s with $proj(s) = xrange(v)$ and the property that one half of the segments in $B \cap (AS(v) \setminus S(v))$ is strictly above s and one half is strictly below s.

end NEW_BRIDGE.

A representative is called *proper* if it is chosen in case 1 of NEW_BRIDGE and *improper* otherwise. Of course, the occurrences of a representative of a block of $AS(v)$ in the children of v are always improper elements of $AS(child(v))$. This implies that representatives of distinct blocks of $AS(v)$ do not intersect (by Invariant 1). We also need the following Invariant.

Invariant 4 *Let $AS(v)$ consist of more than one block, let B be a block of $AS(v)$, and let $t = rep(B)$. Then B contains segments s and s' above and s'' and s''' below t and of the same kind (proper or improper) as t.*

This completes the high-level description of the data structure. The low-level description follows.

- For each node we have several balanced search trees. The lists $S(v)$, $AS(v)$ and $AS(v) \setminus S(v)$ are stored in balanced trees with constant amortized insertion and deletion time for updates at specified positions; e.g., $(2,4)$-trees will do ([?, Vol.1, section III.5.2]). For each block B of $AS(v)$ we also have balanced trees for $B \cap S(v)$ and $B \cap (AS(v) \setminus S(v))$. In addition we keep each augmented node list $AS(v)$ in the dynamic Union-Find-Split-Add-Erase data structure (UFS-structure) of [?]. This data structure requires linear space and supports the following six operations on a list of k items, some of which are marked, in time $O(\log\log k)$ per operation: $Findabove(s)$ returns the closest marked predecessor of item s, $Findbelow(s)$ returns the closest marked successor of item s, $Split(s)$ marks s, $Union(s)$ unmarks s, $Add(s,s')$ adds the unmarked item s immediately after item s', and $Erase(s)$ removes item s. Initialization of the data structure for a list of k items takes linear time $O(k)$. In the UFS-structure on $AS(v)$ the marked items are the improper segments of $AS(v)$. We refer to this structure as the UFS-structure $\mathcal{I}$ on $AS(v)$. A second UFS-structure on $AS(v)$ will be defined in section 2.3.3.

- All occurrences of a segment are stored in a linear list which is ordered in correspondence to the in-order of the tree nodes.

- The tree T is organized as a BB[1/4]-tree for the x-coordinates of the endpoints of segments in S [?,?]. We assume w.l.o.g. that these x-coordinates are distinct (as in [?], we may replace the x-coordinate of an endpoint by a triple consisting of the x-coordinate, the y-coordinate of the point and the slope of the segment).

2.2 The Point Location Query

A point location query determines for a query point $q \in \mathbb{R}^2$ the segment in S immediately above q. A segment s in some set S' of segments is *above* q if a vertical upward ray starting at q intersects s and is *immediately above* q (or the segment in S' immediately above q) if s is the first segment in S' intersected by a vertical upward ray starting at q. The notions *below* and *immediately below* are defined analogously. A segment s is called a *best* segment in S' iff it is either immediately above or immediately below q.

The standard query algorithm in segment trees walks along the search path for the x-coordinate of q and determines in each node of this path the segment in $S(v)$ immediately above q with cost $O(\log n)$ per node. We determine for each node v of the path (in leaf to root fashion) a best improper segment and in some (but not necessarily all) nodes the proper segment immediately above q. Our fractional cascading approach improves the search time to $O(\log\log n)$ per node.

procedure LOCATE(q)

1. s = NIL and t = NIL
2. find the leaf v of T with $proj(q) \in xrange(v)$
3. **while** v is defined
4. **do** LOCATE(q, v); $v = parent(v)$ **od**

end LOCATE.

procedure LOCATE(q, v) (* with global variables s and t *)

case 1: t = NIL (* there is only one block in $AS(v)$ or v is a leaf *)

1. $s(v)$ = the segment in $S(v)$ immediately above q (NIL, if there is no such segment)
2. t = a best segment in $AS(v) \setminus S(v)$ if $AS(v) \setminus S(v) \neq 0$ and NIL otherwise

3. $s = best(s(v), s)$ (* *best* returns the lowest segment above q *)

case 2: otherwise

1. let B be the block in $AS(v)$ with $rep(B) = t$ and let A and C be the neighboring blocks above and below B.
2. $s(v) =$ the segment in $S(v) \cap (A \cup B \cup C)$ immediately above q (NIL, if there is no such segment)
3. **if** $(AS(v) \setminus S(v)) \cap (A \cup B \cup C) \neq \emptyset$
4. **then** $t =$ a best segment in $(AS(v) \setminus S(v)) \cap (A \cup B \cup C)$
5. **else** $t =$ a best segment in $\{Findabove(s(v)), Findbelow(s(v))\}$
6. **fi**
7. $s = best(s(v), s)$

end LOCATE.

Lemma 2 *Point location takes time* $O(\log n + h \log\log n)$.

Proof: Let q be the query point and x_q be the x-coordinate of q. The time bound follows from the observation that we need $O(h)$ time to find the leaf v with $x_q \in xrange(v)$, time $O(\log n)$ in leaf v and time $O(\log a + \log\log n)$ in every other node of the search path.

We next turn to the proof of correctness. We first make a useful observation:

Observation 3 *At most one improper segment from a block above A can intersect the representative of block A and at most one improper segment from a block below C can intersect the representative of block C.*

Proof: Assume for the sake of a contradiction that there are two improper elements p_1 and p_2 in blocks above A which intersect $rep(A)$. According to Invariant 4 A contains a segment s above $rep(A)$. Thus segments p_1 and p_2 intersect s. If s is improper this directly contradicts Invariant 1. If s is proper then let p be the proper segment which separates p_1 and p_2 (Invariant 1). s must intersect p, a contradiction. □

Let $s_0 \in S$ be the segment in S immediately above q. We prove inductively that the following two assertions hold after completion of call LOCATE(q, v).

(1) if $AS(v) \setminus S(v) \neq \emptyset$ then t is a best segment in $AS(v) \setminus S(v)$ and if $AS(v) \setminus S(v) = \emptyset$ then $t =$ NIL

(2) if $s_0 \in S(v)$ then $s(v) = s_0$

Assertions (1) and (2) are obviously true if v is a leaf or $AS(v)$ consists of a single block. So assume that v is an inner node and $AS(v)$ consists of more than one block. The induction hypothesis implies that q lies between the representatives of blocks A and C since $t = rep(B)$ is a best improper segment in a child of v and since representatives do not intersect (Invariant 1).

(1) We distinguish cases according to the **if**-statement in case 2 of LOCATE(q, v):

case 1: There is no improper segment in $A \cup B \cup C$. Then $s(v) \neq$ NIL after execution of line (2) of LOCATE(q, v) and $s(v)$ is the segment in $S(v)$ immediately above q. Thus either $Findbelow(s(v))$ or $Findabove(s(v))$ is a best segment in $AS(v) \setminus S(v)$ (by Observation 3).

case 2: There are improper segments in $A \cup B \cup C$. Since improper segments do not intersect (Invariant 1), a best improper segment in $A \cup B \cup C$ is clearly a best segment in $(AS(v) \setminus S(v)) \cap (A \cup B \cup C)$.

(2) Assume that $s_0 \in S(v)$. If $s_0 \in S(v) \cap (A \cup B \cup C)$ then clearly $s(v) = s_0$. So assume $s_0 \in S(v) \setminus (A \cup B \cup C)$. Then $A \cup B \cup C$ can contain no proper segment above q and s_0 lies either above or below B. In the former case block A must exist. Also $rep(A)$ lies above q and A contains two segments of the same kind as $rep(A)$ above $rep(A)$. These two segments are either proper or separated by a segment in S (by Invariant 1), a contradiction to the fact that $s_0 \in S(v)$. In the latter case block B must exist. Also $rep(B)$ lies below q and the symmetric argument leads to a contradiction.

□

Remark: Let s_v be the segment in $S(v)$ immediately above q. The query algorithm described above does not determine s_v in all nodes v of the search path. In section 3.3 it will be important that the segment s_v is determined for all nodes v of the search path. This can be done as follows. Maintain a second UFS-structure on $AS(v)$ for all v; this time the marked items are the proper segments in $AS(v)$. Use $Find_S_above$ to denote the $Find$ operation in this structure. Also add the following line 6.a to case 2 of procedure LOCATE.

if $s(v) =$ NIL
then $s(v) := Find_S_above(t)$
fi

We claim that $s(v) = s_v$ after execution of line 6.a. If $s_v \in A \cup B \cup C$ then $s(v) = s_v$ after line (2). So assume $s_v \notin A \cup B \cup C$. Then $A \cup B \cup C$ can contain no proper segment above q and s_v lies either above or below $A \cup B \cup C$. In the former case block A must exist. Also $rep(A)$ lies above q and A contains two segments of the same kind as $rep(A)$ and above $rep(A)$. These two segments are necessarily improper and hence are separated by a segment in S. Thus no proper segment in a block above A can be below q and hence $Find_S_above(t) = s_v$ in line (6.a). In the latter case, i.e., s_v would be stored in a block below C and hence $S(v) \cap (A \cup B \cup C) = \emptyset$. Also block C must exist, $rep(C)$ lies below q and C contains two improper segments below $rep(C)$. These segments are separated by a segment in S and hence s_v cannot lie in a block below C.

2.3 Insertions and Deletions

The dynamic behavior of our search tree depends on two components: the dynamic behavior of the underlying search tree and the dynamic behavior of the segment lists and augmented segment lists. We proceed in three steps. For the first two steps the search tree T and the parameter a are assumed to be fixed. In the first step we discuss insertions and deletions of segments using standard methods and arguments and verify the Invariants. In the second step we improve the insertion time by using additional information about the lists and in the last step we discuss how to rebalance T and how to adapt the parameter a.

2.3.1 A First Solution

We first give the insertion and deletion procedures for elementary segments and the procedure to reorganize blocks.

procedure ELEMENTARY_INSERT(s, v)

1. insert s into the appropriate search trees and identify the block B into which s has to be inserted
2. insert s into the search tree for B
3. update the UFS-structure $\mathcal{I}$, i.e., add s to $AS(v)$ and perform $Split(s)$ if s is an improper segment
4. $|B| = |B| + 1$; $pot_1(B) = pot_1(B) + 1$;

5. **if** $pot_1(B) \geq a/4$ **then** REORGANIZE(B).

end ELEMENTARY_INSERT.

procedure ELEMENTARY_DELETE(s, v)

1. delete s from the list $AS(v)$ and from the corresponding search tree
2. delete s from B and from the corresponding search tree of B
3. delete s from the UFS-structure $\mathcal{I}$, i.e., remove s from $AS(v)$ and perform $Union(s)$ if s is improper segment
4. $|B| = |B| - 1$; $pot_1(B) = pot_1(B) + 1$;
5. **if** $pot_1(B) \geq a/4$ **then** REORGANIZE(B).

end ELEMENTARY_DELETE.

procedure REORGANIZE(B)

case 1: $5a/4 \leq |B| \leq 10a/4$

1. ELEMENTARY_DELETE($rep(B), lchild(v)$); ELEMENTARY_DELETE($rep(B), rchild(v)$)
2. $rep(B)$ = NEW_BRIDGE(B)
3. $pot_1(B) = 0$
4. ELEMENTARY_INSERT($rep(B), lchild(v)$); ELEMENTARY_INSERT($rep(B), rchild(v)$)

case 2: $|B| > 10a/4$

1. ELEMENTARY_DELETE($rep(B), lchild(v)$); ELEMENTARY_DELETE($rep(B), rchild(v)$)
2. divide B into blocks B_1 and B_2 of approximately the same size
3. build up the internal structures for B_1 and B_2
4. $pot_1(B_1) = pot_1(B_2) = 0$
5. $rep(B_1)$ = NEW_BRIDGE(B_1); $rep(B_2)$ = NEW_BRIDGE(B_2)
6. ELEMENTARY_INSERT($rep(B_i), lchild(v)$); ELEMENTARY_INSERT($rep(B_i), rchild(v)$) for $i = 1, 2$

case 3: $|B| < 5a/4$

1. ELEMENTARY_DELETE($rep(B), lchild(v)$); ELEMENTARY_DELETE($rep(B), rchild(v)$)
2. **if** B is isolated in $AS(v)$
3. **then** $pot_1(B) = 0$
4. **else begin**
5. let B' be the one of the neighbor blocks of B
6. join B and B' to obtain a new block for $B \cup B'$ with $rep(B')$
7. apply either case 1 or case 2 to the resulting block
8. **end**

end REORGANIZE.

Lemma 4

(a) The cost of line 1 of ELEMENTARY_INSERT *is* $O(\log n)$ *and the cost of lines 2 to 4 is* $O(\log a + \log\log n)$.

(b) The cost of lines 1 to 4 of ELEMENTARY_DELETE *is* $O(\log\log n)$.

(c) The cost of a call of REORGANIZE *(not counting recursive calls) is* $O(a)$.

Proof:

(a),(b) The bounds are obvious.

(c) Note that reorganization of B either leaves B unchanged (case 1), splits B (case 2), fuses B with a neighbor (case 3 and then case 1) or fuses B with a neighbor and then splits the union of the two blocks (case 3 and then case 2). Thus the time for reorganizing a block is $O(a + \log n) = O(a)$.

□

For the amortized analysis we use the (standard) potential function:

$$pot_1(T) = c_1 \sum_{v \in V} \sum_{\substack{B \text{ block} \\ in\ AS(v)}} pot_1(B),$$

where the constant c_1 is chosen such that $c_1(a/4 - 8)$ bounds the time for reorganizing a block.

Lemma 5

(a) The amortized cost of REORGANIZE *is non-positive.*

(b) The amortized cost of ELEMENTARY_INSERT *is* $O(\log n)$, *the amortized cost of lines 2 to 5 of* ELEMENTARY_INSERT *is* $O(\log\log n)$.

(c) The amortized cost of ELEMENTARY_DELETE *is* $O(\log\log n)$.

Proof:

(a) The cost of reorganizing a block is bounded by $c_1(a/4 - 8)$. So we only need to show that the potential drop is at least that much. This holds true since a block is only reorganized if its potential exceeds $a/4$, since all blocks affected by the reorganization have potential zero after the reorganization and since at most two representatives are deleted from and inserted into both children. The worst case is the application of case 3 and then case 2. In this case the stated bound is reached.

(b) The amortized cost of an elementary insert is bounded by the sum of the real cost of lines 1 to 4, the amortized cost of the reorganization process and the potential increase (which is 1). The claim now follows from part (a).

(c) analogous to part (b).

□

The extension to arbitrary segments is now straightforward. In order to insert segment s into S (delete s from S) call ELEMENTARY_INSERT(s, v) (ELEMENTARY_DELETE(s, v)) for all nodes v having s associated with them. We obtain

Lemma 6 *A sequence of* n *insertions and* m *deletions takes time* $O(nh\log n + mh\log\log n)$.

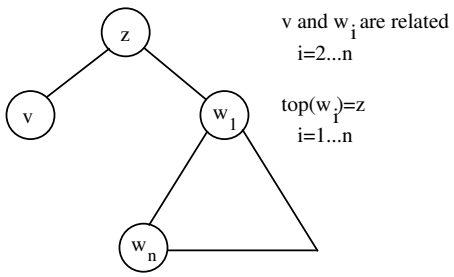

Figure 2: Relations between Nodes

Proof: A non-elementary segment contributes to at most $2h$ segment lists. The amortized time of an insertion is therefore $O(h \cdot \log n)$. The time for a deletion is $O(h \cdot \mathrm{loglog}\, n)$; recall that we postulated that all occurrences of a segment are linked. □

We still need to verify the invariants.

Lemma 7 *The update process maintains the invariants.*

Proof: For Invariants 2 and 3 this is obvious. With respect to Invariant 4 observe that $pot_1(B) \leq a/4$ always (Invariant 2) and that a block contains at least $5a/16$ elements of the same kind as its representative on either side of the representative at the moment the representative is chosen. This holds because at this moment the block has size at least $5a/4$ (see procedure REORGANIZE). Thus at least $a/16$ elements of the same kind lie on both sides of a representative at all times. Thus Invariant 4 holds provided that $a \geq 32$.

It remains to verify Invariant 1. Let v be any node of T and assume inductively that Invariant 1 holds for $parent(v)$. (Observe that Invariant 1 is obviously true for the root of T.) Let b_1 and b_2 be the representatives of distinct blocks B_1 and B_2 of $AS(v)$. By Invariant 4 there are segments in $AS(v)$ with:

$s_{11}, s_{12} \in B_1$ and above b_1, $s_{21}, s_{22} \in B_2$ and below b_2,

s_{i1} and s_{i2} are of the same kind as b_i and do not intersect b_i $(i = 1, 2)$

$b_1, s_{11}, s_{12}, s_{21}, s_{22}, b_2$ intersect $L(v)$ in that order

We now distinguish cases according to the types of b_1 and b_2. If both are proper representatives then each of the four segments s_{ij}, $i, j \in \{1, 2\}$ separates b_1 and b_2. If exactly one is improper, say b_1, then s_{11} and s_{12} are representatives of blocks in $AS(parent(v))$ and are separated by a segment $s_1 \in S$ (induction hypothesis). Thus each of the segments s_1, s_{21}, s_{22} separates b_1 and b_2. If both representatives are improper then a similar argument shows the existence of segments s_1 and s_2 in S separating b_1 and b_2. □

2.3.2 An Improved Insertion Algorithm

Call node v of T the *upper relative* of node w of T if v is the left child of some node z and w lies on the left spine of the subtree rooted at the right child of z or if the symmetric situation holds (cf. Figure 2). We will also simply say that nodes v and w are *related*. Note that every segment $s \in S$ is associated with two sequences of nodes of T such that neighboring nodes in each sequence are related (cf. Figure 3). For related nodes v and w let $L(v, w) = \{s \in S; s \in S(v) \cap S(w)\}$ be the set of segments stored in the segment lists of v and w.

The improved insertion algorithm inserts segments bottom-up. So suppose that segment s has already been added to $S(w)$ and $AS(w)$ and that s has to be added to $S(v)$ and $AS(v)$ next where v is the upper relative of w. Assume inductively that the segments $s', s'' \in L(v,w)$ immediately above and below s are known. The idea is to locate s in $AS(v)$ by a simultaneous finger search (cf. [?, Vol.I, section III.5.3.3.]) starting at the occurrences of s' and s'' in $AS(v)$ and to determine the neighbors of s in $L(z,v)$, where z is the upper relative of v, by a $Find$ in an appropriate UFS-structure. Although the finger search takes time $O(\log n)$ in the worst case we will be able to show that it takes only time $O(d + \log\log n)$ in the amortized sense, where $d = dist(v,w)$ is the distance of nodes v and w in the underlying search tree T. Summation of this bound along the insertion path yields an $O(\log n \cdot \log\log n)$ bound on the amortized insertion cost (Note that the $dist$-terms sum up to $O(h) = O(\log n)$).

In order to carry out this idea we augment our data structure in two ways:

1. For each node w a UFS-structure on $AS(w)$ is maintained where the marked items are precisely the segments in $L(v,w)$ for v the upper relative of w. We call this structure the UFS-structure $\mathcal{L}$ on $AS(w)$.

2. For each node w the blocks of $AS(w)$ are stored in a finger search tree $T_{AS}(w)$, more precisely a level linked (4,8)-tree, cf. [?, Vol.I, section III.5.3.3.]. In such a tree the insertion or deletion of an item takes time $O(\log n)$ and a search for an item in distance D from a finger ($\widehat{=}$a pointer to a leaf) takes time $O(\log D)$. In $T_{AS}(w)$ the leaves correspond to the blocks of $AS(w)$. To simplify reading we use the word vertex for the nodes of the trees T_{AS} and reserve the word node for the nodes of T.

We are now ready for the details of the insertion algorithm. It first determines the two sequences of nodes into which the new segment has to be inserted. Let $w_1, \ldots, w_l$ with $l < h$ be one of the sequences where w_i is the upper relative of w_{i-1} for $i \geq 2$. Let $i \geq 2$ and assume inductively that the segments s' and s'' immediately above and below s in $L(w_i, w_{i-1})$ are known and that s has to be inserted into $AS(w_i)$ next.

(1) Determine the blocks B' and B'' of $AS(w_i)$ containing s' and s'' respectively.

(2) Determine the block B of $AS(w_i)$ into which s has to be inserted by simultaneous finger search in $T_{AS}(w_i)$ starting from B' and B''.

(3) Insert s into B, reorganize B (if necessary) and update the UFS-structures $\mathcal{I}$ and $\mathcal{L}$ on $AS(w_i)$.

(4) Locate s in L_{w_{i+1},w_i} by a $Find$-operation in $\mathcal{L}$ (this determines s' and s'' for the next step.)

The following Lemma is crucial for the amortized analysis of step (2).

Lemma 8 *Let $d = dist(w_i, w_{i-1})$ be the distance between nodes w_i and w_{i-1} in the underlying search tree T and let x and y be the vertices of height d in $T_{AS}(w_i)$ and $T_{AS}(w_{i-1})$ respectively into whose subtrees the segment s is inserted. Then either the finger search for s in $T_{AS}(w_i)$ does not reach a node of height larger than d (and hence the time for step (2) is $O(d)$) or there is no segment $t \in L(w_i, w_{i-1})$ different from s which is stored in the subtrees rooted at x and y.*

Proof: Assume that there is a segment $t \in L(w_i, w_{i-1})$ different from s which is stored in the subtree of $T_{AS}(w_i)$ rooted at x and the subtree of $T_{AS}(w_i)$ rooted at y. Then either the segment s' or the segment s'' has this property. Thus the finger search for s in $T_{AS}(w_i)$ does not leave the subtree rooted at x. □

We next define an essential ingredient for the amortized analysis. Consider any pair (v,w) of related nodes and let $d = dist(v,w)$ be the distance of nodes v and w in the underlying search tree. Call vertices x and y in $T_{AS}(v)$ and $T_{AS}(w)$ *connected* if both of them have height d and there is a segment stored in the subtrees rooted at x and y. Let $con(v,w)$ be the number of connected pairs of vertices in $T_{AS}(v)$ and $T_{AS}(w)$.

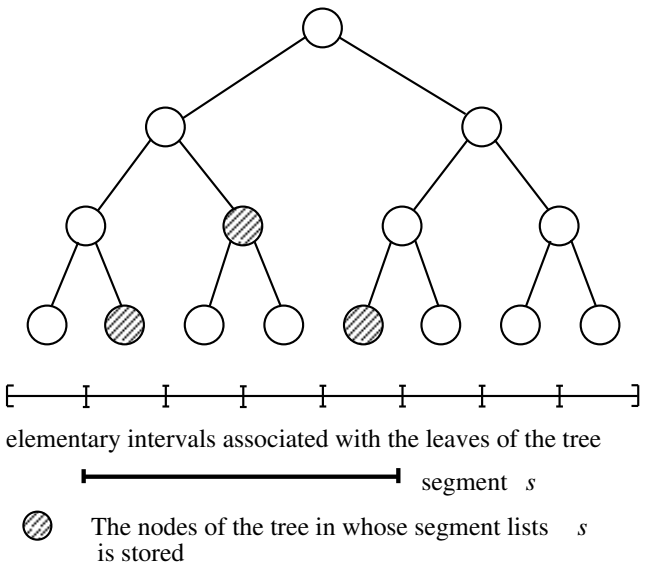

Figure 3: The segment lists in which a segment s is stored

Lemma 9 *a)* $con(v,w) \leq (|AS(v)| + |AS(w)|)/(4^d a)$, *where* $d = dist(v,w)$

b) $\sum_{v,w \text{ related}} con(v,w) = O(n \cdot h/a)$

Proof:

a) Note first that there are at most $|AS(v)|/(4^d a)$ vertices of height d in $T_{AS}(v)$ and at most $|AS(w)|/(4^d a)$ vertices of height d in $T_{AS}(w)$. Observe next that the segments in $L(v,w)$ are linearly ordered by the above-relation. Thus there can be at most $(|AS(v)|+|AS(w)|)/(4^d a)$ connected pairs of vertices in $T_{AS}(v)$ and $T_{AS}(w)$.

b) We sum the bound derived in part a) over all pairs of related nodes. Then every node v contributes $O(h)$ terms involving $|AS(v)|$ to the bound. As a lower relative the contribution is at most $|AS(v)|/a$ and as an upper relative the contribution is at most $\sum_{d \geq 3} |AS(v)|/(4^d a) \leq |AS(v)|/a$. The bound now follows from Lemma 1.

□

We use the following potential function for the amortized analysis:

$$pot(T) = pot_1(T) + pot_2(T) + pot_3(T)$$

where

$$pot_1(T) = c_1 \sum_{\substack{B \text{ block} \\ \text{in } T}} pot_1(B)$$

$$pot_2(T) = c_2 \log n \sum_{\substack{v,w \\ related}} \left(\frac{|AS(v)| + |AS(w)|}{4^{dist(v,w)} a} - con(v,w) \right)$$

and

$$pot_3(T) = c_3 |n - n_0| \log n$$

We maintain the invariant that $|n - n_0| \leq n_0/2$ and $a = max\{36, \lfloor \log^2 n_0 \rfloor\}$; n_0 and a are adjusted whenever n leaves the interval $[n_0/2, 3n_0/2]$. We first extend Lemma 5.

Lemma 10 *The amortized cost of block reorganization is non-positive.*

Proof: We have argued in the previous section that the actual cost of block reorganization is $O(a)$. pot_1 drops by at least $c_1(a/4-8)$ and pot_3 does not change. It remains to treat pot_2. Consider the reorganization of a block B in $T_{AS}(v)$. The reorganization of B may cause a splitting of B into two blocks, a fusing of B with a neighboring block or an exchange of segments with a neighboring block. Consider any ancestor x of B in $T_{AS}(v)$ and let d be its height. If the reorganization of B does not propagate to vertex x then the connections of x do not change. If x is split into nodes x' and x'' then every connection of x becomes a connection of either x' or x'' and hence the number of connections does not decrease. If x is fused with a neighbor vertex x' then the number of connections drops by the number of vertices y which have connections with x as well as x'. Since connections do not "cross" there can be at most two such vertices y, one in the upper relative of v and one in the lower relative of v with distance d from v. We conclude that the reorganization of B removes at most two connections for each ancestor of B in $T_{AS}(v)$ and hence increases $pot_2(T)$ by at most $c_2 \cdot \log^2 n$. For c_1 large enough the cost of block reoganization is therefore non-positive. □

Lemma 11 *The amortized cost of an insertion is* $O(\log n \cdot \log\log n)$.

Proof: Recall that a segment is inserted into two sequences of nodes $w_1, \ldots, w_l$, $l < h$. It suffices to show that the amortized cost arising in node w_i, $i \geq 2$ is $O(d + \log\log n)$ where $d = dist(w_i, w_{i-1})$. This is clear for steps (1) and (4) and follows for step (3) from Lemma 5 and the observation that pot_1 and pot_2 both increase by $O(1)$ by the addition of an elementary segment. It remains to consider step (2). Lemma 8 implies that either the time for the finger search is $O(d)$ or $con(w_i, w_{i-1})$ increases by 1. Since the finger search time is always at most $O(\log n)$ this proves that the amortized cost of step (2) is $O(d)$. □

We treat deletions next. In order to remove a segment s, the segment has to be removed from $O(h)$ augmented node lists. Each deletion of an elementary segment has actual cost $O(\log\log n)$ increases pot_1 by $O(1)$ and increases pot_2 by $O(\log n)$ (Note that the removal of an elementary segment destroys at most one connection). Also pot_3 increases by $O(\log n)$. We summarize in:

Lemma 12 *The amortized cost of a deletion is* $O(\log^2 n)$.

2.3.3 Maintaining the underlying search tree T and the parameter a

Up to this point we worked with a fixed search tree T and a fixed value of the parameter a. In this section we remove both assumptions. Recall that the underlying search tree T is a BB[1/4]-tree for the x-coordinates of the endpoints of the segments and that these coordinates are assumed to be distinct. BB[1/4]-trees are reorganized by rotations. In [?, ?, ?], cf. also [?], it was shown that the amortized rebalancing cost per update operation is $O(\log n)$ provided that a rotation (cf. Figure 4) at a node v takes amortized time $O(th(v))$, where $th(v)$ is the number of leaves in the subtree rooted at v.

Lemma 13 *A rotation at a node v can be performed in amortized time* $O(th(v))$.

Proof: We only treat a rotation to the left as shown in Figure 4 and leave the discussion of a rotation to the right to the reader. Note first that $th(v) \leq 4^{dist(v,w)} th(w)$ for any two nodes v and w of a BB[1/4]-tree (cf. [?], Lemma 11). Observe next that $|S(v)| \leq th(v) = O(th(v))$ for all nodes v since the x-coordinates of the segments are assumed to be distinct. This implies $|AS(v)| = O(th(v))$ for all nodes v. For the root this follows from $|S(root)| = |AS(root)|$ and for the other nodes this follows by induction from $|AS(v)| \leq |S(v)| + |AS(parent(v))|/a$.

A rotation changes the S-sets for nodes u, v, w, x, y, cf. Figure 4. The following formulae define the *xranges* and the segment lists after the rotation (we use primes to denote the situation after the rotation):

$$xrange'(v) = xrange(u) \cup xrange(w)$$

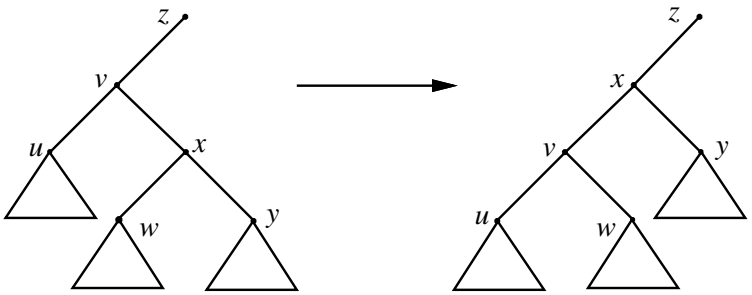

Figure 4: A left rotation at node v

$$xrange'(x) = xrange(v)$$

$$S'(u) = S(u) \setminus (S(u) \cap S(w))$$

$$S'(w) = S(w) \setminus (S(u) \cap S(w)) \cup S(x)$$

$$S'(y) = S(y) \cup S(x)$$

$$S'(x) = S(v)$$

$$S'(v) = S(u) \cap S(w)$$

The new segment list of the involved nodes can be computed according to the formulae above by a linear scan of the corresponding lists. For the augmented segment lists we proceed as follows. We first remove the representatives of $AS(u)$, $AS(w)$ and $AS(y)$ from their children. We then set $AS'(x)$ to $AS(v)$ and compute $AS'(v)$, $AS'(u)$, $AS'(w)$ and $AS'(y)$ from the corresponding S-lists and the representatives of the blocks of the parent nodes. We also propagate the representatives of the blocks of $AS'(u)$, $AS'(w)$ and $AS'(y)$ to their children. All of this takes time linear in the size of the AS-lists handled plus time $O(\log n)$ per representative handled and thus total time $O(th(v) + \log n \cdot th(v)/a) = O(th(v))$. The change in pot_1 is also clearly bounded by $O(th(v))$.

Next we set up the UFS-structures $\mathcal{I}$ and $\mathcal{L}$ for the AS-lists of nodes u, v, w, x, y. For both structures it can be decided in constant time whether an item is marked. This is obvious for the structure $\mathcal{I}$ and follows for the structure $\mathcal{L}$ from the fact that all occurrences of a segment are linked in in-order. We conclude that this step takes time $O(th(v))$.

Finally, we make the required changes to the $\mathcal{L}$-structures of all nodes having a node in $\{u, v, w, x, y\}$ as an upper relative. Let $p \in \{u, v, w, x, y\}$ be arbitrary. All nodes having p as an upper relative lie on the spine of some subtree, e.g., all nodes z having u as the upper relative after the rotation lie on the left spine of the subtree rooted at w. For each node z the new $\mathcal{L}$-structure can be set up in time $O(|AS(z)|) = O(th(z))$. The total time is therefore $O(th(v))$ since the thicknesses of the nodes on a spine form a geometric series.

It remains to estimate the change in pot_2 and pot_3. pot_3 does not change. pot_2 increases due to two effects: The length of some augmented node lists changes and connections may disappear. The first effect increases pot_2 by at most

$$O(\log n \sum_{\substack{p \\ involved}} |AS(p)|/a) = O(th(v)).$$

In order to estimate the number of destroyed connections we distinguish two cases. A connection between $T_{AS}(i)$ and $T_{AS}(j)$ can only be destroyed if either i is the upper relative of v or x and j belongs to the subtree rooted at v or i is one of the nodes u, v, w, x, or y and j belongs to the

subtree rooted at v. We consider the former case and here only the case that i is the upper relative of v. Let $j_0, j_1, j_2, \ldots$ be the nodes on the left spine of the subtree rooted at v; $v = j_0$. Also let $d = dist(i, v)$. Then the number of removed connections is at most

$$\sum_{l \geq 1} con(i, j_l) \leq \sum_{l \geq 1} \frac{|AS(i)| + |AS(j_l)|}{a \cdot 4^{d+l}}$$

where the inequality follows from Lemma 9.

Next observe that $|AS(j_l)| = O(th(j_l)) = O(th(v))$ and that $|AS(i)| = O(th(i)) = O(4^d th(v))$. Thus

$$\sum_{l \geq 1} con(i, j_l) \leq \sum_{l \geq 1} \frac{O(th(v))}{a \cdot 4^l} = O(th(v)/a)$$

A similar argument shows that the number of destroyed connections of the second kind is also $O(th(v)/a)$. Thus pot_2 increases by at most $O(th(v) \log n/a) = O(th(v))$.

Altogether we have shown that the amortized cost of a rotation at v is $O(th(v))$. □

The parameter a is changed whenever n leaves the range $[n_0/2, 3n_0/2]$. In this case we set n_0 to n, a to $max\{36, \lfloor \log^2 n_0 \rfloor\}$ and rebuild the entire structure. All of this has actual cost $O(n \log n)$. pot_3 decreases by $c_3 \cdot n \log n$ (since $pot'_3 = 0$), pot_1 does not increase (since $pot'_1 = 0$) and pot_2 increases by at most $O(n \cdot h \cdot \log n/a) = O(n)$ (since $pot'_2 = O(n \cdot h \cdot \log n/a)$ according to Lemma 9,b)). The amortized cost of adapting a is therefore non-positive.

Putting eveything together we obtain:

Theorem 14 *Augmented segment trees support insertions and point location queries in time* $O(\log n \log\log n)$ *and deletions in time* $O(\log^2 n)$. *The time bounds for insertions and deletions are amortized. Augmented segment trees take space* $O(n \log n)$.

3 The Combined Interval-Segment Tree

In this section we improve the space requirement to linear, while keeping all the time bounds as in section 2. Interval trees and interval-priority-search trees are the additional ingredient.

3.1 The Combined Data Structure

Let S be a set of pairwise non-intersecting (except at endpoints) non-vertical line segments and let T be a search tree for the x-coordinates of the endpoints of the segments in S. Let h be the height of T and let V be the set of nodes of T. We assume $h = O(\log n)$. We associate the segments in S with the nodes in V as in interval trees, i.e., the *node list* of $v \in V$ is defined as

$$N(v) = \{s \in S;\ home(s) = v\}.$$

We split each segment s at the vertical line $L(home(s))$ into segments s^+ and s^-. From now on, we deal only with the segments s^+ ($s \in S$), and write s instead of s^+.

The list $N(v)$ is ordered according to the y-coordinates of the intersections of the segments in $N(v)$ with the vertical line $L(v)$. The list $N(v)$ is partitioned into subsequences, which we call *N-blocks* or simply *blocks*. An N-block is called *isolated* if it the only block of $N(v)$ and *non-isolated* otherwise. For an N-block B, let $win(B)$ be the segment in B with the rightmost right endpoint. Call $win(B)$ the *winner* of block B and let W be the set of all winners. As in the previous section we maintain for every N-block B a counter $pot_1(B)$ which counts the number of insertions into and deletions from B since the creation or last reorganization of B. We maintain Invariant 2 also for N-blocks.

With every node $v \in V$ we also associate an *auxiliary node list* $AN(v)$, a *segment list* $S(v)$, and an *augmented segment list* $AS(v)$. The relation between these lists is governed by Invariants 5, 6 and 7.

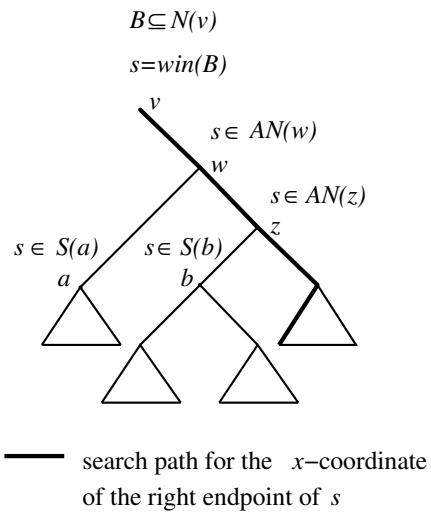

Figure 5: Propagation of Winners

Invariant 5

(a) If B is a non-isolated block of $N(v)$ then $s = win(B) \in AN(w)$ for all descendants w of v on the search path to the x-coordinate of the right endpoint of s which are left through their rchild-pointer.

(b) If $s \in AN(v)$ then $s \in S$ and s intersects $L(top(v))$ and $L(v)$ but does not intersect $RB(v)$, i.e., $home(s) \in \{top(v), top(top(v)), top(top(top(v))), \ldots\}$. Here $RB(v)$ denotes the right boundary of $range(v)$.

(c) $AN(root) = \emptyset$.

(d) $N(v)$ and $AN(v)$ are ordered according to the y-coordinate of the intersections of the segments with the line $L(v)$.

Remark: Winners are propagated from their home along the search path to the right endpoint of the winner (see Figure 5) and are stored in the AN-lists of all nodes in this path which are left through their *rchild*-pointer. The intuition underlying this propagation process is as follows.

Assume first that a point q has to be located with respect to the segments in $N(v)$ for some node v and that the position of q with respect to the winners of the blocks of $N(v)$ is already known, say q lies between $win(A)$ and $win(B)$ with $win(A)$ above $win(B)$. Then only blocks C between A and B inclusive can contain segments which intersect $L(q)$ between $win(A)$ and $win(B)$. Also no block C strictly between A and B contains a segment which intersect $L(q)$ since the winner of no such block does. Thus locating q in $A \cup B$ is tantamount to locating q in $N(v)$. However, locating q in $A \cup B$ takes only time $O(\log a) = O(\log\log n)$.

Assume next that a point q has to be located with respect to the union of $N(z)$ over all nodes z on the right spine of T. Let v and $w = rchild(v)$ be two nodes on the right spine and assume inductively that q lies right of $L(w)$ and has already been located with respect to $AN(w) \cap W$. $AN(w)$ contains all winners of blocks of $N(v)$ which intersect $L(w)$ and also all segments in $AN(v) \cap W$ which intersect $L(w)$; it may also contain other segments (the purpose of the other segments is explained in the remark following Invariant 7). Thus $Find$-operations in appropriate UFS-structures on $AN(w)$ (in one, the marked items are the segments in $N(v) \cap W \cap AN(w)$, and in the other, the marked items are the segments in $AN(v) \cap W \cap AN(w)$) identify the two blocks

in $N(v)$ and $AN(v)$ respectively to which the search for q can be restricted. The incremental cost of locating q in $N(v)$ and $AN(v)$ is therefore $O(\log\log n)$.

Consider finally the general situation that q has to be located with respect to the union of $N(v)$ over all nodes on the search path for the x-coordinate of q. Let $v_1, v_2, v_3, \ldots, v_l$ be a subpath of the search path such that v_1 and v_l are left through their *rchild*-pointer and $v_2, \ldots, v_{l-1}$ through their *lchild*-pointer. By the argument in the preceding paragraph the incremental cost of locating q in $N(v_1)$ and $AN(v_1)$ is $O(\log\log n)$ if the location with respect to the segments in $N(v_1) \cap W$ and $AN(v_1) \cap W$ is known. Every segment in $(N(v_1) \cup AN(v_1)) \cap W$ which intersects $L(q)$ (and hence $L(v_l)$) is also contained in $AN(v_i)$ for some i, $2 \leq i \leq l$. We may assume inductively that the position of q with respect to $AN(v_l) \cap W$ is known. For the nodes v_i, $2 \leq i \leq l-1$, we use the following observation. Every segment s in $(N(v_1) \cup AN(v_1)) \cap AN(v_i)$ completely spans $range(v_{i+1})$, i.e., would stored at v_{i+1} in a segment tree. We therefore store the segments in the AN-lists also in the segment tree data structure of section 2 (cf. Invariant 6) and then use the query algorithm of section 2.2 to locate q with respect to the segments in $(N(v_1) \cup AN(v_1)) \cap AN(v_i)$ for $2 \leq i \leq l-1$.

Invariant 6

(a) $S(root) = \emptyset$

(b) $S(v) = \emptyset$ *if* v *is a right child and* $S(v) = AN(parent(v))$ *if* v *is a left child*

(c) Invariants 1 to 4 hold for $S(v)$ *and* $AS(v)$.

Remark: A segment $s \in S(v)$ satisfies $xrange(v) \subseteq proj(s)$ and $\neg(xrange(parent(v)) \subseteq proj(s)$, i.e., s would be stored at v in a segment tree for S. The S- and AS-lists form the structure of section 2 for the set $\bigcup_v S(v)$. This set contains the set W of winners and is in general a proper subset of the set S. The lists $AS(v)$ and $S(v)$ are structured into blocks as described in section 2.1. By virtue of part (b) of Invariant 6 this also induces a block structure on $AN(parent(v))$. As in section 2 use $L(v,w)$ to denote $S(v) \cap S(w)$.

For related nodes v and w we define the following: An element $s \in L(v,w)$ is called a *winner-connection* or simply *w-connection* if $s \in W$ and an *nw-connection* otherwise.

Invariant 7

(a) Let v *and* w *be related and* $dist(v,w) = d$. *Let* x *and* y *be nodes of height* d *in* $T_{AS}(v)$ *and* $T_{AS}(w)$ *respectively . Then there is at most one nw-connection stored in the subtrees rooted at* x *and* y.

(b) Each segment $s \in S(v)$ *either belongs to* W *or is member of a nw-connection.*

Remark: Invariant 7 states that the segment tree part of our data structure may also contain non-winners. These non-winners connect the S-lists of related nodes and support the insertion process as described in section 2.3.2. Part (a) of Invariant 7 controls the number of non-winning connections.

We can now derive a bound on the total size of all lists.

Lemma 15 *Let* $N = \sum_v |AS(v)|$.

(a) The maximal number of connected pairs of vertices in trees T_{AS} *is at most* $2 \cdot N/a$.

(b) $\sum_v |S(v)| \leq n \cdot h/a + 4 \cdot N/a$.

(c) If $a \geq max\{2h, 10\}$ *then* $N \leq n$.

Proof:

(a) This follows directly from Lemma 9.

(b) By Invariant 7, each element of $S(v)$ is either a winner or a member of a nw-connection. Each element of W belongs to the S-set of at most h nodes, $|W| \leq n/a$ (by the definition of W), and the number of members of nw-connections is bounded by twice the number of connected pairs of vertices in the trees T_{AS}. The bound follows.

(c) We have $|AS(v)| \leq |S(v)| + |AS(parent(v))|/a$ for all nodes v. Thus $(1-1/a)N \leq \sum_v |S(v)| \leq n \cdot h/a + 4 \cdot N/a$. For $a \geq 2 \cdot h$ and $a \geq 10$ this implies $N \leq n$.

□

We close this section with a description of the low-level details of the data structure:

- All occurrences of a segment are linked in in-order.
- For every node v, $N(v)$ is stored in a balanced tree.
- For every node v, every block of $N(v)$ and $AN(v)$ is organized as described in [?], i.e., is stored in a balanced priority search tree. This gives query, insertion and deletion time $O(\log a) = O(\log\log n)$. The update time is $O(\log a)$ instead of $O(\log^2 a)$ as stated in table 1 since all segments in $N(v)$ and $AN(v)$ have their left endpoint on the vertical line $L(v)$, cf. [?].
- The segment tree part of our data structure (S- and AS-lists) is organized as described in section 2.
- Also as in section 2, T is a BB[1/4]-tree for the set of endpoints of the segments in S.
- For every vertex v, the list $AN(v)$ is stored in two UFS-structures. In the first structure the marked items are the segments in $AN(v) \cap W \cap N(top(v))$ and in the second structure the marked items are the segments in $AN(v) \cap W \cap AN(top(v))$. The $Find$-operations are called $Find_N_winner_above$, $Find_N_winner_below$ and $Find_AN_winner_above$ and $Find_AN_winner_below$ respectively. These structures are called the $\mathcal{WN}$- and $\mathcal{WAN}$-structure on $AN(v)$ respectively.
- For every vertex w, the list $S(w)$ is stored in an UFS-structure. The marked items are the nw-connections between $S(w)$ and $S(v)$ where v is the upper relative of w. (Of course, this is equivalent to a UFS-structure on $AN(parent(w))$. However, it is more convenient to view this UFS-structure as a structure on $S(w)$.) We call this the nw-structure on $S(w)$.

Lemma 16 *The space requirement of the data structure is linear.*

Proof: This follows immediately from Lemma 15 and the definition of the data structure. □

In the following two sections we discuss the update and the query algorithms. These sections can be read independently.

3.2 Insertions and Deletions

As in section 2 we proceed in two steps. In the first step we assume that the underlying search tree T and the parameter a are fixed and in the second step we discuss how to rebalance T and how to adjust the parameter a. For the amortized analysis of insertions and deletions we use the same potential function $pot(T)$ as in section 2.3.2.

Consider insertions first. Let s be the segment to be inserted and let $v = home(s)$. We assume w.l.o.g. that s has its left endpoint on $L(v)$.

(1) Determine $v = home(s)$, insert v into $N(v)$ and into the appropriate block A of $N(v)$.

(2) **if** s is the new winner of the block A and A is non-isolated

(3) **then** let t be the old winner of block A. Declare all connections involving t to be non-winning and remove parallel nw-connections;

(4) let $w_0 = v, w_1, \ldots, w_k$ be the nodes on the search path to the x-coordinate of the right endpoint of s which are left through their rchild-pointer and for i with $1 \leq i \leq k$ let x_i be the left child of w_i; insert s into $AN(w_i)$ and $AS(x_i)$ for $1 \leq i \leq k$;

(5) **fi**;

(6) **for** all N- and AS-blocks B touched

(7) **do** increase $pot_1(B)$

(8) **if** $pot_1(B)$ now exceeds $a/4$ **then** REORGANIZE(B) **fi**

(9) **od**

We now discuss lines (3), (4) and (6) to (8) in detail.

Line (3) takes amortized time $O(h \cdot \log\log n)$. This can be seen as follows. First declare all connections involving t to be non-winning. This is possible in time $O(h)$ because all occurrences of a segment are linked and t is in at most $O(h)$ AN-lists and thus S-lists. By unmarking t with *Union(t)* in time $O(\log\log n)$ per list and hence total time $O(h \cdot \log\log n)$ update the nw-UFS-structure for all AN-lists containing t. Then determine whether there are nw-connections parallel to t and if so remove them as follows. Let v and w with v the upper relative of w be nodes such that $t \in L(v, w)$. Let t' and t'' be the nw-connections above and below t (t' and t'' can be found by $Find$-operations) and let x and y be the vertices of height $d = dist(v, w)$ of $T_{AS}(v)$ and $T_{AS}(w)$ containing t in their subtrees. Check whether t' or t'' is also stored in the subtrees rooted at x and y (by a walk of length d towards the root). If so, t is parallel to either t' or t'' and t is removed. All of this takes amortized time $O(d + \log\log n)$. Thus all parallel nw-connections involving t can be removed in amortized time $O(h \cdot \log\log n)$. Note that pot_2 and pot_3 do not change and that pot_1 changes by $O(h)$.

For line (4) we use the results of section 2.3.2. We first add s to $AN(w_k)$ and $S(x_k)$ and then to $AN(w_i)$ and $S(x_i)$ for $i = k - 1$ down to 1. According to Lemma 11 all of this takes amortized time $O(h \cdot \log\log n)$.

It remains to discuss line (8), i.e., the reorganization of N-blocks. The reorganization of N-blocks is similar to the reorganization of S-blocks as described in section 2.3.1. As in procedure REORGANIZE we distinguish 3 cases. If the blocksize is in the range between $5a/4$ and $10a/4$ then we only reset the potential of A. In the other cases we split the block or fuse it with a neighboring block and then possibly split the fused block. Also the winners of the touched blocks are declared non-winners and the new winners of the blocks are propagated. All of this takes non-positive amortized time. Note that pot_1 drops by $\Omega(a)$, pot_2 and pot_3 do not change and that the cost of declaring the old winners to non-winners and propagating the new winners is $O(h \cdot \log\log n)$ according to the preceding two paragraphs.

We summarize in:

Lemma 17 *The amortized cost of an insertion of a line segment into the combined interval tree is* $O(h \cdot \log\log n) = O(\log n \log\log n)$.

Deletions are much simpler to handle. To delete s, remove all copies of s in the data structure, update the UFS-structures, increase the potentials of all blocks touched, and reorganize if necessary. This has actual cost $O(\log n \log\log n)$ and increases the potential by $O(h \cdot \log n)$.

Lemma 18 *The amortized cost of a deletion is* $O(h \cdot \log n) = O(\log^2 n)$.

We next turn to the dynamic behavior of BB[1/4]-tree T. Consider a rotation to the right at inner node v, cf. Figure 4; we leave the treatment of rotations to the left to the reader. As in section 2.3.3 our goal is to show that a rotation at node v has amortized cost $O(th(v))$.

We first show that the size of all lists stored at a node v is $O(th(v))$. For the N- and AN-lists this follows from the observation that $s \in N(v) \cup AN(v)$ implies that the right endpoint of s is stored in the subtree rooted at v and the assumption that all x-coordinates of endpoints are distinct. For the S-list this follows from the previous sentence and Invariant 6, and for the AS-list this follows from $AS(root) = S(root)$ and $|AS(v)| \leq |S(v)| + |AS(parent(v))|/a$ for nodes v different from the root.

A rotation about v is performed in six steps.

1. Declare the winners of all blocks of $N(v)$ and $N(x)$ to non-winners. As described for line (3) above this takes amortized time $O(h \cdot \log\log n)$ per winner and hence $O(((th(v) + th(x))/a) \cdot h \cdot \log\log n) = O(th(v))$ total amortized time.

2. Compute new ranges and lists.

   $$xrange'(v) = xrange(x)$$
   $$xrange'(x) = xrange(w) \cup xrange(y)$$
   $$N'(v) = N(v) \cup \{s \in N(x),\ key(v) \in proj(s)\}$$
   $$N'(x) = N(x) \setminus \{s \in N(x),\ key(v) \in proj(s)\}$$
   $$AN'(v) = AN(v) \cup AN(x)$$
   $$AN'(x) = AN(x)$$
   $$S'(u) = S(u) \cup S(v)$$
   $$S'(w) = S(v)$$
   $$S'(y) = S(y) = \emptyset$$
   $$S'(v) = S(x)$$
   $$S'(x) = \emptyset$$

   All new lists can be computed in time $O(th(v))$ by linear scans.

3. Make the required changes in the segment tree structure of section 2. This takes amortized time $O(th(v))$.

4. Propagate the new winners of the blocks of $N'(x)$ and $N'(v)$. As described for line (4) above this requires only the segment tree structure (which was already updated in step 2) and takes time $O(h \cdot \log\log n)$ per winner and hence $O(th(v))$ total time.

5. Update $\mathcal{WN}$- and $\mathcal{WAN}$-structures. The $\mathcal{WN}$- and $\mathcal{WAN}$-structures change only for the AN-lists of nodes p which either belong to $\{u, v, w, x, y\}$ or lie on the spine of the subtrees with roots u, w or y. For each node p a new $\mathcal{WN}$- and $\mathcal{WAN}$-structure for $AN(p)$ can be constructed in time $O(|AN(p)|) = O(th(p))$ since constant time suffices to decide whether a segment has to be marked (this follows from the fact that all occurences of a segment are linked in in-order). The total time for this step is therefore $O(th(v))$ since the thicknesses of the nodes on a spine form a geometric series.

6. Update nw-structures. The nw-structure on $S(p)$ changes only for nodes p which either belong to $\{u, v, w, x, y\}$ or lie on the spine of the subtrees with roots u, w or y. Let p be one of these nodes and let d be the distance from p to its upper relative p'. One can certainly check in time $O(\log n)$ per nw-connection whether it satisfies Invariant 7 and if not remove the connection. Thus the nw-structure on $S(p)$ can be updated in time $O(\log n \cdot (|S(p)| + |S(p')|)/(4^d a)) = O(\log n \cdot th(p)/a) = O(th(p))$. As for step 5 we conclude that the total time for step 6 is $O(th(v))$.

We have now shown that the actual cost of a rotation about v is $O(th(v))$. The potential also increases by at most that amount (by the argument used in the proof of Lemma 13). We summarize in:

Lemma 19 *A rotation at node v takes amortized time $O(th(v))$.*

The parameter a is changed as discussed in section 2.3.3. Altogether we have shown:

Theorem 20 *The combined interval tree supports insertions in time $O(\log n \log\log n)$ and deletions in time $O(\log^2 n)$. The bounds are amortized.*

3.3 The Query Algorithm

The query algorithm gets a point $q \in \mathbb{R}^2$ and is supposed to return the segments immediately above and below q. It works in two stages. In the first stage, called the segment tree query, it determines for all nodes z on the search path to the x-coordinate of q the segments $s(z)$ and $t(z)$ in $S(z)$ immediately above and below q. In the second stage, called the interval tree query, it uses the results of the first stage and determines the segments in S immediately above and below q. The first stage was already described in section 2.2; c.f. the remark at the end of that section. The intuition underlying the second stage was given in the remark following Invariant 5 in section 3.1. For the detailed description of the second stage we need the concept of a funnel.

Let $w_1, w_2, \ldots, w_k$ be the sequence of nodes in the combined interval tree which are left through their rchild-pointer during the search for the x-coordinate of q. The algorithm in the second stage constructs for $i = k, k-1, \ldots, 1$ a *funnel* F_i. The funnel F_i consists of an upper path UP_i and a lower path LP_i. The upper path UP_i (and similarly the lower path) consists of a sequence of segments $s_1, \ldots, s_r$ and a sequence $j_1, \ldots, j_r$ of indices having the following properties (cf. Figure 6):

(1) $j_1 = i$ and $j_l < j_{l+1}$ for $1 \le l < r$.

(2) For l, $1 \le l \le r$, $s_l \in N(w_{j_l}) \cup AN(w_{j_l})$ and s_l extends at least to$L(w_{j_{l+1}})$, i.e., intersects that line (for $l = r$, let $L(w_{j_{l+1}}) = L(q)$).

(3) s_l intersects $L(w_{j_{l+1}})$ above s_{l+1} for $1 \le l < r$ and s_r is above q

(4) there is no segment $s \in W \cap (N(w_i) \cup AN(w_i))$ which extends all the way to $L(q)$ and which intersects $L(w_i)$ between the two boundaries of the funnel.

The funnel F_k is readily constructed. Locate q in $AN(w_k)$ and $N(w_k)$ and let s and t be the segments immediately above and below q, respectively. Then $UP_k = (s)$ and $LP_k = (t)$ has the desired properties.

We next show how to construct F_i from F_{i+1}. Let

$$UP_{i+1} = (s_1, \ldots, s_r), \qquad LP_{i+1} = (t_1, \ldots, t_p)$$

and let $j_1, j_2, \ldots$ and $k_1, k_2, \ldots$ be the associated index sequences. Let

$$\begin{aligned} \overline{s}_N &= Find_N_winner_above(s_1), \\ \overline{s}_{AN} &= Find_AN_winner_above(s_1), \\ \overline{t}_N &= Find_N_winner_below(t_1), \\ \overline{t}_{AN} &= Find_AN_winner_below(t_1). \end{aligned}$$

Then $\overline{s}_N$ is the lowest segment in $AN(w_{i+1}) \cap W \cap N(w_i)$ above or equal to s_i and $\overline{s}_{AN}$ is the lowest segment in $AN(w_{i+1}) \cap W \cap AN(w_i)$ above or equal to s_i. Analogously, $\overline{t}_N$ and $\overline{t}_{AN}$ are highest segment below or equal to s_i. Let Z_i be the set of nodes on the path from $rchild(w_i)$ (excluding) to w_{i+1} (including). If $w_{i+1} = rchild(w_i)$ then $Z_i = \emptyset$. For every $z \in Z_i$ let $s(z)$ and $t(z)$ be the segments in $S(z)$ immediately above and below q (these segments were determined in stage 1), let $\overline{s}_Z$ be the lowest segment in $\{s(z); z \in Z_i\}$ and $\overline{t}_Z$ be the highest segment in $\{t(z); z \in Z_i\}$.

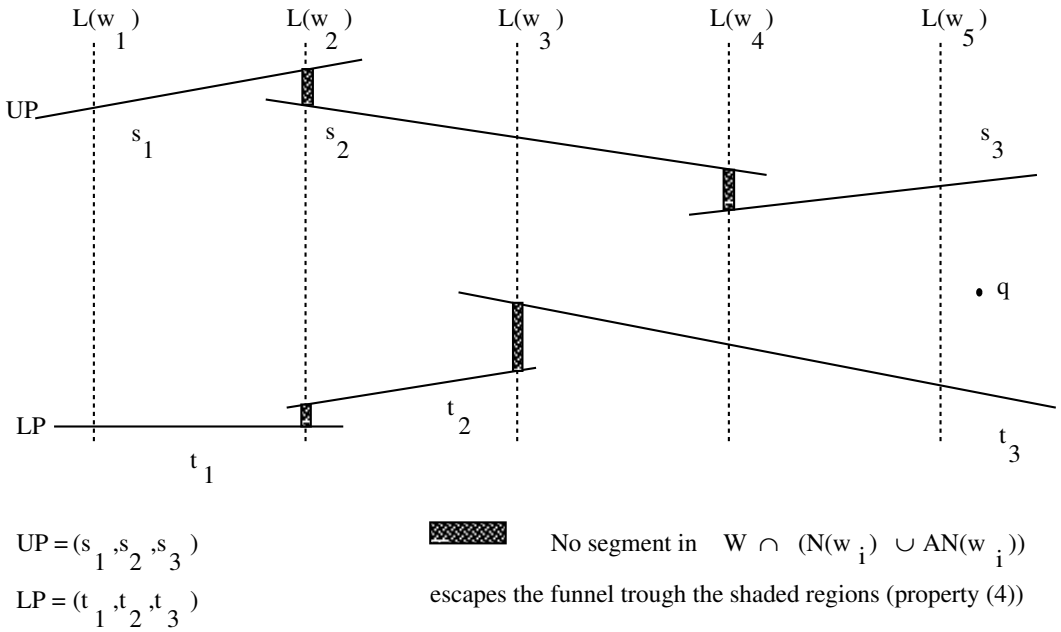

Figure 6: A Funnel

Lemma 21 $\overline{s}_Z,\ \overline{t}_Z \in N(w_i) \cup AN(w_i)$

Proof: Consider any vertex $z \in Z_i$. We show $s(z) \in N(w_i) \cup AN(w_i)$. Note first that $S(z) = AN(parent(z))$ (Invariant 5) and that either $s(z) \in W$ or $s(z)$ is member of a nw-connection (Invariant 7). If $s(z) \in W$ then $s(z) \in N(w_i) \cup AN(w_i)$ (Invariant 5). If $s(z)$ is member of a nw-connection then $s(z) \in L(u, z)$ where u is the left child of w_i. In particular, $s(z) \in S(u)$ and hence $s(z) \in AN(w_i)$ (Invariant 5). □

Let $\widetilde{s}_N$ be the lowest segment above q in the union of the blocks of N containing $\overline{s}_N$, $\overline{t}_N$, $\overline{s}_Z$ (if in $N(w_i)$), and $\overline{t}_Z$ (if in $N(w_i)$) and let $\widetilde{s}_{AN}$ be the lowest segment above q in the union of the blocks of AN containing $\overline{s}_{AN}, \overline{t}_{AN}, \overline{s}_Z$ (if in $AN(w_i)$), and $\overline{t}_Z$ (if in $AN(w_i)$). Finally, let $\widetilde{s}$ be the lowest among the segments $\widetilde{s}_{AN}$, and $\widetilde{s}_N$ and let $l \geq 1$ be minimal such that $\widetilde{s}$ intersects $L(w_{j_l})$ above s_l ($l = r + 1$ if $\widetilde{s}$ intersects $L(q)$ below s_r). Let $UP_i = (\widetilde{s}, s_l, s_{l+1}, \ldots, s_r)$. The lower path LP_i is defined analogously.

Lemma 22 *Properties (1) to (4) of the funnel are maintained.*

Proof: For properties (1), (2) and (3) this follows from Lemma 21 and the construction of the funnel. We now verify property (4). For the sake of a contradiction assume the existence of a segment $s \in W \cap (N(w_i) \cup AN(w_i))$ which extends all the way to $L(q)$ and which intersects $L(w_i)$ in the interior of the funnel. We now distinguish two cases.

Case 1: $s \in S(z)$ for some $z \in Z_i$. By definition of F_i, UP_i is not above $s(z)$ and LP_i is not below $t(z)$ in the vertical strip between $L(w_i)$ and $L(q)$. Also, $s(z)$ and $t(z)$ are adjacent segments in $S(z)$. This contradicts the assumption about segment s.

Case 2: $s \notin S(z)$ for all $z \in Z_i$. Since s is a winner and extends to $L(q)$ we conclude $s \in AN(w_{i+1})$. If $s \in N(w_i)$ then the construction of the funnel guarantees that s is not contained in the same block of $N(w_i)$ as either $\widetilde{s}_N$ or $\widetilde{t}_N$. The definitions of operations $Find_N_winner_above$ and $Find_N_winner_below$ guarantee that s cannot intersect $L(w_{i+1})$ in the intervals covered

by these operations. Thus s intersects $L(w_{i+1})$ in the interior of the funnel, a contradiction to property (4) in the induction hypothesis. If $s \in AN(w_i)$ then use the same argument with N replaced by AN.

□

Lemma 23 *Let s be the segment in S immediately above q, let $w_i = home(s)$, and let s_1 be the first segment in the upper path of funnel F_i. Then $s = s_1$ and s_1 is the only segment in the upper path of the funnel F_i.*

Proof: Assume otherwise. Funnel properties (1) to (3) imply that s must intersect $L(w_i)$ between s_1 and t_1. Here s_1 and t_1 are the first segments of the upper and lower path respectively. Also, $s \neq t_1$. Let B be the block of $N(w_i)$ containing s. Then $win(B)$ extends at least as far as s and hence extends all the way to $L(q)$. Funnel property (4) now implies that $win(B)$ does not intersect $L(w_i)$ between s_1 and t_1. Thus B must contain either s_1 or t_1. But then block B was inspected when funnel F_i was constructed from F_{i+1}. Thus $\widetilde{s} = s$ and hence $s_1 = s$. Also the upper path of F_i consists of only a single segment. □

Lemma 24 *The query algorithm runs in time $O(\log n \cdot \log\log n)$.*

Proof: In each node it takes time $O(\log a)$ to find $\widetilde{t}$ and $\widetilde{s}$. The update of the funnel takes amortized time $O(1)$ since the funnel can grow by at most one segment for each node of the search path and since it takes time $O(1 + d)$ to discard a terminal part of lenght d of either the upper or the lower path. □

Putting everything together we obtain

Theorem 25 *The data structure of this section supports insertions and point location queries in time $O(\log n \cdot \log\log n)$ and deletions in time $O(\log^2 n)$. The time bounds for updates are amortized. The space requirement is $O(n)$.*

4 Conclusions

We close with two open problems. Is there a solution with logarithmic query and update time? Can the data structure be generalized to intersecting line segments? Note that the solution presented here does not even detect when a segment to be inserted intersects segments already in the structure.

References

[A] Alt, H.: "Comparing the Combinatorial Complexities of Arithmetic Functions", *Journal of the ACM*, 35, 1988, pp. 447-460.

[B] Brent, R.P.: "Fast Multiple-Precision Evaluation of Elementary Functions", *Journal of the ACM*, 23, 1976, pp. 242-251.

[M] Mordell, L.J.: "On the Linear Independence of Algebraic Numbers", *Pacific Journal of Mathematics*, 3, 1953, pp. 625-630.

[Sc] Schönhage, A.:"Schnelle Berechnung von Kettenbruchentwicklungen", *Acta Informatica*, 1, 1971, pp. 139-144.

[ScSt] Schönhage, A., Strassen, V.: "Schnelle Multiplikation großer Zahlen", *Computing*, 7, 1971, pp. 281-292.

[S] Siegel, C.L.: "Algebraische Abhängigkeit von Wurzeln", *Acta Arithmetica*, 21, 1972, pp. 59-64.

æ