External Memory Planar Point Location with Logarithmic Updates

Lars Arge[*]
MADALGO[†]
Dept. of Computer Science
University of Aarhus
IT Parken, Aabogade 34
8200 Aarhus N, Denmark
large@madalgo.au.dk

Gerth Stølting Brodal
MADALGO[†]
Dept. of Computer Science
University of Aarhus
IT Parken, Aabogade 34
8200 Aarhus N, Denmark
gerth@madalgo.au.dk

S. Srinivasa Rao
MADALGO[†]
Dept. of Computer Science
University of Aarhus
IT Parken, Aabogade 34
8200 Aarhus N, Denmark
ssrao@madalgo.au.dk

ABSTRACT

Point location is an extremely well-studied problem both in internal memory models and recently also in the external memory model. In this paper, we present an I/O-efficient dynamic data structure for point location in general planar subdivisions. Our structure uses linear space to store a subdivision with N segments. Insertions and deletions of segments can be performed in amortized $O(\log_B N)$ I/Os and queries can be answered in $O(\log_B^2 N)$ I/Os in the worst-case. The previous best known linear space dynamic structure also answers queries in $O(\log_B^2 N)$ I/Os, but only supports insertions in amortized $O(\log_B^2 N)$ I/Os. Our structure is also considerably simpler than previous structures.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations

General Terms

Algorithms, Design

Keywords

External memory, planar subdivisions, point location, dynamic data structure

[*]Supported in part by the US Army Research Office through grant W911NF-04-01-0278, by an Ole Roemer Scholarship from the Danish National Science Research Council, a NABIIT grant from the Danish Strategic Research Council, and by the Danish National Research Foundation.
[†]Center for Massive Data Algorithmics - a Center of the Danish National Research Foundation

1. INTRODUCTION

Planar point location is a classical problem in computational geometry: 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 problem consists of preprocessing Π into a data structure so that the face of Π containing an arbitrary query point p can be reported quickly. This problem has applications in e.g. graphics, spatial databases, and geographic information systems. The planar subdivisions arising in many applications in these areas are too massive to fit in internal memory and must reside on disk. In such instances the I/O communication, rather than the CPU running time, is the bottleneck. Most work on planar point location, especially if we allow the edges and vertices of Π to be changed dynamically, has focused on minimizing the CPU running time under the assumption that the subdivision fits in main memory (e.g., [3, 9, 11, 12, 17, 20, 22]). Only a few results are known for I/O-efficient dynamic point location when the subdivision is stored in external memory [1, 6]. In this paper, we improve the update bound of the previous best known dynamic structure.

1.1 Previous results

In internal memory, Edelsbrunner *et al.* [16] proposed an optimal static data structure for point location in planar monotone subdivision, i.e., subdivisions where the intersection of any face and any vertical line is a (possibly empty) single interval. Their data structure uses $O(N)$ space, can be constructed in $O(N)$ time, and answers queries in $O(\log_2 N)$ time. For arbitrary planar subdivisions, linear space structures with logarithmic query time and $O(N \log_2 N)$ construction time are known; see, e.g., [20, 22]. For the dynamic version of the problem where we allow the edges and vertices to be changed dynamically, Cheng and Janardan [11] gave a structure that answers queries in $O(\log_2^2 N)$ time and supports updates in $O(\log_2 N)$ time. The structure given by Baumgarten *et al.* [9] supports queries in $O((\log_2 N) \log_2 \log_2 N)$ time worst case, insertions in $O((\log_2 N) \log_2 \log_2 N)$ time amortized, and deletions in $O(\log_2^2 N)$ time amortized. Recently, Arge *et al.* [3] gave a structure that supports queries in $O(\log_2 N)$ time worst case, insertions in $O(\log_2^{1+\varepsilon} N)$ time amortized, and deletions in $O(\log_2^{2+\varepsilon} N)$ time amortized, for some arbitrary fixed constant $0 < \varepsilon < 1$. All three structures use linear space. They all basically store the edges of the subdivision in an interval tree [15] constructed on their x-projection (as first suggested in [17]) and use this structure to answer *vertical ray-shooting queries*, that is, the problem of finding the first edge of Π hit by a ray emanating in the $(+y)$-direction from a query point p. After answering a vertical ray-shooting query the face containing p can be easily found in $O(\log_2 N)$ time [21].

In this paper, we are interested in the problem of dynamically maintaining a planar subdivision on disk, such that the number of I/O operations (or *I/Os*) used to perform a query or an update is minimized. We consider the problem in the standard two-level I/O model proposed by Aggarwal and Vitter [2]. In this model, M is the number of elements (vertices/edges) that fit in the internal memory and B is the number of elements per disk block, where $2 \leq B \leq M/2$. An I/O is the operation of reading (or writing) a block from (or into) external memory. Computation can only be performed on elements in internal memory. The measures of performance are the number of I/Os used to solve a problem and the amount of space (disk blocks) used.

In the I/O-model, Goodrich *et al.* [18] designed a linear space ($O(N/B)$ disk blocks) static data structure to store a planar monotone subdivision so that a query can be answered in optimal $O(\log_B N)$ I/Os. Arge *et al.* [4] designed a structure for general subdivisions with the same bounds. Goodrich *et al.* [18] also developed a structure for answering a batch of K queries in $O(\frac{1}{B}(N + K)\log_{M/B} N)$ I/Os. Arge *et al.* [7] extended the batched result to general subdivisions (see also [14]), and Arge *et al.* [5] to an off-line dynamic setting where a sequence of queries and updates are given and all the queries should be answered as the sequence of operations is performed. Vahrenhold and Hinrichs [23] considered the problem under some practical assumptions about the input data. Only two results are known for the dynamic case. Agarwal *et al.* [1] designed a linear space structure for planar monotone subdivisions that supports queries in $O(\log_B^2 N)$ I/Os in the worst case and updates in $O(\log_B^2 N)$ I/Os amortized. Arge and Vahrenhold [6] designed the only previously known dynamic structure for general subdivisions. The structure uses linear space and supports queries in $O(\log_B^2 N)$ I/Os in the worst case, and insertions and deletions in $O(\log_B^2 N)$ and $O(\log_B N)$ I/Os amortized, respectively.

1.2 Our results

In this paper we describe a linear space dynamic structure for point location in general planar subdivisions. The structure supports queries in $O(\log_B^2 N)$ I/Os like the previously known structure [6] but can be updated in $O(\log_B N)$ I/Os amortized. Our structure is also considerably simpler.

Our main contribution is a structure (called a *multislab structure*) for dynamically maintaining a set of segments with endpoints on B^ε, $0 < \varepsilon \leq 1$, vertical lines, such that the segment immediately above a query point can be found in $O(\log_B N)$ I/Os and such that segments can be inserted and deleted in $O(\log_B N)$ I/Os amortized. Such a structure was also used in the previous I/O-efficient dynamic point location structure [6]. However, the previous structure only supported insertions in $O(\log_B^2 N)$ I/Os amortized. Using the new multislab structure our point location structure is obtained with essentially the same method as the previous structures [1, 6], namely by using the multislab structure as secondary structures in an interval tree over the segments projection on the x-axis.

The rest of the paper is organized as follows. In the next section we outline the overall structure of our point location data structure (similar to the previous structures [1, 6]). In Section 3, we then describe our new multislab structure. In these sections we assume for simplicity that the base interval tree is static. In Section 4 we then describe how to rebalance the base interval tree during updates (and how to handle the resulting reorganization of the secondary structures). We give conclusions in Section 5.

2. OVERALL STRUCTURE

In the following, we will concentrate on developing a structure for dynamic *vertical ray-shooting*: Maintain the segments $\mathcal{S}$ in Π such that the first segment hit by a ray emanating from a query point in the $(+y)$-direction can be found efficiently. It is easy to realize that a point location query with p can be easily answered in an additional $O(\log_B N)$ I/Os once a vertical ray shooting query with p has been answered [6].

We will make frequent use of (a, b)-trees [19]. In (a, b)-trees objects are stored in the leaves. All leaves are on the same level of the tree, and all internal nodes have between a and b children, except possibly the root which has between 2 and b children. In this paper, all (a, b)-trees will satisfy that $a, b \in \Theta(B^\varepsilon)$ for some constant $0 < \varepsilon \leq 1$ and each leaf stores $\Theta(B)$ objects. This way each node can be stored in $O(1)$ blocks, a tree storing N objects has height $O(\log_{B^\varepsilon} N) = O(\log_B N)$, and it uses linear space. We refer to an (a, b)-tree with $a = cB^\varepsilon$ and $b = B^\varepsilon$, for some $0 < c \leq 1/2$ as a B^ε-tree. (A normal B-tree [10, 13], or rather B$^+$-tree, is such a structure with $\varepsilon = 1$.) Since the tree has height $O(\log_B N)$ and each node is stored in $O(1)$ blocks, a search can be performed in $O(\log_B N)$ I/Os. Insertions and deletions can also be performed in $O(\log_B N)$ I/Os using $O(\log_B N)$ *split* and *fuse* operations on the nodes on a root-leaf path [19].

The basic idea in our structure is the same as previously applied by Agarwal *et al.* [1] and Arge and Vahrenhold [6], and is similar to the one used in several main memory structures [9, 11, 17]. The set of edges/segments $\mathcal{S}$ of Π is stored in a two-level tree structure, with the first level being an interval tree—here an external interval tree [8]—on their x-projection: the *base (interval) tree* is a B^ε-tree $\mathcal{T}$ over the x-coordinates of the endpoints of segments in $\mathcal{S}$ (a possible value for ε is $\varepsilon = 1/5$); the segments in $\mathcal{S}$ are stored in secondary structures associated with the nodes of $\mathcal{T}$, such that each segment is stored at precisely one node of $\mathcal{T}$. Each node v of $\mathcal{T}$ is associated with a vertical *slab* s_v; the root is associated with the whole plane. For each internal node v, the slab s_v is partitioned into B^ε vertical slabs $s_1, \ldots, s_{B^\varepsilon}$ corresponding to the children of v, separated by vertical lines called *slab boundaries* (the dashed lines in Figure 1(a)). A segment t of $\mathcal{S}$ is stored in the secondary structures associated with the highest node v of $\mathcal{T}$ where it intersects a slab boundary.

In the following we for simplicity assume that the endpoints of the segments in $\mathcal{S}$ have distinct x-coordinates, such that each leaf stores $O(B)$ segments. The segments associated with a leaf are simply stored in $O(1)$ blocks. Let v be an internal node of $\mathcal{T}$ and let $\mathcal{S}_v \subseteq \mathcal{S}$ be the set of segments associated with v. Let $t \in \mathcal{S}_v$ be one of the segments associated with v, and suppose that the left endpoint of t lies in the slab s_ℓ and the right endpoint of t lies in the

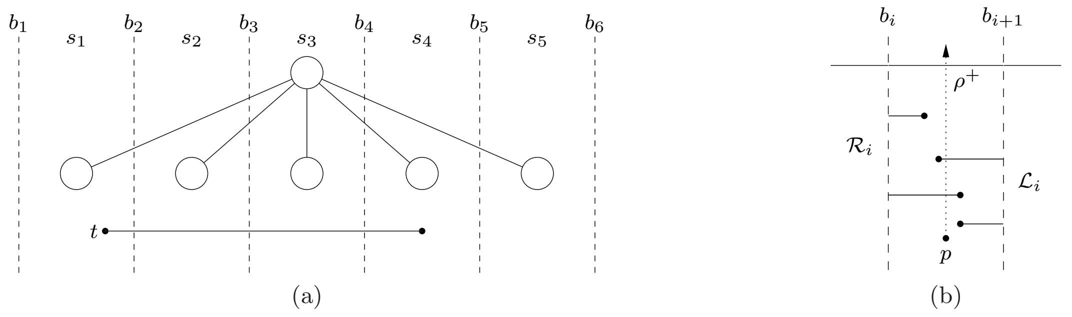

Figure 1: (a) A node in the base tree $\mathcal{T}$. The left subsegment of t is in slab s_1, the right subsegment in slab s_4, and the middle subsegment of t spans s_2 and s_3. (b) Answering a query.

slab s_r associated with the ℓ-th and r-th children of v, respectively. We call the subsegment $t \cap s_\ell$ the *left* subsegment of t, $t \cap s_r$ the *right* subsegment, and the portion of t lying in $s_{\ell+1}, \ldots, s_{r-1}$ the *middle* subsegment. Refer to Figure 1(a). Let $\mathcal{M}$ denote the set of middle subsegments of segments in $\mathcal{S}_v$. For each i, $1 \le i \le B^\varepsilon$, let $\mathcal{L}_i$ (resp., $\mathcal{R}_i$) denote the set of left (resp., right) subsegments that lie in s_i. We store the following secondary structures at v.

(i) A *multislab structure* Δ on the set of middle segments $\mathcal{M}$.

(ii) For each i, $1 \le i \le B^\varepsilon$,

- a *left structure* on all segments of $\mathcal{L}_i$, and
- a *right structure* on all segments of $\mathcal{R}_i$.

A segment in $\mathcal{S}_v$ is thus stored in at most three secondary structures: the multislab structure, a left structure, and a right structure. For example, the segment t in Figure 1(a) is stored in the multislab structure Δ, the left structure of s_1, and in the right structure of s_4. The secondary structures are constructed to use linear space so that each internal node v requires $O(|\mathcal{S}_v|/B)$ disk blocks. This in turn means that overall the data structure requires $O(N/B)$ disk blocks.

Let ρ^+ be the ray emanating from a point p in the $(+y)$-direction. To find the first segment of $\mathcal{S}$ hit by ρ^+, we search $\mathcal{T}$ along a path of length $O(\log_B N)$ from the root to the leaf z where s_z contains p. At each internal node v visited, we compute the first segment of $\mathcal{S}_v$ hit by ρ^+. In particular, in v we first search Δ to find the first segment of $\mathcal{M}$ hit by ρ^+. Next, we find the vertical slab s_i that contains p and search the left and right structures for s_i to find the first segments of $\mathcal{L}_i$ and $\mathcal{R}_i$, respectively, hit by ρ^+. Refer to Figure 1(b). At the leaf z, the first segment of $\mathcal{S}_z$ hit by ρ^+ is computed by testing all segments of $\mathcal{S}_z$ explicitly. The query is then answered by choosing the lowest segment among the $O(\log_B N)$ segments found this way.

Based on ideas due to Cheng and Jarnadan [11], Agarwal *et al.* [1] showed how the left and right structures can be implemented efficiently, basically using B-trees:

LEMMA 1 (AGARWAL *et al.* [1, LEMMA 3]). *A set of K non-intersecting segments all of whose right (left) endpoints lie on a single vertical line can be stored in a linear space data structure such that a vertical ray-shooting query can be answered in $O(\log_B K)$ I/Os. Updates can be performed in $O(\log_B K)$ I/Os. If the set is sorted by the right (left) y-coordinates of the endpoints, then the structure can be constructed in $O(K/B)$ I/Os.*

In the next section, we show how the multislab structure can be implemented efficiently:

LEMMA 2. *For a constant $0 < \varepsilon \le 1/5$, a set of K non-intersecting segments with endpoints on $B^\varepsilon + 1$ vertical lines can be stored in a linear space data structure such that a vertical ray-shooting query can be answered in $O(\log_B K)$ I/Os. Updates can be performed in amortized $O(\log_B K)$ I/Os.*

Lemma 1 and Lemma 2 imply that our overall structure can answer queries in $O(\log_B^2 N)$ I/Os since we use $O(\log_B N)$ I/Os to query the secondary structures at each of the node on a root-leaf path of $\mathcal{T}$ of length $O(\log_B N)$. Ignoring updates in the base tree $\mathcal{T}$ (insertion/deletion of endpoints and rebalancing), updates can be performed in $O(\log_B N)$ I/Os simply by searching down a root-to-leaf path of $\mathcal{T}$ to find the relevant node v and then updating the relevant left and right structures and the multislab structures associated with v. In Section 4 we discuss how the base tree can also be updated in amortized $O(\log_B N)$ I/Os. This way we obtain our main result.

THEOREM 1. *A set $\mathcal{S}$ of N non-intersecting segments in the plane can be stored in a linear space data structure, such that a vertical ray-shooting query can be answered in $O(\log_B^2 N)$ I/Os, and such that updates can be performed in amortized $O(\log_B N)$ I/Os.*

3. MULTISLAB STRUCTURE

Let $\mathcal{M}$ be a set of K non-intersecting segments in the plane with endpoints on $B^\varepsilon + 1$ vertical lines $b_1, \ldots, b_{B^\varepsilon+1}$. For each i, $1 \le i \le B^\varepsilon$, let s_i be the vertical slab bounded by b_i and b_{i+1}. In this section we consider the problem of maintaining $\mathcal{M}$ in a structure using $O(K/B)$ disk blocks that supports vertical ray shooting queries in $O(\log_B K)$ I/Os and updates in amortized $O(\log_B K)$ I/Os.

Our data structure will actually consist of two different data structures for the two cases: (i) $B^{2\varepsilon} = O(\log_B K)$, and (ii) $\log_B K = O(B^{2\varepsilon})$, for some constant ε, $0 < \varepsilon < 1/5$. Intuitively, in Case (i), discussed in Section 3.2, we maintain sorted lists of segments for every pair of slab boundaries. Since the number of these lists is bounded by $B^{2\varepsilon} = O(\log_B K)$, we can support queries efficiently. Intuitively, in Case (ii), we use the logarithmic method [6] to reduce the problem to $O(\log_B K)$ deletion-only problems, and show how to support deletions efficiently using the fact that the number of structures created by the logarithmic method is

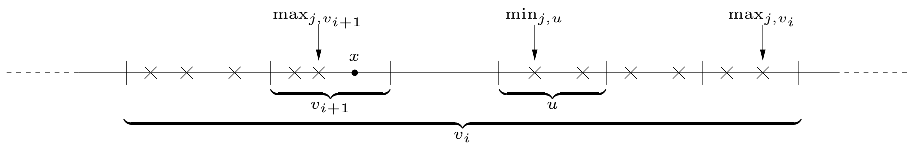

Figure 2: The search for the successor of x in L_j. Crosses indicate elements from L_j. Underbraces show the intervals spanned by the nodes in the B^c-tree.

bounded by $O(\log_B K) = O(B^{2\varepsilon})$. For this case, we first describe a deletion-only structure in Section 3.3, and then we describe how to use the logarithmic method to obtain a fully-dynamic structure in Section 3.4. In both cases we utilize a structure that supports efficient simultaneous searching in $O(B^{1-c})$ sorted lists of total size K, for some positive constant $0 < c < 1$, in $O(\log_B K)$ I/Os. This structure is described in Section 3.1.

In fact, we will implement all multislab structures at the nodes of the base tree $\mathcal{T}$ by either the data structure of Section 3.2 when $B^{2\varepsilon} = O(\log_B N)$, or by the data structure of Section 3.4 when $\log_B N = O(B^{2\varepsilon})$. Here N is the total number of segments stored. This increases the I/O bound for queries to a multislab structure storing K segments from $O(\log_B K)$ to $O(\log_B N)$ in the first case (see Lemma 4)—all other I/O bounds remain unchanged. In order to ensure that we always implement the multislab structures using the correct case (Section 3.2 or Section 3.4), we rebuild all multislab structures whenever $N/2$ operations have been performed to the structure. This only adds amortized $O(\log_B N)$ I/Os to the update bounds.

3.1 Searching in multiple sorted lists

In this section we describe a structure supporting efficient simultaneous searching in m sorted lists $L_1, L_2, \ldots, L_m$ of total size K, assuming $m = O(B^{1-c})$ for some constant c, $0 < c < 1$. We assume that the elements in the lists come from a total order and the aim is to maintain the lists under insertions and deletions such that searching in all the lists simultaneously is supported in $O(\log_B K)$ I/Os. The structure is based on an idea utilized in [6] to search in a single deletion-only multislab structure.

Consider the sorted list $L = \cup_{1\le j\le m} L_j$ stored in $O(K/B)$ blocks, where each element is augmented with information about which list L_j it belongs to. Our structure consists of a B^c-tree over the list L. For an internal node u of the B^c-tree, let $\min_{j,u}$ and $\max_{j,u}$ be the minimum and maximum elements that belong to the list L_j and are stored in the subtree rooted at u; if the subtree does not contain any element from L_j we let $\min_{j,u} = \infty$ and $\max_{j,u} = -\infty$. For each node v in the B^c-tree, we store for each child u of v, and for each list L_j, the elements $\min_{j,u}$ and $\max_{j,u}$. Since $B^c \cdot m \cdot 2 = O(B)$ these elements can be stored in $O(1)$ blocks associated with v, i.e., the tree can be stored in $O(K/B)$ blocks.

Now to find for each list L_j, $1 \le j \le m$, the successor in L_j of a query element x, we search for x along a path $v_1, v_2, \ldots v_h$ from the root to a leaf of the augmented B^c-tree on L. If $x > \max_{j,v_1}$, we know that $x > \max L_j$ and we return ∞ for the list L_j. Otherwise, we find the successor of x in L_j in the node v_i where $\max_{j,v_{i+1}} < x \le \max_{j,v_i}$ simply by returning the successor of x among all $\min_{j,u}$ stored at v_i (where u ranges over the children of v_i). Refer to Figure 2. If no such node exists, i.e., we reach the leaf v_h without having answered the query in L_j, we have $x \le \max_{j,v_h}$, and we simply return the successor of x among the elements from L_j stored in v_h. It is easy to verify that this correctly finds the successor of x in L_j. Since the successors of x in all the lists L_j, $1 \le j \le m$, can be found in one traversal of the root-to-leaf path the query is answered in $O(\log_B K)$ I/Os.

Updates are performed in a straightforward manner similar to a B-tree. To insert/delete an element x into/from a list L_j, we first search for x in the B^c-tree over L to find the leaf block containing x and perform the update. Then we traverse the path back to the root while updating the $\min_{j,v}$ and $\max_{j,v}$ values. This takes $O(\log_B K)$ I/Os since $\min_{j,v}$ and $\max_{j,v}$ can be updated in $O(1)$ I/Os in a node v. Finally, we rebalance the tree as in the case of a standard B-tree update, i.e., we split and fuse nodes on the root-leaf path as required. As the $\min_{j,v}$ and $\max_{j,v}$ values can also easily be maintained in $O(1)$ I/Os during a split or fuse, the rebalancing is performed in $O(\log_B K)$ I/Os.

Our data structure can be constructed from a set of sorted lists $L_1, L_2, \ldots, L_m$ by first forming the list $L = \cup_{1\le j\le m} L_j$ by pairwise merging the L_j lists in a binary tree fashion using $O((K/B)\log_2 m)$ I/Os, and then constructing the B^c-tree over L bottom-up using $O(K/B)$ I/Os. In total the construction requires $O((K/B)\log_2 B)$ I/Os, since $m = O(B^{1-c})$.

LEMMA 3. *Let $L_1, L_2, \ldots, L_m$ be $m = O(B^{1-c})$ sorted lists, for some constant $0 < c < 1$, of total size K whose elements come from a total order. There exists a linear space data structure that supports the simultaneous search for the successor of a query element x in each of the m lists in $O(\log_B K)$ I/Os, and supports the insertion and deletion of an element from a list L_j in $O(\log_B K)$ I/Os. The data structure cane be constructed in $O((K/B)\log_2 B)$ I/Os, provided each L_i is already sorted.*

3.2 Case (i) $B^{2\varepsilon} = O(\log_B K)$

In this section we describe a structure for maintaining a set $\mathcal{M}$ of K non-intersecting segments in the plane with endpoints on $B^\varepsilon+1$ vertical lines, for a constant $0 < \varepsilon < 1/2$, such that vertical ray-shooting queries can be answered I/O-efficiently when $B^{2\varepsilon} = O(\log_B K)$. We will assume without loss of generality that $B^\varepsilon \le B/4$.

We first partition the segments in $\mathcal{M}$ into $O(B^{2\varepsilon})$ *multislab lists* $L_{\ell,r}$, for $1 \le \ell < r \le B^\varepsilon+1$, such that $L_{\ell,r}$ contains all the segments of $\mathcal{M}$ whose left and right endpoints are on the vertical lines b_ℓ and b_r, respectively. All the segments in each multislab list $L_{\ell,r}$ are sorted in increasing order with respect to their left endpoint and stored in order (in a B-tree) on disk. We will maintain the blocks of each $L_{\ell,r}$ such that they contain between $B/4$ and B segments. We call a block *full* if it contains B segments and *sparse* if it contains $B/4$ segments.

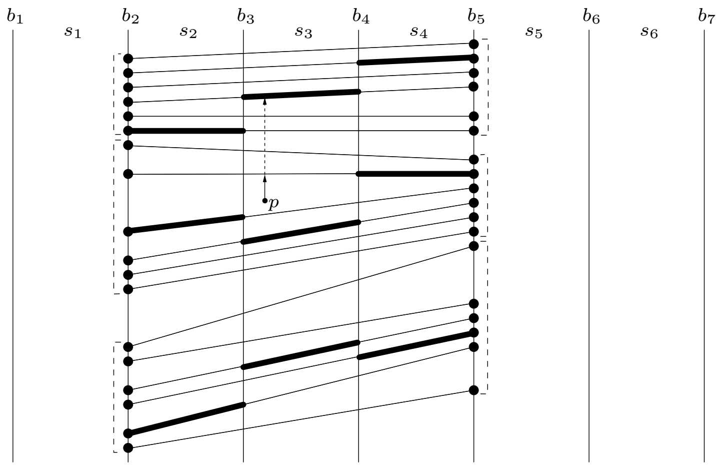

Figure 3: The multislab list $L_{2,5}$. The dashed lines show the partition of the segments into blocks. The sampled segments for the slabs s_2, s_3 and s_5 are emphasized with thick lines.

For each slab s_k and pair of vertical lines b_ℓ and b_r, where $1 \leq \ell \leq k < r \leq B^\varepsilon + 1$, let $L^k_{\ell,r}$ be the list obtained by picking one (arbitrary) segment from each block of $L_{\ell,r}$. We call $L^k_{\ell,r}$ the *sampling list* of $L_{\ell,r}$ for slab s_k. If $|L_{\ell,r}| < B$, we define $L^k_{\ell,r}$ to be empty. We maintain pointers between the same segments in the multislab and sampling lists, and will maintain the sampling lists $L^k_{\ell,r}$ to be disjoint under insertions and deletions of segments in the multislab list $L_{\ell,r}$, i.e., a segment in $L_{\ell,r}$ can be sampled for at most one of the slabs $s_\ell, \ldots, s_{r-1}$. Refer to Figure 3. To do so we label a segment in $L_{\ell,r}$ with k if it is sampled in s_k; since each multislab list $L_{\ell,r}$ can at most span $B^\varepsilon \leq B/4$ slabs and each block of $L_{\ell,r}$ contains at least $B/4$ segments, there will always be at least one unmarked segment in each block. Finally, we for each slab s_k construct a structure for simultaneously searching in the $O(B^{2\varepsilon})$ sampling lists $L^k_{\ell,r}$; since we have a total order on the segments in the $L^k_{\ell,r}$ lists (the order of the intersections between the segments and b_ℓ), and by choosing $c = 1-2\varepsilon$, we can utilize the linear space structure of Lemma 3 for this.

Now to answer a vertical ray-shooting query we first find the slab s_k containing the query point p. Since there are B^ε slabs, we can easily do so in $O(B^\varepsilon)$ I/Os. Next we use the structure of Lemma 3 to answer the query in each of the sample lists $L^k_{\ell,r}$ in $O(\log_B K)$ I/Os. Finally, we for each of the $O(B^{2\varepsilon})$ multislab lists $L_{\ell,r}$, $1 \leq \ell < r \leq B^\varepsilon + 1$, in turn use the result x of the query in $L^k_{\ell,r}$ to answer the query in $L_{\ell,r}$. We can do so in $O(1)$ I/Os since the answer is either in the block of $L_{\ell,r}$ containing x or in its predecessor block; in the case where $|L_{\ell,r}| < B$ (such that $L^k_{\ell,r}$ is empty) we simply search directly in $L_{\ell,r}$ using $O(1)$ I/Os. Refer to Figure 3. Overall we answer the ray-shooting query in $O(B^{2\varepsilon} + \log_B K)$ I/Os.

To insert a segment t spanning slabs s_ℓ through s_r, we first insert t is the multislab list $L_{\ell,r}$ using $O(\log_B K)$ I/Os. If the block containing t is now full we split it into two blocks, each containing roughly half the segments from the original block. This creates the need for updating the pointers between at most B^ε existing samples and the sampling of at most B^ε new segments to be inserted in the $L^k_{\ell,r}$ sample lists, $\ell \leq k < r$, and thus in the corresponding Lemma 3 structures. Since each insertion (an update of pointers between the same segments in multislab and sample lists) takes $O(\log_B K)$ I/Os, the total cost of the split is $O(B^\varepsilon \cdot \log_B K)$ I/Os. Amortizing this cost over the $\Theta(B)$ insertions in the full block since its creation, the split is handled in amortized $O(\log_B K)$ I/Os. Deletions are handled in a similar way, by fusing sparse blocks with an adjacent block (and splitting again if necessary), as in the case of standard B-tree operations (to guarantee that the resulting blocks contain between $\frac{3}{8}B$ and $\frac{3}{4}B$ segments). Furthermore, when a sampled segment for slab s_k is deleted from $L_{\ell,r}$, we replace it with another unsampled segment in the same block of $L_{\ell,r}$. This requires a deletion and an insertion on a Lemma 3 structure. In total deletes are also handled in amortized $O(\log_B K)$ I/Os.

Our structure can be constructed from a set of sorted multislab lists $L_{\ell,r}$, $1 \leq \ell < r \leq B^\varepsilon + 1$, as follows: We first scan the $O(B^{2\varepsilon})$ multislab lists and construct the corresponding sorted sampling lists $L^k_{\ell,r}$ using $O(K/B)$ I/Os. These lists contain a total of $O((K/B)B^\varepsilon)$ samples. We then apply the construction algorithm of Lemma 3 for the $O(B^\varepsilon)$ simultaneous searching structures (one for each slab) using a total of $O(\frac{1}{B}(K/B)B^\varepsilon \log_2 B) = O(K/B)$ I/Os. In total the construction of the data structure requires $O(K/B)$ I/Os.

LEMMA 4. *For a constant $0 < \varepsilon < 1/2$, a set of K non-intersecting segments with endpoints on $B^\varepsilon + 1$ vertical lines can be stored in a linear space data structure such that a vertical ray-shooting query can be answered in $O(B^{2\varepsilon} + \log_B K)$ I/Os. Updates can be performed in amortized $O(\log_B K)$ I/Os. The data structure can be constructed in $O(K/B)$ I/Os, provided the K segments are given in $O(B^{2\varepsilon})$ sorted multislab lists.*

Note that when $B^{2\varepsilon} = O(\log_B K)$ the query bound in Lemma 4 is $O(\log_B K)$ I/Os.

3.3 Deletion-only structure

In this section we describe a structure that supports I/O-efficient ray-shooting queries and deletions on a set $\mathcal{M}$ of K segments in the plane with endpoints on $B^\varepsilon + 1$ vertical lines, where $0 < \varepsilon \leq 1$. In Section 3.4 we will use this

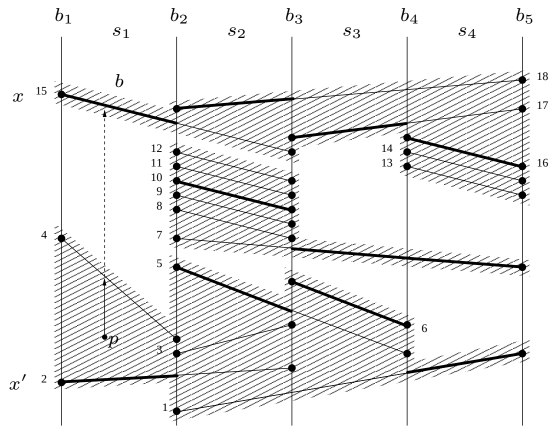

Figure 4: Deletion-only structure. The numbers next to the segments are the ranks of the segments in the sorted list L. The shaded areas show the blocking of L into 3 blocks. The thick lines are samples for the slabs. Note that segment 7 is sampled twice.

structure to develop a fully-dynamic structure for the case where $\log_B K = O(B^{2\varepsilon})$.

Our structure utilizes a partial order on non-intersecting segments in the plane [6].

DEFINITION 1. *A segment x in the plane is* above *a segment y in the plane, $y \prec x$, if there exists a vertical line λ intersecting both x and y such that the intersection between λ and x is above the intersection between λ and y.*

Two segments are incomparable if they cannot be intersected by the same vertical line. The *segment sorting problem* is the problem of extending the partial order $\prec$ to a total order. Arge *et al.* [7, Lemma 3] showed how to solve the segment sorting problem on K segments in $O(\frac{K}{B} \log_{M/B} \frac{K}{B})$ I/Os.

Let L be the sorted list of segments in $\mathcal{M}$ stored in a list of blocks. The sorted order remains valid under deletions of segments from $\mathcal{M}$. In fact, we will let the blocking of L remain unchanged during deletions of segments, that is, segments are simply deleted from the blocks storing L. For each slab s_k we also generate a list $L^k \subseteq L$ of segments crossing slab s_k, by sampling exactly one segment crossing slab s_k from each block b of L (if existing). If at most B^ε segments in b cross the slab s_k we pick the sample arbitrarily. Note that this way a segment can be sampled for several slabs (as segment 7 in Figure 4). If more than B^ε segments in b cross the slab s_k, we make sure to pick a sample that is not used as a sample for any other slab. Each L^k list is represented by a B-tree and we store pointers between the same segments in L^k and L. Our structure uses linear space, since the total size of the L^k lists is $O(\frac{K}{B}B^\varepsilon)$. Using the I/O-efficient segment sorting algorithm [7] it is easy to construct the structure in $O(\frac{K}{B} \log_{M/B} \frac{K}{B})$ I/Os.

To perform a vertical ray-shooting query we first find the slab s_k containing the query point p using $O(\log_B K)$ I/Os. Next we use the B-tree on L^k to find the answer x to the query with respect to the segments in L^k in $O(\log_B K)$ I/Os. Finally, we answer the query with respect to the segments in L in an additional $O(1)$ I/Os simply by inspecting the segments in the blocks of L containing x and the predecessor of x in L^k. Refer to Figure 4. In total we answer the query in $O(\log_B K)$ I/Os.

To delete a segment x in block b of L we simply delete x from b and from each list L^k where x appears as a sample, while replacing x in L^k with a new sample segment from b crossing s_k (if existing). If at most B^ε segments in b cross s_k we pick the sample arbitrarily; otherwise we make sure to pick a segment that is not used as a sample for any other slab.

We now analyze the cost of performing d deletions on our structure. To bound the total number of updates to the L^k lists during the deletions we need the following two observations: 1) a sampled segment in a list L^k remains sampled until the segment is deleted from the structure; 2) if a segment x in a block b is used as a sample for several slabs, at most one slab s_k can have more than B^ε segments in b that span slab s_k. The latter follows from the fact that we never sample a segment for a slab if the segment is already a sample for another slab and there are more than B^ε alternative segments to sample from. A sampled segment x from block b for slab s_k is now denoted a *sparse sample* if at most B^ε segments in block b span slab s_k. By the second observation above, a sequence of d deletions can at most delete d samples that are not sparse samples. Furthermore, during the sequence of deletions each of the $O(K/B)$ blocks can at most have B^ε distinct sparse samples for each of the B^ε slabs. It follows that the total number of samples that need to be updated during a sequence of d deletes is $O(d + \frac{K}{B}B^\varepsilon B^\varepsilon)$. Since each update to an L^k list requires an update to the corresponding B-tree using $O(\log_B K)$ I/Os, the d deletions require a total of $O((d + \frac{K}{B}B^{2\varepsilon}) \log_B K)$ I/Os

LEMMA 5. *For a constant $0 < \varepsilon \leq 1$, a set of K non-intersecting segments with endpoints on at most $B^\varepsilon + 1$ vertical lines can be maintained in a linear space data structure under deletion of segments, such that vertical ray-shooting queries can be answered in $O(\log_B K)$ I/Os. The construction of the structure and d deletions take $O(\frac{K}{B} \log_{M/B}(K/B) + (d + \frac{K}{B}B^{2\varepsilon}) \log_B K)$ I/Os in total.*

3.4 Case (ii) $\log_B K = O(B^{2\varepsilon})$

In this section we describe a structure for maintaining a set $\mathcal{M}$ of K non-intersecting segments in the plane with endpoints on $B^\varepsilon + 1$ vertical lines, for $\log_B K = O(B^{2\varepsilon})$ and

$0 < \varepsilon \leq 1/5$, such that vertical ray-shooting queries can be answered I/O-efficiently.

Our structure uses the deletion-only structure described in Section 3.3. For the case $\log_B K = O(B^{2\varepsilon})$ and $0 < \varepsilon \leq 1/5$ we can restate the I/O bounds of Lemma 5 as follows:

LEMMA 6. *For a constant $0 < \varepsilon \leq 1/5$ and $\log_B K = O(B^{2\varepsilon})$, a set of K non-intersecting segments with endpoints on at most $B^\varepsilon + 1$ vertical lines can be maintained in a linear space data structure, such that vertical ray-shooting queries can be answered in $O(\log_B K)$ I/Os and such that deletions can be performed in amortized $O(\log_B K)$ I/Os. The structure can be constructed in amortized $O(K/B^\varepsilon)$ I/Os.*

The (preprocessing) bound of Lemma 6 follows from the following derivation:

$$\begin{aligned} & O\left(\frac{K}{B}\log_{M/B}(K/B) + \frac{K}{B}B^{2\varepsilon}\log_B K\right) \\ = \; & O\left(\frac{K}{B}\log_B K \cdot \log_2 B + \frac{K}{B}B^{4\varepsilon}\right) \\ = \; & O\left(\frac{K}{B}B^{2\varepsilon}\cdot B^{\varepsilon} + \frac{K}{B^{1-4\varepsilon}}\right) \\ = \; & O\left(\frac{K}{B^{1-4\varepsilon}}\right) \\ = \; & O\left(\frac{K}{B^{\varepsilon}}\right) . \end{aligned}$$

To obtain our structure we use the external version of the logarithmic method described [6]. More precisely, we partition $\mathcal{M}$ into $O(\log_B K)$ sets $S_0, S_1, S_2, \ldots$, such that $|S_i| \leq B^{1+i\varepsilon}$, and store each set S_i in the deletion-only structure of Section 3.3. We will ensure that the number of deletion-only structures is always $O(\log_B K)$. However, in order to be able to efficiently answer the same ray-shooting query on all the deletion-only structures (i.e., on the sets S_i) simultaneously, we slightly modify the deletion-only structures. More precisely, for each slab s_k we replace the B-trees on the L^k lists of all $O(\log_B K)$ deletion-only structures with the linear space structure of Lemma 3 (Section 3.1) for simultaneously searching in the $O(\log_B K) = O(B^{2\varepsilon})$ L^k lists; we can do so since we have a total order on the segments in the L^k lists (the order of the intersections between the segments and b_ℓ) and by choosing $c = 1 - 2\varepsilon$. The overall space use of our structure is linear, since each of the deletion-only structures (Lemma 5) and simultaneous searching structures (Lemma 3) use linear space.

To answer a vertical ray-shooting query we first find the slab s_k containing the query point p using $O(\log_B K)$ I/Os. Next we use the simultaneous searching structure for slab s_k to answer the query in the L^k list of each of the deletion-only structures using $O(\log_B K)$ I/Os. Finally, we answer the query on each of the delete-only structures (using $O(1)$ I/Os on each of the $O(\log_B K)$ structures) as described in Section 3.3. Overall we answer a query in $O(\log_B K)$ I/Os.

In order to insert a new segment x we first determine the smallest i such that $\sum_{j=0}^{i}|S_i| < B^{1+i\varepsilon}$. If $i = 0$ we simply insert x in S_0 by rebuilding it completely. This takes $O(1)$ I/Os since $|S_0| \leq B$. Otherwise, we discard the structures $S_0, S_1, S_2, \ldots, S_{i-1}$ and construct a new deletion-only structure of size $I = 1 + \sum_{j=0}^{i}|S_j| \leq B^{1+i\varepsilon}$ for the (new S_i) set $\{x\}\cup\bigcup_{j\leq i} S_j$. The construction of a new deletion-only structure for S_i also requires updating the simultaneous search structure on the L^k lists. More precisely, we need to delete the $O(\frac{I}{B}B^\varepsilon)$ segments in the old L^k lists and insert $O(\frac{I}{B}B^\varepsilon)$ new segments in the new lists. Each such update requires $O(\log_B K)$ I/Os (Lemma 3). By Lemma 6 we then have that in total the insertion of segment x requires amortized

$$O\left(\frac{I}{B^\varepsilon} + \frac{I}{B}B^\varepsilon\log_B K\right) = O\left(\frac{I}{B^\varepsilon}\right)$$

I/Os (since $\log_B K = O(B^{2\varepsilon})$ and $0 < \varepsilon \leq 1/5$). We charge this cost to the segments in the sets $S_0, S_1, \ldots, S_{i-1}$, which are moved to S_i. Since there are at least $B^{1+(i-1)\varepsilon} \geq I/B^\varepsilon$ such segments, it is sufficient to charge $O(1)$ I/Os to each of the $\frac{1}{2}I/B^\varepsilon$ most recently inserted segments in $S_0, \ldots, S_{i-1}$. This way a segment x is only charged when it moves from a set S_j to a set S_i, with $j < i$, when the number of segments in $\mathcal{M}$ when x was inserted is $K_x \geq \frac{1}{2}B^{1+(i-1)\varepsilon}$. If follows that x can at most be charged $O(\log_B K_x)$ times. Thus an insertion requires amortized $O(\log_B K)$ I/Os.

To delete a segment we first perform a query to locate the deletion-only structure storing the segment. Then we simply delete the segment from the relevant deletion-only structure as described in Section 3.3. This may result in many (sampled) segments being deleted from or inserted into L^k lists. For each such update we also update the corresponding simultaneous search structure using $O(\log_B K)$ I/Os (Lemma 3). Since we in the analysis of the deletion-only structure in Section 3.3 already charged $O(\log_B K)$ I/Os to each update to a L^k list, it follows that we have already accounted for the cost of updating the simultaneous search structures. Thus a deletion requires amortized $O(\log_B K)$ I/Os.

To construct our structure on an initial set $\mathcal{M}$ of K segments we simply create a single additional deletion-only structure S_{-1} on $\mathcal{M}$ using $O(K/B^\varepsilon)$ I/Os (Lemma 6). The structure for S_{-1} is never merged with the other deletion-only structures and is queried and updated separately.

Finally, to limit the number of deletion-only structures to $O(\log_B K)$ we periodically rebuild the structure when half of the inserted segments have been deleted. The rebuilding cost of $O(K/B^\varepsilon)$ I/Os is charged to the $\Theta(K)$ deleted segments.

LEMMA 7. *For a constant $0 < \varepsilon \leq 1/5$ and $\log_B K = O(B^{2\varepsilon})$, a set of K non-intersecting segments with endpoints on $B^\varepsilon + 1$ vertical lines can be stored in a linear space data structure such that a vertical ray-shooting query can be answered in $O(\log_B K)$ I/Os. Updates can be performed in amortized $O(\log_B K)$ I/Os. The structure can be constructed in amortized $O(K/B^\varepsilon)$ I/Os.*

4. REBALANCING THE BASE TREE

In this section we briefly discuss how to handle updates in the base interval tree. To do so efficiently, we use a *weight balanced* B^ε-tree [8] as the B^ε-tree used for the base interval tree $\mathcal{T}$. In such a tree a node v at height h has $K = \Theta(B^{h\varepsilon})$ leaves in its subtree, and if rebalancing (split or fuse) of v (or rather the reorganization of its secondary structures) can be performed in $O(K)$ I/Os one obtain an amortized $O(\log_B N)$ update bound [8]. Below we discuss how to perform rebalancing of a node v when it splits as a result of insertion of new segments. Deletions are handled in a standard way by a periodical global rebuilding of the entire structure.

Consider node v at height h with $K = \Theta(B^{h\varepsilon})$ leaves in its subtree that has to be split into two nodes v_1 and

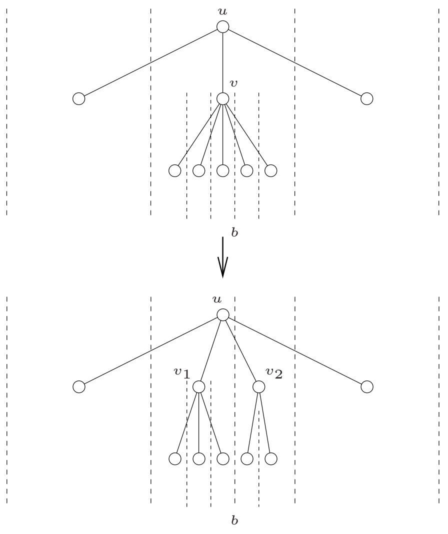

Figure 5: Splitting a node v along b into two nodes v_1 and v_2. The boundary b becomes a new boundary in the parent u.

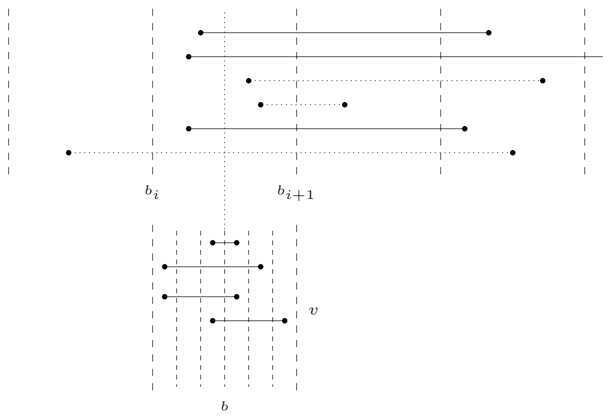

Figure 6: All solid intervals need to move. Intervals in v containing b move to the parent u and some intervals move within the parent u.

v_2, and let u be the parent of v; the number of segments stored in the secondary structures of v and u is $O(K)$ and $O(KB^\varepsilon)$, respectively. Figure 5 illustrates how the slabs associated with v are affected by the split: All the slabs on one side of a slab boundary b get associated with v_1, the slab on the other side of b get associated with v_2, and b becomes a new slab boundary in u. As a result, all segments in the secondary structures of v containing b need to be inserted into the secondary structures of u. The rest of the segments need to be stored in the secondary structures of v_1 and v_2. Furthermore, as a result of the addition of the new boundary b, some of the intervals in u containing b also need to be moved to new secondary structures. Refer to Figure 6.

We first consider the segments in the secondary structures of v and the construction of the secondary structures for v_1 and v_2. Since each segment is stored in a left and a right structure, we can collect all segments containing b (to be moved to u) from v's left and right structures. We first use $O(K/B)$ I/Os to scan through the left structure of each slab s_k in turn, while constructing a list of segments that should stay in the left structure of s_k in v_1 or v_2 and a list of segments that should be inserted in the left structure of the slab in u with right boundary b; the segments in each list are automatically sorted by the y-coordinate of their intersection with the right boundary of s_k [1]. Note that the $O(B^\varepsilon)$ lists of segments that should be inserted in the new left structure in u are also automatically sorted by the y-coordinate of the intersection with b. We can therefore merge these lists into one list of segments sorted by y-coordinate of their intersection with b in a binary tree fashion using $O((K/B)\log_2 B^\varepsilon) = O(K)$ I/Os. Similarly, in $O(K)$ I/Os we can construct a sorted list of segments that should stay in right structures of each slab s_k in v_1 or v_2, as well as a sorted list of segments to be inserted in the right structure of the slab in u with left boundary b. Next we construct the left and right structures for v_1 and v_2 from the $O(B^\varepsilon)$ sorted lists of segments that should not move to u; we can easily do so in $O(K/B)$ I/Os in total (Lemma 1). To construct the multislab structures for v_1 and v_2 we distinguish between two cases. In the case when $B^{2\varepsilon} = O(\log_B N)$ we simply construct the multislab structures from the relevant (already sorted) multislab lists from v in $O(K/B)$ I/Os (Lemma 4); note that all multislab lists containing b are just deleted. In the case when $\log_B N = O(B^{2\varepsilon})$ we simply directly construct the multislab structures for v_1 and v_2 from the relevant segments in v using $O(N/B^\varepsilon)$ I/Os (Lemma 7). Overall, we have constructed all the secondary structure of v_1 and v_2 in $O(K)$ I/Os as required.

Next consider the segments in u. Some of the $O(K)$ segments stored in the left structure of the slab in u containing b (segments with left endpoint in the slab containing b) need to be moved to a new left structure for the new slab to the left of b. We therefore scan through the K segments in the left structure using $O(K/B)$ I/Os and construct a list of segments that should stay in the old left structure and a list of segments for the new left slab structure, both sorted by the y-coordinate of the intersection with the relevant slab boundary. We merge the first list with the segments collected in v using $O(K/B)$ I/Os, and then we construct the two left structures in another $O(K/B)$ I/Os (Lemma 1). Similarly, we can construct the two relevant right structures from the segments in the right structure of the slab containing b in u and the segments collected in v in $O(K/B)$ I/Os. Finally, since u gets a new slab the multislab structure of u also needs to be reconstructed. Again we distinguish between two cases. In the case when $B^{2\varepsilon} = O(\log_B N)$ we simply scan through each of the multislab lists for u and compute the new (sorted) multislab lists in $O(KB^\varepsilon/B) = O(K)$ I/Os. Then we construct the new multislab structure for u in $O(KB^\varepsilon/B) = O(K)$ I/Os (Lemma 4). In the case when $\log_B N = O(B^{2\varepsilon})$ we directly construct the new multislab structures from the segments in the old multislab structure of u using $O(KB^\varepsilon/B^\varepsilon) = O(K)$ I/Os (Lemma 7). Overall, we have constructed all the secondary structures of u in $O(K)$ I/Os as required.

5. CONCLUSION

We have presented an I/O-efficient dynamic point location data structure that stores a planar subdivision of size N using linear space ($O(N/B)$ disk blocks), and supports insertions and deletions in amortized $O(\log_B N)$ I/Os and queries in $O(\log_B^2 N)$ I/Os in the worst-case. This improves the insertion bound of the earlier best known structure [6]. Our structure is also considerably simpler than previous structures. It remains open to improve the query time to $O(\log_B N)$ I/Os.

6. REFERENCES

[1] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1127, 1999.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] L. Arge, G. S. Brodal and L. Georgiadis. Improved dynamic planar point location. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 305–314, 2006.

[4] L. Arge, A. Danner, and S.-H. Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithms* 8, 2003.

[5] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 685–694, 1998.

[6] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Comput. Geom.*, 29(2):147–162, 2004.

[7] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47(1):1–25, 2007.

[8] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal of Computing*, 32(6): 1488-1508, 2003.

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

[10] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[11] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM Journal on Computing*, 21(5):972–999, 1992.

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

[13] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[14] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, June 2001.

[15] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

[16] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[17] H. Edelsbrunner and H. A. Maurer. A space-optimal solution of general region location. *Theoretical Computer Science*, 16:329–336, 1981.

[18] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[19] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

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

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

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

[23] J. Vahrenhold and K. H. Hinrichs. Planar point location for large data sets: To seek or not to seek. In *Proc. Workshop on Algorithm Engineering, LNCS 1982*, pages 184–194, 2001.