

External Memory Planar Point Location with Fast Updates

John Iacono

Université Libre de Bruxelles, Belgium
New York University, USA
<http://www.johniacono.com>
john@johniacono.com

Ben Karsin

Université Libre de Bruxelles, Belgium
<https://www.benkarsin.com>
bkarsin@gmail.com

Grigorios Koumoutsos

Université Libre de Bruxelles, Belgium
<http://homepages.ulb.ac.be/~gkoumout>
gregkoumoutsos@gmail.com

Abstract

We study dynamic planar point location in the *External Memory Model* or Disk Access Model (DAM). Previous work in this model achieves polylog query and polylog amortized update time. We present a data structure with $O(\log_B^2 N)$ query time and $O(\frac{1}{B^{1-\epsilon}} \log_B N)$ amortized update time, where N is the number of segments, B the block size and ϵ is a small positive constant, under the assumption that all faces have constant size. This is a $B^{1-\epsilon}$ factor faster for updates than the fastest previous structure, and brings the cost of insertion and deletion down to subconstant amortized time for reasonable choices of N and B . Our structure solves the problem of vertical ray-shooting queries among a dynamic set of interior-disjoint line segments; this is well-known to solve dynamic planar point location for a connected subdivision of the plane with faces of constant size.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Computational geometry; Theory of computation → Models of computation

Keywords and phrases point location, data structures, dynamic algorithms, computational geometry

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.58

Funding This work was supported by the Fonds de la Recherche Scientifique-FNRS under Grant no MISU F 6001 1 and by NSF Grant CCF-1533564.

1 Introduction

The *dynamic planar point location* problem is one of the most fundamental and extensively studied problems in geometric data structures, and is defined as follows: We are given a connected planar polygonal subdivision Π with N edges. For any given query point p , the goal is to find the face of Π that contains p , subject to insertions and deletions of edges. Here we focus on subdivisions Π such that each face has constant number of edges. An equivalent formulation, which we use here is as follows: given a set S of N interior-disjoint line segments in the plane, for any given query point p , report the first line segment in S that a vertical upwards-facing ray from p intersects, subject to insertions and deletions of segments.

Dynamic planar point location has many applications in spatial databases, geographic information systems (GIS), computer graphics, etc. Moreover it is a natural generalization of the dynamic dictionary problem with predecessor queries; this problem can be seen as the one dimensional variant of planar point location.



© John Iacono, Ben Karsin, and Grigorios Koumoutsos;
licensed under Creative Commons License CC-BY

30th International Symposium on Algorithms and Computation (ISAAC 2019).

Editors: Pinyan Lu and Guochuan Zhang; Article No. 58; pp. 58:1–58:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we focus on the External Memory model, also known as the Disk Access Model (DAM) [2]. The DAM is the standard method of designing algorithms that efficiently execute on large datasets stored in secondary storage. This model assumes a two-level memory hierarchy, called disk and internal memory and it is parameterized by values M and B ; the disk is partitioned into blocks of size B , of which M/B can be stored in memory at any given moment. The cost of an algorithm in the DAM is the number of block transfers between memory and disk, called Input-Output operations (I/Os). The quintessential DAM-model data structure is the B-Tree [11]. See [25, 26] for surveys. Many applications of dynamic planar point location, such as GIS problems, must efficiently process datasets that are too massive to fit in internal memory, thus it is of great relevance and interest to consider the problem in the DAM and to devise I/O efficient algorithms.

1.1 Previous Work

RAM Model. In the RAM model (the leading model for applications where all data fit in the internal memory) the dynamic planar point location problem has been extensively studied [4, 10, 15, 18, 19, 21]. It is a major and long-standing open problem in computational geometry to design a data structure that supports queries and updates in $O(\log N)$ time [16, 17, 24], i.e., to achieve the same bounds as for the dynamic dictionary problem. In a recent breakthrough, Chan and Nekrich in FOCS'15 [15] presented a data structure supporting queries in $O(\log N(\log \log N)^2)$ time and updates in $O(\log N(\log \log N))$ time. They also showed the tradeoff of supporting queries in $O(\log N)$ time and updates in $O((\log N)^{1+\epsilon})$ time or vice-versa for $\epsilon > 0$.

Recently Oh and Ahn [23] presented the first data structure for a more general setting where the polygonal subdivision Π is not necessarily connected; their data structure supports queries in $O(\log N(\log \log N)^2)$ time and updates in $O(\sqrt{N} \log N(\log \log N)^{3/2})$ amortized time.

External Memory model. (See Table 1). Several data structures have been presented over the years which support queries and updates in $\text{polylog}(N)$ I/Os [1, 5, 7]. Table 1 contains a list of results of prior work. The best update bound known is by Arge, Brodal and Rao [5] and achieves $O(\log_B N)$ amortized I/Os. The query time of their data structure is $O(\log_B^2 N)$. Very recently, the first data structure that supports queries in $o(\log_B^2 N)$ I/Os was announced by Munro and Nekrich [22]. In particular they support queries in $O(\log_B N(\log \log_B N)^3)$ I/Os. However their update time is slightly worse than logarithmic, $O(\log_B N(\log \log_B N)^2)$. In all those works the bounds are obtained by solving the problem of vertical ray-shooting.

Fast Updates in External Memory. One of the most intriguing and practically relevant features of the external memory model is that it allows fast updates. For the dynamic dictionary problem with predecessor queries, the optimal update bound in the RAM model is $O(\log N)$. In external memory, however, B -trees achieve the optimal query time of $O(\log_B N)$ and typical update time of $O(\log_B N)$, although substantially faster update times are possible. Brodal and Fagerberg [14] showed that $O(\frac{1}{B^{1-\epsilon}} \log_B N)$ amortized I/Os per update can be supported, for small positive constant, ϵ , while retaining $O(\log_B N)$ -time queries; they further showed that this is an asymptotically optimal tradeoff between updates and queries. Observe that this update bound is a huge speedup from $O(\log_B N)$ and that for reasonable choices of parameters, e.g. $B \geq 1000$, $N < 10^{93}$, $\epsilon = \frac{1}{2}$, this yields a subconstant amortized number of I/Os per update. A similar update bound was later achieved for other dynamic problems like three-sided range reporting and top- k queries [13].

■ **Table 1** Overview of results on dynamic planar point location in external memory. Results marked with M are for monotone subdivisions and G for general ray-shooting among non-intersecting segments. Query bounds are worst-case and update bounds are amortized. Space usage is measured in words. Here ϵ is a constant such that $0 < \epsilon \leq 1/2$.

Reference	Space	Query Time	Insertion Time	Deletion Time	
Agarwal et al. [1]	$O(N)$	$O(\log_B^2 N)$	$O(\log_B^2 N)$	$O(\log_B^2 N)$	M
Arge and Vahrenhold [7]	$O(N)$	$O(\log_B^2 N)$	$O(\log_B^2 N)$	$O(\log_B N)$	G
Arge et al. [5]	$O(N)$	$O(\log_B^2 N)$	$O(\log_B N)$	$O(\log_B N)$	G
Munro and Nekrich [22]	$O(N)$	$O(\log_B N \log^3 \log_B N)$	$O(\log_B N \log^2 \log_B N)$	$O(\log_B N \log^2 \log_B N)$	G
This paper	$O(N)$	$O(\log_B^2 N)$	$O((\log_B N)/B^{1-\epsilon})$	$O((\log_B N)/B^{1-\epsilon})$	G

Given this progress and the fact that in the RAM model the bounds achieved for planar point location and the dictionary problem are believed to coincide, it is natural to conjecture that a similar update bound can be achieved for the dynamic planar point location problem. However, to date no result has been presented that achieves sublogarithmic insertion or deletion time.

1.2 Our Results

We consider the dynamic planar point location problem in the external memory model and present the first data structure with sublogarithmic amortized update time of $O(\frac{1}{B^{1-\epsilon}} \log_B N)$ I/Os. Prior to our work, the best update bound for both insertions and deletions was $O(\log_B N)$, achieved by Arge et al. [5]. Our main result is:

► **Theorem 1 (Main result).** *For any constant $0 < \epsilon \leq 1/2$, there exists a data structure which uses $O(N)$ space, answers planar point location queries in $O((1/\epsilon)^2 \cdot \log_B^2 N) = O(\log_B^2 N)$ I/Os and supports insertions and deletions in $O(\log_B N / (\epsilon \cdot B^{1-\epsilon})) = O((\log_B N) / B^{1-\epsilon})$ amortized I/Os. The data structure can be constructed in $O((N/B) \log_B N)$ I/Os.*

To obtain this result, several techniques are used. Our primary data structure is an augmented *interval tree* [20]. We combine both the primary interval tree and two auxiliary structures described below with the *buffering* technique [3, 14] to improve insertion and deletion bounds. In Section 2 we prove Theorem 1 using our auxiliary structures as black boxes and omit some technical details relating to rebuilding; these details are deferred to Appendix 5.

Similarly to previous work, we focus on solving the problem of vertical ray-shooting queries. Our first auxiliary structure answers vertical ray-shooting queries among non-intersecting segments whose right (left) endpoints lie on the same vertical line. This is called the *left (right) structure* (in Section 2 it will be clear why we choose this terminology and not vice-versa). Left/Right structures of Agarwal et al. [1], which support queries and updates in $O(\log_B K)$ I/Os, are used by several prior works [1, 5, 7]. Our structure improves on their result by reducing the update bound by a factor of $B^{1-\epsilon}$. We obtain the following result, the proof of which is the topic of Section 3:

► **Theorem 2 (Left/right structure).** *For a set of K non-intersecting segments whose right (left) endpoints lie in the same vertical line and any constant $0 < \epsilon \leq 1/2$, we can create a data structure which supports vertical ray-shooting queries in $O((1/\epsilon) \cdot \log_B K) = O(\log_B K)$*

I/Os and insertions and deletions in $O((\log_B K)/(\epsilon \cdot B^{1-\epsilon})) = O((\log_B K)/B^{1-\epsilon})$ amortized I/Os. This data structure uses $O(K)$ space and it can be constructed in $O((K/B) \log_B K)$ I/Os. If the segments are already sorted, it can be constructed in $O(K/B)$ I/Os.

Our second auxiliary structure answers vertical ray-shooting queries among non-intersecting segments whose endpoints lie in a set of $B^{\epsilon/2} + 1$ vertical lines. These vertical lines define $B^{\epsilon/2}$ vertical slabs, hence the structure is called a *multislab* structure. We obtain the following result, the proof of which is the topic of Section 4:

► **Theorem 3 (Multislab structure).** *For any constant $0 < \epsilon \leq 1/2$ and set of K non-intersecting segments whose endpoints lie in $B^{\epsilon/2} + 1$ vertical lines, we can create a data structure which supports vertical ray-shooting queries in $O((1/\epsilon) \cdot \log_B K) = O(\log_B K)$ I/Os and insertions and deletions in $O((\log_B K)/(\epsilon \cdot B^{1-\epsilon})) = O((\log_B K)/B^{1-\epsilon})$ amortized I/Os. This data structure uses $O(K)$ space and it can be constructed in $O((K/B) \log_B K)$ I/Os. If the segments are already sorted according to a total order, it can be constructed in $O(K/B)$ I/Os.*

A major challenge faced by previous multislab structures is how to efficiently support insertions. At a high-level, it is hard to deal with insertions in cases where a total order is maintained: each time a new segment gets inserted we need to determine its position in the total order, which cannot be done quickly. Arge and Vitter [7] developed a deletion-only multislab data structure and then used the so-called logarithmic method [12] which allowed them to handle insertions in $O(\log_B^2 K)$ I/Os. Later Arge, Brodal and Rao [5] developed a more complicated multislab structure supporting insertions in amortized $O(\log_B K)$ I/Os by performing separate case analysis depending on the value of B .

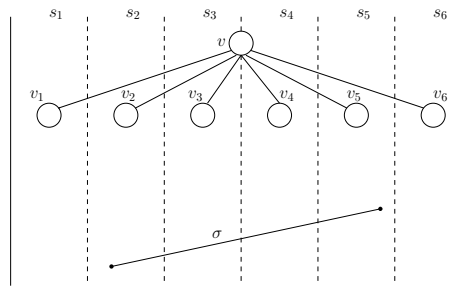
Here, we support insertions in a much simpler way by breaking each inserted segment into smaller *unit segments* whose endpoints lie on two consecutive vertical lines and can be compared easily to the segments already stored. This way, we are able to support insertions easily in $O(\log_B K)$ I/Os. Finally, we add buffering and obtain sublogarithmic update bounds.

1.3 Notation and Preliminaries

External Memory Model. Throughout this paper we focus on the external memory model of computation. N denotes the number of segments in the planar subdivision, B the block size and M the number of elements that fit in internal memory. We assume that $M \ll N$ and $2 \leq B \leq \sqrt{M}$ (the *tall cache assumption*). It is well-known that sorting K elements requires $\Theta((K/B) \log_{M/B}(K/B))$ I/Os [2]. Given that $B \leq \sqrt{M}$, this bound is $O((K/B) \log_B K)$. We use this bound for sorting in many places without further explanation.

Ray-shooting Queries. In the rest of this paper, we focus on answering vertical ray-shooting queries in a dynamic set of non-intersecting line segments. Let S be the set of segments of the polygonal subdivision Π . Given a query point p , the answer to a vertical ray-shooting query is the the first segment of S hit by a vertical ray emanating from a query point in the $(+y)$ direction. Based on standard techniques (see e.g. [7]), for connected polygonal subdivisions Π with faces of size $O(1)$, a planar point location query for a point p can be answered in $O(\log_B N)$ I/Os after answering a vertical ray-shooting query for p .

B^ϵ -Trees. All tree structures that we will use are variants of the B^ϵ -Trees [14] which are B -trees except that the internal nodes have at most B^ϵ (and not B) children; the leaves still store $\Theta(B)$ data items. For constant ϵ , this does not change the asymptotic height of the tree or the search cost, both remain $O((1/\epsilon) \cdot \log_B N) = O(\log_B N)$.



■ **Figure 1** The slab of node v of the interval tree \mathcal{I} is divided into slabs s_1, \dots, s_6 corresponding to its children v_1, \dots, v_6 . Segment σ is assigned to node v , with left subsegment in slab s_2 , right subsegment in s_5 and the middle subsegment crosses slabs s_3, s_4 .

2 Overall Structure

In this Section we prove Theorem 1, using the data structures of Theorems 2 and 3 (detailed in Sections 3 and 4, respectively). Given N non-intersecting segments in the plane and a constant $0 < \epsilon \leq 1/2$, we construct a $O(N)$ -space data structure which answers vertical ray-shooting queries in $O((1/\epsilon)^2 \cdot \log_B^2 N) = O(\log_B^2 N)$ I/Os and supports updates in $O((\log_B N)/(\epsilon \cdot B^{1-\epsilon})) = O((\log_B N)/B^{1-\epsilon})$ amortized I/Os. Throughout this section we let $\epsilon' = \epsilon/2$.

The Data Structure. As in the previous works on planar point location, our primary data structure is based on the *interval tree* (the external interval tree defined in [9]). Our interval tree \mathcal{I} is a $B^{\epsilon'}$ -tree which stores the x -coordinates of segment endpoints in its leaves. Here we assume for clarity of presentation that the interval tree is static, i.e. all new segments inserted share x -coordinates with already stored segments; in Appendix 5 we remove this assumption and extend our data structure to accommodate new x -coordinates and achieve the bounds of Theorem 1.

Each node of \mathcal{I} is associated with several secondary structures, as we explain later, and each segment is stored in the secondary structures of exactly one node of \mathcal{I} . Each node v of \mathcal{I} is associated with a vertical slab s_v . The slab of the root is the whole plane. For an internal node v , the slab s_v is divided into $B^{\epsilon'}$ vertical slabs $s_1, \dots, s_{B^{\epsilon'}}$ corresponding to the children of v , separated by vertical lines called *slab boundaries*, such that each slab s_i contains the same number of vertices of Π from slab s_v .

Let S be the set of segments that compose Π . Each segment $\sigma \in S$ is *assigned* to a node v of \mathcal{I} . This is the highest node v of \mathcal{I} such that σ is completely contained in slab s_v and intersects at least one slab boundary partitioning s_v ; if such an internal node v does not exist, then σ is assigned to a leaf v such that σ is completely contained in its slab s_v . Segments assigned to internal nodes are stored in the secondary structures of those nodes, whereas segments assigned to leaves are stored explicitly in the corresponding leaf. By construction of the slab boundaries, each leaf stores $O(B)$ segments in $O(1)$ blocks.

Consider a segment σ assigned to a node v of \mathcal{I} . Let s_ℓ and s_r be the children slabs of s_v where the left and right endpoints of σ lie. We call the segment $\sigma \cap s_\ell$ the *left subsegment* of σ , the segment $\sigma \cap s_r$ the *right subsegment* of σ and the rest of σ (which spans children slabs $s_{\ell+1}, \dots, s_{r-1}$) is its middle subsegment. See Figure 1 for an illustration. In this example, the left subsegment is $\sigma \cap s_5$, the right subsegment is $\sigma \cap s_2$, and the portion of σ in s_3 and s_4 is the middle subsegment.

Let S_v be the set of segments assigned to a node v of \mathcal{I} . To store segments of S_v , node v of \mathcal{I} contains the following secondary structures:

1. A multislab structure \mathcal{M} which stores the set of middle segments.
2. $B^{\epsilon'}$ left structures L_i , for $1 \leq i \leq B^{\epsilon'}$, storing the left (sub)segments of slab s_i .
3. $B^{\epsilon'}$ right structures R_i , for $1 \leq i \leq B^{\epsilon'}$, storing the right (sub)segments of slab s_i .

In addition, each internal node v contains an insertion buffer I_v and deletion buffer D_v , each storing up to B segments.

Construction and Space Usage. For every node v , the buffers I_v and D_v fit in $O(1)$ blocks, since they store at most B segments. By Theorems 2 and 3, a secondary structure storing K segments uses $O(K)$ space. Since each segment of S_v is stored in at most 3 secondary structures, overall secondary structures of v use $O(|S_v|)$ space. Thus each node v uses $O(|S_v|)$ space. We get that our data structure uses overall $O(\sum_{v \in \mathcal{I}} |S_v|) = O(N)$ space. The interval tree can be constructed in $O((N/B) \log_B N)$ I/Os. This can be done by sorting the segments by their endpoints' x -coordinates and then determining all slab boundaries to create a balanced interval tree. By Theorems 2 and 3, all secondary structures of a node v of \mathcal{I} can be constructed in $O((|S_v|/B) \log_B |S_v|)$ I/Os. Thus, all secondary structures of the tree can be constructed in $O((\sum_{v \in \mathcal{I}} |S_v|/B) \cdot \log_B N) = O((N/B) \log_B N)$ I/Os.

Queries. To answer a vertical ray-shooting query for a point p , we traverse the root-to-leaf path of \mathcal{I} based on the x -coordinate of p , while maintaining a segment σ (initialized to `null`) which is the answer to the query among segments assigned to nodes we have traversed so far. At each node v visited along this path, we first update buffers I_v and D_v by removing from both of them all segments (if any) of $I_v \cap D_v$. Then, we perform a vertical ray-shooting on the secondary structures of v ; in particular we ray-shoot on the multislab structure and the left and right structures L_i and R_i , for i such that the query point p is in slab s_i ¹. After checking the secondary structures, we update σ if a closer segment above p is found as a result. Next, we ray-shoot among segments stored in I_v and update σ if necessary. Finally, we determine which child v_i of v to visit, and flush any segments of D_v that are contained in the slab of v_i to D_{v_i} ; this way we make sure that information about deleted segments is updated throughout the root-to-leaf path and no deleted segment can be considered as an answer to the query. We then continue the process at v_i . Once a leaf node is reached, we simply compare the B segments it contains with p and return the closest segment above p among them and σ .

Bounding the query cost: Since any root-to-leaf path of \mathcal{I} has length $O((1/\epsilon') \cdot \log_B N)$, each secondary data structure supports ray-shooting queries in $O((1/\epsilon') \cdot \log_B N)$ I/Os (due to Theorems 2 and 3) and we check $O(1)$ secondary structures per node, we get that a query is answered in $O((1/\epsilon')^2 \cdot \log_B^2 N) = O(\log_B^2 N)$ I/Os. Note that in each node v of the root-to-leaf path visited, the operations involving I_v and D_v require $O(1)$ I/Os, thus they increase the total cost by at most a $O(1)$ factor.

¹ Minor detail: For each secondary structure considered, we first perform insertions/deletions of the corresponding segments from buffers I_v and D_v .

Insertions. To handle insertions, we use the insertion buffers stored in nodes of \mathcal{I} . When a new segment σ is inserted, we insert it in the insertion buffer of the root. Let v be an internal node with children $v_1, \dots, v_{B^{\epsilon'}}$. Whenever I_v becomes full, it is flushed. Segments of I_v that cross at least one slab boundary partitioning s_v are inserted in the secondary structures of v ; segments that are contained in the slab s_i of v_i are inserted in I_{v_i} , for $1 \leq i \leq B^{\epsilon'}$. In case I_v becomes full for some node v whose children are leaves, we insert those segments explicitly at the corresponding leaves. When a leaf becomes full, we restructure the tree using split operations on full nodes.

Bounding the insertion cost: We compute the amortized cost of an insertion by considering three components:

- (i) The cost for moving segments between insertion buffers. Whenever an insertion buffer I_v gets full, it forwards segments to the buffers of its $B^{\epsilon'}$ children performing $O(B^{\epsilon'})$ I/Os. Since a flushing occurs every B insertions in I_v , the amortized cost of such operations is $O(B^{\epsilon'}/B) = O(1/(B^{1-\epsilon'}))$. Each segment will move in at most $O((1/\epsilon') \log_B N)$ insertion buffers before it is inserted in the secondary structures of a node (or in a leaf). Thus the amortized cost for moving between buffers is $O((\log_B N)/(\epsilon' \cdot B^{1-\epsilon'}))$.
- (ii) The insertion cost in the secondary structures. By Theorems 2 and 3 we get that insertions in secondary structures require $O((\log_B N)/(\epsilon \cdot B^{1-2\epsilon'}))$ I/Os.
- (iii) The cost of restructuring the tree after insertions when a leaf becomes full. We show in Section 5 that the restructuring requires $O(\frac{\log_B N}{\epsilon' \cdot B^{1-\epsilon'}})$ amortized I/Os, by slightly modifying our primary interval tree data structure.

We conclude that our data structure supports insertions in amortized $O(\log_B N/(\epsilon' \cdot B^{1-2\epsilon'})) = O(\log_B N/B^{1-\epsilon})$ I/Os.

Deletions. To support deletions, we use the deletion buffers stored in all nodes of \mathcal{I} . To delete a segment σ , we first check whether σ is in the insertion buffer I_r of the root r and in that case we delete it; otherwise we store it in D_r . Similar to insertions, whenever D_v gets full for some internal node v with children $v_1, \dots, v_{B^{\epsilon'}}$, we flush D_v . The segments of D_v crossing at least one slab boundary partitioning s_v are deleted from the corresponding secondary structures associated with v ; the other segments of D_v are moved to buffers D_{v_i} ; in case a segment σ inserted in $D_{v_i} \cap I_{v_i}$, we delete it from both buffers. In case D_v becomes full for some v parent of leaves, we delete those segments explicitly from the corresponding leaves.

Bounding the deletion cost: The deletion cost has three components:

- (i) Moving segments between the deletion buffers. Using the same argument as for insertions, we get that this requires $O(\log_B N/(\epsilon' \cdot B^{1-\epsilon'}))$ I/Os, amortized.
- (ii) The cost of deletion in the secondary structures. By Theorems 2 and 3 we get that deletions in secondary structures require amortized $O(\log_B N/(\epsilon' \cdot B^{1-2\epsilon'}))$ I/Os.
- (iii) The cost of restructuring the tree. Every $N/2$ deletions, we rebuild the structure using $O((N/B) \log_B N)$ I/Os, to get an amortized restructuring cost of $O((\log_B N)/B)$ I/Os.

Overall deletions are supported in amortized $O(\log_B N/(\epsilon' \cdot B^{1-2\epsilon'})) = O(\log_B N/(B^{1-\epsilon}))$ I/Os.

3 Left and Right Structures

In this section we prove Theorem 2. Given K points all of whose right (left) endpoints lie on a single vertical line, we construct a data structure which answers vertical ray-shooting queries on those segments in $O(\log_B K)$ I/Os and supports insertions and deletions in $O((\log_B K)/B^{1-\epsilon})$ amortized I/Os for a constant $0 < \epsilon \leq 1/2$.

We describe the structure for the case where we are given a set \mathcal{L} of K segments whose right endpoints have the same x -coordinate (left structure)². The case where the left endpoints of the segments have the same x -coordinate (right structure) is completely symmetric. For a segment σ , we will refer to the y -coordinate of its right endpoint as the y -coordinate of σ . Conversely we define the x -coordinate of σ to be the x -coordinate of its left endpoint.

Total Order. We assume that the segments in \mathcal{L} are ordered according to their y -coordinates. We can always order the segments according to this total order in $O((K/B)\log_B K)$ I/Os.

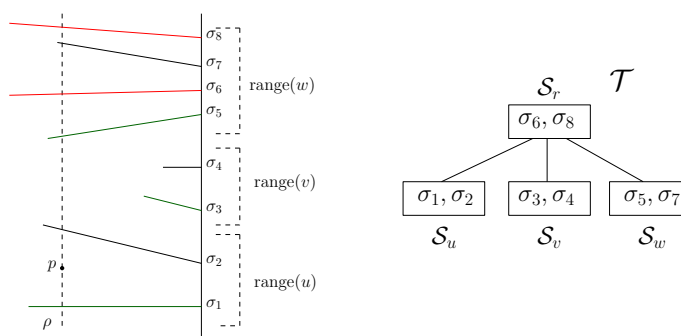
The Data Structure. We store all segments of \mathcal{L} in an augmented B^ϵ -tree \mathcal{T} which supports vertical ray-shooting queries, insertions and deletions. The degree of each node is between $B^\epsilon/2$ and B^ϵ , except the root which might have degree in the range $[2, B^\epsilon]$, and leaves store $\Theta(B)$ elements. For a node $v \in \mathcal{T}$, let \mathcal{T}_v be the subtree rooted at v . Since the segments are sorted according to their y -coordinates, each subtree \mathcal{T}_v corresponds to a range of y -coordinates, which we call the y -range of node v . Let v be an internal node of \mathcal{T} with children $v_1, \dots, v_{B^\epsilon}$. Node v stores the following information:

1. A buffer of segments \mathcal{S}_v of capacity B which contains segments in the y -range of v whose left endpoints have the smallest x -coordinates (i.e., segments that extend the farthest from the vertical line) and are not stored in any buffer \mathcal{S}_w for an ancestor w of v . In other words, \mathcal{T} together with segments of buffers \mathcal{S}_v form an external memory priority search tree [6].
2. An insertion buffer I_v and a deletion buffer D_v , each storing up to B segments.
3. A list \mathcal{M}_v that contains, for each child v_i , the segment with minimum x -coordinate stored in \mathcal{S}_{v_i} . We call this the *minimal* segment for child v_i .

The data structure satisfies the following invariants: For each node $v \in \mathcal{T}$, either $|\mathcal{S}_v| \geq B/2$ or if $|\mathcal{S}_v| < B/2$, then I_v and D_v are empty and all buffers stored in descendants v are empty. Also, for each node v , buffers \mathcal{S}_v, I_v and D_v are disjoint. Finally, for a leaf v , I_v and D_v are empty.

Construction and Space Usage. Overall buffers and lists of each node contain $O(B)$ segments, i.e. they can be stored in $O(1)$ blocks. Thus \mathcal{T} can be stored in $O(K/B)$ blocks, i.e. it requires $O(K)$ space. Construction of \mathcal{T} requires $O(\frac{K}{B}\log_B K)$ I/Os, since we need to sort all K segments according to their y -coordinates. If the segments are already sorted according to their y -coordinate, then \mathcal{T} can be created in $O(K/B)$ I/Os.

² Recall from Section 2 that we call left structures the ones storing the left subsegment of a segment σ , thus all subsegments stored in a left structure have the same x -coordinate of right endpoints.



■ **Figure 2** Example of the query algorithm in the left structure: Left column shows the segments stored in \mathcal{T} , the query point p and the vertical ray ρ emanating from p . Right column shows buffers \mathcal{S} of the nodes of \mathcal{T} . Red segments are stored in the root. For nodes u, v, w , the green segment is their minimal segment, i.e., the one stored in list \mathcal{M}_r . By ray-shooting on ρ among green segments, the first segment hit upwards is σ_5 , which is stored in \mathcal{S}_w , thus we set $v_+ = w$. Note that σ_2 (the correct answer for the query) is not stored in \mathcal{S}_w , i.e., maintaining only v_+ produces an incorrect answer. Thus, our algorithm ray-shoots downwards as well, hitting σ_1 , which is stored in u , and setting $v_- = u$. Then, by ray-shooting on ρ among \mathcal{S}_u and \mathcal{S}_w , the first segment we hit upwards of p is σ_2 .

Queries in the static structure.

To get a feel for how our structure supports queries, we first show how to perform queries in the static case, i.e., assuming there are no insertions and deletions and all buffers I_v and D_v are empty. Later we will give a precise description of performing queries in the fully dynamic structure.

Let ρ^+ be the ray emanating from p in the $(+y)$ direction and ρ^- the ray emanating from p in the $(-y)$ direction. We query the structure by finding the first segment hit by both ρ^+ and ρ^- . We keep two pointers, v_+ and v_- , initialized at the root. We also keep the closest segments σ_+ and σ_- seen so far in the $(+y)$ and $(-y)$ direction respectively (initialized to $+\infty$ and $-\infty$). At each step, we update both v_+ and v_- to move from a node of depth i to a node of depth $i + 1$. While at level i , v_- and v_+ might coincide, or one of them might be undefined (set to null).

We now describe the query algorithm. We start at the root of \mathcal{T} and advance down, while updating v_+ , v_- , and σ_+, σ_- . When at depth i , we find the first segment σ_i hit by ρ^+ among \mathcal{S}_{v_-} and \mathcal{S}_{v_+} and update σ_+ if necessary (i.e. if σ_i is the first segment hit by ρ^+ among all segments seen so far). Similarly, we ray-shoot on ρ^- among \mathcal{S}_{v_-} and \mathcal{S}_{v_+} and update σ_- if necessary. To determine in which nodes of depth $i + 1$ to continue the search, we ray-shoot on ρ^+ among \mathcal{M}_{v_-} and \mathcal{M}_{v_+} and also ray-shoot on ρ^- among \mathcal{M}_{v_-} and \mathcal{M}_{v_+} (i.e., all minimal segments of children of v_- and v_+). Let σ_{m+} be the first segment in $\mathcal{M}_{v_+} \cup \mathcal{M}_{v_-}$ hit by ρ^+ (if such a segment exists) and v_s be the node containing σ_{m+} (if σ_{m+} exists). If the y -range of v_s is higher than the y -coordinate of σ_+ or if σ_{m+} does not exist, we leave v_+ undefined for level $i + 1$. Otherwise, we set $v_+ = v_s$. Similarly, call σ_{m-} the first minimal segment of $\mathcal{M}_{v_+} \cup \mathcal{M}_{v_-}$ hit by ρ^- and v_p be the node containing σ_{m-} (if such a segment exists). If the y -range of v_p is lower than the y -coordinate of σ_- or if σ_{m-} does not exist, we leave v_- undefined for level $i + 1$. Otherwise we set $v_- = v_p$.

If both v_+ and v_- are undefined for the next level $i + 1$, we stop the procedure and output σ_+ as the result to the vertical ray-shooting query. Otherwise we repeat the same procedure in the next level. When we reach a leaf level, we find the first segment hit by ρ^+ among \mathcal{S}_{v_-} and \mathcal{S}_{v_+} , update σ_+ if necessary, and output σ_+ as the result of the query.

Remark: The reader might wonder why we answer vertical ray-shooting queries in both directions and keep two pointers v_- and v_+ . Isn't it sufficient to answer queries in one direction and keep one pointer at each step? Figure 2 shows an example where this is not true and maintaining only the v_+ pointer would result in an incorrect answer.

The formal proof of correctness of this query algorithm is deferred to Appendix A.

Bounding the query cost: To count the cost, observe that in each step we move down the tree by one level and perform operations that require $O(1)$ I/Os, as we check $O(B)$ segments stored in the current nodes v_- and v_+ . Since the height of the tree is $O((1/\epsilon) \log_B K)$, a query is answered in $O((1/\epsilon) \log_B K) = O(\log_B K)$ I/Os.

Insertions. Assume we want to insert a segment σ into the left structure \mathcal{L} . If the x -value of σ is smaller than the maximum x -value of a segment stored in the buffer of the root \mathcal{S}_r , we insert σ into \mathcal{S}_r . Otherwise we store σ in the insertion buffer of the root I_r . Note that insertion of σ in \mathcal{S}_r might cause \mathcal{S}_r to overflow (i.e., $|\mathcal{S}_r| = B + 1$); in that case we move the segment of \mathcal{S}_r with the maximum x -value into the insertion buffer of the root I_r .

Let v be an internal node with children $v_1, \dots, v_{B^\epsilon}$. Whenever the insertion buffer I_v becomes full, we *flush* it, moving the segments to buffers of the corresponding children. For a segment σ that should be stored in child v_i , we repeat the same procedure as in the root: Check whether σ has smaller x -value than the maximum x -value of a segment stored in \mathcal{S}_{v_i} and if yes, store σ in \mathcal{S}_{v_i} , otherwise store it in I_{v_i} . If \mathcal{S}_{v_i} overflows, we move its last segment (i.e. the one with maximum x -value) into I_{v_i} . Also, if σ gets stored in \mathcal{S}_{v_i} and its x -value is smaller than all previous segments of \mathcal{S}_{v_i} , we update the minimal segment of v_i , \mathcal{M}_v .

When \mathcal{S}_v overflows for some leaf v , we split v into two leaves v_1 and v_2 , as in standard B -trees. Note that this might cause recursive splits of nodes at greater height.

Bounding the insertion cost: To flush a buffer I_v and forward segments to buffers \mathcal{S}_{v_i} and I_{v_i} , for $1 \leq i \leq B^\epsilon$ we perform $O(B^\epsilon)$ I/Os. Since I_v becomes full after at least B insertions, the amortized cost of moving a segment from I_v to buffers of a child of v is $O(B^\epsilon/B) = O(1/B^{1-\epsilon})$. Each inserted segment moves between buffers in a root-to-leaf path of length $O((1/\epsilon) \log_B K)$, thus the total amortized cost for moves between buffers is $O(\log_B K / (\epsilon \cdot B^{1-\epsilon}))$ I/Os. The restructuring of \mathcal{T} due to splitting nodes requires amortized $O(1/B)$ I/Os, as in standard B -trees. Thus, insertions are supported in $O(\log_B K / (\epsilon \cdot B^{1-\epsilon}))$ amortized I/Os.

Deletions. To delete a segment σ , we first check whether it is stored in the buffers of the root \mathcal{S}_r or I_r ; in this case we delete it. Otherwise, we insert σ in the deletion buffer of the root D_r .

Let v be an internal node with children $v_1, \dots, v_{B^\epsilon}$. Whenever D_v becomes full we flush it and move the segments to the corresponding children and repeat the same procedure: For a segment σ which moves to child v_i , we check whether it is stored in \mathcal{S}_{v_i} or I_{v_i} : if yes, we delete it and update the minimal segment of v_i in \mathcal{M}_v if necessary. Otherwise, we store σ in the deletion buffer D_{v_i} . If segment buffer \mathcal{S}_v underflows (i.e., $|\mathcal{S}_v| < B/2$), we refill it using segments stored in buffers \mathcal{S}_{v_i} ; the segments moved to \mathcal{S}_v are deleted from \mathcal{S}_{v_i} and all necessary updates in \mathcal{M}_v are performed. This might cause underflowing segment buffers \mathcal{S}_{v_i} for children of v_i ; we handle those in the same way. In case all buffers \mathcal{S}_{v_i} become empty and $|\mathcal{S}_v| < B$, we move the segments from I_v to \mathcal{S}_v until either $|\mathcal{S}_v| = B$ or $|I_v| = 0$.

Bounding the deletion cost: Deletion cost consists of three components:

- (i) Cost for moving segments between buffers: Using the same analysis as for insertions we get that this requires $O(\log_B K / (\epsilon \cdot B^{1-\epsilon}))$ amortized I/Os.

- (ii) Cost due to refilling of buffers \mathcal{S}_v : For a node v with children v_i , while refilling buffer \mathcal{S}_v from \mathcal{S}_{v_i} we perform $O(B^\epsilon)$ I/Os and we move $\Theta(B)$ segments one level higher. Thus the amortized cost of moving a segment up by one level is $O(1/B^{1-\epsilon})$. Since the tree has height $O((1/\epsilon) \cdot \log_B K)$, over a sequence of K deletions the total number of moves of segments by one level is $O((1/\epsilon) \cdot K \cdot \log_B K)$. Thus the total cost due to refilling is at most $O((1/\epsilon B^{1-\epsilon})K \cdot \log_B K)$, which implies that the amortized cost is $O(\log_B K / (\epsilon \cdot B^{1-\epsilon}))$.

A corner case that we did not take into account above is when the total number of segments stored in buffers \mathcal{S}_{v_i} are less than $B/2$. In this case it is not valid that the amortized cost of updating \mathcal{S}_v is $O(B^\epsilon/B)$. To take care of this, we use a simple amortization trick: we double charge all I/Os performed relating to insertions. This way, for each buffer \mathcal{S}_{v_i} there is a saved I/O from the time when segments move from I_v to node v_i . We use this additional saved I/O when \mathcal{S}_{v_i} gets emptied due to the refilling of \mathcal{S}_v .

- (iii) Restructuring requires $O(\frac{\log_B K}{B})$ amortized I/Os, by rebuilding the structure after $K/2$ deletions.

Overall, the amortized deletion cost is $O(\log_B K / (\epsilon \cdot B^{1-\epsilon})) = O(\log_B K / B^{1-\epsilon})$ I/Os.

Queries in the dynamic structure. We now describe how to extend our query algorithm to the dynamic case. In order to ensure that all nodes visited are up-to-date and we do not miss any updates in the insertion/deletion buffers, when moving a pointer from a node u to its child v_i , we flush any deletes in D_u to v_i , i.e. delete segments of D_u that are stored in \mathcal{S}_{v_i} , store the other segments in D_{v_i} and update \mathcal{M}_u if necessary. We then delete any segments found in both I_{v_i} and D_{v_i} . Finally, we compare segments in I_{v_i} with σ_+ (recall this is the first segment hit by ρ^+ among segments considered so far) and, if any segment in I_{v_i} would be hit by ρ^+ before σ_+ we replace σ_+ with it. Clearly this increases the total cost by at most a $O(1)$ factor compared to the static case, thus the query cost is $O((1/\epsilon) \log_B K)$ I/Os.

4 Multislab Structure

In this section we prove Theorem 3. Assume that we are given a set of K non-intersecting segments with endpoints on at most $B^{\epsilon/2} + 1$ vertical lines $l_1, \dots, l_{B^{\epsilon/2}+1}$, for some constant $0 < \epsilon \leq 1/2$. We show that those segments can be stored in a data structure which uses $O(K)$ space, supports vertical ray-shooting queries in $O(\log_B K)$ I/Os, and updates in $O(\log_B K / B^{1-\epsilon})$ amortized I/Os, for $0 < \epsilon \leq 1/2$. This data structure can be constructed in $O((K/B) \log_B K)$ I/Os. We call this data structure a *multislab* structure.

For notational convenience we set $\epsilon' = \epsilon/2$. This way endpoints of the segments lie on at most $B^{\epsilon'} + 1$ vertical lines $l_1, \dots, l_{B^{\epsilon'}+1}$. For $1 \leq i \leq B^{\epsilon'}$, let s_i denote the vertical slab defined by vertical lines l_i and l_{i+1} . We will show that queries are supported in $O(\log_B K)$ I/Os and updates in $O((\log_B K) / B^{1-2\epsilon'})$ I/Os. Theorem 3 then follows.

Total Order. In order to implement the multislab structure we need to maintain an ordering of the segments based on their y -coordinates. Using standard approaches (see e.g. [5, 7]) we can define a partial order for segments that can be intersected by a vertical line. Arge et. al. [8] showed how to extend a partial order into a total order on K segments (not necessarily all intersecting the same vertical line) in $O((K/B) \log_{M/B} \frac{K}{B}) = O((K/B) \log_B K)$ I/Os. We use this total order to create our multislab structure.

The Data Structure. We store the ordered segments in an augmented B-tree \mathcal{T} which supports queries, insertions and deletions. The degree of each node is between $B^{\epsilon'}/2$ and $B^{\epsilon'}$, except the root which might have degree in the range $[2, B^{\epsilon'}]$. Leaves store $\Theta(B)$ elements. For a node $v \in \mathcal{T}$, let \mathcal{T}_v be the subtree rooted at v . Let $v_1, \dots, v_{B^{\epsilon'}}$ be the children of an internal node v . Node v stores the following information:

1. A buffer \mathcal{S}_v of capacity B which contains the highest (according to the total order) segments stored in \mathcal{T}_v which are not stored in any buffer \mathcal{S}_w for an ancestor w of v . In other words, \mathcal{T} together with segments of buffers \mathcal{S}_v form an external memory priority search tree [6].
2. An insertion buffer I_v and a deletion buffer D_v , both storing up to B segments.
3. A list L_v which contains, for each slab s_i , $1 \leq i \leq B^{\epsilon'}$, and each child v_j , $1 \leq j \leq B^{\epsilon'}$, the highest segment (according to the total order) $t_{i,j}$ crossing slab s_i stored in \mathcal{T}_{v_j} .

The data structure satisfies the following invariants: i) for each node $v \in \mathcal{T}$, either $|\mathcal{S}_v| \geq B/2$ or if $|\mathcal{S}_v| < B/2$, then I_v and D_v are empty and all buffers of descendants w of v are empty, ii) for each node v , buffers \mathcal{S}_v, I_v and D_v are disjoint, and iii) for every leaf v , I_v and D_v are empty.

Construction and Space Usage. Overall buffers of each node contain $O(B)$ segments and list L_v contains at most $B^{2\epsilon'} = O(B)$ segments, i.e., they can be stored in $O(1)$ blocks. Thus \mathcal{T} can be stored in $O(K/B)$ blocks, i.e. it requires $O(K)$ space. The structure can be constructed in $O(\frac{K}{B} \log_B K)$ I/Os. If segments are already sorted according to a total order, construction requires $O(K/B)$ I/Os.

Insertions. To insert a new segment σ we need to determine its position in the total order. Clearly, we can not afford to produce a new total order from scratch, as this costs $O((K/B) \log_B K)$ I/Os. Thus, we break σ into at most $B^{\epsilon'}$ *unit segments*, where each segment crosses exactly one slab. In particular, if σ crosses slabs s_ℓ, \dots, s_r , we break it into unit segments $\sigma_\ell, \dots, \sigma_r$, where segment σ_i crosses slab s_i . We call all such unit segments stored in \mathcal{T} *new segments*. The rest of the segments stored in \mathcal{T} are called the *old segments* of \mathcal{T} . Now we can easily update the total order: segment σ_i needs to be compared only with segments crossing slab s_i ; if σ_p and σ_s are the predecessor and successor of σ_i within slab s_i , we locate σ_i in an arbitrary position between σ_p and σ_s in the total order. This way a valid total order is always maintained.

We now describe the insertion algorithm. When segment σ needs to be inserted, we first break it into unit segments $\sigma_\ell, \dots, \sigma_r$. For each segment σ_j , $\ell \leq j \leq r$, we first check whether it should be inserted in the buffer \mathcal{S}_r of the root: if this is the case we store it there; otherwise we store it in the insertion buffer of the root I_r . In case \mathcal{S}_r overflows (i.e. $|\mathcal{S}_r| = B + 1$) we move its last segment (according to the total order) to I_r . Let v be an internal node with children $v_1, \dots, v_{B^{\epsilon'}}$. Each time I_v becomes full, we *flush* it and move the segments to its children v_i , for $1 \leq i \leq B^{\epsilon'}$. For a segment moving from v to v_i , we first check whether it is greater (according to the total order) than the minimum segment stored in \mathcal{S}_{v_i} and if so we store it in \mathcal{S}_{v_i} ; otherwise we store it in buffer I_{v_i} . In case \mathcal{S}_{v_i} overflows (i.e. $|\mathcal{S}_{v_i}| = B + 1$) we move its last segment to I_{v_i} . Also we update information in list L_v if necessary. In case I_{v_i} becomes full, we repeat the same procedure recursively.

When \mathcal{S}_v overflows for some leaf v , we split v into two leaves v_1 and v_2 , as in standard B-trees. Note that this might cause recursive splits of nodes at greater height.

Bounding the insertion cost: To flush a buffer I_v and move segments to buffers of child nodes \mathcal{S}_{v_i} and I_{v_i} , we need to perform $O(B^{\epsilon'})$ I/Os. Since each segment breaks into at most $B^{\epsilon'}$ unit segments, a buffer of size B becomes full after at least $B/B^{\epsilon'} = B^{1-\epsilon'}$ insertions. Thus the amortized cost of moving a segment from a buffer of depth i to depth $i+1$ is $O(B^{\epsilon'}/B^{1-\epsilon'}) = O(1/B^{1-2\epsilon'})$. Since each segment will be eventually stored in a node of depth $O((1/\epsilon') \cdot \log_B K)$, the amortized cost until it gets inserted is $O(\log_B K / (\epsilon' \cdot B^{1-2\epsilon'}))$. The restructuring of \mathcal{T} due to splitting full nodes requires amortized $O(1)$ I/Os, as in standard B-trees. Overall insertions require $O(\log_B K / (\epsilon \cdot B^{1-2\epsilon'})) = O((\log_B K) / B^{1-\epsilon})$ amortized I/Os.

Linear space usage: To avoid increases in space usage due to unit segments, whenever there are $K/B^{\epsilon'}$ new segments, we rebuild the structure. This way the space used is $O(K + (K/B^{\epsilon'}) \cdot B^{\epsilon'}) = O(K)$. This rebuilding requires $O((K/B) \log_B K)$ I/Os, i.e., $O(\log_B K / B^{1-\epsilon'})$ amortized I/Os, thus it does not violate the insertion time bound.

Deletions. The process of deleting a segment, σ , is similar to insertion: we break σ into at most $B^{\epsilon'}$ unit segments $\sigma_\ell, \dots, \sigma_r$ where s_ℓ and s_r are the leftmost and rightmost slabs spanned by σ and apply the deletion procedure for each of those unit segments separately.

The deletion algorithm for a unit segment σ_i is analogous to the one of the left (right) structure of Section 3. For completeness we describe it here. To delete a unit segment σ_i , we first check whether it is stored in the buffers of the root \mathcal{S}_r or I_r ; in this case we delete it. Otherwise, we insert σ_i in the deletion buffer of the root D_r . Let v be an internal node with children $v_1, \dots, v_{B^{\epsilon'}}$. Whenever D_v becomes full we flush it and forward the segments to the corresponding children and repeat the same procedure: For a segment σ which moves to child v_i , we check whether it is stored in \mathcal{S}_{v_i} or I_{v_i} and if this is the case, we delete it and update list L_v if necessary. Otherwise, we store σ_i in the deletion buffer D_{v_i} .

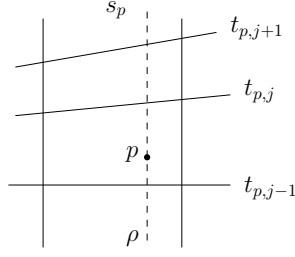
In case segment buffer \mathcal{S}_v underflows (i.e., $|\mathcal{S}_v| < B/2$), we refill it using segments from buffers \mathcal{S}_{v_i} ; segments moved to \mathcal{S}_v are deleted from \mathcal{S}_{v_i} and L_v gets updated (if needed). This might cause underflowing segment buffers \mathcal{S}_{v_i} ; we handle those in the same way. In case all buffers \mathcal{S}_{v_i} become empty and $|\mathcal{S}_v| < B$, we move to \mathcal{S}_v the segments from I_v until either $|\mathcal{S}_v| = B$ or $I_v = 0$. After $K/B^{\epsilon'}$ deletions we rebuild our data structure.

Remark: Note that here we split all segments σ into unit segments $\sigma_\ell, \dots, \sigma_r$. However, the old segments σ are not unit segments and are stored manually in the data structure. However this does not affect our algorithm: whenever the first unit segment σ_i which is a part of σ reaches the node v such that $\sigma \in \mathcal{S}_v$, we delete σ from \mathcal{S}_v and remove σ_i from deletion buffers. The remaining segments σ_j will eventually reach node v and realize that σ is already deleted from \mathcal{S}_v ; at this point σ_j gets deleted.

Bounding the deletion cost: The analysis of the deletion cost is identical to the analysis of deletions in the structure of Section 3. Since each segment breaks into at most $B^{\epsilon'}$ unit segments, we get an amortized deletion cost of $O(\log_B K / B^{1-2\epsilon'}) = O(\log_B K / B^{1-\epsilon})$.

Linear space usage: Similar to insertions, we need to make sure that the total space used is not increasing asymptotically due to the use of at most $B^{\epsilon'}$ unit segments in deletion buffers for each deleted segment σ . The total capacity of deletion buffers is $O(K)$. Since we rebuild the structure after $K/B^{\epsilon'}$ deletions, there are at most $O(K)$ segments stored in deletion buffers, i.e., deletion buffers never get totally full and total space used is $O(K)$.

Queries. Let p be the query point and ρ^+ be the vertical ray emanating from p in the $(+y)$ direction. Let also s_p be the slab containing p . We can find s_p in $O(1)$ I/Os by storing all slab boundaries in a block. We perform a root-to-leaf search and we keep the first segment



■ **Figure 3** Vertical ray-shooting queries in the multislab structure: Query point p is in slab s_p . ρ is the vertical ray emanating from p . While being at node v of \mathcal{T} , to decide in which child to continue our search we examine all minimal segments $t_{p,1}, \dots, t_{p,B^{\epsilon'}}$ stored in list L_v . Among them, the first one hit by ρ is $t_{p,j}$. Thus the search continues at child v_j of v .

σ hit by ρ^+ among segments seen so far. While visiting a node v we do the following: (i) perform a vertical ray-shooting query from p among segments stored in buffers \mathcal{S}_v and I_v , and update σ if necessary (ii) move to the child v_i which contains the successor segment $t_{p,j}$ of p in list L_v (see Figure 3) and (iii) find in I_v (resp. D_v) the segments crossing slab s_p and should be stored (according to the total order) in \mathcal{T}_{v_i} and move them to \mathcal{S}_{v_i} or I_{v_i} (resp. delete them from \mathcal{S}_{v_i} or store it in D_{v_i}). If a segment inserted in D_{v_i} is also stored in I_{v_i} , we delete it from both buffers. Once we reach a leaf v , we first delete from \mathcal{S}_v the segments that are in the deletion buffer of its parent and then we perform ray-shooting query among the segments stored in \mathcal{S}_v and update σ if necessary.

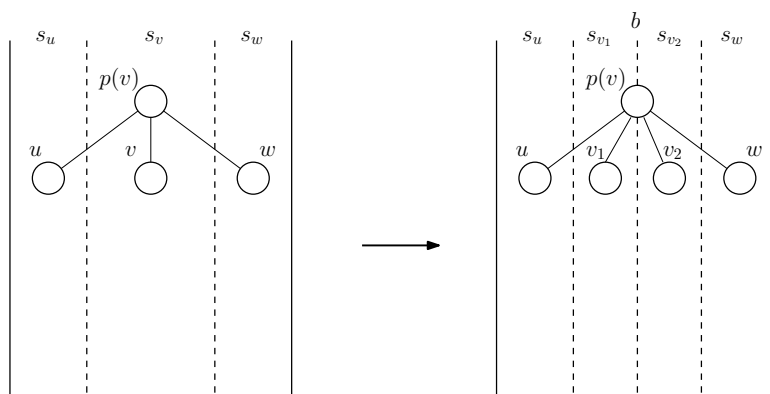
Bounding the query cost: Since we follow a root-to-leaf path, and at each level we need to perform $O(1)$ I/Os, a ray-shooting query is answered in $O((1/\epsilon') \cdot \log_B K)$ I/Os.

5 Counting the Restructuring Cost

In Section 2 we proved the Theorem 1 (query and update bounds of the overall structure) without taking into account the cost of restructuring the interval tree \mathcal{I} due to insertions that cause leaves to become full. In this section we show that Theorem 1 holds while taking into account the restructuring of \mathcal{I} as well.

When a leaf becomes full we need to split it. This split in turn might cause the split of the parent and possibly continue up the tree, thus causing some part of the tree \mathcal{I} to need rebalancing. While rebalancing, we need to perform updates in the secondary structures so that they are adjusted with the updated nodes of the interval tree \mathcal{I} . In this section, we show that we can slightly modify our data structure such that all updates in secondary structures can be performed in $O(\frac{\log_B N}{B^{1-\epsilon}})$ amortized I/Os. This implies that Theorem 1 holds.

Our Approach. We use a variant of the weight-balanced B^ϵ -tree of [9]. Each leaf stores at most B segment endpoints. Let v be a node at height $h - 1$ with parent $p(v)$. Node $p(v)$ stores $w_v = \Theta(B \cdot B^{\epsilon h})$ elements in its subtree $\mathcal{I}_{p(v)}$. We will show that if node v splits, then we can perform all updates needed in the secondary structures in $O(w_v/B^{1-\epsilon})$ I/Os. This implies that a split requires amortized $O(1/B^{1-\epsilon})$ I/Os, since after a restructuring, there should be at least $\Omega(w_v)$ insertions in $\mathcal{I}_{p(v)}$ until the next split is needed. Since each insertion can cause $O(\log_B N)$ splits, we get an amortized restructuring cost of $O(\frac{\log_B N}{B^{1-\epsilon}})$ I/Os for insertion.

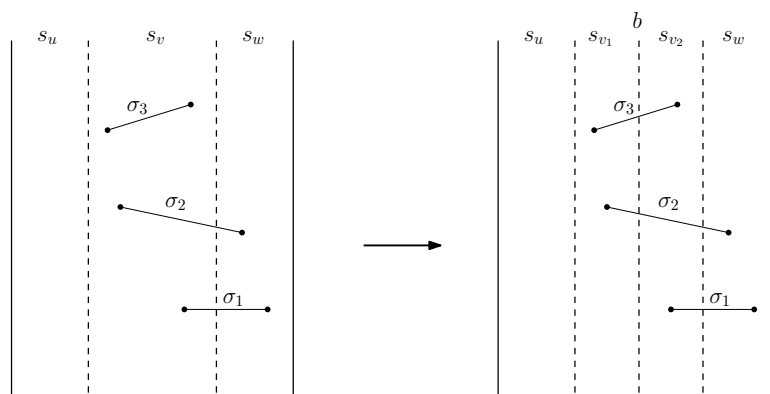


■ **Figure 4** Splitting a node v into v_1 and v_2 : slab s_v is divided into slabs s_{v_1} and s_{v_2} with boundary b .

Splitting a node. Node v splits into two new nodes v_1 and v_2 . The slab s_v of v is divided into two slabs s_{v_1}, s_{v_2} with slab boundary b ; see Figure 4. To capture this change and update our data structure, we need to perform updates in the secondary structures of $p(v)$ and construct the secondary structures for v_1, v_2 . We describe these updates in detail and show that they can be performed in $O(w_v/B^{1-\epsilon})$ I/Os. In our analysis we use the fact that all secondary structures (multislab and left/right) storing K segments can be scanned in $O(K/B)$ I/Os.

Updates in secondary structures of $p(v)$. We begin with the construction of left/right structures for v_1 and v_2 using the previous left/right structures for v . We describe the creation of left structures L_{v_1} and L_{v_2} for v_1 and v_2 , respectively, and the right structures are symmetric. Segments that were stored in L_v and do not cross b (like segment σ_1 in Figure 5) are stored in L_{v_2} ; segments of L_v that cross b (see segment σ_2 in Figure 5) are stored in L_{v_1} . To identify if a segment is stored in L_{v_1} or L_{v_2} we just need to scan L_v , which takes $O(w_v/B)$ I/Os. Moreover, there are some additional segments that need to be stored in left/right structures of $p(v)$: the segments that are strictly inside the slab of v (i.e. they were stored in secondary structures of v) and cross b ; see e.g. segment σ_3 in Figure 5. For those segments, their left subsegments are stored in L_{v_1} and their right subsegments in R_{v_2} . To find such segments we need to scan all secondary structures stored at v . Since each secondary structure can be scanned in $O(w_v/B)$ I/Os and there are $O(B^\epsilon)$ structures stored in each node, all this takes $O((w_v/B) \cdot B^\epsilon) = O(w_v/B^{1-\epsilon})$ I/Os.

We now proceed to the updates of the multislab structure of $p(v)$. Here, we just need to add some segments to the previous multislab structure. The new segments are the segments of L_v that cross b which are not already stored in the multislab (and symmetrically, the segments of R_v that cross b and are not yet in the multislab). For an example, see segment σ_2 in Figure 5; before it was not stored in the multislab and now we store its middle subsegment. Note that the middle subsegment is a unit segment (i.e. crosses exactly one slab) thus we don't need to compute a new total order; we can find its position in the total order by comparing it only with segments that cross slab s_{v_2} . All those segments that need to be added can be found by scanning L_v and R_v in $O(w_v/B)$ I/Os. Insertions in the multislab of $p(v)$ require $O((\log_B w_v)/B^{1-\epsilon}) = O(w_v/B)$ I/Os. Also, all information stored in nodes of the multislab structure can be updated in $O(w_v/B)$ I/Os. Overall, all updates in the multislab structure of v are performed in $O(w_v/B)$ I/Os.



■ **Figure 5** Example of segments that get stored in different secondary structures after a split. Segment σ_1 was stored in L_v and, after the split, gets stored in L_{v_2} . Segment σ_2 was stored in L_v ; following the split its left subsegment is stored in L_{v_1} and its middle subsegment in the multislab structure of $p(v)$. Segment σ_3 was previously stored in secondary structures of v , and after the split it should be stored in structures L_{v_1} and R_{v_2} of $p(v)$.

Construct secondary structures for v_1 and v_2 . The left and right structures for each child slab of v_1 and v_2 will be based on the left/right structure of the same slab in v just by removing the segments that cross b (which are assigned to $p(v)$ as we explained above). Similarly, segments that cross b are excluded from the multislab structure.

We start with the construction of left/right structures of v_1 and v_2 . We describe the left and the right is symmetric. For each slab s_k of v , $1 \leq k \leq B^\epsilon$ we scan the left list L_k ; the segments that do not cross b remain in L_k and the others are deleted. All this takes $O((w_v/B) \cdot B^\epsilon) = O(w_v/B^{1-\epsilon})$ I/Os.

Finally we create the multislab structures for v_1 and v_2 . Again, we need to scan the multislab of v and delete the segments that cross b , which takes $O(w_v/B)$ I/Os. Then we need to build the multislabs of v_1 and v_2 out of the remaining segments. Since all segments are already sorted according to a total order, this can be done in $O(w_v/B)$ I/Os.

References

- 1 Pankaj K. Agarwal, Lars Arge, Gerth Stølting Brodal, and Jeffrey Scott Vitter. I/O-Efficient Dynamic Point Location in Monotone Planar Subdivisions. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–20, 1999. URL: <http://dl.acm.org/citation.cfm?id=314500.314525>.
- 2 Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.
- 3 Lars Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/s00453-003-1021-x.
- 4 Lars Arge, Gerth Stølting Brodal, and Loukas Georgiadis. Improved Dynamic Planar Point Location. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–314, 2006. doi:10.1109/FOCS.2006.40.
- 5 Lars Arge, Gerth Stølting Brodal, and S. Srinivasa Rao. External Memory Planar Point Location with Logarithmic Updates. *Algorithmica*, 63(1-2):457–475, 2012. doi:10.1007/s00453-011-9541-2.
- 6 Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On Two-Dimensional Indexability and Optimal Range Search Indexing. In *PODS*, pages 346–357. ACM Press, 1999.

- 7 Lars Arge and Jan Vahrenhold. I/O-efficient dynamic planar point location. *Comput. Geom.*, 29(2):147–162, 2004. doi:10.1016/j.comgeo.2003.04.001.
- 8 Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-Memory Algorithms for Processing Line Segments in Geographic Information Systems. *Algorithmica*, 47(1):1–25, 2007. doi:10.1007/s00453-006-1208-z.
- 9 Lars Arge and Jeffrey Scott Vitter. Optimal External Memory Interval Management. *SIAM J. Comput.*, 32(6):1488–1508, 2003. doi:10.1137/S009753970240481X.
- 10 Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic Point Location in General Subdivisions. *J. Algorithms*, 17(3):342–380, 1994. doi:10.1006/jagm.1994.1040.
- 11 Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189, 1972. doi:10.1007/BF00288683.
- 12 Jon Louis Bentley. Decomposable Searching Problems. *Inf. Process. Lett.*, 8(5):244–251, 1979. doi:10.1016/0020-0190(79)90117-0.
- 13 Gerth Stølting Brodal. External Memory Three-Sided Range Reporting and Top-k Queries with Sublogarithmic Updates. In *33rd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 23:1–23:14, 2016. doi:10.4230/LIPIcs.STACS.2016.23.
- 14 Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, 2003. URL: <http://dl.acm.org/citation.cfm?id=644108.644201>.
- 15 Timothy M. Chan and Yakov Nekrich. Towards an Optimal Method for Dynamic Planar Point Location. In *IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–409, 2015. doi:10.1109/FOCS.2015.31.
- 16 Bernard Chazelle. Computational Geometry for the Gourmet: Old Fare and New Dishes. In *Automata, Languages and Programming, 18th International Colloquium (ICALP)*, pages 686–696, 1991. doi:10.1007/3-540-54233-7_174.
- 17 Bernard Chazelle. Computational geometry: a retrospective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing (STOC)*, pages 75–94, 1994. doi:10.1145/195058.195110.
- 18 Siu-Wing Cheng and Ravi Janardan. New Results on Dynamic Planar Point Location. *SIAM J. Comput.*, 21(5):972–999, 1992. doi:10.1137/0221057.
- 19 Yi-Jen Chiang and Roberto Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Int. J. Comput. Geometry Appl.*, 2(3):311–333, 1992.
- 20 Herbert Edelsbrunner and Hermann A. Maurer. On the Intersection of Orthogonal Objects. *Inf. Process. Lett.*, 13(4/5):177–181, 1981. doi:10.1016/0020-0190(81)90053-3.
- 21 Michael T. Goodrich and Roberto Tamassia. Dynamic Trees and Dynamic Point Location. *SIAM J. Comput.*, 28(2):612–636, 1998.
- 22 J. Ian Munro and Yakov Nekrich. Dynamic Planar Point Location in External Memory. In *SoCG*, volume 129 of *LIPIcs*, pages 52:1–52:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.
- 23 Eunjin Oh and Hee-Kap Ahn. Point Location in Dynamic Planar Subdivisions. In *34th International Symposium on Computational Geometry, SoCG 2018*, pages 63:1–63:14, 2018. doi:10.4230/LIPIcs.SoCG.2018.63.
- 24 Jack Snoeyink. Point Location. In *Handbook of Discrete and Computational Geometry, Second Edition*, pages 767–785. Chapman & Hall/CRC, 2004. doi:10.1201/9781420035315.pt4.
- 25 Jeffrey Scott Vitter. External memory algorithms and data structures. *ACM Comput. Surv.*, 33(2):209–271, 2001. doi:10.1145/384192.384193.
- 26 Jeffrey Scott Vitter. Algorithms and Data Structures for External Memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006. doi:10.1561/04000000014.

A

 Queries in the Left and Right Structures

In this Section we give further details on the left (right) structure which were omitted from Section 3.

Queries. We begin with the queries and we show the correctness of the query algorithm of the static left (right) structure.

Correctness: The correctness of the query algorithm follows from the next lemma. For a node $v \in \mathcal{T}$ let S_v be the set of segments stored in buffers \mathcal{S} in \mathcal{T}_v .

► **Lemma 4.** *Assume that at the end of the i th step of the query algorithm, either v_+ or v_- is defined. Then σ_+ is the first segment hit by ρ^+ among the segments of $\mathcal{L} - (S_{v_-} \cup S_{v_+})$.*

Proof. We prove the lemma by induction.

Induction Base: At the end of the first step, v_+ and v_- are children of the root r and σ_+ is the first segment hit by ρ^+ among all segments stored at the root (in \mathcal{S}_r and \mathcal{M}_r). By definition of $v_s = v_+$, for any child of the root v with higher y -range than v_+ , σ_+ is below all segments of S_v . Similarly, for any child of the root v' with smaller y -range than v_- (if v_- exists), there is no segment in $S_{v'}$ hit by ρ^+ (since there exists a segment in \mathcal{S}_{v_-} hit by ρ^-). Finally, for any child v'' of the root whose y -range is between the range of v_- and v_+ , by definition of v_+ , there is no segment in $S_{v''}$ hit by ρ^+ . We conclude that σ_+ is the first segment hit by ρ^+ among the segments in $\mathcal{L} - (S_{v_-} \cup S_{v_+})$.

Inductive Step: Assume the lemma holds at the end of step i , i.e. we have at least one of v_+ and v_- at level i and σ_+ is the first segment hit by ρ^+ among all segments in $\mathcal{L} - (S_{v_+} \cup S_{v_-})$.

During $(i+1)$ th step we ray-shoot on ρ^+ among segments stored in $\mathcal{S}_{v_+}, \mathcal{S}_{v_-}, \mathcal{M}_{v_+}$ and \mathcal{M}_{v_-} , and update σ_+ if necessary. Let v_s be the node containing the first segment hit by ρ^+ among \mathcal{M}_{v_+} and \mathcal{M}_{v_-} (if such a segment exists). Let also v_p be the node containing the first segment hit by ρ^- among \mathcal{M}_{v_+} and \mathcal{M}_{v_-} (if such a segment exists).

By definition of v_s , for any node v which is a child of v_- or v_+ with higher y -range than v_s , σ_+ is below all segments of S_v . Similarly, for a node v' which is a child of v_- or v_+ with smaller y -range than v_p (if v_p exists), there is no segment of $S_{v'}$ hit by ρ^+ (since there exists a segment in \mathcal{S}_{v_p} hit by ρ^-). Finally, for any child v'' of v_- or v_+ whose y -range is between the range of v_- and v_+ , by definition of v_+ , there is no segment in $S_{v''}$ hit by ρ^+ .

Recall that by the induction hypothesis σ_+ at the end of the previous step was the first segment hit by ρ^+ among segments of $\mathcal{L} - (S_{v_+} \cup S_{v_-})$. Now we updated σ_+ and showed that there is no segment hit by ρ^+ before σ_+ in any subtree other than \mathcal{T}_{v_s} or \mathcal{T}_{v_p} . We conclude that σ is the first segment hit by ρ^+ among the segments in $\mathcal{L} - (S_{v_s} \cup S_{v_p})$. Since at the end of the $(i+1)$ th step we set $v_- = v_p$ and $v_+ = v_s$, the lemma follows. ◀

We now explain how Lemma 4 implies the correctness of the query algorithm. To see that, let i be the last level where either v_+ or v_- is defined; at the beginning of the query algorithm at level i , σ_+ is the first segment hit by ρ^+ among segments of $\mathcal{L} - (S_{v_-} \cup S_{v_+})$. Moreover at the end of this step, both v_s and v_p are not defined, i.e., for each child v of v_- or v_+ there is no segment in S_v hit by ρ^+ before σ_+ . Since $S_{v_-} \cup S_{v_+} = S_{v_-} \cup S_{v_+} \cup (\cup_v S_v)$, we get that σ_+ is the first segment hit by ρ^+ among segments of $\mathcal{L} - (S_{v_-} \cup S_{v_+})$. By checking all segments of $\mathcal{S}_{v_-} \cup \mathcal{S}_{v_+}$ and updating σ_+ if necessary, we make sure that σ_+ is the first segment hit by ρ^+ among segments of \mathcal{L} .