An Optimal Dynamic Interval Stabbing-Max Data Structure?*

Pankaj K. Agarwal† Lars Arge‡ Ke Yi§

Abstract

In this paper we consider the dynamic stabbing-max problem, that is, the problem of dynamically maintaining a set S of n axis-parallel hyper-rectangles in $\mathbb{R}^d$, where each rectangle $s \in S$ has a weight $w(s) \in \mathbb{R}$, so that the rectangle with the maximum weight containing a query point can be determined efficiently. We develop a linear-size structure for the one-dimensional version of the problem, the *interval stabbing-max* problem, that answers queries in worst-case $O(\log n)$ time and supports updates in amortized $O(\log n)$ time. Our structure works in the pointer-machine model of computation and utilizes many ingredients from recently developed external memory structures. Using standard techniques, our one-dimensional structure can be extended to higher dimensions, while paying a logarithmic factor in space, update time, and query time per dimension. Furthermore, our structure can easily be adapted to external memory, where we obtain a linear-size structure that answers queries and supports updates in $O(\log_B n)$ I/Os, where B is the disk block size.

1 Introduction

In this paper we consider data structures for the *stabbing-max* problem (also sometimes called the *rectangle intersection with priorities* problem). That is, the problem of dynamically maintaining a set S of n axis-parallel hyper-rectangles in $\mathbb{R}^d$, where each rectangle $s \in S$ has a weight $w(s) \in \mathbb{R}$, so that the rectangle with the maximum weight containing a query point can be determined efficiently. This problem has numerous applications in many areas, including databases and networking; see e.g. [1, 9, 12] and the references therein. We are mainly interested in the one-dimensional version of the problem, the *interval stabbing-max* problem, for which we develop an optimal structure.

Previous work. Because of its many applications, the stabbing-max problem has been studied extensively in several communities, including the algorithms, database and networking communities.

For the interval stabbing-max problem, if deletion of intervals is not allowed, then a linear-size balanced search tree on the interval endpoints can be augmented so that queries and insertions can both be supported in $O(\log n)$ time in the (internal memory) pointer-machine model [13]. In the external memory model [2], relevant when processing massive datasets and therefore often considered in the database literature, a similar linear-size data structure with $O(\log_B n)$ I/O query and insertion time is also known [14]; here B is the number of elements that can be transfered between disk and main memory in one I/O (internal computation is ignored). The problem becomes considerably harder when deletions are allowed. Intuitively, deletions are hard because after deleting a maximal weight interval, all the remaining intervals are candidates for being a maximal interval; while performing an insertion, the new interval is the only new candidate. However, by modifying interval trees [5, 8] one can obtain linear-size internal and external structures that support general updates in $O(\log n)$ time (resp. $O(\log_B n)$ I/Os), but require $O(\log^2 n)$ time (resp. $O(\log_B^2 n)$ I/Os) to answer a query [1, 9]. Recently, Kaplan et al. [9] improved the query bound in the internal memory model to the optimal $O(\log n)$ time at the cost of increasing the deletion time to $O(\log n \log\log n)$. A similar improved structure is not known in the external memory model. If any two input intervals are either disjoint or nested, Kaplan et al. [9] improved the deletion time to $O(\log n)$. They posed the existence of a data structure with a similarly improved bound for an arbitrary set of intervals as a major open problem.

Using standard techniques based on segment trees, the one-dimensional stabbing-max structures can be extended to higher dimensions, while paying a logarithmic factor in space, update and query time or I/O per dimension [1, 9]. This leads to an $O(n \log n)$-size

*The first author is supported by US National Science Foundation (NSF) grants CCR-00-86013, EIA-98-70724, EIA-01-31905, and CCR-02-04118, and by a grant from the U.S.–Israel Binational Science Foundation, and the last two authors by NSF grants EIA–9972879, CCR–9984099, EIA–0112849, and INT–0129182 and by a Rømer Scholarship from the Danish National Science Research Council.

†Department of Computer Science, Duke University, Durham, NC 27708, USA. Email: pankaj@cs.duke.edu

‡Department of Computer Science, University of Aarhus, Aarhus, Denmark. This work was done while the author was at Duke University. Email: large@daimi.au.dk

§Department of Computer Science, Duke University, Durham, NC 27708, USA. Email: yike@cs.duke.edu

structure in $\mathbb{R}^2$ that supports queries and insertions in $O(\log^2 n)$ time and deletions in $O(\log^2 n \log\log n)$ time in internal memory, and an $O(n \log_B n)$-size structure supporting queries in $O(\log_B^3 n)$ I/Os and updates in $O(\log_B^2 n)$ I/O in external memory. In external memory, Agarwal et al. [1] developed an alternative structure in $\mathbb{R}^2$ using $O((n/B)\log_B \log_B n)$ space that answers queries using $O(\log_B^5 n)$ I/Os. The structure supports deletions in $O(\log_B^2 n)$ I/Os and insertions in $O(\log_B^2 n \cdot \log_{M/B} \log_B n)$ I/Os, where M is the size of the main memory. They also developed a linear-size static structure supporting queries in $O(\log_B^4 n)$ I/Os. In internal memory, Kaplan et al. [9] developed an improved structure for nested rectangles that uses $O(n \log n)$ space and supports queries and updates in $O(\log n)$ and $O(\log^2 n)$ time, respectively.

Our results. The main result of this paper is a linear-size data structure for the (general) interval stabbing-max problem in the pointer-machine model that answers queries and supports updates (insertions as well as deletions) in $O(\log n)$ time. The query bound is worst-case while the update bounds are amortized. Thus we settle the open problem posed by Kaplan et al. [9]. Furthermore, our structure can easily be adapted to external memory, where we obtain an $O(n/B)$-size structure that answers queries and supports updates in $O(\log_B n)$ I/Os. Finally, as previously, our structures can be extended to higher dimensions using the standard segment-tree techniques, while paying a logarithmic factor in space, update time, and query time or I/O per dimension.

The main idea we use to improve the deletion time to $O(\log n)$ is to increase the fan-out of the base tree of the multi-level structure of Kaplan et al. [9] from 2 to $\Theta(\sqrt{\log n})$. The use of large non-constant fan-out base trees is common in external memory, where a fan-out of $\Theta(B^{1/c})$ is often used for some constant $c \geq 1$. Large fan-out trees have also been used in several efficient internal structures (e.g. [7, 11]). The advantage of a larger fan-out base tree is, of course, that it has small height ($O(\log n/\log\log n)$ in our case), which allows us to use more time ($O(\log\log n)$ in our case) to query or update secondary structures attached to the nodes of a root-leaf path in the base tree. However, the increased fan-out also introduces many difficulties, which we overcome by utilizing ideas from several recently developed external structures [3–5]. We believe that the idea of utilizing techniques from external memory structures is of independent interest and may lead to improvement of other internal structures.

While our interval stabbing-max structure settles the open problem posed by Kaplan et al. [9], it remains an open problem if our structure is truly optimal. Our structure is optimal in the sense that any sequence of n operations take $\Theta(n \log n)$ time, and it is the first data structure that attains this optimal bound for a sequence of mixed operations. A query obviously has to take $\Omega(\log n)$ time in the comparison model. By using an adversary argument [6], one can also show that the insertion cost has to $\Omega(\log n)$ if a query is required to take $O(\log n)$ time. However to our knowledge, known lower bounds do not exclude the existence of a structure with $O(\log n)$ query and insertion bounds but with an $o(\log n)$ deletion bound. We conjecture that such a structure does not exist.

The paper is organized as follows. In Section 2 we first describe our structure with the assumption that the endpoints of all intervals belong to a fixed set of $O(n)$ points. This allows us to disregard rebalancing of the base tree in our multi-level structure. The assumption is then removed in Section 3 and describe how to rebalance the base tree. Finally we mention some extensions to our basic structure in Section 4.

2 Fixed Endpoint-set 1D Structure

In this section we describe our interval stabbing-max structure, assuming that the endpoints of all intervals that are ever in the structure belong to a fixed set of $O(n)$ points. For simplicity, we assume that no two intervals have the same endpoint, and we do not distinguish between an interval and its weight; for example, we use "maximum interval" to refer to the interval with the maximum weight.

2.1 The base tree As earlier structures, our structure is based on the interval tree [8]. An interval tree on a set of n intervals consists of a balanced binary base tree $\mathcal{T}$ on the $O(n)$ endpoints of the intervals, with the intervals stored in secondary structures of the nodes of $\mathcal{T}$. With each node v of $\mathcal{T}$ we associate a *range* σ_v: the range σ_v associated with a leaf z consists of the interval formed by two consecutive endpoints; if v is an internal node with w and z as it children, then $\sigma_v = \sigma_w \cup \sigma_z$. We also associate a point x_v with each internal node v, which is the common boundary point of σ_w and σ_z. An input interval $s \in S$ is associated with the highest node v such that $x_v \in s$ (i.e., $x_{p(v)} \notin s$, where $p(v)$ denotes v's parent). The subset $S_v \subseteq S$ of intervals associated with v is stored in two secondary structures, namely, a dynamic height-balanced tree sorted by the left endpoints of intervals in S_v, and another sorted by the right endpoints. At each node of these secondary trees we store the maximum interval in the subtree rooted at that node. Using these two secondary trees, the maximum interval in S_v containing a query point q can

be found in $(\log n)$ time. Thus a stabbing-max query for a query point q can be answered in $O(\log^2 n)$ time by searching down $\mathcal{T}$ for the leaf z such that $q \in \sigma_z$, computing the maximum interval in S_v at each node v along this path, and then returning the maximum of these $O(\log n)$ intervals. An interval s can be inserted or deleted in $O(\log n)$ time by first finding the node v of $\mathcal{T}$ with which s is associated and then updating the two secondary trees associated with v. The query time can be improved to $O(\log n \log\log n)$, at the cost of increasing the update time to $O(\log n \log\log n)$, using dynamic fractional cascading [10].

Intuitively, the idea in the interval max-stabbing structure of Kaplan et al. [9] is to replace the secondary structures of the interval tree with structures that can answer a max-stabbing query in $O(1)$ time, leading to an $O(\log n)$ query bound. At a node v, let L_v be the set of intervals in $\bigcup S_u$, where u is an ancestor of v, whose left endpoints lie in the range associated with the left child of v. The set R_v is defined similarly with respect to right endpoints. In the structure by Kaplan et al. [9], L_v and R_v are implicitly stored in secondary structures at v, such that the maximum interval in them can be found in $O(1)$ time. By following the path to the leaf z such that σ_z contains the query point, a stabbing-max query can now be answered in $O(\log n)$ time. However, a deletion now requires $O(\log n \log\log n)$ time rather than the desired $O(\log n)$ time, since when deleting an interval $O(\log\log n)$ time is required to update each of the $O(\log n)$ secondary structures on a search path in $\mathcal{T}$. See the original paper for details [9].

Our structure. Intuitively, the idea in our structure is to increase the fan-out of the base tree $\mathcal{T}$ to $\log^c n$, thereby decreasing its height to $O(\log n/\log\log n)$. This allows us to spend $O(\log\log n)$ time at each node on a search path and still obtain an $O(\log n)$ bound. Of course, many complications need to be overcome to make this idea work.

More precisely, our interval max-stabbing data structure consists of a balanced base tree $\mathcal{T}$ over $n/\log n$ leaves with fan-out $f = \sqrt{\log n}$; each leaf contains $\log n$ consecutive interval endpoints. The f children of a node v are organized in a balanced tree such that they can be searched in $O(\log f) = O(\log\log n)$ time. As in the binary case, we associate a range σ_v with each node v of $\mathcal{T}$; σ_v is the union of all ranges associated with the children $v_1, v_2, \ldots, v_f$ of v. The range σ_v is divided into f sub-ranges by the ranges associated with the children of v, which we refer to as *slabs*. We refer to the boundaries of subranges as *slab boundaries* and use $b_i(v)$ to denote the slab boundary between $\sigma_{v_{i-1}}$ and σ_{v_i}. Furthermore, we define a *multislab* to be a continuous range of slabs, that is, $\sigma_v[i:j] = \bigcup_{l=i}^{j} \sigma_{v_l}$ is the multislab consisting of slabs σ_{v_i} through σ_{v_j}, for $1 \le i \le j \le f$. There are $\binom{f}{2} = O(\log n)$ multislabs in v. For example, in Figure 1 node v has five slabs $\sigma_{v_1}, \ldots, \sigma_{v_5}$, which are separated by six boundaries $b_1(v), \ldots, b_6(v)$; note that $b_1(v)$ and $b_6(v)$ are also slab boundaries in the parent $p(v)$ of v. The interval s spans the multislab $\sigma_v[2:3]$.

Intervals in S are associated with the nodes of $\mathcal{T}$ in a way similar to the binary case: An interval s is associated with v if s crosses at least one slab boundary of v but none of the boundaries at the parent of v; if an interval has both endpoints in a leaf z, i.e., it does not cross any slab boundaries, it is stored at z. As earlier, let $S_v \subseteq S$ be the subset of intervals associated with v. Since the secondary structures of v, described below, may contain a superset of S_v, we also store the intervals S_v in a doubly linked list at v. We keep pointers between an interval in S_v and the leaves containing its two endpoints. Note that overall the base tree $\mathcal{T}$ and the lists occupy $O(n)$ space. It remains to describe the secondary structures associated with each node v.

First consider the set S_z of intervals associated with a leaf z of $\mathcal{T}$. For a query point $q \in \sigma_z$, we can easily answer a stabbing-max query on S_z in $O(\log n)$ time simply by scanning the $O(|S_z|) = O(\log n)$ intervals in S_z. Similarly, we can also delete an interval from S_z or insert an interval into S_z in $O(\log n)$ time. Thus we do not need secondary structure for the intervals associated with the leaves of $\mathcal{T}$ and will not consider these intervals in the following.

Now consider the set S_v of intervals associated with one of the $O(n/\log^{3/2} n)$ internal nodes v of $\mathcal{T}$. As in the external interval tree by Arge and Vitter [5], we imagine partitioning each such interval $s \in S_v$ into two or three pieces: Suppose s has its left endpoint in slab σ_{v_i} and right endpoint in slab σ_{v_j} with $j > i+1$. We break s at $b_{i+1}(v)$ and $b_j(v)$ to obtain a left interval s^l in σ_{v_i}, a middle interval s^m spanning the multislab $\sigma_v[i+1, j-1]$ (the largest multislab spanned by s),

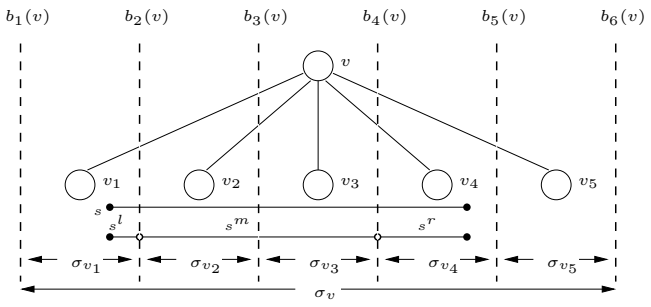

Figure 1: An internal node v in the base tree $\mathcal{T}$.

and a right internal s^r in σ_{v_j}. If $j = i + 1$ we break s at $b_{i+1}(v)$ $(= b_j(v))$ to obtain s^l and s^r. Refer to Figure 1 where s is broken into s^l in σ_{v_1}, s^m spanning $\sigma_v[2:3]$, and s^r in σ_{v_4}. We use S_v^l, S_v^m and S_v^r to denote the set of left, middle, and right intervals obtained from the intervals in S_v, respectively. Furthermore, let $S^l = \bigcup_{v \in \mathcal{T}} S_v^l$, $S^r = \bigcup_{v \in \mathcal{T}} S_v^r$, and $S^m = \bigcup_{v \in \mathcal{T}} S_v^m$ be the sets of all left, right, and middle intervals, respectively. In the following we describe separate secondary structures associated with the internal nodes of $\mathcal{T}$ for storing S^l, S^m, and S^r. Although we construct a separate secondary structure for each node $v \in \mathcal{T}$, the secondary structure associated with v is not constructed on S_v^l, S_v^m, S_v^r. Instead, we use a different criterion, described below, to decide the subsets of S^l, S^m, S^r on which we build the secondary structure associated with v. By querying the secondary structures of a root-leaf path, interval stabbing-max queries can be answered on each of these sets in $O(\log n)$ time. This leads to an overall query bound of $O(\log n)$, since the answer to such a query is either found in a leaf or in S^l, S^m, and S^r. Globally, the secondary structures for each of the sets S^l, S^r and S^m can also be updated in $O(\log n)$ time, leading to an $O(\log n)$ update bound since an update involves either updating a leaf or updating S^l, S^r, and possibly S^m. In Section 2.2 below we describe the secondary structures for S^m. In Section 2.3 we describe the structures for S^l. The structures for S^r are symmetric to those for S^l and thus are not described.

2.2 Middle intervals S^m The secondary structures storing S^m consist of a structure $\mathbb{M}_v$ for each internal node v of $\mathcal{T}$. The structure $\mathbb{M}_v$ stores exactly the set S_v^m. We first describe $\mathbb{M}_v$ and how it can be queried and updated. Then we describe the global query and update procedures.

$\mathbb{M}_v$ structure. The $\mathbb{M}_v$ structure consists of $O(\log n)$ max-heaps: A *multislab max-heap* for each of the $O(\log n)$ multislabs, as well as a *slab max-heap* for each of the $\sqrt{\log n}$ slabs. The multislab max-heap for multislab $\sigma_v[i:j]$ contains all intervals in S_v^m that exactly span $\sigma_v[i:j]$. The slab max-heap for slab σ_{v_l} contains the maximum interval from each of the $O(\log n)$ multislab max-heaps corresponding to multislabs that span σ_{v_l} (i.e., the multislab max-heaps corresponding to $\sigma_v[i:j]$ for $1 \leq i \leq l \leq j \leq f$). Since each interval in S_v^m is stored in exactly one multislab max-heap, all these multislab heaps use linear space in total. Each slab max-heap uses $O(\log n)$ space for a total of $O(\log^{3/2} n)$ space. Thus overall $\mathbb{M}_v$ uses $O(|S_v^m| + \log^{3/2} n)$ space. To construct $\mathbb{M}_v$, we first compute for each interval in S_v^m the multislab it belongs to. This can easily be done in $O(\log \log n)$ time. Then we construct each of the heaps. Since constructing a heap takes linear time, the total construction cost is $O(|S_v^m| \log \log n + \log^{3/2} n)$.

Given a query point $q \in \sigma_v$, we find the maximum interval in S_v^m containing q as follows: We first find the slab that contains q and then we report the maximum interval stored in the corresponding slab max-heap. This takes $O(\log \log n)$ time. To insert or delete an interval in S_v^m we first update the relevant multislab max-heap in $O(\log n)$ time; if the maximum interval of the multislab max-heap changes, we also update the $O(\sqrt{\log n})$ affected slab max-heaps in $O(\sqrt{\log n} \cdot \log \log n) = O(\log n)$ time.

LEMMA 2.1. *The set of middle intervals S_v^m of a node v can be stored in a data structure $\mathbb{M}_v$ using $O(|S_v^m| + \log^{3/2} n)$ space, such that a stabbing-max query can be answered in $O(\log \log n)$ time. The structure can be constructed in $O(|S_v^m| \log \log n + \log^{3/2} n)$ time and updated in $O(\log n)$ time.*

Querying and updating S^m. It is easy to see that all we need to do to answer a stabbing-max query q on S^m is to query the $\mathbb{M}_v$ structures of all the $O(\log n / \log \log n)$ internal nodes v on the path from the root of $\mathcal{T}$ to the leaf containing q, and then return the maximum of these interval. Since we use $O(\log \log n)$ time in each node on the path (Lemma 2.1), we answer a query in $O(\log n)$ time in total. To perform an insertion or deletion on S^m we simply search down $\mathcal{T}$ for the relevant node v in $O(\log n)$ time. Then we update $\mathbb{M}_v$ in $O(\log n)$ time (Lemma 2.1).

Finally, note that even though the size of $\mathbb{M}_v$ is $O(|S_v^m| + \log^{3/2} n)$, the overall size of all the $\mathbb{M}_v$ structures is $O(n)$ since the number of internal nodes in $\mathcal{T}$ (and thus $\mathbb{M}_v$ structures) is $O(n / \log^{3/2} n)$.

LEMMA 2.2. *Using linear space $O(n)$, the set S^m of middle intervals can be stored in secondary structures of $\mathcal{T}$ so that stabbing-max queries can be answered and updates can be performed in $O(\log n)$ time.*

2.3 Left intervals S^l Like the secondary structures for the middle intervals, the secondary structures for the left segments consist of a structure $\mathbb{L}_v$ associated with each internal node v of $\mathcal{T}$. However, unlike for the middle intervals, the structure $\mathbb{L}_v$ is not constructed on the set S_v^l. Instead, similar to the Kaplan et al. structure [9], it stores some of the intervals that belong to $\bigcup_u S_u^l$, where the union is taken over the ancestors of v. We first describe a static version of $\mathbb{L}_v$ and then show how to make it dynamic.

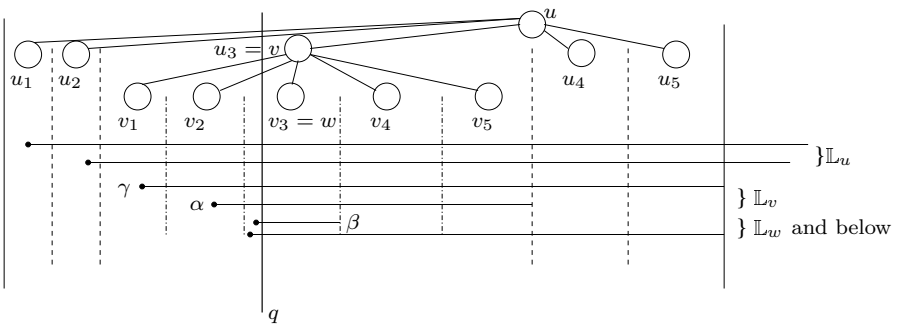

Figure 2: Answering a stabbing-max query on S^l using the $\mathbb{L}_v$ structures.

The static structure. For any two nodes u, v of $\mathcal{T}$, where u is an ancestor of v, let

$$\Psi(u, v) = \{s^l \mid s \in S_u \text{ and } s\text{'s left endpoint is in } \sigma_v\},$$

i.e., $\Psi(u, v)$ is the set of intervals associated with u with left endpoint in the slab associated with v. For example, consider part of the base tree shown in Figure 2, where $u_1, u_2, \ldots$ are the children of u and $v_1, v_2, \ldots$ are the children of v. The interval α belongs to the sets $\Psi(u, v)$ and $\Psi(u, v_2)$, while the interval β belongs to $\Psi(v, w)$. By definition, $\Psi(u, u) = S_u^l$ and $\Psi(u, v_1), \ldots, \Psi(u, v_f)$ form a partition of $\Psi(u, v)$, i.e., $\Psi(u, v) = \bigcup_{i=1}^{f} \Psi(u, v_i)$. Further define the interval $\psi(u, v) = \max \Psi(u, v)$. Thus we have

$$\psi(u, v) = \max_{1 \le i \le f} \psi(u, v_i). \tag{2.1}$$

Next, let $p^{(0)}(v) = v$ and define $p^{(k)}(v) = p(p^{(k-1)}(v))$ for $k \ge 1$. Let

$$\Phi(v) = \bigcup_{k \ge 2} \Psi(p^{(k)}(v), v),$$

i.e., $\Phi(v)$ is the set of intervals that belong to a proper ancestor of the parent of v (i.e., they cross the right boundary of $\sigma_{p(v)}$) and that have their left endpoints inside σ_v. For example, in Figure 2, the interval α belongs to $\Phi(v_2)$, while γ belongs to $\Phi(v_1)$ and $\Phi(v)$. Note that an interval of S^l may belong to $O(\log n / \log\log n)$ of the $\Phi(v)$'s. Finally, let the interval $\phi(v) = \max \Phi(v)$. By definition, we have

$$\phi(v) = \max_{k \ge 2} \psi(p^{(k)}(v), v). \tag{2.2}$$

The secondary structure $\mathbb{L}_v$ is used to find in time $O(\log\log n)$ the maximum interval in $\Phi(v_1) \cup \cdots \cup \Phi(v_i)$, for any $1 \le i \le f$. We first show how this leads to a query procedure with an overall cost of $O(\log n)$, and then describe how to implement $\mathbb{L}_v$. For a query point q, we follow a path from the root of $\mathcal{T}$ to the leaf z such that $q \in \sigma_z$, and perform a query on the $\mathbb{L}_v$ structures of each internal node v on the path. At a node v, if $q \in \sigma_{v_i}$, i.e., v_i is the child of v on the path from the root to the leaf z, we query the $\mathbb{L}_v$ structure to find the maximum interval in $\Phi(v_1) \cup \cdots \cup \Phi(v_{i-1})$. For instance in Figure 2, when we visit u, the two intervals in $\Phi(u_1) \cup \Phi(u_2)$ are considered since $u_3 = w$ is the next node to be visited; similarly the intervals in $\Phi(v_1) \cup \Phi(v_2)$ are considered when we query the structure $\mathbb{L}_v$. When we reach the leaf z, we simply scan the $O(\log n)$ intervals whose left endpoints are stored at z and find the maximum interval. Since we spend $O(\log\log n)$ time at each of the $O(\log n / \log\log n)$ internal nodes visited, this procedure takes $O(\log n)$ time in total. That it indeed returns the maximum interval in S^l follows from the following lemma.

LEMMA 2.3. *Just before a node $v \in \mathcal{T}$ has been visited, we have found the maximum left interval among all intervals of S^l that contain the query point q except those whose left endpoints lie inside σ_v.*

Proof. The lemma is obviously true if v is the root of the base tree $\mathcal{T}$. Assume that the lemma is true for some internal node v, and let v_i be the child of v to be visited next. We will prove that the lemma is still true at v_i, i.e., that after visiting v we have found the maximal left interval among intervals that contain q and have left endpoint outside σ_{v_i}. By induction, we have already considered all intervals with left endpoints outside σ_v, so we only need to show that our query on $\mathbb{L}_v$ returns the maximum interval among those that have left endpoints inside σ_v but not σ_{v_i}. Consider any such interval s^l in S_u^l for some node u. If u is v or any of its descendants, then s^l cannot contain the query point: it is completely outside σ_{v_i} since its left endpoint is outside σ_{v_i}. So u can only be one of v's ancestors. For such an s^l to

contain the query point, it must span σ_{v_i} completely. In other words, it must have its left endpoint inside the multislab $\sigma_v[1 : i-1]$. By the definition of $\Psi(u, v_k)$ and $\Phi(v_k)$, s^l must belong to $\Phi(v_1) \cup \cdots \cup \Phi(v_{i-1})$, and the lemma follows.

We now describe the $\mathbb{L}_v$ structures that can be used to compute the maximal interval in $\Phi(v_1) \cup \cdots \cup \Phi(v_i)$ for any $1 \leq i \leq f$. Since one interval may appear in many of the $\Psi(u, v)$ sets, we cannot afford to store them or the $\Phi(v)$ sets explicitly. However, we can store $\phi(v)$ intervals, since the number of such intervals is exactly equal to the number of nodes in $\mathcal{T}$. For now let us assume that $\phi(v)$, for all $v \in \mathcal{T}$, have been computed—we will describe how to maintain them when we describe how to update S^l. Let $v_1, \ldots, v_f$ be the children of a node $v \in \mathcal{T}$. Then $\mathbb{L}_v$ is a tournament tree on $\phi(v_1), \ldots, \phi(v_f)$, that is, a binary tree with f leaves whose ith leftmost leaf stores $\phi(v_i)$. Each internal node of $\mathbb{L}_v$ stores the maximum of the intervals stored at the leaves of the subtree rooted at v. If ξ is an internal node of $\mathbb{L}_v$ with ζ, η as its two children, then the interval stored at ξ is the maximum of the two intervals stored at ζ and η. Given any $1 \leq i \leq f$, the maximum interval in $\Phi(v_1) \cup \cdots \cup \Phi(v_i)$ can be determined by selecting the maximum interval among the intervals stored at the left children of the nodes on the path from the root of $\mathbb{L}_u$ to its ith leftmost leaf. Since $\phi(v_j) = \max \Phi(v_j)$, this interval, the maximum of $\phi(v_1), \ldots, \phi(v_i)$, is the maximal interval in $\Phi(v_1) \cup \cdots \cup \Phi(v_i)$. The query is answered in $O(\log \log n)$ time, as $\mathbb{L}_v$ has height $O(\log f) = O(\log \log n)$.

LEMMA 2.4. *Using linear space $O(n)$, the set S^l of left intervals can be stored in secondary structures of $\mathcal{T}$ so that stabbing-max queries can be answered in $O(\log n)$ time.*

The dynamic structure. In order to make $\mathbb{L}_v$ dynamic, we need to maintain the $\phi(v)$ as well as $\psi(u, v)$ intervals during insertions and deletions of left intervals. Note that there are $O(n/\log \log n)$ $\psi(u, v)$ intervals since $\mathcal{T}$ has $O(n/\log n)$ nodes and each node v has $O(\log n/\log \log n)$ ancestors u. We need two additional linear-size data structures at the nodes of $\mathcal{T}$ to update $\mathbb{L}_v$'s efficiently:

(i) for an internal node u, $\mathbb{T}_u$ stores the $\psi(u, v)$ intervals for all descendants v of u; and

(ii) for an internal or a leaf node $v \in \mathcal{T}$, $\mathbb{H}_v$ stores the $\psi(u, v)$ intervals for all ancestors u of $p(v)$.

For an internal node u, $\mathbb{T}_u$ is a tree that has the same structure as the subtree $\mathcal{T}_u$ of $\mathcal{T}$ rooted at u. Thus there is a bijection between the nodes of $\mathbb{T}_u$ and $\mathcal{T}_u$, and abusing the notation slightly we will use the same letter to denote a node of $\mathcal{T}_u$ and the corresponding node of $\mathbb{T}_u$. A node $v \in \mathbb{T}_u$ stores the interval $\psi(u, v)$. In other words, the root of $\mathbb{T}_u$ stores $\psi(u, u)$, and $\psi(u, v_i)$ is stored in a child of the node storing $\psi(u, v)$ in $\mathbb{T}_u$ if and only if v_i is a child of v in $\mathcal{T}$. It follows from (2.1) that $\mathbb{T}_v$ is simply a tournament tree with fan-out f. We organize the f intervals in the children of each internal node v of $\mathbb{T}_u$ in a small binary tournament tree (effectively obtaining a binary tournament tree). We also store a small tournament tree at each leaf z of $\mathbb{T}_u$ on the intervals in $\Psi(u, z)$, the interval of S^l_u whose left endpoints lie in σ_z. We sort $\Psi(u, z)$ in increasing order of the left endpoints of its intervals. The ith leaf of the tournament tree stores the ith interval of (the sorted) $\Psi(u, z)$. This leaf tournament tree can be used to maintain the maximum interval of $\Psi(u, z)$. Note that each of the small tournament trees (in internal nodes and leaves) has size $O(\log n)$ and height $O(\log \log n)$. At each leaf of $\mathbb{T}_u$ that stores $\psi(u, z)$, we also keep a pointer to the leaf of $\mathbb{T}_{p(u)}$ that stores $\psi(p(u), z)$. These pointers will help us when we do rebalancing of the base tree in Section 3. Let $|\mathbb{T}_u|$ denote the size of $\mathbb{T}_u$. Then $\sum_{u \in \mathcal{T}} |\mathbb{T}_u| = O(n)$ because each $\psi(u, v)$ is stored only once and each interval of S^l is stored in exactly one of the leaf tournament trees.

The structure $\mathbb{H}_v$ of each node v of the base tree $\mathcal{T}$ is simply a max-heap on $\psi(u, v)$ for all (proper) ancestors u of the parent of v. It follows form (2.2) that the root of $\mathbb{H}_v$ stores $\phi(v)$. We keep pointers between the two copies of $\psi(u, v)$ in a $\mathbb{T}_u$ tree and an $\mathbb{H}_v$ heap. Again, $\sum_{v \in \mathcal{T}} |\mathbb{H}_v| = O(n)$ because each $\psi(u, v)$ is stored once.

We are now ready to describe how to update $\mathbb{L}_v, \mathbb{T}_v$, and $\mathbb{H}_v$. Let s^l be a left interval to be inserted or deleted, and suppose the left endpoint of s^l lies in the range σ_z associated with a leaf z of $\mathcal{T}$. We first identify the node u of $\mathcal{T}$ such that $s^l \in S^l_u$. Among all the $\mathbb{T}$ structures, the s^l update can only affect $\mathbb{T}_u$ since intervals in S^l_u are stored only in $\mathbb{T}_u$. Furthermore, only $O(\log n/\log \log n)$ $\psi(u, v)$ intervals in $\mathbb{T}_u$ can be affected, namely, at nodes along the path from u to the leaf z in $\mathbb{T}_u$. We update $\mathbb{T}_u$ by inserting or deleting s^l in the small tournament tree stored at the leaf z. If $\psi(u, z)$, the interval stored at the root of the tournament tree, changes, then we update $\psi(u, p(z))$ at the parent $p(z)$ of z in $\mathbb{T}_u$. This way the update may propagate up along a path of $\mathbb{T}_u$ of length $O(\log n/\log \log n)$. Since each of these updates involves updating a small tournament tree in $O(\log f) = O(\log \log n)$ time, we use a total of $O(\log n)$ time to update $\mathbb{T}_u$. Next we update each of the $O(\log n/\log \log n)$ updated $\psi(u, v)$ intervals in the relevant $\mathbb{H}_v$ structures. Since the $\psi(u, v)$ intervals in $\mathbb{T}_u$

and $\mathbb{H}_v$ are linked together and since an update of a $\mathbb{H}_v$ heap can be performed in $O(\log(\log n/\log n \log n)) = O(\log\log n)$ time, we can perform these updates in $O(\log n)$ time in total. Finally, if the maximum interval $\phi(v)$ of any $\mathbb{H}_v$ changes during the above updates, we update the value of $\phi(v)$ in the tournament tree $\mathbb{L}_{p(v)}$ at the parent of v. These are the only updates needed on the $\mathbb{L}_v$ structures. Since such an update can also be performed in $O(\log\log n)$ time, the total update time is $O(\log n)$.

LEMMA 2.5. *Using linear space $O(n)$, the set S^l of left intervals can be stored in secondary structures of $\mathcal{T}$ so that updates and stabbing-max queries can be performed in $O(\log n)$ time.*

3 General 1D Structure

In the previous section we assumed that the set of endpoints of input intervals was fixed, and thus the structure of the base tree $\mathcal{T}$ was fixed. In this section we sketch how to remove this restriction, that is, how to update $\mathcal{T}$ (insert/delete endpoints) before an insertion of an interval in the secondary structures and after a deletion of an interval from the secondary structures.

Note that deletions can be handled in a standard way using global rebuilding: When deleting an endpoint we simply mark it as deleted in the relevant leaf of $\mathcal{T}$. After $n/2$ deletions we discard the old structure, completely rebuild the base tree $\mathcal{T}$ without the deleted endpoints, and insert the intervals from the secondary structures of the old structure in the secondary structure of the new structure one by one. We can rebuild $\mathcal{T}$ in $O(n)$ time and perform the $\Theta(n)$ insertions in $O(n\log n)$ time. Thus the amortized cost of an endpoint deletion is $O(\log n)$.

In order to handle insertions we will the base tree $\mathcal{T}$ as a weight-balanced B-tree with branching factor f and leaf parameter $\log n$ [5]. The *weight* of a node v of $\mathcal{T}$, denoted as n_v, is the number of endpoints stored at the leaves of the subtree rooted at v. The weight of each leaf is between $\frac{1}{2}\log n$ and $2\log n$, and the weight of each internal node (except for the root) on level l (leaves are on level 0) is between $\frac{1}{2}f^l\log n$ and $2f^l\log n$. It is easy to see that the condition on the weights implies that the fan-out of each internal node (except for the root) is between $f/4$ and $4f$, and that the root has fan-out between 2 and $4f$ [5]. The slight variation in the number of endpoints in a leaf and the fan-out of the internal nodes of $\mathcal{T}$ does not affect any of the argument we used when discussing the secondary structures in the previous section, i.e., we can still query and update the secondary structures in $O(\log n)$ time (Lemmas 2.2 and 2.5).

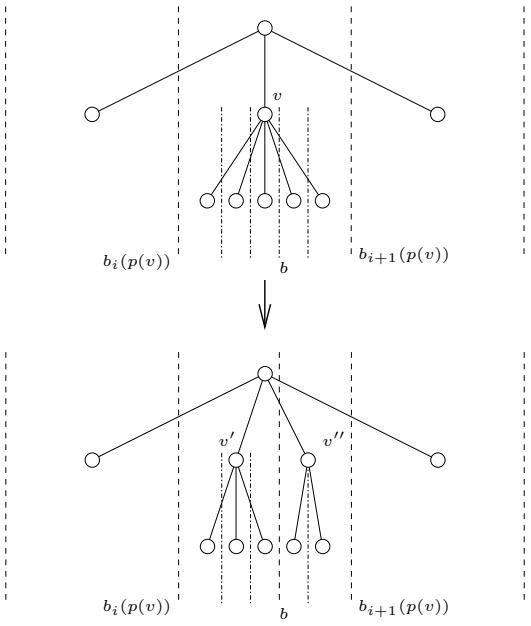

Figure 3: Node v is split into two nodes v' and v''.

After an insertion of an endpoint in a leaf z of the weight-balanced tree $\mathcal{T}$, the weight constraint of the nodes on the path from z to the root of $\mathcal{T}$ may be violated, that is, the weight of a node v on level l of the tree before the insertion may be $n_v = 2f^l\log n$. To rebalance the tree, we split v along a slab boundary b into two nodes v' and v'' of roughly equal weight $f^l\log n$ (more precisely, the weight of each of the two new nodes is between $(f^l - 2f^{l-1})\log n$ and $(f^l + 2f^{l-1})\log n$); b becomes a new slab boundary at $p(v)$ between $b_i(p(v))$ and $b_{i+1}(p(v))$ for some i. Refer to Figure 3. The split obviously not only requires updating the secondary structures at v (v' and v''), but also the nodes on the path from v to the root of $\mathcal{T}$. The secondary structures of the other nodes in the tree are unaffected by the split. Below we sketch how to perform these updates, i.e., the split of v, in $O(n_v\log\log n)$ time. Since the new nodes v' and v'' will not split until $\Theta(n_v)$ insertions have been performed below them (their weight has increased to $2f^l\log n$), the amortized cost of a split is $O(\log\log n)$. Since one insertion increases the weights of $O(\log n/\log\log n)$ nodes on a path of $\mathcal{T}$, the amortized cost of an insertion is $O(\log n)$.

The above discussion assumes that the parameter f is fixed, which in not really the case because $f = \sqrt{\log n}$. However, we can easily fix f for $\Theta(n)$ updates at a time and periodically rebuild the whole structure with an updated f (similar to the way deletions are performed using global rebuilding).

The S_v lists. We first consider how the lists storing the intervals S_v associated with each node v in $\mathcal{T}$ are affected by the split. It is easy to see that only the lists storing S_v, $S_{v'}$, $S_{v''}$ and $S_{p(v)}$ need to be updated when v splits into v' and v''. We distinguish between intervals in S_v that intersect b and that do not. Intervals of the first kind need to be moved to $S_{p(v)}$, and intervals of the second kind need to be distributed into either $S_{v'}$ or $S_{v''}$. If an interval $s \in S_v$ intersects b, we insert it at the end of the list storing $S_{p(v)}$. If s lies to the left (resp. right) of b, we insert it into the list storing $S_{v'}$ (resp. $S_{v''}$). Refer to Figure 4. Altogether we can update the lists in $O(|S_v|) = O(n_v)$ time.

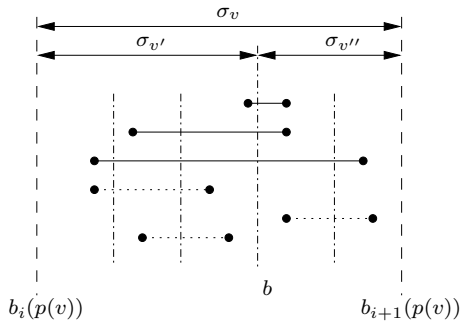

Figure 4: Update the S_v lists: Solid intervals are added to $S_{p(v)}$; dotted intervals are distributed into $S_{v'}$ and $S_{v''}$.

The $\mathbb{M}$ structures. When v splits and intervals in S_v are distributed among $S_{v'}$, $S_{v''}$, and $S_{p(v)}$, we need to update the structures $\mathbb{M}_{v'}$, $\mathbb{M}_{v''}$, and $\mathbb{M}_{p(v)}$ storing middle intervals $S^m_{v'}, S^m_{v''}$, and $S^m_{p(v)}$. Since S^m_u is unaffected for all other nodes u in $\mathcal{T}$, so is $\mathbb{M}_u$ for all other nodes u.

Since $S^m_{v'}$ and $S^m_{v''}$ can easily be computed in a simple scan of $S_{v'}$ and $S_{v''}$, the secondary structures $\mathbb{M}_{v'}$ and $\mathbb{M}_{v''}$ can be constructed in $O(|S_{v'}| \log\log n + |S_{v''}| \log\log n + \log^{3/2} n)$ time (Lemma 2.1). Since $|S_{v'}| + |S_{v''}| \leq n_v$ and $n_v = \Omega(\log^{3/2} n)$ for an internal node of v of $\mathcal{T}$, this cost is $O(n_v \log\log n)$.

Unlike $\mathbb{M}_{v'}$ and $\mathbb{M}_{v''}$, we cannot simply reconstruct the $\mathbb{M}_{p(v)}$ structure from $S_{p(v)}$ since its size can be $\Omega(n_v)$. Therefore we only make the necessary changes to $\mathbb{M}_{p(v)}$. First note that the intervals that moves from S_v to $S_{p(v)}$ because of the split do not generate new middle intervals in $S^m_{p(v)}$ (since they do not span a slab in $P(v)$). However, the addition of the new boundary b in $p(v)$ may still lead to the addition of intervals to $S^m_{p(v)}$: The intervals in $S_{p(v)}$ with left endpoints between $b_i(p(v))$ and b and right endpoints between $b_{i+1}(p(v))$ and $b_{i+2}(p(v))$, as well as those with left endpoints between $b_{i-1}(p(v))$ and $b_i(p(v))$ and right endpoints between b and $b_{i+1}(p(v))$, which did not span a slab before the split, now span a slab and thus generate middle intervals in $S^m_{p(v)}$. The two bottom intervals in Figure 5 are examples of such intervals. Since all such intervals have one endpoint in σ_v, we can easily find them in $O(n_v)$ time by scanning the endpoints stored in the subtree of $\mathcal{T}$ rooted at v.

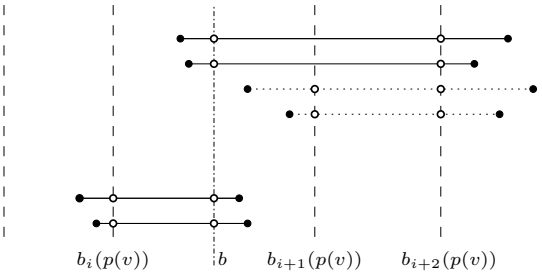

Figure 5: Update the structure $\mathbb{M}_{p(v)}$.

Recall that $\mathbb{M}_{p(v)}$ consists of $O(f^2)$ multislab max-heaps and $O(f)$ slab max-heaps. The addition of the new boundary b results in the addition of a number of new multislabs (with b as one of their boundaries) and thus in the addition of new multislab max-heaps: The new middle intervals computed above need to be inserted into two new multislab max-heaps corresponding to the two new multislabs defined by boundaries $b_i(p(v))$ and b and by boundaries b and $b_{i+1}(p(v))$. Since these max-heaps contain $O(n_v)$ intervals they can be constructed in $O(n_v)$ time. Apart from these two new multislabs, the addition of b also results in new multislabs defined by boundaries b and $b_j(p(v))$ for $j = i+2, \ldots, f$, and by boundaries $b_k(p(v))$ and b, for $k = 1, \ldots, i-1$. This may result in some of the middle intervals in $S^m_{p(v)}$ being extended to the left or right, which in turn results in the need for a split of some of the multislab max-heaps: Intervals in the max-heap of the multislab defined by boundaries $b_{i+1}(p(v))$ and $b_j(p(v))$, for $j = i+2, \ldots, f$, need to be distributed into two max-heaps corresponding to multislabs defined by boundaries b and $b_j(p(v))$ and by boundaries $b_{i+1}(p(v))$ and $b_j(p(v))$; intervals in the max-heap of the multislabs defined by boundaries $b_k(p(v))$ and $b_i(p(v))$, for $k = 1, \ldots, i-1$, need to be distributed into two max-heaps corresponding to multislab defined by boundaries $b_k(p(v))$ and $b_i(p(v))$ and by boundaries $b_k(p(v))$ and b. For example, before the split the middle parts of the top four intervals in Figure 5 all span the same multislab defined by the boundaries $b_{i+1}(p(v))$ and $b_{i+2}(p(v))$; after the addition of b the middle parts of the solid intervals are extended and span the multislab defined by the boundaries b and $b_{i+2}(p(v))$, while the dotted intervals

are unaffected. The max-heaps of all multislabs that span both or neither of $b_i(p(v))$ and $b_{i+1}(p(v))$ remain unchanged. All the distributions and constructions of multislab max-heaps can be performed in $O(n_v)$ time, since all affected intervals have at least one endpoint in σ_v, and thus the total size of all these heaps is $O(n_v)$.

Finally consider the slab max-heaps. At most $O(f)$ multislab max-heaps were modified in the above process. Each of these modifications may affect all the $O(f)$ slab max-heaps. Thus we can maintain the slab max-heaps with $O(f^2)$ updates during the split process. Since each slab max-heap is of size $O(f^2)$, we can perform these updates in $O(f^2 \log f^2) = O(\log n \log\log n) = O(n_v \log\log n)$ time. Thus altogether we have updated the middle interval structures $\mathbb{M}_{v'}$, $\mathbb{M}_{v''}$, and $\mathbb{M}_{p(v)}$ in $O(n_v \log\log n)$ time.

The $\mathbb{L}$, $\mathbb{H}$ and $\mathbb{T}$ structures. Finally, we consider the changes needed in $\mathbb{L}_u$, $\mathbb{H}_u$, and $\mathbb{T}_u$ at each node u of $\mathcal{T}$, as v splits into v' and v''. Recall that $\mathbb{L}_u$ is simply a tournament tree on the $O(f)$ $\phi(u_i)$ intervals corresponding to u's children u_i; that $\mathbb{H}_u$ is a max-heap on the intervals $\psi(p^{(k)}(u), u)$ for $k \geq 2$, with its maximum interval being $\phi(u)$; and that $\mathbb{T}_u$ is a tournament tree with the same structure as the subtree of $\mathcal{T}$ rooted at u, storing all the intervals $\psi(u, w)$'s for all the descendants w of u. As described in Section 2.3 when we discussed how to update these structures during insertions and deletions of left intervals, we can focus on the changes needed for the $\mathbb{T}$ structures: Whenever an interval in a node of a $\mathbb{T}$ structure is changed, the resulting changes can easily be made on the relevant $\mathbb{H}$ and $\mathbb{L}$ structures in $O(\log\log n)$ time. As we will see shortly, the number of intervals changed in all the $\mathbb{T}$ structures is $O(n_v)$, thereby implying that the overall time spent in updating $\mathbb{H}$ and $\mathbb{L}$ structures is $O(n_v \log\log n)$. Below we first consider how to construct the $\mathbb{T}_{v'}$ structure; the $\mathbb{T}_{v''}$ structure can be constructed similarly. Then we consider the updates to $\mathbb{T}_{p(v)}$, as some intervals move from S_v to $S_{p(v)}$ due to the split. Finally we consider the structural changes to the $\mathbb{T}_u$ structures at all nodes u of $\mathcal{T}$.

We construct $\mathbb{T}_{v'}$ in a bottom-up manner. First we construct its leaves by scanning $S_{v'}$ and assigning each interval s to the leaf z containing its left endpoint, and then constructing the small tournament tree in each leaf (recall that interval $\psi(v', z)$ of the leaf corresponding to the leaf z in $\mathcal{T}$ is simply the maximal interval in the root of this tournament tree). After this $\mathbb{T}_{v'}$ is constructed bottom-up in linear time since its basically a tournament tree. Since each leaf contains $O(\log n)$ intervals, we use $O(n_{v'} \log\log n)$ time in total to construct all the leaves. Since there are $O(n_{v'}/\log n)$ nodes in $\mathbb{T}_{v'}$, the bottom-up construction takes $O(n_{v'}/\log n)$ time.

As a result of the movement of left intervals from S_v^l to $S_{p(v)}^l$; we have already collected these intervals (the solid intervals in Figure 4) earlier when we updated the S_v lists. Each such moved interval s^l needs to be inserted in the tournament tree in the leaf of $\mathbb{T}_{p(v)}$ corresponding to the leaf z of $\mathcal{T}$ containing the left endpoint of s^l. We first locate the leaf of $\mathbb{T}_{p(v)}$ containing $\psi(p(v), z)$ by following the pointer from the leaf of $\mathbb{T}_v$ where s^l originally belong to, and then insert s^l in the tournament tree in this leaf of $\mathbb{T}_{p(v)}$. Since this tournament tree has size $O(\log n)$, the insertion takes $O(\log\log n)$ time. There are at most $O(n_v)$ such intervals that we need to move, so updating all the leaves of $\mathbb{T}_{p(v)}$ takes $O(n_v \log\log n)$ time overall. After this we can update all the affected nodes in $\mathbb{T}_{p(v)}$ bottom-up. Since $\mathbb{T}_{p(v)}$ has $O(n_v \cdot f/\log n) = O(n_v)$ nodes, the cost of this bottom-up update is also $O(n_v)$.

Finally let us consider the structural changes to be performed on the $\mathbb{T}_u$ structures for all nodes $u \in \mathcal{T}$. Recall that $\mathbb{T}_u$ has the same structure as the subtree of $\mathcal{T}$ rooted at u. Thus for each of v's ancestors u, the node in $\mathbb{T}_u$ corresponding to v, storing $\psi(u, v)$, needs to be replaced by two new nodes storing $\psi(u, v')$ and $\psi(u, v'')$. There are $O(\log n/\log\log n)$ such nodes. All other $\mathbb{T}_u$'s do not contain a node corresponding to v, hence not affected. Since all the $\psi(u, v)$ intervals stored in the affected nodes are also stored in the heap $\mathbb{H}_v$, we can locate these nodes easily. Each node of $\mathbb{T}_u$ contains a small tournament tree, so we split this tournament tree to construct the two new nodes in $\mathbb{T}_u$. Since each tournament tree has size $O(\log n)$, a split takes $O(\log\log n)$ time, for a total of $O(\log n/\log\log n) \cdot O(\log\log n) = O(\log n) = O(n_v)$ time. Note that if v is a leaf of $\mathcal{T}$, we also need to reattach the pointers between the newly created leaves of the $\mathbb{T}_u$ trees. This can be done in time linear to the number of such new leaves, which is $O(\log n/\log\log n)$.

Altogether, we have updated all the left interval structures in $O(n_v \log\log n)$ time as needed.

Putting everything together, we conclude the following.

THEOREM 3.1. *There exists a linear-size data structure for storing a set of n intervals so that stabbing-max queries can be answered in $O(\log n)$ time worst-case and updates can be performed in $O(\log n)$ time amortized.*

4 Extensions

In this section we discuss some extensions of our one-dimensional interval max-stabbing structure.

External memory structures. Our internal memory interval max-stabbing structure can easily be adapted to external memory by choosing the right base tree parameters: We let each leaf of the base tree contain $B \log_B n$ endpoints, and let the fan-out f be $\max\{\sqrt{\log_B n}, \sqrt{B}\}$. We also change the fanout of all the tournament trees to B such that a tournament tree on K elements can be updated and queried in $O(\log_B K)$ I/Os. We omit the details of the construction since they are very similar to the internal memory structure.

THEOREM 4.1. *There exists an external memory data structure for storing a set of n intervals of size $O(n/B)$ so that stabbing-max queries can be answered in $O(\log_B n)$ I/Os worst-case and updates can be performed in $O(\log_B n)$ I/Os amortized.*

Higher dimensions. Our one-dimensional structures can be extended to higher dimensions in a standard way using segment trees [9]. This adds an extra $O(\log n)$ factor to space, update time, and query time for each dimension.

THEOREM 4.2. *There exists an $O(n \log^{d-1} n)$ size data structure for storing a set of n axis-parallel hyper-rectangles in $\mathbb{R}^d$ so that stabbing-max queries can be answered in $O(\log^{d-1} n)$ time worst-case and updates can be performed in $O(\log^{d-1} n)$ time amortized.*

Similarly, the external structure can also be extended to higher dimensions using standard techniques [1]. Details will appear in the full paper.

THEOREM 4.3. *There exists an $O(\frac{n}{B} \log_B^{d-1} n)$ space external data structure for storing a set of n axis-parallel hyper-rectangles in $\mathbb{R}^d$ so that stabbing-max queries can be answered in $O(\log_B^{d-1} n)$ I/Os worst-case and updates can be performed in $O(\log_B^{d-1} n)$ I/Os amortized.*

References

[1] P. K. Agarwal, L. Arge, J. Yang, and K. Yi, I/O-efficient structures for orthogonal range max and stabbing max queries, *Proc. European Symposium on Algorithms, Lecture Notes in Computer Science*, 2832, 2003, Springer Verlag, pp. 7–18.

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

[3] L. Arge, V. Samoladas, and J. S. Vitter, On two-dimensional indexability and optimal range search indexing, *Proc. 18th ACM Symposium on Principles of Database Systems*, 1999, pp. 346–357.

[4] L. Arge and J. Vahrenhold, I/O-efficient dynamic planar point location, *Computational Geometry: Theory and Applications*, 29 (2004), 147–162.

[5] L. Arge and J. S. Vitter, Optimal external memory interval management, *SIAM Journal on Computing*, 32 (2003), 1488–1508.

[6] G. S. Brodal, S. Chaudhuri, and J. Radhakrishnan, The randomized complexity of maintaining the minimum, *Nordic Journal of Computing*, 3 (1996), 337–351.

[7] B. Chazelle, Filtering search: a new approach to query-answering, *SIAM J. Comput.*, 15 (1986), 703–724.

[8] H. Edelsbrunner, A new approach to rectangle intersections, part I, *Int. J. Computer Mathematics*, 13 (1983), 209–219.

[9] H. Kaplan, E. Molad, and R. E. Tarjan, Dynamic rectangular intersection with priorities, *Proc. 35th ACM Symposium on Theory of Computation*, 2003, pp. 639–648.

[10] K. Mehlhorn and S. Näher, Dynamic fractional cascading, *Algorithmica*, 5 (1990), 215–241.

[11] C. W. Mortensen, Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time, *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms*, 2003, pp. 618–627.

[12] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most specific rule matching, *Proc. 6th International Symposium on Parallel Architectures, Algorithms, and Networks*, 2002, pp. 3–14.

[13] R. E. Tarjan, A class of algorithms that require nonlinear time to maintain disjoint sets, *Journal of Computer and System Sciences*, 18 (1979), 110–127.

[14] J. Yang and J. Widom, Incremental computation and maintenance of temporal aggregates, *Proc. IEEE International Conference on Data Engineering*, 2001, pp. 51–60.