AN OPTIMAL DYNAMIC DATA STRUCTURE FOR STABBING-SEMIGROUP QUERIES[*]

PANKAJ K. AGARWAL[†], LARS ARGE[‡], HAIM KAPLAN[§], EYAL MOLAD[¶], ROBERT E. TARJAN[‖], AND KE YI[**]

Abstract. Let S be a set of n intervals in $\mathbb{R}$, and let $(\mathbf{S}, +)$ be any commutative semigroup. We assign a weight $\omega(s) \in \mathbf{S}$ to each interval in S. For a point $x \in \mathbb{R}$, let $S(x) \subseteq S$ be the set of intervals that contain x. Given a point $q \in \mathbb{R}$, the *stabbing-semigroup* query asks for computing $\sum_{s \in S(q)} \omega(s)$. We propose a linear-size dynamic data structure, under the pointer-machine model, that answers queries in worst-case $O(\log n)$ time, and supports both insertions and deletions of intervals in amortized $O(\log n)$ time. It is the first data structure that attains the optimal $O(\log n)$ bound for all three operations. 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.

For the restricted case of nested family of intervals (every pair of intervals are either disjoint or one contains the other), we present a simpler solution based on dynamic trees.

1. Introduction.

Let S be a set of n intervals in $\mathbb{R}$, and let $(\mathbf{S}, +)$ be any commutative semigroup. We assign a weight $\omega(s) \in \mathbf{S}$ to each interval in S. For a point $x \in \mathbb{R}$ and a set R of intervals, let $R(x) \subseteq R$ be the set of intervals that contain x. Given a point $q \in \mathbb{R}$, a *stabbing-semigroup* query asks for computing $\sum_{s \in S(q)} \omega(s)$. We are interested in developing a dynamic data structure to maintain S dynamically, so that we can answer stabbing-semigroup queries and insert and delete intervals to/from S efficiently. By taking different semigroups, for instance $(\mathbb{Z}, +)$, $(\mathbb{R}, \max)$, $(\mathbb{N}, \gcd)$, $(\{0, 1\}, \vee)$, etc., we obtain different applications of our data structure. If every pair of intervals in S is either disjoint or nested, we call the problem a *nested instance* of the stabbing-semigroup problem.

The so-called *stabbing-max (resp. stabbing-min)* problem is the special case of the problem with the semigroup $(\mathbb{R}, \max)$ (resp. $(\mathbb{R}, \min)$). This problem has applications in object oriented programming [11, 12] and *IP routing* [13, 10, 17]. In IP routing, a router maintains a dynamic table of prefixes of IP addresses which is used to pick the outgoing line for each incoming packet. The decision is done by identifying the longest prefix of the destination address of the packet stored in its table. We can model this problem as a stabbing-min problem where each prefix corresponds to an interval whose weight equals to its length. The destination address of a packet is a

[*]Part of work was done while Arge and Yi were at Duke University. Agarwal was supported by NSF under grants CNS-05-40347, IIS-07-13498, CCF-09-40671, and CCF-1012254, by ARO grants W911NF-07-1-0376 and W911NF-08-1-0452, by an NIH grant 1P50-GM-08183-01, and by a grant from the U.S.–Israel Binational Science Foundation. Arge was supported by MADALGO, a center of the Danish National Research Foundation. Kaplan and Tarjan were supported by Grant no. 2006204 from the U.S.–Israel Binational Science Foundation. Tarjan's work at Princeton was partially supported by NSF grants CCF-0830676 and CCF-0832797. Yi was supported by a DAG and an RPC grant from HKUST, and a Google Faculty Researach Award.

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

[‡]Center for Massive Data Algorithmics (MADALGO), Department of Computer Science, Aarhus University, Aarhus, Denmark. Email: large@madalgo.au.dk

[§]Depatment of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. Email: haimk@tau.ac.il

[¶]Depatment of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel.

[‖]Department of Computer Science, Princeton University, Princeton, NJ and Hewlett-Packard, Palo Alto, CA. Email: ret@cs.princeton.edu

[**]Department of Computer Science and Engieerning, HKUST. Email: yike@cse.ust.hk

point and the shortest interval containing this point corresponds to the longest prefix of the destination address. Note that the family of intervals in this application is nested.

A more general problem arising in routers is *IP packet classification.* A router often classifies each incoming packet into a *flow* according to some fields in the packet header. The router then processes in the same way all packets that are in the same flow. To do the classification, the router maintains a set of rules, each with a priority assigned to it. The highest-priority rule that a packet obeys determines the flow of the packet. The rules may stipulate range constraints on one or more fields in the packets (e.g., source/destination IP addresses, source/destination ports), which corresponds to one or multi-dimensional versions of the stabbing-max problem. In many networking contexts, such as multicast routing protocols and QoS protocols, the set of rules changes over time, in which case we need the dynamic version of the stabbing-max problem.

Previous work. A linear-size static data structure for the stabbing-semigroup problem that supports queries in $O(\log n)$ time can be developed using the segment tree [8] — each node stores the semigroup sum of the intervals associated with it. This structure can be extended to support insertions of intervals in $O(\log n)$ time, without affecting the asymptotic query time, by using a dynamic segment tree [18]. However, the problem becomes considerably harder when deletions are allowed. If the weights are drawn from a group, namely in the *stabbing-group* problem, deleting an interval s with weight $\omega(s)$ can be implemented by inserting s with weight $-\omega(s)$, with periodic re-building to avoid a space blowup. However, this solution does not apply to the semigroup case because there are no inverses. By modifying the segment tree so that each node stores the set of intervals associated with it, a query can be answered in $O(\log n)$ time, but an update takes $O(\log^2 n)$ time and the size of the data structure becomes $O(n \log n)$. Alternatively, by using an interval tree [9] one can obtain a linear-size data structure that supports both insertions and deletions in $O(\log n)$ time but requires $O(\log^2 n)$ time to answer a query. We discuss these structures in more detail in Section 2.

Faster data structures have been developed for the stabbing-max problem in the context of the IP routing problem by exploiting the fact that endpoints of intervals are integers, and using the RAM model. For example, Feldmann and Muthukrishnan [10] proposed the fat inverted segment tree (FIS) data structure. The dynamic version of FIS supports queries in $O(\log\log n + \ell)$ time, where ℓ is the number of levels in the segment tree. The space requirement is $O(n^{1+1/\ell})$, and an insertion or deletion takes $O(n^{1/\ell} \log n)$ time, but there is an upper bound on the total number of insertions and deletions allowed. Thorup [20], improving the result of Feldmann and Muthukrishnan, presents a linear-size data structure with $O(\ell)$ query time and $O(n^{1/\ell})$ update time for $\ell = o(\log n/\log\log n)$, $\ell \geq \min\{\sqrt{\log n/\log\log n}, \log\log N\}$, when the endpoints are integers not exceeding N. See [13] for a survey of such results.

If the input is too large to fit in the main memory, one is interested in an external memory data structure. In the standard two-level I/O model of computation [2], the machine consists of a finite main memory and an infinite-size disk. In this model, a block of B consecutive elements can be transferred between main memory and disk, and this is referred to as one *I/O operation.* The data structure is stored in a number of disk blocks, each of size B, and the cost of an operation is measured by the number of I/O operations. See [4, 21] for surveys on external memory data structures. For the stabbing-semigroup problem, the I/O-efficient interval tree developed by Arge and Vitter [5] can be used to construct a linear-size data structure for answering a stabbing-semigroup query in $O(\log_B n)$ I/Os. An interval can be inserted into S using $O(\log_B n)$ I/Os. Their structure can be modified to handle deletions so that each update takes $O(\log_B n)$ I/Os but then a query requires $O(\log_B^2 n)$ I/Os. An I/O-efficient structure for the stabbing-group problem is presented in [22] that uses linear space, answers a query and performs an update in $O(\log_B n)$ I/Os, but it does not work for the semigroup problem.

Our results. The results in this paper combine and extend results from two conference abstracts [14, 1]. Our main result is a linear-size data structure for the stabbing-semigroup problem, in the pointer-machine model [19] of computation. Our structure 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. Our solution starts from the straightforward solutions based on interval and segment trees mentioned above. We then combine features of these two data structures so that query time is $O(\log n)$, update time is $O(\log n \log\log n)$, and the size of the data structure is $O(n \log\log n)$. Next, we reduce the size and update time by a factor of $\log\log n$ by using a base tree that is a weight-balanced tree with a large fan-out, in which each fat leaf stores the endpoints of many intervals. Our approach also leads to a data structure with a similar performance in the I/O model. More precisely, we obtain a linear-size data structure such that a query can be answered using $O(\log_B n)$ I/Os (worst-case) and each update takes $O(\log_B n)$ I/Os (amortized). We also propose a simpler data structure that uses dynamic trees to solve nested instances of the problem. Finally, we prove that our structure is optimal, in the sense that for certain semigroups none of the query, insertion, or deletion bounds can be improved without sacrificing the others. The lower bound is established in the cell-probe model, and in fact holds for the (easier) stabbing-group problem. Previously an $\Omega(\log n/\log\log n)$ lower bound was known [3] for the problem. Our structure can be extended to higher dimensions using segment trees in a standard way [14], by paying a penalty of an $O(\log n)$ factor in both time and space for each additional dimension, but the results may not be optimal in two or higher dimensions.

The rest of this paper is organized as follows. We begin in Section 2 by describing simple data structures for the stabbing-semigroup problem that use interval and segment trees. In Section 3, we describe our structure under the assumption that the endpoints of all intervals belong to a fixed set P of $O(n)$ points. This allows us to disregard the rebalancing issue of the base tree in our multi-level structure. We remove this assumption in Section 4, where we describe how to rebalance the base tree. In Section 5, we describe how our structure can be adapted to external memory. Section 6 presents our data structure for nested instances. We prove the lower bounds in Section 7 and conclude with some open problems in Section 8.

2. Preliminaries and Basic Data Structures. We denote by S the set of (closed) intervals stored in the structure. We use n to denote the cardinality of S. Note that n changes as S is modified via insertions and deletions. For an interval $x \in S$, we denote by $\omega(x)$ the weight of x. Each weight belongs to a semigroup $\mathbf{S}$. To simplify the presentation, we assume that all the endpoints of the intervals in S, as well as the queries, are distinct. This assumption can easily be removed by fixing an arbitrary order among identical endpoints.

For any $Y \subset S$, let $\omega(Y) = \sum_{s \in Y} \omega(s)$. For a subset $Y \subset S$ and a point q, we denote by $Y(q)$ the subset of Y consisting of all intervals containing q. We assume that the semigroup is a *monoid*, i.e., it has an identity element, which we denote by

0, and define $\omega(\emptyset) = 0$.

Next, we describe the basic building blocks of our main data structure. The basic ingredient we use is a structure for storing a totally ordered set $\mathbb{X}$ such that each $x \in \mathbb{X}$ has a *weight* $\omega(x) \in \mathbf{S}$, subject to the following operations.

(i) INSERT(x): Insert x into $\mathbb{X}$.

(ii) DELETE(x): Delete x from $\mathbb{X}$.

(iii) UPDATEWT(x, w): Given $x \in \mathbb{X}$ and $w \in \mathbf{S}$, Update $\omega(x)$ to be w. We can implement this by deleting x and reinserting x with its new weight.

(iv) WT$(\mathbb{X})$: Return $\omega(\mathbb{X})$.

(v) PREFIXSUM(b): Given $b \in \mathbb{X}$, return $\sum_{x \in \mathbb{X},\, x \le b} \omega(x)$.

We implement this data type by a dynamic balanced binary tree [7], in which we maintain the sum of the weights of the elements in each subtree. Then all operations take time logarithmic in the size of $\mathbb{X}$, except WT$(\mathbb{X})$, which takes $O(1)$ time. The size of the data structure is linear in $|\mathbb{X}|$. We can also support a FIND(x) operation that locates x in the search tree in logarithmic time. If $\mathbb{X}$ is unordered, we can still use this data structure by imposing an arbitrary total order on $\mathbb{X}$. We call such a basic structure a BST (for balanced search tree). Throughout this paper we shall often use the same name for the set and the BST representing it.

Our new data structure can be viewed as a mixture of an interval tree and a segment tree [9], so we start by reviewing these classical structures. We describe the static versions of each of these structures, but they can be made dynamic using the standard techniques [6].

Interval tree. In this section and Section 3, we assume that the endpoints of all intervals in S that are ever in the structure belong to a fixed set P of $m = O(n)$ points. We divide $\mathbb{R}$ into m *atomic* intervals by picking an arbitrary separating point between every two consecutive points in P. We consider these atomic intervals closed (except the leftmost and the rightmost one). Let $\mathcal{T}$ be a balanced full binary tree with m leaves. Each node $v \in \mathcal{T}$ is associated with an interval σ_v. If v is the i-th leftmost leaf of $\mathcal{T}$, then σ_v is the i-th leftmost atomic interval. If v is an interior node with v_1 and v_2 as its children, then the common endpoint x_v of σ_{v_1} and σ_{v_2} is stored at v, and we set $\sigma_v = \sigma_{v_1} \cup \sigma_{v_2}$. For a point $x \in \mathbb{R}$, let Π_x denote the path in $\mathcal{T}$ from the root to the deepest node z such that σ_z contains x. Note that for every point $x \notin P$, Π_x is a path from the root to the leaf z whose atomic interval contains z.

In an interval tree, an interval $s \in S$ is stored at the highest node v such that $x_v \in s$. Note that S_v is empty if v is a leaf. Let $S_v \subseteq S$ be the set of intervals stored at v. Let $s = [a, b]$ be an interval in S_v. We split s into two subintervals $s^\ell = [a, x_v]$ and $s^r = [x_v, b]$. We define $L_v = \{s^\ell \mid s \in S_v\}$, $R_v = \{s^r \mid s \in S_v\}$, $L = \{s^\ell \mid s \in S\}$, and $R = \{s^r \mid s \in S\}$. For a query point $q \in \mathbb{R}$, $\omega(S(q)) = \omega(L(q)) + \omega(R(q))$.[1] We compute $\omega(L(q))$ and $\omega(R(q))$ separately and return their sum.

We add the following secondary structures to the interval tree to compute $\omega(L(q))$ efficiently; the construction is symmetric for computing $\omega(R(q))$. For a node v, let E_v be the set of the left endpoints of the intervals in L_v. We assign to each point of E_v the weight of the corresponding interval, and store E_v in a BST. Clearly the total size of the data structure, including all secondary structures, is $O(n)$.

Let q be a query point, Let $\Pi_q^\ell \subseteq \Pi_q$ be the set of nodes $v \in \Pi_q$ such that v's left child is also in Π_q. Note that $L(q) \subseteq \bigcup_{v \in \Pi_q^\ell} L_v$ and that $\omega(L(q)) = \sum_{v \in \Pi_q^\ell} \omega(L_v(q))$. Moreover, an interval $[a, x_v] \in L_v$ contains q if and only if $a \le q$. To compute $\omega(L(q))$

[1] If $q = x_v$ for some separating point x_v, we query with a q^+ that we consider to be symbolically larger than q.

we traverse the path Π_q. At each node $v \in \Pi_q^\ell$, we perform a PREFIXSUM(q) query on E_v to obtain the weight $\sum_{a \in E_v, a \le q} \omega(a) = \omega(L_v(q))$ in $O(\log n)$ time. Finally, we sum these weights and return the overall weight. Since we spend $O(\log n)$ time at each node v, the overall query time is $O(\log^2 n)$. We can insert or delete an interval s in an interval tree in $O(\log n)$ time by finding the node v such that $s \in S_v$ and updating the BST representing E_v.

Note that it might be tempting to use dynamic fractional cascading [15] to speed up the query procedure, but this does not work because the PREFIXSUM(q) operation on a BST actually relies on retrieving $O(\log n)$ weights in the BST, not just one search location. If one stores the prefix sum at the search location, the update cost of the BST will be high.

Segment tree. A segment tree allows us to compute $\omega(S(q))$ in $O(\log n)$ time, although the update time is $O(\log^2 n)$ and the size is $O(n \log n)$. The base tree $\mathcal{T}$ of the segment tree is the same as that of the interval tree. However, we now store an interval $s = [a, b]$ at a node v if $\sigma_v \subseteq s$ and $\sigma_{p(v)} \nsubseteq s$, where $p(v)$ denotes the parent of v. Note that the parents of the nodes storing s lie on $\Pi_a \cup \Pi_b$, and that we can find these nodes in $O(\log n)$ time. Let $\bar{S}_v \subseteq S$ be the set of intervals stored at v in the segment tree. We maintain $\bar{S}_v$ in a BST (imposing an arbitrary order on these intervals). For a leaf z, let $\mathcal{L}_z$ be the set of intervals with an endpoint inside σ_z. (For now, $\mathcal{L}_z$ contains at most one interval, but denoting it as a set will be convenient later on.) We store $\mathcal{L}_z$ at z.

Since an interval s is stored at $O(\log n)$ nodes, the total size is $O(n \log n)$. An interval can be inserted or deleted in $O(\log^2 n)$ time by first finding in $O(\log n)$ time the nodes at which s is stored and then updating the BST at each such node. For a query point $q \in \mathbb{R}$, let z be the leaf on the path Π_q, then we have

$$S(q) = \bigcup_{v \in \Pi_q} \bar{S}_v \cup \mathcal{L}_z(q). \tag{2.1}$$

Since the sets $\bar{S}_v$ for $v \in \Pi_q$ and $\mathcal{L}_z$ are pairwise disjoint, $\omega(S(q)) = \sum_{v \in \Pi_q} \omega(\bar{S}_v) + \omega(\mathcal{L}_z(q))$. Therefore, $\omega(S(q))$ can be computed in $O(\log n)$ time by traversing the path Π_q, retrieving the value $\omega(\bar{S}_v)$ in $O(1)$ time from the BST representing $\bar{S}_v$ at each node $v \in \Pi_q$, and finally checking if the interval in $\mathcal{L}_z$ contains q.

3. An Optimal Data Structure for Fixed Endpoints. In this section, we continue to assume that although the set S of intervals is dynamic, the endpoints of these intervals belong to a fixed set P of $O(n)$ points. Recall that our assumptions in Section 2 also imply that each point of P is an endpoint of at most one interval. The main result is a linear-size data structure that answers a stabbing-semigroup query in $O(\log n)$ time and performs an update in $O(\log n)$ time. Recall that the segment tree attains an optimal query time whereas the interval tree attains an optimal update time. We first describe how to combine the features of interval and segment trees to construct a data structure of size $O(n \log n)$ on the same base tree that supports queries in $O(\log n)$ time and updates in $O(\log n \log \log n)$ time. We then reduce the size to linear and the update time to $O(\log n)$ without increasing the asymptotic query time, by increasing the fan-out of the base tree and making the leaves fat, as we will explain.

3.1. Binary base tree. Intuitively, we store the intervals in secondary structures as in the interval tree so that each interval is stored at one node of $\mathcal{T}$. However, we maintain the weights as in a segment tree to expedite the query procedure. We now describe the data structure in detail.

As in the interval tree described in Section 2, we split each interval into left and right intervals and process the sets L of left intervals and R of right intervals separately. We describe the secondary structure for L, which allows us to compute $\omega(L(q))$ for a query point q efficiently. The construction for R is symmetric. We assume that each point of P that is an endpoint of an interval s stores a bi-directional pointer to s, and to the node u such that $s \in S_u$.

Decomposition of intervals. In what follows, we first decompose the set of left intervals L in multiple ways that will facilitate the query and update procedures. First, recall that L_u is the set of intervals in L that have x_u as their right endpoints. For a descendant v of a node u, we define $L_{vu} \subseteq L_u$ to be

$$L_{vu} = \{[a, x_u] \in L_u \mid a \in \sigma_v\}. \tag{3.1}$$

This is the set of all intervals in L whose right endpoints are x_u and whose left endpoints lie inside σ_v. Note that a particular interval $[a, x_u] \in L_u$ is included in L_{vu} for every node v on the path from u's left child down to the leaf z where $a \in \sigma_z$. For a node v, let $A_v = \{u \mid u \text{ is a proper ancestor of } p(v)\}$ and

$$\Phi_v = \bigcup_{u \in A_v} L_{vu}. \tag{3.2}$$

See Figure 3.1 for an illustration of these sets.

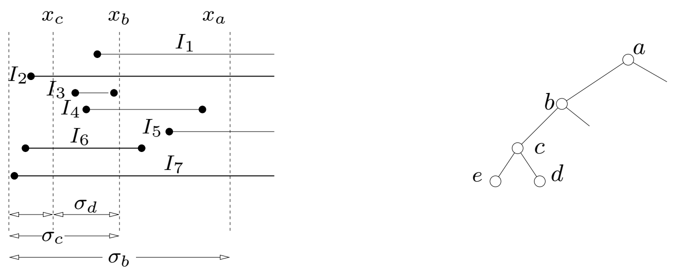

Fig. 3.1. Illustration of the definition of the sets Φ_v, L_{vu}'s: $L_{ba} = \{I_1, I_2, I_5, I_7\}$, $L_{ca} = \{I_1, I_2, I_7\}$, $L_{da} = \{I_1\}$, $L_{ea} = \{I_2, I_7\}$, $L_{db} = \{I_4\}$, $L_{eb} = \{I_6\}$, and $\Phi_e = \bar{L}_d = \{I_2, I_6, I_7\}$. The picture shows the entire interval and not only its part which is in L.

Since for a fixed v, the sets L_{vu} are pairwise disjoint, we have

$$\omega(\Phi_v) = \sum_{u \in A_v} \omega(L_{vu}). \tag{3.3}$$

Let $\bar{L}_w = \{s \in L \mid \sigma_w \subseteq s, \sigma_{p(w)} \not\subseteq s\}$, which is the subset of L stored at w if we place the intervals according to the rules of a segment tree. The following lemma shows how a query should use the Φ_v sets.

Lemma 3.1. *If w is a right child of its parent and v is the left sibling of w, then $\Phi_v = \bar{L}_w$.*

Proof. Let $s = [a, b]$ be an interval in Φ_v, and let $u = p(v) = p(w)$. Then a must be in σ_v, and b must be to the right of σ_u. So clearly $s \in \bar{L}_w$. To prove the converse, assume that $s = [a, b] \in \bar{L}_w$. Then by the definition of a segment tree, $\sigma_w \subseteq s$ but

$\sigma_{p(w)} \not\subseteq s$. It follows that $a \in \sigma_v$. Therefore, there is a proper ancestor u' of u such that $s \in L_{vu'}$. This implies that $s \in \Phi_v$. $\square$

Let w be a node that is the left child of its parent. An interval $s = [a, b] \in L$ that contains σ_w must contain $\sigma_{p(w)}$. Indeed, if this is not case, then $b = x_{p(w)}$, but then a must be in σ_w (otherwise s is not a left interval), which contradicts the assumption that $\sigma_w \subseteq s$. It follows from this observation that if w is a left child, then $\bar{L}_w = \emptyset$. Combining this with Lemma 3.1 and Property (2.1) of a segment tree, we obtain the following. For a query point q, let z be the leaf on the path Π_q and $\mathcal{L}_z$ the set of intervals in L with an endpoint in σ_z (there is at most one), we have

$$L(q) = \left(\bigcup_{w \in \Pi_q} \bar{L}_w \right) \cup \mathcal{L}_z(q) = \left(\bigcup_{\substack{w \in \Pi_q \\ w:\ \text{right child}}} \bar{L}_w \right) \cup \mathcal{L}_z(q) = \left(\bigcup_{\substack{w \in \Pi_q \\ w:\ \text{right child} \\ v:\ \text{sibling of } w}} \Phi_v \right) \cup \mathcal{L}_z(q).$$

Since the Φ_v sets are pairwise disjoint for nodes whose right siblings belong to the path Π_q, we have

$$\omega(L(q)) = \left(\sum_{\substack{w \in \Pi_q \\ w:\ \text{right child} \\ v:\ \text{sibling of } w}} \omega(\Phi_v) \right) + \omega(\mathcal{L}_z(q)). \tag{3.4}$$

The secondary structures. We will use (3.4) to answer a query by adding up all the necessary $\omega(\Phi_v)$'s, while building secondary structures to maintain the $\omega(\Phi_v)$'s according to (3.3) under updates. More precisely, for each node v, we build a BST on A_v using $\omega(L_{vu})$ as the weight of $u \in A_v$. Clearly, this BST has size $O(\log n)$, and $\omega(A_v) = \omega(\Phi_v)$ can be retrieved in $O(1)$ time. When any $\omega(L_{vu})$ changes, the BST can be updated in $O(\log \log n)$ time. The total size of all the secondary data structures is $O(n + \sum_{v \in \mathcal{T}} |A_v|) = O(n \log n)$.

The query procedure is easy. Let q be a query point. We traverse the path Π_q and compute $\omega(L(q))$ in $O(\log n)$ time using (3.4), as follows: If a node $w \in \Pi_q$ is a right child and v is its left sibling, then we retrieve $\omega(\Phi_v)$ in $O(1)$ time, and we add up these weights. Finally, we check if the interval $s \in \mathcal{L}_z$ (if there is one) contains q, and if so, add its weight as well.

To insert or delete an interval $s = [a, b]$, we first find the node u such that $s \in S_u$. Let $s^\ell = [a, x_u]$ and $s^r = [x_u, b]$. We only describe how to handle s^ℓ. We traverse the path Π_a from the leaf z such that $a \in \sigma_z$, to the root of $\mathcal{T}$, updating the secondary structures bottom-up, as follows. At z, we update the weight of u in A_z, $\omega(L_{zu})$, in the BST at z to $\omega(s)$ (for an insertion) or 0 (for a deletion). Next, suppose we are at an internal node $v \in \Pi_a$, a descendant of u, with children v_1 and v_2, where v_1 is the child of v preceding it on Π_a that we have just processed. At this point we already have an updated value of $\omega(L_{v_1 u})$. Since $\sigma_v = \sigma_{v_1} \cup \sigma_{v_2}$, we have

$$\omega(L_{vu}) = \omega(L_{v_1 u}) + \omega(L_{v_2 u}).$$

We retrieve $\omega(L_{v_2 u})$ from A_{v_2}, which is not affected by the insertion/deletion of s, and compute the new $\omega(L_{vu})$. Then we update, in $O(\log \log n)$ time, the weight of u, $\omega(L_{vu})$, in the BST on A_v. We stop this bottom-up traversal of Π_a at the grandchild of u on Π_a, and the total time spent is $O(\log n \log \log n)$.

Remark. Kaplan, Molad, and Tarjan [14] showed that if we redefine A_v to be the set of those ancestors of v for which $L_{vu} \neq \emptyset$, then the size of the structure reduces to $O(n \log \log n)$ without affecting the query or update time. Furthermore by maintaining only the top part of $\mathcal{T}$ explicitly and storing the intervals of nodes of depth larger than $n/\log n$ separately, the space can be made linear and the insertion time $O(\log n)$, but the deletion time remains $O(\log n \log \log n)$. We omit these details as we will show a different approach that achieves optimality on both insertions and deletions.

3.2. Non-binary base tree. We now improve the update time to $O(\log n)$ and the space bound to $O(n)$, without increasing the asymptotic query time. We do this by increasing the fan-out of each node in the base tree $\mathcal{T}$ and by making each leaf of $\mathcal{T}$ fat. As a result of the large fan-out, we need more complicated secondary structures, which will be the focus of this section.

The base tree. As above, let P be the fixed set of $m = \Theta(n)$ points that contains all the endpoints of the intervals in S. We continue to assume that at any time each point in P is an endpoint of at most one interval in S. We divide $\mathbb{R}$ into $\lceil m/\log m \rceil = O(n/\log n)$ atomic intervals, by adding a break point every $\lceil \log m \rceil$ consecutive points of P. Let $f = \lceil \sqrt{\log n} \rceil$. We build an f-ary tree on top of these atomic intervals where each leaf corresponds to one. For ease of exposition, we assume that the number of leaves is a power of f; otherwise, the fan-out of each internal node is $\Theta(f)$, but that does not affect our asymptotic results.

We chose the size of a leaf and the fanout so that $\mathcal{T}$ has $O(n/\log^{3/2} n)$ internal nodes. This choice allows to keep secondary data structures of size $O(\log^{3/2} n) = O(f^3)$ in each internal node while keeping the overall space linear, as we will do.

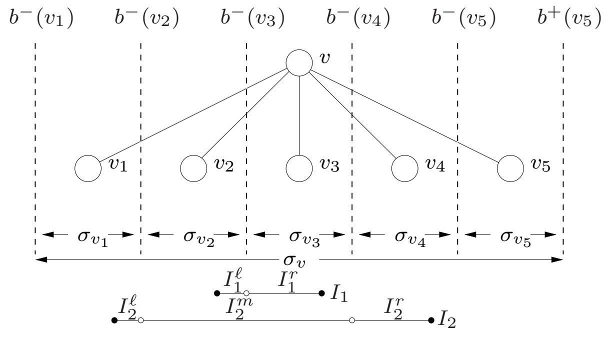

Fig. 3.2. An internal node v in the base tree $\mathcal{T}$. I_1^m is undefined.

As in Section 2, we associate an interval σ_v with every node v of $\mathcal{T}$. If v is a leaf, then σ_v is the corresponding atomic interval. If v is an internal node with children $v_1, \ldots, v_f$, from left to right, then $\sigma_v = \sigma_{v_1} \cup \cdots \cup \sigma_{v_f}$. We refer to each σ_{v_i} as a *slab* of v associated with v_i. For a child v_i of v, let $b^-(v_i)$ be the left endpoint of σ_{v_i} and let $b^+(v_i)$ be the right endpoint of σ_{v_i}. Note that $b^+(v_i) = b^-(v_{i+1})$ for $1 \le i < f$, and $b^-(v_1)$ and $b^+(v_f)$ are the boundaries of the slab of $p(v)$ associated with v. For two children v' and v'' of v, we write $v' < v''$ if v'' is to the right of v', i.e., $b^+(v') \le b^-(v'')$. We write $v' \le v''$ if either $v' = v''$ or $v' < v''$. We store the slab boundaries of each internal node in a balanced binary search tree so that we can determine in $O(\log \log n)$ time the slab of v that contains a point $x \in \sigma_v$. This allows us to traverse the search path Π_x from the root to the leaf v containing a point x in $O(\log n)$ time.

An interval s is associated with v if s is contained in σ_v but not a slab associated with any of v's children. In particular, if an interval s has both endpoints stored in the same leaf z, then s is associated with z. As earlier, let $S_v \subseteq S$ be the subset of intervals associated with v. Clearly each interval is associated with exactly one node v.

As when the base tree was binary, we store with each point in P that is an endpoint of an interval s a bi-directional pointer to s and to the node u such that $s \in S_u$. Given a new interval $s = [a, b]$, we can easily find the node u such that $s \in S_u$ in $O(\log n)$ time. We traverse the path Π_a top-down. For each internal node v on this path, we determine in $O(\log \log n)$ time the slab $\sigma_{v'}$ that contains a. If $b \in \sigma_{v'}$, we recursively visit the child v'. If $b \notin \sigma_{v'}$, then $s \in S_v$. If we reached a leaf z, then $s \in S_z$.

Let $v \in \mathcal{T}$ be an internal node, and let $s = [a, b] \in S_v$. Assume first that v' and v'' are children of v such that $a \in \sigma_{v'}$ and $b \in \sigma_{v''}$. If $b^+(v') = b^-(v'')$, then we split s into two intervals: $s^\ell = [a, b^+(v')]$ and $s^r = [b^+(v'), b]$ (e.g. interval I_1 in Figure 3.2). Otherwise, we split s into three intervals: $s^\ell = [a, b^+(v')]$, $s^m = [b^+(v'), b^-(v'')]$, and $s^r = [b^-(v''), b]$ (e.g. interval I_2 in Figure 3.2). We refer to s^ℓ, s^m, and s^r as the left, middle, and right intervals of s, and their weights are the same as the weight of s. For an internal node v, let

$$L_v = \{s^\ell \mid s \in S_v\}, \quad R_v = \{s^r \mid s \in S_v\}, \quad M_v = \{s^m \mid s \in S_v\},$$

and for a leaf v let $L_v = R_v = M_v = \emptyset$. Let $L = \bigcup_v L_v$, $R = \bigcup_v R_v$, $M = \bigcup_v M_v$, where the union is taken over all internal nodes v of $\mathcal{T}$.

Let q be a query point, and let z be the leaf of $\mathcal{T}$ such that $q \in \sigma_z$. Then

$$\omega(S(q)) = \omega(L(q)) + \omega(R(q)) + \omega(M(q)) + \omega(S_z(q)).$$

For each leaf z, we maintain a linked list of the intervals in S_z. Since each point of P is an endpoint of at most one interval, $|S_z| = O(\log n)$. To compute $\omega(S_z(q))$ we traverse S_z and sum the weights of all intervals in $S_z(q)$.

Below we describe the secondary structures stored at each node in order to compute $\omega(L(q))$, $\omega(M(q))$, and $\omega(R(q))$ efficiently. We first describe the secondary data structures for the middle intervals, which are new. The secondary data structures for the left intervals and right intervals are similar to the ones we had when we used a binary base tree in Section 3.1, but some additional ideas are needed to cope with the large fanout.

Middle intervals. At each internal node v, we use several *multislab* BST structures to store the middle intervals. First, for each pair of children v' and v'' of v with $v' \leq v''$, we have a multislab structure $M_v(v', v'')$ storing the subset of all intervals $s \in S_v$ such that $s^m = [b^-(v'), b^+(v'')]$. (We order these intervals arbitrarily in the BST.) We maintain these $\binom{f}{2} = O(\log n)$ multislab structures in a linked list. Since each interval in M_v is stored in exactly one multislab, the total size of these multislab structures at v is $O(|M_v|)$.

Furthermore, for each child v' of v, we have a *slab* BST structure $M_v(v')$. The structure $M_v(v')$ has an element for each pair of children v_1 and v_2 of v, such that $v_1 \leq v' \leq v_2$. The weight of this element is $\omega(M_v(v_1, v_2))$. We keep a pointer from v' to $M_v(v')$. The size of each slab BST is $O(\log n)$ as there may be $O(\log n)$ pairs of children v_1 and v_2, such that $v_1 \leq v' \leq v_2$. Since there are $O(\sqrt{\log n})$ slab BST structures at v, the total size of all slab BST structures at v is $O(\log^{3/2} n)$. It follows that the total size of all multislab and slab BST structures at v is $O(|M_v| + \log^{3/2} n)$. Since we have $O(n/\log^{3/2} n)$ internal nodes, and each interval contributes a middle part only to one set M_v, all multislab and slab structures at all nodes v take $O(n)$ space.

Let $q \in \mathbb{R}$ be a query point. Since the sets M_v are pairwise disjoint, $\omega(M(q)) = \sum_{v\in\Pi_q} \omega(M_v(q))$.[2] Furthermore, $\omega(M_v(q))$ is exactly $\omega(M_v(v'))$ where v' is the child of v on Π_q. So to compute $\omega(M_v(q))$, we traverse the path Π_q as described before, and when we move from a node v to its child v', we query the structure $M_v(v')$ and obtain $\omega(M_v(v'))$ in $O(1)$ time. We add these values to obtain $\omega(M(q))$. The overall query time is $O(\log n)$.

Next, we consider updates. Suppose we insert or delete an interval s whose middle part s^m exists. Let v be the node such that $s \in S_v$. (Recall that v can be computed in $O(\log n)$ time.) Let $s^m = [b^-(v'), b^+(v'')]$. We find the multislab BST $M_v(v', v'')$ in $O(\log n)$ time by searching the list of the multislab structures. Then we insert s into or delete s from $M_v(v', v'')$ and then query this BST for the updated weight $\omega(M_v(v', v''))$. Finally for every w with $v' \le w \le v''$, we update the weight of the element corresponding to the pair v', v'' in the slab BST $M_v(w)$, to be $\omega(M_v(v', v''))$. This update takes $O(\log\log n)$ time for each w, and $O(\sqrt{\log n} \cdot \log\log n) = O(\log n)$ for all slab structures.

LEMMA 3.2. *The set M of middle intervals is stored in multislab and slab BST structures of the internal nodes of $\mathcal{T}$ so that a stabbing-semigroup query on the middle intervals can be answered in $O(\log n)$ time. Furthermore, when a segment s is inserted or deleted and s^m exists, s^m can be inserted into or deleted from these secondary structures in $O(\log n)$ time.*

Left intervals. We now describe the secondary data structures for maintaining the left intervals. The secondary data structures for maintaining the right intervals are symmetric. To store the left intervals, we follow the same approach as in Section 3.1. The large fanout causes additional complications, however.

Decomposition of intervals. We define L_{vu} and Φ_v the same way as in (3.1) and (3.2), respectively, but bear in mind that the intervals in these sets are different because the base tree is different now. Note that (3.3) still holds. Since each node has f children, we need to refine (3.4). For a node v, let $\Lambda(v)$ be the set of siblings of v that precede v. For a leaf z, let $\mathcal{L}_z$ be the set of intervals in L that have their left endpoints in σ_z. The following lemma generalizes (3.4). See Figure 3.3.

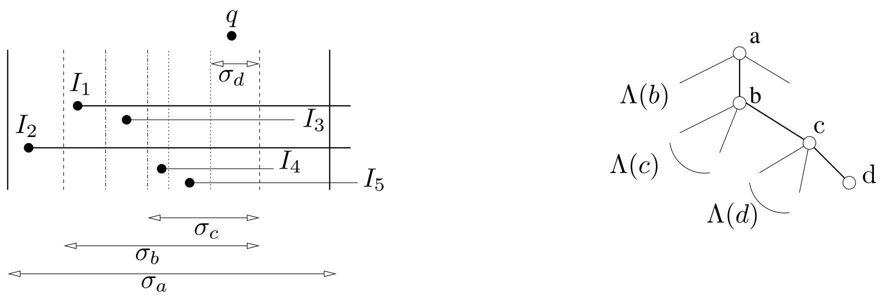

Fig. 3.3. Querying among left intervals; $\bigcup_{y\in\Lambda(b)} \Phi_y = \{I_2\}$, $\bigcup_{y\in\Lambda(c)} \Phi_y = \{I_1, I_3\}$, $\bigcup_{y\in\Lambda(d)} \Phi_y = \{I_4, I_5\}$.

[2]Recall that $M_z = \emptyset$ for the leaf $z \in \Pi_q$.

LEMMA 3.3. *Let q be a query point, let Π_q be the search path of q in $\mathcal{T}$, and let z be the leaf node in Π_q. Then*

$$\omega(L(q)) = \sum_{v \in \Pi_q} \sum_{y \in \Lambda(v)} \omega(\Phi_y) + \omega(\mathcal{L}_z(q)). \tag{3.5}$$

Proof. Clearly the sets Φ_y whose weights we sum in (3.5) are disjoint, and are also disjoint from $\mathcal{L}_z$. So it suffices to show that every left interval $s = [a, x]$ that contains q is contained in either $\mathcal{L}_z$ or one of the sets Φ_y.

Let v be the deepest common node of Π_q and Π_a. If v is a leaf, then $v = z$, $q \in \sigma_z$, and $s \in \mathcal{L}_z$. Otherwise, let v' be the node following v on Π_a and v'' the node following v on Π_q. By the definition of v, $v' \neq v''$, and since s contains q, $v' < v''$. Moreover, $s \in L_u$ for some proper ancestor u of $v = p(v')$; otherwise s must be $[a, b^+(v')]$ and cannot contain q. So $s \in \Phi_{v'}$ and $v' \in \Lambda(v'')$.

For the converse, it is easy to verify that every $s \in \Phi(y)$ for $y \in \Lambda(v)$ and $v \in \Pi_q$, contains q. ☐

Secondary structures. As in the binary case we maintain BST structures that allow us to obtain $\omega(\Phi_v)$ in $O(1)$ time, for every node v. But two new difficulties arise. The first is that the ability to get $\omega(\Phi_v)$ in $O(1)$ time is not sufficient to answer a query in logarithmic time. By Lemma 3.3, to guarantee logarithmic query time we have to be able to compute $\sum_{y \in \Lambda(v)} \omega(\Phi_y)$ in $O(\log\log n)$ time for any node v. Since $|\Lambda(v)| = O(\sqrt{\log n})$, computing this explicitly would be too slow. The second difficulty is how to update the values $\omega(\Phi_v)$. We introduce the following secondary structures to address these difficulties.

- $\mathcal{L}_z$: We store the list $\mathcal{L}_z$ at z. This list has size $O(\log n)$ and allows us to compute the second term of (3.5) in $O(\log n)$ time.
- A_v: As in the binary case, for each internal node v we maintain the set

  $$A_v = \{u \in \mathcal{T} \mid u \text{ is a proper ancestor of } p(v)\}$$

  in a BST where the weight of u in this structure is $\omega(L_{vu})$. So it follows from (3.2) that $\omega(A_v) = \omega(\Phi_v)$. The size of A_v is $O(\log n / \log\log n)$ for our non-binary base tree.
- B_v: For each internal node v we maintain a BST, B_v, over the children of v where the weight of a child w is $\omega(\Phi_w)$. By a PREFIXSUM(w) query to B_v with a child w of v, we get $\sum_{y \in \Lambda(w)} \omega(\Phi_y)$ in $O(\log\log n)$ time. With these BSTs we can answer a query in a straightforward way using (3.5). The size of B_v is $O(\sqrt{\log n})$.
- C_{vu}: To be able to efficiently update the BSTs on A_v's, hence also the B_v's, we introduce a third secondary structure, C_{vu}, for every pair of nodes v and $u \in A_v$. In C_{vu} we have an element for each child w of v, whose weight is $\omega(L_{wu})$. It is clear that $\omega(C_{vu}) = \omega(L_{vu})$. The size of C_{vu} is $O(\sqrt{\log n})$.
  The reason for introducing these C_{vu}'s is the following. When we insert an interval into L_u or delete an interval $s = [a, x]$ from L_u, the weights $\omega(L_{vu})$ may change for some nodes v, on the path from u to the leaf containing a. Let $v_1, \dots, v_f$ be the children of v, we have

  $$\omega(L_{vu}) = \sum_{i=1}^{f} \omega(L_{v_i u}),$$

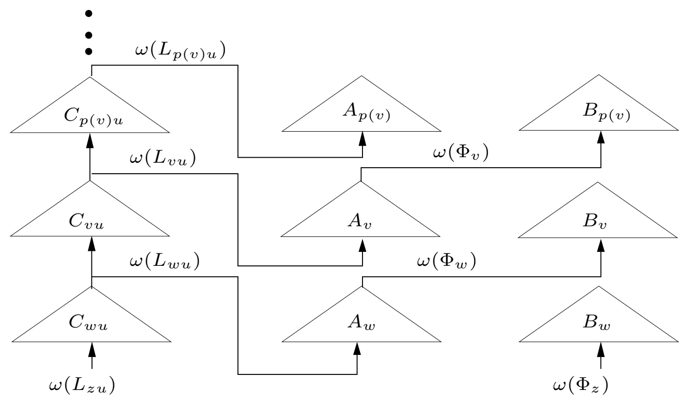

Fig. 3.4. Updating the secondary structures of left intervals. Here z is a leaf, $w = p(z)$ and $v = p(w)$.

and the C_{vu}'s are connected in this way that allows for efficient updates. When $\omega(L_{v_i u})$ changes for some i, we update the weight of v_i in C_{vu}. Then we get the new value of $\omega(L_{vu})$. Once we have $\omega(L_{vu})$ we update the weight of v in $C_{p(v)u}$, and the process continues upward. We also use the new value of $\omega(L_{vu})$ to update the weight of u in A_v. Then from A_v we obtain $\omega(\Phi_v)$ and update the weight of v in $B_{p(v)}$. See Figure 3.4 for an illustration of this process.

We now argue that all secondary structures representing L require $O(n)$ space. The total size of all the $\mathcal{L}_z$ lists is clearly $O(n)$. Consider an internal node v. The structure A_v has size $O(\log n/\log\log n)$; the structure B_v has size $O(\sqrt{\log n})$; we have $O(\log n/\log\log n)$ structures C_{vu}, one for every ancestor u of v each of size $O(\sqrt{\log n})$, so together they occupy $O(\log^{3/2} n/\log\log n)$ space. Summing this bound over all $O(n/\log^{3/2} n)$ internal nodes in $\mathcal{T}$ we get that the total size of all the secondary structures is $O(n)$.

After getting the relationship of all the secondary structures right, the query and update procedures are relatively straightforward. We nevertheless describe them here for completeness.

Answering a query. Let $q \in \mathbb{R}$ be a query point. We compute $\omega(L(q))$ using (3.5). We traverse the path Π_q in a top-down manner. For an internal node $v \in \Pi_q$ that is followed by $v' \in \Pi_q$, we perform a query PREFIXSUM(v') on B_v and get $\sum_{y\in\Lambda(v')}\omega(\Phi_y)$. When we reach the leaf $z \in \Pi_q$, we scan the list $\mathcal{L}_z$ stored at z and compute $\omega(\mathcal{L}_z(q))$ by summing the weights of all intervals in $\mathcal{L}_z$ that contain q. We then sum the values obtained at the nodes of Π_q and return the result. The overall query time is $O(\log n)$ since we spend $O(\log\log n)$ time at each of the $O(\log n/\log\log n)$ nodes on Π_q, and $O(\log n)$ time at the leaf that is the last node on Π_q.

Updating L. Suppose we are to insert or delete an interval $s = [a, b]$. Assume that $s \in S_u$ and let $[a, x]$ be the left interval of s which is in L_u. Let z be the leaf such that $a \in \sigma_z$. We first add/remove s to/from the list $\mathcal{L}_z$ and then traverse $\mathcal{L}_z$. We compute the new value of $\omega(L_{zu})$ by adding the weights of all left intervals in $\mathcal{L}_z \cap L_u$, and $\omega(\Phi_z)$ by adding the weights of all left intervals in $\mathcal{L}_z \setminus L_w$ where $w = p(z)$. Next, the weights of z in B_w is updated to $\omega(\Phi_z)$ and its weight in C_{wu} to $\omega(L_{zu})$. After having the updated value of $\omega(C_{wu})$, we first update the weight of u in A_w to $\omega(L_{wu})$, and then retrieve the new value of $\omega(A_w) = \omega(\Phi_w)$ from A_w.

Next, we continue to the parent v of w. We update the weight of w in B_v to be $\omega(\Phi_w)$. We then update the weight of w in C_{vu} to be (the updated) $\omega(C_{wu})$, and from C_{vu} we obtain the new value of $\omega(L_{vu})$. Then we update the weight of u in A_v to be $\omega(L_{vu})$ and obtain the new value of $\omega(A_v) = \omega(\Phi_v)$. We continue updating the nodes on Π_a bottom-up, maintaining the invariant that after processing node v, we know the value of $\omega(L_{vu})$, we have updated A_v so that $\omega(A_v) = \omega(\Phi_v)$, and that the structures C_{vu} and B_v are updated. We stop when we reach the child of u on Π_a.

We spend $O(\log n)$ time to update the leaf z since the length of $\mathcal{L}_z$ is $O(\log n)$. At each internal node we update and query BST structures of size $O(\log n)$ so these operations take $O(\log \log n)$ time, and overall the update procedure takes $O(\log n)$ time.

LEMMA 3.4. *The set L of left intervals can be maintained in secondary structures using linear space so that a stabbing-semigroup query with respect to L can be answered in $O(\log n)$ time. A left interval can be inserted into or deleted from L in $O(\log n)$ time.*

Using $\mathcal{T}$ and all secondary structures together (Lemmas 3.2 and 3.4) as well as the lists representing S_z at each leaf z, we obtain the main result of this section.

THEOREM 3.5. *A set S of n intervals, whose endpoints belong to a fixed set of $O(n)$ points, can be maintained in a data structure of linear size so that a stabbing-semigroup query can be answered in $O(\log n)$ time. An interval can be inserted into S or deleted from S in $O(\log n)$ time.*

4. Rebalancing the Base Tree.

In the previous section we assumed that the endpoints of the input intervals are from a fixed set of $O(n)$ points, and thus the base tree $\mathcal{T}$ was static. In this section we show how to remove this restriction, that is, how to insert the endpoints of a new interval s into $\mathcal{T}$ before we insert s into the secondary structures, and how to delete the endpoints of s from $\mathcal{T}$ after we delete s from the secondary structures.

The easiest way to handle deletions is using the standard technique of global rebuilding. Say $\mathcal{T}$ contained n intervals when we last rebuilt it. When deleting an endpoint we simply mark it as deleted in the node of $\mathcal{T}$ that contains it. 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 into the secondary structures of the new structure one by one. We can rebuild the base tree (without the secondary structures) $\mathcal{T}$ in $O(n)$ time and perform $\Theta(n)$ insertions in $O(n \log n)$ time to construct the secondary structures. Thus, the amortized cost of a deletion of an endpoint is $O(\log n)$. Since the fan-out $f = \lceil \sqrt{\log_2 n} \rceil$ depends on n we also rebuild $\mathcal{T}$ when the number of intervals that it contains doubles since the last time it was rebuilt. So in between global rebuildings of $\mathcal{T}$ the number of intervals in $\mathcal{T}$ is between $n/2$ and $2n$ and f is fixed to be $\lceil \sqrt{\log_2 n} \rceil$ where n is the number of intervals that were in $\mathcal{T}$ right after its last rebuilding. There are no simple tricks to perform insertions easily, and the rest of this section is devoted to this task.

4.1. The base tree.

In order to handle insertions of endpoints, we make the base tree $\mathcal{T}$ a weight-balanced B-tree with branching factor f and leaf parameter $\log n$ [5]. The *weight* of a node v of $\mathcal{T}$ (not to be confused with the weight of an interval), denoted by 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) at level ℓ (leaves are at level 0) is between $\frac{1}{2} f^{\ell} \log n$ and $2 f^{\ell} \log n$. 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 arguments 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 3.2 and 3.4).

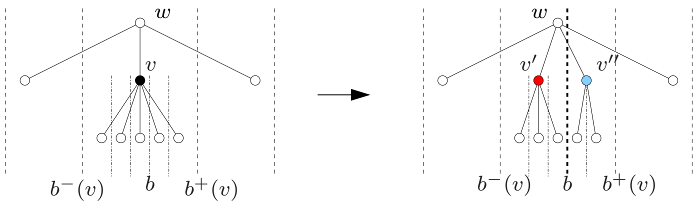

Fig. 4.1. Node v is split into two nodes v' and v'' at the slab boundary b; b becomes a slab boundary at $w = p(v)$ after the split, with $\sigma_{v'} = [b^-(v), b]$ and $\sigma_{v''} = [b, b^+(v)]$.

After an insertion of an endpoint into 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 the leaf z may become larger than $2\log n$, and the weight of an ancestor v of z at level ℓ may become larger than $2f^\ell \log n$. If the weight of z is too large, then we define a new slab boundary b, and split z along b into two leaves z' and z'' of weights $\log n$ and $\log n + 1$, respectively. If an internal node v at level ℓ becomes too large, then we split v along a slab boundary b into two nodes v' and v'' of weight roughly $f^\ell \log n$—more precisely, the weight of each of the two new nodes is between $(f^\ell - 2f^{\ell-1})\log n$ and $(f^\ell + 2f^{\ell-1})\log n$. In either case, b becomes a new slab boundary at $p(z)$ or $p(v)$, respectively, and we update the binary search tree of the slab boundaries at that node. Refer to Figure 4.1.

Next, we need to update all the affected secondary structures following the split. The procedures are different depending on whether a leaf or an interval node is split, and below we describe them separately. In either case, our goal is to update all the necessary secondary structures in $O(n_v \log\log n)$ time. Since we split a node v only once every $O(n_v)$ insertions of endpoints into its subtree, if we charge $O(\log\log n)$ time to each endpoint inserted into the subtree of v since its last split, then the total charges pay for the split. An endpoint may be charged by each ancestor of the leaf to which it belongs. This gives $O(\log n)$ charges in total per endpoint.

4.2. Splitting a leaf. Suppose we are splitting a leaf z into new leaves z' and z'' at a slab boundary $b = b^+(z') = b^-(z'')$, and let $w = p(z)$. This affects a sequence of secondary structures:

(i) It first affects the way how the intervals are associated with the nodes of $\mathcal{T}$. More precisely, the intervals in S_z that cross b will move to S_w, with the other intervals splitting into $S_{z'}$ and $S_{z''}$. An interval moving from S_z to S_w is broken into a left interval and a right interval (we will only talk about the left intervals below), which in turn affects the lists $\mathcal{L}_z, \mathcal{L}_{z'}, \mathcal{L}_{z''}$.

(ii) Since a slab at w is split into two, the secondary structures at w for the middle intervals M_w are affected.

(iii) Some secondary structures for the left intervals are also affected, including B_w and C_{wu} for proper ancestors u of w, because w now has one more child.

Note that although we have some new left intervals (generated from the intervals moving from S_z to S_w), they only appear in the $\mathcal{L}_{z'}, \mathcal{L}_{z''}$ lists, not the A_v, B_v, C_{vu}

structures. Below we describe how to perform all the necessary updates in detail.

The lists $S_z, S_{z'}, S_{z''}, S_w$. We split the list S_z of intervals with both endpoints in z into two lists: $S_{z'}$, containing intervals with both endpoints in z', and $S_{z''}$ containing intervals with both endpoints in z''. The other intervals in S_z have their left endpoint in z' and right endpoint in z'', and all of them now belong to S_w. Any such interval $s = [a_1, a_2]$ added to S_w breaks into a left interval $s^\ell = [a_1, b]$, and a right interval $s^r = [b, a_2]$. We add s^ℓ to the list $\mathcal{L}_{z'}$. We also split $\mathcal{L}_z$ into $\mathcal{L}_{z'}$ and $\mathcal{L}_{z''}$. An interval in $\mathcal{L}_z$ with a left endpoint in z' is added to $\mathcal{L}_{z'}$, and a left interval with a left endpoint in z'' is added to $\mathcal{L}_{z''}$. All these take time $O(\log n) = O(n_z)$ since S_z and $\mathcal{L}_z$ both have size $O(\log n)$.

The middle interval structures at w. The secondary structures for the middle intervals at w are affected by the split of z. Since each middle interval spanning the multislab defined by z and y for some child $y > z$, now spans the multislab defined by z' and y, the multislab BST $M_w(z, y)$ becomes $M_w(z', y)$. Similarly, the multislab BST $M_w(y, z)$ for $y < z$ becomes $M_w(y, z'')$. The multislab $M_w(z, z)$ also becomes $M_w(z', z'')$. These are merely notational changes.

Now suppose z has a right sibling z_r. Consider an interval $s \in S_w$ with left endpoint in z' and right endpoint not in z. If s^m existed before the split of z, then it must have been in a multislab $M_w(z_r, y)$ for some $y \geq z_r$. We delete s from $M_w(z_r, y)$ and insert it into $M_w(z'', y)$ instead. If s^m did not exist before the split of z, then after the split $s^m = [b^-(z''), b^+(z'')]$, so we insert s into the multislab structure $M_w(z'', z'')$. Similarly, suppose z has a left sibling z_ℓ, and consider an interval $s \in S_w$ with left endpoint not in z and right endpoint in z''. If s^m existed before the split of z, then it must have been in a multislab $M_w(y, z_\ell)$ for some $y < z_\ell$. We delete s from $M_w(y, z_\ell)$ and insert it into $M_w(y, z')$ instead. If s^m did not exist before the split of z, then after the split $s^m = [b^-(z'), b^+(z')]$, so we insert s into the multislab structure $M_w(z', z')$. See Figure 4.2.

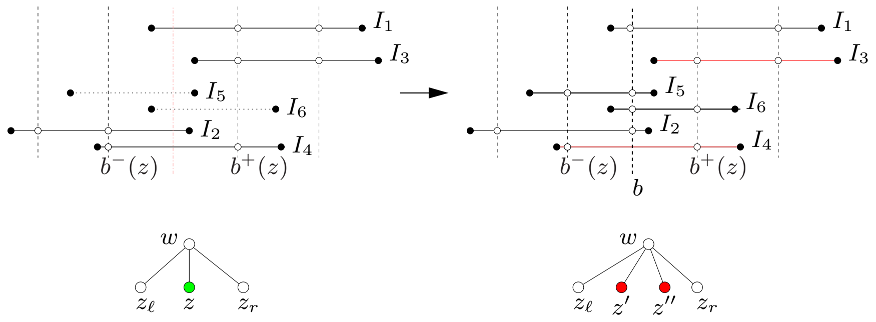

Fig. 4.2. Updating the multislabs at $w = p(z)$ when z splits. Before the split I_5^m, I_6^m are undefined and after the split $I_5^m \in M_w(z', z')$ and $I_6^m \in M_w(z'', z'')$. Before the split $I_1^m \in M_w(z_r, z_r)$ and after the split $I_1^m \in M_w(z'', z_r)$. Before the split $I_2^m \in M_w(z_\ell, z_\ell)$ and after the a split $I_2^m \in M_w(z_\ell, z')$. The intervals I_3^m, and I_4^m remain in the same multislabs.

After updating the multislab structures at w we continue and update the slab structures at w. Let $M_w(z_1)$ be a slab structure of a leaf $z_1 > z$. We delete from $M_w(z_1)$ all the pairs (z, y) for $y > z_1$ and insert the pairs (z', y) and (z'', y) instead with weights $\omega(M_w(z', y))$ and $\omega(M_w(z'', y))$, respectively. We update the slab structures $M_w(z_1)$ for $z_1 < z$ analogously. Finally we discard the slab structure $M_w(z)$ and construct two new slab structures $M_w(z')$ and $M_w(z'')$. The structure $M_w(z')$ contains all the pairs in $M_w(z)$ and pairs (y, z') for $y < z'$. Similarly, the structure $M_w(z'')$

contains all the pairs in $M_w(z)$ and pairs (z'', y) for $z'' < y$. The weight of each pair (y_1, y_2) is $\omega(M_w(y_1, y_2))$. This completes the updates to the data structures representing middle intervals. Note that the middle intervals at nodes other than w are not affected.

The updates to the middle intervals structures at w take $O(\log n \log\log n) = O(n_z \log\log n)$ time. We delete and insert $O(\log n)$ intervals into multislab structures $M_w(z'', \cdot)$ and $M_w(\cdot, z')$ in $O(\log n \log\log n)$ time. We perform $O(\sqrt{\log n})$ deletions and insertions of pairs containing z, z', and z'', into each slab structure $M_w(z_1)$ for $z_1 \neq z', z''$. As there are $O(\sqrt{\log n})$ slab structures, and each update take $O(\log\log n)$ time, all updates to slab structures take $O(\log n \log\log n)$ time. We also construct from scratch the slab structures $M_w(z')$ and $M_w(z'')$, by inserting $O(\log n)$ pairs each in $O(\log\log n)$ time.

Left interval structures. Consider now the left interval structures affected by the split of z. These structures are B_w and C_{wu} for proper ancestors u of w. Other structures for the left intervals are not affected by the split.

We first compute $\omega(\Phi_{z'})$ by summing the weights of all the intervals in $\mathcal{L}_{z'}$ that are not in L_w. We compute $\omega(\Phi_{z''})$ similarly. Then we delete z from B_w and insert z' and z'' with weights $\omega(\Phi_{z'})$ and $\omega(\Phi_{z''})$, respectively. This step takes $O(\log n)$ time.

To update C_{wu}, we sort the intervals s in $\mathcal{L}_{z'}$ according to the ancestor u of w where $s \in L_u$. For each proper ancestor u of w, we sum the weights of all intervals in L_u and obtain $\omega(L_{z'u})$. Similarly we obtain $\omega(L_{z''u})$. Then we delete z from C_{wu} and insert z' and z'' instead with weights $\omega(L_{z'u})$ and $\omega(L_{z''u})$, respectively. Note that $\omega(L_{wu})$ does not change so A_w is not affected. This step takes $O(\log n \log\log n)$ time: It takes $O(\log n \log\log n)$ time to sort and traverse $\mathcal{L}_{z'}$ and $\mathcal{L}_{z''}$, and it takes $O(\log n)$ time to update the $O(\log n/\log\log n)$ structures C_{wu}, each in time $O(\log\log n)$.

This completes the description of the updates when we split a leaf.

4.3. Splitting an internal node. Consider now a split of an internal node v into v' and v'' at a slab boundary b. Let w be the parent of v. The changes needed following this split are similar to those following a leaf split, but the details are more involved. On the high level, we still have the following three steps:

(i) The split first affects the way how the intervals are associated with the nodes of $\mathcal{T}$: The intervals in S_v that cross b will move to S_w, while the others are partitioned into $S_{v'}$ and $S_{v''}$. An interval moving from S_v to S_w is split into a left interval and a right interval at b (we will only talk about the left intervals below). Unlike the leaf split case, here the new left intervals are not only stored in the $\mathcal{L}_z$ lists, but also some A_v, B_v, C_{vu} structures, which need to be updated.

(ii) As in the leaf-split case, a slab at w is split into two slabs, which affects the secondary structures built at w on the middle intervals. Furthermore, since v' and v'' are two new internal nodes, their middle interval structures need to be built.

(iii) Many left interval structures are also affected, and there are three types of changes we need to perform. First, each interval moving from S_v to S_w has a new left interval, so we update these left intervals in the secondary structures. Second, v is split into v' and v'', so we discard A_v, B_v, C_{yv} (for every descendant y of v) and rebuild $A_{v'}, A_{v''}, B_{v'}, B_{v''}$, $C_{yv'}$ (for every descendant y of v'), $C_{yv''}$ (for every descendant y of v''), $C_{v'u}$ and $C_{v''u}$ (for every ancestor u of w). Finally, analogous to the leaf-split case, a child of w is being split, so B_w and all the C_{wu}'s for proper ancestors u of w are updated.

We now describe in detail how all the necessary updates are performed.

The lists $S_v, S_{v'}, S_{v''}, S_w$. The list S_v is partitioned as follows. Let $s = [a_1, a_2]$ be an interval in S_v; $a_1, a_2 \in \sigma_v$. If both a_1, a_2 lie in $\sigma_{v'}$ (resp. $\sigma_{v''}$), then s is moved to $S_{v'}$ (resp. $S_{v''}$); if s intersects the common boundary b of $\sigma_{v'}$ and $\sigma_{v''}$, then s is moved to S_w. When associated with v, s had a left interval, a right interval, and possibly a middle interval when associated with v. When moving to w, the middle interval disappears, while the left and right intervals become $s^\ell = [a_1, b]$ and $s^r = [b, a_2]$, possibly longer than they were previously. See Figure 4.3. We update the left interval with the endpoint a in the list $\mathfrak{L}_z$ where $a \in \sigma_z$ to this new s^ℓ (recall that we have a pointer from s to its left endpoint a, which is stored at z). These steps take $O(|S_v|) = O(n_v)$ time.

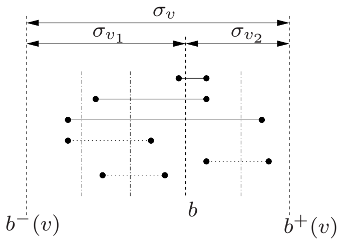

Fig. 4.3. Solid intervals move from S_v to S_w; the left interval derived from each of these intervals changes. The right endpoint of the new left intervals is b. Dotted intervals are now in $S_{v'}$ or $S_{v''}$; their partitions into left, middle, and right intervals do not change.

The middle interval structures at v', v'', w. The multislabs of v' and v'' are multislabs of v corresponding to pairs of children of v that are either both children of v' or both children of v''. They can be copied over directly. We can obtain the slab structures of v' and v'' from slab structures of v. Let y be a child of v'. We obtain $M_{v'}(y)$ by deleting from $M_v(y)$ all pairs which are not both children of v'. We construct the slab structures of children of v'' similarly. All of these operations take time at most $O(|S_v|) = O(n_v)$.

We also update the middle-interval structures at w. This is analogous to update procedure for a leaf split. Here, to identify the middle intervals of the multislabs $M_w(v'', y)$, for the right siblings y of v'', we traverse the subtree of v' to find all intervals whose other endpoint is in σ_w but was not in σ_v. When we find an interval $s \in S_w$ such that s^m after the split is not empty, starting at $b^+(v')$ and ending at $b^+(y)$, we add s^m to $M_w(v'', y)$. We identify middle intervals of the multislabs $M_w(y, v')$ analogously.

We claim that updating middle-interval data structures at w takes $O(n_v \log\log n)$ time. Indeed, inserting $O(n_v)$ pairs into multislab structures $M_w(v'', \cdot)$ and $M_w(\cdot, v')$, after deleting them from the multislab structures that previously contained them, takes $O(n_v \log\log n)$ time. We obtain each of the $O(\sqrt{\log n})$ slab structures of v' and v'' by $O(\log n)$ updates to slab structures of v in $O(\log^{3/2} n \log\log n)$ time. Since $n_v = \Omega(\log^{3/2} n)$, this bound is also $O(n_v \log\log n)$. Constructing the slab structures $M_w(v')$ and $M_w(v'')$ takes $O(\log n \log\log n)$ time, which is again bounded by $O(n_v \log\log n)$.

The left interval structures. We update the affected left interval structures by traversing the subtrees of v' and v'' (i.e., the former subtree of v) bottom-up. For each leaf z in the subtree of v', we scan $\mathfrak{L}_z$ and identify the intervals in $\mathfrak{L}_z \cap L_{v'}$. We sum the weights of these intervals and obtain $\omega(L_{zv'})$. At each proper descendant y

of v', we discard C_{yv} and construct $C_{yv'}$ by inserting every child y' of y into $C_{yv'}$ with weight $\omega(L_{y'v'})$. From the new $C_{yv'}$ we get $\omega(L_{yv'})$, which allows us to continue to build the $C_{yv'}$ structures upward until reaching v'. Meanwhile, we also delete v from A_y and insert v' instead, with weight $\omega(L_{yv'})$. In addition, letting $w = p(v)$, we rebuild the C_{yw} structures for all descendants y of v' during this bottom-up traversal, since L_w now contains new left intervals—the ones that intersect the new slab boundary b. Finally, we also update the weight of w in A_y to be $\omega(L_{yw})$. Similarly, we perform these changes for every descendant y of v''. It is easy to verify that all the updates during this bottom-up traversal takes $O(n_v)$ time. Recall that n_v is the number of points in the subtree of v and the number of internal nodes in this subtree is only $O(n_v/\log n)$.

After we have completed the rebuilding of the structures C_{yw}, $C_{yv'}$, and $C_{yv''}$, and updated A_y for all y in the former subtree of v, we also rebuild the structures B_y for all nodes y in this subtree. For each node y and a child y' of y, we insert y' into B_y with the updated weight $\omega(A_{y'})$. The time for this is proportional to the number of nodes in the subtree of v, which is $O(n_v/\sqrt{\log n})$.

Consider now ancestors u of w. For each such u, we build the structures $C_{v'u}$ and $C_{v''u}$: We insert each child y of v' into $C_{v'u}$ with weight $\omega(L_{yu})$, and similarly build $C_{v''u}$. Then we construct $A_{v'}$ and $A_{v''}$ by inserting each proper ancestor u of v' and v'' to $A_{v'}$ and $A_{v''}$ with weights $\omega(L_{v'u})$ and $\omega(L_{v''u})$, respectively. (We also associate the occurrence of u in $A_{v'}$ with $C_{v'u}$ and similarly for $A_{v''}$.) Lastly, we construct $B_{v'}$ and $B_{v''}$. The time taken by these operations is proportional to the total size of the structures that we construct, which is

$$O(\sqrt{\log n}\log n/\log\log n) = O(\log^{3/2} n) = O(n_v).$$

Finally, for every proper ancestor u of w we update C_{wu}. We delete v from C_{wu} and insert v' and v'' instead, with weights $\omega(L_{v'u})$ and $\omega(L_{v''u})$, respectively. Then we make corresponding updates to B_w.

Putting it together. This completes the description of the split of a node v, whether it is a leaf or an internal node. The time it takes to perform the split is $O(n_v \log\log n)$; the split time at an internal node is dominated by the time it takes to rearrange the middle intervals in the subtree of v into their multislab structures. By the earlier argument, this translates to an amortized cost of $O(\log n)$ per insertion. Combining this with the global rebuilding technique mentioned in the beginning of the section implies that the amortized cost of a delete operation is also $O(\log n)$. We thus obtain the following.

Theorem 4.1. *A set of n intervals can be maintained in linear-size data structure so that a stabbing-semigroup query can be answered in $O(\log n)$ time worst-case, and an interval can be inserted or deleted in amortized $O(\log n)$ time.*

5. External Memory Structure. Let B be the size of a block that we can transfer in one I/O operation from external to internal memory. Our data structure can be easily adapted to external memory by using the following parameters: we let each leaf of the base tree contain $B\log_B n$ endpoints, and let the fan-out f be $\max\{\sqrt{\log_B n}, \sqrt{B}\}$, that is, when $n < B^B$, the fan-out is fixed at $\sqrt{B}$; as n gets larger than B^B, the fan-out increases at $\sqrt{\log_B n}$. We also change the fan-out of all the BST structures to B, so that we can update and query a BST structure on m elements in $O(\lceil \log_B m\rceil)$ I/Os. To see that the analysis goes through, consider the following two cases.

Case 1: $\log_B n \geq B$**.** In this case the fan-out is $f = \sqrt{\log_B n}$. Noticing that $\log_B \log_B n \geq 1$, one can verify that the analysis in Sections 3 and 4 goes through, by replacing every base-2 logarithm with a base-B logarithm.

Case 2: $\log_B n < B$**.** In this case by our choice of f, the height of the base tree $\mathcal{T}$ is $O(\log_B n)$, that is, it is essentially a B-tree. Recall that we attach secondary structures to the base tree for the middle intervals M and the left intervals L and handle them separately. We consider them one by one.

Since there are $O(f^2) = O(B)$ multislabs at each node v, the middle interval structures at v now answer a query in $O(1)$ I/Os, so the total query cost is $O(\log_B n)$ I/Os. The total update cost is also $O(\log_B n)$. The size of the middle interval structures at v takes $O(1 + (|M_v| + f^3)/B)$ blocks. Since there are $O(n/(Bf \log_B n))$ internal nodes and middle interval structures exist at only the internal nodes, the total size of all these structures is still $O(n/B)$ blocks.

Next we consider the secondary structures for the left intervals. We spend $O(1)$ I/Os to query B_v at each node v on a root-to-leaf path, and spend $O(\log_B n)$ I/Os at the leaf, so the total query cost is still $O(\log_B n)$ I/Os. To perform an update, we update one A_v, one B_v, and one C_{uv} structure at each node on a root-to-leaf path. The cost is $O(\lceil \log_B \log_B n \rceil)$ I/Os for updating A_v, which is $O(1)$ since $\log_B n < B$. The cost to update a B_v or a C_{uv} structure is also $O(1)$ I/Os. So the total update cost, summed over all nodes, is $O(\log_B n)$ I/Os. The space requirement of these structures is $O(n/(B \log_B n)) = O(n/B)$ blocks.

Finally, it can be verified that the procedure in Section 4 spends $O(n_v \cdot \lceil \log_B \log_B n \rceil) = O(n_v)$ I/Os to split a node v during rebalancing, so the amortized cost of an update is $O(\log_B n)$ I/Os.

Theorem 5.1. *A set of n intervals can be stored in an external memory data structure using $O(n/B)$ blocks, so that a stabbing-semigroup query can be answered in $O(\log_B n)$ I/Os in the worst-case and each update can be performed in $O(\log_B n)$ amortized I/Os.*

6. Nested Intervals.

In this section we propose a simpler data structure, based on dynamic trees [18], for the special case in which the intervals in S are nested, i.e., at any given time, two intervals in S are either disjoint or one is contained in the other. It requires linear space, and each operation takes $O(\log n)$ time. Without loss of generality, we assume that S always contains the interval $\xi = [-\infty, \infty]$ whose weight is 0.

We define a *containment tree*, $\mathcal{C}$, on the intervals in S. Each interval in S is a node in $\mathcal{C}$, and the parent of an interval s is the smallest interval in S that contains s. We order the children of a node in increasing order of their left endpoints. We define the weight of an edge e in $\mathcal{C}$, denoted by $\omega(e)$, from an interval s to its parent to be $\omega(s)$; see Figure 6.1. For a node $s \in \mathcal{C}$, let $\Pi(s, \mathcal{C})$ be the path in $\mathcal{C}$ from the root to s, and let $p(s)$ denote the parent of s in $\mathcal{C}$. For a point $x \in \mathbb{R}$, let s_x be the smallest interval that contains x. By definition,

$$\omega(S(x)) = \sum_{e \in \Pi(s_x, \mathcal{C})} \omega(e). \tag{6.1}$$

A weakness of $\mathcal{C}$ is that an insertion or a deletion of an interval may require insertions and deletions of many edges. We therefore represent $\mathcal{C}$ by a binary tree $\mathcal{B}$, as follows. The nodes of $\mathcal{B}$ are the same as the nodes of $\mathcal{C}$. The left child of a node v in $\mathcal{B}$ is the first child of v in $\mathcal{C}$, or null if v is a leaf in $\mathcal{C}$. The weight of the edge

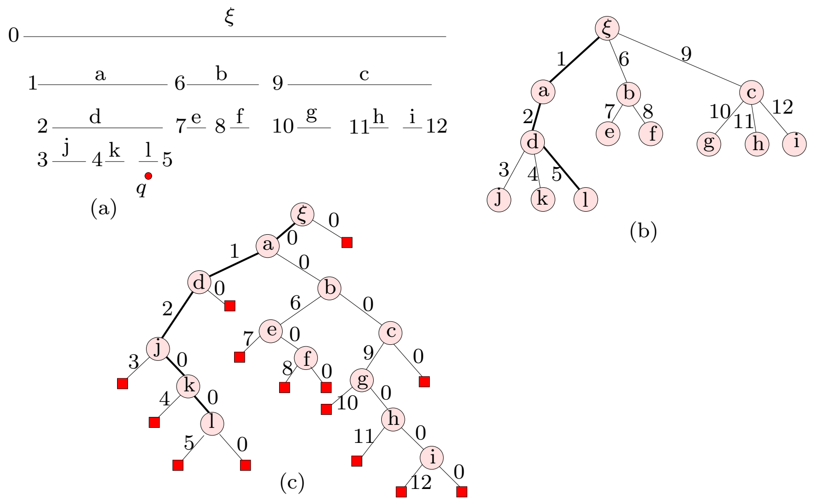
Fig. 6.1. (a) Nested intervals, numbers denote the weights of the intervals; $\xi = (-\infty, \infty)$ is the interval of weight 0 added to S; $s_q = l$. (b) Containment tree $\mathcal{C}$, bold path denotes $\Pi(s_q, \mathcal{C})$. (c) Binary tree $\mathcal{B}$, bold path denotes $\Pi(s_q, \mathcal{B})$.

between v and its left child is the weight of the interval associated with v. The right child of v in $\mathcal{B}$ is the right sibling of v in $\mathcal{C}$, or null if v is the rightmost child of its parent in $\mathcal{C}$. The weight of the edge from v to its right child is 0. For any (non-null) node $s \in \mathcal{B}$,

$$\sum_{e \in \Pi(s,\mathcal{B})} \omega(e) = \sum_{e \in \Pi(p(s),\mathcal{C})} \omega(e), \tag{6.2}$$

which implies that

$$\omega(S(x)) = \sum_{e \in \Pi(s_x,\mathcal{B})} \omega(e) + \omega(s_x). \tag{6.3}$$

See Figure 6.1. It is easy to verify that an insertion or a deletion of an interval requires only $O(1)$ insertion and deletions of edges to/from $\mathcal{B}$.

We maintain $\mathcal{B}$ as a dynamic tree data structure, introduced by Sleator and Tarjan [18].[3] Recall that dynamic trees support each of the following operations in $O(\log n)$ time:

- MinCost(v): finds the minimum cost of an edge on the path from v to the root of its tree.
- Link(v, w, c): v should be the root of a tree and w a node in another tree. This operation connects the tree containing v with the tree containing w by adding an edge with cost c between v and w with w being the parent.
- Cut(v): Splits the tree containing v by removing the edge from v to its parent.

We change the standard implementation of dynamic trees in a straightforward way so that weights of the edges are elements of a semigroup, and instead of MinCost(v) we support the following query:

[3]In this representation $\mathcal{B}$ is an unordered tree, that is, it does not distinguish between a right child of a node $v \in \mathcal{B}$ and its left child. This does not interfere with the correctness of the structure.

- SumCost(v): returns the sum of the weights of the edges on the path from v to the root of its tree.

We also store the endpoints of all intervals in a balanced search tree $\mathcal{T}$. If x is an endpoint of an interval $s \in S$, we store a pointer at the node of $\mathcal{T}$ that stores x to the node of $\mathcal{B}$ corresponding to the interval s. The overall size of the structure is linear.

Let $q \in \mathbb{R}$ be a query point. We compute $\omega(S(q))$ as follows. We first find in $O(\log n)$ time the predecessor x of q in $\mathcal{T}$. Suppose x is an endpoint of the interval $s \in S$. If x is the right endpoint of an interval s, then $s_q = p(s)$. By (6.1) and (6.2),

$$\omega(S(q)) = \sum_{e \in \Pi(s,\mathcal{B})} \omega(e),$$

and therefore we return the value of SumCost(s). If x is the left endpoint of an interval s, then $s = s_q$. In this case, by (6.3), we return $\omega(S(q)) = \text{SumCost}(s) + \omega(s)$.

To insert an interval $s = [a, b]$, we need to update both $\mathcal{T}$ and $\mathcal{B}$. We first update $\mathcal{B}$, and add to it a node representing s and then we insert a and b to $\mathcal{T}$. When we add a and b to $\mathcal{T}$, we also store pointers in the nodes containing a and b to the node containing s in $\mathcal{B}$.

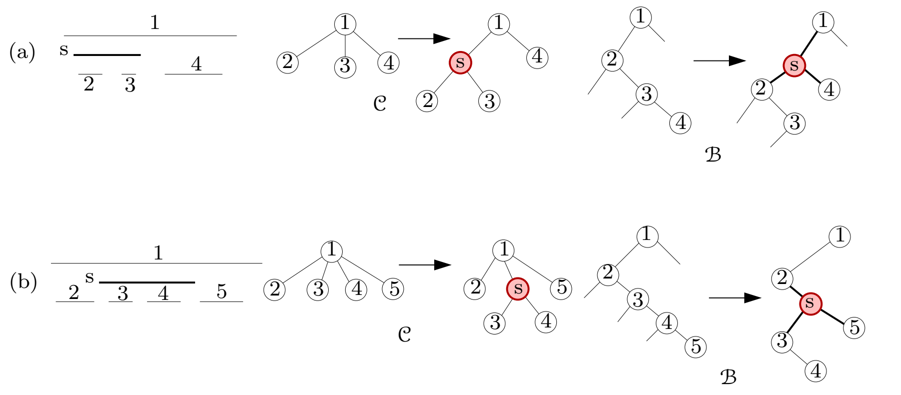

Fig. 6.2. Inserting an interval s: (a) s is the leftmost child of 1; $\ell^- = 1$, $\ell^+ = 2$, $r^- = 3$, and $r^+ = 4$. (b) s has a left sibling 2; $\ell^- = 2$, $\ell^+ = 3$, $r^- = 4$, and $r^+ = 5$. Thick lines indicate the newly created edges in $\mathcal{B}$.

We first find the predecessor and successor a^-, a^+ (resp. b^-, and b^+) of a (resp. b) in $\mathcal{T}$. Suppose a^-, a^+, b^-, and b^+ are the endpoints of the intervals ℓ^-, ℓ^+, r^-, and r^+, respectively. We allocate a new node for s and update its children as follows. (We also use s to refer to the node containing s when no confusion arises.) If $a^+ > b$, then s does not contain an interval of S, so s is a leaf of $\mathcal{C}$. Otherwise, ℓ^+ should be the leftmost child of s in $\mathcal{C}$ and r^- should be the rightmost child of s in $\mathcal{C}$. So we make ℓ^+ the left child of s in $\mathcal{B}$ by performing Cut(ℓ^+) followed by Link($\ell^+, s, \omega(s)$). The right child of s in $\mathcal{B}$ should be the right sibling of s in $\mathcal{C}$. If b^+ is the right endpoint of r^+, then s is the rightmost child of its parent in $\mathcal{C}$ so s does not have another child in $\mathcal{B}$. If b^+ is the left endpoint of r^+, then r^+ should be the next sibling of s in $\mathcal{C}$. So to update $\mathcal{B}$ we perform Cut(r^+) followed by Link($r^+, s, 0$). See Figure 6.2.

Finally, we set the parent of s in $\mathcal{C}$. If a^- is the left endpoint of ℓ^-, then s is the leftmost child of ℓ^- in $\mathcal{C}$, and we perform Link($s, \ell^-, \omega(\ell^-)$); see Figure 6.2 (a). Otherwise, ℓ^- is the left sibling of s in $\mathcal{C}$, and we perform Link($s, \ell^-, 0$); see Figure 6.2 (b). (In the latter case if ℓ^- was not the rightmost child of its parent in $\mathcal{C}$

before the insertion of s, then this right sibling of ℓ^- was either ℓ^+ or r^+. In either case we have made a cut such that ℓ^- has no right child prior to the link which made s its child in $\mathcal{B}$.)

This implementation of insert takes $O(\log n)$ time: By searching in $\mathcal{T}$, a^-, a^+, b^-, and b^+ can be computed in $O(\log n)$ time. Once we locate these points we perform a constant number of links and cuts which also take $O(\log n)$ time. The implementation of delete is similar.

Remark. By exploiting the internal structure of the dynamic trees [18], we can maintain additional information at each node of the dynamic tree so that there is no need for $\mathcal{T}$. Using this additional information we can find the predecessor and the successor of a point using the dynamic tree itself. We omit these details from this paper.

7. Lower Bounds.

In this section, we prove lower bounds for the dynamic *stabbing-group* problem, i.e., returning the sum of the weights of intervals containing a query point, but the weights of intervals are now drawn from a group and thus both addition and subtraction operations are allowed on the weights. Since it is easier to answer stabbing-group queries, these lower bounds hold for stabbing-semigroup queries as well. The lower bounds are proved in the cell-probe model, by using reductions from the partial-sum problem.

The *cell-probe* model, introduced by Yao [23], assumes that the memory is a collection of fixed-size cells (words). To perform a query or an update, the algorithm reads and writes cells of the memory, and the cost of the operation is simply the number of cells read and written. All other computation is free. We assume that a memory cell has $\Theta(\log n)$ bits, to ensure that n can be represented in one word. We also assume that any endpoint or weight of an interval is represented in one word.

The *partial-sum* problem asks to maintain an array $A[1..n]$ subject to the following two operations:
$\textsc{Update}(k, \Delta)$: Set $A[k]$ to be Δ, and
$\textsc{prefixsum}(k)$: Return $\sum_{i=1}^{k} A[i]$.
Pătraşcu and Demaine [16] proved the following lower bound for the partial-sum problem in the cell-probe model. Suppose the array elements belong to the group $\mathbb{Z}/n\mathbb{Z} = \{0, 1, \ldots, n-1\}$ with addition/subtraction modulo n. Let π be the *bit-reversal* permutation, i.e., $\pi(i)$ is the integer obtained by reversing the $\log_2 n$ bits of i (for simplicity assume that n is power of 2). Perform the following alternating $\textsc{Update}$ and $\textsc{prefixsum}$ operations. The i-th operation is $\textsc{Update}(\pi(i), \Delta)$ for odd i, where Δ is chosen uniformly at random from $\{0, \ldots, n-1\}$; and the operation is $\textsc{prefixsum}(\pi(i))$ if i is even. Note that even though the indices affected by the operations are fixed, the Δ values in the update operations define a distribution on input sequences. Let t_u and t_q be the expected amortized time of the $\textsc{Update}$ and $\textsc{prefixsum}$ operations, respectively, on this distribution of input sequences of a data structure for the partial-sum problem. Pătraşcu and Demaine [16] proved that $t_q \log(t_u/t_q) = \Omega(\log n)$ and $t_u \log(t_q/t_u) = \Omega(\log n)$, irrespective of the number of memory cells used by the data structure, initial preprocessing time of the data structure, and the initial values of $A[i]$. For simplicity, we assume that initially $A[i] = 0$ for all i.

A sequence of operations for the partial-sum problem can be solved by performing a sequence of insert and query operations on a dynamic stabbing-group data structure, for the group $\mathbb{Z}/n\mathbb{Z}$, as follows. For an $\textsc{Update}(k, \Delta)$ operation, we insert an interval $[k, n]$ with weight Δ; for a $\textsc{prefixsum}(k)$ query, we issue a stabbing-group query at

k. It is easy to verify that this solves the partial-sum problem on any sequence of operations. Hence, for any stabbing-group data structure with insert and query time t_i and t_q, respectively, $t_q \log(t_i/t_q) = \Omega(\log n)$ and $t_i \log(t_q/t_i) = \Omega(\log n)$.

We also show how to use the lower bound by Pătraşcu and Demaine [16] for partial sums to prove a lower bound on the trade-off between the query time and the deletion time in a deletion-only data structure for the stabbing-group problem. Specifically, we assume that a set of intervals preprocessed into a data structure such that we can delete intervals from the data structure and perform stabbing-group queries, and show a trade-off between the deletion time and the query time.

Let p be a prime number and let $n = p^2$. The weights are chosen from the group $\mathbb{Z}/p\mathbb{Z}$. Define a family S of n intervals $s_{i,j} = [i,p]$, for $0 \le i,j < p$, with weights $\omega(s_{i,j}) = j$. Suppose we have a stabbing-group structure $\mathcal{D}$ initialized to contains S that supports deletions and queries. Consider the sequence of p Update and prefixsum operations in the construction of [16] described above, with array size p and the group $\mathbb{Z}/p\mathbb{Z}$. We can simulate such a sequence by deletions and queries on $\mathcal{D}$ as follows. For Update(k,Δ), we delete the interval $s_{k,p-\Delta}$. Note that $\omega(s_{k,p-\Delta}) = p - \Delta$. For any k, we delete at most one interval among $s_{k,0},\ldots,s_{k,p-1}$, as each Update operation updates a different array element. For prefixsum(k), we issue a stabbing-group query at k. Before the above prefixsum operation was performed, suppose ℓ Update operations were performed on array elements with index at most k, with weights $\Delta_1,\ldots,\Delta_\ell$. Then the weight of the intervals currently in S that contain k is

$$kp(p-1)/2 - \sum_{i=1}^{\ell}(p-\Delta_i) \equiv \sum_{i=1}^{\ell}\Delta_i \pmod{p},$$

which is the same as the output of prefixsum(k).

The following theorem summarizes the lower bounds that we obtain via the reductions from partial sums that we described.

Theorem 7.1. *For any stabbing-group data structure in the cell-probe model storing n intervals, if the amortized insertion, deletion, and query times are t_i, t_d, and t_q, respectively, then the following trade-offs hold:*

$$t_q \log(t_i/t_q) = \Omega(\log n); \qquad t_i \log(t_q/t_i) = \Omega(\log n);$$
$$t_q \log(t_d/t_q) = \Omega(\log n); \qquad t_d \log(t_q/t_d) = \Omega(\log n).$$

Remark. Our deletion-query trade-off holds only for the amortized cost of the first $p = \sqrt{n}$ deletions in a data structure storing $n = p^2$ intervals. Although it is straightforward to prove the same trade-off for the first $O(n^{1-\epsilon})$ deletions, for any small constant ϵ, it seems difficult to extend the argument to the first $\Omega(n)$ deletions. However, we believe that the trade-off indeed still holds if $\Omega(n)$ deletions are considered.

8. Open Problems.

In this paper we consider data structures that work with any semigroup, and present an optimal solution. However, our lower bound does not prevent us from obtaining an improved structure with some special semigroups, such as the stabbing-max problem (using $(\mathbb{R}, \max)$), or the existence problem (using $(\{0,1\}, \vee)$). So far there are no better results on these special problems than our general-purpose stabbing-semigroup data structure in the pointer-machine model, although sub-logarithmic bounds can be obtained on a RAM [20]. Another interesting question to ask is the counting problem, which is the case where we use the group $(\mathbb{Z}/n\mathbb{Z}, +)$, but all weights are fixed to be one. Our lower bound does not hold for this case as it assumes that weights can be arbitrarily chosen from $(\mathbb{Z}/n\mathbb{Z}, +)$.

Acknowledgment. The authors thank Mihai Pătraşcu for helpful discussions on the lower bound of the problem. They also thank two anonymous referees for their useful comments.

REFERENCES

[1] P. K. Agarwal, L. Arge, and K. Yi. An optimal dynamic interval stabbing-max data structure? In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 803–812, 2005.

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

[3] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 534–543, 1998.

[4] L. Arge. External memory data structures. In *Handbook of Massive Data Sets*, pages 313–357. Kluwer Academic Publishers, 2002.

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

[6] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition.* The MIT Press, 2001.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwartzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[9] H. Edelsbrunner. A new approach to rectangle intersections, part I. *International Journal of Computer Mathematics*, 13:209–219, 1983.

[10] A. Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *Proc. IEEE INFOCOM*, pages 1193–1202, 2000.

[11] P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In *Proc. Annual European Symposium on Algorithms*, pages 107–120, 1996.

[12] P. Ferragina, S. Muthukrishnan, and M. de Berg. Multi-method dispatching: a geometric approach with applications to string matching problems. In *Proc. ACM Symposium on Theory of Computing*, pages 483–491, 1999.

[13] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[14] H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic rectangular intersection with priorities. In *Proc. ACM Symposium on Theory of Computing*, pages 639–648, 2003.

[15] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[16] M. Pătraşcu and E. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

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

[18] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[19] R. E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.

[20] M. Thorup. Space efficient dynamic stabbing with fast queries. In *Proc. ACM Symposium on Theory of Computing*, pages 649–658, 2003.

[21] J. S. Vitter. External memory algorithms and data structures. In *External memory algorithms*, pages 1–38. American Mathematical Society, 1999.

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

[23] A. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.