Dynamic Rectangular Intersection with Priorities*

Haim Kaplan
School of computer science,
Tel Aviv University
Tel Aviv, Israel
haimk@cs.tau.ac.il

Eyal Molad
School of computer science,
Tel Aviv University
Tel Aviv, Israel
eyalmolad@hotmail.com

Robert E. Tarjan
Department of computer science, Princeton University
Princeton NJ, and
Hewlett-Packard,
1501 Page Mill Road,
Palo Alto, CA.
ret@cs.princeton.edu

ABSTRACT

We present efficient data structures to maintain dynamic set of rectangles, each with priority assigned to it, such that we can efficiently find the rectangle of maximum priority containing a query point. Our data structures support insertions and deletions of rectangles. In one dimension, when rectangles are intervals, our most efficient data structure supports queries and insertions in $O(\log n)$ time, deletions in $O(\log n \log\log n)$ time and requires linear space. When intervals are guaranteed to be nonoverlapping (but one can be nested within the other) we obtain a simpler data structure that supports all operations in $O(\log n)$ time.

We can generalize these data structures to maintain rectangles with priorities in higher dimension using standard techniques based on segment trees. The space increases by a logarithmic factor per dimension, and we get a logarithmic slowdown for each operation per dimension. For nonoverlapping rectangles, we present a better data structure where we avoid the extra logarithmic factor on the query time.

The general rectangular intersection problem with priorities and its special case of nonoverlapping rectangles both arise in packet classification by IP routers.

Categories and Subject Descriptors

I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet-switching networks*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*

General Terms

Algorithms

*Research of Eyal Molad and Haim Kaplan was supported in part by the Israeli Science Foundation (grant No. 548).

STOC'03, June 9–11, 2003, San Diego, California, USA.

Keywords

Packet classification, rectangular point location, rectangular intersection, orthogonal range searching

1. INTRODUCTION

We consider the problem of maintaining a set of rectangles[1] each with an assigned priority such that given a point we can efficiently find a rectangle with maximum priority containing it. We focus on the dynamic version of the problem where we want a data structure that allows efficient insertions and deletions of rectangles. This problem is related to the computational geometry problem known as *rectangular point intersection.* In the rectangular point intersection problem we maintain rectangles such that we can efficiently report all rectangles containing a query point. One can use a data structure for rectangular intersection searching to solve our problem simply by traversing all rectangles containing the query point and picking the one with largest priority. However the performance of such a solution would depend on the cardinality of the set of rectangles containing the query. When we are only interested in one interval of highest priority we would like to embed the priorities into the data structure in a clever way such that the performance of a query would not depend on the number of rectangles containing the query point. In this paper we develop several techniques that allow us to design such data structures.

We first consider the one dimensional version of the problem where the rectangles are intervals, each with priority associated with it. In this case one can obtain a data structure based on an interval tree [8] by making the secondary structures function also as heaps with respect to priorities. The query time using this data structure is $O(\log^2 n)$. To get a faster solution we develop a technique to embed the secondary structures into the primary tree. Using this technique we obtain a linear space data structure which supports queries in $O(\log n)$ worst-case time, insertions in $O(\log n)$ amortized time, and deletions in $O(\log n \log\log n)$ amortized time.

Motivated by our main application (which we describe below) we also consider a special case of the problem where rectangles are restricted to be nonoverlapping. I.e. any pair of rectangles that are in the data structure at any one time are either disjoint or one contains the other. We call such

[1]We use the word rectangle for a hyper-rectangle in d dimensional space.

set of rectangles *nested*. For a nested set of intervals we describe a very simple data structure that requires linear space, and supports both query and updates in $O(\log n)$ time. We obtain this data structure using dynamic trees [13] modified to allow searches.

We also describe data structures that solve the problem in higher dimensions. By using a standard extension scheme via segment trees we get a logarithmic factor increase both in space and operation times for each additional dimension. For the two dimensional problem restricted to nested rectangles, we manage to beat the obvious $O(\log^2 n)$ time bound for queries. Specifically, we obtain a data-structure that supports queries in $O(\log n)$ time, updates in $O(\log^2 n)$ time, and requires $O(n \log n)$ space. We use a segment tree that breaks the two dimensional problem into many one dimensional problems. But unlike the naive approach we connect the one dimensional problems so that we do not search in each of them independently. The same technique can be applied to get a data structure for nested rectangles in a higher (constant) dimension, which has a logarithmic query time.

The main application of our data structure is for *packet classification* in IP networks. Typically, a router classifies each incoming packet into a *flow* according to one or more fields in the packet header. The router then processes all packets classified to the same flow in the same way. To do the classification, the router maintains a set of rules each with priority assigned to it. The highest priority rule which a packet obeys determines the flow of the packet.

The simplest example of packet classification is forwarding incoming packets into the right outgoing link of the router. For this purpose the router maintains a table of prefixes of IP addresses (the rules). To forward a packet the router finds the longest prefix (highest priority) which matches the destination address of the packet. That prefix determines the flow of the packet, and each flow has a designated next hop for its packets. This special case of packet classification is called *IP-lookup*.

One can think of these classification tasks as special cases of our rectangular point intersection problem. A rule which combines d fields of the packet header corresponds to a rectangle in d dimensions, with a priority assigned to it, and a packet corresponds to a query point. Given a query point we want to quickly determine the highest priority rectangle containing it. This rectangle corresponds to the rule that determines the flow of the packet.

The packet classification problem has been studied extensively mainly in the networking community. Existing packet classification schemes assume that rectangles and query points are defined by integers W bits long. Furthermore, many packet classification algorithms assume that rectangles are defined by prefixes of IP addresses.[2] Note that in one dimension a set of intervals defined by prefixes is a set of nested intervals. In particular in the IP-lookup problem the set of intervals is nested. Typically even in higher dimensions most rectangles would be nested, where smaller rectangles correspond to more specific rules with higher priority. These rules override the typical behavior for particular flows.

We distinguish between packet classification algorithms that are *static* and algorithms that are *dynamic*. Static algorithms build a data structure for a fixed set of rules, and when the rule set changes one has to rebuild the data structure. In contrast, dynamic algorithms build a data structure that supports efficient insertions and deletions of rules. The need for dynamic algorithms arises in various networking contexts such as, supporting multicast routing protocols and QoS protocols. In these applications if an update operation is not fast enough then incoming packets may need to be buffered until the data structure reaches a coherent state.

Most of the previous work on packet classification focus on static data structures (see the survey by Gupta and McKeown [6]). Several dynamic data structures for packet classification have also been suggested. Sahni and Kim [10] describe a one dimensional data structure for the IP-lookup subproblem (intervals defined by prefixes). Their data structure supports all operations in $O(\log n)$ time and requires linear space. Feldmann and Muthukrishnan [4] proposed Fat Inverted Segment tree (FIS). The dynamic one-dimensional 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 $O(n^{1+1/l})$, and insert and delete take $O(n^{1/l} \log n)$ time, but there is an upper bound on the total number of insertions and deletions allowed. [3]

Buddhikot, Suri, and Waldvogel [2] proposed Area-based quadtree (AQT) for packet classification in two dimensions, but it is restricted to rectangles defined by prefixes of IP addresses. AQT supports queries in $O(\alpha W)$ time, updates in $O(\alpha \sqrt[\alpha]{n})$ time, and requires $O(nW)$ space, where α is an integer parameter. Gupta and McKeown [5] propose two simple data-structures for packet classification in d dimensions, heap-on-trie (HoT) and binary-search-tree-on-trie (BoT). HoT achieves $O(W^d)$ query time and $O(W^d \log n)$ update time. BoT achieves $O(W^d \log n)$ query time and $O(W^{d-1} \log n)$ update times. Both data structures take $O(nW^d)$ space.[4]

Our data structures work in a much stronger computational model than the model assumed by all the algorithms mentioned above. We allow rectangles to be defined by arbitrary intervals, and query points to be arbitrary points in the plane. We manipulate the corners of the rectangles and the query points only via comparisons.[5] Furthermore, our data structures are pointer based and could be implemented in the pointer machine model of computation [14]. As a result we are not limited to rules that are defined by prefixes of IP addresses.

The strong computational model that we obey does not come at the cost of increased complexity. Our data structures, in particular the one dimensional data structures are simple. For a nested set of intervals in one dimension we describe a straightforward data structure based on dynamic trees which is very easy to implement. The strong computational model also does not come at the cost of inferior time bounds. Our asymptotic bounds are better than those of previous dynamic data structure in the more restricted

[2]That is the projection of a rectangle into every dimension is an interval of length 2^i whose starting point is an integer divisible by 2^i (for prefix of length $W - i$ bits).

[3]For other one-dimensional data structures see the survey by Sahni *et al* [11].

[4]Notice that data structures in one dimensions and two dimensions typically can be generalized to higher dimensions using segment trees, paying a logarithmic factor per dimension in time and space.

[5]We assume each comparison takes $O(1)$ time.

models. For example our one dimensional data structure for nested intervals solves within $O(\log n)$ time a problem more general than the dynamic IP-lookup solved within the same bounds by Sahni and Kim [10], as we support arbitrary nested set of intervals with arbitrary priorities.

Another application for our data structures is in the field of geographic information system (GIS). In GIS, it is common to maintain a database with a very large number of (possibly overlapping) 2D-images of a geographical area, where each image has a priority (representing say, its quality, or its age). The need is to maintain such a database under insertions and deletions of images, and to quickly report the image with the highest priority which contains a query point.

To simplify the presentation, we assume that all the endpoints of all the rectangles are disjoint. For a node x in a tree T we denote by $n(x)$ the number of descendants of x in T. We also denote by $p(x)$ the parent of x in T. When we use these notations the tree we refer to will be clear from the context. For a rectangle r we denote by r_x and r_y its projections to the x and y dimensions, respectively.

2. INTERVAL TREES

We start by briefly describing a classical geometric data structure called an *interval tree* ([3], also see e.g. [8]). We use interval trees to construct our data structures in Sections 3 and 5.

Interval trees store a set of intervals such that we can efficiently report all intervals intersecting a query point. Let S be the set of intervals, let $|S| = n$, and let X be the set of endpoints of the intervals in S. An interval tree for S consists of a primary balanced binary tree T_p where the leaves store the endpoints X, from left to right in symmetric order. We use the dynamic version of interval tree, where the points of X can be arbitrary points on a line. (The static version of an interval tree is for the case where X is a static set known in advance).

For each internal node $v \in T_p$ we define the followings. The split point of v, denoted by $split(v)$, is the search key of node v. I.e. $split(v)$ is a number such that the leaves of the left subtree of v store points smaller than $split(v)$, and the leaves of the right subtree of v store points larger than $split(v)$. The range of v, denoted by $range(v)$, is defined recursively as follows. The range of the root is $(-\infty, \infty]$. For a node v, where $range(v) = (l, r]$, the range of the left child of v is $(l, split(v)]$, and the range of the right child of v is $(split(v), r]$.

We associate each interval $s \in S$ with the unique node v of T_p such that s is contained in $range(v)$ but not in the range of any child of v. For each node $v \in T_p$ we denote the set of all intervals associated with v by $S(v)$. I.e. $S(v) = \{s \in S \mid split(v) \in s \subseteq range(v)\}$. We also note that $|S(v)| \leq n(v)$, since endpoints of intervals in $S(v)$ must be stored at the leaves of the subtree rooted at v.

For each $v \in T_p$ we maintain two secondary balanced search trees which represent $S(v)$. The first tree, $T_\ell(v)$, stores in its leaves the left endpoints of intervals in $S(v)$, sorted from left to right. The second tree, $T_r(v)$ stores in its leaves the right endpoints of the intervals in $S(v)$, sorted from right to left. It should be clear at this point that the interval tree requires $O(n)$ space, since each interval belongs to one set $S(v)$ and therefore its endpoints appear only in two secondary trees.

We answer a query consisting of a point q by traversing the search path for q in T_p. At each node v on the search path, we search $S(v)$ for all intervals in $S(v)$ that contain q. We perform this search as follows. If q is to the left of $split(v)$ we traverse the tree $T_\ell(v)$ from left to right and report all intervals whose left endpoint is to the left of q. The case where q is to the right of $split(v)$ is symmetric. It is straightforward to check that the query algorithm is correct, and that it takes $O(\log n + k)$ time where k is the number of intervals reported.

To make interval trees dynamic, i.e. to support insertions and deletions of intervals we implement T_p as a $BB(\alpha)$ tree [9]. A $BB(\alpha)$ tree is a kind of balanced binary tree, where for each node v, the ratio between *n(left child of v)* and *n(v)* is in $[\alpha, 1-\alpha]$, where $\alpha \in (\frac{1}{4}, 1 - \frac{\sqrt{2}}{2}]$. As other kinds of balanced binary trees, $BB(\alpha)$ trees allow access and update operations in worst-case logarithmic time. Furthermore, $BB(\alpha)$ trees have the following property [7] (see also [15]).

THEOREM 2.1. *Let the time of a rotation at node v be $n(v) \cdot f(n(v))$ for some non-decreasing function f. Then the amortized rebalancing time of an update (insert or delete) operation is $O(f(n(v)) \log n)$*

This property makes $BB(\alpha)$ trees especially useful for dynamization of augmented trees where each node stores a secondary data structure. Theorem 2.1 shows that with a $BB(\alpha)$ tree as the primary structure we can amortize expensive updates to the secondary structures during rotations.

Specifically for interval trees, we insert a new interval s to an interval tree by first inserting the endpoints of s to T_p. During this insertion we may need to perform a rotation in T_p. When performing a rotation around an edge $(v = p(u), u)$, the sets $S(v)$ and $S(u)$ change and therefore we rebuild the secondary structures of v and u. This rebuilding takes time proportional to $n(v)$ and therefore by Theorem 2.1 we obtain that each insertion takes $O(\log n)$ amortized time. After we inserted the endpoints of s into T_p we find the node w which is the lowest common ancestor of the endpoints of s and insert the left endpoint of s into $T_\ell(w)$ and the right endpoint of s into $T_r(w)$. We can implement the secondary structures by any kind of balanced search tree that supports logarithmic update time, to obtain that the total amortized time required for the insertion is $O(\log n)$. The implementation of delete is similar to the implementation of insert.

3. MAXIMUM PRIORITY QUERIES

We start by describing a straightforward data structure based on interval trees that is capable to find the interval of maximum priority containing the query point. This data structure supports queries in $O(\log^2 n)$ time, updates in $O(\log n)$ time, and it takes linear space. We obtain this data structure by making the secondary trees function also as tournaments with respect to priorities. Then we show how to improve the query time to $O(\log n)$ by embedding the tournaments into the primary structure. However to maintain this embedding the time per update operation increases to $O(\log n \log\log n)$.

We use interval trees as described in Section 2, and maintain the following additional information in the secondary trees. In every node a of a secondary tree ($T_\ell(v)$ or $T_r(v)$

for some $v \in T_p$), we maintain the maximum priority of the intervals corresponding to the nodes of the subtree rooted by a. We denote this number by $m(a)$. Using this additional information we can find the maximum priority of an interval in $S(v)$ containing a point q as follows. If $q < split(v)$, we traverse the search path for q in $T_\ell(v)$, and calculate the maximum of $m(a)$ over all nodes a hanging left of the search path (a left child of a node on the search path, but not on the search path itself). The case where $q > split(v)$ is symmetric.

To perform a query for a point q in the interval tree T, we traverse the search path for q in T_p. At each node v on the search path, we compute the maximum priority of an interval in $S(v)$ which contains a point q. The result is the interval with the highest priority among all these maximum priorities values. It is easy to see that a query takes $O(\log^2 n)$ worst case time, updates still take $O(\log n)$ amortized time, and the data structure requires linear space.

Next we improve the query time by embedding the secondary tournaments into the primary tree. First we describe a particularly simple data structure that supports only insertions of intervals. Then we extend it to allow deletions.

To support insertions it suffices to maintain in each node $v \in T_p$ two numbers $M_\ell(v)$ and $M_r(v)$ which we define as follows.[6] The number $M_\ell(v)$ is the maximum priority of an interval $[x, y]$ such that $x \in range(v)$ and $y \notin range(p(v))$. Note that if v is a left child, and u is the right sibling of v, than $range(u) \subseteq [x, y]$. Also note that x is stored in a descendant of v. Similarly, $M_r(v)$ is the maximum priority of an interval $[x, y]$ such that $y \in range(v)$ and $x \notin range(p(v))$.

To perform a query we traverse the search path defined by q in T_p. We calculate the maximum, M_ℓ, of $M_\ell(u)$ over all nodes u hanging left of the search path Similarly we calculate the maximum, M_r, of $M_r(u)$ over all nodes u hanging right of the search path. Let s be an interval whose priority is $max(M_\ell, M_r)$. Let s' be the interval whose endpoint is stored at the leaf where the search path of q ends. If s' contains q and has higher priority than s then the answer to the query is s'. Otherwise it is s. To insert a new interval $s = [x, y]$ to the data structure, we first insert the endpoints of s to T_p. While carrying out this insertion we may perform a rotation in T_p. When performing a rotation around an edge $(v = p(u), u)$, the ranges of v and u change and therefore we have to recalculate $M_\ell(w)$ and $M_r(w)$ for every node w which is a child of v or u. This can be done by scanning the subtree of T_p rooted by v, in $O(n(v))$ time. Next, we traverse the search path of x in T_p and in every appropriate node u on the search path, we set $M_\ell(u)$ to the maximum between the old value of $M_\ell(u)$ and the priority of the new interval. Similarly we traverse the search path of y in T_p and update $M_r(u)$ for nodes along this path. It is easy to see that all operations take $O(\log n)$ time.

3.1 Fully dynamic data structure

In order to support both insertions and deletions, one has to maintain the values $M_\ell(u)$ and $M_r(u)$ under deletions of intervals. One can do that by maintaining two heaps in every node u containing the set of intervals that determine $M_\ell(u)$ and $M_r(u)$, respectively. In order to reduce the size of these heaps, we group intervals together. We obtain the grouping by explicitly maintaining the secondary trees $T_\ell(v)$ and $T_r(v)$ for every $v \in T_p$, together with the maximum priority values, $m(a)$, in every node a of a secondary tree. Each node in a secondary tree would contribute one node to a single heap on behalf of all its descendant intervals.

Let v be a node in T_p. Let L be the left subtree of v. Recall that $T_\ell(v)$ represents the left endpoints of $S(v)$, which are stored at the leaves of L. Let L' be the subtree of L which consists of all leaves that store left endpoints of $S(v)$, and their ancestors in L. We redefine $T_\ell(v)$ to be a tree isomorphic to L'. Note that $T_\ell(v)$ is not necessarily binary and may contain nodes with a single child.

For every node $a \in T_\ell(v)$ we define $\sigma(a)$ to be its isomorphic image in T_p. The definition of $T_r(v)$ is similar. For every $v \in T_p$, we define $h_\ell(v)$ to be the set of nodes, $a \in T_\ell(u)$ for some $u \in T_p$ (ancestor of v), such that $\sigma(a) = v$. Similarly we define $h_r(v)$. We represent $h_\ell(v)$ and $h_r(v)$ as max-heaps, where the key of a node a in the heap is $m(a)$. Recall that $m(a)$ is the maximum priority of all intervals corresponding to the leaves of the subtree rooted by a.[7] This enables us to maintain $M_\ell(v)$ as the maximum priority of an item in $h_\ell(v)$, and to maintain $M_r(v)$ as the maximum priority of an item in $h_r(v)$. One can easily verify that this definition of $M_\ell(v)$ and $M_r(v)$ is equivalent to the previous definition.

We now describe the operations of the data structure. The query operation is performed exactly the same as the query in the data structure which supports only inserts.

Next we show how to insert a new interval $s = [x, y]$ into our data structure (the implementation of delete is similar). We first insert the endpoints of s to T_p. When we rebalance T_p we may have to perform rotations. We describe how to update the secondary trees while doing a rotation below. Next, we insert s into $S(v)$ of the appropriate node v. We do that by inserting x to $T_\ell(v)$, and inserting y to $T_r(v)$. In order to insert x to $T_\ell(v)$, we create a path in $T_\ell(v)$ which is isomorphic to the path in T_p from the left child of v to the leaf corresponding to x. Next, we traverse this path bottom-up and for every node a we update $m(a)$ to be the maximum of the maximum priority values of the children of a. In addition, we update the corresponding heap $h_\ell(\sigma(a))$ by increasing the key of a in $h_\ell(\sigma(a))$. Similarly, we update $T_r(v)$ and the related heaps.

To complete the description of the insert operation, we show how to update the secondary trees when we do a rotation in T_p. When we do a rotation around the edge $(v = p(u), u)$, we first have to perform the same rotation in all secondary trees $T_\ell(w)$ and $T_r(w)$ that contain a node isomorphic to v. Note that in each of these trees, the node in which we have to do the rotation is stored in $h_\ell(v)$ or $h_r(v)$. Therefore, we can easily scan these heaps and perform the proper rotations. In addition, we have to rebuild $h_\ell(v)$, $h_r(v)$, $h_\ell(u)$ and $h_r(u)$. Next, since the rotation changes $S(v)$ and $S(u)$, we rebuild the trees $T_\ell(v)$, $T_r(v)$, $T_\ell(u)$ and $T_r(u)$ that represent them. This is done by scanning the leaves of the subtree of T_p rooted by v, calculating for each new tree the list of endpoints which correspond to its leaves, and rebuilding the tree bottom-up. In addition, we delete every node of the old trees from its corresponding heap, and insert every node of the new trees to its corresponding heap.

[6] Note that we do not need to maintain the trees $T_\ell(v)$ and $T_r(v)$.

[7] If a is the root of $T_\ell(u)$ or $T_r(u)$ then we artificially exclude it from affecting the maximum by giving it a key of $-\infty$.

3.2 Analysis

The analysis builds upon the observation that for each $v \in T_p$, the size of $h_\ell(v)$ and $h_r(v)$ is $O(\log n)$. Therefore each update operation to the heap $h_\ell(v)$ or $h_r(v)$ takes $O(\log\log n)$ worst case time.

THEOREM 3.1. *The fully dynamic data structure described above supports queries in $O(\log n)$ time in the worst case, and insertions and deletions in $O(\log n \log\log n)$ amortized time.*

PROOF. The time bound of a query is the time for finding the maximum in each heap times the length of the search path, which equals $O(\log n)$.

Next we prove the upper bound on the time of an insertion. The proof for delete is similar. Assume we insert an interval $s = [x, y]$, and let u be the vertex of T_p such that $s \in S(u)$. During the insertion we traverse a single path of $T_\ell(u)$ and a single path of $T_r(u)$, and for each node on these paths, we perform an update to the corresponding heap. Since the size of the heaps is $O(\log n)$ these updates take $O(\log n \log\log n)$ time.

In addition, we need to consider the complexity of rebalancing T_p. We show that the time it takes to carry out a rotation at $v \in T_p$ is $O(n(v) \log\log n)$. Therefore by implementing T_p as a $BB(\alpha)$ tree it follows from Theorem 2.1 that an update operation, including the rebalancing, takes $O(\log n \log\log n)$ amortized.

Consider a rotation around the edge $(v = p(u), u)$ in T_p. Since a tree $T_\ell(w)$ or $T_r(w)$ contains an edge isomorphic to (v, u) only if it contains a descendant leaf of u and each descendant leaf of u is contained only in a single tree $T_\ell(w)$ or $T_r(w)$, then there are at most $O(n(v))$ trees $T_\ell(w)$ and $T_r(w)$ that contain an edge isomorphic to (v, u). Therefore performing the isomorphic rotations in all of these trees takes $O(n(v))$ time. The size of the old and new versions of $T_\ell(v)$, $T_r(v)$, $T_\ell(u)$, and $T_r(u)$ is $O(n(v))$, and each node in these trees requires an update operation to an heap. Therefore, the rebuilding of $T_\ell(v)$, $T_r(v)$, $T_\ell(u)$, and $T_r(u)$ takes $O(n(v) \log\log n)$ and the theorem follows. □

THEOREM 3.2. *Our data structure requires $\Theta(n \log\log n)$ space.*

PROOF. (sketch) We evaluate the sizes of $h_\ell(v)$ and $h_r(v)$ for all nodes $v \in T_p$, since these sizes dominate the size of the data structure.

Let F be the set of internal nodes v in T_p, such that $n(v) \le \log n$, and $n(parent(v)) > \log n$. The nodes in F form a "front" in T_p in the sense that every path from the root of T_p to a leaf, passes through a single node in F. Note that $|F| = \Theta(n/\log n)$. Let T_p^- be the tree remaining of T_p after we prune all the subtrees rooted by nodes of F. It is easy to show that the size of T_p^- is $O(n)$.

Consider a subtree $\tau(v)$ of T_p, which is rooted by a node $v \in F$. Note that $|\tau(v)| = \Theta(\log n)$. The sum of the sizes of $h_\ell(w)$ and $h_r(w)$ for all nodes w of a particular depth d in $\tau(v)$ is bounded by the number of endpoints stored at the leaves of $\tau(v)$, which is $O(\log n)$. Summing over $\Theta(\log\log n)$ levels in $\tau(v)$, and over $\Theta(n/\log n)$ nodes $v \in F$, we get the bound $O(n \log\log n)$. It is possible to construct an example where this bound is tight. □

We can reduce the space requirement of the data structure to be linear without hurting the time bound of the other operations with the following modification. We change the data structure so that we do not explicitly maintain the secondary tree and the heaps for nodes of large depth in T_p. Specifically we define the *front*, F, of T_p to consist of all nodes v such that $n(v) \le \log n$, and $n(parent(v)) > \log n$. We maintain secondary trees and heaps only for nodes which are ancestors of nodes in F.[8] We modify the query algorithm such that when it reaches a node v in F it linearly scans the subtree below v to check the priority of all intervals containing the query with an endpoint in that subtree. The implementation of insert and delete is similar to what we described before, with the modifications required to maintain F. This modification also improves the amortized time bound for insert to $O(\log n)$. The following theorem summarizes the main result of this section.

THEOREM 3.3. *There exists a data structure for dynamically maintaining intervals with priorities such that one can find an interval of maximum priority containing a query point in $O(\log n)$ worst-case time, insert an interval in $O(\log n)$ amortized time, and delete an interval in $O(\log n \log\log n)$ amortized time. The space requirement of the data structure is $O(n)$.*

4. NESTED INTERVALS

In this section we suggest a simpler and faster solution for the case where any two intervals in our data structure at any one time are either disjoint or one is contained within the other. The data structure is based on dynamic trees [13]. It requires $O(n)$ space, and each operation takes $O(\log n)$ time. The time bound is amortized with self-adjusting dynamic trees and worst-case with balanced dynamic trees.

We first recall that dynamic trees [13] is a representation of a forest of rooted trees with costs assigned to their edges that support the following operations, all in logarithmic 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.

cut(v) - Splits the tree containing v by removing the edge from v to its parent.

The intervals currently in the data structure define a *containment forest*, N. Each interval is a node in N and the parent of an interval I is the smallest interval containing I. To simplify the presentation we add a dummy interval $[-\infty, \infty]$, with priority $-\infty$ so that N is always a single tree. We order siblings in N by increasing left (or equivalently right) endpoint of the corresponding intervals.

Given a query point q, the set of intervals containing q is nested. It forms a path $P(q)$ in N, from the root of N to the smallest interval $I(q)$ containing q. The answer to the query is the interval of largest priority along $P(q)$. If we represent N as a dynamic tree then we can find this interval in logarithmic time (if we know $I(q)$). However such a naive representation does not obviously allow neither to search for $I(q)$ nor an efficient implementation of insertions and deletions.[9]

[8] Secondary trees are restricted to nodes isomorphic to nodes in T_p that are ancestors of nodes in F.

[9] Assume we insert a new interval I. Let I' be the smallest

We describe a data structure with two components. The first component is to facilitate the search for $I(q)$. This component allows us to locate $I(q)$ for a query point q. Once knowing $I(q)$ the second component allows us to find the interval of largest priority along the path $P(q)$ in N. This second component will in fact be a dynamic tree of a binarized version of N. Alternatively we can describe a version of the data structure with only a single tree. This tree is the dynamic tree representation of the binarized version of N with additional information so that it supports searches. The two-component version of our data structure, where we separate the search from the evaluation, is easier to describe, but the version using a single dynamic tree has better constant factor in both time and space.

We start with describing the dynamic tree component of the data structure. We define a binarized version B of N in a standard way as follows. The nodes of B are those of N. The left child of a node in B is its first child in N, or null if it has no children in N. The right child of a node in B is its next sibling in N, or null if it has no next sibling. It is easy to check that an insertion or deletion of an interval requires changing only a constant number of edges of B. Specifically, an insertion requires (at most) two edge deletions and three edge insertions; a deletion, (at most) three edge deletions and two edge insertions.

We assign costs to the edges of B as follows. The cost of an edge from a node v to its left child is the minus of the priority of the interval which corresponds to v. The cost of an edge from a node v to its right child, is ∞. It is straightforward to check that the maximum priority of an interval along a path in N from the root to a node I (not including I), equals to the minimum cost of an edge on the corresponding path in B from the root to I. We maintain B with these costs as a dynamic tree.

The search component of the data structure is a balanced search tree F on the endpoints of the intervals. Every endpoint in the search tree points to the node of the corresponding interval in the dynamic tree. Using this search we locate the node in the dynamic tree with a closest endpoint to a query point q in logarithmic time. Note that this interval may be either $I(q)$, or, a child of $I(q)$ in N.

We are now ready to describe the operations of our data structure. To perform a query with a point q, we first use F to find the interval I with the closest endpoint to q. Then, we use the dynamic tree to calculate $-mincost(I)$. If $q \notin I$, then I is a child of $I(q)$ in N, and the result is $-mincost(I)$. Otherwise, I is $I(q)$, and the result is the maximum between $-mincost(I)$ and the priority of I. To perform an insert operation, we need to update both F and the dynamic tree representing B. We update F by adding the endpoints of the new interval with a corresponding pointer to a new node which represent this new interval in the dynamic tree. We update the dynamic tree by a constant number of link and cut operations simulating the edge insertions and deletions required to update B.

5. EXTENSIONS TO HIGHER DIMENSIONS

Using dynamic segment trees [8], in a standard way, one can generalize our data structures to maintain rectangles in higher dimensions. In the plane we store the rectangles in a segment tree, T^y, according to their projections to the y-coordinate. The segment tree T^y partitions the plane into strips parallel to the x-axis each corresponding to a different node of T^y. The rectangles stored at one node v of T^y cover the y-strip corresponding to v and the search among them is a one dimensional problem. To facilitate this search we store the projections of these rectangles into the x-coordinate in a one dimensional data structures, $T^x(v)$. To answer a query we use the segment tree T^y to find $O(\log n)$ one dimensional problems (a path in T^y) containing all rectangles that contain the query point. Then we search these one dimensional problems to find the answer. We can obtain solutions in even higher dimensions using the same technique, while maintaining one additional segment tree for each additional dimension. This general approach has a penalty of $O(\log n)$ factor both in time and space for each additional dimension.

Consider now the problem restricted to a set of nested d dimensional rectangles (any pair of rectangles is either disjoint or one rectangle is contained within the other). It is not hard to prove that the dimension reduction technique via a segment tree carries the nested property into the subproblems in the lower dimension. For example, in the plane, the projections of rectangles stored in a node v of T^y to the x-coordinate, form a set of nested intervals. Therefore we can implement $T^x(v)$ as a data structure particular to nested intervals. Furthermore, there is a special nesting relation between the different subproblems. This relation enables us to tie the queries in the subproblems together, such that each additional segment tree only adds $O(\log n)$ time to the total query time. Thus, we can achieve a logarithmic query time for the d dimensional problem for any fixed constant d. This technique still has a penalty of $O(\log n)$ factor both in update time and space for every dimension.

In the rest of this section we present our ideas via an optimized data structure for the nested problem in the plane for which the amortized query time is $O(\log n)$. The amortized time bounds of the update operations that is $O(\log^2 n)$, and the space bound is $O(n \log n)$.[10] The description of the data structure in even higher dimensions will appear in the full version of this paper.

5.1 The two-dimensional data structure

The dynamic segment tree T^y is a $BB(\alpha)$ tree over the y-coordinates of the rectangles. A rectangle r whose projection to the y-axis is r_y is associated with every node $v \in T^y$ such that $range(v) \subseteq r_y$ and $range(p(v)) \not\subseteq r_y$. (Note that there are $O(\log n)$ such nodes.) We denote the set of rectangles associated with node $v \in T^y$ by $S(v)$. As we have already mentioned the projections to the x-coordinate of the rectangles in $S(v)$ is a nested set of intervals. However, a stronger nesting property in fact holds. Let the *x-interval* of a rectangle be the projection of that rectangle to the x-axis. If we look at a node v in T^y, and a node u descendant of v in T^y, then an x-interval of a rectangle in $S(v)$ must either contain or be disjoint of any x-interval in $S(u)$. We represent $S(v)$ in a one dimensional data structure $T^x(v)$.

interval in the data structure containing I. To update N we have to replace a sequence of consecutive children of I' by I, and make this sequence of children of I' children of I instead. Deleting an interval I has the inverse effect. We contract the node in N corresponding to I into its parent.

[10] We believe but have not carefully checked that our time bounds can be made worst-case rather than amortized using variants of $BB(\alpha)$ trees and biased search trees with good worst-case performance [1]. The resulting data structure would be more complicated.

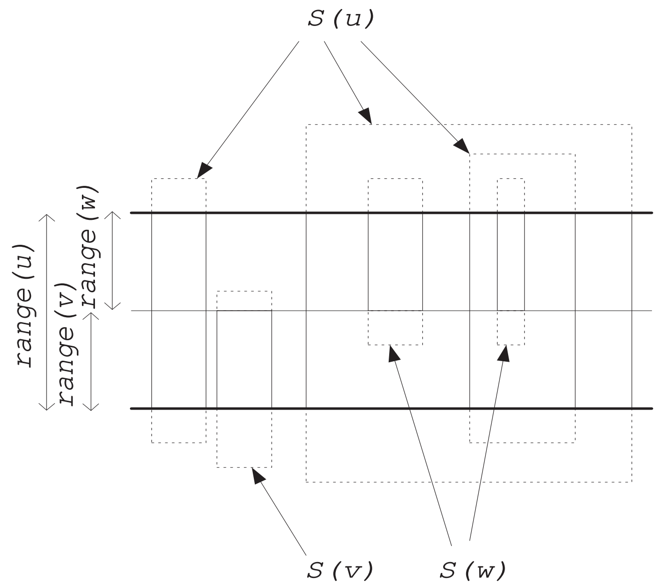

Figure 1: Geometric view of $S(u)$, $S(v)$, $S(w)$, where u, v, w are nodes in the segment tree T^y and v and w are the children of u

Furthermore, we make all the trees $T^x(v)$, $v \in T^y$ isomorphic to subtrees of one tree T^x containing all x-coordinates of corners (which we call *x-endpoints*) of our set of rectangles.

The tree T^x is maintained as a $BB(\alpha)$ tree. To define $T^x(v)$ precisely we first define $\overline{S}(v)$, for each $v \in T^y$, to be union of the sets $S(u)$ over all nodes u descendants of v in T^y. Now $T^x(v)$ is isomorphic to the subtree of T^x induced by the x-endpoints of the rectangles in $\overline{S}(v)$ and their ancestors. Note that $T^x(v)$ is not necessarily a binary tree. It may contain unary nodes and therefore it may contain nodes with the same range. In each node a of $T^x(v)$ we use the same split point as in the isomorphic node α in T^x. In addition, we identify a to be a left child or a right child according to whether α is a left child or right child, respectively. This holds even if a is a single child of its parent. [11]

For each internal node $a \in T^x(v)$ we maintain the following.

1. The largest interval in $S(a)$ if $S(a) \neq \emptyset$.

2. A splay tree $T(a)$ which represents the intervals in $S(a)$ ordered by decreasing lengths, if $S(a) \neq \emptyset$.[12] In each node s of $T(a)$ we maintain the maximum priority of a descendant of s. Using these maximum priority values it is easy to find the maximum priority interval in $S(a)$ that contains a given point q by splaying at the smallest interval containing q.

3. The number leftmost(a), which is the leftmost endpoint of an interval stored in the subtree rooted by a in $T^x(v)$ or ∞ if no such interval exists.[13] Similarly, we maintain rightmost(a), the rightmost endpoint of an interval stored in the subtree rooted by a in $T^x(v)$ or $-\infty$ if no such interval exists.

4. A pointer $\varphi(a)$ to the node α which is isomorphic to a in T^x. The node α in T^x stores a reverse pointer to a. Thus each node of T^x stores a list of pointers to all its isomorphic nodes in trees $T^x(v)$.

5. Two pointers $\ell^y(a)$ and $r^y(a)$ defined as follows. Let v_ℓ and v_r be the left and right children of v in T^y, respectively. If $T^x(v_\ell)$ has a node a' isomorphic to $\varphi(a)$ then $\ell^y(a)$ points to a'. Otherwise it is null. Similarly, if $T^x(v_r)$ has a node a' isomorphic to $\varphi(a)$ then $r^y(a)$ points to a'. Otherwise it is null.

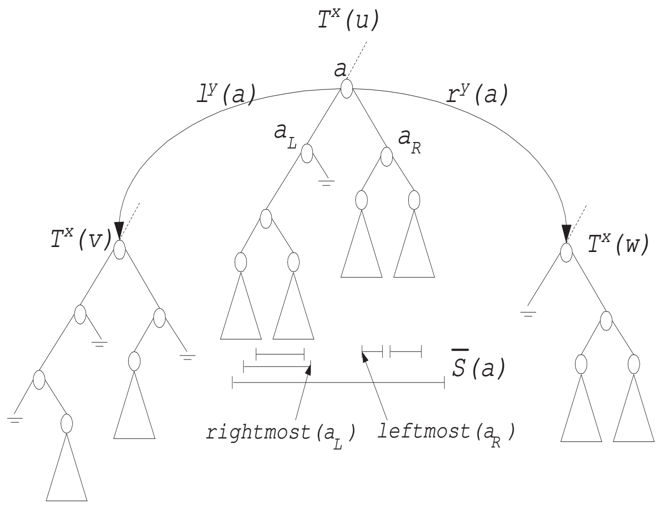

Figure 2: The structure of $T^x(u)$, $T^x(v)$, $T^x(w)$, where u, v, w are nodes in the segment tree T^y and v and w are the children of u. The segments of $T^x(u)$ are illustrated under the tree.

We perform a query for a point $q = (q_x, q_y)$ as follows. We traverse the search path of q_y in T^y. Let $v_0, v_1, \ldots, v_k$ be the nodes on the search path of q_y in T^y. (Note that v_0 is the root of the interval tree T^y.) For each node v_i, $0 \leq i \leq k$, we query $T^x(v_i)$ for the x-interval with the highest priority which contains q_x. We conclude by finding the maximum of all maximum priority x-intervals, over all nodes v_i. When we query $T^x(v_i)$ we generally will not traverse the whole search path of q_x in $T^x(v_i)$. Instead we will traverse just part of it which is guaranteed to contain the relevant intervals. For every node a in $T^x(v_i)$ that we traverse, if the largest

[11] It in fact suffices to make each $T^x(v)$ contain a leaf for each endpoint of an interval in $\overline{S}(v)$, and to relax the requirement of all trees being isomorphic to a single tree T^x. This change reduces the space by a logarithmic factor. We discuss it further in Section 5.4.

[12] A splay tree [12] is a binary search tree where access to node x (which is also called splaying at x) moves x to the root of the tree, by using rotations. Splay trees allow biased access times for different nodes. Specifically, for any positive weight function over the nodes a splay at node x takes $O(\log(\frac{s(root)}{s(x)}))$ amortized time where $s(y)$ is the sum of the weights of descendants of y.

[13] Note that we can define leftmost(a) recursively as follows. If a is a leaf, leftmost$(a) = \infty$. Otherwise, leftmost$(a) = \min(\text{leftmost}(a_\ell), \text{leftmost}(a_r), Left(S(a)), \infty)$, where a_ℓ and a_r are the left and right children of x, respectively, and $Left(S(a))$ is the set of left endpoints of intervals in $S(a)$.

interval in a contains q_x then we search $T(a)$ for the interval with largest priority in $S(a)$ that contains q_x. The answer to the query inside $T^x(v_i)$ is the maximum among the answers from the splay trees which we traversed.

Now we define precisely which nodes on the search path of q_x in $T^x(v_i)$ we traverse. We start the traversal in $T^x(v_0)$ from the root. In general the starting point of the traversal in $T^x(v_{i+1})$ depends on the node in which the traversal in $T^x(v_i)$ terminated. We terminate the traversal in each tree $T^x(v_i)$ when we reach a node b^i such that the following conditions are satisfied.

1. $q_x > \text{rightmost}(b^i_\ell)$, if b^i has a left child b^i_ℓ.
2. $q_x < \text{leftmost}(b^i_r)$, if b^i has a right child b^i_r.

In order to find where to start the search in $T^x(v_{i+1})$, we use the appropriate one of the following cases. If v_{i+1} is a left child of v_i, and $\ell^y(b^i)$ is not null, then we start the search in $\ell^y(b^i)$. If v_{i+1} is a right child of v_i, and $r^y(b^i)$ is not null, then we start the search in $r^y(b^i)$. Otherwise, we stop the search; We do not search any node in $T^x(v_j)$ for $i+1 \le j \le k$.

The proof that our query algorithm is correct has two parts. First one has to show that the path which we traverse in each tree $T^x(v_i)$, is a subpath of the search path of q_x in T^x. From this follows that all the rectangles whose priorities are considered while calculating the answer indeed contain the query point. In the second part of the proof we show that we consider all rectangles containing the query even though we do not traverse the complete search path of q_x in each $T^x(v_i)$. Let b^i be the node in which we stopped the traversal in $T^x(v_i)$. Let c be a descendant of b^i on the search path of q_x in $T^x(v_i)$. Then it is easy to see that our stopping criteria guarantees that there is no interval in $S(c)$ that contains q_x. The next two lemma complete the proof by showing that we do not loose any rectangles by not starting the search from the root of $T^x(v_i)$.

LEMMA 5.1. *Let v and u be two nodes in T^y such that u is a proper descendant of v. Let s be an x-interval in $S(v)$ and let s' be an x-interval in $S(u)$. Then, either $s' \subseteq s$, or s' is disjoint from s.*

PROOF. Let r be the rectangle corresponding to s, and let r' be the rectangle corresponding to s' (i.e, $r_x = s, r'_x = s'$). By definition of the segment tree, T^y, $r_y \supseteq range(v)$ and $r'_y \not\supseteq range(v)$. Therefore, r' cannot contain r. If $r' \subseteq r$, then it follows that $r'_x \subseteq r_x$. If r' is disjoint from r, then r'_x is disjoint from r_x, since r'_y and r_y overlap in $range(u)$. $\square$

LEMMA 5.2. *Suppose the traversal in $T^x(v_i)$ starts at node a^i for some $i > 0$. Let c be a proper ancestor of a^i. Then, there is no interval in $S(c)$ that contains q_x.*

PROOF. Assume that there is an interval s in $S(c)$ that contains q_x. Let c' be the node in $T^x(v_{i-1})$ which is isomorphic to c (such a node must exist by the definition of $T^x(v_{i-1})$). Note that by the definition of $T^x(v_i)$, $s_\ell \le split(c) = split(c') \le s_r$. By Lemma 5.1, s cannot contain an x-endpoint of any rectangle in $S(v_{i-1})$. Therefore, if c' has a left child c'_ℓ, then rightmost(c'_ℓ) which is an x-endpoint of rectangle in $S(v_{i-1})$ is not contained in s and therefore rightmost$(c'_\ell) < s_l$. Similarly, if c' has a right child c'_r, then leftmost$(c'_r) > s_r$. Therefore, by the definition of the query algorithm, the traversal in $T^x(v_{i-1})$ must have ended in c' or in some ancestor of c'. It follows that the traversal in $T^x(v_i)$ must have started in c or in some ancestor of c, and we have a contradiction. $\square$

5.2 Update operations

We now describe how to perform an insert operation. (The implementation of delete is similar.) Let r be the new rectangle. To perform the insert we follow the following steps.

(1) We insert the y-endpoints of r into T^y. While inserting, we may need to perform a rotation in T^y. To perform a rotation around the edge (u, v) in T^y we rebuild $T^x(u)$, $T^x(v)$ and the $T^x()$ structures of the children of u and v. We also update the pointers $\ell^y()$ and $r^y()$ in $T^x(p(v))$.

(2) We insert the x-endpoints of r into T^x. This insertion may trigger a rotation in T^x. To perform a rotation in T^x around an edge (α, β) we need to perform the same rotation in each tree $T^x(v)$ which has a vertex isomorphic to α. We locate these trees using the reverse φ pointers in α. When we perform a rotation in one such tree, say $T^x(v)$, we update the secondary splay trees in the nodes of $T^x(v)$ involved in the rotation by splitting and concatenating them.

(3) Let V be the set of all nodes storing r_y in T^y. We call these nodes the *allocation nodes* of r_y in T^y. Let $\overline{V}$ be the nodes in V and their ancestors in T^y. Note that by the definition of a segment tree the nodes in V are hanging off two paths in T^y and the nodes in $\overline{V} \setminus V$ are exactly the nodes on these two paths. To complete the insert operation we need to insert the x-endpoints of r into all trees $T^x(v)$ where $v \in \overline{V}$. We also need to insert the interval r_x into its allocation node in each tree $T^x(v)$ where $v \in V$.

We insert an x-endpoint, say x, of r into all trees $T^x(v)$ where $v \in \overline{V}$ bottom-up. Assume we have already inserted x into $T^x(v_\ell)$ and into $T^x(v_r)$ (if we had to) where v_ℓ and v_r are the children of v in T^y. We insert x into $T^x(v)$ as follows. Let p be the path in T^x from the root to x. Let p' be the longest path in $T^x(v)$ which is isomorphic to a prefix of p. We extend p' such that the extended path is isomorphic to p. For each new node a that we create, we set $\ell^y(a)$ and $r^y(a)$ so that they point to the corresponding nodes in $T^x(v_\ell)$ and $T^x(v_r)$. We also set leftmost(a) to ∞, rightmost(a) to $-\infty$, and $\varphi(a)$ to the node in T^x which is isomorphic to a.

We insert the interval r_x into an allocation node $v \in V$ as follows. We insert r_x into the splay tree $T(a)$ of the appropriate node a in $T^x(v)$. For each node $b \in T^x(v)$ which is an ancestor of a, we update leftmost(b) and rightmost(b). In addition if the largest interval in b contains r_x we splay $T(b)$ at the node which corresponds to the smallest interval in $S(b)$ that contains r_x. Note that all allocation nodes a of r_x in $T^x(v)$ for every $v \in V$ are isomorphic to the same node in T^x. We denote that node by α.

Similarly, for every $v \in \overline{V} \setminus V$ and every node $b \in T^x(v)$ which is either the node isomorphic to α in $T^x(v)$ or an ancestor of that node, if the largest interval in b contains r_x, we splay $T(b)$ at the node which corresponds to the smallest interval in $S(b)$ that contains r_x.

5.3 Analysis

First, we show that the query takes $O(\log n)$ amortized time. We do that by choosing an appropriate weight function for the splay trees, such that the sum of costs of splays performed by the query, is a telescopic sum, which equals

$O(\log n)$.

Let s be an interval in $T(a)$, where $a \in T^x(v)$ and $v \in T^y$. Let s^+ be the interval preceding s in $T(a)$ (i.e, s^+ is the largest interval in $T(a)$ s.t. $s^+ \subseteq s$, or $s^+ = \emptyset$ if there is no such interval). We define $gap(s)$ to be set of rectangles $r \in S(w)$, where $r_x \subseteq (s \setminus s^+)$, and w is a descendant of v in T^y. We define the weight of $s \in T(a)$, $w(s)$, to be $1 + |gap(s)|$. We also denote by $w(a)$ the sum of weights of items in $T(a)$.

LEMMA 5.3. *Suppose that we search for a point q. Let a be a node in $T^x(v)$ where $v \in T^y$, such that $T(a)$ was splayed by the query algorithm. Let b be a node in $T^x(u)$ where u is a descendant of v or v itself, such that $T(b)$ is the next splay tree that was splayed by the query algorithm. Let s be the interval which was splayed by the search algorithm in $T(a)$. Then, it follows that $w(s) \geq w(b)$*

PROOF. Let s_b be the largest interval in $T(b)$. By the query algorithm, $q \in s \setminus s^+$, and $q \in s_b$. We claim that $s_b \subseteq s \setminus s^+$. The proof then follows from the definition of $gap()$. To establish the claim, note that q is contained in both s and s_b and therefore either $s \subseteq s_b$ or $s_b \subseteq s$. In case $u = v$, b must be a descendant of a, and therefore s_b cannot contain s since $split(a) \in s$ but $split(a) \notin s_b$. In case u is a descendant of v, it follows from lemma 5.1 that s_b cannot contain s. We get that $s_b \subseteq s$. To finish the proof, note that s_b is disjoint from s^+ since $q \in s_b$ but $q \notin s^+$. □

LEMMA 5.4. *Let v, u, a and b be as defined in lemma 5.3. Then, the splay in $T(a)$ takes $1 + 3(\log(w(a)) - \log(w(b)))$ amortized time.*

PROOF. By the access lemma of Sleator and Tarjan [12], the amortized time for splaying s in $T(a)$ is $1+3(\log(w(a)) - \log(w(s)))$. By lemma 5.3, this value is bigger than $1 + 3(\log(w(a)) - \log(w(b)))$. □

From Lemma 5.4 it is easy to see that the search times over all $T^x(v_i)$ is a telescopic sum with $O(\log n)$ terms that sums to $O(\log n)$.

Next, we prove the space bound for our data structure.

LEMMA 5.5. *For every node $w \in T^y$ the tree $T^x(w)$ takes $O(n(w) \log n)$ space.*

PROOF. The secondary structures of $T^x(w)$ take linear space in the size of the primary tree of $T^x(w)$. Therefore, it suffices to estimate the size of this tree. Note that $|\overline{S}(w)|$ is bounded by $n(p(w)) = O(n(w))$. Therefore, the number of leaves in $T^x(w)$ is $O(n(w))$. If $T^x(w)$ was a binary tree then we could conclude that $|T^x(w)| = O(n(w))$. However, $T^x(w)$ is not necessarily binary and may contain unary nodes. Nevertheless $T^x(w)$ is isomorphic to a subtree of T^x and the depth of T^x is $O(\log n)$, so it follows that $|T^x(w)| = O(n(w) \log n)$. □

THEOREM 5.1. *The data structure takes $O(n \log^2 n)$ space.*

PROOF. The space of the data structure is dominated by the total space of $T^x(v)$ for every node $v \in T^y$. The sum of $n(w)$, over all nodes w at a particular depth d in T^y, is $O(n)$. Therefore, in every level of T^y, the total space of $T^x(w)$ for all nodes w in that level is $O(n \log n)$. The theorem now follows since the depth of T^y is logarithmic. □

Finally, we prove that an insert operation takes $O(\log^2 n)$ amortized time. Let r be the new rectangle that we insert. Our proof has two parts. First, we need to show that inserting the corners of r into T^x and T^y take $O(\log^2 n)$ time. We prove this by showing that the time for a rotation both in T^x and T^y is bounded by the size of the subtree rooted at the point of rotation times $\log n$. It then follows from Theorem 2.1, that the amortized time to do an insertion including the time to handle rotations is $O(\log^2 n)$. We omit the details of this part of the proof for this extended abstract. The second part is to show that inserting r_x into the secondary trees $T^x v$ for every $v \in \overline{V}$ takes $O(\log^2 n)$ amortized time. We do this in the following two lemma.

LEMMA 5.6. *Let u be a node in T^y, let b be a node in $T^x(u)$, and let s be a node in $T(b)$, such that $T(b)$ was splayed in s by the insert operation. The set of splayed nodes s, equals to the set of nodes t for which $r_x \in gap(t)$. The same statement holds for a delete operation, where r is the deleted interval.*

PROOF. The lemma is easily derived from the definitions of $gap()$ and of the insert operation. We omit the details. □

LEMMA 5.7. *Inserting r_x into $T^x(v)$ for every $v \in \overline{V}$ takes $O(\log^2 n)$ amortized time.*

PROOF. We show that for each node $v \in \overline{V}$, the insert operation takes $O(\log n)$ amortized time. The lemma then follows from the fact that $|\overline{V}| = O(\log n)$.

First, we note that the cost of inserting r_x into a splay tree $T(a)$ for some node $a \in T^x(v)$ is $O(\log n)$ by the update lemma of Sleator and Tarjan [12]. Extending a tree $T^x(v)$, $v \in \overline{V}$ so that it contains the leaves corresponding to the endpoints of r_x takes $O(\log n)$ time. Last we have to consider the weight changes in nodes of splay trees $T(a)$ associated with nodes a along the affected paths in $T^x(v)$. From Lemma 5.6, we get that nodes corresponding to x-intervals which change their weights as a result of inserting r_x, are at the root of their splay tree. Increasing the weight of the root of $T(a)$ by 1 takes $O(\log(1 + 1/w(a)))$ amortized time by the update lemma of Sleator and Tarjan. Therefore, the total cost of these increments of the weights is $O(\log n)$. □

5.4 Saving space

We can improve the space requirement of our data structure to $O(n \log n)$. We achieve that by maintaining the trees $T^x(v)$ as independent interval trees, giving away the requirement that they all be isomorphic to T^x. We maintain each $T^x(v)$ as a balanced binary tree over the x-endpoints of rectangles in $\overline{S}(v)$. As before, $T^x(v)$ is an interval tree which represents the x-intervals of rectangles in $S(v)$. Each node a of $T^x(v)$ has a splay tree representing the intervals in $S(a)$ ordered by decreasing lengths. We do not need to maintain the tree T^x. In this representation the size of each $T^x(v)$ is $O(n(v))$ (rather than $O(n(v) \log n)$) so an argument similar to the one of Lemma 5.1 shows that the total size of the data structure is $O(n \log n)$.

To keep the query time logarithmic we still have to be able to maintain pointers between nodes in $T^x(v)$ and nodes in $T^x(v_\ell)$ and $T^x(v_r)$ where v_ℓ and v_r are the children of v in T^y. We define these pointers as follows. Consider a maximal sequence A of consecutive leaves in $T^x(v)$ that are all x-endpoint of rectangles in $\overline{S}(v) \backslash S(v)$. Let $left(A)$ be the endpoint immediately preceding A in $T^x(v)$ and let $right(A)$ be the endpoint immediately following A in $T^x(v)$. Let $lca(A)$ be the lowest common ancestor of $left(A)$ and $right(A)$. Note that any node a of $T^x(v)$ could be $lca(A)$ for only a single maximal sequence A in which case it stores $left(A)$ and $right(A)$.

Consider a query for a point $q = (q_x, q_y)$. When we reach a node $a \in T^x(v)$ such that $a = lca(A)$ for some maximal sequence of leaves A as defined above, and $q_x \in [left(A), right(A)]$ we are certain that there are no more relevant rectangles in $T^x(v)$ below a. Therefore at every node a such that $a = lca(A)$ for some A we store pointers $\ell^y(a)$ and $r^y(a)$ to nodes of $T^x(v_\ell)$ and $T^x(v_r)$, respectively. Next we define to which node in $T^x(v_\ell)$, $\ell^y(a)$ points to. The node in $T^x(v_r)$ to which $r^y(a)$ points is defined similarly.

Let A' be the set of (consecutive) leaves in $T^x(v_\ell)$ that store endpoints of A. Let a' be the lowest common ancestor of all leaves in A'. Assuming that $q_x \in [left(A), right(A)]$ then it suffices to start the search in $T^x(v_\ell)$ from a' so we define $\ell^y(a)$ to point to a'. We also maintain a reverse pointer from a' to a. It is easy to check that the correspondence defined by $\ell^y()$ between nodes of $T^x(v)$ and nodes of $T^x(v_\ell)$ is one-to-one. So each node of $T^x(v_\ell)$ has at most one reverse pointer. These reverse pointers are needed to update $\ell^y(a)$ when a rotation in $T^x(v_\ell)$ changes the lowest common ancestor of A'.

To get a logarithmic query time we have to guarantee that the size of the subtree of $T^x(v_\ell)$ rooted at a' is proportional to the size of the subtree of $T^x(v)$ rooted at a. However, since we balance $T^x(v_\ell)$ independently of $T^x(v)$ our definition of a' does not guarantee that. (At the extreme a' could even be the root of $T^x(v_\ell)$.) We overcome this difficulty by maintaining two additional pointers together with $\ell^y(a)$ into particular nodes in $T^x(v_\ell)$. These pointers would allow us to start the search in $T^x(v_\ell)$ at a node below a' whose subtree is not too large. We define these two pointers as follows.

Let A'_ℓ be the subset of A' which are descendants of the left child of a', and let a'_ℓ be the lowest common ancestor of all nodes in A'_ℓ. Note that the size of the subtree rooted at a'_ℓ is proportional to $|A'_l| \le |A|$ that is smaller than the size of the subtree rooted at a. Similarly, we define A'_r, and a'_r. The additional pointers that we keep in a are to a'_ℓ and a'_r. To facilitate the updates we also maintain reverse pointers to these two additional pointers that we keep with $\ell^y(a)$.

When the search in $T^x(v)$ terminates in a we start searching $T^x(v_\ell)$ in a'. After searching a' if the search is to continue in the left child of a' we in fact continue in a'_ℓ and skip all nodes between a' and a'_ℓ (if there are any). Similarly we continue in a'_r if the search is to continue in the right child of a'. The correctness of this query algorithm follows from the fact that a'_ℓ is on the search path of the query in $T^x(v_\ell)$, and all nodes between a' and a'_ℓ that we skipped cannot store any x-interval containing q as otherwise we get a violation to the nested property.

When we insert a rectangle we insert its x-interval into the trees $T^x(v)$ for every node v in T^y that is an allocation node of the y-interval of the rectangle. While performing the insertions into a tree $T^x(v)$ if a rotation occurs then we may have to update a constant number of pointers connecting $T^x(v)$ to $T^x(v_\ell)$, $T^x(v_r)$, and $T^x(p(v))$, paying at most logarithmic time per update. We also have to insert the x-endpoints of the rectangle into the trees $T^x(v)$ where v is an ancestor of an allocation node of the y-interval of the rectangle. These insertions require similar modifications.

6. FURTHER RESEARCH

There are several intriguing questions related to our results. Here are we mention two major ones. For general intervals our one dimensional data structure has a superlogarithmic time bound for delete, specifically the time bound for delete is $O(\log n \log\log n)$. Is there a data structure for general intervals for which all operations take $O(\log n)$ time. (Such a data structure would be optimal in the comparison model by a trivial reduction from sorting.) Another question is related to our technique to handle nested rectangles in two and higher dimensions. Is it possible to find a simpler data structure? Our two dimensional data structure for nested rectangles is much more complicated than the one dimensional data structure for nested intervals and it is not certain that all complications are really necessary.

7. REFERENCES

[1] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

[2] M. Buddhikot, S Suri, and M Waldvogel. Space decomposition techniques for fast Layer-4 switching. In *Protocols for High Speed Networks IV*, pages 25–41, 1999.

[3] H. Edelsbrunner. A new approach to rectangle intersection, part I. *Int. J. Computer Mathematics*, 13:209–219, 1983.

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

[5] P. Gupta and N. McKeown. Dynamic algorithms with worst-case performance for packet classification. In *IFIP Networking*, pages 528–539, 2000.

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

[7] K. Mehlhorn. *Data structures and algorithms, Volume 1: Sorting and Searching*. Springer-Verlag, Berlin, 1984.

[8] K. Mehlhorn. *Data structures and algorithms, Volume 3: Multi-dimensional Searching and Computational Geometry*. Springer-Verlag, Berlin, 1984.

[9] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973.

[10] S. Sahni and K. Kim. O(log n) dynamic packet routing. *IEEE Symposium on Computers and Communications*, pages 443–448, 2002.

[11] S. Sahni, K. Kim, and H. Lu. Data structures for one-dimensional packet classification using most specific rule matching. *ISPAN*, pages 3–14, 2002.

[12] D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.

[13] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–391, 1983.

[14] R. E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *JCSS*, 18(2):110–127, 1979.

[15] D. E. Willard and G. S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985.