Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports
Department of Computer Science

1993

Biased Finger Trees and Three-Dimensional Layers of Maxima

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Michael T. Goodrich

Kumar Ramaiyer

Report Number:

93-035

Atallah, Mikhail J.; Goodrich, Michael T.; and Ramaiyer, Kumar, "Biased Finger Trees and Three-Dimensional Layers of Maxima" (1993). *Department of Computer Science Technical Reports.* Paper 1052.
https://docs.lib.purdue.edu/cstech/1052

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

BIASED FINGER TREES AND THREE-DIMENSIONAL LAYERS OF MAXIMA

Mikhail J. Atallah
Michael T. Goodrich
Kumar Ramaiyer

CSD TR-93-035
June 1993
(Revised 3/94)
(Revised 4/94)

Biased Finger Trees and Three-Dimensional Layers of Maxima

Mikhail J. Atallah* — Dept. of Computer Sciences, Purdue University, W. Lafayette, IN 47907-1398, mja@cs.purdue.edu

Michael T. Goodrich† — Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218-2694, goodrich@cs.jhu.edu

Kumar Ramaiyer‡ — Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218-2694, kumar@cs.jhu.edu

Abstract

We present a method for maintaining biased search trees so as to support fast finger updates (i.e., updates in which one is given a pointer to the part of the tree being changed). We illustrate the power of such biased finger trees by showing how they can be used to derive an optimal $O(n \log n)$ algorithm for the 3-dimensional layers-of-maxima problem and also obtain an improved method for dynamic point location.

1 Introduction

Binary search trees are one of the most useful data structures, and are ubiquitous throughout the design and analysis of efficient algorithms. In some cases they serve as a stand-alone structure (e.g., implementing a dictionary or a heap), while in many cases they are used in tandem with other structures, either as primary or secondary structures (or both, as in the range tree [35]). In many dynamic computational geometry algorithms they may even be found as tertiary structures.

1.1 Background and Motivation

When a binary search tree T is maintained dynamically as a primary structure it is appropriate to count, as a part of the update time, the time to perform a top-down search for the node(s) in T being changed. This may not be appropriate when T is used in tandem with other structures, however, for one may be given, as part of the input to an update operation, pointers, or "fingers" [19, 24, 21], directly into the part of T being changed. This may, in fact, have been a prime motivating factor behind the method of Huddleston and Mehlhorn [21] for designing a dynamic search tree that has an $\bar{O}(1)$ update time performance for insertions and deletions when the search time is not counted, where we use "$\bar{O}(\cdot)$ time" to refer to a worst-case time bound that is amortized over a sequence of updates.

Another important variant concerns the case when each item i in the search tree is given a *weight*, w_i. This weight may represent an access probability, as in an optimal binary search tree structure [4, 23]. Or it may represent the size of some auxiliary structure associated with item i, as in a link-cut structure [41] (which itself has many applications [12, 17, 18]) or in a point location structure built using the trapezoid method [9, 34, 39]. In cases with weighted items such as these

*This research supported by the NSF under Grant CCR-9202807.

†This research supported by the NSF and DARPA under Grant CCR-8908092, by the NSF under Grants CCR-9003299, CDA-9015667, and IRI-9116843.

‡This research supported by the NSF and DARPA under Grant CCR-8908092 and by the NSF under Grant IRI-9116843.

one desires a search tree satisfying a *bias* property that the depth of each item i in the tree be inversely proportional to w_i. Bent, Sleator and Tarjan [6] give a method for maintaining such a structure subject to update operations, such as insertions, deletions, joins, and splits, as well as predecessor query operations. Most of their update and query operations take $O(\log W/w_i)$ time (in some cases as an amortized bound), with the rest taking slightly more time, where W is the sum of all weights in the tree.

1.2 Our Results

In this paper we examine a framework for achieving fast finger-tree updates in a biased search tree, and we refer to the resulting structure as a *biased finger tree.* We know of no previous work for a structure such as this. We show that insertions and deletions in a biased finger tree can be implemented in $\bar{O}(\log w_i)$ time, not counting search time, while still maintaining the property that each item i is at depth $O(\log W/w_i)$ in the tree. Moreover, we show that, while split operations will take $\bar{O}(\log W/w_i)$ time (which is unavoidable), we can implement join operations in $\bar{O}(1)$ time.

Our structure is topologically equivalent to that given by Bent, Sleator, and Tarjan [6]. In fact, if each item i has weight $w_i = 1$, then our structure is topologically equivalent to a red-black tree [14, 20, 42]. It is our update methods and amortized analysis that are different, and this is what allows us to achieve running times that are significant improvements over those obtained by Bent, Sleator, and Tarjan, even if one ignores the search times in their update procedures. Moreover, we provide an alternative proof that red-black trees support constant-time amortized finger updates (which is a fact known to folklore).

We show the utility of the biased finger tree structure by giving an optimal $O(n \log n)$-time space-sweeping algorithm for well-known 3-dimensional layers-of-maxima problem [3, 8, 15, 26] and also give improved methods for dynamic point location in convex subdivision [34, 9]. The space-sweeping algorithm makes use of restricted dynamic point location in rectilinear subdivision, which we deal as a special case of point location in convex subdivision.

In the sections that follow we outline our method for maintaining biased finger trees and we describe how they may be applied for dynamic point location as well as how they lead to an optimal method for the 3-d layers-of-maxima problem.

2 Biased Finger Trees

Suppose we are given a totally ordered universe U of weighted items, and we wish to represent a collection of disjoint subsets of U in binary search trees subject to the "standard" tree search queries, as well as item insertion and deletion in a tree, and join and split operations on trees (consistent with the total order). Aho, Hopcroft, and Ullman [4] refer to these as the *concatenable queue* operations.

In this section we describe a new data structure that efficiently supports all of the these operations. So as to concentrate on the changes required by an update operation, we will assume that each update operation comes with a pointer to the node(s) in the tree(s) where this update is to begin. Formally, we define our update operations as follows:

Insert(i, w_i, p_{i^-}, T) : Insert item i with weight w_i into T, where p_{i^-} is a pointer to the predecessor, i^-, of i in T (if i has no predecessor in T, then we let this point to i's successor).

Delete(i, p_i, T) : Remove item i from the tree T, given a pointer p_i to the node storing i.

Split(i, T) : Partition T into three trees: T_l, which contains all items in T less than i, the item i itself, and T_r, which contains all items in T greater than i.

Join(T_x, T_y) : Construct a single tree from T_x and T_y, where all the items in T_x are smaller than the items in T_y.

Change-weight(i, w'_i, T, p_i) : Change the weight of the item i in T from w_i to $w_{i'}$, given the pointer p_i to the node storing the item i.

Slice(i, T, x, i_1, i_2) : Slice the item i in T into two items i_1 and i_2 such that $i^- \leq i_1 \leq i_2 \leq i^+$, and the weight of item i_1 is $x * w_i$ and the weight of item i_2 is $(1-x) * w_i$, where $x \in \mathbb{R}$, and $0.0 < x < 1.0$, and i^- and i^+ are predecessor and successor items of i in T respectively.

Fuse(i_1, i_2, i, T) : Fuse the items i_1 and i_2 in T into a single item i of weight $w_{i_1} + w_{i_2}$ such that $i_1 \leq i \leq i_2$ in T.

As mentioned above, our structure is topologically similar to the biased search tree[1] of Bent, Sleator and Tarjan [6]. Our methods for updating and analyzing these structures are significantly different, however, and achieve run times better than those of Bent *et al.* in most cases (see Table 1).

We assume that items are stored in the leaves, and each internal node stores two values, *left* and *right*, which are pointers to the largest item in the left subtree and the smallest value in the right subtree, respectively. In addition, the root maintains pointers to the minimum and maximum leaf items. Every node x of the tree stores a rank $r(x)$ that satisfies the natural extensions of red-black tree rank [42] to weighted sets [6]:

1. If x is a leaf, then $r(x) = \lfloor \log w_i \rfloor$, where i is the item x stores.
2. If node x has parent y, then $r(x) \leq r(y)$; if x is a leaf, then $r(x) \leq r(y) - 1$. Node x is *major* if $r(x) = r(y) - 1$ and *minor* if $r(x) < r(y) - 1$.
3. If node x has grandparent y, then $r(x) \leq r(y) - 1$.

In addition to the above rank conditions, we also require a node be minor if and only if its sibling or a child of its sibling is a major leaf [6]. We refer to this as the *bias* property.

We now prove a lemma which relates the depths of the nodes with their ranks. We let $d_T(v)$ denote the depth a node v in the tree T.

Lemma 2.1: *If u is an ancestor of a node v in T, then $d_T(u) - d_T(v) = O(r(v) - r(u))$.*

Proof: The ranks along any leaf-to-root path are non-decreasing and also the rank of a node is strictly less than its grandparent. Moreover, the rank of a minor node is strictly less than the rank of its parent. But the depth along any leaf-to-root path strictly decreases and the depths of adjacent nodes differ exactly by one. So, we have $d_T(u) - d_T(v) = O(r(v) - r(u))$. ■

In the remainder of this section we provide algorithms for various update operations on biased finger trees and also analyze their amortized complexities.

2.1 Rebalancing a Biased Finger Tree

We begin our discussion by analyzing the time needed to rebalance a biased tree after an update has occurred. We use the *banker's view* of amortization [43] to analyze the rebalancing and update operations. In each node x of a biased finger tree we maintain[2] a number, $C(x)$, of "credits", with $0 \leq C(x) \leq c$, for some constant $c > 0$. We assign three types of credits to nodes of a biased finger tree as follows:

[1] Bent, Sleator and Tarjan actually introduce two kinds of biased search trees; our biased finger trees are structurally equivalent to the ones they call *locally biased*.

[2] This credit notion is only used for analysis purposes. No actual credits are stored anywhere.

Update Operation	*Previous Biased Trees [6]*	*Biased Finger Trees*
Search(i,T)	$O(\log W/w_i)$	$O(\log W/w_i)$
Insert(i, w_i, p_{i^-}, T)	$\bar{O}\left(\log \frac{W'}{\min(w_{i^-}+w_{i^+}, w_i)}\right)$	$\bar{O}(\lvert\log \frac{w_i}{w_{i^-}}\rvert + 1)$
Delete(i, T, p_i)	$\bar{O}(\log W/w_i)$	$\bar{O}(\log w_i)$
Split(i, T)	$\bar{O}(\log W/w_i)$	$\bar{O}(\log W/w_i)$
Join(T_x, T_y)	$\bar{O}(\lvert\log W_x/W_y\rvert)$	$\bar{O}(1)$
Change_Weight$(i, w_{i'}, T, p_i)$	$\bar{O}(\log(\frac{\max(W,W')}{\min(w_i,w_{i'})}))$	$\bar{O}(\lvert\log w_i/w_{i'}\rvert)$
Slice(i, T, x, i_1, i_2)	—	$\bar{O}(\log \frac{w_i}{min(w_{i_1}, w_{i_2})})$
Fuse(i_1, i_2, i, T)	—	$\bar{O}(min(\log w_{i_1}, \log w_{i_2}))$

Table 1: **Summary of Time Bounds for Biased Tree Operations.** W and W' are the sum of the weights of all the items before and after the update operation respectively, W_x (W_y) denotes the sum of weights in T_x (T_y), and i^- (i^+) denotes the predecessor (successor) of i. The complexities for update operations only count the time to perform the update, and do not include search times (even for the previous biased trees).

Twin-node credit : A node u is assigned one *twin-node* credit, if u and its sibling have the same rank. This type of credit was suggested by Kosaraju [25]. We use these twin-node credits to amortize the cost of rebalancing operations.

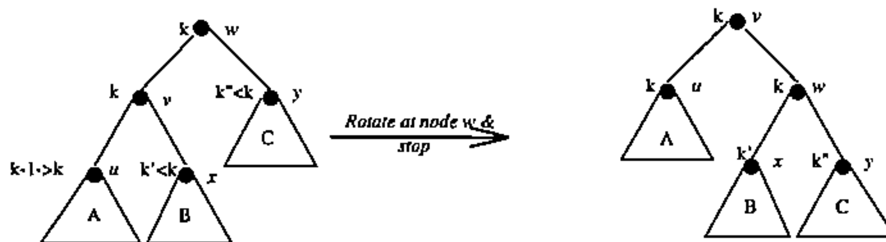
Spine-node credit : A node u is assigned two *spine-node* credits, if u occurs along the leftmost or the rightmost root-to-leaf path of T_x. We use these spine-node credits to amortize the cost of join operations.

Minor-node Credit : A node u with parent v is assigned $\max\{r(v) - r(u) - 1, 0\}$ *minor-node* credits. We note that if u is a major node, then it gets zero credits, and if u is a minor node, then it gets $r(v) - r(u) - 1$ credits. Bent, Sleator and Tarjan [6] use a similar type of credit scheme to analyze their implementation of the split and join operations.

We show that each operation starts with certain number of credits equal to our claimed amortized time and using those credits, it completes the update and rebalancing and also maintains the above credit invariant.

After an update operation, we do promote or demote operations on the ranks of some of the nodes of the biased finger tree, which increase or decrease the rank of the nodes, respectively. These operations locally preserve the rank properties, but may cause violation of the rank property on other nodes, which may require further promotions or demotions or may even require rebalancing to globally preserve the rank properties. We show that the total complexity of promotion, demotion and rebalancing operations due to a single promote or demote operation is $\bar{O}(1)$. The structure of our case analysis follows closely that of Tarjan [42].

Consider a promote operation on the rank of a node u with parent v so that $r(u)$ becomes equal to $r(v) = k$ (see Fig. 1). Let w be the parent of v and let y be the sibling of v. We may have a violation of the rank property, if $r(v) = r(w)$. We have the following cases based on the ranks of the nodes u, v, y and w:



1. Case 2 of promote. Here rotation terminates rebalancing.

2. Case 3 of promote. Here rotation is not possible. Continue the rebalancing after increasing the rank of node w i.e., recurse with u=w.

Figure 1: Promote operation on the rank of the node u.

Case 1. Suppose $r(v) \neq r(w)$. This completes the promote operation, as it does not cause any violation of the rank property.

Case 2. Suppose $r(u) = r(v) = r(w) = k$ and $r(v) \neq r(y)$. In this case we have a violation of the rank property. But since $r(y) \neq r(v)$, we can do a rotation at w to preserve the property (see Fig. 1.1). This completes the promote operation. We require 3 new credits to perform the rotation and also to place one twin-node credit each in u and w, which are new equal rank siblings.

Case 3. $r(u) = r(v) = r(w) = r(y) = k$. Since $r(y) = r(v)$, we cannot do a rotation at w. In this case we increase the rank of w to $k+1$ and continue the promote operation with $u = w$ (see Fig. 1.2). If the old rank of w was equal to the rank of its sibling, say z, we use the twin-node credits placed on w and z to charge this operation. If the old rank of w was not equal to the rank of its sibling, then rank of z must be $k+1$ (for z cannot be a minor node as otherwise, y would be a leaf of rank k and w would be a node of rank $k+1$ by properties 2 and 4 of rank, which contradicts our assumption that $r(y) = r(w)$). But this implies that we are now in case 1 at w. So, the promote operation terminates at w and, in this case, we place one twin-node credit each in w and z.

Analysis: The promote operation does not create any new minor nodes and hence the minor-node credit invariant is preserved. In addition, only a rotation can add new spine nodes, and it is easy to see that at most one node becomes a new spine node which would require 2 new spine-node credits. So, the promotion operation requires at most 5 new credits; hence, it has complexity $\bar{O}(1)$.

Let us therefore consider next the demote operation on the rank of a node u with parent v, with $r(u) = k-1$ (see Fig. 2). Let y be the sibling of u. This operation may cause a violation of the bias property, if u is a major node. We have the following cases based on the (new) ranks of the nodes u, v and y:

Case 1. $r(u) = k-2$ and $r(v) = k-1$. This terminates the demote operation as u remains a major node in this case.

Case 2. $r(v) = k, r(y) = k-1$ and $r(u) = k-2$. We have several subcases for this case.

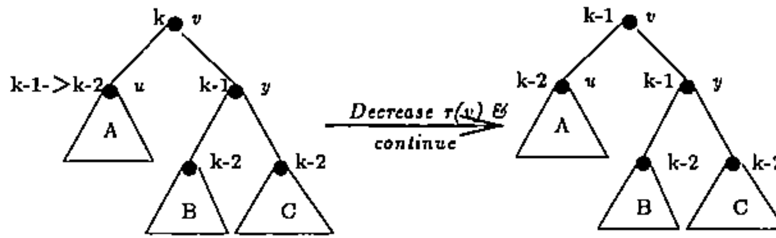
Case 2.1. Both children of y have rank $k-2$. In this case we decrease the rank of v to $k-1$, and continue the rebalancing operation with $u = v$ (see Figure 2.1). The nodes u and y are no longer equal rank siblings. Thus, we can use the credits placed on u and y to charge the operation.

Case 2.2. At least one child of y from u has rank $k-1$. In this case we do a single rotation at v and stop (see Figure 2.2). We place one credit each in equal rank siblings created during the rotation. This operation requires at most 5 credits.

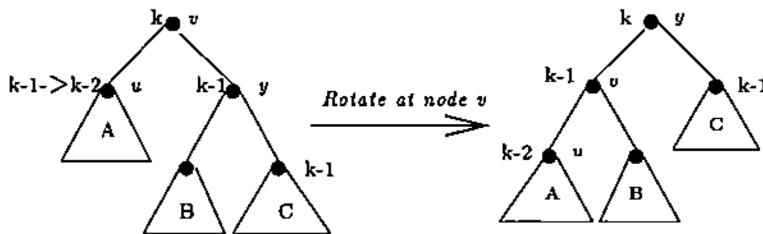
Case 2.3. The child of y closest[3] to u has rank $k-1$ and the other child has rank $k-2$. In this case we do a left double rotation at v and stop (see Figure 2.3). We place credits in equal rank siblings, and this case also requires at most 5 credits.

Case 3. $r(v) = k, r(y) = k, r(u) = k-2$ and both children of y have rank less than or equal to $k-1$ (see Figure 2.4). In this case we perform a single rotation at v, and proceed as in case 2 (since the new sibling of u has rank less than or equal to $k-1$). Here case 2 immediately terminates, as subcase 2.1 is not possible. Hence all subcases in case 2 terminates, and this case requires at most 7 credits.

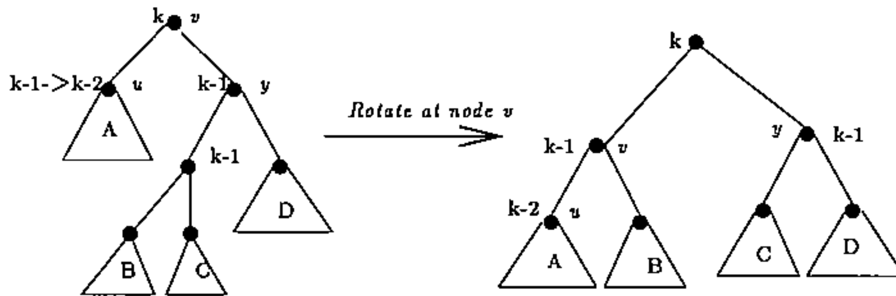
[3]The distance implicitly refered here is with respect to in-order numbering on the nodes.



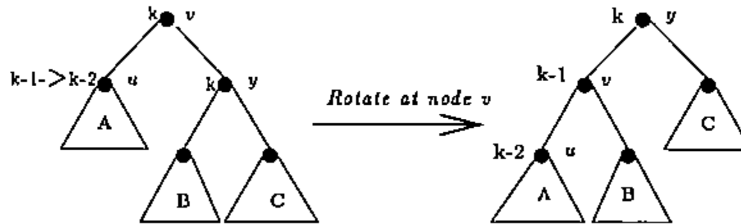
1. Case 2.1 of demote operation. Here rotation is not possible. Continue the rebalancing operation with $u=v$.



2. Case 2.2 of demote operation. Do a single rotation at v and stop.



3. Case 2.4 of demote operation. Do a double rotation at v and stop.



4. Case 3 of demote operation.

Figure 2: Demotion of rank of node u.

Case 4. $r(v) = k, r(u) = k-2$ and $r(y) < k-1$. This case is not possible. For y to be a minor node, the old rank of u must be equal to k, or u must be a leaf of rank $k-1$, both of which contradicts our assumption.

Case 5. $r(u) < r(v) - 2$. This terminates the demote operation, as u is a minor node before the demotion, and demotion does not affect the rank properties. This case requires one new minor-node credit, as rank of u is decreased.

Analysis: The demotion operation creates at most one new minor node. Only rotation can add new spine nodes, and it is easy to see that at most one node becomes a new spine node due to rotation, and requires at most 2 new spine-node credits. So, the demote operation requires at most 8 new credits and hence has complexity $\bar{O}(1)$.

Note that maintenance of *left* and *right* values during promote/demote (actually only rotation requires changes) operations can be done within the time required for rotation. We also note that during promote/demote operations, only rotation creates equal rank siblings (one pair during the promotion operation, and at most two pairs during the demote operation), and at most one rotation takes place during each operation. These observations lead to the following lemma.

Lemma 2.2: *The total complexity of promotion, demotion and rebalancing operations on a biased finger tree due to a single promote/demote operation is $\bar{O}(1)$ (actually at most 8 credits). Also, each operation adds at most two pairs of equal rank siblings to the tree.*

After various update operations on a biased finger tree, we refer to this lemma to bound the complexity of the rebalancing operations to $\bar{O}(1)$.

2.2 Update Operations

We consider insert, delete, split, join and change-weight operations on a biased finger tree and prove an amortized bound for each of these update operations. Let us begin by observing that the maintenance of the *left* and *right* values in the internal nodes and also the *min* and *max* pointers in the root of the tree are quite trivial and can be done within the time bounds. We do not discuss them further in the paper. We use the same credit invariant as in the previous subsection to analyze the update operations.

We begin with the join operation. We describe a "bottom-up" strategy, which contrasts with the "top-down" approach of Bent, Sleator and Tarjan [6].

Join: Consider the join of two biased trees T_x and T_y. Let u and v be the rightmost leaf and the leftmost leaf of T_x and T_y respectively. Let w and l be the parent of u and v respectively (see Fig. 3). The nodes u and v can be accessed using the pointers in the root nodes x and y respectively. We have the following cases:

Case 1. $r(x) = r(y)$. In this case we create a new node z with T_x as the left subtree and T_y as the right subtree, and we assign a rank of $r(x)+1$ to z. We then proceed up the tree as in the promote operation.

Case 2. $r(x) < r(y)$. In this case we traverse the rightmost path of T_x and the leftmost path of T_y bottom up, in the increasing order of ranks of the spine nodes. As we proceed we store the pointers to the nodes encountered and also tags indicating whether they are from T_x or T_y in an array A ordered by node ranks. We also keep track of the nodes of equal ranks last encountered in the two paths and we terminate the traversal when reaching the root x. Suppose t is the smallest rank node along the leftmost path of T_y having rank greater than x.

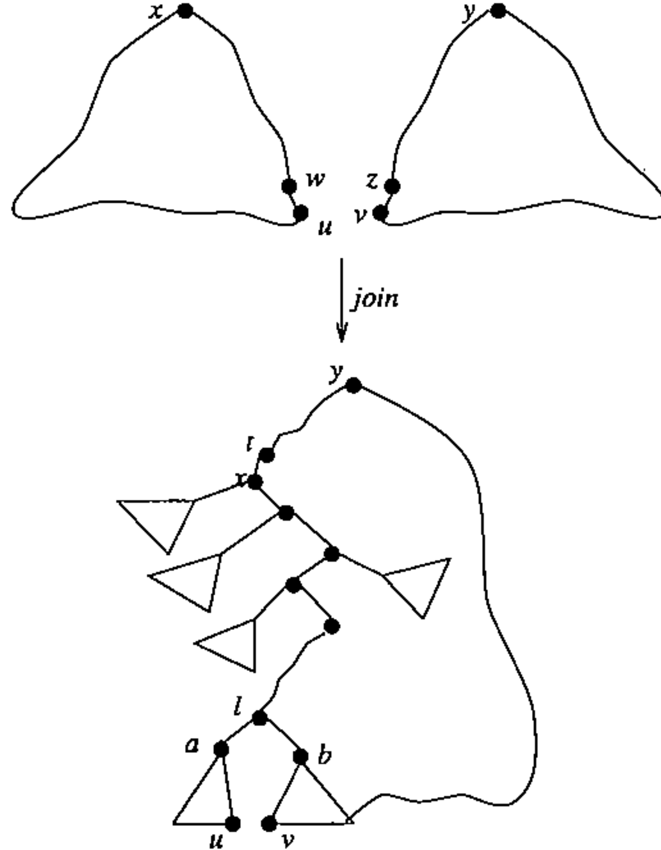


Figure 3: Join of trees T_x and T_y.

Suppose a and b are the last encountered equal rank nodes during the traversal, and note that the node x was stored as the last node in array A. We attach x along with its left subtree to t as the left child of t. For the other nodes in A, we proceed as follows (see Fig. 3): Consider the next node, c, in the array A. Suppose c is part of T_x. If the successor of c in A is a node d from T_x, we attach c and its left subtree as a right child of d. If the successor of c in A is a node d from T_y, we attach c and its left subtree as a left child of d. Suppose c is part of T_y. If the successor of c in A is a node d from T_x, we attach c and its right subtree as a right child of d. If the successor of c in A is a node d from T_y, we attach c and its right subtree as a left child of d. We continue this process till the nodes a and b are encountered. Then, we create a new node z with T_a and T_b as left and right subtrees respectively. We assign a rank of $r(a) + 1$ to z and rebalance if required through a promote operation. This terminates the join. If there are no equal rank spine nodes a and b, then we join all the nodes in the array A in the above manner.

Case 3. $r(y) < r(x)$. Symmetrical to above case.

Analysis: We have 2 spine-node credits in each of the nodes along the rightmost path of T_x and the leftmost path of T_y. In case 1 we add one new spine-node (2 credits), one pair of equal rank siblings (2 credits) and do not add new minor nodes. So, case 1 requires 4 new credits to maintain the credits

invariant and one credit to perform the operation. For cases 2 and 3, we show that with 8 new credits, we can complete the update operation and also maintain the credit invariant. We consider the case 2. The analysis of case 3 follows similarly. We use one of the two credits in each spine node for the upward traversal to locate the node t and the remaining one credit for the downward traversal to construct the resultant tree. We observe that the nodes from x to the rightmost leaf in T_x and the node from t to the leftmost leaf in T_y are not spine nodes in the resultant tree, hence, no longer need to store spine-node credits. The nodes from t to y are still part of the leftmost path, but their credits are in tact. The remaining spine nodes retain their credits from T_x and T_y. We now show that the maintenance of minor-node credits does not require any extra credits. Consider two consecutive nodes c and d on the leftmost path of T_x and two consecutive nodes p and q on the rightmost path of T_y. Suppose the order of nodes is c, p, d and q in the resultant tree. To maintain minor-node credit invariant, we need a total of $r(c) - r(p) - 1 + r(p) - r(d) - 1 + r(d) - r(q) - 1$, i.e., $r(c) - r(q) - 3$ credits in nodes p, d and q. Before we perform the join the nodes d and q contain a total of $r(c) + r(p) - r(q) - r(d) - 2$ credits, which are enough to maintain the minor-node credit invariant, since $r(p) > r(d)$. Similarly for other order of nodes c, p, d and q, we can show that the maintenance of minor-node credits does not require any new credits. Suppose we create the node z during the join, which requires one credit. This operation does not add any new spine or minor nodes. But it creates a pair of equal rank siblings (i.e., a and b) and hence it requires 2 new twin-node credits. Also, the assignment of rank $r(a) + 1$ to node l may trigger more promotions and rebalancing. But it needs at most 5 credits by lemma 2.2. Thus, the complexity of join is at most 8 credits and hence it takes $\bar{O}(1)$ time.

Split: We perform the split operation as in [6]. But we show that with the same complexity we can preserve all three types of credits in the nodes of the resultant trees. The algorithm for *split*(T_x, i) works as follows: Traverse the path from the root node x to the leaf node i and collect the pointers to the fringe subtrees attached to the nodes in the path. Store the pointers to left subtrees and the right subtrees separately in two arrays. Perform repeated joins on the subtrees in the two arrays separately to obtain two biased finger trees, say, T_l and T_r (see Fig. 4). We perform the multiple joins on left subtrees as follows: consider the join of a sequence of left subtrees $l_1, l_2, \cdots, l_k$ obtained during the split. Let $r_1, r_2, \cdots, r_k$, be the ranks of the roots of the subtrees. We first join l_k with l_{k-1} and then join the resultant tree to l_{k-2}. We continue the process till we join the first subtree l_1 to obtain T_l. The construction of T_r follows symmetrically.

Analysis: The length of the path from root to i is bounded by $O(\log W/w_i)$ which also bounds the number of joins. Here the joins do not take $\bar{O}(1)$ time, however, as there are no spine-node credits in the internal nodes to amortize the cost of the joins. The cost of each join operation is bounded by the difference in the ranks of the roots of two subtrees as shown in Bent, Sleator and Tarjan [6]. Thus, the cost of the multiple joins is bounded by $O(\sum_{i=1}^{k-1} r_i - r_{i+1})$, which nicely telescopes to $O(r_1 - r_k)$. This is bounded by $O(\log W/w_i)$, where w_i is the weight of the item i. Similarly the multiple joins of the right subtrees take $O(\log W/w_i)$ time. Also, the nodes of the resultant trees preserve the minor-node credit invariant. This follows because Bent, Sleator and Tarjan [6] show that after the split operation the roots of the two trees have rank $r(x) + 1$ and also all the nodes in the two trees satisfy the minor-node credit invariant. In addition, they show that there are $r(x) + 1 - r(l)$ and $r(x) + 1 - r(r)$ additional credits in the two roots l and r respectively. The number of new spine nodes is bounded by the length of the path from root to i, which is again $O(\log W/w_i)$. Equal rank siblings are created only during rebalancing operations and each takes $\bar{O}(1)$ time (see Lemma 2.2). Finally, the number of rebalancing operations is bounded by the number of joins which is again $O(\log W/w_i)$. So, the split operation requires at most $O(\log W/w_i)$

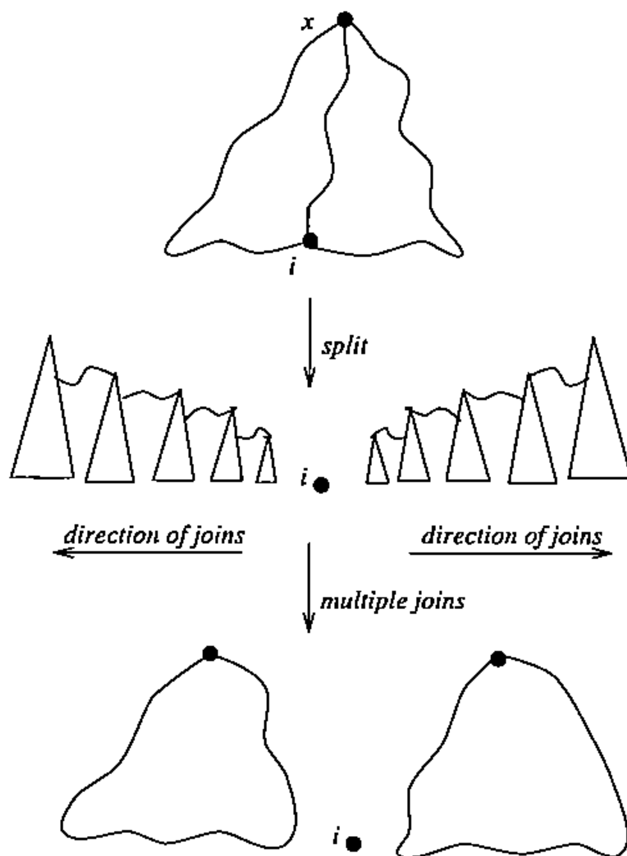


Figure 4: Three-way split of tree T_x with respect to item i.

new credits to complete the update and also to maintain the credit invariant.

Insertion: Consider the insertion of an item i with weight w_i to a biased finger tree T. Let i^- be the immediate predecessor of item i in T if it exists, and immediate successor of i in T otherwise. We provide a pointer to i^-. The algorithm for insert operation $insert(i, w_i, p_j, T)$ proceeds as follows:

Case 1. $r(i) \leq r(i^-)$. Create a new node l with i^- and i as the left and right children respectively and attach l to the parent of i^-. Assign a rank of $r(i^-) + 1$ to the node l. This operation is equivalent to promotion of rank of the node i^- in the original tree and hence may require rebalancing.

Analysis: The rebalancing requires $O(1)$ new twin-node credits by Lemma 2.2. This operation may create at most one new spine node i.e., node l. To maintain minor-node credits, place $r(i^-) + 1 - r(i) - 1$ i.e., $r(i^-) - r(i)$ credits in i. So, this case requires $O(r(i^-) - r(i))$ credits.

Case 2. $r(i^-) < r(i)$. In this case traverse the path from i^- towards the root till a node q with parent p is hit such that $r(q) \leq r(i) < r(p)$. We refer the path from i^- to q as *split path.* Now split the subtree T_q rooted at q as described above into left and right subtrees such that i^-

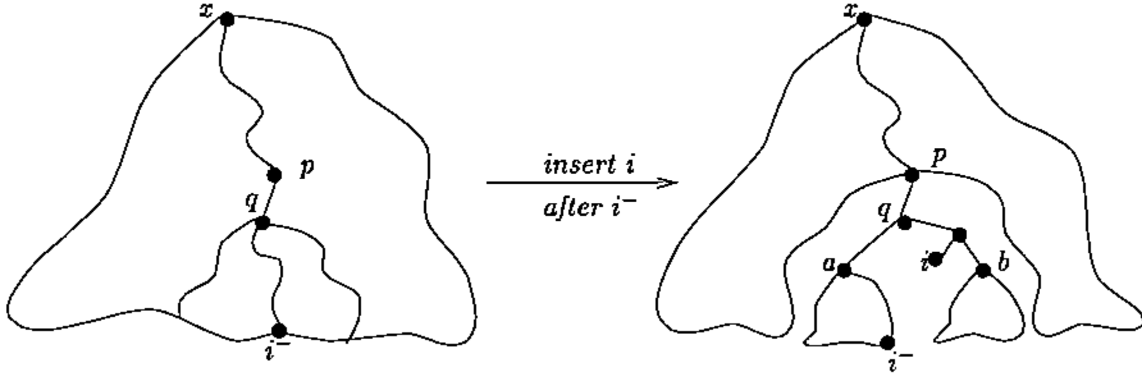


Figure 5: Insertion of node i

is the rightmost element in the left subtree. Let T_a and T_b be the two resultant trees. Now join T_a with the item i and then join the resultant tree with T_b. The tree resulting from the join is shown in Fig. 5. The rank of the root of this tree is $r(i)+1$. Attach the resultant tree in place of q in the original tree. The promotion of rank of node q to $r(i)+1$ may require rebalancing, so we complete this case by performing any promotions needed, as described in section 2.1.

Analysis: Suppose we start with $O(r(i)-r(i^-))$ credits. Computing the split path takes $O(d_T(i^-)-d_T(q))$ time, which is bounded by $O(r(q)-r(i^-))$, by Lemma 2.1. Splitting the tree T_q takes $\bar{O}(r(q)-r(i^-))$ time, which is bounded by $\bar{O}(r(i)-r(i^-))$. The join of the trees T_a, T_i and T_b takes $\bar{O}(1)$ time as i is a leaf and $r(i) \geq \max\{r(a), r(b)\}$. The rebalancing due to promotion of rank of node q requires $O(1)$ new twin-node credits. This case adds at most one new spine node as the tree T_b remains unaffected during the join (see Fig. 5). The split operation preserves the minor-node credits and as the rank of node q is only promoted, no new minor-node credits are needed. So, the complexity of case 2 is $\bar{O}(r(i)-r(i^-))$.

Case 3. i does not have a predecessor in T_x. This case is equivalent to a join of T_i and T. Traverse the path of T_x up from i^- (which now denotes the successor of i) to locate a node a with parent b such that $r(a) \leq r(i) < r(b)$. Now create a new node l with i and a as the left and right children and attach l as the left child of b (see Fig. 6). Assign a rank of $r(i)+1$ to l.

Analysis: The update takes $\bar{O}(1)$ time as we can use the spine-node credits to charge the traversal along the leftmost path. The rebalancing after insertion of node l requires at most $O(1)$ new twin-node credits. This case adds at most two new spine nodes i.e., nodes i and l and hence it requires 4 spine-node credits. To maintain the minor-node credits add $r(i)-r(a)$ credits to node a which is bounded by $r(i)-r(i^-)$, where i^- is the successor of i in T.

So, the complexity of insert operation is $\bar{O}(|(r(i)-r(i^-))|)$, where i^- is the predecessor of i in T. In other words, the time complexity for an insertion is $\bar{O}(\left|\log \frac{w_i}{w_{i^-}}\right|+1)$.

Deletion: Consider the deletion of an item i with weight w_i from the biased finger tree T for which a pointer to the item i is provided. Let j be the sibling of i, let u be the parent of i, let k be the

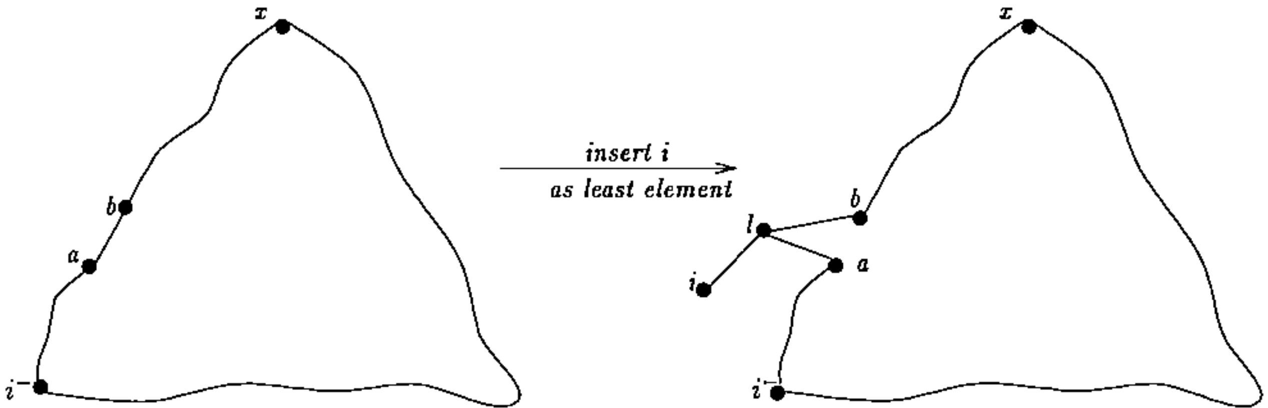


Figure 6: Leftmost Insertion of node i

sibling of u, and let l be the parent of u and k (see Fig. 7). The algorithm for deletion proceeds as follows:

Case 1. The node i is a minor node. Delete i and attach T_j in place of T_u (see Fig. 7.1). This is equivalent to demotion of rank of u and may require rebalancing, as described in section 2.1.

Analysis: The rebalancing requires $O(1)$ new twin-node credits. This operation adds at most 2 spine nodes since the biasing condition forces either j or its right child to be a major leaf. This operation does not add any minor nodes. So, this case requires $O(1)$ credits.

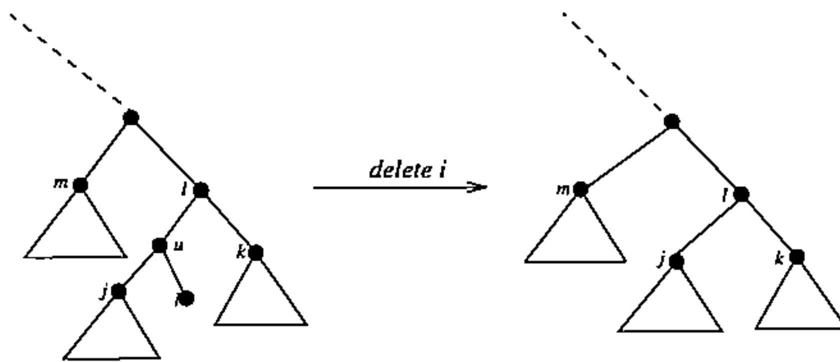
Case 2. The node i is a major node and at most one of j or k is a minor node. Join T_j and T_k and attach the resultant tree in place of T_l (see Fig 7.2). This is equivalent to demotion of rank of l and may require rebalancing.

Analysis: The rebalancing requires $O(1)$ new twin-node credits. Suppose j is the minor node which implies $r(j) < r(k)$. Then the join requires $O(r(k) - r(j))$ credits which is bounded by $O(r(i) - r(j))$. But these credits are available as minor-node credits in j. This operation does not add any new minor nodes and adds at most $O(r(i))$ new spine nodes, if i occurs along either of the extreme paths.

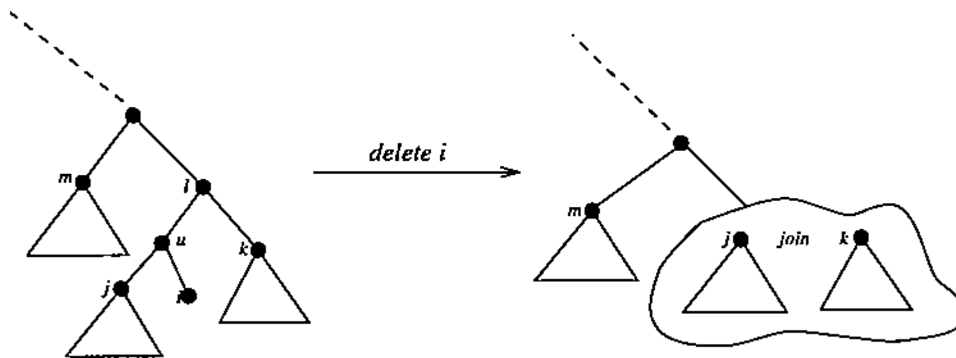
Case 3: The node i is a major node and and both j and k are also major nodes. Join T_j and T_k and attach the resultant tree in place of T_l (see Fig 7.2). This operation if it promotes rank of l requires rebalancing.

Analysis: The rebalancing requires $O(1)$ new twin-node credits. Then the join requires $O(|r(k)-r(j)|)$ credits which is a constant since both k and j are major nodes. This operation does not add any new minor nodes and adds at most $O(r(i))$ new spine nodes, if i occurs along either of the extreme paths.

Case 4. The node i is a major node and and both j and k are minor nodes. Then, the biasing condition insists that the sibling of l, say m, be a major node. Now join T_j with T_k and then join the resultant tree to T_m. Attach the resultant tree in place of the parent of l. This is equivalent to demotion of rank of the parent of l and may require rebalancing.



1. Case 1 of deletion. Node i is a minor node.



2. Case 2 of deletion. Node i is a major node and j or k is a major node.

Figure 7: Deletion of node i.

Analysis: The rebalancing requires $O(1)$ new twin-node credits. Suppose $r(k) \geq r(j)$. The other case is symmetrical. The join of T_j and T_k takes time $\bar{O}(r(k) - r(j))$. Let T_p be the result The node m cannot be a minor node, as otherwise bias property is violated. Suppose l is a minor node. Then either m or right child of m is a major leaf, and hence the join of T_p and T_m takes constant time. In this case, the over all time for the two joins is bounded by $\bar{O}(r(k) - r(j) + c)$, where c is constant, and this in turn is bounded by $\bar{O}(r(i) - r(j)))$, as i is a major node. Suppose l is a major node. Then the join of T_p and T_m takes time $\bar{O}(r(m) - r(k))$. So, the time for the two joins telescope to $\bar{O}(r(m) - r(j))$. This is bounded by $\bar{O}(r(i) - r(k) + 3)$, since i and m are both major nodes, and also the levels of i and m' parent differ by 3 (see Lemma 2.1). This operation preserves minor-node credits and adds at most $O(r(i))$ new spine nodes, if i occurs along either of the extreme paths.

So, the overall complexity of deletion of node i is $\bar{O}(r(i))$, which is bounded by $\bar{O}(\log w_i)$.

Corollary 2.3: *If node i does not occur along either of the extreme paths of T, then deletion of i takes $\bar{O}(1)$ time.*

Proof: Since i does not occur along the extreme paths of T, after deltion of i, no new spine-credits are required to preserve the credit invariant. Also, other operations can be charged by minor-node and twin-node credits. ■

Change-weight: Consider changing the weight of an item i from w_i to $w_{i'}$. We provide a pointer to the node i for this operation. Let j be the sibling of i, let u be the parent of i, let k be the sibling of u and let l be the parent of u and k. Let $r(i')$ be the new rank of i. The algorithm for change-weight operation proceeds as follows:

Case 1. i is a minor node and $r(i') \leq r(i)$. This case does not require any rebalancing as i is still a minor node.

Case 2. i is a minor node and $r(i) < r(i')$. If $r(i') < r(u)$ then, no change is required. Otherwise insert i' as in case 2 of insertion. This takes time $\bar{O}(r(i') - r(i))$.

Case 3. i is a major node and $r(i) < r(i')$. Proceed as in case 2 of insertion to insert i'. This takes time $\bar{O}(r(i') - r(i))$.

Case 4. i is a major node, $r(i') \leq r(i)$ and either i^- or k is a major node. Suppose $r(k) \geq r(j)$. Join T_j with $T_{i'}$ and then join the resultant tree with T_k. Attach the resultant tree in place of l. These joins telescope and take time $\bar{O}(r(k) - r(i'))$, which is bounded by $\bar{O}(r(i) - r(i'))$, since i is a major node.

Case 5. i is a major node, $r(i') \leq r(i)$ and both j and k are minor nodes. Now, the biasing condition insists that the sibling of l, say m, must be a major node. In this case join $T_j, T_{i'}$ and T_k in order and then join the resultant tree with T_m. These joins telescope and take time $\bar{O}(r(m) - r(i'))$, which is bounded by $\bar{O}(r(i) - r(i'))$, since i is a major node.

So, the complexity of change-weight operation is $\bar{O}(|r(i') - r(i)|)$, which is $\bar{O}(\left|\log \frac{w_{i'}}{w_i}\right|)$.

Slice: Consider slicing the item i with weight w_i into two items i_1 and i_2 of weights $x * w_i$ and $(1 - x) * w_i$ respectively, where $x \in \mathbb{R}$, and $0.0 < x < 1.0$. We note that at least one of the two nodes has rank at least $r(i) - 1$. Suppose j is the left sibling of i (the other case is symmetrical). Let u be the parent of i. We show that using $\bar{O}(r(i) - min(r(i_1), r(i_2)))$ credits, we can complete the slicing operation, and also maintain the credit invariant. We have the following cases:

Case 1. i is a minor node. Create a new node l with i_1 and i_2 as the two children. Attach l in place of i to i's parent. Assign $r(i)$ as the rank of l. Place $r(u) - r(i) - 1$ credits in l (which were in i as i is minor). This operation does not require any propagation of ranks, and adds at most one more spine node. At most one of i_1 and i_2 can be minor. If, say i_2, is minor, place $r(i) - r(i_2) - 1$ credits in i_2. This case takes $\bar{O}(r(i) - r(i_2))$ time.

Case 2. i is a major node and j is a minor node, and $r(j) < r(i_1)$. Create a new node m with T_j as left subtree and i_1 as right child. Then create a new node l with l and i_2 as left and right children, respectively. Attach the resultant tree in place of u. Do demotion of rank, if required. Assign a rank of $r(i_1) + 1$ to m and a rank of $r(i)$ to l. Place appropriate minor node credits, if required. This case takes $\bar{O}(r_i - min(r(i_1), r(i_2)))$ time.

Case 3. i is a major node and j is a minor node, and $r(j) \geq r(i_1)$. Create a new node l with i_1 and i_2 as left and right children, and then join T_l with T_j. Attach the resultant tree in place of u. Do a demotion of rank, if required. Assign a rank of $r(i)$ to l. Place appropriate minor node credits, if required. In this case, the join takes $\bar{O}(r(i) - r(j)$ time and other operations require $\bar{O}(r(i) - min(r(i_1), r(i_2)))$. Since $r(j) \geq r(i_1)$, the overall time for this case is bounded by $\bar{O}(r(i) - min(r(i_1), r(i_2)))$.

Case 4. i is a major node and j is also a major node. Create a new node l with i_1 and i_2 as the two children, and then join T_l with T_j. Attach l in place of u. Assign $r(i)$ as the rank of l. Do promotion/demotion of ranks of nodes, if required. To maintain minor-node credit invariant, place appropriate number of credits in i_1 or i_2. As in case 3, the overall time for this case is bounded by $\bar{O}(r(i) - min(r(i_1), r(i_2)))$.

Case 5: $x \approx 1$ or $r(i_2)$ is a constant (the other case $r \approx 0$ is symmetrical). This is a special case. We create a new node l with i_1 and i_2 as children and attach in place of i, if i_2 does not cause any violation of bias property i.e., if right sibling, j, of i is not a minor node. If there is a violation, then we use *left* value of j to identify the left most node in T_j, say q, and then traverse upwards to identify proper sibling of i_2, within constant steps. The complexity of this case is $\bar{O}(1)$.

Analysis: All cases add at most $O(1)$ new spine nodes and equal rank siblings. So, the complexity of slice operation is bounded by $\bar{O}(r(i) - min(r(i_1), r(i_2)))$, which is $\bar{O}(\log \frac{w_i}{min(w_{i_1}, w_{i_2})})$.

Corollary 2.4: *The complexity of slice*(i, w_i, S, x, i_1, i_2), *with* $x \approx 0.5$, *or* $x \approx 1.0$, *or* $x \approx 0.0$, *is* $\bar{O}(1)$.

Proof: Case 5 gives proof for part of the claim. If $x \approx 0.5$, both i_1 and i_2 are major nodes and we do not require any new credits to maintain minor-node credit invariant. Also the joins in all the cases take constant time, since i_1 is a major leaf and its rank is either greater than j or is smaller by at most 2. ■

Fuse: Consider fusing of two leaf items i_1 and i_2 with weights w_1 and w_2 respectively. We deal with the cases when i_1 and i_2 are siblings, and when they are not siblings, separately. When i_1 and i_2 are siblings, let the parent of i_1 and i_2 be u and let v be the sibling of u. Also, let w be the parent of u and v. We have the following cases:

Case 1. $r(i_1) = r(i_2)$ and i_1 and i_2 are siblings. Collapse i_1 and i_2 into a single leaf i of rank $r(i_1) + 1$ and attach i in place of u. No promotion/demotion of ranks of nodes is required.

To preserve the three credit invariants, we do not require any new credits. This case takes $\bar{O}(1)$ time.

Case 2. $r(i_1) > r(i_2)$ and i_1 and i_2 are siblings. Create a new leaf i of rank $r(i_1)$, and then join T_i with T_v. Attach the resultant tree in place of w. Do a demotion of ranks to nodes in the higher levels, if required. The join of T_i and T_v take $\bar{O}(1)$ time. To see this, if u is minor node, then join i and T_v directly. Otherwise, if v is a minor node, then join i and T_v directly; if v is a major node, then the join algorithm traverses constant number of edges along the leftmost or rightmost path of T_v before locating the sibling for i. So, this case takes $\bar{O}(1)$ time.

Case 3. $r(i_2) > r(i_1)$ and i_1 and i_2 are siblings. Symmetrical to above case.

Case 4. $r(i_1) = r(i_2)$ and i_1 and i_2 are not siblings. Delete i_1, rename i_2 as i and increase the rank of i by 1. Do a propagation of ranks to nodes in the higher levels, if required. If i_1 is not on a root-to-leaf path (see Corollary 2.3), then this case takes $\bar{O}(1)$ time, and $\bar{O}(r(i_1))$ time, otherwise.

Case 5. $r(i_1) > r(i_2)$ and i_1 and i_2 are not siblings. Delete i_2, rename i_1 as i and change the rank of i to $\log(w_{i_1} + w_{i_2})$ (this operation may increase the rank of i by at most 1). Do a propagation of ranks to nodes in the higher levels, if required. If i_2 is not on a root-to-leaf path (see Corollary 2.3), then this case takes $\bar{O}(1)$ time, and $\bar{O}(min(r(i_1), r(i_2)))$ time, otherwise.

Case 6. $r(i_2) > r(i_1)$ and i_1 and i_2 are not siblings. Symmetrical to above case.

So, the complexity of fuse operation is $\bar{O}(min(r(i_1), r(i_2)))$, which is $\bar{O}(min(\log w_{i_1}, \log w_{i_2}))$, if smaller weight item among i_1 and i_2 does not appear on a root-to-leaf path, and $\bar{O}(1)$ time, otherwise.

We summarize:

Theorem 2.5: *One can maintain a collection of biased search trees subject to tree searches, element insertion and deletion, change of weight of element, slicing and fusing of elements, as well as tree joining and splitting, in the bounds quoted in Table 1 for biased finger trees, discounting search times.*

3 Some Properties of Biased Finger Trees

In this section, we prove some properties of biased finger trees. For example, Bent *et al.* [6] show that repeated single-node joins on the right hand side can construct a biased finger tree in $O(n)$ worst-case time. In our case, however, we can show the following:

Theorem 3.1: *Any sequence of joins that constructs a biased finger tree of n items can be implemented in $O(n)$ worst-case time.*

Proof: The proof follows immediately from the fact that our join algorithm on biased finger trees takes $\bar{O}(1)$ time. ■

The above result raises the question of whether or not a linear time construction of a biased tree is possible for an arbitrary sequence of insertions of items. We now prove a lower bound that answers the question in the negative and also proves that our time for insertion is optimal.

First we state a result from information theory that is relevant to weighted trees.

Theorem 3.2: *(Abramson [1]) Consider any search tree T for a set of weighted items, S, where item i has weight w_i and depth d_i in T. If every node of T has at most b children, then the total weighted depth $\sum_{i \in S} w_i d_i$ is at least $W \sum_{i \in S} p_i \log_b(1/p_i)$, where $p_i = w_i/W$ and W is the total weight of items in S.*

It is proved in [6] that the depth of an item i in a biased tree is at most $\log W/w_i + 1$, where W is the total weight. We now prove an identity relating the weights and heights of items in a biased tree.

Lemma 3.3: *The height h_i of an item i with weight w_i in a biased tree T is at most $\log w_i/w_s$, where w_s is the weight of the smallest item in the tree.*

Proof: Expand each leaf item i of weight w_i in T, by a biased tree T_i of w_i items, each of weight 1. The depth of an old leaf item i in the new tree is $\log W/w_i + 1$ and the maximum depth of a leaf in the subtree of i is $\log W + 1$. But all the nodes at bottom $\log w_s$ levels are virtual nodes created by the above expansion. So, the height of each item i in the original tree is at most $\log w_i/w_s$, with the smallest item at height 0. ■

So, we have the following identity relating the sum of the depths of the nodes and the sum of the heights of the nodes in a biased tree.

Corollary 3.4: *In a biased tree T representing a set S of weighted items, $\sum_{i \in S} h_i + d_i \leq n \log W/w_s + n$, where h_i and d_i are the height and depth of an item i respectively.*

But the sum of heights of the nodes is always bounded the sum of the depths as shown in the following lemma.

Lemma 3.5: *In a biased tree T representing a set of S weighted items, we have $\sum_{i \in S} \log w_i/w_s \leq \sum_{i \in S} h_i < \sum_{i \in S} d_i \leq \sum_{i \in S} \log W/w_i + n$, where W is the total weight of the items in the tree and w_s is weight of the smallest item in the tree.*

Proof: Omitted as it can be shown easily by induction. ■

The above lemma suggests that updating algorithms that work in a bottom-up fashion on a biased finger tree may have better bounds than top-down algorithms. We prove that this is indeed the case. First we show a lower bound for an arbitrary sequence of insertions into a biased tree and then show that our insertion algorithm achieves that bound whereas any insertion algorithm that works in a top-down fashion cannot reach that bound.

Theorem 3.6: *There is a sequence of insertions that requires $\Omega(\sum_{i \in S} \log w_i/w_s)$ time in any biased tree, where w_i is the weight of the item i and w_s is the smallest weight in the set S.*

Proof: First we observe that any top-down method of insertion must deal with the lower bound to fix the position of the item in the total order, which is $\Omega(\log W/w_i)$ time by theorem 3.2. This observation along with Lemma 3.5 prove that any top-down method of construction of a biased tree cannot contradict the claimed lower bound.

We now prove that any bottom-up method using fingers also cannot contradict this bound. Any algorithm for *insert*(i, w_i, p_j, T) must first raise the item i to its biased height and then perform a split on the underlying set S to separate the items smaller than i and bigger than i and then insert i. Any such split should take time proportional to the height of the item i in the tree in the worst-case, which is $\Omega(\log w_i/w_s)$. For leftmost and rightmost insertions, there is no need for split. We now show an adversary sequence of insertions which requires a time, at least equal to the claimed lower bound. First insert an item s of weight w_s. Then insert $n-1$ items of weights

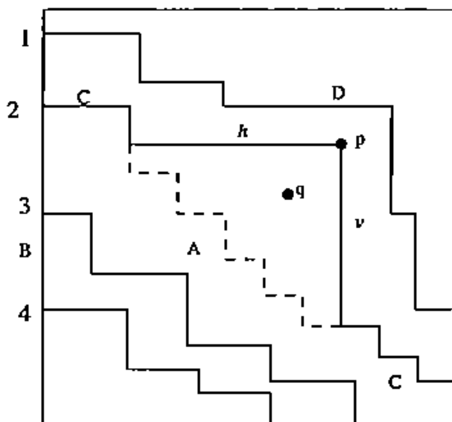


Figure 8: Computation of Layer for a New Point p.

$w_1, w_2, \cdots, w_{n-1}$ respectively with $w_i = (\sum_{j=1}^{i-1} w_j) + w_s$, alternately as successor and predecessor of item s. When inserting an item i, it must be first raised to the level of the current root (since $w_i = (\sum_{j=1}^{i-1} w_j) + w_s$) and then the tree must be split into two trees containing $\lfloor \frac{i-1}{2} \rfloor$ and $\lceil \frac{i-1}{2} \rceil$ items respectively. The depth of the tree at the time of inserting i is $O(\log w_i/w_s)$ and hence the split takes time proportional to the depth. So, the total time required for inserting n items is $O(1 + \sum_{i=1}^{n-1} \log w_i/w_s)$ time which is $\Omega(\sum_{i \in S} \log w_i/w_s)$. ■

Corollary 3.7: *Our insertion algorithm on a biased finger tree is optimal.*

Proof: The complexity of $insert(i, w_i, p_j, T)$ operation is $\bar{O}(|\log w_i/w_j|)$. The bound is larger if w_j's are smaller and hence the complexity for worst-case sequence of insertions is $\bar{O}(\sum_{i \in S} \log w_i/w_s)$, which is optimal by previous theorem. ■

Let us now turn our attention to some non-trivial applications.

4 The Layers-of-Maxima Problem

In this section, we use the biased finger tree data structure to solve an open (static) computational geometry problem: the 3-dimensional layers-of-maxima problem. Before we describe our method, however, we introduce some notations. A point $p \in \mathbb{R}^3$ *dominates* a point $q \in \mathbb{R}^3$, if $x(q) < x(p)$, $y(q) < y(p)$, and $z(q) < z(p)$. Given a set S of points in $\mathbb{R}^3$, a point p is a *maximum* point in S, if it is not dominated by any other point S. The *maxima set* problem is to find all the maximum points in S. Kung, Luccio, and Preparata [26] showed that this problem can be solved in $O(n \log n)$ time. In the related *layers-of-maxima* problem, one imagines an iterative process, where one finds a maxima set M in S, removes all the points of M from S, and repeats this process until S is empty. The iteration number in which a point p is removed from S is called p's *layer*, and we denote it by $l(p)$, and the layers-of-maxima problem is to determine the layer of each point p in S. This is related to the well-known *convex layers* problem [8], and it appears that it can be solved for a 3-dimensional point set S in $O(n \log n \log\log n)$ time [2] using the dynamic fractional cascading technique of Mehlhorn and Näher [31]. We show how to solve the 3-dimensional layers-of-maxima problem in $O(n \log n)$ time, which is optimal.

We solve this problem using a three-dimensional sweep, and a dynamic method for point location in a rectilinear subdivision. Given a set S of n points in $\mathbb{R}^3$, we first sort the points along the z axis and then sweep the points in the decreasing order of their z coordinates to construct the maxima layers. During the construction, we maintain only a subset of points, which belong to each layer. This simplifies the identification of layer for a new point. Let $S' \subset S$ be the current set of points maintained by the algorithm. We show that S' has the property that the projections of the points in S' onto the xy-plane rectilinearly subdivide the xy-plane, with points belonging to the same layer forming a staircase. We show that this property is preserved at each step of the sweep, when we compute the layer for a new point. Hence, at any instant the current set of maxima layers forms a disjoint collection of staircases in a rectilinear subdivision. We call the region in xy-plane between two staircases, a *face*. Hence, if there are m layers, they subdivide the xy-plane (actually, the projection of points in S' uses a much smaller area, which can be bounded by a rectangle) into $m+1$ faces. The projection of each new point onto xy-plane belongs to a unique face among these $m+1$ faces. We work with the rectilinear subdivision and the projection of new point, to identify its layer.

The algorithm for computing the layer number of a new point p is, then, as follows:

1. Identify the two staircases in the rectilinear subdivision between which the new point p lies. Assign p to higher-numbered layer, of these two. For example, in Figure 8, p lies between the layers 1 and 2, and gets assigned to layer 2. If p lies between the highest-numbered layer, say m, and the boundary, then assign p to a new layer $m+1$.

2. Compute the horizontal segment h, and the vertical segment v from p which hits the boundary or some layer.

3. Insert the segment h and the segment v into the subdivision.

4. Delete the segments in the layer $l(p)$, which are dominated by p in the xy-plane. For example, in Figure 8, we delete segments in portion A of layer 2.

Correctness and Analysis: We now show that each new point p is identified with its correct layer. We use the Figure 8, to illustrate the idea. Since the points are processed in the decreasing order of z coordinates, the point p does not dominate any of the points in layer 2. Also the points in layer 2 do not dominate p along x and y coordinates. So, point p belongs to layer 2 (i.e., $l(p) = 2$), as identified by the algorithm. We delete the points in portion A of $l(p)$, as later if point q is introduced, the algorithm will identify q with layer $l(p)$. But q does not belong to layer $l(p)$, since q is dominated by p. So, it should either initiate a new layer, or belong to a layer (see Figure 8). In any case, $l(q) = l(p) + 1$. Also, we observe that after deleting points in layer A, the projections of the current set S' again rectilinearly subdivide the bounding rectangle, with points in same layer forming a staircase, thus preserving the property.

Suppose location, insertion and deletion of a vertex/edge in a rectilinear subdivision take time $Q(n), I(n)$, and $D(n)$ respectively. We implement the step 1 as a point location in rectilinear subdivision, and it takes $O(Q(n))$ time. We represent the staircase corresponding to each layer by two dictionaries, one for ordering along x axis, and the other for ordering along y axis. Using these data structures, we compute in $O(\log n)$ time the horizontal segment h and the vertical segment v for a new point p. Hence, the step 2 takes $O(\log n)$ time. So, the total time for computing the layer of each point is $O(\log n + Q(n) + I(n) + k * D(n))$, where k is the number of points deleted in step 4. Since each point is deleted at most once, and is not inserted back, we amortize the cost of k deletions on each of the k points. Hence, the complexity of computing layer of each new point is $\bar{O}(\log n + Q(n) + I(n) + D(n))$. In the next section, we show a method which achieves

Type	Queries	Insert	Delete	Reference
general	$O(\log n \log\log n)$	$\bar{O}(\log n \log\log n)$	$\bar{O}(\log^2 n)$	Baumgarten *et al.* [5]
connected	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$	Cheng-Janardan [11][5]
monotone	$O(\log n)$	$\bar{O}(\log^2 n)$	$\bar{O}(\log^2 n)$	Chiang-Tamassia [9]
monotone	$O(\log^2 n)$	$O(\log n)$	$O(\log n)$	Goodrich-Tamassia [18]
rectilinear	$O(\log n \log\log n)$	$\bar{O}(\log n \log\log n)$	$\bar{O}(\log n \log\log n)$	Mehlhorn-Näher [31]
convex	$O(\log n + \log N)$	$O(\log n \log N)$	$O(\log n \log N)$	Preparata-Tamassia [39]
convex[6]	$O(\log n)$	$\bar{O}(\log n)$	$\bar{O}(\log^2 n)$	this paper

Table 2: Previous and New results in dynamic point location. N denotes the number of possible y-coordinates for edge endpoints in the subdivision.

$Q(n) = I(n) = D(n) = \bar{O}(\log n)$, resulting in a $\bar{O}(\log n)$ algorithm for computing the layer of a single point. This gives us an $O(n \log n)$ algorithm for computing the three dimensional layers-of-maxima, and is optimal [4].

5 Dynamic Point Location

In this section, we tackle the general problem of dynamic point location in a convex subdivision. We also show how it applies to the layers-of-maxima problem. So, suppose we are given a connected subdivision $\mathcal{S}$ of the plane such that $\mathcal{S}$ partitions the plane into two-dimensional cells bounded by straight line segments. The *point location* problem is to construct a data structure that allows one to determine for any query point p the name of the cell in $\mathcal{S}$ that contains p (see [13, 15, 16, 22, 27, 28, 34, 35, 40]). It is well-known that one can construct a linear-space data structure for answering such queries in $O(\log n)$ time [13, 16, 22, 40].

These optimal data structures are *static*, however, in that they do not allow for any changes to $\mathcal{S}$ to occur after the data structure is constructed. There has, therefore, been an increasing interest more recently into methods for performing point location in a dynamic setting, where one is allowed to make changes to $\mathcal{S}$, such as adding or deleting edges and vertices. It is easy to see that, by a simple reduction from the sorting problem, a sequence of n queries and updates to $\mathcal{S}$ requires $\Omega(n \log n)$ time in the comparison model, yet there is no existing fully dynamic framework that achieves $O(\log n)$ time for both queries and updates (even in an amortized sense). The currently best methods are summarized in Table 2. The results in that table are distinguished by the assumptions they make on the structure of $\mathcal{S}$. For example, a *convex* subdivision is one in which each face is convex (except for the external face) (see Figure 9), a *rectilinear* subdivision is one in which each edge is parallel to the x- or y-axis, a *monotone* subdivision is one in which each face is monotone with respect to (say) the x-axis, a *connected* subdivision is one which forms a connected graph, and a *general* subdivision is one that may contain "holes." We also distinguish between worst-case

[4]A simple linear time reduction can be shown from sorting problem to three-dimensional layers-of-maxima problem, thereby showing a lower bound of $\Omega(n \log n)$ for three-dimensional layers of maxima problem.

[4]Cheng and Janardan's update method is actually a de-amortization of an amortized scheme via the "rebuild-while-you-work" technique of Overmars [33].

[5]Our method can actually used for any dynamic point location environment satisfying a certain *pseudo-edge* property.

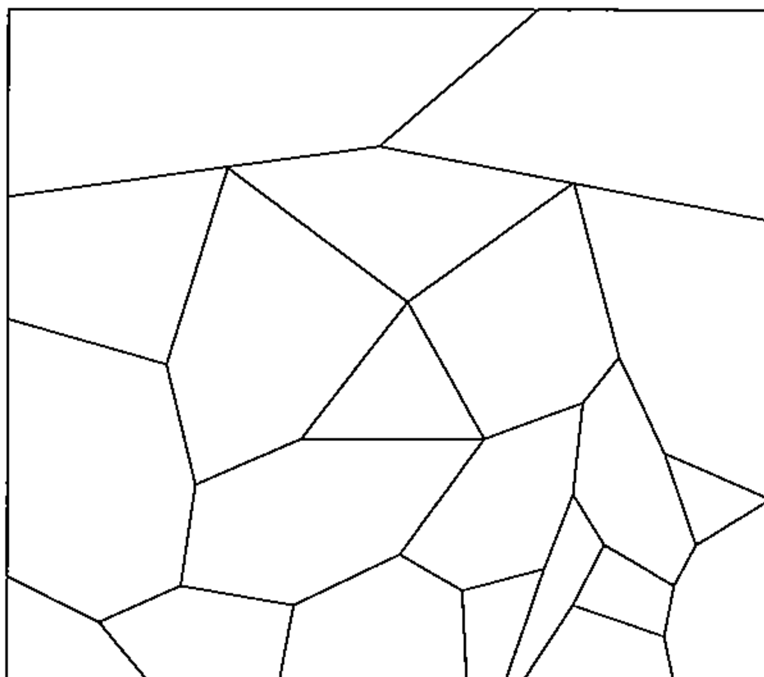


Figure 9: A Planar Convex Subdivision.

running times, which we denote using the usual "Big-Oh" notation, and amortized running times, which we denote using the notation $\bar{O}(*)$. The interested reader is referred to the excellent survey by Chiang and Tamassia [10] for a discussion of these and other results in dynamic computational geometry.

In this paper we give a scheme for dynamic point location in a convex subdivision that achieves $O(\log n)$ time for performing a point location query for any point $p \in \mathbb{R}^2$, $\bar{O}(\log n)$ time for inserting an edge e into $\mathcal{S}$, and $\bar{O}(\log^2 n)$ time for deleting an edge e from $\mathcal{S}$. Thus, our method has an optimal update time of $\bar{O}(\log n)$ in the on-line (also called "semi-dynamic") case. Our method is based upon a variant of the static "trapezoid" method of Preparata [34], dynamized through the use of the biased finger tree data structure. Interestingly, this bound could only be achieved previously by not allowing for deletion at all'[?].

Before we give the main ideas of our data structure for dynamic point location let us review an important data structure, which will comprise the primary structure for our method.

5.1 Weight-Balanced Binary Search Trees

Suppose we wish to maintain an ordered set of elements S in a binary search tree, T, subject to tree-search queries and insert and delete update operations. Moreover, suppose we wish to maintain an auxiliary data structure for each node v in T, such that the cost of performing a rotation at a node v in T takes time proportional to the number of descendents of v. If we implement T as a weight-balanced $BB[\alpha]$-tree data structure [7, 29, 32], then we can maintain T so that tree-search queries take $O(\log n)$ time in the worst case (for this is the worst-case depth of T) and insert and delete operations take $\bar{O}(\log n)$ time (e.g., see Mehlhorn [29]).

5.2 Our Data Structure

Suppose we are given a convex subdivision $\mathcal{S}$ that we would like to maintain dynamically subject to point location queries and edge and vertex insertions and deletions. As mentioned above, our method for maintaining $\mathcal{S}$ is based upon a dynamic implementation of the "trapezoid method" of Preparata [34] for static point location. Incidentally, this is also the approach used by Chiang and Tamassia [9], albeit in a different way. Let us assume, for the time being, that the x-coordinates of the segment endpoints are integers in the range $[1, n]$ (we will show later how to get around this restriction using the the $BB[\alpha]$ tree). We define our structure recursively, following the general approach of Preparata [34]. Our structure is a rooted tree, T, each of whose nodes is associated with a trapezoid τ whose parallel boundary edges are vertical. We imagine the trapezoid τ as being a "window" on $\mathcal{S}$, with the remaining task being that of locating a query point in $\mathcal{S}$ restricted to this trapezoidal window. With the root of T we associate a bounding rectangle for $\mathcal{S}$.

Let v therefore be a node in T with trapezoid τ associated with it. If no vertex or edge of $\mathcal{S}$ intersects the interior of τ, then we say that τ is *empty*, in which case v is a leaf of T. Note that in this case any point determined to be inside τ is immediately located in the cell of $\mathcal{S}$ containing τ. Let us therefore inductively assume that τ contains at least one endpoint of a segment in $\mathcal{S}$. There are two cases:

1. There is no face of $\mathcal{S}$ that intersects τ's left and right boundaries while not intersecting τ's top or bottom boundary. In this case we divide τ in two by a vertical line down the "middle" (we choose a vertical line which balances the height of the tree on both sides) of τ, an action we refer to as a *vertical* cut. This creates two new trapezoids τ_l and τ_r, which are ordered by the "right of" relation. We create two new nodes v_l and v_r, which are respectively the left and right child of v, with v_l associated with τ_l and v_r associated with τ_r.

2. There is at least one face of $\mathcal{S}$ that intersects both the left and right boundaries of τ and does not have a spanning edge of τ as its top or bottom boundary. In this case we "cut" τ through each of the faces of $\mathcal{S}$ that intersect τ's left and right boundaries. This creates a collection of trapezoids $\tau_1, \tau_2, \ldots, \tau_k$ ordered by the "above" relation. We refer to this action as a collection of *horizontal* pseudo-cuts (even though it would be more accurate to call them "non-vertical pseudo-cuts"). We associate a node v_i in T with each τ_i and make this set of nodes be the children of v in T, ordered from left-to-right by the "above" relation on their respective associated trapezoids.

Repeating the above trapezoidal cutting operations recursively at each child of v creates our tree T (see Figure 10). The tree T, of course, cannot yet be used to perform an efficient point location query, since a node v in T may have many children if its associated action forms a collection of horizontal cuts. To help deal with this issue we define the *weight* of a node $v \in T$ to be the number of leaf descendents of v in T, and we use $w(v)$ to denote this quantity. Given this weight function, then, we store the children of each node v in T as leaves in a biased finger tree T_v and doubly link all the leaves of T_v. Of course, such a biased finger tree is a trivial tree for each node v corresponding to a vertical cut, but this is okay, for it gives us a way to efficiently search the children of a node whose corresponding action is a collection of horizontal cuts.

The structure of T satisfies an invariant that if a face f spans a trapezoid, then either it has a spanning edge e of the subdivision on its top or bottom boundary or it is split into two by a pseudo-cut. In either case, the face f has a bounding spanning edge if it spans τ. We say that a face f or an edge e of the subdivision *covers* a trapezoid τ if it spans τ horizontally and it does not span any ancestor of τ in T. The structure of T has the property that any face or edge covers

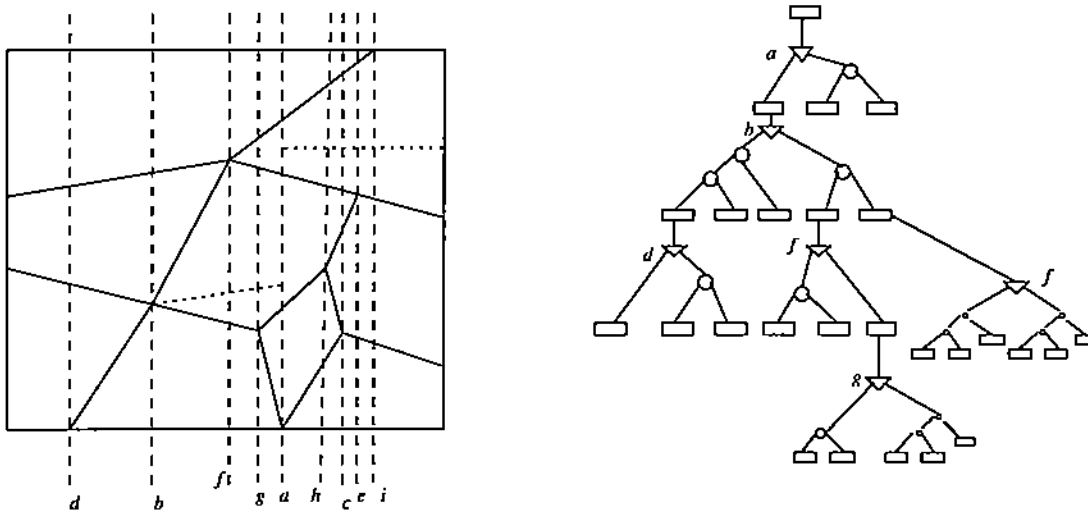


Figure 10: Trapezoidal Decomposition of a Convex Subdivision with Pseudo-Edge Cuts. The triangular nodes denote vertical cuts and the circular nodes denote horizontal cuts.

at most $O(\log n)$ trapezoids and also each face or edge covers at most two nodes at any level of T. These properties follow easily from segment tree like arguments [30].

Along with the tree T we also maintain a DCEL [35] (doubly connected edge list) for the subdivision. We maintain the faces of the subdivision as a dictionary $\mathcal{D}$ of size $O(n)$. With each face f (which is a node in $\mathcal{D}$), we maintain a dictionary $\Delta(f)$, implemented as a linked list, of size $O(\log n)$ containing pointers to the nodes in T that f covers. The ith item in the dictionary stores (at most two) pointers to the nodes in T that are covered by the face f at level i. We also make the items in $\Delta(f)$, for any face f doubly linked.

If we always access the pointers in Δ for any face in level order, then the time to access a single pointer is $\bar{O}(1)$, as we can search for the covered node at the highest level and then use links in the items of Δ to access the other pointers.

5.3 Query Algorithm

We now describe the point-location query algorithm for our data structure. Consider the operation $query(\tau, x, y)$, where x and y represent the coordinates of the query point and τ is a current trapezoid in the subdvision (which represents a node in our primary data structure). We alternately make comparisons with nodes in the primary and secondary data structures. In the primary data structure (triangular nodes representing trapezoids), we compare the x value of the point against the x value of the vertical cut at τ. This identifies the left or right secondary data structure of τ containing the query point. We then use the secondary data structure to identify a trapezoid containing the query point among the several trapezoids separated by horizontal cuts. In the secondary data structure i.e., in the biased finger tree stored in the trapezoid τ, we compare the (x, y) value of the point against the supporting line of the spanning edge or pseudo-cut [7] This

[7]When we use the query algorithm to locate edges, if the edge spans the trapezoid τ then we compare the y-value of the point of intersection of the edge with the left boundary of τ against the y-values of the points of intersections of the horizontal cuts with the left boundary of τ.

identifies a leaf node, say q, in the biased finger tree that represents a trapezoid, say τ_q, in the primary data structure. We now recursively locate the point by calling $query(\tau_q, x, y)$.

The arguments in the previous subsections on the structure of our data structure imply that, starting from the root of T, we can perform the point location query in $O(\log w(\tau) + depth(\tau))$ time in the worst case, where τ denotes the root of T. This is because the times to perform the biased merge tree queries down a path in T form a telescoping sum that is $O(\log w(\tau))$. Noting that $w(\tau)$ is $O(n \log n)$ [34] and $depth(\tau)$ is $O(\log n)$ (since our primary data structure is kept balanced) gives us the desired result that a point location query takes $O(\log n)$ time.

We now turn to the problem of implementing our update operations. We use T_τ to denote the biased finger tree attached to a trapezoid τ and we use ρ_p to denote the trapezoid associated with a leaf p of a biased finger tree. We use *left(τ)* and *right(τ)* to denote the left and right boundaries of a trapezoid τ.

5.4 Edge Insertion

Consider an edge insertion operation $insert(\tau, e, f, f_1, f_2, v_1, v_2, \Delta(f))$, where we insert an edge e with end points v_1 and v_2 into face f of the convex subdivision. The resulting two faces are f_1 and f_2. Let τ be the current trapezoid of T with a biased finger tree T_τ associated with it and let x be the vertical cut at τ. We say that an edge e or face f spans a trapezoid τ, if it intersects both *left(τ)* and *right(τ)*. We use the dictionary $\Delta(f)$ to efficiently access the nodes in our data structure, representing the pseudo-cuts or spanning edges in the trapezoids spanned by the face f. We also use $\Delta(f)$ to create new dictionaries $\Delta(f_1)$ and $\Delta(f_2)$ for the resulting faces f_1 and f_2. The insert operation is a recursive procedure which depends on the way in which e cuts τ. During the operation, we update our data structure dynamically in a way to keep it balanced, and also update the dictionary $\mathcal{D}$ and Δ's for faces f_1 and f_2. We have the following cases: (refer to Figure 11):

Case 1: The edge e does not span τ, and e does not intersect any pseudo-cut in τ. Also, suppose that both ends of e are located in a single leaf, say q, of T_τ. First, we increase the weight of node q in T_τ to account for the edge e. Let ρ_l and ρ_r be the left and right trapezoids of ρ_q separated by the vertical cut x. Let T_{ρ_l} and T_{ρ_r} be the corresponding trees. We note that the edge e cannot simultaneously span both ρ_l and ρ_r, as otherwise it spans τ (see Figure 11.1). If $\Delta(f)$ has a pointer to a node in T_τ, then we add that pointer to $\Delta(f_1)$ or $\Delta(f_2)$ depending on which face is closer to the spanning edge (can be easily checked). Now, we have the following subcases:

Case 1.1: The edge e intersects only ρ_l. Recursively call $insert(\rho_l, e, f, f_1, f_2, v_1, v_2, \Delta(f))$.

Case 1.2: The edge e intersects only ρ_r. Recursively call $insert(\rho_r, e, f, f_1, f_2, v_1, v_2, \Delta(f))$.

Case 1.3: The edge e intersects both ρ_l and ρ_r. Divide e into two segments e_1 and e_2 along x, and recursively call $insert(\rho_l, e_1, f, f_1, f_2, v_1, v^*, \Delta(f))$ and $insert(\rho_r, e_2, f, f_1, f_2, v^*, v_2, \Delta(f))$, where v^* is the point of intersection of e with x.

Analysis: These cases initiate the recursion. The cases 1.1 and 1.2 take $O(1)$ time (not counting recursive calls). The case 1.3 occurs at the first trapezoid which contains e on both sides of the vertical split and it occurs at most once during recursion. It also takes $O(1)$ time (not counting recursive calls).

Case 2: The edge e does not span τ, but e intersects a pseudo-cut in τ. This implies that the two end vertices of e are located in two adjacent leaves, say p and q, of T_τ. We search for one endpoint of e and access the node containing the other endpoint through the links in the

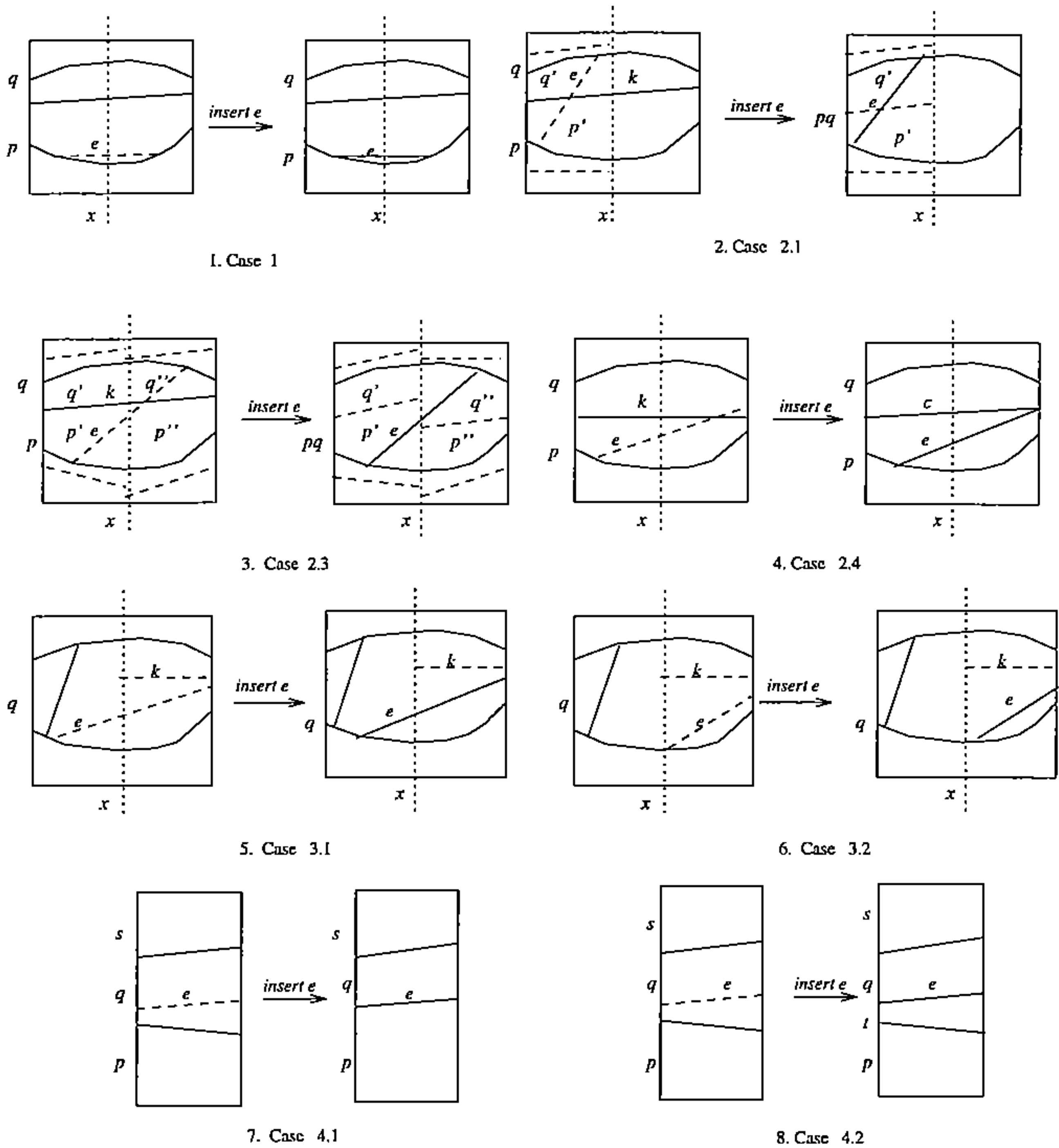


Figure 11: Insertion of Edge e into a Convex Subdivision.

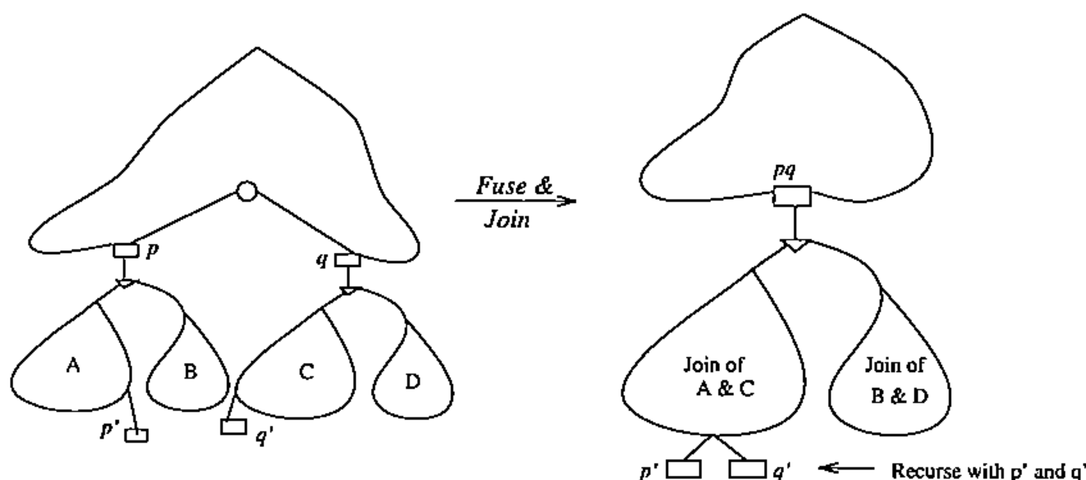


Figure 12: Insertion of Edge e into a Convex Subdivision (Case 2.1).

leaves. Now, the pseudo-cut between p and q, call it k, is no longer valid which leads to the following subcases:

Case 2.1: The edge e lies to the left of x, the vertical dividing line for τ. Fuse the two leaves p and q into a single leaf pq, and attach pq in place of parent of p and q in T_τ. Then, join the trees attached to p and q as shown in the Figure 12. Locate the leaves p' and q' containing the two vertices of e in T_p and T_q respectively, and then recurse on p' and q' (see Figure 11.2) i.e., fuse the nodes p' and q' and recurse on the trapezoid $\rho_{p'q'}$ with $insert(\rho_{p'q'}, e, f, f_1, f_2, v_1, v_2, \Delta(f))$. The faces f_1 and f_2 do not span τ, and hence no new pointers are added to $\Delta(f_1)$ and $\Delta(f_2)$.

Case 2.2: The edge e lies to the right of x. This case is symmetrical to above case.

Case 2.3: The edge e intersects both sides of x. We note that this case occurs at most once during the recursion, as afterwards we search for only one endpoint of the edge and the other endpoint always touches the left or right boundary of the enclosing trapezoid. The cut k between the leaves p and q is no longer valid, and also both f_1 and f_2 do not span τ. So, fuse the two leaves into a single leaf pq, and attach pq in place of parent of p and q in T_τ. Then, join the trees connected to p and q as shown in Figure 13. Locate the leaves p' and q' in left subtrees of T_p and T_q containing the horizontal cut between p and q, and similarly, locate the leaves p'' and q'' in right subtrees of T_p and T_q containing the horizontal cut between p and q. Fuse the leaves p' and q' into a single leaf $p'q'$, and similarly fuse the leaves p'' and q'' into a single leaf $p''q''$. Divide e into two segments e_1 and e_2 along vertical cut x of τ, and recurse on $\rho_{p'q'}$ with $insert(\rho_{p'q'}, e_1, f, f_1, f_2, v_1, v^*, \Delta(f))$, where v^* is the point of intersection of e with x (see Figure 11.3). Also, recurse on $\rho_{p''q''}$ with $insert(\rho_{p''q''}, e_2, f, f_1, f_2, v^*, v_2, \Delta(f))$. The faces f_1 and f_2 do not span τ, and hence no new pointers are added to $\Delta(f_1)$ and $\Delta(f_2)$.

Case 2.4: The edge e crosses the pseudo-cut k between p and q, but unlike case 2.3, one of the resulting faces, say f_1, spans τ (see Figure 11.4). Using the information in Δ for face f about the pseudo-cut k and the coordinates of endpoints of e, we can easily compute a new pseudo-cut, say c, for face f_1. Replace the old pseudo-cut k with the new cut c.

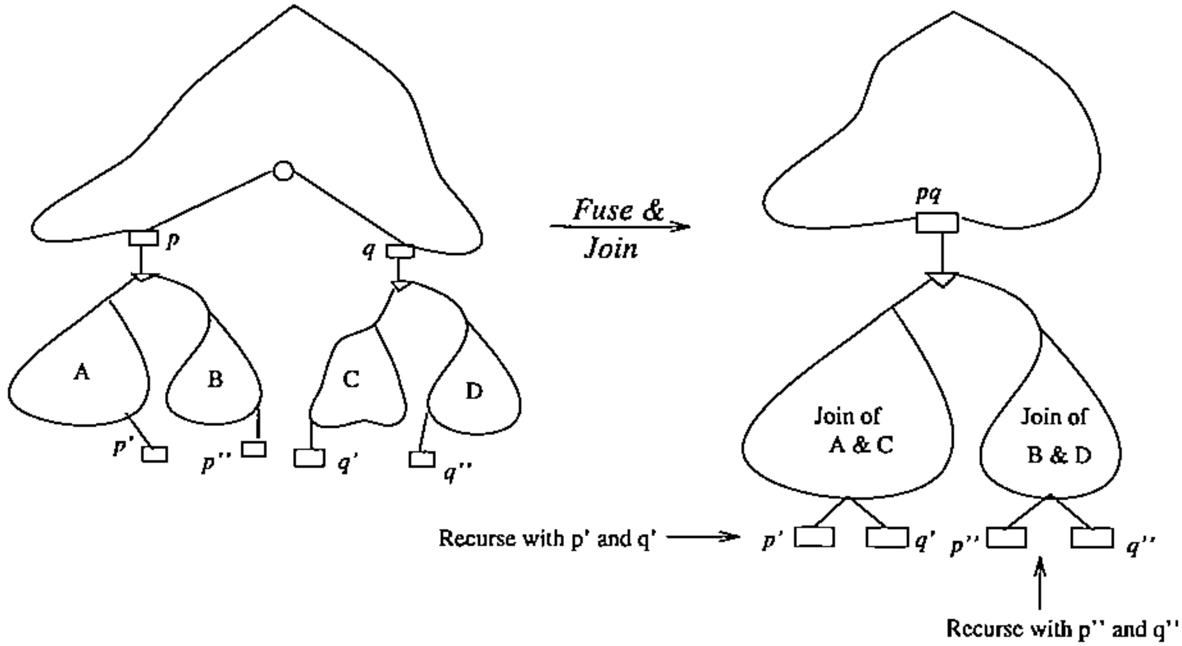


Figure 13: Insertion of Edge e into a Convex Subdivision (Case 2.3).

We add a pointer in $\Delta(f_1)$ to the node representing the new pseudo-cut c in T_τ. Recurse on the left and right with appropriate segments of e.

Analysis: We show that for each of the cases (except 2.3), we recurse on one of the trees of τ and all the operations performed in the other tree require only $\bar{O}(1)$ time. First case 2.1. For this case, we perform two operations i.e., fusing of two leaves p and q of T_τ, and then the join of the trees attached to p and q as shown in Figure 12. The join of two biased finger trees takes $\bar{O}(1)$ time. We now show that the fusing of the two leaves p' and q' also takes $\bar{O}(1)$ time. Now, the leaves p' and q' are not part of the leftmost or rightmost root-to-leaf path. This is crucial for our $\bar{O}(1)$ time fusing algorithm (see the case analysis of fuse operation in section 2.2). Suppose $w_{p'} \geq w_{q'}$. Now, the rank of $p'q'$ is at most $r(p') + 1$. So, change the rank of p' according to the sum of the weights of p' and q' i.e., $\log(w_{p'} + w_{q'})$. This is equivalent to promotion operation and it takes $\bar{O}(1)$ time. We need to delete q' which takes $\bar{O}(1)$ time as q' is not on the either of the root-to-leaf paths (see Corollary 2.3). The case 2.3 occurs in the first trapezoid which contains e on both sides of its vertical cut and it occurs at most once. Again fusing and join takes $\bar{O}(1)$ time. Since the face f spans the trapezoid τ, the pointers to nodes p and q can be accessed on $O(1)$ time from $\Delta(f)$. The leaf nodes p', q', p'' and q'' occur along the leftmost or rightmost root-to-leaf path (see Figure 13). Hence, they can be accessed in $O(1)$ time using *left* or *right* values in the roots of the trees. The case 2.4 involves computation of a new pseudo-cut which can be done in $O(1)$ time from the information about old cut k and the intersection point of e with the $right(\tau)$ (or $left(\tau)$). So, the overall complexity of this case $\bar{O}(1)$ (not counting recursive calls).

Case 3: The edge e spans either *left(τ)* or *right(τ)*, but not both and also e does not intersect any pseudo-cut for τ. This implies that only one end of e is located in a single leaf, say q, of T_τ (see Figures 11.5 and 11.6). First, increase the weight of node q in T_τ to account for the

edge e. Let ρ_l and ρ_r be the left and right trapezoids of ρ_q separated by x. Let T_{ρ_l} and T_{ρ_r} be the corresponding trees. If either f_1 or f_2 spans τ, then we add new pointers to $\Delta(f_1)$ or $\Delta(f_2)$, as in case 1 to the node representing the spanning edge for f_1 (which can be obtained from $\Delta(f)$). We have the following cases:

case 3.1. The edge e spans ρ_l (see Figures 11.5). The other case is symmetrical. Divide e into two segments $e_1 = v_1v^*$ and $e_2 = v^*v_2$, where v^* is the intersection of e with the vertical cut x. Now, recurse on ρ_l and ρ_r with e_1 and e_2 respectively. Note that, using the pointer in Δ for face f (call it *sibling-pointer*), we can access in $\bar{O}(1)$ time the leaf in T_{ρ_r} containing the segment e_2, since f spans ρ_r. We note that the recursive call on ρ_r with e_2 falls under case 4 and terminates immediately.

Case 3.2. The edge e does not span both trapezoids. Suppose it lies entirely in ρ_r (see Figure 11.6). The other case is symmetrical. Now, recursively call $insert(\rho_r, e, f, f_1, f_2, v_1, v_2, \Delta(f))$.

Analysis: This case takes $\log w_\tau / w_q$ time to access q. All other operations take $\bar{O}(1)$ time (not counting recursive calls).

Case 4: The edge e spans the trapezoid τ. By our invariant there must already be a spanning edge or pseudo-cut immediately below (or above) e. This case results as a result of recursive calls made in one of the previous cases, as we are not searching for any end points of e. We directly use pointers in $\Delta(f)$ (or *sibling-pointer* as mentioned in the case 3.1) to locate e in one of the leaves, say q, of T_τ. Let p and s be the predecessor and successor nodes of node q respectively.

Case 4.1: The face f intersects the trapezoid ρ_p (see Figure 11.7). Now the test at the least common ancestor of p and q, say q_1, in T_τ represents a pseudo-cut. Change this test to e. There is a symmetrical case if the face f intersects the trapezoid ρ_s. We include a pointer to the node q_1 in the dictionaries $\Delta(f_1)$ and $\Delta(f_2)$.

Case 4.2: The face f does not intersect both ρ_p and ρ_s. By our invariant, e must be adjacent to a spanning boundary edge of f on top or bottom of q (see Figure 11.8). Now slice q into two leaves, say q and an empty leaf t, and attach them as children to a new node q_1 in T_τ. Introduce the test e at q_1. The slicing of node q takes $\bar{O}(1)$, since t is an empty leaf i.e., $x \approx 1.0$ (see Corollary 2.4). We insert pointers in $\Delta(f_2)$ to nodes q_1 and the node representing the test e in T_τ. Also we insert pointers in $\Delta(f_1)$ to node q_1.

Analysis: These two cases take $\bar{O}(1)$ time, given the pointers to the leaf q and the nodes at which the horizontal tests are performed, in τ. We obtain these pointers from Δ for face f in $\bar{O}(1)$ time.

Analysis: The depth of recursion is $O(\log w(r))$, where r is the root of T, and each case except case 3 takes $\bar{O}(1)$ time (not counting the recursive calls). The complexity of case 3 telescopes to give us an overall complexity of $\bar{O}(\log w(r))$, which is bounded by $\bar{O}(\log n)$, for insert operation. The insert operation creates two new faces f_1 and f_2 by splitting an old face f. We include the new faces f_1 and f_2, and delete the face f from the DCEL and dictionary $\mathcal{D}$. This can be done in $O(\log n)$ time. We detailed the updating the dictionaries Δ's for f_1 and f_2 in each of the cases. So, the overall complexity of insert operation is $\bar{O}(\log n)$.

5.5 Edge Deletion

Consider an edge deletion operation $delete(\tau, e, f, f_1, f_2, v_1, v_2, \Delta(f_1), \Delta(f_2))$, where we delete an edge e with end points v_1 and v_2 separating faces f_1 and f_2 from the convex subdivision. Let the resulting face be f. Let τ be the current trapezoid of T with a biased finger tree T_τ associated with it. The dictionaries $\Delta(f_1)$ and $\Delta(f_2))$ are used to efficiently access pointers to nodes in our data structure, representing the tests at spanning edges or pseudo-cuts in the trapezoids spanned by these two faces. We also use these two dictionaries to create $\Delta(f)$, the dictionary for the resulting face f. Let x be the vertical cut of τ. The deletion operation is a recursive procedure, and it depends on the way e cuts τ. We have the following cases[8] (see Figure 14):

Case 1: The edge e does not span τ, and one of the faces f_1 or f_2 spans τ (see Figures 14.1 and 14.2). We have the following two subcases:

Case 1.1: Both the endpoints of edge e lie in τ (see Figure 14.1). In this case locate e in a leaf node, say q, of T_τ, and decrease the weight of node q by 1 in T_τ. If e intersects both sides of vertical cut x, then split e into two segments, say e_1 and e_2, at the vertical cut, and recurse on both sides of τ with appropriate segments. Otherwise, recurse on one side of τ with e. Suppose the face f_1 spans τ. Then, copy the pointer to the node representing the pseudo-cut of face f_1, in T_τ to dictionary $\Delta(f)$ from $\Delta(f_1)$.

Case 1.2: Only one endpoint of edge e lies in τ, say v_1 (see Figure 14.2). The other case is symmetrical. In this case locate e in a leaf node, say q, of T_τ, and decrease the weight of node q by 1 in T_τ. If e intersects both sides of vertical cut x, then split e into two segments, say e_1 and e_2, at the vertical cut, and recurse on both sides of τ (i.e., ρ_{q_l} and ρ_{q_r}) with e_1 and e_2 respectively. Since, e_2 spans ρ_{q_r}, the node containing e_2 occurs as rightmost leaf in $T_{\rho_{q_r}}$, and its pointer is obtained in $O(1)$ time using the *right* value at the root. Hence, the recursive call with e_2 as parameter terminates in $\bar{O}(1)$ time (it falls under case 3 to be discussed later), If e intersects only one side of τ, then recurse with e on the appropriate side. We update the pointers in $\Delta(f)$ as in the previous case.

Analysis: The time needed to locate q is $O(\log w_\tau/w_q)$. All other operations take $\bar{O}(1)$ time (not counting recursive calls). So, the total time required for this case is $\bar{O}(\log w_\tau/w_q)$.

Case 2: The edge e does not span τ, and both faces f_1 and f_2 do not span τ. We have the following subcases:

Case 2.1: Suppose after deleting e, the resulting face f does not span τ. Locate e in a leaf node, say q, of T_τ. Now, decrease the weight of q by 1 (as one edge is deleted) in T_τ. If e intersects both sides of vertical cut x, then recurse on both sides of τ with appropriate segments of e. Otherwise recurse on one side of τ with e (see Figure 14.3). No new pointers are added to $\Delta(f)$, as faces f_1 and f_2 do not span τ.

Analysis: The time needed to locate q is $O(\log w_\tau/w_q)$. All other operations take $\bar{O}(1)$ time. So, the total time required for this case is $\bar{O}(\log w_\tau/w_q)$.

Case 2.2: Suppose after deleting e, the resulting face f spans τ. Since f spans τ after deleting e, we need to introduce a new pseudo-cut in τ. Let c be the new pseudo-cut. The computation of pseudo-cut c is rather involved, and we describe the details after discussing the subcases. The operations for this case, depends on how the edge e intersects the vertical cut. We have the following subcases:

[8] We have ignored some of the symmetrical cases to simplify the exposition.

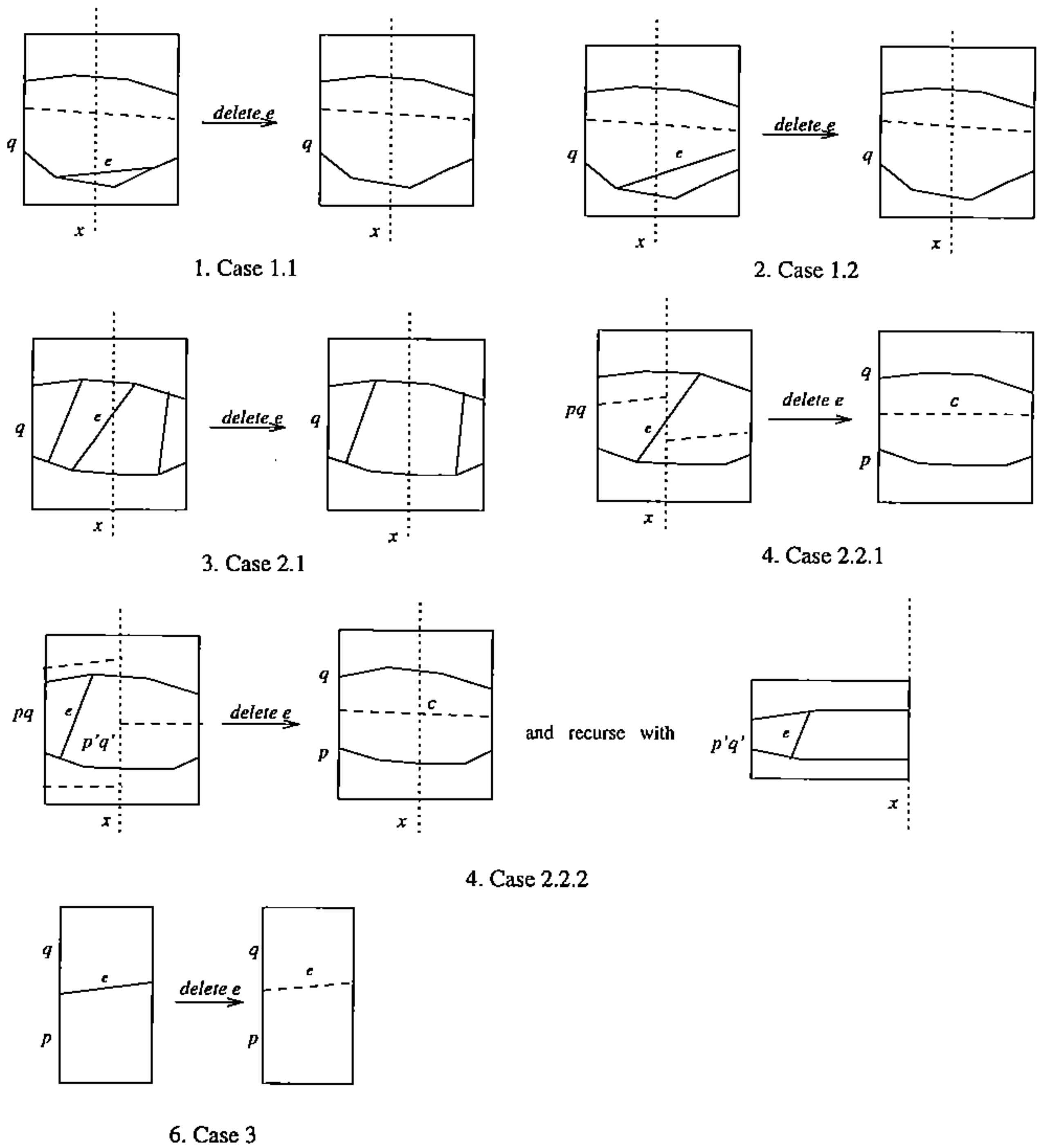


Figure 14: Deletion of Edge e from a Convex Subdivision.

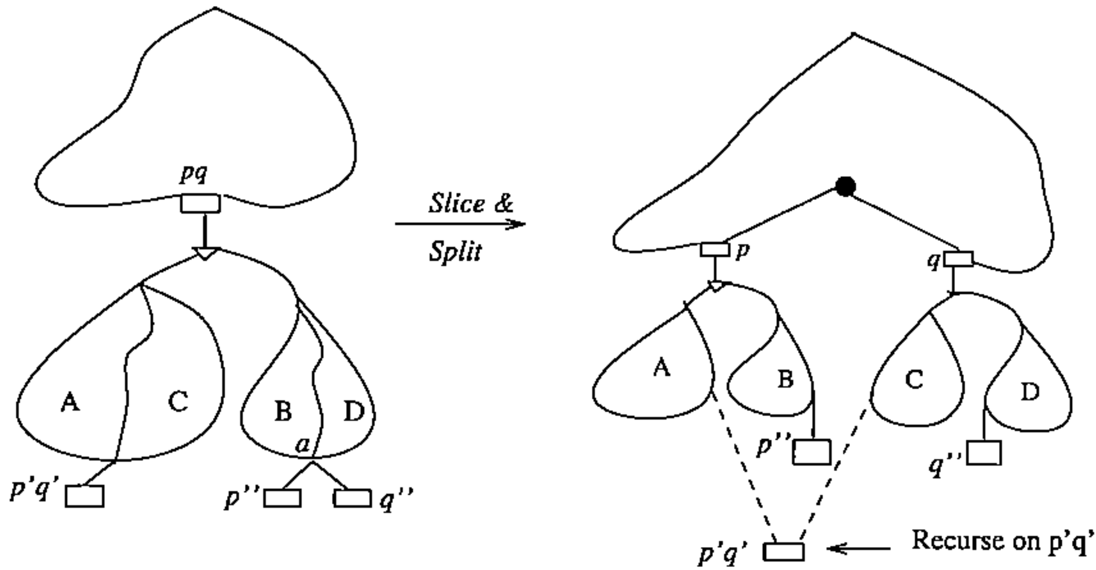


Figure 15: Deletion of Edge e from a Convex Subdivision.

Case 2.2.1: The edge e intersects both sides of x. This is a termination case. There must be a spanning edge or pseudo-cut around e in trapezoids ρ_l and ρ_r, as both faces f_1 and f_2 intersect the two trapezoids (see Figure 14.4). Slice pq into two nodes p and q with a new node q_1 as parent, and introduce a new test c on q_1; attach q_1 in place of pq in T_τ. Since, both f_1 and f_2 span the left and right sides of τ respectively, there must be pseudo-cuts in the two trapezoids ρ_{q_l} and ρ_{q_r}, along the cut c in the higher level. So, split the trees connected to pq along those pseudo-cuts, and then join the two left and two right trees separately as shown in Figure 15.
Analysis: The time needed to locate pq is $O(\log w_\tau/w_{pq})$. We also perform a slice and two split operations in this case. The complexity of split operations dominates the cost of slice operation, and it is bounded by $\bar{O}(\log w_{pq}/w_t)$, where t is the (deeper) node in $T_{\rho_{pq_l}}$ or $T_{\rho_{pq_r}}$ containing the test for pseudo-cut along c.

Case 2.2.2: The edge e does not intersect both sides of x. Suppose e lies to the left of x. The other case is symmetrical. This is the most complicated case, as deletion of edge e introduces a new pseudo-cut for face f in τ at the first invocation of this case, and we again recurse on the same case. Moreover, we show that this case incurs logarithmic cost, at each recursive step. Slice pq into two nodes p and q with a new node q_1 as parent, and attach q_1 in place of pq in T_τ. Since the face f_1 does not span the left side of τ, there exits a node $p'q'$ in T_{pq_l}, similar to pq in T_τ, containing the edge e. So, split T_{pq_l} at $p'q'$; join the two resultant trees as left subtrees of nodes p and q, respectively, and recurse on $p'q'$ with e (see Figure 14.5) Since the face f_2 spans the right side of trapezoid τ, there exists nodes p'' and q'' in tree $T_{\rho_{pq}}$, with a least common ancestor a, and containing the test for pseudo-cut along the pseudo-cut c, between p and q. Hence, split $T_{\rho_{pq}}$ at a, and join the resultant trees as right subtrees to the nodes p and q, respectively (see Figure 15).
Analysis: The time needed to locate node pq is $O(\log w_\tau/w_{pq})$. The search operation for nodes pq, $p'q'$, etc., telescope during recursion yielding an overall complexity of $\bar{O}(\log w(\tau) + depth(\tau))$, as in query algorithm, where τ is the root of T. We can

bound this complexity by $\bar{O}(\log n)$. We also perform a slice and two split operations in this case. The complexities of split operations dominate the cost of slice operation. The complexity of one split operation along the path of recursion is $\bar{O}(\log w_{pq}/w_{p'q'})$. Since, we recurse on the node $p'q'$, these complexties telescope yielding, again, an overall complexity of $\bar{O}(\log w(\tau))$, which is bounded by $\bar{O}(\log n)$. But the split operation on the right tree of $T_{\rho_{pq}}$, which is not on the search path (i.e., split on the tree BD in the Figure 15), takes time $\bar{O}(\log w_{pq}/w_a)$, where a is the least common ancestor of the nodes p'' and q'' in $T_{\rho_{pq}}$ (see Figure 15). But, w_a may have no relationship with $w_{p'q'}$. So, the complexities of these split operations do not telescope. This results in a logarithmic overhead at each step of the recursion, which in the worst case could be, as high as $\bar{O}(\log w(\tau))$, which in turn is bounded by $\bar{O}(\log n)$. So, this case takes (not counting recursive calls) $\bar{O}(\log n)$ time.

We now describe the details of computation of pseudo-cut c, and how to update $\Delta(f)$, for case 2.2. It is easy to compute the pseudo-cut c for face f in case 2.2.1, from $\Delta(f_1)$ and $\Delta(f_2)$. But in case 2.2.2, the face f_1 does not span the left side of τ, and we have information only about pseudo-cut for face f_2. Moreover, from our invariant, we need to introduce the pseudo-cut for face f, only at the highest level where case 2.2.2 is invoked. We observe that, the case 2.2.1 occurs either directly, or as a termination case of recursive calls made in case 2.2.2. If case 2.2.1 arises as a result of recursive calls made in case 2.2.2, compute the pseudo-cut for face f, and pass the result up the recursive path. The pseudo-cut for face f_2 on the right side of τ, is available at each recursive invocation of case 2.2.2. By combining this information with the pseudo-cut for face f_1 obtained recursively, compute the pseudo-cut c for face f.

Case 3: The edge e spans τ. Suppose the edge e separates the nodes p and q in τ. This case results from recursive calls made from one of our previous cases, specifically 3.2. In this case there must be a real split at e in τ, already. Change this real split to a pseudo split (see Figure 14.6). Fuse the node q with p if either of them is empty. Since, resulting face f spans τ, we include a pointer in $\Delta(f)$, to the node representing the pseudo-cut in T_τ, which can be obtained from either $\Delta(f_1)$ or $\Delta(f_2)$.

Analysis: This case takes $\bar{O}(1)$ time as pointers to nodes q, p and the node containing the test e are obtained from the *left* or *right* pointers in the root, as discussed in case 3.2.

Analysis: The complexity of recursion (not counting time for each case) is proportional to the depth of the primary tree, which is bounded by $O(\log n)$. During deletion, the searches and split operations along the path of recursion telescope, yielding a time complexity of $\bar{O}(\log w(\tau))$, which is bounded by $\bar{O}(\log n)$. The cost of case 2.2.2 dominates the costs of all other cases, which in the worst-case takes $\bar{O}(\log n)$ time. This leads to a $\bar{O}(\log^2 n)$ time algorithm for deletion.

We say that an edge e is *properly covered*, if during deletion of e, it does not introduce any new pseudo-cut into the subdivision. We observe that all the edges in a covex subdivision which are surrounded by "long" edges above or below are properly covered.

Corollary 5.1: *The deletion of a properly covered edge e from the convex subdivision takes $\bar{O}(\log n)$ time.*

Proof: The recursive call for deletion of a properly covered edge never invokes cases 2.2.1 and 2.2.2. All other cases of deletion, including the recursive calls can be bounded within $\bar{O}(\log n)$ time. ■

5.6 Rebalancing the Primary Structure

In this section we show how to relax the constraint that the endpoints have x coordinates in the range $[1, n]$.

We now use a $BB[\alpha]$-tree as a primary tree for vertical cuts with biased finger tree as a secondary structure in each node. We briefly review the properties of $BB[\alpha]$-tree. Let $f(l)$ denote the time to update the secondary structures after a rotation at a node whose subtree has l leaves. Also, assume that we perform a sequence of n update operations, each an insertion or a deletion, into an initially empty $BB[\alpha]$-tree. Now, we have the following times for rebalancing [29]:

- If $f(l) = O(l \log^c l)$, with $c \geq 0$, then the rebalancing time for an update operation is $\bar{O}(\log^{c+1} n)$.
- If $f(l) = O(l^a)$, with $a < 1$, then the rebalancing time for an update operation is $\bar{O}(1)$.

In our case, we show $f(l) = O(n)$ and so the rebalancing cost is $\bar{O}(\log n)$. The figures 16 and 17 outline the intuition behind this claim. Consider left rotation in a subtree rooted at node x in the $BB[\alpha]$-tree at level i. Suppose right child of x is y. After rotation, y becomes the root of the subtree with left child x. We observe that the structure of the trapezoids which are descendants of x and y are not affected by rotation. The structure of trapezoids at level i gets affected due to rotation.

After rotation, we need to recompute the structure of the secondary structures at x and y. The right subtree of y is constructed by joining all the right subtrees of y nodes of all leaves of T_x into a single tree to give the right subtree for y. These joins take $O(k)$ time, where k is the number of joins. We construct the left subtree of y and the secondary structure of x nodes as follows: Collect the leaves of left subtree of x and the leaves in the left subtree of y nodes of all leaves of T_x in two separate arrays, say A_1 and A_2 respectively, in left to right order. Traverse the trapezoids in A_1 in order and for each trapezoid check whether its top boundary extends up to y. If not, continue the check with the next trapezoids in A_1. Let τ be the last trapezoid which satisfied the check. Whenever a trapezoid τ_1 satisfies the condition, collapse all the trapezoids from τ to τ_1 into a single trapezoid τ_2, which is a new leaf for the left subtree of y. By traversing the array A_2 also simultaneously, the secondary structure for τ_2 can be computed easily. The left subtree of τ_2 consists of trapezoids from τ to τ_1. Similarly the right subtree consists of the corresponding trapezoids in A_2. All the above operations take time proportional to the number of descendants of x and hence is linear.

5.7 A $\bar{O}(\log n)$ Update Method for Rectilinear Subdivision

We construct a $O(\log n)$ depth trapezoidal structure for rectilinear subsubdivision similar to the one for convex subdivision to efficiently perform the update operations arising in layers-of-maxima algorithm (see section 4). This structure satisfies a slightly different invariant. Here we use an invariant that if a face f spans a trapezoid τ and f can be split into two faces by a (truly) horizontal spanning segment, say e, of τ, then we split τ into two trapezoids using e. So, unlike convex subdivision, not every spanning face of τ will have a pseudo-cut (see Figures 18.a and 18.b) here. The intuition being in the layers-of-maxima algorithm we always insert horizontal edges and pseudo-cut property is required only for those type of edges we insert. From the invariant, it follows that the insert operation during the computation of layer numbers takes $\bar{O}(\log n)$ time.

The deletion of an arbitrary edge in rectilinear subdivision still takes $\bar{O}(\log^2 n)$ time, however, but the edges which we delete in the layers-of-maxima algorithm (refer step 4) are not arbitrary. We now show that they satisfy the proper cover property as required by Corollary 5.1 (see Figure 8).

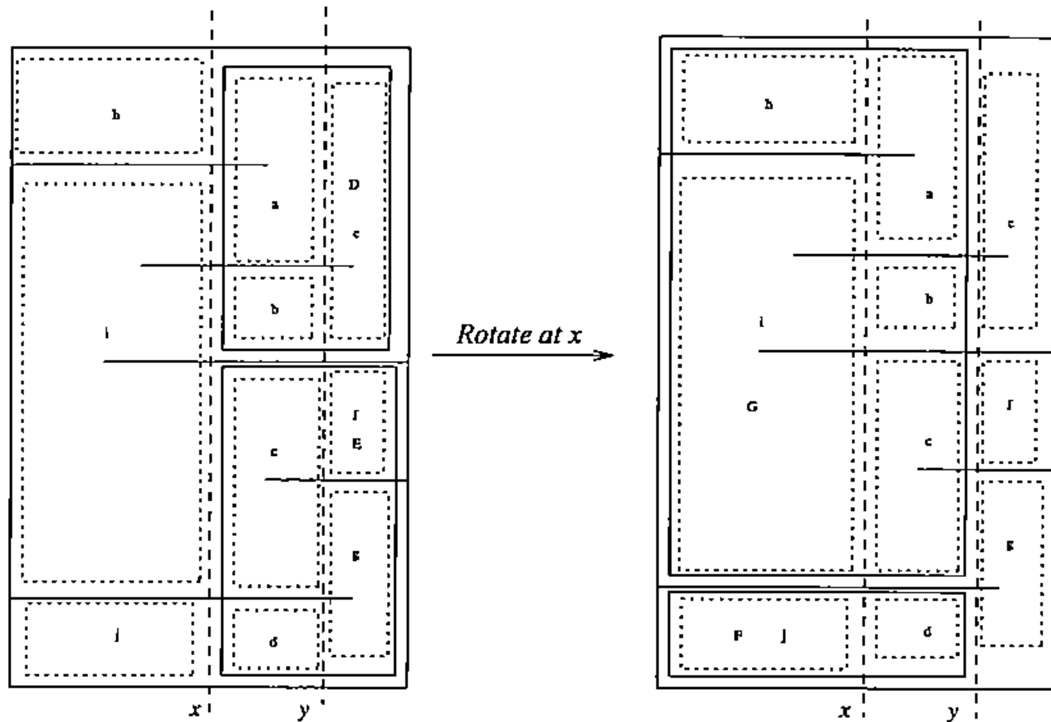


Figure 16: Rotation to Rebalance the $BB[\alpha]$ tree.

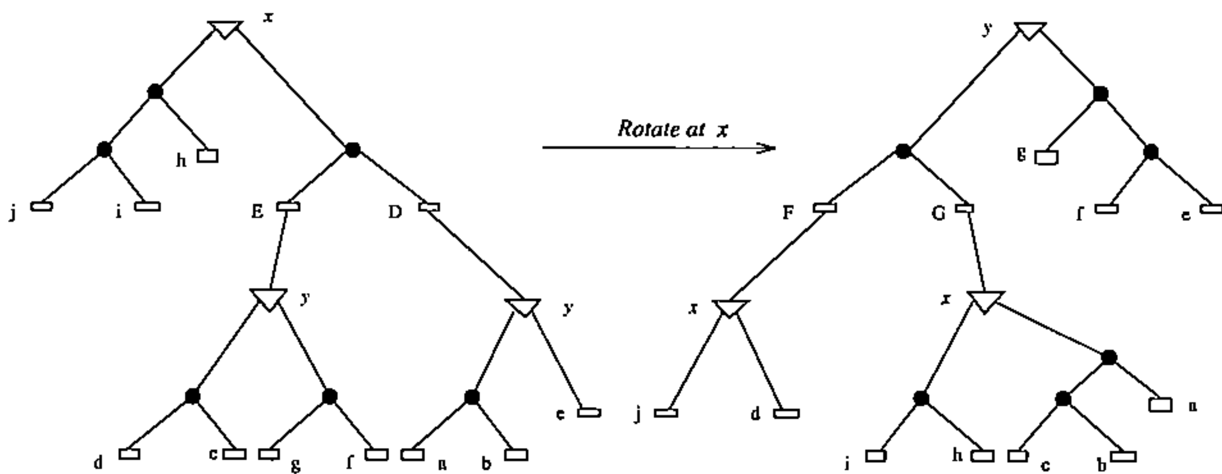
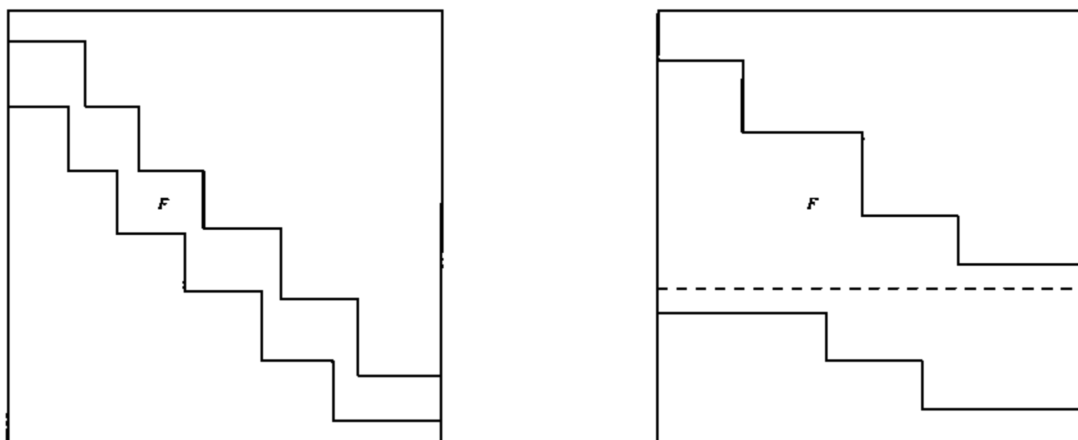


Figure 17: Effect of Rotation of Primary tree (see Figure 15) on Secondary Tree.



a. A Spanning Face without any Pseudo-cut. b. A Spanning Face with a Pseudo-cut.

Figure 18: Pseudo-Cuts for Rectilinear Spanning Face.

Each edge deleted is bounded on the top and on the right by the "long" lines h and v from the new point p. So, the edges deleted in the algorithm, and the edges h and v belong to the same face. This implies that we already have a horizontal spanning pseudo edge for the face resulting after deletion, and hence deletion of an edge does not require an introduction of a new pseudo-cut into the subdivision. Hence, according to the terminology introduced in section 5.5, the edges we delete are properly covered. Hence, by Corollary 5.1, we conclude that the deletion of each edge in our layers-of-maxima algorithm, costs $\bar{O}(\log n)$ time.

References

[1] N. Abramson, *Information Theory and Coding*, McGraw-Hill, New York, 1963.

[2] P. Agarwal, private communication, 1992.

[3] A. Aggarwal and J. Park, "Notes on searching in multidimensional monotone arrays," in *Proc. 29th IEEE Symposium on Foundations of Computer Science*, 1988, 497–512.

[4] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley (Reading, Mass.: 1983).

[5] H. Baumgarten, H. Jung, and K. Mehlhorn, "Dynamic Point Location in General Subdivisions," *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, 1992, 250–258.

[6] S.W. Bent, D.D. Sleator, and R.E. Tarjan, "Biased search trees," *SIAM Journal of Computing*, 14(3):545–568, August 1985.

[7] N. Blum and K. Mehlhorn, "On the Average Number of Rebalancing Operations in Weight-Balanced Trees," *Theoretical Computer Science*, **11**, 1980, 303–320.

[8] B. Chazelle, "On the convex layers of a planar set," *IEEE Trans. Inform. Theory*, **IT-31** 1985, 509–517.

[9] Y.-J. Chiang and R. Tamassia, "Dynamization of the Trapezoid Method for Planar Point Location," *Proc. ACM Symp. on Computational Geometry*, 1991, 61–70.

[10] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," *Proc. of the IEEE*, 80(9), 1992.

[11] S.W. Cheng and R. Janardan, "New Results on Dynamic Planar Point Location," *31st IEEE Symp. on Foundations of Computer Science*, 96–105, 1990.

[12] R.F. Cohen and R. Tamassia, "Dynamic Expression Trees and their Applications," *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, 1991, 52–61.

[13] R. Cole, "Searching and Storing Similar Lists," *J. of Algorithms*, Vol. 7, 202–220 (1986).

[14] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press (Cambridge, Mass.: 1990).

[15] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, NY, 1987.

[16] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing*, Vol. 15, No. 2, 317–340, 1986.

[17] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph," *J. Algorithms*, **13**, 1992, 33–54.

[18] M.T. Goodrich and R. Tamassia, "Dynamic Trees and Dynamic Point Location," *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, 523–533.

[19] L.J. Guibas, E.M. McCreight, M.F. Plass, and J.R. Roberts, "A New Representation for Linear Lists," *Proc. 9th ACM Symp. on Theory of Computing*, 1977, 49–60.

[20] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science*, 1978, 8–21.

[21] S. Huddleston and J. Mehlhorn, "A New Data Structure for Representing Sorted Lists," *Acta Informatica*, **17**, 1982, 157–184.

[22] D. Kirkpatrick, "Optimal Search in Planar Subdivision," *SIAM Journal on Computing*, Vol. 12, No. 1, February 1983, pp. 28–35.

[23] D.E. Knuth, *Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[24] S.R. Kosaraju, "Localized Search in Sorted Lists," in *Proc. 13th Annual ACM Symp. on Theory of Computing*, 1981, pp. 62-69.

[25] S.R. Kosaraju, Personal communication, 1993.

[26] H.T. Kung, F. Luccio, F.P. Preparata, "On Finding the Maxima of a Set of Vectors," *J. ACM*, Vol. 22, No. 4, 1975, pp. 469–476.

[27] D.T. Lee and F.P. Preparata, "Location of a Point in a Planar Subdivision and its Applications," *SIAM J. Computing*, Vol. 6, No. 3, 594–606, 1977.

[28] D.T. Lee and F.P. Preparata, "Computational Geometry—A Survey," *IEEE Trans. on Computers*, C-33(12), 1984, 872–1101.

[29] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, 1984.

[30] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, 1984.

[31] K. Mehlhorn and S. Näher, "Dynamic Fractional Cascading," *Algorithmica*, **5**, 1990, 215–241.

[32] I. Nievergelt and E.M. Reingold, "Binary Search Trees of Bounded Balance," *SIAM J. Comput.*, **2**, 1973, 33–43.

[33] M. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Springer-Verlag, 1983.

[34] F.P. Preparata, "A New Approach to Planar Point Location," *SIAM J. Computing*, Vol. 10, No. 3, 1981, 73–83.

[35] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, NY, 1985.

[36] F.P. Preparata and R. Tamassia, "Fully Dynamic Point Location in a Monotone Subdivision," *SIAM J. Computing*, Vol. 18, No. 4, 811–830, 1989.

[37] F.P. Preparata and R. Tamassia, "Efficient Spatial Point Location," Algorithms and Data Structures (Proc. WADS '89), 3–11, Lecture Notes in Computer Science, Vol. 382, Springer-Verlag, 1989.

[38] Preparata, F.P. and R. Tamassia, "Efficient Point Location in a Convex Spatial Cell-Complex," *SIAM J. Comput.*, 21, 1992, 267–280.

[39] Preparata, F.P. and R. Tamassia, "Dynamic Planar Point Location with Optimal Query Time," *Theoretical Computer Science*, Vol. 74, No. 1, 95–114, 1990.

[40] N. Sarnak and R.E. Tarjan, "Planar Point Location Using Persistent Search Trees," *Communications ACM*, Vol. 29, No. 7, 669–679, 1986.

[41] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Comput. and Sys. Sci.*, 26, 362–391, 1983.

[42] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.

[43] R.E. Tarjan, "Amortized computational complexity," *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, April 1985.