# Biased Finger Trees and Three-Dimensional Layers of Maxima

(Preliminary Version)

MIKHAIL J. ATALLAH*
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-1398, USA
E-mail: mja@cs.purdue.edu

MICHAEL T. GOODRICH†
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218, USA
E-mail: goodrich@cs.jhu.edu

KUMAR RAMAIYER‡
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218, USA
E-mail: 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 [14]. 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 [36]). In many dynamic computational geometry algorithms they may even be found as tertiary structures, e.g., Goodrich and Tamassia [19].

*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.



10th Computational Geometry 94-6/94 Stony Brook, NY, USA
© 1994 ACM 0-89791-648-4/94/0006..$3.50

### 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" [20, 25, 22], directly into the part of $T$ being changed. Such a pointer could come, for example, from a query in some auxiliary data structure. This may, in fact, have been a prime motivating factor behind the method of Huddleston and Mehlhorn [22] 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 is a positive integer that may be proportional to an access probability, as in an optimal binary search tree structure [3, 24]. Or it may represent the size of some auxiliary structure associated with item $i$, as in a link-cut structure [40] (which itself has many applications [12, 17, 18]) or in a point location structure built using the trapezoid method [8, 35, 38]. In cases with weighted items such as these 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 [5] 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 [5]. In fact, if each item $i$ has weight $w_i = 1$, then our structure is topologically equivalent to a red-black tree [14, 21, 41]. 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 the well-known 3-dimensional layers-of-maxima problem [2, 7, 15, 27]. We also give improved methods for dynamic point location in a convex subdivision [35, 8], and present a method for dynamic point location in staircase subdivision with logarithmic query and update times. We note that although the staircase subdivision we consider is a very restricted form of subdivision, it is the only subdivision we know of for which there is a method achieving logarithmic query and update times. In addition, the optimal algorithm for three-dimensional layers of maxima problem uses our dynamic data structure for staircase subdivision.

# 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 [3] 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$(i, T, x, i_1, i_2)$ : Break 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 $0 < x < 1$, and $i^-$ and $i^+$ are predecessor and successor items of $i$ in $T$ respectively.

Fuse$(i_1, i_2, i, T)$ : Combine 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 $T$. The items $i_1$ and $i_2$ need not be siblings, but should be adjacent in the total order.

As mentioned above, our structure is topologically similar to the biased search tree[1] of Bent, Sleator and Tarjan [5]. 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 items, *left* and *right*, which are pointers to the largest item in the left subtree and the smallest item 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 non-negative integer rank $r(x)$ that satisfies the natural extensions of red-black tree rank [41] to weighted sets [5]:

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 that a node be minor if and only if its sibling or a child*

[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.*

| *Update Operation* | *Previous Biased Trees [5]* | *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 $i$. All the complexities for update operations only count the time to perform the update, and do not include search times.

*of its sibling is a major leaf* [5]. We refer to this as the *bias* property.

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 [42] to analyze the rebalancing and update operations. With each node $x$ of a biased finger tree we associate[2] a value, $C(x)$, of "credits", with $0 \leq C(x)$. Moreover, we partition these credits into three types—one type that is similar to those used in the analysis of Bent, Sleator, and Tarjan [5], one type that assigns 1 credit to the nodes on the spine of the tree, and one type suggested by Kosaraju [26]. We omit details here.

After an update operation, we perform 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 Tarjan [41] used for red-black trees. We again omit the details here, and in the full version prove the following:

**Lemma 2.1** *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.*

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

## 2.2 Update Operations

We now discuss the methods for various 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 [5].

**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. 1). 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 $T_y$, both 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 $x$. 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

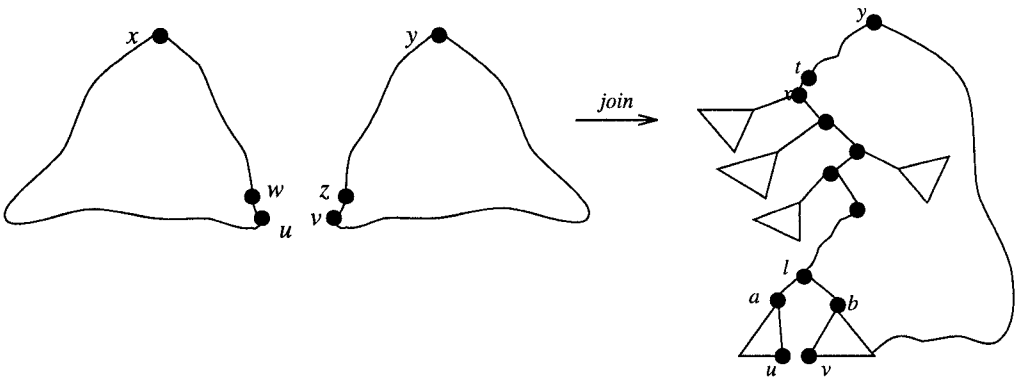



Figure 1: Join of trees $T_x$ and $T_y$.

$t$ as the left child of $t$. For the other nodes in $A$, we proceed as follows (see Fig. 1): 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 until 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 show in the full version that the total number of credits needed to perform this update is $\bar{O}(1)$. This, and Lemma 2.1, establishes that the running time for a join is $\bar{O}(1)$.

**Split:** We perform the split operation as in [5]. We show that with the same complexity we can preserve all three types of credits in the nodes of the resulting trees.

**Insertion:** Consider the insertion of an item $i$ with weight $w_i$ to a biased finger tree $T$. Recall that $i^-$ denotes the immediate predecessor of item $i$ in $T$ if it exists, and immediate successor of $i$ in $T$ otherwise. We provide a pointer to $i^-$. In the full version we describe how to perform a bottom-up insertion from $i^-$. The most interesting case occurs when the new item has weight much larger than $i^-$, for we must then splay $i$ up the tree to its proper position while joining together the trees whose roots become siblings in this process (so as to maintain our minor-node bias property). We show that this can be done in $\bar{O}(|\log w_i/w_{i^-}|)$ time.

**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. This operation is similar to that of insertion and we have different cases based on whether the node deleted is minor/major. We use different operations to preserve bias property in each case. We give details in the full version where we show the complexity of deletion to be $\bar{O}(\log w_i)$.

We summarize:

**Theorem 2.2** *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, not counting the search times.*

Bent *et al.* [5] 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 2.3** *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. ■

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

# 3 The Layers-of-Maxima Problem

In this section, we use the biased finger tree data structure to solve an open (static) computational geometry

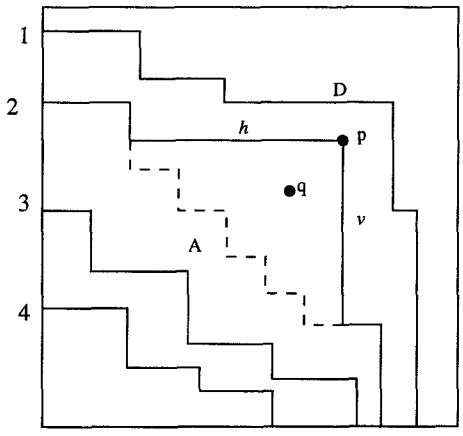



Figure 2: Computation of Layer for a New Point $p$.

problem: the 3-dimensional layers-of-maxima problem. Before we describe our method, however, we introduce some definitions. A point $p \in \Re^3$ *dominates* a point $q \in \Re^3$, if $x(q) < x(p)$, $y(q) < y(p)$, and $z(q) < z(p)$. Given a set $S$ of points in $\Re^3$, a point $p$ is a *maximum* point in $S$, if it is not dominated by any other point in $S$. We define the *dominance region* of a point $p$ in $\Re^3$ as a set $D \subset \Re^3$ such that $p$ dominates every point $q \in D$. The *maxima set* problem is to find all the maximum points in $S$. Kung, Luccio, and Preparata [27] 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 [7], and it appears that it can be solved for a 3-dimensional point set $S$ in $O(n \log n \log \log n)$ time [1] using the dynamic fractional cascading technique of Mehlhorn and Näher [32]. We show how to solve the 3-dimensional layers-of-maxima problem in $O(n \log n)$ time, which is optimal [3].

We solve this problem using a three-dimensional sweep, and a dynamic method for point location in a staircase subdivision. Given a set $S$ of $n$ points in $\Re^3$, 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. When we sweep across a point, we compute its layer using the information about the layers computed so far. The information we maintain for each layer is the union of the dominance regions of the points in that layer. We denote this by $\mathcal{D}(l)$, for a layer $l$. We show that to correctly compute the layers, it is sufficient to maintain for each layer $l$, the boundary of the intersection of the sweep plane, say $\pi$, with $\mathcal{D}(l)$. The intersection region is two dimensional, and we call its boundary a *staircase*.

The shape of the staircase representing a layer changes, as we continue the sweep. During the sweep, we maintain only a subset of points that belong to each layer. This is because, if points, say $p$ and $q$, belong to a layer $l$, and if the projection of dominance region of $p$ onto $\pi$ dominates that of $q$, then point $q$ will not be part of the boundary of intersection. 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 the staircases corresponding to the layers of points in $S'$, subdivide the $xy$-plane into disjoint regions (as defined earlier). We call this a *staircase subdivision*. 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 form a staircase subdivision, and we reduce the computation of layer for a new point to operations on the staircase subdivision. We call the region in the subdivision between two staircases, a *face*. Hence, if there are $m$ layers, they subdivide the $xy$-plane into $m + 1$ faces. The projection of each new point onto the $xy$-plane belongs to a unique face among these $m + 1$ faces. We work with the staircase subdivision, and the projection of a new point, to identify the point's layer.

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

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

2. Compute the horizontal segment $h$, and the vertical segment $v$ from $p$, which hit the boundary or some layer (of course the layer hit by $h$ is the same layer that is hit by $v$, and has same number as $p$'s just computed 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 2, 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 Figure 2 to illustrate the idea. Consider layer 2. The

[3] 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.

staircase of layer 2 is the boundary of the intersection of the union of the dominance regions of the points in the layer 2 with the sweep plane $\pi$. Since the points are processed in decreasing order of their $z$ coordinates, when we insert $p$, the dominance region of 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. Now to update the boundary of intersection with $\pi$, we delete the portion $A$ from layer 2, and include the segments $h$ and $v$ into the subdivision. To see why only the boundary is maintained, we observe that if point $q$ is introduced later, 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 an existing layer (see Figure 2). In any case, $l(q) = l(p) + 1$. Also, we observe that after deleting points in portion $A$, the new boundary for layer 2 is in the form of a staircase, thus preserving the property.

Suppose location, insertion and deletion of a vertex/edge in a staircase subdivision take time $Q(n), I(n)$, and $D(n)$ time respectively. We implement step 1 as a point location in staircase subdivision, and it takes $O(Q(n))$ time. We represent the staircase corresponding to each layer by a dictionary, for ordering along $x$ and $y$ axes (since both are same ordering). 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, step 2 takes $O(\log n)$ time. Therefore, 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 $Q(n) = I(n) = D(n) = \bar{O}(\log n)$, 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 Dynamic Point Location

In this section, we address the general problem of dynamic point location in a convex subdivision. 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, 23, 28, 29, 35, 36, 39]). It is well-known that one can construct a linear-space data structure for answering such queries in $O(\log n)$ time [13, 16, 23, 39].

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), a *staircase* subdivision is one in which each face is a region bounded between two infinite staircases, 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." The interested reader is referred to the excellent survey by Chiang and Tamassia [9] for a discussion of these and other results in dynamic computational geometry.

## 4.1 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 [35] for static point location. Incidentally, this is also the approach used by Chiang and Tamassia [8], 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 [35]. Our structure is a rooted tree, $T$, each of whose nodes is associated with a trapezoid $\tau$ whose parallel boundary edges are vertical. We imagine the trapezoid $\tau$ 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 $\mathcal{S}$.

[4] Cheng and Janardan's update method is actually a deamortization of an amortized scheme via the "rebuild-while-you-work" technique of Overmars [34].

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

| 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.* [4] |
| connected | $O(\log^2 n)$ | $O(\log n)$ | $O(\log n)$ | Cheng-Janardan [11][4] |
| connected | $O(\log n)$ | $\bar{O}(\log^3 n)$ | $\bar{O}(\log^3 n)$ | Chiang *et al.* [10] |
| monotone | $O(\log n)$ | $\bar{O}(\log^2 n)$ | $\bar{O}(\log^2 n)$ | Chiang-Tamassia [8] |
| 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 [32] |
| convex | $O(\log n + \log N)$ | $O(\log n \log N)$ | $O(\log n \log N)$ | Preparata-Tamassia [38] |
| convex[5] | $O(\log n)$ | $\bar{O}(\log n)$ | $\bar{O}(\log^2 n)$ | this paper |
| staircase | $O(\log n)$ | $\bar{O}(\log n)$ | $\bar{O}(\log n)$ | 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.

Let $v$ therefore be a node in $T$ with trapezoid $\tau$ associated with it. If no vertex or edge of $\mathcal{S}$ intersects the interior of $\tau$, then we say that $\tau$ is *empty*, in which case $v$ is a leaf of $T$. Note that in this case any point determined to be inside $\tau$ is immediately located in the cell of $\mathcal{S}$ containing $\tau$. Let us therefore inductively assume that $\tau$ 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 $\tau$'s left and right boundaries while not intersecting $\tau$'s top or bottom boundary. In this case we divide $\tau$ 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 $\tau$, an action we refer to as a *vertical* cut. This creates two new trapezoids $\tau_l$ and $\tau_r$, 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 $\tau_l$ and $v_r$ associated with $\tau_r$.

2. There is at least one face of $\mathcal{S}$ that intersects both the left and right boundaries of $\tau$ and does not have a spanning edge of $\tau$ as its top or bottom boundary. In this case we "cut" $\tau$ through each of the faces of $\mathcal{S}$ that intersect $\tau$'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 $\tau_i$ 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 3). 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 not a problem, 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 $\tau$. We say that a face $f$ or an edge $e$ of the subdivision *covers* a trapezoid $\tau$ if it spans $\tau$ horizontally and it does not span any ancestor of $\tau$ in $T$. The structure of $T$ has the property that any face or edge covers 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 [31].

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 $\tau$ is a current trapezoid in the subdivision (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 $\tau$. This identifies the left or right secondary data structure of $\tau$ 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 $\tau$, we compare the $(x, y)$ value of the point against the sup-

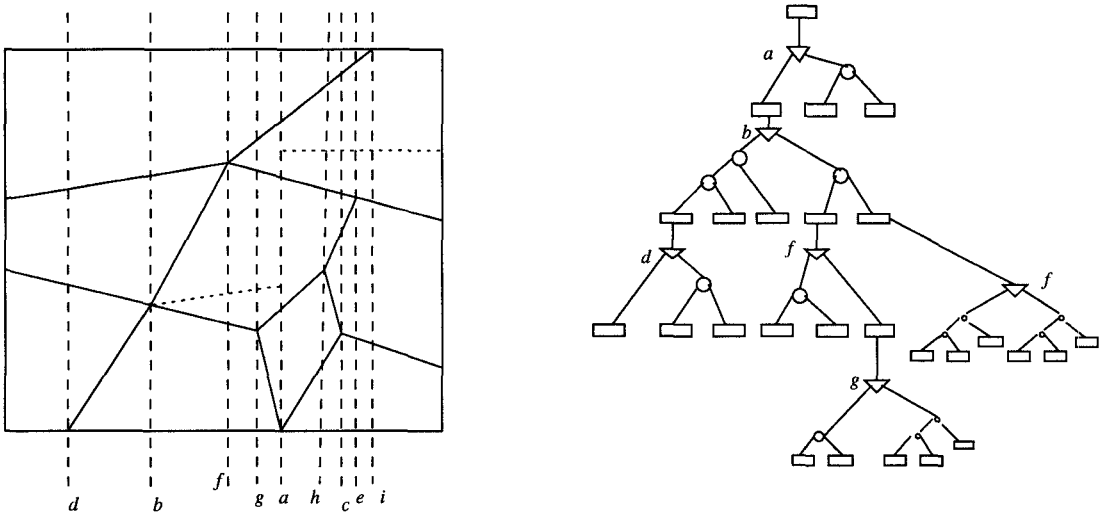



Figure 3: Trapezoidal Decomposition of a Convex Subdivision with Pseudo-Edge Cuts. The triangular nodes denote vertical cuts and the circular nodes denote horizontal cuts.

porting line of the spanning edge or pseudo-cut [6]. This identifies a leaf node, say $q$, in the biased finger tree that represents a trapezoid, say $\tau_q$, in the primary data structure. We now recursively locate the point by calling $query(\tau_q, x, y)$.

The arguments in the previous section on the structure of our data structure imply that, starting from the root of $T$, we can perform the point location query in $O(\log w(r) + depth(\tau))$ time in the worst case, where $r$ 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 $O(\log w(r))$. Noting that $w(r)$ is $O(n \log n)$ [35] 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.

Our method for updating this structure is rather involved; hence, for space reasons, we can only sketch the main ideas in this extended abstract. In the case of an insertion of an edge $e$ into a face $f$ our method traverses down the tree for the endpoints of $e$. At each node that $f$ covers and such that $e$ cuts so that the new faces no longer cover we must perform $O(1)$ join operations. Since there are $O(\log n)$ such nodes, the total time for these updates, then, is $\bar{O}(\log n)$. In addition, there are also $O(\log n)$ places in the tree where we must insert this new edge, each of which can be implemented in $\bar{O}(1)$ time, given a pointer to the location for this insertion (which we obtain from the $\Delta$ list for $f$). Thus, the total time for an insertion is $\bar{O}(\log n)$. The case of a deletion is essentially the reverse of the above operations. This operation runs in $\bar{O}(\log^2 n)$ time, however, since we now may have to perform $O(\log n)$ split operations (and their complexity, unfortunately, does not form a telescoping sum).

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

## 4.2 Rebalancing the Primary Structure

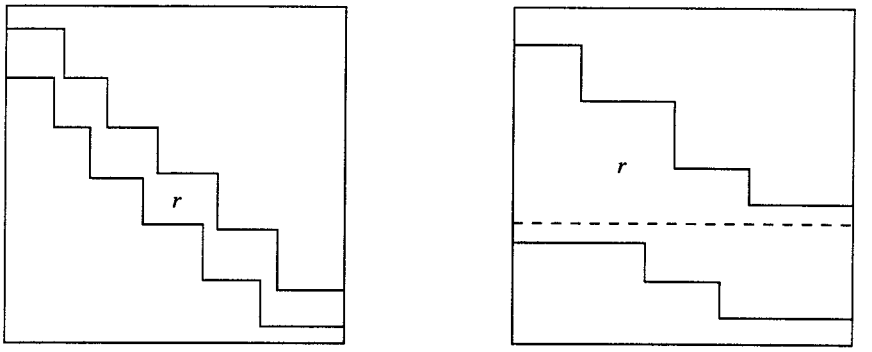
In this section we show how to relax the constraint that the endpoints have $x$ coordinates in the range $[1, n]$. We use a $BB[\alpha]$-tree as a primary tree for vertical cuts with biased finger tree as a secondary structure in each node. We briefly review the properties of $BB[\alpha]$-tree. Let $f(l)$ 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 [30]:

- 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 $f(n) = O(n)$ and so the rebalancing cost is $\bar{O}(\log n)$. We give the details in the full version.

Thus we show,

**Theorem 4.1** *Given a convex subdivision $S$ of $n$ vertices, there exists a data structure which allows point location queries in $O(\log n)$ time, vertex/edge insertion in $\bar{O}(\log n)$ time, and vertex/edge deletion in $\bar{O}(\log^2 n)$ time.*




a. A Spanning Face Without any Pseudo-cut

b. A Spanning Face With a Pseudo-cut

Figure 4: Pseudo-Cut for a staircase spanning face.

### 4.3 Dynamic Point Location in Staircase Subdivisions

In this section, we construct an $O(\log n)$ depth trapezoidal structure for staircase subsubdivisions similar to the one for convex subdivision, which allows us to perform query in $O(\log n)$ time, and updates in $\bar{O}(\log n)$ time. We always require that deletion of an edges (vertices) should always be accompanied by an insertion of an edge (vertices).

The trapezoidal structure for staircase subdivision satisfies a slightly different invariant from that of convex subdivision. Here we use an invariant that if a face $f$ spans a trapezoid $\tau$ and $f$ can be split into two faces by a (truly) horizontal spanning segment, say $e$, of $\tau$, then we split $\tau$ into two trapezoids using $e$. So, unlike convex subdivision, not every spanning face of $r$ will have a pseudo-cut (see Figures 4.a and 4.b) here. Using this invariant, we can easily do an analysis similar to that of convex subdivision to show that the insert operation in a staircase subdivision takes takes $\bar{O}(\log n)$ time. The details are omitted.

Using this invariant, we can easily do an analysis similar to that of convex subdivision to show that the insert operation in a staircase subdivision takes takes $\bar{O}(\log n)$ time. The details are omitted. We observe that, each edge deleted is always bounded on the top and on the right by "long" edges. 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. This eliminates the costlier case of deletion which takes $\bar{O}(\log^2 n)$, and all the other cases take $\bar{O}(\log n)$ time. We give details in full version. Thus, this observation results in a $\bar{O}(\log n)$ time method for deletion of an edge in the staircase subdivision.

Thus we show,

**Theorem 4.2** *Given a staircase subdivision $S$ of $n$ vertices, there exists a data structure which allows point location queries in $O(\log n)$ time, edge (vertex) insertion in $\bar{O}(\log n)$ time, and edge (vertex) deletion in $\bar{O}(\log n)$ time (the deletions are always coupled with insertions, however).*

The edge insertion and deletions performed on staircase subdivision in our three-dimensional layers of maxima algorithm satisfy the constraints specified in Theorem 4.2. Therefore we have,

**Theorem 4.3** *Given a set $S$ of $n$ points in $\Re^3$, one can construct the layers of maxima for $S$ in $O(n \log n)$ time, which is optimal.*

## Acknowledgement

We would like to thank Rao Kosaraju for several helpful conversations concerning the topics of this paper.

## References

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

[2] 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.

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

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

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

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

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

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

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

[10] Y.-J. Chiang, F.P. Preparata, and R. Tamassia "A Unified Approach to Dynamic Point Location, Ray Shooting, and Shortest Paths in Planar Maps," *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, 1993, 44–53.

[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] M.T. Goodrich and R. Tamassia, "Dynamic Ray Shooting and Shortest Paths via Balanced Geodesic Triangulations," *Proc. 9th ACM Symp. on Computational Geometry*, 1993, 318–327.

[20] 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.

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

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

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

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

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

[26] S.R. Kosaraju, "An Optimal RAM Implementation of Catenable Min Double-ended Queues," *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, 1994, 195–203.

[27] 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.

[28] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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