Dynamisation of Geometric Data Structures

by

O. Fries, K. Mehlhorn, St. Näher

Fachbereich 10, Angewandte Mathematik und Informatik
Universität des Saarlandes, 6600 Saarbrücken
West Germany

1. Introduction

Many data structures used in computational geometry are quite involved and therefore hard to dynamize. The purpose of this paper is twofold. In section 1 we describe dynamic fractional cascading. Fractional cascading was recently introduced by Chazelle/Guibas as a common framework for many data structures in computational geometry. They show that it allows to derive numerous old and new results in a uniform way and thus enhance our understanding of geometric data structures tremendously. They distill a common principle which was implicitely used previously in Vaishnavi/Wood, Willard, Edelsbrunner/Guibas/ Stolfi, Imai/Asano and others. In section 2 we give an amortized analysis of update cost in fractional cascading and show that insertions take $O(1)$ amortized time and insertions and deletions take $O(\log\log N)$ amortized time. The analysis is based on the technique developed in Maier/Salveter and Huddleston/Mehlhorn (cf. also Mehlhorn, Vol 1) for analyzing amortized update cost in balanced trees. The efficient set splitting algorithm of Imai/Asano (a new set splitting and union algorithm based on the $O(\log\log N)$ priority queue of v. Emde Boas) is used as an auxiliary data structure in order to support efficient insertions (insertions and deletions). In section 3 we will briefly survey two recent attempts for dynamizing the searching problem in planar subdivisions.

2. Fractional Cascading

2.1. Iterated Search Problems

Chazelle/Guibas noted that many data structures can be viewed as solutions to iterated search problems in iterated search structures.

An *iterated search structure* is defined as follows: $G = (V, E)$ is a directed graph, called the *search graph*, $v_0 \in V$ is a special vertex, called the *entry point*, and U is a linearly ordered set, called the *universe*. For every vertex $v \in V$ we are given a set $C(v) \subseteq U$, called the *catalogue* of v. For every edge $e \in E$ we are given an interval $R(e) = [l(e), r(e)] \subseteq U$, called the range of e. We assume that the search structure is *locally bounded*, i.e. there is an integer d such that for all $x \in U$ and all $v \in V : |\{w \mid w \in V$ and $x \in R(v, w)$ or $x \in R(w, v)\}| \leq d$ An *iterated search problem* in a search structure is given by a key value $k \in U$ and a subtree T of G rooted at v_0. The problem is to determine for every $v \in T$ the maximal element in $C(v)$ not exceeding k.

We want to briefly illustrate these definitions. Let $S \subseteq R^2$ be a set of n points in the plane, and let $X(S)$ be the set of x-coordinates for the points in S. A *range tree* (Bentley, Lueker, Willard, cf. also Mehlhorn Vol. 3)for S consists of a primary tree PT, which is a balanced tree for set $X(S)$. For every vertex v of PT the catalogue $C(v)$ consists of those points $p \in S$ such that a search for x-coordinate $x(p)$ in tree PT goes through v. The catalogues are ordered according to y-coordinate. Range trees can be used to solve *orthogonal range queries* as follows: Let x_0, x_1, y_0, y_1 be reals. We want to enumerate all points in $A = S \cap ([x_0, x_1] \times [y_0, y_1])$. Let P_i be the search path for x_i, $i = 0, 1$, in the primary tree and let Q be the set of roots of maximal subtrees between these paths. Then $A = \bigcup\{C(v) \cap [y_0, y_1] \mid v \in Q\}$. Thus if we have located y_0 in catalogues $C(v)$, $v \in Q$, then A is readily enumerated in time $O(|Q| + |A|) = O(\log n + |A|)$.

Range trees are easily viewed as iterated search structures. G is the primary tree, $U = R$ is the set of y-coordinates, v_0 is the root of the primary tree and $R(e) = U$ for every edge e. The catalogues were defined above. An orthogonal range query problem can now be formulated as follows. The tree T consists of path P_0 and P_1 together with set Q and the key value k is y_0.

Fractional Cascading (Chazelle/Guibas) is a method for solving iterated search problems efficiently.

Theorem 1. (Chazelle/Guibas): Let N be the total size of the catalogues. Then there is a data structure, called fractional cascading, which allows to solve iterated search problems in time $O(\log N + |T|)$. The data structure requires space $O(N + |G|)$ and can be constructed in time $O(N + |G|)$. ∎

Fractional cascading is a generalization of tree search. In tree search we have a single catalogue C which we store in the bottom level of the tree. In the level above the leaves we store a fraction, say every fifth, of the elements in the bottom level, and so on. In the root level we have a set of size $O(1)$. In this way the hierarchy has $O(\log N)$ levels, requires space $O(N)$, and allows for $O(\log N)$ searches because $O(1)$ time per level of the hierarchy is needed. Fractional cascading extends this idea to graphs. With every vertex v of the search graph we associate an augmented catalogue $A(v) \supseteq C(v)$. The augmented catalogue contains $C(v)$ and also serves as a directory for the searches in the descendants of v. This is achieved by copying a fraction of $A(w)$ into $A(v)$ for every descendant w of v. If this fraction is small enough (say every $2d$-th element) then the space requirement will be linear. Also the search will take time $O(\log N)$ in the entry node and time $O(1)$ in every node of T, hence theorem 1. How can we make fractional cascading dynamic, i.e. how can we take care of insertions into and deletions from the catalogues? What can we learn from dynamic trees, e.g. (a, b)-trees, i.e. trees where every node has between a and b sons, $2 \leq a \leq b$. In (a, b)-trees at least one out of every b elements on some level is stored on the next higher level (this ensures the $O(1)$ search time per level), and the presence of an element at some level is justified (supported) by at least $a \geq 2$ elements at the lower level (this ensures the $O(N)$ total space requirement and the logarithmic height). After an insertion or deletion rebalancing operations (splitting and fusing

of internal nodes of the tree) are performed to maintain the structural invariant. Maier/Salveter and Huddleston/Mehlhorn (cf. also Mehlhorn Vol 1) have shown that the amortized number of rebalancing operations after an insertion and deletion is $O(1)$ for an appropriate choice of parameters a and b. We show how to extend the concept of (a, b) trees to fractional cascading and prove the analogous result for amortized rebalancing cost. More precisely, we show:

Theorem 2. (Mehlhorn/Näher):

a) Fractional Cascading can handle queries in time $O(\log N + |T|)$ and insertions in time $O(1)$.

b) Fractional Cascading can handle queries in time $O((1 + |T|) \log\log N)$ and insertions and deletions in time $O(\log\log N)$.

All time bounds are amortized. Also, the time required to locate the point of insertion/deletion is not counted in the insertion and deletion times. Finally, the readers should take care when applying this theorem to, for example, range trees. An insertion of a point $p \in R^2$ into a range tree corresponds to $O(\log n)$ insertions into catalogues (!!!) and hence theorem 2a does *not* improve upon the results of Imai/Asano. However, insertions, deletions and queries into range trees can be handled with time $O(\log n \log\log n)$ an improvement over the $O((\log n)^2)$ bounds of Lueker and Willard. A similar remark applies to segment trees. The $O(\log N \log\log N)$ bound was obtained previously by Lipski for static universes of size N. More importantly theorem 2 gives a general framework for the dynamic behavior of these structures.

2.2. Dynamic Fractional Cascading: Definitions and Static Behavior

Let a and b, $a \leq b$, be two integer parameters. We will define the dynamic fractional cascading data structure, and analyze its space requirements and search time.

As mentioned above, there is *augmented catalogue* $A(v)$ for every vertex v. It satisfies (among other properties to be defined below):

$$1) C(v) \subseteq A(v)$$

$$2) \{l, r\} \subseteq A(v)$$

for every edge $e = (v, w)$ with $R(e) = [l, r]$. (We use [,] for closed intervals and (,) for open intervals). We call $x \in A(v)$ a *genuine* element of $A(v)$ if either $x \in C(v)$ or $x = l(e)$ or $x = r(e)$ for some edge $e = (v, w)$. All other elements of $A(v)$ are called *non-genuine*. For $x \in A(v)$ let $suc(x)$ be the successor of x in $A(v)$ and let $pred(x)$ be its predecessor, i.e. $suc(x) = min\{y \mid y \in A(v) \text{ and } x \leq y\}$. For $x \in A(v)$ and edge $e = (v, w)$ with $x \in R(e)$ let $P_w(x)$ be the predecessor of x in $A(w)$, i.e. $P_w(x) = \max\{y \mid y \in A(w)$ and $y \leq z\}$. We assume that $suc(x)$, $pred(x)$ and $P_w(x)$ are realized as pointers. Finally for $x \in A(v)$, $e = (v, w)$ and $x \in R(e)$ let

$$Int_w(x) = (P_w(pred(x)), P_w(x)) \cap A(w)$$

be the set of elements in $A(w)$ between $P_w(x)$ and $P_w(pred(x))$. If $x = l(e)$ and hence $P_w(pred(x))$ undefined we let $Int_w(x) = \emptyset$. We want our data structures to satisfy the following invariants

1) For all $x \in A(v), e = (v, w)$ and $x \in R(e) : |Int_w(x)| \leq b$

2) For all non-genuine elements $x \in A(x)$ there is a direct descendant $w = support(x)$ of v with $x \in R(e), e = (v, w)$ such that: Let $y \in A(v), y \leq x$ be maximal such that either y is non-genuine and $support(y) = support(x)$ or $y = l(e)$. Then $|S_w(x)| \geq a$ where

$$S_w(x) = [P_w(y), P_w(x)] \cap A(w)$$

Invariant 1 correspondents to the upper bound on the degree in (a, b)-trees and invariant 2 corresponds to the lower bound on the degree.

Lemma 1.

Let $N = \sum_{v \in V} |C(v)|, S = \sum_{v \in V} |A(v)|$. Then $S \leq 2N + 4|E|$ provided that $a \geq 2d$. *Proof*: Consider a fixed $v \in V$. The number of genuine elements in $A(v)$ is bounded by $|C(v)| + 2outdeg(v)$. Consider the non-genuine elements in $A(v)$ next. For a direct descendant w of v, i.e. $e = (v, w) \in E$, let $NG_w(v) = \{x \in A(v) \mid x \text{ is non genuine and } support(x) = w\}$. For $x, y \in NG_w(v), x \neq y$ we have $S_w(x) \cap S_w(y) = \emptyset$, $|S_w(x)| \geq a$, $|S_w(y)| \geq a$, $S_w(x), S_w(y) \subseteq R(e) \cap A(w)$, and hence $|NG_w(v)| \leq |A(w) \cap R(e)|/a$. Thus

$$S = \sum_{v \in V} |A(v)| \leq \sum_{v \in V} (C(v) + 2\text{outdeg}(v)) + \sum_{v \in V} \sum_{(v,w) \in E} |A(w) \cap R((v,w))|/a$$
$$\leq N + 2|E| + (d/a) \sum_{w \in V} |A(w)|$$

since for every $z \in A(w)$ there are at most d edges e with $z \in R(e)$. Thus $S/2 \leq N + 2|E|$. ∎

Let us turn to query time next. We need one additional definition. For $x \in A(v)$ let $find(x) = \max\{y \in C(v); y \leq x\}$ be the predecessor of x in the catalogue (not the augmented catalogue). We assume that we can compute $find(x)$ in time $q(N)$.

Lemma 2.

a) Let $k \in U$ and let $x \in A(v)$ be such that $pred(x) \leq k \leq x$. If $k \in R(e), e = (v, w)$, then time $O(1)$ suffices to locate $y \in A(w)$ with $pred(y) \leq k \leq y$. b) An iterated search problem takes time $O(\log N + |T| q(N))$.

Proof: a) We clearly have $P_w(pred(x)) \leq k$. Also $k \leq x \leq suc(P_w(x))$. Hence there are only $b + 1$ possibilities for item b.

b) in time $O(\log N)$ we can determine $x \in A(v_0)$ with $pred(x) \leq k \leq x$. In additional time $O(|T|)$ we can locate k in all augmented catalogues $A(v), v \in T$. Then time $O(q(N))$ per node of T is needed to locate x in the catalogue itself. ∎

In a static environment we have $q(N) = O(1)$ since operation $find$ can be realized by a pointer from x to $find(x)$. In a dynamic setting $C(v)$ and $A(v)$ change over time and hence this implementation is not adequate. We consider the following additional operations: The names for the operations are motivated by the fact that $C(v)$ partitions $A(v)$ into intervals. For $x \in A(v)$, $split(x)$ makes x into an element of $C(v)$. For $x \notin A(v), y \in A(v)$ with $y = \max\{z \in A(v); z < x\}$, $add(x, y)$ adds x to $A(v)$. For $x \in C(v)$, $union(x)$ makes x into an element of $A(v) - C(v)$ and finally for $x \in A(v)$, $delete(x)$ deletes x from $A(v)$. We assume that each operation above can be executed in (amortized) time $U(N)$. Clearly, if we present for every $x \in C(v)$ the set of $y \in A(v)$ with $x = find(y)$ as a balanced tree then $U(N) = O(\log N)$ and $q(N) = O(\log N)$. There are better solutions, however.

Theorem 3a. (Imai, Asano): Operations Find, Split and Add can be realized with $q(N) = U(N) = O(1)$

Theorem 3b. The full repertoire of all five operations can be realized with $q(N) = U(N) = O(\log \log N)$.

Proof: The data structure is derived from the $O(\log \log n)$ priority queue of v. Emde Boas. Let $n = |A(v)|$. We call an element $x \in A(v)$ marked iff $x \in C(v)$. We store $A(v)$ in groups of between $1/2\sqrt{n}$ and $2\sqrt{n}$ elements. With each group we store pointers to the minimal and maximal marked element in

it (the extrema pointers). Also each group has a representative. A representative is marked if its group contains a marked element. For each element $x \in A(v)$ we have a pointer to the representative of the group containing x (ancestor-pointer) and to the copy of x in this group (copy-pointer). (cf. Fig. 1)

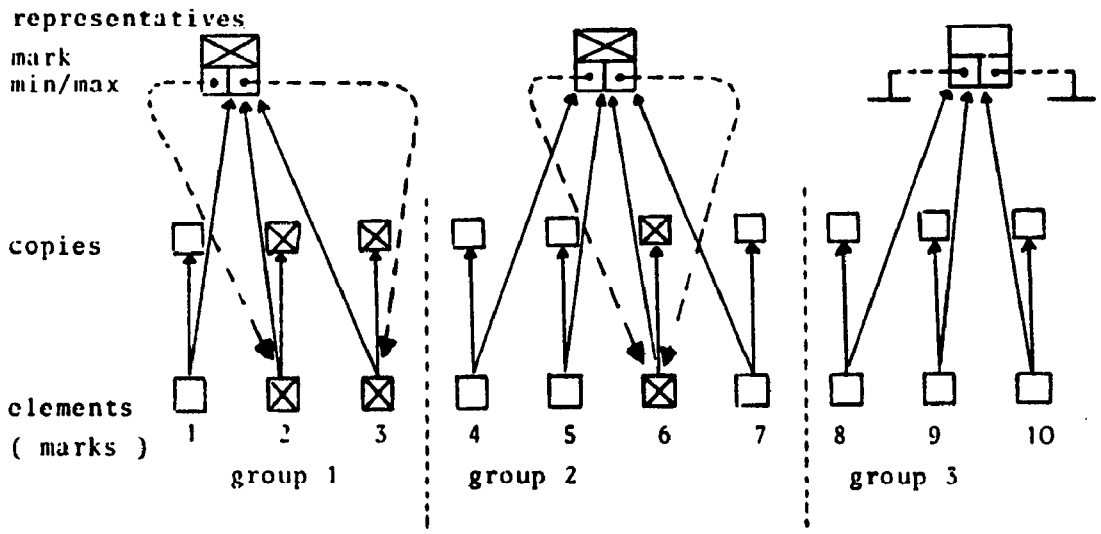

Fig. 1

Each group and the set of representatives is organized recursively in the same way except for one detail. If a group or the set of representatives contains at most one marked element then the extrema-pointers in the substructures are undefined. ∎

Lemma 3.

(v.Emde Boas, cf. also Mehlhorn, Vol I): Find, Union and Split take time $O(\log\log n)$. The data structure requires space $O(n \log\log n)$ and can be constructed in time $O(n \log\log n)$

Proof: The space requirement follows from the observation that there are $O(\log\log n)$ levels of hierarchy. Find, Union, Split are called Pred, Delete and Insert respectively in the priority queue terminology, the time bound follows. ∎

It remains to deal with operations Delete and Add. Delete x is performed by tagging x and all of its copies. Add(x, y) is performed by adding x next to y. The copy pointers are defined in the obvious way and the ancestor pointers of x are set to the ancestor pointers of y. In this way Add and Delete take time $O(\log\log n)$. Note however, that the size of substructures changes by additions and deletions. Let H be a maximal structure which over- or underflows, i.e. the number of elememts in H exceeds $\sqrt{w}$ (counting the tagged elements) or falls below $\sqrt{w}/2$ (not counting the tagged elements) where w was the size of the immediate superstructure of H when it was rebuilt most recently. Let H_1 be an immediate brother structure of H. We combine the elements in H and H_1 and build new structures $S_1, S_2, \ldots, S_c$ for them, $1 \leq c \leq 4$, where each S_i contains between w and $3w/2$ elements. This has cost $O(\sqrt{w} \log\log \sqrt{w})$. It also forces us to do $O(1)$ adds or deletes in the data structure for the representatives. It remains to estimate the amortized rebalancing cost.

Lemma 4. Amortized Rebalancing Cost in $O((\log\log n)^2)$

Proof: We charge the $O(\sqrt{w}\log\log\sqrt{w})$ time units to the $O(\sqrt{w})$ elements inserted into the structures since it was rebuilt most recently, i.e. $O(\log\log\sqrt{w})$ units per element. Since an element is contained in structures of size $n, n^{1/2}, n^{1/4}, \ldots, 1$ this adds up to $O((\log\log n)^2)$ per add or delete. ∎

It is now easy to complete the proof. We divide $A(v)$ into chunks of size between $\log\log n$ and $2\log\log n$. The chunks are realized as linear lists, and representatives for the chunks are stored in the data structures described above. Then only every $O(\log\log n)-th$ add or delete propagates to the data structure described above and hence rebalancing time reduces to $O(\log\log n)$. The time bounds for the other operations are changed by an additive factor of $O(\log\log n)$. ∎

3. Dynamic Behavior

In this section we deal with insertions and deletions. We describe the insertion and deletion algorithms and analyze their amortized cost. The algorithms and their analysis show a strong resemblance to $(a,b)-$ *trees*. We will make the following two assumptions:

1) When r has to be inserted into $C(v)$ we are given a pointer to $x \in A(v)$ with $pred(s) \leq r \leq s$.
2) When r has to be deleted from $C(v)$ we are given a pointer to $r \in A(v)$.

Theorem 2. The amortized cost of an insertion or deletion is $O(U(N))$.

The main tool in our amortized analysis is the concept of a *token*. A token represents the ability to pay for $O(U(N))$ computing time. We will associate accounts with our data structure in which we save tokens. Every insertion or deletion deposits $O(1)$ tokens into these accounts. The cost of rebalancing operations is covered by withdrawing tokens from these accounts. The details are as follows.

Let a', b' c be integers satisfying the following conditions (The nature of these requirements will become clear later on):

1) $a \leq a' \leq b' \leq b, a \leq b, c \geq 0$
2) $\lfloor (b+1)/2 \rfloor \geq b'$
3) $b' \geq d+1$
4) $c \geq a$
5) $c + a' - a \geq d(c+a+2)$

One particular choice for the parameters is $a = c = 2d$, $a' = b' = 2d(d+1)$ and $b = 2b''$.

For every $r \in A(v)$ we maintain up to $d+1$ accounts, namely the wOaccount for every descendant w of v with $r \in R(v,w)$ and if r is non-genuine also an Uaccount. (Here O and U suggest overflow and underflow respectively). The wOaccount for r contains

$$\max(0, |Int_w(r) - b'|)$$

and the Uaccount of a non-genuine element r with support $w = support(r)$ contains

$$\max(0, a' - |S_w(r)|)$$

tokens.

Remark: The larger $Int_w(r)$ the more tokens are in the wOaccount of r. In particular, if $Int_w(r)$ overflows, i.e. $|Int_w(r)| \geq b$, there will be at least $b+1-b'$ tokens in this account. Similarly, the smaller the support of r, the more tokens are in the Uaccount.

During rebalancing we allow elements to temporarily overflow or underflow. Such elements are stored in sets Overflow and Underflow respectively. More precisely, Overflow and Underflow contain triples (r, v, w). If $r \in A(v), r \in R(v, w)$ and $|Int_w(r)| \geq b$ then $(r, v, w) \in Overflow$. If $r \in A(v), w = support(r)$ and $|S_w(r)| \leq a$ then $(r, v, w) \in Underflow$. We also associate an account with set Underflow. The Underflowaccount contains $c|Underflow|$ tokens. We are now ready for the *insertion algorithm*. Suppose that we want to insert key r into catalogue $C(v)$. We perform the following algorithm.

(1) add r to $A(v)$; **co** r is a genuine element in $A(v)$ **oc**
(2) **while** Overflow $\neq \emptyset$
(3) **do** let $(s, v, w) \in$ Overflow be arbitrary;
(4) choose $r \in Int_w(s)$ such that
(5) $|Int_w(s) \cap \{x \mid x \leq r\}| = \lfloor |Int_w(s)|/2 \rfloor$;
(6) add r to $A(v)$; **co** r is a non-genuine element in $A(v)$ **oc**
(7) **od**

It remains to describe how to add a key r to $A(v)$. Let $s \in A(v)$ such that $pred(s) \leq r \leq s$. Key s will be given to us in line (1) and is defined in line (3) otherwise. We add r between $pred(s)$ and s to $A(v)$ and define $P_z(r)$ for all descendants z of v with $r \in R(v, z)$. This takes time O(U(N)). Also (the prime refers to the situation after inserting r) $Int'_z(r) = Int_z(s) \cap \{x \mid x \leq r\}$ and $Int'_z(s) = Int_z(s) - Int'_z(r)$. Let $m(z) = |Int_z(s)|, m_1(z) = |Int'_z(r)|$ and $m_2(z) = |Int'_z(s)|$. Then $m(z) = m_1(z) + m_2(z)$. Also, there were $\max(0, m(z) - b')$ tokens in the zOaccount of s and we need $\max(0, m_1(z) - b') + \max(0, m_2(z) - b')$ tokens in the zOaccount of s and r after insertion. This shows that no additional tokens are needed for the zOaccounts of s and r. In line (6) we can say even more. For $z = w$ we have $m(w) \geq b + 1, m_1(w) = \lfloor m(w)/2 \rfloor$ and $m_2(w) = \lceil m(w)/2 \rceil$. Since $m_1(w) \geq b'$ by requirement 2 we conclude that b' tokens become available in line (6) from the wOaccount of s. Let us look at Uaccounts next. In line (1) there is nothing to show since r is a genuine element. In line (6) we let $support(r) = w$. Note that the support of the element s' following r and having $w = support(s')$ is also at least $m_2(w) \geq b' \geq a'$. Thus no additional tokens are needed for Uaccounts.

Finally, we have to consider the predecessors u of v. Let $h \in A(u)$ be such that r is added to $Int_v(h)$. Then we add one token to the vOaccount of h and also add (h, u, v) to set Overflow if $|Int'_v(h)| \geq b$. This will cost us at most d tokens.

Lemma 5. The insertion algorithm requires $3d + 1$ tokens.

Proof: The cost of maintaining the data structure is $O(U(N))$ and hence one token suffices to pay for it. We also need d tokens for the ancestors u of v. Thus $d + 1$ tokens are needed for line (1). An execution of line (6) requires also $d + 1$ tokens. Since b' tokens become available in line (6) and $b' \geq d + 1$ by requirement 3 the net cost of line (6) is zero. ∎

We will next turn to the deletion algorithm. Suppose that we want to delete key r from $C(v)$. We declare key r a non-genuine element of $A(v)$ (operation union (v)) and add (r, v, w) to set Underflow where w is an arbitrary descendant of v. We also define $support(r) = w, S_w(r) = \emptyset$ and add c tokens to the Underflowaccount and a' tokens to the Uaccount of r.

Lemma 6. A deletion costs $1 + c + a'$ tokens.

Proof: We need one token to pay for operation union (v). We need $c + a'$ tokens for the accounts. ∎

It remains to discuss how to cope with set Underflow. Let $(r, v, w) \in$ Underflow. Then $g := |S_w(r)| \leq a, r \in A(v)$, and there are $a' - g$ tokens in the Uaccount of r. Also we may use c tokens from the Underflowaccount. We distinguish two cases.

Case A: There is a descendant z of v such that

$$m_1 := |\{x \in S_z(s) \mid x \leq P_z(r)\}| \geq a \qquad \text{and}$$
$$m_2 := |\{x \in S_z(s) \mid x \geq P_z(r)\}| \geq a$$

where $s \in A(v)$, $s \geq r$, is minimal with support$(s) = z$. (cf. Fig. 2)

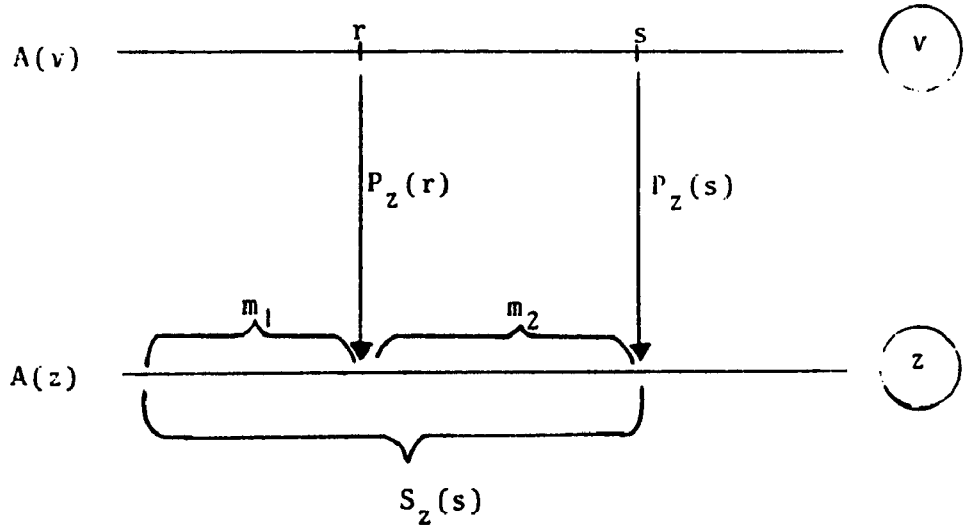

Fig. 2

Thus the total number of available tokens is $a' - g + c + \max(0, a' - m_1 - m_2) \geq a' - a + 1 + c + \max(0, a' - m_1 - m_2)$. The number of required tokens is $\max(0, a' - m_1) + \max(0, a' - m_2) + 1$ where the one accounts for the time needed to delete the triple (r, v, w) from Underflow. If $a' \leq m_1$ or $a' \leq m_2$ then the available tokens clearly suffice: recall that $m_1 \geq a, m_2 \geq a$. So let us assume $a' \geq m_1$ and $a' \geq m_2$. If $a' \geq m_1 + m_2$ then the available tokens suffice since $c \geq a$ by requirement 4 and if $a' \leq m_1 + m_2$ then the available tokens suffice for the very same reason. Thus the net cost of case A is zero.

Case B(= not Case A): No descendant z of v can give sufficient support to r and hence for every descendant z of v either $|Int_z(r)| \leq a$ or $|Int_z(suc(r))| \leq a$. Note that $suc(r) \leq s$ for the s defined in case A. We delete r from $A(v)$ (operation delete (r)) This takes time $O(U(n))$. For all ancestors u of v with $r \in R(u, v)$ let $h \in A(u)$ be such that $r \in S_v(h)$. The item h may not exist. We add one token to the Uaccount of h and add triple (h, u, v) to set Underflow if h overflows. The number of tokens required is $d(1 + c)$ and the running time is $O(1)$.

For all descendants z of v with $r \in R(v, w)$ we do the following. We add $\max(a - 1,$ zOaccount$(r))$ tokens to the zOaccount of $suc(r)$ and we add the triple $(suc(r), v, w)$ to Overflow if $Int'_z(suc(r)) = Int_z(suc(r)) \cup Int_z(r)$ exceeds b elements. Note that at most $\max(a-1,$ zOaccount$(r))$ additional tokens are required in the zOaccount of $suc(r)$ since either $|Int_z(suc(r))| < a$ or $|Int_z(r) < a$. Thus processing the descendants z of v has running time $O(1)$ and requires at most $(a - 1 + 2)d$ tokens.

In summary: We have $c + (a' - g) \geq c + a' - a + 1$ tokens available and we need $1 + d(1 + c + a + 1)$ tokens. By requirement 5 the available tokens suffice and hence the net cost of case B is zero.
This completes the proof of Theorem 4. Theorem 2 follows from Theorems 3, 4, and Lemma 2.

3. Searching Planar Subdivision

We briefly survey results on searching in dynamic planar subdivisions. A planar subdivision is a straight line embedding of a planar graph into a plane. The searching problem is to locate the face of the subdivision containing a query point p. There are several data structures in the literature achieving logarithmic search time and linear space (Lipton/ Tarjan, Kirkpatrick, Edelsbrunner/Guibas/Stolfi). We consider the following dynamic variant. An insertion splits a finite face by a straight-line segment and a deletion merges two finite faces.

Theorem 5.

a) (Fries/Mehlhorn, cf. Mehlhorn Vol 3, section 8.3.2.3) There is a solution with query and insertion time $O(\log^2 n)$. b) (Fries) There is a solution with query time $O(\log^2 n)$, insertion and deletion time $O(\log^4 n)$.

Both solutions are based on path decompositions of planar subdivisions (Lee/Preparata). In part a) path decompositions are combined with weight balanced trees, in part b) path decompositions are combined with a powerful insight about the shape of polygons. Call a polygon *monotone* if its "left" and "right" side are polygonal chains which are monotone with respect to y-coordinate. It is shown that every polygon P can be decomposed into monotone polygons such that every chord of P intersects at most $0(\log n)$ decomposition edges. It is also shown how to maintain such a decomposition dynamically.

References

[1] B. Chazelle, L. Guibas: "Fractional Cascading" *ICALP* 85, to appear

[2] H. Edelsbrunner, L.J. Guibas, L Stolfi: " Optimal Point Location in a Monotone Subdivision" *DEC System Research Report* No. 2, Palo Alto

[3] S. Huddleston, K. Mehlhorn: "A new Representation for Linear Lists" *Acta Informatica* 17, 157–184 (1982)

[4] H. Imai, T. Asano: "Dynamic Segment Intersection with Applications" *25th FOCS* 1984, 393–402

[5] D. Kirkpatrick: "Optimal Search in Planar Subdivisions" *SICOMP* 12 (1983), 28–35

[6] R. Lipton, R.E. Tarjan: "Applications of a Planar Separator Theorem" *18th FOCS*, 1977, 162–170

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

[8] G.S. Lueker: "A Data Structure for Orthogonal Range Queries" *19th FOCS*, 1978, 28–34

[9] D. Maier, S.C. Salveter: "Hysterical B-Trees" *Stony Brook, cs Dep. TR* #79/007

[10] K. Mehlhorn: "Data Structures and Algorithms", Vol 1: Sorting and Searching, Vol 2: Graph-Algorithms and NP-completeness, Vol 3: Multidimensional Searching and Computational Geometry, *Springer Publ. Comp.* 1984

[11] V.K. Vaishnavi, D. Wood: "Rectilinear Line Segment Intersection, Layered Segment Trees, and Dynamization" *Journal of Algorithms*, Vol 3 (1982), 160–176

[12] D.E. Willard: "New Data Structures for Orthogonal Queries" *SIAM J. of Computing*, 1985

[13] D.E. Willard: "New Data Structures for Orthogonal Range Queries" *Technical Report*, Harvard University, 1978

[14] P. v.Emde Boas, R. Kaas, E. Zijlstra: "Design and Implementation of an efficient priority queue" *Math. Systems Theory*, 10, 1977, 99–127

[15] W. Lipski: "Finding a Manhattan Path and Related Problems" *Networks* 13, 1983, 399–409