Dynamic Fractional Cascading * **

by

Kurt Mehlhorn and **Stefan Näher**

Fachbereich 10
Universität des Saarlandes
D-6600 Saarbrücken
Federal Republic of Germany

06/1986

Abstract. The problem of searching for a key in many ordered lists arises frequently in computational geometry. Chazelle and Guibas recently introduced fractional cascading as a general technique for solving this type of problem. In the present paper we show that fractional cascading also supports insertions into and deletions from the lists efficiently. More specifically, we show that a search for a key in n lists takes time $O(\log N + n \log\log N)$ and an insertion or deletion takes time $O(\log\log N)$. Here N is the total size of all lists. If only insertions or deletions have to be supported the $O(\log\log N)$ factor reduces to $O(1)$. As an application we show that queries, insertions and deletions into segment trees or range trees can be supported in time $O(\log n \log\log n)$, when n is the number of segments (points).

* This research was supported by the Deutsche Forschungsgemeinschaft under grants Me 620/6-1 and SFB 124, Teilprojekt B2

** A preliminary version of this research was presented at the ACM Symposium on Computational Geometry, Baltimore, 1985

1. Introduction:

The problem of locating a key in many ordered lists arises frequently in computational geometry, e.g. when searching in range or segment trees. Several researchers, e.g. Vaishnavi/Wood [VW82], Willard [W85], Edelsbrunner/Guibas/Stolfi [EGS], Imai/Asano [IA84], Lipski [L84] observed that the naive strategy of locating the key seperatedly in each list by binary search is far from optimal and that more efficient techniques frequently exist. Chazelle/Guibas [CG85] distilled from these special case solutions a general data structuring technique and called it **fractional cascading**. They describe the problem as follows.

Let U be an ordered set and let $G = (V, E)$ be an undirected graph. Assume also that for each vertex $v \in V$ there is a set $C(v) \subseteq U$, called the **catalogue** of v, and that for every edge $e \in E$ there is given a **range** $R(e) = [\ell(e), r(e)] \subseteq U$. We use $N = \sum_{v \in V} |C(v)|$ to denote the total size of the catalogues. The goal is organize the catalogues in a data structure such that the following operations are supported efficiently.

1. **Query:** The input to a query is an arbitrary element $k \in U$ and a connected subtree $G' = (V', E')$ of G such that $k \in R(e)$ for all $e \in E'$. We use n to denote $|V'|$. The output of the query is for each vertex $v \in V'$ an element $x \in C(v)$ such that "predecessor of x in $C(v)$" $< k \leq x$ i.e. the query locates k in each list $C(v), v \in V'$.

   Chazelle and Guibas [CG85] have shown that fractional cascading supports queries in time $O(n + \log N)$ provided that G has **locally bounded degree**, i.e. there is a constant d such that $\forall v \in V$ and $\forall k \in U$ there are at most d edges $e = (v, w)$ with $k \in R(e)$. We will assume throughout this paper that G has locally bounded degree. In the present paper we are also interested in insertions into and deletions from the catalogues.

2. **Deletion:** Given key $k \in C(v)$ and its position in $C(v)$, delete k from $C(v)$.

3. **Insertion:** Given $k \in U$ and an element $x \in C(v)$ such that "predecessor of x in $C(v)$" $< k \leq x$, insert k into $C(v)$.

In sections 2 to 5 we will prove the main main result of this paper:
We can support queries in time $O(\log(N + |E|) + n \log\log(N + |E|))$ and insertions and deletions in $O(\log\log(N + |E|))$. The bound for the queries is worst case and the bound for the insertions and deletions is amortized. If only insertions or only deletions have to be supported then the bounds reduce to $O(\log(N + |E|) + n)$ and $O(1)$ respectively.

In section 2 we briefly recapitulate the paper of Chazelle and Guibas [CG85]. In section 3 we give the insertion and deletion algorithms and analyse their amortized behavior in

section 4. The amortized analysis is based on the bank account paradigm (cf. [M84a]) and extends the amortized analysis of (a,b) - trees [HM82]. A novel feature of our analysis is the fact that the tokens used for the accounts change their value over time. The insertion and deletion algorithms make use of an efficient data structure for maintaining an interval partition of a linear list. Mor precisely, if B is a linear list of n items, some of which are marked we will consider the following five operations on it:

FIND input : a pointer to some item x
output: a pointer to a marked item y such that all items between x and y are unmarked

ADD input : a pointer to some item x
effect : adds a unmarked item immediately before x to the list

ERASE input : a pointer to a unmarked item x
effect : deletes x from the list

UNION input : a pointer to a marked item x
effect : unmarks item x

SPLIT input : a pointer to a unmarked item x
effect : marks item x

We will prove that FIND, UNION, SPLIT, ADD and ERASE can be supported in time $O(\log\log n)$ per operation and space $O(n)$. The time bound is worst case for FIND, UNION and SPLIT and amortized for ADD and ERASE.

The data structure used to represent interval partitions of linear lists is derived from the $O(\log\log N)$ - priority queue of v. Emde Boas [EKZ77] and is described in section 5. Imai/Asano [IA84] have shown that operations FIND, SPLIT and ADD or FIND, UNION and ERASE can be supported in amortized time $O(1)$. This explains why the amortized time bounds can be reduced to $O(\log(N+|E|)+n)$ for queries and $O(1)$ for updates if only insertions or only deletions have to be supported.

As mentioned above, fractional cascading emerged from the techniques developed for segment and range trees. Applying our general solution to these tree structures we obtain versions of segment and range trees with query, insertion and deletion time $O(\log n \log\log n)$ (n is the number of segments or points). The best previous solution was $O(\log^{\frac{3}{4}} n)$ in [W85]. The application to segment and range trees is described in section 6. Finally section 7 offers some conclusions and open problems.

2. Fractional Cascading

In this section we describe the basic fractional cascading data structure and the algorithms for the static case as introduced by Chazelle and Guibas in [CG85].

At each node v of the catalogue graph G we store a multi-set $A(v) \supseteq C(v)$ as a doubly linked linear list. $A(v)$ is called the **augmented catalogue** at v. The elements in $C(v)$ are called **proper**, those in $A(v) - C(v)$ **non-proper**. Each $x \in A(v)$ is implemented by a record consisting of the following components:

key : a value from U

next : pointer to the successor in $A(v)$

pred : pointer to the predecessor in $A(v)$

target : a node of G incident to v

pointer : a pointer to an element in $A(x.\text{target})$

count : integer

in_S : boolean

kind : (proper, non-proper)

Note that we will often use x in the sense x.key and sometimes we denote by x a pointer to the element x in a certain catalogue. But the meaning will always be clear from the context.

The non-proper elements are used to guide the search between incident catalogues $A(v)$ and $A(w)$, $(v,w) \in E$. To this purpose the entries target and pointer of each non-proper element x have the following meaning (they are not defined for proper elements):
Let $x \in A(v) - C(v)$. If x.target $= w \in V$ then $e = (v,w) \in E$ and x.key $\in R(e)$. If x.pointer $= y$ then y is a non-proper element in $A(x.\text{target})$, x.key $= y$.key and y.pointer $= x$. Thus x.pointer.pointer $= x$ for every non-proper element x.

Let $x \in A(v) - C(v)$ and $y = x.\text{pointer} \in A(w) - C(w)$. The pair (x,y) is called a **bridge** between $A(v)$ and $A(w)$. If (x,y) is a bridge between $A(v)$ and $A(w)$ then

$x = y.\text{pointer}$

$y = x$.pointer
x.target $= w$
y.target $= v$
x.key $= y$.key
x.kind $= y$.kind $=$ non-proper
The field in_S is used to indicate that a bridge (x, y) belongs to a certain set S that will be defined later (section 3).

For each edge $e = (v, w)$ there always exist the two bridges (x_ℓ, y_ℓ) and (x_r, y_r) with x_ℓ.key $= y_\ell$.key $= \ell(e)$ and x_r.key $= y_r$.key $= r(e)$. We denote these two extreme bridges between $A(v)$ and $A(w)$ by $\ell(e)$-bridge and $r(e)$-bridge respectively. The bridges connecting two augmented catalogues $A(v)$ and $A(w)$ divide the interval $[\ell(e), r(e)]$ into **blocks**. Each pair of neighboring bridges $(x', y'), (x, y)$ defines such a block $B(x, y) \subseteq A(v) \cup A(w)$ (cf. figure 1). To identify a block we use the right bridge (x, y) of the pair. Note that the $\ell(e)$ - bridge does not identify any block.

More formally: Let $v \in V$, $x \in A(v) - C(v)$, x.target $= w \in V$ and

$$x' = \max\{y \in A(v) - C(v) \mid y < x \text{ and } y.\text{target} = w\}$$

i.e. $(x', x'.\text{pointer})$ and $(x, x.\text{pointer})$ are adjacent bridges between $A(v)$ and $A(w)$ (cf. figure 1). Then we define

$$I(x) :=]x', x[\cap A(v)$$

where $]x', x[$ denotes the open interval with endpoints x' and x. Each bridge (x, y) except the $\ell(e)$-bridge is used to identify the block

$$B(x, y) := I(x) \cup I(y)$$

The counters x.count and y.count give the size of the intervals $I(x)$ and $I(y)$ i.e.

$$|I(x)| = x.\text{count}$$
$$|I(y)| = y.\text{count}$$
$$|B(x, y)| = x.\text{count} + y.\text{count}$$

The block sizes are crucial for the efficiency of the data structure. The bigger the blocks are, the fewer bridges (non-proper elements) are necessary and the space requirement will be smaller. However (as we will see soon) to guarantee an efficient search algorithm the blocks must not exceed a certain size. For this reason we postulate that all blocks fulfill the following invariant:

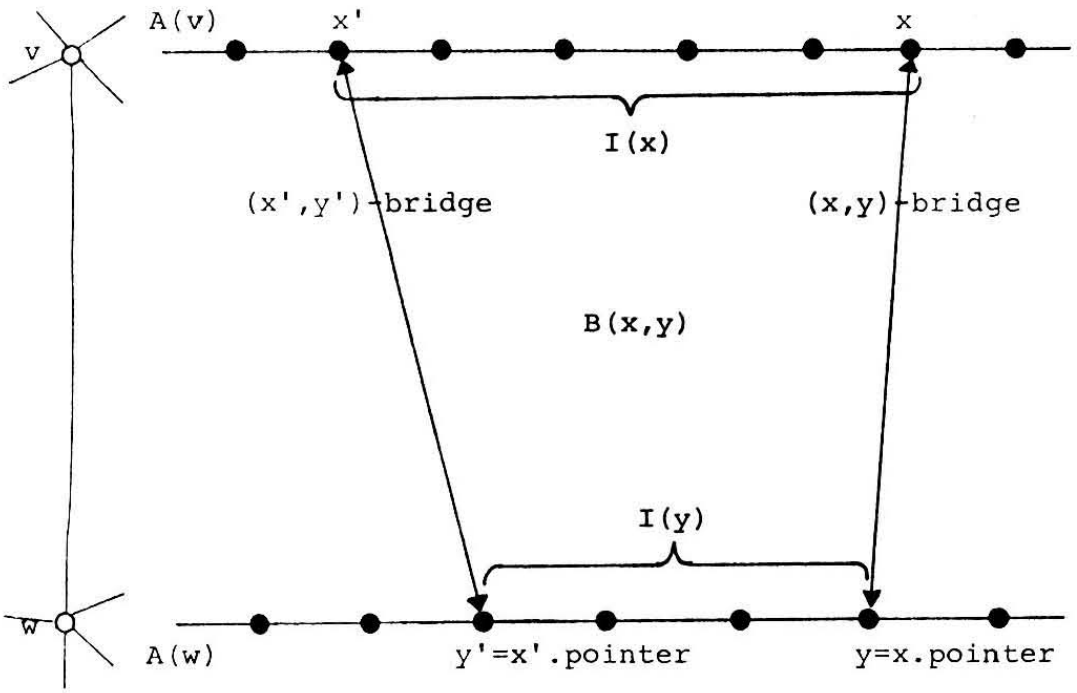

Figure 1

Invariant 1. There are two constants a, b with $a \leq b$ such that for all blocks $B(x,y)$ holds

1. $|B(x,y)| \leq b$
2. $|B(x,y)| \geq a$ or $B(x,y)$ is the only block between $A(y.\text{target})$ and $A(x.\text{target})$. ∎

Invariant 1 generalizes the structural invariant of (a,b)-trees (cf. [M84a]) to fractional cascading.

The following two lemmas show that invariant 1 ensures both linear space and an efficient search algorithm.

Lemma 1. Let $N = \sum_{v \in V} |C(v)|$, $S = \sum_{v \in V} |A(v)|$ and $a \geq 3d$. Then

$$S \leq 3N + 12|E|.$$

Proof:
S is clearly equal to N plus twice the total number of bridges. We have to count the number of bridges. Let $Br(v,w)$ denote the number of bridges between $A(v)$ and $A(w)$.

These bridges define exactly $Br(v,w)-1$ blocks. All but one block contain at least a elements and hence

$$a(Br(v,w)-1) \leq |A(v) \cap R(v,w)| + |A(w) \cap R(v,w)|$$

That means

$$Br(v,w) \leq \frac{1}{a}(|A(v) \cap R(v,w)| + |A(w) \cap R(v,w)|) + 1$$

Thus we have

$$S = N + 2 \sum_{(v,w)\in E} Br(v,w)$$

$$\leq N + \frac{2}{a}\left(\sum_{(v,w)\in E} |A(v) \cap R(v,w)| + \sum_{(v,w)\in E} |A(w) \cap R(v,w)| \right) + 4|E|$$

$$\leq N + \frac{d}{a}\left(\sum_{v\in V} |A(v)| + \sum_{w\in V} |A(w)| \right) + 4|E|$$

since every $x \in A(v)$ is contained in $R(v,w)$ for at most d vertices w

$$= N + \frac{2d}{a}S + 4|E|$$

And finally

$$S \leq \frac{a}{a-2d}(N + 4|E|)$$

$$\leq 3N + 12|E| \quad \text{if } a \geq 3d$$

■

Lemma 2. Given a search key $k \in U$, an element $x \in A(v)$ with $x.\text{pred} < k \leq x$ and an edge $e = (v,w)$ with $k \in R(e)$, the position of k in $A(w)$ can be found in constant time, i.e. $y \in A(w)$ with $y.\text{pred} < k \leq y$ can be computed in time $O(1)$.

Proof:
We proof lemma 2 by giving a constant time algorithm for searching.

Algorithm 1:

1. **function** SEARCH (k, x, w);
(* precondition: $x \in A(v)$ with $x.\text{pred} < k \leq x$, $(v, w) \in E$ and $k \in R(v, w)$
result: $y \in A(w)$ with $y.\text{pred.key} < k \leq y.\text{key}$
running time: $O(1)$
*)
2. **while** $x.\text{target} \neq w$ **do** $x \leftarrow x.\text{next}$ **od** ;
3. $y \leftarrow x.\text{pointer}$;
4. **while** $y.\text{pred.key} \geq k$ **do** $y \leftarrow y.\text{pred}$ **od**;
5. **return**(y);

The running time of SEARCH is at most the size of the block between $A(v)$ and $A(w)$ containing x which is bounded by the constant b (invariant 1). ∎

The above algorithm locates key k in an augmented catalogue $A(w)$, i.e.

$$\text{SEARCH}(k, x, w) = y \in A(w) \text{ with } y.\text{pred.key} < k \leq y.\text{key}$$

But what we want to find is the position of k in the original catalogue $C(w)$, i.e. we have to compute an element $y' \in C(w)$ with $y' = \min\{z \in C(w) | z \geq y\}$. For this reason a function FIND is defined that gives for any element x of an augmented catalogue $A(v)$ its successor y in the corresponding original catalogue $C(v)$. For any $x \in A(v)$

$$\text{FIND}(x) := \min\{y \in C(v) | y \geq x\}$$

For the dynamic case (see next section) we also have to provide operations for inserting or deleting proper and non-proper elements. In section 5 a data structure is presented that supports efficiently the following five operations on a linear list of N items (each item has a mark $\in$ {proper, non-proper}).

FIND input : a pointer to some item x
output : a pointer to a proper item y such that all items between x and y are non-proper

ADD input : a pointer to some item x
effect : adds a non-proper item immediately before x to the list

ERASE input : a pointer to a non-proper item x
effect : deletes x from the list

UNION input : a pointer to a proper item x
effect : changes the mark of x to non-proper

SPLIT input : a pointer to a non-proper item x
effect : changes the mark of x to proper

The names of the last two operations are suggested by the observation that the non-proper items partition the linear list into a family of intervals. Then operation UNION merges two adjacent intervals and SPLIT splits an interval. We will show in section 5 how to implement all five operations in space $O(N)$ and in time $O(\log\log N)$ (worst case for FIND, UNION, SPLIT and amortized for ADD, ERASE).

Remarks:

1. In the static case only operation FIND is needed. It can be realized by giving each non-proper item a pointer to its proper successor. Thus FIND has cost $O(1)$ in this case.

2. In the semi-dynamic case (only insertions or only deletions have to be supported) operations FIND, SPLIT and ADD resp. FIND, UNION and ERASE have to be implemented. Gabow/Tarjan [GT83] and Imai/Asano [IA84] show how to do this in amortized time $O(1)$.

Theorem 1. Let $k \in U$ and $G' = (V', E')$ be a subtree of G such that $k \in R(e)$ for all $e \in E'$. Then k can be located in $C(v)$ for all $v \in V'$ in time $O(\log(N+|E|)+n\log\log(N+|E|))$ where $n = |V'|$ and $N = \sum_{v \in V} |C(v)|$.

Proof:
Let $v_0 \in V'$. We first locate k in $A(v_0)$ by binary search in time $O(\log(|A(v_0)|)) = O(\log(N+|E|))$ (remember that $|A(v)| \leq O(N+|E|)$ for every $v \in V$ by lemma 1).Knowing the position of k in $A(v_0)$ it can be located in all $A(v), v \in V'$ by procedure SEARCH in time $O(n)$. To find the position of k in the original catalogues $C(v)$ FIND must be called once for every $A(v), v \in V'$. Thus the total cost for a search in G' is $O(\log(N + |E|) + n\log\log(N + |E|))$. ∎

3. Dynamization

In this section we describe how to delete elements from the catalogues and how to insert new elements into the catalogues without increasing search time and space requirement. In particular we give algorithms for insertion, deletion and rebalancing the block sizes.

Insertions and deletions may result in blocks $B(x,y)$ violating invariant 1 i.e. $|B(x,y)| > b$ or $|B(x,y)| < a$. We store these blocks out of balance in a set or list S. To check whether a block $B(x,y)$ must be entered into S we use the counters x.count and y.count. The sum of these two counters will always indicate the size of block $B(x,y)$ if $B(x,y)$ is not in S. More precisely we maintain the following invariant:

Invariant 2. Let (x,y) be a bridge with $B(x,y) \notin S$.

Then $|B(x,y)| = x.\text{count} + y.\text{count}$ and $a \leq |B(x,y)| \leq b$ (invariant 1) ∎

Remarks:

a) The values of x.count and y.count are irrelevant if $B(x,y) \in S$

b) If $|B(x,y)| \notin [a..b]$ then $B(x,y) \in S$.

c) There can exist blocks $B(x,y)$ with $B(x,y) \in S$ and $|B(x,y)| \in [a..b]$. These are blocks which have been modified by insertions as well as deletions.

Now we give the algorithm to insert an element x into the augmented catalogue $A(v)$.

Algorithm 2:

```
1. procedure INSERT (x, y0);
(* precondition: y0 ∈ A(v), y0.pred.key < x.key ≤ y0.key
   effect: x is inserted immediately before y0 into A(v)
   running time: O(log lg N), amortized
*)
2. ADD(x, y0)
3. if x.kind = proper then SPLIT(x, y0) fi;
4. insert x into the doubly linked list A(v) before y0
5. y ← y0
6. A ← ∅;
7. do b times
8.     w ← y.target;
```

```
9.    if (y.kind = non-proper) and (w ∉ A) and (x.key ∈ R(v, w))
10.   then A ← A ∪ {w};
11.        y.count ← y.count + 1;
12.        z ← y.pointer;
13.        if (y.in_S = false) and (y.count + z.count > b)
14.        then S ← S ∪ {B(y, z)};
15.             y.in_S ← true;
16.             z.in_S ← true;
17.        fi;
18.   fi;
19.   y ← y.next;
20. od;
```

A call to procedure INSERT needs amortized time $O(\log\log N)$ for the calls to ADD and maybe SPLIT (see section 5) and $O(b)$ time for the loop (lines 8-21).

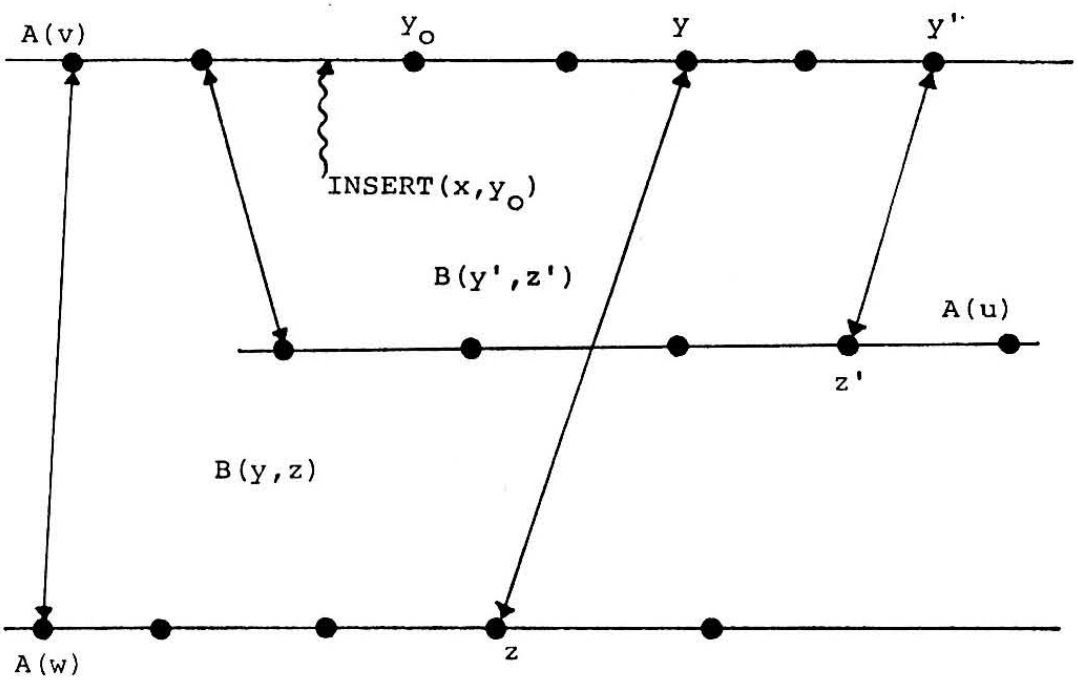

Figure 2

Lemma 3. Procedure INSERT preserves invariant 2.

Proof:
The algorithm inserts x directly before y_0 into $A(v)$. Thus the size of some blocks (at most d) is increased whose identifying bridges (y, z) lie to the right of x (see figure 2).

case 1 : $B(y,z) \in S$ before the execution of INSERT. Then there is nothing to do. (Note: Nevertheless the algorithm possibly increases y.count, but this is irrelevant)

case 2 : $B(y,z) \notin S$ before the execution of INSERT. This implies $|B(y,z)| \leq b$. Therefore y can be reached in b steps starting from y_0 and the counter y.count gets the correct value. Thus block $B(y,z)$ enters S if and only if its size exceeds b. Furthermore the algorithm stores already examined neighbors $w \in V$ of node v in the set A to ensure that for every edge (v,w) at most one block is treated. ∎

The algorithm to delete an element x from $A(v)$ follows.

Algorithm 3:

```
 1. procedure DELETE(x);
(* precondition: x ∈ A(v)
   effect: x is deleted
   running time: O(log log N), amortized
*)
 2. if x.kind = proper then UNION(x) fi;
 3. ERASE(x)
 4. remove x from the doubly linked list A(v)
 5. y ← x.next;
 6. A ← ∅;
 7. do b times
 8.     w ← y.target;
 9.     if (y.kind = non-proper) and (w ∉ A) and (x.key ∈ R(v,w))
10.     then A ← A ∪ {w};
11.          y.count ← y.count + 1;
12.          z ← y.pointer;
13.          if (y.in_S = false) and (y.count + z.count < a) and
14.            (B(y,z) is not the only block between v and w))
15.          then S ← S ∪ {(y,z)};
16.               y.in_S ← true;
17.               z.in_S ← true;
18.          fi;
19.     fi;
20.     y ← y.next;
```

21. **od**;

Procedure DELETE needs time $O(b + \log\log N)$.

Lemma 4. Procedure DELETE preserves invariant 2.

Proof:
see proof of lemma 3. ∎

In order to guarantee the time and space bounds of lemma 3 and lemma 4 in section 2 we have to restore invariant 1 after each call to DELETE or INSERT by calling a procedure REBAL. Note that all blocks violating invariant 1 are in the set S (but not all blocks in S violate it). Now we first describe informally the rebalancing procedure and then give the algorithm REBAL.

REBAL has to examine every block $B(x,y) \in S$. First it must compute the size ℓ of $B(x,y)$. This can easily be done by running to the next parallel bridge starting at (x,y) in time $O(\ell)$. Knowing the block size ℓ there are three different cases:

case 1 : $\ell > b$ i.e. block $B(x,y)$ is too large

In this case the algorithm will divide $B(x,y)$ into $\lfloor 3\ell/b \rfloor + 1$ parts sized as equal as possible. This can be done by inserting $\lfloor 3\ell/b \rfloor$ bridges (i.e. inserting $2 \times \lfloor 3\ell/b \rfloor$ non-proper elements). Then the size of the resulting sub-blocks is between $\frac{b}{4}$ and $\frac{b}{3}$ and fulfills invariant 1 if we choose $a \leq \frac{b}{4}$. The cost for this case is the cost for computing the block size ($O(\ell)$) plus the cost for inserting the new bridges ($\frac{6\ell}{b}$ calls of Insert).

case 2 : $\ell < a$ i.e. block $B(x,y)$ is too small

Here $B(x,y)$ is concatenated with its right (left) neighbor block by deleting the bridge (x,y) (deleting two non-proper elements). Then we have to check whether the neighbor block is already in S or not. To do this we scan the neighbor block until its identifying bridge (x',y') is reached and check the in_S-flag x'.in_S. But if (x',y') is not reached within b steps we can stop this scanning procedure. Then we know that the neighbor block is larger than b and therefore a member of S. This means we can test wether $B(x',y')$ is in S in time $O(b)$. If $B(x',y') \notin S$ then its new size is computed by addition of the counters x.count and x'.count resp. y.count and y'.count. In the case that this size exceeds b $B(x',y')$ is added to S. The immediate cost for this case is the cost for computing ℓ ($O(\ell)$), the cost for scanning the neighbor block ($O(b)$) and the cost for deleting two non-proper elements (2 calls of DELETE).

case 3 : $a \leq \ell \leq b$ i.e. block $B(x,y)$ **has correct size**

In this case $B(x,y)$ can be removed from S without any modification. The cost is only the cost for computing the block size : ℓ

The following procedure REBAL uses in the overflow case a subroutine **DIVIDE**(B,k) which divides block B into k subblocks differing in size by at most 1. This routine simply inserts $k+1$ new bridges by calling $2(k+1)$ times procedure INSERT. The program for DIVIDE is straightforward and left to the interested reader.

Algorithm 4:

```
1.  procedure REBAL;
2.  while S ≠ ∅
3.  do let B(x,y) be an arbitrary element of S;
4.      S ← S − {B(x,y)};
5.      x.in_S ← false;
6.      y.in_S ← false;
        (* compute real size ℓ of B(x,y) *)
7.      xlcount ← 0;
8.      xl ← x.pred;
9.      while xl.target ≠ x.target
10.     do xl ← xl.pred;
11.         xlcount ← xlcount + 1;
12.     od;
13.     ylcount ← 0;
14.     yl ← y.pred;
15.     while yl ≠ xl.pointer
16.     do yl ← yl.pred;
17.         ylcount ← ylcount + 1;
18.     od;
19.     ℓ ← xlcount + ylcount;  (* real size of B(x,y) *)
20.     if ℓ > b
21.     then (* case 1: overflow ⟶ divide B(x,y) into 3ℓ/b parts*)
22.          DIVIDE(B(x,y), 3ℓ/b);
23.     else if ℓ < a
24.          then (* case 2: underflow ⟶ concatenate B(x,y) with right neighbor *)
25.               xr ← x.next;
26.               yr ← y.next;
27.               DELETE(x);
28.               DELETE(y);
```

```
                (* scan right neighbor block *)
29.             xrcount ← 0;
30.             while (xr.target ≠ x.target) and (xrcount ≤ b)
31.             do xr ← xr.next;
32.                xrcount ← xrcount + 1;
33.             od;
34.             yrcount ← 0;
35.             while (yr.target ≠ y.target) and (yrcount ≤ b)
36.             do yr ← yr.next;
37.                yrcount ← yrcount + 1;
38.             od;
39.             if (xr.target ≠ x.target) or (yr.target ≠ y.target)
40.             then (* right neighbor is larger than b  i.e. is a member of S *)
41.                  do nothing
42.             else if xr.in_S = true
43.                  then (* right neighbor is in S *)
44.                       do nothing
45.                  else (* right neighbor is not in S *)
46.                       xr.count ← xrcount + xlcount;
47.                       yr.count ← yrcount + ylcount;
48.                       if (xrcount + yrcount) > b
49.                       then xr.in_S ← true;
50.                            yr.in_S ← true;
51.                            S ← S ∪ {B(xr, yr)};
52.                       fi;
53.                  fi;
54.             fi;
55.        else (* a ≤ ℓ ≤ b block has correct size *)
56.             x.count ← xlcount;
57.             y.count ← ylcount;
58.        fi;
59.   fi;
60. od;
```

4. Analysis of the Dynamic Behavior

We will show in this section that the amortized cost for one update operation (insert or delete) is $O(\log\log N)$ where N is the current size of our data structure i.e. the number of proper elements. More precisely we show that a sequence of m update operations can be executed in time $O(|E| + \sum_{i=1}^{m} \log\log(N_i + |E|))$ where N_i is the number of proper elements before the ith operation. We prove this result using the bank account paradigm (cf. [HM82], [M84a] or [T85]), i.e. we associate an account with the data structure. Each update operation puts $O(1)$ tokens into this account where each token represents the ability to pay for $O(b + \log\log(N + |E|))$ units of computing time. The actual computing time required for the update operation is paid out of the account and the goal is to show that the balance of the account always stays non-negative. As in the amortized analysis of balanced trees (cf. [HM82]) the account is conceptually split into many small accounts, essentially one for each block $B(x,y)$. The number of tokens in the account of block $B(x,y)$ measures the criticality of block $B(x,y)$, i.e. the closer $|B(x,y)|$ is to b or a the larger is the number of tokens in the account. There is also a novel feature in the analysis. The value of the tokens changes over time. Since we will use the tokens to pay for the list operations UNION, SPLIT, ADD and ERASE on augmented catalogues the tokens must suffice to pay for the cost of these operations when they are executed. The actual size of the catalogues at that point of time differs in general from the size of the catalogues, when the tokens where deposited in the account. Hence the value of the tokens must change over time. We will next give a precise statement of the result:

We start with an "empty" data structure D_{-1} with

1. $C(v) = \emptyset$ for all $v \in V$
2. $A(v) = \cup_{(v,w)\in E}\{l(v,w), r(v,w)\}$
3. S = set of all blocks that violate invariant1.

First D_{-1} is transformed by procedure REBAL to a balanced data structure D_0. We will next execute a sequence of m insert/delete operations starting with D_0. The i-th operation is realized by a call of procedure INSERT or DELETE, which brings us from data structure D_{2i-2} to structure D_{2i-1}, and a subsequent call of REBAL, which brings us from D_{2i-1} to D_{2i}.

$$D_{-1} \text{—} rebal_0 \longrightarrow D_0 \text{—} op_1 \longrightarrow D_1 \text{—} rebal_1 \longrightarrow D_2 \text{—} op2 \longrightarrow D_3 \text{—} \cdots \longrightarrow D_{2m}$$

$op_i \in \{INSERT, DELETE\}$ for $i = 1..m$.

Let $cost(D_j \rightarrow D_{j+1})$ denote the cost of going from D_j to D_{j+1}. Then the total cost of

the first m insert and delete operations is the sum of the costs of the calls of procedures INSERT and DELETE and the total rebalancing cost

$$O\Big(\sum_{i=1}^{m-1} cost(D_{2i} \to D_{2i+1}) + \sum_{i=-1}^{m-1} cost(D_{2i+1} \to D_{2i+2})\Big)$$

We show

Theorem 2. The total cost of the first m insert/delete operations is

$$O\Big(\sum_{i=1}^{m} \log\log(N_i + |E|) + |E|\Big)$$

where N_i is the number of proper elements before the i-th operation (i.e. the number of proper elements in D_{2i}).

Proof:
We go from structure D_{2i} to D_{2i+1} by a call of either INSERT or DELETE. The cost of a call of INSERT or DELETE is $O(b)$ plus the cost of ADD and SPLIT (UNION and ERASE) respectively. The list operations are applied to augmented catalogues of size at most $O(N_i + |E|)$. (Note that the total size of all augmented catalogues is $O(N_i + |E|)$ by lemma 1 and hence the size of every single catalogue is bounded by this quantity). Let $listop(n)$ denote the cost of a list operation on a list of length n. Then we have

$$\sum_{i=0}^{m-1} cost(D_{2i} \to D_{2i+1}) = O\Big(\sum_{i=0}^{m-1} (b + listop(N_i + |E|))\Big)$$

From D_{2i+1} to D_{2i+2} we go by calling procedure REBAL. This procedure repeatedly removes blocks from the queue S, i.e. a more detailed view is:

$$D_{2i+1} = D'_0 \longrightarrow D'_1 \longrightarrow D'_2 \longrightarrow D'_3 \longrightarrow \cdots \longrightarrow D'_{k_i} = D_{2i+2},$$

where REBAL removes a block of size ℓ_j from queue S, when going from D'_j to D'_{j+1}. This costs

$$O\left(\ell_j + \begin{cases} \ell_j/b \text{ list operations}, & \text{if } \ell_j > b \\ 1 \text{ list operation}, & \text{if } \ell_j < a \\ 0 \text{ list operations}, & \text{if } a \le \ell_j \le b \end{cases}\right)$$

All these list operations are executed on lists of length at most $O(N_i + |E|)$. So

$$\sum_{i=-1}^{m-1} cost(D_{2i+1} \to D_{2i+2}) \quad \text{is bounded by}$$

$$O\left(\sum_{i=-1}^{m-1}\sum_{j=0}^{k_i-1}\left(\ell_j + \begin{cases} \ell_j/b \times \mathit{listop}(N_i+|E|), & \text{if } \ell_j > b \\ \mathit{listop}(N_i+|E|), & \text{if } \ell_j < a \end{cases}\right)\right)$$

We can now invoke theorem 3 from section 5. Let k be the total number of list operations that are executed when going from D_{-1} to D_{2m}. Then, by theorem 3 of section 5, the cost of these operations is bounded by $\sum_{j=1}^{k} \log\log s_j$,where s_j is the size of the lists before the j-th list operation. If we therefore replace $\mathit{listop}(N_i+|E|)$ by $\log\log(N_i+|E|)$ in the expression above then we have a bound for the total cost of all list operations. Our goal is therefore to bound

$$(*) \qquad O\left(\sum_{i=-1}^{m-1}\sum_{j=0}^{k_i-1}\left(\ell_j + \begin{cases} \ell_j/b \times \log\log(N_i+|E|), & \text{if } \ell_j > b \\ \log\log(N_i+|E|), & \text{if } \ell_j < a \end{cases}\right)\right)$$

Remark: Note that theorem 3 gives us only a bound on the cost of a sequence of list operations, i.e. an amortized bound. But that's all we need for the current proof. ∎

Now we are going to prove an upper bound for the expression (*) using the bank account paradigm (cf. [HM82], [M84a] or [T85]). First we define a function

$$\textbf{bal} : \text{"state of the data structure"} \longrightarrow \mathbb{N}_0$$

that will indicate to what extend the data structure is out of balance. The bal-function is defined as the product

$$bal(D) = f(D) \cdot (b + \log\log(N+|E|))$$

Here $f(D)$ is the number of tokens in the account and the second term gives the current value of one token.

Definition 1. Let a', b' be two constants, $a \le a' \le b' \le b$, satisfying the following conditions (the reason for these requirements will become clear lateron):

$$b/4 \ge a' \tag{1}$$

$$b/3 \le b' \tag{2}$$

$$2b' \le b - 12d - 6 \tag{3}$$

$$a' \ge 2a + 2d + 2 \tag{4}$$

Possible values for a, a', b' and b are:

$$\begin{aligned} a &= 3d \\ a' &= 9d + 4 \\ b' &= 12d + 6 \\ b &= 36d + 18 \end{aligned}$$

Then we define for every block B

$\Delta(B) := \max(0, |B| - b') + \max(0, a' - |B|)$ and

$$f(D) = 2\Big(\sum_{B \notin S} \Delta(B) + \sum_{B \in S^+} \max\Big(b - b' + 1, \Delta(B)\Big) + \sum_{B \in S^-} \max\left(a' - a + 1, \Delta(B)\right)\Big)$$

And finally $bal(D) := f(D)(b + \log\log(N + |E|))$

Here (S^+, S^-) is a partition of S with
S^+ = set of all blocks that came into S because they were too large ($|B| > b$)
S^- = set of all blocks that came into S because they were too small ($|B| < a$)

N is the number of proper elements in D. Note that N is increased by insertions and decreased by deletions. The next lemmas show three very important properties of function *bal*.

Lemma 5. Let $op \in$ {INSERT,DELETE} and let $op(D)$ be the data structure D modified by a call of procedure *op* then

a) $$bal(op(D)) \leq bal(D) + 2d(b + \log\log(N + |E|)) + O(1)$$

Here N is the number of proper elements in D.

b) $$bal(D_{2i+1}) \leq bal(D_{2i}) + 2d(b + \log\log(N_i + |E|)) + O(1)$$

Proof:
Part b) follows immediately from part a).

We prove part a)
case 1: op = INSERT. Insertion of a new element x into the catalogue $A(v)$ can increase

the size of at most d blocks by 1, because there are at most d edges $e = (v, w) \in E$ with $x \in R(e)$. Furthermore the total number of proper elements is raised to $N + 1$.

Thus we have

$$f(op(D)) \leq f(D) + 2d$$

Let $N' = N + |E|$. Then

$$bal(op(D)) - bal(D) \leq (f(D) + 2d)(b + \log\log(N' + 1)) - f(D)(b + \log\log N')$$

$$= 2d(b + \log\log(N' + 1)) + f(D)(\log\log(N' + 1) - \log\log N')$$

$$= 2d(b + \log\log N') + (f(D) + 2d)(\log\log(N' + 1) - \log\log N')$$

What remains to show is that $(f(D) + 2d)(\log\log(N' + 1) - \log\log N') = O(1)$.

Since $f(D) + 2d = O(N')$ (note that the number of blocks is bounded by $O(N')$) we have

$$(f(D) + 2d)(\log\log(N' + 1) - \log\log N') = O(N'(\log\log(N' + 1) - \log\log N'))$$

$$= O(N'(\ln\ln(N' + 1) - \ln\ln N'))$$

and $$N'(\ln\ln(N' + 1) - \ln\ln N') = N' \ln \frac{\ln(N' + 1)}{\ln N'}$$

$$= N' \ln \left(1 + \left(\frac{\ln(N' + 1)}{\ln N'} - 1\right)\right)$$

$$\leq N' \left(\frac{\ln(N' + 1)}{\ln N'} - 1\right)$$

since $\ln(1 + x) \leq x$ for all $x \in \mathbb{R}$

$$= N' \left(\frac{\ln\left(N'\left(1 + \frac{1}{N'}\right)\right)}{\ln N'} - 1\right)$$

$$= N' \left(\frac{\ln N' + \ln\left(1 + \frac{1}{N'}\right)}{\ln N'} - 1\right)$$

$$\leq N' \left(1 + \frac{\frac{1}{N'}}{\ln N'} - 1\right)$$

$$= \frac{1}{\ln N'}$$

$$\leq 1 \qquad \text{for } N' \geq e$$

case 2: op = DELETE
For the same reason as above the size of at most d blocks can be decreased by one i.e.

$$f(op(D)) \leq f(D) + 2d$$

Thus $bal(op(D)) = (f(D) + 2d)(b + \log\log(N + |E| - 1))$

$$= f(D)(b + \log\log(N + |E| - 1)) + 2d(b + \log\log(N + |E| - 1))$$

$$\leq bal(D) + 2d(b + \log\log(N + |E|))$$

■

Lemma 6. $cost(D_{2i+1} \longrightarrow D_{2i+2}) \leq O(bal(D_{2i+1}) - bal(D_{2i+2}))$ for $-1 \leq i \leq m-1$.

Proof:
During the execution of $rebal_i$ data structure D_{2i-1} is transformed by removing blocks from S step by step into the data structure D_{2i}.

$$D_{2i+1} = D'_0 \longrightarrow D'_1 \longrightarrow D'_2 \longrightarrow D'_3 \longrightarrow \cdots \longrightarrow D_{2i+2}$$

In each step one element is removed from S. We show an upper bound for the cost of the i-th step

$$cost(D'_{i-1} \longrightarrow D'_i) \;\leq\; O(bal(D'_{i-1}) - bal(D'_i))$$

and thereby prove the lemma. Let B be the block removed from S in step i and $\ell = |B|$. There are 3 different cases:

case 1: B is bigger than b, i.e. $\ell > b$

B is divided into subblocks of size between $\frac{1}{4}b$ and $\frac{1}{3}b$. Thus by requirement **(1)** and **(2)** the resulting new blocks make no contribution to $bal(D'_i)$). The value of *bal* is decreased by $2(l - b')(b + \log\log(N + |E|))$ because block B is destroyed and increased by $2d\frac{6\ell}{b}(b + \log\log(N + |E|))$ by the insertion of the new bridges. Thus

$$bal(D'_i) = bal(D'_{i-1}) - 2(\ell - b' + d\frac{6\ell}{b})(b + \log\log(N + |E|))$$

The cost for the splitting of B is $O(\ell + \frac{6\ell}{b}(b + \log\log(N + |E|)))$, since we have to perform $6\ell/b$ list operations. What remains to show is

$$\ell + \frac{6\ell}{b}(b + \log\log(N + |E|)) \leq (2(\ell - b') - 2d\frac{6\ell}{b})(b + \log\log(N + |E|))$$

or $$\ell + \frac{6\ell}{b} \leq 2\ell - 2b' - 2d\frac{6\ell}{b}$$

or $$(2d + 1)\frac{6\ell}{b} + \ell \leq 2\ell - 2b'$$

or $$(\frac{12d + 6}{b} + 1)\ell \leq 2\ell - 2b'$$

or $$(1 - \frac{12d + 6}{b})\ell \geq 2b'$$

or $$(1 - \frac{12d + 6}{b})b \geq 2b' \quad (\text{since } \ell > b)$$

or $b - 12d - 6 \geq 2b'$. This holds by requirement **(3)** !

case 2: B is smaller than a i.e. $\ell < a$

In this case the value of the *bal*-function is decreased by $2(b + \log\log(N + |E|))(a' - \ell)$ (removal of B) and increased by $2(b + \log\log(N + |E|))(\ell + 2d)$ (enlargement of the neighbor block and deletion of one bridge). Thus

$$bal(D'_i) = bal(D'_{i-1}) + 2(b + \log\log(N + |E|))(2\ell - a' + 2d)$$

The cost for removing B is $O(\ell + b + 2(b + \log\log(N + |E|)))$.

We show $\ell + b + 2(b + \log\log(N + |E|)) \leq 2(b + \log\log(N + |E|))(a' - 2\ell - 2d)$

This holds if $4(b + \log\log(N + |E|)) \leq 2(b + \log\log(N + |E|))(a' - 2\ell - 2d)$

if $2 \leq a' - 2\ell - 2d$

if $2 \leq a' - 2a - 2d$ (since $\ell < a$)

if $a' \geq 2a + 2d + 2$ (requirement **(4)**)

case 3: B has correct size i.e. $a \leq \ell \leq b$

The contribution of block B to $bal(D'_{i-1})$ is $2(b + \log\log(N + |E|))(b - b' + 1)$ if $B \in S^+$ and $2(b + \log\log(N + |E|))(a' - a + 1)$ if $B \in S^-$. Thus we have

$$\begin{aligned} bal(D'_{i-1}) - bal(D'_i) &\geq 2(b + \log\log(N + |E|)) \min(b - b' + 1, a' - a + 1) \\ &\geq 2(b + \log\log(N + |E|)) \end{aligned}$$

But the cost for removing B is $\ell \leq b \leq O(b + \log\log(N + |E|))$. ∎

Lemma 7. $bal(D_{-1}) = O(|E|)$

Proof:
$bal(D_{-1}) \leq b \sum_{v \in V} |A(v)| = O(|E|)$ by lemma 1 of section 2. ∎

We can now easily complete the proof of the theorem.We have

$$\text{total cost} = O\left(\sum_{i=0}^{m-1} (b + \log\log(N_i + |E|)) + \sum_{i=-1}^{m-1} cost(D_{2i+1} \rightarrow D_{2i+2})\right)$$

It remains to show that the second term does not dominate the first term.

We have by lemma 6

$$\sum_{i=-1}^{m-1} (cost(D_{2i+1} \rightarrow D_{2i+2})) \leq \sum_{i=-1}^{m-1} (bal(D_{2i+1}) - bal(D_{2i+2}))$$

$$\leq bal(D_{-1}) - bal(D_0) + \sum_{i=0}^{m-1} (bal(D_{2i}) + 2d(b + \log\log(N_i + |E|)) - bal(D_{2i+2}))$$

(by lemma 5)

$$= bal(D_{-1}) - bal(D_{2m}) + \sum_{i=0}^{m-1} 2d(b + \log\log(N_i + |E|))$$

$$= O\left(|E| + \sum_{i=0}^{m-1} (b + \log\log(N_i + |E|))\right) \qquad \text{(by lemma 7)}$$

This completes the proof of theorem 2 and thereby the analysis of the dynamic behavior of fractional cascading. ∎

5. Maintaining Dynamic Partitions of Linear Lists

In this section we present a data structure that supports the operations FIND, UNION, SPLIT, ADD and ERASE as defined in section 2. Let B be a sequence of items, some of which may be marked.

FIND(x) computes the nearest marked item right of x in the sequence

SPLIT(x) marks item x

UNION(x) unmarks the marked item x

ADD(x, y) inserts x immediately before y into the sequence

ERASE(x) removes x from the sequence

The data structure is derived from the "$\log\log N$"-priority queue of P.v.Emde Boas [EKZ77]. The major difference is that the tree structure which is implicite in his data structure, is made explicite by pointers. This allows us to make the data structure dynamic and also simplifies the data structure slightly. We first give a formal definition of this structure and then describe in detail the algorithms for building it up and for FIND, UNION, SPLIT, ADD and ERASE. Finally we give an analysis for the cost of all five operations.

Definition 2. A **stratified tree** S for a sequence B of n items consists of:

1. A **representative** r with components mark, min, max
   The values of these fields are defined below by definition 3.

and if $n > 4$:

2. A **copy** B' of B which is divided into segments $B'_1, B'_2, \ldots, B'_\ell$ of size between $1/2\sqrt{n}$ and $2\sqrt{n}$ each, $1/2\sqrt{n} \leq \ell \leq 2\sqrt{n}$.
3. Stratified trees $S'_1, S'_2, \ldots, S'_\ell$ for sequences $B'_1, B'_2, \ldots, B'_\ell$
4. A stratified tree S_r for $R = \{$ representative of $B'_i \mid 1 \leq i \leq \ell\}$ ∎

Remarks:

1. Structures $S'_1, S'_2, \ldots, S'_\ell, S_r$ are called **substructures** of S and S **superstructure** of $S'_1, S'_2, \ldots, S'_\ell, S_r$.

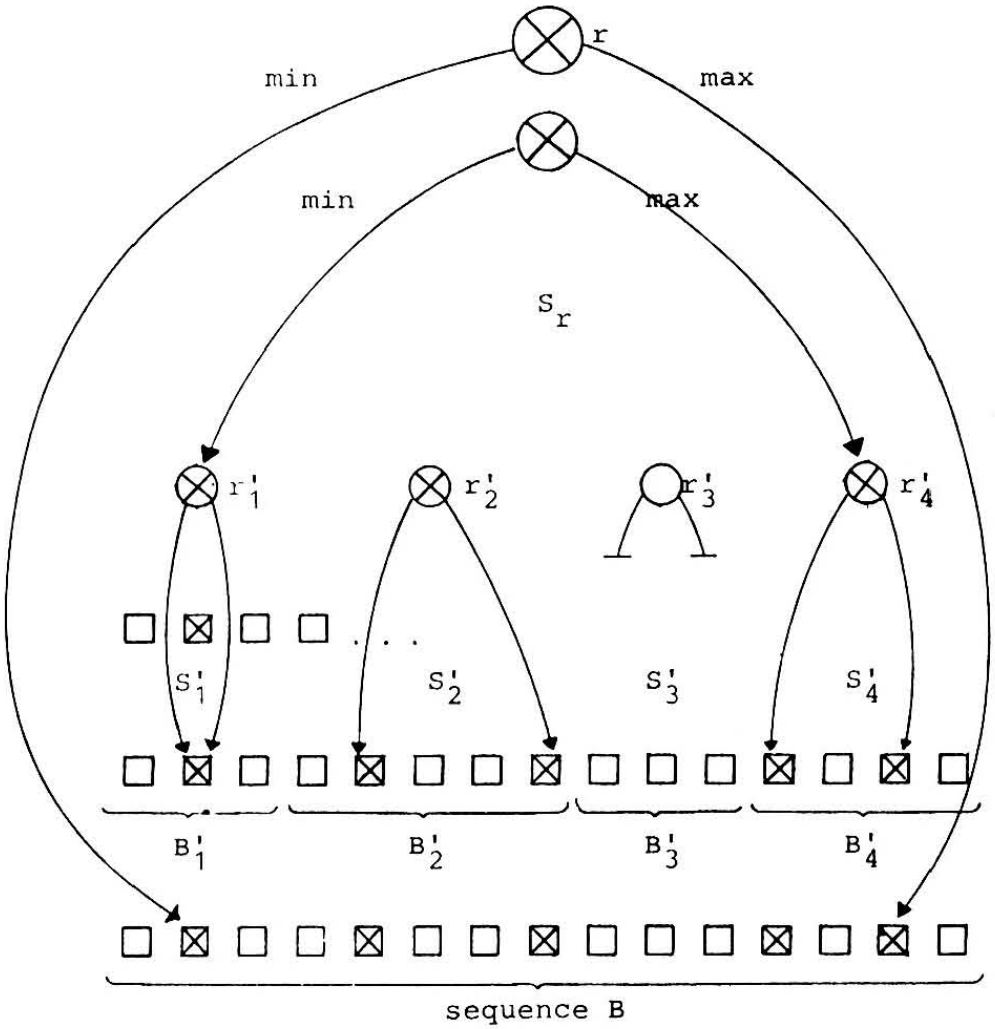

Figure 3

2. Note that the structure of a stratified tree does not depend on the marks of the items of B. It is only the fields of the various representatives which depend on it.

3. In the application to fractional cascading we have a stratified tree for each augmented catalogue $A(v)$. The marked items are the elements of $C(v)$.

We realize stratified trees by pointer structures as follows (cf. figure 3) . Each item or representative is realized by a record of type **node** consisting of 10 fields:

mark : boolean
pred : pointer to node
succ : pointer to node

low : pointer to node
high : pointer to node
rep : pointer to node
list : pointer to node
size : integer
min : pointer to node
max : pointer to node

The **pred** and **succ** fields organize the sequence B as a doubly linked linear list. The fields **low** and **high** are pointers connecting nodes with their copies in substructures and vice versa. **rep** links each node with its representative. The fields **list**, **size**, **min** and **max** are defined only for representatives: **list** is a pointer to the first element of the doubly linked list for B and **size** contains the length of the list. It remains to define the fields **mark**, **min** and **max**.

Definition 3. Let S be a stratified tree for sequence B with representative r

1. r.mark is true iff the sequence B contains at least one marked element. Also r.min and r.max point to the minimal and maximal element of B respectively. If there is no marked item in B then r.min and r.max have value nil.

2. If B contains at most one marked item then all substructures of S are "trivial", i.e. all nodes in these substructures are unmarked and all min and max pointers have value nil.

3. If B contains two or more marked items then the copies (in B') of the marked items of B are marked and the same rules are applied to all substructures S'_i, $1 \leq i \leq \ell$, containing at least one marked element. Note that this defines the marks of the representatives r'_i of S'_i, $1 \leq i \leq \ell$. Finally the same rules are applied to the substructure S_r. ∎

In the application to fractional cascading, the catalogue entries $x \in A(v)$ are extended by a pointer **low** in order to link the fractional cascading structure with the data structure defined above. Create for every $x \in A(v)$ a node k with k.mark = true iff $x \in C(v)$, k.list = k.min = k.max = nil, k.high = x, x.low = k. Use the pred and succ pointers to organize these nodes as a doubly linked linear list B. The following recursive procedure BUILDSTRUCTURE builds a stratified tree for B according to definition 2 and definition 3 (a "$\log\log n$"-priority queue as introduced in [EKZ77] for universe B and set $S = \{k \in B | k.\text{mark} = \text{true}\}$).

Algorithm 5:

```
1.  procedure BUILDSTRUCTURE (B: list of nodes; triv: boolean; var r: node);
(*  input: a doubly linked linear list B of items
    effect: builds a stratified tree for B and returns the representative r
            The stratified tree is trivial if the flag triv is true
    running time: O(|B| log log |B|)
*)
2.  n ← |B|;
3.  r ← new node;
4.  if triv
5.  then r.mark ← nil;
6.       r.max  ← nil;
7.       r.min  ← nil
8.  else r.mark ← ⋁_{k∈B} k.mark;
9.       r.min  ← min{k ∈ B|k.mark = true };
10.      r.max  ← max{k ∈ B|k.mark = true };
11.      s ← |{k ∈ B|k.mark = true }|;
12. fi;
13. r.high ← nil;
14. r.size ← n;
15. for all k ∈ B do k.rep ← r;
16.                  k.low ← nil;
17.              od;
18. if n > 4
19. then         B' ← ∅;
20.              for all k ∈ B
21.              do k' ← new node;
22.                 if triv
23.                 then k'.mark ← false
24.                 else k'.mark ← k.mark
25.                 fi;
26.                 k'.min ← nil;
27.                 k'.max ← nil;
28.                 k'.high ← k;
29.                 k.low  ← k';
30.                 B' ← B' ∪ {k'};
31.              od;
32.              let ℓ ∈ [1/2√n, 2√n]
33.              divide B' into blocks B'_1, B'_2, ..., B'_ℓ of size between √n and 3/2√n
                 (* then 2/3√n ≤ ℓ ≤ √n *)
```

```
34.              R ← ∅;
35.              for i = 1 to ℓ
36.              do organize B'_i as doubly linked linear list;
37.                 BUILDSTRUCTURE(B'_i, r', triv or (s ≤ 1));
38.                 R ← R ∪ {r};
39.              od;
40.              BUILDSTRUCTURE(R, x, triv or (s ≤ 1));
41.  fi;
```

Lemma 8. Let $n = |A(v)|$.

a) The data structure described above needs space $O(n \log\log n)$.

b) It can be build up in time $O(n \log\log n)$.

c) It supports the operations FIND, UNION and SPLIT in time $O(\log\log n)$.

Proof:
Parts a) and b) immediately follow from the observation that there are $O(\log\log n)$ levels of hierarchy. Below the algorithms for FIND, UNION and SPLIT are listed in detail. For their time bound see [EKZ77] or [M84a]. FIND, UNION and SPLIT are called pred, delete and insert respectively in the priority queue terminology of P.v.Emde Boas. ∎

For the formulation of the following algorithms we assume the existence of a boolean function **LEFTOF**(x, y) defined for two marked items x and y of sequence B. It returns true iff item x occurs before item y in B.

Remarks:

1. If we assume that in our application the proper catalogue entries $x \in C(v)$ are pairwise distinct we can realize the function LEFTOF by simply comparing x.key and y.key.

2. A general solution can be found in [T84]. There is shown that the following operations on a list of items

   insert(x, y) : insert x immediately before y into the list

   delete(x) : delete x from the list

   leftof(x, y) : return true iff x occurs before y in the list

can be realized in time $O(1)$ (amortized for insert and delete, worst case for leftof). Furthermore, the time required to build the date structure for a list of n items is $O(n)$ and therefore can be subsumed in the time required for BUILDSTRUCTURE.

Algorithm 6:

```
 1. function FIND (k: node) : node;
(* input: item k of sequence B
   output: closest marked successor of k in B or nil
   running time: O(1) if output is nil and O(log log |B|) otherwise
*)
 2. r ← k.rep;
 3. if r.size ≤ 4
 4. then find the closest marked successor by linear search
 5. else if (r.mark ≠ false) and LEFTOF(k,r.max)
 6.      then (* FIND has defined value *)
 7.           if r.max = r.min
 8.           then (* exactly one marked node *)
 9.                return(r.max)
10.           else (* two or more marked nodes *)
11.                x ← FIND(k.low);
12.                if x = nil
13.                then (* the call of FIND(k.low) took time O(1) *)
14.                     x ← FIND(k.low.rep);
15.                     return(x.min.high)
16.                else return(x.high);
17.                fi;
18.           fi;
19.      else (* FIND has value nil *)
20.           return(nil);
21.      fi;
22. fi;
```

A very important property of function FIND is that the recursion always stops if r.min = r.max for a visited substructure with representative r. Therefore the min and max pointers need not to be defined for the representatives of any substructures of S if S contains less than 2 marked nodes. Thus the correctness of FIND follows immediately from definition 3.

Algorithm 7:

```
1.  procedure SPLIT (k: node);
(*  precondition: k is an unmarked item of sequence B, S is a valid stratified tree for B
    postcondition: k.mark = true and S is still valid
    running time: O(1) if there is no marked item in B and O(log log |B|) otherwise
*)
2.  k.mark ← true
3.  r ← k.rep;
4.  if r.mark = false
5.  then (* structure was empty before *)
6.       r.mark ← true
7.       r.min  ← k;
8.       r.max  ← k;
9.  else if r.min = r.max
10.      then (* one marked item before the split *)
11.           x ← r.min;
12.           if LEFTOF(k, r.min)
13.           then r.min ← k
14.           else r.max ← k
15.           fi;
              (* now B contains 2 marked items: x and k
                 substructures must be updated according to definition 3 *)
16.           if r.size > 4
17.           then (* there are substructures *)
18.                SPLIT(x.low); (* takes time O(1) *)
19.                SPLIT(k.low); (* takes time O(1) *)
20.                if x.low.rep = k.low.rep
21.                then SPLIT(k.low)      (* a non-trivial call *)
22.                else SPLIT(k.low);     (* takes time O(1)  *)
23.                     SPLIT(k.low.rep) (* a non-trivial call *)
24.                fi;
25.           fi;
26.      else (* two or more marked items *)
27.           if LEFTOF(r.max,k) then r.max ← k fi;
28.           if LEFTOF(k, r.min) then r.min ← k fi;
29.           if k.low.rep.mark = false
30.           then SPLIT(k.low);     (* takes time O(1)  *)
31.                SPLIT(k.low.rep);(* a non-trivial call *)
32.           else SPLIT(k.low)      (* a non-trivial call *)
33.           fi;
34.      fi;
```

35. **fi**;

Here are some remarks about the following procedure UNION:

UNION($k, newmin, newmax$) sets the mark-flag of k and of its copies in substructures to false. If k is the only marked node UNION has to be called for its representative too. Otherwise (there is more than one marked node), if k is the minimal (maximal) marked node the min-pointer (max-pointer) must be changed such that it points to the closest marked successor (predecessor) of k. The new min and max pointers are returned in *newmin* and *newmax*.

Algorithm 8:

```
1.  procedure UNION (k: node; var newmin, newmax : node);
(*  precondition:  k is a marked item of sequence B, S is a valid stratified tree for B
    postcondition: k.mark = false, S is still valid and newmin (newmax) is the new
                   leftmost (rightmost) marked item in B
    running time:  O(1) if k is the only marked item in B and O(log log |B|) otherwise
*)
2.  k.mark ← false;
3.  r ← k.rep;
4.  if r.min = r.max
5.  then (* just one marked node k = r.min = r.max *)
6.       r.min ← nil;
7.       r.max ← nil;
8.       r.mark ← false
9.  else (* r.min ≠ r.max *)
10.      if r.size ≤ 4
11.      then do the obvious operations on linear list B
12.      else (* there are non-trivial substructures, we first recursively
13.              unmark k.low and compute the new min and max pointers *)
14.           UNION(k.low,x,y);
15.           if x = nil
16.           then (* k.low was the only marked node in the substructure
                      and hence the recursive call above took time O(1) *)
17.                UNION(k.low.rep, x1, y1);
18.                x ← x1.min;
19.                y ← y1.max;
20.           fi;
21.           if k = r.min then r.min ← x.high fi;
22.           if k = r.max then r.max ← y.high fi;
```

```
23.              if r.min = r.max
24.              then (* only one marked item now; trivialize the substructures *)
25.                   UNION(r.min.low);      (* needs time O(1) *)
26.                   UNION(r.min.low.rep); (* needs time O(1) *)
27.              fi;
28.         fi;
29.   fi;
30.   newmin ← r.min;
31.   newmax ← r.max;
```

Running time $O(\log\log|B|)$ is obvious from the comments in the program.

Operations ADD and ERASE modify the linear list B by adding or erasing an item. We perform ERASE(x) by deleting x and all its copies from the pointer structure. This may lead to a violation of the weight criterion of definition 1. We therefore store the outermost structure (in the hierarchy) which becomes too light in a variable *outbal* and later we call a procedure REBAL which restores balance. Similiary when we perform ADD(x, y) we add x and its copies to the pointer structure by giving them the same representatives as the corresponding copies of y, store the outermost structure which becomes too heavy in *outbal* and then call REBAL to restore the balance. The details are as follows:

Algorithm 9:

```
1.  procedure ADD (x, y);
(*  precondition: y ∈ B and S is a valid stratified tree for B
    effect: adds the unmarked item x immediately before item y to B and stores the
            representative of the first encountered substructure which is too heavy in
            variable outbal and the size of its superstructure in w
    running time: O(log log |B|)
*)
2.  outbal ← nil;
3.  r ← y.rep;
4.  x.rep ← r;
5.  insert x before y into the doubly linked list r.list
6.  r.size ← r.size + 1;
7.  if x.high ≠ nil
8.  then (* superstructure exists *)
9.       r' ← x.high.rep;
10.      if (r.size > 2√(r'.size)) and (outbal = nil)
11.      then (* overflow *)
```

```
12.             outbal ← r;
13.             w ← r'.size;
14.         fi;
15. fi;
16. if r.size > 4
17. then (* substructure *)
18.         z ← new node;
19.         z.mark ← false;
20.         z.high ← x;
21.         x.low ← z;
22.         ADD(z,y.low);
23. fi;
```

Algorithm 10:

```
1.  procedure ERASE (x);
(*  precondition: x is an unmarked item in B and S is a stratified tree for B
    effect: removes x from B and stores the representative of the first encountered
            substructure which is too light in the variable outbal and the size of its
            superstructure in w
    running time: O(log log |B|)
*)
2.  outbal ← nil;
3.  r ← y.rep;
4.  x.rep ← r;
5.  delete x from the doubly linked list r.list
6.  r.size ← r.size - 1;
7.  if x.high ≠ nil
8.  then (* superstructure exists *)
9.          r' ← x.high.rep;
10.         if (r.size < 1/2√(r'.size)) and (outbal = nil)
11.         then (* underflow *)
12.                 outbal ← r;
13.                 w ← r'.size;
14.         fi;
15. fi;
16. if r.size > 4
17. then (* substructure *)
18.         ERASE(x.low);
19. fi;
```

Both procedures obviously have running time $O(\log\log|B|)$ and perform the specified task. It remains to describe procedure REBAL which is called (if *outbal* $\neq$ nil) with parameters *outbal* and w after each call of ADD and ERASE.

Algorithm 11:

```
1.  procedure REBAL (r, w);
(*  precondition: r is the representative of a substructure S;
                  the superstructure of S has size w
    effect: combines S with a brother structure and restores
            the weight criterion of definition 2
    running time: O(|C| log log |C|) where C is the linear list represented by S
*)
2.  if r.succ = nil
3.  then   r' ← r.pred
4.  else   r' ← r.succ
5.  fi;
6.  C ← r.list ∪ r'.list;
7.  R ← r.rep.list;
8.  divide C into sequences B_1, B_2, ..., B_c of length between √w and (3/2)√w;
9.  for i = 1 to c
10. do BUILDSTRUCTURE(B_i, x);
11.    insert x before r into R;
12. od;
13. remove r and r' from R;
14. BUILDSTRUCTURE(R,r.rep);
```

What remains to do is estimating the amortized rebalancing cost.

Lemma 9. Let B be a sequence of n items. A call BUILDSTRUCTURE(B, r) followed by n ADD/ERASE operations has total cost $O(n(\log\log n)^2)$.

Proof:
Let S_0 be the stratified tree generated by BUILDSTRUCTURE(B, r) and let

$$S_0 — op_1 \longrightarrow S_0' — REBAL \longrightarrow S_1 — op_2 \longrightarrow S_1' — REBAL \longrightarrow S_2 — op_3 \cdots \longrightarrow S_{n-1}$$

be a sequence of n update operations each followed by a call of REBAL$(outbal, w)$ where

$op_i \in \{$ ADD, ERASE$\}$ for $1 \le i \le n-1$. Then we have

$$\text{total cost} = \sum_{i=1}^{n} \text{cost of } op_i \quad + \quad \text{total rebalancing cost}$$

$$= \sum_{i=0}^{n-1} cost(S_i \to S_i') + \sum_{j=0}^{n-1} cost(S_j' \to S_{j+1})$$

The first term is bounded by

$$O\left(\sum_{i=0}^{n-1} \log\log(n+i)\right) = O(n \log\log n)$$

which is the immediate coast for a sequence of n ADD or ERASE operations (cf. algorithms ADD and ERASE).

We use the bank account paradigm to estimate the second term. For a stratified tree S we define the following account based on Blum/Mehlhorn [BM80] and Luecker/Willard [WL85].

$$bal(S) := 2 \log\log n \cdot \sum \max(0, r.\text{size} - \frac{3}{2}\sqrt{w}, \sqrt{w} - r.\text{size})$$

The summation is over all representatives r and w denotes the size of the structure of r.

Then $bal(S_0) = 0$ since S_0 was constructed by a call of BUILDSTRUCTURE and hence $\sqrt{w} \le r.\text{size} \le 3/2\sqrt{w}$ for all representatives r in S_0. Next observe that a call of ADD or ERASE changes the size fields of $O(\log\log n)$ representatives and hence

$$bal(S_i') \le bal(S_i) + O((\log\log n)^2), \quad 0 \le i \le n-1$$

Let us finally consider the transition from S_i' to S_{i+1}. If $S_i' = S_{i+1}$, i.e. no call of REBAL was required, then $bal(S_i') = bal(S_{i+1})$ and the transition has cost zero. Assume next that the call REBAL$(outbal_i, w_i)$ was required. This call has cost $O(outbal_i.\text{size} \cdot \log\log n)$ and reduces the balance by at least $outbal_i.\text{size} \cdot \log\log n$, i.e.

$$bal(S_{i+1}) \le bal(S_i') - outbal_i.\text{size} \cdot \log\log n$$

We conclude that the cost of the transition is covered by the decrease in balance and lemma 5 is shown. ∎

Now it is easy to modify the data structure in such a way that all 5 operations FIND, UNION, SPLIT, ADD and ERASE are supported in time $O(\log\log n)$ (amortized for

ADD and ERASE, worst case for FIND, UNION and SPLIT) and space $O(n)$. We simply divide the sequence B into groups of size between $\log\log n$ and $4\log\log n$. Each group is realized as a doubly linked linear list. The $O(n/\log\log n)$ list headers are stored in a stratified tree as described above. This tree requires space $O(n)$. The cost of the operations FIND, UNION and SPLIT is only increased by an additive factor of $O(\log\log n)$ and hence is still $O(\log\log n)$. Operations ADD and ERASE are usually performed on the linear lists; only every $O(\log\log n)$-th operation requires a "real" update operation on the stratified tree (when a linear list grows too long, i.e. has length $4\log\log n+1$ then split it into two lists of length about $2\log\log n$ and if a linear list becomes too short, i.e. has length $\log\log n-1$ then either fuse it with a brother list of length at most $2\log\log n$ or take $(\log\log n)/2$ items away from a brother list if both brothers have length greater than $2\log\log n$). The amortized cost of an ADD or ERASE reduces to $O(\log\log n)$ since lemma 9 now can be applied to a sequence of $n/\log\log n$ items (list headers). We use the strategy described above for a sequence of n update operations and then rebuild the entire data structure. This has total cost $O(n\log\log n)$. We summarize the results of this section in the following theorem:

Theorem 3. FIND, UNION, SPLIT, ADD and ERASE can be supported in time $O(\log\log n)$ per operation and space $O(n)$. The time bound is worst case for FIND,UNION and SPLIT and amortized for ADD and ERASE.

Proof:
See lemma 8, lemma 9 and the discussion above. ∎

6. Applications

Dynamic fractional cascading is a very powerful strategy for improving the search and update time of data structures which consist of a basic frame graph (e.g. a binary tree) and where the data is stored as linear lists in the nodes of the graph. Such data structures are often used in computational geometry, e.g. segment trees, range trees, interval trees,.... In this section we demonstrate how the efficiency of segment and range trees [B77] [B79] [L78] [W78] [W85] can be considerably increased by dynamic fractional cascading. Note that static fractional cascading is a generalization of the techniques initially developed for segment and range trees (cf. [VW82], [EGS], [IA84], [L84], [CG85]). So we are only transfering our results about dynamic fractional cascading back to the origin of all of it. The reader can find the basic facts about segment and range trees in the text books [M84c] and [PS85].

6.1 The augmented segment tree

A segment tree is used to store a set S of horizontal line segments. A line segment is given by the x-coordinates of its endpoints and by their common y-coordinate. In this section we consider the case where the x-coordinates are restricted to a fixed finite universe U, $|U| = N$. This is frequently called the semi-dynamic case; the general case is treated in the next section. A segment tree consists of a search tree of depth $O(\log N)$ for the elements of U. In addition, there is a node list $NL(v) \subseteq S$ for each node of the tree. A line segment $L \in S$ is contained in the nodelist $NL(v)$ iff $range(v) \subseteq proj(L)$ and range(father(v)) $\not\subseteq proj(L)$. Here $proj(L)$ is the projection of L onto the x-axis and $range(v) = \{z \in U | \text{a search for } z \text{ in the underlying search tree goes through } v\}$. The line segments in a node list are ordered by their y-coordinates.

A segment tree can be naturally viewed as an instance of fractional cascading. The catalogue graph is the underlying search tree and the range $R(e)$ of each edge is taken to be the entire universe. Then the local degree is bounded by 3. The catalogue $C(v)$ of node v is the list (of the y-coordinates) of the line segments in $NL(v)$. According to the fractional cascading paradigm we will store in each node an augmented catalogue $A(v)$. We call the data structure obtained in this way an **augmented segment tree.**

Theorem 4. Let $U \subseteq \mathbb{R}, |U| = N$ and S a set of n horizontal segments $((x_1, y), (x_2, y))$ with $x_1, x_2 \in U$ and $y \in \mathbb{R}$, $|S| = n$.

a) An augmented segment tree for S needs space $O(n \log N)$.

b) An augmented segment tree for S can be constructed in time $O(n \log N)$.

c) Insertion or deletion of a segment needs time $O(\log N \log\log(n + N))$.

d) Semidynamic orthogonal segment intersection search: Let q be a vertical segment $((x_0, y_1), (x_0, y_2))$ and $L = \{p \in S | p \text{ intersects } q\}$. Then L can be computed in time $O(|L| + \log n + \log N \log\log(n+N))$.

e) (Imai/Asano [IA84]): All "$\log\log n$" - factors in a) to d) can be omitted if only insertions or only deletions are to be supported.

Proof:

a) The augmented segment tree needs space linear in the space requirement of the original segment tree (see lemma 1 in section 2 and theorem 3 in section 5).

b) By part a) the total length of all augmented catalogues is $O(n \log N)$. Also for an augmented catalogue $A(v)$ the data structure of section 5 requires space $O(|A(v)|)$ and can be constructed in that time.

c) To perform an update operation a segment must be inserted into (deleted from) $O(\log N)$ node lists (catalogues). Theorem 2 of section 4 and theorem 3 of section 5 show how to do that in amortized time $O(\log N \log\log(n+N))$.

d) Let $v_0, v_1, v_2, ..., v_\ell$ $(\ell \leq \log N)$ be the search path for x-coordinate x_0 of search segment q. Then the answer L is given by $\bigcup_{0 \leq i \leq \ell} \{s \in C(v_i) | y_1 \leq y_s \leq y_2\}$. In order to compute L we only have to locate in each catalogue $C(v_i)$, $i = 1, \ldots, \ell$, both y-coordinates y_1 and y_2 and to report all elements lying between them. The search for y_1 and y_2 takes time $O(\log n)$ at the root (binary search) and $O(\log\log(n+N))$ time for every other catalogue on the search path (Lemma 2 in section 2 and theorem 3 in section 5). Thus the total time needed for computing L is $O(|L| + \log n + \log N \log\log(n+N))$

e) See remark 2) at the end of section 2. ∎

6.2 The augmented dynamic segment tree

Now S is an arbitrary set of n horizontal segments i.e. both x-coordinates and y-coordinates of the segments in S are arbitrary real numbers. We first assume that S is **simple**, i.e. that the x-coordinates of all endpoints of segments in S are pairwise distinct. Lemma 15 at the end of this section will show how this restriction can be dropped.

Instead of a static binary tree we use now a balanced search tree as the underlying tree (catalogue graph), more precisely a BB[α]-tree (cf. [M84a]) for the x-coordinates of the line segments in S. BB[α]-trees are defined as follows. For a node v let $th(v)$ be the number of leaves in the subtree rooted at v. A tree is in class BB[α] for real parameter

α if for all nodes v in the tree $\alpha \leq th(v)/th(parent(v)) \leq 1-\alpha$. It is well known that for $1/4 \leq \alpha < 1 - \sqrt{2}/2$ BB[α] - trees can be rebalanced by rotations and double-rotations after insertion or deletion of a leaf. Since BB[α]-trees have logarithmic depth and can be built up in linear time it is clear that parts a), b) and c) of theorem 4 carry over with N replaced by n. For parts c) and e) we have to work harder.

Consider the insertion (deletion) of a segment, say (x_1, x_2, y). We first have to add two additional leaves (corresponding to x_1 and x_2) to the BB[α]-tree and then to restore the BB[α]-property. Once this is done we can actually insert the segment as described in the previous section. Similary if we delete a segment we first delete it from the node lists and then change the underlying BB[α]-tree. It is clear that the bounds of theorem 4, part c) and e) apply to the actual insertion and deletion of a segment. The goal of this section is to bound the cost of rebalancing the underlying BB[α]-tree.

A BB[α]-tree is rebalanced by rotations and double-rotations; the following fact (proved by [BM80], [L78] and [WL85] cf. also [M84a]) will be very important for us.

Fact: Let $1/4 \leq \alpha < 1 - \sqrt{2}/2$ and consider an arbitrary sequence of m insertions and deletions into an initially empty BB[α] - tree. Then the total number of rotations and double rotations about nodes of thickness in $[(\frac{1}{1-\alpha})^i, (\frac{1}{1-\alpha})^{i+1}]$ is $O(m/(1-\alpha)^i)$ for all $i \geq 0$.

Rotations and double rotations can be executed by inserting and deleting edges (modification of the underlying catalogue graph) and simple rearrangements of some node lists (see figure 4). For this reason we first prove a lemma that gives an upper bound on the cost of inserting edges into or deleting edges from catalogue graphs.

Lemma 10. Let $G = (V, E)$ be a catalogue graph.

a) Let $v, w \in V$ and $e = (v, w) \notin E$. Then the edge e can be inserted into the catalogue graph G in time $O((|A(v)| + |A(w)|) \log\log(|A(v)| + |A(w)|))$.

b) Let $v, w \in V$ and $e = (v, w) \in E$. Then the edge e can be deleted from G in time $O((|A(v)| + |A(w)|) \log\log(|A(v)| + |A(w)|))$.

c) $|A(v) \cap R(v, w)| = O(|A(w) \cap R(v, w)|)$ for all edges $e = (v, w) \in E$.

Proof:

a) To insert a new edge between nodes v and w we have to create bridges between $A(v)$ and $A(w)$ such that the resulting blocks B fulfill invariant 1 and make no

contribution to the bal-function defined in section 4, i.e. $a' \leq |B| \leq b'$. This requires at most $2/a'(|A(v)| + |A(w)|)$ insertions of non-proper elements into the catalogues. The positions of the new bridges can be easily determined by scanning $A(v)$ and $A(w)$ from left to right in time $O(|A(v)| + |A(w)|)$. Thus the total time for inserting e is $O((|A(v)| + |A(w)|) \log\log(|A(v)| + |A(w)|))$.

b) All bridges between $A(v)$ and $A(w)$ have to be deleted. This is done by deleting at most $2/a'(|A(v)| + |A(w)|)$ non-proper elements from $A(v)$ and $A(w)$.

c) There are at most $|A(w) \cap R(v,w)|$ bridges between $A(v)$ and $A(w)$. Between two adjacent bridges there are at most b elements in $A(v) \cap R(v,w)$ (invariant 1). Thus we have $|A(v) \cap R(v,w)| \leq (b+1)|A(w) \cap R(v,w)|$ ∎

We will now specialize the discussion to segment trees. Let S be a simple set of n horizontal line segments and consider a segment tree based on a BB[α]-tree for the x-coordinates of the endpoints.

Lemma 11. A rotation at node v costs time $O(|A(v)| \log\log n)$.

Proof:
A rotation at v can be executed by the following 3 steps (see fig. 4):
a) Delete the edges (u,v) and (w,y)

b) Insert the edges (u,w) and (v,y)

c) Rearrange the node lists:

$$C'(w) \leftarrow C(v)$$

$$C'(v) \leftarrow C(x) \cap C(y)$$

$$C'(x) \leftarrow C(x) - C'(v)$$

$$C'(y) \leftarrow (C(y) - C'(v)) \cup C(w)$$

$$C'(z) \leftarrow C(z) \cup C(w)$$

Inserting and deleting two edges by lemma 10 needs time

$$O(4(|A(u)| + |A(v)| + |A(w)| + |A(y)|) \log\log n) = O(|A(v)| \log\log n)$$

(note that $A(v) = O(n + |E|)$ (lemma 1) $= O(n)$ since $|E| \leq n$)

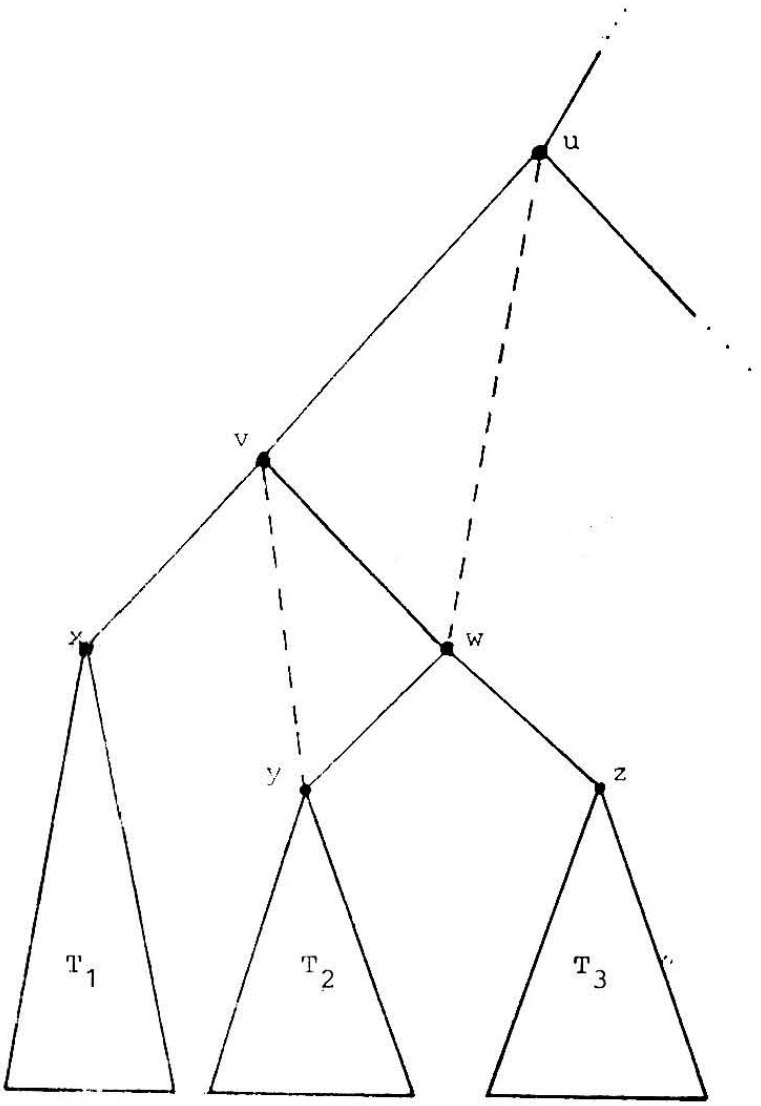

Figure 4

Rearranging the nodes lists requires $O(|C(v)|+|C(w)|+|C(x)|+|C(y)|+|C(z)|)$ insertions and deletions which can be performed in time

$$O((|C(v)| + |C(w)| + |C(x)| + |C(y)| + |C(z)|) \log\log n) = O(|A(v)| \log\log n)$$

Thus the total cost of a single rotation at node v is $O(|A(v)| \log\log n)$.

■

Lemma 12. Let $th(v)$ = number of leaves in the subtree rooted at v and $\frac{1}{4} < \alpha \leq 1-\frac{\sqrt{2}}{2}$.

a) $|C(v)| = O(th(v))$ for all $v \in V$

b) $th(v) \leq (\frac{1}{\alpha})^{dist(v,w)} th(w)$ for all $v, w \in V$
where $dist(v, w)$ is the distance between v and w in the $BB[\alpha]$ tree.

Proof:

a) Observe first that $C(v) = NL(v)$
Next note that if $s \in NL(v)$ then the search path for the x-coordinate of one of the endpoints goes through father(v). Thus $|NL(v)| \leq th(v)$.

b) For any $BB[\alpha]$-tree holds:

$$\alpha \leq \frac{th(v)}{th(w)} \leq 1 - \alpha \quad \text{if } v \text{ is the a son of } w \text{ and hence}$$

$$\frac{th(v)}{th(w)} \leq \max(\frac{1}{\alpha}, 1 - \alpha) = \frac{1}{\alpha} \quad \text{for every edge } (v, w) \in E$$

A simple induction completes the proof. ∎

Lemma 13. Let $G = (V, E)$ be an undirected binary tree (i.e. degree(v) ≤ 3 for all $v \in V$) and $v, w \in V$ with $dist(v, w) = i$. The number of all possible paths of length $j \geq i$ from v to w is at most $2^j 3^{j-i} \leq 6^j$.

Proof:
Let $v = v_0$ —— v_1 —— v_2 —— v_3 —— ... —— v_{i-1} —— $v_i = w$ be the shortest path from v to w. Since G is a tree any path P from v to w must use all nodes $v_0, v_1, ..., v_i$ and all edges $(v_k, v_{k+1}), 0 \leq k \leq i-1$. At each node v_k, $0 \leq k \leq i$, P can form a loop of length ℓ_k with

$$\sum_{k=0}^{i} \ell_k = j - i$$

Since the degree of every node is at most 3 there are at most 3^ℓ possible loops of length

ℓ. Thus we have

$|\{$ paths from v to w of length $j \mid j \geq i = dist(v,w)\}|$

$$\leq \sum_{\ell_0+\ell_1+\ell_2+\ldots+\ell_i=j-i} 3^{\ell_0}3^{\ell_1}3^{\ell_2}\ldots3^{\ell_i}$$

$$\leq \binom{j}{i} 3^{(\ell_0+\ell_1+\ell_2+\ldots+\ell_i)}$$

since there are $\binom{j-1}{i}$ possible ways to write $(j-i)$ as sum of $i+1$ terms

$$= \binom{j}{i} 3^{j-i}$$

$$\leq 2^j 3^{j-i}$$

∎

Lemma 14. $|A(v)| = O(th(v))$ for all nodes v.

Proof:

Let $Br(v,w)$ denote the number of bridges between $A(v)$ and $A(w)$

$$|A(v)| = |C(v)| + \sum_{(v,w)\in E} Br(v,w)$$

$$\leq |C(v)| + \sum_{(v,w)\in E} \frac{|A(v)| + |A(w)|}{a}$$

$$= |C(v)| + \frac{3}{a}|A(v)| + \frac{1}{a}\sum_{(v,w)\in E} |A(w)|$$

Thus $$|A(v)| \leq \frac{a}{a-3}(|C(v)| + \frac{1}{a}\sum_{(v,w)\in E} |A(w)|)$$

Estimating $|A(w)|$ in the same way yields

$$|A(v)| \leq \frac{a}{a-3}\left(|C(v)| + \frac{1}{a}\sum_{(v,w)\in E}\frac{a}{a-3}\left(|C(w)| + \frac{1}{a}\sum_{(w,u)\in E}|A(u)|\right)\right)$$

$$= \frac{a}{a-3}\left(|C(v)| + \frac{1}{a-3}\sum_{(v,w)\in E}|C(w)| + \frac{1}{a-3}\sum_{(v,w)\in E}\sum_{(w,u)\in E}|A(u)|\right)$$

Repeating this procedure k times results in

$$|A(v) \leq \frac{a}{a-3}\left(|C(v)| + \sum_{i=1}^{k}\sum_{w\in P^i(v)}\frac{|C(w)|}{(a-3)^i} + \sum_{u\in P^{k+1}(v)}\frac{|A(u))|}{(a-3)^{k+1}}\right)$$

where $P^i(v)$ is the set of nodes reachable from v by a path of length i. Next observe that (we use $\{\text{path } v \xrightarrow{i} w\}$ to denote the set of paths of length i from v to w)

$$|C(v)| + \sum_{i=1}^{k}\sum_{w\in P^i(v)}\frac{|C(w)|}{(a-3)^i}$$

$$\leq |C(v)| + \sum_{i=1}^{\infty}\sum_{w\in P^i(v)}\frac{|C(w)|}{(a-3)^i}$$

$$= |C(v)| + \sum_{i=1}^{\infty}\sum_{j=0}^{\infty}\sum_{\substack{w\in P^i(v)\\ dist(v,w)=j}}\frac{|C(w)|}{(a-3)^i}$$

$$= |C(v)| + \sum_{j=0}^{\infty}\sum_{\substack{w\in V\\ dist(v,w)=j}}\left(|C(w)|\sum_{i\geq j}\frac{|\{\text{path } v\xrightarrow{i} w\}|}{(a-3)^i}\right)$$

$$\leq th(v) + \sum_{j=0}^{\infty}\sum_{\substack{w\in V\\ dist(v,w)=j}} th(v)(\frac{1}{\alpha})^j\sum_{i\geq j}\left(\frac{6}{a-3}\right)^i$$

$$\leq th(v) + \sum_{j=0}^{\infty} 3\cdot 2^{j-1}th(v)\left(\frac{1}{\alpha}\right)^j\left(\frac{6}{a-3}\right)^j\frac{1}{1-\frac{6}{a-3}}$$

by lemmas 12 and 13 and the fact that most $3\cdot 2^{j-1}$ nodes w have distance j from v.

$$= O(th(v)) \quad \text{if } a > \tfrac{12}{\alpha} + 3 \text{ and } a > 9$$

and that

$$\sum_{u \in P^{k+1}(v)} \frac{|A(u))|}{(a-3)^{k+1}} \leq \sum_{j=0}^{\infty} \sum_{\substack{w \in V \\ dist(v,w)=j}} |A(w)| \frac{|\{\text{path } v \xrightarrow{k+1} w\}|}{(a-3)^{k+1}}$$

$$\leq \sum_{j \geq 0} \sum_{\substack{w \in V \\ dist(v,w)=j}} |A(w)| \left(\frac{6}{a-3}\right)^{k+1} \qquad \text{(lemma 13)}$$

$$= \left(\frac{6}{a-3}\right)^{k+1} \sum_{w \in V} |A(w)|$$

$$\leq \left(\frac{6}{a-3}\right)^{k+1} O(n) \qquad \text{(lemma 1 of section 2)}$$

$$= O(1) \qquad \text{for } k \text{ sufficient large}$$

Putting these estimates together we obtain

$$|A(v)| = O(th(v))$$

■

Now we can estimate the amortized cost for insertions and deletions:

Theorem 5. In the augmented dynamic segment tree a single rotation at any node v costs time $O(th(v) \log \log n)$.

Proof:
Lemma 11 and lemma 14 ■

Theorem 6. The rebalancing of the underlying $BB[\alpha]$-tree for a sequence of m insertions or deletions has cost $O(m \log n \log \log n)$.

Proof:
In a $BB[\alpha]$-tree the rebalancing cost is

$$O(m \sum_{i=0}^{c \log n} f((1-\alpha)^{-i})(1-\alpha)^i) \qquad \text{with } c = -\frac{1}{\log(1-\alpha)}$$

provided that a single rotation at node v needs time $O(f(th(v)))$ (for details see [M84a], [M84c] or [WL85]). In this application we have $f(th(v)) = th(v) \log\log n$ (theorem 5).

Thus an upper bound for the cost of m update operations is

$$O(m \sum_{i=0}^{c \log n} (1-\alpha)^{-i} \log\log n (1-\alpha)^i) = O(m \sum_{i=0}^{c \log n} \log\log n)$$

$$= O(m \log n \log\log n)$$

■

Theorem 7. Let S be a set of n horizontal segments with endpoints in $\mathbb{R} \times \mathbb{R}$

a) An augmented dynamic segment tree for S can be build in time $O(n \log n \log\log n)$.

b) It has space requirement $O(n \log n)$.

c) Insertion or deletion of a segment needs time $O(\log n \log\log n)$.

d) Let q be a vertical segment $((x_0, y_1), (x_0, y_2))$ and $L = \{((x,y),(x',y)) \in S | x \leq x_0 \leq x'$ and $y_1 \leq y \leq y_2\}$ Then L can be computed in time $O(|L| + \log n \log\log n)$ (dynamic orthogonal segment intersection search)

e) (Imai/Asano [IA84]): All "$\log\log n$" - factors in a) to d) can be omitted if only insertions or only deletions are to be supported.

Proof:
See proof of theorem 4 and the discussion above for simple sets S. The following lemma shows how to deal with arbitrary sets of line segments.

Lemma 15. The augmented dynamic orthogonal segment intersection search can be solved for general S in the same time bound as for simple S.

Proof:
We replace the x-coordinates of endpoints by triples and use the lexicographic ordering on the triples. Assume that each segment s has a unique number (name) $num(s)$. Then replace the x-coordinate x_1 of the left endpoint of s by the tripel $(x_1, 1, num(s))$ and the x-coordinate x_2 of the right endpoint by $(x_2, 0, num(s))$. Note that all first coordinates are distinct now. Furthermore left endpoints with the same x-coordinate as right endpoints received a lower first coordinate. A vertical query segment with coordinates (x, y_1, y_2) is simply transformed to $((x, \frac{1}{2}, \text{anything}), y_1, y_2)$. ■

This completes the proof of theorem 7. ∎

6.3 The augmented range tree

The techniques we used for dynamization of segment trees can also be applied to range trees (d-fold trees) [L78] [W78] [B79]. The correctness of the following two theorems is straightforward. They improve the best previous results for dynamic orthogonal range search of Willard [W85] who gives an algorithm with query and update time $O(\log^{3/2} n)$ ($O(\log^{d-1/2} n)$ in the d-dimensional case).

Theorem 8. Let S be a set of n points in the plane

a) An augmented dynamic range tree for S can be constructed in time $O(n \log n \log\log n)$.

b) It has space requirement $O(n \log n)$.

c) Insertion or deletion of a point needs time $O(\log n \log\log n)$.

d) Let $x_0, x_1, y_0, y_1 \in \mathbb{R}$ with $x_0 \leq x_1$ and $y_0 \leq y_1$ and $L = \{(x,y) \in S | x_0 \leq x \leq x_1$ and $y_0 \leq y \leq y_1\}$ Then L can be computed in time $O(|L| + \log n \log\log n)$.

e) (Imai/Asano [IA84]): All "$\log\log n$" - factors in a) to d) can be omitted if only insertions or only deletions are to be supported.

Theorem 8 can be generalized to d-dimensional space ($d > 2$) by well known standard techniques (e.g. [M84] Vol.3):

Theorem 9. Let S be a set of n points in the $\mathbb{R}^d, d > 2$

a) An augmented dynamic range tree for S can be build in time $O(n \log^{d-1} n \log\log n)$.

b) It has space requirement $O(n \log^{d-1} n)$.

c) Insertion or deletion of a point needs time $O(\log^{d-1} n \log\log n)$.

d) Let $x_0, x_1, y_0, y_1 \in \mathbb{R}$ with $x_0 \leq x_1$ and $y_0 \leq y_1$ and $L = \{(x,y) \in S | x_0 \leq x \leq x_1$ and $y_0 \leq y \leq y_1\}$ Then L can be computed in time $O(|L| + \log^{d-1} n \log\log n)$.

e) All "$\log\log n$" - factors in a) to d) can be omitted if only insertions or only deletions are to be supported.

7. Conclusions and Open Problems

In this paper we introduced dynamic fractional cascading and applied it to segment and range trees. An important subproblem which we had to treat is the problem of maintaining interval partitions of dynamic linear lists. The major open problem is whether the $\log \log N$ factor introduced by our solution to that problem is really necessary.

8. References

[B77] J.L. Bentley: "Solutions to Klee's Rectangle Problem", unpubl. manuscript, Carnegie - Mellon Univ., Dept. of Computer Science, 1977

[B79] J.L. Bentley: "Decomposable Searching Problems", Inform. Process. Lett. 8 , 1979, 244-251

[BM80] N. Blum, K. Mehlhorn: "On the Average Number of Rebalancing Operations in Weight-Balanced Trees", Theoretical Computer Science 11, 1980, 303-320

[CG85] B. Chazelle, L. Guibas: "Fractional Cascading: A Data Structuring Technique with Geometric Applications", 12-th ICALP, 1985, 90-100

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

[EKZ77] P. v. Emde Boas, R. Kaas, E. Zijlstra: "Design and Implementation of an Efficient Priority Queue", Math. Systems Theory 10, 1977, 99-127

[FMN85] O. Fries, K. Mehlhorn, St. Näher: "Dynamization of Geometric Data Structures", ACM Symposium on Computational Geometry, 1985, 168-176

[G85] R.H. Güting: "Fast Dynamic Intersection Searching in a Set of Isothetic Line Segments", Inform. Process. Lett. 21, 1985, 165-171

[GT83] H.N. Gabow, R.E. Tarjan: "A linear-time algorithm for a special case of disjoint set union", Proc. 15-th Ann. SIGACT Symp., 1983, 246-251

[HM82] S. Huddleston, K. Mehlhorn: "A new Representation for Linear Lists", Acta Informatica 17, 1982, 157-184

[IA84] T. Imai, T. Asano: "Dynamic Segment Intersection with Applications", 25th FOCS, 1984, 393-402

[L78] G.S. Luecker: "A Data Structure for Orthogonal Range Queries", 19th FOCS, 1978, 28-34

[L84] W. Lipski: "An $O(n \log n)$ Manhattan Path Algorithm", Inform. Proc. Lett. 19, 1984, 99-102

[M84] K. Mehlhorn: "Data Structures and Algorithms", Springer Publ. Comp., 1984
a) Vol. 1: Sorting and Searching
b) Vol. 2: Graph-Algorithms and NP-Completeness
c) Vol. 3: Multidimensional Searching and Computational Geometry

[PS85] F.P. Preparata, M.I. Shamos: "Computational Geometry, An Introduction" Springer Publ. Comp., 1985

[T85] R.E. Tarjan: "Amortized Computational Complexity", SIAM Journal Alg. Disc. Meth. 6, 1985, 306-318

[T84] A.K. Tsakalidis: "Maintaining Order in a Generalized Linked List", Acta Informatica 21, 1984, 101-112

[VW82] V.K. Vaishnavi, D. Wood: "Rectilinear Line Segment Intersection, Layered Segment Trees and Dynamization", Journal of Algorithms, Vol.3, 1982, 160-176

[W78] D.E. Willard: "New Data Structures for Orthogonal Range Queries", Technical Report, Harvard Univ., 1978

[W85] D.E. Willard: "New Data Structures for Orthogonal Queries", SIAM Journal of Computing, 1985, 232-253

[WL85] D.E. Willard, G.S. Luecker: "Adding Range Restriction Capability to Dynamic Data Structures", JACM, Vol.32, 1985, 597-617