Algorithmica (1990) 5: 215-241

Algorithmica

Dynamic Fractional Cascading[1]

Kurt Mehlhorn[2] and Stefan Näher[2]

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 this 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).

Key Words. Computational geometry, Linear lists, Dynamic data structures, Amortized complexity.

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 and Wood [VW82], Willard [W85], Edelsbrunner *et al.* [EGS86], Imai and Asano [IA87], and Lipski [L84] observed that the naive strategy of locating the key separately in each list by binary search is far from optimal and that more efficient techniques frequently exist. Chazelle and Guibas [CG86] 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* $R(e)=[l(e), r(e)]$, here $[l(e), r(e)]$ denotes the closed interval in U with endpoints $l(e)$ and $r(e)$. We use $N=\sum_{v \in V}|C(v)|$ to denote the total size of the catalogues. The goal is to 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 the "predecessor of x in $C(v)$" $< k \leq x$, i.e., the query locates k in each list $C(v)$, $v \in V'$.

[1] 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.
[2] Fachbereich 10, Universität des Saarlandes, D-6600 Saarbrücken, Federal Republic of Germany.

Received September 12, 1986; revised December 1, 1987. Communicated by D. T. Lee.

Chazelle and Guibas [CG86] have shown that fractional cascading supports queries in time $O(n+\log N)$ provided that G has a *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 assume throughout this paper that G has locally bounded degree. In this 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 the "predecessor of x in $C(v)$" $< k \leq x$, insert k into $C(v)$.

In Sections 2–4 we prove the 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 [CG86]. 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 (see [M84a] or [T85]) 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.

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 $O(\log^{3/2} n)$ in [W85]. The application to segment and range trees is described in Section 5. Finally Section 6 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 [CG86].

At each node v of the catalogue graph G we store a multiset $A(v) \supseteq C(v)$ 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 $A(v)-C(v)$ *nonproper*. 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, nonproper).

Note that we 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 nonproper 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 nonproper 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 nonproper element in $A(x$.target$)$, x.key $= y$.key and y.pointer $= x$. Thus x.pointer.pointer $= x$ for every nonproper element x.

Let $x \in A(v) - C(v)$ and $y = x$.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:

$x = y$.pointer,
$y = x$.pointer,
x.target $= w$,
y.target $= v$,
x.key $= y$.key,
x.kind $= y$.kind $=$ nonproper.

The field in_S is used to indicate that a bridge (x, y) belongs to a certain set S that is 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 $= y_l$.key $= l(e)$ and x_r.key $= y_r$.key $= r(e)$. We denote these two extreme bridges between $A(v)$ and $A(w)$ by $l(e)$-bridge and $r(e)$-bridge, respectively. The bridges connecting two augmented catalogues $A(v)$ and $A(w)$ divide the interval $[l(e), r(e)]$ into *blocks*. Each pair of neighboring bridges (x', y'), (x, y) defines such a block $B(x, y) \subseteq A(v) \cup A(w)$ (Figure 1). To identify a block we use the right bridge (x, y) of the pair. Note that the $l(e)$-bridge does not identify any block.

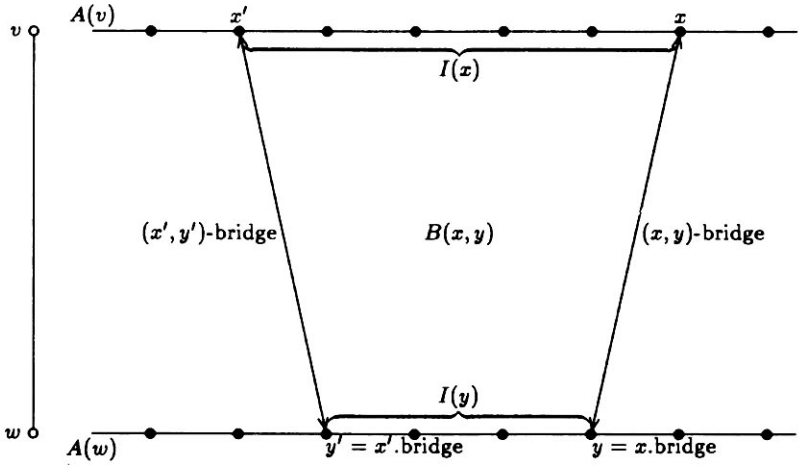

Fig. 1

More formally: let $v \in V$, $x \in A(v) - C(v)$, $x.\text{target} = w \in V$, 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 $A(w)$ (see 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 $l(e)$-bridge is used to identify the block

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

The counters $x.\text{count}$ and $y.\text{count}$ 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 (nonproper 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:

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 (see [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

$$a(Br(v, w)-2) \leq |A(v) \cap R(v, w)| + |A(w) \cap R(v, w)|.$$

That means

$$Br(v, w) \leq \frac{1}{a}(|A(v) \cap R(v, w)| + |A(w) \cap R(v, w)|) + 2.$$

Thus we have

$$\begin{aligned}
S &= N + 2 \sum_{(v,w) \in E} Br(v, w) \\
&\leq N + \frac{2}{a}\left(\sum_{(v,w) \in E} |A(v) \cap R(v, w)| + \sum_{(v,w) \in E} |A(w) \cap R(v, w)| \right) + 4|E| \\
&\leq N + \frac{2}{a} \sum_{u \in V} \sum_{x \in A(u)} |\{z \in V;\, x \in R(u, z)\}| + 4|E| \\
&\leq N + \frac{2d}{a} \sum_{u \in V} |A(u)| + 4|E| \qquad \text{since every } x \in A(u) \text{ is contained in } R(u, z) \text{ for at most } d \text{ vertices } z \\
&= N + \frac{2d}{a} S + 4|E|.
\end{aligned}$$

And finally

$$\begin{aligned}
S &\leq \frac{a}{a-2d}(N + 4|E|) \\
&\leq 3N + 12|E| \qquad \text{if} \quad a \geq 3d.
\end{aligned}$$

□

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 prove 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 $y.\text{pred}.\text{key} < k \leq y.\text{key}$
   running time: $O(1)$
   *)
2. $x' \leftarrow x$;
3. **while** $x'.\text{target} \neq w$ **do** $x' \leftarrow x'.\text{next}$ **od**;

4. $y \leftarrow x'$. pointer;
5. **while** y . pred . key $\geq k$ **do** $y \leftarrow y$. pred **od**;
6. **return**(y);

The validity of the search procedure follows from the observation that $(x', x'$. pointer) is the first bridge between v and w following k. 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.,

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

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 $y' = \min\{z \in C(w) \mid z \geq y\}$. 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 $x \in A(v)$

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

For the dynamic case (see the next section) we also have to provide operations for inserting or deleting proper and nonproper 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, nonproper}).

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 nonproper.
ADD	input: a pointer to some item x, effect: adds a nonproper item immediately before x to the list.
ERASE	input: a pointer to a nonproper item x, effect: deletes x from the list.
UNION	input: a pointer to a proper item x, effect: changes the mark of x to nonproper.
SPLIT	input: a pointer to a nonproper item x, effect: changes the mark of x to proper.

The names of the last two operations are suggested by the observation that the nonproper items partition the linear list into a family of intervals. Then operation UNION merges two adjacent intervals and SPLIT splits an interval.

LEMMA 3 [N87], [M86, Section III 8.2]. *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.

Lemma 3 generalizes a result of van Emde Boas [EKZ77] who has considered the operations FIND, UNION, and SPLIT. Alt *et al.* have shown in [MNA87] that the log log n bound is optimal for pointer machines.

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

2. In the semidynamic 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 and Tarjan [GT85] and Imai and Asano [IA87] 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 $x.\text{count}$ and $y.\text{count}$. 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 $x.\text{count}$ and $y.\text{count}$ 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 $|N(x, y)| \in [a..b]$. 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 = nonproper) 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).

LEMMA 4. *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

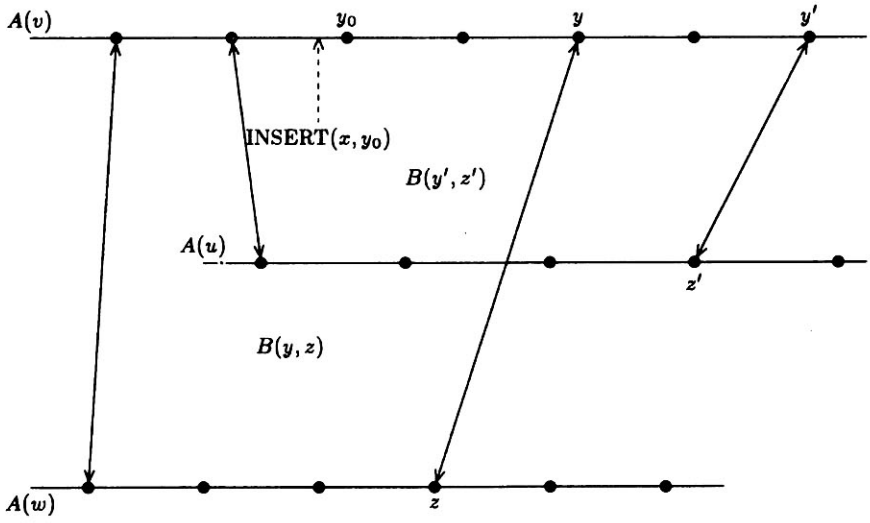

Fig. 2

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 = nonproper) 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 5. *Procedure DELETE preserves Invariant* 2.

PROOF. See proof of Lemma 4. □

In order to guarantee the time and space bounds of Lemmas 4 and 5, 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). 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 $O(l)$. Knowing the block size l there are three different cases:

Case 1: $l > b$, i.e., block $B(x, y)$ is too large. In this case the algorithm will divide $B(x, y)$ into $\lfloor 3l/b \rfloor + 1$ parts sized as equally as possible. This can be done by inserting $\lfloor 3l/b \rfloor$ bridges (i.e., inserting $2 \times \lfloor 3l/b \rfloor$ nonproper elements). Then the size of the resulting subblocks is between $b/4$ and $b/3$ and fulfills Invariant 1 if we choose $a \leq b/4$. The cost for this case is the cost for computing the block size ($O(l)$) plus the cost for inserting the new bridges ($6l/b$ calls of Insert).

Case 2: $l < a$, 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 nonproper 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 whether $B(x', y')$ 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 ($O(l)$), the cost for scanning the neighbor block ($O(b)$), and the cost for deleting two nonproper elements (two calls of DELETE).

Case 3: $a \leq l \leq b$, 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.

4. Analysis of the Dynamic Behavior. We 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 (see [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 nonnegative. As in the amortized analysis of balanced trees (see [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 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 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 were deposited in the account. Hence the value of the tokens must change over time. We 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. $A(v)=\bigcup_{(v,w)\in E}\{l(v, w), r(v, w)\}$.
3. $S=$ set of all blocks that violate Invariant 1.

First D_{-1} is transformed by procedure REBAL to a balanced data structure D_0. We next execute a sequence of m insert/delete operations starting with D_0. The ith 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}.

$$D_{-1}\to rebal_0\to D_0\to op_1\to D_1\to rebal_1\to D_2\to op2\to D_3-\cdots\to D_{2m},$$

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

Let $cost(D_j\to D_{j+1})$ 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

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

We show

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

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

where N_i is the number of proper elements before the ith 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

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

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

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

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

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

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

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

is bounded by

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

We can now invoke Lemma 3 from Section 2. Let k be the total number of list operations that are executed when going from D_{-1} to D_{2m}. 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 jth list operation. If we therefore replace *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. Out goal is therefore to bound

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

REMARK. Note that Lemma 3 gives us only a bound on the cost of a sequence of list operations, i.e., an amortized bound. But that is all we need for the current proof.

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

$$bal\text{: ``state of the data structure''} \to \mathbb{N}_0$$

that will indicate to what extent 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 later):

(1) $$b/4 \ge a',$$

(2) $$b/3 \le b',$$

(3) $$2b' \le b - 12d - 6,$$

(4) $$a' \ge 2a + 2d + 2.$$

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

$$a = 3d,$$
$$a' = 9d + 4,$$
$$b' = 12d + 6,$$
$$b = 36d + 18.$$

Then we define for every block B

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

and

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

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 lemma show three very important properties of function *bal*.

LEMMA 6. *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

$$\begin{aligned} bal(op(D)) - bal(D) &\leq (f(D) + 2d)(b + \log\log(N' + 1)) \\ &\quad - f(D)(b + \log\log N') \\ &= 2d(b + \log\log N') \\ &\quad + (f(D) + 2d)(\log\log(N' + 1) - \log\log N') \\ &= O(1) \end{aligned}$$

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

$$\begin{aligned} \log\log(N' + 1) - \log\log N' &\leq \frac{d}{dx} \log\log x|_{x = N'} \\ &= \frac{1}{N' \log N' (\ln 2)^2}. \end{aligned}$$

Case 2: $op = \text{DELETE}$. For the same reason as above the size of at most d blocks can be decreased by 1, i.e.,

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

Thus

$$\begin{aligned} bal(op(D)) &= (f(D)+2d)(b+\log\log(N+|E|-1)) \\ &\leq bal(D)+2d(b+\log\log(N+|E|)). \end{aligned}$$ □

LEMMA 7. $cost(D_{2i+1} \to D_{2i+2}) \leq O(bal(D_{2i+1}) - bal(D_{2i+2}))$ *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}:

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

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

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

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

Case 1: B is bigger than b, i.e., $l > b$. 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 $2(l-b') \times (b+\log\log(N+|E|))$ because block B is destroyed and increased by $2d(6l/b)$ $(b+\log\log(N+|E|))$ by the insertion of the new bridges. Thus

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

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

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

or

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

or

$$\left(1 - \frac{12d+6}{b}\right) l \geq 2b'$$

or

$$\left(1-\frac{12d+6}{b}\right)b \geq 2b' \qquad \text{(since } l > b\text{)}$$

or

$$b-12d-6 \geq 2b'. \qquad \text{This holds by requirement (3)!}$$

Case 2: B is smaller than a, i.e., $l < a$. In this case the value of the *bal*-function is decreased by $2(b+\log\log(N+|E|))(a'-l)$ (removal of B) and increased by $2(b+\log\log(N+|E|))(l+2d)$ (enlargement of the neighbor block and deletion of one bridge). Thus

$$bal(D_i') = bal(D_{i-1}') + 2(b+\log\log(N+|E|))(2l-a'+2d).$$

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

$$l+b+2(b+\log\log(N+|E|)) \leq 2(b+\log\log(N+|E|))(a'-2l-2d).$$

This holds if

$$4(b+\log\log(N+|E|)) \leq 2(b+\log\log(N+|E|))(a'-2l-2d)$$

if

$$2 \leq a'-2l-2d$$

if

$$2 \leq a'-2a-2d \qquad \text{(since } l < a\text{)}$$

if

$$a' \geq 2a+2d+2 \qquad \text{(requirement (4))}.$$

Case 3: B has correct size, i.e., $a \leq l \leq b$. 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 $l \leq b \leq O(b+\log\log(N+|E|))$. □

LEMMA 8. $bal(D_{-1}) = O(|E|)$.

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

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

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

We have by Lemma 7

$$\begin{aligned}
&\sum_{i=-1}^{m-1} (cost(D_{2i+1} \to D_{2i+2})) \\
&\quad \leq \sum_{i=1}^{m-1} (bal(D_{2i+1}) - bal(D_{2i+2})) \\
&\quad \leq bal(D_{-1}) - bal(D_0) \\
&\qquad + \sum_{i=0}^{m-1} (bal(D_{2i}) + 2d(b + \log\log(N_i + |E|)) - bal(D_{2i+2})) \qquad \text{(by Lemma 6)} \\
&\quad = bal(D_{-1}) - bal(D_{2m}) + \sum_{i=0}^{m-1} 2d(b + \log\log(N_i + |E|)) \\
&\quad = O\left(|E| + \sum_{i=0}^{m-1} (b + \log\log(N_i + |E|))\right) \qquad \text{(by Lemma 8).}
\end{aligned}$$

This completes the proof of Theorem 2 and thereby the analysis of the dynamic behavior of fractional cascading. □

5. 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], [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 (see [VW82], [EGS86], [IA87], [L84], and [CG86]). 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 [M84c] and [PS85].

5.1. 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 semidynamic 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 $range(v) = \{z \in U \mid$ a search for z in the underlying search trees 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 store in each node an augmented catalogue $A(v)$. We call the data structure obtained in this way an *augmented segment tree.*

THEOREM 3. *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 with endpoints (x_0, y_1) and (x_0, y_2) and let $L = \{p \in S \mid p$ intersects $q\}$. Then L can be computed in time $O(|L| + \log n + \log N \log\log(n+N))$.*
(e) [IA87] *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 Lemmas 1 and 3).

(b) By part (a) the total length of all augmented catalogues is $O(n \log N)$. The data structure for an augmented catalogue $A(v)$ 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 Lemma 3 of Section 2 show how to do that in amortized time $O(\log N \log\log(n+N))$.

(d) Let $v_0, v_1, v_2, \ldots, v_l$ ($l \leq \log N$) be the search path for x-coordinate x_0 of search segment q. Then the answer L is given by $\bigcup_{0 \leq i \leq l} \{s \in C(v_i) \mid y_1 \leq y_s \leq y_2\}$. In order to compute L we only have to locate in each catalogue $C(v_i)$, $i = 1, \ldots, l$, 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 (Lemmas 2 and 3 in Section 2). 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. □

REMARK. A restricted form of Theorem 3 was shown by Lipski [L83]. He made the additional assumption that the y-coordinates are also drawn from a fixed universe and that all segments, even the query segments, are known prior to the

construction of the segment tree. Furthermore, N denotes the number of horizontal segments plus the number of vertical query segments.

5.2. 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 14 at the end of this section shows how this restriction can be dropped.

Instead of a static binary tree we now use a balanced search tree as the underlying tree (catalogue graph), more precisely a $BB[\alpha]$-tree (see [M84a]) for the x-coordinates of the line segments in S. $BB[\alpha]$-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 $BB[\alpha]$ 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 $\frac{1}{4} \leq \alpha < 1-\sqrt{2}/2$ $BB[\alpha]$-trees can be rebalanced by rotations and double-rotations after insertion or deletion of a leaf. Since $BB[\alpha]$-trees have logarithmic depth and can be built up in linear time it is clear that parts (a)-(c) of Theorem 3 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 (corrresponding to x_1 and x_2) to the $BB[\alpha]$-tree and then to restore the $BB[\alpha]$-property. Once this is done we can actually insert the segment as described in the previous section. Similarly if we delete a segment we first delete it from the node lists and then change the underlying $BB[\alpha]$-tree. It is clear that the bounds of Theorem 3, parts (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 $BB[\alpha]$-tree.

A $BB[\alpha]$-tree is rebalanced by rotations and double-rotations; the following fact (proved in [BM80], [L78], and [WL85], see also, [M84a]) is very important for us.

FACT. Let $\frac{1}{4} \leq \alpha < 1-\sqrt{2}/2$ and consider an arbitrary sequence of m insertions and deletions into an initially empty $BB[\alpha]$-tree. Then the total number of rotations and double rotations about nodes of thickness in

$$[(1/(1-\alpha))^i, (1/(1-\alpha))^{i+1}]$$

is $O(m/(1-\alpha)^i)$ for all $i \geq 0$.

Rotations and double rotation can be executed by inserting and deleting edges (modification of the underlying catalogue graph) and simple rearrangements of some node lists (see Figure 3). 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 9. *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 $O((|A(v)|+|A(w)|) \log\log(|A(v)|+|A(w)|))$.*

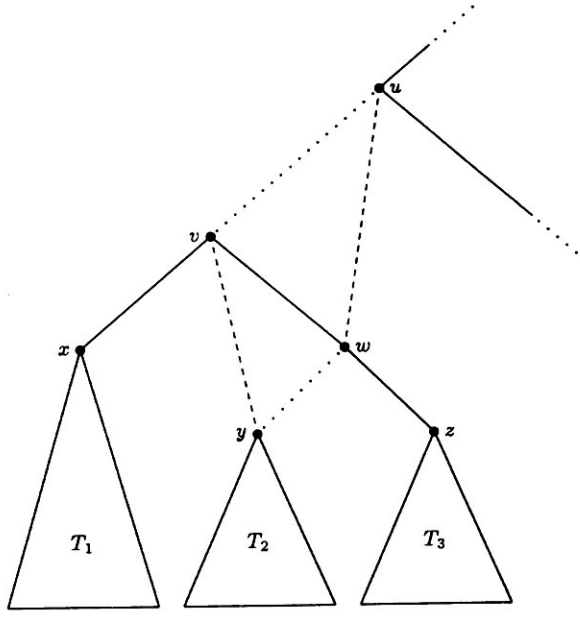

Fig. 3

(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 nonproper 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 $2/a'(|A(v)|+|A(w)|)$ nonproper 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 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 $BB[\alpha]$-tree for the x-coordinates of the endpoints.

LEMMA 10. *A rotation at node v costs time* $O(|A(v)|\log\log n)$.

PROOF. A rotation at v can be executed by the following three steps (see Figure 3):

(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 9 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 $|E| \leq n$). Rearranging the node 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 11. *Let $th(v)$ = number of leaves in the subtree rooted at v and $\frac{1}{4} < \alpha \leq 1 - \sqrt{2/2}$.*

(a) $|C(v)| = O(th(v))$ *for all* $v \in V$.
(b) $th(v) \leq (1/\alpha)^{dist(v,w)} th(w)$ *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

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

and hence

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

A simple induction completes the proof. □

LEMMA 12. *Let $G=(V, E)$ be an undirected binary tree (i.e., degree$(v)\leq 3$ 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 $v=v_0\to v_1\to v_2\to v_3\to\cdots\to v_{i-1}\to v_i=w$ be the shortest path from v to w. Since G is a tree any path P from v to w must use all nodes $v_0, v_1, \ldots, v_i$ 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

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

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

$$\begin{aligned}
&|\{\text{paths from } v \text{ to } w \text{ of length } j \mid j\geq i=dist(v, w)\}| \\
&\leq \sum_{l_0+l_1+l_2+\cdots+l_i=j-i} 3^{l_0}3^{l_1}3^{l_2}\cdots 3^{l_i} \\
&\leq \binom{j}{i} 3^{(l_0+l_1+l_2+\cdots+l_i)} \\
&\quad \text{since there are } \binom{j-1}{i} \text{ possible ways to write } (j-i) \text{ as sum of } i+1 \text{ terms} \\
&= \binom{j}{i} 3^{j-i} \\
&\leq 2^j 3^{j-i}.
\end{aligned}$$

□

LEMMA 13. $|A(v)|=O(th(v))$ *for all nodes v.*

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

$$\begin{aligned}
|A(v)| &= |C(v)| + \sum_{(v,w)\in E} Br(v, w) \\
&\leq |C(v)| + \sum_{(v,w)\in E} \frac{|A(v)|+|A(w)|}{a} \\
&= |C(v)| + \frac{3}{a}|A(v)| + \frac{1}{a}\sum_{(v,w)\in E} |A(w)|.
\end{aligned}$$

Thus

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

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

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

Repeating this procedure k times results in

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

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

$$|C(v)| + \sum_{i=1}^{k}\sum_{w\in P^i(v)}\frac{|C(w)|}{(a-3)^i}$$
$$\le |C(v)| + \sum_{i=1}^{\infty}\sum_{j=0}^{\infty}\sum_{\substack{w\in P^i(v)\\ dist(v,w)=j}}\frac{|C(w)|}{(a-3)^i}$$
$$= |C(v)| + \sum_{j=0}^{\infty}\sum_{\substack{w\in V\\ dist(v,w)=j}}\left(|C(w)|\sum_{i\ge j}\frac{|\{\text{path } v \to^i w\}|}{(a-3)^i}\right)$$
$$= O\left(th(v) + \sum_{j=0}^{\infty}\sum_{\substack{w\in V\\ dist(v,w)=j}}th(v)\left(\frac{1}{\alpha}\right)^j\sum_{i\ge j}\left(\frac{6}{a-3}\right)^i\right)$$
$$= O\left(th(v) + \sum_{j=0}^{\infty}3\times 2^{j-1}th(v)\left(\frac{1}{\alpha}\right)^j\left(\frac{6}{a-3}\right)^j\frac{1}{1-6/(a-3)}\right)$$

by Lemmas 11 and 12 and the fact that most $3\times 2^{j-1}$ nodes w have distance j from v

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

and that

$$\sum_{u\in P^{k+1}(v)}\frac{|A(u)|}{(a-3)^{k+1}} \le \sum_{j=0}^{\infty}\sum_{\substack{w\in V\\ dist(v,w)=j}}|A(w)|\frac{|\{\text{path } v \to^{k+1} w\}|}{(a-3)^{k+1}}$$
$$\le \sum_{j\ge 0}\sum_{\substack{w\in V\\ dist(v,w)=j}}|A(w)|\left(\frac{6}{a-3}\right)^{k+1} \qquad \text{(Lemma 12)}$$

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

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

$$= O(1) \quad \text{for } k \text{ sufficiently large.}$$

Putting these estimates together we obtain

$$|A(v)| = O(th(v)).$$ □

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

THEOREM 4. *In the augmented dynamic segment tree a single rotation at any node v costs time $O(th(v) \log \log n)$.*

PROOF. Lemmas 10 and 13. □

THEOREM 5. *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

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

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

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

$$= O(m \log n \log \log n).$$ □

THEOREM 6. *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 built 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 $L = \{((x, y), (x', y)) \in S \mid x \leq x_0 \leq x' \text{ and } y_1 \leq y \leq y_2\}$. Then L can be computed in time $O(|L| + \log n \log \log n)$ (dynamic orthogonal segment intersection search).*
(e) [IA87] *All "$\log \log n$"-factors in* (a)-(d) *can be omitted if only insertions or only deletions are to be supported.*

PROOF. See proof of Theorem 3 and the discussion above for simple sets S. The following lemma shows how to deal with arbitrary sets of line segments.

LEMMA 14. *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 [illegible] 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 triple $(x_1, 0, num(s))$ and the x-coordinate x_2 of the right endpoint by $(x_2, 1, num(s))$. 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 6. □

5.3. 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 7. *Let S be the 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* $x_0 \le x_1$ *and* $y_0 \le y_1$ *and* $L = \{(x, y) \in S \mid x_0 \le x \le x_1$ *and* $y_0 \le y \le y_1\}$. *Then L can be computed in time* $O(|L| + \log n \log\log n)$.
(e) [IA87] *All "*$\log\log n$*"-factors in* (a)-(d) *can be omitted if only insertions or only deletions are to be supported.*

Theorem 7 can be generalized to d-dimensional space ($d > 2$) by well-known standard techniques (e.g., [M84c]):

THEOREM 8. *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 built 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* $x_0 \le x_1$ *and* $y_0 \le y_1$ *and* $L = \{(x, y) \in S \mid x_0 \le x \le x_1$ *and* $y_0 \le y \le y_1\}$. *Then L can be computed in time* $O(|L| + \log^{d-1} n \log\log n)$.
(e) *All "*$\log\log n$*"-factors in* (a)-(d) *can be omitted if only insertions or only deletions are to be supported.*

6. Conclusions and Open Problems. In this paper we introduced dynamic fractional cascading and applied it to segment and range trees. Finally, we want to point out that there is a connection between the methods developed in this paper and the methods developed by Driscoll *et al.* in [DSST] for making data structures persistent. In fact, we can show that dynamic fractional cascading with insertions can only be used to make a data structure persistent. However, it is unclear at the moment whether this approach, in particular, our ability to support deletions efficiently, yields any additional insights into the persistency problem.

References

[B77] J. L. Bentley: Solutions to Klee's Rectangle Problem, unpublished manuscript, Department of Computer Science, Carnegie-Mellon University, 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, *Theoret. Comput. Sci.* **11**, 1980, 303-320.

[CG86] B. Chazelle, L. Guibas: Fractional Cascading: I, A Data Structuring Technique; II, Applications, *Algorithmica* **1**, 1986, 133-191.

[DSST] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan: Making Data Structures Persistent, *J. Comput. System Sci.*, to appear.

[EGS86] H. Edelsbrunner, L. Guibas, I. Stolfi: Optimal Point Location in a Monotone Subdivision, *SIAM J. Comput.* **15**, 1986, 317-340.

[EKZ77] P. van 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, *Proc. ACM Symposium on Computational Geometry*, 1985, 168-176.

[GT85] H. N. Gabow, R. E. Tarjan: A Linear-Time Algorithm for a Special Case of Disjoint Set Union, *J. Comput. System Sci.* **30**, 1985, 209-221.

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

[HM82] S. Huddleston, K. Mehlhorn: A New Representation for Linear Lists, *Acta Inform.* **17**, 1982, 157-184.

[IA87] T. Imai, T. Asano: Dynamic Orthogonal Segment Intersection Search, *J. Algorithms* **8**, 1987, 1-18.

[L83] W. Lipski: Finding a Manhattan Path and Related Problems, *Networks* **13**, 1983, 399-409.

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

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

[M84a] K. Mehlhorn: *Data Structures and Algorithms*, Vol. 1, Springer-Verlag, Berlin, 1984.

[M84b] Ibid., Vol. 2.

[M84c] Ibid., Vol. 3.

[M86] K. Mehlhorn: *Datenstrukturen und Algorithmen 1*, Teubner, 1986.

[MNA87] K. Mehlhorn, S. Näher, H. Alt: A Lower Bound on the Complexity of the Union-Split-Find Problem, *Proc. 13th ICALP*, 1987, 479-488.

[N87] S. Näher: Dynamic Fractional Cascading oder die Verwaltung vieler linearer Listen, Dissertation, University des Saarlandes, Saarbrücken, 1987.

[PS85] F. P. Preparata, M. I. Shamos: *Computational Geometry, An Introduction*, Springer-Verlag, Berlin, 1985.

[T85] R. E. Tarjan: Amortized Computational Complexity, *SIAM J. Algebraic Discrete Methods* **6**, 1985, 306-318.

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

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

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

[W85] D. E. Willard: New Data Structures for Orthogonal Queries, *SIAM J. Comput.*, 1985, 232-253.

[WL85] D. E. Willard, G. S. Luecker: Adding Range Restriction Capability to Dynamic Data Structures, *J. Assoc. Comput. Mach.* **32**, 1985, 597-617.