A DATA STRUCTURE WITH MOVABLE FINGERS AND DELETIONS

Dov Harel and George Lueker

University of California, Irvine
Irvine, CA 92717

Technical Report #145
December 1979

Keywords: Fingers, bounded balance trees, range queries, threads, iterated logarithms, approximate algorithms.

CR categories: 3.73, 3.74, 4.34, 5.25.

This research was supported by National Science Foundation grants MCS77-04410 and MCS79-04997.

A DATA STRUCTURE WITH MOVABLE FINGERS AND DELETIONS

Dov Harel and George Lueker

Abstract

A finger is a point in a file near which updates and searches can be conducted particularly efficiently. We present a data structure which supports insertions, deletions, searches, and approximate range queries. In each case both a finger f and a key k are specified as arguments to the appropriate algorithm. The total time bound for n updates (insertions or deletions) in an initially empty data structure is the sum over all updates of $O(\log d+\log^* n)$, where d is the linear distance (i.e., the distance in the linear list being represented) from f to k. By an approximate range query, we mean a query of the form "How many keys lie between the finger f and the key k?", where a predetermined percentage of error is allowed in the number returned. The time for each individual search or range query is $O(\log d)$.

1.0 Introduction

Recently data structures which allow fingers have attracted attention. A finger is a point in a file near which updates and searches can be performed especially efficiently. Two major papers have appeared recently on the subject of data structures which support fingers. The first [GMPR77] discussed a data structure which allowed insertions and deletions to be performed in $O(f+\log d)$ time, where d was the linear distance (i.e., the distance in the linear list being represented) from the finger to the key involved, and f was the number of fingers present; searches from a finger had a cost of $O(\log d)$. This structure had the disadvantage, however, that it did not allow the fingers to be moved very easily; moreover, if many fingers were present, the time complexity became excessive. A second paper [BT78] allowed any key to serve as a finger, once that key was located. The time for an insertion or search from a finger was $O(\log d)$. This new approach had two disadvantages:

a) the time bound was not guaranteed if deletions were present, and

b) the time bound was not valid for each individual operation, but rather only over a string of operations. The total cost of a string of insertions could be bounded by summing O(log d) over all of the insertions, even though some individual insertion might take much more than O(log d) time.

In section 2 we present a data structure which overcomes problem (a); it is possible to modify the algorithm to overcome problem (b) as well, although we do not present the details in this paper; they are presented in [Ha79]. The structure also has another interesting feature: define an approximate range query to be a question of the form "How many keys lie between the finger f and the key k?" where a predetermined percentage of error is allowed in the answer. Our structure will answer such queries in O(log d) time.

There is a slight cost for these improvements. Let n be the number of keys which are present, and d be the distance from the finger to the key. The time complexity of an update in our structure (when averaged over a string of operations) is $O(\log d + \log^* n)$, rather than O(log d). (Here $\log^* n$ denotes the iterated logarithm function.)

Note: quite recently Scott Huddleston [Hu79] has observed that if we only wish to overcome problem (a), a time bound of O(log d) is possible, through the use of 2-3-4 trees.

2.0 Threads and balance

2.1 Conventions

Throughout this paper, we will assume that all keys in the data file are distinct. We will also assume that we are using a slightly modified form of binary search tree in which all data is stored at the leaves, and each node has zero or two children. (This is done in order to simplify the algorithm; we suspect that the details could all be worked out for the more conventional arrangement in which all nodes contain data.) In addition, each node has a pointer to its parent. We do not store any field in internal nodes telling the value of some key; instead, we will use threads to guide searches in a manner to be mentioned below.

2.2 Bounded balance trees

Bounded balance trees [NR73] will be useful in our algorithm. Let T be a binary search tree. If x is a node in T, let the rank of x, written r(x), be one more than the number of descendants of x; the rank of an empty subtree is considered to be 1. (Also, we will always consider x to be a descendant of itself.) Define the balance of a node, written $\rho(x)$, to be the ratio of the rank of the left child of x to the rank of x. Say x is α-balanced if $\rho(x)$ is in $[\alpha, 1-\alpha]$. T is said to be a bounded balance tree with parameter α, or more briefly a BB(α) tree, if each node in T is α-balanced. Such trees are guaranteed to have O(log n) height, and have O(log n)-time update and search algorithms; these are based on the rotations shown in Figure 1 [NR73]. In [BM78] it is shown, in fact, that the average number of rebalancings per update can be made to be O(1).

Bounded balance trees have a very useful property: it is easy to show that the change in $\rho(x)$ due to a single update below x is, before rotations, inversely proportional to the rank of x. This fact, combined with the following lemma, enables us to show that nodes high in the tree are infrequently in need of rebalancing. Let T(x) denote the subtree rooted at x. Also, say that a node w actively participates in a rotation if either of its child pointers is changed.

Lemma 1. (This is a slight modification of a lemma used in [BM78, L79, W78].) There exist α, α', and e, with $\alpha' < \alpha$, and a slightly modified rotation algorithm, which make the following true.

a) To perform a rebalance at a node, we do not need to know the exact value of its rank or balance. Rather, the rank can have a relative error of e, and the balances may be calculated based on this slightly incorrect value.

b) Assume that every node in T(x) is α'-balanced and x is not α-balanced. Then after a rebalancing at node x, all of the nodes which actively participated in the rotation will be α-balanced.

Assume now that we have chosen a set of values for the parameters mentioned in Lemma 1. Nodes which are α'-balanced but not α-balanced will be of special interest. To discuss their balance more easily, we let $\beta(x)$ be

$(\alpha-\alpha')^{-1}$ times the distance on the real line from $\rho(x)$ to the interval $[\alpha,1-\alpha]$; it is then not hard to verify that x is α-balanced iff $\beta(x)=0$, and α'-balanced iff $\beta(x)\leq 1$. Further, one may show that the change in $\beta(x)$ due to a single insertion or deletion is inversely proportional to r(x).

2.3 Threads

In order to make searches efficient in balanced trees, we will introduce certain special fields which we will call threads; these are very similar to the threads used in [GMPR77], although our search procedure is considerably different, requiring no neighbor pointers. If a node x has a right child, define its right thread, written RTHREAD(x), to be its rightmost descendant, that is, the last node we visit if we repeatedly follow right links from x. If node x has no right child, define RTHREAD(x) to be its lowest right ancestor, that is, its successor in inorder; if x is the last node in inorder (i.e., x has the largest key in the tree), then RTHREAD(x) is the null pointer. The left thread of x is defined symmetrically. Note that with these threads we can easily compute a number of other quantities in O(1) time. For example, let RIGHTMOST(x) (resp. LEFTMOST(x)) be the rightmost (resp. leftmost) descendant of x. Then RIGHTMOST(x) could be calculated as

```
if RIGHT(x) = null then x else RTHREAD(x);
```

We will say that a node x subtends a key k (not necessarily in the tree) if

KEY(LEFTMOST(x)) $\leq$ k and KEY(RIGHTMOST(x)) $\geq$ k.

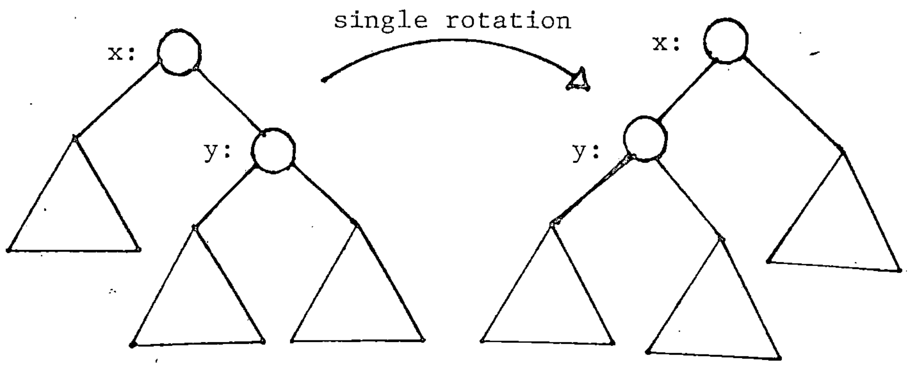
From this it is clear that the threads could be used to guide a search through the tree for a given key in a manner similar to the conventional binary tree search. Also, we could write a procedure to determine whether x was a descendant of y by simply returning

```
KEY(LEFTMOST(x)) ≥ KEY(LEFTMOST(y)) and
    KEY(RIGHTMOST(x)) ≤ KEY(RIGHTMOST(y)).
```

Finally, let LRA(x) (resp. LLA(x)) denote the lowest right (resp. left) ancestor of x. Note that this could be computed in O(1) time as

```
RTHREAD(RIGHTMOST(x))
```

The following two lemmas show that introduction of these threads will not make updates significantly more difficult.



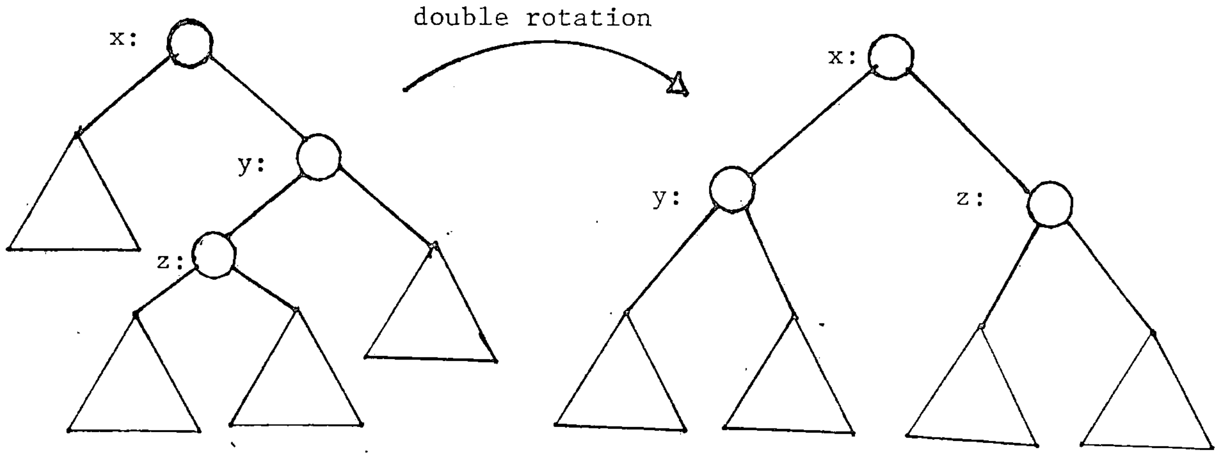


Figure 1. Rebalancing operations for trees of bounded balance [NR73].

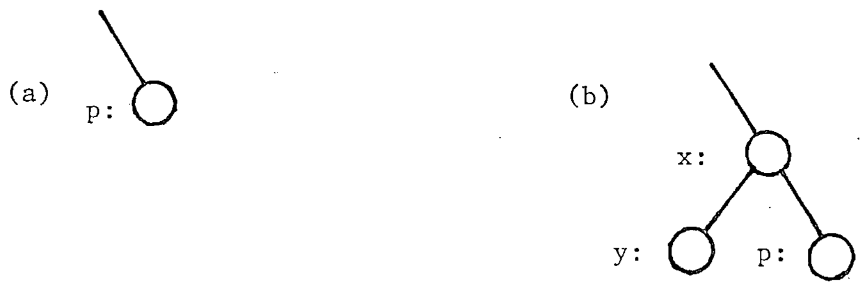


Figure 2. Insertion.

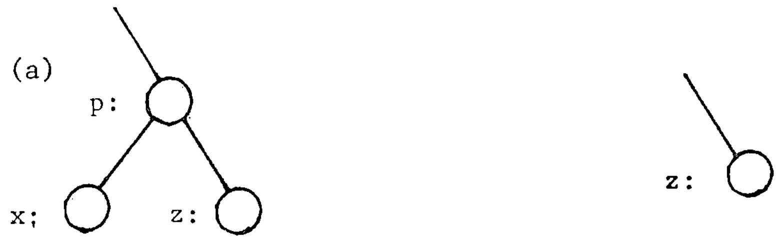


Figure 3. Deletion.

Lemma 2. If we do an insertion or deletion in a binary search tree, but no rotations, we may update the threads appropriately in O(1) time.

Proof. We present algorithms to perform the insertion or deletion. At first it may appear that many thread fields will need to be updated; for example, if the node p in Figure 2a has many ancestors along a right-leaning path, when p obtains a new right child we might expect to have to change many RTHREAD fields in these ancestors. A standard trick can be used to avoid this; we let move data in such a way as to make it possible for the node p to continue to be the rightmost descendant, even though its data might change. An algorithm is given below.

```
procedure INSERT(p,y);
begin comment insert y as a child of p. We assume that the node y has
    already been created, and that its KEY field has been set;
    wlog assume p is a right child;
    create a new node x and update the LEFT, RIGHT, and PARENT fields to
        replace the situation in Figure 2a by that in Figure 2b;
    comment note that no nodes above x have experienced a change in
        either their leftmost or rightmost descendant;
    LTHREAD(y) := PARENT(x);  RTHREAD(y) := x;
    LTHREAD(x) := y;  RTHREAD(x) := p;
    LTHREAD(p) := x;  comment RTHREAD(p) is unchanged;
    if KEY(y) < KEY(p)
        then KEY(y) :=: KEY(p); comment swap the keys;
end;
```

The deletion procedure is similar and is illustrated in Figure 3.

```
procedure DELETE(y);
begin comment delete the key in node y;
  p := the parent of y;
  wlog assume that p is a right child;
  k := the key in the sibling of y;
  x := LEFT(p);  z := RIGHT(p);
  rearrange the LEFT, RIGHT, and PARENT fields to replace Figure 3a by
    Figure 3b;
  comment again, notice that no node above z has experienced a change
    in either its leftmost or rightmost descendant;
  LTHREAD(z) := PARENT(z);  comment RTHREAD(z) is unchanged;
  KEY(z) := k;
end;                                                                   []
```

Lemma 3. For any rotation we do in a bounded balance tree, we may update the threads appropriately in O(1) time.

Proof sketch. Suppose we do a single or double rotation rooted at x. Let the situation before the rotation be as in Figure 4; in degenerate cases one may need to eliminate parts of the figure, or condense serveral items in the figure into a single node. Note that even in degenerate cases, the leftmost and rightmost node in the subtree rooted at x will not change during the rotation; thus we need not change any LTHREAD and RTHREAD fields of proper ancestors of x. In fact, it is not hard to see that the nodes represented by circles are the only nodes for which a change in LTHREAD or RTHREAD can take place. Moreover, one can determine that all of the new values to be placed in these fields are contained in the set of circles of the figure. Finally, using the thread fields we may locate all of the circles in the figure in O(1) time. From these observations it follows that all updating could be done in O(1) time. []

Henceforth we will not explicitly refer to the updating of these threads.

Lemma 4. In a bounded balance tree with threads, one may search for a key k, starting from a finger f, in time proportional to the log of the linear distance from f to k. Moreover, if each node in the tree contains a field telling the approximate number of descendants of that node with a maximum relative error of δ, then we may also return the number of keys between f and

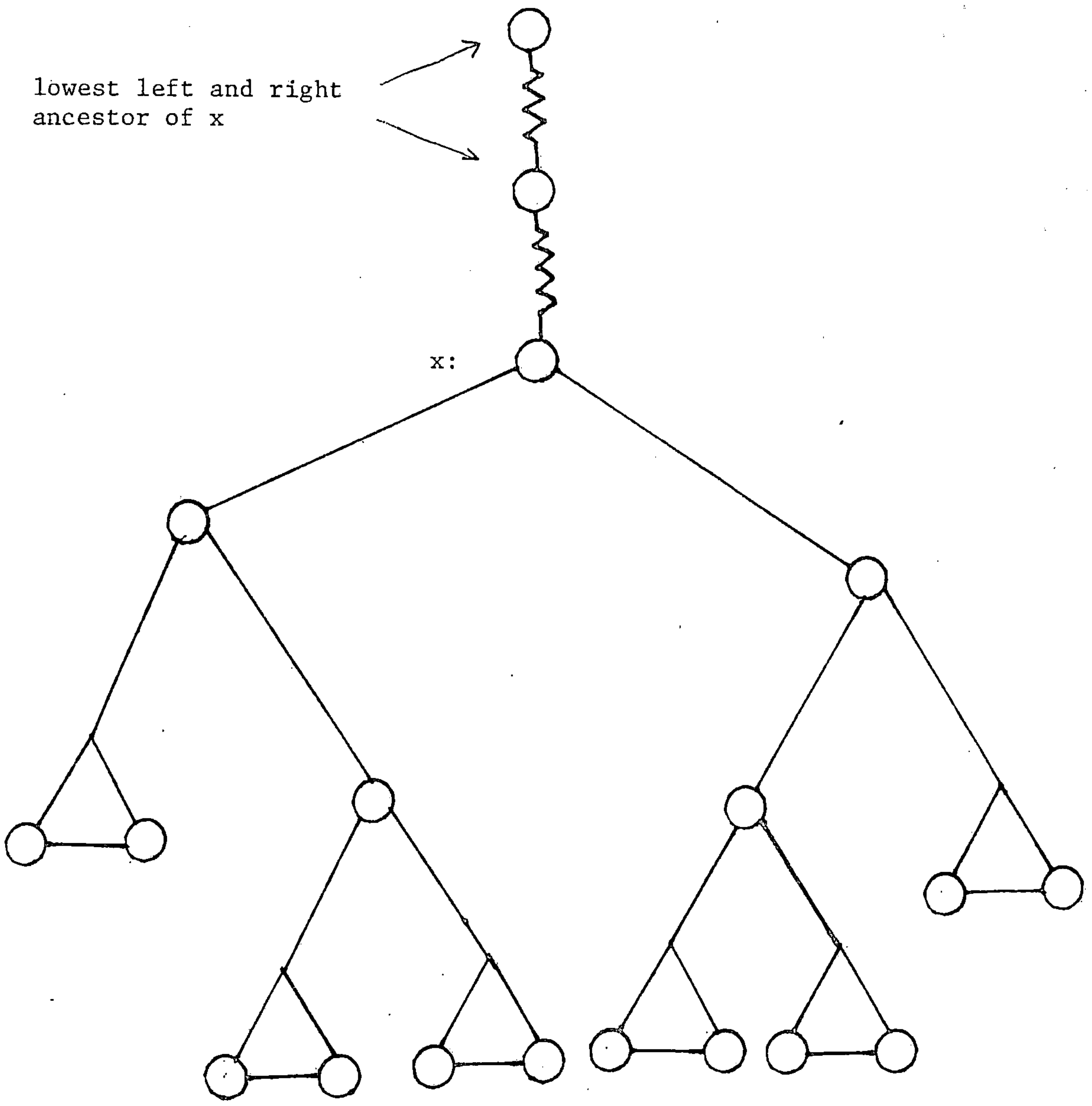


Figure 4. Rotation. Here a curvy line denotes an arbitrary path.

k, with a maximum relative error of δ, in the same time bound. (In Section 3 we will show that the required information about the number of descendants of each node can be efficiently maintained.)

Proof. First we prove the statement about the search time. Consider the following algorithm.

```
procedure SEARCH(f,k);
begin comment search for key k from the finger node f, which is
  assumed to be a leaf;
  if k = KEY(f) then return f;
  assume without loss of generality that k>KEY(f);
  PHASE1: while f≠null and f does not subtend k do
    f := LRA(f);
  if f = null then return a failure indicator; comment this occurs if
    the PHASE1 loop went off the top of the tree;
  f := LEFTMOST(RIGHT(f));
  PHASE2: while f does not subtend k do f := PARENT(f);
  PHASE3: perform a conventional search for k in the tree rooted at
    f;
end;
```

See Figure 5. It is not hard to see that this search procedure will find k if it is in the tree; we will show that it uses O(log d) time, where d is the linear distance from f to k. First consider PHASE1. Assuming that at least one iteration of the while-loop was performed, let f_0 be the last value of f that did not subtend k. Then since all leaves of the right subtree of f_0 lie between f and k, the number of iterations can be seen to be

$$O(\text{height of } f_0) = O(\log(r(f_0))) = O(\log d).$$

For PHASE2, again assume that at least one iteration is performed, and let f_1 be the last value of f during these iterations that did not subtend k. Then all leaves of the subtree rooted at f_1 lie between f and k, so the number of iterations is

$$O(\text{height of } f_1) = O(\log(r(f_1)) = O(\log d).$$

The time in PHASE3 is also bounded above by the height of f_1, so the total time is O(log d).

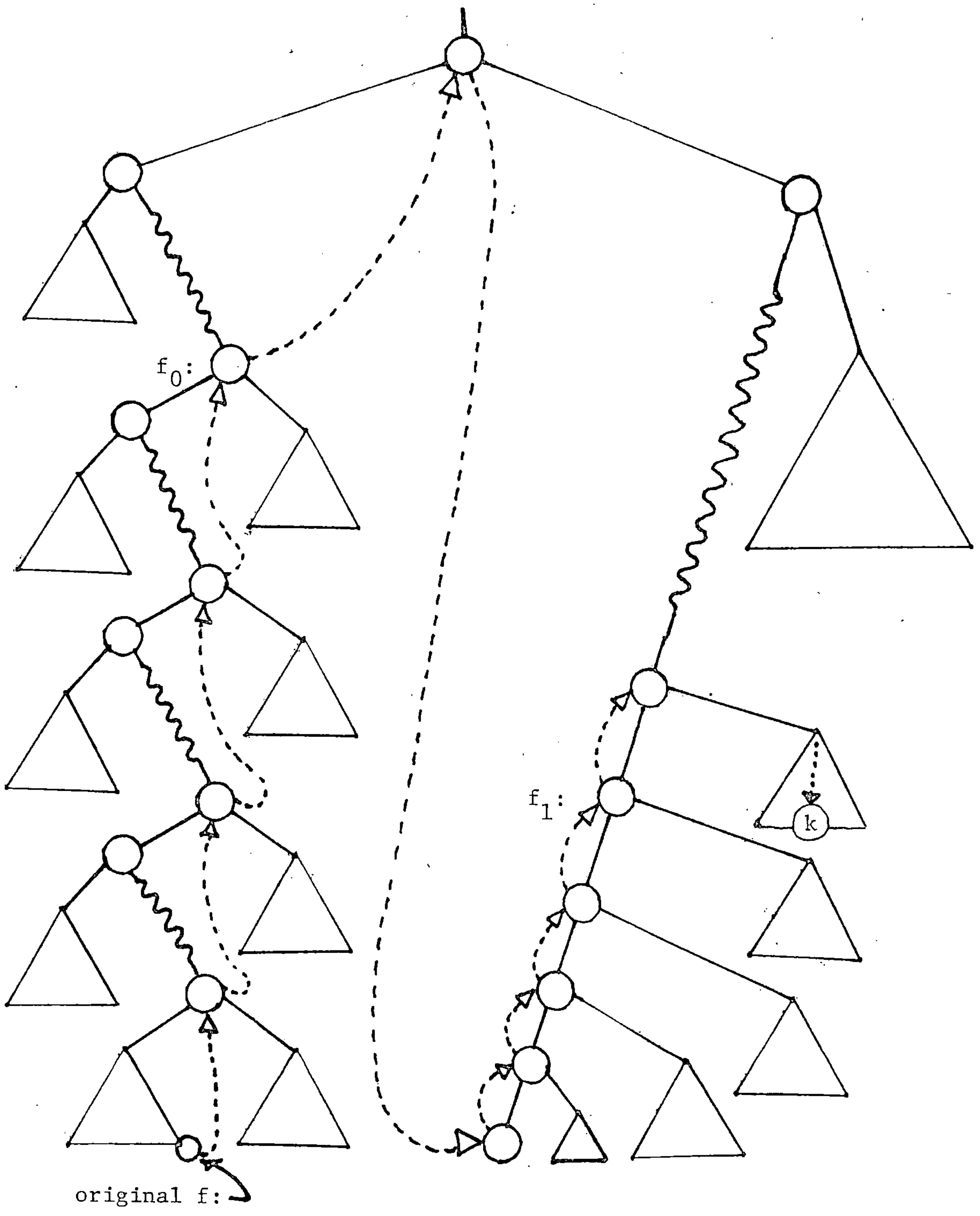


Figure 5. A search from a finger f to a key k. Curvy lines here denote arbitrary left-leaning or right-leaning paths. Arrows with dotted lines denote the movement of the search.

Now consider the second part of the lemma, which deals with finding the approximate number of keys between f and k. It is not hard to see that we can obtain the desired result without increasing the asymptotic search time; the central idea of the algorithm is to sum the fields giving approximate descendant counts, over an appropriate set of nodes chosen as the search algorithm proceeds. We omit the details. []

Note that the problems of searching and updating are largely independent [BT78]. (We are grateful to [R79] for discussing this separation with us.) Since the tree is threaded, searching from a finger is efficient. We must now determine how to update a tree once we know where an insertion or deletion is to occur. This appears to give rise to a question of interest in its own right, namely, how much time is required to update a tree once we know where the insertion/deletion is to take place. Some classes of structures require only a small number of changes to the tree. For example, an AVL tree can always be restored to balance by a single rotation after an insertion. On the other hand, locating the point at which the rotation should take place may require $\Theta(\log n)$ time; moreover, deletions may require $\Theta(\log n)$ rotations [K73]. It is known that for bounded balance trees, a string of n insertions and deletions in an initially empty tree can be done so as to require only $O(n)$ rotations [BM78]; however, one may again need to spend a lot of time deciding where to perform these rotations. For 2-3 trees, a string of n insertions in an initially empty tree requires only $O(n)$ total time if we are given the positions at which the insertions are to take place. However, a <u>single</u> insertion may require $\Theta(\log n)$ time; moreover, even if we only consider total time for n operations, if we allow insertions and deletions to be interspersed it is no longer $O(n)$ [BT78]. In this paper we describe a structure in which a string of n insertions and deletions may be performed in $O(n \log^* n)$ time, provided we are given the positions of the insertions and deletions. In [Ha79] the first author shows how to guarantee $O(\log^* n)$ times for individual operations, by further exploiting the concepts in the current paper. (Recently Scott Huddleston [Hu79] has shown how 2-3-4 trees can support a sequence of n insertions and deletions with total time of $O(n)$, but individual operations may require $O(\log n)$ time.)

3.0 The update algorithm

3.1 The algorithm

Unfortunately, there are two fundamental problems we encounter when we try to use threaded bounded balance trees to come up with an efficient data structure with fingers.

a) Since our desired time bounds do not allow us to scan the entire path from the root to the inserted node, how can we locate the nodes to be rebalanced?

b) Even if we could quickly locate candidates for rebalancing, how would we maintain the field which tells the rank of the node? A single insertion will change this value for $O(\log n)$ nodes.

To solve (a) we will devise a scanning strategy which enables us to scan only $O(\log^* n)$ nodes per insertion, and still detect nodes which are going out of balance before they cause problems. To solve (b) we will not maintain the rank of a node exactly; rather, each node will have a field $\hat{r}$ which contains an approximation to its rank. Each time we scan a node, we will update this approximate value using the $\hat{r}$ values of its children; we will need to be sure to do this often enough so that the approximation is adequate. Below we sketch a systematic way of scanning the nodes, which we will prove to be sufficient.

Define a sequence b_i by

$$b_i = 2^{(b_{i-1}^{1/3})} .$$

The choice of b_0 will be deferred; assuming b_0 is large enough, one readily verifies that $\min\{i \mid b_i > n\} \sim \log^* n$. We will divide the tree into $\Theta(\log^* n)$ <u>plies</u> by classifying nodes according to their rank in relation to the sequence b_i. The nodes which occur at the boundary between two plies will be called <u>ply boundary</u> nodes. The ith ply boundary is defined as

$$B'_i = \{v \text{ in } T \mid r(v) \leq b_i < r(\text{PARENT}(v))\}.$$

One easily establishes that each ply boundary is a cut in T. We may now formally define the ith ply to be

$P'_i = \{v \mid v$ has a descendant in B'_i and an ancestor in $B'_{i+1}\}$.

Note that it follows that ply boundary nodes occur in the plies both above and below the boundary. (Note also that the plies used here are similar to the <u>rank groups</u> employed in [AHU74] in a partial analysis of a UNION-FIND data structure.) In order to obtain the time bounds we desire, we will not be able to maintain the plies exactly as defined here. Rather we will maintain approximate plies P_i and ply boundaries B_i. These approximate ply boundaries will also be cuts, and the nodes on these cuts will be at a distance of at most 1 from a node in the exact ply boundary. Use of these approximate boundaries allows us to wait a bit before adjusting the boundary as the data structure changes. Just as we defined β earlier to measure how badly a node needed rotation, we now define a function λ to measure how badly a ply boundary needs to be adjusted. If w is the parent of v, let

$$\lambda(i,v) = \max \begin{cases} 0 \\ (\ r(v)/b_i - 1\ )\ /\ (\ (1-\alpha')^{-1} - 1) \\ (1 - r(w)/b_i)\ /\ \alpha' \end{cases}$$

The proof of the following lemma is not difficult and will be omitted.

<u>Lemma</u> <u>5</u>. Assume the tree is α' balanced. Then node v is on B'_i iff $\lambda(i,v)$ is zero; moreover, if $\lambda(i,v)$ is less than 1, v is at a distance of at most one from B'_i.

Each node x in B_i will have a field SCANNER(x) which points to some ancestor of x in P_i. The set of ancestors of x in P_i will be called the <u>orbit</u> of x. Each time an operation involving x is done, we will move SCANNER(x) up one step in its orbit. (When a scanner moves off the top of its orbit, it returns to the bottom.) As the scanner circles around this orbit, it updates $\hat{r}$ values and checks for nodes going out of balance.

The algorithm is presented in pidgin-Algol in the appendix. Some further conventions are useful. For a node x in a ply P_i, we will let INDEX(x) equal i. If x is in boundary B_i, B_ANCESTOR(x) will be the ancestor of x, if any, in B_{i+1}; for nodes not on a boundary, B_ANCESTOR is undefined. We will let $\hat{\beta}$, $\hat{\rho}$, and $\hat{\lambda}$ denote the values obtained for β, ρ, and λ if we base the calculation on $\hat{r}$ rather than on r. Also, for convenience, we have suppressed a number of details in the algorithm. In particular, for all nodes of rank less than b_0, the data structure is a conventional bounded balance tree.

Since b_0 is a constant, this does not affect the asymptotic complexity. Also, we have not given details on the process by which ply boundaries are created and destroyed as the rank of the root passes through one of the b_i values. This process, and the analysis of its complexity, are similar to that for the maintenance of the interior ply boundaries.

3.2 Correctness of the update algorithm

We now give a proof that the algorithm correctly maintains the data structure. This proof is complicated somewhat by the fact that certain assumptions interact. In particular, we wish to make sure that the $\hat{r}$ values give good approximations to the true rank, in order to guarantee that the rebalances are done appropriately; on the other hand, we need to assume that the tree is balanced in order to prove a good bound on the errors in $\hat{r}$. In order to state the argument clearly, we will define three sets of propositions which will be involved in an inductive proof.

Q(k) will denote the proposition that at the end of the kth update, and at all previous points in time, the tree was $(\alpha'/2)$-balanced and all λ values were less than 3. Note that this is a weaker condition than that which we ultimately wish to prove about the tree; however, it is strong enough to guarantee that the height of a node varies logarithmically with its rank, and that all ply boundaries are maintained within a distance of O(1) of their correct position.

R(k) will denote that proposition that at all times through the end of the kth operation, the relative error $|\hat{r}(x)-r(x)|/r(x)$ is $O(b_0^{-1/3})$ for all nodes x in the tree. (The implied constants in O-notation will be independent not only of n, but also of b_0; the constant is not independent of α. Thus this proposition says that we may choose b_0 large enough to make the relative error as small as we like.)

S(k) will denote the proposition that at all times through the end of the kth operation, the β and λ values remain less than 1. Note that this is just a stronger version of Q(k).

Lemma 6. For large enough b_0, Q(k)==>R(k). Thus, if the balances and boundaries are maintained even approximately, we can make the relative error in $\hat{r}$ as small as we like by choosing b_0 to be large enough.

Proof. First consider nodes on some ply P_i. Let r_0 be the smallest possible rank of a node on this ply; note that r_0 is $\Theta(b_i)$. Let x be some node on this ply. Let m be the number of operations involving x which have been performed since the last time that $\hat{r}(x)$ was updated. Then it must be the case that during the last m operations, x has not been scanned, and has not actively participated in a rotation in a way which changed its set of descendants. (This is true by inspection of the SCAN and REBALANCE procedures.) Note that x has $O(r(x)/b_i)$ descendants on B_i, and each of these has a scanner whose orbit is of length $O(b_i^{1/3})$. Note also that these scanners have been advanced a total of at least m steps since x was scanned. Using a pidgeon-hole type of argument, we may conclude that

$$m = O(r(x)\ b_i^{-2/3}). \tag{1}$$

Now let e(r) be the maximum value of the relative error (i.e., $|\hat{r}(x)-r(x)|/r(x)$) of a node x of rank r. Note that $\hat{r}(x)$ can be inaccurate for two reasons:

a) When $\hat{r}(x)$ was updated, the approximate values in its children were used. Since the maximum possible rank of x at this point was $r(x)+m$, the rank of its children was at most $(1-\alpha'/2)(r(x)+m)$. Thus the absolute error at this point was bounded by

$$(r(x) + m)\ e((1-\alpha'/2)\ (r(x) + m))$$

Assuming $e(r)\leq 1$, this is bounded by

$$r(x)\ e((1-\alpha'/2)\ (r(x) + m)) + m. \tag{2}$$

Using (1), if we let p be half way between 1 and $(1-\alpha'/2)^{-1}$, we can choose b_0 so that (2) is bounded by

$$r(x)\ e(r(x)/p) + m.$$

b) The value of r(x) has changed, by at most m, since $\hat{r}$ was last updated. The absolute error due to this change is, of course, at most m.

In view of (1), (a), and (b), we may now bound the relative error by

$$e(r) \leq e(r/p) + O(b_i^{-2/3}),$$

which implies that

$$e(r) \leq e(r_0) + O(b_i^{-2/3}) \log_p (r/r_0)$$

$$= e(r_0) + b_i^{-2/3} O(b_i^{1/3}) = e(r_0) + O(b_i^{-1/3}).$$

Thus the increase in the relative error over a single ply P_i is $O(b_i^{-1/3})$. If we sum for i running from 0 to infinity, this converges to $O(b_0^{-1/3})$. Our earlier assumption that $e(r)<1$ is now justified. (This is, of course, not a circular argument; rather, it is an implicit simple induction.) []

Lemma 7. If we choose b_0 large enough, then R(k)==>S(k). Thus, assuming the relative errors in $\hat{r}$ are small enough, the tree remains balanced and the boundaries remain within a distance of 1 of their exact positions.

Proof. Suppose that the relative errors in $\hat{r}$ are within the e of Lemma 1 and are small enough so that the absolute error in $\hat{\beta}$ is at most 0.25. Now the only time $\hat{\beta}(z)$ changes is when $\hat{r}$ of z or of its left child changes. But these can change under only two conditions:

a) We are in procedure SCAN. In this case we check whether $\hat{\beta}$ exceeds 0.5; this means that we will detect any node for which $\beta(z)>0.75$. Now if b_0 is large enough, a single insertion/deletion will change β by less than 0.25, so when we detect $\hat{\beta}(z)>0.5$ we still have $\beta(z)<1$. Then, by Lemma 1 and the definition of β, node z will be returned to a state in which $\beta(z)=0$.

b) We are in procedure REBALANCE. In this case, $\beta(z)$ will, again, be 0 after the call.

A somewhat similar proof holds for the $\hat{\lambda}$ values. []

Lemma 8. If we choose b_0 large enough, S(k)==>Q(k+1). Informally, this says that if the data structure is in a state of strict balance, it cannot cease to be even roughly balanced after one insertion/deletion.

Proof. A single insertion or deletion can change the values of $\beta(x)$ or $\lambda(x)$ by an amount inversely proportional to the rank of x. Thus if b_0 is large enough, the desired condition is easily established for nodes x which do not participate in rotations or boundary adjustments. For nodes which do participate in such activities, the affected $\hat{\beta}$ and $\hat{\lambda}$ values become 0, so again the desired condition is enforced. □

<u>Theorem 1</u>. If we pick b_0 large enough, then the tree is correctly maintained. That is, R(k) and S(k) are true for all k.

<u>Proof</u>. This follows by an easy induction using the past three lemmas. □

<u>Theorem 2</u>. If we choose b_0 large enough, the worst case time complexity of MAINTAIN, when averaged over the entire string of operations, is $O(\log^* n)$.

<u>Proof sketch</u>. The only difficult part is bounding the work done during rebalancing and boundary movements. We use an accounting argument to show that in fact this part of the work is O(1). Define the quantity I(T) to be the sum, over all x in T, of the following quantities:

a) $\beta(x)*(r(x) / (\log(r(x)))^3)$.

b) If x is a boundary node, say in B_i, and if $\lambda(i,x)>0.2$, add in

$$(\lambda(i,x)-0.25)*(r(x) / (\log(r(x)))^3).$$

This quantity I(T) will give a measure of how much work appears to be necessary soon to balance nodes and adjust boundaries. Now it is not hard to establish that a single insertion/deletion, before any rebalancing, can change $\beta(x)$ and $\lambda(x)$ for any node x by at most an amount inversely proportional to r(x). Using this observation and the fact that ranks increase exponentially as we traverse any path toward the root, we may easily conclude that a single insertion or deletion increases I(T) by only O(1).

On the other hand, a rebalance or boundary adjustment requires an amount of time proportional to the decrease in I(T). To see this, assume that b_0 is large enough so that the errors in $\hat{\beta}$ and $\hat{\lambda}$ are at most 0.2. Now note that a boundary movement decreases $\hat{\lambda}$ from at least 0.5 to 0; thus λ in this part of the boundary will decrease from at least 0.3 to at most 0.2, so the new contribution to I(T) is 0. When a rebalance is done, $\hat{\beta}(x)$ is at least 0.5, so $\beta(x)$ is at least 0.3, and $\beta(x)$ is decreased to 0 by the rebalance; moreover, all nodes whose β value is affected by the rotation will have zero β values afterwards, by Lemma 1 and the definition of β. The factor $(r(x)/ \ln(r(x))^3)$ used in defining I(T) is a bound on the size of the partial subtree which needs to be modified during a rebalance or boundary adjustment at x; this follows by inspection of the algorithm, and by the manner in which we defined the plies and the sequence b_i. Thus the cost of the rebalance or boundary adjustment is covered by the decrease in I(T). []

References

[AHU74]✓ Aho, A., Hopcroft, J., and Ullman, J., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[BM78] 0 Blum, N., and Mehlhorn, K., "On the Average Number of Rebalancing Operations in Weight-Balanced Trees," Universität des Saarlandes, A-78/06, June 1978.

[BT78] C Brown, M. R., and Tarjan, R. E., "Design and Analysis of a Data Structure for Representing Sorted Lists," Technical Report STAN-CS-78-709, Computer Sciences Department, Stanford University, December 1978.

[GMPR77]0 Guibas, L. J., McCreight, E. M., Plass, M. F., and Roberts, J. R., "A New Representation for Linear Lists," *Proc. Ninth Annual ACM Symposium on Theory of Computing*, (May 1977), pp. 49-60.

[Ha79] 9 Harel, D., "A Finger Data Structure with Guaranteed Time Bounds per Operation," draft.

[Hu79] 0 Huddleston, S., private communication.

[K73] Knuth, D., *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, 1973.

[L79] ✓ Lueker, G. S., "A Transformation for Adding Range Restriction Capability to Dynamic Data Structures for Decomposable Searching Problems," Technical Report #129, University of California at Irvine, February 1979.

[NR73] 0 Nievergelt, J., Reingold, E. M., "Binary Search Trees of Bounded Balance," *SIAM J. Comput.*, 2:1 (1973), pp. 33-43.

[R79] Rodeh, M., private communication to Dov Harel, 1979.

[T75] Tarjan, R. E., "Efficiency of a Good But Not Linear Set Union Algorithm," *JACM*, 22:2 (April 1975), pp. 215-225.

[W78] Willard, D. E., *Predicate-Oriented Database Search Algorithms*, Ph.D. thesis, Aiken Computation Laboratory, Harvard University, 1978; available as technical report TR-20-78.

Appendix: Pidgin-Algol for the tree maintenance procedures

```
procedure MAINTAIN(u);
begin comment assuming a node has been inserted or deleted, and that u is its
  ancestor on $B_0$, do the necessary maintenance operations on the tree;
  using the B_ANCESTOR fields, let $u_0, u_1, \ldots, u_g$ be the ancestors of u which
    lie on boundaries;
  for i := 0 to g do
    begin
      if $\hat{\lambda}(i, u_i) < 0.5$
        then SCAN($u_i$)
        else
          MOVE_BOUNDARY: begin
            comment we must update the boundary. (Currently $u_i$ is at a
              distance of 1 from the exact boundary);
            make the necessary changes in the data structure to move the ply
              boundary up or down so that all $\hat{\lambda}$ values of the new boundary
              nodes are 0;
            comment the above process is not described in detail since it is
              clear that it can be done in $O(b_i/b_{i-1})$ time--the entire portion
              of the tree consisting of the descendants of $u_i$ in $P_{i-1}$ contains
              only $O(b_i/b_{i-1})$ nodes. (The major cost of this operation stems
              from the fact that all descendants in $B_{i-1}$ of a new ply boundary
              node u in $B_i$ must have their B_ANCESTOR field set to point to
              u);
            for each new boundary node u do SCAN_ORBIT(u);
          end
    end
  end;

procedure SCAN_ORBIT(u);
begin comment scan the entire orbit of u, and set the SCANNER of u to point to
  u;
  SCANNER(u) := u;
  repeat SCAN(u) until SCANNER(u) = u;
end;
```

```
procedure SCAN(u);
begin comment advance the scanner of u, and update the rank of the newly
  scanned node;
  i := INDEX(u);  s := SCANNER(u);
  s := SCANNER(u) := (if PARENT(s) is in $B_{i+1}$ then u else PARENT(s));
  $\hat{r}$(x) := $\hat{r}$(LEFT(x)) + $\hat{r}$(RIGHT(s));
  if $\hat{\beta}$(s) $\geq$ 0.5 then REBALANCE(s);
  if $\hat{\beta}$(PARENT(s)) $\geq$ 0.5 then REBALANCE(PARENT(s));
end;


procedure REBALANCE(x);
begin
  i := INDEX(x);
  do the appropriate rotation to bring $\beta$(x) back to 0;
  for each node y, other than x, which actively participated in the rotation,
    do
    $\hat{r}$(y) := $\hat{r}$(LEFT(y)) + $\hat{r}$(RIGHT(y));
  comment the following loop is done to make sure that scanners remain in
    their orbits;
  PATCH_SCANNERS:  for each node in the ply boundary just below x do
    if SCANNER(w) points to a node which actively participated in the rotation
      then SCAN(w);
  PATCH_BOUNDARIES:  if any nodes which actively participated in the rotation
    were boundary nodes or parents of boundary nodes, then
    begin
      adjust the portion of the tree between x and $B_i$ to guarantee that the
        boundaries have $\hat{\lambda}$ values of 0;
      for each new boundary node u do SCAN_ORBIT(u);
    end;
end
```