FAST UPDATES OF BALANCED TREES WITH A GUARANTEED TIME BOUND PER UPDATE

Dov Harel

University of California, Irvine
Irvine, CA 92717

Technical Report #154
August 1980

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 Earl C. Anthony Fellowship and National Science Foundation grant MCS79-04997.

FAST UPDATES OF BALANCED TREES WITH A GUARANTEED TIME BOUND PER UPDATE

Dov Harel

Abstract

We show how binary trees of bounded balance can be maintained so that the time to perform each individual update (insertion or deletion) in a tree of n nodes is $O(\log^* n)$, once the location of the update is known. This is the first example of a balanced tree scheme in which the time of each individual insertion/deletion is substantially lower than the height of the tree, which is $\Theta(\log n)$. Such a data structure is useful in situations where the search time may be much lower than $O(\log n)$. The main example of such a situation is the framework of data structures which support fingers. In addition to guaranteed time bounds per update, our structure supports arbitrarily many fingers and approximate range queries.

1.0 Introduction

A finger is a point in a file near which searches and updates could be performed particularly efficiently. An overview of the literature concerning fingers is given in [HL79], which contains a particular implementation as well. For other implementations see [GMPR77,BT78,Hu79,MS79]. As it turns out, the problems of searching and updating are largely independent as was already pointed out in [BT78]. To demonstrate this independence consider first two trivial extreme cases. The first example is a linearly ordered array. In an array searches could be done most efficiently using indexing. In particular, binary search, interpolation search, and finger searches are all possible in arrays. Updates on the other hand cause problems since we have to shift data around, which results in excessive time complexity, at least in the worst case for a single update. Another extreme case is given by linear, doubly linked lists, in which individual updates are extremely easy, and take only $O(1)$ time. On the other hand searching a linked list may be time consuming, no matter where the search starts. So we see that it is not difficult to construct examples of data structures in which only one of the above operations could be done efficiently; it is the combination of both of

them which may be challenging.

Balanced tree schemes provide the perfect example of data structures which can be updated, as well as searched, fairly efficiently. Traditionally searching a balanced tree with n nodes, takes O(log n) time, since the search starts at the root of the tree and is proportional to the height of the tree. Using fingers, however, the search starts at the bottom of the tree and depending on the problem may be much faster than $\Theta(\log n)$.

This gives rise to a question of interest in its own right, namely, how much time is required to update a tree once the location of the insertion/deletion is known. For example, only a single rotation is needed in order to restore an AVL tree to balance after a single insertion. Locating the point at which the rotation should take place, on the other hand, may take $\Theta(\log n)$ time; moreover deletions may require $\Theta(\log n)$ rotations [K73]. In [BM78] it is shown that $\Theta(n)$ rotations are sufficient for keeping a BB(α) tree in balance throughout a string of n insertions and deletions which are performed on an initially empty tree. Once again, it may take a long time to find the locations in which the rotations are to be performed. In 2-3 trees the total time for n insertions in an initially empty tree is $\Theta(n)$, although an individual insertion may take $\Theta(\log n)$ time; moreover the time bound is not valid when deletions and insertions are interspersed [BT78]. Huddleston [H79] has shown how 2-3-4 can support a sequence of n insertions and deletions in an initially empty tree in O(n) time, but individual operations may still require $\Theta(\log n)$ time. A similar result for a variant of B-trees was proved independently by Maier and Salveter [MS79]. In [HL79] it is shown how a string of n insertion and deletion on an initially empty bounded balanced tree, could be done in $\Theta(n \log^* n)$ time, provided that we are given the positions of the insertions and deletions. In this writeup we show how to guarantee $\Theta(\log^* n)$ time for each insertion or deletion once their position is known. The two structures, which are based on threaded bounded balance trees, support both fingers and approximate range queries.

This report is written as an extension of [HL79], and thus it is not self contained. We assume familiarity with the concepts which are presented in the above paper, including plies, ply boundaries, scanners, orbits, β and λ values and their basic properties, as well as the algorithm. Although understanding the proof of the main theorem in the above paper is not

necessary in order to follow our arguments, the reader would probably gain an intuitive understanding of the concepts in this paper by reading [HL79] first, and is encouraged to do so. From now on we will use all of the above without further apologies. Some changes will be useful for establishing a time bound per operation. Perhaps the most significant change in the data structure is the introduction of a new representation for the scanners which is the topic of the next section.

1.1 Scanners and Their Representation

The scanning operations, exclusive of rebalancing and boundary adjustments, use only $O(\log^* n)$ time in the worst case so they cause no difficulty. The statement PATCH_SCANNERS in REBALANCE is more troublesome. Recall that a rotation at x may move scanners out of their orbits. This may happen due to the fact that for active participants of the rotation, other than x, the set of boundary descendants changes. Since the number of scanners pointing to active participants of a rotation at x may be $\Theta(r(x)/b_i)$, for $x \in P_i$, it may be time-consuming to use the brute force method of checking each individual scanner and restoring it if needed. To overcome this problem, with each node x we keep a set S(x) containing all scanners pointing to x. The sets are represented using a structure similar to the UNION_FIND structure in [T75], with weighted union but without path compression. This approach enables us to restore all scanners to their orbits in O(1) time; for each active participant $y \neq x$ of the rotation we perform:

```
begin
    S(x) := S(x) ( ) S(y);
    S(y) := Ø;
end;
```

We perform the same block also when we scan the node y moving upwards to its parent x. The effect of this action is that the complete set of scanners which were at y move upwards to x. Thus we advance a whole set of scanners at a time, as opposed to a single scanner, which can only improve the frequency of scanning. In [HL79] a scanner was a pointer to a node of the tree T. Here by a scanner we mean a new kind of record. If s is a scanner we say that s belongs to the boundary node x if SCANNER(x) = s, in which case x is referred to as the owner of s. In addition to a field which contains the value of its weight in the UNION_FIND structure every scanner s will have a pointer to its parent in that tree, which is designated by FATHER(s). If s is a root of a

UNION_FIND tree then FATHER(s) is either null or it is a node of T. We say that s points to FATHER(s) and that it ultimately points to FATHER(r) where r is the root of the UNION_FIND tree which represents the set to which s belongs. Note that s always ultimately points to either a node of T or to null. There are two new problems which are caused by this approach:

a) A scanner s does not necessarily point to a node of T. Rather it may point to another scanner in the case that it is a set element which does not appear at the root of a UNION_FIND tree. In the latter case, finding the node of T to which s ultimately points could be time consuming.

b) In certain cases we have to keep scanners which have no owner on a boundary. For example, when a scanner s reaches the top of its orbit it should return to the bottom of the orbit. At this point, however, s may be the root of a large UNION_FIND tree and thus may have many other scanners pointing to it. In this case setting FATHER(s) to the bottom of the orbit could move other scanners in the set far from their orbits. To avoid this phenomenon the owner of s is given another another scanner s' which will point to the bottom of the orbit. Unfortunately, this increases the cardinality of the set of scanners still in the structure which ever belonged to a node x. It is thus conceivable that the space complexity of our structure is nonlinear. Moreover, if the size of the scanner sets S(x) becomes excessive it may cause the traversal of the UNION_FIND structure to be overly time consuming.

As for problem (a), whenever both f=FATHER(s) and g=FATHER(FATHER(s)) are scanners we do a single FIND_STEP (the assignment FATHER(s):=g). As we shall see later, changing the recurrence for the b_i's to

$$b_i = 2^{(b_{i-1}^{1/4})}$$

will cover the extra cost of the required FIND_STEPS.

With regard to problem (b) we introduce the following terminology. When we adjust a portion of the boundary in the main block of REBOUND, scanners which belonged to members of the old boundary have no owners in the new boundary. We shall refer to such scanners as inactive, and to the process in which a scanner s loses its owner as deactivating s. A scanner s is also deactivated when it reaches the top of its orbit. In this case in addition to deactivating s, we set FATHER(r) to the null pointer, where r is the root of the scanner set to which s belongs. The last operation will be referred to as abandoning the set of scanners represented by r. Members of an abandoned set are also called abandoned, and they may be either active or inactive. For any node x $\in$ T let S_OUT(x) denote the set of scanners which have ever belonged to x, but which have not yet been abandoned. Notice that it makes sense to talk about S_OUT(x) for nodes x which are not presently members of a boundary. The members of S_OUT(x), though not abandoned, may be inactive. In fact, at most one member of S_OUT(x) is active, namely, SCANNER(x) for boundary nodes x.

We enforce the condition that an active scanner of a node in B_i ultimately points either to a node of P_i or to null. We will show that $|S\_OUT(x)|=O(1)$ and thus not only the time complexity is kept under control but also the space complexity can easily be made linear, by a simple modification of the algorithm, as is shown in the last section of the paper.

2.0 Boundaries and Their Adjustment Problem

We call a set of vertices C a tree cut for T (or simply a cut) if C intersects every path from the root to a leaf of T exactly once. We call a set S a super-cut for T if S intersects every path from the root to a leaf of T at least once. Note that the boundaries which we defined in [HL79] were all tree cuts. Bearing in mind that we are trying to establish a time bound per operation, one can easily observe that boundary adjustments cause a problem. Every time the i^{th} boundary B_i is moved the B_ANCESTOR fields of all the members of B_{i-1} which lie below the new portion of B_i have to be updated. We will not be able to do this all at once but rather we will patch this portion of a tree a little bit at a time. Until this portion of the tree is completely patched up the structure is incomplete in the sense that B_ANCESTOR fields may point to nodes which are not necessarily boundary nodes, or even ancestors. We will show how to guarantee that B_ANCESTOR(x) will reside at a distance of at most O(1) from the real boundary ancestor of x. The problem is

very similar to the problem that arose in the per-operation version of [L79], and we will employ similar techniques to solve it. Notice that REBALANCE operations which are rooted one or two levels above a boundary can damage the boundary since the boundary nodes may not even form a tree cut after such rotations. This is the reason for boundary patches following rotations in [HL79]. In the following section we develop some of the machinery which will help us in dealing with the problems stated above.

2.1 Boundaries Neighborhoods and Areas

We redefine the boundaries as follows. We pick constants c_0, c_1, c_2, and e as will be specified later (Lemma 1). For any i s.t. $b_i \leq n$ and for $0 \leq j \leq 2$ we define

$$\lambda(i,j,x) = \text{DIST}((r(x)/b_i - c_j) , [-e,e])$$

where DIST is the distance on the real line. For the above stated values of i and j we choose $B_{i,j}$ to be some tree cut which satisfies:

$$B_{i,j} \subseteq \{x \mid \lambda(i,j,x) \leq 1\}$$

For x in $B_{i,j}$ the value of $\lambda(i,j,x)$ will serve as an indicator of how badly $B_{i,j}$ needs adjustment in a locale of x. The definition of λ is such that the change in the value of $\lambda(i,j,x)$ due to a single update operation below x is $o(r(x)^{-1})$. The constant e is chosen such that the range of interest in $\lambda(i,j,x)$ is the real interval [0,1], thus enabling a uniform treatment of the λ's and the β's. As before we use $\hat{\lambda}(i,j,x)$ or $\hat{\beta}(x)$ when the calculations are based on $\hat{r}(x)$ rather then on r(x).

Lemma 1. We can pick positive constants G, e, and c_j j=0,1,2 which depend only on α such that:

a) For every i, $B_{i,2}$ lies at least three levels above $B_{i,1}$, and $B_{i,1}$ lies at least three levels above $B_{i,0}$.

b) For every i, and for j=0,1 or 2 every path from the root to a leaf contains at least one node x with $\lambda(i,j,x) = 0$.

c) If the relative error is smaller than G then for each pair of a node y and its parent x we can determine which of the two has the lower λ value, given only their $\hat{r}$ values.

The proof of this lemma is immediate and is omitted. Henceforth we assume that we have picked $\mathcal{G}$, e, and c_j's which satisfy Lemma 1. The i^{th} boundary is defined to be $B_{i,0}$. We will omit the zero at our convenience (i.e. $B_i = B_{i,0}$). The i^{th} ply, P_i, is defined by:

$$P_i = \{x \mid x \text{ has a proper descendant in } B_i \text{ and an ancestor in } B_{i+1}\}$$

For $y \in B_{i,1}$ we define the neighborhood rooted at y by:

$$N(y) = T(y) \cap P_i$$

For $z \in B_{i,2}$ we define the area rooted at z by:

$$A(z) = T(z) \cap P_i$$

We extend the above definitions to nodes which are not members of $B_{i,1}$ (resp. $B_{i,2}$) but belong to some neighborhood (resp. area) as follows. If x is a member of some neighborhood (resp. area) then N(x) (resp. A(x)) denotes the neighborhood (resp. the area) to which x belongs. The sets $B_{i,1}$ and $B_{i,2}$ will be called super-boundaries, and the reason for their definition will be clarified in the following two sections in which we describe the process of adjusting the boundaries and of fixing up the tree after such adjustments are made.

2.2 Boundary Adjustments and Unification

Recall that whenever we adjust the boundary B_i we have to start a traversal of a portion of B_{i-1} to update B_ANCESTOR pointers. We do not want to adjust a portion of the boundary B_i while this traversal is still in progress. Thus we want to "freeze" a portion of B_i below which a traversal of B_{i-1} is taking place. On the other hand if a node of B_i actively participates in a rotation of its parent or grandparent we have to adjust B_i or else it will not be a tree cut any more.

To overcome this problem we "freeze" not only boundary nodes, but the complete neighborhood to which they belong. With each boundary node x we associate a boolean field FLAG(x). We say that x is unified if FLAG(x) = true; otherwise x is called disunified. Once the boundary B_i is adjusted in a locale of x it is possible that for some B_{i-1} descendants y of x, B_ANCESTOR(y) do not point to x. Thus we set FLAG(x) := false for any newly appointed boundary member and start a traversal of the portion of B_{i-1}

below it, the purpose of which is to modify B_ANCESTOR fields. Once this traversal is completed we set FLAG(x) := true and x is unified. Let y be in $B_{i,1}$. The neighborhood N(y) is called eligible for reconstruction if all its B_i members are unified. By a reconstruction of the neighborhood N(x) we mean the following. We first perform rotations on descendants of y until every descendant x of y with

$$r(x)/b_i - c_0 \geq -e$$

has $\beta(x) = 0$. We then choose a tree cut D for T(y) such that $\lambda(i,0,x) = 0$ for every $x \in D$, and replace the portion of B_i below y by D. Finally for each x in the new portion of B_i we set FLAG(x) := false. Adjusting the boundary B_i is done only by reconstruction operations. We reconstruct a neighborhood only if it is eligible. In that sense we say that we "freeze" a neighborhood if it contains a disunified boundary node. Notice however that neighborhoods are not fixed during periods of time in which they are not eligible for reconstruction. Any adjustment of the super-boundary $B_{i,1}$ changes some neighborhood. Adjustments of super-boundaries are performed in the following two cases:

a) We readjust $B_{i,j}$ if, due to a rotation in which one of its members actively participated, it ceases to be a tree cut. This is done in block REBOUND of REBALANCE.

b) We also adjust $B_{i,j}$ if for one of its members x, $\hat{\lambda}(i,j,x)$ grew beyond 0.3. This task is performed by the procedure MOVE.

We say that a node x crossed a super-boundary due to an adjustment if before the adjustment x lay weakly below the super-boundary, and after the adjustment x lay strictly above it, or vice versa. We say that x inherited descendants from y due to an adjustment if there is a leaf z such that y subtended z before the adjustment, and x subtended z immediately following the adjustment. In order to minimize the dependence of the super-boundary maintenance on rotations we would like to keep the amount of modification to a bare minimum, in case (a). On the other hand, we would not like rotations to cause too much damage to super-boundaries. A compromise is given by the following lemma.

Lemma 2. Assume that the tree is α' balanced, and that the relative error $e(x) = |\hat{r}(x)-r(x)|/r(x)$ is sufficiently small. Assume further that due to a rotation RN at u in which some member of the super-boundary actively participated, $B_{i,j}$ ceased to be a tree cut. Then block REBOUND of the procedure REBALANCE modifies the super-boundary $B_{i,j}$ in O(1) time so that it continues to be a tree cut and such that:

a) If y was a member of the super-boundary before the modification, and x is a member of it after the modification s.t. x inherited descendants from y, then $\lambda(i,j,x) \leq \lambda(i,j,y)$.

b) If due to the modification x crossed the super-boundary, then after the modification $\beta(x) = 0$.

Proof. We will first describe the modification of the super-boundary in block REBOUND of REBALANCE in more detail. We will then show that conditions (a) and (b) hold. We define the following predicates:

A(x) means x lies strictly below any active participant of the rotation RN.

B(x) means x is a descendant of u which lies below the super-boundary $B_{i,j}$.

C(x) stands for the conjunction $A(x) \wedge B(x)$.

We define G to be the highest tree cut of T(u) all of whose elements satisfy both A and B. Formally

$$G = \{ x \in T(u) \mid C(x) \text{ and not } C(\text{PARENT}(x)) \}$$

G is denoted by a dotted line in Figure 1. It is easy to verify that G is a tree cut for T(u) both before and after the rotation. To modify $B_{i,j}$ we pick some tree cut H for T(u) which lies between u and G, and then replace the portion of $B_{i,j}$ below u by H. Roughly speaking, we want H to be the best possible choice of a tree-cut between u and G in the sense that the $\lambda(i,j,x)$ values of its members x, are as close to zero as possible. More precisely, for any node x in H if we move H up or down one step, within the stated range, the total value of the $\lambda(x,i,j)$ over $x \in H$ can only increase.

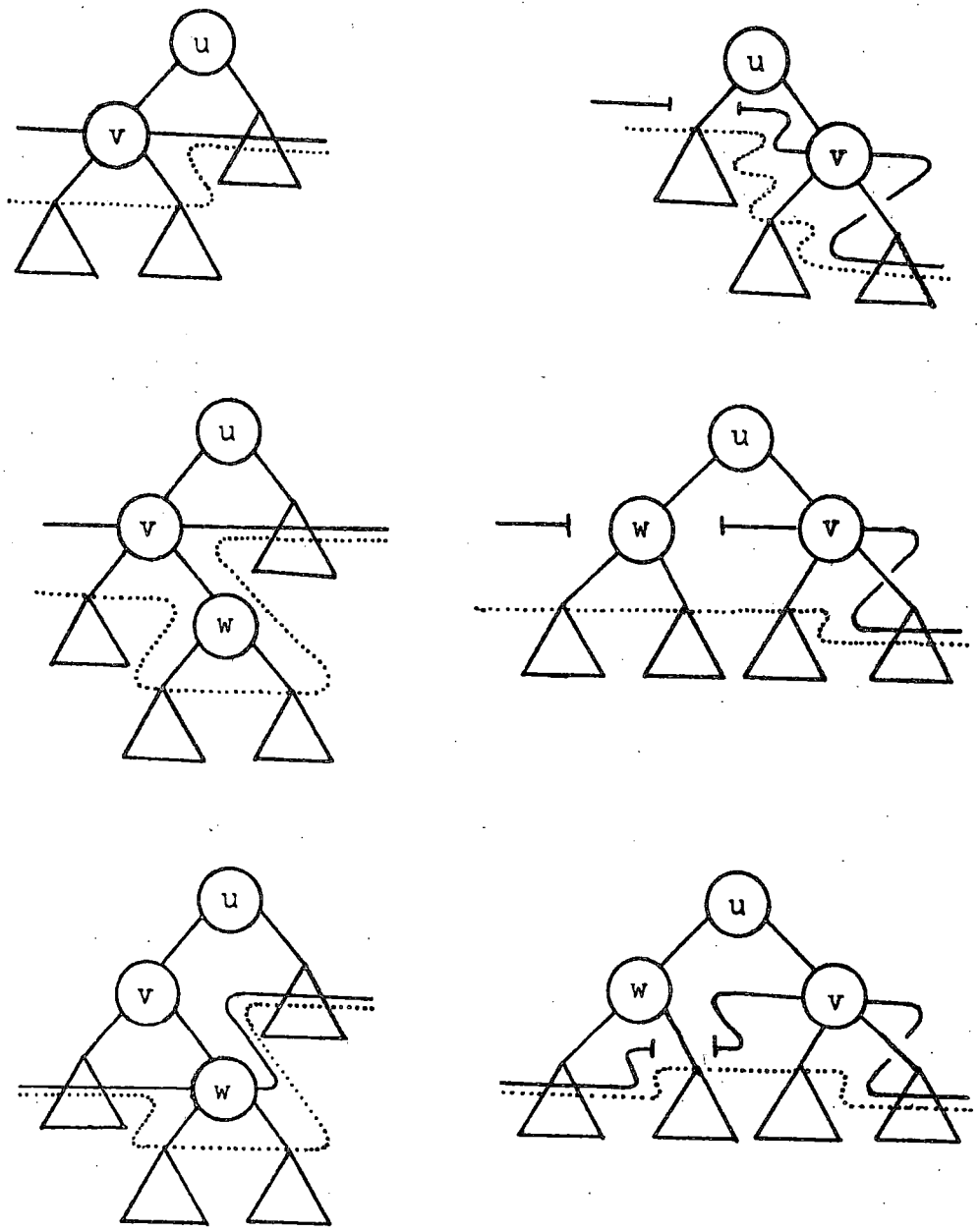

Figure 1 The Effect of a Rotation on Super-Boundaries
Here the super-boundary is denoted by a solid line, and the set G by a dotted line.

Formally, any x in H, with parent p, and children y and z satisfies:

H1. if p is below u then $\lambda(i,j,x) \leq \lambda(i,j,p)$, and

H2. if y and z are above G then $\lambda(i,j,x) \leq \lambda(i,j,y) + \lambda(i,j,z)$

We now show how to construct such an H given only the $\hat{r}$ values. We first construct a super-cut J for T(u) as follows. For each v∈G pick some w on the path P from from u to v s.t. w has a minimal λ value in P, and add w to J. From Lemma 1 it follows that we can construct J given only the $\hat{r}$ values. We then define H to be the subset of all the nodes x in J with no ancestor in J. Since J is a super-cut for T(u) it follows that H is a cut for T(u). From the choice of H it follows that both H1 and H2 hold.

We will now show that H satisfies (a). From definition of λ it follows that if we graph the values of λ on a path from the root to a leaf we notice that the graph is monotonic decreasing until it reaches zero, it stays zero on $\Theta(1)$ nodes, and is monotonic increasing thereafter. Let x be some node of H, and let y be a former member of $B_{i,j}$ from which x inherited descendants. We will distinguish between three cases.

1) $\lambda(i,j,x) = 0$

2) $r(x)/b_i - c_j > e$

3) $r(x)/b_i - c_j < -e$

In case (1) clearly $\lambda(i,j,x) \leq \lambda(i,j,y)$. In case (2) let u and v be the children of x. It follows from Lemma 1 that each child of x either satisfies (2) or has a corresponding λ value of 0. It thus follows from the definition of λ that

$$\lambda(i,j,u) + \lambda(i,j,v) < \lambda(i,j,x)$$

By the definition of H this implies that x must lie on G and thus below y. Notice also that in this case the graph of λ on a path through x to a leaf did not yet pass through zero and thus it is decreasing, which implies again $\lambda(i,j,x) \leq \lambda(i,j,y)$. Case (3) is similar to case (2) and the proof follows by a symmetric argument. (In this case x = u and the graph of λ is monotonic increasing on the way from x to any leaf).

Finally, we want to show that (b) holds. We note that since H lies above G it follows from the definition of G that the only nodes that appear strictly above H and which lay below $B_{i,j}$ before the rotation, are active participants of the rotation. Thus the only possible violation of condition (b) is caused by nodes z which lay strictly above $B_{i,j}$ before the modification and appear below H immediately following the modification. Note that those nodes lie at least three levels above $B_{i,j-1}$. We can thus perform rotations on all those nodes to return their β to zero knowing that no boundary node (or super-boundary node) can actively participate in such a rotation. In other words, although it appears that the procedure REBALANCE may call itself indefinitely, this is not the case. In fact, the execution of a call to REBALANCE that originated inside the REBOUND block, can not enter that block. Thus the execution time is bounded by the size of an area, which is O(1).

[]

The equivalent of Lemma 2 for super-boundary modifications which are caused by a λ value which increased above 0.3 is given by the following lemma.

Lemma 3. Assume that the relative error is sufficiently small that all λ and β values are smaller than 1, and that u $\in B_{i,j}$ with $\hat{\lambda}(i,j,u) \geq 0.3$. Then MOVE(i,j,u) modifies the super-boundary $B_{i,j}$ in a locale of u in a way that:

a) Every newly assigned member x of the super-boundary has $\lambda(i,j,x)=0$.

b) Every node y which crossed the super-boundary as a result of the modification has $\beta(y)=0$ immediately following the modification.

c) No member of $B_{i,j-1}$ actively participated in a rotation which is a result of a call to REBALANCE by MOVE.

The proof of this lemma is similar to the proof of Lemma 2, only simpler, and is omitted.

Let y be in $B_{i,1}$ and z be its ancestor in $B_{i,2}$. After reconstructing N(y) all its B_i members become disunified. In order to remedy this situation we start a traversal of the portion of B_{i-1} below z as will be explained below. A few definitions will be useful. We define the boundary of the neighborhood N(y) by:

$$\bar{N}(y) = N(y) \cap B_i$$

The boundary $\bar{A}(z)$ of A(z) is similarly defined by:

$$\bar{A}(z) = A(z) \cap B_i$$

We keep all the elements of B_i in their natural left-to-right order on a doubly linked list. In addition to that, with each $x \in B_i$ we keep a pointer to its leftmost B_{i-1} descendant. Thus, for every x in B_i we have all its descendants on an ordered list. Once x becomes disunified, we set a special field, AT(x), to point to the beginning of this list. A FIXUP operation on x consists of advancing the pointer AT(x) one step in this list from u to v, and setting B_ANCESTOR(u) := x.

The unification of x consists of FIXUP operations on x. When the pointer AT(x) reaches a node which is not a descendant of x, we know that all the B_{i-1} descendants of x have their B_ANCESTOR field pointing to x and we set FLAG(x) := true. The unification of x takes place as update operations are performed below its $B_{i,2}$ ancestor z. For each update operation below z we perform one fixup on each disunified node in $\bar{A}(z)$.

During the time period in which x is disunified its boundary descendants do not have their B_ANCESTOR fields pointing to it. We will see however that they will point to nodes which lie at a distance of O(1) from x and thus we could quickly find the correct boundary ancestor in O(1) time.

Recall that no neighborhood reconstruction can take place at N(y) before all the members of $\bar{N}(y)$ are unified. This raises the following question. Is it true that by the time N(y) is eligible for another reconstruction all the values $\beta(x)$ of nodes x in N(y) and all the values $\lambda(i,0,x)$ for x in $\bar{N}(y)$ are still less than one? Roughly speaking the number of operations that will cause λ or β values of nodes in the specified locals to grow beyond one is $\Omega(b_i)$. On the other hand it takes only $O(b_i/b_{i-1})$ update operations below z to unify all the members of N(y) which gives a positive answer to the

last question. A more formal treatment of the above fact is part of the proof which is given in the next section. The algorithm, accompanied by a complete list of fields which characterize the data structure, is given in the appendix.

2.3 Linking Records

During the course of modifying the tree it frequently happens that many pointers point to a single target node. When we change the target node we have to modify all those pointers to point to the new target node. One example of such a situation is when the set of descendants changes for an active participant of a rotation. Such nodes may be the target nodes of many scanners which, as a result of the rotation, fell off their orbits. A similar problem arises when we modify B_{i+1}, by reconstructing some neighborhood N, leaving many B_ANCESTOR pointers from B_i pointing to a wrong target node. This by itself is not a serious problem since the new ancestors acquired by vertices in B_i lie within O(1) steps from the old B_ANCESTOR node. Recall however that N continues to change due to rotations in which its root actively participants. It is thus conceivable that the old boundary ancestor will shortly thereafter wonder far away from the new one. The solution to this problem is similar to the one introduced for representing scanners. We introduce a new kind of node called a linking record which has only a single field called LINK. With each element x' of B_i we associate a single linking record, which we call the ancestry record of x', a = A_RECORD(x'). Instead of pointing directly to the boundary ancestor, the B_ANCESTOR field of an element x of B_i will point to the ancestry record a of the ancestor x', and LINK(a) points to x'. Once the neighborhood N of x' is reconstructed, and replaced by M, we do the following. For every linking record a which was associated with some $x \in \bar{N}$, we set LINK(a) to point to some y in $\bar{M}$. Since immediately following the reconstruction, every member y of $\bar{M}$ has $\lambda(i+1,0,y) = 0$, and the β value of y, its parent, and its grandparent is 0, we are guaranteed that the portion $\bar{M}$ of B_{i+1} will remain fixed for $\Omega(b_{i+1})$ operations involving M. And since $|M| = O(1)$ we can still find the correct boundary ancestor of x in time.

Another place where a similar type of problem occurs is the lists of boundary descendants of boundary nodes. Recall that we want to keep with each $x \in B_{i+1}$ a pointer to its leftmost B_i descendant. Once the neighborhood of x is reconstructed we may not be able to update those pointers for the new members of B_{i+1}. One simple solution to this problem is to keep leftmost-boundary-descendant pointers with each member of P_i. Now the problem is that updating B_i causes a change of the target node of many nodes in B_i. The solution to this problem is similar to the previous solution, but simpler. We associate a linking record, called the descendancy record, d = D_RECORD(x) with each boundary node x. We set LINK(d) = x, and for every $y \in P_i$ s.t. x is its leftmost B_i ancestor, we keep a pointer to d. Once x is replaced by y due to an adjustment of the boundary, y inherits the descendancy record d of x, and so LINK(d) := y. Notice that this method is a generalization of the methods used in [HL79] to show that threads can be updated in O(1) time per per update operation.

3.0 Analysis of The Algorithm

By inspection of the algorithm, one can easily convince himself that if the data structure is correctly maintained then the maintenance routine has time complexity $O(\log^* n)$ worst case per operation. The hard part is to show that the data structure is indeed correctly maintained. We say that a node x (resp. a boundary node y) is β-stable (resp. λ-stable) if $\beta(x) < 0.6$ (resp. $\lambda(i,y) < 0.6$ where i=INDEX(y)). In contrast with [HL79], where we enforced the condition that the β and λ values are less than one (Theorem 1 of [HL79]), we will enforce the condition that all β and λ stable. This is a slightly stronger statement, since a β-stable node x is not only α' balanced but it is going to remain α' balanced for at least $\Theta(r(x))$ operations involving it.

3.1 The Predicates

Following [HL79] we define several predicates which will be involved in an inductive proof. The implied constants in the O-notation are independent not only of n but also of the b_i's.

P_0 stands for the property that all β and λ values are less than one and $|S\_OUT(x)| \leq 1$ except perhaps for disunified boundary nodes x for which we have $|S\_OUT(x)| \leq 2$.

P_1 stands for the property that for every i and for every $x \in P_i$ we have $m(x) = O(r(x)\, b_i^{-1/2})$, where $r(x)$ is the rank of x at the last time when $\hat{r}(x)$ was updated, and $m(x)$ stands for the number of operations involving x which have occurred since then.

P_2 stands for the property that the relative error $e(x) = |\hat{r}(x) - r(x)|/r(x)$ is $O(b_0^{-1})$ for every $x \in T$.

P_3 stands for the property that the β and λ values are less than 0.6 and $|S\_OUT(x)| \leq 1$ except for disunified boundary nodes x for which $|S\_OUT(x)| \leq 2$.

For each i, $P_i(k)$ means that the statement designated by the predicate P_i holds at all times through the end of the k^{th} operation. Note that P_3 is a strong version of P_0. The bound of 2 on the size of S_OUT is much stronger than necessary for the results of this section. It will however be useful later when we discuss the space complexity of the data structure. The reader is warned of the somewhat ambiguous notation. P_i may denote either a predicate, or a ply of the tree. This should not cause confusion since from now on a predicate, unlike a ply, will always appear with an argument.

3.2 The Proof

The proof that the data structure is correctly maintained is decomposed into a sequence of lemmas. For the purpose of giving an overview of the proof it will be useful to define the predicates $Q_i(k)$ by the following conjunctions.

$$\text{For } 0 \leq i \leq 3,\ Q_i(k) = \bigwedge_{j=0}^{i} P_j(k)$$

In terms of these predicates the proof goes as follows:

$Q_0(k) \Rightarrow Q_1(k)$ (Lemma 4)

$\Rightarrow Q_2(k)$ (Lemma 5)

$\Rightarrow Q_3(k)$ (Lemma 7)

$\Rightarrow Q_0(k+1)$ (Lemma 8)

The rest of the paper is devoted to the proof. The lemmas are stated in terms of the original predicates, the P_i's. This makes it easier to distinguish between the assumptions and the consequences at each step.

Lemma 4. For a large enough b_0 $P_0(k) \Rightarrow P_1(k)$. Stated informally this means that if the balances and the boundaries are maintained even approximately, and the S_OUT sets are of size O(1), then nodes are updated quite frequently.

The equivalent of this claim in [HL79] is the bound (1) on m, the number of operations involving a node $x \in P_i$, which occurred since the last update of $\hat{r}(x)$. This bound was derived very easily in the old context since whenever a B_i was adjusted below x the entire orbit between the point of the adjustment and B_{i+1} was scanned and updated. Thus we were able to assume that no boundary adjustment of B_i occurred below x since x was last scanned, and the claim follows by a simple pigeon hole type of argument. The situation here is complicated by the fact that we do not scan the entire orbit after every boundary adjustment and thus the set of "pigeon holes" (B_i nodes below x) changes dynamically. The intuition behind the proof is to show that the change in the set of B_i nodes below x is very limited and indeed negligible.

Proof. Let x be some element of P_i, and let t_0 denote the last time when $\hat{r}(x)$ was updated. Recall that $\hat{r}$'s get updated whenever we reconstruct a neighborhood, which is the only way for a node to move from one ply to another. We can thus assume that x did not leave P_i since t_0. Let $op_1,\ldots,op_m$ denote the sequence of the update operations involving x which occurred since t_0. Denote the time at which op_j occurred by t_j for $j=1,\ldots,m$. Recall that that x did not cross a ply boundary during the time period under consideration. Let $S_j = B_i \cap T(x)$ in the time period $[t_j,t_{j+1})$ for $0 \leq j < m$; thus S_j denotes the set of boundary nodes under x between the j^{th} and the j+1-st operations op_j and op_{j+1}.

Notice that every time S_j changes some of its members become disunified. In particular every new member of S_{j+1} (and possibly others) becomes disunified. Let D_j denote the set of nodes which became disunified during the time period $[t_0,t_j)$ and let $E_j = S_j - D_j$. The sequence $\langle D_j \rangle$ is monotonic increasing while $\langle E_j \rangle$ is decreasing. We shall now distinguish between two cases:

CASE I. Some member y of D_m left the boundary before t_m.

Since reconstruction of the neighborhood N to which y belongs takes place only if all the boundary members of N are unified and since the unification of y takes $\Theta(b_i/b_{i-1})$ fixup operations on y it must be that the area A to which y belongs was involved in $\Omega(b_i/b_{i-1})$ operations since y was disunified.

Since y was disunified at the time those operations took place its scanner s was involved in all of them. (That is, each one of them caused either a FIND_STEP(s) or a SCAN(y,s) to be performed.) Now, the condition that all S_OUT sets are of size which is O(1), and the fact that only descendants of a node z can have active scanners pointing to z, implies that

$$\forall z \in P_i \quad |S(z)| = O(b_{i+1}). \tag{1}$$

This in turn implies that the time spent on partial find operations for finding a particular node is $O(\log b_{i+1}) = O(b_i^{1/4})$. Since the tree is α' balanced we know that

$$\text{(the length of y's orbit)} = O(b_i^{1/4}). \tag{2}$$

Combining (1) and (2) we get that after $O(b_i^{1/2})$ operations involving x, x would have been scanned by the scanner of y and since $b_i^{1/2} = o(b_i/b_{i-1})$ it is implied that x indeed must have been scanned by t_m contradicting our assumption.

CASE II. No member of D_m left B_i in the time interval $[t_0,t_m]$.

We shall call op_j an operation of type 1 if it caused no adjustment of B_i (that is, if $S_{j+1} = S_j$) and else we call it an operation of type 2. In the latter case D_j increased by at least $S_{j+1} - S_j$. However the increase in $|D_j|$ is bounded by some constant K times the decrease in $|E_j|$. That is

$$|D_{j+1} - D_j| \leq K\,|E_j - E_{j+1}|. \tag{3}$$

The reason is that the hypothesis $P_0(k)$ implies that the size of a neighborhood is $\Theta(1)$ (i.e. there are constants K_0 and K_1 s.t. $K_0 \leq N \leq K_1$). Notice that $E_j - E_{j+1}$ is the set of nodes which either left the boundary or got disunified due to op_{j+1}. Thus every time an adjustment of the boundary takes place due to a reconstruction of some neighborhood at least one node leaves E_j. Thus clearly (3) holds with $K = K_1$. Now

$$|E_0| = |S_0| = O(r\ b_i^{-1}) \tag{4}$$

By applying (3) repeatedly for all the operations of type 2 we get

$$|D_m - D_0| \leq K\ |E_0 - E_m| = O(r\ b_i^{-1}) \tag{5}$$

Since every new member of B_i gets disunified we have

$$\bigcup_{j=0}^{m} S_j = S_0 \cup D_m \tag{6}$$

and hence

$$\begin{aligned} |\bigcup_{j=0}^{m} S_j| &= |S_0 \cup D_m| \\ &\leq |S_0| + |D_m| \\ &= O(r\ b_i^{-1}) + O(r\ b_i^{-1}) \\ &\quad \text{(from (4) and (5) bearing in mind that } D_0 = \emptyset) \\ &= O(r\ b_i^{-1}) \end{aligned} \tag{7}$$

We now apply an accounting argument as follows. With each member w of the union (6) we associate two counters e(w) and d(w). Initially, all counters are set to 0. Each operation below x is recorded by some counter in the following way. For op_j find the unique $w \in S_{j-1}$ that was involved in op_j. If $w \in D_j$ then increment d(w) by one else increment e(w) by one. (Roughly speaking e(w) counts SCANs or FIND_STEPs of w's old scanner while d(w) counts the same operations with respect to w's new scanner.) Now using an argument similar to (1) and (2) we know that none of the above counters can grow beyond $O(b_i^{1/2})$ before t_m. The reason is that otherwise a complete orbit which passes through x would have been scanned, contrary to our hypothesis.

By (7) the total number of counters does not exceed $O(r\ b_i^{-1})$ so by the pigeon hole principle we get that

$$m = O(r\ b_i^{-1})\ O(b_i^{1/2}) = O(r\ b_i^{-1/2}).$$ []

Lemma 5. For a large enough b_0, $P_0(k) \wedge P_1(k) \Rightarrow P_2(k)$. Informally, if the balances and boundaries are well kept, the S_OUT sets are of size O(1), and the rate of scanning is as stated in P_1, then the relative error can be kept arbitrarily small.

Proof. The proof follows closely the second paragraph of the proof of Lemma 6 in [HL79]. In fact a proof can be obtained from the above paragraph by substituting a 4 for every occurrence of a 3 in the denominator of an exponent. []

Lemma 6. Let x be a node which has been a member of a neighborhood of B_i during some time interval [a,b]. Assume that during this time period the predicates P_0, P_1, and P_2 hold. Assume also that the area to which x belonged was involved in $\Omega(b_i)$ update operations during the stated time period. Let N(x,t) (resp. A(x,t)) denote the neighborhood (resp. the area) of x at time t. Then for every constant $C > 0$ there exists a large enough b_0, a constant D (possibly greater then C), and points in time t' and t", $a < t' < t'' < b$, such that

a) in the time interval [a,t'] at most $D\ b_i/b_{i-1}$ update operations occurred below A(x,t), and

b) in the time interval [t',t"] at least $C\ b_i/b_{i-1}$ updates occurred below A(x,t), while the neighborhood N(x,t) remained fixed (i.e. $\forall t \in [t',t'']\ N(x,t) = N(x,t')$).

Roughly speaking this says that before too long (i.e. within $O(b_i/b_{i-1})$ updates below A(x,t)), N(x,t) will be fixed for a fairly long period of time (i.e. for $\Omega(b_i/b_{i-1})$ updates below A(x,t)).

Proof. Let $C > 0$ be some constant. We want to find a D and points in time t' and t" which satisfy (a) and (b). Let y_0 (resp. z_0) be the root of the neighborhood N(x,a) (resp. of the area A(x,a)). Let P denote the path from z_0 to x_0 at time a. Let $k = |P|$ which is clearly O(1). Let t_i, $i=1,2,\ldots,2k+4$, be the next $2k+4$ points in time following $a=t_0$ at which N(x,t) was modified, and let y_i denote the root of N(x,t) during the time interval $[t_i,t_{i+1})$. We pick $D = (2k+3)C$, and we will show how to find a t' and a t" such that (a) and (b) are satisfied. We distinguish between two cases.

Case 1. For some $j < 2k+3$ at least $C\, b_i/b_{i-1}$ updates occurred below $A(x,t)$ in the time interval $[t_j, t_{j+1})$.

In this case let m be the smallest j which satisfies the above assumption, and choose $t'=t_m$ and $t''=t_{m+1}$. From the choice of m it follows that in the time interval $[a,t')$ less than $(C\, m)b_i/b_{i-1}$ updates occurred below $A(x,t)$. From the choice of D clearly $(C\, m) \leq D$ and thus (a) is satisfied. Also by our assumption at least $C\, b_i/b_{i+1}$ updates occurred below $A(x,t)$ in the time interval $[t_m, t_{m+1}) = [t', t'')$. In this time interval $N(x,t)$ remained fixed by the definition of the t_i's and thus also (b) is satisfied.

Case 2. For every $j < 2k+3$ less than $C\, b_i/b_{i-1}$ updates occurred below $A(x,t)$ during the time interval $[t_j, t_{j+1})$.

We will show that in this case $N(x,t)$ must have stayed fixed while $\Omega(b_i)$ operations have occured below $A(x,t)$ in the time interval $[t_{2k+3}, t_{2k+4})$. To see this first notice that there are three types of modification that could be performed on $N(x,t)$ at time t_j.

1. $N(x,t)$ was reconstructed at time t_j (by a call to procedure RECONSTRUCT).

2. $B_{i,1}$ was adjusted in a locale of y_i due to the increase of some $\hat{\lambda}$ value above 0.3 (by a call to procedure MOVE).

3. $B_{i,1}$ was modified in a locale of y_i due to an active participation of one of its members in a rotation (by an execution of block REBOUND of REBALANCE).

After a reconstruction of $N(x,t)$ all its members have β and λ values of zero. Whenever a new node enters $N(x,t)$ from above, as a result of a modification of $B_{i,1}$ of type (2) or (3), its β value is 0. Furthermore, any super-boundary modification can only improve λ values (Lemmas 2 and 3). This implies that between any two modifications of $N(x,t)$ of type (1) at least one modification of one of the other types should occur. In other words, no two consecutive modifications can be of type (1).

A modification of type (2) leaves $\lambda(i,1,y_j) = 0$, and again no super-boundary modification can increase λ values. Thus, once a modification of either type (1) or (2) occurs it takes at least $\lceil \frac{1}{2}(b_i) \rceil$ updates below $N(x,t)$ until it will be modified by an update of type (2), and thus at most one modification of type (2) can take place in $[t_0, t_{2k+3}]$.

To get a handle on the effects of modifications of type (3) we introduce a conceptual process of marking in the time period $[a, t_{2k+3}]$. We mark a node x, which lies on the path P between the root of $N(x,t)$ and the root of $A(x,t)$, if

A) x has just actively participated in a rotation, or

B) x has just crossed a super boundary.

We note that whenever x is marked $\beta(x)=0$. For nodes which have just actively participated in a rotation this follows from Lemma 1 in [HL79]. For nodes which have just crossed a super-boundary it follows from the nature of modifications of super-boundaries (Lemmas 2 and 3). We further note that a modification of type (3) can be caused only by rotations which did not originate inside block REBOUND of REBALANCE (see the proof of Lemma 2), or in the procedure MOVE (Lemma 3). In other words, only a rotation which is a result of a call to REBOUND which is originated in MAINTAIN can cause a modification of type (3). Such rotations are performed only on nodes x with $\hat{\beta}(x) \geq 0.3$ and hence are unmarked. It follows that once a node x is marked no rotation which is rooted at x can cause a modification of type (3). At time t_0 there were only k unmarked nodes on the path P. Notice that every modification of type (3) is caused by a rotation which is rooted at an unmarked node $x \in P$, and thus adds at least one mark on the path P. On the other hand the number of unmarked nodes in P can not increase since any node that enters P is marked at that point in time. Thus after k modifications of type (3) all the members of P are marked. This implies that no modification of type (3) can change $N(x, t_{2k+3})$ until $\lceil \frac{1}{2}(b_i) \rceil$ updates occurred below $A(x, t_{2k+3})$.

We have established that at most k+1 modifications of types (2) and (3) could have occurred in the time interval $[t_0, t_{2k+3}]$. Since every two modifications of type (1) are separated by at least one modification of type (2) or (3) at most k+2 modifications of type (1) could have occured in $[t_0, t_{2k+3}]$. Since exactly 2k+3 modifications have occurred in this time

interval it must be that at exactly k of those modifications ere of type (3). This implies that after time t_{2k+3}, every node on the path P is marked. Moreover, we can assert that exactly k+3 of the modifications were of type (1) and one of them was of type (2). This implies that at time t_{2k+3}, N(x,t) must remain fixed throughout $\Gamma\!\!\!\!\Gamma(b_i)$ updates below A(x,t). In other words, in the time interval $[t_{2k+3}, t_{2k+4}]$ at least $\Gamma\!\!\!\!\Gamma(b_i)$ updates have occurred below A(x,t), and N(x,t) was fixed for that period of time. Thus the choice $t'=t_{2k+3}$ and $t''=t_{2k+4}$ will satisfy both (a) and (b). []

Lemma 7. For a sufficiently large b_0, $P_0(k) \wedge P_1(k) \wedge P_2(k) \Rightarrow P_3(k)$. Informally, assuming that the boundaries and the balances are well maintained, and that the rate of scanning is sufficient to guarantee that the relative errors are small enough, then all λ and β values are stable.

Proof. Assume that the relative error is smaller than the ϵ of Lemma 1 in [HL79] and is also smaller than the ϵ required in Lemma 1 of this paper. The only way for a $\hat{\lambda}$ or a $\hat{\beta}$ value to increase is due to an update of an $\hat{r}$ value. Suppose that after an update of $\hat{r}(x)$, or of $\hat{r}(y)$ where y is a child of x, at time t_0, we find that $\beta(x)$ has increased beyond 0.3. Assuming that the relative error e(x) is small enough so that $|\beta(x) - \hat{\beta}(x)| \leq 0.1$ we know that $\hat{\beta}(x)$ was ≤ 0.3 before the update, and thus $\beta(x)$ was ≤ 0.4 before the update. If b_0 is large enough so that a single update can change the value of β by at most 0.1 we are guaranteed that following the update $\beta(x) \leq 0.5$. If x is not a member of a neighborhood we immediately perform a rotation which brings $\beta(x)$ back to zero. A similar argument holds for $\lambda(i,j,x)$ values of super-boundary members x (i.e. for j = 1 or 2).

A problem may occur if x is a member of a neighborhood N of B_i at time t_0. In that case the only way to reduce $\beta(x)$ (or resp. $\lambda(i,0,x)$ for boundary members) is by reconstruction of the neighborhood. The problem is that we may not be able to to do it right away if N is not eligible for reconstruction at time t_0. We know however that

$$\left| \bigcup_{y \in N} T(y) \cap B_{i-1} \right| = O(b_i/b_{i-1})$$

and thus there exists some constant K s.t. K b_i/b_{i-1} fixups on disunified members of $\bar{N}$ will unify all of them and thus make N eligible for reconstruction. From Lemma 6, substituting K for C, we know that there is a constant D such that within at most D b_i/b_{i-1} updates below the area A of x, N

will be fixed for a period of time during which A will be involved in more than K b_i/b_{i-1} updates. Each one of those updates triggers a fixup on every disunified member of $\bar{N}$, and thus within (D+K) b_i/b_{i-1} updates below A the neighborhood N will be eligible for reconstruction. If b_0 is large enough (D+K) b_i/b_{i-1} operations can not change $\beta(x)$ by more than 0.2. Since at time t_0 we had $\beta(x) \leq 0.4$ we must still have $\beta(x) \leq 0.6$ when N is eligible for reconstruction. The argument for the $\lambda(i,0,x)$ values is similar. []

Lemma 8. For a sufficiently large b_0, $P_3(k) \Rightarrow P_0(k+1)$ Informally, if the relative error is sufficiently small, the λ and β values are stable, and the S_OUT sets do not grow over their stated limit, then a single update can not increase any λ or β value above one, or overflow an S_OUT set.

Proof. Again a single insertion or deletion can change the value of $\beta(x)$ or $\lambda(i,j,x)$ by an amount inversely proportional to the rank of x unless x actively participates in a rotation so the proof that the λ's and β's remain bounded is similar to the proof of Lemma 8 in [HL79].

We have to show that the given bound on the size of the S_OUT sets still holds. That is, disunified boundary nodes have at most two scanners in their S_OUT sets and other nodes have at most one. Since abandoning and replacing scanners do not increase the size of S_OUT sets, a single update operation can affect the above proposition concerning the i^{th} ply in only one of two ways:

a) It caused a reconstruction of some neighborhood and an adjustment of B_i which deactivated some scanners and activated some new ones.

b) It triggered some FIXUP operations which completed the unification of one or more nodes $x \in B_i$.

In case (a) all the nodes which are members of the new boundary and have just acquired new scanners were either non-boundary nodes or unified boundary nodes before the adjustment. This implies that they all had at most one scanner in their S_OUT set before the adjustment took place, and thus they all have at most two scanners in their S_OUT set after the adjustment. Since they all became disunified, the proposition still holds for them. Also all the nodes that left the boundary due to the adjustment have only one scanner in S_OUT since they were unified at the time just prior to the adjustment.

In case (b) let $x \in B_i$ be a node that just got unified due to a FIXUP(x) operation. Let t_0 denote the point in time when x last became disunified. Let s denote the active scanner which was assigned to x at time t_0, and let s' denote the scanner of x which was deactivated at time t_0. We may conclude that at least $\Theta(b_i/b_{i-1})$ update operations occurred below the area of x since t_0; the reason is that this number of FIXUP's is needed in order to unify x again. Throughout this period of time x was disunified, a fact which caused each one of the above operations to trigger a FIND or a SCAN operation on s. As in the proof of Lemma 4, it takes only $O(b_i^{1/2})$ operations for s to traverse the complete path from x to its B_{i+1} ancestor y. Since $b_i^{1/2} = o(b_i/b_{i-1})$, s will reach the top y of the orbit at least once before x is unified. Let t_1 denote the point in time when s reaches y. If s' does not get abandoned before t_1 it must be that both s and s' reach y, and thus get abandoned, together. To see this notice that s starts its way upwards from the parent of x. Every time that the set of scanners to which s belongs is advanced one step upwards from z to its parent z' we add S(z) to S(z'). Since s' resided somewhere above s on the same orbit at time t_0, it must be that s "passed" s' on the way up at some point, and that s and s' have been members of the same set of scanners ever since. It thus follows that they have reached the top of their orbit together. []

As we saw in the outline of the proof, by combining Lemmas 2-5, we have

Theorem 1. For a sufficiently large b_0 the tree is correctly maintained, i.e., $P_i(k)$ holds at all times, for i=0,1,2, and 3.

It is a simple task to verify that the time spent by a call to each one of the procedures SCAN, NORMALIZE, REBOUND, REBALANCE, and FIXUP is O(1). Consequently, the time of each individual update is determined by the main loop of MAINTAIN. Since an n vertex tree has $O(\log^* n)$ plies this proves

Theorem 2. The time complexity of each individual update (insertion or deletion) in our structure is $O(\log^* n)$ for a tree with n keys.

4.0 Space Complexity

Recall that the traditional union-find structure which is used here to represent scanners, does not have children pointers. This fact may cause the failure of the well known reference count method, for garbage collection, to keep the space required by the data structure linear in the size of the tree. To demonstrate this danger assume that a set S af scanners, with root s_0, is abandoned at time t. At this point each scanner of S could, in principle, be released. To be exact, active scanners could be made to point back to their owners, while inactive scanners could be returned to some storage pool. The problem, however, is that there is no way to traverse the set S, starting from s_0. We first give a simple bound on the space requirement of our structure assuming the reference count method is used for releasing scanners, and then we show how the space complexity may be kept linear by a slight modification of the algorithm.

Lemma 9. Using reference counts to release scanners results in the worst case in O(n log n) space complexity, for a tree of size n.

Proof sketch. Since there are at most two non-abandoned scanners per node in the tree, certainly non-abandoned scanners can not cause problems. As for abandoned scanners we may count only those scanners which have at least one active scanner below them in the union-find set. The reason is that using the reference count method we can insert scanners which do not satisfy the above condition into a queue where they are waiting to be relesed. By an accounting argument one can show that if we release a constant number of nodes per update then the size of such a queue can not grow beyond O(n).

When a set of scanners, say S, is abandoned, $|S| = O(n)$ and thus the height of the tree representing S is O(log n). Now, every active scanner may have at most O(log n) scanners above it, and since there are at most O(n) active scanners in the tree we get that O(n log n) is a bound on the number of abandoned scanners which have an active descendant in the union-find tree.

[]

To get a linear space complexity we may keep all the children of each scanner on a linked list. (This trick is useful also in obtaining a solution to the union-find-delete problem, exercise 4.18 in [AHU74]). Now whenever a set of scanners is abandoned, we can start traversing it, releasing meanwhile a constant number of abandoned nodes per operation. This argument could easily be extended to a proof of

Theorem 3. The structure could be modified to have linear space complexity.

Acknowledgements

I would like to thank my advisor George Lueker for his encouragement, useful suggestions, and the numerous enjoyable disscussions we had considering this manuscript, which greatly improved the quality of presentation. I am especially grateful to him for carefully reading and pointing out an error in a previous version of this paper. I also wish to thank Scott Huddleston for some enjoyable discussions we had on the topic of this manuscript.

References

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

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

[BT78] 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 Science Department, Stanford University, December 1978.

[GMPR77] 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.

[H79] Huddleston, S., private comunication.

[HL79] Harel, D., and Lueker, G., "A Data Structure with Movable Fingers and Deletions," Technical Report #145, Dep. of ICS, University of California at Irvine, December 1979.

[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, Dept. of ICS, University of California at Irvine, February 1979.

[MS79] Maier, D., and Salveter, S., "Hysterical B-Trees," Technical Report #79/007, Dept. of Computer Science, SUNY at Stony Brook, November 1979.

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

[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 A: A List of Fields

In addition to keys, which are kept only at the leaves, for each node x in P_i we keep:

1. LEFT(x) - The left child of x in T.
2. RIGHT(x) - The right child of x in T.
3. L_THREAD(x) - If x has a left child then the leftmost leaf descending from x else the predecessor of x in inorder.
4. R_THREAD(x) - If x has a right child then the rightmost leaf descending from x else the successor of x in inorder.
5. PARENT(x) - The parent of x in T.
6. $\hat{r}(x)$ - The stored rank of x.
7. INDEX(x) - The index of the ply to which x belongs.
8. J(x) - An integer field. For members of boundaries (super-boundaries) J(x) = j iff $x \in B_{i,j}$, for nonmembers J(x) = -1.
9. S(x) - The set of scanners pointing to x.
10. LEFTMOST_BD(x) (resp. RIGHTMOST_BD(x)) - a pointer to a linking record r s.t. LINK(r) is the leftmost (resp. rightmost) descendant y of x in B_{i-1}. (Thus r = D_RECORD(y).)

In addition to the above, we keep for each boundary node $y \in B_i$ the following:

11. SCANNER(y) - The scanner belonging to y.
12. D_RECORD(y) - The descendancy record of y. A pointer to some linking record d such that LINK(d) = y. If y is a left child (resp. a right child) then for every x in P_i such that y is the leftmost (resp. rightmost) descendant of x in B_i, LEFTMOST_BD(x) = y (resp. RIGHTMOST_BD(x) = y).
13. A_RECORD(y) - The ancestry record of y. A pointer to some linking record a such that LINK(a) = y. If y is unified then every descendant z of y in B_{i-1} has B_ANCESTOR(z) = A_RECORD(y).

14. B_ANCESTOR(y) - Apointer to some linking record r. Let z be the ancestor of y in B_{i+1}. If z is unified then r is the ancestry record of z (i.e. A_RECORD(z) = r and LINK(r) = z). Otherwise r is the ancestry record of some z' which lies within a distance of O(1) from z.

15. B_LEFT(y) - y's left neighbor on B_i.

16. B_RIGHT(y) - y's right neighbor on B_i.

17. AT(y) - If y is disunified then a pointer to some B_{i-1} descendant of y. In this case AT(y) gives the point at which we currently are in the traversal of $T(y) \cap B_{i-1}$. Otherwise AT(y) = null.

18. FLAG(y) - A boolean field. FLAG(y) = true iff y is unified.

Notice that we could use the convention that INDEX(x) is kept only for nodes which are not members of any boundary (or super-boundary) setting INDEX(x) = -1 for boundary (super-boundary) members. This eliminates the need for the boolean field B(x). Also notice that FLAG(y) is redundant since y is unified iff AT(y) = null. In fact, even the PARENT(x) field is not really necessary since, in the terminology of [HL79], either the lowest right ancestor or the lowest left ancestor of x is the parent of x, and both are computable in O(1) time using only threads and child pointers. Another curiosity, which further indicates the power of threads, is that using B_ANCESTOR fields, THREADS, and RIGHTMOST_BD, and LEFTMOST_BD fields, it is possible to compute B_LEFT(y) and B_RIGHT(y) in O(1) time and so those fields are also redundant.

The fields S(x), LEFTMOST_BD(x), RIGHTMOST_BD(x), D_RECORD(y), A_RECORD(y), and AT(y), as well as the scanner cells, and the linking records are needed for the per-operation version but not for achieving the global time bound [HL79].

Appendix B: The Algorithm

```
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 pointers to linking records let u_0,...u_g
  be the ancestors of u which lie on boundaries;
  for i := 0 until g do
  begin
    for j := 1,2 do v_j := the ancestor of u_i in B_{i,j};
    A := A(v_2);
    comment A is the area rooted at v_2;
    DA := the disunified boundary members of A;
    for every w ∈ DA ∪ {u_i} do
    begin
      if w ∈ DA then FIXUP(w);
      s := FIND(w);
      if FATHER(s) is either a node of T or null then
      begin
        x := SCAN(w,s);
        r̂(x) := r̂(LEFT(x)) + r̂(RIGHT(x));
        if x belongs to some neighborhood N then
        begin
          v := the root of N;  N̄ := the boundary of N;
          if ((x ∈ N̄ ∧ λ(i,0,x) ≥ 0.3) or (x ∉ N̄ ∧ β(x) ≥ 0.3))
             and v is eligible then RECONSTRUCT(v,i);
        end else if β̂(x) ≥ 0.3 then REBALANCE(x,i);
      end;
      for j := 1,2 do if λ̂(i,j,v_j) ≥ 0.3 then MOVE(i,j,v_j);
    end;
  end;
end;
```

```
procedure FIND(w,i);
begin
    s_0 := SCANNER(w);
    s_1 := if FATHER(s_0) is a scanner then FATHER(s_0) else s_0;
    s_2 := if FATHER(s_1) is a scanner then FATHER(s_1) else s_1;
    if (s_0 ≠ s_1) and (s_1 ≠ s_2) then FATHER(s_0) := s_2;
    return s_2;
end;


procedure SCAN(u,s);
begin comment assuming u ∈ B_i and that s is the root of
    the UNION_FIND structure representing the set of scanners to which
    SCANNER(u) belongs, returns the next node x on u's orbit after
    performing the necessary housekeeping operations for the scanner sets
    s and S(x);
    i := INDEX(u);  y := if s ≠ null then FATHER(s) else null;
    if (y = null) or (y ∉ B_{i+1}) then
    begin comment return to the bottom of the orbit;
        FATHER(s) := null;
        if y ≠ null then S(y) := null;
        s' := a new scanner;
        p := PARENT(u);
        x := FATHER(s') := p;
        S(p) := SCANNER(u) := s';
    end else
    begin comment go up one step;
        x := PARENT(y);
        S(x) := S(x) ∪ S(y);
        S(y) := null;
    end;
    return x;
end;
```

```
procedure RECONSTRUCT(u,i);
begin B := B_i ∩ T(u);  C := (P_i ∩ T(u)) - B;
  perform rotations on descendants of u until for every x below u
    if r(x)/b_i - c_0 ≥ -e then β(x) = 0;
  B' := a tree cut below u with ∀x∈B' λ(i,0,x)=0;
  replace the portion of B_i below u by B';  C' := (P_i ∩ T(u)) - B';
  D := the highest cut for T(u) s.t. all the nodes examined
       so far lie above D;  A := B ∪ C ∪ B' ∪ C' ∪ D;
  for every x ∈ C do S(u) := S(u) ∪ S(x);
  for every x ∈ A - C do if S(x) ≠ null then FATHER(S(x)) := null;
  for every x ∈ A - {u} do S(x) := null;  let b0 and b1 (resp. b0' and b1')
    be the leftmost and rightmost elements of B (resp. B')
  for every x ∈ B' do begin
      create two new linking records a and d, and a new scanner s;
      D_RECORD(x) := if x = b0' then D_RECORD(b0) else
                     if x = b1' then D_RECORD(b1) else d;
      p := PARENT(x);
      A_RECORD(x) := a;  S(p) := SCANNER(x) := s;
      LINK(A_RECORD(x)) := LINK(D_RECORD(x)) := x;   FATHER(s) := p;
      UPDATE_B_LIST:  using B_LEFT and B_RIGHT fields link B'
        in the left to right order;  using B_LEFT(b0) and B_RIGHT(b1)
        replace the portion B of the B-list by the portion C;
      B_ANCESTOR(x) := the A_RECORD of the B_{i+1} ancestor of x;
      FLAG(x) := false;  AT(x) := the leftmost B_{i-1} descendant of x;
  end;
  for every x ∈ B do
    LINK(A_RECORD(x)) := some y ∈ B' from which x inherited ancestors;
  for each x∈A-{u} update r̂(x) from its descendants in D;
  for x∈C' (resp. A-C') do
  begin let l and r be its leftmost and rightmost
      descendants in B' (resp. D);
      RIGHTMOST_BD(x) := if x∈C' then D_RECORD(r) else RIGHTMOST_BD(r);
      LEFTMOST_BD(x) := if x∈C' then D_RECORD(l) else LEFTMOST_BD(l);
  end;
end;
```

```
procedure REBALANCE(u,i);
begin comment perform the appropriate rotation to bring β(u) back to 0.  The
  procedure REBALANCE(u,i) is not called on neighborhood nodes u;
    perform the appropriate rotation RN to bring β(x) back to 0;
    A := the set of active participants of RN;
    B := the set of nodes which did not actively participate in RN
         but whoes parent did;
    for every x ∈ A do
    begin
        S(u) := S(u) ⋓ S(x);
        S(x) := null;
        update LEFTMOST_BD(x) and RIGHTMOST_BD(x) from its descendants in B;
        update r̂(x) from its descendants in B;
    end;
    if some member of B_{i,j} actively participated in RN then
    REBOUND:  begin
        G := the highest cut of T(u) which runs below both B and B_{i,j};
             comment in Figure 1 G is denoted by a dotted line;
        H := a cut of T(u) s.t. ∀x ∈ H with parent p and children y and z
        λ(i,j,x) ≤ min{ λ(i,j,p) , λ(i,j,y) + λ(i,j,z) };
        for every y between H and B_{i,j} with β(y) > 0 do REBALANCE(y);
        replace the portion of B_{i,j} below u by H;
    end;
end;


procedure FIXUP(u);
begin
    x := AT(u);
    x := AT(u) := B_RIGHT(x);
    if u is an ancestor of x then B_ANCESTOR(x) := A_RECORD(u)
                             else FLAG(u) := true;
end;
```

```
procedure MOVE(i,j,u);
begin comment assuming that u ∈ B_{i,j} and
  λ̂(i,j,u) ≥ 0.3 move B_{i,j} up or down in a locale of u;
    if r(x)/b_i - c_j > e then
    begin comment move B_{i,j} downwards;
        for every x below u with r̂(x)/b_i - c_j ≥ -e
          and β(x) > 0 do REBALANCE(x,i);
        H := a tree-cut of T(u) s.t. ∀ x ∈ H λ(i,j,x)=0;
        replace the portion of B_{i,j} below u by H;
    end else
    begin comment move B_{i,j} upwards;
        v := some ancestor of u with λ(i,j,u)=0;
        for every x between v and B_{i,j} with β(x) > 0 do REBALANCE(x,i);
        replace the portion of B_{i,j} below v by {v};
    end;
end;
```