THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 62, No. 10, December 1983
Printed in U.S.A.

Two New Kinds of Biased Search Trees

By J. FEIGENBAUM* and R. E. TARJAN†

(Manuscript received May 20, 1983)

In this paper, we introduce two new kinds of biased search trees: biased, a, b trees and pseudo-weight-balanced trees. A biased search tree is a data structure for storing a sorted set in which the access time for an item depends on its estimated access frequency in such a way that the average access time is small. Bent, Sleator, and Tarjan were the first to describe classes of biased search trees that are easy to update; such trees have applications not only in efficient table storage but also in various network optimization algorithms. Our biased a, b trees generalize the biased 2, b trees of Bent, Sleator, and Tarjan. They provide a biased generalization of B-trees and are suitable for use in paged external memory, whereas previous kinds of biased trees are suitable for internal memory. Our pseudo-weight-balanced trees are a biased version of weight-balanced trees much simpler than Bent's version. Weight balance is the natural kind of balance to use in designing biased trees; pseudo-weight-balanced trees are especially easy to implement and analyze.

I. INTRODUCTION

The following problem, which we shall call the *dictionary problem*, occurs frequently in computer science. Given a totally ordered universe U, we wish to maintain one or more subsets of U under the following operations, where R and S denote any subsets of U and i denotes any item in U:

access (i, S)—If item i is in S, return a pointer to its location. Otherwise, return a special **null** pointer.

* Research done partly while a summer employee of Bell Laboratories and partly while a graduate student supported by Air Force grant AFOSR-80-042.

† Bell Laboratories.

insert (i, S)—Insert i in S, assuming it is not previously there.

delete (i, S)—Delete i from S.

join (R, S) (two-way join)—Return the set consisting of the union of R and S, assuming that every item in R precedes every item in S. This operation destroys R and S, and can be regarded as a concatenation of R and S.

split (i, S)—Split S into three sets L, I, and R, where L and R are the sets of items strictly smaller and strictly larger than i, respectively, and $I = \{i\}$ if i is in S (three-way split), $I = \emptyset$ if i is not in S (two-way split).

One way to solve the dictionary problem is to store the items of each set in the external nodes of a search tree in left-to-right order, one item per external node. To guide the operations, the search tree also contains auxiliary items, called *keys*, in the internal nodes. The worst-case access time in a search tree is proportional to the depth of the tree. By imposing any one of a number of well-known *balance conditions* on the tree, we can guarantee that its depth is $O(\log n)$, where n is the number of items it contains. Such a balance condition can be maintained during update operations by performing appropriate local rearrangements of the tree. With balanced search trees, each of the dictionary operations has an $O(\log n)$ time bound. Examples of balanced trees include height-balanced (AVL) trees,[1] 2, 3 trees,[2] B-trees,[3] and trees of bounded balance.[4]

In many applications of search trees the access frequencies of different items are different, and we would like our data structure to take this into account. To deal with this problem formally we assume that each item i has a known weight w_i providing an estimate of the access frequency. The *biased dictionary problem* is that of implementing the dictionary operations so that operations on heavier items are faster than those on lighter items. In particular, when representing a set S as a search tree, we wish to bias the tree so as to approximately minimize its total weighted depth $\sum_{i \in S} w_i d_i$, where d_i is the depth of the external node containing item i, while preserving the ability to update the tree rapidly. In addition to the five dictionary operations, we allow the following operation for changing the weight of an item:

reweight (i, w, S)—Redefine the weight of item i in set S to be w.

The following theorem, due to Shannon, gives a lower bound on the total weighted depth of a search tree:

Theorem 1:[5] *If T is a search tree for a set S and every node of T has no more than b children, then*

$$\sum_{i \in S} w_i d_i \geq \sum_{i \in S} \frac{w_i}{W} \log_b \frac{W}{w_i},$$

where $W = \sum_{i \in S} w_i$ is the total weight of the items in S.

In light of Theorem 1, our goal is to devise classes of search trees that are easy to update and have $d_i = O(\log_b W/w_i)$ for every item i. We call $O(\log_b W/w_i)$ the ideal access time of item i.

Bent, Sleator, and Tarjan[6-8] have devised several kinds of such biased search trees. Our work is an extension of theirs. A thorough discussion of previous work by others on the biased dictionary problem may be found in Ref. 8.

In their running time analyses, Bent, Sleator, and Tarjan used a technique called *amortization*, which we shall also use. The idea of amortization is to average the running time of individual operations over a (worst-case) sequence of operations. As a tool in deriving amortized time bounds we use *credits*. A credit will pay for one unit of computing time. To each operation we allocate a certain number of credits, called the *credit time* or *amortized time* of the operation. If a given operation does not need all its credits, we can save them for use in later operations; if an operation needs more than its share of credits we can use those previously saved. In any analysis using credits, the objective is to prove that an arbitrary sequence of operations can be performed without running out of credits.

Three points about amortization using credits are worth making. First, credits are a way of charging earlier operations for later ones. If a credit analysis is successful, we can assert that any sequence of operations requires an amount of actual time that is at most a constant multiplied by the sum of the credit times of the individual operations; slow operations are only possible if there are corresponding earlier fast ones. Second, although the word "average" appears in the description of the technique, it is not the usual kind of average-case analysis, and in particular we make no probabilistic assumptions; we obtain worst-case bounds holding for *any* sequence of operations. Third, credits serve as a kind of "potential energy": we place them in regions of search trees that may cause abnormally long update operations. This idea may illuminate the credit invariants we define below.

The remainder of the paper consists of three sections. In Sections II and III, we define and analyze biased a, b trees, which generalize the biased 2, b trees of Ref. 8 and provide a biased version of B-trees. Biased a, b trees are a form of biased tree appropriate for paged external memory; earlier forms of biased trees are more appropriate for internal memory. In Section IV we introduce pseudo-weight-balanced trees, which give a biased version of weight-balanced trees much simpler than Bent's earlier version.[6] Weight balance is the natural kind of balance to use in designing a biased search tree; pseudo-weight-balanced trees are especially easy to implement and analyze.

We shall assume that the reader is familiar with search trees. In particular we shall not discuss the use and updating of keys, and we

shall draw freely on the results and techniques of Ref. 8. We shall use the terminology of Ref. 8 except that we use "external node" in place of "leaf". When appropriate we shall regard a node x of a search tree as denoting the entire subtree rooted at x, with the context resolving whether a node or a tree is meant. The **null** node denotes the empty search tree. We denote the parent of a node x by $p(x)$; $p(x) =$ **null** if x is a tree root.

II. LOCALLY BIASED *a, b* TREES

Our first class of biased trees uses height balance to guarantee fast access and variable node size to allow easy updating. The class is parameterized by two positive integer constants a and b such that $2 \leq a \leq \lceil b/2 \rceil$. The integer b specifies the maximum allowed number of children of an internal node. If an internal node has at least a children, we say it is *filled*; otherwise it is *underfilled*. (An external node is always filled.) Ideally, we would like every node to have at least a children; since in our scheme this is impossible to achieve, we allow underfilled nodes but treat them specially.

If x is a node in a search tree, we define its *weight* $w(x)$ to be the sum of the weights of all items in descendants of x. (We use the convention that every node is a descendant of itself.) We define the *rank* $s(x)$ of x recursively by $s(x) = \lfloor \log_a w(x) \rfloor$ if x is an external node, $s(x) = 1 + \max\{s(y) | p(y) = x\}$ if x is an internal node. A node x is *minor* if x is not a root and either $s(x) < s(p(x)) - 1$ or x is underfilled. All other nodes are *major*. A *locally biased a, b tree* is a search tree with the following properties (see Fig. 1):

1. Every internal node has at least two and at most b children;

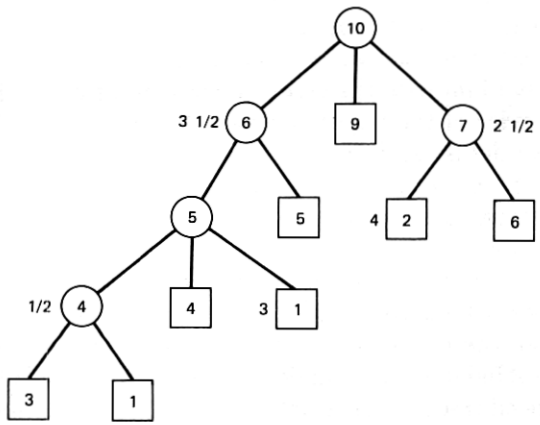

Fig. 1—A biased 3, b tree. The numbers in nodes are ranks, and the numbers beside nodes are credits for the credit invariant.

2. If x is a minor node, any adjacent sibling of x is both major and external. When a node x has this property, we say the tree is *locally biased* at x.

We call two nodes r-compatible if they can be adjacent children of a node of rank r in a biased a, b tree. That is, two nodes are r-compatible if both have rank at most $r-1$ and either at least one has rank $r-1$ and is external or both have rank $r-1$ and at least a children each.

If $a = 2$ we obtain exactly the biased 2, b trees of Bent, Sleator, and Tarjan. Our main new idea is in the definition and handling of underfilled nodes. Our first theorem guarantees that a, b trees have ideal access time if b is chosen appropriately.

Lemma 1: *Consider any biased a, b tree. If x is an external node, $a^{s(x)} \le w(x) < a^{s(x)+1}$. If x is an internal node with at least one minor child, $a^{s(x)-1} \le w(x)$. If x is an internal node with k children, $k'a^{s(x)-2} \le w(x)$, where $k' = m\{k, a\}$.*

Proof: The first part of the lemma is immediate from the definition of rank. The second part follows from the first part and property 2: if x has a minor child, it must have another child that is major and external. We prove the third part by induction on the height of x. If x has a minor child, then $k'a^{s(x)-2} \le a^{s(x)-1} \le w(x)$ by the second part of the lemma. Otherwise, all children of x are major. Let y be a child of x. If y is external, or internal with a minor child, then $a^{s(x)-2} = a^{s(y)-1} \le w(y)$. Otherwise, y has at least a major children, and by the induction hypothesis $a^{s(x)-2} = a \cdot a^{s(y)-2} \le w(y)$. Summing over the k children of x gives $ka^{s(x)-2} \le w(x)$. □

Lemma 2: *If x is an external node in a biased a, b tree of total weight W, the depth of x is at most* $\log_a W/(w(x)) + 3$.

Proof: Let r be the root of the tree and d the depth of x. Since the rank increases by at least one from child to parent, $d \le s(r) - s(x)$. Lemma 1 implies $\log_a w(x) \le s(x) + 1$ and $\log_a W = \log_a w(r) \ge s(r) - 2$. Combining inequalities gives the lemma. □

Theorem 2: *A biased a, b tree has ideal access time, with a constant factor proportional to* $\log_a b$.

Proof: Immediate from Lemma 2. □

According to Theorem 2, to minimize the access time in a, b trees, we should choose b as small as possible. The requirement $b \ge 2a - 1$ is necessary to allow efficient updating. The best choice of b seems to be $2a - 1$ or $2a$. The choice $b = 2a$ allows purely up-down updates (see the end of Section III). The choice $b = 2a - 1$ gives a biased version of ordinary B-trees.[3] Any other choice gives a biased version of "hysterical" or "weak" B-trees.[9-11] Note also that Theorem 2 gives a worst-case example, not an amortized bound on the access time.

As Ref. 8 shows, all the update operations on search trees can be carried out using one or more joins. Our next task is thus to define a join algorithm for biased a, b trees.

Algorithm 1: *local join* (x, y). Join two locally biased a, b trees with roots x and y, assuming that all items in tree x precede all items in tree y.

Case 0—$x =$ **null** or $y =$ **null.** If $x =$ **null,** return y; if $y =$ **null,** return x.

Case 1—$s(x) \geq s(y)$ and x and y are $(s(x) + 1)$-compatible, or $s(x) \leq s(y)$ and x and y are $(s(y) + 1)$-compatible. Create a new node with x and y as its children and return the new node.

Case 2—$s(x) = s(y)$ and x and y are not $(s(x) + 1)$ compatible. Let u be the rightmost child of x and v the leftmost child of y (see Fig. 2a). Perform *join*(u, v), letting w be the root of the resulting tree. If $s(w) < s(x)$, construct a node z whose children are those of x not including u, the node w, and the children of y not including v. If $s(w) = s(x)$, construct a node z whose children are those of x not including u, those of w, and those of y not including v (see Fig. 2b). In either case, if z has more than b children, split it into two nodes z' and z'' with z as parent, dividing the old children of z as evenly as possible between z' and z'' (see Fig. 2c). Return z.

Case 3—$s(x) > s(y)$ and x and y are not $(s(x) + 1)$-compatible. Let u be the rightmost child of x (see Fig. 2d). Perform *join* (u, y), letting v be the root of the resulting tree. If $s(v) < s(x)$, replace u as a child of x by v. If $s(v) = s(x)$, replace u as a child of x by the set of children of v. If x now has more than b children, split it into two nodes x' and x'' with x as parent, dividing the old children of x as evenly as possible between x' and x''. Return x.

Case 4—$s(x) < s(y)$ and x and y are $(s(y) + 1)$-compatible. Symmetric to Case 3.

Theorem 3: *Given two biased a, b trees with roots x and y, the join algorithm produces a biased a, b tree whose root has rank* $\max\{s(x), s(y)\}$ *or* $\max\{s(x), s(y)\} + 1$. *In the latter case, the root of the new tree has exactly two children.*

Proof: An easy induction on rank shows that the root of the tree produced by the join has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\} + 1$ and that in the latter case the root has exactly two children. Furthermore, it is clear that every internal node in the new tree has at least two and at most b children. All that remains is to show that any two adjacent siblings in the new tree are allowed to be adjacent by property 2. We prove this by induction on rank, using the same cases as in the algorithm.

Case 1—Assume without loss of generality that $s(x) \geq s(y)$. Since

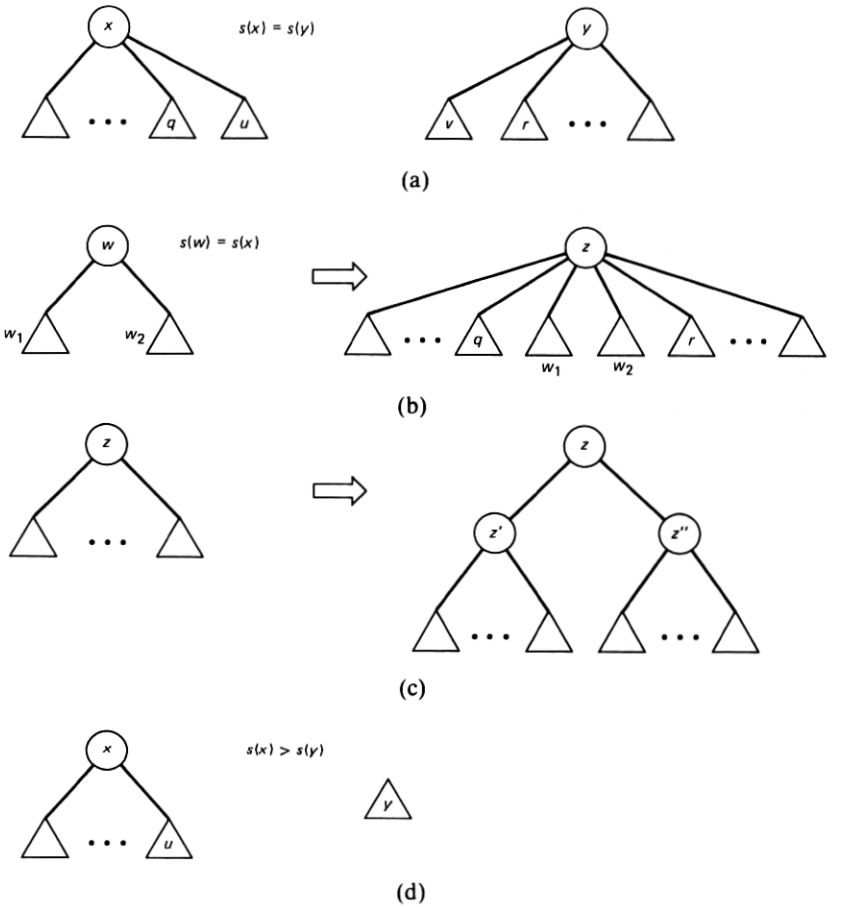

Fig. 2—Cases of the join algorithm. (a) Situation at the beginning of Case 2. (b) Recursive call *join* (u, v) produces a tree of rank $s(x)$ with root w. (c) Division of an overfilled root. (d) Case 3.

x and y are $(s(x) + 1)$-compatible, the new tree is locally biased at x and y.

Case 2—For the moment, ignore the split of z if it takes place. Since x and y are not $(s(x) + 1)$-compatible, neither x nor y is external, and u and v exist. Suppose the left sibling of u, say q, is minor. Then u is major and external, which means that the join of u and v will immediately terminate in Case 1, and the new right sibling of q will be u. The symmetric statement holds for the right sibling of v. Finally, suppose w is minor, i.e., $s(w) < s(x) - 1$ or w has fewer than a children. Then both u and v must be minor as children of x and y, respectively, and both adjacent siblings of w in the new tree will be major and external. Thus the tree before the split has property 2.

Splitting z preserves property 2 since both new children of z will have at least a children of their own (this is where we use $b > 2a - 1$) and each will have rank $s(x)$. (Property 2 implies that before the split, of any two adjacent children of z, at least one has rank $s(z) - 1$).

Case 3—Similar to but simpler than Case 2. Case 4 is symmetric. □

To make our timing analysis as similar as possible to the one in Ref. 8 for biased 2, b trees, we shall assume that credits can be divided in half, and that half a credit will pay for the work in one call of *join*, not counting the work in the recursive call. We use the following credit invariant: A nonroot node x holds $s(p(x)) - s(x) - 1$ credits, plus an additional half credit if x is underfilled.

Theorem 4: The join algorithm runs in $O(|s(x) - s(y)|)$ amortized time. Specifically, carrying out the join while preserving the invariant takes $|s(x) - s(y)| + 1$ credits in Case 1 or 2, $|s(x) - s(y)| + 1/2$ credits in Case 3 or 4.

Proof: We use induction on rank and a surprisingly complicated case analysis.

Case 1—Assume without loss of generality that $s(x) \geq s(y)$ and that if $s(x) = s(y)$, then x is external or filled. We need half a credit for the work in the join and at most $s(x) - s(y) + 1/2$ credits to place on y, for a total of $s(x) - s(y) + 1$.

Case 2—The split of z, if it occurs, does not affect the credit invariant; therefore we ignore it. Assume without loss of generality that $s(u) \geq s(v)$. There are three subcases:

Subcase 2a—$s(w) = s(x)$ and *join* (u, v) is a Case 1 join. The credits originally on u and v suffice to maintain the credit invariant on u and v after the join of x and y is completed. We have one credit on hand to join x and y; we spend half for the work in the outer call and half for the work in the inner call.

Subcase 2b—$s(w) = s(x)$ and *join* (u, v) is not a Case 1 join. To perform the join of x and y we are given one credit and can obtain at least $2s(x) - s(u) - s(v) - 2$ from u and v, for a total of $2s(x) - s(u) - s(v) - 1$. We need half a credit for the work in the outermost call and at most $s(u) - s(v) + 1$ for the recursive call, for a total of at most $s(u) - s(v) + 3/2$. The difference between what we have and what we need is $2(s(x) - s(u)) - 5/2$. Since $s(x) - s(u) \geq 1$, we must find at most an additional half credit to spend.

If *join* (u, v) is a Case 2 join, and $s(x) - s(u) = 1$, then either u or v is underfilled and yields an extra half credit. If *join* (u, v) is a Case 3 join, we save half a credit on the join. Thus, in any case we can obtain the needed half credit.

Subcase 2c—$s(w) < s(x)$. As in Subcase 2b we have at least $2s(x) - s(u) - s(v) - 1$ credits to perform the join of x and y. We need

one half for the work in the outermost call, at most $s(u) - s(v) + 1$ for the recursive call, and at most $s(x) - s(w) - 1/2$ to place on w, for a total of at most $s(x) + s(u) - s(v) - s(w) + 1$. The difference between what we have and what we need is $s(x) - s(u) + s(w) - s(u) - 2$. Since $s(x) - s(u) \geq 1$ and $s(w) \geq s(u)$, we have enough credits unless $s(x) - s(u) = 1$ and $s(w) = s(u)$, in which case we must find an extra credit.

Suppose $s(x) - s(u) = 1$ and $s(w) = s(u)$. Since $s(w) = s(u)$, *join* (u, v) is not a Case 1 join, and u is internal. If *join* (u, v) is a Case 2 join, then both u and v are internal. Either both u and v are underfilled, giving us two additional half credits, or one of u and v is underfilled and w is filled, giving us an extra half credit from u or v and saving a half credit that does not need to go on w. The only other possibility is that *join* (u, v) is a Case 3 join, which saves us half a credit.

Furthermore in this case either u is underfilled or w is filled, either giving us an extra half credit from u or saving us a half credit on w. Thus in all cases we obtain the necessary extra credit.

Case 3—We ignore the possible split of x, which does not affect the credit invariant. There are two subcases:

Subcase 3a—$s(v) = s(x)$. To perform the join we are given $s(x) - s(y) + 1/2$ credits and can obtain at least $s(x) - s(u) - 1$ more from u, for a total of $2s(x) - s(u) - s(y) - 1/2$. We need one half for the outermost call and at most $|s(u) - s(y)| + 1$ for the recursive call, for a total of $|s(u) - s(y)| + 3/2$. The difference between what we need and what we have is $2(s(x) - \max\{s(u), s(y)\}) - 2 \geq 0$.

Subcase 3b—$s(v) < s(x)$. To perform the join we have at least $2s(x) - s(u) - s(y) - 1/2$ credits. We need $|s(u) - s(y)| + 3/2$ for the outermost and recursive calls, plus at most $s(x) - s(v) - 1/2$ to place on v, for a total of $s(x) + |s(u) - s(y)| - s(v) + 1$. The difference between what we have and what we need is $s(x) - \max\{s(u), s(y)\} + s(v) - \max\{s(u), s(y)\} - 3/2$. We have enough credits unless $s(x) - \max\{s(u), s(y)\} = 1$ and $s(v) = \max\{s(u), s(y)\}$, in which case we need an extra half credit.

Suppose $s(x) - \max\{s(u), s(y)\} = 1$ and $s(v) = \max\{s(u), s(y)\}$. Then *join* (u, y) is not a Case 1 join. If it is a Case 2 join, then either u is underfilled, giving us an extra half credit, or v is filled, saving us a half credit on v. If it is a Case 3 join, we save half a credit on the join. Thus in all cases we obtain the necessary half credit.

Case 4—Symmetric to Case 3. □

It is useful to restate Theorem 4 as follows. We say a biased a, b tree with root x is *cast to rank* k if it satisfies the credit invariant and has $k - s(x)$ credits on its root. Theorem 4 implies that if x and y are

the roots of two biased a, b trees cast to a rank $k > \max\{s(x), s(y)\}$, then they can be joined, without using additional credits, to produce a tree cast to rank k.

We can implement a split as a sequence of joins, exactly as described in Ref. 8. The following algorithm splits at an item i in the tree:

Algorithm 2: split (i, r). Split the biased a, b tree with root r at item i, assumed to be in the tree.

Locate the node x containing item i. Initialize the current node to be $p(x)$ and the previous node to be x. Initialize the left and right trees to empty; they will contain the items smaller than and larger than i, respectively. Repeat the following step until the current node is **null** (the previous node is the root of the tree):

Split Step—Delete every child of the current node to the left of the previous node. If there is one such child, join it to the left tree; if there are two or more such children, give them a common parent and join the resulting tree to the left tree. Repeat this process with the children to the right of the previous node, joining the resulting tree to the right tree. Remove the previous node as a child of the current node and destroy it if it is not x. Make the current node the new previous node and its parent the new current node.

Theorem 5: The split algorithm is correct and takes $O(s(r) - s(x))$ *credit time, where* x *is the node containing item* i.

Proof: Correctness follows immediately from the correctness of the join algorithm. The time bound follows as in Ref. 8; we shall sketch the idea. Let *cur*, *prev*, *left*, and *right* be the current node, the previous node, and the roots of the left and right trees, respectively. An induction shows that $s(prev) \geq \max\{s(left), s(right)\}$ just before each split step. Another induction shows that by allocating $O(s(cur) - s(prev))$ credits to a split step, we can carry out the step while preserving the invariant that the left and right trees are cast to a rank of $s(prev) + 1$. That is, the amortized time associated with a single step of the split is proportional to the rank difference of two consecutive nodes along the split path. If we sum over all split steps, then the sum telescopes, and we obtain the time bound in the statement of the theorem. □

Splitting at an item not in the tree is just like splitting at an item in the tree, except that the initialization is different. Let r be the root of the tree, i the item at which the split is to take place, i^- and i^+ the largest item in the tree less than i and the smallest item in the tree greater than i, respectively, and x the *handle* of i, defined to be the nearest common ancestor of the external nodes containing i^- and i^+. We can locate x by searching down the path from r to x if appropriate keys are stored in the tree (see Ref. 8). To carry out the split, we combine all children of x whose descendants contain items less than i

to form the initial left tree and all other children of i to form the initial right tree. Then we initialize the previous and current nodes to be x and $p(x)$, respectively, and repeat the split step until the current node is **null.** Such a split also runs in $O(s(r) - s(x))$ credit time.

We can implement insert, delete, and reweight as combinations of splits and joins: an insertion is a two-way split followed by two joins, a deletion is a three-way split followed by one join, and a weight change is a three-way split followed by two joins. Using the same kind of analysis as in Ref. 8, we obtain the following credit times for these three operations (we have stated the bounds in terms of weights rather than ranks):

$$\text{insert } (i, x)\text{: } O\left(\log_a\left(\frac{w(x) + w_i}{\min\{w_{i^-} + w_{i^+}, w_i\}}\right)\right),$$

where i^- and i^+ are as defined above.

$$\text{delete } (i, x)\text{: } O\left(\log_a\left(\frac{w(x)}{w_i}\right)\right).$$

$$\text{reweight } (i, w, x)\text{: } O\left(\log_a\left(\frac{\max\{w(x), w\}}{\min\{w_i, w\}}\right)\right),$$

where $w(x)$ and w_i are as defined before the weight change.

Remark: In all the time bounds derived in this section and the next the constant factor is proportional to b. □

III. GLOBALLY BIASED *a, b* TREES

Local bias is sufficient to guarantee good amortized but not good worst-case running times for the dictionary operations. If it is important that every single operation be fast, we need a stronger balance condition. Figure 3 shows how a split can take more actual time than

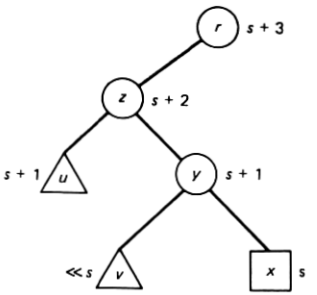

Fig. 3—Locally biased a, b tree that cannot be split at x in actual time $O(s(r) - s(x))$. Not all children of y, z, and v are shown. Join of u and v can take an unbounded amount of time.

its credit time, and illustrates why local bias is not enough: a minor node that is the leftmost child of its parent has a constraint on its right but not on its left side, and symmetrically for a minor node that is the rightmost child of its parent. To overcome this problem we introduce globally biased a, b trees.

If x is a node in a search tree, we define x^+ to be the external node containing the smallest item greater than the largest item in a descendant of x. Symmetrically, x^- is the external node containing the largest item less than the smallest item in a descendant of x. If x is on the rightmost path of the tree, x^+ is undefined; if x is on the leftmost path, x^- is undefined. A *globally biased a, b tree* is a search tree with two properties (see Fig. 4):

1. Every internal node has at least two and at most b children.

2. If x is a minor nonroot node and x^+ is defined, $s(x^+) \geq s(p(x)) - 1$; if x^- is defined, $s(x^-) \geq s(p(x)) - 1$. When a node x has this property, we say the tree is *globally biased* at x.

Global bias implies local bias. Thus globally biased a, b trees have ideal access time. We can join two globally biased a, b trees using almost the same join algorithm as in Section II; the only difference is that the conditions determininng the cases are different. We shall call the algorithm in Section II *local join* to distinguish it from the following *global join* algorithm:

Algorithm 3: global join (x, y). Join two globally biased a, b trees with roots x and y, assuming that all items in tree x precede all items in tree y.

In each case, proceed as in the corresponding case of the local join algorithm:

Case 0—$x =$ **null** or $y =$ **null.**

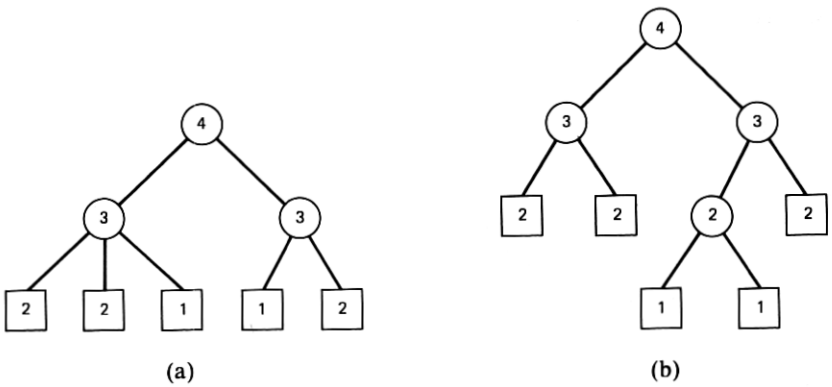

Fig. 4—Two biased 2, 3 trees with the same external nodes. Numbers in nodes are ranks. (a) Locally biased tree. (b) Globally biased tree.

Case 1—$s(x) \geq s(y)$ and x is external, or $s(x) \leq s(y)$ and y is external.

Case 2—$s(x) = s(y)$ and both x and y are internal.

Case 3—$s(x) > s(y)$ and x is internal.

Case 4—$s(x) < s(y)$ and y is internal.

The only difference between this and the local join algorithm is that if x and y have the same rank and both are internal with at least a children, we apply Case 2 instead of Case 1. The algorithm is identical to the join algorithm for globally biased 2, b trees given in Ref. 8.

Theorem 6: The global join algorithm is correct.

Proof: As in the proof of Theorem 3, we use induction on rank and a case analysis.

Case 1—Immediate.

Case 2—Ignore the split of z if it takes place. Let q be the left sibling of u (recall that u is the rightmost child of x). Suppose q or one of the nodes on the rightmost path descending from q is minor. Let this minor node be r. The join changes neither $p(r)$ nor r^+ and thus preserves global bias at r. The symmetric statement holds for the right sibling of v (recall that v is the leftmost child of y). Suppose w (the join of u and v) or one of the nodes on the leftmost path descending from w is minor. Let this minor node be r. There must be a corresponding minor node r' on the leftmost path descending from u, such that $s(p(r)) \leq s(p(r'))$. Since the original tree is globally regular at r', the new tree must be globally regular at r. The symmetric statement holds for the rightmost path descending from w. Thus the new tree is globally biased before the split. The split preserves global bias.

Case 3—Similar to but simpler than Case 2. Case 4 is symmetric. □

Theorem 7: The worst-case running time of the global join algorithm is $O(\max\{s(x), s(y)\} - \max\{s(u), s(v)\})$, *where u is the rightmost external descendant of x and v is the leftmost external descendant of y.*

Proof: The global join algorithm descends rank by rank concurrently along the rightmost path descending from x and the leftmost path descending from y, until reaching a leaf; then it ascends. The theorem follows. □

We split a globally biased a, b tree in exactly the same way as a locally biased a, b tree, *using local rather than global joins.* This method not only produces globally biased trees, it has a worst-case time bound equal to the amortized bound given in Theorem 5.

Theorem 8: Algorithm 2 (or its variant for an item not in the tree) correctly splits a globally biased a, b tree with root r at a node x in $O(s(r) - s(x))$ *worst-case time.*

Proof: The proof is the same as the corresponding proof for globally

biased 2, b trees given in Ref. 8. For completeness, we sketch it here. Let $x_1, x_2, \cdots, x_k$ be the roots of the trees joined to form the final left tree. Let $y_1 = x_1$ and for $i = 2, \cdots, k$ let y_i be the root of the tree formed by executing join (x_i, y_{i-1}). With this definition y_k is the final left tree. Each node x_i is either a child of an ancestor of x, say a_i, in the original tree, or the newly constructed parent of a set of children of such a node a_i; furthermore a_{i+1} is a proper ancestor of a_i for $i = 1, \cdots, k-1$. Consider a node x_i for $i \geq 2$. If x_i is a child of a_i, global bias implies x_i is a major child, for otherwise its right sibling is external, which is impossible since this right sibling has x_1 or its children as ancestors. Thus $s(x_i) = s(a_i) - 1$. If x_i is the new parent of children of a_i, then $s(x_i) = s(a_i)$. It follows that $s(x_i) \leq s(x_{i+1})$ for $i = 1, \cdots, k-1$. As noted in the proof of Theorem 5, an induction shows that $s(y_i) \leq s(a_i)$ for $i = 1, \cdots, k$; if $i \geq 2$ and $s(x_i) = s(a_i)$, x_i has at most $b-1$ children, and the join of x_i and y_{i-1} cannot split x_i. Thus if $i \geq 2$, $s(a_i) - 1 \leq s(x_i) \leq s(y_i) \leq s(a_i)$.

Consider the join of x_{i+1} and y_i. The join will descend rank by rank along the rightmost path from x_{i+1} and the leftmost path from y_i. Global bias in the original tree implies that if the descent from x_{i+1} encounters a minor node z (other than the root of x_{i+1}), the leftmost external node of y_i will have rank at least $s(p(z)) - 1$ and the join will immediately terminate. Thus the join either terminates by reaching an external descendant of x_{i+1} of rank at least $s(y_i)$, in which case the global bias of the joined tree is immediate, or it reaches an internal descendant, say z, of x_{i+1} of rank exactly $s(y_i)$. Now we need a similar but more complicated statement about the leftmost path from y_i. There are several cases.

Case 1—$s(x_i) < s(a_i)$ and x_i is minor in the original tree. This can only happen if $i = 1$, i.e., $x_i = y_i$. Global bias implies that the rightmost external descendant of x_{i+1} has rank at least $s(a_i) - 1$. The join descends along the path from x_{i+1} to this external node and then ascends.

Case 2—$s(x_i) < s(a_i)$, x_i is major, and $s(x_i) < s(y_i)$. In this case y_i is also major and the leftmost paths descending from x_i and y_i, not including x_i and y_i themselves, are identical. If z is underfilled, y_i is external, and the join terminates in Case 1. If z is filled, y_i is either external or filled, and the join also terminates in Case 1.

Case 3—$s(x_i) < s(a_i)$, x_i is major, and $s(x_i) < s(y_i)$. In this case the left child of y_i is major and the join stops descending either at rank $s(y_i)$ (if y_i is external or filled) or at rank $s(y_i) - 1$ (if y is internal).

Case 4—$s(x_i) = s(a_i)$. In this case the leftmost paths descending from x_i and y_i, not including x_i and y_i themselves, are identical. If y_i is external or filled, the join will stop descending at rank $s(y_i)$. If y_i is underfilled, the join will stop at rank $s(y_{i-1})$; either the rightmost child

of z or the leftmost child of y_i is external of rank $s(y_{i-1})$, or both are filled and of rank $s(y_{i-1})$.

In all cases the global bias of the original tree implies that the joined tree is globally biased. This means that the final left tree is globally biased. Furthermore, the time required for joining x_{i+1} and y_i is $O(s(a_{i+1}) - s(a_i))$ in all cases. Summing over all joins gives a time bound of $O(s(r) - s(x))$ to form the final left tree. A symmetric argument applies to the right tree. □

If we implement insertion, deletion, and weight change as described in Section II, we obtain the following worst-case time bounds for the various operations on globally biased a, b trees (see Ref. 8):

$$\text{split } (i, r)\text{: } O\left(\log_a \left(\frac{w(r)}{w_i}\right)\right)$$

if i is in the tree, or

$$O\left(\log_a \left(\frac{w(r)}{w_{i^-} + w_{i^+}}\right)\right)$$

if i is not in the tree, where i^- and i^+ are the items before and after i in the tree, respectively.

$$\text{insert } (i, x)\text{: } O\left(\log_a \frac{w(x)}{w_{i^-} + w_{i^+}} + \log_a \frac{w(x) + w_i}{w_i}\right).$$

$$\text{delete } (i, x)\text{: } O\left(\log_a \frac{w(x)}{w_i} + \log_a \frac{W'}{w_{i^-} + w_{i^+}}\right),$$

where W' is the weight of the tree root after the deletion.

$$\text{reweight } (i, w, x)\text{: } O\left(\log_a \frac{w(x)}{w_i} + \log_a \frac{W'}{w}\right),$$

where $w(x)$ and w_i are as defined before the weight change, and W' is the weight of the tree root after the change.

As compared with the amortized time bounds for locally biased a, b trees, the worst-case bounds for globally biased a, b trees are larger for *join* and *delete* and the same for the other operations.

We conclude our discussion of biased a, b trees with two remarks. First, if $b \geq 2a$ we can implement either local or global join in an iterative, purely top-down fashion by preemptively splitting nodes with b children as they are encountered. By extending this idea we can implement all the operations top-down. This is a reason to choose $b = 2a$ over $b = 2a - 1$.

Second, for appropriate large values of a and b, biased a, b trees offer a biased alternative to B-trees. One of the advantages of B-trees is that there are no underfilled nodes except tree roots; thus if nodes

are stored one node per page in fixed-size pages, the storage efficiency is at least 50 percent, not counting root pages. Biased a, b trees do not share this property. We leave open the problem of devising a space-efficient version of biased a, b trees.

IV. PSEUDO-WEIGHT-BALANCED TREES

In biased a, b trees, we maintain balance through a height constraint. However, there are other possible balance constraints, such as weight balance. Nievergelt and Reingold[4] defined trees of bounded balance by imposing upper and lower bounds on the ratio *leftweight/rightweight* at each internal node, where the left and right weights count the number of items in the left and right subtrees, respectively. Bent developed a biased version of weight-balanced trees.[6] However, his data structure suffers from a complicated seven-case join algorithm that needs up to three recursive calls and also uses rebalancing rotations more complicated than standard single and double rotations. In this section we introduce a form of biased weight-balanced trees much simpler than Bent's. Our simplification comes from two new ideas: we discretize the weights and allow arbitrarily bad imbalance in some situations where balancing is possible. We call our trees pseudo-weight-balanced.

We consider binary trees, in which each internal node x has exactly two children, a left child $l(x)$ and a right child $r(x)$. As in Section II we define the *weight* $w(x)$ of a node x by $w(x) = w_i$ if x is an external node containing item i, $w(x) = w(l(x)) + w(r(x))$ if x is an internal node. We define the *rank* $s(x)$ of a node x differently: $s(x) = \lfloor \lg w(x) \rfloor$. We call a binary search tree *pseudo-weight-balanced* (pwb) if it has two properties:

1. If three nodes in a row, say x, $p(x)$, and $p(p(x))$, have the same rank, then x is external and either x is a left child and $p(x)$ a right child or vice-versa (see Fig. 5a).

2. If x and $l(x)$ (symmetrically x and $r(x)$) are internal nodes of the same rank, then $w(r(l(x))) + w(r(x))$ (symmetrically $w(l(r(x))) + w(l(x))$) is at least $2^{s(x)}$ (see Figure 5b). (This property allows us to do single rotations when necessary to maintain property 1.)

The following result bounds the access time in pwb trees.

Theorem 9: A pwb tree has ideal access time for all items. Specifically, if x is an external node containing item i in a tree of total weight W, then the depth of x is at most $2 \lg(W/w_i) + 3$.

Proof: Let r be the root of the tree and d the depth of x. According to property 1, after the first step up from x, every two steps taken along the path from x to r must cause a rank increase. Thus $\lfloor (d-1)/2 \rfloor \leq s(r) - s(x) = \lfloor \lg W \rfloor - \lfloor \lg w(x) \rfloor$, which implies $d \leq 2\lfloor (d-1)/2 \rfloor + 1 \leq 2(\lg W - \lg w(x) + 1) + 1 \leq 2 \lg (W/w(x)) + 3$. □

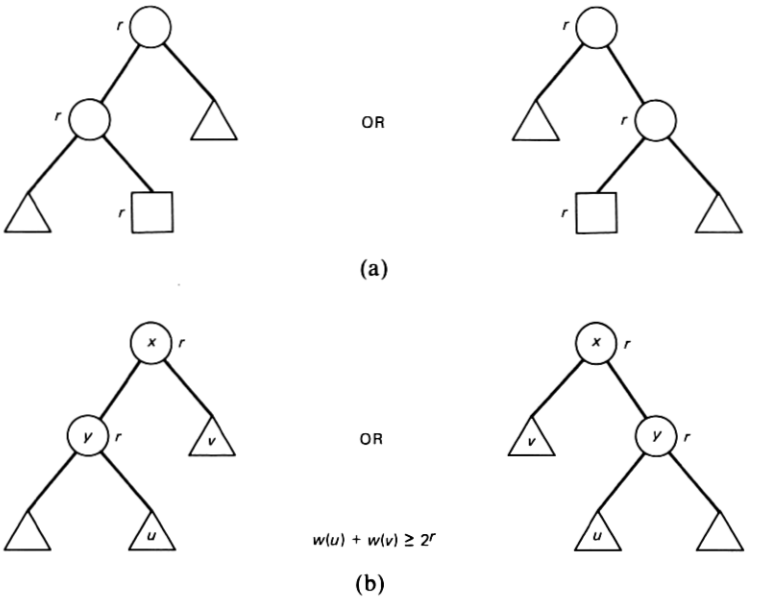

Fig. 5—Legal configurations in a pseudo-weight-balanced tree. (a) Three nodes of the same rank in a row. (b) Two internal nodes of the same rank in a row.

We join two pwb trees using the following algorithm:

Algorithm 4: join (x, y). Join two pwb trees with roots x and y, assuming that all items in tree x precede all items in tree y.

Case 0—$x =$ **null** or $y =$ **null.** Return y if $x =$ **null** or x if $y =$ **null.**

Case 1—$\lg(w(x) + w(y)) \geq 1 + \max\{s(x), s(y)\}$ or the heavier of x and y is an external node. Return a new root with left child x and right child y.

Case 2—$s(x) > s(y)$ and $\lg(w(x) + w(y)) < 1 + s(x)$ and x is not external. If $r(x)$ is external, or internal of rank at most $s(x) - 1$, replace $r(x)$ by *join* $(r(x), y)$. Otherwise, perform a single left rotation at x and replace the right child $r(u)$ of the new root u by *join* $(r(u), y)$ (see Fig. 6).

Case 3—$s(x) < s(y)$ and $\lg(w(x) + w(y)) < 1 + s(y)$. Symmetric to Case 2.

Remark: Although the join algorithm is stated recursively, it is easy to implement it in an iterative, purely top-down fashion, since the rank of a node depends only on the total weight of its descendants and not on their arrangement.

Theorem 10: *Algorithm 4 produces a pwb tree of rank* $\max\{s(x), s(y)\}$ *or* $1 + \max\{s(x), s(y)\}$.

Proof: By induction on the depth of the recursion. The definition of rank implies that the rank of the new tree is $\max\{s(x), s(y)\}$ or $1 +$

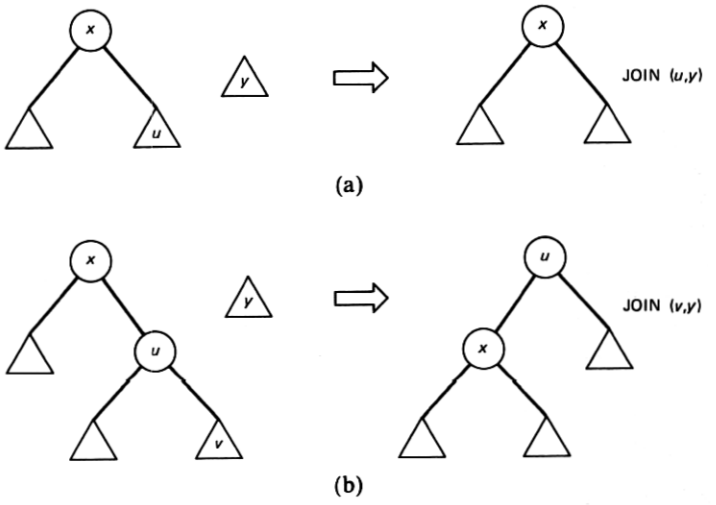

Fig. 6—Case 2 of the join algorithm for pwb trees. (a) Node $u = r(x)$ external or $s(u) < s(x)$: no rotation. (b) Node u internal and $s(u) = s(x)$: rotation.

$\max\{s(x), s(y)\}$. In the latter case, the join must have executed Case 1 and both children of the new root must have rank smaller than the root's rank. Case 1 obviously constructs a tree with properties 1 and 2. In Case 2, property 2 guarantees that the single rotation, if it occurs, creates a pwb tree. If the tree produced by the recursive join has rank less than $s(x)$, the overall joined tree clearly has properties 1 and 2. This is also true if the tree produced by the recursive join has rank $s(x)$, by the observation above. □

In analyzing the running time of Algorithm 4, we use the following credit invariant: Any node x contains $\max\{0, s(p(x)) - s(x) - 1\}$ credits.

Theorem 11: *Algorithm 4 runs in* $O(|s(x) - s(y)|) = O(\lg (w(x) + w(y))/\min\{w(x), w(y)\})$ *amortized time. Specifically, if we assume without loss of generality that* $s(x) \geq s(y)$, *performing the join while maintaining the credit invariant takes at most* $s(x) - s(y) + 1$ *credits.*

Proof: We consider the same cases as in the algorithm.

Case 1—We need one credit to build the new tree and either $s(x) - s(y)$ or $s(x) - s(y) - 1$ to establish the invariant on y, for a total of at most $s(x) - s(y) + 1$.

Case 2—Suppose the single rotation does not take place. We have on hand $s(x) - s(y) + 1$ tokens for performing the join plus $s(x) - s(r(x)) - 1$ from $r(x)$, for a total of $2s(x) - s(y) - s(r(x))$. We need one token for performing the outermost call of *join* plus $\max\{s(r(x)), s(y)\} - \min\{s(r(x)), s(y)\} + 1$ for the recursive call plus $s(x) - \max\{s(r(x)), s(y)\} - 1$ to establish the invariant on the root of the

tree returned by the recursive call, for a total of $s(x) - \min\{s(r(x)), s(y)\} + 1 \leq 2s(x) - s(y) - s(r(x))$, since $s(x) > \max(s(r(x)), s(y))$. Exactly the same argument applies if the rotation does take place, since the rotation preserves the credit invariant. □

The algorithm for splitting a pwb tree is almost identical to but simpler than the algorithm for splitting a biased a, b tree.

Algorithm 5: *split* (x, r). Split a pwb tree with root r at a node x.

Initialize the current node *cur*, the previous node *prev*, and the left and right nodes *left* and *right* to be $p(x)$, x, $l(x)$, and $r(x)$, respectively. Repeat the following step until *cur* = **null:**

Split step—If $prev = l(cur)$, replace *right* by *join* (*right*, $r(cur)$); otherwise, replace *left* by *join* ($l(cur)$, *left*). Remove *prev* as a child of *cur* and destroy it if it is not x. Replace *prev* and *cur* by *cur* and $p(cur)$, respectively.

This algorithm is the same as that described by Bent, Sleator, and Tarjan[8] for splitting biased binary trees, and indeed will work for any class of binary search trees for which a join algorithm is known.

Theorem 12: The amortized time of *split* (x, r) is

$$O\left(\lg\left(\frac{w(r)}{w(x)}\right)\right).$$

More precisely, performing the split while maintaining the credit invariant takes $O(s(r) - s(x))$ credits, where x is the node containing item i.

Proof: The definition of ranks implies that $s(prev) \geq \max\{s(left), s(right)\}$ before each split step. An easy induction shows that if we allocate $O(s(cur) - s(prev) + 1)$ credits to each split step, we can carry out the step while maintaining the credit invariant on the trees *left* and *right* and in addition maintaining $2s(prev) - s(left) - s(right)$ credits on hand. Summing over all split steps and using property 2 gives the theorem. □

Using the appropriate combinations of *join* and *split*, we obtain the same amortized time bounds as in Section II (with binary logarithms) for insertion, deletion, and weight change in pwb trees. Pseudo-weight-balanced trees are a very simple version of locally biased trees, competitive with the biased binary trees presented in Ref. 8. We have been unable to devise a globally biased version of pwb trees and leave this as an open problem.

REFERENCES

1. G. M. Adelson-Vel'skii and Y. M. Landis, "An algorithm for the organization of information," Soviet Math. Dokl., *3*, No. 5 (September 1962), pp. 1259–62.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.

3. R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," Acta Info., *1*, No. 3 (1972), pp. 173–89.
4. J. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," SIAM J. Comput., *2* (1973), pp. 33–43.
5. N. Abramson, *Information Theory and Coding*, New York: McGraw-Hill, 1963.
6. S. W. Bent, "Dynamic weighted data structures," Ph.D. thesis, Computer Science Department, Stanford University, Stanford, CA, 1982.
7. S. W. Bent, D. D. Sleator, and R. E. Tarjan, "Biased 2-3 trees," Proc. Twenty-First Annual IEEE Symp. on Foundations of Computer Science, October 13–15, 1980, pp. 248–54.
8. S. W. Bent, D. D. Sleator, and R. E. Tarjan, "Biased search trees," unpublished work.
9. S. Huddleston and K. Mehlhorn, "Robust balancing in B-trees," Lecture Notes in Computer Science, *104* (1981), Berlin: Springer-Verlag, pp. 234–44.
10. S. Huddleston and K. Mehlhorn, "A new data Structure for representing sorted lists," Acta Info., *17*, No. 2 (1982), pp. 157–84.
11. D. Maier and S. C. Salveter, "Hysterical B-trees," Info. Proc. Letters, *12*, No. 4 (August 1981), pp. 199–205.

AUTHORS

Joan Feigenbaum, B.A. (Mathematics), 1981, Harvard University. Ms. Feigenbaum is a graduate student in the Computer Science Department of Stanford University and holds a grant from the Bell Laboratories Graduate Research Program for Women. She spent the summers of 1980, 1981, and 1982 at Bell Laboratories. During the summer of 1982, she was a member of the Mathematical Foundations of Computing Department. Her current research is in the areas of data structures and cryptography. Member, Phi Beta Kappa.

Robert E. Tarjan, B.S. (Mathematics), 1969, California Institute of Technology; M.S., 1971, and Ph.D., 1972 (Computer Science), Stanford University; Cornell University, 1972–1973; University of California, Berkeley, 1973–1975; Stanford University, 1975–1980; Bell Laboratories, 1980—. Mr. Tarjan is a member of the Mathematical Foundations of Computing Department, where he has been studying the design and analysis of efficient data structures and combinatorial algorithms. Member, ACM, SIAM, AAAS, Tau Beta Pi, and Sigma Xi.