SIAM J. COMPUT.
Vol. 14, No. 3, August 1985

004

BIASED SEARCH TREES*

SAMUEL W. BENT†, DANIEL D. SLEATOR‡ AND ROBERT E. TARJAN‡

Abstract. We consider the problem of storing items from a totally ordered set in a search tree so that the access time for a given item depends on a known estimate of the access frequency of the item. We describe two related classes of *biased search trees* whose average access time is within a constant factor of the minimum and that are easy to update under insertions, deletions and more radical update operations. We present and analyze efficient update algorithms for biased search trees. We list several applications of such trees.

Key words. optimum search trees, binary search trees, data structures, algorithms, complexity

1. Introduction. The following problem, which we shall call the *dictionary problem*, occurs frequently in computer science. We are given a totally ordered universe U. We wish to represent sets $S \subseteq U$ in such a way that the following access and update operations are efficient:

access (i, S): If item i is an element of set S, return a pointer to its location. Otherwise return a special **null** pointer.

insert (i, S): Insert item i into set S. We allow an insertion to take place only if i is not initially an element of S.

delete (i, S): Delete item i from set S. We allow a deletion only if i is initially in S.

In addition to *access*, *insert* and *delete*, the following more radical update operations are often useful:

join (R, S): (two-way join). Return the set consisting of the union of R and S. This operation destroys R and S, and is allowed only if every item in R is less than every item in S. (Thus a join can be regarded as the concatenation of the sorted sets R and S.)

join (R, i, S): (three-way join). Return the set consisting of the union of R, $\{i\}$ and S. This operation destroys R and S, and is allowed only if every item in R is less than i and every item in S is greater than i.

split (i, S): Split S into three sets: P, containing all items in S less than i; Q, containing i if $i \in S$ (three-way split) and nothing if $i \notin S$ (two-way split); and R, containing all elements in S greater than i. This operation destroys S.

One kind of data structure that efficiently supports these operations is a *search tree*. A search tree is an ordered tree (Appendix A contains our tree terminology) containing the items of a set in its leaves, in left-to-right order, one item per leaf. In order to facilitate the access operation, we must also store auxiliary items, called *keys*, in the internal nodes. (Appendix B discusses the placement, use and updating of keys.) To access an item, we start at the root of the tree, compare the item being accessed with the key(s) in this node, go to a child determined by the outcome of the comparison(s), and continue in this way until reaching a leaf. This leaf contains the item if it is in the tree. With this access method, the time to access an item is proportional

* Received by the editors March 15, 1983, and in revised form March 7, 1984. Research partially supported by the National Science Foundation under grant MCS 82-03238.

† Computer Science Department, University of Wisconsin, Madison, Wisconsin 53706.

‡ AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

to the depth of the leaf containing it, if we assume a fixed upper bound on the degree of a node.

If the search tree satisfies an appropriate balance condition, then the height of the tree, and hence the worst-case access time, is $O(\log n)$[1], where n is the number of items in the set. For a comparison-based model of computation, one can prove a lower bound of $\Omega(\log n)$ on the worst-case access time; thus balanced trees have a worst-case access time within a constant factor of the minimum. Kinds of balanced trees include height-balanced trees [2], [20], two-three trees [3], B-trees [6], weight-balanced trees [27], red-black trees [12] and many others. These kinds of trees all have the additional property that each of the update operations can be carried out in $O(\log n)$ time.

In many applications of search trees, the access frequencies are different for different items. In such a situation we would like to *bias* the search tree, so that the more-frequently needed items can be accessed faster than the less-frequently needed ones. In order to treat this problem formally let us assume that each item i has a known *weight* $w_i > 0$ representing the access frequency. A measure of the average access time is

$$\sum_{i \in S} w_i \frac{(d_i + 1)}{W},$$

where $W = \sum_{i \in S} w_i$ is the sum of the weights of the items in the set and d_i is the depth in the search tree of the node containing item i. Our goal is to make the total weighted depth $\sum_{i \in S} w_i d_i$ as small as possible while preserving the ability to update the search tree rapidly. We call this the *biased dictionary problem.* It is natural in this problem to allow an additional operation for changing the weight of an item:

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

In this paper we shall propose two kinds of trees, which we call *biased search trees*, for solving the biased dictionary problem. Our results provide not only specific kinds of search trees, but also a general methodology for converting almost any class of balanced search trees into a similar but more general class of biased search trees. In addition to developing new ideas, the paper extends and refines ideas first presented earlier in preliminary form [7], [8].

The paper contains five sections. Section 2 reviews relevant previous work. Sections 3 and 4 define and analyze the properties of two kinds of biased search trees. Section 3 describes *biased 2, b trees*, which are analogous to 2, 3 trees and B-trees. Section 4 describes *biased binary trees*, which generalize a particular kind of red-black tree [12] sometimes called a symmetric binary B-tree [5]. Section 5 summarizes our results and discusses several applications and related work. Appendix A contains our tree terminology. Appendix B describes the arrangement of the keys in a search tree, their use for search and their updating.

2. Previous research. Several known results bear on the biased dictionary problem. The first and most important is a standard theorem of information theory.

THEOREM A [1]. *Consider any search tree T for a set S. If every node of T has at most b children, then the total weighted depth $\sum_{i \in S} w_i d_i$ is at least $W \sum_{i \in S} p_i \log_b (1/p_i)$, where $p_i = w_i / W$.*

In light of Theorem A, our goal is to devise classes of search trees with the property that $d_i = O(\log(W/w_i))$ for each item i, since any such search tree has minimum average

[1] If f and g are functions of a nonnegative real number x, we write "$f(x)$ is $O(g(x))$" if there are positive constants c_1 and c_2 such that $f(x) \leqq c_1 g(x) + c_2$ for all x. We write "$f(x)$ is $\Omega(g(x))$" if $g(x)$ is $O(f(x))$.

access time to within a constant factor. We shall call $O(\log(W/w_i))$ the *ideal access time* of item i. If all item weights are equal the ideal access time for every item is $O(\log n)$ and any balanced search tree has ideal access time for all items.

Much previous work deals with the case of a *static* search tree. Suppose we are given a fixed set S whose items have known weights and we want to construct a search tree of exactly minimum total weighted depth. We call such a search tree *optimum.* Knuth [19] (see also Yao [33]) has given a dynamic programming algorithm that computes an optimum search tree among binary trees containing one item per internal node (instead of one item per leaf). Knuth's algorithm runs in $O(n^2)$ time and allows for the possibility that weights are given not only for the items in the set but also for the gaps between items; these gaps correspond to the possible ways to search for an item not in the set.

An algorithm of Hu and Tucker [17] (see also Garsia and Wachs [11], Hu, Kleitman and Tamaki [16], and Hu [15]) finds an optimum search tree among binary trees with one item per leaf, assuming no weights are given for the gaps. (This is a special case of the problem solved by Knuth.) The Hu–Tucker algorithm runs in $O(n \log n)$ time and resembles Huffman's algorithm [18] for computing an optimum binary prefix code. Fredman [10], Mehlhorn [24] and Korsch [21] have proposed $O(n)$-time algorithms that construct binary search trees whose total weighted depth is within a constant factor of minimum.

None of these algorithms is satisfactory in the dynamic case, since they require completely restructuring the tree (spending $\Omega(n)$ time) each time an update occurs. Several authors have proposed classes of biased search trees that are easier to update. Baer [4] gave a heuristic for rebalancing biased weight-balanced search trees, but he gave no theoretical results, and indeed his trees do not have ideal access time in the worst case. Unterauer [32] described a class of biased weight-balanced trees that have ideal access time, but he did not analyze the worst-case time required for updates. Mehlhorn [26] described a class of biased search trees based on weight-balanced trees, called *D-trees.* D-trees have ideal worst-case access time and require $O(\min\{n, \log(W'/w_0)\})$ time for insertion, where w_0 is the weight of the smallest item and W' is the total weight of all the items in the set after the insertion. Mehlhorn [25] subsequently showed that a weight change in a D-tree can be performed in $O(\log(\max\{W, W'\}/\min\{w_i, w_i'\}))$ time, where w_i, w_i', W, W' are the old weight of the reweighted item, the new weight of this item, the old total weight and the new total weight, respectively. Kriegel and Vaishnavi [22] proposed another version of biased search trees with time bounds similar to those of Mehlhorn.

The kinds of biased search trees we propose here are simpler than those proposed by Mehlhorn and Kriegel and Vaishnavi. They also have faster running times for insertion, deletion, join and split. (Mehlhorn does not consider join and split; Kriegel and Vaishnavi's bound for split is the same as ours but their bound for join is worse.)

3. Biased 2, *b* trees. Our first class of biased search trees generalizes 2, 3 trees [3]. A 2, b *tree* is a search tree each of whose internal nodes has at least 2 and at most b children, where b is any fixed integer greater than two. We define the *rank* $s(x)$ of a node x in a 2, b tree recursively by

$$s(x) = \begin{cases} \lfloor \lg w_i \rfloor & \text{if } x \text{ is a leaf containing item } i,^2 \\ 1 + \max\{s(y) \mid y \text{ is a child of } x\} & \text{if } x \text{ is an internal node.} \end{cases}$$

[2] We denote $\log_2 x$ by $\lg x$.

This definition implies that if y is a node other than the tree root and x is its parent, then $s(y) \leqq s(x) - 1$. We call y *major* if $s(y) = s(x) - 1$ and *minor* if $s(y) < s(x) - 1$. By convention the root is major. A *locally biased* 2, *b tree* is a 2, b tree satisfying the following property, which constrains the environment of minor nodes (see Fig. 1).

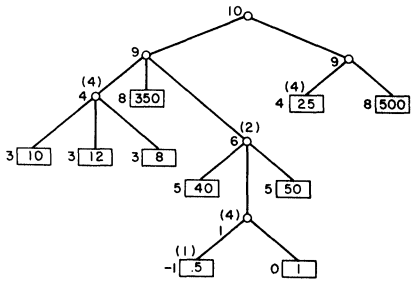

FIG. 1. *A locally biased* 2, 3 *tree. Leaves are rectangles, internal nodes are circles. Numbers inside leaves are weights. Numbers to the left of nodes are ranks; those above nodes (in parentheses) are credit counts. For clarity the items in nodes are omitted.*

Local bias. Any neighboring sibling of a minor node is a major leaf. (We say the tree is *locally biased* at the minor node.)

A 2, b tree is *balanced* if every leaf has the same depth. (Our definition of a balanced 2, 3 tree coincides with the usual definition of a 2, 3 tree.) In the case of equal-weight items, a 2, b tree is balanced if and only if it is biased; if all weights are one, the rank of a node is its height in the tree.

Our first results show that biased 2, b trees have ideal access time for all items. If x is a node, let $w(x)$ be the total weight of the items in leaves that are descendants of x; that is, $w(x)$ equals w_i if x is a leaf containing item i, and $w(x)$ equals $\sum \{w(y) | y$ is a child of $x\}$ if x is an internal node.

LEMMA 1. *For any node* x, $2^{s(x)-1} \leqq w(x)$. *If* x *is a leaf,* $2^{s(x)} \leqq w(x) < 2^{s(x)+1}$.

Proof. By induction on $s(x)$. If x is a leaf, the definition $s(x) = \lfloor \lg w(x) \rfloor$ implies $2^{s(x)} \leqq w(x) < 2^{s(x)+1}$. If x is an internal node with a minor child, x has a major child, say y, that is a leaf, and $2^{s(x)-1} = 2^{s(y)} \leqq w(y) \leqq w(x)$. If x is an internal node with no minor children, x has at least two major children, say y and z, and $2^{s(x)-1} = 2^{s(y)-1} + 2^{s(z)-1} \leqq w(y) + w(z) \leqq w(x)$. □

LEMMA 2. *If* x *is a leaf of depth* d *containing item* i, $d < \lg(W/w_i) + 2$.

Proof. Let r be the root of the tree. Since the rank increases by at least one from child to parent, $d \leqq s(r) - s(x)$. By Lemma 1, $\lg W = \lg w(r) \geqq s(r) - 1$ and $\lg w(x) < s(x) + 1$. Combining inequalities gives the lemma. □

THEOREM 1. *A biased* 2, b *tree has ideal access time for all items.*

Proof. Immediate from Lemma 2. □

In our analysis of the running times of the update operations on biased 2, b trees, we shall use amortization. That is, we shall average the running time of individual update operations over a (worst-case) sequence of updates. In order to make the analysis as concrete as possible, we introduce the concept of *credits* (called *chips* in [7], [8]). A credit represents one unit of computing time. To perform an update operation, we are given a certain number of credits. Spending one credit allows us to perform $O(1)$ computational steps. If we complete the operation before running out of credits, we can save the extra credits to use on future operations. If we run out of credits before completing the operation, we can spend previously saved credits. If we can perform a sequence of operations without running out of credits during the process,

then the number of credits allocated to each operation gives an upper bound on its amortized running time. We call this upper bound the *amortized time* of the operation.

Notice that if this analytical technique is successful, then any sequence of update operations requires actual time at most a constant times the sum of the amortized times of the individual operations; a later operation may require more actual time than its amortized time, but only if a corresponding amount of time was saved in earlier operations. Some algorithms, such as the path compression method of maintaining disjoint sets [31], exhibit the opposite behavior: early operations can be slower-than-average but only if the time lost is made up in later operations.

In the case of biased 2, b trees, we shall keep track of credits saved from previous operations by storing them in the trees. In particular, we say a biased 2, b tree satisfies the *credit invariant* if every minor node y with parent x contains $s(x)-s(y)-1$ credits. (See Fig. 1.) Note that this definition is consistent with a major node having no credits; in particular a single-node tree needs no credits. For each update operation we shall give an upper bound on the number of credits needed to perform the operation, assuming that the initial trees satisfy the credit invariant and requiring that the final trees satisfy the invariant. It is important to remember that the credits in a tree are only a conceptual device to aid in the running time analysis and neither appear in the data structure nor affect the actual implementation of the update algorithms.

We first consider (two-way) join, since all the other update operations can be defined in terms of this one. We shall describe an algorithm that joins two trees with roots x and y and returns the root of the resulting tree. The algorithm is recursive and consists of three main cases, two of which are symmetric (see Fig. 2).

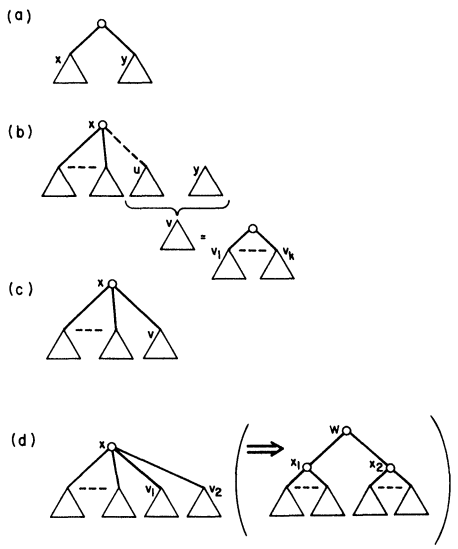

FIG. 2. *Join algorithm for locally biased 2, b trees. Triangles denote subtrees.* (a) *Case* 1 (*terminate*). (b) *Case* 2 (*recurse*). *Join right child u of x to y, forming v.* (c) *Subcase* 2a (*v has rank less than x*). *Attach v as right child of x in place of u.* (d) *Subcase* 2b (*v has same rank as x*). *Attach children of v as children of x. Split x if it has more than b children.*

Case 1. $s(x)=s(y)$, *or* $s(x)>s(y)$ *and x is a leaf, or* $s(x)<s(y)$ *and y is a leaf.* Create and return a new node with nodes x and y as its two children.

Case 2. $s(x)>s(y)$ *and x is not a leaf.* Let u be the right child of x. Remove u as a child of x and recursively join the trees with roots u and y, producing a single tree, say with root v.

Subcase 2a. $s(v) \leqq s(x)-1$. Attach v as the right child of x and return x.

Subcase 2b. $s(v) = s(x)$. In this case v has exactly two children. Attach these as children of x (to the right of the other children of x) and destroy v. Node x has now gained a child. If x now has no more than b children, return x. Otherwise split x into two nodes with $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ children, respectively. Create a new node w with these two nodes as children and return w. The two nodes resulting from the split have the same rank as x; the rank of w is one greater.

Case 3. $s(x) < s(y)$ and y is not a leaf. This case is symmetric to Case 2.

Note. When node x is split in Subcase 2b, it is not necessary to divide its children approximately fifty-fifty; it is sufficient that each new node have at least two children.

The first thing we must verify about this algorithm is its correctness. An easy induction argument shows that the algorithm produces a 2, b tree whose root has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\}+1$; in the latter case the root has exactly two children. This verifies the assertion at the beginning of Subcase 2b. A similar induction based on the following observations shows that the algorithm produces a biased 2, b tree given two biased 2, b trees as input:

Case 1. If x is minor in the new tree, y is a leaf, which means that the new tree is locally biased at x; similarly if y is minor.

Subcase 2a. If v is minor in the new tree, u was minor in the old tree and the old left sibling of u, which is now the left sibling of v, is a major leaf, giving local bias at v. On the other hand, if the left sibling of v is minor in the new tree, it must be the case that u is a leaf of rank $s(x)-1$. But then Subcase 2b would have occurred instead of Subcase 2a. Thus the new tree is locally biased.

Subcase 2b. If the left child of v is minor, the right child of v is a leaf, which can only happen if the children of v are u and y and the latter is a leaf. If the left sibling of u in the old tree is minor, u is a leaf of rank $s(x)-1$, and in this case also the children of v are u and y. It follows that the tree existing after x gains a child but before x splits is locally biased. Splitting preserves local bias, which means that the final tree is locally biased.

Thus the join algorithm is correct. We shall prove by a similar case analysis that if we allocate $|s(x)-s(y)|+1$ credits to the join, we can perform the join while preserving the credit invariant. Thus the join requires $O(|s(x)-s(y)|)$ amortized time. To carry out the analysis, we assume $s(x) \geqq s(y)$. (The case $s(x) < s(y)$ is symmetric.) We begin the join with $s(x)-s(y)+1$ credits in hand.

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

Case 2. We acquire $s(x)-s(u)-1$ credits from u, giving us a total of $2s(x)-s(y)-s(u)$. We need $\max\{s(u), s(y)\}-\min\{s(u), s(y)\}+1$ to recursively join the trees with roots u and y.

Subcase 2a. We need one credit to build the new tree and $s(x)-s(v)-1$ to place on v. If $s(y) \geqq s(u)$, we use a total of $s(y)-s(u)+s(x)-s(v)+1 \leqq 2s(x)-s(y)-s(u)$, since $s(v) \geqq s(y)$ and $s(x) > s(y)$. If $s(y) < s(u)$, we use $s(u)-s(y)+s(x)-s(v)+1 \leqq 2s(x)-s(y)-s(u)$, since $s(x) > s(u)$ and $s(v) \geqq s(u)$.

Subcase 2b. We need one credit to build the new tree. We need no credits to place on the new children of x, since as children of v they already have the proper number of credits. Splitting x preserves the credit invariant. (Local bias implies that both nodes resulting from the split have rank $s(x)$.) Thus, we use a total of $\max\{s(u), s(y)\}-\min\{s(u), s(y)\}+2$ credits. The analysis in Subcase 2a shows that this is at most $2s(x)-s(y)-s(u)$.

Summarizing our analysis, we have the following theorem:

THEOREM 2. *The two-way join algorithm is correct and runs in* $O(|s(x)-s(y)|)$ *amortized time on two trees with roots x and y.*

There is a useful alternative formulation of this theorem. We say a tree with root x is *cast to rank* k if it satisfies the credit invariant and has $k-s(x)$ additional credits on its root. If x and y are the roots of two trees cast to a rank $k > \max\{s(x), s(y)\}$, then Theorem 2 implies that we can join these trees using no extra credits, producing a single tree cast to rank k.

We can describe the behavior of the join algorithm as follows (see Fig. 3): Traverse the right path of the tree rooted at x and the left path of the tree rooted at y concurrently, descending rank-by-rank, until arriving at a leaf in one path or at two nodes of equal rank, one in each path. Merge the traversed parts of the paths, ordering nodes in decreasing order by rank. If the traversal stops at two nodes of equal rank, say k,

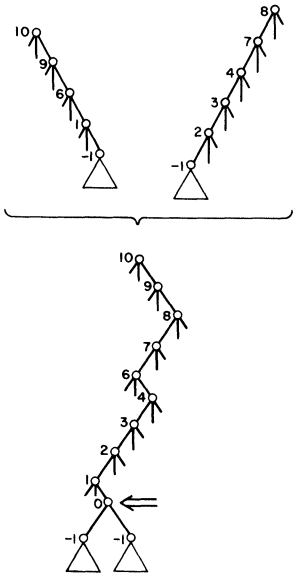

FIG. 3. *A join. Only right path of left tree, left path of right tree are shown. Numbers beside nodes are ranks. Node indicated by arrow is created by join.*

either make them both children of the previous node on the merged path, if the previous node has rank $k+1$, or else make them children of a new node with rank $k+1$, whose parent is the previous node. Starting from the bottom of the merged path and working up toward the root, split nodes as necessary until reaching a node with no more than b children. This description implies the following worst-case time bound for join:

THEOREM 3. *Consider a join of two trees with roots x and y such that the rightmost leaf descendant of x is u and the leftmost leaf descendant of y is v. The worst-case join time is* $O(\max\{s(x), s(y)\} - \max\{s(u), s(v)\}) = O(\log(W/(w_- + w_+)))$, *where W is the total weight of the items in the new tree,* $w_- = w(u)$, *and* $w_+ = w(v)$.

Note. The worst-case time for a join can be either larger or smaller than its amortized time.

Let us now consider the other update operations, beginning with three-way join. We can implement a three-way join as two successive two-way joins. Theorems 2 and 3 give the following time bounds:

THEOREM 4. *Consider the three-way join of a tree with root x to a leaf y and to a tree with root z. The amortized time for the join is* $O(\max\{s(x), s(y), s(z)\} - \min\{s(x), s(y), s(z)\})$. *The worst-case time for the join is* $O(\max\{s(x), s(y), s(z)\} - s(y)) = O(\log(W/w_i))$, *where W is the total weight of the items in the new tree and i is the item in node y.*

Note. The worst-case join time is the same as the access time for item i in the new tree and never exceeds the amortized join time.

A split can be implemented as a sequence of (two-way) joins. Let us first consider splitting at an item i already in the tree. Let x be the root of the tree to be split and y the leaf containing item i. The split will proceed up the path from y to x, accumulating a *left tree* of items less than i and a *right tree* of items greater than i. Initially y is the *previous node*, the parent of y is the *current node*, and the left and right trees are empty. The split consists of repeating the following general step until the root is the previous node (see Fig. 4):

General 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 new 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 y. Make the current node the new previous node and its parent the new current node.

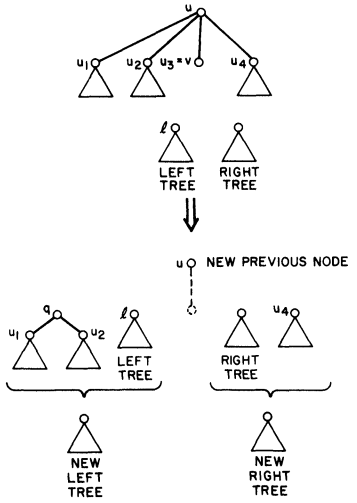

FIG. 4. *One step of split algorithm. Node u is the current node, v the previous node,* $u_1, u_2, \cdots$ *the children of u, q the root of the tree that is joined to the left tree and l the root of the left tree.*

This algorithm is obviously correct. To establish its amortized time, let u be the current node, v the previous node, q the root of the tree containing children of u that is joined to the left tree, and l the root of the left tree. An easy induction shows that $s(v) \geqq s(l)$. Suppose we begin the current execution of the general step with both the left and the right tree cast to rank $s(v)+1$. The following argument shows that with $2(s(u)-s(v))+5$ additional credits, we can carry out the general step and finish with both the left and the right tree cast to rank $s(u)+1$. If we place at most two new credits on q and $s(u)-s(v)$ new credits on l, we can join q and l to produce a new left tree cast to rank $s(u)+1$, since either $s(q)=s(u)$ (if u has two more children to the left

of v) or $s(q) \leqq s(u)-1$ and q has $s(u)-s(q)-1$ credits on it already (if u has one child to the left of v). Similarly we need $s(u)-s(v)+2$ credits for the right join. One new credit accounts for the $O(1)$ time required for the rest of the general step, giving a total credit count of $2(s(u)-s(v))+5$.

Summing over all executions of the general step, we obtain:

THEOREM 5. *The amortized time to split a tree with root x at leaf y is $O(s(x)-s(y)) = O(\log(W/w_i))$, where W is the weight of all the items in the tree and i is the item in leaf y. Each of the (up to) three resulting trees is cast to rank $s(x)+1$.*

Splitting at an item not in the tree is just like splitting at an item in the tree, except that the initial execution of the general step is slightly different. Let x be the root of the tree, i an item not in the tree, i^- and i^+ the largest item in the tree less than i and the smallest item in the tree greater than i, respectively. And let y be the *handle* of i, which is defined to be the nearest common ancestor of the leaves containing i^- and i^+. To split the tree, we combine all children of y containing items smaller than i; the result becomes the original left tree. We combine the remaining children of y (those containing items greater than i) to form the original right tree. Then we make y the previous node and its parent the new current node and repeat the general step as before.

THEOREM 6. *The amortized time to split a tree with root x at an item i not in the tree is $O(s(x)-s(y)) = O(\log(W/(w_{i^-}+w_{i^+})))$, where y is the handle of i, and i^-, i^+ are as defined above. Each of the (up to) two resulting trees is cast to rank $s(x)+1$.*

Unlike join, split does not have a logarithmic bound on its worst-case running time. However, as we shall see at the end of this section, we can get a good bound on the worst-case split time by strengthening the bias property and changing the implementation of join to maintain this stronger property.

We can implement each of the remaining update operations as a combination of a split and a join: an insertion is a two-way split followed by a three-way join, a deletion is a three-way split followed by a two-way join, and a weight change is a three-way split followed by a three-way join. The next theorem gives the amortized time of these operations.

THEOREM 7. *The amortized time to perform an insertion of item i into a tree is $O(\log(W'/\min\{w_{i^-}+w_{i^+}, w_i\}))$, where W' is the weight of the tree after the insertion and i^- and i^+ are the largest item smaller than i and the smallest item larger than i, respectively. The amortized time to perform a deletion of item i from a tree is $O(\log(W/w_i))$, where W is the weight of the tree before the deletion. The amortized time to perform a weight change on item i in a tree is $O(\log(\max\{W, W'\}/\min\{w_i, w_i'\}))$, where W, W', w_i, w_i' are the weights of the tree before and after the update and the weights of i before and after the update, respectively.*

Proof. Consider an insertion. The two-way split takes amortized time $O(\log(W/(w_{i^-}+w_{i^+}))$, where W is the weight of the original tree, and produces trees cast to a rank of at least $\lfloor \lg(\max\{w_{i^-}, w_{i^+}\}) \rfloor$. The three-way join thus requires $O(\log(W'/\min\{w_{i^-}+w_{i^+}, w_i\}))$ additional amortized time. This gives the bound for insertion, since $W \leqq W'$. The three-way split beginning a deletion requires $O(\log W/w_i)$ time; the two-way join completing it takes $O(1)$ additional amortized time since the trees resulting from the split are cast to the same rank. This gives the bound for deletion. The three-way split beginning a weight change also requires $O(\log W/w_i)$ amortized time and produces trees cast to a rank of at least $\lfloor \lg w_i \rfloor$. The three-way join completing the weight change thus requires $O(\log(\max\{W, W'\}/\min\{w_i, w_i\}))$ additional amortized time. This gives the bound for weight change. □

Remark. In practice it may be useful to design customized implementations of insert, delete and weight change, rather than expressing them in terms of join and

split. We leave this as a (nontrivial) exercise; the algorithms so obtained are more complicated than those using join and split.

The data structure we have described and analyzed is a good one if amortized running time is the complexity measure of interest. We shall now describe a modification appropriate if worst-case per-operation running time is important. A *globally biased* 2, *b tree* is a 2, *b* tree with the following property, which is stronger than local bias (see Fig. 5):

Global bias. Any neighboring leaf of a minor node y with parent x has rank at least $s(x)-1$. (We say the tree is *globally biased* at y.)

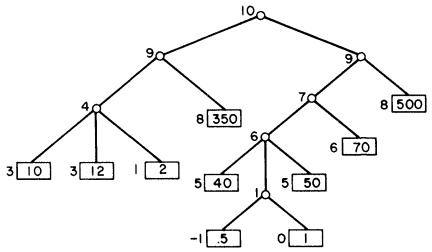

FIG. 5. *A globally biased* 2, 3 *tree. Weights are inside leaves, ranks are to left of nodes.*

Since any globally biased 2, *b* tree is locally biased, globally biased 2, *b* trees have ideal access time. The following version of the join algorithm will join two globally biased 2, *b* trees with roots x and y into a single globally biased 2, *b* tree and return the root of the new tree. (See Figs. 2 and 6.)

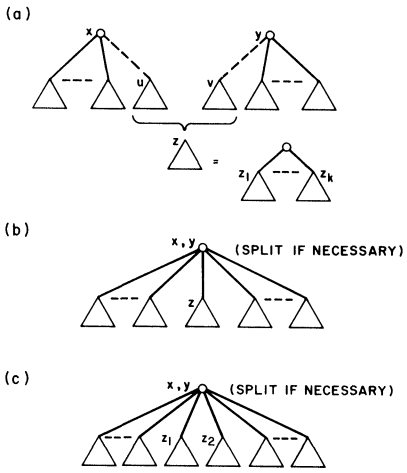

FIG. 6. *Case* 4 *of global join algorithm* (*equal ranks, nonterminating*). (a) *Right child of x, left child of y joined to form z.* (b) *Rank of z less than x. Attach z as right child of x, fuse x and y, split if necessary.* (c) *Ranks of z and x equal. Fuse x, z, y; split if necessary.*

Case 1. $s(x) \geqq s(y)$ *and x is a leaf, or* $s(x) \leqq s(y)$ *and y is a leaf.* Create a new node u with nodes x and y as its two children and return u.

Case 2. $s(x) > s(y)$ *and x is not a leaf.* Proceed as in Case 2 of the join algorithm for locally biased trees.

Case 3. $s(x) < s(y)$ *and y is not a leaf.* Symmetric to Case 2.

Case 4. $s(x) = s(y)$ *and neither* x *nor* y *is a leaf.* Let u be the right child of x and v the left child of y. Remove u as a child of x and v as a child of y. Recursively join the trees rooted at u and v, producing a single tree, say with root z. If $s(z) < s(x)$, attach z as the right child of x; otherwise ($s(z) = s(x)$) attach the two children of z as (right-most) children of x and destroy z. Fuse x and y into a single node. If this new node has no more than b children, return it; otherwise split it and return a new node whose two children are the results of the split.

We shall refer to the join algorithm for locally biased trees as *local join* and to the version for globally biased trees as *global join.* We can describe a global join iteratively as follows (see Fig. 7): Traverse the right path of the left tree and the left

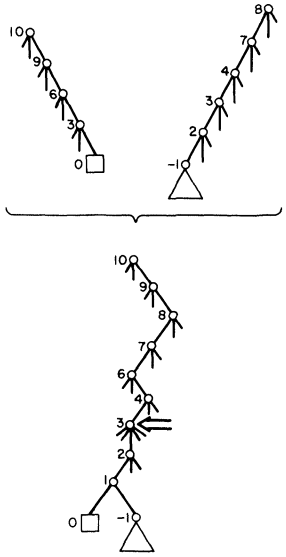

FIG. 7. *A global join. Nodes on spliced path must be split if necessary, working bottom-up. Node indicated by arrow is formed by fusing two nodes, one from each tree.*

path of the right tree concurrently, descending rank-by-rank until reaching a leaf in one of the paths. Merge the traversed parts of the paths, ordering nodes in decreasing order by rank and fusing any two nodes of equal rank. If the traversal stops at two leaves of equal rank, say k, do not fuse them but instead either make them children of the previous node on the spliced path, if the previous node has rank $k+1$, or else make them children of a new node with rank $k+1$ whose parent is the previous node. Proceed back up the merged path, splitting every node with more than b children.

As with a local join, a global join of trees with roots x and y produces a 2, b tree whose root has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\}+1$; in the latter case the root has exactly two children. The following discussion shows that the new tree is globally biased. Let the *left tree* and the *right tree* be the original trees with roots x and y, respectively. Consider the tree produced by applying the global join algorithm without doing any splitting. We call this the *fused tree.* The only possible nodes at which the fused tree might not be globally biased are minor nodes along the spliced path; let v with parent u be such a node. Node v has leaf descendants in either the left tree, the right tree, or both; v is either a node, say q, in the left tree, a node, say r, in the right tree, a node produced by fusing two nodes, say q from the left tree and r from the right tree, or a new node with two children (at least one of which is a leaf),

say q from the left tree and r from the right tree. If q exists, its left neighboring leaf in the left tree has rank at least $s(u)-1$ and is the left neighboring leaf of v in the fused tree. Similarly if r exists, its right neighboring leaf in the right tree has rank at least $s(u)-1$ and is the right neighboring leaf of v in the fused tree. Suppose q does not exist. Then v is a minor node in the right tree. Let g with parent f be the node on the right path of the left tree such that $s(g) \leqq s(v) < s(f)$. (If g does not exist the fused tree is globally biased at v.) Then $s(u) \leqq s(f)$, which when combined with the fact that v is minor ($s(v)+1 < s(u)$) implies that g is minor in the left tree. Thus the neighboring leaf of g, which is the neighboring leaf of v in the fused tree, has rank at least $s(u)-1$. A similar argument applies if r does not exist. Thus in any case the fused tree is globally biased at v. Splitting preserves global bias; thus the tree resulting from the join is globally biased.

THEOREM 8. *The global join algorithm is correct. Furthermore the worst-case time bound given in Theorem* 3 *for local join holds also for global join. Thus a global join runs in* $O(\log(W/(w_- + w_+)))$ *worst-case time, where* W *is the total weight of both trees,* w_- *is the weight of the rightmost item in the left tree, and* w_+ *is the weight of the leftmost item in the right tree.*

Proof. The discussion above verifies correctness; the time bound follows immediately. □

THEOREM 9. *A three-way join of globally biased* 2, *b trees, implemented as two successive global joins, has the same worst-case time bound as given in Theorem* 4 *for three-way local join. Thus a three-way join takes* $O(\log(W/w_i))$ *worst-case time, where* W *is the total weight of the joined tree and* w_i *is the weight of the item inserted between the two trees.*

Proof. Immediate from Theorem 8. □

We can split a globally biased 2, b tree exactly as we did a locally biased 2, b tree, using local joins rather than global joins to build up the left tree and the right tree generated by the split. Below we shall verify that this method results in a globally biased tree, and also get a bound on the running time of the operation. Let u be the current node, v the previous node, q the root of the tree containing the children of u that are to be joined to the left tree, and l the root of the left tree. (See Fig. 4.) The analysis is simplified by the assumption that the subtrees rooted at q and l are not empty and v is not a leaf in the original tree.

We want to verify by induction that after the entire split the resulting left and right trees are globally biased. The induction hypothesis is that the leftmost path descending from l is in the original tree (except possibly for l itself), and that the tree rooted at l is globally biased. The tree with root q is globally biased by construction, and its rightmost path (possibly excluding the node q itself) is a path in the original tree. Since v is not a leaf in the original tree, its left sibling cannot be minor. This implies that $s(q) = s(u)-1$ or $s(q) = s(u)$. We know by the earlier discussion of split that $s(l) \leqq s(v) < s(u)$. Combining these inequalities gives $s(l) \leqq s(q)$. The join of the trees with roots q and l proceeds down the rightmost path from q until reaching the first node t such that t is a leaf or $s(t) \leqq s(l)$. Because of global bias, each node above t is major, and t must also be major unless l is a leaf. Thus the rank decreases by 1 each step down the tree, and either $s(l) = s(t)$ or one of l or t is a leaf. (The important point is that the join does not continue down the left-most path of l as it normally might.) At this point l and t become siblings, and the join terminates (after splitting nodes back up the merged path). We can now verify that our induction holds for the new left tree just created. The leftmost path of this tree is that of the original tree except possibly for the top node (which is new only if $s(q) = s(r)$). We have already

said that the nodes above t must be major. The other nodes on the original rightmost path descending from q also have global bias because they have the same adjacent leaf on the right that they used to have, namely the leftmost leaf of l. This shows that the new left tree is globally biased.

It also follows from this discussion that the number of steps taken by the join is $O(s(q)-s(l))$, because (as mentioned above) the join only traverses down the right path of q until it reaches t. (It does not then propagate down the left path of l.)

To obtain a time bound for split, let us consider the joins that form the left tree. Let $q_1, q_2, \cdots q_k$ be the roots of the successive trees joined into the left tree, let u_i for $1 \leqq i \leqq k$ be the current node when the tree with root q_i is joined with the left tree, let l_i for $1 \leqq i \leqq k$ be root of the left tree after the tree with root q_i is joined (thus $l_1 = q_1$), and finally let x be the root of the tree to be split and y the node at which the split starts. The discussion above implies that $s(q_i) \leqq s(l_i) \leqq s(u_i) \leqq s(u_{i+1}) - 1 \leqq s(q_{i+1}) \leqq s(u_{i+1})$ for $1 \leqq i \leqq k-1$ and that the join of the trees with roots q_{i+1} and l_i takes $O(s(q_{i+1}) - s(l_i))$ time for $1 \leqq i \leqq k-1$. For $2 \leqq i \leqq k-1$, this bound is $O(s(u_{i+1}) - s(u_i))$ by the inequalities above. Consider the case $i = 1$ We have $l_1 = q_1$. If $s(q_1) \geqq s(u_1) - 1$, then $s(q_2) - s(l_1) \leqq s(u_2) - s(u_1) + 1$. If $s(q_1) < s(u_1) - 1$, then q_1 is a minor child of u_1, and the rightmost leaf descendant of q_2 has rank at least $s(u_1) - 1$. Thus the join of trees with roots q_1 and q_2 takes $O(s(q_2) - s(u_1)) = O(s(u_2) - s(u_1))$ time. We conclude that for $1 \leqq i \leqq k-1$, the join of trees with roots l_i and q_{i+1} takes $O(s(u_{i+1}) - s(u_i))$ time. Summing over i, we obtain a bound of $O(s(x) - s(y))$ on the total time to form the left tree. The same argument applies to the right tree. Thus we have the following theorem:

THEOREM 10. *The worst-case time to split a globally biased 2, b tree rooted at x, starting at a node y, is* $O(s(x) - s(y))$. *If y is a leaf containing item i, the time is* $O(\log(W/w_i))$. *If y is an internal node, the time is* $O(\log(W/(w_- + w_+)))$, *where* w_- *and* w_+ *are the weights of the items in the left and right neighboring leaves of y, respectively.*

If we implement each of the operations insert, delete and reweight as a split followed by a global join, we obtain from Theorems 8–10 the following time bounds:

THEOREM 11. *The worst-case time to insert an item i into a globally biased 2, b tree is* $O(\log(W/(w_{i-} + w_{i+})) + \log(W'/w_i))$, *where the various parameters are defined as in Theorem 7. The time to delete an item i is* $O(\log(W/w_i) + \log(W'/(w_{i-} + w_{i+})))$. *The time to change the weight of an item i is* $O(\log(W/w_i) + \log(W'/w_i'))$.

Remark. Based on the time bounds we have derived, the choice between locally and globally biased trees does not seem to be clear-cut but depends upon the application.

We conclude this section by describing a way to build a globally biased 2, b tree of n items in $O(n)$ time. The idea is to form n single-leaf trees, one per item, and join them one-at-a-time, left-to-right, into a large tree, initially empty. To join a single leaf x with the current tree, we use a bottom-up method. We start at the rightmost leaf of the tree and walk up until finding the maximum-rank node, say v, such that $s(v) \leqq s(x)$. If v is the tree root, we create a new root with two children, v and x, and stop. If v is not the tree root, we compare $s(p(v))$ to $s(x)$. If $s(p(v)) = s(x) + 1$, we make x the rightmost child of $p(v)$ and split nodes up the right path as necessary. If $s(p(v)) > s(x) + 1$, we create a new node with two children, v and x, and replace v as a child of its old parent by the new node. (See Fig. 8.)

This bottom-up join method obviously maintains global bias. To obtain a bound on the total time for all $n-1$ joins, we note that, except for $O(1)$ time per join, each step taken by a join either decreases the number of nodes on the rightmost path of the current tree or splits a b-node (a node with b children), thereby reducing the number of b-nodes by one. A single join can only increase the number of nodes on

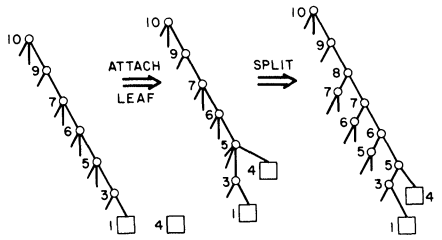

FIG. 8. *Bottom-up join of a single item with a globally biased* 2, 3 *tree.*

the rightmost path by one and can only increase the number of b-nodes by one. It follows that the $n-1$ joins require a total of $O(n)$ time, and we have the following theorem:

THEOREM 12. *Repeated single-node, bottom-up joins will construct a globally biased* 2, *b tree in* $O(n)$ *actual time.*

Note. Theorem 12 does *not* include time corresponding to the credits necessary to establish the credit invariant on the constructed tree (if we are using locally biased trees); the number of credits needed depends upon the relative weight of the items and is not bounded by any function of n. (Consider a tree containing two items with weights 1 and 2^k for k arbitrarily large.)

Remark. Local join and global join as we have described them each consist of a top-down pass (for merging) followed by a bottom-up pass (for splitting). However, if $b \geqq 4$ either form of join can be implemented in a one-pass, purely top-down fashion by preemptively splitting nodes with b or more children during merging.

4. Biased binary trees. In practice, implementations of balanced tree data structures are plagued by a multiplicity of cases, making the resulting code lengthy, opaque and hard to prove correct. In this section we shall describe a class of biased search trees whose update algorithms are relatively easy to program and have a manageable number of cases. We shall only sketch proofs of algorithm correctness and time bounds, since the proofs use exactly the same techniques as in § 3.

A *locally biased binary search tree* is a full binary search tree (every internal node has exactly two children), each of whose nodes x has an integer rank $s(x)$, such that the ranks have the following properties:

(i) If x is a leaf, then $s(x) = \lfloor \lg w(x) \rfloor$.

(ii) If node y has parent x, $s(y) \leqq s(x)$; if y is a leaf, $s(y) \leqq s(x)-1$.

(iii) If node y has grandparent x, then $s(y) \leqq s(x)-1$.

(iv) *Local bias.* A node is *minor* if the rank of its parent is at least two greater than its own rank and *major* otherwise. Let y be a minor node with parent x. If y is the left child of x, either the sibling of y or the left child of that sibling is a leaf of rank $s(x)-1$. If, in addition, x is the right child of its parent and has the same rank as its parent, then either the sibling of x or the right child of that sibling is a leaf of rank $s(x)-1$. A symmetric condition holds if y is the right child of its parent. (See Fig. 9.)

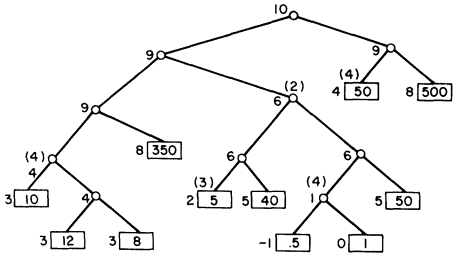

FIG. 9. *A locally biased binary tree.*

Biased binary trees are a binarized version of biased 2, 4 trees; if we take a biased binary tree and condense into a single node all adjacent nodes of the same rank, we obtain a biased 2, 4 tree. (See Fig. 10.) Biased binary trees generalize symmetric binary B-trees [5], which have been described as red-black trees [12]. In the case of equal weights, we obtain the red-black representation of a biased binary tree by calling an edge *red* if parent and child have the same rank and *black* if their ranks differ by one; in this case all nodes are major. If we want to be colorful we can in the general case call an edge *blue* if it joins a minor child with its parent; then we can call biased binary trees *red, black and blue trees.*

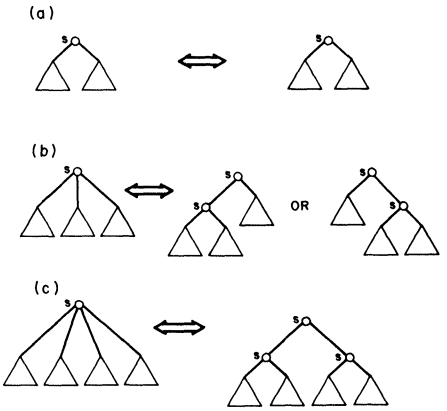

FIG. 10. *Correspondence between biased* 2, 4 *trees and biased binary trees.* (a) 2*-node. Rank of node is s. Roots of subtrees denoted by triangles have ranks less than s.* (b) 3*-node. There are two possible binarized forms.* (c) 4*-node.*

All the theorems presented in § 3 for biased 2, b trees hold for biased binary trees, since we can regard biased binary trees as just a representation of biased 2, 4 trees. In particular biased binary trees have ideal access time for all items. In the remainder of this section we shall give algorithms for joining and splitting biased binary trees. The correspondence with biased 2, 4 trees is somewhat loose because there are two representations of a 3-node.

We begin by presenting an algorithm for join. If x is a node, we denote the left child of x by $l(x)$ and the right child of x by $r(x)$; if x is a leaf, $l(x) = r(x) =$ **null**. By *promoting* a node we mean increasing its rank by one. The algorithm uses two functions, *tilt left* (x) and *tilt right* (x), whose behavior is as follows:

tilt left (x): If both children of internal node x have the same rank as x, promote x and return x. Otherwise if the right but not the left child of x has the same rank as x, perform a single left rotation at node x (see Fig. 11) and return the new parent of x (the old right child of x). In all other cases merely return x.

tilt right (x): Symmetric to *tilt left* (x).

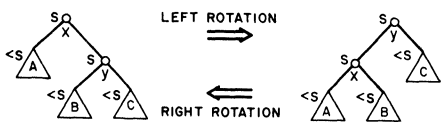

FIG. 11. *A single rotation at node x.*

Remark. If x' is the node returned by *tilt left* (x), then the leaf descendants of x' in the new tree are exactly the leaf descendants of x in the old tree. Also, x' and its right child have different ranks. Achieving the latter condition is the purpose of the *tilt left* operation.

The join algorithm consists of a function *local join* (x, y) that returns the root of the tree formed by joining the trees with roots x and y. The function *local join* (x, y) is defined by the following cases (see Fig. 12):

Case 1. $s(x) = s(y)$, *or* $s(x) > s(y)$ *and* x *is a leaf, or* $s(x) < s(y)$ *and* y *is a leaf.* Create and return a new node with left child x, right child y and rank $\max\{s(x), s(y)\} + 1$.

Case 2. $s(x) > s(y)$ *and* x *is not a leaf.* Replace x by *tilt left* (x). Let z be the right child of x. Define the new right child of x to be *local join* (z, y) and return x.

Case 3. $s(x) < s(y)$ and y is not a leaf. Symmetric to Case 2.

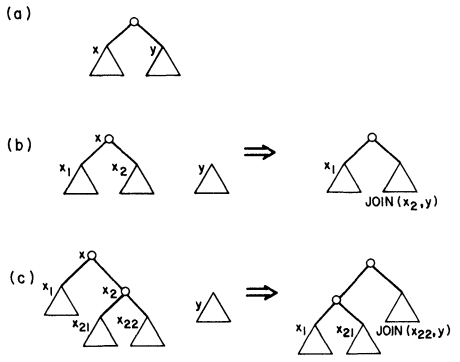

FIG. 12. *Join algorithm for locally biased binary trees.* (a) *Case* 1. *Terminate.* (b) *Case* 2 *with* x *not a right-leaning* 3*-node. Promote* x *if* x *a* 4*-node. Replace right child of* x *by the join of this child and* y. (c) *Case* 2 *with* x *a right-leaning* 3*-node. Rotate left at* x. *Replace right child of root by the join of this child and* y.

We can verify the correctness of this algorithm as follows. The function call *local join* (x, y) returns a tree whose root has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\} + 1$. Furthermore if Case 2 occurs and a promotion takes place (which happens if $s(x) > s(y)$, x is not a leaf, and $s(x) = s(l(x)) = s(r(x))$), then in the next call, which is *local join* $(r(x), y)$, Case 2 also occurs but neither a promotion nor a rotation takes place, by properties (ii) and (iii). Thus *local join* $(r(x), y)$ returns a node of rank $s(r(x))$. Similarly if Case 3 occurs and a promotion takes place, the next call *local join* $(x, l(y))$ returns a node of rank $s(l(y))$. It follows that if the original call *local join* (x, y) returns a node, say x, of rank $\max\{s(x), s(y)\} + 1$, both children of z have rank less than $s(z)$. An inductive case analysis using this fact shows that *local join* is correct.

To establish a time bound for local join we can use the same credit invariant for biased binary trees that we used for biased 2, b trees; a count of credits as in § 3 proves Theorem 2 for biased binary trees. Theorems 3 and 4 for biased binary trees follow immediately. Although we have defined local join recursively, it is easy to give an iterative, purely top-down version. We leave this as an exercise.

We can use the same split algorithm on biased binary trees that we used on biased 2, b trees and the number of cases is much reduced. To split a tree at a leaf x, we initialize v (the current node) to be x, and q and r (the roots of the left and right trees, respectively) to be **null**. Then we repeat the following general step until v is the root of the tree (see Fig. 13), where $p(v)$ denotes the parent of node v:

General step. Case 1. v is the right child of $p(v)$. Let y be the left child of $p(v)$. If $q =$ **null**, replace q by y; if $q \neq$ **null**, replace q by *local join* (y, q). Replace v by $p(v)$. Destroy $r(v)$ if it is not the original leaf x.

Case 2. v is the left child of $p(v)$. Symmetric to Case 1.

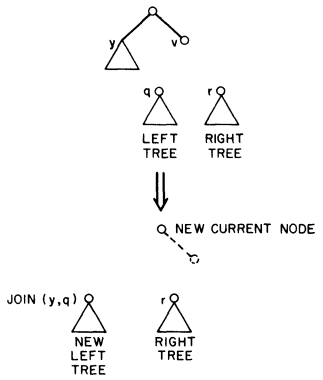

FIG. 13. *One step of split algorithm.*

To split a tree at an internal node x, we proceed as above except that we initialize q to be the left child of x and r to be the right child of x; in this case neither q nor r is ever **null**. The correctness of three-way split and two-way split is immediate. Theorems 5-7 hold for biased binary trees, as we can easily establish using virtually the same proofs as in § 3. Thus the amortized time bounds derived for biased 2, b trees hold for biased trees. Note that when a biased binary tree whose root has rank k is split, the resulting tree(s) all have rank at most $k+1$.

As in § 3, we can improve the worst-case-per-operation behavior of biased binary trees by strengthening the bias property (iv). A *globally biased binary tree* is a full binary search tree having properties (i), (ii), (iii) and the following:

(iv′) *global bias.* If y is a minor node with parent x, then any neighboring leaf of y has rank at least $s(x)-1$.

We can modify the join algorithm so that it produces a globally biased tree if the two input trees are globally biased, although the number of cases increases. As in § 3, the idea is to continue the join until finding a leaf, instead of terminating when encountering two nodes of equal rank. The resulting algorithm *global join* (x, y) consists of the following cases (see Fig. 14):

Case 1. $s(x) \geqq s(y)$ *and* x *is a leaf, or* $s(x) \leqq s(y)$ *and* y *is a leaf.* Create and return a new node with left child x, right child y and rank $\max\{s(x), s(y)\}+1$.

Case 2. $s(x) \geqq s(y)$ *and* x *is not a leaf.* Replace x by *tilt left* (x). Let z be the right child of x. Define the new right child of x to be *global join* (z, y) and return x.

Case 3. $s(x) < s(y)$ *and* y *is not a leaf.* Symmetric to Case 2.

Case 4. $s(x) = s(y)$ *and neither* x *nor* y *is a leaf.* If $s(r(x)) < s(x)$, let $u = x$; otherwise let $u = r(x)$. (In either case $s(r(u)) < s(x)$ and $s(u) = s(x)$.) If $s(l(y)) < s(y)$, let $v = y$; otherwise let $v = l(y)$. Perform *global join* $(r(u), l(v))$; let z be the root of the resulting tree.

Case 4a. $s(z) = s(x)$. Replace $r(u)$ by $l(z)$, $l(v)$ by $r(z)$, $l(z)$ by x, $r(z)$ by y and $s(z)$ by $\max\{s(x), s(y)\}+1$. Return z.

Case 4b. $(s(z) < s(x)$.

Case 4b(i). $u = r(x)$. Replace $r(x)$ by $l(u)$, $l(v)$ by z, $l(u)$ by x, $r(u)$ by y and $s(u)$ by $\max\{s(x), s(y)\}+1$. Return u.

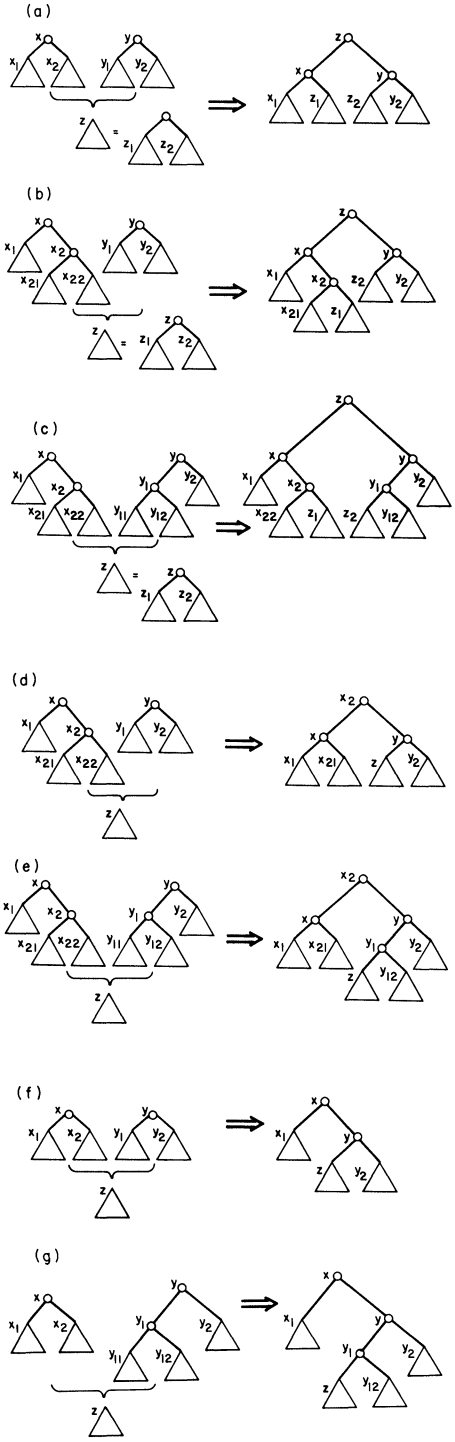

FIG. 14. *Case* 4 *of global join algorithm* (*equal ranks, non terminating*). (a) *Case* 4a ($s(z) = s(x)$) *with* $u = x$, $v = y$. (b) *Case* 4a *with* $u = r(x)$, $v = y$. *Case* 4a *with* $u = x$, $v = l(y)$ *is symmetric.* (c) *Case* 4a *with* $u = r(x)$, $v = l(y)$. (d) *Case* 4b(i) *with* $v = y$. *Case* 4b(ii) *with* $u = x$ *is symmetric.* (e) *Case* 4b(i) *with* $v = l(y)$. *Case* 4b(ii) *with* $u = r(x)$ *is symmetric.* (f) *Case* 4b(iii) *with* $v = y$ *and Case* 4b(v). *Case* 4b(iv) *with* $u = x$ *is symmetric.* (g) *Case* 4b(iii) *with* $v = l(y)$. *Case* 4b(iv) *with* $u = r(x)$ *is symmetric.*

Case 4b(ii). $v = l(y)$. Replace $l(y)$ by $r(v)$, $r(u)$ by z, $l(v)$ by x, $r(v)$ by y and $s(v)$ by $\max\{s(x), s(y)\}+1$. Return v.

Case 4b(iii). $u = x$ and $s(x) = s(l(x))$. Replace $l(v)$ by z, $r(x)$ by y and $s(x)$ by $s(x)+1$. Return x.

Case 4b(iv). $v = y$ and $s(y) = s(r(y))$. Replace $r(u)$ by z, $l(y)$ by x and $s(y)$ by $s(y)+1$. Return y.

Case 4b(v). $u = x$, $s(x) > s(l(x))$, $v = y$ and $s(y) > s(r(y))$. Replace $l(v)$ by z and $r(x)$ by y. Return x.

Remark. Cases 4b(i) and (ii) are nondisjoint, as are Cases 4b(iii) and (iv). If two cases are possible, the choice can be made arbitrarily.

A straightforward but tedious case analysis verifies the correctness of this method. With this implementation of global join and with split implemented using local join, Theorems 8–11 hold for globally biased binary trees. As with local join, global join can be implemented as an iterative, purely top-down method if desired.

Our last result in this section is an algorithm which constructs a globally biased binary tree of n items in $O(n)$ time. We use the same approach as in § 3; namely, we begin with an empty tree and successively join each item into the tree, proceeding left-to-right. To join each new leaf into the tree, we use a bottom-up method. To simplify the joins, we maintain the invariant that nodes down the right path of the tree strictly decrease in rank. To give access to the tree, we maintain a pointer to its rightmost leaf. To join a single leaf y with the tree, we start at the rightmost leaf x and walk up the tree, replacing x by $p(x)$, until x is the tree root or $s(p(x)) > s(y)$. We thus create a new node with left child x, right child y and rank $\max\{s(x), s(y)\}+1$. If x was the old tree root we are finished. If not, we make the new node the right child of the old parent of x. Then we perform a tilt left on this old parent and walk up toward the root, performing a tilt left on each node, until reaching the root or performing a tilt left that does not cause a promotion. (See Fig. 15.)

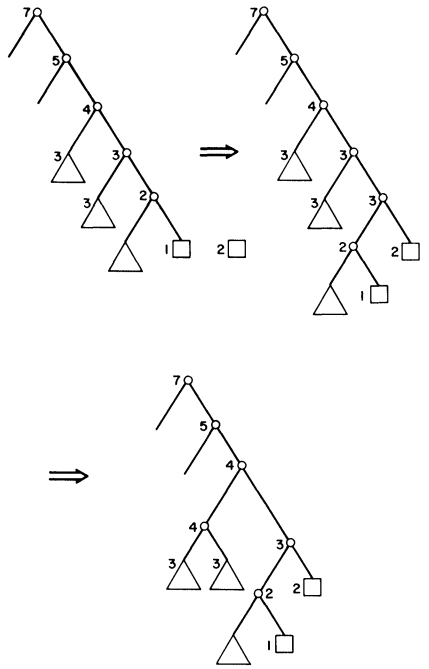

FIG. 15. *Bottom-up join of a single item with a globally biased tree.*

The correctness of this method is obvious; the same argument used in § 3 shows that forming an n-leaf tree by $n-1$ successive bottom-up joins requires $O(n)$ time; that is, Theorem 12 holds for biased binary trees.

There are many alternatives to the specific update algorithms we have presented for biased binary trees. By varying the order in which the basic operations of promotion and rotation are performed, one may obtain a range of different implementations. Guibas and Sedgewick [12] have explored such variants for the equal-weight case; we leave it as an exercise generalizing their algorithms to the biased case.

5. Summary, applications and related work. We have presented two classes of biased search trees. Each has a locally biased and a globally biased version. All our search trees have ideal worst-case access times for all items. Locally biased search trees have fast amortized update times, as given in Theorems 2 and 4–7. Globally biased search trees have fast worst-case update times, as given in Theorems 8–11.

Biased search trees have a number of applications, three of which we list below. This list is meant to be illustrative, not exhaustive.

1. *Dictionaries with access weights.* The most obvious application of a biased search tree is to store a table, such as a name table in a compiler, a natural-language dictionary, or a telephone directory. If we have a priori estimates of the access frequencies, we can use these as weights. Alternatively, we can keep a frequency count for each item and use this as its weight, increasing the count by one each time we access the item. With this method the time to rebalance after increasing a count is proportional to the access time, and the time for an insertion is also proportional to the access time, since an item has an initial count of one.

We can also include weights for unsuccessful searches in this scheme: we assign to the leaf containing item i a weight equal to our estimate of the frequency of successful searches for i plus the frequency of unsuccessful searches for items between item $i-1$ and item i; any such unsuccessful search will terminate at the leaf containing item i (or at an ancestor of that leaf if double keys are used; see Appendix B). When a new item is inserted between items $i-1$ and i, we must somehow apportion the weight for unsuccessful searches between $i-1$ and i to the two new intervals created by the insertion.

By perturbing the weights, we can guarantee an access time of $O(\min\{\log n, \log(W/w_i)\})$ for every item i, thus obtaining the behavior of balanced and biased search trees simultaneously. To do this, we assign to every item i a weight of $1/n + w_i/W$. It is not necessary to update the weights of all the items every time n changes; it suffices to update all the weights whenever n changes by a factor of two. When amortized over a sequence of insertions and the deletions, the time for updating weights is $O(1)$ per insertion or deletion.

2. *Tries and multidimensional search trees.* Suppose the items to be stored are k-dimensional vectors (or equivalently lists of length k) ordered lexicographically, and that comparing the corresponding components of two items takes $O(1)$ time. We can use biased search trees to store collections of such vectors so that access, insertion, deletion, join and split take $O(\log n + k)$ time, either in the amortized sense if we use locally biased trees or in the worst-case sense if we use globally biased trees. See Mehlhorn [26] and Güting and Kriegel [13]. The idea extends to allow the vectors to have weights measuring access frequencies [14] and to allow partial-match queries [23].

3. *Dynamic trees.* A number of network optimization algorithms require a data structure to represent a collection of rooted trees on which we can perform the following two update operations:

link (v, w): If v is the root of one tree and w is a node in another tree, combine the trees containing v and w by adding an edge joining v and w.

cut (v, w): If there is an edge joining v and w, delete it, thereby breaking the tree containing v and w into two trees, one containing v and one containing w.

Using biased search trees we have been able to develop a data structure for such *dynamic trees* in which link or cut operations, as well as other operations of interest, take $O(\log n)$ time per operation [28]. We divide each dynamic tree into a collection of disjoint paths and represent each path by a biased search tree. This data structure leads to improved running times for several network optimization algorithms. For example, we are able to find a maximum network flow in an n-vertex, m-edge graph in $O(nm \log n)$ time.

Recently Feigenbaum and Tarjan [9] have developed two additional types of biased search trees. These are a biased form of B-trees and a biased form of weight-balanced trees. The biased B-trees have $O(\log_b (W/w_i))$ worst-case access time, where b is the maximum number of children per node, and have correspondingly efficient update times. They exist in both locally biased and globally biased forms. Kriegel and Vaishnavi [22] have proposed a data structure with similar access times but less favorable update times.

In another related development Sleator and Tarjan [30], [31] have devised "self-adjusting" binary search trees with amortized access and update times similar to those of biased search trees. The advantage of self-adjusting trees is their simplicity, since there is no balance condition to maintain. The disadvantages of self-adjusting trees are that they must be adjusted frequently (even during accesses), and the time bound for access is amortized rather than worst-case.

Appendix A. Tree terminology. A *rooted tree* is either empty or consists of a single node r, called the *root*, and a set of zero or more rooted trees $T_1, \cdots, T_k$ that are node-disjoint and do not contain r. The roots $r_1, \cdots, r_k$ of $T_1, \cdots, T_k$ are the *children* of r; r is the *parent* of $r_1, \cdots, r_k$. A node without children is a *leaf*; a node with at least one child is an *internal node*. Two nodes with the same parent are *siblings*. The *degree* of a node is the number of its children.

A *path* of *length* $l-1$ in a tree is a sequence of nodes $v_1, v_2, \cdots, v_l$ such that v_{i+1} is a child of v_i for $1 \leqq i < l$. The path goes from v_1 *down* to v_l and from v_l *up* to v_1. A node v is an *ancestor* of a node w and w is a descendant of v if there is a path from v down to w. (Every node is an ancestor and a descendant of itself.) If w is a leaf, it is a *leaf descendant* of v. Two nodes are *unrelated* if neither is an ancestor of the other.

Let v be any node in a tree T. There is a unique path from the root of T down to v; the length of this path is the *depth* of v. The *height* of v is the length of the longest path from v down to a leaf. The *subtree rooted at* v is the tree whose root is v containing all the descendants of v. The *nearest common ancestor* of two nodes v and w is the node of maximum depth that is an ancestor of both v and w.

An *ordered tree* is a rooted tree such that the children of every node v are totally ordered. A child x of v is to the *left* of another child y, and y is to the *right* of x, if x occurs first in the ordering of the children of v. If no sibling occurs between x and y in the ordering, x is the *left* sibling of y, y is the *right sibling* of x, and x and y are *neighboring* siblings. The first child of a node is its *left* (or *leftmost*) *child* and the last child is its *right* (or *rightmost*) *child*.

The ordering of children imposes an order on any two unrelated nodes v and w; v is to the *left* of w, and w is to the *right* of v, if there are siblings v' and w' such that v' is to the left of w', v' is an ancestor of v, and w' is an ancestor of w. This relation totally orders the leaf descendants of any node x; the *left* (or *leftmost*) *leaf descendant* of x is the first leaf in the ordering and the *right* (or *rightmost*) *leaf descendant* of x is the last. The *left path* from x is the path from x down to its leftmost leaf descendant; the *right path* from x is the path from x down to its rightmost leaf descendant. The *left neighboring leaf* of x is the rightmost leaf (if any) to the left of the leftmost leaf descendant of x; the *right neighboring leaf* of x is the leftmost leaf (if any) to the right of the rightmost leaf descendant of x.

Appendix B. Keys in a search tree. Let S be a totally ordered set. A *search tree* for S is an ordered tree containing the items of S in its leaves, one item per leaf, in left-to-right order. In order to use the search tree to access S, we must store auxiliary items, called *keys*, in the internal nodes. We shall consider two possibilities. The first is the *single key* representation: if x is an internal node with k children, x contains $k-1$ keys, called *left keys*, one for each child y of x except the rightmost. The key for y is the largest item in the subtree rooted at y. The second is the *double key* representation: in addition to left keys, every internal node x contains a *right key* for each of its children y except the leftmost. The key for y is the smallest item in the subtree rooted at y. Every item in the tree except the largest occurs exactly once as a left key; every key except the smallest occurs exactly once as a right key. (See Fig. 16.)

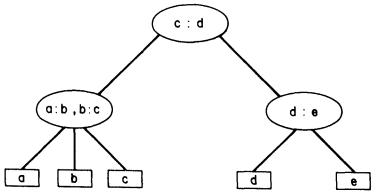

FIG. 16. *Keys in a search tree. Items are letters in alphabetical order. Left keys appear before colons, right keys after. A pair of keys $k_1 : k_2$ consisting of a left key k_1 and a right key k_2 corresponds to an open interval of items missing from the tree.*

Left keys (or right keys) suffice for searching from the root of a given item i. We initialize the current node x to be the root and repeat the following step until x is a leaf. Then either x contains i or i is not in the tree.

Search step. Select the smallest left key in x no less than i. Replace x by the child y of x corresponding to this key.

Using both left and right keys expedites unsuccessful searches. If i is an item, let i^- be the last item before i and i^+ the first item after i. We define the *handle* of i to be the leaf containing i if i is in the search tree and the nearest common ancestor of the leaves containing i^- and i^+ if not; in this case the handle contains i^- as a left key and i^+ as a right key. A search for i can stop at the handle of i: we terminate the search when the current node x is a leaf or i lies strictly between a left key in x and the next larger right key. To deal with the case of an item smaller than the smallest item in the tree or larger than the largest, we maintain a header for the tree containing its smallest and largest items; then unsuccessful searches for items outside the range of the tree take $O(1)$ time.

Updating a search tree generally requires a sequence of local rebalancing steps, each of which changes the structure of the tree. For the binary search trees, considered

in § 4, we need two symmetric rebalancing steps: a left single rotation and a right single rotation. (See Fig. 17.) A single rotation takes $O(1)$ time.

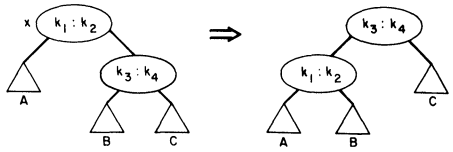

FIG. 17. *Updating of keys during a left single rotation at node x. (Right single rotation is symmetric.)*

For the search trees whose nodes can have more than two children, considered in § 3, we also need two rebalancing operations: a *split*, which splits an internal node into two neighboring siblings, and its inverse, a *fuse*, which combines two neighboring siblings into one. (See Fig. 18.) Either a split or a fuse requires $O(1)$ time, assuming a fixed upper bound on the maximum number of children of a node.

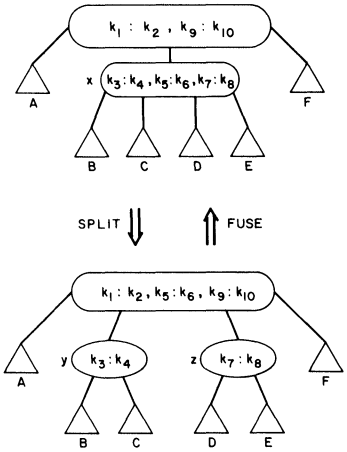

FIG. 18. *Splitting a node x. A left key and a right key move up. Inverse operation is fusing nodes y and z.*

REFERENCES

[1] N. ABRAMSON, *Information Theory and Coding*, McGraw-Hill, New York, 1963.

[2] G. M. ADELSON-VELSKII AND Y. M. LANDIS, *An algorithm for the organization of information*, Soviet Math. Dokl., 3 (1962), pp. 1259-1262.

[3] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[4] J. L. BAER, *Weight-balanced trees*, Proc. AFIPS Nat. Comp. Conf., 44 (1975), pp. 467-472.

[5] R. BAYER, *Symmetric binary B-trees: data structure and maintenance algorithms*, Acta Inform., 1 (1972), pp. 290-306.

[6] R. BAYER AND E. M. MCCREIGHT, *Organization and maintenance of large ordered indexes*, Acta Inform., 1 (1972), pp. 173-189.

[7] S. W. BENT, D. D. SLEATOR AND R. E. TARJAN, *Biased 2-3 trees*, Proc. Twenty-First Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 248-254.

[8] S. W. BENT, *Dynamic weighted data structures*, Ph.D. thesis, Computer Science Dept., Stanford Univ., Stanford, CA, 1982.

[9] J. FEIGENBAUM AND R. E. TARJAN, *Two new kinds of biased search trees*, Bell System Tech. J., 62 (1983), pp. 3139-3158.

[10] M. L. FREDMAN, *Two applications of a probabilistic search technique: sorting X + Y and building balanced search trees*, Proc. Seventh ACM Symposium on Theory of Computing, 1975, pp. 240-244.

[11] A. M. GARSIA AND M. L. WACHS, *A new algorithm for minimal binary search trees*, this Journal, 6 (1977), pp. 622-642.

[12] L. G. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, Proc. Nineteenth Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.
[13] H. GÜTING AND H. P. KRIEGEL, *Multidimensional B-tree: an efficient dynamic file structure for exact match queries*, Informatik-Fachberichte 33, Springer, Berlin 1980, pp. 375–388.
[14] ———, *Dynamic k-dimensional multiway search under time-varying access frequencies*, Lecture Notes in Computer Science 104, Springer, Berlin, 1981, pp. 135–145.
[15] T. C. HU, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.
[16] T. C. HU, D. J. KLEITMAN AND J. K. TAMAKI, *Binary search trees optimum under various criteria*, SIAM J. Appl. Math., 37 (1979), pp. 246–256.
[17] T. C. HU AND A. C. TUCKER, *Optimal computer-search trees and variable-length alphabetic codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.
[18] D. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.
[19] D. E. KNUTH, *Optimum binary search trees*, Acta Inform., 1 (1971), pp. 14–25.
[20] ———, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1975.
[21] J. F. KORSCH, *Greedy binary search trees are nearly optimal*, Inform. Proc. Letters, 13 (1981), pp. 16–19.
[22] H. P. KRIEGEL AND V. K. VAISHNAVI, *Weighted multidimensional B-trees used as nearly optimal dynamic dictionaries*, Mathematical Foundations of Computer Science, Czechoslavakia 1981.
[23] ———, *A nearly optimal dynamic tree structure for partial-match queries with time-varying frequencies*, Proc. CISS, 1981.
[24] K. MEHLHORN, *Nearly optimal binary search trees*, Acta Inform., 5 (1975), pp. 287–295.
[25] ———, *Arbitrary weight changes in dynamic trees*, Bericht 78/04, Angewandte Mathematik und Informatik, Universitat des Saarlandes, 1978.
[26] ———, *Dynamic binary search*, this Journal, 8 (1979), pp. 175–198.
[27] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, this Journal, 2 (1973), pp. 33–43.
[28] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391; see also Proc. Thirteenth Annual ACM Symposium on Theory of Computing, 1981, pp. 114–122.
[29] ———, *Self-adjusting binary trees*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 235–245.
[30] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., to appear.
[31] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.
[32] K. UNTERAUER, *Dynamic weighted binary search trees*, Acta Inform., 11 (1979), pp. 341–362.
[33] F. F. YAO, *Efficient dynamic programming using quadrangle inequalities*, Proc. Twelfth Annual ACM Symposium on Theory of Computing, 1980, pp. 429–435.