COS 423

of the move-to-front list update rule [9, 30] and the development of a self-adjusting form of heap [31]. Some of our results on self-adjusting heaps and search trees appeared in preliminary form in a conference paper [28]. A survey paper by the second author examines a variety of amortized complexity results [35].

2. *Splay Trees*

We introduce splay trees by way of a specific application. Consider the problem of performing a sequence of access operations on a set of items selected from a totally ordered universe. Each item may have some associated information. The input to each operation is an item; the output of the operation is an indication of whether the item is in the set, along with the associated information if it is. One way to solve this problem is to represent the set by a *binary search tree*. This is a binary tree containing the items of the set, one item per node, with the items arranged in *symmetric order*: If x is a node containing an item i, the left subtree of x contains only items less than i and the right subtree only items greater than i. The *symmetric-order position* of an item is one plus the number of items preceding it in symmetric order in the tree.

The "search" in "binary search tree" refers to the ability to access any item in the tree by searching down from the root, branching left or right at each step according to whether the item to be found is less than or greater than the item in the current node, and stopping when the node containing the item is reached. Such a search takes $\Theta(d)$ time, where d is the depth of the node containing the accessed item.

If accessing items is the only operation required, then there are better solutions than binary search trees, e.g., hashing. However, as we shall see in Section 3, binary search trees also support several useful update operations. Furthermore, we can extend binary search trees to support accesses by symmetric-order position. To do this, we store in each node the number of descendants of the node. Alternatively, we can store in each node the number of nodes in its left subtree and store in a tree header the number of nodes in the entire tree.

When referring to a binary search tree formally, as in a computer program, we shall generally denote the tree by a pointer to its root; a pointer to the null node denotes an empty tree. When analyzing operations on binary search trees, we shall use n to denote the number of nodes and m to denote the total number of operations.

Suppose we wish to carry out a sequence of access operations on a binary search tree. For the total access time to be small, frequently accessed items should be near the root of the tree often. Our goal is to devise a simple way of restructuring the tree after each access that moves the accessed item closer to the root, on the plausible assumption that this item is likely to be accessed again soon. As an $O(1)$-time restructuring primitive, we can use *rotation*, which preserves the symmetric order of the tree. (See Figure 1.)

Allen and Munro [4] and Bitner [10] proposed two restructuring heuristics (see Figure 2):

Single rotation. After accessing an item i in a node x, rotate the edge joining x to its parent (unless x is the root).

Move to root. After accessing an item i in a node x, rotate the edge joining x to its parent, and repeat this step until x is the root.

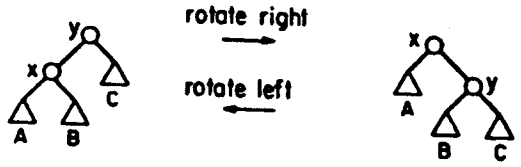


FIG. 1. Rotation of the edge joining nodes x and y. Triangles denote subtrees. The tree shown can be a subtree of a larger tree.

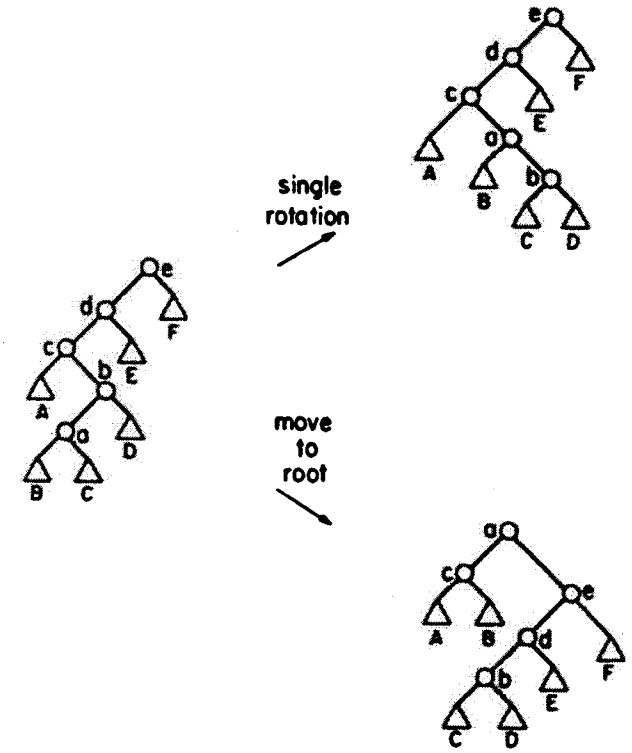


FIG. 2. Restructuring heuristics. The node accessed is a.

Unfortunately, neither of these heuristics is efficient in an amortized sense: for each, there are arbitrarily long access sequences such that the time per access is $O(n)$ [4]. Allen and Munro did show that the move-to-root heuristic has an asymptotic average access time that is within a constant factor of minimum, but only under the assumption that the access probabilities of the various items are fixed and the accesses are independent. We seek heuristics that have much stronger properties.

Our restructuring heuristic, called *splaying*, is similar to move-to-root in that it does rotations bottom-up along the access path and moves the accessed item all the way to the root. But it differs in that it does the rotations in pairs, in an order that depends on the structure of the access path. To splay a tree at a node x, we repeat the following *splaying step* until x is the root of the tree (see Figure 3):

Splaying Step

Case 1 (zig). If $p(x)$, the parent of x, is the tree root, rotate the edge joining x with $p(x)$. (This case is terminal.)

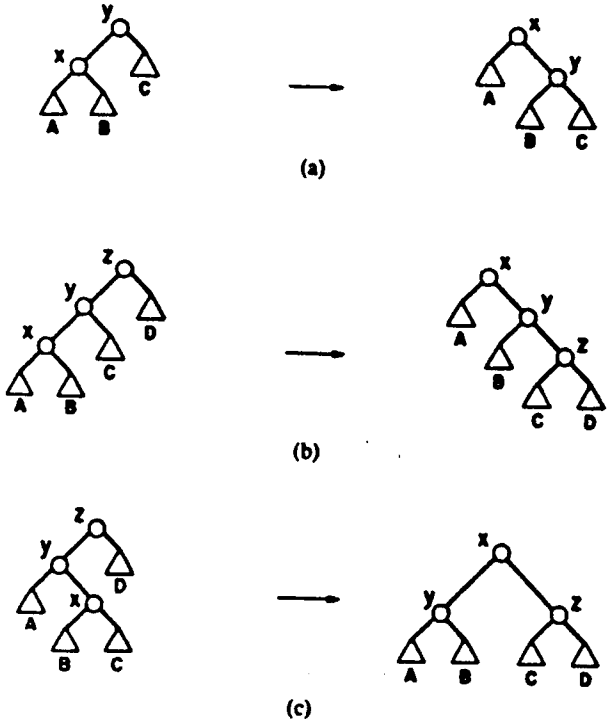


FIG. 3. A splaying step. The node accessed is x. Each case has a symmetric variant (not shown). (a) Zig: terminating single rotation. (b) Zig-zig: two single rotations. (c) Zig-zag: double rotation.

Case 2 (zig-zig). If $p(x)$ is not the root and x and $p(x)$ are both left or both right children, rotate the edge joining $p(x)$ with its grandparent $g(x)$ and then rotate the edge joining x with $p(x)$.

Case 3 (zig-zag). If $p(x)$ is not the root and x is a left child and $p(x)$ a right child, or vice-versa, rotate the edge joining x with $p(x)$ and then rotate the edge joining x with the new $p(x)$.

Splaying at a node x of depth d takes $\Theta(d)$ time, that is, time proportional to the time to access the item in x. Splaying not only moves x to the root, but roughly halves the depth of every node along the access path. (See Figures 4 and 5.) This halving effect makes splaying efficient and is a property not shared by other, simpler heuristics, such as move to root. Producing this effect seems to require dealing with the zig-zig and zig-zag cases differently.

We shall analyze the amortized complexity of splaying by using a *potential function* [31, 35] to carry out the amortization. The idea is to assign to each possible configuration of the data structure a real number called its *potential* and to define the *amortized time* a of an operation by $a = t + \Phi' - \Phi$, where t is the actual time of the operation, Φ is the potential before the operation, and Φ' is the potential after the operation. With this definition, we can estimate the total time of a sequence of m operations by

$$\sum_{j=1}^{m} t_j = \sum_{j=1}^{m} (a_j + \Phi_{j-1} - \Phi_j) = \sum_{j=1}^{m} a_j + \Phi_0 - \Phi_m,$$

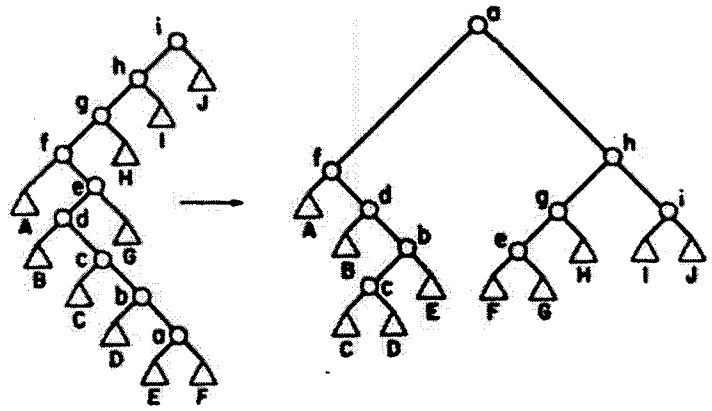
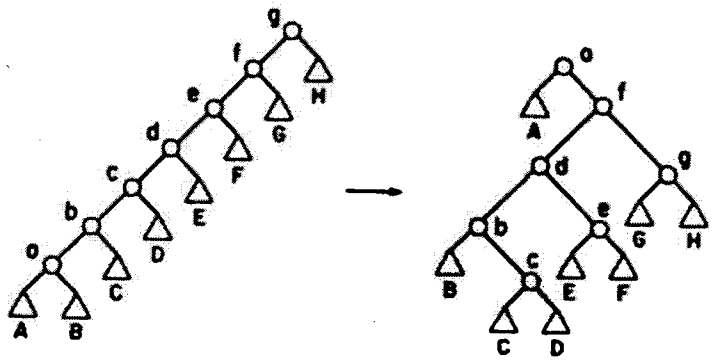
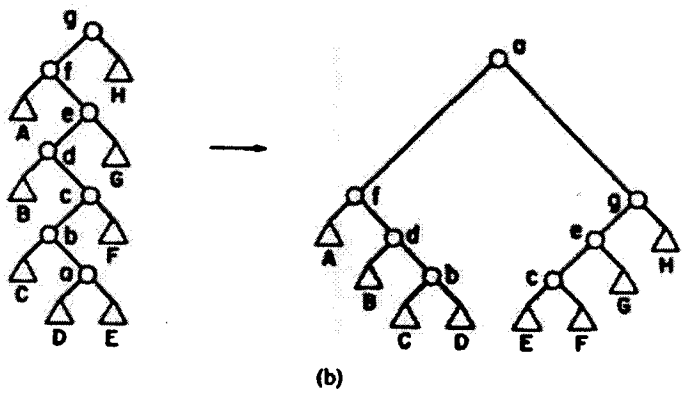


FIG. 4. Splaying at node *a*.



(a)



(b)

FIG. 5. Extreme cases of splaying. (a) All zig–zig steps. (b) All zig–zag steps.

where t_j and a_j are the actual and amortized times of operation j, Φ_0 is the initial potential, and Φ_j for $j \geq 1$ is the potential after operation j. That is, the total actual time equals the total amortized time plus the net decrease in potential from the initial to the final configuration. If the final potential is no less than the initial potential, then the total amortized time is an upper bound on the total actual time.

To define the potential of a splay tree, we assume that each item i has a positive *weight* $w(i)$, whose value is arbitrary but fixed. We define the *size* $s(x)$ of a node x in the tree to be the sum of the individual weights of all items in the subtree rooted at x. We define the *rank* $r(x)$ of node x to be $\log s(x)$.[1] Finally, we define the potential of the tree to be the sum of the ranks of all its nodes. As a measure of the running time of a splaying operation, we use the number of rotations done, unless there are no rotations, in which case we charge one for the splaying.

LEMMA 1 (ACCESS LEMMA). *The amortized time to splay a tree with root t at a node x is at most $3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$.*

PROOF. If there are no rotations, the bound is immediate. Thus suppose there is at least one rotation. Consider any splaying step. Let s and s', r and r' denote the size and rank functions just before and just after the step, respectively. We show that the amortized time for the step is at most $3(r'(x) - r(x)) + 1$ in case 1 and at most $3(r'(x) - r(x))$ in case 2 or case 3. Let y be the parent of x and z be the parent of y (if it exists) before the step.

Case 1. One rotation is done, so the amortized time of the step is

$$\begin{aligned} & 1 + r'(x) + r'(y) - r(x) - r(y) && \text{since only } x \text{ and } y \text{ can change rank} \\ & \le 1 + r'(x) - r(x) && \text{since } r(y) \ge r'(y) \\ & \le 1 + 3(r'(x) - r(x)) && \text{since } r'(x) \ge r(x). \end{aligned}$$

Case 2. Two rotations are done, so the amortized time of the step is

$$\begin{aligned} & 2 + r'(x) + r'(y) + r'(z) \\ & \quad - r(x) - r(y) - r(z) && \text{since only } x, y, \text{ and } z \text{ can change rank} \\ & = 2 + r'(y) + r'(z) - r(x) - r(y) && \text{since } r'(x) = r(z) \\ & \le 2 + r'(x) + r'(z) - 2r(x) && \text{since } r'(x) \ge r'(y) \text{ and } r(y) \ge r(x). \end{aligned}$$

We claim that this last sum is at most $3(r'(x) - r(x))$, that is, that $2r'(x) - r(x) - r'(z) \ge 2$. The convexity of the log function implies that $\log x + \log y$ for $x, y > 0$, $x + y \le 1$ is maximized at value -2 when $x = y = \frac{1}{2}$. It follows that $r(x) + r'(z) - 2r'(x) = \log(s(x)/s'(x)) + \log(s'(z)/s'(x)) \le -2$, since $s(x) + s'(z) \le s'(x)$. Thus the claim is true, and the amortized time in case 2 is at most $3(r'(x) - r(x))$.

Case 3. The amortized time of the step is

$$\begin{aligned} & 2 + r'(x) + r'(y) + r'(z) \\ & \quad - r(x) - r(y) - r(z) \\ & \quad \le 2 + r'(y) + r'(z) - 2r(x) && \text{since } r'(x) = r(z) \text{ and } r(x) \le r(y). \end{aligned}$$

We claim that this last sum is at most $2(r'(x) - r(x))$, that is, that $2r'(x) - r'(y) - r'(z) \ge 2$. This follows as in case 2 from the inequality $s'(y) + s'(z) \le s'(x)$. Thus the amortized time in case 3 is at most $2(r'(x) - r(x)) \le 3(r'(x) - r(x))$.

The lemma follows by summing the amortized time estimates for all the splaying steps, since the sum telescopes to yield an estimate of at most $3(r'(x) - r(x)) + 1 = 3(r(t) - r(x)) + 1$, where r and r' are the rank functions before and after the entire splaying operation, respectively. □

[1] Throughout this paper we use binary logarithms.

The analysis in Lemma 1 shows that the zig–zig case is the expensive case of a splaying step. The analysis differs from our original analysis [28] in that the definition of rank uses the continuous instead of the discrete logarithm. This gives us a bound that is tighter by a factor of two. The idea of tightening the analysis in this way was also discovered independently by Huddleston [17].

The weights of the items are parameters of the analysis, not of the algorithm: Lemma 1 holds for *any* assignment of positive weights to items. By choosing weights cleverly, we can obtain surprisingly strong results from Lemma 1. We shall give four examples. Consider a sequence of m accesses on an n-node splay tree. In analyzing the running time of such a sequence, it is useful to note that if the weights of all items remain fixed, then the net decrease in potential over the sequence is at most $\sum_{i=1}^{n} \log(W/w(i))$, where $W = \sum_{i=1}^{n} w(i)$, since the size of the node containing item i is at most W and at least $w(i)$.

THEOREM 1 (BALANCE THEOREM). *The total access time is* $O((m + n)\log n + m)$.

PROOF. Assign a weight of $1/n$ to each item. Then $W = 1$, the amortized access time is at most $3 \log n + 1$ for any item, and the net potential drop over the sequence is at most $n \log n$. The theorem follows. □

For any item i, let $q(i)$ be the access frequency of item i, that is, the total number of times i is accessed.

THEOREM 2 (STATIC OPTIMALITY THEOREM). *If every item is accessed at least once, then the total access time is*

$$O\left(m + \sum_{i=1}^{n} q(i)\log\left(\frac{m}{q(i)}\right)\right).$$

PROOF. Assign a weight of $q(i)/m$ to item i. Then $W = 1$, the amortized access time of item i is $O(\log(m/q(i)))$, and the net potential drop over the sequence is at most $\sum_{i=1}^{n} \log(m/q(i))$. The theorem follows. □

Assume that the items are numbered from 1 through n in symmetric order. Let the sequence of accessed items be $i_1, i_2, \ldots, i_m$.

THEOREM 3 (STATIC FINGER THEOREM). *If f is any fixed item, the total access time is* $O(n \log n + m + \sum_{j=1}^{m} \log(|i_j - f| + 1))$.

PROOF. Assign a weight of $1/(|i - f| + 1)^2$ to item i. Then $W \le 2\sum_{k=1}^{n} 1/k^2 = O(1)$, the amortized time of the jth access is $O(\log(|i_j - f| + 1))$, and the net potential drop over the sequence is $O(n \log n)$, since the weight of any item is at least $1/n^2$. The theorem follows. □

We can obtain another interesting result by changing the item weights as the accesses take place. Number the accesses from 1 to m in the order they occur. For any access j, let $t(j)$ be the number of different items accessed before access j since the last access of item i_j, or since the beginning of the sequence if j is the first of item i_j. (Note that $t(j) + i$ is the position of item i_j in a linear list maintained by the move-to-front heuristic [30] and initialized in order of first access.)

THEOREM 4 (WORKING SET THEOREM). *The total access time is* $O(n \log n + m + \sum_{j=1}^{m} \log(t(j) + 1))$.

PROOF. Assign the weights $1, 1/4, 1/9, \ldots, 1/n^2$ to the items in order by first access. (The item accessed earliest gets the largest weight; any items never accessed get the smallest weights.) As each access occurs, redefine the weights as follows. Suppose the weight of item i_j during access j is $1/k^2$. After access j, assign weight 1 to i_j, and for each item i having a weight of $1/(k')^2$ with $k' < k$, assign weight $1/(k'+1)^2$ to i. This reassignment permutes the weights $1, 1/4, 1/9, \ldots, 1/n^2$ among the items. Furthermore, it guarantees that, during access j, the weight of item i_j will be $1/(t(j)+1)^2$. We have $W = \sum_{k=1}^{n} 1/k^2 = O(1)$, so the amortized time of access j is $O(\log(t(j)+1))$. The weight reassignment after an access increases the weight of the item in the root (because splaying at the accessed item moves it to the root) and only decreases the weights of other items in the tree. The size of the root is unchanged, but the sizes of other nodes can decrease. Thus the weight reassignment can only decrease the potential, and the amortized time for weight reassignment is either zero or negative. The net potential drop over the sequence is $O(n \log n)$. The theorem follows. □

Let us interpret the meaning of these theorems. The balance theorem states that on a sufficiently long sequence of accesses a splay tree is as efficient as any form of uniformly balanced tree. The static optimality theorem implies that a splay tree is as efficient as any fixed search tree, including the optimum tree for the given access sequence, since by a standard theorem of information theory [1] the total access time for any fixed tree is $\Omega(m + \sum_{i=1}^{n} q(i)\log(m/q(i)))$. The static finger theorem states that splay trees support accesses in the vicinity of a fixed finger with the same efficiency as finger search trees. The working set theorem states that the time to access an item can be estimated as the logarithm of one plus the number of different items accessed since the given item was last accessed. That is, the most recently accessed items, which can be thought of as forming the "working set," are the easiest to access. All these results are to within a constant factor.

Splay trees have all these behaviors automatically; the restructuring heuristic is blind to the properties of the access sequence and to the global structure of the tree. Indeed, splay trees have all these behaviors simultaneously; at the cost of a constant factor we can combine all four theorems into one.

THEOREM 5 (UNIFIED THEOREM). *The total time of a sequence of m accesses on an n-node splay tree is*

$$O\left(n \log n + m + \sum_{j=1}^{m} \log \min\left\{\frac{m}{q(i_j)}, |i_j - f| + 1, t(j) + 1\right\}\right),$$

where f is any fixed item.

PROOF. Assign to each item a weight equal to the sum of the weights assigned to it in Theorems 2–4 and combine the proofs of these theorems. □

Remark. Since $|i_j - f| < n$, Theorem 5 implies Theorem 1 as well as Theorems 2–4. If each item is accessed at least once, the additive term $n \log n$ in the bound of Theorem 5 can be dropped.

3. *Update Operations on Splay Trees*

Using splaying, we can implement all the standard update operations on binary search trees. We consider the following operations:

access(i, t): If item i is in tree t, return a pointer to its location; otherwise, return a pointer to the null node.

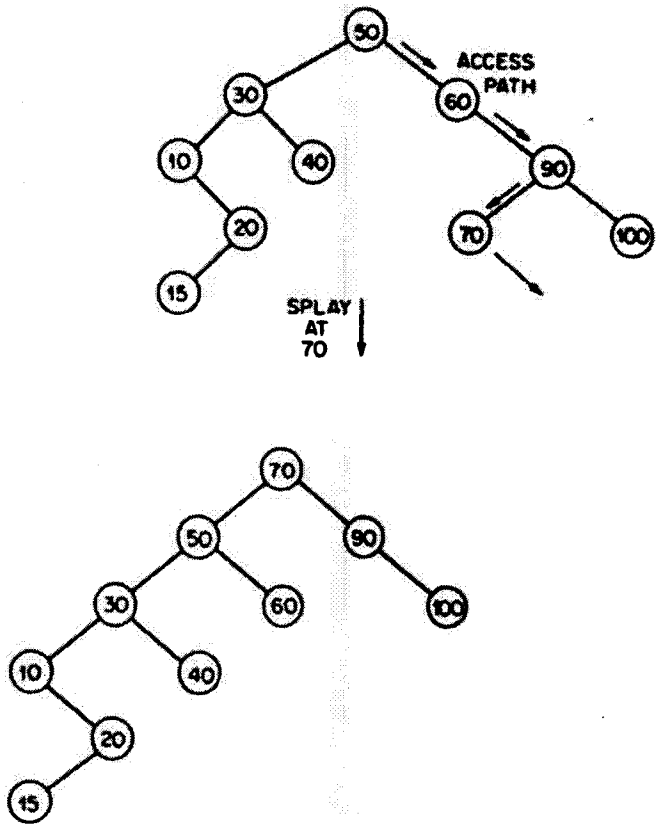


FIG. 6. Splaying after an unsuccessful access of item 80.

insert(i, t): Insert item i in tree t, assuming that it is not there already.

delete(i, t): Delete item i from tree t, assuming that it is present.

join(t_1, t_2): Combine trees t_1 and t_2 into a single tree containing all items from both trees and return the resulting tree. This operation assumes that all items in t_1 are less than all those in t_2 and destroys both t_1 and t_2.

split(i, t): Construct and return two trees t_1 and t_2, where t_1 contains all items in t less than or equal to i, and t_2 contains all items in t greater than i. This operation destroys t.

We can carry out these operations on splay trees as follows. To perform *access*(i, t), we search down from the root of t, looking for i. If the search reaches a node x containing i, we complete the access by splaying at x and returning a pointer to x. If the search reaches the null node, indicating that i is not in the tree, we complete the access by splaying at the last nonnull node reached during the search (the node from which the search ran off the bottom of the tree) and returning a pointer to null. If the tree is empty, we omit the splaying operation. (See Figure 6.)

Splaying's effect of moving a designated node to the root considerably simplifies the updating of splay trees. It is convenient to implement *insert* and *delete* using *join* and *split*. To carry out *join*(t_1, t_2), we begin by accessing the largest item, say

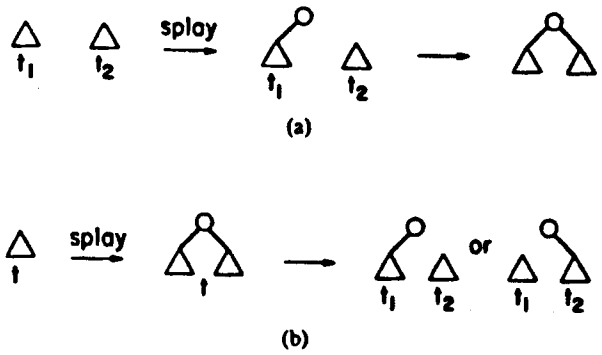


FIG. 7. Implementation of *join* and *split*: (a) *join*(t_1, t_2). (b) *split*(i, t).

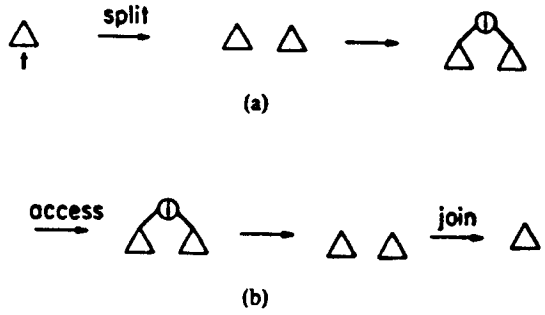


FIG. 8. Implementation of insertion and deletion using *join* and *split*: (a) *insert*(i, t). (b) *delete*(i, t).

i, in t_1. After the access, the root of t_1 contains i and thus has a null right child. We complete the join by making t_2 the right subtree of this root and returning the resulting tree. To carry out *split*(i, t), we perform *access*(i, t) and then return the two trees formed by breaking either the left link or the right link from the new root of t, depending on whether the root contains an item greater than i or not greater than i. (See Figure 7.) In both *join* and *split* we must deal specially with the case of an empty input tree (or trees).

To carry out *insert*(i, t), we perform *split*(i, t) and then replace t by a tree consisting of a new root node containing i, whose left and right subtrees are the trees t_1 and t_2 returned by the split. To carry out *delete*(i, t), we perform *access*(i, t) and then replace t by the join of its left and right subtrees. (See Figure 8.)

There are alternative implementations of *insert* and *delete* that have slightly better amortized time bounds. To carry out *insert*(i, t), we search for i, then replace the pointer to null reached during the search by a pointer to a new node containing i, and finally splay the tree at the new node. To carry out *delete*(i, t), we search for the node containing i. Let this node be x and let its parent be y. We replace x as a child of y by the join of the left and right subtrees of x, and then we splay at y. (See Figure 9.)

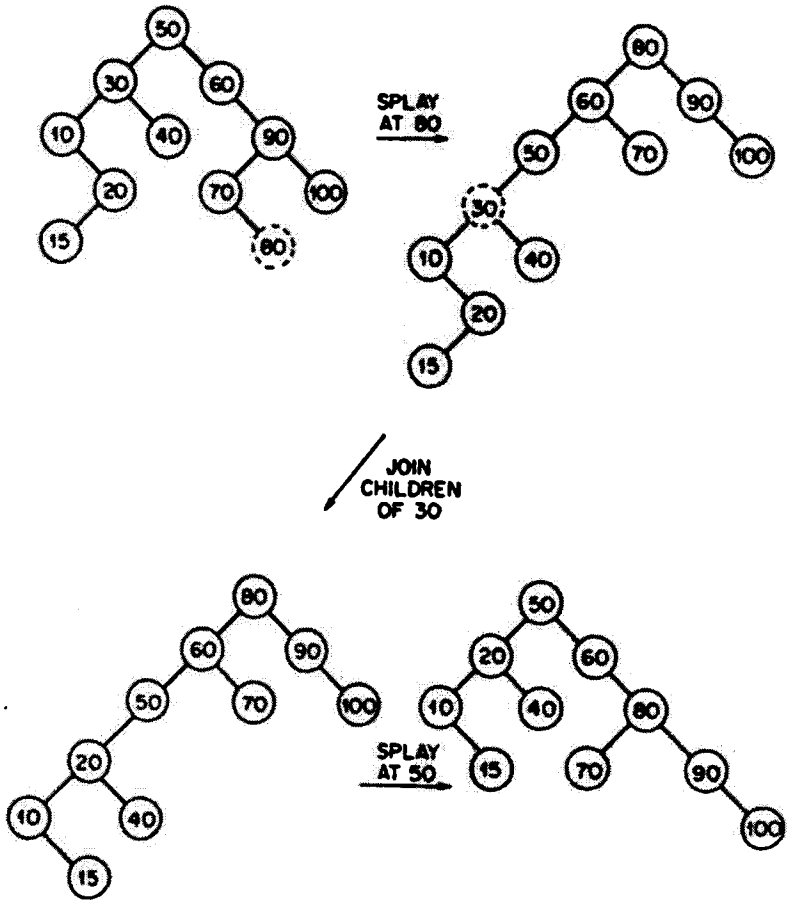


FIG. 9. Alternative implementations of insertion and deletion. Insertion of 80 followed by deletion of 30.

In analyzing the amortized complexity of the various operations on splay trees, let us assume that we begin with a collection of empty trees and that every item is only in one tree at a time. We define the potential of a collection of trees to be the sum of the potentials of the trees plus the sum of the logarithms of the weights of all items not currently in a tree. Thus the net potential drop over a sequence of operations is at most $\sum_{i \in U} \log(w(i)/w'(i))$, where U is the universe of possible items and w and w', respectively, are the initial and final weight functions. In particular, if the item weights never change, the net potential change over the sequence is nonnegative.

For any item i in a tree t, let $i-$ and $i+$, respectively, denote the item preceding i and the item following i in t (in symmetric order). If $i-$ is undefined, let $w(i-) = \infty$; if $i+$ is undefined, let $w(i+) = \infty$.

LEMMA 2 (UPDATE LEMMA). *The amortized times of the splay tree operations have the following upper bounds, where W is the total weight of the items in the*

tree or trees involved in the operation:

$$\textit{access}(i, t): \begin{cases} 3 \log\left(\frac{W}{w(i)}\right) + 1 & \text{if } i \text{ is in } t; \\ 3 \log\left(\frac{W}{\min\{w(i-), w(i+)\}}\right) + 1 & \text{if } i \text{ is not in } t. \end{cases}$$

$$\textit{join}(t_1, t_2): \quad 3 \log\left(\frac{W}{w(i)}\right) + O(1), \quad \text{where } i \text{ is the last item in } t_1.$$

$$\textit{split}(i, t): \begin{cases} 3 \log\left(\frac{W}{w(i)}\right) + O(1) & \text{if } i \text{ is in } t; \\ 3 \log\left(\frac{W}{\min\{w(i-), w(i+)\}}\right) + O(1) & \text{if } i \text{ is not in } t. \end{cases}$$

$$\textit{insert}(i, t): \quad 3 \log\left(\frac{W - w(i)}{\min\{w(i-), w(i+)\}}\right) + \log\left(\frac{W}{w(i)}\right) + O(1).$$

$$\textit{delete}(i, t): \quad 3 \log\left(\frac{W}{w(i)}\right) + 3 \log\left(\frac{W - w(i)}{w(i-)}\right) + O(1).$$

Increasing the weight of the item in the root of a tree t by an amount δ takes at most $\log(1 + \delta/W)$ amortized time, and decreasing the weight of any item takes negative amortized time.

PROOF. These bounds follow from Lemma 1 and a straightforward analysis of the potential change caused by each operation. Let s be the size function just before the operation. In the case of an *access* or *split*, the amortized time is at most $3 \log(s(t)/s(x)) + 1$ by Lemma 1, where x is the node at which the tree is splayed. If item i is in the tree, it is in x, and $s(x) \geq w(i)$. If i is not in the tree, either $i-$ or $i+$ is in the subtree rooted at x, and $s(x) \geq \min\{w(i-), w(i+)\}$. This gives the bound on *access* and *split*. The bound on *join* is immediate from the bound on *access*: the splaying time is at most $3 \log(s(t_1)/w(i)) + 1$, and the increase in potential caused by linking t_1 and t_2 is

$$\log\left(\frac{s(t_1) + s(t_2)}{s(t_1)}\right) \leq 3 \log\left(\frac{W}{s(t_1)}\right).$$

(We have $W = s(t_1) + s(t_2)$.) The bound on *insert* also follows from the bound on *access*: the increase in potential caused by adding a new root node containing i is

$$\log\left(\frac{s(t) + w(i)}{w(i)}\right) = \log\left(\frac{W}{w(i)}\right).$$

The bound on *delete* is immediate from the bounds on *access* and *join*. □

Remark. The alternative implementations of insertion and deletion have the following upper bounds on their amortized times:

$$\textit{insert}(i, t): \quad 2\log\left(\frac{W - w(i)}{\min\{w(i-), w(i+)\}}\right) + \log\left(\frac{W}{w(i)}\right) + O(1).$$

$$\textit{delete}(i, t): \quad 3\log\left(\frac{W - w(i)}{\min\{w(i-), w(i)\}}\right) + O(1).$$

These bounds follow from a modification of the proof of Lemma 1. For the case of equal-weight items, the alternative forms of insertion and deletion each take at most $3 \log n + O(1)$ amortized time on a n-node tree. This bound was obtained independently by Huddleston [17] for the same insertion algorithm and a slightly different deletion algorithm.

The bounds in Lemma 2 are comparable to the bounds for the same operations on biased trees [7, 8, 13], but the biased tree algorithms depend explicitly on the weights. By using Lemma 2, we can extend the theorems of Section 2 in various ways to include update operations. (An exception is that we do not see how to include deletion in a theorem analogous to Theorem 3.) We give here only the simplest example of such an extension.

THEOREM 6 (BALANCE THEOREM WITH UPDATES). *A sequence of m arbitrary operations on a collection of initially empty splay trees takes $O(m + \sum_{j=1}^{m} \log n_j)$ time, where n_j is the number of items in the tree or trees involved in operation j.*

PROOF. Assign to each item a fixed weight of one and apply Lemma 2. □

We can use splay trees as we would use any other binary search tree; for example, we can use them to implement various abstract data structures consisting of sorted sets or lists subject to certain operations. Such structures include dictionaries, which are sorted sets with access, insertion, and deletion, and *concatenatable queues*, which are lists with access by position, insertion, deletion, concatenation, and splitting [3, 22]. We investigate two further applications of splaying in Section 6.

4. *Implementations of Splaying and Its Variants*

In this section we study the implementation of splaying and some of its variants. Our aim is to develop a version that is easy to program and efficient in practice. As a programming notation, we shall use a version of Dijkstra's guarded command language [12], augmented by the addition of procedures and "initializing guards" (G. Nelson, private communication). We restrict our attention to successful accesses, that is, accesses of items known to be in the tree.

Splaying, as we have defined it, occurs during a second, bottom-up pass over an access path. Such a pass requires the ability to get from any node on the access path to its parent. To make this possible, we can save the access path as it is traversed (either by storing it in an auxiliary stack or by using "pointer reversal" to encode it in the tree structure), or we can maintain parent pointers for every node in the tree. If space is expensive, we can obtain the effect of having parent pointers without using extra space, by storing in each node a pointer to its leftmost child and a pointer to its right sibling, or to its parent if it has no right sibling. (See Figure 10.) This takes only two pointers per node, the same as the standard left-child–right-child representation, and allows accessing the left child, right child, or