Self-Adjusting Binary Search Trees

DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, NJ

Abstract. The *splay* tree, a self-adjusting form of binary search tree, is developed and analyzed. The binary search tree is a data structure for representing tables and lists so that accessing, inserting, and deleting items is easy. On an n-node splay tree, all the standard search tree operations have an amortized time bound of $O(\log n)$ per operation, where by "amortized time" is meant the time per operation averaged over a worst-case sequence of operations. Thus splay trees are as efficient as balanced trees when total running time is the measure of interest. In addition, for sufficiently long access sequences, splay trees are as efficient, to within a constant factor, as static optimum search trees. The efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called *splaying*, whenever the tree is accessed. Extensions of splaying give simplified forms of two other data structures: lexicographic or multidimensional search trees and link/cut trees.

Categories and Subject Descriptors: E.1 **[Data]**: Data Structures—*trees*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*sorting and searching*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Amortized complexity, balanced trees, multidimensional searching, network optimization, self-organizing data structures

1. *Introduction*

In this paper we apply the related concepts of *amortized complexity* and *self-adjustment* to binary search trees. We are motivated by the observation that the known kinds of efficient search trees have various drawbacks. Balanced trees, such as height-balanced trees [2, 22], weight-balanced trees [26], and B-trees [6] and their variants [5, 18, 19, 24] have a worst-case time bound of $O(\log n)$ per operation on an n-node tree. However, balanced trees are not as efficient as possible if the access pattern is nonuniform, and they also need extra space for storage of balance information. Optimum search trees [16, 20, 22] guarantee minimum average access time, but only under the assumption of fixed, known access probabilities and no correlation among accesses. Their insertion and deletion costs are also very high. Biased search trees [7, 8, 13] combine the fast average access time of optimum trees with the fast updating of balanced trees but have structural constraints even more complicated and harder to maintain than the constraints of balanced trees. Finger search trees [11, 14, 19, 23, 24] allow fast access in the vicinity of one or more "fingers" but require the storage of extra pointers in each node.

Authors' address: AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

These data structures are all designed to reduce the worst-case time per operation. However, in typical applications of search trees, not one but a sequence of operations is performed, and what matters is the total time the sequence takes, not the individual times of the operations. In such applications, a better goal is to reduce the amortized times of the operations, where by "amortized time" we mean the average time of an operation in a worst-case sequence of operations.

One way to obtain amortized efficiency is to use a "self-adjusting" data structure. We allow the structure to be in an arbitrary state, but during each operation we apply a simple restructuring rule intended to improve the efficiency of future operations. Examples of restructuring heuristics that give amortized efficiency are the move-to-front rule on linear lists [9, 30] and path compression in balanced trees [33, 37].

Self-adjusting data structures have several possible advantages over balanced or otherwise explicitly constrained structures:

(i) In an amortized sense, ignoring constant factors, they are never much worse than constrained structures, and since they adjust according to usage, they can be much more efficient if the usage pattern is skewed.
(ii) They need less space, since no balance or other constraint information is stored.
(iii) Their access and update algorithms are conceptually simple and easy to implement.

Self-adjusting structures have two potential disadvantages:

(i) They require more local adjustments, especially during accesses (look-up operations). (Explicitly constrained structures need adjusting only during updates, not during accesses.)
(ii) Individual operations within a sequence can be expensive, which may be a drawback in real-time applications.

In this paper we develop and analyze the *splay* tree, a self-adjusting form of binary search tree. The restructuring heuristic used in splay trees is *splaying*, which moves a specified node to the root of the tree by performing a sequence of rotations along the (original) path from the node to the root. In an amortized sense and ignoring constant factors, splay trees are as efficient as both dynamically balanced trees and static optimum trees, and they may have even stronger optimality properties. In particular, we conjecture that splay trees are as efficient on any sufficiently long sequence of accesses as any form of dynamically updated binary search tree, even one tailored to the exact access sequence.

The paper contains seven sections. In Section 2 we define splaying and analyze its amortized complexity. In Section 3 we discuss update operations on splay trees. In Section 4 we study the practical efficiency and ease of implementation of splaying and some of its variants. In Section 5 we explore ways of reducing the amount of restructuring needed in splay trees. In Section 6 we use extensions of splaying to simplify two more complicated data structures: lexicographic or multidimensional search trees [15, 25] and link/cut trees [29, 34]. In Section 7 we make some final remarks and mention several open problems, including a formal version of our dynamic optimality conjecture. The appendix contains our tree terminology.

The work described here is a continuation of our research on amortized complexity and self-adjusting data structures, which has included an amortized analysis

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.

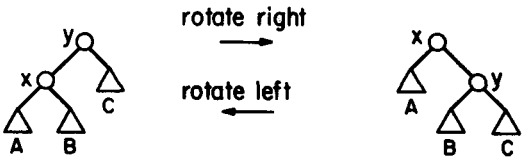


FIG. 1. Rotation of the edge joining nodes x and y. Triangles denote subtrees. The tree shown can be a subtree of a larger tree.

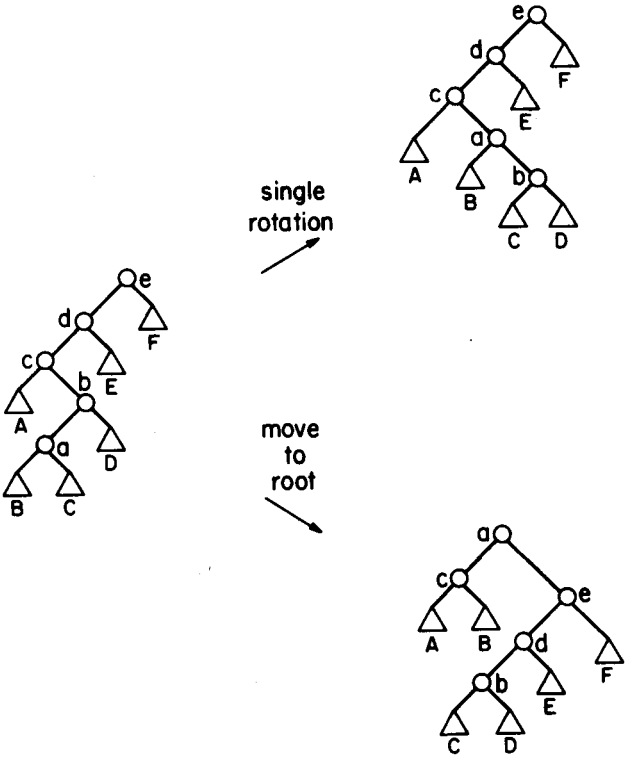


FIG. 2. Restructuring heuristics. The node accessed is a.

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.)

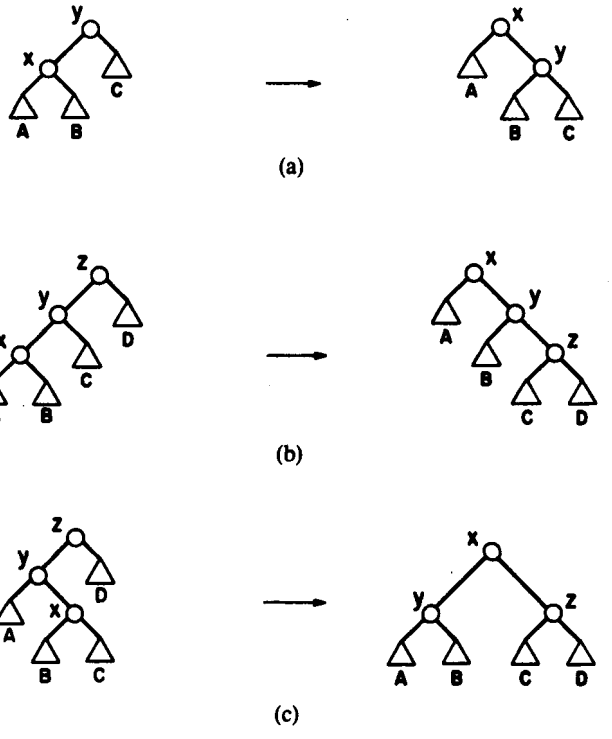


FIG. 3. A splaying step. The node accessed is x. Each case has a symmetric variant (not shown). (a) Zig: terminating single rotation. (b) Zig–zig: two single rotations. (c) Zig–zag: double rotation.

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

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

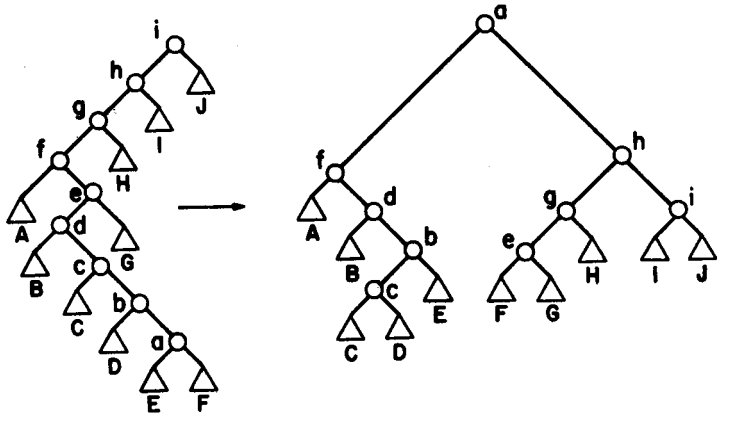
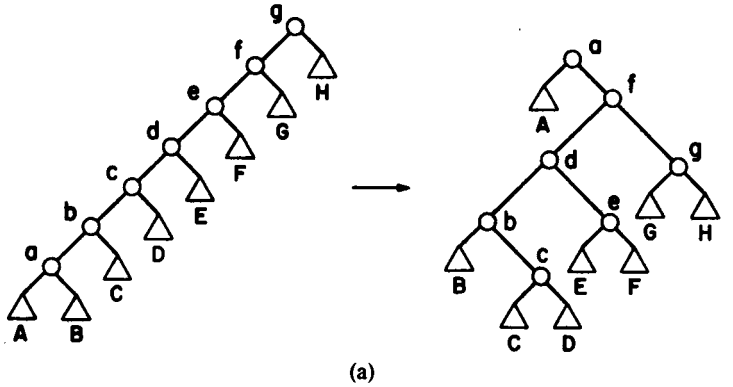
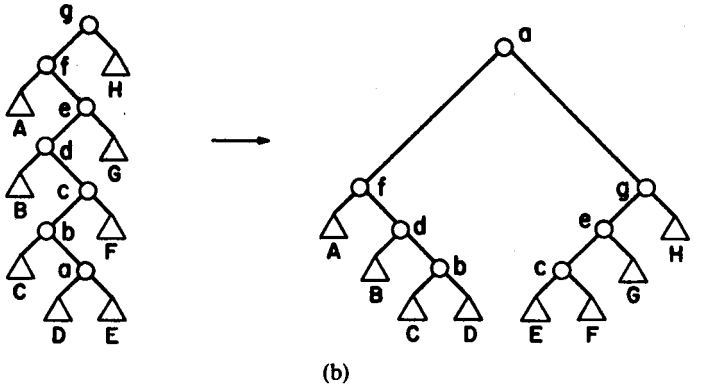


FIG. 4. Splaying at node *a*.



(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

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

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

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

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

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

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 $W \leq 2\sum_{k=1}^{\infty} 1/k^2 = O(1)$, 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.

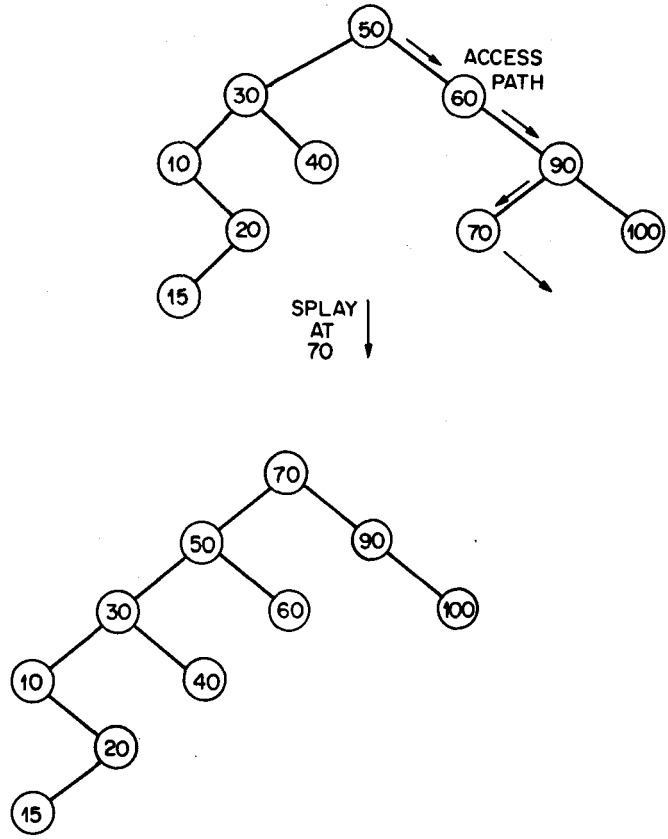


FIG. 6. Splaying after an unsuccessful access of item 80.

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

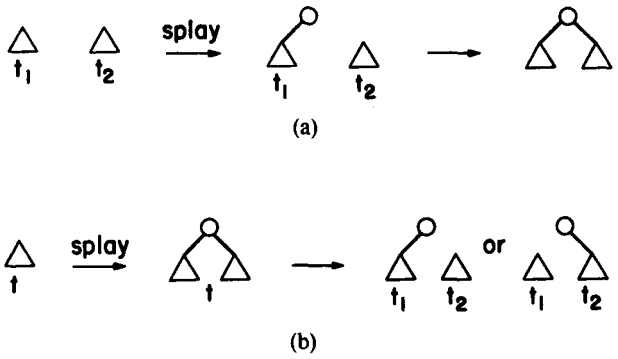


FIG. 7. Implementation of *join* and *split*: (a) *join*(t_1, t_2). (b) *split*(i, t).

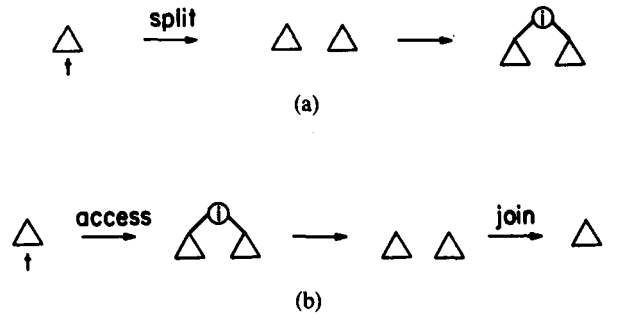


FIG. 8. Implementation of insertion and deletion using *join* and *split*: (a) *insert*(i, t). (b) *delete*(i, t).

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.)

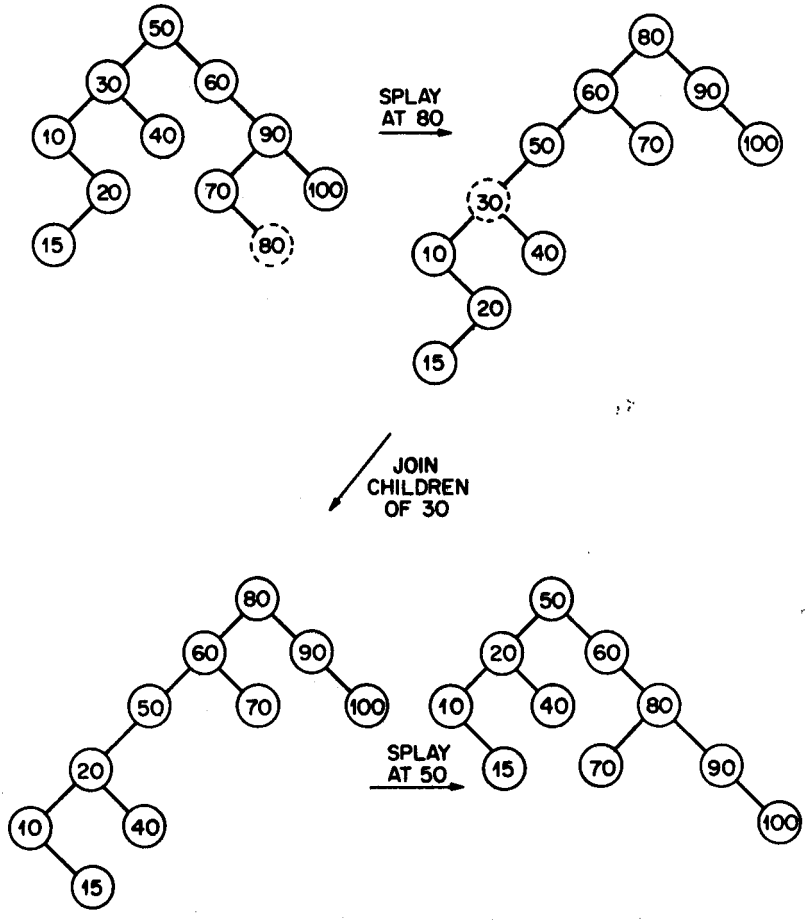


FIG. 9. Alternative implementations of insertion and deletion. Insertion of 80 followed by deletion of 30.

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

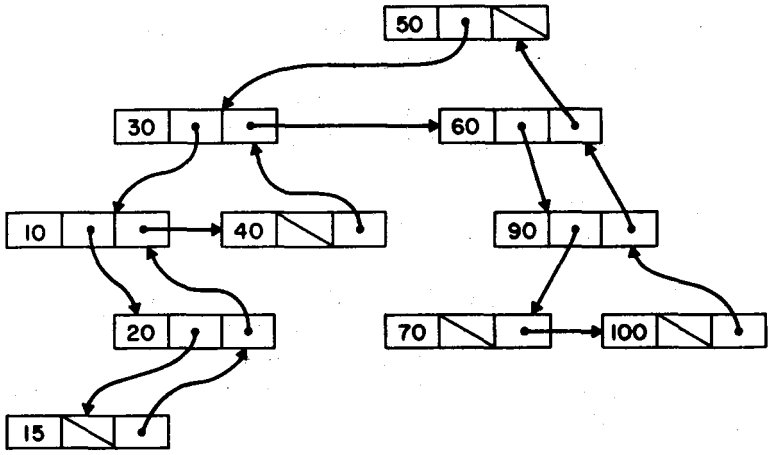


FIG. 10. Leftmost-child–right-sibling representation of the original tree in Figure 6.

parent of any node in at most two steps. The drawback of this representation is loss of time efficiency.

We shall assume that parent pointers are stored explicitly; however, our programs can easily be modified to use other ways of retrieving the access path. The following procedure is a straightforward implementation of splaying. To avoid an explicit test for $p(x) = \text{null}$, we assume that $p(\text{null}) = \mathit{left}(\text{null}) = \mathit{right}(\text{null}) = \text{null}$.

```
procedure splay(x);
  {p(null) = left(null) = right(null) = null}
  do x = left(p(x)) →
    if g(x) = null → rotate right (p(x))
     | p(x) = left(g(x)) → rotate right(g(x)); rotate right(p(x))
     | p(x) = right(g(x)) → rotate right(p(x)); rotate left(p(x))
    fi
   | x = right(p(x)) →
    if g(x) = null → rotate left(p(x))
     | p(x) = right(g(x)) → rotate left(g(x)); rotate left(p(x))
     | p(x) = left(g(x)) → rotate left(p(x)); rotate right(p(x))
    fi
  od {p(x) = null}
end splay;
```

The grandparent function g is defined as follows:

```
function g(x);
  g:= p(p(x))
end g;
```

The procedure *rotate left*(y) rotates the edge joining y and its right child. The procedure *rotate right*(y), whose implementation we omit, is symmetric.

Remark. In the following procedure, the initializing guard "x, z: $x = \mathit{right}(y)$ **and** $z = p(y)$," which is always true, declares two local variables, x and z, and initializes them to $\mathit{right}(y)$ and $p(y)$, respectively.

```
procedure rotate left(y);
  if x, z: x = right(y) and z = p(y) →
    if z = null → skip
     | left(z) = y → left(z) := x
     | right(z) = y → right(z) := x
    fi;
    left(x), right(y) := y, left(x);
    p(x), p(y) := z, x;
    if right(y) = null → skip | right(y) ≠ null → p(right(y)) := y fi
  fi
end rotate left;
```

Inspection of the splay program raises the possibility of simplifying it by omitting the second rotation in the zig–zag case. The resulting program, suggested by M. D. McIlroy (private communication), appears below.

```
procedure simple splay(x);
  {p(null) = left(null) = right(null) = null}
  do x = left(p(x)) →
    if p(x) = left(g(x)) → rotate right(g(x))
     | p(x) ≠ left(g(x)) → skip
    fi;
    rotate right(p(x))
   | x = right(p(x)) →
    if p(x) = right(g(x)) → rotate left(g(x))
     | p(x) ≠ right(g(x)) → skip
    fi;
    rotate left(p(x))
  od{p(x) = null}
end simple splay;
```

An amortized analysis of simple splaying shows that Lemma 1 holds with a constant factor of 3.16+ in place of three. Thus the method, though simpler than the original method, has a slightly higher bound on the number of rotations.

The advantage of implementing splaying using rotation as a primitive is that we can encapsulate all the restructuring, including any required updating of auxiliary fields, in the rotation procedures. The disadvantage is that many unnecessary pointer assignments are done. We can achieve greater efficiency by eliminating the superfluous assignments by expanding the rotation procedures in-line, simplifying, and keeping extra information in the control state.

A bottom-up splaying method is appropriate if we have direct access to the node at which the splaying is to occur. We shall see an example of this in Section 6. However, splaying as used in Sections 2 and 3 only occurs immediately after an access, and it is more efficient to splay on the way down the tree during the access. We present a top-down version of splaying that works in this way.

Suppose we wish to access an item i in a tree. During the access and concurrent splaying operation, the tree is broken into three parts: a *left tree*, a *middle tree*, and a *right tree*. The left and right trees contain all items in the original tree so far known to be less than i and greater than i, respectively. The middle tree consists of the subtree of the original tree rooted at the current node on the access path. Initially the current node is the tree root and the left and right trees are empty. To do the splaying, we search down from the root looking for i, two nodes at a time, breaking links along the access path and adding each subtree thus detached to the bottom right of the left tree or the bottom left of the right tree, with the proviso

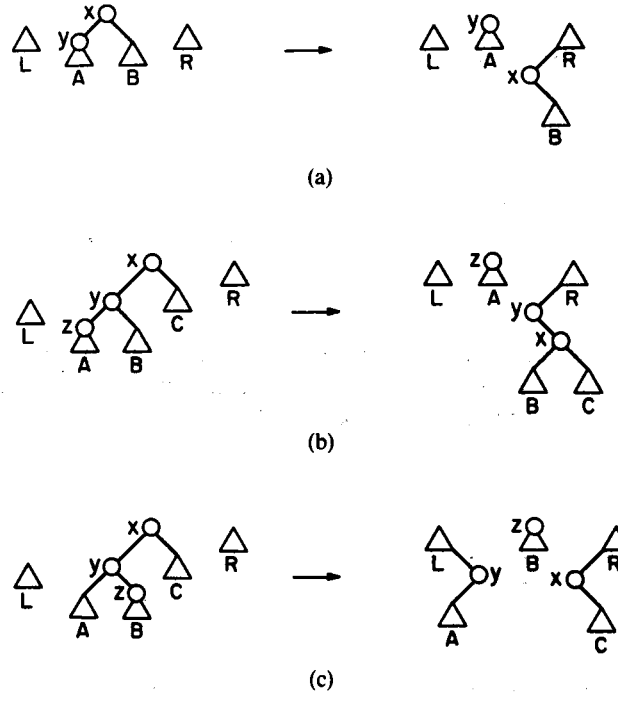


FIG. 11. Top-down splaying step. Symmetric variants of cases are omitted. Initial left tree is L, right tree is R. Labeled nodes are on the access path. (a) Zig: Node y contains the accessed item. (b) Zig–zig. (c) Zig–zag.

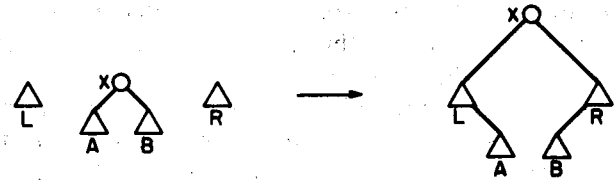


FIG. 12. Completion of top-down splaying. Node x contains the accessed item.

that if we take two steps down in the same direction (left–left or right–right) we do a rotation before breaking a link. More formally, the splaying operation consists of repeatedly applying the appropriate case among those shown in Figure 11 until none applies, then reassembling the three remaining trees as shown in Figure 12. A similar top-down restructuring method, but without the rotations and consequently with poor amortized efficiency, was proposed by Stephenson [32].

The following program implements this method. The program uses three local variables, t, l, and r, denoting the current vertex, the last node in symmetric order in the left tree, and the first node in symmetric order in the right tree, respectively. The updating of these variables occurs in the primitive procedures. These are *rotate left* and *rotate right*, which rotate the edge joining t and its right or left child, respectively; *link left* and *link right*, which break the link between t and its left or right child and attach the resulting tree to the left or right tree, respectively; and *assemble*, which completes the splaying by performing the assembly of Figure 12. The program contains an initializing guard that declares l and r and initializes

them both to null. After the first left link, the right child of null is the root of the left tree; after the first right link, the left child of null is the root of the right tree.

```
procedure top-down splay(i, t);
  if l, r: l = r = null →
    left(null) := right(null) := null;
    do i < item(t) →
      if i = item(left(t)) → link right
       | i < item(left(t)) → rotate right; link right
       | i > item(left(t)) → link right; link left
      fi
     | i > item(t) →
      if i = item(right(t)) → link left
       | i > item(right(t)) → rotate left; link left
       | i < item(right(t)) → link left; link right
      fi
    od{i = item(t)};
    assemble
  fi
end top-down splay;
```

Implementations of *rotate left*, *link left*, and *assemble* appear below; *rotate right* and *link right*, whose definitions we omit, are symmetric to *rotate left* and *link left*, respectively.

```
procedure link left;
  t, l, right(l) := right(t), t, t
end link left;

procedure rotate left;
  t, right(t), left(right(t)) := right(t), left(right(t)), t
end rotate left;

procedure assemble;
  left(r), right(l) := right(t), left(t);
  left(t), right(t) := right(null), left(null)
end assemble;
```

We can simplify top-down splaying as we did bottom-up splaying, by eliminating the second linking operation in the zig–zag case. The resulting program is as follows:

```
procedure simple top-down splay(i, t);
  if l, r: l = r = null →
    left(null) := right(null) := null;
    do i < item(t) →
      if i < item(left(t)) → rotate right
       | i ≥ item(left(t)) → skip
      fi;
      link right
     | i > item(t) →
      if i > item(right(t)) → rotate left
       | i ≤ item(right(t)) → skip
      fi;
      link left
    od{i = item(t)};
    assemble
  fi
end simple top-down splay;
```

Lemma 1 holds for both top-down splaying methods with an unchanged constant factor of three. It is easy to modify the program for either method to carry out the various tree operations discussed in Section 3. For these operations a top-down method is more efficient than a bottom-up method, but the choice between the top-down methods is unclear; the simple method is easier to program but probably somewhat less efficient. Experiments are needed to discover the most practical splaying method.

5. *Semi-adjusting Search Trees*

A possible drawback of splaying is the large amount of restructuring it does. In this section we study three techniques that reduce the amount of restructuring but preserve at least some of the properties of splay trees. As in Section 4, we shall restrict our attention to successful accesses.

Our first idea is to modify the restructuring rule so that it rotates only some of the edges along an access path, thus moving the accessed node only partway toward the root. Semisplaying, our restructuring heuristic based on this idea, differs from ordinary bottom-up splaying only in the zig–zig case: after rotating the edge joining the parent $p(x)$ with the grandparent $g(x)$ of the current node x, we do not rotate the edge joining x with $p(x)$, but instead continue the splaying from $p(x)$ instead of x. (See Figure 13.)

The effect of a semisplaying operation is to reduce the depth of every node on the access path to at most about half of its previous value. If we measure the cost of a semisplaying operation by the depth of the accessed node, then Lemma 1 holds for semisplaying with a reduced constant factor of two in place of three. Furthermore, only one rotation is performed in the zig–zag case, but two steps are taken up the tree.

Like splaying, semisplaying has many variants. We describe only one, a top-down version, related to a method suggested by C. Stephenson (private communication). Think of performing a semisplaying operation as described above, bottom-up, except that if the access path contains an odd number of edges, perform the zig case at the bottom of the path instead of at the top. We can simulate this variant of semisplaying top-down, as follows. As in top-down splaying, we maintain a left tree, a middle tree, and a right tree. In addition we maintain a *top tree* and a node *top* in the top tree having a vacant child. The relationship among the trees is that all items in the left tree are less than the accessed item i and also less than those in the middle tree; all items in the right tree are greater than i and also greater than those in the middle tree; and all items in the left, middle, and right trees fall between the item in *top* and the item in its predecessor in the top tree if the vacant child of *top* is its left, or between the item in *top* and the item in its successor in the top tree if the vacant child of *top* is its right. Initially the left, right, and top trees are empty and the middle tree is the entire original tree.

Let i be the item to be accessed. Each splaying step requires looking down two steps in the middle tree from the root and restructuring the four trees according to whether these steps are to the left or to the right. If i is in the root of the middle tree, we combine the left, middle, and right trees as in the completion of top-down splaying (see Figure 12) and then make the root of the combined tree (which contains i) a child of *top*, filling its vacancy. This completes the splaying. On the other hand, if i is not in the root of the middle tree, we carry out a zig, zig–zig, or zig–zag step as appropriate.

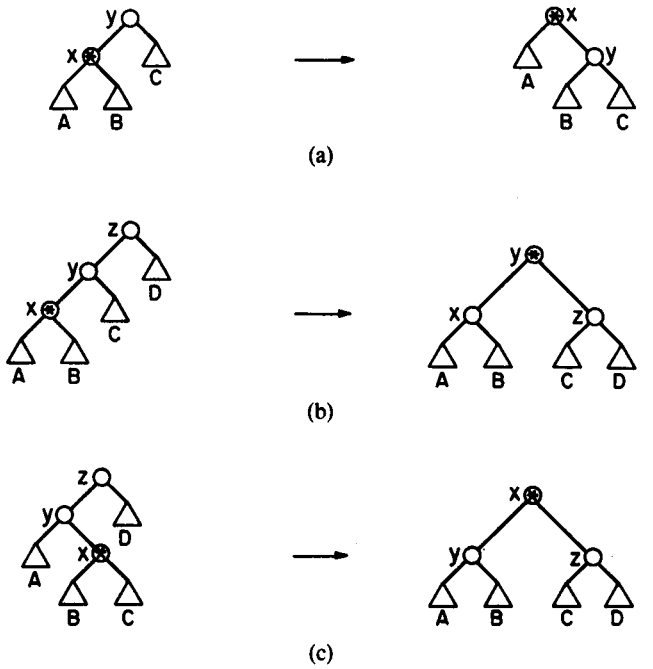


FIG. 13. A semisplaying step. Symmetric variants of cases are omitted. In each case, the starred node is the current node of the splaying operation. (a) Zig. (b) Zig-zig. (c) Zig-zag.

The zig and zig-zag cases are exactly as in top-down splaying; they do not affect the top tree. The zig-zig case is as illustrated in Figure 14. Suppose that the access path to i contains the root x of the middle tree, its left child y, and the left child of y, say z. We perform a right rotation on the edge joining x and y. Then we assemble all four trees as in the terminating case, making node y (now the root of the middle tree) a child of *top* and making the left and right trees the left and right subtrees of y. Finally, we break the link between z and its new parent, making the subtree rooted at z the new middle tree, the rest of the tree the new top tree, and the old parent of z the new *top*. The left and right trees are reinitialized to be empty.

A little thought will verify that top-down semisplaying indeed transforms the tree in the same way as bottom-up semisplaying with the zig step, if necessary, done at the bottom of the access path. Lemma 1 holds for top-down semisplaying with a constant factor of two.

Whether any version of semisplaying is an improvement over splaying depends on the access sequence. Semisplaying may be better when the access pattern is stable, but splaying adapts much faster to changes in usage. All the tree operations discussed in Section 3 can be implemented using semisplaying, but this requires using the alternative implementations of insertion and deletion. Also, the *join* and *split* algorithms become more complicated. The practical efficiency of the various splaying and semisplaying methods remains to be determined.

Another way to reduce the amount of restructuring in splay trees is to splay only sometimes. We propose two forms of conditional splaying. For simplicity we restrict our attention to a sequence of successful accesses on a tree containing a fixed set of items. One possibility is to splay only when the access path is abnormally

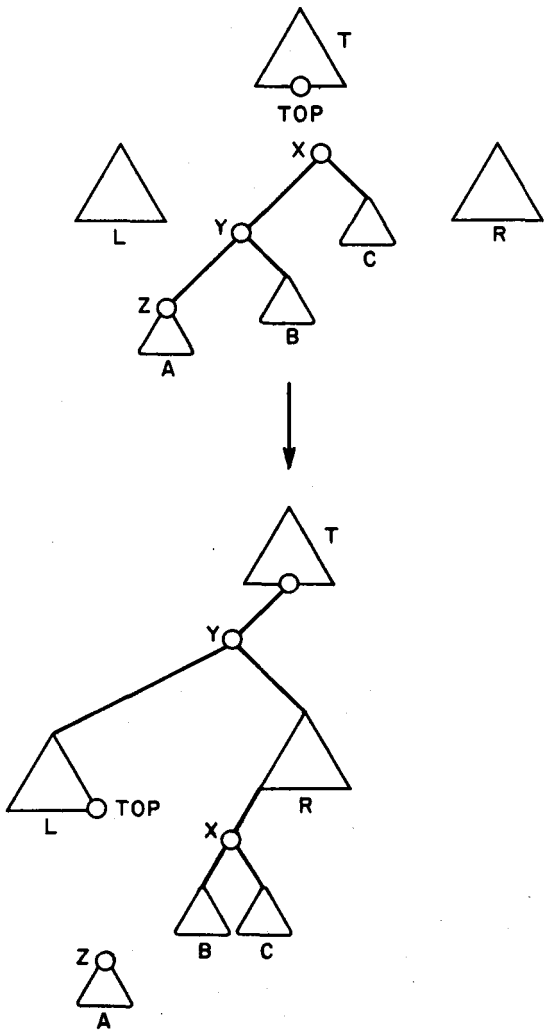


FIG. 14. Zig–zig case of a top-down semisplaying step. The symmetric variant is omitted. The initial left, right, and top trees are L, R, and T, respectively. Nodes x, y, and z are on the access path. Tree A is the new middle tree, the new left and right trees are empty, and the rest of the nodes form the new top tree.

long. To define what we mean by "abnormally long," consider any variant of splaying (or semisplaying) such that Lemma 1 holds with a constant factor of c. Suppose every item i has a fixed weight $w(i)$, and let c' be any constant greater than c. We call an access path for item i *long* if the depth of the node containing i is at least $c'\log(W/w(i)) + c'/c$, and *short* otherwise. (Recall that W is the total weight of all the items.)

THEOREM 7 (LONG SPLAY THEOREM). *If we splay only when the access path is long and do no restructuring when the access path is short, then the total splaying time is* $O(\sum_{i=1}^{n} \log(W/w(i)))$, *that is, proportional to the amortized time to access each item once. The constant factor in this estimate is proportional to* $c'/(c' - c)$. *Thus the total restructuring time is independent of* m, *the number of accesses.*

PROOF. Consider a splaying operation on a node of depth $d \geq c'\log(W/w(i)) + c'/c$. Let ϕ and ϕ', respectively, be the potential before and after the splaying. Since $c'/c > 1$, the actual time of the splaying is d, and we have by Lemma 1 that $d + \phi' - \phi \leq c\log(W/w(i)) + 1$. Thus the splaying reduces the potential of the tree by at least $d - c\log(W/w(i)) - 1$. This means that if we amortize the potential

reduction over the entire splaying operation, each unit of time spent causes a potential drop proportional to at least

$$\frac{d - c\log(W/w(i)) - 1}{d} \geq 1 - \frac{c\log(W/w(i)) + 1}{c'\log(W/w(i)) + c'/c}$$
$$= 1 - \frac{c}{c'}.$$

This means in particular that such a splaying operation can only decrease the potential. Since any access along a short access path causes no restructuring and thus no change in potential, the total splaying time over the sequence is proportional to at most $c'/(c' - c)$ times the overall potential drop. The overall potential drop is at most $\sum_{i=1}^{n} \log(W/w(i))$, giving the theorem. □

We have as a simple application of Theorem 7 that if we splay only when the access path is of length at least $c'\log n + c'/c$, then the total splaying time is $O(n \log n)$.

This form of conditional splaying has two drawbacks: it requires two passes over each long access path (one pass just to determine whether to splay), and the decision whether to splay requires knowing the item weights, which must be predetermined on the basis of known or assumed properties of the access sequence. Thus the method is antithetical to the philosophy of self-adjustment.

Another form of conditional splaying, which does not depend on the item weights, is to splay during the first $j - 1$ accesses, and just before the jth access to stop splaying completely, using the tree resulting from all the splaying operations as a static search tree. The intuition behind this idea is that if splaying is efficient in an amortized sense, then the average state of the data structure ought to be a good one. Under appropriate assumptions, we can prove this. Suppose that each item i has a fixed access probability p_i and that all accesses are independent. For any search tree T, we define the *average access time* of T to be $\sum_{i=1}^{n} p_i d_i$, where d_i is the depth of the node containing item i.

THEOREM 8 (SNAPSHOT THEOREM). *Suppose we begin with any initial tree, carry out a sequence of m accesses, and randomly stop splaying just before access j, with probability $1/m$ for each possible value of j. Then the search tree resulting from the sequence of splaying operations has an expected average access time of $O((n \log n)/m + \sum_{i=1}^{n} p_i \log(1/p_i))$.*

PROOF. For any access sequence α and any item i, let $|\alpha|$ be the length of α (number of accesses), and let αi be the access sequence consisting of α followed by an access of i. Let p_α be the probability of α, that is, the product of the probabilities of all the accesses. Let $t_\alpha(\beta)$ be the total time required to carry out the sequence of accesses β using splaying, assuming that we have first performed the sequence of accesses α using splaying. (If the subscript α is missing, we take α to be the empty sequence.) Let T_α be the average access time of the tree formed by performing the sequence of accesses α using splaying. (If α is the empty sequence, T_α is the average access time of the initial tree.)

We wish to estimate the quantity $\sum_{|\alpha|<m} p_\alpha T_\alpha/m$. We have $T_\alpha = \sum_{i=1}^{n} p_i t_\alpha(i)$, which means that the quantity to be estimated is $\sum_{|\alpha i|\leq m} p_{\alpha i} t_\alpha(i)/m$. We have

$$\sum_{|\alpha|=m-1} p_\alpha t(\alpha) + \sum_{|\alpha i|=m} p_{\alpha i} t_\alpha(i) = \sum_{|\alpha i|=m} p_{\alpha i} t(\alpha i),$$

from which it follows by induction on m that

$$\sum_{|\alpha i|\le m} \frac{p_{\alpha i} t_\alpha(i)}{m} = \sum_{|\alpha|=m} \frac{p_\alpha t(\alpha)}{m} \quad \text{for} \quad m \ge 1.$$

Taking $w(i) = p_i + 1/n$ for each item i, we have by Lemma 1, for any sequence α, that

$$t(\alpha) = O\left(n \log n + m + \sum_{j=1}^{m} \log\left(\frac{1}{p_{\alpha(j)}}\right)\right),$$

where $\alpha(j)$ is the item accessed during access j in sequence α. In this estimate the constant factor is independent of α. Thus the expected value of the average access time is

$$O\left(\frac{n \log n}{m} + \sum_{|a|=m} p_\alpha \left(\sum_{j=1}^{m} \frac{\log(1/p_{\alpha(j)})}{m}\right)\right).$$

For a fixed item i, let us accumulate all terms in the double sum such that $\alpha(j) = i$. For any fixed access j, the sum of terms for all sequences α such that $\alpha(j) = i$ is $p_i \log(1/p_i)/m$. Since there are m possible values of j, the sum for all terms such that $\alpha(j) = i$ is $p_i \log(1/p_i)$. We conclude that the expected average access time is $O((n \log n)/m + \sum_{i=1}^{n} p_i \log(1/p_i))$. □

Theorem 8 and the information-theoretic result quoted in Section 2 imply that if we carry out a sequence of $\Omega(n \log n)$ accesses and splay only during the first $j - 1$ accesses for a random value of j, we are likely to produce a search tree whose average access time is within a constant factor of minimum. The only property of splay trees used to prove Theorem 8 is their amortized efficiency. Thus the theorem generalizes to any self-adjusting data structure that is efficient in an amortized sense. For example, the move-to-front update rule for sequential search [30], if stopped after a sufficiently large random number of accesses, produces a list whose expected average access time is within a constant factor of minimum.

Permanent cessation of restructuring has two drawbacks: there is a small chance that the resulting static data structure will be inefficient; and, if the access pattern changes, an originally efficient static structure can lose its efficiency. Thus this method is probably practical only in situations where the probabilistic assumptions needed to prove Theorem 8 hold.

6. *Two Applications of Splaying*

In this section we discuss two uses of splaying in data structures more elaborate than search trees. These applications derive from the observation that if we are content with amortized efficiency, then splay trees can generally be used in place of the much more cumbersome biased search trees [7, 8, 13] with no degradation in running time and considerable simplification.

Our first application is to lexicographic or multidimensional search trees [15, 25], which are related to digital search trees and tries [22]. Let Σ be a totally ordered alphabet. Suppose we wish to store a set of strings S over Σ so that repeated access operations will be efficient. As a primitive to support accesses, we allow the comparison of any two symbols in Σ at unit cost.

We can represent such a set of strings by a *lexicographic search tree.* This is a ternary tree (every node has a left child, a middle child, and a right child, each possibly null) containing one symbol in each of its nodes, with the following

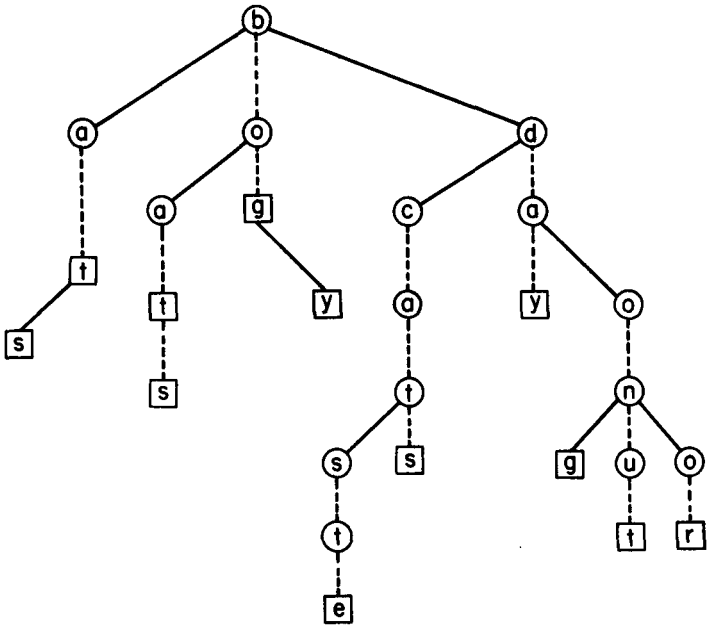


FIG. 15. A lexicographic search tree for the words at, as, bat, bats, bog, boy, caste, cats, day, dog, donut, door. Squares denote terminal nodes.

properties (see Figure 15):

(i) The nodes of the tree correspond exactly to the prefixes of the strings in S in the following sense: If we traverse any path in the tree down from the root, concatenating all the symbols in the nodes from which we leave by a middle edge (an edge to a middle child), then we obtain a prefix, and all possible prefixes are uniquely obtainable in this way.

(ii) If we descend from a node x to a node y by a path containing no middle children, then the symbol in x is less than the symbol in y if and only if the first step from x to y is to a right child.

We store in each node of a lexicographic search tree a *terminal bit* indicating whether the corresponding prefix is actually a string in S. These bits are only needed if S contains a pair of strings, one of which is a prefix of the other. We can eliminate such pairs by adding a special "end of string" symbol to the end of every string. We call each node corresponding to a string in S a *terminal node.*

If we think of the middle edges as *dashed* and the remaining edges as *solid,* then property (ii) is the same as saying that the connected components defined by the solid edges, which we call *solid subtrees,* are binary search trees whose items are symbols in Σ. The entire lexicographic search tree is a hierarchy consisting of these binary search trees joined by the dashed edges. The *middle depth* of a solid subtree (the number of dashed edges on the path from the root to the subtree) determines the symbol position for which the solid subtree is to be used as a search tree.

We can use a lexicographic search tree to access the set of strings it represents as follows. Let σ be any string, and for any position i let $\sigma(i)$ be the ith symbol in σ. We begin with the current node x equal to the root and with the current position i equal to 1 and repeat the following step until we have matched all the symbols in σ or we have reached null by running off the bottom of the tree (in the latter case

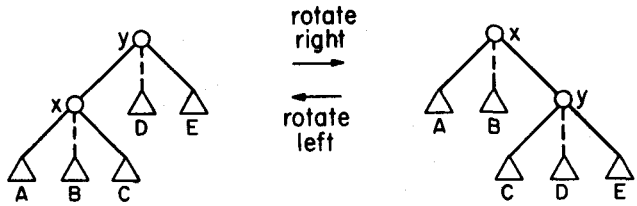


FIG. 16. Rotation of the solid edge joining nodes x and y in a lexicographic search tree.

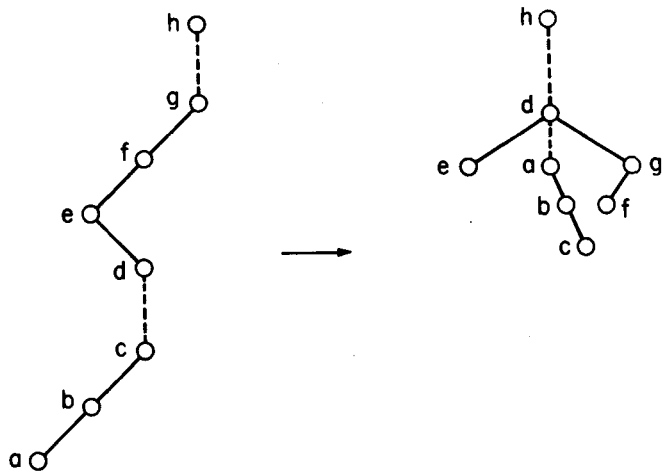


FIG. 17. Splaying at a node a in a lexicographic splay tree. Subtrees of nodes along the access path are deleted for clarity.

the access fails):

Search Step. If $\sigma(i)$ is less than the symbol in x, replace x by *left*(x). If $\sigma(i)$ is greater than the symbol in x, replace x by *right*(x). If $\sigma(i)$ equals the symbol in x and i is not the last position in x, replace x by *middle*(x) and i by $i + 1$. (The ith symbol in string σ has been matched.) Otherwise ($\sigma(i)$ equals the symbol in x and i is the last position in σ) terminate the search, with success if x is a terminal node and failure if not.

The time for an access, either successful or unsuccessful, is bounded by the length of σ plus the number of left and right edges traversed. Thus, to minimize the access time we want to keep the depths of the solid subtrees small.

Lexicographic search trees are susceptible to the same restructuring primitive as binary search trees, namely rotation, which takes $O(1)$ time, rearranges left and right children but not middle children, and preserves properties (i) and (ii). (See Figure 16.) Thus we can extend the standard binary search tree techniques to lexicographic search trees. In particular, we can use an extension of splaying as a restructuring heuristic after each access. We call the resulting data structure a *lexicographic splay tree.* To splay at a node x, we proceed up the access path from x, one or two nodes at a time, carrying out splaying steps, with the additional proviso that whenever x becomes a middle child, we continue from $p(x)$ instead of from x. That is, we splay along each part of the access path that is within a solid subtree. (See Figure 17.)

After splaying at a node x, the path from the root to x contains only dashed edges. (Since rotations do not disturb middle children, they do not change the

middle depth of any node.) We can easily extend the amortized analysis of Section 2 to lexicographic splay trees. We associate with each string $\sigma \in S$ an arbitrary positive weight $w(\sigma)$, define the size $s(x)$ of a node x in the tree to be the sum of the weights of the strings whose terminal nodes are (not necessarily proper) descendants of S, define the rank of x to be $r(x) = \log s(x)$, and finally define the potential of a tree to be the sum of the ranks of its nodes.

LEMMA 3 (ACCESS LEMMA FOR LEXICOGRAPHIC SPLAY TREES). *Splaying a tree with root t at a node x takes $O(d_m(x) + \log(s(t)/s(x)))$ amortized time, where $d_m(x)$ the middle depth of x.*

PROOF. An easy extension of the proof of Lemma 1. □

Lemma 3 implies that the amortized time to access a string σ in a lexicographic splay tree is $O(|\sigma| + \log(W/w(\sigma)))$, where W is the total weight of all the strings represented by the tree. (Recall that $|\sigma|$ is the length of σ.) All the results of Section 2 extend to lexicographic splay trees, with the modification that accessing a string σ has an extra additive $O(|\sigma|)$ overhead. We can easily implement insertion, deletion, and even *join* and *split* on lexicographic splay trees. We obtain amortized time bounds similar to those in Section 3, again with an extra additive term for the string length. Since all these extensions are straightforward, we leave them as exercises. See references [8], [15], [22], and [25] for further discussion of lexicographic search trees, related data structures, and their uses.

Our second and most complicated application of splaying is to the link/cut trees of Sleator and Tarjan [29]. The link/cut tree problem is that of maintaining an abstract data structure consisting of a forest of rooted trees, each node of which has a real-valued cost, under a sequence of the following six kinds of operations (see Figure 18). We regard each tree edge as directed from child to parent.

find cost(v):	Return the cost of node v.
find root(v):	Return the root of the tree containing node v.
find min(v):	Return the node of minimum cost on the path from v to r, the root of the tree containing r. In the case of ties, choose the node that is closest to r.
add cost(v, x):	Add real number x to the cost of every node on the path from v to the root of the tree containing r.
link(v, w):	Add an edge from v to w, thereby making v a child of w in the forest. This operation assumes that v is the root of one tree and w is in another tree.
cut(v):	Delete the edge from v to its parent, thereby dividing the tree containing v into two trees. This operation assumes that v is not a tree root.

Link/cut trees have important applications in algorithms for various network optimization problems, including the maximum flow and minimum cost flow problems [27, 29, 34]. In discussing link/cut trees we shall denote the total number of nodes by n and the total number of operations by m. An obvious way to represent such trees is to store with each node its cost and a pointer to its parent. With this representation, each *find cost*, *link*, or *cut* operation takes $O(1)$ time, and each *find root*, *find min*, and *add cost* operation takes $O(n)$ time. By using a hierarchy of biased search trees [7, 8, 13] to represent each link/cut tree, Sleator and Tarjan obtained a data structure with an $O(\log n)$ time bound per operation. Here we

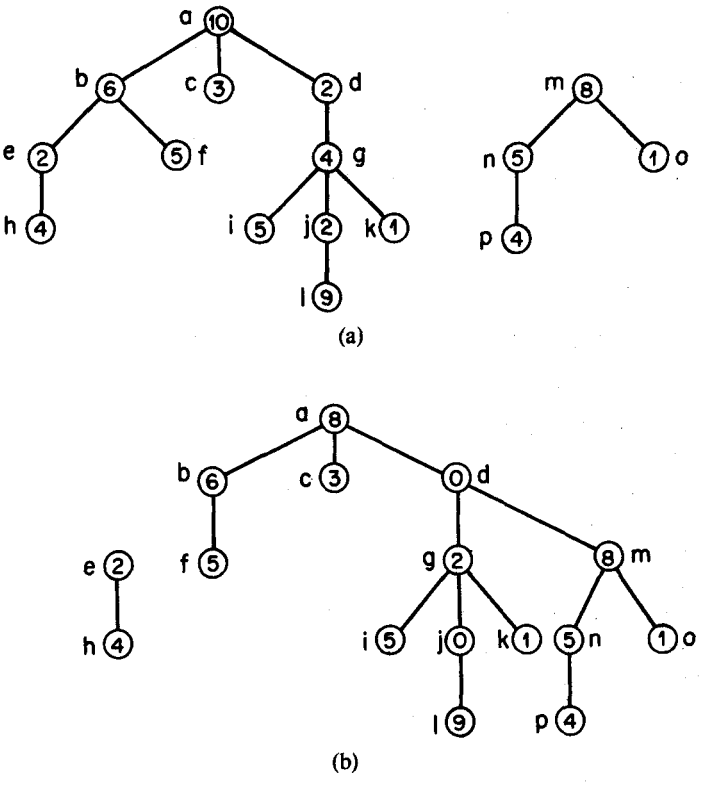


FIG. 18. Link/cut tree operations. (a) Two rooted trees. Numbers in nodes are costs. Operation *find cost*(j) returns 2, *find min*(j) returns d. (b) Trees after *add cost*(j, -2), *link*(m, d), *cut*(e).

present a much-streamlined data structure that uses splay trees in place of biased trees and achieves an $O(\log n)$ amortized time bound per operation. A preliminary version of this structure appears in a monograph by the second author [34].

We represent each link/cut tree T by a *virtual tree* V containing the same nodes as T but having different structure. Each node of V has a left child and a right child, either or both of which may be a null, and zero or more middle children. As in the case of lexicographic search trees, we call an edge joining a middle child to its parent *dashed* and all other edges *solid*. Thus the virtual tree consists of a hierarchy of binary trees, which we call *solid subtrees*, interconnected by dashed edges. The relationship between T and V is that the parent in T of a node x is the symmetric-order successor of x in its solid subtree in V, unless x is last in its solid subtree, in which case its parent in T is the parent of the root of its solid subtree in V. (See Figure 19.) In other words, each solid subtree in V corresponds to a path in T, with symmetric order in the solid subtree corresponding to linear order along the path, from first vertex to last vertex.

Remark. In the preliminary version of this data structure [34], symmetric order in a solid subtree of V corresponds to reverse order along the corresponding path in T. The present definition is equivalent but more natural.

To represent V, we store pointers in each node x to its parent $p(x)$ in V, its left child *left*(x), and its right child *right*(x). These pointers allow us not only to move up and down the tree but also to determine in $O(1)$ time whether a given node x is the root of V or a left, middle, or right child: we merely compare x to *left*($p(x)$)

and $right(p(x))$. To complete the representation we must store information about the node costs. For each node x, let $cost(x)$ be the cost of x, and let $min\ cost(x)$ be the minimum cost of any descendant of x in the same solid subtree. The *min cost* function will allow us to perform *find min* easily. Instead of storing *cost* and *min cost* explicitly, which makes the *add cost* operation expensive, we store these functions in difference form. To be precise, we store the following two values in each node x (see Figure 19):

$$\Delta cost(x) = \begin{cases} cost(x) & \text{if } x \text{ is the root} \\ & \text{of a solid subtree of } V, \\ cost(x) - cost(p(x)) & \text{otherwise;} \end{cases}$$

$$\Delta min(x) = cost(x) - min\ cost(x).$$

Note. We have $\Delta min(x) \geq 0$ for any node x.

To carry out the link/cut tree operations efficiently using virtual trees, we use a form of splaying that moves a designated node to the root of its virtual tree. There are two $O(1)$-time restructuring primitives that are applicable to virtual trees. The first is rotation, which, as in the case of lexicographic splay trees, rearranges left and right children and leaves middle children alone. The following formulas define the necessary updating of $\Delta cost$ and Δmin after a rotation. Let the edge being rotated join a vertex v to its parent w, let a and b be the children of v before the rotation, and let b and c be the children of w after the rotation. The unprimed and primed functions, respectively, denote values before and after the rotation. (See Figure 20.)

$$\begin{aligned} \Delta cost'(v) &= \Delta cost(v) + \Delta cost(w), \\ \Delta cost'(w) &= -\Delta cost(v), \\ \Delta cost'(b) &= \Delta cost(v) + \Delta cost(b), \\ \Delta min'(w) &= \max\{0, \Delta min(b) - \Delta cost'(b), \Delta min(c) - \Delta cost(c)\}, \\ \Delta min'(v) &= \max\{0, \Delta min(a) - \Delta cost(a), \Delta min'(w) - \Delta cost'(w)\}. \end{aligned}$$

All other values are unaffected by the rotation.

The second restructuring primitive is *splicing*. If w is the root of a solid subtree and v is any middle child of w, we can make v the left child of w and the old left child, if any, a middle child by defining $left(w) = v$. (See Figure 21.) This requires the following updating of $\Delta cost$ and Δmin, where u is the old left child of w, and the unprimed and primed functions, respectively, denote values before and after the splice:

$$\begin{aligned} \Delta cost'(v) &= \Delta cost(v) - \Delta cost(w), \\ \Delta cost'(u) &= \Delta cost(u) + \Delta cost(w), \\ \Delta min'(w) &= \max\{0, \Delta min(v) - \Delta cost'(v), \Delta min(right(w)) - \Delta cost(right(w))\}. \end{aligned}$$

We describe the splaying operation as a three-pass bottom-up process. Let x be the node at which the tree is to be splayed. During the first pass, we walk up the virtual tree from x to the root, splaying within each solid subtree exactly as in lexicographic splay trees. After the first pass, the path from x to the root consists entirely of dashed edges. During the second pass, we again walk from x up to the root, this time splicing at each proper ancestor of x so as to make x and all its ancestors (except the root) left children. After the second pass, x and the root are in the same solid subtree. During the third pass, we walk from x up to the root, splaying in the normal fashion. After the third pass, x is the root of the entire virtual tree. (See Figure 22.)

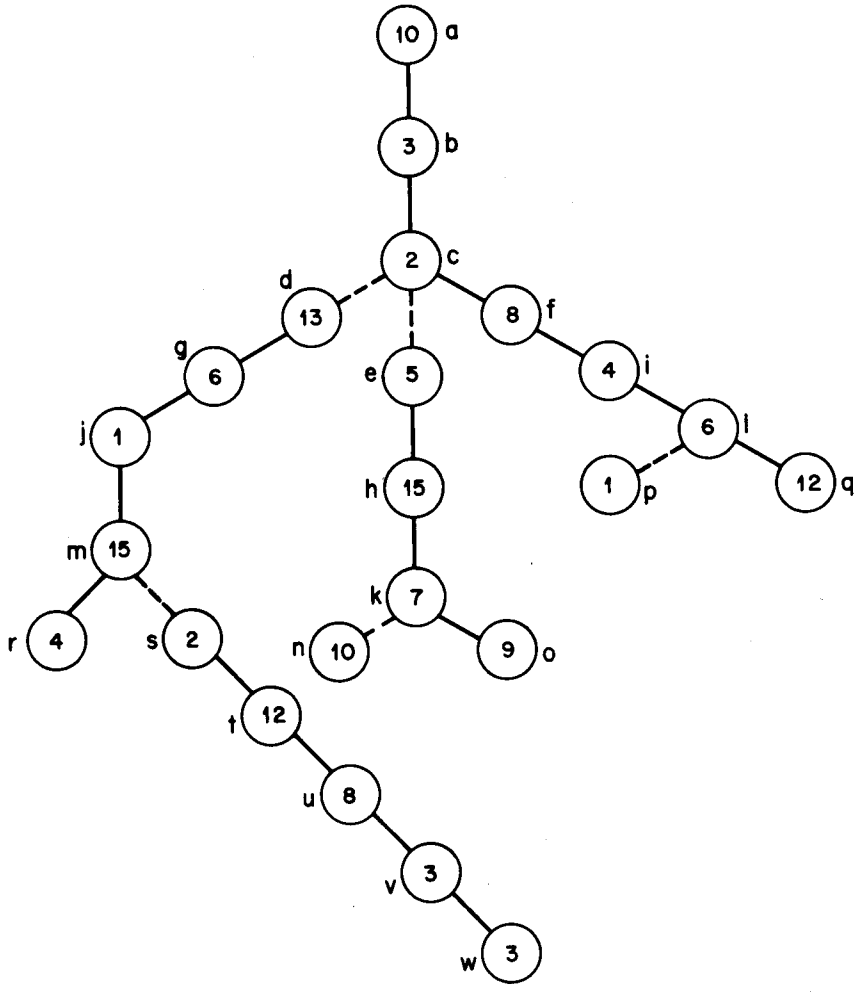


FIG. 19a. Virtual tree representing a link/cut tree: Actual tree. Numbers in nodes are costs. Dashed edges separate paths corresponding to solid subtrees in virtual tree.

Although splaying is defined as consisting of three passes, it can be carried out in a single bottom-up pass using four-node lookahead, and a one-pass implementation will be most efficient in practice. During the third pass, only zig and zig–zig splaying steps can occur: since all remaining ancestors of x except the root are left children, the zig–zag case is impossible.

We can analyze the amortized time for splaying using an extension of the argument in Section 2. Since we are only interested in proving an amortized bound of $O(\log n)$ for splaying, we assign a weight of one to every node. We define the size of a node to be the number of its descendants in its virtual tree (every node counts one since it has a weight of one), the rank of a node to be the binary logarithm of its size, and the potential of a virtual tree to be twice the sum of the ranks of its nodes. Note that this definition does not distinguish among middle, left, and right children.

As a measure of the time to splay at a node x, we use the original depth of x, which is equal to the number of rotations done. The amortized time for the first

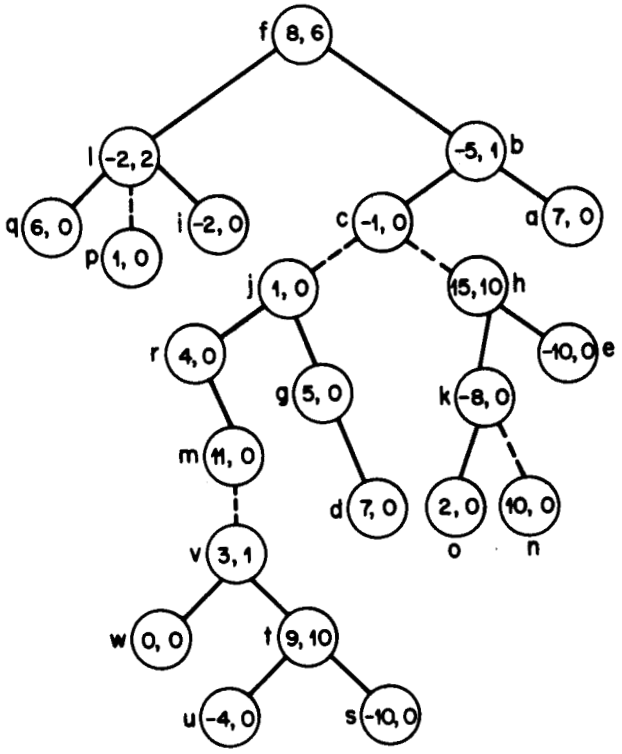


FIG. 19b. Virtual tree representing a link/cut tree: Virtual tree. First and second numbers in nodes are $\Delta cost$ and Δmin, respectively.

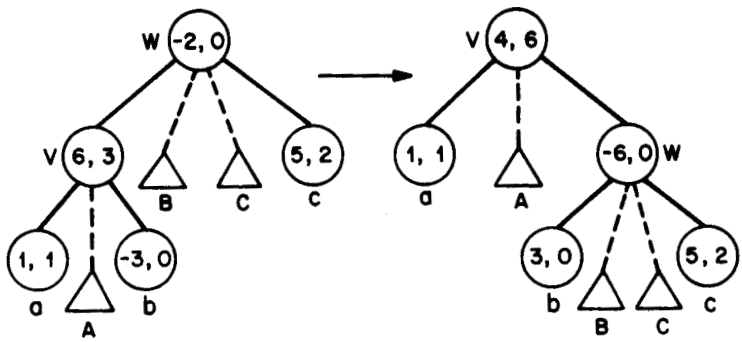


FIG. 20. Rotation of edge joining nodes v and w in a virtual tree. Subtrees of nodes a, b, and c are deleted for clarity.

pass is at most $6 \log n + k$, where k is the depth of x after the first pass. This bound follows from Lemma 1 by summing the splay times within each solid subtree; the sum telescopes to give the bound. Note that the constant factor is six instead of three because we have doubled the potential, but this does not affect the additive k term, which counts the number of times the zig case occurs, at most once per solid subtree. Since pass two does no rotations and does not change the potential (it only exchanges middle and left children), it has an amortized time of zero. To

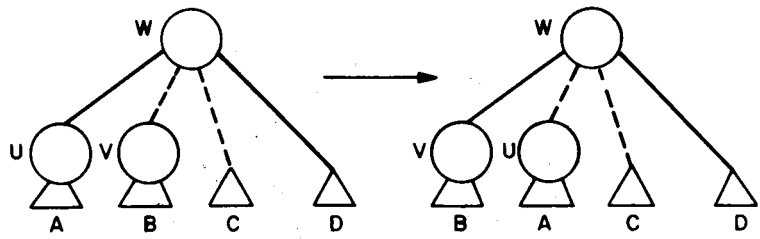


FIG. 21. A splice operation in a virtual tree. Node w is the root of a solid subtree. Node v becomes the left child of node w.

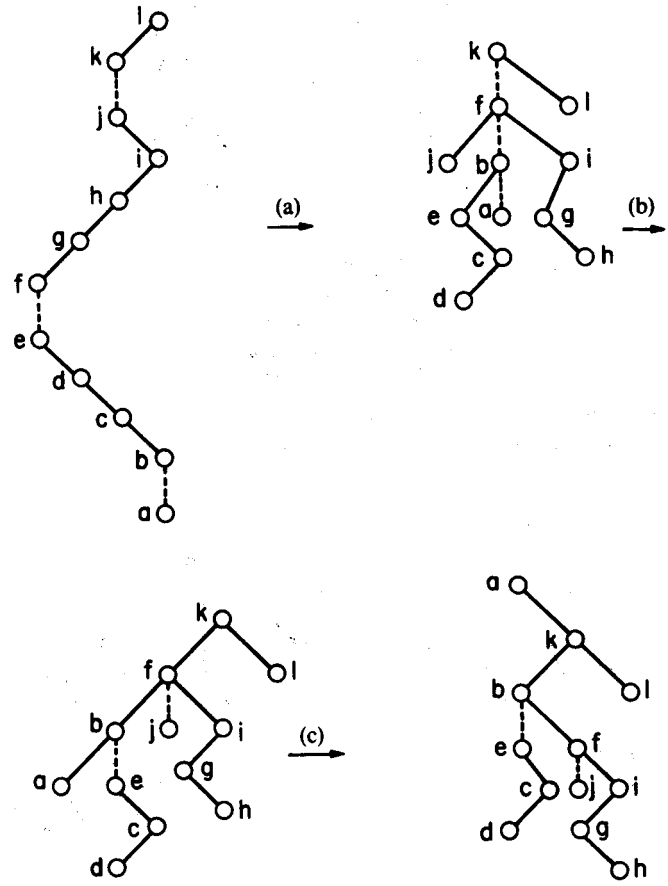


FIG. 22. A splaying operation at node a in a virtual tree. Subtrees of nodes along the access path are deleted for clarity. (a) First pass: splaying inside each solid subtree. (b) Second pass: splicing. (c) Third pass: splaying along final solid path.

account for the extra k rotations in pass one, we charge two for each of the k rotations in pass three. (This is the reason for doubling the potential.) Lemma 1 implies that the amortized time for pass three is at most $6 \log n + 2$, even at a cost of two per rotation. Thus the total amortized time for all three passes is $12 \log n + 2 = O(\log n)$. The constant factor of 12 can be reduced to 8 by combining the Lemma 1 analysis with a separate analysis of the number of splice operations [29, 34].

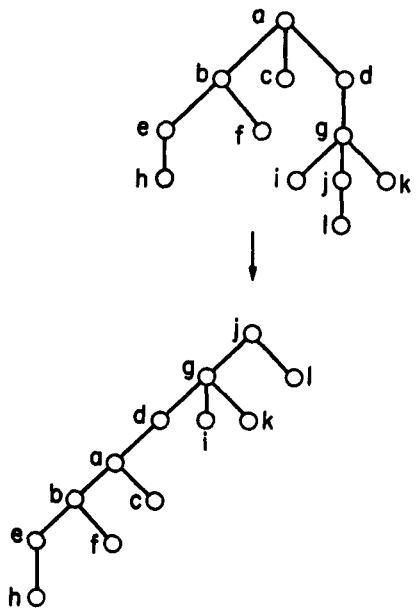


FIG. 23. Everting a link/cut tree at node *j*.

We can easily implement all the link/cut tree operations using splaying. To carry out *find cost*(v), we splay at node v and return *cost*(v). To carry out *find root*(v), we splay at v, follow right pointers until reaching the last node, say w, in the solid subtree containing v, splay at w, and return w. To carry out *find min*(v), we splay at v, use the $\Delta cost$ and Δmin fields to walk down from v to the last minimum-cost node after v in the same solid subtree, say w, splay at w, and return w. To carry out *add cost*(v, x), we splay at v, add x to $\Delta cost(v)$, and subtract x from $\Delta cost(left(v))$ if $left(v) \neq$ null. To carry out *link*(v, w), we splay at v and at w and make v a middle child of w (by defining $p(v)$ to be w). To carry out *cut*(v), we splay at v, add $\Delta cost(v)$ to $\Delta cost(right(v))$, and break the link between v and $right(v)$ by defining $p(right(v))$ = null and $right(v)$ = null. All these operations have an $O(\log n)$ amortized time bound. The initial potential (for a forest of one-node trees) is zero, and the potential remains nonnegative throughout any sequence of operations. Thus there is no net potential drop over a sequence of operations, and the total actual time is bounded by the total amortized time, which is $O(m \log n)$.

Our splaying method for virtual trees has several variants. For example, during the third pass we can use Allen and Munro's move-to-root heuristic instead of splaying [4]. This simplifies the program and preserves the $O(\log n)$ time bound, although a separate analysis is needed to bound splices [29, 34]. The third pass can even be eliminated entirely, although this complicates the implementation of the link/cut tree operations. The most practical variant of virtual tree splaying remains to be determined.

By suitably extending the virtual tree data structure, we can carry out other operations on link/cut trees in $O(\log n)$ amortized time. The most important of these is *evert*(v), which makes node v the root of its tree by reversing the path from v to the original root. (See Figure 23.) To implement this operation, we need to store an additional bit in each node indicating whether the meaning of its left and right pointers is reversed. (This bit must be stored in difference form.) We can also modify the structure so that the edges rather than the nodes have costs. If storage space is expensive, we can reduce the number of pointers needed per node from

three to two by using the representation of Figure 10. Sleator and Tarjan's paper [29] contains additional details about these results and several applications of link/cut trees.

7. *Remarks and Open Problems*

In this paper we have developed three new self-adjusting data structures: self-adjusting forms of binary search trees, lexicographic search trees, and link/cut trees. All of these structures are based on splaying, a restructuring heuristic that moves a designated node to the root of a tree through a sequence of rotations that approximately halves the depths of all nodes along the original path from the designated node to the root. Our structures are simpler than the corresponding balanced structures and at least as efficient in an amortized sense (to within a constant factor). The simplicity and adaptive behavior of our structures makes them potentially very useful in practice. For example, splay trees seem well-suited for implementation of first-fit storage allocation [21, 22], having an amortized bound of $O(\log n)$ per allocation or deallocation while possibly being even more efficient in practice. We intend to implement a splay-tree-based storage allocator and compare it to the "roving pointer" method of Knuth [21].

Our results illustrate but certainly do not exhaust the power of the twin paradigms of amortization and self-adjustment. Remaining open problems include the development of new forms of self-adjusting data structures and a tighter analysis of the total running time of a sequence of access operations on a splay tree. In particular, we may ask whether there is a self-adjusting form of B-tree, namely, a self-adjusting search tree with at most b children per node and an $O(\log_b n)$ amortized access time. Furthermore, we believe but cannot prove that the following three results hold for splay trees.

CONJECTURE 1 (DYNAMIC OPTIMALITY CONJECTURE). *Consider any sequence of successful accesses on an n-node search tree. Let A be any algorithm that carries out each access by traversing the path from the root to the node containing the accessed item, at a cost of one plus the depth of the node containing the item, and that between accesses performs an arbitrary number of rotations anywhere in the tree, at a cost of one per rotation. Then the total time to perform all the accesses by splaying is no more than $O(n)$ plus a constant times the time required by algorithm A.*

CONJECTURE 2 (DYNAMIC FINGER CONJECTURE). *The total time to perform m successful accesses on an arbitrary n-node splay tree is $O(m + n + \sum_{j=1}^{m-1} \log(|i_{j+1} - i_j| + 1))$, where for $1 \le i \le m$ the jth access is to item i_j (we denote items by their symmetric-order position).*

CONJECTURE 3 (TRAVERSAL CONJECTURE). *Let T_1 and T_2 be any two n-node binary search trees containing exactly the same items. Suppose we access the items in T_1 one after another using splaying, accessing them in the order they appear in T_2 in preorder (the item in the root of T_2 first, followed by the items in the left subtree of T_2 in preorder, followed by the items in the right subtree of T_2 in preorder). Then the total access time is $O(n)$.*

The dynamic optimality conjecture, if true, implies that splay trees are a form of universally efficient search tree: in an amortized sense and to within a constant factor, no other form of search tree can beat them. Our faith in the conjecture is based on the truth of the corresponding result for the move-to-front list update rule [30].