COMBINATORICA 5 (4) (1985) 367—378

SEQUENTIAL ACCESS IN SPLAY TREES TAKES LINEAR TIME

R. E. TARJAN

Received 10 July 1984

Sleator and Tarjan have invented a form of self-adjusting binary search tree called the *splay tree*. On any sufficiently long access sequence, splay trees are as efficient, to within a constant factor, as both dynamically balanced and static optimum search trees. Sleator and Tarjan have made a much stronger conjecture; namely, that on any sufficiently long access sequence and to within a constant factor, splay trees are as efficient as *any* form of dynamically updated search tree. This *dynamic optimality conjecture* implies as a special case that accessing the items in a splay tree in sequential order takes linear time, i.e. $O(1)$ time per access. In this paper we prove this special case of the conjecture, generalizing an unpublished result of Wegman. Our *sequential access theorem* not only supports belief in the dynamic optimality conjecture but provides additional insight into the workings of splay trees. As a corollary of our result, we show that splay trees can be used to simulate output-restricted deques (double-ended queues) in linear time. We pose several open problems related to our result.

1. Introduction

The objects of our study are *binary search trees*. A binary search tree is a binary tree whose nodes contain distinct items selected from a totally ordered universe, such that the items are arranged in symmetric order: if x is any node, all items in the left subtree of x are less than the item in x and all items in the right subtree of x are greater. (See Figure 1.) We can access any item e in such a tree by searching down from the root, using the following recursive procedure: If e is in the root, stop; the item has been found. If e is less than the item in the root, search recursively in the left subtree of the root. If e is greater than the item in the root, search recursively in the right subtree. If the tree is represented so that each node contains pointers to its left and right children, then the time to access an item is proportional to the depth of the node containing it. (In this paper we define the *depth* of a node x to be the number of nodes on the path from the tree root to x.)

If a binary search tree is to support a sequence of accesses efficiently, then the depths of the accessed nodes must be small, at least when averaged over the sequence. Sleator and Tarjan [5, 6] proposed a way of guaranteeing this by changing the structure of the tree after every access. Their restructuring operation, called *splaying*, consists of a sequence of *rotations*. A single rotation takes $O(1)$ time and preserves the symmetric order of the items. (See Figure 2.) To splay a tree at a node x we walk up the path from x to the tree root, performing rotations along the path. The rotations

AMS subject classification (1980): 68 E 05, 68 C 25, 05 C 05, 68 E 10

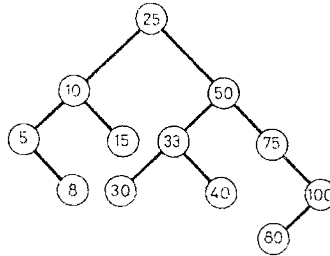

Fig. 1. A binary search tree

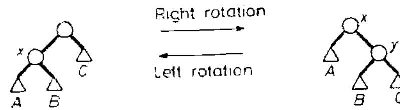

Fig. 2. Rotation of the edge joining nodes x and y. The triangles denote subtrees. The tree shown can be a subtree of a larger tree

are performed in pairs, in an order that depends upon the structure of the path. To be precise, we splay at x by repeating the following step until x is the root of the tree:

Splaying Step. Apply the appropriate one of the following cases. (See Figure 3.)

Zig Case. If the parent $p(x)$ of x is the tree root, rotate the edge joining x to $p(x)$. (This makes x the root and terminates the splaying.)

Zig-zig Case. If $p(x)$ is not the root and x and $p(x)$ are both left children or both right children, rotate the edge joining $p(x)$ to its parent and then rotate the edge joining x to $p(x)$.

Zig-zag Case. 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 to its parent and then rotate the edge joining x to its new parent.

In addition to moving the node x to the root of the tree, splaying roughly halves the depths of the other nodes on the splaying path, while increasing the depth of any node in the tree by at most two. (See Figure 4.) A binary search tree in which we splay after every access at the node containing the accessed item is called a *splay tree*.

It is easy to use splaying to implement such search tree update operations as insertion and deletion of items and joining and splitting [5, 6]. For the moment we shall restrict our attention to a sequence of access operations. Since the set of items is fixed and in one-to-one correspondence with the tree nodes, we shall regard the nodes themselves as the items. We shall identify the nodes by their symmetric-order numbers, from 1 to n. As a measure of the time required by a sequence of m splayings, we shall use the sum of the depths of the nodes at which the splayings occur, i.e. the total number of nodes on splaying paths.

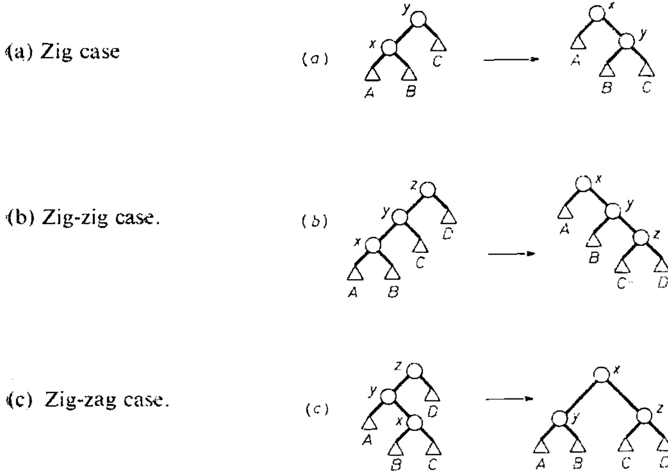

Fig. 3. A splaying step during a splaying at x. Each case has a symmetric variant (not shown)

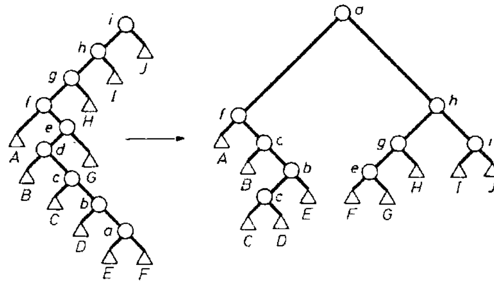

Fig. 4. Splaying at node a

The main known complexity result for splaying is Sleator and Tarjan's *access lemma*. Suppose we assign arbitrary positive weights w_i to the nodes i, $1 \leq i \leq n$. Let $W = \sum_{i=1}^{n} w_i$ be the total weight of all the nodes. Consider a sequence of m splaying operations starting with an arbitrary initial tree. Let $s(j)$ be the node at which the jth splaying takes place.

Access Lemma [5, 6]. *The total splaying time is* $O\left(\sum_{j=1}^{m} \log\left(\frac{W}{w_{s(j)}}\right) + \sum_{i=1}^{n} \log\left(\frac{W}{w_i}\right)\right)$. ∎

If we set all the weights equal to one, the access lemma gives a total time for m accesses of $O((m+n)\log n)$, implying that splay trees are as efficient, on long enough access sequences and to within a constant factor, as any of the many forms of balanced trees. Among other results, the access lemma also implies that on any sufficiently long access sequence and to within a constant factor, a splay tree is as efficient as an

optimum static search tree for the sequence. Sleator and Tarjan made an even stronger conjecture. Consider any sequence of accesses. Suppose we carry out the accesses by beginning with an arbitrary binary search tree and searching it from the root for the desired items in the desired order, with the proviso that between accesses we can change the tree by performing arbitrary rotations. The total cost of the access sequence is the total number of nodes on access paths plus the total number of rotations. Let $T(s)$ be the minimum total cost of access sequence s for any such binary search tree algorithm. Note that the algorithm is allowed to choose the initial tree.

Dynamic Optimality Conjecture. *If s is any access sequence, then the cost of performing s by using splaying is $O(T(s)+n)$, for any initial tree.*

Remark. The additive term of n in the conjecture accounts for the fact that the optimum algorithm is allowed to choose its initial tree, whereas the splaying algorithm is given the worst possible initial tree. Any tree can be converted into any other in at most $2n-2$ rotations [1]; thus allowing the optimal algorithm to choose its initial tree saves it only $O(n)$ time.

If this conjecture is true, splay trees are a form of universally efficient search tree. In this paper we prove the following special case of the conjecture:

Sequential Access Theorem. *If we access each of the nodes of an arbitrary initial tree once, in symmetric order, the total time spent is $O(n)$.*

To see that this theorem is a special case of the conjecture, observe that we can access n nodes in symmetric order in $O(n)$ time by beginning with a tree consisting of only a right path*, accessing the root, and repeating the following pair of steps $n-1$ times: rotate the edge joining the root to its right child; access the root. (See Figure 5.)

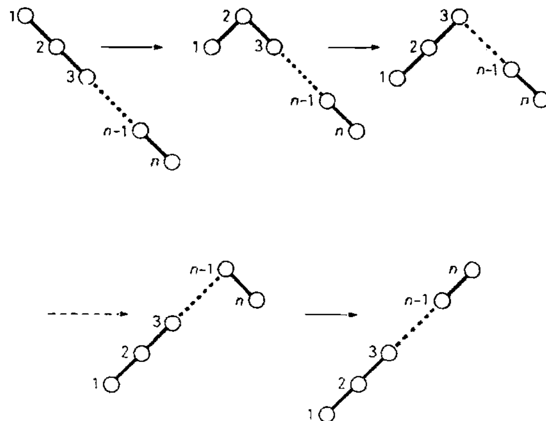

Fig. 5. Efficient sequential access

* By the *right path* of a binary tree we mean the path from the root through right children to the largest node in the tree. We define the *left path* symmetrically.

Our proof of the sequential access theorem is contained in Section 2. The proof is somewhat complicated and uses induction and averaging rather than the "potential" technique used by Sleator and Tarjan to prove the access lemma. As a corollary, we show in Section 3 that splay trees can be used to simulate output-restricted deques (double-ended queues) in linear time. This result confirms our belief that the splay tree, although it is a general-purpose data structure, is as efficient in special cases as customized special-purpose structures. In Section 4 we mention two open problems related to the sequential access theorem.

2. Proof of the sequential access theorem

Our goal is to analyze the time taken to successively splay at nodes $1, 2, \ldots, n$ in an arbitrary n-node tree. Let us restate the problem slightly. Figure 6 illustrates the effect of the ith splaying for $i \geqq 2$. Just before the splaying, the tree consists of the left path, containing the nodes $1, 2, \ldots, i-1$, and the right subtree of $i-1$, which contains i as its least node. Splaying at i restructures the right subtree, removing i and adding it to the left path.

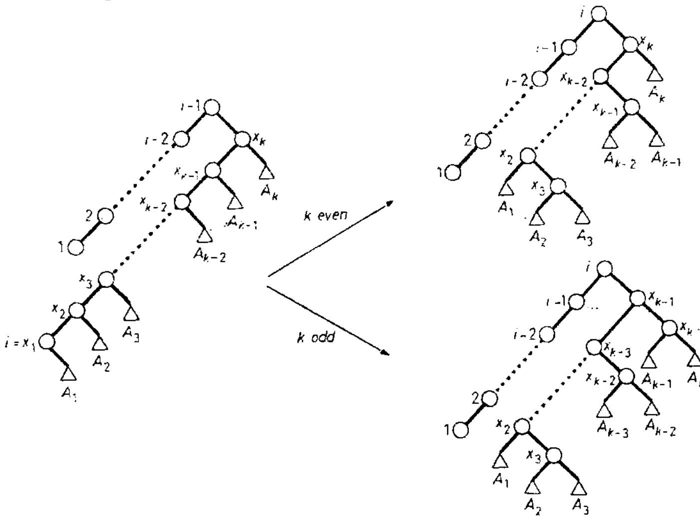

Fig. 6. The effect of the ith splaying. The splaying path contains nodes $x_1 = i, x_2, \ldots, x_k$

In our analysis we shall ignore the left path entirely and only keep track of the right subtree. Figure 7 illustrates the resulting reinterpretation of splaying, which we call *pseudo splaying*, or *p-splaying* for short: the least node is removed and the rest of the left path is halved. There are two cases, depending on whether the number of nodes on the left path is odd or even. We use the number of nodes on the left path as a measure of the p-splaying cost. The second and each successive p-splaying costs one less than the corresponding splaying. Thus the upper bound we shall derive on total p-splaying cost can be converted into a bound on total splaying cost by adding $n-1$.

We need several observations that follow immediately from an inspection of Figure 7. Let x be any node. A node y is a *right ancestor* of x if y is an ancestor of x

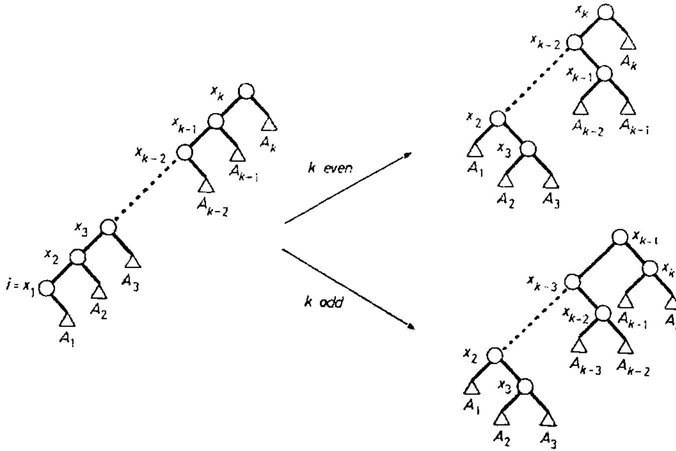

Fig. 7. The effect of the ith p-splaying. The p-splaying path consists of nodes $x_1, x_2, \ldots, x_k$

and $y \geqq x$. (Any node is an ancestor of itself; thus it is also a right ancestor of itself.) We denote the set of right ancestors of x by $A(x)$ and its cardinality by $a(x)$. We define $l(x)$ to be the number of nodes in $A(x)$ on the left path.

Lemma 1. *The subtree rooted at x is unaffected by successive p-splayings until x becomes a node on the left path.* ∎

Lemma 2. *As p-splayings occur, $A(x)$ decreases as a set, i.e. the relation $y \in A(x)$, once false, remains false. The cost of the p-splaying at x is at most $a(x)$, where $a(x)$ is computed at any time before the p-splaying.* ∎

Lemma 3. *Let a and a', l and l' denote the a and l functions just before and just after the p-splaying of some node $y < x$. Then $a'(x) \leqq a(x) - \lceil l(x)/2 \rceil + 1$ and $l'(x) \geqq \lfloor l(x)/2 \rfloor$.* ∎

A p-splaying of cost k knocks at least $\lceil k/2 \rceil - 1$ nodes off the left path. If it were true that every node could return to the left path only a constant number of times, then it would be easy to derive the sequential access theorem. Unfortunately, individual nodes can return arbitrarly often to the left path. To overcome this difficulty, we shall show that on the average a node returns to the left path only a constant number of times. We average only over certain nodes in the tree.

For any node x, let $f(x)$ be the depth of x when it first moves to the left path. (If x starts on the left path, $f(x)$ is its initial depth.) For k an arbitrary positive integer, let the k shallowest nodes on the left path be $x_1, x_2, \ldots, x_k$, from lowest to highest in symmetric order, and let y_i for $1 \leqq i \leqq k$ be the right child of x_i, if it exists. Let $F(k)$ be the maximum possible value of $\sum_{i=1}^{k} f(y_i)$ for any tree that actually has nodes $x_1, \ldots, x_k$ and $y_1, \ldots, y_{k-1}$. (If y_k is missing, we define $f(y_k) = 0$.)

We shall prove by induction that $F(k) = O(k)$, i.e. the average depth of the nodes y_i when they first reach the left path is a constant independent of k. Instead of

bounding $F(k)$ directly, we shall bound $G(k) = \max\{F(i) | 1 \leqq i \leqq k\}$, which is non-decreasing in k. The crucial result, stated in the next lemma, is a recursive bound on $G(k)$.

Lemma 4. *For* $k \geqq 3$, $G(k) \leqq \frac{5}{4} G(\lfloor k/2 \rfloor + 1) + 5(\lfloor k/2 \rfloor + 1)$.

Proof. Let $k \geqq 3$. Consider the first p-splaying. Its most important effect is to pair up all but at most two of the nodes y_i; for roughly half the values of i in the range $1 \leqq i \leqq k$, node x_i leaves the left path, getting y_{i-1} as its new left child and retaining y_i as its right child. Consider the effect of later p-splayings on the triple y_{i-1}, x_i, y_i. Eventually x_i moves to the left path. When it does, y_{i-1} and y_i are still its left and right children by Lemma 1. Furthermore y_{i-1} moves to the left path simultaneously with x_i. Thus $f(y_{i-1}) = f(x_i) + 1$.

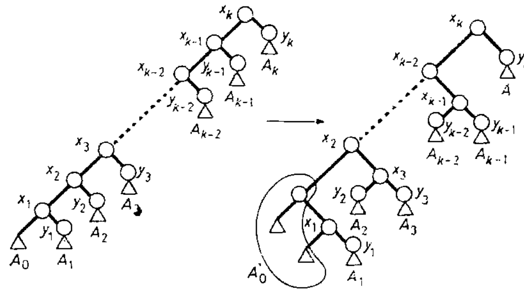

Fig. 8. The effect of p-splaying on the nodes x_i and y_i. There are actually six cases, depending on whether k is even or odd and on whether the length of the p-splaying path is k, greater than k and even, or greater than k and odd. The case shown is k even with a p-splaying path of length greater than k and even. Subtree A_0 is transformed into region A_0' by the p-splaying

When x_i reaches the left path, $a(y_i) = f(x_i)$ and $l(y_i) = f(x_i) - 1$. At least two more p-splayings (of y_{i-1} and x_i) occur before y_i moves to the left path. The first two of these reduce $a(y_i)$ to at most $f(x_i) - \lceil (f(x_i) - 1)/2 \rceil + 1 - \lceil (f(x_i) - 1)/2 \rfloor / 2 \rceil + 1 \leqq f(x_i)/4 + 3$ by Lemma 3 (applied twice). By Lemma 2, $f(y_i) \leqq f(x_i)/4 + 3$.

Combining our bounds we have $f(y_{i-1}) + f(y_i) \leqq \frac{5}{4} f(x_i) + 4$. Thus we can estimate the cost of the paired y_i's using the cost of the y_i's knocked off the left path by the first p-splaying, which we can in turn estimate using G.

The two nodes y_i that may not be paired up by the first p-splaying are y_1 and y_k. This p-splaying reduces $a(y_1)$ to at most $\lfloor k/2 \rfloor + 1$ by Lemma 3, implying that $f(y_1) \leqq \lfloor k/2 \rfloor + 1$. Node y_k, if not paired up, remains a right child of x_k, which remains the root.

Let $z_1, z_2, \ldots, z_j$ be the nodes among $x_1, x_2, \ldots, x_k$ and y_k that are right children of nodes on the left path after the first p-splaying. We have $j \leqq \lfloor k/2 \rfloor + 1$.

The discussion above implies

$$\sum_{i=1}^{k} f(y_k) \leqq \lfloor k/2\rfloor + 1 + \frac{5}{4}\sum_{i=1}^{j} f(z_i) + 4j \leqq \frac{5}{4}\, G(\lfloor k/2\rfloor + 1) + 5(\lfloor k/2\rfloor + 1),$$

which gives us the recurrence

$$G(k) \leqq \frac{5}{4}\, G(\lfloor k/2\rfloor + 1) + 5(\lfloor k/2\rfloor + 1),$$

since the initial tree was arbitrary. ∎

Lemma 5. $G(k) \leqq 8k$.

Proof. We use induction on k. Obviously $G(k) \leqq \sum_{i=1}^{k} i = k(k+1)/2$. An easy calculation shows $G(k) \leqq 8k$ for $k \leqq 30$. Let $k \geqq 30$ and suppose $G(i) \leqq 8i$ for $1 \leqq i < k$. Lemma 4 and the induction hypothesis imply $G(k) \leqq 10(\lfloor k/2\rfloor + 1) + 5(\lfloor k/2\rfloor + 1) \leqq 8k$. By induction $G(k) \leqq 8k$ for all k. ∎

Remark. A more complicated analysis will reduce the constant in Lemma 5, but this constant affects the constant we derive for the sequential access theorem only logarithmically.

Our proof of the sequential access theorem, though not as simple as possible, is designed to yield a small constant factor. We consider p-splaying at each of the nodes of an arbitrary n-node tree in symmetric order. Partition the nodes into *bands* by assigning node x to band 0 if $1 \leqq a(x) \leqq 2$ and to band i for $i \geqq 1$ if $2^{i-1}+2 \leqq a(x) \leqq 2^i+1$. The bands are chosen so that if a band i node with $i \geqq 1$ is on a splaying path, its band after the splaying is at most $i-1$. This follows from Lemma 3; any band zero node remains in band zero. We call a node *deep* if its band is five or greater and *shallow* otherwise.

We shall charge for certain events that occur during the p-splaying process. Specifically, we charge two units each time a p-splaying occurs, one unit each time the band of a shallow node changes, and 3.8 units each time a deep node becomes shallow. The total charge is $2n$ for the p-splayings plus $4n$ for the shallow band transitions plus 3.8n for the deep-shallow transitions, for a grand total of 9.8n.

Lemma 6. *The total charge is an upper bound on the cost of all the p-splayings.*

Proof. Consider a p-splaying path containing $k \leqq 33$ nodes. The two units charged to the p-splaying pay for the at most two band zero nodes. Each node in bands one through four changes band because of the p-splaying and is charged one unit. Each node in band five changes from deep to shallow and is charged 3.8 units. Since every node on the path is in bands zero through five, the cost of the p-splaying is no greater than the charge for events that occur during the p-splaying.

Consider a p-splaying path containing $k > 33$ nodes. Consider the situation just after the p-splaying. Let $x_1, x_2, \ldots, x_j$ be those nodes that were knocked off the left path by the splaying. We have $a(x_i) \leqq \lfloor k/2\rfloor + 1$, which implies by Lemma 5 that

$\sum_{i=1}^{j} G(x_i) \leqq 4k+8$. Suppose there are l nodes x_i that eventually return to the left path as deep nodes, i.e. have $f(x_i) \geqq 18$. Then $18l \leqq 4k+8$, which implies $l \leqq \frac{2}{9}k + \frac{4}{9}$.

There are $k-33$ nodes on the p-splaying path in bands six and higher, of which at least $(k-33)/2-1 = k/2-35/2$ are among the nodes x_i. Thus at least $k/2-35/2-l > \frac{5}{18}k - 18$ nodes in bands six and higher return to the left path only as shallow nodes. The charge for the corresponding deep-to-shallow transitions is at least $3.8\left[\frac{5}{18}k - 18\right] \geqq k - 64.4$. There is an additional charge of $3.8(16) = 60.8$ for the 16 band five nodes that become shallow because of the p-splaying, plus 15 for the 15 nodes in bands one through four that change band because of the p-splaying, plus 2 for the p-splaying itself. The total charge associated with the p-splaying is thus at least $k - 64.4 + 60.8 + 15 + 2 = k + 13.4 \geqq k$. ∎

Theorem 1. *The total cost of all n p-splaying operations is at most 9.8n.*

Proof. Immediate from Lemma 6. ∎

As noted at the beginning of this section, the bound in Theorem 1 underestimates the cost of n sequential splayings in an n-node tree by $n-1$. Thus we obtain a bound for the sequential access theorem of $10.8n$.

3. Splay trees as deques

A *deque (double-ended queue)* is an abstract data structure consisting of a list of items, on which the following operations can be performed:

push (e): Add item e to the front of the deque.

pop: Remove the front item from the deque and return it. If the deque is empty, this operation returns a special null node.

inject (e): Add item e to the rear of the deque.

eject: Remove the rear item from the deque and return it. If the deque is empty, this operation returns null.

If only *push* and *pop* operations are performed, the deque is a *stack*. If only *inject* and *pop* operations are performed, the deque is a *queue*. If only *push*, *pop*, and *inject* operations are performed, the deque is *output-restricted.*

We can implement a deque using a splay tree whose nodes are the deque items, with symmetric order in the tree corresponding to front-to-rear order in the deque. To carry out *push*(e), we make the current tree the right subtree of item e, which becomes the new root. To carry out *pop*, we follow left child pointers from the tree root until reaching a node x with no left child, perform a p-splaying at node x (thereby removing it from the tree), and return x. The implementations of *inject* and *eject* are symmetric.

We wish to study the efficiency of this implementation of deques. The sequential access theorem suggests that the time per deque operation should be $O(1)$ when amortized over a sufficiently long sequence of operations. Thus we make the following conjecture:

Deque Conjecture. *If we perform a sequence of m deque operations on an arbitrary n-node tree, the total time is $O(n+m)$.*

We shall prove this conjecture for output-restricted deques, i.e. no *eject* operations are allowed. We need two preliminary lemmas. Consider an n-node tree in which the first k nodes in symmetric order are colored white and the last $n-k$ nodes are colored black. We shall study the cost of p-splaying at each of the white nodes in symmetric order. To bound the cost, we count white nodes and black nodes on p-splaying paths separately.

Lemma 7. *Suppose the p-splaying path for node 1 contains k black nodes. After the p-splaying, at least $\lfloor (k-1)/2 \rfloor$ of these black nodes are right ancestors of no white nodes.*

Proof. Let $x_1, \dots, x_k$ be the black nodes on the p-splaying path, in increasing symmetric order. These are the k shallowest nodes on the left path. Let x_i for $i \geqq 2$ be knocked off the left path by the p-splaying. The new left subtree of x_i is the old right subtree of x_{i-1} which can contain no white nodes since all white nodes are less than x_{i-1}. Thus x_i satisfies the requirements of the lemma. There are at least $\lfloor (k-1)/2 \rfloor$ such nodes. ▮

Lemma 7 gives us a way to count black nodes on splaying paths. For purposes of counting white nodes, black nodes are irrelevant, as the next lemma shows. Let T be the original tree, and let $W(T)$, the *white tree* of T, be the tree formed from T consisting of the k white nodes, with y the parent of x in $W(T)$ if y is the nearest white ancestor of x in T. (If the parent of x in T is white, y is that parent; otherwise y is found by walking up from x until reaching a node less than x in symmetric order. In the latter case node x is the shallowest white node on the left path of the right subtree of y.) $W(T)$ is indeed a tree, because the nearest common ancestor of any two white nodes is white. The parenthetical remark implies that $W(T)$ is binary.

Lemma 8. *Let T' be formed from T by doing a single right rotation anywhere in the tree. If at least one of the nodes involved in the rotation is black, then $W(T')=W(T)$. Otherwise, let $W(T)'$ be formed from $W(T)$ by performing the same rotation as in T. Then $W(T')=W(T)'$.*

Proof. Referring to Figure 2, we see that the possible colors of nodes x and y are black-black, white-black, or white-white. (Since $x<y$, the black-white case is impossible). In the black-black case, the rotation in T doesn't affect the white tree: any white node in subtrees A, B, or C whose nearest white ancestor in T is an ancestor of y retains the same nearest white ancestor in T'. The same is true in the white-black case: all of subtree C must be black; any white node in A or B whose nearest white ancestor in T is x has x as its nearest white ancestor in T'; the removal of y as an ancestor of x doesn't affect the nearest white ancestor of x. In the white-white case, all of subtrees A and B are white; the right child of y in $W(T)$ is the shallowest white node on the left path of C. It is immediate that $W(T')=W(T)'$. ▮

Corollary 1. *Let T'' be formed from T by p-splaying at node 1 and let $W(T)''$ be formed from $W(T)$ similarly. Then the number of white nodes on the p-splaying paths is the same in both T and $W(T)$, and $W(T'') = W(T)''$.* ▮

Now consider an n-node arbitrary initial tree. Suppose we perform a sequence of m *push, pop,* and *inject* operations, with the proviso that none of the newly injected nodes is popped.

Lemma 9. *The deque operations take a total of $O(n+m)$ time.*

Proof. Color white all the original nodes in the tree as well as all pushed nodes, and color all injected nodes black. All the black nodes follow all the white nodes in symmetric order. Each *push* or *inject* operation takes $O(1)$ time. Thus we need only bound the total time of the *pop* operations, i.e. the total number of nodes on p-splaying paths. When a new node x is pushed, its set of right ancestors is $\{x\}$, and it becomes a right ancestor of no other nodes. It follows from Corollary 1 that all the results of Section 2 hold for the white tree as it changes because of deque operations. The number of white nodes on the p-splaying paths of nodes in the original tree is at most $9.8n$. (When a *push* occurs, what is left of the original tree is unaffected until the pushed node is removed by a *pop*.) If a *pop* causes a p-splaying along a path containing k black nodes, then by Lemma 7 at least $\lfloor (k-1)/2 \rfloor \leq k/2 - 1$ of these black nodes are on no later splaying path. It follows that the number of black nodes on p-splaying paths is at most two per black node plus two per p-splaying, for a total of at most $2(n+m)$. Combining estimates, we find that the total number of nodes on all p-splaying paths is at most $11.8n+3m$. The theorem follows. ▮

Theorem 2. *Consider a sequence of m output-restricted deque operations performed on an arbitrary n-node tree. The deque operations require a total of $O(n+m)$ time.*

Proof. We divide the deque operations into *epochs* such that the restriction of Lemma 9 holds for each epoch. The first epoch ends when the last node in the original tree is popped. The $i+1^{st}$ epoch ends when the last node in the original tree of epoch i is popped. Nodes that are white during one epoch are gone during the next epoch, and nodes that are black during one epoch are white during the next epoch. The theorem follows from Lemma 9 by summing over epochs. ▮

4. Remarks and open problems

Theorems 1 and 2 strengthen our belief that the splay tree, even though it is a general-purpose data structure, adapts sufficiently well to the usage pattern that it can be competitive with special-purpose structures customized to fit their usage. There are several open problems related to our work. The first is to prove the deque conjecture. If true, this conjecture implies that implementing a deque using a splay tree is as efficient, in an amortized sense and ignoring constant factors, as the standard array and doubly-linked list representations [3, 7]. In situations where insertions and deletions in the interior of the deque are occasionally necessary, as in discrete event simulation [2, 4], the splay tree representation may be a good one, since it supports more powerful operations apparently without degrading the efficiency of simple ones.

Proving the deque conjecture requires studying the effect of intermixed splayings on both ends of the tree. As an aid to this study, it would be nice to understand the sequential access theorem better. In particular, it would be useful to have a potential-based proof. (See [8] for a discussion of the idea of potential.) Such a proof might be more easily extendible to a proof of the deque conjecture than the present one seems to be.

Another related question was suggested by Danny Sleator (private communication.) Call the pair of rotations performed during the zig-zig case of a splaying step a *turn*. (See Figure 9.)

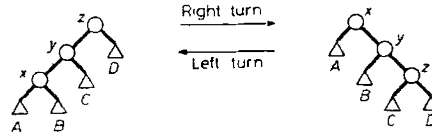

Fig. 9. A turn. The tree shown can be a subtree of a larger tree

Turn Conjecture (Sleator). *If we begin with an arbitrary n-node tree and perform an arbitrary sequence of intermixed right turns and right single rotations, then the total number of turns is $O(n)$. (The number of single rotations can be as high as $\binom{n}{2}$.)*

Acknowledgement. My thanks to Adriano Garsia, Neil Sarnak, Danny Sleator, and Michelle Wachs for extensive discussions that stimulated my thinking and helped to clarify the ideas presented here. My special appreciation to Mark Wegman, whose proof of the sequential access theorem for the special case of a complete binary tree was the spur for my efforts.

References

[1] K. Culik II and D. Wood, A note on some tree similarity measures, *Info. Proc. Lett.* **15** (1982), 39—42.
[2] G. Gordon, *System Simulation*, Prentice-Hall, Englewood Cliffs, NJ. 1969.
[3] D. E. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA, 1973.
[4] T. H. Naylor, J. L. Balinty, D. S. Burdick, and K. Chu, *Computer Simulation Techniques*, Wiley, New York, NY, 1966.
[5] D. D. Sleator and R. E. Tarjan, Self-adjusting binary trees, *Proc. Fifteenth Annual ACM Symp. on Theory of Computing* (1983), 235—245.
[6] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *J. Assoc. Comput. Mach.*, **32** (1885), 652—686.
[7] R. E. Tarjan, *Data Structures and Network Algorithms*, CMBS 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
[8] R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Disc. Meth.*, *to* **6** (1985), 545—568.

Robert Endre Tarjan
AT & T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974
U.S.A.