SIAM J. COMPUT.
Vol. 18, No. 1, pp. 56-67, February 1989

LOWER BOUNDS FOR ACCESSING BINARY SEARCH TREES WITH ROTATIONS*

ROBERT WILBER†

Abstract. Two methods are given for obtaining lower bounds on the cost of accessing a sequence of nodes in a symmetrically ordered binary search tree, in a model where rotations can be done on the tree and the entire sequence is known before accessing begins (but the accesses must be done in the order given). For example, it can be proven that the bit-reversal permutation requires $\Theta(n \log n)$ time to access in this model. It is also shown that the expected cost of accessing random sequences in this model is the same as it is for the case where the tree is static.

Key words. data structures, binary search tree, rotation, lower bound

AMS(MOS) subject classification. 68P

1. Introduction. A binary search tree is a binary tree whose nodes are distinct members of some totally ordered set and for which the nodes are in symmetric order, i.e., each node is greater than all nodes in its left subtree and less than all nodes in its right subtree. Binary search trees can efficiently support such operations as insert, find, delete, find minimum, find maximum, split, and join. A variety of algorithms for maintaining binary search trees have been proposed that provide some combination of these operations [1]-[3], [5]-[9]. Most of these algorithms use *rotations* to modify binary trees. If node u is the left child of node v, then u is rotated over v by making v the new right child of u; this makes the old right child of u (if there is one) the new left child of v, and the old parent of v (if there is one) the new parent of u. The mirror image operation applies if u is the right child of v. A rotation preserves the symmetric order of the tree.

Here we consider the following problem. Given an initial binary search tree T_0, and a sequence s of nodes in T_0, what is the minimum time required to access the nodes in s (in the order given) when we are allowed to do rotations on the tree? We assume that the entire sequence s is known before we start (i.e., the algorithm is "offline"). The cost of assessing a node at depth d (where the root is at depth zero) is $d+1$, and the cost of a rotation is 1. One easy observation is that by losing at most a factor of 2, we may assume that we always access a node by rotating it to the root in some way. For, given an arbitrary rotation algorithm, we may simulate as follows. Whenever the algorithm accesses a node at depth d (at cost $d+1$), we instead do d rotations to bring the node to the root, access it there (at cost 1), and then do the reverse sequence of d rotations to bring the node back to where it was, for a total cost of $2d+1$. So without loss of generality we will consider only those algorithms that access a node by rotating it to the root. Such an algorithm shall be called a *standard search algorithm.* We let $\chi(s, T_0)$ denote the minimum cost of accessing the nodes in sequence s, starting from tree T_0, by a standard search algorithm. We are concerned with finding lower bounds for $\chi(s, T_0)$.

This problem is motivated by a conjecture of Sleator and Tarjan concerning their *splay algorithm* [9]. The splay algorithm is an online algorithm for maintaining binary

* Received by the editors March 11, 1987; accepted for publication (in revised form) March 17, 1988. A preliminary version of this paper appeared in the Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 61-70.

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

search trees using rotations in a way that has provably good amortized behavior. Although any single operation on an n node tree can take $\Omega(n)$ time, Sleator and Tarjan were able to prove the following surprising fact.

THEOREM 0 (Sleator and Tarjan). *Let V be a set of n keys. Let s be a sequence of keys in V and let t be the minimum time required to access s on any static tree (i.e., a tree that is fixed for sequence s and cannot be modified as the sequence is accessed). Let T_0 be an arbitrary binary search tree whose nodes are the keys in V. Then the time required by the splay algorithm to access s starting from tree T_0 is $O(t+n)$.*

Thus the splay algorithm does essentially as well as the best static algorithm, even though the splay algorithm is an online algorithm. Sleator and Tarjan have conjectured that their splay algorithm does as well as the best offline dynamic binary tree algorithm (an algorithm that does rotations and is allowed to pick its initial tree after seeing the sequence). That is, for any sequence s and for any initial tree T_0 they conjecture that the time used by the splay algorithm to access sequence s is $O(\chi(s, T_0)+n)$.

This "dynamic optimality conjecture" implies that, for any initial n node tree, the splay algorithm is able to access the nodes of the tree in sorted order in $O(n)$ time. This corollary of the conjecture has been proved by Tarjan [11]. Even this very restricted case required a difficult proof.

In order to prove (or disprove) the dynamic optimality conjecture, it may be useful to have bounds on $\chi(s, T_0)$. Here two methods for getting lower bounds on $\chi(s, T_0)$ are described. The first method, described in the next section, requires the use of an auxiliary structure called a *lower bound tree.* The bound obtained is a function of the lower bound tree as well as of s and T_0. This method can achieve tight bounds for some particular sequences, such as a $\Theta(n \log n)$ bound for the bit reversal permutation, and can also be used to get tight expected time bounds for sequences generated at random. However, because of the somewhat artificial introduction of a lower bound tree, the method seems likely not to provide tight bounds in general.

Culik and Wood [4] showed that the number of rotations needed to convert one n node symmetrically ordered binary tree into another is at most $2n-2$. Sleator, Tarjan, and Thurston [10] proved that the bound is at most $2n-6$, and that this is tight for infinitely many n. Thus $\chi(s, T_0)$ does not have a strong dependence on T_0; the choice of initial tree cannot change the cost by more than $2n-6$. The second lower bound method, given in § 3, makes use of this—not only is there no dependence upon a lower bound tree, there is no dependence upon the initial tree T_0. The second lower bound method can also be used to get the $\Theta(n \log n)$ bound for accessing the bit reversal permutation (although with a worse constant factor than the first method yields). It seems more likely to give tight bounds for particular sequences than the first method, although it has not proven to be as suitable for analyzing the expected cost of accessing a random sequence.

2. The first lower bound.

For a sequence of integers s and integer i we let $s+i$ denote the sequence obtained by adding i to each element of s. If s is an integer sequence and $x \leqq y$, then $s|_x^y$ denotes the subsequence of s consisting of the elements in the interval $[x, y]$.

We will assume without loss of generality that the nodes of search trees are consecutive integers. If u is a child of v the rotation of u over v is denoted by the pair (u, v). The action of a standard search algorithm on some particular access sequence and some particular initial search tree is completely described by the sequence of rotations generated by the algorithm. If R is a sequence of rotations and $x \leqq y$, then $R|_x^y$ denotes the subsequence of R consisting of the pairs in $[x, y]^2$.

Let T be a search tree whose nodes are the integers in $[j, k]$, for some $j \leqq k$. Then a *lower bound tree for* T is a symmetrically ordered binary tree with $2(k-j)+1$ nodes whose leaves are the integers $j, j+1, \cdots, k$ and whose internal nodes are the half-integers $j+\frac{1}{2}, j+\frac{3}{2}, \cdots, k-\frac{1}{2}$. We define an integer function $\Lambda_1(s, T_0, \Upsilon)$ such that for any search tree with consecutive integer nodes T_0, any sequence s of nodes in T_0, and any lower bound tree Υ for T_0, we have $\chi(s, T_0) \geqq \Lambda_1(s, T_0, \Upsilon)$. For fixed s and T_0 the function Λ_1 will have different values for different lower bound trees Υ; this lower bound method requires judgement in choosing a lower bound tree that will maximize the value of Λ_1.

We now describe how to compute $\Lambda_1(s, T_0, \Upsilon)$. Let m be the length of sequence s and let U be the set of internal nodes of Υ. For each node $u \in U$ we compute its *score*, $\lambda(u)$, as follows. Let l and r be the leftmost and rightmost leaf nodes, respectively, in the subtree of Υ with root u. Let $\sigma' = s|_l^r$ and let h be the length of σ'. Let v be the lowest common ancestor in T_0 of the nodes in $[l, r]$ (we always have $v \in [l, r]$). Let sequence $\sigma = \sigma_0, \sigma_1, \cdots, \sigma_h$, where $\sigma_0 = v$ and, for $i \in [1, h]$, $\sigma_i = \sigma_i'$. Say that an integer $i \in [1, h]$ is a *u-transition* if $\sigma_{i-1} < u$ and $\sigma_i > u$ or if $\sigma_{i-1} > u$ and $\sigma_i < u$. We define $\lambda(u) = |\{i \in [1, h]: i \text{ is a } u\text{-transition}\}|$. The function Λ_1 is computed as

$$\Lambda_1(s, T_0, \Upsilon) = m + \sum_{u \in U} \lambda(u).$$

THEOREM 1. *Let T_0 be a search tree with consecutive integer nodes, let Υ be a lower bound tree for T_0, and let s be a sequence of nodes in T_0. Then $\chi(s, T_0) \geqq \Lambda_1(s, T_0, \Upsilon)$.*

Before we can prove Theorem 1 we need to define a kind of generalized subtree operation. Let T be a search tree with integer nodes, and let V be the set of nodes in T. For any node $v \in V$, the *left inner path of* v is the path that starts at the left child of v and proceeds downward along the right child links until a node is reached that has no right child. If v has no left child its left inner path is the null path. Likewise, the *right inner path* of v is the path that starts at the right child of v and proceeds along the left links. Let $x \leqq y$. The *restriction of T to the interval* $[x, y]$, denoted $T|_x^y$, is defined as follows. The set of nodes of $T|_x^y$ is $V' = V \cap [x, y]$. Let $v \in V'$ and let P be v's left inner path in T. If P has no nodes in V' then v has no left child in $T|_x^y$. Otherwise the first node in P that is in V' is the left child of v in $T|_x^y$. The right child of v in $T|_x^y$ is defined similarly—it is the first node in V' along the right inner path of v in T, if such a node exists, and otherwise v has no right child in $T|_x^y$. Clearly $T|_x^y$ is a symmetrically ordered binary tree, and its root is the lowest common ancestor in T of the nodes in V'. Also, if u is a child of v in T and $u, v \in V'$, then u is a child of v in $T|_x^y$. More generally, if u is a descendent of v in T and $u, v \in V'$, then u is a descendent of v in $T|_x^y$.

LEMMA 2. *Let T_1 be a search tree with integer nodes, let node u be a child of node v in T, and let T_2 be the tree that results when u is rotated over v in T_1. Let $x \leqq y$. If either u or v is not in the interval $[x, y]$ then $T_1|_x^y = T_2|_x^y$. On the other hand, if u and v are both in $[x, y]$ then $T_2|_x^y$ is the tree obtained by rotating u over v in $T_1|_x^y$.*

Proof. Let $T_1' = T_1|_x^y$ and $T_2' = T_2|_x^y$. Consider the case where at least one of u or v is not in $[x, y]$. These are four subcases: u is the left child of v in T_1 and $u < x$, u is the left child of v and $v > y$, u is the right child of v and $u > y$, and u is the right child of v and $v < x$. The fourth case is the mirror image of the second case, the second case is the time reversal of the third case, and the third case is the mirror image of the first case, so we need consider only the first case, where u is the left child of v in T_1 and $u < x$. Let w be any node in $[x, y]$. We must show that the children of w in T_1' are the same as in T_2'. Since $u < x$ and $w \geqq x$ node u cannot be in the right subtree of

w in T_1. Since v is the parent of u, v cannot be in the right subtree of w either. Thus the rotation of u over v has no effect on the right subtree of w, so the right child of w in T_2' is the same as the right child of w in T_1'. Let P be the left inner path of w. If $w \neq v$ and v is not in P then P is unaffected by the rotation, so the left child of w in T_1' is the same as it is in T_2'. If v is in P then the effect of the rotation is to insert u into P, and if $v = w$ the effect is to remove u from P, but since $u \notin [x, y]$ this does not change the left child of w in the restricted tree. Thus the rotation does not change the restricted tree.

The case where u and v are both in $[x, y]$ is just as easy and it is left to the reader. □

Proof of Theorem 1. Let m be the length of sequence s. After a node has been rotated to the root of the search tree the cost of accessing it is 1, so the access cost for sequence s, exclusive of the cost of all rotations, is m. Therefore it suffices to show that $\sum_{u \in U} \lambda(u)$, where U is the set of internal nodes of Υ, is a lower bound on the number of rotations that must be carried out by any standard search algorithm.

Let n be the number of nodes in the search tree. We use induction on n. If $n = 1$ then there is only one possible lower bound tree—the tree with a single leaf node and no internal nodes. In that case $\sum_{u \in U} \lambda(u) = 0$ and there is nothing to prove. So assume $n \geqq 2$. Let R be the sequence of rotations generated by some standard search algorithm that accesses the sequence s starting from tree T_0. Let r be the length of R and, for an integer $t \in [1, r]$, let T_t be the tree that results from applying R_t to T_{t-1}. Let w be the root of Υ and let Υ^1 and Υ^2 be the left and right subtrees, respectively, of w. Let $R^1 = R|_{-\infty}^{w}$ and $R^2 = R|_{w}^{\infty}$. Let M be the subsequence of R obtained by deleting those rotations that are in R^1 or R^2. The sequences R^1, R^2, and M are disjoint, so we have $r = |R^1| + |R^2| + |M|$.

For $i = 1, 2$ let U^i be the set of internal nodes of Υ^i. Let $s^1 = s|_{-\infty}^{w}$ and let $s^2 = s|_{w}^{\infty}$. For $t \in [0, r]$, let $T_t^1 = T_t|_{-\infty}^{w}$ and let $T_t^2 = T_t|_{w}^{\infty}$. Let $T_0^{1\prime}, T_1^{1\prime}, \cdots, T_{r_1}^{1\prime}$ be the sequence of search trees obtained by deleting from the sequence $T_0^1, T_1^1, \cdots, T_r^1$ those trees T_t^1 such that $T_t^1 = T_{t-1}^1$. By Lemma 2, $T_t^{1\prime}$ can be derived from $T_{t-1}^{1\prime}$ by applying rotation R_t^1, for all $t \in [1, r_1]$. Also, for all $t \in [0, r]$ if the root v of T_t is less than w then v is also the root of T_t^1. Therefore R^1 is a sequence of rotations that, starting from T_0^1, brings to the root the successive nodes in s^1. Tree Υ^1 is a lower bound tree for T_0^1, and the scores assigned to its internal nodes are precisely the scores obtained in the computation of $\Lambda_1(s^1, T_0^1, \Upsilon^1)$. So by the induction hypothesis, $|R^1| \geqq \sum_{u \in U^1} \lambda(u)$. Similarly, R^2 is a sequence of rotations that, starting from T_0^2, brings to the root the successive nodes in s^2, and Υ^2 is a lower bound tree for T_0^2. So $|R^2| \geqq \sum_{u \in U^2} \lambda(u)$.

Let σ be the sequence obtained by concatenating the root of T_0 with s. Let integer i be a w-transition. Without loss of generality, we may assume that $\sigma_{i-1} < w$ and $\sigma_i > w$. After the $(i-1)$th access, σ_{i-1} is at the root of the search tree and after the ith access, σ_i is at the root. Thus between the $(i-1)$th and ith accesses there must be a time t such that the root of T_{t-1} is less than w and the root of T_t is greater than w. Let y be the root of T_{t-1} and z be the root of T_t. We must have $R_t = (z, y) \in M$. Thus there is at least one rotation in M for every w-transition so $|M| \geqq \lambda(w)$.

Putting it all together, we have

$$r = |R^1| + |R^2| + |M| \geqq \sum_{u \in U^1} \lambda(u) + \sum_{u \in U^2} \lambda(u) + \lambda(w) = \sum_{u \in U} \lambda(u). \qquad \square$$

As an application of Theorem 1, we show that the bit reversal permutation on n nodes requires $\Theta(n \log n)$ time to access. Let k be a nonnegative integer and let $i \in [0, 2^k - 1]$. If the k-binary representation of i is $b_{k-1} b_{k-2} \cdots b_1 b_0$ then the *k-bit reversal of* i, denoted by $\text{br}_k(i)$, is the integer whose k-bit binary representation is

$b_0b_1 \cdots b_{k-2}b_{k-1}$. The bit reversal permutation on $n = 2^k$ nodes is the sequence $B^k = \mathrm{br}_k(0), \mathrm{br}_k(1), \cdots, \mathrm{br}_k(n-1)$.

COROLLARY 3. *Let k be a nonnegative integer and let $n = 2^k$. Let T_0 be any search tree with nodes $0, 1, \cdots, n-1$. Then $\chi(B^k, T_0) \geqq n \log n + 1$.*

(Note. All logarithms in this paper are base 2.)

PROOF. Let Υ be the balanced lower bound tree for T_0. That is, the root of Υ is $w = \frac{1}{2}n - \frac{1}{2}$ and in general at depth $d \in [0, k-1]$ the internal nodes are $(2i-1)2^{-(d+1)}n - \frac{1}{2}$, for $1 \leqq i \leqq 2^d$. Let U be the set of internal nodes of Υ and let $\lambda(u)$ be the score assigned to a node $u \in U$ for sequence B^k and initial tree T_0. By Theorem 1, we have $\chi(B^k, T_0) \geqq \Lambda_1(B^k, T_0, \Upsilon) = n + \sum_{u \in U} \lambda(u)$. We show by induction that $\sum_{u \in U} \lambda(u) \geqq n \log n - n + 1$.

If $k = 0$ then $n \log n - n + 1 = 0$ and there is nothing to prove. Suppose $k \geqq 1$. Let Υ^1 and Υ^2 be the left and right subtree of Υ, respectively, and let U^1 and U^2 be the sets of their internal nodes. The elements of sequence B^k are alternatively less than w and greater than w. Thus, regardless of the choice of T_0, we have $\lambda(w) \geqq n - 1$. We have $B^k|_{-\infty}^{w} = B^{k-1}$ and $B^k|_{w}^{\infty} = B^{k-1} + 2^{k-1}$. Also, Υ_1 and Υ_2 are balanced lower bound trees, so by induction $\sum_{u \in U^i} \lambda(u) \geqq \frac{1}{2}n \log(\frac{1}{2}n) - \frac{1}{2}n + 1$, for $i = 1, 2$. Thus

$$\sum_{u \in U} \lambda(u) = \sum_{u \in U^1} \lambda(u) + \sum_{u \in U^2} \lambda(u) + \lambda(w) \geqq n \log n - n + 1. \qquad \square$$

We now consider the expected cost of accessing a sequence generated at random. If x is a random variable we denote the expected value of x by $E(x)$. For $i \in [1, n]$ let $p_i \geqq 0$ and let $\sum_{i=1}^n p_i = 1$. Suppose sequence s is generated by choosing m integers in $[1, n]$ independently and at random, where the probability that k is chosen is p_k. If we construct the optimum static tree for the given probabilities the expected cost of accessing s by an algorithm that does not change the tree is $\Theta(m(1 + \sum_{i=1}^n p_i \log(1/p_i)))$ [6]. We now show that the same expected cost applies to offline algorithms that do rotations.

THEOREM 4. *Let sequence s be generated in the manner described above. Then for any initial tree T_0 we have $E(\chi(s, T_0)) = \Theta(m(1 + \sum_{i=1}^n p \log(1/p_i)))$.*

Proof. The upper bound follows from the bound for the static case. For the lower bound it is convenient to assume that $p_i > 0$ for all i. If $p_k = 0$ for some k then that integer will never be selected and it does not contribute to the sum of the $p_i \log(1/p_i)$s, so it can be ignored. We use a lower bound tree Υ that is balanced with respect to probabilities. That is, for each internal node u of Υ we make the probability of accessing a leaf in the left subtree of u as close as possible to the probability of accessing a leaf in the right subtree of u. More precisely, a probability-balanced tree with leaves j through k is constructed recursively as follows (Υ is constructed by applying this procedure to leaves 1 through n). If $j = k$ then the tree has the single leaf node j and we are done. Otherwise, normalize the probabilities by setting $P = \sum_{i=j}^k p_i$ and, for $i \in [j, k]$, $p'_i = p_i/P$. Let c be the least integer such that $\sum_{i=j}^c p'_i \geqq \frac{1}{2}$. If $\sum_{i=j}^{c-1} p'_i + \frac{1}{2}p'_c < \frac{1}{2}$, then set $b = c$, otherwise set $b = c - 1$. The root of the tree is $u = b + \frac{1}{2}$. The left subtree of u is the probability-balanced tree for leaves j through b, and the right subtree of u is the probability-balanced tree for leaves $b+1$ through k. The assumption that $p_i > 0$ for all i ensures that the sets of leaves of the left and right subtrees of u are nonempty. The integer c is called the *center leaf* of u.

Let U be the set of internal nodes of Υ. We modify the accounting for the lower bound as follows. For $u \in U$, let $\alpha(u)$ be the number of times the center leaf of u occurs in sequence s. The *modified score* of u is $\lambda'(u) = \lambda(u) + \frac{1}{2}\alpha(u)$. Each leaf node is the center leaf of at most two internal nodes, so $\sum_{u \in U} \alpha(u) \leqq 2m$. Thus $\Lambda_1(s, T_0, \Upsilon) = m + \sum_{u \in U} \lambda(u) \geqq \sum_{u \in U} \lambda'(u)$.

Suppose we step through the successive elements of s, on the ith step computing the modified scores of the internal nodes on the basis of T_0 and the subsequence $s_1, s_2, \cdots, s_i$. For $u \in U$ let $\lambda_i'(u)$ be the increase in $\lambda'(u)$ due to step i. Let $\lambda_i(u)$ be the increase in $\lambda(u)$, and let $\alpha_i(u)$ be the increase in $\alpha(u)$. Let the leftmost leaf in the subtree with root u be l_u and let the rightmost leaf be r_u. Let $q_1 = \sum_{l_u \leqq j < u} p_j$ and let $q_2 = \sum_{u < j \leqq r_u} p_j$. Let $P = q_1 + q_2$ be the probability that s_i is a leaf of the subtree with root u. For $k = 1, 2$ let $q_k' = q_k / P$. We have $E(\lambda_i(u)) = P\tau$, where τ is the conditional probability that i is a u-transition, given that $s_i \in [l_u, r_u]$. If the last node accessed under u was in u's left subtree then $\tau = q_2'$, and if it was in u's right subtree then $\tau = q_1'$. (If there was no node previously accessed under u then τ is either q_2' or q_1', depending upon whether the lowest common ancestor of nodes l through r in T_0 is less than or greater than u.) In any case $\tau \geqq \min(q_1', q_2')$. Let c be the center leaf of u, and let $p_c' = p_c / P$ be the conditional probability that $s_i = c$, given that $s_i \in [l_u, r_u]$. We have $E(\alpha_i(u)) = Pp_c'$. Thus $E(\lambda_i'(u)) = E(\lambda_i(u)) + \frac{1}{2}E(\alpha_i(u)) \geqq P(\min(q_1', q_2') + \frac{1}{2}p_c')$.

By the construction of Y, we always have $\min(q_1', q_2') + \frac{1}{2}p_c' \geqq \frac{1}{2}$. So $E(\lambda_i'(u)) \geqq \frac{1}{2}P$. Summing over all m elements of s, we have $E(\lambda'(u)) \geqq \frac{1}{2}mP$. Thus

$$E(\Lambda_1(s, T_0, \mathrm{Y})) \geqq \sum_{u \in U} E(\lambda'(u)) \geqq \frac{m}{2} \sum_{u \in U} \sum_{j=l_u}^{r_u} p_j = \frac{m}{2} \sum_{j=1}^{n} d_j p_j,$$

where d_j is the number of internal nodes along the path from the root of Y to leaf j.

As is easily shown by induction, $\sum_{j=1}^n d_j p_j \geqq \sum_{j=1}^n p_j \log(1/p_j)$, for any binary tree with n leaves in which the internal path length of leaf j is d_j (see, for example, Knuth [6, p. 445]). So $E(\Lambda_1(s, T_0, \mathrm{Y})) \geqq \frac{1}{2}m \sum_{i=1}^n p_i \log(1/p_i)$. Also, we always have $\Lambda_1(s, T_0, \mathrm{Y}) \geqq m$. So, by Theorem 1,

$$E(\chi(s, T_0)) \geqq E(\Lambda_1(s, T_0, \mathrm{Y})) \geqq \max\left(m, \frac{m}{2} \sum_{i=1}^n p_i \log(1/p_i)\right)$$

$$\geqq \frac{m}{3}\left(1 + \sum_{i=1}^n p_i \log(1/p_i)\right). \qquad \square$$

3. The second lower bound. One problem with the lower bound of the previous section is that it requires picking a good lower bound tree. For an initial tree with n nodes and an access sequence of length m the optimum lower bound tree (i.e., the one that gives the largest bound) can be found in time $O(mn^3)$ by dynamic programming, since if a lower bound tree with root w is optimal for sequence s its left subtree is optimal for sequence $s|_{-\infty}^{w}$ and its right subtree is optimal for sequence $s|_{w}^{\infty}$. However, if we hope to prove that an offline search algorithm is optimal by showing that the time it takes to access a sequence matches some known lower bound then the dynamic programming technique is not very helpful—it is very difficult to get a grip on how the lower bound that comes out relates to the access sequence. Also, there is good reason to believe that even the optimal lower bound tree does not in general give a good bound. For a long sequence may have an access pattern that varies widely from one section of the sequence to another, so that no single lower bound tree works very well.

In this section an alternative way of computing a lower bound for $\chi(s, T_0)$ is described. It does not depend upon a lower bound tree (or even upon T_0) and seems to be better able to handle shifting access patterns.

Let s be a sequence of length m. We describe below a procedure for computing, for each $i \in [1, m]$, a quantity $\kappa(i)$, called the *score* of access i. The new lower bound is defined as

$$\Lambda_2(s) = m + \sum_{i=1}^{m} \kappa(i).$$

When $\kappa(i) > 0$ the procedure for computing $\kappa(i)$ also determines the *inside accesses of i*, $b_1, b_2, \cdots, b_{\kappa(i)}$, the *inside nodes of i*, $v_j = s_{b_j}$, for $j \in [1, \kappa(i)]$, the *crossing accesses of i*, $c_1, c_2, \cdots, c_{\kappa(i)+1}$, and the *crossing nodes of i*, $w_j = s_{c_j}$, for $j \in [1, \kappa(i)+1]$.

A formal procedure for computing the score of i, together with its inside and crossing accesses, is shown in Fig. 1. Here we give an informal description of the procedure. We find the inside and crossing accesses of i by working backward in time from access i. The first crossing access, c_1, is simply the access prior to access i, namely $i-1$. The corresponding crossing node is $w_1 = s_{c_1} = s_{i-1}$. The second crossing access, c_2, is found by going backward in time from c_1 until we reach an access to a node on the side of s_i opposite from w_1. That is, if $w_1 > s_i$ then c_2 is the latest access prior to c_1 to a node less than or equal to s_i, and if $w_1 < s_i$ it is the latest access prior to c_1 to a node greater than or equal to s_i. The corresponding crossing node is $w_2 = s_{c_2}$. Assume without loss of generality that $w_1 < s_i$ so that $w_2 \geqq s_i$. Once the second crossing access has been found we can determine the first inside node, v_1. It is the greatest node less than s_i accessed after c_2 but before (or at) access c_1. The first inside access, b_1, is the access to v_1 within this time interval (if v_1 has been accessed more than once within the interval b_1 is the latest such access). So we have at this point $c_2 < b_1 \leqq c_1 < i$ and $w_1 \leqq v_1 < s_i \leqq w_2$. If $w_2 = s_i$, we are done; otherwise we proceed with the computation of c_3. The third crossing access, c_3, is the latest access prior to c_2 to a node between

```
procedure compute-kappa($s$, $i$)
    if $i = 1$ then begin $\kappa(i) \leftarrow 0$; quit end;
    $c_1 \leftarrow i - 1$; $w_1 \leftarrow s_{i-1}$;
    if $w_1 < s_i$ then $v_0 \leftarrow +\infty$ else $v_0 \leftarrow -\infty$;
    $l \leftarrow 1$;
    loop
        case
        $w_l = s_i$: begin $\kappa(i) \leftarrow l - 1$; quit end;
        $w_l < s_i$:
            begin
            $Q \leftarrow \{j \in [1, c_l) : s_j \in [s_i, v_{l-1})\}$;
            if $Q = \varnothing$ then begin $\kappa(i) \leftarrow l - 1$; quit end;
            $c_{l+1} \leftarrow \max Q$;
            $w_{l+1} \leftarrow s_{c_{l+1}}$;
            $v_l \leftarrow \max \{s_j : j \in (c_{l+1}, c_l] \text{ and } s_j < s_i\}$;
            $b_l \leftarrow \max \{j \in (c_{l+1}, c_l] : s_j = v_l\}$;
            end
        $w_l > s_i$:
            begin
            $Q \leftarrow \{j \in [1, c_l) : s_j \in (v_{l-1}, s_i]\}$;
            if $Q = \varnothing$ then begin $\kappa(i) \leftarrow l - 1$; quit end;
            $c_{l+1} \leftarrow \max Q$;
            $w_{l+1} \leftarrow s_{c_{l+1}}$;
            $v_l \leftarrow \min \{s_j : j \in (c_{l+1}, c_l] \text{ and } s_j > s_i\}$;
            $b_l \leftarrow \max \{j \in (c_{l+1}, c_l] : s_j = v_l\}$;
            end
        endcase;
    endloop
```

FIG. 1. *Procedure for computing the score of an access.*

v_1 (exclusive) and s_i (inclusive), and the corresponding crossing node is $w_3 = s_{c_3}$. Then the second inside node, v_2, is computed as the smallest node greater than s_i that is accessed after c_3 but before (or at) c_2. The second inside access, b_2, is the latest access to v_2 within this time interval. At this point we have $c_3 < b_2 \leqq c_2 < b_1 \leqq c_1 < i$, and $w_1 \leqq v_1 < w_3 \leqq s_i < v_2 \leqq w_2$. If $w_3 = s_i$ we are done; otherwise c_4 is computed as the latest access prior to c_3 to a node between s_i (inclusive) and v_2 (exclusive), and w_4 is set to s_{c_4}. Then v_3 is computed as the greatest node smaller than s_i that is accessed after c_4 but before (or at) c_3. We continue in this way, with the crossing and inside nodes at alternate sides of s_i, getting closer to s_i as we go back in time. Eventually we reach the previous access of s_i (if there is one) or the beginning of the sequence (if there is not) at which point we stop. The score, $\kappa(i)$, is the number of inside accesses found.

It will help to go through an example. We distinguish between accesses and nodes by using boldface letters (ordered alphabetically), rather than integers, to denote nodes. Suppose the sequence s is **a**, **i**, **h**, **j**, **g**, **f**, **c**, **l**, **k**, **e**, **n**, **d**, **b**, **p**, **m**, **o**, **i**. This sequence is illustrated in Fig. 2. The nodes are in sorted order and the order of the access sequence is given by directed edges above the nodes. Suppose we wish to compute $\kappa(17)$, the score for the second access of node **i**. We start at access 17 and follow the access sequence backward in time. The previous access (16) is c_1, the first crossing access, so $w_1 = \mathbf{o}$. To find the second crossing access we go backward until we find an access to a node on the side of **i** opposite from **o**. The two accesses immediately prior to access 16 are to nodes **m** and **p**, which are on the same side of **i** as **o**. Access 13 is to node **b**, so $c_2 = 13$ and $w_2 = \mathbf{b}$. The first inside node is the closest node to **i** accessed after c_2 that is on the right side of **i**. This node is **m**, reached at access 15, so $b_1 = 15$ and $v_1 = \mathbf{m}$. To get the third crossing access we go backward from access c_2 until we reach an access to a node greater than or equal to **i** and less than $v_1 = \mathbf{m}$. The access before 13 is to node **d**, on the wrong side of **i**, the access before that is to **n**, on the wrong side of **m**, and the access before that is to **e**, on the wrong side of **i**. It is not until access 9, to **k**, that we find c_3. The second inside node is the closest node to **i** on its left side that is accessed after c_3 and before or at access c_2. This is node **e**, so $b_2 = 10$ and $v_2 = \mathbf{e}$. The fourth crossing access is obtained by going backward from c_3 until an access to a node between $v_2 = \mathbf{e}$ (exclusive) and **i** (inclusive) is reached. This is access 6, to node **f**. The third inside node is the closest node to **i** on its right side that is accessed after c_4 and before or at c_3. This is node **k**, the same as the third crossing node, so $b_3 = c_3 = 9$. The fifth crossing access is the first access before c_4 to a node

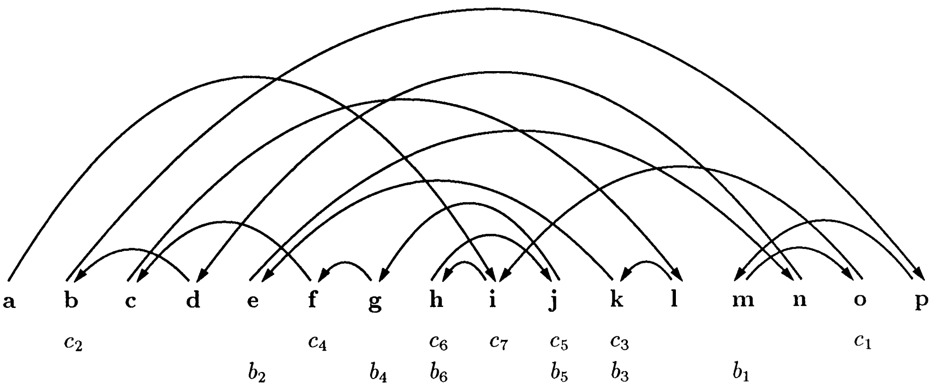

FIG. 2. *Computing the score for the second access of* **i** *in the sequence* **a**, **i**, **h**, **j**, **g**, **f**, **c**, **l**, **k**, **e**, **n**, **d**, **b**, **p**, **m**, **o**, **i**.

greater than or equal to **i** and less than $v_3 = \mathbf{k}$. This is access 4, to node $w_5 = \mathbf{j}$. The fourth inside node is the closest node to **i** on its left side that is accessed after c_5 and before or at c_4, and this is **g**, reached at access 5. Continuing in this way, we find $c_6 = 3$, $w_6 = \mathbf{h}$, and $b_5 = c_5 = 4$, $v_5 = w_5 = \mathbf{j}$. The seventh crossing access is 2, the previous access of **i**, and b_6 is determined to be equal to c_6. At this point the procedure terminates, with $\kappa(17) = 6$.

The crossing and inside accesses of i are all earlier than i and are equal to or later than the previous access of s_i (if there was one). Thus each of the crossing and inside nodes of i is above s_i at some time between the previous access of s_i and access i. Intuitively, after access $i-1$ we expect that the inside nodes of i will be on the path from the root to node s_i. Therefore for access i we expect that a standard search algorithm will have to rotate s_i over at least $\kappa(i)$ nodes, and then once s_i is at the root there will be a cost of 1 for accessing it. Of course, in general many other nodes will have been above s_i at some time since the previous access of s_i, but they can usually be kicked out of the way during the access of some inside node of i. Actually, you can always arrange to have as few inside nodes above s_i after access $i-1$ as you want, but for each inside node that is not above s_i either there will be some other node above s_i in its place or some extra cost will be imposed upon some access prior to the ith access.

The next two simple lemmas formally state properties that by now are probably quite clear.

LEMMA 5. *Let s be a sequence of integers of length m. Let $i \in [1, m]$, let $k = \kappa(i)$, and suppose $k \geqq 1$. Then the inside accesses of $i, b_1, \cdots, b_k$, the inside nodes of $i, v_1, \cdots, v_k$, the crossing accesses of $i, c_1, \cdots, c_{k+1}$, and the crossing nodes of $i, w_1, \cdots, w_{k+1}$, are all well defined. Also, the following relationships hold*:

$$(1) \qquad c_{k+1} < b_k \leqq c_k < b_{k-1} \leqq c_{k-1} < \cdots < b_1 \leqq c_1 < i.$$

If $w_1 < s_i$ and k is odd then

$$(2a) \quad w_1 \leqq v_1 < w_3 \leqq v_3 < \cdots < w_k \leqq v_k < s_i \leqq w_{k+1} < v_{k-1} \leqq w_{k-1} < \cdots < v_2 \leqq w_2,$$

if $w_1 < s_i$ and k is even then

$$(2b) \quad w_1 \leqq v_1 < w_3 \leqq v_3 < \cdots < w_{k-1} \leqq v_{k-1} < w_{k+1} \leqq s_i < v_k \leqq w_k < \cdots < v_2 \leqq w_2,$$

if $w_1 > s_i$ and k is odd then

$$(2c) \quad w_2 \leqq v_2 < w_4 \leqq v_4 < \cdots < w_{k-1} \leqq v_{k-1} < w_{k+1} \leqq s_i < v_k \leqq w_k < \cdots < v_1 \leqq w_1,$$

and if $w_1 > s_i$ and k is even then

$$(2d) \quad w_2 \leqq v_2 < w_4 \leqq v_4 < \cdots < w_k \leqq v_k < s_i \leqq w_{k+1} < v_{k-1} \leqq w_{k-1} < \cdots < v_1 \leqq w_1.$$

Proof. Equations (2a)-(2d) are essentially the same, so assume without loss of generality that $w_1 < s_i$ and k is odd. Everything follows from an easy induction on l. Assume without loss of generality that $w_l < s_i$. If $l \leqq \kappa(i)$ then $\{j \in [1, c_l) : s_j \in [s_i, v_{l-1})\}$ is nonempty so c_{l+1} is well defined and is less than c_l. Also, $s_i \leqq w_{l+1} < v_{l-1}$, and the first branch of the **case** statement ensures that $w_{l+1} = s_i$ only if $l = \kappa(i)$. We have $w_l \in \{s_j : j \in (c_{l+1}, c_l]$ and $s_j < s_i\}$ so v_l is well defined and $v_l \geqq w_l$. The definition of v_l ensures that b_l is well defined and by definition of b_l we have $c_{l+1} < b_l \leqq c_l$. □

LEMMA 6. *Let s be a sequence of integers of length m and let $i \in [1, m]$ with $k = \kappa(i)$ positive. Let $c_1, \cdots, c_{k+1}$ be the crossing accesses of i, let $w_1, \cdots, w_{k+1}$ be the crossing nodes of i, and let $v_1, \cdots, v_k$ be the inside nodes of i. Let $l \in [1, k]$. Let interval I_l be* *either* $(v_l, w_{l+1}]$ *or* $[w_{l+1}, v_l)$, *according to whether* $v_l < s_i$ *or* $v_l > s_i$. *Then* $i = \min\{j > c_{l+1} : s_j \in I_l\}$.

Proof. Assume without loss of generality that $v_l > s_i$. Let $Y = \{j > c_{l+1} : s_j \in (v_l, w_{l+1}]\}$. Let $h = \min Y$. Since $i \in Y$ we have $h \leqq i$. We show that the assumption that $h < i$ leads to a contradiction. Suppose $h < i$. Then $h \leqq i - 1 = c_1$. So, by (1), there is an $m \in [1, l]$ such that $h \in (c_{m+1}, c_m]$. We must have $w_m < s_i$ or $w_m > s_i$.

Suppose $w_m < s_i$. If $s_h < s_i$ then by definition of v_m we have $s_h \leqq v_m$. But $v_m \leqq v_l$ so $s_h \leqq v_l$, contradicting the definition of h. Thus the assumption that $w_m < s_i$ implies that $s_h \geqq s_i$. Since $s_{c_m} = w_m < s_i$ we must have $h < c_m$. Also, $w_{l+1} < v_{l-1} \leqq v_{m-1}$, so $s_h \in [s_i, v_{m-1})$. So by definition of c_{m+1} we have $c_{m+1} \geqq h$, a contradiction.

Therefore we must have $w_m > s_i$. Since $w_l < s_i$ we must have $m \neq l$, i.e., $m \leqq l - 1$. If $s_h > s_i$ then by definition of v_m we have $v_m \leqq s_h$. But $w_{l+1} \leqq w_{m+2} < v_m$, so $w_{l+1} < s_h$, contradicting the definition of h. So we must have $s_h \leqq s_i$ and since $v_{m-1} < v_l$ we have $s_h \in (v_{m-1}, s_i]$. Since $s_{c_m} = w_m > s_i$ we must have $h < c_m$. So by definition of c_{m+1} we have $c_{m+1} \geqq h$, a contradiction. □

THEOREM 7. *For any search tree* T_0 *and sequence* s *of nodes in* T_0 *we have* $\chi(s, T_0) \geqq \Lambda_2(s)$.

Proof. Let m be the length of s, and for each $i \in [1, m]$ define $\kappa(i)$ using the procedure `compute-kappa`(s, i). Once a node is at the root the cost of accessing it is 1, so it suffices to show that $\sum_{i=1}^{m} \kappa(i)$ is a lower bound on the number of rotations required by any standard search algorithm that accesses s.

We use an accounting argument. Let R be the sequence of rotations generated by some standard search algorithm acting on sequence s and initial tree T_0. Let r be the length of R. For $t \in [1, r]$, let T_t be the search tree obtained by applying rotation R_t to tree T_{t-1}. Let $\tau_0 = 0$ and, for $i \in [1, m]$, let τ_i be the smallest integer greater than or equal to τ_{i-1} such that s_i is at the root of T_{τ_i}.

For each time $t \in [1, r]$, we put a dollar on at most one node, determined as follows. Let $R_t = (u, v)$ (node u is rotated over node v). Let $I_{u,v} = (u, v]$ if $u < v$ and $I_{u,v} = [v, u)$ if $u > v$. Let j be the smallest integer such that $\tau_j \geqq t$ and $s_j \in I_{u,v}$, if such an integer exists. If j is undefined then no node gets a dollar at time t. Otherwise, a dollar is put on node s_j.

After rotation R_{τ_i} is carried out all the dollars on node s_i are thrown away. Since at most one dollar is put in the tree per rotation the number of dollars thrown away is a lower bound on the number of rotations used. Thus to get the desired bound it suffices to show that for each $i \in [1, m]$, after rotation R_{τ_i} has been carried out the number of dollars taken off node s_i is at least $\kappa(i)$.

Let $i \in [1, m]$ with $k = \kappa(i)$ positive. Let $b_1, \cdots, b_k$ be the inside accesses of i, let $v_1, \cdots, v_k$ be the inside nodes of i, let $c_1, \cdots, c_{k+1}$ be the crossing accesses of i, and let $w_1, \cdots, w_{k+1}$ be the crossing nodes of i. By setting $l = k$ in Lemma 6 we see that $i = \min\{j > c_{k+1} : s_j = s_i\}$, so no dollars are taken from s_i in the interval $(\tau_{c_{k+1}}, \tau_{c_1}]$. Therefore it suffices to show that for each $l \in [1, k]$ a dollar is placed on node s_i for some $t \in (\tau_{c_{l+1}}, \tau_{b_l}]$. (By Lemma 5 these k intervals are all disjoint.)

Let $l \in [1, k]$. Assume without loss of generality that $w_{l+1} \leqq s_i$. For all $t \in (\tau_{c_{l+1}}, \tau_{b_l}]$ let $a(t)$ be the lowest common ancestor of w_{l+1} and v_l in tree T_t. We have $a(\tau_{c_{l+1}}) = w_{l+1} \leqq s_i$ and $a(\tau_{b_l}) = v_l > s_i$. So there must be a $t \in (\tau_{c_{l+1}}, \tau_{b_l}]$ such that $a(t-1) \leqq s_i$ and $a(t) > s_i$. Let $R_t = (y, z)$. It is straightforward to verify that we must have $a(t-1) = z$ and $a(t) = y$, for otherwise the lowest common ancestor of w_{l+1} and v_l would not change with tth rotation. Also $w_{l+1} \leqq z$ and $y \leqq v_l$. Thus $s_i \in [z, y) \subseteq [w_{l+1}, v_l)$. Let h be the smallest integer such that $\tau_h \geqq t$. Since $t > \tau_{c_{l+1}}$ we have $h > c_{l+1}$. The node that gets the dollar for rotation t is s_j such that $j = \min\{g \geqq h : s_g \in [z, y)\}$ and by Lemma 6 that node is s_i. □

It is perhaps not immediately obvious that $\Lambda_2(s)$ ever gives a bound that is superlinear in the length of s. Here we show that $\Lambda_2(s)$ can be used to get the $\Theta(n \log n)$ bound for accessing the bit reversal permutation.

THEOREM 8. *Let B^k be the bit reversal permutation on $n=2^k$ nodes. Then* $\Lambda_2(B^k) \geqq \frac{1}{2}n \log n + 1$.

Proof. It is convenient to index the sequence B^k from zero rather than 1, so that the ith access is to node $\mathrm{br}_k(i)$. By definition $\Lambda_2(s) = n + \sum_{i=0}^{n-1} \kappa(i)$, where $\kappa(i)$ is the number of inside accesses of i. We use induction on k to show that $\sum_{i=0}^{n-1} \kappa(i) \geqq \frac{1}{2}n \log n - n + 1$. When $k \leqq 1$ the claimed bound is zero so there is nothing to prove.

Suppose $k \geqq 2$. Clearly the sequence $\frac{1}{2}B_0^k, \frac{1}{2}B_1^k, \cdots, \frac{1}{2}B_{n/2-1}^k$ is precisely B^{k-1}. Let $i, j \in [0, n/2-1]$. Then j is an inside access of i for sequence B^k if and only if j is an inside access of i for sequence B^{k-1}. So by the induction hypothesis,

$$\sum_{i=0}^{n/2-1} \kappa(i) \geqq \frac{1}{2}\left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) - \frac{n}{2} + 1.$$

We also have $B^{k-1} = \frac{1}{2}(B_{n/2}^k - 1), \frac{1}{2}(B_{n/2+1}^k - 1), \cdots, \frac{1}{2}(B_{n-1}^k - 1)$. Let $i, j \in [n/2, n-1]$. Then j is a crossing access of i for sequence B^k if and only if $j - n/2$ is a crossing access of $i - n/2$ for sequence B^{k-1}. So each $i \in [n/2, n-1]$ has at least as many inside accesses in sequence B^k as $i - n/2$ has in sequence B^{k-1}. We will show that each $i \in [n/2, n-2]$ actually has at least one more inside access in B^k than $i - n/2$ has in B^{k-1}. When $i = n/2$ access $i - n/2$ has no crossing accesses and no inside accesses in B^{k-1}, but access i (to node 1) has two crossing accesses in B^k, namely, $i-1$ and 0, so it has one inside access. If $i \in [n/2+1, n-2]$ then $i - n/2$ has at least one crossing access in B^{k-1}, so it suffices to show that i has one more crossing access in B^k than $i - n/2$ has in B^{k-1}. Access i has a crossing access in $[n/2, n-1]$, namely $i-1$, so let c be the earliest crossing access of i that is greater than or equal to $n/2$. Let $u = B_i^k$. Suppose c is the lth crossing access of i. Assume without loss of generality that $B_c^k < u$. Let v be the $(l-1)$th inside node of i. (If $l = 1$ we may take v to be $+\infty$.) There is at least one even node in $[u, v)$ (namely $u+1$) and of these the one that is accessed latest is a crossing node of i, and its access is in $[0, n/2-1]$. So by the induction hypothesis,

$$\sum_{i=n/2}^{n-1} \kappa(i) \geqq \frac{1}{2}\left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) - \frac{n}{2} + 1 + \left(\frac{n}{2} - 1\right).$$

Therefore

$$\sum_{i=0}^{n-1} \kappa(i) \geqq \frac{1}{2} n \log n - n + 1. \qquad \square$$

Conclusions. Two methods for obtaining lower bounds on $\chi(s, T_0)$ have been described. The first is useful for getting bounds on the expected value of $\chi(s, T_0)$ when sequence s is chosen at random, whereas the second seems more likely to give tight bounds for specific sequences. The obvious unresolved problem is to obtain tight bounds for $\chi(s, T_0)$. That is, to find an efficient (polynomial-time) procedure for computing an upper bound on $\chi(s, T_0)$ that can be shown to match some known lower bound. Preferably the procedure would give an explicit method for optimally accessing sequence s offline. The function $\Lambda_2(s)$ may give a tight bound for the cost of accessing sequence s (up to a constant factor). However, I have been unable to find a binary search tree algorithm whose performance provably matches the $\Lambda_2(s)$ lower bound.

Of course, the ultimate goal is to find some simple online algorithm that always runs in time $O(\chi(s, T_0) + n)$. The splay algorithm may yet prove to be such an algorithm.

REFERENCES

[1] G. M. ADEL'SON-VEL'SKII AND E. M. LANDIS, *An algorithm for the organization of information*, Sov. Math. Dokl., 3 (1962), pp. 1259-1262.

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

[3] C. A. CRANE, *Linear lists and priority queues as balanced binary trees*, Tech. Report STAN-CS-72-259, Computer Science Department, Stanford University, Stanford, CA, 1972.

[4] K. CULIK II AND D. WOOD, *A note on some tree similarity measures*, Inform. Process. Lett., 15 (1982), pp. 39-42.

[5] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8-21.

[6] D. E. KNUTH, *The Art of Computer Programming*. Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[7] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, SIAM J. Comput., 2 (1973), pp. 33-43.

[8] H. OLIVIÉ, *A new class of balanced search trees: Half-balanced binary search trees*, Tech. Report 80-02, Interstedelijke Industriele Hogeschool Antwerpen-Mechelen, Antwerp, Belgium, 1980.

[9] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652-686.

[10] D. D. SLEATOR, R. E. TARJAN, AND W. P. THURSTON, *Rotation distance, triangulations, and hyperbolic geometry*, Proc. 18th Annual ACM Symposium on Theory of Computing, (1986), pp. 122-135.

[11] R. E. TARJAN, *Sequential access in splay trees takes linear time*, Combinatorica, 5 (1985), pp. 367-378.