ADAPTIVE STRUCTURING OF BINARY SEARCH TREES USING CONDITIONAL ROTATIONS[+]

R. P. Cheetham, B. J. Oommen[@] and D.T.H. Ng[*]

ABSTRACT

Consider a set $\mathbf{A} = \{A_1, A_2,...,A_N\}$ of records, where each record is identified by a unique key. The records are accessed based on a set of access probabilities $\mathbf{S} = [s_1,s_2,...,s_N]$ and are to be arranged lexicographically using a Binary Search Tree (BST). If **S** is known *a priori*, it is well known [10] that an optimal BST may be constructed using **A** and **S**. We consider the case when **S** is not known *a priori* . A new restructuring heuristic is introduced that requires three extra integer memory locations per record. In this scheme the restructuring is performed **only** if it decreases the Weighted Path Length (WPL) of the overall resultant tree. An optimized version of the latter method which requires only one extra integer field per record has also been presented. Initial simulation results which compare our algorithm with various other static and dynamic schemes seem to indicate that this scheme asymptotically produces trees which are an order of magnitude closer to the optimal one than those produced by many of the other BST schemes reported in the literature.

Keywords : Adaptive Data Structures, Binary Search Trees, Dynamic Data Structures, Move-to-Root Heuristic, Self Organizing Structures, Splay Trees.

[+] Partially supported by the Natural Sciences and Engineering Research Council of Canada. A preliminary version of *some* of these results was presented at the 1988 International Conference on Database Theory, in Bruges, Belgium.

[@] Senior Member, IEEE

[*] All the authors can be contacted at the School of Computer Science, Carleton University, Ottawa ; Canada : K1S 5B6.

I. INTRODUCTION

A Binary Search Tree (BST) may be used to store records whose keys are members of an ordered set $\mathbf{A} = \{A_1, A_2, \ldots, A_N\}$. The records are stored in such a way that a symmetric-order traversal of the tree will yield the records in ascending order. This structure has a wide variety of applications, such as for symbol tables and dictionaries.

If we are given **A** and the set of access probabilities $\mathbf{S}=[s_1,s_2,..,s_N]$, the problem of constructing efficient BSTs has been extensively studied. The optimal algorithm due to Knuth [10], uses dynamic programming and produces the optimal BST using $O(N^2)$ time and space. Alternatively, Walker and Gotlieb [20] have used dynamic programming and divide-and-conquer techniques to yield a nearly-optimal BST using O(N) space and $O(N \cdot \log N)$ time. In this paper, we study the problem in which **S**, the access probability vector, is **not** known *a priori*. We seek a scheme which dynamically rearranges itself and generates a tree which is asymptotically optimal.

This topic is closely related to the subject of self-organizing lists. A self-organizing list is a linear list that rearranges itself such that after a long enough period of time, it tends towards the optimal arrangement with the most probable element at the head of the list and the rest of the list recursively ordered in the same manner. Many memory-less schemes have been developed to reorganize a linear list dynamically [3,5,8-10,12]. Among these are the **move-to-front** rule [10,12] and the **transposition** rule [18]. These rules and their extensions and their analytic properties are discussed extensively in the literature [3,5,8-10,12]. Schemes involving the use of extra memory have also been developed ; a review of these is found in [9]. The first of these uses counters to achieve estimation. Another is a stochastic move-to-rear rule [16] due to Oommen and Hansen, which moves the accessed element to the rear with a probability which decreases each time the element is accessed. A stochastic move-to-front [16] and a deterministic move-to-rear scheme [17] due to Oommen *et. al*. have also been reported.

A BST is not quite so simple to reorganize as a linked list because any reorganizing scheme must simultaneously give due consideration to the lexicographic ordering of the records and the statistical information about their access probabilities. Clearly, these can often be conflicting constraints for a reorganization scheme. Recent result relating the two will be mentioned presently.

The primitive tree restructuring operation used in most BST schemes is the well known **Rotation** [1]. A few memory-less tree reorganizing schemes which use this operation have been presented in the literature among which are the **Move-to-Root** and **simple exchange** rules. These rules are analogous in spirit to the move-to-front and transposition rules respectively for linear lists. The Move-to-Root Heuristic was historically the first self-organizing BST scheme in the literature [2] and is due to Allen and Munro. It is both conceptually simple and elegant. Each time a record is accessed, rotations are performed on it in an upwards direction *until* it becomes

the root of the tree. The idea is that a frequently accessed record will be close to the root of the tree as a result of it being frequently moved to the root, and this will minimize the cost of the search and the retrieval operations. Allen and Munro [2] also developed the **simple exchange** rule, which rotates the accessed element one level towards the root, similar to the transposition rule for lists. Contrary to the case of lists, where the transposition rule is better than the move-to-front rule [18], they show that whereas the Move-to-Root scheme has an expected cost that is within a constant factor of the cost of a static optimum BST, the simple exchange heuristic does not have this property. Indeed, it is provably bad.

At this juncture, it is not out of place to mention that various interesting relationships between self-organizing lists and BSTs have recently been reported [11]. For example, suppose that a list ρ is modified using the Move-to-Front heuristic on accessing the element A to yield the list ρ'. Then in [11], Lai and Wood have shown that if from an empty tree a BST μ is created using ρ, and another μ' is created using ρ', the tree μ' can be obtained from μ by applying the Move-to-Root Heuristic defined on BSTs. The paper [11] shows that the transposition heuristic does not possess this property and also presents come fascinating conjectures which claim the existence of various commutative results that relate lists and BSTs for other families of heuristics.

Sleator and Tarjan [19] introduced a technique, which also moves the accessed record up to the root of the tree using a restructuring operation called **splaying** which is a multi-level generalization of rotation. Their structure, called the **splay tree**, was shown to have an amortized time complexity of O(log N) for a complete set of tree operations which included insertion, deletion, access, split, and join. This heuristic is rather ingenious. It is a restructuring move that brings the node accessed up to the root of the tree, and also keeps the tree in a symmetric order, and thus an in-order traversal would access each item in order from the smallest to the largest. Additionally it has the interesting side effect of tending to keep the tree in a form that is nearly height-balanced apart from also capturing the desired effect of keeping the most frequently accessed elements near the root of the tree. The heuristic is somewhat similar to the Move-to-Root scheme, but whereas the latter has an asymptotic average access time within a constant factor of optimum when the access probabilities are independent and time invariant, the splaying operation yields identical results even when these assumptions are relaxed. More explicit details and the analytic properties of the splay tree with its unique "two-level rotations" can be found in [6,19].

The literature also records various schemes which adaptively restructure the tree with the aid of additional memory locations. The two outstanding schemes in this connection are the **monotonic tree** and Mehlhorn's D-Tree [5,10,14]. The monotonic tree is a dynamic version of a tree structuring method suggested by Knuth [10] as a means to structure a nearly optimal static tree. The static monotonic tree is arranged such that the most probable key is the root of the tree,

and the subtrees are recursively ordered in the same manner. The static version of this scheme behaves quite poorly [13]. Walker and Gotlieb [20] have presented simulation results for static monotonic trees, and these results also indicate that this strategy behaves quite poorly when compared to the other static trees known in the literature.

Bitner suggested a dynamic version of this scheme [5], which could be used in the scenario when the access probabilities are not known *a priori* . Each record has one extra memory location, which "counts" the number of accesses made to it. The reorganization of the tree after an access is then very straightforward. When a record is accessed, its counter is incremented, and then the record is rotated upwards in the tree until it becomes the root of the tree, or it has a parent with a higher frequency count than itself. Over a long enough sequence of accesses this will, by the law of large numbers, converge to the arrangement described by the static monotonic tree. Although this scheme is intuitively appealing, Bitner determined bounds for the cost of a monotonic tree, and showed that it is largely dependent on the entropy, H(**S**) of the probability distribution of the keys. If H(**S**) is small, then the monotonic tree is nearly optimal ; but if H(**S**) is large, it will behave quite poorly. Bitner also stated a result of Bayer's [4], that proved that the expected entropy of a randomly chosen probability distribution[1] is log(N)-ln 2, which is nearly the maximum entropy attainable. He concludes from this that, on the average, the monotonic tree scheme will behave poorly . Our experimental results [6] support this viewpoint.

As opposed to the above, Mehlhorn's D-tree is a BST scheme significantly different from the Monotonic Tree [14]. At every node the D-tree maintains counters which record the weights of the two subtrees at the node, where the weight of a subtree is defined as the number of leaves in that subtree. D-trees permit multiple leaves for the same object, for indeed, each time an object is searched, the number of leaves referring to that object is increased by unity. The searching technique in the scheme ensures that all searches will be properly directed to corresponding objects at the leaf level. In any actual implementation of the D-tree, both search-time and space can be saved by coalescing a significant number of leaves into a single "super-leaf". The D-tree uses the weights of the two children subtrees at each node as the input parameters to a balancing function which provides a numeric measure for how "balanced" the subtree itself is. On executing a search for a record, if the balancing function of any node in the tree exceeds this threshold, single and/or double rotations are executed at strategic nodes along the search path which ensure that the D-tree remains balanced after the rotations are executed. The properties of D-Trees are found in [14]. This scheme is closely related to the BB[α]-trees described by Mehlhorn in [15, pp. 189-199]. The latter uses the weight (defined as in the D-tree) of the subtrees of a tree as a method of quantifying how balanced the tree is. A node in the BB[α]-tree scheme is considered

[1]Bitner's analysis assumes that the set of access probabilities is randomly assigned to the records, and thus that each permutation of **S** is equally likely. We are grateful to an anonymous referee who pointed out that this assumption is unrealistic.

to be balanced if the balancing function, which takes the weight of the left and right subtrees as parameters, returns a value which is bounded by the variable α. If the tree is reckoned to be "unbalanced", just as in the case of the AVL-tree [10], the BB[α]-tree reorganizes the nodes using single or double rotations.

In this paper we introduce a new heuristic to reorganize a BST so as to asymptotically arrive at an optimal form. It requires three extra memory locations per record. Whereas the first counts the number of accesses to that record, the second counts the number of accesses to the subtree rooted at that record, and the third evaluates the Weighted Path Length (WPL) of the subtree rooted at that record. The paper specifies an optimal updating strategy for these memory locations. More importantly, however, the paper also specifies how an accessed element can be rotated towards the root of the tree so as to minimize the overall cost of the tree. Finally, unlike most of the algorithms that are currently in the literature, this move is not done on every data access operation. It is performed if and only if the overall WPL of the resulting BST decreases.

From a very naive perspective one can formulate a straightforward algorithm to achieve the above goal. Whenever a record is accessed, the three counters mentioned above can be updated for every node in the tree, and then, in principle, the restructuring can be done by merely evaluating what the effect of the potential rotation operation would be to the overall WPL of the tree. This naive scheme is computationally very expensive, for it involves multiple traversals of the tree for every rotation executed. This is because there seems to no way to "anticipate" whether a certain rotation operation will decrease the overall WPL of the resulting BST. But it is exactly here that we believe that this paper makes a fundamental contribution which distinguishes it from all the other reported schemes. By defining the counters appropriately, we have been able to discover a criterion function that can be locally computed -- i.e., computed at a particular node using only the values of the counters of itself, its parents and its direct offspring. We then present a decision process based on the computation of this **local** criterion function and reports whether the rotation of this accessed node is beneficial in reducing the overall WPL of the resulting BST. Thus, the effect of a rotation can be anticipated, and in this way the restructuring can be rendered both asymptotically optimal in terms of the expected cost and also computationally optimal in terms of the local pointer manipulation operations executed at each time instant. Experimental results show that our scheme asymptotically produces an optimal BST.

Apart from the original three-counter conditional rotation scheme described above we also present a space-optimized version which only requires a single additional counter per node. Using this memory location a scheme identical to the one described above has been designed. The details of the computations required per access in each algorithm is described in the subsequent sections.

Viewed from the perspective of this minimizing the "path length", in one sense, our scheme is reminiscent of Gonnet's scheme for statically constructing BSTs using his path length balanced trees [7]. However in our case, we shall attempt to minimize the *weighted* path length by incorporating the statistical information about the accesses to the various nodes and subtrees rooted at the corresponding nodes. The essential differences between Gonnet's scheme [7] and the technique which we introduce will be apparent in the subsequent sections.

In spite of all their advantages, all of the schemes mentioned above have drawbacks, some of which are more serious than others. The two memoryless schemes have one major disadvantage, which is that both the Move-to-Root and splaying rules always move the record accessed up to the root of the tree. This means that if a nearly-optimal arrangement is reached, a single access of a seldomly-used record will disarrange the tree along the entire access path as the element is moved upwards to the root. Thus the number of operations done on every access is exactly equal to the depth of the node in the tree, and these operations are not merely numeric computations (such as those involved in maintaining counters and "balancing functions") but rotations. Thus the Move-to-Root and splaying rules can be very expensive. We seek a heuristic that solves these problems.

As opposed to these schemes, the monotonic tree rule does not move the accessed element to the root every time. But as we have seen, the monotonic tree rule does not perform well. Our aim is to adopt the philosophy taken by this rule, but in doing so, we would like to achieve the restructuring **conditionally**, depending on the counters of the nodes of the actual physical tree. Hopefully, this strategy will overcome the problems with the memoryless schemes, because an adjustment will not be performed if it brings the structure into a worse state. Also, the weakness of the monotonic tree, which lies in the fact that it considers only the frequency counts for the records will be overcome. Thus, we shall avoid the undesirable property that in a rotation a subtree with a relatively large probability weight may be moved downwards, thus increasing the cost of the tree.

Numerous simulation results comparing our scheme to a number of BST algorithms are found in [6]. Although the details of the experiments have not been presented in this paper it is not inappropriate to add that from the results that we have obtained, we believe that our scheme produces trees which are typically an order of magnitude closer to the optimal one than those normally produced by other adaptive BST schemes.

Throughout this paper we assume that the access probability distribution **S** is time-invariant and unknown, and that the components of **S** sum to unity.

II. THE CONDITIONAL ROTATION HEURISTIC

Before we present our new technique, in the interest of rendering our arguments to be more easily understandable we shall briefly review the Rotation Operation introduced by Adel'son-Velski'i and Landis [1] and state its properties. To explain this operation we use the notation that for a node i in the tree, P(i) is its **unique** parent node. By definition, P(root) is NIL.

Suppose that there exists a node i in a BST, and it has a parent node j, a left child i_L, and a right child i_R. Consider the case that i is a left child (see Figure Ia). A rotation is performed on node i as follows. j now becomes the right child, i_R becomes the left child of node j, and all other nodes remain in their same relative positions (see Figure Ib). The case that node i is a right child is done symmetrically. This operation has the effect of raising a specified node in the tree structure while preserving the lexicographic order of the elements (refer again to Figure Ib). The properties of this operation are stated as Fact I.

********** **Insert Figures Ia & Ib** **********

Fact I.

The following are the properties of a rotation performed on node i.

(i) The subtrees rooted at i_L and i_R remain unchanged.

(ii) After a rotation is performed, i and P(i) interchange roles i.e., i becomes the parent of P(i).

(iii) Except for P(i), nodes which were ancestors of i before a rotation remain as ancestors of i after it.

(iv) Nodes which were not ancestors of i before a rotation do not become ancestors of i after it.

II.1 Principles Motivating the Heuristic

The new heuristic which we introduce requires that each of the records in the BST contains three integer storage locations. The first location contains the number of accesses to that node, the second contains the total number of accesses to the subtree rooted at that node, and the third contains the WPL of the subtree rooted at that node. Every time an access is performed, these fields are updated for the accessed node, and also along the path traversed to achieve the access. The accessed node is rotated upwards (i.e. towards the root) **once** if and only if the WPL of the entire tree decreases as a result of the operation.

We first introduce some elementary definitions. Let i be any node in the given tree, whose left and right children are i_L and i_R respectively. T_i is the subtree rooted at node i. The parent of node i is P(i), and its unique brother is B(i), where B(i) would be NIL if it is non-existent. We define $\alpha_i(n)$ as the total number of accesses of node i up to and including the time instant n. Similarly, we define $\tau_i(n)$ as the total number of accesses to the subtree rooted at node i. Clearly $\tau_i(n)$ satisfies :

$$\tau_i(n) = \sum_{j \ T_i} \alpha_j(n). \qquad (1)$$

Let $\lambda_i(n)$ be the path length of i from the root. By definition, this quantity is at least unity. Then, $\kappa_i(n)$ is defined as the WPL of the tree T_i rooted at node i at time instant n, where :

$$\kappa_i(n) = \sum_{j \ T_i} \alpha_j(n) \cdot \lambda_j(n). \qquad (2)$$

By simple induction $\tau_i(n)$ and $\kappa_i(n)$ can be shown to be related by (3) below :

$$\kappa_i(n) = \sum_{j \ T_i} \tau_j(n). \qquad (3)$$

To simplify the notation, where no ambiguity results, we shall omit all references to time 'n'.

Since the τ and κ values need to be updated each time a record is accessed, we need a method to update them that doesn't require a complete traversal of the tree at every time instant. Obviously such a traversal is not required in order to update α_i, but computational schemes are necessary for the evaluation of τ and κ. The following lemma yields the recursively computable properties of τ_i and κ_i (as opposed to their intrinsic definitions given by (2) and (3) respectively), and these properties shall be used to update them without traversing the entire tree.

Lemma I.

For T_i, the subtree rooted at node i, the following are true :

$$\text{(a)} \quad \tau_i = \alpha_i + \tau_{iL} + \tau_{iR} \qquad (4)$$

$$\text{(b)} \quad \kappa_i = \alpha_i + \tau_{iL} + \tau_{iR} + \kappa_{iL} + \kappa_{iR} \qquad (5)$$

Proof :

(a) This result is indeed obvious.

(b) Expanding (2) in terms of the subtrees at each level, we write[2] ,

$$\kappa_i = \alpha_i + 2\alpha_{iL} + 2\alpha_{iR} + 3\alpha_{(iL)L} + 3\alpha_{(iL)R} + 3\alpha_{(iR)L} + 3\alpha_{(iR)R} + \ldots \qquad (6)$$

whence,

$$\kappa_i = (\alpha_i + \alpha_{iL} + \alpha_{iR} + \alpha_{(iL)L} + \alpha_{(iL)R} + \alpha_{(iR)L} + \alpha_{(iR)R} + \ldots)$$
$$+ (\alpha_{iL} + 2\alpha_{(iL)L} + 2\alpha_{(iL)R} + \ldots) + (\alpha_{iR} + 2\alpha_{(iR)L} + 2\alpha_{(iR)R} + ...) \qquad (7)$$

Using (1) and (2), (7) simplifies to

$$\kappa_i = \tau_i + \kappa_{iL} + \kappa_{iR}.$$

The result follows if we incorporate (4) into the above.

---

[2] This scenario takes care of the cases when the left and right subtrees are not empty. In the case of empty subtrees, the expansion is trivially valid since the corresponding α values would be identically equal to zero.

This lemma implies that to calculate τ and κ for any node, it is necessary **only** to look at the values stored at the node and at the quantities stored for the **children** of the node. Note that the recursion does not involve the quantities in the entire tree nor the entire subtree rooted at the node. Now that we have derived the recursive properties of τ_i and κ_i, we shall obtain a simple method to calculate the α, τ, and κ fields of the tree after each access. We shall do this for both the case in which a rotation is performed and for the case in which no rotation is performed. Whereas the former is rather straightforward, the latter is a little more involved.

Theorem I.

Let j be any arbitrary node in the **entire** tree, T. On accessing i T, the following scheme for updating scheme α, τ, and κ is consistent whether a rotation operation is performed at i or not.

(a) Updating of α :

$\alpha_j := \alpha_j + 1$, j = i

$\alpha_j := \alpha_j$, j i.

(b) Updating of τ :

(i) τ values in the subtrees of node i are unchanged.

(ii) τ values in the subtrees not on the access path from the root to node i are unchanged.

(iii) τ values in the nodes on the path from the root to P(P(i)) are updated according to :

$\tau_j := \tau_j + 1$.

(iv) τ_i and $\tau_{P(i)}$ are updated according to (iii) above, unless a rotation is performed. If a rotation is performed they are updated by applying

$\tau_j := \alpha_j + \tau_{jL} + \tau_{jR}$, where j {i, P(i)},

and $\tau_{P(i)}$ is computed before τ_i.

(c) Updating of κ :

(i) κ values in the subtrees of node i are unchanged.

(ii) κ values in the subtrees not on the access path from the root to node i are unchanged.

(iii) κ values in the nodes on the path from the root to node i are updated by applying

$\kappa_j := \alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{iL} + \kappa_{iR}$

from node i **upwards** to the root.

Proof of (a) :

The updating rule for α_j is quite clear from the definition of α. Indeed, α_i is the number of accesses to node i, and thus, if node i is accessed, **only** α_i needs to be increased.

Proof of (b) :

We will consider the cases (i) through (iv) separately.

Case (i) : This follows from the definition of τ. Note that τ will not change for nodes in the subtrees of i even if a rotation is performed. This is by virtue of the properties of a rotation given

in Fact I. Thus no α value will increase in any node below i, and hence the corresponding values of τ will remain unchanged.

Case (ii) : The result is obvious when no rotation is performed. When a rotation is performed, no subtree which is **not** on the access path can contain i, even if a rotation is performed at i. For every node k, since each τ_k equals the sum of all α_j, j T_k, and since the only α-value that is changed is α_i, due to Lemma I, the value of τ_k is unchanged because for all k not on the access path to i, i T_k. Consequently, no updating is needed on such subtrees.

Case (iii) : The result is again obvious when no rotation is performed at i. Consider the case when a rotation is performed. Every subtree rooted at a node on the access path to node i contains node i; hence, by the definition of τ, an access to node i must increase every τ_j on the access path by unity. The result follows since, by the properties given in Fact I, ancestors of i up to P(P(i)) continue to be ancestors of i in spite of the rotation.

Case (iv) : If no rotation operation is performed, the proof of case (iii) above leads to the result. However, if a rotation is performed, then i and P(i) interchange places, i.e., i becomes the parent of P(i). The application of Lemma I now yields the result except that for the computation to be consistent $\tau_{P(i)}$ must be calculated **before** τ_i.

Proof of (c) :

We consider cases (i) and (ii) together, and case (iii) separately.

Cases (i) and (ii) : These are proved using essentially the same arguments used in cases (i) and (ii) in part (b) above. Since no α–values change in these same circumstances, the κ values also remain unchanged. This follows from the definition of κ.

Case (iii) : To compute the WPL for a node on the access path, we note that this quantity will change for all subtrees containing node i, essentially because α_i changes. Therefore, using Lemma I, we recalculate this quantity for all ancestors of i by applying

$$\kappa_j = \alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{iL} + \kappa_{jR}$$

due to (5). Note that since the value of the WPL at any node depends on the τ and κ values of the children of the node, the κ values must be updated in a bottom-up fashion.

II.1.1 Remarks

(i) Note that after a rotation has been performed, our notation can be misleading if the reader does not realize that the notation refers to the node identities. Observe that we refer to the accessed node and its parent, j, before the rotation is performed as i and P(i)=j respectively. After the rotation has been performed, however, we refer to the **same** nodes as i and j, even though the original relationship between the nodes has been destroyed -- it is completely reversed. To render the notation consistent, P(i) will refer to j. However, we introduce the notation that post-rotational quantities shall have the superscript '. Thus, P(j)' = i in this particular case.

(ii) In the case of the τ values, updating may be performed quite simply on the way down the tree to node i. However, the κ values cannot be so simply updated on the downward pass, and so they must be updated from the bottom up using the recursive relationship derived in Lemma I. For this reason the two cases present in part (b) of Theorem I merge into just one case in part (c).

II.2 Criteria for Performing Rotations

Up to this point we have described the memory locations used for determining whether or not to rotate, and how these locations may be updated for a node, both in the case when a rotation is performed and in the case when it is not. What we have not addressed yet is the question of a criterion to decide whether a rotation should be performed or not.

The basic condition for rotation of a node is that the WPL of the **entire tree** must decrease as a result of a rotation. A brute-force method for determining whether or not to rotate suggests itself immediately as follows. When an access is performed, the record is found, and updating at the record is done. We then retrace our steps back along the access path to the root of the tree, updating the values of τ and κ as we go, simultaneously computing the hypothetical values of τ and κ that would be obtained if a rotation is performed. On reaching the root of the tree, we decide whether to perform the rotation or not by comparing the hypothetical κ-value to the actual κ-value. If the hypothetical value is smaller, then the record is again found, the rotation is performed, and the τ and κ values are again updated upwards along the access path as described in Theorem I.

This method achieves exactly the results that we desire, but it is very expensive. In the case that a rotation is actually performed, a total of four[3] passes must be made between the accessed node and the root of the tree. What we now consider is a method to anticipate **at the level of the rotation itself** whether or not the κ-value of the entire tree will decrease if a rotation is performed, and hence determine if the operation should be performed or not. By applying such a method, we can reduce the number of passes to two.

Before we proceed we shall clarify our notation. Any primed quantity (e.g. α',τ',κ') is a **post-rotational** quantity. That is, it is the value of the specified quantity after the rotation has been performed. If close attention is not paid, the notation can be quite misleading. This may be overcome by noting that when we refer to P(i), we are referring to the node that was actually the parent of node i, even though after a rotation on i it may not be the parent anymore. In such a

[3]Three may be sufficient if the pointers upward towards the root are maintained consistently while the downward traversal to the accessed node is being performed. However, throughout this paper we shall assume that we will not overwhelm ourselves with such additional "book-keeping" operations.

case we still refer to that node as P(i), but the actual physical parent of i will be referred to as P(i)'.

We now proceed to define our rotation criterion. Let θ_i be $\kappa_{P(i)} - \kappa_i'$. θ_i is a criterion function which tells us whether performing a rotation at node i will reduce the κ-value at P(i) or not. We shall prove that the κ-value of the entire tree is reduced by a rotation if and only if $\theta_i > 0$. We call such a rotation a κ-lowering rotation.

Theorem II.

Any κ-lowering rotation performed on any node in the tree will cause the weighted path length of the entire tree to decrease.

Proof :

We consider three mutually exclusive and exhaustive cases, listed as (i), (ii), and (iii). We suppose that node i has been accessed.

Case (i) : Node i is the root of the tree. In this case, the proof is trivial, as P(i) is the null pointer, and node i will never rotate upwards. Thus no κ-lowering rotation can ever be performed.

Case (ii) : P(i) is the root of the tree. Here minimizing θ_i is equivalent to minimizing the WPL of the entire tree and the result follows since the WPLs considered in the decision process are actually the WPLs of the entire tree.

Case (iii) : Neither i nor P(i) is the root of the tree.

Let j be the node that becomes the parent of the accessed node **after** the rotation has been performed (i.e. j = P(i)'). Assume that the quantities α, τ, and κ have been updated at nodes i and P(i). Observe that j P(i). Then at node j, due to (5),

$$\kappa_j = \alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{jL} + \kappa_{jR}.$$

Consider the case when node i is in the left subtree of T_j. We know from Theorem I that the quantities α_j, τ_{jR}, and κ_{jR} will remain unchanged as a result of the rotation performed on node i. We also know that as a result of Theorem I and the properties of the rotation, the latter operation will not cause a change to the total number of accesses to the left subtree. Hence,

$$\begin{aligned}\theta_i &= \kappa_j - \kappa_j' = (\alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{jL} + \kappa_{jR}) - (\alpha_j' + \tau_{jL}' + \tau_{jR}' + \kappa_{jL}' + \kappa_{jR}') \\ &= \alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{jL} + \kappa_{jR} - \alpha_j - \tau_{jL} - \tau_{jR} - \kappa_{jL}' - \kappa_{jR} \\ &= \kappa_{jL} - \kappa_{jL}'\end{aligned}$$

This implies that if a rotation is performed, then $\kappa_{jL} - \kappa_{jL}'$ is greater than zero, resulting in the quantity $\kappa_j - \kappa_j'$ itself being greater than zero. By induction it can be seen that this relationship between $\kappa_a - \kappa_a'$ also holds for every ancestor a of j and thus this inequality bubbles itself up recursively at every level of the tree and ultimately for the root of the entire tree itself. This means

that a κ-lowering rotation performed anywhere in the tree will lower the WPL of the entire tree. The arguments are identical if i is in the right subtree of T_j. Combining both of the above cases yields the desired result.

II.2.1 Remarks

(i) As stated earlier, the importance of this theorem lies in the fact that the decision to rotate or not may be made at the **level of the rotation itself**. It is thus not necessary to backtrack along the access path up to the root of the tree, and thus we can obtain a significant reduction in the amount of time required to calculate the α, τ, and κ values and to perform a restructuring operation. Amazingly enough, the restructuring operation now requires only **constant** time.

(ii) For the purpose of deciding whether or not to rotate, the value of κ' at i may be found without actually performing the rotation. This is done by utilizing the algebraic properties of κ as given by (3) and Lemma I, and using the values at the nodes that **would be** the left and right children of i should the rotation be performed.

For the sake of completeness, we present our access and reorganizing algorithm (called CON_ROT_Naive) based on these results in Appendix I[4]. This algorithm is recursive and covers all the possible cases for any node j. Notice that the τ values are updated on entry, i.e., on the way down the tree, and the κ values are updated on the path back up. The improvement of this scheme over the previously developed schemes in that it reorganizes itself only if it results in a decrease in the WPL of the entire tree, is also clear. Furthermore, the method also takes **all** of the other records in the tree into consideration when a decision is made whether or not to reorganize, in contrast to the monotonic tree scheme, which considers only the accessed record and its parent.

We shall now introduce an optimized version of the conditional rotation heuristic.

III. THE OPTIMIZED CONDITIONAL ROTATION HEURISTIC

Up to this point we have developed a scheme for dynamically restructuring BSTs by considering whether the WPL of the entire tree decreases as a result of performing the operation, and performing any reorganization only if the WPL does actually decrease as a result of the operation. We also introduced a method to determine whether or not to rotate based merely on a criterion specified in terms of the path length at the **parent** of the node that was accessed.

In this section, we shall present a scheme which performs tree reorganization using the same heuristic that was used in the preceding section. However, we shall do this by making a

[4]Since a space-optimizing version of this technique will be presented in the next section, a formal algorithmic submission of this version is probably superfluous. Taking the risk of being marginally repetitive we have however opted to include it so that the reader can understand how the various counters are updated either during the path down the tree or during the path back to the root. Also, the strategy by which the usefulness of a rotation can be anticipated can be rendered more explicit.

single pass through the tree and also by maintaining only a single counter. Thus we will be able to save greatly in both time and space.

First of all we shall show that we can obtain a criterion function which does involve the κ values. Furthermore, as it turns out, not only can we remove the κ values, but we can also eliminate the need for the α values. It is obvious from (1) that the information stored in the α-values is also stored in the τ values, and may be readily extracted. This means that the α values do not have to be explicitly stored. What is not quite so obvious is the fact that the information stored in the τ fields is sufficient to determine whether or not a rotation should be performed. To show this we define a new criterion function ψ which is dependent entirely upon the τ values, in contrast to the previous criterion function θ, which was dependent upon the α, τ, and κ values. Whereas previously, we performed a rotation if the criterion function θ_i was positive, we now perform the rotation if the function ψ_i is positive. We will show that indeed, performing a rotation based on the criterion function ψ_i is equivalent to performing a rotation based on the criterion function θ_i.

Theorem III.

Let i be the accessed node of the BST, and let $\kappa_{P(i)}$ be the WPL of the tree rooted at the parent P(i) if no rotation is performed on node i. Let κ_i' be the WPL of the tree rooted at node i if the rotation is performed. Furthermore, let ψ_i be defined as follows:

$\psi_i = \alpha_i + \tau_{iL} - \alpha_{P(i)} - \tau_{B(i)}$ if i is a left child;

$\psi_i = \alpha_i + \tau_{iR} - \alpha_{P(i)} - \tau_{B(i)}$ if i is a right child.

Then, if $\theta_i = \kappa_{P(i)} - \kappa_i'$, ψ_i 0 if and only if θ_i 0.

Proof :

It is required to prove that ψ_i 0 if and only if θ_i 0. This will indeed imply that performing a rotation operation if ψ_i 0 is equivalent to performing a rotation operation if θ_i 0. Actually, we show a stronger result, which is that θ_i + ψ_i.

We give the proof for the case that node i is a left child; the case that node i is a right child may be proven in exactly the same way.

Suppose that an access is performed on node i, and that the α, τ, and κ values are updated appropriately for nodes i and P(i). Then from the recursive expression for κ given in Lemma II,

$$\kappa_{P(i)} = \alpha_{P(i)} + \tau_i + \tau_{B(i)} + \kappa_i + \kappa_{B(i)}. \quad (8)$$

Suppose that at this point, a rotation is applied to node i in an upwards direction. The resulting tree is that in Figure Ib, and the new expression for κ_i' is

$$\kappa_i' = \alpha_i + \tau_{iL} + \kappa_{iL} + \tau_{P(i)}' + \kappa_{P(i)}'. \quad (9)$$

The quantities $\tau_{P(i)}'$ and $\kappa_{P(i)}'$ may be expanded to give

$$\tau_{P(i)}' = \alpha_{P(i)} + \tau_{iR} + \tau_{B(i)} \ , \quad \text{and,} \quad (10)$$

$$\kappa_{P(i)}' = \alpha_{P(i)} + \tau_{iR} + \tau_{B(i)} + \kappa_{iR} + \kappa_{B(i)}. \qquad (11)$$

Substituting (10) and (11) into (9), we get

$$\kappa_i' = \alpha_i + \tau_{iL} + \kappa_{iL} + 2\alpha_{P(i)} + 2\tau_{iR} + \kappa_{iR} + \kappa_{B(i)}.$$

But from the recursive formulation of κ, we know :

$$\kappa_i = \alpha_i + \tau_{iL} + \tau_{iR} + \kappa_{iL} + \kappa_{iR}.$$

Thus, θ_i has the form :

$$\theta_i = \kappa_{P(i)} - \kappa_i' = \tau_i - \alpha_{P(i)} - \tau_{iR} - \tau_{B(i)}.$$

Observe now that τ_i is the value before the rotation was performed. When it is replaced by its equivalent in terms of quantities which are not changed by the rotation operation, we get

$$\tau_i = \alpha_i + \tau_{iL} + \tau_{iR},$$

whence, $\theta_i = \kappa_{P(i)} - \kappa_i' = \alpha_i + \tau_{iL} - \alpha_{P(i)} - \tau_{B(i)} = \psi_i$

and the theorem is proved.

III.1 Remarks

(i) As stated in the preamble to this section, we can use the criterion function θ_i computed at the **node i** to decide whether or not to restructure the tree. But now we have shown that rotating based on the function ψ_i is equivalent to minimizing the WPL of the overall tree. Unlike θ_i, however, ψ_i only requires the use of the information stored in the α and τ fields in each record. This implies that we do not need to maintain the κ fields at all, and this in turn implies that after the search for the desired record and after performing any reorganization (which takes constant time), we do not need to retrace our steps back up the tree to update the κ values of the ancestors of i. It also pays to note that at any node i,

$$\tau_i = \alpha_i + \tau_{iL} + \tau_{iR} \text{ ; and, } \qquad \alpha_i = \tau_i - \tau_{iL} - \tau_{iR}.$$

This implies that the α values may be expressed in terms of the τ values, and so the former are redundant and they too need not be maintained. Thus we have achieved what we had hoped to by maintaining only one extra memory location per node and simultaneously rendering the second upward pass superfluous. This modified algorithm, which is a space optimizing version of CON_ROT_Naive, is given in Appendix II as Optimized_CON_ROT. Notice that the τ values are updated on the path down the tree (i.e. on entry) during the search. Also, we have expressed ψ_i in terms of τ values by converting the α values to their equivalents in terms of τ.

(ii) Since we have shown that the decision to rotate may be made equivalently by considering either the criterion for θ_i or for ψ_i , Theorem III has the following corollary.

Corollary I.

Algorithms CON_ROT_Naive and Optimized_CON_ROT are stochastically equivalent.

Note that it is not just the average performance (or the asymptotic performance) of these algorithms which is equivalent ; but both of them work in lock-step. Thus given the same initial tree and the same sequence of accesses, both the algorithms restructure the tree identically.

Thus far we have discussed only the effect of these rotations on the WPL of the tree. We conclude this section by specifying the effect of reducing the WPL on the **cost** of the entire tree.

Theorem IV

The normalized weighted path length of a BST asymptotically becomes an arbitrarily good approximation of the actual cost of the tree with an arbitrarily high probability.

Proof:

The cost of a tree T at time n was shown to be

$$C_T(n) = \sum_{j \in T} s_j(n) \cdot \lambda_j(n).$$

But by the law of large numbers, s_i can be estimated by the ratio $\left(\alpha_j(n) \Big/ \sum_{j \in T} \alpha_j(n)\right)$.

But $\sum_{j \in T} \alpha_j(n)$ is identically equal to $\tau(n)$ which is the sum of the α's of the nodes in the tree.

Thus, asymptotically as n tends towards ,

$$C_T(n) = \lim_{n \to \infty} \left(\frac{1}{\tau_T(n)} \left[\sum_{R_i \in T} \lambda_i(n) \alpha_i(n) \right] \right)$$

$$= \lim_{n \to \infty} \left(\frac{1}{\tau_T(n)} \cdot \kappa_T(n) \right)$$

Thus, any algorithm which reduces the normalized WPL of the entire tree, also asymptotically causes C_T to decrease as well.

To sum up, we would like to highlight the sequence of theoretical results we have presented concerning our BST restructuring algorithm. First of all the recursive properties of the three indices used for conditional rotations have been stated, and the techniques for updating them in all scenarios have been specified. Subsequently, the existence of a local (nodal) criterion function has been proven ; using this function a decision rule which reports whether a rotation of the accessed node is beneficial in reducing the overall cost of the entire tree has been presented. Thus, we have shown that the effect of a rotation can be anticipated without performing multiple

traversals on the current BST, and hence the restructuring can be rendered both asymptotically optimal in terms of the cost and computationally optimal in terms of the local pointer manipulations. Apart from proving the existence of this function the actual functional form of this criterion has been derived. Furthermore, the existence and the form of an analogous function which utilizes only a single index has also been derived. Finally, we have shown that the algorithms given here asymptotically reduce the cost of the entire tree.

We have also experimentally compared our scheme to many of the other schemes reported in the literature. The results which are currently available compare our scheme with (i) Allen and Munro's Move-to-Root scheme [2], (ii) Sleator and Tarjan's splay tree [19], (iii) the optimal tree constructed using Knuth's algorithm [10], (iv) a height-balanced tree [1], (v) the monotonic tree described by Bitner [5], and (vi) the nearly-optimal tree constructed using the technique due to Walker and Gotlieb [20]. In comparing these algorithms the simulations were conducted for four types of distributions[5] and in each case the number of nodes was varied from 15 to 511. A fair bit of work has still to be done to compare the scheme with Mehlhorn's D-Tree, the BB[α] tree and Gonnet's path length balancing tree[6] [7].

Our initial experiments [6] seem to indicate that, generally speaking, our scheme outperforms **all** of the other static and dynamic BST schemes mentioned above, excepting, of course, the static optimal tree itself. The results obtained for the Zipf's law distribution are typical. In this case, for trees with 15 nodes, Allen and Munro's Move-to-Root scheme had a cost which was approximately 20.5% greater than the static optimal tree, Sleator and Tarjan's splay tree had a cost approximately 23.5% greater than the static optimal, and the balanced tree was approximately 17% more expensive than the static optimal tree scheme. The results for the monotonic tree verified the fact that it is a poor scheme, as it proved to have an average cost which was 55% greater than the average cost of the static optimal tree. The nearly optimal tree presented by Walker and Gotlieb was not as nearly optimal as it should have been, giving an average cost which was 16.1% greater than the optimal tree's cost. Our scheme had an average cost that was only 3.6% greater than the average cost of the static optimal tree. These results are typical for each of the distributions used.

Various results are currently also available for the above algorithms when the number of nodes in the tree is more realistic and realistic versions of the above four distributions are utilized. In this case too our scheme seems to be far superior to **all** of the other above-mentioned static and

---

[5]These distributions are the Zipf's, the exponential and two families of wedge distributions. In each simulation 100 parallel experiments were done for a large number of accesses so that a statistically dependable ensemble average could be obtained.

[6]Strictly speaking, this comparison would not be so interesting because Gonnet's scheme is not an "adaptive" scheme. However, in one sense, our scheme can be thought of as a generalized "adaptive" version of Gonnet's scheme that is valid for **non-uniform** distributions. The non-uniformity of the distribution would transform Gonnet's path length criterion to be our weighted path length criterion.

dynamic BST schemes , excepting the static optimal tree. In all brevity we now quote the results obtained for a "Realistic" Zipf's distribution[7] when the number of nodes in the tree is 511. In this case, Allen and Munro's Move-to-Root scheme had a cost which was approximately 29.45% greater than the static optimal tree, Sleator and Tarjan's splay tree had a cost approximately 33.89% greater than the static optimal, and the balanced tree was approximately 33.42% more expensive than the static optimal tree scheme. The monotonic tree behaved very poorly -- it had an average cost which was 347% greater than the average cost of the static optimal tree. Our scheme proved to have an average cost that was only 2.42% greater than the average cost of the static optimal tree. Indeed, these results seem to indicate that our strategy is at least an order of magnitude closer to the optimal than the other BST techniques mentioned above. Indeed, the results presented for the exponential distribution are even more remarkable, especially when our method is compared to the Move-to-Root scheme and the splay tree.

It must also be observed that augmented with the fact that the overall expected cost of our strategy is less than the corresponding cost of the other dynamic schemes simulated, the former has the advantage that the data reorganization is performed conditionally. Thus, unlike the latter schemes, a data reorganization is not executed every time an interior node in the tree is accessed. Furthermore, should such a reorganization be executed, whereas our scheme will perform **only** one rotation, the other dynamic schemes could potentially move the accessed element all the way to the root. Thus, from our experience, it is our stand that our technique is superior in terms of both the real execution time and the accuracy obtained, where the accuracy is quantified in terms of the closeness of the tree to the optimal one. We believe that the current on-going study comparing our scheme with Mehlhorn's and Gonnet techniques will give us additional insight into the importance of path length computations and "local" nodal computations in any BST reorganizing strategy.

V. CONCLUSIONS AND OPEN PROBLEMS

In this paper we have introduced a new strategy requiring extra memory which attempts to reorganize a BST to an optimal form. It requires three extra memory locations per record. One counts the number of accesses to that record, a second contains the number of accesses to the subtree rooted at that record, and the third contains the value of the WPL of the subtree rooted at that record. After a record is accessed and these values updated, the record is rotated one level upwards **if** the WPL of the entire **tree** (and not just the subtree rooted at the node) decreases as a result of the rotation. We have shown that this implies that the cost of the entire tree decreases

[7]The term "Realistic" is used because the original distributions would have almost zero probability masses -- less than what the architecture of the processor would allow -- for the less frequently accessed elements especially when the number of nodes is large. We have marginally "skewed" the distributions to force the least frequently elements to have the smallest probability masses permitted by the architecture of the processor.

asymptotically. To achieve this, we perform a straightforward evaluation of a local criterion function and anticipate whether a rotation will yield this result. Thus we have succeeded in presenting a technique by which we can locally decide whether to restructure the tree or not. As well, we have presented an optimized version of this algorithm which requires us to maintain only one extra memory location per node. Both these algorithms and are stochastically equivalent.

Among the open problems that still exist are the analysis of the stochastic performance of this algorithm and its behaviour under various distributions. This problem is by no means trivial. Unlike the underlying Markov Chains of other techniques that have been analyzed in the literature, the chain of this technique is conceptually completely different. Since the rotation is done conditionally, the probability of a tree being transformed into another is conditional on the time-varying contents of the indices and the criterion functions, and thus the chain is not stationary. For the first part the number of states of the chain is of the same order as the number of distinct BSTs. Furthermore, the actual transition probabilities themselves are **time-varying** random variables, and to our knowledge there are no known techniques available in the mathematical literature which even discuss how to tackle such a problem except for some trivial distributions**.** The rate of convergence of the scheme is also unknown. A variation of this scheme that requires sub-linear memory is currently being investigated.

ACKNOWLEDGEMENTS

The authors are grateful to Mr. Zgierski of the School of Computer Science at Carleton for doing some of the programming involved in the simulations. We are also immensely grateful to the both the anonymous referees for their meticulous reviews. Our gratitude to them for suggesting various improvements on the previous version cannot be adequately expressed.

APPENDIX I

Conditional Rotation algorithm using τ, α, κ. The algorithm was discussed in Section II.

Algorithm CON_ROT_Naive { CONditional_ROTation Reorganization }
Input : A binary search tree T and a search key k_i assumed to be in T.
Output: (i) the restructured tree T', and (ii) a pointer to record i containing k_i
Method :

```
τ_j ♦ τ_j + 1                 { increment τ for the present node }
If k_i = k_j Then             { This is the record we want }
    α_j ♦ α_j + 1
    calculate κ_P(j), κ_j' using Lemma I
    If κ_P(j) - κ_j' > 0 Then
        rotate node j upwards
```

```
            recalculate κP(j), κj, τP(j), τj
        Endif
    Else                    { Search the subtrees }
        If ki < kj Then
            perform CON_ROT_Naive on j^.Leftchild
        Else
            perform CON_ROT_Naive on j^.Rightchild
        Endif
        recalculate κj
    Endif
    Return record i
End Method
END Algorithm CON_ROT_Naive.
```

APPENDIX II

Space optimized version of the conditional rotation algorithm.

Algorithm Optimized_CON_ROT
Input : A binary search tree T and a search key k_i assumed to be in T.
Output : The restructured tree T', and a pointer to record i containing k_i
Method :

```
τj ♦ τj + 1            { update τ for the present node }
If ki = kj Then        { Found the record in question }
    If node j is a left child Then
        ψj ♦ 2τj - τjR - τP(j)
    Else
        ψj ♦ 2τj - τjL - τP(j)
    Endif
    If ψj > 0 Then
        rotate node j upwards
        recalculate τj, τP(j)
    Endif
    return record j
Else
    If ki < kj Then    { Search the subtrees }
        perform Optimized_CON_ROT on j^.Leftchild
    Else
        perform Optimized_CON_ROT on j^.Rightchild
    Endif
Endif
End Method
END Algorithm Optimized_CON_ROT
```

REFERENCES

[1] G. M. Adel'son-Velski'i and E.M. Landis, "An algorithm for the organization of information", *Sov. Math. Dokl.*, Vol. 3 pp. 1259-1262, 1962.

[2] B. Allen and I. Munro, "Self-organizing binary search trees", *Journal of the ACM*, Vol. 25, pp.526-535, 1978.

[3 D. M. Arnow and A. M. Tenenbaum, "An investigation of the move-ahead-k rules", *Congressus Numerantium, Proceedings of the Thirteenth Southeastern Conference on Combinatorics, Graph Theory and Computing*, Florida, pp.47-65, Feb. 1982.

[4] P. J. Bayer, "Improved bounds on the costs of optimal and balanced binary search trees", *MAC Technical Memo-69*, Nov. 1975.

[5] J. R. Bitner, "Heuristics that dynamically organize data structures", *SIAM Journal of Comput.*, Vol. 8, pp.82-110, 1979.

[6] R. P. Cheetham, B. J. Oommen and D. T. H. Ng, *Adaptive Structuring of Binary Search Trees Using Conditional Rotations*, Technical Report available as SCS-TR-126 from the School of Computer Science, Carleton University, Ottawa, Canada : K1S 5B6.

[7] G. H. Gonnet, "Balancing Binary Trees by Internal Path Reduction", *Communications of the ACM*, Vol. 26, pp.1074-1081, 1983.

[8] G. H. Gonnet, J. I. Munro and H. Suwanda, "Exegesis of self-organizing linear search", *SIAM Journal of Comput.*, Vol. 10, pp.613-637, 1981.

[9] H. J. Hester and D. S. Hirschberg, "Self-organizing Linear Search", *ACM Computing Surveys*, pp. 295-311, 1976.

[10] Knuth, D. E., *The Art of Computer Programming, Vol. 3*, Addison-Wesley, Reading, Ma., 1973.

[11] T. W. Lai and D. Wood, "A Relationship between Self Organizing Lists and Binary Search Trees", *Proceedings of the 1991 International Conference on Computing and Information*, Ottawa, pp. 111-116, May 1991.

[12] J. McCabe, "On serial files with relocatable records", *Operations Research*, Vol. 12, pp.609-618, 1965.

[13] K. Mehlhorn, "Nearly optimal binary search trees", *Acta Informatica*, Vol. 5, pp.287-295, 1975.

[14] K. Mehlhorn,"Dynamic Binary Search", *SIAM Journal of Comput.*, Vol. 8, pp.175-198, 1979.

[15] K. Mehlhorn, *Data Structures and Algorithms 1 : Sorting and Searching*, Springer Verlag, Berlin, 1984.

[16] B. J. Oommen and E. R. Hansen, "List organizing strategies using stochastic move-to-front and stochastic move-to-rear operations", *SIAM Journal of Comput.*, vol.16, No.4, pp. 705-716.

[17] B. J. Oommen, E. R. Hansen and J. I. Munro, "Deterministic Optimal and Expedient Move-to-Rear List Organizing Strategies", *Theoretical Computer Science*, Vol. 74, pp. 183-197, 1990.

[18] R. L. Rivest, "On self-organizing sequential search heuristics", *Communications of the ACM*, Vol. 19, pp.63-67, 1976.

[19] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees", *Journal of the ACM*, Vol. 32, pp.652-686, 1985.

[20] W. A. Walker and C. C. Gotlieb, "A top-down algorithm for constructing nearly optimal lexicographical trees", in *Graph Theory and Computing*, Academic Press, New York, 1972.

LIST OF INDEX TERMS

Adaptive Data Structures, Binary Search Trees, Dynamic Data Structures, Move-to-Root Heuristic, Self Organizing Structures, Splay Trees.

LIST OF FOOTNOTES

[+] Partially supported by the Natural Sciences and Engineering Research Council of Canada. A preliminary version of *some* of these results was presented at the 1988 International Conference on Database Theory, in Bruges, Belgium.

[@] Senior Member, IEEE

[*] All the authors can be contacted at the School of Computer Science, Carleton University, Ottawa; Canada : K1S 5B6.

[1]Bitner's analysis assumes that the set of access probabilities is randomly assigned to the records, and thus that each permutation of **S** is equally likely. We are grateful to an anonymous referee who pointed out that this assumption is unrealistic.

[2]This scenario takes care of the cases when the left and right subtrees are not empty. In the case of empty subtrees, the expansion is trivially valid since the corresponding α values would be identically equal to zero.

[3]Three may be sufficient if the pointers upward towards the root are maintained consistently while the downward traversal to the accessed node is being performed. However, throughout this paper we shall assume that we will not overwhelm ourselves with such additional "book-keeping" operations.

[4]Since a space-optimizing version of this technique will be presented in the next section, a formal algorithmic submission of this version is probably superfluous. Taking the risk of being marginally repetitive we have however opted to include it so that the reader can understand how the various counters are updated either during the path down the tree or during the path back to the root. Also, the strategy by which the usefulness of a rotation can be anticipated can be rendered more explicit.

[5]These distributions are the Zipf's, the exponential and two families of wedge distributions. In each simulation 100 parallel experiments were done for a large number of accesses so that a statistically dependable ensemble average could be obtained.

[6]Strictly speaking, this comparison would not be so interesting because Gonnet's scheme is not an "adaptive" scheme. However, in one sense, our scheme can be thought of as a generalized "adaptive" version of Gonnet's scheme that is valid for **non-uniform** distributions. The non-uniformity of the distribution would transform Gonnet's path length criterion to be our weighted path length criterion.

[7]The term "Realistic" is used because the original distributions would have almost zero probability masses -- less than what the architecture of the processor would allow -- for the less frequently accessed elements especially when the number of nodes is large. We have marginally "skewed" the distributions to force the least frequently elements to have the smallest probability masses permitted by the architecture of the processor.

LIST OF FIGURE CAPTIONS

Figure Ia : The tree before a rotation is performed. The contents of the nodes are their data values, in this case the characters {a, b, c, d, e}.

Figure Ia : The tree after a rotation is performed on node i. Observe the properties stated in Section II.

BIOGRAPHIES OF AUTHORS

R. P. Cheetham

Robert Cheetham obtained his Bachelor in Computer Science (B.C.S) degree from Carleton University in 1989. He is currently a member of the scientific staff at Bell Northern Research Ltd. in Ottawa. He is a member of the IEEE and of the IEEE Computer Society.

B. John Oommen

John Oommen obtained his B.Tech. degree from the Indian Institute of Technology, Madras, India in 1975. He obtained his M.E. from the Indian Institute of Science in Bangalore, India in 1977. He then went on for his M.S. and Ph. D. which he obtained from Purdue University, in West Lafayette, Indiana in 1979 and 1982 respectively. He joined the School of Computer Science at Carleton University in Ottawa, Canada, in the 1981-82 academic year. He is still at Carleton and holds the rank of a Full Professor. His research interests include Automata Learning, Adaptive Data Structures, Statistical and Syntactic Pattern Recognition, Stochastic Algorithms and Partitioning Algorithms. He is the author of over ninety refereed journal and conference publications and is a senior member of the IEEE.

D. T. H. Ng

David Ng obtained his Bachelor in Computer Science (B.C.S) degree from Carleton University in 1987 and went on to get his Masters in Computer Science (M.C.S) degree from Carleton 1989. He worked for a few years in TYDAC Inc. in Ottawa, and then moved on to Bell Northern Research Ltd. in Ottawa where he is currently employed. His current research interests include fiber optics and telecommunications.