ADAPTIVE STRUCTURING OF BINARY SEARCH TREES USING CONDITIONAL ROTATIONS

by R.P. Cheetham*, B.J. Oommen*
and D.T.H. Ng*

SCS-TR-126

October 1987

School of Computer Science
Carleton University
Ottawa, Ontario
CANADA K1S 5B6

Partially supported by the Natural Sciences and Engineering Research Council of Canada.

*School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

ADAPTIVE STRUCTURING OF BINARY SEARCH TREES USING CONDITIONAL ROTATIONS+

R. P. Cheetham*
B. J. Oommen*
D. T. H. Ng*

ABSTRACT

Consider a set $\mathfrak{A} = \{A_1, A_2, \ldots, A_N\}$ of records, where each record is identified by a unique key. The records are accessed based on a set of access probabilities $\mathfrak{S}=\{s_1,s_2, \ldots, s_N\}$ and are to be arranged lexicographically using a binary search tree. If $\mathfrak{S}$ is known *a priori* , it is well known [7] that an optimal binary search tree may be constructed using $\mathfrak{A}$ and $\mathfrak{S}$. We consider the case when $\mathfrak{S}$ is not known *a priori* . A new restructuring heuristic is introduced that requires three extra integer memory locations per record, and this restructuring of the tree is performed **only** if it decreases the weighted path length of the overall resultant tree. We also present a space optimized version of the latter restructuring mechanism which requires only one extra integer field per record. We show that the cost of the tree is reduced by each restructuring operation, and present experimental results to demonstrate the superiority of our algorithm over all other efficient static and dynamic schemes that exist in the literature.

---

* School of Computer Science, Carleton University, Ottawa, K1S 5B6, Canada.

+ Partially supported by the Natural Sciences and Engineering Research Council of Canada.

I. INTRODUCTION

A binary search tree may be used to store records whose keys are members of an ordered set $\mathfrak{A} = \{A_1, A_2, \ldots, A_N\}$. 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 $\mathfrak{A}$, the set of records, and their set of access probabilities $\mathfrak{S}=\{s_1,s_2,\ldots,s_N\}$, the problem of constructing an optimal binary search tree has been extensively studied. The most well known scheme, due to Knuth [7], uses dynamic programming techniques and produces an optimal binary search tree using $O(N^2)$ time and space. Alternatively, Walker and Gotlieb [14] have used dynamic programming and divide and conquer techniques to yield a nearly optimal binary search tree using O(N) space and O(N log N) time. In this paper, we study the problem in which $\mathfrak{S}$, the access probability vector, is **not** known *a priori*. In this case, a scheme which dynamically rearranges itself and generates a tree which tends towards an optimal configuration is desirable.

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 the list, after a long enough period of time, 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 memoryless schemes have been developed to reorganize a linear list dynamically. Among these are the **move-to-front** rule [8], due to McCabe, which moves the record which has been accessed to the front of the list, and the **transposition** rule [12], also due to McCabe and discussed by Rivest, which exchanges the record accessed with the record preceding it.

As well, schemes involving the use of extra memory have been developed. The first scheme, and indeed one of the initial schemes developed for rearranging linear lists to an optimal form [7], is one in which each element of the list contains a counter which records the number of times that that particular element has been accessed. The list is arranged in such a way that the most frequently accessed element becomes the head of the list, and the rest of the list is ordered similarly. By the law of large numbers this scheme will, over a long enough period of time, converge to the optimal ordering.

Another scheme which requires extra memory is the stochastic move-to-rear rule [10], due to Oommen and Hansen, which moves the accessed element to the rear with a probability which decreases each time the element is accessed. As well, a stochastic move-to-front [10] and a deterministic move-to-rear scheme [11] were also developed by the same authors. Details of these schemes and others which have been developed may be found in the literature [3,5,6,7,8,10,11,12]. We refer the reader to the latter papers and in particular to the review paper by Gonnet et al. [6], which are excellent surveys of the list organizing strategies that have been researched.

A binary search tree is not quite so simple to reorganize as a linked list for the following reasons. Firstly, the lexicographic ordering property of the binary search tree must be preserved throughout the reorganization of the tree. This can sometimes be in conflict with the request to move a record upwards in the tree. As well, a restructuring operation in a binary search tree may move more than just one record; it could also move an entire subtree either up or down in the tree. The above reasons preclude a direct application of any of the successful list organizing schemes because these schemes take no consideration for any ordering property that might have to be maintained among the elements, and they also take no consideration for the fact that the subtree representation of a sublist may also have to be reorganized. An example of these conflicting requirements is illustrated in Figure 1, in which the transposition rule is the primitive list organizing operation.

********** **Insert Figure 1 here** **********

In order to preserve the binary search property, **rotation** is the primitive tree restructuring operation. This operation is well known [1], and tree reorganizing schemes using this operation have been developed which are analogous to various list organizing methods discussed in the literature. The description of the rotation operation and its properties is included in all brevity in the appropriate sections of the body of this paper.

A few memoryless tree reorganizing schemes exist in the literature. Allen and Munro presented the first scheme of this type [2], which is analogous to the move-to-front rule. This scheme uses rotations to move the accessed record up to the root of the tree, and the rule is hence called the **move-to-root** scheme. They also

developed another scheme called the **simple exchange** rule, which rotates the accessed element one level towards the root, similar to the transposition rule. Contrary to the case of lists, where the transposition rule is better than the move-to-front rule [12], 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 binary search tree, the simple exchange heuristic does not have this property. Due to this property of the latter heuristic, which was thoroughly covered by Allen and Munro, we reckon that it is superfluous to consider this scheme in any further detail. Sleator and Tarjan [13] introduced a third scheme, which also moves the accessed record up to the root of the tree. This scheme, which uses a restructuring move called **splaying**, differs from that of Allen and Munro's in that it uses different variations of single and double rotations (see Bitner [5]), unlike the move-to-root scheme which applies only successive single rotations. Both of these methods require O(N) space for the records of the tree and **no** extra space for the reorganization process, and both require, on the average, O(log N)* time. However, our experimental results seem to indicate that the move-to-root scheme has a slightly lower cost than the splaying scheme.

One interesting and intuitively appealing scheme requiring extra memory is the **monotonic tree** scheme, first proposed by Knuth [7] as a possible method to structure a tree to obtain a nearly optimal tree. The monotonic tree was later analyzed by Bitner [5] as a dynamic reorganization tactic. In a monotonic tree the root of every subtree has the property that it is the most probable node among the nodes of the subtree. Indeed, this definition is recursive. For the dynamic version, each record contains a counter indicating the number of times it has been accessed, and after each access the record is rotated upwards until it reaches the root or until its parent has a larger access count. Over a long enough period of time, due to the law of large numbers, the tree that is obtained converges to the monotonic tree, with the most probable key at the root, and the subtrees ordered in similar fashion.

This strategy is very similar to that used in organizing linear lists, for which the optimal

---

* This is based on the fact that a binary search tree has an average path length of O(log N) if each of the possible binary search trees that can be constructed from N nodes are equally likely. We make this assumption throughout the paper. Note that when we refer to log N, unless stated otherwise, log N ≡ $\log_2$ N.

arrangement is that with the most probable element at the front, and the rest of the list ordered recursively in the same manner. Although this arrangement is the optimal one in the case of a linear list, it need not be optimal for a binary search tree. Indeed, Bitner shows that in contrast to the case of reorganizing a linear list, the monotonic tree scheme may be expected to behave quite poorly, especially for distributions which are highly "non-exponential". In such cases a search for a selected item in a monotonic tree may require O(N) time, which is the same time requirement of a linear list.

In this paper we will introduce a new heuristic that attempts to reorganize a binary search tree to an optimal form. It requires three extra memory locations per record. One will count the number of accesses to that record, a second will count the number of accesses to the subtree rooted at that record, and the third will contain the value of the weighted path length of the subtree rooted at that record. After a record is accessed and these values updated, the record will be rotated upwards once **if** the weighted path length of the entire **tree** (and not just the subtree rooted at the node) decreases. We shall show that this implies that the cost of the entire tree decreases as a result. Our experimental results show that this method is far superior to all other dynamic restructuring methods, and produces a tree which is very nearly optimal.

In the next section, we will describe in more detail the existing self-organizing schemes, their performance and their drawbacks, as a foundation for the new scheme. In doing so, we shall not just superficially describe the various schemes and their modes of operation, but shall also discuss in some detail, the theoretical results pertaining to each of them. Thus we intend to render this paper a brief comprehensive survey of the results available in this field.

II. PREVIOUSLY KNOWN METHODS

We note first of all that throughout **all** of the binary search tree dynamic restructuring methods, the primitive operation is the rotation, which was introduced first by Adel'son-Velski'i and Landis [1]. To clarify the discussion, we describe the rotation operation as follows.

Suppose that there exists a node i in a binary search tree, 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 2a). 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 position (see Figure 2b). 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 2b). The properties of this operation will be described in more detail in the next section.

**********Insert Figures 2a and 2b here **********

II.1 The Move-to-Root Heuristic

This scheme was historically the first self-organizing binary search tree scheme in the literature. Allen and Munro developed this heuristic that aimed to restructure a binary search tree dynamically, and attempted to maintain the tree in a nearly optimal form. They assumed that each element of the tree will be requested with a fixed but unknown probability. Furthermore, the access probability of any node is assumed to be independent of the access probabilities of all the other nodes.

Their heuristic is conceptually quite 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 (Figure 3). The idea is that a frequently accessed record will be close to the root of the tree as a result of its being frequently moved to the root, and this will minimize the cost of the search and the retrieval operations.

********** Insert Figure 3 here **********

Let $c_{MR}(\mathfrak{S})$ be the asymptotic average cost of retrieving a record under a move-to-root scheme for the distribution $\mathfrak{S}$. Similarly, let $c_{OPT}(\mathfrak{S})$ be the asymptotic average cost of retrieving a record for the static optimal binary search tree for the distribution $\mathfrak{S}$. Then, we define the cost of retrieving a record A in tree T (where A is in the tree) as $c_T(A)$, where,

$$c_T(A) = 1 + (\text{length of path in T from the root to A}). \qquad (1)$$

We assume that if we are given N records lexicographically arranged as $A_1 < A_2$

$< \ldots < A_N$, which are requested with a probability distribution $\mathfrak{S}$, then the access probabilities satisfy the equation below :

$$\sum_{i=1}^{N} s_i = 1.$$

Let C(T) be the cost of a binary search tree containing $\mathfrak{A}$, where,

$$C(T) = \sum_{i=1}^{N} s_i \, c_T(A_i) . \tag{2}$$

Allen and Munro proved the following bounds for the move-to-root scheme :

I :

$$c_{MR}(\mathfrak{S}) = 1 + 2 \sum_{1 \le i < j \le n} \frac{s_i s_j}{s_i + \ldots + s_j} . \tag{3}$$

II :

$$c_{MR}(\mathfrak{S}) \le (2 \ln 2) \cdot H(\mathfrak{S}) + 1, \text{ where,}$$

$$H(\mathfrak{S}) = \sum_{i=1}^{N} (-s_i \log s_i) . \tag{4}$$

III :

$$\begin{aligned} c_{MR}(\mathfrak{S}) &< (2 \ln 2) \, (c_{OPT}(\mathfrak{S}) + \log c_{OPT}(\mathfrak{S})) + 3 \\ &\approx 1.3863 \, (c_{OPT}(\mathfrak{S}) + \log c_{OPT}(\mathfrak{S})) + 3. \end{aligned} \tag{5}$$

IV: Let $C_1(\mathfrak{S})$ be the expected cost of a tree built by inserting records requested but which are not found into the tree, in which **no** self-organization takes place when a requested record is already existing in the tree. Then

$$C_1(\mathfrak{S}) = c_{MR}(\mathfrak{S}). \tag{6}$$

V : Starting with any tree with N nodes, the average search time is greater than N/8 for the sequence of N (or N-1) requests $A_1, A_2, \ldots, A_{\lfloor N/2 \rfloor}, A_1, A_2, \ldots, A_{\lfloor N/2 \rfloor}$.

VI : Starting with a random binary search tree, after $\lceil (N \ln N)/e \rceil$ requests have been serviced under the move-to-root heuristic, where e is Euler's number, the expected search cost is greater than $c_{MR}(\mathfrak{S})$, the asymptotic cost, by at most unity.

It should be noticed here that the advantage of the move-to-root heuristic is that over a long sequence of requests, the average search time will, according to Allen and Munro, "almost certainly" be within a small constant factor of the optimal. Thus, whereas a bad static tree will remain bad, a structure that is self-organizing can emerge out of a bad initial configuration into a less expensive one. Also, since $c_{MR}(\mathfrak{S})$ is also the expected number of comparisons required to search for the current request, the variance on the number of comparisons was derived, and proved to be a small quantity. This means that the heuristic will **rarely** behave worse than expected. A detailed analysis of these and of the other results pertaining to this strategy may be found in Allen and Munro's paper [2].

II.2 The Splaying Heuristic

Sleator and Tarjan [13] developed a self-organizing tree structure, which is designed in the context of a complete set of tree operations which included insertion, deletion, access, split, and join. Their structure, called the **splay tree**, was shown to have an amortized time complexity of O(log N) for **all** of its operations.

The splaying heuristic is a rather ingenious scheme. It is a restructuring move that brings the node accessed up to the root of the tree, and also keeps the tree in symmetric order (which simply means that an in-order traversal would access each item in order from the smallest to the largest). As well, it has the interesting side effect of tending to keep the tree in a form that is nearly height-balanced, besides 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 developed by Allen and Munro, but whereas the move-to-root has an asymptotic average access time within a constant factor of optimum under the assumption that the access probabilities are independent and time invariant, the splaying operation yields identical results even if the above assumptions are relaxed.

We shall now describe the splaying operation more explicitly. To do so 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, the null pointer.

The splaying operation is defined on a record i as follows :

Case I : If P(i) is the root of the tree, then rotate the edge joining i with P(i). This is the terminating case of the recursion.

Case II : If P(i) is not the root and i and P(i) are both left or both right children, rotate the edge joining P(i) with P(P(i)) and subsequently rotate the edge joining P(i) with i. (See Figure 4a)

Case III : If P(i) is not the root and i is a left child and P(i) is a right child or vice-versa, rotate the edge joining i with P(i) and subsequently rotate the edge joining i with the new P(i). (See Figure 4b)

Obviously, if i is the root of the tree no adjustment will occur.

********** **Insert Figures 4a and 4b here** **********

These steps are performed from the bottom of the tree up towards the root until the record that was accessed is the root of the tree (Figure 5).

********** **Insert Figure 5 here** **********

Sleator and Tarjan defined the following quantities. w(i) is the **weight** of a node i, which is an **arbitrary** positive value, subjectively assigned to the nodes. s(i) is the **size** of a node i, and is the sum of the individual weights of all items in the subtree rooted at i. Similarly, r(i) is the **rank** of a node i, defined as r(i) = log s(i). Finally, q(i) is the access frequency of a node i.

Using these definitions, the following results were proven for a sequence of m accesses on an N-node splay tree.

I : The total access time is O((m+N) log (m+N)).

II : If every item is accessed at least once, then the total access time is

$$O\left(m + \sum_{i} q(i) \log\left(\frac{m}{q(i)}\right)\right)$$

III : If f is **any** fixed item in the tree, the total access time is

$$O\left(N \log N + m + \sum_{j=1}^{m} \log\left(|i_j - f| + 1\right)\right)$$

where the items are assumed to be numbered 1 through N in symmetric order, and the sequence of accessed items is $i_1, i_2, \ldots, i_m$.

IV : Suppose the accesses are numbered from 1 to m in the order in which they occur. For any time instant j at which the record accessed is A_j, let t(j) be the number of distinct items accessed before j subsequent to the last access of item A_j, or subsequent to the beginning of the sequence if j is the first access to A_j. Then the total access time is :

$$O\left(N \log N + m + \sum_{j=1}^{m} \log\left(t(j) + 1\right)\right).$$

V: The total time of a sequence of m accesses on an N-node splay tree is

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

where f is any fixed item, and t(j) is defined as in IV above.

VI. Let W be the total weight of the tree included in the access operation. For any item i in a tree T let i- and i+ denote the item preceding and the item following i, respectively, in the symmetric order, where if i- is undefined, $w(i-) = \infty$, and if i+ is undefined, $w(i+) = \infty$. Then, the upper bound on the amortized time complexities of the access operation is :

$$3 \log\left(\frac{W}{w(i)}\right) + 1 \qquad \text{if } i \in T$$

$$3 \log\left(\frac{W}{\min\{w(i-), w(i+)\}}\right) + 1 \qquad \text{if } i \notin T$$

Results I through IV are particularly significant, because as Sleator and Tarjan explain, they respectively imply that :

I : over a long enough sequence of accesses a splay tree is as efficient as any form of a uniformly balanced tree.

II : a splay tree is as efficient as any fixed search tree, including the optimum tree for the given access sequence.
III : splay trees support accesses in the vicinity of a fixed finger with the same efficiency as finger search trees.
IV : the most recently accessed items, which can be conceptually imagined to form a "working set", are the easiest to access.

Result V unifies all of the above into a single result.

II.3 The Monotonic Tree Scheme

Up to this point we have considered only binary search tree restructuring schemes that require no extra memory. The scheme which we introduce in this subsection is the only scheme in the literature which requires extra memory.

This scheme is a dynamic version of a tree structuring method suggested by Knuth [7] 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, as it turns out, behaves quite poorly. Indeed, Mehlhorn [9] shows that if $C_{MON}(\mathfrak{S})$ is the asymptotic average cost of retrieving a record under the monotonic tree scheme for the distribution $\mathfrak{S}$, then,

$$\frac{C_{MON}(\mathfrak{S})}{C_{OPT}(\mathfrak{S})} \leq \frac{N}{4 \log N}.$$

Walker and Gotlieb [14] have presented simulation results for static monotonic trees, and these results also indicate that this strategy behaves quite poorly as 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 key probabilities are not known *a priori* . Each record has one extra memory location, which contains the total number of accesses to that particular record. 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 (Figure 6). 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 in which the distribution $\mathfrak{S}$ is time invariant and independent.

********** **Insert Figure 6 here** **********

Intuitively, this is a rather appealing scheme. However, Bitner determined bounds for the cost of a monotonic tree, and showed that it is largely dependent on the entropy, H, of the probability distribution of the keys, where

$$H(\mathfrak{S}) = \sum_i -(s_i \log s_i)$$

Let h_i be the ith harmonic number, and let ζ_N be defined by

$$\zeta_N = \sum_{i=1}^{N} e^{\frac{1}{i!}}$$

Further, let f(n) be defined by

$$f(n) = 2 \ln \zeta_N + 1 - \frac{2}{\zeta_N} \sum_{i=1}^{N} e^{\frac{1}{i!}} \left[\frac{1}{i!} + h_i\right]$$

Then, the bounds derived by Bitner are :

$$(2 \ln 2)H(\mathfrak{S}) - f(n) \leq C_{MON}(\mathfrak{S}) \leq (2 \ln 2)H(\mathfrak{S}) + 1.$$

If $H(\mathfrak{S})$ is small, then the monotonic tree scheme is nearly optimal; but if $H(\mathfrak{S})$ is large, it will behave quite poorly. Bitner also stated a result, due to Bayer [4], that proved that the expected entropy of a randomly chosen probability distribution is log(N)-ln 2, which is nearly the maximum entropy attainable. He concludes from this that the monotonic tree scheme may be expected to behave poorly on the average, and indeed, our experimental results support this viewpoint.

II.4 Drawbacks of the existing rules

In spite of all their advantages, all of the schemes given in the preceding sections 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 entire tree along the access path, as the accessed element is moved up by successive steps along the path. Furthermore, on the average, these two schemes perform O(log N) operations on each access. Note that these operations are not merely computations, but rotations. Thus, performing a move-to-root or splaying operation every time a record is accessed can be **very** expensive. We would like a heuristic that could solve these two 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 strategy 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 rule, which lies in the fact that it considers only the frequency count for each record, will thus 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 (Figure 7).

********** **Insert Figure 7 here** **********

In the next section, we will introduce our new scheme that takes into account the subtrees of the rotated nodes. Our experimental evidence shows that this is a scheme which is far superior to any other self-organizing binary search tree rule that exists in the literature.

III. THE CONDITIONAL ROTATION HEURISTIC

III.1 Principles Motivating the Heuristic

The new heuristic which we introduce requires that each of the records in the binary search tree 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 weighted path length 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 weighted path length of the entire tree decreases as a result of the operation.

To ease the readability, 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. It is obvious that $\tau_i(n)$ satisfies (7) below.

$$\tau_i(n) = \sum_{j \in T_i} \alpha_j(n) \tag{7}$$

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 weighted path length of the tree T_i rooted at node i at time instant n, where,

$$\kappa_i(n) = \sum_{j \in T_i} \alpha_j(n) \cdot \lambda_j(n) \tag{8}$$

To ease the notation, where no ambiguity results, we shall omit reference to the time instant '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. The following lemma yields the recursive properties of τ_i and κ_i, 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 :

$$\text{(a)} \quad \tau_i = \alpha_i + \tau_{iL} + \tau_{iR} \tag{9}$$

$$\text{(b)} \quad \kappa_i = \alpha_i + \tau_{iL} + \tau_{iR} + \kappa_{iL} + \kappa_{iR} \tag{10}$$

Proof :

(a) Consider τ_i , which by definition is

$$\tau_i = \sum_{j \in T_i} \alpha_j ,$$

Splitting τ_i into the components contributed by the root i, its left subtree T_{iL}, and its right subtree T_{iR}, we write,

$$\tau_i = \sum_{j \in T_i} \alpha_j = \alpha_i + \sum_{j \in T_{iL}} \alpha_j + \sum_{j \in T_{iR}} \alpha_j$$

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

which is indeed the result (a).

(b) We know that by definition,

$$\kappa_i = \sum_{j \in T_i} \alpha_j \lambda_j .$$

(11)

Expanding (11) in terms of the subtrees at each level, we write*,

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

whence,

$$\kappa_i = (\alpha_i + \alpha_{iL} + \alpha_{iR} + \alpha_{(iL)L} + \alpha_{(iL)R} + \alpha_{(iR)L} + \alpha_{(iR)R} + \ldots)$$
$$+ (\alpha_{iL} + 2\alpha_{(iL)L} + 2\alpha_{(iL)R} + \ldots) + (\alpha_{iR} + 2\alpha_{(iR)L} + 2\alpha_{(iR)R} + \ldots) \tag{12}$$

---

* 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.

Using (7) and (8), (12) simplifies to

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

Whence using (9), we have

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

Hence the result. •••

This lemma means that to calculate τ and κ for any node, it is necessary **only** to look at the values stored at the node itself and at the corresponding quantities stored for the **children** of the node. Observe that the recursion does not involve the quantities in the entire tree nor the entire subtree rooted at the node.

Before we go any further into determining a method to update the α, τ, and κ values, we need to clearly define the effects that the execution of a rotation operation on node i has on the rest of the tree.

Lemma II. (Rotation properties definition)

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 the rotation has been performed, i and P(i) interchange roles i.e. i becomes the parent of P(i).

(iii) Nodes which were ancestors of node i before the rotation operation was performed remain the ancestors of node i after the rotation has been performed, with the exception of P(i).

(iv) Nodes which were not ancestors of node i before the rotation operation was performed do not become ancestors of node i after the rotation has been performed.

Proof of Results (i) and (ii):

These follow directly from the definition of the rotation operation given in the preceding section.

Proof of Result (iii):

There are two cases here. The first case involving P(i) is fairly obvious, since P(i) will clearly not be an ancestor of node i, by the definition of the operation. The second case encompasses all other ancestors of i. All of these nodes were clearly also ancestors of P(i) before the rotation operation was performed. From

part (ii) of this lemma, after the rotation, node i takes the position previously held by node P(i). Thus, it is clear that all of the previous ancestors of P(i), and hence all of the ancestors of i (except P(i)) continue to be ancestors of node i.

Proof of Result (iv):

There are two cases here also. The first is the case of nodes that are on the same level as P(i) or below before the rotation is performed. Since after the rotation, node i moves up to the level of P(i), obviously no such nodes can become an ancestor of node i. The second case is that of nodes on the levels above node P(i). Since all reorganization takes place at the levels of node P(i) and node i, by the definition of the operation, any nodes above the level of node P(i) and which are not ancestors of i will not have any changes made to the contents of their subtrees. Hence the result. •••

Now that we have derived the recursive properties of τ_i and κ_i and clearly defined the effects of the rotation operation, we shall try to 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 at i and for the case in which no rotation is performed. The following theorem states how these fields can be updated.

Theorem I.

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

(a) **Updating of** α: $\alpha_j = \alpha_j + 1$, $j = i$

$\alpha_j = \alpha_j$, $j \neq i$

(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 access path from the root to P(P(i)) are updated according to the following equation :

$$\tau_j = \tau_j + 1.$$

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

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

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 access 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 obviously, if node i is accessed, **only** the access count for node i need 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 because node i will never be contained in one of its own subtrees by virtue of the definition of the rotation operation (see Lemma II). Thus no α value will increase in any node below node i, and hence the corresponding values of τ will remain unchanged.

Case (ii) : The result is obvious when no rotation is performed. For the case when a rotation is performed, from the definition of the rotation operation, no subtree which is **not** on the access path can contain node i, even if a rotation is performed at i. For every node k, since each τ_k equals the sum of all α_j, $j \in T_k$, and since the only α-value that is changed is α_i, due to Lemma II, the value of τ_k is unchanged because for all k not on the access path to i, $i \notin T_k$. 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 Lemma II, ancestors of i up to P(P(i)) remain 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 weighted path length 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 (10). Note that since the value of the weighted path length 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. •••

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. We repeat that we refer to the accessed node and its parent, j, before the rotation operation is performed as i and P(i)=j respectively. After the rotation operation 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, and indeed, the relationship between the two nodes 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).

III.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 after an access of a specified node, both in the case that a rotation is performed and in the case that it is not performed. 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 weighted path length of the **entire tree** must decrease as a result of the operation if the rotation is to be performed. A brute-force method for determining whether or not to rotate suggests itself immediately. This method is performed 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, at the same time computing the hypothetical values of τ and κ that would be obtained had we performed the rotation operation. Once we have reached the root of the tree, we decide whether to perform the rotation operation or not by comparing the hypothetical κ-value to the actual κ-value. If the hypothetical value is the smaller of the two, 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 passes must be made between the accessed node and the root of the tree, which implies that on the average O(4 log N) operations will be required. 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 operation is performed, and hence determine if the operation should be performed or not. By applying such a method, we can reduce the average time requirements of the access operation to O(2 log N).

Before we describe our new criterion for rotation, the notation we referred to

above will be clarified. 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. As stated in the preceding subsection, if close attention is not paid, the notation may be a little 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 operation on node i it may not be the parent anymore. In such a case we still refer to that node as P(i), but the actual physical parent of node i will now 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 operation if and only if θ_i is reduced by the rotation at i. 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) When node i is the root of the tree, the proof is trivial, as P(i) in this case is the null pointer, and node i will never rotate upwards. Thus no κ-lowering rotation can ever be performed.

Case (ii) This is the case when P(i) is the root of the tree. Here minimizing θ_i is equivalent to minimizing the weighted path length of the entire tree and the result follows since the weighted path lengths considered in the decision process are actually the weighted path lengths of the entire tree.

Case (iii) This is the case that 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 $j \neq P(i)$. Then at node j, due to (10),

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

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 operation, the latter operation will not cause a change to the total number of accesses to the left subtree. Hence,

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

$$= \alpha_j + \tau_{jL} + \tau_{jR} + \kappa_{jL} + \kappa_{jR} - \alpha_j - \tau_{jL} - \tau_{jR} - \kappa_{jL}' - \kappa_{jR}$$

$$= \kappa_{jL} - \kappa_{jL}'$$

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. Clearly this inequality bubbles itself up recursively at every level of the tree and ultimately is true for the root of the entire tree. This means that a κ-lowering rotation performed anywhere in the tree will lower the weighted path length 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. •••

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 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 applying the result of Lemma I, and using the values at the nodes that **would be** the left and right children of i should the rotation be actually performed.

For the sake of completeness, we present below our access and reorganizing

algorithm which is based on these results. This algorithm is recursive, and is presented in its recursive form. The algorithm, as seen below, covers the general case for any node j, and covers all possible cases. 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.

```
Algorithm  CON_ROT_Access          { CONditional ROTation Access }
Input :    A binary search tree T and a search key k_i
Output :   (i) the restructured tree T'
           (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 κ_P(j), κ_j, τ_P(j), τ_j
              endif
           else                          { Search the subtrees }
              if k_i < k_j then
                  perform CON_ROT_Access on j^.Leftchild
              else
                  perform CON_ROT_Access on j^.Rightchild
              endif
              recalculate κ_j
           endif
           return record i
           end Method
end Algorithm CON_ROT_Access.
```

This rule is a definite improvement over the previously developed schemes because it only reorganizes itself if the weighted path length of the entire tree decreases as a result of the reorganization. This 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.

Assuming that the average path length of a binary search tree with N records is O(log N), our algorithm will require O(2 log N) time and O(3N) space, over and above that which is required for the tree. We would like to reduce both the space and time requirements for our algorithm. In the next section we introduce a modification of our method that requires only O(N) extra space and O(log N) time.

IV. THE SPACE-OPTIMIZING CONDITIONAL ROTATION HEURISTIC

Up to this point we have developed a scheme for dynamically restructuring binary search trees by considering whether the weighted path length of the entire tree decreases as a result of performing the operation, and performing any reorganization only if the weighted path length 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 would like to present a scheme which performs tree reorganization using the same heuristic that was used in the preceding section. However, we would like to reduce both the time and the space it requires. The previous heuristic demands that we must make a search down the tree, which requires O(log N) time, and also retrace our steps up the tree to update the κ values. The latter requires O(log N) time as well. If we could eliminate the need for storing and updating the κ values, we could save greatly in both time and space.

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 Lemma I 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 operation should be performed.

We define a new criterion function ψ which is dependent entirely upon the τ values stored in each node, in contrast to the previous criterion function θ, which was dependent upon the α, τ, and κ values. Previously, we performed a rotation operation if the criterion function θ_i took a positive value. Now, we perform the rotation if the new criterion function ψ_i takes a positive value. 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.

Before stating and proving these results, we will refresh our notation. Recall that α_i is the number of accesses to node i, τ_i is the total number of accesses to the subtree

rooted at node i, and κ_i is the weighted path length of the subtree rooted at node i. Also, primed quantities indicate the values of the corresponding quantities after a rotation operation has been performed.

We may now state and prove the results which allow us to reduce the space and time requirements for our tree reorganizing heuristic.

Theorem III.

Let i be the accessed node of the binary search tree, and let $\kappa_{P(i)}$ be the weighted path length of the tree rooted at the parent P(i) if no rotation is performed on node i. Let κ_i' be the weighted path length 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

$\psi_i \geq 0$ if and only if $\theta_i \geq 0$,

where,

$\theta_i = \kappa_{P(i)} - \kappa_i'$.

Proof :

It is required to prove that $\psi_i \geq 0$ if and only if $\theta_i \geq 0$. This will indeed imply that performing a rotation operation if $\psi_i \geq 0$ is equivalent to performing a rotation operation if $\theta_i \geq 0$. Actually, we show a stronger result, which is that $\theta_i \equiv \psi_i$.

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 I (See Figure 2a),

$$\kappa_{P(i)} = \alpha_{P(i)} + \tau_i + \tau_{B(i)} + \kappa_i + \kappa_{B(i)}. \qquad (13)$$

Suppose that at this point, a rotation is applied to node i in an upwards direction.

The resulting tree is that in Figure 2b, and the new expression for κ_i' is

$$\kappa_i' = \alpha_i + \tau_{iL} + \kappa_{iL} + \tau_{P(i)}' + \kappa_{P(i)}'. \qquad (14)$$

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

$$\tau_{P(i)}' = \alpha_{P(i)} + \tau_{iR} + \tau_{B(i)} \ , \qquad \text{and,} \qquad (15)$$

$$\kappa_{P(i)}' = \alpha_{P(i)} + \tau_{iR} + \tau_{B(i)} + \kappa_{iR} + \kappa_{B(i)}. \qquad (16)$$

Substituting (15) and (16) into (14), 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 :

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

Notice 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

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

and hence

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

and the theorem is proved. •••

Remarks :

(i) As stated in the preamble to this section, we can use the criterion of testing the weighted path length at the **root** of the tree to determine whether a rotation should be performed or not. This was shown in the preceding section to be equivalent to computing θ_i at the **node i** and using this as a criterion to decide whether or not to restructure the tree. In this section, we have shown that minimizing the new criterion function ψ_i is equivalent to minimizing the weighted path length of the tree at the root. Unlike θ_i, however, ψ_i only requires us to use 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 this takes an average of O(log N) time) 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.

As well, observe that at any node i,

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

Thus,

$$\alpha_i = \tau_i - \tau_{iL} - \tau_{iR}.$$

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 need to maintain only one extra memory location per node, and we are not required to make a second pass upwards on the tree. Our modified algorithm, which is a space optimizing version of CON_ROT_Access, requires O(N) extra space and O(log N) time, the latter being the time required to **access** the node i, which would anyway have to be taken in **any** binary search tree scheme.

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

Corollary I.

Algorithm CON_ROT_Access and its space optimizing version SpaceOptCON_ROT_Access are stochastically equivalent.

Note that it is not just the average performance (or the asymptotic performance) of these algorithms which is equivalent; but they both work in **synchronism** (i.e. in **lock-step**).

For the sake of completeness, the modified algorithm is included here also. As before, this algorithm is recursive and is thus presented in a recursive form for any node j. 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 only by converting the α values to their equivalents in terms of τ values as stated above.

Algorithm SpaceOptCON_ROT_Access
Input : A binary search tree T and a search key k_i.
Output : the restructured tree T', a pointer to record i containing k_i
Method :

```
τ_j ← τ_j + 1                    { update τ for the present node }
if k_i = k_j then                { Found the record in question }
  if node j is a left child then
      ψ_j ← 2τ_j - τ_jR - τ_P(j)
  else
      ψ_j ← 2τ_j - τ_jL - τ_P(j)
  endif
  if ψ_j > 0 then
      rotate node j upwards
      recalculate τ_j, τ_P(j)
  endif
  return record j
else
  if k_i < k_j then              { Search the subtrees }
      perform SpaceOptCON_ROT_Access on j^.Leftchild
  else
      perform SpaceOptCON_ROT_Access on j^.Rightchild
  endif
endif
end Method
end Algorithm SpaceOptCON_ROT_Access
```

Thus far we have discussed only the effect of these rotations on the weighted path length of the tree. We would like to define the effect of such restructuring on the **cost** of the entire tree, as defined in equation (2).

Theorem IV

Any algorithm which reduces the weighted path length of a binary search tree asymptotically reduces the cost of the entire tree.

Proof:

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

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

But by the law of large numbers, s_j can be estimated by the ratio

$$\frac{\alpha_j(n)}{\sum_j \alpha_j(n)}.$$

But $\Sigma\alpha_j(n)$ is identically equal to $\tau(n)$ which is the sum of the α's of the node in the tree. Thus, 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 $\kappa_T(n)$, the weighted path length of the entire tree, also asymptotically causes C_T to decrease as well. Hence the result. •••

In particular, the algorithms CON_ROT_Access and the space optimizing version SpaceOptCON_ROT_Access achieve the above result, and thus we believe that our method is superior to all the other dynamic schemes presented in the literature, as is demonstrated by our experimental results. Our belief is based on the fact that none of the other schemes that exist consider dynamically reducing the weighted path length of the tree, while our scheme is based entirely on this concept.

We now proceed to present some experimental results which compare our scheme to the various static and dynamic tree structures.

V. EXPERIMENTAL RESULTS

A series of experiments was run to compare the asymptotic properties of the various binary search tree structures. The following tree schemes were used in the simulations :

(i) Allen and Munro's move-to-root scheme [2],

(ii) Sleator and Tarjan's splay tree scheme [13],

(iii) the optimal tree, constructed using Knuth's algorithm [7],

(iv) a height-balanced tree [1],

(v) the monotonic tree rule described by Bitner [5],
(vi) the nearly optimal tree rule due to Walker and Gotlieb [14], and
(vii) CON_ROT, the scheme we have presented in this paper.

Since both of our algorithms are stochastically equivalent, the results we present are results of simulations using the first version of the algorithm, which indeed worked in lock-step with the second, which is the space optimized version of the first.

In each simulation, there were 100 parallel experiments run, and each initial tree contained 15 records. The initial tree structure of each experiment was randomly chosen. 30,000 accesses were performed on each tree, each accessed item being randomly chosen, the randomness being specified by a probability distribution vector. Four different types of distributions were used :

(i) Zipf's distribution. The individual probabilities obey the following :

$$s_i = k_1 / i,$$

$$\text{where } k_1 = \frac{1}{\sum_{i=1}^{N} \frac{1}{i}}.$$

(ii) Exponential distribution. The individual probabilities obey :

$$s_i = k_2 / 2^i,$$

$$\text{where } k_2 = \frac{1}{\sum_{i=1}^{N} \frac{1}{2^i}}.$$

(iii) and (iv) Two types of wedge distributions. In both cases, the following equation is satisfied for the individual probabilities.

$$s_i = k_3 \cdot (N - i + 1),$$

$$\text{where } k_3 = \frac{1}{\sum_{i=1}^{N} i}$$

The difference between the latter two distributions is that the first had the probabilities chosen in such a way that the items, when listed in order from the highest access probability to the smallest, were also in lexicographical order from the smallest item to the largest. The second distribution was chosen such that the

items had no relationship between their probability mass orderings and their lexicographical orderings.

Table I contains the estimated asymptotic average cost of retrieval of a record for each of the static and dynamic binary search tree access schemes. This estimate is reported below for each scheme under each of the probability distributions used in the simulations. To minimize the variance on the final results, the estimates given were obtained by obtaining the time average of the ensemble averages across the last 500 iterations of the simulations.

Scheme	Wedge distribution 1	Zipf's law distribution	Exponential distribution	Wedge distribution 2
Move-to-root	3.8075999994	3.5268199995	2.2078799997	3.8093599994
Splaying	3.8713399994	3.6161599994	2.2352599996	3.8736999994
Optimal	3.09765999995	2.9275999995	1.9933799997	3.0965599995
Balanced *	3.2677799995	3.4306199995	3.5974199994	3.2678599995
Monotonic	5.6610999991	4.5317999993	2.0016599997	5.6884799991
Nearly optimal *	3.9592199994	3.3975199995	1.9983399997	3.9671799994
CON_ROT_ Access	3.1957599995	3.0331199995	2.0123999997	3.2023999995

* - static tree schemes

Table I : Performance figures of the various tree schemes. The figure given for each scheme is the estimated expected cost of retrieving a record.

Table II contains the percentage differences between the estimated asymptotic average cost of the static optimal tree scheme and the estimated asymptotic average costs of each of the static and dynamic tree access and restructuring schemes for each of the probability distributions.

Scheme	Wedge distribution 1	Zipf's Law distribution	Exponential distribution	Wedge distribution 2
Move-to-root	22.918591	20.467961	10.760618	23.019092
Splaying	24.976273	23.519607	12.134165	25.096882
Balanced*	5.491888	17.181993	80.468354	5.531945
Monotonic	82.754082	54.795739	0.415370	83.703206
Nearly optimal*	27.813252	16.051373	0.248824	28.115716
CON_ROT_ Access	3.166907	3.604318	0.954158	3.417986

* - static schemes

Table II : Percentage difference between the optimal tree and the other schemes.

It may be seen quite clearly from the results presented above that our new scheme, CON_ROT_Access, outperforms all of the other static and dynamic binary search tree schemes, excepting the static optimal tree itself. The results obtained for the Zipf's law distribution are typical. In this case, 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. (This, it should be noted, is not typical of the best performance of this algorithm. Walker and Gotlieb described parameters which could be chosen to yield the best possible tree, and by fine-tuning these parameters, it is entirely possible that a better performance could be obtained. However, this scheme is still one that requires *a priori* knowledge of the access probabilities of the nodes, and thus does not address the problem that we examined.) Our scheme, CON_ROT_Access, proved to have 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 in the simulations. Of the dynamic schemes there is one case where a competitive scheme performed better than our technique. Amazingly enough, the competitor is the monotonic tree. However, this is not so astounding, when one considers that the probability distribution in this case is the exponential distribution, and it can be easily shown [14] that for the exponential distribution the monotonic tree is the optimal tree. However, except for this "pathological" case, our scheme far outperforms all of the other dynamic tree reorganizing schemes.

VI. CONCLUSIONS AND OPEN PROBLEMS

In this paper we have introduced a new self-organizing tree strategy requiring extra memory which attempts to reorganize a binary search tree to an optimal form. It required 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 weighted path length 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 weighted path length of the entire **tree** (and not just the subtree rooted at the node) decreases as a result of the rotation operation. We have shown that this implies that the cost of the entire tree asymptotically decreases as a result. But to do this, we anticipate at the level of the rotation whether a rotation will yield this result, and decide accordingly whether to restructure the tree or not. As well, we have presented a space optimizing version of this algorithm which requires us to maintain only one extra memory location per node. We have shown that this space optimizing version of the algorithm and the former version are stochastically equivalent. Both of these rules require O(N) extra memory and O(log N) time.

We have also performed a series of extensive simulations, testing the various schemes presented in the literature. Each scheme was simulated for a variety of probability distributions, and the estimated asymptotic expected cost of retrieval of a record was found for each method. Simulation results demonstrate that overall, our new scheme has an asymptotic expected cost of retrieval which is within a small percentage of the asymptotic expected cost of the **static** optimal tree. Furthermore, it is superior to all other self-organizing tree schemes in the literature.

Among the open problems that still exist are the analysis of the stochastic performance of this algorithm and its behaviour under various distributions compared with the optimal performance. The rate of convergence of the scheme is also unknown. Other problems which are yet to be considered are the application of this strategy to heaps and other two-dimensional data structures. A variation of this scheme that requires sub-linear memory is currently being investigated.

REFERENCES

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

[2] Allen, B. and Munro, I., "Self-organizing binary search trees", J.ACM 25(1978), pp.526-535.

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

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

[5] Bitner, J. R., "Heuristics that dynamically organize data structures", SIAM J.Comput., 8(1979), pp.82-110.

[6] Gonnet, G. H., Munro, J. I. and Suwanda, H., "Exegesis of self-organizing linear search", SIAM J.Comput., 10(1981), pp.613-637.

[7] Knuth, D. E., The Art of Computer Programming, vol. 3, Addison-Wesley, Reading, Ma., 1973.

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

[9] Mehlhorn, K., "Nearly optimal binary search trees", Acta Informatica, 5(1975), pp.287-295.

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

[11] Oommen, B. J., Hansen, E. R. and Munro, J. I., "Deterministic Move-to-Rear List Organizing Strategies with Optimal and Expedient Properties", Proc. of the Twenty-Fifth Allerton Conference, Sept. 1987.

[12] Rivest, R. L., "On self-organizing sequential search heuristics", Comm.ACM, 19(1976), pp.63-67.

[13] Sleator, D. D. and Tarjan, R. E., "Self-adjusting binary search trees", J.ACM, 32(1985), pp.652-686.

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

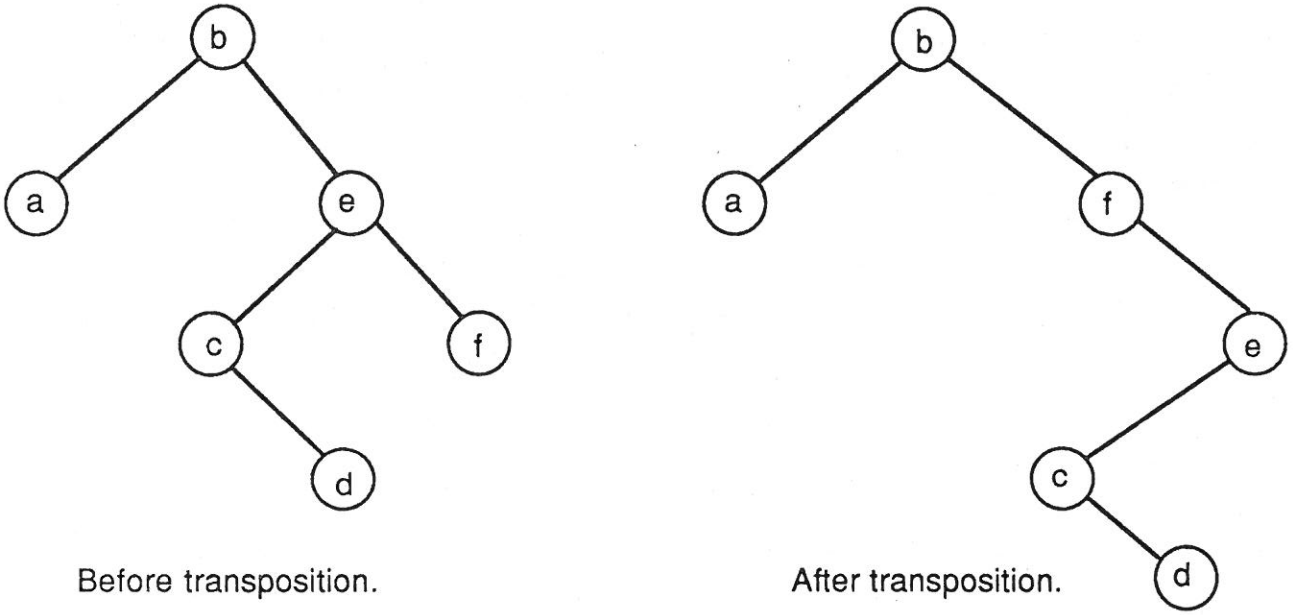


Figure 1 : An example demonstrating where the direct application of a list organizing rule would fail. The data fields of the nodes are the characters {a, b, c, d, e, f}. The transposition rule is here applied to the record whose data field is f, and the tree's binary search property is destroyed by the application of the transposition rule to the list represented by the access path.

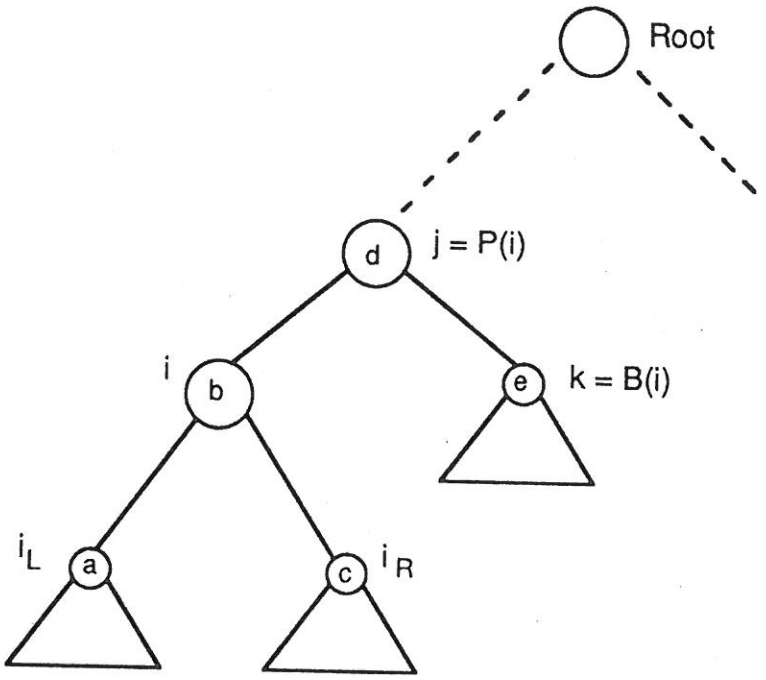


Figure 2a : The tree before a rotation is performed. The contents of the nodes are their data values, in this case the characters {a, b, c, d, e}.

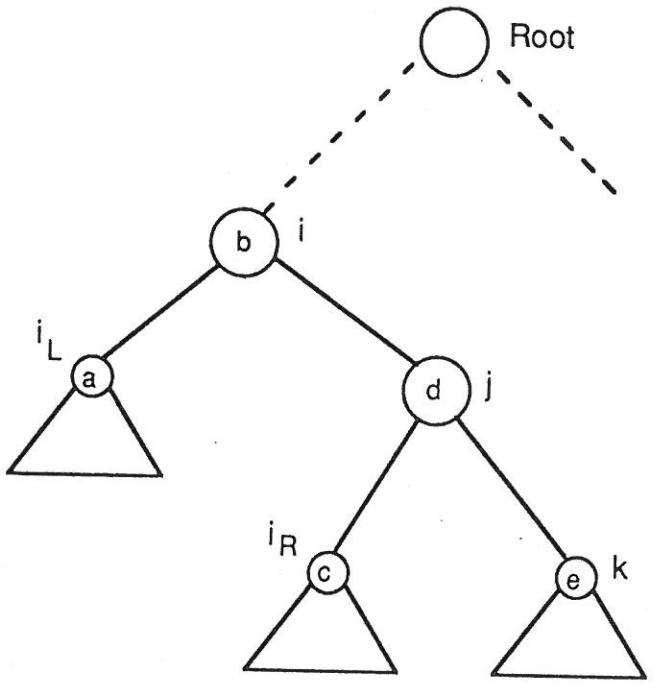


Figure 2b : The tree after a rotation is performed on node i. Observe the properties stated in Lemma II.

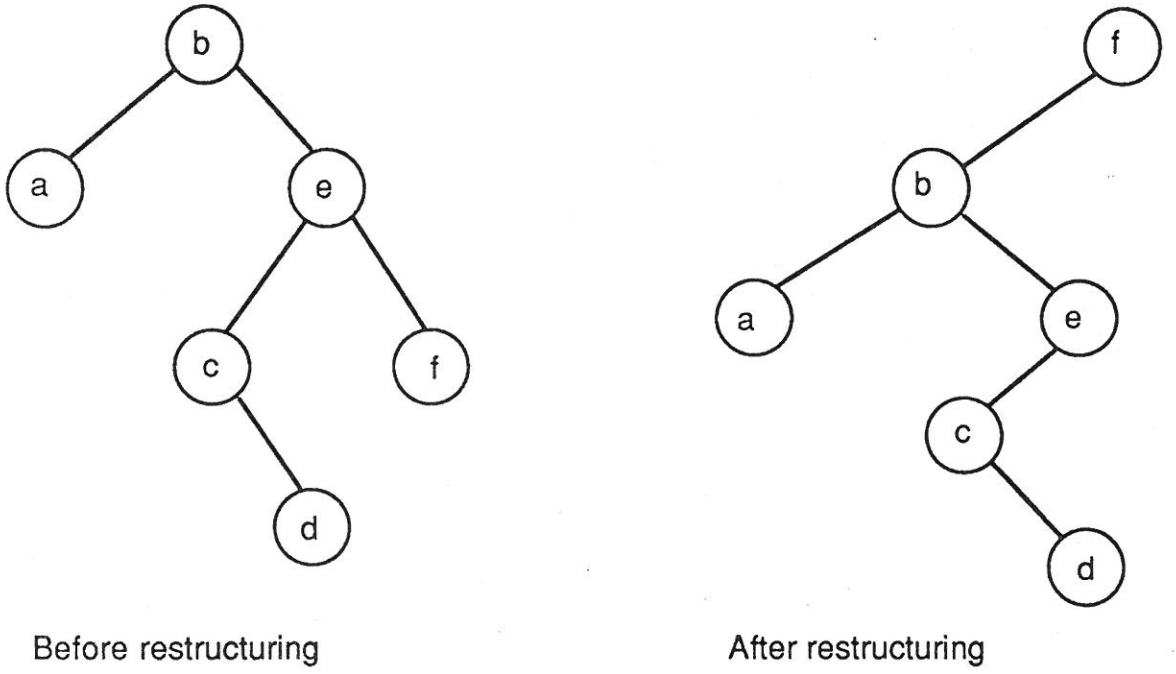
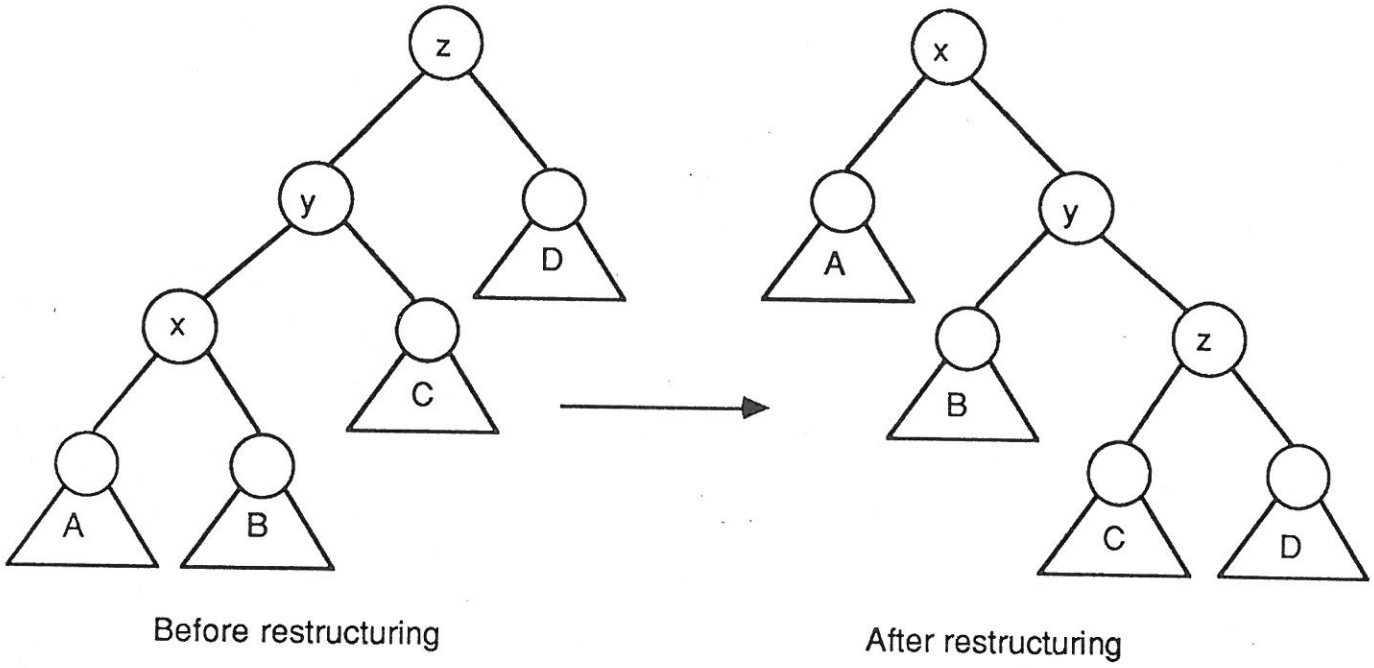


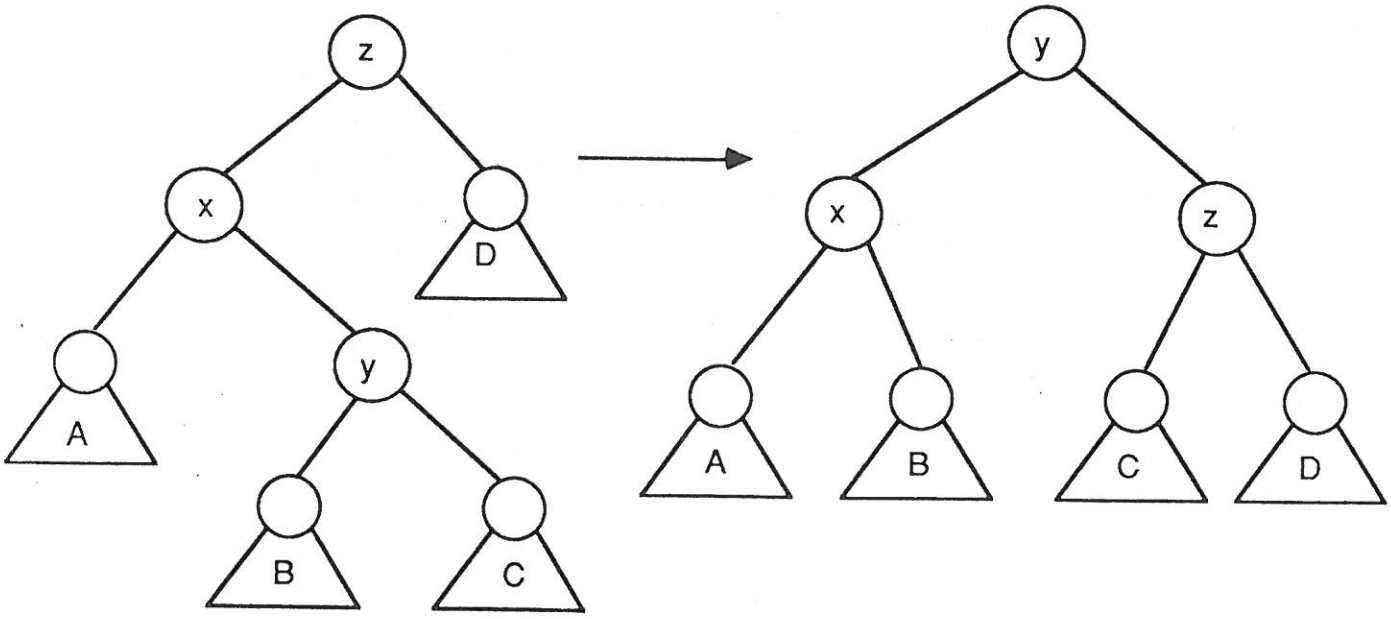
Figure 3 : Restructuring of a binary search tree after an access of item f, using the move-to-root heuristic. As in Figure 2, the data fields of the nodes are the characters {a, b, c, d, e, f}.



Before restructuring

After restructuring

Figure 4a : Case II of the splaying heuristic. Node x is accessed, and both x and y = p(x) are left children. The symmetric case (both are right children) is similar. The data elements of the nodes are the characters {x, y, z}. A, B, C, and D are used to denote subtrees. Note that the trees here may be subtrees themselves.



Before restructuring

After restructuring

Figure 4b : Case III of the splaying heuristic. Node y is accessed, and y and x = p(y) are "opposite-direction" children. The symmetric case to the one shown is done similarly. The data elements of the nodes are the characters {x, y, z}.

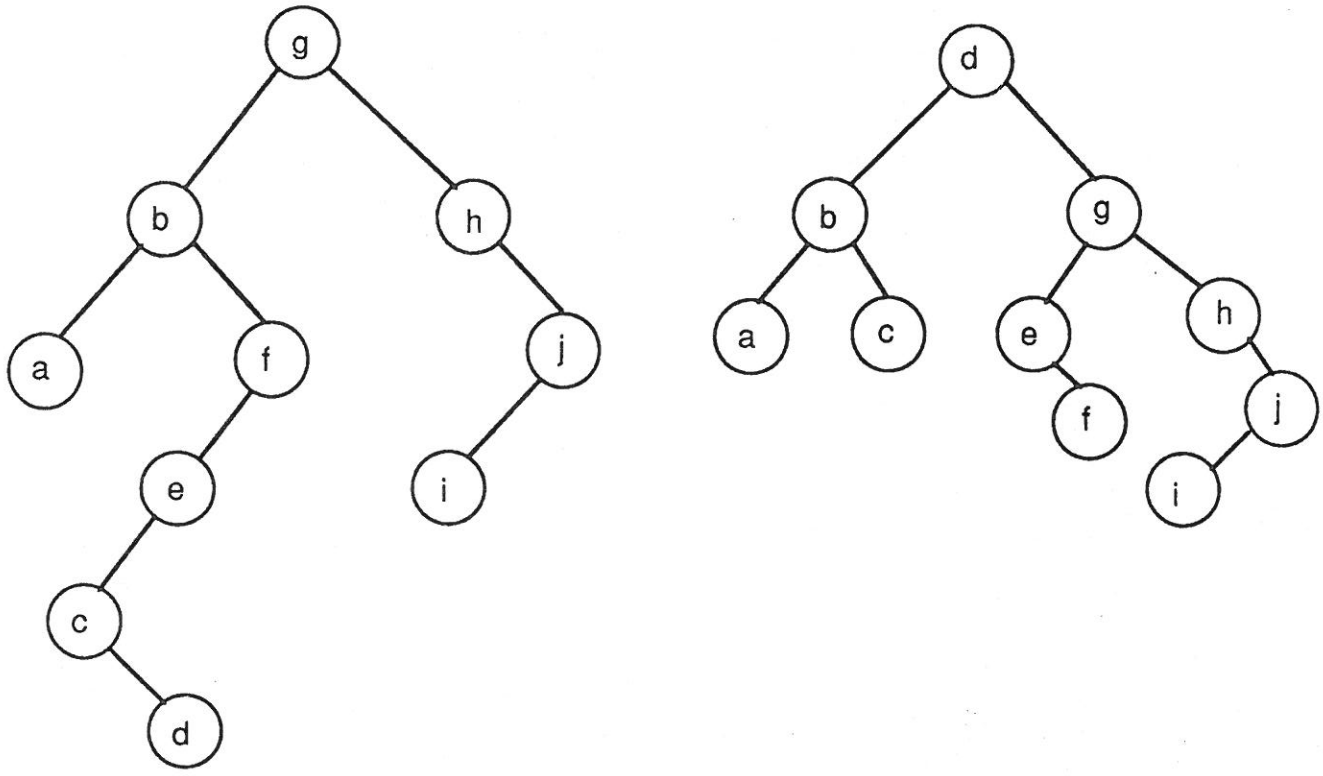
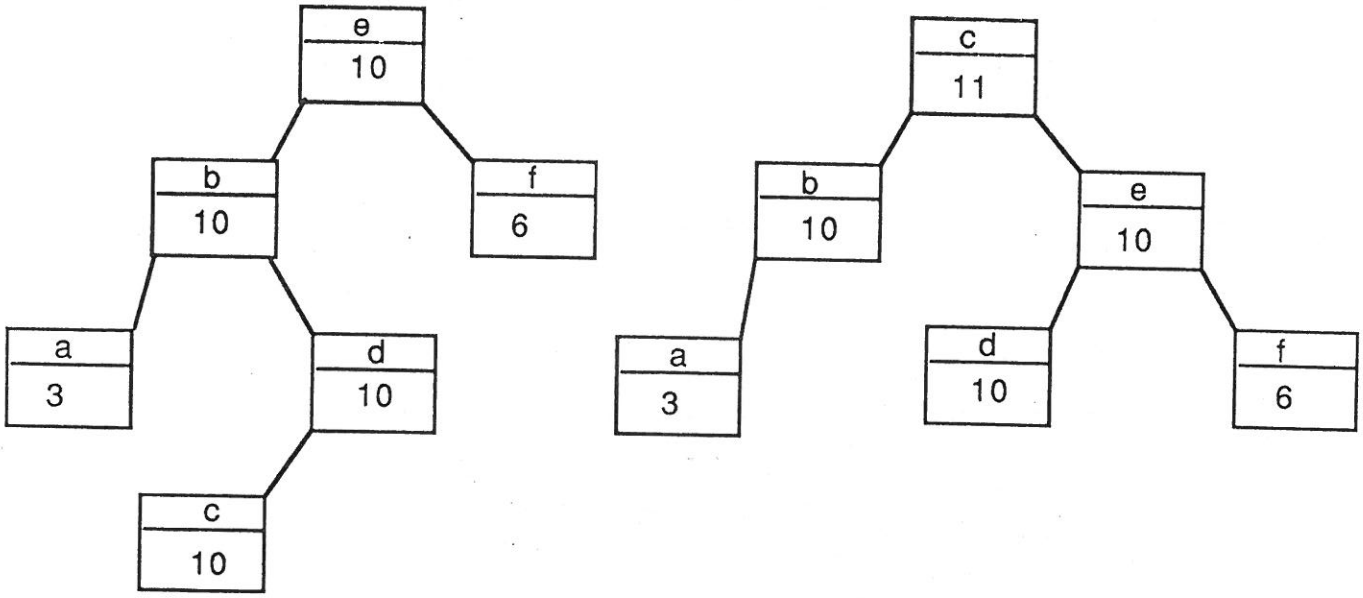


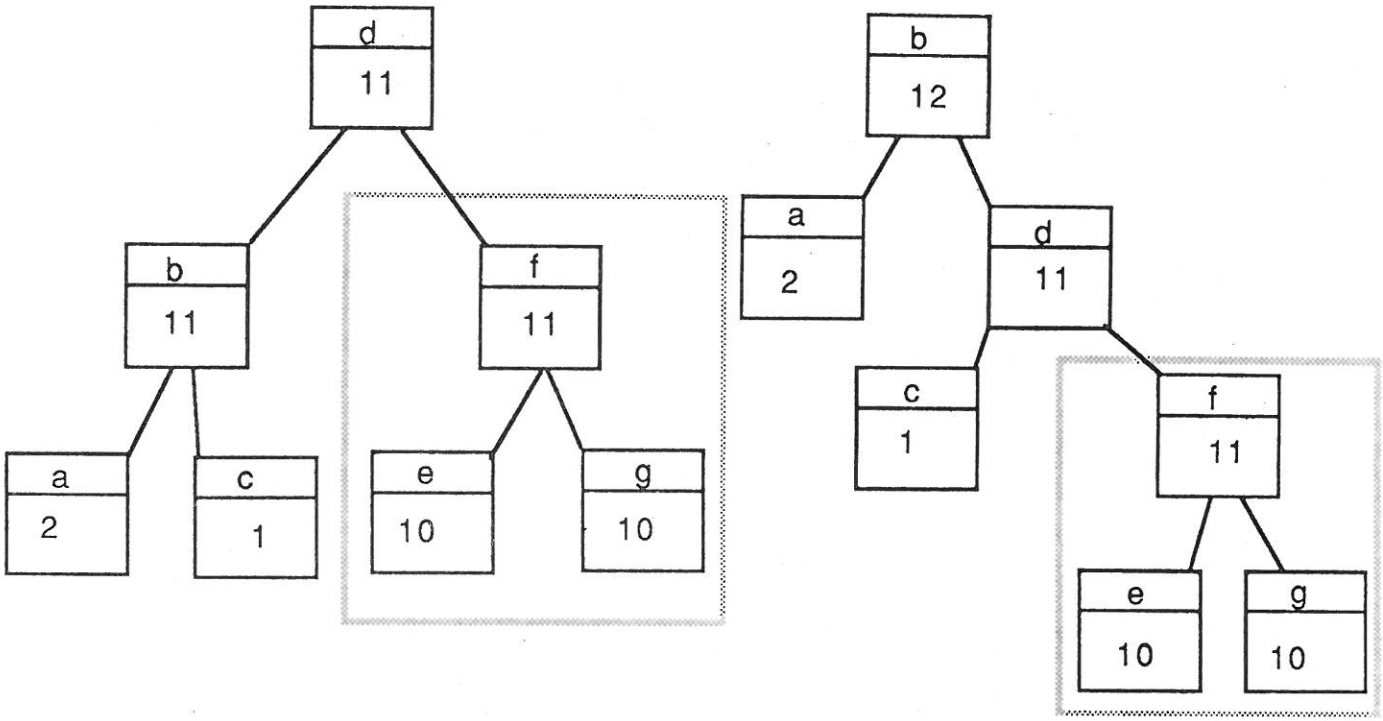
Figure 5 : An example of how a splay tree is restructured. The data fields of the nodes are the characters {a, b, . . ., h}, and the accessed element is d.



Before restructuring

After restructuring

Figure 6 : An example of restructuring using the monotonic tree rule. Each node has a data field, which in this case is the set of characters {a,...,z}. Also, each node has a counter signifying how many times it has been accessed. In this case, item c is accessed.



Before restructuring

After restructuring

Figure 7 : A case when a heavily accessed subtree (outlined) may actually be moved further away from the root of the tree by using the monotonic tree heuristic.The data and counter fields are as described in Figure 6. In this example, node b is accessed.

Carleton University, School of Computer Science
Bibliography of Technical Reports
Publications List (1985 -->)

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
KIS 5B6

SCS-TR-66 **On the Futility of Arbitrarily Increasing Memory Capabilities of Stochastic Learning Automata**
_____ B.J. Oommen, October 1984. Revised May 1985.

SCS-TR-67 **Heaps in Heaps**
_____ T. Strothotte, J.-R. Sack, November 1984. Revised April 1985.

SCS-TR-68 **Partial Orders and Comparison Problems**
out-of-print M.D. Atkinson, November 1984. See Congressus Numerantium 47 ('86), 77-88

SCS-TR-69 **On the Expected Communication Complexity of Distributed Selection**
_____ N. Santoro, J.B. Sidney, S.J. Sidney, February 1985.

SCS-TR-70 **Features of Fifth Generation Languages: A Panoramic View**
_____ Wilf R. LaLonde, John R. Pugh, March 1985.

SCS-TR-71 **Actra: The Design of an Industrial Fifth Generation Smalltalk System**
_____ David A. Thomas, Wilf R. LaLonde, April 1985.

SCS-TR-72 **Minmaxheaps, Orderstatisticstrees and their Application to the Coursemarks Problem**
_____ M.D. Atkinson, J.-R. Sack, N. Santoro, T. Strothotte, March 1985.

SCS-TR-73 **Designing Communities of Data Types**
Wilf R. LaLonde, May 1985.
Replaced by SCS-TR-108

SCS-TR-74 **Absorbing and Ergodic Discretized Two Action Learning Automata**
out-of-print B. John Oommen, May 1985. See IEEE Trans. on Systems, Man and Cybernetics, March/April 1986, pp. 282-293.

SCS-TR-75 **Optimal Parallel Merging Without Memory Conflicts**
_____ Selim Akl and Nicola Santoro, May 1985

SCS-TR-76 **List Organizing Strategies Using Stochastic Move-to-Front and Stochastic Move-to-Rear Operations**
_____ B. John Oommen, May 1985.

SCS-TR-77 **Linearizing the Directory Growth in Order Preserving Extendible Hashing**
_____ E.J. Otoo, July 1985.

SCS-TR-78 **Improving Semijoin Evaluation in Distributed Query Processing**
_____ E.J. Otoo, N. Santoro, D. Rotem, July 1985.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-79 _____
On the Problem of Translating an Elliptic Object Through a Workspace of Elliptic Obstacles
B.J. Oommen, I. Reichstein, July 1985.

SCS-TR-80 _____
Smalltalk - Discovering the System
W. LaLonde, J. Pugh, D. Thomas, October l985.

SCS-TR-81 _____
A Learning Automation Solution to the Stochastic Minimum Spanning Circle Problem
B.J. Oommen, October 1985.

SCS-TR-82 _____
Separability of Sets of Polygons
Frank Dehne, Jörg-R. Sack, October 1985.

SCS-TR-83 out-of-print
Extensions of Partial Orders of Bounded Width
M.D. Atkinson and H.W. Chang, November 1985. See Congressus Numerantium, Vol. 52 (May 1986), pp. 21-35.

SCS-TR-84 _____
Deterministic Learning Automata Solutions to the Object Partitioning Problem
B. John Oommen, D.C.Y. Ma, November 1985

SCS-TR-85 out-of-print
Selecting Subsets of the Correct Density
M.D. Atkinson, December 1985. To appear in Congressus Numerantium, Proceedings of the 1986 South-Eastern conference on Graph theory, combinatorics and Computing.

SCS-TR-86 _____
Robot Navigation in Unknown Terrains Using Learned Visibility Graphs. Part I: The Disjoint Convex Obstacles Case
B. J. Oommen, S.S. Iyengar, S.V.N. Rao, R.L. Kashyap, February 1986

SCS-TR-87 _____
Breaking Symmetry in Synchronous Networks
Greg N. Frederickson, Nicola Santoro, April 1986

SCS-TR-88 _____
Data Structures and Data Types: An Object-Oriented Approach
John R. Pugh, Wilf R. LaLonde and David A. Thomas, April 1986

SCS-TR-89 _____
Ergodic Learning Automata Capable of Incorporating Apriori Information
B. J. Oommen, May 1986

SCS-TR-90 _____
Iterative Decomposition of Digital Systems and its Applications
Vaclav Dvorak, May 1986.

SCS-TR-91 _____
Actors in a Smalltalk Multiprocessor: A Case for Limited Parallelism
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-92 _____
ACTRA - A Multitasking/Multiprocessing Smalltalk
David A. Thomas, Wilf R. LaLonde, and John R. Pugh, May 1986

SCS-TR-93 _____
Why Exemplars are Better Than Classes
Wilf R. LaLonde, May 1986

SCS-TR-94 _____
An Exemplar Based Smalltalk
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-95 _____
Recognition of Noisy Subsequences Using Constrained Edit Distances
B. John Oommen, June 1986

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-96 ——— **Guessing Games and Distributed Computations in Synchronous Networks**
J. van Leeuwen, N. Santoro, J. Urrutia and S. Zaks, June 1986.

SCS-TR-97 ——— **Bit vs. Time Tradeoffs for Distributed Elections in Synchronous Rings**
M. Overmars and N. Santoro, June 1986.

SCS-TR-98 ——— **Reduction Techniques for Distributed Selection**
N. Santoro and E. Suen, June 1986.

SCS-TR-99 ——— **A Note on Lower Bounds for Min-Max Heaps**
A. Hasham and J.-R. Sack, June 1986.

SCS-TR-100 ——— **Sums of Lexicographically Ordered Sets**
M.D. Atkinson, A. Negro, and N. Santoro, May 1987.

SCS-TR-102 ——— **Computing on a Systolic Screen: Hulls, Contours, and Applications**
F. Dehne, J.-R. Sack and N. Santoro, October 1986.

SCS-TR-103 ——— **Stochastic Automata Solutions to the Object Partitioning Problem**
B.J. Oommen and D.C.Y. Ma, November 1986.

SCS-TR-104 ——— **Parallel Computational Geometry and Clustering Methods**
F. Dehne, December 1986.

SCS-TR-105 ——— **On Adding *Constraint Accumulation* to Prolog**
Wilf R. LaLonde, January 1987.

SCS-TR-107 ——— **On the Problem of Multiple Mobile Robots Cluttering a Workspace**
B. J. Oommen and I. Reichstein, January 1987.

SCS-TR-108 ——— **Designing Families of Data Types Using Exemplars**
Wilf R. LaLonde, February 1987.

SCS-TR-109 ——— **From Rings to Complete Graphs - Θ(n logn) to Θ(n) Distributed Leader Election**
Hagit Attiya, Nicola Santoro and Shmuel Zaks, March 1987.

SCS-TR-110 ——— **A Transputer Based Adaptable Pipeline**
Anirban Basu, March 1987.

SCS-TR-111 ——— **Impact of Prediction Accuracy on the Performance of a Pipeline Computer**
Anirban Basu, March 1987.

SCS-TR-112 ——— **ε-Optimal Discretized Linear Reward-Penalty Learning Automata**
B.J. Oommen and J.P.R. Christensen, May 1987.

SCS-TR-113 ——— **Angle Orders, Regular n-gon Orders and the Crossing Number of a Partial Order**
N. Santoro and J. Urrutia, June 1987.

SCS-TR-115 ——— **Time is Not a Healer: Impossibility of Distributed Agreement in Synchronous Systems with Random Omissions**
N. Santoro, June 1987.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-116 **A Practical Algorithm for Boolean Matrix Multiplication**
M.D. Atkinson and N. Santoro, June 1987.

SCS-TR-117 **Recognizing Polygons, or How to Spy**
James A. Dean, Andrzej Lingas and Jörg-R. Sack, August 1987.

SCS-TR-118 **Stochastic Rendezvous Network Performance - Fast, First-Order Approximations**
J.E. Neilson, C.M. Woodside, J.W. Miernik, D.C. Petriu, August 1987.

SCS-TR-120 **Searching on Alphanumeric Keys Using Local Balanced Trie Hashing**
E.J. Otoo, August 1987.

SCS-TR-121 **An O($\sqrt{n}$) Algorithm for the ECDF Searching Problem for Arbitrary Dimensions on a Mesh-of-Processors**
Frank Dehne and Ivan Stojmenovic, October 1987.

SCS-TR-122 **An Optimal Algorithm for Computing the Voronoi Diagram on a Cone**
Frank Dehne and Rolf Klein, November 1987.

SCS-TR-123 **Solving Visibility and Separability Problems on a Mesh-of-Processors**
Frank Dehne, November 1987.

SCS-TR-124 **Deterministic Optimal and Expedient Move-to-Rear List Organizing Strategies**
B.J. Oommen, E.R. Hansen and J.I. Munro, October 1987.

SCS-TR-125 **Trajectory Planning of Robot Manipulators in Noisy Workspaces Using Stochastic Automata**
B.J. Oommen, S. Sitharam Iyengar and Nicte Andrade, October 1987.

SCS-TR-126 **Adaptive Structuring of Binary Search Trees Using Conditional Rotations**
R.P. Cheetham, B.J. Oommen and D.T.H. Ng, October 1987.