An Empirical Evaluation of Algorithms for Dynamically Maintaining Binary Search Trees

William E. Wright

Department of Computer Science
Southern Illinois University at Carbondale
Carbondale, Illinois

ABSTRACT

Algorithms for dynamically maintaining and utilizing binary search trees are empirically compared and evaluated. The evaluation is based on the performance of the algorithms using simulated search requests. Search keys are generated using weights which are unknown and in general unequal. The algorithms provide for inserting new nodes, searching for existing nodes, and in some cases dynamically modifying the tree in an attempt to reduce its weighted path length or search time. Included in the evaluation are algorithms for height-balanced trees, weight-balanced trees, and trees of bounded balance, as well as some combination algorithms. Also included are a basic search algorithm which performs no rebalancing, and an optimizing algorithm. In addition to the standard data, unweighted search keys, specially weighted search keys, and partially ordered key sequences are also considered. The evaluation is based primarily on the execution times of the algorithms, although weighted path lengths are also given. A combination algorithm gives the fastest speeds, although the basic search algorithm is shown to be the best for most purposes.

KEYWORDS AND PHRASES: binary search trees, balanced trees, dynamic trees, weighted trees, AVL trees, BB trees, optimal trees, weighted path length.

CR CATEGORIES: 3.74, 3.72, 4.34.

I. Introduction

A binary search tree can be a very efficient structure for maintaining an ordered table, requiring a relatively small amount of data movement during insertions, a small amount of searching during accesses, and a small amount of time during sequential processing. Nievergelt [18] has compared this structure with sequential linear lists, linked linear lists, and scatter tables, and shown that the binary search tree is generally optimal (for large n) for applications requiring good performance in all three areas.

One measure of the time required to search a tree is its weighted path length or WPL, which is basically the sum over all nodes in the tree, of the probability (weight) of the node times the number of nodes in the path from the root to the given node. A more precise definition is given in Nievergelt [18]. For trees containing n nodes which are all equally likely, Hibbard [8] has shown that WPL is approximately $\log_2 n$ for large n, assuming the trees are balanced, or $1.386 \log_2 n$ if they are not necessarily balanced but instead randomly formed. An algorithm for searching and updating such trees is presented in Knuth [12], and will be referred to here as the Basic Algorithm or Algorithm B.

Knuth [14] has also presented an algorithm for finding the optimum binary search tree (i.e., minimizing WPL) assuming general probabilities of occurences of the keys, and general probabilities for searches that end between two adjacent nodes. His algorithm requires $O(n^2)$ time and space. Hu and Tucker [9] had earlier developed and validated an optimizing algorithm requiring O(n) space and O(n log n) time, but for the special case in which only the "between-nodes" weights were positive.

There are two significant properties of Knuth's optimizing algorithm, apart from time and space requirements, which make it unsuitable for many applications. First, it operates on the entire tree and is hence inefficient for maintaining trees in the presence of dynamic change. Second, it requires known probabilities for the nodes of the tree, plus the between-nodes probabilities, thus making it inapplicable for those cases in which the search key references are unpredictable. Nievergelt [18] makes an interest-

ing categorization of tree maintenance algorithms into a binary tree structure, based on the presence or absence of these two properties plus a third property concerning single-level versus two-level storage. Baer and Schwab [4] also categorize algorithms in the form of a tree structure.

The purpose of this paper is to compare the performances of algorithms for dynamically maintaining binary search trees, with no prior knowledge of search key probabilities. The emphasis is on randomly weighted search keys, although results are also given for unweighted keys and some special kinds of weighting. The central question to be answered is which tree maintenance algorithm is the best for this situation. Performance is evaluated initially in terms of WPL, but ultimately in terms of execution time.

The case of static trees utilizing known weights has been considered by Bruno and Coffman [6], Nievergelt and Wong [20,21], Mehlhorn [15, 16], Walker and Gottlieb [22], and Weiner [23], in addition to Knuth and Hu and Tucker as mentioned previously. The case of dynamic, unweighted trees (i.e., weights assumed to be equal) has been considered by Baer [3], Baer and Schwab [4], and Nievergelt and Reingold [19]. The case of dynamic, weighted trees has been considered by Baer [3], Allan and Munro [2], and Mehlhorn [17]. Baer restricted his attention to WPL. Baer and Schwab compared the performance of several algorithms with regard to execution time, but only for non-weighted trees. Karlton et. al. [11] have studied both theoretically and empirically the properties of a certain class of algorithms called height-balancing algorithms. These latter two papers will be discussed further in Section IX.

II. Algorithms Studied

The following algorithms and corresponding tree structures are compared and evaluated in this study:

a. Algorithm B.

b. Algorithm AVL for maintaining AVL trees, as described by Adel'son-Vel'skii and Landis [1]. Briefly, each node in an AVL tree satisfies the property that the height of its left subtree and the height of its right subtree differ by at most 1.

c. Algorithm BB for maintaining trees of bounded balance, as described by Nievergelt and Reingold [19], with parameter $\alpha = 1 - \sqrt{2}/2$. Briefly, each node in a BB tree satisfies the property that the number of nodes in the subtree for which it is the root, divided into the number of nodes in its left subtree, lies between α and $1 - \alpha$.

d. Algorithm W for maintaining trees with an approximately descending weight, as described in this paper.

e. Algorithm A for maintaining trees with an approximately descending "access", as described in this paper.

f. Algorithm R as described by Allan and Munro [2], which simply rotates a node (other than the root) up in the tree each time it is referenced.

g. Algorithm AB, a combination of Algorithms A and B, described later.

h. Algorithm AVLB, a combination of Algorithms AVL and B, described later.

i. Algorithm O, Knuth's optimizing algorithm.

The comparison is based on the empirical observation of the performance of the algorithms on a given sequence of key searches. Each input key causes a normal tree search, and an insertion if the key is not in the tree. Deletions are not considered in this paper. Each algorithm in addition carries out its own logic for rebalancing the tree if necessary. Both WPL and total processing time are recorded. Processing time includes only the time for the search algorithm itself, i.e., the time for searching and maintaining the tree. It excludes the time for generating the search keys, calling the algorithm, computing averages, printing results, etc. It of course reflects any reduction in WPL made by the algorithm, and any overhead time spent in carrying out the algorithm.

III. Nature of the Sample Data

As noted earlier, this study is concerned with those applications in which there is no prior knowledge of search key probabilities (dynamic trees), and in which there is no tacit assumption that the probabilities are equal (weighted trees). This application is considered to be very realistic for many situations.

Each algorithm was tested over several trials. For each trial there was specified a certain number of distinct keys (NDK) and a certain number of key references (NKR). Raw weights were generated for each distinct key from the Uniform [0,1] distribution, and then normalized to probabilities. These probabilies were then used to randomly generate key references. Twenty sets of weights were generated for each trial, and the results appeared to be sufficiently smooth to be quite reliable. They were also very consistent with several simulations in which 100 sample sequences were run for each trial.

Four variations to this generating procedure were also considered. One variation was to generate the raw weights exponentially instead of uniformly. This change did not significantly alter the relative performances of the algorithms, and the results are not included here.

A second variation was to generate equal weights, so that all distinct keys were equally likely to be referenced. This distribution is the one assumed implicitely for nonweighted trees, and will be referred to here as unweighted data.

A third variation was to generate weights according to Zipf's Law [13], which has been shown to be a reasonable approximation for words in a natural language. This distribution uses the

probabilities $\frac{c}{1}$, $\frac{c}{2}$, ..., $\frac{c}{n}$, where c is the normalizing constant $1/(\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n}) = 1/H_n$. The probabilities were assigned to the search keys in a random fashion. This data will be referred to as Zipf's Law data.

A final modification was to sort the generated key accesses from smallest to largest, thus yielding an ascending sequence of keys. Then the sequence was partially randomized by exchanging a certain number of randomly selected pairs of keys references. The number of exchanges was specified as a fraction of the total number of key references, e.g., 20%. This sample data will be referred to as partially ordered.

IV. Algorithms W and A

Algorithm W is based on the rather obvious principle of moving a node up in the tree if its weight becomes greater than its father's weight. This move can be performed by a single rotation as described by Baer [3]. The algorithm insures that nodes with higher weights will appear above nodes with lower weight (in the limit), but it does not consider the balance of the tree. This algorithm is a dynamic extension of a criterion referred to as "Rule 1" by Mehlhorn [15].

Algorithm A is based on a more sophisticated criterion utilizing a field for each node called its "access". The access of a node is the number of times it has appeared in a search path, i.e., the number of times it has been referenced plus the number of times it has been passed through in referencing a node below it, i.e., the sum of the weights of all the nodes in the subtree having it as the root. This field is referred to as TOTAL by Baer [3] and SIZE by Nievergelt and Reingold [19] (with reference to unweighted trees). The algorithm is one possible dynamic extension of the "Rule 2" criterion described by Mehlhorn [15].

The algorithm performs a single rotation on a referenced node if and only if the rotation would reduce the WPL. The specific criterion for this rotation is given in [3]. In words, a referenced node will be rotated up if and only if 2 times its access is greater than the access of its father plus the access of its son on the same side as the father. The algorithm is a special case of Algorithm WB presented in [3], which considers both single and double rotations on every node in the search path. Even further extensions would consider triple, quadruple, etc. rotations for every node in the search path.

The tendency of Algorithm A is to move up in the tree nodes which are referenced more often or which are passed through more often in referencing nodes below them in the tree. The former consideration tends to reduce the WPL by shortening the path length for nodes which are referenced more often. The latter consideration tends to reduce the WPL by giving the tree more balance. Iverson [10] discussed the importance of these two criteria in minimizing WPL, but he did not formally associate them with optimality.

The algorithm is very efficient compared with reoptimizing the entire tree each time it is searched. It utilizes the accesses of just 3 nodes to determine whether or not the tree is to be modified. If a modification is made, only 3 links and 2 accesses need to be changed. While the resulting tree is not necessarily optimal under the assumed weights, it is in general very nearly optimal and is much closer to optimality in general than the tree produced by Algorithm B.

It will be interesting at this point to show that the WPL of a tree can be computed as a simple function of the accesses of the nodes. In particular, for a tree containing n nodes with accesses $A_1, \cdots, A_n$, the WPL is equal to $\sum_{i=1}^{n} A_i$. This relationship can be seen fairly easily by noting that the access of a node is equal to the sum of the weights of every node in the subtree having the given node as the root. Thus for any node i having level $L_i > 0$, its weight W_i will appear once in its own access at level L_i, once in its father's access at level L_i-1, once in its grandfather's access at level L_i-2, and so on up to level 1. In other words it will appear L_i times in the summation $\sum_{i=1}^{n} A_i$. Therefore $\sum_{i=1}^{n} A_i = \sum_{i=1}^{n} L_i W_i$ = WPL.

Algorithm A is formally given as follows:

Introduction

This algorithm provides for building, searching, and maintaining a binary search tree in which every node P contains the following fields:

- KEY(P) - the key or name or identification of node P
- LLINK(P) - a pointer to the left subtree of node P
- RLINK(P) - a pointer to the right subtree of node P
- ACCESS(P) - the access of node P
- INFO(P) - the information content of node P

The tree has a special node called the head, whose only significant field is the right link, which points to the root of the tree. This pointer is initialized to null (λ) to signify an empty tree.

The algorithm is given an argument with key K, and with information INF if it is a new key. It searches the tree using the pointer P and returns with P pointing to the node requested, having inserted a new node if the key was not already in the tree. The algorithm uses the following pointers in addition to P:

- J1 = pointer to father of P
- J2 = LLINK or RLINK depending on whether P is a left son or right son
- J3 = pointer to grandfather of P

Algorithm (the underlined steps correspond to Algorithm B).

1. (Initialize) Set P ← pointer to head, J2 ← RLINK,J1 ← λ.
2. (Search node P) Set J3 ← J1, J1 ← P, P ← J2(J1). If P = λ, go to Step 3. Set ACCESS(P) ← ACCESS(P) + 1. If K < KEY(P), set J2 ← LLINK and go to Step 2. If K > KEY(P), set J2 ← RLINK and go to Step 2. Otherwise go to Step 4.
3. (Insert new node) Obtain an available cell and set P ← pointer to available cell. Set KEY(P) ← K,LLINK(P) ← λ, RLINK(P) ← λ, ACCESS(P) ← 1, INFO(P) ← INF,J2(J1) ← P. Return.
4. (Check for P pointing to root) If J3 = λ, return (no rotation possible).
5. (Get direction and pointer to son of P on same side as father of P) If J2 = LLINK, then set J4 ← RLINK, else set J4 ← LLINK. Set J5 ← J4(P).
6. (Compute change in WPL due to potential rotation. Return if change is not negative) Set DEL ← ACCESS(J1) + ACCESS(J5) - 2 · ACCESS(P). If DEL ≥ 0, return.
7. (Set new access of P and father of P) Set TEMP ← ACCESS(J1), ACCESS(J1) ← TEMP - ACCESS(P) + ACCESS(J5), ACCESS(P) ← TEMP.
8. (Set new links for P and father of P) Set J4(P) ← J1, J2(J1) ← J5.
9. (Compute direction to grandfather of P and set new link for grandfather) Set J2 ← LLINK. If LLINK(J3) ≠ J1, set J2 ← RLINK. Set J2(J3) ← P. Return.

As can be seen, the most important loop in the algorithm is Step 2. The extra processing needed for Algorithm A (over Algorithm B) is to update the access and to keep the pointer to the grandfather. If a new node is inserted (Step 3), the only extra processing is to initialize its access.

Steps 4-6 are, of course, extra steps required by the algorithm. They are not in an internal loop, however, thus they are executed at most once for each call of the algorithm. Steps 7-9 must in addition be executed once if a rotation is to be made.

The extra storage required by the algorithm for the representation of the tree consists of the ACCESS field for each node. If the INFO field is relatively large, then this extra storage is less significant.

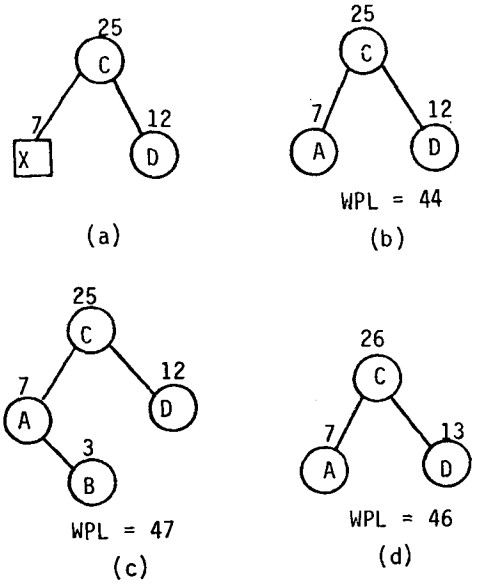
Algorithm W will not be given explicitly since it is similar in many respects to Algorithm A, and can easily be described as a modification to the latter. The ACCESS field becomes a WEIGHT field with probably the same storage requirement. The maintenance and use of the pointers and links is the same, and the rotations are the same except that the accesses (weights) are not changed. The main differences are that the incrementation of the ACCESS(WEIGHT) field is moved outside of the loop in Step 2, and the determination of whether or not to make a rotation involves only a comparison between the weight of node P and the weight of its father.

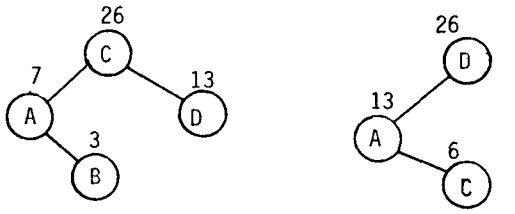
V. Insufficiency of Search Path for Maintaining Optimality

An obvious question arising from this discussion is concerned with whether or not it is possible to maintain an optimal tree utilizing only information along the search path of a key reference. Specifically, assuming that a binary search tree is originally optimal and that an additional reference has just been made to a node in the tree (or to a new node), is it possible to determine whether or not the updated tree is still optimal considering only the accesses of the nodes in the search path from the root to the referenced node? As we shall see, the answer to the question is "no", and hence none of Algorithms W, A, WB, or the suggested extensions to WB can guarantee the maintenance of an optimal tree.

Consider the trees illustrated in Figure 1. The letters A, B, C, D are the keys for the nodes, the letter X is a variable name for a subtree, and the numbers above the nodes are the accesses. Part (a) illustrates the portion of the tree that would be in the search path if node D were referenced, namely C and D. The subtree X would not be in the path. Parts (b) and (c) illustrate two possible configurations for the subtree. Both of these trees are optimal for the weights (or accesses) given, as can be seen by inspection.

If an additional reference is now made to node D, then the trees resulting are given in parts (d) and (e), respectively. It can be seen by inspection that the latter tree is optimal for the weights given, but the former tree is not optimal. Instead the tree in part (f) is optimal for the weights given in part (d). This illustration proves the assertion made earlier. Thus if we restrict ourselves to information along the search path of a node reference, then we cannot guarantee the maintenance of optimality.





Counterexample

Figure 1.

VI. Combination Algorithms

Algorithms A and W have a kind of dual relationship with Algorithms AVL and BB with regard to the criteria for performing rotations. Algorithms AVL and BB perform rotations only when a new node is inserted, whereas A and W perform rotations only on old nodes. Thus the complicated rotation logic will not need to be executed for AVL and BB for duplicate key requests, and in general after all distinct keys have been inserted. On the other hand, A and W will continue to test for rotations on every search, and carry out rotations when the test so dictates.

Most of the rotations in a tree maintained by Algorithm A would occur early with fewer and fewer changes as the tree becomes stablized. Thus it seems desirable to consider modifying A so that its balancing overhead is reduced or even eliminated after an initial volatile period. An obvious possibility is to use a combination of A and B, with A being used initially and then switching over to B when the tree has become relatively stable. This algorithm (call it AB) would combine the advantages of A (reduced WPL) and B (low overhead). Moreover, A would be used when it was most productive, i.e., at the beginning.

A number of criteria could be used for making the switch from A to B. The switch could be made after a specified number of searches or after a specified number of searches with no rotation. Another possibility would be to use B initially, but also maintaining the access field. Then switch to A for a period of time, performing any indicated rotations. Finally switch to a straight Algorithm B. The purpose of this scheme is to use Algorithm A only when it is most likely to yield the greatest reduction in WPL.

The most effective of these criteria according to the test results was the one based on a specified number (the "inactive count") of searches with no rotation, and it will be assumed henceforth that Algorithm AB is based on this switching criterion. The obvious rationale for the criterion is that several searches with no rotation indicates that the tree is rather stable, and that is would probably be more efficient to forego the extra overhead of A in favor of the speed of B. This criterion is more sensitive to different kinds of input data than the criterion based on number of searches alone, and hence seems more generally suitable.

The specification of the inactive count parameter permits an adjustment of the algorithm in terms of WPL and overhead. As the inactive count goes from 0 to infinity, both WPL and execution time go from that of Algorithm B to that of Algorithm A. While WPL is an approximately decreasing function of the inactive count, execution time is approximately unimodal. This adjustment feature is comparable to the adjustment of BB trees by varying the parameter α, and the adjustment of general height-balanced trees by varying the permitted level of imbalance.

The value minimizing the unimodal function should be used to set the inactive count parameter. This value may not be predictable, of course, and may vary with the nature of the data. Nevertheless, it seems generally reasonable that several searches in a row with no rebalancing required indicates the probable benefit of switching to Algorithm B. The execution time curve is relatively flat in the vicinity of the minimum, hence a rather wide range of values would give near optimal results. The results given here are for a parameter of 10.

It is clearly possible to combine Algorithm B with algorithms other than A, and achieve similar effects. A combination of W and B was tried, for example, with less success than AB. A combination of AVL and B gave good results, and will be reported on here as Algorithm AVLB. Even though AVL had less overhead to begin with than A, it still benefitted from a switch to B since the balancing and testing logic was no longer used after stability was attained. The switching criterion was the same as for AB, and a parameter value of 10 was also used.

VII. Results in Terms of WPL

The relative performance of the algorithms was basically independent of NDK and NKR. Values were used for NDK ranging from 5 to 300, and for NKR ranging from 50 to 5000. Ratios of NDK to NKR ranged from .01 to 1.00. WPL and execution time were approximately linear as a function of NKR and logarithmic as a function of NDK. Because of its insensitivity to NDK and NKR, the relative performance of the algorithms is accurately represented by their performance for the specific case of NDK = 200 and NKR = 1000, and most of the results are given for this case.

Two useful measures of algorithm performance are absolute improvement (AI) and fractional improvement (FI). AI gives the fraction by which an algorithm reduces the WPL over what it would be using Algorithm B, i.e.,

$$AI_{algorithm} = \frac{WPL_B - WPL_{algorithm}}{WPL_B}$$

FI gives the ratio of the reduction in WPL to the total possible reduction, i.e., the reduction yielded by Algorithm 0. In other words,

$$FI_{algorithm} = \frac{AI_{algorithm}}{AI_0} = \frac{WPL_B - WPL_{algorithm}}{WPL_B - WPL_0}$$

Table 1 presents the most important results from the simulations. The subscripts R, U, and Z denote random, unweighted, and Zipf's Law data, respectively. The subscript P denotes partially ordered data, with a 10% randomization from completely ordered. The subscript 25 denotes a simulation using NDK = 25 and NKR = 1000. All the other results are for NDK = 200 and NKR = 1000. Additional values of AI and FI, and for algorithm 0, are not given because of our primary interest in execution time as opposed to WPL.

Algorithm 0 is not included in the execution time results because of its nondynamic nature and obvious high overhead if it were executed after every search. It would be possible to design a combination algorithm in which algorithm 0 was activated only on occasion, when some balancing criterion was met. Many such criteria are possible, and Baer and Schwab [4] give some performance measures for one. No such results are included here. It would appear to be quite difficult to determine such an algorithm that would be as good as the combination algorithms used here, and also reasonably independent of the nature of the data.

With regard to WPL, there are several significant patterns and specific results. Algorithm 0 gives an approximately 30% reduction in WPL (AI_R (0) = .295) for the random data. Algorithm A is consistently the best of the remaining algorithms except for the partially ordered data. Its AI was .231 for random data, .215 for unweighted data, .258 for Zipf's Law data, .353 for partially ordered data, and .194 for NDK = 25. For random data, it produced approximately 78% of the total possible reduction in WPL (FI_R (A) = .781). Algorithms AVL and BB yielded very similar WPL's for the different trials. Algorithm R produced very poor WPL's which were in fact much worse than B's. Algorithm W was sometimes worse than B and sometimes better. As would be expected, AB and AVLB were in between B and A or AVL.

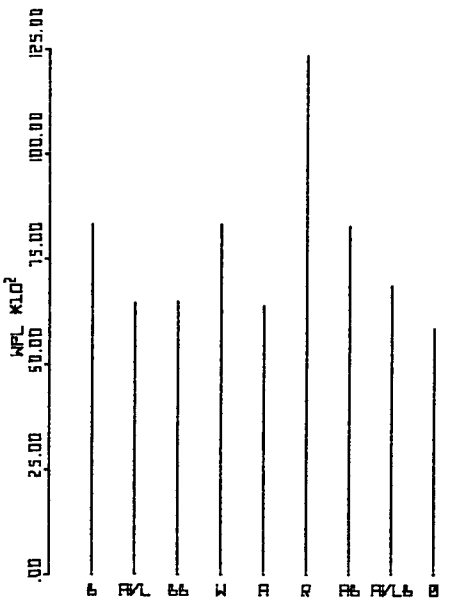
For the algorithms in general, WPL (and execution time) was higher for unweighted data and lower for Zipf's Law data, as compared to random data. This result is consistent with theoretical results of Mehlhorn [16], which show that for an optimal binary search tree, WPL is close to the entropy of the distribution of search key probabilities.

Figures 2, 3, and 4 provide a reasonably good summary of these results by illustrating WPL, AI, and FI, respectively, for the case of random data (negative AI and FI values are drawn as 0).

Performance Measure	ALGORITHM B	AVL	BB	W	A	R	AB	AVLB	0
WPL_R	8314	6465	6490	8316	6397	12350	8285	6851	5858
AI_R	.000	.222	.219	.000	.231	-.485	.003	.176	.295
FI_R	.000	.753	.743	.000	.781	-1.643	.012	.596	1.000
$Time_R$	5.67	8.94	15.69	14.21	12.48	17.41	5.76	5.49	
WPL_U	8202	6517	6519	9744	6440	15441	8202	6921	
$Time_U$	5.58	9.01	15.81	15.20	12.51	19.06	5.66	5.57	
WPL_Z	6320	5613	5599	5143	4692	6079	5995	5539	
$Time_Z$	4.48	7.71	13.24	8.90	9.32	10.26	4.44	4.37	
WPL_P	11867	6640	6584	10800	7674	32500	11291	9803	
$Time_P$	8.10	9.47	16.38	18.53	16.88	10.00	8.46	7.06	
WPL_{25}	4468	3800	3803	4119	3599		4245	3802	
$Time_{25}$	3.35	5.02	8.85	7.32	7.49		3.40	3.10	

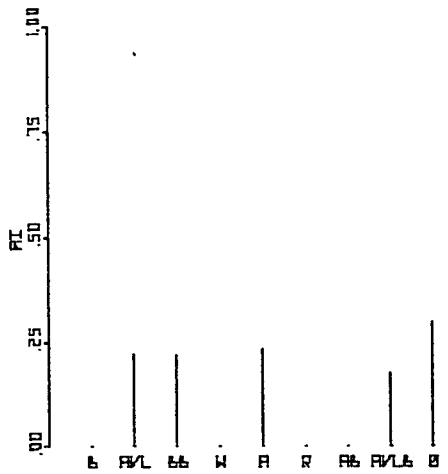
Algorithm Performance Measures

Table 1.



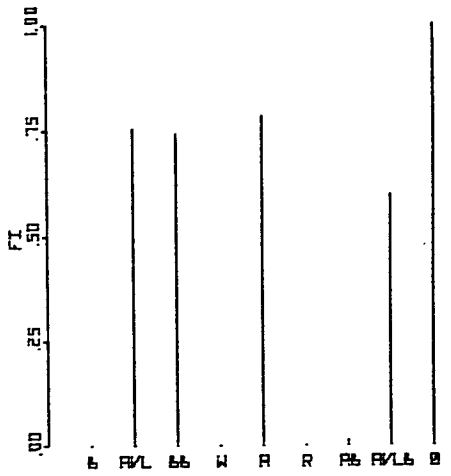
Weighted Path Length (Random Data)

Figure 2.



Absolute Improvement (Random Data)

Figure 3.



Fractional Improvement (Random Data)

Figure 4.

VIII. Results in Terms of Execution Time

Table 1 also includes execution time figures for the various algorithms being compared and the various types of data. As noted earlier, execution time is generally a more complete and hence valuable measure of algorithm performance, since it encompasses algorithm overhead as well as reduction in WPL. The time unit was 1 second.

The implementation consisted of a machine language program on a Cal Data 135 minicomputer. The computer contains a firmware emulation of a PDP 11/40 computer, and runs essentially like a PDP 11/40. Every effort was made to write code for the different algorithms in the same style and with equivalent high efficiency. Most of the algorithms were also implemented and executed in FORTRAN. The results were very similar to the results for the machine language implementation, and hence are not included here.

As for WPL, many important results and patterns are apparent. What might be surprising to some is that the relative algorithm performance is quite different, in fact closer to opposite, when measured in terms of execution time as opposed to WPL. Obviously the extra overhead required in reducing the WPL generally outweighed the benefits of reduced WPL.

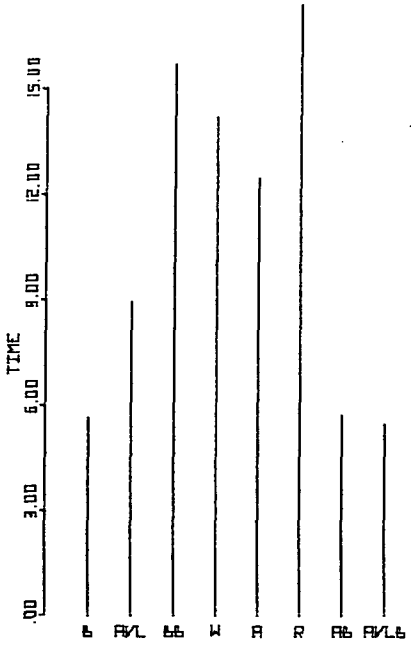
As can be seen, AVLB has the fastest time for every data type, with B a close second and AB a close third for every type except Zipf's Law data, for which they are reversed. The improvement which AVLB makes over B is 3.2%, 0.2%, 2.5%, 12.8%, 87.5% for the 5 different data types, respectively. The inactive count (10) used for AVLB and AB is not optimal for the given data, and in fact slightly improved results could be obtained if it were optimized. However, in most practical situations it would not be possible

to do such an optimization.

None of the straight algorithms AVL, BB, W, A, or R give a very good performance in terms of speed, with AVL being easily the best and R being generally the worst. Algorithm R does show especially good performance for the partially ordered data, which would seem reasonable. AVLB and AB obviously make a nice improvement over AVL and A, respectively.

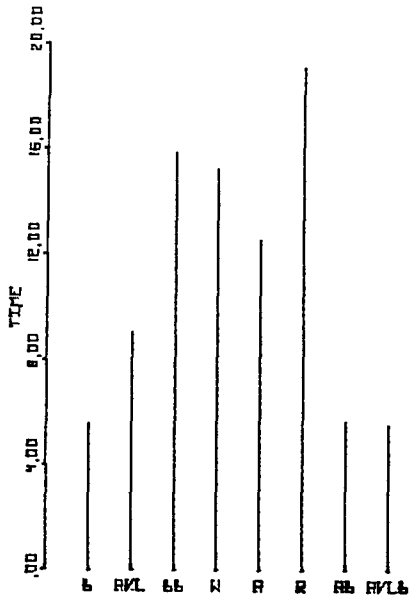
Figures 5, 6, 7, 8, and 9 provide a good illustration of the execution time results for the 5 different data types. Figures 10 and 11 combine and summarize these results by giving execution time for the different data types (10) and for the different algorithms (11).

A final test of the algorithms was made using partially ordered data as described in Section 3. Eight trials were made, using fractional exchanges of 0%, 1%, 5%, 10%, 20%, 50%, 200% and ∞. Algorithms tested were B, AVL, AB, and AVLB, using NDK = 25 and NKR = 1000. The results are illustrated in Figure 12, and again show that except in the most extreme case of partial ordering, AVLB is still the fastest algorithm and B is a close second.



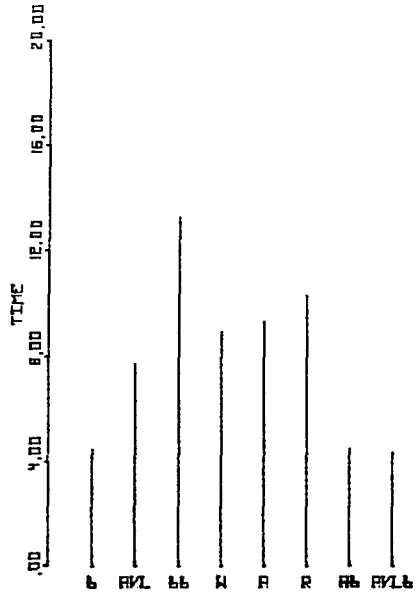
Execution Time (Random Data)

Figure 5.



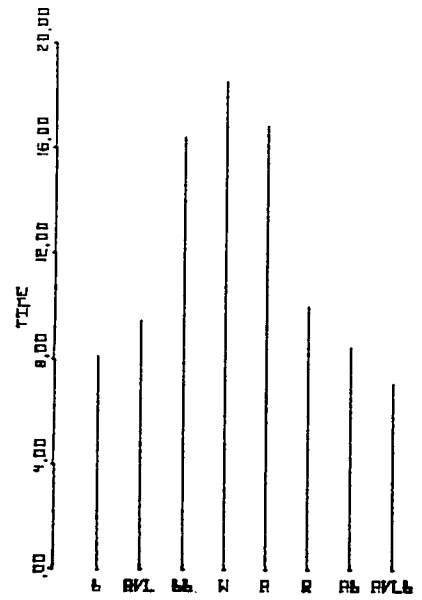
Execution Time (Unweighted Data)

Figure 6.



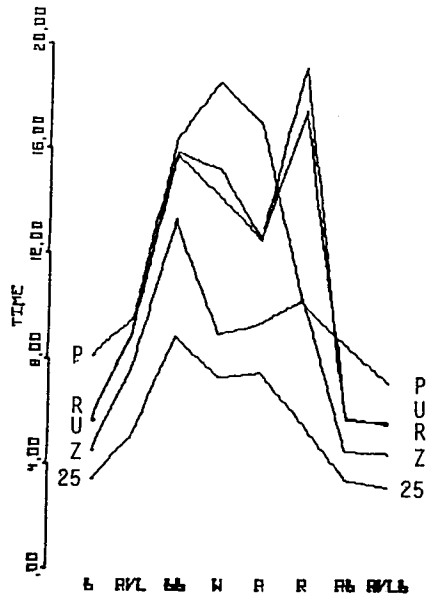
Execution Time (Zipf's Law Data)

Figure 7.



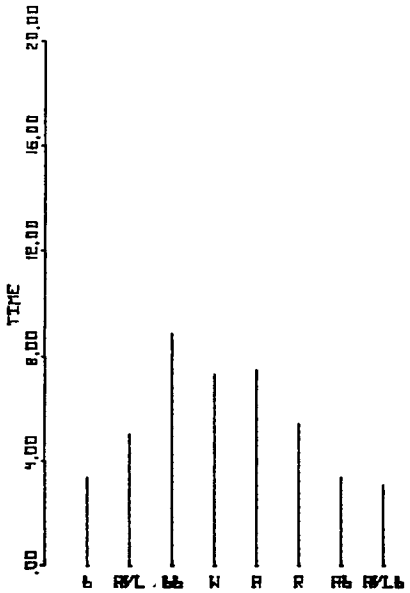
Execution Time (Partially Ordered Data)

Figure 8.



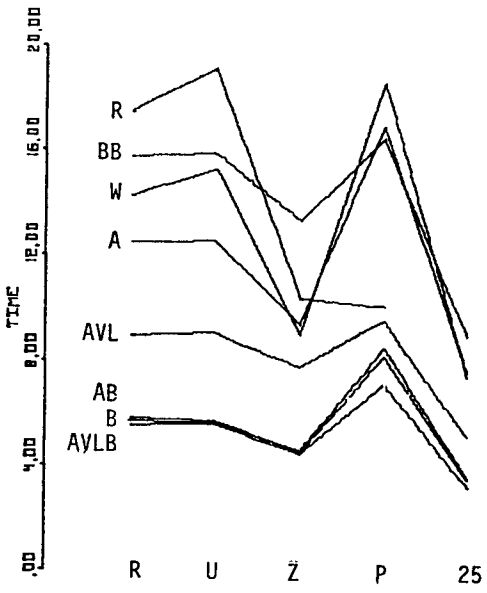
Execution Time (all Data)

Figure 10.



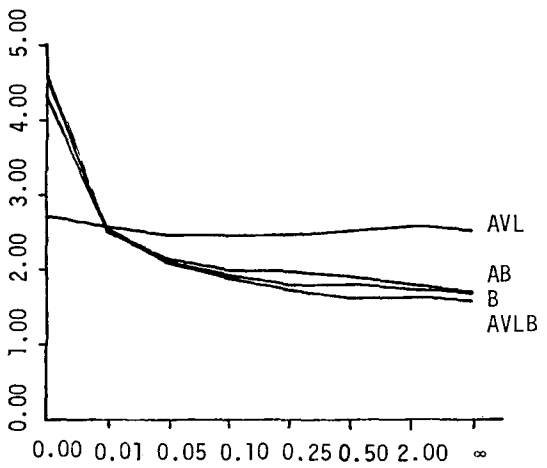
Execution Time (NDK = 25)

Figure 9.



Execution Time (all Algorithms)

Figure 11.



Execution Time for Partially Ordered Data

Figure 12.

IX. Earlier Results

As noted in Section 1, Baer and Schwab [4] and Karlton et. al. [11] have given execution time results for some of the cases considered here. The former paper was restricted to a consideration of nonweighted trees, and the latter to a consideration of height-balanced trees. Both papers include a common important conclusion: for nonweighted data, algorithm AVL is the fastest of the algorithms considered. In particular, AVL is faster than algorithm B. (Baer and Schwab specify that the number of accesses after insertion must be greater than 3.)

This conclusion clearly conflicts with the results of this paper, which show algorithm B to be substantially faster (approximately 35%) than algorithm AVL for both weighted and unweighted data. The conclusions in the other papers were essentially based on the same oversight: that the search time for algorithm AVL is longer than for B even if the key being searched for is already in the tree, i.e., if it is not a new key. In other words, algorithm AVL has extra overhead just for the searching, even if no balancing is attempted.

In particular, the equation for Q_A in [4, p. 329] should be

$$Q_A = C_A + qP_1 \simeq C_A + 1.5\ qnI_A$$
$$\simeq C_A + 1.5qn(.8I_R)$$
$$= C_A + 1.2\ qnI_R = C_A + 1.2\ q\ C_R,$$

where I_A and I_R are the average path lengths for algorithms AVL and B, respectively. Thus, according to the figures of Baer and Schwab, algorithm AVL provides a 20% reduction in WPL, but with a 50% increase in overhead just for the searching, and hence cannot be faster (on the average) than algorithm B. This analysis is quite consistent with the empirical results of this paper, noted earlier.

A similar analysis, incidentally, led to the decision not to test a second algorithm by Allan and Munro [2]. The algorithm, described as "move to root", is a modification of a technique for self-organizing linear tables (Knuth [13]). It is obvious that algorithm overhead would far exceed the benefits of reduced WPL even if the WPL became optimal. Thus the execution time would not be improved even though a nice reduction in WPL might be achieved, as shown by Allan and Munro. The algorithm might be quite favorable for clustered search key requests. Many types of clustering are possible, and this data type is not considered in this paper.

X. Conclusions

The most significant conclusion is that Algorithm B is probably the best algorithm to use in most cases. Although it generally yields the highest WPL, its low overhead gives it a generally faster execution time than all other algorithms except AVLB and possibly AB. The extra size of the latter two algorithms (AVLB is much longer than AB, which is much longer than B), plus the extra access or balance field in each node, would usually not be justified by the small increase in speed.

If speed is of the utmost importance, then AVLB is the best algorithm to use based on the results given here, with an improvement over B ranging from 0% to 13% or more. The improvement of AVLB over B would generally increase as the ratio of NKR to NDK increased, since additional references to a partially optimized tree would take additional advantage of reduced WPL without any increase in overhead. Note the 12.8% improvement for NDK = 25, as compared to the 3.2% improvement for NDK = 200, for random data.

None of the straight algorithms, AVL, BB, W, A, or R seem to be justified for the type of data considered here. Although they generally give a nice improvement in WPL, it is more than offset by the extra overhead. If an algorithm such as AVL is being used, a small extension (size-wise) to Algorithm AVLB would seem to be a reasonable consideration.

The conclusions presented hold for random data, unweighted data, Zipf's Law data, and partially ordered data using varying levels of partial ordering.

Certainly many other experiments could be carried out and analyzed. Different algorithms or versions of algorithms could be considered, different types of input data could be used and implementations could be made using other languages or other computers. The main conclusions of this study, however, seem to be quite reliable and general.

REFERENCES

1. Adel'Son-Vel'skii, G.M. and Landis, Ye.M., "An Algorithm for the Organization of Information", Soviet Math. 3 (1962), pp. 1259-1263.

2. Allan, B. and Munro, P., "Self-Organizing Binary Search Trees", 17th IEEE Symposium on Foundations of Computer Science, 1976, pp. 166-172.

3. Baer, J.L., "Weight-Balanced Trees", Proc. AFIPS 1975 NCC., Volume 44, AFIPS Press, Montvale, N.J., pp. 467-472.

4. Baer, J.L. and Schwab, B., "A Comparison of Tree-Balancing Algorithms", CACM 20, 5, (1977) pp. 322-330.

5. Bertztiss, A.T., Data Structures Theory and Practice, Second Edition, Academic Press, New York, 1975, pp. 245-247.

6. Bruno, J. and Coffman, E.G., "Nearly Optimal Binary Search Trees", IFIP Congress 1971, North-Holland, pp. 99-103.

7. Foster, C.C., "A Generalization of AVL Trees", CACM 16, 8 (1973), pp. 513-517.

8. Hibbard, T., "Some Combinatorial Properties of Certain Trees", JACM 9, (1962), pp. 13-28.

9. Hu, T.C. and Tucker, A.C., "Optimal Computer Search Trees and Variable-Length Alphabetical Codes", SIAM J. Applied Math 21, (1974)., pp. 514-532.

10. Iverson, K.E., A Programming Language, Wiley, New York, 1962, pp. 142-144.

11. Karlton, P.L., Fuller, S.H., Scroggs, R.E., and Kaehler, E.B., "Performance of Height-Balanced Trees", CACM 19, (1976), pp. 23-28.

12. Knuth, D.E., The Art of Computer Programming, Volume 3, Addison-Wesley, Reading, Mass., 1973, pp. 424-425.

13. Ibid, pp. 397-399.

14. Knuth, D.E., "Optimum Binary Search Trees", Acta Informatica 1, (1971), pp. 14-25.

15. Mehlhorn, K., "Nearly Optimal Binary Search Trees", Acta Informatica 5, (1975), pp. 287-295.

16. Mehlhorn, K., "Best Possible Bounds on the Weighted Path Length of Optimum Binary Search Trees", SIAM J. Comput. 6, 2 (1977).

17. Mehlhorn, K., "Dynamic Binary Search", 4th JCALP, Springer Lecture Notes, Vol. 52, pp. 323-336.

18. Nievergelt, J., "Binary Search Trees and File Organization", Computing Surveys 6, 3 (1974), pp. 195-207.

19. Nievergelt, J. and Reingold, E.M., "Binary Search Trees of Bounded Balance", SIAM J. Comput. 2, (1973), pp. 33-43.

20. Nievergelt, J. and Wong, C.K., "On Binary Search Trees", IFIP Congress 1971, North-Holland, 1972, pp. 91-98.

21. Nievergelt, J. and Wong, C.K., "Upper Bounds for the Total Path Length of Binary Trees", JACM 20, (1973), pp. 1-6.

22. Walker, W.A. and Gottlieb, C.C., "A Top Down Algorithm for Constructing Nearly Optimal Lexicographic Trees", Graph Theory and Computing, R.C. Read (ed.), Academic Press, pp. 302-323.

23. Weiner, P., "On the Heuristic Design of Binary Search Trees", Proc. Fifth Annual Princeton Conference, Princeton, New Jersey, Mary 1971 (abstract only).