BINARY SEARCH TREES OF BOUNDED BALANCE[†]

J. Nievergelt[‡] and E. M. Reingold
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

A new class of binary search trees, called trees of bounded balance, is introduced. These trees are easy to maintain in their form despite insertions and deletions of nodes, and the search time is only moderately longer than in completely balanced trees. Trees of bounded balance differ from other classes of binary search trees in that they contain a parameter which can be varied so the compromise between short search time and infrequent restructuring can be chosen arbitrarily.

Introduction

Binary search trees are an important technique for organizing large files, because they are efficient for both random and sequential access of records in a file. Two main problems have received attention in the recent literature, each concerned with the search time in such trees.

The first has to do with trees on a fixed set of names (or keys) and associated probabilities. Knuth [Kn70] and Hu and Tucker [Hu71] have given algorithms for constructing optimal trees. Bruno and Coffman [Br71], and Walker and Gotlieb [Wa71] have given fast algorithms for constructing near-optimal trees. Nievergelt and Wong [Ni71] have shown that asymptotically, both optimal and balanced trees have the same average search time. With these investigations, the problem of trees on a fixed set of names appears to be settled, at least temporarily.

The second problem, which we consider to be of greater practical importance because of its more realistic assumptions, has to do with trees over a set of names which is dynamic, one which changes in time through insertions and deletions. Hibbard [Hi62] determined how the average search time behaves if trees are left to grow at random. To improve the search time over that of trees which have grown at random, one looks for trees which satisfy three conflicting requirements: they must be close to being balanced, so that the search time is short; one must be able to restructure them easily when they have become too unbalanced; and this restructuring should be required only rarely. Adel'son-Vel'skii and Landis [Ad62] (see also [Fo65]) described a class of trees, now known as AVL trees, which strike an elegant compromise between these conflicting requirements.

[†]This work was supported in part by the National Science Foundation (Grant GJ-31222).

[‡]On leave at Computer-Wissenschaften, Federal Institute of Technology, 8006 Zurich, Switzerland.

This paper is intended as a contribution to the second topic. A new class of binary search trees, called trees of bounded balance, or BB trees for short, is described. BB trees share with the AVL trees of [Ad62] the property that they are easy to maintain in their form despite insertions and deletions of nodes, and that search time is only moderately longer than in balanced trees. They differ from AVL trees in two important respects:

- they contain a parameter which can be varied so the compromise between short search time and frequency of restructuring can be chosen arbitrarily
- the insertion and deletion algorithms do not require a pushdown store.

Trees of bounded balance

Definition The empty tree, T_0, of zero nodes is a binary tree. A binary tree T_n of $n \geq 1$ nodes is an ordered triple (T_ℓ, v, T_r), where T_ℓ, T_r are binary trees of ℓ, r nodes respectively, $\ell \geq 0$, $r \geq 0$, $\ell+r=n-1$, and v is a single node called the root of T_n.

Definition The height of a binary tree T_n is zero if $n = 0$ and one if $n = 1$, otherwise it is given by max(height of T_r, height of T_ℓ) + 1.

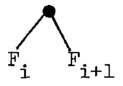
Definition The internal path length $|T_n|$ of a binary tree T_n is zero if $n \leq 1$, otherwise it is given by $|T_n| = |T_\ell| + |T_r| + n - 1$.

Definition The root-balance $\rho(T_n)$ of a binary tree $T_n = (T_\ell, v, T_r)$ of $n \geq 1$ nodes is $\rho(T_n) = \frac{\ell+1}{n+1}$.

Definition A binary tree T_n is said to be of bounded balance α, or in BB[α], for $0 \leq \alpha \leq 1/2$, if and only if either $n \leq 1$ or, for $n > 1$ and $T_n = (T_\ell, v, T_r)$, the following hold:

1. $\alpha \leq \rho(T_n) \leq 1 - \alpha$, and
2. both T_ℓ and T_r are of bounded balance α.

The notion of root-balance is taken, with slight modification, from Nievergelt and Wong [Ni70]. It is always in the range $0 < \rho(T_n) < 1$, and it indicates the relative number of nodes in the left and right subtrees of T_n. Thus the completely balanced trees T_n of $n = 2^k-1$ nodes are in BB[1/2], while the Fibonacci trees defined by

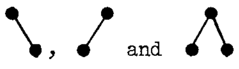
$F_0 = F_1 = \bullet$, $F_{i+2} =$ 

are in BB[1/3].

It is interesting to note that there is a "gap" in the balances of trees:

Theorem 1 For all α in the range $1/3<\alpha<1/2$, $BB[\alpha] = BB[1/2]$.

Proof Clearly $BB[\alpha] \supseteq BB[1/2]$. Suppose there is a tree in $BB[\alpha] - BB[1/2]$. This tree is not completely balanced so it clearly cannot have height one. The only trees of height two are



which have root-balances $1/3$, $2/3$, and $1/2$, respectively. The proof continues by induction. Suppose that the shortest tree T in $BB[\alpha] - BB[1/2]$ has height $k > 2$. Then, by definition, its left and right subtrees must be in $BB[\alpha]$ and by induction they cannot be in $BB[\alpha] - BB[1/2]$, hence they must be in $BB[1/2]$. Let them have 2^s-1 and 2^t-1 nodes, respectively. The root-balance of T is then given by

$$\rho(T) = \frac{2^s-1+1}{2^s-1+2^t-1+1+1} = \frac{2^s}{2^s+2^t}$$

Now, since T is not in $BB[1/2]$, $s \neq t$ and without loss of generality we can assume that $s < t$ (since we can consider the mirror image of the tree). Thus

$$\rho(T) = \frac{1}{1+2^{t-s}} \leq \frac{1}{3}$$

so that $\alpha \leq 1/3$, which is a contradiction. ∎

Search time in BB trees

The height of T_n is the worst case time to search T_n. Since the internal path length T_n can be expressed as the sum, over all nodes, of the length of the (unique) path from the root of T_n to each node, it is clear that $\frac{1}{n}|T_n|$ is the average time to search T_n.

The following theorem is due to Nievergelt and Wong [Ni70]:

Theorem 2 If T_n is in $BB[\alpha]$ then

$$|T_n| \leq \frac{1}{H(\alpha)} (n+1)\log(n+1) - 2n$$

where

$$H(\alpha) = -\alpha\log\alpha - (1-\alpha)\log(1-\alpha).^{\dagger}$$

It is not difficult to show

Theorem 3 If T_n is in $BB[\alpha]$ then the height of T_n is at most

$$\frac{\log(n+1)}{\log(\frac{1}{1-\alpha})}$$

Proof: By induction on n. ∎

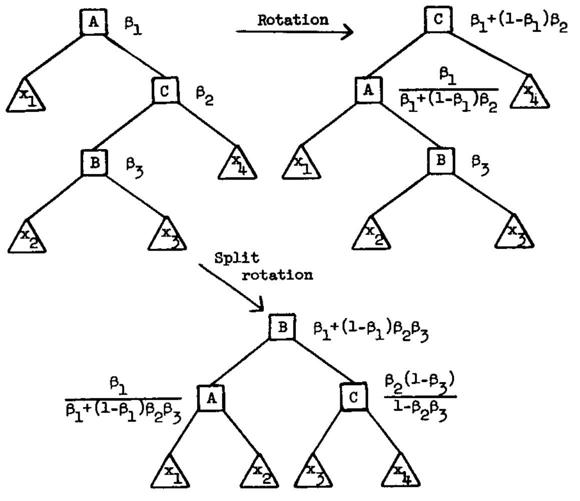
† Throughout this paper, all logarithms are taken base 2.

The bounds in both of these theorems are sharp since they are exact for $\alpha = 1/2$, i.e. the completely balanced trees of $2^k - 1$ nodes.

From these theorems we can easily deduce the average and worst case search times for trees in $BB[\alpha]$, for any α. For example, trees in $BB[1/3]$ look "sparse," but their internal path length is at most 9% longer than it is for completely balanced trees with the same number of nodes; this follows immediately from Theorem 2 since $1/H(1/3) \approx 1.09$. Hence, searching a tree in $BB[1/3]$ will take, on the average, at most 9% longer than searching a completely balanced tree with the same number of nodes. Similarly, it follows from Theorem 3 that searching a tree in $BB[1/3]$ will take, in the worst case, at most 70% longer.

Rebalancing BB Trees

If upon the addition or deletion of a node to a tree in $BB[\alpha]$ the tree becomes unbalanced relative to α, that is, some subtree of T_n has root balance outside the range $[\alpha, 1-\alpha]$, then that subtree can be rebalanced by certain tree transformations which are of two types (ignoring symmetrical variants):


A β_1
Rotation
C $\beta_1+(1-\beta_1)\beta_2$
C β_2
A $\frac{\beta_1}{\beta_1+(1-\beta_1)\beta_2}$
B β_3
x_1
x_2
x_3
x_4
Split rotation
B $\beta_1+(1-\beta_1)\beta_2\beta_3$
$\frac{\beta_1}{\beta_1+(1-\beta_1)\beta_2\beta_3}$ A
C $\frac{\beta_2(1-\beta_3)}{1-\beta_2\beta_3}$

In the above diagram we have used squares to represent nodes and triangles to represent subtrees; the root-balance is given beside each node.

Theorem 4 If $\alpha \leq 1 - \frac{\sqrt{2}}{2}$ and the insertion or deletion of a node in a tree in $BB[\alpha]$ causes a subtree of that tree to have root-balance less than α, and if β_2 is the balance of the right subtree of the unbalanced subtree after the insertion or deletion of the node, then the unbalanced subtree can be rebalanced by performing a rotation if $\alpha \leq \beta_2 < \frac{1-2\alpha}{1-\alpha}$ and performing a split-rotation if β_2 is outside of this range.

Proof Under the various hypotheses, it is not difficult to show that after the transformation has been applied, the new balances are all in the range $[\alpha, 1-\alpha]$. ■

If the balance of a subtree goes above $1-\alpha$, then we use the mirror images of these transformations, and a corresponding theorem. Introducing additional transformations, say split-split-rotations, will probably increase the allowable range of α. However, since $1 - \frac{\sqrt{2}}{2} \approx .2928$ and $BB[\alpha] - BB[1/2]$ is empty for $1/2 > \alpha > 1/3$, $\alpha = 1 - \frac{\sqrt{2}}{2}$ is a reasonable choice. By Theorems 2 and 3 we know that the average search time will be no worse than 15% longer for a $BB[1 - \frac{\sqrt{2}}{2}]$ tree than for a completely balanced tree with the same number of nodes, while the worst case search time will be at most twice as long. Considering the ease with which nodes can be added and deleted from BB trees, this moderate increase in search time is justifiable.

Insertion and deletion in BB trees

Assume that each node N of the tree has the form

LLINK	DATA	SIZE	RLINK

and that the DATA field of every node in the left subtree of N is lexicographically before DATA(N), while the DATA field of every node in the right subtree is lexicographically after DATA(N); thus the tree is a search tree relative to the lexicographic ordering. SIZE(N) is the number of nodes in the subtree whose root is the node N.

The following algorithm, given in detail in the Appendix, inserts the name NEW to the tree, preserving both the balance and the ordering: Follow links down through the tree going left if NEW is less than the node and right otherwise. If NEW is found to be equal to a name in the tree, then carry out the procedure described in the next paragraph. At each stage of the search, check to see whether the addition of a node to the subtree will unbalance the tree; if not, add one to the size field and continue down the tree. If the subtree does become unbalanced, then perform the appropriate transformation before continuing down the tree.

Notice that we may be modifying the tree for nothing in the event that we discover that NEW is already in the tree after modifications have been made. In that case we retrace the path down the tree correcting the SIZE fields, but <u>not</u> restructuring the tree: the restructuring which has been done, albeit unnecessarily has improved the balance of the tree.

Deletion of a node is similar: Follow links down through the tree as before, subtracting one from each SIZE field. If a subtree thus becomes unbalanced, perform the appropriate transformation and continue down the tree. When we arrive at the node to be deleted and it is not a leaf or a semileaf (a node whose only son is a leaf), find its postorder successor (or predecessor, depending which most improves the balance) and promote it to the place of the node to be deleted, taking care to adjust the appropriate SIZE fields and links. If the node to be deleted is a semileaf, promote its son to take its place. If the node to be deleted is a leaf, simply delete it. Again, if transformations have been made and we find that the node to be deleted is not in the tree, we correct the size fields in a second top down pass, but do not restructure the tree.

The time required by the insertion and deletion algorithms is clearly proportional to the search time; thus Theorems 2 and 3 demonstrate that insertion and deletion require $O(\log n)$ time. Of obvious interest is the coefficient of the $\log n$. This coefficient will be the same for insertion and deletion as it is for searching, plus whatever time is required to do the rebalancings. Hence it is important to know the expected number of transformations which must be performed during insertion or deletion.

In order to proceed with such an analysis, we must assume some sort of distribution of root-balances in trees of n nodes. Given a tree in $BB[\alpha]$, insertions and deletions have the effect of shifting the root-balance around in the interval $[\alpha, 1-\alpha]$. The behavior of the root-balance under insertions and deletions is quite similar to a discrete, one dimensional random walk with reflecting barriers -- when a step would take the root-balance outside the interval, a transformation is applied and the root-balance moves closer to 1/2. According to probability theory (see Feller [Fe68, p. 391]), the distribution of positions for such a random walk is <u>uniform</u> over the interval and hence <u>this is the assumption we will make</u>. This assumption is weak, however, since the barriers of the random walk corresponding to the shifting of the root-balance are what might be called "repulsing;" they don't just reflect the particle back the same distance that it tried to go forward, but rather they repulse the particle (quite strongly) to send it closer to 1/2. It is thus likely that a more accurate assumption would be a truncated normal distribution centered at 1/2.

<u>Theorem 4</u> Under the (weak) assumption that distribution of root-balances in a $BB[\alpha]$ tree is uniform over $[\alpha, 1-\alpha]$, the expected number of tree transformations required for insertion or deletion of a node is less than $\frac{2}{1-2\alpha}$.

<u>Proof</u> If T_n is a tree of n nodes in $BB[\alpha]$ with ℓ and r nodes in its left and right subtrees, respectively, then

$$\alpha(n+1) - 1 \leq \ell,\ r \leq (1-\alpha)(n+1) - 1 .$$

For simplicity, we will approximate these lower and upper bounds by αn and $(1-\alpha)n$, respectively. Since the root-balances are uniformly distributed in $[\alpha, 1-\alpha]$, each of the $(1-\alpha)n - \alpha n = (1-2\alpha)n$ possible values for ℓ and r is equally likely to occur as the number of nodes in the left and right subtrees, respectively, of T_n. Of all the $(1-2\alpha)n$ possible values, only two are critical for insertion or deletion, the largest and the smallest; only for these balances can insertion or deletion cause the tree to go out of $BB[\alpha]$. Thus the probability, p_n, of causing the root-balance of T_n to go out of the interval $[\alpha, 1-\alpha]$ is

$$p_n = \frac{1}{(1-2\alpha)n}^{\dagger}$$

and this is also the probability of having to apply a transformation during insertion or deletion.

Now the expected number of nodes in the two subtrees of a tree in $BB[\alpha]$ with m nodes is $m/2$ since the average root-balance is 1/2 by the uniform distribution assumption. Hence the expected number of nodes whose root-balances will go out of $[\alpha, 1-\alpha]$ and will thus need rebalancing is, assuming for simplicity that n is a power of two,

$$p_n + p_{\frac{n}{2}} + p_{\frac{n}{4}} + \dots + p_{\frac{n}{n}}$$

$$= \sum_{i=0}^{\log n} \frac{1}{(1-2\alpha)n/2^i} = \frac{1}{(1-2\alpha)n} \sum_{i=0}^{\log n} 2^i$$

$$= \frac{2n-1}{(1-2\alpha)n} < \frac{2}{1-2\alpha} . \blacksquare$$

It is remarkable that the expected number of rebalancings is <u>independent of the size of the tree</u>. For example, we find that on the average only 4.85 rebalancings will be necessary to insert or delete a node when the tree is in $BB[1-\sqrt{2}/2]$.

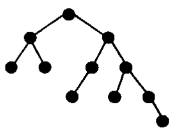
<u>Comparison with AVL trees</u>

AVL trees are characterized by the fact that the difference between the heights of the left and right subtrees of any node is at most one; for example, the Fibonacci trees described earlier are a special case of AVL trees. The same two transformations serve to restructure an AVL tree which has been upset by an insertion or deletion, and, as in Theorem 4, the expected number of transformations which must be applied to rebalance an AVL tree during insertion or deletion is a constant.

AVL trees cannot be described as $BB[\alpha]$ for any α:

<u>Theorem 5</u> For all $\epsilon > 0$ there is an AVL tree with root-balance less than ϵ, and there are trees in $BB[1/3]$ which are not AVL.

<u>Proof</u> The first part is shown considering a tree whose left subtree is the Fibonacci tree of height n and whose right subtree is the completely balanced tree of height n. As $n\to\infty$ the balance of such a tree, which is AVL, goes to zero. The second part of the theorem is shown by considering trees such as



which is in $BB[1/3]$ but which is not AVL. ■

The search time for AVL trees is somewhat better than for BB trees. The worst case search time for AVL trees is about $1.44\log n$ comparisons. The average search time can be as bad as $1.05\log n$ comparisons; this is the average search time for Fibonacci trees, the exact bound is unknown. In contrast, the search times for a tree in $BB[\alpha]$ are given by Theorems 2 and 3. For example, when $\alpha = 1 - \frac{\sqrt{2}}{2}$, we found the worst case search time is about $2\log n$ and the average search time is less than $1.15\log n$.

Insertion and deletion of nodes in AVL trees require a top-down pass over the path from the root to the node to be inserted or deleted, followed by a bottom-up pass over that same path. Typically, this is accomplished by the use of a pushdown stack, yielding algorithms which require $O(\log n)$ time. The use of a stack can be eliminated, but the resulting algorithms are $O((\log n)^2)$. In most cases, insertion and deletion of nodes in BB trees is accomplished by a single top-down pass over the path. In the event of a redundant insertion or deletion, a second top-down pass is necessary. Unlike a bottom-up pass, an additional top-down pass does not require the use of a pushdown stack.

The table at the top of the following page summarizes the comparison of random trees ($BB[0]$), AVL trees, $BB[1-\frac{\sqrt{2}}{2}]$ trees, and completely balanced trees ($BB[1/2]$ if we ignore semileaves).

One important advantage BB trees have over AVL trees is that the trade off between search time and insertion/deletion time can be specified by the appropriate choice of α, the bound on the balance. Thus when insertions and deletions are rare, α could be chosen close to $1 - \frac{\sqrt{2}}{2}$ while if insertions and deletions are very frequent, α could be chosen closer to zero.

<u>Appendix: The insertion algorithm</u>

The algorithm presented in this appendix is given in sufficient detail to make its implementation fairly easy; we have implemented and tested it in SNOBOL. The purpose of giving a detailed version of it here is to display its relative simplicity.

Assume that each node of the tree has the form

LLINK	DATA	SIZE	RLINK

with the four fields as previously described. To simplify notation, if T is a pointer to a tree

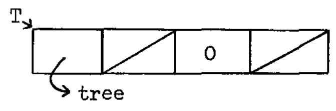
†This tacitly assumes that the node is inserted (deleted) in the heavier subtree (from the lighter subtree) with probability 1/2.

	Worst possible search time	Expected average search time	Restructuring algorithms
Random trees of n nodes (BB[0])	n	$1.39 \lg n$ [†]	Very easy, but in the worst case the time can be as bad as $O(n)$.
AVL trees of n nodes	$1.44 \lg n$	?	$O(\lg n)$ with a pushdown stack; $O((\lg n)^2)$ without the stack.
BB$[1-\frac{\sqrt{2}}{2}]$ trees of n nodes	$2 \lg n$	$? < 1.115 \lg n$ [‡]	$O(\lg n)$ without a pushdown stack.
Completely balanced trees of n nodes (BB[1/2], ignoring semileaves)	$\lg n$	$\lg n$	Difficult, and it can take as long as $O(n)$.

whose root is such a node then

$$\|T\| = \begin{cases} 0 & \text{if } T \text{ is empty} \\ \text{SIZE}(T) & \text{otherwise} \end{cases}$$

The name NEW is to be added to the BB[α] tree pointed to by the header node



R is a pointer which will be used in the search through the tree to find out where N should be added. RP is always one step behind R in the tree; that is, RP will point to the father of the node pointed to by R. S is a variable whose value is either "L" or "R" and S·LINK is either LLINK or RLINK according to the value of S. For example

S = "L"

S·LINK(P)←P

has the same effect as

LLINK(P)←P .

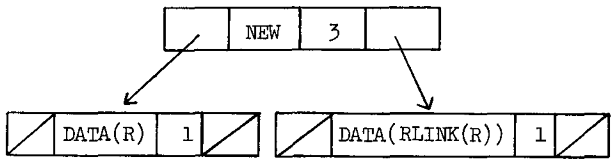
The value of S together with the value of RP tell us which pointer has to be modified when we rebalance a subtree.

Step 1 (Initialize) Set RP←T and S←"L". Now the pointer which is one step behind points to the header node of the tree.

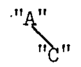
Step 2 (Small tree?) Set R = S·LINK(RP); this moves us down one level in the tree. If $\frac{1}{\|R\|+2} \leq \alpha$ then insert NEW in the subtree pointed to by R using the obvious method, i.e. without any rebalancing; we can do this if the tree is small enough. If in doing this insertion we discover that NEW is already in the tree, then go to Step 9.

Step 3 (Compare) Compare NEW to DATA(R). If NEW = DATA(R) then the name is already in the tree, so we go to Step 9. If NEW < DATA(R), go to Step 7.

Step 4 (Rotation no help?) If $\|R\| = 2$, RLINK(R) is not null, and DATA(RLINK(R)) > NEW. Then set S·LINK(PR) to point to the structure



and stop. When $\alpha < 1/4$ this keeps the insertion algorithm from going into an infinite loop on a subtree of two nodes when the name to be inserted lies between them, e.g.



when we try to insert "B".

Step 5 (Add to right subtree) Compute what the new balance of R will be after insertion:

$$\nu = \frac{\|\text{LEFT}(R)\|+1}{\|R\|+2}$$

† This is due to Hibbard [Hi62].

‡ Experimental evidence suggests that this is less than $1.05 \lg n$.

If $\alpha \leq \nu \leq 1 - \alpha$ then no rebalancing is needed at this level, so set

$$SIZE(R) \leftarrow SIZE(R) + 1$$
$$S \leftarrow "R"$$
$$PR \leftarrow R$$
$$R \leftarrow RLINK(R)$$

and go to Step 2.

Step 6 (Rebalance from right to left) Rebalance, using the transformations given in the figure, before adding the name. If $\|R\| = 2$ then use rotation. Otherwise, compute the value β_2 will have after the insertion of NEW. If $\alpha \leq \beta_2 < \frac{1-2\alpha}{1-\alpha}$ then use rotation, otherwise use split-rotation. Set S·LINK(PR) to point to the rebalanced subtree and go to Step 2.

Step 7 (Add to left subtree) Compute what the new balance of R will be after insertion:

$$\nu = \frac{\|LEFT(R)\|+2}{\|R\|+2}$$

If $\alpha \leq \nu \leq 1 - \alpha$ then no rebalancing is needed at this level, so set

$$SIZE(R) \leftarrow SIZE(R) + 1$$
$$S \leftarrow "L"$$
$$PR \leftarrow R$$
$$R \leftarrow LLINK(R)$$

and go to Step 2.

Step 8 (Rebalance from left to right) Rebalance, using the mirror images of the transformations in the figure, before adding the name. If $\|R\| = 2$ then use rotation. Otherwise compute the value β_2 will have after the insertion of NEW. If $\alpha \leq 1 - \beta_2 < \frac{1-2\alpha}{1-\alpha}$ then use rotation, otherwise use split-rotation. Set S·LINK(PR) to point to the rebalanced subtree and go to Step 2.

Step 9 (Duplicate name) We have found that NEW is already in the tree, so a second top-down pass is needed to correct the size fields. Set R ← LLINK(T) so it points to the top of the tree.

Step 10 (Correct size field) Compare NEW to DATA(R). If NEW = DATA(R) we are done. Otherwise set SIZE(R) ← SIZE(R) - 1. Then, if NEW > DATA(R) set R ← RIGHT(R), otherwise set R ← LEFT(R). Repeat Step 10.

References

[Ad62] Adel'son-Vel'skii, G. M. and Landis, Ye. M. An algorithm for the organization of information, Dokl. Akad. Nauk SSSR 146 (1962), 263-266 (Russian). English translation in Soviet Math. Dokl. 3 (1962), 1259-1262.

[Br71] Bruno, J. and Coffman, E. G. Nearly optimal binary search trees, Proc. IFIP Congress 71 (1971).

[Fe68] Feller, W. An Introduction to Probability Theory and Its Application, Volume 1, 3rd edition, Wiley, New York, 1968.

[Fo65] Foster, C. C. Information storage and retrieval using AVL trees, Proc. of ACM 20th National Conf. (1965), 192-205.

[Hi62] Hibbard, T. Some combinatorial properties of certain trees, J. ACM 9 (1962), 13-28.

[Hu71] Hu, T. C. and Tucker, A. C. Optimal computer search trees and variable-length alphabetical codes, SIAM J. Appl. Math. 21 (1971), 514-532.

[Kn70] Knuth, D. E. Optimum binary search trees, Acta Informatica 1 (1971), 14-25.

[Ni70] Nievergelt, J. and Wong, C. K. Upper bounds for the total path length of binary trees, IBM T. J. Watson Research Center Report Number RC3075 (1970).

[Ni71] _______, and _______. On binary search trees, Proc. IFIP Congress 71 (1971).

[Wa71] Walker, W. A. and Gotlieb, C. C. A top down algorithm for constructing nearly-optimal lexicographic trees, University of Toronto, Department of Computer Science Technical Report Number 26 (1971).