SIAM J. COMPUT.
Vol. 2, No. 1, March 1973

BINARY SEARCH TREES OF BOUNDED BALANCE*

J. NIEVERGELT AND E. M. REINGOLD†

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.

Key words. Balanced trees, lexicographic trees, binary search trees, table look-up.

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 (1971) and Hu and Tucker (1971) have given algorithms for constructing optimal trees; see Knuth (1973). Bruno and Coffman (1972), and Walker and Gotlieb (1972) have given fast algorithms for constructing near-optimal trees. Nievergelt and Wong (1972) have shown that asymptotically, both optimal and balanced trees have the same average search time.

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 (1962) 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 (1962) (see also Foster (1965) and Knuth (1973)) described a class of trees, now known as AVL trees or height-balanced trees, which strike an elegant compromise between these conflicting requirements.

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 height-balanced trees of Adel'son-Vel'skii and Landis (1962) 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 height-balanced trees in one important respect: They contain a parameter which can be varied so the compromise between short search time and frequency of restructuring can be chosen arbitrarily.

* Received by the editors July 5, 1972, and in revised form January 19, 1973.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. This work was supported in part by the National Science Foundation under Grant GJ-31222.

Trees of bounded balance.

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

DEFINITION. The *height* of a binary tree T_n of $n \geqq 1$ nodes is zero if $n = 1$, otherwise it is given by max (height of T_ℓ, height of T_r) + 1.

DEFINITION. The *internal path length* $|T_n|$ of a binary tree T_n is zero if $n \leqq 1$, 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 $n \geqq 1$ nodes is $\rho(T_n) = (\ell + 1)/(n + 1)$.

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

1. $\alpha \leqq \rho(T_n) \leqq 1 - \alpha$, 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 (1973). 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

$$F_0 = \text{empty}, \quad F_1 = \bullet, \quad F_{i+2} = \bullet \text{ with children } F_i, F_{i+1}$$

can be shown to be in BB[1/3].

It is interesting to note that there is a "gap" in the balance of trees.

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

Proof. If T is not completely balanced, i.e., not in BB[1/2], consider a minimal subtree T' of T which is not in BB[1/2]. T' must be of the form (T_ℓ, v, T_r), where both T_ℓ and T_r are in BB[1/2], i.e., $l = 2^s - 1$ and $r = 2^t - 1$, but $s \neq t$ (say $s < t$). Then

$$\rho(T') = \frac{1}{1 + 2^{t-s}} \leqq 1/3,$$

which puts T in BB[α] for some $\alpha \leqq 1/3$.

Search time in BB trees. The height of T_n is a measure of the worst case time required 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 $|T_n|/n$ is a measure of the average time required to search T_n.

The following theorem is due to Nievergelt and Wong (1973).

THEOREM 2. *If T_n is in* BB[α], *then*

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

where[1]

$$H(\alpha) = -\alpha \log \alpha - (1 - \alpha)\log(1 - \alpha).$$

[1] Throughout this paper, all logarithms are taken base 2.

It is not difficult to show the following theorem.

THEOREM 3. *If T_n is in* BB[α], *then the height of T_n is at most*

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

Proof. The proof is by induction on n.

The bounds in both of these theorems are sharp for any tree all of whose subtrees have root-balance α. Since the root-balance of a terminal node is 1/2, the only trees with this property are the completely balanced trees of $n = 2^k - 1$ nodes. It is clear, however, that for trees all of whose root-balances are close to α, these bounds are reasonably tight.

These theorems provide bounds on average and worst case search times for trees in BB[α], 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[α] 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), shown in Fig. 1. In Fig. 1 we have used squares to represent nodes, and triangles to represent subtrees; the root-balance is given beside each node.

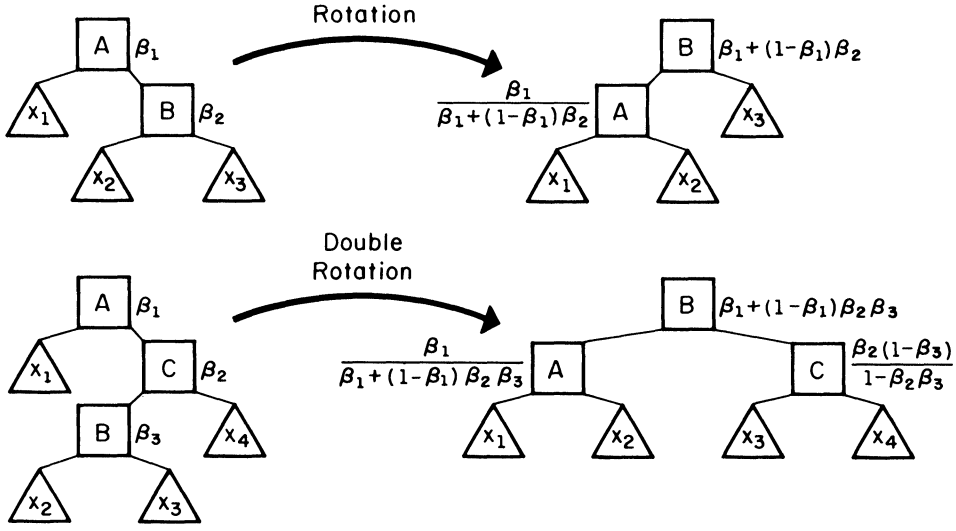


FIG. 1

THEOREM 4. *If* $\alpha \leqq 1 - \sqrt{2}/2$ *and the insertion or deletion of a node in a tree in* BB[α] *causes a subtree T of that tree to have root-balance less than* α, *T can be rebalanced by performing one of the two transformations shown above. More precisely, let* β_2 *denote the balance of the right subtree of T after the insertion or deletion has been done. If* $\beta_2 < (1 - 2\alpha)/(1 - \alpha)$, *then a rotation will rebalance T, otherwise a double rotation will rebalance T.*

Sketch of proof. Under the various hypotheses, it can be shown 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 will probably increase the allowable range of α. However, since $1 - \sqrt{2}/2 \approx .2928$ and BB[α] − BB[1/2] is empty for $1/2 > \alpha > 1/3, \alpha \leqq 1 - \sqrt{2}/2$ 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[$1 - \sqrt{2}/2$] 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 legicographically 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.[2]

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 *not* restructuring the tree: the restructuring which has been done, albeit unnecessarily, has improved the balance of the tree.

---

[2] C. A. Crane and D. E. Knuth have suggested that it may be more useful to have SIZE(N) be one plus the size of the left subtree of node N. This has the effect of simplifying the arithmetic in some algorithms (e.g. in finding the kth data item) while making it slightly more complicated in others. See Crane (1972) and Knuth (1973).

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 N to be deleted, one of three cases arises. If N is a leaf, simply delete it. If N has only one son, link the father of N to the son of N, thus deleting N. Otherwise, find the postorder successor of N (or predecessor, depending which most improves the balance) and promote it to the place of N, taking care to adjust the appropriate SIZE fields and links. 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[α], 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 (1968, p. 391)), the distribution of positions for such a random walk is *uniform* over the interval and hence *this is the assumption we will make.* 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 do not 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, and hence that the expected number of transformations is even smaller than the following theorem indicates.

THEOREM 5. *Under the (weak) assumption that distribution of root-balances in a* BB[α] *tree is uniform over* $[\alpha, 1 - \alpha]$, *the expected number of tree transformations required for insertion or deletion of a node is less than* $2/(1 - 2\alpha)$.

Sketch of proof. If T_n is a tree of n nodes in BB[α] with ℓ and r nodes in its left and right subtrees, respectively, then

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

For simplicity, we shall 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[α]. Thus the probability, p_n,

of causing the root-balance of T_n to go out of the interval $[\alpha, 1 - \alpha]$ is[3]

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

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

Now the expected number of nodes in the two subtrees of a tree in BB[α] 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,

$$p_n + p_{n/2} + p_{n/4} + \cdots + p_{n/n} = \sum_{i=0}^{\log n} \frac{1}{(1 - 2\alpha)n/2^i} = \frac{1}{(1 - 2\alpha)n} \sum_{i=0}^{\log n} 2^i$$

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

This completes the sketch of the proof.

It is remarkable that this bound on the expected number of rebalancings is *independent of the size of the tree*. For example, we find that on the average no more than 4.85 transformations will be necessary to insert or delete a node when the tree is in BB[$1 - \sqrt{2}/2$].

Comparison with height-balanced trees. Height-balanced 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 height-balanced trees. The same two transformations serve to restructure a height-balanced tree which has been upset by an insertion or deletion. Adel'son-Vel'skii and Landis (1962) have shown that one transformation is sufficient to rebalance a height-balanced tree which has been unbalanced by an insertion; the expected number being about 0.3 (see Knuth (1973)). Deletion from height-balanced trees may require rebalancing at each level of the tree, but an argument similar to that in Theorem 5 shows that the expected number of transformations needed is constant.

Height-balanced trees cannot be described as BB[α] for any α.

THEOREM 6. *There are trees in* BB[1/3] *which are not height-balanced and for all* $\varepsilon > 0$ *there is a height-balanced tree with root-balance less than* ε.

Proof. The first part of the theorem is shown by considering a tree such as that in Fig. 2 which is in BB[1/3] but which is not height-balanced. The second part is shown by considering a tree whose left subtree is the Fibonacci tree of height h and whose right subtree is the completely balanced tree of height h. As $h \to \infty$ the balance of such a tree, which is height-balanced, goes to zero. This completes the proof.

[3] This implicitly assumes that the node is inserted in (deleted from) either subtree of T_n with probability 1/2, regardless of the size of these subtrees. If one assumed that the probability of insertion or deletion from a subtree is proportional to the size of this subtree, then this analysis would become less favorable for insertion, and more favorable for deletion.

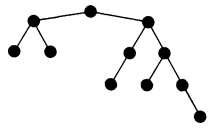


FIG. 2

The search time for height-balanced trees is somewhat better than for BB trees. The worst case search time for height-balanced 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[α] are bounded by Theorems 2 and 3. For example, when $\alpha = 1 - \sqrt{2}/2$, 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 height-balanced 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 (for insertion, the second pass can instead be top-down). Typically, the bottom-up pass is accomplished by the use of a pushdown stack. The use of a stack can be eliminated but only at some cost in time or memory (e.g., every node could store an upward pointer to its father. Alternatively, the return path to the root can be retained by reversing the links on descent, and restoring them on the way back up; an additional bit would be necessary to indicate whether the right link or the left link had been reversed). 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 or its equivalent.

A height-balanced tree requires at least 2 bits of storage for every node to indicate which of the three possible conditions holds between the heights of its two subtrees. By comparison, a node in a BB-tree requires more storage because it has to hold the size of the tree rooted at this node. However, some important benefits compensate for this. Such important operations as finding the kth data element, or the qth quantile, or how many elements there are lexicographically between x and y, can all be done in time $O(\log n)$, while they seem to require time $O(n)$ if the size information is not explicitly stored.

Table 1 summarizes the comparison of random trees (BB[0]), height-balanced trees, BB[$1 - \sqrt{2}/2$] trees, and completely balanced trees (BB[1/2] if we ignore semileaves).

The important advantage BB trees have over height-balanced 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 $1 - \sqrt{2}/2$, while if insertions and deletions are very frequent, α could be chosen closer to zero. It is not easy to generalize height-balanced trees to include a parameter which has the function of α. The obvious choice would be to define a tree T to be of *height-balance* h if and only if for every node N of T, the heights of the two subtrees of N differ by at most h (the special case $h = 1$ yields the conventional height-balanced trees); see Foster (1972). This suffers from the fact that the smallest possible change in h

TABLE 1

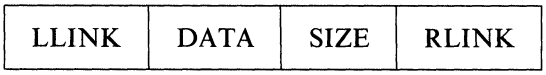
	Number of comparisons needed to search the tree in the worst possible case	Expected number of comparisons needed to search an average tree	Time required to return an unbalanced tree to its class
Random trees of n nodes (BB[0])	n	$1.39 \log (n + 1)$*	0
AVL trees of n nodes	$1.44 \log (n + 1)$	$\log (n + 1) + 0.25$†	$O(\log n)$
BB[$1 - \sqrt{2}/2$] trees of n nodes	$2 \log (n + 1)$	$1.05 \log (n + 1)$†	$O(\log n)$
Completely balanced trees of n nodes (BB[1/2], ignoring semileaves)	$\log (n + 1)$	$\log (n + 1)$	$O(n)$

* Due to Hibbard (1962).
† Based on empirical evidence; the exact bound is unknown.

(say from $h = 1$ to $h = 2$) changes the class of trees very drastically, and thus the compromise between search time and rebalancing time cannot be finely tuned.

Appendix. The insertion algorithm. 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.

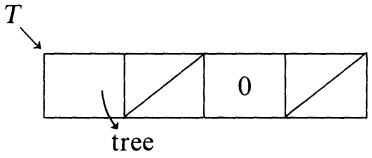
Assume that each node of the tree has the form



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



R is a pointer which will be used in the search through the tree to find out where NEW 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 $S \cdot$ LINK is either LLINK or RLINK according to the value of S. For example,

$$S = \text{"}L\text{"},$$

$$S \cdot \text{LINK}(P) \leftarrow P$$

has the same effect as

$$\text{LLINK}(P) \leftarrow P.$$

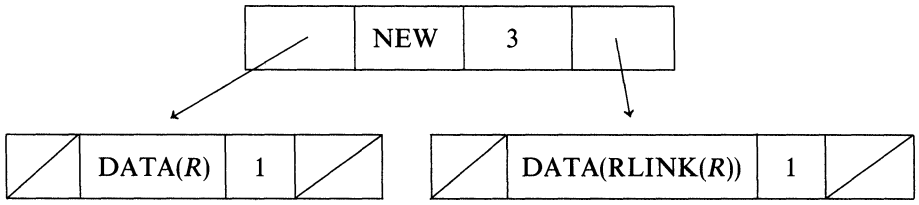
The value of S together with the value of RP tell us which pointer has to be modified when we rebalance a subtree.

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

Step 2 (Small tree?). Set $R = S \cdot \text{LINK}(RP)$; this moves us down one level in the tree. If $1/(\|R\| + 2) \geqq \alpha$ 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.[4]

Step 4 (Rotation no help?). If $\|R\| = 2$, RLINK(R) is not null, and DATA(RLINK(R)) > NEW, then set $S \cdot \text{LINK}(RP)$ 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:

$$v = \frac{\|\text{LLINK}(R)\| + 1}{\|R\| + 2}.$$

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

$$\text{SIZE}(R) \leftarrow \text{SIZE}(R) + 1,$$

$$S \leftarrow \text{"}R\text{"},$$

$$RP \leftarrow R,$$

$$R \leftarrow \text{RLINK}(R)$$

and go to Step 2.

[4] The two cases are *not* handled symmetrically. Step 4 is needed to prevent an infinite loop under the conditions described; if the symmetrical case arises, the algorithm performs a rotation in Step 7 and returns, eventually, to Step 4.

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 $\beta_2 < (1 - 2\alpha)/(1 - \alpha)$, then use rotation, otherwise use double rotation. Set $S \cdot \text{LINK}(RP)$ 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:

$$v = \frac{\|\text{LLINK}(R)\| + 2}{\|R\| + 2}.$$

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

$$\text{SIZE}(R) \leftarrow \text{SIZE}(R) + 1,$$

$$S \leftarrow \text{“}L\text{”},$$

$$RP \leftarrow R,$$

$$R \leftarrow \text{LLINK}(R),$$

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 $1 - \beta_2 < (1 - 2\alpha)/(1 - \alpha)$, then use rotation, otherwise use split-rotation. Set $S \cdot \text{LINK}(RP)$ 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 $R \leftarrow \text{LLINK}(T)$ 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 $\text{SIZE}(R) \leftarrow \text{SIZE}(R) - 1$. Then, if NEW > DATA($R$) set $R \leftarrow \text{RLINK}(R)$, otherwise set $R \leftarrow \text{LLINK}(R)$. Repeat Step 10.

Acknowledgment. We are grateful to C. A. Crane and D. E. Knuth for very helpful comments which were based on an earlier version of this paper.

REFERENCES

G. M. Adel'son-Vel'skii and Ye. M. Landis (1962), *An algorithm for the organization of information*, Dokl. Akad. Nauk SSSR, 146, pp. 263–266 = Soviet Math. Dokl., 3, pp. 1259–1263.

J. Bruno and E. G. Coffman (1972), *Nearly optimal binary search trees*, Information Processing 71, vol. 1, North Holland, Amsterdam, pp. 99–103.

C. A. Crane (1972), *Linear lists and priority queues as balanced binary trees*, Doctoral thesis, Stanford Univ., Stanford, Calif.

W. Feller (1968), *An Introduction to Probability Theory and its Application*, vol. 1, 3rd ed., John Wiley, New York.

C. C. Foster (1965), *Information storage and retrieval using AVL trees*, Proc. ACM 20th National Conference, pp. 192–205.

——— (1972), *A generalization of AVL trees*, Tech. Note TN/CS/00033, Computer and Information Sciences Department, Univ. of Massachusetts, Amherst; Comm. ACM, to appear.

T. HIBBARD (1962), *Some combinatorial properties of certain trees*, J. Assoc. Comput. Mach., 9, pp. 13–28.

T. C. HU AND A. C. TUCKER (1971), *Optimal computer search trees and variable-length alphabetical codes*, SIAM J. Appl. Math., 21, pp. 514–532.

D. E. KNUTH (1971), *Optimum binary search trees*, Acta Informatica, 1, pp. 14–25.

——— (1973), *The Art of Computer Programming. Volume III: Sorting and Searching*, Addison-Wesley, Reading, Mass.

J. NIEVERGELT AND C. K. WONG (1972), *On binary search trees*, Information Processing 71, vol. 1, North Holland, Amsterdam, pp. 91–98.

——— (1973), *Upper bounds for the total path length of binary trees*, J. Assoc. Comput. Mach., 20, pp. 1–6.

W. A. WALKER AND C. C. GOTLIEB (1972), *A top down algorithm for constructing nearly-optimal lexicographic trees*, Graph Theory and Computing, R. C. Read, ed., Academic Press, New York, pp. 303–323.