*International Journal of Computer and Information Sciences, Vol. 9, No. 1, 1980*

# A Method for Constructing Binary Search Trees by Making Insertions at the Root

C. J. Stephenson[1]

*Received December 1976; revised April 1979*

It is possible to construct a binary search tree by inserting items at the root instead of adding them as leaves. When used for sorting, the method has several desirable properties, including (a) fewer comparisons in the best case, (b) fewer comparisons in the worst case, (c) a reduced variance, and (d) good performance when the items are already nearly sorted or nearly reverse sorted. For applications in which the tree is searched for existing items as well as having new items added to it (*e.g.*, in the construction of a symbol table), the tree can be made to exhibit stacklike behavior, so that the fewest comparisons are required to locate the most recently used items.

**KEY WORDS:** Binary search trees; sorting; symbol tables.

## 1. INTRODUCTION

A *binary search tree* (in the sense used here) is a collection of "nodes," each having a "value" and up to two "sons," which are related such that the value of every node exceeds or equals that of its descendants on the left (if it has any) and does not exceed that of its descendants on the right (if it has any). The nodes that comprise a binary search tree are sorted by value, in the sense that a left-to-right traversal of the tree visits them in ascending order. In practice, binary search trees are used for sorting when the number of items is not known in advance, when intermediate searches are to be made (before all the items have been received), or when program simplicity is desired.

[1] IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y,

A binary search tree is normally constructed by adding new nodes one by one as leaves. As the tree grows, the old leaves becomes fathers to new leaves. Given a long sequence of $N$ distinct items, in random order, the expected number of comparisons required to construct the tree approaches $2N \ln N$, which compares well with other sorting methods, and is less than 1.39 times the information theoretic minimum of $N \log_2 N$. However, the method performs badly when the items are already nearly sorted or nearly reverse sorted. These are the worst cases for the algorithm, requiring up to $N(N-1)/2$ comparisons. Various tree-balancing techniques alleviate the degradation, but involve extra bookkeeping and still leave a sorted sequence with a performance no better than that expected with random data. (See reference 8; reference 5; reference 3; and reference 6, pp. 193–194, 427, and 451 *et seq.*)

The method that is presented here constructs a binary search tree by inserting each new item at the root, with the old root and the rest of the tree rearranged around it. As the tree grows, the earlier arrivals percolate down to form the leaves and the intermediate nodes. If there are no duplicate items, an insertion requires the same number of comparisons as if the same item had been added as a leaf to the same tree, but the fact that the tree is constantly being reshaped gives the algorithm different overall characteristics.

## 2. ALGORITHM

The method of root insertion can be described as follows. The item to be inserted in the tree becomes the new root, and a descent is made through the tree, starting at the old root, comparing each node that is visited with the new root. If a node has a value that exceeds that of the new root, then the node (with its descendants on the right) is attached on the right-hand side of the tree, and its left son becomes the next node to be visited; otherwise the node (with its descendants on the left) is attached to the left-hand side of the tree, and its right son becomes the next node to be visited. The points of attachment (on the left and right of the tree) are maintained so that the resulting tree is sorted. The descent stops when the next node to be visited does not exist.

The algorithm is given here, after the corresponding algorithm for leaf insertion, which is included for purposes of comparison. For the meanings of the symbols, and the initial and final conditions, see the notes following the algorithms.

## (A) Leaf insertion

```
if root = null then
    root := new
else
    begin
        node := root;
        while node ≠ null do
            begin
                if value(new) ≥ value(node) then
                    hook := addr(right(node))
                else
                    hook := addr(left(node));
                node := 0(hook)
            end;
        0(hook) := new
    end;

left(new) := null;
right(new) := null
```

## (B) Root insertion

```
node := root;
root := new;
left_hook := addr(left(root));
right_hook := addr(right(root));

while node ≠ null do
    if value(node) > value(root) then
        begin
            0(right_hook) := node;
            right_hook := addr(left(node));
            node := left(node)
        end
    else
        begin
            0(left_hook) := node;
            left_hook := addr(right(node));
            node := right(node)
        end;

0(left_hook) := null;
0(right_hook) := null
```

The following notes apply to these algorithms:

1. Each node contains three fields:

| | |
|---|---|
| left, right | pointers to sons (null if son does not exist); |
| value | value of item to be inserted. |

2. At the beginning of each sequence, the pointer variable "root" contains the address of the existing root node (or is null if the existing tree is empty), and the pointer variable "new" contains the address of the node that is to be inserted (in which the "value" field is already set). At the end, the variable "root" contains the address of the new root node.

3. The function "addr" yields the storage address of its argument. The notation

0 (pointer)

is used to refer to the field in storage addressed by the value of "pointer."

4. In the root-insertion algortihm, as given, an assignment is made to the field addressed by "left_hook" or "right_hook" each time around the loop, and to both fields on exit from the loop, irrespective of whether the fields already contain the required values. About half these assignments are unnecessary, and could be avoided by making a small change to the program. Even then, however, the method of root insertion would in general require more changes to the pointers in the tree than does the method of leaf insertion.

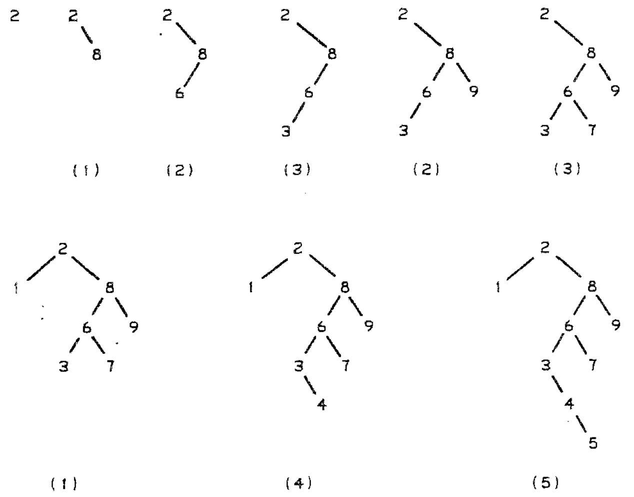
5. In the tree constructed by leaf insertion, any duplicate items are hung on the right of the ancestral nodes that they match, whereas in the tree constructed by root insertion they are hung on the left. This is done deliberately so that duplicates are (in both cases) sorted by order of arrival.

## 3. EXAMPLES

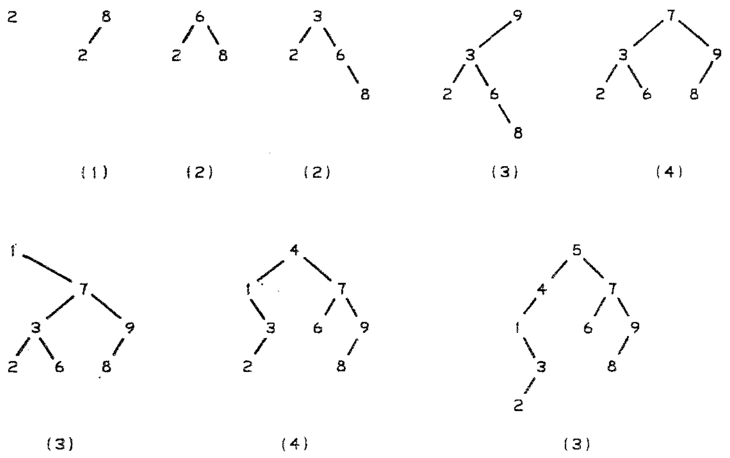
Figure 1 shows the way the trees develop for the sequence 2 8 6 3 9 7 1 4 5. Below each tree is given the number of comparisons required to make the tree from its predecessor. This example has been chosen for purposes of illustration, and the tree that is built by root insertion is restructured more than is typical.

Figure 2 shows the corresponding trees for the nearly sorted sequence 1 2 3 4 6 5 7 8 9. Note that in this case the method of root insertion requires far fewer comparisons.

Figure 3 presents experimentally obtained histograms for the number of comparisons required to sort 1000 random number 100,000 times (see Appendix). The theoretical best and average values are also shown (see Section 4).

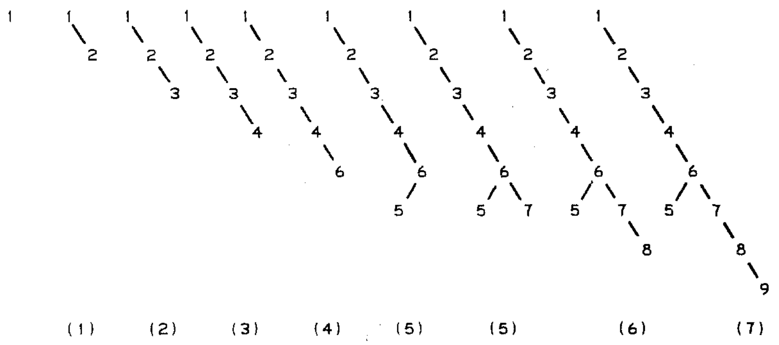


Algorithm (A) – Leaf insertion
(Total number of comparisons = 21)

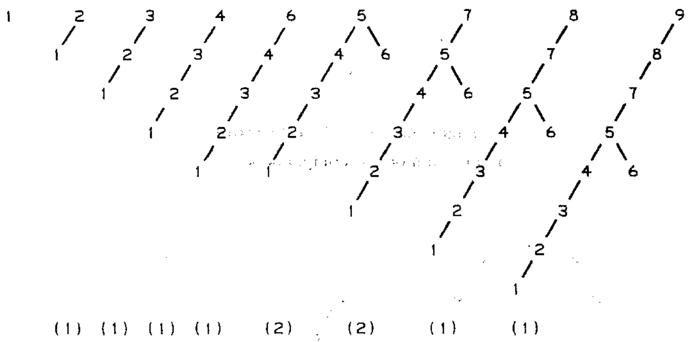


Algorithm (B) – Root insertion
(Total number of comparisons = 22)

Fig. 1. Binary tree construction for the sequence 2 8 6 3 9 7 1 4 5.



Algorithm (A) – Leaf insertion
(Total number of comparisons = 33)



Algorithm (B) – Root insertion
(Total number of comparisons = 10)

Fig. 2. Binary tree construction for the nearly sorted sequence 1 2 3 4 6 5 7 8 9.

## 4. ANALYSIS

(1) If the items are distinct (*i.e.*, there are no duplicates), the number of comparisons required to insert an item at the root is the same as the number of comparisons required to add the same item to the same tree as a leaf.

This follows immediately from the algorithms given in Sec. 2, since the same path through the tree is taken in the two cases.

(2) When a new item is inserted at the root, any existing node can descend at most one level in the tree.

In the root-insertion algorithm, the points of attachment (addressed by "left_hook" and "right_hook") are initialized to the new root and descend at most one level each time around the loop, while the node to be attached (addressed by "node") is initialized to the old root and descends exactly

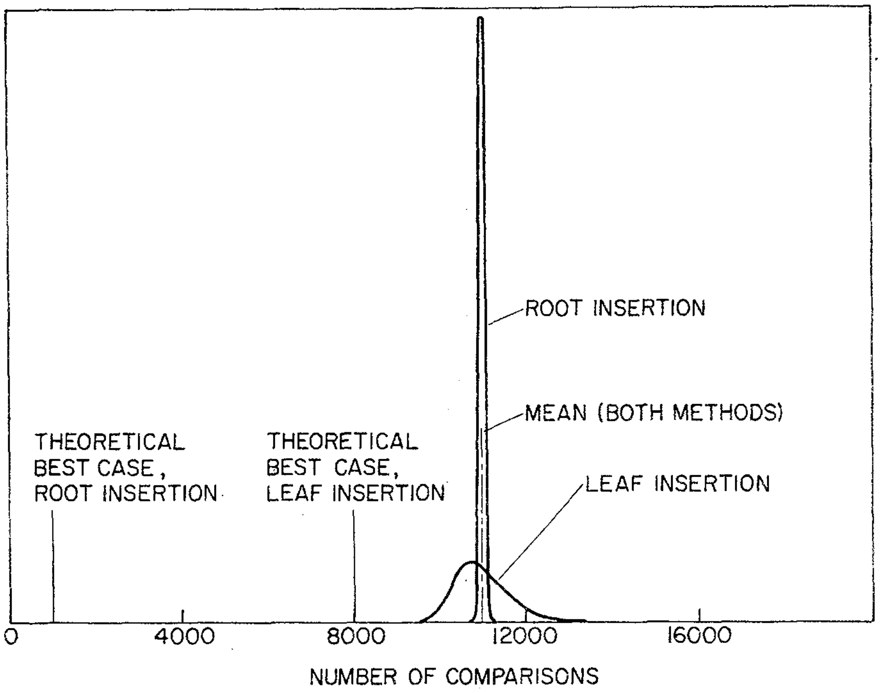


Fig. 3. Histograms for the number of comparisons required to sort 1000 items using leaf insertion (conventional method) and root insertion (new method), showing also the theoretical minimum and average values. Each histogram was obtained by sorting 100,000 sequences of 1000 random numbers.

one level each time around the loop. The old root therefore descends one level, and lower nodes descend at most one level.

(3) In a tree constructed by root insertion, if a left or right pointer field in a node once becomes null, it remains null thereafter, no matter what items are subsequently inserted.

This also follows straightforwardly from the algorithm as given. The left and right pointer fields are set non-null only in the loop, which terminates if they are already null.

*Corollary:* A node that once becomes a leaf remains a leaf.

(4) Consider the tree that is constructed by root insertion from a sequence $y_1 y_2 \cdots y_n$ of distinct items, and also the tree that is constructed by the same method from the same sequence except that after insertion of $y_m$ a leaf containing $y_l$ is removed from the tree ($1 \leqslant l \leqslant m \leqslant n$). The first tree differs from the second tree only in possessing the extra leaf $y_l$.

In the first tree, it follows from (3) that the node $y_l$ remains a leaf after the insertion of $y_m$, and from the root-insertion algorithm the presence of a leaf can affect only its own point of attachment (and not the ancestral connections), since the loop terminates as soon as a leaf has been attached. Therefore, the presence or absence of $y_l$ has no effect on the connections between the other nodes in the tree.

(5) The tree that is constructed by root insertion from any given sequence of distinct items is identical to the tree that is constructed by leaf insertion from the reverse sequence.

Postulate a sequence $S$ that has the property to be proved, and let $r$ be any item not in $S$. Then from (3) and (4) the sequence $rS$ also has the property, since there is a unique position in the tree at which item $r$ can exist as a leaf. By induction, since any sequence of length 1 has the property, all sequences have the property.

(6) In the tree constructed by root insertion from a sequence in which item $p$ precedes item $q$, the node containing $q$ can never become a descendant of the node containing $p$.

This follows directly from (5) and the fact that $q$ precedes $p$ in the reverse sequence.

(7) In a tree constructed by root insertion from a sequence $y_1 y_2 \cdots y_n$ of distinct items in random order, the expected depth in the tree of item $y_r$ is

$$d_r = 2H_{n-r+1} - 2 \tag{1}$$

where the root is reckoned to be at a depth of zero, and

$$H_n \equiv \sum_{1 \leqslant r \leqslant n} \frac{1}{r}$$

From (5), the expected depth of item $y_r$ in the tree constructed by root insertion is the same as the expected depth of item $y_{n-r+1}$ in a tree constructed by leaf insertion. Equation (1) then follows from the expected number of comparisons required for an unsuccessful search in a tree containing $n - r$ items; see, for example, reference 6, p. 427.

(8) *Average number of comparisons.* The average number of comparisons required to construct a tree, from a sequence of $N$ distinct items in random order, is the same for the two algorithms.

Since the sequence is in random order, all possible subsequences of the first $m$ items are equally likely, and in particular any subsequence has the same likelihood as its reverse. Then from (1) and (5) the expected number of comparisons required to insert the $(m + 1)$th item is equal for the two algorithms; hence the average number of comparisons required to insert all $N$ items is also equal, and is known to be

$$C_{\mathrm{av}} = 2(N+1)\,H_N - 4N \tag{2}$$

which for large $N$ approaches the value

$$C_{\mathrm{av}} \approx 2N(\ln N - 2 + \gamma) + \mathrm{O}(\ln N) \tag{3}$$

where $\gamma$ is Euler's constant (approximate value 0.577). (This result can be derived from reference 6, page 427.)

(9) *Best case.* The minimum number of comparisons required to sort $N$ items by leaf insertion occurs when the tree is perfectly balanced, and is

$$C_{\min} = \sum_{1 \leqslant n \leqslant N} \lfloor \log_2 n \rfloor \tag{4}$$

whence, for large $N$,

$$C_{\min} \approx N(\log_2 N - 2) + \mathrm{O}(\log_2 N) \tag{5}$$

The minimum number of comparisons with root insertion is $N - 1$, and occurs when the items are already sorted or reverse sorted (or when they are all the same).

(10) *Worst case.* The maximum number of comparisons required to sort $N$ items by leaf insertion occurs when the items are already sorted or reverse sorted (or when they are all the same), and is $N(N-1)/2$. The maximum number of comparisons required with root insertion is not known for certain, but is also of order $N^2$, and appears to be as follows.

Consider a sequence consisting of two increasing subsequences $y_1 y_2 \cdots y_m$ and $z_1 z_2 \cdots z_{N-m}$ such that $y_1 > z_{N-m}$. This sequence can be made by cutting a sorted deck. The comparisons required to insert the items at the root are as follows:

| Item | Number of Comparisons |
|---|---|
| $y_1$ | 0 |
| $y_2, y_3, \ldots, y_m$ | 1 |
| $z_1$ | $m$ |
| $z_2, z_3, \ldots, z_{N-m}$ | $m+1$ |

Hence the total number of comparisons is

$$\begin{aligned} C &= (m-1) + m + (N-m-1)(m+1) \\ &= -m^2 + Nm + (N-2) \end{aligned} \tag{6}$$

For a given value of $N$, $C$ reaches a maximum when $m = N/2$ ($N$ even) or when $m = (N \pm 1)/2$ ($N$ odd), and is then

$$C = \frac{N^2}{4} + N - K \tag{7}$$

where $K = 2$ ($N$ even) or $9/4$ ($N$ odd).

For $N \leqslant 10$, it has been shown by exhaustive search that the number of comparisons is in fact as given by Eq. (7), and occurs for the "cut decks" described above, for their reflections, and for the closely related sequences in which $y_1$ is exchanged with $z_{N-m}$, and/or $y_m$ is exchanged with $z_1$.

(11) *Variance.* Let $p(N, C)$ be the probability of performing exactly $C$ comparisons when sorting $N$ distinct items that arrive in random order, and let $s^2(N)$ be the variance of the corresponding probability distribution. The variance for leaf insertion is known to be

$$s^2 = 7N^2 - 4(N+1)^2 H_N^{(2)} - 2(N+1) H_N + 13N \tag{8}$$

whence, for large $N$,

$$s^2 \approx \left(7 - \frac{2\pi^2}{3}\right) N^2 + O(N \ln N) \tag{9}$$

Here

$$H_n^{(2)} \equiv \sum_{1 \leqslant r \leqslant n} \frac{1}{r^2}$$

and

$$7 - \frac{2\pi^2}{3} \approx 0.42027$$

(See reference 8; and reference 6, p. 672.)

It is clear from Fig. 3 that the variance for root insertion is less than for leaf insertion. Unfortunately, an exact expression has not been found for it, owing to the difficulty of obtaining a recurrence relation for a tree with a changing shape. Some experimental results are presented in Fig. 4. The points that are circled are either exact, or were derived from a sufficient number of independent tests for there to be *prima facie* confidence that the value of $\log_{10} s$ is known to an accuracy of $\pm 0.01$. The other points were derived from a smaller number of tests, and confidence limits for them were not obtained.

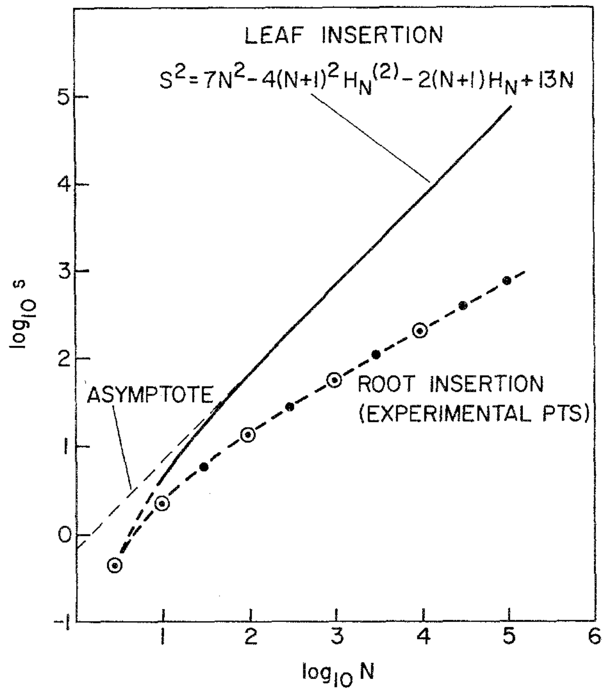


Fig. 4. Variance of number of comparisons required to sort $N$ randomly chosen items, using leaf insertion (conventional method) and root insertion (new method). The values for leaf insertion are exact. The results for root insertion were obtained experimentally: the circled points represent cases for which the value of $\log_{10} s$ is known within $\pm 0.01$; the other points are approximate.

(12) The rearrangement of the tree that accompanies an insertion at the root can be thought of as a process of "splitting" the existing tree, then attaching the two resulting subtrees to the new root. This suggests comparison with the algorithm for splitting balanced trees, given by Crane in reference 3, and summarized in reference 6, on p. 466–467. The algorithms are, however, quite different, and in general the resulting tree is also different. Crane's method restructures the two subtrees, as it creates them, so as to maintain balance. The root insertion algorithm does *not* maintain balance, and in fact some of its desirable properties depend on the *lack* of balance.

(13) For a comparison with a sorting method that uses a set of trees, see references 1 and 2.

Note that the performance analyses given in this section are expressed purely in terms of the *number of comparisons*. It is possible to envisage circumstances in which the cost of modifying a tree pointer might exceed the cost of making a comparison. This would usually hurt the performance of root insertion more than that of leaf insertion.

## 5. AN ARBOREAL STACK

The discussion and analysis given above have been restricted to the situation in which each descent through the tree is associated with the addition of a new node. This is what happens when a tree is used to sort a list of items. A binary search tree can also be used, however, in situations where it has to be searched for existing entries as well as having new entries added to it. An example is the construction of a symbol table for an assembler or compiler: the tree is searched for the name of a variable; if it is already in the tree, the existing entry is used; otherwise a new entry is added. For these applications it is necessary to take account of the work involved in locating an existing entry, as well as in adding a new one.

A tree that is constructed by adding leaves resembles a *list*, in the sense that the first item to be inserted has a depth of 0 (is at the root), the second arrival has a depth of 1 (hangs directly from the root), the third arrival has a depth of 1 or 2, etc. Therefore, the *earliest* arrivals can be located with the fewest comparisons. On the other hand, a tree that is constructed by root insertion resembles a *stack*, since the most recently inserted item has a depth of 0, the penultimate arrival has a depth of 1, the antepenultimate item has a depth of 1 or 2, and in general, from Sec. 4, paragraph (2), the depth of the $r$th item is at most $n - r$, where $n$ is the number of items in the tree. In this case it is the *most recent* arrivals that can be located with the fewest comparisons. [For the expected depth of an item, as a function of its arrival time, see Sec. 4, paragraph (7).]

The method of root insertion can be adapted to combine a search with a conditional insertion in such a way as to leave the matching entry (whether old or new) at the root of the tree. A tree that is manipulated in this way has the property that the items which have been *most recently referred to* are always near the root, irrespective of when they were first encountered. This is desirable when constructing symbol tables, since references to names tend to be clustered.

The method can be described as follows. A name is given which is to be searched for, and inserted if not already present. A temporary root is made, and the existing tree is searched for a match with the given name. So long as a match is not found, the descent through the tree is handled in the same way as for an unconditional insertion, the tree being rearranged as necessary around the new root. If an old entry for the name does not exist, the temporary root is made permanent; otherwise the old entry is promoted to the root, and any descendant nodes are reconnected to the tree without inspection. Thus the number of nodes that are visited is the same as if the same tree had been searched for the same name without any rearrangement, but the fact that the tree is reshaped results in different overall characteristics, with the fewest comparisons being required to locate the most recently used names.

This method shows up to its greatest advantage when the program being assembled or compiled contains many names, most of which are declared at the beginning and not used thereafter, and a few of which are used many times, with clustered references.

The algorithm is given here. See also the notes that follow.

## (C) Root insertion or promotion

```
        node := root;
        left_hook := addr(left(dummy));
        right_hook := addr(right(dummy));

        while node ≠ null do
            if value(node) = name then
                begin
                    0(left_hook) := left(node);
                    0(right_hook) := right(node);
                    root := node;
                    go to bottom
                end;
            if value(node) > name then
                begin
                    0(right_hook) := node;
                    right_hook := addr(left(node));
                    node := left(node)
                end
            else
                begin
                    0(left_hook) := node;
                    left_hook := addr(right(node));
                    node := right(node)
                end;

        0(left_hook) := null;
        0(right_hook) := null;
        root := new_node ( );
        value(root) := name;

    bottom:
        left(root) := left(dummy);
        right(root) := right(dummy)
```

The following notes apply to this algorithm:

1. Each node contains the same three fields described in the corresponding note of Sec. 2. In practice, there would normally be additional fields containing (for example) the attributes associated with the name.

2. At the beginning of the sequence, the pointer variable "root" contains the address of the existing root node (or is null if the existing tree is empty); the pointer variable "dummy" contains the address of a spare node (which is used as a temporary root during the search), and the variable "name" contains the value of the item that is to be inserted or promoted. At the end, the variable "root" contains the address of the new root node, in which the "value" field contains the value given in "name."

3. The function "new_node" obtains storage for a new node and yields its address.

4. Although in the algorithm as written there appear to be two comparisons in the loop, this is a quirk of the Algol-like representation. In practice, most computers require only one comparison to determine the sign of a difference (—, 0 or +).

As noted earlier, the method of root insertion tends to involve more frequent changes to the tree pointers than occurs using conventional leaf insertion. It is possible to envisage circumstances in which the extra cost of making these changes might outweigh the benefit of the stacklike behavior.

## 6. FINAL REMARKS

The method of "root insertion" has several advantages over the conventional method of "leaf insertion."

Root insertion has particular merit when the items arrive nearly sorted or nearly reverse sorted. In these cases, the method wins handsomely in terms of comparisons, and does not necessarily lose in terms of modifications (*cf.* Fig. 2).

As a method for constructing symbol tables, root insertion is most advantageous when there are many names that appear only once, and the references to the remaining names are clustered.

## ACKNOWLEDGMENTS

The writer is indebted to Ashok K. Chandra, for a major contribution to Sec. 4; and to William H. Burge, Marc A. Auslander, and the editor and referees, for some valuable suggestions.

## APPENDIX: GENERATION OF RANDOM NUMBERS

Some initial tests were made with a pseudo-random number generator in which the number $v_{n+1}$ was generated from its predecessor $v_n$ by the rule

$$v_{n+1} = \mathrm{mod}(16807v_n\,,\, 2^{31} - 1)$$

(See references 4 and 7.) It was, however, found that the measured variance for leaf insertion did not agree with the expected values [Eq. (9)]. The generator was then modified to implement the rule

$$v_{n+1} = x \oplus \mathrm{mod}(16807v_n\,,\, 2^{31} - 1)$$

where "$\oplus$" stands for "exclusive-or," and $x$ represents the left-aligned reversal of the 20 low-order bits from the basic System/370 time-of-day clock, the last bit of which ticks every microsecond. This modified generator was found to yield results for leaf insertion which were within 1 % of the expected values, and was used for all the results presented here.

## REFERENCES

1. W. H. Burge, "An Analysis of a Tree Sorting Method and Some Properties of a Set of Trees," *Proceedings of the First USA–Japan Computer Conference* (1972), pp. 372–379.
2. W. H. Burge, "A Correspondence Between Two Sorting Methods," IBM Research Report, RC 6397 (1977).
3. Clark A. Crane, "Linear Lists and Priority Queues as Balanced Binary Trees," Ph.D. Thesis, Stanford University, Computer Science Department Report, STAN-CS-72-259 (1972).
4. Fred G. Gustavson and Werner Liniger, "A fast random number generator with good statistical properties," *Computing* **6**:221–226 (1970).
5. T. N. Hibbard, "Some combinatorial properties of certain trees with applications to searching and sorting," *JACM* **9**(1):13–28 (1962).
6. D. E. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching* (Addison-Wesley, Reading, Massachusetts, 1973).
7. P. A. W. Lewis, A. S. Goodman, and J. M. Miller, "A pseudo-random number generator for the System/360," *IBM Syst. J.* **8**:136–146 (1969).
8. P F. Windley, "Trees, forests and rearranging," *Comput. J.* **3**(2):84–88 (1960).