6.2.3. Balanced Trees

The tree insertion algorithm we have just learned will produce good search trees, when the input data is random, but there is still the annoying possibility that a degenerate tree will occur. Perhaps we could devise an algorithm that keeps the tree optimum at all times; but unfortunately that seems to be very difficult. Another idea is to keep track of the total path length, and to completely reorganize the tree whenever its path length exceeds $5N \lg N$, say. But such an approach might require about $\sqrt{N/2}$ reorganizations as the tree is being built.

A very pretty solution to the problem of maintaining a good search tree was discovered in 1962 by two Russian mathematicians, G. M. Adelson-Velsky and E. M. Landis [*Doklady Akademiia Nauk SSSR* **146** (1962), 263–266; English translation in *Soviet Math.* **3**, 1259–1263]. Their method requires only two extra bits per node, and it never uses more than $O(\log N)$ operations to search the tree or to insert an item. In fact, we shall see that their approach also leads to a general technique that is good for representing arbitrary *linear lists* of length N, so that each of the following operations can be done in only $O(\log N)$ units of time:

i) Find an item having a given key.

ii) Find the kth item, given k.

iii) Insert an item at a specified place.

iv) Delete a specified item.

If we use sequential allocation for linear lists, operations (i) and (ii) are efficient but operations (iii) and (iv) take order N steps; on the other hand, if we use linked allocation, operations (iii) and (iv) are efficient but operations (i) and (ii) take order N steps. A tree representation of linear lists can do *all four* operations in $O(\log N)$ steps. And it is also possible to do other standard operations with comparable efficiency, so that, for example, we can concatenate a list of M elements with a list of N elements in $O\big(\log(M+N)\big)$ steps.

The method for achieving all this involves what we shall call *balanced trees*. (Many authors also call them *AVL trees*, where the AV stands for Adelson-Velsky and the L stands for Landis.) The preceding paragraph is an advertisement for balanced trees, which makes them sound like a universal panacea that makes all other forms of data representation obsolete; but of course we ought to have a balanced attitude about balanced trees! In applications that do not involve all four of the operations above, we may be able to get by with substantially less overhead and simpler programming. Furthermore, there is no advantage to balanced trees unless N is reasonably large; thus if we have an efficient method that takes $64 \lg N$ units of time and an inefficient method that takes $2N$ units of time, we should use the inefficient method unless N is greater than 256. On the other hand, N shouldn't be too large, either; balanced trees are appropriate chiefly for *internal* storage of data, and we shall study better methods for external direct-access files in Section 6.2.4. Since internal memories seem to be getting larger and larger as time goes by, balanced trees are becoming more and more important.

The *height* of a tree is defined to be its maximum level, the length of the longest path from the root to an external node. A binary tree is called *balanced* if the height of the left subtree of every node never differs by more than ± 1 from the height of its right subtree. Figure 20 shows a balanced tree with 17 internal nodes and height 5; the *balance factor* within each node is shown as $+$, $\bullet$, or $-$ according as the right subtree height minus the left subtree height is $+1$, 0, or -1. The Fibonacci tree in Fig. 8 (Section 6.2.1) is another balanced binary tree of height 5, having only 12 internal nodes; most of the balance factors in that tree

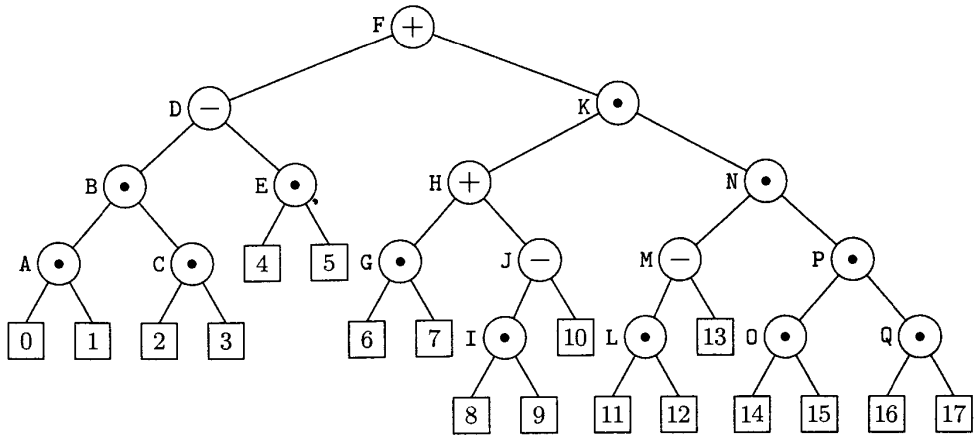


Fig. 20. A balanced binary tree.

are -1. The zodiac tree in Fig. 10 (Section 6.2.2) is *not* balanced, because the height restriction on subtrees fails at both the AQUARIUS and GEMINI nodes.

This definition of balance represents a compromise between *optimum* binary trees (with all external nodes required to be on two adjacent levels) and *arbitrary* binary trees (unrestricted). It is therefore natural to ask how far from optimum a balanced tree can be. The answer is that its search paths will never be more than 45 percent longer than the optimum:

Theorem A (Adelson-Velsky and Landis). *The height of a balanced tree with N internal nodes always lies between* $\lg(N+1)$ *and* $1.4404\lg(N+2) - 0.3277$.

Proof. A binary tree of height h obviously cannot have more than 2^h external nodes; so $N+1 \le 2^h$, that is, $h \ge \lceil \lg(N+1) \rceil$ in any binary tree.

In order to find the maximum value of h, let us turn the problem around and ask for the minimum number of nodes possible in a balanced tree of height h. Let T_h be such a tree with fewest possible nodes; then one of the subtrees of the root, say the left subtree, has height $h-1$, and the other subtree has height $h-1$ or $h-2$. Since we want T_h to have the minimum number of nodes, we may assume that the left subtree of the root is T_{h-1}, and that the right subtree is T_{h-2}. This argument shows that the *Fibonacci tree* of order $h+1$ has the fewest possible nodes among all possible balanced trees of height h. (See the definition of Fibonacci trees in Section 6.2.1.) Thus

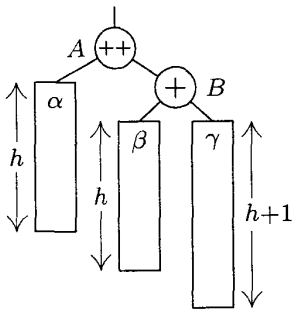
$$N \ge F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2,$$

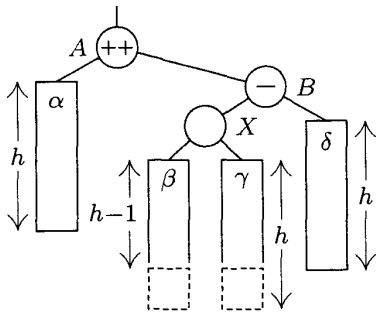
and the stated result follows as in the corollary to Theorem 4.5.3F. ∎

The proof of this theorem shows that a search in a balanced tree will require more than 25 comparisons only if the tree contains at least $F_{28} - 1 = 317{,}810$ nodes.

Consider now what happens when a new node is inserted into a balanced tree using tree insertion (Algorithm 6.2.2T). In Fig. 20, the tree will still be balanced if the new node takes the place of [4], [5], [6], [7], [10], or [13], but

some adjustment will be needed if the new node falls elsewhere. The problem arises when we have a node with a balance factor of $+1$ whose right subtree got higher after the insertion; or, dually, if the balance factor is -1 and the left subtree got higher. It is not difficult to see that trouble arises only in two cases:

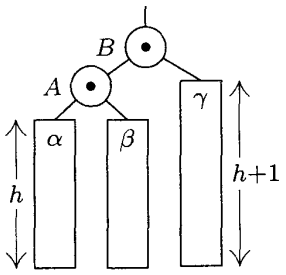
Case 1 

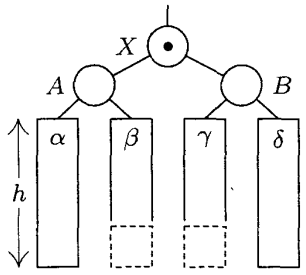
Case 2 

(1)

(Two other essentially identical cases occur if we reflect these diagrams, interchanging left and right.) In these diagrams the large rectangles α, β, γ, δ represent subtrees having the respective heights shown. Case 1 occurs when a new element has just increased the height of node B's right subtree from h to $h+1$, and Case 2 occurs when the new element has increased the height of B's left subtree. In the second case, we have either $h = 0$ (so that X itself was the new node), or else node X has two subtrees of respective heights $(h-1, h)$ or $(h, h-1)$.

Simple transformations will restore balance in both of these cases, while preserving the symmetric order of the tree nodes:

Case 1 

Case 2 

(2)

In Case 1 we simply "rotate" the tree to the left, attaching β to A instead of B. This transformation is like applying the associative law to an algebraic formula, replacing $\alpha(\beta\gamma)$ by $(\alpha\beta)\gamma$. In Case 2 we use a double rotation, first rotating (X, B) right, then (A, X) left. In both cases only a few links of the tree need to be changed. Furthermore, the new trees have height $h+2$, which is exactly the height that was present before the insertion; hence the rest of the tree (if any) that was originally above node A always remains balanced.

For example, if we insert a new node into position [17] of Fig. 20 we obtain the balanced tree shown in Fig. 21, after a single rotation (Case 1). Notice that several of the balance factors have changed.

The details of this insertion procedure can be worked out in several ways. At first glance an auxiliary stack seems to be necessary, in order to keep track of which nodes will be affected, but the following algorithm gains some speed by

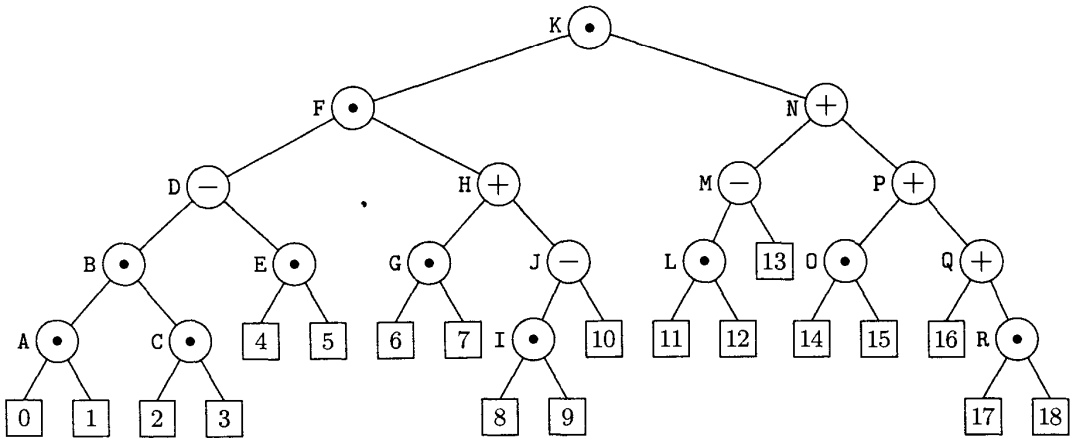


Fig. 21. The tree of Fig. 20, rebalanced after a new key R has been inserted.

exploiting the fact that the balance factor of node B in (1) was zero before the insertion.

You will implement Algorithm A

Algorithm A (*Balanced tree search and insertion*). Given a table of records that form a balanced binary tree as described above, this algorithm searches for a given argument K. If K is not in the table, a new node containing K is inserted into the tree in the appropriate place and the tree is rebalanced if necessary.

The nodes of the tree are assumed to contain KEY, LLINK, and RLINK fields as in Algorithm 6.2.2T. We also have a new field

$$\texttt{B(P)} = \text{balance factor of } \texttt{NODE(P)},$$

the height of the right subtree minus the height of the left subtree; this field always contains either $+1$, 0, or -1. A special header node also appears at the top of the tree, in location HEAD; the value of RLINK(HEAD) is a pointer to the root of the tree, and LLINK(HEAD) is used to keep track of the overall height of the tree. (Knowledge of the height is not really necessary for this algorithm, but it is useful in the concatenation procedure discussed below.) We assume that the tree is *nonempty*, namely that $\texttt{RLINK(HEAD)} \neq \Lambda$.

For convenience in description, the algorithm uses the notation LINK(*a*,P) as a synonym for LLINK(P) if $a = -1$, and for RLINK(P) if $a = +1$.

A1. [Initialize.] Set T $\leftarrow$ HEAD, S $\leftarrow$ P $\leftarrow$ RLINK(HEAD). (The pointer variable P will move down the tree; S will point to the place where rebalancing may be necessary, and T always points to the parent of S.)

A2. [Compare.] If $K <$ KEY(P), go to A3; if $K >$ KEY(P), go to A4; and if $K =$ KEY(P), the search terminates successfully.

A3. [Move left.] Set Q $\leftarrow$ LLINK(P). If Q $= \Lambda$, set Q $\Leftarrow$ AVAIL and LLINK(P) $\leftarrow$ Q and go to step A5. Otherwise if B(Q) $\neq 0$, set T $\leftarrow$ P and S $\leftarrow$ Q. Finally set P $\leftarrow$ Q and return to step A2.

A4. [Move right.] Set Q $\leftarrow$ RLINK(P). If Q $= \Lambda$, set Q $\Leftarrow$ AVAIL and RLINK(P) $\leftarrow$ Q and go to step A5. Otherwise if B(Q) $\neq 0$, set T $\leftarrow$ P and S $\leftarrow$ Q. Finally set

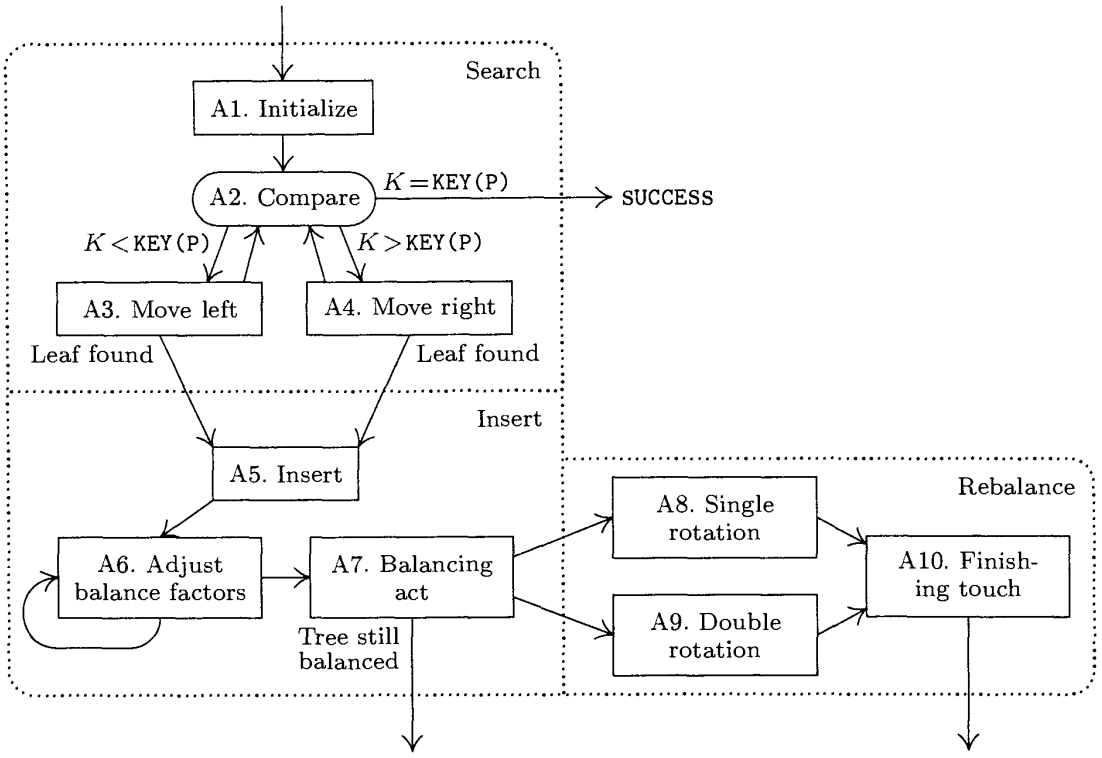


Fig. 22. Balanced tree search and insertion.

P ← Q and return to step A2. (The last part of this step may be combined with the last part of step A3.)

A5. [Insert.] (We have just linked a new node, NODE(Q), into the tree, and its fields need to be initialized.) Set KEY(Q) ← K, LLINK(Q) ← RLINK(Q) ← Λ, and B(Q) ← 0.

A6. [Adjust balance factors.] (Now the balance factors on nodes between S and Q need to be changed from zero to ± 1.) If $K <$ KEY(S) set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. Then set R ← P ← LINK(a,S), and repeatedly do the following operations zero or more times until P = Q: If $K <$ KEY(P) set B(P) ← -1 and P ← LLINK(P); if $K >$ KEY(P), set B(P) ← $+1$ and P ← RLINK(P). (If $K =$ KEY(P), then P = Q and we proceed to the next step.)

A7. [Balancing act.] Several cases now arise:

i) If B(S) = 0 (the tree has grown higher), set B(S) ← a, LLINK(HEAD) ← LLINK(HEAD) + 1, and terminate the algorithm.

ii) If B(S) = $-a$ (the tree has gotten more balanced), set B(S) ← 0 and terminate the algorithm.

iii) If B(S) = a (the tree has gotten out of balance), go to step A8 if B(R) = a, to A9 if B(R) = $-a$.

(Case (iii) corresponds to the situations depicted in (1) when $a = +1$; S and R point, respectively, to nodes A and B, and LINK($-a$,S) points to α, etc.)

A8. [Single rotation.] Set P $\leftarrow$ R, LINK(a,S) $\leftarrow$ LINK($-a$,R), LINK($-a$,R) $\leftarrow$ S, B(S) $\leftarrow$ B(R) $\leftarrow 0$. Go to A10.

A9. [Double rotation.] Set P $\leftarrow$ LINK($-a$,R), LINK($-a$,R) $\leftarrow$ LINK(a,P), LINK(a,P) $\leftarrow$ R, LINK(a,S) $\leftarrow$ LINK($-a$,P), LINK($-a$,P) $\leftarrow$ S. Now set

$$(\dot{B}(S),B(R)) \leftarrow \begin{cases} (-a,0), & \text{if } B(P) = a; \\ (0,0), & \text{if } B(P) = 0; \\ (0,a), & \text{if } B(P) = -a; \end{cases} \tag{3}$$

and then set B(P) $\leftarrow 0$.

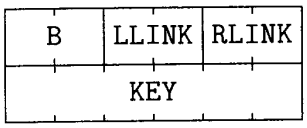
A10. [Finishing touch.] (We have completed the rebalancing transformation, taking (1) to (2), with P pointing to the new subtree root and T pointing to the parent of the old subtree root S.) If S = RLINK(T) then set RLINK(T) $\leftarrow$ P, otherwise set LLINK(T) $\leftarrow$ P. ∎

This algorithm is rather long, but it divides into three simple parts: Steps A1–A4 do the search, steps A5–A7 insert a new node, and steps A8–A10 rebalance the tree if necessary. Essentially the same method can be used if the tree is *threaded* (see exercise 6.2.2–2), since the balancing act never needs to make difficult changes to thread links.

We know that the algorithm takes about $C \log N$ units of time, for some C, but it is important to know the approximate value of C so that we can tell how large N should be in order to make balanced trees worth all the trouble. The following MIX implementation gives some insight into this question.

You need not understand Program A

Program A (*Balanced tree search and insertion*). This program for Algorithm A uses tree nodes having the form



; (4)

rA $\equiv K$, rI1 $\equiv$ P, rI2 $\equiv$ Q, rI3 $\equiv$ R, rI4 $\equiv$ S, rI5 $\equiv$ T. The code for steps A7–A9 is duplicated so that the value of a appears implicitly (not explicitly) in the program.

```
01  B      EQU  0:1
02  LLINK  EQU  2:3
03  RLINK  EQU  4:5
04  START  LDA  K                1      A1. Initialize.
05         ENT5 HEAD               1      T ← HEAD.
06         LD2  0,5(RLINK)         1      Q ← RLINK(HEAD).
07         JMP  2F                 1      To A2 with S ← P ← Q.
08  4H     LD2  0,1(RLINK)         C2     A4. Move right. Q ← RLINK(P).
09         J2Z  5F                 C2     To A5 if Q = Λ.
10  1H     LDX  0,2(B)             C-1    rX ← B(Q).
11         JXZ  *+3                C-1    Jump if B(Q) = 0.
12         ENT5 0,1                D-1    T ← P.
```

```
13  2H    ENT4  0,2              D          S ← Q.
14        ENT1  0,2              C          P ← Q.
15        CMPA  1,1              C          A2. Compare.
16        JG    4B               C          To A4 if K > KEY(P).
17        JE    SUCCESS          C1         Exit if K = KEY(P).
18        LD2   0,1(LLINK)       C1 − S     A3. Move left. Q ← LLINK(P).
19        J2NZ  1B               C1 − S     Jump if Q ≠ Λ.
20  5H    LD2   AVAIL            1 − S      A5. Insert.
21        J2Z   OVERFLOW         1 − S
22        LDX   0,2(RLINK)       1 − S
23        STX   AVAIL            1 − S      Q ⇐ AVAIL.
24        STA   1,2              1 − S      KEY(Q) ← K.
25        STZ   0,2              1 − S      LLINK(Q) ← RLINK(Q) ← Λ.
26        JL    1F               1 − S      Was K < KEY(P)?
27        ST2   0,1(RLINK)       A          RLINK(P) ← Q.
28        JMP   *+2              A
29  1H    ST2   0,1(LLINK)       1 − S − A  LLINK(P) ← Q.
30  6H    CMPA  1,4              1 − S      A6. Adjust balance factors.
31        JL    *+3              1 − S      Jump if K < KEY(S).
32        LD3   0,4(RLINK)       E          R ← RLINK(S).
33        JMP   *+2              E
34        LD3   0,4(LLINK)       1 − S − E  R ← LLINK(S).
35        ENT1  0,3              1 − S      P ← R.
36        ENTX  -1               1 − S      rX ← −1.
37        JMP   1F               1 − S      To comparison loop.
38  4H    JE    7F               F2 + 1 − S To A7 if K = KEY(P).
39        STX   0,1(1:1)         F2         B(P) ← +1 (it was +0).
40        LD1   0,1(RLINK)       F2         P ← RLINK(P).
41  1H    CMPA  1,1              F + 1 − S
42        JGE   4B               F + 1 − S  Jump if K ≥ KEY(P).
43        STX   0,1(B)           F1         B(P) ← −1.
44        LD1   0,1(LLINK)       F1         P ← LLINK(P).
45        JMP   1B               F1         To comparison loop.
46  7H    LD2   0,4(B)           1 − S      A7. Balancing act. rI2 ← B(S).
47        STZ   0,4(B)           1 − S      B(S) ← 0.
48        CMPA  1,4              1 − S
49        JG    A7R              1 − S      To a = +1 routine if K > KEY(S).
50  A7L   J2P   DONE             U1         Exit if rI2 = −a.
51        J2Z   7F               G1 + J1    Jump if B(S) was zero.
52        ENT1  0,3              G1         P ← R.
53        LD2   0,3(B)           G1         rI2 ← B(R).
54        J2N   A8L              G1         To A8 if rI2 = a.
55  A9L   LD1   0,3(RLINK)       H1         A9. Double rotation.
56        LDX   0,1(LLINK)       H1         LINK(a,P ← LINK(−a,R))
57        STX   0,3(RLINK)       H1           → LINK(−a,R).
58        ST3   0,1(LLINK)       H1         LINK(a,P) ← R.
59        LD2   0,1(B)           H1         rI2 ← B(P).
60        LDX   T1,2             H1         −a, 0 or 0
61        STX   0,4(B)           H1           → B(S).
```

```
62          LDX   T2,2              H1      0, 0, or a
63          STX   0,3(B)            H1        → B(R).
64    A8L   LDX   0,1(RLINK)        G1      A8. Single rotation.
65          STX   0,4(LLINK)        G1      LINK(a,S) ← LINK(−a,P).
66          ST4   0,1(RLINK)        G1      LINK(−a,P) ← S.
67          JMP   8F                G1      Join up with the other branch.
68    A7R   J2N   DONE  •           U2      Exit if rI2 = −a.
69          J2Z   6F                G2 + J2 Jump if B(S) was zero.
70          ENT1  0,3               G2      P ← R.
71          LD2   0,3(B)            G2      rI2 ← B(R).
72          J2P   A8R               G2      To A8 if rI2 = a.
73    A9R   LD1   0,3(LLINK)        H2      A9. Double rotation.
74          LDX   0,1(RLINK)        H2      LINK(a,P ← LINK(−a,R))
75          STX   0,3(LLINK)        H2        → LINK(−a,R).
76          ST3   0,1(RLINK)        H2      LINK(a,P) ← R.
77          LD2   0,1(B)            H2      rI2 ← B(P).
78          LDX   T2,2              H2      −a, 0 or 0
79          STX   0,4(B)            H2        → B(S).
80          LDX   T1,2              H2      0, 0, or a
81          STX   0,3(B)            H2        → B(R).
82    A8R   LDX   0,1(LLINK)        G2      A8. Single rotation.
83          STX   0,4(RLINK)        G2      LINK(a,S) ← LINK(−a,P).
84          ST4   0,1(LLINK)        G2      LINK(−a,P) ← S.
85    8H    STZ   0,1(B)            G       B(P) ← 0.
86    A10   CMP4  0,5(RLINK)        G       A10. Finishing touch.
87          JNE   *+3               G       Jump if RLINK(T) ≠ S.
88          ST1   0,5(RLINK)        G3      RLINK(T) ← P.
89          JMP   DONE              G3      Exit.
90          ST1   0,5(LLINK)        G4      LLINK(T) ← P.
91          JMP   DONE              G4      Exit.
92          CON   +1
93    T1    CON   0                         Table for (3).
94    T2    CON   0
95          CON   -1
96    6H    ENTX  +1                J2      rX ← +1.
97    7H    STX   0,4(B)            J       B(S) ← a.
98          LDX   HEAD(LLINK)       J       LLINK(HEAD)
99          INCX  1                 J         + 1
100         STX   HEAD(LLINK)       J           → LLINK(HEAD).
101   DONE  EQU   *                 1 − S   Insertion is complete. ∎
```

Analysis of balanced tree insertion. [Nonmathematical readers, please skip to (10).] In order to figure out the running time of Algorithm A, we would like to know the answers to the following questions:

- How many comparisons are made during the search?
- How far apart will nodes S and Q be? (In other words, how much adjustment is needed in step A6?)
- How often do we need to do a single or double rotation?

It is not difficult to derive upper bounds on the worst case running time, using Theorem A, but of course in practice we want to know the average behavior. No theoretical determination of the average behavior has been successfully completed as yet, since the algorithm appears to be quite complicated, but several interesting theoretical and empirical results have been obtained.

In the first place we can ask about the number B_{nh} of balanced binary trees with n internal nodes and height h. It is not difficult to compute the generating function $B_h(z) = \sum_{n\geq 0} B_{nh} z^n$ for small h, from the relations

$$B_0(z) = 1, \qquad B_1(z) = z, \qquad B_{h+1}(z) = zB_h(z)\big(B_h(z) + 2B_{h-1}(z)\big). \tag{5}$$

(See exercise 6.) Thus

$$\begin{aligned} B_2(z) &= 2z^2 + z^3, \\ B_3(z) &= 4z^4 + 6z^5 + 4z^6 + z^7, \\ B_4(z) &= 16z^7 + 32z^8 + 44z^9 + \cdots + 8z^{14} + z^{15}, \end{aligned}$$

and in general $B_h(z)$ has the form

$$2^{F_{h+1}-1} z^{F_{h+2}-1} + 2^{F_{h+1}-2} L_{h-1} z^{F_{h+2}} + \text{complicated terms} + 2^{h-1} z^{2^h-2} + z^{2^h-1} \tag{6}$$

for $h \geq 3$, where $L_k = F_{k+1} + F_{k-1}$. (This formula generalizes Theorem A.) The total number of balanced trees with height h is $B_h = B_h(1)$, which satisfies the recurrence

$$B_0 = B_1 = 1, \qquad B_{h+1} = B_h^2 + 2B_h B_{h-1}, \tag{7}$$

so that $B_2 = 3$, $B_3 = 3 \cdot 5$, $B_4 = 3^2 \cdot 5 \cdot 7$, $B_5 = 3^3 \cdot 5^2 \cdot 7 \cdot 23$; and, in general,

$$B_h = A_0^{F_h} A_1^{F_{h-1}} \ldots A_{h-1}^{F_1} A_h^{F_0}, \tag{8}$$

where $A_0 = 1$, $A_1 = 3$, $A_2 = 5$, $A_3 = 7$, $A_4 = 23$, $A_5 = 347$, ..., $A_h = A_{h-1}B_{h-2} + 2$. The sequences B_h and A_h grow very rapidly; in fact, they are *doubly exponential*: Exercise 7 shows that there is a real number $\theta \approx 1.43687$ such that

$$B_h = \lfloor \theta^{2^h} \rfloor - \lfloor \theta^{2^{h-1}} \rfloor + \lfloor \theta^{2^{h-2}} \rfloor - \cdots + (-1)^h \lfloor \theta^{2^0} \rfloor. \tag{9}$$

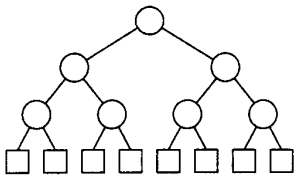
If we consider each of the B_h trees to be equally likely, exercise 8 shows that the average number of nodes in a tree of height h is

$$B_h'(1)/B_h(1) \approx (0.70118)2^h - 1. \tag{10}$$

This indicates that the height of a balanced tree with N nodes is usually much closer to $\log_2 N$ than to $\log_\phi N$.

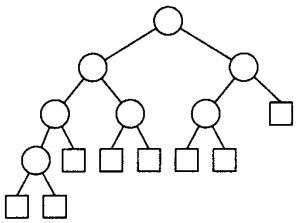
Unfortunately, these results don't really have much to do with Algorithm A, since the mechanism of that algorithm makes some trees significantly more probable than others. For example, consider the case $N = 7$, where 17 balanced trees are possible. There are $7! = 5040$ possible orderings in which seven keys

can be inserted, and the perfectly balanced "complete" tree



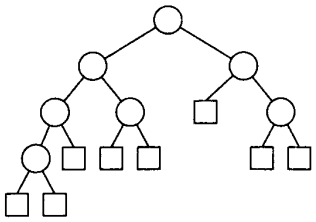
(11)

is obtained 2160 times. By contrast, the Fibonacci tree



(12)

occurs only 144 times, and the similar tree



(13)

occurs 216 times. Replacing the left subtrees of (12) and (13) by arbitrary four-node balanced trees, and then reflecting left and right, yields 16 different trees; the eight generated from (12) each occur 144 times, and those generated from (13) each occur 216 times. It is surprising that (13) is more common than (12).

The fact that the perfectly balanced tree is obtained with such high probability — together with (10), which corresponds to the case of equal probabilities — makes it plausible that the average search time for a balanced tree should be about $\lg N + c$ comparisons for some small constant c. But R. W. Floyd has observed that the coefficient of $\lg N$ is unlikely to be exactly 1, because the root of the tree would then be near the median, and the roots of its two subtrees would be near the quartiles; then single and double rotation could not easily keep the root near the median. Empirical tests indicate that the true average number of comparisons needed to insert the Nth item is approximately $1.01 \lg N + 0.1$, except when N is small.

In order to study the behavior of the insertion and rebalancing phases of Algorithm A, we can classify the external nodes of balanced trees as shown in Fig. 23. The path leading up from an external node can be specified by a sequence of +'s and -'s (+ for a right link, - for a left link); we write down the link specifications until reaching the first node with a nonzero balance factor, or until reaching the root, if there is no such node. Then we write `A` or `B` according as the new tree will be balanced or unbalanced when an internal node is inserted in the given place. Thus the path up from $\boxed{3}$ is `++-B`, meaning "right link, right link, left link, unbalance." A specification ending in `A` requires

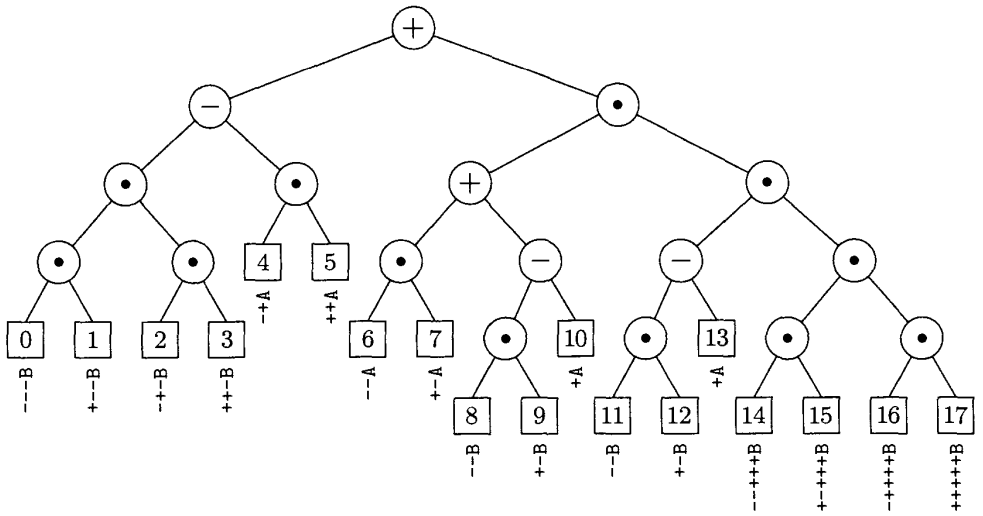


Fig. 23. Classification codes that specify the behavior of Algorithm A after insertion.

no rebalancing after insertion of a new node; a specification ending in `++B` or `--B` requires a single rotation; and a specification ending in `+-B` or `-+B` requires a double rotation. When k links appear in the specification, step A6 has to adjust exactly $k-1$ balance factors. Thus the specifications give the essential facts that govern the running time of steps A6 to A10.

Empirical tests on random numbers for $100 \le N \le 2000$ gave the approximate probabilities shown in Table 1 for paths of various types; apparently these probabilities rapidly approach limiting values as $N \to \infty$. Table 2 gives the exact probabilities corresponding to Table 1 when $N = 10$, considering the 10! permutations of the input as equally probable. (The probabilities that show up as .143 in Table 1 are actually equal to $1/7$, for all $N \ge 7$; see exercise 11. Single and double rotations are equally likely when $N \le 15$, but double rotations occur slightly less often when $N \ge 16$.)

Table 1

APPROXIMATE PROBABILITIES FOR INSERTING THE NTH ITEM

Path length k	No rebalancing	Single rotation	Double rotation
1	.143	.000	.000
2	.152	.143	.143
3	.092	.048	.048
4	.060	.024	.024
5	.036	.010	.010
> 5	.051	.009	.008
ave 2.78	total .534	.233	.232

From Table 1 we can see that k is ≤ 2 with probability about $.143 + .153 + .143 + .143 = .582$; thus, step A6 is quite simple almost 60 percent of the time. The average number of balance factors changed from 0 to ± 1 in that step is

Table 2

EXACT PROBABILITIES FOR INSERTING THE 10TH ITEM

Path length k	No rebalancing	Single rotation	Double rotation
1	1/7	0	0
2	6/35	1/7	1/7
3	4/21	2/35	2/35
4	0	1/21	1/21
ave	247/105	53/105	26/105	26/105

about 1.8. The average number of balanced factors changed from ± 1 to 0 in steps A7 through A10 is approximately $.534+2(.233+.232) \approx 1.5$; thus, inserting one new node adds about $1.8 - 1.5 = 0.3$ unbalanced nodes, on the average. This agrees with the fact that about 68 percent of all nodes were found to be balanced in random trees built by Algorithm A.

An approximate model of the behavior of Algorithm A has been proposed by C. C. Foster [*Proc. ACM Nat. Conf.* **20** (1965), 192–205.] This model is not rigorously accurate, but it is close enough to the truth to give some insight. Let us assume that p is the probability that the balance factor of a given node in a large tree built by Algorithm A is 0; then the balance factor is $+1$ with probability $\frac{1}{2}(1-p)$, and it is -1 with the same probability $\frac{1}{2}(1-p)$. Let us assume further (without justification) that the balance factors of all nodes are independent. Then the probability that step A6 sets exactly $k-1$ balance factors nonzero is $p^{k-1}(1-p)$, so the average value of k is $1/(1-p)$. The probability that we need to rotate part of the tree is $q \approx \frac{1}{2}$. Inserting a new node should increase the number of balanced nodes by p, on the average; this number is actually increased by 1 in step A5, by $-p/(1-p)$ in step A6, by q in step A7, and by $2q$ in step A8 or A9, so we should have

$$p = 1 - p/(1-p) + 3q \approx 5/2 - p/(1-p).$$

Solving for p yields fair agreement with Table 1:

$$p \approx \frac{9-\sqrt{41}}{4} \approx 0.649; \qquad 1/(1-p) \approx 2.851. \tag{14}$$

The running time of the search phase of Program A (lines 01–19) is

$$10C + C1 + 2D + 2 - 3S, \tag{15}$$

where C, $C1$, S are the same as in previous algorithms of this chapter and D is the number of unbalanced nodes encountered on the search path. Empirical tests show that we may take $D \approx \frac{1}{3}C$, $C1 \approx \frac{1}{2}(C+S)$, $C+S \approx 1.01 \lg N + 0.1$, so the average search time is approximately $11.3 \lg N + 3.3 - 13.7S$ units. (If searching is done much more often than insertion, we could of course use a separate, faster program for searching, since it would be unnecessary to look at the balance factors; the average running time for a successful search would then be only about $(6.6 \lg N - 3.4)u$, and the worst case running time would in fact be better than the average running time obtained with Program 6.2.2T.)

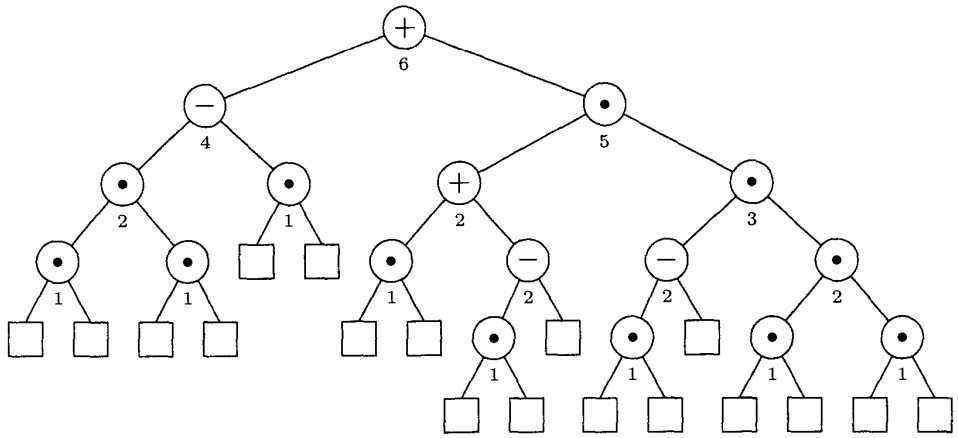


Fig. 24. RANK fields, used for searching by position.

The running time of the insertion phase of Program A (lines 20–45) is $8F + 26 + (0, 1, \text{or } 2)$ units, when the search is unsuccessful. The data of Table 1 indicate that $F \approx 1.8$ on the average. The rebalancing phase (lines 46–101) takes either 16.5, 8, 27.5, or 45.5 (± 0.5) units, depending on whether we increase the total height, or simply exit without rebalancing, or do a single or double rotation. The first case almost never occurs, and the others occur with the approximate probabilities .534, .233, .232, so the average running time of the combined insertion-rebalancing portion of Program A is about $63u$.

These figures indicate that maintenance of a balanced tree in memory is reasonably fast, even though the program is rather lengthy. If the input data are random, the simple tree insertion algorithm of Section 5.2.2 is roughly $50u$ faster per insertion; but the balanced tree algorithm is guaranteed to be reliable even with nonrandom input data.

One way to compare Program A with Program 6.2.2T is to consider the worst case of the latter. If we study the amount of time necessary to insert N keys in increasing order into an initially empty tree, it turns out that Program A is slower for $N \leq 26$ and faster for $N \geq 27$.

Linear list representation. Now let us return to the claim made at the beginning of this section, that balanced trees can be used to represent linear lists in such a way that we can insert items rapidly (overcoming the difficulty of sequential allocation), yet we can also perform random accesses to list items (overcoming the difficulty of linked allocation).

The idea is to introduce a new field in each node, called the RANK field. The field indicates the relative position of that node in its subtree, namely one plus the number of nodes in its left subtree. Figure 24 shows the RANK values for the binary tree of Fig. 23. We can eliminate the KEY field entirely; or, if desired, we can have both KEY and RANK fields, so that it is possible to retrieve items either by their key value or by their relative position in the list.

Using such a RANK field, retrieval by position is a straightforward modification of the search algorithms we have been studying.

You will also implement Algorithm B

Algorithm B (*Tree search by position*). Given a linear list represented as a binary tree, this algorithm finds the kth element of the list (the kth node of the tree in symmetric order), given k. The binary tree is assumed to have LLINK and RLINK fields and a header as in Algorithm A, plus a RANK field as described above.

B1. [Initialize.] Set M $\leftarrow k$, P $\leftarrow$ RLINK(HEAD).

B2. [Compare.] If P $= \Lambda$, the algorithm terminates unsuccessfully. (This can happen only if k was greater than the number of nodes in the tree, or $k \leq 0$.) Otherwise if M $<$ RANK(P), go to B3; if M $>$ RANK(P), go to B4; and if M $=$ RANK(P), the algorithm terminates successfully (P points to the kth node).

B3. [Move left.] Set P $\leftarrow$ LLINK(P) and return to B2.

B4. [Move right.] Set M $\leftarrow$ M $-$ RANK(P) and P $\leftarrow$ RLINK(P) and return to B2. ▮

The only new point of interest in this algorithm is the manipulation of M in step B4. We can modify the insertion procedure in a similar way, although the details are somewhat trickier:

Algorithm C (*Balanced tree insertion by position*). Given a linear list represented as a balanced binary tree, this algorithm inserts a new node just before the kth element of the list, given k and a pointer Q to the new node. If $k = N+1$, the new node is inserted just after the last element of the list.

The binary tree is assumed to be nonempty and to have LLINK, RLINK and B fields and a header, as in Algorithm A, plus a RANK field as described above. This algorithm is merely a transcription of Algorithm A; the difference is that it uses and updates the RANK fields instead of the KEY fields.

C1. [Initialize.] Set T $\leftarrow$ HEAD, S $\leftarrow$ P $\leftarrow$ RLINK(HEAD), U $\leftarrow$ M $\leftarrow k$.

C2. [Compare.] If M $\leq$ RANK(P), go to C3, otherwise go to C4.

C3. [Move left.] Set RANK(P) $\leftarrow$ RANK(P) $+ 1$ (we will be inserting a new node to the left of P). Set R $\leftarrow$ LLINK(P). If R $= \Lambda$, set LLINK(P) $\leftarrow$ Q and go to C5. Otherwise if B(R) $\neq 0$ set T $\leftarrow$ P, S $\leftarrow$ R, and U $\leftarrow$ M. Finally set P $\leftarrow$ R and return to C2.

C4. [Move right.] Set M $\leftarrow$ M $-$ RANK(P), and R $\leftarrow$ RLINK(P). If R $= \Lambda$, set RLINK(P) $\leftarrow$ Q and go to C5. Otherwise if B(R) $\neq 0$ set T $\leftarrow$ P, S $\leftarrow$ R, and U $\leftarrow$ M. Finally set P $\leftarrow$ R and return to C2.

C5. [Insert.] Set RANK(Q) $\leftarrow 1$, LLINK(Q) $\leftarrow$ RLINK(Q) $\leftarrow \Lambda$, B(Q) $\leftarrow 0$.

C6. [Adjust balance factors.] Set M $\leftarrow$ U. (This restores the former value of M when P was S; all RANK fields are now properly set.) If M $<$ RANK(S), set R $\leftarrow$ P $\leftarrow$ LLINK(S) and $a \leftarrow -1$; otherwise set R $\leftarrow$ P $\leftarrow$ RLINK(S), $a \leftarrow +1$, and M $\leftarrow$ M $-$ RANK(S). Then repeatedly do the following operations until P $=$ Q: If M $<$ RANK(P), set B(P) $\leftarrow -1$ and P $\leftarrow$ LLINK(P); if M $>$ RANK(P), set B(P) $\leftarrow +1$ and M $\leftarrow$ M $-$ RANK(P) and P $\leftarrow$ RLINK(P). (If M $=$ RANK(P), then P $=$ Q and we proceed to the next step.)

C7. [Balancing act.] Several cases now arise.