$$H_n = g + \ln n + 1/12n^2 + ...$$

we derive, for large n, the approximate value

$$a_n = 2 * (\ln n + g - 1)$$

Since the average path length in the perfectly balanced tree is approximately

$$a_n' = \log n - 1$$

we obtain, neglecting the constant terms which become insignificant for large n,

$$\lim (a_n/a_n') = 2 * \ln(n) / \log(n) = 2 \ln(2) = 1.386...$$

What does this result teach us? It tells us that by taking the pains of always constructing a perfectly balanced tree instead of the random tree, we could -- always provided that all keys are looked up with equal probability -- expect an average improvement in the search path length of at most 39%. Emphasis is to be put on the word average, for the improvement may of course be very much greater in the unhappy case in which the generated tree had completely degenerated into a list, which, however, is very unlikely to occur. In this connection it is noteworthy that the expected average path length of the random tree grows also strictly logarithmically with the number of its nodes, even though the worst case path length grows linearly.

The figure of 39% imposes a limit on the amount of additional effort that may be spent profitably on any kind of reorganization of the tree's structure upon insertion of keys. Naturally, the ratio between the frequencies of access (retrieval) of nodes (information) and of insertion (update) significantly influences the payoff limits of any such undertaking. The higher this ratio, the higher is the payoff of a reorganization procedure. The 39% figure is low enough that in most applications improvements of the straight tree insertion algorithm do not pay off unless the number of nodes and the access vs. insertion ratio are large.

4.5. Balanced Trees

From the preceding discussion it is clear that an insertion procedure that always restores the trees' structure to perfect balance has hardly any chance of being profitable, because the restoration of perfect balance after a random insertion is a fairly intricate operation. Possible improvements lie in the formulation of less strict definitions of balance. Such imperfect balance criteria should lead to simpler tree reorganization procedures at the cost of only a slight deterioration of average search performance. One such definition of balance has been postulated by Adelson-Velskii and Landis [4-1]. The balance criterion is the following:

A tree is *balanced* if and only if for every node the heights of its two subtrees differ by at most 1.

Trees satisfying this condition are often called AVL-trees (after their inventors). We shall simply call them balanced trees because this balance criterion appears a most suitable one. (Note that all perfectly balanced trees are also AVL-balanced.)

The definition is not only simple, but it also leads to a manageable rebalancing procedure and an average search path length practically identical to that of the perfectly balanced tree. The following operations can be performed on balanced trees in O(log n) units of time, even in the worst case:

1. Locate a node with a given key.
2. Insert a node with a given key.
3. Delete the node with a given key.

These statements are direct consequences of a theorem proved by Adelson-Velskii and Landis, which guarantees that a balanced tree will never be more than 45% higher than its perfectly balanced counterpart, no matter how many nodes there are. If we denote the height of a balanced tree with n nodes by $h_b(n)$, then

$$\log(n+1) < h_b(n) < 1.4404*\log(n+2) - 0.328$$

The optimum is of course reached if the tree is perfectly balanced for n = 2k-1. But which is the structure of the worst AVL-balanced tree? In order to find the maximum height h of all balanced trees with n nodes, let us consider a fixed height h and try to construct the balanced tree with the minimum number of nodes. This strategy is recommended because, as in the case of the minimal height, the value can be attained only for certain specific values of *n*. Let this tree of height *h* be denoted by T_h. Clearly, T0 is the empty tree, and T1 is the tree with a single node. In order to construct the tree T_h for h > 1, we will provide the root with two subtrees which again have a minimal number of nodes. Hence, the subtrees are also T's. Evidently, one subtree must have height *h-1*, and the other is then allowed to have a height of one less, i.e. *h-2*. Figure 4.30 shows the trees with height 2, 3, and 4. Since their composition principle very strongly resembles that of Fibonacci numbers, they are called *Fibonacci-trees* (see Fig. 4.30). They are defined as follows:

1. The empty tree is the Fibonacci-tree of height 0.
2. A single node is the Fibonacci-tree of height 1.
3. If T_{h-1} and T_{h-2} are Fibonacci-trees of heights h-1 and h-2, then $T_h = <T_{h-1}, x, T_{h-2}>$ is a Fibonacci-tree.
4. No other trees are Fibonacci-trees.

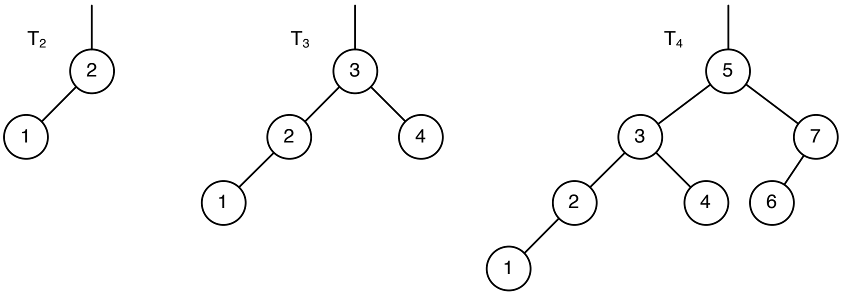

Fig. 4.30. Fibonacci-trees of height 2, 3, and 4

The number of nodes of T_h is defined by the following simple recurrence relation:

$$N_0 = 0,\ N_1 = 1$$
$$N_h = N_{h-1} + 1 + N_{h-2}$$

The N_i are those numbers of nodes for which the worst case (upper limit of h) can be attained, and they are called *Leonardo numbers*.

4.5.1. Balanced Tree Insertion

Let us now consider what may happen when a new node is inserted in a balanced tree. Given a root r with the left and right subtrees L and R, three cases must be distinguished. Assume that the new node is inserted in L causing its height to increase by 1:

1. $h_L = h_R$: L and R become of unequal height, but the balance criterion is not violated.
2. $h_L < h_R$: L and R obtain equal height, i.e., the balance has even been improved.
3. $h_L > h_R$: the balance criterion is violated, and the tree must be restructured.

Consider the tree in Fig. 4.31. Nodes with keys 9 and 11 may be inserted without rebalancing; the tree with root 10 will become one-sided (case 1); the one with root 8 will improve its balance (case 2). Insertion of nodes 1, 3, 5, or 7, however, requires subsequent rebalancing.

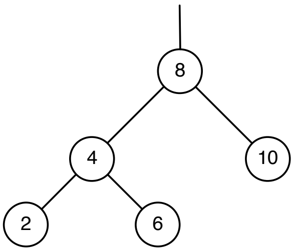

Fig. 4.31. Balanced tree

Some careful scrutiny of the situation reveals that there are only two essentially different constellations needing individual treatment. The remaining ones can be derived by symmetry considerations from those two. Case 1 is characterized by inserting keys 1 or 3 in the tree of Fig. 4.31, case 2 by inserting nodes 5 or 7.

The two cases are generalized in Fig. 4.32 in which rectangular boxes denote subtrees, and the height added by the insertion is indicated by crosses. Simple transformations of the two structures restore the desired balance. Their result is shown in Fig. 4.33; note that the only movements allowed are those occurring in the vertical direction, whereas the relative horizontal positions of the shown nodes and subtrees must remain unchanged.

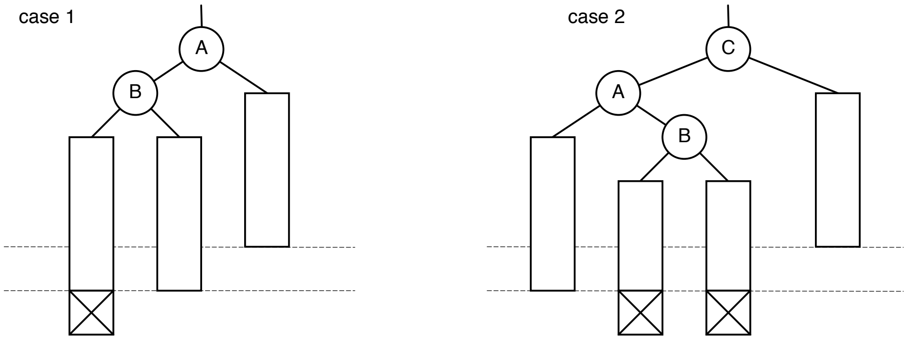

Fig. 4.32. Imbalance resulting from insertion

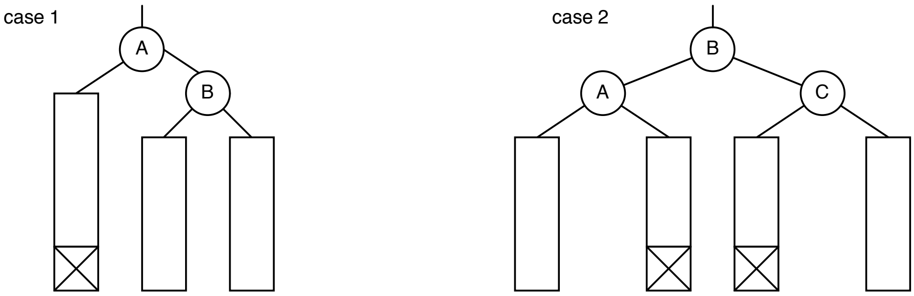

Fig. 4.33. Restoring the balance

An algorithm for insertion and rebalancing critically depends on the way information about the tree's balance is stored. An extreme solution lies in keeping balance information entirely implicit in the tree structure itself. In this case, however, a node's balance factor must be rediscovered each time it is affected by an insertion, resulting in an excessively high overhead. The other extreme is to attribute an explicitly stored balance factor to every node. The definition of the type Node is then extended into

```
TYPE Node = POINTER TO RECORD
                key, count, bal: INTEGER;  (*bal = -1, 0, +1*)
                left, right: Node
            END
```

We shall subsequently interpret a node's balance factor as the height of its right subtree minus the height of its left subtree, and we shall base the resulting algorithm on this node type. The process of node insertion consists essentially of the following three consecutive parts:

1. Follow the search path until it is verified that the key is not already in the tree.
2. Insert the new node and determine the resulting balance factor.
3. Retreat along the search path and check the balance factor at each node. Rebalance if necessary.

Although this method involves some redundant checking (once balance is established, it need not be checked on that node's ancestors), we shall first adhere to this evidently correct schema because it can be implemented through a pure extension of the already established search and insertion procedures. This procedure describes the search operation needed at each single node, and because of its recursive formulation it can easily accommodate an additional operation on the way back along the search path. At each step, information must be passed as to whether or not the height of the subtree (in which the insertion had been performed) had increased. We therefore extend the procedure's parameter list by the Boolean h with the meaning the subtree height has increased. Clearly, h must denote a variable parameter since it is used to transmit a result.

Assume now that the process is returning to a node p^ from the left branch (see Fig. 4.32), with the indication that it has increased its height. We now must distinguish between the three conditions involving the subtree heights prior to insertion:

1. $h_L < h_R$, p.bal = +1, the previous imbalance at p has been equilibrated.
2. $h_L = h_R$, p.bal = 0, the weight is now slanted to the left.
3. $h_L > h_R$, p.bal = -1, rebalancing is necessary.

In the third case, inspection of the balance factor of the root of the left subtree (say, p1.bal) determines whether case 1 or case 2 of Fig. 4.32 is present. If that node has also a higher left than right subtree, then we have to deal with case 1, otherwise with case 2. (Convince yourself that a left subtree with a balance factor equal to 0 at its root cannot occur in this case.) The rebalancing operations necessary are entirely expressed as sequences of pointer reassignments. In fact, pointers are cyclically exchanged, resulting in either a single or a double rotation of the two or three nodes involved. In addition to pointer rotation, the respective node balance factors have to be updated. The details are shown in the search, insertion, and rebalancing procedures.

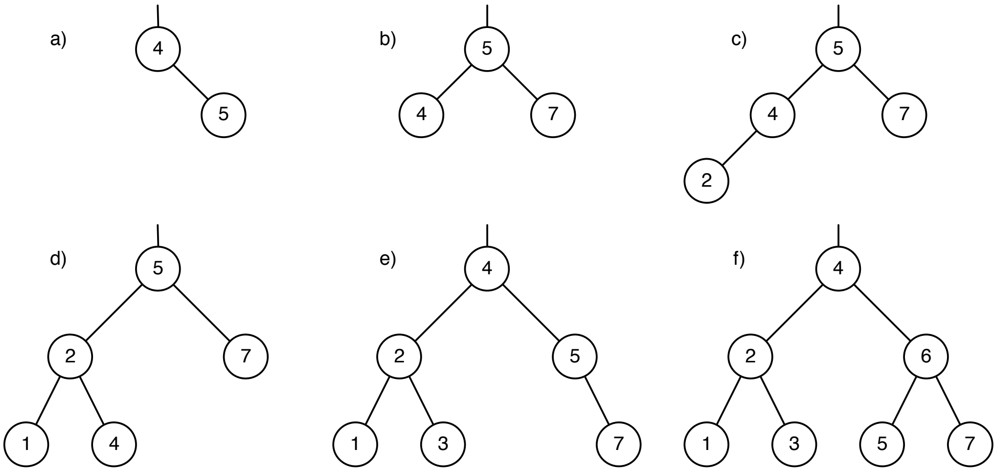

Fig. 4.34. Insertions in balanced tree

The working principle is shown by Fig. 4.34. Consider the binary tree (a) which consists of two nodes only. Insertion of key 7 first results in an unbalanced tree (i.e., a linear list). Its balancing involves a RR single rotation, resulting in the perfectly balanced tree (b). Further insertion of nodes 2 and 1 result in an imbalance of the subtree with root 4. This subtree is balanced by an LL single rotation (d). The subsequent insertion of key 3 immediately offsets the balance criterion at the root node 5. Balance is thereafter reestablished by the more complicated LR double rotation; the outcome is tree (e). The only candidate for losing balance after a next insertion is node 5. Indeed, insertion of node 6 must invoke the fourth case of rebalancing outlined below, the RL double rotation. The final tree is shown in Fig.4.34 (f).

```
PROCEDURE insert(x: INTEGER; VAR p: Node; VAR h: BOOLEAN);
  VAR p1, p2: Node; (*~h*)
BEGIN
  IF p = NIL THEN (*insert*)
    NEW(p); h := TRUE;
    p.key := x; p.count := 1; p.left := NIL; p.right := NIL; p.bal := 0
  ELSIF p.key > x THEN
    search(x, p.left, h);
    IF h THEN (*left branch has grown*)
      IF p.bal = 1 THEN p.bal := 0; h := FALSE
      ELSIF p.bal = 0 THEN p.bal := -1
      ELSE (*bal = -1, rebalance*) p1 := p.left;
        IF p1.bal = -1 THEN (*single LL rotation*)
          p.left := p1.right; p1.right := p;
          p.bal := 0; p := p1
        ELSE (*double LR rotation*) p2 := p1.right;
          p1.right := p2.left; p2.left := p1;
          p.left := p2.right; p2.right := p;
          IF p2.bal = -1 THEN p.bal := 1 ELSE p.bal := 0 END ;
          IF p2.bal = +1 THEN p1.bal := -1 ELSE p1.bal := 0 END ;
          p := p2
        END ;
        p.bal := 0; h := FALSE
      END
    END
  ELSIF p.key < x THEN
    search(x, p.right, h);
```

```
      IF h THEN  (*right branch has grown*)
       IF p.bal = -1 THEN p.bal := 0; h := FALSE
       ELSIF p.bal = 0 THEN p.bal := 1
       ELSE (*bal = +1, rebalance*) p1 := p.right;
           IF p1.bal = 1 THEN  (*single RR rotation*)
             p.right := p1.left; p1.left := p;
             p.bal := 0; p := p1
           ELSE (*double RL rotation*) p2 := p1.left;
             p1.left := p2.right; p2.right := p1;
             p.right := p2.left; p2.left := p;
             IF p2.bal = +1 THEN p.bal := -1 ELSE p.bal := 0 END ;
             IF p2.bal = -1 THEN p1.bal := 1 ELSE p1.bal := 0 END ;
             p := p2
           END ;
           p.bal := 0; h := FALSE
       END
      END
    ELSE INC(p.count)
    END
  END search
```

Two particularly interesting questions concerning the performance of the balanced tree insertion algorithm are the following:

1. If all n! permutations of n keys occur with equal probability, what is the expected height of the constructed balanced tree?
2. What is the probability that an insertion requires rebalancing?

Mathematical analysis of this complicated algorithm is still an open problem. Empirical tests support the conjecture that the expected height of the balanced tree thus generated is h = log(n)+c, where c is a small constant (c ≈ 0.25). This means that in practice the AVL-balanced tree behaves as well as the perfectly balanced tree, although it is much simpler to maintain. Empirical evidence also suggests that, on the average, rebalancing is necessary once for approximately every two insertions. Here single and double rotations are equally probable. The example of Fig. 4.34 has evidently been carefully chosen to demonstrate as many rotations as possible in a minimum number of insertions.

The complexity of the balancing operations suggests that balanced trees should be used only if information retrievals are considerably more frequent than insertions. This is particularly true because the nodes of such search trees are usually implemented as densely packed records in order to economize storage. The speed of access and of updating the balance factors -- each requiring two bits only -- is therefore often a decisive factor to the efficiency of the rebalancing operation. Empirical evaluations show that balanced trees lose much of their appeal if tight record packing is mandatory. It is indeed difficult to beat the straightforward, simple tree insertion algorithm.

4.5.2. Balanced Tree Deletion

Our experience with tree deletion suggests that in the case of balanced trees deletion will also be more complicated than insertion. This is indeed true, although the rebalancing operation remains essentially the same as for insertion. In particular, rebalancing consists again of either single or a double rotations of nodes.

The basis for balanced tree deletion is the ordinary tree deletion algorithm. The easy cases are terminal nodes and nodes with only a single descendant. If the node to be deleted has two subtrees, we will again replace it by the rightmost node of its left subtree. As in the case of insertion, a Boolean variable parameter h is added with the meaning “the height of the subtree has been reduced”. Rebalancing has to be considered only when h is true. h is made true upon finding and deleting a node, or if rebalancing itself reduces the height of a subtree. We now introduce the two (symmetric) balancing operations in the form of procedures, because they have to be invoked from more than one point in the deletion algorithm. Note that *balanceL* is applied when the left, *balanceR* after the right branch had been reduced in height.