Data Structures in Pascal

Edward M. Reingold
University of Illinois at Urbana-Champaign

Wilfred J. Hansen
Carnegie-Mellon University

Little, Brown and Company
Boston Toronto

Library of Congress Cataloging-in-Publication Data

Reingold, Edward M., 1945–
Data structures in Pascal.

(Little, Brown computer systems series)
Includes index.
1. Data structures (Computer science) 2. PASCAL
(Computer program language) I. Hansen, Wilfred J.
II. Title. III. Series.
QA76.9.D35R443 1986 005.7'3 85-19874
ISBN 0-316-73931-6

Library of Congress Catalog Card No. 85-19874

ISBN 0-316-73931-6

9 8 7 6 5 4 3 2 1

ALP

Published simultaneously in Canada
by Little, Brown & Company (Canada) Limited

Printed in the United States of America

The cover is a reproduction of the first panel of the triptych *Three Trees* by Karel Appel, reproduced by permission of Karel Appel.

Acknowledgment

Figures 8.1–8.13, 8.15: From Reingold/Nievergelt/Deo, *Combinatorial Algorithms: Theory and Practice*, © 1977, pp. 280, 282, 284, 285, 291, 292, 293, 296, 300, 304, 306, and 315. Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Height-Balanced Trees. An extended binary tree is *height-balanced* if and only if it consists of a single external node, or the two subtrees T_l and T_r of the root satisfy

1. $| h(T_l) - h(T_r) | \leqslant 1$, and
2. T_l and T_r are height-balanced.

In other words, at any node in a height-balanced tree the two subtrees of that node differ in height by at most one. Figure 7.13 shows two trees, one height-balanced and the other not.

We want to use height-balanced trees as a storage structure for dynamic tables, but in order for height-balanced trees to be useful we must demonstrate that search times are at worst $O(\log n)$ and that insertions and deletions are easily and efficiently accommodated. Once we have shown that a height-balanced tree of n nodes has height $O(\log n)$, then the worst-case search time is $O(\log n)$ and, since the insertion/deletion time will also be proportional to the height, we will be done.

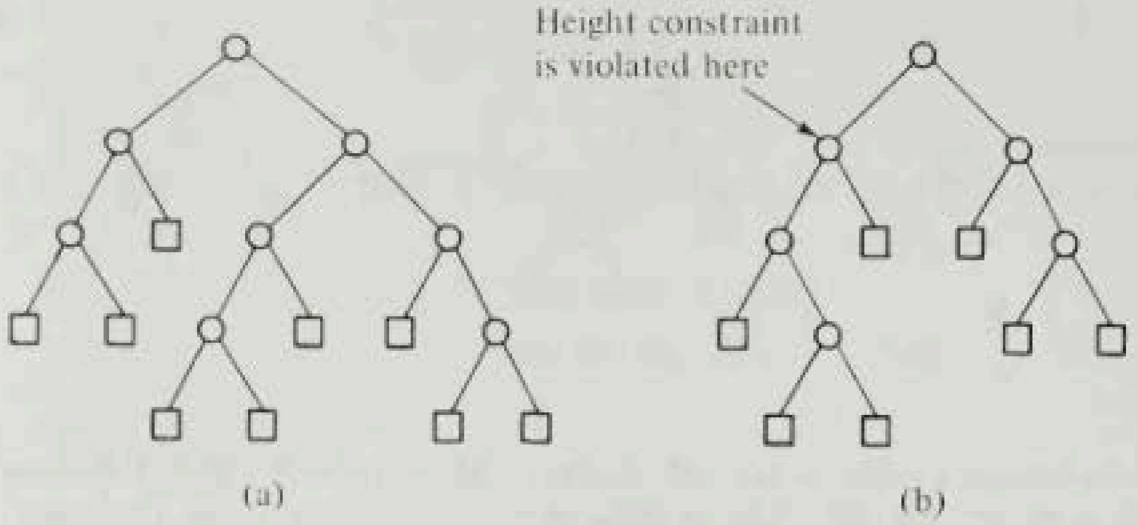


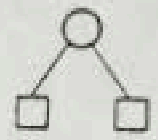
Figure 7.13
Two extended binary trees: (a) height-balanced and (b) not height-balanced

What is the height of the tallest height-balanced tree containing n internal nodes and $n + 1$ external nodes? To answer this question we will turn it around and ask what is the least number of internal nodes necessary to achieve height h in a height-balanced tree.

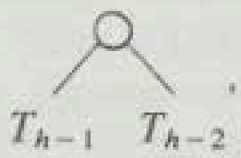
Let T_h be a height-balanced tree of height h with n_h internal nodes, the fewest possible. Obviously,

$$T_0 = \square, \qquad n_0 = 0$$

and

$T_1 =$  , $\qquad n_1 = 1 \qquad$ (7.17)

since these are the only extended binary trees with heights 0 and 1, respectively. Now consider T_h, $h \geq 2$. Since T_h is height-balanced and has height h, it must have a tree of height $h - 1$ as its left or right subtree and a tree of height $h - 1$ or $h - 2$ as its other subtree. For any k, a tree of height k has a subtree of height $k - 1$ and thus $n_k > n_{k-1}$; this tells us that T_h has one subtree of height $h - 1$ and the other of height $h - 2$, for if T_h had two subtrees of height $h - 1$, we would replace one of them by T_{h-2} and, since $n_{h-1} > n_{h-2}$, this would contradict the assumption that T_h had as few nodes as possible for a height-balanced tree of height h. Similarly, the two subtrees of T_h must be height-balanced and have the fewest nodes possible, for otherwise we could replace one or both subtrees with same height subtrees of fewer nodes, again contradicting the assumption that T_h has as few nodes as possible. Thus T_h has T_{h-1} as one subtree and T_{h-2} as the other,

$T_h =$ 

, $\qquad n_h = n_{h-1} + n_{h-2} + 1 \qquad$ (7.18)

Readers familiar with Fibonacci numbers will notice the resemblance of the construction of T_h and the recurrence relation for n_h to those for the Fibonacci numbers.

The solution of n_h in terms of h can be shown to be

$$n_h = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{h+2} - 1$$

or, one less than the $(h + 2)$nd Fibonacci number. Since $|(1 - \sqrt{5})/2| < 1$, the term $[(1 - \sqrt{5})/2]^{h+2}/\sqrt{5}$ is always quite small, so that

$$n_h + 1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} + O(1)$$

Since the tree of height h with the fewest nodes has n_h nodes, it follows that any tree with fewer than n_h nodes has height less than h. Therefore, if a height-balanced tree of n nodes has height h, then

$$n + 1 \geqslant n_h + 1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} + O(1)$$

implying that

$$\begin{aligned} h &\leqslant \frac{1}{\lg \frac{1+\sqrt{5}}{2}} \lg(n + 1) + O(1) \\ &\approx 1.44 \lg(n + 1) \end{aligned}$$

(we have written $n + 1$ and not n so the function will be properly defined for $n = 0$, that is, a tree with only a single external node).

Thus in the worst case the number of probes by Algorithm 7.5 into a height-balanced tree of n internal nodes will be about $1.44 \lg(n + 1)$ and the total search time will be logarithmic, as desired. It remains to be seen, however, whether insertions and deletions can be done efficiently so that a height-balanced tree remains height-balanced afterward.

To make an insertion or deletion we will use the method outlined above for the case when the tree can change in an unconstrained manner, but we will follow it with a rebalancing pass that verifies or restores the height-balanced state of the tree. In order to verify/restore the tree we need to be able to test whether the element inserted or deleted has changed the relationship between the heights of the subtrees of a node so as to violate the height constraints. For this purpose, we will store a *condition code* in each node of a height-balanced tree. The condition code is one of the following:

- / Means the left subtree of this node is taller (by one) than its right subtree.
- = Means the two subtrees of this node have equal height.
- \ Means the right subtree of this node is taller (by one) than its left subtree.

Storing condition codes requires an extra two-bit field per node in the tree. This very modest additional storage requirement can be made even more modest by the techniques of either Exercise 15 or Exercise 16, which reduce the condition code to a single bit, but at the expense of the efficiency and (relative) simplicity of the algorithms for rebalancing a tree that has become unbalanced through an insertion or deletion.

Roughly speaking, the rebalancing pass consists of retracing the path upward from the newly inserted node (or from the site of the deletion) to the root. If *PARENT* pointers are available, they are the most efficient way to accomplish this. If they are not available—and they usually are not—then we have two choices: either store the path, node by node, on a stack as we go down the tree from the root to the site of the modification, or use the trick of Algorithm 6.5(b) (see Exercise 6 of Section 6.1.2) to change the pointers as we go down the tree; this latter choice will require an additional two bits per node. In the case of an insertion there is a third possibility, that of simply retracing a portion of the path downward—see Exercise 11.

As the path is followed upward, we check for instances of the taller subtree growing taller (on an insertion) or the shorter subtree becoming shorter (on a deletion). When we find such an occurrence, we apply a local transformation to the tree at that point. In the case of an insertion it will turn out that applying the transformation at the first such occurrence will completely rebalance the tree. In the case of a deletion the transformations may need to be applied at many points along the way up to the root. We must be careful that any transformation made does not affect the inorder of the nodes in the tree (why?).

Since the rebalancing after an insertion is a bit simpler than after a deletion, we consider it first. What could have happened using Algorithm 7.7 to add an element to a height-balanced tree? The only difficulty is that the new element may have been added to the bottom of the taller of two subtrees of some node. Without loss of generality, suppose the right subtree was the taller before the insertion, as in Figure 7.14.

The way to repair the newly created imbalance depends on where within the taller subtree T the insertion was made. Suppose it was in the right subtree of T; we then have a situation that can be repaired as shown in Figure 7.15(a). The transformation shown there is called a *rotation*, and it is considered to be applied to the element A. Obviously, if the left subtree in Figure 7.14 had been taller and the

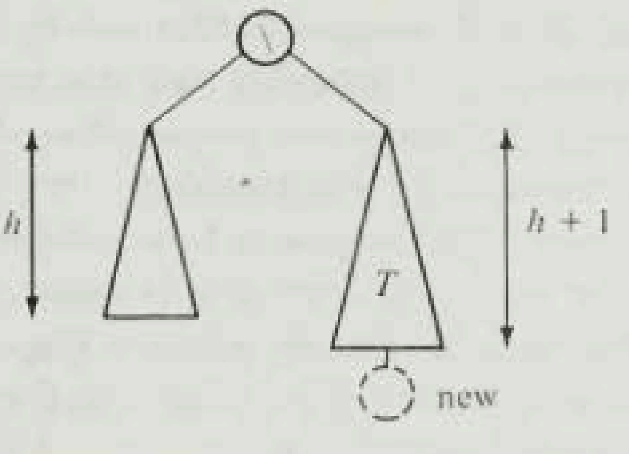


Figure 7.14
Insertion making a node unbalanced

insertion made it even taller, we would have to rotate in the other direction, using the mirror image of the transformation shown in Figure 7.15(a).

The transformation of Figure 7.15(a) would not have helped if the insertion had been to the left subtree of T in Figure 7.14—that is, to T_2 in Figure 7.15(a). In this case the repair is made as shown in Figure 7.15(b). This transformation, called

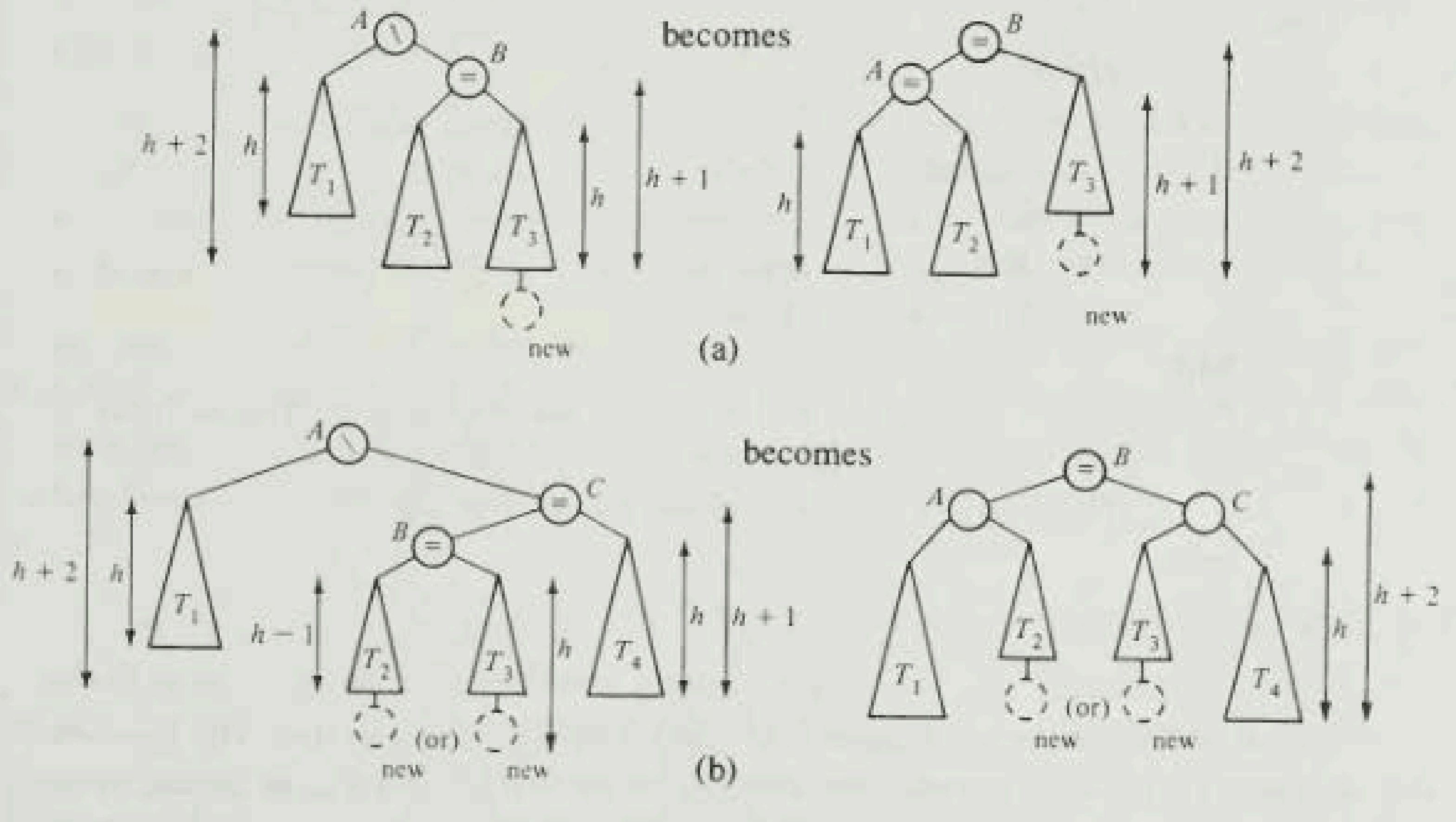


Figure 7.15
The transformations used to rebalance a height-balanced tree after the insertion of a new element: (a) rotation around A, (b) double rotation around A. The height condition codes in A and C in the right-hand drawing of (b) depend on whether the new element is at the bottom of T_2 or T_3. Both T_2 and T_3 are empty when B is the new element (see also Exercise 9). Notice that in each transformation the inorder of the tree is unchanged and the height of the tree *after* the transformation is the same as the height of the tree *before* the insertion. In each case, there are corresponding mirror-image transformations.

a *double rotation*, is considered to be applied at A. The new element can be at the bottom of either T_2 or T_3. Both T_2 and T_3 can be empty, in which case B is the new element (see Exercise 9). Again, a mirror-image transformation would be needed in the comparable case where the left subtree in Figure 7.14 was the taller.

Before continuing, the reader should try enough examples to be convinced that Figures 7.15(a) and (b), along with their mirror images, are the only possible cases.

We will generally show rotations in diagrams rather than program fragments, but the corresponding fragments are not intricate; they have no loops and few conditional tests. Here, for instance, is the code for Figure 7.15(a):

```
A↑.RIGHT := B↑.LEFT;
B↑.LEFT := A;
A↑.CONDITION := "=";
"pointer to A" := B
```

We utilize here and below the identifiers shown in Declarations 7.3.

```
type
    BalanceType = ("/", "=", "\");
    pHbTree = ↑HbTree;
    HbTree = record
          KEY: KeyType;
          LEFT, RIGHT: pHbTree;
          CONDITION: BalanceType
    end;
    PathStack = record   {used to record the nodes on path from root}
          TOP: integer;      {if TOP is -1, the stack is empty}
          STACK: array [0.."maximum depth"] of pHbTree
    end;
```

Declarations 7.3

The transformations of Figure 7.15 have two critical properties: the inorder of the elements of the tree remains the same after the transformation as it was before, and the overall height of the tree is the same after the transformation as it was before the insertion. The first property is necessary if we are to be able to search the tree properly by Algorithm 7.5. The second property means that after the insertion the appropriate transformation needs to be applied *only* at the lowest unbalanced spot in the tree.

The insertion algorithm is thus as follows. Use Algorithm 7.7 to insert the new element into its proper place, setting its condition code to = and storing the path followed down the tree, node by node, in a stack (the new element ends up as the top stack entry and the root is the bottom stack entry). Then, retrace that path backward, up the tree, popping nodes off the stack and correcting the height-condition codes until either the root is reached and its height-condition code corrected, or we reach a point when no more height-condition codes need to be corrected, or we reach a point at which a rotation or double rotation is necessary to rebalance the tree. More specifically, we follow this path backward, node by node, taking actions as defined by the following rules, where *current* is the current node on the path, *child* is the node before *current* on the path (that is, its child), and *grandchild* is the node before *child* on the path (the grandchild of *current*). Initially *child* is the new element just inserted, *current* is its parent, and *grandchild* is **nil**. These rules are expanded to Pascal in Algorithm 7.8.

1. If *current* has height condition =, change it to \ if *child* = *current*↑.RIGHT and to / if *child* = *current*↑.LEFT. In this case the subtree rooted at *child* grew taller by one unit, causing the subtree rooted at *current* to grow taller by one unit, so we continue up the path, unless *current* is the root, in which case we are done. To continue up the path we set *grandchild* := *child*; *child* := *current*, and *current* to the top stack entry, which is removed from the stack.

2. If *current* has height condition / and *child* = *current*↑.RIGHT or *current* has height condition \ and *child* = *current*↑.LEFT, change the height condition of *current* to =, and the procedure terminates. In this case the shorter of the two subtrees of *current* has grown one unit taller, making the tree better balanced.

3. If *current* has height condition / and *child* = *current* ↑.LEFT or *current* has height condition \ and *child* = *current*↑.RIGHT, then the taller of the two subtrees of *current* has become one unit taller, unbalancing the tree at *current*. A transformation is performed according to the following four cases:

	grandchild = *child*↑.RIGHT	*grandchild* = *child*↑.LEFT
child = *current*↑.RIGHT	Rotate around *current* using Figure 7.15(a)	Double-rotate around *current* using Figure 7.15(b)
child = *current*↑.LEFT	Double-rotate around *current* using the mirror image of Figure 7.15(b)	Rotate around *current* using the mirror image of Figure 7.15(a)

In each case, the height conditions are set as shown in Figure 7.15. The procedure terminates, having rebalanced the tree at its lowest point of imbalance.

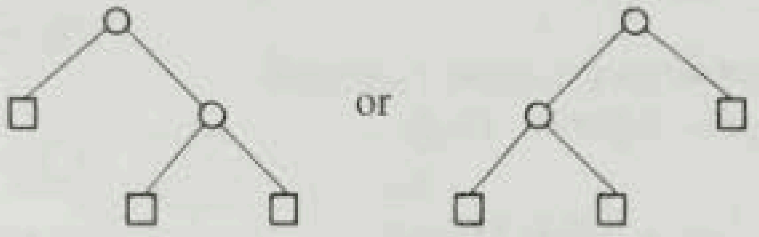
```
procedure RebalanceAfterInsert(var S: PathStack);
        {rebalance a height-balanced tree whose most recent insertion path
        is recorded in S}
var
    current, child, grandchild: pHbTree;
    balancing: boolean;
begin
    child := Pop(S);
    current := Pop(S);
    grandchild := nil;
    balancing := true;
    while balancing do begin
        {Assert: S has the path from current to root. current is the parent
        of child. child is the parent of grandchild. The insertion was made
        in subtree at child which is height-balanced.}
        if current↑.CONDITION = "=" then begin
            {Rule 1: current was balanced:
                propagate imbalance up tree}
            if child = current↑.RIGHT then
                current↑.CONDITION := "\"
            else current↑.CONDITION := "/";
            if IsEmpty(S) then
                balancing := false
            else begin
                grandchild := child;
                child := current;
                current := Pop(S)
            end
        end
        else if ((current↑.CONDITION = "/")
                    and (child = current↑.RIGHT))
                or ((current↑.CONDITION = "\")
                    and (child = current↑.LEFT)) then begin
            {Rule 2: short side of current grew taller:
                now better balanced}
            current↑.CONDITION := "=";
            balancing := false
        end
```

```
                else begin
                        {Rule 3: tall side of current grew taller: do a rotation}
                        if child = current↑.RIGHT then
                                if grandchild = child↑.RIGHT then
                                        {rotate around current}
                                        "apply Figure 7.15(a)"
                                else
                                        {double-rotate around current}
                                        "apply Figure 7.15(b)"
                        else
                                if grandchild = child↑.RIGHT then
                                        {double-rotate around current}
                                        "apply the mirror image of Figure 7.15(b)"
                                else
                                        {rotate around current}
                                        "apply the mirror image of Figure 7.15(a)";
                        balancing := false
                end
        end
end
```

Algorithm 7.8
Rebalance a tree after the insertion whose path has been recorded in a stack

As an example of the insertion process, consider inserting the letter T into the tree of Figure 7.16. Algorithm 7.7 inserts T as the right child of S, and the path back up the tree is T, S, R, P, K, U, H. We thus begin rebalancing with *current* = S, *child* = T, and *grandchild* = **nil**, while the stack contents are R, P, K, U, H. Case 1 applies to *current*, so we change the condition code of S to $\searrow$ and set *grandchild* := T, *child* := S, and *current* := R from the top of the stack. Again case 1 applies to *current*, so we set the condition code of R to $\searrow$, *grandchild* := S, *child* := R, and *current* := P from the top of the stack. Again case 1 applies, so we set the condition code of P to $\searrow$, *grandchild* := R, *child* := P, and *current* := K from the top of the stack. Now case 3 applies and, since *child* = *current*↑.RIGHT and *grandchild* = *child*↑.RIGHT, we apply the rotation of Figure 7.15(a) to the node *current* = K, setting the height-condition codes of K and P to =, as indicated in Figure 7.15(a). The procedure then terminates (ignoring the remainder of the stack contents). If we had been inserting O instead, then upon reaching the node K we would apply the double rotation of Figure 7.15(b).

The deletion process is more complex than insertion because it will not always be sufficient to apply a transformation only at the lowest point of imbalance; transformations may need to be applied at many levels between the site of the deletion and the root. To delete a node we may need to begin, as with unconstrained trees, by replacing it with the contents of another node, which is then deleted. If the node has two non-**nil** children, it is replaced by its inorder successor or predecessor, which is guaranteed to have at most one non-**nil** child (see Figure 7.12). In the case of a height-balanced tree, however, if a node has only one non-**nil** child, it must look like



because the height constraint. In either case, replacing the node by its only child has the identical effect on the heights as deleting the node with no children. We thus need consider *only* the case of deleting a node with two null children.

As in the insertion algorithm, we store on a stack the path followed down the tree to the site of the node to be deleted, then we retrace the path backward up the tree, popping nodes off the stack, correcting height-condition codes, and making transformations as needed. As we go back up the path, actions are taken as defined by the following rules, where (as before) *current* is the current node on the path and *child* is the node before *current*. Initially, *child* is the node to be deleted and *current* is its parent, if any. The actual deletion occurs after the rebalancing because the algorithm may need to test the pointer from parent to deleted child. A Pascal version of these rules is given as Algorithm 7.9.

1. If *current* has height condition =, then shortening either subtree does not affect the height of the tree rooted at *current*. The condition code of *current* is changed to ╲ if *child* = *current*↑.LEFT and to ╱ if *child* = *current*↑.RIGHT. The procedure then exits from the balancing process.

2. If *current* has height condition ╲ and *child* = *current*↑.RIGHT or *current* has height condition ╱ and *child* = *current*↑.LEFT, the condition code of *current* is changed to =.

3. If *current* has height condition ╲ and *child* = *current*↑.LEFT, then the height constraint is violated at *current*. There are three subcases, depending on the height-condition code at *current*↑.RIGHT, the sibling of *child*. The subcases are as given in Figure 7.17.

4. If *current* has height condition ╱ and *child* = *current*↑.RIGHT, then the height constraint is violated at *current*. There are three subcases, depending on the height-condition code at *current*↑.LEFT, the sibling of *child*. The subcases are the mirror images of those given in Figure 7.17, and we leave them as Exercise 12.

5. The balancing process may have terminated in step 1, 3(a), or 4(a); otherwise the height of the subtree rooted at *current* is now one less than it was before the deletion, so we continue up the path, unless *current* is the root, in which case we are done. To continue up the path we set *child* := *current* and set *current* by popping the top element off the stack. We then repeat this set of rules.

As an example of the deletion procedure, consider deleting the node B from the tree of Figure 7.16. Case 2 applies to the node A and then case 3(c) applies to the node C, so a double rotation is applied there. Then case 3(a) applies to the node H and a rotation is applied there. The tree is then completely rebalanced. The reader can become familiar with the algorithm by going through the steps necessary to delete D or X from the tree of Figure 7.16.

The deletion algorithm will clearly require only time proportional to the height of the tree; thus deletion can be accomplished in $O(\log n)$ time, as can insertion. An insertion, however, will need at most one rotation/double rotation to rebalance the tree, while a deletion from a height-balanced tree of height h can require as many as $\lfloor h/2 \rfloor$ rotations/double rotations, but no more (Exercise 14).

What do height-balanced trees look like "on the average"—that is, when they are generated by a random sequence of insertions and deletions? There is no known mathematical analysis to answer this question, but empirical evidence strongly suggests that the average search time in such a tree is $\lg n + O(1)$ probes. This suggests that on the average height-balanced trees are almost as good as the completely balanced trees that correspond to binary search. Of course, unlike the case of binary search, height-balanced trees can be modified by insertions and deletions in logarithmic time.

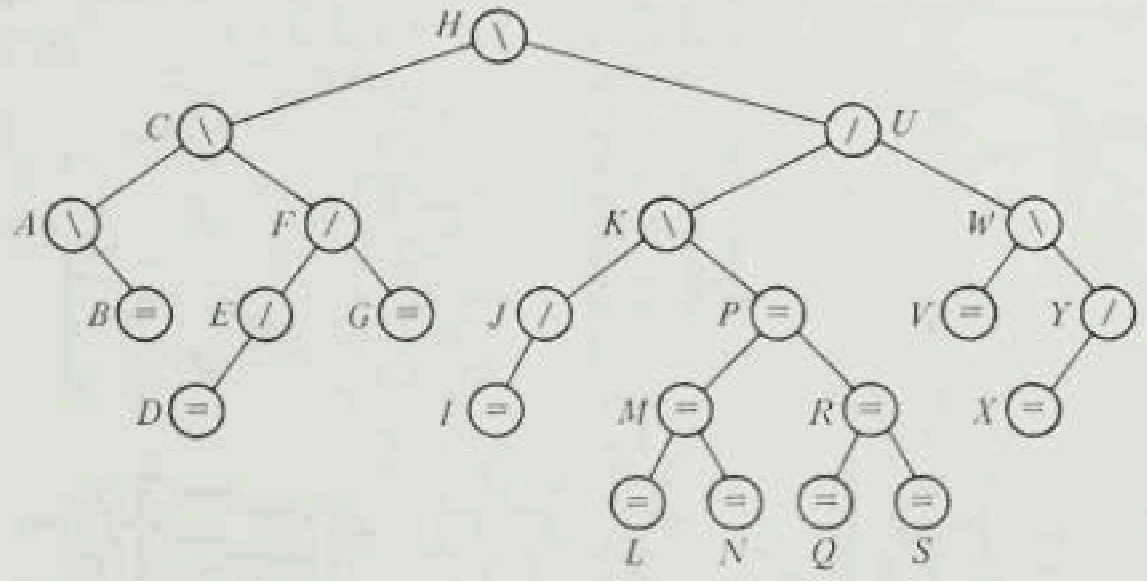
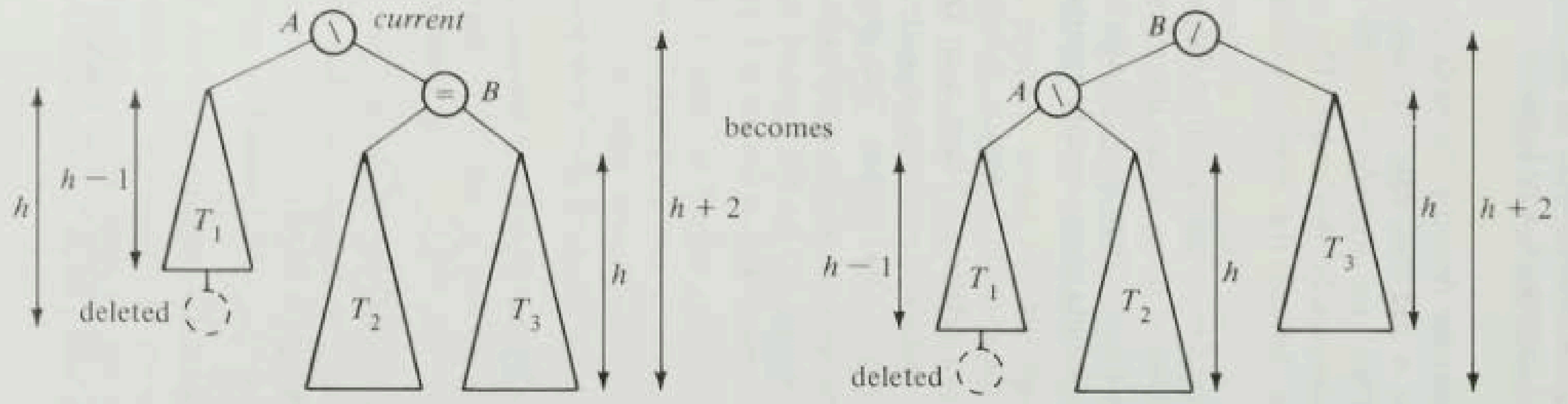
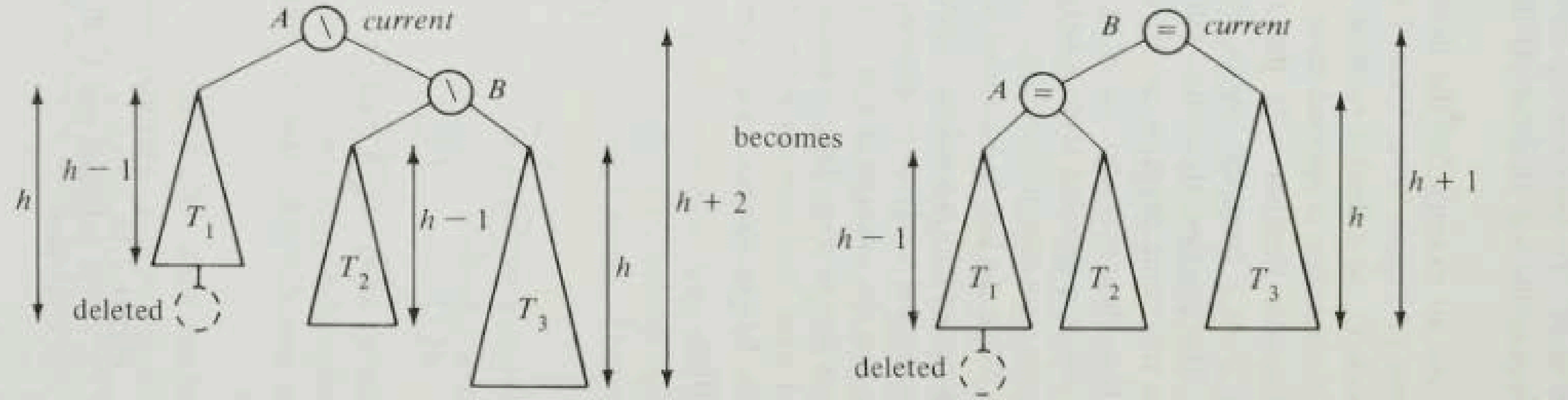


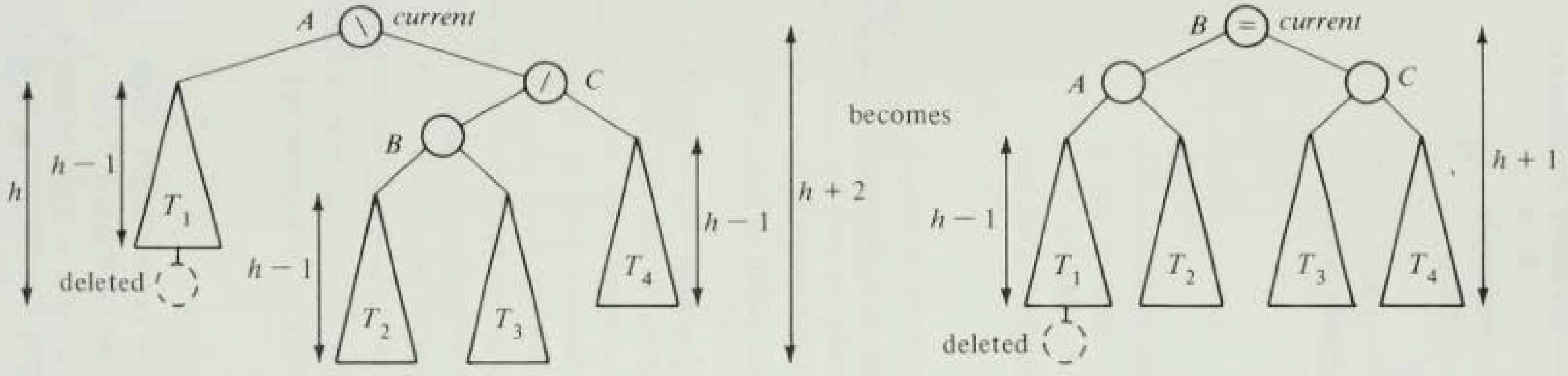
Figure 7.16
A height-balanced tree to illustrate the rebalancing algorithms after an insertion or a deletion



(a) Apply the rotation of Figure 7.15(a) to *current* and exit from the balancing operation since the height-balance has been restored and the height of the tree after the transformation is the same as it was before the deletion. (See also Exercise 13.)



(b) Apply the rotation of Figure 7.15(a) to *current* and change *current* to point to *B*.



(c) Apply the double rotation of Figure 7.15(b) to *current* and change *current* to point to *B*. The height-condition codes of *A* and *C* are both = if that of *B* was =. If *B* was \, then *A* is / and *C* is =. If *B* was /, then *A* is = and *C* is \.

Figure 7.17
The three rotations utilized in Algorithm 7.9 when *current* has height condition \ and *child* = *current*↑.LEFT. Three more required rotations are derived as mirror images of those here (Exercise 12).

```
procedure RebalanceAndDelete(var T: pHbTree; var S: PathStack);
var
    current: pHbTree;   {node currently considered}
    child: pHbTree;     {current↑.LEFT or ↑.RIGHT}
    balancing: boolean; {loop control flag}
    bereft: pHbTree;    {node whose leaf is to be deleted}
    wasLeft: boolean;   {which side of bereft to delete}
begin
    child := Pop(S);
    if IsEmpty(S) then
        T := nil
    else begin
        current := Pop(S);
        bereft := current;
        wasLeft := (child = bereft↑.LEFT);
        balancing := true;
        while balancing do begin
            {Assert: S has path from current to root. bereft is parent of
            the node to be deleted. current is the parent of child. bereft
            is in the subtree rooted at child.}
            if current↑.CONDITION = "=" then begin
                {Rule 1: deletion from either subtree can be absorbed
                here}
                if child = current↑.LEFT then
                    current↑.CONDITION := "\"
                else current↑.CONDITION := "/";
                balancing := false
            end
            else if ((current↑.CONDITION = "\")
                          and (child = current↑.RIGHT))
                     or ((current↑.CONDITION = "/")
                          and (child = current↑.LEFT))      then
                {Rule 2: current becomes balanced, but its subtree is
                shorter so imbalance must be propagated up}
                current↑.CONDITION := "="
            else {Rules 3 and 4: Rotate}
                if (current↑.CONDITION = "\")
                          and (child = current↑.LEFT) then
                    "apply the rotations of Figure 7.17"
                else "apply the mirror images of the rotations in
                          Figure 7.17";
                {the rotations may have set *balancing* to *false*, otherwise we
                continue up the tree}
                **if** *balancing* **then**   {**Rule 5**}
                    **if** *IsEmpty*(*S*) **then**
                        {we are at *root*; it remains unbalanced}
                        *balancing* := **false**
                    **else begin**
                        *child* := *current*;
                        *current* := *Pop*(*S*)
                    **end**
            **end**;
            {Delete designated node}
            **if** *wasLeft* **then**
                *bereft*↑.*LEFT* := **nil**
            **else** *bereft*↑.*RIGHT* := **nil**
        **end**
    **end**
```

Algorithm 7.9
Rebalance a height-balanced tree and complete a deletion. The path to the node to be deleted has been recorded in stack *S*. Note that the deletion is not actually performed until after the rebalancing.