We can therefore use weight-balanced trees or height-balanced trees augmented with *SIZE* fields as a compromise between the linked and sequential storage structures for representing linear lists. The normal sequential storage structure (Section 4.1)

could be easily searched by index position but was expensive to modify, while the normal linked structure (Section 4.2) was easy to modify once the location was known but expensive to search. Both searching by index position and insertions/deletions can be done with height- or weight-balanced trees in logarithmic time. This compromise is especially useful in implementing a priority queue (see the introduction to Section 4.2) which operates in a first-in, highest-priority-out-first order. As the elements arrive they are inserted into the tree according to their value by Algorithm 7.7 and the appropriate rebalancing scheme. The element deleted is always the one with the highest priority. In this case we do not even need the full power of Algorithm 7.10 to find the element to be deleted (why?).

Using balanced trees as a storage structure for linear lists suggests the need to be able to concatenate them together and split them apart, just as we can do with linked lists. Can these operations also be done in logarithmic time? Yes they can for both height- and weight-balanced trees, although the operations are slightly more complex for weight-balanced trees (see Exercise 31).

Suppose, first, we want to concatenate height-balanced tree U to the right of height-balanced tree T and have the result be a height-balanced tree. We proceed as follows. Compute the heights of T and U in logarithmic time (see Exercise 17). Assume that $heightHB(T) \geq heightHB(U)$; the other case is essentially the mirror image. Delete the leftmost inorder element of U, call it q, and rename the remaining tree V. Then use a *paste* operation to paste everything together:

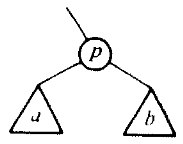
$$PasteHB(T, q, V)$$

This routine constructs a height-balanced tree from node q and trees T and V given that all nodes of T precede q in inorder and q precedes all nodes of V.

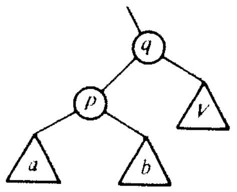
In *PasteHB*, Algorithm 7.11, we first compute the heights of the two trees and determine the taller so we can insert q and the smaller tree at a place of proper height in the taller. By assumption above, T is the taller. We descend within T following *RIGHT* links to a node p at about the same height as the height of V. The computation begins with the initial height of T and at each node subtracts either 1 or 2 depending on the height-condition code, 1 if the code was $=$ or $\diagdown$ and 2 if it was $\diagup$ (why?). This continues until we find a node p in T such that

$$0 \leq heightHB(p) - heightHB(V) \leq 1$$

(see Exercise 32). The node



is replaced by



```
function PasteHB(T, q, V: pHbTree): pHbTree;
    {Construct a height-balanced tree from trees T and V and node q. If the
    result is to be in inorder, we must have the initial inorder condition:
    last(T) ≤ q ≤ first(V)}
var
    p, parent: pHbTree;
    hT, hV, hp: integer;   {heights of the subtrees}
    S: PathStack;
begin
    hT := heightHB(T);
    hV := heightHB(V);
    if hT ≥ hV then begin
        Empty(S);
        p := T;
        hp := hT;
        parent := nil;
        while (hp' − hV) > 1 do begin
            {Assert: S contains the nodes on the path from p to root. p is
            right child of parent. hp is height of tree rooted at p.}
            Push(p, S);
            if p↑.CONDITION = "/" then
                hp := hp − 2
            else hp := hp − 1;
            parent := p;
            p := p↑.RIGHT
        end;
        q↑.LEFT := p;
        q↑.RIGHT := V;
        if hp = hV then q↑.CONDITION := "="
        else q↑.CONDITION := "/"
        if parent ≠ nil then parent↑.RIGHT := q;
        Push(q, S);
        PasteHB := RebalanceAfterInsert(S)     {Exercise 33}
    end
    else begin
        "this case is a mirror-image of the above"
    end
end
```

Algorithm 7.11
Concatenate two height-balanced trees by pasting a node between them

with the height-condition code of q being either $=$ or $/$ depending on *heightHB(p)* − *heightHB(V)*. Having stored the nodes encountered along the right boundary of T on a stack we conclude by going up that boundary beginning at the original parent of p, correcting height-condition codes and performing a rotation or double rotation as though we had done an insertion into the subtree rooted q and had thereby increased its height by one unit.

The problem of splitting a tree in two, corresponding to splitting a linked list into two pieces, is solved by disconnecting portions of the tree and reassembling them with *PasteHB*. To understand the idea, consider the tree of Figure 7.20. The list represented is the inorder of the tree, $S_1aS_2bS_3cS_4dT_1eT_2fT_3$, and suppose this is to be split into two lists $S_1aS_2bS_3cS_4d$ and $T_1eT_2fT_3$; in other words, the list is to be split after node d, as shown in Figure 7.20 by the dashed line. Assume that in tracing the path from the root to d the nodes have been stored on a stack, as in the other height-balanced tree algorithms. We now go back up that path toward the root, breaking the tree apart and concatenating the pieces together to form the desired lists. First d is inserted at the extreme right of S_4, to give S_4d. S_3 and S_4d are then concatenated using c as the paste node in the concatenation algorithm to form S_3cS_4d. The node e is then used as the paste node in concatenating T_1 and T_2 to form T_1eT_2. The node b is then used as the paste node in concatenating S_2 to S_3cS_4d, giving $S_2bS_3cS_4d$, which is in turn concatenated to S_1 using a as the paste node, giving $S_1aS_2bS_3cS_4d$.

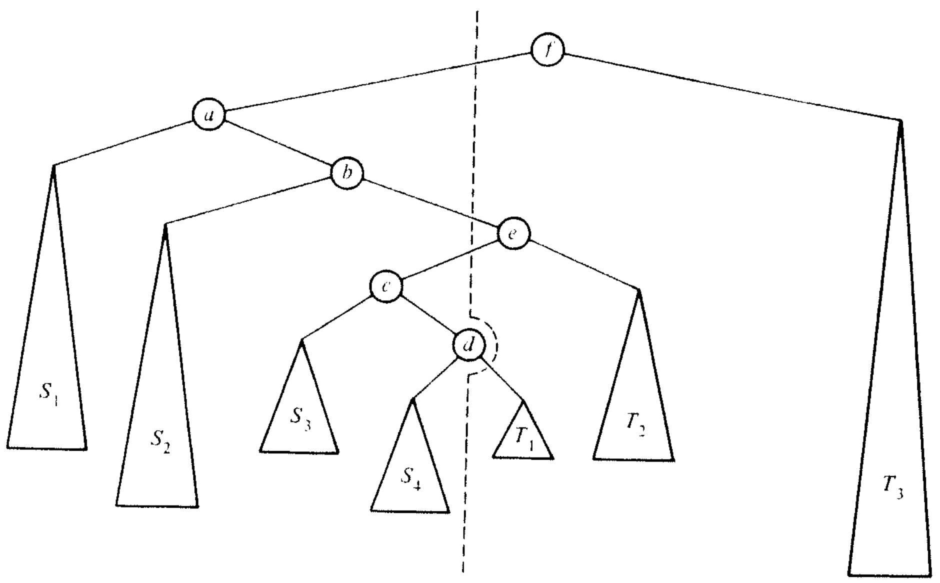


Figure 7.20
An example of splitting a binary tree into two pieces based on the inorder of the nodes

On the last step, f is used as the paste node in concatenating T_3 to T_1eT_2, giving $T_1eT_2fT_3$.

Algorithm 7.12 gives an outline of the procedure in general. Notice that the algorithm as outlined will work to split any binary tree, not just a height-balanced tree. For example, using the insertion and concatenation procedures for weight-balanced trees, Algorithm 7.12 serves to split a weight-balanced tree, resulting in two weight-balanced trees. The time required by Algorithm 7.12 will be proportional to the total required by the insertion and the sequence of concatenations. In height- or weight-balanced trees these are potentially $O(\log n)$, and the concatenation process requires $O(\log n)$ time, suggesting that the splitting algorithm might require time proportional to $(\log n)^2$ in the worst case for such balanced trees. Fortunately, however, the concatenation algorithm requires logarithmic time only to delete the node that will be used to paste the trees together. If given that node, as is the case in the concatenations done in the splitting process, the concatenation will require only time proportional to the difference in height of the two trees being concatenated. This leads to a logarithmic worst-case time for splitting balanced trees (Exercise 34).

```
procedure SplitTree(var P: PathStack; var S, T: pHbTree);
        {P contains the nodes on the path to a split node. Construct two
        trees in S and T from those nodes. The split node will be at the end
        of S.}
var
    current, child: pHbTree;
begin
    current := Pop(P);
    S := current↑.LEFT;
    T := current↑.RIGHT;
    S := PasteHB(S, current, nil);  {insert current in left result tree}
    while not IsEmpty(P) do begin
        {Assert: P has path from current to root. S and T contain, respec-
        tively, the left and right results of splitting the subtree at current in
        such a way as to keep the original split node at the end of S.}
        child := current;
        current := Pop(P);
        if child = current↑.RIGHT then
            S := PasteHB(current↑.LEFT, current, S)
        else
            T := PasteHB(T, current, current↑.RIGHT)
    end
end
```

Algorithm 7.12
Splitting a binary tree in two pieces based on the inorder of the nodes

Two final remarks about representing lists by trees are in order. First, the algorithms described can be used *without* the rebalancing parts, essentially allowing the trees to grow randomly. If the insertions, deletions, concatenations, splittings, and searches were all random, the resulting trees would probably maintain logarithmic height on the average. But in most applications it is extremely unlikely that the sequences of operations would be truly random; rather, biases would occur that would cause the trees to deteriorate badly. Second, if possible when using binary trees to represent lists, *PARENT* pointers should be maintained in the nodes. This will greatly facilitate the algorithms that require retracing the path from a node to the root: insertion, deletion, concatenation, and splitting. Furthermore, it will allow the deletion of a node given only a pointer to the node; in this sense *PARENT* pointers give something of an analog to doubly linked lists.

Exercises

1. Use elementary integral calculus to show that $H_n = \sum_{i=1}^{n} 1/i \approx \ln n$. [*Hint*: Compare H_n to $\int_1^n dx/x$ and use the rectangle rule.]

2. This exercise is an alternative way of deriving the result of Equation (7.16), which gives the external path length of a randomly constructed binary search tree of n elements. Let U_n be the average number of probes in an unsuccessful search in a given binary search tree T of n nodes. Let U_{n+1} be the average number of probes in an unsuccessful search after a random insertion has been made in T.
   (a) What is U_1?
   (b) What is the relationship between U_n and the E_n of Equation (7.16)?
   (c) Prove that $U_{n+1} = U_n + 2/(n + 2)$. [*Hint*: If the insertion is at level i, by how much does the external path length increase? What is the average level of a leaf in the tree?]
   (d) Use the above results, along with Exercise 1, to obtain (7.16).
   (e) Let S_n be the average number of probes in a successful search of a randomly constructed binary search tree of n elements. Prove that $S_n = 1 + (U_0 + U_1 + \cdots + U_{n-1})/n$.

3. Prove that if a node X in a binary tree has a non-**nil** left (right) child, then its inorder predecessor (successor) has a **nil** right (left) child.

4. Write out the details of the algorithm to delete an element from a binary search tree. (See page 340.)

5. Derive a recurrence relation for N_h, the number of distinct height-balanced trees of height h. How fast does N_h grow?