Using *Insert* as a guide, it should not be difficult to define *DeleteMin* and *Delete* for AVL trees. We leave these definitions as exercises, and simply observe that all that is necessary is to balance the tree, if necessary, after each recursive call has returned.

Efficiency and Verification

Has all this work been worth it? By using *BalanceRight* and *BalanceLeft* after every recursive call to *Insert* or *Delete,* we have guaranteed that a tree with n nodes will always have height $O(\log n)$, so the insertion, deletion, and *Find* operations will make no more than a logarithmic number of recursive calls. It would appear that we've done what we set out to, but we have to be a bit careful. Two things could mess us up: The balancing functions could take longer than constant time, or they might be called more than $O(\log n)$ times. It is easy to see that the balancing algorithms run in constant time; after all, they consist of nothing but straight-line code, with no loops or recursive calls.

If you look at the definition of *Insert,* you'll see that the balancing functions are called at most once for each recursive call to *Insert.* That's all we need; the balancing functions contribute a constant amount of time for each of the (no more than logarithmic) times they're called.

Finally, we mentioned that a balance operation at a node will restore balance at that node, but might change the height of that node in such a way as to require a balancing of the node's parent. Fortunately, that's taken care of by the nature of calls in *Insert.* Notice that *Insert* builds a stack of pending function calls as it walks its way down the tree to find where the new element belongs. Once the new node has been built and inserted, the pending calls are popped from the stack, effectively retracing the path used to find where the new node belonged. Look at where the balancing functions are called: just before the end of each call. That means that as *Insert* backs its way up the tree, it calls the balancing functions to restore balance from bottom up, exactly as required.

7.2 B-TREES

By now you should be completely comfortable with the principle that the timing of the search tree operations is driven by the height of the tree. As long as we can keep our tree broad and shallow, we can be sure that all our tree operations will be fast, which is to say logarithmic. With AVL trees, we kept the tree shallow by forcing the heights of the subtrees of any node to be nearly equal, using rotations to restore balance each time we inserted or deleted an element. The best case, the one in which the tree had the maximum number of elements for a given height, is clearly obtained with a complete binary tree, and we saw in Chapter 6 that the height of a complete bina-

ry tree with n nodes is always less than $\log_2 n$. It would seem that we can't do any better than that, but in this section we will show a tree data structure that does indeed have less depth than $\log_2 n$ for n nodes, and we will explore the reasons that this data structure has become the standard in many applications.

k-ary Trees, Again

Up to now, we have concentrated almost entirely on binary trees, on the grounds that (1) they are easy to implement, (2) there are many good applications using binary trees, and (3) any tree can be mapped to a binary tree, anyway, with leftmost children becoming left children and right siblings becoming right children. The problem with this mapping is that we usually wind up with a binary tree that is higher than the original. Since we are interested in keeping the height of the tree as small as possible, it might be worth exploring these trees in more depth (pun intended).

Recall that a k-ary tree (also called a **multiway tree** of order k) is just a tree in which every node may have as many as k children, for some integer k. In Figure 7.8, we show a 4-ary search tree. Notice that each node has at most four children, and each node has one less data element in it than it has children. Notice also that this tree generalizes the binary search tree property, in that the pointers between data elements in each node point to subtrees, each of which has all its data values strictly between the values that bracket the pointer. For instance, suppose we are searching for the element 71 in the tree. We begin at the root and find (perhaps by a sequential search) that 71 lies between the elements 34 and 80. We then take the pointer to the subtree whose values lie between 34 and 80 and repeat the process. We find that 71 is larger than the last element, 68, in that node, so we follow the pointer to the subtree whose values are all larger than 68 (and necessarily less than 80). We repeat the search at the leaf node and find 71 among the values in that leaf node, so the search is successful. If we were seeking 70, we would attempt to take the leftmost pointer in the 71

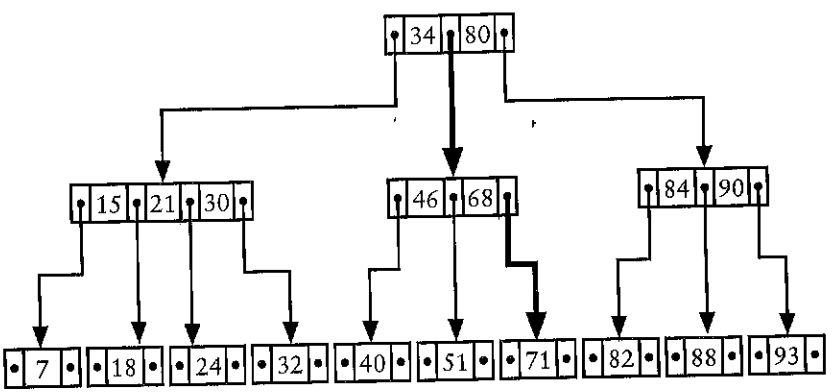


FIGURE 7.8
A 4-ary search tree, showing the search path for 71

node, find that it was NULL, and report failure, since there is no subtree that could possibly contain 70.

In the tree of Figure 7.8, notice that all the leaves are at the same level. This is certainly a stronger condition than we had for AVL trees. This will be one of the properties of the data structure we will introduce in this section; and it, along with a condition we will require on the sizes of the nodes, will enable us to get a good estimate on the height of such trees if we know the number of nodes in the tree. For now, we will just note that because of the higher fanout—that is, the number of pointers out of the nodes—we can pack much more information in a multiway tree than we could in a binary tree of the same height. For instance, Figure 7.8 shows 19 data elements in a tree of height 2, whereas a binary tree would need to be at least twice that height to store the same amount of information.

B-Trees Explained

We define a B-tree of order d to have as its structure a $(2d+1)$-ary tree with the following properties:

1. The data elements (or at least the key fields) in each node are assumed to be linearly ordered. If the data elements are records consisting of several fields, then one field of the record is a key field, with values taken from a linearly ordered set.
2. Each internal node has one more child than it does data elements (and leaves, of course, have no children).
3. The root contains between 1 and $2d$ data elements.
4. Each node except for the root contains between d and $2d$ data elements.
5. All leaves are at the same depth in the tree.
6. The tree has the **extended search tree property:** If the keys in a node n are arranged in their linear order, $k_1, k_2, \ldots, k_m$, then there is an associated linear order among the subtrees, $S_0, S_1, \ldots, S_m$, of that node such that (a) every key in S_0 is less than k_1, (b) every key in S_m is greater than k_m, and (c) for $1 \leq i < m$, every key in S_i lies strictly between k_i and k_{i+1}.

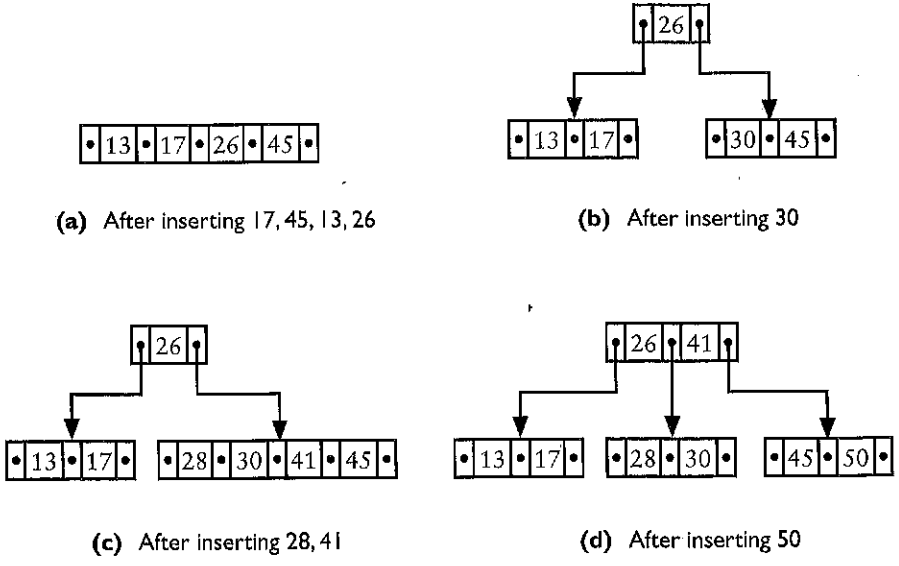
Note that the binary search tree property is just condition 6 with $m=1$, with S_0 representing the left subtree and S_1 the right subtree. Condition 4 can be restated this way: Each node except for the root is at least half full. B-trees were introduced by R. Bayer and E. M. McCreight in 1972, and folklore has it that they have never explained the choice of name for this particular data structure, so you are free to speculate on what the "B" stands for. B-trees were designed to support the operations *Insert, Delete, Find* a data element, given its key; *Update* an element without changing its key; and *FindNext,* which, when given a key value k, returns the least key k_i in the tree for which

$k<k_i$. The operation *Insert* is particularly ingenious and differs from the insertion algorithms for the tree structures we've seen so far in that if the tree ever needs to change its height, it does so by growing up from the root rather than down from a leaf.

To illustrate the action of *Insert,* consider a B-tree of order 2. In such a tree, the root may have from one to four elements, while all the rest of the nodes must have either two, three, or four elements. In Figure 7.9a, we see the root after having inserted the elements 17, 45, 13, and 26. As each new element arrives, it is placed in order in the root. When the element 30 arrives, it cannot be placed in the root node—there's no more room. What we do in this case is the heart of the insertion algorithm: We split the root into equal parts, one containing all the elements below the median, 26, and the other containing all the elements above the median. The median value itself gets promoted to a new root node, which has the split nodes as children, as in Figure 7.9b.

If we continue by inserting 28 and 41, we see that they are both greater than 26, so, by the search tree property, they belong in the right subtree of the root node, as they are in Figure 7.9c. Now, if we insert 50, we see that it should also go in the rightmost node. Of course, there's no room for it, so we again split the node into two and promote the median value, 41, to the parent node. If the parent node had been full, we would have had to split it and promote its median to a new root node. That's all there is to insertion: We try to place each node in its proper leaf, and if that would cause overflow, we split the node, promote the median value, and try to insert the promoted value into the parent node, splitting when necessary, tracking up the tree until we arrive at a node that does not overflow.

FIGURE 7.9
Inserting into a B-tree



(a) After inserting 17, 45, 13, 26

(b) After inserting 30

(c) After inserting 28, 41

(d) After inserting 50

If it sounds pretty simple, that's because it is—at least for people. Things get a bit more complicated when we try to translate the insertion process into algorithmic form, largely because we humans can pretty much automatically take care of such details as inserting an element into a node, splitting a node, and deciding where an element belongs in the tree—operations that require a moderate amount of fussy detail to program. We will provide a detailed account of the insertion algorithm from the top down; then, having done that, we will provide a sketch of the details of deletion and leave the programming to you.

The node declarations are immediate. Each node in the B-tree will consist of an array of data elements, a larger array of pointers to children, and a field for the present size of the array.

```
const int ORDER = 4;  // Global constant for order of tree

template<class T>
class BTreeNode
{
public:
   // The usual constructors go here.

   T data[1 + 2 * ORDER];  // We'll never use data[0].
   BTreeNode<T>* child[1 + 2 * ORDER];
   int size;
}
```

The insertion routine itself is little more than a shell. Most of the work is done by a recursive function, *RecursiveInsert*, which inserts an atom *a* into the tree rooted at *root*. If the insertion causes the root to be split, *root* then points to the left split node, *rp* points to the right split node (and is NULL if no splitting occurs at the root), and *promoted* is the atom that must be promoted to the new root.

```
void Insert(T a, BTreeNode<T>*& root)
{
   BTreeNode* temp, rp;
   T promoted;
   RecursiveInsert(a, root, rp, promoted);

   if (rp)     // Root was split, so build a new root.
   {
      temp = root;
      root = new BTreeNode<T>;
      root->size = 1;
      root->data[1] = promoted;
      root->child[0] = temp;
      root->child[1] = rp;
   }
}
```