When you're done evaluating the postfix string, you pop the one remaining item off the stack. Somewhat amazingly, this item is a complete tree depicting the algebraic expression. You can see the prefix and infix representations of the original postfix (and recover the postfix expression) by traversing the tree as we described. We'll leave an implementation of this process as an exercise.

Finding Maximum and Minimum Values

Incidentally, we should note how easy it is to find the maximum and minimum values in a binary search tree. In fact, this process is so easy we don't include it as an option in the Workshop applet, nor show code for it in Listing 8.1. Still, understanding how it works is important.

For the minimum, go to the left child of the root; then go to the left child of that child, and so on, until you come to a node that has no left child. This node is the minimum, as shown in Figure 8.12.

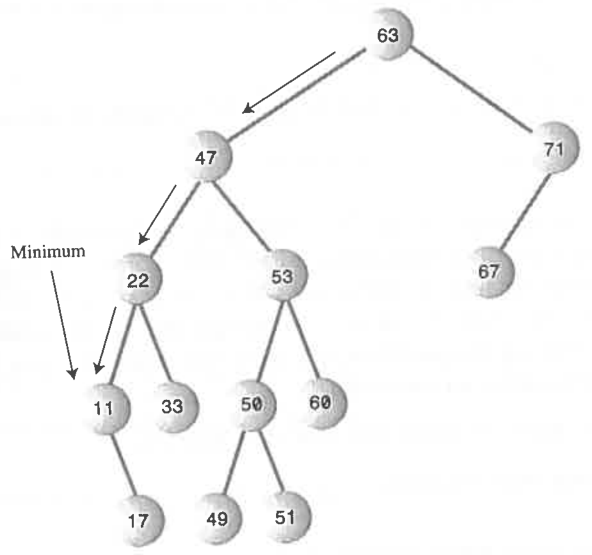


FIGURE 8.12 Minimum value of a tree.

Here's some code that returns the node with the minimum key value:

```
public Node minimum()     // returns node with minimum key value
   {
   Node current, last;
   current = root;                    // start at root
   while(current != null)             // until the bottom,
```

```
      {
      last = current;                 // remember node
      current = current.leftChild;    // go to left child
      }
   return last;
   }
```

We'll need to know about finding the minimum value when we set about deleting a node.

For the *maximum* value in the tree, follow the same procedure, but go from right child to right child until you find a node with no right child. This node is the maximum. The code is the same except that the last statement in the loop is

```
current = current.rightChild;  // go to right child
```

Deleting a Node

Deleting a node is the most complicated common operation required for binary search trees. However, deletion is important in many tree applications, and studying the details builds character.

You start by finding the node you want to delete, using the same approach we saw in `find()` and `insert()`. When you've found the node, there are three cases to consider:

1. The node to be deleted is a leaf (has no children).
2. The node to be deleted has one child.
3. The node to be deleted has two children.

We'll look at these three cases in turn. The first is easy; the second, almost as easy; and the third, quite complicated.

Case 1: The Node to Be Deleted Has No Children

To delete a leaf node, you simply change the appropriate child field in the node's parent to point to `null`, instead of to the node. The node will still exist, but it will no longer be part of the tree. This is shown in Figure 8.13.

Because of Java's garbage collection feature, we don't need to worry about explicitly deleting the node itself. When Java realizes that nothing in the program refers to the node, it will be removed from memory. (In C and C++ you would need to execute `free()` or `delete()` to remove the node from memory.)

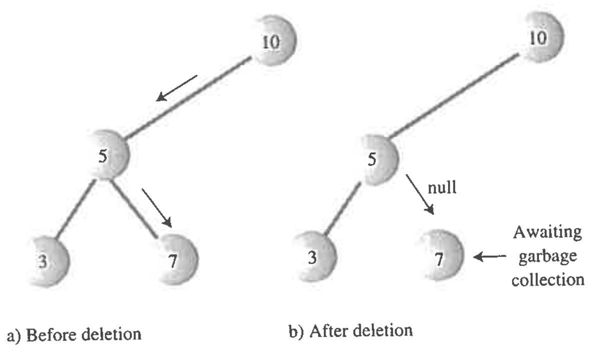


FIGURE 8.13 Deleting a node with no children.

Using the Workshop Applet to Delete a Node with No Children

Assume you're going to delete node 7 in Figure 8.13. Press the Del button and enter 7 when prompted. Again, the node must be found before it can be deleted. Repeatedly pressing Del will take you from 10 to 5 to 7. When the node is found, it's deleted without incident.

Java Code to Delete a Node with No Children

The first part of the `delete()` routine is similar to `find()` and `insert()`. It involves finding the node to be deleted. As with `insert()`, we need to remember the parent of the node to be deleted so we can modify its child fields. If we find the node, we drop out of the `while` loop with `parent` containing the node to be deleted. If we can't find it, we return from `delete()` with a value of `false`.

```
public boolean delete(int key) // delete node with given key
   {                           // (assumes non-empty list)
   Node current = root;
   Node parent = root;
   boolean isLeftChild = true;

   while(current.iData != key)        // search for node
      {
      parent = current;
      if(key < current.iData)         // go left?
         {
         isLeftChild = true;
         current = current.leftChild;
         }
      else                            // or go right?
         {
```

```
         isLeftChild = false;
         current = current.rightChild;
         }
      if(current == null)             // end of the line,
         return false;                // didn't find it
      }  // end while
   // found node to delete
   // continues...
   }
```

After we've found the node, we check first to verify that it has no children. When this is true, we check the special case of the root. If that's the node to be deleted, we simply set it to `null`; this empties the tree. Otherwise, we set the parent's `leftChild` or `rightChild` field to `null` to disconnect the parent from the node.

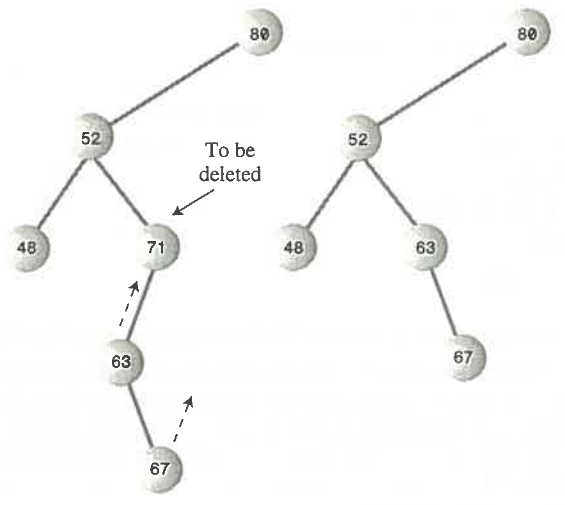
```
// delete() continued...
// if no children, simply delete it
if(current.leftChild==null &&
                              current.rightChild==null)
   {
   if(current == root)             // if root,
      root = null;                 // tree is empty
   else if(isLeftChild)
      parent.leftChild = null;     // disconnect
   else                            // from parent
      parent.rightChild = null;
   }
// continues...
```

Case 2: The Node to Be Deleted Has One Child

This second case isn't so bad either. The node has only two connections: to its parent and to its only child. You want to "snip" the node out of this sequence by connecting its parent directly to its child. This process involves changing the appropriate reference in the parent (`leftChild` or `rightChild`) to point to the deleted node's child. This situation is shown in Figure 8.14.

Using the Workshop Applet to Delete a Node with One Child

Let's assume we're using the Workshop applet on the tree in Figure 8.14 and deleting node 71, which has a left child but no right child. Press Del and enter 71 when prompted. Keep pressing Del until the arrow rests on 71. Node 71 has only one child, 63. It doesn't matter whether 63 has children of its own; in this case it has one: 67.



a) Before deletion b) After deletion

FIGURE 8.14 Deleting a node with one child.

Pressing Del once more causes 71 to be deleted. Its place is taken by its left child, 63. In fact, the entire subtree of which 63 is the root is moved up and plugged in as the new right child of 52.

Use the Workshop applet to generate new trees with one-child nodes, and see what happens when you delete them. Look for the subtree whose root is the deleted node's child. No matter how complicated this subtree is, it's simply moved up and plugged in as the new child of the deleted node's parent.

Java Code to Delete a Node with One Child

The following code shows how to deal with the one-child situation. There are four variations: The child of the node to be deleted may be either a left or right child, and for each of these cases the node to be deleted may be either the left or right child of its parent.

There is also a specialized situation: the node to be deleted may be the root, in which case it has no parent and is simply replaced by the appropriate subtree. Here's the code (which continues from the end of the no-child code fragment shown earlier):

```
// delete() continued...
// if no right child, replace with left subtree
else if(current.rightChild==null)
   if(current == root)
```

```
      root = current.leftChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.leftChild;
   else                           // right child of parent
      parent.rightChild = current.leftChild;

// if no left child, replace with right subtree
else if(current.leftChild==null)
   if(current == root)
      root = current.rightChild;
   else if(isLeftChild)           // left child of parent
      parent.leftChild = current.rightChild;
   else                           // right child of parent
      parent.rightChild = current.rightChild;
// continued...
```

Notice that working with references makes it easy to move an entire subtree. You do this by simply disconnecting the old reference to the subtree and creating a new reference to it somewhere else. Although there may be lots of nodes in the subtree, you don't need to worry about moving them individually. In fact, they "move" only in the sense of being conceptually in different positions relative to the other nodes. As far as the program is concerned, only the reference to the root of the subtree has changed.

Case 3: The Node to Be Deleted Has Two Children

Now the fun begins. If the deleted node has two children, you can't just replace it with one of these children, at least if the child has its own children. Why not? Examine Figure 8.15, and imagine deleting node 25 and replacing it with its right subtree, whose root is 35. Which left child would 35 have? The deleted node's left child, 15, or the new node's left child, 30? In either case 30 would be in the wrong place, but we can't just throw it away.

We need another approach. The good news is that there's a trick. The bad news is that, even with the trick, there are a lot of special cases to consider. Remember that in a binary search tree the nodes are arranged in order of ascending keys. For each node, the node with the next-highest key is called its *inorder successor*, or simply its successor. In Figure 8.15a, node 30 is the successor of node 25.

Here's the trick: To delete a node with two children, *replace the node with its inorder successor*. Figure 8.16 shows a deleted node being replaced by its successor. Notice that the nodes are still in order. (There's more to it if the successor itself has children; we'll look at that possibility in a moment.)

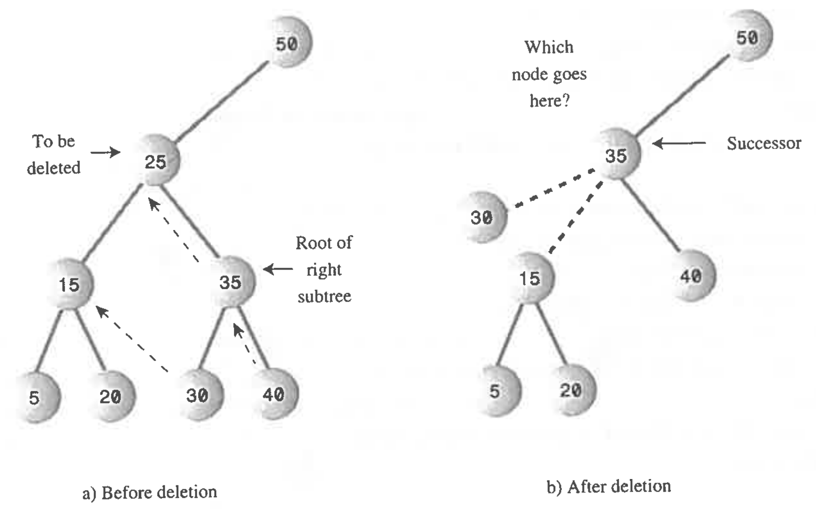


FIGURE 8.15 Cannot replace with subtree.

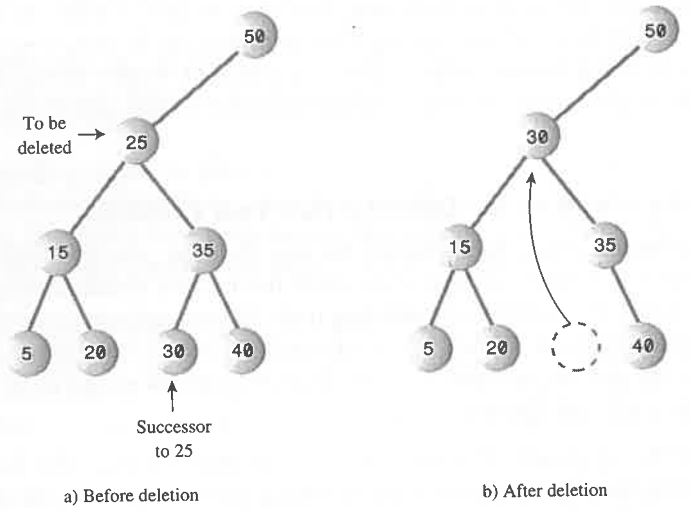


FIGURE 8.16 Node replaced by its successor.

Finding the Successor

How do you find the successor of a node? As a human being, you can do this quickly (for small trees, anyway). Just take a quick glance at the tree and find the next-largest number following the key of the node to be deleted. In Figure 8.16 it doesn't take

long to see that the successor of 25 is 30. There's just no other number that is greater than 25 and also smaller than 35. However, the computer can't do things "at a glance"; it needs an algorithm. Here it is:

First, the program goes to the original node's right child, which must have a key larger than the node. Then it goes to this right child's left child (if it has one), and to this left child's left child, and so on, following down the path of left children. The last left child in this path is the successor of the original node, as shown in Figure 8.17.

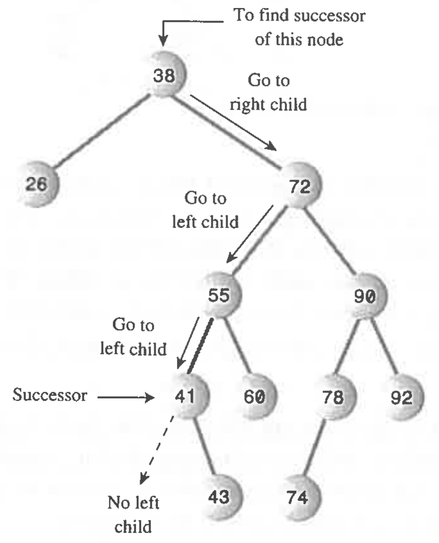


FIGURE 8.17 Finding the successor.

Why does this algorithm work? What we're really looking for is *the smallest of the set of nodes that are larger than the original node.* When you go to the original node's right child, all the nodes in the resulting subtree are greater than the original node because this is how a binary search tree is defined. Now we want the smallest value in this subtree. As we learned, you can find the minimum value in a subtree by following the path down all the left children. Thus, this algorithm finds the minimum value that is greater than the original node; this is what we mean by its successor.

If the right child of the original node has no left children, this right child is itself the successor, as shown in Figure 8.18.

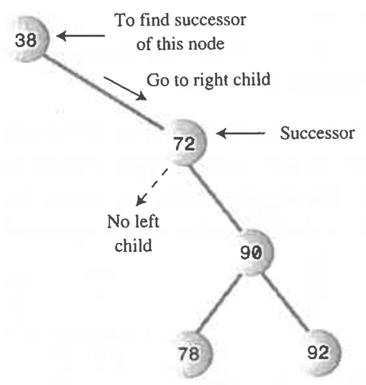


FIGURE 8.18 The right child is the successor.

Using the Workshop Applet to Delete a Node with Two Children

Generate a tree with the Workshop applet, and pick a node with two children. Now mentally figure out which node is its successor, by going to its right child and then following down the line of this right child's left children (if it has any). You may want to make sure the successor has no children of its own. If it does, the situation gets more complicated because entire subtrees are moved around, rather than a single node.

After you've chosen a node to delete, click the Del button. You'll be asked for the key value of the node to delete. When you've specified it, repeated presses of the Del button will show the red arrow searching down the tree to the designated node. When the node is deleted, it's replaced by its successor.

Let's assume you use the Workshop applet to delete the node with key 30 from the example shown earlier in Figure 8.15. The red arrow will go from the root at 50 to 25; then 25 will be replaced by 30.

Java Code to Find the Successor

Here's some code for a method `getSuccessor()`, which returns the successor of the node specified as its `delNode` argument. (This routine assumes that `delNode` does indeed have a right child, but we know this is true because we've already determined that the node to be deleted has two children.)

```
// returns node with next-highest value after delNode
// goes to right child, then right child's left descendants

private node getSuccessor(node delNode)
   {
   Node successorParent = delNode;
```

```
   Node successor = delNode;
   Node current = delNode.rightChild;   // go to right child
   while(current != null)               // until no more
      {                                 // left children,
      successorParent = successor;
      successor = current;
      current = current.leftChild;      // go to left child
      }
                                        // if successor not
   if(successor != delNode.rightChild)  // right child,
      {                                 // make connections
      successorParent.leftChild = successor.rightChild;
      successor.rightChild = delNode.rightChild;
      }
   return successor;
   }
```

The routine first goes to `delNode`'s right child and then, in the `while` loop, follows down the path of all this right child's left children. When the `while` loop exits, `successor` contains `delNode`'s successor.

When we've found the successor, we may need to access its parent, so within the `while` loop we also keep track of the parent of the current node.

The `getSuccessor()` routine carries out two additional operations in addition to finding the successor. However, to understand them, we need to step back and consider the big picture.

As we've seen, the successor node can occupy one of two possible positions relative to `current`, the node to be deleted. The successor can be `current`'s right child, or it can be one of this right child's left descendants. We'll look at these two situations in turn.

Successor Is Right Child of `delNode`

If `successor` is the right child of `current`, things are simplified somewhat because we can simply move the subtree of which successor is the root and plug it in where the deleted node was. This operation requires only two steps:

1. Unplug `current` from the `rightChild` field of its parent (or `leftChild` field if appropriate), and set this field to point to `successor`.
2. Unplug `current`'s left child from `current`, and plug it into the `leftChild` field of `successor`.

Here are the code statements that carry out these steps, excerpted from `delete()`:

1. `parent.rightChild = successor;`
2. `successor.leftChild = current.leftChild;`

This situation is summarized in Figure 8.19, which shows the connections affected by these two steps.

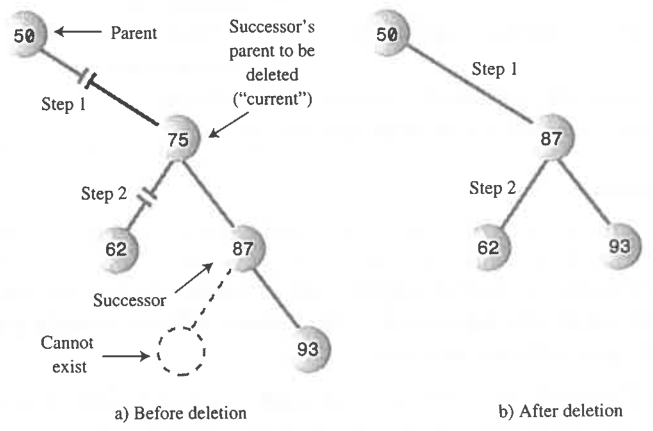


FIGURE 8.19 Deletion when the successor is the right child.

Here's the code in context (a continuation of the `else-if` ladder shown earlier):

```
// delete() continued
else  // two children, so replace with inorder successor
   {
   // get successor of node to delete (current)
   Node successor = getSuccessor(current);

   // connect parent of current to successor instead
   if(current == root)
      root = successor;
   else if(isLeftChild)
      parent.leftChild = successor;
   else
      parent.rightChild = successor;
   // connect successor to current's left child
   successor.leftChild = current.leftChild;
   }  // end else two children
// (successor cannot have a left child)
```

```
return true;
}  // end delete()
```

Notice that this is—finally—the end of the `delete()` routine. Let's review the code for these two steps:

- Step 1: If the node to be deleted, `current`, is the root, it has no parent so we merely set the root to the successor. Otherwise, the node to be deleted can be either a left or right child (Figure 8.19 shows it as a right child), so we set the appropriate field in its parent to point to `successor`. When `delete()` returns and `current` goes out of scope, the node referred to by `current` will have no references to it, so it will be discarded during Java's next garbage collection.
- Step 2: We set the left child of `successor` to point to `current`'s left child.

What happens if the successor has children of its own? First of all, *a successor node is guaranteed not to have a left child.* This is true whether the successor is the right child of the node to be deleted or one of this right child's left children. How do we know this?

Well, remember that the algorithm we use to determine the successor goes to the right child first and then to any left children of that right child. It stops when it gets to a node with no left child, so the algorithm itself determines that the successor can't have any left children. If it did, that left child would be the successor instead.

You can check this out on the Workshop applet. No matter how many trees you make, you'll never find a situation in which a node's successor has a left child (assuming the original node has two children, which is the situation that leads to all this trouble in the first place).

On the other hand, the successor may very well have a right child. This isn't much of a problem when the successor is the right child of the node to be deleted. When we move the successor, its right subtree simply follows along with it. There's no conflict with the right child of the node being deleted because the successor *is* this right child.

In the next section we'll see that a successor's right child needs more attention if the successor is not the right child of the node to be deleted.

Successor Is Left Descendant of Right Child of `delNode`

If `successor` is a left descendant of the right child of the node to be deleted, four steps are required to perform the deletion:

1. Plug the right child of `successor` into the `leftChild` field of the successor's parent.
2. Plug the right child of the node to be deleted into the `rightChild` field of `successor`.

3. Unplug `current` from the `rightChild` field of its parent, and set this field to point to `successor`.
4. Unplug `current`'s left child from `current`, and plug it into the `leftChild` field of `successor`.

Steps 1 and 2 are handled in the `getSuccessor()` routine, while 3 and 4 are carried out in `delete()`. Figure 8.20 shows the connections affected by these four steps.

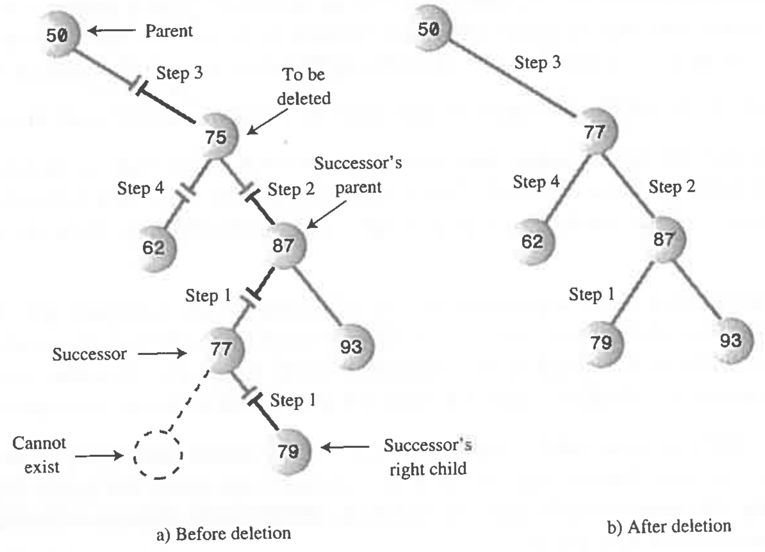


FIGURE 8.20 Deletion when the successor is the left child.

Here's the code for these four steps:

1. `successorParent.leftChild = successor.rightChild;`
2. `successor.rightChild = delNode.rightChild;`
3. `parent.rightChild = successor;`
4. `successor.leftChild = current.leftChild;`

(Step 3 could also refer to the left child of its parent.) The numbers in Figure 8.20 show the connections affected by the four steps. Step 1 in effect *replaces the successor with its right subtree*. Step 2 keeps the right child of the deleted node in its proper place (this happens automatically when the successor is the right child of the deleted node). Steps 1 and 2 are carried out in the `if` statement that ends the `getSuccessor()` method shown earlier. Here's that statement again:

```
                                       // if successor not
if(successor != delNode.rightChild)    // right child,
   {                                   // make connections
   successorParent.leftChild = successor.rightChild;
   successor.rightChild = delNode.rightChild;
   }
```

These steps are more convenient to perform here than in `delete()`, because in `getSuccessor()` we can easily figure out where the successor's parent is while we're descending the tree to find the successor.

Steps 3 and 4 we've seen already; they're the same as steps 1 and 2 in the case where the successor is the right child of the node to be deleted, and the code is in the `if` statement at the end of `delete()`.

Is Deletion Necessary?

If you've come this far, you can see that deletion is fairly involved. In fact, it's so complicated that some programmers try to sidestep it altogether. They add a new Boolean field to the `node` class, called something like `isDeleted`. To delete a node, they simply set this field to `true`. Then other operations, like `find()`, check this field to be sure the node isn't marked as deleted before working with it. This way, deleting a node doesn't change the structure of the tree. Of course, it also means that memory can fill up with "deleted" nodes.

This approach is a bit of a cop-out, but it may be appropriate where there won't be many deletions in a tree. (If ex-employees remain in the personnel file forever, for example.)

The Efficiency of Binary Trees

As you've seen, most operations with trees involve descending the tree from level to level to find a particular node. How long does it take to do this? In a full tree, about half the nodes are on the bottom level. (More accurately, if it's full, there's one more node on the bottom row than in the rest of the tree.) Thus, about half of all searches or insertions or deletions require finding a node on the lowest level. (An additional quarter of these operations require finding the node on the next-to-lowest level, and so on.)

During a search we need to visit one node on each level. So we can get a good idea how long it takes to carry out these operations by knowing how many levels there are. Assuming a full tree, Table 8.2 shows how many levels are necessary to hold a given number of nodes.