shown at line 22, initializes all of the data fields of the `BinaryNode`. Notice that these constructors require initialization of either both child references or neither child reference. In this way, we avoid a partially initialized object.

Access to data fields is allowed to other package classes, so we do not provide accessors or mutators. The `duplicate` method, declared at line 36, is used to replicate a copy of the tree rooted at the current node. The routines `size` and `height`, declared at lines 26 and 28, compute the named properties for the node referenced by parameter `t`. These routines are implemented in Section 17.3. We also provide, at lines 30 to 34, routines that print out the contents of a tree rooted at the current node using various recursive traversal strategies. Tree traversals are discussed in Section 17.4. Why do we pass a parameter for `size` and `height` but use the current object for the traversals and `duplicate`? There is no particular reason; it is a matter of style. Both styles are shown here. The implementations will show that the difference between them is when the required test for a `null` tree is performed.

Many of the `BinaryNode` routines are recursive. The `BinaryTree` methods use the `BinaryNode` routines on the `root`.

This section describes the implementation of the `BinaryTree` class. A separate class, `BinaryNode`, is provided to simplify the implementation of some of the recursive routines. The `BinaryTree` class skeleton is shown in Figure 17.13 (page 466). For the most part, the routines are simple to implement because they call `BinaryNode` methods. Line 44 declares the only data field, a reference to the root node.

The `BinaryNode` class is implemented separately from the `BinaryTree` class. The only data field in the `BinaryTree` class is a reference to the root node.

Two constructors are provided. The one at line 19 creates an empty tree, while the one at line 21 creates a one-node tree. Routines to traverse the tree are declared at lines 24 to 29. They apply a `BinaryNode` method to the `root`, after verifying that the tree is not empty. An alternative traversal strategy that can be implemented is *level-order traversal*. All these traversal routines are discussed in Section 17.4. Routines to make an empty tree and to test for emptiness are given, with their implementations, at lines 32 and 30, respectively. Two routines remain in the class.

The `duplicate` method is defined at lines 41 and 42. After testing for aliasing, we call the `duplicate` method in `BinaryNode` to get a copy of `rhs`'s tree. Then we assign the result as the root of the tree. Notice that before we can apply the `BinaryNode` method to the node referenced by `root`, we must verify that the `root` is not `null`.

Before we can apply the `BinaryNode` method to the node referenced by `root`, we must verify that `root` is not `null`.

The last method in the class is the `merge` routine. `merge` uses two trees — `t1` and `t2` — and an element to create a new tree, with the element at the root and the two existing trees as left and right subtrees. In principle, this is a one-line routine:

```
root = new BinaryNode( rootItem, t1.root, t2.root );
```

`merge` is a one-line routine in principle. However, we must also handle aliasing, making sure that a node is not in two trees, and do error checking.

If things were always this simple, programmers would be unemployed. Fortunately for our careers, there are complications. Figure 17.14 (page 467) shows the result of the simple one-line `merge`. A problem becomes apparent: Nodes in `t1` and `t2`'s trees are now in two trees (their original trees and the merged result). This sharing could be a problem if we want to remove or otherwise alter subtrees (because multiple subtrees may be removed or altered unintentionally).