The typical implementation of a binary tree, as shown below, with pointers being **null** if subtree left or right are null, is not OO-oriented. This implementation can lead to code with lots of tests to determine whether a subtree is empty or not. We see this in the implementation of method contains to the right.

/\*\* = ''this tree contains d''. \*/

**public** **int** contains(T d) {

**return** data.equals(d) ||

(left != **null** && left.contains(d)) ||

(right != **null** && right.contains(d));

}

**public class** TreeNode<T> {

**private** T data;

**private** TreeNode<T> left; // null if empty

**private** TreeNode<T**>** right; // null if empty

...

}

We now give the idea behind an OO implementation of a binary tree, with different object types for an empty tree and a nonempty tree. We start off to the right with an interface that contains two abstract methods, the size of a tree and a method that returns true iff a node of the tree contains the value d.

**public interface** BTree<T> {

/\*\* Number of nodes in the tree \*/

**int** size();

/\*\* = "this tree contains d" \*/

**boolean** contains(T d);

}

To the right is class Empty, which implements interface BTree. Since this class is for an empty tree, its size is 0 and it certainly does not contain a value! Methods size and contains are simple. We don't have to write a constructor; Java will insert the constructor public Empty(){};

**public class** Empty<T>  
 **implements** Btree<T> {

**public** **int** size() { **return** 0; }

**public** **boolean** contains(T d)  
 { **return** **false**; }

}

Below, we give class Node, whose instances represent a node of a non-empty binary tree. The OO approach eliminates the need for tests to determine whether a node is empty or not. Compare method contains with method contains given at the top of the page using the conventional non-OO implementation of a binary tree. See how methods become simpler when an OO approach is used.

/\*\* A tree with a root value and left and right subtrees \*/

**public class** Node<T> implements Btree<T> {

**private** T data; // not null

**private** Tree<T> left, right; // not null  
 /\*\* Constructor: Tree with root value d, left tree le,  
 and right tree ri.  
 Precondition: le and ri are not null. \*/  
 Node(T d, Tree<T> le, Tree<T> ri)  
 { data= d; left= le; right= ri; }

**public** T rootValue() { **return** data; }

**public** Tree<T> left() { **return** left; }

....**public** Tree<T> right() { **return** right; }

@Override  
 **public** **int** size() {  
 return 1 + left.size() + right.size();  
 }

@Override  
 **public** **boolean** contains(T d) {  
 **return** data.equals(d) ||   
 left.contains(d) ||   
 right.contains(d);  
} }