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.

```
public interface BTree<T> {
    /** Number of nodes in the tree */
    int size();
    /** = "this tree contains d" */
    boolean contains(T d);
}
```

Below is the code for the non-OO implementation of a binary tree.

```
public class Empty<T> implements BTree<T> {
    public int size() {
        return 0;
    }
    public boolean contains(T d) {
        return false;
    }
}
```

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();

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.

```
/** = "this tree contains d" */
public int contains(T d) {
    return data.equals(d) ||
           (left != null && left.contains(d)) ||
           (right != null && right.contains(d));
}
```

```
/** A tree with a root value and left and right subtrees */
public class TreeNode<T> {
    private T data;
    private TreeNode<T> left;  // null if empty
    private TreeNode<T> right; // null if empty
    ...
}
```

```
/** = "this tree contains d" */
public class Node<T> implements BTree<T> {
    private T data; // not null
    private TreeNode<T> left, right; // null if empty
    /** Constructor: Tree with root value d, left tree le, and right tree ri. */
    Node(T d, TreeNode<T> le, TreeNode<T> ri)
    { data= d; left= le; right= ri; }
    public T rootValue() { return data; }  
    @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);
    }
}
```