This development of a recursive method uses the *Mañana Principle* (look it up in JavaHyperText), which says: When, during implementation of a method, you wish you had a certain support method, write your code as if you had it and implement the method later. The support method shows again how useful it is to think of adding a second method, a recursive one, with a new parameter. Finally, this development makes clear how important it is to respect the recursive definition of a binary tree —we’ll point that out later what this means.

We deal with a binary tree whose nodes are instances of class TreeNode, given to the right.

/\*\* An instance is a node of a binary tree. \*/

**public** **class** TreeNode {

**int** val; // Value associated with this node.

TreeNode left; // Left child of this node   
 // ---null if none.

TreeNode right; // Right child of this node  
 // ---null if none.

}

We develop a function isBST to determine whether a tree is a Binary Search Tree —a BST.

/\*\* = “Tree n is a BST” \*/  
**public** **static** **boolean** isBST(TreeNode n)

We define BST, written in terms of class TreeNode:

A binary tree is a BST iff for each node n of the tree, all val fields in subtree n.left are less than n.val and all val fields in subtree n.right are greater than n.val.

By the definition, an empty binary tree is a BST.

Since we are going to write a recursive function, it may help to rewrite the definition of a BST recursively:

 A binary tree with root n is a BST iff  
 (1) n.left is null or a BST,  
 (2) all val fields in subtree n.left are less than n.val,  
 (4) n.right is null or a BST,  
 (3) all val fields in subtree n.right are greater than n.val,

We place a diagram of a binary tree on the right to give guidance in thinking about implementing the function.

**Fight a bad tendency**

There must be a test n.left.val < n.val. (if n.left != **null**). The tendency is to write that test almost immediately. Fight that tendency! If you don’t, you’ll be complicating the method by having three tests in the method —testing n.val, n.left.val, and n.right.val.

Look at the above diagram of a binary tree. A binary tree is a node together with a left binary tree and a right binary tree. The diagram doesn’t show you anything about the two subtrees. So, when processing node n, it may a bad idea to “look inside” the two subtrees at n.left.val and n.right.val.

Instead, think of having a second function isBST(n, max), write a call isBST(n.left, n.val) and have that call verify that n.val < max. In the same way, we will need a third parameter min to verify that all values in subtree n.right are greater than min.

This leads us to use the *Mañana Principle* and stub in a second function isBST with three parameters:

/\*\* = “Tree n is a BST and all val fields in it satisfy min < val < max”\*/  
 **private** **static** **boolean** isBST (TreeNode n, **int** min, **int** max) {  
 **return false**;  
 }

We can now write function isBST with no parameters as shown to the right. It simply calls the function isBST with three parameters, and this second function will be recursive. In the earlier part of this JavaHyperText where we discussed the possible need for extra parameters in recursive methods, we also discussed the standard practice of having two methods like this.

/\*\* = “Tree n is a BST” \*/  
**public** **static** **boolean** isBST(TreeNode n) {

**return** isBST(n, Integer.MIN\_VALUE,  
 Integer.MAX\_VALUE);  
}

**A small change**

Suppose we want to check whether a binary tree with root n is a BST. We write the call:

isBST(n)

This call immediately results in the call

isBST(n, Integer.MIN\_VALUE, Integer.MAX\_VALUE);

and because of the requirement that min < val < max, the smallest and largest values of type **int** cannot be in a BST. That’s not good, so we change the spec of the two-parameter function to use ≤ instead of <:

/\*\* = “Tree n is a BST and all val fields in it satisfy min ≤ val ≤ max”\*/  
 **private** **static** **boolean** isBST (TreeNode n, **int** min, **int** max)

**Implementing the 3-parameter function**

We again keep our diagram of a binary tree handy to help guide us in implementing the 3-parameter function, whose spec is above. We proceed in four steps.

1. The empty tree is a binary tree:

**if** (n == **null**) **return** **true**;

1. The spec of isBST (n, min, max) requires that the val fields of all nodes of n satisfy min ≤ val ≤ max. Looking at the diagram of a binary tree, node n has to be checked, and we do that first:

**if** (n.val < min || max < n.val) **return** **false**;

1. If subtree n.left is not empty, then the following properties must be satisfied:
2. Subtree n.left is a BST (by the first part of the spec and part (1) of the definition of a BST)
3. All val fields in subtree n.left satisfy min ≤ val ≤ max (by the second part of the spec)
4. All val fields in subtree n.left satisfy val < n.val (for tree n to be a BST)

Properties (2) and (3) can be merged into a single property:

1. All val fields in subtree n.left satisfy min ≤ val ≤ n.val-1

Based on properties (1) and (4) a call tells us whether the left subtree has the necessary properties:

isBST(n.left, min, n.val-1);

1. The right subtree is handled in a similar fashion:

isBST(n.right, val+1, max);

This results in method isBST given in the box below.

/\*\* = “Tree n is a BST and all val fields in it satisfy min ≤ val ≤ max” \*/  
**private** **static** **boolean** isBST (TreeNode n, **int** min, **int** max) {

**if** (n == **null**) **return** **true**.

**if** (n.val < min || max < n.val) **return** **false**;

**return** isBST(n.left, min, n.val-1) && isBST(n.right, val+1, max);

}

What a beautiful, simple function!