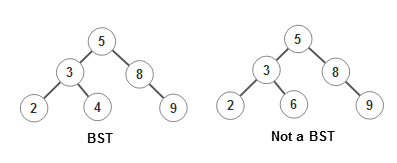
Binary search tree (BST)

A binary search tree, or BST, is a binary tree that satisfies:

1. Each node contains a value.
2. For each node, all the values in its left subtree are less than its value.
3. For each node, all the values in its right subtree are greater than its value.

An example of a BST appears to the right above. It’s important to realize that *all* nodes in the left subtree must be less than the node’s value. The rightmost binary tree above is not a BST because the value 6 is not less than 5.

# Here’s one important consequence of the definition of a BST: *A BST cannot contain duplicate values*. This is a consequence of properties 2 and 3.

# Below, in discussing algorithms that process BSTs, we’ll assume that the nodes are of class Node and that the class contains the usual fields: int val, Node left, and Node right.

**Advantages of a BST**

/\*\* = “t is not null and v is in tree t” \*/

public static boolean isIn(Node t, int v) {  
 // inv: if v is in original tree, v in t

# while (t != null) { if (v == t.val) return true; t= v < t.val ? t.left : t.right; } return false;

}

# A BST has two obvious advantages over a normal binary tree.

# (1) When searching for a value v, if v is less than than the root value, one has to look only in the left subtree. Similarly if v is greater than the root value, one has to look only in the right subtree. Therefore, the maxi­mum number of nodes to be tested is 1 + (height of the tree). If the tree is balanced, searching for a value takes time O(log n) for a tree with n nodes.

# To the right above, function isIn searches a BST for a value. It uses a loop instead of recursion. It is just as simple as a recursive version, and it is faster because no method calls are necessary.

/\*\* Print t’s values, sorted.

\* Precondition: t is not null. \*/

public static boolean print(Node t) {  
 if (t.left != null) print(t.left);

# System.out.println(t.val); if (t.right != null) print(t.right);

}

(2) We can enumerate the values in a BST in sorted (ascending) order simply by performing an inorder traversal. See the method to the right.

**Inserting a value into a BST**

# Consider inserting 6 into the BST at the top of the page. 6 > 5, so 6 belongs in 5’s right subtree. 6 < 8, so 6 belongs in 8’s left subtee. Since 8 doesn’t have a left subtree, insert 6 as 8’s left subtree.

/\*\* Insert v into BST t if it is not there.

\* Return true if it was inserted, false otherwise

\* Precondition: t is not null. \*/

public static boolean insert(Node t, int v) {

// inv: t != null; if v is in original tree, v in t

while (true) {

if (t.val == v) return false;

# if (v < t.val) {

if (t.left != null) t= t.left;

else { t.left= new Node(v); return true; }

} else { // v > t.val

if (t.right != null) t= t.right;

else { t.right= new Node(v); return true; }

}

}

}

# }

# if (t.right != null) print(t.right);

}

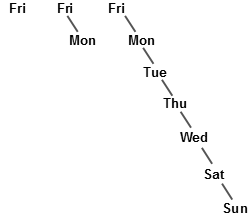
Consider inserting 10 into the BST. 10 > 5, so 10 belongs in 5’s right subtree. 10 > 8, so 10 belongs in 8’s right subtree. 10 > 9, so 10 belongs in 9’s right subtree. Since 9 doesn’t have a right subtree, insert 10 as 9’s right subtree.

Evidently, inserting v into BST t requires searching t for v; when that search ends without finding v, v is added.

To the right we write insert. We started with the body of isIn(t, v) and massaged it to fit the new specification.

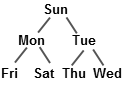
Note the new condition in the loop invariant: t is not null. The loop has condition true because the repetend will return when v is found in t or v is added to t.

The main change comes when testing which subtree v belongs in (left or right). If that subtree exists, assign the subtree to t. If that subtree doesn’t exist, create a new node with value v and null subtrees and make it that subtree.

**The problem of balancing a BST**

Suppose we insert the days of the week into an empty BST (whose values are strings) in alphabetical order. To the right, we show a BST with one node, Fri (day), inserted, then a BST with Mon inserted into that BST, and finally the BST with all the days of the week inserted. Because we inserted the days in alphabetical order, the BST looks like a linked list.

One of the main advantages tauted of a BST is the efficient search for a value. But in such an unbalanced tree, the worst-case time search time for a BST with n nodes is O(n).

On the other hand, if we inserted the values into the BST in a different order, we could produce the BST tree that appears to the left. And in such a complete tree, thus the worst-case search time, as well as the height of the tree, is O(log n).

The problem is that inserting nodes into a BST (or even deleting nodes) may make it become very imbalanced, so that its height is nowhere near the logarithm of its size. Further, there is no simple and efficient way to make it balanced again. Because of this, computer scientists in the 1960s, 1970s, later explored extensions to BSTs that kept them balanced. We explore them in a separate document in JavaHyperText.

The term *balanced* needs explaining! Several authors who developed an extension to BSTs came up with their own definition of balanced to fit their needs, usually with the goal to have the tree height be is roughly the logarithm of its size. Two definitions of *balanced* appear to stand out:

1. A binary tree is weight-balanced if for each node the numbers of inner nodes in its left subtree and right subtree differ by at most 1.
2. A binary tree is height-balanced if for each node the heights of its left and right subtree differ by at most 1.

The tree to the right is height-balanced. It is not weight-balanced because the left subtree of the root as 3 inner nodes but the right subtree of the root has only 1 inner node. It has been shown that a weight-balanced tree is also height-balanced.

Don’t be concerned with formal definitions of balanced. Just use the fact that *balanced* should imply that the height of a tree is close to the logarithms of its size.

**Determining whether a binary tree is a BST**

# Consider determining whether a binary tree t with int values is a BST. Let’s develop a recursive boolean function isBST(t, …).

/\*\* = “t is a BST with values in the range h..k.”

\* Precondition: t is not null.\*/

public static boolean isBST(Node t, int h, int k) {  
 if (t.val < h || k < t.val) return false;

# if (t.left != null && !isBST(t.left, h, t.val–1))

# return false;

# return t.right != null && !isBST(t.right, t.val+1, k); }

# Property 3 of a BST —all values in the right subtree of a node are greater than the node’s value— requires that a recursive call on the right subtree be given a minimum value for nodes in the subtree. For example, if the root value is 50, all nodes in the right subtree have to be ≥ 51. Therefore, function isBST needs a parameter h (say) with the requirement that all nodes values be ≥ h.

# In the same way, because of property 2, isBST needs a parameter k (say) with the requirement that all nodes values be ≤ k. Therefore, we envision a method with specification given in the method above.

A call on the root of a binary tree will be

isBST(root, Integer.MIN\_VALUE, Integer.MAX\_VALUE)

Now think of writing the body of function isBST. First, return false if the root is not in range h..k.

Second, if the left subtree is not null, (1) it must be a BST and (2) all values in it have to be in the range h..t.val -1. The second if statement checks these conditions and returns false if they are not true.

Third, the last statement, a return statement, checks the same kind of conditions for the right subtree.

# How did we think of this function and its specification? By looking at the definition of a BST. The need for parameters h and k are based on that definition.