

Introduction to Binary Search Trees (BST)

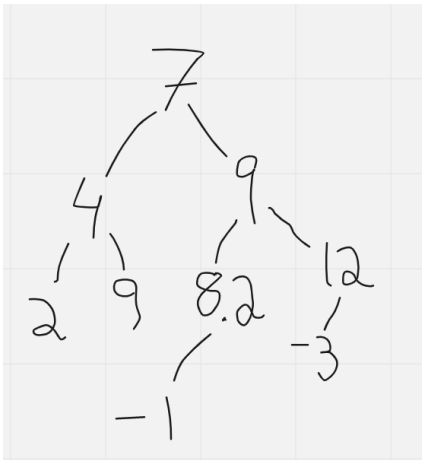
A special kind of binary tree which satisfies the following properties for every node:

- The node's left child is less than the node. (Could also be equal to if duplicates are allowed).
- The node's right child is greater than the node.

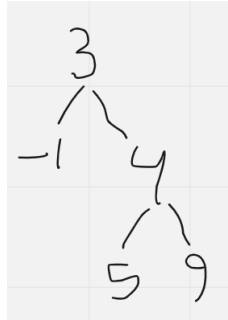
Any tree that satisfies this is valid BST.

This means that the tree stores the values in sorted order.

Determine if the following are binary search trees, and classify the tree by a special type if possible:



Yes. Looking at each node, the left child is always less than the node, and the right child is always greater than the node.
This tree is balanced.

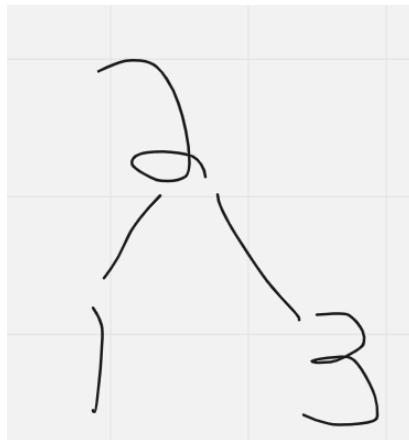
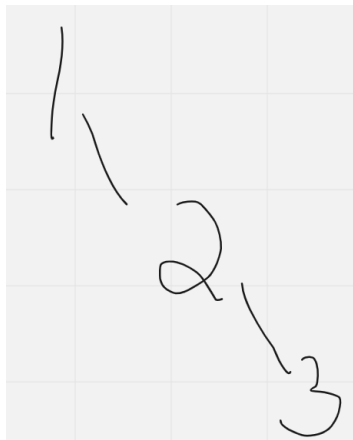


No. Although the root node satisfies the property, the left child of node(4) is node(5) and $5 > 4$.
This tree is full and balanced.

Notice you can arrange a set of values into multiple valid binary search tree configurations:

Add 2, 1, and 3 to an empty binary search tree.

Two valid answers:

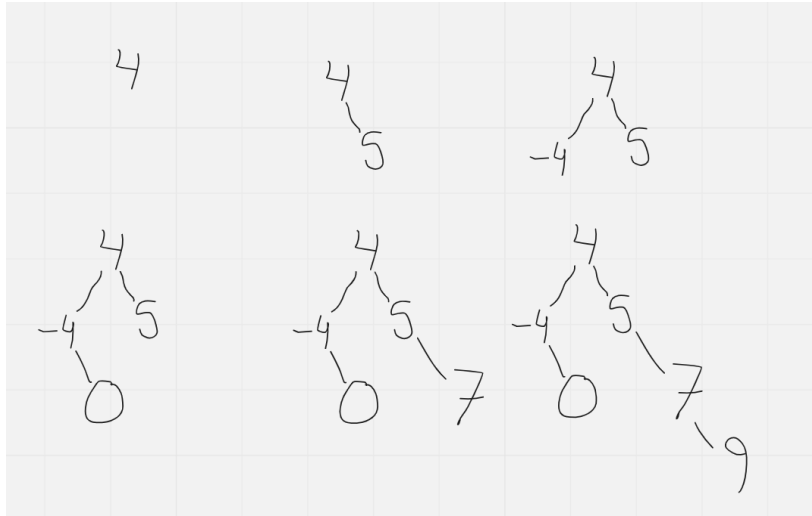


However, if we insert the values in the input order, and obey the following extra property, only one BST will be possible.

Extra property:

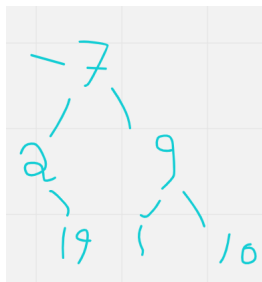
- For every node X, every single node in the left subtree has a lesser value than X
- For every Node X, every single node in the right subtree has a greater value than X.

Insert 4,5,-4, 0, 7, 9 into an empty BST:



Simplifications we make for our BST:

From now on, all trees are assumed to be built in the above way. Therefore, the following kind of tree in blue would not be considered, even though it is a valid BST:



In addition, we will assume that a given value never repeats in the tree - it appears at most once.

Time Complexity of Searching a BST

Use this idea to find a value *target* in a BST:

- Visit a node.
 - If the node value is target, you found it!
 - If the node value is less than target, then by BST properties the value must be in the right subtree
 - If the node value is greater than the target, then by BST properties the value must be in the left subtree.
- Not in the tree at all.

The algorithm:

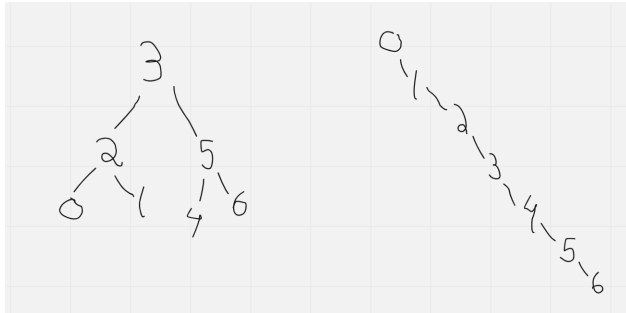
```
//accepts a target value. If the value is contained in the tree,
return that node. Otherwise returns null.
//Assume the root of the tree is called overallRoot
public IntTreeNode find(int target){
return find(target, overallRoot);
}
//helper method
public IntTreeNode find(int target, IntTreeNode node) {
if(node == null) {
    return null;
}
if(node.data == target) {
    return node;
}
if(node.data < target) {
    if(find(target, node.rightchild) != null) {
        return find(target, node.rightchild);
    }
}
else {
    if(find(target, node.leftchild) != null) {
        return find(target, node.leftchild);
    }
}
//if not in the left or right subtree, not in the tree at all.
return null;
}
```

The time complexity varies on how the tree is set up:

Fastest Time complexity of searching a BST : $O(\log(n))$. This is for a BST that is balanced or very close to balanced. The idea is that we cut the search space in half every time because we only look at the left subtree or the right subtree.

Slowest Time Complexity: A valid BST could basically look like a sorted array/arraylist, and the time complexity of searching would be $O(n)$.

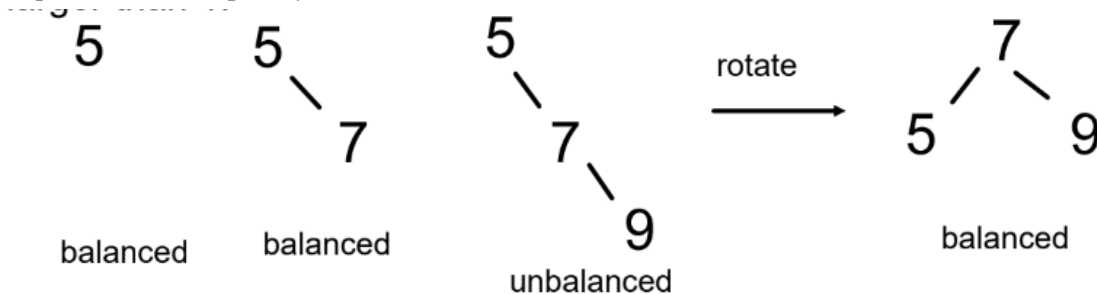
The greater the height difference between left/right subtree, the worse the time complexity.



The picture is an extreme example. A valid BST could basically look like a sorted list. If you tried to search for the number “5”, you would need to traverse through far fewer nodes in the left tree (3→ 5) as compared to the right tree (0 → 1 → 2 → 3 → 4 → 5).

Optimizing Time Complexity of Searching a BST

Some kind of Binary Tree implementations, like AVL, will “rotate” a tree as soon as it is detected to be unbalanced. They rotate it one node left or right and re-balance it. This minimizes the height difference for optimal time complexity.



Add Nodes to a BST

Once the node is added, it will have exactly one parent. Working backwards from this idea:

1. Navigate to the parent node (use the same logic from the find method for this)
 - a. Assign one of its children to be a newly created node.

If the overallRoot is null, add it there.

```
private IntTreeNode add(IntTreeNode root, int value) {
    if(root == null) {
        overallRoot = new IntTreeNode(value);
        return root;
    }
    if(value < root.data) {
        if(root.leftchild == null) {
            root.leftchild = new IntTreeNode(value);
            return root.leftchild;
        }
    }
}
```

```

    return add(root.leftchild, value);
}

if(root.rightchild == null) {
    root.rightchild = new IntTreeNode(value);
    return root.rightchild;
}
return add(root.rightchild, value);
}

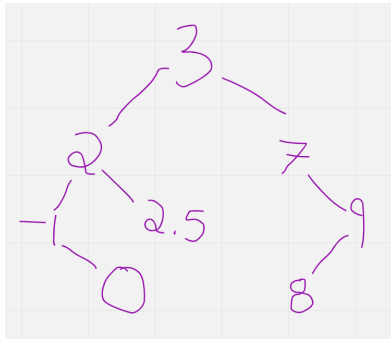
```

Remove Nodes from a BST

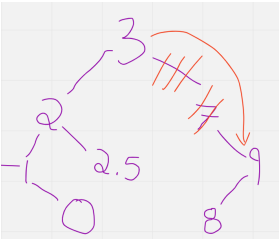
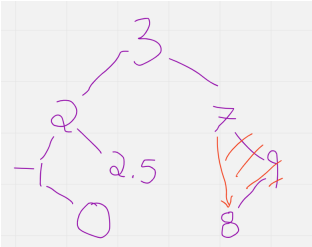
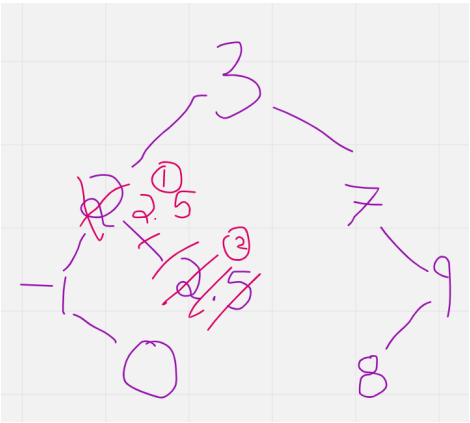
When you are removing a *target* node from the tree you need to consider how to connect the target's child(ren) to the target's parent (notice, only singular!).

There are four cases to consider for the node itself:

(Example) Before Picture:



Node to be Removed is / has...	Replace the node with...	(Example) After Picture
Leaf	null	<p>Remove Node(0)</p> <p>The target removal node was the rightchild of its parent. The new rightchild of the parent is null.</p>

Right Child	Right child	<p>Remove node(7)</p>  <p>The target removal node was the rightchild of its parent. The new rightchild of the parent is the right child of the target.</p>
Left Child	Left Child	<p>Remove node(9)</p>  <p>The target removal node was the rightchild of its parent. The new rightchild of the parent is the left child of the target.</p>
Right and Left Child	Minimum node in right subtree*	<p>Remove node(2)</p>  <p>Edit the value in the target node to be the minimum value of the right subtree. (The target removal node was) Recursivley call to remove that node in the right subtree. (Above, in total we only had to call twice: the original call and 1 recursive call).</p>

*or replace it with the maximum node from the left subtree.

Similar arguments can be made for when the target removal node is the other child of the parent.

Pseudo Code Algorithm:

```
TreeNode Remove (TreeNode node, target_value) {
    if node is null:
        return null
    if node is target {
        if node is leaf:
            return null
        if node has right child only:
            return right child
        if node has left child only:
            return left child
        if node has two children:
            Get min value in right subtree, store in var
            Remove(node, var)
            Edit node's value to be var.
            return node
    }

    if left_child is target:
        node's new left child is return value of Remove(node.left, target_value)
    if right_child is target:
        node's new right child is return value of Remove(node.right, target_value)
    //otherwise continue search
    if target_value is bigger than node:
        Remove(node.right, target_value)
    else:
        Remove(node.left, target_value)
}
```