Binary Search Trees

20 minutes

Binary Search Trees

Now that we have a solid grasp of **Binary Trees**, let's add another constraint to the data structure. A Binary **Search** Tree (BST) has an additional criteria where:

- given any node of the tree, the values in the left subtree must all be strictly less than the given node's value.
- and the values in the right subtree must all be greater than or equal to the given node's value

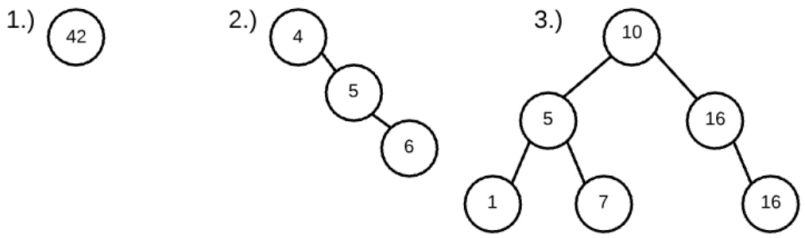
BST Definition

We can also describe a BST using a recursive definition. A **Binary Tree** is a **Binary Search Tree** if:

- the left subtree contains values less than the root
- AND the right subtree contains values greater than or equal to the root
- AND the left subtree is a Binary Search Tree
- AND the right subtree is a Binary Search Tree

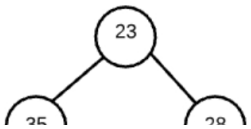
It's worth mentioning that the empty tree (a tree with 0 nodes) is indeed a BST (did someone say base case?).

Here are a few examples of BSTs:



Take a moment to verify that the above binary trees are BSTs. Note that image 2 has the same chain structure as a linked list. This will come into play later.

Below is an example of a binary tree that is **not** a search tree because a left child (35) is greater than it's parent (23):



A BST is a Sorted Data Structure

So what's the big deal with BSTs? Well, because of the properties of a BST, we can consider the tree as having an order to the values. That means the values are fully sorted! By looking at the three BST examples above, you are probably not convinced of things being sorted. This is because the ordering is encoded by an in-order traversal. Let's recall our previous `inOrderPrint` function:

```
function inOrderPrint(root) {
  if (!root) return;

  inOrderPrint(root.left);
  console.log(root.val);
  inOrderPrint(root.right);
}
```

If we run `inOrderPrint` on the three BSTs, we will get the following output:

```
BST 1: 42
BST 2: 4, 5, 6
BST 3: 1, 5, 7, 10, 16, 16
```

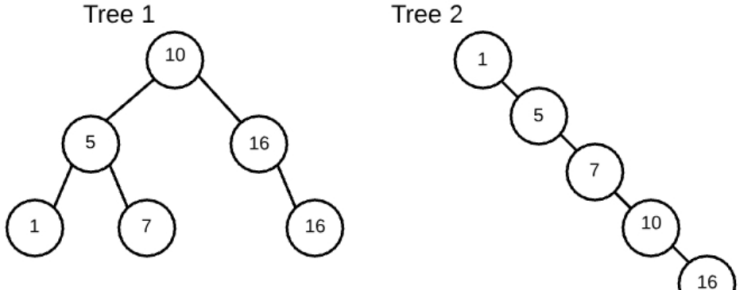
For each tree, we printed out values in increasing order! A binary search tree contains sorted data; this will come into play when we perform algorithms on this data structure.

Once you create a binary search tree class `BST`, you can call `insert` to build up the `BST` without worrying about breaking the search tree property. Here are two different trees built with the `BST` class that you'll write.

```
let tree1 = new BST();
tree1.insert(10);
tree1.insert(5);
tree1.insert(16);
tree1.insert(1);
tree1.insert(7);
tree1.insert(16);

let tree2 = new BST();
tree2.insert(1);
tree2.insert(5);
tree2.insert(7);
tree2.insert(10);
tree2.insert(16);
tree2.insert(16);
```

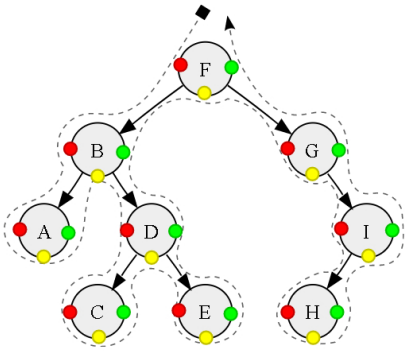
The insertions above will yield the following trees:



Are you cringing at `tree2`? You should be. Although we have the same values in both trees, they display drastically different structures because of the insertion order we used. This is why we have been referring to our `BST` implementation as **naive**. Both of these trees are Binary Search Trees, however not all BSTs are created equal. A worst case BST degenerates into a linked list. The "best" BSTs are **height balanced**, we'll explore this concept soon™.

A special traversal case

In a binary search tree, in-order traversal retrieves the keys in **ascending sorted order**. Please review the image that you saw before about tree traversal.



Note that the in-order sort represented by where the dotted line touches the yellow dots results in node visiting in the order A, B, C, D, E, F, G, H, I. In-order traversal *always* visits the nodes in sequential order in a binary search tree.

Finished with this task? Click **Mark as Complete** to continue to the next page!