

Binary Search Trees

Introduction

- These are the specific types of binary trees.
- These are inspired by the binary search algorithm.
- Time complexity on insertion, deletion, and searching reduces significantly as it works on the principle of binary search rather than linear search, as in the case of normal binary trees (will discuss it further).

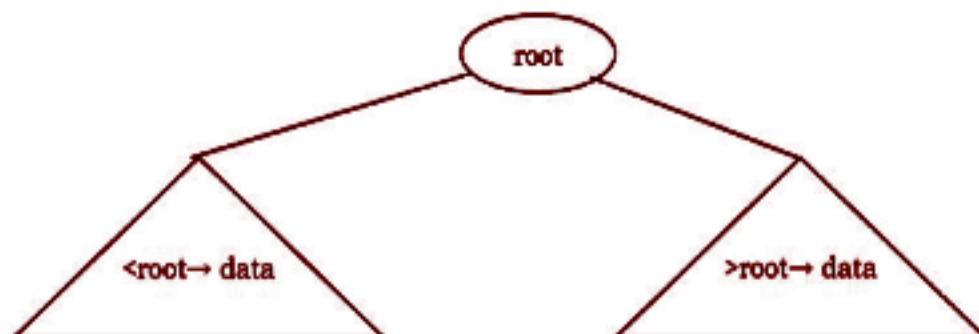
Binary Search Tree Property

In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called

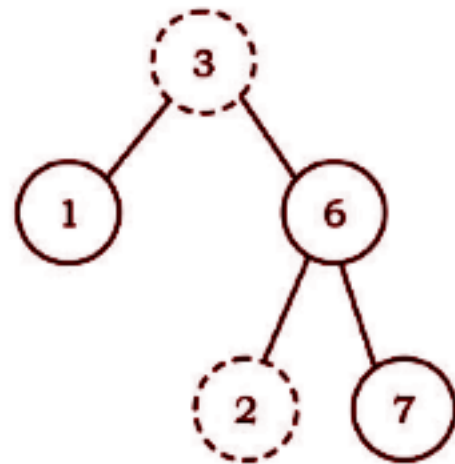
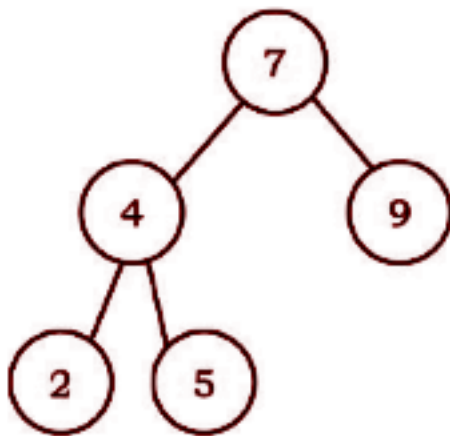
Binary Search Tree property.

- The left subtree of a node ONLY contains nodes with keys less than the node's key.
- The right subtree of a node ONLY contains nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Note: The **BST** property should be satisfied at every node in the tree.



Example: The left tree is a binary search tree and the right tree is not a binary search tree (This because the BST property is not satisfied at node 6. Its child with key 2, is less than its parent with key 3, which is a violation, as all the nodes on the right subtree of root node 3, must have keys greater than or equal to 3).



Store Data in BST

Example: Insert {45, 68, 35, 42, 15, 64, 78} in a BST in the order they are given.

Solution: Follow the steps below:

1. Since the tree is empty, so the first node will automatically be the root node.



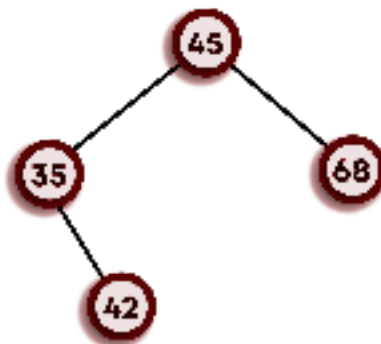
2. Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.



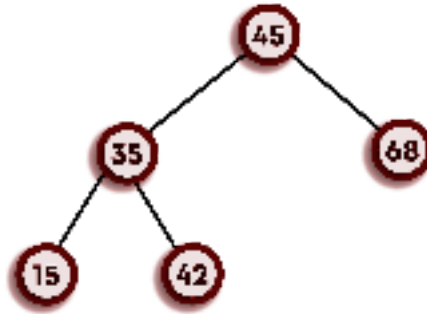
3. To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



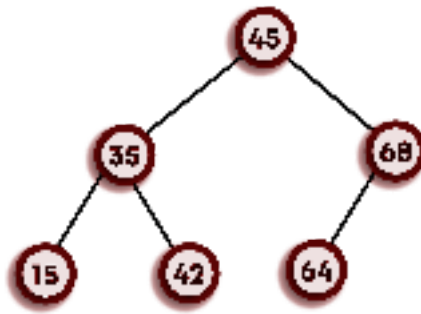
4. Moving on to inserting 42. We can see that $42 < 45$, so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that $42 > 35$ means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



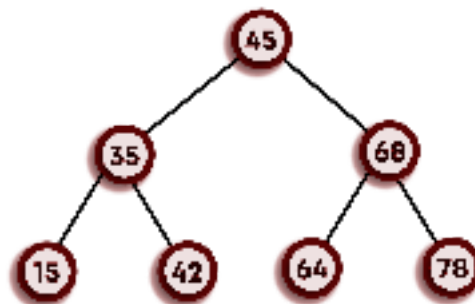
5. Now, to insert 15, we will follow the same approach starting from the root node. Here, $15 < 45$, which means 15 will be a part of the left subtree. As $15 < 35$, we will continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



6. Continuing further, to insert 64, we know that $64 >$ root node's data, but less than 68, hence 64 will be the left child of 68.



7. Finally, we have to insert 78. We can see that $78 > 45$ and $78 > 68$, so 78 will be the right child of 68.



In this way, the data is stored in a BST.

Note:

- If we follow the **inorder traversal** of the final BST, we will get the sorted array.
- As seen above, to insert an element in the BST, we will be traversing till either the left subtree's leaf node or right subtree's leaf node, in the worst-case scenario.
- Hence, the **time complexity of insertion** for each node is **$O(\log(H))$** (where **H** is the height of the tree).
- For inserting **N** nodes, complexity will be **$O(N \cdot \log(H))$** .

Problem Statement: Search in BST

Given a BST and a target value(x), we have to return the binary tree node with data x if it is present in the BST; otherwise, return NULL.

Approach: As the given tree is BST, we can use the **binary search algorithm**. Using recursion will make it easier.

Base Case:

- If the tree is empty, it means that the root node is NULL, then we will simply return NULL as the node is not present.
- Suppose if root's data is equal to **x**, we don't need to traverse forward in this tree as the target value has been found out, so we will simply return the **root** from here.

Small Calculation:

- In the case of BST, we'll only check for the condition of binary search, i.e., if x is greater than the root's data, then we will make a recursive call over the right subtree; otherwise, the recursive call will be made on the left subtree.

- This way, we are entirely discarding half the tree to be searched as done in case of a binary search. Therefore, the time complexity of searching is $O(\log(H))$ (where H is the height of BST).

Recursive call: After figuring out which way to move, we can make recursive calls on either left or right subtree. This way, we will be able to search the given element in a BST.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Problem Statement: Print elements in a range

Given a BST and a range (L, R), we need to figure out all the elements of BST that are present in the given range inclusive of L and R.

Approach: We will be using recursion and binary searching for the same.

Base case: If the root is NULL, it means we don't have any tree to check upon, and we can simply return.

Small Calculation: There are three conditions to be checked upon:

- If the root's data lies in the given range, then we can print it.
- We will compare the root's data with the given range's maximum. If root's data is smaller than R, then we will have to traverse only the right subtree.
- Now, we will compare the root's data with the given range's minimum. If the root's data is greater than L, then we will traverse only the left subtree.

Recursive call: Recursive call will be made as per the small calculation part onto the left and right subtrees. In this way, we will be able to figure out all the elements in the range.

Note: Try to code this yourself, and refer to the solution tab in case of any doubts.

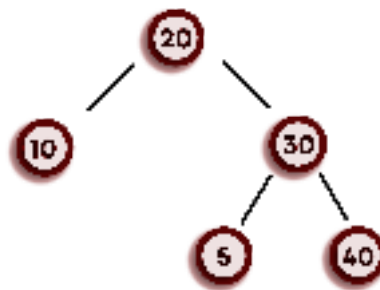
Problem Statement: Check BST

Given a binary tree, we have to check if it is a BST or not.

Approach: We will simply traverse the binary tree and check if the nodes satisfy the BST Property. Thus we will check the following cases:

- If the node's value is greater than the value of the node on its left.
- If the node's value is smaller than the value of the node on its right.

Important Case: Don't just compare the direct left and right children of the node; instead, we need to compare every node in the left and right subtree with the node's value. Consider the following case:



- Here, it can be seen that for root, the left subtree is a BST, and the right subtree is also a BST (individually).
- But the complete tree is not a BST. This is because a node with value 5 lies on the right side of the root node with value 20, whereas it should be on the left side of the root node.
- Hence, even though the individual subtrees are BSTs, it is also possible that the complete binary tree is not a BST. Hence, this third condition must also be checked.

To check over this condition, we will keep track of the minimum and maximum values of the right and left subtrees correspondingly, and at last, we will simply compare them with root.

- The left subtree's maximum value should be less than the root's data.
- The right subtree's minimum value should be greater than the root's data.

Now, let's look at the Python code for this approach using this approach.

```
public int maximum(BinaryTreeNode<Integer> root) {
    if (root == NULL) {          // If root is NULL, then we simply return
        return Integer.MIN_VALUE;    //  $-\infty$  (negative infinity)
    }
    // Otherwise returning maximum of left/right subtree and root's data
    int left = maximum(root.left);
    int right = maximum(root.right);
    return Math.max(root.data, Math.max(left, right));
}
```

```
public int minimum(BinaryTreeNode<Integer> root) {
    if (root == NULL) {          // If root is NULL, then we simply return
        return Integer.MAX_VALUE;    //  $+\infty$  (positive infinity)
    }
    // Otherwise returning minimum of left/right subtree and root's data
    int left = maximum(root.left);
    int right = maximum(root.right);
    return Math.min(root.data, Math.min(left, right));
}
```

```
public bool isBST(BinaryTreeNode<Integer> root) {
    if (root == NULL) {          // Base case
        return true;
    }

    int leftMax = maximum(root.left);    // Figuring out left's maximum
    int rightMin = minimum(root.right); // Figuring out right's minimum
    bool output = (root.data > leftMax) && (root.data <= rightMin) &&
                  isBST(root.left) && isBST(root.right);
    //Checked the conditions discussed above
    return output;
}
```

Time Complexity: In the `isBST()` function, we are traversing each node, and for each node, we are then calculating the minimum and maximum value by again

traversing that complete subtree's height. Hence, if there are **N** nodes in total and the height of the tree is **H**, then the time complexity will be **O(N*H)**.

Improved Solution for Check BST

- To improve our solution, observe that for each node, the minimum and maximum values are being calculated separately.
- We now wish to calculate these values, while checking the **isBST** condition itself, to get rid of another cycle of iterations.
- We will follow a similar approach as that of the diameter calculation of binary trees.
- At each stage, we will return the maximum value, minimum value, and the BST status (True/False) for each node of the tree, in the form of a tuple.

Let's look at its implementation now:

```
class IsBSTReturn {           // Class to store data for each node of tree
    bool isBST;
    int minimum;
    int maximum;
}

-----

public IsBSTReturn isBST2(BinaryTreeNode<Integer> root) {
    if (root == NULL) {       // Base Case
        IsBSTReturn output = new IsBSTReturn(); //Object created
        output.isBST = true;   // Empty tree is a BST
        output.minimum = INT_MAX;
        output.maximum = INT_MIN;
        return output;
    }
    IsBSTReturn leftOutput = isBST2(root.left); // Left subtree Recursion
    IsBSTReturn rightOutput = isBST2(root.right); //Right subtree
                                                //Recursive call

    // Small Calculation
    // Minimum and maximum values figured out side-by-side preventing
    //extra traversals
    int minimum = Math.min(root.data, Math.min(leftOutput.minimum,
                                                rightOutput.minimum));
    int maximum = max(root.data, max(leftOutput.maximum,
                                      rightOutput.maximum));
}
```

```

// Checking out for the subtree if it's a BST or not
bool isBSTFinal = (root.data > leftOutput.maximum) && (root.data <=
                    rightOutput.minimum) && leftOutput.isBST &&
                    rightOutput.isBST;

// Assigning values to the output class object
IsBSTReturn output;
output.minimum = minimum;
output.maximum = maximum;
output.isBST = isBSTFinal;
return output;
}

```

Time Complexity: Here, we are going to each node and doing a constant amount of work. Hence, the time complexity for **N** nodes will be of **O(N)**.

Another Improved Solution for Check BST

The time complexity for this problem can't be improved further, but there is a better approach to this problem, which makes our code look more robust. Let's discuss that approach now.

Approach: We will be checking on the left subtree, right subtree, and combined tree without returning a tuple of maximum and minimum values. We will be using the concept of default arguments over here. Check the code below:

```

// This time function is using default arguments for storing minimum and
// maximum value for each node
private bool isBST3Helper(BinaryTreeNode<Integer> root, int min, int max) {
    if (root == NULL) {          // Base case: Empty tree
        return true;
    }
    // checking out the special condition first and returning false if not
    // satisfied
    if (root->data < min || root->data > max) {
        return false;
    }
    // Checking out left and right subtrees
    bool isLeftOk = isBST3(root.left, min, root.data - 1);
    bool isRightOk = isBST3(root.right, root.data, max);
    // Returning true if both are BST and false otherwise.
    return isLeftOk && isRightOk;
}

```

```
public bool isBST3(BinaryTreeNode<Integer> root) {  
    return isBST3Helper(root, Integer.MIN_VALUE, Integer.MAX_VALUE);  
}
```

Time Complexity: Here also, we are just traversing each node and doing constant work on each of them; hence time complexity remains the same, i.e. **O(n)**.

Problem Statement: Construct BST from sorted array

Given a sorted array, we have to construct a BST out of it.

Approach:

- Suppose we take the first element as the root node, then the tree will be skewed as the array is sorted.
- To get a balanced tree (so that searching and other operations can be performed in **O(log(n))** time), we will be using the binary search technique.
- Figure out the middle element and mark it as the root.
- This is done so that the tree can be divided into almost 2 equal parts, as half the elements which will be greater than the root, will form the right subtree (These elements are present to its right).
- The elements in the other half, which are less than the root, will form the left subtree (These elements are present to its left).
- Just put root's left child to be the recursive call made on the left portion of the array and root's right child to be the recursive call made on the right portion of the array.
- Try it yourself and refer to the solution tab for code.

BST to Sorted LL

Problem statement: Given a BST, we have to construct a sorted linked list out of it.

Approach: As discussed earlier, the inorder traversal of the BST, provides elements in a sorted fashion, so while creating the linked list, we will be traversing the tree in inorder style.

- **Base case:** If the root is NULL, it means the head of the linked list is NULL; hence we return NULL.
- **Small calculation:** Left subtree will provide us the head of LL, and the right subtree will provide the tail of LL; hence root node will be placed after the LL obtained from the left subtree and right LL will be connected after the root.

Left subtree's LL	Root	Right subtree's LL
-------------------	------	--------------------

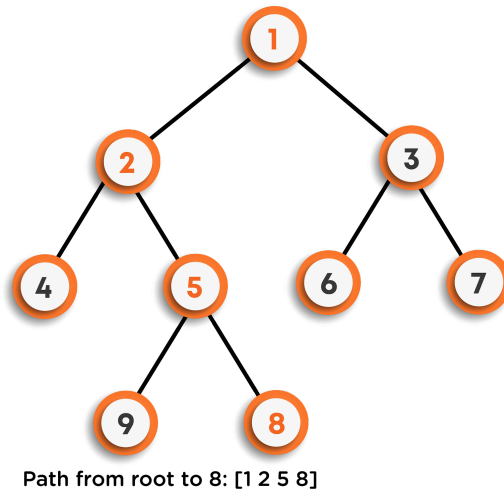
- **Recursive call:** Simply call the left subtree, connect the root node to the end of it, and then connect the right subtree's recursive call after root.

Try it yourselves, and for code, refer to the solution tab of the corresponding question.

Root to node path in a Binary Tree

Problem statement: Given a Binary tree, we have to return the path of the root node to the given node.

For Example: Refer to the image below...



Approach:

1. Start from the root node and compare it with the target value. If matched, then simply return, otherwise, recursively call over left and right subtrees.
2. Continue this process until the target node is found and return if found.
3. While returning, you need to store all the nodes that you have traversed on the path from the root to the target node in a vector.
4. Now, in the end, you will be having your solution vector.

Now try to create logic and code the same problem with BST instead of a binary tree.

BST Class

Here, we will be creating our own BST class to perform operations like insertion, deletion, etc...

Kindly, follow the code below:

```

class BST {
    BinaryTreeNode<Integer> root;    // root node

    BST() {                          // Constructor to initialize root to NULL
        root = null;
    }

    public bool hasData(int data, BinaryTreeNode<Integer> node) {
        if (node == NULL) {          // the presence of a
            return false;             // node in BST
        }

        if (node->data == data) {
            return true;
        } else if (data < node->data) {
            return hasData(data, node->left);
        } else {
            return hasData(data, node->right);
        }
    }

    public bool hasData(int data) {
        return hasData(data, root); // from here the value is returned
    }
}

```

You can observe that hasData() function has been declared under the private section to prevent it from being manipulated, presenting the concept of data abstraction.

Try insertion and deletion on your own, and in case of any difficulty, refer to the hints and solution code below...

Insertion in BST:

We are given the root of the tree and the data to be inserted. Follow the same approach to insert the data as discussed above using Binary search algorithm. Check the code below for insertion:

```
public BinaryTreeNode<Integer> insert(int data, BinaryTreeNode<Integer> node){
    // Using Binary Search algorithm
    if (node == null) {
        BinaryTreeNode<Integer> newNode = new BinaryTreeNode<>(data);
        return newNode;
    }

    if (data < node.data) {
        node.left = insert(data, node.left);
    } else {
        node.right = insert(data, node.right);
    }
    return node;
}

public void insert(int data) {          // Insertion function
    this.root = insert(data, this.root);
}
```

Deletion in BST:

Recursively, find the node to be deleted.

- **Case 1:** If the node to be deleted is the leaf node, then simply delete that node with no further changes and return NULL.
- **Case 2:** If the node to be deleted has only one child, then delete that node and return the child node.
- **Case 3:** If the node to be deleted has both the child nodes, then we have to delete the node such that the properties of BST remain unchanged. For this, we will replace the node's data with either the left child's largest node or right child's smallest node and then simply delete the replaced node.

Now, let's look at the code below:

```
BinaryTreeNode<Integer> deleteData(int data, BinaryTreeNode<Integer> node){
    if (node == NULL) {          // Base case
        return NULL;
    }
    // Finding that node by traversing the tree
```

```

    if (data > node.data) {
        node.right = deleteData(data, node.right);
        return node;
    } else if (data < node.data) {
        node.left = deleteData(data, node.left);
        return node;
    } else { //found the node
        if (node.left == NULL && node.right == NULL) { //Leaf node
            delete node;
            return NULL;
        } else if (node.left == NULL) { //node having only left child
            BinaryTreeNode<Integer> temp = node.right;
            node.right = NULL;
            delete node;
            return temp;
        } else if (node.right == NULL) { //node having only right child
            BinaryTreeNode<Integer> temp = node.left;
            node.left = NULL;
            delete node;
            return temp;
        } else { //node having both the childs
            BinaryTreeNode<Integer> minNode = node.right;
            while (minNode.left != NULL) { // replacing node with
                minNode = minNode.left; // right subtree's min
            }
            int rightMin = minNode.data;
            node.data = rightMin;
            // now simply deleting that replaced node using recursion
            node.right = deleteData(rightMin, node.right);
            return node;
        }
    }
}

public void deleteData(int data) { // Function to delete
    root = deleteData(data, root);
}

```