Binary Search Tree (BST)

Introduction

- These are the specific types of binary trees.
- These are inspired by the binary search algorithm that elements on the left side of a particular element are smaller and greater element in case of the right side.
- 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).

How is data stored?

In BSTs, we always insert the node with smaller data towards the left side of the compared node and the larger data node as its right child. To be more concise, consider the root node of BST to be N, then:

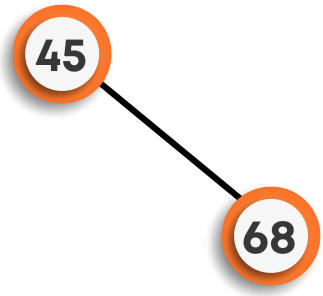
- Everything lesser than N will be placed in the left subtree.
- Everything greater than N will be placed in the right subtree.

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

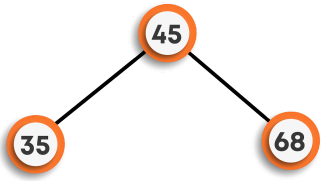
Solution: Follow the steps below:

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

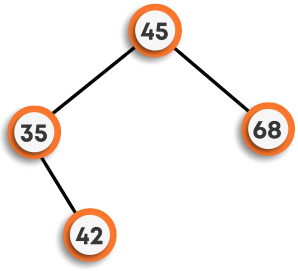
2. Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.



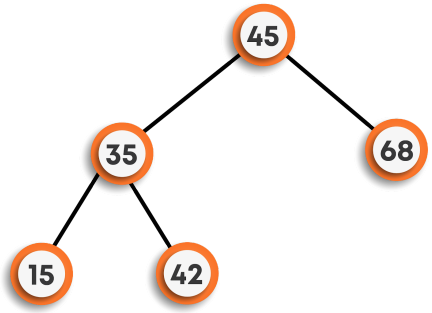
3. To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



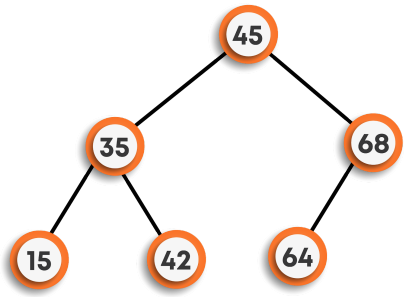
4. Moving on to inserting 42, we can see that 42 is less than 45 so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that 42 > 35 means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



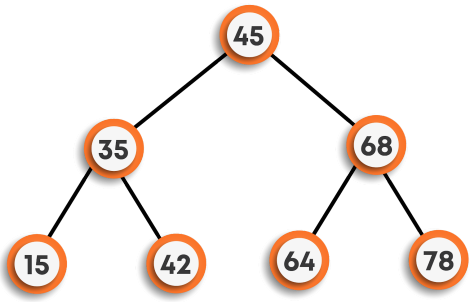
5. Now, on inserting 15, we will follow the same approach starting from the root node. Here, 15 < 45, means left subtree. Again, 15 < 35, means continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



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



7. Finally, inserting 78, we can see that 78 > 45 and 78 > 68, so will be the right child of 68.



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

If we follow the **inorder traversal** of the final BST, we will get the sorted array, hence to search we can now directly apply binary search algorithm technique over it to search for any particular data.

As seen above, to insert an element, we will be at most traversing either the left subtree's leaf node or right subtree's leaf node ignoring the other half straight away. Hence, the **time complexity of insertion** for each node is O(logh) (where h is the height of the tree).

For inserting n nodes, complexity will be O(nlogh).

Search in BST

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

Approach: As the given tree is BST, we can use the binary search algorithm here as we know that the left side of the node contains all elements smaller than it and the right side contains all elements greater than the value of node. Using recursion will make it easier as we just have to work on a small task, and rest recursion will handle itself.

- **Base Case:** If the tree is empty, it means 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 root from here.
- **Small Calculation:** In the case of binary trees, we were required to traverse both the left and right subtree of the root in order to find out the target value. But in case of BST, we'll only check for the condition of binary search, i.e., if x is greater than 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 the half tree to be searched as done in case of a binary search. Therefore, the time complexity of searching is O(logh) (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 figure out the search in a BST.

Print elements in a range:

Problem statement: 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 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 the right subtree.
    - Now, we will compare the root's data with the given range's minimum. If root's data is greater than L, then we will traverse on the left subtree.
- **Recursive call:** Recursive call will be made as per the small calculation part onto the left and right subtrees.

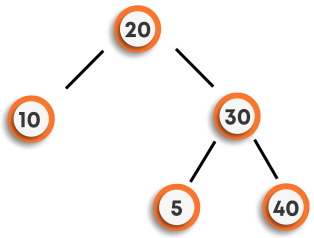
In this way, we will be able to figure out all the elements in the range. Try to code this yourself, and for the solution, refer to the solution tab of the same in the module.

Check BST

Problem statement: 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 for the following cases:

- If the node's value is greater than the value of the node on it's left.
- If the node's value is smaller than the value of node on it's 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, left subtree is a BST, and right subtree is also a BST (individually), but the complete tree is not as 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, individual subtrees are BSTs, but it is possible that the complete binary tree is not a BST, hence, the third condition must also be checked.

To check over this condition, we will keep track of minimum and maximum values of 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 code for this approach starting from the brute-force approach.

```
int maximum(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // If root is NULL, then we simply return
                return INT_MIN;                // -∞ (negative infinity)
        }
        // Otherwise returning maximum of left/right subtree and root's data
        return max(root->data, max(maximum(root->left), maximum(root->right)));
}


int minimum(BinaryTreeNode<int>* root) {
        if (root == NULL) {                    // If root is NULL, then we simply return
                return INT_MAX;                // +∞ (positive infinity)
        }
        // Otherwise returning minimum of left/right subtree and root's data
        return min(root->data, min(minimum(root->left), minimum(root->right)));
}

bool isBST(BinaryTreeNode<int>* 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(), 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 height of the tree is *h*, the time complexity will be O(n*h).

To improve this further, we can see that for each node, minimum and maximum values are calculated separately. We want to calculate these along with checking the isBST condition.

We will follow a similar approach as that of the diameter calculation of binary trees. We will create a class that will be storing maximum value, minimum value, and BST status (True/False) for each node of the tree.

Let's look at its implementation now...

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

IsBSTReturn isBST2(BinaryTreeNode<int>* root) {
        if (root == NULL) {                            // Base Case
                IsBSTReturn output;                    // Object created for class and then values initialized
                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 Recursive call
        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 = min(root->data, 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).

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 using class. We will be using the concept of default argument over here. Check the code below:

```
// This time function is using default arguments for storing minimum and
// maximum value for each node
bool isBST3(BinaryTreeNode<int>* root, int min = INT_MIN, int max = 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;
}
```

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

Construct BST from sorted array

Problem statement: 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. In order to get a balanced tree (so that searching and other operations can be performed in O(logn) time), we will be using the binary search technique. Figure out the middle element and mark it as root. Now the elements on it's left form the left subtree, and elements on the right will form the right subtree. Just put root's left to be the recursive call made on the left portion of the array and root's right 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

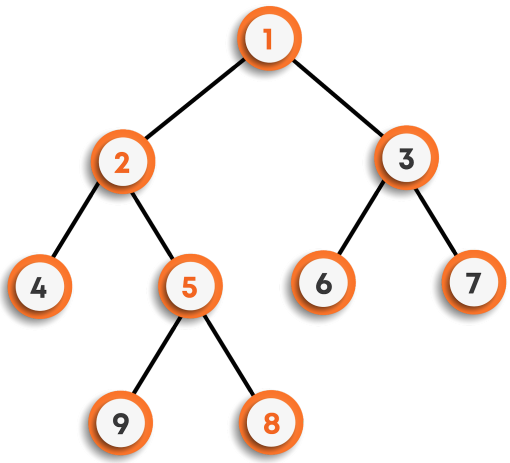
- **Recursive call:** Simply call the left subtree, connect the root node to the end of it, and then connect the right subtree's recursive call after root.

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...



Path from root to 8: [1 2 5 8]

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.

Code:

```
vector<int>* getRootToNodePath(BinaryTreeNode<int>* root, int data) {
        if (root == NULL) {                    // Base Case:
                return NULL;
        }
```

```
        if (root->data == data) {                // Small calculation part: when node found
                vector<int>* output = new vector<int>();
                output->push_back(root->data); // inserted the node in solution vector
                return output;
        }
        // getting output vector out of left subtree
        vector<int>* leftOutput = getRootToNodePath(root->left, data);
        if (leftOutput != NULL) {
                leftOutput->push_back(root->data);
                return leftOutput;
        }
       // getting output vector out of right subtree
        vector<int>* rightOutput = getRootToNodePath(root->right, data);
        if (rightOutput != NULL) {
                        rightOutput->push_back(root->data);
                        return rightOutput;
        } else {
                return NULL;
        }
}
```

Now try to 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:

```
#include <iostream>
using namespace std;
class BST {
        BinaryTreeNode<int>* root;              // root node

        public:

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

```
        ~BST() {                                    // Destructor to delete the BST
                delete root;
        }

        private:
        bool hasData(int data, BinaryTreeNode<int>* node) {     // function to detect
                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:

```
private:
        BinaryTreeNode<int>* insert(int data, BinaryTreeNode<int>* node) {
                // Using Binary Search algorithm
                if (node == NULL) {
```

```
                BinaryTreeNode<int>* newNode = new BinaryTreeNode<int>(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:

```
private:
        BinaryTreeNode<int>* deleteData(int data, BinaryTreeNode<int>* 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<int>* temp = node->right;
                node->right = NULL;
                delete node;
                return temp;
            } else if (node->right == NULL) {    // node having only right child
                BinaryTreeNode<int>* temp = node->left;
                node->left = NULL;
                delete node;
                return temp;
            } else {            // node having both the childs
                BinaryTreeNode<int>* 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);
    }
```

Types of balanced BSTs

For a balanced BST:

|Height_of_left_subtree - Height_of_right_subtree| <= 1

This equation must be valid for each and every node present in the BST.

By mathematical calculations, it was found that the height of a Balanced BST is log(n), where n is the number of nodes in the tree.

This can be summarised as the time complexity of operations like searching, insertion, and deletion can be performed in O(logn).

There are many Binary search types that maintain balance. We will not be discussing them over here. These are as follows:

- AVL Trees (also known as self-balancing BST, uses rotation to balance)
- Red-Black Trees
- 2 - 4 Tree

Practice Problems:

- **https://www.hackerearth.com/practice/data-structures/trees/binary-search-tree/practice-problems/algorithm/dummy3-4/**
- **https://www.codechef.com/problems/KJCP01**
- **https://www.codechef.com/problems/BEARSEG**