Binary Search Trees- 1

Introduction

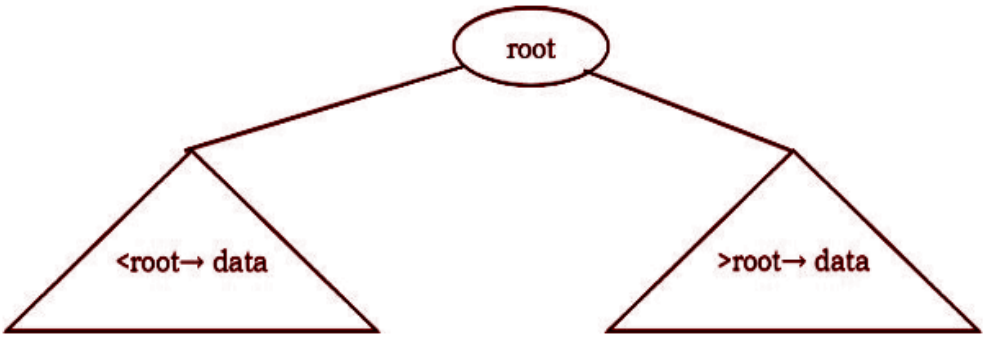
- These are the specific types of binary trees.
- These are inspired by the binary search algorithm.
- Time complexity on insertion, deletion, and searching reduces significantly as it works on the principle of binary search rather than linear search, as in the case of normal binary trees (will discuss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

Binary Search Tree Property

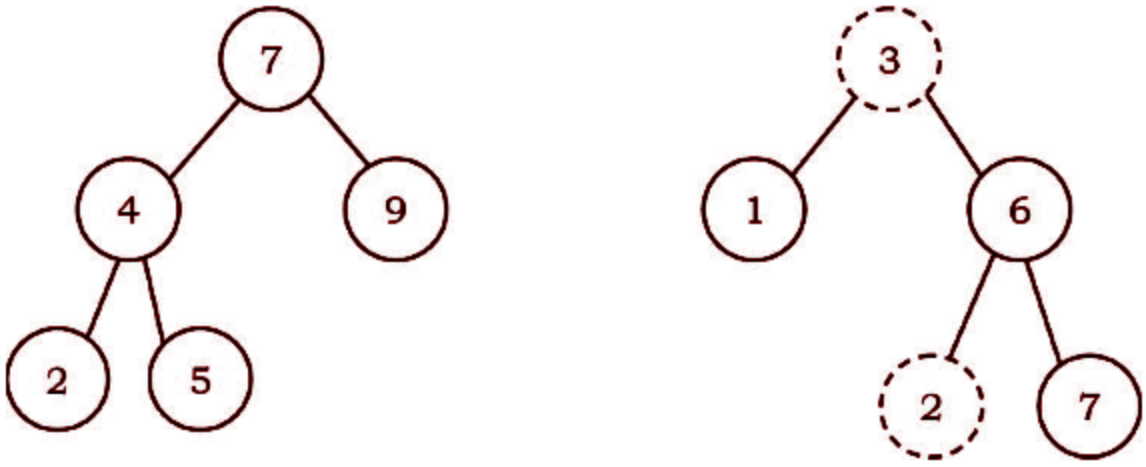
In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called **Binary Search Tree property**.

- 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)*.



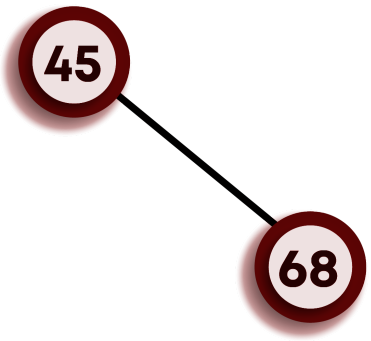
Store Data in BST

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

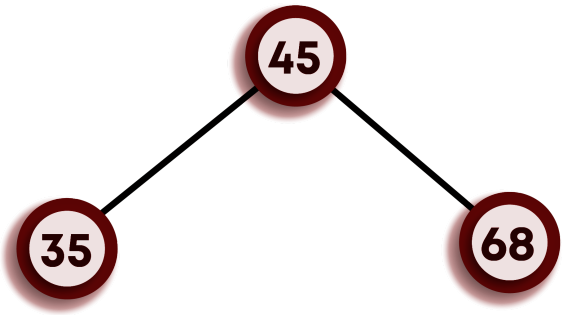
Solution: Follow the steps below:

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

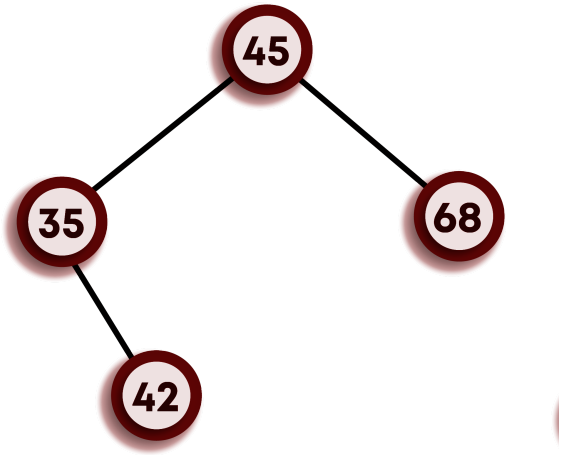
2. Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.



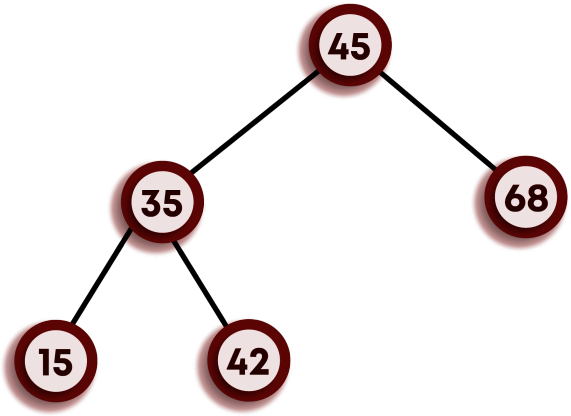
3. To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



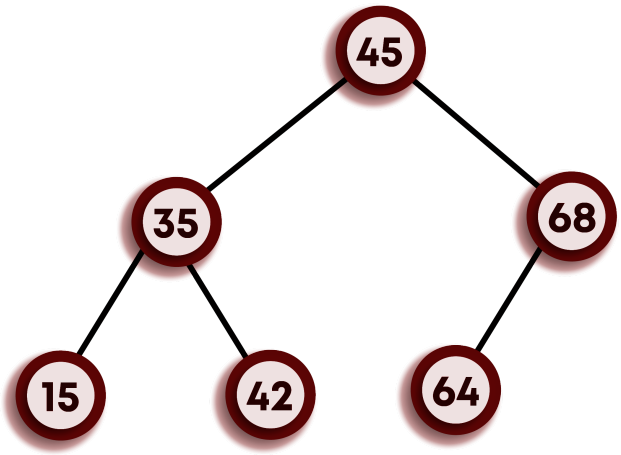
4. Moving on to inserting 42. We can see that 42 < 45, so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that 42 > 35 means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



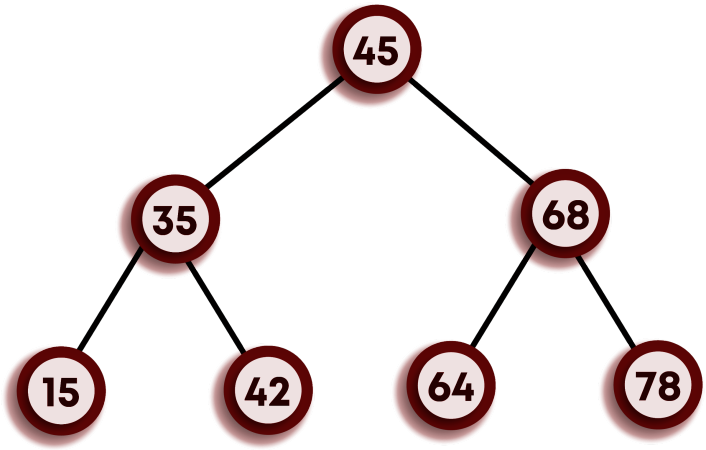
5. Now, to insert 15, we will follow the same approach starting from the root node. Here, 15 < 45, which means 15 will be a part of the left subtree. As 15 < 35, we will continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



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*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.

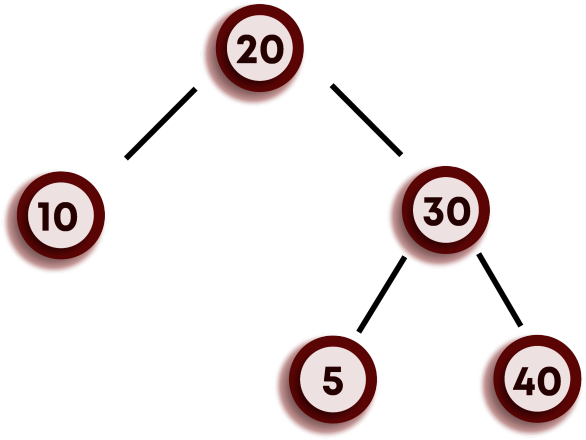
Problem Statement: Check BST

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 it's 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.

```
def minTree(root):#Returns minimum value in a subtree
      if (root == None):#If root is None, it implies empty tree
            return 100000 #Return a large number in this case
      leftMin = minTree(root.left) #Find min in left subtree
      rightMin = minTree(root.right) #Find min in right subtree
      return min(leftMin, rightMin, root.data) #Return overall min

def maxTree(root):#Returns minimum value in a subtree
      if (root == None): #If root is None, it implies empty tree
            return -100000 #Return a very small number in this case
      leftMax = maxTree(root.left) #Find max in left subtree
      rightMax = maxTree(root.right) #Find max in right subtree
      return max(leftMax, rightMax, root.data) #Return overall max

def isBST(root)
      if (root== None):#If root is None, it implies empty tree
            return True #Empty tree is a BST

      leftMax = maxTree(root.left) #Max in left subtree
      rightMin = minTree(root.right) #Min in right subtree
      if root.data > rightMin or root.data <= leftMax:
            return False #Checking BST Property

      isLeftBST = isBST(root.left)#Recursive call on left subtree
      isRightBST = isBST(root.right)#Recursive call on right subtree
      return isLeftBST and isRightBST #Both must be BST
```

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:

```
def isBST2(root):
      if root == None:
            return 100000, -100000, True

      leftMin, leftMax, isLeftBST= isBST2(root.left)
      rightMin, rightMax, isRightBST = isBST2(root.right)

      minimum= min(leftMin rightMin, root.data) #Minimum value
      maximum = max(leftMax, rightMax, root.data) #Maximum value
      isTreeBST=True

      #Checking the BST Property
      if root.data <= leftMax or root.data > rightMin:
            isTreeBST = False
      if not(isLeftBST) or not(isRightBST):
            isTreeBST = False

      return minimum, maximum, isTreeBST
```

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:

```
def isBST3(root, min_range, max_range):
      if root==None:#Empty tree is a BST
            return True

      if root.data < min_range or root.data > max_range:
            return False #Check the BST Property

      isLeftWithinConstraints = isBST3(root.left, min_range, root.data - 1)
      isRightwithinConstraints = isBST3(root.right, root.data, max_range)

      return isLeftWithinConstraints and isRightWithinConstraints
```

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.

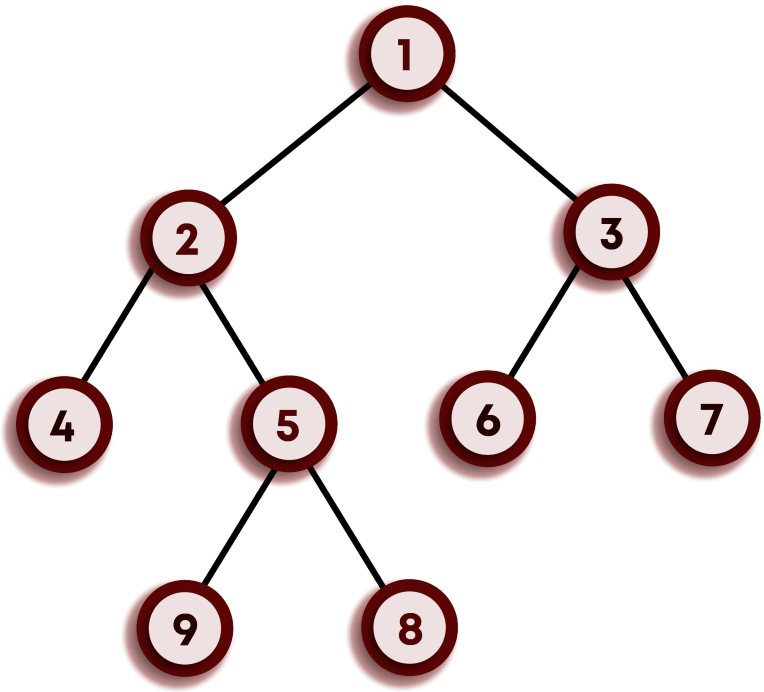
Binary Search Trees- 2

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 list.
4. Now, in the end, you will be having your solution list.

Python Code:

```
def nodeToRootPath(root, s):
      if root == None: #Empty tree
            return None
      if root.data == s: #If data found at root node
            l = list() #Make a list and append -> Store the path
            l.append(root.data)
            return l

      leftOutput = nodeToRootPath(root.left, s)#Recursive call
      if leftOutput != None:
            leftOutput.append(root.data)
            return leftOutput

      rightOutput = nodeToRootPath(root.right, s)#Recursive call
      if rightOutput != None:
            rightOutput.append(root.data)
            return rightOutput
      else:
            return None
```

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

BST class

Here, we will be creating our own BST class to perform operations like insertion, deletion, etc. Follow the given template for constructing your own BST Class. We will be creating a class with the given set of methods:

```
class BST:
      def __init__(self):#Constructor for the class
            self.root = None
            self.numNodes = 0

      def isPresent(self, data): #Check if an element is present
            return False

      def insert(self, data): #Insert a Node in BST
            return

      def deleteData(self, data): #Delete a Node
            return False

      def count(self):
            return self.numNodes
```

Search a Node in BST - `isPresent()`

The solution to this problem would be a recursive function that searches the node in the given BST, similar to what we have seen earlier. Go through the given code for better understanding:

```
def isPresentHelper(self, root, data):
      root == None:  # If the tree is empty, the node is not present
            return False

      root.data  == data: #If data is found at the root node
            return True

      if root.data>data: #call on left
            return isPresentHelper(root.left, data)
```

```
        else #call on right
                return isPresentHelper(root.right, data)

def isPresent(self, data): #Search a Node in the BST
        return isPresentHelper(self.root, data)
```

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 in a BST:

```
def insertHelper(self, root, data):
        if root == None: #Empty tree
                node = BinaryTreeNode(data)
                return node

        if root.data >data: #Left recursion
                root.left = self.insertHelper(root.left, data)
                return root

        else: #Right recursion
                root.right = self.insertHelper(root.right, data)
                return root

def insert(self, data):
        self.numNodes +=1 # Update the number of nodes
        self.root = self.insertHelper(self.root, data) #Call helper
```

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 **None**.
- **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 the right child's smallest node and then simply delete the replaced node.

Now, let's look at the code below:

```
def min(self, root):
      if root == None:
            return 10000
      if root.left == None:
            return root.data
      return self.min(root.left)

def deleteData(self, data):
      deleted, newRoot = self.deleteDataHelper(self.root, data)
      if deleted:
            self.numNodes -= 1
      self.root = newRoot
      return deleted

def helper(self, root, data):
      if root = = None:
            return False, None
       #BST PROPERTY
      if root.data < data:
            deleted, newRightNode = self.helper(root.right, data)
            root.right = newRightNode
            return deleted, root

     if root.data > data:
            deleted, newleftNode = self.helper(root.left, data)
            root.left = newLeftNode
            return deleted, root

     # root is a leaf node
     if root.left == None and root.right == None:
            return True, None
```

```
        #root has one child
        if root.left == None:
             return True, root.right
        if root.right == None:
             return True, root.left

        # root has two children
        replacement = self.min(root.right)
        root.data = replacement
        deleted, newRightNode = self.helper(root.right, replacement)
        root.right = newRightNode
        return True, root
```

Types of Balanced BSTs

- For a balanced BST:

```
|Height_of_left_subtree - Height_of_right_subtree| <= 1
```

- This equation must be valid for 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(log(n))**.
- Many BST types 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