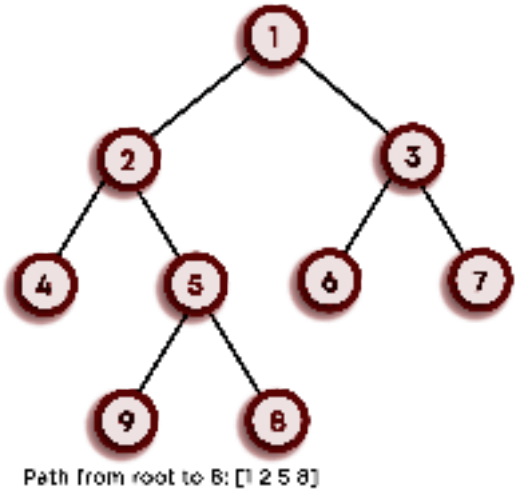
Binary Search Trees- 2

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...



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