Binary Search Tree

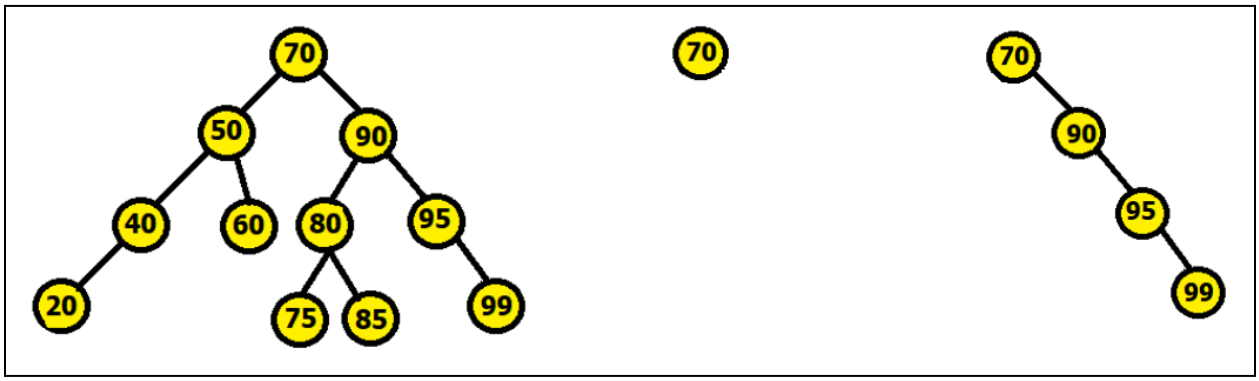
Characteristics of a BST:

Binary Search Tree is a binary tree data with the following fundamental properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtrees must also be binary search trees.
- Each node must have a distinct key, so no duplicate values are allowed.

The goal of using BST data structure is to search any element within O(log(n)) time complexity.

Examples of Binary Search Tree:



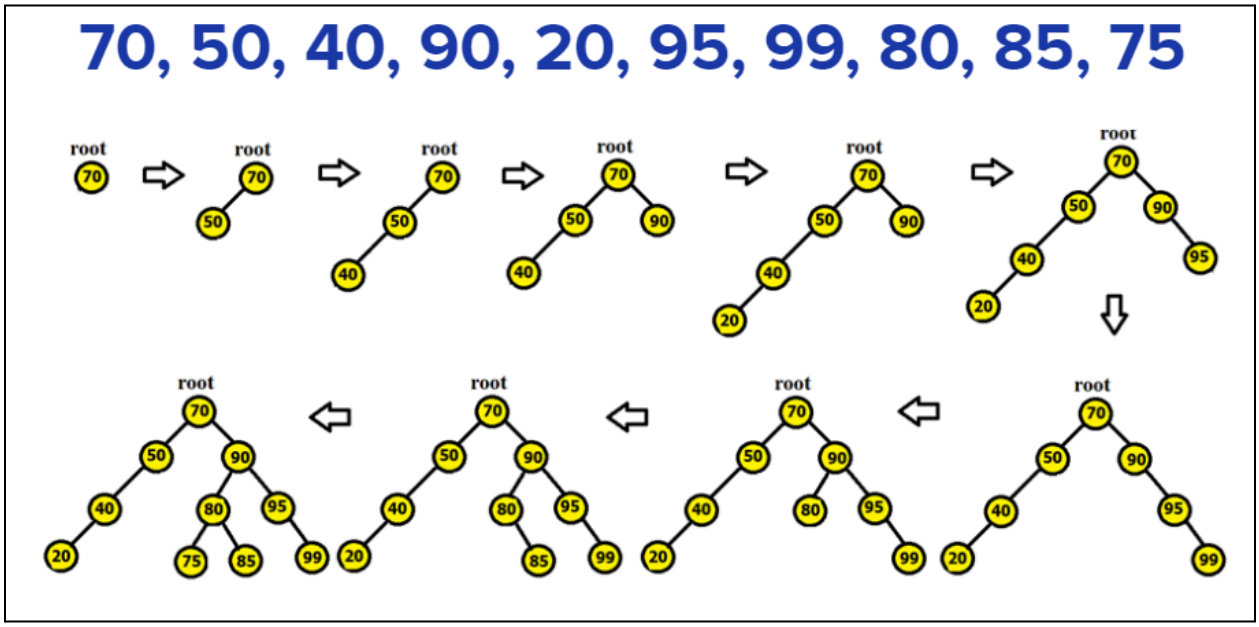
Binary Search Tree Operations:

Any operation done in a BST must not violate its fundamental properties.

- **Creation:**

Create the BST by inserting the following numbers from left to right:
70, 50, 40, 90, 20, 60, 20, 95, 99, 80, 85, 75

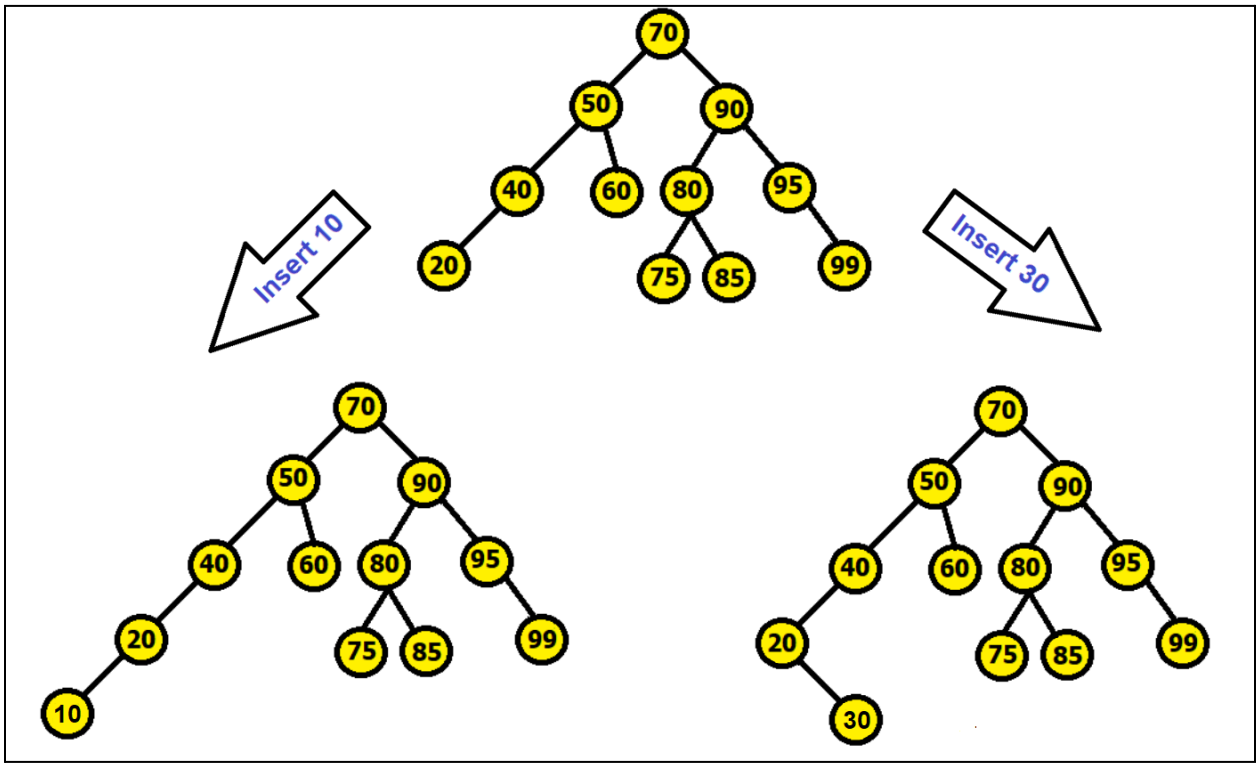
i) Take the value and compare with root.
ii) If the value < root, go to left subtree
iii) Else if the value > root, go to right subtree
iv) Keep going until a vacant space is found and insert the new value



- **Insertion:**

Similar to Creation

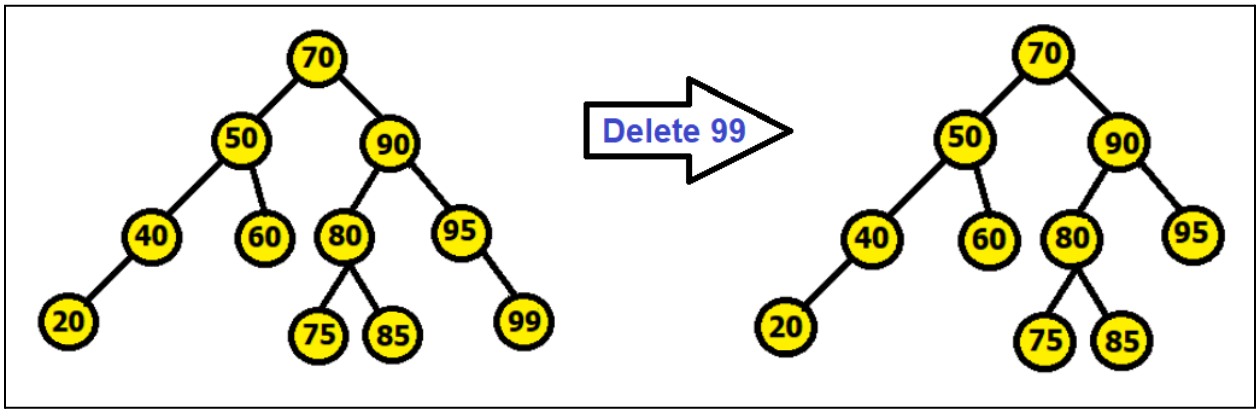
i) Take the node and compare with root.
ii) If node value < root, go to left subtree
iii) Else if node value > root, go to right subtree
iv) Keep going until a vacant space is found and insert the new node



- **Deletion:**

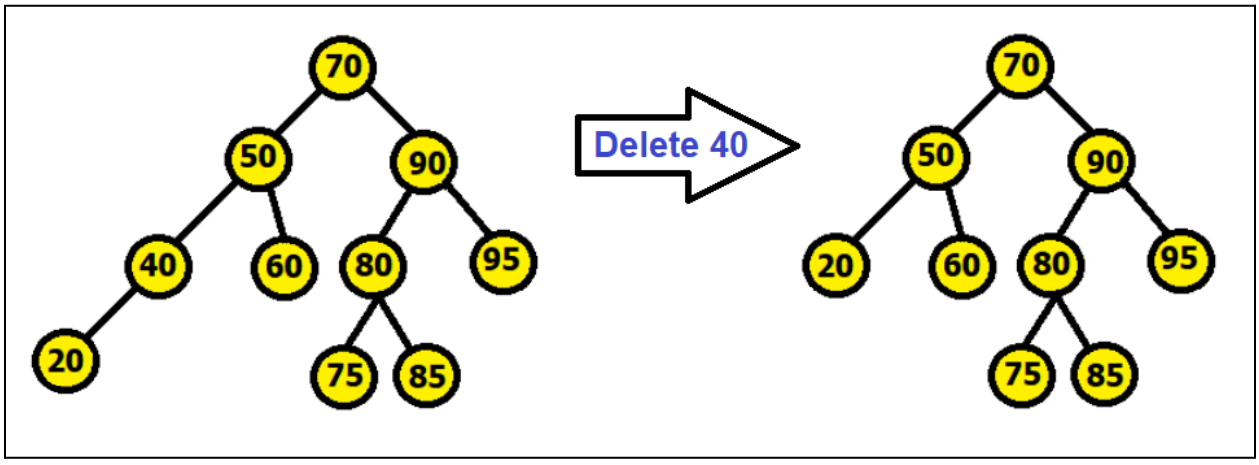
i) Case 1: No subtree or children:

You can simply delete the node without any additional actions.



ii) Case 2: One subtree or one child:

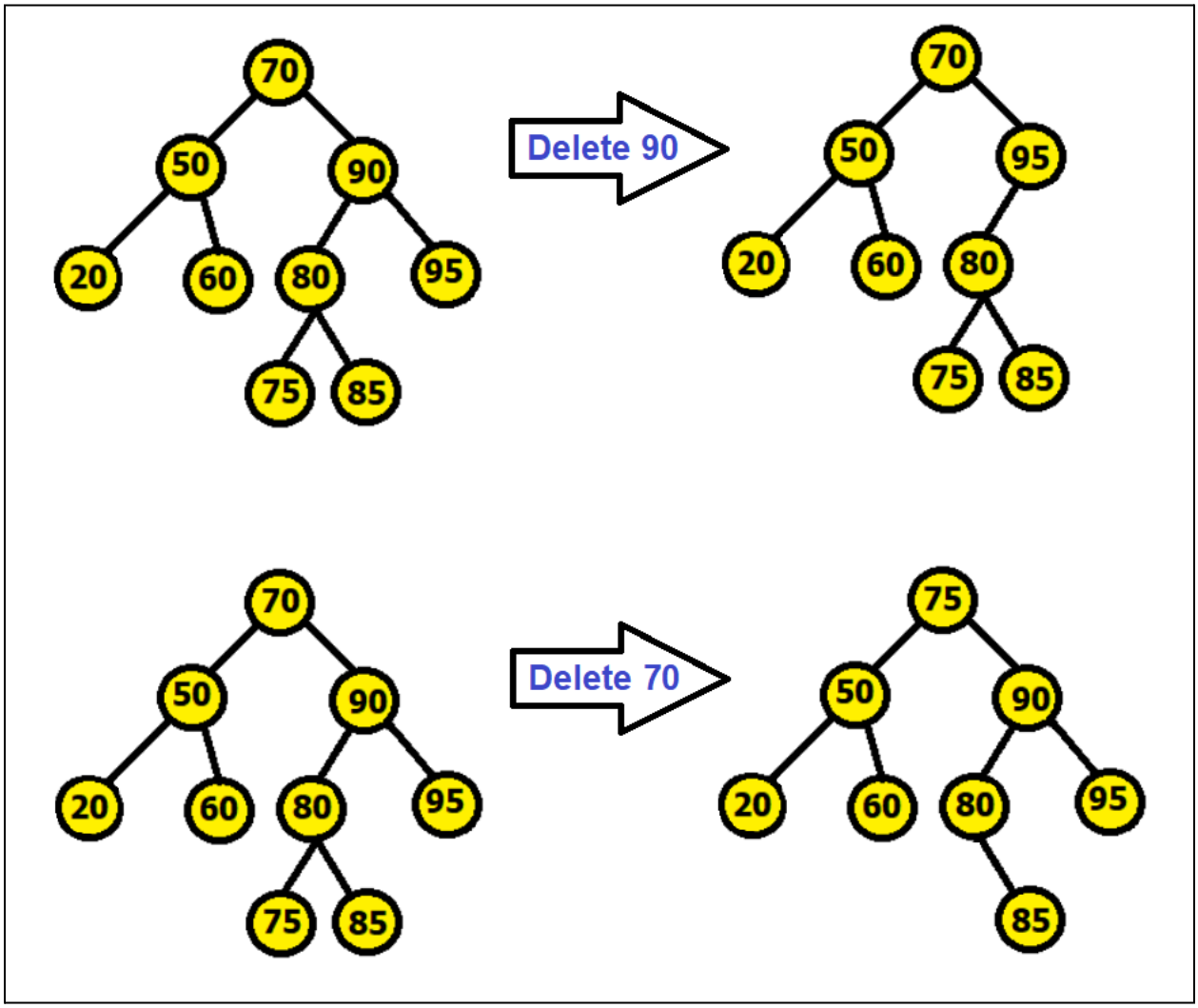
You have to ensure that after the node is deleted, its child is connected to the deleted node's parent.



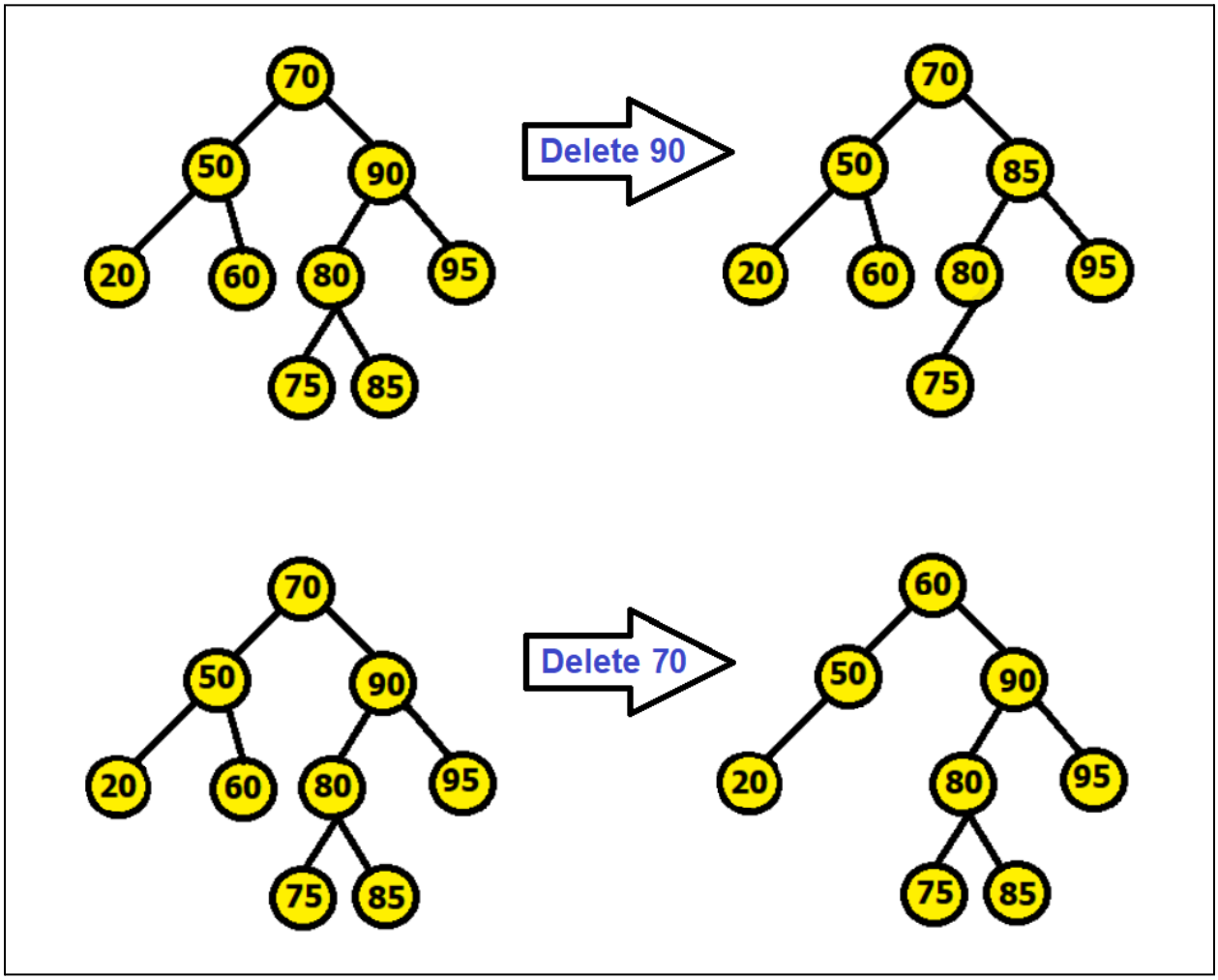
ii) Case 3: Two subtree or two children:

There are two ways a node can be deleted in Case 3.

Inorder Successor: Replace the deleted node with the leftmost node of its right subtree



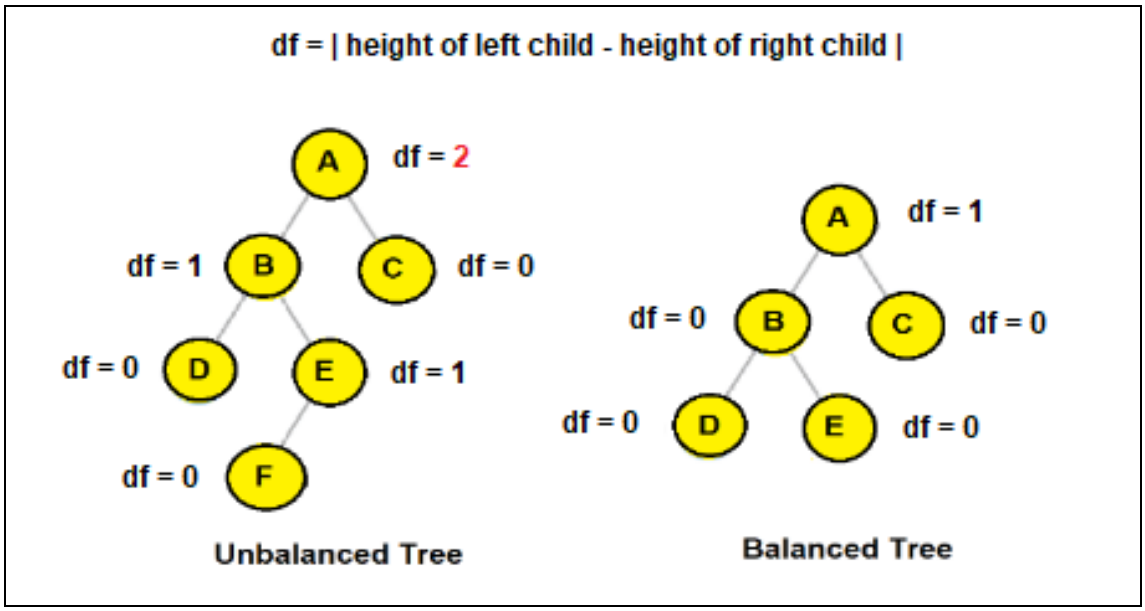
Inorder Predecessor: Replace the deleted node with the rightmost node of its left subtree



Balanced Binary Search Tree:

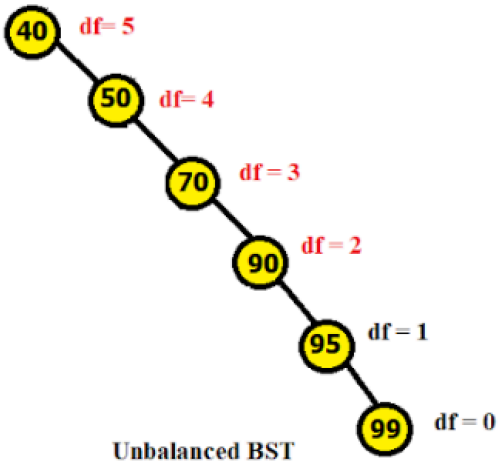
After Insertion and Deletion, a tree might be unbalanced. We have to make it balanced.

If the height difference between the left and right subtree of any node in the BST is more than one, it is an unbalanced BST. Otherwise, it is a balanced one. In a balanced BST, any searching operation can take up to O(log(n)) time complexity. In an unbalanced one, it may take up to O(n) time complexity, making BST usage obsolete. Therefore, we should only work with balanced BST and after conducting any operation on a BST, we must first check if it became unbalanced or not. If it does become unbalanced, we have to balance it.



Conversion of unbalanced BST into balanced BST:

- Traverse given BST in inorder and store result in an array. This will give an ascending sorted order of all the values.
- Take the middlemost value in the (size//2) position in the array and make it the root.
- The left subtree of the root will be all the values from 0 to (size//2)-1 positions of the array. The right subtree of the root will be all the values from (size//2)+1 to (size-1) positions of the array.
- Again choose the middlemost values from the left subtree and right subtree and connect these to the root. Keep on repeating the process until all the elements of the array have been taken.



Unbalanced BST

Inorder Traversal

40	50	70	90	95	99

array for original tree

size = 6
root position = (size//2) = 3
Pick 90 as the root

root

90

40	50	70

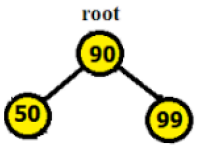
array for left subtree of node 90

size = 3
node position = (size//2) = 1
Pick 50 as next left node

95	99

array for right subtree of node 90

size = 2
node position = (size//2) = 1
Pick 99 as next right node



40

array for left subtree of node 50

size = 1
node position = (size//2) = 0
Pick 40 as next left node

70

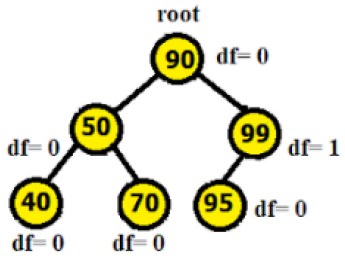
array for right subtree of node 50

size = 1
node position = (size//2) = 0
Pick 70 as next right node

95

array for left subtree of node 99

size = 1
node position = (size//2) = 0
Pick 95 as next left node



Balanced BST

Binary Search Tree Coding:

- **BST Creation or Insertion:**

```
class Node:
    def __init__(self, elem):
        self.elem = elem
        self.left = None
        self.right = None
```

```
def addNode(root, i):
    if i < root.elem and root.left == None:
        n = Node(i)
        root.left = n
    elif i > root.elem and root.right == None:
        n = Node(i)
        root.right = n
    if i < root.elem and root.left != None:
        addNode(root.left, i)
    elif i > root.elem and root.right != None:
        addNode(root.right, i)

arr = [70, 50, 40, 90, 20, 60, 20, 95, 99, 80, 85, 75]
root = Node(arr[0])
for i in arr[1:]:
    addNode(root, i)
```

- **Searching Items:**

```
def search(root, item):
        if root.elem == item:
                return root
        elif item < root.elem and root.left == None:
                return None
        elif item > root.elem and root.right == None:
                return None
        elif item < root.elem and root.left != None:
                return search(root.left, item)
        elif item > root.elem and root.right != None:
                return search(root.right, item)
```

- **Deletion (Inorder Successor):**

```
def leftMostNode(node):  #Finding leftmost node
        current = node
        while current.left is not None:
                current = current.left
        return current

def deleteNode(root, key):
        if root is None:
                return root
        if key < root.elem:
                root.left = deleteNode(root.left, key)
        if key > root.elem:
                root.right = deleteNode(root.right, key)
        else:
                # Case 1: No children
                if root.left is None and root.right is None:
                        root = None
```

```
                return root
            # Case 2: One child
            if root.left is None:  # Only right child exists
                temp = root.right
                root = None
                return temp
            elif root.right is None:  # Only left child exists
                temp = root.left
                root = None
                return temp
            # Case 3: Two children
            temp = leftMostNode(root.right)  #Leftmost of right subtree
            root.elem =  temp.elem  #Replace leftmost node with root
            root.right = deleteNode(root.right, temp.elem)
        return root
```

- **Balancing BST:**

```
def pushTreeNodes(root, arr):
    if root is None:
        return
    pushTreeNodes(root.left, arr)
    arr.append(root)
    pushTreeNodes(root.right, arr)

def buildBalancedBST(arr, start, end):
    if start > end:
        return None
    mid = (start + end) // 2
    root = Node(arr[mid])
    root.left = buildBalancedBST(arr, start, mid - 1)
    root.right = buildBalancedBST(arr, mid+1, end)
    return root
```

```
arr = [ ]
pushTreeNodes(root, arr)
newRoot = buildBalancedBST(arr, 0, len(arr) - 1)
```