Trees | Data Structures | Student Version

Topics to Cover

1. **Tree Basics [Class 1]**
    - 1.1 Introduction to Tree Data Structure
    - 1.2 Terminologies of a Tree
        - 1.2.1 Root/Leaf/Non-Leaf Node
        - 1.2.2 Parent/Child
        - 1.2.3 Siblings
        - 1.2.4 Edge/Path
        - 1.2.5 Degree
        - 1.2.6 Depth
        - 1.2.7 Height
        - 1.2.8 Level
        - 1.2.9 Subtree
    - 1.3 Characteristics of a Tree
    - 1.4 Tree Coding
        - 1.4.1 Tree Construction using Linked List (Dynamic Representation)
    - Practice Problems on Tree Basics

2. **Binary Trees [Class 2]**
    - 2.1 Characteristics of a Binary Tree
    - 2.2 Binary Tree Traversal: Pre-order, In-order, Post-order
    - 2.3 Types of Binary Trees
        - 2.3.1 Full Binary Tree
        - 2.3.2 Complete Binary tree
        - 2.3.3 Perfect Binary Tree
        - 2.3.4 Balanced Binary Tree
    - 2.4 Binary Tree Coding
        - 2.4.1 Tree Construction using Array (Sequential Representation)
        - 2.4.2 Level, Height, and Depth Finding
        - 2.4.3 Number of Nodes Finding
        - 2.4.4 Identifying Tree Types: Full, Complete and Perfect
    - Practice Problems on Binary Trees

3. **Binary Search Trees (BST) [Class 3]**
    - 3.1 Characteristics of a BST
    - 3.2 Basic Operations on a BST
        - 3.2.1 Creation of a BST
        - 3.2.2 Inserting a Node
        - 3.2.3 Removing a Node
        - 3.2.4 BST Traversal: Pre-order, In-order, Post-order
    - 3.3 Balanced vs Unbalanced BST
    - 3.4 BST Coding
        - 3.4.1 Creating a BST / Inserting a Node
        - 3.4.2 BST Traversal: Pre-order, In-order, Post-order
        - 3.4.3 Searching for an element
        - 3.4.4 Removing a Node
        - 3.4.5 Balancing BST
    - Practice Problems on BSTs

Trees | Data Structures | Student Version

1. Tree Basics

Tree is a non-linear data structure that allows us to organize, access and update data efficiently by representing non-linear data in a form of hierarchy.

1.1 Introduction to Tree Data Structure

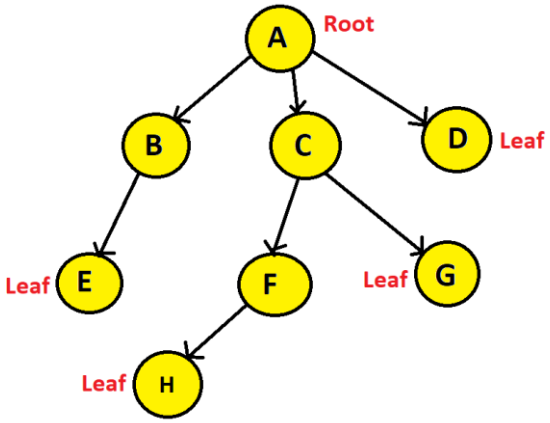
We have covered a number of data structures so far in this course. However, trees are extremely unique as some feats can only be achieved efficiently using trees. Following are some real life applications of trees:

- Continuous Sorting can be achieved using trees. Imagine you have a sorted array of 1 million data. Now you have to add 1000 more data into it. Then comes the responsibility of sorting the entire array again. However, using one variation of trees (Binary Search Tree), we can efficiently insert data later in a sorted manner without having to sort all the data again and again.
- Trees are widely used in folder/file system structure. For example, your desktop folder is the root of the tree and in your desktop you have multiple folders, and in those folders, you have some files. These files are the leaf nodes of the tree.
- Trees are used in electrical circuit designing and electricity transmission. For example, suppose you have a main center point from where all the electricity is generated and from there on it will be spread into branches until it reaches every home, office or industry.
- Router/Computer Network algorithms construct trees of the locations across the network to determine the route that data packets must follow to reach their destination efficiently.

1.2 Terminologies of a Tree

1.2.1 Root/Leaf/Non-Leaf Node:

Each tree consists of one or more nodes that are interlinked. The topmost node is called the **root** node. Below the root there are one or more nodes. The nodes residing at the bottom, or in other words the nodes that do not lead to any other node are called **leaf** nodes. If we have access to the root node, we have access to the entire tree.



Here we can see that node A is the root node, Nodes E, H, G, and D are the leaf nodes. Nodes A, B, F, and C are non-leaf nodes. The root node and the non-leaf nodes can be considered as internal nodes.

1.2.2 Parent/Child:

The direction of the tree is from top to bottom. Which means Node B is the immediate successor of node A, and Node A is the immediate predecessor of node B. Therefore, node B is the child of A, whereas, A is the parent of B.

1.2.3 Siblings:

All the nodes having the same parent are known as siblings. Therefore, B, C, and D are siblings, and F and G are siblings.

1.2.4 Edge/Path:

The link between two nodes is called an edge. A path is known as the consecutive edges from the source node to the destination node. So, if we asked what is the path from node A to E? The answer would be A→B→E. A tree having n number of nodes will have (n-1) number of edges. Here, we have 8 nodes and 7 edges in total.

1.2.5 Degree:

The number of children of a node is known as degree. The degree of node A is 3, node B is 1 and Node E is 0.

1.2.6 Depth:

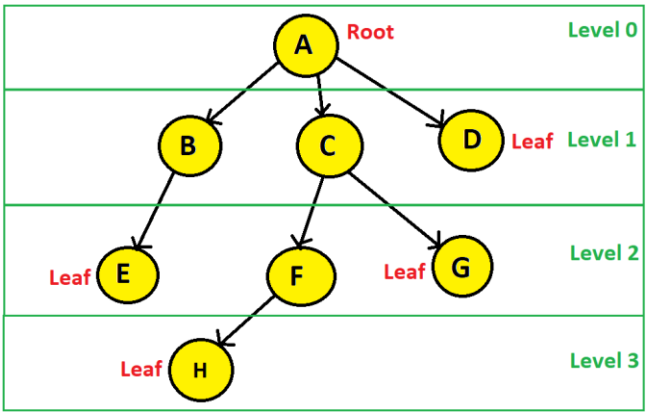
The length of the path from a node to the root node is known as the depth. The depth of nodes E, F, and G is 2; depth of B, C, and D is 1; depth of A is 0; depth of H is 3.

1.2.7 Height:

The length of the longest path from a node to one of its leaf nodes is known as the height. From node A to the leaf nodes there are four paths: A→B→E, A→C→F→H, A→C→G, and A→D. Of these four paths, A→C→F→H is the longest path. Hence, the height of Node A is 3.

1.2.8 Level:

Each hierarchy starting from the root is known as the level.



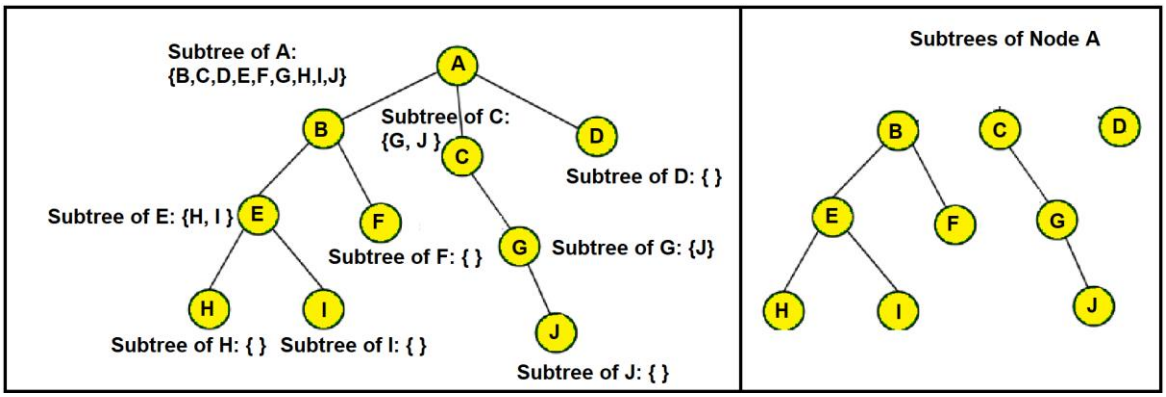
From the above figure, we can see that the level of node A is 0; level of nodes B, C, and D is 1; level of nodes E, F, and G is 2; and level of node H is 3.

Points to remember:

- *The depth and height of a node may not be the same. The depth of A is 0, whereas, the height of A is 3.*
- *Level of a node == Depth of that node.*

1.2.9 Subtree:

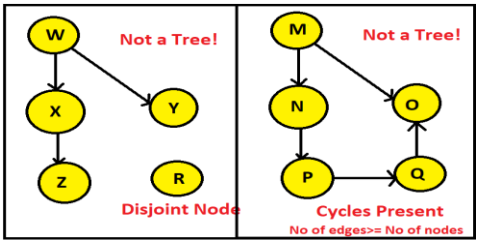
A tree that is the child of a Node.



Any tree can be further divided into subtrees with respect to a particular node. Here node A has three subtrees that are shown on the right side.

1.3 Characteristics of a Tree

- A tree must be continuous and not disjoint, which means every single node must be traversable starting from the root node.
- A tree cannot have a cycle. A tree having n number of nodes will have n or more edges if it contains one or more cycles. Which means, for a tree, no of edges == no of nodes - 1.

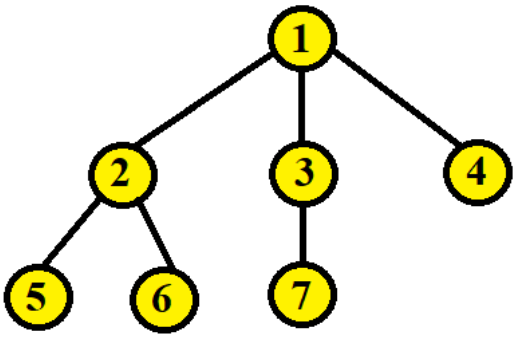


1.4 Tree Coding

1.4.1 Tree Construction using Linked List (Dynamic Representation)

A tree can be represented using a linked list (Dynamic Representation) or an array (Sequential Representation). Here we shall see how to dynamically represent trees.

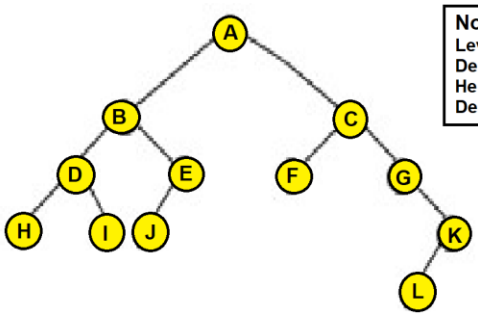
```
class Node:
    def __init__(self, elem):
        self.elem = elem
        self.children = []

root = Node(1)
root.children += [Node(2)]
root.children += [Node(3)]
root.children += [Node(4)]
root.children[0].children += [Node(5)]
root.children[0].children += [Node(6)]
root.children[1].children += [Node(7)]
```



Trees | Data Structures | Student Version

Practice Problems on Tree Basics

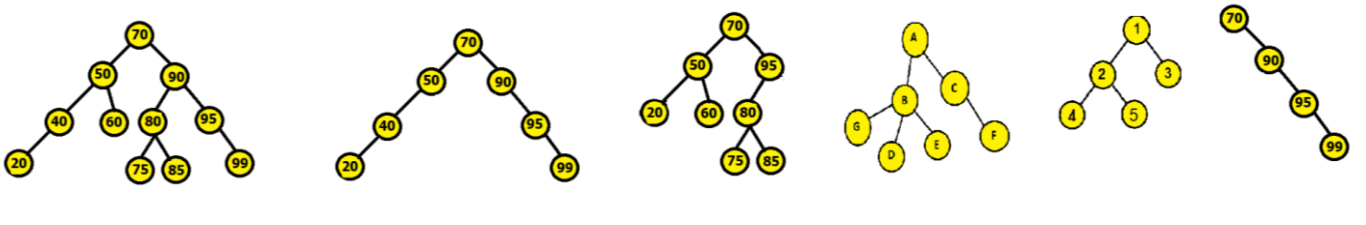
1. Find the level, depth, height and degree of the specified nodes of the following tree.



Node A	Node D	Node F	Node C
Level:	Level:	Level:	Level:
Depth:	Depth:	Depth:	Depth:
Height:	Height:	Height:	Height:
Degree:	Degree:	Degree:	Degree:

Node E	Node G	Node L
Level:	Level:	Level:
Depth:	Depth:	Depth:
Height:	Height:	Height:
Degree:	Degree:	Degree:

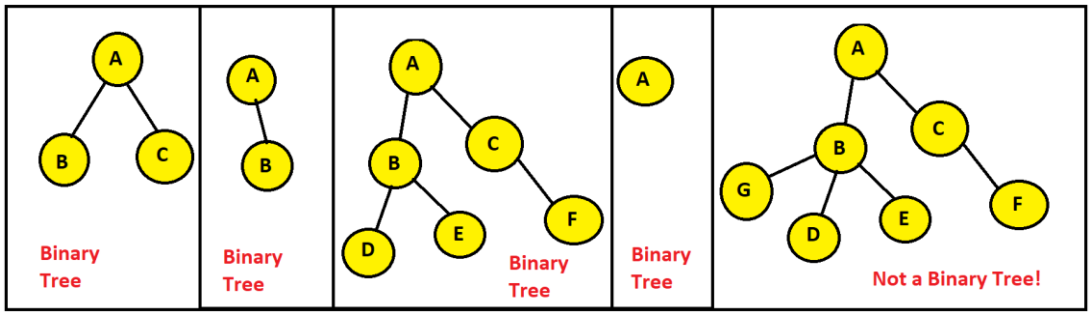
2. Identify which of the following trees are full, complete, perfect and balanced.



3. Write the code to construct a tree of height 3 and minimum number of 9 nodes. Use your imagination while designing the tree.

2. Binary Trees

A tree is a binary tree if every single node of the tree has at most 2 child nodes.



2.1 Characteristics of a Binary Tree

- Each node in a binary tree can have at most two child nodes
- If the number of internal nodes is n, number of external nodes is n+1, number of edges is 2n, number of internal edges is n-1, number of external edges is n+1.
- The maximum number of nodes possible in a binary tree of height 'h' is: 2h – 1
- The maximum number of nodes at height h is: $2^{h+1} - 1$
- The maximum number of nodes at level i is: $2^{}$

2.2 Binary Tree Traversal: Pre-order, In-order, Post-order

Pre-order

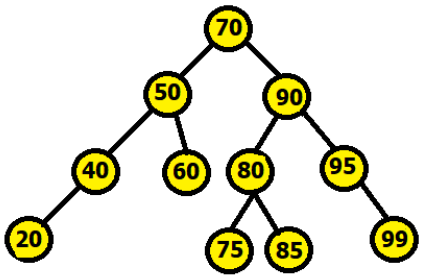
Whenever a node is visited for the **first** time, its element is printed. We start from the root and print its element. Then its left subtree is traversed. If a node does not have a left child, we return to that node After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the second time and check if it has any right child, and if it does not have a right child either, we again return to that node After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the third time and then go towards its parent node. After all the nodes of its entire left subtree have been traversed twice, we head back to the root node. After that, its right subtree is traversed. The root is printed at first.

In-order

Whenever a node is visited for the **second** time, its element is printed. We start from the root and traverse its left subtree. If a node does not have a left child, we return to that node for the second time and print its element. Then we check if it has any right child, and if it does not have a right child either, we again return to that node for the third time and then go towards its parent node. After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node for the second time and print its element. The root is printed after all the nodes of its left subtree are printed. After that, its right subtree is traversed. After all the nodes of its entire right subtree have been traversed thrice, we head back to the root node for the third time. **In-order traversal also sorts the data in ascending order.**

Post-order

Whenever a node is visited for the **third** time, its element is printed. We start from the root and traverse its left subtree. If a node does not have a left child, we return to that node for the second time and check if it has any right child, and if it does not have a right child either, we again return to that node for the third time, print its element and then go towards its parent node. After all the nodes of its entire left subtree have been traversed thrice, we head back to the root node and then traverse its right subtree. After all the nodes of its entire right subtree have been traversed thrice, we head back to the root node for the third time and print its element. The root is printed at the last.



Pre-Order: 70, 50, 40, 20, 60, 90, 80, 75, 85, 95, 99

In-Order: 20, 40, 50, 60, 70, 75, 80, 85, 90, 95, 99

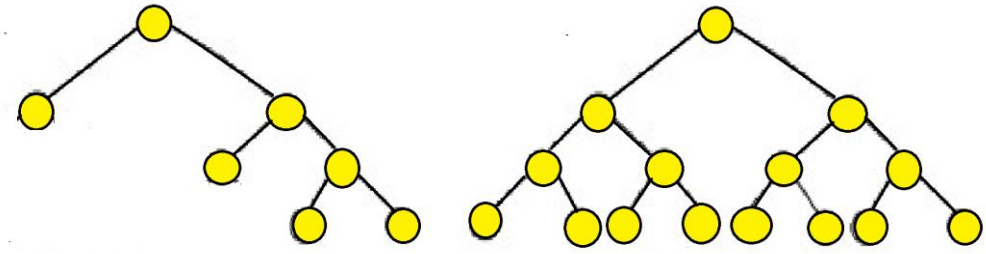
Post-Order: 20, 40, 60, 50, 75, 85, 80, 99, 95, 90, 70

2.3 Types of a Binary Tree

2.3.1 Full/Strict Binary Tree

In a full binary tree, internal nodes (every node except the leaf nodes) have two children. How can we identify a full binary tree? Any binary tree that maintains the following condition is a full binary tree:

No of leaf nodes = no of internal nodes + 1



Full Binary Tree

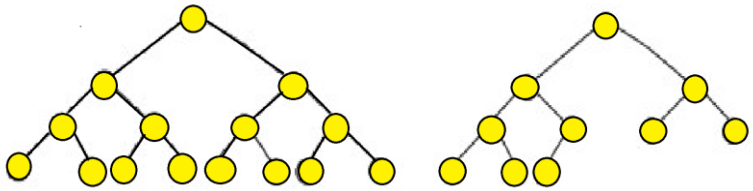
Here, in the leftmost tree, the no of internal nodes is 3 and the no of leaf nodes is 4. Again in the rightmost tree, the no of internal nodes is 7 and the no of leaf nodes is 8.

2.3.2 Complete Binary Tree

In a complete binary tree, all the levels are filled entirely with nodes, except the lowest level of the tree. Also, in the lowest level of this binary tree, every node should possibly reside on the left side.

How can we identify a complete binary tree?

- If all the internal nodes (every node except the leaf nodes) have two childs, then it is a complete binary tree.
- If any of the internal nodes has only one child, the child must reside in the left side and not in the right side.

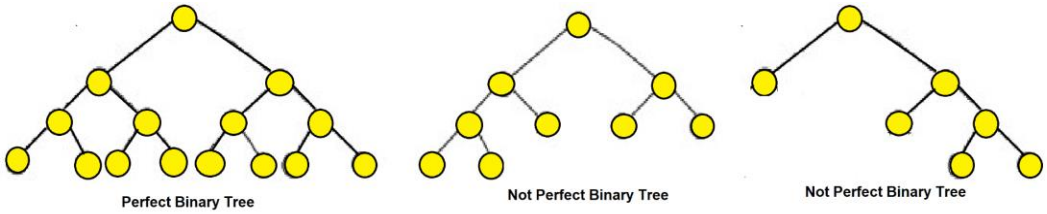


Complete Binary Tree

In the leftmost tree, all internal nodes have two children. Therefore, it is a complete binary tree. In the rightmost tree, all internal nodes except one have two children. The only internal node that has one child, has its child residing on its left side. Therefore, it is also a complete binary tree.

2.3.3 Perfect Binary Tree

In a perfect binary tree, every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Perfect Binary Tree

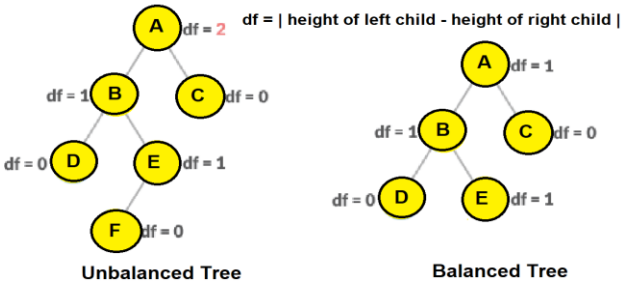
Not Perfect Binary Tree

Not Perfect Binary Tree

In the tree in the middle, all the leaf nodes are not on the same level. In the rightmost tree, not all internal nodes have two children.

2.3.4 Balanced Binary Tree

In a balanced binary tree, the height of the left and right subtree of any node differ by not more than 1. Balanced binary trees are also referred to as a height-balanced binary tree.



In the leftmost tree, the height of Node A's left subtree is 2 and right subtree is 0. Therefore, the difference between these two is 2. On the other hand, in the rightmost tree, no nodes have a height difference of more than 1 in between their left and right subtrees.

Points to remember:

- *Every perfect binary tree is also a full binary tree, but every full binary tree is not a perfect binary tree.*

Trees | Data Structures | Student Version

2.4 Binary Tree Coding

2.4.1 Tree Construction using Array (Sequential Representation)

Have you ever thought about how to represent binary trees in your program? If you can recall, we have been using linked lists so far to represent trees. So, there are two ways to represent binary trees:

- Dynamic Representation (Using Linked List)
- Sequential Representation (Using Array)

We have already covered the dynamic representation of trees. Now let us look at how to sequentially represent binary trees.

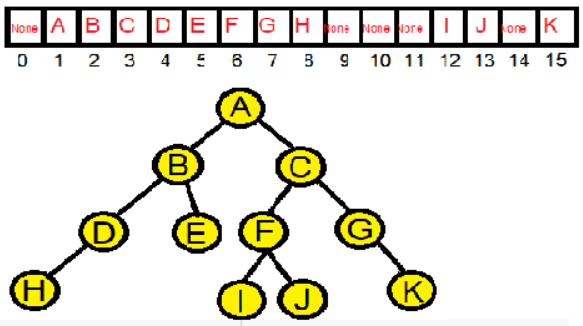
Sequential Representation (Using Array) Conditions:

- If the height of the binary tree if h, An array of maximum h**2 length is required.
- The root is placed at index 1.
- Any node that is placed at index i, will have its left child placed at 2i and its right child at 2i+1.

```
#Binary Tree from Array
class Node:
    def __init__(self, elem):
        self.elem = elem
        self.left = self.right = None

def tree_construction(arr, i, n):
    root = None
    if i < n:
        if (arr[i]!=None):
          root = Node(arr[i])
          root.left = tree_construction(arr, 2 * i, n) # insert left child
          root.right = tree_construction(arr, 2 * i + 1, n) # insert right child
    return root

def print_inOrder(root): # function to print tree nodes in InOrder fashion
    if root != None:
        print_inOrder(root.left)
        print(root.elem,end=" ")
        print_inOrder(root.right)

array_rep = [None, "A", "B", "C", "D", "E", "F", "G", "H", None, None, None, "I", "J", None, "K"]
root= tree_construction(array_rep, 1, len(array_rep))
print_inOrder(root)
```

```
H D B E A I F J C G K
```

```
#Array from Binary Tree
array_rep = [None] * 16 # Root height is 3, so total length= (3+1)**2

def array_construction(n,i):
  if n==None:
    return None
  else:
    array_rep[i]= n.elem
    array_construction(n.left, 2*i)
    array_construction(n.right, 2*i+1)

array_construction(root,1)
print(array_rep)
```




```
[None, 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', None, None, None, 'I', 'J', None, 'K']
```

Trees | Data Structures | Student Version

2.4.2 Level, Height, and Depth Finding [Complete by Yourself]

```python
def get_level(node, node_level, elem):
    if (node == None):
        return 0
    if (node.elem == elem):
        return node_level
    downlevel = get_level(node.left, node_level + 1, elem)
    if (downlevel != 0):
        return downlevel
    downlevel = get_level(node.right, node_level + 1, elem)
    return downlevel

print(get_level(root,0, 4))
'''Parameters are: root node, root node's level (0) and
the element of the node whose level is to be found'''
```

2

Pseudocode of get_height(node):
if, node is null, return -1
else, return 1 + maximum of
(recursive call with node's left child + recursive call with node's right child)

```python
def get_height(n):
    #Complete the code by youself
```

2

2

Since, level==depth, get_level function will also work for getting the depth of any node.

2.4.3 Number of Nodes Finding [Complete by Yourself]

```python
def get_no_of_nodes(n):
    #Complete the code by youself
```

Pseudocode of get_no_of_nodes (node):
if node is null, return 0
else, return 1 + recursive call with node's left child
+ recursive call with node's right child

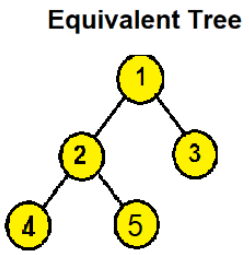
5

2.4.4 Identifying Tree Types: Full, Complete and Perfect

```python
# Count the number of nodes
def count_nodes(root):
    if root is None:
        return 0
    return (1 + count_nodes(root.left) + count_nodes(root.right))

# Check if the tree is a complete binary tree
def is_complete(root, index, numberNodes):
    if root is None: # Check if the tree is empty
        return True
    if index >= numberNodes:
        return False
    return (is_complete(root.left, 2 * index + 1, numberNodes)
    and is_complete(root.right, 2 * index + 2, numberNodes))

if is_complete(root, 0, count_nodes(root)):
    print("Complete binary tree")
else:
    print("Not a complete binary tree")
```

```
Complete binary tree
```

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

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```




```python
# Check if the tree is a full binary tree
def isFullTree(root):
    if root is None: # Tree empty case
        return True
    if root.left is None and root.right is None:
        return True
    if root.left is not None and root.right is not None:
        return (isFullTree(root.left) and isFullTree(root.right))
    return False

if isFullTree(root):
    print("Full binary tree")
else:
    print("Not a full binary tree")
```

```
Full binary tree
```

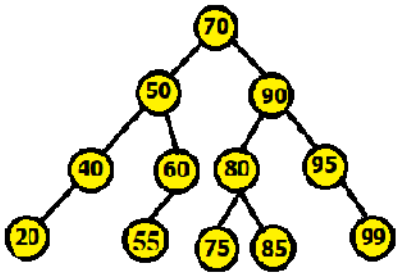
```python
# Calculate the depth
def calculateDepth(node):
    d = 0
    while (node is not None):
        d += 1
        node = node.left
    return d

# Check if the tree is a perfect binary tree
def is_perfect(root, d, level=0):
    if (root is None):# Check if the tree is empty
        return True
    if (root.left is None and root.right is None):
        return (d == level + 1)
    if (root.left is None or root.right is None):
        return False
    return (is_perfect(root.left, d, level + 1)
    and is_perfect(root.right, d, level + 1))

if (is_perfect(root, calculateDepth(root))):
    print("Perfect binary tree")
else:
    print("Not a perfect binary tree")
```

```
Not a perfect binary tree
```

Trees | Data Structures | Student Version

Practice Problems on Binary Trees

1. Traverse the following trees in pre-order, in-order and post-order and print the elements. Show both simulation and code.

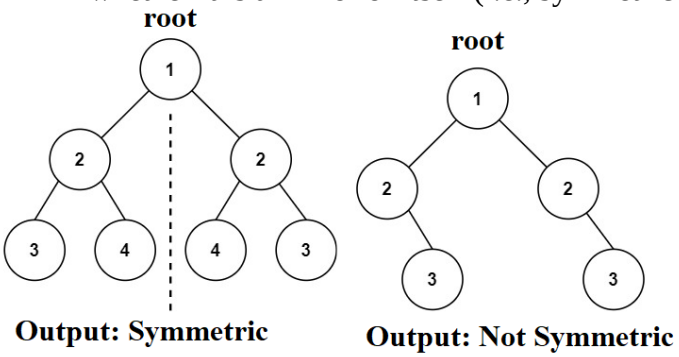


Pre-order:
In-order:
Post-order:

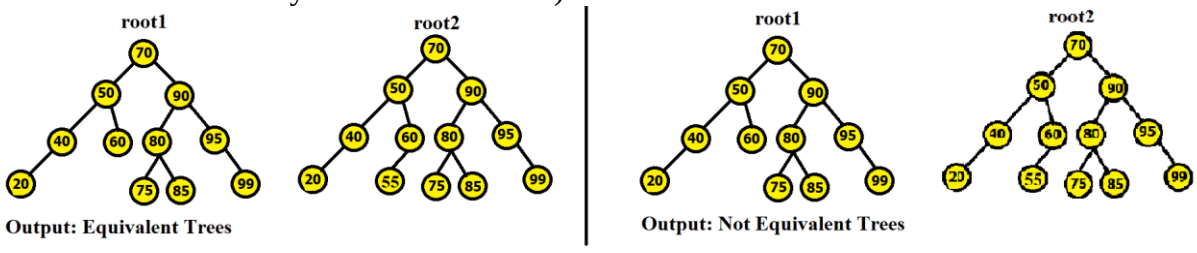
2. Consider the following array and convert it into a binary tree. Show simulation and code.

```
[None, 15, 25, 35, 10, 35, 15, 18, None, None, None, 33, None, 5, None, 19, None, None, None, 16]
```

3. Write a Python function **isSymmetric(root)** that takes the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).



4. Write a Python function **isIdentical(root1, root2)** that takes the roots of two binary trees, check whether they are identical or not).



3. Binary Search Trees (BST)

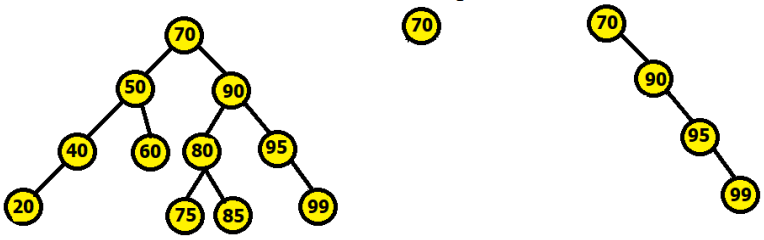
3.1 Characteristics of a BST

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 subtree each must also be a binary search tree.
- Each node must have a distinct key, which means no duplicate values are allowed.

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

BST Examples

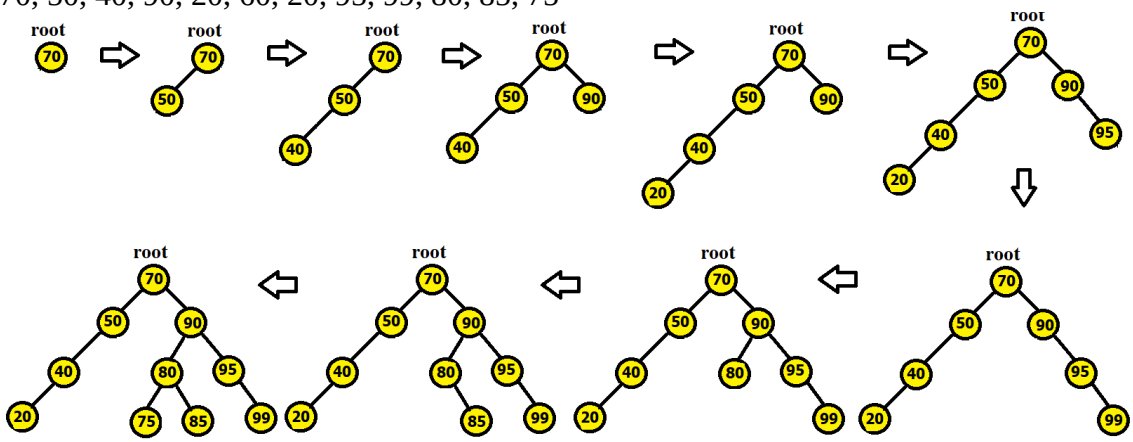


3.2 Basic Operations on a BST

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

3.2.1 Creation of a BST

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



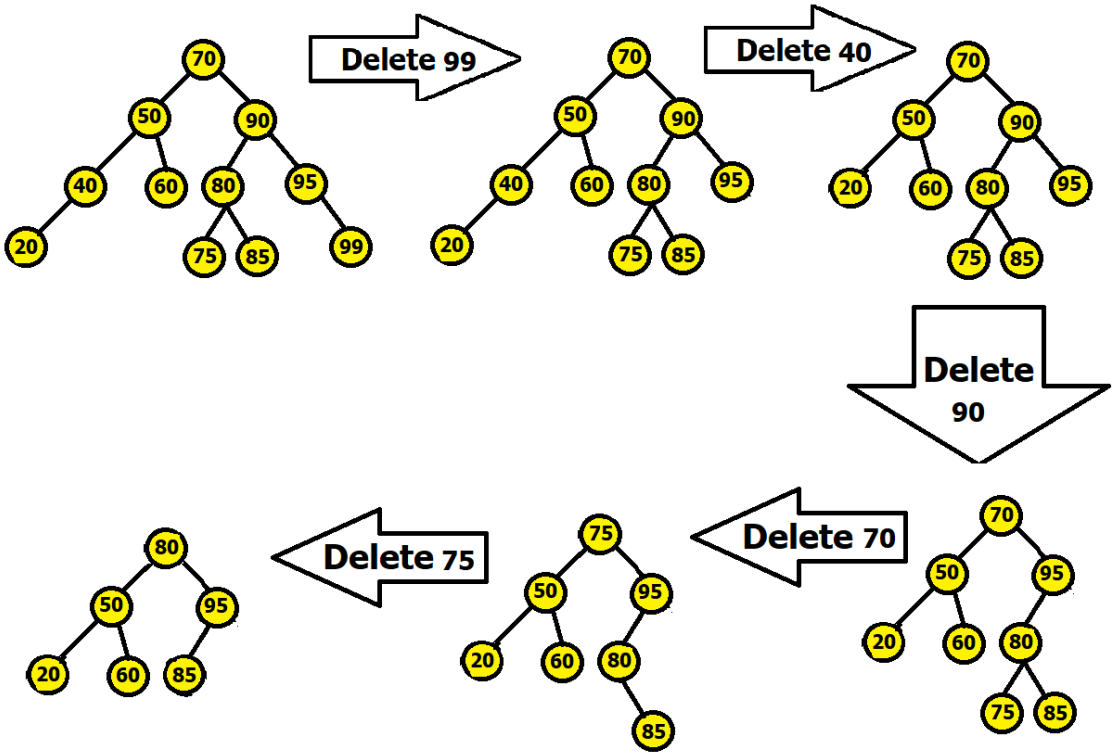
3.2.2 Inserting a Node

While inserting a node to an existing BST, the process is similar to that of creating a BST. We take the new node data which must not be repetitive or already present in the BST. We start comparing it with the root node, if the new node data is greater we go towards the right subtree of the root, if smaller we go towards the left subtree. We keep on going like this until we find a free space, and then make a node using the new node data and attach the new node at the vacant place. Note that, after insertion, a balanced BST may become **unbalanced** and therefore any searching operation done on it may take a lot longer than O(log(n)). We have to balance the tree if it becomes unbalanced.

3.2.3 Removing a Node

3 possible cases can occur while deleting a node:

- *Case 1* | No subtree or children: This one is the easiest one. You can simply just delete the node, without any additional actions required.
- *Case 2* | One subtree (one child): You have to make sure that after the node is deleted, its child is then connected to the deleted node's parent.
- *Case 3* | Two subtrees (two children): You have to find and replace the node you want to delete with its leftmost node in the right subtree (inorder successor) or rightmost node in the left subtree (inorder predecessor).



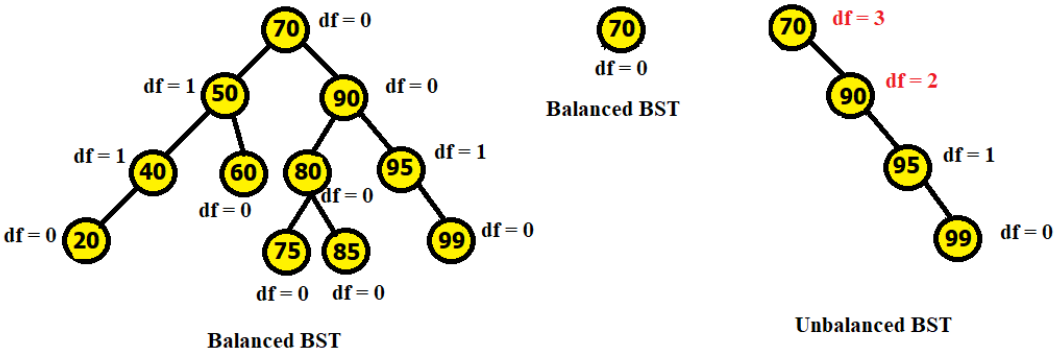
Here deletion of 99 falls under case 1, deletion of 40 falls under case 2, and deletion of 90, 70 and 75 fall under case 3. While deleting 75, we replaced 75 with its leftmost child from its right subtree, 80. After that 85 was put in the 80's previous place.

Note that, after deletion, a balanced BST may become **unbalanced** and therefore any searching operation done on it may take a lot longer than O(log(n)). We have to balance the tree if it becomes unbalanced.

3.3 Balanced vs Unbalanced BST

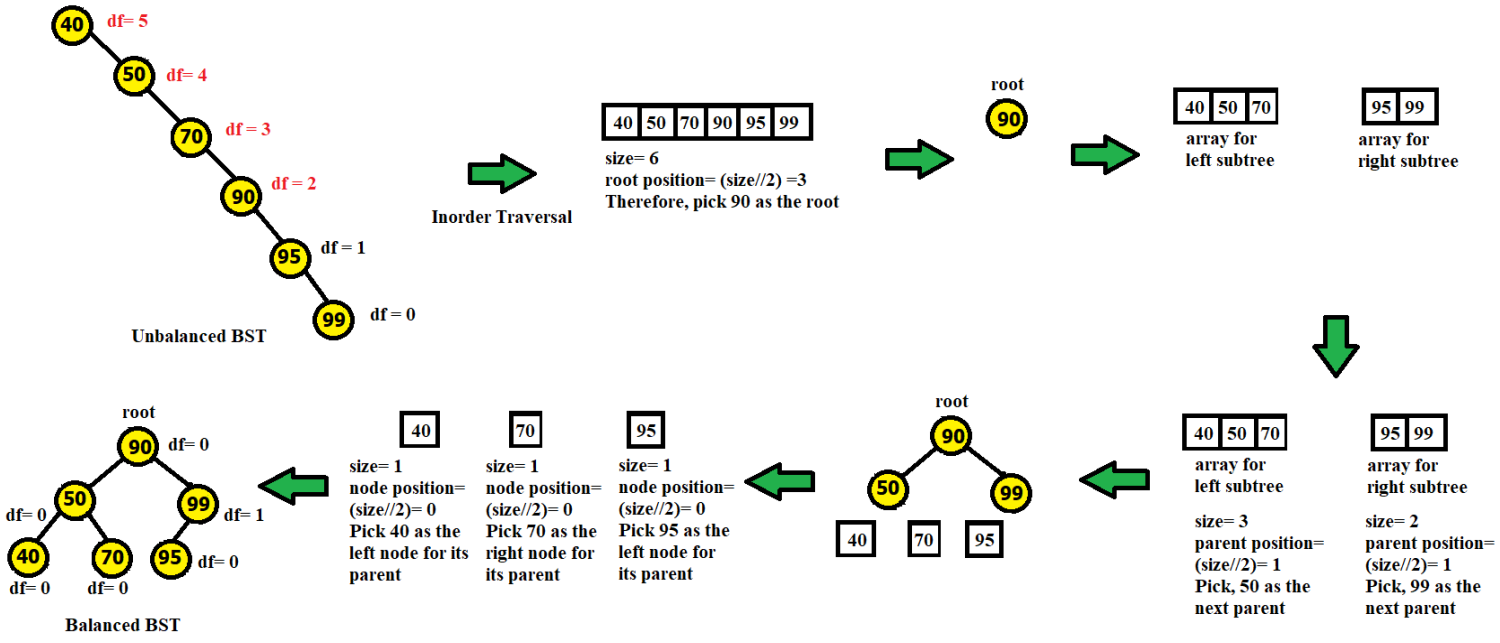
If the height difference between left and right subtree of any node in the BST is more than one, it is an unbalanced BST. Otherwise, it is a balanced one. In a balanced BST, any searching operation can take upto **O(log(n))** time complexity. In an unbalanced one, it may take upto **O(n)** time complexity, which renders the usage of BST 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.

df = | height of left child - height of right child |



How to convert an unbalanced BST into a balanced BST?

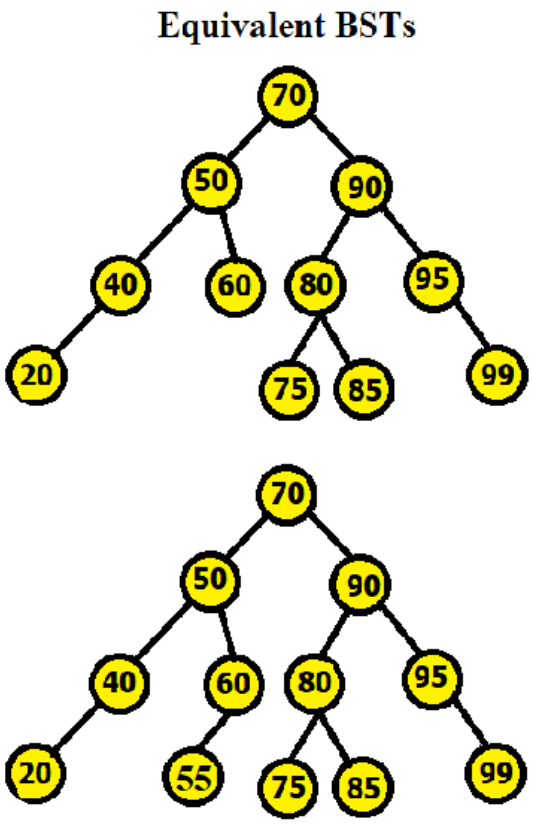
- Traverse given BST in inorder and store result in an array. This will give us the ascending sorted order of all the data.
- Now take the data in the (size//2) position in the array and make it the root. Now the left subtree of the root will be all the data residing in from 0 to (size//2)-1 positions of the array. The right subtree of the root will be all the data residing in from (size//2)+1 to (size-1) positions of the array.
- Now 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.



3.4 BST Coding

3.4.1 Creating a BST / Inserting a Node

```
#Creating a BST/Inserting in a BST
class Node:
    def __init__(self, elem):
        self.elem = elem
        self.left = self.right = None

def addNode(root, i): #Adding Nodes
  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)

def in_order(n): #To check the BST
  if n!=None:
    in_order(n.left)
    print(n.elem, end="--> ")
    in_order(n.right)

#Driver Code
l1=[70, 50, 40, 90, 20, 60, 20, 95, 99, 80, 85, 75]
root = Node(l1[0])
for i in l1[1:]:
  addNode(root, i)
in_order(root)
print()
addNode(root, 55)
in_order(root)
```




```
20--> 40--> 50--> 60--> 70--> 75--> 80--> 85--> 90--> 95--> 99-->
20--> 40--> 50--> 55--> 60--> 70--> 75--> 80--> 85--> 90--> 95--> 99-->
```

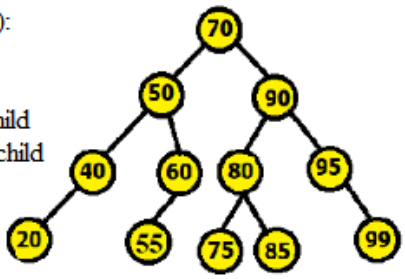
3.4.2 BST Traversal: Pre-order, In-order, Post-order [Complete by Yourself]

Pre-Order (Own | Left-Child | Right-Child)

```
[10]  1 def pre_order(n):
      2   if n!=None:
      3     print(n.elem, end="--> ")
      4     pre_order(n.left)
      5     pre_order(n.right)
      6 pre_order(root)
```

```
70--> 50--> 40--> 20--> 60--> 55--> 90--> 80--> 75--> 85--> 95--> 99-->
```

Pseudocode of pre_order (node):
if node is not null:
  print node's element
  recurive call with node's left child
  recurive call with node's right child



In-Order (Left-Child | Own | Right-Child)

```
[11]  1 def in_order(n):
          #Complete the code by yourself
```

Pseudocode of in_order (node):
if node is not null:
  recurive call with node's left child
  print node's element
  recurive call with node's right child

```
20--> 40--> 50--> 55--> 60--> 70--> 75--> 80--> 85--> 90--> 95--> 99-->
```

Post-Order (Left-Child | Right-Child | Own)

```
  1 def post_order(n):
        #Complete the code by yourself
```

Pseudocode of post_order (node):
if node is not null:
  recurive call with node's left child
  recurive call with node's right child
  print node's element

```
20--> 40--> 55--> 60--> 50--> 75--> 85--> 80--> 99--> 95--> 90--> 70-->
```

3.4.3 Searching for an element [Complete by Yourself]

Pseudocode:

search(root, item):

1. If root's element is equal to the item, return the root
2. If root's element is less than the item, and the root has a left child, return a recursive function call with root's left child and the item. If root's element is less than the item but there's no left child, return none
3. If root's element is greater than the item, and the root has a right child, return a recursive function call with root's right child and the item. If root's element is greater than the item but there's no right child, return none

```
1 def search(root, item):
      #Complete the code by yourself
```

```
13 x= search(root,70)
14 if x==None:
15   print("Not found")
16 else:
17   print("Found")
```

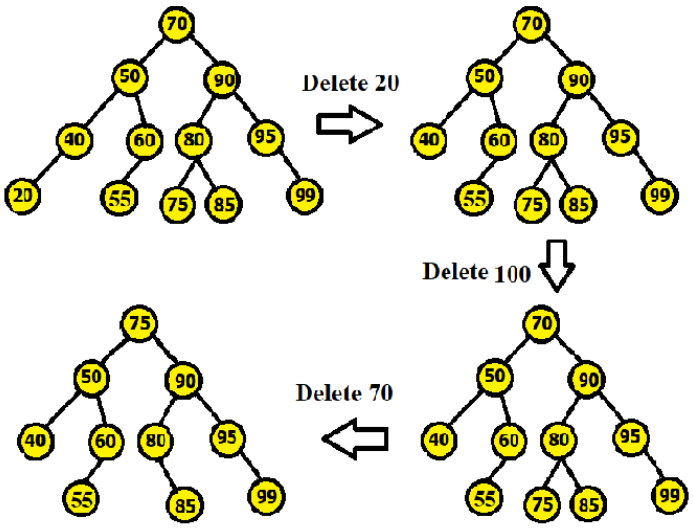
```
Found
```

3.4.4 Removing a Node

```python
def minValueNode(node):
    current = node
    while(current.left is not None): # loop down to find the leftmost leaf
        current = current.left
    return current
# Given a binary search tree and a key, this function
# delete the key and returns the new root


def deleteNode(root, key):
    if root is None:
        return root
    # If the key to be deleted is smaller than the root's key then it lies in  left subtree
    if key < root.elem:
        root.left = deleteNode(root.left, key)
    # If the kye to be delete is greater than the root's key then it lies in right subtree
    elif(key > root.elem):
        root.right = deleteNode(root.right, key)

    # If key is same as root's key, then this is the node to be deleted
    else:
        # Node with only one child or no child
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        # Node with two children:
        # Get the inorder successor (smallest in the right subtree)
        temp = minValueNode(root.right)
        # Copy the inorder successor's content to this node
        root.key = temp.elem
        root.right = deleteNode(root.right, temp.elem) # Delete the inorder successor
    return root
```

```python

print("Inorder traversal of the given tree")
in_order(root)
print("\nDelete 20")
root = deleteNode(root, 20)
print("Inorder traversal of the modified tree")
in_order(root)
print("\nDelete 100")
root = deleteNode(root, 100)
print("Inorder traversal of the modified tree")
in_order(root)
print("\nDelete 70")
root = deleteNode(root, 70)
print("Inorder traversal of the modified tree")
in_order(root)
```

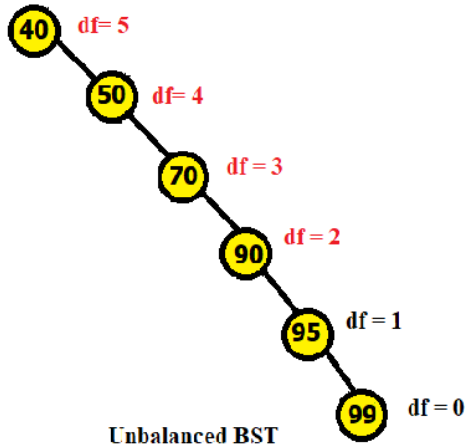
```
Inorder traversal of the given tree
20--> 40--> 50--> 60--> 70--> 75--> 80--> 85--> 90--> 95--> 99-->
Delete 20
Inorder traversal of the modified tree
40--> 50--> 60--> 70--> 75--> 80--> 85--> 90--> 95--> 99-->
Delete 100
Inorder traversal of the modified tree
40--> 50--> 60--> 70--> 75--> 80--> 85--> 90--> 95--> 99-->
Delete 70
Inorder traversal of the modified tree
40--> 50--> 60--> 70--> 80--> 85--> 90--> 95--> 99-->
```



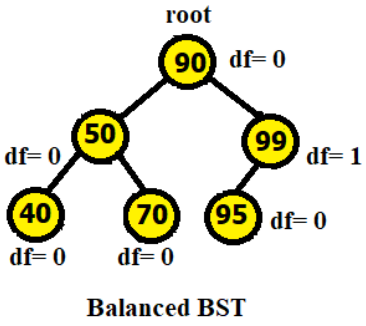
3.4.5 Balancing BST

For this, we shall need 3 recursive functions:

- **pushTreeNodes**: To push Nodes into an array/list in inorder (ascending order)
- **buildBalancedBST**: Finds the midpoint in the list and creates root, left child and right child
- **pre_order**: To print the tree in preorder fashion to see the differences made




```python
#Appends into a list with tree nodes using inorder traversal
def pushTreeNodes(root, arr):
    if root is None:
        return
    pushTreeNodes(root.left, arr)
    arr.append(root)
    pushTreeNodes(root.right, arr)

# Recursive function to construct a height-balanced BST from
# given nodes in sorted order
def buildBalancedBST(arr, start, end):
    if start > end:
        return None
    mid = (start + end) // 2 # find the middle index
    root = arr[mid] # The root node will be a node present at the mid-index
    # recursively construct left and right subtree
    root.left = buildBalancedBST(arr, start, mid - 1)
    root.right = buildBalancedBST(arr, mid + 1, end)
    return root

def pre_order(n): #preorder traversal
  if n!=None:
    print(n.elem, end="--> ")
    pre_order(n.left)
    pre_order(n.right)

print(f"Unbalance State Pre-Order:")
pre_order(root) #preorder traversal
arr = []
pushTreeNodes(root, arr)
newRoot= buildBalancedBST(arr, 0, len(arr) - 1)
print(f"\nBalanced State Pre-Order:")
pre_order(newRoot)
```

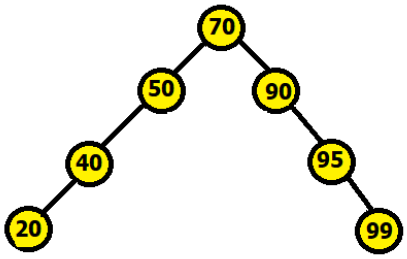


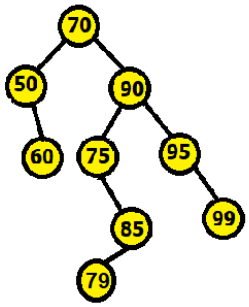

```
Unbalance State Pre-Order:
40--> 50--> 70--> 90--> 95--> 99-->
Balanced State Pre-Order:
70--> 40--> 50--> 95--> 90--> 99-->
```

Trees | Data Structures | Student Version

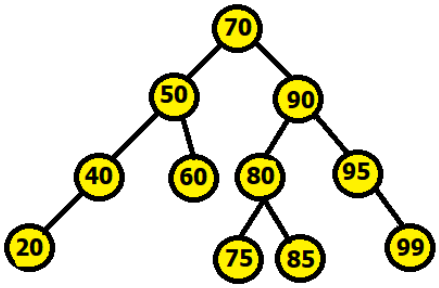
Practice Problems on BSTs

1. Convert the following unbalanced BSTs into balanced BSTs. Show simulation.

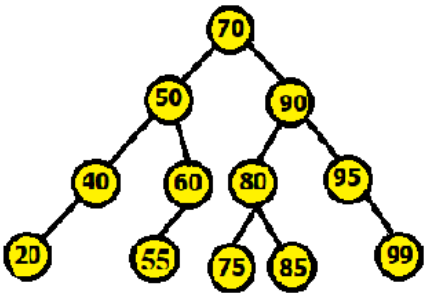




2. Insert keys 65, 105, 69 into the following BST and show the steps. Show simulation and code.

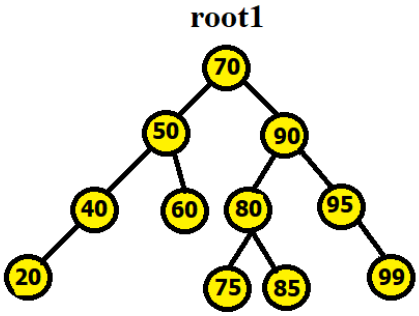


3. Delete keys 20, 95, 50, 70, 75 into the following BST and show the steps. Show simulation and code..



4. How can you print the contents of a tree in descending order with and without using stack? Solve using code.

5. Write a python program that takes the root of a tree and finds its inorder successor and predecessor.

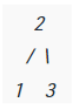


Output: In-order Successor: 75
In-order Predecessor: 60

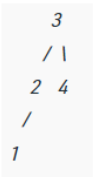
6. Given a sorted array, write a function that creates a Balanced Binary Search Tree using array elements. Follow the steps mentioned below to implement the approach:
    a. Set The middle element of the array as root.
    b. Recursively do the same for the left half and right half.
        i. Get the middle of the left half and make it the left child of the root created in step 1.
        ii. Get the middle of the right half and make it the right child of the root created in step 1.
    c. Print the preorder of the tree.

Given Array#1: [1, 2, 3]
Output: Pre-order of created BST: 2 1 3

BST#1



BST#2



Given Array#2: [1, 2, 3,4]
Output: Pre-order of created BST: 3 2 1 4