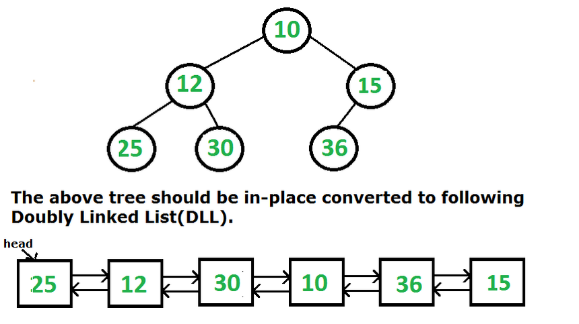
**DSA – ASSIGNMENT 22**

💡 1. Given a Binary Tree (Bt), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be the same as in Inorder for the given Binary Tree. The first node of Inorder traversal (leftmost node in BT) must be the head node of the DLL.

Example:



**Solution. :-**

* Define a class Node to represent each node in the binary tree. Each node will have a value, left pointer, and right pointer.
* Initialize a global variable prev as None. This variable will keep track of the previously visited node during the inorder traversal.
* Define a function binaryTreeToDLL that takes the root node of the binary tree as an argument.
* In the binaryTreeToDLL function:
  + Check if the current node is None. If so, return.
  + Recursively convert the left subtree by calling binaryTreeToDLL with the left child of the current node.
  + If prev is None, it means we are at the leftmost node of the inorder traversal. Set the head of the DLL as the current node.
  + Otherwise, set the right pointer of prev to the current node and the left pointer of the current node to prev.
  + Update prev to the current node.
  + Recursively convert the right subtree by calling binaryTreeToDLL with the right child of the current node.
* After the binaryTreeToDLL function finishes, the DLL will be formed.
* Return the head node of the DLL.

**class Node:**

**def \_\_init\_\_(self, value):**

**self.data = value**

**self.left = None**

**self.right = None**

**def binaryTreeToDLL(root):**

**global prev**

**if root is None:**

**return**

**binaryTreeToDLL(root.left)**

**if prev is None:**

**head = root**

**else:**

**root.left = prev**

**prev.right = root**

**prev = root**

**binaryTreeToDLL(root.right)**

**return head**

**# Example usage**

**# Create a binary tree**

**root = Node(10)**

**root.left = Node(12)**

**root.right = Node(15)**

**root.left.left = Node(25)**

**root.left.right = Node(30)**

**root.right.left = Node(36)**

**# Convert the binary tree to DLL**

**head = binaryTreeToDLL(root)**

**# Traverse the DLL and print the values**

**current = head**

**while current is not None:**

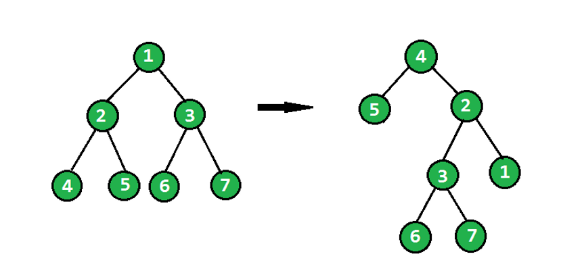
**print(current.data, end=" ")**

**current = current.right**

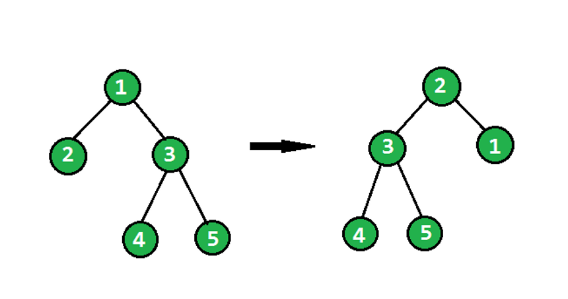
💡 2. A Given a binary tree, the task is to flip the binary tree towards the right direction that is clockwise. See the below examples to see the transformation.

In the flip operation, the leftmost node becomes the root of the flipped tree and its parent becomes its right child and the right sibling becomes its left child and the same should be done for all left most nodes recursively.

Example1:



Exampl2:



**Solution. :-**

* Define a class Node to represent each node in the binary tree. Each node will have a value, left pointer, and right pointer.
* Define a function flipBinaryTree that takes the root node of the binary tree as an argument.
* In the flipBinaryTree function:
  + Check if the current node is None or if it is a leaf node (i.e., both left and right pointers are None). If so, return the current node.
  + Recursively flip the left subtree by calling flipBinaryTree with the left child of the current node.
  + Store the right child of the current node in a temporary variable.
  + Set the right pointer of the current node to None.
  + Recursively flip the right subtree by calling flipBinaryTree with the temporary variable (the original right child).
  + Set the left pointer of the current node to the flipped right subtree.
  + Return the root node of the flipped tree.

**class Node:**

**def \_\_init\_\_(self, value):**

**self.data = value**

**self.left = None**

**self.right = None**

**def flipBinaryTree(root):**

**if root is None or (root.left is None and root.right is None):**

**return root**

**flippedLeft = flipBinaryTree(root.left)**

**flippedRight = flipBinaryTree(root.right)**

**root.left = flippedRight**

**root.right = None**

**current = flippedLeft**

**while current is not None and current.right is not None:**

**current = current.right**

**if current is not None:**

**current.right = root**

**return flippedLeft**

**# Example usage**

**# Create a binary tree**

**root = Node(1)**

**root.left = Node(2)**

**root.right = Node(3)**

**root.left.left = Node(4)**

**root.left.right = Node(5)**

**root. right.left = Node(6)**

**root. right.right = Node(7)**

**# Flip the binary tree**

**flippedRoot = flipBinaryTree(root)**

**# Print the flipped binary tree**

**def printFlippedTree(root):**

**if root is None:**

**return**

**print(root.data, end=" ")**

**printFlippedTree(root.right)**

**printFlippedTree(flippedRoot)**

💡 3. Given a binary tree, print all its root-to-leaf paths without using recursion. For example, consider the following Binary Tree.

Input:

6

/ \\

3 5

/ \ \ 2 5 4 / \ 7 4

Output:

There are 4 leaves, hence 4 root to leaf paths - 6->3->2 6->3->5->7 6->3->5->4 6->5>4

**Solution. :-**

* Define a class Node to represent each node in the binary tree. Each node will have a value, left pointer, and right pointer.
* Define a function printRootToLeafPaths that takes the root node of the binary tree as an argument.
* Create an empty stack and push a tuple containing the root node and an empty string representing the path onto the stack.
* While the stack is not empty, do the following:
  + Pop the top element from the stack. Let the popped element be currentNode and currentPath.
  + If currentNode is a leaf node (i.e., both left and right pointers are None), print the currentPath followed by the value of currentNode.data.
  + If currentNode.right is not None, push a tuple containing currentNode.right and currentPath + "->" + str(currentNode.data) onto the stack.
  + If currentNode.left is not None, push a tuple containing currentNode.left and currentPath + "->" + str(currentNode.data) onto the stack.
* Repeat steps 4 until the stack is empty.
* The output will be the printed root-to-leaf paths.

**class Node:**

**def \_\_init\_\_(self, value):**

**self.data = value**

**self.left = None**

**self.right = None**

**def printRootToLeafPaths(root):**

**if root is None:**

**return**

**stack = [(root, "")]**

**while stack:**

**currentNode, currentPath = stack.pop()**

**if currentNode.left is None and currentNode.right is None:**

**print(currentPath + "->" + str(currentNode.data))**

**if currentNode.right is not None:**

**stack.append((currentNode.right, currentPath + "->" + str(currentNode.data)))**

**if currentNode.left is not None:**

**stack.append((currentNode.left, currentPath + "->" + str(currentNode.data)))**

**# Example usage**

**# Create a binary tree**

**root = Node(6)**

**root.left = Node(3)**

**root.right = Node(5)**

**root.left.left = Node(2)**

**root.left.right = Node(5)**

**root.right.right = Node(4)**

**root.right.right.left = Node(7)**

**root.right.right.right = Node(4)**

**# Print all root-to-leaf paths**

**printRootToLeafPaths(root)**

💡 4. Given Preorder, Inorder and Postorder traversals of some tree. Write a program to check if they all are of the same tree.

**Examples:**

Input :

Inorder -> 4 2 5 1 3

Preorder -> 1 2 4 5 3

Postorder -> 4 5 2 3 1

Output :Yes

Explanation :

All of the above three traversals are of the same tree

1

/ \\

2 3

/ \\

4 5

Input :

Inorder -> 4 2 5 1 3

Preorder -> 1 5 4 2 3

Postorder -> 4 1 2 3 5

Output :No

**Solution. :-**

* Define a class Node to represent each node in the binary tree. Each node will have a value, left pointer, and right pointer.
* Define a function checkTraversal that takes the preorder, inorder, and postorder traversals as arguments.
* If any of the given traversals is empty, return True if all traversals are empty, indicating that they represent the same empty tree.
* Extract the root value from the preorder traversal (the first element).
* Create a new node with the extracted root value.
* Find the index of the root value in the inorder traversal. Let this index be rootIndex.
* Recursively call checkTraversal with the following arguments:
  + For the left subtree:
    - Preorder traversal: sublist from the second element to rootIndex in the preorder traversal.
    - Inorder traversal: sublist from the first element to rootIndex - 1 in the inorder traversal.
    - Postorder traversal: sublist from the first element to rootIndex - 1 in the postorder traversal.
  + For the right subtree:
    - Preorder traversal: sublist from rootIndex + 1 to the end of the preorder traversal.
    - Inorder traversal: sublist from rootIndex + 1 to the end of the inorder traversal.
    - Postorder traversal: sublist from rootIndex to the second last element in the postorder traversal.
* Return True if both recursive calls return True and the root value extracted from the preorder traversal matches the root value from the postorder traversal. Otherwise, return False.

**class Node:**

**def \_\_init\_\_(self, value):**

**self.data = value**

**self.left = None**

**self.right = None**

**def checkTraversal(preorder, inorder, postorder):**

**if not preorder and not inorder and not postorder:**

**return True**

**rootValue = preorder[0]**

**rootIndex = inorder.index(rootValue)**

**leftPreorder = preorder[1:rootIndex+1]**

**leftInorder = inorder[:rootIndex]**

**leftPostorder = postorder[:rootIndex]**

**rightPreorder = preorder[rootIndex+1:]**

**rightInorder = inorder[rootIndex+1:]**

**rightPostorder = postorder[rootIndex:-1]**

**return checkTraversal(leftPreorder, leftInorder, leftPostorder) and checkTraversal(rightPreorder, rightInorder, rightPostorder) and (rootValue == postorder[-1])**

**# Example usage**

**inorder = [4, 2, 5, 1, 3]**

**preorder = [1, 2, 4, 5, 3]**

**postorder = [4, 5, 2, 3, 1]**

**if checkTraversal(preorder, inorder, postorder):**

**print("Yes")**

**else:**

**print("No")**