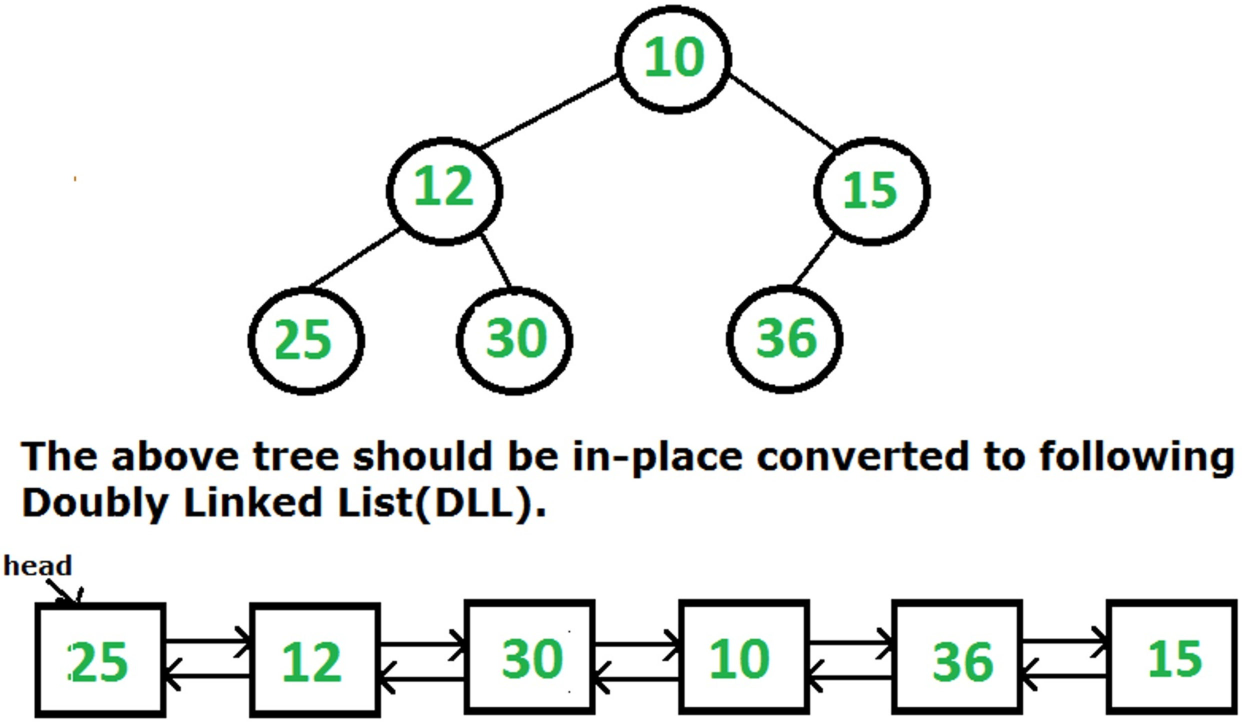
Assignment 22 Solutions

Q1. 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:

In [1]:

Out[1]:



The above tree should be in-place converted to following Doubly Linked List(DLL).

In [25]:

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


def binary_tree_to_dll(root):
    if root is None:
        return None

    # Helper function to perform inorder traversal
    def inorder(node):
        nonlocal prev, head

        if node is None:
            return

        # Recursively convert left subtree
        inorder(node.left)

        # Update the links for the doubly linked list
        if prev:
            prev.right = node
            node.left = prev
        else:
            # First node in inorder traversal, set it as the head of the DLL
            head = node

        prev = node

        # Recursively convert right subtree
        inorder(node.right)
    # Initialize variables
    prev = None
    head = None

    # Perform inorder traversal to convert the tree to DLL
    inorder(root)

    return head


def print_dll(head):
    while head:
        print(head.data, end=" ")
        head = head.right
    print()
```

In [30]:

```
# 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 binary tree to DLL
head = binary_tree_to_dll(root)

# Print the DLL
print_dll(head)
```

```
25 12 30 10 36 15
```

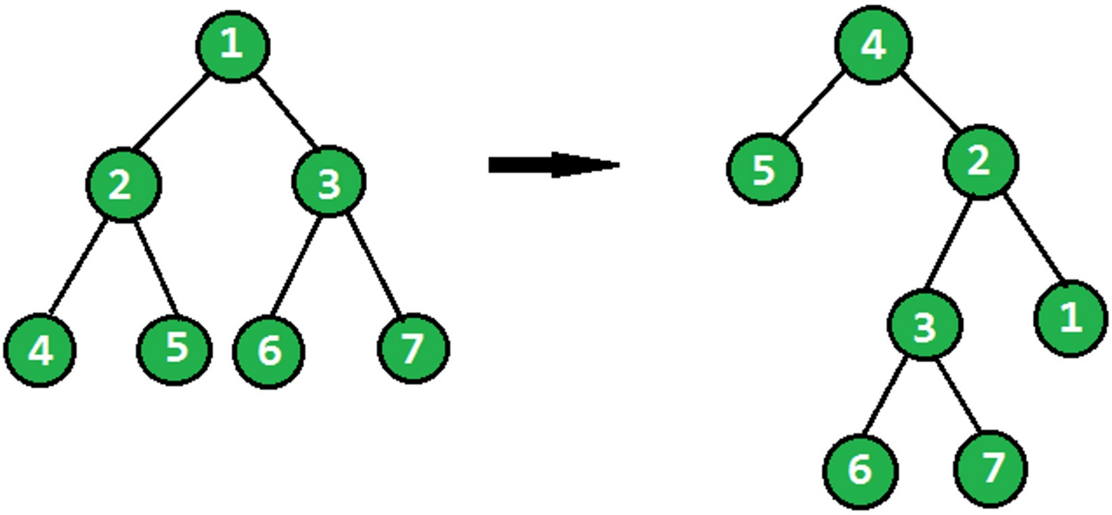
Q2. A Given a binary tree, the task is to flip the binary tree towards the right direction that is clockwise. See the below examples to see the transformation.

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:

In [32]:

Out[32]:

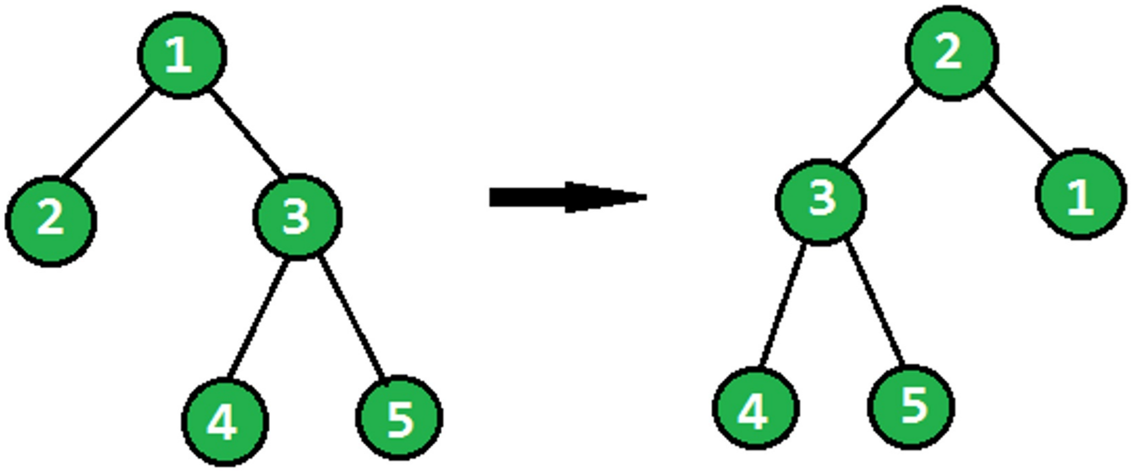


Example2:

In [34]:

```
from IPython.display import Image
Image(r"C:\Users\hrush\OneDrive\Pictures\Saved Pictures\Untitled (2).png")
```

Out[34]:



In [44]:

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

def flipBinaryTree(root):
    if root is None or (root.left is None and root.right is None):
        # Base case: empty tree or leaf node
        return root

    flippedLeft = flipBinaryTree(root.left)

    root.left.left = root.right
    root.left.right = root
    root.left = None
    root.right = None

    return flippedLeft
```

In [49]:

```
# Example 1
# Create the binary tree
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

# Print the original tree
print("Original Tree:")
# Perform an in-order traversal to print the tree
def inorderTraversal(node):
    if node is None:
        return
    inorderTraversal(node.left)
    print(node.data, end=" ")
    inorderTraversal(node.right)
inorderTraversal(root)
print()

# Flip the binary tree
flippedRoot = flipBinaryTree(root)

# Print the flipped tree
print("Flipped Tree:")
inorderTraversal(flippedRoot)
```

```
Original Tree:
4 2 5 1 3
Flipped Tree:
5 4 3 2 1
```

In [70]:

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

def flipBinaryTree(root):
    if root is None or (root.left is None and root.right is None):
        # Base case: empty tree or leaf node
        return root

    flippedLeft = flipBinaryTree(root.left)

    root.left.left = root.right
    root.left.right = root
    root.left = None
    root.right = None

    return flippedLeft
```

In [71]:

```
# Example usage
# Create the 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)

# Print the original tree
print("Original Tree:")
# Perform an in-order traversal to print the tree
def inorderTraversal(node):
    if node is None:
        return
    inorderTraversal(node.left)
    print(node.data, end=" ")
    inorderTraversal(node.right)
inorderTraversal(root)
print()

# Flip the binary tree
flippedRoot = flipBinaryTree(root)

# Print the flipped tree
print("Flipped Tree:")
inorderTraversal(flippedRoot)
```

```
Original Tree:
4 2 5 1 6 3 7
Flipped Tree:
5 4 6 3 7 2 1
```

Q3. 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

In [72]:

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

def printRootToLeafPaths(root):
    if root is None:
        return

    stack = [(root, str(root.data))]  # stack to perform iterative traversal
    paths = []  # store the paths

    while stack:
        node, path = stack.pop()

        if node.left is None and node.right is None:
            # leaf node, add the path to the list
            paths.append(path)
        if node.right is not None:
            stack.append((node.right, path + "->" + str(node.right.data)))
        if node.left is not None:
            stack.append((node.left, path + "->" + str(node.left.data)))

    # Print the paths
    for path in paths:
        print(path)
```

In [73]:

```
# Example usage
# Create the 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.left.right.left = Node(7)
root.left.right.right = Node(4)

# Print all root-to-leaf paths
printRootToLeafPaths(root)
```

```
6->3->2
6->3->5->7
6->3->5->4
6->5->4
```

Q4. 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
```

In [79]:

```python
def areTraversalsSame(inorder, preorder, postorder):
    if not inorder or not preorder or not postorder:
        return False

    if len(inorder) == 1 and len(preorder) == 1 and len(postorder) == 1:
        return inorder[0] == preorder[0] == postorder[0]

    root = preorder[0]
    rootIndex = inorder.index(root)

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

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

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

    leftSame = areTraversalsSame(leftInorder, leftPreorder, leftPostorder)
    rightSame = areTraversalsSame(rightInorder, rightPreorder, rightPostorder)

    return leftSame and rightSame and (root == postorder[-1])
```

In [80]:

```python
# Example usage
inorder = [4, 2, 5, 1, 3]
preorder = [1, 2, 4, 5, 3]
postorder = [4, 5, 2, 3, 1]

if areTraversalsSame(inorder, preorder, postorder):
    print("Yes")
else:
    print("No")
```

```
Yes
```

In [81]:

```python
def areTraversalsSame(inorder, preorder, postorder):
    if not inorder or not preorder or not postorder:
        return False

    if len(inorder) == 1 and len(preorder) == 1 and len(postorder) == 1:
        return inorder[0] == preorder[0] == postorder[0]

    root = preorder[0]
    rootIndex = inorder.index(root)

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

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

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

    leftSame = areTraversalsSame(leftInorder, leftPreorder, leftPostorder)
    rightSame = areTraversalsSame(rightInorder, rightPreorder, rightPostorder)

    return leftSame and rightSame and (root == postorder[-1])
```

In [82]:

```python
# Example usage
inorder = [4, 2, 5, 1, 3]
preorder = [1, 5, 4, 2, 3]
postorder = [4, 1, 2, 3, 5]

if areTraversalsSame(inorder, preorder, postorder):
    print("Yes")
else:
    print("No")
```

```
No
```

In [ ]: