 # Python3 Program to print sum of all the elements of a binary tree

```
# Binary Tree Node

""" utility that allocates a new Node
with the given key """
class newNode:

        # Construct to create a new node
        def __init__(self, key):
                self.key = key
                self.left = None
                self.right = None
```

 # Function to find sum of all the element

```
def addBT(root):
        if (root == None):
                return 0
        return (root.key + addBT(root.left) + addBT(root.right))
```

```
# Driver Code
if __name__ == '__main__':
        root = newNode(1)
        root.left = newNode(2)
        root.right = newNode(3)
        root.left.left = newNode(4)
        root.left.right = newNode(5)
        root.right.left = newNode(6)
        root.right.right = newNode(7)
        root.right.left.right = newNode(8)

        sum = addBT(root)

        print("Sum of all the nodes is:", sum)
```

Write a Program to perform Sum of leaf nodes of a binary tree

```
# Class for node creation
class Node:

        # Constructor
        def __init__(self, data):
                self.data = data
                self.left = None
                self.right = None

# Utility function to calculate the sum of all leaf nodes
def leafSum(root):
        global total
        if root is None:
                return
        if (root.left is None and root.right is None):
                total += root.data
        leafSum(root.left)
        leafSum(root.right)



# Binary tree Fromation
if __name__=='__main__':
        root = Node(1)
        root.left = Node(2)
        root.left.left = Node(4)
        root.left.right = Node(5)
        root.right = Node(3)
        root.right.right = Node(7)
        root.right.left = Node(6)
        root.right.left.right = Node(8)

# Variable to store the sum of leaf nodes
        total = 0
        leafSum(root)
# Printing the calculated sum
        print(total)
```

Given two binary trees, the task is to find if both of them are identical or not.

Example 1:

```
Input:
     1          1
   /   \      /   \
  2     3    2     3
Output: Yes
Explanation: There are two trees both
having 3 nodes and 2 edges, both trees
are identical having the root as 1,
left child of 1 is 2 and right child
of 1 is 3.
```

Example 2:

```
Input:
    1       1
  /  \     /  \
 2    3   3    2
Output: No
Explanation: There are two trees both
having 3 nodes and 2 edges, but both
trees are not identical.
```

Algorithm:

```
sameTree(tree1, tree2)

1. If both trees are empty then return 1.

2. Else If both trees are non -empty

     (a) Check data of the root nodes (tree1->data ==  tree2->data)

     (b) Check left subtrees recursively  i.e., call sameTree( tree1->left_subtree,
tree2->left_subtree)

     (c) Check right subtrees recursively  i.e., call sameTree( tree1->right_subtree,
tree2->right_subtree)

     (d) If a,b and c are true then return 1.

3  Else return 0 (one is empty and other is not)
```

```
def identicalTrees(a, b):

    if a is None and b is None:
        return True

    if a is not None and b is not None:
        return ((a.data == b.data) and
                identicalTrees(a.left, b.left)and
                identicalTrees(a.right, b.right))

    return False
```

The height of a tree is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
def height(node):
    if node is None:
        return 0
    else :
        # Compute the height of each subtree
        lheight = height(node.left)
        rheight = height(node.right)

        #Use the larger one
        if lheight > rheight :
            return lheight+1
        else:
            return rheight+1
```

Find level order traversal of Binary Tree

```
# A node structure
class Node:

        # A utility function to create a new node
        def __init__(self, key):
                self.data = key
                self.left = None
                self.right = None


# Function to print level order traversal of tree
def printLevelOrder(root):
        h = height(root)
        for i in range(1, h+1):
                printCurrentLevel(root, i)


# Print nodes at a current level
def printCurrentLevel(root , level):
        if root is None:
                return
        if level == 1:
                print(root.data,end=" ")
        elif level > 1 :
                printCurrentLevel(root.left , level-1)
                printCurrentLevel(root.right , level-1)


""" Compute the height of a tree--the number of nodes
        along the longest path from the root node down to
        the farthest leaf node
"""
def height(node):
        if node is None:
                return 0
        else :
                # Compute the height of each subtree
                lheight = height(node.left)
                rheight = height(node.right)

                #Use the larger one
                if lheight > rheight :
                        return lheight+1
                else:
                        return rheight+1

# Driver program to test above function
```

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

print("Level order traversal of binary tree is -")
printLevelOrder(root)
```

```
# Python program to print level
# order traversal using Queue

# A node structure
class Node:
        # A utility function to create a new node
        def __init__(self ,key):
                self.data = key
                self.left = None
                self.right = None

def printLevelOrder(root):
        # Base Case
        if root is None:
                return
        queue = []
        queue.append(root)

        while(len(queue) > 0):
                print (queue[0].data)
                node = queue.pop(0)


                if node.left is not None:
                        queue.append(node.left)

                if node.right is not None:
                        queue.append(node.right)

#Driver Program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print ("Level Order Traversal of binary tree is -")
printLevelOrder(root)
```

Given a Binary Tree with all **unique** values and two nodes value, **n1** and **n2**. The task is to find the **lowest common ancestor** of the given two nodes. We may assume that either both n1 and n2 are present in the tree or none of them are present.

Example 1:

```
Input:
n1 = 2 , n2 = 3
       1
      / \
     2   3
Output: 1
Explanation:
LCA of 2 and 3 is 1.
```

Example 2:

```
Input:
n1 = 3 , n2 = 4
           5
          /
         2
        / \
       3   4
Output: 2
Explanation:
LCA of 3 and 4 is 2.
```

Method (By Storing root to n1 and root to n2 paths):

Following is a simple O(n) algorithm to find LCA of n1 and n2.

1) Find a path from the root to n1 and store it in a vector or array.

2) Find a path from the root to n2 and store it in another vector or array.

3) Traverse both paths till the values in arrays are the same. Return the common element just before the mismatch.

```
#Finds the path from root node to given root of the tree.
# Stores the path in a list path[], returns true if path exists otherwise false

def findPath( root, path, k):

    if root is None:
        return False

    path.append(root.key)

    if root.key == k :
        return True

    if ((root.left != None and findPath(root.left, path, k)) or
            (root.right!= None and findPath(root.right, path, k))):
        return True

    path.pop()
    return False
```

```
# Returns LCA if node n1 , n2 are present in the given
# binary tree otherwise return -1
def findLCA(root, n1, n2):

    # To store paths to n1 and n2 fromthe root
    path1 = []
    path2 = []

    # Find paths from root to n1 and root to n2.
    # If either n1 or n2 is not present , return -1
    if (not findPath(root, path1, n1) or not findPath(root, path2, n2)):
        return -1

    # Compare the paths to get the first different value
    i = 0
    while(i < len(path1) and i < len(path2)):
        if path1[i] != path2[i]:
            break
        i += 1

    return path1[i-1]
```