Check whether the two Binary Search Trees are Identical or Not

def isIdentical(root1, root2) :

    # Check if both the trees are empty

    if (root1 == None and root2 == None) :

        return 1

    # If any one of the tree is non-empty

    # and other is empty, return false

    elif (root1 != None and root2 == None) :

        return 0

    elif (root1 == None and root2 != None) :

        return 0

    else: # Check if current data of both trees

          # equal and recursively check for left

          # and right subtrees

        if (root1.data == root2.data and

            isIdentical(root1.left, root2.left)

            and isIdentical(root1.right, root2.right)) :

            return 1

        else:

            return 0

Another Code:-

# Iterative method to find height of

# Binary Tree

def areIdentical(root1, root2):

    # Return true if both trees are empty

    if (root1 and root2):

        return True

    # Return false if one is empty and

    # other is not

    if (root1 or root2):

        return False

    # Create an empty queues for

    # simultaneous traversals

    q1 = Queue()

    q2 = Queue()

    # Enqueue Roots of trees in

    # respective queues

    q1.put(root1)

    q2.put(root2)

    while (not q1.empty() and not q2.empty()):

        # Get front nodes and compare them

        n1 = q1.queue[0]

        n2 = q2.queue[0]

        if (n1.data != n2.data):

            return False

        # Remove front nodes from queues

        q1.get()

        q2.get()

        # Enqueue left children of both nodes

        if (n1.left and n2.left):

            q1.put(n1.left)

            q2.put(n2.left)

        # If one left child is empty and

        # other is not

        elif (n1.left or n2.left):

            return False

        # Right child code (Similar to

        # left child code)

        if (n1.right and n2.right):

            q1.put(n1.right)

            q2.put(n2.right)

        elif (n1.right or n2.right):

            return False

    return True

One important Observation in Recursion:-

def factorial(n):

print("factorial has been called with n = " + str(n))

if n == 1:

return 1

else:

res = n \* factorial(n-1)

print("intermediate result for ", n, " \* factorial(" ,n-1, "): ",res)

return res

print(factorial(5))

This Python script outputs the following results:

factorial has been called with n = 5

factorial has been called with n = 4

factorial has been called with n = 3

factorial has been called with n = 2

factorial has been called with n = 1

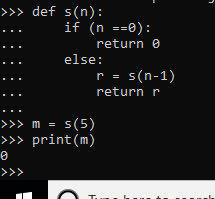
intermediate result for 2 \* factorial( 1 ): 2

intermediate result for 3 \* factorial( 2 ): 6

intermediate result for 4 \* factorial( 3 ): 24

intermediate result for 5 \* factorial( 4 ): 120

120



Calculate the height of the tree:-

""" 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 == None):

        return 0

    else:

        """ compute the height of each subtree """

        lheight = height(node.left)

        rheight = height(node.right)

        """ use the larger one """

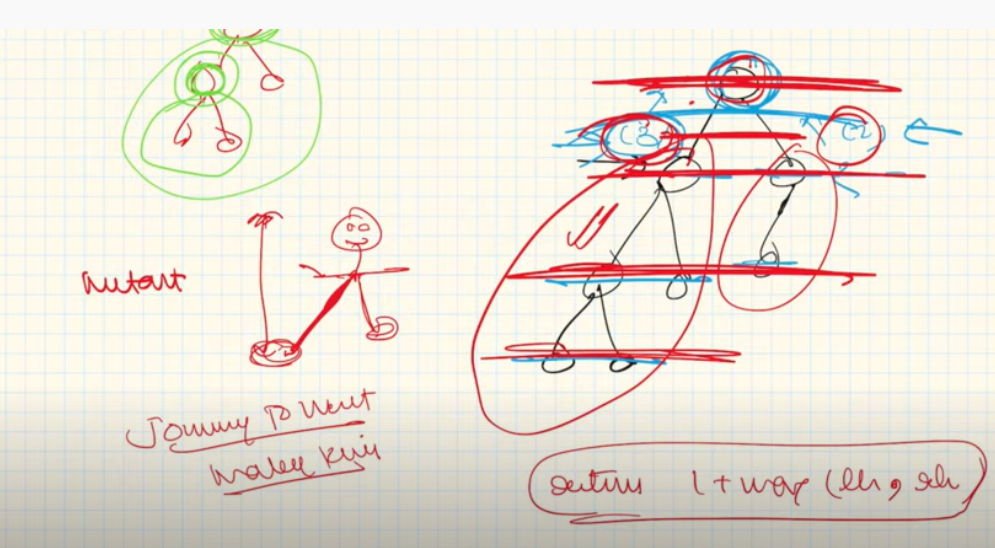
        if (lheight > rheight):

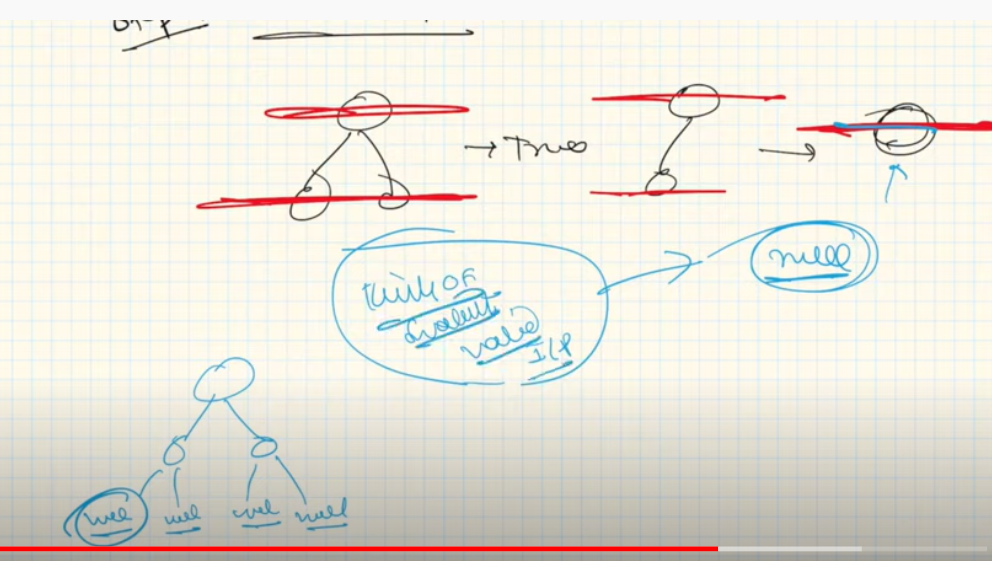
            return(lheight + 1)

        else:

            return(rheight + 1)

Note:- Here we are adding +1 because we need to consider root node as well. Lheight is the height of the left sub tree, So we need to add +1 to get the height of the left subtree and the root node.

ss



Here In the base condition, we are returning the 0 value, because where the leaf node has the height as 0. There is no height of leaf node. So we are returning the value here.

# Convert a Binary Tree into its Mirror Tree

# 

# Python3 program to convert a Binary

# Tree to its mirror

# A binary tree node has data, pointer to

# left child and a pointer to right child

# Helper function that allocates a new node

# with the given data and None left and

# right pointers

class newNode:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

''' Change a tree so that the roles of the left

    and right pointers are swapped at every node.

    So the tree...

        4

        / \

        2 5

        / \

    1 3

    is changed to...

        4

        / \

        5 2

            / \

        3 1

    '''

def mirror( root):

    if (root == None):

        return

    q = []

    q.append(root)

    # Do BFS. While doing BFS, keep swapping

    # left and right children

    while (len(q)):

        # pop top node from queue

        curr = q[0]

        q.pop(0)

        # swap left child with right child

        curr.left, curr.right = curr.right, curr.left

        # append left and right children

        if (curr.left):

            q.append(curr.left)

        if (curr.right):

            q.append(curr.right)

""" Helper function to print Inorder traversal."""

def inOrder( node):

    if (node == None):

        return

    inOrder(node.left)

    print(node.data, end = " ")

    inOrder(node.right)

# Driver code

root = newNode(1)

root.left = newNode(2)

root.right = newNode(3)

root.left.left = newNode(4)

root.left.right = newNode(5)

""" Print inorder traversal of the input tree """

print("Inorder traversal of the constructed tree is")

inOrder(root)

""" Convert tree to its mirror """

mirror(root)

""" Print inorder traversal of the mirror tree """

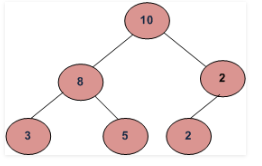
print("\nInorder traversal of the mirror tree is")

inOrder(root)

# Check for Children Sum Property in a Binary Tree

Given a binary tree, write a function that returns true if the tree satisfies below property.

For every node, data value must be equal to sum of data values in left and right children. Consider data value as 0 for NULL children. Below tree is an example



|  |
| --- |
| # Python3 program to check children  # sum property    # Helper class that allocates a new  # node with the given data and None  # left and right pointers.  class newNode:      def \_\_init\_\_(self, data):          self.data = data          self.left = None          self.right = None    # returns 1 if children sum property  # holds for the given node and both  # of its children  def isSumProperty(node):        # left\_data is left child data and      # right\_data is for right child data      left\_data = 0      right\_data = 0        # If node is None or it's a leaf      # node then return true      if(node == None or (node.left == None and                          node.right == None)):          return 1      else:            # If left child is not present then          # 0 is used as data of left child          if(node.left != None):              left\_data = node.left.data            # If right child is not present then          # 0 is used as data of right child          if(node.right != None):              right\_data = node.right.data            # if the node and both of its children          # satisfy the property return 1 else 0          if((node.data == left\_data + right\_data) and                          isSumProperty(node.left) and                          isSumProperty(node.right)):              return 1          else:              return 0 |

# Iterative approach to check for children sum property in a Binary Tree

# Python3 program to check

# children sum property

# A binary tree node

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

# Function to check if the tree holds

# children sum property

def CheckChildrenSum(root):

    q = []

    # Push the root node

    q.append(root)

    while len(q) != 0:

        temp = q.pop()

        # If the current node has both

        # left and right children

        if temp.left and temp.right:

            # If the current node is not equal

            # to the sum of its left and right

            # children, return false

            if (temp.data != temp.left.data +

                             temp.right.data):

                return False

            q.append(temp.left)

            q.append(temp.right)

        # If the current node has right child

        elif not temp.left and temp.right:

            # If the current node is not equal

            # to its right child return false

            if temp.data != temp.right.data:

                return False

            q.append(temp.right)

        # If the current node has left child

        elif not temp.right and temp.left:

            # If the current node is not equal

            # to its left child return false

            if temp.data != temp.left.data:

                return False

            q.append(temp.left)

    # If the given tree has children

    # sum property return true

    return True

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

    root = Node(10)

    root.left = Node(8)

    root.right = Node(2)

    root.left.left = Node(3)

    root.left.right = Node(5)

    root.right.right = Node(2)

    if CheckChildrenSum(root):

        print("Yes")

    else:

        print("No")

# This code is contributed

# by Rituraj Jain

# Program to count leaf nodes in a binary tree

A node is a leaf node if both left and right child nodes of it are NULL.

getLeafCount(node)

1) If node is NULL then return 0.

2) Else If left and right child nodes are NULL return 1.

3) Else recursively calculate leaf count of the tree using below formula.

Leaf count of a tree = Leaf count of left subtree +

Leaf count of right subtree

# Python program to count leaf nodes in Binary Tree

# A Binary tree node

class Node:

    # Constructor to create a new node

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = None

        self.right = None

# Function to get the count of leaf nodes in binary tree

def getLeafCount(node):

    if node is None:

        return 0

    if(node.left is None and node.right is None):

        return 1

    else:

        return getLeafCount(node.left) + getLeafCount(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 "Leaf count of the tree is %d" %(getLeafCount(root))

# Iterative program to count leaf nodes in a Binary Tree

# Python3 program to count leaf nodes

# in a Binary Tree

from queue import Queue

# Helper function that allocates a new

# Node with the given data and None

# left and right pointers.

class newNode:

    def \_\_init\_\_(self, data):

        self.data = data

        self.left = self.right = None

# Function to get the count of leaf

# Nodes in a binary tree

def getLeafCount(node):

    # If tree is empty

    if (not node):

        return 0

    # Initialize empty queue.

    q = Queue()

    # Do level order traversal

    # starting from root

    count = 0 # Initialize count of leaves

    q.put(node)

    while (not q.empty()):

        temp = q.queue[0]

        q.get()

        if (temp.left != None):

            q.put(temp.left)

        if (temp.right != None):

            q.put(temp.right)

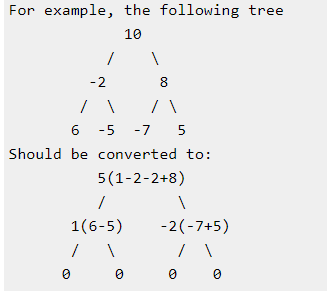
        if (temp.left == None and

            temp.right == None):

            count += 1

    return count

# Convert binary tree to its Sum tree



**Explanation with example:**

Let’s see how the function solves the above example.

Original tree:

10

/ \

-2 8

/ \ / \

6 -5 -7 5

In the main function it calls toSumTree (10)

------------------------------------------------------------

toSumTree (-2) //call at toSumTree (10)

node is not NULL

temp=-2

new node value = toSumTree (-2->left) + toSumTree (-2->right)

=toSumTree (6) + toSumTree (-5)

Thus call to toSumTree (6) and toSumTree (-5)

Return new node value + temp (-2)

------------------------------------------------------------

toSumTree (8) //call at toSumTree (10)

node is not NULL

temp=8

new node value = toSumTree (8->left) + toSumTree (8->right)

=toSumTree (-7) + toSumTree (5)

Thus call to toSumTree (-7) and toSumTree (5)

Return new node value + temp (8)

------------------------------------------------------------

toSumTree (6) //call at toSumTree (-2)

node is not NULL

temp=6

new node value = toSumTree (6->left) + toSumTree (6->right)

=toSumTree (NULL) + toSumTree (NULL)

Thus call to toSumTree (NULL) and toSumTree (NULL)

Return new node value + temp (6)

------------------------------------------------------------

toSumTree (-5) //call at toSumTree (-2)

node is not NULL

temp=-5

new node value = toSumTree (-5->left) + toSumTree (-5->right)

=toSumTree (NULL) + toSumTree (NULL)

Thus call to toSumTree (NULL) and toSumTree (NULL)

Return new node value + temp (-5)

------------------------------------------------------------

toSumTree (-7)//call at toSumTree(8)

node is not NULL

temp=-7

new node value = toSumTree (-7->left) + toSumTree (-7->right)

=toSumTree (NULL) + toSumTree (NULL)

Thus call to toSumTree (NULL) and toSumTree (NULL)

Return new node value + temp (-7)

------------------------------------------------------------

toSumTree (5)//call at toSumTree (8)

node is not NULL

temp=5

new node value = toSumTree (5->left) + toSumTree (5->right)

=toSumTree (NULL) + toSumTree (NULL)

Thus call to toSumTree (NULL) and toSumTree (NULL)

Return new node value + temp (5)

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree(6)

node is NULL

return 0

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree (6)

node is NULL

return 0

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree(-5)

node is NULL

return 0

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree(-5)

node is NULL

return 0

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree(-7)

node is NULL

return 0

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree(-7)

node is NULL

return 0

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree(5)

node is NULL

return 0

------------------------------------------------------------

toSumTree (NULL)//call at toSumTree(5)

node is NULL

return 0

At toSumTree (6) //call at toSumTree(-2)

new node value = toSumTree (6->left) + toSumTree (6->right)

=toSumTree (NULL) + toSumTree (NULL)

=0

6->0

It returns 0+6=6

------------------------------------------------------------

At toSumTree (-5) //call at toSumTree(-2)

new node value = toSumTree (-5->left) + toSumTree (-5->right)

=toSumTree (NULL) + toSumTree (NULL)

=0

-5->0

It returns 0-5=-5

------------------------------------------------------------

At toSumTree (-7) //call at toSumTree(8)

new node value = toSumTree (-7->left) + toSumTree (-7->right)

=toSumTree (NULL) + toSumTree (NULL)

=0

-7->0

It returns 0-7=-7

------------------------------------------------------------

At toSumTree (5) //call at toSumTree (8)

new node value = toSumTree (5->left) + toSumTree (5->right)

=toSumTree (NULL) + toSumTree (NULL)

=0

5->0

It returns 0+5=5

------------------------------------------------------------

At toSumTree (-2) //call at toSumTree(10)

new node value = toSumTree (-2->left) + toSumTree (-2->right)

=toSumTree (6) + toSumTree (-5)

=6 + (-5) =1

-2->1

It returns -2+1=-1

------------------------------------------------------------

At toSumTree (8) //call at toSumTree (10)

new node value = toSumTree (-8->left) + toSumTree (-8->right)

=toSumTree (-7) + toSumTree (5)

=-7 + (5) =-2

8->-2

It returns -2+8=6

------------------------------------------------------------

At toSumTree (10) //call at main

new node value = toSumTree (10->left) + toSumTree (10->right)

=toSumTree (-2) + toSumTree (8)

=-1 + 6=5

10->5

It returns 10+5=15

**So, the binary tree is sum converted to:**

5

/ \

1 -2

/ \ / \

0 0 0 0

class Node:

*# Constructor to create a new node*

def \_\_init\_\_(self, key):

self.node = key

self.left = None

self.right = None

class SumTree:

def toSumTree( self,root):

*# Base case*

if root is None:

return 0

*# Store the old value*

old\_val = root.node

*# Recursively call for left and right subtrees and store the sum as*

*# new value of this node*

root.node=self.toSumTree(root.left) + self.toSumTree(root.right)

*# Return the sum of values of nodes in left and right subtrees and*

*# old\_value of this node*

return root.node + old\_val