**Kth Smallest Element in a BST**

Given a binary search tree, write a function kthSmallest to find the **k**th smallest element in it.

**Example 1:**

**Input:** root = [3,1,4,null,2], k = 1

3

/ \

1 4

\

  2

**Output:** 1

**Example 2:**

**Input:** root = [5,3,6,2,4,null,null,1], k = 3

5

/ \

3 6

/ \

2 4

/

1

**Output:** 3

Best Solution:-

Apply Inorder tranversal for only for K elemenets.

class Solution(object):

def kthSmallest(self, root, k):

"""

:type root: TreeNode

:type k: int

:rtype: int

"""

self.k = k

def visit(node):

if node is None:

return None

left = visit(node.left)

if left is not None:

return left

if self.k == 1:

return node

self.k -= 1

return visit(node.right)

return visit(root).val

It is because it is Binary Search Tree which will keep the value less on the left and greater value on the right.

Easy Solution:-

class Solution:

def kthSmallest(self, root: TreeNode, k: int) -> int:

numList = []

def getelement(root):

if root:

getelement(root.left)

numList.append(root.val)

getelement(root.right)

getelement(root)

return numList[k-1]

**Binary Tree Right Side View**

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

**Example:**

**Input:** [1,2,3,null,5,null,4]

**Output:** [1, 3, 4]

**Explanation:**

1 <---level 0

/ \

2 3 <---Level 1

\ \

5 4 <---

Solution:-

Right View means the in a given level, the node number should be (Size of the level-1). Suppose, in the first level the size is 2 as there are two elements, 2 and 3. 2 is number as 0 and 3 is number as 1.

So size-1 = 1, should be the solution.

class Solution:

def rightSideView(self, root: TreeNode) -> List[int]:

if not root:

return

queu = [root]

result = []

while queu:

size = len(queu)

for i in range(size):

node = queu.pop(0)

if i == size - 1:

result.append(node.val)

if node.left: queu.append(node.left)

if node.right: queu.append(node.right)

return result

**Populating Next Right Pointers in Each Node**

ou are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

struct Node {

int val;

Node \*left;

Node \*right;

Node \*next;

}

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

**ollow up:**

* You may only use constant extra space.
* Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

**Example 1:**



**Input:** root = [1,2,3,4,5,6,7]

**Output:** [1,#,2,3,#,4,5,6,7,#]

Solution:-

Here we need to go for level order tranveeral as we need to go level by level and print the next soltion. In Recursive ,we don’t go level by level.

**A simple iterative solution with a small tweak. Right node is visited first before the left. There is also a notion of level. At each level switches, next is not applicable.**

class Solution:

def connect(self, root: 'Node') -> 'Node':

if not root:

return root

q = [root]

while q:

next\_level = []

for i, node in enumerate(q):

if i != len(q) - 1:

q[i].next = q[i+1]

if node.left:

next\_level.append(node.left)

if node.right:

next\_level.append(node.right)

q = next\_level

return root

import queue

class Node:

def \_\_init\_\_(self, val: int = 0, left = None, right = None, next = None):

self.val = val

self.left = left

self.right = right

self.next = next

def connect\_next(root: Node) -> Node:

q = queue.Queue()

if root is None:

return root

if root.left is None and root.right is None:

return root

prev\_level = -1

prev\_node = None

q.put((root, 0))

while not q.empty():

node\_tuple = q.get()

node = node\_tuple[0]

level = node\_tuple[1]

print("{}".format(node.val), end = ",")

if level != prev\_level:

node.next = None

prev\_level = level

prev\_node = node

else:

node.next = prev\_node

prev\_node = node

if node.right is not None:

q.put((node.right, level+1))

if node.left is not None:

q.put((node.left, level+1))

return root

class Solution:

def connect(self, root: Node) -> Node:

return connect\_next(root)

**Best Solution:-**

**Here we are inserting NULL at the end of each level. We will insert NULL value along with root node. Here we will go through the queue and check if we have got NULL value. If we get NULL value then we will assign this as next pointer. Also insert NULL to the right of the queue.**

**If we reached NULL means it will be the end of the level,**

Code:-

class Solution:

def connect(self, root: 'Node') -> 'Node':

def level(root):

queue = [root,None]

root.next = None

while queue:

ele = queue.pop(0)

if ele:

if ele.left:queue.append(ele.left)

if ele.right:queue.append(ele.right)

else:

if queue:

queue.append(None)

for i in range(1,len(queue)):

queue[i-1].next = queue[i]

if not root:return None

level(root)

return roo

**Binary Tree Zigzag Level Order Traversals**

Given a binary tree, return the *zigzag level order* traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:  
Given binary tree [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

return its zigzag level order traversal as:

[

[3],

[20,9],

[15,7]

]

Simple Solution:-

class Solution:

def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:

if not root:

return []

q = deque()

q.append(root)

zz = []

while len(q)>0:

len\_ = len(q)

z = []

for i in range(len\_):

node = q.popleft()

z.append(node.val)

if node.left:

q.append(node.left)

if node.right:

q.append(node.right)

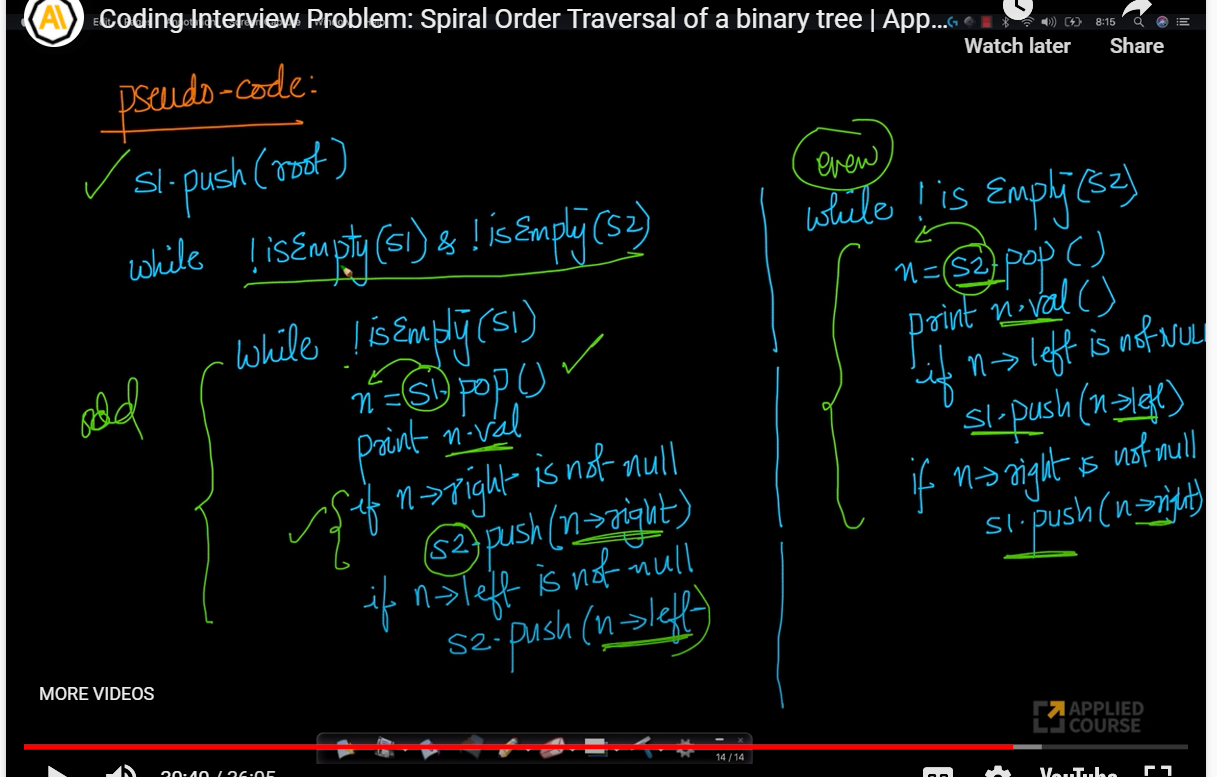
zz.append(z)

for i in range(len(zz)):

if i%2==1:

zz[i] = zz[i][::-1]

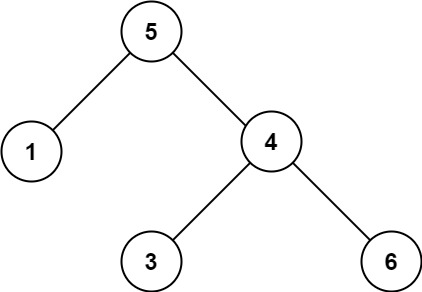
return zz



**Validate Binary Search Tree**

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.
* **Example 1:**
* 
* **Input:** root = [2,1,3]
* **Output:** true
* **Example 2:**
* 
* **Input:** root = [5,1,4,null,null,3,6]
* **Output:** false
* **Explanation:** The root node's value is 5 but its right child's value is 4.

Solution:-

We will go for a recursive solution. Here we are taking minimum value as 0 and maximum value as infinity. Every node should be less than the maximum value and more than minimum value. If it satisfies, we will ruturn True.

Def check(root, lower, upper):

If not root:

Return True

Val = root.val

If val <=lower or val >=upper:

Return False

If not check(root.left,lower,val):

Return False

If not check(root.right,val, upper):

Return False

Return True

Lower = float(‘-inf’)

Upper = float(‘+inf’)

Return check(root,lower,upper)

**Invert Binary Tree**

Invert a binary tree.

**Example:**

Input:

4

/ \

2 7

/ \ / \

1 3 6 9

Output:

4

/ \

7 2

/ \ / \

9 6 3 1

class Solution:  
def invertTree(self, root: TreeNode) -> TreeNode:  
if root is not None:  
temp = root.left  
root.left = root.right  
root.right = temp  
self.invertTree(root.left)  
self.invertTree(root.right)  
return root

def invertTree(self, root: TreeNode) -> TreeNode:

if not root:

return

left = self.invertTree(root.left)

right = self.invertTree(root.right)

root.left, root.right = right, left

return root

# iterative - bfs approach

def iterative\_invert(self, root):

if root:

q = Queue()

q.put(root)

while q.qsize() > 0:

node = q.get()

node.left, node.right = node.right, node.left

if node.left:

q.put(node.left)

if node.right:

q.put(node.right)

return root

**Merge Two Binary Trees**

Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

**Example 1:**

**Input:**

Tree 1 Tree 2

1 2

/ \ / \

3 2 1 3

/ \ \

5 4 7

**Output:**

Merged tree:

3

/ \

4 5

/ \ \

5 4 7

lass Solution:

def mergeTrees(self, t1: TreeNode, t2: TreeNode) -> TreeNode:

def merge(t1, t2):

if not t1 and not t2:

return None

elif t1 and not t2:

node = t1

elif t2 and not t1:

node = t2

else:

node = TreeNode(t1.val + t2.val)

node.left = merge(t1.left, t2.left)

node.right = merge(t1.right, t2.right)

return node

return merge(t1, t2)