**TREE QUESTIONS**

Table of Contents

[**Theory** 2](#_Toc157576486)

[What is Tree data structure? 2](#_Toc157576487)

[What are components of tree? 2](#_Toc157576488)

[Binary Tree 3](#_Toc157576489)

[Tree traversal 3](#_Toc157576490)

[Binary Search Tree 4](#_Toc157576491)

[Use cases of Tree and its use in system design 4](#_Toc157576492)

[Array representation of Binary Tree 4](#_Toc157576493)

[**Important Codes** 6](#_Toc157576494)

[Binary Search Tree Implementation: 6](#_Toc157576495)

[Traversals: 7](#_Toc157576496)

[Breadth First : 8](#_Toc157576497)

[Morris Traversal: 9](#_Toc157576498)

[LEVEL 1: **EASY** 12](#_Toc157576499)

[1. Check if two binary trees are same or not. 12](#_Toc157576500)

[3. Merge two binary trees. 12](#_Toc157576501)

[4. Convert a BST to tree with only right nodes. 12](#_Toc157576502)

[5. Average of levels in binary tree. 12](#_Toc157576503)

[6. Sorted array to BST 12](#_Toc157576504)

[7. Binary Tree tilt 12](#_Toc157576505)

[8. Diameter of binary tree 12](#_Toc157576506)

[9. Subtree of another tree 12](#_Toc157576507)

[LEVEL 2: **Medium** 13](#_Toc157576508)

[LEVEL 3: **Difficult** 14](#_Toc157576509)

[**SOLUTIONS:** 15](#_Toc157576510)

[**LEVEL 1:** 15](#_Toc157576511)

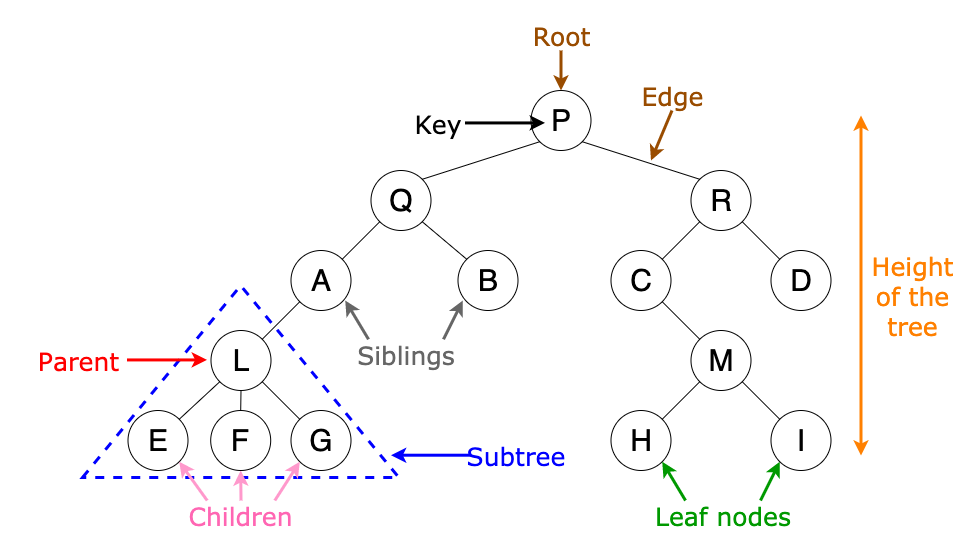
# **Theory**

## What is Tree data structure?

A tree is a hierarchical data structure that starts with a root node and consists of connected nodes. It organizes elements in a parent-child relationship, where each node can have multiple children. It is widely used for representing hierarchical relationships and enables efficient searching, insertion, and deletion operations.

## What are components of tree?

1. **Root:** A tree has a unique node called the root, which serves as the starting point of the tree. All other nodes in the tree are descendants of the root.
2. **Nodes:** A tree consists of nodes that are connected by edges. Each node can have zero or more child nodes. Nodes other than the root are referred to as internal nodes, while nodes without any children are called leaf nodes or leaves.
3. **Edges:** Edges are the connections between nodes in a tree. They represent the relationships between the nodes. Each node (except the root) has exactly one incoming edge from its parent node, and it may have zero or more outgoing edges to its child nodes.
4. **Parent and Child Nodes:** Each node in a tree (except the root) has a unique parent node, which is the node directly above it. The parent node is connected to its child nodes through edges. Conversely, child nodes are the nodes directly below a parent node.
5. **Ancestors and Descendants:** An ancestor of a node is any node that exists on the path from the root to that node, excluding the node itself. A descendant of a node is any node that can be reached by following edges from the node, again excluding the node itself.
6. **Path:** A path in a tree is a sequence of nodes connected by edges. It represents the traversal from one node to another within the tree.
7. **Depth and Height:** The depth of a node is the length of the path from the root to that node. The height of a tree is the maximum depth of any node in the tree. The height represents the length of the longest path from the root to a leaf node.
8. **Subtree:** A subtree is a portion of the tree that consists of a node and all its descendants, including their child nodes and their child nodes, and so on.
9. **Degree:** The degree of a node is the number of its child nodes. A node with a degree of zero is a leaf node, while a node with a degree greater than zero is an internal node.



## Binary Tree

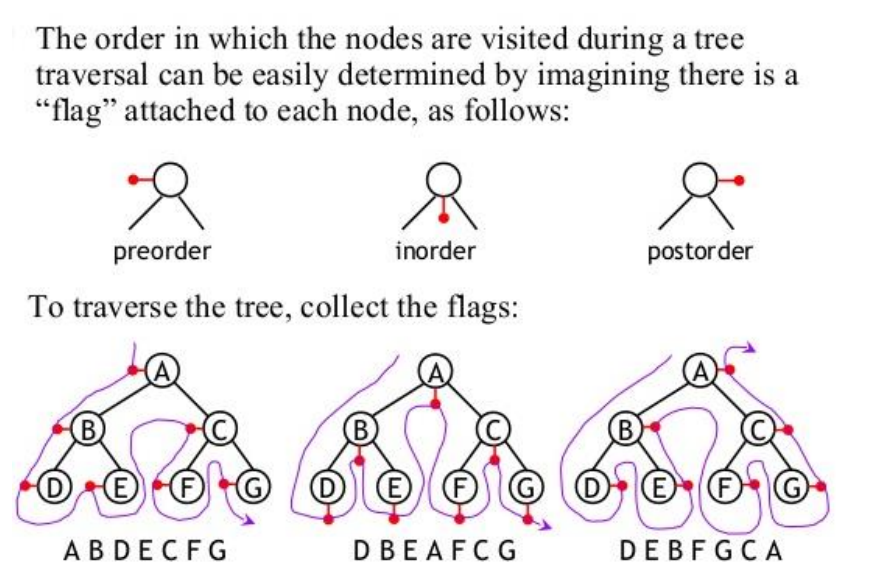
A binary tree is a tree data structure where each node can have at most two children, the left child and the right child.

Some types of binary trees are:-

1. **Full Binary Tree:** A full binary tree is a binary tree in which every node has either 0 or 2 children. In other words, every level of the tree is completely filled, except for the last level, which may or may not be full.
2. **Complete Binary Tree:** A complete binary tree is a binary tree in which all levels are completely filled, except possibly for the last level, which is filled from left to right. It ensures that the nodes are as balanced as possible.
3. **Perfect Binary Tree:** A perfect binary tree is a binary tree in which all levels are completely filled with maximum possible nodes. It is a type of complete binary tree with exactly 2^h - 1 nodes, where h is the height of the tree.
4. **Balanced Binary Tree:** A balanced binary tree is a binary tree in which the difference in height between the left and right subtrees of any node is at most 1. It ensures that the height of the tree remains relatively small, resulting in efficient operations.
5. **Degenerate (or Pathological) Binary Tree:** A degenerate binary tree is a binary tree in which each parent node has only one child (either left or right). It essentially becomes a linked list.

## Tree traversal

1. Inorder traversal ( left, root, right)
2. Preorder traversal ( root, left , right)
3. Postorder traversal ( root, left, right)



## Binary Search Tree

A binary search tree is a type of binary tree that follows a specific ordering of the nodes. In a binary search tree, the left child of a node contains values that are less than the parent node, and the right child contains values that are greater than the parent node. This ordering property allows for efficient searching, insertion, and deletion operations.

For BST inorder traversal gives numbers in ascending order.

## Use cases of Tree and its use in system design

1. **File System:** Trees are widely used to represent file systems in operating systems. Each directory is represented as a node in the tree, with subdirectories and files as its children.
2. **Routing Algorithms:** Trees are employed in routing algorithms used in networking and telecommunications systems. Examples include the spanning tree protocol (STP) for network redundancy and the hierarchical routing protocol used in the Internet's Border Gateway Protocol (BGP) to organize routing tables.
3. **Data Compression:** Huffman coding is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
4. **Compiler Design:** In compiler design, a syntax tree is used to represent the structure of a program.
5. **Database Indexing:** B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.
6. **User Interfaces:** Trees are utilized in the design of graphical user interfaces (GUIs) to represent the hierarchy of UI elements. This allows for efficient event handling, layout management, and navigation within the UI.

## Array representation of Binary Tree

* If parent is in k th position:

1. Left child – 2K position
2. Right child – 2k+1 position

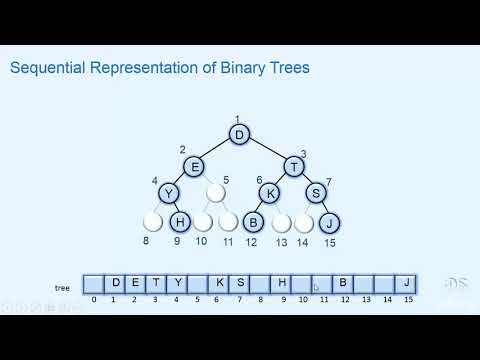
* If child is in k th position:

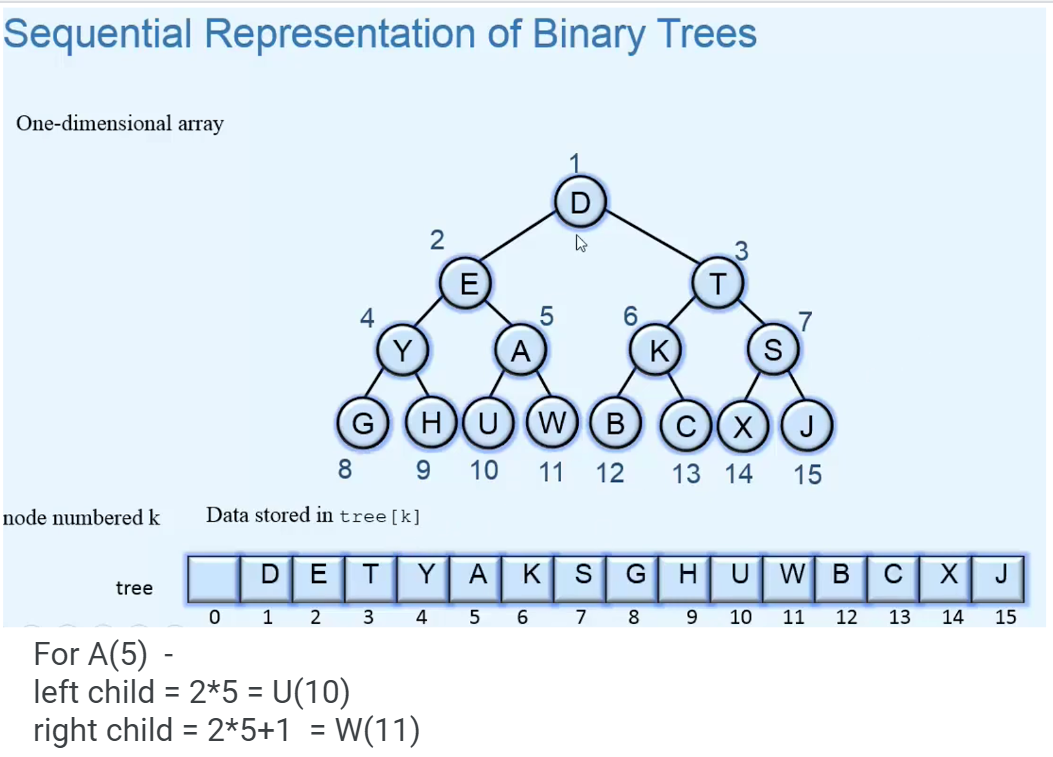
Parent position = floor(k/2)

* Size of array needed to represent a binary tree of height h

= Maximum number of nodes possible in binary tree = 2h-1

* For n nodes:- complete binary tree will have minimum height , hence will require least size of array to represent
* Null nodes are also considered in array representation.

[](https://www.youtube.com/watch?v=0iHD18D072c)



# **Important Codes**

## Binary Search Tree Implementation:

class Node:

    def \_\_init\_\_(self,data):

*self*.data = data

*self*.left = None

*self*.right = None

class BST:

    def \_\_init\_\_(self):

*self*.root = None

    def insert(self,data):

        def insert\_recursive(node,data):

            if node is None:

                return Node(data)

            if data < node.data:

                node.left = insert\_recursive(node.left,data)

            else:

                node.right = insert\_recursive(node.right,data)

            return node

*self*.root = insert\_recursive(*self*.root, data)

    def inorder(self):

        def inorder\_traversal(node):

            if node is not None:

                inorder\_traversal(node.left)

                print(node.data,end=" ")

                inorder\_traversal(node.right)

        inorder\_traversal(*self*.root)

    def search(self,data):

        def search\_recursive(node,data):

            if node is None:

                return False

            if node.data == data:

                return True

            if data < node.data:

                return search\_recursive(node.left,data)

            else:

                return search\_recursive(node.right,data)

        return search\_recursive(*self*.root,data)

bst = BST()

bst.insert(3)

bst.insert(5)

bst.insert(0)

bst.insert(9)

bst.inorder()

print(bst.search(9))

## Traversals:

*# While using global array or variable, always remember to clear it first before use. Else if we run code for more than one output in global array answers get appended for all test cases.*

*# We can also write this without the use of global varialbe, as explained in method 2 and 3.*

*#Method 1 ( use of global variable)*

class Solution:

    res=[]

    def inOrder(self,root):

        if root is None:

            return

*self*.inOrder(root.left)

*self*.res.append(root.val)

*self*.inOrder(root.right)

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

*self*.res.clear()     *#clear the data*

*self*.inOrder(root)

        return *self*.res

*#Method 2 (without global variable)*

class Solution:

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

        res = []

        if root is None:

            return res

        res+=*self*.inorderTraversal(root.left)

        res+=[root.val]

        res+=*self*.inorderTraversal(root.right)

        return res

*# Method 3 : One liner*

class Solution:

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

        return  *self*.inorderTraversal(root.left) + [root.val] + *self*.inorderTraversal(root.right) if root else []

*#For Preorder*

class Solution:

    def preorder\_traversal(self, root: Optional[TreeNode]) -> List[int]:

        return [root.val]+*self*.preorder\_traversal(root.left) + *self*.preorder\_traversal(root.right) if root else []

## https://assets.leetcode.com/uploads/2021/02/19/tree1.jpgBreadth First :

Problem: so if input is given tree root = [3,9,20,null,null,15,7]

[](https://www.youtube.com/watch?v=6ZnyEApgFYg)Output should be = [[3], [9,20], [15,7]]

import collections

class Node:

    def \_\_init\_\_(self, key):

*self*.val = key

*self*.left = None

*self*.right = None

def printLevelOrder(root):

    res=[]

    q=collections.deque()

    q.append(root)

    while(q):

        q\_len=len(q)

        level=[]

        for i in range(q\_len):

            node=q.popleft()

            if node:

                level.append(node.val)

                q.append(node.left)

                q.append(node.right)

        if level:

            res.append(level)

    return res

*# Driver Program to test above function*

if \_\_name\_\_ == '\_\_main\_\_':

    root = Node(3)

    root.left = Node(9)

    root.right = Node(20)

    root.right.left = Node(15)

    root.right.right = Node(7)

    print("Level Order Traversal of binary tree is -")

    print(printLevelOrder(root))

## Morris Traversal:

Morris Traversal is a tree traversal algorithm that allows for the traversal of a binary tree without using recursion or an explicit stack

Recursive solution: O(n) time and O(n) space (function call stack);

**Morris traversal**: O(n) time and **O(1)** space.

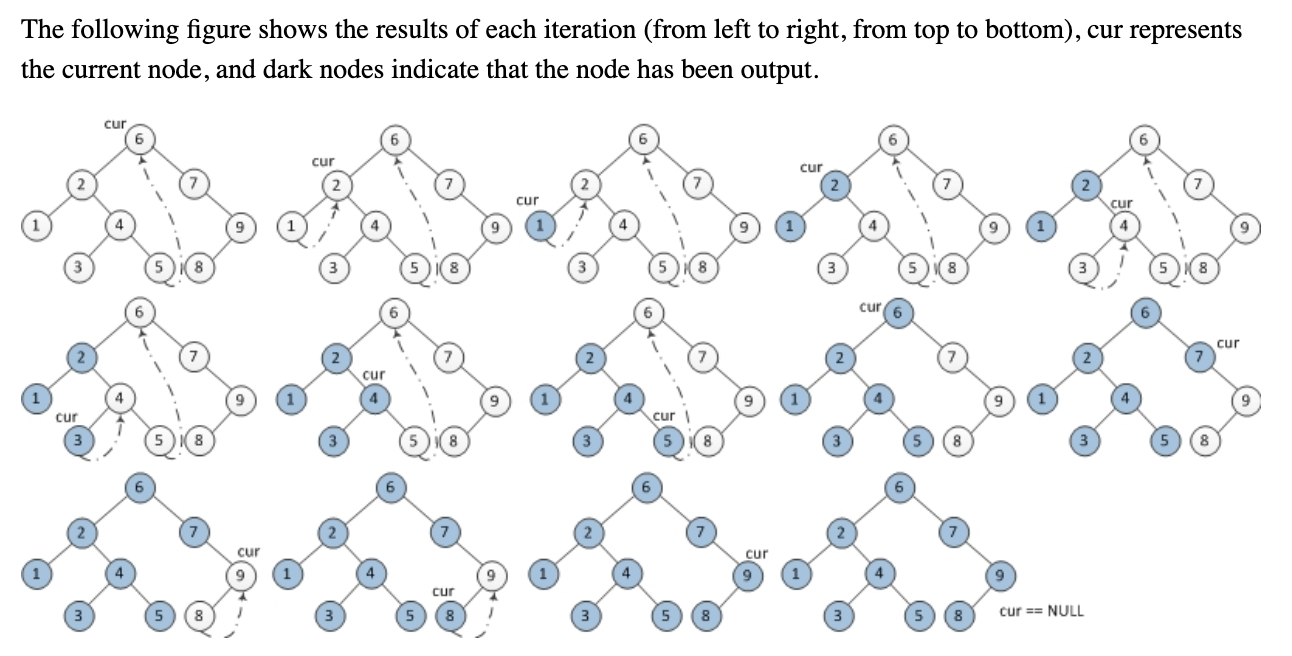
In-order Morris Traversal:

🌳 1st case: if left is null, print current node and go right

🌳 2nd case: before going left, make right most node on left subtree connected to current node, then go left 🌳 3rd case: if thread is already pointed to current node, then remove the thread

Iterate until the current node is not NULL.

This is Morris traversal for inorder, can have similar for pre and post order too.



Lecture:-

[](https://www.youtube.com/watch?v=80Zug6D1_r4)

Code

class TreeNode:

    def \_\_init\_\_(self, value):

*self*.value = value

*self*.left = None

*self*.right = None

def morris\_inorder\_traversal\_bst(root):

    result = []

    cur = root

    while cur:

        if not cur.left:

            result.append(cur.value)

            cur = cur.right

        else:

*# Find the inorder predecessor(right most node of left subtree)*

            prev = cur.left

            while prev.right and prev.right != cur:

                prev = prev.right

*# Make current as the right child of its inorder predecessor*

            if not prev.right:

                prev.right = cur

                cur = cur.left

            else:

*#if there is already linkage*

*# Revert the changes made in the 'if' part to restore the original tree*

                prev.right = None

                result.append(cur.value)

                cur = cur.right

    return result

*# Example usage:*

*# Construct a sample Binary Search          4*

*#                                          / \*

*#                                         2   5*

*#                                       /  \*

*#                                      1    3*

root\_bst = TreeNode(4)

root\_bst.left = TreeNode(2)

root\_bst.left.left = TreeNode(1)

root\_bst.left.right = TreeNode(3)

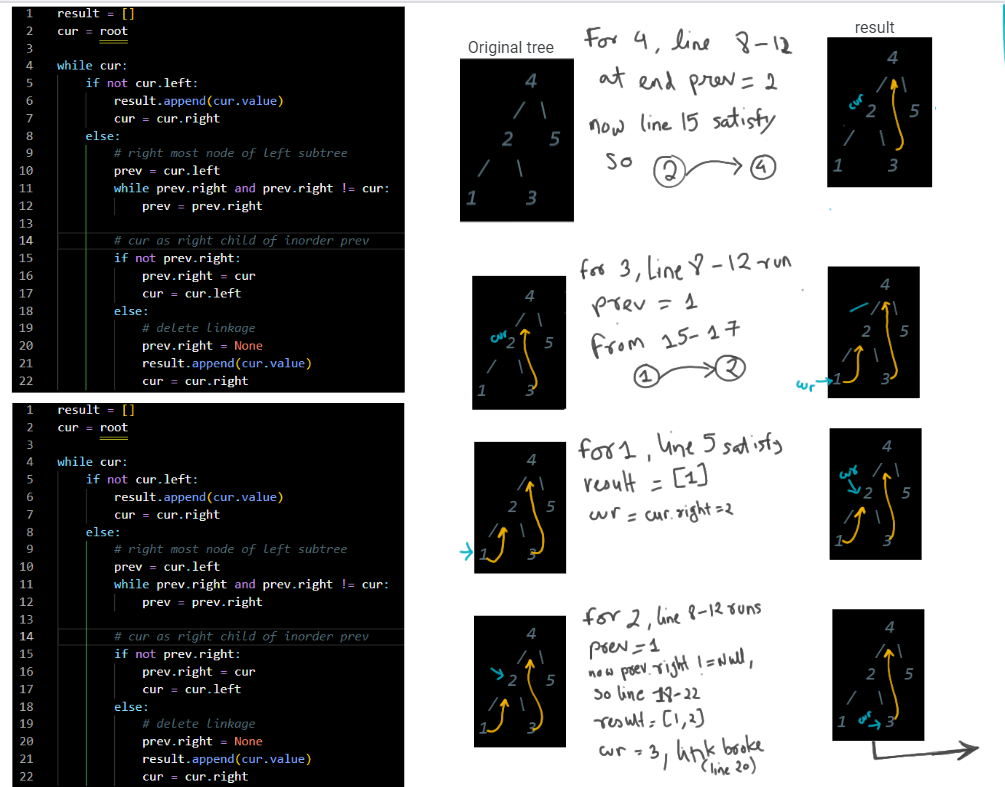
root\_bst.right = TreeNode(5)

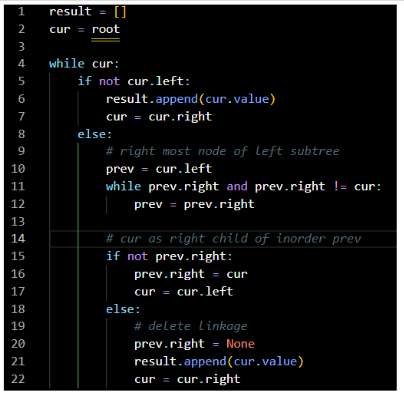
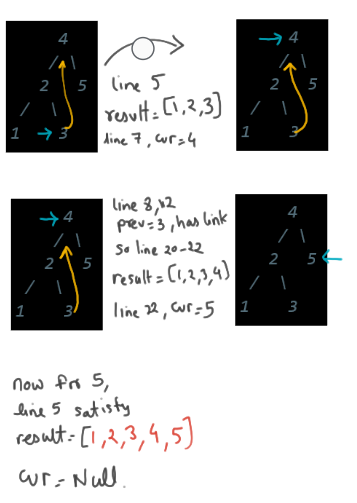
result\_bst = morris\_inorder\_traversal\_bst(root\_bst)

print(result\_bst)

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

Execution:



# LEVEL 1: **EASY**

### Check if two binary trees are same or not.

Link: <https://leetcode.com/problems/same-tree/description/>

1. Given the root node of a binary search tree and two integers low and high, return the sum of values of all nodes with a value in the inclusive range [low, high].

Link: <https://leetcode.com/problems/range-sum-of-bst/description/>

### Merge two binary trees.

Link: <https://leetcode.com/problems/merge-two-binary-trees/description/>

### Convert a BST to tree with only right nodes.

Link: <https://leetcode.com/problems/increasing-order-search-tree/description/>

### Average of levels in binary tree.

Link: <https://leetcode.com/problems/average-of-levels-in-binary-tree/description/>

### Sorted array to BST

Link: <https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/description/>

### Binary Tree tilt

Link: <https://leetcode.com/problems/binary-tree-tilt/description/>

### Diameter of binary tree

Link: <https://leetcode.com/problems/binary-tree-tilt/description/>

### Subtree of another tree

Link: <https://leetcode.com/problems/subtree-of-another-tree/description/>

# LEVEL 2: **Medium**

# LEVEL 3: **Difficult**

# **SOLUTIONS:**

## **LEVEL 1:**

1. Same tree

class Solution:

    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:

        self.ans=True

        def helper(p,q):

            if not p and not q:

                return

            elif(p is None) or (q is None):

*self*.ans=False

                return

            else:

                helper(p.left,q.left)

                if(p.val!=q.val):

*# print("vd")*

*self*.ans=False

                helper(p.right,q.right)

        helper(p,q)

        return *self*.ans

1. Range sum of bst

[Solution](https://leetcode.com/problems/range-sum-of-bst/solutions/4558641/python-fastest-optimized-with-explanation/)

class Solution:

    def rangeSumBST(self, root: Optional[TreeNode], low: int, high: int) -> int:

        def travel(root):

            global ans

            if (root is None) :

                return

            else:

                if root.val>=low and root.val<=high:

                    ans+=root.val

                if root.val >low:

                    travel(root.left)

                if root.val <high:

                    travel(root.right)

        global ans

        ans=0

        travel(root)

        return ans

1. Pair sum divisible by 5

class Solution:

    def mergeTrees(self, root1: TreeNode, root2: TreeNode) -> TreeNode:

        if root1 is None:

            return root2

        if root2 is None:

            return  root1

        root1.val = root1.val + root2.val

        root1.left = *self*.mergeTrees(root1.left,root2.left)

        root1.right = *self*.mergeTrees(root1.right,root2.right)

        return root1

1. Convert BST to inorder tree with only right node

[Solution](https://leetcode.com/problems/increasing-order-search-tree/solutions/4558733/best-python-solutions-2-approach-optimization/)

class Solution:

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

        def convert\_to\_inorder(root):

            global node

            if root is None:

                return

            else:

                convert\_to\_inorder(root.left)

                x=node

                while(x.right):

                    x=x.right

                x.right=TreeNode(root.val)

                convert\_to\_inorder(root.right)

        global node

        node=TreeNode()

        convert\_to\_inorder(root)

        return node.right

*#Approach 2*

class Solution:

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

        dummy = curr =TreeNode(None)

        def dfs(root):

            if not root: return

            nonlocal curr

            dfs(root.left)

            curr.right = root

            curr = root

            curr.left = None

            dfs(root.right)

        dfs(root)

        return dummy.right

1. Average of levels in binary tree

[Solution](https://leetcode.com/problems/average-of-levels-in-binary-tree/solutions/4563843/python-code-explanation-general-approach/)

import collections

class Solution:

    def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:

        res=[]

        q = collections.deque()

        q.append(root)

        while(q):

            level=[]

            qlen =len(q)

            for i in range(qlen):

                node = q.popleft()

                if node:

                    level.append(node.val)

                    q.append(node.left)

                    q.append(node.right)

            if level:

                res.append(sum(level)/len(level))

        return res

1. Sorted Array to binary Search tree

[Solution](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/solutions/4565640/python-code-fastest-solution-explanation/)

[](https://www.youtube.com/watch?v=0K0uCMYq5ng)

class Solution:

    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:

        def helper(l,r):

            if l>r:

                return

            else:

                m = (l+r)//2

                root = TreeNode(nums[m])

                root.left = helper(l,m-1)

                root.right = helper(m+1,r)

                return root

        return helper(0,len(nums)-1)

1. Binary tree tilt

[Solution](https://leetcode.com/problems/binary-tree-tilt/solutions/4578246/python-2-solutions-easy-one-optimized-version-with-explanation/)

class Solution:

    def findTilt(self, root: TreeNode) -> int:

        def helper(root):

            if not root: return 0

            lv, rv = helper(root.left), helper(root.right)

*self*.ans += abs(lv - rv)

            return root.val + lv + rv

*self*.ans = 0

        helper(root)

        return *self*.ans

1. Diameter Of Binary tree

[Solution](https://leetcode.com/problems/diameter-of-binary-tree/solutions/1143907/python-thought-process/)

[](https://www.youtube.com/watch?v=bkxqA8Rfv04)

def diameterOfBinaryTree(self, root):

    def recurse(node):

        if not node: return 0

        left, right = recurse(node.left), recurse(node.right)

*self*.result = max(*self*.result, left+right)

        return 1 + max(left, right)

*self*.result = 0

    recurse(root)

    return *self*.result

Similar way try to code for finding height of binary tree.

1. Subtree of another tree

class Solution:

    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:

        if subRoot == None :

            return True

        if root == None :

            return False

        if *self*.same(root , subRoot):

            return True

        return *self*.isSubtree(root.left , subRoot) or *self*.isSubtree(root.right , subRoot)

*#This is code to check if one tree is equal to other*

*#check root vals and left and right subtree vals, can't do r==s(this don't work)*

    def same(self , r , s):

        if r == None and s == None :

            return True

        if r and s and r.val == s.val:

            return *self*.same(r.right , s.right) and *self*.same(r.left , s.left)

        return False