-> **Binary tree traversals iterative:**

1. preorder:

```python
class Solution:
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return

        stk = [root]
        res = []

        while stk:
            top = stk.pop()
            res.append(top.val)

            if top.right:
                stk.append(top.right)

            if top.left:
                stk.append(top.left)

        return res
```

2. inorder:

```python
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []

        stk = [root]
        l=1

        res = []

        while stk:
            top = stk[l-1]
            stk.pop()
            l-=1

            f1,f2 = 0,0
            if top.right:
                stk.append(top.right)
                l+=1
                top.right = None
                f1 = 1

            stk.append(top)
            l+=1

            if top.left:
                stk.append(top.left)
                l+=1
                top.left = None
                f2 = 1

            if not f1 and not f2:
                top = stk.pop()
                l-=1
                res.append(top.val)

        return res
```

more intiuitive:

```python
class Solution:
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        node = root
        stk = []
        res = []

        while True:
            if node:
                stk.append(node)
                node = node.left
            else:
                if not stk:
                    break
                top = stk.pop()
                res.append(top.val)

                node = top.right
        return res
```

3. postorder:

```python
class Solution:
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        stk = [root]
        res = []
        l = 1

        while stk:
            top = stk[l-1]
            f = 0
            if top.right:
                stk.append(top.right)
                l+=1
                f = 1
                top.right = None

            if top.left:
                stk.append(top.left)
                l+=1
                f = 1
                top.left = None

            if not f:
                top = stk.pop()
                l-=1

                res.append(top.val)

        return res
```

-> max depth of binary tree:

1. using dfs:
-> find left and right subtree's height and max of both will be returned.

```python
class Solution:
    def dfs(self, root):
        if not root:
            return 0

        left = 1 + self.dfs(root.left)
        right = 1 + self.dfs(root.right)

        return max(left, right)

    def maxDepth(self, root: Optional[TreeNode]) -> int:
        return self.dfs(root)
```

2. using bfs:
-> do level order traversal, the number of levels in binary tree is the max depth/height of binary tree.

```
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root:
            return 0
        q = [root]
        d = 0

        while q:
            l = len(q)
            d+=1
            while l:
                front = q.pop(0)

                if front.left:
                    q.append(front.left)

                if front.right:
                    q.append(front.right)
                l-=1

        return d
```

-> check if given binary tree is balanced binary tree or not.
-> a binary tree is balanced if abs(height(left_subtree) – height(right_subtree)) <= 1.
-> find height of left and right subtree respectively, if above condition doesn't satisfy return height as -1 (-1 represents that given tree is not balanced binary tree). If any of left_height or right_height is -1, return -1, otherwise return maximum of left or right subtree's height.

```
class Solution:
    def dfs(self, root):
        if not root:
            return 0

        left = self.dfs(root.left)
        right = self.dfs(root.right)

        if left == -1 or right ==-1:
            return -1

        return 1+max(left, right) if abs(left -right) <=1 else -1

    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        h= self.dfs(root)
        if h==-1:
            return False
        return True
```

-> diameter of binary tree:
-> find left and right height of root node, sum them to get diameter of tree.

```python
class Solution:
    def dfs(self, root):
        if not root:
            return 0

        left = self.dfs(root.left)
        right = self.dfs(root.right)

        self.maxi = max(self.maxi, left+right)

        return 1+ max(left, right)

    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        self.maxi = 0
        self.dfs(root)
        return self.maxi
```

-> maximum path sum in a binary tree: (imp)
-> think in same way as finding diameter of binary tree using max heights of left and right subtree.
-> for every node find left and right subtree height(take 0 if heights are negative), check whether this height is maximum.

```python
class Solution:
    def dfs(self, root):
        if not root:
            return 0

        left = max(0,self.dfs(root.left))
        right = max(0,self.dfs(root.right))

        self.maxi = max(self.maxi, root.val+left+right)

        return root.val + max(left, right)


    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        self.maxi = float('-inf')
        self.dfs(root)
        return self.maxi
```

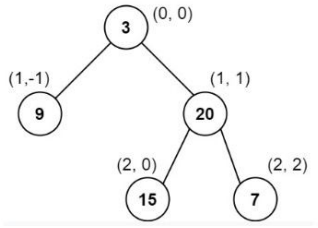
-> zig-zag (reverse level order traversal) of binary tree:
-> do level order traversal, at even level print the level in normal way otherwise in reverse order.

```python
class Solution:
    def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if not root:
            return []
        q = [root]
        res = []
        c = 1

        while q:
            l = len(q)
            temp = []
            while l:
                front = q.pop(0)
                l-=1
                temp.append(front.val)

                if front.left:
                    q.append(front.left)

                if front.right:
                    q.append(front.right)


            if c:
                res.append(temp[:])
                c=0
            else:
                res.append(temp[::-1])
                c=1

        return res
```

-> boundary traversal of binary tree:
-> take root node in traversal list.
-> for left boundary, travel only left, if not possible then travel right, but do not travel leaf node or root node as root node is already taken intitally.
-> for leaf, use dfs and only pick leaf nodes and avoid taking root.
-> for right boundary, travel right, if not possible travel left but avoid root and leaf nodes.

```
def printBoundaryView(self, root):
    # Code here
    if not root:
        return []

    self.res = [root.data]

    def left_boundary(node):
        if not node or (not node.left and not node.right):
            return

        self.res.append(node.data)
        if node.left:
            left_boundary(node.left)
        elif node.right:
            left_boundary(node.right)

    left_boundary(root.left)


    def dfs(node):
        if not node:
            return

        if not node.left and not node.right:
            self.res.append(node.data)
            return

        dfs(node.left)
        dfs(node.right)

    dfs(root.left)
    dfs(root.right)


    def right_boundary(node):
        if not node or (not node.left and not node.right):
            return

        if node.right:
            right_boundary(node.right)
        elif node.left:
            right_boundary(node.left)
        self.res.append(node.data)

    right_boundary(root.right)

    return self.res
```

-> vertical order traversal of binary tree:



-> think of traversing binary tree as column wise.

Like -> 9 at -1 level,

3, 15 at 0 level

20 at 1 level

7 at 2 level

-> maintain a dict, which has ‘y’ cordinate as key and (‘x’ cordinate, [nodes]) as value.

-> do level order or dfs, when moving left (x+1, y-1), when moving right (x+1, y+1), as per these values, fill the dict.

-> also, maintain minimum and maximum of y cordinate (helpful in travelling from left to right).

-> now, travel from left to right, and sort according to ‘x’.

```
class Solution:
    def dfs(self, root, x, y):
        if not root:
            return

        if self.vertical_level.get(y):
            self.vertical_level[y].append((x,root.val))
        else:
            self.vertical_level[y] = [(x,root.val)]

        self.left_most = min(self.left_most, y)
        self.right_most = max(self.right_most, y)

        self.dfs(root.left, x+1, y-1)
        self.dfs(root.right, x+1, y+1)


    def verticalTraversal(self, root: Optional[TreeNode]) -> List[List[int]]:
        self.vertical_level = {}
        self.left_most = 0
        self.right_most = 0

        self.dfs(root, 0, 0)

        res = []
        while self.left_most <= self.right_most:
            res.append([i[1] for i in sorted(self.vertical_level[self.left_most])])
            self.left_most+=1

        return res
```

-> top view of binary tree:
-> similar approach as used vertical order traversal.
-> can maintain a mapping specifying, a dict, which has ‘x’ cordinate as key and (‘y’ cordinate, [nodes]) as value.
-> do bfs traversal and fill the mapping, if any node which has already entry in mapping, do not append it again as only the first node will be used as top view, other nodes at same x cordinate will be in shadow.

```
class Solution:

    #Function to return a list of nodes visible from the top view
    #from left to right in Binary Tree.
    def topView(self,root):

        # code here
        x, y = 0, 0
        l, r = 0, 0
        self.mapping = {}
        q = [(x,y,root)]

        while q:
            x,y,front = q.pop(0)

            if not self.mapping.get(x):
                self.mapping[x] = (y, front.data)

            l, r = min(l, x), max(r, x)

            if front.left:
                q.append((x-1, y+1, front.left))

            if front.right:
                q.append((x+1, y+1, front.right))

        res = []
        while l<=r:
            res.append(self.mapping[l][1])
            l+=1

        return res
```

-> bottom view of binary tree:
-> same of top view, with only difference that everytime a node with x cordinate already present in mapping, it gets overwrite with latest node as we have to take bottomost nodes.

```
class Solution:
    def bottomView(self, root):
        # code here
        l, r = 0, 0
        mapping = {}
        x, y = 0, 0
        q = [(x, y, root)]

        while q:
            x, y, front = q.pop(0)

            mapping[x] = (y, front.data)
            l, r = min(l, x), max(r, x)

            if front.left:
                q.append((x-1, y+1, front.left))

            if front.right:
                q.append((x+1, y+1, front.right))

        res = []
        while l<=r:
            res.append(mapping[l][1])
            l+=1

        return res
```

-> right view of binary tree:

-> using level order traversal, take last node in every level order traversal.

```python
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        res = []
        q = [root]

        while q:
            l = len(q)
            r = q[0].val

            while l:
                front = q.pop(0)
                l-=1

                r = front.val

                if front.left:
                    q.append(front.left)
                if front.right:
                    q.append(front.right)

            res.append(r)

        return res
```

-> using dfs, give priority to right traversal with level and max_level, check at every level, if level > max_level, then only take that node in result.

```python
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        res = []
        self.max_level = -1

        def dfs(node, level):
            if not node:
                return

            if level > self.max_level:
                self.max_level = level
                res.append(node.val)

            if node.right:
                dfs(node.right, level+1)

            if node.left:
                dfs(node.left, level+1)

        dfs(root, 0)

        return res
```

-> check if binary tree symmetric around center:

-> check if node.left == node.right and vice-versa, check this condition for every node.

```python
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True

        def dfs(node1, node2):
            if not node1 and not node2:
                return True

            if node1 and node2:
                if node1.val == node2.val:
                    l = dfs(node1.left, node2.right)
                    r = dfs(node1.right, node2.left)

                    if l and r:
                        return True

            return False

        return dfs(root.left, root.right)
```

-> root to leaf paths:

```python
class Solution:
    def binaryTreePaths(self, root: Optional[TreeNode]) -> List[str]:

        res = []

        def dfs(node, curr):
            if not node:
                return

            if not node.left and not node.right:
                curr.append(str(node.val))
                res.append(curr[:])
                curr.pop()
                return

            curr.append(str(node.val))
            dfs(node.left, curr)
            dfs(node.right, curr)
            curr.pop()

        dfs(root, [])

        return ["->".join(path) for path in res]
```

-> lowest common ancestor:

-> there can be two cases:

1. when both nodes are present on same side of root, in that case, the first node which is found is that is the only LCA.
2. when both node are present on different side of root, in this case, find the node in which both nodes are present as left and right subtree.

```python
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':

        def dfs(node, p, q):
            if not node :
                return None

            if node.val == p.val or node.val == q.val:
                return node

            left = dfs(node.left, p, q)
            right = dfs(node.right, p, q)

            if left and right:
                return node

            return left if left else right

        return dfs(root, p, q)
```

-> maximum width of a binary tree:
-> use indexing inorder to index every node in binary tree and and take the first node index which when subtracted by last node of that level will give width of that level.
-> left child: 2*x +1, right child: 2*x +2

```python
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        maxi = 0
        q = [(0,root)]

        while q:
            l = len(q)
            left, right = -1, -1
            while l:
                x, front = q.pop(0)
                l-=1

                if left==-1:
                    left, right = x, x
                else:
                    right = x

                if front.left:
                    q.append((2*x +1 , front.left))

                if front.right:
                    q.append((2*x +2, front.right))

            maxi = max(maxi, right-left +1)


        return maxi
```

-> Print all nodes at distance k from target node:
-> first, maintain a map having every node's parent's reference.
-> then using the target node reference, start dfs from target node and accordingly travel to left, right and parent node. Also maintain a visited map to track a node is never visited twice.

```python
class Solution:
    def distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:

        self.map = {}

        def dfs(node):
            if not node:
                return

            if node.left:
                self.map[node.left.val] = node
                dfs(node.left)

            if node.right:
                self.map[node.right.val] = node
                dfs(node.right)

        res = []
        vis = {}
        def dfs2(node, k, curr):
            if not node:
                return

            vis[node.val] = True
            if curr == k:
                res.append(node.val)
                return

            if node.left and not vis.get(node.left.val):
                dfs2(node.left, k, curr+1)

            if node.right and not vis.get(node.right.val):
                dfs2(node.right, k, curr+1)

            if self.map.get(node.val) and not vis.get(self.map[node.val].val):
                dfs2(self.map[node.val], k, curr+1)


        dfs(root)
        dfs2(target, k, 0)

        return res
```

-> minimum time to burn the binary tree:
-> first find references to the parent of each node.
-> start dfs or bfs from given target node, also maintain a visited list so that same node cannot be traversed twice. Maintain a timer which increases with traversal.

```
def minTime(self, root,target):
    # code here
    self.parent = {}
    self.target_ref = None
    def bfs(node, target):
        q = [node]

        while q:
            front = q.pop(0)

            if front.data == target:
                self.target_ref = front

            if front.left:
                self.parent[front.left.data] = front
                q.append(front.left)

            if front.right:
                self.parent[front.right.data] = front
                q.append(front.right)

    bfs(root, target)

    vis = {}
    self.maxi = 0

    def dfs(node, t):
        if not node:
            return
        vis[node.data] = 1
        f = 0
        self.maxi = max(self.maxi, t)
        if node.left and not vis.get(node.left.data):
            dfs(node.left, t+1)
            f= 1

        if node.right and not vis.get(node.right.data):
            dfs(node.right, t+1)
            f= 1

        if self.parent.get(node.data) and not vis.get(self.parent[node.data].data):
            dfs(self.parent[node.data], t+1)
            f=1

    dfs(self.target_ref, 0)
    return self.maxi
```

-> Count complete binary tree nodes:
-> if binary tree is complete, like all levels are filled completely with nodes, for such a tree, total number of nodes is 2^h – 1, where h is the height of the binary tree.

-> if above case is not found, then, left_subtree_height (height of tree when traveling only left until possible) + right_subtree_height (height of tree when traveling only right until possible)+ 1 willbe our answer.

-> if complete binary tree is found then we do not need to travel whole tree.

```
class Solution:
    def countNodes(self, root: Optional[TreeNode]) -> int:
        def left_height(node):
            if not node:
                return 0

            lh= 1
            if node.left:
                lh += left_height(node.left)

            return lh


        def right_height(node):
            if not node:
                return 0

            rh= 1
            if node.right:
                rh += right_height(node.right)

            return rh


        def dfs(node):
            if not node:
                return 0

            lh = left_height(node)
            rh = right_height(node)

            if lh == rh:
                return 2**lh -1
            else:
                return 1 + dfs(node.left) + dfs(node.right)

        return dfs(root)
```

-> construct binary tree from inorder and preorder :
-> take 1st node in preorder and make it root, now find that node in inorder and use that index to divide preorder and inorder.

```python
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        if len(preorder) == 0 or len(inorder) == 0:
            return

        root = TreeNode(preorder[0])
        ind = 0

        for i in range(len(inorder)):
            if inorder[i] == preorder[0]:
                ind= i
                break

        root.left = self.buildTree(preorder[1:1+ind], inorder[:ind])
        root.right = self.buildTree(preorder[1+ind:], inorder[ind+1:])

        return root
```

-> construct binary tree from inorder and postorder :
-> take 1st node in postorder and make it root, now find that node in inorder and use that index to divide postorder and inorder.

```python
class Solution:
    def buildTree(self, inorder: List[int], postorder: List[int]) -> Optional[TreeNode]:
        if len(inorder)==0 or len(postorder) == 0:
            return
        root = TreeNode(postorder[-1])

        ind = 0
        for i in range(len(inorder)):
            if inorder[i] == root.val:
                ind = i
                break

        root.right = self.buildTree(inorder[ind+1:], postorder[-1-(len(postorder)-ind-1) : -1])
        root.left = self.buildTree(inorder[:ind], postorder[:-1-(len(postorder)-ind-1)])


        return root
```

-> flatten a binary tree into linked list (should look like preorder with each node's left set to NONE):
-> use the intuition behind morris traversal, in which we played with left and right pointers and made them point to their parents or some other node. In similar way, we can do this.
-> as it is to be done in preorder, two cases:
1. find the last rightmost node in the left subtree if exists.
2. get the parent node, if rightmost node not present.
-> made the last rightmost node point to current node's right node, and make current node's right node set to it's left node.
-> do above process for every node.

```python
class Solution:
    def flatten(self, root: Optional[TreeNode]) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        node = root

        while node:
            if node.left:
                temp = node.left

                while temp.right:
                    temp = temp.right

                temp.right = node.right
                node.right = node.left
                node.left = None

            node = node.right
```

-> search in binary search tree:

```python
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:

        def search(node, val):
            if not node:
                return

            if node.val == val:
                return node

            if node.val > val:
                return search(node.left, val)

            if node.val < val:
                return search(node.right, val)

        return search(root, val)
```

-> ceil in BST:

-> maintain a prev variable which will store which was the last greater number found till now.

```python
class Solution:
    def findCeil(self,root, inp):
        # code here

        def search(node, val, prev):
            if not node:
                return prev

            if node.key == val:
                return node.key

            if node.key > val:
                prev = node.key
                return search(node.left, val, prev)

            return search(node.right, val, prev)

        res = search(root, inp, -1)
        return res
```

-> floor in bst:

-> same as we found ceil.

```python
class Solution:
    def floor(self, root, x):
        # Code here

        def search(node, x, prev):
            if not node:
                return prev

            if node.data == x:
                return x

            if node.data > x:
                return search(node.left, x, prev)

            return search(node.right, x, node.data)

        return search(root, x, -1)
```

-> insert node in bst:
-> move left or right until found an appropriate position for node, it will be some leaf node where this new node will be inserted.

```python
class Solution:
    def insertIntoBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        if not root:
            root = TreeNode(val)
            return root

        node = root

        while node:
            if val > node.val:
                if node.right:
                    node = node.right
                else:
                    node.right = TreeNode(val)
                    break
            else:
                if node.left:
                    node = node.left
                else:
                    node.left = TreeNode(val)
                    break

        return root
```

-> delete a node from bst:
-> find that node, and make it's left child connected to last left node in node's right subtree.

```python
class Solution:
    def deleteNode(self, root: Optional[TreeNode], key: int) -> Optional[TreeNode]:
        def find_last_left(node):
            if not node.left:
                return node

            return find_last_left(node.left)

        def helper(node):
            if not node.right:
                return node.left
            if not node.left:
                return node.right

            left = node.left
            last_left_in_right = find_last_left(node.right)
            last_left_in_right.left = left
            return node.right
```

```python
        if not root:
            return


        if root.val == key:
            return helper(root)

        node = root

        while root:
            if root.val < key:
                if root.right and root.right.val == key:
                    root.right = helper(root.right)
                    break
                else:
                    root = root.right

            else:
                if root.left and root.left.val == key:
                    root.left = helper(root.left)
                    break
                else:
                    root = root.left


        return node
```

-> kth smallest node in bst:
-> in bst, if we do inorder traversal, we'll get nodes in sorted way.

```python
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        node = root
        stk = []

        while True:
            if node:
                stk.append(node)
                node = node.left
            else:
                if not stk:
                    break
                top = stk.pop()
                k-=1

                if k==0:
                    return top.val

                node = top.right
```

-> check if valid bst:
-> use inorder property of sorting, and while traversing if any node is found which is greater than the maximum element found till now then it is not a valid bst.

```python
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        stk = []
        node = root
        maxi = float('-inf')

        while True:
            if node:
                stk.append(node)
                node = node.left
            else:
                if not stk:
                    break

                top = stk.pop()
                if top.val <= maxi:
                    return False
                maxi = max(maxi, top.val)

                node = top.right

        return True
```

-> lca in bst:
-> if both given nodes (p & q) are smaller than root, move left, else, move right.

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        if p.val < q.val:
            pass
        else:
            p, q = q, p

        def lca(node, p, q):
            if not node:
                return

            if node.val > p.val and node.val > q.val:
                return lca(node.left, p, q)

            if node.val < p.val and node.val < q.val:
                return lca(node.right, p, q)

            return node


        return lca(root, p, q)
```

-> construct bst from preorder:

-> two ways:

1. take first node from given preorder, make it root, find the nodes which are lesser than this root node and nodes which are greater than this root node and apply same function.

```
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        if not preorder:
            return

        root = TreeNode(preorder[0])

        ind = 1
        while ind < len(preorder):
            if preorder[ind] > root.val:
                break
            ind += 1

        root.left = self.bstFromPreorder(preorder[1:ind])
        root.right = self.bstFromPreorder(preorder[ind:])

        return root
```

2. maintain current element position and also maintain an upper bound (till now which is the greatest node). If current node is less than upper bound use this node, now to find left and right child. Call same function with upper bound for left = root.val, upper bound for right = previous upper bound . Initially upper bound is int_max.

```python
class Solution:
    def bstFromPreorder(self, preorder: List[int]) -> Optional[TreeNode]:
        self.pos = 0
        def construct(preorder, ub):
            if self.pos >= len(preorder):
                return

            if preorder[self.pos] >= ub:
                return

            root = TreeNode(preorder[self.pos])
            self.pos += 1

            root.left = construct(preorder, root.val)
            root.right = construct(preorder, ub)

            return root

        return construct(preorder, float('inf'))
```

-> recover bst for bt where two nodes are swapped

-> there can be two cases:

1. elements to be swapped are adjacent in inorder traversal
2. elements are not adjacent.

-> to handle both cases, we need to maintain first, middle and second pointer to point to the elements that need to be swapped.

-> if elements are adjacent then, there will be only first and middle pointer, so swap them.

-> else, swap the first and second pointer

-> also maintain a prev pointer that is pointing to the previous element in inorder traversal.

```python
class Solution:
    def recoverTree(self, root: Optional[TreeNode]) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        self.first = None
        self.middle = None
        self.second = None
        self.prev = None

        def inorder(node):
            if not node:
                return

            inorder(node.left)

            if self.prev:
                if self.prev.val >= node.val:
                    if not self.first:
                        self.first = self.prev
                        self.middle = node
                    else:
                        self.second = node
            self.prev = node
            inorder(node.right)

        inorder(root)

        if self.second:
            self.first.val, self.second.val = self.second.val, self.first.val
        else:
            self.first.val, self.middle.val = self.middle.val, self.first.val
```

-> find largest bst in bt:
-> start from bottom of bt and maintain (size, left_max, right_min).
-> if maximum of left bst < curr val and minimum of right bst > curr val, then include current node in bst with size = left_size + right_size + 1, and, left_max will be max(curr_val, right_max) and right_min will be min(curr_val, left_min).
-> if above condition doesn't satisfy, return size = max(left_size, right_size) and left_max = float(inf) and right_min = float(-inf), as this is not proper bst,in future to avoid comparison with this subtree we set such values.
-> for leaf node, return size = 1, left_max = curr_val, right_min = curr_val
-> for null node, return size =0 ,left_max = float(-inf), right_min = float(inf)

```python
class Solution:
    # Return the size of the largest sub-tree which is also a BST
    def largestBst(self, root):
        #code here
        def postorder(node):
            if not node:
                return (0, float('-inf'), float('inf'))

            if not node.left and not node.right:
                return (1, node.data, node.data)

            lc, lmax, lmin = postorder(node.left)

            rc, rmax, rmin = postorder(node.right)

            if lmax < node.data and rmin > node.data:
                s, maxi, mini = lc+rc+1, max(rmax, node.data), min(lmin, node.data)
            else:
                s, maxi, mini = max(lc,rc), float('inf'), float('-inf')

            return (s,maxi, mini)

        return postorder(root)[0]
```