Contents

[1. Reverse Linked List in k groups 2](#_Toc476489678)

[2. Sort Linked list using Merge sort 3](#_Toc476489679)

[3. Find intersection of two LinkedLists 3](#_Toc476489680)

[4. Valid Parentheses 4](#_Toc476489681)

[5. WildCard 4](#_Toc476489682)

[6. Longest Valid Parentheses 4](#_Toc476489683)

[7. Expression evaluator 4](#_Toc476489684)

[8. Palindrome partitions 5](#_Toc476489685)

[9. Is Bst 6](#_Toc476489686)

[10. Flip a Tree (create mirror image) 6](#_Toc476489687)

[11. Print root to leaf path 7](#_Toc476489688)

[12. Build Tree (InOrder-Pre/PostOrder) 7](#_Toc476489689)

[13. BST to Doubly Circular Linked List (InOrder sort) 9](#_Toc476489690)

[14. BST Iterator (small to larger number – InOrder sort) 9](#_Toc476489691)

[15. Clone Binary Tree 10](#_Toc476489692)

[16. Merge two BST 11](#_Toc476489693)

[17. Post-Order Traversal 14](#_Toc476489694)

[18. Lowest Common Ancestor of a Tree 15](#_Toc476489695)

[19. Identical Trees 16](#_Toc476489696)

[20. Symmetric Binary Tree 16](#_Toc476489697)

[21. Kth smallest in BST 17](#_Toc476489698)

[22. IsBalanced Tree 17](#_Toc476489699)

[23. Find Next Greater Number in BST 18](#_Toc476489700)

[24. Height of a Tree (K-ary tree?) 18](#_Toc476489701)

[25. Find K largest (or smallest) elements from a Tree 18](#_Toc476489702)

[26. Sorted Array to Balanced BST 18](#_Toc476489703)

[27. Sibling/Next Pointer in Binary Tree 18](#_Toc476489704)

[28. Single Value binary tree 20](#_Toc476489705)

[29. Largest BST in a tree 21](#_Toc476489706)

[30. New problem 21](#_Toc476489707)

# Reverse Linked List in k groups

<todo>

# Sort Linked list using Merge sort

**Tags:** LinkedList, Sorting, Merge sort, LeetCode

Merge sort’s merge operation usually requires additional space for merging into new array. But, using LinkstList that need can be avoided.

public int getLen(ListNode h) { }

public ListNode merge(ListNode l, ListNode r) { }

public ListNode mergeRecursive(ListNode l, ListNode r)

{

// Stack overflow problem on LeetCode with around 31k size length

}

public ListNode sortList(ListNode head, int len)

{

int mid = len/2;

ListNode left = sortList(firstHalf, mid);

ListNode right = sortList(secondHalf, len - mid);

return merge(left, right);

}

# Find intersection of two LinkedLists

**Tags:** LinkedList, LeetCode

Given that m and n are lengths of two LLs -

1. Time O(m\*n); Space O(1)

For each node in m, search entire LinkedList n. If you find a match return that node.

1. **Time O(m+n); Space O(1)**

Find length of each list; Then for longest list traverse abs(m-n) nodes; last step - traverse both lists one at a time till they are same;

1. Time O(m+n); space O(m+n)

Using hashing (set can be used too)

Traverse first list and hash each node; for second list while traversing each node check in hash if it is present; if matched report that node as intersection

1. Time O(m+n); space O(m+n)

Use stack; Idea is use the fact that both list share same size from the end.

While traversing both the list put them in 2 separate stacks; now start poping each element from two stacks simultaneously, if they match report that node as intersection.

# Valid Parentheses

**Tags:** Stack, LeetCode

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not

Solution:

Use a stack to push (, { and ]. Pop during closing brackets.

# WildCard

**Tags:** Recursion

Given a string 123?, replace ? with ‘0’ or ‘1’. So, output will be 1230, 1231

# Longest Valid Parentheses

**Tags:** Stack, LeetCode

**Problem statement:**

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

**Solution [1]**: Time O(n); Space O(n)

Keep pushing **index** of ‘(‘ in the stack. On finding ‘)’ pop the index and calculate the length (current\_index – index\_of\_top\_element\_from\_stack)

# Expression evaluator

**Tags:** Leetcode, recursion

**Problem statement:**

Given a string that contains only digits 0-9 and a target value, return all possibilities to add binary operators (not unary) +, -, or \* between the digits so they evaluate to the target value.

Examples:

"123", 6 -> ["1+2+3", "1\*2\*3"]

"232", 8 -> ["2\*3+2", "2+3\*2"]

"105", 5 -> ["1\*0+5","10-5"]

"00", 0 -> ["0+0", "0-0", "0\*0"]

"3456237490", 9191 -> []

**Solution:**

Time: O(3^(n-1)); n – number of digits and 3 – number of operators

Space: O(n); size of stack

Using backtracking approach, one can quickly code it’s solution

# Palindrome partitions

**Tags:** LeetCode, recursion

**Problem statement:**

If given string is ‘aab’, then return palindrome partitions like this [ [“aa”, “b”], [“a”, “a”, “b”]]

**Solution:**

Time: O(n\* 2^n); n – Palindrome check; 2^n total substring partitioning.

Space: O(n) –

T(n) = T(n-1) + T(n-2) + … + 1

T(n+1) = T(n) + T(n-1) + … +1 = T(n) + T(n) = 2\*T(n)

Using backtracking, it can be solved easily.

public void findPartitions(string s, int start, List<IList<string>> res, List<string> buff) {

if (s.Length == start) {

res.Add(new List<string> (buff));

return;

}

for(int i=start ; i<s.Length ; i++) {

if (isPalindrome(s, start, i)) {

buff.Add(s.Substring(start, i-start+1));

findPartitions(s, i+1, res, buff);

buff.RemoveAt(buff.Count-1);

}

}

}

# Is Bst

**Tags:** LeetCode, homework

**Problem statement:** Check if a Binary Tree is a BST. That tree does not have any duplicates

**Answer:** Time O(n), Space O(n)

<http://articles.leetcode.com/determine-if-binary-tree-is-binary>

Start with left and right values and keep reducing based on sub tree we explore.

# Flip a Tree (create mirror image)

**Tags:** LeetCode, homework

**Problem statement:** Convert a tree into its mirror image

**Answer:**

[1] Recursive approach: Time O(n); Space O(h)

Change left and right from its root. After that make two recursive calls to same function with left and right subtrees.

[2] Iterative approach: Time O(n); Space O(n/2)

Use BFS using queue. Then while putting before left and right into queue, change pointer of left and right tree

<https://leetcode.com/articles/invert-binary-tree/>

# Print root to leaf path

**Tags:** LeetCode, Homework

**Solution:** Time O(n\*h) (n: Total nodes; h: string from root to leaf); Space O(h)

Recursive solution.

class Solution {

static void helper(TreeNode root, LinkedList<int> buff) {

if (root == null) {

return;

}

buff.AddLast(root.val);

if (root.left == null && root.right == null) {

Console.WriteLine(string.Join(" ", buff));

}

else {

if (root.left != null) {

helper(root.left, buff);

}

if (root.right != null) {

helper(root.right, buff);

}

}

buff.RemoveLast();

}

}

# Build Tree (InOrder-Pre/PostOrder)

**Tags:** Homework

**Question:** Given the in-order and pre-order traversing results of a binary tree (as arrays), write a function to rebuild the tree. The function should return the pointer to the root node of the tree. Then take that pointer, and print your tree level by level (level order).

**Trivia:** Generally speaking, one needs to be given in-order traversal (with either pre or post or level), as input, in order to re-construct a binary tree. Without in-order traversal given, it’s not possible to re-construct a binary tree without ambiguity, even if all other 3 traversal orders are given. The only exception, is if we know something more about the tree e.g. if the binary tree is full and complete, then we can resolve the ambiguity without having to know the in-order traversal. [Something to read: <http://www.geeksforgeeks.org/if-you-are-given-two-traversal-sequences-can-you-construct-the-binary-tree/> ]

**Solutions:**

<http://articles.leetcode.com/2011/04/construct-binary-tree-from-inorder-and-preorder-postorder-traversal.html>

<http://edwardliwashu.blogspot.com/2013/01/construct-binary-tree-from-preorder-and.html>

<https://www.youtube.com/watch?v=PAYG5WEC1Gs>

**Two approaches:**

* First node in preorder represents root node. We must search that node in in-order array. Left side of remaining nodes represents left subtree and right side represents right subtree

static TreeNode create(int[] inArr, int[] preArr, Dictionary<int, int> hash, int remaining, int preStart, int inStart)

{

if (remaining == 0)

{

return null;

}

int val = preArr[preStart];

int i = hash[val] - inStart;

TreeNode root = new TreeNode(val);

root.left = create(inArr, preArr, hash, i, preStart + 1, inStart);

Console.WriteLine($"start: {preStart + i + 1}, offset: {hash[val] + 1}");

root.right = create(inArr, preArr, hash, remaining - i - 1, preStart + i + 1, hash[val] + 1);

return root;

}

static TreeNode create\_using\_Length(int[] inArr, int inStart, int inEnd, int[] preArr, int preStart, int preEnd,

Dictionary<int, int> hash)

{

if (inStart > inEnd)

{

return null;

}

int val = preArr[preStart];

int len = hash[val] - inStart;

TreeNode root = new TreeNode(val);

root.left = create\_using\_Length(inArr, inStart, hash[val] - 1, preArr, preStart + 1, preStart + len - 1, hash);

root.right = create\_using\_Length(inArr, hash[val] + 1, inEnd, preArr, preStart + len + 1, preEnd, hash);

return root;

}

# BST to Doubly Circular Linked List (InOrder sort)

**Tags**: Homework, LeetCode

**Question:** Convert a BST to a sorted circular doubly-linked list (list will be sorted InOrder manner). Think of the left and right pointers as synonymous to the previous and next pointers in a doubly-linked list.

**Solution**:

[1] We can use divide and conquer approach. From the root node, we’ll split the problem to left and right subtree. After that, we stitch left-subtree-result with root and right-subtree-result. (<http://cslibrary.stanford.edu/109/TreeListRecursion.html> )

TreeNode BstToDLLHelper(TreeNode node) {

if (node == null) {

return null;

}

var lList = BstToDLLHelper(node.left);

var rList = BstToDLLHelper(node.right);

var selfLoop = SelfLoop(node);

lList = append(lList, selfLoop);

lList = append(lList, rList);

return lList;

}

[2] There is 2nd approach as well refer: <http://articles.leetcode.com/convert-binary-search-tree-bst-to/>

# BST Iterator (small to larger number – InOrder sort)

**Tags:** LeetCode, Homework, InterviewBit

**Question:** Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

1. Calling next() will return the next smallest number in the BST.

2. Calling hasNext() should return whether the next element exists.

Both functions should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

**Solutions:**

Choice of the solution will depend on what the interviewer asks you to do. #2 is generally preferred i.e. without assuming there is a parent pointer.

1. With parent pointer: <http://stackoverflow.com/questions/12850889/in-order-iterator-for-binary-tree>

2. Without parent pointer, but with stack: <https://leetcode.com/discuss/20001/my-solutions-in-3-languages-with-stack>

With approach #2, there is amortized cost associated. So, even if it looks otherwise; the average time complexity is O(1).

Basically, we put every element from root to leaf on left edge of tree. As we pop the element we do same thing for popped element’s right node. If stack is non empty, it will have next element available.

public class BSTIterator {

public Stack<TreeNode> stk = new Stack<TreeNode> ();

private void stackAll(TreeNode node) {

for (; node != null ; node = node.left) {

stk.Push(node);

}

}

public BSTIterator(TreeNode root) {

stackAll(root);

}

public bool HasNext() {

return stk.Count > 0;

}

public int Next() {

var node = stk.Pop();

stackAll(node.right);

return node.val;

}

}

# Clone Binary Tree

**Tags:** Homework

**Question:** Given a binary tree (represented by its root node, like usual), clone it. Return the root node of the cloned tree.

**Remember:** Cloning or copying a tree is best done recursively. Notice how clean and succinct the code is. Some of you may be tempted to do it breadth-first. But that's more complicated to handle in implementation.

TreeNode cloneTree(TreeNode node) {

if (node == null) {

return null;

}

var copyNode = new TreeNode(node.val);

copyNode.left = cloneTree(node.left);

copyNode.right = cloneTree(node.right);

return copyNode;

}

# Merge two BST

**Tags:** Homework

**Question:** Merge two BSTs in O(N1 + N2) time, where N1 and N2 are number of nodes in the two trees respectively. The merged tree should contain all the elements of both trees and also be a balanced BST. Finally, print the new tree level by level.

e.g.

Tree-1: 2->1,3

Tree-2: 7->6,8

Output:

6

2 7

1 3 8

The output above is a tree that’s printed level by level.

**Hint:** (This is a very good question. It's not hard at all, but will need you to write several functions: one to parse, one to sort, one to merge, one to reconstruct and one to print. Each of these can be separate short interview questions)

<http://stackoverflow.com/questions/7540546/merging-2-binary-search-trees>

(This doesn’t have the code to print the tree. But that’s quite easy. Just do BFS with a Queue and insert a sentinel at the beginning of each level)

**Solution:**

1. Time: O(n+m); Space: O(1) (ignoring stack size)

static TreeNode convertToList(TreeNode r, TreeNode list) {

if (r == null) {

return list;

}

var leftList = convertToList(r.left, list);

leftList.right = r;

leftList = r;

r.left = null;

var rightList = convertToList(r.right, leftList);

return rightList;

}

static TreeNode mergeTwoLists(TreeNode h1, TreeNode h2) {

if (h1 == null) return h2;

if (h2 == null) return h1;

if (h1.val < h2.val) {

h1.right = mergeTwoLists(h1.right, h2);

return h1;

} else {

h2.right = mergeTwoLists(h1, h2.right);

return h2;

}

}

static TreeNode makeBST(ref TreeNode list, int len) {

if (len <= 0) {

return null;

}

int rem = len - (len / 2) - 1;

var leftTree = makeBST(ref list, len / 2);

var head = list;

head.left = leftTree;

list = list.right;

var rightTree = makeBST(ref list, rem);

head.right = rightTree;

return head;

}

static TreeNode mergeTrees(TreeNode node1, TreeNode node2) {

// 1. convert into a linked list

var h1 = new TreeNode(-1); // dummy node

convertToList(node1, h1);

var h2 = new TreeNode(-1); // dummy node

convertToList(node2, h2);

// 2. Merge two linked lists into one

var head = mergeTwoLists(h1.right, h2.right);

// 3. Divide & Conquer on a linkedlist to create a tree

int len = 0;

var tmp = head;

while (tmp != null) {

tmp = tmp.right;

len++;

}

return makeBST(ref head, len);

}

1. Time: O((n+m) \* log(n+m)); Space: O(1)

class Solution {

static TreeNode convertToList(TreeNode r, ref TreeNode prev) {

if (r == null) {

return null;

}

TreeNode h1 = convertToList(r.left, ref prev);

if (prev != null) {

prev.right = r;

}

prev = r;

r.left = null;

TreeNode h2 = convertToList(r.right, ref prev);

return (h1 == null) ? r : h1;

}

static TreeNode mergeTwoLists(TreeNode h1, TreeNode h2) {

var n1 = h1;

var n2 = h2;

var t = new TreeNode(0);

var head = t;

while (n1!=null && n2!=null) {

if (n1.val < n2.val) {

t.right = n1;

n1 = n1.right;

} else {

t.right = n2;

n2 = n2.right;

}

t = t.right;

}

if (n1 == null) {

t.right = n2;

} else if (n2 == null) {

t.right = n1;

}

return head.right;

}

static TreeNode convertToBST(TreeNode r, int len) {

if (len == 0 || len == 1) {

if (r != null) {

r.left = r.right = null;

}

return r;

}

int mid = len/2;

int c=1;

var n = r;

TreeNode prev = null;

while (c <= mid) {

prev = n;

n = n.right;

c++;

}

prev.right = null;

n.left = convertToBST(r, mid);

n.right = convertToBST(n.right, len - mid - 1);

return n;

}

static int findLength(TreeNode r) {

int c=0;

while (r != null) {

c++;

r = r.right;

}

return c;

}

static TreeNode mergeTrees(TreeNode node1, TreeNode node2) {

TreeNode prev = null;

var h1 = convertToList(node1, ref prev);

prev = null;

var h2 = convertToList(node2, ref prev);

h1 = mergeTwoLists(h1, h2);

h1 = convertToBST(h1, findLength(h1));

return h1;

}

}

# Post-Order Traversal

**Tags:** Homework, LeetCode, InterviewBit

**Problem:** Do a post order traversal without recursion

**Solution:** Use stack

<http://articles.leetcode.com/binary-tree-post-order-traversal/>

static void postOrderTraversal(TreeNode r) {

Stack<TreeNode> stack = new Stack<TreeNode>();

Stack<int> output = new Stack<int>();

stack.Push(r);

while (stack.Count != 0) {

var node = stack.Pop();

output.Push(node.val);

if (node.left != null) {

stack.Push(node.left);

}

if (node.right != null) {

stack.Push(node.right);

}

}

while (output.Count != 0) {

Console.Write(output.Pop() + " ");

}

}

# Lowest Common Ancestor of a Tree

**Tags:** Homework, LeetCode, InterviewBit

**Question:** Find the lowest common ancestor of a tree

**Caveat:** (1) Constraint may state that both the nodes are in tree or may not be in a tree. (Question on InterviewBit says it may not have both the nodes in a Tree)

(2) One of the LeetCode article has a version where it says each node does have parent link. This one can be solved without recursion.

(1) <http://articles.leetcode.com/lowest-common-ancestor-of-a-binary-tree-part-i/>

(2) <http://www.leetcode.com/2011/07/lowest-common-ancestor-of-a-binary-tree-part-ii.html>

**Solution:**

class Solution {

static TreeNode findLCA(TreeNode node) {

int n1 = Convert.ToInt32(Console.ReadLine());

int n2 = Convert.ToInt32(Console.ReadLine());

if (find(node, n2) == false || find(node, n1) == false) {

return null;

}

return helper(node, n1, n2);

}

public static TreeNode helper(TreeNode node, int n1, int n2) {

if (node == null) {

return null;

}

else if (node.val == n1 || node.val == n2) {

return node;

}

var lRes = helper(node.left, n1, n2);

var rRes = helper(node.right, n1, n2);

if (lRes != null && rRes != null) {

return node;

}

return (lRes != null) ? lRes : rRes;

}

public static bool find(TreeNode node, int n) {

if (node == null) {

return false;

}

return (

(node.val == n) ||

(find(node.left, n) == true) ||

(find(node.right, n) == true)

);

}

}

# Identical Trees

**Tags:** InterviewBit

**Question:** Check if two trees are identical.

**Solution:**

public bool isSameTree(TreeNode A, TreeNode B) {

if (A == null && B == null) {

return true;

}

if (A == null || B == null) {

return false;

}

return (A.val == B.val) && isSameTree(A.left, B.left) && isSameTree(A.right, B.right);

}

# Symmetric Binary Tree

**Tags:** InterviewBit

**Question:** Check if two trees are Symmetric

**Solution:**

# Kth smallest in BST

**Tag:** InterviewBit, IK-Test

**Question:** Find Kth smallest element in BST

**Solution:**

public int kthsmallest(TreeNode root, int k) {

return helper(root, ref k);

}

public int helper(TreeNode root, ref int k) {

if (root == null) {

return -1;

}

var res = helper(root.left, ref k);

if (k == 0) {

return res;

}

k--;

if (k == 0) {

return root.val;

}

return helper(root.right, ref k);

}

# IsBalanced Tree

**Tags:** InterviewBit

**Question:** Find if a tree is balanced.

**Solution:**

public bool isBalanced(TreeNode root) {

var res = isBalancedHelper(root);

return res.Item2;

}

public Tuple<int, bool> isBalancedHelper(TreeNode root) {

if (root == null)

{

return new Tuple<int, bool>(0, true);

}

var l = isBalancedHelper(root.left);

if (l.Item2 == false)

{

return new Tuple<int, bool>(0, false);

}

var r = isBalancedHelper(root.right);

if (r.Item2 == false)

{

return new Tuple<int, bool>(0, false);

}

var diff = Math.Abs(l.Item1 - r.Item1);

return new Tuple<int, bool>(1 + Math.Max(l.Item1, r.Item1), diff <= 1);

}

# Find Next Greater Number in BST

# Height of a Tree (K-ary tree?)

# Find K largest (or smallest) elements from a Tree

# Sorted Array to Balanced BST

**Tags:** IK-Test, InterviewBit

**Solution:**

public TreeNode SortedArrayToBST(int[] nums) {

return helper(nums, 0, nums.Length - 1);

}

public TreeNode helper(int[] nums, int start, int end) {

if (start > end) {

return null;

}

int mid = start + ((end - start) / 2);

TreeNode node = new TreeNode(nums[mid]);

node.left = helper(nums, start, mid - 1);

node.right = helper(nums, mid + 1, end);

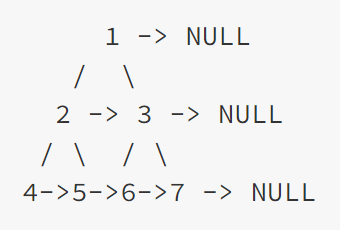
return node;

}

# Sibling/Next Pointer in Binary Tree

**Tags:** Homework, InterviewBit

**Question:**



**Solution:**

public void connect(TreeLinkNode root) {

while (root != null) {

TreeLinkNode p = root;

TreeLinkNode c = null;

TreeLinkNode start = null;

while (p != null) {

if (p.left != null) {

if (c == null) {

c = p.left;

start = c;

} else {

c.next = p.left;

c = c.next;

}

}

if (p.right != null) {

if (c == null) {

c = p.right;

start = c;

} else {

c.next = p.right;

c = c.next;

}

}

p = p.next;

}

root = start;

}

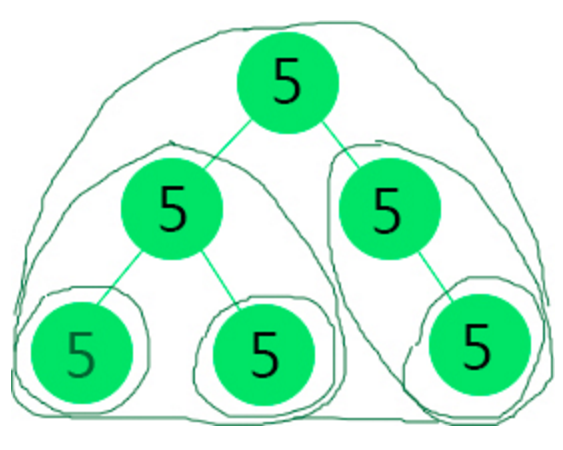
}

# Single Value binary tree

**Tags:** Homework

**Question:**

Given a binary tree, we need to count the number of uni-value **subtrees** (all nodes that have same value)



**Solution:**

**Approach –** We need at least two types of information to pass from leaf to top node. One is left/right subtree is unival

static bool findUniVal(TreeNode r, ref int count) {

if (r == null) {

return true;

}

var lVal = findUniVal(r.left, ref count);

var rVal = findUniVal(r.right, ref count);

if (lVal == true && rVal == true) {

// leaf

if (r.left == null && r.right == null) {

count++;

return true;

}

if (r.left != null && r.right != null && r.val == r.left.val && r.val == r.right.val) {

count++;

return true;

} else if (r.left == null && r.right != null && r.val == r.right.val) {

count++;

return true;

} else if (r.left != null && r.right == null && r.val == r.left.val) {

count++;

return true;

}

}

return false;

}

static int findNumberOfSVT(TreeNode node) {

int count = 0;

findUniVal(node, ref count);

return count;

}

# Largest BST in a tree

**Tags:** Homework

**Solution:** We need to find out if a given tree is a BST (left tree < root > right tree). We pass this info from bottom to top.

static int findLargestBSTHelper(TreeNode node, ref TreeNode small, ref TreeNode large, ref int size) {

if (node == null) {

return 0;

}

bool isBST = true;

int leftNodes = findLargestBSTHelper(node.left, ref small, ref large, ref size);

var lSmall = (leftNodes == 0) ? node : small;

var lLarge = (leftNodes == 0) ? node : large;

int rightNodes = findLargestBSTHelper(node.right, ref small, ref large, ref size);

var rSmall = (rightNodes == 0) ? node : small;

var rLarge = (rightNodes == 0) ? node : large;

if (leftNodes == -1 || (leftNodes != 0 && lLarge.val >= node.val) || rightNodes == -1 || (rightNodes != 0 && rSmall.val < node.val))

{

isBST = false;

}

if (isBST == true) {

small = lSmall;

large = rLarge;

int total = leftNodes + rightNodes + 1;

if (size < total) {

size = total;

}

return total;

}

return -1; // not BST

}

static int findLargestBST(TreeNode node) {

TreeNode small = null;

TreeNode large = null;

int size = 0;

findLargestBSTHelper(node, ref small, ref large, ref size);

return size;

}

# New problem