虽然有很多分类，但是Tree的大多数题目都可以用Recursion来做

Traversal-经典的三序遍历+level order traversal

144. Binary Tree Preorder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

**S1. Recursion**

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> list = new ArrayList<>();

if (root == null) return list;

subTree(list, root);

return list;

}

public void subTree(List<Integer> list, TreeNode root) {

list.add(root.val);

if (root.left != null) {

subTree(list, root.left);

}

if (root.right != null) {

subTree(list, root.right);

}

}

**S2, Iteration**

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<>();

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

TreeNode cur = root;

while (cur != null || !s.isEmpty()) {

if (cur != null) {

res.add(cur.val); // 先 add, 再继续走

s.push(cur); *// will be used for looking for cur.right*

cur = cur.left;

} else {

cur = s.pop();

cur = cur.right;

}

}

return res;

}

Inorder Traversal-

如果是BST, 那么inorder的顺序刚好是从小到大的顺序。

94. Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

**S1.Iteration**

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<>();

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

TreeNode cur = root;

while (cur != null || !s.isEmpty()) {

if (cur != null) {

s.push(cur); // used to get cur node and cur.right node

cur = cur.left;

} else {

cur = s.pop();

res.add(cur.val);

cur = cur.right;

}

}

return res;

}

655. Print Binary Tree

Print a binary tree in an m\*n 2D string array following these rules:

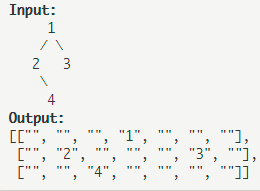
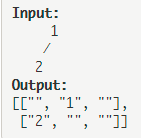
The row number m should be equal to the height of the given binary tree.

The column number n should always be an odd number.

The root node's value (in string format) should be put in the exactly middle of the first row it can be put. The column and the row where the root node belongs will separate the rest space into two parts (left-bottom part and right-bottom part). You should print the left subtree in the left-bottom part and print the right subtree in the right-bottom part. The left-bottom part and the right-bottom part should have the same size. Even if one subtree is none while the other is not, you don't need to print anything for the none subtree but still need to leave the space as large as that for the other subtree. However, if two subtrees are none, then you don't need to leave space for both of them.

Each unused space should contain an empty string "".

Print the subtrees following the same rules.



题解：

这道题其实是类似binary search

输出的list中一共有几个list，每个list的size其实在一开始就根据tree确定了，所以分为两步：

1. 初始化
2. 从root开始，recursion，把每个node的数值放入list的合适位置

如何确定合适位置？其实就是靠“root所在位置一定是left, right的中间点”，也就是相当于通过(start+end)/2来确定要放node的地点

public List<List<String>> printTree(TreeNode root) {

List<List<String>> res = new ArrayList<>();

int height = root == null ? 1 : getHeight(root);

int rows = height, cols = (int) Math.pow(2, height) - 1;

*// 1. create the lists, and original value in all lists:*

List<String> row = new ArrayList<>();

for (int i = 0;i < cols;i++) {

row.add("");

}

for (int i = 0;i < rows;i++) {

res.add(new ArrayList<>(row));

}

*// 2.set the values in each row:*

helper(res, root, 0, rows, 0, cols - 1);

return res;

}

private void helper(List<List<String>> res, TreeNode root, int row, int rows, int i, int j) {

if (row == rows || root == null) return;

int mid = (i + j) / 2;

res.get(row).set((i + j)/2, Integer.toString(root.val));

helper(res, root.left, row + 1, rows, i, mid - 1);

helper(res, root.right, row + 1, rows, mid + 1, j);

}

public int getHeight(TreeNode root) {

if (root == null) return 0;

return 1 + Math.max(getHeight(root.left), getHeight(root.right));

}

230. Kth Smallest Element in a BST

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

**S1. inorder**

最容易想到的方法是inorder traversal直接数到第k个即可。

public int kthSmallest(TreeNode root, int k) {

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

int count = 1;

while (root != null || !s.isEmpty()) {

if (root != null) {

s.push(root);

root = root.left;

} else {

TreeNode cur = s.pop();

if (count == k) return tmp.val;

count++;

root = cur.right;

}

}

return 0;

}

**S2. Recursion**

每次数left-subtree中node的个数，根据count与k的关系进行recursion,

注意k是从1开始算，所以要看k与count+1的关系

public int kthSmallest(TreeNode root, int k) {

int count = getCount(root.left);

if (k <= count) return kthSmallest(root.left, k);

else if (k == count + 1) return root.val;

else return kthSmallest(root.right, k - count - 1);// - left-subtree and root

}

// get # of nodes for the tree.

int getCount(TreeNode root) {

if (root == null) return 0;

return 1 + getCount(root.left) + getCount(root.right);

}

538. Convert BST to Greater Tree

Given a Binary Search Tree (BST), convert it to a Greater Tree such that every key of the original BST is changed to the **original key plus sum of all keys greater than the original key in BST.**

**S1 inorder traversal反过来**

这就是要按right-root-left来做，注意在加之前保存一下cur node的val值，

public TreeNode convertBST(TreeNode root) {

if (root == null) return null;

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

TreeNode cur = root;

int sum = 0;

while (!s.isEmpty() || cur != null) {

if (cur != null) {

s.push(cur);

cur = cur.right;

} else {

TreeNode tmp = s.pop();

int save = tmp.val;

tmp.val += sum;

sum += save;

cur = tmp.left;

}

}

return root;

}

285. Inorder Successor in BST

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

Note: If the given node has no in-order successor in the tree, return null.

**Follow up1:** 当TreeNode有parent指针的时候，如何优化空间

**Follow up2:** 当TreeNode没有parent指针，但是给出了root的时候，如何优化空间，不用stack

**S1. Inorder traversal – basic, Space O(h)**

public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {

boolean reached = false;

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

TreeNode cur = root;

while (cur != null || !s.isEmpty()) {

if (cur != null) {

s.push(cur);

cur = cur.left;

} else {

cur = s.pop();

if (reached) return cur;

if (cur == p) reached = true;

cur = cur.right;

}

}

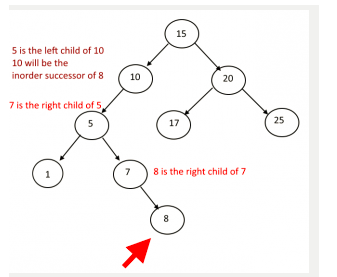
return null;

}

**Follow UP1 优化空间：当TreeNode 有parent指针的时候：**

需要考虑inorder successor的各种情况：

* 如果有right-subtree, successor就是right-subtree的left-most node
* 如果没有right-subtree, traverse parent until current node is the left child of parent，and the parent 就是要找的inorder successor, 比如下面这个图，8的inorder successor就是10；



TreeNode findInorderSuccessor(TreeNode node) {

if (node == null) return null;

if (node.right != null) {

return leftMost(node.right);

}

TreeNode parent = node.parent;

while (parent != null && node == parent.right) {

node = parent;

parent = parent.parent;

}

return parent;

}

TreeNode leftMost(TreeNode node) {

while (node.left != null) {

node = node.left;

}

return node;

}

**FollowUp2: 没有parent，但是提供root**

情况还是和之前一样，只是这一次在没有右子树的时候，需要从root往下走，来找node

根据root.val与node.val之间的大小关系来判断向左还是向右，然后root指针只负责继续走，直到走到node（两者指向一个点），用一个prev来记录可能的successor，prev就相当于是root的parent，不一定是direct parent.

因为inorder successor.val一定大于node,val, 所以只有在root.val > node.val的时候才更新prev的值。

TreeNode findInorderSuccessorWithRoot(TreeNode node, TreeNode root) {

if (node == null) return null;

if (node.right != null) {

return leftMost(node);

}

TreeNode prev = null;

while (root != null) {

if (root.val > node.val) {

prev = root;

root = root.left;

} else if (root.val < node.val) {

root = root.right;

} else break;

}

return prev;

}

173. Binary Search Tree Iterator

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

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

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

**S1 Inorder traversal**

其实就是一个个node走，不一次遍历完而已。

把inorder-traversal拆开即可。

public class BSTIterator {

Stack<TreeNode> stack;

TreeNode cur; // 这个其实不用作为全局变量，next()中的局部变量也可以

public BSTIterator(TreeNode root) {

stack = new Stack<>();

putAll(root); // prepare for the methods later.

cur = null;

}

// walk all the way to the left, save all the nodes on the way, till find min.

private void putAll(TreeNode root) {

while (root != null) {

stack.push(root);

root = root.left;

}

}

*/\*\** ***@return*** *whether we have a next smallest number \*/*

public boolean hasNext() {

return !stack.isEmpty();

}

*/\*\** ***@return*** *the next smallest number \*/*

public int next() {

cur = stack.pop();

putAll(cur.right);

return cur.val;

}

}

145. Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

**S. Stack, Iteration**

需要存两个node，第一个是current node，正在被process的node；

第二个是prev node, 上一个被process的node，不一定是parent

有三种情况：

1. left or right sub-tree not finished, 左子树/右子树还没有走完
2. just finished left sub-tree, check right side，此时cur.left == prev
3. already finished left and right sub-trees, time to deal with cur.

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> res = new ArrayList<>();

if (root == null) return res;

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

TreeNode cur = root;

TreeNode prev = null;

s.push(root);

while (!s.isEmpty()) {

cur = s.peek();

if (prev == null || prev.left == cur || prev.right == cur) { // 1.

if (cur.left != null) s.push(cur.left);

else if (cur.right != null) s.push(cur.right);

} else if (cur.left == prev) { // 2.

if (cur.right != null) s.push(cur.right);

} else { // 3.

res.add(cur.val);

s.pop();

}

prev = cur;

}

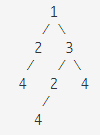
return res;

}

652. Find Duplicate Subtrees

Given a binary tree, return all duplicate subtrees. For each kind of duplicate subtrees, you only need to return the root node of any one of them.

Two trees are duplicate if they have the same structure with same node values.

 has two duplicate trees:  and 

Therefore, you need to return above trees' root in the form of a list.

**S1. Post-order traversal**

这个思路非常好：

map中存的是<subtree, count>, 其中subtree是以tree的string形式存起来的；

只在count == 1时加入res保证对于duplicate subtree而言，每个只会被加进一次。

public List<TreeNode> findDuplicateSubtrees(TreeNode root) {

List<TreeNode> res = new LinkedList<>();

postorder(root, new HashMap<>(), res);

return res;

}

public String postorder(TreeNode cur, Map<String, Integer> map, List<TreeNode> res) {

if (cur == null) return "#";

String serial = cur.val + "," + postorder(cur.left, map, res)

+ "," + postorder(cur.right, map, res);

if (map.getOrDefault(serial, 0) == 1) {

res.add(cur);

}

map.put(serial, map.getOrDefault(serial, 0) + 1);

return serial;

}

105. Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

**S1. Recursion**

通过preorder和inorder traversal重建tree的重点就是，

1. 先通过preorder的第一个节点找到pivot,
2. 然后在inorder中找到这个点，可以用来区分left/right part
3. 再通过inorder中的left part的长度来区分preorder

public TreeNode buildTree(int[] preorder, int[] inorder) {

if (preorder == null || inorder == null

|| preorder.length == 0 || inorder.length == 0) return null;

return subBuildTree(0, 0, inorder.length - 1, preorder, inorder);

}

public TreeNode subBuildTree(int prestart, int instart, int inend, int[] preorder, int[] inorder) {

if (prestart > preorder.length - 1 || instart > inend) return null;

TreeNode root = new TreeNode(preorder[prestart]);

*// find the key point that divide left and right parts*

int key = instart;

while (inorder[key] != root.val) key++;

root.left = subBuildTree(prestart + 1, instart, key - 1, preorder, inorder);

*// the length of left part: (key-instart), so start of right part in preorder tree:*

*// prestart + (key-instart) + 1*

root.right = subBuildTree(prestart + key - instart + 1, key + 1, inend, preorder, inorder);

return root;

}

106. Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

**S1. Recursion**

和上道题的思路是相似的，但是preorder->postorder, 判断的不是第一个点，而是最后一个点，

并且是利用right-subtree的长度来判断inorder的断点

public TreeNode buildTree(int[] inorder, int[] postorder) {

return helper(inorder, postorder, postorder.length - 1, 0, inorder.length - 1);

}

private TreeNode helper(int[] inorder, int[] postorder, int postEnd, int inStart, int inEnd) {

if (postEnd < 0 || inStart > inEnd) return null;

TreeNode root = new TreeNode(postorder[postEnd]);

int rootPos = inStart;

while (inorder[rootPos] != root.val) rootPos++;

int rightSubTreeLen = inEnd - rootPos;

root.left = helper(inorder, postorder, postEnd - rightSubTreeLen - 1, inStart, rootPos-1);

root.right = helper(inorder, postorder, postEnd - 1, rootPos + 1, inEnd);

return root;

}

**Level Order Traversal**

102. Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

**S1.用queue来做**

每次把一个level的node全部放进去，并且需要记录

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null) return res;

Queue<TreeNode> q = new LinkedList<>();

q.offer(root);

while (!q.isEmpty()) {

int size = q.size();

List<Integer> list = new ArrayList<>();

for (int i = 0;i < size;i++) {

TreeNode cur = q.poll();

list.add(cur.val);

if (cur.left != null) q.offer(cur.left);

if (cur.right != null) q.offer(cur.right);

}

res.add(list);

}

return res;

}

107 太简单了，直接不用写，就是把加入的那一行改成res.add(0, list)

Easy - 515. Find Largest Value in Each Tree Row

You need to find the largest value in each row of a binary tree.

**S1. Queue**

public List<Integer> largestValues(TreeNode root) {

List<Integer> res = new ArrayList<>();

if (root == null) return res;

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()) {

int size = queue.size();

int max = Integer.MIN\_VALUE;

for (int i = 0;i < size;i++) {

TreeNode cur = queue.poll();

max = Math.max(max, cur.val);

if (cur.left != null) queue.offer(cur.left);

if (cur.right != null) queue.offer(cur.right);

}

res.add(max);

}

return res;

}

637 和上面一题方法一样，加起来求平均即可。

513 同理

404. Sum of Left Leaves

Find the sum of all left leaves in a given binary tree.

**S1. Recursion**

注意这道题要求的是sum of Left leaves, 只有左叶子要加进去，right leaves不加。

所以需要在一开始判定，如果是leaf也返回0. 因为left leaf会在之前已经判断好。

public int sumOfLeftLeaves(TreeNode root) {

if (root == null || (root.left == null && root.right == null)) return 0;

int sum = 0;

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

sum += root.left.val;

} else {

sum += sumOfLeftLeaves(root.left);

}

sum += sumOfLeftLeaves(root.right);

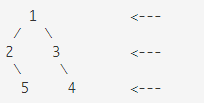
return sum;

}

199. 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.

For example: Given the following binary tree, You should return [1, 3, 4].



**S1. Level-order traversal**

每个level从右往左，并且加入最右侧的值即可。

public List<Integer> rightSideView(TreeNode root) {

List<Integer> res = new ArrayList<>();

if (root == null) return res;

Queue<TreeNode> queue = new LinkedList<>();

queue.offer(root);

while (!queue.isEmpty()) {

int size = queue.size();

res.add(queue.peek().val);

for (int i = 0;i < size;i++) {

TreeNode cur = queue.poll();

if (cur.right != null) queue.offer(cur.right);

if (cur.left != null) queue.offer(cur.left);

}

}

return res;

}

**S2. Recursion**

根据目前的depth来决定是否将当前值加入，root-right-left

public List<Integer> rightSideView(TreeNode root) {

List<Integer> list = new ArrayList<>();

if (root == null) return list;

rightView(list, root, 0);

return list;

}

public void rightView(List<Integer> list, TreeNode root, int curDepth) {

if (root == null) return;

if (curDepth == list.size())

list.add(root.val);

rightView(list, root.right, curDepth + 1);

rightView(list, root.left, curDepth + 1);

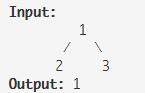
}

563. Binary Tree Tilt

Given a binary tree, return the tilt of the whole tree.

The tilt of a tree node is defined as the absolute **difference between the sum of all left subtree node values and the sum of all right subtree node values.** Null node has tilt 0.

The tilt of the whole tree is defined as the sum of all nodes' tilt.



Tilt of node 2 : 0，Tilt of node 3 : 0，Tilt of node 1 : |2-3| = 1

Tilt of binary tree : 0 + 0 + 1 = 1

**S1. Recursion**

int res = 0;

public int findTilt(TreeNode root) {

helper(root);

return res;

}

*// get the sum of subtree of root.*

private int helper(TreeNode root) {

if (root == null) return 0;

int left = helper(root.left);

int right = helper(root.right);

res += Math.abs(left - right);

return root.val + left + right;

}

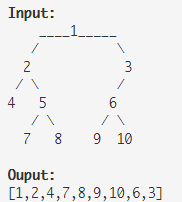
545. Boundary of Binary Tree

Given a binary tree, return the values of its boundary in anti-clockwise direction starting from root. Boundary includes left boundary, leaves, and right boundary in order without duplicate nodes.

Left boundary is defined as the path from root to the left-most node. Right boundary is defined as the path from root to the right-most node. If the root doesn't have left subtree or right subtree, then the root itself is left boundary or right boundary. Note this definition only applies to the input binary tree, and not applies to any subtrees.

The left-most node is defined as a leaf node you could reach when you always firstly travel to the left subtree if exists. If not, travel to the right subtree. Repeat until you reach a leaf node.

The right-most node is also defined by the same way with left and right exchanged.



**S1：**

这道题值得好好看一下，是前面几道find Leaves, find left leaves, find left-most of each level的结合版，而且需要考虑顺序，

注意是逆时针的顺序，并且从root开始，意味着root.val是在最前面加进去的，并且leaf不能直接调用root, 而需要调用root.left, root.right

并且，由于需要是逆时针，所以leftBound, rightBound中调用recursion和加入res list的顺序相反。

public List<Integer> boundaryOfBinaryTree(TreeNode root) {

List<Integer> res = new ArrayList<>();

if (root == null) return res;

res.add(root.val);

leftBound(root.left, res);

leaf(root.left, res);

leaf(root.right, res);

rightBound(root.right, res);

return res;

}

private void leftBound(TreeNode root, List<Integer> res) {

if (root == null || (root.left == null && root.right == null)) return;

res.add(root.val);

if (root.left != null) leftBound(root.left, res);

else leftBound(root.right, res);

}

private void rightBound(TreeNode root, List<Integer> res) {

if (root == null || (root.left == null && root.right == null)) return;

if (root.right == null) rightBound(root.left, res);

else rightBound(root.right, res);

res.add(root.val);

}

private void leaf(TreeNode root, List<Integer> res) {

if (root == null) return;

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

res.add(root.val);

return;

}

leaf(root.left, res);

leaf(root.right, res);

}

**S2. level order traversal**

觉得可以这样写，还没有测试：每一行的最左边，最后边分别放进res，

最后一行除了最左，最右放进一个res，然后拼在一起

103. Binary Tree Zigzag Level Order Traversal

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).



return its zigzag level order traversal as: [3], [20,9], [15,7]

**S1.Queue**

用一个boolean来记录是从左向右还是从右向左，每个level翻转一次。

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

if (root == null) return res;

boolean LtoR = true;

Queue<TreeNode> q = new LinkedList<>();

q.add(root);

while (!q.isEmpty()) {

int size = q.size();

List<Integer> list = new ArrayList<>();

for (int i = 0;i < size;i++) {

TreeNode cur = q.poll();

if (LtoR) list.add(cur.val);

else list.add(0, cur.val);

if (cur.left != null) q.add(cur.left);

if (cur.right != null) q.add(cur.right);

}

LtoR = !LtoR;

res.add(list);

}

return res;

}

Tree的小分类 - Nested Integer

Lintcode. Flatten List

Given a list, each element in the list can be a list or integer. flatten it into a simply list with integers.

**S1. Recursion**

public List<Integer> flatten(List<NestedInteger> nestedList) {

List<Integer> res = new ArrayList<>();

dfs(nestedList, res);

return res;

}

private void dfs(List<NestedInteger> nestedList, List<Integer> res) {

for (NestedInteger item:nestedList) {

if (item.isInteger()) {

res.add(item.getInteger());

} else {

dfs(item.getList(), res);

}

}

}

339. Nested List Weight Sum

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Example 1: Given the list [[1,1],2,[1,1]], return 10. (four 1's at depth 2, one 2 at depth 1)

**S1. Recursion**

如果是数字，直接加；

如果是list，继续调用helper，并且depth+1

public int depthSum(List<NestedInteger> nestedList) {

return helper(nestedList, 1);

}

private int helper(List<NestedInteger> nestedList, int depth) {

if (nestedList == null || nestedList.size() == 0) {

return 0;

}

int sum = 0;

for (NestedInteger ni : nestedList) {

if (ni.isInteger()) {

sum += ni.getInteger() \* depth;

} else {

sum += helper(ni.getList(), depth + 1);

}

}

return sum;

}

364. Nested List Weight Sum II

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1: Given the list [[1,1],2,[1,1]], return 8. (four 1's at depth 1, one 2 at depth 2)

Example 2:

Given the list [1,[4,[6]]], return 17. (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; 1\*3 + 4\*2 + 6\*1 = 17)

和之前的区别是depth的算法反过来。

**S1. Naive**

先遍历一遍，看maxDepth，然后减就好了

public int depthSumInverse(List<NestedInteger> nestedList) {

*// First we find the depth of the List*

*// Then backward*

if (nestedList == null || nestedList.size() == 0) {

return 0;

}

int depth = findDepth(nestedList);

return helper(nestedList, depth);

}

private int findDepth(List<NestedInteger> nestedList) {

int depth = 1;

for (NestedInteger n : nestedList) {

if (!n.isInteger()) {

depth = Math.max(depth, findDepth(n.getList()) + 1);

}

}

return depth;

}

private int helper(List<NestedInteger> nestedList, int depth) {

int sum = 0;

if (nestedList == null || nestedList.size() == 0) {

return 0;

}

for (NestedInteger n : nestedList) {

if (n.isInteger()) {

sum += n.getInteger() \* depth;

} else {

sum += helper(n.getList(), depth - 1);

}

}

return sum;

}

**S2. 优化**

通过传递curSum的方法，让在外层的数字多加。

*// [1, [2]]*

public int depthSumInverse(List<NestedInteger> nestedList) {

*// the key idea here is to pass the integer sum of this layer*

*// to the next layer so that it would be added every time when*

*// we go to the next layer, just like we multiply its depth.*

return dfs(nestedList, 0);

}

private int dfs(List<NestedInteger> nestedList, int currSum) {

List<NestedInteger> nextLevel = new ArrayList<NestedInteger>();

for (NestedInteger ni : nestedList) {

if (ni.isInteger()) {

currSum += ni.getInteger();

} else {

nextLevel.addAll(ni.getList());

}

}

// 重点在这里：这里的curSum其实已经加过外层的1了，但是带到dfs里面又计算了一次，

// 于是加了两遍，相当于反着算。

int sum = nextLevel.isEmpty() ? 0 : dfs(nextLevel, currSum);

return sum + currSum;

}

341. Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list -- whose elements may also be integers or other lists.

Given the list [[1,1],2,[1,1]],

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: [1,1,2,1,1].

**S1. Recursion**

如果以后只是每次调用的时候，取下一个数字，

那么最好在一开始的时候就把nestedList里的所有数字都放到list中，用index来保存目前访问到的index

public class NestedIterator implements Iterator<Integer> {

List<Integer> list;

int index; *// current index points to.*

public NestedIterator(List<NestedInteger> nestedList) {

if (nestedList == null) return;

index = 0;

list = new ArrayList<>();

putToList(nestedList);

}

*// add all the numbers in nestedList to list.*

private void putToList(List<NestedInteger> nestedList) {

for (NestedInteger i:nestedList) {

if (i.isInteger()) {

list.add(i.getInteger());

} else {

putToList(i.getList());

}

}

}

@Override

public Integer next() {

if (!hasNext()) return null;

return list.get(index++);

}

@Override

public boolean hasNext() {

return index < list.size();

}

}

Path，Path Sum – 通常用backtracking

涉及的通常也是backtracking的三类问题：

1. 求有没有解
2. 求所有解
3. 求最优解

Easy - 112. Path Sum

Given a binary tree and a sum, determine **if** the tree has a **root-to-leaf** path such that adding up all the values along the path equals the given sum.

S1 Recursion

public boolean hasPathSum(TreeNode root, int sum) {

if (root == null) return false;

return backtrack(root, sum);

}

boolean backtrack(TreeNode root, int sum) {

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

return sum == root.val;

}

boolean left = false, right = false;

if (root.left != null) left = backtrack(root.left, sum - root.val);

if (root.right != null) right = backtrack(root.right, sum - root.val);

return left || right;

}

113. Path Sum II

Given a binary tree and a sum, find all **root-to-leaf** paths where each path's sum equals the given sum.

返回路径

**题解：**

和上一道题相比，就在helper function中多加一个参数，表示走到现在经过的路径。

public List<List<Integer>> pathSum(TreeNode root, int sum) {

List<List<Integer>> list = new ArrayList<>();

if (root == null) return list;

List<Integer> curPath = new ArrayList<>();

curPath.add(root.val);

helper(list, curPath, root, sum - root.val);

return list;

}

private void helper(List<List<Integer>> list, List<Integer> path, TreeNode root, int sum) {

if (sum == 0 && root.left == null && root.right == null) {

list.add(new ArrayList<>(path));

return;

}

if (root.left != null) {

path.add(root.left.val);

helper(list, path, root.left, sum - root.left.val);

path.remove(path.size() - 1);

}

if (root.right != null) {

path.add(root.right.val);

helper(list, path, root.right, sum - root.right.val);

path.remove(path.size() - 1);

}

}

Easy- 129. Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers. For example,

1

/ \

2 3

The root-to-leaf path 1->2 represents the number 12. The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

**S1 backtrack**

非常简单的backtrack，判断条件，继续往下走，在走到leaf的时候把cur加进sum即可。

public int sumNumbers(TreeNode root) {

if (root == null) return 0;

int[] sum = new int[]{0};

backtrack(root, sum, 0);

return sum[0];

}

void backtrack(TreeNode root, int[] sum, int cur) {

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

sum[0] += cur \* 10 + root.val;

return;

}

if (root.left != null) backtrack(root.left, sum, cur\*10+ root.val);

if (root.right != null) backtrack(root.right, sum, cur\*10 + root.val);

}

Pocket Gems: Max sum path from root to leaf

,给一棵二叉树的root，求从root到leaf的sum最大的path，并且输出路径

思路：

1. 第一遍往下走，找sum最大的路径，把<kid, parent>关系存进map，并保存满足max path sum 的leaf node
2. 第二遍从leaf node往上走，根据map中的关系一直走到root

*// max sum path*

static TreeNode maxLeaf = null; *// store the leaf node of max path*

public static String maxSumPath(TreeNode root) {

if (root == null) return "";

Map<TreeNode, TreeNode> map = new HashMap<>(); *// <kid, parent>*

*// 1.find the max sum path, and store the path in map*

getMaxPath(null, root, new int[]{Integer.MIN\_VALUE}, 0, map);

*// 2.print out the path*

return printPath(root, map);

}

static void getMaxPath(TreeNode prev, TreeNode root, int[] maxSum, int curSum,

Map<TreeNode, TreeNode> map) {

if (root == null) return;

curSum += root.val;

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

if (curSum > maxSum[0]) {

maxSum[0] = curSum;

maxLeaf = root;

}

}

map.put(root, prev);

getMaxPath(root, root.left, maxSum, curSum, map);

getMaxPath(root, root.right, maxSum, curSum, map);

}

static String printPath(TreeNode root, Map<TreeNode, TreeNode> map) {

String res = "";

while (maxLeaf != null) {

res = maxLeaf.val + "->" + res;

maxLeaf = map.get(maxLeaf);

}

return res.substring(0, res.length() - 2);

}

124. Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node and does not need to go through the root.

重点是：只需要找一个maximum path sum, 并且path不需要经过root

**S1. backtrack**

通过全局变量来更新sum值，

return这一点往下left, right两边的最优解。

public class Solution {

int sum = Integer.MIN\_VALUE;

public int maxPathSum(TreeNode root) {

if (root == null) return 0;

subMaxPathSum(root);

return sum;

}

*// find the path that has highest sum*

public int subMaxPathSum(TreeNode node) {

if(node == null) return 0;

int left = Math.max(0, subMaxPathSum(node.left));

int right = Math.max(0, subMaxPathSum(node.right));

// possible not pass root, compare with the path here:

sum = Math.max(sum, left + right + node.val);

return Math.max(left, right) + node.val; // return only one path, up or down

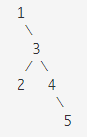
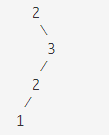
}

}

298. Binary Tree Longest Consecutive Sequence

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

First, Longest consecutive sequence path is 3-4-5, so return 3.

Second, Longest consecutive sequence path is 2-3,not3-2-1, so return 2.

**S1 Recursion**

方法其实很简单，用recursion就可以，其实也相当于DFS, 一路走，通过target判断当前点能不能和之前的连上，

区别就在于longest consecutive path不需要从root开始，而是从任意一点开始都可以，

所以在target连不上，不能满足条件的情况下，不是回头，而是重新设为1，继续往下走

public int longestConsecutive(TreeNode root) {

if (root == null) return 0;

int[] max = new int[]{1};

helper(root, 0, root.val, max);

return max[0];

}

*// cur: current valid longest consecutive path length.*

*// target: the next number searching for to remain consecutive.*

public void helper(TreeNode root, int cur, int target, int[] max) {

if (root == null) return;

if (root.val == target) {

cur++;

max[0] = Math.max(max[0], cur);

} else {

cur = 1;

}

helper(root.left, cur, root.val + 1, max);

helper(root.right, cur, root.val + 1, max);

}

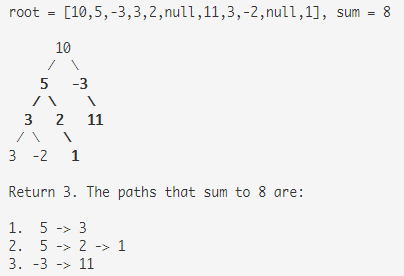
437. Path Sum III

You are given a binary tree in which each node contains an integer value.

Find **the number of paths that sum to a given value.**

The path **does not need to start or end at the root or a leaf, but it must go downwards** (traveling only from parent nodes to child nodes).

The tree has no more than 1,000 nodes and the values are in the range -1,000,000 to 1,000,000.



**S1 Backtrack**

对于每个node，都要考虑contains / not contain两种情况

O(n^2) if bias, O(nlogn) if complete tree.

public int pathSum(TreeNode root, int sum) {

if (root == null) return 0;

// path contains root | path on left/right which don't contain root

return helper(root, sum)

+ pathSum(root.left, sum) + pathSum(root.right, sum);

}

// Get the # of paths sum is “sum”, and go through root.

private int helper(TreeNode root, int sum) {

if (root == null) return 0;

int res = 0;

if (sum == root.val) res++;

res += helper(root.left, sum - root.val);

res += helper(root.right, sum - root.val);

return res;

}

**S2. 利用prefix\_sum来做memorized search**

到某个node的cur是从root一路加下来的值，可能>= sum,

用map来记录走到node的每个sum有多少种可能性，比如1, 2, -1, 1, ... sum==2有两种可能性，

所以map.get(cur - sum)就是到这个node取得sum的可能性

*// Map: <prefix\_sum, # of ways to get prefix\_sum>*

public int pathSum(TreeNode root, int sum) {

Map<Integer, Integer> map = new HashMap<>();

map.put(0, 1);

return helper(root, 0, sum, map);

}

*/\*\**

*\* root.val not included when calling helper*

*\* cur: from root, level down, 每一层都必然会加一个node.val, 所以取值可能>=sum*

*\*/*

private int helper(TreeNode root, int cur, int sum, Map<Integer, Integer> map) {

if (root == null) return 0;

cur += root.val;

int res = map.getOrDefault(cur - sum, 0);

map.put(cur, map.getOrDefault(cur, 0) + 1); *// update cur*

res += helper(root.left, cur, sum, map) + helper(root.right, cur, sum, map);

map.put(cur, map.get(cur) - 1); *// this node not included,won't affect other calculation*

return res;

}

树的结构 – 通常用recursion

100. Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

题解：

1.RECURSION, check root, then left-subtree, right-subtree

2.using 2 queue, traverse through all nodes.

572. Subtree of Another Tree

Given two non-empty binary trees s and t, check whether tree t has exactly the same structure and node values with a subtree of s. A subtree of s is a tree consists of a node in s and all of this node's descendants. The tree scould also be considered as a subtree of itself.

题解：这道题其实就是上一道的加强版，

查看t是不是和s的某个子树是same tree，也就是对每个node.val == t.val的子树做判断

public boolean isSubtree(TreeNode s, TreeNode t) {

if (s == null) return false;

if (isSameTree(s, t)) return true;

return isSameTree(s.left, t) || isSameTree(s.right, t);

}

101. Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree [1,2,2,3,4,4,3] is symmetric.

题解：

写一个helper function，left/right交换检查一下就可以了

left-subtree 的左子树应该个right-subtree的右子树一样

public boolean isSymmetric(TreeNode root) {

if (root == null) return true;

return subSymmetric(root.left, root.right);

}

public boolean subSymmetric(TreeNode a, TreeNode b) {

if (a == null || b == null) {

return a == b;

}

if (a.val != b.val) return false;

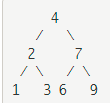
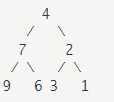
*// pay attention here!*

return subSymmetric(a.left, b.right) && subSymmetric(a.right, b.left);

}

226. Invert Binary Tree

Invert a binary tree.

From:  to 

题解

非常明显要用recursion来做，先invert左子树，右子树，再在root调换左右子树

public TreeNode invertTree(TreeNode root) {

if (root == null return null;

TreeNode left = null, right = null;

if (root.left != null) {

left = invertTree(root.left);

}

if (root.right != null) {

right = invertTree(root.right);

}

root.left = right;

root.right = left;

return root;

}

Easy - 104. Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

public int maxDepth(TreeNode root) {

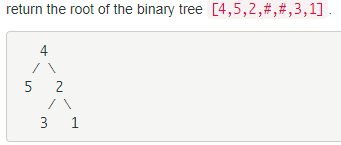
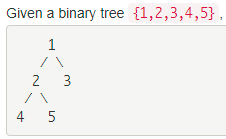
if (root == null) return 0;

return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));

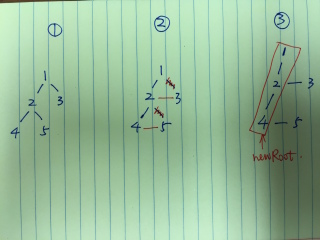
}

156. Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.



其实具体要做的就是：



**S1 Recursion** time O(n), space O(n)

find the leftmost node as the root.

Return repoint each new parent - root.left to previous root and root.right;

public TreeNode upsideDownBinaryTree(TreeNode root) {

if (root == null || root.left == null) return root;

TreeNode newRoot = upsideDownBinaryTree(root.left);

*// now all the node has been processed,*

*// except the relation between root, root.left, root.right*

root.left.left = root.right;

root.left.right = root;

root.left = null;

root.right = null;

return newRoot;

}

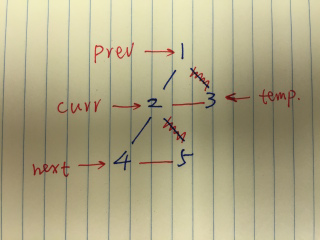
**S2. Iteration time O(n), space O(1)**

每次处理一个node，调换指向，

pre: previous point after repoint, 上一个已经转换好了的node

并且使用tmp来track right node of previous root

使用next来track下一个需要process的left node,



public TreeNode upsideDownBinaryTree(TreeNode root) {

TreeNode cur = root;

TreeNode pre = null;

TreeNode tmp = null;

TreeNode next = null;

while(cur != null){

next = cur.left;

*// update cur's left/right-subtree*

*// tmp: to keep the previous right child*

cur.left = tmp;

tmp = cur.right;

cur.right = pre;

*// set for next iteration:*

pre = cur;

cur = next;

}

return pre;

}

子树结构 – 根据子树状态决定parent

543. Diameter of Binary Tree

Given a binary tree, you need to compute the length of the diameter of the tree. The diameter of a binary tree is the **length of the longest path between any two nodes** in a tree. This path may or may not pass through the root.

题解：

非常显然需要用recursion来做，但有一点特别：recursion返回当前最长的path，

并且同时需要更新max

public int diameterOfBinaryTree(TreeNode root) {

int[] max = new int[]{0};

maxDepth(root, max);

return max[0];

}

*// update max: max is the path of both sides,*

*// return single: the longer path of left/right side.*

int maxDepth(TreeNode root, int[] max) {

if (root == null) return 0;

int left = maxDepth(root.left, max);

int right = maxDepth(root.right, max);

max[0] = Math.max(max[0], left + right);

return Math.max(left, right) + 1;

}

508.Most Frequent Subtree Sum

Given the root of a tree, you are asked to find the most frequent subtree sum.

The subtree sum of a node is defined as the sum of all the node values **formed by the subtree rooted at that node** (including the node itself). So what is the most frequent subtree sum value? If there is a tie, return all the values with the highest frequency in any order.

eg: tree is 5-2-3, return [2, -3, 4] since all values happen only once.

题解：

这道题和上一道非常相似，在每个recursion中更新global max，并且return sum

int max = 0; *// global max count, keep update*

public int[] findFrequentTreeSum(TreeNode root) {

Map<Integer, Integer> map = new HashMap<>(); // <sum, frequency>

helper(root, map);

// check how many numbers fit max.

List<Integer> list = new ArrayList<>();

for (Integer key:map.keySet()) {

if (map.get(key) == max) {

list.add(key);

}

}

int[] out = new int[list.size()]; // convert list into array.

for (int i = 0;i < list.size();i++) {

out[i] = list.get(i);

}

return out;

}

// get the frequency of each sum

private int helper(TreeNode root, Map<Integer, Integer> map) {

if (root == null) return 0;

int left = helper(root.left, map);

int right = helper(root.right, map);

int sum = left + right + root.val;

int count = 1 + map.getOrDefault(sum, 0);

map.put(sum, count);

max = Math.max(max, count); // update global

return sum;

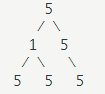
}

250. Count Univalue Subtrees

Given a binary tree, **count the number of uni-value subtrees**.就是某个subtree里只包含一个数字。

A Uni-value subtree means all nodes of the subtree have the same value.

For example: Given binary tree, return 4.



题解：

这道题parent的状态需要通过kids的状态来判断，从leaf node一层层往上走，

所以kid需要保存：本身是不是univalue subtree的信息+它自己和它的sub-tree中一共有多少univalue subtrees；

这样可以一层一层count = left.count + right.count + 1, 进行传递。

通过建立新的class来实现

class Result {

int count; // # of univalue subtree in its subtree

Integer val; // 当前node的值

boolean same; // whether this is a univalue subtree

Result(int count, Integer val, boolean same) {

this.count = count;

this.val = val;

this.same = same;

}

}

public int countUnivalSubtrees(TreeNode root) {

return helper(root).count;

}

Result helper(TreeNode root) {

if (root == null) return new Result(0, null, true);

Result left = helper(root.left);

Result right = helper(root.right);

if ((left.val != null && !left.val.equals(root.val)) ||

right.val != null && !right.val.equals(root.val) ||

left.same == false || right.same == false) {

return new Result(left.count + right.count, root.val, false);

} else { // 以这里的root为root的subtree是uni-value subtree

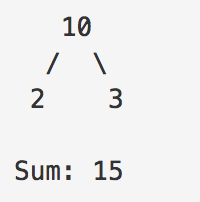
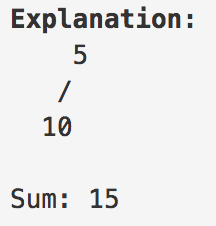
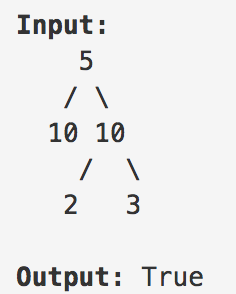
return new Result(left.count + right.count + 1, root.val, true);

}

}

663. Equal Tree Partition

Given a binary tree with n nodes, your task is to check if it's possible to partition the tree to two trees which have the equal sum of values after removing exactly one edge on the original tree.



思路：

遍历，并且记录一个map，包含的信息是<node, 以该node为root的subtree的sum>

并且按照每个node进行遍历，寻找有没有sum of this subtree = sum of all nodes – sub of this subtree.

如果有满足这种情况，就存在，为true

333. Largest BST Subtree

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.



The Largest BST Subtree in this case is the highlighted one. The return value is the subtree's size, which is 3.

Do this in O(n) time.

**S1. Recursion**

这道题处理的重点，就是先从子树开始处理，在确认left-subtree, right-subtree都是BST,

并且和current node满足BST 的关系后（大于左子树最大值，小于右子树最小值），再处理current node.

用class Result, 来记录某个subtree的最大值，最小值，和size。如果不是BST, 则size设为-1来辨别。

class Result {

int size; *// size of the BST whose root is this node.*

int lower; *// min value in current node and its subtree*

int upper; *// max value in current node ...*

Result(int size, int lower, int upper) {

this.size = size;

this.lower = lower;

this.upper = upper;

}

}

int max = 0;

public int largestBSTSubtree(TreeNode root) {

if (root == null) return 0;

helper(root);

return max;

}

private Result helper(TreeNode root) {

if (root == null) return new Result(0, Integer.MAX\_VALUE, Integer.MIN\_VALUE);

Result left = helper(root.left);

Result right = helper(root.right);

*// conditions: not a BST:*

if (left.size == -1 || right.size == -1

|| left.upper >= root.val || root.val >= right.lower) {

return new Result(-1, 0, 0);

}

*// this node, left-subtree, right-subtree is a BST:*

int size = left.size + right.size + 1;

*// update global max.*

max = Math.max(max, size);

return new Result(size, Math.min(root.val, left.lower), Math.max(root.val, right.upper));

}

366. Find Leaves of Binary Tree

Given a binary tree, collect a tree's nodes as if you were doing this:

Collect and remove all leaves, repeat until the tree is empty. Given the following tree, Returns [4, 5, 3], [2], [1].



**S1. Recursion**

这道题也是从底向上，并且level也是由下向上来决定，通过选择left-side, right-side中max的一点来做到。

然后把node的数值放到对应level的位置中。

public List<List<Integer>> findLeaves(TreeNode root) {

List<List<Integer>> res = new ArrayList<>();

getLevel(root, res);

return res;

}

*// level start from 0. Level of leaf node is 0.*

int getLevel(TreeNode root, List<List<Integer>> res) {

if (root == null) return -1;

*// level of leaf node: 0,*

*// and is decided by the deeper depth of left/right side.*

// 在这里，确定当前的TreeNode root所在的level，并放入对应的list:

int level = 1 + Math.max(getLevel(root.left, res), getLevel(root.right, res));

if (res.size() <= level) {

res.add(new ArrayList<>());

}

res.get(level).add(root.val);

return level;

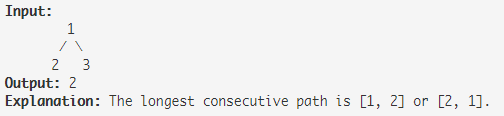
}

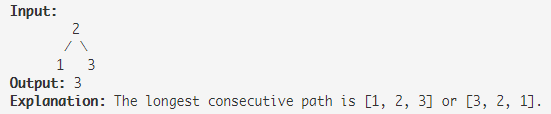
549. Binary Tree Longest Consecutive Sequence II

Given a binary tree, you need to find the length of Longest Consecutive Path in Binary Tree.

Especially, this path can be either increasing or decreasing. For example, [1,2,3,4] and [4,3,2,1] are both considered valid, but the path [1,2,4,3] is not valid.

On the other hand, the path **can be in the child-Parent-child order, where not necessarily be parent-child** order.





**S1 Recursion**

走到每个点的时候，判断当前点的decrease和increase并返回。

并且更新global max.

int maxval = 0;

public int longestConsecutive(TreeNode root) {

longestPath(root);

return maxval;

}

*// return {increase, decrease}*

public int[] longestPath(TreeNode root) {

if (root == null) return new int[] {0,0};

int inr = 1, dcr = 1;

if (root.left != null) {

int[] l = longestPath(root.left);

if (root.val == root.left.val + 1) dcr = l[1] + 1;

else if (root.val == root.left.val - 1) inr = l[0] + 1;

}

if (root.right != null) {

int[] r = longestPath(root.right);

if (root.val == root.right.val + 1) dcr = Math.max(dcr, r[1] + 1);

else if (root.val == root.right.val - 1) inr = Math.max(inr, r[0] + 1);

}

maxval = Math.max(maxval, dcr + inr - 1);

return new int[] {inr, dcr};

}

**Height/Depth**

110. Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

**S1 Recursion**

根据tree的depth来判断即可；

public boolean isBalanced(TreeNode root) {

if (root == null) return true;

int left = getDepth(root.left);

int right = getDepth(root.right);

if (Math.abs(left - right) > 1) return false;

return isBalanced(root.left) && isBalanced(root.right);

}

int getDepth(TreeNode node) {

if (node == null) return 0;

return Math.max(getDepth(node.left), getDepth(node.right)) + 1;

}

111. Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

**S1. Recursion**

public int minDepth(TreeNode root) {

if (root == null) return 0;

int left = minDepth(root.left);

int right = minDepth(root.right);

return (left != 0 && right != 0) ? 1 + Math.min(left, right) : 1 + Math.max(left, right);

}

**S2. Level-order traversal**

一层一层走，每走到新的一层，更新count，

第一次遇到leaf的时候return count.

222. Count Complete Tree Nodes

Given a complete binary tree, count the number of nodes.

题解：

complete tree，至少有一边应该是全满的 full tree，

对于一棵full tree来说，# of nodes = 2^k – 1. 假设height = 3, 那么 # of nodes = 7.

所以假设左边全满，left-subtree的# of nodes = 2^(height of left-subtree) – 1

public int countNodes(TreeNode root) {

if (root == null) return 0;

int l = getHeight(root.left);

int r = getHeight(root.right);

if (l == r) { *// left is full*

return (1 << l) + countNodes(root.right); // 1 << l 相当于 Math.pow(2,l-1)

} else { *// right subTree is full*

return countNodes(root.left) + (1 << r);

}

}

int getHeight(TreeNode root) {

int h = 0;

while (root != null) {

h++;

root = root.left;

}

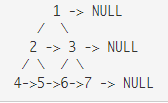
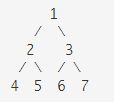
return h;

}

116. Populating Next Right Pointers in Each Node

Given a binary tree，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.



**S1. Recursion**

用recursion来做，每次先处理当前的next pointer, 然后调用next level

public void connect(TreeLinkNode root) {

if (root == null || (root.left == null && root.right == null)) return;

helper(root.left, root.right);

}

private void helper(TreeLinkNode p, TreeLinkNode q) {

if (p == null || q == null) return;

p.next = q;

q.next = null;

helper(p.left, p.right);

helper(p.right, q.left);

helper(q.left, q.right);

}

**S2. Iteration**

用level order traversal, 用prev, cur两个node即可

public void connect(TreeLinkNode root) {

if (root == null || (root.left == null && root.right == null)) return;

Queue<TreeLinkNode> q = new LinkedList<>();

TreeLinkNode prev = null;

TreeLinkNode cur = null;

q.add(root);

while (!q.isEmpty()) {

prev = null;

int size = q.size();

for (int i = 0;i < size;i++) {

cur = q.poll();

if (prev != null) prev.next = cur;

prev = cur;

if (cur.left != null) q.add(cur.left);

if (cur.right != null) q.add(cur.right);

}

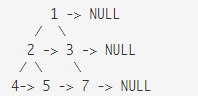
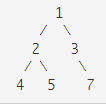
}

}

117. Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?



**S1. Iteration**

其实和上面那道题一模一样 – 上面的解法就可以支持这里的情况。

只是换了中写法，只用一个cur node，然后用queue.peek即可

public void connect(TreeLinkNode root) {

if (root == null) return;

Queue<TreeLinkNode> q = new LinkedList<>();

q.add(root);

while (!q.isEmpty()) {

int size = q.size();

for (int i = 0;i < size;i++) {

TreeLinkNode cur = q.poll();

if (!q.isEmpty() && i < size - 1) cur.next = q.peek();

if (cur.left != null) q.add(cur.left);

if (cur.right != null) q.add(cur.right);

}

}

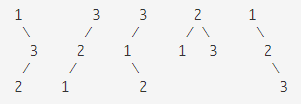
}

**BST性质**

95. Unique Binary Search Trees II

Given an integer n**, generate all structurally unique BST's** (binary search trees) that store values 1...n.

For example, Given n = 3, your program should return all 5 unique BST's shown below.



**S1. Recursion**

用helper function, 来建立[start, end] 范围的tree，root可以是[start, end],

并且得到left subtree, right subtree, 左子树，右子树得到后，root和左子树，右子树的哪个节点相连又有各种可能性，通过两个内部的for循环实现；

corner case：当i == start, end时，其中又一边会是null, 需要把”null”作为一个节点加入list, 这样就不需要处理left, right == null的情况

public List<TreeNode> generateTrees(int n) {

return n > 0 ? build(1, n) : new ArrayList<>();

}

*// build the tree of [start, end]*

private List<TreeNode> build(int start, int end) {

List<TreeNode> res = new ArrayList<>();

*// root of the tree is i:*

for (int i = start;i <= end;i++) {

List<TreeNode> left = build(start, i - 1); *// left sub-tree*

List<TreeNode> right = build(i + 1, end); *// right sub-tree*

for (int m = 0;m < left.size();m++) {

for (int n = 0;n < right.size();n++) {

TreeNode root = new TreeNode(i);

root.left = left.get(m);

root.right = right.get(n);

res.add(root);

}

}

}

if (res.size() == 0) res.add(null);

return res;

}

501. Find Mode in Binary Search Tree

Given a binary search tree (BST) with duplicates, find all the mode(s) (the most frequently occurred element) in the given BST.

Assume a BST is defined as follows:

The left subtree of a node contains only nodes with keys less than or equal to the node's key.

The right subtree of a node contains only nodes with keys greater than or equal to the node's key.

Both the left and right subtrees must also be binary search trees.

return [2].

108. Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

**S1. Recursion**

非常典型的recursion

public TreeNode sortedArrayToBST(int[] nums) {

if (nums == null || nums.length == 0) return null;

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

}

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

if (start > end) return null;

int mid = (start + end) / 2;

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

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

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

return root;

}

450. Delete Node in a BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

-Search for a node to remove.

-If the node is found, delete the node.

Note: Time complexity should be O(height of tree).

**S1 Recursion**

先通过recursion转移到特定位置，再开始delete root，每次处理的时候都按照delete root处理，分为2种情况：

1. only has left sub-tree or only have right sub-tree, 返回另一侧的root.right/left即可
2. both left, right exist, 这里有两种处理方式，把右子树最小值移到root，或者把左子树最大值移到root，采用的是第一种
3. 并且记得把右子树最小值在右子树中删掉，相当于再调用一次deleteNode.

public TreeNode deleteNode(TreeNode root, int key) {

if (root == null) return null;

if (root.val > key) {

root.left = deleteNode(root.left, key);

} else if (root.val < key) {

root.right = deleteNode(root.right, key);

} else {

if (root.left == null) return root.right;

if (root.right == null) return root.left;

**TreeNode rightMin = findMin(root.right);**

root.val = rightMin.val;

root.right = deleteNode(root.right, rightMin.val);

}

return root;

}

private TreeNode findMin(TreeNode node) {

while (node.left != null) node = node.left;

return node;

}

**S2 Iteration**

下次看到再分析

public TreeNode deleteNode(TreeNode root, int key) {

if (root == null) return null;

TreeNode cur = root;

TreeNode prev = null;

while (cur != null && cur.val != key) {

prev = cur;

if (key < cur.val) cur = cur.left;

else if (key > cur.val) cur = cur.right;

}

if (prev == null) return deleteRoot(root);

else if (prev.left == cur) {

prev.left = deleteRoot(cur);

} else {

prev.right = deleteRoot(cur);

}

return root;

}

private TreeNode deleteRoot(TreeNode root) {

if (root == null) return null;

if (root.left == null) return root.right;

if (root.right == null) return root.left;

TreeNode rightMin = root.right;

TreeNode prev = null;

while (rightMin.left != null) {

prev = rightMin;

rightMin = rightMin.left;

}

rightMin.left = root.left;

if (root.right != rightMin) { *// prev == null*

*// delete rightMin from original position:*

prev.left = rightMin.right;

*// replace rightMin with original root:*

rightMin.right = root.right;

}

return rightMin;

}

270. Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Given target value is a floating point.

You are guaranteed to have only one unique value in the BST that is closest to the target.

**S1. Iteration**

通过target 和root.val的关系来判断，只走一边即可。

public int closestValue(TreeNode root, double target) {

int closest = root.val;

while (root != null) {

if (Math.abs(closest - target) >= Math.abs(root.val - target)) {

closest = root.val;

}

root = target < root.val ? root.left : root.right;

}

return closest;

}

272. Closest Binary Search Tree Value II

Given a non-empty binary search tree and a target value**, find k values in the BST that are closest** to the target

Given target value is a floating point.

You may assume k is always valid, that is: k ≤ total nodes.

You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

**S1.Inorder traversal**

用中序遍历，从最小的node开始，一直到最大的，对于每个node求difference, 然后判断是否属于k closet.

如果target本身较小，那么越往后遇到的数值只可能更大；

如果target本身较大，那么先被替换掉的也是最小的值；

public List<Integer> closestKValues(TreeNode root, double target, int k) {

LinkedList<Integer> res = new LinkedList<>();

inorder(root, target, k, res);

return res;

}

private void inorder(TreeNode root, double target, int k, LinkedList<Integer> res) {

if (root == null) return;

inorder(root.left, target, k, res);

if (res.size() == k) {

if (Math.abs(res.get(0) - target) >= Math.abs(root.val - target)) {

res.removeFirst();

res.add(root.val);

}

} else {

res.add(root.val);

}

inorder(root.right, target, k, res);

}

654. Maximum Binary Tree

Given an integer array with no duplicates. A maximum tree building on this array is defined as follow:

The root is the maximum number in the array.

The left subtree is the maximum tree constructed from left part subarray divided by the maximum number.

The right subtree is the maximum tree constructed from right part subarray divided by the maximum number.

Construct the maximum tree by the given array and output the root node of this tree.

**LCA – 利用traversal**

236. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

LCA: The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”

**S1. Recursion**

非常容易理解

public TreeNode lowestCommonAncestor(TreeNode root,TreeNode p,TreeNode q) {

if (root == null || root == p || root == q) return root;

TreeNode left = lowestCommonAncestor(root.left, p, q);

TreeNode right = lowestCommonAncestor(root.right, p, q);

if (left == null) return right;

if (right == null) return left;

return root;

}

**S2 HashMap + Stack**

其实也挺好理解的，map里存的是每个mode<node, it’s parent> 的pairs，

然后在一层层往下走的时候，把所有node 的child-parent关系都存好。

然后用一个set，先从p开始，把从p到它的parent....一直上溯，直到为null全都存进set里，此时LCA一定在set中。

再从q开始上溯，第一个满足set.contains的，就是LCA.

public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {

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

HashMap<TreeNode, TreeNode> parent = new HashMap<>(); *// child, parent*

stack.push(root);

parent.put(root, null);

while (!parent.containsKey(p) || !parent.containsKey(q)) {

TreeNode cur = stack.pop();

if (cur.left != null) {

stack.push(cur.left);

parent.put(cur.left, cur);

}

if (cur.right != null) {

stack.push(cur.right);

parent.put(cur.right, cur);

}

}

Set<TreeNode> set = new HashSet<>();

while (p != null) {

set.add(p);

p = parent.get(p);

}

while (!set.contains(q)) {

q = parent.get(q);

}

return q;

}

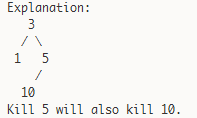
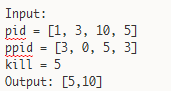
其他分类

582. Kill Process

Given n processes, each process has a unique PID (process id) and its PPID (parent process id).

Each process only has one parent process, but may have one or more children processes. This is just like a tree structure. Only one process has PPID that is 0, which means this process has no parent process. All the PIDs will be distinct positive integers.

We use two list of integers to represent a list of processes, where the first list contains PID for each process and the second list contains the corresponding PPID.



**S1 BFS**

先遍历确认<parent, List of kids>关系并放入map，

然后从被kill的pid开始，用queue做BFS，一个个往下找到被kill的process，加入res

public List<Integer> killProcess(List<Integer> pid, List<Integer> ppid, int kill) {

if (pid == null || ppid == null || pid.size() != ppid.size()) return null;

*// 1. store all <ppid, List of pid>, which is <parent, kids> in map:*

Map<Integer, List<Integer>> map = new HashMap<>();

int size = ppid.size();

for (int i = 0;i < size;i++) {

int key = ppid.get(i);

if (!map.containsKey(ppid.get(i))) {

map.put(key, new ArrayList<Integer>());

}

map.get(key).add(pid.get(i));

}

*// 2. handle one number at a time, add to res and push its kids to queue*

*// for further processing.*

Queue<Integer> queue = new LinkedList<>();

List<Integer> res = new ArrayList<>();

queue.offer(kill);

while (!queue.isEmpty()) {

int key = queue.poll();

if (map.containsKey(key)) {

for (int num:map.get(key)) {

queue.offer(num);

}

}

res.add(key);

}

return res;

}

**Recursion**

606. Construct String from Binary Tree

You need to construct a string consists of parenthesis and integers from a binary tree with the preorder traversing way.

The null node needs to be represented by empty parenthesis pair "()". And you need to omit all the empty parenthesis pairs that don't affect the one-to-one mapping relationship between the string and the original binary tree.

题解：Recursion

public String tree2str(TreeNode t) {

if (t == null) return "";

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

return String.valueOf(t.val);

} else if (t.left == null) {

return String.valueOf(t.val) + "()" + "(" + tree2str(t.right) + ")";

} else if (t.right == null) {

return String.valueOf(t.val) + "(" + tree2str(t.left) + ")";

} else {

return String.valueOf(t.val) + "(" + tree2str(t.left) + ")"

+ "(" + tree2str(t.right) + ")";

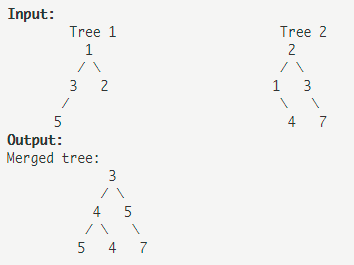
}

}

617. 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.



**S1. Recursion**

先处理root, 再recursion，处理left, right side

这么看的话，是preorder的顺序，root-left-right.

*// Preorder Traversal, make changes on t1.*

public TreeNode mergeTrees(TreeNode t1, TreeNode t2) {

if (t1 == null && t2 == null) return null;

int val = (t1 == null ? 0 : t1.val) + (t2 == null ? 0 : t2.val);

TreeNode cur = new TreeNode(val);

cur.left = mergeTrees(t1 == null ? null : t1.left, t2 == null ? null : t2.left);

cur.right = mergeTrees(t1 == null ? null : t1.right, t2 == null ? null : t2.right);

return cur;

}

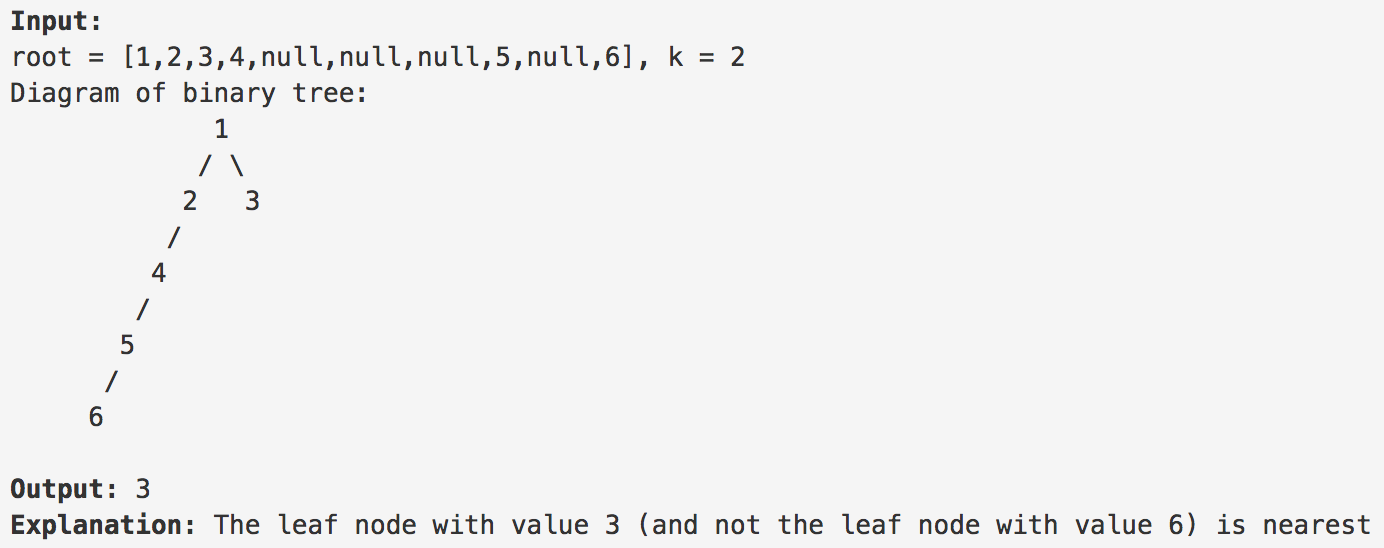
涉及其他数据结构

743. Closest Leaf in a Binary Tree

Given a binary tree where every node has a unique value, and a target key k, find the value of the nearest leaf node to target k in the tree.

Here, nearest to a leaf means the least number of edges travelled on the binary tree to reach any leaf of the tree. Also, a node is called a leaf if it has no children.

In the following examples, the input tree is represented in flattened form row by row. The actual root tree given will be a TreeNode object.



这道题的重点，其实是找到距离k这个node最近的leaf。可以转换成graph来处理。

针对中心点k做bfs, 直到遇到Leaf node