98. Validate Binary Search Tree

要求：给出一个二叉树，来决定该二叉树是否是二叉搜索树。

这道题标识为middle。

这道题可以用迭代和递归两种方法做。它就是一道中序遍历二叉树的题目。

迭代：

使用中序遍历，每遍历一个节点，就比较这个节点的val是否大于上一个节点。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {boolean}

\*/

var isValidBST = function(root) {

let stack=[];

let node=root;

let cur;

while(node!==null||stack.length!==0){

while(node!==null){

stack.push(node);

node=node.left;

}

if(stack.length!==0){

node=stack.pop();

if(cur===undefined){

cur=node.val;

}else if(cur>=node.val){

return false;

}else if(cur<node.val){

cur=node.val

}

node=node.right;

}

}

return true;

};

递归：递归定义了min和max来比较和根节点之间的大小。

var isValidBST = function(root) {

let min;

let max;

return BST(root,min,max);

};

function BST(root,min,max){

if(root===null){

return true;

}

if(root.val<=min||root.val>=max){

return false;

}

return BST(root.left,min,root.val)&&BST(root.right,root.val,max);

}

100. Same Tree

要求：这道题要求判断两颗树是否结构和每个节点的值都一样。

标示为简单。

这道题是我看了答案，因为我想找一找做二叉树题目的感觉。

答案如下：

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} p

\* @param {TreeNode} q

\* @return {boolean}

\*/

var isSameTree = function(p, q) {

if(p===null&&q===null){

return true;

}

if(p===null||q===null){

return false;

}

if(p.val===q.val){

return isSameTree(p.left,q.left)&&isSameTree(p.right,q.right);

}

return false;

};

可以看到判断结构的是这两句：

if(p===null&&q===null){

return true;

}

if(p===null||q===null){

return false;

}

判断每个节点数值是否相同的是这两句：  
if(p.val===q.val){

return isSameTree(p.left,q.left)&&isSameTree(p.right,q.right);

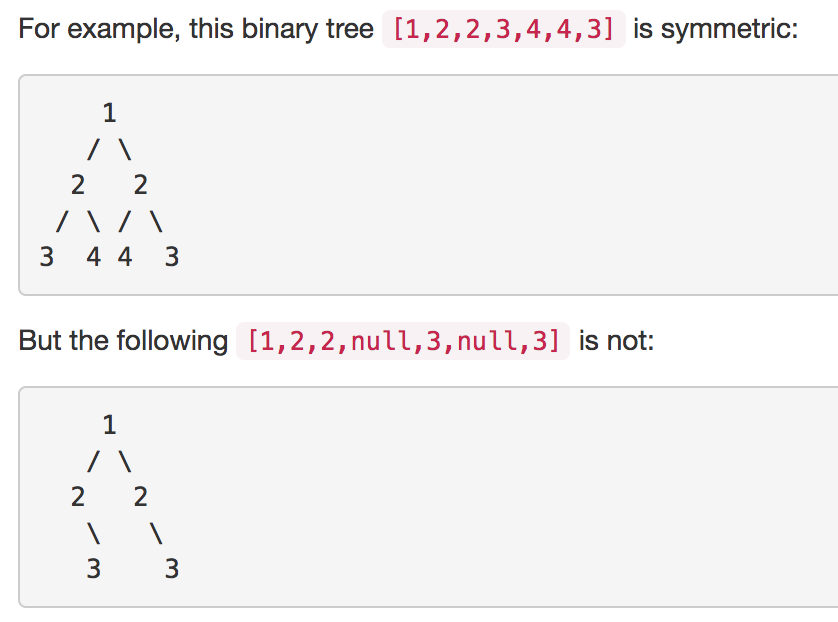
}

return false;

所以递归很神奇，，，嗯。

101. Symmetric Tree

要求：给了一个二叉树，检查它是否是它自己的镜面。



标示为简单。

因为要检查是否为镜面，则肯定需要比较**两个值是否相同，**isSymmetric函数的参数只有一个，所以需要定义一个函数用于递归处理。

我们详细解释一下isSame函数。

可以看到判断结构的是这两句：

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

return true;

}

if(left===null||right===null){

return false;

}

其中

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

return true;

}是递归结束的判断。

可以看到判断数值是否相同是这两句：

if(left.val!==right.val){

return false;

}

如果上述条件都不符合则说明暂时都满足条件，则继续调用递归。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {boolean}

\*/

var isSymmetric = function(root) {

return root===null||isSame(root.left,root.right);

};

function isSame(left,right){

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

return true;

}

if(left===null||right===null){

return false;

}

if(left.val!==right.val){

return false;

}

return isSame(left.left,right.right)&&isSame(left.right,right.left);

}

104. Maximum Depth of Binary Tree

要求：这道题要求出树的最大深度。最大深度是从根节点到最远叶节点的最长路径中的节点数。

这道题标识为简单。

递归：

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {number}

\*/

var maxDepth = function(root) {

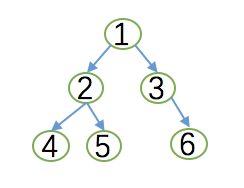
if(root===null){return 0;}

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

};

迭代：

这里要讲的是二叉树的广度优先遍历。



这是一个二叉树，使用广度优先遍历的顺序应该是1、2、3、4、5、6。

思路是定义一个列表，先将root节点push进去作为初始值，并计算当前层所包含的节点数，root层就为1，将root 从列表最前面弹出，然后访问root的left和right，将访问到的节点存入列表中。此时root层遍历结束，列表中存储的是下一层的所有节点，计算当前层所包含的节点数，然后从列表中依次弹出当前层的每个节点，并且访问每个节点的left和right节点，并存入列表中。

知道广度优先遍历的思路后，就可以在此基础上多加一个count计数器，来计算出二叉树的最大深度。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {number}

\*/

var maxDepth = function(root) {

let a=[];

let tmp;

if(root===null){

return 0;

}

a.push(root);

let count,size=0;

while(a.length!==0){

count=a.length;

while(count){

tmp=a.shift();

if(tmp.left!==null){a.push(tmp.left);}

if(tmp.right!==null){a.push(tmp.right);}

count--;

}

size++;

}

return size;

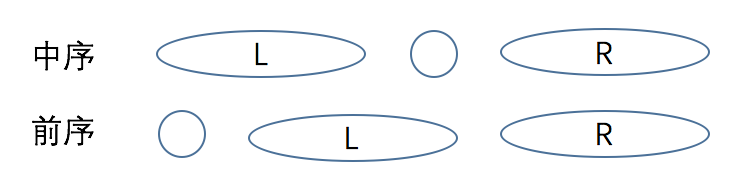
};

105. Construct Binary Tree from Preorder and Inorder Traversal

要求：给出一个树的前序和中序数组，求出这棵二叉树。

这道题标识为middle。

使用递归。



/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {number[]} preorder

\* @param {number[]} inorder

\* @return {TreeNode}

\*/

var buildTree = function(preorder, inorder) {

if(preorder.length===0||inorder.length===0){

return null;

}

let tree=new TreeNode(preorder[0]);

let index=0;

for(let i=0;i<inorder.length;i++){

if(inorder[i]===preorder[0]){//前序遍历的第一个元素是根节点，所以根据这个，找出中序遍历中根节点的位置。

index=i;

break;

}

}

console.log(index);

tree.left=buildTree(preorder.slice(1,index+1),inorder.slice(0,index));//然后用先序的左子树和中序的左子树，来计算出左子树。

tree.right=buildTree(preorder.slice(index+1),inorder.slice(index+1)); //然后用先序的右子树和中序的右子树，来计算出右子树。

return tree;

};

106. Construct Binary Tree from Inorder and Postorder Traversal

要求：给出一个树的后序和中序数组，求出这棵二叉树。

这道题与105类似，所以不细说了。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {number[]} inorder

\* @param {number[]} postorder

\* @return {TreeNode}

\*/

var buildTree = function(inorder, postorder) {

if(inorder.length===0||postorder.length===0){

return null;

}

let postLen=postorder.length;

let tree=new TreeNode(postorder[postLen-1]);

let index;

for(let i=0;i<inorder.length;i++){

if(inorder[i]===postorder[postLen-1]){

index=i;

}

}

tree.left=buildTree(inorder.slice(0,index),postorder.slice(0,index));

tree.right=buildTree(inorder.slice(index+1),postorder.slice(index,postLen-1));

return tree;

};

111. Minimum Depth of Binary Tree

要求：给定一个二叉树，找到它的最小深度。最小深度是从根节点到最近叶节点的最短路径的节点数。

这道题标识为easy。

这道题需要注意的是不能单纯的将求最大深度的max改成min，因为这里还需要考虑当一个节点左右子树至少有一个为空的情况，如果只是将max改成min，那么最短路径只会计算到当前节点，而不会算上当前节点的非空子节点。

所以需要改成如下形式：

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {number}

\*/

var minDepth = function(root) {

if(root===null){

return 0;

}

let left=minDepth(root.left);

let right=minDepth(root.right);

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

};

分为两种情况，一个是当前节点的子节点中至少一个为空和当前节点的两个子节点都不为空。

108. Convert Sorted Array to Binary Search Tree

要求：给定一个数组，其中元素按升序排序，将其转换为高度上平衡的BST。

这道题标识为easy。

这道题需要注意的高度上平衡，所以每次只需要找到数组中间的值（如果是偶数就取中间的前一个）作为根节点，然后让他左侧的数组作为左子树，右侧的数组作为右子树，然后对左子树和右子树递归。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {number[]} nums

\* @return {TreeNode}

\*/

var sortedArrayToBST = function(nums) {

if(nums.length===0){

return [];

}

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

};

function toBST(nums,start,end){

if(start>end){

return null;

}

let mid=Math.floor((start+end)/2);

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

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

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

return node;

}

110. Balanced Binary Tree

要求：这道题要判断一棵二叉树是否平衡。对于这个问题，高度平衡二叉树被定义为二叉树，其中每个节点的两个子树的深度从不相差大于1。

这道题标识为easy。

第一个思路：就是对每个节点而言，要计算以该节点为root的子树是否平衡，就需要知道，root节点的root.left和root.right的高度差，以及root.left和root.right节点是否平衡。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {boolean}

\*/

var isBalanced = function(root) {

if(root===null){

return true;

}

let leftDep = depth(root.left);

let rightDep = depth(root.right);

return Math.abs(leftDep-rightDep)<=1&&isBalanced(root.left)&&isBalanced(root.right)

};

function depth(root){

if(root===null){

return 0;

}

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

}

第二个思路：不用算出每个节点的高度差，返回值要么是-1（代表存在子树为非二叉平衡），要么返回树的高度。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {boolean}

\*/

var isBalanced = function(root) {

return dfsHeight(root)!==-1;

};

function dfsHeight(root){

if(root===null){

return 0;

}

let left=dfsHeight(root.left);

let right=dfsHeight(root.right);

if(left===-1||right===-1||Math.abs(left-right)>1){

return -1;

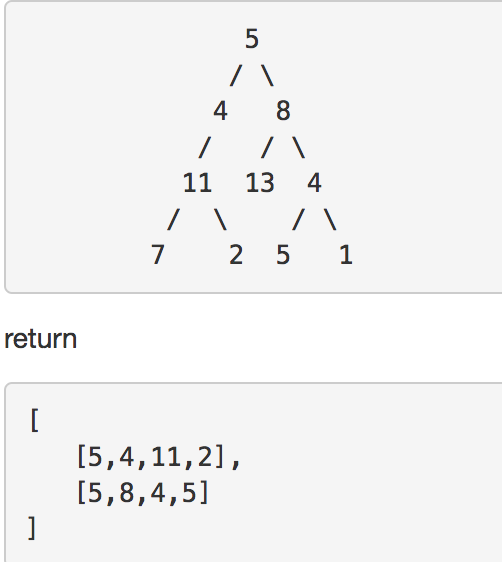
}

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

}

113. Path Sum II( 找出二叉树所有路径 )

要求：给定一个二叉树和一个sum值，找出所有从根节点到叶节点的路径，使这个路径上节点值的总和相加为sum。比如：



这道题标识为middle。

这道题可以分两步走：先找出**二叉树中的所有路径，再找出和为sum的路径。**

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @param {number} sum

\* @return {number[][]}

\*/

var pathSum = function(root, sum) {

let cur=[];

let result=[];

path(root,cur,0,result);

return result;

};

function path(root,curResult,depth,result){

// 空节点

if(root===null){

return;

}

// 叶子节点

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

curResult[depth] = root.val;

depth++;

let cur=curResult.slice(0, depth);

result.push(cur);

return;

}

curResult[depth++] = root.val;

path(root.left,curResult,depth,result);

path(root.right,curResult,depth,result);

}

如上代码可以找出二叉树所有从根到叶子的路径。使用depth记录当前所在深度。curResult记录当前路径。result记录所有路径。

最终代码如下。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @param {number} sum

\* @return {number[][]}

\*/

var pathSum = function(root, sum) {

let cur=[];

let result=[];

path(root,cur,0,result,sum);

return result;

};

function path(root,curResult,depth,result,sum){

// 空节点

if(root===null){

return;

}

// 叶子节点

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

curResult[depth] = root.val;

depth++;

let cur=curResult.slice(0, depth);

result.push(cur);

return;

}

curResult[depth++] = root.val;

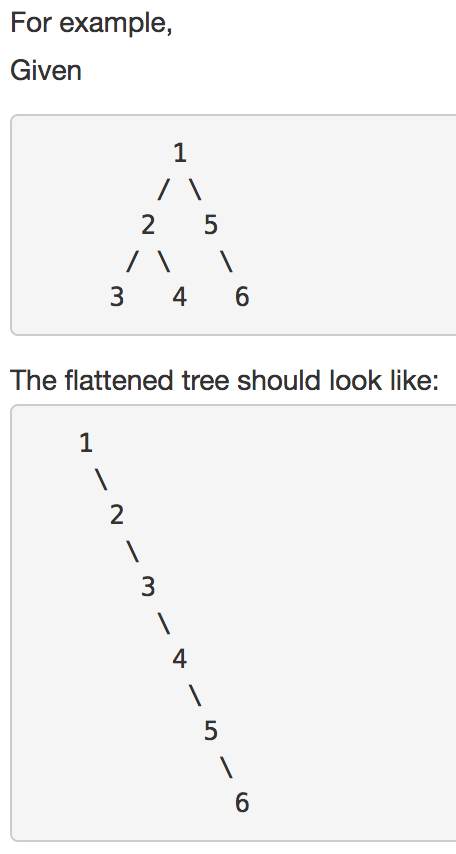
path(root.left,curResult,depth,result,sum-root.val);

path(root.right,curResult,depth,result,sum-root.val);

}

114. Flatten Binary Tree to Linked List

要求：给定一个二叉树，将其平坦化到原来的链表。比如：



这道题标识为middle。

思路一：

这是别人的思路，使用了迭代。

最终返回的是一棵每个节点只有右子树的原树深度优先遍历的结果。

思路是每次找到当前now节点左子树的最右叶子节点，然后将now节点的右子树添加到找到的最右叶子节点上，然后将now节点的左子树赋给右子树。将now节点的左子树置空。然后now节点指向now.right，接着进行下一次循环。

/\*\*

\* Definition for a binary tree node.

\* function TreeNode(val) {

\* this.val = val;

\* this.left = this.right = null;

\* }

\*/

/\*\*

\* @param {TreeNode} root

\* @return {void} Do not return anything, modify root in-place instead.

\*/

var flatten = function(root) {

let now=root;

while(now){

if(now.left){

let pre=now.left;

while(pre.right){

pre=pre.right;

}

pre.right=now.right;

now.right=now.left;

now.left=null;

}

now=now.right;

}

};

思路二：

这也是别人的思路，使用了递归。

将二叉树进行深度优先遍历，将每次遍历到的节点存入数组中。然后遍历数组，将数组每一项节点的左节点置null，右节点置为下一项节点。代码如下：

var flatten = function(root) {

const queue=[];

bfs(root,queue);

for(let i=0;i<queue.length-1;i++){

queue[i].left=null;

queue[i].right=queue[i+1];

}

};

function bfs(root,queue){

if(root===null){

return;

}

queue.push(root);

if(root.left){

bfs(root.left,queue);

}

if(root.right){

bfs(root.right,queue);

}

}