94. Binary Tree Inorder Traversal二叉树的中序遍历

给定一个二叉树，返回它的中序遍历。

示例:输入: [1,null,2,3] 输出: [1,3,2]

【分析】二叉树的中序遍历顺序为左-根-右，可以有递归和非递归来解，其中非递归解法又分为两种，一种是使用栈来接，另一种不需要使用栈。我们先来看递归方法，十分直接，对左子结点调用递归函数，根节点访问值，右子节点再调用递归函数。

【CODE】// Recursion

class Solution {

public:

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> res;

inorder(root, res);

return res;

}

void inorder(TreeNode \*root, vector<int> &res) {

if (!root) return;

if (root->left) inorder(root->left, res);

res.push\_back(root->val);

if (root->right) inorder(root->right, res);

}

};

【分析】非递归使用栈的解法，需要用栈来做，思路是从根节点开始，先将根节点压入栈，然后再将其所有左子结点压入栈，然后取出栈顶节点，保存节点值，再将当前指针移到其右子节点上，若存在右子节点，则在下次循环时又可将其所有左子结点压入栈中。这样就保证了访问顺序为左-根-右。

【CODE】// Non-recursion

class Solution {

public:

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> res;

stack<TreeNode\*> s;

TreeNode \*p = root;

while (p || !s.empty()) {

while (p) {

s.push(p);

p = p->left;

}

p = s.top();

s.pop();

res.push\_back(p->val);

p = p->right;

}

return res;

}

};

【分析】另一种很巧妙的解法，这种方法不需要使用栈，所以空间复杂度为常量，这种非递归不用栈的遍历方法有个专门的名字，叫Morris Traversal，在介绍这种方法之前，我们先来引入一种新型树，叫 Threaded binary tree，螺纹二叉树，维基百科上关于它的英文定义：

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.

就是说螺纹二叉树实际上是把所有原本为空的右子节点指向了中序遍历顺序之后的那个节点，把所有原本为空的左子节点都指向了中序遍历之前的那个节点。由于我们既不能用递归，又不能用栈，那我们如何保证访问顺序是中序遍历的左-根-右呢。原来我们需要构建一个螺纹二叉树，我们需要将所有为空的右子节点指向中序遍历的下一个节点，这样我们中序遍历完左子结点后，就能顺利的回到其根节点继续遍历了。

【算法】

1. 初始化指针cur指向root

2. 当cur不为空时

　 - 如果cur没有左子结点

　 a) 打印出cur的值

　　 b) 将cur指针指向其右子节点

　 - 反之

　 将pre指针指向cur的左子树中的最右子节点

　　　 \* 若pre不存在右子节点

　　　 a) 将其右子节点指回cur

　　　　 b) cur指向其左子节点

　　　 \* 反之

　　　　　 a) 将pre的右子节点置空

　　　　　 b) 打印cur的值

　　　　　 c) 将cur指针指向其右子节点

【CODE】// Non-recursion and no stack

class Solution {

public:

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> res;

if (!root) return res;

TreeNode \*cur, \*pre;

cur = root;

while (cur) {

if (!cur->left) {

res.push\_back(cur->val);

cur = cur->right;

} else {

pre = cur->left;

while (pre->right && pre->right != cur) pre = pre->right;

if (!pre->right) {

pre->right = cur;

cur = cur->left;

} else {

pre->right = NULL;

res.push\_back(cur->val);

cur = cur->right;

}

}

}

return res;

}

};

Morris遍历不仅仅对中序遍历有用，对先序和后序同样有用。所以对二叉树的三种常见遍历顺序(先序，中序，后序)就有三种解法(递归，非递归，Morris遍历)，总共有九段代码呀，熟练掌握这九种写法才算初步掌握了树的遍历~~ 至于二叉树的层序遍历也有递归和非递归解法，至于有没有Morris遍历的解法还有待大神们的解答~~

96. Unique Binary Search Trees不同的二叉搜索树

给定一个整数 n，求以 1 ... n 为节点组成的二叉搜索树有多少种？

【知识点】卡特兰数（Catalan number）

【分析】我们把n = 0 时赋为1，因为空树也算一种二叉搜索树，那么n = 1时的情况可以看做是其左子树个数乘以右子树的个数，左右字数都是空树，所以1乘1还是1。那么n = 2时，由于1和2都可以为跟，分别算出来，再把它们加起来即可。n = 2的情况可由下面式子算出：

dp[2] = dp[0] \* dp[1]　　　(1为根的情况)

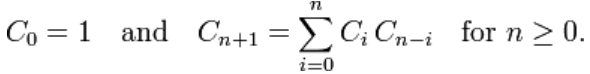
　　　　+ dp[1] \* dp[0]　　 (2为根的情况)

同理可写出 n = 3 的计算方法：

dp[3] = dp[0] \* dp[2]　　　(1为根的情况)

　　　　+ dp[1] \* dp[1]　　 (2为根的情况)

　　　 + dp[2] \* dp[0]　　 (3为根的情况)

由此可以得出卡塔兰数列的递推式为：

【CODE】

class Solution {

public:

int numTrees(int n) {

vector<int> dp(n + 1, 0);

dp[0] = 1;

dp[1] = 1;

for (int i = 2; i <= n; ++i) {

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

dp[i] += dp[j] \* dp[i - j - 1];

}

}

return dp[n];

}

};

95. Unique Binary Search Trees II不同的二叉搜索树 II

给定一个整数 n，生成所有由 1 ... n 为节点所组成的二叉搜索树。

|  |  |
| --- | --- |
| 输入: 3  输出:[  [1,null,3,2],  [3,2,null,1],  [3,1,null,null,2],  [2,1,3],  [1,null,2,null,3]  ] | 解释:  以上的输出对应以下 5 种不同结构的二叉搜索树：  1 3 3 2 1  \ / / / \ \  3 2 1 1 3 2  / / \ \  2 1 2 3 |

【分析】之前那个只要求算出所有不同的二叉搜索树的个数，这道题让把那些二叉树都建立出来。这种建树问题一般来说都是用**递归**来解，这道题也不例外，划分左右子树，递归构造。至于递归函数中为什么都用的是指针，若不用指针，全部实例化的话会存在大量的对象拷贝，要调用拷贝构造函数。

【CODE】

class Solution {

public:

vector<TreeNode \*> generateTrees(int n) {

if (n == 0) return {};

return \*generateTreesDFS(1, n);

}

vector<TreeNode\*> \*generateTreesDFS(int start, int end) {

vector<TreeNode\*> \*subTree = new vector<TreeNode\*>();

if (start > end) subTree->push\_back(NULL);

else {

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

vector<TreeNode\*> \*leftSubTree = generateTreesDFS(start, i - 1);

vector<TreeNode\*> \*rightSubTree = generateTreesDFS(i + 1, end);

for (int j = 0; j < leftSubTree->size(); ++j) {

for (int k = 0; k < rightSubTree->size(); ++k) {

TreeNode \*node = new TreeNode(i);

node->left = (\*leftSubTree)[j];

node->right = (\*rightSubTree)[k];

subTree->push\_back(node);

}

}

}

}

return subTree;

}

};

98. Validate Binary Search Tree验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉搜索树。

【分析】这道验证二叉搜索树有很多种解法，可以利用它本身的性质来做，即左<根<右，也可以通过利用中序遍历结果为有序数列来做，下面我们先来看最简单的一种，就是利用其本身性质来做，初始化时带入系统最大值和最小值，在递归过程中换成它们自己的节点值，用long代替int就是为了包括int的边界条件。

【CODE】

// Recursion without inorder traversal

class Solution {

public:

bool isValidBST(TreeNode \*root) {

return isValidBST(root, LONG\_MIN, LONG\_MAX);

}

bool isValidBST(TreeNode \*root, long mn, long mx) {

if (!root) return true;

if (root->val <= mn || root->val >= mx) return false;

return isValidBST(root->left, mn, root->val) && isValidBST(root->right, root->val, mx);

}

};

【分析】一般的二叉搜索树是左<=根<右，而这道题设定为左<根<右，那么就可以用中序遍历来做。因为如果不去掉左=根这个条件的话，那么下边两个数用中序遍历无法区分：

20 20

/ \

20 20

它们的中序遍历结果都一样，但是左边的是BST，右边的不是BST。去掉等号的条件则相当于去掉了这种限制条件。下面我们来看使用中序遍历来做，这种方法思路很直接，通过中序遍历将所有的节点值存到一个数组里，然后再来判断这个数组是不是有序的。或者，不将遍历结果存入一个数组遍历完成再比较，而是每当遍历到一个新节点时和其上一个节点比较，如果不大于上一个节点那么则返回false，全部遍历完成后返回true。

也可以用非递归来做，需要用到栈，因为中序遍历可以非递归来实现。最后还有一种方法，由于中序遍历还有非递归且无栈的实现方法，称之为Morris遍历，可以参考Binary Tree Inorder Traversal，这种实现方法虽然写起来比递归版本要复杂的多，但是好处在于是O(1)空间复杂度。

99. Recover Binary Search Tree恢复二叉搜索树

二叉搜索树中的两个节点被错误地交换。

请在不改变其结构的情况下，恢复这棵树。

进阶:使用 O(n) 空间复杂度的解法很容易实现。

你能想出一个只使用常数空间的解决方案吗？

【分析】传统的中序遍历递归，不过再应该输出节点值的地方，换成了判断pre和当前节点值的大小，如果pre的大，若first为空，则将first指向pre指的节点，把second指向当前节点。这样中序遍历完整个树，若first和second都存在，则交换它们的节点值即可。这个算法的空间复杂度仍为O(n)。

常数空间解决方案：用的Morris遍历，这是一种非递归且不使用栈，空间复杂度为O(1)的遍历方法，可参见之前Binary Tree Inorder Traversal 二叉树的中序遍历，在其基础上做些修改，加入first, second和parent指针，来比较当前节点值和中序遍历的前一节点值的大小，跟递归算法的思路相似。

【CODE】// Now O(1) space complexity

class Solution {

public:

void recoverTree(TreeNode \*root) {

TreeNode \*first = NULL, \*second = NULL, \*parent = NULL;

TreeNode \*cur, \*pre;

cur = root;

while (cur) {

if (!cur->left) {

if (parent && parent->val > cur->val) {

if (!first) first = parent;

second = cur;

}

parent = cur;

cur = cur->right;

} else {

pre = cur->left;

while (pre->right && pre->right != cur) pre = pre->right;

if (!pre->right) {

pre->right = cur;

cur = cur->left;

} else {

pre->right = NULL;

if (parent->val > cur->val) {

if (!first) first = parent;

second = cur;

}

parent = cur;

cur = cur->right;

}

}

}

if (first && second) swap(first->val, second->val);

}

};

100. Same Tree相同的树

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

【分析】判断两棵树是否相同和之前的判断两棵树是否对称都是一样的原理，利用深度优先搜索DFS来递归。

【知识点】DFS深度优先搜索；二叉树的四种遍历(层序，先序，中序，后序)

【CODE】class Solution {

public:

bool isSameTree(TreeNode \*p, TreeNode \*q) {

if (!p && !q) return true;

if ((p && !q) || (!p && q) || (p->val != q->val)) return false;

return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

}

};

101. Symmetric Tree对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

|  |  |
| --- | --- |
| 二叉树 [1,2,2,3,4,4,3] 是对称的。  1  / \  2 2  / \ / \  3 4 4 3 | [1,2,2,null,3,null,3] 则不是镜像对称的:  1  / \  2 2  \ \  3 3 |

【分析】判断二叉树是否是平衡树，比如有两个节点n1, n2，我们需要比较n1的左子节点的值和n2的右子节点的值是否相等，同时还要比较n1的右子节点的值和n2的左子结点的值是否相等，以此类推比较完所有的左右两个节点。我们可以用递归和迭代两种方法来实现，写法不同，但是算法核心都一样。

【CODE】class Solution {

public:

bool isSymmetric(TreeNode \*left, TreeNode \*right) {

if (!left && !right) return true;

if (left && !right || !left && right || left->val != right->val) return false;

return isSymmetric(left->left, right->right) && isSymmetric(left->right, right->left);

}

bool isSymmetric(TreeNode \*root) {

if (!root) return true;

return isSymmetric(root->left, root->right);

}

};

102. Binary Tree Level Order Traversal二叉树的层次遍历

给定一个二叉树，返回其按层次遍历的节点值。

（即逐层地，从左到右访问所有节点）。

【分析】层序遍历二叉树是典型的广度优先搜索BFS的应用，但是这里稍微复杂一点的是，我们要把各个层的数分开，存到一个二维向量里面，大体思路还是基本相同的，建立一个queue，然后先把根节点放进去，这时候找根节点的左右两个子节点，这时候去掉根节点，此时queue里的元素就是下一层的所有节点，用一个for循环遍历它们，然后存到一个一维向量里，遍历完之后再把这个一维向量存到二维向量里，以此类推，可以完成层序遍历。

【CODE】// Iterative

class Solution {

public:

vector<vector<int> > levelOrder(TreeNode \*root) {

vector<vector<int> > res;

if (root == NULL) return res;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

vector<int> oneLevel;

int size = q.size();

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

TreeNode \*node = q.front();

q.pop();

oneLevel.push\_back(node->val);

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

res.push\_back(oneLevel);

}

return res;

}

};

【分析】递归的写法，核心就在于我们需要一个二维数组，和一个变量level，当level递归到上一层的个数，我们新建一个空层，继续往里面加数字

【CODE】// Recursive

class Solution {

public:

vector<vector<int>> levelOrder(TreeNode\* root) {

vector<vector<int> > res;

levelorder(root, 0, res);

return res;

}

void levelorder(TreeNode \*root, int level, vector<vector<int> > &res) {

if (!root) return;

if (res.size() == level) res.push\_back({});

res[level].push\_back(root->val);

if (root->left) levelorder(root->left, level + 1, res);

if (root->right) levelorder(root->right, level + 1, res);

}

};

103. Binary Tree Zigzag Level Order Traversal二叉树的锯齿形层次遍历

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

|  |  |
| --- | --- |
| 给定二叉树 [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7 | 返回锯齿形层次遍历如下：  [  [3],  [20,9],  [15,7]  ] |

【分析】这道二叉树的之字形层序遍历是之前那道[LeetCode] Binary Tree Level Order Traversal 二叉树层序遍历的变形，不同之处在于一行是从左到右遍历，下一行是从右往左遍历，交叉往返的之字形的层序遍历。根据其特点我们用到栈的后进先出的特点，这道题我们维护两个栈，相邻两行分别存到两个栈中，进栈的顺序也不相同，一个栈是先进左子结点然后右子节点，另一个栈是先进右子节点然后左子结点，这样出栈的顺序就是我们想要的之字形。

104. Maximum Depth of Binary Tree二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

【分析】求二叉树的最大深度问题用到深度优先搜索DFS，递归的完美应用，跟求二叉树的最小深度问题原理相同。

【CODE】class Solution {

public:

int maxDepth(TreeNode\* root) {

if (!root) return 0;

return 1 + max(maxDepth(root->left), maxDepth(root->right));

}

};

105. Construct Binary Tree from Preorder and Inorder Traversal

从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

注意:你可以假设树中没有重复的元素。

【分析】先序的顺序的第一个肯定是根，所以原二叉树的根节点可以知道，题目中给了一个很关键的条件就是树中没有相同元素，有了这个条件我们就可以在中序遍历中也定位出根节点的位置，并以根节点的位置将中序遍历拆分为左右两个部分，分别对其递归调用原函数。

【思考】怎么没有由先序和后序遍历建立二叉树呢？这是因为先序和后序遍历不能唯一的确定一个二叉树。对于先序遍历都为1 2 3的五棵二叉树，它们的中序遍历都不相同，而它们的后序遍历却有相同的，所以只有和中序遍历一起才能唯一的确定一棵二叉树。

|  |  |  |  |
| --- | --- | --- | --- |
| 1  / \  2 3 | preorder: 1 2 3  inorder: 2 1 3  postorder: 2 3 1 |  | preorder: 1 2 3  inorder: 3 2 1  postorder: 2 3 1 |
|  | preorder: 1 2 3  inorder: 2 3 1  postorder: 2 3 1 |  | preorder: 1 2 3  inorder: 1 3 2  postorder: 2 3 1 |
|  | preorder: 1 2 3  inorder: 1 2 3  postorder: 2 3 1 |  |  |

106. Construct Binary Tree from Inorder and Postorder Traversal

从中序与后序遍历序列构造二叉树

根据一棵树的中序遍历与后序遍历构造二叉树。

注意:你可以假设树中没有重复的元素。

【分析】要求从中序和后序遍历的结果来重建原二叉树，我们知道中序的遍历顺序是左-根-右，后序的顺序是左-右-根，对于这种树的重建一般都是采用递归来做，可参见Convert Sorted Array to Binary Search Tree 将有序数组转为二叉搜索树。针对这道题，由于后序的顺序的最后一个肯定是根，所以原二叉树的根节点可以知道，题目中给了一个很关键的条件就是树中没有相同元素，有了这个条件我们就可以在中序遍历中也定位出根节点的位置，并以根节点的位置将中序遍历拆分为左右两个部分，分别对其递归调用原函数。

107. Binary Tree Level Order Traversal II二叉树的层次遍历 II

给定一个二叉树，返回其节点值自底向上的层次遍历。

（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

|  |  |
| --- | --- |
| 给定二叉树 [3,9,20,null,null,15,7],  3  / \  9 20  / \  15 7 | 返回其自底向上的层次遍历为：  [  [15,7],  [9,20],  [3]  ] |

【分析】从底部层序遍历其实还是从顶部开始遍历，只不过最后存储的方式有所改变。另一种递归的解法，核心就在于我们需要一个二维数组，和一个变量level，当level递归到上一层的个数，我们新建一个空层，继续往里面加数字。

108. Convert Sorted Array to Binary Search Tree将有序数组转换为二叉搜索树

将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

【分析】将有序数组转为二叉搜索树，所谓二叉搜索树，是一种始终满足左<根<右的特性，如果将二叉搜索树按中序遍历的话，得到的就是一个有序数组了。那么反过来，我们可以得知，根节点应该是有序数组的中间点，从中间点分开为左右两个有序数组，在分别找出其中间点作为原中间点的左右两个子节点，这正是二分查找法的核心思想。

【CODE】

class Solution {

public:

TreeNode \*sortedArrayToBST(vector<int> &num) {

return sortedArrayToBST(num, 0 , num.size() - 1);

}

TreeNode \*sortedArrayToBST(vector<int> &num, int left, int right) {

if (left > right) return NULL;

int mid = (left + right) / 2;

TreeNode \*cur = new TreeNode(num[mid]);

cur->left = sortedArrayToBST(num, left, mid - 1);

cur->right = sortedArrayToBST(num, mid + 1, right);

return cur;

}

};

110. Balanced Binary Tree平衡二叉树

给定一个二叉树，判断它是否是高度平衡的二叉树。

【分析】求二叉树是否平衡，根据题目中的定义，高度平衡二叉树是每一个节点的两个字数的深度差不能超过1，那么我们肯定需要一个求各个点深度的函数，然后对每个节点的两个子树来比较深度差，时间复杂度为O(NlgN)。

上法正确但不是很高效，因为每一个点都会被上面的点计算深度时访问一次，我们可以进行优化。方法是如果我们发现子树不平衡，则不计算具体的深度，而是直接返回-1。那么优化后的方法为：对于每一个节点，我们通过checkDepth方法递归获得左右子树的深度，如果子树是平衡的，则返回真实的深度，若不平衡，直接返回-1，此方法时间复杂度O(N)，空间复杂度O(H)。

【CODE】class Solution {

public:

bool isBalanced(TreeNode \*root) {

if (checkDepth(root) == -1) return false;

else return true;

}

int checkDepth(TreeNode \*root) {

if (!root) return 0;

int left = checkDepth(root->left);

if (left == -1) return -1;

int right = checkDepth(root->right);

if (right == -1) return -1;

int diff = abs(left - right);

if (diff > 1) return -1;

else return 1 + max(left, right);

}

};

111. Minimum Depth of Binary Tree二叉树的最小深度

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明: 叶子节点是指没有子节点的节点。

【分析】二叉树的经典问题之最小深度问题就是就最短路径的节点个数，还是用深度优先搜索DFS来完成，万能的递归。

【CODE】

class Solution {

public:

int minDepth(TreeNode \*root) {

if (root == NULL) return 0;

if (root->left == NULL && root->right == NULL) return 1;

if (root->left == NULL) return minDepth(root->right) + 1;

else if (root->right == NULL) return minDepth(root->left) + 1;

else return 1 + min(minDepth(root->left), minDepth(root->right));

}

};