树

树的最小深度：

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.

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，万能的递归啊。。。请看代码：

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

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));

}

};

树的最大深度

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.

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。代码如下：

class Solution {

public:

int maxDepth(TreeNode\* root) {

if (!root) return 0;

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

}

};

二叉树的层次遍历

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

For example:  
Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its level order traversal as:

[

[3],

[9,20],

[15,7]

]

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。代码如下：

// 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;

}

};

二叉树的层次遍历

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

For example:  
Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its bottom-up level order traversal as:

[

[15,7],

[9,20],

[3]

]

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

// Iterative

class Solution {

public:

vector<vector<int> > levelOrderBottom(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.insert(res.begin(), oneLevel);

}

return res;

}

};

判断树是否对称

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

For example, this binary tree is symmetric:

1

/ \

2 2

/ \ / \

3 4 4 3

But the following is not:

1

/ \

2 2

\ \

3 3

Note:  
Bonus points if you could solve it both recursively and iteratively.

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

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool isSymmetric(TreeNode \*root) {

if (!root) return true;

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

}

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);

}

};

判断树是否相同

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.

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。代码如下：

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);

}

};

树的Z字形层次遍历

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).

For example:  
Given binary tree {3,9,20,#,#,15,7},

3

/ \

9 20

/ \

15 7

return its zigzag level order traversal as:

[

[3],

[20,9],

[15,7]

]

confused what "{1,#,2,3}" means? [> read more on how binary tree is serialized on OJ.](https://oj.leetcode.com/problems/binary-tree-zigzag-level-order-traversal/)

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](http://www.cnblogs.com/grandyang/p/4051321.html)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了，代码如下：

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

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

vector<vector<int> >res;

if (!root) return res;

stack<TreeNode\*> s1;

stack<TreeNode\*> s2;

s1.push(root);

vector<int> out;

while (!s1.empty() || !s2.empty()) {

while (!s1.empty()) {

TreeNode \*cur = s1.top();

s1.pop();

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

if (cur->left) s2.push(cur->left);

if (cur->right) s2.push(cur->right);

}

if (!out.empty()) res.push\_back(out);

out.clear();

while (!s2.empty()) {

TreeNode \*cur = s2.top();

s2.pop();

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

if (cur->right) s1.push(cur->right);

if (cur->left) s1.push(cur->left);

}

if (!out.empty()) res.push\_back(out);

out.clear();

}

return res;

}

};

二叉树的路径和

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.

For example:  
Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ \

7 2 1

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

这道求二叉树的路径需要用深度优先算法DFS的思想来遍历每一条完整的路径，也就是利用递归不停找子节点的左右子节点，而调用递归函数的参数只有当前节点和sum值。首先，如果输入的是一个空节点，则直接返回false，如果如果输入的只有一个根节点，则比较当前根节点的值和参数sum值是否相同，若相同，返回true，否则false。 这个条件也是递归的终止条件。下面我们就要开始递归了，由于函数的返回值是Ture/False，我们可以同时两个方向一起递归，中间用或||连接，只要有一个是True，整个结果就是True。递归左右节点时，这时候的sum值应该是原sum值减去当前节点的值。代码如下：

/\*\*

\* Definition for binary tree

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

\*/

class Solution {

public:

bool hasPathSum(TreeNode \*root, int sum) {

if (root == NULL) return false;

if (root->left == NULL && root->right == NULL && root->val == sum ) return true;

return hasPathSum(root->left, sum - root->val) || hasPathSum(root->right, sum - root->val);

}

};

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

For example:  
Given the below binary tree and sum = 22,

5

/ \

4 8

/ / \

11 13 4

/ \ / \

7 2 5 1

return

[

[5,4,11,2],

[5,8,4,5]

]

这道二叉树路径之和在之前的基础上又需要找出路径 (可以参见我之前的博客 [http://www.cnblogs.com/grandyang/p/4036961.html](http://www.cnblogs.com/grandyang/p/4036961.html" \t "http://www.cnblogs.com/grandyang/p/_blank))，但是基本思想都一样，还是需要用深度优先搜索DFS，只不过数据结构相对复杂一点，需要用到二维的vector，而且每当DFS搜索到新节点时，都要保存该节点。而且每当找出一条路径之后，都将这个保存为一维vector的路径保存到最终结果二位vector中。并且，每当DFS搜索到子节点，发现不是路径和时，返回上一个结点时，需要把该节点从一维vector中移除。代码如下：

class Solution {

public:

vector<vector<int> > pathSum(TreeNode \*root, int sum) {

vector<vector<int>> res;

vector<int> out;

helper(root, sum, out, res);

return res;

}

void helper(TreeNode\* node, int sum, vector<int>& out, vector<vector<int>>& res) {

if (!node) return;

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

if (sum == node->val && !node->left && !node->right) {

res.push\_back(out);

}

helper(node->left, sum - node->val, out, res);

helper(node->right, sum - node->val, out, res);

out.pop\_back();

}

};