# 题目

给你一个二叉树，请你返回其按层序遍历得到的节点值。（即逐层地，从左到右访问所有节点）。

示例：

二叉树：[3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回其层次遍历结果：

[

[3],

[9,20],

[15,7]

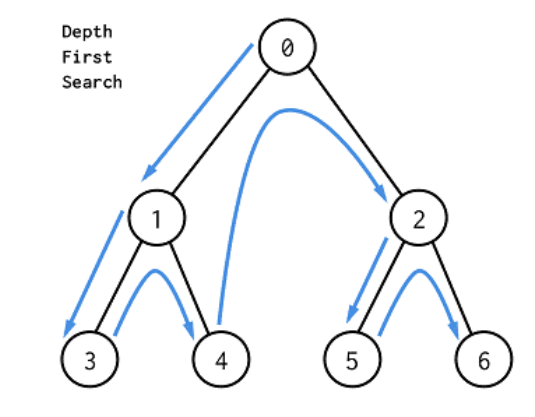
]

注：面试题32 - II 从上到下打印二叉树 II

# 分析

## 方法一：递归法/深度优先遍历

**思路：**深度优先遍历



先一路走到底，然后再回头搜索。

由于题目要求每一层的节点都是从左到右遍历，因此递归时也要先递归左子树、再递归右子树。

DFS做本题的主要问题是：DFS不是按照层次遍历的。为了让递归的过程中同一层的节点放到同一个列表中，在递归时要记录每个节点的深度level。递归到新节点要把该节点放入level对应列表的末尾。

**当遍历到一个新的深度level，而最终结果res中还没有创建level对应的列表时，应该在res中新建一个列表用来保存该level的所有节点。**

**代码：**

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

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

\* };

\*/

class Solution {

public:

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

vector<vector<int>> res;

dfs(res, root, 0);

return res;

}

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

if (!root) return;

if (level >= res.size())//开辟存储下一个层次的数组

res.push\_back(vector<int>());

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

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

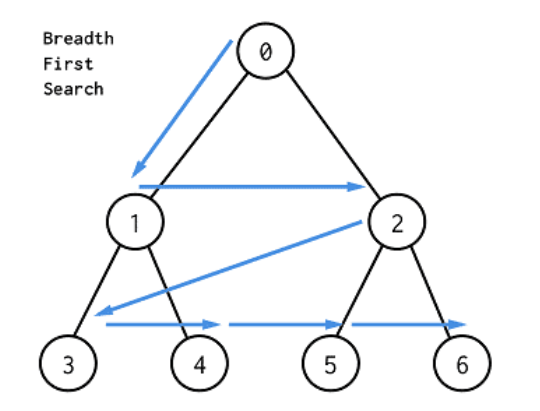
dfs(res, root->right, level + 1);

}

};

## 方法二：迭代法/宽度优先遍历

**思路：**

****

BFS使用队列，把每个还没有搜索到的点依次放入队列，然后再弹出队列的头部元素当做当前遍历点。BFS总共有两个模板：

1、如果不需要确定当前遍历到了哪一层，BFS模板如下。

while queue 不空：

cur = queue.pop()

for 节点 in cur的所有相邻节点：

if 该节点有效且未访问过：

queue.push(该节点)

2、如果要确定当前遍历到了哪一层，BFS模板如下。

这里增加了level表示当前遍历到二叉树中的哪一层了，也可以理解为在一个图中，现在已经走了多少步了。size表示在当前遍历层有多少个元素，也就是队列中的元素数，我们把这些元素一次性遍历完，即把当前层的所有元素都向外走了一步。

level = 0

while queue 不空：

size = queue.size()

while (size --) {

cur = queue.pop()

for 节点 in cur的所有相邻节点：

if 该节点有效且未被访问过：

queue.push(该节点)

}

level ++;

上面两个是通用模板，在任何题目中都可以用，是要记住的！

本题要求二叉树的层次遍历，所以同一层的节点应该放在一起，故使用模板二。

使用队列保存每层的所有节点，每次把队列里的原先所有节点进行出队列操作，再把每个元素的非空左右子节点进入队列。因此即可得到每层的遍历。

**代码：**

class Solution {

public:

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

vector <vector <int>> ret;

if (!root) return ret;

queue <TreeNode\*> q;

q.push(root);

while (!q.empty()) {

int currentLevelSize = q.size();

ret.push\_back(vector <int> ());

for (int i = 1; i <= currentLevelSize; ++i) {

auto node = q.front();

q.pop();

ret.back().push\_back(node->val);//具体的操作

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

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

}

}

return ret;

}

};

**复杂度分析：**

记树上所有节点的个数为n。

时间复杂度：每个点进队出队各一次，故渐进时间复杂度为O(n)。

空间复杂度：队列中元素的个数不超过n个，故渐进空间复杂度为O(n)。

**或：**

class Solution {

public:

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

        if(nullptr == root) return {};

        vector<vector<int>> ret;

        queue<TreeNode\*> que;

        que.push(root);

        while(!que.empty())

        {

            int num = que.size();

            vector<int> vec;

            for(int i=0;i<num;i++)

            {

                auto node = que.front();

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

                que.pop();

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

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

                if(i==num-1) //到达最后一个节点的时候存入到数组中

                    ret.push\_back(vec);

            }

        }

        return ret;

    }

};

**或：**

class Solution {

public:

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

queue<TreeNode\*> que;

que.push(root);

vector<vector<int>> res;

while (que.size() != 0) { //或者使用empty()

int size = que.size(); //当前层节点个数

vector<int> level;//存储每一层的node

while (size --) {

TreeNode\* cur = que.front();

que.pop();

if (!cur) { //必不可少，跳过null节点

continue;

}

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

que.push(cur->left);

que.push(cur->right);

}

if (level.size() != 0) { //防止最后输出null

res.push\_back(level);//存储当前层的node数组

}

}

return res;

}

};

注：上述输出不要求输出null，则需要过滤这种情况（标红处）。