# 题目

给定一棵二叉树，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。

**示例:**

输入: [1,2,3,null,5,null,4]

输出: [1, 3, 4]

解释:

1 <---

/ \

2 3 <---

\ \

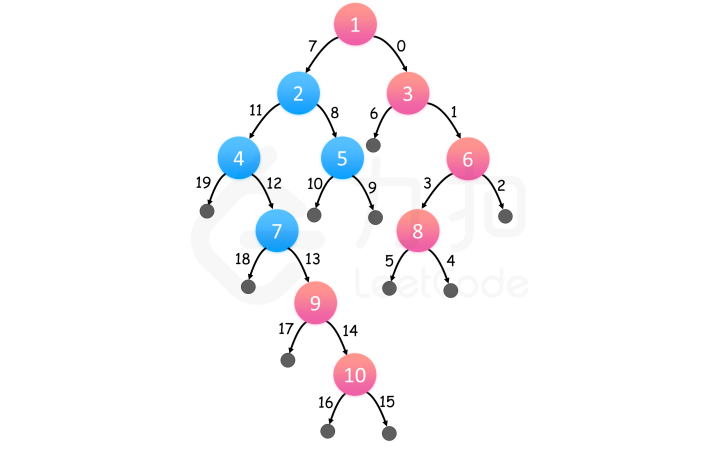
5 4 <---

# 分析

## 方法一：深度优先遍历

**思路：**

我们对树进行深度优先搜索，在搜索过程中，我们总是先访问右子树。那么对于每一层来说，我们在这层见到的第一个结点一定是最右边的结点。



**代码：**

class Solution {

public:

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

unordered\_map<int, int> rightmostValueAtDepth;

int max\_depth = -1;

stack<TreeNode\*> nodeStack;

stack<int> depthStack;

nodeStack.push(root);

depthStack.push(0);

while (!nodeStack.empty()) {

TreeNode\* node = nodeStack.top();nodeStack.pop();

int depth = depthStack.top();depthStack.pop();

if (node != NULL) {

// 维护二叉树的最大深度

max\_depth = max(max\_depth, depth);

// 如果不存在对应深度的节点我们才插入

if (rightmostValueAtDepth.find(depth) == rightmostValueAtDepth.end()) {

rightmostValueAtDepth[depth] = node -> val;

}

nodeStack.push(node -> left);

nodeStack.push(node -> right);

depthStack.push(depth + 1);

depthStack.push(depth + 1);

}

}

vector<int> rightView;

for (int depth = 0; depth <= max\_depth; ++depth) {

rightView.push\_back(rightmostValueAtDepth[depth]);

}

return rightView;

}

};

**复杂度分析：**

时间复杂度：O(n)。深度优先搜索最多访问每个结点一次，因此是线性复杂度。

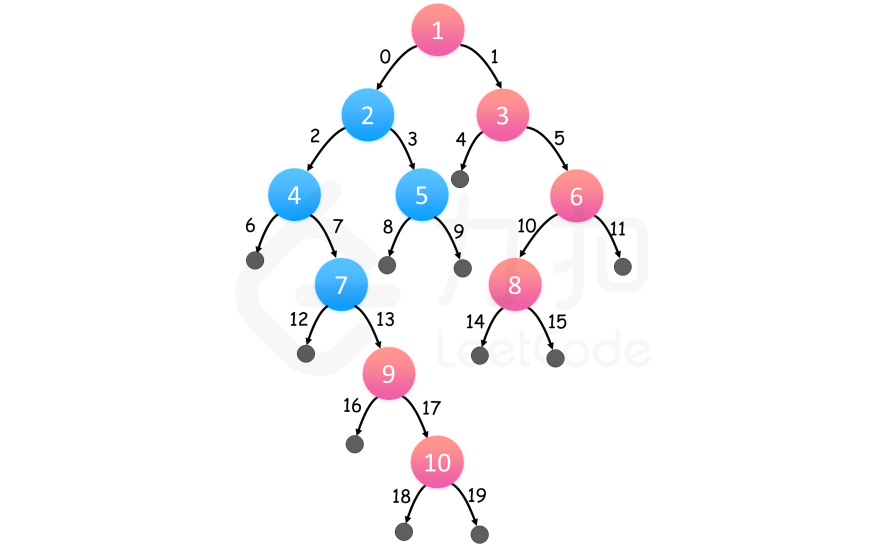
空间复杂度：O(n)。最坏情况下，栈内会包含接近树高度的结点数量，占用O(n)的空间。

## 方法二：广度优先遍历

**思路：**

我们可以对二叉树进行层次遍历，那么对于每层来说，最右边的结点一定是最后被遍历到的。二叉树的层次遍历可以用广度优先搜索实现。

执行广度优先搜索，左结点排在右结点之前，这样，我们对每一层都从左到右访问。因此，只保留每个深度最后访问的结点，我们就可以在遍历完整棵树后得到每个深度最右的结点。除了将栈改成队列，并去除了rightmost\_value\_at\_depth之前的检查外，算法没有别的改动。



上图表示了同一个示例，红色结点自上而下组成答案，边缘以访问顺序标号。

**代码：**

class Solution {

public:

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

unordered\_map<int, int> rightmostValueAtDepth;

int max\_depth = -1;

queue<TreeNode\*> nodeQueue;

queue<int> depthQueue;

nodeQueue.push(root);

depthQueue.push(0);

while (!nodeQueue.empty()) {

TreeNode\* node = nodeQueue.front();

nodeQueue.pop();

int depth = depthQueue.front();

depthQueue.pop();

if (node != NULL) {

// 维护二叉树的最大深度

max\_depth = max(max\_depth, depth);

// 由于每一层最后一个访问到的节点才是我们要的答案，因此不断更新对应深度的信息即可

rightmostValueAtDepth[depth] = node -> val;

nodeQueue.push(node -> left);

nodeQueue.push(node -> right);

depthQueue.push(depth + 1);

depthQueue.push(depth + 1);

}

}

vector<int> rightView;

for (int depth = 0; depth <= max\_depth; ++depth) {

rightView.push\_back(rightmostValueAtDepth[depth]);

}

return rightView;

}

};

**复杂度分析：**

时间复杂度：O(n)。每个节点最多进队列一次，出队列一次，因此广度优先搜索的复杂度为线性。

空间复杂度：O(n)。每个节点最多进队列一次，所以队列长度最大不不超过n，所以这里的空间代价为O(n)。

或（推荐）：

class Solution {

public:

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

        if(nullptr == root) return {};

        vector<int> ret;

        queue<TreeNode\*> que;

        que.push(root);

        while(!que.empty())

        {

            int num = que.size();

            for(int i=0;i<num;i++)

            {

                auto node = que.front();

                if(i == num-1)

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

                que.pop();

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

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

            }

        }

        return ret;

    }

};