# 题目

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。

**示例：**

给定二叉树 [3,9,20,null,null,15,7]，

3

/ \

9 20

/ \

15 7

返回它的最大深度 3 。

**类似题目：面试题55 - I**

# 分析

递归、栈循环实现深度优先遍历；用队列循环实现层遍历。

## 方法一：递归

**思路：**

直观的方法是通过递归来解决问题。利用DFS（深度优先遍历），DFS可以采用递归和栈实现。

**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) {}

\* };

\*/

//深度优先遍历，递归

class Solution {

public:

int dfs(TreeNode\* root, int max\_dept) {

if(!root) return max\_dept;

//到叶子节点的时候返回，此时max\_dept是递增后的

int left\_dept = dfs(root->left, max\_dept+1);

int right\_dept = dfs(root->right, max\_dept+1);

//不能用max\_dept++/++max\_depth

return max(left\_dept, right\_dept);

}

//这个求最大深度的通用做法，务必掌握！

int maxDepth(TreeNode\* root) {

int max\_dept = 0;

return dfs(root, max\_dept);

}

/\*另外一种写法：

int maxDepth(TreeNode \*node)

{

if(node==nullptr)return 0;

int left\_depth = maxDepth(node->left);

int right\_depth = maxDepth(node->right);

return max(left\_depth,right\_depth)+1;

}

\*/

};

另外一种写法：

class Solution {

public:

int maxDepth(TreeNode\* root) {

if (root == nullptr) return 0;

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

}

};

或：

class Solution {

public:

int maxDepth(TreeNode\* root) {

if(root==NULL) return 0;

int l=maxDepth(root->left)+1;

int r=maxDepth(root->right)+1;

return l>r?l:r;

}

};

**复杂度：**

时间复杂度：我们每个结点只访问一次，因此时间复杂度为O(N)，其中N是结点的数量。

空间复杂度：在最糟糕的情况下，树是完全不平衡的，例如每个结点只剩下左子结点，递归将会被调用N次（树的高度），因此保持调用栈的存储将是O(N)。但在最好的情况下（树是完全平衡的），树的高度将是log(N)。因此，在这种情况下的空间复杂度将是 O(log(N))。

## 方法二：迭代

### 思路一：深度优先遍历

**思路：**利用栈实现（可能会栈溢出）

**代码：**

class Solution {

public:

int maxDepth(TreeNode\* root) {

if(root==NULL) return 0;

stack<pair<TreeNode\*,int>> s;

TreeNode\* p=root;

int Maxdeep=0;

int deep=0;

while(!s.empty()||p!=NULL)

//若栈非空，则说明还有一些节点的右子树尚未探索；若p非空，意味着还有一些节点的左子树尚未探索

{

while(p!=NULL)//优先往左边走

{

s.push(pair<TreeNode\*,int>(p,++deep));

p=p->left;

}

p=s.top().first;

//若左边无路，就预备右拐。右拐之前，记录右拐点的基本信息

deep=s.top().second;

if(Maxdeep<deep) Maxdeep=deep;

//预备右拐时，比较当前节点深度和之前存储的最大深度

//典型错误：if(Maxdeep<s.top().second) Maxdeep=s.top().second;这里没有对dep重新赋值，需要对dep重新赋值

s.pop();

//将右拐点出栈；此时栈顶为右拐点的前一个结点。在右拐点的右子树全被遍历完后，会预备在这个节点右拐

p=p->right;

}

return Maxdeep;

}

};

另一种写法：

/\*\*

 \* Definition for a binary tree node.

 \* struct TreeNode {

 \*     int val;

 \*     TreeNode \*left;

 \*     TreeNode \*right;

 \*     TreeNode() : val(0), left(nullptr), right(nullptr) {}

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

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

 \* };

 \*/

class Solution {

public:

    int maxDepth(TreeNode\* root) {

        if(nullptr == root) return 0;

        stack<pair<TreeNode\*,int>> stk;

        TreeNode \*tmpNode = root;

        int maxDep = 0;

        int dep = 0;

        while(tmpNode || !stk.empty())

        {

            while(tmpNode)

            {

                dep++;

                stk.push(pair<TreeNode\*,int>(tmpNode,dep));

                tmpNode = tmpNode->left;

            }

            tmpNode = stk.top().first;

            dep = stk.top().second;

            if(stk.top().second > maxDep)   maxDep=dep;

            stk.pop();

            tmpNode = tmpNode->right;

        }

        return maxDep;

    }

};

### 思路二：宽度优先遍历

**思路：**广度/宽度优先遍历（利用队列，防止栈溢出）

我们也可以用「广度优先搜索」的方法来解决这道题目，但我们需要对其进行一些修改，此时我们广度优先搜索的队列里存放的是「当前层的所有节点」。每次拓展下一层的时候，不同于广度优先搜索的每次只从队列里拿出一个节点，我们需要将队列里的所有节点都拿出来进行拓展，这样能保证每次拓展完的时候队列里存放的是当前层的所有节点，即我们是一层一层地进行拓展，最后我们用一个变量ans来维护拓展的次数，该二叉树的最大深度即为ans。

**说明：**这个题目与前面相同树中使用队列遍历树的操作不同，LeetCode100不是严格的广度优先遍历，只是借助队列实现扫描，这里是严格的层次遍历，因此需要获取每一行的队列大小，然后依次出队。

**代码：**

class Solution {

public:

int maxDepth(TreeNode\* root) {

queue<TreeNode\*> q;

int depth = 0;//需要初始化

if (root != nullptr) q.push(root);

while (!q.empty()) {

++depth ;

int num = q.size();

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

//不能使用int i=0;i<q.size();i++，因为后面操作中q是不断变化的，这里的条件也是动态变化了

TreeNode\* cur = q.front();

q.pop();

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

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

}

}

return depth ;

}

};

**复杂度：**

时间复杂度：O(N)。

空间复杂度：O(N)。

**另一种写法：**

class Solution {

public:

int maxDepth(TreeNode\* root) {

if (root == nullptr) return 0;

queue<TreeNode\*> Q;

Q.push(root);

int ans = 0;

while (!Q.empty()) {

int sz = Q.size();

while (sz > 0) {

TreeNode\* node = Q.front();Q.pop();

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

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

sz -= 1;

}

ans += 1;

}

return ans;

}

};

典型错误：

class Solution {

public:

    int maxDepth(TreeNode\* root) {

        if(nullptr == root)

            return 0;

        queue<TreeNode \*> que;

        que.push(root);

        int depth  = 0;

        while(!que.empty())

        {

            depth++;

            TreeNode \*node = que.front();

            que.pop(); //这里如果有左右两个子节点，则这里只弹出一个

            if(node->left)

                que.push(node->left);

            if(node->right)

                que.push(node->right);

        }

        return depth;

    }

};

或者（双端队列）：

/\*\*

\* 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:

int maxDepth(TreeNode\* root) {

if(root==NULL) return 0;

deque<TreeNode\*> q;

q.push\_back(root);

int deep=0;

while(!q.empty())

{

deep++;

int num=q.size();

for(int i=1;i<=num;i++)

{

TreeNode\* p=q.front();

q.pop\_front();

if(p->left) q.push\_back(p->left);

if(p->right) q.push\_back(p->right);

}

}

return deep;

}

};