# 题目

给定一个二叉树，编写一个函数来获取这个树的最大宽度。树的宽度是所有层中的最大宽度。这个二叉树与满二叉树（full binary tree）结构相同，但一些节点为空。

每一层的宽度被定义为两个端点（该层最左和最右的非空节点，两端点间的null节点也计入长度）之间的长度。

示例 1:

输入:

1

/ \

3 2

/ \ \

5 3 9

输出: 4

解释: 最大值出现在树的第3层，宽度为4 (5,3,null,9)。

示例 2:

输入:

1

/

3

/ \

5 3

输出: 2

解释: 最大值出现在树的第3层，宽度为2 (5,3)。

示例 3:

输入:

1

/ \

3 2

/

5

输出: 2

解释: 最大值出现在树的第2层，宽度为2 (3,2)。

示例 4:

输入:

1

/ \

3 2

/ \

5 9

/ \

6 7

输出: 8

解释: 最大值出现在树的第4层，宽度为8 (6,null,null,null,null,null,null,7)。

注意: 答案在32位有符号整数的表示范围内。

# 分析

## 方法一：宽度优先遍历

**思路：**

因为涉及一层一层的遍历，基本想到就是广度优先

1、宽度优先会使用一个queue，而这个queue就可以帮助我们计算出目前层的宽度

2、数值转换(关键一步，减少内存，加速计算)：为了计算宽度，我们需要记录每个节点出现的编号，比如[1,3,2]这样的二叉树，对应1->0 3->0 2->1

3、下一层，基于当前层的编号推出下一层：

left: parent->val \* 2

right: parent->val \* 2 +1

4、当前层的宽度：back-val - front->val + 1

如front->0 back=4，那么宽度就是5

5、额外，为了避免数字过大，我们会让val每次基于queue front编号做缩小

queue front的编号是100

front缩小为100-100=0，后续每个编号都需要减去100

**代码：**

class Solution {

public:

int widthOfBinaryTree(TreeNode\* root) {

if (root == nullptr) return 0;

// 保存最大的宽度

int ret = 0;

// 队列用于广度优先遍历

queue<TreeNode\*> q;

// 对于根节点的编号为0

root->val = 0;

q.push(root);

while (!q.empty())

{

// 基于目前队列头和尾获得当前层的宽度

ret = max(ret, q.back()->val - q.front()->val + 1);

// 编号缩小的差值

int offset = q.front()->val;

// 遍历完当前层

int n = q.size();

for (int i = 0; i < n; ++i)

{

TreeNode\* curr = q.front();

q.pop();

// 缩小数值

curr->val -= offset;

if (curr->left)

{

// 转换为对应的编号

curr->left->val = curr->val\*2;

q.push(curr->left);

}

if (curr->right)

{

// 转换为对应的编号

curr->right->val = curr->val\*2+1;

q.push(curr->right);

}

}

}

return ret;

}

};

## 方法二：深度优先遍历

**思路：**

按照深度优先的顺序，我们记录每个节点的position 。对于每一个深度，第一个到达的位置会被记录在left[depth] 中。

然后对于每一个节点，它对应这一层的可能宽度是pos - left[depth] + 1。我们将每一层这些可能的宽度去一个最大值就是答案。

**代码：**

class Solution {

int ans;

Map<Integer, Integer> left;

public int widthOfBinaryTree(TreeNode root) {

ans = 0;

left = new HashMap();

dfs(root, 0, 0);

return ans;

}

public void dfs(TreeNode root, int depth, int pos) {

if (root == null) return;

left.computeIfAbsent(depth, x-> pos);

ans = Math.max(ans, pos - left.get(depth) + 1);

dfs(root.left, depth + 1, 2 \* pos);

dfs(root.right, depth + 1, 2 \* pos + 1);

}

}

**复杂度分析：**

时间复杂度：O(N)，其中N是树中节点的数目，我们需要遍历每个节点。

空间复杂度：O(N)，这部分空间是因为我们DFS递归过程中有N层的栈。