# 题目

给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。

你需要将他们合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为NULL的节点将直接作为新二叉树的节点。

示例 1:

输入:

Tree 1 Tree 2

1 2

/ \ / \

3 2 1 3

/ \ \

5 4 7

输出:

合并后的树:

3

/ \

4 5

/ \ \

5 4 7

注意: 合并必须从两个树的根节点开始。

# 分析

## 方法一：深度优先遍历

**思路：**

可以使用深度优先搜索合并两个二叉树。从根节点开始同时遍历两个二叉树，并将对应的节点进行合并。

两个二叉树的对应节点可能存在以下三种情况，对于每种情况使用不同的合并方式。

如果两个二叉树的对应节点都为空，则合并后的二叉树的对应节点也为空；

如果两个二叉树的对应节点只有一个为空，则合并后的二叉树的对应节点为其中的非空节点；

如果两个二叉树的对应节点都不为空，则合并后的二叉树的对应节点的值为两个二叉树的对应节点的值之和，此时需要显性合并两个节点。

对一个节点进行合并之后，还要对该节点的左右子树分别进行合并。这是一个递归的过程。

**代码：**

class Solution {

public:

TreeNode\* mergeTrees(TreeNode\* t1, TreeNode\* t2) {

if (t1 == nullptr) {

return t2;

}

if (t2 == nullptr) {

return t1;

}

auto merged = new TreeNode(t1->val + t2->val);

merged->left = mergeTrees(t1->left, t2->left);

merged->right = mergeTrees(t1->right, t2->right);

return merged;

}

};

复杂度分析：

时间复杂度：O(min(m,n))，其中m和n分别是两个二叉树的节点个数。对两个二叉树同时进行深度优先搜索，只有当两个二叉树中的对应节点都不为空时才会对该节点进行显性合并操作，因此被访问到的节点数不会超过较小的二叉树的节点数。

空间复杂度：O(min(m,n))，其中m和n分别是两个二叉树的节点个数。空间复杂度取决于递归调用的层数，递归调用的层数不会超过较小的二叉树的最大高度，最坏情况下，二叉树的高度等于节点数。

## 方法二：广度优先遍历

**思路：**

也可以使用广度优先搜索合并两个二叉树。首先判断两个二叉树是否为空，如果两个二叉树都为空，则合并后的二叉树也为空，如果只有一个二叉树为空，则合并后的二叉树为另一个非空的二叉树。

如果两个二叉树都不为空，则首先计算合并后的根节点的值，然后从合并后的二叉树与两个原始二叉树的根节点开始广度优先搜索，从根节点开始同时遍历每个二叉树，并将对应的节点进行合并。

使用三个队列分别存储合并后的二叉树的节点以及两个原始二叉树的节点。初始时将每个二叉树的根节点分别加入相应的队列。每次从每个队列中取出一个节点，判断两个原始二叉树的节点的左右子节点是否为空。如果两个原始二叉树的当前节点中至少有一个节点的左子节点不为空，则合并后的二叉树的对应节点的左子节点也不为空。对于右子节点同理。

如果合并后的二叉树的左子节点不为空，则需要根据两个原始二叉树的左子节点计算合并后的二叉树的左子节点以及整个左子树。考虑以下两种情况：

如果两个原始二叉树的左子节点都不为空，则合并后的二叉树的左子节点的值为两个原始二叉树的左子节点的值之和，在创建合并后的二叉树的左子节点之后，将每个二叉树中的左子节点都加入相应的队列；

如果两个原始二叉树的左子节点有一个为空，即有一个原始二叉树的左子树为空，则合并后的二叉树的左子树即为另一个原始二叉树的左子树，此时也不需要对非空左子树继续遍历，因此不需要将左子节点加入队列。

对于右子节点和右子树，处理方法与左子节点和左子树相同。

**代码：**

class Solution {

public:

TreeNode\* mergeTrees(TreeNode\* t1, TreeNode\* t2) {

if (t1 == nullptr) {

return t2;

}

if (t2 == nullptr) {

return t1;

}

auto merged = new TreeNode(t1->val + t2->val);

auto q = queue<TreeNode\*>();

auto queue1 = queue<TreeNode\*>();

auto queue2 = queue<TreeNode\*>();

q.push(merged);

queue1.push(t1);

queue2.push(t2);

while (!queue1.empty() && !queue2.empty()) {

auto node = q.front(), node1 = queue1.front(), node2 = queue2.front();

q.pop();

queue1.pop();

queue2.pop();

auto left1 = node1->left, left2 = node2->left, right1 = node1->right, right2 = node2->right;

if (left1 != nullptr || left2 != nullptr) {

if (left1 != nullptr && left2 != nullptr) {

auto left = new TreeNode(left1->val + left2->val);

node->left = left;

q.push(left);

queue1.push(left1);

queue2.push(left2);

} else if (left1 != nullptr) {

node->left = left1;

} else if (left2 != nullptr) {

node->left = left2;

}

}

if (right1 != nullptr || right2 != nullptr) {

if (right1 != nullptr && right2 != nullptr) {

auto right = new TreeNode(right1->val + right2->val);

node->right = right;

q.push(right);

queue1.push(right1);

queue2.push(right2);

} else if (right1 != nullptr) {

node->right = right1;

} else {

node->right = right2;

}

}

}

return merged;

}

};

**复杂度分析：**

时间复杂度：O(min(m,n))，其中m和n分别是两个二叉树的节点个数。对两个二叉树同时进行广度优先搜索，只有当两个二叉树中的对应节点都不为空时才会访问到该节点，因此被访问到的节点数不会超过较小的二叉树的节点数。

空间复杂度：O(min(m,n))，其中m和n分别是两个二叉树的节点个数。空间复杂度取决于队列中的元素个数，队列中的元素个数不会超过较小的二叉树的节点数。