# 题目

给定两个二叉树，编写一个函数来检验它们是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

**示例 1:**

输入: 1 1

/ \ / \

2 3 2 3

[1,2,3], [1,2,3]

**示例 2:**

输入: 1 1

/ \

2 2

[1,2], [1,null,2]

输出: false

**示例 3:**

输入: 1 1

/ \ / \

2 1 1 2

[1,2,1], [1,1,2]

输出: false

# 分析

## 方法一：递归

**思路：**最简单的策略是使用递归。首先判断p和q是不是None，然后判断它们的值是否相等。

若以上判断通过，则递归对子结点做同样操作。

**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:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

if (nullptr == p && nullptr == q) return true;

if (nullptr == p || nullptr == q) return false;

if (p->val != q->val) return false;

return isSameTree(p->left,q->left) &&

isSameTree(p->right,q->right);

}

};

**复杂度：**

时间复杂度：O(N)，其中N是树的结点数，因为每个结点都访问一次。

空间复杂度：最优情况（完全平衡二叉树）时为O(log(N))，最坏情况下（完全不平衡二叉树）时为O(N)，用于维护递归栈。

## 方法二：迭代

### 思路一：利用栈

可以采用深度优先遍历的方式逐一比较节点（两个树同一个位置的节点同时入栈，同时出栈，然后比较，只要出现不符合的直接返回），此时使用栈。

**代码：**

class Solution {

public:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

stack<TreeNode\*> S; // 也可以采用两个栈

S.push(q);

S.push(p); //注意入栈的顺序

TreeNode\* p\_temp;

TreeNode\* q\_temp;

while(!S.empty()){ //栈非空（而不是nullptr!=S）

p\_temp=S.top();

S.pop();

q\_temp=S.top();

S.pop();

if(p\_temp==NULL&&q\_temp==NULL)continue;

//为空则继续，不能返回true（与递归不同！）

else if(p\_temp!=NULL&&q\_temp==NULL)return false;

else if(p\_temp==NULL&&q\_temp!=NULL)return false;

else if(p\_temp->val!=q\_temp->val)return false;

else{

S.push(q\_temp->right);

S.push(p\_temp->right);

//右孩子先入栈，不能一个左孩子一个右孩子，如果比较p的左孩子和q的右孩子，则只需要适当调整一下入栈顺序即可

S.push(q\_temp->left);

S.push(p\_temp->left);

}

}

return true; //栈为空的情况（容易遗漏！）

}

};

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:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

stack<TreeNode\*> stk;

stk.push(q);

stk.push(p);

TreeNode \*p\_tmpNode;

TreeNode \*q\_tmpNode;

while(!stk.empty())

{

p\_tmpNode = stk.top();

stk.pop();

q\_tmpNode = stk.top();

stk.pop();

if(nullptr==p\_tmpNode && nullptr==q\_tmpNode) continue;

else if(nullptr==p\_tmpNode || nullptr==q\_tmpNode) return false;

else if(p\_tmpNode->val != q\_tmpNode->val) return false;

else

{

stk.push(q\_tmpNode->left);

stk.push(p\_tmpNode->left);

stk.push(q\_tmpNode->right);

stk.push(p\_tmpNode->right);

}

}

return true;

}

};

**复杂度：**

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。

### 思路二：利用队列

**思路：**

可以采用广度优先的方式遍历，此时使用队列。

从根开始，每次迭代将当前结点从双向队列中弹出。然后，进行方法一中的判断：

p和q不是None，

p.val等于q.val，

若以上均满足，则压入子结点。

**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:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

if(!p && !q) return true;

queue<TreeNode\*> q1,q2;

q1.push(p);

q2.push(q);

while(!q1.empty() && !q2.empty()){

TreeNode\* node1 = q1.front();

TreeNode\* node2 = q2.front();

q1.pop();

q2.pop();

if((!node1 && node2) || (node1 && !node2)) return false;

if(node1 && node2){

if(node1->val != node2->val) return false;

q1.push(node1->left);

q1.push(node1->right);

q2.push(node2->left);

q2.push(node2->right);

}

}

return true;

}

};

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:

bool isSameTree(TreeNode\* p, TreeNode\* q) {

if(nullptr==p && nullptr==q)

return true;

queue<TreeNode\*> que1,que2;

que1.push(p);

que2.push(q);

while(!que1.empty() && !que2.empty())

{

TreeNode \*node1 = que1.front();

TreeNode \*node2 = que2.front();

que1.pop();

que2.pop();

if(nullptr==node1 && nullptr==node2) continue;

else if(nullptr==node1 || nullptr==node2) return false;

else if(node1->val != node2->val) return false;

else

{

que1.push(node1->left);

que1.push(node1->right);

que2.push(node2->left);

que2.push(node2->right);

}

}

return true;

}

};

**复杂度：**

时间复杂度：O(N)，其中N是树的结点数，因为每个结点都访问一次。

空间复杂度：最优情况（完全平衡二叉树）时为O(log(N))，最坏情况下（完全不平衡二叉树）时为O(N)，用于维护双向队列。