# 题目

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树[1,2,2,3,4,4,3] 是对称的。

1

/ \

2 2

/ \ / \

3 4 4 3

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

1

/ \

2 2

\ \

3 3

**进阶：**

你可以运用递归和迭代两种方法解决这个问题吗？

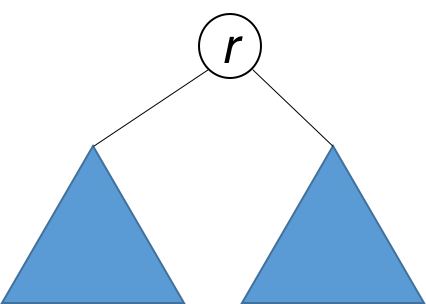
类似题目：剑指offer 28

# 分析

## 方法一：递归

**思路：**可以采用判断二者是否完全一样的思路改写，即判断二叉树的左右子树是否互为镜像。

如果一个树的左子树与右子树镜像对称，那么这个树是对称的。

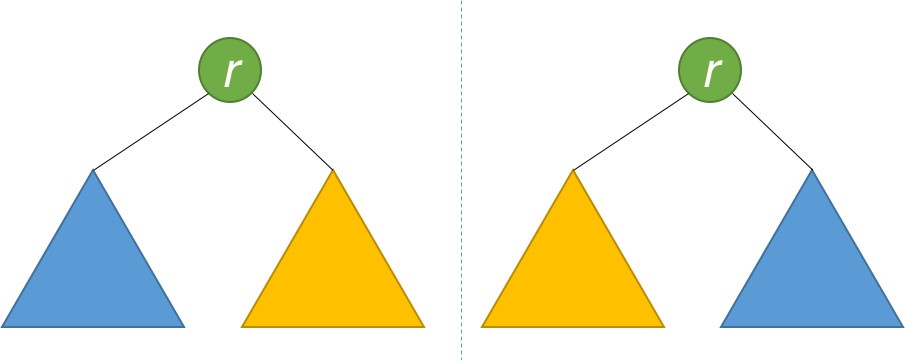


因此，该问题可以转化为：两个树在什么情况下互为镜像？

如果同时满足下面的条件，两个树互为镜像：

1、它们的两个根结点具有相同的值。

2、每个树的右子树都与另一个树的左子树镜像对称。



就像人站在镜子前审视自己那样。镜中的反射与现实中的人具有相同的头部，但反射的右臂对应于人的左臂，反之亦然。

**说明：**与LeetCode100判断两个树是否完全一样类似，需要修改一下入参。

这种判断是否一致的树，都是采用拆分两个输入参数的形式，这样比较容易理解和编码。

**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 {

private:

bool isMirror(TreeNode\* p,TreeNode\* q)

{

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

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

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

return isMirror(p->left,q->right) && isMirror(p->right,q->left);

/\*

这里是求镜像，如果判断完全一样则为：

isMirror(p->left,q->left) && isMirror(p->right,q-> right);

\*/

/\*

或者可以将最后的else if和return合成一个：

return (p->val != q->val)

&& isMirror(p->left,q->right)

&& isMirror(p->right,q->left);

\*/

}

public:

bool isSymmetric(TreeNode\* root) {

return isMirror(root,root);

}

};

**复杂度：**

时间复杂度：O(n)，因为我们遍历整个输入树一次，所以总的运行时间为O(n)，其中n是树中结点的总数。

空间复杂度：递归调用的次数受树的高度限制。在最糟糕情况下，树是线性的，其高度为O(n)。因此，在最糟糕的情况下，由栈上的递归调用造成的空间复杂度为O(n)。

## 方法二：迭代

### 思路一：深度搜索+栈

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 isSymmetric(TreeNode\* root) {

std::stack<TreeNode\*> tmpArr;

tmpArr.push(root);

tmpArr.push(root);

TreeNode\* tmpLeft;

TreeNode\* tmpRight;

while(!tmpArr.empty())

{

tmpLeft = tmpArr.top();

tmpArr.pop();

tmpRight = tmpArr.top();

tmpArr.pop();

if(nullptr == tmpLeft && nullptr == tmpRight)continue;

else if(nullptr == tmpLeft || nullptr == tmpRight)return false;

else if(tmpLeft->val != tmpRight->val)return false;

else

{

tmpArr.push(tmpLeft->left);

tmpArr.push(tmpRight->right);

tmpArr.push(tmpLeft->right);

tmpArr.push(tmpRight->left);

}

}

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：广度搜索+队列

除了递归的方法外，我们也可以利用队列进行迭代。队列中每两个连续的结点应该是相等的，而且它们的子树互为镜像。最初，队列中包含的是root以及root。该算法的工作原理类似于BFS，但存在一些关键差异。每次提取两个结点并比较它们的值。然后，将两个结点的左右子结点按相反的顺序插入队列中。当队列为空时，或者我们检测到树不对称（即从队列中取出两个不相等的连续结点）时，该算法结束。

**代码：**

/\*\*

 \* 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 isSymmetric(TreeNode\* root) {

        if(nullptr==root)

            return true;

        queue<TreeNode\*> que1,que2;

        que1.push(root);

        que2.push(root);

        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);

                que2.push(node2->right);

                que1.push(node1->right);

                que2.push(node2->left);

            }

        }

        return true;

    }

};

**复杂度：**

时间复杂度：O(n)，因为我们遍历整个输入树一次，所以总的运行时间为O(n)，其中n是树中结点的总数。

空间复杂度：搜索队列需要额外的空间。在最糟糕情况下，我们不得不向队列中插入 O(n)个结点。因此，空间复杂度为O(n)。