

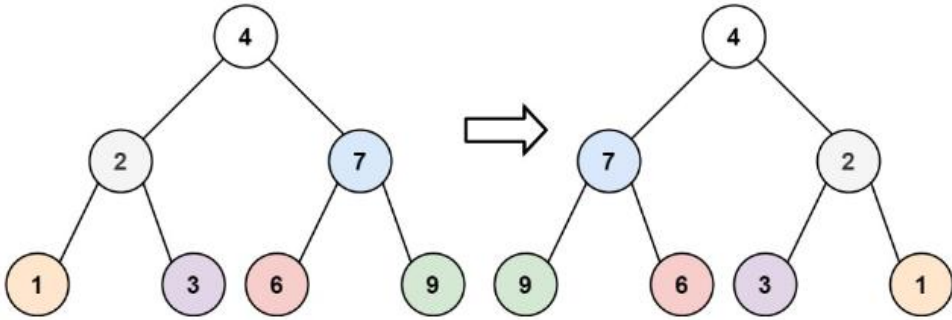
题目：

LeetCode 热题 HOT 100

题目描述 | 笔记 × | 题解 | 提交记录

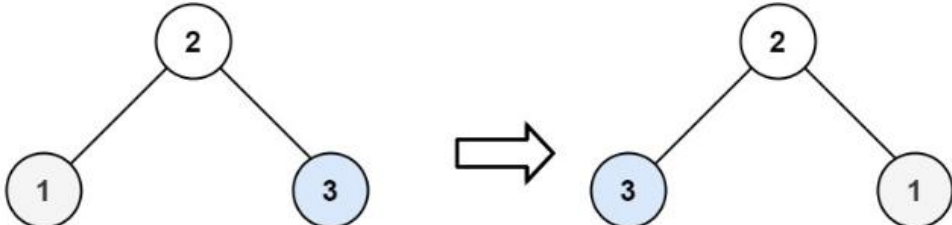
给你一棵二叉树的根节点 `root`，翻转这棵二叉树，并返回其根节点。

示例 1:



输入: `root = [4,2,7,1,3,6,9]`
输出: `[4,7,2,9,6,3,1]`

示例 2:



输入: `root = [2,1,3]`
输出: `[2,3,1]`

示例 3:

输入: `root = []`
输出: `[]`

解题思路一：

递归：

每次进入函数时，形参改为 `root` 的左右子树，递归交换即可

```
#include<iostream>
#include<stack>
using namespace std;
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) {}
};
//递归
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == nullptr) {
            return nullptr;
        }
        //递归交换左右子树
        TreeNode* left = invertTree(root->left);
        TreeNode* right = invertTree(root->right);
        root->left = right;
        root->right = left;
        return root;
    }
};

```

解题思路二：

辅助栈：

从栈顶元素开始入栈，结束条件是栈空，对于每个节点，先弹出当前栈顶，如果它的左右孩子节点选择不为空的入栈，交换它的左右孩子，然后当前栈顶元素变成下一个循环处理的对象，直到栈为空，代表所有元素都处理完毕

代码：

```

class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == nullptr) return nullptr;
        stack<TreeNode*> stack;
        stack.push(root); //根节点压入栈顶
        //从左子树开始深度优先遍历交换左右孩子
        while (!stack.empty())

```

```
{
    TreeNode* node = stack.top();
    stack.pop();
    if (node->left != nullptr) stack.push(node->left);
    if (node->right != nullptr) stack.push(node->right);
    //交换当前节点的左右孩子
    TreeNode* tmp = node->left;
    node->left = node->right;
    node->right = tmp;

}
return root;
}
};
```

142. 环形链表 II

已解答

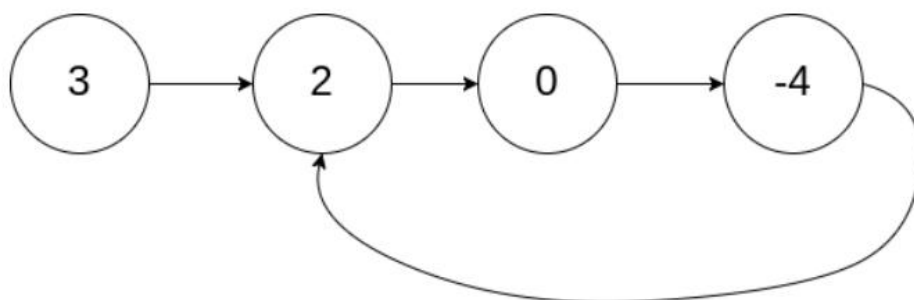
中等
相关标签
相关企业
A+

给定一个链表的头节点 `head`，返回链表开始入环的第一个节点。如果链表无环，则返回 `null`。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 `pos` 是 `-1`，则在该链表中没有环。注意：`pos` 不作为参数进行传递，仅仅是为了标识链表的实际情况。

不允许修改 链表。

示例 1:



输入: `head = [3,2,0,-4]`, `pos = 1`

输出: 返回索引为 1 的链表节点

解释: 链表中有一个环，其尾部连接到第二个节点。

示例 2:

解题思路一:

快慢指针:

初始化快慢指针等于 `head`，快指针走的速度是慢指针的两倍，如果他俩能够相遇，说明存在环，这时它们第一次相遇，但是我们并不知道相遇的节点是在环的入口节点还是其他节点。由于速度关系，快指针走了 $2n$ 个环，慢指针走了 n 个环，假设环的长度是 b ，头节点到入口的距离是 a ，那么从头节点出发，走到环的入口节点可能的距离是 $k=a+nb$ 。这时，我们选择一个指针指向 `head`，假设我们选择把慢指针设置为 `head`，那么这时快指针的走距离是 $2nb$ ，慢指针走的距离是 0 ，两个指针每次只走一步，在走 a 步过后，两者相遇，慢指针走了 $a+0b$ ，快指针走了 $2nb+a$ ，此时两者相遇的节点就是入口节点，返回两者中任意一个即可

```
class Solution {
```

```

public:
    ListNode* detectCycle(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        //非循环链表会有 fast 为空直接退出
        while (fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            //存在环一定会相遇
            if (fast == slow) {
                slow = head;
                //循环结束条件是第二次相遇
                while (fast != slow) {
                    fast = fast->next;
                    slow = slow->next;
                }
                return slow; //fast 也可以
            }
        }
        return nullptr; //没有环返回空指针
    }
};

```

解题思路二:

哈希表:

使用 unordered_set 存储访问过的节点, 由于值唯一, 每次遍历节点时检查对应的访问节点是否有记录, 有记录说明就是环, 第一个有记录过的节点就是环的入口节点

```

class Solution {
public:
    ListNode* detectCycle(ListNode* head) {
        unordered_set<ListNode*> visited;
        while (head != nullptr) {
            if (visited.count(head)) {
                return head;
            }
            visited.insert(head);
            head = head->next;
        }
        return nullptr;
    }
};

```

