# 题目

给定一个二叉树，返回它的中序遍历。

**示例:**

**输入:** [1,null,2,3]

1

\

2

/

3

**输出:** [1,3,2]

**进阶:**递归算法很简单，你可以通过迭代算法完成吗？

# 分析

## 方法一：递归

**思路：**

第一种解决方法是使用递归。这是经典的方法，直截了当。我们可以定义一个辅助函数来实现递归。

中序遍历是先遍历左子树，然后访问根节点，然后遍历右子树。

**代码：**

class Solution {

public:

vector<int> ans;

vector<int> inorderTraversal(TreeNode\* root) {

if(root != NULL) {

inorderTraversal(root -> left); //左子树

ans.push\_back(root -> val); //根节点

inorderTraversal(root -> right); //右子树

}

return ans;

}

};

或：

class Solution {

private:

vector<int> ans; // 用于存储结果

public:

vector<int> inorderTraversal(TreeNode\* root) {

// 空节点直接返回空结果

if (!root) return {};

// 当前节点是叶子节点，直接加入答案

if (root && !root->left && !root->right) {

ans.push\_back(root->val);

return ans;

}

// 递归遍历左子树

inorderTraversal(root->left);

// 访问当前节点

ans.push\_back(root->val);

// 递归遍历右子树

inorderTraversal(root->right);

return ans;

}

};

分析：

ans 是类成员变量，可能有副作用。这个代码实现有个潜在的问题：

1. ans 是类成员变量；
2. 每次递归都用的是同一个 ans；
3. 如果你用同一个 Solution实例调用多次 inorderTraversal()，结果会叠加上一次的结果。

假设你在主程序里写了这样两次调用：

Solution sol;

TreeNode\* root1 = ...; // 第一棵树

TreeNode\* root2 = ...; // 第二棵树

vector<int> result1 = sol.inorderTraversal(root1);

vector<int> result2 = sol.inorderTraversal(root2);

期望行为：

result1 是对 root1 的中序遍历；

result2 是对 root2 的中序遍历，不受 result1 影响。

实际行为（当前写法）：

你的代码中 ans 是类成员：

class Solution {

private:

vector<int> ans; // 全局的，属于对象实例的一部分

public:

vector<int> inorderTraversal(TreeNode\* root) {

...

return ans;

}

};

所以在第一次调用 inorderTraversal(root1) 时，ans 被填充为 [a1, a2, a3]。

而**第二次调用时，这个 ans 没有被清空！**所以：

第二次调用等于在原来的 ans 上继续追加；

result2 变成了 [a1, a2, a3, b1, b2, b3]；

它本应该是 [b1, b2, b3]！

本质问题：类成员变量跨调用“记住”了状态

1）类中的 ans 是持久的，属于对象；

2）函数多次调用复用这个变量，导致数据“串台”；

3）这破坏了函数的“幂等性和纯粹性” —— 同样输入不一定返回同样结果。

正确做法：

✅ 方法1：每次手动清空 ans

vector<int> inorderTraversal(TreeNode\* root) {

ans.clear(); // 清空旧状态

...

}

但这样容易遗漏，不推荐。

✅ 方法2（推荐）：使用局部变量 + helper 函数**【Leetcode写法】**

class Solution {

public:

vector<int> inorderTraversal(TreeNode\* root) {

vector<int> ans;

helper(root, ans); // 等于封装了一层，传入局部变量，递归调用helper

return ans;

}

private:

void helper(TreeNode\* node, vector<int>& ans) {

if (!node) return;

// one node

if (root && !root->left && !root->right) {

ans.push\_back(root->val);

return;

}

helper(node->left, ans);

ans.push\_back(node->val);

helper(node->right, ans);

}

};

这样 ans 是局部变量，不会受多次调用干扰，每次调用都是干净的。

或：

class Solution {

public:

void inorder(TreeNode \*root, vector<int> &res) {

if (root == nullptr) {

return;

}

// one node

if (root && !root->left && !root->right) {

ans.push\_back(root->val);

return;

}

inorder(root->left, res);

res.push\_back(root->val);

inorder(root->right, res);

// return;

}

vector<int> inorderTraversal(TreeNode \*root) {

vector<int> res;

inorder(root, res);

return res;

}

};

## 方法二：迭代/栈

**思路：**

方法一的递归函数我们也可以用迭代的方式实现，两种方式是等价的，区别在于递归的时候隐式地维护了一个栈，而我们在迭代的时候需要显式地将这个栈模拟出来，其他都相同。

每到一个节点A，因为根的访问在中间，将A入栈。然后遍历左子树，接着访问A，最后遍历右子树。

在访问完A后，A就可以出栈了。因为A和其左子树都已经访问完成。

**说明：**

深度优先遍历，基于栈实现（广度优先遍历，基于队列实现）。

广度优先遍历中，是按照根节点，“一层一层”的遍历，这里所述的“按层”就是一种顺序，所以底层实现是队列。

深度优先遍历中，是按照从“自左向右”的顺序，这里所述的“左右”就是一种顺序，底层需要是栈，因为先入栈的左树，后出栈。

**伪代码：**

栈S;

p= root;

while(p || S不空){

while(p){

p入S;

p = p的左子树;

}

p = S.top

出栈;

访问p;

p = p的右子树;

}

**代码：**

vector<int> inorderTraversal(TreeNode\* root) {

stack<TreeNode\*> S;

vector<int> v;

TreeNode\* rt = root;

while(rt || S.size()){

//第二个判断条件S.size()针对空的节点

while(rt){

S.push(rt);

rt=rt->left; //遍历左子树

}

//左子树全部依次入栈，则最后一个就是树的顶，下一步top访问即可

rt=S.top();

S.pop();

v.push\_back(rt->val);//这两行可以互换位置

rt=rt->right;

}

return v;

}

**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:

vector<int> inorderTraversal(TreeNode\* root) {

stack<TreeNode\*> stk;

vector<int> vec;

TreeNode \*tmpNode = root;

while(tmpNode || !stk.empty())

{

while(tmpNode)

{

stk.push(tmpNode);

tmpNode = tmpNode->left;

}

tmpNode = stk.top();

stk.pop();

vec.push\_back(tmpNode->val);

tmpNode = tmpNode->right;

}

return vec;

}

};

# 总结

## 前序遍历迭代算法

二叉树的遍历，整体上看都是好理解的。

三种遍历的迭代写法中，数前序遍历最容易理解。

**递归思路：**先树根，然后左子树，然后右子树。每棵子树递归。

在迭代算法中，思路演变成，每到一个节点 A，就应该立即访问它。

因为，每棵子树都先访问其根节点。对节点的左右子树来说，也一定是先访问根。

在 A 的两棵子树中，遍历完左子树后，再遍历右子树。

因此，在访问完根节点后，遍历左子树前，要将右子树压入栈。

**思路：**

栈S;

p= root;

while(p || S不空){

while(p){

访问p节点；

p的右子树入S;

p = p的左子树;

}

p = S栈顶弹出;

}

**代码：**

class Solution {

public:

vector<int> preorderTraversal(TreeNode\* root) {

stack<TreeNode\*> st;

vector<int> result;

if (root == NULL) return result;

st.push(root);

while (!st.empty()) {

TreeNode\* node = st.top(); // 中

st.pop();

result.push\_back(node->val);

if (node->right) st.push(node->right); // 右（空节点不入栈）

if (node->left) st.push(node->left); // 左（空节点不入栈）

}

return result;

}

};

或：

vector<int> preorderTraversal(TreeNode\* root) {

stack<TreeNode\*> S;

vector<int> v;

TreeNode\* rt = root;

while(rt || S.size()){

while(rt){

S.push(rt->right);

v.push\_back(rt->val);

rt=rt->left;

}

rt=S.top();

S.pop();

}

return v;

}

## 中序遍历迭代算法

中序遍历与前序和后序遍历不同，前序遍历可以直接将节点存入到栈中，因为扫描顺序与入栈一致，但是中序不同，需要先达到叶子节点才可以，所以一开始需要深度优先遍历。

代码：

vector<int> inorderTraversal(TreeNode\* root) {

stack<TreeNode\*> S;

vector<int> v;

TreeNode\* rt = root;

while(rt || S.size()){

//第二个判断条件S.size()针对空的节点

while(rt){

S.push(rt);

rt=rt->left; //遍历左子树

}

//左子树全部依次入栈，则最后一个就是树的顶，下一步top访问即可

rt=S.top();

S.pop();

v.push\_back(rt->val);//这两行可以互换位置

rt=rt->right;

}

return v;

}

**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:

vector<int> inorderTraversal(TreeNode\* root) {

stack<TreeNode\*> stk;

vector<int> vec;

TreeNode \*tmpNode = root;

while(tmpNode || !stk.empty())

{

while(tmpNode)

{

stk.push(tmpNode);

tmpNode = tmpNode->left;

}

tmpNode = stk.top();

stk.pop();

vec.push\_back(tmpNode->val);

tmpNode = tmpNode->right;

}

return vec;

}

};

## 后序遍历迭代算法

二叉树的后序遍历有两种方法：

第一种比第二种要容易理解，但多了个结果逆序的过程。

### 方法一

第一种方法：

我们可以用与前序遍历相似的方法完成后序遍历。

后序遍历与前序遍历相对称。

**思路：**每到一个节点 A，就应该立即访问它。 然后将左子树压入栈，再次遍历右子树。

遍历完整棵树后，结果序列逆序即可。

**伪代码：**

栈S;

p= root;

while(p || S不空){

while(p){

访问p节点；

p的左子树入S;

p = p的右子树;

}

p = S栈顶弹出;

}

结果序列逆序;

**代码：**

vector<int> postorderTraversal(TreeNode\* root) {

stack<TreeNode\*> S;

vector<int> v;

TreeNode\* rt = root;

while(rt || S.size()){

while(rt){

S.push(rt->left);

v.push\_back(rt->val);

rt=rt->right;

}

rt=S.top();S.pop();

}

reverse(v.begin(),v.end());

return v;

}

### 方法二

第二种方法：

按照左子树-根-右子树的方式，将其转换成迭代方式。

**思路：**每到一个节点A，因为根要最后访问，将其入栈。然后遍历左子树，遍历右子树，最后返回到A。

但是出现一个问题，无法区分是从左子树返回，还是从右子树返回。

因此，给A节点附加一个标记T。在访问其右子树前，T置为True。之后子树返回时，当T为True表示从右子树返回，否则从左子树返回。

当T为false时，表示A的左子树遍历完，还要访问右子树。

同时，当T为True 时，表示A的两棵子树都遍历过了，要访问A了。并且在A访问完后，A这棵子树都访问完成了。

**伪代码：**

栈S;

p= root;

T<节点,True/False> : 节点标记;

while(p || S不空){

while(p){

p入S;

p = p的左子树;

}

while(S不空 且 T[S.top] = True){

访问S.top;

S.top出S;

}

if(S不空){

p = S.top 的右子树;

T[S.top] = True;

}

}

**代码：**

vector<int> postorderTraversal(TreeNode\* root) {

stack<TreeNode\*> S;

unordered\_map<TreeNode\*,int> done;

vector<int> v;

TreeNode\* rt = root;

while(rt || S.size()){

while(rt){

S.push(rt);

rt=rt->left;

}

while(S.size() && done[S.top()]){

v.push\_back(S.top()->val);

S.pop();

}

if(S.size()){

rt=S.top()->right;

done[S.top()]=1;

}

}

return v;

}