# 题目

给定一个二叉树，返回它的后序遍历。

**示例:**

输入: [1,null,2,3]

1

\

2

/

3

输出: [3,2,1]

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：递归法

/\*\*

\* 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:

vector<int> val; // 使用临时变量，不推荐

public:

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

if (NULL == root)

return val;

postorderTraversal(root->left);

postorderTraversal(root->right);

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

return val;

}

};

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:

void postorderTraversal(TreeNode\* root, vector<int>& ans) {

if (!root) return;

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

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

// return;

//} // 可以删除

postorderTraversal(root->left, ans);

postorderTraversal(root->right, ans);

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

return;

}

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

vector<int> ans;

postorderTraversal(root, ans);

return ans;

}

};

**复杂度分析：**

时间复杂度：O(n)，其中n是二叉搜索树的节点数。每一个节点恰好被遍历一次。

空间复杂度：O(n)，为递归过程中栈的开销，平均情况下为O(log n)，最坏情况下树呈现链状，为O(n)。

## 方法二：迭代法

**思路：**

**代码：**

class Solution {

public:

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

vector<int> res;

if (root == nullptr) {

return res;

}

stack<TreeNode \*> stk;

TreeNode \*prev = nullptr;

while (root != nullptr || !stk.empty()) {

while (root != nullptr) {

stk.emplace(root);

root = root->left;

}

root = stk.top();

stk.pop();

if (root->right == nullptr || root->right == prev) {

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

prev = root;

root = nullptr;

} else {

stk.emplace(root);

root = root->right;

}

}

return res;

}

};

**复杂度分析：**

时间复杂度：O(n)，其中n是二叉搜索树的节点数。每一个节点恰好被遍历一次。

空间复杂度：O(n)，为迭代过程中显式栈的开销，平均情况下为O(logn)，最坏情况下树呈现链状，为O(n)。

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：前序遍历+逆序

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; //与前序遍历稍有不同

}

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

}

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

return v;

}

### 方法二：迭代

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;

}

# 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**

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;

}