代码实现-二叉树的遍历

# 三种遍历方式：

## 回顾定义

### **NLR：前序遍历(Preorder Traversal 亦称（先序遍历））**

——**访问根结点的操作**发生在遍历其左右子树之前；遍历顺序规则为【根左右】。

### **LNR：中序遍历(Inorder Traversal)**

——**访问根结点的操作**发生在遍历其左右子树之中（间）；遍历顺序规则为【左根右】

### LRN：后序遍历(**Postorder Traversal**)

——**访问根结点的操作**发生在遍历其左右子树之后；遍历顺序规则为【左右根】

## 英文

**traversal** 英[træ'vɜ:sl] 美['trævɜ:səl]n. 横越，横断物，（横向）往返移动;**遍历**

**recursion** 英 [rɪˈkɜ:ʃn] 美 [rɪˈkɜ:rʃn] n. 递推;递归，递归式

**recursive** 英 [rɪˈkɜ:sɪv] 美 [rɪˈkɜ:rsɪv] adj. 回归的，递归的

命名规范：**先序遍历preOrder或preOrderTraversal**

**中序遍历inOrder或inOrderTraversal**

**后序遍历postOrder或postOrderTraversal**

方法: **递归方法**：preOrderRecur、inOrderRecur、postOrderRecur

**非递归方法**：preOrderUnRecur、inOrderUnRecur、postOrderUnRecur

## 遍历书写技巧：用递归思想去写。

从根节点root开始逐步对二叉树进行分割成三部分：root、左子树、右子树。

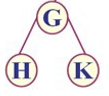
用这样就一直用**递归思想去**写出这个结果。

先序遍历: root ==>**左子树**==>**右子树 A(B(CD))(E(F(GHK)))去括号为**ABCDEFGHK

中序遍历: **左子树**==>root ==>**右子树 (B(DC))A(E((HGK)F))去括号为**BDCAEHGKF

后序遍历: **左子树**==>**右子树** ==>root **((DC)B) (((HKG)F)E) A去括号为**DCBHKGFEA

最小单元：

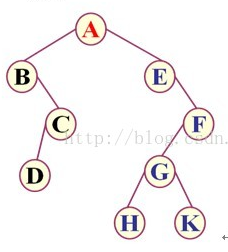


**先序遍历：GHK**

**中序遍历：HGK**

**后序遍历：HKG**

真正在草稿纸上写的时候，直接用笔画圈就更好写出来了。



先序遍历：ABCDEFGHK

中序遍历：BDCAEHGKF

后序遍历：DCBHKGFEA

## 证明：后序遍历的“左右根”与倒序的“根右左”结果一致性。

利用实例证明：上面那个图的后序遍历为：DCBHKGFEA

其“根右左”：A(E(F(GKH)))(B(CD)) 去括号：AEFGKHBCD

将其倒过来就和后序遍历一致了。

# 二叉树的前序遍历

前序遍历 外文名 **Preorder Traversal**

遍历利用**栈Stack结构**实现起来最佳。

一般有两种算法实现，一是递归实现，二是非递归实现。

## 递归算法的实现思路

优点：很好理解，代码实现简单，耗时短，效率快。

也有缺点：

思路：把每一个节点都看作是一个根节点，然后递归调用。

由于一个节点对应左右两个子树，所以需要在递归方法中调用两次自己。

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

// recursion method

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

**preOrder(root,result);//result作为容器**

return result;

}

private void preOrder(TreeNode root,List<Integer> result){

if(root == null){

return;

}

**result.add(root.val);**

preOrder(root.left,result);//递归遍历**左子树**

preOrder(root.right,result);//递归遍历**右子树**

}

}

理解注意：**把左子树和右子树都当成一颗新的大树，就很好理解了**。

## 非递归算法实现思路：

构建一个栈stack，通过入栈和出栈分别遍历每一个节点。

具体思路：Java代码中利用**ArrayDeque<TreeNode>**作为栈的数据结构，利用while循环进行遍历判断，条件是**栈非空或当前节点非null**；

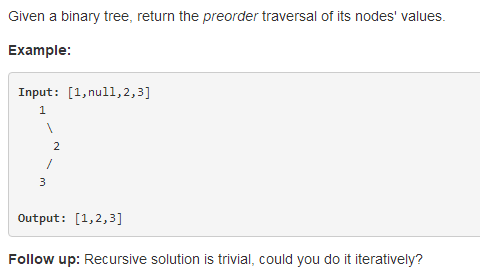
条件说明：栈为空时，说明刚刚从root开始或者返回到了root需要遍历右子树；当前节点为null，说明已经到了叶子节点，下一步需要执行出栈操作。只有当栈为空且当前节点为null时，才能说明已经遍历结束。

栈为空的三种情况：①刚开始，栈肯定为空；②当左子树遍历结束，A需要先出栈，再遍历右子树，此时栈为空；③最后右子树也遍历结束，此时栈为空，当前节点也为空，结束标志。

遍历实现体：当前节点非null，则需要将当前节点入栈，然后获取该节点的值，再将左子树节点赋值给当前节点，若当前节点为null，则需要从栈中获取当前节点的父节点（出栈意味着在栈中删除并返回），并将父节点的左子树节点赋给当前节点。



LeetCode上的第144题：



/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

Deque<TreeNode> stack = new ArrayDeque<TreeNode>();//数组队列做栈

while(!stack.isEmpty() || root != null){

//遍历结束条件，很重要：**只有栈为空且当前节点为null时**才说明遍历结束

**if(root != null){**

**stack.push(root);**

**result.add(root.val); //先序遍历与中序遍历的区别之处（将此句放在else的pop后面就是对应中序遍历）**

**root = root.left;//遍历左子树**

**}else{**

**root = stack.pop();//从栈中取出父亲节点**

**root = root.right;//右子树**

**}**

}

return result;

}

}

非递归算法重点掌握：1. **while循环的终止条件**；2. **root非null及null时执行内容**。

# 后序遍历（Postoreder Traversal）

## 递归算法

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

//recursion method

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

inOrder(root,result);

return result;

}

private void inOrder(TreeNode root,List<Integer> result){

if(root == null){

return ;

}

inOrder(root.left,result);

**result.add(root.val);**

inOrder(root.right,result);

}\*/

}

## 非递归算法

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

// not recursion method

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

Deque<TreeNode> stack = new ArrayDeque<TreeNode>();

while(!stack.isEmpty() || root != null){

if(root != null){

stack.push(root);

root = root.left;

}else{

root = stack.pop();

**result.add(root.val);**

root = root.right;

}

}

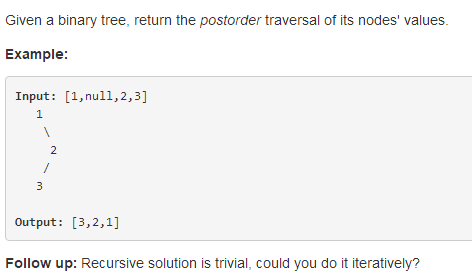
return result;

}

}

# 后序遍历(Postorder Traversal)





## 递归算法

class Solution {

//recursion method

public List<Integer> inorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

inOrder(root,result);

return result;

}

private void inOrder(TreeNode root,List<Integer> result){

if(root == null){

return ;

}

inOrder(root.left,result);

result.add(root.val);

inOrder(root.right,result);

}

}

## 非递归算法

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

List<Integer> res = new LinkedList<>();

Deque<TreeNode> stack = new ArrayDeque<>();

while (root != null || !stack.isEmpty()) {

if (root != null) **{//后序遍历是“前后中”，既然要逆过来实现，那就按照“中后前”实现即可**

stack.push(root);

res.add(0, root.val);//中

root = root.right;//后

} else {

root = stack.pop();

root = root.**left**;

}

}

return res;

}

}

第二种方法：（理解）

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

public List<Integer> postorderTraversal(TreeNode root) {

LinkedList<Integer> result = new LinkedList<Integer>();

Deque<TreeNode> stack = new ArrayDeque<TreeNode>();

if(root == null) return result;

stack.push(root);

while(!stack.isEmpty()){//**这个思想也是将“前后中”逆过来，就是“中后前”**

root = stack.pop();

if(root != null){

**result.addFirst(root.val);//关键：使用addFirst方法**

}

if(root.left != null){

stack.push(root.left);

}

if(root.right != null){

stack.push(root.right);

}

}

return result;

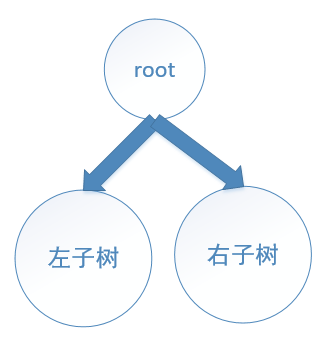
}

}

# 总结：三种遍历算法

## 递归实现很简单，只需要调节result.add(root.val);的顺序即可。

**很符合三种遍历的定义，容易记忆和理解。**



/\*

每次调用一次递归方法：把二叉树看成三部分，root节点、左子树和右子树。终止递归的条件很重要，就是当前root节点为null，就是直接返回。

下面三句话正好代表了访问中、左、右。关键是访问root节点的这句，只要调整该句执行的位置，即可得到先序遍历、中序遍历、后序遍历。

**先序遍历**：

**result.add(root.val);**

preOrder(root.left,result);//递归遍历**左子树**

preOrder(root.right,result);//递归遍历**右子树**

**中序遍历：**

**result.add(root.val);**

preOrder(root.left,result);//递归遍历**左子树**

preOrder(root.right,result);//递归遍历**右子树**

**后序遍历**：

**result.add(root.val);**

preOrder(root.left,result);//递归遍历**左子树**

preOrder(root.right,result);//递归遍历**右子树**

\*/

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

class Solution {

// recursion method

public List<Integer> preorderTraversal(TreeNode root) {

List<Integer> result = new ArrayList<Integer>();

**preOrder(root,result);//result作为容器**

return result;

}

**//递归方法**

private void **preOrder**(TreeNode root,List<Integer> result){

if(root == null){

return;

}

**result.add(root.val);**

preOrder(root.left,result);//递归遍历**左子树**

preOrder(root.right,result);//递归遍历**右子树**

}

}

## 非递归实现算法：利用栈实现

利用**while循环和栈结构**实现的非递归算法。其中先序遍历和中序遍历比较相似，只需要改变一句代码的位置即可。**区别**：先序遍历把获取值放在root != null里，而中序遍历放在eles即root == null里面。**result.add(root.val);**

对于后序遍历需要绕一下：采用的是LinkedList和栈数据结构。之所以使用LinkedList因为需要每次添加数据都是添加到List的第一个位置，利用addFrist方法。整体上把后序遍历的“前后中”变成按照**“中后前”**访问。

“中后前”与先序遍历的“中前后”只是right和left替换一下，因此后序遍历的非递归算法与前序遍历的非递归算法只有两点差别：

1. **后序遍历必须用LinkedList因为需要在最前面插入**，2. **right和left需要交换次序。**

总结，中序遍历和后序遍历与先序遍历都很相似，因此把先序遍历搞明白，然后弄清楚三者区别，即可全部掌握。