遍历二叉树的方法合集

二叉树的中序遍历

二叉树的前序遍历

二叉树的后序遍历

文章有一点长，代码居多。会给出比较详细的动画演示以及视频讲解请大家耐心阅读。

递归解法

前序遍历

Java

public static void preOrderRecur(TreeNode head) {

if (head == null) {

return;

}

System.out.print(head.value + " ");

preOrderRecur(head.left);

preOrderRecur(head.right);

}

中序遍历

Java

public static void preOrderRecur(TreeNode head) {

if (head == null) {

return;

}

preOrderRecur(head.left);

System.out.print(head.value + " ");

preOrderRecur(head.right);

}

后序遍历

Java

public static void postOrderRecur(TreeNode head) {

if (head == null) {

return;

}

postOrderRecur(head.left);

postOrderRecur(head.right);

System.out.print(head.value + " ");

}

代码解释

这里没有对递归遍历做过多的解释，仅仅给出了简单的动画演示。

要理解递归的思路并且熟练的使用它，就是要想清楚你想做什么，什么时候停止。

如前序遍历：

我想先打印头节点对吧？那我打印完了头节点，我现在想打印左边节点了，我只是告诉计算机我想打印左边结点，之后打印右边结点。

那么后序遍历呢？

这个时候你应该知道了，我就是想操作左边然后右边，最后打印中间的元素。

我们并不需要太过于在意具体的递归过程，而是要想清楚让计算机干什么。

计算机都可能溢出，用人脑去遍历就不现实了。

迭代解法

本质上是在模拟递归，因为在递归的过程中使用了系统栈，所以在迭代的解法中常用Stack来模拟系统栈。

前序遍历

首先我们应该创建一个Stack用来存放节点，首先我们想要打印根节点的数据，此时Stack里面的内容为空，所以我们优先将头结点加入Stack，然后打印。

之后我们应该先打印左子树，然后右子树。所以先加入Stack的就是右子树，然后左子树。

此时你能得到的流程如下:

Java

public static void preOrderIteration(TreeNode head) {

if (head == null) {

return;

}

Stack<TreeNode> stack = new Stack<>();

stack.push(head);

while (!stack.isEmpty()) {

TreeNode node = stack.pop();

System.out.print(node.value + " ");

if (node.right != null) {

stack.push(node.right);

}

if (node.left != null) {

stack.push(node.left);

}

}

}

中序遍历

同理创建一个Stack，然后按 左 中 右的顺序输出节点。

尽可能的将这个节点的左子树压入Stack，此时栈顶的元素是最左侧的元素，其目的是找到一个最小单位的子树(也就是最左侧的一个节点)，并且在寻找的过程中记录了来源，才能返回上层,同时在返回上层的时候已经处理完毕左子树了。。

当处理完最小单位的子树时，返回到上层处理了中间节点。（如果把整个左中右的遍历都理解成子树的话，就是处理完 左子树->中间(就是一个节点)->右子树）

如果有右节点，其也要进行中序遍历。

当整个左子树退栈的时候这个时候输出了该子树的根节点 2，之后输出中间节点 1。然后处理根节点为3右子树。

Java

public static void inOrderIteration(TreeNode head) {

if (head == null) {

return;

}

TreeNode cur = head;

Stack<TreeNode> stack = new Stack<>();

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

while (cur != null) {

stack.push(cur);

cur = cur.left;

}

TreeNode node = stack.pop();

System.out.print(node.value + " ");

if (node.right != null) {

cur = node.right;

}

}

}

后序遍历

代码片段 1

前序遍历的过程 是 中左右。

将其转化成 中右左。也就是压栈的过程中优先压入左子树，在压入右子树。

然后将这个结果返回来，这里是利用栈的先进后出倒序打印。

Java

public static void postOrderIteration(TreeNode head) {

if (head == null) {

return;

}

Stack<TreeNode> stack1 = new Stack<>();

Stack<TreeNode> stack2 = new Stack<>();

stack1.push(head);

while (!stack1.isEmpty()) {

TreeNode node = stack1.pop();

stack2.push(node);

if (node.left != null) {

stack1.push(node.left);

}

if (node.right != null) {

stack1.push(node.right);

}

}

while (!stack2.isEmpty()) {

System.out.print(stack2.pop().value + " ");

}

}

代码片段 2

用一个指针cur标记当前退出的节点是什么。

后序遍历的过程中在遍历完左子树跟右子树cur都会回到根结点。所以当前不管是从左子树还是右子树回到根结点都不应该再操作了，应该退回上层。

如果是从右边再返回根结点，应该回到上层。

Java

public static void postOrderIteration2(TreeNode head) { 感谢[@ben-ben-niu](/u/ben-ben-niu/)指出错误，代码确实存在问题，已经在原文中修复

if (head == null) {

return;

}

TreeNode cur = head;

Stack<TreeNode> stack = new Stack<>();

stack.push(head);

while (!stack.isEmpty()) {

TreeNode peek = stack.peek();

if (peek.left != null && peek.left != cur && peek.right != cur) {

stack.push(peek.left);

} else if (peek.right != null && peek.right != cur) {

stack.push(peek.right);

} else {

System.out.print(stack.pop().val + " ");

cur = peek;

}

}

}

作者：gre-z

链接：https://leetcode-cn.com/problems/binary-tree-preorder-traversal/solution/leetcodesuan-fa-xiu-lian-dong-hua-yan-shi-xbian-2/

来源：力扣（LeetCode）

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。