```
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
        return head;
    }

    public Node generateNodeByString(String val) {
        if (val.equals("#")) {
            return null;
        }
        return new Node(Integer.valueOf(val));
    }
```

遍历二叉树的神级方法

【题目】

给定一棵二叉树的头节点 head，完成二叉树的先序、中序和后序遍历。如果二叉树的节点数为 N，要求时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。

【难度】

将 ★★★★

【解答】

本题真正的难点在于对复杂度的要求，尤其是额外空间复杂度为 $O(1)$的限制。之前的题目已经剖析过如何用递归和非递归的方法实现遍历二叉树，很不幸，之前所有的方法虽然常用，但都无法做到额外空间复杂度为 $O(1)$。这是因为遍历二叉树的递归方法实际使用了函数栈，非递归的方法使用了申请的栈，两者的额外空间都与树的高度相关，所以空间复杂度为 $O(h)$，h 为二叉树的高度。如果完全不用栈结构能完成三种遍历吗？可以。答案是使用二叉树节点中大量指向 null 的指针，本题实际上就是大名鼎鼎的 Morris 遍历，由 Joseph Morris 于 1979 年发明。

首先来看普通的递归和非递归解法，其实都使用了栈结构，在处理完二叉树某个节点后可以回到上层去。为什么从下层回到上层会如此之难？因为二叉树的结构如此，每个节点都有指向孩子节点的指针，所以从上层到下层容易，但是没有指向父节点的指针，所以

从下层到上层需要用栈结构辅助完成。

Morris 遍历的实质就是避免用栈结构，而是让下层到上层有指针，具体是通过让底层节点指向 null 的空闲指针指回上层的某个节点，从而完成下层到上层的移动。我们知道，二叉树上的很多节点都有大量的空闲指针，比如，某些节点没有右孩子，那么这个节点的 right 指针就指向 null，我们称为空闲状态，Morris 遍历正是利用了这些空闲指针。

在介绍 Morris 先序和后序遍历之前，我们先举例展示 Morris 中序遍历的过程。

假设一棵二叉树如图 3-9 所示，Morris 中序遍历的具体过程如下：

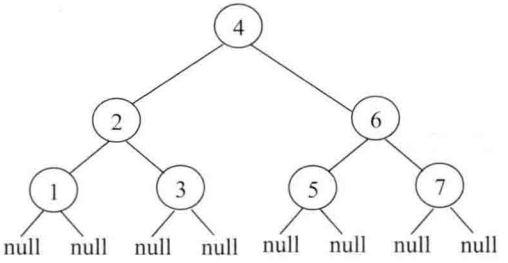

图 3-9

1．假设当前子树的头节点为 h，让 h 的左子树中最右节点的 right 指针指向 h，然后 h 的左子树继续步骤 1 的处理过程，直到遇到某一个节点没有左子树时记为 node，进入步骤 2。

举例：图 3-9 的二叉树在开始时 h 为节点 4，通过步骤 1 让节点 3 的 right 指针指向节点 4，接下来以节点 2 为头的子树继续进入步骤 1，然后让节点 1 的 right 指针指向 2，接下来以节点 1 为头的子树没有左子树了，步骤 1 停止，节点 1 进入步骤 2，此时结构调整为图 3-10。

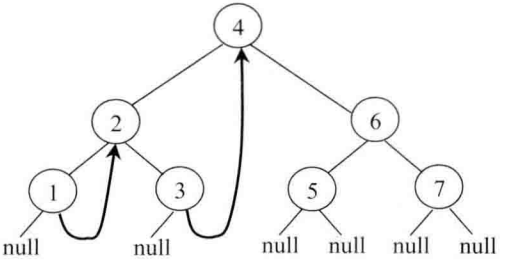

图 3-10

2．从 node 开始通过每个节点的 right 指针进行移动，并依次打印，假设移动到的节点为 cur。对每一个 cur 节点都判断 cur 节点的左子树中最右节点是否指向 cur。

① 如果是。让 cur 节点的左子树中最右节点的 right 指针指向空，也就是把步骤 1 的调整后再逐渐调整回来，然后打印 cur，继续通过 cur 的 right 指针移动到下一个节点，重复步骤 2。

② 如果不是，以 cur 为头的子树重回步骤 1 执行。

用例子说明这个过程如下：

节点 1 先打印，通过节点 1 的 right 指针移动到节点 2。

发现节点 2 符合步骤 2 的条件①，所以令节点 1 的 right 指针指向 null，然后打印节点 2，再通过节点 2 的 right 指针移动到节点 3。

发现节点 3 符合步骤 2 的条件②，节点 3 为头的子树进入步骤 1 处理，但因为这个子树只有节点 3，所以步骤 1 迅速处理完，又回到节点 3，打印节点 3，然后通过节点 3 的 right 指针移动到节点 4。

发现节点 4 符合步骤 2 的条件①，所以令节点 3 的 right 指针指向 null，然后打印节点 4，再通过节点 4 的 right 指针移动到节点 6。到目前为止，二叉树的结构又回到了图 3-9 的样子。

发现节点 6 符合步骤 2 的条件②，所以，以节点 6 为头的子树进入步骤 1 进行处理，处理之后，二叉树变成图 3-11 所示的样子。

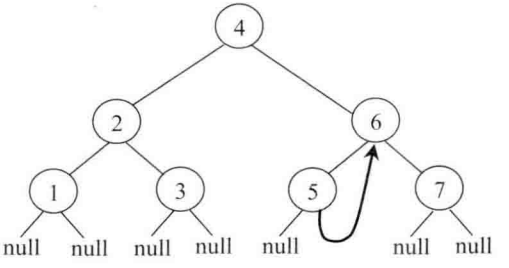

图 3-11

重新来到步骤 2 的第一个节点是以节点 6 为头的子树的最左节点，即节点 5，发现节点 5 符合步骤 2 的条件②，节点 5 为头的子树进入步骤 1 处理，但因为这棵子树只有节点 5，所以步骤 1 迅速处理完，打印节点 5，然后通过节点 5 的 right 指针移动到节点 6。

发现节点 6 符合步骤 2 的条件①，所以令节点 5 的 right 指针指向 null，然后打印节点 6，再通过节点 6 的 right 指针移动到节点 7。到目前为止，二叉树的结构又回到了图 3-9 的样子。

节点 7 符合步骤 2 的条件②，以节点 7 的子树经历步骤 1、步骤 2 和步骤 3 并打印。

然后通过节点 7 的 right 指针移动到 null，整个过程结束。

3．步骤 2 最终移动到 null，整个过程结束。

通过上述步骤描述我们知道，先序遍历在打印某个节点时，一定是在步骤 2 开始移动的过程中，而步骤 2 最初开始时的位置一定是子树的最左节点，在通过 right 指针移动的过程中，我们发现要么是某个节点移动到其右子树上，比如，节点 2 向节点 3 的移动、节点 4 向节点 6 的移动，以及节点 6 向节点 7 的移动，发生这种情况的时候，左子树和根节点已经打印结束，然后开始右子树的处理过程；要么是某个节点移动到某个上层的节点，比如节点 1 向节点 2 的移动、节点 3 向节点 4 的移动，以及节点 5 向节点 6 的移动，发生这种情况的时候，必然是这个上层节点的左子树整体打印完毕，然后开始处理根节点（也就是这个上层节点）和右子树的过程。Morris 中序遍历的具体实现请参看如下代码中的 morrisIn 方法。

```
public class Node {
        public int value;
        Node left;
        Node right;

        public Node(int data) {
                this.value = data;
        }
}

public void morrisIn(Node head) {
        if (head == null) {
                return;
        }
        Node cur1 = head;
        Node cur2 = null;
        while (cur1 != null) {
                cur2 = cur1.left;
                if (cur2 != null) {
                        while (cur2.right != null && cur2.right != cur1) {
                                cur2 = cur2.right;
                        }
                        if (cur2.right == null) {
                                cur2.right = cur1;
                                cur1 = cur1.left;
                                continue;
                        } else {
                                cur2.right = null;
                        }
                }
                System.out.print(cur1.value + " ");
                cur1 = cur1.right;
        }
```

```
        System.out.println();
    }
```

从代码可以轻易看出，Morris 中序遍历的额外空间复杂度为 $O(1)$，只使用了有限几个变量。时间复杂度方面可以这么分析，二叉树的每条边都最多经历一次步骤 1 的调整过程，再最多经历一次步骤 3 的调回来的过程，所有边的节点个数为 N，所以调整和调回的过程，其时间复杂度为 $O(N)$，打印所有节点的时间复杂度为 $O(N)$。所以，总的时间复杂度为 $O(N)$。

Morris 先序遍历的实现就是 Morris 中序遍历实现的简单改写。先序遍历的打印时机放在了步骤 2 所描述的移动过程中，而先序遍历只要把打印时机放在步骤 1 发生的时候即可。步骤 1 发生的时候，正在处理以 h 为头的子树，并且是以 h 为头的子树首次进入调整过程，此时直接打印 h，就可以做到先根打印。

Morris 先序遍历的具体实现请参看如下代码中的 morrisPre 方法。

```
public void morrisPre(Node head) {
        if (head == null) {
                return;
        }
        Node cur1 = head;
        Node cur2 = null;
        while (cur1 != null) {
                cur2 = cur1.left;
                if (cur2 != null) {
                        while (cur2.right != null && cur2.right != cur1) {
                                cur2 = cur2.right;
                        }
                        if (cur2.right == null) {
                                cur2.right = cur1;
                                System.out.print(cur1.value + " ");
                                cur1 = cur1.left;
                                continue;
                        } else {
                                cur2.right = null;
                        }
                } else {
                        System.out.print(cur1.value + " ");
                }
                cur1 = cur1.right;
        }
        System.out.println();
}
```

Morris 后序遍历的实现也是 Morris 中序遍历实现的改写，但包含更复杂的调整过程。总的来说，逻辑很简单，就是依次逆序打印所有节点的左子树的右边界，打印的时机放在步骤 2 的条件①被触发的时候，也就是调回去的过程发生的时候。

还是以图 3-9 的二叉树来举例说明 Morris 后序遍历的打印过程，头节点（即节点 4）在经过步骤 1 的调整过程之后，形成如图 3-10 所示的形式。

节点 1 进入步骤 2，不打印节点 1，而是直接通过节点 1 的 right 指针移动到节点 2。

发现节点 2 符合步骤 2 的条件①，此时先把节点 1 的 right 指针指向 null（调回来），节点 2 左子树的右边界只有节点 1，所以打印节点 1，通过节点 2 的 right 指针移动到节点 3。

发现节点 3 符合步骤 2 的条件②，节点 3 为头的子树进入步骤 1 处理，回到节点 3 后不打印节点 3，而是直接通过节点 3 的 right 指针移动到节点 4。

发现节点 4 符合步骤 2 的条件①，此时二叉树如图 3-12 所示。

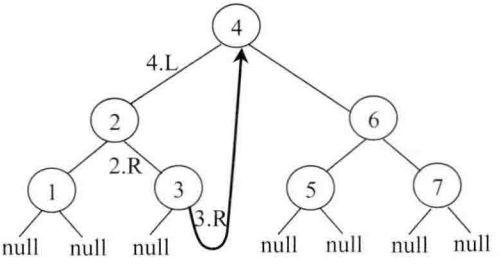

图 3-12

将节点 4 左子树的右边界（节点 2 和节点 3）逆序打印，但这里的逆序打印不能使用额外的数据结构，因为我们的要求是额外空间复杂度为 $O(1)$，所以采用调整右边界上节点的 right 指针的方式。为了更好地说明整个过程，下面举一个右边界比较长的例子，如图 3-13 所示。

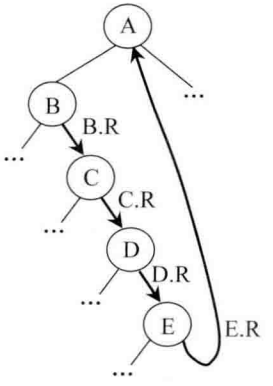

图 3-13

假设现在要逆序打印节点 A 左子树的右边界，首先将 E.R 指向 null，然后将右边界逆序调整成图 3-14 所示的样子。

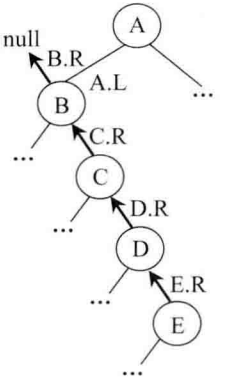

图 3-14

这样我们就可以从节点 E 开始，依次通过每个节点的 right 指针逆序打印整个左边界。在打印完 B 后，把右边界再逆序一次，调回来即可。

回到原来的二叉树（即图 3-12），先把节点 3 的 right 指针指向 null（调回来），二叉树变为图 3-9 所示的样子，然后将节点 4 左子树的右边界逆序打印(3，2)，通过节点 4 的 right 指针移动到节点 6。

发现节点 6 符合步骤 2 的条件②，所以，以节点 6 为头的子树进入步骤 1 进行处理，处理之后的二叉树变成图 3-11 所示的样子。

节点 5 重新来到步骤 2，发现节点 5 符合步骤 2 的条件②，进入步骤 1 并迅速处理完，不打印节点 5，而是直接通过节点 5 的 right 指针移动到节点 6。

发现节点 6 符合步骤 2 的条件①，先将节点 5 的 right 指针指向 null，节点 6 左子树的右边界只有节点 5，打印节点 5，然后通过节点 6 的 right 指针移动到节点 7。

发现节点 7 符合步骤 2 的条件②，进入步骤 1 并迅速处理完，不打印节点 7，通过节点 7 的 right 指针移动到 null，过程结束。

至此，已经依次打印了 1、3、2、5，但还没有打印 7、6、4，这是因为整棵二叉树并不属于任何节点的左子树，所以，整棵树的右边界就没在上述过程中逆序打印。最后，单独逆序打印一下整棵树的右边界即可。

Morris 后序遍历的具体实现请参看如下代码中的 morrisPos 方法。

```
public void morrisPos(Node head) {
        if (head == null) {
```

```
                return;
        }
        Node cur1 = head;
        Node cur2 = null;
        while (cur1 != null) {
                cur2 = cur1.left;
                if (cur2 != null) {
                        while (cur2.right != null && cur2.right != cur1) {
                                cur2 = cur2.right;
                        }
                        if (cur2.right == null) {
                                cur2.right = cur1;
                                cur1 = cur1.left;
                                continue;
                        } else {
                                cur2.right = null;
                                printEdge(cur1.left);
                        }
                }
                cur1 = cur1.right;
        }
        printEdge(head);
        System.out.println();
}

public void printEdge(Node head) {
        Node tail = reverseEdge(head);
        Node cur = tail;
        while (cur != null) {
                System.out.print(cur.value + " ");
                cur = cur.right;
        }
        reverseEdge(tail);
}

public Node reverseEdge(Node from) {
        Node pre = null;
        Node next = null;
        while (from != null) {
                next = from.right;
                from.right = pre;
                pre = from;
                from = next;
        }
        return pre;
}
```