第 3 章

二叉树问题

分别用递归和非递归方式实现二叉树先序、中序和后序遍历

【题目】

用递归和非递归方式，分别按照二叉树先序、中序和后序打印所有的节点。我们约定：先序遍历顺序为根、左、右；中序遍历顺序为左、根、右；后序遍历顺序为左、右、根。

【难度】

校 ★★★☆

【解答】

用递归方式实现三种遍历是教材上的基础内容，本书不再详述，直接给出代码实现。先序遍历的递归实现请参看如下代码中的 preOrderRecur 方法。

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

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

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

```
                return;
        }
        System.out.print(head.value + " ");
        preOrderRecur(head.left);
        preOrderRecur(head.right);
}
```

中序遍历的递归实现请参看如下代码中的 inOrderRecur 方法。

```
public void inOrderRecur(Node head) {
        if (head == null) {
                return;
        }
        inOrderRecur(head.left);
        System.out.print(head.value + " ");
        inOrderRecur(head.right);
}
```

后序遍历的递归实现请参看如下代码中的 posOrderRecur 方法。

```
public void posOrderRecur(Node head) {
        if (head == null) {
                return;
        }
        posOrderRecur(head.left);
        posOrderRecur(head.right);
        System.out.print(head.value + " ");
}
```

用递归方法解决的问题都能用非递归的方法实现。这是因为递归方法无非就是利用函数栈来保存信息，如果用自己申请的数据结构来代替函数栈，也可以实现相同的功能。

用非递归的方式实现二叉树的先序遍历，具体过程如下：

1．申请一个新的栈，记为 stack。然后将头节点 head 压入 stack 中。

2．从 stack 中弹出栈顶节点，记为 cur，然后打印 cur 节点的值，再将节点 cur 的右孩子（不为空的话）先压入 stack 中，最后将 cur 的左孩子（不为空的话）压入 stack 中。

3．不断重复步骤 2，直到 stack 为空，全部过程结束。

下面举例说明整个过程，一棵二叉树如图 3-1 所示。

节点 1 先入栈，然后弹出并打印。接下来先把节点 3 压入 stack，再把节点 2 压入，stack 从栈顶到栈底依次为 2，3。

节点 2 弹出并打印，把节点 5 压入 stack，再把节点 4 压入，stack 从栈顶到栈底为 4，5，3。

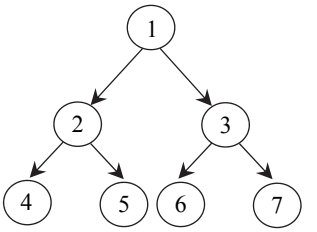

图 3-1

节点 4 弹出并打印，节点 4 没有孩子压入 stack，stack 从栈顶到栈底依次为 5，3。

节点 5 弹出并打印，节点 5 没有孩子压入 stack，stack 从栈顶到栈底依次为 3。

节点 3 弹出并打印，把节点 7 压入 stack，再把节点 6 压入，stack 从栈顶到栈底为 6，7。

节点 6 弹出并打印，节点 6 没有孩子压入 stack，stack 目前从栈顶到栈底为 7。

节点 7 弹出并打印，节点 7 没有孩子压入 stack，stack 已经为空，过程停止。

整个过程请参看如下代码中的 preOrderUnRecur 方法。

```
public void preOrderUnRecur(Node head) {
        System.out.print("pre-order: ");
        if (head != null) {
                Stack<Node> stack = new Stack<Node>();
                stack.add(head);
                while (!stack.isEmpty()) {
                        head = stack.pop();
                        System.out.print(head.value + " ");
                        if (head.right != null) {
                                stack.push(head.right);
                        }
                        if (head.left != null) {
                                stack.push(head.left);
                        }
                }
        }
        System.out.println();
}
```

用非递归的方式实现二叉树的中序遍历，具体过程如下：

1．申请一个新的栈，记为 stack。初始时，令变量 cur=head。

2．先把 cur 节点压入栈中，对以 cur 节点为头的整棵子树来说，依次把左边界压入栈中，即不停地令 cur=cur.left，然后重复步骤 2。

3．不断重复步骤 2，直到发现 cur 为空，此时从 stack 中弹出一个节点，记为 node。打印 node 的值，并且让 cur=node.right，然后继续重复步骤 2。

4．当 stack 为空且 cur 为空时，整个过程停止。

还是用图 3-1 的例子来说明整个过程。

初始时 cur 为节点 1，将节点 1 压入 stack，令 cur=cur.left，即 cur 变为节点 2。(步骤 1+步骤 2)

cur 为节点 2，将节点 2 压入 stack，令 cur=cur.left，即 cur 变为节点 4。(步骤 2)

cur 为节点 4，将节点 4 压入 stack，令 cur=cur.left，即 cur 变为 null，此时 stack 从栈顶到栈底为 4，2，1。(步骤 2)

cur 为 null，从 stack 弹出节点 4(node)并打印，令 cur=node.right，即 cur 为 null，此时 stack 从栈顶到栈底为 2，1。(步骤 3)

cur 为 null，从 stack 弹出节点 2(node)并打印，令 cur=node.right，即 cur 变为节点 5，此时 stack 从栈顶到栈底为 1。(步骤 3)

cur 为节点 5，将节点 5 压入 stack，令 cur=cur.left，即 cur 变为 null，此时 stack 从栈顶到栈底为 5，1。(步骤 2)

cur 为 null，从 stack 弹出节点 5(node)并打印，令 cur=node.right，即 cur 仍为 null，此时 stack 从栈顶到栈底为 1。(步骤 3)

cur 为 null，从 stack 弹出节点 1(node)并打印，令 cur=node.right，即 cur 变为节点 3，此时 stack 为空。(步骤 3)

cur 为节点 3，将节点 3 压入 stack，令 cur=cur.left 即 cur 变为节点 6；此时 stack 从栈顶到栈底为 3。(步骤 2)

cur 为节点 6，将节点 6 压入 stack，令 cur=cur.left 即 cur 变为 null，此时 stack 从栈顶到栈底为 6，3。(步骤 2)

cur 为 null，从 stack 弹出节点 6(node)并打印，令 cur=node.right，即 cur 仍为 null，此时 stack 从栈顶到栈底为 3。(步骤 3)

cur 为 null，从 stack 弹出节点 3(node)并打印，令 cur=node.right，即 cur 变为节点 7，此时 stack 为空。(步骤 3)

cur 为节点 7，将节点 7 压入 stack，令 cur=cur.left，即 cur 变为 null，此时 stack 从栈顶到栈底为 7。(步骤 2)

cur 为 null，从 stack 弹出节点 7(node)并打印，令 cur=node.right，即 cur 仍为 null，此时 stack 为空。(步骤 3)

cur 为 null，stack 也为空，整个过程停止。(步骤 4)

通过与例子结合的方式我们发现，步骤 1 到步骤 4 就是依次先打印左子树，然后是每

棵子树的头节点，最后打印右子树。

全部过程请参看如下代码中的 inOrderUnRecur 方法。

```
public void inOrderUnRecur(Node head) {
        System.out.print("in-order: ");
        if (head != null) {
                Stack<Node> stack = new Stack<Node>();
                while (!stack.isEmpty() || head != null) {
                        if (head != null) {
                                stack.push(head);
                                head = head.left;
                        } else {
                                head = stack.pop();
                                System.out.print(head.value + " ");
                                head = head.right;
                        }
                }
        }
        System.out.println();
}
```

用非递归的方式实现二叉树的后序遍历有点麻烦，本书实现两种方法供读者参考。

先介绍用两个栈实现后序遍历的过程，具体过程如下：

1．申请一个栈，记为 s1，然后将头节点 head 压入 s1 中。

2．从 s1 中弹出的节点记为 cur，然后依次将 cur 的左孩子和右孩子压入 s1 中。

3．在整个过程中，每一个从 s1 中弹出的节点都放进 s2 中。

4．不断重复步骤 2 和步骤 3，直到 s1 为空，过程停止。

5．从 s2 中依次弹出节点并打印，打印的顺序就是后序遍历的顺序。

还是用图 3-1 的例子来说明整个过程。

节点 1 放入 s1 中。

从 s1 中弹出节点 1，节点 1 放入 s2，然后将节点 2 和节点 3 依次放入 s1，此时 s1 从栈顶到栈底为 3，2；s2 从栈顶到栈底为 1。

从 s1 中弹出节点 3，节点 3 放入 s2，然后将节点 6 和节点 7 依次放入 s1，此时 s1 从栈顶到栈底为 7，6，2；s2 从栈顶到栈底为 3，1。

从 s1 中弹出节点 7，节点 7 放入 s2，节点 7 无孩子节点，此时 s1 从栈顶到栈底为 6，2；s2 从栈顶到栈底为 7，3，1。

从 s1 中弹出节点 6，节点 6 放入 s2，节点 6 无孩子节点，此时 s1 从栈顶到栈底为 2；s2 从栈顶到栈底为 6，7，3，1。

从 s1 中弹出节点 2，节点 2 放入 s2，然后将节点 4 和节点 5 依次放入 s1，此时 s1 从

栈顶到栈底为 5，4；s2 从栈顶到栈底为 2，6，7，3，1。

从 s1 中弹出节点 5，节点 5 放入 s2，节点 5 无孩子节点，此时 s1 从栈顶到栈底为 4；s2 从栈顶到栈底为 5，2，6，7，3，1。

从 s1 中弹出节点 4，节点 4 放入 s2，节点 4 无孩子节点，此时 s1 为空；s2 从栈顶到栈底为 4，5，2，6，7，3，1。

过程结束，此时只要依次弹出 s2 中的节点并打印即可，顺序为 4，5，2，6，7，3，1。

通过如上过程我们知道，每棵子树的头节点都最先从 s1 中弹出，然后把该节点的孩子节点按照先左再右的顺序压入 s1，那么从 s1 弹出的顺序就是先右再左，所以从 s1 中弹出的顺序就是中、右、左。然后，s2 重新收集的过程就是把 s1 的弹出顺序逆序，所以 s2 从栈顶到栈底的顺序就变成了左、右、中。

使用两个栈实现后序遍历的全部过程请参看如下代码中的 posOrderUnRecur1 方法。

```
public void posOrderUnRecur1(Node head) {
        System.out.print("pos-order: ");
        if (head != null) {
                Stack<Node> s1 = new Stack<Node>();
                Stack<Node> s2 = new Stack<Node>();
                s1.push(head);
                while (!s1.isEmpty()) {
                        head = s1.pop();
                        s2.push(head);
                        if (head.left != null) {
                                s1.push(head.left);
                        }
                        if (head.right != null) {
                                s1.push(head.right);
                        }
                }
                while (!s2.isEmpty()) {
                        System.out.print(s2.pop().value + " ");
                }
        }
        System.out.println();
}
```

最后介绍只用一个栈实现后序遍历的过程，具体过程如下：

1．申请一个栈，记为 stack，将头节点压入 stack，同时设置两个变量 h 和 c。在整个流程中，h 代表最近一次弹出并打印的节点，c 代表 stack 的栈顶节点，初始时 h 为头节点，c 为 null。

2. 每次令 c 等于当前 stack 的栈顶节点，但是不从 stack 中弹出，此时分以下三种情况。

① 如果 c 的左孩子不为 null，并且 h 不等于 c 的左孩子，也不等于 c 的右孩子，则把

c 的左孩子压入 stack 中。具体解释一下这么做的原因，首先 h 的意义是最近一次弹出并打印的节点，所以如果 h 等于 c 的左孩子或者右孩子，说明 c 的左子树与右子树已经打印完毕，此时不应该再将 c 的左孩子放入 stack 中。否则，说明左子树还没处理过，那么此时将 c 的左孩子压入 stack 中。

② 如果条件①不成立，并且 c 的右孩子不为 null，h 不等于 c 的右孩子，则把 c 的右孩子压入 stack 中。含义是如果 h 等于 c 的右孩子，说明 c 的右子树已经打印完毕，此时不应该再将 c 的右孩子放入 stack 中。否则，说明右子树还没处理过，此时将 c 的右孩子压入 stack 中。

③ 如果条件①和条件②都不成立，说明 c 的左子树和右子树都已经打印完毕，那么从 stack 中弹出 c 并打印，然后令 h=c。

3．一直重复步骤 2，直到 stack 为空，过程停止。

依然用图 3-1 的例子来说明整个过程。

节点 1 压入 stack，初始时 h 为节点 1，c 为 null，stack 从栈顶到栈底为 1。

令 c 等于 stack 的栈顶节点——节点 1，此时步骤 2 的条件①命中，将节点 2 压入 stack，h 为节点 1，stack 从栈顶到栈底为 2，1。

令 c 等于 stack 的栈顶节点——节点 2，此时步骤 2 的条件①命中，将节点 4 压入 stack，h 为节点 1，stack 从栈顶到栈底为 4，2，1。

令 c 等于 stack 的栈顶节点——节点 4，此时步骤 2 的条件③命中，将节点 4 从 stack 中弹出并打印，h 变为节点 4，stack 从栈顶到栈底为 2，1。

令 c 等于 stack 的栈顶节点——节点 2，此时步骤 2 的条件②命中，将节点 5 压入 stack，h 为节点 4，stack 从栈顶到栈底为 5，2，1。

令 c 等于 stack 的栈顶节点——节点 5，此时步骤 2 的条件③命中，将节点 5 从 stack 中弹出并打印，h 变为节点 5，stack 从栈顶到栈底为 2，1。

令 c 等于 stack 的栈顶节点——节点 2，此时步骤 2 的条件③命中，将节点 2 从 stack 中弹出并打印，h 变为节点 2，stack 从栈顶到栈底为 1。

令 c 等于 stack 的栈顶节点——节点 1，此时步骤 2 的条件②命中，将节点 3 压入 stack，h 为节点 2，stack 从栈顶到栈底为 3，1。

令 c 等于 stack 的栈顶节点——节点 3，此时步骤 2 的条件①命中，将节点 6 压入 stack，h 为节点 2，stack 从栈顶到栈底为 6，3，1。

令 c 等于 stack 的栈顶节点——节点 6，此时步骤 2 的条件③命中，将节点 6 从 stack 中弹出并打印，h 变为节点 6，stack 从栈顶到栈底为 3，1。

令 c 等于 stack 的栈顶节点——节点 3，此时步骤 2 的条件②命中，将节点 7 压入 stack，h 为节点 6，stack 从栈顶到栈底为 7，3，1。

令 c 等于 stack 的栈顶节点——节点 7，此时步骤 2 的条件③命中，将节点 7 从 stack 中弹出并打印，h 变为节点 7，stack 从栈顶到栈底为 3，1。

令 c 等于 stack 的栈顶节点——节点 3，此时步骤 2 的条件③命中，将节点 3 从 stack 中弹出并打印，h 变为节点 3，stack 从栈顶到栈底为 1。

令 c 等于 stack 的栈顶节点——节点 1，此时步骤 2 的条件③命中，将节点 1 从 stack 中弹出并打印，h 变为节点 1，stack 为空。

过程结束。

只用一个栈实现后序遍历的全部过程请参看如下代码中的 posOrderUnRecur2 方法。

```
public void posOrderUnRecur2(Node h) {
        System.out.print("pos-order: ");
        if (h != null) {
                Stack<Node> stack = new Stack<Node>();
                stack.push(h);
                Node c = null;
                while (!stack.isEmpty()) {
                        c = stack.peek();
                        if (c.left != null && h != c.left && h != c.right) {
                                stack.push(c.left);
                        } else if (c.right != null && h != c.right) {
                                stack.push(c.right);
                        } else {
                                System.out.print(stack.pop().value + " ");
                                h = c;
                        }
                }
        }
        System.out.println();
}
```

打印二叉树的边界节点

【题目】

给定一棵二叉树的头节点 head，按照如下两种标准分别实现二叉树边界节点的逆时针打印。

标准一：

1．头节点为边界节点。

2．叶节点为边界节点。

3．如果节点在其所在的层中是最左或最右的，那么也是边界节点。

标准二：

1．头节点为边界节点。

2．叶节点为边界节点。

3．树左边界延伸下去的路径为边界节点。

4．树右边界延伸下去的路径为边界节点。

例如，如图 3-2 所示的树。

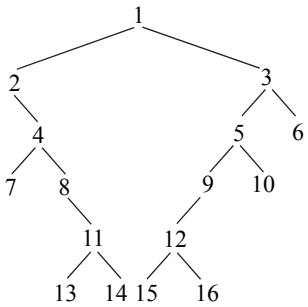

图 3-2

按标准一的打印结果为：1，2，4，7，11，13，14，15，16，12，10，6，3

按标准二的打印结果为：1，2，4，7，13，14，15，16，10，6，3

【要求】

1．如果节点数为 N，两种标准实现的时间复杂度要求都为 $O(N)$，额外空间复杂度要求都为 $O(h)$，h 为二叉树的高度。

2．两种标准都要求逆时针顺序且不重复打印所有的边界节点。

【难度】

尉 ★★☆☆

【解答】

按照标准一的要求实现打印的具体过程如下：

1．得到二叉树每一层上最左和最右的节点。以题目的例子来说，这个记录如下：

	最左节点	最右节点
第一层	1	1
第二层	2	3
第三层	4	6
第四层	7	10
第五层	11	12
第六层	13	16

2．从上到下打印所有层中的最左节点。对题目的例子来说，即打印：1，2，4，7，11，13。

3．先序遍历二叉树，打印那些不属于某一层最左或最右的节点，但同时又是叶节点的节点。对题目的例子来说，即打印：14，15。

4．从下到上打印所有层中的最右节点，但节点不能既是最左节点，又是最右节点。对题目的例子来说，即打印：16，12，10，6，3。

按标准一打印的全部过程请参看如下代码中的 printEdge1 方法。

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

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

public void printEdge1(Node head) {
        if (head == null) {
                return;
        }
        int height = getHeight(head, 0);
        Node[][] edgeMap = new Node[height][2];
        setEdgeMap(head, 0, edgeMap);
        // 打印左边界
        for (int i = 0; i != edgeMap.length; i++) {
                System.out.print(edgeMap[i][0].value + " ");
        }
        // 打印既不是左边界，也不是右边界的叶子节点
        printLeafNotInMap(head, 0, edgeMap);
        // 打印右边界，但不是左边界的节点
        for (int i = edgeMap.length - 1; i != -1; i--) {
                if (edgeMap[i][0] != edgeMap[i][1]) {
                        System.out.print(edgeMap[i][1].value + " ");
                }
        }
```

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

    public int getHeight(Node h, int l) {
        if (h == null) {
            return l;
        }
        return Math.max(getHeight(h.left, l + 1), getHeight(h.right, l + 1));
    }

    public void setEdgeMap(Node h, int l, Node[][] edgeMap) {
        if (h == null) {
            return;
        }
        edgeMap[l][0] = edgeMap[l][0] == null ? h : edgeMap[l][0];
        edgeMap[l][1] = h;
        setEdgeMap(h.left, l + 1, edgeMap);
        setEdgeMap(h.right, l + 1, edgeMap);
    }

    public void printLeafNotInMap(Node h, int l, Node[][] m) {
        if (h == null) {
            return;
        }
        if (h.left == null && h.right == null && h != m[l][0] && h != m[l][1]) {
            System.out.print(h.value + " ");
        }
        printLeafNotInMap(h.left, l + 1, m);
        printLeafNotInMap(h.right, l + 1, m);
    }
```

按照标准二的要求实现打印的具体过程如下：

1. 从头节点开始往下寻找，只要找到第一个既有左孩子，又有右孩子的节点，记为 h，则进入步骤 2。在这个过程中，找过的节点都打印。对题目的例子来说，即打印：1，因为头节点直接符合要求，所以打印后没有后续的寻找过程，直接进入步骤 2。但如果二叉树如图 3-3 所示，此时则打印：1，2，3。节点 3 是从头节点开始往下第一个符合要求的。如果二叉树从上到下一直找到叶节点也不存在符合要求的节点，说明二叉树是棒状结构，那么打印找过的节点后直接返回即可。

2. h 的左子树先进入步骤 3 的打印过程；h 的右子树再进入步骤 4 的打印过程；最后返回。

3. 打印左边界的延伸路径以及 h 左子树上所有的叶节点，具体请参看 printLeftEdge 方法。

4. 打印右边界的延伸路径以及 h 右子树上所有的叶节点，具体请参看 printRightEdge 方法。

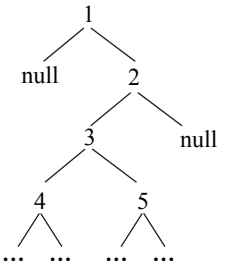

图 3-3

按标准二打印的全部过程请参看如下代码中的 printEdge2 方法。

```
public void printEdge2(Node head) {
        if (head == null) {
                return;
        }
        System.out.print(head.value + " ");
        if (head.left != null && head.right != null) {
                printLeftEdge(head.left, true);
                printRightEdge(head.right, true);
        } else {
                printEdge2(head.left != null ? head.left : head.right);
        }
        System.out.println();
}

public void printLeftEdge(Node h, boolean print) {
        if (h == null) {
                return;
        }
        if (print || (h.left == null && h.right == null)) {
                System.out.print(h.value + " ");
        }
        printLeftEdge(h.left, print);
        printLeftEdge(h.right, print && h.left == null ? true : false);
}

public void printRightEdge(Node h, boolean print) {
        if (h == null) {
                return;
        }
        printRightEdge(h.left, print && h.right == null ? true : false);
        printRightEdge(h.right, print);
        if (print || (h.left == null && h.right == null)) {
                System.out.print(h.value + " ");
        }
}
```

如何较为直观地打印二叉树

【题目】

二叉树可以用常规的三种遍历结果来描述其结构，但是不够直观，尤其是二叉树中有重复值的时候，仅通过三种遍历的结果来构造二叉树的真实结构更是难上加难，有时则根本不可能。给定一棵二叉树的头节点 head，已知二叉树节点值的类型为 32 位整型，请实现一个打印二叉树的函数，可以直观地展示树的形状，也便于画出真实的结构。

【难度】

尉 ★★☆☆

【解答】

这是一道较开放的题目，面试者不仅要设计出符合要求且不会产生歧义的打印方式，还要考虑实现难度，在面试时仅仅写出思路必然是不满足代码面试要求的。本书给出一种符合要求且代码量不大的实现，希望读者也能实现并优化自己的设计。具体过程如下：

1．设计打印的样式。实现者首先应该解决的问题是用什么样的方式来无歧义地打印二叉树。比如，二叉树如图 3-4 所示。

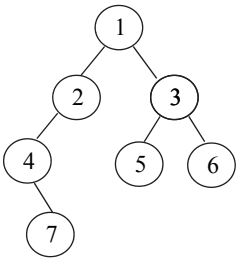

图 3-4

对如图 3-4 所示的二叉树，本书设计的打印样式如图 3-5 所示。

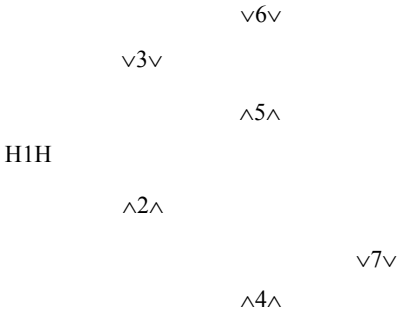

图 3-5

下面解释一下如何看打印的结果。首先，二叉树大概的样子是把打印结果顺时针旋转 90°，读者可以把图 3-4 的打印结果（也就是图 3-5 顺时针旋转 90°之后）做一下对比，两幅图是存在明显对应关系的；接下来，怎么清晰地确定任何一个节点的父节点呢？如果一个节点打印结果的前缀与后缀都有“H”(比如图 3-5 中的“H1H”)，说明这个节点是头节点，当然就不存在父节点。如果一个节点打印结果的前缀与后缀都有“v”，表示父节点在该节点所在列的前一列，在该节点所在行的下方，并且是离该节点最近的节点。比如图 3-5 中的“v3v”、“v6v”和“v7v”，父节点分别为“H1H”、“v3v”和“^4^”。如果一个节点打印结果的前缀与后缀都有“^”，表示父节点在该节点所在列的前一列，在该节点所在行的上方，并且是离该节点最近的节点。比如，图 3-5 中的“^5^”、“^2^”和“^4^”，父节点分别为“v3v”、“H1H”和“^2^”。

2．一个需要重点考虑的问题——规定节点打印时占用的统一长度。我们必须规定一个节点在打印时到底占多长。试想一下，如果有些节点的值本身的长度很短，比如“1”、“2”等，而有些节点的值本身的长度很长，比如“43323232”、“78787237”等，那么如果不规定一个统一的长度，在打印一个长短值交替的二叉树时必然会出现格式对不齐的问题，进而产生歧义。在 Java 中，整型值占用长度最长的值是 Integer.MIN_VALUE（即-2147483648），占用的长度为 11，加上前缀和后缀(“H”、“v”或“^”)之后占用长度为 13。为了在打印之后更好地区分，再把前面加上两个空格，后面加上两个空格，总共占用长度为 17。也就是说，长度为 17 的空间必然可以放下任何一个 32 位整数，同时样式还不错。至此，我们约定，打印每一个节点的时候，必须让每一个节点在打印时占用长度都为 17，如果不足，前后都用空格补齐。比如节点值为 8，假设这个节点加上“v”作为前后缀，那么实质内容为“v8v”，长度才为 3，在打印时在“v8v”的前面补 7 个空格，后面也补 7 个空格，让总

长度为 17。再如节点值为 66，假设这个节点加上“v”作为前后缀，那么实质内容为“v66v”，长度才为 4，在打印时在“v66v”的前面补 6 个空格，后面补 7 个空格，让总长度为 17。总之，如果长度不足，前后贴上几乎数量相等的空格来补齐。

3．确定了打印的样式，规定了占用长度的标准，最后来解释具体的实现。打印的整体过程结合了二叉树先右子树、再根节点、最后左子树的递归遍历过程。如果递归到一个节点，首先遍历它的右子树。右子树遍历结束后，回到这个节点。如果这个节点所在层为 l，那么先打印 l×17 个空格（不换行），然后开始制作该节点的打印内容，这个内容当然包括节点的值，以及确定的前后缀字符。如果该节点是其父节点的右孩子，前后缀为“v”，如果是其父节点的左孩子，前后缀为“^”，如果是头节点，前后缀为“H”。最后在前后分别贴上数量几乎一致的空格，占用长度为 17 的打印内容就制作完了，打印这个内容后换行。最后进行左子树的遍历过程。

直观地打印二叉树的所有过程请参看如下代码中的 printTree 方法。

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

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

public void printTree(Node head) {
        System.out.println("Binary Tree:");
        printInOrder(head, 0, "H", 17);
        System.out.println();
}

public void printInOrder(Node head, int height, String to, int len) {
        if (head == null) {
                return;
        }
        printInOrder(head.right, height + 1, "v", len);
        String val = to + head.value + to;
        int lenM = val.length();
        int lenL = (len - lenM) / 2;
        int lenR = len - lenM - lenL;
        val = getSpace(lenL) + val + getSpace(lenR);
        System.out.println(getSpace(height * len) + val);
        printInOrder(head.left, height + 1, "^", len);
}

public String getSpace(int num) {
```

```
            String space = " ";
            StringBuffer buf = new StringBuffer("");
            for (int i = 0; i < num; i++) {
                    buf.append(space);
            }
            return buf.toString();
    }
```

【扩展】

有关功能设计的面试题，其实最难的部分并不是设计，而是在设计的优良性和实现的复杂程度之间找到一个平衡性最好的设计方案。在满足功能要求的同时，也要保证在面试场上能够完成大致的代码实现，同时对边界条件的梳理能力和代码逻辑的实现能力也是一大挑战。读者可以看到本书提供的方法在完成功能的同时其代码很少，也请读者设计自己的方案并实现它。

二叉树的序列化和反序列化

【题目】

二叉树被记录成文件的过程叫作二叉树的序列化，通过文件内容重建原来二叉树的过程叫作二叉树的反序列化。给定一棵二叉树的头节点 head，并已知二叉树节点值的类型为 32 位整型。请设计一种二叉树序列化和反序列化的方案，并用代码实现。

【难度】

士 ★☆☆☆

【解答】

本书提供两套序列化和反序列化的实现，供读者参考。

方法一：通过先序遍历实现序列化和反序列化。

先介绍先序遍历下的序列化过程，首先假设序列化的结果字符串为 str，初始时 str=""。先序遍历二叉树，如果遇到 null 节点，就在 str 的末尾加上“#!”，“#”表示这个节点为空，节点值不存在，“!”表示一个值的结束；如果遇到不为空的节点，假设节点值为 3，就在 str 的末尾加上“3!”。比如图 3-6 所示的二叉树。

根据上文的描述，先序遍历序列化，最后的结果字符串 str 为：12!3!#!#!#!。

为什么在每一个节点值的后面都要加上“!”呢？因为如果不标记一个值的结束，最后产生的结果会有歧义，如图 3-7 所示。

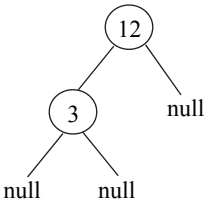

图 3-6

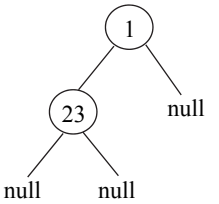

图 3-7

如果不在一个值结束时加入特殊字符，那么图 3-6 和图 3-7 的先序遍历序列化结果都是 123###。也就是说，生成的字符串并不代表唯一的树。

先序遍历序列化的全部过程请参看如下代码中的 serialByPre 方法。

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

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

public String serialByPre(Node head) {
        if (head == null) {
                return "#!";
        }
        String res = head.value + "!";
        res += serialByPre(head.left);
        res += serialByPre(head.right);
        return res;
}
```

接下来介绍如何通过先序遍历序列化的结果字符串 str，重构二叉树的过程，即反序列化。

把结果字符串 str 变成字符串类型的数组，记为 values，数组代表一棵二叉树先序遍历的节点顺序。例如，str="12!3!#!#!#! "，生成的 values 为["12","3","#","#","#"]，然后用 values[0..4]按照先序遍历的顺序建立整棵树。

1．遇到"12"，生成节点值为 12 的节点(head)，然后用 values[1..4]建立节点 12 的左子树。

2．遇到"3"，生成节点值为 3 的节点，它是节点 12 的左孩子，然后用 values[2..4]建立节点 3 的左子树。

3．遇到"#"，生成 null 节点，它是节点 3 的左孩子，该节点为 null，所以这个节点没有后续建立子树的过程。回到节点 3 后，用 values[3..4]建立节点 3 的右子树。

4．遇到"#"，生成 null 节点，它是节点 3 的右孩子，该节点为 null，所以这个节点没有后续建立子树的过程。回到节点 3 后，再回到节点 1，用 values[4]建立节点 1 的右子树。

5．遇到"#"，生成 null 节点，它是节点 1 的右孩子，该节点为 null，所以这个节点没有后续建立子树的过程。整个过程结束。

先序遍历反序列化的全部过程请参看如下代码中的 reconByPreString 方法。

```
public Node reconByPreString(String preStr) {
        String[] values = preStr.split("!");
        Queue<String> queue = new LinkedList<String>();
        for (int i = 0; i != values.length; i++) {
                queue.offer(values[i]);
        }
        return reconPreOrder(queue);
}

public Node reconPreOrder(Queue<String> queue) {
        String value = queue.poll();
        if (value.equals("#")) {
                return null;
        }
        Node head = new Node(Integer.valueOf(value));
        head.left = reconPreOrder(queue);
        head.right = reconPreOrder(queue);
        return head;
}
```

方法二：通过层遍历实现序列化和反序列化。

先介绍层遍历下的序列化过程，首先假设序列化的结果字符串为 str，初始时 str="空"。然后实现二叉树的按层遍历，具体方式是利用队列结构，这也是宽度遍历图的常见方式。例如，图 3-8 所示的二叉树。

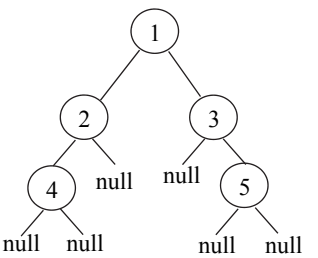

图 3-8

按层遍历图 3-8 所示的二叉树，最后 str="1!2!3!4!#!#!5!#!#!#!#! "。

层遍历序列化的全部过程请参看如下代码中的 serialByLevel 方法。

```
public String serialByLevel(Node head) {
        if (head == null) {
                return "#!";
        }
        String res = head.value + "!";
        Queue<Node> queue = new LinkedList<Node>();
        queue.offer(head);
        while (!queue.isEmpty()) {
                head = queue.poll();
                if (head.left != null) {
                        res += head.left.value + "!";
                        queue.offer(head.left);
                } else {
                        res += "#!";
                }
                if (head.right != null) {
                        res += head.right.value + "!";
                        queue.offer(head.right);
                } else {
                        res += "#!";
                }
        }
        return res;
}
```

先序遍历的反序列化其实就是重做先序遍历，遇到"#"就生成 null 节点，结束生成后续子树的过程。

与根据先序遍历的反序列化过程一样，根据层遍历的反序列化是重做层遍历，遇到"#"就生成 null 节点，同时不把 null 节点放到队列里即可。

层遍历反序列化的全部过程请参看如下代码中的 reconByLevelString 方法。

```
public Node reconByLevelString(String levelStr) {
        String[] values = levelStr.split("!");
        int index = 0;
        Node head = generateNodeByString(values[index++]);
        Queue<Node> queue = new LinkedList<Node>();
        if (head != null) {
                queue.offer(head);
        }
        Node node = null;
        while (!queue.isEmpty()) {
                node = queue.poll();
                node.left = generateNodeByString(values[index++]);
                node.right = generateNodeByString(values[index++]);
                if (node.left != null) {
```

```
                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：

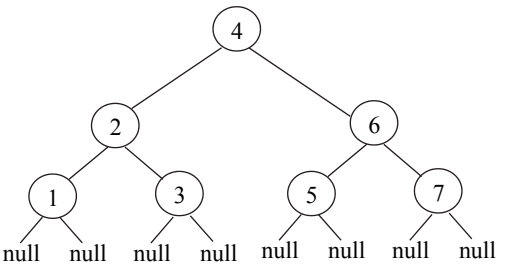

图 3-9

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。

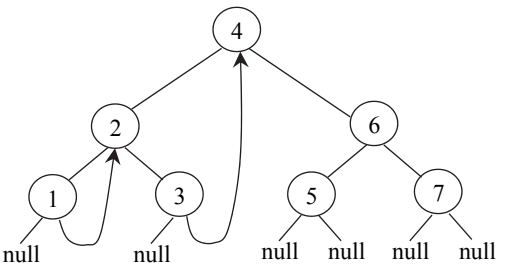

图 3-10

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。

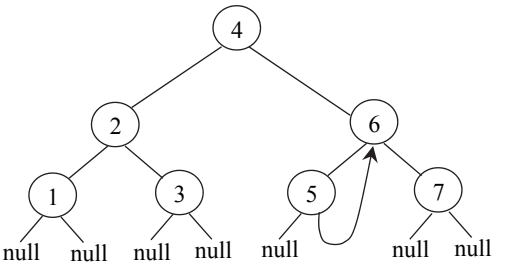

图 3-11

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。

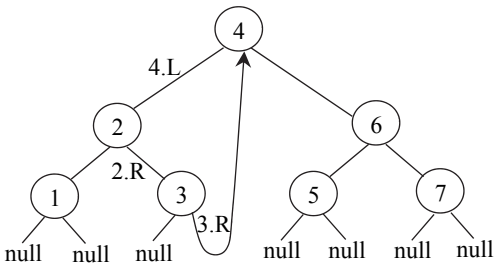

图 3-12

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。

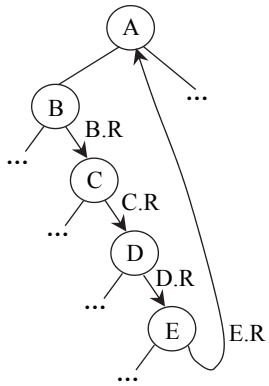

图 3-13

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。

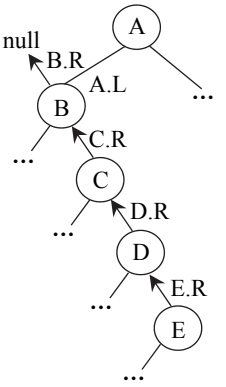

图 3-14

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;
}
```

在二叉树中找到累加和为指定值的最长路径长度

【题目】

给定一棵二叉树的头节点 head 和一个 32 位整数 sum，二叉树节点值类型为整型，求累加和为 sum 的最长路径长度。路径是指从某个节点往下，每次最多选择一个孩子节点或者不选所形成的节点链。

例如，二叉树如图 3-15 所示。

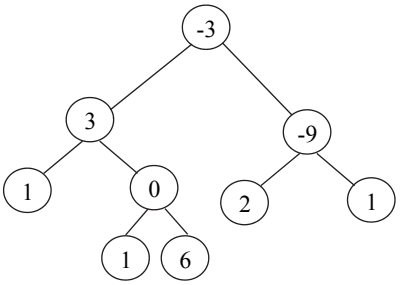

图 3-15

如果 sum=6，那么累加和为 6 的最长路径为：-3，3，0，6，所以返回 4。

如果 sum=-9，那么累加和为-9 的最长路径为：-9，所以返回 1。

注：本题不用考虑节点值相加可能溢出的情况。

【难度】

尉 ★★☆☆

【解答】

在阅读本题的解答之前，请读者先阅读本书“求未排序数组中累加和为规定值的最长子数组长度”问题。针对二叉树，本文的解法改写了这个问题的实现。如果二叉树的节点数为 N，本文的解法可以做到时间复杂度为 $O(N)$，额外空间复杂度为 $O(h)$，其中，h 为二叉树的高度。

具体过程如下：

1．二叉树头节点 head 和规定值 sum 已知；生成变量 maxLen，负责记录累加和等于

sum 的最长路径长度。

2．生成哈希表 sumMap。在“求未排序数组中累加和为规定值的最长子数组长度”问题中也使用了哈希表，功能是记录数组从左到右的累加和出现情况，在遍历数组的过程中，再利用这个哈希表来求得累加和为规定值的最长子数组。sumMap 也一样，它负责记录从 head 开始的一条路径上的累加和出现情况，累加和也是从 head 节点的值开始累加的。sumMap 的 key 值代表某个累加和，value 值代表这个累加和在路径中最早出现的层数。如果在遍历到 cur 节点的时候，我们能够知道从 head 到 cur 节点这条路径上的累加和出现情况，那么求以 cur 节点结尾的累加和为指定值的最长路径长度就非常容易。究竟如何去更新 sumMap，才能够做到在遍历到任何一个节点的时候都能有从 head 到这个节点的路径上的累加和出现情况呢？步骤 3 详细地说明了更新过程。

3．首先在 sumMap 中加入一个记录(0,0)，它表示累加和 0 不用包括任何节点就可以得到。然后按照二叉树先序遍历的方式遍历节点，遍历到的当前节点记为 cur，从 head 到 cur 父节点的累加和记为 preSum，cur 所在的层数记为 level。将 cur.value+preSum 的值记为 curSum，就是从 head 到 cur 的累加和。如果 sumMap 中已经包含了 curSum 的记录，说明 curSum 在上层中已经出现过，那么就不更新 sumMap；如果 sumMap 不包含 curSum 的记录，说明 curSum 是第一次出现，就把(curSum,level)这个记录放入 sumMap。接下来是求解在必须以 cur 结尾的情况下，累加和为规定值的最长路径长度，详细过程这里不再详述，请读者阅读“求未排序数组中累加和为规定值的最长子数组长度”问题。然后是遍历 cur 左子树和右子树的过程，依然按照步骤 3 描述的使用和更新 sumMap。以 cur 为头节点的子树处理完，当然要返回到 cur 父节点，在返回前还有一项重要的工作要做，在 sumMap 中查询 curSum 这个累加和（key）出现的层数（value），如果 value 等于 level，说明 curSum 这个累加和的记录是在遍历到 cur 时加上去的，那就把这一条记录删除；如果 value 不等于 level，则不做任何调整。

4．步骤 3 会遍历二叉树所有的节点，也会求解以每个节点结尾的情况下，累加和为规定值的最长路径长度。用 maxLen 记录其中的最大值即可。

全部求解过程请参看如下代码中的 getMaxLength 方法。

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

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

```
public int getMaxLength(Node head, int sum) {
        HashMap<Integer, Integer> sumMap = new HashMap<Integer, Integer>();
        sumMap.put(0, 0); // 重要
        return preOrder(head, sum, 0, 1, 0, sumMap);
}

public int preOrder(Node head, int sum, int preSum, int level,
                int maxLen, HashMap<Integer, Integer> sumMap) {
        if (head == null) {
                return maxLen;
        }
        int curSum = preSum + head.value;
        if (!sumMap.containsKey(curSum)) {
                sumMap.put(curSum, level);
        }
        if (sumMap.containsKey(curSum - sum)) {
                maxLen = Math.max(level - sumMap.get(curSum - sum), maxLen);
        }
        maxLen = preOrder(head.left, sum, curSum, level + 1, maxLen, sumMap);
        maxLen = preOrder(head.right, sum, curSum, level + 1, maxLen, sumMap);
        if (level == sumMap.get(curSum)) {
                sumMap.remove(curSum);
        }
        return maxLen;
}
```

找到二叉树中的最大搜索二叉子树

【题目】

给定一棵二叉树的头节点 head，已知其中所有节点的值都不一样，找到含有节点最多的搜索二叉子树，并返回这棵子树的头节点。

例如，二叉树如图 3-16 所示。

这棵树中的最大搜索二叉子树如图 3-17 所示。

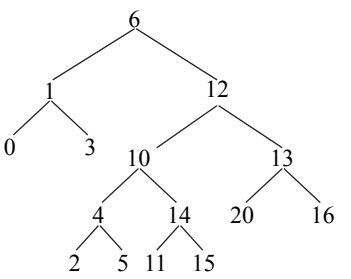

图 3-16

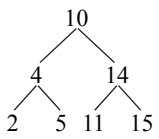

图 3-17

【要求】

如果节点数为 N，要求时间复杂度为 $O(N)$，额外空间复杂度为 $O(h)$，h 为二叉树的高度。

【难度】

尉 ★★☆☆

【解答】

以节点 node 为头的树中，最大的搜索二叉子树只可能来自以下两种情况。

第一种：如果来自 node 左子树上的最大搜索二叉子树是以 node.left 为头的；来自 node 右子树上的最大搜索二叉子树是以 node.right 为头的；node 左子树上的最大搜索二叉子树的最大值小于 node.value；node 右子树上的最大搜索二叉子树的最小值大于 node.value，那么以节点 node 为头的整棵树都是搜索二叉树。

第二种：如果不满足第一种情况，说明以节点 node 为头的树整体不能连成搜索二叉树。这种情况下，以 node 为头的树上的最大搜索二叉子树是来自 node 的左子树上的最大搜索二叉子树和来自 node 的右子树上的最大搜索二叉子树之间，节点数较多的那个。

通过以上分析，求解的具体过程如下：

1．整体过程是二叉树的后序遍历。

2．遍历到当前节点记为 cur 时，先遍历 cur 的左子树收集 4 个信息，分别是左子树上最大搜索二叉子树的头节点（lBST）、节点数（lSize）、最小值（lMin）和最大值（lMax）。再遍历 cur 的右子树收集 4 个信息，分别是右子树上最大搜索二叉子树的头节点（rBST）、节点数（rSize）、最小值（rMin）和最大值（rMax）。

3．根据步骤 2 所收集的信息，判断是否满足第一种情况，如果满足第一种情况，就返回 cur 节点，如果满足第二种情况，就返回 lBST 和 rBST 中较大的一个。

4．可以使用全局变量的方式实现步骤 2 中收集节点数、最小值和最大值的问题。

找到最大搜索二叉子树的具体过程请参看如下代码中的 biggestSubBST 方法。

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

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

```
public Node biggestSubBST(Node head) {
        int[] record = new int[3];
        return posOrder(head, record);
}

public Node posOrder(Node head, int[] record) {
        if (head == null) {
                record[0] = 0;
                record[1] = Integer.MAX_VALUE;
                record[2] = Integer.MIN_VALUE;
                return null;
        }
        int value = head.value;
        Node left = head.left;
        Node right = head.right;
        Node lBST = posOrder(left, record);
        int lSize = record[0];
        int lMin = record[1];
        int lMax = record[2];
        Node rBST = posOrder(right, record);
        int rSize = record[0];
        int rMin = record[1];
        int rMax = record[2];
        record[1] = Math.min(lMin, value);
        record[2] = Math.max(rMax, value);
        if (left == lBST && right == rBST && lMax < value && value < rMin) {
                record[0] = lSize + rSize + 1;
                return head;
        }
        record[0] = Math.max(lSize, rSize);
        return lSize > rSize ? lBST : rBST;
}
```

找到二叉树中符合搜索二叉树条件的最大拓扑结构

【题目】

给定一棵二叉树的头节点 head，已知所有节点的值都不一样，返回其中最大的且符合搜索二叉树条件的最大拓扑结构的大小。

例如，二叉树如图 3-18 所示。

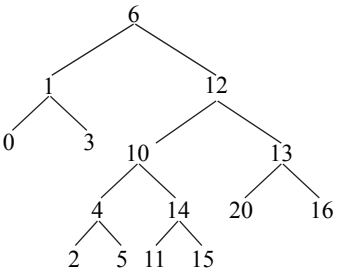

图 3-18

其中最大的且符合搜索二叉树条件的最大拓扑结构如图 3-19 所示。

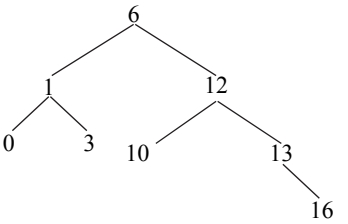

图 3-19

这个拓扑结构节点数为 8，所以返回 8。

【难度】

校 ★★★☆

【解答】

方法一：二叉树的节点数为 N，时间复杂度为 $O(N^2)$的方法。

首先来看这样一个问题，以节点 h 为头的树中，在拓扑结构中也必须以 h 为头的情况下，怎么找到符合搜索二叉树条件的最大结构？这个问题有一种比较容易理解的解法，我们先考查 h 的孩子节点，根据孩子节点的值从 h 开始按照二叉搜索的方式移动，如果最后能移动到同一个孩子节点上，说明这个孩子节点可以作为这个拓扑的一部分，并继续考查这个孩子节点的孩子节点，一直延伸下去。

我们以题目的例子来说明一下，假设在以 12 这个节点为头的子树中，要求拓扑结构也必须以 12 为头，如何找到最多的节点，并且整个拓扑结构是符合二叉树条件的？初始时考

查的节点为 12 节点的左右孩子，考查队列={10,13}。

考查节点 10。最开始时 10 和 12 进行比较，发现 10 应该往 12 的左边找，于是节点 10 被找到，节点 10 可以加入整个拓扑结构，同时节点 10 的孩子节点 4 和 14 加入考查队列，考查队列为{13,4,14}。

考查节点 13。13 和 12 进行比较，应该向右，于是节点 13 被找到，它可以加入整个拓扑结构，同时它的两个孩子节点 20 和 16 加入考查队列，{4,14,20,16}。

考查节点 4。4 和 12 比较，应该向左，4 和 10 比较，继续向左，节点 4 被找到，可以加入整个拓扑结构。同时它的孩子节点 2 和 5 加入考查队列，为{14,20,16,2,5}。

考查节点 14。14 和 12 比较，应该向右，接下来的查找过程会一直在 12 的右子树上，依然会找下去，但是节点 14 不可能被找到。所以它不能加入整个拓扑结构，它的孩子节点也都不能，此时考查队列为{20,16,2,5}。

考查节点 20。20 和 12 比较，应该向右，20 和 13 比较，应该向右，节点 20 同样再也不会被发现了，所以它不能加入整个拓扑结构，此时考查队列为{16,2,5}。

按照如上方法，最后这三个节点（16,2,5）都可以加入拓扑结构，所以我们找到了必须以 12 为头，且整个拓扑结构是符合二叉树条件的最大结构，这个结构的节点数为 7。

也就是说，我们根据一个节点的值，根据这个值的大小，从 h 开始，每次向左或者向右移动，如果最后能移动到原来的节点上，说明该节点可以作为以 h 为头的拓扑的一部分。

解决了以节点 h 为头的树中，在拓扑结构也必须以 h 为头的情况下，怎么找到符合搜索二叉树条件的最大结构？接下来只要遍历所有的二叉树节点，并在以每个节点为头的子树中都求一遍其中的最大拓扑结构，其中最大的那个就是我们想找的结构，它的大小就是我们的返回值。

具体过程请参看如下代码中的 bstTopoSize1 方法。

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

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

public int bstTopoSize1(Node head) {
        if (head == null) {
                return 0;
        }
        int max = maxTopo(head, head);
```

```
        max = Math.max(bstTopoSize1(head.left), max);
        max = Math.max(bstTopoSize1(head.right), max);
        return max;
}

public int maxTopo(Node h, Node n) {
        if (h != null && n != null && isBSTNode(h, n, n.value)) {
                return maxTopo(h, n.left) + maxTopo(h, n.right) + 1;
        }
        return 0;
}

public boolean isBSTNode(Node h, Node n, int value) {
        if (h == null) {
                return false;
        }
        if (h == n) {
                return true;
        }
        return isBSTNode(h.value > value ? h.left : h.right, n, value);
}
```

对于方法一的时间复杂度分析，我们把所有的子树(N 个)都找了一次最大拓扑，每找一次所考查的节点数都可能是 $O(N)$个节点，所以方法一的时间复杂度为 $O(N^2)$。

方法二：二叉树的节点数为 N、时间复杂度最好为 $O(N)$、最差为 $O(N\log N)$的方法。

先来说明一个对方法二来讲非常重要的概念——拓扑贡献记录。还是举例说明，请注意题目中以节点 10 为头的子树，这棵子树本身就是一棵搜索二叉树，那么整棵子树都可以作为以节点 10 为头的符合搜索二叉树条件的拓扑结构。如果对这个拓扑结构建立贡献记录，是如图 3-20 所示的样子。

在图 3-20 中，每个节点的旁边都有被括号括起来的两个值，我们把它称为节点对当前头节点的拓扑贡献记录。第一个值代表节点的左子树可以为当前头节点的拓扑贡献几个节点，第二个值代表节点的右子树可以为当前头节点的拓扑贡献几个节点。比如 4(1,1)，括号中的第一个 1 代表节点 4 的左子树可以为节点 10 为头的拓扑结构贡献 1 个节点，第二个 1 代表节点 4 的右子树可以为节点 10 为头的拓扑结构贡献 1 个节点。同样，我们也可以建立以节点 13 为头的记录，如图 3-21 所示。

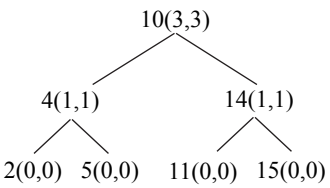

图 3-20

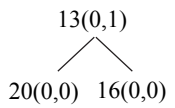

图 3-21

整个方法二的核心就是如果分别得到了 h 左右两个孩子为头的拓扑贡献记录，可以快速得到以 h 为头的拓扑贡献记录。比如图 3-20 中每一个节点的记录都是节点对以节点 10 为头的拓扑结构的贡献记录，图 3-21 中每一个节点的记录都是节点对以节点 13 为头的拓扑结构的贡献记录，同时节点 10 和节点 13 分别是节点 12 的左孩子和右孩子。那么我们可以快速得到以节点 12 为头的拓扑贡献记录。在图 3-20 和图 3-21 中的所有节点的记录还没有变成节点 12 为头的拓扑贡献记录之前，是图 3-22 所示的样子。

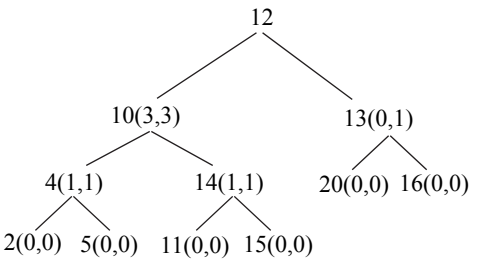

图 3-22

如图 3-22 所示，在没有变更之前，节点 12 左子树上所有节点的记录和原来一样，都是对节点 10 负责的；节点 12 右子树上所有节点的记录也和原来一样，都是对节点 13 负责的。接下来我们详细展示一下，所有节点的记录如何变更为都对节点 12 负责，也就是所有节点的记录都变成以节点 12 为头的拓扑贡献记录。

先来看节点 12 的左子树，只需依次考查左子树右边界上的节点即可。先考查节点 10，因为节点 10 的值比节点 12 的值小，所以节点 10 的左子树原来能给节点 10 贡献多少个节点，当前就一定都能贡献给节点 12，所以节点 10 记录的第一个值不用改变，同时节点 10 左子树上所有节点的记录都不用改变。接下来考查节点 14，此时节点 14 的值比节点 10 要大，说明以节点 14 为头的整棵子树都不能成为以节点 12 为头的拓扑结构的左边部分，那么删掉节点 14 的记录，让它不作为节点 12 为头的拓扑结构即可，同时只要删掉节点 14 一条记录，就可以断开节点 11 和节点 15 的记录，让节点 14 的整棵子树都不成为节点 12 的拓扑结构。后续的右边界节点也无须考查了。进行到节点 14 这一步，一共删掉的节点数可以直接通过节点 14 的记录得到，记录为 14(1,1)，说明节点 14 的左子树 1 个，节点 14 的右子树 1 个，再加上节点 14 本身，一共有 3 个节点。接下来的过程是从右边界的当前节点重回节点 12 的过程，先回到节点 10，此时节点 10 记录的第二个值应该被修改，因为节点 10 的右子树上被删掉了 3 个节点，所以记录由 10(3,3)修改为 10(3,0)，根据这个修改后的记录，节点 12 记录的第一个值也可以确定了，节点 12 的左子树可以贡献 4 个节点，其中 3

个来自节点 10 的左子树，还有 1 个是节点 10 本身，此时记录变为图 3-23 所示的样子。

以上过程展示了怎么把关于 h 左孩子的拓扑贡献记录更改为以 h 为头的拓扑贡献记录。为了更好地展示这个过程，我们再举一个例子，如图 3-24 所示。

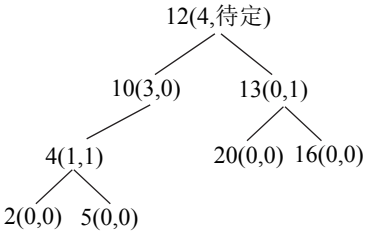

图 3-23

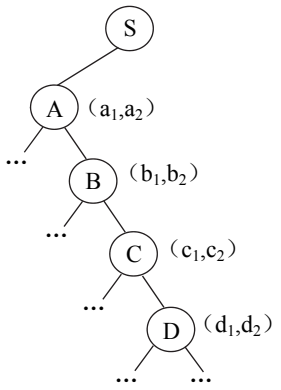

图 3-24

在图 3-24 中，假设之前已经有以节点 A 为头的拓扑贡献记录，现在要变更为以节点 S 为头的拓扑贡献记录。只用考查 S 左子树的右边界即可(A,B,C,D...)，假设 A，B，C 的值都比 S 小，到节点 D 才比节点 S 大。那么 A，B，C 的左子树原来能给 A 的拓扑贡献多少个节点，现在就都能贡献给 S，所以这三个节点记录的第一个值一律不发生变化，并且它们所有左子树上的节点记录也不用变化。而 D 的值比 S 的值大，所以删除 D 的记录，从而让 D 子树上的所有记录都和以 S 为头的拓扑结构断开，总共删掉的节点数为 $d1+d2+1$。然后再从 C 回到 S，沿途所有节点记录的第二个值统一减掉 $d1+d2+1$。最后根据节点 A 改变后的记录，确定 S 记录的第一个值，如图 3-25 所示。

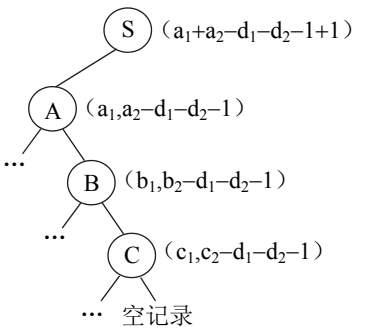

图 3-25

关于怎么把 h 左孩子的拓扑贡献记录更改为以 h 为头的拓扑贡献记录的问题就解释完了。把关于 h 右孩子的拓扑贡献记录更改为以 h 为头的拓扑贡献记录与之类似，就是依次考查 h 右子树的左边界即可。回到以节点 12 为头的拓扑贡献记录问题，最后生成的整个记录如图 3-26 所示。

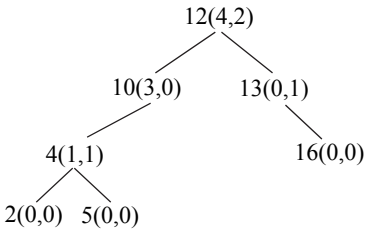

图 3-26

当我们得到以 h 为头的拓扑贡献记录后，相当于求出了以 h 为头的最大拓扑的大小。方法二正是不断地用这种方法，从小树的记录整合成大树的记录，从而求出整棵树中符合搜索二叉树条件的最大拓扑的大小。所以，整个过程大体说来是利用二叉树的后序遍历，对每个节点来说，先生成其左孩子的记录，然后是右孩子的记录，接着把两组记录修改成以这个节点为头的拓扑贡献记录，并找出所有节点的最大拓扑大小中最大的那个。

方法二的全部过程请参看如下代码中的 bstTopoSize2 方法。

```
public class Record {
        public int l;
        public int r;

        public Record(int left, int right) {
                this.l = left;
                this.r = right;
        }
}

public int bstTopoSize2(Node head) {
        Map<Node, Record> map = new HashMap<Node, Record>();
        return posOrder(head, map);
}

public int posOrder(Node h, Map<Node, Record> map) {
        if (h == null) {
                return 0;
        }
        int ls = posOrder(h.left, map);
        int rs = posOrder(h.right, map);
        modifyMap(h.left, h.value, map, true);
```

```
        modifyMap(h.right, h.value, map, false);
        Record lr = map.get(h.left);
        Record rr = map.get(h.right);
        int lbst = lr == null ? 0 : lr.l + lr.r + 1;
        int rbst = rr == null ? 0 : rr.l + rr.r + 1;
        map.put(h, new Record(lbst, rbst));
        return Math.max(lbst + rbst + 1, Math.max(ls, rs));
}

public int modifyMap(Node n, int v, Map<Node, Record> m, boolean s) {
        if (n == null || (!m.containsKey(n))) {
                return 0;
        }
        Record r = m.get(n);
        if ((s && n.value > v) || ((!s) && n.value < v)) {
                m.remove(n);
                return r.l + r.r + 1;
        } else {
                int minus = modifyMap(s ? n.right : n.left, v, m, s);
                if (s) {
                        r.r = r.r - minus;
                } else {
                        r.l = r.l - minus;
                }
                m.put(n, r);
                return minus;
        }
}
```

对于方法二的时间复杂度分析，如果二叉树类似棒状结构，即每一个非叶节点只有左子树或只有右子树，如图 3-27 所示。

在图 3-27 的二叉树中，假设节点 a 到节点 c 的若干节点只有右子树记为区域 A，从节点 d 到节点 f 的若干节点只有左子树记为区域 B，从节点 g 到节点 i 的若干节点只有右子树记为区域 C，从节点 j 到节点 k 的若干节点又只有左子树记为区域 D。如果二叉树是这种形状，并且整棵二叉树都符合搜索二叉树条件，现在我们分析一下在方法二的整个过程中将走过多少个节点。

区域 D：区域 D 的每个节点在生成自己的记录时，只有左子树记录，同时自己左子树的右边界只有自己的左孩子。所以对区域 D 的所有节点来说，每一个节点都只检查一个节点，就是自己的左孩子，所以走过节点的总数量就是区域 D 的节点数，记为 numD。

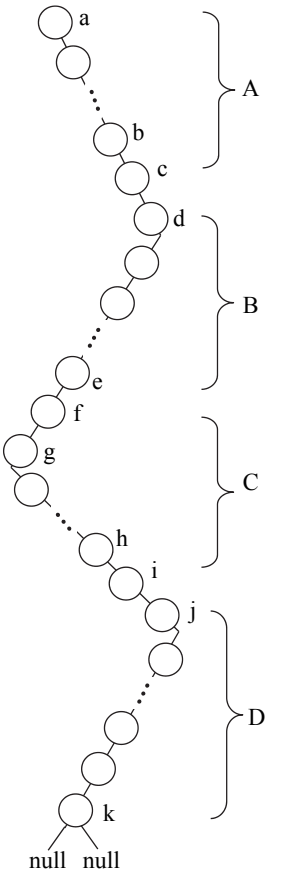

图 3-27

区域 C：在区域 C 中的节点 i 很特殊，这个节点右子树的左边界是区域 D 的全部节点，全部都要走一遍，数量为 numD。除这个节点外，区域 C 中的其他节点又是只走过一个节点，是自己的右孩子，走过节点的总数量相当于 C 区域的节点数，记为 numC。处理区域 C 时走过的总数量为 numD+numC。

区域 B 同理，总数量为 numB+numC。

区域 A 同理，总数量为 numA+numB。

所以，如果二叉树的节点数为 N，那么整个过程走过的节点数大致为 $2N$，时间复杂度为 $O(N)$。这是方法二最好的情况，也就是二叉树趋近于棒状结构的时候。

如果二叉树是满二叉树结构，即每一个非节点左子树和右子树全都有，如图 3-28 所示。

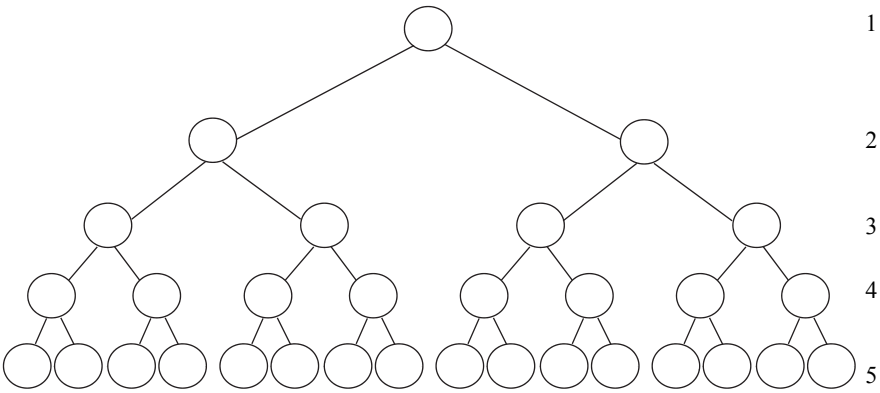

图 3-28

图 3-28 的二叉树为一棵满二叉树结构，层数为 5。

第 1 层的节点数量为 1，第 1 层的节点在生成记录时左子树的右边界节点数为 4，右子树的左边界节点数为 4，总共走过 8 个节点。

第 2 层的节点数量为 2，第 2 层每个节点在生成记录时左子树的右边界节点数为 3，右子树的左边界节点数为 3，总共走过 12 个节点。

第 3 层的节点数量为 4，第 3 层每个节点在生成记录时左子树的右边界节点数为 2，右子树的左边界节点数为 2，总共走过 16 个节点。

……

我们做一下扩展，如果一棵满二叉树，层数为 l。

第 1 层的节点数量为 l，第 1 层的节点在生成记录时左子树的右边界节点数为 l-1，右子树的左边界节点数为 l-1，总共走过 $2(l\text{-}1)$个节点。

第 2 层的节点数量为 2，第 2 层的节点在生成记录时左子树的右边界节点数为 l-2，右子树的左边界节点数为 l-2，总共走过 $2\times2\times(l\text{-}1)$个节点。

……

第 i 层的节点数量为 2^{i-1}，第 i 层的节点在生成记录时左子树的右边界节点数为 l-i，右子树的左边界节点数为 l-i，总共走过$(2^{i-1})\times2\times(l\text{-}i)=2^{i}(l\text{-}i)$个节点。

……

所以全部层的所有节点走过的节点数为：

$$\sum_{i=1}^{i\leqslant l}(l-i)\times2^{i}=\frac{3}{2}\times l\times2^{l}+2^{l-1}-4$$

在满二叉树中，i -> $O(\log N)$，2^l -> N，所以走过的节点总数为 $O(N\log N)$。

二叉树越趋近于棒状结构，方法二的时间复杂度越低，也越趋近于 $O(N)$；二叉树越趋近于满二叉树结构，方法二的时间复杂度越高，但最差也仅仅是 $O(N\log N)$。

方法二的详细证明略。

二叉树的按层打印与 ZigZag 打印

【题目】

给定一棵二叉树的头节点 head，分别实现按层打印和 ZigZag 打印二叉树的函数。

例如，二叉树如图 3-29 所示。

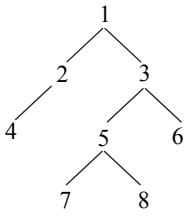

图 3-29

按层打印时，输出格式必须如下：

```
Level 1 : 1
Level 2 : 2 3
Level 3 : 4 5 6
Level 4 : 7 8
```

ZigZag 打印时，输出格式必须如下：

```
Level 1 from left to right: 1
Level 2 from right to left: 3 2
Level 3 from left to right: 4 5 6
Level 4 from right to left: 8 7
```

【难度】

尉 ★★☆☆

【解答】

- 按层打印的实现。

按层打印原本是十分基础的内容，对二叉树做简单的宽度优先遍历即可，但本题确有额外的要求，那就是同一层的节点必须打印在一行上，并且要求输出行号。这就需要我们在原来宽度优先遍历的基础上做一些改进。所以关键问题是如何知道该换行。只需要用两个 node 类型的变量 last 和 nLast 就可以解决这个问题，last 变量表示正在打印的当前行的最右节点，nLast 表示下一行的最右节点。假设我们每一层都做从左到右的宽度优先遍历，如果发现遍历到的节点等于 last，说明该换行了。换行之后只要令 last=nLast，就可以继续下一行的打印过程，此过程重复，直到所有的节点都打印完。那么问题就变成了如何更新 nLast？只需要让 nLast 一直跟踪记录宽度优先队列中的最新加入的节点即可。这是因为最新加入队列的节点一定是目前已经发现的下一行的最右节点。所以在当前行打印完时，nLast 一定是下一行所有节点中的最右节点。接下来结合题目的例子来说明整个过程。

开始时，last=节点 1，nLast=null，把节点 1 放入队列 queue，遍历开始，queue={1}。

从 queue 中弹出节点 1 并打印，然后把节点 1 的孩子依次放入 queue，放入节点 2 时，nLast=节点 2，放入节点 3 时，nLast=节点 3，此时发现弹出的节点 1==last。所以换行，并令 last=nLast=节点 3，queue={2,3}。

从 queue 中弹出节点 2 并打印，然后把节点 2 的孩子放入 queue，放入节点 4 时，nLast=节点 4，queue={3,4}。

从 queue 中弹出节点 3 并打印，然后把节点 3 的孩子放入 queue，放入节点 5 时，nLast=节点 5，放入节点 6 时，nLast=节点 6，此时发现弹出的节点 3==last。所以换行，并令 last=nLast=节点 6，queue={4,5,6}。

从 queue 中弹出节点 4 并打印，节点 4 没有孩子，所以不放入任何节点，nLast 也不更新。

从 queue 中弹出节点 5 并打印，然后把节点 5 的孩子依次放入 queue，放入节点 7 时，nLast=节点 7，放入节点 8 时，nLast=节点 8，queue={6,7,8}。

从 queue 中弹出节点 6 并打印，节点 6 没有孩子，所以不放入任何节点，nLast 也不更新，此时发现弹出的节点 6==last。所以换行，并令 last=nLast=节点 8，queue={7,8}。

用同样的判断过程打印节点 7 和节点 8，整个过程结束。

按层打印的详细过程请参看如下代码中的 printByLevel 方法。

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

```
        public Node right;

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

public void printByLevel(Node head) {
        if (head == null) {
                return;
        }
        Queue<Node> queue = new LinkedList<Node>();
        int level = 1;
        Node last = head;
        Node nLast = null;
        queue.offer(head);
        System.out.print("Level " + (level++) + " : ");
        while (!queue.isEmpty()) {
                head = queue.poll();
                System.out.print(head.value + " ");
                if (head.left != null) {
                        queue.offer(head.left);
                        nLast = head.left;
                }
                if (head.right != null) {
                        queue.offer(head.right);
                        nLast = head.right;
                }
                if (head == last && !queue.isEmpty()) {
                        System.out.print("\nLevel " + (level++) + " : ");
                        last = nLast;
                }
        }
        System.out.println();
}
```

- ZigZag 打印的实现。

先简单介绍一种不推荐的方法，即使用 ArrayList 结构的方法。两个 ArrayList 结构记为 list1 和 list2，用 list1 去收集当前层的节点，然后从左到右打印当前层，接着把当前层的孩子节点放进 list2，并从右到左打印，接下来再把 list2 的所有节点的孩子节点放入 list1，如此反复。不推荐的原因是 ArrayList 结构为动态数组，在这个结构中，当元素数量到一定规模时将发生扩容操作，扩容操作的时间复杂度为 $O(N)$是比较高的，这个结构增加和删除元素的时间复杂度也较高。总之，用这个结构对本题来讲数据结构不够纯粹和干净，如果读者不充分理解这个结构的底层实现，最好不要使用，而且还需要两个 ArrayList 结构。

本书提供的方法只使用了一个双端队列，具体为 Java 中的 LinkedList 结构，这个结构的底层实现就是非常纯粹的双端队列结构，本书的方法也仅使用双端队列结构的基本操作。

先举题目的例子来展示大体过程，首先生成双端队列结构 dq，将节点 1 从 dq 的头部放入 dq。

原则 1：如果是从左到右的过程，那么一律从 dq 的头部弹出节点，如果弹出的节点没有孩子节点，当然不用放入任何节点到 dq 中；如果当前节点有孩子节点，先让左孩子从尾部进入 dq，再让右孩子从尾部进入 dq。

根据原则 1，先从 dq 头部弹出节点 1 并打印，然后先让节点 2 从 dq 尾部进入，再让节点 3 从 dq 尾部进入，如图 3-30 所示。

原则 2：如果是从右到左的过程，那么一律从 dq 的尾部弹出节点，如果弹出的节点没有孩子节点，当然不用放入任何节点到 dq 中；如果当前节点有孩子节点，先让右孩子从头部进入 dq，再让左孩子从头部进入 dq。

根据原则 2，先从 dq 尾部弹出节点 3 并打印，然后先让节点 6 从 dq 头部进入，再让节点 5 从 dq 头部进入，如图 3-31 所示。

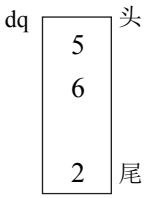

图 3-30

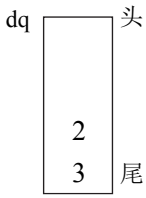

图 3-31

根据原则 2，先从 dq 尾部弹出节点 2 并打印，然后让节点 4 从 dq 头部进入，如图 3-32 所示。

根据原则 1，依次从 dq 头部弹出节点 4、5、6 并打印，这期间先让节点 7 从 dq 尾部进入，再让节点 8 从 dq 尾部进入，如图 3-33 所示。

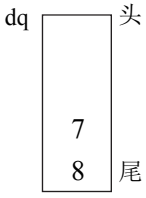

图 3-22

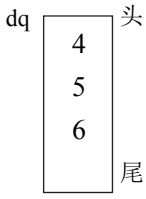

图 3-33

最后根据原则 2，依次从 dq 尾部弹出节点 8 和 7 并打印即可。

用原则 1 和原则 2 的过程切换，我们可以完成 ZigZag 的打印过程，所以现在只剩一个

问题，如何确定切换原则 1 和原则 2 的时机，其实还是如何确定每一层最后一个节点的问题。

在 ZigZag 的打印过程中，下一层最后打印的节点是当前层有孩子的节点中最先进入 dq 的节点。比如，处理第 1 层的第 1 个有孩子的节点，也就是节点 1 时，节点 1 的左孩子节点 2 最先进的 dq，那么节点 2 就是下一层打印时的最后一个节点。处理第 2 层的第一个有孩子的节点，也就是节点 3 时，节点 3 的右孩子节点 6 最先进的 dq，那么节点 6 就是下一层打印时的最后一个节点。处理第 3 层的第一个有孩子的节点，也就是节点 5 时，节点 5 的左孩子节点 7 最先进的 dq，那么节点 7 就是下一层打印时的最后一个节点。

ZigZag 打印的全部过程请参看如下代码中的 printByZigZag 方法。

```
public void printByZigZag(Node head) {
        if (head == null) {
                return;
        }
        Deque<Node> dq = new LinkedList<Node>();
        int level = 1;
        boolean lr = true;
        Node last = head;
        Node nLast = null;
        dq.offerFirst(head);
        pringLevelAndOrientation(level++, lr);
        while (!dq.isEmpty()) {
                if (lr) {
                        head = dq.pollFirst();
                        if (head.left != null) {
                                nLast = nLast == null ? head.left : nLast;
                                dq.offerLast(head.left);
                        }
                        if (head.right != null) {
                                nLast = nLast == null ? head.right : nLast;
                                dq.offerLast(head.right);
                        }
                } else {
                        head = dq.pollLast();
                        if (head.right != null) {
                                nLast = nLast == null ? head.right : nLast;
                                dq.offerFirst(head.right);
                        }
                        if (head.left != null) {
                                nLast = nLast == null ? head.left : nLast;
                                dq.offerFirst(head.left);
                        }
                }
                System.out.print(head.value + " ");
                if (head == last && !dq.isEmpty()) {
                        lr = !lr;
                        last = nLast;
                        nLast = null;
```

```
                        System.out.println();
                        pringLevelAndOrientation(level++, lr);
                    }
                }
                System.out.println();
        }

        public void pringLevelAndOrientation(int level, boolean lr) {
                System.out.print("Level " + level + " from ");
                System.out.print(lr ? "left to right: " : "right to left: ");
        }
```

调整搜索二叉树中两个错误的节点

【题目】

一棵二叉树原本是搜索二叉树，但是其中有两个节点调换了位置，使得这棵二叉树不再是搜索二叉树，请找到这两个错误节点并返回。已知二叉树中所有节点的值都不一样，给定二叉树的头节点 head，返回一个长度为 2 的二叉树节点类型的数组 errs，errs[0]表示一个错误节点，errs[1]表示另一个错误节点。

进阶：如果在原问题中得到了这两个错误节点，我们当然可以通过交换两个节点的节点值的方式让整棵二叉树重新成为搜索二叉树。但现在要求你不能这么做，而是在结构上完全交换两个节点的位置，请实现调整的函数。

【难度】

原问题：尉 ★★☆☆

进阶问题：将 ★★★★

【解答】

原问题——找到这两个错误节点。如果对所有的节点值都不一样的搜索二叉树进行中序遍历，那么出现的节点值会一直升序，所以，如果有两个节点位置错了，就一定会出现降序。

如果在中序遍历时节点值出现了两次降序，第一个错误的节点为第一次降序时较大的节点，第二个错误的节点为第二次降序时较小的节点。

比如，原来的搜索二叉树在中序遍历时的节点值依次出现{1，2，3，4，5}，如果因为

两个节点位置错了而出现{1，5，3，4，2}，第一次降序为 5->3，所以第一个错误节点为 5，第二次降序为 4->2，所以第二个错误节点为 2，把 5 和 2 换过来就可以恢复。

如果在中序遍历时节点值只出现了一次降序，第一个错误的节点为这次降序时较大的节点，第二个错误的节点为这次降序时较小的节点。

比如，原来的搜索二叉树在中序遍历时节点值依次出现{1，2，3，4，5}，如果因为两个节点位置错了而出现{1，2，4，3，5}，只有一次降序为 4->3，所以第一个错误节点为 4，第二个错误节点为 3，把 4 和 3 换过来就可以恢复。

寻找两个错误节点的过程可以总结为：第一个错误节点为第一次降序时较大的节点，第二个错误节点为最后一次降序时较小的节点。

所以，只要改写一个基本的中序遍历，就可以完成原问题的要求，改写递归、非递归或者 Morris 遍历都可以。

找到两个错误节点的过程请参看如下代码中的 getTwoErrNodes 方法。

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

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

public Node[] getTwoErrNodes(Node head) {
        Node[] errs = new Node[2];
        if (head == null) {
                return errs;
        }
        Stack<Node> stack = new Stack<Node>();
        Node pre = null;
        while (!stack.isEmpty() || head != null) {
                if (head != null) {
                        stack.push(head);
                        head = head.left;
                } else {
                        head = stack.pop();
                        if (pre != null && pre.value > head.value) {
                                errs[0] = errs[0] == null ? pre : errs[0];
                                errs[1] = head;
                        }
                        pre = head;
                        head = head.right;
                }
        }
```

```
        return errs;
}
```

进阶问题——在结构上交换这两个错误节点。若要在结构上交换两个错误节点，首先应该找到两个错误节点各自的父节点，随便改写一个二叉树的遍历即可。

找到两个错误节点各自父节点的过程请参看如下代码中的 getTwoErrParents 方法，该方法返回长度为 2 的 Node 类型的数组 parents，parents[0]表示第一个错误节点的父节点，parents[1]表示第二个错误节点的父节点。

```
public Node[] getTwoErrParents(Node head, Node e1, Node e2) {
        Node[] parents = new Node[2];
        if (head == null) {
                return parents;
        }
        Stack<Node> stack = new Stack<Node>();
        while (!stack.isEmpty() || head != null) {
                if (head != null) {
                        stack.push(head);
                        head = head.left;
                } else {
                        head = stack.pop();
                        if (head.left == e1 || head.right == e1) {
                                parents[0] = head;
                        }
                        if (head.left == e2 || head.right == e2) {
                                parents[1] = head;
                        }
                        head = head.right;
                }
        }
        return parents;
}
```

找到两个错误节点的父节点之后，第一个错误节点记为 e1，e1 的父节点记为 e1P，e1 的左孩子记为 e1L，e1 的右孩子记为 e1R。第二个错误节点记为 e2，e2 的父节点记为 e2P，e2 的左孩子记为 e2L，e2 的右孩子记为 e2R。

在结构上交换两个节点，实际上就是把两个节点互换环境。粗略地说，就是让 e2 成为 e1P 的孩子节点，让 e1L 和 e1R 成为 e2 的孩子节点；让 e1 成为 e2P 的孩子节点，让 e2L 和 e2R 成为 e1 的孩子节点。但这只是粗略的理解，在实际交换的过程中有很多情况需要我们做特殊处理。比如，如果 e1 是头节点，意味着 e1P 为 null，那么让 e2 成为 e1P 的孩子节点时，关于 e1P 的任何 left 指针或 right 指针操作都会发生错误，因为 e1P 为 null 根本没有 Node 类型节点的结构。再如，如果 e1 本身就是 e2 的左孩子，即 e1==e2L，那么让 e2L

成为 e1 的左孩子时，e1 的 left 指针将指向 e2L，将会指向自己，这会让整棵二叉树发生严重的结构错误。

换句话说，我们必须理清楚 e1 及其上下环境之间的关系、e2 及其上下环境之间的关系，以及两个环境之间是否有联系。有以下三个问题和一个特别注意是必须关注的。

问题一：e1 和 e2 是否有一个是头节点？如果有，谁是头？

问题二：e1 和 e2 是否相邻？如果相邻，谁是谁的父节点？

问题三：e1 和 e2 分别是各自父节点的左孩子还是右孩子？

特别注意：因为是在中序遍历时先找到 e1，后找到 e2，所以 e1 一定不是 e2 的右孩子，e2 也一定不是 e1 的左孩子。

以上三个问题与特别注意之间相互影响，情况非常复杂。经过仔细整理，情况共有 14 种，每一种情况在调整 e1 和 e2 各自的拓扑关系时都有特殊处理。

1．e1 是头，e1 是 e2 的父，此时 e2 只可能是 e1 的右孩子。

2．e1 是头，e1 不是 e2 的父，e2 是 e2P 的左孩子。

3．e1 是头，e1 不是 e2 的父，e2 是 e2P 的右孩子。

4．e2 是头，e2 是 e1 的父，此时 e1 只可能是 e2 的左孩子。

5．e2 是头，e2 不是 e1 的父，e1 是 e1P 的左孩子。

6．e2 是头，e2 不是 e1 的父，e1 是 e1P 的右孩子。

7．e1 和 e2 都不是头，e1 是 e2 的父，此时 e2 只可能是 e1 的右孩子，e1 是 e1P 的左孩子。

8．e1 和 e2 都不是头，e1 是 e2 的父，此时 e2 只可能是 e1 的右孩子，e1 是 e1P 的右孩子。

9．e1 和 e2 都不是头，e2 是 e1 的父，此时 e1 只可能是 e2 的左孩子，e2 是 e2P 的左孩子。

10．e1 和 e2 都不是头，e2 是 e1 的父，此时 e1 只可能是 e2 的左孩子，e2 是 e2P 的右孩子。

11．e1 和 e2 都不是头，谁也不是谁的父节点，e1 是 e1P 的左孩子，e2 是 e2P 的左孩子。

12．e1 和 e2 都不是头，谁也不是谁的父节点，e1 是 e1P 的左孩子，e2 是 e2P 的右孩子。

13．e1 和 e2 都不是头，谁也不是谁的父节点，e1 是 e1P 的右孩子，e2 是 e2P 的左孩子。

14．e1 和 e2 都不是头，谁也不是谁的父节点，e1 是 e1P 的右孩子，e2 是 e2P 的右孩子。

当情况 1 至情况 3 发生时，二叉树新的头节点应该为 e2，当情况 4 至情况 6 发生时，二叉树新的头节点应该为 e1，其他情况发生时，二叉树的头节点不用发生变化。

从结构上调整两个错误节点的全部过程请参看如下代码中的 recoverTree 方法。

```
public Node recoverTree(Node head) {
        Node[] errs = getTwoErrNodes(head);
        Node[] parents = getTwoErrParents(head, errs[0], errs[1]);
        Node e1 = errs[0];
        Node e1P = parents[0];
        Node e1L = e1.left;
        Node e1R = e1.right;
        Node e2 = errs[1];
        Node e2P = parents[1];
        Node e2L = e2.left;
        Node e2R = e2.right;
        if (e1 == head) {
                if (e1 == e2P) { // 情况 1
                        e1.left = e2L;
                        e1.right = e2R;
                        e2.right = e1;
                        e2.left = e1L;
                } else if (e2P.left == e2) { // 情况 2
                        e2P.left = e1;
                        e2.left = e1L;
                        e2.right = e1R;
                        e1.left = e2L;
                        e1.right = e2R;
                } else { // 情况 3
                        e2P.right = e1;
                        e2.left = e1L;
                        e2.right = e1R;
                        e1.left = e2L;
                        e1.right = e2R;
                }
                head = e2;
        } else if (e2 == head) {
                if (e2 == e1P) { // 情况 4
                        e2.left = e1L;
                        e2.right = e1R;
                        e1.left = e2;
                        e1.right = e2R;
                } else if (e1P.left == e1) { // 情况 5
                        e1P.left = e2;
                        e1.left = e2L;
                        e1.right = e2R;
                        e2.left = e1L;
                        e2.right = e1R;
                } else { // 情况 6
                        e1P.right = e2;
                        e1.left = e2L;
                        e1.right = e2R;
                        e2.left = e1L;
                        e2.right = e1R;
                }
                head = e1;
```

```
} else {
        if (e1 == e2P) {
                if (e1P.left == e1) { // 情况 7
                        e1P.left = e2;
                        e1.left = e2L;
                        e1.right = e2R;
                        e2.left = e1L;
                        e2.right = e1;
                } else { // 情况 8
                        e1P.right = e2;
                        e1.left = e2L;
                        e1.right = e2R;
                        e2.left = e1L;
                        e2.right = e1;
                }
        } else if (e2 == e1P) {
                if (e2P.left == e2) { // 情况 9
                        e2P.left = e1;
                        e2.left = e1L;
                        e2.right = e1R;
                        e1.left = e2;
                        e1.right = e2R;
                } else { // 情况 10
                        e2P.right = e1;
                        e2.left = e1L;
                        e2.right = e1R;
                        e1.left = e2;
                        e1.right = e2R;
                }
        } else {
                if (e1P.left == e1) {
                        if (e2P.left == e2) { // 情况 11
                                e1.left = e2L;
                                e1.right = e2R;
                                e2.left = e1L;
                                e2.right = e1R;
                                e1P.left = e2;
                                e2P.left = e1;
                        } else { // 情况 12
                                e1.left = e2L;
                                e1.right = e2R;
                                e2.left = e1L;
                                e2.right = e1R;
                                e1P.left = e2;
                                e2P.right = e1;
                        }
                } else {
                        if (e2P.left == e2) { // 情况 13
                                e1.left = e2L;
                                e1.right = e2R;
                                e2.left = e1L;
                                e2.right = e1R;
```

```
                                e1P.right = e2;
                                e2P.left = e1;
                            } else { // 情况 14
                                e1.left = e2L;
                                e1.right = e2R;
                                e2.left = e1L;
                                e2.right = e1R;
                                e1P.right = e2;
                                e2P.right = e1;
                            }
                        }
                    }
                }
                return head;
            }
```

判断 t1 树是否包含 t2 树全部的拓扑结构

【题目】

给定彼此独立的两棵树头节点分别为 t1 和 t2，判断 t1 树是否包含 t2 树全部的拓扑结构。例如，图 3-34 所示的 t1 树和图 3-35 所示的 t2 树。

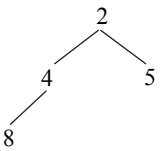

图 3-34

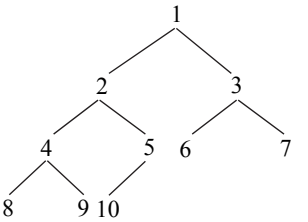

图 3-35

t1 树包含 t2 树全部的拓扑结构，所以返回 true。

【难度】

士 ★☆☆☆

【解答】

如果 t1 中某棵子树头节点的值与 t2 头节点的值一样，则从这两个头节点开始匹配，匹配的每一步都让 t1 上的节点跟着 t2 的先序遍历移动，每移动一步，都检查 t1 的当前节点

是否与 t2 当前节点的值一样。比如，题目中的例子，t1 中的节点 2 与 t2 中的节点 2 匹配，然后 t1 跟着 t2 向左，发现 t1 中的节点 4 与 t2 中的节点 4 匹配，t1 跟着 t2 继续向左，发现 t1 中的节点 8 与 t2 中的节点 8 匹配，此时 t2 回到 t2 中的节点 2，t1 也回到 t1 中的节点 2，然后 t1 跟着 t2 向右，发现 t1 中的节点 5 与 t2 中的节点 5 匹配。t2 匹配完毕，结果返回 true。如果匹配的过程中发现有不匹配的情况，直接返回 false，说明 t1 的当前子树从头节点开始，无法与 t2 匹配，那么再去寻找 t1 的下一棵子树。t1 的每棵子树上都有可能匹配出 t2，所以都要检查一遍。

所以，如果 t1 的节点数为 N，t2 的节点数为 M，该方法的时间复杂度为 $O(N \times M)$。

具体过程请参看如下代码中的 contains 方法，

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

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

public boolean contains(Node t1, Node t2) {
        return check(t1, t2) || contains(t1.left, t2) || contains(t1.right, t2);
}

public boolean check(Node h, Node t2) {
        if (t2 == null) {
                return true;
        }
        if (h == null || h.value != t2.value) {
                return false;
        }
        return check(h.left, t2.left) && check(h.right, t2.right);
}
```

判断 t1 树中是否有与 t2 树拓扑结构完全相同的子树

【题目】

给定彼此独立的两棵树头节点分别为 t1 和 t2，判断 t1 中是否有与 t2 树拓扑结构完全相同的子树。

例如，图 3-36 所示的 t1 树和图 3-37 所示 t2 树。

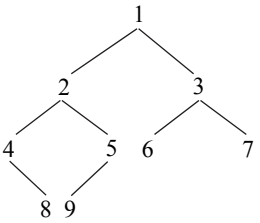

图 3-36

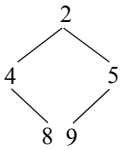

图 3-37

t1 树有与 t2 树拓扑结构完全相同的子树，所以返回 true。但如果 t1 树和 t2 树分别如图 3-38 和图 3-39 所示，则 t1 树就没有与 t2 树拓扑结构完全相同的子树，所以返回 false。

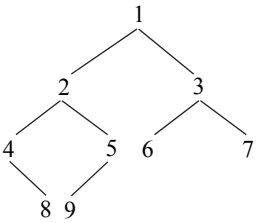

图 3-38

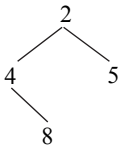

图 3-39

【难度】

校 ★★★☆

【解答】

如果 t1 的节点数为 *N*，t2 的节点数为 *M*，本题最优解是时间复杂度为 *O*(*N*+*M*)的方法。先简单介绍一个时间复杂度为 *O*(*N*×*M*)的方法，对于 t1 的每棵子树，都去判断是否与 t2 树的拓扑结构完全一样，这个过程的复杂度为 *O*(*M*)，t1 的子树一共有 *N* 棵，所以时间复杂度为 *O*(*N*×*M*)，这种方法本书不再详述。

下面重点介绍一下时间复杂度为 *O*(*N*+*M*)的方法，首先是把 t1 树和 t2 树按照先序遍历的方式序列化，关于这个内容，请阅读本书“二叉树的序列化和反序列化”问题。以题目的例子来说，t1 树序列化后的结果为“1!2!4!#!8!#!#!5!9!#!#!#!3!6!#!#!7!#!#!”，记为 t1Str。t2 树序列化后的结果为“2!4!#!8!#!#!5!9!#!#!#!”，记为 t2Str。接下来只要验证 t2Str 是否是 t1Str 的子串即可，这个用 KMP 算法可以在线性时间内解决。所以 t1 序列化的过程为 *O*(*N*)，t2 序列化的过程为 *O*(*M*)，KMP 解决 t1Str 和 t2Str 的匹配问题 *O*(*M*+*N*)，所以时间复杂度为

$O(M+N)$。有关 KMP 算法的内容，请读者阅读本书“KMP 算法”问题，关于这个算法非常清晰的解释，这里不再详述。

本题最优解的全部过程请参看如下代码中的 isSubtree 方法。

```
public boolean isSubtree(Node t1, Node t2) {
        String t1Str = serialByPre(t1);
        String t2Str = serialByPre(t2);
        return getIndexOf(t1Str, t2Str) != -1;
}

public String serialByPre(Node head) {
        if (head == null) {
                return "#!";
        }
        String res = head.value + "!";
        res += serialByPre(head.left);
        res += serialByPre(head.right);
        return res;
}

// KMP
public int getIndexOf(String s, String m) {
        if (s == null || m == null || m.length() < 1 || s.length() < m.length())
        {
                return -1;
        }
        char[] ss = s.toCharArray();
        char[] ms = m.toCharArray();
        int si = 0;
        int mi = 0;
        int[] next = getNextArray(ms);
        while (si < ss.length && mi < ms.length) {
                if (ss[si] == ms[mi]) {
                        si++;
                        mi++;
                } else if (next[mi] == -1) {
                        si++;
                } else {
                        mi = next[mi];
                }
        }
        return mi == ms.length ? si - mi : -1;
}

public int[] getNextArray(char[] ms) {
        if (ms.length == 1) {
                return new int[] { -1 };
        }
        int[] next = new int[ms.length];
        next[0] = -1;
```

```
        next[1] = 0;
        int pos = 2;
        int cn = 0;
        while (pos < next.length) {
                if (ms[pos - 1] == ms[cn]) {
                        next[pos++] = ++cn;
                } else if (cn > 0) {
                        cn = next[cn];
                } else {
                        next[pos++] = 0;
                }
        }
        return next;
}
```

判断二叉树是否为平衡二叉树

【题目】

平衡二叉树的性质为：要么是一棵空树，要么任何一个节点的左右子树高度差的绝对值不超过 1。给定一棵二叉树的头节点 head，判断这棵二叉树是否为平衡二叉树。

【要求】

如果二叉树的节点数为 N，要求时间复杂度为 $O(N)$。

【难度】

士 ★☆☆☆

【解答】

解法的整体过程为二叉树的后序遍历，对任何一个节点 node 来说，先遍历 node 的左子树，遍历过程中收集两个信息，node 的左子树是否为平衡二叉树，node 的左子树最深到哪一层记为 lH。如果发现 node 的左子树不是平衡二叉树，无须进行任何后续过程，此时返回什么已不重要，因为已经发现整棵树不是平衡二叉树，退出遍历过程；如果 node 的左子树是平衡二叉树，再遍历 node 的右子树，遍历过程中再收集两个信息，node 的右子树是否为平衡二叉树，node 的右子树最深到哪一层记为 rH。如果发现 node 的右子树不是平衡二叉树，无须进行任何后续过程，返回什么也不重要，因为已经发现整棵树不是平衡二叉树，退出遍历过程；如果 node 的右子树也是平衡二叉树，就看 lH 和 rH 差的绝对值是否大

于 1，如果大于 1，说明已经发现整棵树不是平衡二叉树，如果不大于 1，则返回 lH 和 rH 较大的一个。

判断的全部过程请参看如下代码中的 isBalance 方法。在递归函数 getHeight 中，一旦发现不符合平衡二叉树的性质，递归过程会迅速退出，此时返回什么根本不重要。boolean[] res 长度为 1，其功能相当于一个全局的 boolean 变量。

```
public boolean isBalance(Node head) {
        boolean[] res = new boolean[1];
        res[0] = true;
        getHeight(head, 1, res);
        return res[0];
}

public int getHeight(Node head, int level, boolean[] res) {
        if (head == null) {
                return level;
        }
        int lH = getHeight(head.left, level + 1, res);
        if (!res[0]) {
                return level;
        }
        int rH = getHeight(head.right, level + 1, res);
        if (!res[0]) {
                return level;
        }
        if (Math.abs(lH - rH) > 1) {
                res[0] = false;
        }
        return Math.max(lH, rH);
}
```

整个后序遍历的过程中，每个节点最多遍历一次，如果中途发现不满足平衡二叉树的性质，整个过程会迅速退出，没遍历到的节点也不用遍历了，所以时间复杂度为 $O(N)$。

根据后序数组重建搜索二叉树

【题目】

给定一个整型数组 arr，已知其中没有重复值，判断 arr 是否可能是节点值类型为整型的搜索二叉树后序遍历的结果。

进阶：如果整型数组 arr 中没有重复值，且已知是一棵搜索二叉树的后序遍历结果，通过数组 arr 重构二叉树。

【难度】

士 ★☆☆☆

【解答】

原问题的解法。二叉树的后序遍历为先左、再右、最后根的顺序，所以，如果一个数组是二叉树后序遍历的结果，那么头节点的值一定会是数组的最后一个元素。搜索二叉树的性质，所以比后序数组最后一个元素值小的数组会在数组的左边，比数组最后一个元素值大的数组会在数组的右边。比如 arr=[2,1,3,6,5,7,4]，比 4 小的部分为[2,1,3]，比 4 大的部分为[6,5,7]。如果不满足这种情况，说明这个数组一定不可能是搜索二叉树后序遍历的结果。接下来数组划分成左边数组和右边数组，相当于二叉树分出了左子树和右子树，只要递归地进行如上判断即可。

具体过程请参看如下代码中的 isPostArray 方法。

```
public boolean isPostArray(int[] arr) {
        if (arr == null || arr.length == 0) {
                return false;
        }
        return isPost(arr, 0, arr.length - 1);
}

public boolean isPost(int[] arr, int start, int end) {
        if (start == end) {
                return true;
        }
        int less = -1;
        int more = end;
        for (int i = start; i < end; i++) {
                if (arr[end] > arr[i]) {
                        less = i;
                } else {
                        more = more == end ? i : more;
                }
        }
        if (less == -1 || more == end) {
                return isPost(arr, start, end - 1);
        }
        if (less != more - 1) {
                return false;
        }
        return isPost(arr, start, less) && isPost(arr, more, end - 1);
}
```

进阶问题的分析与原问题同理，一棵树的后序数组中最后一个值为二叉树头节点的值，

数组左部分都比头节点的值小，用来生成头节点的左子树，剩下的部分用来生成右子树。

具体过程请参看如下代码中的 posArrayToBST 方法。

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

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

public Node posArrayToBST(int[] posArr) {
        if (posArr == null) {
                return null;
        }
        return posToBST(posArr, 0, posArr.length - 1);
}

public Node posToBST(int[] posArr, int start, int end) {
        if (start > end) {
                return null;
        }
        Node head = new Node(posArr[end]);
        int less = -1;
        int more = end;
        for (int i = start; i < end; i++) {
                if (posArr[end] > posArr[i]) {
                        less = i;
                } else {
                        more = more == end ? i : more;
                }
        }
        head.left = posToBST(posArr, start, less);
        head.right = posToBST(posArr, more, end - 1);
        return head;
}
```

判断一棵二叉树是否为搜索二叉树和完全二叉树

【题目】

给定一个二叉树的头节点 head，已知其中没有重复值的节点，实现两个函数分别判断这棵二叉树是否是搜索二叉树和完全二叉树。

【难度】

士 ★☆☆☆

【解答】

判断一棵二叉树是否是搜索二叉树，只要改写一个二叉树中序遍历，在遍历的过程中看节点值是否都是递增的即可。本书改写的是 Morris 中序遍历，所以时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。有关 Morris 中序遍历的介绍，请读者阅读本书“遍历二叉树的神级方法”问题。需要注意的是，Morris 遍历分调整二叉树结构和恢复二叉树结构两个阶段，所以，当发现节点值降序时，不能直接返回 false，这么做可能会跳过恢复阶段，从而破坏二叉树的结构。

通过改写 Morris 中序遍历来判断搜索二叉树的过程请参看如下代码中的 isBST 方法。

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

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

public boolean isBST(Node head) {
        if (head == null) {
                return true;
        }
        boolean res = true;
        Node pre = null;
        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;
                        }
                }
```

```
                if (pre != null && pre.value > cur1.value) {
                        res = false;
                }
                pre = cur1;
                cur1 = cur1.right;
        }
        return res;
}
```

判断一棵二叉树是否是完全二叉树，依据以下标准会使判断过程变得简单且易实现：

1．按层遍历二叉树，从每层的左边向右边依次遍历所有的节点。

2．如果当前节点有右孩子，但没有左孩子，直接返回 false。

3. 如果当前节点并不是左右孩子全有，那之后的节点必须都为叶节点，否则返回 false。

4．遍历过程中如果不返回 false，遍历结束后返回 true。

判断是否是完全二叉树的全部过程请参看如下代码中的 isCBT 方法。

```
public boolean isCBT(Node head) {
        if (head == null) {
                return true;
        }
        Queue<Node> queue = new LinkedList<Node>();
        boolean leaf = false;
        Node l = null;
        Node r = null;
        queue.offer(head);
        while (!queue.isEmpty()) {
                head = queue.poll();
                l = head.left;
                r = head.right;
                if ((leaf&&(l!=null||r!=null)) || (l==null&&r!=null)) {
                        return false;
                }
                if (l != null) {
                        queue.offer(l);
                }
                if (r != null) {
                        queue.offer(r);
                } else {
                        leaf = true;
                }
        }
        return true;
}
```

通过有序数组生成平衡搜索二叉树

【题目】

给定一个有序数组 sortArr，已知其中没有重复值，用这个有序数组生成一棵平衡搜索二叉树，并且该搜索二叉树中序遍历的结果与 sortArr 一致。

【难度】

士 ★☆☆☆

【解答】

本题的递归过程比较简单，用有序数组中最中间的数生成搜索二叉树的头节点，然后用这个数左边的数生成左子树，用右边的数生成右子树即可。

全部过程请参看如下代码中的 generateTree 方法。

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

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

public Node generateTree(int[] sortArr) {
        if (sortArr == null) {
                return null;
        }
        return generate(sortArr, 0, sortArr.length - 1);
}

public Node generate(int[] sortArr, int start, int end) {
        if (start > end) {
                return null;
        }
        int mid = (start + end) / 2;
        Node head = new Node(sortArr[mid]);
        head.left = generate(sortArr, start, mid - 1);
        head.right = generate(sortArr, mid + 1, end);
        return head;
}
```

在二叉树中找到一个节点的后继节点

【题目】

现在有一种新的二叉树节点类型如下：

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

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

该结构比普通二叉树节点结构多了一个指向父节点的 parent 指针。假设有一棵 Node 类型的节点组成的二叉树，树中每个节点的 parent 指针都正确地指向自己的父节点，头节点的 parent 指向 null。只给一个在二叉树中的某个节点 node，请实现返回 node 的后继节点的函数。在二叉树的中序遍历的序列中，node 的下一个节点叫作 node 的后继节点。

例如，图 3-40 所示的二叉树。

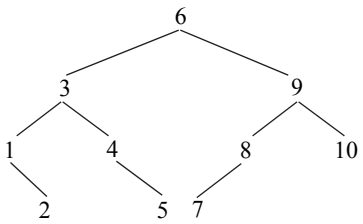

图 3-40

中序遍历的结果为：1，2，3，4，5，6，7，8，9，10

所以节点 1 的后继为节点 2，节点 2 的后继为节点 3，……，节点 10 的后继为 null。

【难度】

尉 ★★☆☆

【解答】

先简单介绍一种时间复杂度和空间复杂度较高但易于理解的方法。既然新类型的二叉树节点有指向父节点的指针，那么一直往上移动，自然可以找到头节点。找到头节点之后，再进行二叉树的中序遍历，生成中序遍历序列，然后在这个序列中找到 node 节点的下一个节点返回即可。如果二叉树的节点数为 N，这种方法要把二叉树的所有节点至少遍历一遍，生成中序遍历的序列还需要大小为 N 的空间，所以该方法的时间复杂度与额外空间复杂度都为 $O(N)$。本书不再详述。

最优解法不必遍历所有的节点，如果 node 节点和 node 后继节点之间的实际距离为 L，最优解法只用走过 L 个节点，时间复杂度为 $O(L)$，额外空间复杂度为 $O(1)$。接下来详细说明最优解法是如何找到 node 的后继节点的。

情况 1：如果 node 有右子树，那么后继节点就是右子树上最左边的节点。

例如，题目所示的二叉树中，当 node 为节点 1、3、4、6 或 9 时，就是这种情况。

情况 2：如果 node 没有右子树，那么先看 node 是不是 node 父节点的左孩子，如果是左孩子，那么此时 node 的父节点就是 node 的后继节点；如果是右孩子，就向上寻找 node 的后继节点，假设向上移动到的节点记为 s，s 的父节点记为 p，如果发现 s 是 p 的左孩子，那么节点 p 就是 node 节点的后继节点，否则就一直向上移动。

例如，题目所示的二叉树中，当 node 为节点 7 时，节点 7 的父节点是节点 8，同时节点 7 是节点 8 的左孩子，此时节点 8 就是节点 7 的后继节点。

再如，题目所示的二叉树中，当 node 为节点 5 时，节点 5 的父节点是节点 4，但是节点 5 是节点 4 的右孩子，所以向上寻找 node 的后继节点。当向上移动到节点 4，节点 4 的父节点是节点 3，但是节点 4 还是节点 3 的右孩子，继续向上移动。当向上移动到节点 3 时，节点 3 的父节点是节点 6，此时终于发现节点 3 是节点 6 的左孩子，移动停止，节点 6 就是 node(节点 5)的后继节点。

情况 3：如果在情况 2 中一直向上寻找，都移动到空节点时还是没有发现 node 的后继节点，说明 node 根本不存在后继节点。

比如，题目所示的二叉树中，当 node 为节点 10 时，一直向上移动到节点 6，此时发现节点 6 的父节点已经为空，说明 node 没有后继节点。

情况 1 和情况 2 遍历的节点就是 node 到 node 后继节点这条路径上的节点；情况 3 遍历的节点数也不会超过二叉树的高度。

最优解的具体过程请参看如下代码中的 getNextNode 方法。

```
public Node getNextNode(Node node) {
        if (node == null) {
                return node;
        }
        if (node.right != null) {
                return getLeftMost(node.right);
        } else {
                Node parent = node.parent;
                while (parent != null && parent.left != node) {
                        node = parent;
                        parent = node.parent;
                }
                return parent;
        }
}

public Node getLeftMost(Node node) {
        if (node == null) {
                return node;
        }
        while (node.left != null) {
                node = node.left;
        }
        return node;
}
```

在二叉树中找到两个节点的最近公共祖先

【题目】

给定一棵二叉树的头节点 head，以及这棵树中的两个节点 o1 和 o2，请返回 o1 和 o2 的最近公共祖先节点。

例如，图 3-41 所示的二叉树。

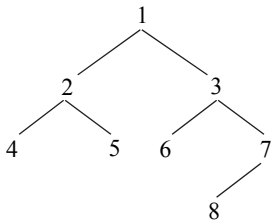

图 3-41

节点 4 和节点 5 的最近公共祖先节点为节点 2，节点 5 和节点 2 的最近公共祖先节点

为节点 2，节点 6 和节点 8 的最近公共祖先节点为节点 3，节点 5 和节点 8 的最近公共祖先节点为节点 1。

进阶：如果查询两个节点的最近公共祖先的操作十分频繁，想法让单条查询的查询时间减少。

再进阶：给定二叉树的头节点 head，同时给定所有想要进行的查询。二叉树的节点数量为 N，查询条数为 M，请在时间复杂度为 $O(N+M)$内返回所有查询的结果。

【难度】

原问题：士 ★☆☆☆

进阶问题：尉 ★★☆☆

再进阶问题：校 ★★★☆

【解答】

先来解决原问题。后序遍历二叉树，假设遍历到的当前节点为 cur。因为是后序遍历，所以先处理 cur 的两棵子树。假设处理 cur 左子树时返回节点为 left，处理右子树时返回 right。

1．如果发现 cur 等于 null，或者 o1、o2，则返回 cur。

2．如果 left 和 right 都为空，说明 cur 整棵子树上没有发现过 o1 或 o2，返回 null。

3．如果 left 和 right 都不为空，说明左子树上发现过 o1 或 o2，右子树上也发现过 o2 或 o1，说明 o1 向上与 o2 向上的过程中，首次在 cur 相遇，返回 cur。

4．如果 left 和 right 有一个为空，另一个不为空，假设不为空的那个记为 node，此时 node 到底是什么？有两种可能，要么 node 是 o1 或 o2 中的一个，要么 node 已经是 o1 和 o2 的最近公共祖先。不管是哪种情况，直接返回 node 即可。

以题目二叉树的例子来说明一下，假设 o1 为节点 6，o2 为节点 8，过程为后序遍历。

- 依次遍历节点 4、节点 5、节点 2，都没有发现 o1 或 o2，所以节点 1 的左子树返回为 null；
- 遍历节点 6，发现节点 6 等于 o1，返回节点 6，所以节点 3 左子树的返回值为节点 6；
- 遍历节点 8，发现节点 8 等于 o2，返回节点 8，所以节点 7 左子树的返回值为节点 8；
- 节点 7 的右子树为 null，所以节点 7 右子树的返回值为 null；
- 遍历节点 7，左子树返回节点 8，右子树返回 null，根据步骤 4，此时返回节点 8，

所以节点 3 的右子树的返回值为节点 8；

- 遍历节点 3，左子树返回节点 6，右子树返回节点 8，根据步骤 3，此时返回节点 3，所以节点 1 的右子树的返回值为节点 3；
- 遍历节点 1，左子树返回 null，右子树返回节点 3，根据步骤 4，最终返回节点 3。

找到两个节点最近公共祖先的详细过程请参看如下代码中的 lowestAncestor 方法。

```
public Node lowestAncestor(Node head, Node o1, Node o2) {
        if (head == null || head == o1 || head == o2) {
                return head;
        }
        Node left = lowestAncestor(head.left, o1, o2);
        Node right = lowestAncestor(head.right, o1, o2);
        if (left != null && right != null) {
                return head;
        }
        return left != null ? left : right;
}
```

进阶问题其实是先花较大的力气建议一种记录，以后执行每次查询时就可以完全根据记录进行查询。记录的方式可以有很多种，本书提供两种记录结构供读者参考，两种记录各有优缺点。

结构一：建立二叉树中每个节点对应的父节点信息，是一张哈希表。

如果对题目中的二叉树建立这种哈希表，哈希表中的信息如下：

key	value
节点 1	null
节点 2	节点 1
节点 3	节点 1
节点 4	节点 2
节点 5	节点 2
节点 6	节点 3
节点 7	节点 3
节点 8	节点 7

key 代表二叉树中的一个节点，value 代表其对应的父节点。只用遍历一次二叉树，这张表就可以创建好，以后每次查询都可以根据这张哈希表进行。

假设想查节点 4 和节点 8 的最近公共祖先，方法是使用如上的哈希表，把包括节点 4 在内的所有节点 4 的祖先节点放进另一个哈希表 A 中，A 表示节点 4 到头节点这条路径上

所有节点的集合。所以 A={节点 4，节点 2，节点 1}。然后使用如上的哈希表，从节点 8 开始往上逐渐移动到头节点。首先是节点 8，发现不在 A 中，然后是节点 7，发现也不在 A 中，接下来是节点 3，依然不在 A 中，最后是节点 1，发现在 A 中，那么节点 1 就是节点 4 和节点 8 的最近公共祖先。只要在移动过程中发现某个节点在 A 中，这个节点就是要求的公共祖先节点。

结构一的具体实现请参看如下代码中 Record1 类的实现，构造函数是创建记录过程，方法 query 是查询操作。

```
public class Record1 {
        private HashMap<Node, Node> map;

        public Record1(Node head) {
                map = new HashMap<Node, Node>();
                if (head != null) {
                        map.put(head, null);
                }
                setMap(head);
        }

        private void setMap(Node head) {
                if (head == null) {
                        return;
                }
                if (head.left != null) {
                        map.put(head.left, head);
                }
                if (head.right != null) {
                        map.put(head.right, head);
                }
                setMap(head.left);
                setMap(head.right);
        }

        public Node query(Node o1, Node o2) {
                HashSet<Node> path = new HashSet<Node>();
                while (map.containsKey(o1)) {
                        path.add(o1);
                        o1 = map.get(o1);
                }
                while (!path.contains(o2)) {
                        o2 = map.get(o2);
                }
                return o2;
        }

}
```

很明显，结构一建立记录的过程时间复杂度为 $O(N)$、额外空间复杂度为 $O(N)$。查询操作时，时间复杂度为 $O(h)$，其中，h 为二叉树的高度。

结构二：直接建立任意两个节点之间的最近公共祖先记录，便于以后查询时直接查。

建立记录的具体过程如下：

1．对二叉树中的每棵子树（一共 N 棵）都进行步骤 2。

2．假设子树的头节点为 h，h 所有的后代节点和 h 节点的最近公共祖先都是 h，记录下来。h 左子树的每个节点和 h 右子树的每个节点的最近公共祖先都是 h，记录下来。

为了保证记录不重复，设计一种好的实现方式是这种结构实现的重点。

结构二的具体实现请参看如下代码中 Record2 类的实现。

```
public class Record2 {
        private HashMap<Node, HashMap<Node, Node>> map;

        public Record2(Node head) {
                map = new HashMap<Node, HashMap<Node, Node>>();
                initMap(head);
                setMap(head);
        }

        private void initMap(Node head) {
                if (head == null) {
                        return;
                }
                map.put(head, new HashMap<Node, Node>());
                initMap(head.left);
                initMap(head.right);
        }

        private void setMap(Node head) {
                if (head == null) {
                        return;
                }
                headRecord(head.left, head);
                headRecord(head.right, head);
                subRecord(head);
                setMap(head.left);
                setMap(head.right);
        }

        private void headRecord(Node n, Node h) {
                if (n == null) {
                        return;
                }
                map.get(n).put(h, h);
                headRecord(n.left, h);
                headRecord(n.right, h);
```

```
        }

        private void subRecord(Node head) {
                if (head == null) {
                        return;
                }
                preLeft(head.left, head.right, head);
                subRecord(head.left);
                subRecord(head.right);
        }

        private void preLeft(Node l, Node r, Node h) {
                if (l == null) {
                        return;
                }
                preRight(l, r, h);
                preLeft(l.left, r, h);
                preLeft(l.right, r, h);
        }

        private void preRight(Node l, Node r, Node h) {
                if (r == null) {
                        return;
                }
                map.get(l).put(r, h);
                preRight(l, r.left, h);
                preRight(l, r.right, h);
        }

        public Node query(Node o1, Node o2) {
                if (o1 == o2) {
                        return o1;
                }
                if (map.containsKey(o1)) {
                        return map.get(o1).get(o2);
                }
                if (map.containsKey(o2)) {
                        return map.get(o2).get(o1);
                }
                return null;
        }

}
```

如果二叉树的节点数为N，想要记录每两个节点之间的信息，信息的条数为$((N-1)\times N)/2$。所以建立结构二的过程的额外空间复杂度为$O(N^2)$，时间复杂度为$O(N^2)$，单次查询的时间复杂度为$O(1)$。

再进阶的问题：请参看下一题“Tarjan 算法与并查集解决二叉树节点间最近公共祖先的批量查询问题”。

Tarjan 算法与并查集解决二叉树节点间最近公共祖先的批量查询问题

【题目】

如下的 Node 类是标准的二叉树节点结构：

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

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

再定义 Query 类如下：

```
public class Query {
        public Node o1;
        public Node o2;

        public Query(Node o1, Node o2) {
                this.o1 = o1;
                this.o2 = o2;
        }
}
```

一个 Query 类的实例表示一条查询语句，表示想要查询 o1 节点和 o2 节点的最近公共祖先节点。

给定一棵二叉树的头节点 head，并给定所有的查询语句，即一个 Query 类型的数组 Query[] ques，请返回 Node 类型的数组 Node[] ans，ans[i]代表 ques[i]这条查询的答案，即 ques[i].o1 和 ques[i].o2 的最近公共祖先。

【要求】

如果二叉树的节点数为 N，查询语句的条数为 M，整个处理过程的时间复杂度要求达到 $O(N+M)$。

【难度】

校 ★★★☆

【解答】

本题的解法利用了 Tarjan 算法与并查集结构的结合。二叉树如图 3-42 所示，假设想要进行的查询为 ques[0]=(节点 4 和节点 7)，ques[1]=(节点 7 和节点 8)，ques[2]=(节点 8 和节点 9)，ques[3]=(节点 9 和节点 3)，ques[4]=(节点 6 和节点 6)，ques[5]=(null 和节点 5)，ques[6]=(null 和 null)。

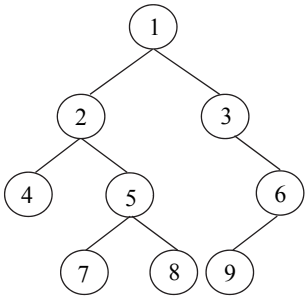

图 3-42

首先生成和 ques 长度一样的 ans 数组，如下三种情况的查询是可以直接得到答案的：

1．如果 o1 等于 o2，答案为 o1。例如，ques[4]，令 ans[4]=节点 6。

2．如果 o1 和 o2 只有一个为 null，答案是不为空的那个。例如，ques[5]，令 ans[5]=节点 5。

3．如果 o1 和 o2 都为 null，答案为 null。例如 ques[6]，令 ans[6]=null。

对不能直接得到答案的查询，我们把查询的格式转换一下，具体过程如下：

1．生成两张哈希表 queryMap 和 indexMap。queryMap 类似于邻接表，key 表示查询涉及的某个节点，value 是一个链表类型，表示 key 与那些节点之间有查询任务。indexMap 的 key 也表示查询涉及的某个节点，value 也是链表类型，表示如果依次解决有关 key 节点的每个问题，该把答案放在 ans 的什么位置。也就是说，如果一个节点为 node，node 与哪些节点之间有查询任务呢？都放在 queryMap 中；获得的答案该放在 ans 的什么位置呢？都放在 indexMap 中。

比如，根据 ques[0～3]，queryMap 和 indexMap 生成记录如下：

Key	Value
节点 4	queryMap 中节点 4 的链表：{节点 7} indexMap 中节点 4 的链表：{　0　}
节点 7	queryMap 中节点 7 的链表：{节点 4，节点 8} indexMap 中节点 7 的链表：{　0　，　1　}
节点 8	queryMap 中节点 8 的链表：{节点 7，节点 9} indexMap 中节点 8 的链表：{　1　，　2　}
节点 9	queryMap 中节点 9 的链表：{节点 8，节点 3} indexMap 中节点 9 的链表：{　2　，　3　}
节点 3	queryMap 中节点 3 的链表：{节点 9} indexMap 中节点 9 的链表：{　3　}

读者应该会发现一条(o1,o2)的查询语句在上面的两个表中其实生成了两次。这么做的目的是为了处理时方便找到关于每个节点的查询任务，也方便设置答案，介绍完整个流程之后，会有进一步说明。

接下来是 Tarjan 算法处理 *M* 条查询的过程，整个过程是二叉树的先左、再根、再右、最后再回到根的遍历。以图 3-42 的二叉树来说明。

1）对每个节点生成各自的集合，{1}，{2}，…，{9}，开始时每个集合的祖先节点设为空。

2）遍历节点 4，发现它属于集合{4}，设置集合{4}的祖先为节点 4，发现有关于节点 4 和节点 7 的查询任务，发现节点 7 属于集合{7}，但集合{7}的祖先节点为空，说明还没遍历到，所以暂时不执行这个查询任务。

2．遍历节点 2，发现它属于集合{2}，设置集合{2}的祖先为节点 2，此时左孩子节点 4 属于集合{4}，将集合{4}与集合{2}合并，两个集合一旦合并，小的不再存在，而是生成更大的集合{4,2}，并设置集合{4,2}的祖先为当前节点 2。

3．遍历节点 7，发现它属于集合{7}，设置集合{7}的祖先为节点 7，发现有关节点 7 和节点 4 的查询任务，发现节点 4 属于集合{4,2}，集合{4,2}的祖先节点为节点 2，说明节点 4 和节点 7 都已经遍历到，根据 indexMap 知道答案应放在 0 位置，所以设置 ans[0]=节点 2；又发现有节点 7 和节点 8 的查询任务，发现节点 8 属于集合{8}，但集合{8}的祖先节点为空，说明还没遍历到，忽略。

4．遍历节点 5，发现它属于集合{5}，设置集合{5}的祖先为节点 5，此时左孩子节点 7 属于集合{7}，两集合合并为{7,5}，并设置集合{7,5}的祖先为当前节点 5。

5．遍历节点 8，发现它属于集合{8}，设置集合{8}的祖先为节点 8，发现有节点 8 和节点 7 的查询任务，发现节点 7 属于集合{7,5}，集合{7,5}的祖先节点为节点 5，设置 ans[1]=节点 5；发现有节点 8 和节点 9 的查询任务，忽略。

6．从节点 5 的右子树重新回到节点 5，节点 5 属于{7,5}，节点 5 的右孩子节点 8 属于{8}，两个集合合并为{7,5,8}，并设置{7,5,8}的祖先节点为当前的节点 5。

7．从节点 2 的右子树重新回到节点 2，节点 2 属于集合{2,4}，节点 2 的右孩子节点 5 属于集合{7,5,8}，合并为{2,4,7,5,8}，并设置这个集合的祖先节点为当前的节点 2。

8．遍历节点 1，{2,4,7,5,8}与{1}合并为{2,4,7,5,8,1}，这个集合祖先节点为当前的节点 1；

9．遍历节点 3，发现属于集合{3}，集合{3}祖先节点设为节点 3，发现有节点 3 和节点 9 的查询任务，但节点 9 没遍历到，忽略。

10．遍历节点 6，发现属于集合{6}，集合{6}祖先节点设为节点 6。

11．遍历节点 9，发现属于集合{9}，集合{9}祖先节点设为节点 9；发现有节点 9 和节点 8 的查询任务，节点 8 属于{2,4,7,5,8,1}，这个集合的祖先节点为节点 1，根据 indexMap 知道答案应放在 2 位置，所以设置 ans[2]=节点 1；发现有节点 9 和节点 3 的查询任务，节点 3 属于{3}，这个集合的祖先节点为节点 3，根据 indexMap，答案应放在 3 位置，所以设置 ans[3]=节点 1。

12．回到节点 6，合并{6}和{9}为{6,9}，{6,9}的祖先节点设为节点 6。

13．回到节点 3，合并{3}和{6,9}为{3,6,9}，{3,6,9}的祖先节点设为节点 3。

14．回到节点 1，合并{2,4,7,5,8,1}和{3,6,9}为{1,2,3,4,5,6,7,8,9}，祖先节点设为节点 1。

15．过程结束，所有的答案都已得到。

现在我们可以解释生成 queryMap 和 indexMap 的意义了，遍历到一个节点时记为 a，queryMap 可以让我们迅速查到有哪些节点和 a 之间有查询任务，如果能够得到答案，indexMap 还能告诉我们把答案放在 ans 的什么位置。假设 a 和节点 b 之间有查询任务，如果此时 b 已经遍历过，自然可以取得答案，然后在有关 a 的链表中，删除这个查询任务；如果此时 b 没有遍历过，依然在属于 a 的链表中删除这个查询任务，这个任务会在遍历到 b 的时候重新被发现，因为同样的任务 b 也存了一份。所以遍历到一个节点，有关这个节点的任务列表会被完全清空，可能有些任务已被解决，有些则没有也不要紧，一定会在后序的过程中被发现并得以解决。这就是 queryMap 和 indexMap 生成两遍查询任务信息的意义。

上述流程很好理解，但大量出现生成集合、合并集合和根据节点找到所在集合的操作，如果二叉树的节点数为 N，那么生成集合操作 $O(N)$次，合并集合操作 $O(N)$次，根据节点找

到所在集合 $O(N+M)$次。所以，如果上述整个过程想达到 $O(N+M)$的时间复杂度，那就要求有关集合的单次操作，平均时间复杂度要求为 $O(1)$，请注意这里说的是平均。存在这么好的集合结构吗？存在。这种集合结构就是接下来要介绍的并查集结构。

并查集结构由 Bernard A. Galler 和 Michael J. Fischer 在 1964 年发明，但证明时间复杂度的工作却持续了数年之久，直到 1989 才彻底证明完毕。有兴趣的读者请阅读《算法导论》一书来了解整个证明过程，本书由于篇幅所限，不再详述证明过程，这里只重点介绍并查集的结构和各种操作的细节，并实现针对二叉树结构的并查集，这是一种经常使用的高级数据结构。

请读者注意，上述流程中提到一个集合祖先节点的概念与接下来介绍并查集时提到的一个集合代表节点（父节点）的概念不是一回事。本题的流程中有关设置一个集合祖先节点的操作也不属于并查集自身的操作，关于这个操作，我们在介绍完并查集结构之后再详细说明。

并查集由一群集合构成，比如步骤 1 中对每个节点都生成各自的集合，所有集合的全体构成一个并查集={ {1}, {2}, …, {9} }。这些集合可以合并，如果最终合并成一个大集合(步骤 14)，那么此时并查集中有一个元素，这个元素是这个大集合，即并查集={ {1,2，…，9} }。其实主要是想说明并查集是集合的集合这个概念。

并查集先经历初始化的过程，就向流程中的步骤 1 一样，把每个节点都生成一个只含有自己的集合。那么并查集中的单个集合是什么结构呢？如果集合中只有一个元素，记为节点 a 时，如图 3-43 所示。

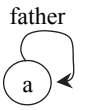

图 3-43

当集合中只有一个元素时，这个元素的 father 为自己，也就意味着这个集合的代表节点就是唯一的元素。实现记录节点 father 信息的方式有很多，本书使用哈希表来保存所有并查集中所有集合的所有元素的 father 信息，记为 fatherMap。比如，对于这个集合，在 fatherMap 中肯定有某一条记录为(节点 a(key)，节点 a(value))，表示 key 节点的 father 为 value 节点。每个元素除了 father 信息，还有另一个信息叫 rank，rank 为整数代表一个节点的秩，秩的概念可以粗略地理解为一个节点下面还有多少层节点，但是并查集结构对每个节点秩的更新并不严格，所以每个节点的秩只能粗略描述该节点下面的深度，正是由于秩在更新

上的不严格，换来了极好的时间复杂度，而也正是因为这种不严格增加了并查集时间复杂度证明的难度。集合中只有一个元素时，这个元素的 rank 初始化为 0。所有节点的秩信息保存在 rankMap 中。

对二叉树结构并查集初始化的具体过程请参看如下 DisjointSets 类中的 makeSets 方法。

当集合有多个节点时，下层节点的 father 为上层节点，最上层的节点 father 指向自己，最上层的节点又叫集合的代表节点，如图 3-44 所示。

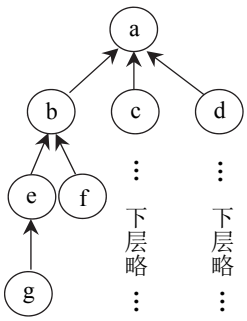

图 3-44

在并查集中，若要查一个节点属于哪个集合，就是在查这个节点所在集合的代表节点是什么，一个节点通过 father 信息逐渐找到最上面的节点，这个节点的 father 是自己，代表整个集合。比如图 3-44 中，任何一个节点最终都找到节点 a，比如节点 g。如果另外一个节点假设为 z，找到的代表节点不是节点 a，那么可以肯定节点 g 和节点 z 不在一个集合中。通过一个节点找到所在集合代表节点的过程叫作 findFather 过程。findFather 最终会返回代表节点，但过程并不仅是单纯的查找过程，还会把整个查找路径压缩。比如，执行 findFather(g)，通过 father 逐渐向上，找到最上层节点 a 之后，会把从 a 到 g 这条路径上所有节点的 father 都设置为 a，则集合变成图 3-45 的样子。

经过路径压缩之后，路径上每个节点下次在找代表节点的时候都只需经过一次移动的过程。这也是整个并查集结构的设计中最重要的优化。

根据一个节点查找所在集合代表节点的过程请参看如下 DisjointSets 类中的 findFather 方法。

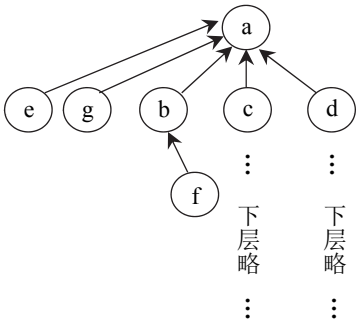

图 3-45

前面已经展示了并查集中的集合如何初始化，如何根据某一个节点查找所在集合的代表元素以及如何做路径压缩的过程，接下来介绍集合如何合并。首先，两个集合进行合并操作时，参数并不是两个集合，而是并查集中任意的两个节点，记为 a 和 b。所以集合的合并更准确的说法是，根据 a 找到 a 所在集合的代表节点是 findFather(a)，记为 aF，根据 b 找到 b 所在集合的代表节点是 findFather(b)，记为 bF，然后用如下策略决定由哪个代表节点作为合并后大集合的代表节点。

1．如果 aF==bF，说明 a 和 b 本身就在一个集合里，不用合并。

2．如果 aF!=bF，那么假设 aF 的 rank 值记为 aFrank，bF 的 rank 值记为 bFrank。根据对 rank 的解释，rank 可以粗描一个节点下面的层数，而 aF 和 bF 本身又是各自集合中最上面的节点，所以 aFrank 粗描 a 所在集合的总层数，bFrank 粗描 b 所在集合的总层数。如果 aFrank<bFrank，那么把 aF 的 father 设为 bF，表示 a 所在集合因为层数较少，所在挂在了 b 所在集合的下面，这样合并之后的大集合 rank 不会有变化。如果 aFrank>bFrank，就把 bF 的 father 设为 aF。如果 aFrank==bFrank，那么 aF 和 bF 谁做大集合的代表都可以，本文的实现是用 aF 作为代表，即把 bF 的 father 设为 aF，此时 aF 的 rank 值增加 1。

合并过程如图 3-46 和图 3-47 所示。

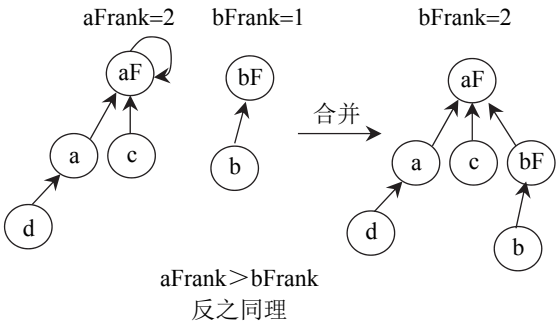

图 3-46

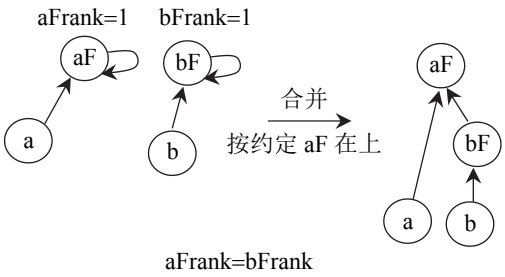

图 3-47

根据两个节点合并两个集合的过程请参看如下 DisjointSets 类中的 union 方法。

```
public class DisjointSets {
        public HashMap<Node, Node> fatherMap;
        public HashMap<Node, Integer> rankMap;

        public DisjointSets() {
                fatherMap = new HashMap<Node, Node>();
                rankMap = new HashMap<Node, Integer>();
        }

        public void makeSets(Node head) {
                fatherMap.clear();
                rankMap.clear();
                preOrderMake(head);
        }

        private void preOrderMake(Node head) {
                if (head == null) {
                        return;
                }
                fatherMap.put(head, head);
```

```
                rankMap.put(head, 0);
                preOrderMake(head.left);
                preOrderMake(head.right);
        }

        public Node findFather(Node n) {
                Node father = fatherMap.get(n);
                if (father != n) {
                        father = findFather(father);
                }
                fatherMap.put(n, father);
                return father;
        }

        public void union(Node a, Node b) {
                if (a == null || b == null) {
                        return;
                }
                Node aFather = findFather(a);
                Node bFather = findFather(b);
                if (aFather != bFather) {
                        int aFrank = rankMap.get(aFather);
                        int bFrank = rankMap.get(bFather);
                        if (aFrank < bFrank) {
                                fatherMap.put(aFather, bFather);
                        } else if (aFrank > bFrank) {
                                fatherMap.put(bFather, aFather);
                        } else {
                                fatherMap.put(bFather, aFather);
                                rankMap.put(aFather, aFrank + 1);
                        }
                }
        }

}
```

介绍完并查集的结构之后，最后解释一下在总流程中如何设置一个集合的祖先节点，如上流程中的每一步都有把当前点 node 所在集合的祖先节点设置为 node 的操作。在整个流程开始之前，建立一张哈希表，参看如下 Tarjan 类中的 ancestorMap，我们知道在并查集中，每个集合都是用该集合的代表节点来表示的。所以，如果想把 node 所在集合的祖先节点设为 node，只用把记录(findFather(node) , node)放入 ancestorMap 中即可。同理，如果想得到一个节点 a 所在集合的祖先节点，令 key 为 findFather(a)，然后从 ancestorMap 中取出相应的记录即可。ancestorMap 同时还可以表示一个节点是否被访问过。

全部的处理流程请参看如下代码中的 tarJanQuery 方法。

```
// 主方法
public Node[] tarJanQuery(Node head, Query[] quries) {
```

```
        Node[] ans = new Tarjan().query(head, quries);
        return ans;
    }

    // Tarjan 类实现处理流程
    public class Tarjan {
        private HashMap<Node, LinkedList<Node>> queryMap;
        private HashMap<Node, LinkedList<Integer>> indexMap;
        private HashMap<Node, Node> ancestorMap;
        private DisjointSets sets;

        public Tarjan() {
            queryMap = new HashMap<Node, LinkedList<Node>>();
            indexMap = new HashMap<Node, LinkedList<Integer>>();
            ancestorMap = new HashMap<Node, Node>();
            sets = new DisjointSets();
        }

        public Node[] query(Node head, Query[] ques) {
            Node[] ans = new Node[ques.length];
            setQueries(ques, ans);
            sets.makeSets(head);
            setAnswers(head, ans);
            return ans;
        }

        private void setQueries(Query[] ques, Node[] ans) {
            Node o1 = null;
            Node o2 = null;
            for (int i = 0; i != ans.length; i++) {
                o1 = ques[i].o1;
                o2 = ques[i].o2;
                if (o1 == o2 || o1 == null || o2 == null) {
                    ans[i] = o1 != null ? o1 : o2;
                } else {
                    if (!queryMap.containsKey(o1)) {
                        queryMap.put(o1, new LinkedList<Node>());
                        indexMap.put(o1, new LinkedList<Integer>());
                    }
                    if (!queryMap.containsKey(o2)) {
                        queryMap.put(o2, new LinkedList<Node>());
                        indexMap.put(o2, new LinkedList<Integer>());
                    }
                    queryMap.get(o1).add(o2);
                    indexMap.get(o1).add(i);
                    queryMap.get(o2).add(o1);
                    indexMap.get(o2).add(i);
                }
            }
        }

        private void setAnswers(Node head, Node[] ans) {
```

```
                if (head == null) {
                        return;
                }
                setAnswers(head.left, ans);
                sets.union(head.left, head);
                ancestorMap.put(sets.findFather(head), head);
                setAnswers(head.right, ans);
                sets.union(head.right, head);
                ancestorMap.put(sets.findFather(head), head);
                LinkedList<Node> nList = queryMap.get(head);
                LinkedList<Integer> iList = indexMap.get(head);
                Node node = null;
                Node nodeFather = null;
                int index = 0;
                while (nList != null && !nList.isEmpty()) {
                        node = nList.poll();
                        index = iList.poll();
                        nodeFather = sets.findFather(node);
                        if (ancestorMap.containsKey(nodeFather)) {
                                ans[index] = ancestorMap.get(nodeFather);
                        }
                }
        }

}
```

二叉树节点间的最大距离问题

【题目】

从二叉树的节点 A 出发，可以向上或者向下走，但沿途的节点只能经过一次，当到达节点 B 时，路径上的节点数叫作 A 到 B 的距离。

比如，图 3-48 所示的二叉树，节点 4 和节点 2 的距离为 2，节点 5 和节点 6 的距离为 5。给定一棵二叉树的头节点 head，求整棵树上节点间的最大距离。

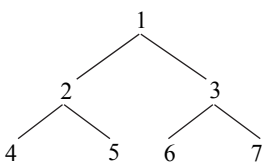

图 3-48

【要求】

如果二叉树的节点数为 N，时间复杂度要求为 $O(N)$。

【难度】

尉 ★★☆☆

【解答】

一个以 h 为头的树上，最大距离只可能来自以下三种情况：

- h 的左子树上的最大距离。
- h 的右子树上的最大距离。
- h 左子树上离 h.left 最远的距离 + 1(h) + h 右子树上离 h.right 最远的距离。

三个值中最大的那个就是整棵 h 树中最远的距离。

根据如上分析，设计解法的过程如下：

1．整个过程为后序遍历，在二叉树的每棵子树上执行步骤 2。

2．假设子树头为 h，处理 h 左子树，得到两个信息，左子树上的最大距离记为 lMax，左子树上距离 h 左孩子的最远距离记为 maxfromLeft。同理，处理 h 右子树得到右子树上的最大距离记为 rMax 和距离 h 右孩子的最远距离记为 maxFromRight。那么 maxfromLeft + 1 + maxFromRight 就是跨 h 节点情况下的最大距离，再与 lMax 和 rMax 比较，把三者中的最值作为 h 树上的最大距离返回，maxfromLeft+1 就是 h 左子树上离 h 最远的点到 h 的距离，maxFromRight+1 就是 h 右子树上离 h 最远的点到 h 的距离，选两者中最大的一个作为 h 树上距离 h 最远的距离返回。如何返回两个值？一个正常返回，另一个用全局变量表示。

具体过程请参看如下代码中的 maxDistance 方法，其中，record[0]就表示另一个返回值。

```
public int maxDistance(Node head) {
        int[] record = new int[1];
        return posOrder(head, record);
}

public int posOrder(Node head, int[] record) {
        if (head == null) {
                record[0] = 0;
                return 0;
        }
        int lMax = posOrder(head.left, record);
        int maxfromLeft = record[0];
```

```
        int rMax = posOrder(head.right, record);
        int maxFromRight = record[0];
        int curNodeMax = maxfromLeft + maxFromRight + 1;
        record[0] = Math.max(maxfromLeft, maxFromRight) + 1;
        return Math.max(Math.max(lMax, rMax), curNodeMax);
    }
```

先序、中序和后序数组两两结合重构二叉树

【题目】

已知一棵二叉树的所有节点值都不同，给定这棵二叉树正确的先序、中序和后序数组。请分别用三个函数实现任意两种数组结合重构原来的二叉树，并返回重构二叉树的头节点。

【难度】

先序与中序结合　士　★☆☆☆
中序与后序结合　士　★☆☆☆
先序与后序结合　尉　★★☆☆

【解答】

先序与中序结合重构二叉树的过程如下：

1. 先序数组中最左边的值就是树的头节点值，记为 h，并用 h 生成头节点，记为 head。然后在中序数组中找到 h，假设位置是 i。那么在中序数组中，i 左边的数组就是头节点左子树的中序数组，假设长度为 l，则左子树的先序数组就是先序数组中 h 往右长度也为 l 的数组。

比如：先序数组为[1,2,4,5,8,9,3,6,7]，中序数组为[4,2,8,5,9,1,6,3,7]，二叉树头节点的值是 1，在中序数组中找到 1 的位置，1 左边的数组为[4,2,8,5,9]，是头节点左子树的中序数组，长度为 5；先序数组中 1 的右边长度也为 5 的数组为[2,4,5,8,9]，就是左子树的先序数组。

2．用左子树的先序和中序数组，递归整个过程建立左子树，返回的头节点记为 left。

3．i 右边的数组就是头节点右子树的中序数组，假设长度为 r。先序数组中右侧等长的部分就是头节点右子树的先序数组。

比如步骤 1 的例子，中序数组中 1 右边的数组为[6,3,7]，长度为 3；先序数组右侧等长的部分为[3,6,7]，它们分别为头节点右子树的中序和先序数组。

4．用右子树的先序和中序数组，递归整个过程建立右子树，返回的头节点记为 right。

5．把 head 的左孩子和右孩子分别设为 left 和 right，返回 head，过程结束。

如果二叉树的节点数为 N，在中序数组中找到位置 i 的过程可以用哈希表来实现，这样整个过程时间复杂度为 $O(N)$。

具体过程请参看如下代码中的 preInToTree 方法。

```
public Node preInToTree(int[] pre, int[] in) {
        if (pre == null || in == null) {
                return null;
        }
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < in.length; i++) {
                map.put(in[i], i);
        }
        return preIn(pre, 0, pre.length - 1, in, 0, in.length - 1, map);
}

public Node preIn(int[] p, int pi, int pj, int[] n, int ni, int nj,
                HashMap<Integer, Integer> map) {
        if (pi > pj) {
                return null;
        }
        Node head = new Node(p[pi]);
        int index = map.get(p[pi]);
        head.left = preIn(p, pi + 1, pi + index - ni, n, ni, index - 1, map);
        head.right = preIn(p, pi + index - ni + 1, pj, n, index + 1, nj, map);
        return head;
}
```

中序和后序重构的过程与先序和中序的过程类似。先序和中序的过程是用先序数组最左的值来对中序数组进行划分，因为这是头节点的值。后序数组中头节点的值是后序数组最右的值，所以用后序最右的值来划分中序数组即可。

具体过程请参看如下代码中的 inPosToTree 方法。

```
public Node inPosToTree(int[] in, int[] pos) {
        if (in == null || pos == null) {
                return null;
        }
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < in.length; i++) {
                map.put(in[i], i);
        }
        return inPos(in, 0, in.length - 1, pos, 0, pos.length - 1, map);
}

public Node inPos(int[] n, int ni, int nj, int[] s, int si, int sj,
```

```
                HashMap<Integer, Integer> map) {
        if (si > sj) {
                return null;
        }
        Node head = new Node(s[sj]);
        int index = map.get(s[sj]);
        head.left = inPos(n, ni, index - 1, s, si, si + index - ni - 1, map);
        head.right = inPos(n, index + 1, nj, s, si + index - ni, sj - 1, map);
        return head;
}
```

先序和后序结合重构二叉树。要求面试者首先分析出节点值都不同的二叉树，即便得到了正确的先序与后序数组，在大多数情况下也不能通过这两个数组把原来的树重构出来。这是因为很多结构不同的树中，先序与后序数组是一样的，比如，头节点为 1、左孩子为 2、右孩子为 null 的树，先序数组为[1,2]，后序数组为[2,1]。而头节点为 1、左孩子为 null、右孩子为 2 的树也是同样的结果。然后需要分析出什么样的树可以被先序和后序数组重建，如果一棵二叉树除叶节点之外，其他所有的节点都有左孩子和右孩子，只有这样的树才可以被先序和后序数组重构出来。最后才是通过划分左右子树各自的先序与后序数组的方式重建整棵树，具体过程请参看如下代码中的 prePosToTree 方法。

```
// 每个节点的孩子数都为 0 或 2 的二叉树才能被先序与后序重构出来
public Node prePosToTree(int[] pre, int[] pos) {
        if (pre == null || pos == null) {
                return null;
        }
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < pos.length; i++) {
                map.put(pos[i], i);
        }
        return prePos(pre, 0, pre.length - 1, pos, 0, pos.length - 1, map);
}

public Node prePos(int[] p, int pi, int pj, int[] s, int si, int sj,
                HashMap<Integer, Integer> map) {
        Node head = new Node(s[sj--]);
        if (pi == pj) {
                return head;
        }
        int index = map.get(p[++pi]);
        head.left = prePos(p, pi, pi + index - si, s, si, index, map);
        head.right = prePos(p, pi + index - si + 1, pj, s, index + 1, sj, map);
        return head;
}
```

通过先序和中序数组生成后序数组

【题目】

已知一棵二叉树所有的节点值都不同，给定这棵树正确的先序和中序数组，不要重建整棵树，而是通过这两个数组直接生成正确的后序数组。

【难度】

士 ★☆☆☆

【解答】

举例说明生成后序数组的过程，假设 pre=[1,2,4,5,3,6,7]，in=[4,2,5,1,6,3,7]。

1. 根据 pre 和 in 的长度，生成长度为 7 的后序数组 pos，按以下规则从右到左填满 pos。

2．根据[1,2,4,5,3,6,7]和[4,2,5,1,6,3,7]，设置 pos[6]=1，即先序数组最左边的值。根据 1 把 in 划分成[4,2,5]和[6,3,7]，pre 中 1 的右边部分根据这两部分等长划分出[2,4,5]和[3,6,7]。[2,4,5]和[4,2,5]一组，[3,6,7]和[6,3,7]一组。

3. 根据[3,6,7]和[6,3,7]，设置 pos[5]=3，再次划分出[6]（来自[3,6,7]）和[6]（来自[6,3,7]）一组，[7]（来自[3,6,7]）和[7]（来自[6,3,7]）一组。

4．根据[7]和[7]设置 pos[4]=7。

5．根据[6]和[6]设置 pos[3]=6。

6. 根据[2,4,5]和[4,2,5]，设置 pos[2]=2，再次划分出[4]（来自[2,4,5]）和[4]（来自[4,2,5]）一组，[5]（来自[[2,4,5]）和[5]（来自[4,2,5]）一组。

7．根据[5]和[5]设置 pos[1]=5。

8．根据[4]和[4]设置 pos[0]=4。

如上过程简单总结为：根据当前的先序和中序数组，设置后序数组最右边的值，然后划分出左子树的先序、中序数组，以及右子树的先序、中序数组，先根据右子树的划分设置好后序数组，再根据左子树的划分，从右边到左边依次设置好后序数组的全部位置。

具体过程请参看如下代码中的 getPosArray 方法。

```
public int[] getPosArray(int[] pre, int[] in) {
        if (pre == null || in == null) {
                return null;
```

```
        }
        int len = pre.length;
        int[] pos = new int[len];
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < len; i++) {
                map.put(in[i], i);
        }
        setPos(pre, 0, len - 1, in, 0, len - 1, pos, len - 1, map);
        return pos;
}

// 从右往左依次填好后序数组 s
// si 为后序数组 s 该填的位置
// 返回值为 s 该填的下一个位置
public int setPos(int[] p, int pi, int pj, int[] n, int ni, int nj,
                int[] s, int si, HashMap<Integer, Integer> map) {
        if (pi > pj) {
                return si;
        }
        s[si--] = p[pi];
        int i = map.get(p[pi]);
        si = setPos(p, pj - nj + i + 1, pj, n, i + 1, nj, s, si, map);
        return setPos(p, pi + 1, pi + i - ni, n, ni, i - 1, s, si, map);
}
```

统计和生成所有不同的二叉树

【题目】

给定一个整数 N，如果 $N<1$，代表空树结构，否则代表中序遍历的结果为{1,2,3,…，N}。请返回可能的二叉树结构有多少。

例如，$N=-1$ 时，代表空树结构，返回 1；$N=2$ 时，满足中序遍历为{1, 2}的二叉树结构只有如图 3-49 所示的两种，所以返回结果为 2。

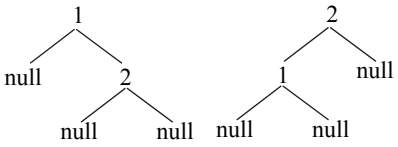

图 3-49

进阶：N 的含义不变，假设可能的二叉树结构有 M 种，请返回 M 个二叉树的头节点，每一棵二叉树代表一种可能的结构。

【难度】

尉 ★★☆☆

【解答】

如果中序遍历有序且无重复值，则二叉树必为搜索二叉树。假设 num(a)代表 *a* 个节点的搜索二叉树有多少种可能，再假设序列为{1, …, *I*, …, *N*}，如果以 1 作为头节点，1 不可能有左子树，故以 1 作为头节点有多少种可能的结构，完全取决于 1 的右子树有多少种可能结构，1 的右子树有 *N*-1 个节点，所以有 num(N-1)种可能。

如果以 *i* 作为头节点，*i* 的左子树有 *i*-1 个节点，所以可能的结构有 num(i-1)种，右子树有 *N*-*i* 个节点，所以有 num(N-i)种可能。故以 *i* 为头节点的可能结构有 num(i-1)×num(N-i)种。

如果以 *N* 作为头节点，*N* 不可能有右子树，故以 *N* 作为头节点有多少种可能，完全取决于 *N* 的左子树有多少种可能，*N* 的左子树有 *N*-1 个节点，所以有 num(N-1)种。

把从 1 到 *N* 分别作为头节点时，所有可能的结构加起来就是答案，可以利用动态规划来加速计算的过程，从而做到 $O(N^2)$的时间复杂度。

具体请参看如下代码中的 numTrees 方法。

```
public int numTrees(int n) {
        if (n < 2) {
                return 1;
        }
        int[] num = new int[n + 1];
        num[0] = 1;
        for (int i = 1; i < n + 1; i++) {
                for (int j = 1; j < i + 1; j++) {
                        num[i] += num[j - 1] * num[i - j];
                }
        }
        return num[n];
}
```

进阶问题与原问题的过程其实是很类似的。如果要生成中序遍历是{a…b}的所有结构，就从 a 开始一直到 b，枚举每一个值作为头节点，把每次生成的二叉树结构的头节点都保存下来即可。假设其中一次是以 *i* 值为头节点的($a \leqslant i \leqslant b$)，以 *i* 头节点的所有结构按如下步骤生成：

1．用{a…*i*-1}递归生成左子树的所有结构，假设所有结构的头节点保存在 listLeft 链表中。

2．用{a…*i*+1}递归生成右子树的所有结构，假设所有结构的头节点保存在 listRight 链表中。

3．在以 *i* 为头节点的前提下，listLeft 中的每一种结构都可以与 listRight 中的每一种结构构成单独的结构，且和其他任何结构都不同。为了保证所有的结构之间不互相交叉，所以对每一种结构都复制出新的树，并记录在总的链表 res 中。

具体过程请参看如下代码中的 generateTrees 方法。

```
public List<Node> generateTrees(int n) {
        return generate(1, n);
}

public List<Node> generate(int start, int end) {
        List<Node> res = new LinkedList<Node>();
        if (start > end) {
                res.add(null);
        }
        Node head = null;
        for (int i = start; i < end + 1; i++) {
                head = new Node(i);
                List<Node> lSubs = generate(start, i - 1);
                List<Node> rSubs = generate(i + 1, end);
                for (Node l : lSubs) {
                        for (Node r : rSubs) {
                                head.left = l;
                                head.right = r;
                                res.add(cloneTree(head));
                        }
                }
        }
        return res;
}

public Node cloneTree(Node head) {
        if (head == null) {
                return null;
        }
        Node res = new Node(head.value);
        res.left = cloneTree(head.left);
        res.right = cloneTree(head.right);
        return res;
}
```

统计完全二叉树的节点数

【题目】

给定一棵完全二叉树的头节点 head，返回这棵树的节点个数。

【要求】

如果完全二叉树的节点数为 N，请实现时间复杂度低于 $O(N)$的解法。

【难度】

尉 ★★☆☆

【解答】

遍历整棵树当然可以求出节点数，但这肯定不是最优解法，本书不再详述。

如果完全二叉树的层数为 h，本书的解法可以做到时间复杂度为 $O(h^2)$，具体过程如下：

1．如果 head==null，说明是空树，直接返回 0。

2．如果不是空树，就求树的高度，求法是找到树的最左节点看能到哪一层，层数记为 h。

3．这一步是求解的主要逻辑，也是一个递归过程记为 bs(node,l,h)，node 表示当前节点，l 表示 node 所在的层数，h 表示整棵树的层数是始终不变的。bs(node,l,h)的返回值表示以 node 为头的完全二叉树的节点数是多少。初始时 node 为头节点 head，l 为 1，因为 head 在第 1 层，一共有 h 层始终不变。那么这个递归的过程可以用两个例子来说明，如图 3-50 和图 3-51 所示。

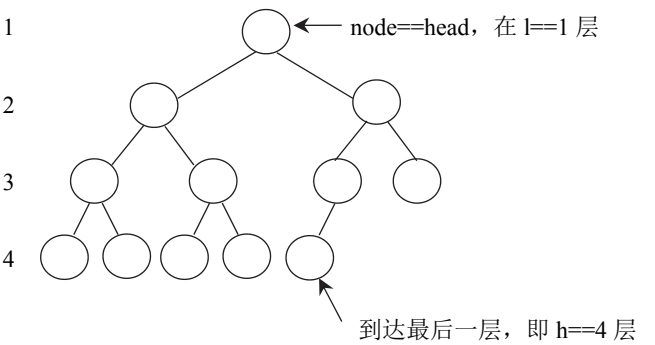

图 3-50

找到 node 右子树的最左节点，如果像图 3-51 的例子一样，发现它能到达最后一层，即 h==4 层。此时说明 node 的整棵左子树都是满二叉树，并且层数为 *h*-l 层，一棵层数为 *h*-l 的满二叉树，其节点数为 2^{h-1}-1 个。如果加上 node 节点自己，那么节点数为 2^(h-1)-1+1==2^(h-1)个。此时如果再知道 node 右子树的节点数，那么以 node 为头的完全二叉树上到底有多少个节点就求出来了。那么 node 右子树的节点数到底是多少呢？就是 bs(node.right,l+1,h)的结果，递归去求即可。最后整体返回 2^(h-1)+bs(node.right,l+1,h)。

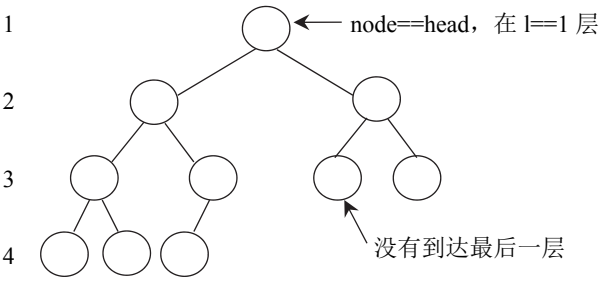

图 3-51

找到 node 右子树的最左节点，如果像图 3-51 的例子一样，发现它没有到达最后一层，说明 node 的整棵右子树都是满二叉树，并且层数为 *h*-*l*-1 层，一棵层数为 *h*-*l*-1 的满二叉树，其节点数为 2^{h-l-1}-1 个。如果加上 node 节点自己，那么节点数为 2^(h-l-1)-1+1==2^(h-l-1)个。此时如果再知道 node 左子树的节点数，那么以 node 为头的完全二叉树上到底有多少个节点就求出来了。node 左子树的节点数到底是多少呢？就是 bs(node.left,l+1,h)的结果，递归去求即可，最后整体返回 2^(h-l-1)+bs(node.left,l+1,h)。

全部过程请参看如下代码中的 nodeNum 方法。

```
public int nodeNum(Node head) {
        if (head == null) {
                return 0;
        }
        return bs(head, 1, mostLeftLevel(head, 1));
}

public int bs(Node node, int l, int h) {
        if (l == h) {
                return 1;
        }
        if (mostLeftLevel(node.right, l + 1) == h) {
                return (1 << (h - l)) + bs(node.right, l + 1, h);
        } else {
                return (1 << (h - l - 1)) + bs(node.left, l + 1, h);
        }
```

```
    }

    public int mostLeftLevel(Node node, int level) {
        while (node != null) {
            level++;
            node = node.left;
        }
        return level - 1;
    }
```

每一层只会选择一个节点 node 进行 bs 的递归过程，所以调用 bs 函数的次数为 $O(h)$。每次调用 bs 函数时，都会查看 node 右子树的最左节点，所以会遍历 $O(h)$个节点，整个过程的时间复杂度为 $O(h^2)$。