```
                    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。

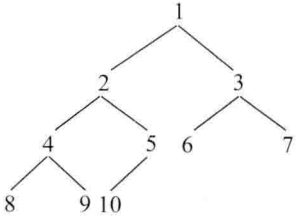

图 3-34

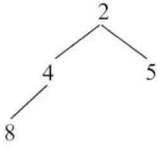

图 3-35

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