在满二叉树中，l -> $O(\log N)$，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。

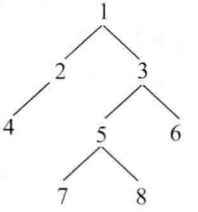

图 3-29

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。

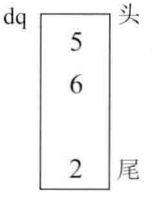

图 3-30

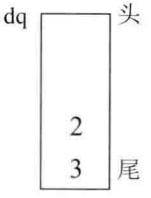

图 3-31

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。

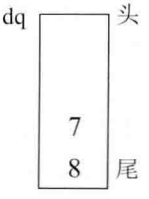

图 3-22

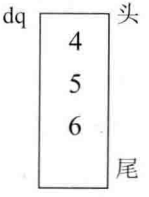

图 3-33

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