```
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。

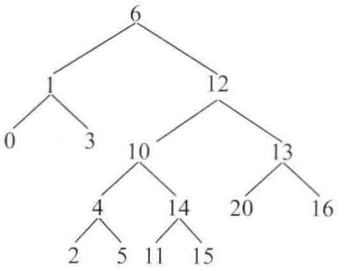

图 3-18

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。

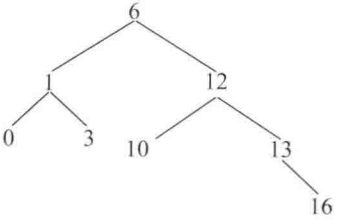

图 3-19

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) {  从每个节点出发寻找
        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);
    }
```

从当前节点形成拓扑结构的节点数

是否能在当前拓扑结构中找到

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。

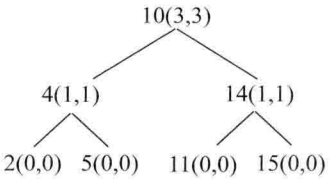

图 3-20

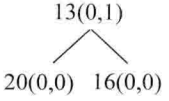

图 3-21

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。

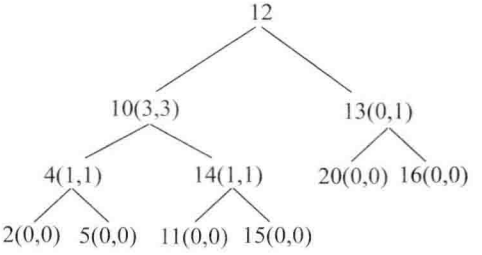

图 3-22

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。

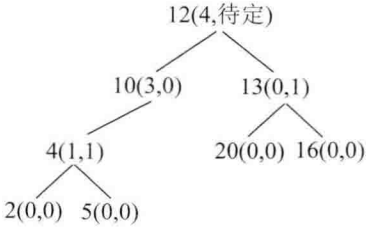

图 3-23

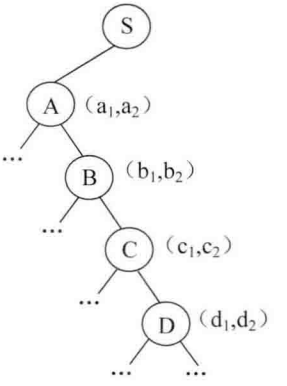

图 3-24

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。

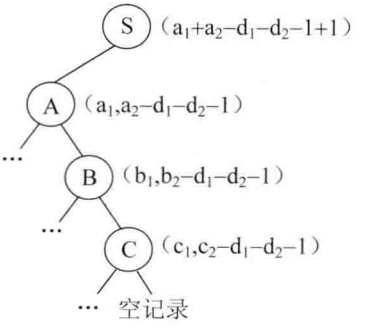

图 3-25

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。

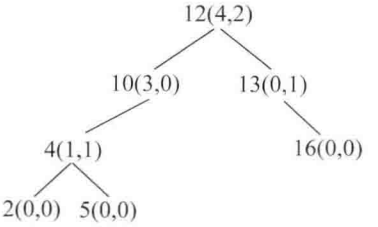

图 3-26

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);
```

左右子树贡献度

```
        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;
        }
}
```

修改贡献度

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。

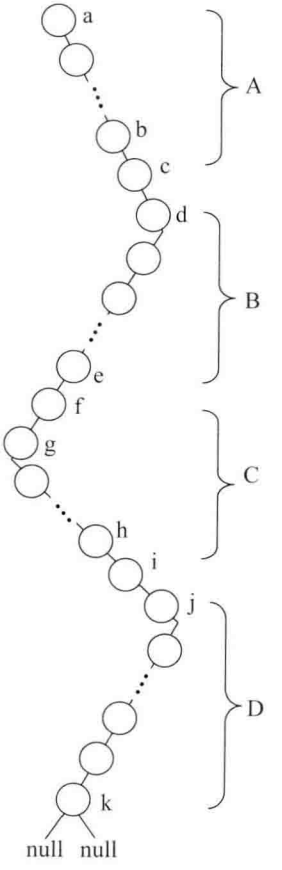

图 3-27

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。

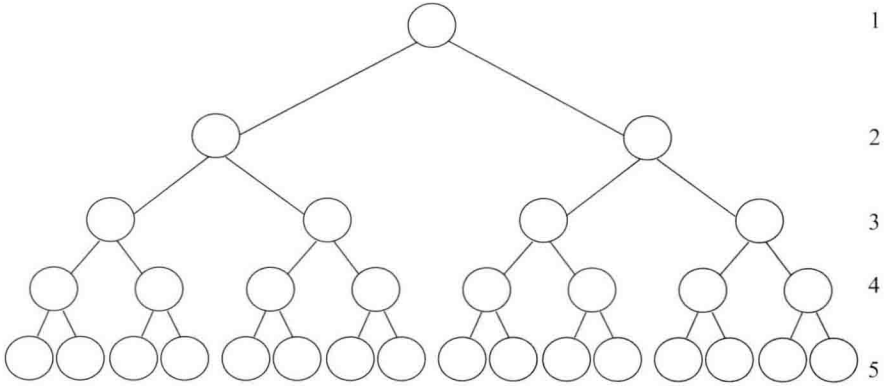

图 3-28

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：

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

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。

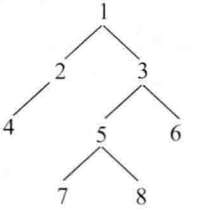

图 3-29

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 ★★☆☆