```
}
```

关于方法二中时间复杂度与空间复杂度的解释，可以用 process 递归函数发生的次数来估算时间复杂度，process 会处理所有的子树，子树的数量就是二叉树节点的个数。所以时间复杂度为 $O(N)$，process 递归函数最多占用二叉树高度为 h 的栈空间，所以额外空间复杂度为 $O(h)$。

【扩展】

相信读者已经注意到，本题在复杂度方面能够达到的程度完全取决于二叉树遍历的实现，如果一个二叉树遍历的实现在时间和空间复杂度上足够好，那么本题就可以做到在时间复杂度和空间复杂度上同样好。如果二叉树的节点数为 N，有没有时间复杂度为 $O(N)$、额外空间复杂度为 $O(1)$的遍历实现呢？如果有这样的实现，那本题也一定有时间复杂度为 $O(N)$、额外空间复杂度为 $O(1)$的方法。既不用栈，也不用递归函数，只用有限的几个变量就可以实现，这样的遍历实现是有的。欢迎有兴趣的读者阅读本书“遍历二叉树的神级方法”问题，然后结合神级的遍历方法再重新实现这道题。

单链表的选择排序

【题目】

给定一个无序单链表的头节点 head，实现单链表的选择排序。

要求：额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

既然要求额外空间复杂度为 $O(1)$，就不能把链表装进数组等容器中排序，排好序之后再重新连接，而是要求面试者在原链表上利用有限几个变量完成选择排序的过程。选择排序是从未排序的部分中找到最小值，然后放在排好序部分的尾部，逐渐将未排序的部分缩小，最后全部变成排好序的部分。本书实现的方法模拟了这个过程。

1．开始时默认整个链表都是未排序的部分，对于找到的第一个最小值节点，肯定是整

个链表的最小值节点，将其设置为新的头节点记为 newHead。

2．每次在未排序的部分中找到最小值的节点，然后把这个节点从未排序的链表中删除，删除的过程当然要保证未排序部分的链表在结构上不至于断开，例如，2->1->3，删除节点 1 之后，链表应该变成 2->3，这就要求我们应该找到要删除节点的前一个节点。

3．把删除的节点（也就是每次的最小值节点）连接到排好序部分的链表尾部。

4．全部过程处理完后，整个链表都已经有序，返回 newHead。

和选择排序一样，如果链表的长度为 N，时间复杂度为 $O(N^2)$，额外空间复杂度为 $O(1)$。

本题依然是考查调整链表的代码技巧，具体过程请参看如下代码中的 selectionSort 方法。

```
public static class Node {
        public int value;
        public Node next;

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

public static Node selectionSort(Node head) {
        Node tail = null; // 排序部分尾部
        Node cur = head; // 未排序部分头部
        Node smallPre = null; // 最小节点的前一个节点
        Node small = null; // 最小的节点
        while (cur != null) {
                small = cur;
                smallPre = getSmallestPreNode(cur);
                if (smallPre != null) {
                        small = smallPre.next;
                        smallPre.next = small.next;
                }
                cur = cur == small ? cur.next : cur;
                if (tail == null) {
                        head = small;
                } else {
                        tail.next = small;
                }
                tail = small;
        }
        return head;
}

public Node getSmallestPreNode(Node head) {
        Node smallPre = null;
        Node small = head;
        Node pre = head;
        Node cur = head.next;
        while (cur != null) {
```

```
                if (cur.value < small.value) {
                        smallPre = pre;
                        small = cur;
                }
                pre = cur;
                cur = cur.next;
        }
        return smallPre;
}
```

一种怪异的节点删除方式

【题目】

链表节点值类型为 int 型，给定一个链表中的节点 node，但不给定整个链表的头节点。如何在链表中删除 node？请实现这个函数，并分析这么会出现哪些问题。

要求：时间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

本题的思路很简单，举例就能说明具体的做法。

例如，链表 1->2->3->null，只知道要删除节点 2，而不知道头节点。那么只需把节点 2 的值变成节点 3 的值，然后在链表中删除节点 3 即可。

这道题目出现的次数很多，这么做看起来非常方便，但其实是有很大问题的。

问题一：这样的删除方式无法删除最后一个节点。还是以原示例来说明，如果知道要删除节点 3，而不知道头节点。但它是最后的节点，根本没有下一个节点来代替节点 3 被删除，那么只有让节点 2 的 next 指向 null 这一种办法，而我们又根本找不到节点 2，所以根本没法正确删除节点 3。读者可能会问，我们能不能把节点 3 在内存上的区域变成 null 呢？这样不就相当于让节点 2 的 next 指针指向了 null，起到节点 3 被删除的效果了吗？不可以。null 在系统中是一个特定的区域，如果想让节点 2 的 next 指针指向 null，必须找到节点 2。

问题二：这种删除方式在本质上根本就不是删除了 node 节点，而是把 node 节点的值改变，然后删除 node 的下一个节点，在实际的工程中可能会带来很大问题。比如，工程上