剑指Offer学习心得

极客学院出版

前言

剑指 Offer 是程序员面试的宝典，有很多前人总结的经典案例。本教程是作者学习剑指 Offer 这本书的心得总结，主要讲解了经典例题，并附加思路和源码，是一本指导程序员顺利就职的教程。

适用人群

适合初级程序员找工作面试前的准备。

学习前提

需要读者掌握 Java 语言的基础。

鸣谢：http://blog.csdn.net/DERRANTCM/article/category/3151215

目录

1

实现 Singleton 模式——七种实现方式

#

题目：设计一个类，我们只能生成该类的一个实例

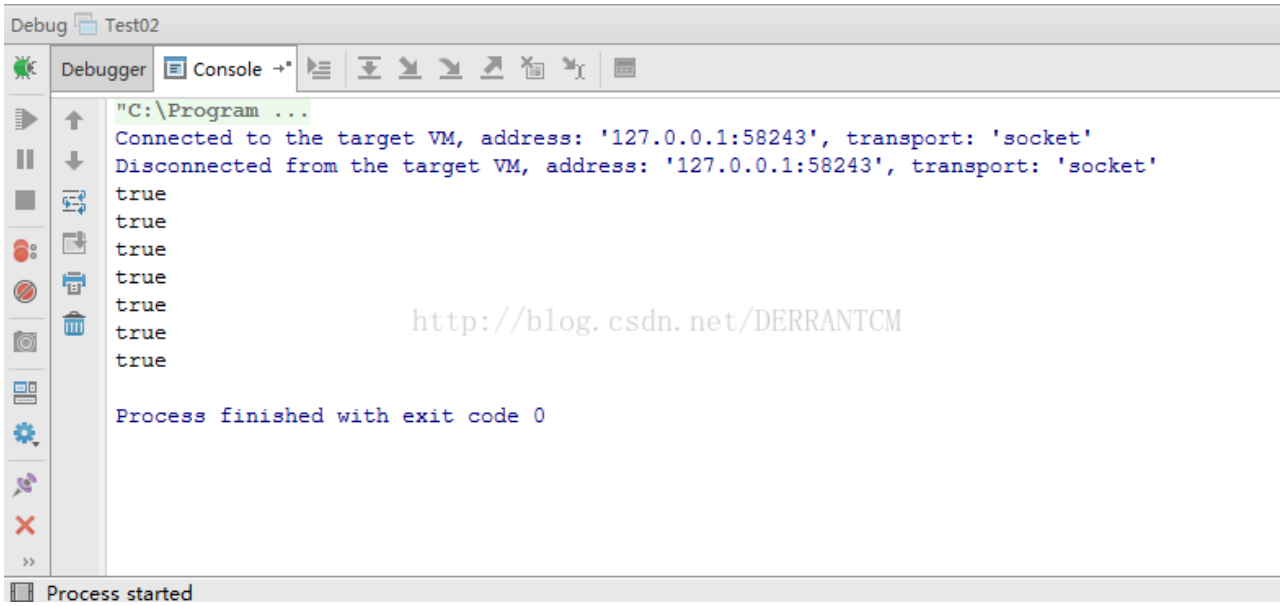
```
[java] view plaincopyprint?
public class Test02 {
    /**
     * 单例模式，懒汉式，线程安全
     */
    public static class Singleton {
        private final static Singleton INSTANCE = new Singleton();
        private Singleton() {
        }
        public static Singleton getInstance() {
            return INSTANCE;
        }
    }
    /**
     * 单例模式，饿汉式，线程不安全
     */
    public static class Singleton2 {
        private static Singleton2 instance = null;
        private Singleton2() {
        }
        public static Singleton2 getInstance() {
            if (instance == null) {
                instance = new Singleton2();
            }
            return instance;
        }
    }
    /**
     * 单例模式，饿汉式，线程安全，多线程环境下效率不高
     */
    public static class Singleton3 {
        private static Singleton3 instance = null;
        private Singleton3() {
        }
        public static synchronized Singleton3 getInstance() {
            if (instance == null) {
                instance = new Singleton3();
            }
            return instance;
```

```
    }
}
/**
 * 单例模式，懒汉式，变种，线程安全
 */
public static class Singleton4 {
    private static Singleton4 instance = null;
    static {
        instance = new Singleton4();
    }
    private Singleton4() {
    }
    public static Singleton4 getInstance() {
        return instance;
    }
}
/**
 * 单例模式，使用静态内部类，线程安全【推荐】
 */
public static class Singleton5 {
    private final static class SingletonHolder {
        private static final Singleton5 INSTANCE = new Singleton5();
    }
    private Singleton5() {
    }
    public static Singleton5 getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
/**
 * 静态内部类，使用枚举方式，线程安全【推荐】
 */
public enum Singleton6 {
    INSTANCE;
    public void whateverMethod() {
    }
}
/**
 * 静态内部类，使用双重校验锁，线程安全【推荐】
 */
public static class Singleton7 {
    private volatile static Singleton7 instance = null;
    private Singleton7() {
    }
    public static Singleton7 getInstance() {
```

```
            if (instance == null) {
                synchronized (Singleton7.class) {
                    if (instance == null) {
                        instance = new Singleton7();
                    }
                }
            }
            return instance;
        }
    }
    public static void main(String[] args) {
        System.out.println(Singleton.getInstance() == Singleton.getInstance());
        System.out.println(Singleton2.getInstance() == Singleton2.getInstance());
        System.out.println(Singleton3.getInstance() == Singleton3.getInstance());
        System.out.println(Singleton4.getInstance() == Singleton4.getInstance());
        System.out.println(Singleton5.getInstance() == Singleton5.getInstance());
        System.out.println(Singleton6.INSTANCE == Singleton6.INSTANCE);
        System.out.println(Singleton7.getInstance() == Singleton7.getInstance());
    }
}
```

#

运行结果：



2

二维数组中的查找

# #

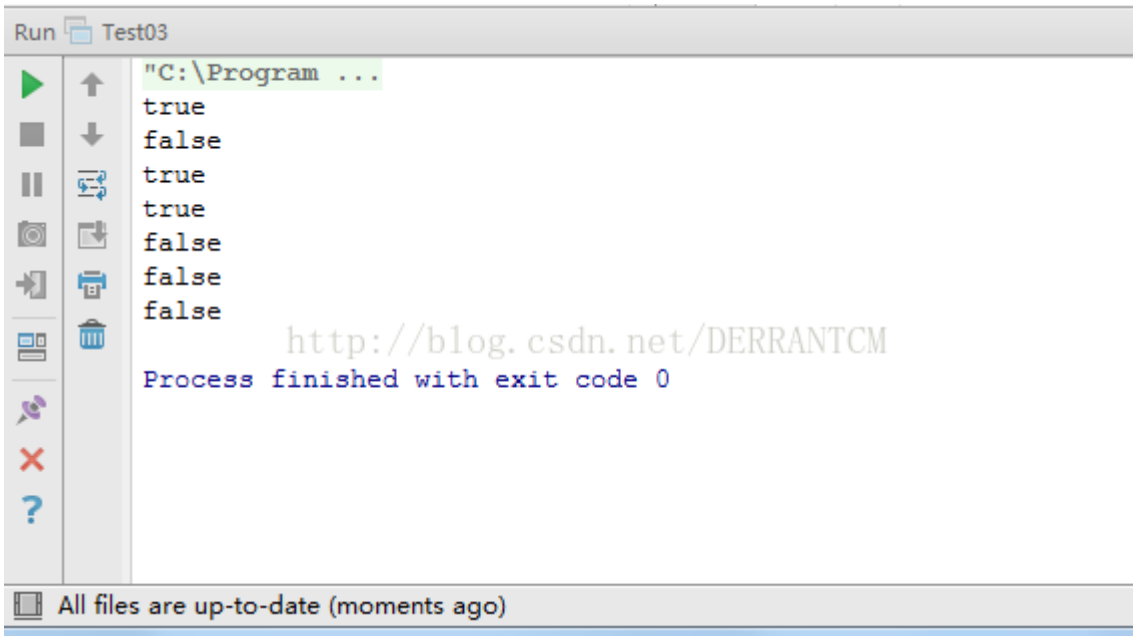
题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

```
[java] view plaincopyprint?
public class Test03 {
    /**
     * 在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。
     * 请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。
     * <p/>
     * 规律：首先选取数组中右上角的数字。如果该数字等于要查找的数字，查找过程结束：
     * 如果该数字大于要查找的数字，剔除这个数字所在的列：如果该数字小于要查找的数字，剔除这个数字所在的行。
     * 也就是说如果要查找的数字不在数组的右上角，则每一次都在数组的查找范围中剔除（行或者一列，这样每一步都可以缩小
     * 查找的范围，直到找到要查找的数字，或者查找范围为空。
     *
     * @param matrix 待查找的数组
     * @param number 要查找的数
     * @return 查找结果，true找到，false没有找到
     */
    public static boolean find(int[][] matrix, int number) {
        // 输入条件判断
        if (matrix == null || matrix.length < 1 || matrix[0].length < 1) {
            return false;
        }
        int rows = matrix.length; // 数组的行数
        int cols = matrix[1].length; // 数组行的列数
        int row = 0; // 起始开始的行号
        int col = cols - 1; // 起始开始的列号
        // 要查找的位置确保在数组之内
        while (row >= 0 && row < rows && col >= 0 && col < cols) {
            if (matrix[row][col] == number) { // 如果找到了就直接退出
                return true;
            } else if (matrix[row][col] > number) { // 如果找到的数比要找的数大，说明要找的数在当前数的左边
                col--; // 列数减一，代表向左移动
            } else { // 如果找到的数比要找的数小，说明要找的数在当前数的下边
                row++; // 行数加一，代表向下移动
            }
        }
        return false;
    }
    public static void main(String[] args) {
```

```
        int[][] matrix = {
                {1, 2, 8, 9},
                {2, 4, 9, 12},
                {4, 7, 10, 13},
                {6, 8, 11, 15}
        };
        System.out.println(find(matrix, 7));    // 要查找的数在数组中
        System.out.println(find(matrix, 5));    // 要查找的数不在数组中
        System.out.println(find(matrix, 1));    // 要查找的数是数组中最小的数字
        System.out.println(find(matrix, 15));   // 要查找的数是数组中最大的数字
        System.out.println(find(matrix, 0));    // 要查找的数比数组中最小的数字还小
        System.out.println(find(matrix, 16));   // 要查找的数比数组中最大的数字还大
        System.out.println(find(null, 16));     // 健壮性测试，输入空指针
    }
}
```

#

运行结果：



3

替换空格

# #

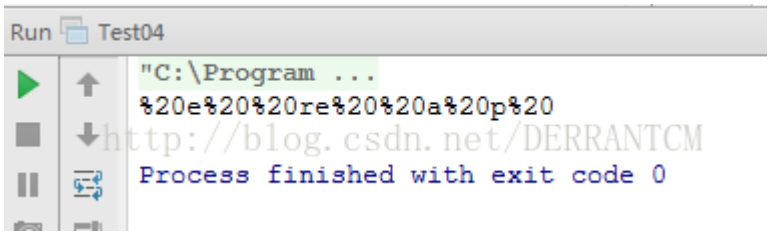
题目：请实现一个函数，把字符串中的每个空格替换成"%20"，例如“We are happy.”，则输出“We%20are%20happy.”。

```
[java] view plaincopyprint?
public class Test04 {
    /**
     * 请实现一个函数，把字符串中的每个空格替换成"%20"，例如“We are happy.“，则输出”We%20are%20happy.“。
     *
     * @param string     要转换的字符数组
     * @param usedLength 已经字符数组中已经使用的长度
     * @return 转换后使用的字符长度，-1表示处理失败
     */
    public static int replaceBlank(char[] string, int usedLength) {
        // 判断输入是否合法
        if (string == null || string.length < usedLength) {
            return -1;
        }
        // 统计字符数组中的空白字符数
        int whiteCount = 0;
        for (int i = 0; i < usedLength; i++) {
            if (string[i] == ' ') {
                whiteCount++;
            }
        }
        // 计算转换后的字符长度是多少
        int targetLength = whiteCount * 2 + usedLength;
        int tmp = targetLength; // 保存长度结果用于返回
        if (targetLength > string.length) { // 如果转换后的长度大于数组的最大长度，直接返回失败
            return -1;
        }
        // 如果没有空白字符就不用处理
        if (whiteCount == 0) {
            return usedLength;
        }
        usedLength--; // 从后向前，第一个开始处理的字符
        targetLength--; // 处理后的字符放置的位置
        // 字符中有空白字符，一直处理到所有的空白字符处理完
        while (usedLength >= 0 && usedLength < targetLength) {
            // 如是当前字符是空白字符，进行"%20"替换
            if (string[usedLength] == ' ') {
```

```
                string[targetLength--] = '0';
                string[targetLength--] = '2';
                string[targetLength--] = '%';
            } else { // 否则移动字符
                string[targetLength--] = string[usedLength];
            }
            usedLength--;
        }
        return tmp;
    }
    public static void main(String[] args) {
        char[] string = new char[50];
        string[0] = ' ';
        string[1] = 'e';
        string[2] = ' ';
        string[3] = ' ';
        string[4] = 'r';
        string[5] = 'e';
        string[6] = ' ';
        string[7] = ' ';
        string[8] = 'a';
        string[9] = ' ';
        string[10] = 'p';
        string[11] = ' ';
        int length = replaceBlank(string, 12);
        System.out.println(new String(string, 0, length));
    }
}
```

#

运行结果：



4

从尾到头打印链表

# #

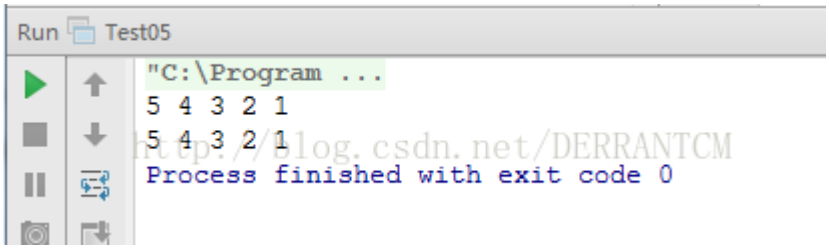
题目：输入个链表的头结点，从尾到头反过来打印出每个结点的值。

```
[java] view plaincopyprint?
public class Test05 {
    /**
     * 结点对象
     */
    public static class ListNode {
        int val; // 结点的值
        ListNode nxt; // 下一个结点
    }
    /**
     * 输入个链表的头结点，从尾到头反过来打印出每个结点的值
     * 使用栈的方式进行
     *
     * @param root 链表头结点
     */
    public static void printListInverselyUsingIteration(ListNode root) {
        Stack<ListNode> stack = new Stack<>();
        while (root != null) {
            stack.push(root);
            root = root.nxt;
        }
        ListNode tmp;
        while (!stack.isEmpty()) {
            tmp = stack.pop();
            System.out.print(tmp.val + " ");
        }
    }
    /**
     * 输入个链表的头结点，从尾到头反过来打印出每个结点的值
     * 使用栈的方式进行
     *
     * @param root 链表头结点
     */
    public static void printListInverselyUsingRecursion(ListNode root) {
        if (root != null) {
            printListInverselyUsingRecursion(root.nxt);
            System.out.print(root.val + " ");
        }
    }
```

```
    public static void main(String[] args) {
        ListNode root = new ListNode();
        root.val = 1;
        root.nxt = new ListNode();
        root.nxt.val = 2;
        root.nxt.nxt = new ListNode();
        root.nxt.nxt.val = 3;
        root.nxt.nxt.nxt = new ListNode();
        root.nxt.nxt.nxt.val = 4;
        root.nxt.nxt.nxt.nxt = new ListNode();
        root.nxt.nxt.nxt.nxt.val = 5;
        printListInverselyUsingIteration(root);
        System.out.println();
        printListInverselyUsingRecursion(root);
    }
}
```

#

运行结果：



5

重建二叉树

# #

题目：输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如：前序遍历序列 { 1， 2， 4，7，3，5，6，8 } 和中序遍历序列 { 4，7，2，1，5，3，8，6}，重建出下图所示的二叉树并输出它的头结点。

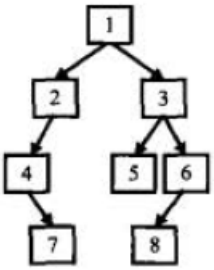


图 2.6　根据前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}重建的二叉树

#

代码如下：

```
[java] view plaincopyprint?
public class Test06 {
    /**
     * 二叉树节点类
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }
    /**
     * 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二节树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。
     *
     * @param preorder 前序遍历
     * @param inorder  中序遍历
     * @return 树的根结点
     */
    public static BinaryTreeNode construct(int[] preorder, int[] inorder) {
        // 输入的合法性判断，两个数组都不能为空，并且都有数据，而且数据的数目相同
        if (preorder == null || inorder == null || preorder.length != inorder.length || inorder.length < 1) {
            return null;
        }
        return construct(preorder, 0, preorder.length - 1, inorder, 0, inorder.length - 1);
    }
    /**
     * 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二节树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。
     *
     * @param preorder 前序遍历
     * @param ps       前序遍历的开始位置
     * @param pe       前序遍历的结束位置
     * @param inorder  中序遍历
     * @param is       中序遍历的开始位置
     * @param ie       中序遍历的结束位置
     * @return 树的根结点
     */
    public static BinaryTreeNode construct(int[] preorder, int ps, int pe, int[] inorder, int is, int ie) {
        // 开始位置大于结束位置说明已经没有需要处理的元素了
        if (ps > pe) {
            return null;
```

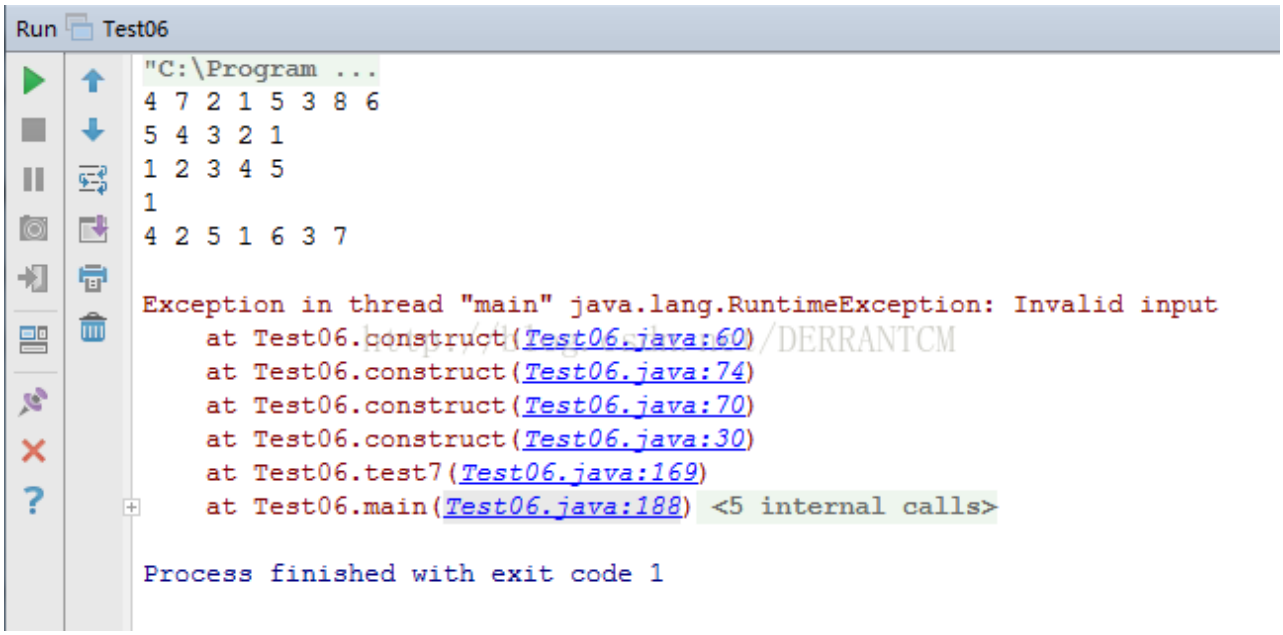
```
    }
    // 取前序遍历的第一个数字，就是当前的根结点
    int value = preorder[ps];
    int index = is;
    // 在中序遍历的数组中找根结点的位置
    while (index <= ie && inorder[index] != value) {
        index++;
    }
    // 如果在整个中序遍历的数组中没有找到，说明输入的参数是不合法的，抛出异常
    if (index > ie) {
        throw new RuntimeException("Invalid input");
    }
    // 创建当前的根结点，并且为结点赋值
    BinaryTreeNode node = new BinaryTreeNode();
    node.value = value;
    // 递归构建当前根结点的左子树，左子树的元素个数：index-is+1个
    // 左子树对应的前序遍历的位置在[ps+1, ps+index-is]
    // 左子树对应的中序遍历的位置在[is, index-1]
    node.left = construct(preorder, ps + 1, ps + index - is, inorder, is, index - 1);
    // 递归构建当前根结点的右子树，右子树的元素个数：ie-index个
    // 右子树对应的前序遍历的位置在[ps+index-is+1, pe]
    // 右子树对应的中序遍历的位置在[index+1, ie]
    node.right = construct(preorder, ps + index - is + 1, pe, inorder, index + 1, ie);
    // 返回创建的根结点
    return node;
}
// 中序遍历二叉树
public static void printTree(BinaryTreeNode root) {
    if (root != null) {
        printTree(root.left);
        System.out.print(root.value + " ");
        printTree(root.right);
    }
}
// 普通二叉树
//            1
//         /    \
//        2      3
//       /      / \
//      4      5   6
//       \      /
//        7     8
private static void test1() {
    int[] preorder = {1, 2, 4, 7, 3, 5, 6, 8};
    int[] inorder = {4, 7, 2, 1, 5, 3, 8, 6};
```

```
    BinaryTreeNode root = construct(preorder, inorder);
    printTree(root);
}
// 所有结点都没有右子结点
//           1
//          /
//         2
//        /
//       3
//      /
//     4
//    /
//   5
private static void test2() {
    int[] preorder = {1, 2, 3, 4, 5};
    int[] inorder = {5, 4, 3, 2, 1};
    BinaryTreeNode root = construct(preorder, inorder);
    printTree(root);
}
// 所有结点都没有左子结点
//           1
//            \
//             2
//              \
//               3
//                \
//                 4
//                  \
//                   5
private static void test3() {
    int[] preorder = {1, 2, 3, 4, 5};
    int[] inorder = {1, 2, 3, 4, 5};
    BinaryTreeNode root = construct(preorder, inorder);
    printTree(root);
}
// 树中只有一个结点
private static void test4() {
    int[] preorder = {1};
    int[] inorder = {1};
    BinaryTreeNode root = construct(preorder, inorder);
    printTree(root);
}
// 完全二叉树
//             1
//          /    \
```

```
    //         2      3
    //        / \    / \
    //       4   5  6   7
    private static void test5() {
        int[] preorder = {1, 2, 4, 5, 3, 6, 7};
        int[] inorder = {4, 2, 5, 1, 6, 3, 7};
        BinaryTreeNode root = construct(preorder, inorder);
        printTree(root);
    }
    // 输入空指针
    private static void test6() {
        construct(null, null);
    }
    // 输入的两个序列不匹配
    private static void test7() {
        int[] preorder = {1, 2, 4, 5, 3, 6, 7};
        int[] inorder = {4, 2, 8, 1, 6, 3, 7};
        BinaryTreeNode root = construct(preorder, inorder);
        printTree(root);
    }
    public static void main(String[] args) {
        test1();
        System.out.println();
        test2();
        System.out.println();
        test3();
        System.out.println();
        test4();
        System.out.println();
        test5();
        System.out.println();
        test6();
        System.out.println();
        test7();
    }
}
```

#

运行结果：


Run
Test06
"C:\Program ...
4 7 2 1 5 3 8 6
5 4 3 2 1
1 2 3 4 5
1
4 2 5 1 6 3 7
Exception in thread "main" java.lang.RuntimeException: Invalid input
at Test06.construct(Test06.java:60)
at Test06.construct(Test06.java:74)
at Test06.construct(Test06.java:70)
at Test06.construct(Test06.java:30)
at Test06.test7(Test06.java:169)
at Test06.main(Test06.java:188) <5 internal calls>
Process finished with exit code 1

6

用两个栈实现队列

# #

题目：用两个栈实现一个队列。队列的声明如下，请实现它的两个函数appendTail 和deleteHead，分别完成在队列尾部插入结点和在队列头部删除结点的功能。

我们通过一个具体的例子来分析往该队列插入和删除元素的过程。首先插入一个元素 a，不妨先把它插入到 stack1，此时 stack1 中的元素有{a}，stack2 为空。再压入两个元素 b 和 c，还是插入到 stack1 中，此时 stack1 中的元素有{a, b, c}，其中 c 位于栈顶，而 stack2 仍然是空的（如图 2.8（a）所示）。

这个时候我们试着从队列中删除一个元素。按照队列先入先出的规则，由于 a 比 b、c 先插入到队列中，最先被删除的元素应该是 a。元素 a 存储在 stack1 中，但并不在栈顶上，因此不能直接进行删除。注意到 stack2 我们还一直没有使用过，现在是让 stack2 发挥作用的时候了。如果我们把 stack1 中的元素逐个弹出并压入 stack2，元素在 stack2 中的顺序正好和原来在 stack1 中的顺序相反。因此经过 3 次弹出 stack1 和压入 stack2 操作之后，stack1 为空，而 stack2 中的元素是{c,b,a}，这个时候就可以弹出 stack2 的栈顶 a 了。此时的 stack1 为空，而 stack2 的元素为{c,b}，其中 b 在栈顶（如图 2.8（b）所示）。

如果我们还想继续删除队列的头部应该怎么办呢？剩下的两个元素是 b 和 c，b 比 c 早进入队列，因此 b 应该先删除。而此时 b 恰好又在栈顶上，因此直

#

代码如下：

```
[java] view plaincopyprint?
public class Test07 {
    /**
     * 用两个栈模拟的队列
     * 用两个核实现一个队列。队列的声明如下，诸实现它的两个函数appendTail和deleteHead，
     * 分别完成在队列尾部插入结点和在队列头部删除结点的功能。
     */
    public static class MList<T> {
```

```
    // 插入栈，只用于插入的数据
    private Stack<T> stack1 = new Stack<>();
    // 弹出栈，只用于弹出数据
    private Stack<T> stack2 = new Stack<>();
    public MList() {
    }
    // 添加操作，成在队列尾部插入结点
    public void appendTail(T t) {
        stack1.add(t);
    }
    // 删除操作，在队列头部删除结点
    public T deleteHead() {
        // 先判断弹出栈是否为空，如果为空就将插入栈的所有数据弹出栈，
        // 并且将弹出的数据压入弹出栈中
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.add(stack1.pop());
            }
        }
        // 如果弹出栈中还没有数据就抛出异常
        if (stack2.isEmpty()) {
            throw new RuntimeException("No more element.");
        }
        // 返回弹出栈的栈顶元素，对应的就是队首元素。
        return stack2.pop();
    }
  }
}
```

7

旋转数组的最小数字

# #

题目： 把一个数组最开始的若干个元素搬到数组的末尾， 我们称之数组的旋转。输入一个递增排序的数组的一个旋转， 输出旋转数组的最小元素。例如数组{3，4, 5, 1, 2 } 为 { l1,2,3, 4，5}的一个旋转，该数组的最小值为1。

#

实现代码如下：

```
[java] view plaincopyprint?
public class Test08 {
    /**
     * 把一个数组最开始的若干个元素搬到数组的末尾， 我们称之数组的旋转。
     * 输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。
     * 例如数组{3, 4, 5, 1, 2 } 为 { l ,2, 3, 4, 5}的一个旋转，该数组的最小值为
     *
     * @param numbers 旋转数组
     * @return 数组的最小值
     */
    public static int min(int[] numbers) {
        // 判断输入是否合法
        if (numbers == null || numbers.length == 0) {
            throw new RuntimeException("Invalid input.");
        }
        // 开始处理的第一个位置
        int lo = 0;
        // 开始处理的最后一个位置
        int hi = numbers.length - 1;
        // 设置初始值
        int mi = lo;
        // 确保lo在前一个排好序的部分，hi在排好序的后一个部分
        while (numbers[lo] >= numbers[hi]) {
            // 当处理范围只有两个数据时，返回后一个结果
            // 因为numbers[lo] >= numbers[hi]总是成立，后一个结果对应的是最小的值
            if (hi - lo == 1) {
                return numbers[hi];
            }
            // 取中间的位置
            mi = lo + (hi - lo) / 2;
            // 如果三个数都相等，则需要进行顺序处理，从头到尾找最小的值
```

```
        if (numbers[mi] == numbers[lo] && numbers[hi] == numbers[mi]) {
            return minInorder(numbers, lo, hi);
        }
        // 如果中间位置对应的值在前一个排好序的部分，将lo设置为新的处理位置
        if (numbers[mi] >= numbers[lo]) {
            lo = mi;
        }
        // 如果中间位置对应的值在后一个排好序的部分，将hi设置为新的处理位置
        else if (numbers[mi] <= numbers[hi]) {
            hi = mi;
        }
    }
    // 返回最终的处理结果
    return numbers[mi];
}
/**
 * 找数组中的最小值
 *
 * @param numbers 数组
 * @param start   数组的起始位置
 * @param end     数组的结束位置
 * @return 找到的最小的数
 */
public static int minInorder(int[] numbers, int start, int end) {
    int result = numbers[start];
    for (int i = start + 1; i <= end; i++) {
        if (result > numbers[i]) {
            result = numbers[i];
        }
    }
    return result;
}
public static void main(String[] args) {
    // 典型输入，单调升序的数组的一个旋转
    int[] array1 = {3, 4, 5, 1, 2};
    System.out.println(min(array1));
    // 有重复数字，并且重复的数字刚好的最小的数字
    int[] array2 = {3, 4, 5, 1, 1, 2};
    System.out.println(min(array2));
    // 有重复数字，但重复的数字不是第一个数字和最后一个数字
    int[] array3 = {3, 4, 5, 1, 2, 2};
    System.out.println(min(array3));
    // 有重复的数字，并且重复的数字刚好是第一个数字和最后一个数字
    int[] array4 = {1, 0, 1, 1, 1};
    System.out.println(min(array4));
```

```
        // 单调升序数组，旋转0个元素，也就是单调升序数组本身
        int[] array5 = {1, 2, 3, 4, 5};
        System.out.println(min(array5));
        // 数组中只有一个数字
        int[] array6 = {2};
        System.out.println(min(array6));
        // 数组中数字都相同
        int[] array7 = {1, 1, 1, 1, 1, 1, 1};
        System.out.println(min(array7));
        System.out.println(min(array6));
        // 输入NULL
        System.out.println(min(null));
    }
}
```

#

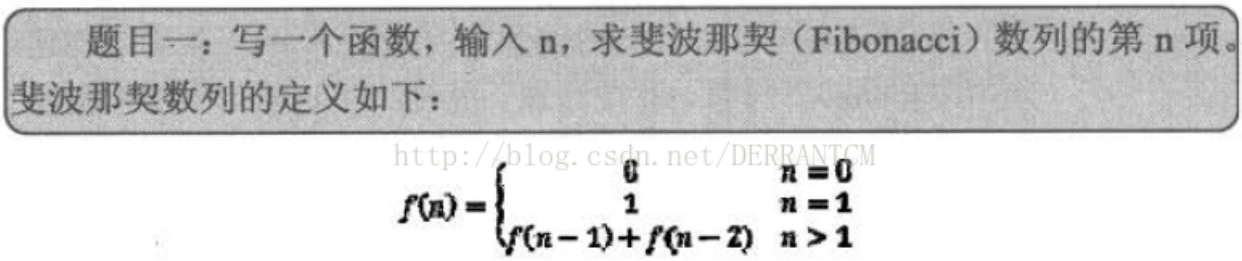
运行结果：

8

斐波那契数列

#

O（n）时间 O（1）空间实现：




```
[java] view plaincopyprint?
public class Test09 {
    /**
     * 写一个函数，输入n，求斐波那契（Fibonacci）数列的第n项
     * @param n Fibonacci数的项数
     * @return 第n项的结果
     */
    public static long fibonacci(int n) {
        // 当输入非正整数的时候返回0
        if (n <= 0) {
            return 0;
        }
        // 输入1或者2的时候返回1
        if (n == 1 || n == 2) {
            return 1;
        }
        // 记录前两个（第n-2个）的Fibonacci数的值
        long prePre = 1;
        // 记录前两个（第n-1个）的Fibonacci数的值
        long pre = 1;
        // 记录前两个（第n个）的Fibonacci数的值
        long current = 2;
        // 求解第n个的Fibonacci数的值
        for (int i = 3; i <= n ; i++) {
            // 求第i个的Fibonacci数的值
            current = prePre + pre;
            // 更新记录的结果，prePre原先记录第i-2个Fibonacci数的值
            // 现在记录第i-1个Fibonacci数的值
            prePre = pre;
            // 更新记录的结果，pre原先记录第i-1个Fibonacci数的值
            // 现在记录第i个Fibonacci数的值
```

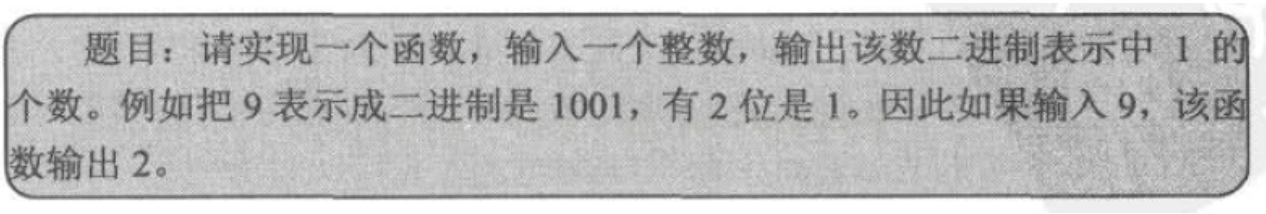
```
            pre = current;
        }
        // 返回所求的结果
        return current;
    }
    public static void main(String[] args) {
        System.out.println(fibonacci(0));
        System.out.println(fibonacci(1));
        System.out.println(fibonacci(2));
        System.out.println(fibonacci(3));
        System.out.println(fibonacci(4));
        System.out.println(fibonacci(5));
        System.out.println(fibonacci(6));
        System.out.println(fibonacci(7));
    }
}
```

#

运行结果：

9

二进制中1的个数


题目：请实现一个函数，输入一个整数，输出该数二进制表示中 1 的个数。例如把 9 表示成二进制是 1001，有 2 位是 1。因此如果输入 9，该函数输出 2。

#

代码如下，请在 JDK7 及以上版本运行：

```
[java] view plaincopyprint?
public class Test10 {
    /**
     * 请实现一个函数， 输入一个整数，输出该数二进制表示中1的个数。
     * 例如把9表示成二进制是1001 ，有2位是1. 因此如果输入9，该出2。
     *
     * @param n 待的数字
     * @return 数字中二进制表表的1的数目
     */
    public static int numberOfOne(int n) {
        // 记录数字中1的位数
        int result = 0;
        // JAVA语言规范中，int整形占四个字节，总计32位
        // 对每一个位置与1进行求与操作，再累加就可以求出当前数字的表示是多少位1
        for (int i = 0; i < 32; i++) {
            result += (n & 1);
            n >>>= 1;
        }
        // 返回求得的结果
        return result;
    }
    /**
     * 请实现一个函数， 输入一个整数，输出该数二进制表示中1的个数。
     * 例如把9表示成二进制是1001 ，有2位是1. 因此如果输入9，该出2。
     * 【 这种方法的效率更高 】
     *
     * @param n 待的数字
     * @return 数字中二进制表表的1的数目
     */
    public static int numberOfOne2(int n) {
        // 记录数字中1的位数
        int result = 0;
        // 数字的二进制表示中有多少个1就进行多少次操作
```

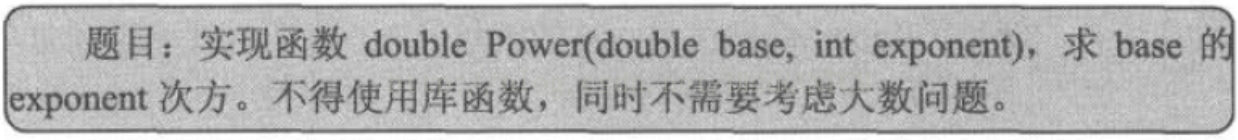
```
        while (n != 0) {
            result++;
            // 从最右边的1开始，每一次操作都使n的最右的一个1变成了0，
            // 即使是符号位也会进行操作。
            n = (n - 1) & n;
        }
        // 返回求得的结果
        return result;
    }
    public static void main(String[] args) {
        System.out.println(numberOfOne(0B00000000_00000000_00000000_00000000)); // 0
        System.out.println(numberOfOne(0B00000000_00000000_00000000_00000001)); // 1
        System.out.println(numberOfOne(0B11111111_11111111_11111111_11111111)); // -1
        System.out.println(0B01111111_11111111_11111111_11111111 == Integer.MAX_VALUE);
        System.out.println(numberOfOne(0B01111111_11111111_11111111_11111111)); // Integer.MAX_VALUE
        System.out.println(0B10000000_00000000_00000000_00000000 == Integer.MIN_VALUE);
        System.out.println(numberOfOne(0B10000000_00000000_00000000_00000000)); // Integer.MIN_VALUE
        System.out.println("");
        System.out.println(numberOfOne2(0B00000000_00000000_00000000_00000000)); // 0
        System.out.println(numberOfOne2(0B00000000_00000000_00000000_00000001)); // 1
        System.out.println(numberOfOne2(0B11111111_11111111_11111111_11111111)); // -1
        System.out.println(numberOfOne2(0B01111111_11111111_11111111_11111111)); // Integer.MAX_VALUE
        System.out.println(numberOfOne2(0B10000000_00000000_00000000_00000000)); // Integer.MIN_VALUE
    }
}
```

#

运行结果：

10

数值的整数次方



#

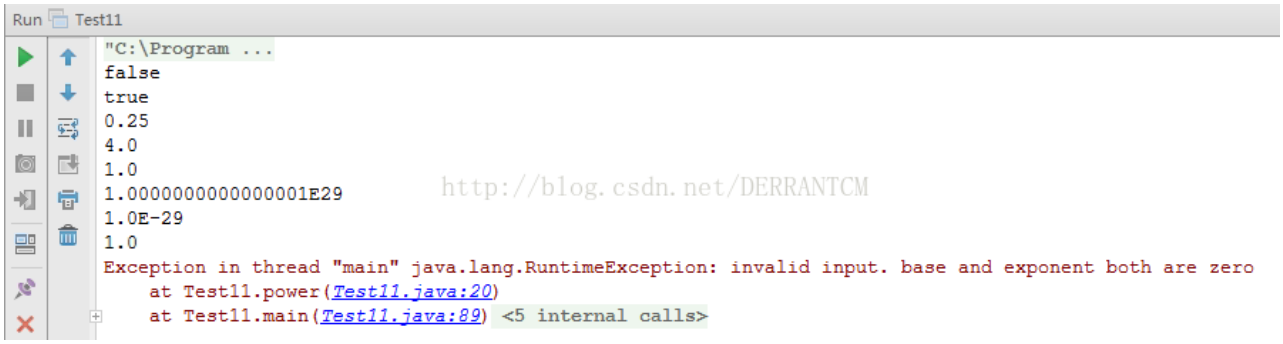
代码实现：

```
[java] view plaincopyprint?
public class Test11 {
    /**
     * 实现函数double Power(double base, int exponent)，求base的exponent次方。
     * 不得使用库函数，同时不需要考虑大数问题。
     *
     * @param base     指次
     * @param exponent 幂
     * @return 结果
     */
    public static double power(double base, int exponent) {
        // 指数和底数不能同时为0
        if (base == 0 && exponent == 0) {
            throw new RuntimeException("invalid input. base and exponent both are zero");
        }
        // 指数为0就返回1
        if (exponent == 0) {
            return 1;
        }
        // 求指数的绝对值
        long exp = exponent;
        if (exponent < 0) {
            exp = -exp;
        }
        // 求幂次方
        double result = powerWithUnsignedExponent(base, exp);
        // 指数是负数，要进行求倒数
        if (exponent < 0) {
            result = 1 / result;
        }
        // 返回结果
        return result;
    }
    /**
     * 求一个数的正整数次幂，不考虑溢出
```

```
     *
     * @param base     指次
     * @param exponent 幂
     * @return 结果
     */
    public static double powerWithUnsignedExponent(double base, long exponent) {
        // 如果指数为0，返回1
        if (exponent == 0) {
            return 1;
        }
        // 指数为1，返回底数
        if (exponent == 1) {
            return base;
        }
        // 递归求一半的值
        double result = powerWithUnsignedExponent(base, exponent >> 2);
        // 求最终的值，如果是奇数就还要剩以一次底数
        result *= result;
        if (exponent % 2 != 0) {
            result *= base;
        }
        // 返回结果
        return result;
    }
    public static void main(String[] args) {
        System.out.println(0.00000000000000000000000001111 == 0);
        System.out.println(0.00000000000000000000000000000 == 0);
        System.out.println(power(2, -4));
        System.out.println(power(2, 4));
        System.out.println(power(2, 0));
        System.out.println(power(0.00000000000000000000000000001, -1));
        System.out.println(power(0.00000000000000000000000000001, 1));
        System.out.println(power(0.00000000000000000000000000001, 0));
        System.out.println(power(0.00000000000000000000000000000, 0));
    }
```

#

运行结果：


Run
Test11
"C:\Program ...
false
true
0.25
4.0
1.0
1.0000000000000001E29
1.0E-29
1.0
Exception in thread "main" java.lang.RuntimeException: invalid input. base and exponent both are zero
at Test11.power(Test11.java:20)
at Test11.main(Test11.java:89) <5 internal calls>
http://blog.csdn.net/DERRANTCM

11

打印 1 到最大的 n 位数

题目：输入数字 n，按顺序打印出从 1 最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数即 999。

#

代码实现：

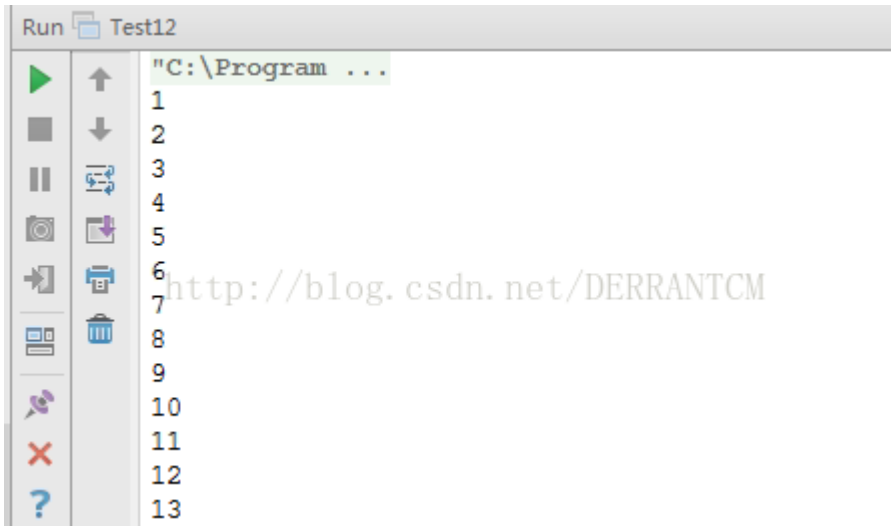
```
[java] view plaincopyprint?
public class Test12 {
    /**
     * 输入数字n，按顺序打印出从1最大的n位十进制数。比如输入3，则打印出1、2、3 一直到最大的3位数即999。
     *
     * @param n 数字的最大位数
     */
    public static void printOneToNthDigits(int n) {
        // 输入的数字不能为小于1
        if (n < 1) {
            throw new RuntimeException("The input number must larger than 0");
        }
        // 创建一个数组用于打存放值
        int[] arr = new int[n];
        printOneToNthDigits(0, arr);
    }
    /**
     * 输入数字n，按顺序打印出从1最大的n位十进制数。
     *
     * @param n  当前处理的是第个元素，从0开始计数
     * @param arr 存放结果的数组
     */
    public static void printOneToNthDigits(int n, int[] arr) {
        // 说明所有的数据排列选择已经处理完了
        if (n >= arr.length) {
            // 可以输入数组的值
            printArray(arr);
        } else {
            // 对
            for (int i = 0; i <= 9; i++) {
                arr[n] = i;
                printOneToNthDigits(n + 1, arr);
            }
        }
    }
```

```
/**
 * 输入数组的元素，从左到右，从第一个非0值到开始输出到最后的元素。
 *
 * @param arr 要输出的数组
 */
public static void printArray(int[] arr) {
    // 找第一个非0的元素
    int index = 0;
    while (index < arr.length && arr[index] == 0) {
        index++;
    }
    // 从第一个非0值到开始输出到最后的元素。
    for (int i = index; i < arr.length; i++) {
        System.out.print(arr[i]);
    }
    // 条件成立说明数组中有非零元素，所以需要换行
    if (index < arr.length) {
        System.out.println();
    }
}
/**
 * 输入数字n，按顺序打印出从1最大的n位十进制数。比如输入3，则打印出1、2、3 一直到最大的3位数即999。
 * 【第二种方法，比上一种少用内存空间】
 *
 * @param n 数字的最大位数
 */
public static void printOneToNthDigits2(int n) {
    // 输入值必须大于0
    if (n < 1) {
        throw new RuntimeException("The input number must larger than 0");
    }
    // 创建一个长度为n的数组
    int[] arr = new int[n];
    // 为数组元素赋初始值
    for (int i = 0; i < arr.length; i++) {
        arr[i] = 0;
    }
    // 求结果，如果最高位没有进位就一直进行处理
    while (addOne(arr) == 0) {
        printArray(arr);
    }
}
/**
 * 对arr表示的数组的最低位加1 arr中的每个数都不能超过9不能小于0，每个位置模拟一个数位
 *
```

```
     * @param arr 待加数组
     * @return 判断最高位是否有进位，如果有进位就返回1，否则返回0
     */
    public static int addOne(int[] arr) {
        // 保存进位值，因为每次最低位加1
        int carry = 1;
        // 最低位的位置的后一位
        int index = arr.length;
        do {
            // 指向上一个处理位置
            index--;
            // 处理位置的值加上进位的值
            arr[index] += carry;
            // 求处理位置的进位
            carry = arr[index] / 10;
            // 求处理位置的值
            arr[index] %= 10;
        } while (carry != 0 && index > 0);
        // 如果index=0说明已经处理了最高位，carry>0说明最高位有进位，返回1
        if (carry > 0 && index == 0) {
            return 1;
        }
        // 无进位返回0
        return 0;
    }
    public static void main(String[] args) {
        printOneToNthDigits2(2);
        System.out.println();
        printOneToNthDigits(2);
    }
}
```

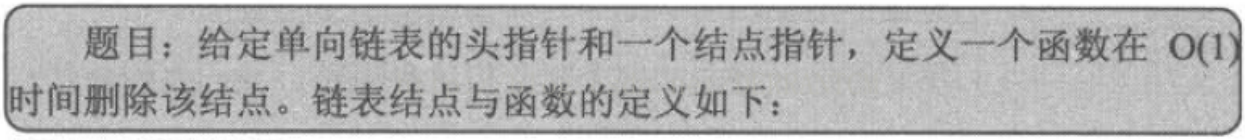
#

运行结果：


Run
Test12
"C:\Program ...
1
2
3
4
5
6
7
8
9
10
11
12
13
http://blog.csdn.net/DERRANTCM

12

在 O（1）时间删除链表结点


题目：给定单向链表的头指针和一个结点指针，定义一个函数在 O(1)时间删除该结点。链表结点与函数的定义如下：

#

代码实现：

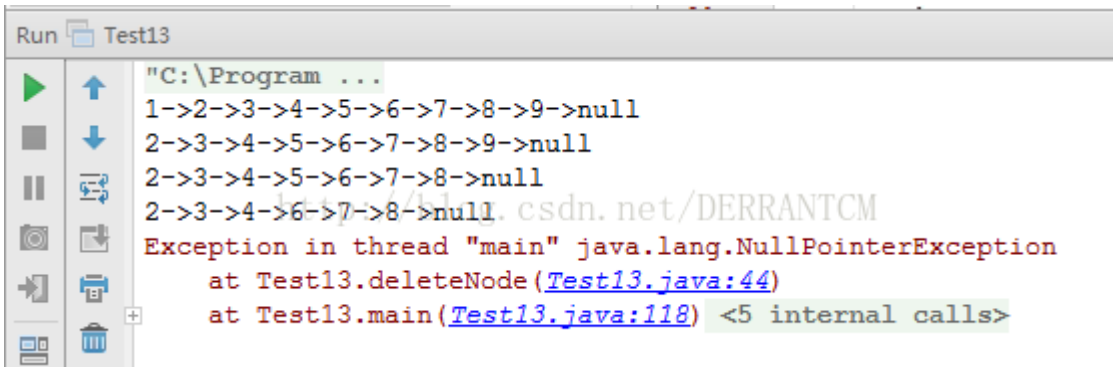
```
[java] view plaincopyprint?
public class Test13 {
    /**
     * 链表结点
     */
    public static class ListNode {
        int value; // 保存链表的值
        ListNode next; // 下一个结点
    }
    /**
     * 给定单向链表的头指针和一个结点指针，定义一个函数在0(1)时间删除该结点，
     *【注意1：这个方法和文本上的不一样，书上的没有返回值，这个因为JAVA引用传递的原因，
     * 如果删除的结点是头结点，如果不采用返回值的方式，那么头结点永远删除不了】
     *【注意2：输入的待删除结点必须是待链表中的结点，否则会引起错误，这个条件由用户进行保证】
     *
     * @param head       链表表的头
     * @param toBeDeleted 待删除的结点
     * @return 删除后的头结点
     */
    public static ListNode deleteNode(ListNode head, ListNode toBeDeleted) {
        // 如果输入参数有空值就返回表头结点
        if (head == null || toBeDeleted == null) {
            return head;
        }
        // 如果删除的是头结点，直接返回头结点的下一个结点
        if (head == toBeDeleted) {
            return head.next;
        }
          // 下面的情况链表至少有两个结点
        // 在多个节点的情况下，如果删除的是最后一个元素
        if (toBeDeleted.next == null) {
            // 找待删除元素的前驱
            ListNode tmp = head;
            while (tmp.next != toBeDeleted) {
                tmp = tmp.next;
```

```
        }
        // 删除待结点
        tmp.next = null;
    }
    // 在多个节点的情况下，如果删除的是某个中间结点
    else {
        // 将下一个结点的值输入当前待删除的结点
        toBeDeleted.value = toBeDeleted.next.value;
        // 待删除的结点的下一个指向原先待删除引号的下下个结点，即将待删除的下一个结点删除
        toBeDeleted.next = toBeDeleted.next.next;
    }
    // 返回删除节点后的链表头结点
    return head;
}
  /**
 * 输出链表的元素值
 *
 * @param head 链表的头结点
 */
public static void printList(ListNode head) {
    while (head != null) {
        System.out.print(head.value + "->");
        head = head.next;
    }
    System.out.println("null");
}
 public static void main(String[] args) {
    ListNode head = new ListNode();
    head.value = 1;
    head.next = new ListNode();
    head.next.value = 2;
    head.next.next = new ListNode();
    head.next.next.value = 3;
    head.next.next.next = new ListNode();
    head.next.next.next.value = 4;
    ListNode middle = head.next.next.next.next = new ListNode();
    head.next.next.next.next.value = 5;
    head.next.next.next.next.next = new ListNode();
    head.next.next.next.next.next.value = 6;
    head.next.next.next.next.next.next = new ListNode();
    head.next.next.next.next.next.next.value = 7;
    head.next.next.next.next.next.next.next = new ListNode();
    head.next.next.next.next.next.next.next.value = 8;
    ListNode last = head.next.next.next.next.next.next.next.next = new ListNode();
    head.next.next.next.next.next.next.next.next.value = 9;
```

```
        head = deleteNode(head, null); // 删除的结点为空
        printList(head);
        ListNode node = new ListNode();
        node.value = 12;
        head = deleteNode(head, head); // 删除头结点
        printList(head);
        head = deleteNode(head, last); // 删除尾结点
        printList(head);
        head = deleteNode(head, middle); // 删除中间结点
        printList(head);
        head = deleteNode(head, node); // 删除的结点不在链表中
        printList(head);
    }
}
```

#

运行结果：



13

调整数组顺序使奇数位于偶数前面

# #

题目:输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位予数组的后半部分。

这个题目要求把奇数放在数组的前半部分， 偶数放在数组的后半部分，因此所有的奇数应该位于偶数的前面。也就是说我们在扫描这个数组的时候， 如果发现有偶数出现在奇数的前面，我们可以交换它们的顺序，交换之后就符合要求了。

因此我们可以维护两个指针，第一个指针初始化时指向数组的第一个数字，它只向后移动：第二个指针初始化时指向数组的最后一个数字， 它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是偶数，并且第二个指针指向的数字是奇数，我们就交换这两个数字。

#

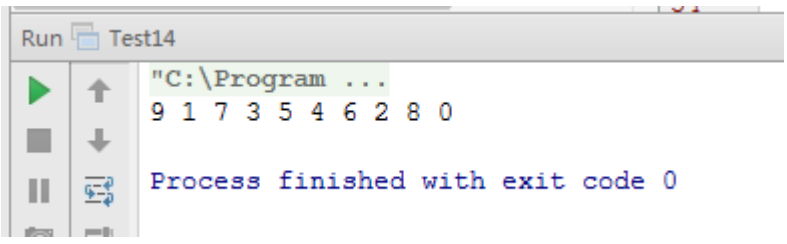
代码实现：

```
public class Test14 {
    /**
     * 输入一个整数数组，实现一个函数来调整该数组中数字的顺序，
     * 使得所有奇数位于数组的前半部分，所有偶数位予数组的后半部分。
     *
     * @param arr 输入的数组
     */
    public static void reorderOddEven(int[] arr) {
        // 对于输入的数组为空，或者长度小于2的只接返回
        if (arr == null || arr.length < 2) {
            return;
        }
        // 从左向右记录偶数的位置
        int start = 0;
        // 从右向左记录奇数的位置
        int end = arr.length - 1;
        // 开始调整奇数和偶数的位置
        while (start < end) {
            // 找偶数
            while (start < end && arr[start] % 2 != 0) {
                start++;
            }
            // 找奇数
```

```
            while (start < end && arr[end] % 2 == 0) {
                end--;
            }
            // 找到后就将奇数和偶数交换位置
            // 对于start=end的情况，交换不会产生什么影响
            // 所以将if判断省去了
            int tmp = arr[start];
            arr[start] = arr[end];
            arr[end] = tmp;
        }
    }
    /**
     * 输出数组的信息
     *
     * @param arr 待输出数组
     */
    public static void printArray(int[] arr) {
        if (arr != null && arr.length > 0) {
            for (int i : arr) {
                System.out.print(i + " ");
            }
            System.out.println();
        }
    }
    public static void main(String[] args) {
        int[] array = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        reorderOddEven(array);
        printArray(array);
    }
}
```

#

运行结果：



14

链表中倒数第 k 个结点

# #

题目：输入一个链表，输出该链表中倒数第k 个结点．为了符合大多数人的习惯，本题从 1 开始计数，即链表的尾结点是倒数第 1 个结点．例如一个链表有 6 个结点，从头结点开始它们的值依次是 1 、2、3、4、5 、6。这个个链表的倒数第 3 个结点是值为 4 的结点。

#

链表结点定义如下：

```
public static class ListNode {
    int value;
    ListNode next;
}
```

#

解题思路：

为了实现只遍历链表一次就能找到倒数第 k 个结点，我们可以定义两 个指针。第一个指针从链表的头指针开始遍历向前走 k−1步，第二个指针保持不动；从第k 步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在 k−1 ， 当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第 k 个结点。

#

代码实现：

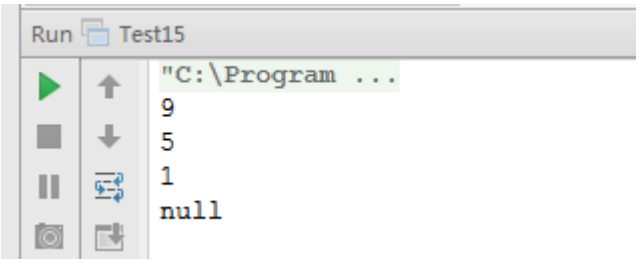
```
public class Test15 {
    public static class ListNode {
        int value;
        ListNode next;
    }
    /**
     * 输入一个键表，输出该链表中倒数第k 个结点．为了符合大多数人的习惯，
     * 本题从1开始计数，即链表的尾结点是倒数第1个结点．例如一个链表有6个结点，
     * 从头结点开始它们的值依次是1、2、3、4、5 6。这个链表的倒数第3个结点是值为4的结点.
```

```
     *
     * @param head 链表的头结点
     * @param k    倒数第k个结点
     * @return 倒数第k个结点
     */
    public static ListNode findKthToTail(ListNode head, int k) {
        // 输入的链表不能为空，并且k大于0
        if (k < 1 || head == null) {
            return null;
        }
        // 指向头结点
        ListNode pointer = head;
        // 倒数第k个结点与倒数第一个结点相隔k-1个位置
        // pointer先走k-1个位置
        for (int i = 1; i < k; i++) {
            // 说明还有结点
            if (pointer.next != null) {
                pointer = pointer.next;
            }
            // 已经没有节点了，但是i还没有到达k-1说明k太大，链表中没有那么多的元素
            else {
                // 返回结果
                return null;
            }
        }
        // pointer还没有走到链表的末尾，那么pointer和head一起走，
        // 当pointer走到最后一个结点即，pointer.next=null时，head就是倒数第k个结点
        while (pointer.next != null) {
            head = head.next;
            pointer = pointer.next;
        }
        // 返回结果
        return head;
    }
    public static void main(String[] args) {
        ListNode head = new ListNode();
        head.value = 1;
        head.next = new ListNode();
        head.next.value = 2;
        head.next.next = new ListNode();
        head.next.next.value = 3;
        head.next.next.next = new ListNode();
        head.next.next.next.value = 4;
        head.next.next.next.next = new ListNode();
        head.next.next.next.next.value = 5;
```

```
        head.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.value = 6;
        head.next.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.next.value = 7;
        head.next.next.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.next.next.value = 8;
        head.next.next.next.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.next.next.next.value = 9;
        System.out.println(findKthToTail(head, 1).value); // 倒数第一个
        System.out.println(findKthToTail(head, 5).value); // 中间的一个
        System.out.println(findKthToTail(head, 9).value); // 倒数最后一个就是顺数第一个
        System.out.println(findKthToTail(head, 10));
    }
}
```

#

运行结果：



15

反转链表

# #

题目：定义一个函数，输入一个链表的头结点，反转该链表并输出反转后链表的头结点。

#

链表结点定义如下：

```
public static class ListNode {
    int value;
    ListNode next;
}
```

#

解题思路：

在单链表的表头临时接入一个节点，然后进行尾插法操作。反转单链表。

#

代码实现：

```
public class Test16 {
    public static class ListNode {
        int value;
        ListNode next;
    }
    /**
     * 定义一个函数，输入一个链表的头结点，反转该链表并输出反转后链表的头结点。
     *
     * @param head 链表的头结点
     * @return 反转后的链表的头结点
     */
    public static ListNode reverseList(ListNode head) {
        // 创建一个临时结点，当作尾插法的逻辑头结点
        ListNode root = new ListNode();
        // 逻辑头结点点的下一个结点为空
```

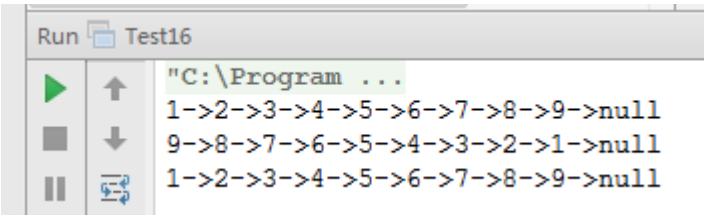
```
    root.next = null;
    // 用于记录要处理的下一个结点
    ListNode next;
    // 当前处理的结点不为空
    while (head != null) {
        // 记录要处理的下一个结点
        next = head.next;
        // 当前结点的下一个结点指向逻辑头结点的下一个结点
        head.next = root.next;
        // 逻辑头结点的下一个结点指向当前处理的结点
        root.next = head;
        // 上面操作完成了一个结点的头插
        // 当前结点指向下一个要处理的结点
        head = next;
    }
    // 逻辑头结点的下一个结点就是返回后的头结点
    return root.next;
}
/**
 * 定义一个函数，输入一个链表的头结点，反转该链表并输出反转后链表的头结点。
 *【书本上的方法，不使用逻辑头结点】
 *
 * @param head 链表的头结点
 * @return 反转后的链表的头结点
 */
public static ListNode reverseList2(ListNode head) {
    // 用于记录反转后的链表的头结点
    ListNode reverseHead = null;
    // 用于记录当前处理的结点的
    ListNode curr = head;
    // 用于记录当前结点的前驱结点
    // 前驱结点开始为null，因为了是反转后的最后一个结点的下一个结点，即null
    ListNode prev = null;
    // 当前结点的下一个结点
    ListNode next;
    // 对链表进行尾插法操作
    while (curr != null) {
        // 记录当前处理的结点，最后一个记录的结点就是反转后的头结点
        //【注意：与书上的不同，因为curr.next=null时，curr此时就最后一个处理的结点，
        // 对应到反转后的链表就是第一个结点，书上那样做更精确，只是多了一些判断，可以不要if】
        reverseHead = curr;
        // 记录当然前下一个结点
        next = curr.next;
        // 当前结点的下一个结点指向前驱结点，这样当前结点就插入到了反转链表的头部
        curr.next = prev;
```

```
            // 记录当前结点为前驱结点
            prev = curr;
            // 当前结点点移动到下一个结点
            curr = next;
        }
        // 返回转后的头结点
        return reverseHead;
    }
    /**
     * 输出链表的元素值
     *
     * @param head 链表的头结点
     */
    public static void printList(ListNode head) {
        while (head != null) {
            System.out.print(head.value + "->");
            head = head.next;
        }
        System.out.println("null");
    }
    public static void main(String[] args) {
        ListNode head = new ListNode();
        head.value = 1;
        head.next = new ListNode();
        head.next.value = 2;
        head.next.next = new ListNode();
        head.next.next.value = 3;
        head.next.next.next = new ListNode();
        head.next.next.next.value = 4;
        head.next.next.next.next = new ListNode();
        head.next.next.next.next.value = 5;
        head.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.value = 6;
        head.next.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.next.value = 7;
        head.next.next.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.next.next.value = 8;
        head.next.next.next.next.next.next.next.next = new ListNode();
        head.next.next.next.next.next.next.next.next.value = 9;
        printList(head);
        head = reverseList(head);
        printList(head);
        head = reverseList2(head);
        printList(head);
```

```
    }
}
```

#

运行结果：


Run
Test16
"C:\Program ...
1->2->3->4->5->6->7->8->9->null
9->8->7->6->5->4->3->2->1->null
1->2->3->4->5->6->7->8->9->null

16

合并两个排序的链表

# #

题目：输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递增排序的

#

链表结点定义如下：

```
public static class ListNode {
    int value;
    ListNode next;
}
```

#

解题思路：

见代码注释

#

代码实现：

```
public class Test17 {
    public static class ListNode {
        int value;
        ListNode next;
    }
    /**
     * 输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递增排序的
     *
     * @param head1 第一个有序链表
     * @param head2 第二个有序链表
     * @return 合并后的有序链表头
     */
    public static ListNode merge(ListNode head1, ListNode head2) {
        // 如果第一个链表为空，返回第二个链表头结点
        if (head1 == null) {
```

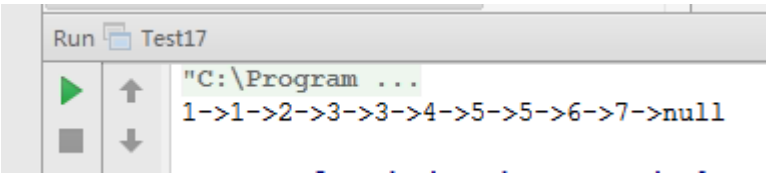
```
        return head2;
    }
    // 如果第二个结点为空，返回第一个链表头结点
    if (head2 == null) {
        return head1;
    }
    // 创建一个临时结点，用于添加元素时方便
    ListNode root = new ListNode();
    // 用于指向合并后的新链的尾结点
    ListNode pointer = root;
    // 当两个链表都不为空就进行合并操作
    while (head1 != null && head2 != null) {
        // 下面的操作合并较小的元素
        if (head1.value < head2.value) {
            pointer.next = head1;
            head1 = head1.next;
        } else {
            pointer.next = head2;
            head2 = head2.next;
        }
        // 将指针移动到合并后的链表的末尾
        pointer = pointer.next;
    }
    // 下面的两个if有且只一个if会内的内容会执行
    // 如果第一个链表的元素未处理完将其，接到合并链表的最后一个结点之后
    if (head1 != null) {
        pointer.next = head1;
    }
    // 如果第二个链表的元素未处理完将其，接到合并链表的最后一个结点之后
    if (head2 != null) {
        pointer.next = head2;
    }
    // 返回处理结果
    return root.next;
}
/**
 * 输入两个递增排序的链表，合并这两个链表并使新链表中的结点仍然是按照递增排序的
 *【使用的是递归的解法，不推荐，递归调用的时候会有方法入栈，需要更多的内存】
 *
 * @param head1 第一个有序链表
 * @param head2 第二个有序链表
 * @return 合并后的有序链表头
 */
public static ListNode merge2(ListNode head1, ListNode head2) {
    // 如果第一个链表为空，返回第二个链表头结点
```

```
    if (head1 == null) {
        return head2;
    }
    // 如果第二个链表为空，返回第一个链表头结点
    if (head2 == null) {
        return head1;
    }
    // 记录两个链表中头部较小的结点
    ListNode tmp = head1;
    if (tmp.value < head2.value) {
        // 如果第一个链表的头结点小，就递归处理第一个链表的下一个结点和第二个链表的头结点
        tmp.next = merge2(head1.next, head2);
    } else {
        // 如果第二个链表的头结点小，就递归处理第一个链表的头结点和第二个链表的头结点的下一个结点
        tmp = head2;
        tmp.next = merge2(head1, head2.next);
    }
    // 返回处理结果
    return tmp;
}
/**
 * 输出链表的元素值
 *
 * @param head 链表的头结点
 */
public static void printList(ListNode head) {
    while (head != null) {
        System.out.print(head.value + "->");
        head = head.next;
    }
    System.out.println("null");
}
public static void main(String[] args) {
    ListNode head = new ListNode();
    head.value = 1;
    head.next = new ListNode();
    head.next.value = 2;
    head.next.next = new ListNode();
    head.next.next.value = 3;
    head.next.next.next = new ListNode();
    head.next.next.next.value = 4;
    head.next.next.next.next = new ListNode();
    head.next.next.next.next.value = 5;
    ListNode head2 = new ListNode();
    head2.value = 1;
```

```
        head2.next = new ListNode();
        head2.next.value = 3;
        head2.next.next = new ListNode();
        head2.next.next.value = 5;
        head2.next.next.next = new ListNode();
        head2.next.next.next.value = 6;
        head2.next.next.next.next = new ListNode();
        head2.next.next.next.next.value = 7;
//        head = merge(head, head2);
        head = merge2(head, head2);
        printList(head);
    }
```

#

运行结果：



17

树的子结构

# #

题目：输入两棵二叉树 A 和 B，判断 B 是不是 A 的子结构。

#

二叉树结点的定义：

```
/**
 * 二叉树的树结点
 */
 public static class BinaryTreeNode {
    int value;
    BinaryTreeNode left;
    BinaryTreeNode right;
}
```

#

解题思路：

要查找树 A 中是否存在和树 B 结构一样的子树，我们可以分成两步： 第一步在树 A 中找到和 B 的根结点的值一样的结点 R， 第二步再判断树 A 中以 R 为根结点的子树是不是包含和树 B 一样的结构。

#

代码实现

```
public class Test18 {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }
    /**
```

```
 * 输入两棵二叉树A和B，判断B是不是A的子结构。
 * 该方法是在A树中找到一个与B树的根节点相等的元素的结点，
 * 从这个相等的结点开始判断树B是不是树A的子结构，如果找到其的一个就返回，
 * 否则直到所有的结点都找完为止。
 *
 * @param root1 树A的根结点
 * @param root2 树B的根结点
 * @return true：树B是树A的子结构，false：树B是不树A的子结构
 */
public static boolean hasSubtree(BinaryTreeNode root1, BinaryTreeNode root2) {
    // 只要两个对象是同一个就返回true
    //【注意此处与书本上的不同，书本上的没有这一步】
    if (root1 == root2) {
        return true;
    }
    // 只要树B的根结点点为空就返回true
    if (root2 == null) {
        return true;
    }
    // 树B的根结点不为空，如果树A的根结点为空就返回false
    if (root1 == null) {
        return false;
    }
    // 记录匹配结果
    boolean result = false;
    // 如果结点的值相等就，调用匹配方法
    if (root1.value == root2.value) {
        result = match(root1, root2);
    }
    // 如果匹配就直接返回结果
    if (result) {
        return true;
    }
    // 如果不匹配就找树A的左子结点和右子结点进行判断
    return hasSubtree(root1.left, root2) || hasSubtree(root1.right, root2);
}
/**
 * 从树A根结点root1和树B根结点root2开始，一个一个元素进行判断，判断B是不是A的子结构
 *
 * @param root1 树A开始匹配的根结点
 * @param root2 树B开始匹配的根结点
 * @return 树B是树A的子结构，false：树B是不树A的子结构
 */
public static boolean match(BinaryTreeNode root1, BinaryTreeNode root2) {
    // 只要两个对象是同一个就返回true
```

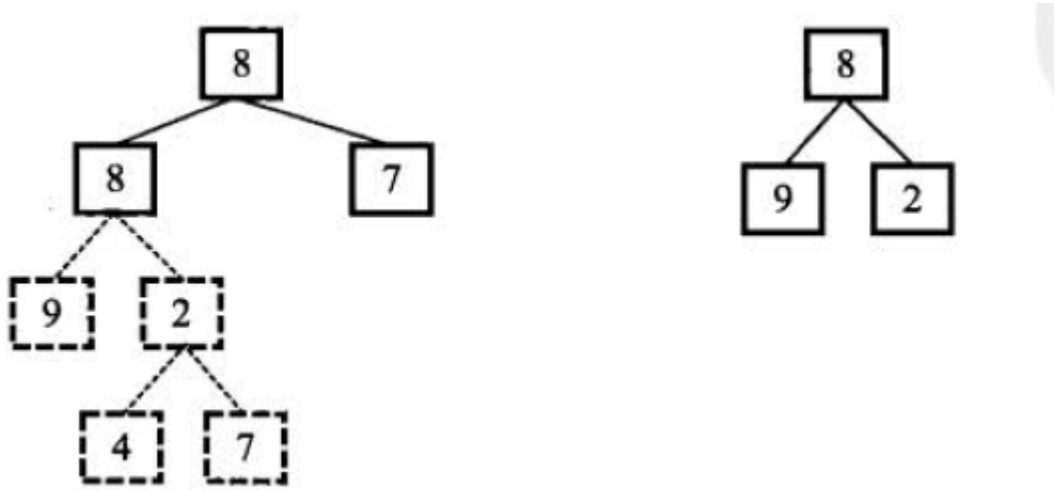
```
    if (root1 == root2) {
        return true;
    }
    // 只要树B的根结点点为空就返回true
    if (root2 == null) {
        return true;
    }
    // 树B的根结点不为空，如果树A的根结点为空就返回false
    if (root1 == null) {
        return false;
    }
    // 如果两个结点的值相等，则分别判断其左子结点和右子结点
    if (root1.value == root2.value) {
        return match(root1.left, root2.left) && match(root1.right, root2.right);
    }
    // 结点值不相等返回false
    return false;
}
public static void main(String[] args) {
    BinaryTreeNode root1 = new BinaryTreeNode();
    root1.value = 8;
    root1.right = new BinaryTreeNode();
    root1.right.value = 7;
    root1.left = new BinaryTreeNode();
    root1.left.value = 8;
    root1.left.left = new BinaryTreeNode();
    root1.left.left.value = 9;
    root1.left.right = new BinaryTreeNode();
    root1.left.right.value = 2;
    root1.left.right.left = new BinaryTreeNode();
    root1.left.right.left.left = new BinaryTreeNode();
    root1.left.right.left.left.value = 4;
    root1.left.right.left.right = new BinaryTreeNode();
    root1.left.right.left.right.value = 7;
    BinaryTreeNode root2 = new BinaryTreeNode();
    root2.value = 8;
    root2.left = new BinaryTreeNode();
    root2.left.value = 9;
    root2.right = new BinaryTreeNode();
    root2.right.value = 2;
    System.out.println(hasSubtree(root1, root2));
    System.out.println(hasSubtree(root2, root1));
    System.out.println(hasSubtree(root1, root1.left));
    System.out.println(hasSubtree(root1, null));
    System.out.println(hasSubtree(null, root2));
```

```
        System.out.println(hasSubtree(null, null));
    }
}
```

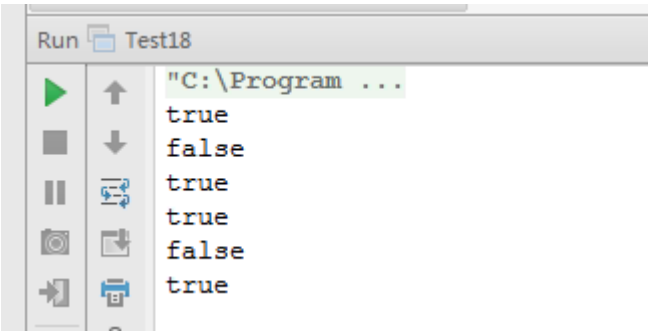
#

运行结果

输入的树：



输出结果：



18

二叉树的镜像

# #

题目：请完成一个函数，输入一个二叉树，该函数输出它的镜像。

#

二叉树结点的定义：

```
/**
 * 二叉树的树结点
 */
public static class BinaryTreeNode {
    int value;
    BinaryTreeNode left;
    BinaryTreeNode right;
}
```

#

解题思路：

我们先前序遍历这棵树的每个结点，如果遍历到的结点有子结点，就交换它的两个子结点。当交换完所有非叶子结点的左右子结点之后，就得到了树的镜像。

#

代码实现：

```
public class Test19 {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }
    /**
```

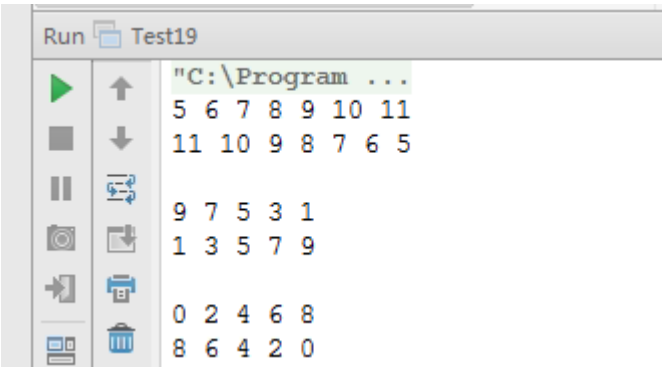
```
 * 请完成一个函数，输入…个二叉树，该函数输出它的镜像
 *
 * @param node 二叉树的根结点
 */
public static void mirror(BinaryTreeNode node) {
    // 如果当前结点不为空则进行操作
    if (node != null) {
        // 下面是交换结点左右两个子树
        BinaryTreeNode tmp = node.left;
        node.left = node.right;
        node.right = tmp;
        // 对结点的左右两个子树进行处理
        mirror(node.left);
        mirror(node.right);
    }
}
public static void printTree(BinaryTreeNode node) {
    if (node != null) {
        printTree(node.left);
        System.out.print(node.value + " ");
        printTree(node.right);
    }
}
public static void main(String[] args) {
    //       8
    //    /    \
    //   6     10
    //  / \    / \
    // 5   7 9  11
    BinaryTreeNode root = new BinaryTreeNode();
    root.value = 8;
    root.left = new BinaryTreeNode();
    root.left.value = 6;
    root.left.left = new BinaryTreeNode();
    root.left.left.value = 5;
    root.left.right = new BinaryTreeNode();
    root.left.right.value = 7;
    root.right = new BinaryTreeNode();
    root.right.value = 10;
    root.right.left = new BinaryTreeNode();
    root.right.left.value = 9;
    root.right.right = new BinaryTreeNode();
    root.right.right.value = 11;
    printTree(root);
    System.out.println();
```

```
mirror(root);
printTree(root);
//        1
//       /
//      3
//     /
//    5
//   /
//  7
// /
// 9
BinaryTreeNode root2 = new BinaryTreeNode();
root2.value = 1;
root2.left = new BinaryTreeNode();
root2.left.value = 3;
root2.left.left = new BinaryTreeNode();
root2.left.left.value = 5;
root2.left.left.left = new BinaryTreeNode();
root2.left.left.left.value = 7;
root2.left.left.left.left = new BinaryTreeNode();
root2.left.left.left.left.value = 9;
System.out.println("\n");
printTree(root2);
System.out.println();
mirror(root2);
printTree(root2);
// 0
//  \
//   2
//    \
//     4
//      \
//       6
//        \
//         8
BinaryTreeNode root3 = new BinaryTreeNode();
root3.value = 0;
root3.right = new BinaryTreeNode();
root3.right.value = 2;
root3.right.right = new BinaryTreeNode();
root3.right.right.value = 4;
root3.right.right.right = new BinaryTreeNode();
root3.right.right.right.value = 6;
root3.right.right.right.right = new BinaryTreeNode();
root3.right.right.right.right.value = 8;
```

```
        System.out.println("\n");
        printTree(root3);
        System.out.println();
        mirror(root3);
        printTree(root3);
    }
}
```

#

运行结果：



19

顺时针打印矩阵

# #

题目：输入一个矩阵，按照从外向里以顺时针的顺序依次扫印出每一个数字

#

解题思路：

把打印一圈分为四步：第一步从左到右打印一行，第二步从上到下打印一列，第三步从右到左打印一行，第四步从下到上打印一列。每一步我们根据起始坐标和终止坐标用一个循环就能打印出一行或者一列。

不过值得注意的是，最后一圈有可能退化成只有一行、只有一列，甚至只有一个数字，因此打印这样的一圈就不再需要四步。图 4.4 是几个退化的例子， 打印一圈分别只需要三步、两步甚至只有一步。

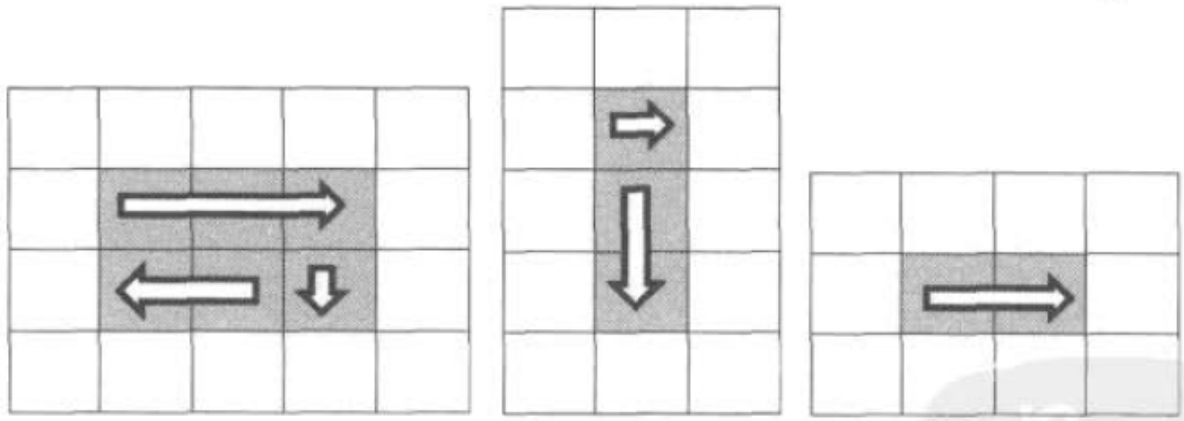


图 4.4　打印矩阵最里面一圈可能只需要三步、两步甚至一步

因此我们要仔细分析打印时每一步的前提条件。第一步总是需要的， 因为打印一圈至少有一步。如果只有一行，那么就不用第二步了。也就是需要 第二步的前提条件是终止行号大于起始行号。需要第三步打印的前提条件是圈内至少有两行两列，也就是说除了要求终止行号大于起始行号之外，还要求终止列号大于起始列号。同理，需要打印第四步的前提条件是至少有三行两列，因此要求终止行号比起始行号至少大 2，同时终止列号大于起始列号。

#

代码实现：

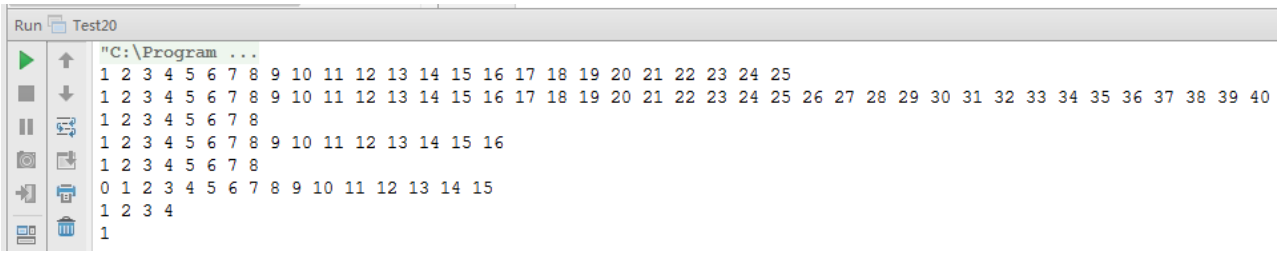
```
public class Test20 {
    /**
     * 输入一个矩阵，按照从外向里以顺时针的顺序依次打印每一个数字
     *
     * @param numbers 输入的二维数组，二维数组必须是N*M的，否则分出错
     */
    public static void printMatrixClockWisely(int[][] numbers) {
        // 输入的参数不能为空
        if (numbers == null) {
            return;
        }
        // 记录一圈（环）的开始位置的行
        int x = 0;
        // 记录一圈（环）的开始位置的列
        int y = 0;
        // 对每一圈（环）进行处理，
        // 行号最大是(numbers.length-1)/2
        // 列号最大是(numbers[0].length-1)/2
        while (x * 2 < numbers.length && y * 2 < numbers[0].length) {
            printMatrixInCircle(numbers, x, y);
            // 指向下一个要处理的的环的第一个位置
            x++;
            y++;
        }
    }
    public static void printMatrixInCircle(int[][] numbers, int x, int y) {
        // 数组的行数
        int rows = numbers.length;
        // 数组的列数
        int cols = numbers[0].length;
        // 输出环的上面一行，包括最中的那个数字
        for (int i = y; i <= cols - y - 1; i++) {
            System.out.print(numbers[x][i] + " ");
        }
        // 环的高度至少为2才会输出右边的一列
        // rows-x-1：表示的是环最下的那一行的行号
        if (rows - x - 1 > x) {
            // 因为右边那一列的最上面那一个已经被输出了，所以行呈从x+1开始，
            // 输出包括右边那列的最下面那个
            for (int i = x + 1; i <= rows - x - 1; i++) {
                System.out.print(numbers[i][cols - y - 1] + " ");
            }
        }
        // 环的高度至少是2并且环的宽度至少是2才会输出下面那一行
        // cols-1-y：表示的是环最右那一列的列号
```

```
    if (rows - x - 1 > x && cols - 1 - y > y) {
        // 因为环的左下角的位置已经输出了，所以列号从cols-y-2开始
        for (int i = cols - y - 2; i >= y; i--) {
            System.out.print(numbers[rows - 1 - x][i] + " ");
        }
    }
    // 环的宽度至少是2并且环的高度至少是3才会输出最左边那一列
    // rows-x-1：表示的是环最下的那一行的行号
    if (cols - 1 - y > y && rows - 1 - x > x + 1) {
        // 因为最左边那一列的第一个和最后一个已经被输出了
        for (int i = rows - 1 - x - 1; i >= x + 1; i--) {
            System.out.print(numbers[i][y] + " ");
        }
    }
}
public static void main(String[] args) {
    int[][] numbers = {
            {1, 2, 3, 4, 5},
            {16, 17, 18, 19, 6},
            {15, 24, 25, 20, 7},
            {14, 23, 22, 21, 8},
            {13, 12, 11, 10, 9},
    };
    printMatrixClockWisely(numbers);
    System.out.println();
    int[][] numbers2 = {
            {1, 2, 3, 4, 5, 6, 7, 8},
            {22, 23, 24, 25, 26, 27, 28, 9},
            {21, 36, 37, 38, 39, 40, 29, 10},
            {20, 35, 34, 33, 32, 31, 30, 11},
            {19, 18, 17, 16, 15, 14, 13, 12},
    };
    printMatrixClockWisely(numbers2);
    System.out.println();
    int[][] numbers3 = {
            {1, 2, 3, 4, 5, 6, 7, 8}
    };
    printMatrixClockWisely(numbers3);
    System.out.println();
    int[][] numbers4 = {
            {1, 2, 3, 4, 5, 6, 7, 8},
            {16, 15, 14, 13, 12, 11, 10, 9}
    };
    printMatrixClockWisely(numbers4);
    System.out.println();
```

```
    int[][] numbers5 = {
            {1},
            {2},
            {3},
            {4},
            {5},
            {6},
            {7},
            {8}
    };
    printMatrixClockWisely(numbers5);
    System.out.println();
    int[][] numbers6 = {
            {0, 1},
            {15, 2},
            {14, 3},
            {13, 4},
            {12, 5},
            {11, 6},
            {10, 7},
            {9, 8}
    };
    printMatrixClockWisely(numbers6);
    System.out.println();
    int[][] numbers7 = {
            {1, 2},
            {4, 3}
    };
    printMatrixClockWisely(numbers7);
    System.out.println();
    int[][] numbers8 = {
            {1}
    };
    printMatrixClockWisely(numbers8);
    System.out.println();
    // 0个元素的数组
    printMatrixClockWisely(new int[][]{{}});
    // 空数组
    printMatrixClockWisely(null);
}
```

#

运行结果


Run
Test20
"C:\Program ...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 2 3 4
1

20

包含 min 函数的钱

# #

题目： 定义栈的数据结构，请在该类型中实现一个能够得到栈的最小素的 min 函数。在该栈中，调用 min、push 及 pop 的时间复杂度都是 0(1)

#

解题思路：

把每次的最小元素（之前的最小元素和新压入战的元素两者的较小值）都保存起来放到另外一个辅助栈里。

如果每次都把最小元素压入辅助栈， 那么就能保证辅助栈的栈顶一直都是最小元素. 当最小元素从数据栈内被弹出之后，同时弹出辅助栈的栈顶元素，此时辅助栈的新栈顶元素就是下一个最小值。

#

代码实现：

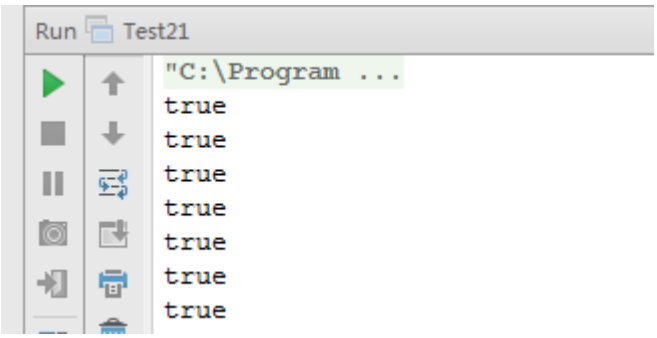
```
public class Test21 {
    /**
     * 定义栈的数据结构，请在该类型中实现一个能够得到校的最小元素的min函数。
     * 在该栈中，调用pop、push 及min的时间复杂度都是0(1)
     *
     * @param <T> 泛型参数
     */
    public static class StackWithMin<T extends Comparable<T>> {
        // 数据栈，用于存放插入的数据
        private Stack<T> dataStack;
        // 最小数位置栈，存放数据栈中最小的数的位置
        private Stack<Integer> minStack;
        // 构造函数
        public StackWithMin() {
            this.dataStack = new Stack<>();
            this.minStack = new Stack<>();
        }
        /**
         * 出栈方法
         * @return 栈顶元素
         */
```

```
public T pop() {
    // 如果栈已经为空，再出栈抛出异常
    if (dataStack.isEmpty()) {
        throw new RuntimeException("The stack is already empty");
    }
    // 如果有数据，最小数位置栈和数据栈必定是有相同的元素个数，
    // 两个栈同时出栈
    minStack.pop();
    return dataStack.pop();
}
/**
 * 元素入栈
 * @param t 入栈的元素
 */
public void push(T t) {
    // 如果入栈的元素为空就抛出异常
    if (t == null) {
        throw new RuntimeException("Element can be null");
    }
    // 如果数据栈是空的，只接将元素入栈，同时更新最小数栈中的数据
    if (dataStack.isEmpty()) {
        dataStack.push(t);
        minStack.push(0);
    }
    // 如果数据栈中有数据
    else {
        // 获取数据栈中的最小元素（未插入t之前的）
        T e = dataStack.get(minStack.peek());
        // 将t入栈
        dataStack.push(t);
        // 如果插入的数据比栈中的最小元素小
        if (t.compareTo(e) < 0) {
            // 将新的最小元素的位置入最小栈
            minStack.push(dataStack.size() - 1);
        } else {
            // 插入的元素不比原来的最小元素小，复制最小栈栈顶元素，将其入栈
            minStack.push(minStack.peek());
        }
    }
}
/**
 * 获取栈中的最小元素
 * @return 栈中的最小元素
 */
public T min() {
```

```
            // 如果最小数公位置栈已经为空（数据栈中已经没有数据了），则抛出异常
            if (minStack.isEmpty()) {
                throw new RuntimeException("No element in stack.");
            }
            // 获取数据栈中的最小元素，并且返回结果
            return dataStack.get(minStack.peek());
        }
    }
    public static void main(String[] args) {
        StackWithMin<Integer> stack = new StackWithMin<>();
        stack.push(3);
        System.out.println(stack.min() == 3);
        stack.push(4);
        System.out.println(stack.min() == 3);
        stack.push(2);
        System.out.println(stack.min() == 2);
        stack.push(3);
        System.out.println(stack.min() == 2);
        stack.pop();
        System.out.println(stack.min() == 2);
        stack.pop();
        System.out.println(stack.min() == 3);
        stack.push(0);
        System.out.println(stack.min() == 0);
    }
}
```

#

运行结果：



21

栈的压入、弹出序列

# #

题目：输入两个整数序列，第一个序列表示栈的压入顺序，请判断二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。

#

解题思路：

解决这个问题很直观的想法就是建立一个辅助栈，把输入的第一个序列中的数字依次压入该辅助栈，并按照第二个序列的顺序依次从该栈中弹出数字。

判断一个序列是不是栈的弹出序列的规律：如果下一个弹出的数字刚好是栈顶数字，那么直接弹出。如果下一个弹出的数字不在栈顶，我们把压栈序列中还没有入栈的数字压入辅助栈，直到把下一个需要弹出的数字压入栈顶为止。如果所有的数字都压入栈了仍然没有找到下一个弹出的数字，那么该序列不可能是一个弹出序列。

#

代码实现：

```
public class Test22 {
    /**
     * 输入两个整数序列，第一个序列表示栈的压入顺序，请判断二个序列是否为该栈的弹出顺序。
     * 假设压入栈的所有数字均不相等。例如序列1 、2、3 、4、5 是某栈压栈序列，
     * 序列4、5、3、2、1是该压栈序列对应的一个弹出序列，
     * 但4、3、5、1、2就不可能是该压棋序列的弹出序列。
     * 【 与书本的的方法不同 】
     *
     * @param push 入栈序列
     * @param pop  出栈序列
     * @return true：出栈序列是入栈序列的一个弹出顺序
     */
    public static boolean isPopOrder(int[] push, int[] pop) {
        // 输入校验，参数不能为空，并且两个数组中必须有数字，并且两个数组中的数字个数相同
        // 否则返回false
        if (push == null || pop == null || pop.length == 0 || push.length == 0 || push.length != pop.length) {
            return false;
        }
        // 经过上面的参数校验，两个数组中一定有数据，且数据数目相等
```

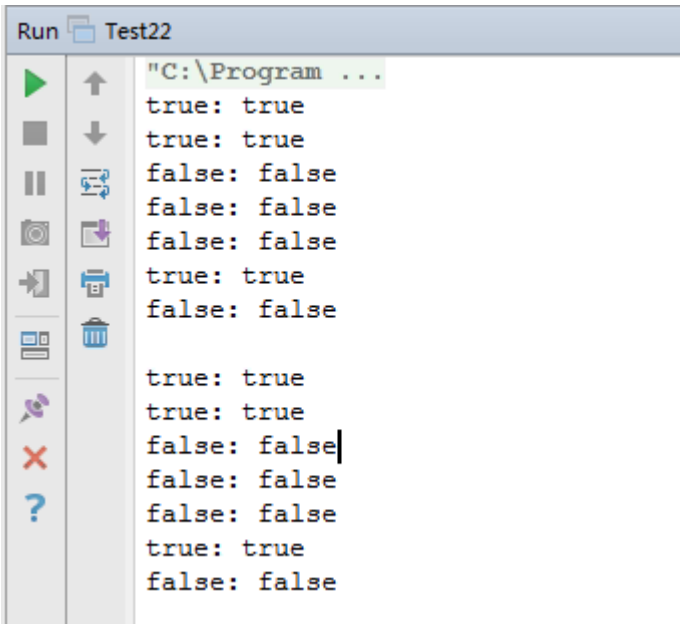
```
    // 用于存放入栈时的数据
    Stack<Integer> stack = new Stack<>();
    // 用于记录入栈数组元素的处理位置
    int pushIndex = 0;
    // 用于记录出栈数组元素的处理位置
    int popIndex = 0;
    // 如果还有出栈元素要处理
    while (popIndex < pop.length) {
        // 入栈元素还未全部入栈的条件下，如果栈为空，或者栈顶的元素不与当前处理的相等，则一直进行栈操作，
        // 直到入栈元素全部入栈或者找到了一个与当出栈元素相等的元素
        while (pushIndex < push.length && (stack.isEmpty() || stack.peek() != pop[popIndex])) {
            // 入栈数组中的元素入栈
            stack.push(push[pushIndex]);
            // 指向下一个要处理的入栈元素
            pushIndex++;
        }
        // 如果在上一步的入栈过程中找到了与出栈的元素相等的元素
        if (stack.peek() == pop[popIndex]) {
            // 将元素出栈
            stack.pop();
            // 处理下一个出栈元素
            popIndex++;
        }
        // 如果没有找到与出栈元素相等的元素，说明这个出栈顺序是不合法的
        // 就返回false
        else {
            return false;
        }
    }
    // 下面的语句总是成立的
    // return stack.isEmpty();
    // 为什么可以直接返回true：对上面的外层while进行分析可知道，对每一个入栈的元素，
    // 在stack栈中，通过一些入栈操作，总可以在栈顶上找到与入栈元素值相同的元素，
    // 这就说明了这个出栈的顺序是入栈顺序的一个弹出队列，这也可以解释为什么stack.isEmpty()
    // 总是返回true，所有的入栈元素都可以进栈，并且可以被匹配到，之后就弹出，最后栈中就无元素。
    return true;
}
/**
 * 输入两个整数序列，第一个序列表示栈的压入顺序，请判断二个序列是否为该栈的弹出顺序。
 * 【按书本上的思路进行求解，两者相差不大】
 *
 * @param push 入栈序列
 * @param pop  出栈序列
 * @return true：出栈序列是入栈序列的一个弹出顺序
 */
```

```
public static boolean isPopOrder2(int[] push, int[] pop) {
    // 用于记录判断出栈顺序是不是入栈顺的一个出栈序列，默认false
    boolean isPossible = false;
    // 当入栈和出栈数组者都不为空，并且都有数据，并且数据个数都相等
    if (push != null && pop != null && push.length > 0 && push.length == pop.length) {
        // 用于存放入栈时的数据
        Stack<Integer> stack = new Stack<>();
        // 记录下一个要处理的入栈元素的位置
        int nextPush = 0;
        // 记录下一个要处理的出栈元素的位置
        int nextPop = 0;
        // 如果出栈元素没有处理完就继续进行处理
        while (nextPop < pop.length) {
            // 如果栈为空或者栈顶的元素与当前处理的出栈元素不相同，一直进行操作
            while (stack.isEmpty() || stack.peek() != pop[nextPop]) {
                // 如果入栈的元素已经全部入栈了，就退出内层循环
                if (nextPush >= push.length) {
                    break;
                }
                // 执行到此处说明还有入栈元素可以入栈
                // 即将元素入栈
                stack.push(push[nextPush]);
                // 指向下一个要处理的入栈元素的位置
                nextPush++;
            }
            // 执行到此处有两种情况：
            // 第一种：在栈顶上找到了一个与入栈元素相等的元素
            // 第二种：在栈顶上没有找到一个与入栈元素相等的元素，而且输入栈的元素已经全部入栈了
            // 对于第二种情况就说弹出栈的顺序是不符合要求的，退出外层循环
            if (stack.peek() != pop[nextPop]) {
                break;
            }
            // 对应到第一种情况：需要要栈的栈顶元素弹出
            stack.pop();
            // 指向下一个要处理的出栈元素的位置
            nextPop++;
        }
        // 执行到此处有两种情况
        // 第一种：外层while循环的在第一种情况下退出，
        // 第二种：所有的出栈元素都被正确匹配
        // 对于出现的第一种情况其stack.isEmpty()必不为空，原因为分析如下：
        // 所有的入栈元素一定会入栈，但是只有匹配的情况下才会出栈，
        // 匹配的次数最多与入栈元素个数元素相同（两个数组的长度相等），如果有不匹配的元素，
        // 必然会使出栈的次数比入栈的次数少，这样栈中至少会有一个元素
        // 对于第二种情况其stack.isEmpty()一定为空
```

```
            // 所以书本上的nextPop == pop.length（pNextPop-pPop==nLength）是多余的
            if (stack.isEmpty()) {
                isPossible = true;
            }
        }
        return isPossible;
    }
    public static void main(String[] args) {
        int[] push = {1, 2, 3, 4, 5};
        int[] pop1 = {4, 5, 3, 2, 1};
        int[] pop2 = {3, 5, 4, 2, 1};
        int[] pop3 = {4, 3, 5, 1, 2};
        int[] pop4 = {3, 5, 4, 1, 2};
        System.out.println("true: " + isPopOrder(push, pop1));
        System.out.println("true: " + isPopOrder(push, pop2));
        System.out.println("false: " + isPopOrder(push, pop3));
        System.out.println("false: " + isPopOrder(push, pop4));
        int[] push5 = {1};
        int[] pop5 = {2};
        System.out.println("false: " + isPopOrder(push5, pop5));
        int[] push6 = {1};
        int[] pop6 = {1};
        System.out.println("true: " + isPopOrder(push6, pop6));
        System.out.println("false: " + isPopOrder(null, null));
        // 测试方法2
        System.out.println();
        System.out.println("true: " + isPopOrder2(push, pop1));
        System.out.println("true: " + isPopOrder2(push, pop2));
        System.out.println("false: " + isPopOrder2(push, pop3));
        System.out.println("false: " + isPopOrder2(push, pop4));
        System.out.println("false: " + isPopOrder2(push5, pop5));
        System.out.println("true: " + isPopOrder2(push6, pop6));
        System.out.println("false: " + isPopOrder2(null, null));
    }
}
```

#

运行结果：


Run
Test22
"C:\Program ...
true: true
true: true
false: false
false: false
false: false
true: true
false: false
true: true
true: true
false: false
false: false
false: false
true: true
false: false

22

从上往下打印二叉树

# #

题目：从上往下打印出二叉树的每个结点，同一层的结点按照从左向右的顺序打印。

#

二叉树结点的定义：

```
public static class BinaryTreeNode {
    int value;
    BinaryTreeNode left;
    BinaryTreeNode right;
}
```

#

解题思路：

这道题实质是考查树的遍历算法。从上到下打印二叉树的规律：每一次打印一个结点的时候，如果该结点有子结点， 则把该结点的子结点放到一个队列的末尾。接下来到队列的头部取出最早进入队列的结点，重复前面的打印操作，直至队列中所有的结点都被打印出来为止。

#

代码实现：

```
public class Test23 {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }
    /**
     * 从上往下打印出二叉树的每个结点，向一层的结点按照从左往右的顺序打印。
     * 例如下的二叉树，
```

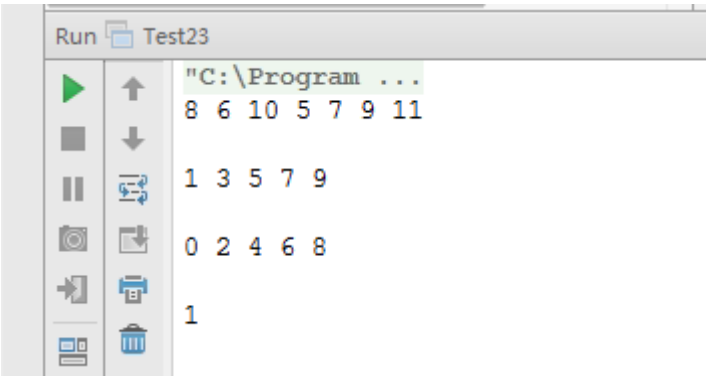
```
     *      8
     *    /   \
     *   6     10
     *  / \   / \
     * 5   7 9  11
     * 则依次打印出8、6、10、5、3 、9、11.
     *
     * @param root 树的结点
     */
    public static void printFromToBottom(BinaryTreeNode root) {
        // 当结点非空时才进行操作
        if (root != null) {
            // 用于存放还未遍历的元素
            Queue<BinaryTreeNode> list = new LinkedList<>();
            // 将根结点入队
            list.add(root);
            // 用于记录当前处理的结点
            BinaryTreeNode curNode;
            // 队列非空则进行处理
            while (!list.isEmpty()) {
                // 删除队首元素
                curNode = list.remove();
                // 输出队首元素的值
                System.out.print(curNode.value + " ");
                // 如果左子结点不为空，则左子结点入队
                if (curNode.left != null) {
                    list.add(curNode.left);
                }
                // 如果右子结点不为空，则左子结点入队
                if (curNode.right != null) {
                    list.add(curNode.right);
                }
            }
        }
    }
    public static void main(String[] args) {
        //      8
        //    /   \
        //   6     10
        //  / \   / \
        // 5   7 9  11
        BinaryTreeNode root = new BinaryTreeNode();
        root.value = 8;
        root.left = new BinaryTreeNode();
        root.left.value = 6;
```

```
root.left.left = new BinaryTreeNode();
root.left.left.value = 5;
root.left.right = new BinaryTreeNode();
root.left.right.value = 7;
root.right = new BinaryTreeNode();
root.right.value = 10;
root.right.left = new BinaryTreeNode();
root.right.left.value = 9;
root.right.right = new BinaryTreeNode();
root.right.right.value = 11;
printFromToBottom(root);
//        1
//       /
//      3
//     /
//    5
//   /
//  7
// /
// 9
BinaryTreeNode root2 = new BinaryTreeNode();
root2.value = 1;
root2.left = new BinaryTreeNode();
root2.left.value = 3;
root2.left.left = new BinaryTreeNode();
root2.left.left.value = 5;
root2.left.left.left = new BinaryTreeNode();
root2.left.left.left.value = 7;
root2.left.left.left.left = new BinaryTreeNode();
root2.left.left.left.left.value = 9;
System.out.println("\n");
printFromToBottom(root2);
// 0
// \
//  2
//   \
//    4
//     \
//      6
//       \
//        8
BinaryTreeNode root3 = new BinaryTreeNode();
root3.value = 0;
root3.right = new BinaryTreeNode();
root3.right.value = 2;
```

```
        root3.right.right = new BinaryTreeNode();
        root3.right.right.value = 4;
        root3.right.right.right = new BinaryTreeNode();
        root3.right.right.right.value = 6;
        root3.right.right.right.right = new BinaryTreeNode();
        root3.right.right.right.right.value = 8;
        System.out.println("\n");
        printFromToBottom(root3);
        // 1
        BinaryTreeNode root4 = new BinaryTreeNode();
        root4.value = 1;
        System.out.println("\n");
        printFromToBottom(root4);
        // null
        System.out.println("\n");
        printFromToBottom(null);
    }
}
```

#

运行结果：



23

二叉搜索树的后序遍历序列

# #

题目：输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则返回 true。否则返回 false。假设输入的数组的任意两个数字都互不相同。

#

解题思路：

在后序遍历得到的序列中， 最后一个数字是树的根结点的值。数组中前面的数字可以分为两部分： 第一部分是左子树结点的值，它们都比根结点的值小： 第二部分是右子树结点的值，它们都比根结点的值大。

#

代码实现：

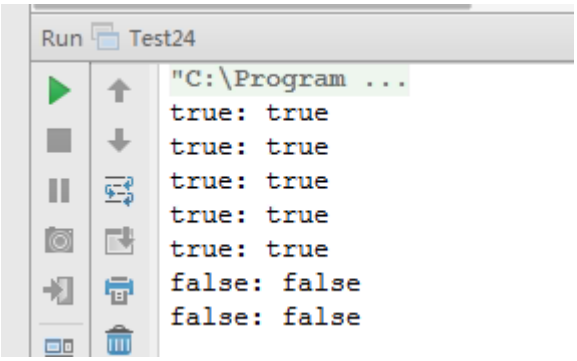
```
public class Test24 {
    /**
     * 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。
     * 如果是则返回true。否则返回false。假设输入的数组的任意两个数字都互不相同。
     *
     * @param sequence 某二叉搜索树的后序遍历的结果
     * @return true：该数组是某二叉搜索树的后序遍历的结果。false：不是
     */
    public static boolean verifySequenceOfBST(int[] sequence) {
        // 输入的数组不能为空，并且有数据
        if (sequence == null || sequence.length <= 0) {
            return false;
        }
        // 有数据，就调用辅助方法
        return verifySequenceOfBST(sequence, 0, sequence.length - 1);
    }
    /**
     * 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。
     * 【此方法与上一个方法不同，未进行空值判断，对于数组度为0的情况返回的true也于上题不同，
     * 此方法只是上面一个方法的辅助实现，对于数数组为null和数组长度为0的情况，执行结果并非相同】
     * 【也就是说此方法只有数组中有数据的情况下才与上面的方法返回同样的结点，
     * verifySequenceOfBST(sequence) ===
     * verifySequenceOfBST(sequence, 0, sequence.length - 1)
```

```
     * 当sequence中有数据才成立
     * 】
     *
     * @param sequence 某二叉搜索树的后序遍历的结果
     * @param start    处理的开始位置
     * @param end      处理的结束位置
     * @return true：该数组是某二叉搜索树的后序遍历的结果。false：不是
     */
    public static boolean verifySequenceOfBST(int[] sequence, int start, int end) {
        // 如果对应要处理的数据只有一个或者已经没有数据要处理（start>end）就返回true
        if (start >= end) {
            return true;
        }
        // 从左向右找第一个不大于根结点（sequence[end]）的元素的位置
        int index = start;
        while (index < end - 1 && sequence[index] < sequence[end]) {
            index++;
        }
        // 执行到此处[end, index-1]的元素都是小于根结点的（sequence[end]）
        // [end, index-1]可以看作是根结点的左子树
        // right用于记录第一个不小于根结点的元素的位置
        int right = index;
        // 接下来要保证[index, end-1]的所有元素都是大于根根点的【A】
        // 因为[index, end-1]只有成为根结点的右子树
        // 从第一个不小于根结点的元素开始，找第一个不大于根结点的元素
        while (index < end - 1 && sequence[index] > sequence[end]) {
            index++;
        }
        // 如果【A】条件满足，那么一定有index=end-1，
        // 如果不满足那说明根结点的右子树[index, end-1]中有小于等于根结点的元素，
        // 不符合二叉搜索树的定义，返回false
        if (index != end - 1) {
            return false;
        }
        // 执行到此处说明直到目前为止，还是合法的
        // [start, index-1]为根结点左子树的位置
        // [index, end-1]为根结点右子树的位置
        index = right;
        return verifySequenceOfBST(sequence, start, index - 1) && verifySequenceOfBST(sequence, index, end - 1);
    }
    public static void main(String[] args) {
        //          10
        //        /  \
        //       6    14
        //      /\    /\
```

```
        //     4  8  12  16
        int[] data = {4, 8, 6, 12, 16, 14, 10};
        System.out.println("true: " + verifySequenceOfBST(data));
        //          5
        //         / \
        //        4   7
        //          /
        //         6
        int[] data2 = {4, 6, 7, 5};
        System.out.println("true: " + verifySequenceOfBST(data2));
        //              5
        //             /
        //            4
        //           /
        //          3
        //         /
        //        2
        //       /
        //      1
        int[] data3 = {1, 2, 3, 4, 5};
        System.out.println("true: " + verifySequenceOfBST(data3));
        // 1
        //  \
        //   2
        //    \
        //     3
        //      \
        //       4
        //        \
        //         5
        int[] data4 = {5, 4, 3, 2, 1};
        System.out.println("true: " + verifySequenceOfBST(data4));
        // 树中只有1个结点
        int[] data5 = {5};
        System.out.println("true: " + verifySequenceOfBST(data5));
        int[] data6 = {7, 4, 6, 5};
        System.out.println("false: " + verifySequenceOfBST(data6));
        int[] data7 = {4, 6, 12, 8, 16, 14, 10};
        System.out.println("false: " + verifySequenceOfBST(data7));
    }
}
```

#

运行结果：


Run
Test24
"C:\Program ...
true: true
true: true
true: true
true: true
true: true
false: false
false: false

24

二叉树中和为某一值的路径

# #

题目：输入一棵二叉树和一个整数， 打印出二叉树中结点值的和为输入整数的所有路径。从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。

#

二叉树结点的定义：

```
public static class BinaryTreeNode {
    int value;
    BinaryTreeNode left;
    BinaryTreeNode right;
}
```

#

解题思路：

由于路径是从根结点出发到叶结点， 也就是说路径总是以根结点为起始点，因此我们首先需要遍历根结点。在树的前序、中序、后序三种遍历方式中，只有前序遍历是首先访问根结点的。

当用前序遍历的方式访问到某一结点时， 我们把该结点添加到路径上，并累加该结点的值。如果该结点为叶结点并且路径中结点值的和刚好等于输入的整数， 则当前的路径符合要求，我们把它打印出来。如果当前结点不是叶结点，则继续访问它的子结点。当前结点访问结束后，递归函数将自动回到它的父结点。因此我们在函数退出之前要在路径上删除当前结点并减去当前结点的值，以确保返回父结点时路径刚好是从根结点到父结点的路径。

不难看出保存路径的数据结构实际上是一个枝， 因为路径要与递归调用状态一致， 而递归调用的本质就是一个压栈和出栈的过程。

#

代码实现：

```
public class Test25 {
    /**
     * 二叉树的树结点
```

```
 */
public static class BinaryTreeNode {
    int value;
    BinaryTreeNode left;
    BinaryTreeNode right;
}
/**
 * 输入一棵二叉树和一个整数， 打印出二叉树中结点值的和为输入整数的所有路径。
 * 从树的根结点开始往下一直到叶销点所经过的结点形成一条路径。
 *
 * @param root       树的根结点
 * @param expectedSum 要求的路径和
 */
public static void findPath(BinaryTreeNode root, int expectedSum) {
    // 创建一个链表，用于存放根结点到当前处理结点的所经过的结点
    List<Integer> list = new ArrayList<>();
    // 如果根结点不为空，就调用辅助处理方法
    if (root != null) {
        findPath(root, 0, expectedSum, list);
    }
}
/**
 * @param root       当前要处理的结点
 * @param curSum     当前记录的和（还未加上当前结点的值）
 * @param expectedSum 要求的路径和
 * @param result     根结点到当前处理结点的所经过的结点，（还未包括当前结点）
 */
public static void findPath(BinaryTreeNode root, int curSum, int expectedSum, List<Integer> result) {
    // 如果结点不为空就进行处理
    if (root != null) {
        // 加上当前结点的值
        curSum += root.value;
        // 将当前结点入队
        result.add(root.value);
        // 如果当前结点的值小于期望的和
        if (curSum < expectedSum) {
            // 递归处理左子树
            findPath(root.left, curSum, expectedSum, result);
            // 递归处理右子树
            findPath(root.right, curSum, expectedSum, result);
        }
        // 如果当前和与期望的和相等
        else if (curSum == expectedSum) {
            // 当前结点是叶结点，则输出结果
            if (root.left == null && root.right == null) {
```

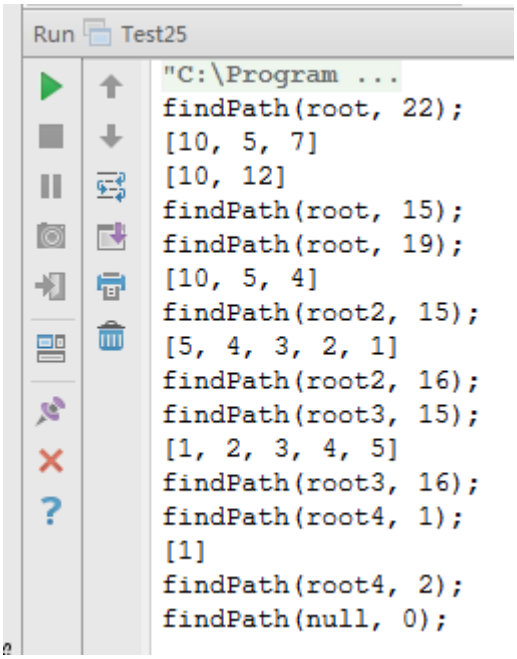
```
                System.out.println(result);
            }
        }
        // 移除当前结点
        result.remove(result.size() - 1);
    }
}
public static void main(String[] args) {
    //           10
    //        /     \
    //       5       12
    //      /\
    //     4  7
    BinaryTreeNode root = new BinaryTreeNode();
    root.value = 10;
    root.left = new BinaryTreeNode();
    root.left.value = 5;
    root.left.left = new BinaryTreeNode();
    root.left.left.value = 4;
    root.left.right = new BinaryTreeNode();
    root.left.right.value = 7;
    root.right = new BinaryTreeNode();
    root.right.value = 12;
    // 有两条路径上的结点和为22
    System.out.println("findPath(root, 22);");
    findPath(root, 22);
    // 没有路径上的结点和为15
    System.out.println("findPath(root, 15);");
    findPath(root, 15);
    // 有一条路径上的结点和为19
    System.out.println("findPath(root, 19);");
    findPath(root, 19);
    //              5
    //             /
    //            4
    //           /
    //          3
    //         /
    //        2
    //       /
    //      1
    BinaryTreeNode root2 = new BinaryTreeNode();
    root2.value = 5;
    root2.left = new BinaryTreeNode();
    root2.left.value = 4;
```

```
root2.left.left = new BinaryTreeNode();
root2.left.left.value = 3;
root2.left.left.left = new BinaryTreeNode();
root2.left.left.left.value = 2;
root2.left.left.left.left = new BinaryTreeNode();
root2.left.left.left.left.value = 1;
// 有一条路径上面的结点和为15
System.out.println("findPath(root2, 15);");
findPath(root2, 15);
// 没有路径上面的结点和为16
System.out.println("findPath(root2, 16);");
findPath(root2, 16);
// 1
//  \
//   2
//    \
//     3
//      \
//       4
//        \
//         5
BinaryTreeNode root3 = new BinaryTreeNode();
root3.value = 1;
root3.right = new BinaryTreeNode();
root3.right.value = 2;
root3.right.right = new BinaryTreeNode();
root3.right.right.value = 3;
root3.right.right.right = new BinaryTreeNode();
root3.right.right.right.value = 4;
root3.right.right.right.right = new BinaryTreeNode();
root3.right.right.right.right.value = 5;
// 有一条路径上面的结点和为15
System.out.println("findPath(root3, 15);");
findPath(root3, 15);
// 没有路径上面的结点和为16
System.out.println("findPath(root3, 16);");
findPath(root3, 16);
// 树中只有1个结点
BinaryTreeNode root4 = new BinaryTreeNode();
root4.value = 1;
// 有一条路径上面的结点和为1
System.out.println("findPath(root4, 1);");
findPath(root4, 1);
// 没有路径上面的结点和为2
System.out.println("findPath(root4, 2);");
```

```
        findPath(root4, 2);
        // 树中没有结点
        System.out.println("findPath(null, 0);");
        findPath(null, 0);
    }
}
```

#

运行结果：



25

复杂链表的复制

# #

题目：请实现函数 ComplexListNode clone(ComplexListNode head)，复制一个复杂链表。在复杂链表中，每个结点除了有一个 next 域指向下一个结点外，还有一个 sibling 指向链表中的任意结点或者 null。

#

结点结构定义：

```
public static class ComplexListNode {
    int value;
    ComplexListNode next;
    ComplexListNode sibling;
}
```

#

解题思路：

图 4.8 是一个含有 5 个结点的复杂链表。图中实线箭头表示 next 指针，虚线箭头表示 sibling 指针。为简单起见，指向 null 的指针没有画出。

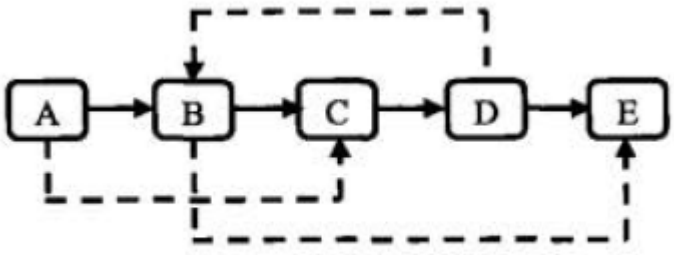


图 4.8　一个含有 5 个结点的复杂链表

在不用辅助空间的情况下实现 O(n)的时间效率。

第一步：仍然是根据原始链表的每个结点N 创建对应的 N’。把 N’链接在N的后面。图 4.8 的链表经过这一步之后的结构，如图 4.9 所示。

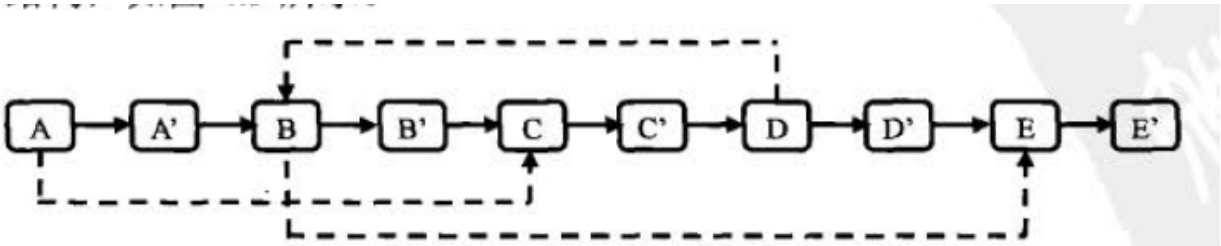
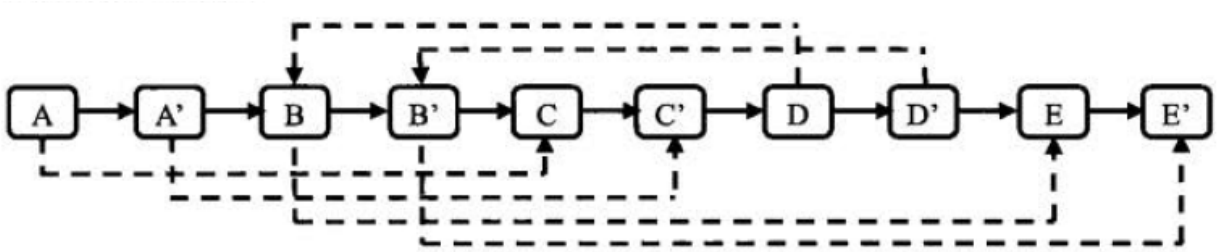


图 4.9　复制复杂链表的第一步

第二步：设置复制出来的结点的 sibling。假设原始链表上的 N 的 sibling 指向结点 S，那么其对应复制出来的 N’是 N的 pext 指向的结点，同样 S’也是 S 的 next 指向的结点。设置 sibling 之后的链表如图 4.10 所示。



第三步：把这个长链表拆分成两个链表。把奇数位置的结点用 next 链接起来就是原始链表，把偶数位置的结点用 next 链接起来就是复制 出来的链表。图 4.10 中的链表拆分之后的两个链表如图 4.11 所示。

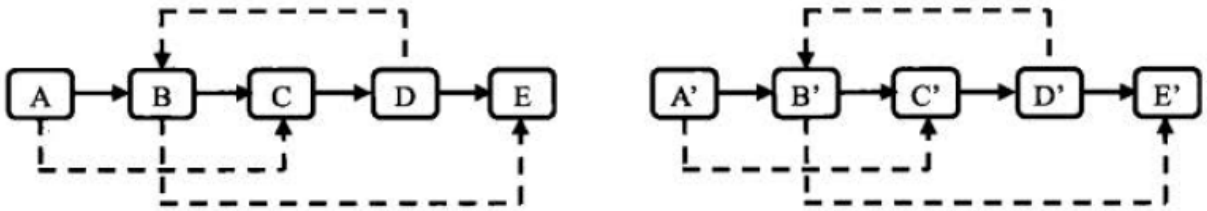


图 4.11　复制复杂链表的第三步

#

代码实现：

```
public class Test26 {
    /**
     * 复杂链表结点
     */
    public static class ComplexListNode {
        int value;
        ComplexListNode next;
        ComplexListNode sibling;
```

```
    }
    /**
     * 实现函复制一个复杂链表。在复杂链表中，每个结点除了有一个next字段指向下一个结点外，
     * 还有一个sibling字段指向链表中的任意结点或者NULL
     *
     * @param head 链表表头结点
     * @return 复制结点的头结点
     */
    public static ComplexListNode clone(ComplexListNode head) {
        // 如果链表为空就直接返回空
        if (head == null) {
            return null;
        }
        // 先复制结点
        cloneNodes(head);
        // 再链接sibling字段
        connectNodes(head);
        // 将整个链表拆分，返回复制链表的头结点
        return reconnectNodes(head);
    }
    /**
     * 复制一个链表，并且将复制后的结点插入到被复制的结点后面，只链接复制结点的next字段
     *
     * @param head 待复制链表的头结点
     */
    public static void cloneNodes(ComplexListNode head) {
        // 如果链表不空，进行复制操作
        while (head != null) {
            // 创建一个新的结点
            ComplexListNode tmp = new ComplexListNode();
            // 将被复制结点的值传给复制结点
//            tmp.value = head.value;
            ////////////////////////////////////////////////////////////////////////////////
            // TODO 此处为了做测试，让复制结点的值都增加了100，如果不需要可以将下面一个行注释掉，打开上一行。
            tmp.value = head.value + 100;
            ////////////////////////////////////////////////////////////////////////////////
            // 复制结点的next指向下一个要被复制的结点
            tmp.next = head.next;
            // 被复制结点的next指向复制结点
            head.next = tmp;
            // 到些处就已经完成了一个结点的复制并且插入到被复制结点的后面
            // heed指向下一个被复制结点的位置
            head = tmp.next;
        }
    }
```

```
/**
 * 设置复制结点的sibling字段
 *
 * @param head 链表的头结
 */
public static void connectNodes(ComplexListNode head) {
    // 如链表不为空
    while (head != null) {
        // 当前处理的结点sibling字段不为空，则要设置其复制结点的sibling字段
        if (head.sibling != null) {
            // 复制结点的sibling指向被复制结点的sibling字段的下一个结点
            // head.next：表求复制结点，
            // head.sibling：表示被复制结点的sibling所指向的结点，
            // 它的下一个结点就是它的复制结点
            head.next.sibling = head.sibling.next;
        }
        // 指向下一个要处理的复制结点
        head = head.next.next;
    }
}
/**
 * 刚复制结点和被复制结点拆开，还原被复制的链表，同时生成监制链表
 *
 * @param head 链表的头结点
 * @return 复制链表的头结点
 */
public static ComplexListNode reconnectNodes(ComplexListNode head) {
    // 当链表为空就直接返回空
    if (head == null) {
        return null;
    }
    // 用于记录复制链表的头结点
    ComplexListNode newHead = head.next;
    // 用于记录当前处理的复制结点
    ComplexListNode pointer = newHead;
    // 被复制结点的next指向下一个被复制结点
    head.next = newHead.next;
    // 指向新的被复制结点
    head = head.next;
    while (head != null) {
        // pointer指向复制结点
        pointer.next = head.next;
        pointer = pointer.next;
        // head的下一个指向复制结点的下一个结点，即原来链表的结点
        head.next = pointer.next;
```

```
            // head指向下一个原来链表上的结点
            head = pointer.next;
        }
        // 返回复制链表的头结点
        return newHead;
    }
    /**
     * 输出链表信息
     *
     * @param head 链表头结点
     */
    public static void printList(ComplexListNode head) {
        while (head != null) {
            System.out.print(head.value + "->");
            head = head.next;
        }
        System.out.println("null");
    }
    /**
     * 判断两个链表是否是同一个链表，不是值相同
     *
     * @param h1 链表头1
     * @param h2 链表头2
     * @return true：两个链表是同一个链表，false：不是
     */
    public static boolean isSame(ComplexListNode h1, ComplexListNode h2) {
        while (h1 != null && h2 != null) {
            if (h1 == h2) {
                h1 = h1.next;
                h2 = h2.next;
            } else {
                return false;
            }
        }
        return h1 == null && h2 == null;
    }
    public static void main(String[] args) {
        //         ------------------
        //        \|/             |
        //  1-------2-------3-------4-------5
        //  |      |     /|\           /|\
        //  --------+--------             |
        //         ------------------------
        ComplexListNode head = new ComplexListNode();
        head.value = 1;
```

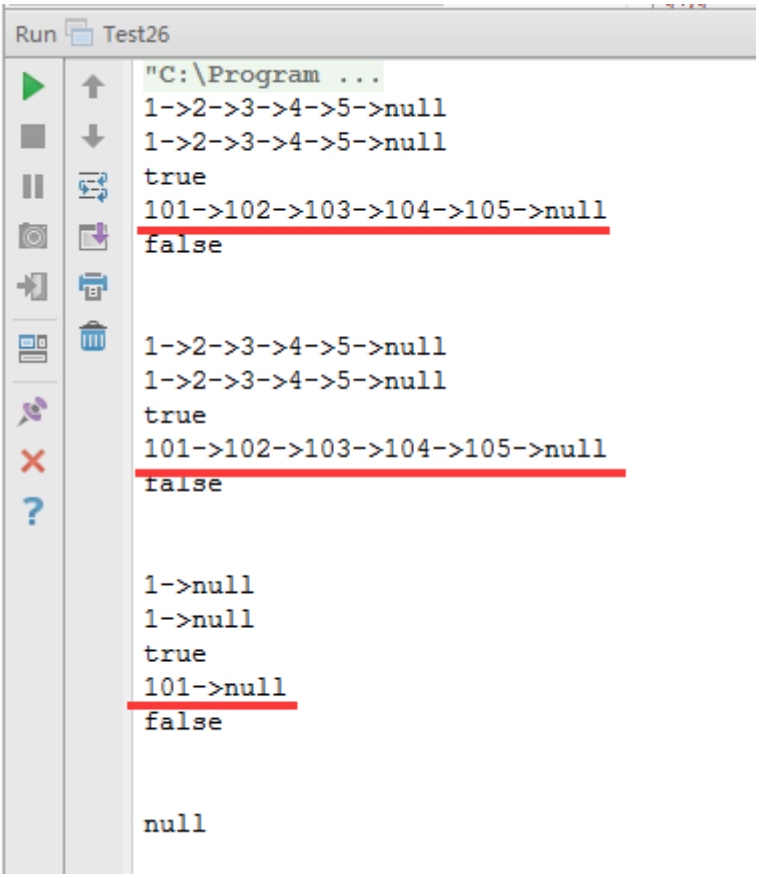
```
head.next = new ComplexListNode();
head.next.value = 2;
head.next.next = new ComplexListNode();
head.next.next.value = 3;
head.next.next.next = new ComplexListNode();
head.next.next.next.value = 4;
head.next.next.next.next = new ComplexListNode();
head.next.next.next.next.value = 5;
head.sibling = head.next.next;
head.next.sibling = head.next.next.next.next.next;
head.next.next.next.next.sibling = head.next;
ComplexListNode tmp = head;
printList(head);
ComplexListNode newHead = clone(head);
printList(head);
System.out.println(isSame(head, tmp));
printList(newHead);
System.out.println(isSame(head, newHead));
// 有指向自身的情况
//         ------------------
//        \|/             |
//  1-------2--------3-------4-------5
//        |      | /|\          /|\
//        |      | --             |
//        |------------------------|
ComplexListNode head2 = new ComplexListNode();
head2.value = 1;
head2.next = new ComplexListNode();
head2.next.value = 2;
head2.next.next = new ComplexListNode();
head2.next.next.value = 3;
head2.next.next.next = new ComplexListNode();
head2.next.next.next.value = 4;
head2.next.next.next.next = new ComplexListNode();
head2.next.next.next.next.value = 5;
head2.next.sibling = head2.next.next.next.next;
head2.next.next.next.sibling = head2.next.sibling;
head2.next.next.sibling = head2.next.next;
System.out.println("\n");
tmp = head2;
printList(head2);
ComplexListNode newHead2 = clone(head2);
printList(head2);
System.out.println(isSame(head2, tmp));
printList(newHead2);
```

```
        System.out.println(isSame(head2, newHead2));
        ComplexListNode head3 = new ComplexListNode();
        head3.value = 1;
        System.out.println("\n");
        tmp = head3;
        printList(head3);
        ComplexListNode newHead3 = clone(head3);
        printList(head3);
        System.out.println(isSame(head3, tmp));
        printList(newHead3);
        System.out.println(isSame(head3, newHead3));
        System.out.println("\n");
        ComplexListNode head4 = clone(null);
        printList(head4);
    }
}
```

#

运行结果：

注意：划红线部分是为了区分原链表和复制链表，具体还原见代码注释。



26

二叉搜索树与双向链表

# #

题目：输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

比如输入图 4.12 中左边的二叉搜索树，则输出转换之后的排序现向链表。

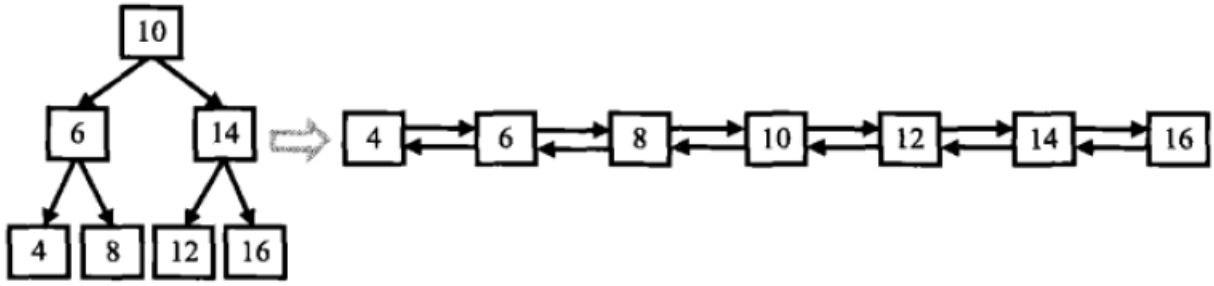


图 4.12　一棵二叉搜索树及转换之后的排序双向链表

#

结点定义：

```
public static class BinaryTreeNode {
    int value;
    BinaryTreeNode left;
    BinaryTreeNode right;
}
```

#

解题思路：

在二叉树中，每个结点都有两个指向子结点的指针。在双向链表中，每个结点也有两个指针，它们分别指向前一个结点和后一个结点。由于这两种结点的结构相似，同时二叉搜索树也是一种排序的数据结构，因此在理论上有可能实现二叉搜索树和排序的双向链表的转换。在搜索二叉树中，左子结点的值总是小于父结点的值，右子结点的值总是大于父结点的值。因此我们在转换成排序双向链表时，原先指向左子结点的指针调整为链表中指向前一个结点的指针，原先指向右子结点的指针调整为链表中指向后一个结点指针。接下来我们考虑该如何转换。

由于要求转换之后的链表是排好序的，我们可以中序遍历树中的每一个结点， 这是因为中序遍历算法的特点是按照从小到大的顺序遍历二叉树的每一个结点。当遍历到根结点的时候，我们把树看成三部分：值为 10 的结点、根结点值为 6 的左子树、根结点值为 14 的右子树。根据排序链表的定义，值为 10 的结点将和它的左子树的最大一个结点（即值为 8 的结点）链接起来，同时它还将和右子树最小的结点（即值为 12 的结点）链接起来，如图 4.13 所示。

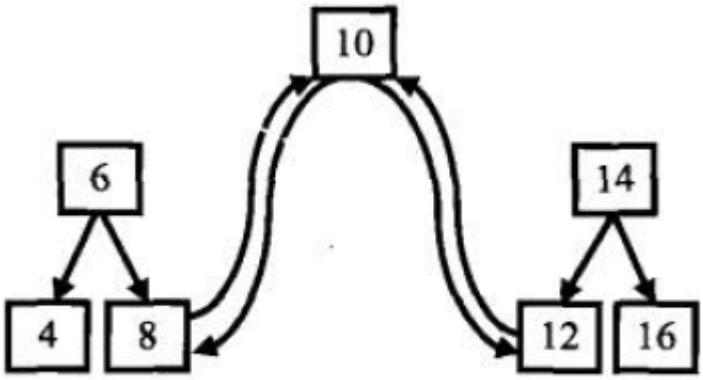


图 4.13　把二叉搜索树看成三部分

按照中序遍历的顺序， 当我们遍历转换到根结点（值为 10 的结点）时，它的左子树已经转换成一个排序的链表了， 并且处在链表中的最后一个结点是当前值最大的结点。我们把值为 8 的结点和根结点链接起来，此时链表中的最后一个结点就是 10 了。接着我们去地历转换右子树， 并把根结点和右子树中最小的结点链接起来。至于怎么去转换它的左子树和右子树，由于遍历和转换过程是一样的，我们很自然地想到可以用递归。

#

代码实现：

```
public class Test27 {
    /**
     * 二叉树的树结点
     */
    public static class BinaryTreeNode {
        int value;
        BinaryTreeNode left;
        BinaryTreeNode right;
    }
    /**
     * 题目：输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。
     * 要求不能创建任何新的结点，只能调整树中结点指针的指向。
     *
```

```
 * @param root 二叉树的根结点
 * @return 双向链表的头结点
 */
public static BinaryTreeNode convert(BinaryTreeNode root) {
    // 用于保存处理过程中的双向链表的尾结点
    BinaryTreeNode[] lastNode = new BinaryTreeNode[1];
    convertNode(root, lastNode);
    // 找到双向链表的头结点
    BinaryTreeNode head = lastNode[0];
    while (head != null && head.left != null) {
        head = head.left;
    }
    return head;
}
/**
 * 链表表转换操作
 *
 * @param node     当前的根结点
 * @param lastNode 已经处理好的双向链表的尾结点，使用一个长度为1的数组，类似C++中的二级指针
 */
public static void convertNode(BinaryTreeNode node, BinaryTreeNode[] lastNode) {
    // 结点不为空
    if (node != null) {
        // 如果有左子树就先处理左子树
        if (node.left != null) {
            convertNode(node.left, lastNode);
        }
        // 将当前结点的前驱指向已经处理好的双向链表（由当前结点的左子树构成）的尾结点
        node.left = lastNode[0];
        // 如果左子树转换成的双向链表不为空，设置尾结点的后继
        if (lastNode[0] != null) {
            lastNode[0].right = node;
        }
        // 记录当前结点为尾结点
        lastNode[0] = node;
        // 处理右子树
        if (node.right != null) {
            convertNode(node.right, lastNode);
        }
    }
}
public static void main(String[] args) {
    test01();
    test02();
    test03();
```

```
    test04();
    test05();
}
private static void printList(BinaryTreeNode head) {
    while (head != null) {
        System.out.print(head.value + "->");
        head = head.right;
    }
    System.out.println("null");
}
private static void printTree(BinaryTreeNode root) {
    if (root != null) {
        printTree(root.left);
        System.out.print(root.value + "->");
        printTree(root.right);
    }
}
//          10
//        /    \
//       6      14
//      /\      /\
//     4  8    12  16
private static void test01() {
    BinaryTreeNode node10 = new BinaryTreeNode();
    node10.value = 10;
    BinaryTreeNode node6 = new BinaryTreeNode();
    node6.value = 6;
    BinaryTreeNode node14 = new BinaryTreeNode();
    node14.value = 14;
    BinaryTreeNode node4 = new BinaryTreeNode();
    node4.value = 4;
    BinaryTreeNode node8 = new BinaryTreeNode();
    node8.value = 8;
    BinaryTreeNode node12 = new BinaryTreeNode();
    node12.value = 12;
    BinaryTreeNode node16 = new BinaryTreeNode();
    node16.value = 16;
    node10.left = node6;
    node10.right = node14;
    node6.left = node4;
    node6.right = node8;
    node14.left = node12;
    node14.right = node16;
    System.out.print("Before convert: ");
    printTree(node10);
```

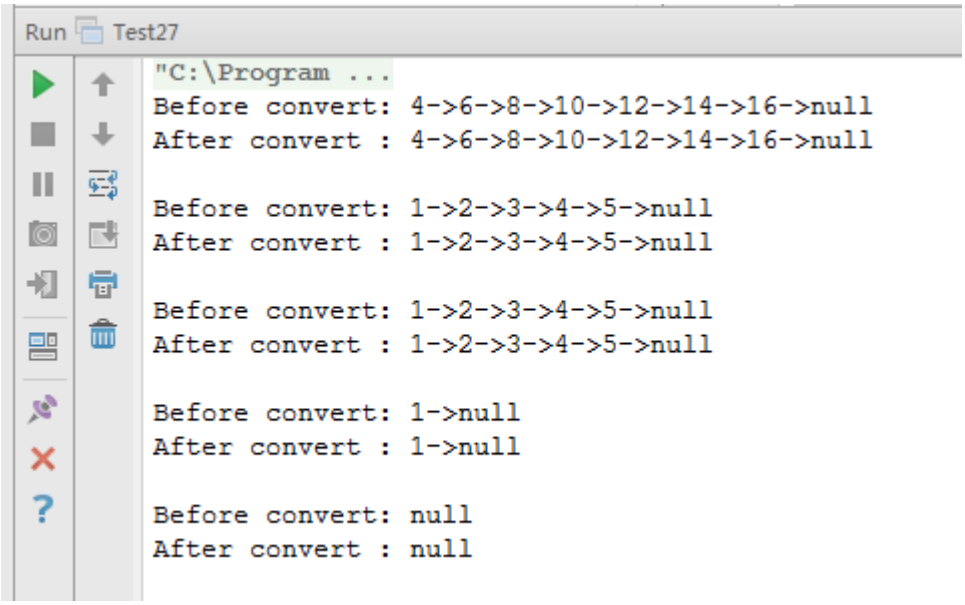
```
    System.out.println("null");
    BinaryTreeNode head = convert(node10);
    System.out.print("After convert : ");
    printList(head);
    System.out.println();
}
//              5
//             /
//            4
//           /
//          3
//         /
//        2
//       /
//      1
private static void test02() {
    BinaryTreeNode node1 = new BinaryTreeNode();
    node1.value = 1;
    BinaryTreeNode node2 = new BinaryTreeNode();
    node2.value = 2;
    BinaryTreeNode node3 = new BinaryTreeNode();
    node3.value = 3;
    BinaryTreeNode node4 = new BinaryTreeNode();
    node4.value = 4;
    BinaryTreeNode node5 = new BinaryTreeNode();
    node5.value = 5;
    node5.left = node4;
    node4.left = node3;
    node3.left = node2;
    node2.left = node1;
    System.out.print("Before convert: ");
    printTree(node5);
    System.out.println("null");
    BinaryTreeNode head = convert(node5);
    System.out.print("After convert : ");
    printList(head);
    System.out.println();
}
// 1
//  \
//   2
//    \
//     3
//      \
//       4
```

```
//       \
//        5
private static void test03() {
    BinaryTreeNode node1 = new BinaryTreeNode();
    node1.value = 1;
    BinaryTreeNode node2 = new BinaryTreeNode();
    node2.value = 2;
    BinaryTreeNode node3 = new BinaryTreeNode();
    node3.value = 3;
    BinaryTreeNode node4 = new BinaryTreeNode();
    node4.value = 4;
    BinaryTreeNode node5 = new BinaryTreeNode();
    node5.value = 5;
    node1.right = node2;
    node2.right = node3;
    node3.right = node4;
    node4.right = node5;
    System.out.print("Before convert: ");
    printTree(node1);
    System.out.println("null");
    BinaryTreeNode head = convert(node1);
    System.out.print("After convert : ");
    printList(head);
    System.out.println();
}
// 只有一个结点
private static void test04() {
    BinaryTreeNode node1 = new BinaryTreeNode();
    node1.value = 1;
    System.out.print("Before convert: ");
    printTree(node1);
    System.out.println("null");
    BinaryTreeNode head = convert(node1);
    System.out.print("After convert : ");
    printList(head);
    System.out.println();
}
// 没有结点
private static void test05() {
    System.out.print("Before convert: ");
    printTree(null);
    System.out.println("null");
    BinaryTreeNode head = convert(null);
    System.out.print("After convert : ");
    printList(head);
```

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

#

运行结果：



27

字符串的排列

# #

题目：输入一个字符串，打印出该字符串中字符的所有排列。例如输入字符串 abc。则打印出由字符 a、b、c 所能排列出来的所有字符串 abc、acb、bac 、bca、cab 和 cba 。

#

解题思路：

把一个字符串看成由两部分组成：第一部分为它的第一个字符，第二部分是后面的所有字符。在图 4.14 中，我们用两种不同的背景颜色区分字符串的两部分。

我们求整个字符串的排列，可以看成两步：首先求所有可能出现在第一个位置的字符，即把第一个字符和后面所有的字符交换。图 4.14 就是分别把第一个字符 a 和后面的 b、c 等字符交换的情形。首先固定第一个字符（如图 4.14 (a ）所示 〉，求后面所有字符的排列。这个时候我们仍把后面的所有字符分成两部分：后面字符的第一个字符，以及这个字符之后的所有字符。然后把第一个字符逐一和它后面的字符交换（如图 4.14 （b ）所示）。。。。。。

这其实是很典型的递归思路。

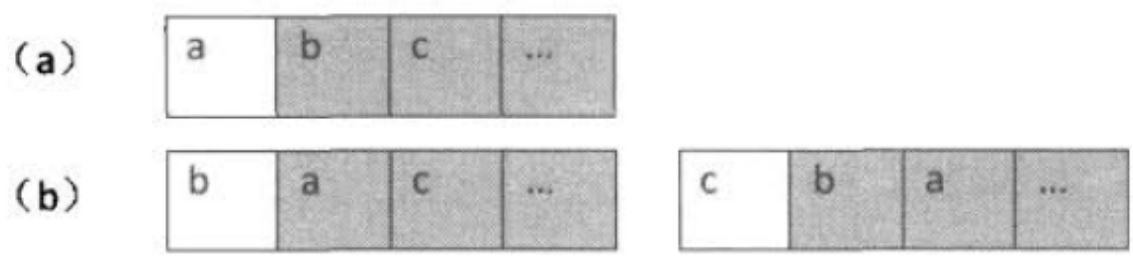


图 4.14　求字符串的排列的过程

#

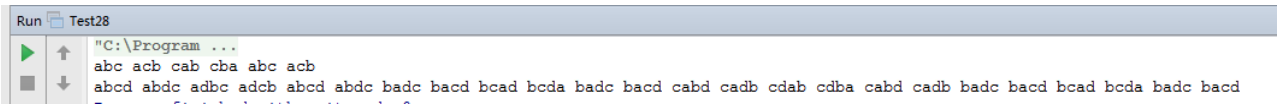
代码实现：

```
public class Test28 {
    /**
     * 题目：输入一个字符串，打印出该字符事中字符的所有排列。例如输入字符串abc。
     * 则打印出由字符a、b、c 所能排列出来的所有字符串abc、acb、bac、bca、cab和cba。
```

```
     *
     * @param chars 待排序的字符数组
     */
    public static void permutation(char[] chars) {
        // 输入较验
        if (chars == null || chars.length < 1) {
            return;
        }
        // 进行排列操作
        permutation(chars, 0);
    }
    /**
     * 求字符数组的排列
     *
     * @param chars 待排列的字符串
     * @param begin 当前处理的位置
     */
    public static void permutation(char[] chars, int begin) {
        // 如果是最后一个元素了，就输出排列结果
        if (chars.length - 1 == begin) {
            System.out.print(new String(chars) + " ");
        } else {
            char tmp;
            // 对当前还未处理的字符串进行处理，每个字符都可以作为当前处理位置的元素
            for (int i = begin; i < chars.length; i++) {
                // 下面是交换元素的位置
                tmp = chars[begin];
                chars[begin] = chars[i];
                chars[i] = tmp;
                // 处理下一个位置
                permutation(chars, begin + 1);
            }
        }
    }
    public static void main(String[] args) {
        char[] c1 = {'a', 'b', 'c'};
        permutation(c1);
        System.out.println();
        char[] c2 = {'a', 'b', 'c', 'd'};
        permutation(c2);
    }
}
```

#

运行结果：


Run
Test28
"C:\Program ...
abc acb cab cba abc acb
abcd abdc adbc adcb abcd abdc badc bacd bcad bcda badc bacd cabd cadb cdab cdba cabd cadb badc bacd bcad bcda badc bacd

28

数组中出现次数超过一半的数字

#

题目：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字

#

例子说明：

如输入一个长度为 9 的数组 { 1, 2, 3, 2, 2, 2, 5, 4, 2 } 。由于数字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2 。

#

解题思路：

#

解法一：基于 Partition 函数的 O（n）算法

数组中有一个数字出现的次数超过了数组长度的一半。如果把这个数组排序，那么排序之后位于数组中间的数字一定就是那个出现次数超过数组长度一半的数字。也就是说，这个数字就是统计学上的中位数，即长度为 n 的数组中第 n/2 大的数字。

这种算法是受快速排序算法的启发。在随机快速排序算法中，我们先在数组中随机选择一个数字，然后调整数组中数字的顺序， 使得比选中的数字小数字都排在它的左边，比选中的数字大的数字都排在它的右边。如果这个选中的数字的下标刚好是 n/2，那么这个数字就是数组的中位数。如果它的下标大于 n/2 ，那么中位数应该位于它的左边，我们可以接着在它的左边部分的数组中查找。如果它的下标小于 n/2，那么中位数应该位于它的右边，我们可以接着在它的右边部分的数组中查找。这是一个典型的递归过程。

#

解法二：根据数组组特点找出O(n)的算法

数组中有一个数字出现的次数超过数组长度的一半，也就是说它出现的次数比其他所有数字出现次数的和还要多。因此我们可以考虑在遍历数组的时候保存两个值： 一个是数组中的一个数字， 一个是次数。当我们遍历到

下～个数字的时候，如果下一个数字和我们之前保存的数字相同，则次数加 l ： 如果下一个数字和我们之前保存的数字，不同，则次数减 1 。如果次数为霉，我们需要保存下一个数字，并把次数设为 1 。由于我们要找的数字出现的次数比其他所有数字出现的次数之和还要多，那么要找的数字肯定是最后一次把次数设为 1 时对应的数字。

本题采用第二种实现方式。

#

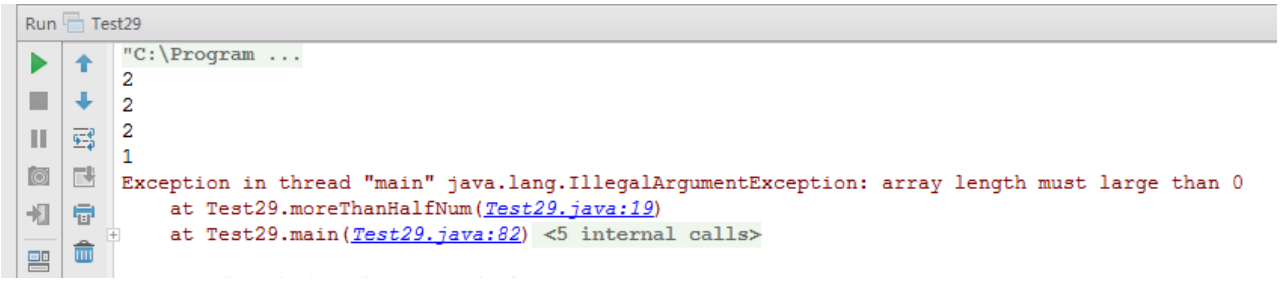
代码实现：

```
public class Test29 {
    /**
     * 题目：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字
     *
     * @param numbers 输入数组
     * @return 找到的数字
     */
    public static int moreThanHalfNum(int[] numbers) {
        // 输入校验
        if (numbers == null || numbers.length < 1) {
            throw new IllegalArgumentException("array length must large than 0");
        }
        // 用于记录出现次数大于数组一半的数
        int result = numbers[0];
        // 于当前记录的数不同的数的个数
        int count = 1;
        // 从第二个数开始向后找
        for (int i = 1; i < numbers.length; i++) {
            // 如果记数为0
            if (count == 0) {
                // 重新记录一个数，假设它是出现次数大于数组一半的
                result = numbers[i];
                // 记录统计值
                count = 1;
            }
            // 如果记录的值与统计值相等，记数值增加
            else if (result == numbers[i]) {
                count++;
            }
            // 如果不相同就减少，相互抵消
            else {
                count--;
            }
```

```
        }
        // 最后的result可能是出现次数大于数组一半长度的值
        // 统计result的出现次数
        count = 0;
        for (int number : numbers) {
            if (result == number) {
                count++;
            }
        }
        // 如果出现次数大于数组的一半就返回对应的值
        if (count > numbers.length / 2) {
            return result;
        }
        // 否则输入异常
        else {
            throw new IllegalArgumentException("invalid input");
        }
    }
    public static void main(String[] args) {
        // 存在出现次数超过数组长度一半的数字
        int numbers[] = {1, 2, 3, 2, 2, 2, 5, 4, 2};
        System.out.println(moreThanHalfNum(numbers));
        // 出现次数超过数组长度一半的数字都出现在数组的前半部分
        int numbers2[] = {2, 2, 2, 2, 2, 1, 3, 4, 5};
        System.out.println(moreThanHalfNum(numbers2));
        // 出现次数超过数组长度一半的数字都出现在数组的后半部分
        int numbers3[] = {1, 3, 4, 5, 2, 2, 2, 2, 2};
        System.out.println(moreThanHalfNum(numbers3));
        // 只有一个数
        int numbers4[] = {1};
        System.out.println(moreThanHalfNum(numbers4));
        // 输入空指针
        moreThanHalfNum(null);
        // 不存在出现次数超过数组长度一半的数字
        int numbers5[] = {1, 2, 3, 2, 4, 2, 5, 2, 3};
        moreThanHalfNum(numbers5);
    }
}
```

#

运行结果：


Run
Test29
"C:\Program ...
2
2
2
1
Exception in thread "main" java.lang.IllegalArgumentException: array length must large than 0
at Test29.moreThanHalfNum(Test29.java:19)
at Test29.main(Test29.java:82) <5 internal calls>

29

最小的 k 个数

#

例子说明：

例如输入 4 、5 、1、6、2、7、3 、8 这 8 个数字，则最小的 4 个数字是 1 、2、3 、4

#

解题思路：

#

解法一：O(n)时间算法，只有可以修改输入数组时可用。

可以基于 Partition 函数来解决这个问题。如果基于数组的第k个数字来调整，使得比第k个数字小的所有数字都位于数组的左边，比第 k 个数字大的所有数字都位于数组的右边。这样调整之后，位于数组中左边的 k 个数字就是最小的 k 个数字（这 k 个数字不一定是排序的）。

#

解法二： O（nlogk）的算法，精剧适合处理海量数据。

先创建一个大小为 k 的数据容器来存储最小的 k 个数字，接下来我们每次从输入的 n 个整数中读入一个数. 如果容器中已有的数字少于 k 个，则直接把这次读入的整数放入容器之中：如果容器中己有k 数字了，也就是容器己满，此时我们不能再插入新的数字而只能替换已有的数字。找出这己有的 k 个数中的最大值，然后 1 在这次待插入的整数和最大值进行比较。如果待插入的值比当前己有的最大值小，则用这个数替换当前已有的最大值：如果待插入的值比当前已有的最大值还要大，那么这个数不可能是最小的k个整数之一，于是我们可以抛弃这个整数。

因此当容器满了之后，我们要做 3 件事情： 一是在 k 个整数中找到最大数： 二是有可能在这个容器中删除最大数： 三是有可能要插入一个新的数字。我们可以使用一个大顶堆在 O(logk）时间内实现这三步操作。

本题实现了两种方法。

#

代码实现：

```
public class Test30 {
    /**
     * 大顶堆
     *
     * @param <T> 参数化类型
     */
    private final static class MaxHeap<T extends Comparable<T>> {
        // 堆中元素存放的集合
        private List<T> items;
        // 用于计数
        private int cursor;
        /**
         * 构造一个椎，始大小是32
         */
        public MaxHeap() {
            this(32);
        }
        /**
         * 造诣一个指定初始大小的堆
         *
         * @param size 初始大小
         */
        public MaxHeap(int size) {
            items = new ArrayList<>(size);
            cursor = -1;
        }
        /**
         * 向上调整堆
         *
         * @param index 被上移元素的起始位置
         */
        public void siftUp(int index) {
            T intent = items.get(index); // 获取开始调整的元素对象
            while (index > 0) { // 如果不是根元素
                int parentIndex = (index - 1) / 2; // 找父元素对象的位置
                T parent = items.get(parentIndex);  // 获取父元素对象
                if (intent.compareTo(parent) > 0) { //上移的条件，子节点比父节点大
                    items.set(index, parent); // 将父节点向下放
                    index = parentIndex; // 记录父节点下放的位置
                } else { // 子节点不比父节点大，说明父子路径已经按从大到小排好顺序了，不需要调整了
                    break;
                }
            }
            // index此时记录是的最后一个被下放的父节点的位置（也可能是自身），所以将最开始的调整的元素值放入index位置即可
            items.set(index, intent);
```

```
}
/**
 * 向下调整堆
 *
 * @param index 被下移的元素的起始位置
 */
public void siftDown(int index) {
    T intent = items.get(index);  // 获取开始调整的元素对象
    int leftIndex = 2 * index + 1; // // 获取开始调整的元素对象的左子结点的元素位置
    while (leftIndex < items.size()) { // 如果有左子结点
        T maxChild = items.get(leftIndex); // 取左子结点的元素对象，并且假定其为两个子结点中最大的
        int maxIndex = leftIndex; // 两个子节点中最大节点元素的位置，假定开始时为左子结点的位置
        int rightIndex = leftIndex + 1;  // 获取右子结点的位置
        if (rightIndex < items.size()) {  // 如果有右子结点
            T rightChild = items.get(rightIndex);  // 获取右子结点的元素对象
            if (rightChild.compareTo(maxChild) > 0) {  // 找出两个子节点中的最大子结点
                maxChild = rightChild;
                maxIndex = rightIndex;
            }
        }
        // 如果最大子节点比父节点大，则需要向下调整
        if (maxChild.compareTo(intent) > 0) {
            items.set(index, maxChild); // 将子节点向上移
            index = maxIndex; // 记录上移节点的位置
            leftIndex = index * 2 + 1; // 找到上移节点的左子节点的位置
        } else { // 最大子节点不比父节点大，说明父子路径已经按从大到小排好顺序了，不需要调整了
            break;
        }
    }
    // index此时记录是的最后一个被上移的子节点的位置（也可能是自身），所以将最开始的调整的元素值放入index位置即可
    items.set(index, intent);
}
/**
 * 向堆中添加一个元素
 *
 * @param item 等待添加的元素
 */
public void add(T item) {
    items.add(item); // 将元素添加到最后
    siftUp(items.size() - 1); // 循环上移，以完成重构
}
/**
 * 删除堆顶元素
 *
 * @return 堆顶部的元素
```

```
     */
    public T deleteTop() {
        if (items.isEmpty()) { // 如果堆已经为空，就报出异常
            throw new RuntimeException("The heap is empty.");
        }
        T maxItem = items.get(0); // 获取堆顶元素
        T lastItem = items.remove(items.size() - 1); // 删除最后一个元素
        if (items.isEmpty()) { // 删除元素后，如果堆为空的情况，说明删除的元素也是堆顶元素
            return lastItem;
        }
        items.set(0, lastItem); // 将删除的元素放入堆顶
        siftDown(0); // 自上向下调整堆
        return maxItem; // 返回堆顶元素
    }
    /**
     * 获取下一个元素
     *
     * @return 下一个元素对象
     */
    public T next() {
        if (cursor >= items.size()) {
            throw new RuntimeException("No more element");
        }
        return items.get(cursor);
    }
    /**
     * 判断堆中是否还有下一个元素
     *
     * @return true堆中还有下一个元素，false堆中无下五元素
     */
    public boolean hasNext() {
        cursor++;
        return cursor < items.size();
    }
    /**
     * 获取堆中的第一个元素
     *
     * @return 堆中的第一个元素
     */
    public T first() {
        if (items.size() == 0) {
            throw new RuntimeException("The heap is empty.");
        }
        return items.get(0);
    }
```

```
    /**
     * 判断堆是否为空
     *
     * @return true是，false否
     */
    public boolean isEmpty() {
        return items.isEmpty();
    }
    /**
     * 获取堆的大小
     *
     * @return 堆的大小
     */
    public int size() {
        return items.size();
    }
    /**
     * 清空堆
     */
    public void clear() {
        items.clear();
    }
    @Override
    public String toString() {
        return items.toString();
    }
}
/**
 * 题目： 输入n个整数，找出其中最小的k个数。
 * 【第二种解法】
 * @param input  输入数组
 * @param output 输出数组
 */
public static void getLeastNumbers2(int[] input, int[] output) {
    if (input == null || output == null || output.length <= 0 || input.length < output.length) {
        throw new IllegalArgumentException("Invalid args");
    }
    MaxHeap<Integer> maxHeap = new MaxHeap<>(output.length);
    for (int i : input) {
        if (maxHeap.size() < output.length) {
            maxHeap.add(i);
        } else {
            int max = maxHeap.first();
            if (max > i) {
                maxHeap.deleteTop();
```

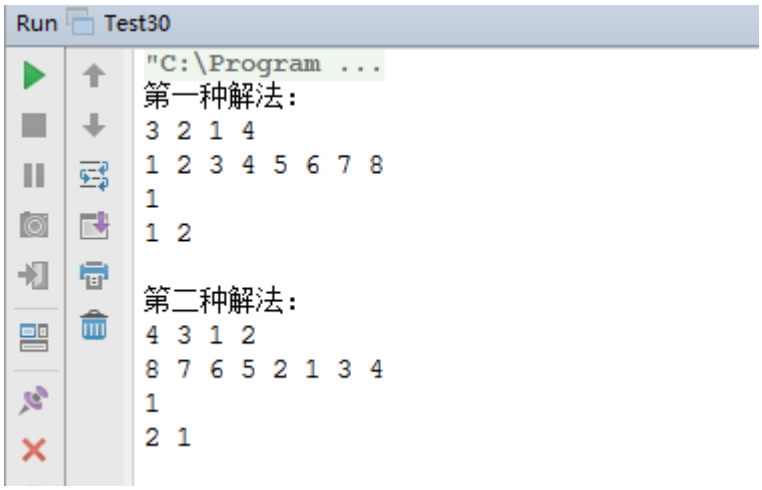
```
                maxHeap.add(i);
            }
        }
    }
    for (int i = 0; maxHeap.hasNext(); i++) {
        output[i] = maxHeap.next();
    }
}
/**
 * 题目： 输入n个整数，找出其中最小的k个数。
 * 【第一种解法】
 * @param input  输入数组
 * @param output 输出数组
 */
public static void getLeastNumbers(int[] input, int[] output) {
    if (input == null || output == null || output.length <= 0 || input.length < output.length) {
        throw new IllegalArgumentException("Invalid args");
    }
    int start = 0;
    int end = input.length - 1;
    int index = partition(input, start, end);
    int target = output.length - 1;
    while (index != target) {
        if (index < target) {
            start = index + 1;
        } else {
            end = index - 1;
        }
        index = partition(input, start, end);
    }
    System.arraycopy(input, 0, output, 0, output.length);
}
/**
 * 分区算法
 *
 * @param input 输入数组
 * @param start 开始下标
 * @param end   结束下标
 * @return 分区位置
 */
private static int partition(int[] input, int start, int end) {
    int tmp = input[start];
    while (start < end) {
        while (start < end && input[end] >= tmp) {
            end--;
```

```
        }
        input[start] = input[end];
        while (start < end && input[start] <= tmp) {
            start++;
        }
        input[end] = input[start];
    }
    input[start] = tmp;
    return start;
}
public static void main(String[] args) {
    System.out.println("第一种解法：");
    test1();
    System.out.println();
    System.out.println("第二种解法：");
    test2();
}
private static void test1() {
    int[] data = {4, 5, 1, 6, 2, 7, 3, 8};
    int[] output = new int[4];
    getLeastNumbers(data, output);
    for (int i : output) {
        System.out.print(i + " ");
    }
    System.out.println();
    int[] output2 = new int[8];
    getLeastNumbers(data, output2);
    for (int i : output2) {
        System.out.print(i + " ");
    }
    System.out.println();
    int[] output3 = new int[1];
    getLeastNumbers(data, output3);
    for (int i : output3) {
        System.out.print(i + " ");
    }
    System.out.println();
    int[] data2 = {4, 5, 1, 6, 2, 7, 2, 8};
    int[] output4 = new int[2];
    getLeastNumbers(data2, output4);
    for (int i : output4) {
        System.out.print(i + " ");
    }
    System.out.println();
}
```

```
    private static void test2() {
        int[] data = {4, 5, 1, 6, 2, 7, 3, 8};
        int[] output = new int[4];
        getLeastNumbers2(data, output);
        for (int i : output) {
            System.out.print(i + " ");
        }
        System.out.println();
        int[] output2 = new int[8];
        getLeastNumbers2(data, output2);
        for (int i : output2) {
            System.out.print(i + " ");
        }
        System.out.println();
        int[] output3 = new int[1];
        getLeastNumbers2(data, output3);
        for (int i : output3) {
            System.out.print(i + " ");
        }
        System.out.println();
        int[] data2 = {4, 5, 1, 6, 2, 7, 2, 8};
        int[] output4 = new int[2];
        getLeastNumbers2(data2, output4);
        for (int i : output4) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

#

运行结果


Run
Test30
"C:\Program ...
第一种解法:
3 2 1 4
1 2 3 4 5 6 7 8
1
1 2
第二种解法:
4 3 1 2
8 7 6 5 2 1 3 4
1
2 1

!!!!!!!!!!!!!!!

30

连续子数组的最大和

# #

题目：输入一个整型数组，数组里有正数也有负数。数组中一个或连续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为 O(n)。

#

例子说明：

例如输入的数组为{1, −2, 3, 10, −4, 7, 2, −5}，和最大的子数组为 { 3, 10, −4, 7, 2}。因此输出为该子数组的和 18 。

#

解题思路：

#

解法一：举例分析数组的规律。

我们试着从头到尾逐个累加示例数组中的每个数字。初始化和为 0。第一步加上第一个数字 1， 此时和为 1。接下来第二步加上数字 −2，和就变成了 −1。第三步刷上数字3。我们注意到由于此前累计的和是 −1 ，小于 0，那如果用−1 加上 3 ，得到的和是 2 ， 比 3 本身还小。也就是说从第一个数字开始的子数组的和会小于从第三个数字开始的子数组的和。因此我们不用考虑从第一个数字开始的子数组，之前累计的和也被抛弃。

我们从第三个数字重新开始累加，此时得到的和是 3。接下来第四步加 10，得到和为 13 。第五步加上 −4， 和为 9。我们发现由于 −4 是一个负数，因此累加 −4 之后得到的和比原来的和还要小。因此我们要把之前得到的和 13 保存下来，它有可能是最大的子数组的和。第六步加上数字 7，9 加 7 的结果是 16，此时和比之前最大的和 13 还要大， 把最大的子数组的和由 13 更新为 16。第七步加上 2，累加得到的和为 18，同时我们也要更新最大子数组的和。第八步加上最后一个数字 −5，由于得到的和为 13 ，小于此前最大的和 18，因此最终最大的子数组的和为 18 ，对应的子数组是 { 3, 10, −4, 7, 2 } 。

#

解法二： 应用动态归划法。

可以用动态规划的思想来分析这个问题。如果用函数 f(i)表示以第 i 个数字结尾的子数组的最大和，那么我们需要求出 max[f(i)]，其中 0 <= i < n。我们可用如下边归公式求 f(i):

$$f(i)=\begin{cases} pData[i] & i=0或者f(i-1)\le 0 \\ f(i-1)+pData[i] & i\neq 0并且f(i-1)>0 \end{cases}$$

这个公式的意义：当以第 i-1 个数字结尾的子数组中所有数字的和小于 0 时，如果把这个负数与第 i 个数累加，得到的结果比第 i 个数字本身还要小，所以这种情况下以第 i 个数字结尾的子数组就是第 i 个数字本身。如果以第 i-1 个数字结尾的子数组中所有数字的和大于 0，与第 i 个数字累加就得到以第 i 个数字结尾的子数组中所有数字的和。

本题采用第一种实现方式。

#

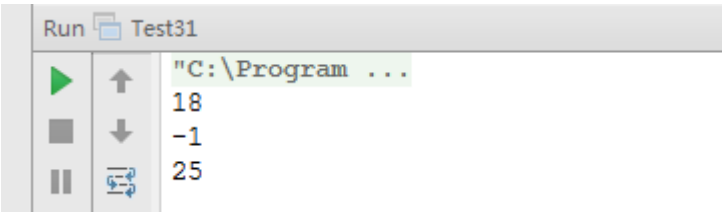
代码实现：

```
public class Test31 {
    /**
     * 题目2 输入一个整型数组，数组里有正数也有负数。数组中一个或连
     * 续的多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)。
     *
     * @param arr 输入数组
     * @return 最大的连续子数组和
     */
    public static int findGreatestSumOfSubArray(int[] arr) {
        // 参数校验
        if (arr == null || arr.length < 1) {
            throw new IllegalArgumentException("Array must contain an element");
        }
        // 记录最大的子数组和，开始时是最小的整数
        int max = Integer.MIN_VALUE;
        // 当前的和
        int curMax = 0;
        // 数组遍历
        for (int i : arr) {
            // 如果当前和小于等于0，就重新设置当前和
            if (curMax <= 0) {
                curMax = i;
            }
            // 如果当前和大于0，累加当前和
```

```
            else {
                curMax += i;
            }
            // 更新记录到的最在的子数组和
            if (max < curMax) {
                max = curMax;
            }
        }
        return max;
    }
    public static void main(String[] args) {
        int[] data = {1, -2, 3, 10, -4, 7, 2, -5};
        int[] data2 = {-2, -8, -1, -5, -9};
        int[] data3 = {2, 8, 1, 5, 9};
        System.out.println(findGreatestSumOfSubArray(data));
        System.out.println(findGreatestSumOfSubArray(data2));
        System.out.println(findGreatestSumOfSubArray(data3));
    }
}
```

#

运行结果：



31

求从 1 到 n 的整数中 1 出现的次数

# #

题目：输入一个整数 n 求从 1 到 n 这 n 个整数的十进制表示中 1 出现的次数。

#

举例说明：

例如输入 12 ，从 1 到 12 这些整数中包含 1 的数字有 1、10、11 和 12，1 一共出现了 5 次。

#

题解思路：

#

第一种：不夸虑时间效率的解法

累加 1 到 n 中每个整数 1 出现的次数。我们可以每次通过对 10 求余数判断整数的个位数字是不是 1。如果这个数字大于 10，除以 10 之后再判断个位数字是不是 1 。

#

第二种：从数字规律着手明显提高时间效率的解法

21345 作为例子来分析。我们把从 1 到 21345 的所有数字分为两段， 一段是从 1 到1345，另一段是从 1346 到 21345。

我们先看从 1346 到 21345 中 1 出现的次数。1 的出现分为两种情况。首先分析1出现在最高位（本例中是万位）的情况。从 1346 到 21345 的数字中， 1 出现在 10000～19999 这 10000 个数字的万位中， 一共出现了 10000(10^4)个。

值得注意的是， 并不是对所有 5 位数而言在万位出现的次数都是 10000 个。对于万位是 1 的数字比如输入 12345，1 只出现在 10000～12345 的万位，出现的次数不是 10^4 次，而是 2346 次，也就是除去最高数字之后剩下的数字再加上 1（即2345+1=2346 次）。

接下来分析 1 出现在除最高位之外的其他四位数中的情况。例子中 1346～21345 这 20000 个数字中后 4 位中 1 出现的次数是 2000 次。由于最 高位是 2，我们可以再把 1346～21345 分成两段， 1346～11345 和 1346～21345 。每一段剩下的 4 位数字中， 选择其中一位是 1 ，其余三位可以在 0～9 这 10 个数字中任意选择，因此根据排列组合原则，总共出现的次数是 2*10^3=2000。

至于从 l 到 1345 中 1 出现的次数，我们就可以用递归求得了。这也是我们为什么要把 1～21345 分成 1～ 1345 和 1346～21345 两段的原因。因为把 21345 的最高位去掉就变成 1345 ，便于我们采用递归的思路。

本题采用第二种解法。

#

代码实现：

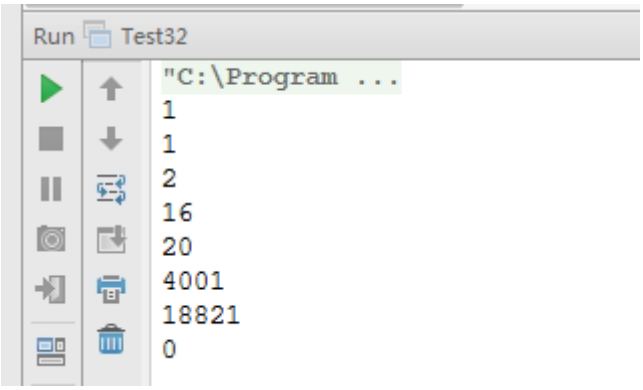
```
public class Test32 {
    /**
     * 题目：输入一个整数n求从1 到n这n个整数的十进制表示中1 出现的次数。
     * @param n 最大的数字
     * @return 1~n中，各个数位1出现的次数
     */
    public static int numberOf1Between1AndN(int n) {
        if (n <= 0) {
            return 0;
        }
        String value = n + "";
        int[] numbers = new int[value.length()];
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = value.charAt(i) - '0';
        }
        return numberOf1(numbers, 0);
    }
    /**
     * 求0~numbers表的数字中的1的个数
     *
     * @param numbers 数字，如{1, 2, 3, 4, 5}表示数字12345
     * @param curIdx  当前处理的位置
     * @return 1的个数
     */
    private static int numberOf1(int[] numbers, int curIdx) {
        if (numbers == null || curIdx >= numbers.length || curIdx < 0) {
            return 0;
        }
```

```
    // 待处理的第一个数字
    int first = numbers[curIdx];
    // 要处理的数字的位数
    int length = numbers.length - curIdx;
    // 如果只有一位且这一位是0返回0
    if (length == 1 && first == 0) {
        return 0;
    }
    // 如果只有一位且这一位不是0返回1
    if (length == 1 && first > 0) {
        return 1;
    }
    // 假设numbers是21345
    // numFirstDigit是数字10000-19999的第一个位中的数目
    int numFirstDigit = 0;
    // 如果最高位不是1，如21345，在[1236, 21345]中，最高位1出现的只在[10000, 19999]中，出现1的次数是10^4方个
    if (first > 1) {
        numFirstDigit = powerBase10(length - 1);
    }
    // 如果最高位是1，如12345，在[2346, 12345]中，最高位1出现的只在[10000, 12345]中，总计2345+1个
    else if (first == 1) {
        numFirstDigit = atoi(numbers, curIdx + 1) + 1;
    }
    // numOtherDigits，是[1346, 21345]中，除了第一位之外（不看21345中的第一位2）的数位中的1的数目
    int numOtherDigits = first * (length - 1) * powerBase10(length - 2);
    // numRecursive是1-1234中1的的数目
    int numRecursive = numberOf1(numbers, curIdx + 1);
    return numFirstDigit + numOtherDigits + numRecursive;
}
/**
 * 将数字数组转换成数值，如{1, 2, 3, 4, 5}，i = 2，结果是345
 * @param numbers 数组
 * @param i 开始黑气的位置
 * @return 转换结果
 */
private static int atoi(int[] numbers, int i) {
    int result = 0;
    for (int j = i; j < numbers.length; j++) {
        result = (result * 10 + numbers[j]);
    }
    return result;
}
/**
 * 求10的n次方，假定n不为负数
 * @param n 幂，非负数
```

```
     * @return 10的n次方
     */
    private static int powerBase10(int n) {
        int result = 1;
        for (int i = 0; i < n; i++) {
            result *= 10;
        }
        return result;
    }
    public static void main(String[] args) {
        System.out.println(numberOf1Between1AndN(1)); // 1
        System.out.println(numberOf1Between1AndN(5)); // 1
        System.out.println(numberOf1Between1AndN(10)); // 2
        System.out.println(numberOf1Between1AndN(55)); // 16
        System.out.println(numberOf1Between1AndN(99)); // 20
        System.out.println(numberOf1Between1AndN(10000)); // 4001
        System.out.println(numberOf1Between1AndN(21345)); // 18821
        System.out.println(numberOf1Between1AndN(0)); // 0
    }
}
```

#

运行结果：



32

把数组排成最小的数

#

题目：输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。

#

例子说明：

例如输入数组{3， 32, 321}，则扫描输出这 3 个数字能排成的最小数字 321323。

#

解题思路：

#

第一种：直观解法

先求出这个数组中所有数字的全排列，然后把每个排列拼起来，最后求出拼起来的数字的最大值。

#

第二种：排序解法

找到一个排序规则，数组根据这个规则排序之后能排成一个最小的数字。要确定排序规则，就要比较两个数字，也就是给出两个数字 m 和 n，我们需要确定一个规则判断 m 和 n 哪个应该排在前面，而不是仅仅比较这两个数字的值哪个更大。

根据题目的要求，两个数字 m 和 n 能拼接成数字 m 和 m。如果 mn < nm，那么我们应该打印出 m，也就是 m 应该排在 n 的前面，我们定义此时 m 小于 n：反之，如果 nm < mn，我们定义 n 小于 m。如果 mn=nm，m 等于 n。在下文中，符号“<”、“>”及“ = ”表示常规意义的数值的大小关系，而文字“大于”、“小于”、“等于”表示我们新定义的大小关系。

接下来考虑怎么去拼接数字，即给出数字 m 和 n，怎么得到数字 m 和 m 并比较它们的大小。直接用数值去计算不难办到，但需要考虑到一个潜在的问题就是 m 和n 都在 int 能表达的范围内，但把它们拼起来的数字 m 和 m 用 int 表示就有可能溢出了，所以这还是一个隐形的大数问题。

一个非常直观的解决大数问题的方法就是把数字转换成字符串。另外，由于把数字 m 和 n 拼接起来得到 m 和 m，它们的位数肯定是相同的，因此比较它们的大小只需要按照字符串大小的比较规则就可以了。

本题采用第二种方法实现。

#

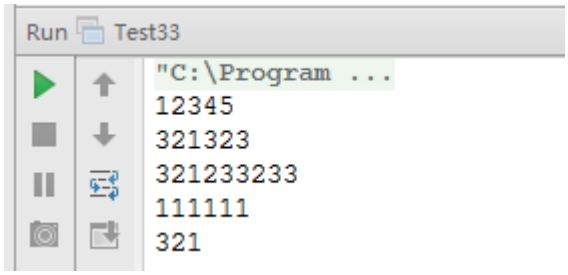
代码实现：

```
public class Test33 {
    /**
     * 自定义的排序比较器，实现算法说明的排序原理
     */
    private static class MComparator implements Comparator<String> {
        @Override
        public int compare(String o1, String o2) {
            if (o1 == null || o2 == null) {
                throw new IllegalArgumentException("Arg should not be null");
            }
            String s1 = o1 + o2;
            String s2 = o2 + o1;
            return s1.compareTo(s2);
        }
    }
    /**
     * 快速排序算法
     *
     * @param array      待排序数组
     * @param start      要排序的起始位置
     * @param end        要排序的结束位置
     * @param comparator 自定义的比较器
     */
    private static void quickSort(String[] array, int start, int end, Comparator<String> comparator) {
        if (start < end) {
            String pivot = array[start];
            int left = start;
            int right = end;
            while (start < end) {
                while (start < end && comparator.compare(array[end], pivot) >= 0) {
                    end--;
                }
                array[start] = array[end];
                while (start < end && comparator.compare(array[start], pivot) <= 0) {
```

```
                start++;
            }
            array[end] = array[start];
        }
        array[start] = pivot;
        quickSort(array, left, start - 1, comparator);
        quickSort(array, start + 1, end, comparator);
    }
}
/**
 * 题目：输入一个正整数数组，把数组里所有数字拼接起来排成一个数，
 * 打印能拼接出的所有数字中最小的一个。
 * @param array 输入的数组
 * @return 输出结果
 */
public static String printMinNumber(String[] array) {
    if (array == null || array.length < 1) {
        throw new IllegalArgumentException("Array must contain value");
    }
    MComparator comparator = new MComparator();
    quickSort(array, 0, array.length - 1, comparator);
    StringBuilder builder = new StringBuilder(256);
    for (String s : array) {
        builder.append(s);
    }
    return builder.toString();
}
public static void main(String[] args) {
    String[] data = {"3", "5", "1", "4", "2"};
    System.out.println(printMinNumber(data));
    String[] data2 = {"3", "32", "321"};
    System.out.println(printMinNumber(data2));
    String[] data3 = {"3", "323", "32123"};
    System.out.println(printMinNumber(data3));
    String[] data4 = {"1", "11", "111"};
    System.out.println(printMinNumber(data4));
    String[] data5 = {"321"};
    System.out.println(printMinNumber(data5));
}
}
```

#

运行结果：


Run
Test33
"C:\Program ...
12345
321323
321233233
111111
321

33

丑数

#

题目：我们把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。求从小到大的顺序的第 1500个丑数。

#

举例说明：

例如 6、8 都是丑数，但 14 不是，它包含因子 7。习惯上我们把 1 当做第一个丑数。

#

解题思路：

#

第一种：逐个判断每个数字是不是丑数的解法，直观但不够高效。

#

第二种：创建数组保存已经找到丑数，用空间换时间的解法。

根据丑数的定义， 丑数应该是另一个丑数乘以 2、3 或者 5 的结果（1 除外）。因此我们可以创建一个数组，里面的数字是排好序的丑数，每一个丑数都是前面的丑数乘以 2、3 或者 5 得到的。

这种思路的关键在于怎样确保数组里面的丑数是排好序的。假设数组中已经有若干个丑数排好序后存放在数组中，并且把已有最大的丑数记做M，我们接下来分析如何生成下一个丑数。该丑数肯定是前面某一个丑数乘以 2、3 或者 5 的结果， 所以我们首先考虑把已有的每个丑数乘以 2。在乘以 2 的时钝能得到若干个小于或等于 M 的结果。由于是按照顺序生成的，小于或者等于 M 肯定已经在数组中了，我们不需再次考虑：还会得到若干个大于 M 的结果，但我们只需要第一个大于 M 的结果，因为我们希望丑数是按从小到大的顺序生成的，其他更大的结果以后再说。我们把得到的第一个乘以 2 后大于 M 的结果记为 M2，同样，我们把已有的每一个丑数乘以 3 和 5，能得到第一个大于 M 的结果 M3 和 M，那么下一个丑数应该是 M2、M3 和 M5 这 3 个数的最小者。

前面分析的时候，提到把已有的每个丑数分别都乘以 2、3 和 5。事实上这不是必须的，因为已有的丑数是按顺序存放在数组中的。对乘以 2 而言， 肯定存在某一个丑数 T2，排在它之前的每一个丑数乘以 2 得到的结果都会小

于已有最大的丑数，在它之后的每一个丑数乘以 2 得到的结果都会太大。我们只需记下这个丑数的位置， 同时每次生成新的丑数的时候，去更新这个 T2。对乘以 3 和 5 而言， 也存在着同样的 T3 和 T5。

本题实现了两种方法。

#

代码实现：

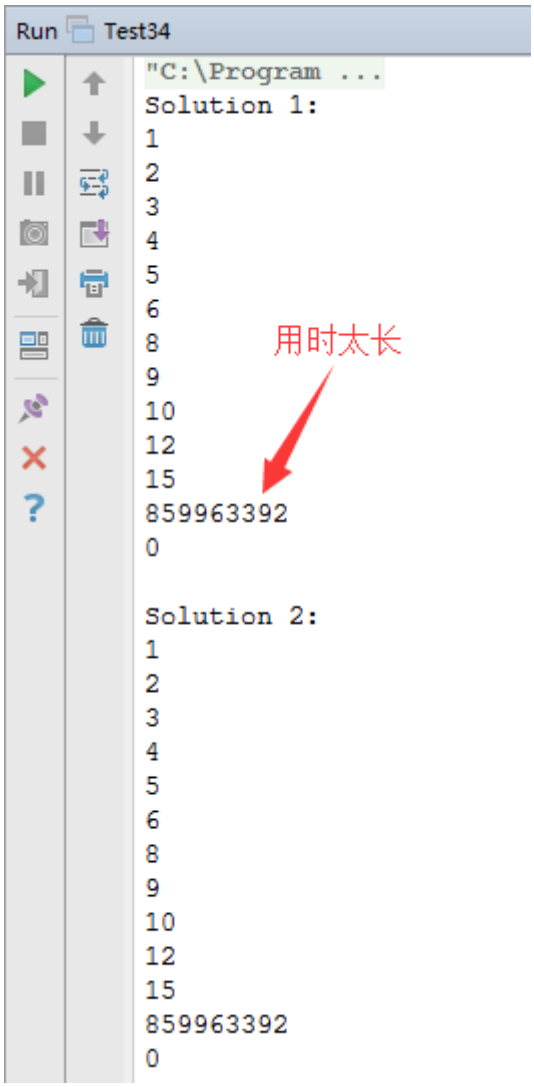
```
public class Test34 {
    /**
     * 判断一个数是否只有2，3，5因子（丑数）
     *
     * @param num 待判断的数，非负
     * @return true是丑数，false丑数
     */
    private static boolean isUgly(int num) {
        while (num % 2 == 0) {
            num /= 2;
        }
        while (num % 3 == 0) {
            num /= 3;
        }
        while (num % 5 == 0) {
            num /= 5;
        }
        return num == 1;
    }
    /**
     * 找第index个丑数，速度太慢
     *
     * @param index 第index个丑数
     * @return 对应的丑数值
     */
    public static int getUglyNumber(int index) {
        if (index <= 0) {
            return 0;
        }
        int num = 0;
        int uglyFound = 0;
        while (uglyFound < index) {
            num++;
            if (isUgly(num)) {
```

```
            ++uglyFound;
        }
    }
    return num;
}
/**
 * 找第index个丑数，【第二种方法】
 *
 * @param index 第index个丑数
 * @return 对应的丑数值
 */
public static int getUglyNumber2(int index) {
    if (index <= 0) {
        return 0;
    }
    int[] pUglyNumbers = new int[index];
    pUglyNumbers[0] = 1;
    int nextUglyIndex = 1;
    int p2 = 0;
    int p3 = 0;
    int p5 = 0;
    while (nextUglyIndex < index) {
        int min = min(pUglyNumbers[p2] * 2, pUglyNumbers[p3] * 3, pUglyNumbers[p5] * 5);
        pUglyNumbers[nextUglyIndex] = min;
        while (pUglyNumbers[p2] * 2 <= pUglyNumbers[nextUglyIndex]) {
            p2++;
        }
        while (pUglyNumbers[p3] * 3 <= pUglyNumbers[nextUglyIndex]) {
            p3++;
        }
        while (pUglyNumbers[p5] * 5 <= pUglyNumbers[nextUglyIndex]) {
            p5++;
        }
        nextUglyIndex++;
    }
    return pUglyNumbers[nextUglyIndex - 1];
}
private static int min(int n1, int n2, int n3) {
    int min = n1 < n2 ? n1 : n2;
    return min < n3 ? min : n3;
}
public static void main(String[] args) {
    System.out.println("Solution 1:");
    test1();
    System.out.println();
```

```
        System.out.println("Solution 2:");
        test2();
    }
    private static void test1() {
        System.out.println(getUglyNumber(1)); // 1
        System.out.println(getUglyNumber(2)); // 2
        System.out.println(getUglyNumber(3)); // 3
        System.out.println(getUglyNumber(4)); // 4
        System.out.println(getUglyNumber(5)); // 5
        System.out.println(getUglyNumber(6)); // 6
        System.out.println(getUglyNumber(7)); // 8
        System.out.println(getUglyNumber(8)); // 9
        System.out.println(getUglyNumber(9)); // 10
        System.out.println(getUglyNumber(10)); // 12
        System.out.println(getUglyNumber(11)); // 15
        System.out.println(getUglyNumber(1500)); // 859963392
        System.out.println(getUglyNumber(0)); // 0
    }
    private static void test2() {
        System.out.println(getUglyNumber2(1)); // 1
        System.out.println(getUglyNumber2(2)); // 2
        System.out.println(getUglyNumber2(3)); // 3
        System.out.println(getUglyNumber2(4)); // 4
        System.out.println(getUglyNumber2(5)); // 5
        System.out.println(getUglyNumber2(6)); // 6
        System.out.println(getUglyNumber2(7)); // 8
        System.out.println(getUglyNumber2(8)); // 9
        System.out.println(getUglyNumber2(9)); // 10
        System.out.println(getUglyNumber2(10)); // 12
        System.out.println(getUglyNumber2(11)); // 15
        System.out.println(getUglyNumber2(1500)); // 859963392
        System.out.println(getUglyNumber2(0)); // 0
    }
}
```

#

运行结果：


Run
Test34
"C:\Program ...
Solution 1:
1
2
3
4
5
6
8
9
10
12
15
859963392
0
用时太长
Solution 2:
1
2
3
4
5
6
8
9
10
12
15
859963392
0

34

第一个只出现一次的字符

# #

题目：在字符串中找出第一个只出现一次的字符。

#

解题思路：

#

第一种：直接求解：

从头开始扫描这个字符串中的每个字符。当访问到某字符时拿这个字符和后面的每个字符相比较，如果在后面没有发现重复的字符，则该字符就是只出现一次的字符。如果字符串有 n 个字符，每个字符可能与后面的 O(n ）个字符相比较，因此这种思路的时间复杂度是 O(n^2)。

#

第二种：记录法

由于题目与字符出现的次数相关， 我们是不是可以统计每个字符在该字符串中出现的次数？要达到这个目的，我们需要一个数据容器来存放每个字符的出现次数。在这个数据容器中可以根据字符来查找它出现的次数，也就是说这个容器的作用是把一个字符映射成二个数字。在常用的数据容器中， 哈希表正是这个用途。

为了解决这个问题，我们可以定义哈希表的键（ Key ）是字符， 而值(Value ）是该字符出现的次数。同时我们还需要从头开始扫描字符串两次。第一次扫描字符串时，每扫描到一个字符就在哈希表的对应项中把次数加 1 。接下来第二次扫描时， 每扫描到一个字符就能从哈希表中得到该字符出现的次数。这样第一个只出现一次的字符就是符合要求的输出。

第一次扫描时，在哈希表中更新一个字符出现的次数的时间是 O(n) 。如果字符串长度为 n， 那么第一次扫描的时间复杂度是 O(n)。第二次扫描时，同样 0(1)能读出一个字符出现的次数，所以时间复杂度仍然是 O(n)。这样算起来，总的时间复杂度是 O(n)。

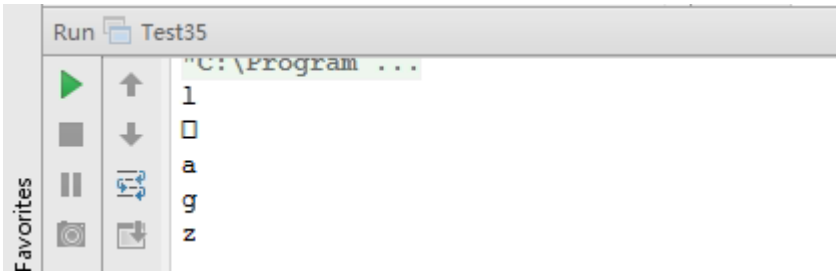
#

代码实现：

```
public class Test35 {
    public static char firstNotRepeatingChar(String s) {
        if (s == null || s.length() < 1) {
            throw new IllegalArgumentException("Arg should not be null or empty");
        }
        Map<Character, Integer> map = new LinkedHashMap<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (map.containsKey(c)) {
                map.put(c, -2);
            } else {
                map.put(c, i);
            }
        }
        Set<Map.Entry<Character, Integer>> entrySet = map.entrySet();
        // 记录只出现一次的字符的索引
        int idx = Integer.MAX_VALUE;
        // 记录只出现一次的字符
        char result = '\0';
        // 找最小索引对应的字符
        for (Map.Entry<Character, Integer> entry : entrySet) {
            if (entry.getValue() >= 0 && entry.getValue() < idx) {
                idx = entry.getValue();
                result = entry.getKey();
            }
        }
        return result;
    }
    public static void main(String[] args) {
        System.out.println(firstNotRepeatingChar("google")); // l
        System.out.println(firstNotRepeatingChar("aabccdbd")); // '\0'
        System.out.println(firstNotRepeatingChar("abcdefg")); // a
        System.out.println(firstNotRepeatingChar("gfedcba")); // g
        System.out.println(firstNotRepeatingChar("zgfedcba")); // g
    }
}
```

#

运行结果：



35

数组中的逆序对

# #

题目：在数组中的两个数字如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。

#

举例分析

例如在数组 { 7, 5, 6, 4}中， 一共存在 5 个逆序对，分别是 (7, 6) 、 (7， 5) ，(7, 4) 、 (6, 4) 和 (5, 4) 。

#

解题思路：

#

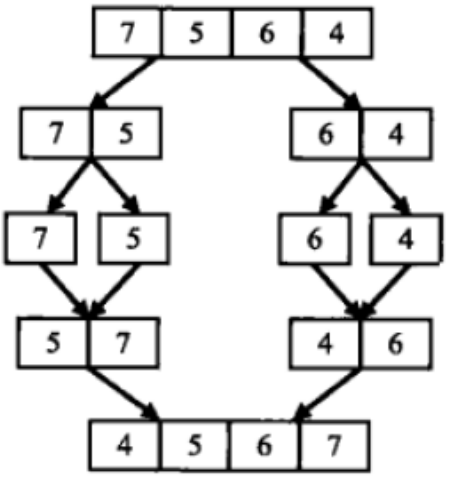
第一种：直接求解

顺序扫描整个数组。每扫描到一个数字的时候，逐个比较该数字和它后面的数字的大小。如果后面的数字比它小，则这两个数字就组成了一个逆序对。假设数组中含有 n 个数字。由于每个数字都要和 O(n) 个数字作比较，因此这个算法的时间复杂度是 O(n^2)。

#

第二种：分析法

我们以数组 { 7, 5, 6, 4 } 为例来分析统计逆序对的过程。每次扫描到一个数字的时候，我们不能拿它和后面的每一个数字作比较，否则时间复杂度就是 O(n^5)，因此我们可以考虑先比较两个相邻的数字。



（a）把长度为 4 的数组分解成两个长度为 2 的子数组

（b） 把长度为 2 的数组分解成两个长度为 1 的子数组

（c） 把长度为 1 的子数组合并、排序，并统计逆序对

（d） 把长度为 2 的子数组合并、排序，并统计逆序对

图 5.1 统计数组{7, 5, 6, 4} 中逆序对的过程

如图 5.1 (a)和图 5.1 (b) 所示，我们先把数组分解成两个长度为 2 的子数组， 再把这两个子数组分别拆分成两个长度为 1 的子数组。接下来一边合并相邻的子数组， 一边统计逆序对的数目。在第一对长度为 1 的子数组 { 7 } 、{ 5 } 中7 大于 5 ， 因此 (7, 5) 组成一个逆序对。同样在第二对长度为 1 的子数组 { 6 } 、{ 4 } 中也有逆序对 (6, 4) 。由于我们已经统计了这两对子数组内部的逆序对，因此需要把这两对子数组排序 (图 5.1 (c) 所示) ，以免在以后的统计过程中再重复统计。

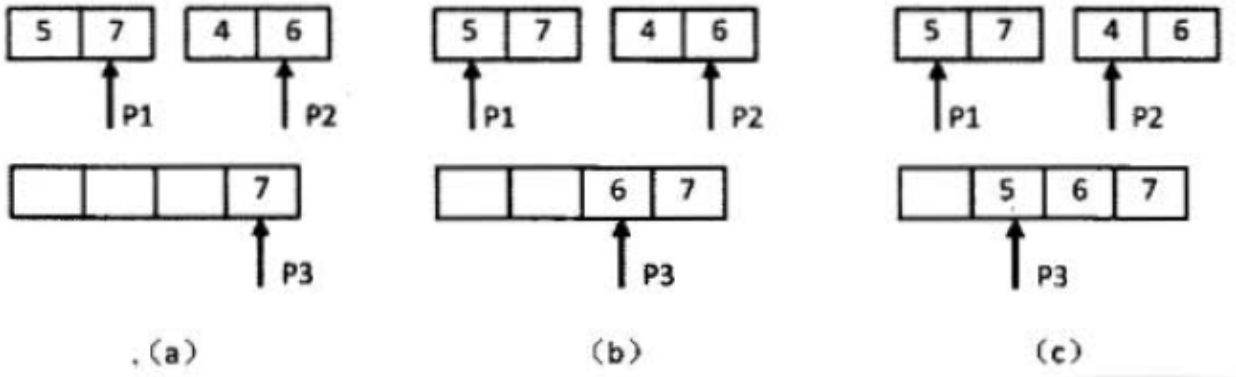


图 5.2 图 5.1（d）中合并两个子数组并统计逆序对的过程

注：图中省略了最后一步， 即复制第二个子数组最后剩余的 4 到辅助数组中。

- (a) P1 指向的数字大于 P2指向的数字，表明数组中存在逆序对。P2 指向的数字是第二个子数组的第二个数字， 因此第二个子数组中有两个数字比 7 小. 把逆序对数目加 2，并把 7 复制到辅助数组，向前移动 P1 和 P3。
- (b) P1 指向的数字小子 P2 指向的数字，没有逆序对。把 P2 指向的数字复制到辅助数组，并向前移动 P2 和 P3 。

- (c) P1 指向的数字大于 P2 指向的数字，因此存在逆序对。由于 P2 指向的数字是第二个子数组的第一个数字，子数组中只有一个数字比 5 小. 把逆序对数目加 1，并把 5 复制到辅助数组，向前移动 P1 和 P3。

接下来我们统计两个长度为 2 的子数组之间的逆序对。我们在图 5.2 中细分图 5.1（d）的合并子数组及统计逆序对的过程。

我们先用两个指针分别指向两个子数组的末尾，并每次比较两个指针指向的数字。如果第一个子数组中的数字大于第二个子数组中的数字，则构成逆序对，并且逆序对的数目等于第二个子数组中剩余数字的个数（如图 5.2(a)和图 5.2 (c)所示）。如果第一个数组中的数字小于或等于第二个数组中的数字，则不构成逆序对（如图 5.2(b)所示〉。每一次比较的时候，我们都把较大的数字从 · 后往前复制到一个辅助数组中去，确保辅助数组中的数字是递增排序的。在把较大的数字复制到辅助数组之后，把对应的指针向前移动一位，接下来进行下一轮比较。

经过前面详细的诗论， 我们可以总结出统计逆序对的过程：先把数组分隔成子数组， 先统计出子数组内部的逆序对的数目，然后再统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。如果对排序贺，法很熟悉，我们不难发现这个排序的过程实际上就是归并排序。

#

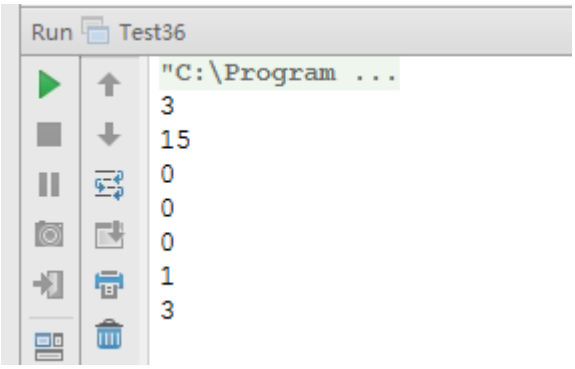
代码实现：

```
public class Test36 {
    public static int inversePairs(int[] data) {
        if (data == null || data.length < 1) {
            throw new IllegalArgumentException("Array arg should contain at least a value");
        }
        int[] copy = new int[data.length];
        System.arraycopy(data, 0, copy, 0, data.length);
        return inversePairsCore(data, copy, 0, data.length - 1);
    }
    private static int inversePairsCore(int[] data, int[] copy, int start, int end) {
        if (start == end) {
            copy[start] = data[start];
            return 0;
        }
        int length = (end - start) / 2;
        int left = inversePairsCore(copy, data, start, start + length);
        int right = inversePairsCore(copy, data, start + length + 1, end);
        // 前半段的最后一个数字的下标
        int i = start + length;
        // 后半段最后一个数字的下标
        int j = end;
```

```
        // 开始拷贝的位置
        int indexCopy = end;
        // 逆序数
        int count = 0;
        while (i >= start && j >= start + length + 1) {
            if (data[i] > data[j]) {
                copy[indexCopy] = data[i];
                indexCopy--;
                i--;
                count += j - (start + length); // 对应的逆序数
            } else {
                copy[indexCopy] = data[j];
                indexCopy--;
                j--;
            }
        }
        for (; i >= start; i--) {
            copy[indexCopy] = data[i];
            indexCopy--;
            i--;
        }
        for (; j >= start + length + 1; j--) {
            copy[indexCopy] = data[j];
            indexCopy--;
            j--;
        }
        return count + left + right;
    }
    public static void main(String[] args) {
        int[] data = {1, 2, 3, 4, 7, 6, 5};
        System.out.println(inversePairs(data)); // 3
        int[] data2 = {6, 5, 4, 3, 2, 1};
        System.out.println(inversePairs(data2)); //  15
        int[] data3 = {1, 2, 3, 4, 5, 6};
        System.out.println(inversePairs(data3)); // 0
        int[] data4 = {1};
        System.out.println(inversePairs(data4)); // 0
        int[] data5 = {1, 2};
        System.out.println(inversePairs(data5)); // 0
        int[] data6 = {2, 1};
        System.out.println(inversePairs(data6)); // 1
        int[] data7 = {1, 2, 1, 2, 1};
        System.out.println(inversePairs(data7)); // 3
    }
}
```

#

运行结果：



36

两个链表的第一个公共结点

# #

题目：输入两个链表，找出它们的第一个公共结点。

#

链表结点定义

```
/**
 * 链表结点类
 */
private static class ListNode {
    int val;
    ListNode next;
    public ListNode() {
    }
    public ListNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

解题思路：

#

第一种：直接法

在第一个链表上顺序遍历每个结点，每遍历到一个结点的时候，在第二个链表上顺序遍历每个结点。如果在第二个链表上有一个结点和第一个链表上的结点一样，说明两个链表在这个结点上重合，于是就找到了它们的公共结点。如果第一个链表的长度为 m，第二个链表的长度为 n，显然该方法的时间复杂度是 O(mn ）。

#

第二种：使用栈

所以两个有公共结点而部分重舍的链衰，拓扑形状看起来像一个 Y， 而不可能像 X（如图 5.3 所示）。

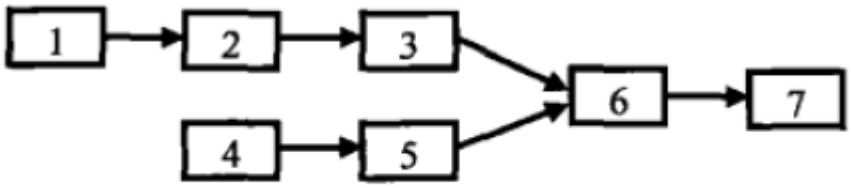


图 5.3　两个链表在值为 6 的结点处交汇

经过分析我们发现，如果两个链表有公共结点，那么公共结点出现在两个链表的尾部。如果我们从两个链衰的尾部开始往前比较，最后一个相同的结点就是我们要找的结点。

在上述思路中，我们需要用两个辅助钱。如果链表的长度分别为 m 和 n，那么空间复杂度是 O(m+n）。这种思路的时间复杂度也是 O(m+n）。和最开始的蛮力法相比，时间效率得到了提高，相当于是用空间消耗换取了时间效率。

#

第三种：先行法

在图 5.3 的两个链表中，我们可以先遍历一次得到它们的长度分别为 5 和 4， 也就是较长的链表与较短的链表相比多一个结点。第二次先在长的链表上走 1 步，到达结点 2。接下来分别从结点 2 和结点 4 出发同时遍历两个结点， 直到找到它们第一个相同的结点 6，这就是我们想要的结果。

第三种思路和第二种思路相比，时间复杂度都是 O(m＋n)， 但我们不再需要辅助的拢，因此提高了空间效率。

本题采用第三种解法。

#

代码实现

```
public class Test37 {
    /**
```

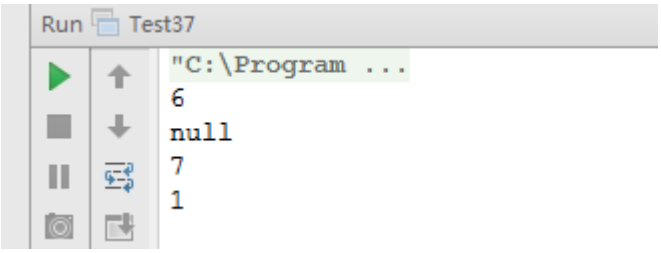
```
 * 链表结点类
 */
private static class ListNode {
    int val;
    ListNode next;
    public ListNode() {
    }
    public ListNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
/**
 * 找两个结点的第一个公共结点，如果没有找到返回null，方法比较好，考虑了两个链表中有null的情况
 *
 * @param head1 第一个链表
 * @param head2 第二个链表
 * @return 找到的公共结点，没有返回null
 */
public static ListNode findFirstCommonNode(ListNode head1, ListNode head2) {
    int length1 = getListLength(head1);
    int length2 = getListLength(head2);
    int diff = length1 - length2;
    ListNode longListHead = head1;
    ListNode shortListHead = head2;
    if (diff < 0) {
        longListHead = head2;
        shortListHead = head1;
        diff = length2 - length1;
    }
    for (int i = 0; i < diff; i++) {
        longListHead = longListHead.next;
    }
    while (longListHead != null && shortListHead != null && longListHead != shortListHead) {
        longListHead = longListHead.next;
        shortListHead = shortListHead.next;
    }
    // 返回第一个相同的公共结点，如果没有返回null
    return longListHead;
}
private static int getListLength(ListNode head) {
    int result = 0;
```

```
    while (head != null) {
        result++;
        head = head.next;
    }
    return result;
}
public static void main(String[] args) {
    test1();
    test2();
    test3();
    test4();
}
private static void test1() {
    // 第一个公共结点在链表中间
    // 1 - 2 - 3 \
    //           6 - 7
    //     4 - 5 /
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(2);
    ListNode n3 = new ListNode(3);
    ListNode n4 = new ListNode(4);
    ListNode n5 = new ListNode(5);
    ListNode n6 = new ListNode(6);
    ListNode n7 = new ListNode(7);
    n1.next = n2;
    n2.next = n3;
    n3.next = n6;
    n6.next = n7;
    n4.next = n5;
    n5.next = n6;
    System.out.println(findFirstCommonNode(n1, n4)); // 6
}
private static void test2() {
    // 没有公共结点
    // 1 - 2 - 3 - 4
    //
    // 5 - 6 - 7
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(2);
    ListNode n3 = new ListNode(3);
    ListNode n4 = new ListNode(4);
    ListNode n5 = new ListNode(5);
    ListNode n6 = new ListNode(6);
    ListNode n7 = new ListNode(7);
    n1.next = n2;
```

```
        n2.next = n3;
        n3.next = n4;
        n5.next = n6;
        n6.next = n7;
        System.out.println(findFirstCommonNode(n1, n5)); // null
    }
    private static void test3() {
        // 公共结点是最后一个结点
        // 1 - 2 - 3 - 4 \
        //               7
        //         5 - 6 /
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(2);
        ListNode n3 = new ListNode(3);
        ListNode n4 = new ListNode(4);
        ListNode n5 = new ListNode(5);
        ListNode n6 = new ListNode(6);
        ListNode n7 = new ListNode(7);
        n1.next = n2;
        n2.next = n3;
        n3.next = n4;
        n4.next = n7;
        n5.next = n6;
        n6.next = n7;
        System.out.println(findFirstCommonNode(n1, n5)); // 7
    }
    private static void test4() {
        // 公共结点是第一个结点
        // 1 - 2 - 3 - 4 - 5
        // 两个链表完全重合
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(2);
        ListNode n3 = new ListNode(3);
        ListNode n4 = new ListNode(4);
        ListNode n5 = new ListNode(5);
        ListNode n6 = new ListNode(6);
        ListNode n7 = new ListNode(7);
        n1.next = n2;
        n2.next = n3;
        n3.next = n4;
        n4.next = n5;
        System.out.println(findFirstCommonNode(n1, n1)); // 1
    }
}
```

#

运行结果



37

数字在排序数组中出现的次数

# #

题目：统计一个数字：在排序数组中出现的次数。

#

举例说明

例如输入排序数组 { 1, 2, 3, 3, 3, 3, 4, 5 } 和数字 3 ，由于 3 在这个数组中出现了 4 次，因此输出 4 。

#

解题思路

利用改进的二分算法。

如何用二分查找算法在数组中找到第一个 k，二分查找算法总是先拿数组中间的数字和 k 作比较。如果中间的数字比 k 大，那么 k 只有可能出现在数组的前半段，下一轮我们只在数组的前半段查找就可以了。如果中间的数字比 k 小，那么 k 只有可能出现在数组的后半段，下一轮我们只在数组的后半乓查找就可以了。如果中间的数字和 k 相等呢？我们先判断这个数字是不是第一个 k。如果位于中间数字的前面一个数字不是 k,此时中间的数字刚好就是第一个 k。如果中间数字的前面一个数字也是 k，也就是说第一个 k 肯定在数组的前半段， 下一轮我们仍然需要在数组的前半段查找。

同样的思路在排序数组中找到最后一个 k。如果中间数字比 k 大，那么 k 只能出现在数组的前半段。如果中间数字比 k 小，k 就只能出现在数组的后半段。如果中间数字等于 k 呢？我们需要判断这个 k 是不是最后一个 k，也就是中间数字的下一个数字是不是也等于 k。如果下一个数字不是 k，则中间数字就是最后一个 k 了，否则下一轮我们还是要在数组的后半段中去查找。

#

代码实现

```
public class Test38 {
    /**
     * 找排序数组中k第一次出现的位置
     *
```

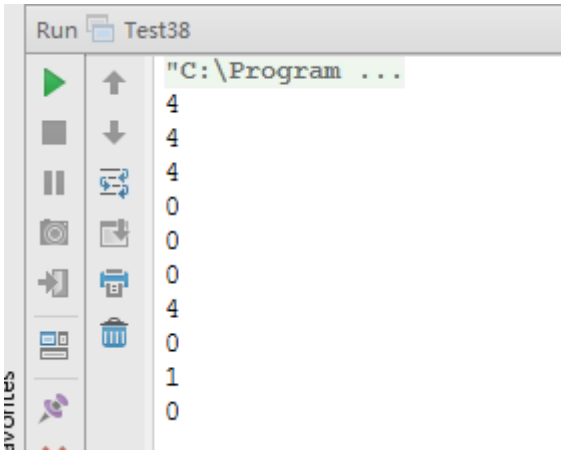
```
     * @param data
     * @param k
     * @param start
     * @param end
     * @return
     */
    private static int getFirstK(int[] data, int k, int start, int end) {
        if (data == null || data.length < 1 || start > end) {
            return -1;
        }
        int midIdx = start + (end - start) / 2;
        int midData = data[midIdx];
        if (midData == k) {
            if (midIdx > 0 && data[midIdx - 1] != k || midIdx == 0) {
                return midIdx;
            } else {
                end = midIdx - 1;
            }
        } else if (midData > k) {
            end = midIdx - 1;
        } else {
            start = midIdx + 1;
        }
        return getFirstK(data, k, start, end);
    }
    /**
     * 找排序数组中k最后一次出现的位置
     *
     * @param data
     * @param k
     * @param start
     * @param end
     * @return
     */
    private static int getLastK(int[] data, int k, int start, int end) {
        if (data == null || data.length < 1 || start > end) {
            return -1;
        }
        int midIdx = start + (end - start) / 2;
        int midData = data[midIdx];
        if (midData == k) {
            if (midIdx + 1 < data.length && data[midIdx + 1] != k || midIdx == data.length - 1) {
                return midIdx;
            } else {
                start = midIdx + 1;
```

```
        }
    } else if (midData < k) {
        start = midIdx + 1;
    } else {
        end = midIdx - 1;
    }
    return getLastK(data, k, start, end);
}
/**
 * 题目：统计一个数字：在排序数组中出现的次数
 * @param data
 * @param k
 * @return
 */
public static int getNumberOfK(int[] data, int k) {
    int number = 0;
    if (data != null && data.length > 0) {
        int first = getFirstK(data, k, 0, data.length - 1);
        int last = getLastK(data, k, 0, data.length - 1);
        if (first > -1 && last > -1) {
            number = last - first + 1;
        }
    }
    return number;
}
public static void main(String[] args) {
    // 查找的数字出现在数组的中间
    int[] data1 = {1, 2, 3, 3, 3, 3, 4, 5};
    System.out.println(getNumberOfK(data1, 3)); // 4
    // 查找的数组出现在数组的开头
    int[] data2 = {3, 3, 3, 3, 4, 5};
    System.out.println(getNumberOfK(data2, 3)); // 4
    // 查找的数组出现在数组的结尾
    int[] data3 = {1, 2, 3, 3, 3, 3};
    System.out.println(getNumberOfK(data3, 3)); // 4
    // 查找的数字不存在
    int[] data4 = {1, 3, 3, 3, 3, 4, 5};
    System.out.println(getNumberOfK(data4, 2)); // 0
    // 查找的数字比第一个数字还小，不存在
    int[] data5 = {1, 3, 3, 3, 3, 4, 5};
    System.out.println(getNumberOfK(data5, 0)); // 0
    // 查找的数字比最后一个数字还大，不存在
    int[] data6 = {1, 3, 3, 3, 3, 4, 5};
    System.out.println(getNumberOfK(data6, 0)); // 0
    // 数组中的数字从头到尾都是查找的数字
```

```
        int[] data7 = {3, 3, 3, 3};
        System.out.println(getNumberOfK(data7, 3)); // 4
        // 数组中的数字从头到尾只有一个重复的数字，不是查找的数字
        int[] data8 = {3, 3, 3, 3};
        System.out.println(getNumberOfK(data8, 4)); // 0
        // 数组中只有一个数字，是查找的数字
        int[] data9 = {3};
        System.out.println(getNumberOfK(data9, 3)); // 1
        // 数组中只有一个数字，不是查找的数字
        int[] data10 = {3};
        System.out.println(getNumberOfK(data10, 4)); // 0
    }
}
```

#

运行结果



38

二叉树的深度

# #

题目一：输入一棵二叉树的根结点，求该树的深度。从根结点到叶子点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

#

二叉树的结点定义

```
private static class BinaryTreeNode {
    int val;
    BinaryTreeNode left;
    BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
}
```

#

解题思路

如果一棵树只有一个结点，它的深度为。 如果根结点只有左子树而没有右子树， 那么树的深度应该是其左子树的深度加 1，同样如果根结点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加 1。如果既有右子树又有左子树， 那该树的深度就是其左、右子树深度的较大值再加 1。 比如在图 6.1 的二叉树中，根结点为 1 的树有左右两个子树，其左右子树的根结点分别为结点 2 和 3。根结点为 2 的左子树的深度为 3， 而根结点为 3 的右子树的深度为 2，因此根结点为 1 的树的深度就是 4 。

这个思路用递归的方法很容易实现， 只儒对遍历的代码稍作修改即可。

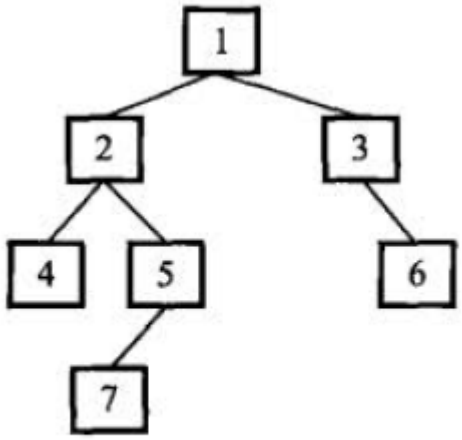


图 6.1　深度为 4 的二叉树

#

代码实现

```
public static int treeDepth(BinaryTreeNode root) {
    if (root == null) {
        return 0;
    }
    int left = treeDepth(root.left);
    int right = treeDepth(root.right);
    return left > right ? (left + 1) : (right + 1);
}
```

#

题目二：输入一棵二叉树的根结点，判断该树是不是平衡二叉树。如果某二叉树中任意结点的左右子树的深度相差不超过 1，那么它就是一棵平衡二叉树。

#

解题思路

#

解法一：需要重蟹遍历结点多次的解法

在遍历树的每个结点的时候，调用函数 treeDepth 得到它的左右子树的深度。如果每个结点的左右子树的深度相差都不超过 1，按照定义它就是一棵平衡的二叉树。

```
public static boolean isBalanced(BinaryTreeNode root) {
    if (root == null) {
        return true;
    }
    int left = treeDepth(root.left);
    int right = treeDepth(root.right);
    int diff = left − right;
    if (diff > 1 || diff < −1) {
        return false;
    }
    return isBalanced(root.left) && isBalanced(root.right);
}
```

#

解法二：每个结点只遍历一次的解法

用后序遍历的方式遍历二叉树的每一个结点，在遍历到一个结点之前我们就已经遍历了它的左右子树。只要在遍历每个结点的时候记录它的深度（某一结点的深度等于它到叶节点的路径的长度），我们就可以一边遍历一边判断每个结点是不是平衡的。

```
/**
 * 判断是否是平衡二叉树，第二种解法
 *
 * @param root
 * @return
 */
public static boolean isBalanced2(BinaryTreeNode root) {
    int[] depth = new int[1];
    return isBalancedHelper(root, depth);
}
public static boolean isBalancedHelper(BinaryTreeNode root, int[] depth) {
    if (root == null) {
        depth[0] = 0;
        return true;
    }
    int[] left = new int[1];
    int[] right = new int[1];
    if (isBalancedHelper(root.left, left) && isBalancedHelper(root.right, right)) {
        int diff = left[0] - right[0];
        if (diff >= -1 && diff <= 1) {
            depth[0] = 1 + (left[0] > right[0] ? left[0] : right[0]);
            return true;
        }
    }
    return false;
}
```

#

完整代码

```
public class Test39 {
    private static class BinaryTreeNode {
        int val;
        BinaryTreeNode left;
        BinaryTreeNode right;
        public BinaryTreeNode() {
        }
        public BinaryTreeNode(int val) {
            this.val = val;
        }
    }
    public static int treeDepth(BinaryTreeNode root) {
        if (root == null) {
```

```
        return 0;
    }
    int left = treeDepth(root.left);
    int right = treeDepth(root.right);
    return left > right ? (left + 1) : (right + 1);
}
/**
 * 判断是否是平衡二叉树，第一种解法
 *
 * @param root
 * @return
 */
public static boolean isBalanced(BinaryTreeNode root) {
    if (root == null) {
        return true;
    }
    int left = treeDepth(root.left);
    int right = treeDepth(root.right);
    int diff = left - right;
    if (diff > 1 || diff < -1) {
        return false;
    }
    return isBalanced(root.left) && isBalanced(root.right);
}
/**
 * 判断是否是平衡二叉树，第二种解法
 *
 * @param root
 * @return
 */
public static boolean isBalanced2(BinaryTreeNode root) {
    int[] depth = new int[1];
    return isBalancedHelper(root, depth);
}
public static boolean isBalancedHelper(BinaryTreeNode root, int[] depth) {
    if (root == null) {
        depth[0] = 0;
        return true;
    }
    int[] left = new int[1];
    int[] right = new int[1];
    if (isBalancedHelper(root.left, left) && isBalancedHelper(root.right, right)) {
        int diff = left[0] - right[0];
        if (diff >= -1 && diff <= 1) {
            depth[0] = 1 + (left[0] > right[0] ? left[0] : right[0]);
```

```
            return true;
        }
    }
    return false;
}
public static void main(String[] args) {
    test1();
    test2();
    test3();
    test4();
}
// 完全二叉树
//           1
//        /     \
//       2       3
//      /\      / \
//     4  5    6   7
private static void test1() {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(1);
    BinaryTreeNode n3 = new BinaryTreeNode(1);
    BinaryTreeNode n4 = new BinaryTreeNode(1);
    BinaryTreeNode n5 = new BinaryTreeNode(1);
    BinaryTreeNode n6 = new BinaryTreeNode(1);
    BinaryTreeNode n7 = new BinaryTreeNode(1);
    n1.left = n2;
    n1.right = n3;
    n2.left = n4;
    n2.right = n5;
    n3.left = n6;
    n3.right = n7;
    System.out.println(isBalanced(n1));
    System.out.println(isBalanced2(n1));
    System.out.println("----------------");
}
// 不是完全二叉树，但是平衡二叉树
//           1
//        /     \
//       2       3
//      /\       \
//     4  5       6
//       /
//      7
private static void test2() {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
```

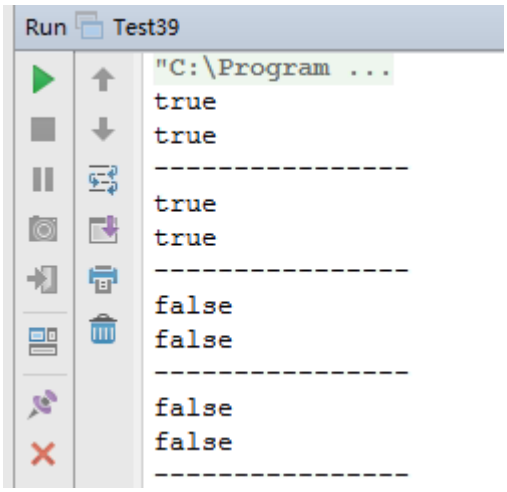
```
    BinaryTreeNode n2 = new BinaryTreeNode(1);
    BinaryTreeNode n3 = new BinaryTreeNode(1);
    BinaryTreeNode n4 = new BinaryTreeNode(1);
    BinaryTreeNode n5 = new BinaryTreeNode(1);
    BinaryTreeNode n6 = new BinaryTreeNode(1);
    BinaryTreeNode n7 = new BinaryTreeNode(1);
    n1.left = n2;
    n1.right = n3;
    n2.left = n4;
    n2.right = n5;
    n5.left = n7;
    n3.right = n6;
    System.out.println(isBalanced(n1));
    System.out.println(isBalanced2(n1));
    System.out.println("----------------");
}
// 不是平衡二叉树
//           1
//        /     \
//       2       3
//      /\
//     4  5
//       /
//      7
private static void test3() {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(1);
    BinaryTreeNode n3 = new BinaryTreeNode(1);
    BinaryTreeNode n4 = new BinaryTreeNode(1);
    BinaryTreeNode n5 = new BinaryTreeNode(1);
    BinaryTreeNode n6 = new BinaryTreeNode(1);
    BinaryTreeNode n7 = new BinaryTreeNode(1);
    n1.left = n2;
    n1.right = n3;
    n2.left = n4;
    n2.right = n5;
    n5.left = n7;
    System.out.println(isBalanced(n1));
    System.out.println(isBalanced2(n1));
    System.out.println("----------------");
}
//             1
//            /
//           2
//          /
```

```
//          3
//         /
//        4
//       /
//      5
private static void test4() {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(1);
    BinaryTreeNode n3 = new BinaryTreeNode(1);
    BinaryTreeNode n4 = new BinaryTreeNode(1);
    BinaryTreeNode n5 = new BinaryTreeNode(1);
    BinaryTreeNode n6 = new BinaryTreeNode(1);
    BinaryTreeNode n7 = new BinaryTreeNode(1);
    n1.left = n2;
    n2.left = n3;
    n3.left = n4;
    n4.left = n5;
    System.out.println(isBalanced(n1));
    System.out.println(isBalanced2(n1));
    System.out.println("----------------");
}
// 1
//  \
//   2
//    \
//     3
//      \
//       4
//        \
//         5
private static void test5() {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(1);
    BinaryTreeNode n3 = new BinaryTreeNode(1);
    BinaryTreeNode n4 = new BinaryTreeNode(1);
    BinaryTreeNode n5 = new BinaryTreeNode(1);
    BinaryTreeNode n6 = new BinaryTreeNode(1);
    BinaryTreeNode n7 = new BinaryTreeNode(1);
    n1.right = n2;
    n2.right = n3;
    n3.right = n4;
    n4.right = n5;
    System.out.println(isBalanced(n1));
    System.out.println(isBalanced2(n1));
    System.out.println("----------------");
```

```
    }
}
```

#

运行结果


Run
Test39
"C:\Program ...
true
true
----
true
true
----
false
false
----
false
false

39

数组中只出现一次的数字

# #

题目：一个整型数组里除了两个数字之外，其他的数字都出现了两次，请写程序找出这两个只出现一次的数字。要求时间复杂度是 O(n)，空间复杂度是 O(1)。

#

举例说明

例如输入数组 { 2, 4, 3, 6, 3, 2, 5 }，因为只有 4 、6 这两个数字只出现一次，其他数字都出现了两次，所以输出 4 和 6 。

#

解题思路

这两个题目都在强调一个（或两个）数字只出现一次，其他的出现两次。这有什么意义呢？我们想到异或运算的一个性质：任何一个数字异或它自己都等于 0。也就是说， 如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现一次的数字，因为那些成对出现两次的数字全部在异或中抵消了。

想明白怎么解决这个简单问题之后，我们再回到原始的问题，看看能不能运用相同的思路。我们试着把原数组分成两个子数组，使得每个子数组包含一个只出现一次的数字，而其他数字都成对出现两次。如果能够这样拆分成两个数组， 我们就可以按照前面的办法分别找出两个只出现一次的数字了。

我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其他数字都出现了两次，在异或中全部抵消了。由于这两个数字肯定不一样，那么异或的结果肯定不为 0，也就是说在这个结果数字的二进制表示中至少就有一位为 1 。我们在结果数字中找到第一个为 1 的位的位置，记为第 n 位。现在我们以第 n 位是不是 1 为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第 n 位都是 1，而第二个子数组中每个数字的第 n 位都是 0。由于我们分组的标准是数字中的某一位是 1 还是 0 ， 那么出现了两次的数字肯定被分配到同一个子数组。因为两个相同的数字的任意一位都是相同的，我们不可能把两个相同的数字分配到两个子数组中去，于是我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其他数字都出现了两次。我们已经知道如何在数组中找出唯一一个只出现一次数字，因此到此为止所有的问题都已经解决了。

#

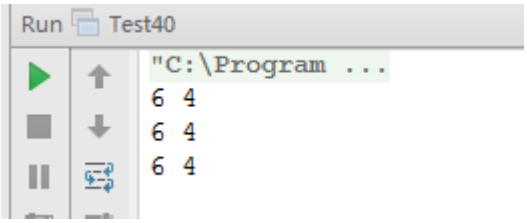
代码实现

```
public class Test40 {
    public static int[] findNumbersAppearanceOnce(int[] data) {
        int[] result = {0, 0};
        if (data == null || data.length < 2) {
            return result;
        }
        int xor = 0;
        for (int i : data) {
            xor ^= i;
        }
        int indexOf1 = findFirstBit1(xor);
        for (int i : data) {
            if (isBit1(i, indexOf1)) {
                result[0] ^= i;
            } else {
                result[1] ^= i;
            }
        }
        return result;
    }
    private static int findFirstBit1(int num) {
        int index = 0;
        while ((num & 1) == 0 && index < 32) {
            num >>>= 1;
            index++;
        }
        return index;
    }
    private static boolean isBit1(int num, int indexBit) {
        num >>>= indexBit;
        return (num & 1) == 1;
    }
    public static void main(String[] args) {
        int[] data1 = {2, 4, 3, 6, 3, 2, 5, 5};
        int[] result1 = findNumbersAppearanceOnce(data1);
        System.out.println(result1[0] + " " + result1[1]);
        int[] data2 = {4, 6};
        int[] result2 = findNumbersAppearanceOnce(data2);
        System.out.println(result2[0] + " " + result2[1]);
```

```
        int[] data3 = {4, 6, 1, 1, 1, 1};
        int[] result3 = findNumbersAppearanceOnce(data3);
        System.out.println(result3[0] + " " + result3[1]);
    }
}
```

#

运行结果



40

和为 s 的两个数字 vs 和为 s 的连续正数序列

# #

题目一：输入一个递增排序的数组和一个数字 s，在数组中查找两个数，得它们的和正好是 s。如果有多对数字的和等于 s，输出任意一对即可。

#

举例说明

例如输入数组 { 1 、2 、4、7 、11 、15 } 和数字 15。由于 4+11 = 15 ，因此输出 4 和 11 。

#

解题思路

我们先在数组中选择两个数字，如果它们的和等于输入的 s，我们就找到了要找的两个数字。如果和小于 s 呢？我们希望两个数字的和再大一点。由于数组已经排好序了，我们可以考虑选择较小的数字后面的数字。因为排在后面的数字要大一些，那么两个数字的和也要大一些， 就有可能等于输入的数字 s 了。同样， 当两个数字的和大于输入的数字的时候，我们可以选择较大数字前面的数字，因为排在数组前面的数字要小一些。

#

代码实现

```
/**
 * 输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得得它们的和正好是s。
 * 如果有多对数字的和等于s，输出任意一对即可。
 *
 * @param data
 * @param sum
 * @return
 */
public static List<Integer> findNumbersWithSum(int[] data, int sum) {
    List<Integer> result = new ArrayList<>(2);
    if (data == null || data.length < 2) {
        return result;
    }
```

```
    int ahead = data.length - 1;
    int behind = 0;
    long curSum; // 统计和，取long是防止结果溢出
    while (behind < ahead) {
        curSum = data[behind] + data[ahead];
        if (curSum == sum) {
            result.add(data[behind]);
            result.add(data[ahead]);
            break;
        } else if (curSum < sum) {
            behind++;
        } else {
            ahead--;
        }
    }
    return result;
}
```

#

题目二：输入一个正数 s，打印出所有和为 s 的连续正数序列（至少两个数）。

#

举例说明

例如输入 15，由于 1+2+3+4+5=4＋5+6＝7+8=15，所以结果打出 3 个连续序列 1～5、4～6 和 7～8。

#

解题思路

考虑用两个数 small 和 big 分别表示序列的最小值和最大值。首先把 small 初始化为 1，big 初始化为 2。如果从 small 到 big 的序列的和大于 s，我们可以从序列中去掉较小的值，也就是增大 small 的值。如果从 small 到 big 的序列的和小于 s，我们可以增大 big，让这个序列包含更多的数字。因为这个序列至少要有两个数字，我们一直增加 small 到(1+s)/2 为止。

以求和为 9 的所有连续序列为例，我们先把 small 初始化为 1，big 初始化为 2。此时介于 small 和 big 之间的序列是 { 1，2}，序列的和为 3，小于 9，所以我们下一步要让序列包含更多的数字。我们把 big 增加 1 变成 3，此时序列为 { I, 2，坷。由于序列的和是 6，仍然小于 9，我们接下来再增加 big 变成 4，介于 small 和 big 之间的序列也随之变成 { I, 2, 3, 4 } 。由于列的和 10 大于 9，我们要删去去序列中的一些数字， 于是我们增加 small 变成 2，此时得到的序列是 { 2, 3, 4 } ，序列的和E好是 9。我们找到了第一个和为 9 的连续序列，把它打印出来。接下来我们再增加 big，重复前面的过程，可以找到第二个和为 9 的连续序列 { 4，5}。

#

代码实现

```
public static List<List<Integer>> findContinuousSequence(int sum) {
    List<List<Integer>> result = new ArrayList<>();
    if (sum < 3) {
        return result;
    }
    int small = 1;
```

```
    int big = 2;
    int middle = (1 + sum) / 2;
    int curSum = small + big;
    while (small < middle) {
        if (curSum == sum) {
            List<Integer> list = new ArrayList<>(2);
            for (int i = small; i <= big; i++) {
                list.add(i);
            }
            result.add(list);
        }
        while (curSum > sum && small < middle) {
            curSum -= small;
            small++;
            if (curSum == sum) {
                List<Integer> list = new ArrayList<>(2);
                for (int i = small; i <= big; i++) {
                    list.add(i);
                }
                result.add(list);
            }
        }
        big++;
        curSum += big;
    }
    return result;
}
```

#

完整代码

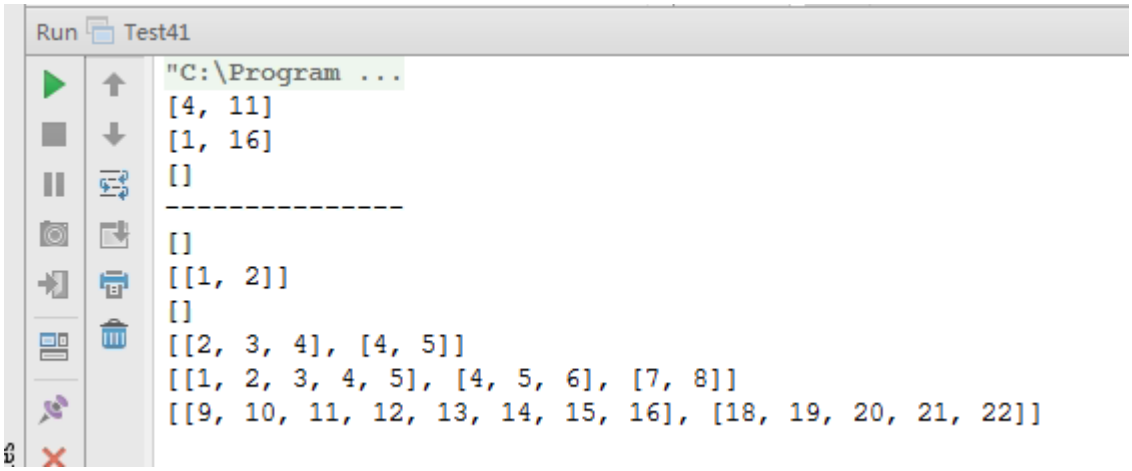
```
public class Test41 {
    /**
     * 输入一个递增排序的数组和一个数字s，在数组中查找两个数，使得得它们的和正好是s。
     * 如果有多对数字的和等于s，输出任意一对即可。
     *
     * @param data
     * @param sum
     * @return
     */
    public static List<Integer> findNumbersWithSum(int[] data, int sum) {
        List<Integer> result = new ArrayList<>(2);
        if (data == null || data.length < 2) {
```

```
        return result;
    }
    int ahead = data.length - 1;
    int behind = 0;
    long curSum; // 统计和，取long是防止结果溢出
    while (behind < ahead) {
        curSum = data[behind] + data[ahead];
        if (curSum == sum) {
            result.add(data[behind]);
            result.add(data[ahead]);
            break;
        } else if (curSum < sum) {
            behind++;
        } else {
            ahead--;
        }
    }
    return result;
}
/**
 * 题目二：输入一个正数s，打印出所有和为s 的连续正数序列（至少两个数）。
 * @param sum
 * @return
 */
public static List<List<Integer>> findContinuousSequence(int sum) {
    List<List<Integer>> result = new ArrayList<>();
    if (sum < 3) {
        return result;
    }
    int small = 1;
    int big = 2;
    int middle = (1 + sum) / 2;
    int curSum = small + big;
    while (small < middle) {
        if (curSum == sum) {
            List<Integer> list = new ArrayList<>(2);
            for (int i = small; i <= big; i++) {
                list.add(i);
            }
            result.add(list);
        }
        while (curSum > sum && small < middle) {
            curSum -= small;
            small++;
            if (curSum == sum) {
```

```
                List<Integer> list = new ArrayList<>(2);
                for (int i = small; i <= big; i++) {
                    list.add(i);
                }
                result.add(list);
            }
        }
        big++;
        curSum += big;
    }
    return result;
}
public static void main(String[] args) {
    test01();
    System.out.println("----------------");
    test02();
}
private static void test01() {
    int[] data1 = {1, 2, 4, 7, 11, 15};
    System.out.println(findNumbersWithSum(data1, 15));
    int[] data2 = {1, 2, 4, 7, 11, 16};
    System.out.println(findNumbersWithSum(data2, 17));
    int[] data3 = {1, 2, 4, 7, 11, 16};
    System.out.println(findNumbersWithSum(data3, 10));
}
public static void test02(){
    System.out.println(findContinuousSequence(1));
    System.out.println(findContinuousSequence(3));
    System.out.println(findContinuousSequence(4));
    System.out.println(findContinuousSequence(9));
    System.out.println(findContinuousSequence(15));
    System.out.println(findContinuousSequence(100));
}
}
```

#

运行结果


Run
Test41
"C:\Program ...
[4, 11]
[1, 16]
[]

[]
[[1, 2]]
[]
[[2, 3, 4], [4, 5]]
[[1, 2, 3, 4, 5], [4, 5, 6], [7, 8]]
[[9, 10, 11, 12, 13, 14, 15, 16], [18, 19, 20, 21, 22]]

41

翻转单词顺序 vs 左旋转字符串

# #

题目一：输入一个英文句子，翻转句子中单词的顺序，但单词内字啊的顺序不变。为简单起见，标点符号和普通字母一样处理。

#

举例说明

例如输入字符串”I am a student. ”，则输出”student. a am I”。

#

解题思路

第一步翻转句子中所有的字符。比如翻转“I am a student. ”中所有的字符得到”.tneduts a m a I”，此时不但翻转了句子中单词的顺序，连单词内的字符顺序也被翻转了。第二步再翻转每个单词中字符的顺序，就得到了”student. a am I”。这正是符合题目要求的输出。

#

代码实现

```
/**
 * 将data中start到end之间的数字反转
 *
 * @param data
 * @param start
 * @param end
 */
public static void reverse(char[] data, int start, int end) {
    if (data == null || data.length < 1 || start < 0 || end > data.length || start > end) {
        return;
    }
    while (start < end) {
        char tmp = data[start];
        data[start] = data[end];
```

```
            data[end] = tmp;
            start++;
            end--;
        }
    }
    /**
     * 题目一：输入一个英文句子，翻转句子中单词的顺序，但单词内字啊的顺序不变。
     * 为简单起见，标点符号和普通字母一样处理。
     *
     * @param data
     * @return
     */
    public static char[] reverseSentence(char[] data) {
        if (data == null || data.length < 1) {
            return data;
        }
        reverse(data, 0, data.length - 1);
        int start = 0;
        int end = 0;
        while (start < data.length) {
            if (data[start] == ' ') {
                start++;
                end++;
            } else if (end == data.length || data[end] == ' ') {
                reverse(data, start, end - 1);
                end++;
                start = end;
            } else {
                end++;
            }
        }
        return data;
    }
```

#

题目二：字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。

#

举例说明

比如输入字符串”abcefg”和数字 2，该函数将返回左旋转 2 位得到的结”cdefab”。

#

解题思路

以”abcdefg”为例，我们可以把它分为两部分。由于想把它的前两个字符移到后面，我们就把前两个字符分到第一部分，把后面的所有字符都分到第二部分。我们先分别翻转这两部分，于是就得到”bagfedc”。接下来我们再翻转整个字符串， 得到的”cde 也 ab”同 lj 好就是把原始字符串左旋转 2 位的结果。

#

代码实现

```
/**
 * 题目二：字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。
 * 请定义一个函数实现字符串左旋转操作的功能。
 * @param data
 * @param n
 * @return
 */
public static char[] leftRotateString(char[] data, int n) {
    if (data == null || n < 0 || n > data.length) {
        return data;
    }
    reverse(data, 0, data.length - 1);
    reverse(data, 0, data.length - n - 1);
    reverse(data, data.length - n, data.length - 1);
```

```
    return data;
}
```

#

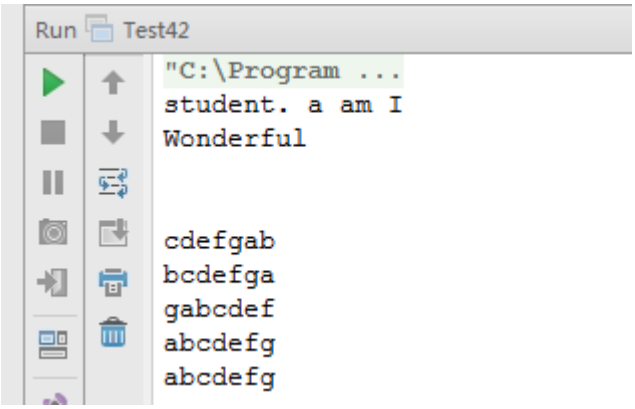
完整代码

```
public class Test42 {
    /**
     * 将data中start到end之间的数字反转
     *
     * @param data
     * @param start
     * @param end
     */
    public static void reverse(char[] data, int start, int end) {
        if (data == null || data.length < 1 || start < 0 || end > data.length || start > end) {
            return;
        }
        while (start < end) {
            char tmp = data[start];
            data[start] = data[end];
            data[end] = tmp;
            start++;
            end--;
        }
    }
    /**
     * 题目一：输入一个英文句子，翻转句子中单词的顺序，但单词内字啊的顺序不变。
     * 为简单起见，标点符号和普通字母一样处理。
     *
     * @param data
     * @return
     */
    public static char[] reverseSentence(char[] data) {
        if (data == null || data.length < 1) {
            return data;
        }
        reverse(data, 0, data.length - 1);
        int start = 0;
        int end = 0;
        while (start < data.length) {
            if (data[start] == ' ') {
                start++;
```

```
            end++;
        } else if (end == data.length || data[end] == ' ') {
            reverse(data, start, end - 1);
            end++;
            start = end;
        } else {
            end++;
        }
    }
    return data;
}
/**
 * 题目二：字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。
 * 请定义一个函数实现字符串左旋转操作的功能。
 * @param data
 * @param n
 * @return
 */
public static char[] leftRotateString(char[] data, int n) {
    if (data == null || n < 0 || n > data.length) {
        return data;
    }
    reverse(data, 0, data.length - 1);
    reverse(data, 0, data.length - n - 1);
    reverse(data, data.length - n, data.length - 1);
    return data;
}
public static void main(String[] args) {
    test01();
    test02();
}
private static void test01() {
    System.out.println(new String(reverseSentence("I am a student.".toCharArray())));
    System.out.println(new String(reverseSentence("Wonderful".toCharArray())));
    System.out.println(new String(reverseSentence("".toCharArray())));
    System.out.println(new String(reverseSentence("    ".toCharArray())));
}
private static void test02() {
    System.out.println(new String(leftRotateString("abcdefg".toCharArray(), 2)));
    System.out.println(new String(leftRotateString("abcdefg".toCharArray(), 1)));
    System.out.println(new String(leftRotateString("abcdefg".toCharArray(), 6)));
    System.out.println(new String(leftRotateString("abcdefg".toCharArray(), 7)));
    System.out.println(new String(leftRotateString("abcdefg".toCharArray(), 0)));
}
}
```

#

运行结果：


Run
Test42
"C:\Program ...
student. a am I
Wonderful
cdefgab
bcdefga
gabcdef
abcdefg
abcdefg

42

n 个锻子的点数

#

题目：把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 s。输入 n，打印出 s 的所有可能的值出现的概率。

#

解题思路

#

解法一：基于通归求解，时间效率不够高。

先把 n 个骰子分为两堆：第一堆只有一个，另一个有 n-1 个。单独的那一个有可能出现从 1 到 6 的点数。我们需要计算从 1 到 6 的每一种点数和剩下的 n-1 个骰子来计算点数和。接下来把剩下的 n-1 个骰子还是分成两堆，第一堆只有一个， 第二堆有 n-2 个。我们把上一轮那个单独骰子的点数和这一轮单独骰子的点数相加， 再和剩下的 n-2 个骰子来计算点数和。分析到这里，我们不难发现这是一种递归的思路，递归结束的条件就是最后只剩下一个骰子。

我们可以定义一个长度为“6n-n+1”的数组， 和为 s 的点数出现的次数保存到数组第 s-n 个元素里。

#

解法二：基于循环求解，时间性能好

我们可以考虑用两个数组来存储骰子点数的每一个总数出现的次数。在一次循环中， 第一个数组中的第 n 个数字表示骰子和为 n 出现的次数。在下一循环中，我们加上一个新的骰子，此时和为 n 的骰子出现的次数应该等于上一次循环中骰子点数和为 n-1 、n-2 、n-3 、n-4, n-5 与 n-6 的次数的总和，所以我们把另一个数组的第 n 个数字设为前一个数组对应的第 n-1 、n-2 、n-3 、n-4、n-5 与 n-6 之和。

#

代码实现

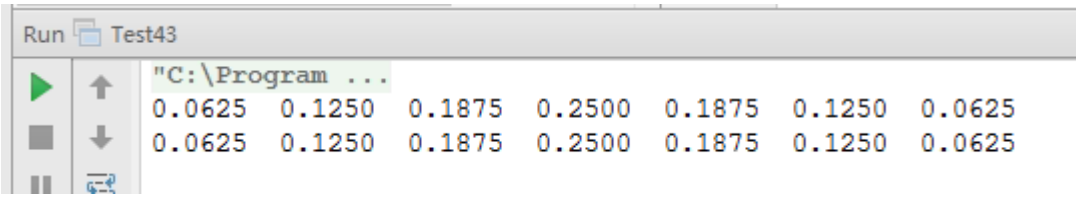
```
public class Test43 {
    /**
     * 基于通归求解
     *
     * @param number 色子个数
     * @param max    色子的最大值
     */
    public static void printProbability(int number, int max) {
        if (number < 1 || max < 1) {
            return;
        }
        int maxSum = number * max;
        int[] probabilities = new int[maxSum - number + 1];
        probability(number, probabilities, max);
        double total = 1;
        for (int i = 0; i < number; i++) {
            total *= max;
        }
        for (int i = number; i <= maxSum; i++) {
            double ratio = probabilities[i - number] / total;
            System.out.printf("%-8.4f", ratio);
        }
        System.out.println();
    }
    /**
     * @param number        色子个数
     * @param probabilities 不同色子数出现次数的计数数组
     * @param max           色子的最大值
     */
    private static void probability(int number, int[] probabilities, int max) {
        for (int i = 1; i <= max; i++) {
            probability(number, number, i, probabilities, max);
        }
    }
    /**
     * @param original      总的色子数
     * @param current       当前处理的是第几个
     * @param sum           已经前面的色子数和
     * @param probabilities 不同色子数出现次数的计数数组
     * @param max           色子的最大值
     */
    private static void probability(int original, int current, int sum, int[] probabilities, int max) {
        if (current == 1) {
            probabilities[sum - original]++;
        } else {
```

```
        for (int i = 1; i <= max; i++) {
            probability(original, current - 1, i + sum, probabilities, max);
        }
    }
}
/**
 * 基于循环求解
 * @param number 色子个数
 * @param max    色子的最大值
 */
public static void printProbability2(int number, int max) {
    if (number < 1 || max < 1) {
        return;
    }
    int[][] probabilities = new int[2][max * number + 1];
    // 数据初始化
    for (int i = 0; i < max * number + 1; i++) {
        probabilities[0][i] = 0;
        probabilities[1][i] = 0;
    }
    // 标记当前要使用的是第0个数组还是第1个数组
    int flag = 0;
    // 抛出一个骰子时出现的各种情况
    for (int i = 1; i <= max; i++) {
        probabilities[flag][i] = 1;
    }
    // 抛出其它骰子
    for (int k = 2; k <= number; k++) {
        // 如果抛出了k个骰子，那么和为[0, k-1]的出现次数为0
        for (int i = 0; i < k; i++) {
            probabilities[1 - flag][i] = 0;
        }
        // 抛出k个骰子，所有和的可能
        for (int i = k; i <= max * k; i++) {
            probabilities[1 - flag][i] = 0;
            // 每个骰子的出现的所有可能的点数
            for (int j = 1; j <= i && j <= max; j++) {
                // 统计出和为i的点数出现的次数
                probabilities[1 - flag][i] += probabilities[flag][i - j];
            }
        }
        flag = 1 - flag;
    }
    double total = 1;
    for (int i = 0; i < number; i++) {
```

```
            total *= max;
        }
        int maxSum = number * max;
        for (int i = number; i <= maxSum; i++) {
            double ratio = probabilities[flag][i] / total;
            System.out.printf("%-8.4f", ratio);
        }
        System.out.println();
    }
    public static void main(String[] args) {
        test01();
        test02();
    }
    private static void test01() {
        printProbability(2, 4);
    }
    private static void test02() {
        printProbability2(2, 4);
    }
}
```

#

运行结果



43

扑克牌的顺子

# #

题目：从扑克牌中随机抽 5 张牌，判断是不是一个顺子， 即这 5 张牌是不是连续的。2～10 为数字本身， A 为 1。 J 为 11、Q 为 12、 为 13。小王可以看成任意数字。

#

解题思路

我们可以把 5 张牌看成由 5 个数字组成的数组。大、小王是特殊的数字，我们不妨把它们都定义为 0，这样就能和其他扑克牌区分开来了。

接下来我们分析怎样判断 5 个数字是不是连续的，最直观的方法是把数组排序。值得注意的是，由于 0 可以当成任意数字，我们可以用 0 去补满数组中的空缺。如果排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但只要我们有足够的。可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0，1，3，4，5}在 1 和 3 之间空缺了一个 2，刚好我们有一个 0，也就是我们可以把它当成 2 去填补这个空缺。

于是我们需要做 3 件事情： 首先把数组排序，再统计数组中 0 的个数，最后统计排序之后的数组中相邻数字之间的空缺总数。如果空缺的总数小于或者等于 0 的个数，那么这个数组就是连续的：反之则不连续。

最后，我们还需要注意一点： 如果数组中的非 0 数字重复出现，则该数组不是连续的。换成扑克牌的描述方式就是如果一副牌里含有对子，则不可能是顺子。

#

算法实现

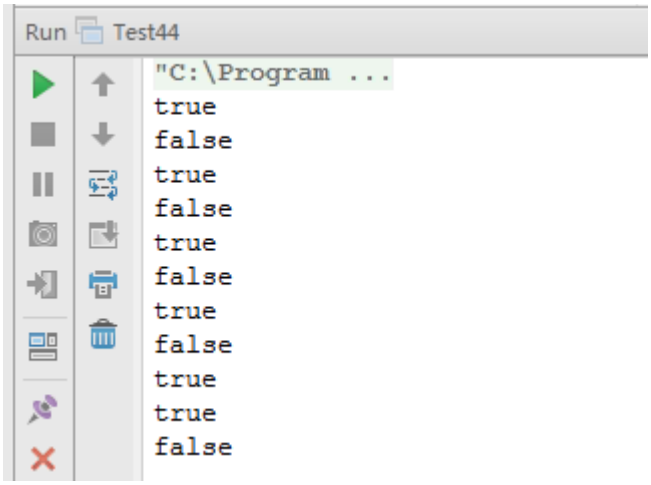
```
import java.util.Arrays;
public class Test44 {
    /**
     * 题目：从扑克牌中随机抽5张牌，判断是不是一个顺子， 即这5张牌是不是连续的。
     * 2～10为数字本身， A为1。 J为11、Q为12、 为13。小王可以看成任意数字。
     * @param numbers
     * @return
     */
    public static boolean isContinuous(int[] numbers) {
        if (numbers == null || numbers.length != 5) {
```

```
            return false;
        }
        // 对元素进行排序
        Arrays.sort(numbers);
        int numberOfZero = 0;
        int numberOfGap = 0;
        for (int i = 0; i < numbers.length && numbers[i] == 0; i++) {
            numberOfZero++;
        }
        // 第一个非0元素的位置
        int small = numberOfZero;
        int big = small + 1;
        while (big < numbers.length) {
            if (numbers[small] == numbers[big]) {
                return false;
            }
            numberOfGap += (numbers[big] - numbers[small] - 1);
            small = big;
            big++;
        }
        return numberOfGap <= numberOfZero;
    }
    public static void main(String[] args) {
        int[] numbers1 = {1, 3, 2, 5, 4};
        System.out.println(isContinuous(numbers1));
        int[] numbers2 = {1, 3, 2, 6, 4};
        System.out.println(isContinuous(numbers2));
        int[] numbers3 = {0, 3, 2, 6, 4};
        System.out.println(isContinuous(numbers3));
        int[] numbers4 = {0, 3, 1, 6, 4};
        System.out.println(isContinuous(numbers4));
        int[] numbers5 = {1, 3, 0, 5, 0};
        System.out.println(isContinuous(numbers5));
        int[] numbers6 = {1, 3, 0, 7, 0};
        System.out.println(isContinuous(numbers6));
        int[] numbers7 = {1, 0, 0, 5, 0};
        System.out.println(isContinuous(numbers7));
        int[] numbers8 = {1, 0, 0, 7, 0};
        System.out.println(isContinuous(numbers8));
        int[] numbers9 = {3, 0, 0, 0, 0};
        System.out.println(isContinuous(numbers9));
        int[] numbers10 = {0, 0, 0, 0, 0};
        System.out.println(isContinuous(numbers10));
        int[] numbers11 = {1, 0, 0, 1, 0};
        System.out.println(isContinuous(numbers11));
```

```
    }
}
```

#

运行结果


Run
Test44
"C:\Program ...
true
false
true
false
true
false
true
false
true
true
false

44

圆圈中最后剩下的数字(约瑟夫环问题)

# #

题目：0, 1, ⋯ , n−1 这 n 个数字排成一个圈圈，从数字 0 开始每次从圆圈里删除第 m 个数字。求出这个圈圈里剩下的最后一个数字。

#

解题思路

#

第一种：经典的解法， 用环形链表模拟圆圈。

创建一个总共有 n 个结点的环形链表，然后每次在这个链表中删除第 m 个结点。

#

代码实现

```
public static int lastRemaining(int n, int m) {
    if (n < 1 || m < 1) {
        return -1;
    }
    List<Integer> list = new LinkedList<>();
    for (int i = 0; i < n; i++) {
        list.add(i);
    }
    // 要删除元素的位置
    int idx = 0;
    // 开始计数的位置
    int start = 0;
    while (list.size() > 1) {
        // 只要移动m-1次就可以移动到下一个要删除的元素上
        for (int i = 1; i < m; i++) {
            idx = (idx + 1) % list.size(); // 【A】
        }
        list.remove(idx);
        // 确保idx指向每一轮的第一个位置
```

```
        // 下面的可以不用，【A】已经可以保证其正确性了，可以分析n=6，m=6的第一次删除情况
    //  if (idx == list.size()) {
    //      idx = 0;
    // }
    }
    return list.get(0);
}
```

#

第二种：分析法

首先我们定义一个关于 n 和 m 的方程町矶时，表示每次在 n 个数字 0，1， … ，n−1中每次删除第 m 个数字最后剩下的数字。

在这 n个数字中， 第一个被删除的数字是(m−1)%n。为了简单起见，我们把(m− 1)%n 记为 k，那么删除k之后剩下的 n−1 个数字为 0，1，… ，k−1，k+1，… ，n−1，并且下一次删除从数字 k+1 开始计数。相当于在剩下的序列中， k+1 排在最前面，从而形成 k+1，... ，n− 1，0，I，… ，k−1 。该序列最后剩下的数字也应该是关于 n 和 m 的函数。由于这个序列的规律和前面最初的序列不一样（最初的序列是从 0 开始的连续序列），因此该函数不同于前面的函数，记为 f’（n−1,m)。最初序列最后剩下的数字 f(n, m）一定是删除一个数字之后的序列最后剩下的数字，即 f(n, m)=f’（n−1, m）。

接下来我们把剩下的这 n−1 个数字的序列 k−1， …，n−1，0，1，… ，k−1 做一个映射，映射的结果是形成一个从 0 到 n−2 的序列：

k+1 → 0

k+2 → 1

...

n-1 → n-k-2

0 → n-k-1

1 → n-k

...

k-1 → n-2

我们把映射定义为 p，则 p(x)=(x-k-1)%n。它表示如果映射前的数字是 x，那么映射后的数字是(x-k-1)%n。该映射的逆映射是 $p^{-1}(x)=(x+k+1)\%n$。

由于映射之后的序列和最初的序列具有同样的形式，即都是从 0 开始的连续序列，因此仍然可以用函数 f 来表示，记为 f(n-1, m)。根据我们的映射规则，映射之前的序列中最后剩下的数字 $f'(n-1, m)=p^{-1}[f(n-1, m)]=[f(n-1, m)+k+1]\%n$，把 $k=(m-1)\%n$ 代入得到 $f(n, m)=f'(n-1,m)=[f(n-1, m)+m]\%n$。

经过上面复杂的分析，我们终于找到了一个递归公式。要得到 n 个数字的序列中最后剩下的数字，只需要得到 n-1 个数字的序列中最后剩下的数字，并以此类推。当 n=1 时，也就是序列中开始只有一个数字 0，那么很显然最后剩下的数字就是 0。我们把这种关系表示为：

$$f(n,m)=\begin{cases}0 & n=1\\ [f(n-1,m)+m]\%n & n>1\end{cases}$$

#

代码实现

```
public static int lastRemaining2(int n, int m) {
    if (n < 1 || m < 1) {
        return -1;
    }
    int last = 0;
    for (int i = 2; i <=n ; i++) {
        last = (last + m)%i;
    }
```

```
    return last;
}
```

#

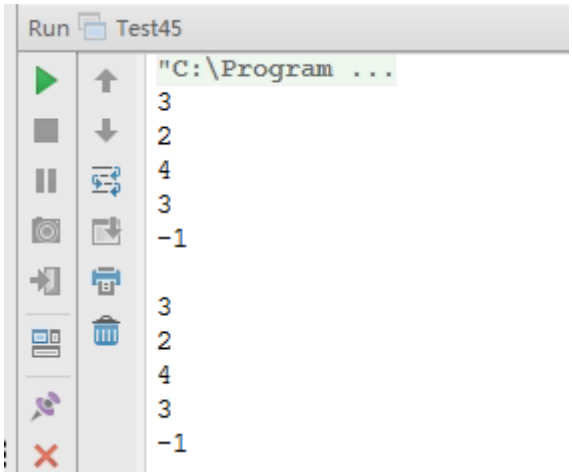
完整代码

```
import java.util.LinkedList;
import java.util.List;
public class Test45 {
    public static int lastRemaining(int n, int m) {
        if (n < 1 || m < 1) {
            return -1;
        }
        List<Integer> list = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            list.add(i);
        }
        // 要删除元素的位置
        int idx = 0;
        // 开始计数的位置
        int start = 0;
        while (list.size() > 1) {
            // 只要移动m-1次就可以移动到下一个要删除的元素上
            for (int i = 1; i < m; i++) {
                idx = (idx + 1) % list.size(); // 【A】
            }
            list.remove(idx);
            // 确保idx指向每一轮的第一个位置
            // 下面的可以不用，【A】已经可以保证其正确性了，可以分析n=6，m=6的第一次删除情况
        //  if (idx == list.size()) {
        //      idx = 0;
        //  }
        }
        return list.get(0);
    }
    public static int lastRemaining2(int n, int m) {
        if (n < 1 || m < 1) {
            return -1;
        }
        int last = 0;
        for (int i = 2; i <=n ; i++) {
            last = (last + m)%i;
        }
```

```
        return last;
    }
    public static void main(String[] args) {
        test01();
        System.out.println();
        test02();
    }
    private static void test01() {
        System.out.println(lastRemaining(5, 3)); // 最后余下3
        System.out.println(lastRemaining(5, 2)); // 最后余下2
        System.out.println(lastRemaining(6, 7)); // 最后余下4
        System.out.println(lastRemaining(6, 6)); // 最后余下3
        System.out.println(lastRemaining(0, 0)); // 最后余下-1
    }
    private static void test02() {
        System.out.println(lastRemaining2(5, 3)); // 最后余下3
        System.out.println(lastRemaining2(5, 2)); // 最后余下2
        System.out.println(lastRemaining2(6, 7)); // 最后余下4
        System.out.println(lastRemaining2(6, 6)); // 最后余下3
        System.out.println(lastRemaining2(0, 0)); // 最后余下-1
    }
}
```

#

运行结果



45 不用加减乘除做加法

# #

题目：写一个函数，求两个整数之和，要求在函数体内不得使用 + 、 − 、 × 、 ÷ 四则运算符号。

#

解题思路

5 的二进制是 101，17 的二进制是 10001 。还是试着把计算分成三步：第一步各位相加但不计进位， 得到的结果是 10100（ 最后一位两个数都是1,相加的结果是二进制的 10 。这一步不计进位， 因此结果仍然是 0 。第二步记下进位。在这个例子中只在最后一位相加时产生一个进位，结果是二进制的 10。 第三步把前两步的结果相加，得到的结果是 10110 ， 转换成十进制正好是 22。由此可见三步走的策略对二进制也是适用的。

接下来我们试着把二进制的加法用位运算来替代。第一步不考虑进位对每一位相加。0 加 0 、1 加 1 的结果都 0。 0 加 1 、1 加 0 的结果都是 1 。我们注意到，这和异或的结果是一样的。对异或而言， 0 和 0、1 和 1 异或的结果是 0， 而 0 和 1 、1 和 0 的异或结果是 1 。接着考虑第二步进位，对加 0 、0 加 1 、1 加 0 而言， 都不会产生进位，只有 1 加 1 时，会向前产生一个进位。此时我们可以想象成是两个数先做位与运算，然后再向左移动一位。只有两个数都是 1 的时候，位与得到的结果是 1，其余都是 0。第三步把前两个步骤的结果相加。第三步相加的过程依然是重复前面两步， 直到不产生进位为止。

#

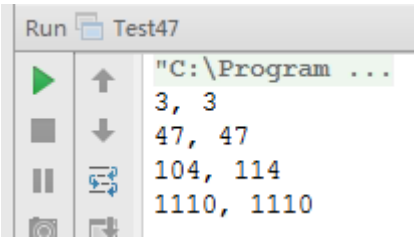
算法实现

```
public class Test47 {
    public static int add(int x, int y) {
        int sum;
        int carry;
        do {
            sum = x ^ y;
            // x&y的某一位是1说明，它是它的前一位的进位，所以向左移动一位
            carry = (x & y) << 1;
            x = sum;
            y = carry;
        } while (y != 0);
        return x;
    }
    public static void main(String[] args) {
```

```
        System.out.println(add(1, 2) + ", " + (1 + 2));
        System.out.println(add(13, 34)+ ", " + (13 + 34));
        System.out.println(add(19, 85)+ ", " + (19 + 95));
        System.out.println(add(865, 245)+ ", " + (865 + 245));
    }
}
```

#

运行结果



46

把字符串转换成整数

# #

题目：实现一个函数 stringToInt，实现把字符串转换成整数这个功能，不能使用 atoi 或者其他类似的库函数。

#

题目解析

这看起来是很简单的题目，实现基本功能 ，大部分人都能用10行之内的代码解决。可是，当我们要把很多特殊情况即测试用例都考虑进去，却不是件容易的事。解决数值转换问题本身并不难，但我希望在写转换数值的代码之前，应聘者至少能把空指针，空字符串”“，正负号，溢出等方方面面的测试用例都考虑到，并且在写代码的时候对这些特殊的输入都定义好合理的输出。当然，这些输出并不一定要和atoi完全保持一致，但必须要有显式的说明，和面试官沟通好。

这个应聘者最大的问题就是还没有养成在写代码之前考虑所有可能的测试用例的习惯，逻辑不够严谨，因此一开始的代码只处理了最基本的数值转换。后来我每次提醒他一处特殊的测试用例之后，他改一处代码。尽管他已经做了两次修改，但仍然有不少很明显的漏洞，特殊输入空字符串”“，边界条件比如最大的正整数与最小的负整数等。由于这道题思路本身不难，因此我希望他把问题考虑得极可能周到，代码尽量写完整。

#

代码实现

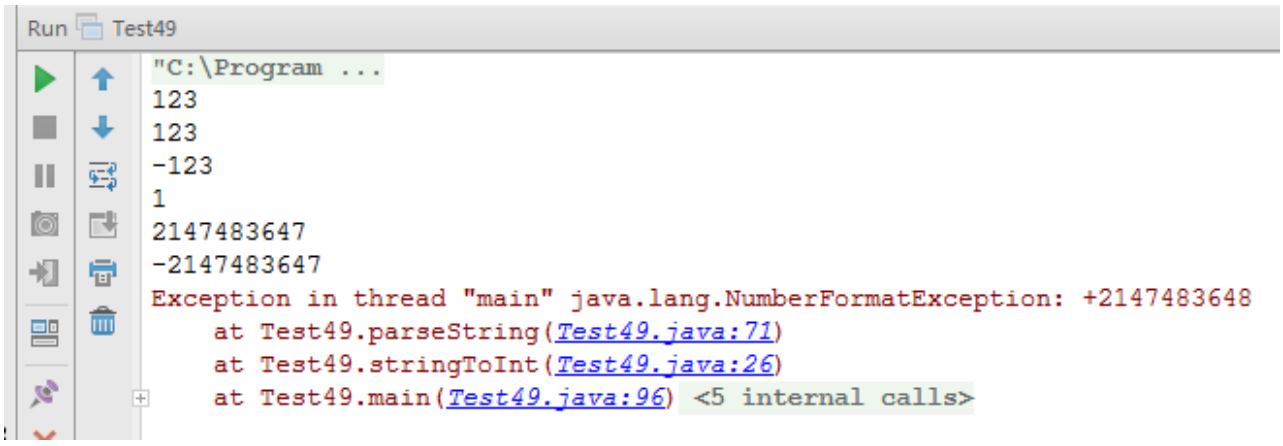
```
public class Test49 {
    /**
     * 题目：实现一个函数stringToInt,实现把字符串转换成整数这个功能，
     * 不能使用atoi或者其他类似的库函数。
     *
     * @param num
     * @return
     */
    public static int stringToInt(String num) {
        if (num == null || num.length() < 1) {
            throw new NumberFormatException(num);
        }
        char first = num.charAt(0);
        if (first == '-') {
            return parseString(num, 1, false);
```

```
    } else if (first == '+') {
        return parseString(num, 1, true);
    } else if (first <= '9' && first >= '0') {
        return parseString(num, 0, true);
    } else {
        throw new NumberFormatException(num);
    }
}
/**
 * 判断字符是否是数字
 *
 * @param c 字符
 * @return true是，false否
 */
private static boolean isDigit(char c) {
    return c >= '0' && c <= '9';
}
/**
 * 对字符串进行解析
 *
 * @param num      数字串
 * @param index    开始解析的索引
 * @param positive 是正数还是负数
 * @return 返回结果
 */
private static int parseString(String num, int index, boolean positive) {
    if (index >= num.length()) {
        throw new NumberFormatException(num);
    }
    int result;
    long tmp = 0;
    while (index < num.length() && isDigit(num.charAt(index))) {
        tmp = tmp * 10 + num.charAt(index) - '0';
        // 保证求的得的值不超出整数的最大绝对值
        if (tmp > 0x8000_0000L) {
            throw new NumberFormatException(num);
        }
        index++;
    }
    if (positive) {
        if (tmp >= 0x8000_0000L) {
            throw new NumberFormatException(num);
        } else {
            result = (int) tmp;
        }
```

```
        } else {
            if (tmp == 0x8000_0000L) {
                result = 0x8000_0000;
            } else {
                result = (int) -tmp;
            }
        }
        return result;
    }
    public static void main(String[] args) {
//        System.out.println(Integer.parseInt(Integer.MIN_VALUE + ""));
//        System.out.println(0x8000_0000L);
//        System.out.println(stringToInt(""));
        System.out.println(stringToInt("123"));
        System.out.println(stringToInt("+123"));
        System.out.println(stringToInt("-123"));
        System.out.println(stringToInt("1a123"));
        System.out.println(stringToInt("+2147483647"));
        System.out.println(stringToInt("-2147483647"));
        System.out.println(stringToInt("+2147483648"));
        System.out.println(stringToInt("-2147483648"));
//        System.out.println(stringToInt("+2147483649"));
//        System.out.println(stringToInt("-2147483649"));
//        System.out.println(stringToInt("+"));
//        System.out.println(stringToInt("-"));
    }
}
```

#

运行结果



47

树中两个结点的最低公共祖先

# #

题目：求树中两个结点的最低公共祖先，此树不是二叉树，并且没有指向父节点的指针。

#

树的结点定义

```
private static class TreeNode {
    int val;
    List<TreeNode> children = new LinkedList<>();
    public TreeNode() {
    }
    public TreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return  val + "";
    }
}
```

#

题目解析

假设还是输入结点 F 和 H。

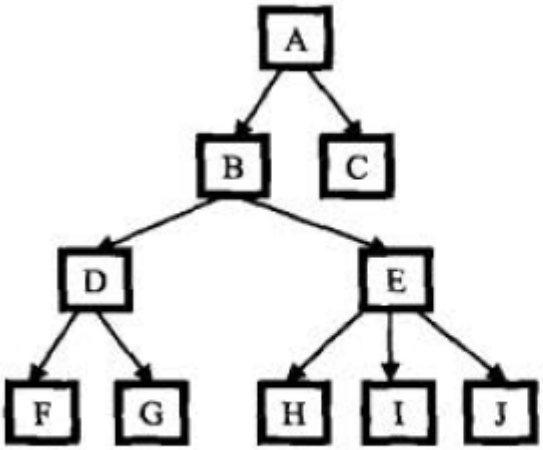


图 7.2 一棵普通的树，树中的结点没有指向父结点的指针

我们首先得到一条从根结点到树中某一结点的路径，这就要求在遍历的时候，有一个辅助内存来保存路径。比如我们用前序遍历的方法来得到从根结点到 H 的路径的过程是这样的：（ 1 ）遍历到 A，把 A 存放到路径中去，路径中只有一个结点 A;（ 2 ）遍历到 B，把 B 存到路径中去，此时路径为 A->B;（ 3 ）遍历到 D，把 D 存放到路径中去，此，时路径为 A->B->D;（ 4 ）：遍历到 F，把 F 存放到路径中去，此时路径为 A->B->D->F;（ 5) F 已经没有子结点了，因此这条路径不可能到这结点 H。把 F 从路径中删除，变成 A->B->D;（ 6 ）遍历 G。和结点F 一样，这条路径也不能到达 H。边历完 G 之后，路径仍然是 A->B->D;（ 7 ）由于 D 的所有子结点都遍历过了，不可能到这结点 H，因此 D 不在从 A 到 H 的路径中，把 D 从路径中删除，变成 A->B;（ 8 ）遥历 E，把 E 加入到路径中，此时路径变成 A->B->E,（ 9 ）遍历 H，已经到达目标给点， A->B->E 就是从根结点开始到达 H 必须经过的路径。

同样，我们也可以得到从根结点开始到达 F 必须经过的路径是 A->B。接着，我们求出这两个路径的最后公共结点，也就是 B。B 这个结点也是 F 和 H 的最低公共祖先.

为了得到从根结点开始到输入的两个结点的两条路径，需要追历两次树，每边历一次的时间复杂度是 O(n ）。得到的两条路径的长度在最差情况时是 0（通常情况丁两条路径的长度是 O(logn)。

注意：可以在只遍历树一次就找到两个结点的路径，这部分留给读者自己去完成。

#

代码实现

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
public class Test50 {
```

```
/**
 * 树的结点定义
 */
private static class TreeNode {
    int val;
    List<TreeNode> children = new LinkedList<>();
    public TreeNode() {
    }
    public TreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
/**
 * 找结点的路径
 *
 * @param root   根结点
 * @param target 目标结点
 * @param path   从根结点到目标结点的路径
 */
public static void getNodePath(TreeNode root, TreeNode target, List<TreeNode> path) {
    if (root == null) {
        return;
    }
    // 添加当前结点
    path.add(root);
    List<TreeNode> children = root.children;
    // 处理子结点
    for (TreeNode node : children) {
        if (node == target) {
            path.add(node);
            return;
        } else {
            getNodePath(node, target, path);
        }
    }
    // 现场还原
    path.remove(path.size() - 1);
}
/**
 * 找两个路径中的最后一个共同的结点
 *
```

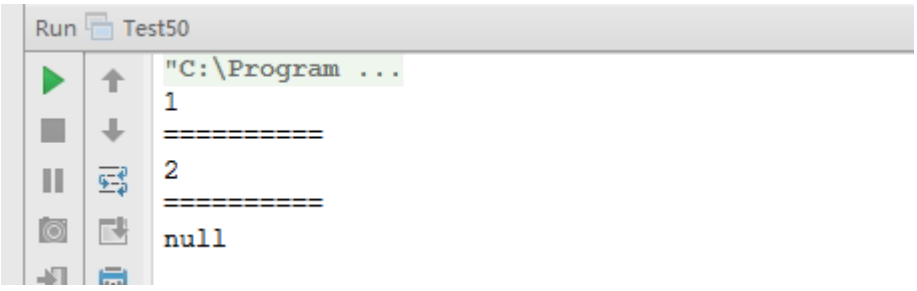
```
     * @param p1 路径1
     * @param p2 路径2
     * @return 共同的结点，没有返回null
     */
    public static TreeNode getLastCommonNode(List<TreeNode> p1, List<TreeNode> p2) {
        Iterator<TreeNode> ite1 = p1.iterator();
        Iterator<TreeNode> ite2 = p2.iterator();
        TreeNode last = null;
        while (ite1.hasNext() && ite2.hasNext()) {
            TreeNode tmp = ite1.next();
            if (tmp == ite2.next()) {
                last = tmp;
            }
        }
        return last;
    }
    /**
     * 找树中两个结点的最低公共祖先
     * @param root 树的根结点
     * @param p1 结点1
     * @param p2 结点2
     * @return 公共结点，没有返回null
     */
    public static TreeNode getLastCommonParent(TreeNode root, TreeNode p1, TreeNode p2) {
        if (root == null || p1 == null || p2 == null) {
            return null;
        }
        List<TreeNode> path1 = new LinkedList<>();
        getNodePath(root, p1, path1);
        List<TreeNode> path2 = new LinkedList<>();
        getNodePath(root, p2, path2);
        return getLastCommonNode(path1, path2);
    }
    public static void main(String[] args) {
        test01();
        System.out.println("==========");
        test02();
        System.out.println("==========");
        test03();
    }
    // 形状普通的树
    //            1
    //          /   \
    //        2      3
    //       /        \
```

```
//     4           5
//    / \       / | \
//   6   7     8   9  10
public static void test01() {
    TreeNode n1 = new TreeNode(1);
    TreeNode n2 = new TreeNode(2);
    TreeNode n3 = new TreeNode(3);
    TreeNode n4 = new TreeNode(4);
    TreeNode n5 = new TreeNode(5);
    TreeNode n6 = new TreeNode(6);
    TreeNode n7 = new TreeNode(7);
    TreeNode n8 = new TreeNode(8);
    TreeNode n9 = new TreeNode(9);
    TreeNode n10 = new TreeNode(10);
    n1.children.add(n2);
    n1.children.add(n3);
    n2.children.add(n4);
    n4.children.add(n6);
    n4.children.add(n7);
    n3.children.add(n5);
    n5.children.add(n8);
    n5.children.add(n9);
    n5.children.add(n10);
    System.out.println(getLastCommonParent(n1, n6, n8));
}
// 树退化成一个链表
//              1
//             /
//            2
//           /
//          3
//         /
//        4
//       /
//      5
private static void test02() {
    TreeNode n1 = new TreeNode(1);
    TreeNode n2 = new TreeNode(2);
    TreeNode n3 = new TreeNode(3);
    TreeNode n4 = new TreeNode(4);
    TreeNode n5 = new TreeNode(5);
    n1.children.add(n2);
    n2.children.add(n3);
    n3.children.add(n4);
    n4.children.add(n5);
```

```
        System.out.println(getLastCommonParent(n1, n4, n5));
    }
    // 树退化成一个链表，一个结点不在树中
    //            1
    //             /
    //            2
    //           /
    //          3
    //         /
    //        4
    //       /
    //      5
    private static void test03() {
        TreeNode n1 = new TreeNode(1);
        TreeNode n2 = new TreeNode(2);
        TreeNode n3 = new TreeNode(3);
        TreeNode n4 = new TreeNode(4);
        TreeNode n5 = new TreeNode(5);
        TreeNode n6 = new TreeNode(6);
        n1.children.add(n2);
        n2.children.add(n3);
        n3.children.add(n4);
        n4.children.add(n5);
        System.out.println(getLastCommonParent(n1, n5, n6));
    }
}
```

#

运行结果



48

数组中重复的数字

# #

题目：在一个长度为n的数组里的所有数字都在 0 到 n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。

#

举例说明

例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字 2 或者 3。

#

题目分析

解决这个问题的一个简单的方法是先把输入的数组排序。从排序的数组中找出重复的数字时间很容易的事情，只需要从头到尾扫描排序后的数组就可以了。排序一个长度为 n 的数组需要 O(nlogn)的时间。

还可以利用哈希表来解决这个问题。从头到尾按顺序扫描数组的每个数，每扫描一个数字的时候，都可以用 O(1)的时间来判断哈希表里是否已经包含了该数字。如果哈希表里还没有这个数字，就把它加入到哈希表里。如果哈希表里已经存在该数字了，那么就找到一个重复的数字。这个算法的时间复杂度是 O(n)，但它提高时间效率是以一个大小为 O(n)的哈希表为代价的。我们再看看有没有空间复杂度为 O(1)的算法。

我们注意到数组中的数字都在 0 到 n-1 中。如果这个数组中没有重复的数字，那么当数组排序之后数字 i 将出现在下标为 i 的位置。由于数组中有重复的数字，有些位置可能存在多个数字，同时有些位置可能没有数字。

现在让我们重排这个数组，依然从头到尾一次扫描这个数组中的每个数字。当扫描到下标为 i 的数字时，首先比较这个数字（用 m 表示）是不是等于 i。如果是，接着扫描下一个数字。如果不是，再拿它和第 m 个数字进行比较。 如果它和第m个数字相等，就找到了一个重复的数字（该数字在下标为 i 和 m 的位置都出现了）。如果它和第 m 个数字不相等，就把第 i 个数字和第 m 个数字交换，把 m 放到属于它的位置。接下来再重读这个比较、交换的过程，直到我们发现一个重复的数字。

以数组{2,3,1,0,2,5,3}为例来分析找到重复数字的步骤。数组的第 0 个数字（从 0 开始计数，和数组的下标保持一致）是 2，与它的下标不相等，于是把它和下标为 2 的数字 1 交换。交换之后的数组是{1.3.2.0.2.5.3}。此时第 0 个数字是 1，仍然与它的下标不相等，继续把它和下标为 1 的数字 3 交换，得到数组{3,1,2,0,2,5,3}.接下来继续交换第 0 个数字 3 和第 3 个数字 0，得到数组{0,1,2,3,2,5,3}。此时第 0 个数字的数值为 0，接着扫描下一

个数字。在接下来的几个数字中，下标为 1，2，3 的三个数字分别为 1，2，3 它们的下标和数值都分别相等，因此不需要做任何操作。接下来扫描到下标为 4 的数字 2。由于它的数值与它的下标不相等，再比较它和下标为 2 的数字。注意到此时数组中下标为 2 的数字也是 2，也就是数字在下标为 2 和下标为 4 的两个位置都出现了，因此找到一个重复的数字。

#

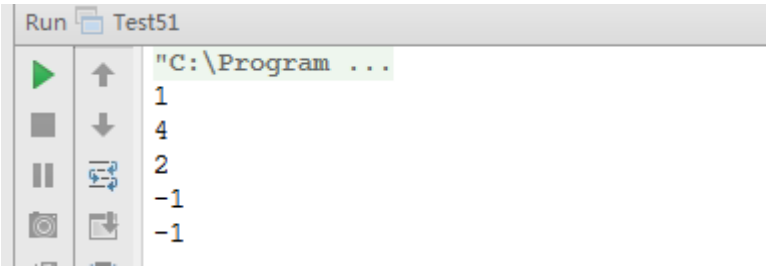
代码实现

```
public class Test51 {
    /**
     * 题目：在一个长度为n的数组里的所有数字都在0到n-1的范围内。数组中某些数字是重复的，
     * 但不知道有几个数字重复了，也不知道每个数字重复了几次。请找出数组中任意一个重复的数字。
     * 例如，如果输入长度为7的数组{2,3,1,0,2,5,3}，那么对应的输出是重复的数字2或者。
     *
     * @param number
     * @return
     */
    public static int duplicate(int[] number) {
        if (number == null || number.length < 1) {
            return -1;
        }
        // 判断输入的是否在[0, number.length-1]之间
        for (int i : number) {
            if (i < 0 || i >= number.length) {
                return -1;
            }
        }
        for (int i = 0; i < number.length; i++) {
            // 当number[i]与i不相同的时候一直交换
            while (number[i] != i) {
                // 如果i位置与number[i]位置的数字相同，说明有重复数字
                if (number[i] == number[number[i]]) {
                    return number[i];
                }
                // 如果不同就交换
                else {
                    swap(number, i, number[i]);
                }
            }
        }
        return -1;
    }
```

```
    private static void swap(int[] data, int x, int y) {
        int tmp = data[x];
        data[x] = data[y];
        data[y] = tmp;
    }
    public static void main(String[] args) {
        int[] numbers1 = {2, 1, 3, 1, 4};
        System.out.println(duplicate(numbers1));
        int[] numbers2 = {2, 4, 3, 1, 4};
        System.out.println(duplicate(numbers2));
        int[] numbers3 = {2, 4, 2, 1, 4};
        System.out.println(duplicate(numbers3));
        int[] numbers4 = {2, 1, 3, 0, 4};
        System.out.println(duplicate(numbers4));
        int[] numbers5 = {2, 1, 3, 5, 4};
        System.out.println(duplicate(numbers5));
    }
}
```

#

运行结果



49

构建乘积数组

# #

题目：给定一个数组 A[0,1,…,n−1]，请构建一个数组 B[0,1,…,n−1]，其中 B 中的元素 B[i]=A[0]×A[1]×…×A[i−1]×A[i+1]×…×A[n−1]，不能使用除法。

#

解题思路

例如：

A[]={1,2,3}求B[]

B[0]=A[1]×A[2]=2×3=6

B[1]=A[0]×A[2]=1×3=3

B[2]=A[0]×A[1]=1×2=2

1. B[0]初始化为 1，从下标 i=1 开始，先求出 C[i]的值并放入 B[i],即 B[i]=C[i]=C[i−1]×A[i−1]，所以 B[1]=B[1−1]×A[1−1]=B[0]×A[0]=1×1=1，i++
2. B[2]=B[2−1]×A[2−1]=B[1]×A[1]=1×2=2，i++超出长度停止循环
3. C[i]计算完毕求 D[i],设置一个临时变量 temp 初始化为 1
4. 从后往前变量数组，LengthA=3 初始化 i=LengthA−2=1，结束条件为 i>=0
5. 第一次循环，temp=temp×A[i+1]=1×A[2]=3，计算出A中最后一个元素的值放入 temp，temp 相当于 D[i]的值
6. 因为之前的 B[i]=C[i]，所以让 B[i]×D[i] 就是要保存的结果，即 B[i]=B[1]=B[1]×temp=1×3=3,i－=0
7. 计算 B[i]=B[0]，temp上一步中的值是A[2]，在这次循环中 temp=temp×A[0+1]=A[2]×A[1]=3×2=6
8. B[i]=B[0]=B0]×temp=1×6=6，i－<0循 环结束

所以 B 数组为{6,3,2}

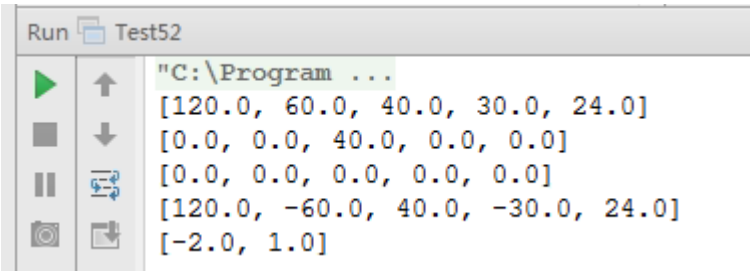
#

代码实现

```
import java.util.Arrays;
public class Test52 {
    public static double[] multiply(double[] data) {
        if (data == null || data.length < 2) {
            return null;
        }
        double[] result = new double[data.length];
        // result[0]取1
        result[0] = 1;
        for (int i = 1; i < data.length; i++) {
            // 第一步每个result[i]都等于于data[0]*data[1]...data[i-1]
            // 当i=n-1时，此时result[n-1]的结果已经计算出来了【A】
            result[i] = result[i -1] * data[i - 1];
        }
        // tmp保存data[n-1]*data[n-2]...data[i+1]的结果
        double tmp = 1;
        // 第二步求data[n-1]*data[n-2]...data[i+1]
        //【A】result[n-1]的结果已经计算出来，所以从data.length-2开始操作
        for (int i = data.length - 2; i >= 0; i--) {
            tmp *= data[i + 1];
            result[i] *= tmp;
        }
        return result;
    }
    public static void main(String[] args) {
        double[] array1 = {1, 2, 3, 4, 5};
        System.out.println(Arrays.toString(multiply(array1))); // double expected[] = {120, 60, 40, 30, 24};
        double[] array2 = {1, 2, 0, 4, 5};
        System.out.println(Arrays.toString(multiply(array2))); // double expected[] = {0, 0, 40, 0, 0};
        double[] array3 = {1, 2, 0, 4, 0};
        System.out.println(Arrays.toString(multiply(array3))); // double expected[] = {0, 0, 0, 0, 0};
        double[] array4 = {1, -2, 3, -4, 5};
        System.out.println(Arrays.toString(multiply(array4))); // double expected[] = {120, -60, 40, -30, 24};
        double[] array5 = {1, -2};
        System.out.println(Arrays.toString(multiply(array5))); // double expected[] = {-2, 1};
    }
}
```

#

运行结果


Run
Test52
"C:\Program ...
[120.0, 60.0, 40.0, 30.0, 24.0]
[0.0, 0.0, 40.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0]
[120.0, -60.0, 40.0, -30.0, 24.0]
[-2.0, 1.0]

50

正则表达式匹配

# #

题目：请实现一个函数用来匹配包含‘.’和‘’*的正则表达式。模式中的字符’.’表示任意一个字符，而‘’*表示它前面的字符可以出现任意次（含0次）。本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串“aaa”与模式“a.a”和“ab*aca”匹配，但与“aa.a”及“ab*a”均不匹配。

#

题目解析

每次从字符串里拿出一个字符和模式中的字符去匹配。先来分析如何匹配一个字符。如果模式中的字符 ch 是‘.’，那么它可以匹配字符串中的任意字符。如果模式中的字符 ch不是’.’而且字符串中的字符也是 ch，那么他们相互匹配。当字符串中的字符和模式中的字符相匹配时，接着匹配后面的字符。

相对而言当模式中的第二个字符不是‘*’时问题要简单很多。如果字符串中的第一个字符和模式中的第一个字符相匹配，那么在字符串和模式上都向后移动一个字符，然后匹配剩余的字符串和模式。如果字符串中的第一个字符和模式中的第一个字符不相匹配，则直接返回 false。

当模式中的第二个字符是‘’*时问题要复杂一些，因为可能有多种不同的匹配方式。一个选择是在模式上向后移动两个字符。这相当于‘’*和它面前的字符被忽略掉了，因为‘*’可以匹配字符串中 0 个字符。如果模式中的第一个字符和字符串中的第一个字符相匹配时，则在字符串向后移动一个字符，而在模式上有两个选择：我们可以在模式上向后移动两个字符，也可以保持模式不变。

#

代码实现

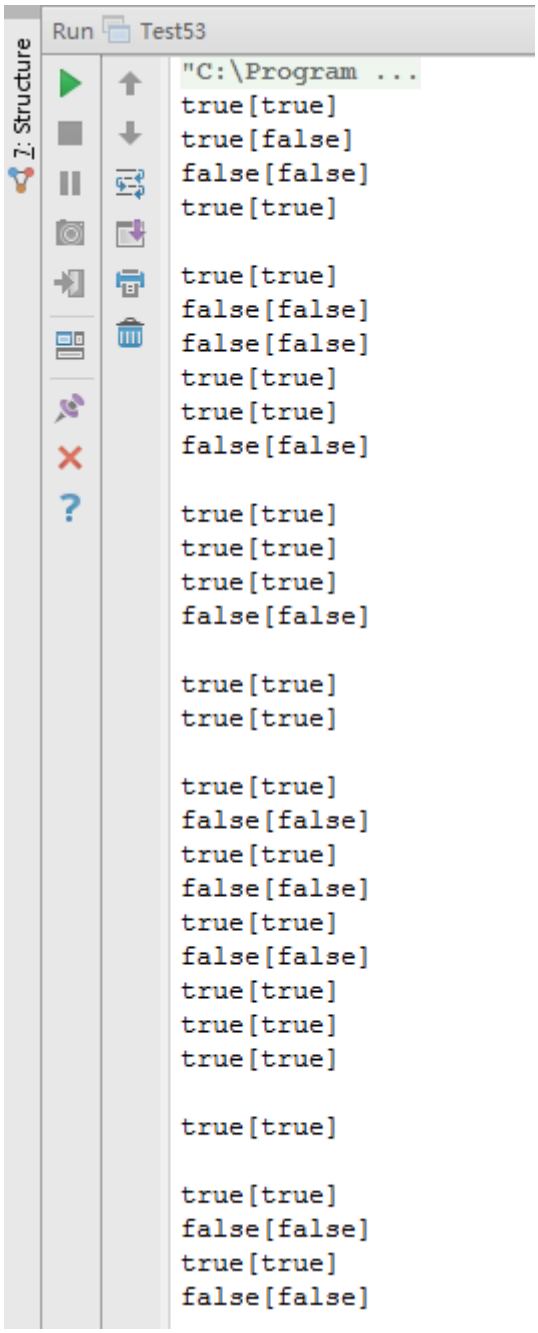
```
public class Test53 {
    /**
     * 题目：请实现一个函数用来匹配包含‘.’和‘*’的正则表达式。模式中的字符'.'表示任意一个字符，
     * 而‘*’表示它前面的字符可以出现任意次（含0次）。本题中，匹配是指字符串的所有字符匹配整个模式。
     *
     * @param input
     * @param pattern
     * @return
     */
    public static boolean match(String input, String pattern) {
```

```
    if (input == null || pattern == null) {
        return false;
    }
    return matchCore(input, 0, pattern, 0);
}
private static boolean matchCore(String input, int i, String pattern, int p) {
    // 匹配串和模式串都到达尾，说明成功匹配
    if (i >= input.length() && p >= pattern.length()) {
        return true;
    }
    // 只有模式串到达结尾，说明匹配失败
    if (i != input.length() && p >= pattern.length()) {
        return false;
    }
    // 模式串未结束，匹配串有可能结束有可能未结束
    // p位置的下一个字符中为*号
    if (p + 1 < pattern.length() && pattern.charAt(p + 1) == '*') {
        // 匹配串已经结束
        if (i >= input.length()) {
            return matchCore(input, i, pattern, p + 2);
        }
        // 匹配串还没有结束
        else {
            if (pattern.charAt(p) == input.charAt(i) || pattern.charAt(p) == '.') {
                return
                        // 匹配串向后移动一个位置，模式串向后移动两个位置
                        matchCore(input, i + 1, pattern, p + 2)
                                // 匹配串向后移动一个位置，模式串不移动
                                || matchCore(input, i + 1, pattern, p)
                                // 匹配串不移动，模式串向后移动两个位置
                                || matchCore(input, i, pattern, p + 2);
            } else {
                return matchCore(input, i, pattern, p + 2);
            }
        }
    }
    //
    // 匹配串已经结束
    if (i >= input.length()) {
        return false;
    }
    // 匹配串还没有结束
    else {
        if (input.charAt(i) == pattern.charAt(p) || pattern.charAt(p) == '.') {
            return matchCore(input, i + 1, pattern, p + 1);
```

```
            }
        }
        return false;
    }
    public static void main(String[] args) {
        System.out.println(match("", "") + "[" + true + "]");
        System.out.println(match("", ".*") + "[" + false + "]");
        System.out.println(match("", ".") + "[" + false + "]");
        System.out.println(match("", "c*") + "[" + true + "]");
        System.out.println();
        System.out.println(match("a", ".*") + "[" + true + "]");
        System.out.println(match("a", "a.") + "[" + false + "]");
        System.out.println(match("a", "") + "[" + false + "]");
        System.out.println(match("a", ".") + "[" + true + "]");
        System.out.println(match("a", "ab*") + "[" + true + "]");
        System.out.println(match("a", "ab*a") + "[" + false + "]");
        System.out.println();
        System.out.println(match("aa", "aa") + "[" + true + "]");
        System.out.println(match("aa", "a*") + "[" + true + "]");
        System.out.println(match("aa", ".*") + "[" + true + "]");
        System.out.println(match("aa", ".") + "[" + false + "]");
        System.out.println();
        System.out.println(match("ab", ".*") + "[" + true + "]");
        System.out.println(match("ab", ".*") + "[" + true + "]");
        System.out.println();
        System.out.println(match("aaa", "aa*") + "[" + true + "]");
        System.out.println(match("aaa", "aa.a") + "[" + false + "]");
        System.out.println(match("aaa", "a.a") + "[" + true + "]");
        System.out.println(match("aaa", ".a") + "[" + false + "]");
        System.out.println(match("aaa", "a*a") + "[" + true + "]");
        System.out.println(match("aaa", "ab*a") + "[" + false + "]");
        System.out.println(match("aaa", "ab*ac*a") + "[" + true + "]");
        System.out.println(match("aaa", "ab*a*c*a") + "[" + true + "]");
        System.out.println(match("aaa", ".*") + "[" + true + "]");
        System.out.println();
        System.out.println(match("aab", "c*a*b") + "[" + true + "]");
        System.out.println();
        System.out.println(match("aaca", "ab*a*c*a") + "[" + true + "]");
        System.out.println(match("aaba", "ab*a*c*a") + "[" + false + "]");
        System.out.println(match("bbbba", ".*a*a") + "[" + true + "]");
        System.out.println(match("bcbbabab", ".*a*a") + "[" + false + "]");
    }
}
```

#

运行结果


Run
Test53
7: Structure
"C:\Program ...
true[true]
true[false]
false[false]
true[true]
true[true]
false[false]
false[false]
true[true]
true[true]
false[false]
true[true]
true[true]
true[true]
false[false]
true[true]
true[true]
true[true]
false[false]
true[true]
false[false]
true[true]
false[false]
true[true]
true[true]
true[true]
true[true]
true[true]
false[false]
true[true]
false[false]

51

表示数值的字符串

# #

题目：请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。

#

例子说明

例如，字符串“+100”，“5e2”，“-123”，“3.1416”及”-1E-16”都表示数值，但“12e”，”1a3.14”，”1.2.3”，”+-5”及“12e+5.4”都不是。

#

解题思路

在数值之前可能有一个表示正负的’-‘或者’+’。接下来是若干个 0 到 9 的数位表示数值的整数部分（在某些小数里可能没有数值的整数部分）。如果数值是一个小数，那么在小数点后面可能会有若干个 0 到 9 的数位表示数值的小数部分。如果数值用科学计数法表示，接下来是一个’e’或者‘E’，以及紧跟着的一个整数（可以有正负号）表示指数。

判断一个字符串是否符合上述模式时，首先看第一个字符是不是正负号。如果是，在字符串上移动一个字符，继续扫描剩余的字符串中 0 到 9 的数位。如果是一个小数，则将遇到小数点。另外，如果是用科学计数法表示的数值，在整数或者小数的后面还有可能遇到’e’或者’E’。

#

代码实现

```
public class Test54 {
    /**
     * 题目：请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。
     *
     * @param string
     * @return
     */
    public static boolean isNumeric(String string) {
        if (string == null || string.length() < 1) {
```

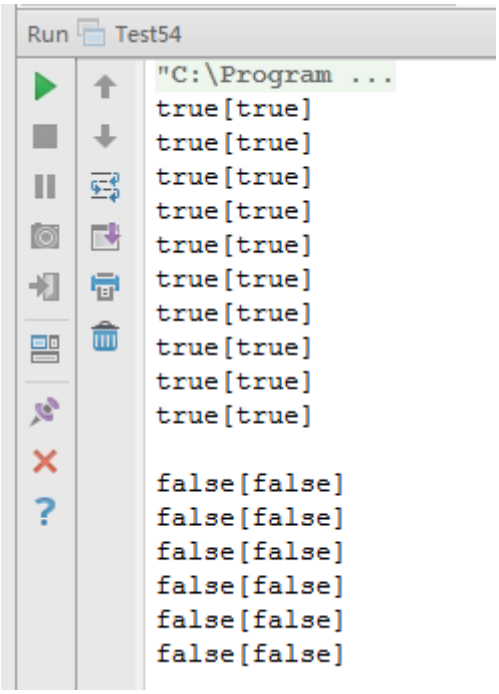
```
        return false;
    }
    int index = 0;
    if (string.charAt(index) == '+' || string.charAt(index) == '-') {
        index++;
    }
    // 已经到达字符串的末尾了
    if (index >= string.length()) {
        return false;
    }
    boolean numeric = true;
    index = scanDigits(string, index);
    // 还未到字符串的末尾
    if (index < string.length()) {
        // 如果是小数点
        if (string.charAt(index) == '.') {
            // 移动到下一个位置
            index++;
            index = scanDigits(string, index);
            // 已经到了字符串的末尾了
            if (index >= string.length()) {
                numeric = true;
            }
            // 还未到字符串结束位置
            else if (index < string.length() && (string.charAt(index) == 'e' || string.charAt(index) == 'E')) {
                numeric = isExponential(string, index);
            } else {
                numeric = false;
            }
        }
        // 如果是指数标识
        else if (string.charAt(index) == 'e' || string.charAt(index) == 'E') {
            numeric = isExponential(string, index);
        } else {
            numeric = false;
        }
        return numeric;
    }
    // 已经到了字符串的末尾了，说明其没有指数部分
    else {
        return true;
    }
}
/**
 * 判断是否是科学计数法的结尾部分，如E5，e5，E+5，e-5，e(E)后面接整数
```

```
     *
     * @param string 字符串
     * @param index  开始匹配的位置
     * @return 匹配的结果
     */
    private static boolean isExponential(String string, int index) {
        if (index >= string.length() || (string.charAt(index) != 'e' && string.charAt(index) != 'E')) {
            return false;
        }
        // 移动到下一个要处理的位置
        index++;
        // 到达字符串的末尾，就返回false
        if (index >= string.length()) {
            return false;
        }
        if (string.charAt(index) == '+' || string.charAt(index) == '-') {
            index++;
        }
        // 到达字符串的末尾，就返回false
        if (index >= string.length()) {
            return false;
        }
        index = scanDigits(string, index);
        // 如果已经处理到了的数字的末尾就认为是正确的指数
        return index >= string.length();
    }
    /**
     * 扫描字符串部分的数字部分
     *
     * @param string 字符串
     * @param index  开始扫描的位置
     * @return 从扫描位置开始第一个数字字符的位置
     */
    private static int scanDigits(String string, int index) {
        while (index < string.length() && string.charAt(index) >= '0' && string.charAt(index) <= '9') {
            index++;
        }
        return index;
    }
    public static void main(String[] args) {
        System.out.println(isNumeric("100") + "[" + true + "]");
        System.out.println(isNumeric("123.45e+6") + "[" + true + "]");
        System.out.println(isNumeric("+500") + "[" + true + "]");
        System.out.println(isNumeric("5e2") + "[" + true + "]");
        System.out.println(isNumeric("3.1416") + "[" + true + "]");
```

```
        System.out.println(isNumeric("600.") + "[" + true + "]");
        System.out.println(isNumeric("−.123") + "[" + true + "]");
        System.out.println(isNumeric("−1E−16") + "[" + true + "]");
        System.out.println(isNumeric("100") + "[" + true + "]");
        System.out.println(isNumeric("1.79769313486232E+308") + "[" + true + "]");
        System.out.println();
        System.out.println(isNumeric("12e") + "[" + false + "]");
        System.out.println(isNumeric("1a3.14") + "[" + false + "]");
        System.out.println(isNumeric("1+23") + "[" + false + "]");
        System.out.println(isNumeric("1.2.3") + "[" + false + "]");
        System.out.println(isNumeric("+−5") + "[" + false + "]");
        System.out.println(isNumeric("12e+5.4") + "[" + false + "]");
    }
}
```

#

运行结果



52

字符流中第一个不重复的字符

# #

题目：请实现一个函数用来找出字符流中第一个只出现一次的字符。

#

举例说明

例如，当从字符流中只读出前两个字符“go”时，第一个只出现一次的字符是‘g’。当从该字符流中读出前六个字符“google”时，第一个只出现 1 次的字符是”l”。

#

解题思路

字符只能一个接着一个从字符流中读出来。可以定义一个数据容器来保存字符在字符流中的位置。当一个字符第一次从字符流中读出来时，把它在字符流中的位置保存到数据容器里。当这个字符再次从字符流中被读出来时，那么它就不是只出现一次的字符，也就可以被忽略了。这时把它在数据容器里保存的值更新成一个特殊的值（比如负值）。

为了尽可能高校地解决这个问题，需要在 O(1)时间内往容器里插入一个字符，以及更新一个字符对应的值。这个容器可以用哈希表来实现。用字符的 ASCII 码作为哈希表的键值，而把字符对应的位置作为哈希表的值。

#

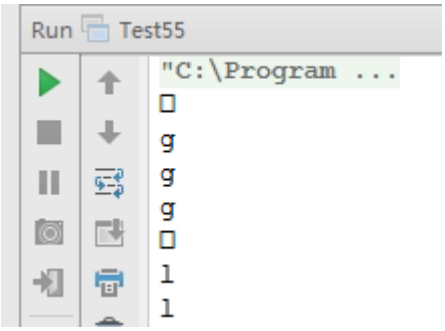
代码实现

```
public class Test55 {
    /**
     * 题目：请实现一个函数用来找出字符流中第一个只出现一次的字符。
     */
    private static class CharStatistics {
        // 出现一次的标识
        private int index = 0;
        private int[] occurrence = new int[256];
        public CharStatistics() {
            for (int i = 0; i < occurrence.length; i++) {
```

```
            occurrence[i] = -1;
        }
    }
    private void insert(char ch) {
        if (ch > 255) {
            throw new IllegalArgumentException( ch + "must be a ASCII char");
        }
        // 只出现一次
        if (occurrence[ch] == -1) {
            occurrence[ch] = index;
        } else {
            // 出现了两次
            occurrence[ch] = -2;
        }
        index++;
    }
    public char firstAppearingOnce(String data) {
        if (data == null) {
            throw new IllegalArgumentException(data);
        }
        for (int i = 0; i < data.length(); i++) {
            insert(data.charAt(i));
        }
        char ch = '\0';
        // 用于记录最小的索引，对应的就是第一个不重复的数字
        int minIndex = Integer.MAX_VALUE;
        for (int i = 0; i < occurrence.length; i++) {
            if (occurrence[i] >= 0 && occurrence[i] < minIndex) {
                ch = (char) i;
                minIndex = occurrence[i];
            }
        }
        return ch;
    }
}
public static void main(String[] args) {
    System.out.println(new CharStatistics().firstAppearingOnce("")); // '\0'
    System.out.println(new CharStatistics().firstAppearingOnce("g")); // 'g'
    System.out.println(new CharStatistics().firstAppearingOnce("go")); // 'g'
    System.out.println(new CharStatistics().firstAppearingOnce("goo")); // 'g'
    System.out.println(new CharStatistics().firstAppearingOnce("goog")); // '\0'
    System.out.println(new CharStatistics().firstAppearingOnce("googl")); // l
    System.out.println(new CharStatistics().firstAppearingOnce("google")); // l
}
}
```

#

运行结果



53

链表中环的入口结点

# #

题目：一个链表中包含环，如何找出环的入口结点?

#

解题思路

可以用两个指针来解决这个问题。先定义两个指针 P1 和 P2 指向链表的头结点。如果链表中环有 n 个结点，指针 P1 在链表上向前移动 n 步，然后两个指针以相同的速度向前移动。当第二个指针指向环的入口结点时，第一个指针已经围绕着环走了一圈又回到了入口结点。

剩下的问题就是如何得到环中结点的数目。我们在面试题 15 的第二个相关题目时用到了一快一慢的两个指针。如果两个指针相遇，表明链表中存在环。两个指针相遇的结点一定是在环中。可以从这个结点出发，一边继续向前移动一边计数，当再次回到这个结点时就可以得到环中结点数了。

#

结点定义

```
private static class ListNode {
    private int val;
    private ListNode next;
    public ListNode() {
    }
    public ListNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val +"";
    }
}
```

#

代码实现

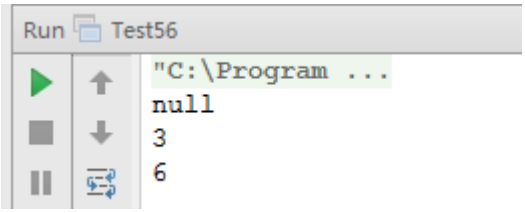
```
public class Test56 {
    private static class ListNode {
        private int val;
        private ListNode next;
        public ListNode() {
        }
        public ListNode(int val) {
            this.val = val;
        }
        @Override
        public String toString() {
            return val +"";
        }
    }
    public static ListNode meetingNode(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast != null && fast.next != null) {
            fast = fast.next.next;
            slow = slow.next;
            if (fast == slow) {
                break;
            }
        }
        // 链表中没有环
        if (fast == null || fast.next == null) {
            return null;
        }
        // fast重新指向第一个结点
        fast = head;
        while (fast != slow) {
            fast = fast.next;
            slow = slow.next;
        }
        return fast;
    }
    public static void main(String[] args) {
        test01();
        test02();
        test03();
    }
    // 1->2->3->4->5->6
    private static void test01() {
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(2);
```

```
    ListNode n3 = new ListNode(3);
    ListNode n4 = new ListNode(4);
    ListNode n5 = new ListNode(5);
    ListNode n6 = new ListNode(6);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;
    System.out.println(meetingNode(n1));
}
// 1->2->3->4->5->6
//      ^       |
//      |       |
//      +--------+
private static void test02() {
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(2);
    ListNode n3 = new ListNode(3);
    ListNode n4 = new ListNode(4);
    ListNode n5 = new ListNode(5);
    ListNode n6 = new ListNode(6);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;
    n6.next = n3;
    System.out.println(meetingNode(n1));
}
// 1->2->3->4->5->6 <-+
//              |  |
//              +---+
private static void test03() {
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(2);
    ListNode n3 = new ListNode(3);
    ListNode n4 = new ListNode(4);
    ListNode n5 = new ListNode(5);
    ListNode n6 = new ListNode(6);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;
```

```
        n6.next = n6;
        System.out.println(meetingNode(n1));
    }
}
```

#

运行结果



54

删除链表中重复的结点

# #

题目：在一个排序的链表中，如何删除重复的结点?

#

解题思路

解决这个问题的第一步是确定删除的参数。当然这个函数需要输入待删除链表的头结点。头结点可能与后面的结点重复，也就是说头结点也可能被删除，所以在链表头添加一个结点。

接下来我们从头遍历整个链表。如果当前结点的值与下一个结点的值相同，那么它们就是重复的结点，都可以被删除。为了保证删除之后的链表仍然是相连的而没有中间断开，我们要把当前的前一个结点和后面值比当前结点的值要大的结点相连。我们要确保prev要始终与下一个没有重复的结点连接在一起。

#

结点定义

```
private static class ListNode {
    private int val;
    private ListNode next;
    public ListNode() {
    }
    public ListNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

代码实现

```
public class Test57 {
    private static class ListNode {
        private int val;
        private ListNode next;
        public ListNode() {
        }
        public ListNode(int val) {
            this.val = val;
        }
        @Override
        public String toString() {
            return val + "";
        }
    }
    private static ListNode deleteDuplication(ListNode head) {
        // 为null
        if (head == null) {
            return null;
        }
//        // 只有一个结点
//        if (head.next == null) {
//            return head;
//        }
        // 临时的头结点
        ListNode root = new ListNode();
        root.next = head;
        // 记录前驱结点
        ListNode prev = root;
        // 记录当前处理的结点
        ListNode node = head;
        while (node != null && node.next != null) {
            // 有重复结点，与node值相同的结点都要删除
            if (node.val == node.next.val) {
                // 找到下一个不同值的节点，注意其有可能也是重复节点
                while (node.next != null && node.next.val == node.val) {
                    node = node.next;
                }
                // 指向下一个节点，prev.next也可能是重复结点
                // 所以prev不要移动到下一个结点
                prev.next = node.next;
            }
            // 相邻两个值不同，说明node不可删除，要保留
            else {
                prev.next = node;
                prev = prev.next;
```

```
            }
            node = node.next;
        }
        return root.next;
    }
    private static void print(ListNode head) {
        while (head != null) {
            System.out.print(head + "->");
            head = head.next;
        }
        System.out.println("null");
    }
    public static void main(String[] args) {
        test01();
        test02();
        test03();
        test04();
        test05();
        test06();
        test07();
        test08();
        test09();
        test10();
    }
    // 1->2->3->3->4->4->5
    private static void test01() {
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(2);
        ListNode n3 = new ListNode(3);
        ListNode n4 = new ListNode(3);
        ListNode n5 = new ListNode(4);
        ListNode n6 = new ListNode(4);
        ListNode n7 = new ListNode(5);
        n1.next = n2;
        n2.next = n3;
        n3.next = n4;
        n4.next = n5;
        n5.next = n6;
        n6.next = n7;
        ListNode result = deleteDuplication(n1);
        print(result);
    }
    // 1->2->3->4->5->6->7
    private static void test02() {
        ListNode n1 = new ListNode(1);
```

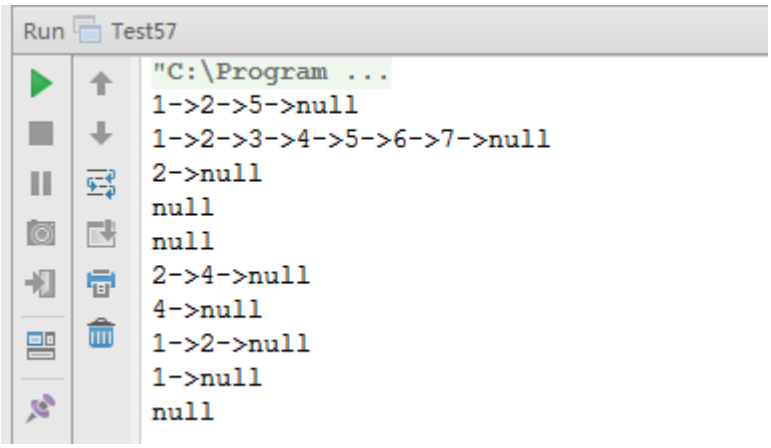
```
    ListNode n2 = new ListNode(2);
    ListNode n3 = new ListNode(3);
    ListNode n4 = new ListNode(4);
    ListNode n5 = new ListNode(5);
    ListNode n6 = new ListNode(6);
    ListNode n7 = new ListNode(7);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;
    n6.next = n7;
    ListNode result = deleteDuplication(n1);
    print(result);
}
// 1->1->1->1->1->1->2
private static void test03() {
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(1);
    ListNode n3 = new ListNode(1);
    ListNode n4 = new ListNode(1);
    ListNode n5 = new ListNode(1);
    ListNode n6 = new ListNode(1);
    ListNode n7 = new ListNode(2);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;
    n6.next = n7;
    ListNode result = deleteDuplication(n1);
    print(result);
}
// 1->1->1->1->1->1->1
private static void test04() {
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(1);
    ListNode n3 = new ListNode(1);
    ListNode n4 = new ListNode(1);
    ListNode n5 = new ListNode(1);
    ListNode n6 = new ListNode(1);
    ListNode n7 = new ListNode(1);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
```

```
    n4.next = n5;
    n5.next = n6;
    n6.next = n7;
    ListNode result = deleteDuplication(n1);
    print(result);
}
// 1->1->2->2->3->3->4->4
private static void test05() {
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(1);
    ListNode n3 = new ListNode(2);
    ListNode n4 = new ListNode(2);
    ListNode n5 = new ListNode(3);
    ListNode n6 = new ListNode(3);
    ListNode n7 = new ListNode(4);
    ListNode n8 = new ListNode(4);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;
    n6.next = n7;
    n7.next = n8;
    ListNode result = deleteDuplication(n1);
    print(result);
}
// 1->1->2->3->3->4->5->5
private static void test06() {
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(1);
    ListNode n3 = new ListNode(2);
    ListNode n4 = new ListNode(3);
    ListNode n5 = new ListNode(3);
    ListNode n6 = new ListNode(4);
    ListNode n7 = new ListNode(5);
    ListNode n8 = new ListNode(5);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;
    n6.next = n7;
    n7.next = n8;
    ListNode result = deleteDuplication(n1);
    print(result);
```

```
    }
    // 1->1->2->2->3->3->4->5->5
    private static void test07() {
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(1);
        ListNode n3 = new ListNode(2);
        ListNode n4 = new ListNode(2);
        ListNode n5 = new ListNode(3);
        ListNode n6 = new ListNode(3);
        ListNode n7 = new ListNode(4);
        ListNode n8 = new ListNode(5);
        ListNode n9 = new ListNode(5);
        n1.next = n2;
        n2.next = n3;
        n3.next = n4;
        n4.next = n5;
        n5.next = n6;
        n6.next = n7;
        n7.next = n8;
        n8.next = n9;
        ListNode result = deleteDuplication(n1);
        print(result);
    }
    // 1->2
    private static void test08() {
        ListNode n1 = new ListNode(1);
        ListNode n2 = new ListNode(2);
        n1.next = n2;
        ListNode result = deleteDuplication(n1);
        print(result);
    }
    // 1
    private static void test09() {
        ListNode n1 = new ListNode(1);
        ListNode result = deleteDuplication(n1);
        print(result);
    }
    // null
    private static void test10() {
        ListNode result = deleteDuplication(null);
        print(result);
    }
}
```

#

运行结果


Run
Test57
"C:\Program ...
1->2->5->null
1->2->3->4->5->6->7->null
2->null
null
null
2->4->null
4->null
1->2->null
1->null
null

55

二叉树的下一个结点

# #

题目：给定一棵二叉树和其中的一个结点，如何找出中序遍历顺序的下一个结点？树中的结点除了有两个分别指向左右子结点的指针以外，还有一个指向父节点的指针。

#

解题思路

如果一个结点有右子树，那么它的下一个结点就是它的右子树中的左子结点。也就是说右子结点出发一直沿着指向左子结点的指针，我们就能找到它的下一个结点。

接着我们分析一个结点没有右子树的情形。如果结点是它父节点的左子结点，那么它的下一个结点就是它的父结点。

如果一个结点既没有右子树，并且它还是它父结点的右子结点，这种情形就比较复杂。我们可以沿着指向父节点的指针一直向上遍历，直到找到一个是它父结点的左子结点的结点。如果这样的结点存在，那么这个结点的父结点就是我们要找的下一个结点。

#

结点定义

```
private static class BinaryTreeNode {
    private int val;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    private BinaryTreeNode parent;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

代码实现

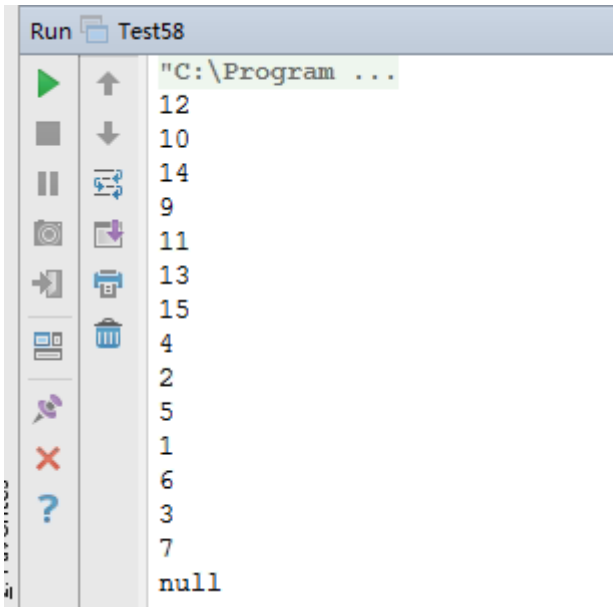
```
public class Test58 {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;
        private BinaryTreeNode parent;
        public BinaryTreeNode() {
        }
        public BinaryTreeNode(int val) {
            this.val = val;
        }
        @Override
        public String toString() {
            return val + "";
        }
    }
    public static BinaryTreeNode getNext(BinaryTreeNode node) {
        if (node == null) {
            return null;
        }
        // 保存要查找的下一个节点
        BinaryTreeNode target = null;
        if (node.right != null) {
            target = node.right;
            while (target.left != null) {
                target = target.left;
            }
            return target;
        } else if (node.parent != null){
            target = node.parent;
            BinaryTreeNode cur = node;
            // 如果父新结点不为空，并且，子结点不是父结点的左孩子
            while (target != null && target.left != cur) {
                cur = target;
                target = target.parent;
            }
            return target;
        }
        return null;
```

```
}
private static void assemble(BinaryTreeNode node,
                             BinaryTreeNode left,
                             BinaryTreeNode right,
                             BinaryTreeNode parent) {
    node.left = left;
    node.right = right;
    node.parent = parent;
}
public static void main(String[] args) {
    test01();
}
//                          1
//               2                    3
//           4       5        6         7
//        8    9   10   11   12   13    14   15
public static void test01() {
    BinaryTreeNode n1 = new BinaryTreeNode(1); // 12
    BinaryTreeNode n2 = new BinaryTreeNode(2); // 10
    BinaryTreeNode n3 = new BinaryTreeNode(3); // 14
    BinaryTreeNode n4 = new BinaryTreeNode(4); // 9
    BinaryTreeNode n5 = new BinaryTreeNode(5); // 11
    BinaryTreeNode n6 = new BinaryTreeNode(6); // 13
    BinaryTreeNode n7 = new BinaryTreeNode(7); // 15
    BinaryTreeNode n8 = new BinaryTreeNode(8); // 4
    BinaryTreeNode n9 = new BinaryTreeNode(9); // 2
    BinaryTreeNode n10 = new BinaryTreeNode(10); // 5
    BinaryTreeNode n11 = new BinaryTreeNode(11); // 1
    BinaryTreeNode n12 = new BinaryTreeNode(12); // 6
    BinaryTreeNode n13 = new BinaryTreeNode(13); // 3
    BinaryTreeNode n14 = new BinaryTreeNode(14); // 7
    BinaryTreeNode n15 = new BinaryTreeNode(15); // null
    assemble(n1, n2, n3, null);
    assemble(n2, n4, n5, n1);
    assemble(n3, n6, n7, n1);
    assemble(n4, n8, n9, n2);
    assemble(n5, n10, n11, n2);
    assemble(n6, n12, n13, n3);
    assemble(n7, n14, n15, n3);
    assemble(n8, null, null, n4);
    assemble(n9, null, null, n4);
    assemble(n10, null, null, n5);
    assemble(n11, null, null, n5);
    assemble(n12, null, null, n6);
    assemble(n13, null, null, n6);
```

```
        assemble(n14, null, null, n7);
        assemble(n15, null, null, n7);
        System.out.println(getNext(n1));
        System.out.println(getNext(n2));
        System.out.println(getNext(n3));
        System.out.println(getNext(n4));
        System.out.println(getNext(n5));
        System.out.println(getNext(n6));
        System.out.println(getNext(n7));
        System.out.println(getNext(n8));
        System.out.println(getNext(n9));
        System.out.println(getNext(n10));
        System.out.println(getNext(n11));
        System.out.println(getNext(n12));
        System.out.println(getNext(n13));
        System.out.println(getNext(n14));
        System.out.println(getNext(n15));
    }
}
```

#

运行结果



56 对称的二叉树

# #

题目：请实现一个函数来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

#

解题思路

通常我们有三种不同的二叉树遍历算法，即前序遍历、中序遍历和后序遍历。在这三种遍历算法中，都是先遍历左子结点再遍历右子结点。我们是否可以定义一种遍历算法，先遍历右子结点再遍历左子结点？比如我们针对前序遍历定义一种对称的遍历算法，即先遍历父节点，再遍历它的右子结点，最后遍历它的左子结点。

我们发现可以用过比较二叉树的前序遍历序列和对称前序遍历序列来判断二叉树是不是对称的。如果两个序列一样，那么二叉树就是对称的。

#

结点定义

```
private static class BinaryTreeNode {
    private int val;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

代码实现

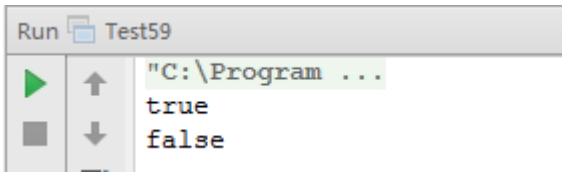
```
public class Test59 {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;
        public BinaryTreeNode() {
        }
        public BinaryTreeNode(int val) {
            this.val = val;
        }
        @Override
        public String toString() {
            return val + "";
        }
    }
    public static boolean isSymmetrical(BinaryTreeNode root) {
        return isSymmetrical(root, root);
    }
    private static boolean isSymmetrical(BinaryTreeNode left, BinaryTreeNode right) {
        if (left == null && right == null) {
            return true;
        }
        if (left == null || right == null) {
            return false;
        }
        if (left.val != right.val ) {
            return false;
        }
        return isSymmetrical(left.left, right.right) && isSymmetrical(left.right, right.left);
    }
    public static void main(String[] args) {
        test01();
        test02();
    }
    private static void assemble(BinaryTreeNode node,
                                 BinaryTreeNode left,
                                 BinaryTreeNode right) {
        node.left = left;
        node.right = right;
    }
    //                      1
    //              2               2
    //          4       6       6       4
    //        8   9  10  11  11   10  9   8
    public static void test01() {
```

```
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(2);
    BinaryTreeNode n3 = new BinaryTreeNode(2);
    BinaryTreeNode n4 = new BinaryTreeNode(4);
    BinaryTreeNode n5 = new BinaryTreeNode(6);
    BinaryTreeNode n6 = new BinaryTreeNode(6);
    BinaryTreeNode n7 = new BinaryTreeNode(4);
    BinaryTreeNode n8 = new BinaryTreeNode(8);
    BinaryTreeNode n9 = new BinaryTreeNode(9);
    BinaryTreeNode n10 = new BinaryTreeNode(10);
    BinaryTreeNode n11 = new BinaryTreeNode(11);
    BinaryTreeNode n12 = new BinaryTreeNode(11);
    BinaryTreeNode n13 = new BinaryTreeNode(10);
    BinaryTreeNode n14 = new BinaryTreeNode(9);
    BinaryTreeNode n15 = new BinaryTreeNode(8);
    assemble(n1, n2, n3);
    assemble(n2, n4, n5);
    assemble(n3, n6, n7);
    assemble(n4, n8, n9);
    assemble(n5, n10, n11);
    assemble(n6, n12, n13);
    assemble(n7, n14, n15);
    assemble(n8, null, null);
    assemble(n9, null, null);
    assemble(n10, null, null);
    assemble(n11, null, null);
    assemble(n12, null, null);
    assemble(n13, null, null);
    assemble(n14, null, null);
    assemble(n15, null, null);
    System.out.println(isSymmetrical(n1));
}
//                    1
//              2              2
//          4       5       6       4
//        8   9  10  11  11   10  9   8
public static void test02() {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(2);
    BinaryTreeNode n3 = new BinaryTreeNode(2);
    BinaryTreeNode n4 = new BinaryTreeNode(4);
    BinaryTreeNode n5 = new BinaryTreeNode(5);
    BinaryTreeNode n6 = new BinaryTreeNode(6);
    BinaryTreeNode n7 = new BinaryTreeNode(4);
    BinaryTreeNode n8 = new BinaryTreeNode(8);
```

```
        BinaryTreeNode n9 = new BinaryTreeNode(9);
        BinaryTreeNode n10 = new BinaryTreeNode(10);
        BinaryTreeNode n11 = new BinaryTreeNode(11);
        BinaryTreeNode n12 = new BinaryTreeNode(11);
        BinaryTreeNode n13 = new BinaryTreeNode(10);
        BinaryTreeNode n14 = new BinaryTreeNode(9);
        BinaryTreeNode n15 = new BinaryTreeNode(8);
        assemble(n1, n2, n3);
        assemble(n2, n4, n5);
        assemble(n3, n6, n7);
        assemble(n4, n8, n9);
        assemble(n5, n10, n11);
        assemble(n6, n12, n13);
        assemble(n7, n14, n15);
        assemble(n8, null, null);
        assemble(n9, null, null);
        assemble(n10, null, null);
        assemble(n11, null, null);
        assemble(n12, null, null);
        assemble(n13, null, null);
        assemble(n14, null, null);
        assemble(n15, null, null);
        System.out.println(isSymmetrical(n1));
    }
}
```

#

运行结果



57

把二叉树打印出多行

# #

题目：从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层打印一行。

#

解题思路

用一个队列来保存将要打印的结点。为了把二叉树的每一行单独打印到一行里，我们需要两个变量：一个变量表示在当前的层中还没有打印的结点数，另一个变量表示下一次结点的数目。

#

结点定义

```
private static class BinaryTreeNode {
    private int val;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

代码实现

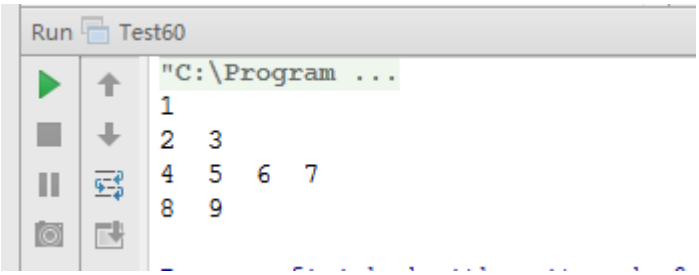
```
import java.util.LinkedList;
import java.util.List;
public class Test60 {
    private static class BinaryTreeNode {
```

```
    private int val;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
/**
 * 题目：从上到下按层打印二叉树，同一层的结点按从左到右的顺序打印，每一层打印一行。
 * @param root
 */
public static void print(BinaryTreeNode root) {
    if (root == null) {
        return;
    }
    List<BinaryTreeNode> list = new LinkedList<>();
    BinaryTreeNode node;
    // 当前层的结点个数
    int current = 1;
    // 记录下一层的结点个数
    int next = 0;
    list.add(root);
    while (list.size() > 0) {
        node = list.remove(0);
        current--;
        System.out.printf("%-3d", node.val);
        if (node.left != null) {
            list.add(node.left);
            next++;
        }
        if (node.right != null) {
            list.add(node.right);
            next++;
        }
        if (current ==0) {
            System.out.println();
            current = next;
            next = 0;
        }
```

```
        }
    }
    public static void main(String[] args) {
        BinaryTreeNode n1 = new BinaryTreeNode(1);
        BinaryTreeNode n2 = new BinaryTreeNode(2);
        BinaryTreeNode n3 = new BinaryTreeNode(3);
        BinaryTreeNode n4 = new BinaryTreeNode(4);
        BinaryTreeNode n5 = new BinaryTreeNode(5);
        BinaryTreeNode n6 = new BinaryTreeNode(6);
        BinaryTreeNode n7 = new BinaryTreeNode(7);
        BinaryTreeNode n8 = new BinaryTreeNode(8);
        BinaryTreeNode n9 = new BinaryTreeNode(9);
        n1.left = n2;
        n1.right = n3;
        n2.left = n4;
        n2.right = n5;
        n3.left = n6;
        n3.right = n7;
        n4.left = n8;
        n4.right = n9;
        print(n1);
    }
}
```

#

运行结果



58

按之字形顺序打印二叉树

# #

题目：请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，即第一行按照从左到右的顺序打印，第二层按照从右到左顺序打印，第三行再按照从左到右的顺序打印，其他以此类推。

#

解题思路

按之字形顺序打印二叉树需要两个栈。我们在打印某一行结点时，把下一层的子结点保存到相应的栈里。如果当前打印的是奇数层，则先保存左子结点再保存右子结点到一个栈里；如果当前打印的是偶数层，则先保存右子结点再保存左子结点到第二个栈里。

#

结点定义

```
private static class BinaryTreeNode {
    private int val;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

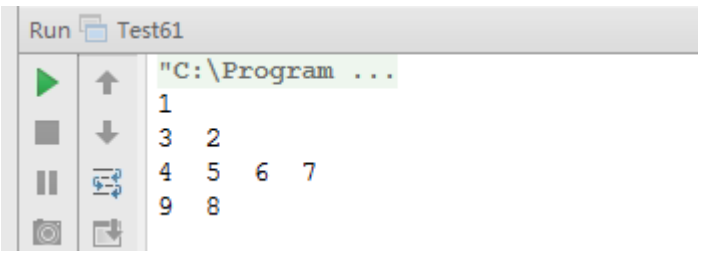
代码实现

```
import java.util.LinkedList;
import java.util.List;
public class Test61 {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;
        public BinaryTreeNode() {
        }
        public BinaryTreeNode(int val) {
            this.val = val;
        }
        @Override
        public String toString() {
            return val + "";
        }
    }
    public static void print(BinaryTreeNode root) {
        if (root == null) {
            return;
        }
        List<BinaryTreeNode> current = new LinkedList<>();
        List<BinaryTreeNode> reverse = new LinkedList<>();
        int flag = 0;
        BinaryTreeNode node;
        current.add(root);
        while (current.size() > 0) {
            // 从最后一个开始取
            node = current.remove(current.size() - 1);
            System.out.printf("%-3d", node.val);
            // 当前是从左往右打印的，那就按从左往右入栈
            if (flag == 0) {
                if (node.left != null) {
                    reverse.add(node.left);
                }
                if (node.right != null) {
                    reverse.add(node.right);
                }
            }
            // 当前是从右往左打印的，那就按从右往左入栈
            else {
                if (node.right != null) {
                    reverse.add(node.right);
                }
                if (node.left != null) {
```

```
                reverse.add(node.left);
            }
        }
        if (current.size() == 0) {
            flag = 1 - flag;
            List<BinaryTreeNode> tmp = current;
            current = reverse;
            reverse = tmp;
            System.out.println();
        }
    }
}
public static void main(String[] args) {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(2);
    BinaryTreeNode n3 = new BinaryTreeNode(3);
    BinaryTreeNode n4 = new BinaryTreeNode(4);
    BinaryTreeNode n5 = new BinaryTreeNode(5);
    BinaryTreeNode n6 = new BinaryTreeNode(6);
    BinaryTreeNode n7 = new BinaryTreeNode(7);
    BinaryTreeNode n8 = new BinaryTreeNode(8);
    BinaryTreeNode n9 = new BinaryTreeNode(9);
    n1.left = n2;
    n1.right = n3;
    n2.left = n4;
    n2.right = n5;
    n3.left = n6;
    n3.right = n7;
    n4.left = n8;
    n4.right = n9;
    print(n1);
}
}
```

#

运行结果



59

序列化二叉树

# #

题目：请实现两个函数，分别用来序列化和反序列化二叉树。

#

解题思路

通过分析解决前面的面试题 6。我们知道可以从前序遍历和中序遍历构造出一棵二叉树。受此启发，我们可以先把一棵二叉树序列化成一个前序遍历序列和一个中序序列，然后再反序列化时通过这两个序列重构出原二叉树。

这个思路有两个缺点。一个缺点是该方法要求二叉树中不能用有数值重复的结点。另外只有当两个序列中所有数据都读出后才能开始反序列化。如果两个遍历序列的数据是从一个流里读出来的，那就可能需要等较长的时间。

实际上如果二叉树的序列化是从根结点开始的话，那么相应的反序列化在根结点的数值读出来的时候就可以开始了。因此我们可以根据前序遍历的顺序来序列化二叉树，因为前序遍历是从根结点开始的。当在遍历二叉树碰到 NULL 指针时，这些 NULL 指针序列化成一个特殊的字符（比如‘$’）。另外，结点的数值之间要用一个特殊字符（比如’,’）隔开。

#

结点定义

```
private static class BinaryTreeNode {
    private int val;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

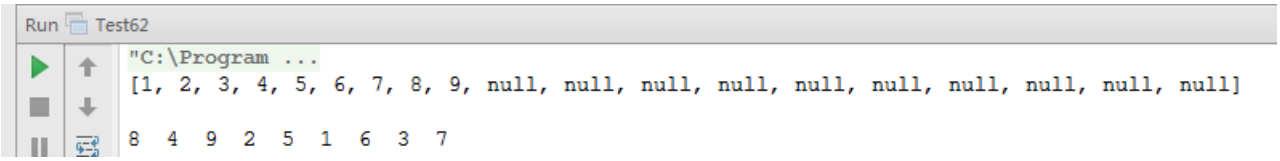
代码实现

```
import java.util.LinkedList;
import java.util.List;
public class Test62 {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
        private BinaryTreeNode right;
        public BinaryTreeNode() {
        }
        public BinaryTreeNode(int val) {
            this.val = val;
        }
        @Override
        public String toString() {
            return val + "";
        }
    }
    public static void serialize(BinaryTreeNode root, List<Integer> result) {
        List<BinaryTreeNode> list = new LinkedList<>();
        list.add(root);
        BinaryTreeNode node;
        while (list.size() > 0) {
            node = list.remove(0);
            if (node == null) {
                result.add(null);
            }else {
                result.add(node.val);
                list.add(node.left);
                list.add(node.right);
            }
        }
    }
    public static BinaryTreeNode deserialize(List<Integer> result, int idx) {
        if (result.size() < 1 || idx < 0 || result.size() <= idx || result.get(idx) == null) {
            return null;
        }
        BinaryTreeNode root = new BinaryTreeNode(result.get(idx));
        root.left = deserialize(result, idx * 2 + 1);
        root.right = deserialize(result, idx * 2 + 2);
```

```
        return root;
    }
    public static void main(String[] args) {
        test01();
    }
    private static void test01() {
        BinaryTreeNode n1 = new BinaryTreeNode(1);
        BinaryTreeNode n2 = new BinaryTreeNode(2);
        BinaryTreeNode n3 = new BinaryTreeNode(3);
        BinaryTreeNode n4 = new BinaryTreeNode(4);
        BinaryTreeNode n5 = new BinaryTreeNode(5);
        BinaryTreeNode n6 = new BinaryTreeNode(6);
        BinaryTreeNode n7 = new BinaryTreeNode(7);
        BinaryTreeNode n8 = new BinaryTreeNode(8);
        BinaryTreeNode n9 = new BinaryTreeNode(9);
        n1.left = n2;
        n1.right = n3;
        n2.left = n4;
        n2.right = n5;
        n3.left = n6;
        n3.right = n7;
        n4.left = n8;
        n4.right = n9;
        List<Integer> result = new LinkedList<>();
        serialize(n1, result);
        System.out.println(result);
        System.out.println();
        BinaryTreeNode root = deserialize(result, 0) ;
        print(root);
    }
    private static void print(BinaryTreeNode root) {
        if (root != null) {
            print(root.left);
            System.out.printf("%-3d", root.val);
            print(root.right);
        }
    }
}
```

#

运行结果


Run
Test62
"C:\Program ...
[1, 2, 3, 4, 5, 6, 7, 8, 9, null, null, null, null, null, null, null, null, null, null]
8 4 9 2 5 1 6 3 7

60

二叉搜索树的第 k 个结点

# #

题目：给定一棵二叉搜索树，请找出其中的第k大的结点。

#

解题思路

如果按照中序遍历的顺序遍历一棵二叉搜索树，遍历序列的数值是递增排序的。只需要用中序遍历算法遍历一棵二叉搜索树，就很容易找出它的第k大结点。

#

结点定义

```
private static class BinaryTreeNode {
    private int val;
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
```

#

代码实现

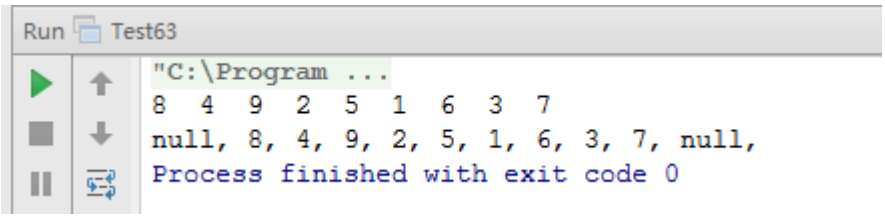
```
public class Test63 {
    private static class BinaryTreeNode {
        private int val;
        private BinaryTreeNode left;
```

```
    private BinaryTreeNode right;
    public BinaryTreeNode() {
    }
    public BinaryTreeNode(int val) {
        this.val = val;
    }
    @Override
    public String toString() {
        return val + "";
    }
}
public static BinaryTreeNode kthNode(BinaryTreeNode root, int k) {
    if (root == null || k < 1) {
        return null;
    }
    int[] tmp = {k};
    return kthNodeCore(root, tmp);
}
private static BinaryTreeNode kthNodeCore(BinaryTreeNode root, int[] k) {
    BinaryTreeNode result = null;
    // 先成左子树中找
    if (root.left != null) {
      result =  kthNodeCore(root.left, k);
    }
    // 如果在左子树中没有找到
    if (result == null) {
        // 说明当前的根结点是所要找的结点
        if(k[0] == 1) {
            result = root;
        } else {
            // 当前的根结点不是要找的结点，但是已经找过了，所以计数器减一
            k[0]--;
        }
    }
    // 根结点以及根结点的右子结点都没有找到，则找其右子树
    if (result == null && root.right != null) {
        result = kthNodeCore(root.right, k);
    }
    return result;
}
public static void main(String[] args) {
    BinaryTreeNode n1 = new BinaryTreeNode(1);
    BinaryTreeNode n2 = new BinaryTreeNode(2);
    BinaryTreeNode n3 = new BinaryTreeNode(3);
    BinaryTreeNode n4 = new BinaryTreeNode(4);
```

```
        BinaryTreeNode n5 = new BinaryTreeNode(5);
        BinaryTreeNode n6 = new BinaryTreeNode(6);
        BinaryTreeNode n7 = new BinaryTreeNode(7);
        BinaryTreeNode n8 = new BinaryTreeNode(8);
        BinaryTreeNode n9 = new BinaryTreeNode(9);
        n1.left = n2;
        n1.right = n3;
        n2.left = n4;
        n2.right = n5;
        n3.left = n6;
        n3.right = n7;
        n4.left = n8;
        n4.right = n9;
        print(n1);
        System.out.println();
        for (int i = 0; i <= 10; i++) {
            System.out.printf(kthNode(n1, i) + ", ");
        }
    }
    /**
     * 中序遍历一棵树
     * @param root
     */
    private static void print(BinaryTreeNode root) {
        if (root != null) {
            print(root.left);
            System.out.printf("%-3d", root.val);
            print(root.right);
        }
    }
}
```

#

运行结果



61

数据流中的中位数

# #

题目：如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有值排序之后位于中间的数值。如果数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

#

解题思路

由于数据是从一个数据流中读出来的，数据的数目随着时间的变化而增加。如果用一个数据容器来保存从流中读出来的数据，当有新的数据流中读出来时，这些数据就插入到数据容器中。这个数据容器用什么数据结构定义更合适呢？

数组是最简单的容器。如果数组没有排序，可以用 Partition 函数找出数组中的中位数。在没有排序的数组中插入一个数字和找出中位数的时间复杂度是 O(1)和 O(n)。

我们还可以往数组里插入新数据时让数组保持排序，这是由于可能要移动 O(n)个数，因此需要 O(n)时间才能完成插入操作。在已经排好序的数组中找出中位数是一个简单的操作，只需要 O(1)时间即可完成。

排序的链表时另外一个选择。我们需要 O(n)时间才能在链表中找到合适的位置插入新的数据。如果定义两个指针指向链表的中间结点（如果链表的结点数目是奇数，那么这两个指针指向同一个结点），那么可以在 O（1）时间得出中位数。此时时间效率与及基于排序的数组的时间效率一样。

二叉搜索树可以把插入新数据的平均时间降低到 O(logn)。但是，当二叉搜索树极度不平衡从而看起来像一个排序的链表时，插入新数据的时间仍然是 O(n)。为了得到中位数，可以在二叉树结点中添加一个表示子树结点数目的字段。有了这个字段，可以在平均 O(logn)时间得到中位数，但差情况仍然是 O(n)。

为了避免二叉搜索树的最差情况，还可以利用平衡的二叉搜索树，即 AVL 树。通常 AVL 树的平衡因子是左右子树的高度差。可以稍作修改，把 AVL 的平衡因子改为左右子树结点数目只差。有了这个改动，可以用 O(logn)时间往 AVL 树中添加一个新结点，同时用 O(1)时间得到所有结点的中位数。

AVL 树的时间效率很高，但大部分编程语言的函数库中都没有实现这个数据结构。应聘者在短短几十分钟内实现 AVL 的插入操作是非常困难的。于是我们不得不再分析还有没有其它的方法。

如果能够保证数据容器左边的数据都小于右边的数据，这样即使左、右两边内部的数据没有排序，也可以根据左边最大的数及右边最小的数得到中位数。如何快速从一个容器中找出最大数？用最大堆实现这个数据容器，因为位于堆顶的就是最大的数据。同样，也可以快速从最小堆中找出最小数。　　因此可以用如下思路来解决这个问

题：用一个最大堆实现左边的数据容器，用最小堆实现右边的数据容器。往堆中插入一个数据的时间效率是 O(log n)。由于只需 O(1)时间就可以得到位于堆顶的数据，因此得到中位数的时间效率是 O(1)。

接下来考虑用最大堆和最小堆实现的一些细节。首先要保证数据平均分配到两个堆中，因此两个堆中数据的数目之差不能超过 1（为了实现平均分配，可以在数据的总数目是偶数时把新数据插入到最小堆中，否则插入到最大堆中）。

还要保证最大堆中里的所有数据都要小于最小堆中的数据。当数据的总数目是偶数时，按照前面分配的规则会把新的数据插入到最小堆中。如果此时新的数据比最大堆中的一些数据要小，怎么办呢？

可以先把新的数据插入到最大堆中，接着把最大堆中的最大的数字拿出来插入到最小堆中。由于最终插入到最小堆的数字是原最大堆中最大的数字，这样就保证了最小堆中的所有数字都大于最大堆中的数字。 当需要把一个数据插入到最大堆中，但这个数据小于最小堆里的一些数据时，这个情形和前面类似。

#

辅助类实现

#

堆实现

```
private static class Heap<T> {
    // 堆中元素存放的集合
    private List<T> data;
    // 比较器
    private Comparator<T> cmp;
    /**
     * 构造函数
     *
     * @param cmp 比较器对象
     */
    public Heap(Comparator<T> cmp) {
        this.cmp = cmp;
        this.data = new ArrayList<>(64);
    }
    /**
     * 向上调整堆
     *
     * @param idx 被上移元素的起始位置
     */
```

```
public void shiftUp(int idx) {
    // 检查是位置是否正确
    if (idx < 0 || idx >= data.size()) {
        throw new IllegalArgumentException(idx + "");
    }
    // 获取开始调整的元素对象
    T intent = data.get(idx);
    // 如果不是根元素，则需要上移
    while (idx > 0) {
        // 找父元素对象的位置
        int parentIdx = (idx - 1) / 2;
        // 获取父元素对象
        T parent = data.get(parentIdx);
        //上移的条件，子节点比父节点大，此处定义的大是以比较器返回值为准
        if (cmp.compare(intent, parent) > 0) {
            // 将父节点向下放
            data.set(idx, parent);
            idx = parentIdx;
            // 记录父节点下放的位置
        }
        // 子节点不比父节点大，说明父子路径已经按从大到小排好顺序了，不需要调整了
        else {
            break;
        }
    }
    // index此时记录是的最后一个被下放的父节点的位置（也可能是自身），
    // 所以将最开始的调整的元素值放入index位置即可
    data.set(idx, intent);
}
/**
 * 向下调整堆
 *
 * @param idx 被下移的元素的起始位置
 */
public void shiftDown(int idx) {
    // 检查是位置是否正确
    if (idx < 0 || idx >= data.size()) {
        throw new IllegalArgumentException(idx + "");
    }
    // 获取开始调整的元素对象
    T intent = data.get(idx);
    // 获取开始调整的元素对象的左子结点的元素位置
    int leftIdx = idx * 2 + 1;
    // 如果有左子结点
    while (leftIdx < data.size()) {
```

```
        // 取左子结点的元素对象，并且假定其为两个子结点中最大的
        T maxChild = data.get(leftIdx);
        // 两个子节点中最大节点元素的位置，假定开始时为左子结点的位置
        int maxIdx = leftIdx;
        // 获取右子结点的位置
        int rightIdx = leftIdx + 1;
        // 如果有右子结点
        if (rightIdx < data.size()) {
            T rightChild = data.get(rightIdx);
            // 找出两个子节点中的最大子结点
            if (cmp.compare(rightChild, maxChild) > 0) {
                maxChild = rightChild;
                maxIdx = rightIdx;
            }
        }
        // 如果最大子节点比父节点大，则需要向下调整
        if (cmp.compare(maxChild, intent) > 0) {
            // 将较大的子节点向上移
            data.set(idx, maxChild);
            // 记录上移节点的位置
            idx = maxIdx;
            // 找到上移节点的左子节点的位置
            leftIdx = 2 * idx + 1;
        }
        // 最大子节点不比父节点大，说明父子路径已经按从大到小排好顺序了，不需要调整了
        else {
            break;
        }
    }
    // index此时记录是的最后一个被上移的子节点的位置（也可能是自身），
    // 所以将最开始的调整的元素值放入index位置即可
    data.set(idx, intent);
}
/**
 * 添加一个元素
 *
 * @param item 添加的元素
 */
public void add(T item) {
    // 将元素添加到最后
    data.add(item);
    // 上移，以完成重构
    shiftUp(data.size() - 1);
}
/**
```

```
     * 删除堆顶结点
     *
     * @return 堆顶结点
     */
    public T deleteTop() {
        // 如果堆已经为空，就抛出异常
        if (data.isEmpty()) {
            throw new RuntimeException("The heap is empty.");
        }
        // 获取堆顶元素
        T first = data.get(0);
        // 删除最后一个元素
        T last = data.remove(data.size() - 1);
        // 删除元素后，如果堆为空的情况，说明删除的元素也是堆顶元素
        if (data.size() == 0) {
            return last;
        } else {
            // 将删除的元素放入堆顶
            data.set(0, last);
            // 自上向下调整堆
            shiftDown(0);
            // 返回堆顶元素
            return first;
        }
    }
    /**
     * 获取堆顶元素，但不删除
     *
     * @return 堆顶元素
     */
    public T getTop() {
        // 如果堆已经为空，就抛出异常
        if (data.isEmpty()) {
            throw new RuntimeException("The heap is empty.");
        }
        return data.get(0);
    }
    /**
     * 获取堆的大小
     *
     * @return 堆的大小
     */
    public int size() {
        return data.size();
    }
```

```
    /**
     * 判断堆是否为空
     *
     * @return 堆是否为空
     */
    public boolean isEmpty() {
        return data.isEmpty();
    }
    /**
     * 清空堆
     */
    public void clear() {
        data.clear();
    }
    /**
     * 获取堆中所有的数据
     *
     * @return 堆中所在的数据
     */
    public List<T> getData() {
        return data;
    }
}
```

#

升序比较器

```
/**
 * 升序比较器
 */
private static class IncComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
}
```

#

降序比较器

```
/**
 * 降序比较器
 */
private static class DescComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
}
```

#

代码实现

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
public class Test64 {
    private static class Heap<T> {
        // 堆中元素存放的集合
        private List<T> data;
        // 比较器
        private Comparator<T> cmp;
        /**
         * 构造函数
         *
         * @param cmp 比较器对象
         */
        public Heap(Comparator<T> cmp) {
            this.cmp = cmp;
            this.data = new ArrayList<>(64);
        }
        /**
         * 向上调整堆
         *
         * @param idx 被上移元素的起始位置
         */
        public void shiftUp(int idx) {
            // 检查是位置是否正确
            if (idx < 0 || idx >= data.size()) {
                throw new IllegalArgumentException(idx + "");
            }
            // 获取开始调整的元素对象
            T intent = data.get(idx);
            // 如果不是根元素，则需要上移
```

```
        while (idx > 0) {
            // 找父元素对象的位置
            int parentIdx = (idx - 1) / 2;
            // 获取父元素对象
            T parent = data.get(parentIdx);
            //上移的条件，子节点比父节点大，此处定义的大是以比较器返回值为准
            if (cmp.compare(intent, parent) > 0) {
                // 将父节点向下放
                data.set(idx, parent);
                idx = parentIdx;
                // 记录父节点下放的位置
            }
            // 子节点不比父节点大，说明父子路径已经按从大到小排好顺序了，不需要调整了
            else {
                break;
            }
        }
        // index此时记录是的最后一个被下放的父节点的位置（也可能是自身），
        // 所以将最开始的调整的元素值放入index位置即可
        data.set(idx, intent);
    }
    /**
     * 向下调整堆
     *
     * @param idx 被下移的元素的起始位置
     */
    public void shiftDown(int idx) {
        // 检查是位置是否正确
        if (idx < 0 || idx >= data.size()) {
            throw new IllegalArgumentException(idx + "");
        }
        // 获取开始调整的元素对象
        T intent = data.get(idx);
        // 获取开始调整的元素对象的左子结点的元素位置
        int leftIdx = idx * 2 + 1;
        // 如果有左子结点
        while (leftIdx < data.size()) {
            // 取左子结点的元素对象，并且假定其为两个子结点中最大的
            T maxChild = data.get(leftIdx);
            // 两个子节点中最大节点元素的位置，假定开始时为左子结点的位置
            int maxIdx = leftIdx;
            // 获取右子结点的位置
            int rightIdx = leftIdx + 1;
            // 如果有右子结点
            if (rightIdx < data.size()) {
```

```
            T rightChild = data.get(rightIdx);
            // 找出两个子节点中的最大子结点
            if (cmp.compare(rightChild, maxChild) > 0) {
                maxChild = rightChild;
                maxIdx = rightIdx;
            }
        }
        // 如果最大子节点比父节点大，则需要向下调整
        if (cmp.compare(maxChild, intent) > 0) {
            // 将较大的子节点向上移
            data.set(idx, maxChild);
            // 记录上移节点的位置
            idx = maxIdx;
            // 找到上移节点的左子节点的位置
            leftIdx = 2 * idx + 1;
        }
        // 最大子节点不比父节点大，说明父子路径已经按从大到小排好顺序了，不需要调整了
        else {
            break;
        }
    }
    // index此时记录是的最后一个被上移的子节点的位置（也可能是自身），
    // 所以将最开始的调整的元素值放入index位置即可
    data.set(idx, intent);
}
/**
 * 添加一个元素
 *
 * @param item 添加的元素
 */
public void add(T item) {
    // 将元素添加到最后
    data.add(item);
    // 上移，以完成重构
    shiftUp(data.size() - 1);
}
/**
 * 删除堆顶结点
 *
 * @return 堆顶结点
 */
public T deleteTop() {
    // 如果堆已经为空，就抛出异常
    if (data.isEmpty()) {
        throw new RuntimeException("The heap is empty.");
```

```
        }
        // 获取堆顶元素
        T first = data.get(0);
        // 删除最后一个元素
        T last = data.remove(data.size() - 1);
        // 删除元素后，如果堆为空的情况，说明删除的元素也是堆顶元素
        if (data.size() == 0) {
            return last;
        } else {
            // 将删除的元素放入堆顶
            data.set(0, last);
            // 自上向下调整堆
            shiftDown(0);
            // 返回堆顶元素
            return first;
        }
    }
    /**
     * 获取堆顶元素，但不删除
     *
     * @return 堆顶元素
     */
    public T getTop() {
        // 如果堆已经为空，就抛出异常
        if (data.isEmpty()) {
            throw new RuntimeException("The heap is empty.");
        }
        return data.get(0);
    }
    /**
     * 获取堆的大小
     *
     * @return 堆的大小
     */
    public int size() {
        return data.size();
    }
    /**
     * 判断堆是否为空
     *
     * @return 堆是否为空
     */
    public boolean isEmpty() {
        return data.isEmpty();
    }
```

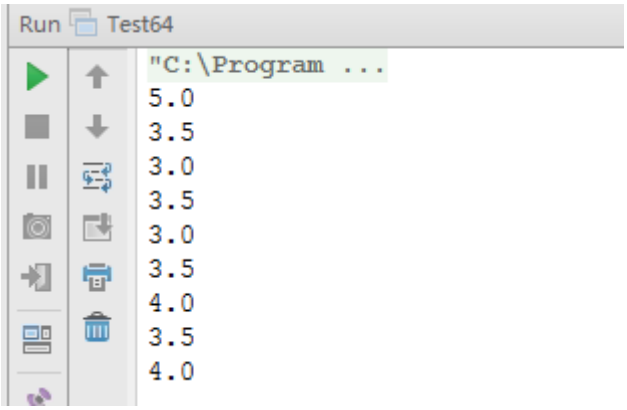
```
    /**
     * 清空堆
     */
    public void clear() {
        data.clear();
    }
    /**
     * 获取堆中所有的数据
     *
     * @return 堆中所在的数据
     */
    public List<T> getData() {
        return data;
    }
}
/**
 * 升序比较器
 */
private static class IncComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
}
/**
 * 降序比较器
 */
private static class DescComparator implements Comparator<Integer> {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
}
private static class DynamicArray {
    private Heap<Integer> max;
    private Heap<Integer> min;
    public DynamicArray() {
        max = new Heap<>(new IncComparator());
        min = new Heap<>(new DescComparator());
    }
    /**
     * 插入数据
     *
     * @param num 待插入的数据
     */
```

```
    public void insert(Integer num) {
        // 已经有偶数个数据了（可能没有数据）
        // 数据总数是偶数个时把新数据插入到小堆中
        if ((min.size() + max.size()) % 2 == 0) {
            // 大堆中有数据，并且插入的元素比大堆中的元素小
            if (max.size() > 0 && num < max.getTop()) {
                // 将num加入的大堆中去
                max.add(num);
                // 删除堆顶元素，大堆中的最大元素
                num = max.deleteTop();
            }
            // num插入到小堆中，当num小于大堆中的最大值进，
            // num就会变成大堆中的最大值，见上面的if操作
            // 如果num不小于大堆中的最大值，num就是自身
            min.add(num);
        }
        // 数据总数是奇数个时把新数据插入到大堆中
        else {
            // 小堆中有数据，并且插入的元素比小堆中的元素大
            if (min.size() > 0 && num > min.size()) {
                // 将num加入的小堆中去
                min.add(num);
                // 删除堆顶元素，小堆中的最小元素
                num = min.deleteTop();
            }
            // num插入到大堆中，当num大于小堆中的最小值进，
            // num就会变成小堆中的最小值，见上面的if操作
            // 如果num不大于大堆中的最小值，num就是自身
            max.add(num);
        }
    }
    public double getMedian() {
        int size = max.size() + min.size();
        if (size == 0) {
            throw new RuntimeException("No numbers are available");
        }
        if ((size & 1) == 1) {
            return min.getTop();
        } else {
            return (max.getTop() + min.getTop()) / 2.0;
        }
    }
}
public static void main(String[] args) {
    DynamicArray array = new DynamicArray();
```

```
        array.insert(5);
        System.out.println(array.getMedian()); // 5
        array.insert(2);
        System.out.println(array.getMedian()); // 3.5
        array.insert(3);
        System.out.println(array.getMedian()); // 3
        array.insert(4);
        System.out.println(array.getMedian()); // 3.5
        array.insert(1);
        System.out.println(array.getMedian()); // 3
        array.insert(6);
        System.out.println(array.getMedian()); // 3.5
        array.insert(7);
        System.out.println(array.getMedian()); // 4
        array.insert(0);
        System.out.println(array.getMedian()); // 3.5
        array.insert(8);
        System.out.println(array.getMedian()); // 4
    }
}
```

#

运行结果



62

滑动窗口的最大值

# #

题目：给定一个数组和滑动窗口的大小，请找出所有滑动窗口里的最大值。

#

举例说明

例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小，那么一共存在 6 个滑动窗口，它们的最大值分别为{4,4,6,6,6,5}。

#

解题思路

如果采用蛮力法，这个问题似乎不难解决：可以扫描每一个滑动窗口的所有数字并找出其中的最大值。如果滑动窗口的大小为 k，需要 O(k)时间才能找出滑动窗口里的最大值。对于长度为 n 的输入数组，这个算法总的时间复杂度是 O(nk)。

实际上一个滑动窗口可以看成是一个队列。当窗口滑动时，处于窗口的第一个数字被删除，同时在窗口的末尾添加一个新的数字。这符合队列的先进先出特性。如果能从队列中找出它的最大数，这个问题也就解决了。

在面试题 21 中。我们实现了一个可以用 O(1)时间得到最小值的栈。同样，也可以用 O(1)时间得到栈的最大值。同时在面试题 7 中，我们讨论了如何用两个栈实现一个队列。综合这两个问题的解决方法，我们发现如果把队列用两个栈实现，由于可以用 O(1)时间得到栈中的最大值，那么也就可以用 O(1)时间得到队列的最大值，因此总的时间复杂度也就降到了 O(n)。

我们可以用这个方法来解决问题。不过这样就相当于在一轮面试的时间内要做两个面试题，时间未必够用。再来看看有没有其它的方法。

下面换一种思路。我们并不把滑动窗口的每个数值都存入队列中，而只把有可能成为滑动窗口最大值的数值存入到一个两端开口的队列。接着以输入数字{2,3,4,2,6,2,5,1}为例一步分析。

数组的第一个数字是 2，把它存入队列中。第二个数字是3.由于它比前一个数字 2 大，因此 2不可能成为滑动窗口中的最大值。2 先从队列里删除，再把3存入到队列中。此时队列中只有一个数字 3。针对第三个数字 4 的步骤类似，最终在队列中只剩下一个数字 4。此时滑动窗口中已经有 3 个数字，而它的最大值 4 位于队列的头部。

接下来处理第四个数字 2。2 比队列中的数字 4 小。当 4 滑出窗口之后 2 还是有可能成为滑动窗口的最大值，因此把 2 存入队列的尾部。现在队列中有两个数字 4 和 2，其中最大值 4 仍然位于队列的头部。

下一个数字是 6。由于它比队列中已有的数字 4 和 2 都大，因此这时 4 和 2 已经不可能成为滑动窗口中的最大值。先把 4 和 2 从队列中删除，再把数字 6 存入队列。这个时候最大值 6 仍然位于队列的头部。

第六个数字是 2。由于它比队列中已有的数字 6 小，所以 2 也存入队列的尾部。此时队列中有两个数字，其中最大值 6 位于队列的头部。

接下来的数字是 5。在队列中已有的两个数字 6 和 2 里，2 小于 5，因此 2 不可能是一个滑动窗口的最大值，可以把它从队列的尾部删除。删除数字 2 之后，再把数字 5 存入队列。此时队列里剩下两个数字 6 和 5，其中位于队列头部的是最大值 6。

数组最后一个数字是 1，把 1 存入队列的尾部。注意到位于队列头部的数字 6 是数组的第 5 个数字，此时的滑动窗口已经不包括这个数字了，因此应该把数字 6 从队列删除。那么怎么知道滑动窗口是否包括一个数字？应该在队列里存入数字在数组里的下标，而不是数值。当一个数字的下标与当前处理的数字的下标之差大于或者等于滑动窗口的大小时，这个数字已经从滑动窗口中滑出，可以从队列中删除了。

#

代码实现

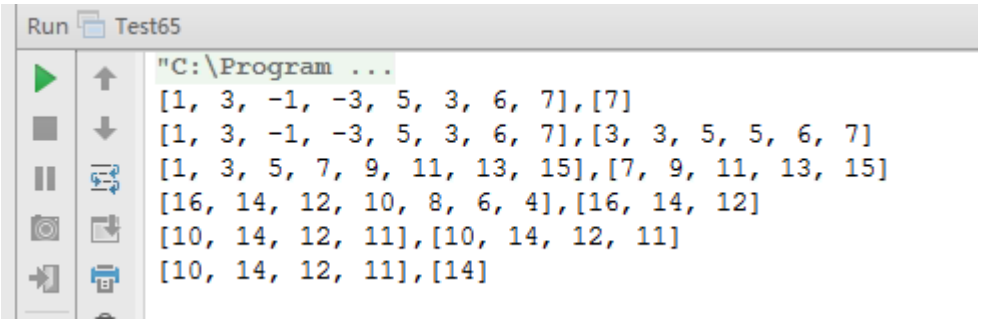
```
import java.util.*;
public class Test65 {
    private static List<Integer> maxInWindows(List<Integer> data, int size) {
        List<Integer> windowMax = new LinkedList<>();
        // 条件检查
        if (data == null || size < 1 || data.size() < 1) {
            return windowMax;
        }
        Deque<Integer> idx = new LinkedList<>();
        // 窗口还没有被填满时，找最大值的索引
        for (int i = 0; i < size && i < data.size(); i++) {
            // 如果索引对应的值比之前存储的索引值对应的值大或者相等，就删除之前存储的值
            while (!idx.isEmpty() && data.get(i) >= data.get(idx.getLast())) {
                idx.removeLast();
            }
            // 添加索引
            idx.addLast(i);
        }
        // 窗口已经被填满了
```

```
    for (int i = size; i < data.size(); i++) {
        // 第一个窗口的最大值保存
        windowMax.add(data.get(idx.getFirst()));
        // 如果索引对应的值比之前存储的索引值对应的值大或者相等，就删除之前存储的值
        while (!idx.isEmpty() && data.get(i) >= data.get(idx.getLast())) {
            idx.removeLast();
        }
        // 删除已经滑出窗口的数据对应的下标
        if (!idx.isEmpty() && idx.getFirst() <= (i - size)) {
            idx.removeFirst();
        }
        // 可能的最大的下标索引入队
        idx.addLast(i);
    }
    // 最后一个窗口最大值入队
    windowMax.add(data.get(idx.getFirst()));
    return windowMax;
}
private static List<Integer> arrayToCollection(int[] array) {
    List<Integer> result = new LinkedList<>();
    if (array != null) {
        for (int i : array) {
            result.add(i);
        }
    }
    return result;
}
public static void main(String[] args) {
    // expected {7};
    List<Integer> data1 = arrayToCollection(new int[]{1, 3, -1, -3, 5, 3, 6, 7});
    System.out.println(data1 + "," + maxInWindows(data1, 10));
    // expected {3, 3, 5, 5, 6, 7};
    List<Integer> data2 = arrayToCollection(new int[]{1, 3, -1, -3, 5, 3, 6, 7});
    System.out.println(data2 + "," + maxInWindows(data2, 3));
    // expected {7, 9, 11, 13, 15};
    List<Integer> data3 = arrayToCollection(new int[]{1, 3, 5, 7, 9, 11, 13, 15});
    System.out.println(data3 + "," + maxInWindows(data3, 4));
    // expected  {16, 14, 12};
    List<Integer> data5 = arrayToCollection(new int[]{16, 14, 12, 10, 8, 6, 4});
    System.out.println(data5 + "," + maxInWindows(data5, 5));
    // expected  {10, 14, 12, 11};
    List<Integer> data6 = arrayToCollection(new int[]{10, 14, 12, 11});
    System.out.println(data6 + "," + maxInWindows(data6, 1));
    // expected  {14};
    List<Integer> data7 = arrayToCollection(new int[]{10, 14, 12, 11});
```

```
        System.out.println(data7 + "," + maxInWindows(data7, 4));
    }
}
```

#

运行结果



63

矩阵中的路径

题目：请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中任意一格开始，每一步可以在矩阵中间向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。

#

举例分析

例如在下面的 3*4 的矩阵中包含一条字符串“bcced”的路径。但矩阵中不包含字符串“abcb”的路径，因为字符串的第一个字符 b 占据了矩阵中的第一行第二格子之后，路径不能再次进入这个格子。

```
a b c e
s f c s
a d e e
```

#

解题思路

这是一个可以用回朔法解决的典型题。首先，在矩阵中任选一个格子作为路径的起点。假设矩阵中某个格子的字符为 ch，那么这个格子不可能处在路径上的第 i 个位置。如果路径上的第 i 个字符不是 ch，那么这个格子不可能处在路径上的第 i 个位置。如果路径上的第 i 个字符正好是 ch，那么往相邻的格子寻找路径上的第 i+1 个字符。除在矩阵边界上的格子之外，其他格子都有 4 个相邻的格子。重复这个过程知道路径上的所有字符都在矩阵中找到相应的位置。

由于回朔法的递归特性，路径可以被开成一个栈。当在矩阵中定位了路径中前 n 个字符的位置之后，在与第 n 个字符对应的格子的周围都没有找到第 n+1 个字符，这个时候只要在路径上回到第 n-1 个字符，重新定位第 n 个字符。

由于路径不能重复进入矩阵的格子，还需要定义和字符矩阵大小一样的布尔值矩阵，用来标识路径是否已经进入每个格子。

当矩阵中坐标为（row,col）的格子和路径字符串中下标为 pathLength 的字符一样时，从 4 个相邻的格子 (row,col-1),(row-1,col),(row,col+1) 以及 (row+1,col) 中去定位路径字符串中下标为 pathLength+1 的字符。

如果 4 个相邻的格子都没有匹配字符串中下标为 pathLength+1 的字符，表明当前路径字符串中下标为pathLength的字符在矩阵中的定位不正确，我们需要回到前一个字符 (pathLength-1)，然后重新定位。

一直重复这个过程，直到路径字符串上所有字符都在矩阵中找到合适的位置

#

代码实现

```
public class Test66 {
    /**
     * 题目：请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。
     * 路径可以从矩阵中任意一格开始，每一步可以在矩阵中间向左、右、上、下移动一格。
     * 如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。
     *
     * @param matrix 输入矩阵
     * @param rows   矩阵行数
     * @param cols   矩阵列数
     * @param str    要搜索的字符串
     * @return 是否找到 true是，false否
     */
    public static boolean hasPath(char[] matrix, int rows, int cols, char[] str) {
        // 参数校验
        if (matrix == null || matrix.length != rows * cols || str == null || str.length < 1) {
            return false;
        }

        // 变量初始化
        boolean[] visited = new boolean[rows * cols];
        for (int i = 0; i < visited.length; i++) {
            visited[i] = false;
        }

        // 记录结果的数组，
        int[] pathLength = {0};
        // 以每一个点为起始进行搜索
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (hasPathCore(matrix, rows, cols, str, visited, i, j, pathLength)) {
                    return true;
                }
            }
        }

        return false;
    }

    /**
```

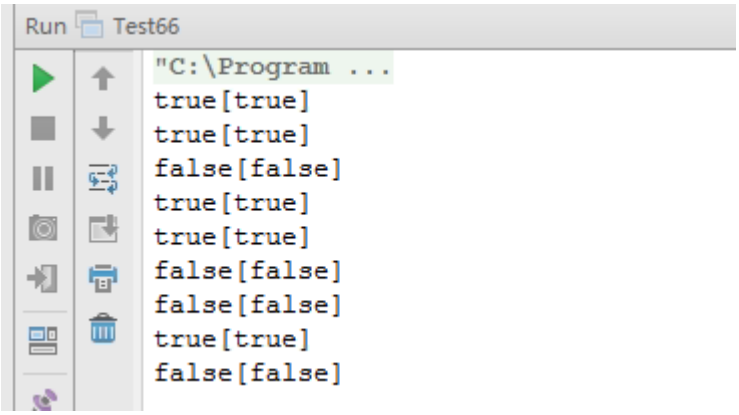
```
     * 回溯搜索算法
     *
     * @param matrix     输入矩阵
     * @param rows       矩阵行数
     * @param cols       矩阵列数
     * @param str        要搜索的字符串
     * @param visited    访问标记数组
     * @param row        当前处理的行号
     * @param col        当前处理的列号
     * @param pathLength 已经处理的str中字符个数
     * @return 是否找到 true是，false否
     */
    private static boolean hasPathCore(char[] matrix, int rows, int cols, char[] str, boolean[] visited,
                                       int row, int col, int[] pathLength) {

        if (pathLength[0] == str.length) {
            return true;
        }

        boolean hasPath = false;

        // 判断位置是否合法
        if (row >= 0 && row < rows
                && col >= 0 && col < cols
                && matrix[row * cols + col] == str[pathLength[0]]
                && !visited[row * cols + col]) {

            visited[row * cols + col] = true;
            pathLength[0]++;

            // 按左上右下进行回溯
            hasPath = hasPathCore(matrix, rows, cols, str, visited, row, col - 1, pathLength)
                    || hasPathCore(matrix, rows, cols, str, visited, row - 1, col, pathLength)
                    || hasPathCore(matrix, rows, cols, str, visited, row, col + 1, pathLength)
                    || hasPathCore(matrix, rows, cols, str, visited, row + 1, col, pathLength);

            if (!hasPath) {
                pathLength[0]--;
                visited[row * cols + col] = false;
            }

        }

        return hasPath;
    }
```

```
public static void main(String[] args) {
    //ABCE  //ABCCED
    //SFCS
    //ADEE
    System.out.println(hasPath("ABCESFCSADEE".toCharArray(), 3, 4,
            "ABCCED".toCharArray()) + "[true]");// true

    //ABCE  //SEE
    //SFCS
    //ADEE
    System.out.println(hasPath("ABCESFCSADEE".toCharArray(), 3, 4,
            "SEE".toCharArray()) + "[true]");// true

    //ABCE  //ABCB
    //SFCS
    //ADEE
    System.out.println(hasPath("ABCESFCSADEE".toCharArray(), 3, 4,
            "ABCB".toCharArray()) + "[false]");// false

    //ABCEHJIG  //SLHECCEIDEJFGGFIE
    //SFCSLOPQ
    //ADEEMNOE
    //ADIDEJFM
    //VCEIFGGS
    System.out.println(hasPath("ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS".toCharArray(), 5, 8,
            "SLHECCEIDEJFGGFIE".toCharArray()) + "[true]");// true


    //ABCEHJIG  //SGGFIECVAASABCEHJIGQEM
    //SFCSLOPQ  //
    //ADEEMNOE
    //ADIDEJFM
    //VCEIFGGS
    System.out.println(hasPath("ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS".toCharArray(), 5, 8,
            "SGGFIECVAASABCEHJIGQEM".toCharArray()) + "[true]");// true

    //ABCEHJIG  //SGGFIECVAASABCEEJIGOEM
    //SFCSLOPQ
    //ADEEMNOE
    //ADIDEJFM
    //VCEIFGGS
    System.out.println(hasPath("ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS".toCharArray(), 5, 8,
            "SGGFIECVAASABCEEJIGOEM".toCharArray()) + "[false]");// false
```

```
        //ABCEHJIG  //SGGFIECVAASABCEHJIGQEMS
        //SFCSLOPQ
        //ADEEMNOE
        //ADIDEJFM
        //VCEIFGGS
        System.out.println(hasPath("ABCEHJIGSFCSLOPQADEEMNOEADIDEJFMVCEIFGGS".toCharArray(), 5, 8,
                "SGGFIECVAASABCEHJIGQEMS".toCharArray()) + "[false]");// false

        //AAAA  //AAAAAAAAAAAA
        //AAAA
        //AAAA
        System.out.println(hasPath("AAAAAAAAAAAA".toCharArray(), 3, 4,
                "AAAAAAAAAAAA".toCharArray()) + "[true]");// true

        //AAAA  //AAAAAAAAAAAA
        //AAAA
        //AAAA
        System.out.println(hasPath("AAAAAAAAAAAA".toCharArray(), 3, 4,
                "AAAAAAAAAAAAA".toCharArray()) + "[false]");// false

    }

}
```

#

运行结果


Run
Test66
"C:\Program ...
true[true]
true[true]
false[false]
true[true]
true[true]
false[false]
false[false]
true[true]
false[false]

64

机器人的运动范围

题目：地上有个 m 行 n 列的方格。一个机器人从坐标(0,0)的格子开始移动，它每一次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和大于 k 的格子。

#

举例分析

例如，当 k 为 18 时，机器人能够进入方格(35,37)，因为 3+5+3+7=18 但它不能进入方格(35,38)，因为 3+5+3+8=19 请问该机器人能够达到多少格子?

解题思路

和前面的矩阵中的路径类似，这个方格也可以看出一个 m*n 的矩阵。同样在这个矩阵中，除边界上的格子之外其他格子都有四个相邻的格子。

机器人从坐标(0,0)开始移动。当它准备进入坐标为(i,j)的格子时，通过检查坐标的数位和来判断机器人是否能够进入。如果机器人能够进入坐标为(i,j)的格子，我们接着再判断它能否进入四个相邻的格子(i,j−1)、(i−1,j),(i,j+1) 和(i+1,j)。

# #

代码实现

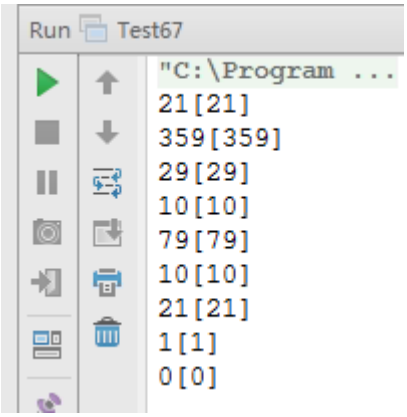
```
public class Test67 {
    /**
     * 题目：地上有个m行n列的方格。一个机器人从坐标(0,0)的格子开始移动，
     * 它每一次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数
     * 位之和大于k的格子。例如，当k为18时，机器人能够进入方格(35,37)，
     * 因为3+5+3+7=18.但它不能进入方格(35,38)，因为3+5+3+8=19.
     * 请问该机器人能够达到多少格子?
     *
     * @param threshold 约束值
     * @param rows      方格的行数
     * @param cols      方格的列数
     * @return 最多可走的方格
     */
    public static int movingCount(int threshold, int rows, int cols) {
        // 参数校验
        if (threshold < 0 || rows < 1 || cols < 1) {
            return 0;
        }

        // 变量初始化
        boolean[] visited = new boolean[rows * cols];
        for (int i = 0; i < visited.length; i++) {
            visited[i] = false;
        }

        return movingCountCore(threshold, rows, cols, 0, 0, visited);
    }

    /**
     * 递归回溯方法
     *
     * @param threshold 约束值
     * @param rows      方格的行数
     * @param cols      方格的列数
     * @param row       当前处理的行号
     * @param col       当前处理的列号
     * @param visited   访问标记数组
     * @return 最多可走的方格
     */
```

```
private static int movingCountCore(int threshold, int rows, int cols,
                                   int row, int col, boolean[] visited) {

    int count = 0;

    if (check(threshold, rows, cols, row, col, visited)) {
        visited[row * cols + col] = true;
        count = 1
                + movingCountCore(threshold, rows, cols, row − 1, col, visited)
                + movingCountCore(threshold, rows, cols, row, col − 1, visited)
                + movingCountCore(threshold, rows, cols, row + 1, col, visited)
                + movingCountCore(threshold, rows, cols, row, col + 1, visited);
    }

    return count;
}

/**
 * 断机器人能否进入坐标为(row, col)的方格
 *
 * @param threshold 约束值
 * @param rows      方格的行数
 * @param cols      方格的列数
 * @param row       当前处理的行号
 * @param col       当前处理的列号
 * @param visited   访问标记数组
 * @return 是否可以进入，true是，false否
 */
private static boolean check(int threshold, int rows, int cols,
                             int row, int col, boolean[] visited) {
    return col >= 0 && col < cols
            && row >= 0 && row < rows
            && !visited[row * cols + col]
            && (getDigitSum(col) + getDigitSum(row) <= threshold);
}

/**
 * 一个数字的数位之和
 *
 * @param number 数字
 * @return 数字的数位之和
 */
private static int getDigitSum(int number) {
    int result = 0;
    while (number > 0) {
```

```
            result += (number % 10);
            number /= 10;
        }

        return result;
    }

    public static void main(String[] args) {
        System.out.println(movingCount(5, 10, 10) + "[21]");
        System.out.println(movingCount(15, 20, 20) + "[359]");
        System.out.println(movingCount(10, 1, 100) + "[29]");
        System.out.println(movingCount(10, 1, 10) + "[10]");
        System.out.println(movingCount(15, 100, 1) + "[79]");
        System.out.println(movingCount(15, 10, 1) + "[10]");
        System.out.println(movingCount(5, 10, 10) + "[21]");
        System.out.println(movingCount(12, 1, 1) + "[1]");
        System.out.println(movingCount(-10, 10, 10) + "[0]");
    }
}
```

#

运行结果


Run
Test67
"C:\Program ...
21[21]
359[359]
29[29]
10[10]
79[79]
10[10]
21[21]
1[1]
0[0]