**剑指Offer算法题整理**

**二维数组：**

**在一个二维数组中（每个一维数组的长度相同），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。**

*/\*\*  
 \** ***@return*** *boolean  
 \** ***@Author*** *Liruilong  
 \** ***@Description*** *方法一，自己想的笨办法,对角线遍历。缩小区域后，暴力寻找。  
 \** ***@Description 方法二，利用二维数组由上到下，由左到右递增的规律，那么选取右上角或者左下角的元素a[row][col]与target进行比较，  
 \* 当target小于元素a[row][col]时，那么target必定在元素a所在行的左边,即col--；当target大于元素a[row][col]时，  
 \* 那么target必定在元素a所在列的下边,即row++；*** *\*\*/***public static boolean Find1(int target, int[][] array) {  
 int row = 0;  
 int col = array[0].length - 1;  
 while (row <= array.length - 1 && col >= 0) {  
 if (target == array[row][col])  
 return true;  
 else if (target > array[row][col])  
 row++;  
 else  
 col--;  
 }  
 return false;  
}**

**替换空格：**

**请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。**

**public static String replaceSpace(StringBuffer str) {  
 String string = str.toString();  
 *// 字符序列替换  
 //string = string.replace(" ","%20");  
 // 正则表达式换* string = string.replaceAll(" ", "%20");  
 return string;  
}**

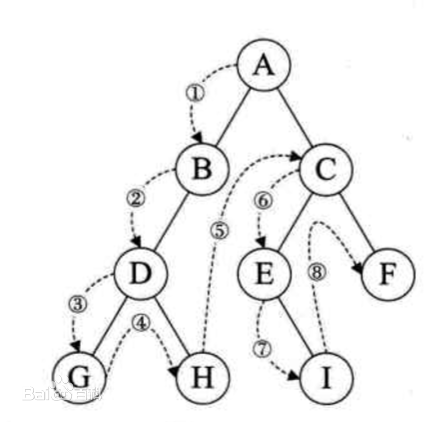
**从尾到头打印列表**

**输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。**

**public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {  
 ListNode temp = listNode;  
 ArrayList list = new ArrayList();  
 ArrayList lists = new ArrayList();  
 if(temp == null){  
 return list;  
 }  
 while(temp != null){  
 list.add(temp.val);  
 temp = temp.next;  
 }  
 for(int i = list.size() -1; i >=0; i--){  
 lists.add(list.get(i));  
 }  
 return lists;  
}**

**重建二叉树**

**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。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回**



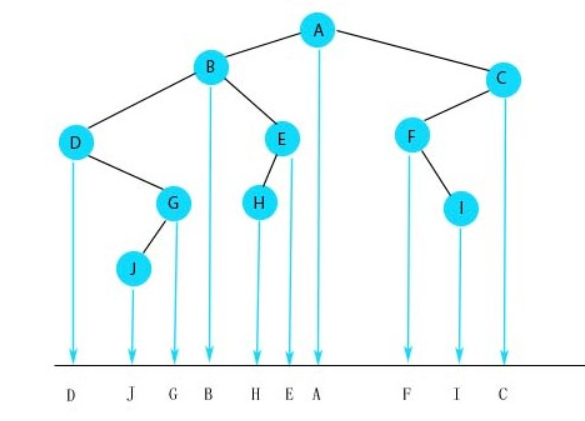
**1．先（根）序遍历的递归算法定义：**

**若二叉树非空，则依次执行如下操作：**

**⑴ 访问根结点；**

**⑵ 遍历左子树；**

**⑶ 遍历右子树。**



**2．中（根）序遍历的递归算法定义：**

**若二叉树非空，则依次执行如下操作：**

**⑴遍历左子树；**

**⑵访问根结点；**

**⑶遍历右子树。**

**二叉树学习: https://segmentfault.com/a/1190000014743964**

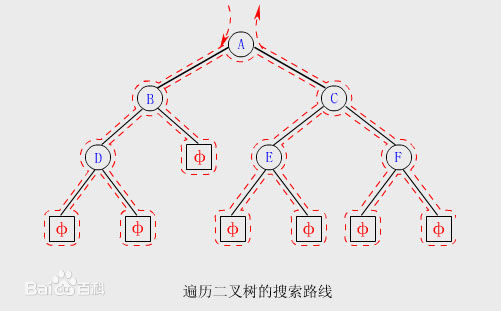
**3．后（根）序遍历得递归算法定义：**

**若二叉树非空，则依次执行如下操作：**

**⑴遍历左子树；**

**⑵遍历右子树；**

**⑶访问根结点。**



**算法思路；先确定根节点，由根节点确定左右子树，依次递归实现。**

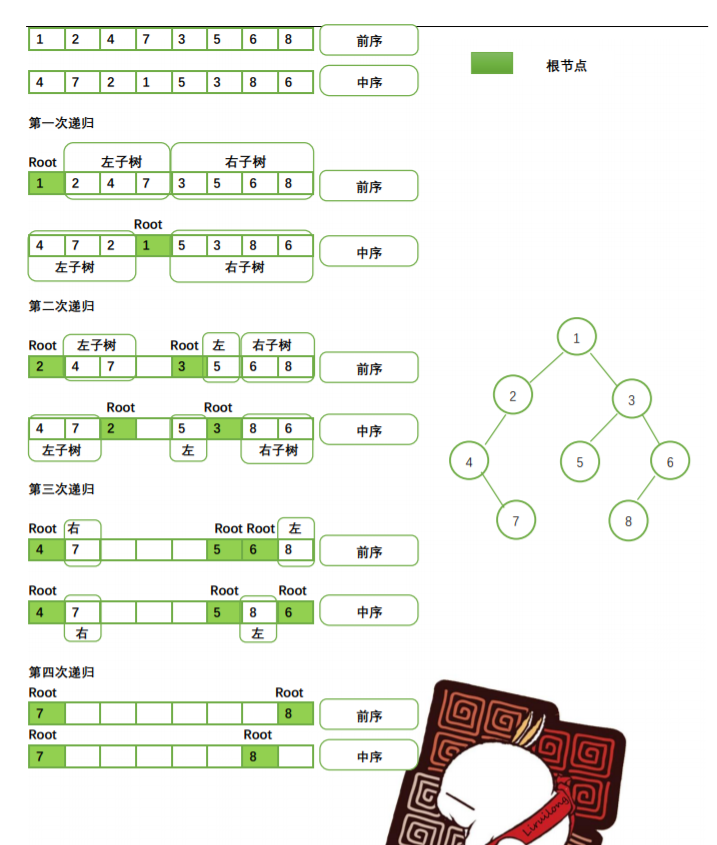
**依据：前序遍历的每个节点都可以看做是后续节点的根节点。中序遍历的根节点将整个二叉树左右子树分开。**

**public TreeNode reConstructBinaryTree(int [] pre,int [] in) {  
 if(pre == null || in == null){  
 return null;  
 }  
 TreeNode root = reConstructBinaryTree(pre,0,pre.length-1,in,0,in.length-1);  
 return root;  
}**

**private TreeNode reConstructBinaryTree(  
 int[] pre,int startPre,int endPre,int[] in,int startIn,int endIn){  
 if(startPre>endPre||startIn>endIn){  
 return null;  
 }  
 *//前序遍历的第一个数字是根节点的值* TreeNode root = new TreeNode(pre[startPre]);  
 *//遍历中序数组，找到根节点的位置,创建左右子树* for(int i = startIn;i <=endIn; i++){  
 if(in[i]==pre[startPre]){  
 root.left =**

**reConstructBinaryTree(pre,startPre+1,startPre+i-startIn,in,startIn,i-1);  
 root.right =**

**reConstructBinaryTree(pre,startPre+i+1-startIn,endPre,in,i+1,endIn);  
 }  
 }  
 return root;  
}**



**用两个栈实现队列**

**用两个栈来实现一个队列，完成队列的Push和Pop操作。 队列中的元素为int类型。**

**public static class Solutions {  
 Stack<Integer> stack1 = new Stack<Integer>();  
 Stack<Integer> stack2 = new Stack<Integer>();  
  
 public void push(int node) {  
 stack1.push(node);  
 }  
  
 public int pop() {  
 if(stack1.empty()&&stack2.empty()){  
 throw new RuntimeException("Queue is empty!");  
 }  
 if(stack2.empty()){  
 while(!stack1.empty()){  
 stack2.push(stack1.pop());  
 }  
 }  
 return stack2.pop();  
 }  
}**

**旋转数组的最小数字**

**把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。 输入一个非减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{1,2,3,4,5}的一个旋转，该数组的最小值为1。 NOTE：给出的所有元素都大于0，若数组大小为0，请返回0。**

**public int minNumberInRotateArray(int [] array) {  
 if(array.length == 0){  
 return 0;  
 }  
 for(int i = 0; i < array.length; i++){  
 if( array[i] > array[i+1]){  
 return array[i+1];  
 }  
 }  
 return 0;  
}**

**斐波那契数列**

**大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。**

**n<=39 嘻嘻，三种方法。**

**public static long fibonacci1(int n){  
 long a = 1L, b = 1L, c=0L;  
 if (n < 1){  
 return 0;  
 }  
 if (n > 2){  
 for (int i=3; i <= n; i++){  
 c = a + b;  
 a = b;  
 b = c;  
 }  
 }  
 if (n ==1 || n==2){  
 return 1;  
 }  
 return c;  
}**

**//递归方式**

**public static long fibonacci0(int n){  
 if (n == 1 || n == 2)  
 return 1;  
 if (n > 2)  
 return *fibonacci0*(n-1) + *fibonacci0*(n-2);  
 return 0;  
}**

**// 数组方式**

**public static long fibonacci2(int n){  
 long a = 1L, b = 1L, c=0L;  
 long[] array = new long[n];  
 array[0] = a;  
 array[1] = b;  
 if (n < 1)  
 return 0;  
 if (n == 1 || n == 2)  
 return 1;  
 if (n > 2)  
 for (int i = 2; i < array.length; i++)  
 array[i] = array[i-1] + array[i-2];  
 return array[n-1];  
}**

**跳台阶**

**一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。**

**对于本题,前提只有 一次 1阶或者2阶的跳法。**

**a.如果两种跳法，1阶或者2阶，那么假定第一次跳的是一阶，那么剩下的是n-1个台阶，跳法是f(n-1);**

**b.假定第一次跳的是2阶，那么剩下的是n-2个台阶，跳法是f(n-2)**

**c.由a\b假设可以得出总跳法为: f(n) = f(n-1) + f(n-2)**

**d.然后通过实际的情况可以得出：只有一阶的时候 f(1) = 1 ,只有两阶的时候可以有 f(2) = 2**

**e.可以发现最终得出的是一个斐波那契数列：**

**| 1, (n=1)**

**f(n) =     | 2, (n=2)**

**| f(n-1)+f(n-2) ,(n>2,n为整数)**

**public int JumpFloor(int target) {  
 if(target ==1){  
 return 1;  
 }else if(target == 2){  
 return 2;  
 }else{  
 return JumpFloor(target -1) + JumpFloor(target -2);  
 }  
}**

**矩形覆盖**

**我们可以用2\*1的小矩形横着或者竖着去覆盖更大的矩形。请问用n个2\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？**

**public int** RectCover(**int** target) {  
 **if**(target == 1 ||target == 2){  
 **return** target;  
 }**else if**(target == 0){  
 **return** 0;  
 }**else**{  
 **return** RectCover(target -1) + RectCover(target -2);  
 }  
}

**依旧是斐波那契数列**

**变态跳台阶**

**一只青蛙一次可以跳上1级台阶，也可以跳上2级……它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法**

**贪心算法，获取所有的可能，：**

**因为n级台阶，第一步有n种跳法：跳1级、跳2级、到跳n级**

**跳1级，剩下n-1级，则剩下跳法是f(n-1)**

**跳2级，剩下n-2级，则剩下跳法是f(n-2)**

**所以f(n)=f(n-1)+f(n-2)+...+f(1)**

**因为f(n-1)=f(n-2)+f(n-3)+...+f(1)**

**所以f(n)=2\*f(n-1)**

**public int JumpFloorII(int target) {  
 if(target == 1){  
 return 1;  
 }else if(target == 2){  
 return 2;  
 }else{  
 return JumpFloorII(target - 1) << 1;  
 }  
}**

**二进制数中1的个数**

**输入一个整数，输出该数二进制表示中1的个数。其中负数用补码表示。**

**思路：在二进制整数中，一个数减一可使二进制数最后一位1变成零，之后的0都变成1，利用这个特性，原二进制数与减1的二进制数做与运算，得到的是去掉最有一位1的二进制数，依次循环，可以知道有几个1；**

**方法一：位运算 方法二：Integer方法**

**public int NumberOf1s(int n) {  
 int t=0;  
 *// toBinaryString(n) 以二进制（基数 2）无符号整数形式返回一个整数参数的字符串表示形式。* char[]ch=Integer.*toBinaryString*(n).toCharArray();  
 for(int i=0;i<ch.length;i++){  
 if(ch[i]=='1'){  
 t++;  
 }  
 }  
 return t;  
}**

**public int NumberOf1(int n) {  
 int count = 0;  
 while(n!= 0){  
 count++;  
 n = n & (n - 1);  
 }  
 return count;  
}**

**public double Power(double base, int exponent) {  
 double result=1;  
 for(int i=0;i<Math.*abs*(exponent);i++){  
 result\*=base;  
 }  
 if(exponent<0){  
 result=1/result;  
 }  
 return result;  
}**

**数值的整数次方**

**给定一个double类型的浮点数**

**base和int类型的整数exponent。**

**求base的exponent次方。**

**public double Power(double base, int exponent) {  
 double result=1;  
 result = Math.*pow*(base, exponent);  
 return result;  
}**

**调整数组顺序使奇数位于偶数前**

**public void reOrderArrays(int [] array) {  
 if (array != null) {  
 int[] even = new int[array.length];  
 int indexOdd = 0;  
 int indexEven = 0;  
 for (int num : array) {  
 if ((num & 1) == 1) {  
 array[indexOdd++] = num;  
 } else {  
 even[indexEven++] = num;  
 }  
 }  
  
 for (int i = 0; i < indexEven; i++) {  
 array[indexOdd + i] = even[i];  
 }  
 }  
}**

**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，使得所有的奇数位于**

**数组的前半部分，所有的偶数位于数组的**

**后半部分，并保证奇数和奇数，偶数和**

**偶数之间的相对位置不变。**

**新开数组空间换时间的解法,**

**a.遍历数组,如果是奇数从头部放入到原数组中,并记录指针**

**b.如果是偶数,放入到新数组中,并记录指针**

**c.将新数组的元素安顺序,从最后一个奇数后边插入到原数组中**

**链表中倒数第k个节点**

**public ListNode FindKthToTail(ListNode head,int k) {  
 ListNode pre=null,p=null;  
 *//两个指针都指向头结点* p=head;  
 pre=head;  
 *//记录k值* int a=k;  
 *//记录节点的个数* int count=0;  
 *//p指针先跑，并且记录节点数，当p指针跑了k-1个节点后，pre指针开始跑，  
 //当p指针跑到最后时，pre所指指针就是倒数第k个节点* while(p!=null){  
 p=p.next;  
 count++;  
 if(k<1){  
 pre=pre.next;  
 }  
 k--;  
 }  
 *//如果节点个数小于所求的倒数第k个节点，则返回空* if(count<a) return null;  
 return pre;  
}**

**输入一个链表，输出该链表中倒数第k个结点。**

**// 笨方法**

**import java.util.ArrayList;  
public class Solution {  
 public ListNode FindKthToTail(ListNode head,int k) {  
 if(head == null){  
 return null;  
 }  
 ListNode root = head;  
 ArrayList list =new ArrayList();  
 while(root != null){  
 list.add(root);  
 root = root.next;  
 }  
 return k > list.size() || k <= 0 ? null:(ListNode) list.get(list.size()-k);  
 }  
}**

**好办法：**

**相当于制造了一个K长度的尺子，把尺子从头往后移动，**

**当尺子的右端与链表的末尾对齐的时候，**

**尺子左端所在的结点就是倒数第k个结点！**

**递归方法：**

**public static ListNode Merge(ListNode list1,ListNode list2) {  
 if (list1 == null){  
 return list2;  
 }  
 if (list2 == null){  
 return list1;  
 }  
 if (list1.val >= list2.val){  
 list2.next = *Merge*(list1, list2.next);  
 return list2;  
 }else {  
 list1.next = *Merge*(list1.next, list2);  
 return list1;  
 }  
}**

**合并两个排序的列表**

**输入两个单调递增的链表，**

**输出两个链表合成后的链表，**

**当然我们需要合成后的链表**

**满足单调不减规则。**

**普通方法：**

**public ListNode Merges(ListNode list1,ListNode list2) {  
 ListNode newHead = new ListNode(-1);  
 ListNode current = newHead;  
 while (list1 != null && list2 != null) {  
 if (list1.val < list2.val) {  
 current.next = list1;  
 list1 = list1.next;  
 } else {  
 current.next = list2;  
 list2 = list2.next;  
 }  
 current = current.next;  
 }  
 if (list1 != null) current.next = list1;  
 if (list2 != null) current.next = list2;  
 return newHead.next;  
}**

**树的子结构**

**输入两棵二叉树A，B，判断B是不是A的子结构。**

**（ps：我们约定空树不是任意一个树的子结构）**

**思路：先判断A树和B树根节点是否相同，相同，遍历左右节点，不相同，A树的左右节点做根节点与B树点比较。依次递归。当B树遍历完之后，返回true。当A遍历完B树未遍历完，返回false。当A节点与B值不等时，返回false。**

**public boolean HasSubtree(TreeNode root1,TreeNode root2) {  
 boolean result = false;  
 *//当Tree1和Tree2都不为零的时候，才进行比较。否则直接返回false* if (root2 != null && root1 != null) {  
 *//如果找到了对应Tree2的根节点的点* if(root1.val == root2.val){  
 *//以这个根节点为为起点判断是否包含Tree2* result = *doesTree1HaveTree2*(root1,root2);  
 }  
 *//如果找不到，那么就再去root的左儿子当作起点，去判断时候包含Tree2* if (!result) {  
 result = HasSubtree(root1.left,root2);  
 }  
 *//如果还找不到，那么就再去root的右儿子当作起点，去判断时候包含Tree2* if (!result) {  
 result = HasSubtree(root1.right,root2);  
 }  
 }  
 *//返回结果* return result;  
}  
public static boolean doesTree1HaveTree2(TreeNode node1, TreeNode node2) {  
 *//如果Tree2已经遍历完了都能对应的上，返回true* if (node2 == null) {  
 return true;  
 }  
 *//如果Tree2还没有遍历完，Tree1却遍历完了。返回false* if (node1 == null) {  
 return false;  
 }  
 *//如果其中有一个点没有对应上，返回false* if (node1.val != node2.val) {  
 return false;  
 }  
 *//如果根节点对应的上，那么就分别去子节点里面匹配* return *doesTree1HaveTree2*(node1.left,node2.left)   
 && *doesTree1HaveTree2*(node1.right,node2.right);  
}**

**public void Mirror(TreeNode root) {  
 if(root == null) {  
 return ;  
 }  
 TreeNode temp = root.left;  
 root.left = root.right;  
 root.right = temp;  
 Mirror(root.left);  
 Mirror(root.right);  
}**

**二叉树的镜像**

**操作给定的二叉树，**

**将其变换为源二叉树的镜像。**

**顺时针打印矩阵**

**输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下4 X 4矩阵： 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 则依次打印出数字1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10.**

**public ArrayList<Integer> printMatrix(int [][] matrix) {  
 ArrayList<Integer> result=new ArrayList<Integer>(); *//将顺时针输出的矩阵元素都存放在结果列表里* if(matrix.length==0) return result; *//如果矩阵行数为0，说明是空矩阵，则直接返回空列表* int row=matrix.length,column=matrix[0].length; *//初始化矩阵行数和列数* if(column==0) return result; int layers=(Math.*min*(row,column)-1)/2+1;  
 for(int i=0;i<layers;i++){  
 for(int j=i;j<column-i;j++) result.add(matrix[i][j]);  
 for(int k=i+1;k<row-i;k++) result.add(matrix[k][column-1-i]);  
 for(int j=column-2-i;(j>=i)&&(row-1-i!=i);j--) result.add(matrix[row-1-i][j]);  
 for(int k=row-2-i;(k>i)&&(column-1-i!=i);k--) result.add(matrix[k][i]);  
 }  
 return result;  
}**

**public ArrayList<Integer> printMatrix\_2(int[][] matrix) {  
 ArrayList<Integer> al = new ArrayList<>();  
 int row = matrix.length;  
 while (row != 0) {  
 for (int i = 0; i < matrix[0].length; i++)   
 al.add(matrix[0][i]);  
 if (row == 1)  
 break;  
 matrix = turn(matrix);  
 row = matrix.length;  
 }  
 return al;  
}**

**private int[][] turn(int[][] matrix) {**

**int col = matrix[0].length;  
 int row = matrix.length;  
 int[][] newMatrix = new int[col][row - 1];  
 for (int j = col - 1; j >= 0; j--)   
 for (int i = 1; i < row; i++)   
 newMatrix[col - 1 - j][i - 1] = matrix[i][j];   
 return newMatrix;  
}**

**public ArrayList<Integer> printMatrix\_1(int[][] matrix) {  
 ArrayList<Integer> al = new ArrayList<>();  
 int row = matrix.length;  
 if (row == 0)  
 return al;  
 int col = matrix[0].length;  
 *// 短的边/2，向上取整* int circle = ((row > col ? col : row) + 1) / 2;  
 for (int i = 0; i < circle; i++) {  
 *// 从左向右打印，j=i; j<col-i, 这一行的前i个已经在第i圈从下往上被打印，故j=i  
 // 倒数i个都已经在第i圈从上往下被打印，故j=col-i-1<col-i* for (int j = i; j < col - i; j++)  
 al.add(matrix[i][j]);  
 for (int j = i + 1; j < row - i; j++)  
 al.add(matrix[j][col - i - 1]);  
*// 从右往左打印，j=col-i-2;j>=i&&row-i-1!=i;，这一行倒数i个已经在第i圈从上往下被打印  
 // 这一行倒数第i+1个已经在从上往下时被打印，故j=col-1-(i+1)=col-i-2  
 // 这一行的前i个已经在从下往上时被打印，故j=i>=i  
 // 当第i圈为0时即从未由上往下打印时，col有多列时，会造成重复打印，故判断row-i-1!=i以避免* for (int j = col - i - 2; j >= i && row - i - 1 != i; j--)  
 al.add(matrix[row - i - 1][j]);  
 for (int j = row - i - 2; j > i && col - i - 1 != i; j--)  
 al.add(matrix[j][i]);  
 }  
 return al;  
}**

**public class Solution {  
 private ArrayList<Integer> dataList = new ArrayList<>();  
 private ArrayList<Integer> minList = new ArrayList<>();  
 private Integer min = Integer.*MAX\_VALUE*;  
 public void push(int node) {  
 dataList.add(node);  
 if (node <= min) {  
 minList.add(node);  
 min = node;  
 } else { minList.add(min); }  
 }  
 public int getSize() { return dataList.size(); }  
 public void pop() {  
 int end = getSize() - 1;  
 dataList.remove(end);  
 minList.remove(end);  
 min = minList.get(getSize() - 1);  
 }  
 public int top() { return dataList.get(getSize() - 1); }  
 public int min() { return min; }  
}**

**包含min方法的栈**

**定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的min函数（时间复杂度应为O（1））。**

**栈的压入，弹出序列**

**输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否可能为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，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就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）**

**public boolean IsPopOrder(int [] pushA,int [] popA) {  
 if(pushA.length == 0 || popA.length == 0)  
 return false;  
 Stack<Integer> s = new Stack<Integer>();  
 *//用于标识弹出序列的位置* int popIndex = 0;  
 for(int i = 0; i< pushA.length;i++){  
 s.push(pushA[i]);  
 *//如果栈不为空，且栈顶元素等于弹出序列* while(!s.empty() &&s.peek() == popA[popIndex]){  
 *//出栈* s.pop();  
 *//弹出序列向后一位* popIndex++;  
 }  
 }  
 return s.empty();  
}**

**重上往下打印二叉树**

**从上往下打印出二叉树的每个节点，同层节点从左至右打印。**

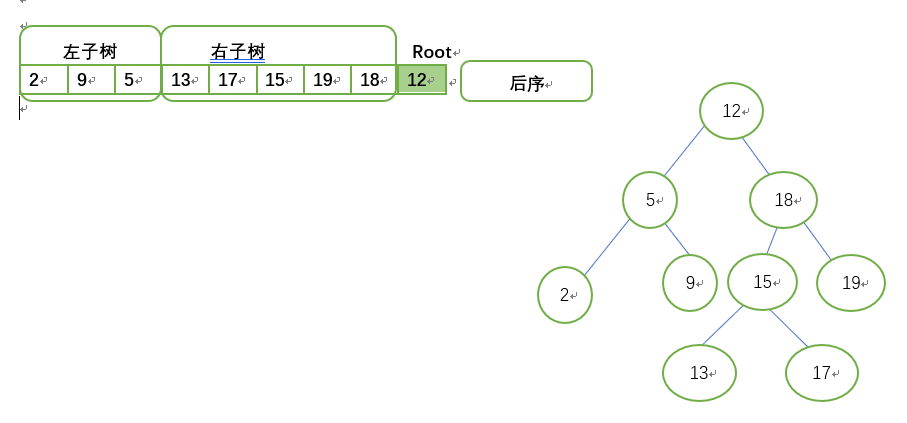
**思路：用两个列表，一个存值，一个存节点，节点列表每次获取值后移除，**

**public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {  
 ArrayList<Integer> list = new ArrayList<>();  
 ArrayList<TreeNode> queue = new ArrayList<>();  
 if (root == null) {  
 return list;  
 }  
 queue.add(root);  
 while (queue.size() != 0) {  
 TreeNode temp = queue.remove(0);  
 if (temp.left != null){  
 queue.add(temp.left);  
 }  
 if (temp.right != null) {  
 queue.add(temp.right);  
 }  
 list.add(temp.val);  
 }  
 return list;  
}**

**二叉搜索树的后序遍历**

**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。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。**

**思路：  
BST的后序序列的合法序列是，对于一个序列S，最后一个元素是x （也就是根），如果去掉最后一个元素的序列为T，那么T满足：T可以分成两段，前一段（左子树）小于x，后一段（右子树）大于x，且这两段（子树）都是合法的后序序列。完美的递归定义 : ) 。**

****

**public boolean VerifySquenceOfBST(int [] sequence) {  
 if(sequence.length == 0){  
 return false;  
 }  
 if(sequence.length == 1){  
 return true;  
 }  
 return judge(sequence,0,sequence.length-1);  
}  
public boolean judge(int[] a,int start,int end){  
 if(start >= end){  
 return true;  
 }  
 int i = start;  
 while(a[i] < a[end]){  
 ++i;// 确定根节点  
 }  
 for(int j=i;j<end;j++){  
 if(a[j] < a[end]){  
 return false;  
 }  
 }  
 return judge(a,start,i-1) && judge(a,i,end-1);  
}**

**二叉树的深度**

**输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（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。**

**public int TreeDepth(TreeNode pRoot){  
 return pRoot == null? 0 : Math.*max*(TreeDepth(pRoot.left),TreeDepth(pRoot.right)) + 1;  
}**

**public int TreeDepth(TreeNode pRoot)  
 {  
 if(pRoot == null){  
 return 0;  
 }  
 Queue<TreeNode> queue = new LinkedList<TreeNode>();  
 queue.add(pRoot);  
 int depth = 0, count = 0, nextCount = 1;  
 while(queue.size()!=0){  
 TreeNode top = queue.poll();  
 count++;  
 if(top.left != null){  
 queue.add(top.left);  
 }  
 if(top.right != null){  
 queue.add(top.right);  
 }  
 if(count == nextCount){  
 nextCount = queue.size();  
 count = 0;  
 depth++;  
 }  
 }  
 return depth;  
 }  
}**

****

**public static void main(List<Integer> nums,int num) {  
 nums.sort((a, b) ->a.compareTo(b));  
 List<Integer> list1 = new ArrayList<>();  
 int avg = nums.stream().reduce(Integer::*sum*).get() / 2;  
 for (int i = nums.size() - 1; i >= 0; i--){  
 int s1 = list1== null || list1.size() <= 0 ? 0 :list1.stream().reduce(Integer::*sum*).get() ;  
 if (Math.*abs*(avg - s1) > Math.*abs*(avg - (s1 + nums.get(i)) )){  
 list1.add(nums.remove(i));  
 }else {  
 break;  
 }  
 }  
 System.*out*.println(Math.*abs*(nums.stream().reduce(Integer::*sum*).get()  
 - list1.stream().reduce(Integer::*sum*).get()));  
 System.*out*.println(Math.*abs*(nums.size() - list1.size()));  
}**

**滑动窗口的最大值  
给定一个数组和滑动窗口的大小，找出所有滑动窗口里数值的最大值。例如，如果输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小3，那么一共存在6个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}； 针对数组{2,3,4,2,6,2,5,1}的滑动窗口有以下6个： {[2,3,4],2,6,2,5,1}， {2,[3,4,2],6,2,5,1}， {2,3,[4,2,6],2,5,1}， {2,3,4,[2,6,2],5,1}， {2,3,4,2,[6,2,5],1}， {2,3,4,2,6,[2,5,1]}。**

**思路：**

**public static ArrayList<Integer> maxInWindows(int[] num, int size) {  
 *// 存放最大值* ArrayList<Integer> ret = new ArrayList<>();  
 *// 存放滑动窗口* Queue<Integer> integerQueue = new LinkedList<>();  
  
 if (num == null){  
 return ret;  
 }  
 if ( num.length < size || size < 1){  
 return ret;  
 }  
 for (int i = 0; i < num.length; i++){  
 if (i == num.length - size + 1)  
 break;  
 for (int j = 0; j < size; j++){  
 ((LinkedList<Integer>) integerQueue).addFirst(num[i + j]);  
 }  
 int max = 0;  
 Integer sum = integerQueue.stream().max((a, b) -> a.compareTo(b)).get();  
 ret.add(sum);  
 integerQueue.clear();  
 }  
 return ret;  
}**

**机器人的运动范围**

**地上有一个m行和n列的方格。一个机器人从坐标0,0的格子开始移动，每一次只能向左，右，上，下四个方向移动一格，但是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。但是，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个格子？**

**思路：**

**算法本质：**

**DFS||BFS 寻找连通分量：**

**机器人在一个矩阵上的m\*n个格子上移动，可进入的格子的集合可抽象为以下点集：**

**{ (row, col) | (i%10+i/10+j%10+j/10) <= threshold }。且路径节点可重复，无步数限制。**

**问：机器人能到达多少个格子？**

**倘若我们把矩阵的每一个“格子”抽象成一个“结点”，把“格子相邻”抽象为“结点连通”（结点之间存在无向边），**

**把“无法进入的格子”抽象成“与所有普通结点都不连通（不存在无向边）的孤点”，则整个问题可以抽象为：**

**从某个结点出发，寻找无向图的连通分量的节点个数。很显然，可以使用DFS或者BFS进行实现**

**算法实现：**

**这里选择DFS进行实现。**

**设置两个辅助boolean矩阵：visited与isWall。前者是DFS中的典型辅助矩阵，记录每个节点是否已访问过。 后者用来表示每个节点是否是不能进入的“孤点”。**

**设置静态变量nodeCnt，用于在DFS的过程中记录访问过的结点数**

**DFS递归函数的出口条件设置为：**

**(outOfBoundary(rows, cols, row, col) || visited[row][col] || isWall[row][col] )**

**即：“若超过边界（到矩阵之外）”或“访问过”或“是无法进入的结点” 则 return**

**然后进行DFS。**

**深度优先搜索和广度优先搜索：**

**深度优先搜索属于图算法的一种，是一个针对图和树的遍历算法，英文缩写为DFS即Depth First Search。深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。一般用堆数据结构来辅助实现DFS算法。其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次。**

**广度优先搜索（也称宽度优先搜索，缩写BFS，以下采用广度来描述）是连通图的一种遍历算法这一算法也是很多重要的图的算法的原型，基本过程，BFS是从根节点开始，沿着树(图)的宽度遍历树(图)的节点。如果所有节点均被访问，则算法中止。一般用队列数据结构来辅助实现BFS算法**。

因为开始位置固定，所以不需要

**public class DFSDemo {  
 *// 存放 格子数* int nodeCnt = 0;  
 *// 访问过的路径矩阵* boolean[][] visited;  
 *// 无法到达的孤点。* boolean[][] isWall;  
 int threshold;  
 int rows;  
 int cols;  
 public int movingCount(int threshold, int rows, int cols){  
 if (threshold<0 || rows<=0 || cols<=0) *//robust* return 0;*//牛客示例是0* this.nodeCnt = 0;  
 this.threshold = threshold;  
 this.rows = rows;  
 this.cols = cols;  
 this.visited = new boolean[rows][cols];  
 this.isWall = new boolean[rows][cols];  
 for (int i=0;i<rows;i++){  
 for (int j=0;j<cols;j++){  
 this.visited[i][j]=false;  
 if ( (i%10+i/10+j%10+j/10) > threshold )  
 this.isWall[i][j]=true;  
 else  
 this.isWall[i][j]=false;  
 }  
 }DFS(0,0);  
 return this.nodeCnt;  
 }  
 public void DFS(int row, int col){  
 *// 若超过边界（到矩阵之外）”或“访问过”或“是无法进入的结点”* if ( outOfBoundary(rows, cols, row, col)  
 || visited[row][col]  
 || isWall[row][col] )  
 return;visited[row][col]=true;  
 nodeCnt++;DFS(row+1, col);  
 DFS(row-1, col);  
 DFS(row, col+1);  
 DFS(row, col-1);  
 }  
 public boolean outOfBoundary(int rows, int cols, int row, int col){  
 return ( row<0 || row>=rows || col<0 || col>=cols );  
 }  
}**

**二叉树中和为某一值的路径**

**输入一颗二叉树的根节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为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。(注意: 在返回值的list中，数组长度大的数组靠前)**

**思路：list.remove(list.size() - 1);的作用是实现一种回退，深度遍历完未满足条件，清空list。为下一次遍历准备。**

**private ArrayList<ArrayList<Integer>> listAll = new ArrayList<ArrayList<Integer>>();  
private ArrayList<Integer> list = new ArrayList<Integer>();  
  
public ArrayList<ArrayList<Integer>> FindPath(TreeNode root, int target) {  
 if (root == null)  
 return listAll;  
 list.add(root.val);  
 target -= root.val;  
 if (target == 0 && root.left == null && root.right == null)  
 listAll.add(new ArrayList<Integer>(list));  
 FindPath(root.left, target);  
 FindPath(root.right, target);  
 list.remove(list.size() - 1);  
 return listAll;  
}**

**字符串排序**

**输入一个字符串,按字典序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。**

**思路：**

**对于无重复值的情况， 固定第一个字符，递归取得首位后面的各种字符串组合， 再把第一个字符与后面每一个字符交换，并同样递归获得首位后面的字符串组合，递归的出口，就是只剩一个字符的时候，递归的循环过程，就是从每个子串的第二个字符开始依次与第一个字符交换，然后继续处理子串。**

**假如有重复值呢？**

**由于全排列就是从第一个数字起，每个数分别与它后面的数字交换，我们先尝试加个这样的判断——如果一个数与后面的数字相同那么这两个数就不交换了。**

**例如abb，第一个数与后面两个数交换得bab，bba。然后abb中第二个数和第三个数相同，就不用交换了。 \* 但是对bab，第二个数和第三个数不 同，则需要交换，得到bba。**

**\* 由于这里的bba和开始第一个数与第三个数交换的结果相同了，因此这个方法不行。**

**\* 换种思维，对abb，第一个数a与第二个数b交换得到bab，然后考虑第一个数与第三个数交换，此时由于第三个数等于第二个数，**

**\* 所以第一个数就不再用与第三个数交换了。再考虑bab，它的第二个数与第三个数交换可以解决bba。此时全排列生成完毕！**

**public static ArrayList<String> Permutation(String str) {  
 ArrayList<String> list = new ArrayList<>();  
 if (str != null && str.length() > 0) {  
 *PermutationHelper*(str.toCharArray(), 0, list);  
 }  
 return (ArrayList<String>) list.stream().distinct().sorted().collect(*toList*());  
}  
*// 获取当前子序列的所有排序*private static void PermutationHelper(char[] chars, int i, ArrayList<String> list) {  
 *// 当 交换到最后一个的时候，将字符串添加。* if (i == chars.length - 1) {  
 list.add(String.*valueOf*(chars));  
 } else {  
 for (int j = i; j < chars.length; ++j) {  
 *// 交换a和b  
 swap*(chars, i, j);  
 *// 进行第 b的交换。  
 PermutationHelper*(chars, i + 1, list);  
 *// 将原来的状态恢复。  
 swap*(chars, j, i);  
 }  
 }  
}  
private static void swap(char[] cs, int i, int j) {  
 char temp = cs[i];  
 cs[i] = cs[j];  
 cs[j] = temp;  
}**

**数组中出现次数超过一半的数字**

**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。如果不存在则输出0。**

**思路：使用Map解决，出现一次加一，没有添加，最后得到结果遍历。**

**public int MoreThanHalfNum\_Solution(int [] array) {  
 int length = array.length / 2;  
 Map<Integer, Integer> intMap = new HashMap();  
 for (int i = 0; i < array.length; i++) {  
 if (intMap.containsKey(array[i])) {  
 intMap.put(array[i], intMap.get(array[i]) + 1);  
 } else {  
 intMap.put(array[i], 1);  
 }  
 }  
 for (Integer i:intMap.keySet()  
 ) {  
 if (intMap.get(i) > length){  
 return i;  
 }  
 }  
 return 0;  
}**

**最小的k个数：**

**输入n个整数，找出其中最小的K个数。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,。**

**public static ArrayList<Integer> GetLeastNumbers\_Solution(int [] input, int k) {  
 ArrayList<Integer> list = new ArrayList<>();  
 if (input.length < k){  
 return list;  
 }  
 for(int i = 0; i < k; i++){  
 for(int j = i + 1; j < input.length; j++){  
 if(input[i] > input[j]){  
 *swap*(input, i, j);  
 }  
 }  
 list.add(input[i]);  
 }  
 return list;  
}**

**连续子数组的最大和：**

**HZ偶尔会拿些专业问题来忽悠那些非计算机专业的同学。今天测试组开完会后,他又发话了:在古老的一维模式识别中,常常需要计算连续子向量的最大和,当向量全为正数的时候,问题很好解决。但是,如果向量中包含负数,是否应该包含某个负数,并期望旁边的正数会弥补它呢？例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为8(从第0个开始,到第3个为止)。给一个数组，返回它的最大连续子序列的和，你会不会被他忽悠住？(子向量的长度至少是1)**

**public int FindGreatestSumOfSubArray(int[] array) {  
 int res = array[0]; *//记录当前所有子数组的和的最大值* int max = array[0]; *//包含array[i]的连续数组最大值* for (int i = 1; i < array.length; i++) {  
 max = Math.*max*(max + array[i], array[i]);  
 res = Math.*max*(max, res);  
 }  
 return res;  
}**

**整数一出现的次数**

**求出1~13的整数中1出现的次数,并算出100~1300的整数中1出现的次数？为此他特别数了一下1~13中包含1的数字有1、10、11、12、13因此共出现6次,但是对于后面问题他就没辙了。ACMer希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中1出现的次数（从1 到 n 中1出现的次数）**。

**public int NumberOf1Between1AndN\_Solution(int n) {  
 int count=0;  
 StringBuffer s=new StringBuffer();  
 for(int i=1;i<n+1;i++){  
 s.append(i);  
 }  
 String str=s.toString();  
 for(int i=0;i<str.length();i++){  
 if(str.charAt(i)=='1')  
 count++;  
 }  
 return count;  
}**

**求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。**

**public int Sum\_Solution(int n) {  
 int temp = n;  
 boolean as = (n>0) && (temp += Sum\_Solution(n-1))>0;  
 return temp;  
}**

**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。**

**public int Add(int num1,int num2) {  
 *// 将 BigInteger 的十进制字符串表示形式转换为 BigInteger。* BigInteger bi1=new BigInteger(String.*valueOf*(num1));  
 BigInteger bi2=new BigInteger(String.*valueOf*(num2));  
 *// 返回其值为 (this + val) 的 BigInteger。* return bi1.add(bi2).intValue();  
}**

**剪绳子**

**给你一根长度为n的绳子，请把绳子剪成m段（m、n都是整数，n>1并且m>1），每段绳子的长度记为k[0],k[1],...,k[m]。请问k[0]xk[1]x...xk[m]可能的最大乘积是多少？例如，当绳子的长度是8时，我们把它剪成长度分别为2、3、3的三段，此时得到的最大乘积是18。**

## **输入**

[复制](javascript:void(0);)

8

## **输出**

[复制](javascript:void(0);)

18

**贪婪解法**： **当n大于等于5时，我们尽可能多的剪长度为3的绳子；当剩下的绳子长度为4时，把绳子剪成两段长度为2的绳子。 为什么选2，3为最小的子问题？因为2，3包含于各个问题中，如果再往下剪得话，乘积就会变小。 为什么选长度为3？因为当n≥5时，3(n−3)≥2(n−2)**

**public class Solution {  
 public int cutRope(int target) {  
 if (target == 2){  
 return 1;  
 }  
 if (target == 3){  
 return 1\*2;  
 }  
 int c = target / 3;  
 int y = target % 3;  
 if (y == 0){  
 return (int) Math.*pow*(3, c);  
 }else if (y == 1){  
 return (int) (Math.*pow*(3,c - 1) \* 2 \* 2);  
 }else if (y == 2){  
 return (int) (Math.*pow*(3, c) \* 2);  
 }  
 return target;  
 }  
}**

**矩阵中的路径**

**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。路径可以从矩阵中的任意一个格子开始，每一步可以在矩阵中向左，向右，向上，向下移动一个格子。如果一条路径经过了矩阵中的某一个格子，则该路径不能再进入该格子。 例如 a b c e s f c s a d e e 矩阵中包含一条字符串"bccced"的路径，但是矩阵中不包含"abcb"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个格子之后，路径不能再次进入该格子。**

**思路：**用***递归来实现DFS***  
1.确定***出口***：***false***: 1.**边界条件不满足**，2.当前字符**不匹配**，3.已经**遍历过**  
***true***：字符串str已经**遍历结束**  
2.***递***：**设置访问过**  
递归方式，按照***上下左右***递归  
3.***归***：**复位，未访问**

**public class Solution {  
 public boolean hasPath(char[] matrix, int rows, int cols, char[] str)  
 {  
 if(matrix == null || matrix.length != rows \* cols  
 || str == null || str.length == 0  
 || str.length > matrix.length) return false;  
 boolean[] visited = new boolean[matrix.length];  
 for (int j = 0; j < rows; j++) {  
 for (int i = 0; i < cols; i++) {*//每个节点都有可能是起点* if(dfs(matrix,rows,cols,str,i,j,0,visited)) return true;  
 }*//这里多了个k=0来充当str的索引* }  
 return false;  
 }  
 *//递归开始，真是短啊***

**private boolean dfs(char[] matrix, int rows, int cols, char[] str, int i, int j,int k,  
 boolean[] visited) {  
 if(i < 0 || i >= cols || j < 0 || j >= rows  
 || visited[i + j \* cols] || matrix[i + j \* cols] != str[k])  
 return false;  
 if(k == str.length - 1) return true;*//出口* visited[i + j \* cols] = true;*//递* if(dfs(matrix, rows, cols, str, i, j - 1, k + 1, visited)  
 || dfs(matrix, rows, cols, str, i + 1, j, k + 1, visited)  
 || dfs(matrix, rows, cols, str, i, j + 1, k + 1, visited)  
 || dfs(matrix, rows, cols, str, i - 1, j, k + 1, visited))  
 return true;  
 visited[i + j \* cols] = false;*//归* return false;  
 }**

**数据流中的中位数**

**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数值排序之后位于中间的数值。如果从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。我们使用Insert()方法读取数据流，使用GetMedian()方法获取当前读取数据的中位数。**

**思路。构建两个堆，将数据分为两部分。**

**import java.util.PriorityQueue;  
public class Solution {  
 private int count = 0;  
 *// 此队列的头 是按指定排序方式确定的最小元素，即升序排序* private PriorityQueue<Integer> minHeap = new PriorityQueue<>();  
 *// 指定比较器降序排序* private PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(15, (i1, i2)->i2 - i1);  
 public void Insert(Integer num) {  
 if (count %2 == 0) {  
 *//当数据总数为偶数时，新加入的元素，应当进入小根堆  
 //1.新加入的元素先入到大根堆，由大根堆筛选出堆中最小的元素  
 // 将指定元素插入优先级队列* maxHeap.offer(num);  
 *// 获取并移除此对象的头，如果没有，返回null。* int filteredMaxNum = maxHeap.poll();  
 *//2.筛选后的【大根堆中的最小元素】进入小根堆* minHeap.offer(filteredMaxNum);  
 } else {  
 *//当数据总数为奇数时，新加入的元素，应当进入大根堆  
 //1.新加入的元素先入到小根堆，由小根堆筛选出堆中最大的元素* minHeap.offer(num);  
 *// 获取并移除此对象的头，如果没有，返回null。* int filteredMinNum = minHeap.poll();  
 *//2.筛选后的【小根堆中的最大元素】进入大根堆* maxHeap.offer(filteredMinNum);  
 }  
 count++;  
 }  
 public Double GetMedian() {  
 if (count %2 == 0) {  
 *// peek（）获取但不移除此队列的头；如果此队列为空，则返回 null。* return new Double((minHeap.peek() + maxHeap.peek())) / 2;  
 } else {  
 return new Double(minHeap.peek());  
 }  
 }  
}**

**二叉树的第k个节点**

**给定一棵二叉搜索树，请找出其中的第k小的结点。例如， （5，3，7，2，4，6，8）    中，按结点数值大小顺序第三小结点的值为4。**

***/\*  
public class TreeNode {  
 int val = 0;  
 TreeNode left = null;  
 TreeNode right = null;  
 public TreeNode(int val) {this.val = val; }  
}  
\*/* public class Solution {  
 int index = 0; *//计数器* TreeNode KthNode(TreeNode root, int k)  
 {  
 if(root != null){ *//中序遍历寻找第k个* TreeNode node = KthNode(root.left,k);  
 if(node != null)  
 return node;  
 index ++;  
 if(index == k)  
 return root;  
 node = KthNode(root.right,k);  
 if(node != null)  
 return node;  
 }  
 return null;  
 }  
 }**

**if(node != null) return node;的原因  
必须要对每一个递归调用返回值进行判断if(node != null){return node;}，  
判断返回值是否为null，如果为null就说明在没找到，要继续执行index++ ; if(index == k){...}的寻找过程，  
如果返回不为空，则说明在递归调用判断子节点的时候已经找到符合要求的节点了，则将找到的节点  
逐层向上传递返回。直到返回到第一次调用KthNode的地方。  
如果不对递归调用的返回值做判断，即不执行if(node != null){return node;}，那所找到符合 要求的节点只能返回到上一层，不能返回到顶层（可以自己调试，然后观察方法栈的调用变化）**

**序列化二叉树**

**二叉树的序列化是指：把一棵二叉树按照某种遍历方式的结果以某种格式保存为字符串，从而使得内存中建立起来的二叉树可以持久保存。序列化可以基于先序、中序、后序、层序的二叉树遍历方式来进行修改，序列化的结果是一个字符串，序列化时通过 某种符号表示空节点（#），以 ！ 表示一个结点值的结束（value!）。  
二叉树的反序列化是指：根据某种遍历顺序得到的序列化字符串结果str，重构二叉树。**

**思路，中序遍历**

**public class Synchronization {  
 public int index = -1;  
   
 String Serialize(TreeNode root) {  
 StringBuilder string = new StringBuilder();  
 if (root == null){  
 string.append("#!");  
 return string.toString();  
 }  
 string.append(Serialize(root.left));  
 string.append(root.val);  
 string.append(Serialize(root.right));  
 return string.toString();  
 }  
 TreeNode Deserialize(String str) {  
 index++;  
 int len = str.length();  
 if (index >= len){  
 return null;  
 }  
 String[] strings = str.split(",");  
 TreeNode node = null;  
 if (!strings[index].equals("#")){  
 node = new TreeNode(Integer.*parseInt*(strings[index]));  
 node.left = Deserialize(str);  
 node.right = Deserialize(str);  
 }  
 return node;  
 }  
}**

**把二叉树打印成多行**

从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

***/\*  
 \* 队列LinkedList完成层序遍历，用end记录每层结点数目  
 \*/*public class Solution {  
 ArrayList<ArrayList<Integer>> Print(TreeNode pRoot) {  
 ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
 if(pRoot == null){  
 return result;  
 }  
 Queue<TreeNode> layer = new LinkedList<TreeNode>();  
 ArrayList<Integer> layerList = new ArrayList<Integer>();  
 layer.add(pRoot);  
 int start = 0, end = 1;  
 while(!layer.isEmpty()){  
 TreeNode cur = layer.remove();  
 layerList.add(cur.val);  
 start++;  
 if(cur.left!=null){  
 layer.add(cur.left);  
 }  
 if(cur.right!=null){  
 layer.add(cur.right);  
 }  
 if(start == end){  
 end = layer.size();  
 start = 0;  
 result.add(layerList);  
 layerList = new ArrayList<Integer>();  
 }  
 }  
 return result;  
 }  
}**

**对称的二叉树**

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

**public static boolean isSymmetrical(TreeNode pRoot){  
 if(pRoot == null) return true;  
 return *isSymmetrical*(pRoot.left, pRoot.right);  
 }  
 private static boolean isSymmetrical(TreeNode left, TreeNode right) {  
 if(left == null && right == null) return true;  
 if(left == null || right == null) return false;  
 return left.val == right.val *//为镜像的条件：左右节点值相等* && *isSymmetrical*(left.left, right.right);*//2.对称的子树也是镜像*}**