//1.字符型转化为int数组(1)

char c='1';

int i=(int)c;

System.out.println(i);//i=49.

//2.字符型转化为int数组(2)

char a = '1';

int k= Integer.parseInt(String.valueOf(a));

System.out.println(k);//输出1

//3.字符串转int

String s="123";

int ii=Integer.parseInt(s);

System.out.println(ii);//ii=123

1、输入的是字符串

Scanner sc = **new** Scanner(System.in);

String line = sc.nextLine();

nextLine()：方法返回的是Enter键之前的所有字符,它是可以得到带空格的字符串的

2、输入的是整数

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

1. hasNext()和Next()效果其实是一样的，系统都会等待输入下一个字符，**只是返回值不同，hasNext()会返回true，next()返回输入的字符。**
2. **substring(x)**是从字符串的的第x个字符截取
3. **substring(x,y）**是从x到**y前的位置**停止（**注意是从0开始**）

6、int x = a.lastIndexOf(b),**表示b字符串在a字符串中最后出现的位置**。如

a= "abcdabcd";b="d";那么x的值为7.

/\*

public class Solution {

public int FirstNotRepeatingChar(String str) {

if(str==null || str.length()==0)

return -1;

List<Character> list=new ArrayList<>();

for(int i=0;i<str.length();i++)

{

char ch=str.charAt(i);

if(!list.contains(ch))

{

list.add(Character.valueOf(ch));

}

else{

list.remove(Character.valueOf(ch));

}

}

if(list.size()<=0)

return -1;

return str.indexOf(list.get(0));

//Character:Java语言为内置数据类型char提供了包装类Character类。

// **str.charAt(i);**返回字符串中索引为i的字母，而且字符串的索引是从0开始的，和数组一样

**Str.toCharArray();**

**list.add(Character.valueOf(ch)):**将ch转换成Character类型

**str.indexOf(list.get(0)):返回字母的索引，**这个indexOf很重要呀

ArrayList<Integer> list=new ArrayList<Integer>();这里ArrayList里面装的是Integer对象，而不是int,

List.add(“5”)成立；但是list.remove（Integer对象）而不是list.remove（“int”）;

1、java 中要将一个字符串转换成一个字符数组需要使用 String 类中提供的 toCharArray() 方法进行操作.

例:

|  |  |
| --- | --- |
| 1  2 | String str = "abcde";  **char**[] ch = str.toCharArray(); |

# 2、 java中怎么把一个整数数组转化成字符串

// 第一种

**int**[] is = {1,2,3,4};

String str = Arrays.toString(is);

// 第二种

System.out.println(str);

StringBuffer sb = **new** StringBuffer();

**for**(**int** i=0;i<is.length;i++){

   sb.append(is[i]);

}

System.out.println(sb.toString());

// 第三种

Char [] ch= new char[];

String stroutput=String.valueOf(ch1);

3、下面这个函数是最重要的表示根据空格来分成多个字符串，然后变成数组，很重要

String string=str;

String[] a=string.split(" ");

4、if(str.trim().equals(""))///trim()函数很重要，下面是它的作用例子

return str;

String s=" abc ";  
String s2=s.trim();  
s2="abc"

# break和continue的区别

break：

(1).**结束当前整个循环**，执行当前循环下边的语句。忽略循环体中任何其它语句和循环条件测试。

(2).**只能跳出一层循环**，如果你的循环是嵌套循环，那么你需要按照你嵌套的层次，逐步使用break来跳出。

continue：

(1).**终止本次循环的执行**，即跳过当前这次循环中continue语句后尚未执行的语句，接着进行下一次循环条件的判断。

(2).**结束当前循环**，进行下一次的循环判断。

(3).终止当前的循环过程，但他并不跳出循环,而是继续往下判断循环条件执行语句.他只能结束循环中的一次过程,但不能终止循环继续进行。

1. **当编程的时候要引入包的话就引这个**

**import java.util.\*;**

7、isEmpty() 或者(list.size() == 0)用于判断List内容是否为空，即表里一个元素也没有， 但是使用isEmpty()和size()的前提是，list是一个空集合，而不是null，

**null判断 ：**list等于null，**可理解为没有对list集合分配内存空间，实际上压根就不存在。**

# break和continue的区别

1、**判断是否为空**:对于栈Stack为: **S.empty()**;

对于Queue：**S.isEmpty()**;

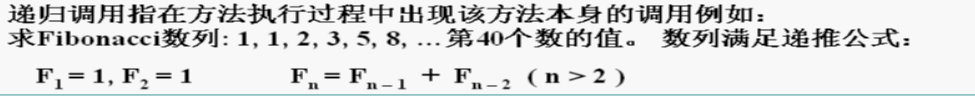
对于集合ArrayList: **S.isEmpty()**;

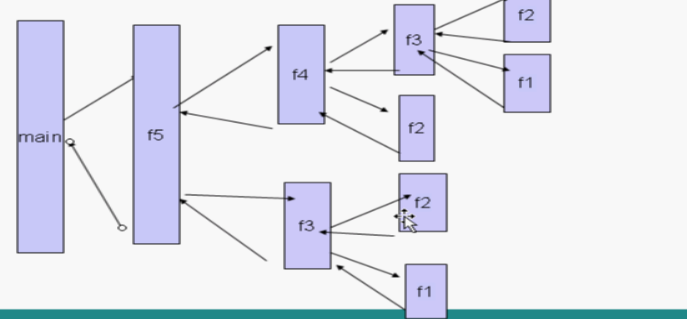
2、对于队列Queue来说，添加元素：**offer()**;

删除元素：**poll()**;

因为此时 Queue<TreeNode> s1 = **new LinkedList**<TreeNode>();相当LinkedList实现了Queue接口。

# 9、对递归的理解





1. **public** classFab {
3. **public** **static** **void** main(String args[]){
4. System.out.println(fab(5));
5. }
7. **private** **static** **int** fab(**int** index){
8. **if**(index==1 || index==2){
9. **return** 1;
10. }**else**{
11. **return** fab(index-1)+fab(index-2);
12. }
13. }
14. }

# Collections.sort(list)：针对集合的排序方法

# Arrays.sort(a)：针对数组的排序方法

前者要加入**import java.util.Collections;**

**后者要加入import java.util.Arrays;**

1. **HashMap<Character,Integer> map=new HashMap<Character,Integer>();**

**记住这个HashMap的写法，哪个地方应该大写，哪个地方应该小写**

**而且头文件也是只需要写import java.util.\*;**

**if(map.containsKey(ch))，记住这个，意思当HashMap中存在这个键的时候，用的是containsKey(),而ArrayList用的是contains();**

**12、StringBuilder sb=new StringBuilder();**

**return sb.toString()；**

**注意toString();这个方法的写法，有可能会忘**

**11、**

**在SpringMVC后台控制层获取参数的方式主要有两种，**

**一种是request.getParameter(“name”)，另外一种是用注解@RequestParam直接获取。**

**这里主要讲这个注解 @RequestParam**

**接下来我们看一下@RequestParam注解主要有哪些参数：**

**value：参数名字，即入参的请求参数名字，如username表示请求的参数区中的名字为username的参数的值将传入；**

**required：是否必须，默认是true，表示请求中一定要有相应的参数，否则将报404错误码；**

**defaultValue：默认值，表示如果请求中没有同名参数时的默认值，例如：**

**public List getItemTreeNode(@RequestParam(value=”id”,defaultValue=”0”)long parentId)**

**2、@PathVariable绑定URI模板变量值**

**@PathVariable用于将请求URL中的模板变量映射到功能处理方法的参数上。**

**//配置url和方法的一个关系**

**@RequestMapping(“item/{itemId}”)**

**/\* @RequestMapping 来映射请求，也就是通过它来指定控制器可以处理哪些URL请求,类似于struts的.action请求-**

**\* @responsebody表示该方法的返回结果直接写入HTTP response body中**

**一般在异步获取数据时使用，在使用@RequestMapping后，返回值通常解析为跳转路径，加上@responsebody后返回结果不会被解析为跳转路径，而是直接写入HTTP response body中。**

**比如异步获取json数据，加上@responsebody后，会直接返回json数据。\***

**@Pathvariable注解绑定它传过来的值到方法的参数上**

**用于将请求URL中的模板变量映射到功能处理方法的参数上，即取出uri模板中的变量作为参数**

**\*/**

**@ResponseBody**

**public TbItem getItemById(@PathVariable Long itemId){**

**public class BinaryTree<T>**

**{**

**/\***

**\* 先序创建二叉树**

**\* 返回：根节点**

**\*/**

**public TreeNode<T> creatBinaryPre(LinkedList<T> treeData)**

**{**

**TreeNode<T> root=null;**

**T data=treeData.removeFirst();**

**if (data!=null)**

**{**

**root=new TreeNode<T>(data, null, null);**

**root.left=creatBinaryPre(treeData);**

**root.right=creatBinaryPre(treeData);**

**}**

**return root;**

**}**

**///////////////////////////////////////////////////////////////////////////////////////////////////////////////////**

**/\***

**\* 先序遍历二叉树（递归）**

**\*/**

**public void PrintBinaryTreePreRecur(TreeNode<T> root)**

**{**

**if (root!=null)**

**{**

**System.out.print(root.data);**

**PrintBinaryTreePreRecur(root.left);**

**PrintBinaryTreePreRecur(root.right);**

**}**

**}**

**///////////////////////////////////////////////////////////////////////////////////////////////////////////**

**/\***

**\* 中序遍历二叉树（递归）**

**\*/**

**public void PrintBinaryTreeMidRecur(TreeNode<T> root)**

**{**

**if (root!=null)**

**{**

**PrintBinaryTreeMidRecur(root.left);**

**System.out.print(root.data);**

**PrintBinaryTreeMidRecur(root.right);**

**}**

**}**

**/////////////////////////////////////////////////////////////////////////////////////////////////////**

**/\***

**\* 后序遍历二叉树（递归）**

**\*/**

**public void PrintBinaryTreeBacRecur(TreeNode<T> root)**

**{**

**if (root!=null)**

**{**

**PrintBinaryTreeBacRecur(root.left);**

**PrintBinaryTreeBacRecur(root.right);**

**System.out.print(root.data);**

**}**

**}**

**}**

**////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////**

**1、二维数组的查找**

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。

**public class Solution {**

**public boolean Find(int target, int [][] a) {**

**int row=a.length;**

**int clum=a[0].length;**

**int i=row-1;**

**int j=0;**

**while(i>=0 && j<clum){//这里是最重要的**

**if(target>a[i][j]){**

**j++;**

**}**

**else if(target<a[i][j]){**

**i--;**

**}**

**else**

**return true;**

**}**

**return false;**

**}**

**}**

**2、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 class Solution {**

**public String replaceSpace(StringBuffer str) {**

**if(str.length()==0)**

**return "";**

**StringBuilder sb=new StringBuilder();**

**for(int i=0;i<str.length();i++){**

**if(str.charAt(i)==' '){**

**sb.append("%20");**

**}**

**else{**

**sb.append(str.charAt(i));**

**}**

**}**

**return sb.toString();**

**}**

**}**

1. **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。

**/\***

**public class ListNode {**

**int val;**

**ListNode next = null;**

**ListNode(int val) {**

**this.val = val;**

**}**

**}\*/**

**/\***

**思路： 如果存在共同节点的话，那么从该节点，两个链表之后的元素都是相同的。**

**\*\*也就是说两个链表从尾部往前到某个点，节点都是一样的,这句话是超级，最重要的，没有这句话就没理解到这道题的意思\*\***

**我们可以用两个栈分别来装这两条链表。一个一个比较出来的值。**

**找到第一个相同的节点。**

**\*/**

import java.util.Stack;

public class Solution {

public ListNode FindFirstCommonNode(ListNode pHead1, ListNode pHead2) {

if(pHead1==null||pHead2==null)

return null;

Stack<ListNode> stack1=new Stack<>();

Stack<ListNode> stack2=new Stack<>();

**//将第一个链表的元素压入第一个栈**

while(pHead1!=null)

{

stack1.push(pHead1);

pHead1=pHead1.next;

}

**//将第二个链表的元素压入第二个栈**

while(pHead2!=null)

{

stack2.push(pHead2);

pHead2=pHead2.next;

}

**//创建需要返回的公共节点**

ListNode commonList=null;

**//1、下面这个句子好好理解，这句话是说当第一个栈的栈顶元素和第二个栈的栈顶元素相等时，就把第一个栈的栈顶元素去掉，同时**

**//2、公共节点的值被赋予第二个栈的栈顶元素，一直循环，到这两个栈的栈顶元素不相同为止，此时commonList里面的值就是第一个公共节点**

while(!stack1.isEmpty()&&!stack2.isEmpty()&&stack1.peek()==stack2.peek())

{

stack1.pop();**//去掉栈1的栈顶元素**

commonList=stack2.pop();**//去掉栈2的栈顶元素，并把这个值赋予公共节点。**

}

return commonList;

}

}

**4、输入一个链表，按链表值从尾到头的顺序返回一个ArrayList。**

public class Solution {

public ArrayList<Integer> printListFromTailToHead(ListNode listNode) {

ArrayList<Integer> arr= new ArrayList<Integer>();

if(listNode==null)

return arr;

Stack<Integer> stack=new Stack<>();

while(listNode!=null){

stack.push(listNode.val);

listNode=listNode.next;

}

while(!stack.isEmpty()){

arr.add(stack.pop());

}

return arr;

}

}

**5、输入一个链表，反转链表后，输出新链表的表头**

public class Solution {

public ListNode ReverseList(ListNode head) {

if(head ==null)

return null;

ListNode prepoint=null;

ListNode nextpoint=null;

while(head!=null){

//这句话的意思是将head后面一点先存在nextpoint这里，便于后面用

nextpoint=head.next;

**head.next=prepoint;**

**prepoint=head;**

head=nextpoint;

**//转换的时候其实用的prepoint和head这两个点，nextpoint其实就是中转作用**

}

return prepoint;

}

}

1. **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：1、先求每个滑动窗口，运用ArrayList

// 2、再将窗口里的值进行排序

**//3、排完序后,最大值就是这个窗口中的最后一位**

**//4、依次将每个窗口的最后一位给存到另外一个ArrayList中**

//5?注意需要运用这个Collections这样的一个类中sort方法，因为ArrayList**继承了Collection**

import java.util.\*;

import java.util.Collections;

public class Solution {

public ArrayList<Integer> maxInWindows(int [] num, int size)

{

ArrayList<Integer> list1=new ArrayList<Integer>();

//这里要注意限制条件，因为我就是忽略了size==0，所以就只过了75%的测试用例

if(num.length==0 || size==0)

return list1;

int length=num.length;

//下面的这个条件是说当窗口长度比数组长度还长的时候，那么就不行

**if(size>num.length){**

**return list1;**

**}**

**for(int i=0;i<=length-size;i++){**

**//这里每次循坏都要新建一个集合，所以每次list都会被赋一个全新的没有值的集合，这样就解释了为什么list每一次循坏添加的都是新的值**

**ArrayList<Integer> list=new ArrayList<Integer>();**

**for(int j=i;j<i+size;j++){**

**list.add(num[j]);**

**}**

//Arrays.sort(a); //数组排序方法的调用，对比一下

**Collections.sort(list);**

//这下面就是直接添加每个窗口的最后一个元素了

**list1.add(list.get(list.size()-1));**

**}**

return list1;

}

}

1. **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.

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

import java.util.ArrayList;

public class Solution {

public ArrayList<Integer> printMatrix(int [][] matrix) {

ArrayList<Integer> result=new ArrayList<Integer>();

//求多少行

//求多少列

int n = matrix.length,m = matrix[0].length;

if(n==0)

return result;

if(m==0)

return result;

int layers=(Math.min(n,m)-1)/2+1;

**for(int i=0;i<layers;i++)**

**{**

**// n指的是行数，m指的是列数，这里m-i是指每循环一层就会减1**

**for(int k=i;k<m-i;k++)//变的是列数m**

**{**

**result.add(matrix[i][k]);**

**}**

**// 这里变的是行，但是要从现有层数加1开始**

**for(int j=i+1;j<=n-i-1;j++)//数n**

**{**

**result.add(matrix[j][m-1-i]);**

**}**

**// 这里i只是层数，为什么要k>=i呢**

**for(int k=m-1-i-1;(k>=i)&&(n-1-i!=i);k--) ///数m**

**//for(int k=m-1-i-1;k>=i;k--)**

**{**

**result.add(matrix[n-1-i][k]);**

**}**

**// for(int j=n-1-i-1;(j>i)&&(m-i-1!=i);j--)**

**for(int j=n-1-i-1;(j>i)&&(m-i-1!=i);j--)**

**//b变的是行数n;下面的两个循环真的需要仔细去分析，很是蛋疼**

**{**

**result.add(matrix[j][i]);**

**}**

}

return result;

}

}

1. **合并两个排序的链表**

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。

import java.util.ArrayList;

import java.util.Collections;

//import java.util.Comparator;

public class Solution {

public ListNode Merge(ListNode list1,ListNode list2) {

if(list1==null && list2==null)

return null;

ListNode head1=list1;

ListNode head2=list2;

ArrayList<Integer> list=new ArrayList<Integer>();

while(head1!=null){

list.add(head1.val);

head1=head1.next;

}

while(head2!=null){

list.add(head2.val);

head2=head2.next;

}

Collections.sort(list);

**//这里要注意啊，主要是要新建一个头，我之前用的是ListNode head =null;错了**

ListNode head=new ListNode(list.get(0));

ListNode head3=head;

for(int i=1;i<list.size();i++){

ListNode node=new ListNode(list.get(i));

head3.next=node;

head3=head3.next;

}

return head;

}

}

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 class Solution {

public int TreeDepth(TreeNode root) {

return depth(root,0);

}

public int depth(TreeNode root,int d){

if(root==null){

return d;

}else

**d++**;

int leftdepth=depth(root.left,d);

int rightdepth=depth(root.right,d);

return Math.max(leftdepth,rightdepth);

}

}

1. **序列化二叉树**

**请实现两个函数，分别用来序列化和反序列化二叉树**

public class Solution {

int index = -1; //计数变量

// 前序遍历：根节点-左子树-右子树

String Serialize(TreeNode root) {

StringBuilder sb = new StringBuilder();

if(root == null){

sb.append("#,");

// 下面的代码是递归的出口，也就是最底层不满足条件后一层一层的往上返回值

return sb.toString();

//return null;

}

sb.append(root.val + ",");

sb.append(Serialize(root.left));

sb.append(Serialize(root.right));

//这段代码指的是每一个递归返回的值，也就是一个

return sb.toString();

}

TreeNode Deserialize(String str) {

index++;

//int len = str.length();

//if(index >= len){

// return null;

// }

String[] strr = str.split(",");

TreeNode node = null;

if(!strr[index].equals("#")){

node = new TreeNode(Integer.valueOf(strr[index]));

node.left = Deserialize(str);

node.right = Deserialize(str);

}

return node;

}

}

1. **判断一颗二叉树是否是平衡二叉树**

题目：输入一棵二叉树，判断该二叉树是否是平衡二叉树

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

思路：平衡二叉树性质：它的左子树和右子树都是平衡二叉树，**且左子树和右子树的深度之差的绝对值不超过1**，这是最重要的

1、**当根节点 root是null的时候，这也是个平衡二叉树**，我之前不知道这个性质

2、**求出根节点左右子节点的深度，然后判断这两个子节点的深度只差的绝对值是否不超过1，**没超过，就是平衡二叉树，反之，则不是

public class Solution {

public boolean IsBalanced\_Solution(TreeNode root) {

//这里我是真的忽略了，要注意

if(root==null)

return true;

if(root.left==null && root.right!=null)

return false;

if(root.left!=null && root.right==null)

return false;

if(root.left==null && root.right==null)

return true;

// Solution m=new Solutoin();

//**求出根节点左右子节点的深度,记住从0开始**

int i= depth(root.left,0);

int j= depth(root.right,0);

if( Math.abs(i-j)<=1)

return true;

else

return false;

}

//求深度的函数，切记

public int depth(TreeNode root,int d)

{

if(root==null)

return d;

else

d++;

int leftdepth=depth(root.left,d);

int rightdepth=depth(root.right,d);

return Math.max(leftdepth,rightdepth);

}

}

1. **二进制中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**。

public class Solution {

public int NumberOf1(int n) {

int count=0;

if(n<0){

n=n & 0x7FFFFFFF;这里是要将负数的最左边的那个1给去掉，变成正数，同时要注意的是这里为什么要用8位数来去掉，而不是用9位或者10位,**这里很重要啊**

count++;

}

while(n!=0){

count=count+(n&1);//这里n&1是很有味道的

n=n>>1; //这里>>是右移的意思，就是去除最右边的位

}

return count;

}

}

1. **连续子数组的最大和**

public class Solution {

public int FindGreatestSumOfSubArray(int[] array) {

ArrayList<Integer> list=new ArrayList<Integer>();

//int sum=0;我之前就是放到这里的，但是这是不对的

//int result=0;

for(int i=0;i<array.length;i++){

**int sum=0;**//这个地方是关键的，这句话不能放到上面，因为每个子数组的和都是从0开始的，如果放到上面的话 ，那每次改变i的时候就不会重新赋于sum=0，而是会继续累加起来，这就不好了。

for(int j=i;j<array.length;j++){

sum=sum+array[j];

list.add(sum);

}

}

if(list.size() <=0) return 0;

Collections.sort(list);

return list.get(list.size()-1);

}

}

1. **判从上往下打印二叉树**

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**。

public class Solution {

public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {

ArrayList<TreeNode> Nodelist=new ArrayList<TreeNode>();//**这里是要存储结点**

ArrayList<Integer> Vallist=new ArrayList<Integer>();//**这里是要存储结点里的值**

if(root==null)

return Vallist;

Nodelist.add(root);

Vallist.add(root.val);

**//这句话的意思是遍历Nodlist中的所有结点，如果其左右儿子结点都不为null，则添加到Nodelist中**

for(int i=0; i<Nodelist.size();i++){

TreeNode node=Nodelist.get(i);

if(node.left!=null){

Nodelist.add(node.left);

Vallist.add(node.left.val);

}

if(node.right!=null){

Nodelist.add(node.right);

Vallist.add(node.right.val);

}

}

return Vallist;

}

}

1. **判链表中倒数第k个结点**

输入一个链表，输出该链表中倒数第k个结点。

import java.util.\*;

public class Solution {

public ListNode FindKthToTail(ListNode head,int k) {

**if(head==null || k<=0)**//这里我没有考虑k=0的情况，所以会报错

return null;

ArrayList<ListNode> list =new ArrayList<ListNode>();

while(head!=null){

list.add(head);

head=head.next;

}

//下面的代码表示ArrayList这个集合中的元素数量如果要小于k的话，就会返回null;

if(list.size()<k){

return null;

}

**ListNode A=list.get(list.size()-k);**

return A;

}

}

1. **判第一个只出现一次的字符**

在一个字符串(0<=字符串长度<=10000，全部由字母组成)**中找到第一个只出现一次的字符,并返回它的位置**, 如果没有则返回 -1（需要区分大小写）.

**思路：1、遍历字符串，用hashmap来存字符串中的字符和其对应出现的次数。**

1. **在遍历字符串的过程中，如果hashmap中存在当前的字符串，则将其次数加上1并赋值到map中，如果hashmap不存在当前的字符串，则直接将它的次数赋1；**
2. **遍历该字符串，如果其次数等于1，则返回当前字符串的索引**

import java.util.\*;

// use linkedhashmap to keep the order

public class Solution {

public int FirstNotRepeatingChar(String str) {

if(str.length()==0){

return -1;

}

HashMap<Character,Integer> map =new HashMap<Character,Integer>();

for(int i=0;i<str.length();i++){

if(map.containsKey(str.charAt(i))){

int time=map.get(str.charAt(i));**//这里很重要，求出该字符串出现的次数**

map.put(str.charAt(i),++time);**//这里必须是++time而不能是time++,因为要先加1**

}else

{

map.put(str.charAt(i),1);

}

}

for(int j=0;j<str.length();j++){

if(map.get(str.charAt(j))==1){

return j;

}

}

return -1;

}

}

1. **链表中环的入口结点**

给一个链表，若其中包含环，请找出该链表的环的入口结点，否则，输出null。

//第一步，找环中相汇点。分别用p1，p2指向链表头部，p1每次走一步，p2每次走二步，直到p1==p2找到在环中的相汇点。

//第二步，找环的入口。接上步，当p1==p2时，**p2所经过节点数为2x,p1所经过节点数为x**,设环中有n个节点,p2比p1多走一圈有2x=n+x;

//n=x;可以看出p1实际走了一个环的步数，再让p2指向链表头部，p1位置不变，p1,p2每次走一步直到p1==p2; 此时p1指向环的入口。

public class Solution {

public ListNode EntryNodeOfLoop(ListNode pHead)

{

if(pHead==null)

return null;

ListNode slower=pHead;//走一步

ListNode faster=pHead;//走两步

**while(faster!=null && faster.next!=null)**{

slower=slower.next;

faster=faster.next.next;

if(slower==faster){**//这句话的意思是两个指针在环中重叠了**

slower=pHead;//**将slower指针重置到头结点**

while(slower!=faster){

slower=slower.next;

faster=faster.next;

}

return slower;

}

}

return null;

}

}