

# **ACM-ICPC 培训资料汇编**

## **(2)**

### **基本数据结构与算法分册**

**(版本号 1.0.0)**

**哈尔滨理工大学 ACM-ICPC 集训队**

**2012 年 10 月**

## 序

2012 年 5 月，哈尔滨理工大学承办了 ACM-ICPC 黑龙江省第七届大学生程序设计竞赛。做为本次竞赛的主要组织者，我还是很在意本校学生是否能在此次竞赛中取得较好成绩，毕竟这也是学校的脸面。因此，当 2011 年 10 月确定学校承办本届竞赛后，我就给齐达拉图同学很大压力，希望他能认真训练参赛学生，严格要求参训队员。当然，齐达拉图同学半年多的工作还是很有成效，不仅带着黄李龙、姜喜鹏、程宪庆、卢俊达等队员开发了我校的 OJ 主站和竞赛现场版 OJ，还集体带出了几个比较像样的新队员，使得今年省赛我校取得了很好的成绩（当然，也承蒙哈工大和哈工程关照，没有派出全部大牛来参赛）。

在 2011 年 9 月之前，我对 ACM-ICPC 关心甚少。但是，我注意到我校队员学习、训练没有统一的资料，也没有按照竞赛所需知识体系全面系统培训新队员。2011-2012 年度的学生教练们做了一个较详细的培训计划，每周都会给 2011 级新队员上课，也会对老队员进行训练，辛辛苦苦忙活了一年——但是这些知识是根据他们个人所掌握情况来给新生讲解的，新生也是杂七杂八看些资料和做题。在培训的规范性上欠缺很多，当然这个责任不在学生教练。2011 年 9 月，我曾给老队员提出编写培训资料这个任务，一是老队员人数少，有的还要去百度等企业实习；二是老队员要开发、改造 OJ；三是培训新队员也很耗费精力，因此这项工作虽很重要，但却不是那时最迫切的事情，只好被搁置下来。

2012 年 8 月底，2012 级新生满怀梦想和憧憬来到学校，部分同学也被 ACM-ICPC 深深吸引。面对这个新群体的培训，如何提高效率和质量这个老问题又浮现出来。市面现在已经有了各种各样的 ACM-ICPC 培训教材，主要算法和解题思路都有了广泛深入的分析和讨论。同时，互联网博客、BBS 等中也隐藏着诸多大牛对某些算法的精彩论述和参赛感悟。我想，做一个资料汇编，采撷各家言论之精要，对新生学习应该会有较大帮助，至少一可以减少他们上网盲目搜索的时间，二可以给他们构造一个相对完整的知识体系。

感谢 ACM-ICPC 先辈们作出的杰出工作和贡献，使得我们这些后继者们可以站在巨人的肩膀上前行。

感谢校集训队各位队员的无私、真诚和抱负的崇高使命感、责任感，能够任劳任怨、以苦为乐的做好这件我校的开创性工作。

唐远新

2012 年 10 月

## 编写说明

本资料为哈尔滨理工大学 ACM-ICPC 集训队自编自用的内部资料，不作为商业销售目的，也不用于商业培训，因此请各参与学习的同学不要外传。

本分册大纲由黄李龙编写，内容由姜喜朋、侯生智、包春志、孟祥凤、曾卓敏、彭文文等分别编写。

本分册内容由彭文文、孟祥凤校核。

本分册内容大部分采编自各 OJ、互联网和部分书籍。在此，对所有引用文献和试题的原作者表示诚挚的谢意！

由于时间仓促，本资料难免存在表述不当和错误之处，格式也不是很规范，请各位同学对发现的错误或不当之处向[acm@hrbust.edu.cn](mailto:acm@hrbust.edu.cn)邮箱反馈，以便尽快完善本文档。在此对各位同学的积极参与表示感谢！

哈尔滨理工大学在线评测系统（Hrbust-OJ）网址：<http://acm.hrbust.edu.cn>，欢迎各位同学积极参与AC。

国内部分知名 OJ：

杭州电子科技大学：<http://acm.hdu.edu.cn>

北京大学：<http://poj.org>

浙江大学：<http://acm.zju.edu.cn>

以下百度空间列出了比较全的国内外知名 OJ：  
[http://hi.baidu.com/leo\\_xxx/item/6719a5ffe25755713c198b50](http://hi.baidu.com/leo_xxx/item/6719a5ffe25755713c198b50)

哈尔滨理工大学 ACM-ICPC 集训队  
2012 年 10 月

# 目 录

序.....	I
编写说明.....	II
<b>第 1 章 基本数据结构.....</b>	<b>1</b>
1.1 顺序表.....	1
1.1.1 基本原理.....	1
1.1.2 解题思路.....	5
1.1.3 模板代码.....	5
1.1.4 经典题目.....	8
1.2 单链表.....	10
1.2.1 基本原理.....	10
1.2.2 解题思路.....	19
1.2.3 模板代码.....	19
1.2.4 经典题目.....	26
1.2.5 扩展变形.....	31
1.3 双向链表.....	31
1.3.1 基本原理.....	31
1.3.2 解题思路.....	33
1.3.3 模板代码.....	33
1.3.4 经典题目.....	37
1.3.5 扩展变形.....	37
1.4 循环链表.....	37
1.4.1 基本原理.....	37
1.4.2 解题思路.....	39
1.4.3 模板代码.....	39
1.4.4 经典题目.....	41
1.5 栈.....	43
1.5.1 基本原理.....	43
1.5.2 解题思路.....	46
1.5.3 模板代码.....	46
1.5.4 经典题目.....	48
1.5.5 扩展变形.....	49
1.6 队列.....	49
1.6.1 基本原理.....	49
1.6.2 解题思路.....	55
1.6.3 模板代码.....	55
1.6.4 经典题目.....	58
1.6.5 扩展变形.....	59
1.7 串.....	59
1.7.1 基本原理.....	59
1.7.2 经典题目.....	65
1.7.3 扩展变形.....	67
1.8 二叉树.....	67

1.8.1 基本原理	67
1.8.2 二叉树	68
1.8.3 模板代码	69
<b>第 2 章 排序</b>	<b>72</b>
2.1 冒泡排序	72
2.1.1 解题思路	72
2.1.2 模板代码	72
2.1.3 经典题目	72
2.1.4 扩展变型	73
2.2 插入排序	73
2.2.1 基本原理	74
2.2.2 模板代码	74
2.2.3 经典题目	74
2.3 归并排序	75
2.3.1 基本原理	75
2.3.2 解题思路	76
2.3.3 模板代码	76
2.3.4 经典题目	76
2.4 快速排序	78
2.4.1 基本原理	78
2.4.2 解题思路	78
2.4.3 模板代码	79
2.4.4 经典题目	79
2.5 桶式排序（基数排序）	81
2.5.1 解题思路	81
2.5.2 模板代码	81
2.5.3 经典题目	82
<b>第 3 章 基本算法</b>	<b>83</b>
3.1 二分查找	83
3.1.1 模板代码	83
3.1.2 经典题目	83
3.2 模拟	85
3.2.1 基本原理	85
3.2.2 解题思路	85
3.2.3 经典题目	85
3.3 枚举	89
3.3.1 解题思路	89
3.3.2 模板代码	89
3.3.3 经典题目	89
3.4 贪心	91
3.4.1 基本原理	91
3.4.2 解题思路	92
3.4.3 模板代码	92
3.4.4 经典题目	92
3.4.5 扩展变型	94

3.5 递归.....	94
3.5.1 解题思路.....	94
3.5.2 模板代码.....	94
3.5.3 经典题目.....	94
3.5.4 扩展变型.....	95
3.6 递推.....	96
3.6.1 基本原理.....	96
3.6.2 解题思路.....	96
3.6.3 模板代码.....	96
3.6.4 经典题目.....	96
3.6.5 扩展变型.....	97
3.7 分治.....	97
3.7.1 基本原理.....	97
3.7.2 解题思路.....	97
3.7.3 模板代码.....	98
3.7.4 经典题目.....	98
3.7.5 扩展变型.....	99
3.8 高精度计算.....	99
3.8.1 基本原理.....	99
3.8.2 解题思路.....	99
3.8.3 模板代码.....	99
3.8.4 经典题目.....	103
3.9 动态规划入门.....	104
3.9.1 基本原理.....	104
3.9.2 解题思路.....	104
3.9.3 模板代码.....	105
3.9.4 经典题目.....	105
3.9.5 扩展变型.....	106
附记.....	107

## 第1章 基本数据结构

### 1.1 顺序表

#### 1.1.1 基本原理

##### 1.1.1.1 存储方法

线性表的顺序存储结构,也称为顺序表。它是线性表的一种最简单的存储结构。其存储方式为:在内存中开辟一片连续存储空间,但该连续存储空间的大小要大于或等于顺序表的长度和线性表中一个元素所需要的存储字节数的乘积,然后让线性表中第一个元素存放在连续存储空间第一个位置,第二个元素紧跟着第一个之后,其余依此类推。数据元素之间前趋与后继关系体现在存放位置的前后关系上。

##### 1.1.1.2 数据元素的位置确定

假设线性表中元素为  $(a_0, a_1, \dots, a_{n-1})$ , 设第一个元素  $a_0$  的内存地址为  $LOC(a_0)$  而每个元素在计算机内占  $d$  个存储单元, 则第  $i$  个元素  $a_{i-1}$  的地址为  $LOC(a_{i-1}) = LOC(a_0) + (i-1) \times d$  (其中  $0 \leq i \leq n-1$ )

$a_0$	$a_1$	.....	$a_i$	.....	$a_{n-1}$
$loc(a_0)$	$loc(a_0)+d$		$loc(a_0)+i*d$		$loc(a_0)+(n-1)*d$

注意:

在顺序表中, 每个结点  $a_i$  的存储地址是该元素在表中的位置  $i$  的线性函数。只要知道基地址和每个元素占用的存储单元个数, 就可在相同时间内求出任一结点的存储地址。是一种随机存取结构。

##### 1.1.1.3 线性表顺序存储的创建程序

可用数组存放线性表, 用 C 语言描述为:

```
#define MAXSIZE 100
elemtype list[MAXSIZE];
int n; /* n 是线性表的当前长度 */
elemtype 可以是 int, char, float, struct student 等。

void creat_sr_list(int n, elemtype list[])
{
    int i;
    for(i=0; i<n; i++)
        ... /* 输入每个数据元素 例: scanf("%d", &list[i]) */;
}
```

用数组来实现表时, 我们利用了数组单元在物理位置上的邻接关系来表示表中元素之间的逻辑关系。由于这个原因, 用数组来实现表有如下的优缺点。

优点是:

无须为表示表中元素之间的逻辑关系增加额外的存储空间;

可以方便地随机访问表中任一位置的元素。

缺点是：

插入和删除运算不方便，除表尾的位置外，在表的其他位置上进行插入或删除操作都必须移动大量元素，其效率较低；

由于数组要求占用连续的存储空间，存储分配只能预先进行静态分配。因此，当表长变化较大时，难以确定数组的合适的大小。确定大了将造成内存浪费。

顺序表上实现的基本运算

与数据结构密切相关的是定义在其上的一组基本运算，其它复杂的运算（应用）需要调用基本运算来完成。

常见线性表的运算有：

1. 置空表 SETNULL (&L) 将线性表 L 置成空表
2. 求长度 LENGTH (L) 求给定线性表 L 的长度
3. 取元素 GET (L, i) 若  $1 \leq i \leq \text{length}(L)$ , 则取第 i 个位置上的元素，否则取得的元素为 NULL。
4. 求直接前趋 PRIOR (L, X) 求线性表 L 中元素值为 X 的直接前趋，若 X 为第一个元素，前驱为 NULL。
5. 求直接后继 NEXT (L, X) 求线性表 L 中元素值为 X 直接后继，若 X 为最后一个元素，后继为 NULL。
6. 定位函数 LOCATE (L, X) 在线性表 L 中查找值为 X 的元素位置，若有多个值为 X，则以第一个为准，若没有，位置为 0。
7. 插入 INSERT (&L, X, i) 在线性表 L 中第 i 个位置上插入值为 X 的元素。
8. 删除 DELETE (&L, i) 删除线性表 L 中第 i 个位置上的元素。

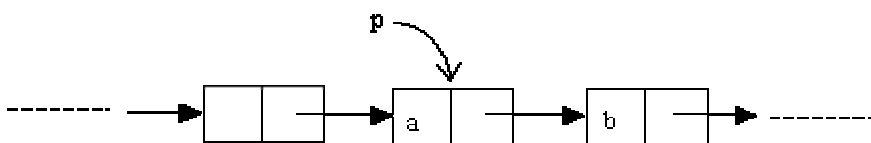
插入运算:在长度为 n 的线性表( $a_0, a_1, a_2, \dots, a_{n-1}$ )中,插入一个新的数据元素 x 到线性表的第 i( $0 \leq i \leq n$ )个位置,使其变为长度为 n+1 的线性表( $a_0, a_1, a_2, \dots, a_{i-1}, x, a_i, \dots, a_{n-1}$ )。

删除运算:在长度为 n 的线性表( $a_0, a_1, a_2, \dots, a_{n-1}$ )中,删除线性表的第 i( $0 \leq i \leq n$ )个位置上的数据元素,使其变为长度为 n-1 的线性表( $a_0, a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_{n-1}$ )。

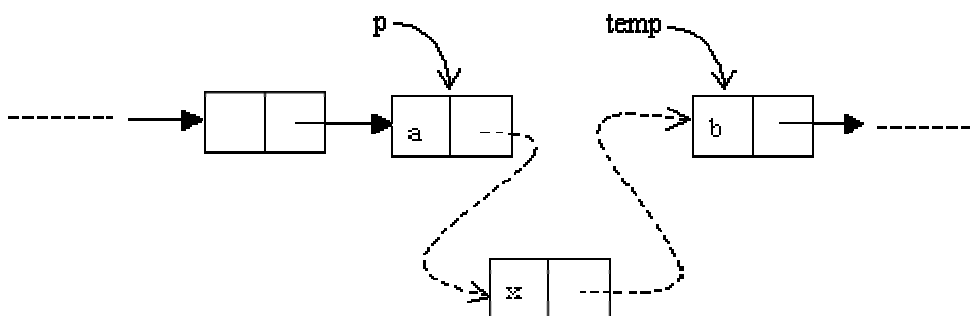
线性表的插入运算（顺序存储结构）

算法：

- 1、将  $a_i, a_{i+1}, \dots, a_{n-1}$  依次后移一个位置，使第 i 位置留空
- 2、将新元素 x 放在空出的位置上。
- 3、线性表长度加 1。







程序:

```
int sq_ins(elemtype list[],int *pn,int i,elemtype x)
{
    int j;
    if(i<0||i>*pn) return(1);
    if(*pn==MAXSIZE) return(2);
    for(j=*pn;j>i;j--)
        list[j]=list[j-1];
    list[i]=x;
    (*pn)++;
    return(0);
}
#define MAXSIZE 100
elemtype list[MAXSIZE];
int n;
main()
{
    int i,n,x;
    scanf("%d",&n);
    create_sq_list(n,list);
    scanf("%d,%d",&i,&x);
    m=sq_ins(list,&n,i,x);
    if(m==1) printf("i error");
    if(m==2) printf("Overflow");
    if(m==0) printf("ins success");
    for(i=0;i<n;i++)
        printf("%d",list[i])
}
```

插入算法花费的时间，主要在于循环中元素移（其它语句花费的时间可以省去），即从插入位置到最后位置的所有元素都要后移一位，使空出的第  $i$  个位置位置插入元素值  $x$ ，但是，插入的位置是不固定的，当插入位置  $i=0$  时，全部元素都得移动，需  $n$  次移动，当  $i=n$  时，不需移动元素，也就是说该算法在最好情况下需要  $O(1)$ 时间复杂度，在最坏情况下需要  $O(n)$ 时间复杂度。由于插入可能在表中任何位置上进行，因此，有必要分析算法的平均性能。设在长度为  $n$  的表中进行插入运算所需的元素移动次数的平均值为  $E_{IN}(n)$ 。由于在表中第  $i$  个位置上插入元素需要的移动次数为  $n-i+1$ ，故

$$E_{IN}(n) = \sum_{i=1}^{n+1} p_i \cdot (n-i+1)$$

其中， $p_i$  表示在表中第  $i$  个位置上插入元素的概率。考虑最简单的情形即假设在表中任何合法位置  $i$  ( $1 \leq i \leq n+1$ ) 上插入元素的机会是均等的，从而，在等概率插入的情况下， $E_{IN}(n)=n/2$  也就是说，用数组实现表时，在表中做插入运算，平均要移动表中一半的元

素，因而算法所需的平均时间仍为  $O(n)$ 。

删除

1、删除运算的逻辑描述

线性表的删除运算是指将表的第  $i$  ( $1 \leq i \leq n$ ) 个结点删去，使长度为  $n$  的线性表

$(a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$

变成长度为  $n-1$  的线性表

$(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

注意：

当要删除元素的位置  $i$  不在表长范围（即  $i < 1$  或  $i > L \rightarrow \text{length}$ ）时，为非法位置，不能做正常的删除操作

2、顺序表删除操作过程

在顺序表上实现删除运算必须移动结点，才能反映出结点间的逻辑关系的变化。若  $i=n$ ，则只要简单地删除终端结点，无须移动结点；若  $1 \leq i \leq n-1$ ，则必须将表中位置  $i+1$ ， $i+2$ ， $\dots$ ， $n$  的结点，依次前移到位置  $i$ ， $i+1$ ， $\dots$ ， $n-1$  上，以填补删除操作造成的空缺。

其删除过程

线性表的删除运算

算法：

1、将  $a_{i+1}, \dots, a_n$  依次前移一个位置

2、线性表长度减 1。

程序：

```
int sq_del(elemtype list[], int *pn, int i)
{
    int j;
    if (i < 0 || i > *pn) return(1);
    for(j=i+1; j < *pn; j++)
        list[j-1] = list[j];
    (*pn)--;
    return(0);
}
```

删除算法花费的时间，主要在于循环中元素的前移（其它语句花费的时间可以省去），即从删除位置到最后位置的所有元素都要前移一位。但是，删除的位置是不固定的，当删除位置  $i=1$  时，全部元素都得移动，需  $n-1$  次移动，当  $i=n$  时，不需移动元素。删除运算的平均性能分析与插入运算类似。设在长度为  $n$  的表中删除一个元素所需的平均移动次数为  $E_{DE}(N)$ 。由于删除表中第  $i$  个

位置上的元素需要的移动次数为  $n-i$ ，故

其中， $p_i$  表示删除表中第  $i$  个位置上元素的概率。在等概率的假设下，

$$p_1 = p_2 = \dots = p_{n+1} = \frac{1}{n+1} \quad \text{这时} \quad E_{DE}(n) = \sum_{i=1}^n (n-i) / n = (n-1) / 2$$

也就是说用数组实现表时，在表中做删除运算，平均要移动表中约一半的元素，因而删除运算所需的平均时间为  $O(n)$ 。

## 1.1.2 解题思路

## 1.1.3 模板代码

```
#include<iostream>
using namespace std;
#define MaxSize 25

typedef int DataType;

class SeqList
{
    DataType list[MaxSize];
    int length;
public:
    SeqList(){length=0;}
    void SLCreat(int n);//创建顺序表
    void SLInsert(int i,DataType x);//在顺序表 L 中的第 i 个位置插入数据元素 x
    void SLDelete(int i);          //在顺序表 L 中的第 i 个位置删除数据元素
    DataType SLGet(int i);        //获取第 i 个位置的元素位置
    DataType SLSum();//求和
    int SLIsEmpty();//判断顺序表是否为空
    void SLPrint();//将顺序表显示在屏幕上
};

//创建顺序表
void SeqList::SLCreat (int n)
{
    DataType x;
    cout<<"请输入数据元素值: ";
    for(int i=0;i<n;i++)
    {
        cin>>x;
        list[i]=x;
        length++;
    }
}

//在顺序表 L 中的 i 位置插入数据元素 x
void SeqList::SLInsert (int i,DataType x)
{
    int k;
    if(length>=MaxSize)
        cout<<"表已满,无法插入!"<<endl;
    else if(i<0||i>length)
        cout<<"参数 i 不合理!"<<endl;
    else
    {
        for(k=length;k>i;k--)
        {
            list[k]=list[k-1];

```

```

    }
    list[i]=list[i-1];
    list[i-1]=x;
    length++;
}
}

//删除第 i 个位置的数据元素
void SeqList::SLDelete (int i)
{
    int k;
    if(!SLIsEmpty())
        cout<<"表已空,无法删除!"<<endl;
    else if(i<0||i>length)
        cout<<"参数 i 不合理!"<<endl;
    else
    {
        for(k=i-1;k<length;k++)
            list[k]=list[k+1];
        length--;
    }
}

```

```

//获取第 i 个位置的元素的数值
DataType SeqList::SLGet (int i)
{
    if (i<0||i>length)
    {
        cout<<"参数 i 不合理!"<<endl;
        return 0;
    }
    else
        return list[i-1];
}

```

```

//判断顺序表是否为空
int SeqList::SLIsEmpty ()
{
    if(length<=0)return 0;
    else return 1;
}

```

```

//将顺序表显示在屏幕上
void SeqList::SLPrint ()
{
    if(!SLIsEmpty())
        cout<<"空表!"<<endl;
    else
        for(int i=0;i<length;i++)
            cout<<list[i]<<" ";
}

```

```

        cout<<endl;
    }

    //求和
    DataType SeqList::SLSum()
    {
        int m=0;
        for(int i=0;i<length;i++)
        {
            m=m+list[i];
        }
        return m;
    }

    int main()
    {
        SeqList mylist;
        int i,n,flag=1,select;
        DataType x;
        cout<<" 1.建立顺序表\n";
        cout<<" 2.求第 i 个位置上的数值\n";
        cout<<" 3.在第 i 个位置上插入数值元素 x\n";
        cout<<" 4.删除第 i 个位置上的数值\n";
        cout<<" 5.该顺序表上各元素之和\n";
        cout<<" 6.输出显示\n";
        cout<<" 7.退出\n";
        cout<<"特别说明:第一次请选择 1,以后就不要选择 1 了!"<<endl;
        while(flag)
        {
            cout<<"请选择: ";
            cin>>select;
            switch(select)
            {
                case 1:
                    cout<<"请输入顺序表长度:";
                    cin>>n;
                    mylist.SLCreat(n);
                    cout<<"顺序表为:  ";
                    mylist.SLPrint();
                    break;
                case 2:
                    cout<<"请输入位置 i: ";
                    cin>>i;
                    cout<<"第"<<i<<"个位置上的数值为: "<<mylist.SLGet (i)<<endl;
                    break;
                case 3:
                    cout<<"请输入要插入元素的位置 i 和数值 x: ";
                    cin>>i>>x;
                    mylist.SLInsert (i,x);
            }
        }
    }
}

```

```

        mylist.SLPrint ();
        break;
    case 4:
        cout<<"请输入要删除的数值的位置: ";
        cin>>i;
        mylist.SLDelete (i);
        cout<<"删除后的顺序表为: ";
        mylist.SLPrint ();
        break;
    case 5:
        cout<<"求和的值: "<<mylist.SLSum()<<endl;
        break;
    case 6:
        cout<<"顺序表为: ";
        mylist.SLPrint ();
        break;
    case 7:
        flag=0;
        break;
    }
}
return 0;
}

```

### 1.1.4 经典题目

题目出处/来源

**Hrbust 1545 基础数据结构——顺序表（2）**

题目描述

在长度为  $n$  ( $n < 1000$ ) 的顺序表中可能存在着一些值相同的“多余”数据元素（类型为整型），编写一个程序将“多余”的数据元素从顺序表中删除，使该表由一个“非纯表”（值相同的元素在表中可能有多个）变成一个“纯表”（值相同的元素在表中只能有一个）。

输入

第一行输入表的长度  $n$ ；

第二行依次输入顺序表初始存放的  $n$  个元素值。

输出

第一行输出完成多余元素删除以后顺序表的元素个数；

第二行依次输出完成删除后的顺序表元素。

示例输入

12

5 2 5 3 3 4 2 5 7 5 4 3

示例输出

5

5 2 3 4 7

代码:

```
#include<stdio.h>

//在顺序表中删除元素。
//参数为顺序表 list，表长 len，要删除元素在 list 中的位置 i。
int del(int list[], int len, int i){
    for(int j = i+1; j < len; j++) {
        list[j-1] = list[j];
    }
    return --len; //返回删除之后的表长。
}

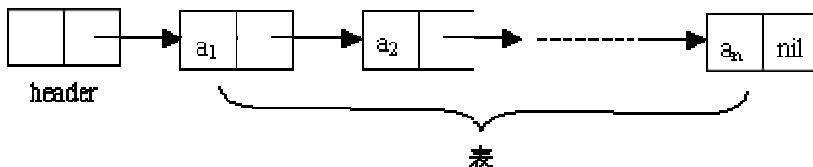
//在顺序表中查找元素。
//参数为顺序表 list，表长 len，要查找的元素 i。
int find(int list[], int len, int i){
    for(int j = 0; j < len; j++) {
        if(i == list[j]) {
            return 1; //查找成功。
        }
    }
    return 0; //查找失败。
}

int main(){
    int n;
    int list[1001];
    while(scanf("%d", &n) != EOF) {
        for(int i = 0; i < n; i++) {
            scanf("%d", &list[i]);
        }
        for(int i = 0, j = 0; i < n; i++) {
            //j 为当前表长，从 0 开始，因为最开始没有加入任何元素。
            int t = find(list, j, list[i]);
            if( !t ) {
                j++; //若没有找到 list[i],则把 list[i]放入 list 中，表长加 1。
            } else {
                //若 list[i]在之前出现过了，则把它删除掉，i 要减 1。
                n = del(list, n, i--);
            }
        }
        printf("%d\n", n);
        for(int i = 0; i < n; i++) {
            printf("%d%c", list[i], i == n-1 ? '\n' : ' ');
        }
    }
    return 0;
}
```

## 1.2 单链表

### 1.2.1 基本原理

在定义的链表中，若只含有一个指针域来存放下一个元素地址，称这样的链表为单链表或线性链表。



单链表可用 C 描述为:

```

struct node
{
    elemtype data; /*元素类型*/
    node *link; /*指针类型,存放下一个元素地址*/
}
    
```

#### 1.2.1.1 链接存储方法

链接方式存储的线性表简称为链表 (Linked List)。

链表的具体存储表示为:

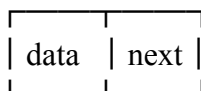
① 用一组任意的存储单元来存放线性表的结点 (这组存储单元既可以是连续的，也可以是不连续的)

② 链表中结点的逻辑次序和物理次序不一定相同。为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其后继结点的地址 (或位置) 信息 (称为指针 (pointer) 或链(link))

注意:

链式存储是最常用的存储方式之一，它不仅可用来表示线性表，而且可用来表示各种非线性的数据结构。

#### 1.2.1.2 链表的结点结构



data 域--存放结点值的数据域

next 域--存放结点的直接后继的地址 (位置) 的指针域 (链域)

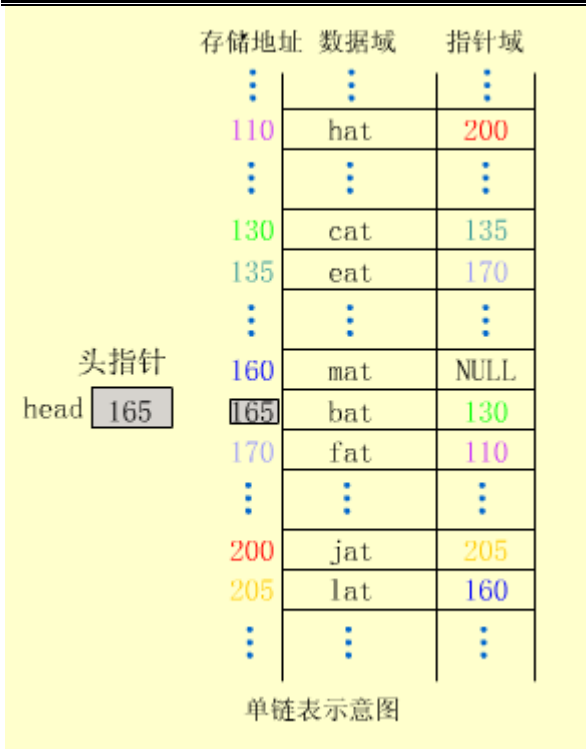
注意:

① 链表通过每个结点的链域将线性表的  $n$  个结点按其逻辑顺序链接在一起的。

② 每个结点只有一个链域的链表称为单链表 (Single Linked List)。

【例】线性表 (bat, cat, eat, fat, hat, jat, lat, mat) 的单链表示如示意图





### 1.2.1.3 头指针 head 和终端结点指针域表示

单链表中每个结点的存储地址是存放在其前趋结点的 next 域中，而开始结点无前趋，故应设头指针 head 指向开始结点。

注意：

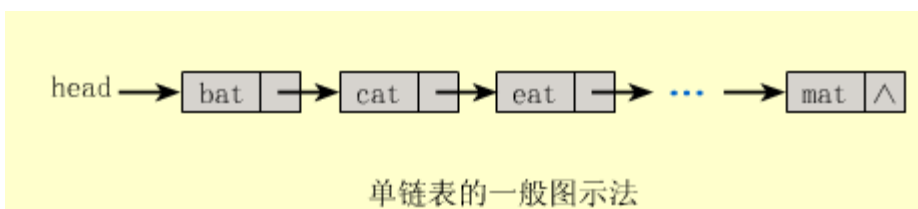
链表由头指针唯一确定，单链表可以用头指针的名字来命名。

【例】头指针名是 head 的链表可称为表 head。

终端结点无后继，故终端结点的指针域为空，即 NULL。

### 1.2.1.4 单链表的一般图示法

由于我们常常只注重结点间的逻辑顺序，不关心每个结点的实际位置，可以用箭头来表示链域中的指针，线性表 (bat, cat, fat, hat, jat, lat, mat) 的单链表就可以表示为下图形式。



### 1.2.1.5 单链表类型描述

```

typedef char DataType; //假设结点的数据域类型为字符
typedef struct node {    //结点类型定义
    DataType data;       //结点的数据域
    struct node *next;   //结点的指针域
} ListNode;
typedef ListNode *LinkList;
    
```

```
ListNode *p;
LinkedList head;
```

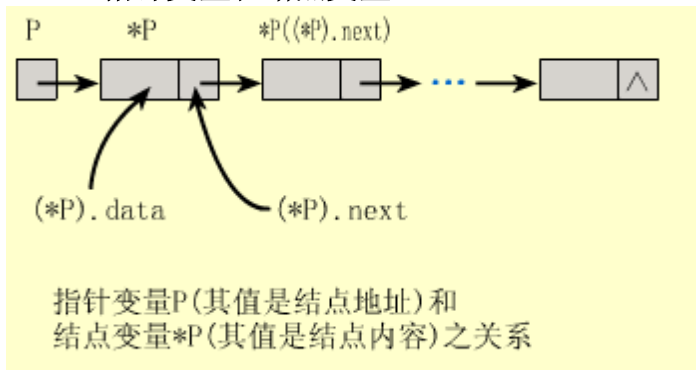
注意：

①LinkedList 和 ListNode \*是不同名字的同一个指针类型（命名的不同是为了概念上更明确）

②LinkedList 类型的指针变量 head 表示它是单链表的头指针

③ListNode \*类型的指针变量 p 表示它是指向某一结点的指针

### 1.2.1.6 指针变量和结点变量



	指针变量	结点变量
定义	在变量说明部分显式定义	在程序执行时，通过标准函数 malloc 生成
取值	非空时，存放某类型结点的地址	实际存放结点各域内容
操作方式	通过指针变量名访问	通过指针生成、访问和释放

①生成结点变量的标准函数

```
p=( ListNode *)malloc(sizeof(ListNode));
```

//函数 malloc 分配一个类型为 ListNode 的结点变量的空间,并将其首地址放入指针变量 p 中

②释放结点变量空间的标准函数

```
free(p); //释放 p 所指的结点变量空间
```

③结点分量的访问

利用结点变量的名字\*p 访问结点分量

方法一: (\*p).data 和(\*p).next

方法二: p->data 和 p->next

④指针变量 p 和结点变量\*p 的关系

指针变量 p 的值——结点地址

结点变量\*p 的值——结点内容

(\*p).data 的值——p 指针所指结点的 data 域的值

(\*p).next 的值——\*p 后继结点的地址

\*((\*p).next)——\*p 后继结点

注意：

① 若指针变量  $p$  的值为空 (NULL)，则它不指向任何结点。此时，若通过  $*p$  来访问结点就意味着访问一个不存在的变量，从而引起程序的错误。

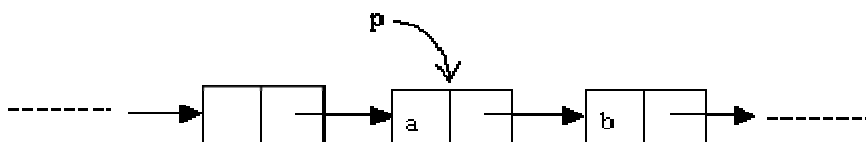
② 有关指针类型的意义和说明方式的详细解释。

### 1.2.1.7 单链表的运算

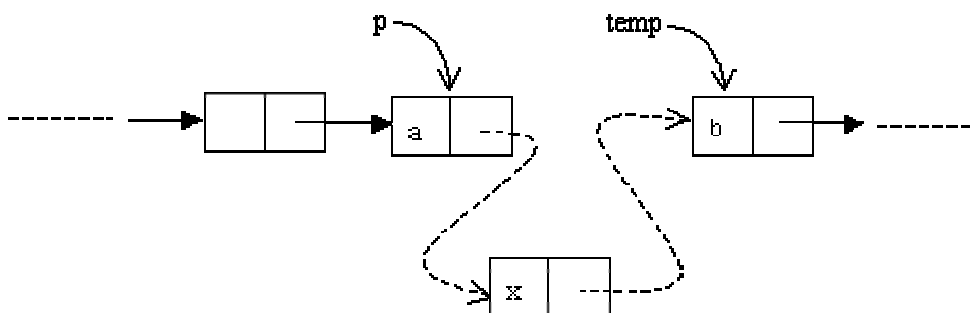
#### 1、建立单链表

假设线性表中结点的数据类型是字符，我们逐个输入这些字符型的结点，并以换行符 '\n' 为输入条件结束标志符。动态地建立单链表的常用方法有如下两种：

插入前：



插入后：



```
int link_ins(NODE **head,int i,elemtype x)
```

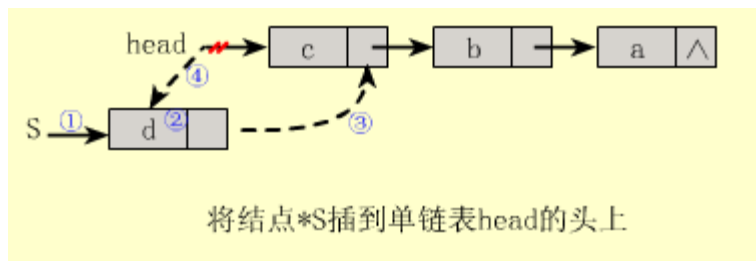
```
{
    int j=1;
    NODE *p,*q;
    q=(NODE *)malloc(sizeof(NODE));
    q->data=x;
    if(i==0){ q->link=*head;
        *head=q;
        return(0);
    }
    p=*head;
    j=0;
    while(++j<i&&p!=NULL)
        p=p->link;
    if(i<0||j<i)
        return(1);
    else
    {
        q->link=p->link;
        p->link=q;
        return(0);
    }
}
```

分析：在上面的插入算法中，不需要移动别的元素，但必须从头开始查找第  $i$  结点的地址，一旦找到插入位置，则插入结点只需两条语句就可完成。该算法的时间复杂度为  $O(n)$ 。

## (1) 头插法建表

### ① 算法思路

从一个空表开始,重复读入数据,生成新结点,将读入数据存放在新结点的数据域中,然后将新结点插入到当前链表的表头上,直到读入结束标志为止。



注意:

该方法生成的链表的结点次序与输入顺序相反。

### ② 具体算法实现

LinkedList CreatListF(void)

{//返回单链表的头指针

char ch;

LinkedList head;//头指针

ListNode \*s; //工作指针

head=NULL; //链表开始为空

ch=getchar(); //读入第 1 个字符

while(ch!='\n'){

s=(ListNode \*)malloc(sizeof(ListNode));//生成新结点

s->data=ch; //将读入的数据放入新结点的数据域中

s->next=head;

head=s;

ch=getchar(); //读入下一字符

}

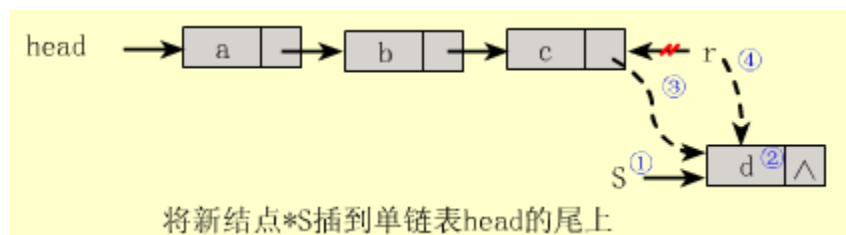
return head;

}

## (2) 尾插法建表

### ① 算法思路

从一个空表开始,重复读入数据,生成新结点,将读入数据存放在新结点的数据域中,然后将新结点插入到当前链表的表尾上,直到读入结束标志为止。



注意:

- 1.采用尾插法建表,生成的链表中结点的次序和输入顺序一致
- 2.必须增加一个尾指针 r,使其始终指向当前链表的尾结点

## ②具体算法实现

LinkedList CreatListR(void)

{//返回单链表的头指针

char ch;

LinkedList head;//头指针

ListNode \*s,\*r; //工作指针

head=NULL; //链表开始为空

r=NULL;//尾指针初值为空

ch=getchar();//读入第 1 个字符

while(ch!='\n'){

s=(ListNode \*)malloc(sizeof(ListNode));//生成新结点

s->data=ch; //将读入的数据放入新结点的数据域中

if (head!=NULL)

head=s;//新结点插入空表

else

r->next=s;//将新结点插到\*r 之后

r=s;//尾指针指向新表尾

ch=getchar(); //读入下一字符

}//endwhile

if (r!=NULL)

r->next=NULL;//对于非空表，将尾结点指针域置空 head=s;

return head;

}

注意：

### 1.开始结点插入的特殊处理

由于开始结点的位置是存放在头指针(指针变量)中,而其余结点的位置是在其前趋结点的指针域中,插入开始结点时要将头指针指向开始结点。

### 2.空表和非空表的不同处理

若读入的第一个字符就是结束标志符,则链表 head 是空表,尾指针 r 亦为空,结点\*r 不存在;否则链表 head 非空,最后一个尾结点\*r 是终端结点,应将其指针域置空。

## (3) 尾插法建带头结点的单链表

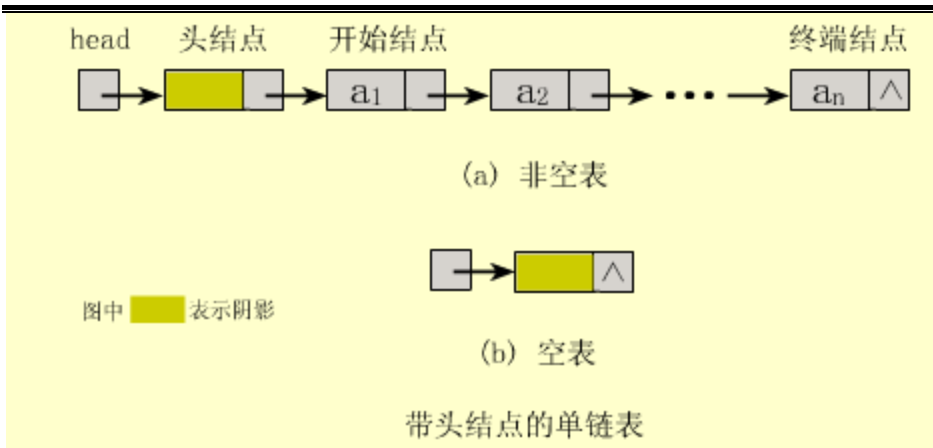
### ①头结点及作用

头结点是在链表的开始结点之前附加一个结点。它具有两个优点:

1.由于开始结点的位置被存放在头结点的指针域中,所以在链表的第一个位置上的操作就和在表的其它位置上操作一致,无须进行特殊处理;

2.无论链表是否为空,其头指针都是指向头结点的非空指针(空表中头结点的指针域空),因此空表和非空表的处理也就统一了。

## ②带头结点的单链表



注意：

头结点数据域的阴影表示该部分不存储信息。在有的应用中可用于存放表长等附加信息。

### ③尾插法建带头结点链表算法

LinkedList CreatListR1(void)

{//用尾插法建立带头结点的单链表

char ch;

LinkedList head=(ListNode \*)malloc(sizeof(ListNode));//生成头结点

ListNode \*s,\*r; //工作指针

r=head; //尾指针初值也指向头结点

while((ch=getchar())!='\n'){

s=(ListNode \*)malloc(sizeof(ListNode));//生成新结点

s->data=ch; //将读入的数据放入新结点的数据域中

r->next=s;

r=s;

}

r->next=NULL;//终端结点的指针域置空，或空表的头结点指针域置空

return head;

}

注意：

上述算法里，动态申请新结点空间时未加错误处理，这对申请空间极少的程序而言不会出问题。但在实用程序里，尤其是对空间需求较大的程序，凡是涉及动态申请空间，一定要加入错误处理以防系统无空间可供分配。

### (4) 算法时间复杂度

以上三个算法的时间复杂度均为  $O(n)$ 。

## 2.单链表的查找运算

### (1) 按序号查找

#### ① 链表不是随机存取结构

在链表中，即使知道被访问结点的序号  $i$ ，也不能像顺序表中那样直接按序号  $i$  访问结点，而只能从链表的头指针出发，顺链域 `next` 逐个结点往下搜索，直至搜索到第  $i$  个结点为止。因此，链表不是随机存取结构。

#### ② 查找的思想方法

计数器  $j$  置为 0 后，扫描指针  $p$  指针从链表的头结点开始顺着链扫描。当  $p$  扫描下一

个结点时，计数器  $j$  相应地加 1。当  $j=i$  时，指针  $p$  所指的结点就是要找的第  $i$  个结点。而当  $p$  指针指为 `null` 且  $j \neq i$  时，则表示找不到第  $i$  个结点。

注意：

头结点可看做是第 0 个结点。

### ③具体算法实现

```
ListNode* GetNode(LinkList head,int i)
{//在带头结点的单链表 head 中查找第 i 个结点，若找到 ( $0 \leq i \leq n$ ),
//则返回该结点的存储位置，否则返回 NULL。
int j;
ListNode *p;
p=head;j=0;//从头结点开始扫描
while(p->next&& j<i){//顺指针向后扫描，直到 p->next 为 NULL 或 i=j 为止
    p=p->next;
    j++;
}
if(i==j)
    return p;//找到了第 i 个结点
else return NULL;//当 i<0 或 i>0 时，找不到第 i 个结点
}
```

### ④算法分析

算法中，`while` 语句的终止条件是搜索到表尾或者满足  $j \geq i$ ，其频度最多为  $i$ ，它和被寻找的位置有关。在等概率假设下，平均时间复杂度为：

$$\sum_{i=1}^n i / (n+1) = 1/(n+1) * \sum_{i=1}^n i = n/2 = O(n)$$

## (2) 按值查找

### ①思想方法

从开始结点出发，顺着链逐个将结点的值和给定值 `key` 作比较，若有结点的值与 `key` 相等，则返回首次找到的其值为 `key` 的结点的存储位置；否则返回 `NULL`。

### ②具体算法实现

```
ListNode* LocateNode (LinkList head,DataType key)
{//在带头结点的单链表 head 中查找其值为 key 的结点
    ListNode *p=head->next;//从开始结点比较。表非空，p 初始值指向开始结点
    while(p&& p->data!=key)//直到 p 为 NULL 或 p->data 为 key 为止
        p=p->next;//扫描下一结点
    return p;//若 p=NULL，则查找失败，否则 p 指向值为 key 的结点
}
```

### ③算法分析

该算法的执行时间亦与输入实例中 `key` 的取值相关，其平均时间复杂度分析类似于按序号查找，为  $O(n)$ 。

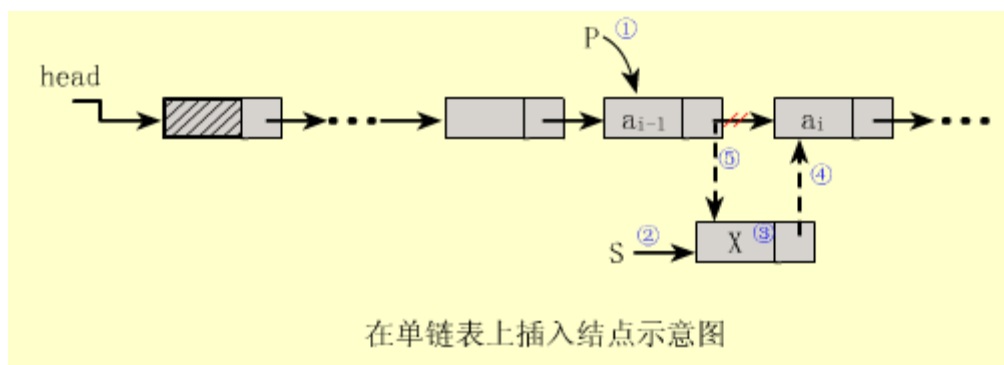
## 3.插入运算

### (1) 思想方法

插入运算是将值为  $x$  的新结点插入到表的第  $i$  个结点的位置上，即插入到  $a_{i-1}$  与  $a_i$  之间。

具体步骤：

- (1) 找到  $a_{i-1}$  存储位置  $p$
- (2) 生成一个数据域为  $x$  的新结点  $s$
- (3) 令结点  $p$  的指针域指向新结点
- (4) 新结点的指针域指向结点  $a_i$ 。



### (2) 具体算法实现

```
void InsertList(LinkList head,DataType x,int i)
{//将值为 x 的新结点插入到带头结点的单链表 head 的第 i 个结点的位置上
    ListNode *p;
    p=GetNode(head,i-1);//寻找第 i-1 个结点
    if (p==NULL)//i<1 或 i>n+1 时插入位置 i 有错
        Error("position error");
    s=(ListNode *)malloc(sizeof(ListNode));
    s->data=x;s->next=p->next;p->next=s;
}
```

### (3) 算法分析

算法的时间主要耗费在查找操作 `GetNode` 上，故时间复杂度亦为  $O(n)$ 。

## 4. 删除运算

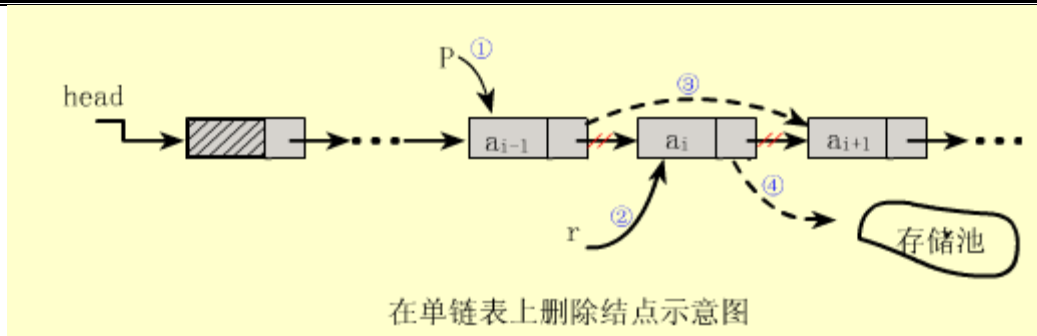
### (1) 思想方法

删除运算是将表的第  $i$  个结点删去。

具体步骤：

- (1) 找到  $a_{i-1}$  的存储位置  $p$ （因为在单链表中结点  $a_i$  的存储地址是在其直接前趋结点  $a_{i-1}$  的指针域 `next` 中）
- (2) 令  $p \rightarrow next$  指向  $a_i$  的直接后继结点（即把  $a_i$  从链上摘下）
- (3) 释放结点  $a_i$  的空间，将其归还给“存储池”。





## (2) 具体算法实现

```
void DeleteList(LinkList head, int i)
{
    // 删除带头结点的单链表 head 上的第 i 个结点
    ListNode *p, *r;
    p = GetNode(head, i-1); // 找到第 i-1 个结点
    if (p == NULL || p->next == NULL) // i < 1 或 i > n 时, 删除位置错
        Error("position error"); // 退出程序运行
    r = p->next; // 使 r 指向被删除的结点 a_i
    p->next = r->next; // 将 a_i 从链上摘下
    free(r); // 释放结点 a_i 的空间给存储池
}
```

注意:

设单链表的长度为  $n$ , 则删去第  $i$  个结点仅当  $1 \leq i \leq n$  时是合法的。

当  $i = n+1$  时, 虽然被删结点不存在, 但其前趋结点却存在, 它是终端结点。因此被删结点的直接前趋  $*p$  存在并不意味着被删结点就一定存在, 仅当  $*p$  存在 (即  $p \neq \text{NULL}$ ) 且  $*p$  不是终端结点 (即  $p->\text{next} \neq \text{NULL}$ ) 时, 才能确定被删结点存在。

## (3) 算法分析

算法的时间复杂度也是  $O(n)$ 。

链表上实现的插入和删除运算, 无须移动结点, 仅需修改指针。

## 1.2.2 解题思路

## 1.2.3 模板代码

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

// 联系人节点结构体
typedef struct _LinkNode
{
    char name[9];           // 姓名
    char phone[14];         // 电话
    struct _LinkNode* next; // 下一个联系人指针
} LinkNode;
```

```
//初始化操作
LinkNode* InitList()
{
    LinkNode* head = NULL;

    head = (LinkNode*)malloc(sizeof(LinkNode));
    if(head == NULL)
    {
        printf("内存分配失败\n");
        return NULL;
    }

    //初始化 将其指向下一节点指针置空
    memset(head, 0, sizeof(LinkNode));
    head->next = NULL;

    return head;
}

//取元素操作, i 为需要取的元素排序
LinkNode* GetElem(LinkNode* ls, int i)
{
    LinkNode* temp = NULL;
    int j = 0;

    if(ls == NULL)
    {
        printf("头结点不存在\n");
        return NULL;
    }

    if(i < 1)
    {
        printf("参数输入错误\n");
        return NULL;
    }

    //temp 指向第一个元素
    temp = ls->next;
    j = 1;

    //循环找到第 i 个元素
    while(temp != NULL && j < i)
    {
        temp = temp->next;
        j++;
    }

    //第 i 个元素为空 或者 元素数量少于 i
    if(temp == NULL || j != i)
```

```
{
    printf("元素不存在\n");
    return NULL;
}

return temp;
}

//插入新元素到第 i 个位置之前
void InsertElem(LinkNode* ls, int i, LinkNode* elem)
{
    LinkNode* temp = NULL;
    LinkNode* newnode = NULL;
    int j = 0;

    if(ls == NULL)
    {
        printf("线性表不存在\n");
        return ;
    }

    if(i < 1)
    {
        printf("参数输入错误\n");
        return NULL;
    }

    if(elem == NULL)
    {
        printf("待插入元素不存在");
        return ;
    }

    temp = ls;
    j = 1;

    //寻找第 i-1 个节点
    while(temp != NULL && j < i)
    {
        temp = temp->next;
        j++;
    }

    if(j < i - 1)
    {
        printf("第%d 个元素不存在\n", i - 1);
        return ;
    }

    //为新节点分配内存
```

```

newnode = (LinkNode*)malloc(sizeof(LinkNode));
if(newnode == NULL)
{
    printf("内存分配失败\n");
    return ;
}

memcpy(newnode, elem, sizeof(LinkNode));

newnode->next = temp->next;
temp->next = newnode;
}

//销毁链表
void DestroyList(LinkNode* ls)
{
    LinkNode* temp = NULL;
    LinkNode* next = NULL;

    if(ls == NULL)
    {
        printf("链表不存在\n");
        return ;
    }

    temp = ls;
    while(temp != NULL)
    {
        next = temp->next;
        free(temp);
        temp = next;
    }
}

void PrintNode(LinkNode* elem)
{
    if(elem == NULL)
        return ;

    printf("%8s    %13s\n", elem->name, elem->phone);
}

//遍历输出链表
void TravList(LinkNode* ls)
{
    LinkNode* temp = NULL;

    if(ls == NULL)
    {
        printf("链表不存在\n");
    }
}

```

```

        return ;
    }

    temp = ls->next;

    while(temp != NULL)
    {
        PrintNode(temp);
        temp = temp->next;
    }
}
//删除元素
LinkNode DeleteElem(LinkNode* ls, int i)
{
    LinkNode ret = {"", "", NULL};
    LinkNode* temp = NULL;
    LinkNode* del = NULL;
    int j = 0;

    if(ls == NULL)
    {
        printf("链表不存在\n");
        return ret;
    }

    if(i < 1)
    {
        printf("输入参数错误\n");
        return ret;
    }

    temp = ls;
    j = 1;

    while(temp != NULL && j < i)
    {
        temp = temp->next;
        j++;
    }

    if(temp == NULL || j != i)
    {
        printf("第%d 个元素不存在\n", i - 1);
        return ret;
    }

    if(temp->next == NULL)
    {
        printf("第%d 个元素不存在\n", i);
        return ret;
    }
}

```

//修改 i-1 元素的指针 指向 i 的下一位置, 拷贝第 i 个元素信息到 ret 中, 然后说删除第 i 个元素

```

    del = temp->next;
    temp->next = del->next;
    memcpy(&ret, temp->next, sizeof(LinkNode));
    free(del);

    return ret;
}
//修改第 i 个元素
void ModifyElem(LinkNode* ls, int i, LinkNode* mod)
{
    LinkNode* temp = NULL;
    int j = 0;

    if(ls == NULL || mod == NULL || i < 1)
    {
        printf("参数无效\n");
        return ;
    }

    temp = ls->next;
    j = 1;

    while(temp != NULL && j < i)
    {
        temp = temp->next;
        j++;
    }

    if(temp == NULL || j != i)
    {
        printf("第%d 个元素不存在\n", i);
        return ;
    }

    strcpy(temp->name, mod->name);
    strcpy(temp->phone, mod->phone);
}

int main()
{
    LinkNode* head = NULL;
    int i = 0;
    LinkNode temp;
    LinkNode* result;

    //初始化链表
    head = InitList();

```

```

if(head == NULL)
    return 1;

//链表操作
for(i = 0; i < 20; i++)
{
    sprintf(temp.name, "stu%03d", i + 1);
    sprintf(temp.phone, "1311122%04d", i + 10);
    InsertElem(head, i + 1, &temp);
}

TravList(head);

//查找操作验证 查找元素
printf("\n 分别查找第 1,20,10,21 个元素\n");

result = GetElem(head, 1);
if(result != NULL)
{
    printf("%8s    %13s\n", result->name, result->phone);
}
result = GetElem(head, 20);
if(result != NULL)
{
    printf("%8s    %13s\n", result->name, result->phone);
}
result = GetElem(head, 10);
if(result != NULL)
{
    printf("%8s    %13s\n", result->name, result->phone);
}
result = GetElem(head, 21);
if(result != NULL)
{
    printf("%8s    %13s\n", result->name, result->phone);
}
result = GetElem(head, -1);
if(result != NULL)
{
    printf("%8s    %13s\n", result->name, result->phone);
}

//删除元素验证
printf("\n 删除元素验证\n");
DeleteElem(head, 1);
DeleteElem(head, 10);
DeleteElem(head, 20);
DeleteElem(head, 21);

TravList(head);
    
```

```

//修改元素验证
printf("\n 修改元素验证\n");
strcpy(temp.name, "mod");
strcpy(temp.phone, "123");

ModifyElem(head, 0, &temp);
ModifyElem(head, 1, &temp);
ModifyElem(head, 10, &temp);
ModifyElem(head, 18, &temp);
ModifyElem(head, 21, &temp);

travList(head);

//销毁链表
DestroyList(head);

return 0;
}
    
```

## 1.2.4 经典题目

### 1.2.4.1 题目 1

题目出处/来源

**Hrbust 1546 基础数据结构——单链表（1）**

题目描述

输入  $n$  个整数，先按照数据输入的顺序建立一个带头结点的单链表，再输入一个数据  $m$ ，将单链表中的值为  $m$  的结点全部删除。分别输出建立的初始单链表和完成删除后的单链表。

输入

第一行输入数据个数  $n$ ；  
第二行依次输入  $n$  个整数；  
第三行输入欲删除数据  $m$ 。

输出

第一行输出原始单链表的长度；  
第二行依次输出原始单链表的数据；  
第三行输出完成删除后的单链表长度；  
第四行依次输出完成删除后的单链表数据。

样例输入

```

10
56 25 12 33 66 54 7 12 33 12
12
    
```

样例输出

```

10
56 25 12 33 66 54 7 12 33 12
7
56 25 33 66 54 7 33
    
```

代码

```
#include<stdio.h>
```



```

#include<stdlib.h>
struct node{
    int data;
    struct node *next;
};
//删除操作，参数为链表头指针，删除元素的位置 i。
void ListDel(struct node *head, int i){
    int j;
    struct node *p, *q;
    p = head;
    j = 1;
    while(p->next && j < i-1) { //遍历寻找第 i-1 个元素。
        p = p->next;
        ++j;
    }
    q = p->next;
    p->next = q->next; //将 q 的后继赋给 p 的后继。
    free(q); //释放内存。
}

//清空链表，释放空间。
void ClearList(struct node *head){
    struct node *p, *q;
    p = head->next; //p 指向第一个节点。
    while(p) {
        q = p->next;
        free(p);
        p = q;
    }
    head->next = NULL; //头节点指针域为空。
}

int main(){
    int n;
    while(scanf("%d", &n) != EOF) {
        struct node *head = NULL; //定义头指针，初始化为空。
        struct node *p, *q; //指向节点的指针 p, q。
        //创建单链表。
        p = (struct node *)malloc(sizeof(struct node));
        scanf("%d", &p->data);
        head = p;

        for(int i = 1; i < n; i++) {
            q = (struct node *)malloc(sizeof(struct node));
            scanf("%d", &q->data);
            p->next = q;
            p = q;
        }
        p->next = NULL;
        //创建单链表结束。
        int m; //输入要删除的数据 m。
    }
}

```

```
scanf("%d", &m);
```

//遍历链表中的数据并输出。

```
struct node *point = head;
printf("%d\n%d", n, point->data);
point = point->next;
while(point) {
    printf(" %d", point->data);
    point = point->next;
}
printf("\n");
```

//删除链表中为 m 的元素。

```
while(true) {
    struct node *pos = head;
    int tot = 1;
    while(pos) {
        if(pos->data == m) {
            if(tot == 1) {
                head = pos->next;
                free(pos);
                n--;
                break;
            }
            ListDel(head, tot);
            n--;
            break;
        }
        pos = pos->next;
        tot++;
    }
    if( pos == NULL ) {
        break;
    }
}
```

//遍历删除数据后的链表，并将其中的元素输出。

```
struct node *pos = head;
if(n != 0) {
    printf("%d\n%d", n, pos->data);
    pos = pos->next;
    while(pos) {
        printf(" %d", pos->data);
        pos = pos->next;
    }
    printf("\n");
} else {
    printf("0\n");
}
ClearList(head);
}
```

```
return 0;
}
```

### 1.2.4.2 题目 2

题目出处/来源

#### Hrbust 1547 基础数据结构——单链表（2）

题目描述

1997-1998 年欧洲西南亚洲区预赛之后，举办了一场隆重的聚会。主办方发明了一个特殊的方式去挑选那些自愿去洗脏碟子的参赛选手。先让那些选手一个挨着一个的排成一条队。每个选手都获得一个编号，那些编号是从 2 开始的，第一个的编号是 2，第二个人的编号是 3，第三个人的编号是 4，以此类推。

第一个选手将被问到他的编号（编号为 2）。他将不用去清洗了，直接参加聚会，但是他身后的所站的位置是 2 的倍数的人必须去厨房（那些人的编号分别为 4，6，8 等等）。然后在那队伍中的下一个选手必须报数。他回答 3，他可以离开去参加聚会，但是在他身后的每个是三的倍数的选手将会被选上（那些人的编号分别为 9,15,21 等等）。下一个被选上的人的编号是 5，并且将可以离开去参加聚会，但是在他身后并且站的位置是 5 的倍数的人将会被选上去清洗碟子（那些人的编号分别为 19,35,49 等等）。下一个被选上的人的编号是 7，并且将可以离开去参加聚会，但是在他身后并且站的位置是 7 的倍数的人将会被选上去清洗碟子，以此类推。

让我们称那些没有被选上去洗碟子的那些选手的编号为幸运数字。继续这个挑选的方式，那些幸运的数字是 2，3，5，7，11，13，17 等等的递增序列。为下一次的聚会寻找幸运数字！

输入

本题有多组测试数据，每组测试数据包含一个整数  $n$ ， $1 \leq n \leq 3000$ 。

输出

对于每组测试数据输出一个数字，代表对应的幸运号码。

样例输入

```
1
2
10
20
```

样例输出

```
2
3
29
83
```

代码

```
#include<stdio.h>
#include<stdlib.h>
struct node{
```

```

int data;
struct node *next;
};
int main(){
    struct node *head = NULL;
    struct node *p, *q;
    //创建单链表。
    p = (struct node *)malloc(sizeof(struct node));
    p->data = 0;
    head = p;
    for(int i = 2; i <= 35000; i++) {
        q = (struct node *)malloc(sizeof(struct node));
        q->data = i;
        p->next = q;
        p = q;
    }
    p->next = NULL;
    //创建完毕。
    struct node *point = head;
    struct node *top;
    int ans = 0;
    //遍历链表。
    while(point->next != NULL) {
        int tmp = point->next->data;
        top = point->next;
        while(1) {
            for(int i = 0; i < tmp-1; i++) {
                if(top->next == NULL) {
                    ans = 1;
                    break;
                }
                top = top->next;
            }
            if(ans) {
                ans = 0;
                break;
            } else {
                if(top->next == NULL) {
                    break;
                } else {
                    //删除操作。
                    q = top->next;
                    top->next = q->next;
                    free(q);
                }
            }
        }
        point = point->next;
    }
    int n;
    while(scanf("%d", &n) != EOF) {

```

```

    struct node *point = head;
    //查找对应的幸运号码。
    for(int i = 0; i < n; i++) {
        point = point->next;
    }
    printf("%d\n", point->data);
}
return 0;
}

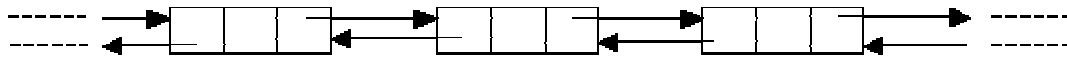
```

## 1.2.5 扩展变形

## 1.3 双向链表

### 1.3.1 基本原理

在单链表中，从某个结点出发可以直接找到它的直接后继，时间复杂度为  $O(1)$ ，但无法直接找到它的直接前驱；在单循环链表中，从某个结点出发可以直接找到它的直接后继，时间复杂仍为  $O(1)$ ，直接找到它的直接前驱，时间复杂为  $O(n)$ 。有时，希望能快速找到一个结点的直接前驱，这时，可以在单链表中的结点中增加一个指针域指向它的直接前驱，这样的链表，就称为双向链表(一个结点中含有两个指针)。如果每条链构成一个循环链表，则会得双向循环链表。



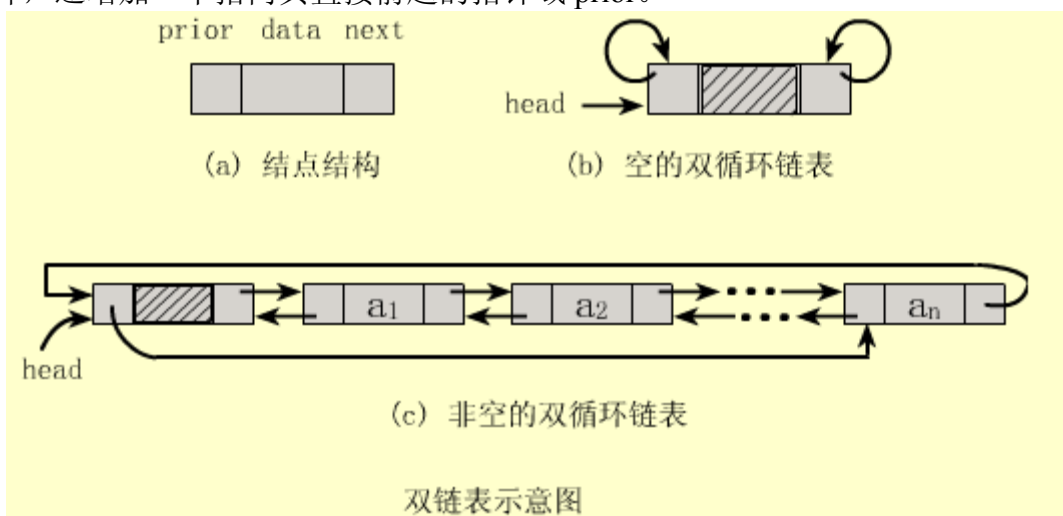
双向链表可用 C 描述如下：

```

struct node
{
    elemtype data; /*结点的数据域,类型设定为 elemtype*/
    node *link1,*link2; /*定义指向直接后继和直接前驱的指针*/
}

```

双（向）链表中有两条方向不同的链，即每个结点中除 `next` 域存放后继结点地址外，还增加一个指向其直接前趋的指针域 `prior`。



注意：

①双链表由头指针 `head` 惟一确定的。

- ②带头结点的双链表的某些运算变得方便。
- ③将头结点和尾结点链接起来，为双（向）循环链表。

### 1.3.1.1 双向链表的结点结构和形式描述

①结点结构(见上图 a)

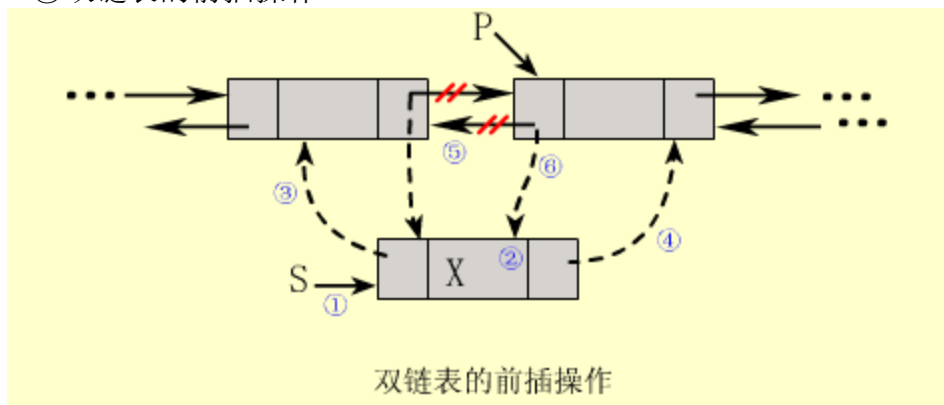
②形式描述

```
typedef struct dlistnode{
    DataType data;
    struct dlistnode *prior,*next;
}DListNode;
typedef DListNode *DLinkedList;
DLinkedList head;
```

### 1.3.1.2 双向链表的前插和删除本结点操作

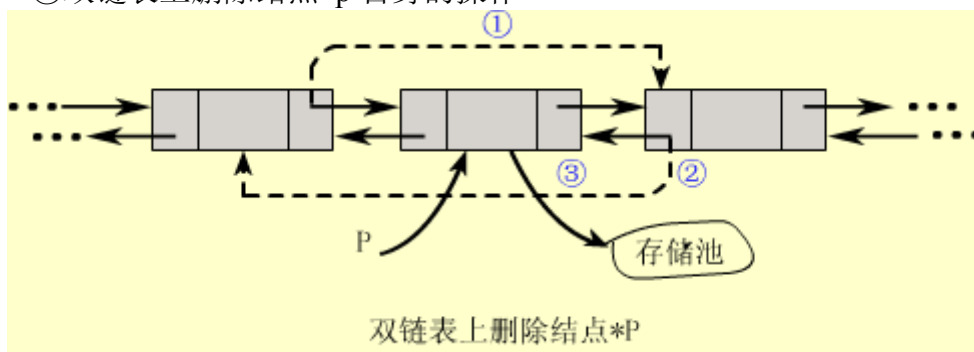
由于双链表的对称性，在双链表能方便地完成各种插入、删除操作。

①双链表的前插操作



```
void DInsertBefore(DListNode *p,DataType x)
{//在带头结点的双链表中，将值为 x 的新结点插入 *p 之前，设 p≠NULL
    DListNode *s=malloc(sizeof(DListNode));//①
    s->data=x;//②
    s->prior=p->prior;//③
    s->next=p;//④
    p->prior->next=s;//⑤
    p->prior=s;//⑥
}
```

②双链表上删除结点\*p 自身的操作



```
void DDeleteNode(DListNode *p)
```

```

{ //在带头结点的双链表中，删除结点*p，设*p 为非终端结点
  p->prior->next=p->next; //①
  p->next->prior=p->prior; //②
  free(p); //③
}

```

注意：

与单链表上的插入和删除操作不同的是，在双链表中插入和删除必须同时修改两个方向上的指针。

上述两个算法的时间复杂度均为  $O(1)$ 。

### 1.3.2 解题思路

### 1.3.3 模板代码

```

typedef struct DuLNode
{
    //自己定义数据类型。
    ElemType data;
    struct DuLNode *prior,*next;
}DuLNode,*DuLinkList;
// 带头结点的双向循环链表的基本操作
void InitList(DuLinkList *L)
{ /* 产生空的双向循环链表 L */
    *L=(DuLinkList)malloc(sizeof(DuLNode));
    if(*L)
        (*L)->next=(*L)->prior=*L;
    else
        exit(OVERFLOW);
}
void DestroyList(DuLinkList *L)
    // 销毁双向循环链表 L
{
    DuLinkList q,p=(*L)->next; /* p 指向第一个结点 */
    while(p!=*L) /* p 没到表头 */
    {
        q=p->next;
        free(p);
        p=q;
    }
    free(*L);
    *L=NULL;
}
// 重置链表为空表
void ClearList(DuLinkList L) /* 不改变 L */
{ DuLinkList q,p=L->next; /* p 指向第一个结点 */
    while(p!=L) /* p 没到表头 */
    {
        q=p->next;
        free(p);
        p=q;
    }
}

```

```

    }
    L->next=L->prior=L; /* 头结点的两个指针域均指向自身 */
}
// 验证是否为空表
Status ListEmpty(DuLinkedList L)
{ /* 初始条件: 线性表 L 已存在 */
    if(L->next==L&&L->prior==L)
        return TRUE;
    else
        return FALSE;
}
// 计算表内元素个数
int ListLength(DuLinkedList L)
{ /* 初始条件: L 已存在。操作结果:  */
    int i=0;
    DuLinkedList p=L->next; /* p 指向第一个结点 */
    while(p!=L) /* p 没到表头 */
    {
        i++;
        p=p->next;
    }
    return i;
}
// 赋值
Status GetElem(DuLinkedList L,int i,ElemType *e)
{ /* 当第 i 个元素存在时, 其值赋给 e 并返回 OK, 否则返回 ERROR */
    int j=1; /* j 为计数器 */
    DuLinkedList p=L->next; /* p 指向第一个结点 */
    while(p!=L&&j<=i) {
        j++;
    }
    if(p==L||j>i) /* 第 i 个元素不存在 */
        return ERROR;
    *e=p->data; /* 取第 i 个元素 */
    return OK;
}
// 查找元素
int LocateElem(DuLinkedList L,ElemType e,Status(*compare)(ElemType,ElemType))
{ /* 初始条件: L 已存在, compare()是数据元素判定函数 */
    /* 操作结果: 返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。 */
    /* 若这样的数据元素不存在, 则返回值为 0 */
    int i=0;
    DuLinkedList p=L->next; /* p 指向第 1 个元素 */
    while(p!=L)
    {
        i++;
        if(compare(p->data,e)) /* 找到这样的数据元素 */
            return i;
        p=p->next;
    }
}

```



```

    }
    return 0;
}
// 查找元素前驱
Status PriorElem(DuLinkedList L, ElemType cur_e, ElemType *pre_e)
{ /* 操作结果: 若 cur_e 是 L 的数据元素, 且不是第一个, 则用 pre_e 返回它的前
驱, */
    /* 否则操作失败, pre_e 无定义 */
    DuLinkedList p=L->next->next; /* p 指向第 2 个元素 */
    while(p!=L) /* p 没到表头 */
    {
        if(p->data==cur_e)
        {
            *pre_e=p->prior->data;
            return TRUE;
        }
        p=p->next;
    }
    return FALSE;
}
// 查找元素后继
Status NextElem(DuLinkedList L, ElemType cur_e, ElemType *next_e)
{ /* 操作结果: 若 cur_e 是 L 的数据元素, 且不是最后一个, 则用 next_e 返回它的后
继, */
    /* 否则操作失败, next_e 无定义 */
    DuLinkedList p=L->next->next; /* p 指向第 2 个元素 */
    while(p!=L) /* p 没到表头 */
    {
        if(p->prior->data==cur_e)
        {
            *next_e=p->data;
            return TRUE;
        }
        p=p->next;
    }
    return FALSE;
}
// 查找元素地址
DuLinkedList GetElemP(DuLinkedList L, int i) /* 另加 */
{ /* 在双向链表 L 中返回第 i 个元素的地址。i 为 0, 返回头结点的地址。若第 i 个元
素不存在, */
    /* 返回 NULL */
    int j;
    DuLinkedList p=L; /* p 指向头结点 */
    if(i<0||i>ListLength(L)) /* i 值不合法 */
        return NULL;
    for(j=1; j<=i; j++)
        p=p->next;
    return p;
}

```

```

    }
    // 元素的插入
    Status ListInsert(DuLinkList L,int i,ElemType e)
    { /* 在带头结点的双链循环线性表 L 中第 i 个位置之前插入元素 e, i 的合法值为  $1 \leq i \leq \text{表长} + 1$  */
        /* 改进算法 2.18, 否则无法在第表长+1 个结点之前插入元素 */
        DuLinkList p,s;
        if(i<1||i>ListLength(L)+1) /* i 值不合法 */
            return ERROR;
        p=GetElemP(L,i-1); /* 在 L 中确定第 i 个元素前驱的位置指针 p */
        if(!p) /* p=NULL,即第 i 个元素的前驱不存在(设头结点为第 1 个元素的前驱) */
            return ERROR;
        s=(DuLinkList)malloc(sizeof(DuLNode));
        if(!s)
            return OVERFLOW;
        s->data=e;
        s->prior=p; /* 在第 i-1 个元素之后插入 */
        s->next=p->next;
        p->next->prior=s;
        p->next=s;
        return OK;
    }
    // 元素的删除
    Status ListDelete(DuLinkList L,int i,ElemType *e)
    { /* 删除带头结点的双链循环线性表 L 的第 i 个元素, i 的合法值为  $1 \leq i \leq \text{表长}$  */
        DuLinkList p;
        if(i<1) /* i 值不合法 */
            return ERROR;
        p=GetElemP(L,i); /* 在 L 中确定第 i 个元素的位置指针 p */
        if(!p) /* p=NULL,即第 i 个元素不存在 */
            return ERROR;
        *e=p->data;
        p->prior->next=p->next;
        p->next->prior=p->prior;
        free(p);
        return OK;
    }
    // 正序查找
    void ListTraverse(DuLinkList L,void(*visit)(ElemType))
    { /* 由双链循环线性表 L 的头结点出发, 正序对每个数据元素调用函数 visit() */
        DuLinkList p=L->next; /* p 指向头结点 */
        while(p!=L)
        {
            visit(p->data);
            p=p->next;
        }
        printf("\n");
    }
    void ListTraverseBack(DuLinkList L,void(*visit)(ElemType))

```

```

// 逆序查找
{ /* 由双链循环线性表 L 的头结点出发，逆序对每个数据元素调用函数 visit()。另加
*/
    DuLinkList p=L->prior; /* p 指向尾结点 */
    while(p!=L)
    {
        visit(p->data);
        p=p->prior;
    }
    printf("\n");
}
    
```

### 1.3.4 经典题目

### 1.3.5 扩展变形

## 1.4 循环链表

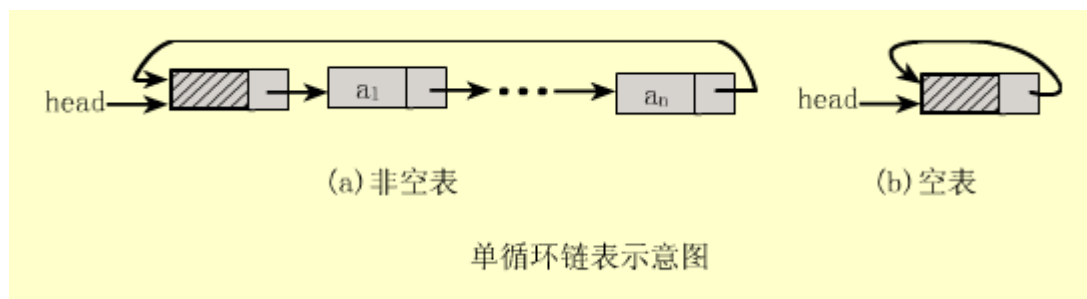
### 1.4.1 基本原理

循环链表是一种首尾相接的链表。

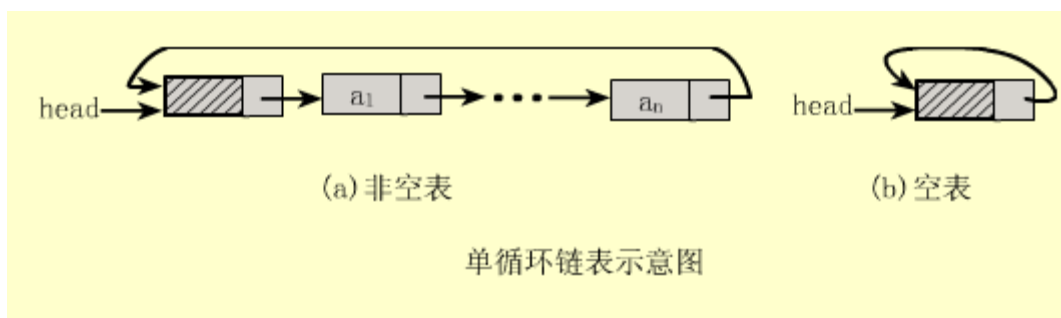
#### 1、循环链表

(1) 单循环链表——在单链表中，将终端结点的指针域 NULL 改为指向表头结点或开始结点即可。

(2) 多重链的循环链表——将表中结点链在多个环上。



#### 2、带头结点的单循环链表

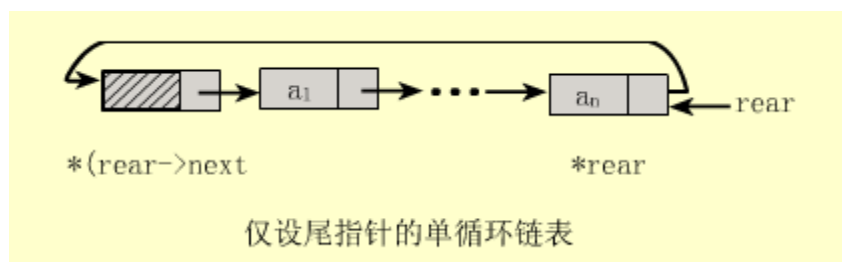


注意：

判断空链表的条件是 `head==head->next;`

### 3、仅设尾指针的单循环链表

用尾指针  $rear$  表示的单循环链表对开始结点  $a_1$  和终端结点  $a_n$  查找时间都是  $O(1)$ 。而表的操作常常是在表的首尾位置上进行，因此，实用中多采用尾指针表示单循环链表。带尾指针的单循环链表可见下图。



注意：

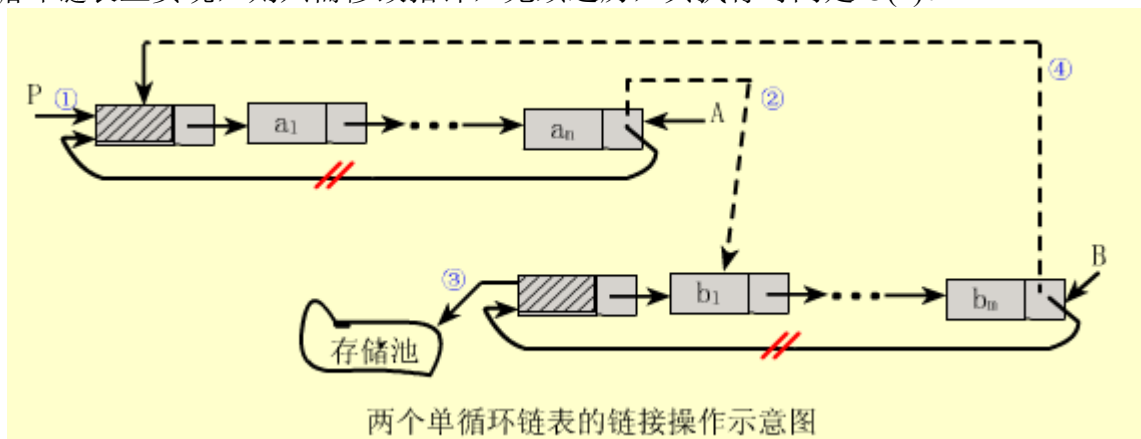
判断空链表的条件为  $rear == rear \rightarrow next$ ;

### 4、循环链表的特点

循环链表的特点是无须增加存储量，仅对表的链接方式稍作改变，即可使得表处理更加方便灵活。

【例】在链表上实现将两个线性表  $(a_1, a_2, \dots, a_n)$  和  $(b_1, b_2, \dots, b_m)$  连接成一个线性表  $(a_1, \dots, a_n, b_1, \dots, b_m)$  的运算。

分析：若在单链表或头指针表示的单循环表上做这种链接操作，都需要遍历第一个链表，找到结点  $a_n$ ，然后将结点  $b_1$  链到  $a_n$  的后面，其执行时间是  $O(n)$ 。若在尾指针表示的单循环链表上实现，则只需修改指针，无须遍历，其执行时间是  $O(1)$ 。



相应的算法如下：

```

LinkedList Connect(LinkedList A, LinkedList B)
{
    //假设 A, B 为非空循环链表的尾指针
    LinkedList p = A->next; //①保存 A 表的头结点位置
    A->next = B->next->next; //②B 表的开始结点链接到 A 表尾
    free(B->next); //③释放 B 表的头结点
    B->next = p; //④
    return B; //返回新循环链表的尾指针
}
    
```

注意：

①循环链表中没有 NULL 指针。涉及遍历操作时，其终止条件就不再是像非循环链表那样判别  $p$  或  $p \rightarrow next$  是否为空，而是判别它们是否等于某一指定指针，如头指针或尾指

针等。

②在单链表中，从一已知结点出发，只能访问到该结点及其后续结点，无法找到该结点之前的其它结点。而在单循环链表中，从任一结点出发都可访问到表中所有结点，这一优点使某些运算在单循环链表上易于实现。

## 1.4.2 解题思路

## 1.4.3 模板代码

```
#include <stdio.h>
#include <stdlib.h>
typedef struct CLNode
{
    int data;
    struct CLNode* next;
}CLNode,*CLinkList;

void InitCL(CLinkList CL)
{
    if(CL!=NULL)
        CL->next=CL;
    else
        CL=(CLinkList)malloc(sizeof(CLNode));
}

void InsertFront(CLinkList CL,int value)
{
    CLinkList p;
    p=(CLinkList)malloc(sizeof(CLNode));
    p->data=value;
    p->next=CL->next;
    CL->next=p;
}

void InsertEnd(CLinkList CL,int value)
{
    CLinkList p,q;
    p=(CLinkList)malloc(sizeof(CLNode));
    p->data=value;
    q=CL;
    while(q->next!=CL)
    {
        q=q->next;
    }
    p->next=q->next;
    q->next=p;
}

void Print(CLinkList CL)
{
    CLinkList p;
```

```

        p=CL->next;
        while(p!=CL)
        {
            printf("%d\t",p->data);
            p=p->next;
        }
        printf("\n");
    }

void DeleteFront(CLinkList CL,int *val)
{
    CLinkList p;
    p=CL->next;
    if(p!=CL)
    {
        CL->next=p->next;
        *val=p->data;
    }
    else
        printf("LinkList is empty!\n");
}

void DeleteEnd(CLinkList CL,int *val)
{
    CLinkList p,q;

    q=CL;
    p=CL->next;
    if(p!=CL)
    {
        while(p->next!=CL)
        {
            q=p;
            p=p->next;
        }
        *val=p->data;
        q->next=p->next;
    }
    else
        printf("LinkList is empty!\n");
}

int main()
{
    CLinkList ML;
    int i,res;
    ML=(CLinkList)malloc(sizeof(CNode));
    InitCL(ML);
    DeleteFront(ML,&res);
    DeleteEnd(ML,&res);
    for(i=1;i<=8;i++)

```

```

    {
        InsertFront(ML,i+38);
        InsertEnd(ML,i+100);
    }
    Print(ML);
    DeleteFront(ML,&res);
    printf("%d is deleted from linklist./n",res);
    DeleteEnd(ML,&res);
    printf("%d is deleted from linklist./n",res);
    Print(ML);
    printf("Hello, world/n");
    getch();
    return 0;
}

```

#### 1.4.4 经典题目

题目出处/来源

**Hrbust 1548 基础数据结构——循环链表**

题目描述

n 个人想玩残酷的死亡游戏，游戏规则如下：

n 个人进行编号，分别从 1 到 n，排成一个圈，顺时针从 1 开始数到 m，数到 m 的人被杀，剩下的人继续游戏，活到最后的一个人是胜利者。

请输出最后一个人的编号。

输入

输入 n 和 m 值。m>1。

输出

输出胜利者的编号。

样例输入

5 3

样例输出

4

提示

第一轮：3 被杀第二轮：1 被杀第三轮：5 被杀第四轮：2 被杀

代码

```

#include<stdio.h>
#include<stdlib.h>

```

```

struct node
{
    int data;

```

```

    struct node *next;
};

//删除操作
void Del(struct node *head, int m)
{
    struct node *p, *q;
    int i = 1; //计数。
    p = q = head;
    while(p != NULL) {
        if(i == m) {
            //删除链表中元素。
            q->next = p->next;
            free(p);
            p = q->next;
            i = 1;
        }
        q = p;
        p = p->next;
        if(q == p) {
            //最后一个元素，按照题意应该输出。
            printf("%d\n", p->data);
            break;
        }
        i++;
    }
}

int main()
{
    int n, m;
    while(scanf("%d %d", &n, &m) != EOF) {
        struct node *head = NULL;
        struct node *p, *q;
        //创建循环链表。head 为表头指针。
        p = (struct node *)malloc(sizeof(struct node));
        p->data = 1;
        head = p;
        for(int i = 2; i <= n; i++) {
            q = (struct node *)malloc(sizeof(struct node));
            q->data = i;
            p->next = q;
            p = q;
        }
        p->next = head;
        //创建完毕。
        Del(head, m);
    }
    return 0;
}

```



## 1.5 栈

### 1.5.1 基本原理

#### 1.5.1.1 栈的定义

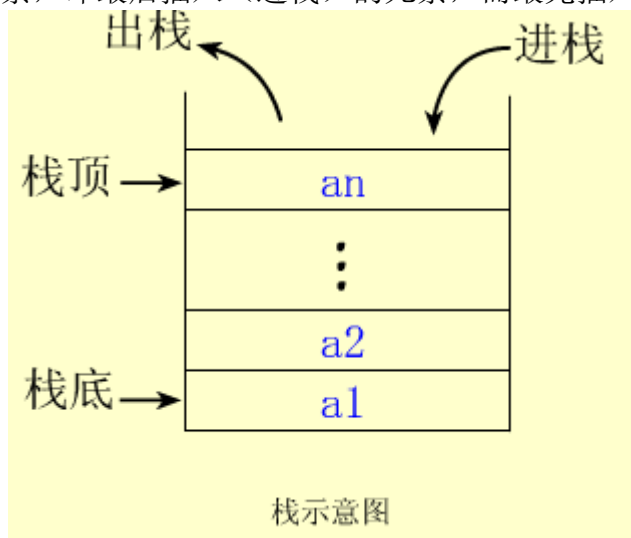
栈(stack)是限制线性表中元素的插入和删除只能在线性表的同一端进行的一种特殊线性表。允许插入和删除的一端,为变化的一端,称为栈顶(Top),另一端为固定的一端,称为栈底(Bottom)。

(1)通常称插入、删除的这一端为栈顶 (Top), 另一端称为栈底 (Bottom)。

(2)当表中没有元素时称为空栈。

(3)栈为后进先出 (Last In First Out) 的线性表, 简称为 LIFO 表。

栈的修改是按后进先出的原则进行。每次删除 (退栈) 的总是当前栈中"最新"的元素, 即最后插入 (进栈) 的元素, 而最先插入的是被放在栈的底部, 要到最后才能删除。



【示例】元素是以  $a_1, a_2, \dots, a_n$  的顺序进栈, 退栈的次序却是  $a_n, a_{n-1}, \dots, a_1$ 。

#### 1.5.1.2 栈的特点

根据栈的定义可知, 最先放入栈中元素在栈底, 最后放入的元素在栈顶, 而删除元素刚好相反, 最后放入的元素最先删除, 最先放入的元素最后删除。

也就是说, 栈是一种后进先出(Last In First Out)的线性表, 简称为 LIFO 表。

#### 1.5.1.3 栈的运算

1.初始化栈: `INISTACK(&S)`

将栈  $S$  置为一个空栈(不含任何元素)。

2.进栈: `PUSH(&S,X)`

将元素  $X$  插入到栈  $S$  中, 也称为“入栈”、“插入”、“压入”。

3.出栈: `POP(&S)`

删除栈  $S$  中的栈顶元素, 也称为“退栈”、“删除”、“弹出”。

4.取栈顶元素: `GETTOP(S)`

取栈  $S$  中栈顶元素。

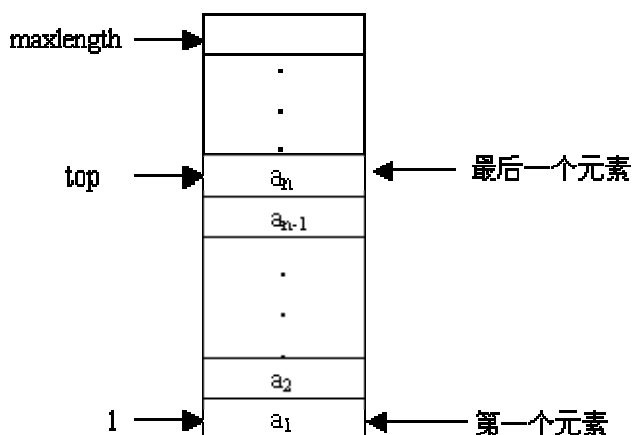
5.判栈空: `EMPTY(S)`

判断栈  $S$  是否为空, 若为空, 返回值为 1, 否则返回值为 0。

### 1.5.1.4 顺序栈

栈的顺序存储结构简称为顺序栈,它是运算受限的顺序表。

#### 1. 栈的顺序存储结构



#### 一、栈的数组表示

```
#define MAXN 100; //定义栈的最大容量为 MAXN
```

```
char stack[MAXN]; //将栈中元素定义为字符类型
```

```
int top; //指向栈顶位置的指针
```

#### 二、栈的变化

栈总是处于栈空、栈满或不空不满三种状态之一，它们是通过栈顶指针 `top` 的值体现出来的。

规定：`top` 的值为下一个进栈元素在数组中的下标值。

栈空时（初始状态），`top=0`;

栈满时，`top=MAXN`。

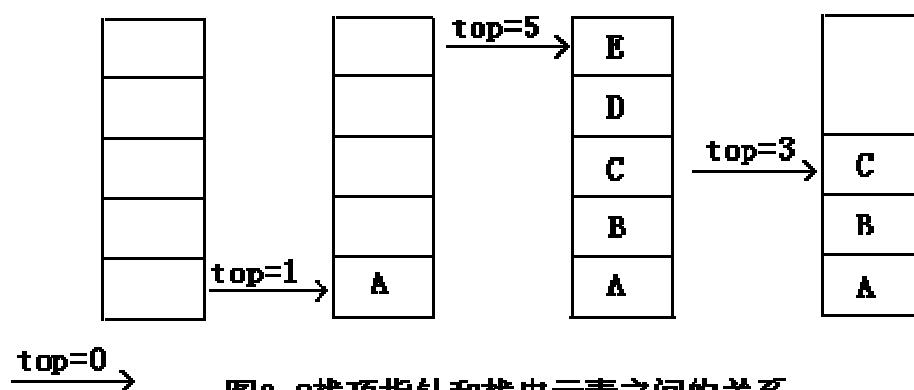


图3.2 栈顶指针和栈中元素之间的关系

#### 三、栈的五种运算

##### (一) 进栈

##### (一) 进栈

##### 1) 进栈算法

(1) 检查栈是否已满，若栈满，进行“溢出”处理。

(2) 将新元素赋给栈顶指针所指的单元。

(3) 将栈顶指针上移一个位置（即加1）。

注意：若 `top=1`，执行此顺序 1->2->3；`top=0` 执行 1->3->2 的顺序；

## 2) 实现程序

```
int top=0
int push(char x)
{
    if (top>=MAXN) return(1);
    stack[top++]=x; return(0);
}
```

### (二) 出栈

#### 1) 出栈算法

(1) 检查栈是否为空，若栈空，进行“下溢”处理。

(2) 将栈顶指针下移一个位置（即减1）。

(3) 取栈顶元素的值，以便返回给调用者。



#### 2) 实现程序

```
int pop(char *p_x)
{
    if (top==0) return(1);
    *p_x= stack[--top]; return(0);
}
```

### (三) 初始化栈

```
void inistack(stack )
{
    top=0;
}
```

### (四) 取栈顶元素

```
char gettop(stack)
{
    int i;
    if (top==0)
    {printf("underflow"); return (1);}
    else {i=top-1;
    return (stack[i]);}
}
```



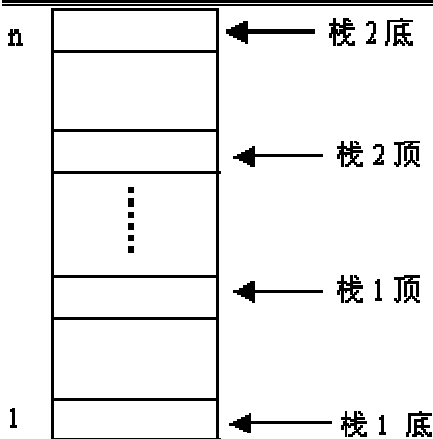
### (五) 判栈空否

```
int empty(stack)
{
    if (top==0)
    return (0);
    else return (1);
}
```

## 四. 栈的共享存储单元

有时，一个程序设计中，需要使用多个同一类型的栈,这时候，可能会产生一个栈空间过小，容量发生溢出，而另一个栈空间过大，造成大量存储单元浪费的现象。为了充分利用各个栈的存储空间，这时可以采用多个栈共享存储单元，即给多个栈分配一个足够大的

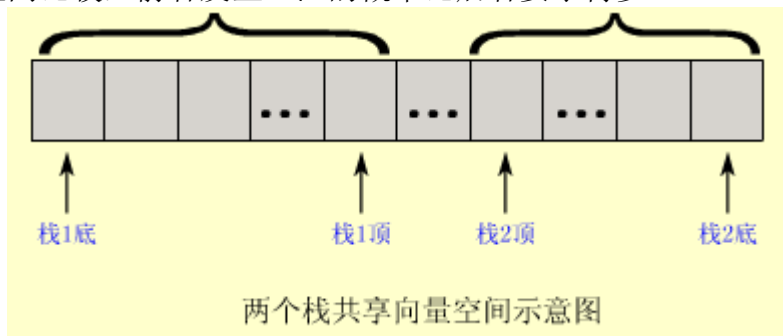
存储空间，让多个栈实现存储空间优势互补。



#### 4、两个栈共享同一存储空间

当程序中同时使用两个栈时，可以将两个栈的栈底设在向量空间的两端，让两个栈各自向中间延伸。当一个栈里的元素较多，超过向量空间的一半时，只要另一个栈的元素不多，那么前者就可以占用后者的部分存储空间。

只有当整个向量空间被两个栈占满（即两个栈顶相遇）时，才会发生上溢。因此，两个栈共享一个长度为  $m$  的向量空间和两个栈分别占用两个长度为  $\lfloor m/2 \rfloor$  和  $\lceil m/2 \rceil$  的向量空间比较，前者发生上溢的概率比后者要小得多。



两个栈共享向量空间示意图

### 1.5.2 解题思路

### 1.5.3 模板代码

```
/*栈的顺序存储表示 */
#define STACK_INIT_SIZE 10 /* 存储空间初始分配量 */
#define STACK_INCREMENT 2 /* 存储空间分配增量 */

typedef struct SqStack
{
    SElemType *base; /* 在栈构造之前和销毁之后，base 的值为 NULL */
    SElemType *top; /* 栈顶指针 */
    int stacksize; /* 当前已分配的存储空间，以元素为单位 */
} SqStack; /* 顺序栈 */

void InitStack(SqStack *S)
{ /* 构造一个空栈 S */
    (*S).base=(SElemType *)malloc(STACK_INIT_SIZE*sizeof(SElemType));
    if(!(*S).base)
        exit(OVERFLOW); /* 存储分配失败 */
}
```

```

        (*S).top=(*S).base;
        (*S).stacksize=STACK_INIT_SIZE;
    }

void DestroyStack(SqStack *S)
{ /* 销毁栈 S, S 不再存在 */
    free((*S).base);
    (*S).base=NULL;
    (*S).top=NULL;
    (*S).stacksize=0;
}

Status StackEmpty(SqStack S)
{ /* 若栈 S 为空栈, 则返回 TRUE, 否则返回 FALSE */
    if(S.top==S.base)
        return TRUE;
    else
        return FALSE;
}

Status GetTop(SqStack S,SElemType *e)
{ /* 若栈不空, 则用 e 返回 S 的栈顶元素, 并返回 OK; 否则返回 ERROR */
    if(S.top>S.base)
    {
        *e=*(S.top-1);
        return OK;
    }
    else
        return ERROR;
}

void Push(SqStack *S,SElemType e)
{ /* 插入元素 e 为新的栈顶元素 */
    if((*S).top-(*S).base>=(*S).stacksize) /* 栈满, 追加存储空间 */
    {
        (*S).base=(SElemType
*)realloc((*S).base,((*S).stacksize+STACK_INCREMENT)*sizeof(SElemType));
        if(!(*S).base)
            exit(OVERFLOW); /* 存储分配失败 */
        (*S).top=(*S).base+(*S).stacksize;
        (*S).stacksize+=STACK_INCREMENT;
    }
    *((*S).top)++=e;
}

Status Pop(SqStack *S,SElemType *e)
{ /* 若栈不空, 则删除 S 的栈顶元素, 用 e 返回其值, 并返回 OK; 否则返回 ERROR */
    if((*S).top==(*S).base)
        return ERROR;

```

```

        *e=*--(*S).top;
        return OK;
    }

```

### 1.5.4 经典题目

题目出处/来源

**Hrbust 1549 基础数据结构——栈**

题目描述

给你一串字符，不超过 50 个字符，可能包括括号、数字、字母、标点符号、空格，你的任务是检查这一串字符中的( ), [ ], { } 是否匹配。

输入

输入数据有多组，每组数据不超过 100 个字符并含有( , ) , [ , ] , { , } 一个或多个。处理到文件结束。

输出

如果匹配就输出 “yes”，不匹配输出 “no”

样例输入

```

sin(20+10)
{[]}]

```

样例输出

```

yes
no

```

代码

```

#include<stdio.h>
#include<string.h>

int main()
{
    char stack[101]; //栈。
    char str[101];
    while(gets(str)) {
        int top = -1; //栈顶指针。
        for(int i = 0; str[i]; i++) {
            if(str[i] == '(' || str[i] == '[' || str[i] == '{') {
                stack[++top] = str[i];
            }

            if(str[i] == ')' && stack[top] != '(') {
                top = 0; break;
            }
            if(str[i] == ']' && stack[top] != '[') {
                top = 0; break;
            }
            if(str[i] == '}' && stack[top] != '{') {
                top = 0; break;
            }
        }
    }
}

```

```

    }

    if(str[i] == ')' && stack[top] == '(') top--;
    if(str[i] == ']' && stack[top] == '[') top--;
    if(str[i] == '}' && stack[top] == '{') top--;
    }
    if(top == -1) printf("yes\n");
    else printf("no\n");
    }
    return 0;
}

```

### 1.5.5 扩展变形

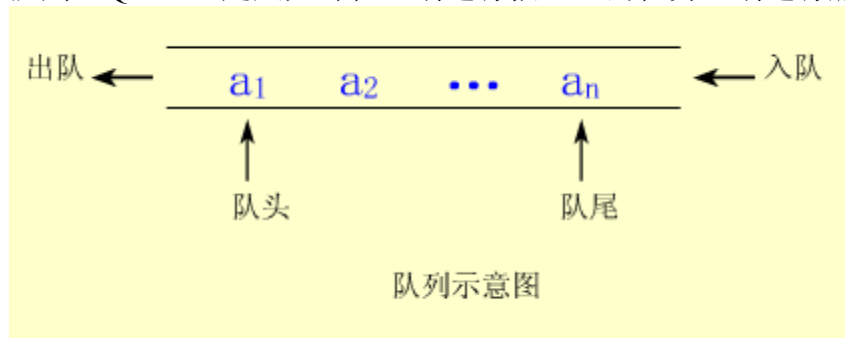
Hrbust 1182 栈

## 1.6 队列

### 1.6.1 基本原理

#### 1.6.1.1 定义

队列（Queue）是只允许在一端进行插入，而在另一端进行删除的运算受限的线性表



- (1) 允许删除的一端称为队头（Front）。
- (2) 允许插入的一端称为队尾（Rear）。
- (3) 当队列中没有元素时称为空队列。
- (4) 队列亦称作先进先出（First In First Out）的线性表，简称为 FIFO 表。

队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（即不允许“加塞”），每次离开的成员总是队列头上的（不允许中途离队），即当前“最老的”成员离队。

除了栈和队列之外，还有一种限定性数据结构是双端队列(Deque)。

双端队列是限定插入和删除操作在表的两端进行的线性表。这两端分别称为端点 1 和端点 2。也可象栈一样，可以用一个铁道转轨网络来比喻双端队列。在实际使用中，还可以有输出受限的双端队列(即一个端点允许插入和删除，另一个端点只允许插入的双端队列)。而如果限定双端队列从某个端点插入的元素只能从该端点删除，则该双端队列就蜕变为两个栈底相邻接的栈了。

尽管双端队列看起来似乎比栈和队列更灵活，但实际上在程序系统中远不及栈和队列有用，故在此不作详细讨论。

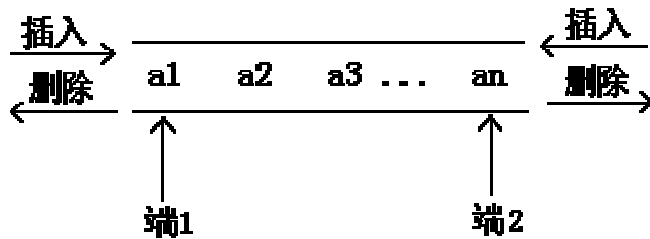


图3.11双端队列示意图

### 1.6.1.2 队列的基本逻辑运算

#### (1) InitQueue (Q)

置空队。构造一个空队列 Q。

#### (2) QueueEmpty (Q)

判队空。若队列 Q 为空，则返回真值，否则返回假值。

#### (3) QueueFull (Q)

判队满。若队列 Q 为满，则返回真值，否则返回假值。

注意：

此操作只适用于队列的顺序存储结构。

#### (4) EnQueue (Q, x)

若队列 Q 非满，则将元素 x 插入 Q 的队尾。此操作简称入队。

#### (5) DeQueue (Q)

若队列 Q 非空，则删去 Q 的队头元素，并返回该元素。此操作简称出队。

#### (6) QueueFront (Q)

若队列 Q 非空，则返回队头元素，但不改变队列 Q 的状态。

### 1.6.1.3 顺序队列

队列的顺序存储结构称为顺序队列，顺序队列实际上是运算受限的顺序表，和顺序表一样，顺序队列也是必须用一个数组来存放当前队列中的元素。由于队列的队头和队尾的位置是变化的，因而要设两个指针和分别指示队头和队尾元素在队列中的位置。

#### 1、顺序队列

##### (1) 顺序队列的定义

队列的顺序存储结构称为顺序队列，顺序队列实际上是运算受限的顺序表。

##### (2) 顺序队列的表示(数组表示)

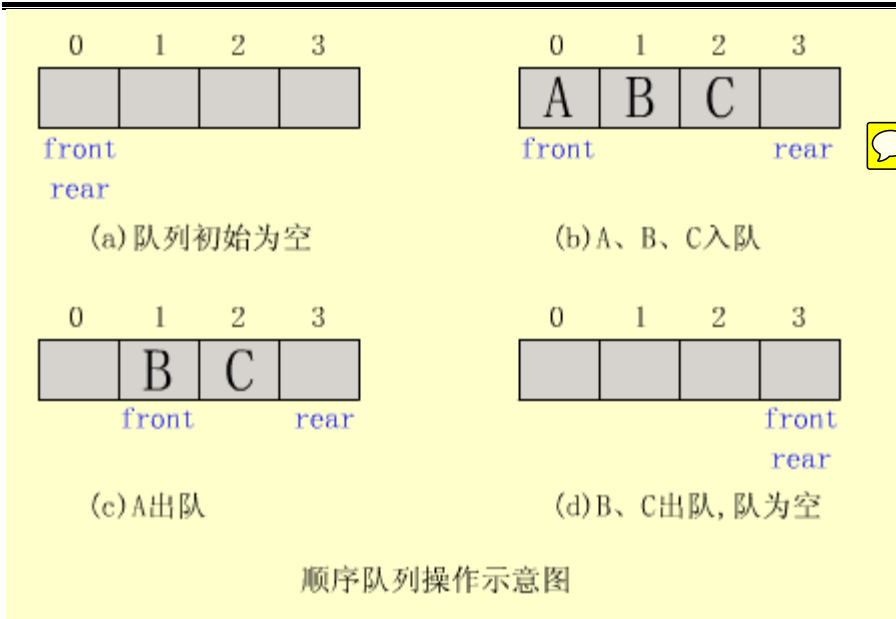
```
#define maxsize= maxlen; //定义队列的最大容量为 maxlen
```

```
elemtype squeue[maxsize]; //将队列中元素定为数组型，元素类型为 elemtype
```

```
int head; //队头指针
```

```
int tail; //队尾指针
```





### (3) 顺序队列的基本操作

队列也有队空、队满或不空不满三种情况。

#### 1. 第一种表示方法

规定：**head** 指向**队首元素**的位置，**tail** 指向**队尾元素**的位置。队列初始状态设为 **head=0, tail=-1**。

当队列非空时， $\text{tail} \geq \text{head}$ ;

当队列空时， $\text{head} > \text{tail}$ ;

当队列满时， $\text{tail} = \text{maxsize} - 1$ 。

#### 2. 第二种表示方法

规定：**head** 指向**队首元素的前一个位置**，**tail** 指向**队尾元素的位置**。队列初始状态设为 **head=tail=-1**。

1) 当队列非空时， $\text{tail} > \text{head}$ ;

2) 当队列空时， $\text{head} = \text{tail}$ ;

3) 当队列满时， $\text{tail} = \text{maxsize} - 1$ 。

① 入队时：将新元素插入 **rear** 所指的位置，然后将 **rear** 加 1。

② 出队时：删去 **front** 所指的元素，然后将 **front** 加 1 并返回被删元素。

注意：

① 当**头尾指针相等**时，队列为空。

② 在非空队列里，**队头指针始终指向队头元素**，**尾指针始终指向队尾元素的下一位置**。

### (4) 顺序队列中的溢出现象

#### ① "下溢"现象

当队列为空时，做出队运算产生的溢出现象。“**下溢**”是正常现象，常用作程序控制转移的条件。

#### ② "真上溢"现象

当**队列满**时，做进栈运算产生空间溢出的现象。“**真上溢**”是一种出错状态，应设法避免。

### ③ "假上溢"现象

从顺序存储的队列可以看出，有可能出现这样情况，尾指针指向一维数组最后，但前面有很多元素已经出队，即空出很多位置，这时要插入元素，仍然会发生溢出。例如，在下图中，若队列的最大容量  $\text{maxsize}=4$ ，此时， $\text{tail}=3$ ，再进队时将发生溢出。我们将这种溢出称为“假溢出”。

要克服“假溢出”，可以将整个队列中元素向前移动，直到头指针  $\text{head}$  为零，或者每次出队时，都将队列中元素前移一个位置。因此，顺序队列的队满判定条件为  $\text{tail}=\text{maxsize}-1$ 。但是，在顺序队列中，这些克服假溢出的方法都会引起大量元素的移动，花费大量的时间，所以在实际应用中很少采用，一般采用下面的循环队列形式。

#### 1.6.1.4 循环队列

##### 一) 定义

为了克服顺序队列中假溢出，通常将一维数组  $\text{queue}[0]$  到  $\text{queue}[\text{maxsize}-1]$  看成是一个首尾相接的圆环，即  $\text{queue}[0]$  与  $\text{queue}[\text{maxsize}-1]$  相接在一起。将这种形式的顺序队列称为循环队列。

若  $\text{tail}+1=\text{maxsize}$ ，则令  $\text{tail}=0$ 。这样运算很不方便，可利用数学中的求模运算来实现。

入队： $\text{tail}=(\text{tail}+1) \bmod \text{maxsize}$ ;  $\text{queue}[\text{tail}]=x$ ;

出队： $\text{head}=(\text{head}+1) \bmod \text{maxsize}$ 。

##### 二) 循环队列的变化

在循环队列中，若  $\text{head}=\text{tail}$ ，则称为队空，若  $(\text{tail}+1) \bmod \text{maxsize}=\text{head}$ ，则称为队满，这时，循环队列中能装入的元素个数为  $\text{maxsize}-1$ ，即浪费一个存储单元，但是这样可以给操作带来较大方便。

##### 三) 循环队列上五种运算实现

###### 1. 进队列

###### 1) 进队列算法

- (1) 检查队列是否已满，若队满，则进行溢出错误处理；
- (2) 将队尾指针后移一个位置（即加 1），指向下一单元；
- (3) 将新元素赋给队尾指针所指单元。

###### 2) 进队列实现程序

```
int head=0,tail=0;
int enqueue (elemtype queue[], elemtype x)
{
    if ((tail+1)%maxsize == head) return(1);
    else
    {
        tail=(tail+1)%maxsize;
        queue[tail]=x;
        return(0);
    }
}
```

###### 2. 出队列

###### 1) 出队列算法

- (1) 检查队列是否为空，若队空，则进行下溢错误处理；
- (2) 将队首指针后移一个位置（即加 1）；
- (3) 取队首元素的值。

###### 2) 出队列实现程序

```

int head=0,tail=0;
int dlqueue(elemtype queue[ ],elemtype *p_x )
{
    if (head==tail) return(1);
    else
    {
        head =(head+1) % maxsize;
        *p_x=queue[head];
    }
    return(0);
}
    
```

(3) 队列初始化

head=tail=0;

(4) 取队头元素 (注意得到的应为头指针后面一个位置值)

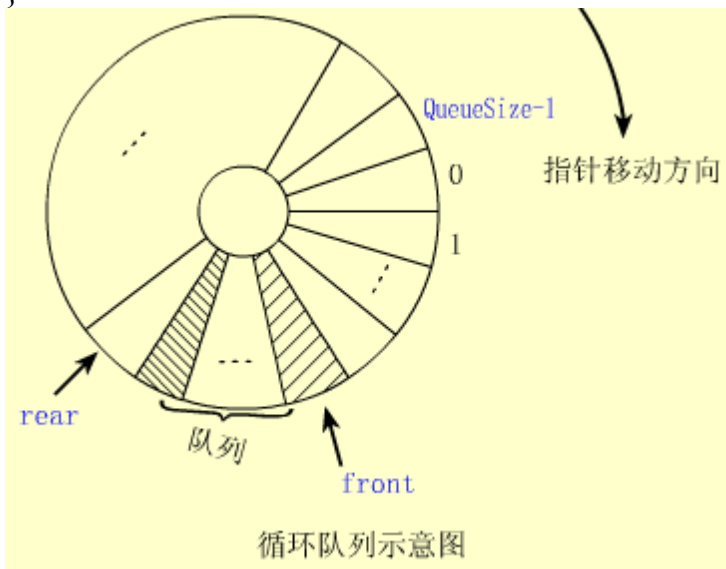
```

elemtype gethead(elemtype queue[ ])
{
    if (head==tail) return(null);
    else
    return (queue[(head+1)%maxsize]);
}
    
```

(5) 判队列空否

```

int empty(elemtype queue[ ])
{
    if (head==tail) return (1);
    else return (0);
}
    
```



(1) 循环队列的基本操作

循环队列中进行出队、入队操作时，头尾指针仍要加 1，朝前移动。只不过当头尾指针指向向量上界 (QueueSize-1) 时，其加 1 操作的结果是指向向量的下界 0。这种循环意义下的加 1 操作可以描述为：

① 方法一：

```

if(i+1==QueueSize) //i 表示 front 或 rear
    i=0;
else
    
```

```
i++;
```

## ② 方法二--利用"模运算"

```
i=(i+1)%QueueSize;
```

### (2) 循环队列边界条件处理

循环队列中，由于入队时尾指针向前追赶头指针；出队时头指针向前追赶尾指针，造成队空和队满时头尾指针均相等。因此，无法通过条件 `front==rear` 来判别队列是"空"还是"满"。

解决这个问题的方法至少有三种：

- ① 另设一布尔变量以区别队列的空和满；
- ② 少用一个元素的空间。约定入队前，测试尾指针在循环意义下加 1 后是否等于头指针，若相等则认为队满（注意：`rear` 所指的单元始终为空）；
- ③ 使用一个计数器记录队列中元素的总数（即队列长度）。

### (3) 循环队列的类型定义

```
#define QueueSize 100    //应根据具体情况定义该值
typedef char QueueDataType; //DataType 的类型依赖于具体的应用
typedef struct {
    int front;             //头指针，队非空时指向队头元素
    int rear;              //尾指针，队非空时指向队尾元素的下一位置
    int count;             //计数器，记录队中元素总数
    QueueDataType data[QueueSize]
}CirQueue;
```

### (4) 循环队列的基本运算

用第三种方法，循环队列的六种基本运算：

#### ① 置队空

```
void InitQueue(CirQueue *Q)
{
    Q->front=Q->rear=0;
    Q->count=0;    //计数器置 0
}
```

#### ② 判队空

```
int QueueEmpty(CirQueue *Q)
{
    return Q->count==0; //队列无元素为空
}
```

#### ③ 判队满

```
int QueueFull(CirQueue *Q)
{
    return Q->count==QueueSize; //队中元素个数等于 QueueSize 时队满
}
```

#### ④ 入队

```
void EnQueue(CirQueue *Q, QueueDataType x)
{
    if(QueueFull(Q))
```

```

        Error("Queue overflow");           //队满上溢
        Q->count++;                         //队列元素个数加 1
        Q->data[Q->rear]=x;                 //新元素插入队尾
        Q->rear=(Q->rear+1)%QueueSize;      //循环意义下将尾指针加 1
⑤ 出队
        DataType DeQueue(CirQueue *Q)
        {
            DataType temp;
            if(QueueEmpty((Q))
                Error("Queue underflow");   //队空下溢
            temp=Q->data[Q->front];
            Q->count--;                      //队列元素个数减 1
            Q->front=(Q->front+1)&QueueSize; //循环意义下的头指针加 1
            return temp;
        }

⑥取队头元素
        DataType QueueFront(CirQueue *Q)
        {
            if(QueueEmpty(Q))
                Error("Queue if empty.");
            return Q->data[Q->front];
        }
    
```

## 1.6.2 解题思路

## 1.6.3 模板代码

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef int QElemType;

typedef struct QNode
{
    QElemType data;
    struct QNode *next;
}QNode,*QueuePtr;

typedef struct
{
    QueuePtr front;
    QueuePtr rear;
}LinkQueue;

int InitQueue(LinkQueue *Q);
int DestroyQueue(LinkQueue *Q);
int ClearQueue(LinkQueue *Q);
int QueueEmpty(LinkQueue Q);
    
```

```

int QueueLength(LinkQueue Q);
int GetHead(LinkQueue Q, QElemType *e);
int InsertQueue(LinkQueue *Q, QElemType e);
int DelQueue(LinkQueue *Q, QElemType *e);
int PrintQueue(LinkQueue Q);

int InitQueue(LinkQueue *Q)
{
    Q->front = Q->rear = (QueuePtr )malloc(sizeof(QNode));
    if(!Q->front)
    {
        perror("malloc error\n");
        return -1;
    }
    Q->front->next = NULL;
    Q->front->data = 0;
    return 0;
}

int DestroyQueue(LinkQueue *Q)
{
    while(Q->front)
    {
        Q->rear = Q->front->next;
        free(Q->front);
        Q->front = Q->rear;
    }

    Q = NULL;
    return 0;
}

int ClearQueue(LinkQueue *Q)
{
    return 0;
}

int QueueEmpty(LinkQueue Q)
{
    if(Q.front == Q.rear)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int QueueLength(LinkQueue Q)
{
    return Q.front->data;
}

int GetHead(LinkQueue Q, QElemType *e)
{

```

```

    if(Q.front->next == NULL)
    {
        perror("Queue is empty!\n");
        *e = -1;
        return -1;
    }
    *e = Q.front->next->data;
    return 0;
}

int InsertQueue(LinkQueue *Q, QElemType e)
{
    QueuePtr p = (QueuePtr )malloc(sizeof(QNode));
    if(p == NULL)
    {
        perror("malloc error!\n");
        return -1;
    }

    p->data = e;
    p->next = NULL;
    (Q->rear)->next = p;
    Q->rear = p;

    Q->front->data++;
    return 0;
}

int DelQueue(LinkQueue *Q, QElemType *e)
{
    if(Q->front == Q->rear)
    {
        perror("The queue is empty!");
        return -1;
    }

    QueuePtr p = (QueuePtr )malloc(sizeof(QNode));
    p = Q->front->next;
    *e = p->data;
    Q->front->next = p->next;
    if(Q->rear == p)
    {
        Q->rear = Q->front;
    }
    free(p);
    Q->front->data--;
    return 0;
}

int PrintQueue(LinkQueue Q)
{
    Q.front = Q.front->next;
    while(Q.front != NULL)
    {

```

```

        printf("%d-----",Q.front->data);
        Q.front = Q.front->next;
    }
    return 0;
}

```

### 1.6.4 经典题目

题目出处/来源


**Hrbust 1180 报数** 

题目描述

有  $N$  个人围成一圈，按顺时针给他们编号为  $1-N$ 。

紧接着，指定编号为  $M$  的人开始报数，报数按顺时针进行。

报到  $D$  的人出列，下一个人重新开始报数。按此规律，每次报到  $D$  的人都出列。

要求同学编程求出出列的顺序。 

输入

输入包括多组测试用例。

对于每组用例，第一行是一个整数  $N$ ，表示人数。 $N < 100$ 。

接下来  $N$  行是每个人的姓名。姓名为长度不超过 20 连续字符串。

最后是以两个以","分割的整数  $M, D$ 。代表从  $M$  个人开始，每报  $D$  个数出列。

输出

输出所求的顺序

样例输入

```

8
Zhao
Qian
Sun
Li
Zhou
Wu
Zheng
Wang
4,4

```

样例输出

```

Zheng
Sun
Wang
Zhou
Li
Wu
Qian
Zhao

```

代码

```

#include<iostream>
#include<queue>

```



```

#include<string>
#include<stdio.h>
using namespace std;
int main(){

    int n;
    while(cin>>n){
        queue<string> a;
        for(int i=0;i<n;i++){
            string name;
            cin>>name;
            a.push(name);
        }
        int w,s;
        scanf("%d,%d",&w,&s);
        for(int i=0;i<w-1;i++){
            string temp = a.front();
            a.pop();
            a.push(temp);
        }
        while(a.size()){
            for(int i=0;i<s-1;i++){
                string temp = a.front();
                a.pop();
                a.push(temp);
            }
            cout<<a.front()<<endl;
            a.pop();
        }
    }
}
    
```

## 1.6.5 扩展变形

Hrbust 1181 移动（广度优先搜索）

Hrbust 1522 子序列的和（单调队列）

## 1.7 串

### 1.7.1 基本原理

#### 1.7.1.1 串的基本概念

##### 1. 串的定义

串( string) 是由零个或多个字符组成的有限序列，记作  $s = "a_1a_2 \cdots a_n"$ ，其中  $s$  为串的名字，用成对的双引号括起来的字符序列为串的值，但两边的双引号不算串值，不包含在串中。 $a_i (1 \leq i \leq n)$  可以是字母、数字或其它字符。（取决于程序设计语言所使用的字符集）， $n$  为串中字符的个数，称为串的长度。

##### 2. 空串

不含任何字符的串称为空串，它的长度  $n=0$ ，记为  $s=""$ 。

### 3. 空白串

含有一个或多个空格的串，称为空白串，它的长度是串中空格字符的个数，记为  $s="?"$ 。注意与空串的区别。

### 4. 子串、主串

若一个串是另一个串中连续的一段，则这个串称为另一个串的子串，而另一个串相对于该串称为主串。例如，串  $s1="abcdefg"$ ， $s2="abcdefghxyz"$ ，则  $s1$  为  $s2$  的子串， $s2$  相对于  $s1$  为主串。

另外，空串是任意串的子串，任意串是自身的子串。通常称字符在序列中的序号为该字符在串中的位置，子串在主串的位置则以子串的第一个字符在主串中的位置来表示。

### 5. 串变量和串常量

通常在程序中使用的串可分为：串变量和串常量。

#### (1) 串变量

串变量和其它类型的变量一样，其取值是可以改变的。

#### (2) 串常量

串常量和整常数、实常数一样，在程序中只能被引用但不能改变其值。即只能读不能写。

①串常量由直接量来表示的：

【例】Error("overflow") 中 "overflow" 是直接量。

②串常量命名

有的语言允许对串常量命名，以使程序易读、易写。

【例】C++中，可定义串常量 path

```
const char path[]="dir/bin/appl";
```

## 1.7.1.2 串的基本运算概述

为描述方便，假定用大写字母表示串名，小写字母表示组成串的字符。

### 1. 串复制 strcpy(S,T)

表示将 T 串的值赋给 S 串。

### 2. 联接 strcat(S,T)

表示将 S 串和 T 串联接起来，使 T 串接入 S 串的后面。

### 3. 求串长度 strlen(T)

求 T 串的长度。

### 4. 子串 strstr(S,i,j,T)

表示截取 S 串中从第 i 个字符开始连续 j 个字符，作为 S 的一个子串，存入 T 串。

### 5. 串比较大小 strcmp(S,T)

比较 S 串和 T 串的大小，若  $S<T$ ，函数值为负，若  $S=T$ ，函数值为零，若  $S>T$ ，函数值为正。

### 6. 串插入 strins(S,i,T)

在 S 串的第 i 个位置插入 T 串。

### 7. 串删除 strdel(S,i,j)

删除串 S 中从第 i 个字符开始连续 j 个字符。

8. 求子串位置  $\text{index}(S, T)$

求 T 子串在 S 主串中首次出现的位置，若 T 串不是 S 串的子串，则位置为零。

9. 串替换  $\text{replace}(S, i, j, T)$

将 S 串中从第 i 个位置开始连续 j 个字符，用 T 串替换。

利用上述九种基本运算还可以组合成字符串的其他有关操作。

### 1.7.1.3 串的顺序存储

串的顺序存储结构，也称为顺序串，与第二章介绍的顺序表类似，就是用一组地址连续的存储单元依次存放串的各个字符。但由于串中元素全部为字符，故存放形式与顺序表有所区别。

计算机的编址方式：

按字节编址（以字节为存取单位）

按字编址（以字为存取单位） 紧缩存储/非紧缩存储

1. 串的非紧缩存储

一个存储单元中只存储一个字符，和顺序表中一个元素占用一个存储单元类似。具体形式见图 4-1，设串 S="How do you do"。

A			
B			
C			
D			
E			
F			
G			

图一. 串值存储的非紧缩格式示例

2. 串的紧缩存储

根据各机器字的长度，尽可能将多个字符存放在一个字中。假设一个字可存储 4 个字符，则紧缩存储具体形式。

A	B	C	D
E	F	G	

图二. 串值存储的紧缩格式示例

从上面介绍的两种存储方式可知，紧缩存储能够节省大量存储单元，但对串的单个字符操作很不方便，需要花费较多时间分离同一个字中的字符，运算效率较低。而非紧缩存储的特点刚好相反，操作方便，但将占用较多的内存单元。

两种方式的共同缺点是：插入或删除一个字符的相应算法效率较低（顺序结构的共同缺点。）

3. 串的字节存储

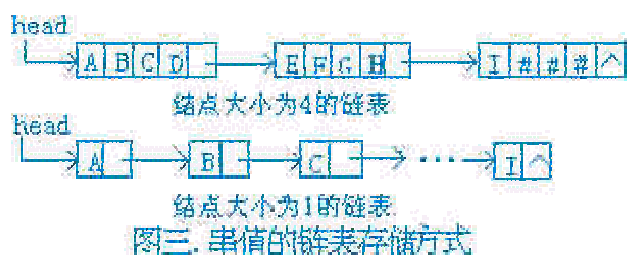
h	o	w		d	o		y	o	u		d	o	
---	---	---	--	---	---	--	---	---	---	--	---	---	--

### 1.7.1.4 串的链式存储

### 1. 结点大小为 1 的链式存储

和前面介绍到的单链表一样，每个结点为一个字符，链表也可以带头结点。

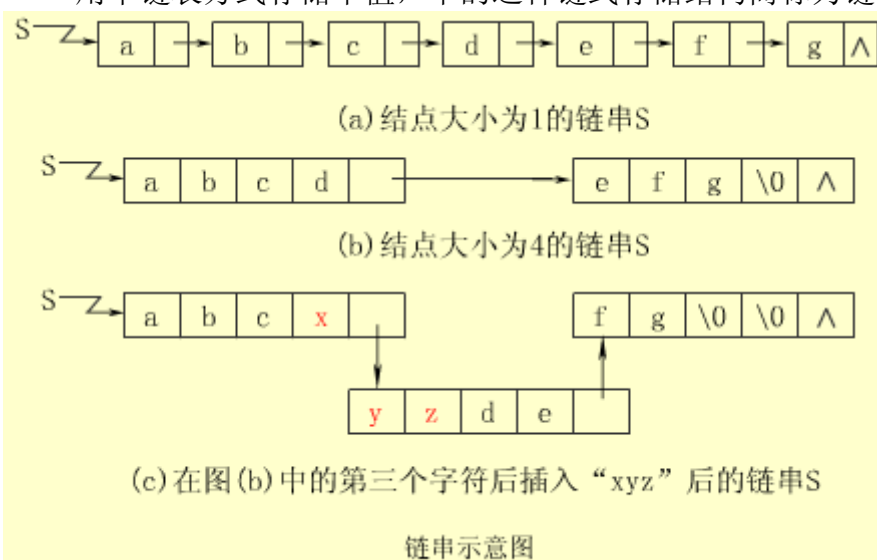
S="ABCDEFGHI"的存储结构具体形式见下图



### 2. 结点大小为 K 的链式存储

和紧缩存储类似，假设一个字中可以存储 K 个字符，则一个结点有 K 个数据域和一个指针域，若最后一个结点中数据域少于 K 个，那么必须在串的末尾加一个串的结束标志。例如串 S="ABCDEFGHI"的存储结构具体形式见上图。假设 K=4，并且链表带头结点。

用单链表方式存储串值，串的这种链式存储结构简称为链串。



### 3. 链串的结构类型定义

```
typedef struct node{
    char data;
    struct node *next;
}LinkStrNode; //结点类型
typedef LinkStrNode *LinkString; //LinkString 为链串类型
LinkString S; //S 是链串的头指针
```

注意：

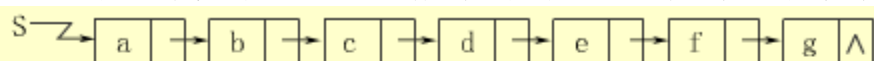
- ①链串和单链表的差异仅在于其结点数据域为单个字符；
- ②一个链串由头指针唯一确定。

### 4. 链串的结点大小

通常，将结点数据域存放的字符个数定义为结点的大小。结点的大小的值越大，存储密度越高。

(1) 结点大小为 1 的链串

【例】串值为"abcdef"的结点大小为 1 的链串 S 如下图所示。

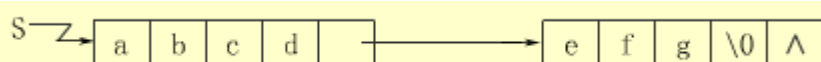


(a) 结点大小为1的链串S

这种结构便于进行插入和删除运算，但存储空间利用率太低。

(2) 结点大小>1 的链串

【例】串值为"abcdef"的结点大小为 4 的链串 S 如下图所示。



(b) 结点大小为4的链串S

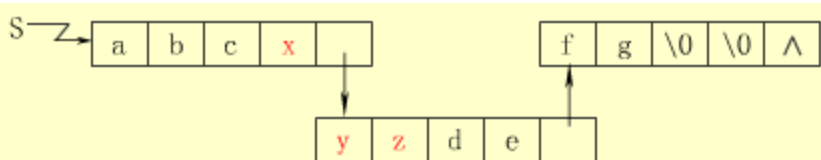
注意：

①为了提高存储密度，可使每个结点存放多个字符。

②当结点大小大于 1 时，串的长度不一定正好是结点大小的整数倍，因此要用特殊字符来填充最后一个结点，以表示串的终结。

③虽然提高结点的大小使得存储密度增大，但是做插入、删除运算时，可能会引起大量字符的移动，给运算带来不便。

【例】上图中，在 S 的第 3 个字符后插入“xyz”时，要移动原来 S 中后面 4 个字符的位置，结果见下图。



(c) 在图(b)中的第三个字符后插入“xyz”后的链串S

### 1.7.1.5 子串定位运算

串是特殊的线性表，故顺序串和链串上实现的运算分别与顺序表和单链表上进行的操作类似。

C 语言的串库<string.h>里提供了丰富的串函数来实现各种基本运算，因此我们对各种串运算的实现不作讨论。利用串函数实现串的基本运算部分内容请大家在 OJ (<http://acm.hrbust.edu.cn>) 上多多练习。

下面讨论在顺序串和链串上实现的子串定位运算。

#### 1、子串定位运算

子串定位运算类似于串的基本运算中的字符定位运算。只不过是找子串而不是找字符在主串中首次出现的位置。此运算的应用非常广泛。

【例】在文本编辑中，我们经常要查找某一特定单词在文本中出现的位置。解此问题的有效算法能极大地提高文本编辑程序的响应性能。

子串定位运算又称串的模式匹配或串匹配。

## 2、目标（串）和模式（串）

在串匹配中，一般将主串称为目标（串），子串称为模式（串）。

假设  $T$  为目标串， $P$  为模式串，且不妨设：

$$T = "t_0t_1t_2 \dots t_{n-1}"$$

$$P = "p_0p_1p_2 \dots p_{m-1}" (0 < m \leq n)$$

## 3、串匹配

串匹配就是对于合法的位置（又称合法的位移） $0 \leq i \leq n-m$ ，依次将目标串中的子串  $"t_i t_{i+1} \dots t_{i+m-1}"$  和模式串  $"p_0 p_1 p_2 \dots p_{m-1}"$  进行比较：

①若  $"t_i t_{i+1} \dots t_{i+m-1}" = "p_0 p_1 p_2 \dots p_{m-1}"$ ，则称从位置  $i$  开始的匹配成功，或称  $i$  为有效位移。

②若  $"t_i t_{i+1} \dots t_{i+m-1}" \neq "p_0 p_1 p_2 \dots p_{m-1}"$ ，则称从位置  $i$  开始的匹配失败，或称  $i$  为无效位移。

因此，串匹配问题可简化为找出某给定模式串  $P$  在给定目标串  $T$  中首次出现的有效位移。

注意：

有些应用中要求求出  $P$  在  $T$  中所有出现的有效位移。

## 4、顺序串上的子串定位运算

（1）朴素的串匹配算法的基本思想

即用一个循环来依次检查  $n-m+1$  个合法的位移  $i$  ( $0 \leq i \leq n-m$ ) 是否为有效位移。

（2）顺序串上的串匹配算法

以下以第二种定长的顺序串类型作为存储结构。给出串匹配的算法：

```
#define MaxStrSize 256 //该值依赖于应用，由用户定义
typedef struct{
    char ch[MaxStrSize]; //可容纳 256 个字符，并依次存储在 ch[0..n]中
    int length;
}SeqString;
int Naive StrMatch(SeqString T,SeqString P)
{//找模式 P 在目标 T 中首次出现的位置，成功返回第 1 个有效位移，否则返回-1
    int i,j,k;
    int m=P.length; //模式串长度
    int n=T.length; //目标串长度
    for(i=0;i<=n-m;i++){ //0<=i<=n-m 是合法的位移
        j=0;k=i; //下面用 while 循环判定 i 是否为有效位移
        while(j<m&&T.ch[k]==P.ch[j]){
            k++;j++;
        }
        if(j==m) //既 T[i..i+m-1]=P[0..m-1]
            return i; //i 为有效位移，否则查找下一个位移
    }//endfor
    return -1; //找不到有效位移，匹配失败
} //NaiveStrMatch
```

### (3) 算法分析

#### ①最坏时间复杂度

该算法最坏情况下的时间复杂度为  $O((n-m+1)m)$ 。

分析：当目标串和模式串分别是 "an-1b" 和 "am-1b" 时，对所有  $n-m+1$  个合法的位移，均要比较  $m$  个字符才能确定该位移是否为有效位移，因此所需比较字符的总次数为  $(n-m+1)m$ 。

#### ②模式匹配算法的改进

朴素的串匹配算法虽然简单，但效率低。其原因是在检查位移  $i$  是否为有效位移时，没有利用检查位移  $i-1, i, \dots, 0$  时的部分匹配结果。

若利用部分匹配结果，模式串右滑动的距离就不会是每次一位，而是每次使其向右滑动得尽可能远。这样可使串匹配算法的最坏时间控制在  $O(m+n)$  数量级上。

### 5、链串上的子串定位运算

用结点大小为 1 的单链表做串的存储结构时，实现朴素的串匹配算法很简单。只是现在的位移  $shift$  是结点地址而非整数，且单链表中没有存储长度信息。若匹配成功，则返回有效位移所指的结点地址，否则返回指针。具体算法如下：

```

LinkStrNode *LinkStrMatch(LinkString T, LinkString P)
{
    //在链串上求模式 P 在目标 T 首次出现的位置
    LinkStrNode * shift, *t, *p;
    shift=T; //shift 表示位移
    t=shift; p=P;
    while(t && p) {
        if(t->data == p->data) { //继续比较后续结点中字符
            t=t->next;
            p=p->next;
        }
        else { //已确定 shift 为无效位移
            shift=shift->next; //模式右移，继续判定 shift 是否为有效位移
            t=shift;
            p=P;
        }
    }
    //endwhile
    if(p == NULL)
        return shift; //匹配成功
    else
        return NULL; //匹配失败
}
    
```

该算法的时间复杂度与顺序表上朴素的串匹配算法相同。

## 1.7.2 经典题目

题目出处/来源

**Hrbust 1550 基础数据结构——字符串**

题目描述

A+B 问题相信大家都已经做过，简单的问题相信各位都有能力解决了。现在 A+B 又回来了，只不过这次 A B 的值会很大以至于没有数据类型能够存下这么大的数据。因此对于很大的整数加法我们采用字符数组模拟的方法来处理。这个过程非常的类似我们小学时笔算加法的过程。

输入

第一行输入一个整数 T (T 不大于 20)，表示测试数据组数。对于每组数据输入两个正整数 A B (位数不超过 100)。

输出

输出 A+B 的结果并换行

样例输入

1

1 1

样例输出

2

代码

```
#include<iostream>
#include<string.h>
using namespace std;
char a[1005],b[1005];
int num[2000],i,T,temp,m,n,k,w;
int max(int,int);
int main(void)
{
    cin>>T;
    while(T--)
    {
        cin>>a>>b;
        m=strlen(a);
        n=strlen(b);
        for(i=0;i<2000;i++) num[i]=0;
        temp=max(m,n);
        for(i=0;i<temp;i++)
        {
            if(m-1-i<0) k=0;
            else k=a[m-i-1]-48;
            if(n-1-i<0) w=0;
            else w=b[n-i-1]-48;
            num[i]+=(k+w);
        }
        for(i=0;i<temp;i++)
        {
            if(num[i]>=10)
            {
                num[i+1]+=num[i]/10;
                num[i]%=10;
            }
        }
    }
}
```



```

    }
    for(i=1900;i>=1;i--)
    {
        if(num[i]!=0) break;
    }
    if(m==1&&a[0]=='0') cout<<b<<endl;
    else if(n==1&&b[0]=='0') cout<<a<<endl;
    else
    {
        for(;i>=0;i--) cout<<num[i];
        cout<<endl;
    }
}
return 0;
}
int max(int a,int b)
{
    return a>b?a:b;
}

```

### 1.7.3 扩展变形

Hrbust 1309 入侵检测

Hrbust 1358 Leyni 的 U 盘

## 1.8 二叉树

编写：姜喜朋

校核：孟祥凤

### 1.8.1 基本原理

树 (tree) 是包含  $n$  ( $n>0$ ) 个结点的有穷集合，其中：

- (1) 每个元素称为结点 (node)；
- (2) 有一个特定的结点被称为根结点或树根 (root)。
- (3) 除根结点之外的其余数据元素被分为  $m$  ( $m\geq 0$ ) 个互不相交的集合  $T_1, T_2, \dots, T_{m-1}$ ，其中每一个集合  $T_i$  ( $1\leq i\leq m$ ) 本身也是一棵树，被称作原树的子树 (subtree)。

树也可以这样定义：树是有根结点和若干颗子树构成的。树是由一个集合以及在该集合上定义的一种关系构成的。集合中的元素称为树的结点，所定义的关系称为父子关系。父子关系在树的结点之间建立了一个层次结构。在这种层次结构中有一个结点具有特殊的地位，这个结点称为该树的根结点，或称为树根。

我们可以形式地给出树的递归定义如下：

单个结点是一棵树，树根就是该结点本身。

设  $T_1, T_2, \dots, T_k$  是树，它们的根结点分别为  $n_1, n_2, \dots, n_k$ 。用一个新结点  $n$  作为  $n_1, n_2, \dots, n_k$  的父亲，则得到一棵新树，结点  $n$  就是新树的根。我们称  $n_1, n_2, \dots, n_k$  为一组兄弟结点，它们都是结点  $n$  的子结点。我们还称  $n_1, n_2, \dots, n_k$  为结点  $n$  的子树。

空集合也是树，称为空树。空树中没有结点。

节点的度：一个节点含有的子树的个数称为该节点的度；

叶节点或终端节点：度为零的节点称为叶节点；

非终端节点或分支节点：度不为零的节点；

双亲节点或父节点：若一个结点含有子节点，则这个节点称为其子节点的父节点；

孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点；

兄弟节点：具有相同父节点的节点互称为兄弟节点；

树的度：一棵树中，最大的节点的度称为树的度；

节点的层次：从根开始定义起，根为第 1 层，根的子结点为第 2 层，以此类推；

树的高度或深度：树中节点的最大层次；

堂兄弟节点：双亲在同一层的节点互为堂兄弟；

节点的祖先：从根到该节点所经分支上的所有节点；

子孙：以某节点为根的子树中任一节点都称为该节点的子孙。

森林：由  $m$  ( $m \geq 0$ ) 棵互不相交的树的集合称为森林；

## 1.8.2 二叉树

二叉树的特点是每个结点至多只有两棵子树，即二叉树中不存在度大于 2 的结点，并且，二叉树的子树有左右之分，其次序不能任意颠倒。

### 1、二叉树的性质

(1) 在二叉树中，第  $i$  层的结点总数不超过  $2^{(i-1)}$ ；

(2) 深度为  $h$  的二叉树最多有  $2^h - 1$  个结点( $h \geq 1$ )，最少有  $h$  个结点；

(3) 对于任意一棵二叉树，如果其叶结点数为  $N_0$ ，而度数为 2 的结点总数为  $N_2$ ，则  $N_0 = N_2 + 1$ ；

(4) 具有  $n$  个结点的完全二叉树的深度为  $\text{int}(\log_2 n) + 1$

(5) 有  $N$  个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系：

若  $I$  为结点编号则 如果  $I < 1$ ，则其父结点的编号为  $I/2$ ；

如果  $2*I \leq N$ ，则其左儿子（即左子树的根结点）的编号为  $2*I$ ；若  $2*I > N$ ，则无左儿子；

如果  $2*I + 1 \leq N$ ，则其右儿子的结点编号为  $2*I + 1$ ；若  $2*I + 1 > N$ ，则无右儿子。

(6) 给定  $N$  个节点，能构成  $h(N)$  种不同的二叉树。

$h(N)$  为卡特兰数的第  $N$  项。  $h(n) = C(n, 2*n) / (n+1)$ 。

(7) 设有  $i$  个枝点， $I$  为所有枝点的道路长度总和， $J$  为叶的道路长度总和  $J = I + 2i$

### 2、二叉树的抽象数据类型

```
typedef struct _bitnode {
    void* data;          /* 数据域的指针 */
    struct _bitnode* left; /* 指向左子树根结点的指针 */
    struct _bitnode* right; /* 指向右子树根结点的指针 */
} BITNODE, *PBITNODE;

/*按先序次序建立一棵二叉树，由 bitnode 返回根结点指针*/
Status CreateBiTree(BITNODE** bitnode, Status (*visit)(void* arg));

/*前序遍历一棵二叉树*/
Status PreOrderTraverse(BITNODE* bitnode, Status (*visit)(void* arg));
/*中序遍历一棵二叉树*/
Status InOrderTraverse(BITNODE* bitnode, Status (*visit)(void* arg));
/*后序遍历一棵二叉树*/
Status PostOrderTraverse(BITNODE* bitnode, Status (*visit)(void* arg));
/*层次遍历一棵二叉树*/
```

---

```

    Status LevelOrderTraverse(BITNODE* bitnode, Status (*visit)(void* arg));
    
```

### 1.8.3 模板代码

```

#include <stdio.h>
#include <stdlib.h>

typedef struct BitTreeNode
{
    int data;
    struct BitTreeNode* Lchild;
    struct BitTreeNode* Rchild;
}BTNode;

BTNode* CreateBitTree(int tdata[], int num);
void PreOrderTraverse(BTNode* tnode, void (*Visit)(BTNode* tnode));
void MidOrderTraverse(BTNode* tnode, void (*Visit)(BTNode* tnode));
void PostOrderTraverse(BTNode* tnode, void (*Visit)(BTNode* tnode));
void PrintData(BTNode* tnode);

int main(void)
{
    BTNode* troot;
    int tdata[12] = {1,2,3,4,5,6,7,8,9,10,11,12};
    troot = CreateBitTree(tdata, 3);
    if(NULL == troot) {
        printf("Create binary tree failed.\n");
        exit(0);
    }
    printf("PreOrderTraverse:\n");
    PreOrderTraverse(troot, PrintData);
    printf("MidOrderTraverse:\n");
    MidOrderTraverse(troot, PrintData);
    printf("PostOrderTraverse:\n");
    PostOrderTraverse(troot, PrintData);
    return 0;
}

/**
 * @brief 二叉树的创建
 *
 * @param [in] tdata 树中结点存放的数据
 *          [in] num 数据个数
 *
 * @return 创建好的树的根结点指针
 */
BTNode* CreateBitTree(int tdata[], int num)
{
    BTNode* tnode;
    if(num != 0) {
    
```

```

        tnode = (BTNode*)malloc(sizeof(BTNode));
        if(NULL == tnode) {
            return NULL;
        }
        tnode->data = tdata[num-1];
        tnode->Lchild = CreateBitTree(tdata, num-1);
        tnode->Rchild = CreateBitTree(tdata, num-1);
        return tnode;
    } else {
        return NULL ;
    }
}

/**
 * @brief 二叉树的前序遍历
 *
 * @param [in] tnode 以该结点为根结点进行遍历
 *          [in] Visit 对结点调用的函数指针
 *
 * @return 空
 */
void PreOrderTraverse(BTNode* tnode, void (*Visit)(BTNode* tnode))
{
    if(tnode != NULL) {
        Visit(tnode);
        PreOrderTraverse(tnode->Lchild, Visit);
        PreOrderTraverse(tnode->Rchild, Visit);
    }
}

/**
 * @brief 二叉树的中序遍历
 *
 * @param [in] tnode 以该结点为根结点进行遍历
 *          [in] Visit 对结点调用的函数指针
 *
 * @return 空
 */
void MidOrderTraverse(BTNode* tnode, void (*Visit)(BTNode* tnode))
{
    if(tnode != NULL) {
        MidOrderTraverse(tnode->Lchild, Visit);
        Visit(tnode);
        MidOrderTraverse(tnode->Rchild, Visit);
    }
}

/**
 * @brief 二叉树的后序遍历
 *
```

```

* @param [in] tnode 以该结点为根结点进行遍历
*          [in] Visit 对结点调用的函数指针
*
* @return 空
*/
void PostOrderTraverse(BTNode* tnode, void (*Visit)(BTNode* tnode))
{
    if(tnode != NULL) {
        PostOrderTraverse(tnode->Lchild, Visit);
        PostOrderTraverse(tnode->Rchild, Visit);
        Visit(tnode);
    }
}

/**
* @brief 二叉树遍历函数
*
* @param [in] tnode 对该结点调用本函数
*
* @return 空
*/
void PrintData(BTNode* tnode)
{
    printf("%d\n", tnode->data);
}

```

## 第2章 排序

### 2.1 冒泡排序

冒泡排序是交换排序中一种简单的排序方法。它的基本原理是对所有相邻记录的关键字值进行比较，如果是逆序（如果要求出从小到大的话，那么 $r[j]>r[j+1]$ 就是逆序，反之，亦然。），则将其交换，最终达到有序化。时间复杂度为 $O(n^2)$ 。

编写：曾卓敏

校核：彭文文

基本原理

冒泡排序是交换排序中一种简单的排序方法。它的基本原理是对所有相邻记录的关键字值进行比较，如果是逆序，则将其交换，通过两层循环来排序，每一趟（外层循环）就让最小的（或者最大的）交换到最前面（也就是下文中的有序区），内层循环是进行交换的，只要存在逆序的就交换，以此达到有序化。

#### 2.1.1 解题思路

（1）将整个待排序的记录序列划分成有序区和无序区。初始状态有序区为空，无序区包括所有待排序的记录。

（2）对无序区从前向后依次将相邻记录的关键字进行比较，若逆序则将其交换，从而使得关键字值小的记录向上“飘”（左移），关键字值大的记录向下“沉”（右移）。每经过一趟冒泡排序，都使无序区中关键字值最大的记录进入有序区，对于由  $n$  个记录组成的记录序列，最多经过  $n-1$  趟冒泡排序，就可以将这  $n$  个记录重新按关键字顺序排列。

#### 2.1.2 模板代码

对由  $n$  个记录组成的记录序列，最多经过  $(n-1)$  趟冒泡排序，就可以使记录序列成为有序序列，第一趟定位第  $n$  个记录，此时有序区只有一个记录；第二趟定位第  $n-1$  个记录，此时有序区有两个记录；以此类推，直到最后所有的记录都进入有序区，排序结束。

完整的冒泡排序算法如下。

```
void BubbleSort1(LineList r[], int n)
{
    int i, j;
    LineList temp;
    for (i=n-1; i>0; i--)                /*i 为每趟排序的数组最大下标值*/
        for (j=0; j<=i-1; j++)          /*一趟交换排序*/
            if(r[j].Key>r[j+1].Key)      /*若逆序*/
            {
                temp=r[j];
                r[j]=r[j+1];
                r[j+1]=temp;
            }
}
```

#### 2.1.3 经典题目

##### 1. 题目描述

已知有 10 个待排序的记录，它们的关键字序列为{43, 12, 35, 18, 26, 57, 7, 21, 43, 46}，给出冒泡排序法进行排序的过程。（两个相同的关键字 43，后面 43 用方框框上）

## 2. 分析

解：冒泡排序的过程如图所示。其中括号内表示有序区。

初始状态：	43	12	35	18	26	57	7	21	<span style="border: 1px solid black;">43</span>	46
第一趟排序结果：	12	35	18	26	43	7	21	<span style="border: 1px solid black;">43</span>	46	( 57 )
第二趟排序结果：	12	18	26	35	7	21	43	<span style="border: 1px solid black;">43</span>	( 46	57 )
第三趟排序结果：	12	18	26	7	21	35	43	( <span style="border: 1px solid black;">43</span>	46	57 )
第四趟排序结果：	12	18	7	21	26	35	( 43	<span style="border: 1px solid black;">43</span>	46	57 )
第五趟排序结果：	12	7	18	21	26	( 35	43	<span style="border: 1px solid black;">43</span>	46	57 )
第六趟排序结果：	7	12	18	21	( 26	35	43	<span style="border: 1px solid black;">43</span>	46	57 )
第七趟排序结果：	7	12	18	( 21	26	35	43	<span style="border: 1px solid black;">43</span>	46	57 )
第八趟排序结果：	7	12	( 18	21	26	35	43	<span style="border: 1px solid black;">43</span>	46	57 )
第九趟排序结果：	7	( 12	18	21	26	35	43	<span style="border: 1px solid black;">43</span>	46	57 )

### 2.1.4 扩展变型

在冒泡排序过程中，一旦发现某一趟没有进行交换操作，就表明此时待排序记录序列已经成为有序序列，冒泡排序再进行下去已经没有必要，应立即结束排序过程。

在上题中，在第六趟排序后，序列已经成为有序序列，从第七次到第九次排序就没有必要。为实现这一方法我们在循环体内设一个查看是否有记录交换的变量，在每趟比较时查看是否有交换，如果没有，则提前结束循环。

改进的冒泡排序算法如下

```

void BubbleSort2(LineList r[],int n)
{
    int i,j, exchange;          /* exchange 为标记是否交换的标识变量*/
    LineList temp;
    for (i=n-1; i>0; i--)       /*i 为每趟排序的数组最大下标值*/
    {
        exchange=0;

        for (j=0; j<=i-1; j++)  /*一趟交换排序*/
            if (r[j].Key>r[j+1].Key) /*若逆序*/
            {
                temp=r[j];
                r[j]=r[j+1];
                r[j+1]=temp;
                exchange=1;
            }
        if (exchange==0) return;
    }
}
    
```

## 2.2 插入排序

编写：曾卓敏

校核：彭文文

### 2.2.1 基本原理

直接插入排序是一种最简单的排序方法，它的基本思想是依次将记录序列中的每一个记录插入到有序段中，使有序段的长度不断地扩大。

直接插入排序算法分析：

从空间角度来看，它只需要一个辅助空间  $r[0]$ 。

从时间耗费角度来看，主要时间耗费在关键字比较和移动元素上。算法执行时间在最坏的情况下是  $O(n^2)$ 。

在这种插入过程中如果存在两个相同的数字，那么先后位置不会发生变化，所以直接插入排序是一种稳定排序方法。

#### 解题思路

有  $n$  个记录的无序序列具体的排序过程可以描述如下：

(1) 首先将待排序记录序列中的第一个记录作为一个有序段，此时这个有序段中只有一个记录。

(2) 从第 2 个记录起到最后一个记录，依次将记录和前面子序列中的记录比较，确定记录插入的位置，该位置满足，前面元素小于等于它，后面元素大于它，即可插入，如果是最大，或者最小，插入到最前面或者最后面需要单独写出来。

(3) 将记录插入到子序列中，子序列中的记录个数加 1，直至子序列长度和原来待排序序列长度一致时排序结束。

一共经过  $n-1$  趟就可以将初始序列的  $n$  个记录重新排列成按关键字值从小到大排列的有序序列。

为了防止在比较过程中数组下标的溢出，我们设一个监视哨  $r[0]$ ，即先将要比较的关键字存入监视哨  $r[0]$  中，然后再用  $r[0]$  从后向前进行比较。若  $r[0]$  小于所比较的关键字，则将该关键字向后移一位，并且继续向前比较，直到  $r[0]$  大于等于所比较的关键字时结束。因为我们是边比较边移动记录的，所以在当前比较记录的后面位置是空出来的，直接将  $r[0]$  存入即可。

### 2.2.2 模板代码

```
void InsertSort(LineList r[],int n)
{
    int i,j;
    for(i=2; i<=n; i++)          /*一共需要比较 n-1 趟*/
    {
        r[0]=r[i];              /*将 r[0]赋为监视哨*/
        j=i-1;
        while(r[0].Key<r[j].Key) /*搜索插入位置*/
        {
            r[j+1]=r[j];
            j=j-1;
        }
        r[j+1]=r[0];             /*将原来 r[i]中的记录放入第 j+1 个位置*/
    }
}
```

### 2.2.3 经典题目

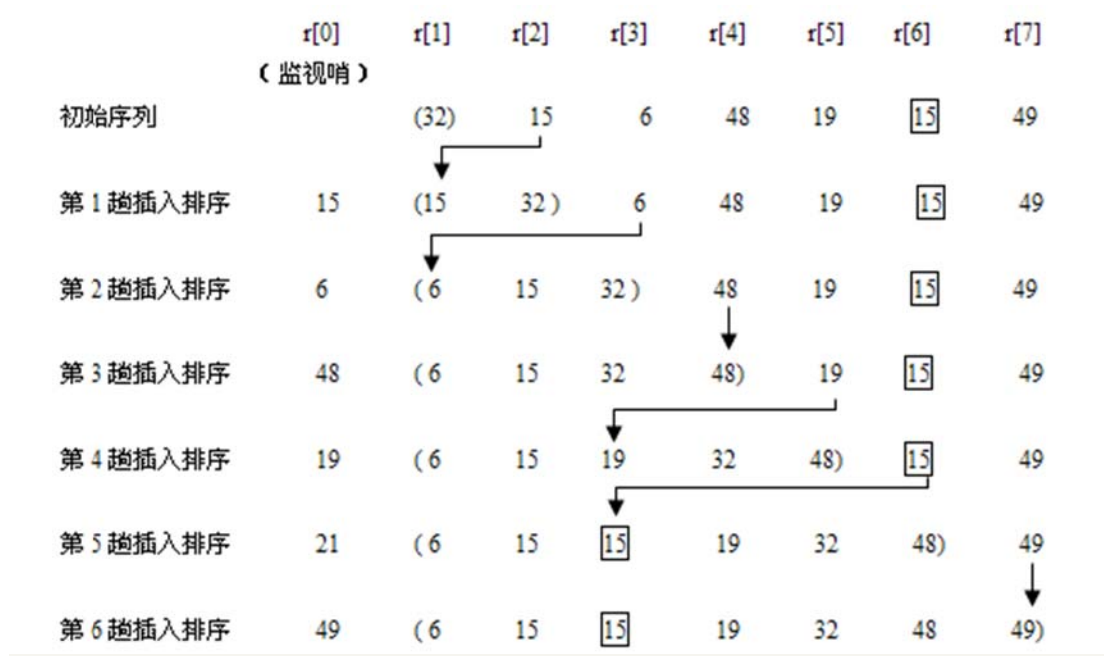
#### 1. 题目描述



设待排序的记录序列有  $n=7$  个记录，其关键字的初始序列为：{32, 15, 6, 48, 19, 15, 49}，请给出直接插入排序的过程。在序列中有两个相同关键字 15，我们用方框将后一个 15 框上加以区分。

## 2. 分析

直接插入排序过程如图所示。就是对此记录序列进行直接插入排序的过程示意图。其中括号内部的关键字为已排好序的部分。



## 2.3 归并排序

编写：姜喜朋

校核：彭文文

### 2.3.1 基本原理

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。归并排序的平均时间复杂度  $O(N\log N)$ 。

归并操作(merge)，也叫归并算法，指的是将两个已经排序的序列合并成一个序列的操作。

如：设有数列 {6, 202, 100, 301, 38, 8, 1}

初始状态： [6] [202] [100] [301] [38] [8] [1] 比较次数

i=1 [6 202] [100 301] [8 38] [1] 3

i=2 [6 100 202 301] [1 8 38] 4

i=3 [1 6 8 38 100 202 301] 4

总计： 11 次

归并操作的工作原理如下：

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置。

4.重复步骤 3 直到某一指针达到序列尾

5.将另一序列剩下的所有元素直接复制到合并序列尾

### 2.3.2 解题思路

归并排序的题目往往不是直接考查排序，而最常见的一类题目就是利用归并排序来解决逆序数问题。即求给定的一个序列它的逆序数。（在一个排列中，如果一对数的前后位置与大小顺序相反，即前面的数大于后面的数，那么它们就称为一个逆序。一个排列中逆序的总数就称为这个排列的逆序数。）此类问题可用归并排序来解，具体请看下边的例题。另外逆序数算法也可以采用树状数组来解决。

### 2.3.3 模板代码

```
void Merge(int data[], int l, int m, int r)
{
    int i, j, k;
    int *pd;
    pd = (int*) malloc ((r - l + 1) * sizeof(int));

    i = l; j = m + 1; k = 0;
    while(i <= m && j <= r)  pd[k++] = data[i] < data[j] ? data[i++] : data[j++];
    while(i <= m) pd[k++] = data[i++];
    while(j <= r) pd[k++] = data[j++];
    for(k = 0; i = l; i <= r; i++, k++) data[i] = pd[k];
    free(pd);
}

void MergeSort(int data[], int l, int r)
{
    int m;
    if(l < r) {
        m = (l + r) >> 1;
        MergeSort(data, l, m);
        MergeSort(data, m+1, r);
        Merge(data, l, m, r);
    }
}
```

### 2.3.4 经典题目

1. 题目出处/来源

POJ-2299 Ultra-QuickSort

2. 题目描述

给定一个无序的序列，它们都是由 32 位整数组成的。序列长度最大可达 500000.现在通过不断的两两交换，我们可以把这个序列排成由小到大的有序序列。请问交换的次数是多少。

3. 分析

一提起交换，我们很容易想到冒泡排序法。而此题来说，冒泡排序是符合该题的解的。只要将在序列后面的较小数不断与它前面比它大的数交换，最终较小的数浮到了前面，我们就完成了对此数的交换次数统计，如此对每一个数都进行此操作，最终我们可以得到答案。但是序列长度可达 50 万之多，如果真的使用冒泡排序算法解决此问题的话，其复杂度 $n^2$ 是难以在 7 秒内完成这么大的数据量的。因此我们需要换种算法。想想小的数不断交换变到前面，那整个不就是求序列的逆序数嘛。有了求逆序数的思路我们由此展

开。事先交待求逆序数的算法有归并排序和树状数组两种。本次我们讲述采用归并排序的方式来解决这个问题。

归并排序的解法求此问题是如果左边第  $i$  个数（编号从 0 开始）大于右边某个数  $x$ ，由于两个子序列都是有序的，那么左边序列第  $i$  个以及之后的数都大于  $x$ ，所以  $count$  要加上  $m-i+1$

最后由于 50 万数的逆序数可能非常庞大，最终的结果可能不是  $int$  所能装下的，因此我们采用  $long\ long$  这个类型来存放结果。其格式控制符在  $linux$  服务器下的 OJ 用  $\%lld$  控制，而在  $windows$  服务器下的 OJ 用  $\%I64d$  控制。

#### 4. 代码

```
#include <stdio.h>
#include <stdlib.h>
#define NN 500005

int a[NN];
long long count;

void Merge(int data[], int l, int m, int r)
{
    int i, j, k;
    int *pd;
    pd = (int*) malloc ((r - l + 1) * sizeof(int));

    i = l; j = m + 1; k = 0;
    while(i <= m && j <= r) {
        if (data[i] <= data[j]) {
            pd[k++] = data[i++];
        } else {
            pd[k++] = data[j++];
            count += m - i + 1;    /* 计算逆序数 */
        }
    }

    while(i <= m) pd[k++] = data[i++];
    while(j <= r) pd[k++] = data[j++];
    for(k = 0; i = l; i <= r; i++, k++) data[i] = pd[k];
    free(pd);
}

void MergeSort(int data[], int l, int r)
{
    if(l < r) {
        int m;
        m = (l + r) >> 1;
        MergeSort(data, l, m);
        MergeSort(data, m+1, r);
        Merge(data, l, m, r);
    }
}

int main(void)
{

```

```

int n, i;
while (scanf("%d", &n), n) {
    count = 0;
    for (i = 0; i < n; i++) {
        scanf("%d", a + i);
    }
    MergeSort(a, 0, n - 1);
    printf("%lld\n", count);
}
return 0;
}

```

## 2.4 快速排序

编写：姜喜朋

校核：彭文文

### 2.4.1 基本原理

快速排序（Quicksort）是对冒泡排序的一种改进。由 C. A. R. Hoare 在 1962 年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。快速排序的平均时间复杂度  $O(N\log N)$ 。

### 2.4.2 解题思路

设要排序的数组是  $A[0] \cdots A[N-1]$ ，首先任意选取一个数据（通常选用第一个数据）作为关键数据，然后将所有比它小的数都放到它前面，所有比它大的数都放到它后面，这个过程称为一趟快速排序。值得注意的是，快速排序不是一种稳定的排序算法，也就是说，多个相同的值的相对位置也许会在算法结束时产生变动。

一趟快速排序的算法是：

- 1) 设置两个变量 I、J，排序开始的时候：I=0，J=N-1；
- 2) 以第一个数组元素作为关键数据，赋值给 key，即  $key=A[0]$ ；
- 3) 从 J 开始向前搜索，即由后开始向前搜索（J=J-1 即 J--），找到第一个小于 key 的值  $A[j]$ ， $A[j]$  与  $A[i]$  交换；
- 4) 从 I 开始向后搜索，即由前开始向后搜索（I=I+1 即 I++），找到第一个大于 key 的  $A[i]$ ， $A[i]$  与  $A[j]$  交换；
- 5) 重复第 3、4、5 步，直到 I=J；（3,4 步是在程序中没找到时候  $j=j-1$ ， $i=i+1$ ，直至找到为止。找到并交换的时候 i，j 指针位置不变。另外当  $i=j$  这过程一定正好是 i+或 j-完成的最后令循环结束。）

示例：待排序的数组 A 的值分别是：（初始关键数据：key=49） 注意关键 key 永远不变，永远是和 key 进行比较，无论在什么位置，最后的目的就是把 key 放在中间，小的放前面大的放后面。

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
49	38	65	97	76	13	27

进行第一次交换后：27 38 65 97 76 13 49

（按照算法的第三步从后面开始找，此时：J=6）

进行第二次交换后：27 38 49 97 76 13 65

（按照算法的第四步从前面开始找 >key 的值，65>49，两者交换，此时：I=2）

进行第三次交换后：27 38 13 97 76 49 65

(按照算法的第五步将又一次执行算法的第三步从后开始找

进行第四次交换后：27 38 13 49 76 97 65

(按照算法的第四步从前面开始找大于 key 的值，97>49，两者交换，此时：I=3,J=5)

此时再执行第三步的时候就发现 I=J=3，从而结束一趟快速排序，那么经过一趟快速排序之后的结果是：27 38 13 49 76 97 65，即所有大于 key 的数全部在 49 的后面，所有小于 key (49) 的数全部在 key (49) 的前面。

### 2.4.3 模板代码

```

/**
 * @brief 快速排序
 * @param data 排序数组， l 左边界， r 右边界
 * @return NULL
 */
void QuickSort(int data[], int l, int r)
{
    int i, j;
    i = l; j = r;
    data[0] = data[i];
    while(i < j) {
        while(i < j && data[j] > data[0]) j--;
        if (i < j) data[i++] = data[j];
        while(i < j && data[i] < data[0]) i++;
        if (i < j) data[j--] = data[i];
    }
    data[i] = data[0];
    if(l < r) {
        QuickSort(data, l, i - 1);
        QuickSort(data, i + 1, r);
    }
}

```

### 2.4.4 经典题目

#### 1. 题目出处/来源

HRBUST 1552 超过一半的数字

#### 2. 题目描述

给定一个 n 元素的序列，其中存放的都是整数类型的数据。现在保证给出的序列中有一个数，这个数的个数超过整个序列元素总个数的一半，请你找出这个数是多少。

#### 3. 分析

首先，拿到此题第一反应一般都是先对序列排序，然后取其中位数，即处在数组中下标最中间的那个位置的数就是超半数的了。可以这样想，假设这超过半数的数字最小，那它肯定就从第一个位置一直占据到数组元素个数一半的位置再加 1 的地方。而假如还有比它小的数那整个序列还要右移，因此排序后能处在中间的数一定就是超过半数的。有了这个想法后，如果单纯的排序，肯定不行，因为数据量很大，有 100 万，就算是快速排序也会用  $O(n\log n)$  的时间，对此题数据量来说还是太大了。再想优化。考虑到快排中调用了一个叫 partition 的函数，该函数的作用就是取一个数作枢轴，然后将比枢轴小的数放在枢轴左边，比其大的数放到枢轴右边。关键就在这枢轴，假如有特别的情况，第一次选到的枢轴中定位后就在数组的中间，那还需要进行下边的排序吗？当然不需要，中间位置的数只要

一定，那我们的答案也就定下来了。因此我们可以只进行有限次的 `partition` 调用，来不断循找到底哪个数最终能放在中间的位置上，我们的答案也就出来了。这个算法的复杂度是  $O(n)$  的，要比 `qsort` 好多啦。本题还可以用 `hash` 和二分或者别的方法，可以自己思考。

#### 4. 代码

```
#include <stdio.h>
#include <stdlib.h>
#define NN 1000010

int a[NN];

int partition(int number[], int len, int l, int r)
{
    int i = l, j = r;
    number[0] = number[i];
    while(i < j) {
        while(i < j && number[j] > number[0]) j--;
        if (i < j) number[i++] = number[j];
        while(i < j && number[i] < number[0]) i++;
        if (i < j) number[j--] = number[i];
    }
    number[i] = number[0];
    return i;
}

int get_more_than_half_num(int number[], int len)
{
    int left, right, mid, index;
    left = 1;
    right = len;
    mid = len / 2 + 1;
    index = partition(number, len, left, right);
    while (index != mid) {
        if (index < mid) {
            left = index + 1;
        } else {
            right = index - 1;
        }
        index = partition(number, len, left, right);
    }
    return number[index];
}

int main()
{
    int T, n, i;
    scanf("%d", &T);
    while (T--) {
        scanf("%d", &n);
        for (i = 1; i <= n; i++) {
            scanf("%d", &a[i]);
        }
    }
}
```

```

        printf("%d\n", get_more_than_half_num(a, n));
    }
    return 0;
}

```

## 2.5 桶式排序（基数排序）

编写：曾卓敏      校核：彭文文

基本原理：首先定义桶，桶为一个数据容器，每个桶存储一个区间内的数。依然有一个待排序的整数序列 A，元素的最小值不小于 0，最大值不超过 K。

### 2.5.1 解题思路

假设我们有 M 个桶，第 i 个桶 Bucket[i] 存储  $i \cdot K/M$  至  $(i+1) \cdot K/M$  之间的数，有如下桶排序的一般方法：

1. 扫描序列 A，根据每个元素的值所属的区间，放入指定的桶中(顺序放置)。
2. 对每个桶中的元素进行排序，什么排序算法都可以，例如快速排序。
3. 依次收集每个桶中的元素，顺序放置到输出序列中。

对该算法简单分析，如果数据是期望平均分布的，则每个桶中的元素平均个数为  $N/M$ 。如果对每个桶中的元素排序使用的算法是快速排序，每次排序的时间复杂度为  $O(N/M \cdot \log(N/M))$ 。则总的时间复杂度为  $O(N) + O(M) \cdot O(N/M \cdot \log(N/M)) = O(N + N \cdot \log(N/M)) = O(N + N \cdot \log N - N \cdot \log M)$ 。当 M 接近于 N 是，桶排序的时间复杂度就可以近似认为是  $O(N)$  的。就是桶越多，时间效率就越高，而桶越多，空间却就越大，由此可见时间和空间是一个矛盾的两个方面。

桶中元素的顺序放入和顺序取出是有必要的，因为这样可以确定桶排序是一种稳定排序算法，配合基数排序是很好用的。

### 2.5.2 模板代码

```

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
using namespace std;
void BucketSort(int *A, int *B, int N, int K)
{
    int *C = new int[K+1];
    int i, j, k;
    memset(C, 0, sizeof(int) * (K+1));
    for (i=1; i<=N; i++) //把 A 中的每个元素按照值放入桶中
        C[A[i]]++;
    for (i=j=1; i<=K; i++, j=k) //统计每个桶元素的个数，并输出到 B
        for (k=j; k<j+C[i]; k++)
            B[k]=i;
}
int main()
{
    int *A, *B, N=1000, K=1000, i;

```

```

A=new int[N+1];
B=new int[N+1];
for (i=1;i<=N;i++)
    A[i]=rand()%K+1; //生成 1..K 的随机数
BucketSort(A,B,N,K);
for (i=1;i<=N;i++)
    printf("%d ",B[i]);
return 0;
}

```

### 2.5.3 经典题目

#### 1. 题目出处/来源

Hrbustoj 1030

#### 2. 题目描述

本题就是给  $n$  个数，然后进行排序，用 `sort` 等排序也可以，但是由于数的大小范围小，所以用桶排序或者基数排序则会很快。

#### 3. 分析

可以用此题练习基数排序或者桶排序的思维。

#### 4. 代码

以下是基数排序代码：

```

#include<stdio.h>
int power[100002],POWER[100002];
int Counting_Sort(int a[],int b[],int l){
    int j;
    int c[100002]={0};
    for(j=0;j<l;j++)
        c[a[j]]++;
    for(j=1;j<100002;j++)
        c[j]+=c[j-1];
    for(j=l-1;j>=0;j--){
        b[c[a[j]]-1]=a[j];
        c[a[j]]--;
    }
}
int main (){
    int i,n;
    scanf("%d",&n);
    for(i=0;i<n;i++){
        scanf("%d",&power[i]);
    }
    Counting_Sort(power,POWER,n);
    for (i=0;i<n;i++){
        printf("%d\n",POWER[i]);
    }
}

```

#### 5. 思考与扩展：参看《算法导论》P100~104 基数排序和桶排序。



## 第3章 基本算法

### 3.1 二分查找

编写：包春志      校核：彭文文

基本原理

算法名称：二分查找

算法类别：分治法，查找

算法要求：1.必须采用顺序存储结构 2.必须按关键字大小有序排列。

算法描述：对待查找有序表进行平均分区，以中间值为标识符。判断待查找关键字 KEY 与其大小，进行细分，继续查找，直到待查找表结束或者找到为止。

算法复杂度：时间复杂度----- $\log(2)N$

算法优点：速度快，次数少，平均性能好

算法缺点：要求待查找表是有序表，插入困难

算法适用情景：不经常变动而查找频繁的有序表

#### 3.1.1 模板代码

```
int binary_search(int a[],int left,int right,int key )
{
    while(left<right)
    {
        int mid=(left+right)>>1;
        if(key>a[mid]) left=mid+1;
        else right=mid;
    }
    return left;
}
```

#### 3.1.2 经典题目

1. 题目出处/来源

HRBUST-1039

2. 题目描述

前段时间，某省发生干旱，B 山区的居民缺乏生活用水，现在需要从 A 城市修一条通往 B 山区的路。假设有 A 城市通往 B 山区的路由 m 条连续的路段组成，现在将这 m 条路段承包给 n 个工程队 ( $n \leq m \leq 300$ )。为了修路的便利，每个工程队只能分配到连续的若干条路段（当然也可能只分配到一条路段或未分配到路段）。假设每个工程队修路的效率一样，即每修长度为 1 的路段所需的时间为 1。现在给出路段的数量 m，工程队的数量 n，以及 m 条路段的长度（这 m 条路段的长度是按照从 A 城市往 B 山区的方向依次给出，每条路段的长度均小于 1000），需要你计算出修完整条路所需的最短的时间（即耗时最长的工程队所用的时间）。

**Input**

第一行是测试样例的个数 T，接下来是 T 个测试样例，每个测试样例占 2 行，第一行是路段的数量 m 和工程队的数量 n，第二行是 m 条路段的长度。

**Output**

对于每个测试样例，输出修完整条路所需的最短的时间。

**Sample Input**

2

```

4 3
100 200 300 400
9 4
250 100 150 400 550 200 50 700 300
    
```

### Sample Output

```

400
900
    
```

### 3. 分析

本题可以用动态规划，但是时间很慢，而二分会很快。想一下，最短的路就是每  $m$  条路的最大的值，而如果  $n=1$  的时候，也就是只有一工程队，那么必须修这  $m$  段路的总长，所以就找到了二分上下界，一一枚举，然后用 `check` 函数检查这段距离是否满足  $n$  个工程队修  $m$  段路。

### 4. 代码

```

#include <stdio>
int c[301],n,m;
int check(int u){//check 函数检查是否合法。
    int temp=0,cnt1=1;
    for(int i=0;i<n;i++){
        if(temp+c[i]<=u){
            temp+=c[i];
        }
        else{
            temp=c[i];
            cnt1++;
            if(cnt1>m) return -1;
        }
    }
    return -2;
}
int find(int low,int high){
    int mid;
    while(low<high){
        mid=(high+low)/2;
        int flag=check(mid);
        if(flag==-1)//如果此时 mid 的值太小
            low=mid+1;
        else if(flag==-2) //如果此时 mid 的值太大
            high=mid;
    }
    return low;
}
int main(){
    int T;
    while(scanf("%d",&T)!=EOF)
        while(T--){
            int cnt=0,_max=0;
            scanf("%d%d",&n,&m);
            for(int i=0;i<n;i++){
                scanf("%d",&c[i]);
                cnt+=c[i];
            }
        }
    }
    
```

```

        if(c[i]>_max) _max=c[i];
    }
    // _max 是下届, cnt 是上界
    printf("%d\n",find(_max,cnt));
}
}

```

## 3.2 模拟

编写：包春志

校核：曾卓敏

### 3.2.1 基本原理

模拟算法是 ACM 竞赛中经常使用的算法，其算法经常与其他算法一起使用，是一种实现手段。模拟时候最大的阻碍就是各种特殊情况的判断，想不到一种特殊条件就是 wa 下去，所以做模拟题一定要细心，而且要注意代码的整洁性。

做这种题主要是根据思路去实现代码。解题思路

### 3.2.2 解题思路

仔细看题，然后根据思路写代码。

注意要把各种情况都考虑清楚了。尤其是边缘情况，也就是特殊情况。还有注意数据结构的合理使用，有些题如果用对了数据存储方式就很容易模拟了。

### 3.2.3 经典题目

#### 3.2.3.1 题目 1

1. 题目出处/来源

Hrbustoj 1178

2. 题目描述

Description

实现两个分数之间的加减法。

Input

输入包括多组测试用例

每行是 "a/b-c/d"，或 "a/b+c/d" 的形式。

其中 a, b, c, d 是 0-9 的整数。

输入数据保证合法。

Output

输出分数结果。

注意输出没有多余符号，若结果为正，不需要 "+" 号。

结果应为最简形式，例如结果应为 1/2 而非 2/4、2 而非 2/1。

Sample Input

1/4-1/2

1/3-1/3

Sample Output

-1/4

0

3. 分析

如果是我们手动做这题的话，就是把分母通分了，然后把分子乘以通分的那个数再计

算，最后把结果约分。

所以代码模拟手动过程，先根据求最小公倍数的函数(函数原理是最小公倍数 =  $(a*b) \div \text{最大公约数}$ )求出两个数的最小公倍数，然后就可以通分了。

然后根据最大公约数的函数求出最大公约数，(最大公约数的函数实现原理是欧几里得算法)，这样就可以给通分后完成了加或减的分式进行约分。

其中有两个要注意的地方就是 ①. 如果是  $5 \div 1$  答案是 1 所以输出应该是 5

②. 如果是  $5 \div (-3)$  输出的时候应该是 -5/3

#### 4. 代码

```
#include<stdio.h>
int gcd(int a,int b){
    if(b==0) return a;
    return gcd(b,a%b);
}
int lcm(int a,int b){
    int c=gcd(a,b);
    return a*b/c;
}
int main(){
    int a,b,c,d;
    char ch;
    while(scanf("%d/%d%c%d/%d",&a,&b,&ch,&c,&d)!=EOF){
        int m=lcm(b,d);
        int n;
        if(ch=='+') n=a*(m/b)+c*(m/d);
        else n=a*(m/b)-c*(m/d);
        if(n==0) printf("0\n");
        else{
            int t=gcd(m,n);
            n=n/t;m=m/t;
            if(m<0) m=-m,n=-n;
            if(m==1) printf("%d\n",n);
            else
                printf("%d/%d\n",n,m);
        }
    }
    return 0;
}
```

### 3.2.3.2 题目 2

#### 1. 题目来源

Hrbust OJ 1371

#### 2. 题目描述

##### Description

Leyni has very good skills, he is currently writing an Operating System called "Leyni OS", however, this system supports only two commands, namely, "cd" and "pwd".

This operating system's root directory is denoted as "/", other areas are hierarchical tree structures similar to our current operating system. The directory name contains only lowercase letters. Each directory contains a parent directory, denoted as "..".

The "cd" command has one parameter, which is a path. This path consists of several "/" and several directory names. (Directory name may be "..", meaning its parent directory)

If this path begins with "/", it will change the current directory to this path; if the path

begins with a directory name, it will be entering a sub directory of the current directory according to this path, on the basis of the current directory.

The "pwd" command contains no argument, this command will output the current directory's path, which contains no "..".

Initially, the current directory is "/". It is guaranteed that the input data does not attempt to enter the root directory's parent directory. For more information, see the details in the sample.

Input

There are multiple test cases. The first line of input is an integer T indicating the number of test cases. Then T test cases follow.

For each test case:

Line 1. This line contains an integer n ( $1 \leq n \leq 20$ ) indicating the number of commands. Then n lines follow.

Line 2..1+n. Each line contains a command either "pwd" or "cd" followed by a single space and a parameter. The length of the parameter is in the range [1, 200].

Output

For each test case:

Line i. Output the path of the current directory ending with a slash if the command is "pwd".

Sample Input

```
2
7
pwd
cd /xxx/yyy
pwd
cd ..
pwd
cd xxx/../yyy
pwd
4
```

```
cd /aaa/bbb
pwd
cd ../aaa/bbb
pwd
```

Sample Output

```
/
/xxx/yyy/
/xxx/
/xxx/yyy/
/aaa/bbb/
/aaa/aaa/bbb/
```

### 3. 分析

这道题大意就是根据意思模拟 命令行比较直接的方法是按照题目意思，读取字符串，有一个指针标记当前文件夹位置。如果是返回上一个文件夹就把指针指到字符串中上一个文件夹的位置。但是这样子很容易出错，所以可以用一个二维数组来存，这样如果是返回上一个文件夹就直接用指针指到第一维数组中的上一个位置，这样就可以很好根据命令行显示路径了。

### 4. 代码

```
#include <stdio>
#include <cstring>
```

```
#include <iostream>
#include <algorithm>
using namespace std;
int top;
char st[32][256];
int main(){
    int nTest;
    scanf("%d", &nTest);
    while (nTest--){
        int n;
        scanf("%d", &n);
        top = 0;
        strcpy(st[0], "");
        for (int i = 0; i < n; ++i) {
            char cmd[256], path[256];
            scanf("%s", cmd);
            if (0 == strcmp(cmd, "pwd")) {
                for (int i = 0; i <= top; ++i) {
                    printf("%s/", st[i]);
                }
                puts("");
            } else {
                char dir[256];
                int l = 0;
                scanf("%s", path);
                for (int i = 0; ; ++i) {
                    if ('/' != path[i] && path[i]) {
                        dir[l++] = path[i];
                    } else {
                        dir[l] = 0;
                        if (l == 0) {
                            top = 0;
                        } else if (0 == strcmp("../", dir)) {
                            if (top) {
                                --top;
                            }
                        } else {
                            ++top;
                            strcpy(st[top], dir);
                        }
                        l = 0;
                    }
                    if (path[i] == 0) break;
                }
            }
        }
    }
    return 0;
}
```

### 3.3 枚举

编写：包春志

校核：孟祥凤

基本原理

枚举也称穷举，就是一种基本的算法思想，就是按问题本身的性质，一一列举出该问题所有可能的解，并在逐一系列举的过程中，检验每个可能解是否是问题的真正解，若是，我们采纳这个解，否则抛弃它。在列举的过程中，既不能遗漏也不应重复。通过生活实例，理解枚举算法的定义，找出枚举算法的关键步骤及

注意点：

1. 在枚举算法中往往把问题分解成二部分：（1）一一列举：这是一个循环结构。要考虑的问题是如何设置循环变量、初值、终值和递增值。循环变量是否参与检验。（要强调本算法的主要是利用计算机的运算速度快这一特点，不必过多地去做算法优化工作。）（2）检验：这是一个分支结构。要考虑的问题是检验的对象是谁？逻辑判断后的二个结果该如何处理？

2. 分析出以上二个核心问题后，再合成：要注意循环变量与判断对象是否是同一个变量。

3. 该算法的输入和输出处理：输入：大部分情况下是利用循环变量来代替。输出：一般情况下是判断的一个分支中实现的。用循环结构实现一一列举的过程，用分支结构实现检验的过程，理解枚举算法流程图的基本框架。

问题求解的目标就是确定这些变量的值。根据问题的描述和相关的知识，能为这些变量分别确定一个大概的取值范围。在这个范围内对变量依次取值，判断所取的值是否满足数学模型中的条件，直到找到(全部)符合条件的值为止。这种解决问题的方法称作“枚举”。例如“求小于 N 的最大素数”。数学模型是：一个整型变量 n，满足(1) n 不能被 [2, n] 中的任意一个素数整除；(2) n 与 N 之间没有素数。利用已有的知识，能确定 n 的大概取值范围  $\{2, 2^{i+1}-1, 2^{i+1}, N\}$ 。在这个范围内从小到大依次取值，如果 n 不能被 [2, n] 中的任意一个素数整除，则满足条件(1)。在这个范围内找到的最后一个素数也一定满足条件(2)，即：问题的解。

#### 3.3.1 解题思路

枚举行遍所有的可能，从问题可能的解的集合中一一枚举各元素。实现的方法有很多：多重循环（for 循环，while 循环）这样一来效率就会较低，解决办法是：对问题加以分析，减少循环次数和重数；寻找更好的方法。合理的顺序和剪枝可以提高效率。

#### 3.3.2 模板代码

此专题没有固定模版，就是例举所有情况。

#### 3.3.3 经典题目

##### 3.3.3.1 题目 1

1. 题目出处/来源

POJ-2965

2. 题目描述

Description

The game “The Pilots Brothers: following the stripy elephant” has a quest where a player needs to open a refrigerator.

There are 16 handles on the refrigerator door. Every handle can be in one of two states: open or closed. The refrigerator is open only when all handles are open. The handles are represented as

a matrix 4x4. You can change the state of a handle in any location  $[i, j]$  ( $1 \leq i, j \leq 4$ ). However, this also changes states of all handles in row  $i$  and all handles in column  $j$ .

The task is to determine the minimum number of handle switching necessary to open the refrigerator.

Input

The input contains four lines. Each of the four lines contains four characters describing the initial state of appropriate handles. A symbol "+" means that the handle is in closed state, whereas the symbol "-" means "open". At least one of the handles is initially closed.

Output

The first line of the input contains  $N$  – the minimum number of switching. The rest  $N$  lines describe switching sequence. Each of the lines contains a row number and a column number of the matrix separated by one or more spaces. If there are several solutions, you may give any one of them.

Sample Input

```
--+--
----
----
--+--
```

Sample Output

```
6
1 1
1 3
1 4
4 1
4 3
4 4
```

### 3. 分析

先看一个简单的问题,如何把 '+' 变成 '-' 而不改变其他位置上的状态?

答案是将该位置  $(i, j)$  及位置所在的行  $(i)$  和列  $(j)$  上所有的 handle 更新一次。

结果该位置被更新了 7 次,相应行  $(i)$  和列  $(j)$  的 handle 被更新了 4 次,剩下的被更新了 2 次。

被更新偶数次的 handle 不会造成最终状态的改变。

因此得出高效解法,在每次输入碰到 '+' 的时候, 计算所在行和列的需要改变的次数

当输入结束后,遍历数组,所有为奇数的位置则是操作的位置,而奇数位置的个数之和则是最终的操作次数。

注: 该题不会有 "Impossible" 的情况。

### 4. 代码

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    int gird[4][4];
    char GIRD[4][4];
    string abc[4];
    int i,j,k;
    int count;
    while(cin>>abc[0]){
        cin>>abc[1];
        cin>>abc[2];
        cin>>abc[3];
```



```

        for(i=0; i<4; ++i)
            for(j=0; j<4; ++j)
                gird[i][j]=0;
        for(i=0; i<4; ++i)
            for(j=0; j<4; ++j){
                GIRD[i][j]=abc[i][j];
                if(GIRD[i][j]=='+'){
                    gird[i][j]++;
                    for(k=0; k<4; ++k){
                        gird[i][k]++;
                        gird[k][j]++;
                    }
                }
            }
        }
        count=0;
        for(i=0; i<4; ++i)
            for(j=0; j<4; ++j){
                if(gird[i][j]%2==1) count++;
            }
        cout<<count<<endl;
        for(i=0; i<4; ++i)
            for(j=0; j<4; ++j){
                if(gird[i][j]%2==1) cout<<i+1<<" "<<j+1<<endl;
            }
        }
        return 0;
    }
}

```

### 3.4 贪心

编写：孟祥凤

校核：彭文文

#### 3.4.1 基本原理

顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解。如单源最短路径问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

从问题的某一个初始解出发，向给定的目标递推。推进的每一步不是依据某一固定的递推式，而是做一个当时看似最佳的贪心选择，不断地将问题实例归纳为更小的相似的问题，并期望通过所做的局部最优选择产生出一个全局最优解。

用贪心算法求解的问题中看到这类问题一般具有 2 个重要的性质：贪心选择性质和最优子结构性质。

##### 1. 贪心选择性质——可通过做局部最优(贪心)选择来达到全局最优解

贪心策略通常是自顶向下做的。第一步为一个贪心选择，将原问题变成一个相似的、但规模更小的问题，而后的每一步都是当前看似最佳的选择。这种选择可能依赖于已作出的所有选择，但不依赖于有待于做的选择或子问题的解。从求解的全过程来看，每一次贪心选择都将当前问题归纳为更小的相似子问题，而每一个选择都仅做一次，无重复回溯

过程,因此贪心法有较高的时间效率。

## 2.最优子结构——问题的最优解包含了子问题的最优解

贪心选择和最优子结构:整体的最优解可通过一系列局部最优解达到.每次的选择可以依赖以前作出的选择,但不能依赖于后面的选择,最优子结构:问题的整体最优解中包含着它的子问题的最优解。

当一个问题的最优解包含其子问题的最优解时,称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

### 3.4.2 解题思路

将问题的求解过程看作是一系列选择,每次选择一个输入,每次选择

都是当前状态下的最好选择(局部最优解).每作一次选择后,所求问题会简化为一个规模更小的子问题.从而通过每一步的最优解逐步达到整体的最优解

动态规划算法通常以自底向上的方式解各子问题,而贪心算法则通常以自顶向下的方式进行,以迭代的方式作出相继的贪心选择,每作一次贪心选择就将所求问题简化为规模更小的子问题。

### 3.4.3 模板代码

由于贪心属于思想类算法,无固定的模板。具体解题方式需依题而行。

### 3.4.4 经典题目

#### 1. 题目出处/来源

HDU 2037 今年暑假不 AC (活动安排问题)

#### 2. 题目描述

##### Problem Description

作为球迷,一定想看尽量多的完整的比赛,当然,作为新时代的好青年,你一定还会看一些其它的节目,比如新闻联播(永远不要忘记关心国家大事)、非常 6+7、超级女生,以及王小丫的《开心辞典》等等,假设你已经知道了所有你喜欢看的电视节目的转播时间表,你会合理安排吗?(目标是能看尽量多的完整节目)

##### Input

输入数据包含多个测试实例,每个测试实例的第一行只有一个整数  $n(n \leq 100)$ ,表示你喜欢看的节目的总数,然后是  $n$  行数据,每行包括两个数据  $T_i_s, T_i_e$  ( $1 \leq i \leq n$ ),分别表示第  $i$  个节目的开始和结束时间,为了简化问题,每个时间都用一个正整数表示。 $n=0$  表示输入结束,不做处理。

##### Output

对于每个测试实例,输出能完整看到的电视节目的个数,每个测试实例的输出占一行。

##### Sample Input

```
12
1 3
3 4
0 7
3 8
15 19
15 20
10 15
8 18
6 12
5 10
```

4 14

2 9

0

Sample Output

5

### 3. 分析

一维数组里保存的就是以当前节目作为开始，最多能完整地看多少个不同的节目。很明显，播出时间最晚的节目只是能 1。我采取从后往前的规划方法。

这样，当循环到  $i$  时，能保证数组里  $D[i+1] > D[n-1]$  保存的都是最优解。

所以让  $j$  从  $i+1$  到  $n-1$  循环，找出看完第  $i$  个节目后最多还能看的节目数  $max$ 。(不要忘了判断能否完整收看哦)把  $max+1$  保存到  $D[i]$ 里。如此下去直到结束。

### 4. 代码

```
#include <stdio.h>
#include <stdlib.h>
struct c
{
    int x;
    int y;
    int ord;
} d[100];
int cmp(const struct c *a, const struct c *b)
{
    if ((*a).x == (*b).x)
        return (*a).y - (*b).y;
    else
        return (*a).x - (*b).x;
}
int main(void)
{
    int i, j, n, max;
    while (scanf("%d", &n), n)
    {
        for (max = i = 0; i < n; i++)
        {
            scanf("%d%d", &d[i].x, &d[i].y);
            d[i].ord = 1;
        }
        qsort(d, n, sizeof(struct c), cmp);
        d[n-1].ord = 1;
        for (i = n - 2; i >= 0; i--)
        {
            for (j = i + 1; j < n; j++)
            {
                if (d[i].y <= d[j].x && d[i].ord < d[j].ord + 1)
                    d[i].ord = d[j].ord + 1;
            }
            if (max < d[i].ord)
                max = d[i].ord;
        }
    }
}
```

```

        printf("%d\n", max);
    }

    return 0;
}

```

### 3.4.5 扩展变型

POJ 1700 Crossing River（过河问题）

POJ 3253 Fence Repair（哈夫曼树）

## 3.5 递归

编写：孟祥凤      校核：彭文文

直接或间接地调用自身的算法称为递归算法。

用函数自身给出定义的函数称为递归函数。

让我们来看看计算  $n$  的阶乘的计算机程序的写法,很直接地我们会用一个循环语句将  $n$  以下的数都乘起来:

```

int n, res = 1;
for(int i = 2; i <= n; i++)
    res *= i;
printf(“%d 的阶乘是%d\n”, n, res);

```

因为  $n$  的阶乘定义为  $n$  乘以  $n-1$  的阶乘,所以还可以用下面的方法来求  $n$  的阶乘:

```

int factorial(int n)
{
    if(n <= 0) return (-1);
    if(n == 1) return 1;
    else return n*factorial(n - 1);
}

```

上面这两种实现方式体现了两种不同的解决问题的思想方法。第一种通过一个循环语句来计算阶乘,其前提是了解阶乘的计算过程,并用语句把这个计算过程模拟出来。

第二种解决问题的思想是不直接找到计算  $n$  的阶乘的方法,而是试图找到  $n$  的阶乘和  $n-1$  的阶乘的递推关系,通过这种递推关系把原来问题缩小成一个更小规模的同类问题,并延续这一缩小规模的过程,直到在某一规模上,问题的解是已知的。这样一种解决问题的思想我们称为递归的思想。

### 3.5.1 解题思路

将待求解问题的解看作输入变量  $x$  的函数  $f(x)$ ,通过寻找函数  $g$ ,使得  $f(x) = g(f(x-1))$ ,并且已知  $f(0)$  的值,就可以通过  $f(0)$  和  $g$  求出  $f(x)$  的值。这样一个思想也可以推广到多个输入变量  $x, y, z$  等,  $x-1$  也可以推广到  $x - x1$ ,只要递归朝着出口的方向走就可以了。

### 3.5.2 模板代码

此类题目也无固定模板代码。递归函数可参考深度优先搜索部分大体模型。

### 3.5.3 经典题目

1. 题目出处/来源  
POJ 1664 放苹果
2. 题目描述

把  $M$  个同样的苹果放在  $N$  个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？（用  $K$  表示）注意：5, 1, 1 和 1, 5, 1 是同一种分法。

输入  $M$  和  $N$ ，且  $1 \leq M, N \leq 10$ 。输出相应的  $K$ 。例如  $M=7, N=3$  时， $K=8$ 。

### 3. 分析

所有不同的摆放方法可以分为两类：至少有一个盘子空着和所有盘子都不空。我们可以分别计算这两类摆放方法的数目，然后把它们加起来。对于至少空着一个盘子的情况，则  $N$  个盘子摆放  $M$  个苹果的摆放方法数目与  $N-1$  个盘子摆放  $M$  个苹果的摆放方法数目相同。对于所有盘子都不空的情况，则  $N$  个盘子摆放  $M$  个苹果的摆放方法数目等于  $N$  个盘子摆放  $M-N$  个苹果的摆放方法数目。我们可以据此来用递归的方法求解这个问题。

设  $f(m, n)$  为  $m$  个苹果， $n$  个盘子的放法数目，则先对  $n$  作讨论，如果  $n > m$ ，必定有  $n-m$  个盘子永远空着，去掉它们对摆放苹果方法数目不产生影响；即  $\text{if}(n > m) f(m, n) = f(m, m)$ 。当  $n \leq m$  时，不同的放法可以分成两类：即有至少一个盘子空着或者所有盘子都有苹果，前一种情况相当于  $f(m, n) = f(m, n-1)$ ；后一种情况可以从每个盘子中拿掉一个苹果，不影响不同放法的数目，即  $f(m, n) = f(m-n, n)$ 。总的放苹果的放法数目等于两者的和，即  $f(m, n) = f(m, n-1) + f(m-n, n)$ 。整个递归过程描述如下：

```
int f(int m, int n){
    if(n == 1 || m == 0) return 1;
    if(n > m) return f(m, m);
    return f(m, n-1) + f(m-n, n);
}
```

出口条件说明：当  $n=1$  时，所有苹果都必须放在一个盘子里，所以返回 1；当没有苹果可放时，定义为 1 种放法；递归的两条路，第一条  $n$  会逐渐减少，终会到达出口  $n=1$ ；第二条  $m$  会逐渐减少，因为  $n > m$  时，我们会  $\text{return } f(m, m)$  所以终会到达出口  $m=0$ 。

### 4. 代码

```
#include <stdio.h>
int count(int x, int y){
    if(y == 1 || x == 0)
        return 1;
    if(x < y)
        return count(x, x);
    return count(x, y-1) + count(x-y, y);
}
int main(){
    int t, m, n;
    scanf("%d", &t);
    for(int i = 0; i < t; i++){
        scanf("%d%d", &m, &n);
        printf("%d\n", count(m, n));
    }
}
```

## 3.5.4 扩展变型

POJ 2083 Fractal （递归画图形）

## 3.6 递推

编写：孟祥凤

校核：彭文文

### 3.6.1 基本原理

给定一个数的序列  $H_0, H_1, \dots, H_n, \dots$  若存在整数  $n_0$ ，使当  $n > n_0$  时，可以用等号(或大于号、小于号)将  $H_n$  与其前面的某些项  $H_i (0 \leq i < n)$  联系起来，这样的式子就叫做递推关系。

一个数列的第 0 项为 0，第 1 项为 1，以后每一项都是前两项的和，这个数列就是著名的斐波那契数列，求斐波那契数列的第  $N$  项。

由问题，可写出递推方程

$F[0] := 1; F[1] := 2;$

For  $i := 2$  to  $N$  do

$F[i] := F[i - 1] + F[i - 2];$

从这个问题可以看出，在计算斐波那契数列的每一项目时，都可以由前两项推出。这样，相邻两项之间的变化有一定的规律性，我们可以将这种规律归纳成如下简捷的递推关系式： $F[n] = g(F[n-1])$ ，这就在数的序列中，建立起后项和前项之间的关系。然后从初始条件（或是最终结果）入手，按递推关系式递推，直至求出最终结果（或初始值）。很多问题就是这样逐步求解的。

### 3.6.2 解题思路

对一个试题，我们要是能找到后一项与前一项的关系并清楚其起始条件（或最终结果），问题就可以递推了，接下来便是让计算机一步步了。让高速的计算机从事这种重复运算，真正起到“物尽其用”的效果。那么解题的重点就是：如何建立递推关系。

### 3.6.3 模板代码

此类题目无固定模板代码。

### 3.6.4 经典题目

#### 1. 题目出处/来源

Hrbust 走台阶问题（斐波那契数列）

#### 2. 题目描述

有一楼梯共  $M$  级，刚开始时你在第一级，若每次只能跨上一级或二级，要走上第  $M$  级，共有多少种走法？

例如走到第二级有 1 种方法，即为从第一级走一步走到第二级，走到第三级有 2 种方法，即为①走一步，但是跨两级②走两步，每步一级。

#### 3. 分析

由题目可知，每次只能走一级或两级。

因此从第一级走上第二级只能走一步，只有 1 种走法。

从第一级走上第三级，可以从第一级直接走两步，也可以从第二级走一步。有 2 种走法，走上第  $n$  级，可以从第  $n-1$  级走一步上来，也可以从第  $n-2$  级走两步上来。即：

$$f(2) = 1$$

$$f(3) = 2$$

$$f(n) = f(n-1) + f(n-2) \quad (n > 3) \text{ 是一个斐波那契函数。}$$

注意：数值可能很大，用 `unsigned long` 可能溢出。要用 `__int64`(VC++) 或 `long long`(GCC)。

#### 4. 代码

```
#include <stdio.h>
int main(){
    int i, n;
    __int64 m[41] = {0, 1};
    for (i = 2; i < 41; i++)
        m[i] = m[i-1] + m[i-2];

    scanf("%d", &n);
    while (n-- && scanf("%d", &i))
        printf("%I64d\n", m[i]);

    return 0;
}
```

#### 5. 思考与扩展

刚开始时你在第一级，若每次只能跨上一级或二级或三级，要走上第  $M$  级，共有多少种走法？

### 3.6.5 扩展变型

POJ 3070 Fibonacci（矩阵乘法求解斐波那契数列）

HDU 2048（错排公式）

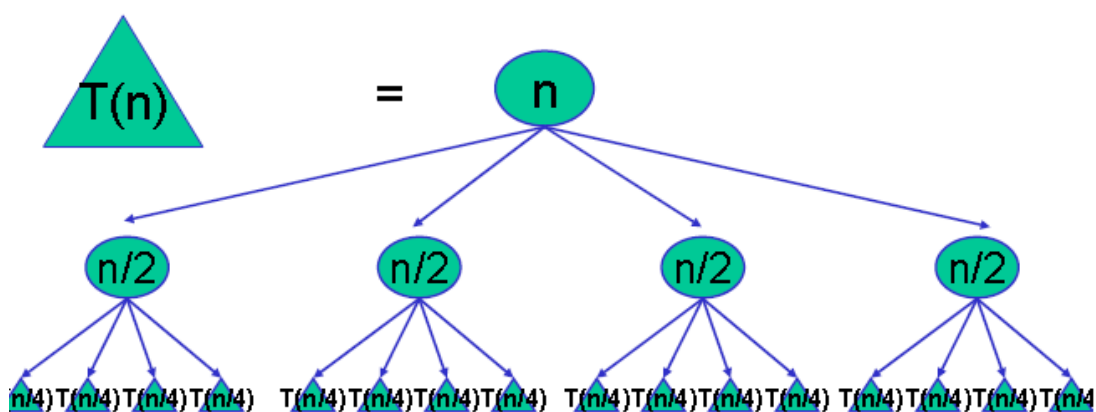
## 3.7 分治

编写：孟祥凤

校核：彭文文

### 3.7.1 基本原理

将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。



将要求解的较大规模的问题分割成  $k$  个更小规模的子问题。对这  $k$  个子问题分别求解。如果子问题的规模仍然不够小，则再划分为  $k$  个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。

### 3.7.2 解题思路

将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

### 3.7.3 模板代码

无固定模板代码。

### 3.7.4 经典题目

#### 1. 题目出处/来源

HRBUST 1215 一元三次方程求解

#### 2. 题目描述

有形如： $ax^3+bx^2+cx+d=0$  这样的一个一元三次方程。给出该方程中各项的系数(a, b, c, d 均为实数)，并约定该方程存在三个不同实根(根的范围在-100 至 100 之间)，且根与根之差的绝对值 $\geq 1$ 。要求由小到大依次在同一行输出这三个实根(根与根之间留有空格)，并精确到小数点后 4 位。

提示：记方程  $f(x)=ax^3+bx^2+cx+d$ ，若存在 2 个数  $x_1$  和  $x_2$ ，且  $x_1 < x_2$ ， $f(x_1) \cdot f(x_2) < 0$ ，则在  $(x_1, x_2)$  之间一定有一个根。

样例

输入：1    -5    -4    20

输出：-2.00    2.00    5.00

#### 3. 分析

如果精确到小数点后两位，可用简单的枚举法：将  $x$  从-100.00 到 100.00（步长 0.01）逐一枚举，得到 20000 个  $f(x)$ ，取其值与 0 最接近的三个  $f(x)$ ，对应的  $x$  即为答案。而题目已改成精度为小数点后 4 位，枚举算法时间复杂度将达不到要求。

直接使用求根公式，极为复杂。加上本题的提示给我们以启迪：采用二分法逐渐缩小根的范围，从而得到根的某精度的数值

当已知区间  $(a,b)$  内有一个根时，用二分法求根，若区间  $(a,b)$  内有根，则必有  $f(a) \cdot f(b) < 0$ 。重复执行如下的过程：

(1) 若  $a+0.0001 > b$  或  $f((a+b)/2)=0$ ，则可确定根为  $(a+b)/2$  并退出过程；

(2) 若  $f(a) \cdot f((a+b)/2) < 0$ ，则由题目给出的定理可知根在区间  $(a, (a+b)/2)$  中，故对区间重复该过程；

(3) 若  $f(a) \cdot f((a+b)/2) > 0$ ，则必然有  $f((a+b)/2) \cdot f(b) < 0$ ，根在  $((a+b)/2, b)$  中，对此区间重复该过程。

执行完毕，就可以得到精确到 0.0001 的根。

求方程的所有三个实根

所有的根的范围都在-100 至 100 之间，且根与根之差的绝对值 $\geq 1$ 。因此可知：在  $[-100, -99]$ 、 $[-99, -98]$ 、……、 $[99, 100]$ 、 $[100, 100]$  这 201 个区间内，每个区间内至多只能有一个根。即：除区间  $[100, 100]$  外，其余区间  $[a, a+1]$ ，只有当  $f(a)=0$  或  $f(a) \cdot f(a+1) < 0$  时，方程在此区间内才有解。若  $f(a)=0$ ，解即为  $a$ ；若  $f(a) \cdot f(a+1) < 0$ ，则可以利用 A 中所述的二分法迅速找出解。

#### 4. 代码

```
#include<stdio.h>
#include<map>
#include<math.h>
#include<string.h>
int main()
{
    double a, b, c, d, i;
    while(scanf("%lf%lf%lf%lf", &a, &b, &c, &d) != EOF)
```



```

    {
        int times=0;
        for(i=-100; i<=99; i++)
        {
            int m=i+1;
            double aa=a*i*i*i+b*i*i+c*i+d, bb=a*m*m*m+b*m*m+c*m+d;
            if((aa<=0&&bb>=0)|| (aa>=0&&bb<=0))
            {
                double r=(i+1)*1.0, l=i*1.0;
                while(r-l>0.001)
                {
                    double mid=(l+r)/2;

                    if((a*l*l*l+b*l*l+c*l+d)*(a*mid*mid*mid+b*mid*mid+c*mid+d)<=0)
                        r=mid;
                    else
                        l=mid;
                }
                times++;
                if(times<3)
                    printf("%.2lf ", r);
                else
                    printf("%.2lf\n", r);
                if(times==3)
                    break;
                if(r-(i+1)==0)
                    i++;
            }
        }
    }
}

```

### 3.7.5 扩展变型

POJ 2299 Ultra-QuickSort

## 3.8 高精度计算

编写：包春志

校核：曹振海

### 3.8.1 基本原理

通常运算范围超出语言所提供的基本类型所能存储的数据范围时，我们一般采用高精度运算，这种算法是采用字符串存储待运算的数字，然后用字符串来进行加减乘除操作。

### 3.8.2 解题思路

通常高精度都可直接使用模板，JAVA 更提供了大数类可以直接利用。

### 3.8.3 模板代码

```

#include <iostream>
#include <string>
using namespace std;

```

```

string bigintadd(string a,string b)
{
    string t,res="0";
    int numa[1000],numb[1000],result[1000],i,j;
    int la,lb,tt,left=0,jinwei=0;
    la=a.length();
    lb=b.length();
    if(la<lb)                                //make la the longest;
    {
        tt=la;
        la=lb;
        lb=tt;
        t=a;
        a=b;
        b=t;
    }
    for(i=1; i<=lb; i++)
    {
        numa[i]=a[i-1]-'0';
        numb[i]=b[i-1]-'0';
    }
    for(i=lb+1; i<=la; i++)
    {
        numa[i]=a[i-1]-'0';
    }
    for(i=la; i>la-lb; i--)
    {
        numb[i]=numb[i-la+lb];
    }
    numa[0]=numb[0]=0;
    for(i=1; i<=la-lb; i++)
    {
        numb[i]=0;
    }
    for(i=la; i>=0; i--)
    {
        result[i]=numa[i]+numb[i]+jinwei;
        jinwei=result[i]/10;
        left=result[i]%10;
        result[i]=left;
    }

    if(result[0]!=0) cout<<result[0];
    for(i=1; i<=la-1; i++)
        res=res+"0";
    for(i=1; i<=la; i++)
    {
        res[i-1]=result[i]+'0';
    }
    return res;
}
    
```

## 大数减法

```

#include <iostream>
#include <string>
using namespace std;
string bigIntMin(string a,string b)
{
    string c="0",d="0",t;
    int aL,bL,LL,resultL;
    int i,j,k,g,count,jiewei;
    int A[10000],B[10000],result[10000];

    if(a.length()<b.length())
        {t=a;a=b;b=t;}

    aL=a.length();bL=b.length();

    resultL=aL;

    //按结果最大长度初始化结果数组
    for(i=0;i<resultL;i++)
        result[i]=0;

    for(i=0;i<aL;i++)                // 处
        A[i]=a[i]-'0';                // 理
    for(i=0;i<bL;i++)                // a
        B[i]=b[i]-'0';                // b
    for(i=resultL-1;i>=resultL-aL;i--) // ,
        A[i]=A[i-(resultL-aL)];        // 转
    for(i=0;i<resultL-aL;i++)          // 化
        A[i]=0;                        // 到
    for(i=resultL-1;i>=resultL-bL;i--) // 数
        B[i]=B[i-(resultL-bL)];        // 组
    for(i=0;i<resultL-bL;i++)          // 中
        B[i]=0;                        //

    for(i=resultL-1;i>=0;i--)
    {
        if(A[i]>=B[i])
            result[i]=A[i]-B[i];
        else {result[i]=A[i]-B[i]+10;A[i-1]=A[i-1]-1;}
    }
    count=0;
    for(i=0;i<resultL;i++)
    {
        if(result[i]!=0) break;
        else count++;
    }
    for(i=0;i<resultL-count;i++)
    {

```

```

        result[i]=result[i+count];
    }
    for(i=1;i<resultL-count;i++)
        c=c+d;

    for(i=0;i<resultL-count;i++)
        c[i]=result[i]+'0';
    return c;
}

大数乘法
#include <iostream>
#include <string>
using namespace std;
string bigIntMul(string a,string b)
{
    string c="0",d="0",t;
    int aL,bL,LL,resultL;
    int i,j,k,g,count,jinwei;
    int A[10000],B[10000],result[10000];
    if(a.length()<b.length())
    {
        t=a;
        a=b;
        b=t;
    }
    aL=a.length();
    bL=b.length();
    resultL=aL+bL+1;
    //按结果最大长度初始化结果数组
    for(i=0; i<resultL; i++)
        result[i]=0;
    for(i=0; i<aL; i++)                // 处
        A[i]=a[i]-'0';                  // 理
    for(i=0; i<bL; i++)                // a
        B[i]=b[i]-'0';                  // b
    for(i=resultL-1; i>=resultL-aL; i--) // ,
        A[i]=A[i-(resultL-aL)];          // 转
    for(i=0; i<resultL-aL; i++)          // 化
        A[i]=0;                          // 到
    for(i=resultL-1; i>=resultL-bL; i--) // 数
        B[i]=B[i-(resultL-bL)];          // 组
    for(i=0; i<resultL-bL; i++)          // 中
        B[i]=0;                          //
    //开始做乘法。两个循环，b 的循环包着 a 的循环，依次进行相乘保存在 result 中
    for(i=resultL-1; i>=resultL-bL; i--)
    {
        g=i;
        jinwei=0;
    }
}

```

```

        for(j=resultL-1; j>=resultL-aL; j--)
        {
            result[g]=B[i]*A[j]+result[g]+jinwei;
            jinwei=result[g]/10;
            result[g]=result[g]%10;
            g--;
        }
        if(jinwei>0) result[g]=result[g]+jinwei;
    }
    count=0;
    for(i=0; i<resultL; i++)
    {
        if(result[i]!=0) break;
        else count++;
    }
    for(i=0; i<resultL-count; i++)
    {
        result[i]=result[i+count];
    }
    for(i=1; i<resultL-count; i++)
        c=c+d;
    for(i=0; i<resultL-count; i++)
        c[i]=result[i]+'0';
    return c;
}

```

### 3.8.4 经典题目

#### 1. 题目出处/来源

Hrbust 1550 大数相加

#### 2. 题目描述

给出两个数 a 和 b 计算 a+b 的值，a，b 能不超过  $10^{100}$

#### 3. 分析

本题的要求就是计算 a+b，题目中给出这个数字会非常大，数字位数不超过 1000 位，int、long long 当然存不下了，所以这个题应该用大数加法运算。直接调用，大数运算的加法就可以了。

#### 4. 代码

```

#include <stdio.h>
#include <string.h>
#define max 100+10
char a[max],b[max];
int main(){
    int i,j,T,k;
    scanf("%d",&T);
    while(T--){
        int A[max]={0},B[max]={0};
        scanf("%s%s",a,b);
        int x=strlen(a),flagx=1;
        int y=strlen(b),flagy=1;
        for(i=0;i<x;i++){if(a[i]!='0')flagx=0;A[i]=a[x-1-i]-'0';}
    }
}

```

```

for(i=0;i<y;i++){if(b[i]!='0')flagy=0;B[i]=b[y-1-i]-'0';}
if(flagx&&flagy){printf("0\n");continue;}
int c=0;
for(i=0;i<max;i++){
    int s=(A[i]+B[i]+c);
    A[i]=s%10;
    c=s/10;
}
for( i=max-1;i>=0;i--) if(A[i])break;
for(j=i;j>=0;j--)      printf("%d",A[j]);
printf("\n");
}
return 0;
}

```

## 3.9 动态规划入门

编写：孟祥凤

校核：彭文文

### 3.9.1 基本原理

动态规划（DP）是运筹学（OR）的一个分支，是解决多阶段决策过程最优化的一种方法或是一种分析多阶段决策过程的数学方法，这种方法可根据人们所采取的措施，一步步地控制过程的发展，以实现预定的要求。

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题但是经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。在用分治法求解时，有些子问题被重复计算了许多次。如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

动态规划算法的基本要素

#### 1、最优子结构

矩阵连乘计算次序问题的最优解包含着其子问题的最优解。这种性质称为最优子结构性质。

在分析问题的最优子结构性时，所用的方法具有普遍性：首先假设由问题的最优解导出的子问题的解不是最优的，然后再设法说明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。

利用问题的最优子结构性质，以自底向上的方式递归地从子问题的最优解逐步构造出整个问题的最优解。最优子结构是问题能用动态规划算法求解的前提。

#### 2、重叠子问题

递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。这种性质称为子问题的重叠性质。

动态规划算法，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。通常不同的子问题个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。

### 3.9.2 解题思路

找出最优解的性质，并刻画其结构特征。

递归地定义最优值。

以自底向上的方式计算出最优值。

根据计算最优值时得到的信息，构造最优解。

### 3.9.3 模板代码

无固定模板代码。

### 3.9.4 经典题目

#### 3.9.4.1 题目 1

1. 题目出处/来源

HDU 2408 数塔

2. 题目描述

```

      7
     3 8
    8 1 0
   2 7 4 4
  4 5 2 6 5

```

上图给出了一个数字三角形。从三角形的顶部到底部有很多条不同的路径。对于每条路径，把路径上面的数加起来可以得到一个和，和最大的路径称为最佳路径。求出最佳路径上的数字之和。路径上的每一步只能从一个数走到下一层上和它最近的左边的数或者右边的数。以上样例答案为 30。

3. 分析

这道题目可以用递归的方法解决。基本思路是：

以  $D(r, j)$  表示第  $r$  行第  $j$  个数字 ( $r, j$  都从 1 开始算)，以  $\text{MaxSum}(r, j)$  代表从第  $r$  行的第  $j$  个数字到底部的最佳路径的数字之和，则本题是要求  $\text{MaxSum}(1, 1)$ 。从某个  $D(r, j)$  出发，显然下一步只能走  $D(r+1, j)$  或者  $D(r+1, j+1)$ 。如果走  $D(r+1, j)$ ，那么得到的  $\text{MaxSum}(r, j)$  就是  $\text{MaxSum}(r+1, j) + D(r, j)$ ；如果走  $D(r+1, j+1)$ ，那么得到的  $\text{MaxSum}(r, j)$  就是  $\text{MaxSum}(r+1, j+1) + D(r, j)$ 。所以，选择往哪里走，就看  $\text{MaxSum}(r+1, j)$  和  $\text{MaxSum}(r+1, j+1)$  哪个更大了。

这种将一个问题分解为子问题递归求解，并且将中间结果保存以避免重复计算的办法，就叫做“动态规划”。动态规划通常用来求最优解，能用动态规划解决的求最优解问题，必须满足，最优解的每个局部解也都是最优的。以上题为例，最佳路径上面的每个数字到底部的那一段路径，都是从该数字出发到达底部最佳路径。

实际上，递归的思想在编程时未必要实现为递归函数。在上面的例子里，有递推公式：

$$\text{anMaxSum}[r][j] = \begin{cases} D[r][j] & r = N \\ \max(\text{anMaxSum}[r+1][j], \text{anMaxSum}[r+1][j+1]) + D[r][j] & \text{其他情况} \end{cases}$$

因此，不需要写递归函数，从  $\text{anMaxSum}[N-1]$  这一行元素开始向上逐行递推，就能求得最终  $\text{anMaxSum}[1][1]$  的值了。

4. 代码

```

#include <stdio.h>
#define MAX 100+10
int D[MAX][MAX];
int N;
int aMaxSum[MAX][MAX];
int main(){
    int i, j;
    scanf("%d", &N);

```

```
    for(i = 1; i <= N; i++)
        for(j = 1; j <= i; j++)
            scanf("%d", &D[i][j]);
    for(j = 1; j <= N; j++)
        aMaxSum[N][j] = D[N][j];
    for(i = N; i > 1; i--)
        for(j = 1; j < i; j++){
            if(aMaxSum[i][j] > aMaxSum[i][j+1])
                aMaxSum[i-1][j] = aMaxSum[i][j] + D[i-1][j];
            else
                aMaxSum[i-1][j] = aMaxSum[i][j+1] + D[i-1][j];
        }
    printf("%d\n", aMaxSum[1][1]);
}
```

### 3.9.5 扩展变型

POJ 3903 Stock Exchange ( $n\log(n)$ 算法最长上升子序列)



## 附记