巍巍交大　百年书香
www.jiaodapress.com.cn
bookinfo@sjtu.edu.cn

丛书策划　张荣昌
责任编辑　王　清　孟海江
封面设计　唐逸设计

新时代计算机人才培养系列教材

大数据
- 大数据基础
- 大数据采集与预处理技术
- 数据结构与算法（第2版）
- 大数据集群搭建维护与数据存储
- 大数据采集与数据处理
- Hadoop应用与开发
- 数据可视化技术与应用
- 大数据分析技术与应用
- 数据挖掘技术与应用

云计算
- 云计算基础
- 私有云基础架构与运维
- 公有云服务架构与运维
- 云平台配置与管理
- 云安全技术应用
- 云网络技术应用
- 云计算运维开发
- 云计算应用开发

人工智能
- 人工智能应用基础
- 人工智能数学基础
- 人工智能数据服务
- 计算机视觉应用开发
- 深度学习应用开发
- 机器学习应用开发
- 自然语言处理及应用开发
- 智能语音处理及应用开发
- 人工智能系统部署与运维
- 人工智能综合项目开发

区块链
- 区块链应用基础
- 区块链核心技术
- 区块链部署与运维
- 区块链应用设计与开发
- 区块链项目综合实践
- 智能合约开发

物联网
- 物联网基础
- 物联网工程导论
- 物联网嵌入式技术
- 物联网设备装调与维护
- 物联网系统部署与运维
- 物联网工程设计与管理
- 物联网终端智能应用开发基础案例教程（基于HAL/LL库）

信息安全
- 信息安全技术
- 信息安全基础
- 信息安全标准与法规
- 信息安全工程与管理
- 信息安全风险评估
- 密码产品部署与应用
- 电子商务安全技术
- 无线网络安全技术
- 大数据安全技术
- 电子数据取证技术
- 终端数据安全及防泄密
- 数据存储与容灾
- Web应用安全与防护
- 渗透测试

专业基础
- 数据库应用技术（基于达梦数据库）
- MySQL数据库应用技术项目教程
- 计算机组装与维护
- 计算机网络基础
- Linux系统管理与服务器配置
- Linux服务器安全高级运维
- Windows Server操作系统基础
- 现代通信原理
- 无线传输技术
- 自动识别应用技术
- 智能传感与检测技术
- 路由交换技术项目实战教程
- 虚拟化技术基础

本书提供教学资源包
网址：https://www.sjhtbook.com

扫描二维码
关注上海交通大学出版社
官方微信

ISBN 978-7-313-33365-0
9 787313 333650 >
定价：56.00元

新时代计算机人才培养系列教材
数据结构与算法（第2版）
主编◎黄龙军
上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

浙江省线上一流课程“数据结构”配套教材
新时代计算机人才培养系列教材

数据结构与算法（第2版）

主编◎黄龙军

上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

浙江省线上一流课程“数据结构”配套教材
新时代计算机人才培养系列教材

数据结构与算法
（第2版）

主　编◎黄龙军
副主编◎胡珂立　李　平

扫一扫
学习资源库

上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

内容提要

本书介绍数据结构和算法相关知识，引入部分 C++ 标准模板库和程序设计竞赛知识，培养学生的计算思维、分析与解决具体问题的能力及创新能力。本书包括绪论、线性表、栈与队列、其他线性结构、树、图、查找和排序 8 章内容，重点介绍的内容主要包括线性表、栈、队列、二叉树和图等数据结构的概念、逻辑结构、存储结构及相应算法，二分查找、二叉排序树和哈希查找等查找算法，插入排序、快速排序、堆排序和二路归并排序等排序算法。本书力求把数据结构的抽象理论知识和算法实现讲得通俗易懂，以 C++ 语言描述算法，注重问题求解，可作为高等院校计算机类、电子信息类及应用统计学等相关专业程序设计类课程的教材，也可作为程序设计类竞赛和信息学竞赛的参赛者、数据结构与算法课程的教师或自学者的参考用书。

图书在版编目（CIP）数据

数据结构与算法 / 黄龙军主编. -- 2版. -- 上海 ：上海交通大学出版社，2025. 10. -- ISBN 978-7-313-33365-0

Ⅰ. TP311.12

中国国家版本馆CIP数据核字第2025X5M070号

数据结构与算法（第 2 版）
SHUJU JIEGOU YU SUANFA（DI 2 BAN）

主　　编：黄龙军
出版发行：上海交通大学出版社
邮政编码：200030
印　　制：北京荣玉印刷有限公司
开　　本：889 mm × 1194 mm　1/16
字　　数：489 千字
版　　次：2022 年 7 月第 1 版
　　2025 年 10 月第 2 版
书　　号：ISBN 978-7-313-33365-0
定　　价：56.00 元

地　　址：上海市番禺路 951 号
电　　话：021-6407 1208

经　　销：全国新华书店
印　　张：17

印　　次：2025 年 10 月第 3 次印刷

电子书号：ISBN 978-7-89564-430-4

目录

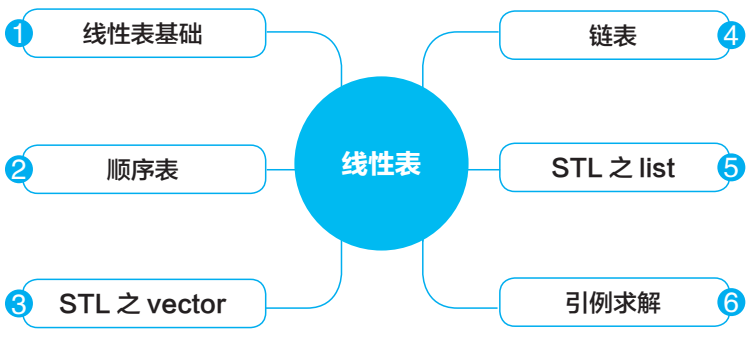
CONTENTS

第 2 章 线性表

学习目标

1. 记忆和理解线性表、顺序表和链表的基础知识。
2. 掌握顺序表、链表的实现。
3. 学会运用顺序表、链表求解具体问题。
4. 学会运用 STL 之 vector、list 求解具体问题。
5. 树立攻坚克难、勇于挑战的信念。
6. 获取脚踏实地、精诚合作的精神动力。

知识导图



导言

思政链接：佩利

线性表是最简单、最基本的一种线性数据结构。线性表可以采用顺序存储和链式存储，这两种存储方式同样适用于其他线性结构和非线性结构。因此，对应顺序存储的顺序表和对应链式存储的链表是学习其他数据结构及相关算法的重要基础，需熟练掌握。可以说，链表没学好是很多人学不好数据结构课程的主要原因。因此，熟练掌握链表至关重要。本章介绍线性表基础知识、顺序表、链表等方面的内容，重点关注顺序表和链表的实现、顺序表及链表的运用。另外，本章还介绍了C++标准模板库（standard template library，STL）中的向量vector，双向链表list。最后，介绍运用顺序表和链表求解具体问题。

【引例】序列合并

已知序列A、B各有m、n个元素，且元素按值非递减序排列，现要求把A和B合并为一个新的序列C，且C中的元素仍然按值非递减序排列。

例如，若A=(3，5，8，11)，B=(2，6，8，9，11，15，20)，则C=(2，3，5，6，8，8，9，11，11，15，20)。

输入：

首先输入一个正整数T，表示测试数据的组数，共有T组测试数据。每组测试数据输入两行，其中第一行先输入A的元素个数m（$1 \leqslant m \leqslant 100$），再输入$m$个$A$中的元素（整数）。第二行先输入$B$的元素个数$n$（$1 \leqslant n \leqslant 100$），再输入$n$个$B$中的元素（整数）。

输出：

对于每组测试数据，输出序列C中的全部元素，每两个元素之间留一个空格。

输入样例：

```
1
4 3 5 8 11
7 2 6 8 9 11 15 20
```

输出样例：

```
2 3 5 6 8 8 9 11 11 15 20
```

2.1 线性表基础

线性表$(a_1, a_2, \cdots, a_i, \cdots, a_n)$是一种最简单的线性结构，是$n$个特性相同的数据元素构成的有限序列。其中，$n$称为线性表的表长；$n$=0的线性表称为空表；$i$称为数据元素$a_i$的位序。

线性表$(a_1, a_2, \cdots, a_i, \cdots, a_n)$具有以下基本特征：

（1）存在唯一的一个“首元素”（第一个元素），即a_1。

（2）存在唯一的一个“尾元素”（最后一个元素），即a_n。

（3）除首元素之外，其他元素均仅有一个前驱，如a_2的前驱为a_1，a_n的前驱为a_{n-1}。

（4）除尾元素之外，其他元素均仅有一个后继，如a_1的后继为a_2，a_{n-1}的后继为a_n。

例如，6个英文字母可构成线性表(A, B, C, D, E, F)，5个整数可构成线性表(1, 2, 3, 4, 5)。

线性表具有创建、遍历、插入、删除、修改和查询等基本操作。线性表既可采用顺序存储结构，又可采用链式存储结构，相应的线性表分别称为顺序表和链表。

2.2 顺序表

2.2.1 顺序表基础

顺序表是采用顺序存储结构的线性表，是用一组地址连续的存储单元依次存放线性表中的数据元素。例如，线性表 $(a_1, a_2, \cdots, a_i, \cdots, a_n)$ 的顺序存储示意图如图 2–1 所示。

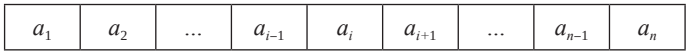


图 2–1　线性表的顺序存储示意图

在顺序表中，以“存储位置相邻”表示有序对 $<a_{i-1}, a_i>$，则 a_i 的存储地址 LOC(a_i) 按式 (2–1) 计算。

$$\mathrm{LOC}(a_i) = \mathrm{LOC}(a_{i-1}) + C \tag{2–1}$$

其中，C 表示一个数据元素所占的存储量。首元素 a_1 的地址 LOC(a_1) 称为线性表的起始地址或基地址。实际上，线性表中的所有其他数据元素的存储位置均取决于首元素 a_1 的存储位置，具体计算如式 (2–2) 所示。

$$\mathrm{LOC}(a_i) = \mathrm{LOC}(a_1) + (i-1) \times C \tag{2–2}$$

2.2.2 顺序表的定义与实现

这里把顺序表类型定义为一个结构体，该结构体中包含数据成员（属性）和成员函数（方法），具体如下：

```
const int N=80;                          // 顺序表的最大容量为 N
typedef int ElemType;                    // 为 int 类型取别名 ElemType
struct SqList {
    ElemType *elem;                      // 存储空间基址
    int length;                          // 当前长度
    void Init( );                        // 初始化
    void Clear();                        // 清除线性表
    int Locate(ElemType e);              // 查找
    void Insert(int i, ElemType e);      // 插入元素
    void Delete(int i, ElemType &e);     // 删除元素
    bool Empty();                        // 判断空表
    void Traverse();                     // 遍历
};
```

结构体类型 SqList 中的数据成员 *elem* 是一个 ElemType 类型的指针，用于存放动态数组的首地址；数据成员 *length* 表示当前顺序表中的元素个数（长度）。结构体类型 SqList 中的一个成员函数对应顺序表的一个基本操作。后文中也经常使用类似的表达，即一个基本操作用一个成员函数描述。可以通过把数据成员和成员函数构成一个整体来实现封装性，对于调用基本操作完成相应功能的用户而言，只需知道基本操作名和参数即可，而不必关心基本操作的具体实现。

用 typedef 把已有类型 int 取别名为 ElemType，从而提升代码的可维护性。例如，当顺序表的数据元素类型由 int 改为 char 类型时，仅需把该语句中的 int 改为 char 即可，而基本操作无需修改。

下面讨论顺序表基本操作的具体实现。

1. 初始化

顺序表的初始化操作申请一个动态数组并用数据成员 *elem* 指向，并置数据成员 *length* 为 0，具体代码如下：

```
// 初始化顺序表
void SqList::Init ( ) {                          // 构造空的顺序表
    elem=new ElemType [N];                       // 创建动态数组，首地址存放在 elem 中
    length=0;
}
```

Init 算法用 new 运算符创建长度为 *N* 的动态数组，把该数组的首地址存放在指针变量 *elem*（可视为数组名）中，并置表长为 0（表示空表）。显然，此算法的时间复杂度为 O(1)。

:: 是类属运算符，表明其后的函数（此处为 Init）属于其前的结构体类型（此处为 SqList）。

2. 查找

在顺序表中查找元素 *e* 时，从头开始依次把 *e* 与顺序表中元素进行比较，若相等则返回其位序，否则继续比较下一个元素；若最终都没有与 *e* 相等的元素，则返回 0。顺序表的查找算法 Locate 具体实现如下。

```
// 在顺序表中查找第一个满足判定条件的数据元素，若存在，则返回它的位序，否则返回 0
int SqList:: Locate (ElemType e) {                    // 查找
    int i=1;                                          // i 的初值为第一个元素的位序（1）
    while (i <=length && elem[i-1]!=e) {
        i++;
    }
    if (i <=length)  return i;
    else  return 0;
}
```

设等于 *e* 的元素在长度为 *n* 的顺序表的最后一个位置，则需要循环 *n* 次才能找到，因此 Locate 算法的最坏时间复杂度为 O(*n*)。考虑 *e* 在各个位置出现的概率相同，则相应循环次数分别为 1，2，3，…，*n*，平均循环次数等于 $\frac{1}{n}\times\frac{n(n+1)}{2}=\frac{n+1}{2}$，因此 Locate 算法的平均时间复杂度也为 O(*n*)。

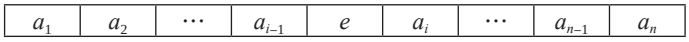
3. 插入

顺序表操作 Insert (*i*, *e*) 实现将元素 *e* 插入为顺序表的第 *i*（$1 \leqslant i \leqslant n+1$）个元素。

在第 *i* 个元素之前插入元素 *e*，将使线性表从 (a_1, …, a_{i-1}, a_i, …, a_n) 改变为 (a_1, …, a_{i-1}, *e*, a_i, …, a_n)，即从一个有序对 <a_{i-1}, a_i> 变为两个有序对 <a_{i-1}, *e*>、<*e*, a_i>，且表长增加了 1。顺序表的插入操作示意图如图 2–2 所示。

a_1	a_2	…	a_{i-1}	a_i	…	a_{n-1}	a_n

（a）



（b）

图 2-2　顺序表的插入操作示意图

（a）插入前；（b）插入后

为了在第 i 个元素之前插入元素 e，需把第 i 个位置空出来，即把 a_n，a_{n-1}，⋯，a_i 依次后移一个位置，再把 e 存放到第 i 个位置，最后表长 *length* 增加了 1。顺序表的插入算法 Insert 具体实现如下：

```
// 将新元素 e 插入为顺序表的第 i 个元素（下标为 i-1）
void SqList:: Insert( int i, ElemType e) {
    if (i < 1 || i > length+1) return ;              // 若插入位置非法，则不做插入操作
    for (int j=length-1; j >=i-1;  j--)
        elem[j+1]=elem[j];                           // 插入位置及之后的元素后挪
    elem[i-1]=e;                                     // 插入元素 e
    length ++;                                       // 表长增 1
}
```

考虑移动元素的平均情况，假设插入为第 i 个元素的概率为 p_i，则在长度为 n 的线性表中插入一个元素所需移动元素次数的期望值（平均次数）如式 (2-3) 所示。

$$E_{is}=\sum_{i=1}^{n+1}p_i(n-i+1) \tag{2-3}$$

若假定在线性表中任何一个位置 i（$1 \leqslant i \leqslant n+1$）上进行插入的概率都是相等的，则移动元素的平均次数如式 (2-4) 所示。

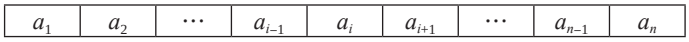
$$E_{is}=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{n}{2} \tag{2-4}$$

因此，Insert 算法的平均时间复杂度为 O(n)。Insert 算法的最坏情况是插入元素到第一个元素之前，时间复杂度也为 O(n)。

4. 删除

顺序表操作 Delete (i, &e) 实现删除顺序表中的第 i 个元素，并通过引用参数 e 返回该元素。

设删除元素 a_i，则线性表从 (a_1, ⋯, a_{i-1}, a_i, a_{i+1}, ⋯, a_n) 改变为 (a_1, ⋯, a_{i-1}, a_{i+1}, ⋯, a_n)，即从两个有序对 <a_{i-1}, a_i>, <a_i, a_{i+1}> 变为一个有序对 <a_{i-1}, a_{i+1}>，且表长 *length* 减 1。顺序表删除前后的示意图如 2-3 所示。



（a）

（b）

图 2-3　顺序表删除前后的示意图

（a）删除前；（b）删除后

可见，若要删除元素 a_i，则可通过把其后的元素 a_{i+1}, a_{i+2}, ⋯, a_{n-1}, a_n 逐个前移覆盖原有元素。删除算法 Delete 具体实现如下：

```
// 顺序表中删除第 i 个元素，通过引用参数 e 返回该元素
void SqList:: Delete (int i, ElemType &e) {
    if (i < 1 || i > length)  return;          // 待删位置非法，则返回
    e=elem[i-1];                               // 被删除元素的值赋给 e
    // 被删除元素之后的元素左移
    for (int j=i; j <=length-1; j++)  elem[j-1]=elem[j];
    length--;                                  // 表长减 1
}
```

待删元素通过引用参数 e 返回，若删除之后无须再用到该元素，则可省略引用参数 e。

考虑移动元素的平均情况，假设删除第 i 个元素的概率为 q_i，则在长度为 n 的线性表中删除一个元素所需移动元素次数的期望值（平均次数）如式 (2–5) 所示。

$$E_{del}=\sum_{i=1}^{n}q_i(n-i) \tag{2–5}$$

若假定在线性表中任何一个位置上进行删除的概率都是相等的，则移动元素的平均次数如式 (2–6) 所示。

$$E_{del}=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2} \tag{2–6}$$

因此，Delete 算法的平均时间复杂度为 O(n)。Delete 算法的最坏情况是删除第一个元素，时间复杂度也为 O(n)。

5. 其他操作的算法实现

顺序表的其他操作比较简单，下面直接给出它们的算法实现：

```
// 清空顺序表，时间复杂度为 O(1)
void SqList::Clear() {
    delete [] elem;
    length=0;
}
// 判断是否空顺序表，时间复杂度为 O(1)
bool SqList::Empty() {
    return length==0;
}
// 遍历顺序表，时间复杂度为 O(n)
void SqList::Traverse() {
    for (int i=0; i < length; i++) {
        if (i>0) cout<<" ";
        cout<<elem[i];
    }
    cout<<endl;
}
```

6. 顺序表基本算法的调用

顺序表基本操作的算法实现之后，就可以调用这些算法完成相应功能。下面给出简单的调用示例：

```
#include <iostream>
using namespace std;
int main() {
    SqList L;                          // 定义顺序表
    L.Init();                          // 初始化顺序表
    int n;
    cin>>n;
    for(int i=0;i<n;i++) {             // 尾插创建顺序表，时间复杂度为平方阶
        ElemType t;
        cin>>t;
        L.Insert(i+1, t);              // 在顺序表中插入元素
    }
    L.Traverse();                      // 遍历顺序表
    cout<<L.Empty()<<endl;             // 顺序表判空
    ElemType m;
    cin>>m;
    cout<<L.Locate(m)<<endl;           // 查找顺序表
    L.Delete(3, m);                    // 删除顺序表中的第 3 个元素
    L.Traverse();                      // 遍历顺序表
    L.Clear();                         // 清空顺序表
    return 0;
}
```

2.2.3　顺序表的应用

例 2.2.1　顺序表的就地逆置（HLOJ 9503）

试写一算法，实现顺序表的就地逆置，即利用原表的存储空间将线性表 $(a_1, a_2, \cdots, a_n)$ 逆置为 $(a_n, a_{n-1}, \cdots, a_1)$。

输入：

测试数据有多组，处理到文件尾。每组测试数据在一行上输入数据个数 n（$n \leqslant 15$）及 n 个整数。

输出：

对于每组测试，将顺序表中的元素逆置存储于原表后输出。每两个整数间留一个空格。

输入样例：

```
12 32 22 88 25 75 29 5 41 89 83 24 1
7 36 83 39 86 62 89 24
```

输出样例：

```
1 24 83 89 41 5 29 75 25 88 22 32
24 89 62 86 39 83 36
```

解析：

实现顺序表的就地逆置的一种算法思想是以中间位置为界，逐个交换左右对称位置上的元素。顺序表的初始化、创建、遍历的实现与前述类似。具体代码如下：

```
#include<iostream>
using namespace std;
typedef int ElemType;
struct SqList {
    ElemType *elem;
    int length;
    void Init(int n);
    void Create(int n);
    void Traverse();
};
// 逆置顺序表，因需把逆置之后的结果返回，故使用引用参数
void reverseSqList(SqList &l) {
    for(int i=0;i<l.length/2;i++) {
        swap(l.elem[i], l.elem[l.length-1-i]);
    }
}
int main() {
    int n;
    while(cin>>n) {
        SqList sl;
        sl.Create(n);
        reverseSqList(sl);
        sl.Traverse();
    }
    return 0;
}
// 遍历顺序表
void SqList::Traverse() {
    for(ElemType *p=elem; p<elem+length; p++) {
        if(p!=elem) cout<<" ";
        cout<<*p;
    }
    cout<<endl;
}
// 初始化顺序表
void SqList::Init(int n) {
    elem=new ElemType[n];
    length=0;
}
// 建立顺序表
void SqList::Create(int n) {
    Init(n);
    for(ElemType *p=elem; p<elem+n; p++) {
        cin>>*p;
        length++;
    }
}
```

在 reverseSqList 函数中，考虑到需把逆置之后的结果返回，故使用引用参数。实际上，因结构体

SqList 中的 *elem* 成员是指针，该成员在进行参数传递时传的是地址，即将实参 *sl* 的 *elem* 的值（地址）传给形参 *l*，在函数调用期间 *l.elem* 和 *sl.elem* 值相同，即它们是同一个数组，因此对 *l.elem* 数组的改变就是对 *sl.elem* 数组的改变，因此 *l* 可不必作为引用参数，即 *l* 前的 & 可省略。

以上这种代码编写方式是为了便于读者体会如何以“数据结构方式”（先定义数据结构并实现基本操作，再调用已实现的基本操作求解问题）编写程序。而在比赛时，一般采用更简洁的代码编写方式，例如，直接用一维数组或一维向量（vector）表示顺序表。

2.3 STL 之 vector

1. 基础知识

STL 之 vector（向量）可用于表示顺序表，头文件是 vector。定义向量时，可同时指定其长度。例如：

```
int n;
cin>>n;
vetcor <int> a(n);
```

语句“ vetcor <int> a(n); ”是 vector 的实例化，即在 < > 中指定 *a* 的每个元素的类型都是 int，*a* 后面小括号中的 *n* 表明该向量的长度为 *n*。

vetcor 部分常用成员函数如表 2–1 所示，其中，参数 *val* 的类型同向量中的元素，可以是 int、char、double 等基本数据类型、string 类型及结构体类型等，*pos* 是迭代器（ iterator）类型（类似指针类型）参数，*n* 是整型参数。设一维向量名为 *a*，其中的元素为 1、3、5、7。

表 2–1　vector 部分常用成员函数

成员函数	含义与示例
begin()	指向首元素的迭代器。例如，a.begin() 指向元素 1
end()	指向尾元素后一位置的迭代器。例如，a.end() 指向元素 7 之后的位置
clear()	清空向量。例如，a.clear() 清空向量 a
empty()	向量判空，若为空，则返回 true，否则返回 false。例如，a.clear() 之后 a.empty() 为 true
erase(pos)	删除迭代器 pos 所指元素并返回下一元素的迭代器。例如，a.earse(a.begin()) 将删除 a 的首元素 1 并返回指向元素 3 的迭代器
insert(pos, val)	在迭代器 pos 所指位置插入一个值为 val 的元素并返回其迭代器 例如，a.insert(a.begin(), 9) 将把 9 插入 a 的头部并返回指向该元素的迭代器
pop_back()	删除向量的尾元素。例如，a.pop_back() 将删除元素 7
push_back(val)	将 val 插入向量的最后位置。例如，a.push_back(9) 将把元素 9 插入 a 的最后位置
size()	vector 的大小（一维 vector 的元素个数或二维 vector 的行数、列数）。例如，a.size() 为 4
resize(n, val)	重置 vector 大小为 n，若 n 小于向量原来的大小（设为 m），则保留前面 n 个元素及其值，否则前 m 个元素保留原值，后 n-m 个元素置值为 val（该参数默认的参数值为 0 ）。例如，a.resize(10) 将 a 的长度重定义为 10，且前 4 个元素保留原值为 1、3、5、7，后 6 个元素为 0

2. 一维向量

例 2.3.1　一维 vector 使用方法

具体代码如下：

```
#include <iostream>
#include <vector>
using namespace std;
void prtVector(vector<int> a);
int main() {
 int n,i;
    cin>>n;
    vector <int> v(n);                          // 长度为 n 的一维向量，实现动态数组
    for(i=0; i<v.size(); i++) v[i]=i+1;
    v.push_back(123);                           // 在尾部插入
    v.insert(v.begin(), 456);                   // 在头部插入
    v.erase(v.begin()+1);                       // 删除第二个元素
    prtVector(v);                               // 输出向量元素
    v.clear();                                  // 清空向量
    v.resize(n*2);                              // 重新定义长度
    cout<<v.size()<<endl;
    return 0;
}
// 输出向量中所有元素，每两个数据之间间隔一个空格
void prtVector(vector<int> a) {
    for(int i=0; i<a.size(); i++) {
        if (i>0) cout<<" ";
        cout<<a[i];
    }
    cout<<endl;
}
```

读者可自行编程测试 vector 常用成员函数的使用。

3. 二维向量

例 2.3.2　二维 vector 简单示例

具体代码如下：

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int i;
    int r,c;
    cin>>r>>c;
    // 下面的语句直接定义 r 行 c 列的二维动态数组
    vector < vector<int> > tv(r, vector<int> (c));
    for(i=0; i<r; i++) {
        for(int j=0; j<c; j++)
            cin>>tv[i][j];
    }
```

```
    for(i=0; i<r; i++) {
        for(int j=0; j<c; j++) {
            if (j>0) cout<<" ";
            cout<<tv[i][j];
        }
        cout<<endl;
    }
    tv.clear();
    return 0;
}
```

语句“vector < vector<int> > tv (r, vector<int> (c));”定义一个 *r* 行 *c* 列的二维向量，注意“> >”中两个 > 之间有空格，以免被编译器理解为运算符“>>”而出错（PTA 平台和 Dev–C++ 等编译器允许两个 > 之间无空格）；定义二维向量还可以采用如下写法：

```
vector <vector <int> > tv;                  // 定义二维向量
tv.resize(r);                               // 重置第一维长度（行数）为 r
for(i=0; i<r; i++) tv[i].resize(c);         // 重置第二维长度（列数）为 c
```

实际上，二维 vector 的每行的列数可以不一样，这样使用起来更加灵活，例如：

```
for(i=0; i<r; i++) tv[i].resize(i+1);       // 定义了一个下三角矩阵
```

因此，可用第二维长度不一的二维向量模拟实现图的邻接表存储结构。

可以用 *tv*.size() 得到 *tv* 的行数，用 *tv*[*i*].size() 得到第 *i* 行的列数，例如，以下代码输出各行的列数：

```
for(i=0; i<r; i++) cout<<tv[i].size()<<endl;
```

2.4 链表

2.4.1 链表概述

链表用一组地址任意的存储单元存放线性表中的数据元素。链表可认为是以“结点的序列”表示的线性表。链表中的一个结点包含数据域和指针域两部分，其中，数据域存放数据元素，而指针域存放其他结点的地址。

设结点包含一个整型数据域 *data* 和一个指针域 *next*，则结点的结构体类型声明如下：

```
struct LNode {                              // 结点结构体类型
    int data;                               // 数据域
    LNode *next;                            // 指针域，用于存放下一个结点的地址
};
```

其中，数据成员 *next* 是一个 LNode* 类型的指针，即用来存放下一个结点（LNode 类型变量）的地址。

这里主要介绍带头结点（该结点的数据域不存放有效数据）的单链表。例如，共有 4 个数据结点（数据域存放有效数据）的带头结点的单链表示意图如图 2–4 所示。

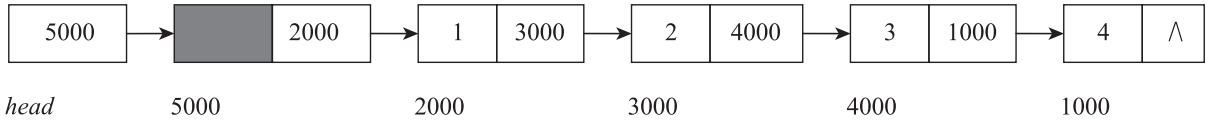


图 2-4　共有 4 个数据结点的带头结点的单链表示意图

在图 2-4 中，数据域为灰色（值为某个随机数）的结点是头结点，地址设为 5000，存放在指针变量 *head* 中，则 *head* 指向头结点，称为头指针；其余结点的地址分别设为 2000、3000、4000、1000。因地址指向该地址所在的存储单元，故后继（后一个结点）的地址存放在前驱（前一个结点）的指针域时，前驱的指针域就指向后继，从而构成一个链表。在带头结点的单链表中，第一个数据结点的前趋是头结点，最后一个数据结点没有后继，其指针域的值为空指针 NULL（链表结束标志），图中以“∧”表示。

思政链接：
巴克斯

在单链表中，前一结点的指针域指向后一结点，只能通过前一结点才能找到后一结点。因此，单链表的访问规则是“从头开始、顺序访问”，即通过指向头结点的头指针，逐个结点依次访问。

在顺序表中插入、删除元素时，需要大量移动数据元素，时间效率较低。而在单链表中，若在某个结点之后插入或删除结点，只需简单修改结点指针域的指向而不必大量移动元素，因此插入和删除操作频繁时宜用链表结构。

画示意图是学习链表的一个重要方法，读者可通过画图理解链表的基本操作。

2.4.2　单链表基本操作及其实现

结点类型 LNode 如前述，把链表的基本操作和头指针构成为一个整体，定义带头结点的链表的结构体类型 LinkList 的具体代码如下：

```
typedef int ElemType;                       // 定义 ElemType 为 int 类型的别名
struct LinkList {                           // 声明链表结构体类型
    LNode *head;                            // 头指针，指向头结点
    void Init();                            // 初始化
    void Clear();                           // 清空链表
    void Create(int n);                     // 建立含 n 个结点的单链表
    int Locate(ElemType e);                 // 查找
    void Insert(int i, ElemType e);         // 插入元素
    void Delete(int i, ElemType &e);        // 删除元素
    void Traverse();                        // 遍历
    bool Empty();                           // 判断空链表
};
```

单链表基本操作的具体实现如下。

1. 初始化（创建空链表）

具体代码如下：

```
// 创建结点并用头指针指向，然后置其指针域为空指针
void LinkList::Init() {
    head=new LNode;
    head->next=NULL;
}
```

Init 算法的时间复杂度为 O(1)。

2. 判断是否空链表

具体代码如下：

```
// 若头指针所指结点的指针域为空指针则返回 true，否则返回 false
bool LinkList::Empty() {
    return head->next==NULL;
}
```

Empty 算法的时间复杂度为 O(1)。

3. 插入结点

插入算法 Insert(*i*, *e*) 实现的是将数据域值为 *e* 的结点插入为带头结点的单链表（头指针为 *head*）的第 *i*（1 ⩽ *i* ⩽ *n*+1）个结点。首先需要从头开始找到第 *i*–1 个结点（设由 *p* 指向，*i* 为 1 时 *p* 指向头结点），然后在其后插入新结点（设由 *q* 指向）。设 *i*=3、*e*=5，则第 3 个结点插入前后的示意图如图 2–5 所示。

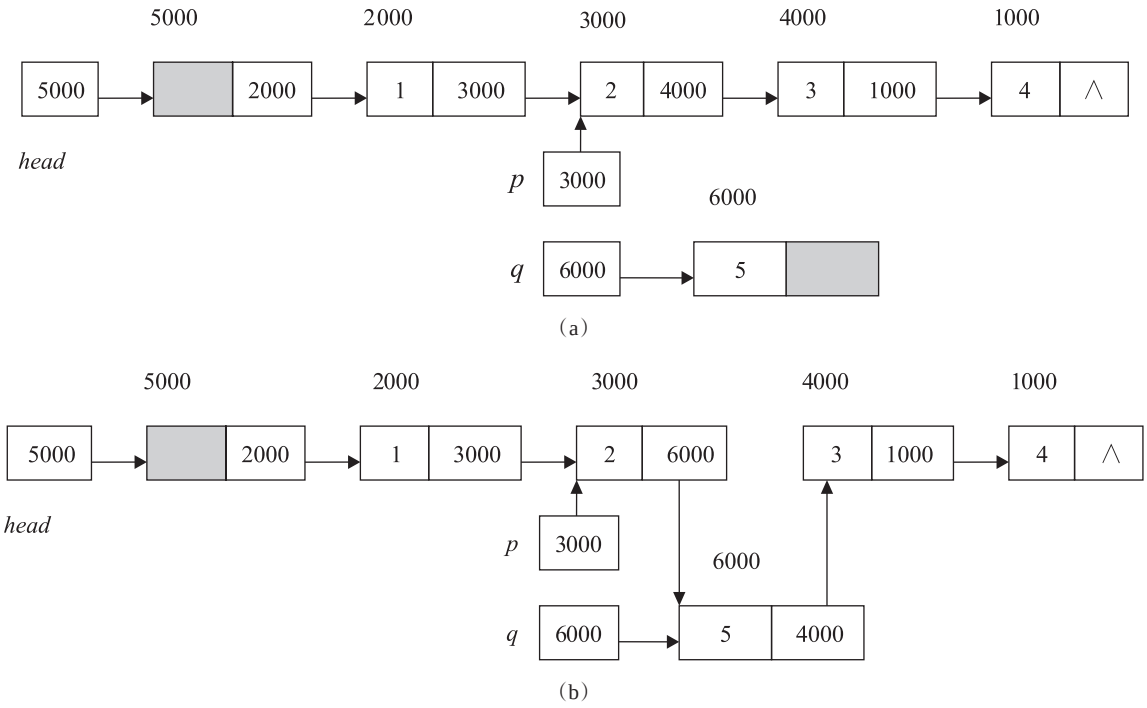


图 2–5　第 3 个结点插入前后的示意图

（a）插入前；（b）插入后

插入算法 Insert 的具体代码如下：

```
// 将数据域值为 e 的结点插入为带头结点的单链表的第 i（1<=i<=n+1）个结点
void LinkList::Insert(int i, ElemType e) {
    LNode* p=head;
    while (p!=NULL && i>1) {                    // 找第 i-1 个结点，用 p 指向该结点
        p=p->next;
        i--;
    }
```

```
        if (p==NULL || i <1)  return;              // 若 i 超出范围，则插入位置非法
        LNode* q=new LNode;                        // 生成新结点，用 q 指向该结点
        q->data=e;
        q->next=p->next;                           // 新结点链接到第 i 个结点之前
        p->next=q;                                 // 新结点链接到第 i-1 个结点之后
    }
```

Insert 算法包含了查找第 i–1 个结点的过程，因此时间复杂度为 O(n)。若不做查找操作，则仅需语句“q->next = p->next; p->next = q;”就可完成插入操作，此时的时间复杂度为 O(1)。

4. 删除结点

删除算法 Delete (i, &e) 实现的是在头指针为 *head* 的单链表中删除第 i（$1 \leqslant i \leqslant n$）个结点，并通过引用参数 e 返回其数据域值。首先需要从头开始找到该结点的前驱结点（设由 q 指向），则待删结点由 *p=q->next* 指向，然后只要把 p 所指结点的后继的地址 *p->next* 放到 q 所指结点的指针域 *q->next* 即可完成删除操作。设 i 为 3，则删除第 3 个结点前后的示意图如图 2–6（a）、图 2–6（b）所示，释放所删结点的空间之后的示意图如图 2–6（c）所示。

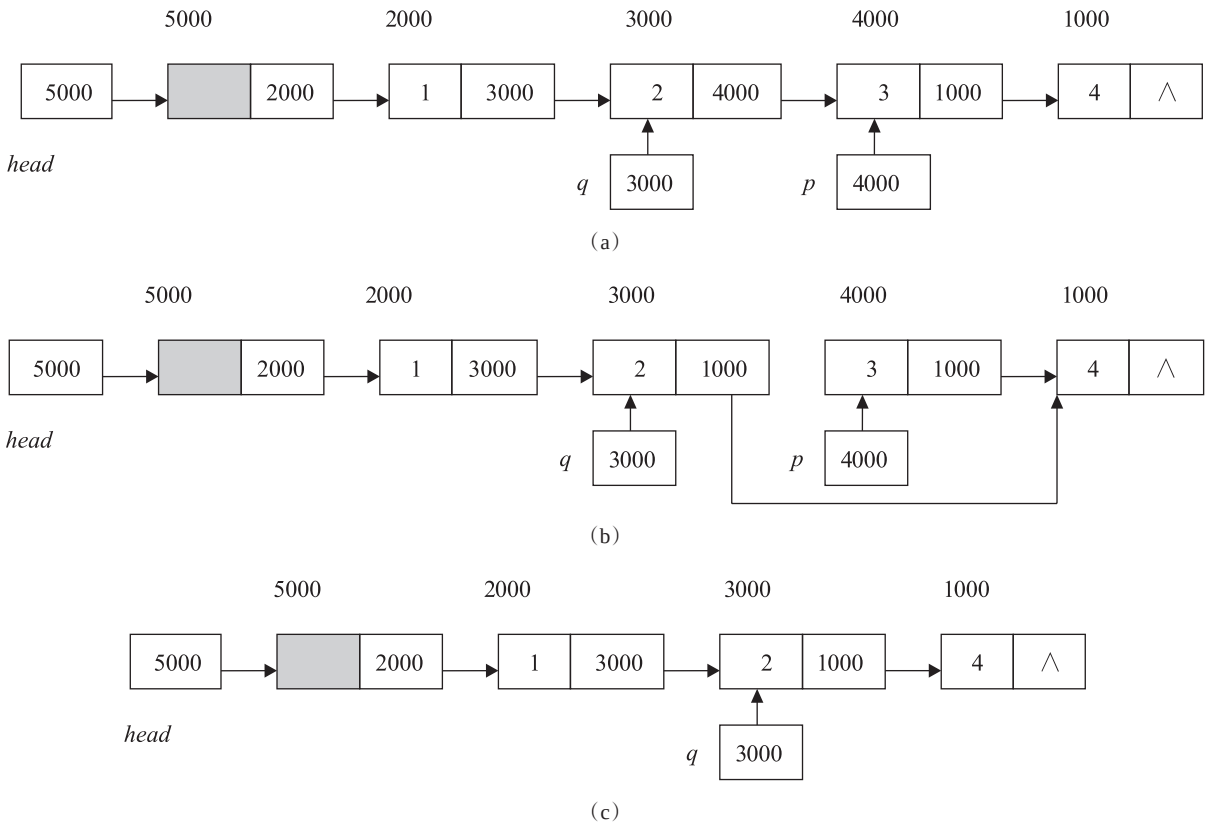


图 2–6　删除第 3 个结点前后的示意图

（a）删除前；（b）删除后；（c）释放所删结点的空间之后

删除算法 Delete 的具体代码如下：

```
// 删除带头结点的单链表中第 i (1<=i<=n) 个结点，并用引用参数 e 返回其数据域值
void LinkList::Delete (int i, ElemType &e) {
    LNode* q=head;
    int j=0;
```

```
    // 找到第 i-1 个结点，用 q 指向该结点，则 q->next 指向第 i 个结点
    while (q->next!=NULL && j < i-1) {
        q=q->next;
        j++;
    }
    if (!q->next || j > i-1) return;          // i 超出范围，删除位置不合理
    // 删除并释放结点
    LNode* p=q->next;
    q->next=p->next;
    e=p->data;
    delete p;
}
```

Delete 算法包含了查找第 $i-1$ 个结点的过程，因此时间复杂度为 O(n)。若不做查找操作，也不考虑释放已删结点所占空间，则仅需语句“q->next = q->next->next;”就可删除 q 所指结点的后继，此时的时间复杂度为 O(1)。

5. 清空链表

Clear 算法实现的是清空链表，可以通过逐个删除第一个数据结点的方法来完成，具体代码如下：

```
// 将单链表重新置为一个空表
void LinkList::Clear( ) {
    while (head->next!=NULL) {
        LNode* p=head->next;
        head->next=p->next;
        delete p;
    }
}
```

Clear 算法的时间复杂度为 O(n)。

6. 查找结点

算法 Locate(e) 实现的是在链表中查找数据域值为 e 的结点，若找到，返回该结点的位序，否则返回 0。只需从头开始、顺序查找，即逐个比较待查找的 e 是否等于当前结点数据域的值。

查找算法 Locate 的具体代码如下：

```
// 在单链表中查找数据域值为 e 的结点，若找到，返回指向该结点的位序，否则返回 0
int LinkList::Locate(ElemType e) {
    LNode *p=head->next;
    int i=1;
    while(p!=NULL) {
        if (e==p->data) break;
        i++;
        p=p->next;
    }
    if (!p) return 0;
    else return i;
}
```

Locate 算法的时间复杂度为 O(*n*)。

7. 创建链表

链表是一个动态的结构，它不需要预先分配空间，因此创建链表是一个“逐个插入”结点的过程。建立带头结点的单链表常用两种思想：

（1）尾插法：新结点插入尾结点之后，所得链表称为顺序链表。

（2）头插法：新结点插入头结点之后，第一个数据结点之前，所得链表称为逆序链表。

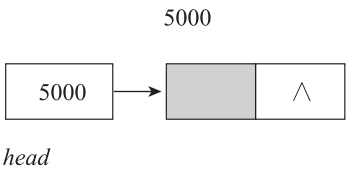


图 2-7　带头结点的空链表

算法 Create(*n*) 采用头插法创建带头结点的单链表。以建立如图 2-4 所示的带头结点的单链表为例，数据输入顺序为 4、3、2、1。创建逆序链表的过程有如下三个步骤。

第一，建立一个带头结点的空链表，如图 2-7 所示，可以直接调用初始化算法 Init 实现。

第二，申请新结点由指针 *q* 指向，输入数据，并链接到头结点之后、第一个数据结点（由 *head->next* 指向，第一次为 NULL）之前，示意图如图 2-8 所示（输入数据 4）。

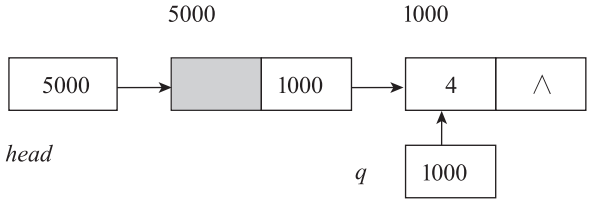
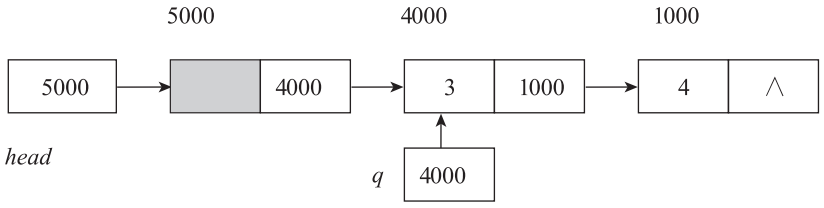
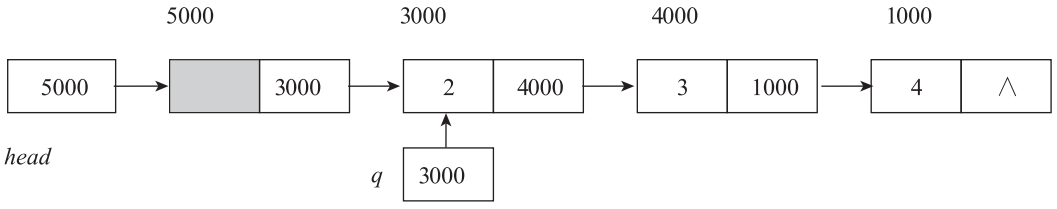


图 2-8　插入第 1 个数据构成的结点

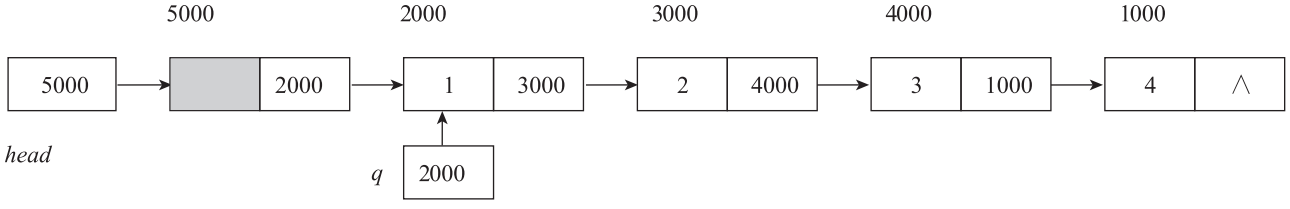
第三，重复第二步，直到所有数据结点都插入链表中，如图 2-9 所示。



(a)



(b)



(c)

图 2-9　分别插入第 2、3、4 个数据构成的结点

（a）插入第 2 个数据构成的结点；（b）插入第 3 个数据构成的结点；（c）插入第 4 个数据构成的结点

创建逆序链表的 Create 算法具体代码如下：

```
// 创建包含 n 个元素的逆序链表
void LinkList:: Create(int n) {
    Init();
    for (int i=n; i>0; i--) {
        LNode* q=new LNode;
        cin>>q->data;
        q->next=head->next;
        head->next=q;
    }
}
```

Create 算法的时间复杂度为 O(n)。

创建逆序链表也可以逐个输入数据暂存到变量 e 中，并调用 Insert(1, e) 逐个插入链表的第一个数据结点之前，此时的时间复杂度也为 O(n)。

顺序链表的创建过程和代码实现留给读者自行思考和实现。

8. 遍历链表

遍历链表的 Traverse 算法根据“从头开始，顺序访问”的单链表访问规则，用一个指针变量 p 一开始指向头结点之后的结点，即第一个数据结点，在链表还未结束时不断输出 p 所指结点的数据域的值并使 p 指向下一个结点。另外，采用计数器的方法控制每两个数据之间留一个空格。

链表遍历算法 Traverse 具体代码如下：

```
// 遍历单链表
void LinkList::Traverse() {
    LNode* p=head->next;
    int cnt=0;
    while (p!=NULL) {
        cnt++;
        if (cnt>1) cout<<" ";
        cout<<p->data;
        p=p->next;
    }
    cout<<endl;
}
```

Traverse 算法的时间复杂度为 O(n)。

9. 调用链表基本操作

实现单链表的基本操作之后，可以调用这些基本操作以实现相应功能。具体代码如下：

```
#include<iostream>
using namespace std;
int main() {
    int m, n, t;
    LinkList L;                                    // 定义链表
```

```
    L.Init();                                    // 初始化链表
    cin>>n;
    // 调用 Insert 算法创建顺序链表，时间复杂度为平方阶
    for(int i=0;i<n;i++) {
        cin>>t;
        L.Insert(i+1,t);                         // 插入结点
    }
    L.Traverse();                                // 遍历链表
    cout<<L.Locate(5)<<endl;                     // 查找结点
    L.Delete(5, m);                              // 删除结点
    L.Traverse();                                // 遍历链表
    L.Clear();                                   // 清空链表
    return 0;
}
```

2.4.3 链表的运用

例 2.4.1 单链表中重复元素的删除（HLOJ 9513）

按照数据输入的顺序建立一个单链表，并将单链表中重复的元素删除（值相同的元素只保留最早输入的那一个）。

输入：

测试数据有多组，处理到文件尾。对于每组测试，第一行输入元素个数 n；第二行输入 n 个整数。

输出：

对于每组测试，输出两行，第一行输出删除重复元素后的单链表元素个数；第二行输出删除重复元素后的单链表。

输入样例：

```
10
21 30 14 55 32 63 11 30 55 30
```

输出样例：

```
7
21 30 14 55 32 63 11
```

解析：

链表中删除重复元素的一种算法思想是遍历链表，对每个数据结点取数据域值与其后结点相比较，若相等，则删除其后结点。为便于删除操作，可设一个指针指向待删结点的前驱结点。具体代码如下：

```
#include<iostream>
using namespace std;
typedef int ElemType;
struct LNode {
    ElemType data;                    // 数据域
    LNode* next;                      // 指针域
```

```
};
struct LinkList {
    LNode *head;                          // 头指针（带头结点）
    void Create(int n);                   // 建立含 n 个结点的单链表
    void Traverse();                      // 遍历，并输出内容
};
void deleteSame(LinkList La, int &n) {
    LNode *p=La.head->next;               // p 指向第一个数据结点
    while(p!=NULL) {                      // 当链表未结束时重复处理
        LNode *q=p, *r=p->next;           // r 用于指向待删结点，q 指向其前驱结点
        while(r!=NULL) {
            if (r->data==p->data) {       // 若 r 所指元素等于 p 所指元素，则删除 r 所指结点
                LNode* s=r;
                q->next=r->next;
                r=r->next;
                delete s;
                n--;
            }
            else {                        // 若 r 所指元素不等于 p 所指元素，则 q、r 往下走
                q=r;
                r=r->next;
            }
        }
        p=p->next;                        // p 指向下一个结点
    }
}
int main() {
    int n;
    while(cin>>n) {
        LinkList lst;
        lst.Create(n);
        deleteSame(lst,n);
        cout<<n<<endl;
        lst.Traverse();
    }
    return 0;
}
// 建立带头结点的单链表
void LinkList::Create(int n) {
    head=new LNode;
    head->next=NULL;                      // 先建立一个带头结点的单链表
    LNode* q=head;                        // 尾指针
    for (int i=0; i<n; i++) {
        LNode* p=new LNode;
        cin>>p->data;                     // 输入元素值
        p->next=NULL;
        q->next=p;                        // 链接到尾指针 q 之后
```

```
            q=p;
        }
    }
    // 遍历链表
    void LinkList::Traverse() {
        LNode* p=head->next;
        int cnt=0;
        while (p!=NULL) {
            cnt++;
            if (cnt>1) cout<<" ";
            cout<<p->data;
            p=p->next;
        }
        cout<<endl;
    }
```

注意，在 DeleteSame 函数中，在删除 *r* 所指结点后，*r* 接着指向下一个结点，但其前驱指针 *q* 不能同时往下指，请读者思考缘由，建议画图进一步理解。

本题也可以采用 list 实现，删除元素可用 list 的 erase 成员函数，读者可自行尝试实现。

2.4.4 其他形式的链表简介

本节介绍的其他形式的链表包括循环单链表和双向链表。

1. 循环单链表

循环单链表是最后一个结点的指针域的指针又指回头结点的链表。如图 2-10 所示的是包含一个数据域和一个指针域的循环单链表。

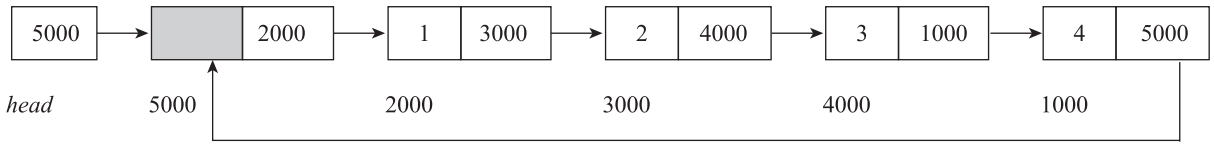


图 2-10　包含一个数据域和一个指针域的循环单链表

循环单链表和单链表的差别仅在于：判别链表中最后一个结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。基本操作不再赘述，留给读者自行实现。

2. 双向链表

实际上，链表的每个结点也可以包含多个数据域或多个指针域。双向链表包含两个指针域，其中一个指向前驱，另一个指向后继。如图 2-11 所示的是包含一个数据域和两个指针域（分别指向前驱和后继）的不带头结点的双向链表。

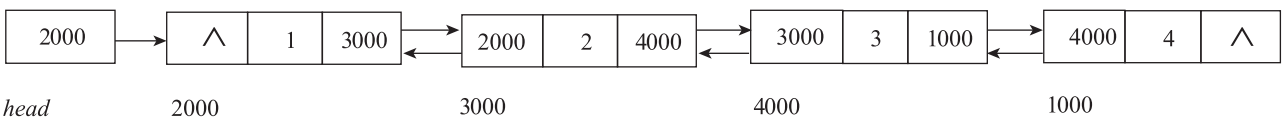


图 2-11　包含一个数据域和两个指针域的不带头结点的双向链表

设双向链表中的指向前驱的指针域为 *prior*，指向后继的指针域为 *next*，则在 *p* 所指结点之后插入

一个 *q* 所指的新结点的语句序列如下：

```
q->next=p->next;
p->next->prior=q;
p->next=q;
q->prior=p;
```

删除 *p* 所指结点的后继的语句序列（不考虑释放所删结点的空间）如下：

```
p->next->next->prior=p;
p->next=p->next->next;
```

双向链表的其他操作不再赘述，读者可自行思考和实现。

2.5 STL 之 list

1. 基础知识

STL 之 list 对应双向链表，相当于每个结点既有后继指针，又有前驱指针，则 list 可以从前往后访问，也可以从后往前访问。这里仅讨论从前往后访问 list。

使用 list 首先需要包含头文件 list。定义数据元素类型为整型 list 的代码如下：

```
list <int> myList;                                    // 用 int 类型实例化 list
```

list 的部分常用成员函数如表 2–2 所示。

表 2–2　list 的部分常用成员函数

成员函数	含义
begin()	指向首元素的迭代器
end()	指向尾元素后一个位置的迭代器
clear()	清空链表
empty()	链表判空
size()	链表长度
push_back(val)	将 val 链接到链表的表尾
push_front(val)	将 val 链接到链表的表头
reverse()	逆置链表
merge(lst[,cmp])	归并有序链表，将有序链表 lst 有序地归并到原链表（需有序）中。默认按“小于”比较，可通过指定第二个参数 cmp（比较函数）表明比较规则
sort([cmp])	链表排序，默认按非递减序排序，可通过指定参数 cmp（比较函数）表明比较规则
unique()	去除链表中的重复元素，链表需先排序
erase(pos)	删除迭代器 pos 所指元素，返回 pos 所指元素的后继的迭代器
erase(start, end)	删除迭代器区间 [start，end) 之间的元素

续表

成员函数	含义
insert(pos,val)	在迭代器 pos 所指位置插入一个值为 val 的元素并返回其迭代器
pop_back()	删除链表的尾元素
pop_front()	删除链表的首元素
remove(val)	删除链表中所有值为 val 的元素

2. 应用举例

例 2.5.1　建立顺序链表（*HLOJ 9504*[①]）

先输入一个整数 n，再输入 n 个整数，按照输入的顺序建立链表，并遍历所建立的链表，输出这些数据。

例如，输入：5 1 2 3 4 5，输出：5 4 3 2 1。要求用 STL 之 list 实现。

可以把输入的数据不断用 push_back 函数插入表尾。输出链表则通过迭代器（类似于指针）从头开始顺序访问，具体代码如下：

```
#include <iostream>
#include <list>                                  // 包含 list 头文件
using namespace std;
list<int> createList(int n) {                   // 建立顺序链表
    int t;
    for(int i=0; i<n; i++) {
        cin>>t;
        myList.push_back(t);                     // 元素 t 链接到表尾
    }
    return myList;
}
void prtList(list<int> myList) {                 // 输出链表，每两个数据之间留一个空格
    list<int>::iterator it;                      // list<int>::iterator 类型迭代器
    for(it=myList.begin(); it!=myList.end(); it++) {
        if (it!=myList.begin())                  // 若 it 所指的不是第一个数据
            cout<<" ";                           // 则先输出一个空格
        cout<<*it;                               // 输出 it 所指的数据
    }
    cout<<endl;
}
int main() {
    int n;
    cin>>n;                                      // 输入数据个数
    list<int> lst=createList(n);                 // 建立顺序链表
    prtList(lst);
    return 0;
}
```

① HLOJ 为绍兴文理学院原有 OJ，9504 为本例在 HLOJ 中的编号。

注意，STL 之迭代器没有重载小于号，因此不能用条件 *it*<*myList*.end() 表示链表没有结束，而应用 *it*!=*myList*.end() 表达。如果要建立逆序链表，只需把语句“myList.push_back(t);”改为“myList.push_front(t);”即可。

注意，在用成员函数 erase 删除 *myList* 中迭代器 *it* 所指元素时，最好不要写成“myList.erase(it);”，因为此语句已使迭代器 *it* 销毁，若需要用 *it* 继续指向后续元素，需使 *it* 重新有明确指向，即写成“it=myList.erase(it);”或“myList.erase(it++);”。

读者可自行编程测试 list 常用成员函数的使用。

3. 运用 list 实现大整数加法

例 2.5.2　大整数加法（HLOJ 9515）

给定非负整数 *A* 和 *B*，请计算 *A*+*B* 的值。要求采用链表或 list 完成。

例如，输入：2222222222222222 58944444444444444444，则输出：589466666666666666666。

采用链表或 list 实现两个大整数加法的基本思路：两个非负整数作为字符串输入，并用输入的字符串创建逆序链表，再根据“右对齐、逐位相加”的方法进行运算。由于创建的是逆序链表，因此按从头开始、顺序访问即可实现右对齐逐位相加。另外，在进位处理方面，可以用一个整型变量表示，其初值一开始设为 0，在计算过程中把其加到当前和中，并不断更新为最新的进位。

这里给出使用 list 的具体实现代码，读者可参考此代码写出使用自定义链表求解本例的代码。

```
#include <iostream>
#include <list>
#include <string>
using namespace std;
// 利用 list 实现非负整数加法，两个整数以链表形式存储在 La 和 Lb 中，结果存放在 Lc 中
list<int> bigAddList(list<int> La, list<int> Lb) {
    list<int> Lc;
    list<int> ::iterator it1,it2;
    it1=La.begin();
    it2=Lb.begin();
    int carry=0;                              // 保存进位
    // 当两个链表没有处理完毕时，逐位相加
    while(it1!=La.end() || it2!=Lb.end()) {
        int c=carry;                          // c 暂存当前和
        if(it1!=La.end()) {                   // 若第一个链表未结束，则把相应数位加到 c 中
            c=c + *it1;
            it1++;
        }
        if(it2!=Lb.end()) {                   // 若第二个链表未结束，则把相应数位加到 c 中
            c=c + *it2;
            it2++;
        }
        carry=c/10;                           // 求得新的进位
        Lc.push_front(c%10);                  // 相加得到的当前和的余数插入结果链表头部
    }
```

```
    if(carry>0) Lc.push_front(carry);// 处理最后的进位
    return Lc;
}

int main() {
    string s,t ;
    cin>>s>>t;                              // 非负整数作为字符串输入
    list <int> La, Lb, Lc;
    for(int i=0; i<s.size(); i++) {         // 用第一个字符串创建逆序链表
        La.push_front(s[i]-'0');
    }
    for(int i=0; i<t.size(); i++) {         // 用第二个字符串创建逆序链表
        Lb.push_front(t[i]-'0');
    }
    Lc=bigAddList(La, Lb);                  // 调用 bigAddList 求解两个非负整数相加
    // 遍历输出结果
    for (list<int>::iterator it=Lc.begin(); it!=Lc.end(); it++) {
        cout<<*it;
    }
    cout<<endl;
    return 0;
}
```

设两个整数中最长的整数有 n 位，则 bigAddList 算法的时间复杂度为 O(n)。

若把 bigAddList 函数中的 push_front 改为 push_back，则要如何修改以上代码？读者可以自行思考并实现。

2.6 引例求解

因序列可分别用顺序表或链表存储，故引例可采用顺序表或链表实现。

1. 采用自定义顺序表求解

若采用顺序表 a、b 存储序列，则可先用指针（下标）i、j 分别指向顺序表 a、b 中的首元素，当 i、j 都还有指向时循环处理：比较它们所指的元素，将小者存入结果顺序表 c 的尾部（设以指针 k 指向），并使指向小者的指针（i 或 j）和 k 都增 1。最后，将 a 或 b 中剩余的元素拷贝到 c 的尾部。

采用自定义顺序表求解的具体代码如下。

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

struct SqList {                         // 顺序表结构体
    int *elem;                          // 存放动态数组的首地址
    int length;                         // 表长
    void Init(int n);                   // 初始化顺序表
    void Create(int n);                 // 创建顺序表
```

```
    void Traverse();                        // 遍历顺序表
};

// 将非递减序的顺序表 a、b 合并为非递减序的顺序表 c
void mergeList(SqList a, SqList b, SqList &c) {
    // 求得 a、b 的长度存于 m、n 中
    int m=a.length, n=b.length;
    c.length=m+n;                           // 置 c 的长度为 a、b 长度之和
    int i=0, j=0, k=0;                      // i、j、k 分别指向 a、b、c 的首元素位置
    while(i<m && j<n) {                     // 当 i、j 都还有指向时循环处理
        if (a.elem[i]<=b.elem[j])           // 将 i、j 所指元素中的小者存入 c 中 k 所指位置
            c.elem[k++]=a.elem[i++];        // 将 i 所指元素存入 c 的尾部，并使 i、k 都增 1
        else
            c.elem[k++]=b.elem[j++];        // 将 j 所指元素存入 c 的尾部，并使 j、k 都增 1
    }

    while(i<m) c.elem[k++]=a.elem[i++];     // 若 a 中还有剩余元素，则将它们拷贝到 c 的尾部
    while(j<n) c.elem[k++]=b.elem[j++];     // 若 b 中还有剩余元素，则将它们拷贝到 c 的尾部
}

int main() {
    int T;
    cin>>T;
    while(T--) {
        SqList a, b, c;                     // 定义顺序表
        int m, n;
        cin>>m;
        a.Create(m);                        // 调用成员函数 Create 创建顺序表
        cin>>n;
        b.Create(n);                        // 调用成员函数 Create 创建顺序表
        c.Init(m+n);                        // 创建顺序表 c，长度为 m+n
        mergeList(a, b, c);                 // 调用算法 mergeList 合并顺序表
        c.Traverse();                       // 调用成员函数 Traverse 遍历顺序表
    }
    return 0;
}

void SqList::Traverse() {                   // 遍历顺序表
    for(int i=0; i<length; i++) {
        if(i>0) cout<<" ";
        cout<<elem[i];
    }
    cout<<endl;
}

void SqList::Init(int n) {                  // 初始化顺序表
    elem=new int[n];
    length=0;
}
```

```
void SqList::Create(int n) {                          // 创建顺序表
    Init(n);
    length=n;
    for(int i=0; i<length; i++) cin>>elem[i];
}
```

2. 采用一维数组求解

直接用一维数组作为顺序表可简化编程。

采用一维数组求解的具体代码如下。

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

// 将非递减序的 a、b 数组（长度分别为 m、n）合并为非递减序的 c 数组
void mergeList(int a[], int b[], int c[], int m, int n) {
    int i=0, j=0, k=0;                        // i、j、k 分别指向 a、b、c 的首元素位置
    while(i<m && j<n) {                       // 当 i、j 都还有指向时循环处理
        if (a[i]<=b[j])                       // 将 i、j 所指元素中的小者存入 c 数组 k 所指位置
            c[k++]=a[i++];                    // 将 a[i] 存入 c[k]，并使 i、k 都增 1
        else
            c[k++]=b[j++];                    // 将 b[j] 存入 c[k]，并使 j、k 都增 1
    }

    while(i<m) c[k++]=a[i++];                 // 若 a 中还有剩余元素，则将它们拷贝到 c 中
    while(j<n) c[k++]=b[j++];                 // 若 b 中还有剩余元素，则将它们拷贝到 c 中
}

int main() {
    int T;
    cin>>T;
    while(T--) {
        int i, m, n,  a[100], b[100], c[200];
        cin>>m;
        for(i=0; i<m; i++) cin>>a[i];         // 输入 a 序列
        cin>>n;
        for(i=0; i<n; i++) cin>>b[i];         // 输入 b 序列
        mergeList(a, b, c, m, n);             // 将 a、b 序列合并到 c 数组中
        for(i=0; i<m+n; i++) {                // 遍历输出数组 c 中的元素
            if (i>0) cout<<" ";
            cout<<c[i];
        }
        cout<<endl;
    }
    return 0;
}
```

3. 采用 vector 求解

直接用一维 vector 作为顺序表可简化编程。

采用 vector 求解的具体代码如下。

```
#include<iostream>
#include<vector>
using namespace std;

// 将非递减序的 a、b 向量合并为非递减序的 c 向量
void mergeList(vector<int> a, vector<int> b, vector<int> &c) {
    // 求得 a、b 的长度存于 m、n 中
    int m=a.size(), n=b.size();
    int i=0, j=0, k=0;                          // i、j、k 分别指向 a、b、c 的首元素位置
    while(i<m && j<n) {                         // 当 i、j 都还有指向时循环处理
        if (a[i]<=b[j])                         // 将 i、j 所指元素中的小者存入 c 数组 k 所指位置
            c[k++]=a[i++];                      // 将 a[i] 存入 c[k]，并使 i、k 都增 1
        else
            c[k++]=b[j++];                      // 将 b[j] 存入 c[k]，并使 j、k 都增 1
    }

    while(i<m) c[k++]=a[i++];                   // 若 a 中还有剩余元素，则将它们拷贝到 c 中
    while(j<n) c[k++]=b[j++];                   // 若 b 中还有剩余元素，则将它们拷贝到 c 中
}

int main() {
    int T;
    cin>>T;
    while(T--) {
        int i, m, n;
        cin>>m;
        vector<int> a(m);                       // 定义长度为 m 的向量 a
        for(i=0; i<m; i++) cin>>a[i]; // 输入 a 序列

        cin>>n;
        vector<int> b(n);                       // 定义长度为 n 的向量 b
        for(i=0; i<n; i++) cin>>b[i];           // 输入 b 序列

        vector<int> c(m+n);
        mergeList(a, b, c);                     // 将 a、b 序列合并到 c 向量中

        for(i=0; i<c.size(); i++) {             // 遍历输出向量 c 中的元素
            if (i>0) cout<<" ";
            cout<<c[i];
        }
        cout<<endl;
    }
    return 0;
}
```

因需将更新的结果通过参数返回，故用来存放合并后结果的参数 *c* 应设为引用参数（在其前面加 &）。

4. 采用自定义链表求解

若采用带头结点的单链表存储序列，则要求将两个非递减序的单链表 *La*、*Lb* 合并为一个非递减序的单链表 *Lc*。一种算法思想是从头开始分别用一个指针指向链表 *La*、*Lb* 中的首元素，然后逐个比较两个链表中的当前元素，把其中的小者链接到结果链表的尾部。

采用自定义链表求解的具体代码如下。

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

struct LNode {
    int data;                              // 数据域
    LNode* next;                           // 指针域
};
struct LinkList {
    LNode *head;                           // 头指针（带头结点）
    void Create(int n);                    // 建立含 n 个结点的单链表
    void Traverse();                       // 遍历，并输出内容
};
// 合并升序链表（允许重复元素）
void mergeList(LinkList &La,LinkList &Lb,LinkList &Lc) {
    LNode *pa,*pb,*pc;
    pa=La.head->next;                      // pa 指向第一个链表 La 的首元素
    pb=Lb.head->next;                      // pb 指向第二个链表 Lb 的首元素
    pc=Lc.head=La.head;                    // 用 La 的头结点作为 Lc 的头结点
    while(pa!=NULL && pb!=NULL) {          // 当两个链表都没有结束时进行比较并处理
        if(pa->data<=pb->data) {           // pa 所指元素不大于 pb 所指元素
            pc->next=pa;                   // pa 所指结点链接到结果链表的最后
            pc=pc->next;                   // pc 指向其后继
            pa=pa->next;                   // pa 指向第 1 个链表 La 的下一个结点
        }
        else {                             // pa 所指元素大于 pb 所指元素
            pc->next=pb;                   // pb 所指结点链接到结果链表的最后
            pc=pc->next;                   // pc 指向其后继
            pb=pb->next;                   // pb 指向第 2 个链表 Lb 的下一个结点
        }
    }
    pc->next= pa!=NULL ? pa : pb;          // 插入 La 或 Lb 中的剩余结点
    delete Lb.head;                        // 释放 Lb 的头结点
}
int main() {
    int T;
    cin>>T;
    while(T--) {
        LinkList a,b,c;                    // 定义链表 a、b、c
        int n;
        cin>>n;                            // 输入第一个链表的长度 n
        a.Create(n);                       // 调用 Create 成员函数创建 a 链表
        cin>>n;
        b.Create(n);                       // 调用 Create 成员函数创建 b 链表
```

```
        mergeList(a,b,c);                    // 调用 mergeList 函数合并 a、b 链表
        c.Traverse();                        // 输出结果链表
    }
    return 0;
}
// 创建带头结点的单链表
void LinkList::Create(int n) {
    head=new LNode;                          // 创建头结点
    head->next=NULL;                         // 头结点的指针域置为空指针
    LNode *q=head;                           // q 为尾指针，指向尾结点
    for (int i=0; i<n; i++) {
        LNode* p=new LNode;                  // 创建新结点由 p 指向
        cin>>p->data;                        // 输入元素值
        q->next=p;                           // 新结点链接到尾结点之后
        q=p;                                 // q 指向新的尾结点
    }
    q->next=NULL;                            // 尾结点指针域置为空指针
}
// 遍历链表
void LinkList::Traverse() {
    LNode* p=head->next;                     // p 指向第一个数据结点
    while (p!=NULL) {                        // 当链表未结束时循环
        if (p!=head->next) cout<<" ";        // 若不是第一个数据，则先输出一个空格
        cout<<p->data;                       // 输出 p 所指数据域值
        p=p->next;                           // p 指向下一个结点
    }
    cout<<endl;
}
```

5. 采用 STL 之 list 求解

可用 STL 之 list 简化编程，对于两个非递减序的链表 *La* 和 *Lb*，直接用语句“La.merge(Lb);”即可将 *Lb* 合并到 *La* 中。

采用 list 求解的具体代码如下。

```
#include<iostream>
#include<list>
using namespace std;

int main() {
    int T;
    cin>>T;
    while(T--) {
        int n,t;
        list<int> La, Lb;                    // 定义链表 La、Lb
        cin>>n;
        for(int i=0; i<n; i++) {             // 创建链表 La
            cin>>t;
            La.push_back(t);                 // 将 t 链接到链表 La 的尾部
        }
```

```
        cin>>n;
        for(int i=0; i<n; i++) {                    // 创建链表 La
            cin>>t;
            Lb.push_back(t);                        // 将 t 链接到链表 La 的尾部
        }
        La.merge(Lb);                               // 将链表 Lb 合并到链表 La 中
                                                    // 遍历链表 La，输出各个元素，每两个数据之间留一空格
        for(list<int>::iterator it=La.begin(); it!=La.end(); it++) {
            if (it!=La.begin()) cout<<" ";
            cout<<*it;
        }
        cout<<endl;
    }
    return 0;
}
```

对于支持 C++ 11 标准的编译器，遍历链表 *La* 的 for 循环中的“list<int>::iterator”可用 auto 代替。auto 关键字可根据 =（赋值符号）右边表达式的类型自动确定 = 左边变量的类型。

习题 2

一、选择题

1. 等概率情况下，在表长为 *n* 的顺序表中插入一个元素所需移动的元素平均个数为（　）。

A. $(n-1)/2$　　B. $n/2$　　C. $(n+1)/2$　　D. $n-i$

2. 假设某个带头结点的单链表的头指针为 *head*，则判定该表为空表的条件是（　）。

A. *head*==NULL　　B. *head->next*==NULL

C. *head*!=NULL　　D. *head->next==head*

3. 在不带头结点的单链表中，某个结点的指针域指向该结点的（　）。

A. 直接前趋　　B. 直接后继　　C. 开始结点　　D. 尾端结点

4. 线性表 *L* 在（　）情况下适用于使用链式结构实现

A. 需不断对 *L* 进行删除插入　　B. 需经常修改 *L* 中的结点值

C. *L* 中含有大量的结点　　D. *L* 中含有少量的结点

5. 单链表的结点指针域为 *next*，其头结点由指针 *head* 指向，则删除第一个数据结点（由指针 *p* 指向）的语句序列为（　）。

A. *p->next = head->next*;　　B. *head->next = p*;

C. *p = head->next*;　　D. *head->next = p->next*;

6. 创建一个包括 *n* 个结点的有序单链表的算法的时间复杂度是（　）。

A. O(1)　　B. O(*n*)　　C. $O(n^2)$　　D. $O(n\log_2 n)$

7. 单链表的指针域为 *next*，其头结点由指针 *head* 指向，则把指针 *p* 指向的结点链接到头结点之后的语句序列为（　）。

A. *p->next=head->next*; *head->next=p*;　　B. *head->next=p*; *p->next=head->next*;

C. *head->next = p->next*; *p= head->next*;　　D. *p->next=head*; *head=p*;

8.（　）结构中的结点最多只有一个前驱和一个后继。

A. 二叉树　　B. 图　　C. 线性表　　D. 以上都不是

9. 设两个单链表不带头结点且只提供头指针，则将长度为 n 的单链表链接在长度为 m 的单链表之后的算法的时间复杂度为（　）。

A. O(1)　　B. O(n)　　C. O(m)　　D. O($m+n$)

10. 线性表 $L=(a_1, a_2, \cdots, a_n)$，下列说法正确的是（　）。

A. 每个元素都有一个直接前驱和一个直接后继

B. 表中至少有一个元素

C. 表中元素要有序

D. 除第一个和最后一个元素外，其他元素都有且仅有一个直接前驱和一个直接后继

11. 在 n 个数据元素的顺序表中，算法时间复杂度为 O(1) 的操作是（　）。

（1）访问第 i 个结点（$1 \leqslant i \leqslant n$）。

（2）求第 i 个结点的直接前驱（$2 \leqslant i \leqslant n$）。

（3）求第 i 个结点的直接后继（$1 \leqslant i \leqslant n-1$）。

（4）在第 i 个结点后插入一个新结点（$0 \leqslant i \leqslant n$）。

（5）删除第 i 个结点（$1 \leqslant i \leqslant n$）。

（6）排序。

A.（1）（2）（3）（4）（5）　　B.（1）（2）（3）

C.（4）（5）　　D.（6）

12. 在 n 个数据元素的单链表（仅有头指针）中，算法时间复杂度为 O(1) 的操作是（　）。

（1）访问第 i 个结点（$1 \leqslant i \leqslant n$）。

（2）求第 i 个结点的直接前驱（$2 \leqslant i \leqslant n$）。

（3）求第 i 个结点的直接后继（$1 \leqslant i \leqslant n-1$）。

（4）在第 i 个结点后插入一个新结点（$0 \leqslant i \leqslant n$）。

（5）删除第 i 个结点（$1 \leqslant i \leqslant n$）。

（6）排序。

A.（1）（2）（3）　　B.（4）（5）

C.（6）　　D. 以上都不是

13. 线性表 $(a_1, a_2, \cdots, a_n)$ 以顺序存储结构存储时，访问第 i 位置元素的时间复杂度为（　）。

A. O(1)　　B. O(n)　　C. O(i)　　D. O($i-1$)

14. 线性表 $(a_1, a_2, \cdots, a_n)$ 以仅有头结点的单链表存储时，访问第 i 位置元素的时间复杂度为（　）。

A. O(1)　　B. O(n)　　C. O(i)　　D. O($i-1$)

15. 不带头结点的单链表（头指针为 *head*）为空的判定条件是（　）。

A. *head*==NULL　　B. *head->next*==NULL

C. *head->next=head*　　D. *head*!=NULL

16. 带头结点的循环单链表（头指针为 *head*）为空的判定条件是（　）。

A. *head*==NULL　　B. *head->next*==NULL

C. *head->next==head*　　D. *head*!=NULL

17. 创建一个包括 n 个结点的单链表的时间复杂度是（　）。

A. O(1)　　B. O(n)　　C. O(n^2)　　D. O($n\log_2 n$)

18. 在单链表中，指针域为 *next*，要将 *q* 所指结点链接到 *p* 所指结点之后，其语句序列应为（　　）。

A. *q->next=p->next*; *p->next=q*;　　B. *p->next=q*; *q->next=p->next*;

C. *q->next=p*+1; *p->next=q*;　　D. *p->next=q*; *q->next=p*;

19. 在双向链表中，前驱指针为 *prior*，后继指针为 *next*，在 *p* 指针所指的结点后插入 *q* 所指向的新结点，其语句序列是（　　）。

A. *q->prior=p*; *q->next=p->next*; *p->next=q*; *p->next->prior=q*;

B. *q->prior=p*; *q->next=p->next*; *p->next->prior=q*; *p->next=q;*

C. *p->next=q*; p-*>next->prior=q*; *q->prior=p*; *q->next=p->next*;

D. *p->next=q*; *q->prior=p*; *p->next->prior=q*; *q->next=q*;

20. 在双向链表中，前驱指针为 *prior*，后继指针为 *next*，删除 *p* 所指的结点的语句序列为（　　）。

A. *p->prior->next=p*; *p->prior=p->prior->prior*;

B. *p->prior*=p-*>next->next*; *p->next=p->prior->prior*;

C. *p->next->prior=p->prior*; *p->prior->next=p->next*;

D. *p->next=p->next->next*; *p->next->prior=p*;

二、应用题

1. 在如图 2–12 所示的静态链表中数组 *A* 中链接存储了一个线性表，表头指针为 *A*[0].*next*，试写出该线性表。

下标	0	1	2	3	4	5	6	7
next	3	5	7	2	0	4		1
data		60	50	78	90	34		40

图 2–12　静态链表

2. 对于带头结点的单循环链表，要求画出插入、删除结点的示意图，并写出相应的操作语句。

三、编程题

1. 顺序表中插入元素（HLOJ 9501）。

设顺序表中的数据元素递增有序。请写一算法，将 *x* 插入顺序表的适当位置上，以保持该表的有序性。

输入：

首先输入一个整数 *T*，表示测试数据的组数，然后是 *T* 组测试数据。每组测试数据先在第一行输入数据个数 *n*（*n* ≤ 15）及 *n* 个递增有序的整数；再在第二行输入一个整数 *x*。

输出：

对于每组测试，将插入 *x* 后保持该顺序表递增有序的顺序表中的整数输出，每两个整数间留一个空格。

输入样例：

```
2
10 124 392 410 487 757 940 989 3633 4228 4977
567
14 113 238 307 396 487 536 616 798 834 1697 2498 2640 2665 3271
100
```

输出样例：

```
124 392 410 487 567 757 940 989 3633 4228 4977
100 113 238 307 396 487 536 616 798 834 1697 2498 2640 2665 3271
```

2. 顺序表中删除元素（HLOJ 9502）。

输入若干个不超过 100 的整数，存储到顺序表中，请写一算法，删除顺序表中所有值为 *item* 的数据元素，然后保持原数据顺序输出。

输入：

首先输入一个整数 *T*，表示测试数据的组数，然后是 *T* 组测试数据。每组测试数据先在第一行输入数据个数 *n*（$n \leqslant 40$）及 *n* 个不超过 100 的整数；再在第二行输入一个整数，表示要删除的的数据元素 *item*。

输出：

对于每组测试，保持原数据顺序输出删除顺序表中所有值为 *item* 后的数据元素，每两个数据之间留一个空格，若删除所有值为 *item* 的数据元素后为空表，则输出“empty”。

输入样例：

```
2
30 38 76 77 53 38 38 80 16 79 38 48 50 59 88 15 47 15 95 39 38 46 73 52 77 26 28 31 98 60 26
38
10 1 1 1 1 1 1 1 1 1
1
```

输出样例：

```
76 77 53 80 16 79 48 50 59 88 15 47 15 95 39 46 73 52 77 26 28 31 98 60 26
empty
```

3. 两个有序单链表求差集（HLOJ 9507）。

依次输入递增有序若干个不超过 100 的整数，分别建立两个递增有序单链表，分别表示集合 *A* 和集合 *B*。设计算法求出两个集合 *A* 和 *B* 的差集（仅由在集合 *A* 中出现而不在集合 *B* 中出现的元素所构成的集合），并存放于 *A* 链表中。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。然后输出结果链表。测试数据保证结果链表至少存在一个元素。

输入：

首先输入一个整数 *T*，表示测试数据的组数，然后是 *T* 组测试数据。每组测试数据先在第一行输入数据个数 *n* 及 *n* 个依次递增有序的不超过 100 的整数，再在第二行输入数据个数 *m* 及 *m* 个依次递增有序的不超过 100 的整数。

输出：

对于每组测试，输出集合 *A* 与集合 *B* 的差集的单链表，每两个数据之间留一个空格。

输入样例：

```
1
11 10 14 23 25 26 31 34 42 51 65 90
10 10 41 42 46 51 58 59 60 68 97
```

输出样例：

```
14 23 25 26 31 34 65 90
```

4. 拆分单链表（HLOJ 9508）。

输入若干个绝对值不超过 100 的整数，建立单链表 *A*，设计算法将单链表 *A* 分解为两个具有相同结构的链表 *B*、*C*，其中链表 *B* 的结点为链表 *A* 中值小于零的结点，而链表 *C* 的结点为链表 *A* 中值大于零的结点（链表 *A* 的元素类型为整型，要求链表 *B*、*C* 利用链表 *A* 的结点，不另外占用其他的存储空间，若采用带头结点的单链表实现则允许再申请一个头结点）。然后分两行按原数据顺序输出链表 *B*、*C*。测试数据保证每个结果链表至少存在一个元素。

输入：

首先输入一个整数 *T*，表示测试数据的组数，然后是 *T* 组测试数据。每组测试数据在一行上输入数据个数 *n* 及 *n* 个不含整数 0 且绝对值不超过 100 的整数。

输出：

对于每组测试，分两行按原数据顺序输出链表 *B*、*C*，每行的每两个数据之间留一个空格。

输入样例：

```
1
10 49 53 -26 79 -69 -69 18 -96 -11 68
```

输出样例：

```
-26 -69 -69 -96 -11
49 53 79 18 68
```

5. 链表的逆置（HLOJ 9509）。

输入若干个不超过 100 的整数，建立单链表，然后将链表中所有结点的链接方向逆置，要求仍利用原表的存储空间。输出逆置后的单链表。

输入：

首先输入一个整数 *T*，表示测试数据的组数，然后是 *T* 组测试数据。每组测试数据在一行上输入数据个数 *n* 及 *n* 个不超过 100 的整数。

输出：

对于每组测试，输出逆置后的单链表，每两个数据之间留一个空格。

输入样例：

```
1
11 55 50 45 40 35 30 25 20 15 10 5
```

输出样例：

```
5 10 15 20 25 30 35 40 45 50 55
```

6. 在单链表中删除 *mink* ~ *maxk* 的元素（HLOJ 9510）。

输入若干个值不超过 1000 的整数，建立递增有序的单链表，设计一个高效的算法，删除表中所有值大于 *mink* 且小于 *maxk* 的元素（若表中存在这样的元素），*mink* 和 *maxk* 可以与表中的元素相同，也可以不同。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据首先在第一行上输入数据个数 n 及 n 个递增有序的整数，存储在带头结点的单链表中；然后在下一行中输入整数 *mink*、*maxk*（*mink*<*maxk*），表示要删除表中所有值大于 *mink* 且小于 *maxk* 的元素。

输出：

对于每组测试，输出删除表中所有值大于 *mink* 且小于 *maxk* 的元素后单链表中的元素，每两个元素之间留一个空格。

输入样例：

```
1
9 8 125 251 351 467 492 508 731 742
100 300
```

输出样例：

```
8 351 467 492 508 731 742
```

7. 单链表中确定值最大的结点（HLOJ 9512）。

输入若干个不超过 100 的整数，建立单链表，然后通过一趟遍历在单链表中确定值最大的结点。输出该结点的值及其序号。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据在一行上输入数据个数 n 及 n 个不超过 100 的整数。

输出：

对于每组测试，输出单链表中值最大的结点的值和该结点的序号。输出如下：

“max=d_{max} num=d_{num}”

其中，d_{max} 表示最大的结点的值，d_{num} 表示最大的结点的值所在结点的序号。若有多个相同的最大值，则以首次出现的为准。

输入样例：

```
1
30 85 97 43 70 69 29 77 22 64 25 55 39 95 69 99 61 97 69 59 12 88 55 75 66 13 75 36 85 67 69
```

输出样例：

```
max=99 num=15
```

8. 约瑟夫环（HLOJ 9514）。

约瑟夫问题的一种描述是：编号为 1, 2, …, n 的 n 个人按顺时针方向围坐一圈，每人持有一个密码（正整数）。一开始以 6 作为报数上限值 m，从第一个人开始按顺时针方向自 1 开始顺序报数，报到 m 时停止报数。报 m 的人出列，将他的密码作为新的 m 值，再从他在顺时针方向上的下一个人开始重新从 1 报数，如此下去，直至所有人全部出列。要求用单向循环链表存储结构模拟约瑟夫环，并输出出列顺序。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据第一行输入 n，第二行输入 n 个不过 20 的正整数，依次表示这 n 个人所持有的密码。

输出：

对于每组测试，按照出列的顺序输出各人的编号。每两个编号之间留一个空格。

输入样例：

```
2
7
3 1 7 2 4 8 4
10
6 9 6 14 9 7 9 13 14 2
```

输出样例：

```
6 1 4 7 2 3 5
6 3 10 2 7 4 8 9 5 1
```

9. 大斐波数（HDOJ 1715）。

斐波那契（Fibonacci）数列的定义：$f(1)=f(2)=1$，$f(n)=f(n-1)+f(n-2)$，（$n \geqslant 3$）；请计算 Fibonacci 数列的第 n 项。要求用链表或 STL 之 list 实现。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据输入一个整数 n（$1 \leqslant n \leqslant 1000$）。

输出：

对于每组测试，输出 Fibonacci 数列的第 n 项 $f(n)$。

输入样例：

```
2
3
105
```

输出样例：

```
2
3928413764606871165730
```

10. 合并升序单链表（HLOJ 9506）。

各依次输入递增有序若干个不超过 100 的整数，分别建立两个单链表，将这两个递增的有序单链表合并为一个递增的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间；且合并后的单链表中不允许包含重复的数据。然后输出合并后的单链表。

输入：

首先输入一个整数 T，表示测试数据的组数，然后是 T 组测试数据。每组测试数据首先在第一行输入数据个数 n；再在第二行和第三行分别输入 n 个依次递增有序的不超过 100 的整数。

输出：

对于每组测试，输出合并后的单链表，每两个数据之间留一个空格。

输入样例：

```
1
15
14 16 19 22 23 31 33 67 68 75 76 78 81 91 95
23 25 26 37 38 39 46 52 55 68 75 78 79 86 92
```

输出样例：

```
14 16 19 22 23 25 26 31 33 37 38 39 46 52 55 67 68 75 76 78 79 81 86 91 92 95
```

习题 2 解析