21世纪高等院校计算机教材

数据结构
习题解答与实验指导

罗文劼　王苗　石强　编著

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

前　言

数据结构课程是计算机专业的一门核心课程。在应用计算机解决实际问题时，首先要解决的就是如何将问题以计算机能接受的形式表示出来，然后再将解决问题的方法步骤用计算机能识别的形式告诉计算机，让计算机自动执行求解。这一过程就是我们所说的程序设计过程。数据结构课程的内容正是在长期的程序设计实践中提炼、升华而成的，主要研究信息的逻辑结构及其基本操作在计算机中的表示和实现，是计算机科学的算法理论基础和软件设计的技术基础。在计算机专业中数据结构是操作系统、数据库原理、编译原理等后续课程的基础，在计算机专业课程的学习中起着承上启下的作用。

由于数据结构具有概念性强，内容灵活，所涉及数据的组织、存储以及操作的方法比较抽象等特点，因此，对于初学者来说，往往找不到感觉，不知道如何学习这门课，面对习题更是无从下手。作者想借编写本书的机会结合多年讲授本门课程的经验，将各章的知识要点进行归纳和总结，对难以理解的问题进行通俗的讲解和指导，对涉及到重要知识点的典型题目进行分析解答，目的是帮助读者理解数据结构的内容，尽快掌握各种数据结构的表示方法及应用实现，同时求解数据结构习题的能力也能有一个明显的提高。

此外，数据结构还是一门实践性很强的课程，只是看书本、做习题是绝对不够的。因此，在数据结构的教学中，除了课堂教学外，每周还应有不少于两个机时的实验课。在数据结构的课程实验中要解决的问题更接近于实际，不同于平时的编写功能单一的“小”算法的练习，实验是软件设计的综合训练，包括问题分析、总体结构设计、用户界面设计、程序设计基本技能和技巧，多人合作，以至一整套软件工作规范的训练和科学作风的培养。为此，作者根据数据结构课程内容的需要，给出 7 个实验题目，并对每个题目提出明确的实验要求，同时还对实验步骤和实验报告进行了规范。

本书与中国铁道出版社出版的《数据结构》教材相配套，主要内容由两部分组成：理论知识与习题解答部分和实验题目与指导部分。其中理论知识与习题解答部分与《数据结构》一书相对应，也分 10 章，每一章都由内容概述、重点难点指导、典型例题解析以及课后习题选解等部分组成；实验题目与指导部分根据数据结构课程的教学重点，给出 7 个实验题目，每个实验题目采取了统一的格式，由问题描述、数据结构设计、功能（函数）设计、界面设计、编码实现、运行与测试几个部分组成，为学生提出明确的实验要求，并对实验步骤给予指导。

在本书的编写过程中，参考了一些国内外优秀教材及数据结构习题集和辅导书。刘振鹏、张晓莉等老师对本书的编写提出许多宝贵意见，并给予大力支持，在此表示诚挚谢意。

本书在编写过程中力求概念清晰，表述正确，通俗易懂，便于自学。希望读者通过对本书的学习，能够更全面、更透彻地理解和掌握数据结构这门课程。但由于编者水平有限，书中难免出现错误或不妥之处，恳请读者批评指正，编者不胜感激。

编　者

2004 年 10 月

目　录

第一篇　理论知识与习题解答

第二篇 实验指导

第1章 概 论

本章主要介绍一些关于数据结构的概念、学习数据结构的意义，以及描述算法和分析算法的方法。

重点提示：

- 有关数据结构的术语及概念
- 理解数据结构的内容
- 理解逻辑结构、存储结构和数据运算3方面的概念及相互之间的关系
- 掌握估算时间复杂度和空间复杂度的度量方法

1-1 重点难点指导

1-1-1 相关术语

1．数据、数据元素、数据项、数据对象

数据：是信息的载体，它能够被计算机识别、存储和加工处理。它是计算机程序加工的原料，应用程序处理各种各样的数据。

数据元素：是数据的基本单位。在不同的条件下，数据元素又可称为元素、结点、顶点、记录等。

数据项：数据项是具有独立含义的最小单位。有些数据元素是由若干个数据项组成的。

数据对象或数据元素类：是具有相同性质的数据元素的集合。

2．数据类型

数据类型是一个值的集合以及在这些值上定义的一组操作的集合。

要点：值集 操作集

（1）原子类型：其值不可分解。如C语言中的整型、实型、字符型等。

（2）结构类型：其值可分解为若干成分。如C语言中的数组、结构等。

3．抽象数据类型（ADT）

是指抽象数据的组织和与之相关的操作。它可以看作是数据的逻辑结构及其在逻辑结构上定义的操作。

4．数据结构

是指数据元素之间的相互关系，即数据的组织形式，它包括以下3方面的内容：

（1）逻辑结构：数据之间的逻辑关系。

（2）存储结构：数据元素及其关系在计算机存储器内的表示。

（3）数据的运算：即对数据对象施加的操作。

5．两类逻辑结构

（1）线性结构

线性结构的逻辑特点是：若结构为非空集，有且仅有一个开始结点和一个终端结点，并且所有结点都最多只有一个直接前趋和直接后继，如线性表。

要点：只有一个直接前趋和直接后继。

（2）非线性结构

非线性结构的逻辑特点是：一个结点可能有多个直接前趋和直接后继。如树形结构和图形结构。

要点：可能有多个直接前趋和多个直接后继。

6．数据逻辑结构的 4 种基本形态

（1）集合结构：数据元素间的关系是“属于同一个集合”。

（2）线性结构：数据元素之间存在着一对一的关系。

（3）树形结构：数据元素之间存在着一对多的关系。

（4）图形结构：数据元素之间存在着多对多的关系。

7．4 种常见的存储结构

（1）顺序存储

顺序存储方法是把逻辑上相邻的元素存储在物理位置相邻的存储单元中，由此得到的存储表示称为顺序存储结构。顺序存储结构是一种最基本的存储表示方法，通常借助于程序设计语言中的数组来实现。

要点：物理位置相邻的存储单元。

（2）链式存储

链式存储方法对逻辑上相邻的元素不要求其物理位置相邻，元素间的逻辑关系通过附设的指针字段来表示，由此得到的存储表示称为链式存储结构，链式存储结构通常借助于程序设计语言中的指针类型来实现。

要点：不要求物理位置相邻的存储单元，元素间的逻辑关系通过附设的指针域来表示。

（3）索引存储方式

要点：是通过建立索引表存储结点信息的方法，其中索引表一般存储结点关键字和一个地点信息，可通过该地址找到结点其他信息。有稠密索引和稀疏索引之分。

（4）散列存储方式

要点：是根据结点的关键字直接计算出该结点的存储地址的方法。

1-1-2 算法的描述和分析

1．算法

算法是对特定问题求解步骤的一种描述，是指令的有限序列。其中每一条指令表示一个或多个操作。一个算法应该具有下列特性：

（1）有穷性。一个算法必须在有穷步之后结束，即必须在有限时间内完成。

（2）确定性。算法的每一步必须有确切的定义，无二义性。算法的执行对应着的相同的输入仅有惟一的一条执行路径。

（3）可行性。算法中的每一步都可以通过已经实现的基本运算的有限次执行得以实现。

（4）输入。一个算法具有零个或多个输入，这些输入取自特定的数据对象集合。

（5）输出。一个算法具有一个或多个输出，这些输出同输入之间存在某种特定的关系。

2．对算法的设计要求

（1）正确。算法的执行结果应当满足预先规定的功能和性能要求。

（2）可读。一个算法应当思路清晰、层次分明、简单明了、易读易懂。

（3）健壮。当输入不合法数据时，应能做适当处理，不致于引起严重后果。

（4）高效。有效地使用存储空间和有较高的时间效率。

3．算法的性能分析与度量

（1）时间复杂度

① 某算法的时间复杂度是执行该算法所耗费的时间。通常某算法的时间复杂度是问题规模 n 的函数 T(n)。

② 大 O 记法：表示算法的渐进时间复杂度。

③ 常见的渐进时间复杂度有：

$$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$$

（2）空间复杂度

一个算法的空间复杂度是指该算法所耗费的存储空间。它通常也是问题规模 n 的函数 T(n)。

1-2 典型例题解析

1-2-1 选择题

1．数据结构是一门研究非数值计算程序设计中计算机的__①__以及它们之间的__②__和运算等的学科。

①　A．操作对象　　B．计算方法　　C．逻辑存储　　D．数据映像

②　A．结构　　B．关系　　C．运算　　D．算法

【分析】数据结构是一门研究非数值计算程序设计中计算机的操作对象以及它们之间关系和运算等的学科。

【答案】① A　② B

2．从逻辑上可以把数据结构分为_____。

A．动态结构和静态结构　　B．紧凑结构和非紧凑结构

C．线性结构和非线性结构　　D．内部结构和外部结构

【分析】数据的逻辑结构有两类，分别是线性结构和非线性结构。

【答案】C

3．线性结构的顺序存储结构是一种__①__的存储结构，线性结构的链式存储是一种__②__的存储结构。

A．随机存取　　B．顺序存取　　C．索引存取　　D．散列存取

【分析】顺序存储结构是一种随机存取结构，链式存储结构是一种顺序（一个个结点的查找）存取结构。

【答案】① A　② B

4．下面程序的时间复杂度为_____。

```
for (i=0;i<m;i++)
  for (j=0;j<n;j++)
    A[i][j]=i*j;
```

A．$O(m^2)$　　B．$O(n^2)$　　C．$O(m\times n)$　　D．$O(m+n)$

【分析】第一个 for 循环执行 m 次，第二个 for 循环执行 n 次，两个 for 循环嵌套起来共执行 m×n 次。

【答案】C

1-2-2 判断题

1．顺序存储方式只能用于线性结构，不能用于非线性结构。

【分析】顺序存储方式能用于存储非线性结构。如存储树形结构中，完全二叉树的数组存储和堆排序中堆的存储。

【答案】错误

2．基于某种逻辑结构之上的运算，其实现是惟一的。

【分析】基于某种逻辑结构，其存储结构不是惟一的，因此运算也就不惟一。

【答案】错误

3．数据元素是数据的最小单位。

【分析】数据元素是数据的基本单位，数据元素可以由数据项组成，且数据项是数据的最小单位。

【答案】错误

4．算法可以用不同的语言描述，如果用 C 语言或 PASCAL 语言等高级语言来描述，则算法实际上就是程序了。

【分析】算法用各种计算机语言描述则表现为一个程序但并不等于程序，因为程序的逻辑不一定能满足有穷性。

【答案】错误

5．数据结构是带有结构的数据元素的集合。

【分析】若把数据结构进行形式化描述，则可从逻辑上认为数据结构 DS 是数据元素的集合 D 和 D 上关系的集合 R 所构成的二元组：DS＝（D，S），这里关系 R 用于描述数据之间的逻辑关系，即数据的结构。

【答案】正确

1-2-3 填空题

1．线性结构中元素的关系是____①____，树形结构中元素的关系是____②____，图形中元素的关系是____③____。

【答案】①一对一；② 一对多；③多对多

2．算法的 5 个重要特性是_____、_____、_____、_____、_____。

【答案】有穷性；确定性；可行性；输入；输出

1-3 课后习题选解

1．简述下列术语：数据，数据元素，数据对象，数据结构，存储结构，线性结构，算法，数据类型。

参见教材。

2．说明数据结构的概念与程序设计语言中数据类型概念的区别和联系。

数据类型是一个值的集合以及在这些值上定义的一组运算的集合。如 C 语言中的 int 类型，表示集合{x|x 是整数且-32768≤x≤32767}，其运算包括＋、－、×、/、％等。

数据结构指存在特定关系的数据元素的集合。数据结构包括 3 方面的内容：①数据的逻辑结构：指数据元素之间的逻辑关系；②数据的存储结构：指数据元素及其关系在计算机中的实现方法；③数据的运算：指对数据元素施加的操作。

二者都是数据元素和运算的集合。但是数据结构还涉及数据元素之间的逻辑关系。

3．讨论顺序存储结构和链式存储结构各自的特点、适用范围，并说明在实际应用中应如何选取数据存储结构。

① 顺序存储结构用存储位置的相邻表示逻辑关系的相邻。需要占用连续的存储单元。可以随机访问数据元素。

② 链式存储结构用附加的指针表示逻辑关系。不需要占用连续的存储空间。只能顺序地访问数据元素。

③ 顺序存储结构是静态分配内存的方式，链式存储结构是动态分配内存的方式，它在需要时分配存储单元，不需要时释放存储单元。当对于数据元素需要频繁地插入、删除时应采用链式存储结构，反之，采用顺序存储结构。

4．试写一算法，从大至小依次输出顺序读入的 3 个整数 x、y 和 z 的值。

【提示】通过比较和交换使得 3 个数从大至小有序。

```
main()
{   int x,y,z,t;
    cin>>x; cin>>y;cin>>z;                    //读入 3 个整数 x、y 和 z
    if(x<y) {t=x;x=y;y=t;}
    if(x<z) {t=x;x=z;z=t;}
    if(y<z) {t=y;y=z;z=t;}
    cout<<x<<y<<z<<endl;                      //输出从大至小有序的 3 个数
}
```

5．已知 k 阶裴波那契序列的定义为

$f_0=0$，$f_1=0$，…，$f_{k-2}=0$，$f_{k-1}=1$；

$f_n=f_{n-1}+f_{n-2}+\cdots+f_{n-k}$，n=k,k+1,…

试编写求 k 阶裴波那契序列的第 m 项值的函数算法，k 和 m 均以参数的形式在参量表中出现。

【提示】采用递归的方法实现。

```
int fun(int k,int m)                          //求 k 阶裴波那契序列的第 m 项值
{    int f;
     if(m<k-1) f=0;                           //f0=0，f1=0，…，fk-2=0
     else if(m==k-1;) f=1;                    //fk-1=1
        else {   f=0;
             for(j=1;j<=k;j++)
             f+=fun(k,m-j);
         }                                    //fn=fn-1+fn-2+…+fn-k
     return(f);
}
```

6. 编写一个程序计算一元多项式 $P_n(x)=P_0+P_1x+P_2x^2+\ldots+P_nx^n$ 的值 $P_n(x)$，设 n、x 和 $P_i(0\leqslant i\leqslant n)$均为已知量，从键盘读入。你的程序中 $P_i(0\leqslant i\leqslant n)$的值是采用什么结构存储的？

【提示】将一元多项式改写为如下形式：

$$P_n(x)=P_0+P_1x+P_2x^2+\ldots+P_nx^n=(\ldots((P_nx+P_{n-1})x+P_{n-2})x+\ldots+P_1)x+P_0$$

```
main()
{ int n,i;
  double x,p,s=0;
  cin>>n; cin>>x;                          //读入 n，x 的值
  for(i=0;i<=n;i++)
  { cin>>p;
    s=s*x+p;
  }                                        //计算多项式的值
  cout<<s<< endl;
}
```

P_i 的读入次序为：P_n，P_{n-1}，…，P_1，P_0

7. 试编写算法计算 $i!\times 2^i$ 的值并存入数组 a[aaisize]的第 i 个分量中 i=1,2,…,n。假设计算机中允许的整数最大值为 Maxint，则当 n>arrsize 或对某个 $k(1\leqslant k\leqslant n)$使 $k!\times 2^k$>maxint 时，应按出错处理。可有下列 3 种不同的处理方式：

① 用 ERROR 语句终止执行并报告错误。

② 用返回值 0 或 1 实现算法以区别正确返回或错误返回。

③ 在参数表中设置一整型变量以区别正确返回或某种错误返回。

试讨论这 3 种方法各自的优缺点，并以你认为较好的方式实现。

```
int fun1(int i)                            //计算 i!
{   int j=1,s=1;
    for(j=1;j<=i;j++) s*=j;
    return(s);
}
int fun2(int i)                            //计算 2^i
{   int j=1,s=1;
    for(j=1;j<=i;j++) s*=2;
    return(s);
}
main()
{   int a[arrsize+1],i;
    for(i=1;i<=n;i++)
        a[i]=fun1(i)*fun2(i);              //计算 i!*2^i
}
```


第 2 章　线性表

线性表是一种最基本、最常用的数据结构，它有两种存储结构——顺序表和链表。本章主要介绍线性表的定义、表示和基本运算的实现。重点讨论了线性表的存储结构，以及在顺序、链式两种存储结构上基本运算的实现。

重点提示：

- 线性表的逻辑结构特征
- 线性表的顺序存储和链式存储两种存储结构的特点
- 在两种存储结构下基本操作的实现

2-1　重点难点指导

2-1-1　相关术语

1. 线性表

线性表是具有相同数据类型的 n（n≥0）个数据元素的有限序列，通常记为：$(a_1,a_2,\ldots,a_n)$，其中 n 为表长，n=0 时称为空表。

要点：一种逻辑结构，其数据元素属于相同数据类型，之间的关系是线性关系。

2. 顺序表

顺序存储的线性表。

要点：按线性表中的元素的逻辑顺序依次存放在地址连续的存储单元里，其存储特点：用物理上的相邻实现逻辑上的相邻！

3. 链表

用链表存储的线性表。

要点：链表是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，对每个结点的地址是否连续没有要求。

4. 单链表

每个结点除了数据域外还有一个指向其后继的指针域。

要点：通常将每个元素的值和其直接后继的地址作为一个结点，通过每个结点中指向后继的指针表示线性表的逻辑结构。

5. 头指针

要点：是一个指针变量，里面存放的是链表中首结点的地址，并以此来标识一个链表。如链表 H，链表 L 等，表示链表中第一个结点的地址存放在 H、L 中。通常用头指针来惟一标识一个链表。

6. 头结点

要点：附加在第一个元素结点之前的一个结点，头指针指向头结点。当该链表表示一个非空的线性表时，头结点的指针域指向第一个元素结点；为空表时，该指针域为空。

7．头结点的作用

要点：其作用有两个，一是使对空表和非空表的处理得到统一；二是在链表的第一个位置上的操作和在其他位置上的操作一致，无需特殊处理。

2-1-2 线性表的顺序存储

1．顺序表

顺序存储的线性表称为顺序表。其特点是：用一组地址连续的存储单元来依次存放线性结点，因此结点的逻辑结构和物理次序一致（这是顺序存储的核心所在）。

具体实现：在程序设计语言中，一维数组在内存中占用的存储空间就是一组连续的存储区域，因此，用一维数组来表示顺序表的数据存储区域是再合适不过的。考虑到线性表的运算有插入、删除等，即表长是可变的，因此，数组的容量需设计得足够大，设用 data[ListSize] 来表示，其中 ListSize 是一个根据实际问题定义的足够大的整数，线性表中的数据从 data[0] 开始依次顺序存放，但当前线性表中的实际元素个数可能未达到 ListSize 个，因此需用一个变量 length 记录当前线性表中实际元素的个数，因此，表空时 length=0。

这种存储思想的具体实现可以是多样的。

方法一：可以用一个数组和表示长度的变量共同完成上述思想，如：

```
DataType  data[ListSize];
int  length
```

这样表示的顺序表如图 2-1 所示。数据元素分别存放在 data[0]到 data[length-1]中。

这样使用简单方便，但有时不便管理。

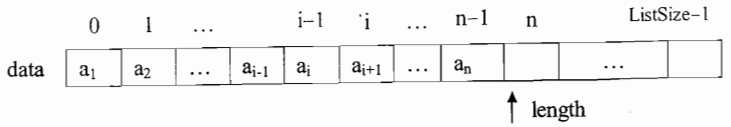


图 2-1 线性表的顺序存储示意图

方法二：从结构上考虑，通常将 data 和 length 封装成一个结构作为顺序表的类型：

```
typedef  struct
  { DataType  data[ListSize];
    int  length;  //length 表示线性表长度，也可定义为 last 表示线性表最后一个元素位置。
  } SeqList;
```

定义一个顺序表变量：SeqList　L；

这样表示的线性表如图 2-2（a）所示。表长＝L.length（或 L.last+1），线性表中的数据元素 a_1 至 a_n 分别存放在 L.data[0]至 L.data[L.length-1]中（或在 L.data[0]至 L.data[L.last]中）。

方法三：由于书中的算法用 C 语言描述，根据 C 语言中的一些规则，有时定义一个指向 SeqList 类型的指针更为方便：

```
SeqList  *L ;
```

L 是一个指针变量，线性表的存储空间通过 malloc(sizeof(SeqList)) 操作来获得。

L 中存放的是顺序表的地址，这样表示的线性表如图 2-2(b)所示。表长表示为(*L).length 或 L->length，线性表中数据元素的存储空间为：

```
L->data[0] ~ L->data[L->length-1]。
```


读者通过上述介绍的几种表示方式，进一步体会顺序存储的“理念”，做题时根据题意灵活掌握，在读写算法时注意相关数据结构的类型说明。

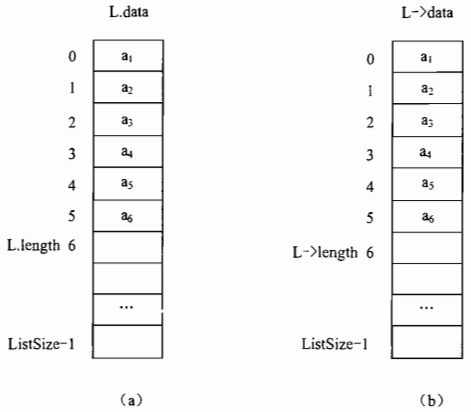


图 2-2 线性表的顺序存储示意图

2．顺序表的优缺点

优点 1：顺序表是由地址连续的向量实现的，因此具有按序号随机访问的特点。

设 a_1 的存储地址为 Loc(a_1)，每个数据元素占 d 个存储地址，则第 i 个数据元素的地址为：

$$Loc(a_i)=Loc(a_1)+(i-1)*d \qquad 1\leqslant i\leqslant n$$

这就是说只要知道顺序表首地址和每个数据元素所占地址单元的个数就可求出第 i 个数据元素的地址来，这就是顺序表具有按数据元素的序号随机存取的特点。

优点 2：存储密度高。

缺点 1：做插入和删除运算时，平均需移动大约表中一半的元素；

缺点 2：顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模，因此分配不足则会造成溢出，分配过大，又可能造成存储单元的浪费。

2-1-3 链表

1．单链表

链表是通过一组任意的存储单元来存储线性表中的数据元素的，为建立起数据元素之间的线性关系，对每个数据元素 a_i，除了存放数据元素自身的信息 a_i 之外，还需要和 a_i 一起存放其后继 a_{i+1} 所在的存储单元的地址，这两部分信息组成一个“结点”，结点的结构如图 2-3 所示，每个元素都如此。因此 n 个元素的线性表通过每个结点的指针域拉成了一个“链子”，称之为链表。因为每个结点中只有一个指向后继的指针，所以称其为单链表。

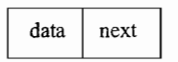


图 2-3 单链表结点结构

链表是由一个个结点构成的，结点定义如下：

```
typedef struct node
        {  DataType data;
           struct node *next;
} LinkNode, * LinkList;
```

LinkNode 是结点的类型，*LinkList 是指向 LinkNode 结点的指针类型。

作为线性表的一种存储结构，我们关心的是结点间的逻辑结构，而对每个结点的实际地址并不关心，所以通常的单链表用图 2-4 的形式表示。

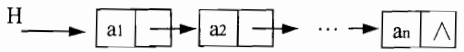


图 2-4　单链表

通常用“头指针”来标识一个单链表，如单链表 L、单链表 H 等，是指某链表的第一个结点的地址放在了指针变量 L、H 中，头指针为“NULL”则表示一个空表。

2．单循环链表

对于单链表而言，最后一个结点的指针域是空指针，如果将该链表的头指针置入该指针域，则使得链表头尾结点相连，就构成了单循环链表。

对于单链表只能从头结点开始遍历整个链表，而对于单循环链表则可以从表中任意结点开始遍历整个链表，不仅如此，有时对链表常做的操作是在表尾、表头进行，此时可以改变一下链表的标识方法，不用头指针而用一个指向尾结点的指针 R 来标识，可以使操作效率得以提高。

3．双循环链表

在单链表的结点中只有一个指向其后继结点的指针域 next，因此若已知某结点的指针为 p，其后继结点的指针则为 p->next，而找其前驱则只能从该链表的头指针开始，顺着各结点的 next 域进行，也就是说找后继的时间性能是 O(1)，找前驱的时间性能是 O(n)，如果也希望找前驱的时间性能达到 O(1)，则只能付出空间的代价：每个结点再加一个指向前驱的指针域，结点的结构如图 2-5 所示，用这种结点组成的链表称为双向链表。

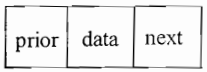


图 2-5　双向链表

双向链表结点及指针类型定义如下：

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

和单链表类似，双向链表通常也是用头指针标识。

通过双向链表中某结点的指针 p 即可以直接得到它的后继结点的指针 p->next，也可以直接得到它的前驱结点的指针 p->prior。这样在有些操作中需要找前驱时，则勿需再用循环。

设 p 指向双向循环链表中的某一结点，即 p 中是该结点的指针，则 p->prior->next 表示的是 p 所指结点之前驱结点的后继结点的指针，即与 p 相等；类似，p->next->prior 表示的是 p 所指结点之后继结点的前驱结点的指针，也与 p 相等，所以有以下等式：

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

4．静态链表

静态链表是指以数组方式存放链表的数据，数组的每个元素包含有数据域 data 和指针域 next，这里的指针域 next 与链表中的指针域不同的是其存放的是该结点逻辑上的后继结点的相对地址（即在数组中的下标），也称之为静态指针。

在以静态链表方式存储链表时，数组中含有两个单链表，一个是数据元素的链表，一个是空结点的链表，故还需要设置两个整型变量，分别用于存放链表首元素（或头结点）在数组中的位置和空结点链表的首位置。

静态链表存储定义如下：

```
#define MAXSIZE …              //足够大的数
typedef struct {
    DataType  data;
    int   next;
   }SNode;                     //结点类型
SNode sd[MAXSIZE];             //数组 sd 以静态链表方式存放链表数据
int SL, AV;                    //两个头指针变量
```

5．链式存储的优缺点

链式存储的优缺点与顺序存储互补：

优点 1：做插入和删除运算时，只需改变指针，不需移动数据。

优点 2：不需要事先分配空间，便于表的扩充。

缺点 1：存储密度降低，因为每个结点中除了存放数据元素的值还有一个指针域。

缺点 2：不具有按序号随机访问第 i 个元素的特点，必须通过标识链表的头指针（或尾指针）“顺藤摸瓜”才能找到第 i 个结点。

2-1-4　线性表的基本运算

1．基于顺序表的运算

顺序表中常做的运算有插入、删除、合并、查找等，做这些运算时，要掌握一个原则：时刻体现顺序存储的思想，即元素之间物理相邻和逻辑相邻的一致性。

（1）在顺序表中插入元素时：

① 插入元素时，检查顺序表是否已满，已满则不能做插入。上溢

② 根据具体问题确定插入位置。

③ 移动有关元素，以便为待插入的元素让出位置来。

④ 将元素插入。

⑤ 修改表长。

（2）在顺序表中删除元素时：下溢

① 检查顺序表是否已空，空则不能做插入。

② 根据具体问题确定删除元素的位置。

③ 将其后面的有关元素移动，“挤掉”被删除的元素。

④ 修改表长。

结论：

顺序表中插入一个数据元素平均需要移动（n+1）/2 个元素。具体某一次的插入中，移动数据的个数与表长和插入位置有关。

顺序表中删除一个数据元素需要平均移动 n/2 个元素。具体某一次的删除中，移动数据的个数与表长和删除元素的位置有关。

2．基于链表的运算

链表中的操作最好通过图示进行，以便清楚指针的变化情况。链表操作过程中，主要是指针的变化，因此必须清楚指针和动态结点的问题。

（1）头结点的使用

在对不带头结点的单链表中进行操作时，对第一个结点的处理和其他结点是不同的。

头结点的加入完全是为了运算的方便，它的数据域无定义，指针域中存放的是第一个数据结点的地址，空表时为空。

（2）插入

① 后插结点：设 p 指向单链表中某结点，s 指向待插入的值为 x 的新结点，将*s 插入到*p 的后面，插入示意图如图 2-6 所示。

操作如下：

```
s->next=p->next;
p->next=s;
```

要点：两个指针的操作顺序不能交换。

② 前插结点

设 p 指向链表中某结点，s 指向待插入的值为 x 的新结点，将*s 插入到*p 的前面，插入示意图如图 2-7 所示，与后插不同的是：首先要找到*p 的前驱*q，然后再完成在*q 之后插入*s，设单链表头指针为 L，操作如下：

```
q=L;
while (q->next!=p)
  q=q->next;              //找*p 的直接前驱
s->next=q->next;
q->next=s;
```

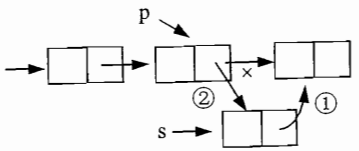


图 2-6 在*p 之后插入*s

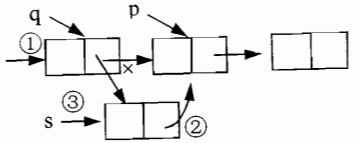


图 2-7 在*p 之前插入*s

后插操作的时间复杂性为 O(1)，前插操作因为要找 *p 的前驱，时间性能为 O(n)；其实我们更关心的是数据元素之间的逻辑关系，所以仍然可以将 *s 插入到 *p 的后面，然后将 p->data 与 s->data 交换即可，这样即满足了逻辑关系，也能使得时间复杂性为 O(1)。

由此得知：在单链表中插入一个结点必须知道其前趋结点。

③ 在单链表第 i 个元素之前插入元素 x。

i．从头指针开始找到第 i-1 个元素结点，若存在继续步骤 ii，否则结束。

ii．申请、填装新结点。

iii．将新结点插入，结束。

（3）删除

① 删除结点：

i．删除*p 的后继结点（假设存在），则可以直接完成：

```
s=p->next;   p->next=s->next;
delete s;
```

该操作的时间复杂性为 O(1)。

ii．若要删除*p，即 p 所指结点，通过示意图可见，要实现对结点*p 的删除，首先要找到*p 的前驱结点*q，然后完成指针的操作即可。

指针的操作由下列语句实现：

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

显然找*p 前驱的时间复杂性为 O(n)。

操作示意图如图 2-8 所示。

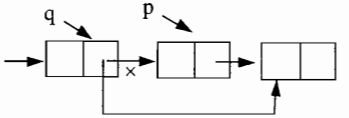


图 2-8 删除*p

当*p 有后继时，对*p 的删除可以这样处理：把*p 后继结点的值存储到结点*p 中，再删除*p 的后继结点，这样逻辑上删除了*p，而时间性能为 O(1)。

② 删除单链表的第 i 个结点 Del_LinkList(L,i)：在单链表中删除第 i 个结点，必须知道第 i-1 个结点的指针。算法思路：

i．找到第 i-1 个结点；若存在继续步骤 ii，否则结束。

ii．若存在第 i 个结点则继续步骤iii，否则结束。

iii．删除第 i 个结点，结束。

（4）查找

在单链表中按值查找或按序号查找某个元素，需通过头指针从第一个元素结点开始，逐个与 x 进行比较或记数，若查找成功，则返回指向该元素的指针，若查找失败，则返回 NULL。

（5）双向链表中结点的插入

设 p 指向双向链表中某结点，s 指向待插入的值为 x 的新结点，将*s 所指结点插入到*p 所指结点的前面，插入示意图如图 2-9 所示。

操作如下：

① s->prior=p->prior;

② p->prior->next=s;

③ s->next=p;

④ p->prior=s;

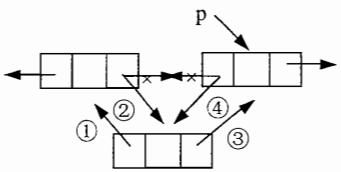


图 2-9 双向链表中的结点插入

指针操作的顺序不是惟一的，但也不是任意的，操作①必须要放到操作④的前面完成，否则*p 的前驱结点的指针就丢掉了。读者把每条指针操作的涵义搞清楚，就不难理解了。

（6）双向链表中结点的删除

设 p 指向双向链表中某结点，删除*p 所指结点。

操作示意图如图 2-10 所示。

操作如下：

① p->prior->next=p->next;

② p->next->prior=p->prior;
delete p;

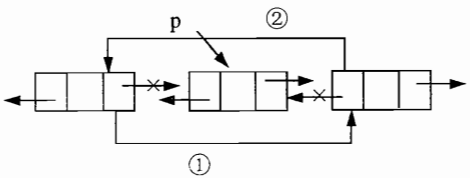


图 2-10 双向链表中结点的删除

2-2 典型例题解析

2-2-1 选择题

1．带头结点的单链表 head 为空的判断条件是（ ）。

A．head=NULL　　B．head->next=NULL

C．head->next=head　　D．head!=NULL

【分析】链表为空时，头结点的指针域为空。

【答案】B

2．若某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则采用（ ）存储方式最节省运算时间。

A．单链表　　B．仅有头指针的单循环链表

C．双链表　　D．仅有尾指针的单循环链表

【分析】根据题意要求，该线性表的存储应能够很方便地找到线性表的第一个元素和最后一个元素，A 和 B 都能很方便地通过头指针找到线性表的第一个元素，却要经过所有元素才能找到最后一个元素；选项 C 双链表若存为双向循环链表，则能很方便地找到线性表的第一个元素和最后一个元素，但存储效率要低些，插入和删除操作也略微复杂；选项 D 可通过尾指针直接找到线性表的最后一个元素，通过线性表的最后一个元素的循环指针就能很方便地找到第一个元素。

【答案】D

3．线性表的静态链表存储结构与顺序存储结构相比优点是（ ）。

A．所有的操作算法简单　　B．便于插入和删除

C．便于利用零散的存储器空间　　D．便于随机存取

【分析】静态链表采用的是链式方式存储线性表，因此其具有链式存储的特点。

【答案】B

4．若长度为 n 的线性表采用顺序存储结构，在其第 i 个位置插入一个新元素算法的时间复杂度为（ ）。

A．$O(\log_2 n)$　　B．O(1)

C．O(n)　　D．$O(n^2)$

【分析】在第 i 个位置上插入新元素需要从最后一个元素开始后移直到第 i 个元素后移为止，后移元素的次数为 n-i+1，即时间复杂度为 O(n)。

【答案】C

5．将两个各有 n 个元素的有序表归并成一个有序表，其最少的比较次数是（ ）。

A．n　　B．2n-1　　C．2n　　D．n-1

【分析】当一个表的最小元素大于另一个表的最大元素时比较次数为最少，共需 n 次。

【答案】A

6．在双循环链表 p 所指结点之后插入 s 所指结点的操作是（ ）。

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

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

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

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

【分析】由于要将 s 所指结点插入到 p 所指结点之后，*p 为*s 的前驱，*s 为*P 的新后继，而*p 的原后继为*s 的后继，*s 为*p 的原后继的前驱。在选项 A、B 和 C 中均是先执行操作 p->next=s，就是修改了*p 的后继为*s，然后再执行操作 p->next->prior=s，因此，无法使得*s 为*p 的原后继的前驱，这样的赋值会使*s 为其自身的前驱。应先执行操作 p->next->prior=s，再执行操作 p->next=s。

【答案】D

7．在一个单链表中，已知 q 所指结点是 p 所指结点的前趋结点，若在 q 和 p 之间插入 s 结点，则执行（ ）

A．s->next=p->next;p->next=s;

B．p->next=s >next;s->next=p;

C．q->next=s;s->next=p;

D．p->next=s;s->next=q;

【分析】由于是将 s 所指结点插入到 q 和 p 所指结点之间．即使其为 q 所指结点的后继，为 p 所指结点的前驱，因此 s->next 的取值应为 p，p->next 的取值无需改动，q->next 的取值应改为 s，故 A、B 和 D 均是错误的。

【答案】C

2-2-2 判断题

1．顺序存储的线性表可以按序号随机存取。

【分析】因为顺序表在内存是用地址连续的空间存储的，设 a_1 的存储地址为 $Loc(a_1)$，每个数据元素占 d 个存储地址，则第 i 个数据元素的地址为：

$$Loc(a_i)=Loc(a_1)+(i-1)*d \qquad 1<=i<=n$$

这就是说只要知道顺序表首地址和每个数据元素所占地址单元的个数就可求出第 i 个数据元素的地址来，这也是顺序表具有按数据元素的序号随机存取的特点。

【答案】正确

2．在单链表中和顺序表中插入一个元素其时间复杂度均为 O（n），因此说它们的执行时间是相等的。

【分析】大 O 记法表示时间渐近复杂度，是指一个算法中的时间耗费，往往是问题规模 n 的函数 T(n)，当 n 趋向于无穷大时，T(n)的数量级称为算法的时间渐近复杂度。虽然两种存储结构下的插入操作时间复杂度均为 O（n），但由于两者的基本操作不同，因此不能说它们的执行时间是相等的。

【答案】错误

3．静态链表即有顺序存储的优点，又有动态链表的优点。所以，它存取表中第 i 个元素的时间与 i 无关。

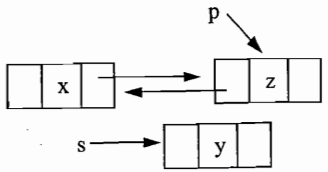
【分析】因为静态链表的存取特性与动态链表是一样的，只能顺序地找到第 i 个元素，不能随机地取到第 i 个元素，故其存取表中第 i 个元素的时间与 i 有关。

【答案】错误

2-2-3 简答题

1. 如图所示的双向链表中，欲在*p 前插入一个结点*s，请完成有关操作。

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



【解答】

只能是：s->prior->next=s; 而不能为：p->prior->next=s;

因为在上面的第二条语句中已经改变了*p 的前驱，*p 的前驱已经为*s，而不是为操作前的前驱了。在下面的语句顺序下，可有两个答案的选择。

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

读者做这种题时，最好予以图示，不易出错。

2. 已知线性表非递减有序，存储于一个向量 A[0..n-1] 中（表长为 n，设为全局量），下面算法的功能是什么？

```
void  del (DataType A[ ])
{ int i=1,j;
  while(i<=n-1)
   if (A[i]!=A[i+1]) i++;
   else
     {  for (j=(i+2);j<n;j++)  A[j-1]=A[j];
        n--;
     }
}
```

【解答】

由于向量中的元素按元素非递减有序排列，值相同的元素必为相邻的元素，因此依次比较相邻两个元素，若值相等，则删除其中一个，否则继续向后查找。故算法功能是删除向量中多余的值相同的元素。

3. 下述算法的功能是什么？

```
LinkList Demo(LinkList L)
{    //  L是无头结点的单链表
    ListNode  *Q , *P;
    if(L && L->next) {
       Q=L;L=>next;P=L;
       while (->next) P=P->next;
       P->next=Q; Q->next=NULL;
       }
       return L;
}
```

【解答】

算法的功能是把单链表的第一个结点从标头移到了链尾。返回的 L 指向原链表的第二个

结点。若原链表表示的线性表是（$a_1, a_2,\ldots, a_n$），则操作后表示的线性表为（$a_2, a_3,\ldots, a_n, a_1$）。

4. 试描述头指针、头结点、开始结点的区别，并说明头指针和头结点的作用。

【解答】

头指针：是一个指针变量，里面存放的是链表中首结点的地址，并以此来标识一个链表。如链表 H，链表 L 等，表示链表中第一个结点的地址存放在 H、L 中。

头结点：附加在第一个元素结点之前的一个结点，头指针指向头结点。当该链表表示一个非空的线性表时，头结点的指针域指向第一个元素结点，为空表时，该指针域为空。

开始结点：第一个元素结点。

头指针的作用是用来惟一标识一个单链表。

头结点的作用有两个：一是使得对空表和非空表的处理得以统一。二是使得在链表的第一个位置上的操作和在其他位置上的操作一致，无需特殊处理。

5. 在单链表、双链表和单循环链表中，若仅知道指针 p 指向某结点，而不知道头指针，能否将结点*p 从相应的链表中删除？若可以，其时间复杂度各为多少？

【解答】

单链表：若*p 有后继结点，则可以实现：将*p 的后继数据放入*p 中，再将后继删除。时间复杂度为 O(1)。若*p 无后继结点，则不可以实现。

双链表：可以实现，时间复杂度为 O(1)。

单循环链表：像单链表那样做，也可以从 p 开始，找*p 的直接前驱，然后再删除*p，时间复杂度为 O(n)。

2-2-4 算法设计题

1. 设线性表存放在向量 A[arrsize]的前 num 个分量中，且递增有序。试写一算法，将 x 插入到线性表的适当位置上，以保持线性表的有序性。并且分析算法的时间复杂度。

【分析】直接用题目中所给定的数据结构（顺序存储的思想是用物理上的相邻表示逻辑上的相邻，不一定要将向量和表示线性表长度的变量封装成一个结构体），因为是顺序存储，分配的存储空间是固定大小的，所以首先确定是否还有存储空间，若有，则根据原线性表中元素的有序性，来确定插入元素的插入位置，后面的元素为它让出位置（也可以从高下标端开始一边比较，一边移位），然后插入 x，最后修改表示表长的变量。

【算法】

```
InsertSeq(DataType  A[], int *num, DataType x)    //设 num 为表的最大下标
{ if (*num==arrsize-1)  return(0);
  else { i=*num;
       while (i>=0 && A[i]>x)                      //边找位置边移动
          { A[i+1]=A[i];
            i--;
          }
       A[i+1]=x;                                   //找到的位置是插入位的下一位
       (*num) ++;
return 1;
}
```

时间复杂度为 O(n)。

2. 假设在长度大于 1 的单循环链表中，既无头结点也无头指针。s 为指向链表中某个结

点的指针，试编写算法删除结点*s 的直接前驱结点。

【分析】利用循环单链表的特点，通过 s 指针可循环找到其前驱结点 p 及 p 的前驱结点 q，然后可删除结点*p。

【算法】

```
viod  delepre(s)
  LinkNode *s;
{ LinkNode  *p, *q;
   p=s;
   while (p->next=s)
     { q=p;
       p=p->next;
     }
     q->next=s;
     delete p;
}
```

3．已知两个单链表 A 与 B 分别表示两个集合，其元素递增排列，编写一个函数求出 A 和 B 的交集 C，要求 C 同样以元素值递增的单链表形式存储。

【分析】交集指的是两个单链表的元素值相同的结点的集合，为了操作方便，先让单链表 C 带有一个头结点，最后将其删除掉。算法中指针 p 用来指向 A 中的当前结点，指针 q 用来指向 B 中的当前结点，将其值进行比较，两者相等时，属于交集中的一个元素，两者不等时，将其较小者跳过，继续后面的比较。

【算法】

```
LinkNode *inter(A,B)
  LinkList A,B;
{ LinkNode *q, *p, *r, *s, *C;
  C=new LinkNode;
  r=C;
  p=A; q=B;
  while (p && q )
    if (p->data<q->data) p=p->next;
    else  if (p->data==q->data)
        { s=new LinkNode;
          s->data=p->data;
          r->next=s;
          r=s;
          p=p->next;
          q=q->next;
        }
    else q=q->next;
  r->next=NULL;
  C=C->next;
  return C;
}
```

4．单链表 L 是一个递减有序表，试写一个高效算法，删除表中值大于 min 且小于 max 的结点（若表中有这样的结点），同时释放被删结点的空间，这里 min 和 max 是两个给定的参数。请分析你的算法时间复杂度。

【分析】由于单链表 L 是一个递减有序表，即由大到小有序，故可从表头开始查找第一个比 max 小的结点，记住其位置，再接着向后查找第一个不大于 min 的结点，然后将它们之间的结点删除。

【算法】

```
LinkList  delete (LinkList L, int min, int max)
 //  设 L 为带头结点的循环链表
  {LinkNode   *p, *q, *s, *k;
   if ( L->next)
   { p=L->next; s=L;
     while ( p->data>=max)
          {s=p;p=p->next; }        //p 指向第一个值小于 max 的结点,s 指向其前驱
     while (p!=L&&p->data>min)     //p 继续下移
           p=p->next;              //p 指向第一个值不大于 min 的结点
      while (s->next!=p)           //删除*s 的后继至* p 的前驱之间的结点
           {k=s->next ;  s->next=k->next  ; delete k;  }
   }
}
```

前两个 while 循环和起来最多循环 n 次，第三个 while 循环最多循环 n 次，即删除 n 个结点，故算法的时间复杂度为 O(n)。

2-3　课后习题选解

1．设线性表存放在向量 A[arrsize]的前 num 个分量中，且递增有序。试写一个算法，将 x 插入到线性表的适当位置上，以保持线性表的有序性。并且分析算法的时间复杂度。

（参考典型例题解析中的算法设计题 1）

2．已知一顺序表 A，其元素值非递减有序排列，编写一个算法删除顺序表中多余的值相同的元素。

【提示】对顺序表 A，从第一个元素开始，查找其后与之值相同的所有元素，将它们删除；再对第二个元素做同样处理，依此类推。

```
typedef struct
{ datatype data[maxsize];
  int last;
}seqlist;                                    //定义顺序表类型 seqlist
void delete(seqlist *A)
{ i=0;
  while(i<A->last)                           //将第 i 个元素后与其值相同的元素删除
     {   k=i+1;
         while(k<=A->last&&A->data[i]==A->data[k])
                  k++;                       //使 k 指向第一个与 A[i]不同的元素
         n=k-i-1;                            //n 表示要删除元素的个数
         for(j=k;j<=A->last;j++)
                A->data[j-n]=A->data[j];     //删除多余元素
         A->last= A->last -n;
         i++;
     }
 return;
}
```

3．写一个算法，从给定的顺序表 A 中删除值在 x～y（x<=y）之间的所有元素，要求以较高的效率来实现。

【提示】对顺序表 A，从前向后依次判断当前元素 A->data[i]是否介于 x 和 y 之间，若是，并不立即删除，而是用 n 记录删除时应前移元素的位移量；若不是，则将 A->data[i]向前移动 n 位。n 用来记录当前已删除元素的个数。

```
typedef struct
{ int data[maxsize];
   int last;
}seqlist;
void delete(seeqlist *A,int x,int y)      //删除顺序表A中值在x~y之间的所有元素
{ i=0;n=0;
   while(i<A->last)
     { if(A->data[i]>=x&&A->data[i]<=y) n++; //若A->data[i] 介于x和y之间,n自增
       else     A->data[i-n]=A->data[i];   //否则向前移动A->data[i]
      i++;
     }
   A->last-=n;
   return;
}
```

4．线性表中有 n 个元素，每个元素是一个字符，现存于向量 R[n]中，试写一算法，使 R 中的字符按字母字符、数字字符和其他字符的顺序排列。要求利用原来的存储空间，元素移动次数最小。

【提示】对线性表进行两次扫描，第一次将所有的字母放在前面，第二次将所有的数字放在字母之后、其他字符之前。

```
int fch(char c)                  //判断c是否为字母
{ if(c>='a'&&c<='z'||c>='A'&&c<='Z') return (1);
                        else          return (0);
}
int fnum(char c)                 //判断c是否为数字
{ if(c>='0'&&c<='9')      return (1);
              else        return (0);
}
void process(char R[n])
{ low=0;high=n-1;
  while(low<high)                //将字母放在前面
      {   while(low<high&&fch(R[low])) low++;
          while(low<high&&!fch(R[high])) high--;
          if(low<high){ k=R[low];R[low]=R[high];R[high]=k;}
      }
  low=low+1;    high=n-1;
  while(low<high)                //将数字放在字母后面、其他字符前面
      {   while(low<high&&fnum(R[low])) low++;
          while(low<high&&!fnum(R[high])) high--;
          if(low<high) {k=R[low];R[low]=R[high];R[high]=k;}
      }
  return;
}
```


5．线性表用顺序存储，设计一个算法，用尽可能少的辅助存储空间将顺序表中前 m 个元素和后 n 个元素进行整体互换。即将线性表

$(a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n)$改变为：$(b_1, b_2, \ldots, b_n, a_1, a_2, \ldots, a_m)$。

【提示】比较 m 和 n 的大小，若 m<n，则将表中元素依次前移 m 次；否则将表中元素依次后移 n 次。

```
void process(seqlist *L,int m,int n)//将顺序表中前m个元素和后n个元素进行整体互换
{ if(m<=n)
  for(i=1;i<=m;i++)
      {   x=L->data[0];
          for(k=1;k<=L->last;k++)
                  L->data[k-1]=L->data[k];
          L->data[L->last]=x;
      }
  else for(i=1;i<=n;i++)
      {   x=L->data[L->last];
          for(k=L->last-1;k>=0;k--)
          L->data[k+1]=L->data[k];
          L->data[0]=x;
      }
  return;
}
```

6．已知带头结点的单链表 L 中的结点是按整数值递增排列的，试写一个算法，将值为 x 的结点插入到表 L 中，使得 L 仍然有序。并且分析算法的时间复杂度。

【提示】先在单链表中顺序比较，找到插入位置，再生成结点，插入。

```
typedef struct node
{ int data;
  struct node *next;
}lnode,*linklist;
linklist insert(linklist L,int x)     //将值为x的结点插入到有序的单链表L中
{ p=L;
  while(p->next&&x>p->next->data)
  p=p->next;                          //找插入位置
  s=(lnode *)malloc(sizeof(lnode));
  s->data=x;                          //申请、填装结点
  s->next=p->next;
  p->next=s;                          //插入到链表中去
  return(L);
}
```

7．假设有两个已排序的单链表 A 和 B，编写一个算法将它们合并成一个链表 C 而不改变其排序性。

【提示】同时从单链表 A 和单链表 B 的第一个结点开始进行比较，将值较小的结点插入到链表 C 的表尾。

```
typedef struct node
{ datatype data;
  struct node *next;
}lnode,*linklist;          //定义单链表结点类型、指针类型
```

```
linklist combine(linklist A,linklist B)
{ C=A;rc=C;
  pa=A->next;          //pa 指向 A 的第一个结点
  pb=B->next;          //pb 指向 B 的第一个结点
  free(B);             //释放 B 的头结点
  while(pa&&pb)
      if(pa->data<pb->data)   {rc->next=pa;rc=pa;pa=pa->next;}
      else                    {rc->next=pb;rc=pb;pb=pb->next;}
                              //将 pa、pb 所指向结点中值较小的一个插入到链表 C 的表尾
  if(pa)  rc->next=pa;
      else        rc->next=pb;     //将链表 A 或 B 中剩余的部分链接到链表 C 的表尾
  return(C);
}
```

8. 假设在长度大于 1 的单循环链表中，既无头结点也无头指针。s 为指向链表中某个结点的指针，试编写算法，删除结点*s 的直接前驱结点。

（参考典型例题解析中的算法设计题 2）

9. 已知两个单链表 A 与 B 分别表示两个集合，其元素递增排列，编写一个函数求出 A 和 B 的交集 C，要求 C 同样以元素值递增的单链表形式存储。

（参考典型例题解析中的算法设计题 3）

10. 设有一个双向链表，每个结点中除有 prior、data 和 next 域外，还有一个访问频度 freq 域，在链表被起用之前，该域的值初始化为零。每当在链表进行一次 Locata(L,x)运算后，令值为 x 的结点中的 freq 域增 1，并调整表中结点的次序，使其按访问频度的递减序列排列，以便使频繁访问的结点总是靠近表头。试写一个满足上述要求的 Locata(L,x)算法。

【提示】在定位操作的同时，需要调整链表中结点的次序：每次进行定位操作后，要查看所查找结点的 freq 域，将其同前面结点的 freq 域进行比较，同时进行结点次序的调整。

```
typedef struct dnode
{ datatype data;
  int freq;
  struct dnode *prior,*next;
}dnode,*dlinklist;
dlinklist locate(dilnklist L,datatype x)
{ p=L->next;
  while(p&&p->data!=x) p=p->next;              //查找值为 x 的结点，使 p 指向它
  if(!p) return(NULL);                         //若查找失败，返回空指针
  p->freq++;                                   //修改 p 的 freq 域
  while(p->prior!=L&&p->prior->freq<p->freq) //调整结点的次序
  {   k=p->prior->data;
      p->prior->data=p->data;
      p->data=k;
      k=p->prior->freq;
      p->prior->freq=p->freq;
      p->freq=k;
      p=p->prior;
  }
  return(p);                                   //返回找到的结点的地址
}
```


第 3 章　栈和队列

本章通过对栈和队列这两种特殊线性结构的介绍，要求掌握栈和队列的操作特点、不同存储结构上的操作实现，以及其各自适用的应用问题。

重点提示：

- 栈的操作特点与存储实现（顺序栈、链栈）
- 队列的操作特点与存储实现（循环队列、链队列）
- 栈和队列适用的应用问题

3-1　重点难点指导

3-1-1　相关术语

1. 有关栈的名词

（1）栈

要点：限制仅在表的一端进行插入、删除运算的线性表。

操作原则：后进先出（或先进后出）。

（2）栈顶

要点：栈中进行插入、删除的一端。

（3）栈底

要点：与栈中插入、删除一端相对的另一端。

（4）顺序栈

要点：采用顺序存储结构实现的栈。

（5）链栈

要点：采用链式存储结构实现的栈。

（6）上溢

要点：栈满时再做进栈运算所产生的空间溢出。

（7）下溢

要点：栈空时再做退栈运算所产生的无元素可退的情况。

2. 有关队列的名词

（1）队列

要点：仅允许在表的一端进行插入，而在表的另一端进行删除的受限的线性表。

操作原则：先进先出（或后进后出）。

（2）顺序队列

要点：采用顺序存储结构实现的队列。

（3）假上溢

要点：在顺序队列中由于队的头尾指针只增不减，导致队尾指针已到队列空间的上界不能再做入队操作，而随着出队操作的进行，队首指针增加，使队列空间尚有空闲单元，这种

队列已满的假现象被称为假上溢。

（4）循环队列

要点：将顺序队列的向量空间视为一个首尾相接的圆环进行运算的队列；克服了假上溢现象，充分利用了向量空间。

（5）链队列

要点：采用链式存储结构实现的队列。

3-1-2 栈

1. 栈的基本运算

（1）Init_Stack(S)构造一个空栈。

（2）Empty_Stack(S)判断栈空。若 S 为空栈，则返回 True，否则返回 False。

（3）Full_Stack(S)判断栈满。若 S 为满栈，则返回 True，否则返回 False。

（4）Push_Stack(S,x)进栈。若栈 S 不满，则将元素 x 插入 S 的栈顶。

（5）Pop_Stack(S)退栈。若栈 S 非空，则将 S 的栈顶元素删除。

（6）Top_Stack(S)取栈顶元素。若栈 S 非空，则返回栈顶元素，不改变栈顶的位置。

2. 顺序栈

（1）特点

① 数据元素的存储与顺序表相似。

② 仅在 top 端进行插入和删除。

（2）存储表示

```
#define StackSize 100                  //设置栈空间的大小
typedef char DataType;                 //设置栈元素类型
typedef struct{
     DataType data[StackSize];         //data 为栈向量
     int  top;                         //top 为栈顶的位置
}SeqStack;
```

由此可看出顺序栈的存储是将顺序表存储定义中的 length 域改名为 top，并将其作为栈顶位置的指针。

（3）存储示意

以定义 SeqStack *S 为例，在栈 S 中已依次压入了 A、B、C 三个字符，其存储示意如图 3-1 所示。

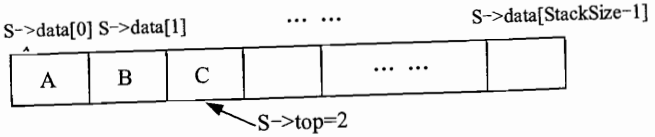


图 3-1 顺序栈的存储示意图

（4）基本操作实现要点

① 判满条件

```
S->top==StackSize-1
```

② 判空条件


```
S->top==-1
```

③ 初始栈

```
S->top=-1;
```

④ 入栈

```
if(栈不满) {S->top++;   S->data[S->top]=x;}
```

⑤ 出栈

```
if(栈不空) {x= S->data[S->top];  S->top--;}
```

⑥ 取栈顶元素

```
if(栈不空) {x= S->data[S->top];  return(x);}
```

3. 链栈

（1）特点

① 数据元素的存储与不带头结点的单链表相似。

② 用指针 top 指向单链表的第一个结点。

③ 插入和删除在 top 端进行。

（2）存储表示

```
typedef struct stacknode {              //定义栈的结点结构
        DataType  data;                 //定义栈元素 data 的类型
        struct stacknode  *next;
}StackNode;
typedef struct {
    StackNode  *top;                    //栈顶指针
}LinkStack;
```

（3）存储示意

以定义 LinkStack *S 为例，在栈 S 中已依次压入了 A、B、C 三个字符，其存储示意如图 3-2 所示。

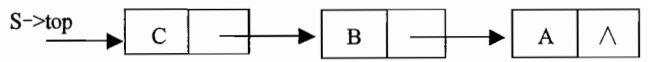


图 3-2 链栈的存储示意图

图中 S->top->data='C', S->top->next 为第二个结点的地址。

（4）基本操作实现要点

① 判空条件

```
S->top==NULL
```

② 初始栈

```
S->top=NULL;
```

③ 入栈

```
p=(StackNode*)malloc(sizeof(StackNode));
p->data=x;  p->next=S->top;  S->top=p;
```

④ 出栈

```
if(栈不空)
  {p=S->top;   S->top=p->next;   x=p->data;   free(p);}
```

⑤ 取栈顶元素

```
if(栈不空)   {x=S->top->data;   return(x);}
```

3-1-3 队列

1. 栈的基本运算

（1）Init_Queue(Q)构造一个空队列。

（2）Empty_Queue(Q)判断队空。若 Q 为空队列，则返回 True，否则返回 False。

（3）Full_Queue(Q)判断队列满。若 Q 为满队列，则返回 True，否则返回 False。

（4）In_Queue(Q,x)入队列。若队列 Q 不满，则将元素 x 插入到 Q 的队尾。

（5）Out_Queue(Q)出队列。若队列 Q 非空，则将队列 Q 的队头元素删除。

（6）Front_Queue(Q)取队头元素。若队列 Q 非空，则返回队头元素，不改变队头的位置。

（7）Length_Queue(Q)取队长。返回队列 Q 的长度。

说明：一般定义 True 为 1，False 为 0。

2. 循环队列

（1）特点

① 数据元素的存储与顺序表相似。

② 将顺序表的向量空间视为一个首尾相接的圆环（即当位置增 1 时，通过对整个空间大小取余运算来实现对空间首尾相接的使用）。

③ 在线性圆环上设置队尾 rear 端，队头 front 端。

④ 限制在队尾 rear 端进行插入，在队头 front 端进行删除。

（2）存储表示

```
#define QueueSize 100                    //定义队列空间大小
typedef char DataType;                   //定义队列元素类型
typedef struct {
     DataType data[QueueSize];           //定义队列的元素空间
     int  front;                         //定义队头指针
     int  rear;                          //定义队尾指针
     int  count;                         //定义队中元素总数
}CirQueue;
```

（3）存储示意

以定义 CirQueue *Q 为例，在队列*Q 中已依次入队了 A、B、C 三个字符，其存储示意如图 3-3 所示。

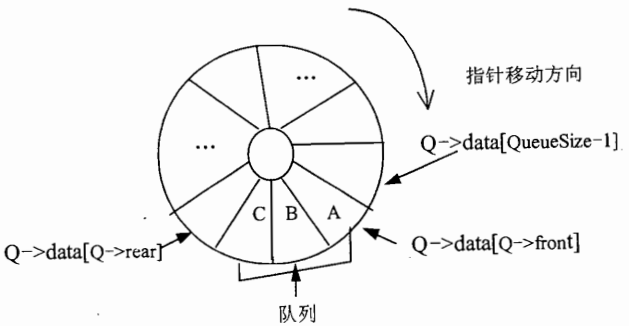


图 3-3 循环队列的存储示意

图中 Q->front=0，Q->rear=3，Q->count=3，队头元素为 Q->data[Q->front]='A'。

（4）基本操作实现要点

① 初始队列

```
Q->count=0;  Q->front=0;  Q->rear=0;
```

② 判满条件

```
Q->count==QueueSize
```

③ 判空条件

```
Q->count==0
```

④ 取队长

```
return(Q->count);
```

⑤ 入队列

```
if(队列不满)
  {Q->data[Q->rear]=x;  Q->count++;
   Q->rear=(Q->rear+1) % QueueSize;}
```

⑥ 出队列

```
if(队列不空)
  {x=Q->data[Q->front];  Q->count--;
   Q->front=(Q->front+1) % QueueSize;}
```

⑦ 取队头元素

```
if(队列不空)
  {x=Q->data[Q->front];  return(x);}
```

说明：若存储队列不设 count 域，上述基本操作的实现略有变化，其实现要点如下：

① 初始队列

```
Q->front=0;Q->rear=0;
```

② 判空条件

```
Q->front==Q->rear=0
```

③ 判满条件

```
Q->rear+1%QueueSize= =Q->front
```

其实，此时 Q->rear 所指单元为可用，但为了与判空条件相区别，舍弃一个单元不用，即这种循环队列的实现方式队列最多存放 QueueSize-1 个元素。

④ 取队长

```
return((Q->rear-Q->front+QueueSize)%QueueSize);
```

⑤ 入队列

```
if(队列不满)
  {Q->data[Q->rear]=x;
   Q->rear=(Q->rear+1) % QueueSize;}
```

⑥ 出队列

```
if(队列不空)
  {x=Q->data[Q->front];
   Q->front=(Q->front+1) % QueueSize;}
```

⑦ 取对头元素

```
if(队列不空)
  {x=Q->data[Q->front];  return(x);}
```

3. 链队列

（1）特点

要点：

① 数据元素的存储与带头结点的单链表相似。

② 头结点设为含有 front 和 rear 两个指针域的结构头结点。

③ front 指向单链表的第一个元素，被视为队头。

④ rear 指向单链表的最后一个元素，被视为队尾。

⑤ 限制在队尾 rear 端进行插入，在队头 front 端进行删除。

（2）存储表示

```
typedef  struct queuenode {                    //链队列中的结点类型
  DataType data;                               //定义队中元素类型
  Struct queuenode  * next;
}QueueNode;
typedef struct{                                //定义头结点类型
  QueueNode *front;                            //队头指针
  QueueNode *rear;                             //队尾指针
}LinkQueue;
```

（3）存储示意

以定义 LinkQueue *Q 为例，在队列 Q 中已依次入队了 A、B、C 三个字符，其存储示意如图 3-4 所示。

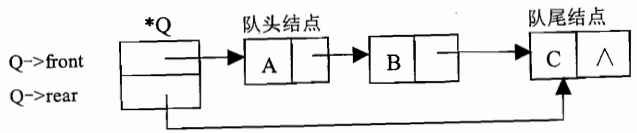


图 3-4 链式队列的存储示意

图中队头结点 Q->front->data='A'，队尾结点 Q->rear->data='C'。

（4）基本操作实现要点

① 判空条件

```
Q->front==NULL
```

② 初始队列

```
Q->front=NULL;   Q->rear=NULL;
```

③ 入队列

```
p=(QueueNode*)malloc(sizeof(QueueNode));
p->data=x;
p->next=NULL;
if(队列不空)  Q->rear->next=p;
else Q->front=p;
Q->rear=p;
```

④ 出队列


```
if(队列不空)
  {p=Q->front;
   if (p==Q->rear) Q->rear=NULL;
   Q->front=p->next;  x=p->data;   free(p);
  }
```

⑤ 取队头元素

```
if(队列不空)
  {x=Q->front->data;  return(x);}
```

⑥ 取队长

```
i=0; p=Q->front;
while(p){i++;p=p->next;}
return(i);
```

3-1-4 栈的应用

1. 适合的应用

对数据的处理顺序符合后遇到先处理，即后进先出的原则。

2. 举例

字串的逆序输出、括号匹配、迷宫问题求解、前缀中缀和后缀表达式求值、二叉树的先序、中序、后序遍历、递归调用和进制转换等。

3-1-5 队列的应用

1. 应用

对数据的处理顺序符合先遇到先处理，即先进先出的原则。

2. 举例

迷宫问题求解、银行事务处理、模拟进程调度、二叉树的层次遍历、图的广度优先遍历和拓扑排序等。

3-2 典型例题解析

3-2-1 选择题

1. 设有一顺序栈 S，元素 s_1、s_2、s_3、s_4、s_5、s_6 依次进栈，如果 6 个元素出栈的顺序是 s_2、s_4、s_3、s_6、s_5、s_1，则栈的容量至少应该是（　　）。

A. 2　　B. 3　　C. 5　　D. 6

【分析】s_1、s_2 进栈后，此时栈中有 2 个元素，接着 s_2 出栈，栈中尚有 1 个元素 s_1；s_3、s_4 进栈后，此时栈中有 3 个元素，接着 s_4、s_3 出栈，栈中尚有 1 个元素 s_1；s_5、s_6 进栈后，此时栈中有 3 个元素，接着 s_6、s_5 出栈，栈中尚有 1 个元素 s_1；s_1 出栈后，此时栈为空栈。由此可知，栈的容量至少应该是 3。

【答案】B

2. 设已将元素 a_1、a_2、a_3 依次入栈，元素 a_4 正等待进栈。那么下列 4 个序列中不可能出现的出栈序列是（　　）。

A. a_3 a_1 a_4 a_2　　B. a_3 a_2 a_4 a_1　　C. a_3 a_4 a_2 a_1　　D. a_4 a_3 a_2 a_1

【分析】由于 a_1、a_2、a_3 已进栈，此时，栈顶元素为 a_3，不管 a_4 何时进栈，出栈后，

a_1、a_2、a_3的相对位置一定是不变的，这就是 a_3 一定在前，a_2 一定居中，a_1 一定在后。比较上述 4 个答案，只有 A 中的 a_1 出现在 a_2 的前面，这显然是错误的。

【答案】A

3．向一个栈顶指针为 Top 的链栈中插入一个 s 所指结点时，其操作步骤为（　）。

A．Top->next=s;　　B．s->next=Top->next;Top->next=s;

C．s->next=Top;Top=s;　　D．s->next=Top;Top=Top->next;

【分析】本操作是在链栈上的进栈操作。操作顺序应该是先插入结点，再改变栈顶指针。

【答案】C

4．从栈顶指针为 top 的链栈中删除一个结点，并将被删除结点的值保存到 x 中，其操作步骤为（　）。

A．x=top->data;top=top->next;　　B．top=top->next;x=top->data;

C．x=top;top=top->next;　　D．x=top->data;

【分析】本操作是在链栈上进行出栈操作，操作顺序应该是先保存被删除结点的值，然后再改变栈顶指针的值。

【答案】A

5．链栈与顺序栈相比，有一个较明显的优点是（　）。

A．通常不会出现栈满的情况　　B．通常不会出现栈空的情况

C．插入操作更加方便　　D．删除操作更加方便

【分析】不管是链栈还是顺序栈，其插入、删除操作都是在栈顶进行的，都比较方便，所以不可能选 C，D。对链栈来说，当栈中没有元素而又要执行出栈操作时，就会出现栈空现象，故 B 也是不正确的。只要内存足够大，链栈上就不会出现栈满现象。而对顺序栈来讲，由于其大小是事先确定好的，因此可能会出现栈满现象。

【答案】A

6．设数组 A[0…m]作为循环队列 sq 的存储空间，front 为队头指针，rear 为队尾指针，则执行入队操作时修改指针的语句是（　）。

A．sq.front=(sq.front+1)%m　　B．sq.front=(sq.front+1)%(m+1)

C．sq.rear=(sq.rear+1)%m　　D．sq.rear=(sq.rear+1)%(m+1)

【分析】首先，执行的入队操作是在队尾进行的，故 A，B 肯定是不对的。其次，由于存放 sq 的数组为 A[0…m]，共有 m+1 个单元，%运算中的模应为 m+1。

【答案】D

7．在一个链队列中，若 f、r 分别为队首、队尾指针，则插入 s 所指结点的操作为（　）。

A．f->next=c;f=s;　　B．r->next=s;r=s;

C．s->next=r;r=s;　　D．s->next=f;f=s;

【分析】插入操作总是在队尾进行，而且应该先插入结点，再改变队尾指针的值。

【答案】B

3-2-2　填空题

1．在未设定 count 域的具有 n 个单元的循环队列中，队满时共有（　）个元素。

【分析】在循环队列中，为了能正确区分队满、队空两个判断条件，尾指针 rear 所指单元总是空的。因此最多（队满时）能存放 n-1 个元素。

【答案】n-1

2．对于循环向量的循环队列，求队列长度的公式为（　）。

【分析】可以根据队尾与队首的差值求得队列的长度，但需要注意是循环队列，应将差值加上队列空间大小，再对空间大小进行取余运算得到。

【答案】(sq->rear－sq->front+maxsize) % maxsize

3．栈的逻辑特点是（　）。队列的逻辑特点是（　）。二者的共同点是只允许在它们的（　）处插入和删除数据元素，区别是（　）。

【分析】由于只能在栈顶处执行插入、删除操作，使得数据元素的进栈顺序恰好与出栈顺序相反，所以栈的逻辑特点是先进后出（或后进先出）。而对于队列来说，插入、删除操作必须在队列的两端进行，数元素的进队顺序是一致的，所以队列的逻辑特点是先进先出（或后进后出）。两者的共同特点是所有操作只能在端点处进行。

【答案】先进后出（或后进先出）；先进先出（或后进后出）；端点；栈是在同一端插入和删除，队列是在一端插入而在另一端删除。

3-2-3　简答题

1．链栈中为何不设置头结点？

【解答】

因为链栈是运算受限制的单链表，其插入和删除操作都限制在表头位置上进行，由于只能在链表头部进行操作，故链栈不需要设置头结点。

2．循环队列的优点是什么？如何判别它的空和满？

【解答】

循环队列的优点是克服假上溢现象。

设有循环队列 sq

以(sq->rear+1)%maxsize=sq->front 或 sq->count=maxsize 来判别队满。

以 sq->rear=sq->front 来判别队空。

3．设长度为 n 的链队列用单循环链表表示，若只设头指针，则入队出队操作的时间为多少？若只设尾指针呢？

【解答】

只设头指针：O(n)，只设尾指针：O(1)。

只设头指针的链队结构如图 3-5 所示。

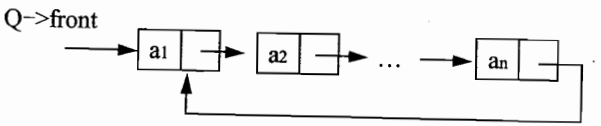


图 3-5　只设头指针的链队存储结构示意

因为可通过 Q->front 直接得到队头，而找队尾则需经过 n 个点，故出队的时间复杂度为 O(1)，入队的为 O(n)。

只设尾指针的链队结构如图 3-6 所示。

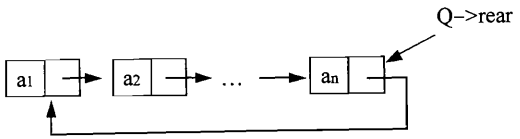


图 3-6　只设尾指针的链队存储结构示意

因为可通过 Q->rear 直接得到队尾，通过 Q->rear->next 找到队头，故入队、出队操作的时间复杂度均为 O (1)。

3-2-4　算法设计题

1. 回文是指正读和反读均相同的字符序列，如“abba”和“abdba”均是回文，但“good”不是回文。试写一个算法判定给定的字符向量是否为回文。

方法一

【分析】使用栈。将字符串的前一半入栈，再依次出栈，与后一半进行比较，若有不等则不是回文，若依次相等，则是回文。

【算法】

```
int  Hw1(char  a[ ],int n)
{SeqStack  S;
 int  i,j;
 j=1;i=0;
 InitStack(& S);                    //初始化栈 s
 while (i<n/2)                      //将字符串的前一半入栈 s
  {push (&S,a[i]);
   i=i+1;
  }
 if(n %2!=0)i=i+1;                  //若 n 为奇数 i 加 1，越过中间的一个数
 while (i<n && j==1)
   if (a[1]==pop(&S))
      i=i+1;
   else  {j=0; break;}
return(j);                          //返回值 j=0，则不是回文；若 j=1，则是回文
}
```

方法二

【分析】不用栈，而是用两变量如 i 和 j 分别指向字符串的首和尾，将 i 和 j 所指字符进行比较，若有不等，则不是回文；若直到 i≥j 以前 a[i]与 a[j]均相等，则是回文。

【算法】

```
int  Hw2 (char  a[],int n)
{int  i,j;
  i=0;j=n-1;
  while (i<j)
    if(a[i]==a[j])
      {i++;j--;}
    else  break;
    if(i<j)  return (0);
    else    return(1);
}
```


2. 一个双向栈 S 是在同一向量空间内实现的两个栈，它们的栈底分别设在向量空间的两端。试为此双向栈 S 设计初始化 InitStack(S)、入栈 Push(S,i,x)和出栈 Pop(S,i)等算法，其中 i 为 0 或-1，用以指示栈号。

【分析】双向栈是两个栈共享一个向量空间，栈 0 的底下标为 0，栈 1 的底下标为 m-1，初始时栈 0 的顶为-1，栈 1 的顶为 m；当栈 0 有元素进栈时，让栈 0 的栈顶指针加 1；退栈时减 1。当栈 1 有元素进栈时，让栈 1 的栈顶指针减 1，退栈时加 1。

【算法】

```
define    m    100                  //设定双向栈的向量空间大小
typedef   int   datatype              //定义栈元素类型
typedey   struct {
      datatype    v[m];               //定义栈空间
      int   top0,top1;                //top0 为 0 栈栈顶位置，top1 为 1 栈栈顶位置
}dstack;
void push (dstack  *s, int  i, datatype x, )
if (i!=0 || i!=1) printf("error!");
  else  if(i==1)
      {if(s->top1-1==s->top0)  printf("overflow!");                    //栈已满
      else {s->top1;
          s->v[s->top1]=x;}
      }
     else {if (s->top0+1== s->top1)  printf("overflow!");               //栈已满
        else  {s->top0++;
             s->v[s->top0]=x;
             }
}
datatype pop(dstack *s,int  i)
{if(i!=0 ||i!=1)  printf("error!");
  else  if (i==1)
       {if(s->top1== m)  printf("underflow!");                         //栈 1 空
        else {s->top1++;
            return(s->v[s->top1-1]);
           }
       }
       else { if(s->top0== -1)  printf("underflow!");                  //栈 0 空
          else {s->top0--;
              return(s->v[s->top0+1]);
             }
          }
}
void Initstack(dstack *s)
{s->top0= -1;
 s->top1=m;
}
```

3. 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点（注意不设头指针），试编写相应的置空队、判队空、入队和出队等算法。

【分析】带头结点的循环链表表示的队列如图 3-7 所示，仅有队尾指针 rear，但可通过 rear->next 找到头结点，再通过头结点找到队头，即 rear->next->next。

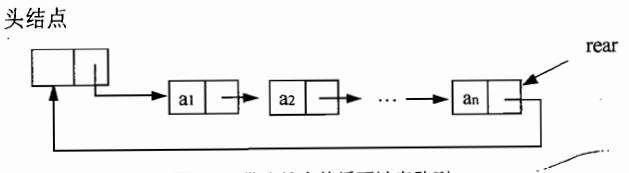


图 3-7　带头结点的循环链表队列

【算法】

① 置空队

```
LinkList SetNull()
{LinkList   rear;
 rear=malloc(sizeof(ListNode));
 rear->nex =rear;
return (rear);
}
```

② 判队空

```
int Empty(Linklist  rear)
{if(rear->next== rear)  return (1); //若队列为空返回 1
 else  return(0);                   //否则返回 0
}
```

③ 入队

```
LinkList  ENQUEUE(LinkList  rear, datatype  x)
{LinkList  p;
p=malloc(sizeof(linklist));
p->data=x;
p->next=rear->next;                 //将 p 插入到 rear->next 之后
rear->next=p;
rear=p;
return  rear;                       //返回新的队尾指针
}
```

④ 出队

```
Linklist  DEQUEUE(Linklist  raer,Datatype  *x)
 { if(rear->next= =rear)            //若队空，则输出队空信息
     printf("EMPTY");
  else {q=rear->next;  p=q->next; } //否则 q 指向头结点，p 指向队头
  if(p= =rear)  rear=q;             //若队中仅有一个元素，则将 rear 指向头结点
  q->next=p->next;                  //将 p 所指结点出队
  *x=p->data;
  free(p);                          //将对头结点的值赋给形参*x
  return(rear);                     //返回队尾指针
}
```

4．假设循环队列中只设 rear 和 quelen 来分别指示队尾元素的位置和队中元素的个数，试给出判断此循环队列的队满条件，并写出相应的入队和出队的算法，要求出队时需返回队头元素。

【分析】设用长度为 m 的向量存放队中元素，则可用 quelen 是否等于 m 作为循环队列的队满条件。对于入队操作，较简单，只要在 rear 处插入一个元素即可。而对于出队操作，

由于队头元素的序号未知，所以需先计算出队头元素序号，计算公式为：

```
front=(sq->rear - sq->quelen+1+m)%m
```

采用的存储结构定义为：

```
typedef   struct {
   int   rear,quelen;
   Datatype  data[m];
}Cirqueue;
```

【解答】

① 队满条件

```
sq->quelen==m
```

② 入队

```
void Enqueue(Cirqueue  *sq,datatype  x)
{if(sq->quelen==m)
  printf("Queue  overflow");
else { sq->rear=(sq->rear+1)%m;
       sq->data[sq->rear]=x;
       sq->quelen=sq->quelen+1;
     }
}
```

③ 出队

```
Datatype  Dequeue(Cirqueue  *sq)
{ int front;
  if(sq->quelen== 0)
       printf("underflow");
  else {front=(sq->rear-sq->quelen+1 +m)%m;
       sq->quelen=sq->quelen-1;
       return(sq->data[front]);
       }
}
```

3-3　课后习题选解

1．栈和队列数据结构各有什么特点，什么情况下用到栈，什么情况下用到队列？

栈和队列都是操作受限的线性表，栈的运算规则是“后进先出”，队列的运算规则是“先进先出”。栈的应用如数制转换、递归算法的实现等，队列的应用如树的层次遍历等。

2．设有编号为 1，2，3，4 的 4 辆车，顺序进入一个栈式结构的站台，试写出这 4 辆车开出车站的所有可能的顺序（每辆车可能入站，可能不入站，时间也可能不等）。

1234，1243，1324，1342，1432，2134，2143，2314，2341，2431，3214，3241，3421，4321

3．回文是指正读和反读均相同的字符序列，如“abba”和“abdba”均是回文，但“good”不是回文。试写一个算法判定给定的字符向量是否为回文。

（参考典型例题解析中的算法设计题 1）

4．设以数组 se[m]存放循环队列的元素，同时设变量 rear 和 front 分别作为队头队尾指针，且队头指针指向队头前一个位置，写出这样设计的循环队列入队出队的算法。

①
```
int in_queue(datatype se[m],int rear,int front,datatype e)
{if((rear+1)%m==front) return(-1);          //若队列满，返回-1
   rear=(rear+1)%m;
   se[rear]=e;
   return(1);
   }
```
②
```
int out_queue(datatype se[m],int rear,int front,datatype *e)
{   if(rear==front) return(-1);             //若队列空，返回-1
   front=(front+1)%m;
   *e=se[front];
      return(1);
   }
```

5. 假设以数组 se[m]存放循环队列的元素，同时设变量 rear 和 num 分别作为队尾指针和队中元素个数记录，试给出判别此循环队列的队满条件，并写出相应入队和出队的算法。

①
```
int in_queue(datatype se[m],int rear,int num,datatype e)
{  if(num==m) return(-1);
   rear=(rear+1)%m;
   se[rear]=e;
   num++;
   return(1);
}
```
②
```
int out_queue(datatype se[m],int rear,int num,datatype *e)
{  if(num==0) return(-1);
   *e=se[(rear-num+1)%m];
   num--;
   return(1);
}
```

6. 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点（注意不设头指针），试编写相应的置空队、判队空、入队和出队等算法。

（参考典型例题解析中的算法设计题 3）

7. 设计一个算法判别一个算术表达式的圆括号是否正确配对。

【提示】利用栈的特点，进行判断。

```
int process()
{ init_stack(s);
  cin>>c;
  while(c!='@')
  { if(c=='(') push_stack(s,c);              //当前字符是左括号，则入栈
    if(c==')') {if(!empty_stack(s))
                pop_stack(s,&a);
           else return(-1);
        }//当前字符是右括号时，判断栈的状态：若栈空，则返回-1，否则出栈
    cin>>c;
  }
  if(empty_stack(s))  return(1);
  else  return(-1);        //若栈空，说明表达式中括号配对，返回 1；否则，返回-1
}
```


8. 写一个算法，借助栈将一个单链表置逆。

【提示】利用栈后进先出的特点，将单链表中的结点从链表头开始依次压栈，然后再依次出栈，采用尾插法重新生成单链表。

```
linklist process(linklist a)
{ p=A->next;
  r=A;
  A->next=NULL;
  init_stack(s);                          //初始化栈
  while(p)                                //将链表中结点依次压栈
  { push_stack(s,p);
    p=p->next;
  }
  while(!empty_stack(s))                  //将栈中元素依次出栈
  { pop_stack(s,&q);
    r->next=q;
    r=q;
  }
  r->next=NULL;
  return(A);
}
```

9. 一个双向栈 S 是在同一向量空间内实现的两个栈，它们的栈底分别设在向量空间的两端。试为此双向栈 S 设计初始化 InitStack(S)、入栈 Push(S,i,x)和出栈 Pop(S,i)等算法，其中 i 为 0 或-1，用以指示栈号。

（参考典型例题解析中的算法设计题 2）

第4章　串

串是每个数据仅由一个字符组成的一种特殊的线性表，它在计算机文字编辑、词法分析等方面有着广泛的应用。本章通过对串的概念、串的存储方法和串上基本运算实现的介绍，要求掌握串这种特殊线性结构的存储和基本算法实现。

重点提示：

- 串与线性表的关系
- 串的两种存储表示（顺序存储、链式存储）
- 串的基本运算
- 串的模式匹配算法

4-1　重点难点指导

4-1-1　相关术语

1．串

要点：零个或多个字符组成的有限序列。

2．串的长度

要点：串中所包含字符的个数。

3．空串

要点：长度为零的串。

4．空白串

要点：仅由一个或多个空格组成的串。

5．子串

要点：串中任意个连续字符组成的子序列。

6．主串

要点：包含子串的串。

4-1-2　串的基本运算

1．StrLength(s)求串长。即求串 s 的长度。

2．StrAssign(s，t)串复制。即将串 t 复制到串 s 中。

3．StrConcat(s，t)串连接。即将串 t 复制到串 s 的末尾，构成一个新串。

4．StrCom(s，t)串比较。即比较串 s 和串 t 的大小，若 s>t，返回正数；若 s=t，返回 0；若 s<t，返回负数。

5．Substr(S，i，j)求子串。即从串 S 的第 i 个字符起连续取 j 个字符构成的子串。

6．StrInsert(S，i，T)串插入。即将串 T 插入到串 S 的第 i 个字符之后。

7．StrDelete(S，i，j)串删除。即从串 S 的第 i 个字符起连续删除 j 个字符。

8．StrIndex(S，T)串匹配，也称串定位。即当串 S 中存在与串 T 相等的子串时，则返回第一个子串 T 在串 S 中第一个字符的序号，否则返回 0。通常称 T 为模式串，S 为主串。

9．StrRep(S，T，R)串替换。即用子串 R 替换串 S 中的所有子串 T。

4-1-3　串的存储结构

1．顺序存储

（1）特点

① 数据元素的存储与顺序表相似。

② 每个数据元素的类型是字符型。

③ 也可直接采用一维字符数组存放。

（2）存储表示

① 定义顺序串的静态存储分配类型

方式 1：

```
#define MaxStrSize  256        //定义字符串的最大长度
typedef struct {
     char ch[MaxStrSize];
     int length;
}SeqString;
```

特点：便于获取串的长度；字符串最大长度固定，对字符串的存储可能产生冗余或溢出现象。

方式 2：

```
typedef char SeqString[MaxStrSize];
```

特点：简单易实现；字符串最大长度固定，对字符串的存储可能产生冗余或溢出现象。

② 定义顺序串的动态存储分配类型

方式 1：

```
typedef struct {
     char *ch;
     int length;
}HString;
```

特点：便于获取串的长度；字符串长度可依据具体情况获取，不会出现冗余或溢出现象。

方式 2：

```
typedef char *String;
```

特点：简单易实现；字符串长度可依据具体情况获取，不会出现冗余或溢出现象。

（3）存储示意

以静态方式 1 定义 SeqString　*S 为例，在*S 中存放字符串“abcde”，其存储示意如图 4-1 所示。

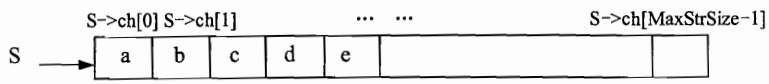


图 4-1　串的顺序存储示意

其中 S->length=5。

2. 链式存储

（1）特点

① 数据元素的存储与不带头结点的单链表相似。

② 每个结点的数据域均为字符型。

（2）存储表示

① 简单链式存储

```
typedef struct node{
        char   data;
        struct node  *next;
}LinkStrNode,*LinkString;
```

特点：便于字符串的插入、删除和连接等操作；存储空间冗余较大。

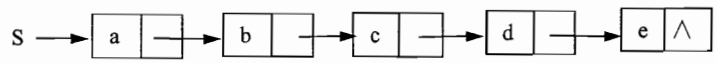
② 块链式存储

```
#define NodeSize 8
typedef struct node {
      char data[NodeSize];
      struct node *next;
}LinkSteNode,*LinkString;
```

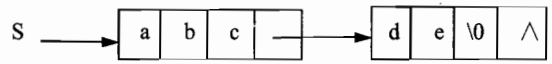
特点：结点内用一维数组存储字符串，结点间用指针连接；减少了字符串存储时存储空间的冗余；给字符串中的插入、删除等操作带来不便。

（3）存储示意

以定义 LinkString *S 为例，在*S 中存放字符串“abcde”，其存储示意如图 4-2 所示。（a）图为简单链式存储的示意，（b）图为块链式存储的示意（NodeSize=3）。



（a）简单链式存储



（b）块链式存储

图 4-2 串的链式存储示意

其中，图（a）所示的存储 S->data='a'，S->next 为第二个结点的地址；图（b）的存储 S->data[0]='a'，S->data[1]='b'，S->data[2]='c'，S->next 为第二个结点的地址。

3. 串的堆存储

（1）特点

① 将串名与串值分开存储，串名存储在索引表中，串值存储在堆空间中。

② 串名通过索引表对应到堆空间中的串值。

③ 索引表的形式可以有多种。

④ 可以根据每个字符串的长度动态地在堆空间中申请相应大小的存储空间。

（2）堆空间的存储表示

```
堆空间为: char store[SMAX+1];          //SMAX 为能存储足够多串的内存空间大小
自由区指针: int free;                  //free 为堆空间中未使用区域的首位置
```

（3）索引表的存储表示

① 带串长度的索引表

```
typedef struct {
        char   name[MAXNAME];          //串名
        int    length;                 //串长
        char   *stradr;                //起始地址
}LNode;
```

特点：便于取得串的长度。

② 带末尾指针的索引表

```
typedef struct {
        char   name[MAXNAME];          //串名
        char   stradr, enadr;          //起始地址，末尾地址
    }ENode;
```

特点：便于取得串的尾部。

③ 带特征位的索引表

```
typedef struct {
        char   name[MAXNAME];          //串名
        int    tag;                    //特征位
        union{                         //起始地址或串值
           char   *stradr;             //起始地址
           char   value[4];            //串值
}TNode;
```

特点：对于短字符串的存取既节省空间又可快速读取。

4-1-4 字符串匹配算法实现

1. 用串的其他运算构造字符串匹配的运算 Index[S,T].

【算法思想】利用 Length 运算求出 S、T 两串的长度，然后再用 Substr 运算从 S 的第一个字符开始依次取得与 T 串长度相同的子串，调用 Equal 运算，将该子串与 T 串比较，若相等，则匹配成功，返回 T 串在 S 串中第一个字符的位置；否则匹配失败，返回 0。

【算法】

```
int  index(string s,t)
{i=1;
 slen=length(s);tlen=length(t);          // 求串 s、t 的长度
 while((i+tlen-1<=slen)&&(! Equal(Substr(s,i,tlen),t)))
   i+ +;                                 // 往后匹配
 if(i+tlen<=slen) retum(i);              // 匹配成功返回 i 的值
 else  return(0);                        // 匹配不成功返回 0
 }
```

2. 顺序串上的实现

【算法思想】设 T 为主串，P 为模式串，i 为主串中当前匹配子串的首位置，k 为主串中当前匹配字符的位置，j 为模式串中当前匹配字符的位置。比较 T.ch[k]和 P.ch[j]的值。

① 若 T.ch[k]=P.ch[j]，则 k 加 1，j 加 1，均移向 T 和 P 的下一个字符，继续比较。若到 P 串结束前一直相等，则匹配成功。

② 若 T.ch[k]≠P.ch[j]，则说明主串中当前 i 开始的子串与 P 不匹配，i 加 1 取下一个子串，置 k 等于 i，j 等于 0，重新将模式串与主串中的子串比较。

③ 当主串中无法取得与模式串长度相等的子串时（即 i>n-m 时），则匹配失败。

【算法】
```
int NaiveStrMatch(SeqString T,SeqString P)
{// 找模式串 P 在目标 T 中首次出现的位置，成功时返回第 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)                       //即 T[i..i+m-1]=P[0..m-1]
            return i;                  //i 为有效位移，否则考查下一个位移
     } // endfor
     return -1;                        //找不到有效位移，匹配失败
} // NaiveStrMatch
```

3. 链串上的实现

【算法思想】由于是链式串，所以对元素的存储表示与顺序串不同，在链式串中不是用下标指示元素，而是用指针指示元素。在算法中用到的指针变量 shift、t 和 p 分别相当于顺序串匹配算法中的 i、k 和 j 的作用。

【算法】
```
LinkStrNode *LinkStrMath(LinkString  T,LinkString  P)
{ //在链串上求模式串 P 在目标 T 中首次出现的位置
    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的字符
              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;
          }
}
if(p= =NULL)
    return  shift;                     //匹配成功
else
    return  NULL;                      //匹配失败
}
```

4-2 典型例题解析

4-2-1 字符串的基本运算题

1. 设有 A=' '，B='mule'，C='old'，D='my'，计算下列运算的结果（注："A+B" 是 Concat 的简写）。

（a）A+B　（b）B+A　（c）D+C+B　（d）SubStr(B,3,2)
（e）SubStr(C,1,0)　（f）StrLength(A)　（g）StrLength(D)　（h）StrIndex(B,D)
（i）StrIndex(C,"d")　（j）StrInsert(D,2,C)　（k）StrInsert (B,I,A)　（l）StrDelete(B,2,2)
（m）StrDelete(B,2,0)　（n）StrRep(C,2,2, "k")

【解答】

（a）A+B="#mule";　（b）B+A="mule#"
（c）D+C+B="myoldmule"　（d）SubStr(B,3,2)= "le"
（e）SubStr(C,1,0)= " "　（f）StrLength(A)=1
（g）StrLength(D)=2　（h）StrIndex(B,D)=0
（i）StrIndex(C, "d")=3　（j）StrInsert(D,2,C)= "myldy"
（k）StrInsert (B,I,A)="m#ule"　（l）StrDelete(B,2,2)="me"
（m）StrDelete(B,2,0)="mule"　（n）StrRep(C,2,2, "k")="ok"

2. 假设有如下的串说明：

char sl[30]="Stocktom,CA",s2[30]= "March 5,1999",s3[30];

（1）调用函数 strCmp(s1,s2)的返回值是什么？

（2）调用函数 strCmp(&s1[5], "ton")的返回值是什么？

（3）调用函数 Strlength(StrConcat(s1,s2))的返回值是什么？

【解答】

（1）正数，表示 s_1 大于 s_2。

（2）负数，表示 "tom，CA" 小于 "ton"。

（3）23，即为 s_1 与 s_2 连接后的长度。

4-2-2 算法设计题

1. 在顺序串上实现串的判等运算 Equal（S，T）。

【算法】
```
int Equal (HString s,t)
{ // 串 S、T 为顺序存储结构
  if(s.length !=t.length)          //判断串 S、T 长度是否相等
     return(0);                    //两串不相同
  else{i=0;
     while((i<s.length)&&(s.ch[i]==t.ch[i]))
        i+ +;                      //跳过前面相同元素
      return(i>=s.length);         //i>s.length 表示两串相同，返回 1，否则返回 0
     }
}
```

2. 在链串上实现串的判等运算 Equal（S，T）。

【算法】
```
int Equal (LinkStrlist s,t)
{ // 串 s、t 为链式存储结构
   p1=s;p2=t;
   while((p1 !=NULL)&&(p2 !=NULL)&&(p1->ch==p2->ch))
     {p1=p1->next;p2=p2->next;}      //p1、p2 元素值相等，p1、p2 后移
```

```
        return((p1==NULL)&&(p2==NULL))
        //若 p1、p2 同时为空指针时表示两串相同返回真值，否则返回假值
}
```

3. 若 X 和 Y 是用结点大小为 1 的单链表表示的串，设计一个算法找出 X 中第一个不在 Y 中出现的字符。

【分析】将 X 串的结点值依次与 Y 串中的结点值比较，若找到第一个不在 Y 中出现的结点，则查找成功，返回该值，否则用一个特殊符号“$”表示 X 中的结点均在 Y 中出现过。

【算法】

```
char locate (LinkString X,LinkString Y)
{ // 串用无头结点的单链表存储
  p=X;                                    //指向 X 串第一个结点
  while(p!=NULL)
   {q=Y;                                  //在 Y 表中查找是否有 p->data 元素
    while((q!=NULL)&&(q->data!=p->data))
      q= q->next;                         //子串后移
    if(q==NULL)  return(p->data);         //查找成功
   p =p -> next                           //查找 X 表中下一个元素
  }
  return('$');
}
```

4. 若 S 和 T 是用结点大小为 1 的单链表存储的两个串，设计一个算法将串 S 中首次与串 T 匹配的子串逆置。

【分析】本算法的实现分 3 部。

① 链串中的匹配；② 匹配成功后将子串逆置；③ 将逆置后的子串连到原串中。

其中，串匹配前面已介绍过，这里主要说明串逆置的过程。若 t 是所找到的子串，则 prior 指向子串 t 的第一个结点的直接前趋，p 指针指向子串 t 最后一个结点的直接后继，如图 4-3 所示。为了对串 t 逆置，从子串 t 第一个结点开始设置 3 个指针 q、r、u，再依次将 r 所指向结点的 next 域指向它的前一个结点，即将 r->next=q，再将 q、r、u 右移而 q=r;r=u;u=r->next 直到 p=r 时逆置完毕，逆置后的结果如图 4-4 所示。而逆置后 tl 结点的直接后继是*p，*prior 的直接后继是 t_n。

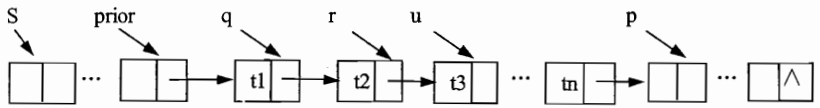


图 4-3　串找子串 t 的示意图

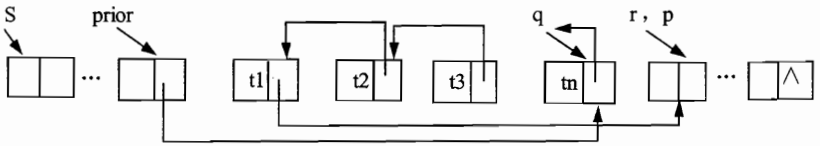


图 4-4　子串 t 逆置的示意图

【算法】

```
int  invert substr(LinkString  s,t)
{ // s、t 均带头结点
   prior=s;p=prior->next;                 //prior 为 p 的前驱
   tl=t->next;
   if((p==NULL)||(tl==NULL)
     {printf("不匹配"); return(0); }      //s、t 两串至少有一个为空串，返回
   while((p !=NULL)&&(tl !=NULL))
     if(p->data==tl->data)
       {p=p->next;tl=tl->next;}           //继续往后比较
     else{ prior=prior->next;             //匹配不成功，向后移
           p=prior->next;
           tl=t->next;
         }
     if(tl !=NULL){printf("不匹配");return(0);}
     else{ q=prior->next;r=q->next;       //匹配成功，开始逆置
           while(r!=p)
             {u=r->next;r->next=q;
              q=r;r=u;
             }
           prior->next->next=p;           //将逆置后的子串连到原串中
           prior->next=q;
         }
}
```

4-3　课后习题选解

1. 利用 C 的库函数 strlen、strcpy 和 strcat 写一个算法 void StrInsert(char *S,char *T,int i)，将串 T 插入到 S 的第 i 个位置上。若 i 大于 S 的长度，则插入不执行。

```
void strinsert(char *s,char *t, int i)
{   if(i>strlen(s)) return;
       strcopy(s+i,r);
       s[i]='\0';
       strcat(s,t);
       strcat(s,r);
}
```

2. 利用 C 的库函数 strlen、strcpy（或 strncpy）写一个算法 void StrDelete(char *S,int i,int m)，删除串 S 中从位置 i 开始连续的 m 个字符。若 i≥strlen(S)，则没有字符被删除；若 i+m≥strlen(S)，则 S 中从位置 i 开始直至末尾的字符均被删去。

```
void strdelete(char *s, int m)
{   if(i>=strlen(s)) return;
       if(i+m>=strlen(s)) s[i]='\0';
          else {  for(k=i+m;s[k]!='\0';k++)
                    s[k-m]=s[k];
                  s[k]='\0';
                }
       return;
}
```

3．采用顺序结构存储串，编写一个函数，求串 s 和串 t 的一个最长的公共子串。

【提示】需要采用三重循环来实现。

```
typedef struct
{ char *data;
  int len;
}string;
void maxsubstr(char *s,char *t,char *r)          //求 S 和 T 的最长公共子串
{int i,j,k,num,maxnum=0,index=0;
  i=0;
  while(i<s->len)
  { j=0;
    while(j<t->len)
      {if(s->data[i]==t->data[j])
         {   num=1;
           for(k=1;s->data[i+k]==t->data[j+k];k++)
             num=num+1;
           if(num>maxnum)
           { index=i;
             maxnum=num;
           }
           j+=num; i=0;
         }
       else  j++;
    }
    i++;
    }
  for(i=index,j=0;i<index+maxnum;i++,j++)
        r->data[j]=s->data[i];
  return;
}
```

4．采用顺序存储结构存储串，编写一个函数计算一个子串在一个字符串中出现的次数，如果该子串不出现则为 0。

```
int count(char *s,char *t)
{   int i=0,c=0,j;
    while(i<strlen(s))
    {j=strindex_KMP(s,t,i,next);
      if(j!=0) c++;
      i=i+strlen(t);
    }
    return(c);
}
```


第 5 章　多维数组与广义表

前几章所介绍的线性结构中的数据元素都是非结构的原子类型，即其元素的值是不可再分解的，而本章所讨论的多维数组和广义表是对线性表的推广，其特点是数据元素仍可被视为一个表。要求熟悉多维数组的逻辑结构、存储结构，广义表的逻辑结构、表示形式，以及矩阵的压缩存储的有关内容。

重点提示：

- 多维数组的存储方式和存取特点
- 特殊矩阵的存储
- 稀疏矩阵的存储
- 广义表的表示形式

5-1　重点难点指导

5-1-1　相关术语

1．特殊矩阵

要点：矩阵中非零元素或零元素的分布有一定规律的矩阵。

2．对称矩阵

要点：一种特殊矩阵；n 阶方阵的元素满足性质：$a_{ij}=a_{ji}$　（$0\leqslant i，j\leqslant n-1$）。

3．三角矩阵

要点：以主对角线划分，有上三角矩阵和下三角矩阵两种；主对角线以下，不包括主对角线中的元素，均为常数 c，称为上三角矩阵；主对角线以上，不包括主对角线中的元素，均为常数 c，称为下三角矩阵。

4．对角矩阵

要点：非零元素集中在以主对角线为中心的带状区域中，也称带状矩阵。

5．稀疏矩阵

要点：矩阵中非零元素的个数远小于矩阵元素总数的矩阵。

6．三元组表

要点：是稀疏矩阵的一种存储结构；将稀疏矩阵的非零元素的三元组（行、列和值）按行优先的顺序排列；得到结点均是三元组的线性表。

7．广义表

要点：是线性表的推广；是 n 个元素 $a_1,a_2,\cdots,a_n$ 的有限序列；其中 a_i 或者是原子或者是广义表；通常记为 $LS=(a_1,a_2,\cdots,a_n)$，LS 为广义表的名字。

5-1-2　多维数组

1．对 n 维数组逻辑结构的理解

n 维数组可视为由 n-1 维数组为元素的线性结构。

举例：一个 m 行 n 列的二维数组可视为由 m 个一维数组为元素组成的线性结构，其中每个一维数组又由 n 个单元素组成。

$$Amn=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mnv} \end{pmatrix}=\begin{pmatrix} A_1 \\ A_2 \\ \cdots \\ A_m \end{pmatrix} \quad 其中A_i=[a_{i0}, a_{i1}, \cdots, a_{in}]$$

2. 数组的顺序存储方式

（1）行优先顺序——将数组元素按行向量排列，即第 i+1 行紧接在第 i 行后面。如 Amn 的存储序列为：

a_{11} a_{12} $\cdots$ a_{1n}	a_{20} a_{21} $\cdots$ a_{2n}	$\cdots$	a_{m1} a_{m2} $\cdots$ a_{mn}
第 1 行	第 2 行	…	第 m 行

（2）列优先顺序——将数组元素按列向量排列，即第 i+1 列紧接在第 i 列后面，如 Amm 的存储序列为：

a_{11} a_{21} $\cdots$ a_{m1}	a_{12} a_{22} $\cdots$ a_{m2}	$\cdots$	a_{1n} a_{2n} $\cdots$ a_{mn}
第 1 列	第 2 列	…	第 n 列

3. 数组元素的存储地址（以行优先顺序计算，假设每个元素为 d 个单元）

（1）二维数组中 a_{ij} 的地址

$LOC(a_{ij})= LOC(a_{11})+[(i-1)*n+j-1]*d$

在 C 语言中，因数组下标从 0 起，故

$LOC(a_{ij})=LOC(a_{00})+(i*n+j)*d$

（2）三维数组中 a_{ijk} 的地址（A_{mnp}）

$LOC(a_{ijk})= LOC(a_{111})+ [(i-1)*n*p+(j-1)*p+(k-1)]*d$

在 C 语言中，因数组下标从 0 起，故

$LOC(a_{ijk})=LOC(a_{000})+(i*n*p+j*p+k)*d$

（3）对于一般的二维数组 $A[c_1 \cdots d_1, c_2 \cdots d_2]$

$LOC(a_{ij})=LOC(a_{c1c2})+[(i-c_1)*(d_2-c_2+1)+j-c_2]*d$

5-1-3 特殊矩阵

矩阵在存储中常用二维数组，而对于一些特殊的矩阵，可将其按一定规律压缩存储在一维向量中，如 sa[ ]中。

1. 对称矩阵

（1）特点：在 n 阶方阵 A 中有 $a_{ij}=a_{ji}$ （$0 \leqslant i, j \leqslant n-1$）。

（2）存储：按行优先的存储顺序为：$a_{00}, a_{10}, a_{11}, a_{21}, a_{22}, \cdots a_{n-1\,0}, a_{n-1\,1} \cdots a_{n-1\,n-1}$

（3）元素的二维下标(i, j)与其在一维向量中的位置 k 的对应关系：

```
I=max(i, j),  J=min(i, j)
k=I*(I+1)/2+J
LOC(aij)=LOC(sa[k])
LOC=LOC(sa[0])+k*d=LOC(sa[0]+[I*(I+1)/2+J]*d
```


2. 三角矩阵

（1）特点：n 阶方阵 A 中，上三角或下三角（不包括主对角线）元素均为常数 c。

（2）存储：按行优先存放。

上三角矩阵的存储序列为：

LOC(sa[0])　　LOC(sa[n*(n+1)/2])

a_{00} a_{01} $\cdots$ a_{0n-1}	a_{11} a_{12} $\cdots$ a_{1n-1}	$\cdots$	a_{n-1n-1}	c
第 0 行	第 1 行	…	第 n-1 行	常数

下三角矩阵的存储序列为：

LOC(sa[0])　　LOC(sa[n*(n+1)/2])

a_{00}	a_{11} a_{12}	$\cdots$	a_{n-10} a_{n-11} $\cdots$ a_{n-1n-1}	c
第 0 行	第 1 行	…	第 n-1 行	常数

（3）元素的二维下标（i,j）与其在一维向量中的位置 k 的对应关系：

上三角矩阵：

$$k=\begin{cases} i*(2n-i+1)/2+j-i & i \leqslant j \\ n*(n+1)/2 & i>j \end{cases}$$ （常数 c 的位置）

下三角矩阵：

$$k=\begin{cases} i*(i+1)/2+j & i \leqslant j \\ n*(n+1)/2 & i>j \end{cases}$$ （常数 c 的位置）

即有：$LOC(a_{ij})=LOC(sa[k])$

3. 对角矩阵

（1）特点：除对角线和其相邻两侧的若干条对角线上的元素之外，其余元素皆为零。

（2）存储：以 3 条带的带状矩阵为例，即主对角线两侧各有一条带的元素，见图 5-1。其存储特点为第一行和最后一行为两个元素，其他各行为 3 个元素。

$$\begin{pmatrix} a_{00} & a_{01} & & & & \\ a_{10} & a_{11} & a_{12} & & & \\ & a_{21} & a_{22} & a_{23} & & \\ & & \cdots & \cdots & \cdots & \\ & & & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ & & & & a_{n-1n-2} & a_{n-1n-1} \end{pmatrix}$$

图 5-1　n×n 的带状矩阵

（3）按行优先的存储序列为：

LOC(sa[0])　　…　　LOC(sa[3n-3])

a_{00} a_{01}	a_{10} a_{11} a_{12}	a_{21} a_{22} a_{23}	$\cdots$	a_{n-2n-3} a_{n-2n-2} a_{n-2n-1}	a_{n-1n-2} a_{n-1n-1}
第 0 行	第 1 行	第 2 行	…	第 n-2 行	第 n-1 行

元素的二维下标(i,j)与其在一维向量中的位置 k 的对应关系：

k=3i−1+j−i+1=2i+j

即有：$LOC(a_{ij})=LOC(sa[k])$

说明：由于对向量起始下标的定义不同，对特殊矩阵压缩存储位置的计算表达式也会有所不同，读者应掌握其分析计算的方法，根据具体情况推出相应计算表达式。

5-1-4 稀疏矩阵

（1）特点：矩阵中非零元素的个数远小于矩阵元素的总数。

（2）存储：

① 三元组表

特点：按行优先（或列优先）；仅存非零元素；每个非零元素存贮 3 个信息（行、列、值）；还需存上矩阵的行数、列数及非零元素个数。

存储类型定义：

```
# define Smax  1024                  //定义三元组表的大小
typedef int DataType;                //定义三元组表的元素值的类型
typedef struct {                     //三元组表元素结构
   int  i, j;                        //非零元素的行、列号
   DataType  v ;                     //非零元素的值
} SPNode;
typedef  struct {                    //三元组表类型
   SPNode  data[SMax];               //三元组表空间
   int  mu,nu,tu;                    //矩阵的行数、列数及非零元个数
}SPMatrix;
```

例：如图 5-2（a）所示的稀疏矩阵 A 存储的三元组表如图 5-2（b）所示。

其中定义三元组表类型的变量 SPMatrix *a，用*a 存储矩阵 A。

$$A_{4\times5}=\begin{pmatrix}0&0&0&0&8\\1&0&3&0&0\\0&2&0&0&0\\6&0&0&0&0\end{pmatrix}$$

	i	j	v
a->data [0]	0	4	8
1	1	0	1
2	1	2	3
3	2	1	2
4	3	0	6
	…	…	…
MaxSize−1			

a->RowTab[0]	0
1	1
2	3
3	4

(a)稀疏矩阵 A　　(b)A 的三元组表*a 中的 a->data　　(c)A 的带行表*a 中的 a->RowTab

图 5-2　稀疏矩阵 A 与其三元组表和行表的示意

② 带行表的三元组表

特点：在按行优先存储的三元表中，加入一行表记录，稀疏矩阵每行的非零元素在三元组表的起始位置。

目的：便于矩阵运算。

存储类型定义：

```
#difine MaxRow  100
typedef  struct  {
   SPNode  data [MaxSize]
   int RowTab[MaxRow];
   int  mu,nu,tu;
}RSPMatrix;
```

例：如图 5-2（a）所示的稀疏矩阵 A 存储的三元组表如图 5-2（b）所示；行表如图 5.2（c）所示。其中定义*a 为带行表的三元组表类型的变量，用*a 存储矩阵 A。

5-1-5 广义表

（1）特点

① n 个元素的有限序列。

② 每个元素或者是原子或者是一个广义表。

（2）有关概念：

① 广义表的长度

要点：若广义表 $LS=(a_1,a_2,\cdots,a_n)$，则其长度为 n。

② 表头

要点：广义表的第一个元素。

若广义表 $LS=(a_1,a_2,\cdots,a_n)$，则 a_1 是其表头。

注意表头是原表中的一个元素。

③ 表尾

要点：广义表除第一个元素之外其余元素组成的子表。

若广义表 $LS=(a_1,a_2,\cdots,a_n)$，则（a_2，…，a_n）是其表尾。

注意表尾仍是表。

④ 纯表

要点：与树对应的广义表；限制了表中成份的共享与递归。

⑤ 再入表

要点：允许结点共享的广义表。

⑥ 递归表

要点：允许递归的广义表。

⑦ 表的深度

要点：表展开后所含括号的层数。

（3）表示

① 括号表示法

格式：表名（表元素列表）

举例：

i）线性表　L(a,b)

说明：表名为 L，表的两个元素 a 和 b 为同一类型，是广义表退化为线性表。

表 L 长度为 2，深度为 1。

ii）纯表　A(x,L(a,b))

说明：表名为 A，表内有两个元素，一个是原子 x，另一个是表 L。

表 A 长度为 2，深度为 2。

iii）再入表　C(A(x,L(a,b)),B(A(x,L(a,b)),y))

说明：表名为 C，表内有两个表元素，分别是表 A 和表 B。

其中表 A 又是表 B 中的第一个元素，即被表 B 所共享。

表 C 长度为 2，深度为 4。

iv）递归表　D(a,D(a,D(…)))

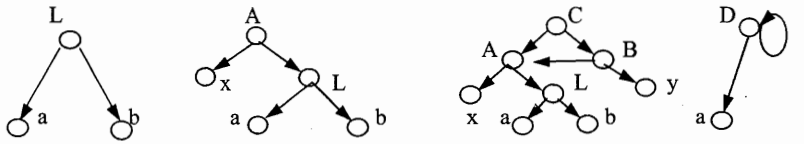
说明：表名为 D，表内有两个元素，一个是原子，另一个是表 D 自身。

表 D 长度为 2，深度为∞。

② 图形表示法

方法：结点表示表或是原子；若表示有 n 个元素的表结点，该表结点就有 n 条出边指向这 n 个元素；原子结点只有入边没有出边。

举例：用括号表示法列举的 4 个例子的图形表示如图 5-3 所示。



（a）L=(a,b)　（b）A=(x,L)　（c）C=(A,B)　（d）D=(a,D)

图 5-3　广义表的图形表示

（4）几种广义表之间的关系

递归表 ⊃ 再入表 ⊃ 纯表 ⊃ 线性表

（5）两个基本运算

① head(LS) 取表 LS 的表头。

② tail(LS) 取表 LS 的表尾。

5-2　典型例题解析

5-2-1　填空题

1．数组 M 中每个元素的长度是 3 个字节，行下标 i 从 0 到 7，列下标 j 从 0 到 9，从首地址 EA 开始连续存放在存储器中。若按行优先方式存放，元素 M[7][5]的起始地址为（　）；若按列优先方式存放，元素 M[7][5]的起始地址为（　）。

【分析】

按行优先方式存储时，M[7][5]的前面已经存放了 75（7×10＋5=75）个元素，它们共占用了 75×3=225 个字节，所以 M[7][5]的起始地址为 EA＋225。

按列优先方式存储时，M[7][5]的前面已经存放了 47（5×8+7=47）个元素，它们共占用了 47×3=141 个字节，所以 M[8][5]的起始地址为 EA＋141。

【答案】EA+22；EA+141

2．二维数组 M 的成员是 6 个字符（每个字符占一个存储单元）组成的串，行下标 i 的范围从 0 到 8，列下标 j 的范围从 0 到 9，则存放 M 至少需要（　）个字节；M 的第 8 列和第 5 行共占（　）个字节；若 M 按行优先方式存储，元素 M[8][6]的起始地址与当 M 按列优先方式存储时的（　）元素的起始地址一致。

【分析】

按题意二维数组 M 共有 9 行（行号 0～8），每行有 10 列（列号 0～9），合计有 90 个元素，每个元素占 6 个字节，所以存放 M 至少需要 90×6＝540 个字节。

M 的第 8 列上若有 9 个元素，第 5 行上共有 10 个元素，合计有 19 个元素，但是其中有两个元素是重复的（M[5][8]），故实际有 18 个元素，共需占用 18×6＝108 个字节。

元素 M[8][6]在数组中位于第 8 行的第 6 列，其前有 8 行 6 列，当按行优先方式存储时，在其前面已经存储了 8×10＋6＝86 个元素，因此 M[8][5]是顺序存储的第 87 个元素。而当按列优先方式存储时，第 87 个元素应该是位于第 9 列的第 5 行上（前 9 列共有 81 个元素，第 9 列有 6 个元素），即元素 M[5][9]。

【答案】540；108；M[5][9]

5-2-2　简答题

1．设有上三角矩阵（a_{ij}）n×n，将其上三角元素逐行存于数组 B[1：m]中（m 充分大），使得 B[k]＝a_{ij}，且 k＝$f_1(i)+f_2(j)+C$。试推导出函数 f_1、f_2 和常数 C（要求 f_1 和 f_2 中不含常数项）。

【解答】

对上三角形矩阵：

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n-1} & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n-1} & a_{2n} \\ 0 & 0 & a_{33} & \cdots & a_{3n-1} & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-1n-1} & a_{n-1n} \\ 0 & 0 & 0 & \cdots & 0 & a_{nn} \end{pmatrix}$$

若 i<=j，$k=LOC(a_{ij})=\underbrace{n+(n-1)+(n-2)+\cdots(n-i+1)}_{\text{前 i-1 行}}+\underbrace{(j-i)}_{\text{第 i 行}}$

$=i*(n-(i+1)/2)+j-n$

k=f1(i)+f2(j)+c=i*(n-(i+1)/2)+j-n　　(当 i<=j)

即 f1(i)=i*(n-(i+1)/2)，f2(j)=j，c=-n

2．设有三对角矩阵 $A_{n\times n}$，将其 3 条对角线上的元素逐行存储到向量 B[0…3n-3]中，使得 B[k]= a_{ij}，求：

（1）用 i、j 表示 k 的下标交换公式；

（2）用 k 表示 i、j 的下标交换公式。

【解答】

设三对角矩阵如前面图 5-1 所示。

因为三对角矩阵中除首尾行各需保存两个元素外各行均需保存 3 个元素，故需保存 3n-2 个元素。

（1）第 i 行前有 3i-1 个元素，a_{ij} 在第 i 行上前面有 j-i+1 个非零元素，故在 a_{ij} 前共有

3i-1+j-i+1 个非零元素，所以 k=2i+j。

（2）对三角对称矩阵，若知道元素在一维中的下标为 k，求所在的行数应考虑第一行有两个元素，其余各行都有 3 个元素，所以所在行数为 i=(k+1)/3（注意，这里的除法为整除；k 加 1 的原因是 C 语言下标从 0 起）。由于 i=(k+1) /3、j=k-2i，所以 j-k-2(k+1)/3。

3．设二维数组 A_{5*6} 的每个元素占 4 个字节，已知 LOC（a_{00}）=1000，A 共占多少个字节？A 的终端结点 a_{45} 的起始地位为何？按行和按列优先存储时，a_{25} 的起始地址分别为何？

【解答】

因为 4×5×6=120，所以二维数组 $A_{5\times6}$ 共占 120 个字节；

$LOC(a_{45})$=1000+120-4=1116；

按行优先时 $LOC(a_{25})$=1000+4×(2×6+5)=1068；

按列优先时 $LOC(a_{25})$=1000+4×(5×5+2)=1108。

4．设广义表 L=(( ),( )),试问 head(L)、tail(L)、L 的长度、深度各为多少？

【解答】

head(L)=( )；Tail(L)=(( ))；L 的长度为 2，深度为 2。

5-2-3 算法设计题

1．当三对角矩阵采用 5-2-2 简答题中题 2 所述的压缩存储时，写一算法求三对角矩阵在这种压缩存储表示下的转置矩阵。

【分析】

设将压缩放在 B[]中的三对角矩阵的转置矩阵存放在 C[]中。方法是依次取出 B []中元素，设其下标为 kb，换算出 kb 对应的 i、j，再计算出 j 行 i 列对应的 kc，并使 C[kc]=B[kb]。

【算法】

```
void  TransMatrix(DataType  *B,  DataType  *C, int n)
  {int  kb, kc;
   int i, j;
   C[0]=B[0];
   for (kb=2;kb<=n;kb++)
    {i=(kb+1) /3;
     j=kb-2*i;
     kc=2*j+i;
     C[kc]=B[kb];
     }
  }
```

2．当具有相同行值和列值的稀疏矩阵 A 和 B 均以三元组表作为存储结构时，试写出矩阵相加算法，其结果存放在三元组表 C 中。

【分析】按照行优先的顺序同时扫描稀疏矩阵 A 和 B，若当前 A 与 B 中数的行值、列值均相同，则将两数相加，若相加结果不为 0，则将其放入结果三元组表 C 中；若当前 A 中数的行值小，或其行值与当前 B 中数的行值相等，而列值小于当前 B 中数的列值，则将当前 A 中数放入结果三元组表 C 中；否则将当前 B 中数放入结果三元组表 C 中。当稀疏矩阵 A 和 B 有一个先扫描结束，则将另一个的剩余数据依次放入 C 中。

【算法】

```
define SMAX  1024
typedef   struct
{ int i;
  int j;
  datatype v;
}SPNode;                              //三元组类型
typedef   struct
{ SPNode data[SMAX];                  //0 号单元未用
  int mu;
  int nu;
  int tu;
}SPMatrix;                            //三元组表
SPMatrix  *Matrix_add (SPMatrix *A,SPMatrix  *B)
{ SPMatrix  *C;
  C->mu = A->mu;
  C->nu = A->nu;
  C->tu = 0;
  pa=1 ; pb =1 ; pc=1;
  while (pa<=A->tu && pb<=B->tu)
  {if ((A->data[pa].i==B->data[pb].i)&&(A->data[pa].j==B->data[pb].j))
// 行号、列号相等时
    { C->data[pc].i=A->data[pc].i;
      C->data[pc].j=A->data[pc].j;
      C->data[pc].v=A->data[pa].v + B->data[pb].v;
      C->tu++ ;pc++ ; pa++ ; pb++ ;
    }
  else  // 行号、列号不等时
    if ((A->data[pa].i < B->data[pb].i)||
        (A->data[pa].i==B->data[pb].i&&A->data[pa].j< B->data[pb].j))
        { C->data[pc].i = A->data[pa].i;
          C->data[pc].j = A->data[pa].j;
          C->data[pc].v = A->data[pa].v;
          C->tu++ ; pc++ ;pa++;
        }
  else { C->data[pc].i = B->data[pb].i;
         C->data[pc].j = B->data[pb].j;
         C->data[pc].v = B->data[pb].v;
         C->tu++;   pc++; pb++;
       }
  }
  while( pa<=A->tu )              //A 中有剩余元素时
  { C->data[pc].i = A->data[pa].i ;
    C->data[pc].j = A->data[pa].j ;
    C->data[pc].v = A->data[pa].v ;
    pc++ ;pa++ ;
  }
  while ( pb<=B->tu )             //B 中有剩余元素时
  { C->data[pc].i = B->data[pb].i;
    C->data[pc].j = B->data[pb].j;
    C->data[pc].v = B->data[pb].v;
```

```
        pc++ ; pb++ ;
      }
      return(C);
    }
```

5-3 课后习题选解

1．假设按行优先存储整数数组 A[9][3][5][8]时，第一个元素的字节地址是 100，每个整数占 4 个字节。问下列元素的存储地址是什么：（1）a0000 （2）a1111 （3）a3125 （4）a8247

①100; ②776; ③1784; ④4416

2．设有三对角矩阵 An×n，将其 3 条对角线上的元素存于数组 B[3][n]中，使得元素 B[u][v]=a_{ij}，试推导出从(i,j)到(u,v)的下标变换公式。

```
u=j-i+1
v=j-1
```

3．假设一个准对角矩阵：

$$\begin{pmatrix} a_{11} & a_{12} & & & & & & & \\ a_{21} & a_{22} & & & & & & & \\ & & a_{33} & a_{34} & & & & & \\ & & a_{43} & a_{44} & & & & & \\ & & & & \ddots & & & & \\ & & & & & a_{ij} & & & \\ & & & & & & \ddots & a_{2m-1,2m-1} & a_{2m-1,2m} \\ & & & & & & & a_{2m,2m-1} & a_{2m,2m} \end{pmatrix}$$

按以下方式存储于一维数组 B[4m]中：

0	1	2	3	4	5	6		k			4m-2	4m-1
a_{11}	a_{12}	a_{21}	a_{22}	a_{33}	a_{34}	a_{43}	…	a_{ij}	…	$a_{2m-1,2m}$	$a_{2m,2m-1}$	$a_{2m,2m}$

写出由一对下标(i,j)求 k 的转换公式。

k=i+j-i%2-1

4．现有如下的稀疏矩阵 A（如图所示），要求画出以下各种表示方法。

（1）三元组表示法。

（2）十字链表法。

$$\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & -15 \\ 0 & 13 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

① 三元组表示法

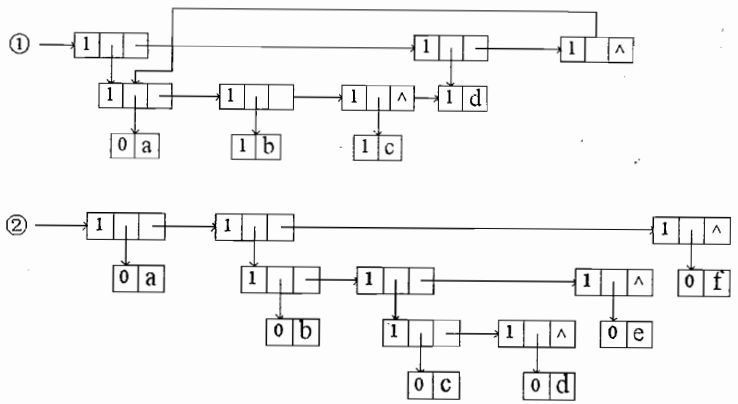
	i	j	v
1	1	4	22
2	1	6	-15
3	2	2	13
4	2	3	3
5	3	4	-6
6	5	1	91
7	6	3	28

② 略

5．画出下列广义表的存储结构示意图。

（1）A=((a,b,c),d,(a,b,c))

（2）B=(a,(b,(c,d),e),f)



6．对于二维数组 A[m][n]，其中 m<=80，n<=80，先读入 m、n，然后读该数组的全部元素，对如下 3 种情况分别编写相应算法：

（1）求数组 A 靠边元素之和。

（2）求从 A[0][0]开始的互不相邻的各元素之和。

（3）当 m=n 时，分别求两条对角线的元素之和，否则打印 m!=n 的信息。

```
main( )
  {int A[80][80];
   int m,n;
   int s1,s2,s3;
   cin>>m; cin>>n;
   for(i=1;i<m;i++)
        for(j=1;j<n;j++)
            cin>>A[i][j];
   s1=fun1(A,m,n);
   s2= fun2(A,m,n);
```

```
    if(m==n)
       s3= fun3(A,m,n);
    else cout<<"m!=n"<<endl;
}
int fun1(int A[80][80],int m,int n)
{int i,s=0;
  for(i=0; i<n; i++)  s+=A[0][i]+A[m-1][i];
  for(i=1; i<m-1;i++) s+=A[i][0]+A[i][n-1];
  renturn(s);
  }
int fun2(int A[80][80],int m,int n)
{int i,j,s=0;
  for(i=0; i<m; i++)
    for(j=i%2; j<n;j+=2)  s+= A[i][j];
  renturn(s);
}
int fun3(int A[80][80],int m)
{int i,s=0;
  for(i=0; i<m; i++)  s+=A[i][i]+A[i][m-i-1];
  if(i%2) s-=A[m/2+1][m/2+1];
  renturn(s);
}
```

7. 有数组 A[4][4]，把 1 到 16 个整数分别按顺序放入 A[0][0]...A[0][3]，A[1][0]...A[1][3]，A[2][0]...A[2][3]，A[3][0]...A[3][3]中，编写一个算法获取数据并求出两条对角线元素的乘积。

```
int  mul (int  A[4][4])
{ int K=1,s=1;
  for (i=0; i<4 ;i++)
    for (j=0 ;j<4 ;j++)
       { A[i][j] =K ; K++; }
  for (i=0;  i<4 ;i++)
    {s* = A[i][i];s* = A[i][3-i];}          // 计算两条对角线元素的乘积
  return(s);
}
```

8. n 只猴子要选大王，选举办法如下：所有猴子按 1、2、...、n 编号围坐一圈，从第 1 号开始按 1、2、...、m 报数，凡报 m 号的退出圈外，如此循环报数，直到圈内剩下一只猴子时，这只猴子就是大王。n 和 m 由键盘输入，打印出最后剩下的猴子号。编写一个算法实现。

【提示】采用不带头结点的单循环链表实现。

```
typedef  struct  node
{ int  data;
  struct  node  *next;
}Node ,*LkList;
void  process (int n, int m)
{ L=(Node *) malloc (sizeof(Node));
  L->data=1;  L->next =L ; rear =L;          //生成链表中第一个结点
  for (i=2 ;i<=n ;i++)
```



```
  { s=(Node *) malloc (sizeof(Node));
    s->data=i ; s->next=rear->next;
    rear->next =s;  rear = s;
  }                                          //生成链表中其他 n-1 个结点
  p=L;
  while (p->next != p)                       //逐个输出元素
  { i=1;
    while(i<m)
      {q=p ; p=p->next ; i++}                //找报数为 m 的结点
    q->next=p->next;
    cout<<p->data<<endl ;
    free (p);
    p=q->next;
  }
cout<<"\n the last monky is :"<< p->data<<endl;
  }
```

9. 当具有相同行值和列值的稀疏矩阵 A 和 B 均以三元组表作为存储结构时，试写出矩阵相加算法，其结果存放在三元组表 C 中。

（参考典型例题解析中的算法设计题 2）

10. 假设稀疏矩阵 A 和 B（分别为 m×n 和 n×1 矩阵）采用三元组表示，编写一个算法计算 C=A×B，要求 C 也是采用稀疏矩阵的三元组表示。

（参见教材中的算法 5-4）

11. 假设稀疏矩阵只存放其非 0 元素的行号、列号和数值，以一维数组顺次存放，行号为-1 结束标志。例如：如图所示的稀疏矩阵 M：

$$M=\begin{pmatrix} 1 & 0 & 0 & 0 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

则存在一维数组 D 中：

D[0]=1，D[1]=1，D[2]=1，D[3]=1，D[4]=5

D[5]=10，D[6]=3，D[7]=9，D[8]=5，D[9]=-1

现有两个用如上方法存储的稀疏矩阵 A 和 B，它们均为 m 行 n 列，分别存放在数组 A 和 B 中，编写求矩阵加法 C=A+B 的算法，C 也放在数组 C 中。

【提示】注意，当 A、B 中元素的行和列都相同时才能相加。

```
void  add (int  A[], int B[], int C[])
{ pa=0; pb=0; pc=0;
  while (A[pa+2] && B[pb+2])                 //当 A、B 都未结束时
  { if (A[pa]= =B[pb] && A[pa+1]= =B[pb+1])  //当行号、列号都相同时
       { C[pc]=A[pa]; C[pc+1]=A[pc+1];
         C[pc+2]=A[pa+2]+B[pb+2] ;
         pa+=3; pb+=3 ;pc+=3 ;
       }
```

```
else                                                    //当行号、列号不等时
      if (A[pa]<B[pb]||(A[pa]= =B[pb] && A[pa+1]<B[pb+1]))
        {    C[pc]=A[pa];
             C[pc+1]=A[pc+1];
             C[pc+2]=A[pc+2];
             pa+=3; pc+=3;
        }
    else{    C[pc]=B[pb];
             C[pc+1]=B[pb+1];
             C[pc+2]=B[pb+2];
             pb+=3 ;pc+=3;
        }
}
 hile (A[pa+2]!=0)                                      //A 中有剩余元素
{    C[pc]=A[pa];
     C[pc+1]=A[pa+1];
     C[pc+2]=A[pc+2];
     pa+=3; pc+=3;
}
while (B[pa+2]!=0)                                      //B 中有剩余元素
{    C[pc]=B[pb];
     C[pc+1]=B[pb+1];
     C[pc+2]=B[pb+2];
     pb+=3; pc+=3;
}
return;
}
```

12. 已知 A 和 B 为两个 n×n 阶的对称矩阵，输入时，对称矩阵只输入下三角形元素，按压缩存储方法存入一维数组 A 和 B 中，编写一个计算对称矩阵 A 和 B 的乘积的算法。

【提示】注意，对对称矩阵采用压缩存储时，元素的表示方法；乘积矩阵仍然采用压缩存储的方法。

```
void mul (int A[] , int B[], int C[], int n)
{ // 计算A和B的乘积C，其中A、B、C均为压缩存储的n阶对称矩阵
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
    { mi=max(i,j);
      mj=min(i,j);
      x=mi*(mi-1)/2 + mj-1;    //计算矩阵元素C[i][j]压缩后的存放地址
      C[x]=0;
      for (k=0; k<n; k++)
      { u1=max(i,k) ;vi=min(i,k);
        u2=max(k,j) ;v2=min(k,j);
        w1=u1*(u1-1)/2 +v1-1;  //计算A[i][k]的存放地址
        w2=u2*(u2-1)/2 +v2-1;  //计算B[k][j]的存放地址
        C[x]+ = A[w1]*B[w2];
      }
    }
```



```
  return;
}
```

13. 假设 L 为非递归并且不带共享子表的广义表，设计一个复制广义表 L 的算法。（参见教材中的算法 5-12）

第 6 章　二叉树

二叉树结构是一种重要的非线性结构，是一种特殊的树形结构。由于二叉树的存储表示和基本操作的实现相对简单，故在应用计算机解决非线性问题时经常采用。本章介绍了二叉树的定义、二叉树的性质、存储结构、二叉树的遍历、线索二叉树、哈夫曼树及应用哈夫曼树进行编码等内容。

重点提示：

- 二叉树的二叉链表存储结构
- 二叉树的遍历
- 线索二叉树表示、建立与遍历
- 哈夫曼树的构造与应用

6-1　重点难点指导

6-1-1　相关术语

1．二叉树的概念及术语

（1）二叉树的定义：二叉树是 n（n>=0）个结点的有限集，它或者为空（n=0），或者由一个根结点及两棵互不相交的、分别称为这个根的左子树和右子树的二叉树组成。左子树和右子树又是一棵二叉树。

要点：二叉树也是一种递归的数据结构。

（2）二叉树不是二度树，它和树是两种不同的树形结构。

要点：二叉树的某一结点即使有一棵非空的子树也要表明它是左子树还是右子树；而树中若某结点的度为 1，则无需表明次序。

（3）满二叉树：若二叉树中每一层结点的个数都达到了最大，则称之为一棵满二叉树。

（4）完全二叉树：若一棵二叉树至多只有最下面的两层上结点的度数可以小于 2，并且最下一层上的结点都集中在该层最左边的若干位置上，则此二叉树称为完全二叉树。

满二叉树可以看作完全二叉树的特例，但完全二叉树未必是满二叉树。

（5）二叉树的性质

性质 1：二叉树的第 i 层上最多有 2^{i-1} 个结点。

性质 2：深度为 k 的二叉树最多有 2^k-1 个结点。

性质 3：任意一棵二叉树中，若终端结点的个数为 n_0，度数为 2 的结点为 n_2，则：$n_2=n_0-1$。

性质 4：具有 n 个结点的完全二叉树的深度 $k=\log_2 n+1$。

性质 5：对于具有 n 个结点的完全二叉树，按层次从上至下，每层从左至右对每个结点进行编号，则编号为 i 的结点有如下性质：

① 若 i=1，则 k_i 为根结点，无双亲，否则 k_i 的双亲编号为 i/2。

② 若 2i<=n，则 k_i 有左孩子，左孩子的编号为 2i，否则，k_i 无左孩子，k_i 肯定是叶子。

③ 若 2i+1<=n，则 k_i 有右孩子，右孩子的编号为 2i+1，否则，k_i 无右孩子。

④ 除根外，编号为奇数的结点为其双亲的右孩子，偶数结点为其双亲的左孩子。

2．线索二叉树

（1）线索二叉树：所谓线索二叉树是利用二叉链表中的 n+1 个空指针域存放指向结点在某种遍历次序下的前趋和后继结点的指针，为了区别指向左右孩子结点的指针，把前者称为线索，加了线索的二叉链表称为线索链表，相应的二叉树称为线索二叉树。按中序线索建立的线索树称为中序线索树；同理，有前序线索树和后序线索树。

（2）加线索的目的：遍历不加线索的二叉树（二叉链表存储）时，通常用递归方法，不用递归方法时，就要用到栈；遍历线索二叉树则不需要递归和堆栈，其时间性能为 O(n)。

3．哈夫曼树

（1）树的路径长度：是树根到树中每一结点的路径长度之和。显然，在结点数目相同的二叉树中，完全二叉树的路径长度最短。

（2）结点的带权路径长度：是结点到树根之间的路径长度与该结点上权的乘积。

（3）树的带权路径长度：是树中所有叶子结点的带权路径长度之和。树的带权路径长度也称为树的代价，公式如下：

$$WPL = \sum_{i=1}^{n} w_i l_i$$

其中：n 表示叶子结点的数目，w_i 是相应叶子的权值，l_i 是相应叶子到根结点的路径长度。

（4）最优二叉树（哈夫曼树）：在权为 w_1、w_2、…、w_n 的 n 个叶子所构成的所有二叉树中，带权路径长度最小（即树的代价最小）的二叉树称为最优二叉树或哈夫曼树。

（5）编码：将文件中的每个字符均转换为一个惟一的二进制串。

（6）解码：是编码的逆过程，即将二进制位串转换为对应的字符。

（7）前缀（编）码：为了保证解码的惟一性，因此在编码时要求字符集中任一字符的编码都不是其他字符编码的前缀，这种编码称为前缀（编）码。

（8）平均码长：设对 n 个字符$\{c_1, c_2, \ldots, c_n\}$进行编码，设字符 c_i 出现的概率为 p_i，码长为 l_i，则：

$$\text{平均码长} = \sum_{i=1}^{n} p_i l_i$$

（9）最优前缀码：平均码长最小的前缀码称为最优前缀码。

（10）哈夫曼编码：由哈夫曼树求得的编码是最优前缀码，称为哈夫曼编码。

（11）严格二叉树：只有度数为 2 和度数为 0 结点的二叉树称为严格二叉树。哈夫曼树是严格二叉树。

6-1-2　二叉树的存储结构

1．顺序存储

（1）完全二叉树的顺序存储

要点：按性质 5 对每一个结点进行编号，以编号为下标依次将每个结点存储在向量 bt[n+1] 中，其中 bt[1]～bt[n]用来存放编号为 1 到 n 的各结点，bt[0]用来存放结点的个数，显然这样

的存储方法直接满足性质5。

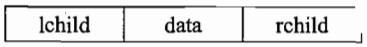
（2）非完全二叉树的顺序存储

要点：首先对该二叉树进行“完全化”，然后按完全二叉树的方法存储，所谓“完全化”是指按完全二叉树的编号对该二叉树进行编号，不存在的结点虚设一个空结点，存储时空结点也要占一个存储单元。

2．二叉链表存储

（1）存储思想：二叉链表中每一个结点存储 3 方面的信息：本结点的数据元素值和该结点的左、右孩子结点的指针（地址）。当结点的左右孩子为空时，相应的指针域为空指针。每个结点都是如此存储，则构成了二叉链表。

（2）二叉链表的结点结构：



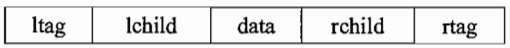

```
typedef struct node
   {  DataType     data;
      struct node  *lchild,*rchild;
   }BiTNode;
  typedef BiTNode   *BiTree;
```

注意：BiTNode 是结点的类型，BiTree 是指向 BitNode 的指针类型。

3．线索二叉树

（1）存储思想：利用二叉链表中的 n+1 个空指针域存放指向结点在某种遍历次序下的前趋和后继结点的指针并称其为线索，为了区别出某结点的指针域是指针还是线索，每个结点要加上两个标志域。按中序线索建立的线索树称为中序线索树；同理，有前序线索树和后序线索树。

（2）结点结构：



其中：ltag= 0 lchild 中为左孩子指针；1 lchild 中为前趋线索

rtag= 0 rchild 中为右孩子指针；1 rchild 中为后继线索

6-1-3 二叉树的遍历运算

1．二叉树的遍历

所谓二叉树的遍历是给定一种搜索方法，使得对二叉树中每个结点均做一次且仅做一次访问。

（1）先根遍历

若二叉树非空，则访问根结点，先先根遍历左子树，再先根遍历右子树。

算法实现提示：可以用递归方式实现，也可用非递归方式实现，访问结点的时机是在第

一次遇到结点的时候，即进入其左子树之前时访问。在用非递归方法实现时，需要用栈或其他别的方式记住当前进入的是哪个点的左子树，以便将来返回时可进入其右子树。

（2）中根遍历

若二叉树非空，则先中根遍历左子树，再访问根结点，最后中根遍历右子树。

算法实现提示：可以用递归方式实现，也可用非递归方式实现，访问结点的时机是在第二次遇到结点的时候，即进入其左子树之后并从其左子树返回时访问。在用非递归方法实现时，需要用栈或其他别的方式记住当前进入的是哪个点的左子树，以便将来返回时可访问该结点并进入其右子树。

（3）后根遍历

若二叉树非空，则先后根遍历左子树，再后根遍历右子树，最后访问根结点。

算法实现提示：可以用递归方式实现，也可用非递归方式实现。访问结点的时机是在第三次遇到结点的时候，即进入其左子树之后返回，又进入其右子树之后返回时才访问该结点。在用非递归方法实现时，需要用栈或其他别的方式记住当前进入的是哪个点的左子树，同时还需要设置标记标识返回前进入的是左子树还是右子树，以便将来从右子树返回时访问该结点。

（4）层次遍历

从二叉树的根结点开始，按自上而下，从左到右的顺序进行遍历。

算法实现提示：可以利用队列帮助实现遍历。

（5）有关结论

由以上遍历的定义得知：

一棵二叉树先序遍历所得序列中第一个结点为根结点；一棵二叉树中序遍历序列中，根结点在其左子树中序序列和右子树中序序列之间 ；一棵二叉树后序遍历所得序列中最后一个结点为根结点 。

因此，可以根据先序遍历所得序列及中序遍历所得序列确定原二叉树的结构：由先序序列知道根后，再由中序序列划分出左右子树，对每一棵子树再依次类推。

同样也可以根据后序遍历所得序列及中序遍历所得序列确定原二叉树的结构：由后序序列知道根后，再由中序序列划分出左右子树，对每一棵子树再依次类推。

但由先序遍历所得序列及后序遍历所得序列均不能确定原二叉树的结构：由先序序列和后序序列都知道根，而无法划分出左右子树。

2．线索二叉树的遍历

（1）对未建立线索的二叉树可通过对二叉树的先根、中根和后根遍历分别建立先序线索树、中序线索树和后序线索树。

（2）对已建立线索的二叉树，可以不用递归、不用堆栈，只要通过线索树的头结点找到某种二叉树遍历顺序下的第一个结点或最后一个结点，就可以利用线索二叉树的特点，找到其后继或前驱结点，从而依次访问到二叉树的所有结点。遍历的效率比较高。

6-1-4 二叉树的基本应用

1．哈夫曼树的构造过程

（1）前提条件：给出 n 个实数{ $w_1,w_2,\ldots,w_n$}。

（2）操作结果：以给出的 n 个实数作为叶子的权，构造出哈夫曼树。

（3）构造过程如下：

需要进行 n-1 次合并。将开始给定的 n 个权值看作只有根结点的 n 棵二叉树，每次合并是将选出的两棵根结点的权值最小的树分别作为左子树和右子树合并成一棵新树，为了保证新树是二叉树，需要增加一个新结点作为新树的根结点，新树的权值为其左右子树根结点的权值之和，这样的合并进行 n-1 次后，最后的二叉树就是所得的哈夫曼树。

（4）有关哈夫曼树的结论

① 哈夫曼树是严格二叉树。

② 对于有 n 个叶子结点的哈夫曼树共有 2×n-1 个结点。

③ 哈夫曼树的形状不是惟一的，但带权路径长度是最小的。

2. 哈夫曼编码过程

（1）前提条件：给出 n 个字符及每个字符出现的频度{ $w_1,w_2,\ldots,w_n$}。

（2）操作结果：

① 以给出的 n 个频度（实数）作为叶子的权，构造出哈夫曼树。

② 根据所构造的哈夫曼树对每个字符进行编码。

（3）编码过程：树中从根到每个叶子都有一条路径，对路径上的各分支约定指向左子树根的分支编码为“0”，指向右子树根的分支编码为“1”，那么从根到每个叶子相应路径上的“01”序列就是这个叶子（字符）的编码。

6-2 典型例题解析

6-2-1 选择题

1. 某二叉树 T 有 n 个结点，设按某种顺序对 T 中的每个结点进行编号，编号值为 1、2、...n。且有如下性质：T 中任意结点 v，其编号等于左子树上的最小编号减一，而 v 的右子树的结点中，其最小编号等于 v 左子树上结点的最大编号加一，这是按（　）编号的。

A．中序遍历序列　B．前序遍历序列　C．后序遍历序列　D．层次顺序

【分析】据题意，对二叉树的结点进行编号应满足根结点的编号要小于其左子树所有结点的编号，而其左子树所有结点的编号要小于其右子树所有结点的编号，因此可按照根左右（即前序）顺序遍历进行编号。

【答案】B

2．下列说法正确的是（　）。

A．二叉树中任何一个结点的度都为 2

B．二叉树的度为 2

C．一棵二叉树的度可小于 2

D．任何一棵二叉树中至少有一个结点的度为 2

【分析】二叉树是树形结构的一种，但不是二度树，也不是二度树的特例，比如二叉树包括空树，也包括只有一个根结点的情况，也包括只有度为 0 和度为 1 结点的情况。

【答案】C

3．欲实现任意二叉树的后序遍历的非递归算法而不使用栈结构，最佳的方案是二叉树

采用（　）存储结构。

A．三叉链表　B．广义表　C．二叉链表　D．顺序表

【分析】由于三叉链表有一个指针指向父结点，不用栈便可在不需要向分支结点深入时或无法深入时，回退到父结点。

【答案】A

4．一棵有 124 个叶子结点的完全二叉树，最多有（　）个结点。

A．247　B．124　C．248　D．125

【分析】由 $n_0=n_2+1$ 可知：$n_2=123$，又因 n_0 为偶数，所以 $n_1=0$。因此 n=247。

【答案】A

5．若以二叉树的任一结点出发到根的路径上所经过的结点序列按其关键字有序，则该二叉树是（　）。

A．二叉排序树　B．哈夫曼树　C．堆　D．线索二叉树

【分析】哈夫曼树、线索二叉树与其结点的关键字无关。一般情况下，从二叉排序树任一结点出发到根的路径上所经过的结点序列的关键字序列不是有序的，而在堆中，从任一结点出发到根的路径上所经过的结点序列的关键字序列一定是有序的。

【答案】C

6．一棵非空的二叉树的先序遍历序列与后序遍历序列正好相反，则该二叉树一定满足（　）。

A．所有的结点均无左孩子　B．所有的结点均无右孩子

C．只有一个叶子结点　D．是任意一棵二叉树

【分析】先序遍历顺序为：根左右，后序遍历顺序为：左右根。若要使先序遍历序列与后序遍历序列相反，则先序和后序遍历应具有的形式为：根左、左根，或者为根右、右根。

【答案】C

6-2-2 判断题

1．若一个结点是某二叉树子树的中序遍历序列中的第一个结点，则它必是该子树的后序遍历序列中的第一个结点。

【分析】中序遍历按照左根右的顺序对二叉树进行遍历，后序遍历按照左右根的顺序对二叉树进行遍历，都是从二叉树的最左端开始，因此对于同一棵二叉树，它们遍历的第一个结点是相同的。

【答案】正确

2．在前序遍历二叉树的序列中，任何结点的子树中的所有结点不一定在该结点之后。

【分析】前序遍历是按照根左右的顺序对二叉树进行遍历，其任何结点的左右子树结点都应该在该结点之后被遍历到。

【答案】错误

3．在后序线索二叉树中，能够很方便地找到任意结点的后继。

【分析】在后序线索二叉树中，当一个结点的右指针不是线索时，其后继结点可能是其双亲结点，也可能是其双亲结点右子树的最左端结点，不能够很方便地得到。

【答案】错误

4．哈夫曼树是带权路径长度最短的树，路径上权值较大的结点离根较近。

【分析】因为这样才能保证其带权路径长度最短。

【答案】正确

5．完全二叉树可采用顺序存储结构实现存储，非完全二叉树则不能。

【分析】对于非完全二叉树进行顺序存储时需按照完全二叉树顺序存储的顺序将空结点也存入。

【答案】错误

6．存在这样的二叉树，对其采用任何次序的遍历，结果相同。

【分析】当二叉树只有一个结点时，三种遍历序列均相同。

【答案】正确

6-2-3 填空题

1．深度为 k 的完全二叉树至少有_____个结点，至多有_____个结点。

【分析】深度为 k 的完全二叉树最少结点时是前 k−1 层为满二叉树，第 k 层只有一个结点，故有 2^{k-1} 个结点；深度为 k 的完全二叉树最多结点时是 k 层的满二叉树。

【答案】2^{k-1}；2^k-1

2．具有 n 个结点的完全二叉树，若按自上而下、从左至右依次给结点编号，则编号最小的叶子结点的序号是_____。当 i 为奇数且不等于 1 时，结点 i 的左兄弟是结点_____，否则结点 i 没有左兄弟；当 i 为偶数且不等于 n 时，结点 i 的右兄弟是结点_____，否则结点 i 没有右兄弟。

【分析】设完全二叉树的深度为 k，因为编号最小的叶子结点是第 k 层上最左边的结点，它的编号是第 k−1 层最右边结点的编号加 1，而前 k−1 层共有 $2^{k-1}-1$ 个结点，所以它的编号为 2^{k-1}，根据二叉树的性质知：$k=\lfloor \log_2 n \rfloor+1$。

【答案】2^{k-1}（其中 $k=\lfloor \log_2 n \rfloor+1$）；i−1；i+1

3．在具有 n 个结点的 k 叉树（k≥2）的 k 叉树链表表示中，空指针有_____个。

【分析】因为每个结点有 k 个指针域，共有 n×k 个指针域，而只有除根结点之外的 n−1 个结点的地址存到了相应的指针域中，故还有 n×k−（n−1）=(k−1)×n+1 个空指针域。

【答案】(k−1)×n+1

4．高度为 h 的严格二叉树至少有_____个结点？至多有_____个结点。

【分析】因为满二叉树是严格二叉树，所以高度为 h 的严格二叉树至多有 2^h-1 个结点。严格二叉树除第一层根结点有一个，其他层上至少有两个结点，所以严格二叉树最少有 2h−1 个结点。

【答案】2h−1；2^h-1

5．在_____情况下，等长编码是最优的前缀码。

【分析】对于使用频度相等的字符集进行编码时，等长编码是最优的前缀码。

【答案】当字符集中的各字符使用频率相等

6-2-4 应用题

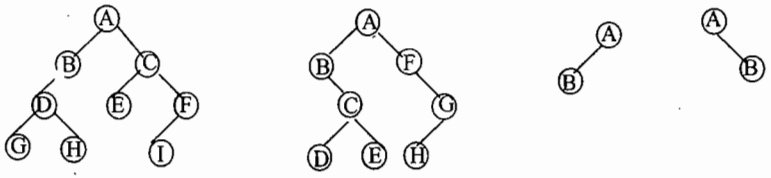
1．（1）已知一棵二叉树的前序序列和中序序列分别为 ABDGHCEFI 和 GDHBAECIF，求其对应的二叉树。（2）已知一棵二叉树的中序序列和后序序列分别为 BDCEAFHG 和 DECBHGFA，求其对应的二叉树。（3）若前序序列和后序序列均为 AB 和 BA 能否惟一确定一棵二叉树。

【解答】

（1）知道二叉树的前序序列和中序序列能惟一确定一棵二叉树，（1）对应的二叉树如图（1）所示。

（2）知道二叉树的中序序列和后序序列能惟一确定一棵二叉树，（2）对应的二叉树如图（2）所示。

（3）知道二叉树的前序序列和后序序列不能惟一确定一棵二叉树，如图（3）所示的两棵二叉树的前序和后序序列是相同的。



（1） （2） （3）

根据前序、中序和后序的遍历特点，如果知道一棵树的前序序列，则第一个结点就是根结点，后序序列的最后一个结点是根结点，但都没有办法确定左右子树，而对于中序来说，当知道了根结点之后，恰好能分离出左右子树来，因此在二叉树中的结点值均不相同的情况下，则由二叉树的前序序列和中序序列，或由其后序序列和中序序列均能惟一地确定一棵二叉树，而由前序序列和后序序列却不能惟一地确定一棵二叉树。

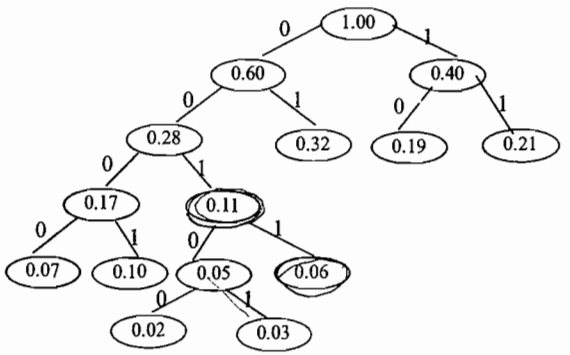
2．用于通信的电文由字符集{a,b,c,d,e,f,g,h}中的字母构成，这 8 个字母在电文中出现的概率分别为{0.07，0.19，0.02，0.06，0.32，0.03，0.21，0.10}。

（1）为这 8 个字母设计哈夫曼编码。

（2）若用三位二进制数（0～7）对这 8 个字母进行等长编码，则哈夫曼编码的平均码长是等长编码的百分之几？它使电文总长平均压缩多少？

【解答】

（1）首先构造哈夫曼树：



根据哈夫曼树，各字符的编码如下：

字符	频率	编码
a	0.07	0000
b	0.19	10
c	0.02	00100
d	0.06	0011
e	0.32	01
f	0.03	00101
g	0.21	11
h	0.10	0001

（2）哈夫曼编码平均码长为：

$4\times0.07+2\times0.19+5\times0.02+4\times0.06+5\times0.03+2\times0.21+4\times0.1=2.71$

因为等长编码的平均码长为 3，所以哈夫曼编码是等长编码的 2.71/3=0.90，它使电文平均压缩了近 10%。

6-2-5 算法设计题

1．已知二叉树用二叉链表存储，写一个算法将二叉树中的叶子结点按从右至左的顺序建立一个单链表。

【分析】

这是一个非常典型的基于二叉树遍历算法，通过遍历找到各个叶子结点，因为不论前序遍历、中序遍历和后序遍历，访问叶子结点的相对顺序都是相同的，即都是从左至右，而题目要求是将二叉树中的叶子结点按从右至左顺序建立一个单链表，因此，可以采用 3 种遍历中的任意一种方法遍历，当访问的结点是叶子结点时，建立一个叶子结点，将此结点插到单链表的头部，恰好满足题目的要求，即单链表采用头插法建立，下面算法采用中序遍历的方法，为了简便，算法中的 Leafhead 是一个全局变量，表示最后单链表的头指针。

【算法】

```
void  Inorder(BiTree T)
  {  if(T)
    { Inorder(T->lchild);
      If ((!T->lchild)&& (!T->rchild))
          { s=(ListNode *)malloc(sizeof( ListNode)); //申请一个单链表结点
            s->data=T->data;
            s->next=Leafhead;
            Leafhead=s;
          }
       Inorder(T->rchild);
    }
}
```

2．以二叉链表为存储结构，分别写出求二叉树高度及宽度的算法。所谓宽度是指在二叉树的各层上，具有结点数最多的那一层上的结点总数。

（1）求树的高度的算法：

【分析】

空二叉树的深度为 0，对非空二叉树，其深度等于根结点的左子树和右子树中深度较大

者的子树的深度加 1。故根据如下表达式采用后序递归遍历算法实现：

$$\text{二叉树的深度}=\begin{cases}0 & \text{若二叉树为空}\\ \max(\text{左子树深度，右子树深度})+1 & \text{若二叉树非空}\end{cases}$$

【算法】

```
int Depth(BiTree T)
  { int dep1,dep2;
    if(T==NULL) return(0);
    else
      {dep1=Depth(T->lchild);      //求左子树的深度
       dep2=Depth(T->rchild);      //求右子树的深度
       if(dep1>dep2) return(dep1+1);
       else return(dep2+1);
      }
}
```

（2）求树的宽度的算法：

【分析】

按层次遍历二叉树，c1 用来存储当前层次中的结点数，c2 用来存储到目前为止某层中结点最多的数目，为了进行层次遍历，采用一个队列 Q，队尾指针为 rear，队头指针为 front，指针 p 用来标记本层中最右边的元素在队列中的位置。开始时让根结点入队列，队列非空时，队头结点出队列，若该结点有左右子树，则左右子树根结点入队列，如此反复，一层一层地处理，直到队列为空，c2 中保存的就是队列的宽度。

【算法】

```
int Width(BiTree T)
{ BiTree Q[MAXNODE];                    //用作队列
  int front=0,rear=-1;                  //队列初始化
  int flag=0,count=0,p;
    //  p 用于指向树中层的最右边的结点，标志 flag 记录层中结点数的最大值。
if(T!=NULL)
  { rear++;
    Q[rear]=T;
    flag=1;
    p=rear;
   }
  while(front<p)                        //当前层的结点未访问完
   {
     front++;
     T=Q[front];                        //出队
     if(T->lchild!=NULL)                //若出队结点有左孩子，左孩子入队
      { rear++;
        Q[rear]=T->lchild;
        count++;
      }
     if(T->rchild!=NULL)                //若出队结点有右孩子，右孩子入队
      {rear++;
       Q[rear]=T->rchild;
```

```
        count++;
        }
      if(front==p)                          //当前层已遍历完毕
       {if(flag<count) flag=count;          //刚遍历的这一层的结点数更多
        count=0;
        p=rear;
       }
     }
    return(flag);
   }
```

3. 一棵 n 个结点的完全二叉树以向量作为存储结构，试写一非递归算法实现对该树的前序遍历。

【分析】

根据顺序存储的完全二叉树的特点：对于编号为 i 的结点的左孩子的编号为 2i，右孩子的编号为 2i+1，又根据前序遍历的特点，先访问根结点，然后前序遍历其左子树，再前序遍历其右子树，为了遍历完一棵左子树后回到其根的右子树则必须将其根结点的指针入栈保存，因此当前序、中序、后序遍历一棵由二叉链表存储的二叉树时都要用到栈，具体做法是：访问根结点，将根结点指针入栈。当栈不空时，出栈，对于出栈结点如有非空的左右孩子，因为前序遍历是先处理其左子树，为了处理完其左子树后回到根的右子树，则先让右孩子入栈，然后左孩子入栈，如此反复实现前序遍历。

【算法】

```
Perorder(char a[])
{ // 前序遍历由向量a[]存储的具有n个结点的完全二叉树
  int s[n];              //定义一个向量用作栈的存储空间
  int top=-1;            //top为栈顶指针，将栈置为空
  int i=1;
  if(n==0)return;
  else
    { top++; s[top]=i;
      while(i>-1)
        { printf("%c",a[s[top]]);
          top--;
          if((2*i+1)<n)
            { i=2*i;
              top++;
              s[top]=i+1;
              top++;
              s[top]=i;
            }
            else if(2*i<n)
                 { i=2*i;
                   top++;
                   s[top]=i;
                 }
        }
    }
}
```


4. 以二叉链表为存储结构，写一算法交换各结点的左右子树。

【分析】

依题意，设 t 为一棵用二叉链表存储的二叉树，则交换各结点的左右子树的运算基于后序遍历实现：交换左子树上各结点的左右子树；交换右子树上各结点的左右子树；再交换根结点的左右子树。

【算法】

```
Exchg(BiTree *t)
 { BinNode *p;
   if (t)
     { Exchg(&((*t)->lchild));
       Exchg(&((*t)->rchild));
       P=(*t)->lchild;
       (*t)->lchild=(*t)->rchild;
       (*t)->rchild=p;
     }
}
```

5. 假设二叉树用二叉链表表示，试编写一算法，判别给定二叉树是否为完全二叉树。

【分析】

根据完全二叉树的定义可知，对完全二叉树按照从上到下，从左到右的次序遍历应满足：① 若某结点没有左孩子，则一定无右孩子；② 若某结点缺（左或右）孩子，则其所有后继一定无孩子。因此，可采用按层次遍历二叉树的方法依次对每个结点进行判断。这里增加一个标志以表示所有已扫描过的结点均有左、右孩子，将局部判断结果存入 CM 中，CM 表示整个二叉树是否是完全二叉树，B 为 1 表示到目前为止所有结点均有左右孩子。

【算法】

```
int CBiTee(BiTree *t)
{Init_Queue(Q);                         //初始化队列Q
 B=1;
 CM=1;
 if(t!=NULL)
   {In_Queue(Q,t);
    while(!Empty_Queue(Q))              //当队列不为空时执行循环
     {p=Out_Queue(Q);                   //出队
      if(p->lchild==NULL)
        {B=0;                           //B=0表示缺少左、右孩子
         if(rchild!=NULL)
         CM=0;                          //CM=0表示不是完全二叉树
        }
      else
          {CM=B;
           In_Queue(Q,p->lchild);       //左孩子入队列
           if(p->rchild==NULL)
              B=0;
           else
              In_Queue(Q,p->rchild);    //右孩子入队列
          }
     }
```

```
        return(CM);
      }
  }
```

6. 假设二叉树用二叉链表表示，数据域 data 是字符型数据，试编写一算法，求任意二叉树中第一条最长的路径，并输出此路径上各结点的值。

【分析】

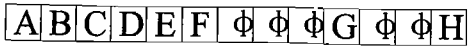
可采用非递归后序遍历二叉树，当后序遍历访问到由 p 所指的树叶结点时，此时 stack 中所有结点均为 p 所指结点的祖先，由这些祖先便构成了一条从根结点到此树叶结点的路径。此外，另设一 longestpath 数组来保存二叉树中最长的路径结点值，m 为最长的路径长度。

【算法】

```
void LongPath(BiTree *root)
{Bitree stack[Maxsize],s;
 char longestPath[Maxsize];
 int I,m,top;
 m=0;
 top=0;
 s=root;
 while((s!=NULL)||(top!=0))
   {while(s!=NULL)                    //扫描左孩子，入栈
      {top++;
       stack[top]=s;
       tag[top]=0;                    //右孩子还未访问过
       s=s->lchild;
      }
      if(top>0)
        {if(tag[top]==1)              //左、右孩子均已访问过，则访问该结点
          {if((stack[top]->lchild==NULL)&&( (stack[top]->rchild==NULL)&&(top>m))
            {for(i=1;i<top;i++)
             longestPath[i]=stack[i]->data;
             m=top;
            }
           top--;
          }
        else
          {s=stack[top];
           if(top>0)
             {s=s->rchild;            //扫描右孩子
              tag[top]=1;             //置当前结点右子树已访问过标志
             }
          }
        }
   }
for(i=1;i<=m;i++)
  cout <<longestPath[i];
cout<<"longest=";
cout<<m;
}
```


6-3　课后习题选解

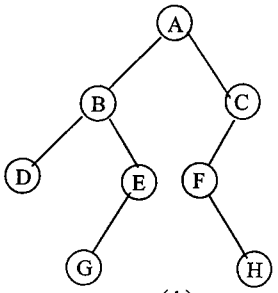
1. 对于下图所示二叉树，试给出：



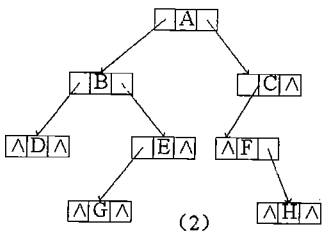
（1）它的顺序存储结构示意图。

（2）它的二叉链表存储结构示意图。

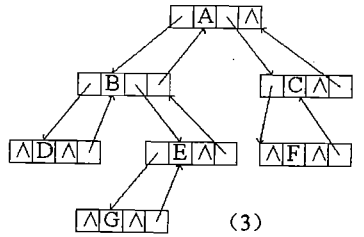
（3）它的三叉链表存储结构示意图。



（1）



（2）



（3）

2. 证明：在结点数多于 1 的哈夫曼树中不存在度为 1 的结点。

证明：由哈夫曼树的构造过程可知，哈夫曼树的每一分支结点都是由两棵子树合并产生的新结点，其度必为 2，所以哈夫曼树中不存在度为 1 的结点。

3. 证明：若哈夫曼树中有 n 个叶结点 ，则树中共有 2n-1 个结点。

证明：n 个叶结点，需经 n-1 次合并形成哈夫曼树，而每次合并产生一个分支结点，所以树中共有 2n-1 个结点。

4. 证明：由二叉树的前序序列和中序序列可以惟一地确定一棵二叉树。

由教材中的算法 6-11 可知结论成立。

5. 已知一棵度为 m 的树中有 n_1 个度为 1 的结点，n_2 个度为 2 的结点，……，n_m 个度为 m 的结点，问该树中共有多少个叶子结点？有多少个非终端结点？

解：设树中共有 n 个结点，n_0 个叶结点，那么

$$n=n_0+n_1+\ldots+n_m \quad (1)$$

树中除根结点外，每个结点对应着一个分支，而度为 k 的结点发出 k 个分支，所以

$$n=n_1+2\times n_2+\ldots+m\times n_m+1 \quad (2)$$

由（1）（2）可知 $n_0=n_2+2\times n_3+3\times n_4+\ldots+(m-1)\times n_m+1$

6．设高度为 h 的二叉树上只有度为 0 和度为 2 的结点，问该二叉树的结点数可能达到的最大值和最小值。

最大值：2^h-1；最小值：2h-1

7．求表达式(a+b*(c-d))-e/f 的波兰式（前缀式）和逆波兰式（后缀式）。

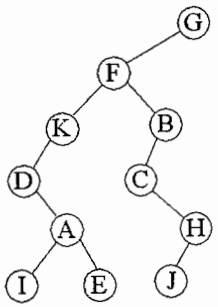
波兰式：- + a * b - c d / e f

逆波兰式：a b c d - * + e f / -

8．画出和下列已知序列对应的二叉树：

二叉树的先序次序访问序列为：GFKDAIEBCHJ。

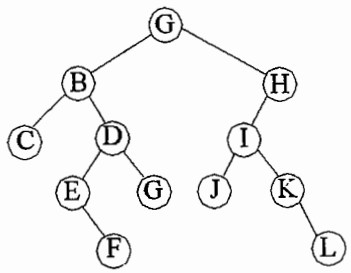
二叉树的中序访问次序为：DIAEKFCJHBG。



9．画出和下列已知序列对应的二叉树：

二叉树的后序次序访问序列为：CFEGDBJLKIHA。

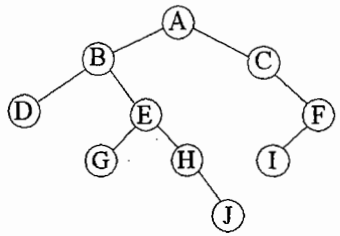
二叉树的中序访问次序为：CBEFDGAJIKLH。



10．画出和下列已知序列对应的二叉树：

二叉树的层次序列为：ABCDEFGHIJ。

二叉树的中序次序为：DBGEHJACIF。



11．给定一棵二叉树，其根指针为 root。试写出求二叉树结点数目的算法（递归算法或非递归算法）。

【提示】采用递归算法实现。

$$\text{二叉树结点的数目}=\begin{cases}0 & \text{当二叉树为空}\\ \text{左子树结点数目}+\text{右子树结点数目}+1 & \text{当二叉树非空}\end{cases}$$

```
int count (BiTree t)
{if (t= =NULL)    return (0);
         else     return (count(t->lchild)+count(t->rchild)+1);
}
```

12．已知二叉树用二叉链表存储，写一个算法将二叉树中的叶子结点按从右至左的顺序建立一个单链表。

（参考典型例题解析中的算法设计题 1）

13．以二叉链表为存储结构，分别写出求二叉树高度及宽度的算法。所谓宽度是指在二叉树的各层上，具有结点数最多的那一层上的结点总数。

（参考典型例题解析中的算法设计题 2）

14．给定一棵用链表表示的二叉树，其根指针为 root。试写出求二叉树各结点的层数的算法。

【提示】采用先序递归遍历算法实现。

$$\text{二叉树结点的层次数}=\begin{cases}1 & \text{当结点为根结点}\\ \text{其双亲结点的层次数}+1 & \text{当结点非根结点}\end{cases}$$

```
void  fun (BiTree  t, int n)
{ if (t= =NULL)     return;
  else { cout<<n<<endl;
         fun(t->lchild,n+1);
         fun(t->rchild,n+1);
       }
}
```

15．以二叉链表为存储结构，写一算法交换各结点的左右子树。

（参考典型例题解析中的算法设计题 4）

16．一棵 n 个结点的完全二叉树以向量作为存储结构，试写一非递归算法实现对该树的前序遍历。

（参考典型例题解析中的算法设计题 3）

17. 在二叉树中查找值为 x 的结点，试设计打印值为 x 的结点的所有祖先结点的算法。

【提示】对二叉树进行先序非递归遍历，查找值为 x 的结点。进入子树时，将子树的根压栈；从子树返回时，将栈顶元素出栈。当找到值为 x 的结点时，栈中存放的就是其祖先结点，依次打印各结点的值。

```
void printnode (BiTree t, elemtype x)
{ init_Stack(s);                    //初始化空栈
  p=t;
  while (p|| !Empty_Stack(s))       //若 p 非空，或栈非空
  {while (p)
    {if (p->data==x)                //若当前结点值为 x，依次输出栈中元素的值
      {while (!Empty_Stack(s))
        { pop(s,q);
          printf(q->data);
        }
      return;
      }
    else{ Push(s,p);}               //若当前结点值不是 x，压栈
          p=p->lchild;
       }
  if(!Empty_Stack(s)) {pop(s,r);  //当栈非空，栈顶元素出栈，进入右子树
                       p=r->rchild;
                       }
    else return;
  }
}
```

18. 已知一棵二叉树的后序遍历序列和中序遍历序列，写出可以惟一确定一棵二叉树的算法。

【提示】根据后序遍历和中序遍历的特点，采用递归算法实现。

```
void InPost (char in [ ] , char post [ ] , int il , int ir, int pl, int pr, BiTree *t)
   // 数组 in 和数组 post 中存放着二叉树的中序遍历序列和后序遍历序列，il 和 ir 表示
  // 中序遍历序列的左右端点，pl 和 pr 表示后序遍历序列的左右端点，t 表示二叉树的根
{ *t=(BiTNode *) malloc (sizeof(BiTNode));
  *t->data=post[pr];
  m=il;
  while (in[m]! = post [pr] )  m++;
  if (m== il) *t->lchild=NULL ;
        else  InPost ( in, post,il,m-l,pl,pl+m-l-il, &(*t->lchild));
  if (m= =ir) *t->rchild=NULL;
        else  InPost (in,post,m+l,ir,pr-ir+m,pr-l,&(*t->rchild));
}
```

19. 在中序线索二叉树上插入一个结点 p 作为树中某结点 q 的左孩子，试给出实现上述要求的算法。

【提示】在线索二叉树上插入结点，除了修改结点之间的逻辑关系，还要修改线索。


```
void InsertThrLeft(BiThrTree q,BiThrTree p)
{
   BiThrTree w;
   p->lchild=q->lchild;
   p->ltag=q->ltag;
   p->rchild=q;
   p->rtag=l;
   q->lchild=p;
   q->ltag=0;
   if(p->ltag==0)
     { w=p->lchild;
       while(w->rtag==0)w=w->rchild
       w->rchild=p;
     }
}
```

第 7 章　树和森林

树形结构是一类重要的非线性结构。树形结构的特点是结点之间具有层次关系。本章介绍树的定义、存储结构、树的遍历方法、树和森林与二叉树之间的转换以及树的应用等内容。

重点提示：

- 树的存储结构
- 树的遍历
- 树和森林与二叉树之间的转换

7-1　重点难点指导

7-1-1　相关术语

1．树的定义：树是 n（n>=0）个结点的有限集 T，T 为空时称为空树，否则它满足如下两个条件：①有且仅有一个特定的称为根的结点；②其余的结点可分为 m（m>=0）个互不相交的子集 T1，T2，…，Tm，其中每个子集本身又是一棵树，并称为根的子树。

要点：树是一种递归的数据结构。

2．结点的度：一个结点拥有的子树数称为该结点的度。

3．树的度：一棵树的度指该树中结点的最大度数。如图 7-1 所示的树为 3 度树。

4．分支结点：度大于 0 的结点为分支结点或非终端结点。如结点 a、b、c、d。

5．叶子结点：度为 0 的结点为叶子结点或终端结点。如 e、f、g、h、i。

6．结点的层数：树是一种层次结构，根结点为第一层，根结点的孩子结点为第二层，…依次类推，可得到每一结点的层次。

7．兄弟结点：具有同一父亲的结点为兄弟结点。如 b、c、d；e、f；h、i。

8．树的深度：树中结点的最大层数称为树的深度或高度。

9．有序树：若将树中每个结点的子树看成从左到右有次序的（即不能互换），则称该树为有序树（Ordered Tree），否则称为无序树（Unodered Tree）。

10．森林：是 m 棵互不相交的树的集合。

7-1-2　树的存储结构

1．双亲链表表示法

以图 7-1 所示的树为例。

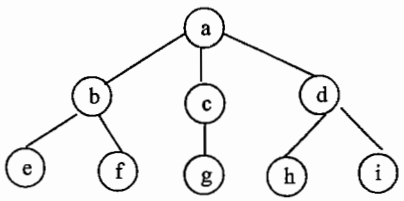


图 7-1　一棵深度为 3 的树

（1）存储思想:因为树中每个元素的双亲是惟一的，因此对每个元素，将其值和一个指向双亲的指针 Parent 构成一个元素的结点，再将这些结点存储在向量中。

（2）存储示意图:

	0	1	2	3	4	5	6	7	8	9	…	MaxTreeSize-1
data:	a	b	c	d	e	f	g	h	i		…	
Parrent:	-1	0	0	0	1	1	2	3	3		…	

（3）注意: Parrent 域存储其双亲结点的存储下标，而不是存放结点值。

下面的存储是不正确的：

	0	1	2	3	4	5	6	7	8	9	…	MaxTreeSize-1
data:	a	b	c	d	e	f	g	h	i		…	
Parrent:	-1	a	a	a	b	b	c	d	d		…	

2．孩子链表表示法

（1）存储思想：

将每个数据元素的孩子拉成一个链表，链表的头指针与该元素的值存储为一个结点，树中各结点顺序存储起来，一般根结点的存储号为 0。

（2）存储示意图:

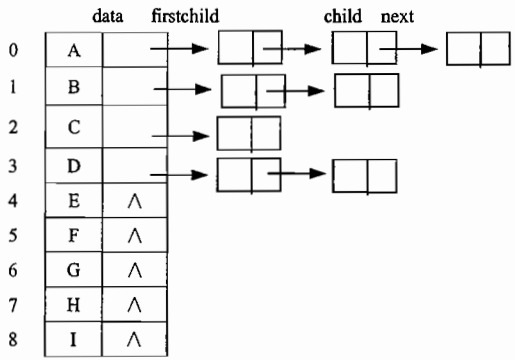


图 7-2　孩子链表存储示意图

（3）需注意的问题：

① 每个结点的孩子结点的 child 域存放该孩子的存储序号，而不是存放其结点的值。

② 孩子链表的最后一个结点的 next 域置入空指针。

③ 叶子结点的 firstchild 域也要置入空指针。

3．双亲-孩子兄弟链表表示法

将双亲表示和孩子兄弟链表表示法结合起来。

4．孩子兄弟链表表示法

（1）存储思想是：树中每个数据元素存储为一个结点，结点的结构如下：

leftmostchild	data	rightsilling

其中：data 为该数据元素的信息；

leftmostchild 为该元素第一个孩子的指针；

rightsilling 为该元素右兄弟的指针。

（2）存储示意图：

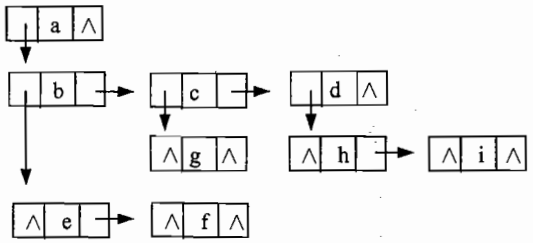


图 7-3　孩子-双亲链表存储示意图

7-1-3　树的基本运算

1．树的遍历

（1）前序遍历:若树 T 非空，则：

① 访问根结点 R；

② 依次前序遍历根 R 的各个子树 T_1、T_2、…、T_k。

（2）后序遍历:若树 T 非空，则：

① 依次后序遍历根 R 的各个子树 T_1、T_2、…、T_k；

② 访问根结点 R。

2．森林的遍历

（1）前序遍历：若森林 F 非空，则前序遍历森林中的第一棵树，第二棵树，… ，最后一棵树。

（2）后序遍历：若森林 F 非空，则后序遍历森林中的第一棵树，第二棵树，… ，最后一棵树。

7-1-4　树、森林和二叉树的相互转换

1．树转换成为二叉树

要点：

（1）转换规则：

① 原树的根仍为转换后二叉树的根；

② 原树中每一个非终端结点的第一个孩子转换后成为其双亲的左孩子；

③ 原树中每一个结点右边的第一个兄弟孩子转换后成为它的右孩子。

（2）转换为二叉树后的特点：

① 所转换成的二叉树其根结点的右子树为空；

② 其二叉链表可解释为原树的孩子-兄弟链表。

2．森林转换为二叉树

要点：

转换规则：将森林中第二棵树的树根结点视为第一棵树树根的第一个兄弟，第三棵树的树根结点视为第二棵树树根的第一个兄弟，依次类推，仍按树的转换规则进行即可。

7-2　典型例题解析

7-2-1　选择题

1．以下说法错误的是（　）。

A．存在这样的二叉树，对其采用任何次序的遍历其结点访问序列均相同。

B．二叉树是树的特殊情形。

C．由树转换成二叉树，其根结点的右子树总是空的。

D．在二叉树只有一棵子树的情况下，也要指出是左子树还是右子树。

【分析】二叉树的特点是仅有一个分支时也有左右之分，而树无此特点，也即二叉树不是树的特殊情形。

【答案】B

2. 如果 T′是由有序树 T 转换而来的二叉树，那幺 T 中结点的后序就是 T′中结点的(　)。

A．先序　　B．中序　　C．后序　　D．层次序

【分析】树的先序遍历序列与其转换的二叉树的先序遍历序列相同，树的后序遍历序列与其转换的二叉树的中序遍历序列相同。

【答案】B

3．若一个具有 N 个顶点，K 条边的无向图是一个森林（N>K），则该森林中必有（　）棵树。

A．K　　B．N　　C．N－K　　D．1

【分析】由于一棵有 n 个顶点的树具有 n－1 条边，故设森林中有 m 棵树，每棵树有 vi 个顶点（1≤i≤m），则有：

$$v1+v2+\ldots+vm=N$$

$$(v1-1)+(v2-1)+\ldots+(vm-1)=K$$

以上两式相减得：m=N-K。

【答案】C

4．设 F 是一个森林，B 是由 F 变换得到的二叉树。若 F 中有 n 个非终端结点，则 B 中右指针域为空的结点有（　）个。

A．n-1　　B．n　　C．n+1　　D．n+2

【分析】通过一个简单的森林及其对应二叉树的实例即可确定答案。

【答案】C

7-2-2　判断题

1．由树转换成二叉树，根结点没有左子树。

【分析】由树转换成二叉树，根结点必有左子树而没有右子树。

【答案】错误

2．用树对应二叉树的前序遍历和中序遍历可以导出树的后根遍历序列。

【分析】因为后根遍历树和中序遍历与该树对应的二叉树结果相同。

【答案】正确

3．先根遍历一棵树和前序遍历与该树对应的二叉树，其结果不同。

【分析】先根遍历一棵树和前序遍历与该树对应的二叉树结果相同。

【答案】错误

4．前序遍历森林和前序遍历与该森林对应的二叉树，其结果不同。

【分析】前序遍历森林和前序遍历与该森林对应的二叉树结果相同。

【答案】错误

5．后序遍历森林和中序遍历与该森林对应的二叉树，其结果不同。

【分析】后序遍历森林和中序遍历与该森林对应的二叉树结果相同。

【答案】错误

7-2-3 填空题

1．树在计算机内的表示方式有：_____、_____、_____。

【答案】双亲表示法；孩子表示法；孩子兄弟表示法

2．设 F 是由 T_1、T_2、T_3 三棵树组成的森林，与 F 对应的二叉树为 B。已知 T_1、T_2、T_3 的结点数分别为 n_1、n_2 和 n_3，则二叉树 B 的左子树中有_____个结点，二叉树 B 的右子树中有_____ 个结点。

【分析】根据森林转换为二叉树的方法知，根和左子树为森林的第一棵树，而其余的树都在根的右子树上。

【答案】n_1-1；n_2+n_3

3．树和二叉树的主要差别是：_____、_____、_____。

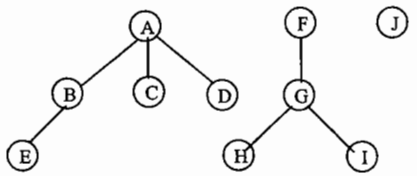
【答案】树的叶结点数至少为 1，而二叉树的结点数可以为 0；树中结点最大度数没有限制，而二叉树结点最大度数为 2；树的结点无左、右之分，而二叉树结点有左、右之分。

4．将树转换成二叉树的目的是：_____。

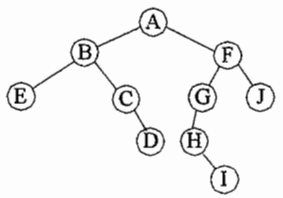
【答案】可采用二叉树的存储结构，并利用二叉树有关算法解决树的有关问题。

7-2-4 应用题

1．画出下图所示的森林经转换后所对应的二叉树，并指出在二叉链表中某结点所对应的森林中结点为叶子结点的条件。



【解答】



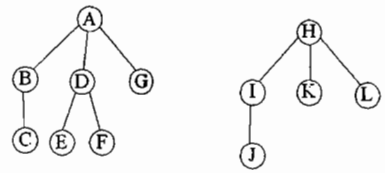
在二叉链表中某结点所对应的森林中结点为叶子结点的条件是该结点在森林中既没有孩子也没有右兄弟。

2．画出和下列已知序列对应的森林 F：

森林的先序访问序列为：ABCDEFGHIJKL；森林的中序访问序列为：CBEFDGAJIKLH。

【解答】方法是先根据森林的先序序列和中序序列画出对应的二叉树，然后将二叉树转换成森林。

该题目最后转换的森林为：



7-2-5 算法设计题

1．对以孩子－兄弟链表表示的树编写计算树的深度的算法。

【分析】

采用递归算法实现。

$$\text{树的深度} = \begin{cases} 0 & \text{若树为空} \\ \max(\text{第一棵子树的深度}+1,\ \text{兄弟子树的深度}) & \text{若树非空} \end{cases}$$

存储表示定义为：

```
typedef  struct  TreeNode
{ elemtype  data;
  struct   TreeNode   *child  , *nextsibling ;
}NodeTtpe , *CSTree;                    // 孩子－兄弟链表类型定义
```

【算法】

```
int high(CSTree  t )
{ if ( t= =NULL )    return ( 0 ) ;       // 若树为空，返回 0
      else   {h1=high (t->child ) ;       // h1 为 t 的第一棵子树的深度
          h2=high(t->nextsibling );       // h2 为 t 的兄弟子树的深度
               return(max(h1+1,h2));
          }
}
```

2．对以双亲链表表示的树编写计算树的深度的算法。

【分析】

从每一个结点开始，从下向上搜索至根结点，记录结点的层次数，其中的最大值就是树的深度。存储表示定义为：

```
typedef  struct
{ elemtype  data;
   int  parent ;
}NodeType;                  // 双亲链表的结点类型
```

【算法】

```
int high (NodeType   t[ ],  int  n)
```

```
{  // 求有 n 个结点的树 t 的深度
   maxh=0;
   for (i=0 ;i<n ;i++)
   {   if (t[i].parent= = -1 ) h=1;
               else {     s=i; h=1;
                       while (t[s].parent != -1 )
                       { s=t[s].parent ;
                         h++;
                       }
           }
           if  (h>maxh)    maxh=h;
     }
   return(maxh) ;
   }
```

7-3 课后习题选解

1. 一棵度为 2 的树与一棵二叉树有何区别？树与二叉树之间有何区别？

① 二叉树是有序树，度为 2 的树是无序树，二叉树的度不一定是 2。

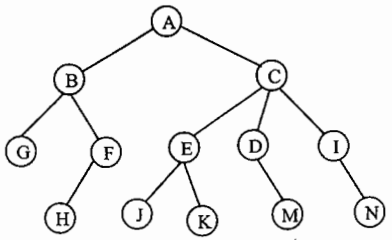
② 二叉树是有序树，每个结点最多有两棵子树，树是无序树，且每个结点可以有多棵子树。

2. 对于如图所示的树，试给出：

（1）双亲数组表示法示意图；

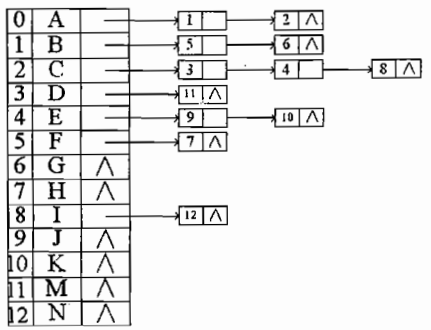
（2）孩子链表表示法示意图；

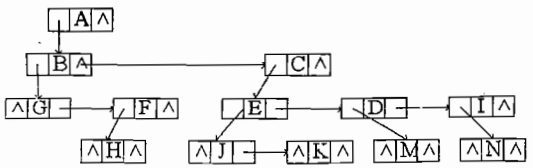
（3）孩子兄弟链表表示法示意图。



（题 2 图）

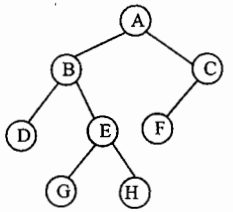
0	A	-1
1	B	0
2	C	0
3	D	2
4	E	2
5	F	1
6	G	1
7	H	5
8	I	2
9	J	4
10	K	4
11	M	3
12	N	8





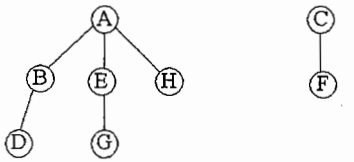
3. 画出下图所示的森林经转换后所对应的二叉树，并指出在二叉链表中某结点所对应的森林中结点为叶子结点的条件。

（参考典型例题解析中的应用题 1）



（题 4 图）

4. 将如图所示的二叉树转换成相应的森林。

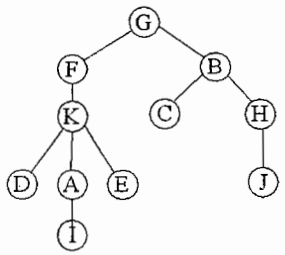


5. 在具有 n（n>1）个结点的各棵树中，其中深度最小的那棵树的深度是多少？它共有多少叶子和非叶子结点？其中深度最大的那棵树的深度是多少？它共有多少叶子和非叶子结点？

2；n-1；1；n；1；n-1

6. 画出和下列已知序列对应的树 T：

树的先根访问序列为：GFKDAIEBCHJ；树的后根访问序列为：DIAEKFCJHBG。



7. 画出和下列已知序列对应的森林 F：

森林的先序访问序列为：ABCDEFGHIJKL；森林的中序访问序列为：CBEFDGAJIKLH。

（参考典型例题解析中的应用题 2）

8. 对以孩子－兄弟链表表示的树编写计算树的深度的算法。

（参考典型例题解析中的算法设计题 1）

9. 对以孩子链表表示的树编写计算树的深度的算法。

【提示】采用递归算法实现。

$$\text{树的深度}=\begin{cases}1 & \text{若根结点没有子树}\\ \max(\text{所有子树的深度})+1 & \text{若根结点有子树}\end{cases}$$

```
#define MAXNODE <树中结点的最大个数>
typedef struct ChildNode
{int childcode;
struct ChildNode *nextchild;
} ChildNode;                                    // 孩子链表的表结点
typedef struct
{DataType data;
struct ChildNode *firstchild;
}NodeType;                                      // 孩子链表的头结点
int high(NodeType t[MAXNODE],int j)
{if(t[j].firstchild==NULL) return(1);           // 若根结点没有子树
    else                                        // 若根结点有子树
        {p=t[i].firstchild;
        max=high(t,p->data);
        p=p->nextchild;
        while(p)
         {h=high(t,p->data);
          if(h>max) max=h;
          p=p->nextchild;
         }
       return(max+1);
       }
}
```

10. 对以双亲链表表示的树编写计算树的深度的算法。

（参考典型例题解析中的算法设计题 2）

第 8 章　图

图状结构是指任意两结点都可能相关联的一种结构，它是数据结构的四类基本逻辑结构中最为复杂的一类。本章通过对图的相关概念、图的存储表示、图的遍历方法及图上应用问题的介绍，要求掌握图的存储表示方法，学会应用图的遍历解决一些简单应用问题，并对图上一些典型应用的解决方法、步骤及时空效率有所了解。

重点提示：

- 图的有关概念
- 图的存储表示（邻接矩阵、邻接表）
- 图的遍历（遍历算法及其简单应用）
- 图的应用（最小生成树、最短路径、拓扑排序和关键路径基本思想和时空性能）

8-1　重点难点指导

8-1-1　相关术语

（1）无向图

要点：图中任意两顶点构成的偶对(v_i, v_j)E 是无序的，即顶点之间连线无方向。

（2）有向图

要点：图中任意两顶点构成的偶对(v_i, v_j)E 是有序的，即顶点之间连线有方向。

（3）顶点、边、弧、弧头、弧尾

要点：图中，数据元素 v_i 称为顶点（vertex）；

(v_i, v_j)或$<v_i, v_j>$表示在顶点 v_i 和顶点 v_j 之间有一条直接连线；

在无向图中，称这条连线为边；在有向图中，称这条连线为弧；

边用顶点的无序偶对(v_i, v_j)来表示，称顶点 v_i 和顶点 v_j 互为邻接点，边(v_i, v_j)依附于顶点 v_i 与顶点 v_j；

弧用顶点的有序偶对$<v_i, v_j>$来表示，有序偶对的第一个结点 v_i 被称为始点（或弧尾），在图中就是不带箭头的一端；有序偶对的第二个结点 v_j 被称为终点（或弧头），在图中就是带箭头的一端。

（4）无向完全图

要点：无向图中任意两顶点都有一条直接边相连接；

在一个含有 n 个顶点的无向完全图中，有 n(n-1)/2 条边。

（5）有向完全图

要点：有向图中任意两顶点之间都有方向互为相反的两条弧相连接；

在一个含有 n 个顶点的有向完全图中，有 n(n-1)条边。

（6）稠密图、稀疏图

要点：若一个图接近完全图，称为稠密图；

而称边数很少的图为稀疏图。

（7）顶点的度、入度、出度。

要点：顶点的度（degree）是指依附于某顶点 v 的边数，通常记为 TD (v)；

在有向图中，要区别顶点的入度与出度的概念；

顶点 v 的入度是指以顶点 v 为终点的弧的数目，记为 ID (v)；

顶点 v 的出度是指以顶点 v 为始点的弧的数目，记为 OD (v)；

有 TD (v)=ID (v)＋OD (v)。

具有 n 个顶点、e 条边的图，顶点的度数与 n 和 e 满足如下关系：

$$e^n = (TD(v_i))/2$$

（8）边的权、网图。

要点：与边有关的数据信息称为权（weight）；

边上带权的图称为网图或网络（network）；

有无向网图和有向网图之分。

（9）路径、路径长度

要点：顶点 v_p 到顶点 v_q 之间的路径（path）是指顶点序列 $v_p,v_{i1},v_{i2}, \ldots, v_{im},v_q$；

其中，(v_p,v_{i1})，(v_{i1},v_{i2})，…，(v_{im}, v_q)分别为图中的边；

路径上边的数目称为路径长度。

（10）回路、简单路径、简单回路

要点：路径 $v_p,v_{i1},v_{i2}, \ldots, v_{im},v_q$，若 v_p 与 v_q 是一个点，则该路径为回路或者环（cycle）；

序列中顶点不重复出现的路径称为简单路径；

除第一个顶点与最后一个顶点外，其他点不重复出现的回路称为简单回路。

（11）子图

要点：对于图 G=(V,E)，其中 V 为顶点的集合，E 为边的集合，$G'=(V',E')$，若 $V'\subseteq V$，$E'\subseteq E$，则 G' 是 G 的一个子图。

（12）连通的、连通图、连通分量

要点：在无向图中；

从一个顶点 v_i 到另一个顶点 v_j（$i\neq j$）有路径，则顶点 v_i 和 v_j 是连通的；

图中任意两顶点都是连通的，则称该图是连通图；

无向图的极大连通子图（添上任意一点均为不连通的子图）称为连通分量。

（13）强连通图、强连通分量

要点：在有向图中；

若图中任意一对顶点 v_i 和 v_j（$i\neq j$）均有从一个顶点 v_i 到另一个顶点 v_j 的路径，也有从 v_j 到 v_i 的路径，则称该有向图是强连通图；

有向图的极大强连通子图（添上任意一点均没有强连通性）称为强连通分量。

（14）生成树

要点：包含连通图 G 全部 n 个顶点的一个极小连通子图（去掉任意一边均不连通）；

必定包含且仅包含 G 的 n-1 条边；

添上任意一条图 G 中的边将产生回路；

生成树中减少任意一条边，则必然成为非连通的。

（15）生成森林

要点：在非连通图中；

由每个连通分量都可得到一个极小连通子图，即一棵生成树；

这些连通分量的生成树就组成了一个非连通图的生成森林。

（16）自由树

要点：无回路的连通图；

未确定哪一个结点是根。

8-1-2　图的基本操作

（1）CreatGraph(G)输入图 G 的顶点和边，建立图 G 的存储。

（2）DestroyGraph(G)释放图 G 占用的存储空间。

（3）GetVex(G,v)在图 G 中找到顶点 v，并返回顶点 v 的相关信息。

（4）PutVex(G,v,value)在图 G 中找到顶点 v，并将 value 值赋给顶点 v。

（5）InsertVex(G,v)在图 G 中增添新顶点 v。

（6）DeleteVex(G,v)在图 G 中，删除顶点 v 以及所有和顶点 v 相关联的边或弧。

（7）InsertArc(G,v,w)在图 G 中增添一条从顶点 v 到顶点 w 的边或弧。

（8）DeleteArc(G,v,w)在图 G 中删除一条从顶点 v 到顶点 w 的边或弧。

（9）DFSTraverse(G,v)在图 G 中，从顶点 v 出发深度优先遍历图 G。

（10）BFSTtaverse(G,v)在图 G 中，从顶点 v 出发广度优先遍历图 G。

（11）LocateVex(G,u)在图 G 中找到顶点 u，返回该顶点在图中的位置。

（12）FirstAdjVex(G,v)在图 G 中，返回 v 的第一个邻接点。若顶点在 G 中没有邻接顶点，则返回“空”。

（13）NextAdjVex(G,v,w)在图 G 中，返回 v 的（相对于 w 的）下一个邻接顶点。若 w 是 v 的最后一个邻接点，则返回“空”。

8-1-3　图的存储表示

1. 邻接矩阵表示

定义：

邻接矩阵（Adjacency Matrix）是表示顶点之间相邻关系的矩阵。设 G=（V，E），V 为顶点的集合，E 为边的集合，若图 G 具有 n 个顶点的图，则 G 的邻接矩阵是具有如下性质的 n 阶方阵：

$$A[i,j]=\begin{cases}1 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 E(G)中的边}\\ 0 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 E(G)中的边}\end{cases}$$

若 G 是网图，则邻接矩阵可定义为：

$$A[i,j]=\begin{cases}w_{ij} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 E(G)中的边}\\ 0\text{或}\infty & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 E(G)中的边}\end{cases}$$

其中，w_{ij} 表示边(v_i,v_j)或<vi,vj>上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

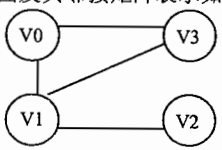
（1）存储类型定义

```
#define MaxVertexNum 100              // 最大顶点数设为 100
typedef char VertexType;              // 顶点类型设为字符型
typedef int EdgeType;                 // 边的权值设为整型
```

```
typedef struct {
   VertexType vexs[MaxVertexNum];       // 顶点表
   EdeType edges[MaxVertexNum][MaxVertexNum]; // 邻接矩阵，即边表
   int n,e;                             // 顶点数和边数
  }Mgragh;                              // Mgragh 是以邻接矩阵存储的图类型
```

（2）举例

一个无向图及其邻接矩阵表示如图 8-1 所示。



$$A=\begin{pmatrix}0&1&0&1\\1&0&1&1\\0&1&0&0\\1&1&0&0\end{pmatrix}$$

图 8-1　一个无向图的邻接矩阵表示

（3）建立算法思想

若定义 Mgragh *G，则建立图 G 邻接矩阵存储的算法步骤为：

① 输入顶点数 G->n;

② 输入边数 G->e;

③ 初始邻接矩阵 G->edges[][]=0;

④ for(k=0;k<G->n;k++) 输入第 k 个顶点值 G->vexs[k];

⑤ for (k=0;k<G->e;k++)

{输入第 k 条边的两顶点序号 i 和 j；置邻接矩阵 G->edges[i][j]=1;

若为无向图，还需置 G->edges[j][i]=1;}

说明：若为网图，可在第⑤步的循环中输入边的权值赋给 G->edges[i][j]。

2．邻接表表示

（1）定义

邻接表表示法类似于树的孩子链表表示法。就是对于图 G 中的每个顶点 v_i，将所有邻接于 v_i 的顶点 v_j 连成一个单链表，这个单链表就称为顶点 v_i 的邻接表（AdjacencyList）。在邻接表表示中有两种结点结构，如图 8-2 所示。

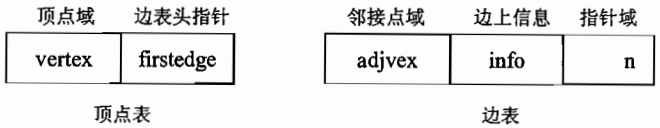


图 8-2　邻接矩阵表示的结点结构

一种是顶点表的结点结构，它由顶点域（vertex）和指向第一条邻接边的指针域（firstedge）构成，另一种是边表（即邻接表）结点，它由邻接点域（adjvex）和指向下一条邻接边的指针域（next）构成。

（2）存储类型定义

```
#define MaxVerNum 100              // 最大顶点数为 100
typedef struct node{               // 边表结点
     int adjvex;                   // 邻接点域
     struct node  * next;          // 指向下一个邻接点的指针域
     // 若要表示边上权值，则应增加一个数据域，如 info
```



```
}EdgeNode;
typedef struct vnode{              // 顶点表结点
     VertexType vertex;            // 顶点域
     EdgeNode  * firstedge;        // 边表头指针
}VertexNode;
typedef VertexNode AdjList[MaxVertexNum];  // AdjList 是邻接表类型
typedef struct{
    AdjList adjlist;               // 邻接表
    int n,e;                       // 顶点数和边数
}ALGraph;                          // ALGraph 是以邻接表方式存储的图类型
```

（3）举例

图 8-1 中无向图对应的邻接表表示如图 8-3 所示。其中定义 ALGraph *G。

（4）建立算法

若定义 ALGraph *G，则建立图 G 邻接表存储的算法步骤为：

① 输入顶点数 G->n;

② 输入边数 G->e;

③ for(k=0;k<G->n;k++)

{输入第 k 个顶点值 G->adjlist[k].vertex;

初始边表指针 G->adjlist[k].firstedge=NULL;

}

④ for(k=0;k<G->e;k++)

{ 输入第 k 条边的两顶点序号 i 和 j;

申请一个边表结点 s;

s->adjvex=j;

将 s 插入到顶点 i 的边链表中;

若为无向图，则再申请一个边表结点 s;

{s->adjvex=i;

将 s 插入到顶点 j 的边链表中；}

}

说明：若为网图，可在第④步的循环中输入边的权值，赋给 s->info。

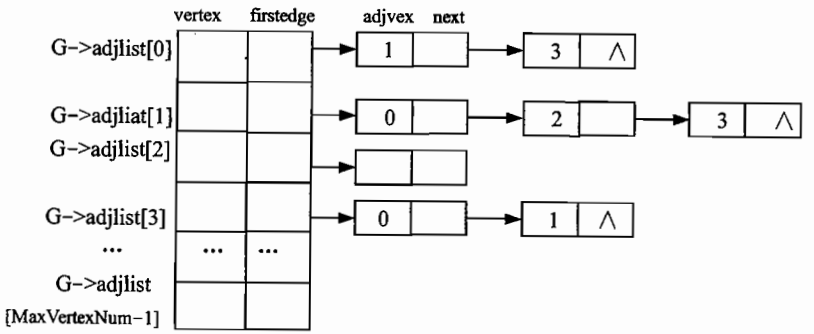


图 8-3　图的邻接表表示

8-1-4 图的遍历

1. 深度优先遍历

（1）方法

图的深度优先遍历（简称 DFS）类似于树的先序遍历，是树的先序遍历的推广。

假设初始状态是图中所有顶点未曾被访问，则深度优先搜索可从图中某个顶点 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作为起始点，重复上述过程，直至图中所有顶点都被访问到为止。

（2）举例

对无向图 8-1，从 V_0 开始进行深度优先遍历的结点序列为 $V_0V_1V_2V_3$ 或 $V_0V_1V_3V_2$ 或 $V_0V_3V_1V_2$。

（3）算法思想

① 设某顶点 v_i 为出发点，首先访问出发点 v_i，并把 v_i 标记为“已访问过”；

② 从 v_i 出发，依次搜索 v_i 的每一个邻接点 v_j；

③ 若 v_j 未访问过，以 v_j 为新的出发点重复①步骤（即出现递归调用）；若 v_j 已访问过，则继续搜索 v_i 的下一个邻接点（即重复②步骤），直到 v_i 的所有邻接点都被访问过。

2. 广度优先遍历

（1）方法

广度优先遍历（简称 BFS）类似于树的按层次遍历的过程。

假设从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作为起始点，重复上述过程，直至图中所有顶点都被访问到为止。

广度优先遍历图的过程也可理解为以 v 为起始点，由近至远，依次访问和 v 有路径相通且路径长度为 1、2、…的顶点。

（2）举例

对无向图 8-1，从 V_0 开始进行广度优先遍历的结点序列为 $V_0V_1V_3V_2$ 或 $V_0V_3V_1V_2$。

（3）算法思想

① 首先访问出发点 v_i；

② 将 V_i 标记为“已访问过”，并让它加入队列；

③ 出队列得到当前队头元素 v_j；

④ 依次搜索 v_j 的每一个邻接点 v_k；

⑤ 若 v_k 未访问过，重复第①步；若 v_k 已访问过，重复第④步，搜索下一邻接点，直到 v_j 所有邻接点都被访问过；

⑥重复第③步，直到队列为空。

8-1-5 图的应用

1. 最小生成树

（1）概念

要点：在连通网络的所有生成树中，各边权值总和最小的生成树；

连通网络的最小生成树不一定是惟一的，但代价是惟一的。

（2）普里姆（Prim）算法思想

基本思想：初始时，最小生成树集中仅含一个任选的结点；

选取距离最小生成树集最近的点（即距生成树集最短边所关联的点）；

将该点和其与生成树中的点所关联的边，添入最小生成树中；

直至将所有结点添入为止。

算法步骤：

设 S 为最小生成树点的集合，T 为最小生成树边的集合。

① 初始 T 为空集，任选一点放入 S 中；

② 计算每个不在 S 中的点距 S 中点的最短距离（即距 S 中点的最短边）；

③ 在所有不在 S 中的点距 S 的最短距离中选择距离值最小的一个；

④ 将距离最小的这条边添入集合 T 中，该边关联的不在 S 中的点添入到 S 中；

⑤ 若图中所有点均已添入到 S 中则结束，否则转到②。

克鲁斯卡尔（Kruskal）算法思想

基本思想：在图中依次选取 n－1 条边，连接图的 n 个结点；

每次选取不会产生回路的权值最小的边。

算法步骤：

① 将所有的结点放入最小生成树中；

② 将边的权值按由小到大排序；

③ 按权值排序的次序选取边进行判定，判定将其添入生成树中是否产生回路；

若产生回路，则不添入；否则，将其添入；

（说明：判定是否产生回路的方法是看添入的边所关联的两点，是在同一棵树上，还是在两棵不同的树上，若为前者，则有回路；为后者则无回路。）

④ 直至添入 n－1 条边为止。

2. 最短路径

（1）问题定义：

① 单源最短路径问题：已知有向带权图（简称有向网）G=(V,E)，希望找出从某个源点 s∈V 到 V 中其余各顶点的最短路径。而其他的最短路径问题均可用单源最短路径算法予以解决。

② 单目标最短路径问题：找出图中每一顶点 v 到某指定顶点 u 的最短路径。只需将图中每条边反向，就可将这一问题变为单源最短路径问题，单目标 u 变为单源点 u。

③ 单顶点对间最短路径问题：对于某对顶点 u 和 v，找出从 u 到 v 的一条最短路径。显然，若解决了以 u 为源点的单源最短路径问题，则上述问题也迎刃而解。而且从数量级来说，两问题的时间复杂度相同。

④ 每一对顶点对间最短路径问题：对图中每对顶点 u 和 v，找出 u 和 v 的最短路径问题。这一问题可用每个顶点作为源点调用一次单源最短路径问题算法予以解决。

（2）解决单源点最短路径问题的迪杰斯特拉（Dijkstra）算法思想

设 S 为最短路径确定的顶点集，V-S 是最短路径尚未确定的顶点集。D[v]为 u 到 v 的最短路径长度。

① 初始时，将源点 u 放入 S 中；

② 初始 D[v]，v∈V-S，D[v]=w<u,v> ；

③ 选取点 k 使 D[k]=min{D[v],v∈V-S }；

④ 将 k 添入 S；

⑤ 若 V－S 不定，重新计算 D[v]值，其中 v∈V-S

```
if(D[v]>D[k]+w<k,v>)
  { D[v]=D[k]+w<k,v>;
    并修改源点到 v 的路径上的点为源点到 k 的路径加上点 v};
```

⑥ 若 V－S 为空，则结束，否则重复步骤③。

说明：教材中将 S 称为红点集，将 V－S 称为蓝点集。

（3）解决每一对顶点对间的最短路径问题的弗洛伊德（Floyd）算法思想

应用 Floyd 算法求解每一对顶点对间的最短路径问题依据的是以下递推关系：

$$\begin{cases} D^{(-1)}[i][j]=edges[i][j] \\ D^{(k)}[i][j]=Min\{D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\} \quad 0\leqslant k\leqslant n-1 \end{cases}$$

其中二维数组 edges 存放的是带权图的邻接矩阵的值，$D^{(k)}[i][j]$是从 v_i 到 v_j 的中间顶点的个数不大于 k 的最短路径的长度，因此，$D^{(n-1)}[i][j]$是从 v_i 到 v_j 最短路径的长度。

此外，在算法中还需另外设置一个二维数组用于记录最短路径上的结点，比如可用 P[v][w]存放从点 v 到点 w 的最短路径上点 w 的前驱结点的序号。

3．拓扑排序

（1）问题定义

对一有向图，如果从 v_i 到 v_j 存在一条路径，且在由图中所有顶点构成的线性序列中，v_i 总在 v_j 之前，那么这样的线性序列就被称为拓扑序列。构造一个有向图的拓扑序列的过程称为拓扑排序。

（2）有向图有拓扑排序的条件

① 初始时有向图中必须至少有一个入度为零的顶点；

② 有向图中不存在回路。

（3）算法思想

① 扫描顶点表，将入度为 0 的顶点入栈；

② 当栈非空时，当前栈顶元素 v_j 出栈，并输出。检查 v_j 的出边表，将每条出边<v_j,v_k>的终点 v_k 的入度减 1。若 v_k 的入度变为 0，则让 v_k 入栈；

③ 重复第②步，直到栈为空；

④ 检查输出顶点数，若小于 n，表示“图中有环，无拓扑序列”；否则算法正常结束。

说明：算法中用的是栈，也可改为队列，只是得到的是两种不同的拓扑输出序列。

4．关键路径

（1）问题定义

在带权图中，以顶点表示事件，以有向边表示活动，边上的权值表示活动的开销（如该活动持续的时间），则此带权的有向图称为 AOE 网。在 AOE 网中从源点到汇点的最大长度的路径称为关键路径，关键路径上的活动成为关键活动。关键路径问题就是要找出所给 AOE 网的关键路径和关键活动。

在实际应用中，常常用 AOE 网来表示一项工程，此时的 AOE 网必须是有向无环图。通过确定关键路径，得到完成整个工程的最短时间，并找出哪些活动是影响工程进度的关键。

（2）计算关键路径用到的参量及其计算方法

① 事件的最早发生时间 ve[k]

$$\begin{cases} ve[1]=0 \\ ve[k]=Max\{ve[j]+dut(<v_j, v_k>)\} \quad <v_j, v_k>\in p[k] \end{cases}$$

其中，p[k]表示所有到达 v_k 的有向边的集合；dut(<v_j,v_k>)为有向边<v_j,v_k>上的权值。

② 事件的最迟发生时间 vl[k]

$$\begin{cases} vl[n]=ve[n] \\ vl[k]=Min\{vl[j]-dut(<v_k- v_j>)\} \quad <v_k, v_j>\in s[k] \end{cases}$$

其中，s[k]为所有从 v_k 发出的有向边的集合。

③ 活动 a_i 的最早开始时间 e[i]

$$e[i]=ve[k]$$

④ 活动 a_i 的最晚开始时间 l[i]

$$l[i]=vl[j]-dut(<v_k, v_j>)$$

（3）关键活动及关键路径的判定

根据每个活动的最早开始时间 e[i]和最晚开始时间 l[i]就可判定该活动是否为关键活动，也就是那些 l[i]=e[i]的活动就是关键活动，而那些 l[i]>e[i]的活动则不是关键活动，l[i]-e[i]的值为活动的时间余量。关键活动确定之后，关键活动所在的路径就是关键路径。

（4）确定关键活动及关键路径的算法步骤

① 输入 e 条弧<j,k>，建立 AOE 网的存储结构；

② 从源点 v_1 出发，令 ve[0]=0，按拓扑有序求其余各顶点的最早发生时间 ve[i]（1≤i≤n-1）。如果得到的拓扑有序序列中顶点个数小于网中顶点数 n，则说明网中存在环，不能求关键路径，算法终止；否则执行步骤③；

③ 从汇点 v_n 出发，令 vl[n-1]=ve[n-1]，按逆拓扑有序求其余各顶点的最迟发生时间 vl[i]（n-2≥i≥2）；

④ 根据各顶点的 ve 和 vl 值，求每条弧 s 的最早开始时间 e[s]和最迟开始时间 l[s]。若某条弧满足条件 e[s]=l[s]，则为关键活动。

8-2 典型例题解析

8-2-1 判断题

1．求最小生成树的 Prim 算法在边较少、结点较多时效率较高。

【分析】Prim 算法的时间复杂度为 $O(n^2)$，说明其在结点较少时效率较高。

【答案】错误

2．图的最小生成树的形状可能不惟一。

【分析】图的最小生成树的代价是惟一的，但形状可能不惟一。

【答案】正确

3．用邻接矩阵法存储一个图时，在不考虑压缩存储的情况下，所占用的存储空间大小只与图中结点个数有关，而与图的边数无关。

【分析】邻接矩阵的行数和列数与图的顶点数相同，而与图的边数无关。

【答案】正确

4．邻接表法只用于有向图的存储，邻接矩阵对于有向图和无向图的存储都适用。

【分析】邻接表法也可用于存放无向图，只是需要注意每条边被存放了两次。

【答案】错误

5．任何有向网络（AOV 网络）拓扑排序的结果是惟一的。

【分析】拓扑排序不一定惟一，只要满足偏序关系即可。

【答案】错误

6．用邻接矩阵 A 表示图，判定任意两个结点 v_i 和 v_j 之间是否有长度为 m 的路径相连，则只要检查 A^m 的第 i 行第 j 列的元素是否为 0 即可。

【分析】A^m 中的元素 a_{ij} 是从顶点 v_i 到顶点 v_j 的长度为 m 的路径总数。

【答案】正确

7．一个图的广度优先搜索树是惟一的。

【分析】在对图进行广度优先搜索时，由于在同一层上搜索的结点顺序不同，可能存在多棵树。

【答案】错误

8．缩短关键路径上活动的工期一定能够缩短整个工程的工期。

【分析】在用 AOE 网表示的工程工期图中，其从源点到终点的最长路径可能不只一条，因此就可能有不只一条的关键路径。

【答案】错误

9．对于有向图，除了拓扑排序方法外，还可以通过对有向图进行深度优先遍历的方法来判断有向图中是否有环。

【分析】在拓扑排序中，若输出之后余下的顶点均有前驱，则说明该有向图含有回路；对有向图进行深度优先遍历时，若遇到回边，则说明该有向图含有回路。

【答案】正确

10．在 n 个结点的无向图中，若边数大于 n−1，则该图必是连通图。

【分析】该图可能包含多个连通子图，但其本身可以是不连通的。

【答案】错误

8-2-2 选择题

1．在 N 条边的无向图的邻接表的存储中，边表的个数有（　）。

A．N　　B．2N　　C．N/2　　D．N×N

【分析】采用邻接表存放无向图时，每条边在边表中重复存放一次。

【答案】B

2．在 N 条边的无向图的邻接多重表的存储中，边表的个数有（　）。

A．N　　B．2N　　C．N/2　　D．N×N

【分析】采用邻接多重表存放无向图时，每条边用一个边表存放。

【答案】A

3．用 DFS 遍历一个有向无环图，并在 DFS 算法退栈返回时打印出相应顶点，则输出的顶点序列是（　）。

A．逆拓扑有序的　　B．拓扑有序的　　C．无序的　　D．DFS 遍历序列

【分析】按形成的深度优先搜索树由树叶到树根输出顶点序列，即为逆拓扑有序。

【答案】A

4．有拓扑排序的图一定是（　）。

A．有环图　　B．无向图　　C．强连通图　　D．有向无环图

【分析】只有有向无环图才有拓扑排序。

【答案】D

5．设有向图有 n 个顶点和 e 条边，进行拓扑排序时，总的计算时间为（　）。

A．O (n$\log_2$e)　　B．O (en)　　C．O (e$\log_2$n)　　D．O (n+e)

【分析】拓扑排序的时间复杂度为 O (n+e)。

【答案】D

6．对于含有 n 个顶点 e 条边的无向连通图，利用 Kruskal 算法生成最小代价生成树其时间复杂度为（　）。

A．O (e$\log_2$e)　　B．O (en)　　C．O (e$\log_2$n)　　D．O (n$\log_2$n)

【分析】采用 Kruskal 算法生成最小代价生成树的时间复杂度与有向图的边数有关，即为 O (e$\log_2$e)。

【答案】A

7．关键路径是事件结点网络中（　）。

A．从源点到汇点的最长路径　　B．从源点到汇点的最短路径

C．最长的回路　　D．最短的回路

【分析】关键路径是事件结点网络中从源点到汇点的最长路径。

【答案】A

8．n 个顶点的强连通图至少有（①）条边，这样的有向图的形状是（②）。

① A．n　　B．n+1　　C．n−1　　D．n（n−1）

② A．无回路　　B．有回路　　C．环状　　D．树状

【分析】根据强连通图的定义，n 个顶点的强连通图至少有 n 条边，有向图的形状是环状。

【答案】①A；②C

8-2-3 填空题

1．具有 n 个顶点的无向完全图，边的总数为_____条；而具有 n 个顶点的有向完全图中，边的总数为_____条。

【分析】根据无向完全图和有向完全图的定义可以确定。

【答案】n(n−1)/2；n(n−1)

2．一个图的生成树的顶点是图的_____顶点。

【分析】一个含有 n 个顶点的连通图的生成树应含有 n 个顶点和 n−1 条边。

【答案】所有

3．一个连通图的生成树是一个_____连通子图，n 个顶点的生成树有_____条边。

【分析】n 个结点的连通图的生成树是其一个无回路的连通子图，含有 n−1 条边。

【答案】无回路的；n-1

4．邻接表和十字链表适合于存储_____图，邻接多重表适合于存储_____图。

【分析】邻接表和十字链表适合于存储有向图，邻接多重表适合于存储无向图。

【答案】有向；无向

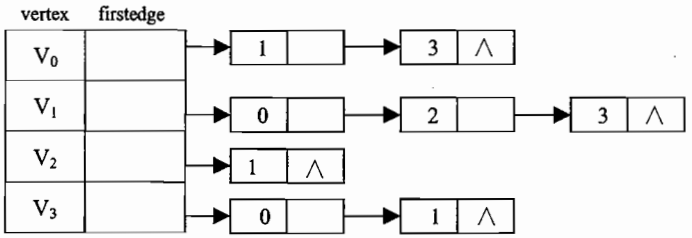
5．DFS 和 BFS 遍历分别采用_____ 和 _____的数据结构来暂存顶点，当要求连通图的生成树的高度最小，应采用的遍历方法是_____ 。

【分析】DFS 采用栈来暂存顶点；BFS 采用队列来暂存顶点；当要求连通图的生成树的高度最小，应采用的遍历方法是 BFS。

【答案】栈；队列；BFS

8-2-4 应用题

1．图的邻接表如下图所示：



题 1 图

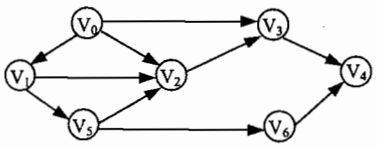
（1）写出从顶点 V_0 出发进行深度优先搜索，经历的结点顺序。

（2）写出从顶点 V_0 出发进行广度优先搜索，经历的结点顺序。

【解答】

（1）V_0，V_1，V_2，V_3　　（2）V_0，V_1，V_3，V_2

2．写出下图所示的所有拓扑序列_____。

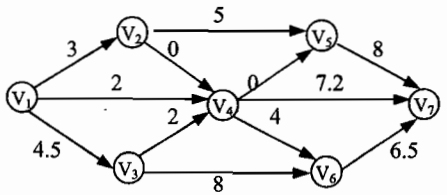


题 2 图

【解答】

拓扑序列为：$V_0V_1V_5V_2V_3V_6V_4$、$V_0V_1V_5V_2V_6V_3V_4$ 和 $V_0V_1V_5V_6V_2V_3V_4$

3．写出在如下图所示的网络计划图中关键路径及全部计划完成的最短时间。

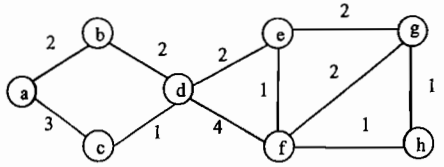


题 3 图

【解答】

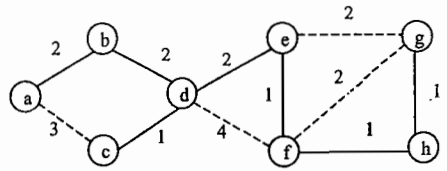
关键路径为：$V_1V_3V_6V_7$；计划完成的最短时间是 19。

4．对下面连通图，请分别用 Prim 和 Kruskal 算法构造其最小生成树。



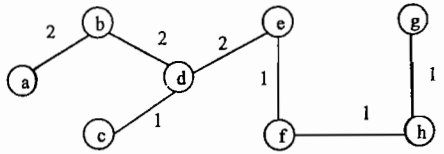
【解答】

（1）Pirm 算法构造的生成树。



说明：实线为最小生成树的边，虚线为在构造过程中不被选择的边。

（2）Kruskal 算法构造的生成树。



8-2-5 算法设计题

1．写一算法求有向图的所有根（若存在），分析算法的时间复杂度。

【分析】

以有向图的每一个结点为源点对图进行搜索，若能搜索到每个结点。则该结点为根。否则不是。

【算法】

```
void searchroot (ALGraph *G)
{ int i;
  for(i=0;i<G->n;i++)
    {for(j=0;j<G->n;j++)
     visited[j]=false;                    // 标志向量初始化
     DFS(G,i);                            // 以 Vi 为源点开始 DFS 搜索
    }
 }
void DFS(ALGraph *G,int i)
{ int count=0;
  EdgeNode *p;
  visited[i]=true;
  count ++;
```

```
p=G->adjlist[i]->firstedge;
while(p)
{if(!visited[p->adjvex])  DFS(G,p->adjvex);
 p=p->next;
}
if (count==G->n)                    // 该结点是根结点
 cout<<G->adjlist[i]->vertex;
}
```

2. 假设以邻接矩阵作为图的存储结构，编写算法判断在给定的有向图中是否存在一个简单有向回路，若存在，则以顶点序列的方式输出该回路（找到一条路即可）。（注：图中不存在顶点到自己的弧。）

【分析】

给定一个图判定是否有回路，可以有以下几种方法：

（1）利用拓扑排序进行判定。即在拓扑排序输出结束后，所余下的顶点均有前驱，则说明只得到了部分顶点的拓扑有序序列，AOV 网中存在有向回路。

（2）设 G 是 n 个顶点的无向连通图，若 G 的边数 e 大于或等于 n，则 G 中一定有回路存在。因此，只要计算出 G 中的边数，就可判定图 G 中是否存在回路。

（3）设 G 是 n 个顶点的无向连通图，若 G 的每个顶点的度大于或等于 2，则图中一定存在回路。

（4）利用深度优先遍历算法可以判定图 G 中是否存在回路。对无向图来说，若深度优先遍历过程中遇到了回边，则必存在环；对有向图来说，这条回边可能是指向深度优先森林中另一棵生成树上顶点的弧；但是，如果从有向图上的某个顶点 v 出发 进行深度优先遍历，若在 DFS(v)结束之前出现一条从顶点 u 到顶点 v 的回边，因 u 在生成树上是 v 的子孙，则有向图必存在包含顶点 v 和顶点 u 的环。

这里采用深度优先遍历方法，在遍历中若发现回路则输出回路的顶点序列。

【算法】

```
Display( )
{ int v,I,flag;
  flag=0;
  v=vex[p];                 //取出在遍历深度为 p 时形成回路的顶点赋给 v
  for(i=1;i<=p;i++)
      {if(vex[i]==v)        //在数组 vex 中寻找与顶点 v 相关联的顶点 i，置其标志 flag 为 1
      flag=1;
      if(flag==1)           //输出回路顶点序列
      cout<<vex[i];
    }
}
Dfs(int v)
{ if(found==1)
   return(0);
  p++;                      //记录此次深度优先遍历的结点数
  vex[p]=v;                 //将当前遍历的顶点置为与 v 有关的标志，并记入 vex 数组
  if(visited[v]==1)         //如果顶点 v 已访问过，则出现回路
   {found=1;                //置发现回路标志为真
    Display( );             //调用 Display 函数输出该回路的顶点序列
```



```
    }
else
  {visited[v]=1;            //将顶点 v 置为访问过标志
   for(i=1;i<=n;i++)
     if(g[v,i]>0) Dfs(i); //对与 v 邻接的顶点进行深度优先遍历
    }
p=p-1;                      //清除此次深度优先遍历访问过的顶点标志
visited[v]=0;
}
Find_circle( )
{int v;
 for(v=1;v<=n;v++)
   visited[i]=0;            //初始化图中每个顶点的访问标志
   p=0;
   found=0;
   for(v=1;v<=n;v++)        //对 n 个顶点依次进行深度优先遍历来寻找回路
     Dfs(v);
}
```

3. 以邻接表为存储结构，写一个基于 DFS 遍历策略的算法，求图中通过某顶点 v_k 的简单回路（若存在）。

【分析】

从给定顶点 v_k 出发进行深度优先搜索，在搜索过程中判断当前访问的顶点是否为 v_k。若是，则找到一条回路。否则继续搜索。为此设一个顺序栈 cycle 记录构成回路的顶点序列，把访问顶点的操作改为将当前访问的顶点入栈；相应地，若从某一顶点出发搜索完再回溯，则做退栈操作，同时要求找到的回路的路径应大于 2。另外还设置一个 found，初值为 0，当找到回路后为 1。

【算法】

```
void  dfscycle (ALGraph *G,int   k)
{int i,j,top=0,v=k,found=0,w;
 int visited[100],cycle[100];
 EdgeNode  *p, *stack[100];
 i=1;
cycle[i]=k;                                          // 从 vk 开始搜索
visited[k]==1;
P=G[k]->firstedge;
while(p!=NULL!!top>0)&&!found)
{ while(p!=NULL&&!found)
    if (p->adjvex==k&&i<2)found=1;                   // 找到回路
    else if(visited[p->adjvex]==0) p=p->next;        // 找到下一个邻接点
       else {w=p->adjvex;                            // 记下路径，继续搜索
             visited[w]=1;
             i++;
             cycle[i]=w;
             top++;
             stack[top]=p;
             p=G[w]->firstedge;
            }
```

```
if(!found&&top>0)                          // 沿原路径退回后，另选路径进行搜索
{ p=stack[top];
  top--;
   p=p->next;
   i--;
 }
} // end while
if(found)
 {for(j=1;j<=i;j++)
    cout<<cycle[j];                        // 打印回路的顶点序列
  cout<<v;
 }
else cout<<"设有通过点 Vk 的回路！ ";
}
```

4. 利用拓扑排序算法的思想写一算法判别有向图中是否存在有向环，当有向环存在时，输出构成环的顶点。

【分析】

利用拓扑排序的方法将入度为 0 的结点入栈；每出栈一个结点，将其关联结点的入度减 1；若减 1 后为 0，再入栈；这样反复直到栈空为止；若此时进过栈的结点个数小于 n，则说明该图为有向环；否则不是有向环。

【算法】

```
void Topo_Sort (AlGraph *G)
{// 对以带入度的邻接链表为存储结构的图 G，输出其一种拓扑序列
  int  top = -1;                                  //  栈顶指针初始化
  for (i=0; i<n; i++)                             //  依次将入度为 0 的顶点压入链式栈
    {  if ( G->adjlist[i].indegree = = 0)         // indegree 域为存放结点 i 的入度
         { G->adjlist[i]. indegree = top;
           top = i;
         }
    }
   for (i=0;i<n;i++)
    { if (top= -1)
       {cout<<"The network has a cycle";
        return;
       }
       j=top;
       top=G->adjlist[top]. indegree;             //  从栈中退出一个顶点并输出
       cout<<G->adjlist[j].vertex;
       ptr=G->adjlist[j].firstedge;
       while (ptr!=null)
          { k=ptr->adjvex;
            G->adjlist[k]. indegree--;            // 当前输出顶点邻接点的入度减 1
            if(G->adjlist[k]. indegree = =0)      // 新的入度为 0 的顶点进栈
              {G->adjlist[k]. indegree =top;
                top=k;
              }
            ptr=ptr->next;                        // 找到下一个邻接点
          }
    }
}
```


5. 设计算法，求出无向连通图中距离顶点 v_0 的最短路径长度（最短路径长度以边数为单位计算）为 k 的所有结点，要求尽可能地节省时间。

【分析】

该问题需用广度优先遍历的层次特性来求解，即以 v_0 为起点调用 BFS 算法输出第 k+1 层上的所有顶点。因此，在访问顶点时需要知道层数，而每个顶点的层数是由其前驱决定的（起点除外）。所以，可以从第一个顶点开始，每访问到一个顶点就根据其前驱的层次计算该顶点的层次，并将层数值与顶点编号一起入队、出队。实现时可增加一个队列来保存顶点的层数值，并且将层数的相关操作与对应顶点的操作保持同步，即一起置空、出队和入队。

【算法】

```
Bfs_kLevel(ALGraph G, int v0,int k)
{int i;
 Init_Queue(Q);                                //初始空队列 Q
 Init_Queue(Q1);                               //初始空队列 Q1
 for(i=1;i<n;i++)                              //初始化图的每个顶点的访问标志
   visited[i]=0;
 visited[v0]=1;                                //置起点的访问标志
 level=1;                                      //起点的层次数为 1
In_Queue(Q,v0);
In_Queue(Q1,level);
while((!Empty_Queue(Q))&&(level<k+1))          //当队列 Q 不空且层次数小于 k+1 时
  {v=Out_Queue(Q);
   level=Out_Queue(Q1);
   w=FirstAdjVex(G,v);                         //将 v 的第一个邻接点赋给 w
   while(w!=0)                                 //当邻接点存在时
     {if(visited[w]==0)                        //如果 w 为未访问过的顶点
        {if(level==k)                          //如果 w 的层次数为 k 则输出
           cout<<w;
         visited[w]=1;
         In_Queue(Q,w);                        //w 入队列
         In_Queue(Q1,level+1);                 //w 的层次数入队列
         }
          w=NextAdjVex(G,v,w);                 //寻找 v 的下一个邻接点
      }
   }
}
```

8-3　课后习题选解

1. 对于如图所示的有向图，试给出：

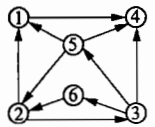
① 每个顶点的入度和出度；

② 邻接矩阵；

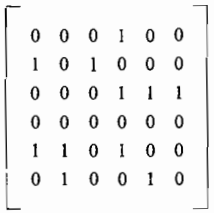
③ 邻接表；

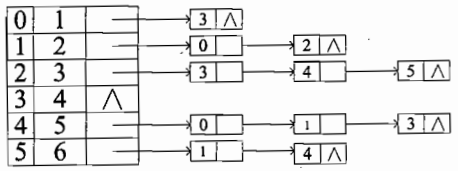
④ 逆邻接表；

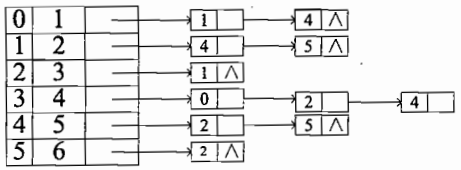
⑤ 强连通分量。

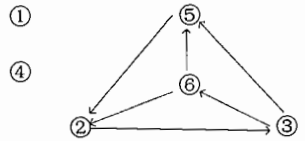


（题 1 图）









2. 设无向图 G 如图所示，试给出：

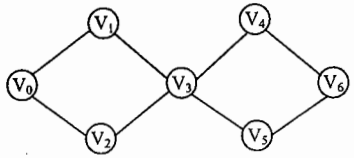
① 该图的邻接矩阵；

② 该图的邻接表；

③ 该图的多重邻接表；

④ 从 A 出发的“深度优先”遍历序列；

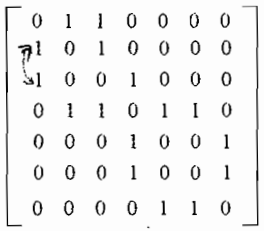
⑤ 从 A 出发的“广度优先”遍历序列。

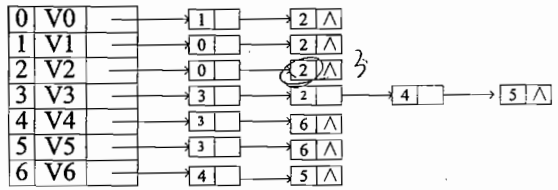


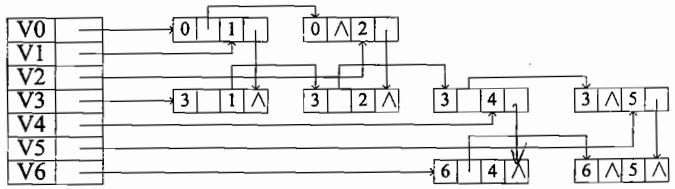
（题 2 图）

深度优先遍历序列：$V_0V_1V_3V_2V_4V_6V_5$

广度优先遍历序列：$V_0V_1V_2V_3V_4V_5V_6$



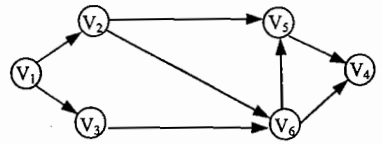




$V_0\ V_1\ V_3\ V_2\ V_4\ V_6\ V_5$

$V_0\ V_1\ V_2\ V_3\ V_4\ V_5\ V_6$

3. 设有向图 G 如图所示，试画出图 G 的十字链表结构，并写出图 G 的两个拓扑序列。



（题 3 图）

$V_1V_2V_3V_6V_5V_4$；$V_1V_3\ V_2V_6V_5V_4$

6. 试以邻接矩阵为存储结构实现图的基本操作：InsertVex (G,v)、InsertArc (G,v,w)、DeleteVex (G,v)和 DeleteArc (G,v,w)。

```
  void InsertVex (MGraph *G,int k) //在邻接矩阵表示的图 G 中插入一个顶点作为第 k 个顶点
  { int i,j;
G->n++;
for(i=G->n;i>k;i--)                    //腾出第 k 个顶点值的位置
   G->vexs[i]=G->vexs[i-1];
G->vexs[k]=getchar( );                 //插入第 k 个顶点的值
for(i=G->n;i>k;i--)
{ for(j=G->n;j>k;j--)                  //第 k 行以后和第 k 列以后的元素后移一行一列
     G->edges[i][j]=G->edges[i-1][j-1];
  G->edges[i][k]=0;                    //将第 k 行以后的 k 列赋 0
  for(j=k-1;j>=0;j--)                  //第 k 行以后第 k 列以前的元素后移一行
         G->edges[i][j]=G->edges[i-1][j];
}
for(j=G->n;j>=0;j--) G->edges[k][j]=0;     //将第 k 行的元素赋 0
for(i=k-1;i>=0;j--)                    //第 k 行以前第 k 列以后的元素后移一列
{ for(j=G->n;j>k;j--)
    G->edges[i][j]=G->edges[i][j-1];
  G->edges[i][k]=0                     //将第 k 行以前的 k 列赋 0
 }
}

  void InsertArc (MGraph *G,int m,int n)
  { G->edges[m][n]=1;G->edges[n]p[m]=1;G->e++;
  }

  void deleMGraph(MGraph *G,int k) //在邻接矩阵表示的图 G 中删除序号为 k 的点
  {
   int i,j;
   G->n--;
   for(i=k;i<=g->n;i++)  G->vexs[i]=G->vexs[i+1];//删除顶点值
   for(i=0;i<k;i++)                    //k 行以前 k 列以后前移一列
  for(j=k;j<=n;j++)  G->edges[i][j]=G->edges[i][j+1];
     for(i=k;i<=G->n;i++)              //k 行以后的元素前移一行
         { for(j=0;j<k;j++)  G->edges[i][j]=G->edges[i+1][j];
           for(j=k;j<=n;j++)           //k 行以后 k 列以后前移一列
                 G->edges[i][j]=G->edges[i][j+1];
           }
  }
void InsertLMGraph(MGraph *G,int m,int n)
```

```
{ G->edges[m][n]=0;
G->edges[n][m]=0;
G->e--;
}
```

7. 试以邻接表为存储结构实现图的基本操作：InsertVex (G,v)、InsertArc (G,v,w)、DeleteVex (G,v)和 DeleteArc (G,v,w)。

```
  void InsertVex(ALGraph *G,int k)   //在邻接表表示的图 G 中插入一个顶点，作为第 k 个顶点
   { int i;
G->n++;
for(i=G->n;i>k;i--)                  //将原图中从 n 到 k 的结点后移一个位置
  { G->adjlist[i].vertex=G->adjlist[i-1].vertex;
    G->adjlist[i].firstedge=G->adjlist[i-1].Firstedge;
    }
  G->adjlist[k].vertex=getchar( );//将插入点放入
  G->adjlist[k].firstedge=NULL;
  }

  void  InsertArc (ALGraph *G,int m,int n)
  { EdgeNode *E;
EdgeNode *sm=(EdgeNode *)malloc(sizeof(EdgeNode));
EdgeNode *sn=(EdgeNode *)malloc(sizeof(EdgeNode));
sm->adjvex=n;                        //将边结点插入到边表首部
sm->next=G->adjlist[m].firstedge;
sn->adjvex=m;
sn->next= G->adjlist[m].firstedge;
G->adjlist[m].firstedge = sm;
G->adjlist[m].firstedge = sn;
G->e++;
  }

  void DeleteVex (ALGraph *G,int k)//在邻接表表示的图 G 中删除序号为 k 的点
  { int i;
G->n--;
   EdgeNode *e,*s,*p,*q;
   for(q=e=G->adjlist[k].firstedge;e;q=e,e=e->next,free(q))
      {s=G->adjlist[e->adjvex].firstedge;
    p=s;
    for(;s;p=s, s=s->next)
     if(s->adjvex==k)
       {if(p!=s) p->next=s->next;
        else (G->adjlist[e->adjvex].firstedge=s->next;)}
        if(s) free(s);
        break;
       }
   }
for(i=k;i<G->n;i++)                  //删除顶点值
   { G->adjlist[i].vertex=G->adjlist[i+1].vextex;
     G->adjlist[i].first[i].firstedge=G->adjlist[i+1].firstedge;
   }
```



```
  }

   void dellALGraph(ALGraph *G,int m,int n)
   {EdgeNode *s, *p;
  s=G->adjlist[m].firstedge;
  p=s;
  for(;s;s=s->next);                 //在与 m 邻接的点中找 n
    { if(s->adjvex= =n)              //若找到邻接点 n，则将该边从边表中脱出
       {if(p!=s) p->next=s->next;
        else G->adjlist[m].firstedge=s->next;
        }
   if(s) free(s);                    //释放要删除的边结点
    }
s=G->adjlist[n].firstedge;p=s;
for(;s;p=s,s=s->next)                //在与 n 邻接的点中找 m
      {if(s->adjvex= =m)             //若找到邻接点 m，则将该边从边表中脱出
           {if(p!=s) p->next=s->next;
            else G->adjlist[n].firstedge=s->next;
           }
    if(s) free(s);                   //释放要删除的边结点
   }
G->e--;
  }
```

10. 试写一算法，由图的邻接链表存储得到图的十字链表存储。

```
#define MaxVerNum 100                // 最大顶点数为 100
typedef struct node{                 // 边表结点
int adjvex;                          // 邻接点域
struct node  *next;                  // 指向下一个邻接点的指针域
                                     // 若要表示边上信息，则应增加一个数据域 info
}EdgeNode;
typedef struct vnode{                // 顶点表结点
VertexType vertex;                   // 顶点域
EdgeNode  * firstedge;               // 边表头指针
}VertexNode;
typedef VertexNode AdjList[MaxVertexNum];
                                     // AdjList 是邻接表类型
typedef struct{
AdjList adjlist;                     // 邻接表
int n,e;                             // 顶点数和边数
}ALGraph;                            // ALGraph 是以邻接表方式存储的图类型

#define  MAX_VERTEX_NUM 20
typedef struct ArcBox {
int tailvex,headvex;                 // 该弧的尾和头顶点的位置
struct  ArcBox  * hlink, *tlink;
                                     // 分别为弧头相同和弧尾相同的弧的链域
InfoType   info;                     // 该弧相关信息的指针
}ArcBox;
typedef struct VexNode {
VertexType vertex:
ArcBox  *fisrin, *firstout;          // 分别指向该顶点第一条入弧和出弧
}VexNode;
```

```
typedef struct {
VexNode xlist[MAX_VERTEX_NUM];
                                    // 表头向量
int  vexnum,arcnum;                 // 有向图的顶点数和弧数
}OLGraph;

void CreatOG(ALGraph *H,OLGraph *G)
                                    // 由邻接表H得到十字链表G
{G->vexnum=H->n;
G->arcnum=H->e;
for(i=0;i< H->n;i++)
  {G->xlist[i].vertex=H->adjlist[i].vertex;
   G->xlist[i].firstin= G->xlist[i].firstout=NULL;
   }                                // 初始化十字链表
for(i=0;i<H->n;i++)
   {p=H->adjlist[i].firstedge;
    while(p)
      {s=new ArcBox;
       j=p->adjvex;
       s->tailvex=i; s->headvex=j;
       s->tlink=G->xlist[i].firstout;
       G->xlist[i].firstout=s;
       s->hlink=G->xlist[j].firstin;
       G->xlist[j].firstin=s;
       P=p->next;
     }                              // 生成十字链表
  }
}
```

11. 写一算法，由依次输入图的顶点数目、边的数目、各顶点的信息和各条边的信息建立无向图的邻接多重表。

```
#define MAX_VERTEX_NUM 20
typedef emnu{ unvisited,visited) VisitIf;
typedef struct EBox{
VisitIf mark:                       // 访问标记
int ivex,jvex;                      // 该边依附的两个顶点的位置
struct EBox ilink, jlink;           // 分别指向依附这两个顶点的下一条边
InfoType  info;                     // 该边信息指针
}EBox;
typedef struct VexBox{
VertexType data;
EBox  fistedge;                     // 指向第一条依附该顶点的边
}VexBox;
typedef struct{
VexBox adjmulist[MAX_VERTEX_NUM];
int vexnum,edgenum;                 // 无向图的当前顶点数和边数
}AMLGraph;

void CreatAMLGraph(AMLGraph *G)
{Cin>>G->vexnum>>G->edgenum;
 for(i=0;i<vexnum;i++)              // 读入顶点集，初始化邻接多重表
  {Cin>>G->adjmulist[i].data;
   G-> adjmulist[i].firstedge=NULL;
  }
    for(k=0;k<G->edgenum;k++)       // 读入边集，生成邻接多重表
```



```
  {Cin>>i>>j;
   s=new EBox;
   Cin>>s->info;
   s->ivex=i; s->jvex=j;
   s->ilink=G->adjmulist[i].firstedge;
   G->adjmulist[i].firstedge=s;
   s->jlink=G->adjmulist[j].firstedge;
   G->adjmulist[j].firstedge=s;
  }
}
```

12. 试写一个算法，判别以邻接表方式存储的有向图中是否存在由顶点 v_i 到顶点 v_j 的路径（i≠j）。假设分别基于下述策略：

① 图的深度优先搜索;

② 图的宽度优先搜索。

【提示】①从顶点 v_i 开始，进行深度优先搜索，在搜索过程中判断是否遇到顶点 v_j，如果遇到，那么存在由顶点 v_i 到顶点 v_j 的路径，否则不存在。②从顶点 v_i 开始，进行广度优先搜索，在搜索过程中判断是否遇到顶点 v_j，如果遇到，那么存在由顶点 v_i 到顶点 v_j 的路径，否则不存在。下面给出基于深度优先搜索的实现算法。

```
int DFSAL(ALGraph *G,int i,int j)  //以 Vi 为出发点对邻接表存储的图 G 进行 DFS 搜索
{ EdgeNode *p;
 visited[i]=1;                     //标记 Vi 已访问
 p=G->adjlist[i].firstedge;        //取 vi 边表的头指针
 while (p)                         //依次搜索 vi 的邻接点 vj，j=p->adjva
  { if (!visited[p->adjvex])       //若 vj 尚未访问，则以 vj 为出发点纵向深搜索
    {if(p->adjvex!=j)   DFSAL(G,p->adjvex,j);
            else    return 1;
    }
    p=p->next;                     //找 vi 的下一个邻接点
 }
 return 0;
}
```

13. 试修改 Prim 算法，使之能在邻接表存储结构上实现求图的最小生成森林，并分析其时间复杂度（森林的存储结构为孩子－兄弟链表）。

【提示】对于非连通的网图，采用普里姆算法进行构造，会产生最小生成森林。因此，设计算法时要考虑到对于非连通图需要对各连通分支分别求最小生成树，那么算法应采用三重循环来实现。

14. 以邻接表作为存储结构实现求从源点到其余各顶点的最短路径的 Dijkstra 算法。

【提示】采用邻接表实现 Dijkstra 算法时，为了表示有向边的信息，邻接表的边表结点除了包含邻接点和指针域，还要增加表示边上信息的数据域。边表可定义为：

```
typedef struct vNode          //顶点表结点
{ VertexType vertex;          //顶点域
   EdgeNode *firstedge;       //边表头指针
    int info;                 //边上的信息
}VertexNode;
```

Dijkstra 算法参见教材中的算法中的 8-11。采用邻接表作为存储结构算法实现时，要注意循环条件的表示方式。

第9章 查 找

查找即在某种数据结构中找出满足给定特征的结点，若找到则查找成功，否则，查找失败。查找的基本问题是如何提高查找效率。被查找的对象集称为查找表，查找表是一个集合结构，常见的存储表示方法有顺序存储、树形存储和散列存储，存储方法不同，其查找效率也不同。本章内容是上述3种存储方法下的查找实现及各种查找方法的平均查找长度。

重点提示：

- 二分查找
- 二叉排序树建立与查找
- 平衡二叉树建立与查找
- B-树和B+树的概念
- 哈希表查找

9-1 重点难点指导

9-1-1 相关术语

1. 查找表

要点：一种以集合为逻辑结构，以查找为核心的数据结构。

2. 静态查找

要点：仅包括建表、查找、读表元3种基本运算；

不包括对表做修改操作（如插入和删除）。

3. 动态查找

要点：在查找的同时对表做修改操作（如插入和删除）；

它的基本运算包括建表、查找、读表元、插入和删除等基本运算。

4. 平均查找长度

要点：在查找表中查找第i个结点所需要的平均比较次数。

9-1-2 线性表查找

1. 顺序查找

（1）基本思想：从表的一端开始，将查找表中的数据元素关键字与查找关键字依次进行比较，在表中找到了要找的元素，则查找成功；否则，查找失败。

（2）存储方式：顺序存储或链式存储。

（3）顺序存储实现的表示：

```
typedef struct {
   Keytype key; // 关键字域
   InfoType  otherinfo; // 数据元素的其他部分
 }NodeType;
typedef  NodeType  SeqList[n+1];
```


查找表：SeqList R; // 查找表的n个元素存在R[1]…R[n]单元，R[0]留做监视哨。

（4）顺序存储下的顺序查找算法：

① 设置哨兵：将待查数据的关键字K置入R[0]中；

② 循环：从表的高下标端开始顺序比较，R[i].key==K时循环结束；

③ 返回i：i==0时则表明在R[1]…[n]中没有要找的数据，查找失败；否则查找成功，i为其存储下标。

（5）平均查找长度：

成功时：$ASL_{succ}=(n+1)/2$

不成功时：$ASL_{unsucc}=n+1$

时间复杂度：O(n)

（6）监视哨的作用：是为了在循环中省去判断防止下标越界的条件，提高查找效率。

2. 二分查找

（1）基本思想：在关键字已有序的查找表R[1]…R[n]中确定一个中间位置：mid=(1+n)/2，将查找的关键字K与R[mid]进行比较，若K=R[mid].key，则查找成功；若K>R[mid].key，则把查找区间确定为R[mid+1]…R[n]；若K<R[mid].key，则把查找区间确定为R[1]…R[mid-1]；…；依次类推，每查找一次，查找区间就缩小一倍，因此称它为二分查找或折半查找。

（2）存储方式：查找表一定要顺序存储且按关键字有序。链式存储的有序表不能进行二分查找，因为链式存储的情况下，不能通过查找区间中第一个结点的地址和最后一个结点的地址求出中间结点的地址来。

（3）存储实现的表示：同顺序查找中的顺序存储的实现。

（4）查找性能：

成功时平均查找长度：$ASL_{succ}=\log_2(n+1)-1$

不成功时的比较次数是判定树的深度。

平均查找长度：$O(\log_2 n)$

（5）判定树：根据二分查找的基本思想，把查找区间中一次查找成功的结点作为树的根结点，两次查找成功的结点作为根的左右结点，…，依次类推，这棵树就是判定树。通过一棵判定树可以直观地看到查找第i个结点所比较的元素及其次数。判定树的形状与查找表中的元素的个数n有关，与数据元素的值无关。

求给定n值的判定树见典型例题解析。

3. 分块查找

（1）基本思想：对关键字已按块有序的查找表每块建立一个索引项，将这些索引项顺序存储起来组成索引表。查找分两部分进行，首先从索引表中确定查找项所属的块，再从指定的块内进行查找。因为查找表按块有序，故索引表是一个有序表，在索引表中查找时可以按顺序查找也可以按二分查找，而因为块内无序，块内查找只能用顺序查找。

（2）存储方式：顺序存储或链式存储实现。

（3）性能分析：因为查找分为两步：确定块和块内查找，因此，分块查找的平均查找长度也有两部分组成，设表长为n，分为b块，每块内表长s=n/b：

① 若确定块时用二分查找（设索引表用顺序存储方式），成功时平均查找长度：

$$ASL_{succ}=ASL_{块}+ASL_{块内}$$

$$=\log_2(b+1)-1+(s+1)/2$$
$$=\log_2(n/s+1)+s/2$$

② 若确定块时用顺序查找，查找长度成功时平均查找长度：

$$ASL_{succ}=ASL_{块}+ASL_{块内}$$
$$=(b+1)/2+(s+1)/2$$
$$=(s^2+2s+n)/(2s)$$

9-1-3 排序树上的查找

1. 二叉排序树的定义

二插排序树或者是空树，或者是满足如下性质的二叉树：

（1）若它的左子树非空，则左子树上所有结点的值均小于根结点的值；

（2）若它的右子树非空，则右子树上所有结点的值均大于根结点的值；

（3）左、右子树本身又是一棵二叉排序树。

要点：由定义可知，它是一棵加了限制条件的二叉树。其限制条件是任一结点的关键字大于其左子树上的所有结点的值，而小于其右子树上的所有结点的值。因此得到二叉排序树的一个重要性质：中序遍历一棵二叉排序树，所得到的中序序列是一个按关键字递增有序的序列。定义中的条件（3）不能丢掉。

2. 二叉排序树上的查找

根据二叉排序树的特点，在二叉排序树上的查找过程为：

（1）若二叉排序树为空，查找失败，否则继续（2）。

（2）二叉排序树非空时，将给定值 k_x 与二叉排序树的根结点关键码比较，若相等，查找成功，返回结点的指针，结束查找过程；否则：

① 当给定值 k_x 小于根结点关键码，查找将在以左孩子为根的子树上继续进行，转（1）；

② 当给定值 k_x 大于根结点关键码，查找将在以右孩子为根的子树上继续进行，转（1）。

3. 二叉排序树上的插入

在二叉排序树上插入一个结点，要保证插入后仍然是二叉排序树。

设待插入结点的关键码为 k_x，为将其插入，先要在二叉排序树中进行查找，若查找成功，按二叉排序树定义，待插入结点已存在，不用插入；查找不成功时，则将新结点插入到适当位置上，插入的基础是查找。因为只有找到了叶子才知道要插入的结点不存在，因此，新插入结点一定是作为叶子结点添加上去的。

构造一棵二叉排序树则是在空二叉树的基础上逐个插入结点的过程。

4. 二叉排序树上的删除

在二叉排序树上删除一个结点，要保证删除后仍然是二叉排序树。

设待删除结点的关键码为 kx，为将其删除，先要在二叉排序树中进行查找，若查找不成功，不用删除；查找成功时，将其删除。

设待删结点为*p（p 为指向待删结点的指针），其双亲结点为*f，以下分 3 种情况进行讨论：

（1）*p 结点为叶结点，由于删去叶结点后不影响整棵树的特性，所以，只需将被删结点的双亲结点相应指针域改为空指针。

（2）*p 结点是一个单支结点（即只有右子树 p_r 或只有左子树 p_l），此时，只需将 p_r 或 p_l 替换*f 结点的*p 子树即可。

（3）*p 结点既有左子树 P_l 又有右子树 P_r，可按中序遍历保持有序进行调整：

用*p 结点的中序直接前驱结点（设指针为 p_1）或中序直接后继结点（设指针为 p_2）替换*p 结点，然后再删除*p_1 或*p_2，因为结点*p_1 或*p_2 肯定是一个单支结点，则可以按上述（2）进行删除。

*p 的中序直接前驱结点是中序遍历*p 的左子树时的最后一个结点，它在*p 的左子树的最右下角；*p 的中序直接后继结点是中序遍历*p 的右子树时的第一个结点，它在*p 的右子树的最左下角。用这样的方法做删除不会增加二叉排序树的深度。

5. 平衡二叉树（AVL 树）

（1）平衡二叉树的定义

平衡二叉树或是一棵空树，或是具有下列性质的二叉排序树：其左子树和右子树都是平衡二叉树，且左子树和右子树高度之差（即平衡因子）的绝对值不超过 1。

（2）平衡二叉树上查找、插入和删除

平衡二叉树是一种对二叉树的左子树和右子树高度差有要求的二叉排序树，因此，平衡二叉树上的查找、插入和删除操作与二叉排序树上的操作基本相同，不同之处是在插入、删除之后，需查找失衡结点，并对其进行调整。

对失衡的最小子树进行调整有以下 4 种情况：

① 左单旋转（RR 型）

失衡是因为在失衡结点的右孩子的右子树上插入结点造成的。需进行一次左旋调整，即将失衡结点的右孩子调整成为该失衡最小子树的根，其左孩子作为失衡结点的右孩子，失衡结点作为其新的左孩子。

② 右单旋转（LL 型）

失衡是因为在失衡结点的左孩子的左子树上插入结点造成的。需进行一次右旋调整，即将失衡结点的左孩子调整成为该失衡最小子树的根，其右孩子做为失衡结点的左孩子，失衡结点做为其新的右孩子。

③ 先左后右双向旋转（LR 型）

失衡是因为在失衡结点的左孩子的右子树上插入结点造成的。需先对失衡结点的左孩子的右子树进行一次左旋调整，再对失衡最小子树进行一次右旋调整。

④ 先右后左双向旋转（RL 型）

失衡是因为在失衡结点的右孩子的左子树上插入结点造成的。需先对失衡结点的右孩子的左子树进行一次右旋调整，再对失衡最小子树进行一次左旋调整。

5. B-树

B-树是一种平衡的多路查找树，它在文件系统中很有用。

（1）B-树定义

一棵 m 阶的 B-树，或者为空树，或为满足下列特性的 m 叉树：

① 树中每个结点至多有 m 棵子树；

② 若根结点不是叶子结点，则至少有两棵子树；

③ 除根结点之外的所有非终端结点至少有 m/2 棵子树；

④ 所有的非终端结点中包含以下信息数据：$(n,P_0,K_1,P_1,K_2,\cdots,K_n,P_n)$。

其中：K_i（i=1,2,…,n）为关键码，且 $K_i<K_{i+1}$，P_i 为指向子树根结点的指针（i=0,1,…,n），且指针 A_{i-1} 所指子树中所有结点的关键码均小于 K_i（i=1,2,…,n），A_n 所指子树中所有结点的关键码均大于 K_n，$m/2-1\leqslant n\leqslant m-1$，n 为关键码的个数。

⑤ 所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。叶子的层数为树的高度。

（2）B-树上的查找

B-树的查找类似于二叉排序树的查找，所不同的是 B-树每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码，查找失败。即在 B-树上的查找过程是一个顺指针查找结点和在结点中查找关键码交叉进行的过程。

（3）B-树上的插入

在 B-树上插入关键码与在二叉排序树上插入结点不同，关键码的插入不是在叶结点上进行，而是在最底层的某个非终端结点中添加一个关键码，若该结点上关键码个数不超过 m-1 个，则可直接插入到该结点上；否则，该结点上关键码个数至少达到 m 个，因而使该结点的子树超过了 m 棵，这与 B-树定义不符。所以要进行调整，即结点的“分裂”。方法为：关键码加入结点后，将结点中的 m 个关键码分成 3 部分：以中间一个关键码为分界线，前面一部分中有 m/2-2 个关键码，因此有 m/2-1 棵子树，仍留在原结点中，而后一部分中有大于等于 m/2-2 个关键码，因此大于等于 m/2-1 棵子树，将这一部分置入新加入的结点中，中间的关键码和新结点的指针一起插入到父结点中。若插入父结点而使父结点中关键码个数超过 m-1，则父结点继续分裂，直到插入某个父结点，其关键码个数小于 m。可见，B-树是从底向上生长的。

（4）B-树上的删除

分两种情况：

① 删除最底层结点中的关键码

a）若结点中关键码个数大于 m/2-1，直接删去。

b）否则若左兄弟（无左兄弟，则找右兄弟）项数之和大于等于 2(m/2-1) 就与它们父结点中的有关项一起重新分配。

② 若删除后，余项与左兄弟或右兄弟之和均小于 2(m/2-1)，就将余项与左兄弟（无左兄弟时，与右兄弟）合并。由于两个结点合并后，父结点中相关项不能保持，把相关项也并入合并项。若此时父结点被破坏，则继续调整直到根。

9-1-4 哈希表

1. 哈希函数及建立哈希表的思想

选取某个函数（通常它是关键字的函数），将每个元素的关键字代入该函数得到一个函数值，就依此作为该元素的存储位置，按这个思想构造的查找表称为哈希表，求其哈希地址的函数即为哈希函数。

同义词：哈希地址相同的不同关键码，称为同义词。

冲突：同义词之间争夺同一地址空间的现象。

堆积：非同义词之间争夺同一后继哈希地址的现象，这通常是由于解决冲突的方式造成的。

填装因子 α：将查找表的表长 n 除以哈希表的容量 m 定义为填装因子。

2. 常用的哈希函数

哈希函数的选择有两条标准：简单和均匀。简单是指哈希函数的计算简单快速；均匀是指，哈希函数能等概率地将其关键字集合中的每一关键字影射到表空间的任意位置上，即将不同的元素均匀地分布在表的地址集{0，1，m-1}上，以便使冲突最小化。

几种常见的哈希函数：平方取中法、除余法、相乘取整法和随机数法。

需要说明的是，哈希函数的选取不一定是这 4 种方法，要根据题目要求来定。

3. 几种解决冲突的方法

（1）开放地址法

要点：所谓开放定址法，是指当发生冲突之后，就按照某种规律去探查下一个哈希地址，如该地址也存放了数据元素，仍按照同样的规律再去探查下一个哈希地址，这样，形成了一个探查序列，只要哈希表足够大，空的散列地址总能找到，并将数据元素存入。如果找不到空单元，则说明哈希函数有问题或散列空间表过小。

根据形成探查规律的方法不同，开放地址法有以下几种形式，设某关键字的散列地址为 d：

① 线性探查法：探查序列为：d,d+1,…,m-1,0,1,…,d-1　即：(d+i) % m，其中：i=0,1,2,…,m-1。

优点：简单，能探查到整个表空间；

缺点：容易产生堆积。

② 二次探测法：探查序列为：$d,d+1^2,d+2^2\ldots$，即：$(d+i^2)\%m$，i=0,1,2,…。

优点：简单，不容易产生堆积。

缺点：不易探查到整个哈希表空间。

③ 双哈希函数探测法：

设两个哈希函数为 h (key)和 h_1(key)，探测序列为：

$$d,\ (d+h_1(key))\%m,\ \ (d+2*h_1(key))\%m,\ \ (d+3*h_1(key))\%m\ \ldots,$$

即：$(d+i\times h_1(key))\%m$　i=0,1,2,…

（2）拉链法

要点：拉链法解决冲突的做法是，将所有的同义词结点链接成一个单链表，再将链表的头指针置入与散列地址对应的存储单元中。

优点：处理冲突的方法简单；不会产生堆积现象，平均查找长度较短；删除操作易于实现；填装因子α可以>1。

缺点：指针需要增加额外的存储开销。

9-2 典型例题解析

9-2-1 判断题

1．折半查找只能在有序的顺序表上进行而不能在有序链表上进行。

【分析】折半查找时需要根据查找区间的下限和上限来定位中间位置，这对于单链表存储的查找表是不能做到的，因为单链表中的每个结点的地址可能不是连续分配的，因此，单链表不能进行折半查找，只能从头结点开始逐步搜索。

【答案】正确

2．对于给定的关键字集合，以不同的次序插入到初始为空的二叉排序树中，得到的二叉排序树是相同的。

【分析】得到的二叉排序树不一定相同，因为第一个关键字是二叉排序树的根结点，以后每个结点的插入都是从根结点开始比较，按照二叉排序树的性质确定新结点的位置，因此以不同的次序插入到初始为空的二叉排序树中得到的二叉排序树可能是不相同的。如：关键字集合{1,2,3}，以{1,2,3}插入与{2,3,1}的插入是不同的，以{2,1,3}插入与{2,3,1}的插入是相同的。

【答案】错误

3．将二叉排序树 T 的先序序列中的关键字依次插入初始为空的树中，所得到的二叉排序树 T 与 T′ 是相同的。

【分析】T 与 T′ 完全相同。构造二叉排序树就是从根开始的，根据先序遍历的特点，按先序遍历的顺序依次插入结点，新建的二叉排序树与原树完全相同。如果按中序、后序遍历的顺序依次插入结点，新建的二叉排序树与原树可能不相同。

【答案】正确

4．哈希表的查找效率完全取决于所选取的哈希函数和处理冲突的方法。

【分析】哈希表的查找效率还受装载因子的影响，填装因子越大，存取元素时发生冲突的可能性就越大，查找效率就会降低。

【答案】错误

9-2-2 选择题

1．静态查找表与动态查找表的根本区别在于 _____。

A．它们的逻辑结构不一样　　B．施加在其上的操作不一样

C．所包含的数据元素类型不一样　　D．存储实现不一样

【分析】动态查找表上运算包括插入删除操作，静态查找表上运算不包括插入删除操作。

【答案】B

2．与其他查找方法相比，散列查找法的特点是_____。

A．通过关键字的比较进行查找

B．通过关键字计算元素的存储地址进行查找

C．通过关键字计算元素的存储地址并进行一定的比较进行查找

D．以上都不是

【分析】散列查找时，首先按关键字计算元素的存储地址，然后再进行比较，如果不是查找的元素，则继续按解决冲突的方法进行探查。

【答案】C

3．顺序查找适用于存储结构为_____的线性表

A．哈希存储　　B．压缩存储

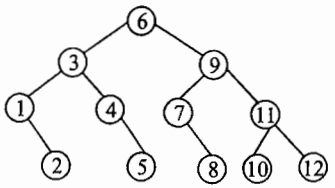
C．顺序存储或链接存储　　D．索引存储

【分析】对于顺序和链式存储的数据都可以进行顺序查找。

【答案】C

9-2-3 应用题

1．有一个 n=12 的有序表且顺序存储，画出二分查找过程的查找树，并求查找成功情况下的平均查找长度。



【解答】

第一次查找成功的是位置 6 上的结点，第二次成功的是位置 3、9 上的结点，第三次成功的是位置 1、4、7、11 上的结点，第 2、5、8、10、12 位置上的结点是比较 4 次才成功的，因此有上面的查找树。

平均查找长度 ASLSUCC=(1×1+2×2+3×4+4×5)/12=37/12

2．已知如下所示长度为 12 的查找表：

{Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec}

试按表中元素的顺序依次插入到一棵初始为空的二叉排序树中。

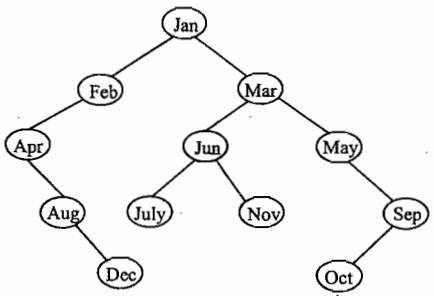
（1）画出插入完成后的二叉排序树；

（2）并求其在等概率的情况下查找成功的平均查找长度；

（3）若对表中元素先进行排序构成有序表再构造，所构造的二叉排序树是什么形状，并求在等概率的情况下查找成功的平均查找长度。

【解答】

（1）按关键字的顺序构造的二叉排序树：



（2）根据构造的二叉排序树，求成功时平均查找长度：

$$ASL_{SUCC}=(1\times1+2\times2+3\times3+4\times4+5\times2)/12=10/3$$

（3）若对表中元素先进行排序构成有序表再构造二叉排序树，则构造的二叉排序树是一棵单支树，在等概率的情况下查找成功的平均查找长度则为：

$$ASL_{SUCC}=(1+2+\ldots+12)/12=6.5$$

这种情况就退化成为顺序查找。

3．已知查找表为{19,01,23,14,55,20,84,27,68,11,10,77}，设定哈希函数为：H(key)=key % 13，采用开放地址法的二次探测法解决冲突，试在 0～18 的散列地址空间中对该关键字构造哈希表，并求出查找成功时的平均查找长度。

【解答】

依题意，哈希表的容量为 m=19，二次探测法的探查序列是：

$$h_i=(H(key)\pm i\times i)\ \%\ m \quad 0\leqslant i\leqslant m-1$$

即探查序列为 d=H(key),d+1^2, d-1^2,d+2^2, d-2^2…。计算各元素的散列地址如下：

H(19)=19%13=6

H(01)=01%13=1

H(23)=23%13=10

H(14)=14%13=1（冲突）　H(14)=（1+1^2）%19=2

H(55)=55%13=3

H(20)=20%13=7

H(84)=84%13=6（冲突）　H(84)=（6+1^2）%19=7（仍冲突）

H(84)=（6-1^2）%19=5

H(27)=27%13=1（冲突）　H(27)=（1+1^2）%19=2（仍冲突）

H(27)=（1-1^2）%19=0

H(68)=68%13=3（冲突）　H(63)=（3+1^2）%19=4

H(11)=11%13=11

H(10)=10%13=10（冲突）　H(10)=（10+1^2）%19=11（仍冲突）

H(10)=（10-1^2）%19=9

H(77)=77%13=12

因此哈希表如下：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
27	1	14	55	68	84	19	20		10	23	11	77						
3	1	2	1	2	3	1	1		3	1	1	1						

查找成功时的平均查找长度 ASL_{succ}=(3+1+2+1+2+3+1+1+3+1+1+1)/12=20/12

9-2-4　算法设计题

1．编写一个算法，利用二分法查找算法在一个有序表中插入一个元素 x，并保持表的有序性。

【分析】

依题意，先在有序表 R 中利用二分法查找关键字值小于或等于 x 的结点，mid 指向正好等于 x 的结点或 low 指向关键字正好大于 x 的结点，然后采用移动法插入 x 结点即可。

【算法】

```
BinInsert (SeqList  R,KeyType  x)
{ int low=1,hig=n,mid,inplace,i,find=0;
  while (low<=hig && !find)
    { mid=(low+hig)/2;
      if (x.key<R[mid].key)  hig=mid-1;
      else if (x.key>R[mid].key)  low=mid+1;
           else { i=mid;
                  find=1;
                }
    }
  if (find) inspace=mid;
  else inspace=low;
  for (i=n;i>=inspace;i--)    R[i+1]=R[i];
  R[inspace]=x;
}
```

2．设顺序表中关键字是递增有序的，试写一顺序查找算法，将哨兵设在表的高下标端。然后求出等概率情况下查找成功与失败时的 ASL。

【分析】

因为顺序表中关键字是递增有序的，所以从低下标端开始顺序查找，若当前的关键码比要查找的关键码大，说明查找失败，终止查找。

【解答】

（1）算法

```
int seqsearch(SeqList R,KeyType K)
 { int i;
   R[n]=K;  // 设置哨兵
   i=0;
   while(R[i].key<K)  i++;
   if (R[i].key==K)
     return(i% n);
   else
     return (0);
   Z}
```

（2）查找长度

成功情况：ASL_{succ}=(1+2+3+……+n)/n=(n+1)/2

失败情况：K<R[0].key 时，比较 1 次，就终止查找；

R[0].key<K< R[1].key 时，比较 2 次，就终止查找；

R[1].key<K< R[2].key 时，比较 3 次，就终止查找；

…

R[n-2].key<K< R[n-1].key 时，比较 n 次，就终止查找；

R[n-1].key<K 时，比较 n+1 次，就终止查找；

所以 ASL_{unsucc}=((1+2+3+……+n+（n+1）)/(n+1)=（n+2）/2。

3．试写出二分查找的递归算法。

【分析】

从查找表的中间切割一次，查找表的区间就缩小一半，所以二分查找也叫折半查找。对于每次查找的思想是相同的，只是查找区间发生了变化，因此写为递归算法时要将查找区间的上下限作为参数，递归调用时查找区间的上下限是变化的。

【算法】

```
int Binsrch(SeqList R,int low,int hig,KeyType K)
{ int mid;
  if (low<hig)
        return(0);
   else
      { mid=(low+hig)/2;
       if(K==R[mid].key)
            return(mid);
        else if (K>R[mid].key)
            return(Binsrch(R, mid+1,hig,K));
            else  return( Binsrch (R,low,mid-1,K) );
      }
}
```

4. 试写一算法判别给定的二叉树是否为二叉排序树，设此二叉树以二叉链表为存储结构，且树中结点的关键字均不相同。

【分析】

判断一棵二叉树是否是二叉排序树，可以采用递归算法。对于树中所有结点，检查其左子树上所有结点的关键字是否都小于它的关键字，检查其右子树上所有结点的关键字是否都大于它的关键字。

【算法】

```
int  binsorttree(BSTree  t)
 {// 判别由二叉链表存储的二叉树是否为二叉排序树，是则返回 1，否则返回 0
  int l,r;
    if(!t ) return 1;// 空树是二叉排序树
    else  if(t->lchild==NULL && t->rchild=NULL )
            return  1;// 只有一个根结点是二叉排序树
    else
        { l= binsorttree( t->lchild);
          r = binsorttree( t->rchild);
          if(t->lchild && t->rchild )
           if (t->key>t->lchild->key && t->key<t->rchild->key && l && r)
               return 1;
           else  ruturn 0;
          else   if  (t->lchild==NULL )    // 只有一个根结点是二叉排序树
                 return  ( t->key < t->rchild->key  && r)
             else                          // 只有一个根结点是二叉排序树
                  return  ( t->key > t->lchild->key && l)
       }
  }
```

5. 试写一递归算法，从大到小输出二叉排序树中所有其值不小于 X 的关键字。要求算法时间为 $O(\log_2 n+m)$，n 为树中结点数，m 为输出的关键字个数。

【分析】

由于若按照左根右的顺序（即中序）遍历二叉排序树，则得到的是由小到大的顺序，因此，可以按照右根左的顺序遍历二叉排序就能很快找到值不小于 X 的关键字。

【算法】

```
inorder(BSTree  bt, KeyType  X)
```

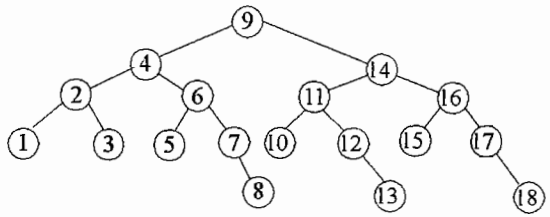


```
{
  if (bt)
     { inorder(bt->rchild,X);
        if(bt->key>=X) printf("%c",bt->key);
       inorder(bt->lchild,X);
      }
}
```

9-3　课后习题选解

1. 画出对长度为 18 的有序的顺序表进行折半查找时的判定树，并指出在等概率时查找成功的平均查找长度，以及查找失败时所需的最多的关键字比较次数。

判定树为：



平均查找长度为 1/18(1+2×2+3×4+4×8+5×3)=32/9

查找最多比较 5 次。

2. 已知如下所示长度为 12 的查找表：

{Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec}

试按表中元素的顺序依次插入到一棵初始为空的二叉排序树。

（1）画出插入完成后的二叉排序树；

（2）并求其在等概率的情况下查找成功的平均查找长度。

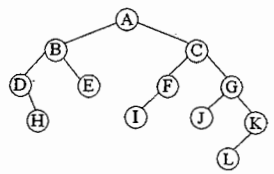
若对表中元素先进行排序构成有序表再构造，所构造的二叉排序树是什么形状，并求在等概率的情况下查找成功的平均查找长度。

（参考典型例题解析中的应用题 2）

3. 试推导含有 12 个结点的平衡二叉树的最大深度，并画出一棵这样的树。

令 F_k 表示含有最少结点的深度为 k 的平衡二叉树的结点数目，那么，可知道 $F_1=1,F_2=2,.....Fn=Fn-2+Fn-1+1$。

含有 12 个结点的平衡二叉树的最大深度为 5。例如：

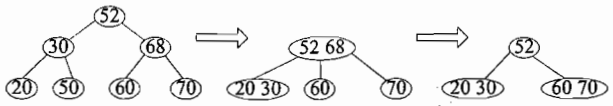


4. 含有 9 个叶子结点的 3 阶 B-树中至少有多少个非叶子结点？含有 10 个叶子结点的 3

阶 B-树中至少有多少个非叶子结点？

4 个；7 个。

5．试从空树开始，画出按以下次序向 3 阶 B-树中插入关键码的建树过程：20，30，50，52，60，68，70。如果此后删除 50 和 68，画出每一步执行后 3 阶 B-树的状态。



6．在地址空间为 0～16 的散列区中，对以下关键字序列构造两个哈希表：

{Jan, Feb, Mar, Apr, May, June, July, Aug, Sep, Oct, Nov, Dec}

（1）用线性探测开放地址法处理冲突。

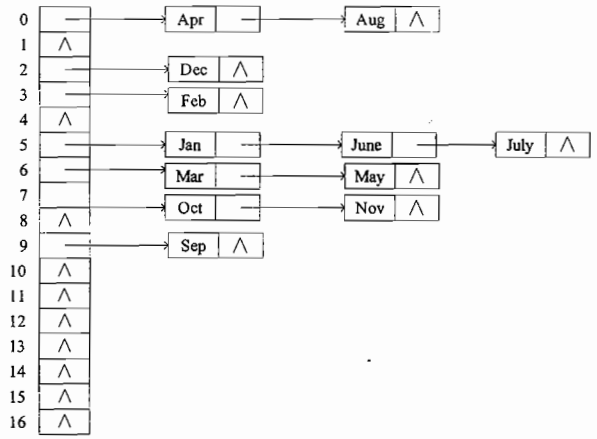
9.6(1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Apr	Aug	Dec	Feb		Jan	Mar	May	June	July	Sep	Oct	Nov				

平均查找长度为:31/12

（2）用链地址法处理冲突。

并分别求这两个哈希表在等概率情况下查找成功和不成功的平均查找长度。设哈希函数为 H(key) =i/2，其中 i 为关键字中第一个字母在字母表中的序号。



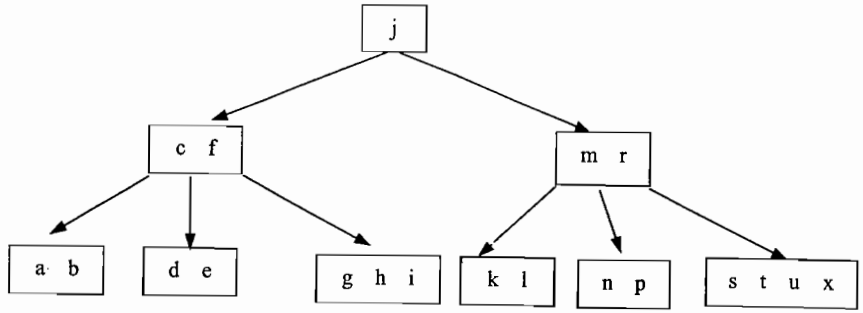
平均查找长度为:3/2

7．哈希函数 H(key)=(3*key) % 11。用开放定址法处理冲突，d_i=i ((7*key) % 10+1)，i=1,2,3,…。试在 0～10 的散列地址空间中对关键字序列(22,41,53,46,30,13,01,67)构造哈希表，并求等概率情况下查找成功时的平均查找长度。

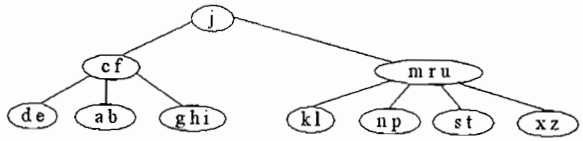
0	1	2	3	4	5	6	7	8	9	10
22	67	41	30		53	46		13		01

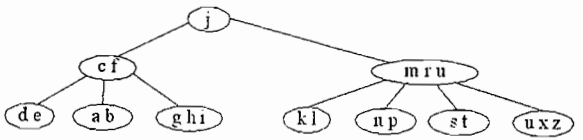
平均查找长度为:17/8

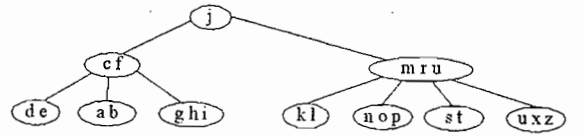
8．画出依次插入 z，v，o，q，w，y 到下图所示的 5 阶 B-树的过程。

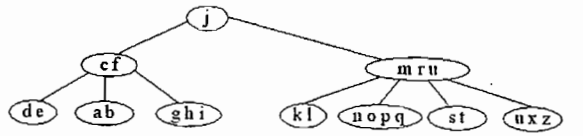


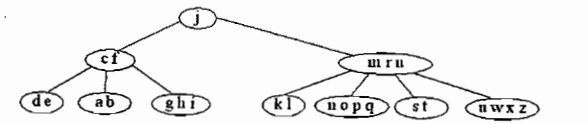
（题 8 图）

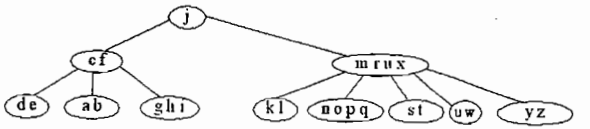












9. 设顺序表中关键字是递增有序的，试写一顺序查找算法，将哨兵设在表的高下标端。然后求出等概率情况下查找成功与失败时的 ASL。

（参考典型例题解析中的算法设计题 2）

10. 试写出二分查找的递归算法。

（参考典型例题解析中的算法设计题 3）

11. 在平衡二叉排序树的每个结点中增设一个 lsize 域，其值为它的左子树中的结点数加 1。试写以时间复杂度为 O(log2n)的算法，确定树中第 k 小的结点的位置。

【提示】由二叉排序树的性质和 lsize 域的定义可知，第 k 小的结点即 lsize 域的值为 k 的结点；因此，可以采用递归算法在二叉排序树中查找这样的结点。

```
typedef  struct  BTNode
{ Elemtype   data;
   int  lsize;
   struct  BTNode  *lchild ,  *rchild;
}BTNode ,*BiTree;                          // 二叉树的类型定义

BiTree   index (BiTree  t,  int k)
{ if  (t=NULL)   return(NULL);             // 若根结点 t 为空，则查找失败
   if(t->lsize= =k)  return(t);            // 找到第 k 小结点，返回其地址
       else if(t->lsize>k) return (t->lchild,k); // 若 k 小于根结点，在左子树继续查找
          else    return(t->rchild,k);     // 若 k 大于根结点，在右子树继续查找
}
```

12. 假设哈希表长为 m，哈希函数为 H(key)，用链地址法处理冲突。试编写输入一组关键字并建立哈希表的算法。

【提示】输入关键字后，应先确定其哈希地址，再将其插入到相应的单链表中去。

```
typedef    struct
{ KeyType    key;
  ……
}ElemType;
typedef   struct   HNode
{ElemType   data;
   struct   HNode   *next;
}Hnode ,*Hlist;                                  // 哈希表的类型定义
void   CreateHList (Hlist   L[m])
{ for( i=0; i<m; i++ )
  L[i]=NULL;
  Cin>>x ;
  while  (x!='#')
    {  h=H(x);                                   // 计算 x 的哈希地址
       s=(Hnode *) malloc (sizeof(HNode)) ;
       s->data.key=x;                            // 申请、填装结点
       s->next=L[h] ;
       L[h] = s;                                 // 插入到哈希表中去
       Cin>>x;
    }
    return;
    }
```


13. 已知一棵二叉排序树上所有关键字中的最小值为-max，最大值为 max，又知-max<x<max。编写递归算法，求该二叉排序树上的小于 x 且最靠近 x 的值 a 和大于 x 且最靠近 x 的值 b。

【提示】解决本题的关键是递归函数的参数设置，采用逐渐缩小查找区间的方法来实现。

```
typedef  struct  BTNode
{ struct  { int  key;
            …
          }data;
  struct  BTNode  *lchild ,  *rchild;
}BTNode ,*BiTree ;                    // 二叉排序树的类型定义
void  fun(BiTree  t,  int x,  int  *a,  int  *b)
{ if (T)
     { if (x<t->data.key)             // 当 x 小于根结点时，修改 b，在左子树上继续查找
        {  *b = t->data.key ;
           fun(t->lchild, x,a,b);
        }
     else  if (x>t->data.key)         // 当 x 大于根结点时，修改 a，在右子树上继续查找
           {   *a = t->data.key ;
               fun(t->rchild,x,a,b);
           }
         else // 当 x 等于根结点时，a 是其左子树的最右下结点，b 是其右子树的最左下结点
           {  if (t->lchild)
                 {   p=t->lchild ;
                     while (p->rchild)   p=p->rchild;
                     *a=p->data.key;
                 }
                 if  (t->rchild)
              {    p=t->rchild;
                   while (p->lchild )  p=p->lchild;
                   *b=p->data.key;
              }
           }
     }
  return;
  }
```

14. 编写一个算法，利用二分法查找算法在一个有序表中插入一个元素 x，并保持表的有序性。

（参考典型例题解析中的算法设计题 1）

15. 假设二叉排序树 t 的各个元素值均不相同，设计一个算法按递减次序打印各元素的值。

【提示】调整中序递归遍历算法，先遍历右子树，然后输出根结点的值，再遍历左子树。

```
void  printnode (BiTree  t)
{ if (t)
    {printnode(t->rchild);
     cout<< t->data<<endl ;
     printnode(t->lchild);
     }
   return;
    }
```

16. 设计在有序顺序表上进行斐波那契查找的算法，并画出长度为 20 的有序表进行斐波那契查找的判定树，求出在等概率下查找成功的平均查找长度。

【提示】采用类似折半查找的算法，按照菲波那契数列分割查找表，实现查找。

```
int  fibo_search(S_table  L,  KeyType  x,  int  *fibo)
{low = 1; high = L.length ;                          // 设L.length=fibo[K]-1
 while  (low<=high)
 { mid =low +fibo[K-1] ;
  if( x<L.elem[mid] )    high=mid-1;
        else if (x>L.elem[mid])    low=mid+1;
                 else          return(mid) ;
 }
 return(0);
 }
```

17. 试写一算法判别给定的二叉树是否为二叉排序树，设此二叉树以二叉链表为存储结构，且树中结点的关键字均不相同。

（参考典型例题解析中的算法设计题 4）

第 10 章　排　序

排序和查找是计算机科学中十分重要的一类问题，因为就总体而言，计算机的运行时间中有相当大的一部分是在处理此任务。因此，在有的关于数据结构的论述中，将排序和查找归结为计算机程序设计中的两项常用技术。

本章介绍排序。重点介绍了依据插入思想的直接插入排序和希尔排序；依据交换思想的冒泡排序和快速排序；依据选择思想的直接选择排序和堆排序；依据合并思想的归并排序；以及依据分配思想的基数排序。要求熟悉各种排序算法的基本思想、排序过程和时间、空间的性能分析，了解各种排序算法的特点与使用的情况。

重点提示：

- 掌握各种排序方法的排序过程、时空效率、稳定性和适用情况
- 一些排序算法的实现（如插入排序、选择排序、冒泡排序、快速排序、堆排序、归并排序和基数）
- 堆的概念和建立

10-1　重点难点指导

10-1-1　相关术语

1. 排序

要点：输入 n 个记录 R_1、R_2、…、R_n，其相应的关键字分别为 K_1、K_2、…、K_n。

输出 R_{i1}、R_{i2}、…、R_{in}，使得 $K_{i1}\leqslant K_{i2}\leqslant\cdots\leqslant K_{in}$（或 $K_{i1}\geqslant K_{i2}\geqslant\cdots\geqslant K_{in}$）。

2. 排序方法的稳定性

要点：具有相同关键字的记录之间的相对次序保持不变的排序方法是稳定的，否则是不稳定的。

3. 内部排序

要点：对内存中数据进行排序；

排序时不涉及数据的内、外存交换。

4. 外部排序

要点：按照某种策略将外存数据分批调入内存进行排序，从而达到整体有序；

排序时涉及到数据的内、外存交换。

5. 关键字

要点：用于标识记录的数据项。

6. 堆

要点：是指 n 个关键字序列 $K_1,K_2,\ldots K_n$，当且仅当该序列满足如下性质：

① $K_i\leqslant K_{2i}$ 且 $K_i\leqslant K_{2i+1}$ 或 ②$K_i\geqslant K_{2i}$ $K_i\geqslant K_{2i+1}$ $(1\leqslant i\leqslant n/2)$

7. 就地排序

要点：排序算法所需的辅助空间不依赖于问题的规模 n，如 O(1)。

10-1-2 插入排序

插入排序的基本思想：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中的适当位置，直到全部记录插入完成为止。

1．直接插入排序

（1）基本思想

先将待排序的第一个元素视为一个元素的有序序列，然后将其后从第二个元素到第 n 个元素依次插入，形成两个元素的有序序列、3 个元素的有序序列，直至形成 n 个元素的有序序列。

（2）算法效率

时间复杂度最好的情况是已经排序，比较次数为 $C_{min}=n-1$，移动次数为 $M_{min}=0$；最坏的情况是反序时进行插入排序，$C_{max}=(n+2)(n-1)/2=O(n^2)$，$M_{max}=(n-1)(n+4)/2=O(n^2)$；平均的比较次数和移动次数约为 $n^2/4$。

空间复杂度为 O(1)。

（3）排序特点

① 是一种稳定的排序方法；

② 适用于接近排序的情况；

③ 适用于 n 较小的情况；

④ 直至最后一趟排序过程才能确定一个元素的最终位置。

2．希尔排序

（1）基本思想

先取定一个小于 n 的整数 d_1 作为第一个增量，把文件的全部记录分成 d_1 个组，所有距离为 d_1 的倍数的记录放在同一个组中，在各组内进行直接插入排序；然后，取第二个增量 $d_2<d_1$ 重复上述的分组和排序，直至所取的增量的 $d_t=1$（$d_t<d_{t-1}<\cdots<d_2<d_1$），即所有记录都放在同一组中进行直接插入排序为止。

（2）希尔排序增量的选取

增量序列可有各种取法，但需注意：应使增量序列中的值没有 1 之外的公因子，并且最后一个增量值必须等于 1。

（3）算法效率

排序开始时，增量较大，分组较多，每组内记录数目少，故组内直接插入排序较快；随着增量的减小，分组减少，每组内记录数目增加，但由于各记录已接近有序状态，因此新的一趟排序过程也较快。可证明希尔排序的时间效率比直接插入排序要快，当 n 在某个特定范围内，希尔排序所需的比较和移动次数约为 $n^{1.3}$，当 $n->\infty$ 时，可减少到 $n(\log_2 n)^2$。空间复杂度为 0(1)。

（4）排序特点

① 仍然采用的是插入的思想进行排序，是对直接插入法进行改进，扬其长避其短；

② 是不稳定的一种排序方法；

③ 适用于对稳定性不做要求、对排序效率要求不太高的数据进行排序；

④ 每一趟排序会使数据更接近于有序；

⑤ 直至最后一趟排序过程才能确定一个元素的最终位置。

10-1-3 交换排序

交换排序的基本思想：是两两比较待排序记录的关键字，发现两个记录的次序相反时即进行交换，直到没有反序的记录为止。

1．冒泡排序

（1）基本思想

设想被排序的记录数组 R[1…n]垂直竖立，将每个记录 R[i]看做是重量为 R[i].key 的气泡。根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 R，凡扫描到违反原则的轻气泡，就使其向上“漂浮”，如此反复进行，直到最后任何两个气泡都是轻者在上，重者在下为止。

（2）算法效率

最好的情况为已有序，比较次数为 n 次，移动次数为 0 次，时间复杂度为 O(n)，最坏和平均情况下的时间复杂度为 $O(n^2)$，最坏的情况是逆序，比较次数为 n(n-1)/2，移动次数为 3n(n-1)/2。

（3）排序特点

① 是一种稳定的排序方法；

② 适用于接近排序的情况；

③ n 较小的情况；

④ 每一趟排序过程都能确定一个元素的最终位置。

2．快速排序

（1）基本思想

在待排序的 n 个记录中任取一个记录（例如，取第一个记录），以该记录的键值为标准，将所有记录分为两组（一般都是无序的），使得第一组中各记录的键值均小于或等于该键值，第二组中各记录的键值均大于该键值。然后把该记录排在这两组关键值的中间（这也是该记录最终点位置）。此段为一趟快速排序（一次划分），对所分成的两组再分别重复上述方法，直到所有记录都排在适当位置为止。

（2）算法效率

快速排序不是一种稳定的排序方法。从平均性能而言，快速排序最佳，其时间复杂性为 $O(n\log_2 n)$，但在最坏情况下，即对几乎是有序的序列，该算法的效率很低，近似于 $O(n^2)$。对于较少的值，该算法效果不明显；反之，对于较大的 n 值，效果较好。

（3）排序特点

① 是一种不稳定的排序方法；

② 不适用于接近或已有序的情况，即适用于离排序较远的情况；

③ 适用于 n 较大的情况；

④ 每一趟排序过程都能确定一个元素的最终位置。

10-1-4 选择排序

选择排序的基本思想是：第 i 趟在 n-i+1（i=1,2…n-1）个记录中选取键值最小的记录作为有序序列第 i 个记录。

1．直接选择排序

（1）基本思想

首先在所有的记录中选出键值最小的记录，把它与第一个记录交换，然后在其余的记录中再选出键值最小的记录与第二个记录交换；依次类推，直至所有记录排序完成。

（2）算法效率

对 n 个待排序的记录进行直接选择排序，需进行 n-1 趟排序，在第 i 趟中，需通过 n-i 次键值比较选出所需记录，故直接选择排序的时间复杂性为 $O(n^2)$。

只需增加一个辅助空间。

（3）排序特点

① 不是一种稳定的排序方法；

② 算法简单、易懂，容易实现；

③ 不适宜 n 较大的情况。

④ 每一趟排序过程都能确定一个元素的最终位置。

2.堆排序

（1）基本思想

在排序过程中，将 R[1...n]看成是一棵完全二叉树以顺序存储结构存放，利用完全二叉树中双亲结点和孩子结点之间的内在关系，将其建成堆，从而在当前无序区中选择关键字最大（或最小）的记录，然后将最大（或最小）键值取出，用剩下的键值再建堆，便得到次最大（或次最小）的键值。如此反复进行，直到最小（或最大）值，从而将全部键值排好序为止。

（2）算法步骤

一般按由小到大排序建大顶堆，按由大到小排序建小顶堆，堆排序大体可分如下两大步：

① 建堆过程；

将待排的记录看做完全二叉树，从最后一个非叶子结点开始调整建堆，直到根结点，使整棵二叉树为一个堆，根即为最大（小）的元素。

② 堆的调整过程。

将根与完全二叉树的最后一个结点交换；

并将交换后完全二叉树的最后一个结点从完全二叉树中去掉，即视其为完全二叉树之外的点；

对新的根进行调整，使剩余的完全二叉树成为堆；

这样如此反复直到完全二叉树中仅剩一个结点为止。

（3）算法效率

堆排序在最坏和平均情况下的时间复杂度均为 O(nlog2n)。但由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

堆排序是就地排序，辅助存储空间为 O(1)。

（4）排序特点

① 不是一种稳定的排序方法；

② 时空效率高；

③ 适于 n 较大的情况；

④ 每一趟排序过程都能确定一个元素的最终位置；

⑤ 适用于在较大的 n 中找序列的前几个数。

10-1-5　归并排序

（1）基本思想

将有序的子序列进行合并，从而得到有序的序列。其中，合并算法思想是：比较各子序列第一个记录的键值，最小的一个就是排序后序列的第一个记录的键值。取出这个记录，继续比较各子序列现在的第一记录的键值，便可找出排序后的第二个记录。如此继续下去，最终可得到排序结果。

（2）算法步骤

方法一：

对几个记录排序的步骤为：将每一个待排记录视为一个元素的有序序列，通过两两合并，得到含有两个元素的有序序列，再进行两两合并，得到含有 4 个元素的有序序列；……，如此进行直到合并为一个有 n 个元素的有序序列。

方法二：

采用递归的方法。即对几个记录进行排序的步骤可递归地描述为：

① 对前 1 至 n/2 个记录排序

② 对后 n/2+1 个记录排序

③ 将两有序序列合并

（3）算法效率

时间复杂度为 O(nlog2n)，执行归并排序需附加一倍的存储开销。

（4）排序特点

① 是一种稳定的排序方法；

② 时间效率高，但空间效率低；

③ 适于 n 较大的情况；

④ 直到最后一趟排序过程才能确定每个元素的最终位置。

10-1-6　基数排序

（1）基本思想

前面的几种排序方法中的基本运算都是比较和移动，而基数排序则是根据要排序对象的表示形式进行排序。它的原理和卡片分类相类似。

假设排序对象是由 n 位数字构成，先按最小一位数字排序，然后依顺序收集起来，再按第二位数字排序，依次类推。令 X_i 和 X_j 两个关键字分别为：

$$X_i = X_k^{(i)} X_{k-1}^{(i)} \dots X_2^{(i)} X_1^{(i)}$$

$$X_j = X_k^{(j)} X_{k-1}^{(j)} \dots X_2^{(j)} X_1^{(j)}$$

若存在 $t \leq k$，使得当 $k \geq l > t$, $X_l^{(j)} = X_l^{(i)}$，但 $X_t^{(j)} < X_t^{(i)}$，则 $X_j < X_i$。当然 $X_k^{(j)} < X_k^{(i)}$，也导致 $X_j < X_i$。

基数分类法是模拟分卡片的办法。即先按第一位数 $X_1^{(i)}$把卡片分到 0 到 9 的接收器，然后依次从 0 到 9 的接收器中把卡片收集起来，所得的卡片按第一位数排序完毕。

接着把卡片按第二位数 $X_2^{(i)}$再分到 0 到 9 的接收器，然后再依次把卡片收集起来。再按第三位数 $X_3^{(i)}$把卡片分到从 0 到 9 的接收器，并依顺序收集起来。依次类推。

（2）算法步骤

若关键字的长度为 d，基数为 r_d 则算法的基本步骤为：i 从 1 到 d 对几个关键字按第 i 位进行分配；将分配得的 r_d 组数回收。

（3）算法效率

基数排序中，若 r_d 为基数即卡片箱的个数，d 为最长整数的位数，n 为待排序记录的个数，其一趟的分配时间为 O(n)，收集时间为 O(n+rd)，d 趟排序后总的时间为 O(d(2n+rd))，若 d 为常数，r_d 为常数，则基数排序的时间复杂性是线性的，即为 O(n)。此外，基数排序所需的辅助存储空间为 $O(n+r_d)$。

（4）排序特点

① 是一种稳定的排序方法；

② 排序中的基本操作不是比较和交换，而是分配和回收；

③ 适于一个关键字由多个部分组成，对关键字的比较可由对多个部分的比较进行；

④ 最后一趟分配、回收过程才能确定每个元素的最终位置。

10-1-7 外部排序

（1）基本思想

外部排序由两个相互独立的阶段组成。第一个阶段是将外存上的含有 n 个记录的文件按照可用内存的大小分成若干长度为 k 的子文件或段，依次将这些段读入内存进行排序，使得各子文件成为内部有序的；第二个阶段是将这些有序段进行逐趟归并，直至得到整个有序文件为止。

（2）算法效率

外部排序所需总时间＝内部排序所需时间＋外部信息读写时间＋内部归并排序所需时间。

10-2 典型例题解析

10-2-1 判断题

1．用希尔（Shell）方法排序时，若关键字的初始排序杂乱无序，则排序效率就低。对有 n 个记录的集合进行归并排序时，所需要的辅助空间数与初始记录的排列状况有关。

【分析】希尔排序是每趟只对相同增量距离的关键字进行比较，这与关键字序列初始有序或无序无关。

【答案】错误

2．堆排序所需要附加空间数与待排序的记录个数无关。

【分析】堆排序无论待排序的记录个数有多少，所需的附加空间数都是 O(1)。

【答案】正确

3．对有 n 个记录的集合进行冒泡排序，所需时间决定于初始记录的排列情况；在初始记录无序的情况下最好。

【分析】对有 n 个记录的集合进行冒泡排序，在初始记录有序的情况下效率最好。

【答案】错误

4．快速排序的速度在所有排序方法中为最快，而且所需附加空间也最少。

【分析】快速排序需要一个额外的栈空间，直接插入排序、冒泡排序、直接选择排序和堆排序所需附加空间最少。

【答案】错误

5．对一个堆，按二叉树层次进行遍历可以得到一个有序序列。

【分析】堆的定义只规定了结点与其左、右孩子结点间的大小关系，而同一层上属不同双亲的结点之间并无明确的大小关系。

【答案】错误

10-2-2 选择题

1．在待排序的元素序列基本有序的前提下，效率最高的排序方法是（ ）。

A．插入排序　　B．选择排序　　C．快速排序　　D．归并排序

【分析】在待排序的元素序列基本有序的前提下，插入排序所需的交换次数和比较次数最少，效率最低的是快速排序。

【答案】A

2．设有 1000 个无序的元素，希望用最快的速度挑选出其中前 10 个最大的元素，最好选用（ ）排序法。

A．冒泡排序　　B．快速排序　　C．堆排序　　D．基数排序

【分析】选项中只有冒泡排序和堆排序分别进行 10 趟冒泡和 10 趟堆的调整即可得到所要结果，而快速排序和基数排序有可能到最后一趟全排好序后，才能确定前 10 个最大元素。在不要求稳定性的情况下，堆排序又比冒泡排序效率高。

【答案】C

3．对于具有 12 个记录的序列，采用冒泡排序最少的比较次数是（ ）。

A．1　　B．144　　C．11　　D．66

【分析】冒泡排序所需最少的比较次数的情况是待排序记录已有序，只需进行一趟比较即可，12 个记录，一趟要进行 11 次比较。

【答案】C

4．下列 4 种排序方法中，要求内存容量最大的是（ ）。

A．插入排序　　B．选择排序　　C．快速排序　　D．归并排序

【分析】插入排序和选择排序所需辅助空间为 O(1)，快速排序所需的辅助空间为 $O(\log_2 n)$，归并排序所需要的辅助空间为 O(n)。

【答案】D

5．下列 4 种排序方法，在排序过程中，关键码比较的次数与记录的初始排列顺序无关的是（ ）。

A．直接插入排序和快速排序　　B．快速排序和归并排序

C．直接选择排序和归并排序　　D．直接插入排序和归并排序

【分析】在备选答案中只有直接选择排序和归并排序中，关键码比较的次数与记录的初始排列顺序无关。

【答案】C

6．用某种排序方法对线性表(25,84,21,47,15,27,68,35,20)进行排序时，元素序列的变化情况如下：

1）25,84,21,47,15,27,68,35,20

2）20,15,21,25,47,27,68,35,84

3）15,20,21,25,35,27,47,68,84

4）15,20,21,25,27,35,47,68,84

则采用的排序方法是（　）。

A．选择排序　　B．希尔排序　　C．归并排序　　D．快速排序

【分析】根据各种排序方法的步骤，进行确定。

【答案】D

10-2-3　填空题

1．对一个由 n 个元素组成的序列，要借助排序过程找出其中的最大值，希望比较次数和移动次数最少，那么在下列 3 种排序方法：归并排序、直接插入排序、直接选择排序中应选择_____。

【分析】采用归并方法和直接插入方法找最大值所需的移动次数要比直接选择方法多。

【答案】直接选择排序

2．在堆排序、快速排序和归并排序中，若只从存储空间的角度考虑，则应首先选取_____方法，其次选取_____方法，最后选取 _____方法；若只从排序的稳定性考虑，则应选取_____方法；若只从平均情况下排序最快考虑，则应选取_____方法；若只从最坏情况下排序最快并且要节省内存考虑，则应选取_____方法。

【分析】堆排序的空间复杂度为 O(1)，快速排序的空间复杂度为 $O(\log_2 n)$，归并排序的空间复杂度为 O(n)；归并排序是稳定的，堆排序和快速排序是不稳定的；在平均的情况下，快速排序是最快的；在最坏的情况下，快速排序的时间复杂度为 $O(n^2)$，空间复杂度为 O(n)，而堆排序和归并排序性能基本稳定。

【答案】堆排序，快速排序，归并排序；归并排序；快速排序；堆排序

3．_____排序不需要进行记录关键字的比较。

【分析】在众多的排序方法中，基数排序不需要进行记录关键字的比较，而是通过多趟的分配和回收进行的。

【答案】基数

4．在时间复杂度为 $O(\log_2 n)$的排序方法中，_____排序方法是稳定的；在时间复杂度为 $O(n^2)$的排序方法中，_____排序方法是不稳定的。

【分析】堆排序、快速排序、归并排序等的时间复杂度均为 $O(\log_2 n)$ ，只有归并排序是稳定的；直接插入、直接选择、冒泡排序等的时间复杂度均为 $O(n^2)$，只有直接选择排序是不稳定的。

【答案】归并排序；直接选择排序

5．对 n 个记录的表 r[1..n]进行直接选择排序，所需进行的关键字间的比较次数为____。

【分析】比较次数为：(n-1)+(n-2)+…+2+1= n×(n-1)/2。

【答案】n×(n-1)/2

6．对于关键字序列(12,13,11,18,60,15,7,20,25,100)，用筛选法建堆，必须从键值为____的关键字开始。

【分析】建堆必须从第 n/2 结点开始，而 10/2＝5 位置的结点值为 60。

【答案】60

10-2-4　算法设计题

1．以单链表为存储结构，写一个直接选择排序算法。

【分析】

每趟从单链表表头开始，顺序查找当前链值最小的结点。找到后，插入到当前的有序表区的最后。

【算法】

```
void Selesort (LinkList L)
  {// 设链表 L 带头结点
   q=L;                                  // 指向第一数据前趋
   while(q->mext!=NULL)
     { pl=q->next;
       minp=pl;                          // minp 指向当前已知的最小数
       while(pl->next!=NULL)
         { if(pl->next->data<minp->data)
             minp=pl->next;              // 找到了更小数
             pl=pl->next;                // 继续往下找
           }
        if(minp!=q->next)                // 将最小数交换到第一个位置上
          { rl=minp->next;
            minp->next=rl->next;         // 删除最小数
            r2=q->next;
            q->next=r2->next;            // 删除当前表中第一个数
            rl->next=q->next;
            q->next=rl;                  // 将最小插入到第一位置上
            r2->next=minp->next;
            minp->next=r2;               // 将原第一个数放到最小数原位置上
           }
       q=q>next;                         // 选择下一个最小数
     }
}
```

2．设计一个用链表表示的直接插入排序算法。

【分析】

本算法采用的存储结构是带头结点的双循环链表。首先找元素插入位置然后把元素从原链表中删除，插入到相应的位置处。请注意双链表上插入和删除操作的步骤。

【算法】

```
void sort (dlklink  h)
   {// 链表 h 采用带头结点的双循环链表
    pre=h->next;
    while (p!=h)
      { p=pre->next;                     // 是有序表的末尾
        q=p->next;                       // 保存下一个插入元素
        while((pre!=h)&&(p->data<pre->data))
           pre=pre->prior;               // 从后往前找插入位置
        if(pre!=p->prior)
          { p->prior->next=p->next;
            p->next->prior=p>prior;      // 删除 p
```

```
            p->next=pre->next;
            pre->next->prior=p;
            p->prior=pre;pre->next=p;  // 插入到 pre 之后
           }
          p=q;
          }
}
```

3．已知$(k_1,k_2,\ldots k_n)$是堆，试写一个算法将$(k_1,k_2,\ldots k_n,k_{n+1})$调整为堆。按此思想写一个从空堆开始一个一个添入元素的建堆算法（提示：增加一个 k_{n+1} 后应从叶子向根的方向调整）。

【分析】

此问题分为两个算法，第一个算法 shift 假设 a[1],a[2],…a[k]为堆，增加一个元素 a[k+1]，把数组 a[1],a[2],…a[k+1]调整为堆。第二个算法 heep 从 1 开始调用算法 shift 将整个数组调整为堆。

【算法】

```
void  shift(datatype  a[n];int k)
   {//  a[1],a[2],…a[k] 为堆，将 a[1],a[2],…a[k+1]调整为堆，n>=k+1
    x=a[k+1];
    j=k+1;
    while((i/2>0)&&(a[i/2]>x))
       {a[i]=a[i/2];              // 从下往上找插入位置
        i=i/2;
        }
       a[i]=x;
    }
  void heap(datatype  a[n]);
    {for(k=1;k<=n-1;k++)         // 从 1 开始调用算法 shift 将整个数组调整为堆
       shift(a,k);
    }
```

4．一个线性表中的元素为正整数或负整数。设计一个算法，将正整数或负整数分开，使线性表前一半为负整数，后一半为正整数。不要求对这些元素排序，但要求尽量减少交换次数。

【分析】

先设置好上、下界，然后分别从线性表两端查找正数和负数，找到后进行交换，直到上、下界相遇为止。

【算法】

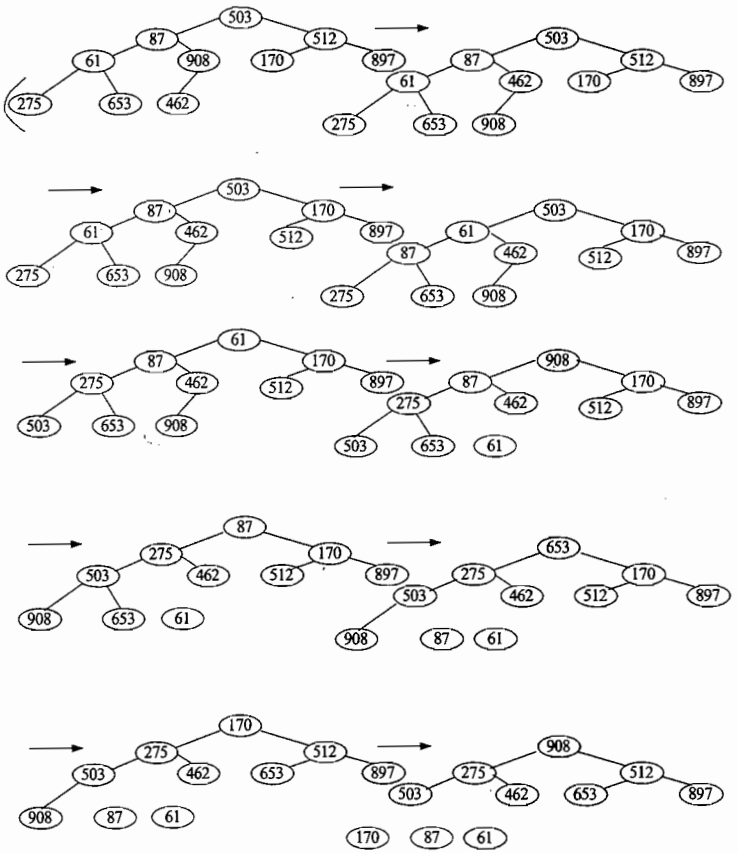
```
void example (datatype a[n])
     { i=1,j=n;                          // i,j 为左右边界
       while(i<j)
          {while((i<j)&&(a[j]<0))
              i++;                       // 在左边界找正数
           while ((i<j)&&(a[j]>0))
              j--;                       // 在右边界找负数
```

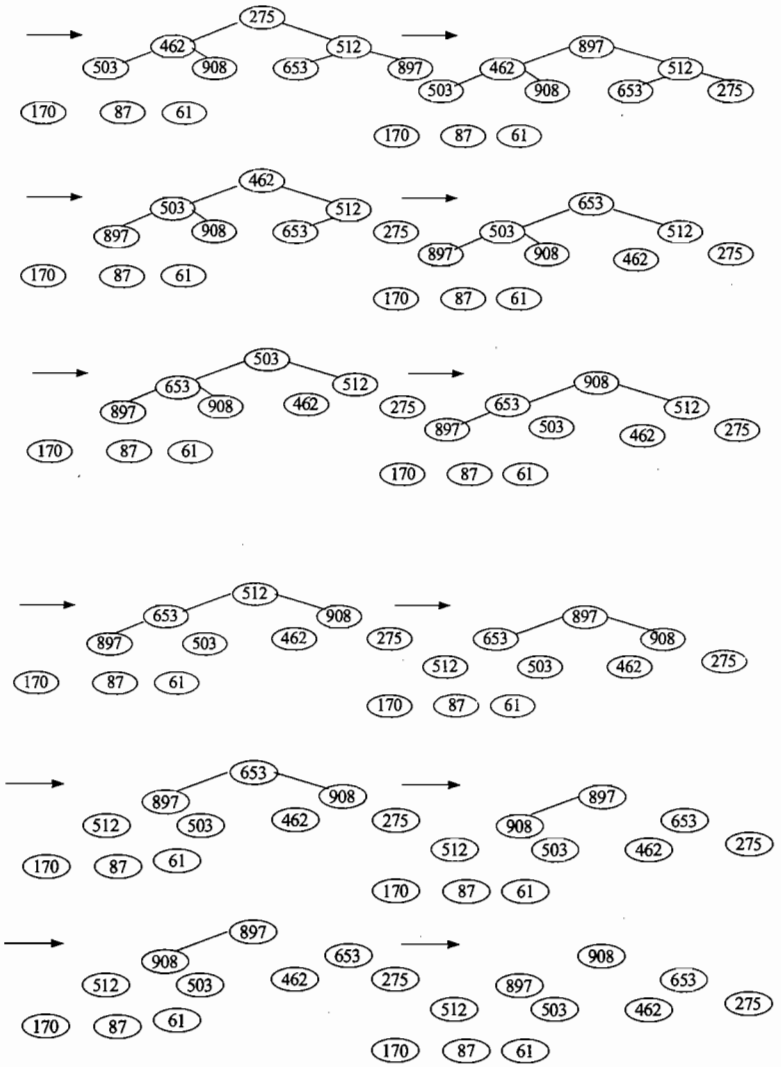


```
          if(i<j)
            { temp=a[i];
              a[i]=a[j];
              a[j]=a[temp];              // 交换两个元素的值
              i++;
              j--;
            }
       }
}
```

10-3　课后习题选解

1．已知序列{503,87,512,61,908,170,897,275,653,462}，请给出采用堆排序对该序列做升序排序时的每一趟结果。





2．有 n 个不同的英文单词，它们的长度相等，均为 m，若 n>>50，m<5，试问采用什么排序方法时间复杂度最佳？为什么？

【提示】采用基数排序。

3．如果只想得到一个序列中第 k 小元素前的部分排序序列，最好采用什么排序方法？为什么？如有这样一个序列：{57,40,38,11,13,34,48,75,25,6,19,9,7}得到其第 4 个最小元素之前的部分序列{6,7,9,11}，使用所选择的算法实现时，要执行多少次比较？

【提示】采用堆排序。

4．以单链表为存储结构，写一个直接选择排序算法。

（参考典型例题解析中的算法设计题 1）

5．请以单链表为存储结构实现直接插入排序的算法。

【提示】注意在链式结构上插入元素的实现方式。

```
typedef   struct
{ KeyType  key;
  …
}ElemType;
typedef   struct  Lnode
{ ElemType  data;
      struct  LNode  *next;
}Lnode ,*LinkList;
LinkList   sort (ListList  L)          // L 为带头结点的单链表
{ p=L->next->next;                     // p 指向第二个结点
L->next->next=NULL;                    // 设置 L 为只含有一个结点的单链表
while(p)  // 当结点尚未插入结束时，将结点 p 逐个插入到单链表 L 中去，同时要保持其有序性
      {   s=p->next;
          pre=L ; q=L->next;
          while ( q && p->data.key > q->data.key)
          {pre=q;  q=q->next; }        // 寻找插入位置
          p->next=q;    pre->next=p;   // 插入元素
          p=s;
      }
  return(L)
}
```

6．编写一个双向冒泡的算法，即相邻两边向相反方向冒泡。

【提示】注意算法的结束条件。

```
typedef   struct
{ KeyType  key;
      …
}ElemType;
typedef   struct
{ ElemType   *elem;
  int  length;
}S_table;
void  sort(S_table *L)
{ flag = 1 ;j = 1
  while (flag)
  {   flag = 0 ;
      for( i=j ; i<=L->length-j; i++)                    // 正向冒泡
          if (L->elem[i].key>L->elem[i+1].key )
              {   L->elem [i] <=>L->elem[i+1] ;
                  flag=1 ;}
      for (i=L->length-j; i>=j+1; i--)                   // 反向冒泡
          if (L->elem[i].key<L->elem[i-1].key)
              {   L->elem[i]<=> L->elem[i-1] ;
                  flag=1; }
```

```
    j++;
    }
    return;
    }
```

10. 已知记录序列 a[1..n]中的关键字各不相同，可按如下所述实现计数排序：另设数组 c[1..n]，对每个记录 a[i]，统计序列中关键字比它小的记录个数存于 c[i]，则 c[i]=0 的记录必为关键字最小的记录，然后依 c[i]值的大小对 a 中记录进行重新排列，试编写实现上述排序的算法。

【提示】注意对 a 中记录进行重新排列的算法实现。

```
typedef   struct
{ KeyType   key;
     …
}ElemType;
void  sort (ElemType  *A,  ElemType  *B,  int  n)
{ int  C[n+1] ;
  for (i=1 ;i<=n; i++)  C[i] = 0;
  for (i=1; i<=n ; i++)
      { for (j=1 ; j<=n ; j++)
          if(A[j].key<A[i].key)   C[i] ++;     //统计每个记录按关键字大小的次序
      }
  for (i=1 ; i<=n; i++)
      B[C[i]+1] = A[i] ;                       //重排记录
  return;
}
```

11. 已知奇偶交换排序算法如下描述：第一趟对所有奇数的 i，将 a[i]和 a[i+1]进行比较，第二趟对所有偶数的 i，将 a[i]和 a[i+1]进行比较，每次比较时若 a[i]>a[i+1]，则将二者交换，以后重复上述二趟过程，直至整个数组有序。

（1）试问排序结束的条件是什么？

（2）编写一个实现上述排序过程的算法。

【提示】排序结束的条件是没有交换发生。

```
void sort ( ElemType   A[n+1])                 // 0 号单元不用
{ flag = 1 ;
  while ( flag )
  { flag=0;
      for ( i=1; i+1<=n ; i=i+2)               // 奇数 i
          if (A[i].key>A[i+1].key )
              { flag=1;  A[i]<=>A[i+1]  ;}
      for ( i=0; i+1<=n ; i=i+2)               // 偶数 i
          if (A[i].key>A[i+1].key )
              { flag=1;  A[i]<=>A[i+1]  ;}
      }
  return;
}
```

12. 编写算法，对 n 个关键字取整数值的记录进行整理，以使得所有关键字为负值的记录排在关键字为非负值的记录之前，要求：

（1）采用顺序存储结构，至多使用一个记录的辅助存储空间；

（2）算法的时间复杂度为 O(n)；

（3）讨论算法中记录的最大移动次数。

【提示】采用类似快速排序的算法来实现。

```
void  process (int A[n])
{ low = 0 ; high = n-1;
      while ( low<high )
      { while (low<high  &&  A[low] <0)   low++;
          while (low<high  &&  A[high]>0 )high++;
              if (low<high)   {   A[low] <=>A[high];
                                  low++ ;high-- ;}
      }
      return;
      }
```

13. 序列的“中值记录”指的是：如果将此序列排序后，它是第 n/2 个记录。试编写一个求中值记录的算法。

【提示】注意不要排序。采用类似快速排序的算法实现：以某个序列中第一个元素为基准分割序列为两个子序列，判断最后基准值的位置，若为 n/2，则找到中值记录；若大于 n/2，则在前半区间继续查找；若小于 n/2 则在后半区间继续查找。

```
int  fun (S_table  *L,  int  low, int  high)
{ // 以第一个元素为基准分割下标在区间[low, high]内的元素序列
  L->elem[0] = L->elem[low] ;
  K=L->elem[0].key;
  while  (low<high)
      { while(low<high && L->elem[high]>=k)  high--;
          L->elem[low] = L->elem[high] ;
          while (low<high && L->elem[low]<=k)    low++;
          L->elem[high] = L->elem[low] ;
      }
  L->elem[low] =L->elem[0];
  return(low) ;
}
ElemType middle(S_table*L,  int low ,  int high )
{ if (low<high)
      {   s = fun(L,low,high);
      if(s==(high-low)/2 )     return (L->elem[s]) ;       //找到中值记录
          if(s>(high-low)/2)   return(middle(L,low,s-1)) ; //在前半区间继续查找
                       else    return(middle(L,s+1,high)); //在后半区间继续查找
      }
}
```

第 11 章　实验的一般步骤

11-1　概述

数据结构的上机实验是对学生的一种全面的综合训练，与单纯的编程语言教学时的验证性的上机实验不同，这门课程的实验多是创造性的活动。实验中要解决的问题更接近于实际，也要比理论上的习题更复杂一点。实验是一种自主性很强的学习过程，一方面要着眼于原理与应用的结合，使学生学会如何把书上学到的知识用于解决实际问题，培养软件工作原理与应用的结合，使读者学会如何把书上学到的知识用于解决实际问题，培养软件工作所需要的动手能力，另一方面，能使书上的知识变“活”，起到深化理解和灵活掌握教学内容的目的。平时的练习较偏重如何编写功能单一的“小”算法，而实验是软件设计的综合训练，包括问题分析、总体结构设计、用户界面设计、程序设计基本技能和技巧，多人合作，以至一整套软件工作规范的训练和科学作风的培养。

为了达到上述目的，本书安排了 7 个实验题目，一个实验题目要涉及几部分的教学内容。每个实验题目采取了统一的格式，由问题描述、数据结构设计、功能（函数）设计、界面设计、编码实现、运行与测试几个部分组成。其中问题描述是为学生建立问题提出的背景环境，指明问题“是什么”并对实验要实现的基本要求做一布置。数据结构设计是在问题描述的基础上将实际问题中需要用到的具体数据抽象成数据类型模型，并与教材中的数据结构对应给出相应的定义。功能（函数）设计是对具体问题进一步的分析，将一个“大问题”拆分成若干个容易解决的小模块，并确定他们的调用关系和模块的出口与入口。界面设计是对各个实验题目中用户的操作界面提出的一些简略的要求。编码实现部分主要由学生来完成，就不再给出。运行与测试部分是给出一些实际测试的数据，用来检查学生所编制程序的正确性、健壮性等。

虽然各个题目给出的分析和要求比较详细，但是需要注意的是，这些分析和要求不应该限制学生的想象力和创造力。对于一个实际问题，每个人可能会有不同的解决思路，本书所给出的问题说明和问题分解的范例，只是希望能够把学生的思路引到正轨上来，在无形中学会模仿，同时也传授了系统划分方法和程序设计的一些具体技术，保证实现预定的训练意图，使某些难点和重点不会被绕过去，而且也便于教学检查。但需要强调的是，学生不应拘泥于此，而应努力开发更好的方法和结构。

如何高效地利用机时是一个普遍存在的问题。调试程序绝不是凭运气的。调试程序是一个艰苦但是充满乐趣的过程。很多学生都有过这样的经历：花了一两个小时的上机时间却连一个错误都没有改正过来。为什么？一方面，对调试工具的使用不熟悉，出现了错误提示却不知道这种错误是如何产生的，当然，这种问题不属于数据结构要重点解决的问题，在此不再赘述；另一方面，很多人只认识到找错误，而没有认识到努力预先避免错误的重要性，也不知道应该如何努力。实际上，结构不好、思路和概念不清的程序可能是根本无法调试正确的。严格按照实验步骤规范进行实验不但能有效地避免上述种种问题，更重要的是有利于培养软件工作者不可缺少的科学工作方法和作风。下面就给出实验步骤的严格规范。

11-2　实验步骤

1．问题描述和问题分析

通常，实验题目的陈述比较简洁，或者说是有模棱两可的含义。因此，在进行设计之前，首先应该充分地分析和理解问题，明确问题要求做什么，限制条件是什么。注意：本步骤强调的是做什么，而不是怎么做。对问题的描述应避开算法和所涉及的数据类型，而是对所需完成的任务做出明确的回答。例如：输入数据的类型、值的范围以及输入的形式；输出数据的类型、值的范围及输出的形式；若是会话式的输入，则结束标志是什么，是否接受非法的输入，对非法输入的回答方式是什么等。这一步还应该为调试程序准备好测试数据，包括合法的输入数据和非法的输入数据。

2．数据类型和系统设计

在设计这一步骤中需分逻辑设计和详细设计两步实现。逻辑设计指的是，对问题描述中涉及的操作对象定义相应的数据类型，并按照以数据结构为中心的原则划分模块，定义主程序模块和各抽象数据类型；详细设计则为定义相应的存储结构并写出各函数的伪码算法。在这个过程中，要综合考虑系统功能，使得系统结构清晰、合理、简单和易于调试，抽象数据类型的实现尽可能做到数据封闭，基本操作的规格说明尽可能明确具体。作为逻辑设计的结果，应写出每个抽象数据的定义（包括数据结构的描述和每个基本操作的规格说明），各个主要模块的算法，并画出模块之间的调用关系图。详细设计的结果是对数据结构和基本操作的规格说明做出进一步的优化，写出数据存储结构的类型定义，按照算法书写规范用类 C 语言写出函数形式的算法框架。在优化的过程中，应尽量避免陷入语言细节，不必过早表述辅助数据结构和局部变量。

3．编码实现和静态检查

编码是把详细设计的结果进一步优化为程序设计语言程序。程序的每行不要超过 60 个字符。每个函数体，即不计首部和规格说明部分，一般不要超过 40 行，最长不得超过 60 行，否则应该分割成较小的函数。要控制 if 语句连续嵌套的深度。其他要求参见第一篇的算法书写规范。如何编写程序才能较快地完成调试是特别要注意的问题。对于编程熟练的读者，如果基于详细设计的伪码算法就能直接在键盘上输入程序的话，则可以不必用笔在纸上写出编码，而将这一步的工作放在上机准备之后进行，即在上机调试之前直接用键盘输入。

然而，不管你是否写出编码的程序，在上机之前，认真的静态检查是必不可少的。多数初学者在编好程序后处于以下两种状态之一：一种是对自己的“精心作品”的正确性确信不疑；另一种是认为上机前的任务已经完成，纠查错误是上机的工作。这两种态度是极为有害的。事实上，非训练有素的程序设计者编写的程序长度超过 50 行时，极少不含有除语法错误以外的错误。上机动态调试决不能代替静态检查，否则调试效率将是极低的。

静态检查主要有两种方法，一是用一组测试数据手工执行程序（通常应先分模块检查）；二是通过阅读或给别人讲解自己的程序而深入全面地理解程序逻辑，在这个过程中再加入一些注解。如果程序中逻辑概念清楚，后者将比前者有效。

4．上机准备和上机调试

上机准备包括以下几个方面：

（1）高级语言文本（体现于编译程序用户手册）的扩充和限制。例如，常用的 Borland C（C++）和 Visual C（C++）与标准 C（C++）的差别，以及相互之间的差别。

（2）如果使用 C 或 C++语言，要特别注意与教科书的类 C 语言之间的细微差别。

（3）熟悉机器的操作系统和语言集成环境的用户手册，尤其是最常用的命令操作，以便顺利进行上机的基本活动。

（4）掌握调试工具，考虑调试方案，设计测试数据并手工得出正确结果。“磨刀不误砍柴工”。计算机各专业的学生应该能够熟练运用高级语言的程序调试器 DEBUG 调试程序。

上机调试程序时要带一本高级语言教材或手册。调试最好分模块进行，自底向上，即先调试低层函数。必要时可以另写一个调用驱动程序。这种表面上麻烦的工作实际上可以大大降低调试所面临的复杂性，提高调试工作效率。

在调试过程中可以不断借助 DEBUG 的各种功能，提高调试效率。调试中遇到的各种异常现象往往是预料不到的，此时不应“冥思苦想”，而应动手确定疑点，通过修改程序来证实它或绕过它。调试正确后，认真整理源程序及其注释，印出带有完整注释的且格式良好的源程序清单和结果。

5. 总结和整理实验报告

第 12 章　实验安排

实验一　Josephus 环问题

本实验的目的是进一步理解线性表的逻辑结构和存储结构，进一步提高使用理论知识指导解决实际问题的能力。

一、问题描述

设编号为 1、2、…n 的 n 个人围坐一圈，约定编号为 k（1≤k≤n）的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

二、数据结构设计

1．由于当某个人退出圆圈后，报数的工作要从下一个人开始继续，剩下的人仍然是围成一个圆圈的，可以使用循环表，由于退出圆圈的工作对应着表中结点的删除操作，对于这种删除操作频繁的情况，选用效率较高的链表结构，为了程序指针每一次都指向一个具体的代表一个人的结点而不需要判断，链表不带头结点。所以，对于所有人围成的圆圈所对应的数据结构采用一个不带头结点的循环链表来描述。设头指针为 p，并根据具体情况移动。

可以采用数据类型定义：

```
Typedef struct node
{
Int number;          //每个人的编号
Struct node *next;   //指向表示下一个人的结点的指针
}Lnode, *Llinlist
Linklist p;
```

2．为了记录退出的人的先后顺序，采用一个顺序表进行存储。程序结束后再输出依次退出的人的编号顺序。由于只记录各个结点的 number 值就可以，所以定义一个整型一维数组。如：Int quit[N]；N 为一个根据实际问题定义的一个足够大的整数。

三、功能（函数）设计

由于函数功能比较简单，对子模块可以不再细分，而使用主函数 main( )做链表和数组的初始化工作，使用函数 Josephus( )做删除结点和输出顺序的工作。

四、界面设计

注意提示用户每一步操作输入的格式和限制。指导用户按照正确的格式输入数据。

五、编码实现

略。

六、运行与测试

1．当 m 的初值为 5，n 的初值为 7，从第 1 个人开始数，则正确的出列顺序应为 5、3、2、4、7、1、6，经程序运行测试，结果正确。

2．程序容错性的测试，当输入 k 的值不符合问题约定时应有错误提示给用户，指导用户正确输入，并做出相应处理保证程序正常运行。

实验二　一元多项式相加问题

本实验的目的是进一步熟练掌握应用链表处理实际问题的能力。

一、问题描述

一元多项式相加是通过键盘输入两个形如 $P_0+P_1X^1+P_2X^2+\ldots+P_nX^r$ 的多项式，经过程序运算后在屏幕上输出它们的相加和。

二、数据结构设计

分析任意一元多项式的描述方法可知，一个一元多项式的每一个子项都是有由“系数－指数”两部分来组成的，所以可以将它抽象成一个由“系数－指数对”构成的线性表，由于对多项式中系数为 0 的子项可以不记录它的指数值，对于这样的情况就不再付出存储空间来存放它了。基于这样的分析，我们可以采用一个带有头结点的单链表来表示一个一元多项式。具体数据类型定义为：

```
Typedef struct node
{
Float coef;          //系数域
Int exp;             //指数域
Int flag;            //标记结点是否已经过运算加入新的链表的标志位
Struct node next;    //指针域 指向下一个系数不为 0 的子项
}PolyNode;
PolyNode *head_a,*head_b,*head_c;
```

这 3 个指针分别作为链表 A、B、C 的头指针。

三、功能（函数）设计

1．输入并建立多项式的功能模块

此模块要求按照指数递增的顺序和一定的输入格式输入各个系数不为 0 的子项的“系数－指数对”，输入一个子项建立一个相关结点，当遇到输入结束标志的时候就停止输入，而转去执行程序下面的部分。

例如按照这些操作步骤输入多项式：

屏幕提示：

```
请输入第一个多项式，按照“系数，指数”的形式输入每一个子项！
1，1
2，2
－3，100
0，0　（输入结束的条件，当系数和指数同时为 0 的时候认为输入结束）
请输入第二个多项式，按照“系统，指数”的形式输入每一个子项！
100，1
200，2
300，200
0，0
```

输入后程序将分别建立两个链表来描述两个一元多项式：

```
A(X)=X+2X2-3X100
B(X)=100X+200X2+300X200
这两个多项式的相加结果应该为:
C(X)=101X+202X2-3X100+300X200
```

2．多项式相加的功能模块

此模块根据在 1 中建立的两个多项式进行相加的运算，并存放在以 C 为头指针的一个新链表中。可以采用如下的方法进行设计：

设指针 p、q、r 分别指向描述多项式的链表 A、B、C 头部，p 指针按照 A 中结点顺序一个一个地移动，每移动一个结点，就通过 q 指针的移动来寻找 B 中是否有与 p->exp 相同的。

（1）如果有，在 C 中合适的位置（注意多项式存储是按照指数递增有序的）建立新结点，并做如下赋值操作：

```
r->coef=p->coef+q->coef;
r->exp=p->exp;
p->flag=1;
q->flag=1;  /*标志子项已参与过运算*/
p=p->next;
q=head_b;
```

（2）如果没有，在 C 中合适的位置（注意多项式存储是按照指数递增有序的）建立新结点，并做如下赋值操作：

```
r->coef=p->coef;
r->exp=p->exp;
p->flag=1;
p=p->next;
q=head_b;
```

注意：当 A、B 中可以相加的两项系数和为 0 的时候，在 C 中就不用分配新的空间来进行存储了。

3．多项式显示的功能模块

此模块用于多项式的显示，程序可以使用图形界面，通过调整指数应该出现的坐标位置来表示指数形式，如 $X+2X^2-3X^{100}$ 的形式，也可以使用文本界面，用“系数－指数对”的形式表示表达式，如(1，1)、(2，2)、(-3，100)。

四、界面设计

注意提示用户每一步操作输入的格式和限制。指导用户按照正确的格式输入数据。

五、编码实现

略。

六、运行与测试

1．测试以下数据，比较结果：

（1）、$(2x+5x^8-3.1x^{11})+(7-5x^8+11x^9)=(-3.1x^{11}+11x^9+2x+7)$

（2）、$(x+x^3)+(-x-x^3)=0$

（3）、$(x+x^2+x^3)+0=x+x^2+x^3$

（4）、互换上述测试数据中的前后两个多项式。

2．不按指数递增的顺序输入多项式，验证程序的健壮性和容错性。

实验三　停车厂模拟管理程序的设计与实现

本实验的目的是进一步理解栈和队列的逻辑结构和存储结构，进一步提高使用理论知识指导解决实际问题的能力。

一、问题描述

设停车厂只有一个可停放几辆汽车的狭长通道，且只有一个大门可供汽车进出。汽车在停车场内按车辆到达的先后顺序依次排列，若车场内已停满几辆汽车，则后来的汽车只能在门外的便道上等候，一旦停车场内有车开走，则排在便道上的第一辆车即可进入；当停车场内某辆车要离开时，由于停车场是狭长的通道，在它之后开入的车辆必须先退出车场为它让路，待该车辆开出大门，为它让路的车辆再按原次序进入车场。在这里假设汽车不能从便道上开走，试设计这样一个停车厂模拟管理程序。为了以下描述的方便，停车厂的停车场用“停车位”进行叙述，停车厂的便道用“便道”进行叙述。

二、数据结构设计

1．为了便于区分每辆汽车并了解每辆车当前所处的位置，需要记录汽车的牌照号码和汽车的当前状态，所以为汽车定义一个新的类型 CAR，具体定义如下：

```
typedef struct
{
char *license_plate;     //汽车牌照号码，定义为一个字符指针类型
char state;              //汽车当前状态，字符 s 表示停放在停车位上，
                         //字符 p 表示停放在便道上，每辆车的初始状态用字符 i 来表示
}
```

2．由于车位是一个狭长的通道，所以不允许两辆车同时出入停车位，当有车到来要进入停车位的时候也要顺次停放，当某辆车要离开时，比它后到的车要先暂时离开停车位，而且越后到的车就越先离开停车位，显然这和栈的“后进先出”特点相吻合，所以可以使用一个栈来描述停车位。

由于停车位只能停放有限的几辆车，而且为了便于停车厂的管理，要为每个车位分配一个固定的编号，不妨设为 1、2、3、4、5（可利用数组的下标），分别表示停车位的 1 车位、2 车位、3 车位、4 车位、5 车位，针对这种情况使用一个顺序栈比较方便，具体定义如下：

```
#define MAX_STOP 5
typedef struct
{
CAR STOP[MAX_STOP]; //各汽车信息的存储空间
int top;            //用来指示栈顶位置的静态指针
}STOPPING;
```

3．当停车场的停车位上都已经停满了汽车，又有新的汽车到来时要把它调度到便道上，

便道上的车辆要按照进入便道的先后顺序顺次存放在便道上，为便道上的每个位置也分配一个固定的编号，当有车从停车位上离开后，便道上的第一辆汽车就立即进入停车位上的某个车位，由于问题描述中限制了便道上的汽车不能从便道上开走，即便道上的汽车只有在停车位上停放过之后才能离开停车厂，这样越早进入便道的汽车就越早进入停车位，而且每次进入停车位的汽车都是处于便道“最前面”的汽车，显然，这和队列的“先进先出”特点相吻合，所以，这里使用一个顺序队来描述便道，可以利用数组的下标表示便道的位置，具体定义如下：

```
#define MAX_PAVE 100     //便道不限制停放车辆的数目，设为足够大
typedef struct
{
CAR PAVE[MAX_PAVE];      //各汽车信息的存储空间
int front,rear;          //用来指示队头和队尾位置的静态指针
}PAVEMENT;
```

4．当某辆车要离开停车厂的时候，比它后进停车位的车要为它让路，而且当它开走之后让路的车还要按照原来的停放次序再次进入停车位的某个车位上，为了完成这项功能，再定义一个辅助栈，停车位中让路的车依次“压入”辅助栈，待提出开走请求的车开走后再从辅助栈的栈顶依次“弹出”到停车位中。对辅助栈也采用顺序栈，具体定义与停车位栈类似，如下：

```
typedef struct
{
CAR BUFFER[MAX_STOP];    //各汽车信息的存储空间
int top;                 //用来指示栈顶位置的静态指针
}BUFFER;
```

当然，辅助栈直接利用在 2 中定义的类型 STOPPING 也是可以的。

由于程序的各函数要对这些数据结构中的数据进行操作，而且每次操作的结果都要动态反应到以上数据结构中，所以在程序设计的时候使用以上新类型定义的变量都采用全局变量的形式。

三、功能（函数）设计

1．本程序从总体上分为 4 个大的功能模块，分别为：程序功能介绍和操作提示模块、汽车进入停车位的管理模块、汽车离开停车位的管理模块、查看停车厂停车状态的查询模块，具体功能描述如下：

1）程序功能介绍和操作提示模块：此模块给出程序欢迎信息，介绍本程序的功能，并给出程序功能所对应的键盘操作的提示，具体屏幕显示如下所示：

```
      ● 欢迎使用本程序。●
本程序为停车场的模拟管理程序，有车到来时请按C键。
然后根据屏幕提示进行相关操作，有车要走时请按 1 键。
然后根据屏幕提示进行相关操作，要退出程序请按Q键。

请选择您要做的操作!
```

函数原型为 void welcome( )；

2）汽车进入停车位的管理模块：此模块用来登记停车厂的汽车的车牌号和对该车的调

度过程并修改该车的状态，其中调度过程要以屏幕信息的形式反馈给用户来指导用户对车辆的调度。例如，当前停车位上 1、2、3 车位分别停放着牌照为 JF001、JF002、JF003 的汽车，便道上无汽车，当牌照为 JF004 的汽车到来后屏幕应给出如下提示信息：

```
牌照为 JF004 的汽车停入停车位的 4 号车位!

按回车键继续程序的运行。
```

函数原型为 void come( );

此函数还要调用其他对于栈和队列的基本操作。

3）汽车离开停车位的管理模块：此模块用来为提出离开停车厂的车辆做调度处理，并修改相关车辆的状态，其中调度过程要以屏幕信息的形式反馈给用户来指导用户对车辆的调度，当有车离开停车厂后应该立刻检查便道上是否有车，如果有车的话立即让便道上的第一辆汽车进入停车位。例如，当前停车位上 1、2、3、4、5 车位分别停放着牌照为 JF001、JF002、JF003、JF004、JF005 的汽车，便道上的 1、2 位置分别停放着牌照为 JF006、JF007 的汽车，当接收到 JF003 要离开的信息时，屏幕应给出如下提示信息：

```
牌照为 JF005 的汽车暂时退出停车位;
牌照为 JF004 的汽车暂时退出停车位;
牌照为 JF003 的汽车从停车厂开走;
牌照为 JF004 的汽车停回停车位的 3 车位;
牌照为 JF005 的汽车停回停车位的 4 车位;
牌照为 JF006 的汽车从便道上进入停车位的 5 车位;

按回车键继续程序的运行。
```

函数原型为 void leave( );

此函数还要调用其他对于栈和队列的基本操作。

4）查看停车厂停车状态的查询模块：此模块用来在屏幕上显示停车位和便道上各位置的状态，例如，当前停车位上 1、2、3、4、5 车位分别停放着牌照为 JF001、JF002、JF003、JF004、JF005 的汽车，便道上的 1、2 位置分别停放着牌照为 JF006、JF007 的汽车，当接受到查看指令后，屏幕上应显示：

```
停车位的情况:
1 车位——JF001;
2 车位——JF002;
3 车位——JF003;
4 车位——JF004;
5 车位——JF005;
便道上的情况:
1 位置——JF006;
2 位置——JF007;

按回车键继续程序的运行。
```

函数原型为 void display( );

此函数还要调用其他对于栈和队列的基本操作。

2．以上 4 个总体功能模块要用到的栈和队列的基本操作所对应的主要函数如下表所示：

函 数 原 型	函 数 功 能
STOPPING *init_stopping( )	初始化“停车位栈”
BUFFER *init_buff( )	初始化“辅助栈”
PAVEMENT *init_pavement( )	初始化“便道队列”
Int car_come(int pos)	将 pos 指定的汽车信息插入“停车位栈”，并修改该车状态
Int car_leave(int pos)	将 pos 指定的汽车信息从“停车位栈”删除，并修改该车状态
Int stop_to_buff(int pos)	将 pos 指定的汽车信息从“停车位栈”移动到“辅助栈”
Int buff_to_stop(int pos)	将 pos 指定的汽车信息从“辅助栈”移动到“停车位栈”
Int pave_to_stop(int pos)	将 pos 指定的汽车信息从“便道队列”移动到“停车位栈”
Int car_disp(int pos)	将 pos 指定的汽车信息显示在屏幕上

其他函数的定义和说明请参照源代码。

3．由于程序应该能够随时处理用户所提出的各种操作请求，所以在主函数中用一个 DO_WHILE 循环结构随时监控键盘的按键操作，遇到相应的按键就转到对应函数继续运行，运行完该函数继续监控键盘按键，如此往复，直到接到“退出”指令程序才能结束。部分编码如下：

```
welcome( );
flushall( );
do
{
    key=getchar( );
    if(key=='C'||key=='c')
        come( );
    else    if(key=='L'||key=='l')
            leave( );
        else welcome( );
}
while((key!='Q')&&(key!='q'));
```

四、界面设计

本程序的界面力求简洁、友好，每一步需要用户操作的提示以及每一次用户操作产生的调度结果都以中文的形式显示在屏幕上，使用户对要做什么和已经做了什么一目了然。文字表述精练、准确。具体设计可参阅功能设计中的相关部分，这里就不再赘述。

五、编码实现

略。

六、运行与测试

对于测试用的设计注重所定义的数据结构的边界以及各种数据结构共存的可能性。例如：

1．连续有 7 辆车到来，牌照号分别为 JF001、JF002、JF003、JF004、、JF005、JF006、JF007，前 5 辆车应该进入停车位 1～5 车位，第 6、7 辆车应停入便道的 1、2 位置上。

2．1 中的情况发生后，让牌照号为 JF003 的汽车从停车厂开走，应显示 JF005、JF004 的让路动作和 JF006 从便道到停车位上的动作。

3．随时检查停车位和便道的状态，不应该出现停车位有空位而便道上还有车的情况。

4．其他正常操作的一般情况。

5．程序容错性的测试，当按键输入错误的时候是否有错误提示给用户指导用户正确操作，并做出相应处理保证程序正常运行。

实验四　农夫过河问题的求解

本实验的目的是进一步理解顺序表和队列的逻辑结构和存储结构，进一步提高使用理论知识指导解决实际问题的能力。

一、问题描述

一个农夫带着一只狼、一只羊和一棵白菜，身处河的南岸。他要把这些东西全部运到北岸。他面前只有一条小船，船只能容下他和一件物品，另外只有农夫才能撑船。如果农夫在场，则狼不能吃羊，羊不能吃白菜，否则狼会吃羊，羊会吃白菜，所以农夫不能留下羊和白菜自己离开，也不能留下狼和羊自己离开，而狼不吃白菜。请求出农夫将所有的东西运过河的方案。

二、数据结构设计

求解这个问题的简单方法是一步一步进行试探，每一步搜索所有可能的选择，对前一步合适的选择再考虑下一步的各种方案。

要模拟农夫过河问题，首先需要对问题中每个角色的位置进行描述。一个很方便的办法是用 4 位二进制数顺序分别表示农夫、狼、白菜和羊的位置。用 0 表示农夫或者某东西在河的南岸，1 表示在河的北岸。例如整数 5（其二进制表示为 0101）表示农夫和白菜在河的南岸，而狼和羊在北岸。

现在问题变成：从初始状态二进制 0000（全部在河的南岸）出发，寻找一种全部由安全状态构成的状态序列，它以二进制 1111（全部到达河的北岸）为最终目标，并且在序列中的每一个状态都可以从前一状态到达。为避免瞎费功夫，要求在序列中不出现重复的状态。

实现上述求解的搜索过程可以采用两种不同的策略：一种是广度优先（breadth_first）搜索，另一种是深度优先（depth_first）搜索。本书只介绍在广度优先搜索方法中采用的数据结构设计。

广度优先就是在搜索过程中总是首先搜索下面一步的所有可能状态，再进一步考虑更后面的各种情况。要实现广度优先搜索，可以使用队列。把下一步所有可能的状态都列举出来，放在队列中，再顺序取出来分别进行处理，处理过程中把再下一步的状态放在队列里……。由于队列的操作遵循先进先出的原则，在这个处理过程中，只有在前一步的所有情况都处理完后，才能开始后面一步各种情况的处理。这样，具体算法中就需要用一个整数队列 moveTo，它的每个元素表示一个可以安全到达的中间状态。另外还需要一个数据结构记录已被访问过的各个状态，以及已被发现的能够到达当前这个状态的路径。由于在这个问题的解决过程中需要列举的所有状态（二进制 0000 到 1111）一共 16 种，所以可以构造一个包含 16 个元素的整数顺序表来实现。顺序表的第 i 个元素记录状态 i 是否已被访问过，若已被访问过则在这个顺序表元素中记入前驱状态值，把这个顺序表叫做 route。route 的每个分量初始值均为-1。route 的一个元素具有非负值表示这个状态已访问过，或是正被考虑。最后可以利用 route 顺序表元素的值建立起正确的状态路径。于是得到农夫过河问题

的广度优先算法。

在具体应用时，采用链队和顺序队均可，为叙述的方便，不妨设为使用顺序队。

三、功能（函数）设计

1．确定农夫、狼、羊和白菜位置的功能模块。

用整数 location 表示上述 4 位二进制描述的状态，由于采用 4 位二进制的形式表示农夫、狼、白菜和羊，所以要使用位操作的“与”操作来考察每个角色所在位置的代码是 0 还是 1。函数返回值为真表示所考察的角色在河的北岸，否则在南岸。例如某个状态和 1000 做“与”操作后所得结果为 0，则说明农夫的位置上的二进制数为 0，即农夫在南岸，如果所得结果为 1，则说明农夫的位置上的二进制数为 1，即农夫在北岸。狼、羊和白菜的处理办法以此类推。可以如下设计函数：

```
int farmer(int locat on) {
    return (0 != (location & 0x08));
}
int wolf(int location) {
    return (0 != (location & 0x04));
}
int cabbage(int location) {
    return (0 != (location & 0x02));
}
int goat(int location) {
    return (0 !=(location & 0x01));
}
```

2．确定安全状态的功能模块。

此功能模块通过位置分布的代码来判断当前状态是否安全。若状态安全返回 1，状态不安全返回 0。可以如下设计函数：

```
int safe(int location)  // 若状态安全则返回 1
{   if((goat(location) == cabbage(location)) &&(goat(location)!=farmer(location)))
    return (0);          // 羊吃白菜
        if((goat(location) == wolf(location)) && (goat(location)!= farmer(location)))
      return (0);        // 狼吃羊
return (1);              // 其他状态是安全的
}
```

3．将各个安全状态还原成友好的提示信息的功能模块。

由于程序中 route 表中最终存放的是整型的数据，如果原样输出不利于最终用户理解问题的解决方案，所以要把各个整数按照 4 位二进制的各个位置上的 0、1 代码所表示的含义输出成容易理解的文字。

如遇到状态 9，则在屏幕上显示输出相关的信息。

```
                南岸                    北岸
狼        白菜        农夫        羊
```

四、界面设计

如果能力和时间允许，可以使用动画设计将运送的过程演示出来。一般情况下可以使用最终的状态表描述出来就可以了。

五、编码实现

略

六、运行与测试

使用状态表，程序应在屏幕上得出如下结果：

步骤	状态		注释（此内容不显示）
	南岸	北岸	
0	农夫 狼 白菜 羊		
1	狼 白菜	农夫 羊	农夫带羊从南岸到北岸
2	狼	农夫 白菜 羊	农夫自己从北岸回南岸再带白菜从南岸到北岸
3	农夫 狼 羊	白菜	农夫带羊从北岸回南岸
4	羊	农夫 狼 白菜	农夫带狼从南岸到北岸
5	农夫 羊	狼 白菜	农夫自己回南岸
6		农夫 狼 白菜 羊	农夫带羊到北岸（问题解决完毕）

实验五　简单哈夫曼编/译码的设计与实现

本实验的目的是通过对简单哈夫曼编/译码系统的设计与实现来熟练掌握树型结构在实际问题中的应用。此实验可以作为综合实验，阶段性实验时可以选择其中的几个功能来设计和实现。

一、问题描述

利用哈夫曼编码进行通信可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码，在接收端将传来的数据进行译码，此实验即设计这样的一个简单编/码系统。系统应该具有如下的几个功能：

1．接收原始数据

从终端读入字符集大小 n，以及 n 个字符和 n 个权值，建立哈夫曼树，并将它存于文件 hfmtree.dat 中。

2．编码

利用已建好的哈夫曼树（如不在内存，则从文件 hfmtree.dat 中读入），对文件中的正文进行编码，然后将结果存入文件 codefile.dat 中。

3．译码

利用已建好的哈夫曼树将文件 codefile.dat 中的代码进行译码，结果存入文件 textfile.dat 中。

4．打印编码规则

即字符与编码的一一对应关系。

5．打印哈夫曼树

将已在内存中的哈夫曼树以直观的方式显示在终端上。

二、数据结构设计

1．构造哈夫曼树时使用静态链表作为哈夫曼树的存储。

在构造哈夫曼树时，设计一个结构体数组 HuffNode 保存哈夫曼树中各结点的信息，根据二叉树的性质可知，具有 n 个叶子结点的哈夫曼树共有 2n-1 个结点，所以数组 HuffNode 的大小设置为 2n-1，描述结点的数据类型为：


```
Typedef strcut
{
Int weight; /*结点权值*/
Int parent;
Int lchild;
Int rchild;
}HnodeType;
```

2．求哈夫曼编码时使用一维结构数组 HuffCode 作为哈夫曼编码信息的存储。

求哈夫曼编码，实质上就是在已建立的哈夫曼树中，从叶子结点开始，沿结点的双亲链域回退到根结点，每回退一步，就走过了哈夫曼树的一个分支，从而得到一位哈夫曼码值，由于一个字符的哈夫曼编码是从根结点到相应叶子结点所经过的路径上各分支所组成的 0、1 序列，因此先得到的分支代码为所求编码的低位码，后得到的分支代码为所求编码的高位码，所以设计如下数据类型：

```
#define MAXBIT 10
Typedef struct
{
Int bit[MAXBIT];
Int start;
}HcodeType;
```

3．文件 hfmtree.dat、codefile.dat 和 textfile.dat。

三、功能（函数）设计

1．初始化功能模块

此功能模块的功能为从键盘接收字符集大小 n，以及 n 个字符和 n 个权值。

2．建立哈夫曼树的功能模块

此模块功能为使用 1 中得到的数据按照教材中的构造哈夫曼树的算法构造哈夫曼树，即将 HuffNode 数组中的各个位置的各个域都添上相关的值，并将这个结构体数组存于文件 hfmtree.dat 中。

3．建立哈夫曼编码的功能模块

此模块功能为从文件 hfmtree.dat 中读入相关的字符信息进行哈夫曼编码，然后将结果存入 codefile.dat 中，同时将字符与 0、1 代码串的一一对应关系打印到屏幕上。

4．译码的功能模块

此模块功能为接收需要译码的 0、1 代码串，按照 3 中建立的编码规则将其翻译成字符集中字符所组成的字符串形式，存入文件 textfile.dat，同时将翻译的结果在屏幕上打印输出。

5．打印哈夫曼树的功能模块

此模块功能为从 HuffNode 数组中读入相关的结点信息，以图形的方式将各个结点以及叶子结点的权值和左分支上的 0 和右分支上的 1 画出来。

四、编码实现

略

五、运行与测试

1．令叶子结点个数 n 为 4，权值集合为{1,3,5,7}，字符集合为{A,B,C,D}，并有如下对

应关系，A——1、B——3，C——5，D——7，调用初始化功能模块可以正确接收这些数据。

2．调用建立哈夫曼树的功能模块，构造静态链表 HuffNode 的存储。

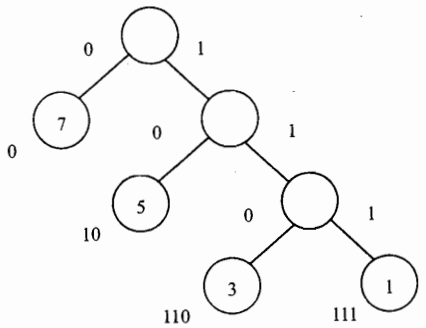
3．调用建立哈夫曼编码的功能模块，在屏幕上显示如下对应关系：

A——111、B——110、C——10、D——0

4．调用译码的功能模块，输入代码串“111110100”后，屏幕上显示译码结果：

111110100 ———— ABCD

5．调用打印哈夫曼树的功能模块。在屏幕上显示如下的树：



实验六　简单校园导游程序的设计与实现

本实验的目的是通过对“校园导游程序”的设计与实现来熟练掌握图型结构在实际问题中的应用。

一、问题描述

当人们到一个陌生的地方去旅游的时候可能会找一个导游为自己在游玩的过程中提供方便。导游可以提供很多服务，比如介绍参观景点的历史背景等相关信息，推荐到下一个景点的最佳路径，以及解答旅游者所提出的关于旅游经典的相关问询等等。对于新生刚刚来到校园，对校园环境不熟悉的情况也如此，一般都是高年级的学生充当了“校园导游”的角色，如果能够提供一个程序让新生或来访的客人自主地通过与机器的“对话”来获得相关的信息的话，将会节省大量的人力和时间。而且所提供的信息也能够做到尽可能的准确、详尽。一个成功的校园导游程序可以替代现实生活中这些“校园导游”，更方便了大家查询校园相关的信息。

本次实验需要开发一个简单的校园导游程序，程序的主体功能为：

1．显示校园平面图，方便用户直观地看到校园的全景示意图，并确定自己的位置。

2．为用户提供对平面图中任意场所的相关信息的查询。

3．为用户提供对平面图中任意场所的问路查询。

二、数据结构设计

由于各个场所通过校园中的道路相连，各个场所和连接它们的道路构成了整个校园的地理环境，所以使用图这种数据结构对它们进行描述。以图中的顶点表示校园内各个场所，应

包含场所名称、代号、简介等信息；以边表示连接各个场所的道路，应包含道路的代号、路径的长度等信息。顶点和边均使用结构体定义，整个图的数据结构可以采用教材中介绍的各种表示方法，例如带权的邻接矩阵。

三、功能（函数）设计

1．显示校园平面图的功能模块

此模块调用已用其他工具绘制好的 BMP 等程序可以识别的图片来显示校园平面图。平面图中应醒目地标识出场所的准确名称以备用户查询。

2．查询任意场所的相关信息的功能模块

此模块接收用户所输入的场所名称，并将场所的简介信息反馈给用户。

3．求单源点到其他各点的最短路径的功能模块

此模块计算并记录从校园门口到各个场所的最短路径。

4．任意场所的问路查询的功能模块

此模块接收用户所输入的场所名称，并在 3 中计算出的最短路径集合中找到相关项的信息反馈给用户。

四、界面设计

本程序为方便用户所设计，由于使用的最终用户大多对校园的情况并不熟悉，所以在图中给出的任何提示信息一定要准确，尽量避免歧义。

五、编码实现

略

六、运行与测试

略

实验七　简单个人书籍管理系统的设计与实现

本实验的目的是通过“简单个人书籍管理系统”的设计与实现，锻炼学生运用所学数据结构的知识来解决实际问题的综合能力。重点锻炼查找和排序在实际系统中要使用的数据结构以及施加在这些数据结构上的算法。

一、问题分析

学生在自己的学习和生活中会拥有很多的书籍，对所购买的书籍进行分类和统计是一种良好的习惯。可以便于对这些知识资料的整理和查找使用。如果用文件来存储相关书籍的各种信息，包括分类、购买日期、价格、简介等等，辅之以程序来使用这些文件对里面的书籍信息进行统计和查询的工作将使得这种书籍管理工作变得轻松而有趣。简单个人书籍管理系统的开发就是为了解决这个实际问题的。

这个系统具备如下的功能：

1．存储书籍各种相关信息。

2．提供查找功能，按照多种关键码查找需要的书籍，查找成功后可以修改记录的相关项。

3．提供排序功能，按照多种关键码对所有的书籍进行排序，例如按照购买日期进行排序。

4．其他辅助的维护工作。

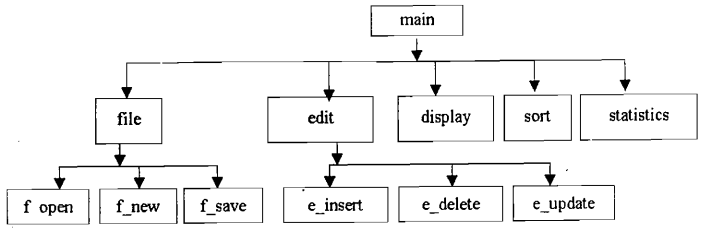
二、数据结构设计

由于书籍的册数较多，而且要在程序不再运行的时候仍然要保持里面的数据，所以采用文件的形式放到外存储器中。需要操作时，从文件中调入内存来进行查找和排序的工作，为了接收文件中的内容，要有一个数据结构与之对应，可以采用这样的结构体数组来接收数据。

```
typedef struct
{
int year,month,day;
}DATE;
typedef  struct
{
    DATE  date;                  /*购买日期*/
        char  name[20];          /*书名*/
        char  author[8];         /*作者*/
        int type;                /* 书籍类别编号 */
        float  price;            /* 价格 */
    }BOOK;
BOOK  books[200];       /*全局数组，保存全部书籍的数据*/
```

三、功能（函数）设计

程序整体的功能模块如下所示：



各个功能模块的含义如下：

1．“主函数”模块 main( )

此模块循环显示第一级操作命令菜单，接收键盘输入的命令，检查命令是否合法，若合法则调用相应下层函数。命令菜单中应包含“退出系统”命令，当接收到该命令时立即终止整个程序的运行。

2．“文件”模块 file( )

此模块循环显示“文件操作”命令菜单，接收键盘输入的命令，检查命令是否合法，若合法则调用相应下层函数。命令菜单中应包含“返回上一级菜单”命令。

3．“新建”模块 f_new( )

此模块清空 books 数组；进入输入状态，接收键盘输入的全部数据保存在 books 数组中，按“购买日期”顺序输入记录。

4．“打开”模块 f_open( )

此模块清除 books 数组中原有数据，从 A 盘上已经存在的数据文件（books.dat）中读入全部数据到 books 数组中，并将全部数据按读入顺序显示在屏幕上（可使用本程序 file->new

和 file->save 功能建立 books.dat，或者用纯文本编辑器建立数据文件，文件中的数据要求按“购买日期”顺序存储）。

5．“保存”模块 f_save( )

此模块将 books 数组中全部有效数据保存到 A 盘 books.dat 文件中。

6．“编辑”模块 edit( )

此模块循环显示“编辑操作”命令菜单，接收键盘输入命令，如命令合法则调用相应下层函数。命令菜单中应包含“返回上一级菜单”命令。

7．“插入”模块 e_insert( )

此模块接收从键盘输入的一条新的记录，按“购买日期”顺序插入到 books 数组中。插入时应考虑到 books 数组中无数据情况下的处理。

8．“删除”模块 e_delele( )

此模块接收从键盘输入的一条记录的“购买日期”和“书名”，在 books 数组中查找，如找到则从 books 数组中删除该记录，否则显示“未找到”。

9．“更新”模块 e_update( )

此模块接收键盘输入的一条记录的“购买日期”和“书名”，在 books 数组中查找，如找到则显示该记录的原数据并提示键盘输入新数据用以替换原有数据，如未找到则显示“未找到”。

10．“显示”模块 display( )

此模块显示类别名称和编号，提示用户输入类别编号，显示 books 数组中指定类别的书籍记录，或输入“all”显示全部书籍记录。

11．“排序”模块 sort( )

此模块对 books 数组中所有记录按“类别”排序，类别相同的按“书名”排序（字典序），显示排序结果（最好不要改变 books 数组中数据的原顺序，将 books 中的数据复制到另一工作数组中再排序）。

12．“统计”模块 statistics( )

此模块统计每类书籍的数量，显示统计结果。

四、界面设计

为了更接近一个真实的应用软件，建议采用用户易于选择的命令输入方式，可以通过方向键选择命令，通过回车确认命令的方式；也可以采用鼠标指点输入的方式。

五、编码实现

略。

六、运行与测试

略。

附录：实验报告范例

数据结构实验报告

——实验三　停车厂模拟管理程序的设计与实现

本实验的目的是进一步理解线性表的逻辑结构和存储结构，进一步提高使用理论知识指导解决实际问题的能力。

一、问题描述

设停车厂只有一个可停放几辆汽车的狭长通道，且只有一个大门可供汽车进出。汽车在停车场内按车辆到达的先后顺序依次排列，若车场内已停满几辆汽车，则后来的汽车只能在门外的便道上等候，一旦停车场内有车开走，则排在便道上的第一辆车即可进入；当停车场内某辆车要离开时，由于停车场是狭长的通道，在它之后开入的车辆必须先退出车场为它让路，待该车辆开出大门，为它让路的车辆再按原次序进入车场。在这里假设汽车不能从便道上开走，试设计这样一个停车厂模拟管理程序。为了以下描述的方便，停车厂的停车场用“停车位”进行叙述，停车厂的便道用“便道”进行叙述。

二、数据结构设计

1．为了便于区分每辆汽车并了解每辆车当前所处的位置，需要记录汽车的牌照号码和汽车的当前状态，所以为汽车定义一个新的类型 CAR，具体定义如下：

```
typedef struct
{
char *license_plate;     //汽车牌照号码，定义为一个字符指针类型
char state;              //汽车当前状态，字符 s 表示停放在停车位上，
//字符 p 表示停放在便道上，每辆车的初始状态用字符 i 来表示
}
```

2．由于车位是一个狭长的通道，所以不允许两辆车同时出入停车位，当有车到来要进入停车位的时候也要顺次停放，当某辆车要离开时，比它后到的车要先暂时离开停车位，而且越后到的车就越先离开停车位，显然这和栈的“后进先出”特点相吻合，所以可以使用一个栈来描述停车位。

由于停车位只能停放有限的几辆车，而且为了便于停车厂的管理，要为每个车位分配一个固定的编号，不妨设为 1、2、3、4、5（可利用数组的下标），分别表示停车位的 1 车位、2 车位、3 车位、4 车位、5 车位，针对这种情况使用一个顺序栈比较方便，具体定义如下：

```
#define MAX_STOP 5
typedef struct
{
CAR STOP[MAX_STOP];      //各汽车信息的存储空间
int top;                 //用来指示栈顶位置的静态指针
}STOPPING;
```


3．当停车场的停车位上都已经停满了汽车，又有新的汽车到来时要把它调度到便道上，便道上的车辆要按照进入便道的先后顺序顺次存放在便道上，为便道上的每个位置也分配一个固定的编号，当有车从停车位上离开后，便道上的第一辆汽车就立即进入停车位上的某个车位，由于问题描述中限制了便道上的汽车不能从便道上开走，即便道上的汽车只有在停车位上停放过之后才能离开停车厂，这样越早进入便道的汽车就越早进入停车位，而且每次进入停车位的汽车都是处于便道“最前面”的汽车，显然，这和队列的“先进先出”特点相吻合，所以，这里使用一个顺序队来描述便道，可以利用数组的下标表示便道的位置，具体定义如下：

```
#define MAX_PAVE 100     /*便道不限制停放车辆的数目，设为足够大*/
typedef struct
{
CAR PAVE[MAX_PAVE];      //各汽车信息的存储空间
int front,rear;          //用来指示队头和队尾位置的静态指针
}PAVEMENT;
```

4．当某辆车要离开停车厂的时候，比它后进停车位的车要为它让路，而且当它开走之后让路的车还要按照原来的停放次序再次进入停车位的某个车位上，为了完成这项功能，再定义一个辅助栈，停车位中让路的车依次“压入”辅助栈，待提出开走请求的车开走后再从辅助栈的栈顶依次“弹出”到停车位中。对辅助栈也采用顺序栈，具体定义与停车位栈类似，如下：

```
typedef struct
{
CAR BUFFER[MAX_STOP];    //各汽车信息的存储空间
int top;                 //用来指示栈顶位置的静态指针
}BUFFER;
```

当然，辅助栈直接利用在 2 中定义的类型 STOPPING 也是可以的。

由于程序的各函数要对这些数据结构中的数据进行操作，而且每次操作的结果都要动态反应到以上数据结构中，所以在程序设计的时候使用以上新类型定义的变量都采用全局变量的形式。

三、功能（函数）设计

1．本程序从总体上分为 4 个大的功能模块，分别为：程序功能介绍和操作提示模块、汽车进入停车位的管理模块、汽车离开停车位的管理模块、查看停车厂停车状态的查询模块，具体功能描述如下：

1）程序功能介绍和操作提示模块：此模块给出程序欢迎信息，介绍本程序的功能，并给出程序功能所对应的键盘操作的提示，具体屏幕显示如下所示：

```
● 欢迎使用本程序。●
本程序为停车场的模拟管理程序，有车到来时请按 C 键。
然后根据屏幕提示进行相关操作，有车要走时请按 1 键。
然后根据屏幕提示进行相关操作，要退出程序请按 Q 键。
请选择您要做的操作！
```

函数原型为 void welcome( )；

2）汽车进入停车位的管理模块：此模块用来登记停车厂的汽车的车牌号和对该车的调

度过程并修改该车的状态，其中调度过程要以屏幕信息的形式反馈给用户来指导用户对车辆的调度。例如，当前停车位上 1、2、3 车位分别停放着牌照为 JF001、JF002、JF003 的汽车，便道上无汽车，当牌照为 JF004 的汽车到来后屏幕应给出如下提示信息：

```
牌照为 JF004 的汽车停入停车位的 4 号车位!

按回车键继续程序的运行。
```

函数原型为 void come( );

此函数还要调用其他对于栈和队列的基本操作。

3）汽车离开停车位的管理模块：此模块用来为提出离开停车厂的车辆做调度处理，并修改相关车辆的状态，其中调度过程要以屏幕信息的形式反馈给用户来指导用户对车辆的调度，当有车离开停车厂后应该立刻检查便道上是否有车，如果有车的话立即让便道上的第一辆汽车进入停车位。例如，当前停车位上 1、2、3、4、5 车位分别停放着牌照为 JF001、JF002、JF003、JF004、JF005 的汽车，便道上的 1、2 位置分别停放着牌照为 JF006、JF007 的汽车，当接收到 JF003 要离开的信息时，屏幕应给出如下提示信息：

```
牌照为 JF005 的汽车暂时退出停车位;
牌照为 JF004 的汽车暂时退出停车位;
牌照为 JF003 的汽车从停车厂开走;
牌照为 JF004 的汽车停回停车位的 3 车位;
牌照为 JF005 的汽车停回停车位的 4 车位;
牌照为 JF006 的汽车从便道上进入停车位的 5 车位;

按回车键继续程序的运行。
```

函数原型为 void leave( );

此函数还要调用其他对于栈和队列的基本操作。

4）查看停车厂停车状态的查询模块：此模块用来在屏幕上显示停车位和便道上各位置的状态，例如，当前停车位上 1、2、3、4、5 车位分别停放着牌照为 JF001、JF002、JF003、JF004、JF005 的汽车，便道上的 1、2 位置分别停放着牌照为 JF006、JF007 的汽车，当接受到查看指令后，屏幕上应显示：

```
停车位的情况:
1 车位——JF001;
2 车位——JF002;
3 车位——JF003;
4 车位——JF004;
5 车位——JF005;
便道上的情况:
1 位置——JF006;
2 位置——JF007;

按回车键继续程序的运行。
```

函数原型为 void display( );

此函数还要调用其他对于栈和队列的基本操作。

2．以上 4 个总体功能模块要用到的栈和队列的基本操作所对应的主要函数如下表所示：

函 数 原 型	函 数 功 能
STOPPING *init_stopping( )	初始化“停车位栈”
BUFFER *init_buff( )	初始化“辅助栈”
PAVEMENT *init_pavement( )	初始化“便道队列”
Int car_come(int pos)	将 pos 指定的汽车信息插入“停车位栈”，并修改该车状态
Int car_leave(int pos)	将 pos 指定的汽车信息从“停车位栈”删除，并修改该车状态
Int stop_to_buff(int pos)	将 pos 指定的汽车信息从“停车位栈”移动到“辅助栈”
Int buff_to_stop(int pos)	将 pos 指定的汽车信息从“辅助栈”移动到“停车位栈”
Int pave_to_stop(int pos)	将 pos 指定的汽车信息从“便道队列”移动到“停车位栈”
Int car_disp(int pos)	将 pos 指定的汽车信息显示在屏幕上

其他函数的定义和说明请参照源代码。

3．由于程序应该能够随时处理用户所提出的各种操作请求，所以在主函数中用一个 DO_WHILE 循环结构随时监控键盘的按键操作，遇到相应的按键就转到对应函数继续运行，运行完该函数继续监控键盘按键，如此往复，直到接到“退出”指令程序才能结束。部分编码如下：

```
welcome( );
flushall( );
do
{
    key=getchar( );
    if(key=='C'||key=='c')
        come( );
    else    if(key=='L'||key=='1')
            leave( );
        else welcome( );
}
while((key!='Q')&&(key!='q'));
```

四、界面设计

本程序的界面力求简洁、友好，每一步需要用户操作的提示以及每一次用户操作产生的调度结果都以中文的形式显示在屏幕上，使用户对要做什么和已经做了什么一目了然。文字表述精练、准确。具体设计可参阅功能设计中的相关部分，这里就不再赘述。

五、编码实现

略。详见具体源代码。

六、运行与测试

对于测试用的设计注重所定义的数据结构的边界以及各种数据结构共存的可能性。例如：

1．连续有 7 辆车到来，牌照号分别为 JF001、JF002、JF003、JF004、、JF005、JF006、JF007，前 5 辆车应该进入停车位 1～5 车位，第 6、7 辆车应停入便道的 1、2 位置上。

2．1 中的情况发生后，让牌照号为 JF003 的汽车从停车厂开走，应显示 JF005、JF004 的让路动作和 JF006 从便道到停车位上的动作。

3．随时检查停车位和便道的状态，不应该出现停车位有空位而便道上还有车的情况。

4．其他正常操作的一般情况。

5．程序容错性的测试，当按键输入错误的时候是否有错误提示给用户指导用户正确操作，并做出相应处理保证程序正常运行。

经过反复的运行和测试，程序的容错性能良好，界面简洁友好，运行稳定可靠，符合实际操作规范，基本达到了模拟停车厂管理的要求。

七、实验完成后的思考

1．通过本程序熟练掌握了栈和队列的定义和使用方法，对使用 C 语言编码来验证数据结构的理论知识有了更深层次的理解，达到了实验目的。

2．通过在设计过程中的讨论和思考，对使用现有知识利用计算机来解决现实生活中的实际问题确立了信心，对软件工程思想和模块化程序思想有了比较清晰的理解，为今后的程序设计奠定了一定的心理和技术上的准备。

3．由于时间仓促和个人能力有限，程序中还有一些需要完善的地方，例如：对停放到停车厂的汽车按时间的收费管理，程序界面可以使用图形方式使之更美观，使用鼠标或菜单方式使之能够适应更多人的使用习惯。在今后的实验中要多多练习这方面的能力。

实验人：×××

实验完成日期：××年×月×日

实验报告提交日期：××年×月×日