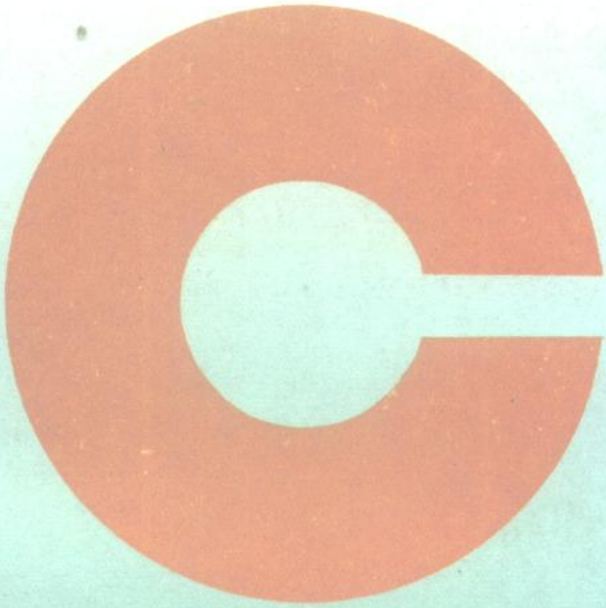

用C语言实现的

数据结构

秦小麟 林钧海 编著

航空工业出版社

用 C 语言实现的

数据结构

秦小麟　　林钧海　编著

航空工业出版社

1996

内 容 提 要

数据结构是计算机专业一门重要的核心基础课，计算机领域内只要涉及到程序的地方，均要用到数据结构的知识。随着C语言在我国的广泛普及，越来越多的程序设计员采用C语言作为编程工具。本书是为了适应实际需要而编写的以C语言为描述语言的数据结构教材。全书共分九章，详细介绍了各种基本的数据结构：线性表、栈、队列、串、数组、广义表、树和图等，以及查找、排序和文件结构。在各章后配有较复杂的应用实例，借以训练学生灵活运用所学数据结构的能力。

图书在版编目（CIP）数据

用C语言实现的数据结构/秦小麟，林钧海编著. —北京：航空工业出版社，1996.7
ISBN 7-80134-020-5

Ⅰ.用… Ⅱ.①秦… ②林… Ⅲ.C语言-程序设计 Ⅳ.TP312C

中国版本图书馆CIP数据核字（96）第09867号

航空工业出版社出版发行
（北京市安定门外小关东里14号 100029）
河北香河县印刷厂印刷 全国各地新华书店经售
1996年8月第1版 1996年8月第1次印刷
开本：787×1092 1/16 印张：14.125 字数：328千字
印数：1—3000 定价：16.50元

前　言

数据结构是计算机专业一门重要的核心基础课。计算机领域内只要涉及到程序的地方，均要用到数据结构的知识；许多课程，例如操作系统、编译原理、数据库和人工智能等也均要用到数据结构的知识。数据结构主要是研究和解决非数值计算的问题，讨论如何合理地组织、存储和处理数据的方法与技术。

自我国高等院校开设数据结构课程以来，兄弟院校已出版了许多有价值的数据结构教材，其描述语言基本上是采用PASCAL语言或类PASCAL语言。但是，随着C语言在我国的广泛普及，不仅系统程序员，而且越来越多的应用程序员也采用C语言作为编程工具。我们也发现计算机专业学生毕业后所用的高级语言主要是C语言。在编写本书之前，我们在近几届数据结构教学中，虽然采用的教材是以类PASCAL语言为描述语言，但却鼓励学生用C语言做作业与上机实习，得到了学生的热烈响应。尤其在上机实习(包括一个20机时的课程设计)中几乎有 $\frac{1}{2}\sim\frac{2}{3}$ 的学生使用C语言。众所周知，C语言入门容易，但真正掌握其精髓则较难。如何保证学以致用，更好地掌握C语言及用C语言描述、实现各种数据结构与算法，我们着手编写这本以C语言为描述语言的数据结构教材。该教材已用了两届三次(两次本科生、一次专科生)，效果很好。我们认为数据结构、操作系统等课程采用C语言为描述语言只是一个时间问题，是必然的趋势。

本书详细介绍了各种基本的数据结构：线性表、栈、队列、串、数组、广义表、树和图等，以及查找、排序和文件结构等，全书共分为九章。第一章介绍了数据结构课程的背景及有关的概念和术语，并讨论了抽象数据类型、复型数据类型和递归函数等内容。第二章至第六章分别介绍了上述的各种数据结构。第七章介绍了各种基本的查找方法。第八章讨论了各种常用的排序算法。第九章讨论了文件的物理结构，并介绍了支持空间数据的索引文件——R树。在各章后均配有较复杂的应用实例，借以训练学生灵活使用所学数据结构的能力。并从第七章开始，对各种算法进行简单的时间复杂性分析。

本书可作为计算机专业的本科生教材，大约为70～80学时。若低于70学时，则可删去有关的应用实例，这不会破坏教学内容的完整性。数据结构是一门实践性很强的课程，故每章均附有一定量的习题，只有通过大量的编程及上机实习才能更好地掌握该课程的内容。根据我们的教学经验，在该课程的教学中必须要求学生养成良好的编程风格，尽量多加注释语句，最好在具体编程前先给出其算法思想。学习本书前，读者应具有C语言的基础，若有离散数学和概率论的知识则更好。

本书第一、二、三章由林钧海同志编写；第四、五、六、七、八和九章由秦小麟同志编写。

李立新同志仔细地审阅了初稿，并提出了许多宝贵的意见。在编写该教材过程中得到了学校教材科与计算机系有关领导的大力支持和帮助，教材科李定鉴科长一直关心和参与本书的编写与出版，促成了本书的出版能够顺利进行。杨晓明、刘小廷等同志为排版做了大量的工作，在此一并表示衷心的感谢。

由于编者水平有限，书中难免有错误及不妥之处，恳请读者与同行批评指正。

编　者

1996.2 于南京航空航天大学

目　录

第一章 绪 论

1.1 数据结构课程的历史背景、内容及其意义

数据结构不仅是计算机专业教学计划中的核心课程，而且被越来越多的与计算机应用有关的专业作为重要的选修课。在60年代没有独立开设这门课，它的大部分内容多散布在《表处理语言》、《图论》、《离散数学结构》等教材中。1968年美国唐·欧·克努特出版的专著《计算机程序设计技巧》第1卷《基本算法》首次系统地阐述了数据结构的主要内容，即数据的逻辑结构、存储结构以及对数据进行各种操作的算法。到70年代中期和80年代初各种数据结构著作大量问世。在我国独立开设数据结构课程则是70年代末的事了。

大凡用计算机解题，总是要把客观对象抽象为某种形式的数据，然后设计在这些数据上进行操作的算法，由计算机执行这些算法，最后获得问题的解答。比如高等学校招生录取工作的计算机处理，它要解决哪些考生可以被录取的问题。首先把考生抽象为姓名、各科考试成绩、总分等数据项组成的记录(一类数据结构)，然后设计对考生记录进行操作的算法。这里我们看到解题过程中要解决的数据结构的设计和算法的设计问题。N.沃思的著作取名为≪算法＋数据结构＝程序≫，点破了计算机解题的真谛。他认为程序就是在数据的某些特定的表示方式和结构的基础上对抽象算法的具体描述，数据结构和算法是程序设计过程相辅相成不可分割的两个方面。不了解施加于数据上的算法就无法决定数据结构，反之算法的结构和选取往往很大程度上依赖于数据结构。

由于计算机应用的初期侧重于解决数值计算的问题，处理的对象相对简单，如处理整数运算、实数运算等，用简单变量或数组这些形式的数据结构足以表示要解决的问题，因此那时的程序设计重点是制定最佳的算法。但是随着计算机应用扩展到非数值计算领域，如行政事务处理、人工智能模拟、实验数据的采集与处理等，处理的数据量和复杂度大大增加了。合理的组织、存储和处理数据的重要性已越来越明显地显露出来。现在有关数据结构的知识已成为设计和实现编译系统、操作系统、数据库系统及其它系统程序和一些大型应用系统的重要基础。

1.2 学习方法

数据结构是一门实践性很强的课程，要掌握本课程的概念与方法，必须在计算机上进行足够时间的实习。但是为了掌握基本概念与方法，仔细阅读书中介绍的每个算法是非常重要的。在阅读程序的过程中掌握概念和算法，从中学习编制程序求解问题，然后完成足够数量的习题并将它在计算机上实现，是学习数据结构课程不可缺少的步骤。

1.3 关于描述数据结构和算法的语言选择

我们主张选择实际使用的程序设计语言来描述数据结构和算法，使得教材中的许多实例可以成为可在计算机上运行并被众多程序员调用的标准程序。尽管PASCAL语言也有较丰富

的数据结构和较强的表示算法的能力。但毕竟在实际中使用较少，较多限于教学上使用。C 语言已经是普遍流行的程序设计语言。不仅系统程序员而且越来越多的应用程序员也转向采用 C 语言作为编程工具。高级 C 语言结构也提供丰富的描述数据结构和算法的能力。本书采用 C 语言描述数据结构和算法，读者阅读本书时如对 C 语言不甚熟悉，可先参阅有关 C 语言的书籍。

1.4 基本术语

1. 数据(data)与信息(information)

数据是指描述客观事物的数、字符、正文、图形、图像和语音等所有能输入到计算机中被计算机处理的符号集合，这些符号集最终在计算机内部都表示为二进制位串的形式，所以数据是待计算机加工的二进制位串，它是计算机加工的"原料"。信息则是经计算机加工以后产生的有意义的数据。它是加工后产生的"产品"。习惯上，数据和信息两词互相通用，但实际上是有区别的。例如经过摄像机摄取的图像信号输入到计算机中成了未加工的数据，经过计算机加工(去噪声、增强等)后形成的清晰的图像，成为对于人们有意义的信息。

2. 数据项和数据元素

数据元素是数据的基本单位，相当于物质的"分子"，比如学生成绩登记表是一种数据结构，它由许多学生成绩记录组成，成绩记录则是构成登记表的元素即"分子"。有时数据元素又由许多项组成，数据项是数据的最小单位，相当于"原子"，例如每个学生成绩记录中可能含有姓名、课程名、成绩等数据项。

3. 数据结构

数据结构是带有某种结构的数据元素的集合，即数据元素之间存在某种形式的联系，如任意两个数据之间可以区分为哪个在前哪个在后，即数据元素之间存在"有序"的结构。不同类型的数据结构，它们的数据元素之间可有不同形式的联系，即存在不同的结构。读者可以在今后各章中进一步了解其内涵。

1.5 抽象数据类型

抽象数据类型是带有某些操作的集合，有时称这些操作为元操作，例如带有加、减、乘、除运算的整数集合就是一抽象数据类型，C 语言提供的内部 int 类型是整数抽象数据类型的软件模型，抽象数据类型的软件模型最重要的方面是它们的函数性质，而不是它们的计算机表示的细节。它们的内部结构与实现有关，但与用户无关。抽象数据类型提供模块性即只有它们的性质保持不变，它们的基本数据结构可以改变而不影响使用抽象数据类型的程序，抽象数据类型改善了可移植性。硬件和系统软件相关的影响涉及到数据的存储方式和处理，仅仅限于抽象数据类型的实现部分的低级编码。

数据结构是表示抽象数据类型的特定方法，通常可以有好几种方法，例如有理数可以表示为浮点变量(float)或者表示为一个结构，它包含一对整数(int)组成的商数。

数据类型有三类：

· 原子数据类型

这些数据类型存放单个数据值。通常内部定义数据类型适合于这种类型的数据，但是有时

也需要建立自己的原子数据类型，比如人们要有这样的整数，它包含好几千位数字，也就超出了 C 语言内部整数型的能力。

· 固定聚集数据类型

这种类型的值含有固定个数的原子分量。例如“复数”抽象数据类型可以看成是一对实数分量，一个是实部，一个是虚部。这种数据类型通常用结构或数组表示。

· 变聚集数据类型

这种数据类型的值含有可变个数的分量。例如抽象数据类型“有序整数表”含有任意长的有序整数序例。虽然数组有时能表示这种抽象数据类型，但通常比较适合用动态地分配存储空间并用指针将它们链接在一起的数据结构来表示。本书主要处理这种数据类型。

抽象数据类型的软件模型由一组操作符组成。这些操作符用以建立、操纵和删除某特定类型的数据对象。例如在 C 语言中，整数是说明为 int 变量而建立起来的，是用操作符 +，-，* 和 / 进行操作的，它们的删除是自动进行的。因为 int 类型是抽象数据类型的模型，所以在使用 int 变量时可以不问它们是如何存储或者操作符是如何工作的。内部数据类型可以直接用相应的内部抽象数据类型操作符进行操作，但是用户定义类型的变量则没有内部抽象数据类型操作符存在，它们必须由用户定义的函数进行访问。

抽象数据模型的软件模型通常是不完善的，例如不可能表示无穷集。注意，C 语言为整数(int，short，long)和实数(float，double)提供多种表示方法，允许程序员在存储空间和精度之间进行选择。当建立不完善的软件模型时，通常希望提供某种机制可以表示失败(例如，溢出、下溢)。在 C 语言中表示失败的一种办法是使用元操作的回送值。

```
typedef enum {OK,ERROR} status;
typedef enum {FALSE=0,TRUE=1}bool;
```

这里定义枚举类取值 OK，ERROR 为类型 status。它能回送值 OK 和 ERROR。而那些不执行出错检测的函数被说明为类型 void，那些回送布尔值(true 或 false)的函数被说明为类型 bool。注意置 FALSE 为零。

对于元操作在实践中对出错的反应最好是回送值而不要采取明显的活动，比如打印出错信息或夭折该程序。这样处于抽象层次高的调用者函数，具有最大的灵活性。

有时可能对一个表示抽象数据类型的变量施以不合适的操作符。例如 C 语言中的 int 已知在特定的计算机上是以二进制机器字存储，程序员可能对 int 变量直接施以移位操作符而不是用操作符 * 作乘 2 运算，不用操作符 / 作除 2 运算。但是，一个变量用不属于该抽象数据类型的操作符进行运算以后，不能再说成是属于该类型的变量。这种方法破坏了实现的抽象，失去了前述有关抽象数据类型的好处。实质上，变量的类型不仅依赖于它的定义，而且也依赖于它所接受的操作。虽然对表示整数的 int 变量施以移位操作不合适，但是对表示其它抽象数据类型的 int 变量施以移位操作可能是合适的。

考虑抽象数据类型“复数”的实现。虽然有多种方式实现复数，但是与该抽象数据类型的函数接口不必改变。所以在实际编码之前，可以选择对此抽象数据类型的函数接口，这包括用于访问变量的函数原型和必要的类型说明，函数原型是一个高级函数声明，它包含类型信息，虽然不是 C 语言的所有实现都支持这个性能，但 ANSIC 的标准有这个性能。

变量被说明为复数抽象数据类型的实例：

```
complex variablename;
```

为实现这个抽象数据类型的函数有：存储一个复数，检索复数的实部和虚部，加法和乘法。

关于减法、除法和测试复数相等的函数留作习题。

有多种方法表示复数。例如可以用笛卡尔坐标：$z=\alpha+i\beta$，这里 α,β 是实数，α 是实部，β 是虚部。也可以用极坐标：$z=re^{i\theta}$，这里 $r=\sqrt{\alpha^2+\beta^2}$，$\theta=\tan^{-1}(\beta/\alpha)$。这两种方式很容易进行转换，选哪一种方法可依各人所好。在本例中采用笛卡尔坐标：

```
void load_complex(complex * p_complex,double real,double imaginary);
void retrieve_complex(complex * p_complex,double * p_real,
                      double * p_imaginary);
void add_complex(complex * p_sum,complex * p_complex1,
                 complex * p_complex2);
void multiply_complex(complex * p_product,complex * p_complex1,
                      complex * p_complex2);
typedef struct {
    double real;
    double imaginary;
}complex;
```

即使选取笛卡尔坐标作为原子数据类型和复数数据类型之间的接口，内部表示仍然不一定相同。如果希望用极坐标形式存储复数在函数“load”和“retrieve”中可以作一简单的转换，复数函数的实现是浅显的。函数 mutiply_comliex( ) 用了 $i=\sqrt{-1}$ 和 $i^2=-1$。

```
void load_complex(p_complex,real,imaginary)
complex * p_complex;
double real,imaginary;
{
    p_complex->real = real;
    p_complex->imaginary = imaginary;
}

void retrieve_complex(p_complex,p_real,p_imaginary)
complex * p_complex;
double * p_real, * p_imaginary;
{
    * p_real = p_complex->real;
    * p_imaginary= p_complex->imaginary;
}

void add_complex(p_sum,p_complex1,p_complex2)
complex * p_sum, * p_complex1, * p_complex2;
{
      /*
       *   p_sum = p_complex1+p_complex2;
       *         = (a+bi)+(c+di)
       *         = (a+c)+(b+d)i
       */
   p_sum->real = p_complex1->real+p_complex2->real;
 p_sum->imaginary = p_complex1->imaginary + p_complex2->imaginary;
}

void multiply_complex(p_product,p_complex1,p_complex2)
complex * p_product, * p_complex1, * p_complex2;
{
```

```
    /*
     *   p_sum = p_complex1 * p_complex2;
     *         = (a+bi) * (c+di)
     *         = (ac-bd)+(ad+bc)i
     */
    p_product->real = p_complex1->real * p_complex2->real
        - p_complex1->imaginary * p_complex2->imaginary;
    p_product->imaginary= p_complex1->real * p_complex2
        ->imaginary+p_complex1->imaginary * p_complex2->real;
}
```

在实践中，函数原型和 typedef 语句应放在头文件中。然后使用此新类型的文件应该用宏处理语句 #include 来引用此头文件，而且一般说来应将指向该新类型的变量的指针作为函数的参数传送，而不是变量本身。这个可以避免当 C 编译程序不支持结构作为参数时发生移植性问题，也可以减少将大型结构作为参数时的开销从而提高效率。

1.6 复型数据类型

考察抽象数据类型“表”，是元素的有序序列，它的元操作包括增加一新元素、删除元素、确定表中元素的个数、对表中每一个元素使用特定的函数。当然如果表抽象数据类型的元素是特定类型的原子元素，这就可能很麻烦，可能需要建立新的不相容的表类型和元操作。例如 int 型表、float 型表等等。事实上甚至不可能造一个特定抽象数据类型的表，例如复数的表，因为可能要求表程序依赖于表示抽象数据类型的数据结构，而且没有办法建立异构型表包含各种不同类型的数据，所以希望建立复型数据类型，即其数据类型与元素的类型无关。为了建立复型数据类型，有必要进一步了解类型的一般概念。

数据类型之所以重要，是由于它提供了两个基本信息：一个变量用了多大的内存空间，以及该数据是如何解释的。所以复型类型必须与占用的空间大小和内存储数据精确性质函数无关。为了达到第 1 个目标，复型数据结构不直接访问其元素，而是使用指针。指针能提供对任何大小的元素的定长访问。在 C 语言中指向字符的指针(char *)可以用于指向任何数据类型，类型 generic_ptr 可以用于达到此目的的指针。

```
typedef char *generic_ptr;
```

为了达到第 2 个目标，应遵守下列规则：用于表示复型数据类型的数据结构对它存储的数据不作任何假定。在语义上，这意味着所有与特定类型有关的信息均应通过参数传递送给元操作。通常这些信息将包括指向特定类型函数的指针。

1.7 递归函数

递归在算法设计与程序开发中是一个重要概念。在解题过程中，一个函数又调用自己去解决问题，这种控制技术称为递归。很多问题用迭代结构比如 for，while 和 do 循环结构难解决但用递归却很方便。在数学上和计算机科学中递归意味着自我调用。所以一个递归函数的定义是建立在自己的基础之上的。例如阶乘函数可以递归地定义为：

$$n!=\begin{cases}1 & \text{if } n=0\\ n\times(n-1)! & \text{if } n>0\end{cases}$$

注意：$n!$ 由 $(n-1)!$ 来定义。显然阶乘的定义链是不会出现环形的，因为：

1. 当函数调用它自己时，它是用了比给定量更小的变量。

2. 对于最小情况时的函数值的定义中没有自我调用，所以递归链会终止。

递归函数可以直接用 C 语言实现，计算阶乘的递归函数可以写为：

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

阶乘函数用了尾递归，即在最后一个语句含有简单的递归调用。因为尾递归的结构决定了尾递归函数总是有直接迭代与其等价，例如阶乘函数可以写为：

```
int  factorial(n)
int  n;
{
    if (n == 0)
        return 1;
    else {
        int i,fact = 1;
        for (i = 2;i <= n;i++)
            fact *= i;
        return fact;
    }
}
```

迭代稍为有效一点，但是选用尾递归或是迭代很大程度上是风格问题，并且与如何看问题有关。例如迭代的阶乘函数是以问题的迭代定义为基础的：

$$0! = 1$$

$$n! = \prod_{i=1}^{n} i$$

同样计算一个数的正整数幂也可以有递归和迭代两种定义。递归定义是

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \times x^{n-1} & \text{if } n > 0 \end{cases}$$

以上述定义为基础的递归函数 power( )为

```
double power(x,n)
double x
int n;
{
```

```
    if (n == 0)
        return(1);
    else
        return(x * power(x,n-1));
}
```

迭代定义是

$$x^0 = 1$$

$$x^n = \prod^{n} x$$

迭代函数 power( )留作习题。

递归算法是基于这样的解题原理，即将问题分解为更小的问题。尾递归从第 $n-1$ 的情况的解获得第 n 情况的解，而且更一般地，在产生解之前，递归函数可能重组好几个子问题。这种技术称为"分而治之，各个击破"。通常导致比朴素的迭代更有效的解。

例如考虑在一个整数数组中同时找最大值和最小值问题。下面是朴素的迭代函数：

```
minmax(a,n,p_max,p_min)
int a[ ],n, * p_max, * p_min;
{
    int i;

    * p_max = * p_min = a[0];

    for (i = 1;i<n;i++){
        if (a[i]> * p_max)
            * p_max = a[i];
        if (a[i]< * p_min)
            * p_min = a[i];
    }
}
```

这个函数作了 $2*(n-1)=2n-2$ 次比较，因为在循环中 $n-1$ 次迭代的每一次要作两次比较，现在考虑递归算法：

如果 $n=1$

    min=max=a[0];

如果 $n=2$

    比较 a[0]和 a[1]并且较小者赋给 min，较大者赋给 max。

否则：

    将数组一分为二并且对其中的每一半，递归地找 min 和 max。
    将两个 min 中的小者赋予 min，将两个 max 中的大者赋予 max。

利用第 8 章中的技术可以证明这个算法只作了 $1.5\times n-2$ 次比较。虽然两方法的差额不过是一常数，但是"分而治之，各个击破"的技术可以用于更大的改进。该函数的 C 语言实现如下：

```
void minmax(numberlist,n,p_min,p_max)
int numberlist[ ],n, * p_min, * p_max;
{
```

```
    int min2,max2;

    if (n == 1)
        *p_min = *p_max = numberlist[0];
    else if (n == 2){
        if (numberlist[0] <numberlist[1] {
            *p_min = numberlist[0];
            *p_max = numberlist[1];
        } else {
            *p_min = numberlist[1];
            *p_max = numberlist[0];
        }
    } else {
        minmax(numberlist,n /2,p_min,p_max);
        minmax(numberlist +n /2,n-(n /2),&min2,&max2);
        if (min2 < *p_min)
            *p_min = min2;
        if (max2 > *p_max)
            *p_max = max2;
    }
}
```

习　题

1. 为复数抽象数据类型写下列函数：

(a)subtract_complex( )　两个复数之差

```
void subtract_complex(complex *p_difference,complex *p_complex1,
                      complex *p_complex2);
```

(b)equal_complex( ) 如果两复数相等则回送 TRUE，否则回送 FALSE

```
bool equal_complex(complex *p_complex1,complex *p_complex2);
```

(c)divide_complex( )计算两复数之商

```
void divide_complex(complex *p_quotient,complex *p_complex1,
                    complex *p_complex2);
```

2. 编写 C 语言程序实现有理数抽象数据类型。有理数用包含一对 int 型变量的结构表示。写出下列函数。

```
void load_rational(rational *p_rational,
                   int numerator,intdenominator);
void retrieve_rational(rational *p_rational,int *p_numerator,int
                       *p_denominator);
void add_rational(rational *p_result,rational *p_rational1,
                  rational *p_rational2);
void subtract_rational(rational *p_result,rational *p_rational1,
                       rational *p_rational2);
void multiply_rational(rational *p_result,rational *p_rational1,
```

```
                    rational * p_rational2);
status divide_rational(rational * p_result,rational * p_rational1,
                    rational * p_rational2);
bool equal_rational(rational * p_rational1,rational * p_rational2);
```

3. 以数字为元素的数组是可以用于表示特大正整数的数据结构。下面是这种数据结构的 4 种形式：

(a)
```
typedef struct {
        int digits[100],number_of_digits;
}integer;
```
这里 digits[0]含有整数最高位数字，number_of_digits 存放整数位数。

(b) 结构同上，但 digits[0]存放最低位。

(c)
```
typedef struct {
        int digits[101];
}integer;
```
这里 digits[0]包含整数的最高位，而最低位之后放一个－1，表示整数结束。

(d)用(c)的结构，但 digits[0]包含最低位。

利用上述数据结构编写 4 功能计算器(加、减、乘和除)，这个计算器能处理多达 100 位正整数。(提示：输入输出时要进行数字数组到字符串之间的转换，转换要与字符的机器表示无关，如与 ASCII 码或 EBCDIC 码无关)。

把计算器的功能扩展为既可处理大的正整数，也可处理负整数，并且讨论上述 4 种数据结构的优点(如对数学运算的方便性和输入输出的简易方便等)。

4. 编写一迭代程序计算 x^n。

5. 编写迭代程序计算 fib(n)。把你的程序与书中的递归程序比较，基于斐波那齐数的定义讨论哪个程序更简单？你能给出斐波那齐数的非递归定义吗？

6. (a)假定 C 语言中没有内部乘函数(*)，但仅有加(+)。利用递归地加，编写递归函数 mult，它以两个非负整数 m 和 n 为参数，回送乘积 $m \times n$ 的值。

(b)假定 C 语言没有内部加函数，只有加 1 运算。编写递归函数 add，它以两个非负整数为参数，回送 $m+n$ 之值。

7. 编写一 C 语言程序，递归地计算两个数的最大公约数，建议用如下的算法。

$$gcd(n,m)=\begin{cases} gcd(m,n) & \text{如果 } n<m \\ m & \text{如果 } n\geqslant m \text{ 且 } n \bmod m = 0 \\ gcd(m,n \bmod m) & \text{其它情况} \end{cases}$$

8. 组合函数 $c(n,r)$可以递归地定义为 $c(n-1,r)+c(n-1,r-1)$，编写一递归程序计算 $c(n,r)$。

9. 对任意的 64×64 矩阵实现斯特拉伸(Strassen)矩阵乘算法。对于大型矩阵，此算法优于传统算法。

斯特拉伸算法：给定 $n \times n$ 矩阵 A 和 B，这里 n 是 2 的幂，求解 $C=A \times B$。

如果 n 不等于 2：

将 A，B 和 C 分解为 4 个 $n/2 \times n/2$ 子矩阵，并且递归地对这 4 个矩阵施用本算法。

否则：

计算下列部分积：

$$M1 = (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})$$

$$M2 = (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2})$$

$$M3 = (A_{1,1} - A_{2,1}) \times (B_{1,1} + B_{1,2})$$

$$M4 = (A_{1,1} + A_{1,2}) \times B_{2,2}$$

$$M5 = A_{1,1} \times (B_{1,2} - B_{2,2})$$

$$M6 = A_{2,2} \times (B_{2,1} - B_{1,1})$$

$$M7 = (A_{2,1} \times A_{2,2}) \times B_{1,1}$$

由它们组成矩阵 C：

$$c_{1,1} = M1 + M2 - M4 + M6$$

$$c_{1,2} = M4 + M5$$

$$c_{2,1} = M6 + M7$$

$$c_{2,2} = M2 - M3 + M5 - M7$$

使用随机数组测试你的程序。你如何验证输出？

第二章　表

本章讨论的表结构可以作为实现其它数据结构的基础。

2.1　表的概念

表是一些对象的有序集合，这并不意味着表已经排序，而是指当插入数据对象时可以明确它在表中应处的相对位置。例如我们可以定义一个运算符，它增加一对象（通称数据项）到表的最前面即新数据项总是作为表的第一个元素，其它数据项相应地挪动一个位置。或者可以增加一数据项到表的末尾或是任何其它位置。

表具有如下特性：

- 表有零个或多个数据项；
- 新的数据项可以插入到表中的某一位置；
- 可以删除表中的任一数据项；
- 可以访问表中的任一数据项；
- 表中的数据项可以逐个地访问，通称遍历一个表。

把表作为抽象数据类型实现有一定困难，因为可能在表上要实行很多不同的操作。即便是特定的操作如插入和删除一数据项，也可能有多种方式。

例如，可能希望插入数据项到表的开头，也可能要插入到特定的位置（insert_atposition），或是在特定结点的前面或后面（insert_before，insent_after）。还可以附加一数据项到表的末尾。对于删除一数据项也有类似的情况。

尽管如此，只要能清晰地定义界面，表仍然可以作为抽象数据类型加以实现。

C 函数的原型，只描述元操作的子集。而其余的元操作可以作为应用开发来定义。init_list（）有一个参数即指向需初始化的表的指针。因为此函数可能牵涉动态的空间分配，它会回送一出错标志。empty_list()也有一个参数，如果表非空则回送 TRUE，若为空则回送 FALSE。函数 append（）在表末添加一数据项，此函数有两个参数，被插入的数据和指向表的指针。同样，函数 insert（）插入一数据项到表的前端，它也有两个参数。delete（）是 insert（）的对偶，它从表中删去第 1 个数据项。把被删除的数据项置入参数。

```
status init_list(list *p_L);
bool empty_list(list L);
status append(list *p_L,generic_ptr data);
status insert(list *p_L,generic_ptr data);
status delete(list *p_L,generic_ptr *p_data);
```

注意，这些函数原型是独立实现的，类型 generic_ptr 用于表示任一类型的数据。有关概念稍后说明。

多数程序员熟悉表的一种实现即数组。如果表中少数几个数据项可能被删除，则数组是表