第1章　绪　　论

数据结构是计算机专业的一门专业基础课程，很多后续课程都要用到本课程所涉及的知识。例如，程序设计、编译技术和操作系统等课程都要使用一些基本的数据结构及其相关的算法；本课程讨论的其他一些数据结构，如广义表、集合以及图等也是很多应用领域经常涉及的。本课程的目的是介绍一些最常用的数据结构，阐明数据结构内在的逻辑关系，讨论它们在计算机中的存储表示，并结合各种数据结构，讨论其各种操作的实现算法。

本章将概括地介绍一些基本概念和术语，包括数据和数据结构、数据的逻辑结构和存储结构、数据类型、算法的概念及描述、算法的复杂度分析等。

1.1　引　　言

自从1946年第一台计算机问世以来，计算机产业飞速发展，计算机的功能不断增加，速度也不断提高。计算机的应用已从最初的科学计算发展到人类生活的各个领域。计算机加工处理的对象也由纯粹的数值数据发展到字符、表格和图像等各种具有一定结构的数据，这就给程序设计带来一些新的问题。为了设计出一个“好”的程序，必须分析待处理的对象的特性以及各对象之间存在的关系，这就是数据结构这门学科形成和发展的背景。

用计算机解决一个具体问题时，一般需要经过以下几个步骤：首先分析实际问题并从中抽象出一个适当的数学模型，然后设计一个解决此数学模型的算法，最后编制出程序上机调试，直至得到最终的解答，其过程如图1-1所示。

图1-1　计算机解决具体问题的步骤

寻求数学模型的实质是分析问题，从中提取操作的对象，并找出这些操作对象之间的关系，然后使用数学模型加以描述。数值计算问题的数学模型一般可用数学方程或数学公式来描述，而更多的非数值计算问题无法用数学方程加以描述。下面给出几个例子。

【例1-1】 某单位职工档案管理问题。为了简单，假定每个职工的档案只包括以下5项：职工号、姓名、性别、出生日期和职称。一般来说，档案管理人员很可能将这些档案组织成表格形式，如表1-1所示，表中的每一行反映了一个职工的基本情况。

表 1-1 职工档案表

职 工 号	姓 名	性 别	出生日期	职 称
0001	曾国庆	男	1965.03.20	高工
0002	吕 晴	女	1966.04.16	助工
0003	许建莉	女	1967.12.09	工程师
0004	张国玉	男	1969.07.28	工程师
0005	关小西	男	1970.09.14	工程师
…	…	…	…	…

本问题中的处理要求包括所有能用计算机完成的档案管理工作,至少应包含以下功能:

(1)查找,需要时在表格中找出某人的档案。

(2)读取,阅读通过查找找到的档案。

(3)插入,调入新职工时将该职工的档案加入到表格中。

(4)删除,某职工调离时,从表格中撤销其档案。

(5)更新,某职工情况变化(如晋升职称)时,修改档案的有关内容。

上述每一项功能又可称为一个“运算”。由此,从职工档案管理问题抽象出来的数学模型便包含职工档案表和对此表进行的各种运算。在这类管理问题的数学模型中,计算机处理的对象之间通常存在着一种最简单的线性关系,这类数学模型可称为线性的数据结构。诸如此类的还有电话自动查号系统、排课系统、仓库库存管理系统等各种应用系统。

【例 1-2】 UNIX 操作系统中文件系统的系统结构图。UNIX 操作系统中文件系统的底层是根目录,根目录下包含 bin、lib、user、etc 等一级子目录,而 bin 目录下又包含 math、ds、sw 等二级子目录,user 目录下又包含 yin、xie、tang 等二级子目录,yin 目录下有 Linklist.c、Stack.c 和 Tree.c 等若干个文件,如图 1-2 所示。如果将这些目录和文件都画在一张图上,它们之间所呈现的是一种层次关系,从上到下按层进行展开形成一棵倒长的“树”。“树根”是系统的根目录,其他目录和文件是这棵树上的“树叶”,从根结点到某个叶子结点经过的路径就是当前文件的绝对路径。由此可见,“树”也可以是某些非数值计算问题的数学模型,它也是一种数据结构。“树”状结构的模型在生活中接触的也比较多,如国家行政区域规划、书籍目录等。

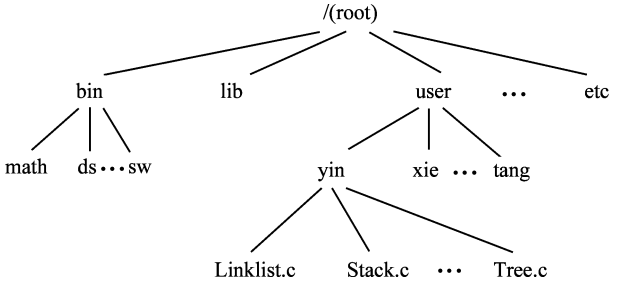


图 1-2 UNIX 操作系统中的文件系统的系统结构图

【例 1-3】 比赛编排问题。有 6 支球队进行足球比赛,分别用 V_1、V_2、V_3、V_4、V_5、V_6 表示这 6 支球队,它们之间的比赛情况也可以用一个称为“图”的数据结构来表示,图中的每个顶点表示一个球队,如果从顶点 V_i 到 V_j 之间存在有向边$<V_i,V_j>$,则表示球队 V_i 战胜球

队 V_j，如 V_1 队战胜 V_2 队，V_2 队战胜 V_3 队，V_3 队战胜 V_5 队等，这种胜负情况可以用图 1-3 表示。

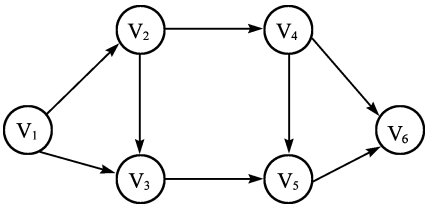


图 1-3 比赛胜负情况图

由此可以看出，用点、点与点之间的线所构成的图也可以反映实际生产和生活中的某些特定对象之间的特定关系。诸如此类有铁路交通图、教学编排图等。

综合以上 3 个例子可见，描述非数值计算问题的数学模型不再是数学方程，而是诸如表、树、图之类的数据结构及其运算。因此可以说，数据结构课程主要是研究非数值性程序设计中所出现的计算机操作对象以及它们之间的关系和运算等的学科。

在美国，数据结构作为一门独立的课程开始于 1968 年。在这之前，它的某些内容曾在其他课程中有所阐述。1968 年，美国一些大学计算机系的教学计划中，虽然把数据结构规定为一门课程，但对其内容并未作明确规定。当时，数据结构几乎和图论，特别是表、树的理论为同义语。随后，数据结构的概念被扩充到网络、集合代数论、格、关系等方面，从而变成了现在称之为“离散结构”的内容。由于数据必须在计算机中进行处理，所以不仅要考虑数据本身的数学特性，而且要考虑数据的存储结构，这就进一步扩充了数据结构的内容。

数据结构在计算机学科中起着承上启下的作用，在计算机技术的各个领域也有着广泛的应用。学习这门课需要程序设计语言作基础，学习前要有比较扎实的程序设计基本功，同时该课程又为后续的数据库等一系列课程奠定基础。数据结构的研究不仅涉及计算机硬件（特别是编码理论、存储装置和存取方法等）的研究范围，而且与计算机软件的研究有着密切的关系，无论是编译程序还是操作系统，都涉及到如何组织数据，使检索和存取数据更为方便。因此可以认为，数据结构是介于数学、计算机硬件和计算机软件三者之间的一门核心课程。

学习数据结构的目的是了解计算机处理对象的特性，将实际问题中所涉及的处理对象在计算机中表示出来并对它们进行处理。与此同时，通过算法训练来提高学生的思维能力，通过程序设计的技能训练来促进学生的综合应用能力和专业素质的提高。

1.2 基 本 术 语

为了便于以后各章的学习，本节将对全书中常用到的一些基本概念和术语给以确切的定义。

1.2.1 数据及相关概念

数据(data)：数据是客观事物的符号表示，在计算机科学中是指所有能输入到计算机中

并被计算机程序处理的符号的总称。数据是计算机进行程序处理的必要的“原料”信息。例如,一个代数方程求解程序所用到的数据是整数和实数,一个编译程序处理的数据是字符串(源程序)。在计算机科学中,数据的含义相当广泛,如表格处理软件中的各类表格信息,多媒体处理软件中的各种多媒体信息,包括音频、视频信息等。总之,数据是通过编码形成的各种客观事物的信息。

数据元素(data element):数据元素是数据的基本单位,在计算机程序中通常把它作为一个整体来处理。例如,【例 1-2】中的文件系统的系统结构图的一个目录,【例 1-3】中的“图”的一个圆圈都被称为一个数据元素。有时,一个数据元素又可以由若干个**数据项**(data item)组成。如表 1-2 所示学生成绩表中的一条记录为一个数据元素,而记录中的学号、姓名、语文等都分别为一个数据项。数据项是数据的不可分割的最小单位。数据元素也可以仅有一个数据项。

表 1-2 学生成绩表

学 号	姓 名	语 文	数 学	C 语言
6201001	张三	85	54	92
6201002	李四	92	84	64
6201003	王五	87	74	73
…	…	…	…	…

数据对象(data object):数据对象是性质相同的数据元素组成的集合,是数据的一个子集。数据元素是数据对象的数据成员。例如,正整数的数据对象是集合 N={1,2,3,4,…},字母字符数据对象是集合 N={'A','B','C',…,'Z'}。

表 1-2 所示的学生成绩表也是一个数据对象。每个学生的记录为一个数据元素。每个数据元素由学号、姓名、语文、数学、C 语言这些数据项组成。

数据结构(data structure):是相互之间存在一种或多种特定关系的数据元素的集合。在任何实际问题中,各数据元素之间都不可能是孤立的,它们之间总是存在着这样或那样的关系,这种数据元素之间的关系就称为结构。数据结构包括数据的逻辑结构和存储结构。

1.2.2 数据的逻辑结构

根据数据元素之间关系的不同,可以把逻辑结构分成以下 4 种形式:

(1)集合:数据元素间为松散的关系。在集合结构中各元素之间,除了“同属于某一个数据对象”的关系外,再无其他的关系,如图 1-4(a)所示。

(2)线性结构:数据元素间为严格的一对一关系,除第一个元素外,其他元素只有一个前驱,除最后一个元素外,其他元素只有一个后继,如图 1-4(b)所示。表 1-2 的学生成绩表也是一个线性结构,在这个结构中,数据元素(由一个人的学号、姓名和其成绩组成)的排列十分有序,第一个元素之后紧跟着第二个元素,第二个元素之后紧跟着第三个元素,以此类推,故称“线性结构”。

(3)树状结构:数据元素之间为严格的一对多的关系,即一个元素只有一个前驱,但可以有多个后继,如图 1-4(c)所示。这种结构像自然界中倒立的“树”,呈分支、层次状态。例如,家谱、行政组织结构等都可用树状结构来表示。

(4)图状结构:数据元素之间存在多对多的关系,也就是说元素间的逻辑关系可以是任意的。图状结构如图 1-4(d)所示。

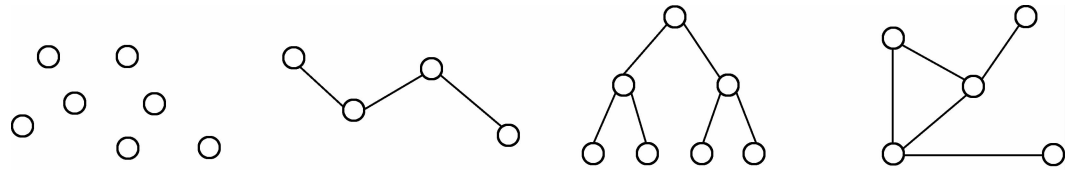


图 1-4 4 种结构示意图

任何数据对象中各数据元素之间都存在特定的关系,而这些存在着相互关系的数据元素的集合就是数据结构。简单地说,数据结构就是带有结构的数据元素的集合。

数据结构可以用二元组来进行数学意义上的形式定义:

$$Data\_Structure=(D,R)$$

其中,D 是数据元素的有限集,即数据对象;R 是 D 中所有数据元素之间的关系的有限集。

【例 1-4】 为毕业设计小组设计一个数据结构。假设每个小组的关系是 1 名教师指导 1～10 名学生,则数据结构的二元组表示如下:

$$Group=(P,R)$$

其中,$P=\{T,G_1,\cdots,G_n,1\leqslant n\leqslant 10\}$;$R=\{<T,G_i>|1\leqslant i\leqslant n,1\leqslant n\leqslant 10\}$。

上述数据结构的定义是一种数学意义上的定义。“数据结构”定义中的“关系”是指数据间的逻辑关系,故也称数据结构为逻辑结构。可将图 1-4 中的集合、树状结构、图状结构归纳为非线性结构。因此,数据的逻辑结构可分为两大类,即线性结构和非线性结构。

然而,讨论数据结构的目的是在计算机中实现对它的操作,因此还需研究如何在计算机中表示它,即数据的存储结构,又称物理结构。

1.2.3 数据的存储结构

数据元素之间的逻辑结构是根据实际问题的需要而选择设计的,因此是面向问题而与计算机本身无关的,或者说是不依赖于机器的。而数据元素的存储结构是指数据在计算机中的物理表示和存储的方式,涉及数据元素及其关系在存储器中的物理位置、机器响应的速度等方面的因素,因而物理结构的设计是与计算机本身密切相关的。在计算机中存储信息的最小单位叫做**位**(bit),8 位可表示一个**字节**(byte),两个字节称为一个**字**(word),字节、字或更多的二进制位可称为**位串**,这个位串称为**元素**(element)或**结点**(node)。当数据元素由若干个数据项组成时,则位串中对应于每个数据项的子位串称为**数据域**(data field)。

对于线性表,其逻辑结构是:除第一个元素外,其他元素只有一个前驱,除最后一个元素外,其他元素只有一个后继。它在计算机中的表示和存储有以下两种方式:用连续的存储单元存储;用分散的存储单元存储,并用指针将其连接。

数据元素之间的关系在计算机中有两种不同的存储方法:顺序存储和链式存储。

顺序存储:其方法是把数据元素依次存储在一组地址连续的存储单元中,元素间的逻辑关系由存储单元的位置直接体现,由此得到的存储表示称为顺序存储。高级语言中,常用一维数组来实现顺序存储。该方法主要用于线性结构,非线性结构也可通过某种线性化的处

理后来实现顺序存储。顺序存储结构的特点是所有元素存放在一片连续的存储单元中,逻辑结构上相邻的元素存放到计算机内后仍然相邻,如图 1-5 所示。

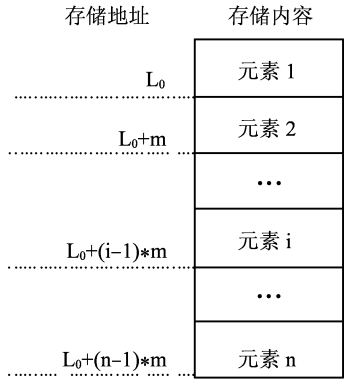


图 1-5 顺序存储结构示意图

链式存储:将数据元素存储在一组任意的存储单元中,而用附加的指针(pointer)表示元素之间的逻辑关系,由此得到的存储表示称为链式存储。存储结构的每个结点都由两部分组成:数据域和指针域。其中,数据域存放元素本身的数据,指针域存放指针,如图 1-6 所示。链式存储结构的特点是所有元素可以存放在不连续的存储单元中,元素之间的关系可以通过地址确定,逻辑结构上相邻的元素存放到计算机内后不一定是相邻的。

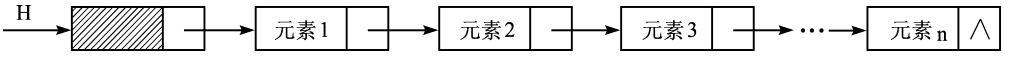


图 1-6 链式存储结构示意图

数据的逻辑结构与存储结构是密不可分的两个方面,任何一种算法的设计都取决于选定的数据的逻辑结构,而算法的实现则依赖于所采用的存储结构。

1.2.4 数据类型

如何描述存储结构呢? 虽然存储结构涉及数据元素及其在存储器中的物理位置,但由于本书是在高级程序设计语言——C 语言的层次上讨论数据结构的操作,所以不能直接以内存地址来描述存储结构,但可以借用高级程序设计语言中提供的“数据类型”来描述它,例如,可以用 C 语言中的“一维数组”来描述顺序存储结构。

1. 数据类型

数据类型是和数据结构密切相关的一个概念,几乎所有高级程序设计语言都提供这一概念。**数据类型**(data type)是一个值的集合和定义在这个值集上的一组操作的总称。例如,C 语言中的整型变量,其值集为某个区间上的整数,在典型的 16 位计算机中整型变量一般取−32 768～+32 767 的整数值,定义在其上的操作为加、减、乘、除和取模等运算;而一个布尔型的变量仅允许取 true 和 false 这两个值,定义在其上的操作为逻辑运算。

按“值”是否可分解,把数据类型分为两类:

(1)**原子类型**:每一个对象仅由单值构成的类型,其值不可分解,如 C 语言中的基本类型(整型、字符型、实型、枚举型)、指针类型和空类型。

(2)**结构类型**:每一个对象可由若干成分按某种结构构成,其值可分解成若干成分(或称分量)。如C语言中的数组类型就是一种结构类型,它由固定个数的同一类型顺序排列而成,数组类型中每一个数组值包含有固定个数的同一类型的数据,每个数据都可以通过下标直接访问。记录也是一种结构类型,它由固定个数的不同(也可以相同)类型数据项顺序排列而成,记录类型中的每一个记录值包含有固定个数的不同类型数据,每个数据也都可以直接访问。

在某种意义上,数据结构可以看成一组具有相同结构的值,而数据类型可以看成由一种数据结构和定义在其上的一组操作组成。

2.抽象数据类型

抽象数据类型(abstract data type,ADT)是指一个数学模型及定义在该模型上的一组操作。抽象数据类型不仅包括数据类型的定义,而且为这个新类型指明了一个有效的操作集合。抽象强调数据类型的数学特性,而不管它们在不同处理器上的实现方法和细节,即不论抽象数据类型内部结构如何变化,只要它的数学特性不变,都不影响其外部的使用。因此,抽象数据类型和数据类型实质上是同一个概念。例如,可以用线性表描述学生成绩表,用树描述遗传关系。

除此之外,抽象数据类型的范畴更广。它不仅仅局限于高级程序设计语言中已实现了的数据类型,还包括用户在设计软件系统时自己定义的数据类型。为了提高软件的复用率,近代程序设计方法学指出,一个软件系统的框架应建立在数据之上,而不是建立在操作之上,即在构成软件系统的每一个相对独立的模块中,定义一组数据和施于这些数据上的一组操作,并在模块内部给出这些数据的表示及实现细节,而在模块外部,只使用抽象数据和抽象操作。使用抽象数据类型可提高软件的复用程度。使用它的人可以只关心它的逻辑特征,不需要了解它的存储方式;定义它的人同样不必关心它如何存储。

线性表是一个抽象数据类型,其数学模型是:数据元素的集合,该集合内的元素有这样的关系——除第一个和最后一个元素,每个元素有唯一的前驱和唯一的后继。可以有这样一些操作——插入一个元素、删除一个元素等。

数据结构是一个二元组,它的逻辑定义是数据元素的集合以及数据元素间关系的集合。而在研究数据结构时更为重要的是要设计定义在其上的一组操作。如果把一个数据结构以及定义在其上的一组操作封装起来,使之成为一个抽象数据类型,则可以提高它们的复用率。

因此抽象数据类型一般可以由数据对象、数据关系及基本操作3个要素来定义。和数据结构的形式定义相对应,抽象数据类型可用以下三元组表示:

$$ADT=(D,S,P)$$

其中,D是数据对象,用数据元素的有限集合表示;S是D上的关系集,用结点间序偶的集合表示;P是对D的基本操作集。

根据上述定义,可用以下格式定义抽象数据类型:

```
ADT 抽象数据类型名{
    数据对象:<数据对象的定义>
    数据关系:<数据关系的定义>
    基本操作:<基本操作的定义>
}ADT 抽象数据类型名
```

其中,数据对象和数据关系的定义用伪代码描述;基本操作的定义格式为:

基本操作名(参数表)

初始条件:<初始条件描述>

操作结果:<操作结果描述>

基本操作有两种参数:赋值参数为操作提供输入值;引入参数以 & 开始,除了提供输入值外,还可以返回操作结果。“初始条件”说明了操作之前数据结构和参数应满足的条件,若不满足,则操作失败,并返回错误信息。“操作结果”说明了操作正常完成之后,数据结构的变化情况和应返回的结果。

【例 1-5】 使用三元组定义描述抽象数据类型栈。

ADT Stack{

数据对象:栈的数据对象是一个数据元素序列 $a_1,a_2,\cdots,a_n(n\geqslant 0)$,其中,数据元素 a_i 属于用户自定义的数据类型。

数据关系:栈是一种特殊的线性表,因此其数据元素也是一一对应关系,并且约定 a_n 端为栈顶,a_1 端为栈底。

基本操作:

InitStack(&S)

初始条件:栈 S 不存在。

操作结果:构造一个空栈 S。

DestoryStack(&S)

初始条件:栈 S 已存在。

操作结果:栈 S 被销毁。

StackEmpty(S)

初始条件:栈 S 已存在。

操作结果:若栈 S 为空,则返回 TRUE,否则返回 FALSE。

StackLength(S)

初始条件:栈 S 已存在。

操作结果:返回 S 的元素个数,即栈的长度。

GetTop(S,&e)

初始条件:栈 S 已存在且非空。

操作结果:用 e 返回 S 的栈顶元素。

ClearStack(&S)

初始条件:栈 S 已存在。

操作结果:将 S 清为空栈。

Push(&S,e)

初始条件:栈 S 存在。

操作结果:插入元素 e 为新的栈顶元素。

Pop(&S,&e)

初始条件:栈 S 已存在且非空。

操作结果:删除 S 的栈顶元素,并用 e 返回其值。

}ADT Stack

至于上述抽象数据类型如何具体实现,读者可以根据所学知识思考一下,在此不再赘述。

1.3 算法分析

基本操作是通过算法来描述的,因此,讨论算法是数据结构课程的重要内容之一。

1.3.1 算法的概念及描述

1.算法

算法(algorithm)是对特定问题求解步骤的描述,是指令的有限序列,其中每条指令表示一个或多个操作。对于实际问题,不仅要选择合适的数据结构,还要有好的算法,才能更好地求解问题。一个算法还需具备下列5个重要特性:

(1)有穷性:一个算法必须总是在执行有穷步之后结束,且每一步都可在有穷时间内完成。

(2)确定性:算法中的每条指令的含义都必须明确,无二义性。在任何情况下,对于相同的输入,必须能得出相同的输出。

(3)可行性:算法是可行的,即算法中描述的操作均可通过已经实现的基本运算的有限次执行来实现。

(4)输入:一个算法有零个或者多个输入,这些输入取自于某个特定的对象的集合。

(5)输出:一个算法至少有一个输出,这些输出是同输入有着某些特定关系的量。没有输出的算法是没有意义的。

2.算法的描述

可以借助各种工具来描述算法,通常有以下几种描述工具:自然语言、流程图、形式化语言、具体的程序设计语言等。一般来说,用自然语言来描述的算法,易于理解但不直观,对于复杂的算法自然语言难以描述;用流程图来描述的算法比较直观,但移植性差;用形式化语言来描述的算法能方便地表达思想,但不能直接在机器上运行;用具体的程序设计语言来描述算法,编写的程序能直接运行,但受具体语言语法的限制。

本书采用的描述工具是类C语言,是在精选的C语言的一个核心子集上增加了若干扩充修改,从而提高了语言的描述功能。以下对其作简要说明:

(1)预定义常量和类型:

```
// 函数结果状态代码
#define  TRUE  1
#define  FALSE  0
#define  OK  1
#define  ERROR  0
#define  OVERFLOW  -2
// Status 是函数的类型,其值是函数结果状态代码
typedef int Status;
```

(2)数据结构的表示(存储结构)用类型定义(typedef)描述。数据元素类型约定为ElemType,由用户在使用该数据类型时自行定义。

(3)基本操作的算法都用以下形式的函数描述:

```
函数类型 函数名(函数参数表){
    // 算法说明
    语句序列
}// 函数名
```

一般而言,a、b、c、d、e 等用作数据元素名,i、j、k、l、m、n 等用作整型变量名,p、q、r 等用作指针变量名。当函数返回值为函数结果状态代码时,函数定义为 Status 类型。为了便于算法描述,除了值调用方式外,增添了 C++ 语言的引用调用的参数传递方式。在形式参数列表中,以 & 开头的参数即为引用参数。

(4)赋值语句有:

- 简单赋值语句:变量名=表达式;
- 串联赋值语句:变量名 1=变量名 2=…=变量名 k=表达式;
- 成组赋值语句:(变量名 1,…,变量名 k)=(表达式 1,…,表达式 k);

```
结构名=结构名;
结构名=(值 1,…,值 k);
变量名[ ]=表达式;
变量名[起始下标..终止下标]=变量名[起始下标..终止下标];
```

- 交换赋值语句:变量名<——>变量名;
- 条件赋值语句:变量名=条件表达式? 表达式 T:表达式 F;

(5)选择语句有:

- 条件语句 1:if(表达式) 语句;
- 条件语句 2:if(表达式) 语句;

```
else 语句;
```

- 开关语句:switch(表达式){

```
case 值 1:语句序列 1;break;
…
case 值 n:语句序列 n;break;
default:语句序列 n+1;
}
```

(6)循环语句有:

- for 语句:for(赋初值表达式序列;条件;修改表达式序列) 语句;
- while 语句:while(条件)语句;
- do…while 语句:do{

```
语句序列;
}while(条件);
```

(7)结束语句有:

- 函数结束语句:return 表达式;

```
return;
```

- case 结束语句:break;
- 异常结束语句:exit(异常代码);

(8)输入和输出语句有:

- 输入语句:scanf([格式串],变量 1,…,变量 n);
- 输出语句:printf([格式串],表达式 1,…,表达式 n);

(9)注释:

- 单行注释: // 文字序列

(10)基本函数有:

- 求最大值:max(表达式 1,…,表达式 n)
- 求最小值:min(表达式 1,…,表达式 n)
- 求绝对值:abs(表达式)
- 文件结束:eof(文件变量)或 eof
- 行结束:eoln(文件变量)或 eoln

(11)逻辑运算约定:

- 与运算 &&:对于 A&&B,当 A 的值为 0 时,不再对 B 求值。
- 或运算 ‖:对于 A‖B,当 A 的值为非 0 时,不再对 B 求值。

【例 1-6】 计算两数中的较大数,可以用类 C 语言描述。

```
Status max(int a,int b){
    if(a>=b)return(a);
    return(b);
}
void main(){
    int a=3,b=4,c;
    c=max(a,b);
    printf("The Max is %d",c);
}
```

1.3.2 算法的复杂度分析

1.算法的时间复杂度

算法的执行时间需通过依据该算法编制的程序在计算机上运行时所消耗的时间来度量。度量一个程序的执行时间通常有两种方法:

(1)事前分析估算法。用数学方法直接对算法的效率进行分析。因为这种分析方法是在计算机实际运行根据该算法编制的程序前进行的,所以称为事前分析估算法。事前分析估算法要考虑以下因素:

- 问题的规模,如对 100 个学生的成绩排序要比对 10 个学生的成绩排序花的时间多。
- 编写程序时所用的程序设计语言,对于同一个算法,编写算法的语言级别越高,执行的效率越低。
- 机器的速度,很明显用 386 机器去执行一个算法要比用奔腾 4 机器慢得多。

- 算法所用的策略。

(2)事后统计法。计算机内部均设有计时功能,可设计一组或若干组测试数据,然后分别运行根据不同的算法编制的程序,并比较这些程序的实际运行时间,从而确定算法的优劣。但这种方法有两个缺陷:一是必须先运行根据算法编制的程序,而这通常是比较麻烦的;二是这种方法得出的结果往往对计算机的硬件、软件环境依赖性较强,有时容易掩盖算法本身的优劣。

此外,编译程序时所产生的机器代码的质量也会对执行效率产生影响。显然,在各种因素都不能确定的情况下,用绝对的机器运行时间来衡量算法的效率是不科学的,撇开这些与机器软硬件有关的因素,可以认为一个特定算法"运行工作量"的大小只依赖于问题的规模,或者说只是问题规模的函数。

算法由控制结构和原操作构成,其执行的时间取决于两者的综合效果。为了客观地比较同一问题的不同算法的优劣,通常从算法中选取一种对于所研究的问题来说是基本运算的原操作,用该原操作执行的次数作为算法的时间度量。

一般情况下,算法中基本原操作重复执行的次数是问题规模 n 的某个函数 f(n),因而算法的时间复杂度记作:

$$T(n)=O(f(n))$$

当问题的规模 n 增大时,算法执行时间的增长率和 f(n)的增长率相同,称为算法的渐近时间复杂度(asymptotic time complexity),简称时间复杂度。

当算法的时间复杂度T(n)与数据个数 n 无关时,算法的时间复杂度 $T(n)=O(1)$,称为常量阶;当算法的时间复杂度 T(n)与数据个数 n 为线性关系时,此时算法的时间复杂度 $T(n)=O(n)$,称为线性阶;当算法的时间复杂度 T(n)与数据个数 n 为平方关系时,算法的时间复杂度 $T(n)=O(n^2)$,称为平方阶。此外,算法还可能呈现的时间复杂度有对数阶 $O(\log n)$、指数阶 $O(2^n)$等。可见,分析一个算法中基本语句执行次数与数据个数 n 的函数关系,就可求出该算法的时间复杂度。通常把基本语句重复执行的次数称为语句的频度。例如,在下列 3 个程序段中:

- {++x;sum=0;}
- for(i=1;i<=n;++i){++x;sum+=x;}
- for(j=1;j<=n;++j)
  for(k=1;k<=n;++k){++x;sum+=x;}

含基本操作"x 增 1"的语句的频度分别为 1、n 和 n^2,则这 3 个程序段的时间复杂度分别为 $O(1)$、$O(n)$和 $O(n^2)$。下面看两个例子。

【例 1-7】 求两个 n 阶方阵的乘积 C=A * B 的算法。

```
#define n 10
void MultiMatrix(int A[n][n],int B[n][n],int C[n][n]){
    for(i=0;i<n;i++)                                    ①
        for(j=0;j<n;j++){                               ②
            C[i][j]=0;                                  ③
            for(k=0;k<n;k++)                            ④
                C[i][j]=C[i][j]+A[i][k]*B[k][j];        ⑤
        }
```

```
}
```

语句①到语句⑤执行的次数依次是 n、n^2、n^2、n^3、n^3。这些语句频度之和就是该算法的时间开销，即时间复杂度：

$$T(n)=2n^3+2n^2+n$$

当 n 充分大的时候，$T(n)$ 与 $f(n)=n^3$ 的数量级相同。于是，该算法的时间复杂度 $T(n)=O(n^3)$，其原语句是语句⑤。

【例 1-8】 下面的算法是用冒泡排序法对数组 a 中的 n 个整型元素从小到大进行排序，求该算法的时间复杂度。

```
void BubbleSort(int a[ ],int n){
    for(i=n-1,change=TRUE;i>1 && change;--i){
        change=FALSE;
        for(j=0;j<i;++j)
            if(a[j]>a[j+1]){a[j]<-->a[j+1];change=TRUE;}
    }
}
```

其算法思想是：从 a[0]起依次比较相邻两个数 a[j]和 a[j+1](j=0,1,…,n-2)，若前一个数比后一个数大，则两者相互交换位置，如此从前往后检查一遍，必然将其中值最大的数交换到 a[n-1]的位置上。之后对 a[0..n-2]进行同样操作，直至所检查的区间长度到 1 为止。当 a 中初始序列为自小至大有序时，基本操作的执行次数为 0；当初始序列为自大至小有序时，基本操作的执行次数为 $n(n-1)/2$。这种情况下算法的时间复杂度除了依赖于所要解决的问题的规模外，还与问题的实际输入情况有关。

对此类算法的分析，一种解决的办法是计算它的平均值，即考虑它对所有可能的输入数据集的期望值，此时相应的时间复杂度为算法的平均时间复杂度。然而，在很多情况下，各种输入数据集出现的概率难以确定，算法的平均时间复杂度也就难以确定。因此，另一种更可行也更常用的办法是讨论算法在最坏情况下的时间复杂度，即分析最坏情况以估算算法执行时间的一个上界。在本书以后各章中讨论的时间复杂度，除特别指明外，均指最坏情况下的时间复杂度。因此，最坏情况下该算法的时间复杂度分析如下。

设基本操作的频度为 $f(n)$，最坏情况下有：

$$f(n)=n+4\times n^2/2$$

所以该算法的最坏时间复杂度为 $T(n)=O(n^2)$。

实践中可以把事前估算和事后统计两种办法结合起来使用。以两个矩阵相乘为例，若上机运行两个 10×10 的矩阵相乘，执行时间为 12 ms，则由算法的时间复杂度 $T(n)=O(n^3)$ 可估算两个 27×27 的矩阵相乘所需时间为 $(27/10)^3\times12$ ms$\approx$236 ms。

2. 算法的空间复杂度

类似于算法的时间复杂度，本书以空间复杂度(space complexity)作为算法所需存储空间的量度，记作：

$$S(n)=O(f(n))$$

其中，n 为问题的规模。请注意，这里所说的算法所需的存储空间，通常不含输入数据和程序本身所占的存储空间，而是指算法对输入数据进行运算所需的辅助工作单元和存储实现计算所需信息的辅助性空间，这类空间也称额外空间。算法的输入数据所占的空间一般是

由具体问题决定的,不会因算法不同而改变。算法本身占用的空间不仅和算法有关,而且和编译程序产生的目标代码的质量有关,所以也难以计算。算法所占的额外空间却是与算法的质量密切相关,好的算法既节省时间又节省额外空间。有时,算法的输入数据所占的空间不是由问题本身决定的,而是由算法决定的,在这种情况下,算法所需的存储空间应包括输入数据的存储空间。

1.3.3 算法的设计要求

实际应用中的算法及其涉及的数据结构是比较复杂的。在这里有必要了解一下被计算机界公认的著名公式:

程序=算法+数据结构

它是由瑞士科学家 Niklaus Wirth 提出的,从逻辑上描述了算法与数据结构和程序之间的关系。由此可见,当选好了数据的逻辑结构和物理结构之后,如果没有行之有效的算法,则不能对数据进行操作,不能解决实际问题,同样,没有数据结构,算法也就无用武之地了。

在计算机内如何组织和存储一个给定问题的初始数据,使其既节省空间,又便于加工处理;同时,如何选择与设计相适应的算法,以提高求解问题的效率和可靠性,这些都是至关重要的。

对于某一个特定问题的求解,究竟采用何种数据结构以及选择何种算法,需要对该问题进行分析。根据问题的具体要求和现实环境的各种条件,把数据结构与算法有机地结合起来,这样才能设计出高质量的计算机程序。算法必须采用与之相适应的数据结构,才能有效地解决所求解的问题。

同一个问题可能有许多求解的算法。在利用计算机解决这些问题时,就需要对这些求解算法进行筛选,从中挑选一个最好的算法来解决问题。算法选择得是否恰当将直接影响问题解决的效果。一个好的算法通常应注意以下方面:

(1)正确性(correctness):算法本身应该没有语法错误,一个有语法错误的程序是不能在计算机上运行的。这里所说的正确性主要指算法没有逻辑错误,即对一切合法的输入数据都能产生满足要求的输出结果。显然,要验证一个程序完全正确是件极为困难的事情,目前也只处于理论研究阶段。普通意义上算法的正确性是指算法的执行能达到预期的目的,并且对于精心选择的典型、苛刻而带有刁难性的输入数据也能够得出符合要求的结果。

(2)可读性(readability):算法主要是为了人的阅读与交流,其次才是机器执行。可读性好有助于人对算法的理解,能方便人们的交流与对算法的改进。所以,一个好的算法应该是易读易懂的。必要的注释是增强算法可读性的手段之一。

(3)健壮性(robustness):当输入数据非法或运行环境改变时,算法也能作出快速、恰当的反应,不会产生莫名其妙的输出结果,同时,输出一定的错误提示,终止程序的执行。例如,一个求 3 个整数最大值的算法。当输入的参数集合是字符集合时,不应继续计算,而应报告输入出错。

(4)效率与存储量要求:算法的效率指的是算法的执行时间。算法在执行过程中所占用的最大存储空间叫做算法的空间存储量,简称存储量。算法的效率和存储量是选择算法的主要依据。一个好的算法要求其执行时间尽可能得短,占用的存储空间尽可能得少。但是这两方面往往是相互矛盾的,节省了时间可能牺牲空间,反之亦然。在选择算法时,必须权衡这两个方面的因素。

1.4 实例解析

【例 1-9】 设有数据逻辑结构为:tree=(D,R),其中

D={A,B,C,D,E,F,G,H,I,J}

R={<A,B>,<A,C>,<A,D>,<B,E>,<B,F>,<C,G>,<C,H>,<C,I>,<D,J>}

试分析该数据结构属于哪种逻辑结构?

数据的树状结构示意图如图 1-7 所示。

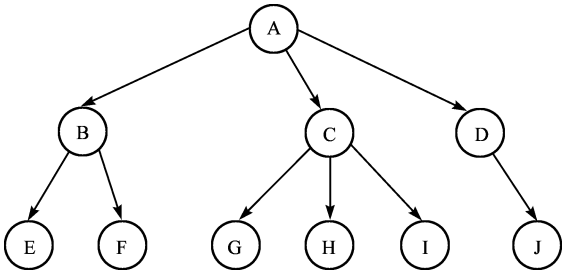


图 1-7 数据的树状结构示意图

图 1-9 像倒着画的一棵树,在这棵树中,最上面的一个结点没有前驱只有后继,称为树根结点,最下面一层的结点只有前驱没有后继,称为树叶结点。在一棵树中,每个结点有且只有一个前驱结点(除树根结点外),但可以有任意多个后继结点(树叶结点可看做具有 0 个后继结点)。这种数据结构的特点是数据元素之间为 1 对 N 关系(N≥0),即层次关系,因此本题所给定的数据结构为树状结构。

【例 1-10】 设一个数据结构的逻辑结构如图 1-8 所示,写出其二元组的描述。

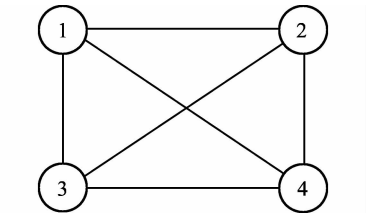


图 1-8 某数据结构的逻辑结构

本题所给定的数据结构为图状结构,其二元组的描述为:

Graph=(D,R)

其中,D={1,2,3,4},R={(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}。

注意:在【例 1-9】中尖括号<A,B>表示的是一条有向边,在【例 1-10】中圆括号(1,2)表示的是一条无向边。

【例 1-11】 设 n 为一个正整数,求下列各算法的时间复杂度。

```
(1)void prime(int n){
        i=2;
        while((n%i)!=0 && i*1.0 < sqrt(n))
```

```
            i++;
        if(i*1.0>sqrt(n))
            printf("%d是一个素数\n",n);
        else
            printf("%d不是一个素数\n",n);
    }
(2)int sum1(int n){
        p=1;sum=0;
        for(i=1;i<=n;i++){
            p*=i;
            sum+=p;
        }
        return sum;
    }
(3)int sum2(int n){
        sum=0;
        for(i=1;i<=n;i++){
            p=1;
            for(j=1;j<=i;j++)
                p*=j;
            sum+=p;
        }
        return sum;
    }
```

算法的时间复杂度是由嵌套最深层语句的执行次数决定的。

(1)prime 算法的嵌套最深层语句为:

```
i++;
```

它的执行次数由条件((n%i)!=0 && i*1.0 < sqrt(n))决定。显然执行次数小于 sqrt(n),所以 prime 算法的时间复杂度是 $O(n^{1/2})$。

(2)sum1 算法的嵌套最深层语句为:

```
p*=i;
sum+=p;
```

它们的执行次数均为 n 次,所以 sum1 算法的时间复杂度是 O(n)。

(3)sum2 算法的嵌套最深层语句为:

```
p*=j;
```

它的执行次数为 1+2+3+…+n=n(n+1)/2 次,所以 sum2 算法的时间复杂度是 $O(n^2)$。

本章小结

数据是对客观事物的符号表示，是计算机程序加工的“原料”。

数据元素是数据的基本单位，可由若干个数据项组成，数据项是数据的不可分割的最小单位。

数据对象是性质相同的数据元素组成的集合，是数据的一个子集。

数据结构包含数据元素及数据元素之间的关系。数据的逻辑结构通常有4种，即集合、线性结构、树状结构和图状结构。存储结构即数据的物理结构，是数据结构在计算机中的表示，包括数据元素的表示和关系的表示，主要有顺序存储结构和链式存储结构。

数据类型是一个值的集合和定义在这个值集上的一组操作的总称。数据类型有两种，即原子类型和结构类型。

抽象数据类型是指一个数学模型及定义在该模型上的一组操作。

算法是对特定问题求解步骤的描述，是指令的有限序列，其中每条指令表示一个或多个操作。算法具有有穷性、确定性、可行性、输入和输出5个特性。好的算法应具有较高的正确性、较强的可读性、较好的健壮性、高效率与低存储量需求的特点。

一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数f(n)，算法的时间量度记作T(n)=O(f(n))，T(n)就为该算法的时间复杂度。有时要考虑平均时间复杂度和最坏时间复杂度。

空间复杂度是算法所需存储空间的量度，记作S(n)=O(f(n))，其中n为问题的规模。

习 题 1

1. 数据的逻辑结构有哪几种？常用的存储结构有哪几种？
2. 举一个数据结构的例子，叙述其逻辑结构、存储结构和运算三方面的内容。
3. 什么叫算法？它有哪些特性？
4. 按增长率由小到大排列下列各函数：

   $(3/4)^n$，$(4/3)^n$，n^n，$n!$，2^n，$\log_2 n$，$n^{3/2}$
5. 设有如图1-9所示的逻辑结构示意图，分析它的逻辑结构。

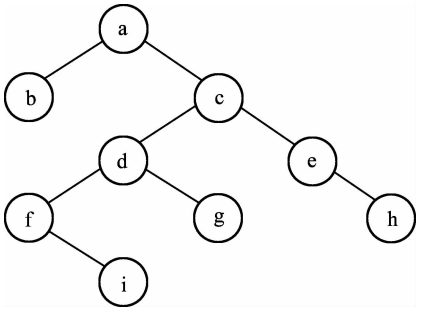


图1-9 逻辑结构示意图

6. 有下列几种用二元组表示的数据结构,画出它们分别对应的逻辑结构图,并指出它们分别属于何种结构。

(1)A=(K,R),其中

K={a,b,c,d,e,f,g,h}

R={r}

r={<a,b>,<b,c>,<c,d>,<d,e>,<e,f>,<f,g>,<g,h>}

(2)B=(K,R),其中

K={a,b,c,d,e,f,g,h}

R={r}

r={<d,b>,<d,g>,<d,a>,<b,c>,<g,e>,<g,h>,<e,f>}

(3)C=(K,R),其中

K={1,2,3,4,5,6}

R={r}

r={(1,2),(2,3),(2,4),(3,4),(3,5),(3,6),(4,5),(4,6)}

7. 设 n 为整数,求下列各程序段的时间复杂度。

(1)
```
i=1;k=2;
while(i<n){
    k=k+10*i;
    i=i+1;
}
```

(2)
```
i=1;j=0;
while(i+j<=n)
    if(i>j)j=j+1;
    else i=i+1;
```

(3)
```
x=91;y=100;
while(y>0)
    if(x>100){
        x=x-10;
        y=y-1;
    }
    else x=x+1;
```

(4)
```
x=n;
while(x>=(y+1)*(y+1))
    y=y+1;
```

第2章　线　性　表

第2章到第4章，将讨论线性结构。线性结构的基本特征是：在数据元素的非空有限集中，数据元素间存在着一一对应的逻辑关系，即

(1)存在唯一的一个没有直接前驱的数据元素，称为“第一元素”，除此之外，其余元素有且仅有一个直接前驱。

(2)存在唯一的一个没有直接后继的数据元素，称为“最后元素”，除此之外，其余元素有且仅有一个直接后继。

线性表、栈、队列和串的逻辑结构都属于线性结构。线性表是一种最简单的线性结构。

2.1　线性表的基本概念

线性表是计算机程序设计中经常要操作的对象，在大多数情况下，这些表并非是无组织的数据集，而是在数据元素间存在着某种结构关系。

2.1.1　线性表的定义及特性

线性表是最常见、最简单的一种具有线性特征的数据结构。简单地说，线性表就是具有相同特性的数据元素的有限集合，其元素可以是数值、符号或是由许多不同的数据项组成的复杂信息。线性表中数据元素的类型可以根据需要进行具体定义，但在同一个线性表中每个元素都应具有相同的数据类型。

例如，英文字母表(a,b,c,d,…,z)是一个线性表，表中的每一个英文字母是一个数据元素；又如一副扑克牌的点数(2,3,4,5,6,7,8,9,10,J,Q,K,A)可以构成一个线性表，其中每一张牌的点数是一个数据元素。

在较为复杂的线性表中，一个数据元素可以由若干个数据项组成。在这种情况下，常把数据元素称为记录(record)，或称结点(node)，由大量记录构成的线性表又称为文件(file)。

例如，计算机系某一个班级的学生成绩表如表2-1所示，每个学生的成绩情况是一个数据元素，即一个记录，它由学号、姓名、性别、各科成绩及总分数据项组成。

表2-1　计算机系某班学生成绩表

学　号	姓　名	性　别	高　数	物　理	英　语	C语言	汇　编	总　分
990001	凡大林	男	83	91	87	94	75	430
990002	程　婷	女	79	84	93	76	81	413
990003	聂小倩	女	86	74	95	79	88	422
990004	王文波	男	79	88	95	82	77	421
990005	董双飞	男	90	77	82	71	84	404

综上所述,线性表是 n(n≥0)个具有相同类型的数据元素 $a_1,a_2,\cdots,a_n$ 的有限序列。线性表可以表示成如下形式:

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

线性表具有如下特性:

(1)不同线性表中数据元素的类型可以是各种各样的,但同一线性表中的元素必须是同一类型的。

(2)在表中 a_{i-1} 领先于 a_i,a_i 领先于 a_{i+1},称 a_{i-1} 是 a_i 的直接前驱,a_{i+1} 是 a_i 的直接后继。

(3)在线性表中,除第一个元素和最后一个元素之外,其他元素都有且仅有一个直接前驱,有且仅有一个直接后继,这是所有线性结构的共同特征。线性表是一种线性结构。

(4)线性表中元素的个数 n 称为线性表的长度,n=0 时称为空表。

(5)元素 a_i 是线性表的第 i 个元素,称 i 为数据元素 a_i 的位序,每一个元素在线性表中的位置,仅取决于它的位序。

2.1.2 线性表的抽象数据类型

回顾第 1 章中有关抽象数据类型的定义可知,抽象数据类型是指一个数学模型及定义在该模型上的一组操作。因此,线性表的抽象数据类型可以由数据对象、数据关系及基本操作 3 个要素来定义。定义如下:

```
ADT List{
    数据对象:D={ai|ai∈ElemSet,i=1,2,…,n,n≥1}
    数据关系:R1={<ai-1,ai>|ai-1,ai∈D,i=2,…,n}
    基本操作:
      InitList(&L)
          初始条件:线性表 L 不存在。
          操作结果:构造一个空的线性表 L。
      ListEmpty(L)
          初始条件:线性表 L 已存在。
          操作结果:若 L 为空表,则返回 TRUE,否则返回 FALSE。
      ListLength(L)
          初始条件:线性表 L 已存在。
          操作结果:返回 L 中数据元素的个数。
      ListInsert(&L,i,e)
          初始条件:线性表 L 已存在,1≤i≤ListLength(L)。
          操作结果:在 L 中第 i 个位置之前插入新的数据元素 e,L 的长度加 1。
      ListDelete(&L,i,&e)
          初始条件:线性表 L 已存在且非空,1≤i≤ListLength(L)。
          操作结果:删除 L 的第 i 个数据元素,并用 e 返回其值,L 的长度减 1。
      GetElem(L,i,&e)
          初始条件:线性表 L 已存在,1≤i≤ListLength(L)。
          操作结果:用 e 返回 L 中第 i 个数据元素的值。
}ADT List
```

以上仅列出了线性表的一组最基本的操作。不同的应用问题中使用的线性表所需要执行的操作可能不同,通常的做法是给出一组最基本的操作,对于实际应用中所涉及到的其他更复杂的操作,可以用基本操作的组合来实现。

【例 2-1】 已知线性表 L 中的元素按元素值非递减有序排列,编写一个函数删除线性表中值相同的多余的元素。例如,设

L=(1,3,3,5,9,13,13,13,17,21)

执行过函数后,则

L=(1,3,5,9,13,17,21)

本题的算法思想是:由于线性表中的元素按非递减有序排列,值相同的元素必为相邻的元素,所以依次比较相邻的两个元素,若值相等,则删除其中的一个,否则继续向后查找,直到表中所有元素都查找结束。上述删除过程如算法 2.1 所示。

```
void DeleteSame(List &L){
    i=1;
    Len=ListLength(L);
    while(i<Len){
        GetElem(L,i,a1);
        GetElem(L,i+1,a2);
        if(a1!=a2)  i++;
        else{ListDelete(L,i,a2);Len--;}
    }
}
```

算法 2.1

此算法的时间复杂度取决于抽象数据类型 List 定义中基本操作的执行时间。假设 GetElem、ListLength 和 ListDelete 这 3 个基本操作的执行时间和表长无关,则算法 2.1 的时间复杂度为 O(ListLength(L))。

前面已经给出了线性表的抽象数据类型定义。那么如何在计算机中存储线性表? 如何在计算机中实现线性表的基本操作?

要存储线性表,至少要保存两类信息:

(1)线性表中的数据元素。

(2)线性表中数据元素的顺序关系。

下面分别研究线性表的顺序存储和链式存储及操作。

2.2 线性表的顺序存储及操作

一种数据结构的顺序实现,是指按照顺序存储方式建立数据结构的存储结构,并在这个顺序存储结构上实现数据结构的基本运算,即给出实现这些运算的算法。

2.2.1 顺序表

顺序表是指采用顺序存储结构的线性表,即用一组地址连续的存储单元依次存放线性

表中的数据元素。这种存储方法是借助数据元素在计算机内的物理位置表示线性表中数据元素之间的逻辑关系,即逻辑关系相邻的两个数据元素在计算机内的存储位置也相邻。

因为线性表的数据元素属于同一数据类型,所以每个元素所占用的存储空间的大小是相同的。假设每个元素占用 l 个存储单元,并以所占的第一个单元的存储地址作为数据元素的存储位置,则线性表中第 i 个数据元素的存储位置 Loc(a_i)可以由下式计算:

$$Loc(a_i)=Loc(a_1)+(i-1)\times l \qquad (1\leqslant i\leqslant n)$$

其中,n 是线性表的长度;Loc(a_1)是线性表的存储空间的首地址,又称基地址。

从上式可见,在顺序表中,元素 a_i 的存储地址是该元素在表中的序号 i 的线性函数,只要确定了基地址和每个元素占用的存储单元,就可以随机存取线性表中的任意一个数据元素,所以,顺序表是一种随机存取结构。

由于高级程序设计语言中的数组类型也具有随机存取的特性,所以通常都用数组来描述顺序表,顺序表的长度(即表中元素的数目)可用一个整型变量来表示,描述如下:

```
#define LIST_SIZE  100              // 顺序表的存储空间
typedef struct{
    ElemType *elem;                 // 存储空间基址
    int length;                     // 当前顺序表长度
}SqList;
```

上述定义中,常量 LIST_SIZE 是顺序表存储空间的大小,这里假设为 100,实际使用时其值可根据问题规模大小动态设置。数组指针 elem 指示顺序表的基地址,length 指示顺序表的当前长度。SqList 是自定义数据类型,可以用它说明一个顺序表变量,例如:

```
SqList L;
```

则 L. length 表示顺序表 L 的长度,L. elem 表示顺序表 L 的基地址。顺序表 L 的起始元素是 L. elem[0],终端元素是 L. elem[L. length-1],第 i 个元素是 L. elem[i-1]。若 L 是 SqList 类型的指针变量,则第 i 个元素是 L->elem[i-1]。

上述定义的顺序表,其存储空间由编译程序分配,称静态分配,需要在使用前分配好存储空间。假设表中已存放了 n 个元素,即 $a_1,a_2,\cdots,a_n$,则顺序表中数据元素在内存中的表示如图 2-1 所示。其中,b 表示数组 elem 的基地址,l 是该数组元素应占的存储单元数。

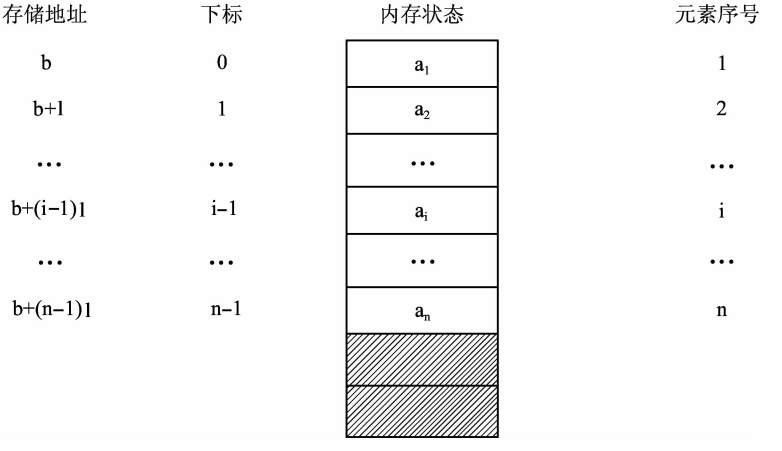


图 2-1 顺序表存储结构示意图

2.2.2 顺序表的基本操作

使用顺序存储结构，容易实现线性表的某些操作。例如，求表长操作只需返回L.length。下面着重介绍初始化、插入、删除操作。

1.初始化操作

顺序存储结构的线性表在使用前必须进行初始化操作，即需要为其分配好内存空间。具体过程如算法2.2所示。

```
Status InitList_Sq(SqList &L){
    // 构造一个空的线性表
    L.elem=(ElemType * )malloc(LIST_SIZE * sizeof(ElemType));
    if(!L.elem)exit(OVERFLOW);
        L.length=0;
    return OK;
}// InitList_Sq
```

算法 2.2

在初始化操作中分配空间使用了malloc函数，格式为malloc(size)，其功能是在系统内存中分配size个存储单元，并返回该空间的基地址。与其对应，释放空间的是free函数，格式为free(p)，其功能是将指针变量p所指示的存储空间回收到系统内存空间中去。

2.插入操作

顺序表的插入操作是指在表的第i(1≤i≤n+1)个位置上插入一个新元素e，使长度为n的线性表

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

变为长度为n+1的顺序表

$$(a_1,\cdots,a_{i-1},e,a_i,\cdots,a_n)$$

数据元素a_{i-1}和a_i之间的逻辑关系发生了变化，由$<a_{i-1},a_i>$变为$<a_{i-1},e>$，$<e,a_i>$。由于顺序表中逻辑上相邻的元素的存储位置也必须相邻，所以除非i=n+1，否则必须将表中位置为n，n−1，…，i的元素依次后移一个位置，空出第i个位置，然后在该位置上插入新元素e，插入过程如图2-2所示。如果i=n+1，则不需移动元素，直接把e插入第n+1个位置即可。

算法思想如下：在插入前先判断插入位置是否合法及是否有剩余空间。如果条件满足则定位要插入的位置，然后把第i个元素及其后的元素都向后移动一个位置，在原来的第i个位置上插入新元素，最后对表长进行处理。具体过程如算法2.3所示。

```
Status ListInsert_Sq(SqList &L,int i,ElemType e){
    // 在顺序表L的第i个元素之前插入新的元素e
    // i的合法值为1≤i≤ListLength_Sq(L)+1
    if(i<1||i>L.length+1)return ERROR;      // i值不合法
    if(L.length>=L.listsize)return OVERFLOW; // 存储空间已满，则退出
    q=&(L.elem[i-1]);                        // q为插入位置
    for(p=&(L.elem[L.length-1]);p>=q;--p)   *(p+1)=*p;
```

```
                                                    // 插入位置及之后的元素右移
    *q=e;                                           // 插入 e
    ++L.length;                                     // 表长增 1
    return OK;
}// ListInsert_Sq
```

算法 2.3

此算法需要注意以下几点：

(1)由于函数要回传对顺序表插入后的结果，所以参数 L 要定义为引用参数，即 SqList &L。如果把 L 定义为 SqList 类型是错误的，将无法对 L 进行改变。

(2)为了避免元素间的覆盖，算法中元素的后移必须从表中最后一个元素开始，直到第 i 个元素为止。

(3)线性表中元素的序号比数组元素下标大 1。

3. 删除操作

线性表的删除操作是使长度为 n 的线性表

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

变为长度为 n－1 的线性表

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

数据元素 a_{i-1}、a_i 和 a_{i+1}之间的逻辑关系发生变化。当 i＝n 时，可直接删除第 i 个元素；否则，必须将表中位置为 i＋1，…，n 的元素依次前移到第 i，…，n－1 个元素的位置上，以填补由于删除操作造成的空缺。删除过程如图 2-3 所示。

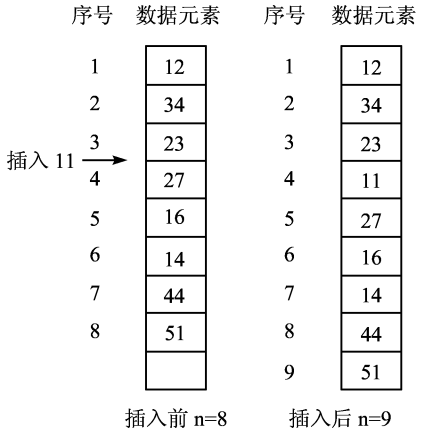


图 2-2 顺序表中插入元素前后的状况

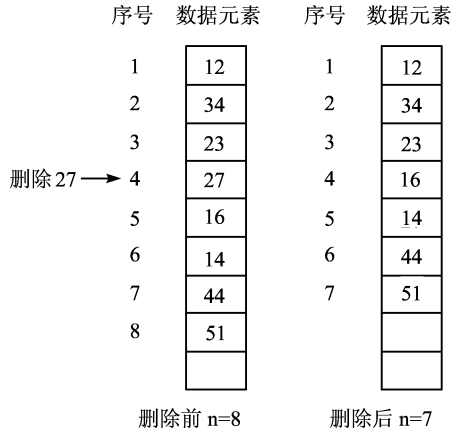


图 2-3 顺序表中删除元素前后的状况

算法思想如下：在删除前先判断删除位置是否合法。若合法则定位要删除的位置，并把要删除的元素值赋给 e，然后把第 i＋1 个元素及其后的元素都向前移动一个位置，最后对表长进行处理。具体过程如算法 2.4 所示。

```
Status ListDelete_Sq(SqList &L,int i,ElemType &e){
    // 在顺序表 L 中删除第 i 个元素,并用 e 返回其值
    // i 的合法值为 1≤i≤ListLength_Sq(L)
    if(i<1 || i>L.length)return ERROR;          // i 值不合法
```

```
    p=&(L.elem[i-1]);                    // p为被删除元素的位置
    e= *p;                               // 被删除元素的值赋给e
    q=L.elem+L.length-1;                 // 表尾元素的位置
    for(++p;p<=q;++p)  *(p-1)= *p;       // 被删除元素之后的元素左移
    --L.length;                          // 表长减1
    return OK;
}// ListDelete_Sq
```

算法 2.4

注意:删除算法中元素的前移是从第 i+1 个元素开始,到第 n 个元素为止。

2.2.3 顺序表基本操作的算法分析

顺序表的初始化操作比较简单,下面只分析插入与删除操作的时间复杂度。

在顺序表中插入元素,时间主要耗费在移动元素上,而移动的次数取决于插入的位置以及表的长度。插入位置与移动元素个数的关系如图 2-4 所示。

	插入元素位置	移动元素个数
a_1	1	n
a_2	2	n-1
…	…	…
a_{i-1}	i-1	n-i+2
a_i	i	n-i+1
a_{i+1}	i+1	n-i
…	…	…
a_n	n	1

图 2-4 插入位置与移动次数的关系

假设 p_i 是在第 i 个元素之前插入元素的概率,则在长度为 n 的顺序表中插入一个元素,所需移动元素次数的数学期望值(平均次数)为:

$$E_i = \sum_{i=1}^{n+1}(n-i+1)\times p_i$$

假设在线性表的任何位置插入元素的概率相同,即

$$p_i=\frac{1}{n+1}$$

则有

$$E_i = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2}$$

删除第 i 个元素(下标为 i-1)需移动 n-i 次元素,设在相应位置删除元素的概率是 p_d,则删除的平均移动次数是:

$$E_d = \sum_{i=1}^{n}(n-i)\times p_d$$

假设在线性表的任何位置删除元素的概率相同,即

$$p_d=\frac{1}{n}$$

则有

$$E_d = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2}$$

可见,在顺序表中插入或删除一个数据元素平均需要移动表中一半的元素。若表长为 n,则插入操作和删除操作算法的平均时间复杂度为 O(n)。

现在来讨论【例 2-1】的操作在顺序表中的实现方法和时间复杂度的分析。容易看出,顺序表的"求表长"和"取第 i 个元素"的时间复杂度均为 O(1),则其主要时间都消耗在元素删除上,算法 2.5 是【例 2-1】在顺序表中的实现算法。

```
void DeleteSame(List &L){
    p=L.elem;
    while(p<L.elem+L.length-1)
        if(*p!=*(p+1))  p++;
        else{
            for(++p;p<L.elem+L.length-1;++p)
                *(p-1)=*p;
            --L.length;
        }
}
```

算法 2.5

假设线性表 L 的长度为 n,由算法 2.5 可得,【例 2-1】的操作在顺序表中实现的时间复杂度为 $O(n^2)$。在这个算法中要注意的一点是,由于存在元素的删除操作,表长 L.length 会发生变化。

2.3 线性表的链式存储及操作

在 2.2 节介绍了线性表的顺序存储结构,其特点是用元素物理位置上的邻接关系来表示元素间的逻辑关系。因此可以随机访问顺序表中的任意元素,但其插入、删除操作效率较低。并且由于数组的存储空间采用静态分配,表的最大长度必须预先确定。要预先确定表的最大长度有时是件困难的事,因为估计过小会表满溢出,估计过大又会造成存储空间的浪费。

本节将介绍线性表的另一种存储结构——链式存储结构。该结构可以克服顺序表的上述困难,但随之而来的却是随机存取性能的消失。采用链式存储的线性表简称为链表。链式存储结构是最常用的一种存储结构,它不仅可以用来表示线性表的数据结构,也可以用来表示各种非线性表的数据结构。

2.3.1 单链表及其基本操作

1. 单链表

线性表的顺序存储结构是用一组地址连续的存储单元依次存放线性表的数据元素,因此逻辑上相邻的元素物理位置也相邻。而线性表的链式存储结构是用一组地址任意的存储

单元存放线性表中的数据元素。用链式存储结构存储数据元素 a_i 时，除了存储数据元素本身的信息之外，为了表示数据元素 a_i 与其直接后继元素 a_{i+1} 之间的逻辑关系，还需存储一个指示其后继元素存储位置的信息。这两部分信息一起组成了链表的一个结点。其中，存储数据元素信息的域称为数据域，存储直接后继存储位置的域称为指针域。指针域中存储的信息称为指针或链。每个结点只包括一个指针域的链表称为线性链表或单链表。单链表中一个结点的结构如图 2-5 所示。

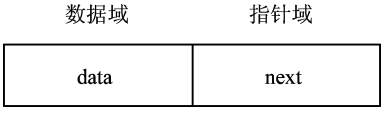


图 2-5　单链表结点结构

图 2-5 中，data 域为数据域，next 域为指针域。n 个结点连接成一个链表，即为线性表的链式存储结构。

由于在单链表中，逻辑上相邻的两个数据元素物理位置不要求相邻，每个结点的地址都是放在其前驱结点的指针域中，因此整个链表的存取必须从头指针开始进行，头指针是存放链表中第一个数据元素存储位置的指针。同时，最后一个结点无后继，则最后一个结点的指针域应是空指针。在类 C 语言中，空指针用符号常量 NULL 表示，在图示时用 ∧ 表示。

例如，如图 2-6 所示为线性表(ZI，CHOU，YIN，MAO，CHEN，SI，WU，WEI)的单链表存储结构。

头指针 H

31

存储地址	数据域	指针域
1	MAO	43
7	CHOU	13
13	YIN	1
19	WEI	NULL
25	SI	37
31	ZI	7
37	WU	19
43	CHEN	25

图 2-6　单链表示例

通常把链表画成用箭头连接的结点序列，结点之间的箭头表示链域中的指针。如图 2-6 所示的单链表可画成如图 2-7 所示的形式。这样做是因为在使用链表时，关心的只是它所表示的线性表中数据元素之间的逻辑顺序，而不是每个数据元素在存储器中的实际位置。

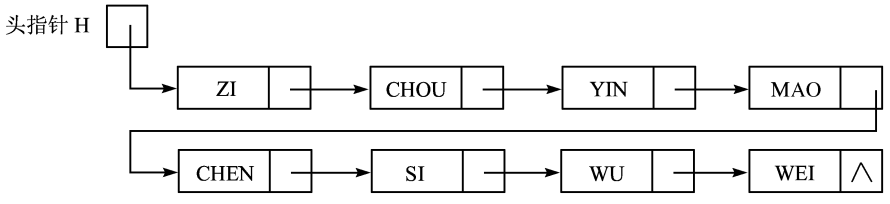


图 2-7　单链表的逻辑状态

通常使用结构指针来描述单链表，具体描述如下：

```
typedef struct LNode{
    ElemType data;              // 数据域
    struct LNode * next;        // 指针域
}LNode, * LinkList;
```

在上述定义中定义了一个结构类型 LNode,它包含两个域:数据域 data 与指针域 next;又定义了结构类型 LNode 的指针类型 LinkList,用于存放 LNode 结构类型变量的地址,即 LinkList 和 LNode * 是等价的。例如,若 L 是 LinkList 类型的变量,则 L 只能指向 LNode 结构类型的某一结点。

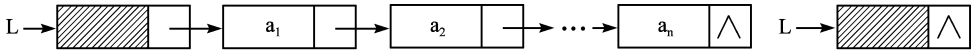
本书以后会反复使用指针,使用时要注意以下几点:

(1)要严格区分指针变量和它所指向的结点变量这两个概念。指针变量定义后,要使它有确定的指向必须给它赋值或者使用 malloc 函数为指针变量分配一个结点空间,这就是说指针变量所指向的结点变量在程序的执行过程中,当需要时才产生,故称动态变量。若指针变量所指向的结点不再需要,可通过 free 函数来释放其所指的结点变量所占的空间。

(2)若指针变量 p 的值为空指针 NULL,则 p 不指向任何结点。此时,若试图通过 * p 来访问结点就会引起程序错误,因为变量 * p 并不存在。

(3)不要使用无确定指向的指针。指针变量定义之后,如果编译程序不为它赋值,则它的指向就是不确定的。使用无确定指向的指针引起的程序错误不但不易查出,而且可能危及系统的安全。

有时为了操作方便,在第一个结点之前需加一个头结点,并以指向头结点的指针为链表的头指针。通常,头结点的数据域不存储任何信息。如图 2-8(a)所示,单链表的头指针指向头结点。若线性表为空表,则头结点的指针域为空,如图 2-8(b)所示。



(a)非空表　　(b)空表

图 2-8　带头结点的单链表

在线性表的顺序存储结构中,由于逻辑上相邻的两个元素在物理位置上也相邻,则每个元素的存储位置都可从线性表的起始位置计算得到。而在单链表中,逻辑上相邻的两个元素的存储位置之间没有固定的联系,但每个元素的存储位置都包含在其直接前驱结点的信息之中。假设 p 是指向线性表中第 i 个数据元素(结点 a_i)的指针,则 p->next 是指向第 i+1 个数据元素(结点 a_{i+1})的指针。也就是说,若 p->data=a_i,则 p->next->data=a_{i+1}。可见,在单链表中,要取得第 i 个数据元素必须从头指针出发,因此,单链表是非随机存取的存储结构。

2. 单链表的基本操作

下面将介绍使用单链表表示线性表时,如何实现线性表的几种基本操作。

1)单链表的建立操作

要对单链表进行查找、插入和删除等操作,首先要建立单链表。那么如何建立一个单链表?

单链表是一种动态结构,它不需要预先分配存储空间,生成链表的过程就是结点“逐个插入”的过程,即从“空表”的初始状态起,依次建立各元素结点,并逐个插入链表。根据结点

插入的位置不同,建立单链表可分为两种方式:逆序建立单链表和顺序建立单链表。

逆序建立单链表就是从表尾到表头逆向建立单链表,每次结点插入到头结点与第一个结点之间,如图 2-9 所示。具体过程如算法 2.6 所示。操作步骤如下:

(1)建立一个空的单链表(头结点)。

(2)输入数据元素,建立结点并插入到头结点之后。

(3)依次插入直至插入完毕。

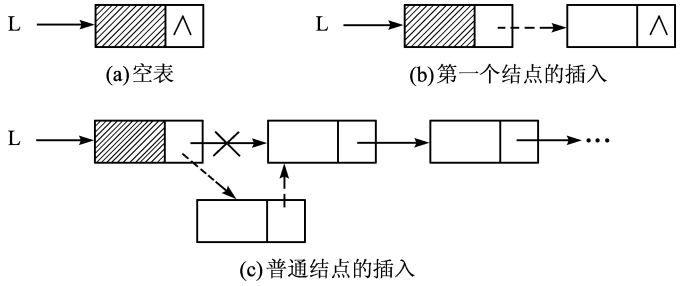


图 2-9 逆序建立单链表

```
void CreateList_L(LinkList &L,int n){
    // 逆位序输入 n 个元素的值,建立带头结点的单链表 L
    L=(LinkList)malloc(sizeof(LNode));              // 生成头结点
    L->next=NULL;
    for(i=n;i>0;--i){
        p=(LinkList)malloc(sizeof(LNode));          // 生成新结点
        scanf(&p->data);                            // 输入元素值
        p->next=L->next; L->next=p;                 // 插入到头结点之后
    }
}// CreateList_L
```

算法 2.6

顺序建立单链表是每次把新结点插入到链表尾部,因为单链表没有专门指示最后一个结点的指针,所以需要另设一个指针指向最后一个结点。算法 2.7 是一个顺序建立单链表的算法。

```
void CreateList_L(LinkList &L,int n){
    // 顺序建立单链表,尾插法
    L=(LinkList)malloc(sizeof(LNode));      // 生成头结点
    for(i=n;i>0;--i){
        p=(LinkList)malloc(sizeof(LNode));
        scanf(&p->data);                    // 输入元素值
        last->next=p;last=p;                // 使用 last 指针指向最后一个结点
    }
    last->next=NULL;
}// CreateList_L
```

算法 2.7

容易看出,算法 2.6 和算法 2.7 的时间复杂度均为 O(n)。

2)单链表的查找操作

查找操作就是找出单链表中的第 i(1≤i≤n)个结点,并返回该结点数据域的值。单链表是一种顺序存取结构,要找到第 i 个元素,则必须先找到第 i−1 个元素。因为单链表一般用头指针表示,所以必须从头结点开始搜索。

操作步骤:

(1)用指针 p 扫描单链表,用 j 作为计数器。初始化 p 指向第一个结点,j 为 1。

(2)当 p 扫描一个结点时,j 就相应地加 1,则 j 的值就是 p 已扫描过的结点数。

(3)当 j 的值刚好为 i 时,p 的值就是第 i 个结点的指针。

具体过程如算法 2.8 所示。

```
Status GetElem_L(LinkList &L,int i,ElemType &e){
    // L 为带头结点的单链表的头指针
    // 当第 i 个元素存在时,其值赋给 e 并返回 OK,否则返回 ERROR
    p=L->next;
    j=1;                    // 初始化 p 指向头结点,计数器 j 置 1
    while(p && j<i){        // 顺指针向后查找,直到 p 指向第 i 个元素或 p 为空
        p=p->next;++j;
    }
    if(!p || j>i)return ERROR;      // 当 i<0 或 i>n 时,表中无第 i 个结点
    e=p->data;                      // 取第 i 个元素
    return OK;
}// GetElem_L
```

算法 2.8

查找算法的基本操作是指针后移,其执行的频度与被查找元素在表中的位置有关,若 1≤i≤n,则频度为 i−1,否则为 n,因此算法 2.8 的时间复杂度为 O(n)。

单链表的长度是表中除头结点外的结点数目。求单链表的长度运算,可用类似于算法 2.8 的算法来实现,即用 p 扫描单链表的结点,j 计数 p 已扫描过的结点数,当 p 指向最后一个结点时,j 就计数到了表的长度。读者可自行完成求单链表长度的算法。

3)单链表的插入操作

插入操作是将新数据元素 e 插入到表中第 i 个元素之前,显然 i 的合法值是 1≤i≤n+1,n 为插入前表的长度。为插入数据元素 e,首先要生成一个数据域为 e 的结点,然后插入到单链表中。插入过程中,线性表的逻辑关系由 $<a_{i-1},a_i>$ 改变为 $<a_{i-1},e>$ 和 $<e,a_i>$,因此需要修改结点 a_{i-1} 中的指针域,令其指向结点 e,而结点 e 中的指针域应指向结点 a_i,从而实现 3 个元素之间逻辑关系的变化。插入过程如图 2-10 所示。

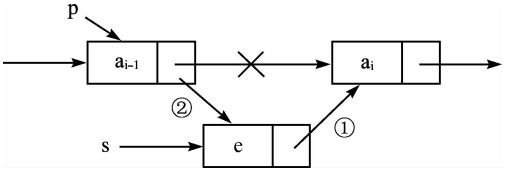


图 2-10 单链表中结点的插入过程

假设 s 为指向结点 e 的指针，则上述指针修改用语句描述为：

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

操作步骤：

(1)寻找第 i-1 个结点。

(2)判断 i 的取值是否合法，合法的取值应为：i∈[1,n+1]。

(3)给待插入元素分配存储空间。

(4)插入新结点。

具体过程如算法 2.9 所示。

```
Status ListInsert_L(LinkList &L,int i,ElemType e){
    // 在带头结点的单链表 L 的第 i 个元素之前插入元素 e
    p=L;j=0;                                  // 初始化
    while(p && j < i-1){                      // 寻找第 i-1 个结点
        p=p->next;
        ++j;
    }
    if(!p || j > i-1)return ERROR;            // 未找到,i<1 或 i>n+1
    s=(LinkList)malloc(sizeof(LNode));        // 生成新结点
    s->data=e;
    s->next=p->next; p->next=s;               // 插入新结点
    return OK;
}// ListInsert_L
```

算法 2.9

在单链表的某个结点之后插入一个新结点是常见的操作，应熟练掌握。另外，插入过程中的两个修改指针的操作是不可以颠倒的，因为如果先执行第二个操作，将无法定位第 i 个结点的位置。

算法 2.9 的时间主要消耗在查找第 i-1 个结点上，所以，算法的平均时间复杂度为 O(n)。若不考虑寻找结点的因素，仅就插入而言，其时间复杂度为 O(1)。

4)单链表的删除操作

删除操作是指删除单链表中第 i 个结点，i 的合法值是 1≤i≤n。注意，头结点是不能删除的。具体方法是：先找到第 i-1 个结点，若找到并且该结点的直接后继(即第 i 个结点)存在，则把第 i 个结点删除。要注意，删除一个结点只需要修改它的直接前驱的指针域，但因被删除结点已经无用途，所以要向系统申请释放被删除结点的空间。

如图 2-11 所示，在线性表中删除元素 a_i 时，为在单链表中实现元素 a_{i-1}、a_i 和 a_{i+1} 之间逻辑关系的变化，仅需修改结点 a_{i-1} 的指针域即可。

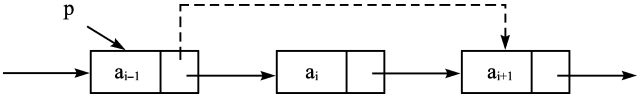


图 2-11 单链表中结点的删除过程

假设 p 为指向结点 a_{i-1} 的指针，则修改指针的语句为：

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

操作步骤:

(1)寻找第 i−1 个结点。

(2)判断 i 的取值是否合法,合法的取值应为:i∈[1,n]。

(3)修改第 i−1 个元素的指针来删除第 i 个元素。

(4)释放第 i 个元素所占的存储空间。

具体过程如算法 2.10 所示。

```
Status ListDelete_L(LinkList &L,int i,ElemType &e){
    // 在带头结点的单链表 L 中,删除第 i 个元素,并由 e 返回其值
    p=L;j=0;
    while(p->next && j < i-1){          // 寻找第 i 个结点,并令 p 指向其前驱
        p=p->next;
        ++j;
    }
    if(!(p->next) || j > i-1)return ERROR;  // 删除位置不合理
    q=p->next;
    p->next=q->next;                    // 删除结点
    e=q->data;
    free(q);                            // 释放结点空间
    return OK;
}// ListDelete_L
```

算法 2.10

整个删除算法的时间也消耗在查找第 i−1 个结点上,其时间复杂度为 O(n)。若不考虑查找所消耗的时间,仅就其删除过程而言一步就可实现,其时间复杂度为 O(1)。

由算法 2.9 和算法 2.10 可知,在单链表的第 i 个结点之前插入一个新结点或删除第 i 个结点,必须先找到第 i 个结点的直接前驱——第 i−1 个结点。只要找到了这个直接前驱,插入或删除的过程就非常简单,仅需修改指针即可,无须像顺序表那样移动元素。

【例 2-2】 编写一个算法,统计单链表中值为 x 的元素的个数,统计结果由函数值返回。

本题的算法思想:从表头依次访问链表中的每个结点,判断当前结点的数据域是否为 x,若是则结点个数加 1,结点个数存储在变量 count 中。具体过程如算法 2.11 所示。

```
int Count_x(LinkList L,ElemType x){
    p=L->next;                          // 指针 p 指向单链表的第一个结点
    count=0;                            // 计数器 count 清零
    while(p!=NULL){
        if(p->data==x)count++;          // 若当前数据域为 x,则计数器加 1
        p=p->next;                      // 指针 p 指向下一个结点
    }
    return count;
}// Count_x
```

算法 2.11

此算法和查找算法类似,其时间复杂度为 O(n)。

【例 2-3】 在非空链表 L 中，p 结点不是第一个结点，试编写删除 p 的直接前驱结点的算法。

由前面介绍过的单链表的删除算法可知，要删除某个结点必须先确定其前驱结点的位置。而本题要求删除的是 p 的前驱结点，若按以前方法则需要确定 p 的前驱结点的前驱结点，操作很不方便。可采取以下方法：查找 p 结点的直接前驱 q 结点，交换 p 结点和 q 结点的数据域，然后删除 p 结点。此时不需要重新定位，并且数据元素间的相对逻辑关系不变。具体过程如算法 2.12 所示。

```
void Delete_Prior(LinkList &L,LinkList p){
    q=L->next;
    while(q->next!=p) q=q->next;        // 移动指针 q 直到 q 的后继是 p
    p->data<-->q->data;                 // 交换 p 和 q 的数据域
    q->next=p->next;                    // 删除并释放结点 p
    free(p);
}
```

算法 2.12

【例 2-4】 设有两个非递减的单链表 La 和 Lb，现归并 La 和 Lb 得到 Lc，要求不能重新申请新的结点。

算法思想：利用单链表 La 和 Lb 非递减的特点，依次进行比较，将当前值较小的结点摘下，插入到链表 Lc 的表头。归并过程中需设立 3 个指针 pa、pb 和 pc，其中 pa 和 pb 分别指向 La 表和 Lb 表中当前待比较插入的结点，而 pc 指向 Lc 表中当前最后一个结点，若 pa->data≤pb->data，则将 pa 所指结点链接到 pc 所指结点之后，否则将 pb 所指结点链接到 pc 所指结点之后。显然指针的初始状态为：pa 和 pb 分别指向 La 和 Lb 表中第一个结点；pc 指向空表 Lc 中的头结点，因不能重新申请结点，使用 La 的头结点作为 Lc 的头结点。当其中一个链表归并完，只要将另一个表的剩余段链接在 pc 所指结点之后即可。由此得到归并两个单链表的算法，如算法 2.13 所示。

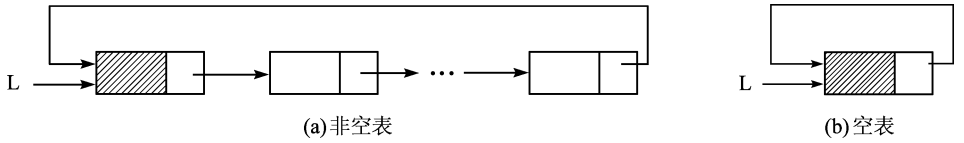
```
void MergeList_L(LinkList &La,LinkList &Lb,LinkList &Lc){
    // 归并非递减单链表 La 和 Lb 得 Lc
    pa=La->next; pb=Lb->next;
    Lc=pc=La;                           // 用 La 的头结点作为 Lc 的头结点
    while(pa && pb){
        if(pa->data<=pb->data){         // 插入值较小的结点
            pc->next=pa; pc=pa; pa=pa->next;
        }else {pc->next=pb; pc=pb; pb=pb->next;}
    }
    pc->next=pa ? pa : pb;              // 插入剩余段
    free(Lb);                           // 释放 Lb 的头结点
}// MergeList_L
```

算法 2.13

在算法 2.13 中，要依次查完单链表 La 和 Lb，因此该算法的时间复杂度为 O(m+n)，其中，m、n 分别是单链表 La 和 Lb 的长度。

2.3.2 循环链表及其基本操作

单链表中最后一个结点的指针域的值为空,如果将这个空的指针域的指针指向头结点,则整个链表就形成了一个环形结构,故称为循环链表。这样从表中的任意结点出发均可找到其他的结点。如图 2-12 所示为单链的循环链表,即单循环链表。类似地还有多重链的循环链表。



(a)非空表 (b)空表

图 2-12 单循环链表

循环链表和单链表的差别仅在于,判别链表中最后一个结点的条件不再是“后继是否为空”,而是“后继是否为头结点”。

【例 2-5】 将单循环链表倒置。

所谓倒置,就是把单循环链表$(a_1,a_2,\cdots,a_n)$变为$(a_n,\cdots,a_2,a_1)$,为此,需要逆转每个结点的指针域,即把原来指向其直接后继的指针,改为指向其直接前驱的指针。可用指针 p 扫描表的结点,并用指针 q、r 来指示 * p 的直接前驱与直接后继,每次均逆转 * p 的指针域。具体过程如算法 2.14 所示。

```
void InvertList(LinkList &L){
    q=L;                    // 初始化指针 q、p、r
    p=L->next;
    r=p->next;
    do{
        p->next=q;          // 使 p 的指针域指向其前驱
        q=p;                // q、p、r 在原逻辑关系上后移一位
        p=r;
        r=r->next;
    }while(q!=L);           // 当 q=L 时全部指针域均改完,结束
}
```

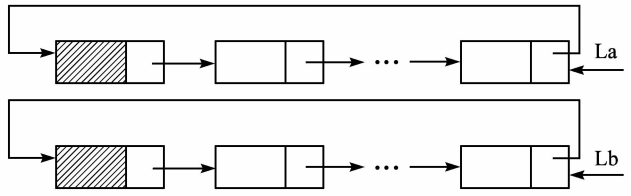
算法 2.14

此算法用一个指针 q 指向当前结点的前驱,否则每次改变指针域都要进行寻找前驱操作,在单循环链表中找前驱操作的时间复杂度为 O(n),节约了算法执行时间。算法 2.14 的时间复杂度为 O(n)。

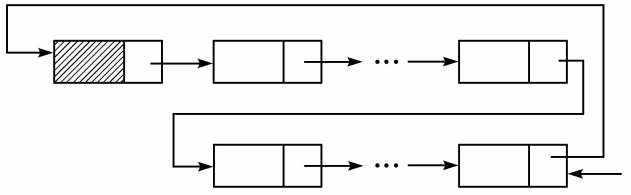
用头指针表示的单循环链表查找尾结点时和单链表一样麻烦,要从头结点搜索到尾结点,其时间复杂度也为 O(n)。许多问题中,对线性表的操作可能经常在表的首尾两端进行,此时用头指针表示的单循环链表就不够方便,可改用尾指针来表示单循环链表,这样可以简化某些操作。

【例 2-6】 有两个尾指针表示的单循环链表 La 和 Lb,编写一个算法将 Lb 链接到 La,链接后的链表仍保持单循环链表的形式。

算法思想:将两个单循环链表合并成一个单循环链表时,仅需将一个表的表尾和另一个表的表头相接。当单循环链表以尾指针表示的循环链表作为存储结构时,这个操作仅需改变两个指针值即可,如图 2-13 所示。具体过程如算法 2.15 所示。



(a)两个单循环链表



(b)合并后的单循环链表

图 2-13　尾指针指示的单循环链表的合并

```
LinkList MergeCircle_L(LinkList &La,LinkList &Lb){
    p=La->next;q=Lb->next;
    La->next=q->next;
    Lb->next=p;
    free(q);
    return Lb;
}
```

算法　2.15

此合并操作的时间复杂度为 O(1)。

2.3.3　双向链表及其基本操作

1.双向链表

单链表中,从已知结点出发,可以访问到该结点及其后继结点,却无法访问到该结点之前的其他结点。在单循环链表中,虽然从已知结点出发可访问到表中所有结点,但要找到其直接前驱却要遍历整个表,因为表中每个结点只有指向其直接后继的指针。

有些应用问题希望快速找到某个结点的前驱,这种情况下可在每个结点内再增加一个指向其直接前驱的指针域,这样形成的链表中有两条方向不同的链,故称为双向链表。在 C 语言中如下描述:

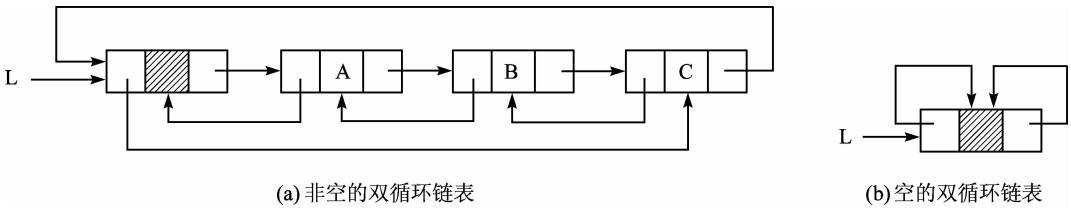
```
typedef struct DuLNode{
    ElemType data;                  // 数据域
    struct DuLNode *prior;          // 指向前驱的指针域
    struct DuLNode *next;           // 指向后继的指针域
}DuLNode, *DuLinkList;
```

与单链表相似,DuLNode 是一个双向链表的结构类型,它包含 3 个域:数据域 data、前驱指针域 prior 与后继指针域 next。DuLinkList 为双向链表中的指针变量。

双向链表也可以构成双循环链表,如图 2-14(a)所示,链表中存在两个环。如图 2-14(b)所示为只有表头结点的空双循环链表。在双向链表中,若 p 为指向表中某一结点的指针,则显然有:

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

这个表达式恰当地反映了这种结构的特性。



(a) 非空的双循环链表　　(b) 空的双循环链表

图 2-14　双向链表

2. 双向链表的基本操作

在双向链表中,有些操作,如求表长操作、取元素操作和定位操作等仅涉及一个方向的指针,则它们的算法描述和单链表的操作相同,但在插入、删除时有很大的不同,在双向链表中需同时修改两个方向上的指针。

1)双向链表的插入操作

在插入结点时,需要将插入点的前一个结点的 next 域指针和插入点的后一个结点的 prior 域指针及被插入点的两个指针作相应修改。在某一结点前插入一个新结点的处理过程如图 2-15 所示。

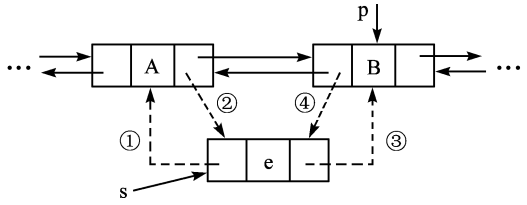


图 2-15　双向链表中插入结点时指针的变化

具体的操作步骤如下:

①s->prior=p->prior

②p->prior->next=s

③s->next=p

④p->prior=s

若要在第 i 个位置之前插入,则令指针 p 指向第 i 个元素,然后给待插入元素 e 分配新结点,令指针 s 指向待插结点,再按以上步骤进行插入。插入过程如算法 2.16 所示,其时间复杂度为 O(n)。

```
Status ListInsert_DuL(DuLinkList &L,int i,ElemType e){
    // 在带头结点的双循环链表 L 的第 i 个元素之前插入元素 e
    p=L->next;j=1;                    // 在 L 中确定第 i 个元素的位置指针 p
```

```
    while(p!=L && j<i){p=p->next;++j;}
    if(p==L && j<i)return ERROR;              // 第 i 个元素不存在
    s=(DuLinkList)malloc(sizeof(DuLNode));    // 分配空间存储待插元素
    s->data=e;
    s->prior=p->prior;                        // 插入元素时指针的变化
    p->prior->next=s;
    s->next=p;
    p->prior=s;
    return OK;
}// ListInsert_DuL
```

算法 2.16

2)双向链表的删除操作

在删除双向链表的某一个结点时,需要修改前一个结点的后继指针和后一个结点的前驱指针,如图 2-16 所示。

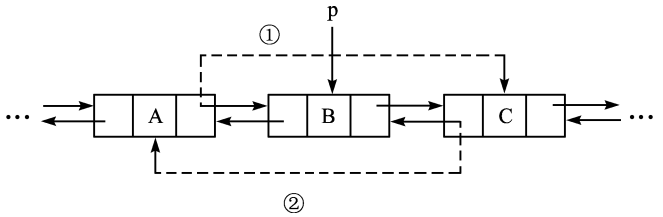


图 2-16 双向链表中删除结点时指针的变化

具体的操作步骤如下:

①p->prior->next=p->next

②p->next->prior=p->prior

同样若要删除双循环链表中的第 i 个结点,需要先找到第 i 个结点的位置。删除过程如算法 2.17 所示,时间复杂度也是 O(n)。

```
Status ListDelete_DuL(DuLinkList &L,int i,ElemType &e){
    // 删除带头结点的双循环链表 L 的第 i 个元素
    p=L->next;j=1;                    // 在 L 中确定第 i 个元素的位置指针 p
    while(p!=L && j<i){p=p->next;++j;}
    if(p==L && j<i)return ERROR;      // 第 i 个元素不存在
    e=p->data;                        // 返回删除结点的值
    p->prior->next=p->next;           // 删除
    p->next->prior=p->prior;
    free(p);
    return OK;
}// ListDelete_DuL
```

算法 2.17

【例 2-7】 试编写一个将双循环链表逆置的算法。

算法思想:

(1)设 L 指向双循环链表的头结点,p 的初值为 L->next。

(2)从原双循环链表的第一个结点开始向后扫描,依次修改每个结点的 next 和 prior 域指针,使之分别指向其结点的前驱和后继。在向后扫描的过程中,令 q 结点指向 p 结点的后继。

(3)修改头结点的指针,使之指向新的双循环链表的第一个及最后一个结点。

逆置过程如算法 2.18 所示。

```
Status Convert_DuL(DuLinkList &L){
    // 将双循环链表逆置
    p=L->next;                  // p 初始指向第一个结点
    while(p!=L){                // 顺链向后扫描到原双循环链表的最后一个结点
        q=p->next;              // q 指向 p 的后继结点
        p->next=p->prior;       // 修改 p 的 next 指针,使之指向其前驱
        p->prior=q;             // 修改 p 的 prior 指针,使之指向其后继
        p=q;                    // 顺链向后移动指针 p
    }
    q=L->next;
    L->next=p->prior; // 修改 L 的 next 指针,使之指向新双循环链表的第一个结点
    L->prior=q;  // 修改 L 的 prior 指针,使之指向新双循环链表的最后一个结点
    return OK;
}// Convert_DuL
```

算法 2.18

2.4 各种存储结构的线性表的比较

前面已经给出线性表的两种截然不同的实现方法,下面分析一下两种存储结构的优缺点。

顺序表是用一组地址连续的存储单元依次存放线性表的元素,元素的存储位置顺序与元素的逻辑顺序一致。这种存储方式决定了顺序表有如下优势:

(1)它是一种随机存取结构,存取速度快。存取表中第 i 个元素所需的时间与 i 无关,也与表长 n 无关,即时间复杂度为 O(1)。

(2)存储效率高,节省空间。线性表的存储空间的利用率可用存储密度衡量。存储密度是指结点的数据本身所占的存储量与整个结点存储量之比。显然存储密度≤1,而且存储密度越大,存储空间的利用率越高。显而易见,顺序表的存储密度为 1,也就是说顺序表的空间被完全用来存放元素本身而不附加任何信息。

但是,这种存储方式也造成了顺序表的缺点:插入或删除操作时要移动大量元素、效率低,时间复杂度为 O(n)。

链表用一组任意的存储单元存储线性表的元素,为了表示元素间的逻辑关系,还必须附加指针信息。因此,链表的优势是插入、删除方便,不需移动元素。缺点有:

(1)只适于顺序存取,存取速度慢。存取第i个结点,需从头结点开始沿着链扫描才能取得,时间复杂度是O(n)。因此,线性表的某些在顺序表上时间复杂度为O(1)的运算,如求表长、按序号查找等,在链表上实现却需花费时间,时间复杂度为O(n)。

(2)存储空间的利用率不高。链表的存储密度小于1。

总之,线性表的顺序存储结构与链式存储结构各有优缺点,不能笼统地说哪种实现更好,只能根据实际问题的具体需要,对各方面的优缺点加以综合平衡,才能最终选定适宜的实现方法。

2.5 实例解析与编程实现

【例2-8】 设线性表A采用顺序存储结构,且递增有序,试编写一个算法,将x插入到线性表的适当位置。

算法思想:先找到要插入的位置i,然后将i之后的元素均后移一位,将x插入到i的位置即可。具体过程如算法2.19所示。

```
void InsertOrder_Sq(Sq_List &A,ElemType x){
    i=0;
    while(x<A.elem[i])++i;          // 寻找x的插入位置
    for(j=A.length-1;j>=i;j--)      // 后移元素
        A.elem[j+1]=A.elem[j];
    A.elem[i]=x;                    // 插入x
    A.length++;                     // 表长加1
}// InsertOrder_Sq
```

算法 2.19

【例2-9】 编写用链式存储结构实现【例2-8】操作的算法。

算法思想:先找到待插入的位置,然后为待插入结点分配空间,最后按单链表的插入方法插入即可。为了便于操作,另设了一个指针q指向待插结点的前驱,如图2-17所示。具体过程如算法2.20所示。

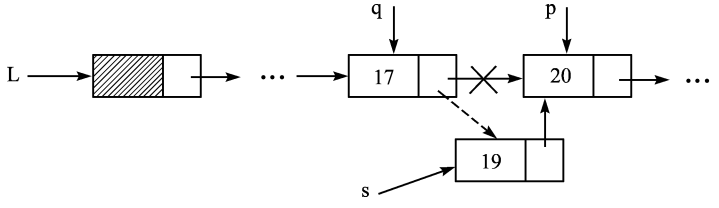


图2-17 顺序链表中结点的插入

```
void InsertOrder_L(LinkList &L,ElemType x){
    q=L;
    p=L->next;
    s=(LinkList)malloc(sizeof(LNode));      // 为x分配空间
    s->data=x;
```

```
    while(p->next!=NULL && p->data<x){  // 寻找 x 的插入位置,q 指向其前驱
        q=p;
        p=p->next;
    }
    if(P->next==NULL && P->date<x){   // 若最后一个结点的值仍然小于 x,
                                      则直接追加到链表末尾
    s->next=NULL;
    p->next=s;
    }
    else{
        s->next=q=next;               //插入新结点
        q->next=s;
    }
}// InsertOrder_L
```

算法 2.20

【例 2-10】 有两个一元多项式 Pa 和 Pb,采用链式存储结构,编写算法实现两个多项式相加:Pa=Pa+Pb,利用两个多项式的结点构成“和多项式”,且“和多项式”与 Pa 共用头结点。

一般情况下的一元多项式可写成:

$$P_n(x)=p_1x^{e_1}+p_2x^{e_2}+\cdots+p_mx^{e_m}$$

其中,p_i是指数为 e_i的项的非零系数,且满足:

$$0\leqslant e_1<e_2<\cdots<e_m=n$$

若用一个长度为 m 且每个元素有两个数据项(系数项和指数项)的集合来表示 $P_n(x)$,则可表示为:

$$((p_1,e_1),(p_2,e_2),\cdots,(p_m,e_m))$$

例如,一元多项式 $A_{17}(x)=7+3x+9x^8+5x^{17}$和一元多项式 $B_8(x)=8x+22x^7-9x^8$ 可用如图 2-18 所示的两个线性链表表示。其中每个结点表示多项式中的一项。

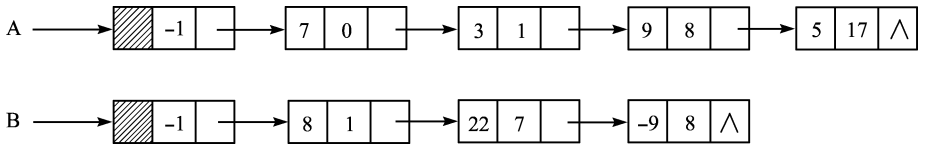


图 2-18 多项式的单链表存储结构

多项式的单链表存储结构和单链表只有数据元素上不同,可将多项式结点的数据元素类型 ElemType 定义为:

```
typedef struct{          // 项的表示
    float coef;          // 系数
    int expn;            // 指数
}ElemType;
```

一元多项式相加的运算规则:

对于两个一元多项式中所有指数相同的项，对应系数相加，如果其和不为零，则构成“和多项式”中的一项；对于两个一元多项式中所有指数不相同的项，则分别复制到“和多项式”中去。

在“和多项式”链表里的结点不需要另外生成，而是从两个多项式的链表中摘取。

两个一元多项式相加，用指针 qa、qb 分别扫描两个多项式，比较结点指数，有 3 种情况：

(1)qb 的指数值大于 qa 的指数值，取 qa 指针所指结点插入到“和多项式”。

(2)qb 的指数值小于 qa 的指数值，取 qb 指针所指结点插入到“和多项式”。

(3)p->expn=q->expn，将两结点系数相加：

- 和为 0，删除结点 qa、qb。
- 和不为 0，修改 qa 系数并插入，删除 qb。

多项式相加如算法 2.21 所示。

```
void AddPolyn(LinkList &Pa,LinkList &Pb){
    // 多项式加法:Pa=Pa+Pb,利用两个多项式的结点构成“和多项式”
    ha=Pa; hb=Pb;                    // ha 和 hb 分别指向 Pa 和 Pb 的头结点
    qa=ha->next;                     // qa 和 qb 分别指向 Pa 和 Pb 中当前结点
    qb=hb->next;
    while(qa && qb){                 // Pa 和 Pb 均非空
        a=qa->data;                  // a 和 b 为两表中当前比较元素
        b=qb->data;
        switch(Compare(a,b)){
        case -1:                     // 多项式 Pa 中当前结点的指数值小
            ha=qa;
            qa=qa->next;
            break;
        case 0:                      // 两者的指数值相等
            sum=a.coef+b.coef;
            if(sum!=0.0){            // 修改多项式 Pa 中当前结点的系数值
               temp.coef=sum;
               temp.expn=a.expn;
               qa->data=temp;
               ha=qa;
            }else{                   // 删除多项式 Pa 中 qa 所指的当前结点
               ha->next=qa->next;
               free(qa);
            }
            hb->next=qb->next; // 删除多项式 Pb 中 qb 所指的当前结点
            free(qb);
            qb=hb->next;             // qa 和 qb 指向未比较结点
            qa=ha->next;
```

```
            break;
        case 1:                          // 多项式 Pb 中当前结点的指数值小
            hb->next=qb->next; // 把 qb 所指结点插入到 hb 之后
            ha->next=qb;qb->next=qa;
            qb=hb->next;
            ha=ha->next;                 // ha 后移使之重新指向 qa 的前驱
            break;
        }// switch
    }// while
    if(qb)qa->next=qb;                   // 链接 Pb 中剩余结点
    free(hb);                            // 释放 Pb 的头结点
}// AddPolyn
```

算法 2.21

假设 Pa 多项式有 m 项,Pb 多项式有 n 项,则 AddPolyn 算法的时间复杂度为 O(m+n)。

本 章 小 结

线性表是具有 n 个数据元素的一个有限序列,元素的个数 n 为线性表的长度。线性表的特点是数据元素之间是一对一的关系。除第一个元素外,每个元素有且仅有一个直接前驱;除最后一个元素外,每个元素有且仅有一个直接后继。

采用顺序存储结构的线性表称为顺序表。顺序表的特点是逻辑位置相邻的数据元素其物理位置也相邻,因此可以进行随机存储,它是一种随机存储结构。顺序表中第 i 个数据元素的存储位置为 $Loc(a_i)=Loc(a_1)+(i-1)\times l$,l 为每个数据元素所占的存储空间。

在线性表的顺序存储结构中,主要掌握顺序表的插入和删除操作。等概率的情况下,顺序表中插入一个元素平均移动 n/2 个元素,删除一个元素平均移动(n-1)/2 个元素,所以插入和删除操作的时间复杂度均为 O(n)。

采用链式存储结构的线性表称为链表。一般提到链表即指单链表。链表中的数据元素使用一组任意的存储单元存储,不要求逻辑位置相邻的数据元素物理位置也相邻,而是采用附加的指针表示元素之间的逻辑关系。因此链表中的每个结点包含两个域:数据域存放的是元素的值,指针域存放的是该元素直接后继的地址。

链表采用头指针表示,为了简化链表的操作,通常在链表的表头增加一个头结点,即头指针指向头结点。在头结点中通常不存储其他信息。

在链表中插入和删除结点均不需要移动其他结点,但是,其查找运算必须从头指针开始顺序查找,其时间复杂度为 O(n)。

将单链表的最后一个结点的指针指向头结点时就得到单循环链表,在单循环链表中,从链表的每一个结点可以到达任何结点,但还是不可逆的。

双向链表的每个结点有两个指针,一个指向直接后继,一个指向直接前驱,它解决了单链表中不能向前查找的缺点。

习 题 2

1. 什么叫线性表？它有哪些特点？

2. 在链表的设计中，为什么通常采用带头结点的链表结构？

3. 对比顺序表与单链表，说明顺序表与单链表的主要优点和主要缺点。

4. 试编写算法实现顺序表的逆置，即把顺序表 A 中的数据元素$(a_1,a_2,\cdots,a_n)$逆置为$(a_n,a_{n-1},\cdots,a_1)$。

5. 已知 A 和 B 为两个非递减的线性表，现要求实现如下操作：从 A 中删除在 B 中出现的元素。试编写在顺序表中实现上述操作的算法。

6. 试编写算法实现链表的就地逆置(不增加存储空间)，即把链表 A 中的数据元素$(a_1,a_2,\cdots,a_n)$逆置为$(a_n,a_{n-1},\cdots,a_1)$。

7. 假设有两个非递减的线性表 A 和 B，均采用链式存储结构，试编写算法将 A 和 B 归并成一个按元素非递增顺序排列的线性表 C。

8. 假设有一个单循环链表，其结点含有 3 个域：data 域为数据域，next 域为后继指针域，prior 域为空。试编写算法将其改成双循环链表。

9. 试编写算法求单循环链表的表长。

第 3 章　栈 和 队 列

栈和队列是两种常用的线性结构，其逻辑结构与前面所介绍的线性表相同，不同的是线性表的插入和删除不受任何限制，而栈只能在一端进行插入和删除操作，队列只能在一端进行插入操作，另一端进行删除操作。因此，栈和队列都是操作受到限制的线性表。栈和队列在各种类型的软件中应用十分广泛，本章除介绍其基本概念和实现外，还将给出一些具体的应用实例。

3.1　栈

栈是一种重要的线性结构，有着广泛的应用。数据表达式的处理、函数及过程的调用等，都要用到栈的有关特性。下面将详细介绍栈的相关内容。

3.1.1　栈的基本概念

栈是限定仅能在表的一端进行插入和删除操作的线性表。能进行插入和删除的一端为栈顶(top)，另一端为栈底(bottom)。不含任何元素的栈称为空栈。

栈的插入操作称为进栈(压栈)，删除操作称为出栈(退栈)。进栈和出栈操作只能在栈顶进行。栈顶的位置在插入和删除中是动态变化的，通常设置一个变量指示栈顶的位置，称之为栈顶指针。而栈底的位置是保持不变的。

图 3-1 是栈的示意图。栈中已经按顺序存放了 a_1，a_2，…，a_n 共 n 个数据元素。其中 a_1 是最先进栈的元素，处于栈底；a_n 是最后进栈的元素，处于栈顶。如果要插入一个新的数据元素 a_{n+1}，即 a_{n+1} 进栈，则其插入位置只能是在 a_n 之上；如果要删除一个数据元素，则只能删除当前的栈顶元素，在图 3-1 中只能删除 a_n，即此时只有 a_n 可以出栈。

从图 3-1 中可以看出，最先进栈的元素 a_1 放在栈底，只能等它上面的元素全部出栈以后，a_1 才能出栈；而最后进栈的元素 a_n 放在栈顶，总是最先出栈。也就是说最后进栈的结点必须最先出栈，所以栈具有后进先出的特性，因此，栈又称为后进先出的线性表，简称 LIFO (last in first out)表。

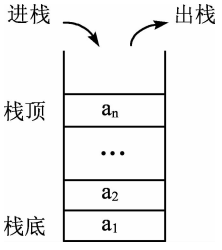


图 3-1　栈的示意图

日常生活中栈的例子很多，例如，在洗碗时总是将洗好的碗一个摞一个放置，而在使用时一般是从最上面依次取碗。如果把这摞碗看做一个栈，则栈顶就是最后洗好的那只碗，是第一个被使用的，最先出栈；而最先洗好的碗处于栈底，总是最后被使用的，最后一个出栈。

【例 3-1】 一个栈的进栈序列是 A、B、C，试给出全部可能的输出序列和不可能的输出序列。

栈的操作特点是后进先出，但不一定进栈顺序是 A、B、C，出栈顺序就一定是 C、B、A。因为可以调节输出的时机，如可以每次进栈一个元素就出栈，那么输出顺序就是 A、B、C。因此输出序列有：

- A 进，A 出，B 进，B 出，C 进，C 出，输出序列为 ABC。
- A 进，A 出，B 进，C 进，C 出，B 出，输出序列为 ACB。
- A 进，B 进，B 出，A 出，C 进，C 出，输出序列为 BAC。
- A 进，B 进，B 出，C 进，C 出，A 出，输出序列为 BCA。
- A 进，B 进，C 进，C 出，B 出，A 出，输出序列为 CBA。

由 A、B、C 组成的数据项，除上述 5 种不同组合外，尚有 CAB 组合。序列 CAB 是不可能的出栈序列。因为要想 C 先出栈，则 C 必是栈顶元素，即 A、B、C 需要全部进栈，C 出栈后，B 是栈顶元素，A 处于栈底，所以此时不可能再让 A 出栈。所以序列 CAB 不可能由输入序列 A、B、C 通过栈得到。

输出序列为 BAC 的操作过程如图 3-2 所示。

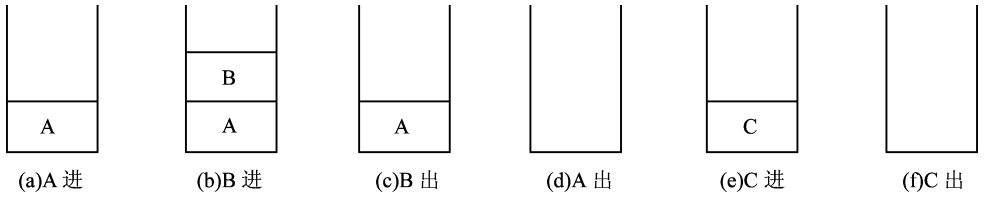


图 3-2　输出序列 BAC 的操作过程

在软件设计中，需要使用栈进行数据转换的例子很多。如数制转换、算术运算中的中缀表达式转换为后缀表达式等。

3.1.2　栈的顺序存储及顺序栈的操作

栈是一种特殊的线性表，因此栈和线性表一样具有两种存储结构，即顺序存储结构和链式存储结构。下面介绍栈的顺序存储结构及其操作。

1. 栈的顺序存储结构

栈的顺序存储结构称为顺序栈。顺序栈同顺序表一样可以使用一维数据表示，即利用一块地址连续的存储单元来存放栈中的元素。因为栈只能对栈顶元素进行插入和删除操作，所以设置了一个指示器来指示栈顶元素的位置。

栈的顺序存储结构如图 3-3 所示。其中，a_1、a_2、…、a_n 是栈所存储的数据元素序列，base 表示栈的连续存储空间的基地址，约定栈顶指针 top 指向栈顶元素的下一个位置，STACK_MAX_SIZE 表示栈可以存储的数据元素的最大个数。顺序栈描述如下：

```
#define STACK_MAX_SIZE 100          // 允许存放的最大元素个数
typedef struct{
```

```
    ElemType *base;                    // 栈的基地址
    ElemType *top;                     // 栈顶指针
}SqStack;
```

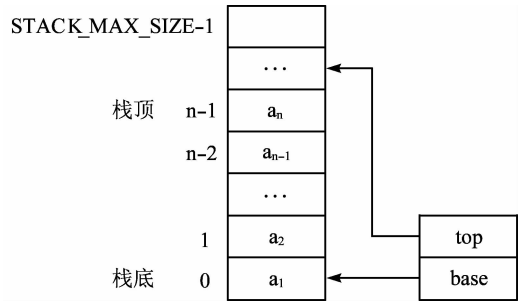


图 3-3 栈的顺序存储结构示意图

这里假设栈中数据元素的类型是用户自定义的 ElemType 类型。比较顺序栈和顺序表的定义可以发现,除了结构体名和数据域名不同外,顺序栈使用栈顶指针 top 取代了顺序表中的表长 length。这是因为出栈及进栈操作都发生在栈顶,可以便于出栈、进栈操作。

此外,规定了栈顶指针指向栈顶元素的下一个位置,这样做有两个好处:一是便于求得栈中元素的个数,例如,S 是 SqStack 类型的变量,栈中元素的个数就是 S. top－S. base;二是便于判断栈是否为空,此时栈空的标记是 S. top＝S. base。

图 3-4(a)表示一个空栈,图 3-4(b)表示有 3 个元素的栈,图 3-4(c)表示栈已满。插入新栈顶元素时指针 top 增 1;删除栈顶元素时指针 top 减 1。

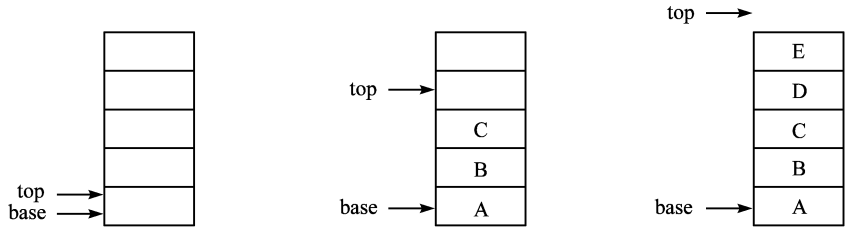


图 3-4 栈的状态及栈顶指针示意图

2. 顺序栈的基本操作

在对栈的顺序存储结构进行描述的基础上,栈的基本操作都可以用顺序栈来实现,下面是几个常用的顺序栈的基本操作。

1)初始化操作 InitStack(&S)

栈的初始化操作的目的是构建一个空栈,也就是说为栈 S 开辟一块存储空间,用 S. base 记录栈 S 的基地址,并且使栈顶指针 S. top 也指向栈的基地址,如果初始化成功则返回 OK,具体过程如算法 3.1 所示。

```
Status InitStack(SqStack &S){
    // 构造一个空栈 S
    S.base=(ElemType *)malloc(STACK_MAX_SIZE * sizeof(ElemType));
    S.top=S.base;          // 栈顶指针初始化
```

```
    return OK;
}// InitStack
```

算法　3.1

2)栈判空操作 StackEmpty(S)

栈判空操作是根据约定栈空的判定标志(S.top 是否等于 S.base)判断栈 S 是否为空栈,如果栈空则返回 TRUE,否则返回 FALSE,具体过程如算法 3.2 所示。

```
Status StackEmpty(SqStack S){
    // 判断栈 S 是否为空
    if(S.top==S.base)return TRUE;
    return FALSE;
}// StackEmpty
```

算法　3.2

3)求栈长操作 StackLength(S)

求栈长操作要求返回当前栈中数据元素的个数。在顺序栈的结构定义中并没有设置专门的数据域来存放栈中数据元素的个数。但顺序栈采用的是随机存储结构,并且已知栈的基地址和栈顶指针的值,它们相减就可以得到栈中元素所占的存储空间,并由指针的性质可知 S.top－S.base 就是栈中数据元素的个数,即栈的长度。具体过程如算法 3.3 所示。

```
int StackLength(SqStack S){
    // 求栈的长度
    return S.top－S.base;
}// StackLength
```

算法　3.3

4)取栈顶元素操作 GetTop(&S,&e)

取栈顶元素操作是用一个引用参数 e 返回栈顶元素的值,如果操作成功返回 OK,否则返回 ERROR。取栈顶元素操作与出栈操作不同的是,取栈顶元素操作并不删除栈顶元素。在定义中约定了栈顶指针指向栈顶元素的下一个位置,因此要取得的元素地址为栈顶指针所指位置的上一个位置。在取元素之前要判断栈是否为空,为空时不能操作。取栈顶元素操作如算法 3.4 所示。

```
Status GetTop(SqStack &S,ElemType &e){
    // 栈 S 不空,用 e 返回栈顶元素
    if(S.top==S.base)return ERROR;      // 栈空
    e= *(S.top－1);                     // 栈不为空,返回栈顶元素
    return OK;
}// GetTop
```

算法　3.4

5)进栈操作 Push(&S,e)

进栈操作就是在栈顶插入一个新数据元素,如果操作成功返回 OK,否则返回 ERROR。因为栈顶指针指向栈顶元素的下一个位置,只需将新元素存储到栈顶指针所指位置即可,然后再把栈顶指针加 1。对于顺序栈,进栈时首先判断栈是否已满,栈满的判定条件为:S.top－S.base==STACK_MAX_SIZE。栈满时,不能进栈,否则会出现空间溢出,引起错

误,这种现象称为上溢。进栈操作如图 3-5 所示,具体过程如算法 3.5 所示。

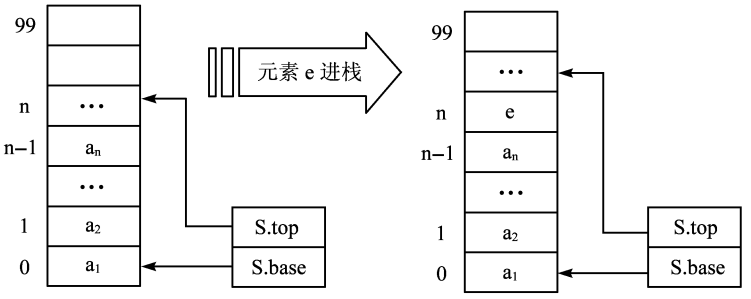


图 3-5 进栈示意图

```
Status Push(SqStack &S,ElemType e){
    if(S.top-S.base==STACK_MAX_SIZE)      // 栈满,无法进栈
       return ERROR;
    *S.top=e;                             // 插入新元素
    S.top++;                              // 栈顶指针指向下一个位置
    return OK;
}// Push
```

算法 3.5

6)出栈操作 Pop(&S,&e)

出栈操作就是弹出栈顶元素,并由引用参数 e 回传其值。如果操作成功返回 OK,否则返回 ERROR。与进栈操作相反,出栈操作要先移动指针再取值。出栈操作时先判断栈是否为空。出栈操作如图 3-6 所示,具体过程如算法 3.6 所示。

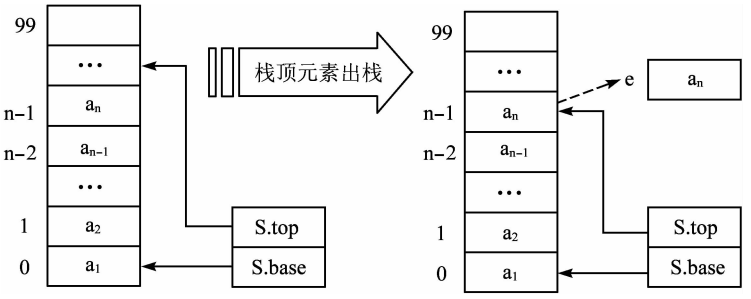


图 3-6 出栈示意图

```
Status Pop(SqStack &S,ElemType &e){
    // 若栈不空,则删除 S 的栈顶元素,用 e 返回其值
    if(S.top==S.base)return ERROR;
    --S.top;                    // 栈顶指针减 1
    e=*S.top;                   // 取得栈顶元素的值
    return OK;
}// Pop
```

算法 3.6

观察算法 3.6 可以发现,算法本身并没有具体进行数据元素的删除操作,那么是如何进

行数据元素的删除操作的呢？这是因为约定了 top 指向栈顶元素的下一个位置，虽然没从物理位置上直接删除，但在逻辑结构上栈中已经没有这个数据元素了。

3.1.3　栈的链式存储及链栈的操作和应用

栈同样也可以采用链式存储结构，栈的链式存储形式简称为链栈。链栈是一个特殊的单链表，但其运算受限制，插入和删除只能在链栈的一端进行。

1. 栈的链式存储结构

栈有两端，插入元素和删除元素的一端称为栈顶，另一端称为栈底。对于链栈来说，把靠近头指针的一端定义为栈顶时，插入元素和删除元素时不需要访问栈中所有的结点，其时间复杂度为 O(1)；否则，如果链表尾的一端定义为栈顶，则每次插入或删除一个元素都要遍历整个链，其时间复杂度为 O(n)。因此，在设计链栈时都把靠近头指针的一端定义为栈顶。链栈的头结点对操作的实现影响不大，因此可有可无，本书为了与前面单链表的形式保持一致，采用了带头结点的链栈。采用头插法依次向带头结点的链栈插入 a_1，a_2，…，a_n 后，链栈如图 3-7 所示。

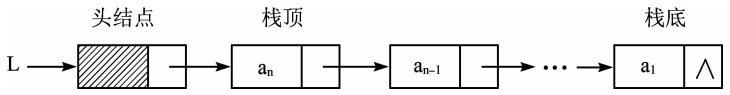


图 3-7　栈的链式存储结构

链栈中的每个结点 SNode 由两个域组成：一个是存放数据元素值的数据域 data，一个是存放该结点直接后继结点地址的指针域 next。链栈结点的结构体定义如下：

```
typedef struct SNode{
    ElemType data;
    struct SNode * next;
}SNode;
```

2. 链栈的基本操作

因为链栈中的结点是动态生成的，一般不会出现上溢，因而可以不考虑栈满的情况。链栈在设计上采用了带头结点的链式存储结构，则进栈和出栈操作改变的是链表头指针所指头结点的 next 域的值，因此可以把头结点的 next 域看做是栈顶指针。其具体操作的算法描述如下。

1)初始化操作 InitStack_L()

构造一个空的链栈。其操作是生成一个头结点，其 next 域置为空，即栈顶指针置空，最后以函数名的形式返回头指针的值，初始化链栈操作如算法 3.7 所示。

```
SNode * InitStack_L(){
    // 构造一个空栈 S
    S=(SNode *)malloc(sizeof(SNode));
    S->next=NULL;                    // 置栈顶指针为空
    return S;                        // 返回头指针的值
}// InitStack_L
```

算法　3.7

2)栈判空操作 StackEmpty(S)

栈判空操作是判断栈 S 是否为空栈,如果栈空则返回 TRUE,否则返回 FALSE。当栈中没有元素时,头指针的 next 域为空,即栈顶指针为空时,链栈为空。而在顺序栈中,栈是否为空是判定栈的栈顶指针是否指向栈的基地址,栈判空操作如算法 3.8 所示。

```
Status StackEmpty_L(SNode * S){
    // 判断栈 S 是否为空
    if(S->next==NULL)return TRUE;      // 栈顶指针为空,返回 TRUE
    return FALSE;
}// StackEmpty_L
```

算法 3.8

3)取栈顶元素操作 GetTop(S,&e)

取栈顶元素操作是用一个引用参数 e 返回栈顶元素的值,如果操作成功返回 OK,否则返回 ERROR。在取元素之前要判断栈是否为空,为空时不能进行取元素操作。如果栈不为空,取链栈中第一个元素的值,即头指针 next 域所指结点的值。取栈顶元素操作如算法 3.9 所示。

```
Status GetTop_L(SNode * S,ElemType &e){
    // 栈 S 不空,用 e 返回栈顶元素
    p=S->next;                         // 使指针 p 指向栈顶元素
    if(p==NULL)return ERROR;           // 栈为空,操作失败
    e=p->data;                         // 用引用参数返回栈顶元素的值
    return OK;
}// GetTop_L
```

算法 3.9

4)进栈操作 Push(&S,e)

进栈操作就是在栈顶插入一个新数据元素,如果操作成功返回 OK,否则返回 ERROR。链栈的进栈操作与单链表的插入操作基本相同,区别在于进栈操作是把新元素插入到头结点之后,因此不需要定位操作。进栈操作如图 3-8 所示,具体过程如算法 3.10 所示。

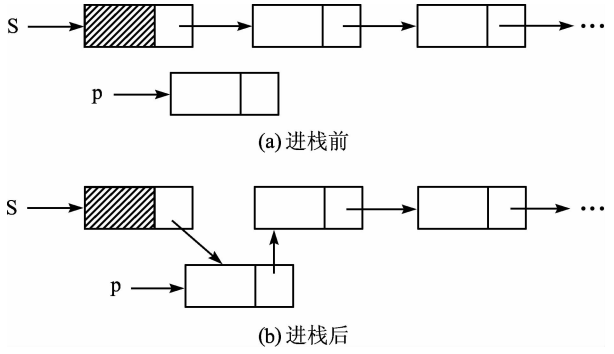


图 3-8 链栈的进栈操作

```
Status Push_L(SNode * S,ElemType e){
    p=(SNode * )malloc(sizeof(SNode));          // 分配新结点
```

```
    p->data=e;
    p->next=S->next;                    // 新结点入栈
    S->next=p;
    return OK;
}// Push_L
```

算法　3.10

5)出栈操作 Pop(&S,&e)

出栈操作就是弹出栈顶元素,并由引用参数 e 回传其值。如果操作成功返回 OK,否则返回 ERROR。出栈操作与取栈顶元素操作类似,区别在于出栈操作取值后,从栈中删除当前栈顶元素,具体过程如算法 3.11 所示。

```
Status Pop_L(SNode *S,ElemType &e){
    // 若栈不空,则删除 S 的栈顶元素,用 e 返回其值
    p=S->next;                          // 使 p 指向栈顶元素
    if(p==NULL)return ERROR;
    S->next=p->next;                    // 删除栈顶元素
    e=p->data;                          // 用引用参数返回栈顶元素的值
    free(p);                            // 释放栈顶元素所占的空间
    return OK;
}// Pop_L
```

算法　3.11

3.链栈的应用

链栈同顺序栈一样具有后进先出的特性,因此可以用顺序栈解决的问题也都可以用链栈来解决。链栈不需要预先设定存储空间的大小,克服了顺序栈的缺点,但链栈的存储密度较小。一般在解决事先不能确定规模的问题时可使用链栈。

【例 3-2】 假设一个算术表达式中包含圆括号和方括号两种类型的括号,编写一个判别表达式中括号是否正确配对的函数。

算法思想:算术表达式中右括号和左括号匹配的次序正好符合后到的括号要最先被匹配的后进先出的特点,因此,可以借助一个栈来进行判断。

括号匹配共有 4 种情况:

(1)左右括号配对次序不正确,如[( ]]或(()]。

(2)左括号多于右括号,如(([ ])。

(3)右括号多于左括号,如([ ])]。

(4)左右括号匹配正确,如([ ]())。

具体方法:顺序扫描算术表达式(表现为一个字符串),当遇到两种类型的左括号时让括号进栈;当扫描到某一种类型的右括号时,比较当前栈顶括号是否与之匹配,若匹配,则出栈继续进行判断;若当前栈顶括号与当前扫描的括号不相同,则左右括号配对次序不正确;若字符串当前为某种类型右括号而栈已空,则右括号多于左括号;字符串循环扫描结束时,若栈非空(即栈中尚有某种类型左括号),则说明左括号多于右括号;否则,左右括号匹配正确。若匹配正确则函数返回 OK;匹配不正确时均返回 FALSE。具体过程如算法 3.12所示。

```
Status Check(char exp[]){
    S=InitStack_L();i=0;            // 初始化栈
    while(exp[i]!='\0'){            // 当表达式不空时,进行判断
        switch(exp[i]){
        case '(':                   // 若当前为'(',则进栈,再判断下一字符
            Push_L(S,exp[i]);i++;break;
        case '[':                   // 若当前为'[',也进栈
            Push_L(S,exp[i]);i++;break;
        case ')':                   // 若当前为')',取栈顶元素判断是否匹配
            if(!StackEmpty_L(S)){
               GetTop_L(S,e);
               if(e=='('){          // 栈顶为'(',则匹配成功,再判断下一字符
                   Pop_L(S,e);i++;
               }else if(e=='[')     // 栈顶为'[',则匹配次序不正确,返回 FALSE
                   return FALSE;
    }else return FALSE;             // 若栈为空,则右圆括号多于左圆括号
    break;
    case ']':                       // 若当前为']',取栈顶元素判断是否匹配
        if(!StackEmpty_L(S)){
           GetTop_L(S,e);
           if(e=='['){              // 栈顶为'[',则匹配成功,再判断下一字符
              Pop_L(S,e);i++;
           }else if(e=='(')         // 栈顶为'(',则匹配次序不正确,返回 FALSE
               return FALSE;
        }else return FALSE;         // 若栈为空,则右中括号多于左中括号
        break;
        default: i++;
        }
    }
    if(StackEmpty_L(S))
        return OK;
    else
        return FALSE;
}
```

算法 3.12

3.1.4 栈的简单应用与递归

递归是程序设计中一种有用的算法,有助于提高编程的效率,增强算法的可读性。本书中的许多数据结构,如广义表、树和二叉树等,都是通过递归方式定义的。递归在计算机内部是通过栈来实现的,是栈的一种实际应用,下面将对递归作详细介绍。

1. 递归的相关概念

一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称为递归函数。图3-9(a)中函数A在执行过程中调用了自己，称为直接递归；图3-9(b)中函数B通过函数C又调用了自己，称为间接递归。

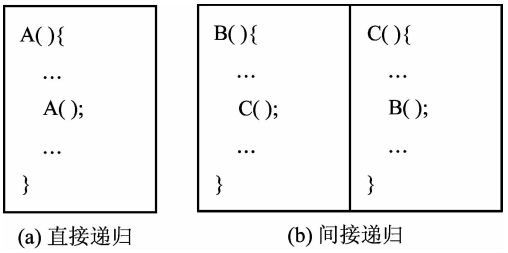


图3-9　递归示意图

许多数学函数也是递归定义的。如求整数n的阶乘函数的递归定义如下：

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

可见，如果要求n!，先要求出(n−1)!，而要求(n−1)!，必须先求出(n−2)!，以此类推，直到n=0时，可以直接求得0!=1，此时的情况称为递归的终止条件。然后再返回上层求得1!=1，再次返回求得2!=2，以此类推，最终求得n!。如果用Fac(n)表示求n的阶乘的函数，根据上面的定义很容易写出求n的阶乘的递归函数，如算法3.13所示。

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

算法　3.13

如果用main主函数来调用Fac(n)，并令初始值n=3，则这一函数的递归调用如图3-10所示。

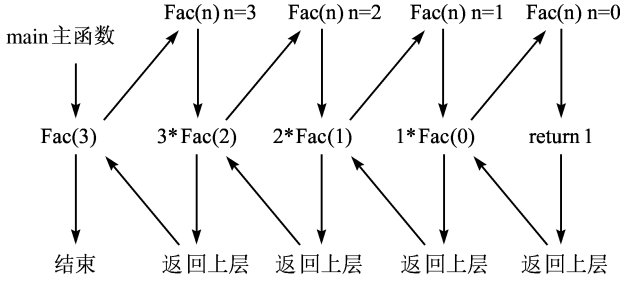


图3-10　Fac(3)的递归调用过程

2. 递归程序的编写

在实际应用中，许多问题本身或其所涉及到的数据结构是递归定义的，有些问题可以通过分析、抽象得到其递归定义，对于这些问题就可以使用递归函数求解。在编写递归函数时，要遵循以下几点规则：

(1)明确程序所要完成的功能,确定程序中所涉及的变量,特别是递归过程中的主变量。

(2)为了保证递归的正常终止,必须有一个能直接获得的最小子问题解,即确定递归的终止条件,并写出对应的操作。

(3)假设目前所定义的函数功能正确,考虑在非递归终止条件下,通过改变参数的值递归调用函数本身,完成剩余的工作。

因此,对于一个可递归解决的问题可描述如下(递归定义包括两项):

基本项(终止项):描述递归终止时问题的求解。

递归项:将问题分解为与原问题性质相同,但规模较小的问题。

例如,阶乘函数 n! =1×2×3×4×…×(n−1)×n 就可以描述为:

基本项:n! =1　　当 n=1 时

递归项:n! =n×(n−1)!　　当 n>1 时

【例 3-3】 试编写一个递归函数,求两个正整数 m 和 n 的最大公约数,其中最大公约数的 Gcd(m,n)的求解公式为:

$$Gcd(m,n)=\begin{cases} m & n=0 \\ Gcd(n,m\%n) & \text{其他情况}(n<m) \end{cases}$$

以上求最大公约数 Gcd 的定义本身就是递归定义,已经给出了基本项与递归项,因此采用递归方法求 m 和 n 的最大公约数十分方便。当 n=0 时递归终止,其他情况下只需按公式递归调用即可。具体实现过程如算法 3.14 所示。

```
int Gcd(int m,int n){
    // 求 m 和 n 的最大公约数
    if(n==0)return m;          // 基本项
    else return Gcd(n,m%n);    // 递归项
}
```

算法　3.14

3.2 队　　列

队列也是一种特殊的线性表,队列的数据元素及数据元素之间的逻辑关系与线性表完全相同,其差别是线性表允许在任意位置插入和删除元素,而队列只允许在其中的一端进行插入操作,另一端进行删除操作。

3.2.1 队列的基本概念

队列是限定仅能在一端进行删除,另一端进行插入的线性表。允许删除的一端称为队头,允许插入的一端称为队尾。当队列中没有数据元素时称为空队列。

图 3-11 是队列的示意图。队列中已经按顺序存放了 a_1,a_2,…,a_n 共 n 个数据元素。其中,a_1 是最先插入的元素,处于队头;a_n 是最后插入的元素,处于队尾。如果要插入一个新的数据元素 a_{n+1},则只能在队尾 a_n 之后进行插入,插入的过程也称为入队。如果要删除一个数据元素,则只能删除当前处于队头的数据元素,在图 3-11 中只能删除 a_1,在队列中删除数据元素的过程也称为出队。

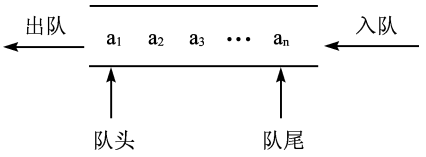


图 3-11　队列的示意图

前面介绍的栈是在栈的一端进行插入和删除操作，栈底不动，栈顶元素的位置随插入和删除不断变化。而队列的插入和删除操作是在队列的两端进行的，在插入的过程中队尾的位置发生变化，在删除的过程中队头发生变化。最先入队的数据元素 a_1 放在队头，总是最先出队；而最后入队的数据元素 a_n 放在队尾，必须等它前面的元素全部出队以后，a_n 才能出队。由此可见，队列中数据元素的入队和出队过程是按照“先进先出”的原则进行的，因此队列又称为“先进先出”的线性表，简称 FIFO(first in first out)表。

日常生活中也有很多队列的例子。例如，到医院看病，首先需要到挂号处挂号，然后按号码顺序就诊；乘坐公共汽车，应该在车站排队，车来后按顺序上车。

3.2.2　队列的顺序存储、操作及应用

1. 队列的顺序存储结构

队列也是一种特殊的线性表，线性表的存储结构也适用于队列的存储，因此队列也有两种存储结构：顺序存储结构和链式存储结构。

按顺序存储结构存储的队列称为顺序队列，同顺序表一样，使用一块连续的存储单元来存放队列中的元素。因为队列的各种操作要在队列的队头和队尾进行，因此要设置两个指示器分别指示队头和队尾的位置，称之为队头指针和队尾指针。

图 3-12 是一个具有 6 个存储空间的顺序队列的动态示意图。图中 front 为队头指针，rear 为队尾指针。为了方便，约定队头指针指向队头元素，队尾指针指向队尾元素的下一个位置。因此，当队列为空时，front＝rear＝0，每当有新元素入队时，插入到队尾，队尾指针 rear 加 1；每当队头元素出队时，队头指针加 1。

图 3-12(a)表示一个空队列；图 3-12(b)表示数据元素 A、B、C 入队后的状态；图 3-12(c)表示数据元素 A、B、C 出队，D、E、F 入队后的状态；图 3-12(d)表示数据元素 D、E、F 出队后的状态。

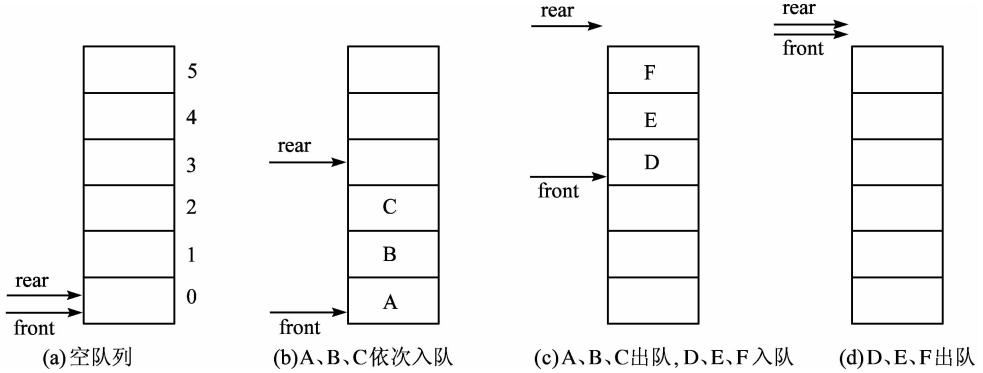


图 3-12　队列的顺序存储结构

假设当前为队列分配的最大存储空间为 6,则当队列处于图 3-12(c)和 3-12(d)的状态时,不可再继续插入新的队尾元素。因为,若此时插入元素,顺序队列将因队尾指针越出数组下界而“溢出”。

此时的“溢出”是因为队尾指针 rear 的值超出了顺序队列定义存储空间的范围而引起的,但此时队列中还有若干个存储空间可供存储,因此,这时的“溢出”并不是由于存储空间不够而产生的溢出。假设顺序队列定义的存储空间大小为 QUEUE_MAX_SIZE,则:

当 front=0,rear=QUEUE_MAX_SIZE 时,再有元素入队发生溢出,称之为“真溢出”,此时存储空间已满。

当 front>0,rear=QUEUE_MAX_SIZE 时,再有元素入队发生溢出,称之为“假溢出”,存储空间还有剩余。

2. 顺序存储的循环队列

由于顺序队列存在假溢出问题,所以顺序队列很少在实际的软件系统中使用。

为了改进这种状况,可以将顺序队列想象为一个首尾相接的环状空间,如图 3-13 所示,称之为循环队列。

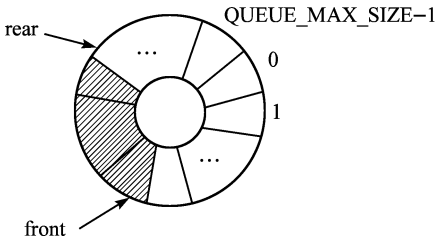


图 3-13 循环队列示意图

循环队列也是顺序存储的队列,它的基本特点与普通的顺序队列相似,区别在于队尾指针 rear 和队头指针 front 可以越过 QUEUE_MAX_SIZE-1 所指单元进行下一轮循环。这样,就不会出现“假溢出”的问题。

循环队列队头指针和队尾指针逻辑循环的方法,可以利用高级程序设计语言中的取模(或称求余)运算%来实现:

入队:rear=(rear+1)% QUEUE_MAX_SIZE;

出队:front=(front+1)% QUEUE_MAX_SIZE;

例如,设 QUEUE_MAX_SIZE=6,当队尾指针 rear=5 时,若此时作入队操作,即 rear 加 1 则有 rear=(rear+1)% 6=0,即实现了队尾指针 rear 的下一个取值为 0。

设循环队列的存储空间大小 QUEUE_MAX_SIZE=6,图 3-14(a)是其一般情况下的状态,循环队列有 3 个数据元素 A、B、C,队头指针 front=3,队尾指针 rear=0。当数据元素 D、E、F 也入队后,循环队列已满,此时队头指针 front=3,队尾指针 rear=3,即 front==rear,其状态如图 3-14(b)所示。当数据元素 A、B、C、D、E、F 依次出队后,循环队列空,此时队头指针 front=3,队尾指针 rear=3,即 front==rear,其状态如图 3-14(c)所示。由此可见,在循环队列中,队列满时的状态为 front==rear,队列空时的状态也为 front==rear,这将导致算法设计出现无法区分队空状态和队满状态的问题。

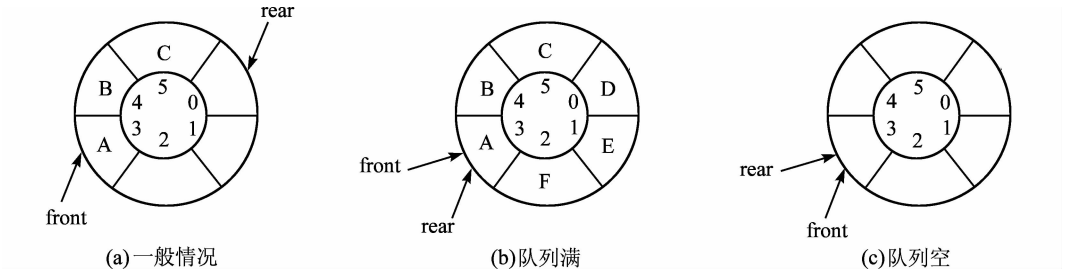


图 3-14　循环队列的几种状态表示

解决循环队列的队列满和队列空状态的判断问题通常有 3 种方法：

(1)少用一个存储单元。当少用一个存储单元时，以队尾指针 rear 加 1 再对 QUEUE_MAX_SIZE 取模等于队头指针 front 为队列满的判断条件。

队列空的判断条件为：

rear==front

队列满的判断条件为：

(rear+1)%QUEUE_MAX_SIZE==front

(2)设置一个标志位。设置一个标志位 tag，初始时置 tag=0；每当入队列操作成功就置 tag=1；每当出队列操作成功就置 tag=0。

队列空的判断条件为：

rear==front && tag==0

队列满的判断条件为：

rear==front && tag==1

(3)设置一个计数器。设置一个计数器 count，初始时置 count=0；每当入队列操作成功就使 count 加 1；每当出队列操作成功就使 count 减 1。这样，该计数器不仅具有计数功能，而且还具有像标志位一样的标志作用。

队列空的判断条件为：

count==0

队列满的判断条件为：

count > 0 && rear==front

设置一个标志位和设置一个计数器都需要改变原来顺序队列的结构，因此在下边的循环队列的实现中采用少用一个存储单元的方法来判断队空状态和队满状态。

循环队列的结构体定义描述如下：

```
#define QUEUE_MAX_SIZE 100              // 最大队列长度
typedef struct{
    ElemType *base;                     // 动态分配存储空间
    int front;                          // 队头指针
    int rear;                           // 队尾指针
}SqQueue;
```

这里假设队列中数据元素的类型是用户自定义的 ElemType 类型。为了便于在队头队尾的操作，在结构体中设置了两个指针：队头指针 front 和队尾指针 rear。注意这两个指针

不是普通意义上的指针类型,而是整型变量,相当于可来回移动的游标。

3. 循环队列的基本操作

循环队列具体操作的算法描述如下。

1)初始化操作 InitQueue(&Q)

首先为循环队列 Q 分配一块存储空间,并用 Q. base 记录这块空间的地址;然后使队头指针和队尾指针都为 0,相当于指向第一个存储空间。如果初始化成功则返回 OK,具体实现过程如算法 3.15 所示。

```
Status InitQueue(SqQueue &Q){
   // 构造一个空队列 Q
   Q.base=(ElemType *)malloc(QUEUE_MAX_SIZE * sizeof(ElemType));
   Q.front=Q.rear=0;              // 置队头指针和队尾指针为 0
   return OK;
}
```

算法 3.15

2)判空操作 QueueEmpty(Q)

对于少用一个存储单元的方法表示的循环队列,其队列空的标志是判断队尾指针 rear 是否等于队头指针 front,如果等于则返回 TRUE,否则返回 FALSE,具体实现过程如算法 3.16 所示。

```
Status QueueEmpty(SqQueue Q){
    // 判断队列 Q 是否为空
    if(Q.front==Q.rear) return TRUE;  // 队头指针等于队尾指针,返回 TRUE
    return FALSE;
}
```

算法 3.16

3)求队列长度操作 QueueLength(Q)

求循环队列长度的操作要求返回当前循环队列中数据元素的个数。根据约定队头指针指向队头元素,队尾指针指向队尾元素的下一个位置。队尾指针减去队头指针的值就是当前队列中元素的个数,因为循环队列中队尾指针可能小于队头指针,所以要加上队列存储空间大小 QUEUE_MAX_SIZE 后再作取模运算,具体实现过程如算法 3.17 所示。

```
int QueueLength(SqQueue Q){
    // 返回队列 Q 中元素的个数,即队列的长度
    return(Q.rear-Q.front+QUEUE_MAX_SIZE) % QUEUE_MAX_SIZE;
}
```

算法 3.17

4)取队头元素操作 GetHead(Q,&e)

取队头元素操作是用一个引用参数 e 返回队头元素的值,如果操作成功返回 OK,否则返回 ERROR。取队头元素操作与出队操作不同的是,取队头元素操作并不删除队头元素。在定义中约定了队头指针指向队头元素,而实际上队头指针就是队头元素在一维数组中的下标,因此 Q. base[Q. front]就是队头元素。在取队头元素之前要判断队列是否为空,为空时不能取元素。具体实现过程如算法 3.18 所示。

```
Status GetHead(SqQueue Q,ElemType &e){
    // 取循环队列 Q 中当前队头元素并赋给 e
    if(Q.front==Q.rear) return ERROR;
    e=Q.base[Q.front];
    return OK;
}
```

算法　3.18

5)入队操作 EnQueue(&Q,e)

入队操作就是在队尾插入一个新数据元素,如果操作成功返回 OK,否则返回 ERROR。因为队尾指针指向队尾元素的下一个位置,因此只需将新元素存储到队尾指针所指示的位置,然后再把队尾指针加 1 即可。入队前首先判定队列是否已满,队满的判定条件为:(Q.rear+1) % QUEUE_MAX_SIZE==Q.front(少用一个存储单元),队满时,不能作入队操作。具体实现过程如算法 3.19 所示。

```
Status EnQueue(SqQueue &Q,ElemType e){
    // 插入元素 e 为 Q 的新的队尾元素
    if((Q.rear+1)%QUEUE_MAX_SIZE==Q.front)
        return ERROR;                          // 队列满
    Q.base[Q.rear]=e;                          // 插入元素到队尾
    Q.rear=(Q.rear+1)%QUEUE_MAX_SIZE;          // 队尾指针加 1
    return OK;
}
```

算法　3.19

6)出队操作 DeQueue(&Q,&e)

出队操作就是删除队头元素,并由引用参数 e 回传其值。如果操作成功返回 OK,否则返回 ERROR。因为队头指针指示了队头元素,所以可以直接取得其值 Q.base[Q.front],然后再移动队头指针,使其加 1。出队操作也要先判断队列是否为空。如何实现过程如算法 3.20 所示。

```
Status DeQueue(SqQueue &Q,ElemType &e){
    // 若队列不空,则删除 Q 的队头元素,用 e 返回其值,并返回 OK; 否则返回 ERROR
    if(Q.front==Q.rear) return ERROR;
    e=Q.base[Q.front];
    Q.front=(Q.front+1)%QUEUE_MAX_SIZE;
    return OK;
}
```

算法　3.20

同出栈操作一样,出队操作也没有从物理位置对队头元素进行删除,只是在逻辑结构上删除了队头元素。

4.队列的应用

许多实际问题都可以使用队列“先进先出”的特性来解决。例如,在计算机的操作系统中,很多算法就是遵循先到先服务的原则。下面是使用队列和栈来实现判断回文问题。

【例 3-4】 编程判断一个字符串是否是回文。要求使用到队列结构。

算法思想:设字符数组 str 中存放了要判断的字符串。把字符数组中的每个字符分别进栈和入队,当所有字符都进栈和入队后,再逐个出栈和出队,并比较出栈的字符和出队的字符是否相等,若全部相等则该字符序列是回文,否则不是回文。算法中使用了队列的"先进先出"的特性及栈的"后进先出"的特性来比较原字符串的序列,具体实现过程如算法 3.21 所示。

```
Status Palindrome(ElemType str[]){
    InitStack(S);                    // 构造空栈
    InitQueue(Q);                    // 构造空队列
    i=0;
    while(str[i]!='\0'){             // 扫描字符串,所有字符进栈和入队
        Push(S,str[i]);
        EnQueue(Q,str[i]);
        i++;
    }
    while(!StackEmpty(S)&& !QueueEmpty(Q)){     // 比较字符串
        Pop(S,x);                    // 出栈和出队进行比较
        DeQueue(Q,y);
        if(x!=y)return FALSE;        // 有字符不等,则不是回文,返回 FALSE
    }
    return OK;                       // 当所有字符都比较完,返回 OK
}// Palindrome
```

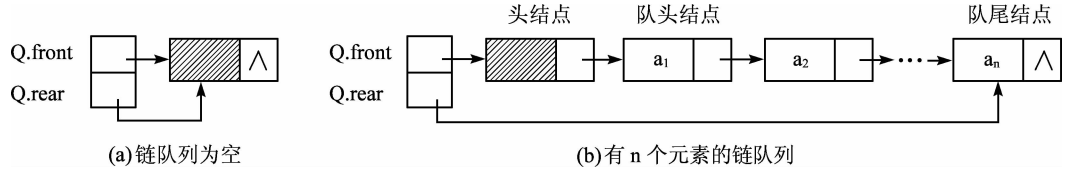
算法 3.21

3.2.3 队列的链式存储及链队列的操作

队列也可以采用链式存储结构,队列的链式存储结构简称为链队列。链队列在形式上与单链表相同,但其操作仍受到限制,插入和删除操作只能在链队列的队尾和队头进行。

1. 队列的链式存储结构

由于链队列的各种操作是在链队列的队尾和队头进行,所以必须用队尾指针和队头指针来确定所要操作的数据的位置。为了和前面介绍的单链表保持一致,在这里采用带头结点的链队列。链队列的队头指针指向链表的头结点,队尾指针指向队列的当前队尾结点。链队列为空的判定条件为队头指针和队尾指针均指向头结点,如图 3-15(a)所示。图 3-15(b)表示了一个有 n 个元素的链队列。



(a)链队列为空　　(b)有 n 个元素的链队列

图 3-15 链队列示意图

链队列中结点的结构体可定义如下：

```
typedef struct QNode{                    // 结点类型
    ElemType data;
    struct QNode *next;
}QNode, *QLink;
typedef struct{                          // 链队列类型
    QLink front;                         // 队头指针
    QLink rear;                          // 队尾指针
}LinkQueue;
```

链队列中的 QNode 由两个域组成，即存放数据元素值的数据域 data 和存放该结点直接后继结点地址的指针域 next。还定义了一个指向链队列结点的指针类型 QLink。链队列 LinkQueue 用队头指针 front 和队尾指针 rear 来描述。队头指针 front 和队尾指针 rear 均为 QLink 类型的变量。

2. 链队列的基本操作

因为链队列中的结点是动态生成的，一般不会出现上溢，因而可以不考虑队列满的情况。其具体操作的算法描述如下。

1)初始化操作 InitQueue_L(Q)

构造一个空的链队列，即为链队列生成一个头结点，其 next 域置为空，并且使队头指针和队尾指针都指向这个头结点，具体实现过程如算法 3.22 所示。

```
Status InitQueue_L(LinkQueue &Q){
    // 构造一个空队列 Q
    Q.front=Q.rear=(QLink)malloc(sizeof(QNode));  // 队头、队尾指针初始化
    Q.front->next=NULL;
    return OK;
}
```

算法 3.22

2)判空操作 QueueEmpty_L(Q)

链队列的判空操作是判断链队列 Q 是否为空，如果为空则返回 TRUE，否则返回 FALSE。当链队列为空时只存在头结点，队头指针和队尾指针都指向这个结点，因此判空条件是看队头指针是否等于队尾指针，具体实现过程如算法 3.23 所示。

```
Status QueueEmpty_L(LinkQueue Q){
    // 判断队列 Q 是否为空
    if(Q.front==Q.rear) return TRUE;  // 若队头指针等于队尾指针，则队列为空
    return FALSE;
}
```

算法 3.23

3)入队操作 EnQueue_L(&Q,e)

链队列的入队操作是在队尾插入一个新数据元素，如果操作成功返回 OK，否则返回 ERROR。因为有一个指示队尾元素的指针，所以入队操作比较简单，直接把新元素连接到队尾指针的 next 域即可。入队操作如图 3-16 所示，实现过程如算法 3.24 所示。

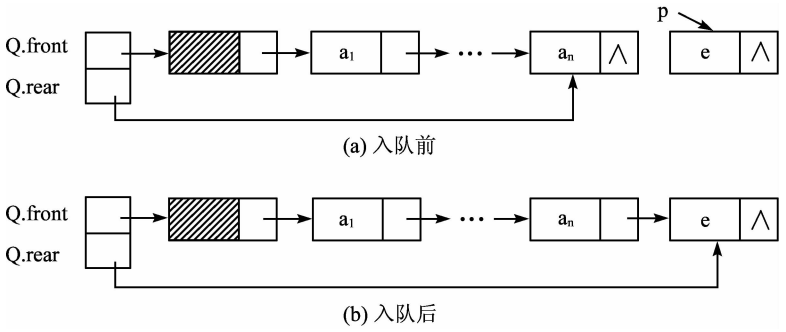


图 3-16 链队列的入队操作

```
Status EnQueue_L(LinkQueue &Q,ElemType e){
    // 插入元素 e 为 Q 的新的队尾元素
    p=(QLink)malloc(sizeof(QNode));          // 为新元素分配空间
    p->data=e;p->next=NULL;
    Q.rear->next=p;                          // 插入到队尾
    Q.rear=p;
    return OK;
}
```

算法 3.24

4)出队操作 DeQueue_L(&Q,&e)

出队操作就是删除队头元素,并由引用参数 e 回传其值。如果操作成功返回 OK,否则返回 ERROR。出队之前要先判断队列是否为空。一般情况下,链队列的出队操作和链栈的出栈操作一样,但当出队的元素是队列的最后一个元素时,要对队尾指针重新赋值。两种状态下的出队过程如图 3-17 所示,具体实现过程如算法 3.25 所示。

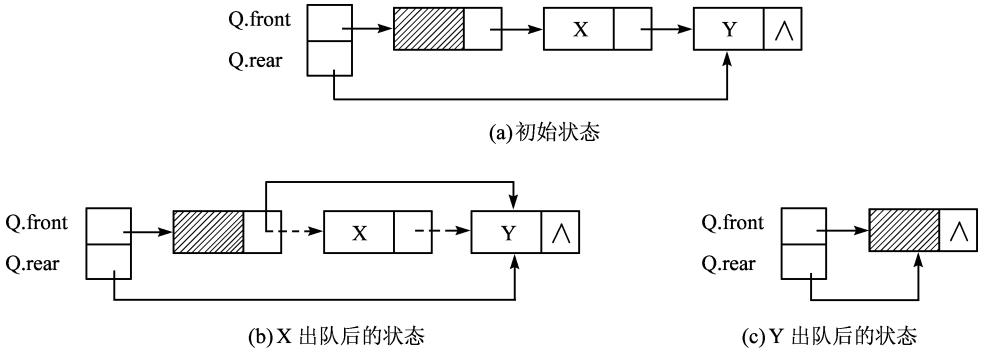


图 3-17 两个元素的链队列中的元素依次出队的过程

```
Status DeQueue_L(LinkQueue &Q,ElemType &e){
    // 若队列不空,则删除 Q 的队头元素,并用 e 返回其值
    if(Q.front==Q.rear) return ERROR;        // 队列空
    p=Q.rear->next;
    e=p->data;
    Q.front->next=p->next;                   // 删除队头元素
```

```
    if(Q.rear==p)Q.rear=Q.front;          // 只有一个元素,处理队尾指针
    free(p);
    return OK;
}
```

算法　3.25

5)销毁操作 DestroyQueue_L(&Q)

链队列的销毁操作就是依次回收链队列动态申请的空间,直到链队列为空时为止。链队列有两个指针分别指示队头和队尾的位置,理论上可以从任意一端开始回收空间。但由于指针是单向的,只能从前向后访问,所以回收空间应从队头开始。在队列销毁的过程中,把队尾指针当做一个普通指针变量来使用,使其指向待销毁队列头结点的下一个结点。链队列销毁过程就是逐个出队的过程,具体实现过程如算法 3.26 所示。

```
Status DestroyQueue_L(LinkQueue &Q){
    // 回收链队列动态申请的空间
    while(Q.front!=NULL){
        Q.rear=Q.front->next;
        free(Q.front);
        Q.front=Q.rear;
    }
    return OK;
}
```

算法　3.26

3.3　实例解析与编程实现

【例 3-5】 铁路进行列车调度时,常把站台设计成栈式结构的站台。设有编号为 1、2、3、4、5、6 的 6 辆列车,若进站的 6 辆列车顺序如上所述,那么是否能够得到 435612、325641、154623 和 135426 的出站序列,如果不能,说明原因;如果能,说明如何得到(即写出进栈和出栈的序列)。

本题不能得到 435612 和 154623 这两个出栈序列。因为若在 4、3、5、6 之后再将 1、2 出栈,则 1、2 必须一直在栈中,此时 1 先进栈,2 后进栈,2 应压在 1 上面,不可能 1 先于 2 出栈。154623 也是这种情况。出栈序列 325641 和 135426 可以得到,进栈和出栈序列如图 3-18 所示。

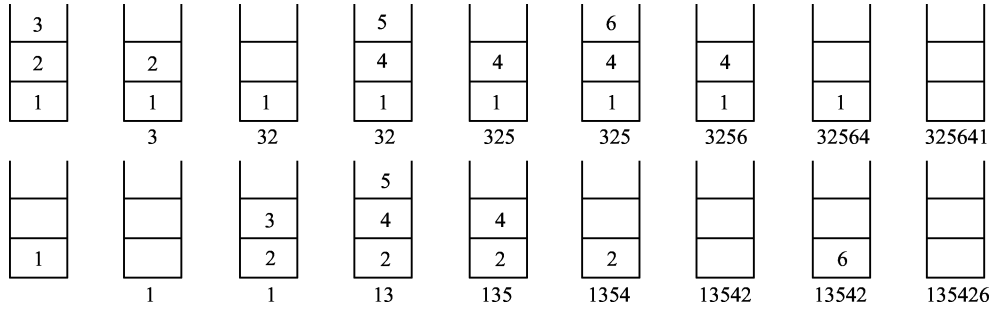


图 3-18　325641 和 135426 的进栈和出栈序列

【例 3-6】 设计一个简单的行编辑器,每当用户输完一行时,把用户输入的整行内容存储到用户数据区。在每行输入的过程中,允许用户进行差错更正。例如,如果发现前一个字符有误,可输入一个“#”表示前一个字符无效;如果发现当前输入的字符错误较多,可输入一个“@”表示当前行中的字符均无效。假如终端接收了这样的两行字符:

```
if(Q—＞##.fo#ront]#)
  print@return;
```

则实现有效的字符是:

```
if(Q.front)
```

算法思想:当用户输入时,可设置一个用户缓冲区,用以接收用户输入的一行字符,然后逐行存入用户数据区。为了便于退字或退行操作,可以把用户缓冲区设置为一个栈结构,每当从终端接收到一个字符后就进行判别:如果它既不是退格符也不是退行符,则将该字符入栈;如果是一个退格符,则从栈顶删去一个字符;如果是一个退行符,则将栈清空。编程过程中还要考虑输入字符是不是行结束符或全文结束符,若是行结束符则栈内字符存入到用户数据区;若是全文结束符则退出程序。为了验证方便,编写算法时将栈内字符传送至数据区的过程改为直接输出,具体过程如算法 3.27 所示。

```
void LineEdit(){
    // 利用字符栈 S,从终端接收一行字符并传送至调用过程的数据区
    InitStack(S);                              // 构造空栈 S 作为用户数据缓冲区
    printf("请输入一行(#:退格;@:清行):\n");
    ch=getchar();                              // 从终端接收第一个字符
    while(ch!=EOF){                            // 若为全文结束符则结束
        while(ch!=EOF && ch!='\n'){            // 若为行结束符则进行数据传送
            switch(ch){
                case '#': Pop(S,ch); break;    // 若为退格符则删去栈顶元素
                case '@': ClearStack(S);break; // 若为退行符则重置 S 为空栈
                default: Push(S,ch); break;    // 其他字符进栈
            }
        ch=getchar();                          // 从终端接收下一个字符
        }
        // 这里将栈内字符传送至数据区的过程改为直接输出
        temp=S.base;
        while(temp!=S.top){
            printf("%c",*temp);
            ++temp;
        }
        ClearStack(S);                         // 重置 S 为空栈
        printf("\n");
        if(ch!=EOF){
           printf("请输入一行(#:退格;@:清行):\n");
           ch=getchar();
```

```
        }
    }
    DestroyStack(S);
}// LineEdit
```

算法　3.27

【例 3-7】 用队列计算并打印杨辉三角。

算法思想：从杨辉三角的输出图(图 3-19)可以看出，每行的第一个数和最后一个数都是1。从第二行开始，中间数是上一行对应位置的两个数之和。如第二行的第二个数 2，是第一行上两个数 1 和 1 相加的结果；第四行上的第二个数 4，是第三行上两个数 1 和 3 相加的结果；第四行上的第三个数 6，是第三行上两个数 3 和 3 相加的结果；以此类推。

```
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
1   6   15  20  15  6   1
```

图 3-19　杨辉三角示例

如果使用一个队列来存放杨辉三角形的数据，首先把第一行上的第一个 1 放入队列中，然后把第一行上的最后一个 1 放入队列中，获取队头元素的值 t，打印 t，出队列，从第二行开始，将出队元素的值 t 与上一次出队元素的值 s 相加后，放入队列中，如果是第一个元素则 s 为 0，如果是最后一个元素，则直接把 1 放入队列中，具体过程如算法3.28 所示。

```
void YangHui(int n){
    // 利用队列打印杨辉三角
    InitQueue(Q);                    // 初始化一个队列
    EnQueue(Q,1);                    // 把第一行上的 1 放入队列
    for(i=1;i<=n;i++){               // 输出第 i 行上数据，计算第 i+1 行数据
        s=0;
        EnQueue(Q,1);                // 把每行最后一个 1 放入队列
        for(j=1;j<=i+1;j++){
            GetHead(Q,t);            // 取队头元素并输出
            printf("%3d",t);
            DeQueue(Q,e);            // 出队
            EnQueue(Q,s+t);          // 把当前数据与其前一个数据相加并入队
            s=t;
        }
    printf("\n");                    // 输完一行后换行
    }
}
```

算法　3.28

【例 3-8】 假设在周末舞会上,男士们和女士们进入舞厅时,各自排成一队。跳舞开始时,依次从男队和女队的队头上各出一人配成舞伴。若两队初始人数不相同,则较长的那一队中未配对者等待下一轮舞曲。现要求写一算法模拟上述舞伴配对问题。

算法思想:先入队的男士或女士亦先出队配成舞伴。因此该问题具有典型的先进先出特性,可用队列作为算法的数据结构。

在算法中,假设男士和女士的记录存放在一个数组中作为输入,然后依次扫描该数组的各元素,并根据性别来决定是进入男队还是女队。当这两个队列构造完成之后,依次将两队当前的队头元素出队来配成舞伴,直至某队列变空为止。此时,若某队仍有等待配对者,算法输出此队列中第一个等待者的名字,他(或她)将是下一轮舞曲开始时第一个可获得舞伴的人。

由上面分析可知,每个人应该至少有两个信息:姓名和性别。与以前问题中 ElemType 都是原子类型有所不同,所以在设计算法前首先应构造一个数据类型。

```
typedef struct{
  char name[20];                    // 姓名
  char sex;                         // 性别,F 表示女性,M 表示男性
}Person;
typedef Person ElemType;            // 将队列中元素的数据类型改为 Person
```

具体模拟过程如算法 3.29 所示。

```
void DancePartners(Person dancer[],int num){
    // 结构数组 dancer 中存放跳舞的男女,num 是跳舞的人数
    InitQueue(Mdancers);            // 男士队列初始化
    InitQueue(Fdancers);            // 女士队列初始化
    for(i=0;i<num;i++){             // 依次将跳舞者依其性别入队
        p=dancer[i];
        if(p.sex=='F')
            EnQueue(Fdancers,p);    // 排入女队
        else
            EnQueue(Mdancers,p);    // 排入男队
    }
    printf("The dancing partners are: \n \n");
    while(!QueueEmpty(Fdancers) && !QueueEmpty(Mdancers)){
        // 依次输入男女舞伴名
        DeQueue(Fdancers,p);        // 女士出队
        printf("%s\n",p.name);      // 打印出队女士名
        DeQueue(Mdancers,p);        // 男士出队
        printf("%s\n",p.name);      // 打印出队男士名
    }
    if(!QueueEmpty(Fdancers)){      // 输出女士队列队头者名字
        GetHead(Fdancers,p);
        printf("%s will be the first to get a partner. \n",p.name);
```

```
    }else if(!QueueEmpty(Mdancers)){ // 输出男士队列队头者名字
        GetHead(Mdancers,p);
        printf("%s will be the first to get a partner.\n",p.name);
    }
}// DancerPartners
```

算法　3.29

【例 3-9】 求 2 阶 Fibonacci 数列的计算函数，形式如下，试写出函数的递归算法。

$$\mathrm{Fib}(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2) & n\geqslant 2\end{cases}$$

算法思想：对于这个求 2 阶 Fibonacci 数列的函数，第一项为 0，第二项为 1，从第三项开始，每项的值都是其前两项值的和，可见它是一个递归的定义。当 n＝0 时，Fib(n)＝0，当 n＝1时，Fib(n)＝1，这是递归的终止条件，对应的操作可用条件语句来实现。当 n≥2 时，处于非递归终止条件，可通过改变参数 n 的值递归调用函数本身即 Fib(n－1)＋Fib(n－2)，完成剩余的工作。详细过程如算法 3.30 所示。

```
int Fib(int n){
    if(n==0)return 0;
    else if(n==1)
        return 1;
    else return(Fib(n-1)+Fib(n-2));
}
```

算法　3.30

本章小结

栈和队列都是运算受限的线性表，是软件设计中比较基础的数据结构。

栈是限定在表的一端进行插入和删除操作的线性表。允许操作的一端称为栈顶，另一端称为栈底。栈的操作是按照后进先出原则进行的，因此又称为后进先出的线性表。

栈的顺序存储表示又称顺序栈，它利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时设栈顶指针指示栈顶元素的下一个位置。顺序栈是较方便的栈的存储形式，但可能会有溢出。链栈是栈的链式存储结构。栈是实现程序调用的基本数据结构，递归调用是栈的最典型应用实例。

队列是只允许在表的一端进行插入操作，而在另一端进行删除操作的线性表。允许插入的一端称队尾，允许删除的一端称队头。队列的操作是按照先进先出原则进行的，因此又称为先进先出的线性表。

队列的顺序存储也是用一组地址连续的存储空间存放队列中的元素，并设置两个分别指示队头元素和队尾元素位置的指针。由于顺序队列可能会出现“假溢出”而引入循环队列的概念。循环队列是把数组想象为一个首尾相接的环。为区分循环队列的满和空的状态，一般要将队列的最大元素个数限定为比存储空间元素个数少 1。队列是软件设计中有关排

队问题的基础。链队列是队列的链式存储形式。

习 题 3

1. 简述栈与队列的相同点与不同点。

2. 在顺序队列中,什么叫真溢出? 什么叫假溢出? 为什么顺序队列通常都采用循环队列结构?

3. 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(不设头指针),试编写相应的入队、出队算法。

4. 设有两个栈 s1 和 s2 都采用顺序表示,并且共享一个存储区。现采用栈顶相对,迎面增长的方式存储。请分别编写 s1 和 s2 进栈操作 push 函数和出栈操作 pop 函数。

5. 设计一个输出如图 3-20 所示形式数值的递归算法。

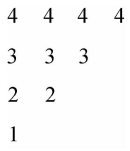


图 3-20 第 5 题输出图形