第 1 章 绪　　论

数据结构作为一门学科主要研究数据的各种逻辑结构和存储结构，以及对数据的各种操作。数据结构反映数据的内部构成，是数据存在的形式，即一个数据由哪些成分数据构成，以什么方式构成，呈现什么结构。在我们研究各种数据结构时，需要弄清楚：为什么要使用这个结构，如何使用这个结构，以及何时使用这个结构。数据结构有逻辑上的数据结构和物理上的数据结构之分。逻辑上的数据结构反映成分数据之间的逻辑关系，而物理上的数据结构则反映成分数据在计算机内部的存储安排。数据结构是信息的一种组织方式，其目的是提高算法的效率，它通常与一组算法的集合相对应，通过这组算法集合可以对数据结构中的数据进行某种操作。

在开始学习数据结构之前，需要弄清楚什么是数据结构，它研究什么，采用什么方法学习数据结构，相对应的算法集合如何评价，数据结构的描述规范等。

本章学习要点

- 数据结构的基本概念。
- 抽象数据类型。
- 算法描述。
- 算法分析。
- 数据结构的 C 语言表示。

1.1 数据结构的基本概念

本节我们将给出数据结构的一些基本概念，在这之前先看几个数据结构的实例。如表 1-1 所示的学生选课表中，学号、课程名、教师、成绩都是数据项(item)，是有独立含义的最小单位，其取值范围是数据域；一行称为一条记录(record)。我们可以用 C 语言或者其他语言描述这些记录，将其定义为结构，那么整个选课表就是一个线性表结构。

表 1-1　学生选课表

学　　号	课 程 名	教　　师	成　　绩
101	操作系统	卢春燕	90
……	……	……	……

如图 1-1 所示的某报刊订阅管理系统结构图中，登录、用户、管理员、录入、查询等都是结点；登录和管理员、管理员和查询之间是层次关系或祖先后裔关系。这个订阅管理系统结构图就是一个非线性结构，我们称之为树。

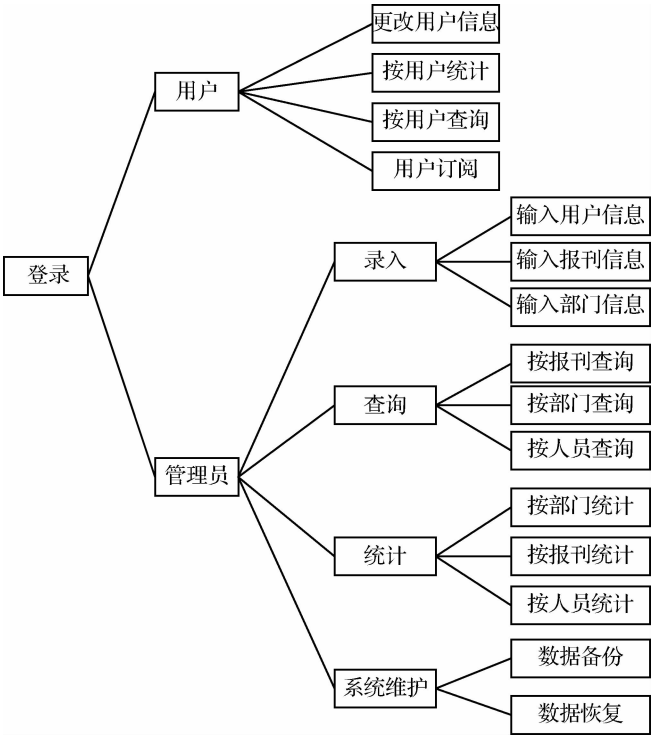


图 1-1　某报刊订阅管理系统结构图

1)数据结构中的一些基本概念

下面，我们给出数据结构中一些基本概念的定义。

(1)数据。数据(data)是对客观事物的描述，是能被输入到计算机中进行加工处理的各种符号集合，如数值、字符、声音、图像、视频等。对计算机科学而言，数据的含义非常宽泛。简而言之，数据就是计算机化的信息。

(2)数据元素。数据元素(data element)是构成数据的基本单位，在计算机程序中通常被作为一个整体进行考虑和处理。如表 1-1 中所示的选课记录，图 1-1 中所示的树都被称为数据元素。

(3)数据项。数据项(date item)是数据不可分割的最小单位，如学号、课程名、成绩等。一个数据元素可由一个或若干个数据项组成。例如，一个学生的选课信息作为一个数据元素，该数据元素包括学号、课程名、成绩等数据项。

(4)数据对象。数据对象(data object)是由性质相同的数据元素组成的集合，是数据的

一个子集。例如，由整数组成的数据集合 $D=\{0,\pm1,\pm2,\cdots\}$，由 26 个字母组成的集合 $C=\{A,B,\cdots,Z\}$，其中，D 是无限集，C 是有限集。表 1-1 所示的选课表也是一个数据对象。

(5)数据结构。数据结构(data structure)是指集合中数据元素相互之间存在一种或多种特定关系，由一组数据对象及其数据成员之间的联系组成，用以表述数据的组织形式。例如，表结构(表 1-1 所示的学生选课表)、树结构(图 1-1 所示的某报刊订阅管理系统结构图)。

数据结构的形式化为一个二元组，记为：

$$\text{Data\_Struture}=(D,R) \tag{1-1}$$

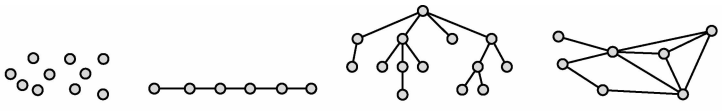
其中，D 为数据对象的有限集，R 是 D 上关系的有限集。

(6)数据类型。数据类型(data type)是一个值的集合以及定义在这个集合上的一组操作的总称。数据类型显式或隐式地定义了取值范围，以及在该值上允许进行的一组运算。例如，高级程序语言中的数据类型就是已经实现的数据结构的实例，分为非结构类型和结构类型两类，非结构类型如 C 语言中的整型、实型、字符型、枚举类型、指针类型等，结构类型如结构体和共用体。数据类型是程序语言中允许的变量种类，是已经实现的数据结构。例如，C 语言中的整型可能的取值范围是－32 768～＋32 767，可允许的操作集合为加、减、乘、除、乘方、取模。

2)数据结构的内容

数据结构包括逻辑结构、物理结构和操作集合。

(1)逻辑结构。数据对象内各元素之间的逻辑关系称为逻辑结构，其形式化描述见公式(1-1)所示。根据这些逻辑关系的性质划分，通常有 4 类基本结构，即集合结构、线性结构、树形结构和图形结构，如图 1-2 所示。



(a) 集合结构　(b) 线性结构　(c) 树形结构　(d) 图形结构

图 1-2　4 类基本结构

在这些数据对象内，元素之间的关系存在如下一些特点：

①集合中的数据元素之间除了同属于一个集合的关系外，无任何其他关系。

②线性结构中的数据元素之间存在一对一的关系。

③树形结构中的数据元素之间存在一对多的关系。

④图形结构中的数据元素之间存在多对多的关系。

因此，数据的逻辑结构可概括如下：

$$\text{逻辑结构}\begin{cases}\text{线性结构——所有元素按次序排列在一个序列中}\\ \text{非线性结构——每个元素可以与若干个其他元素关联}\end{cases}$$

(2)物理结构。逻辑结构在计算机中的实现称为数据的物理结构，又称为存储结构。物理结构是逻辑结构和数据的物理映像，逻辑结构是数据结构的模型抽象，两者共同确立了数据对象中元素之间的结构关系。物理结构的形式化描述如下：

计算机中要存入 D，建立从 D 到存储空间 S 的映像 M，$D\rightarrow S$，即对每一个 $d(d\in D)$ 都有唯一的 $z\in S$，使得 $M(D)=Z$，且映像必须包含关系 R。

顺序存储结构和链式存储结构是最主要的两种存储方式。

①顺序存储结构。顺序存储结构是指用数据元素在存储器中的相对位置来表示元素之间的逻辑关系。

例如,令 y 的存储位置和 x 的存储位置之间差一个常量 C,C 是一个隐含值,整个存储结构中只含数据元素本身的信息,如图 1-3(a)所示。

②链式存储结构。链式存储结构是指用数据元素存储地址的指针来表示数据元素之间的逻辑关系。

例如,以指针表示后继关系,需要用一个和 x 在一起的附加信息指示 y 的存储位置,如图 1-3(b)所示。

(a) 顺序存储结构　　(b) 链式存储结构

图 1-3　存储结构示意图

(3)操作集合。建立数据结构的目的是在计算机中实现对数据的操作。例如,对数据的增加、删除、修改、查找、排序等操作。

综上所述,按某种逻辑模型组织起来的一批数据通过映像将它们存放在计算机的存储器中,并给这些数据定义操作的集合,称为数据结构。

1.2　抽象数据类型

抽象是人类认识世界的一种方式,是指从同类型的众多事物中舍弃个别的、非本质的属性和行为,而抽取共同的、本质的属性和行为的过程。例如,要表示空间上的一个四边形,就要对其进行抽象,四边形有正方形、长方形、菱形以及任意的不规则四边形,只要是由四条边组成的平面上的封闭图形就是四边形。它们的边长可能各不相同;各边的颜色也可能不一样;它们的形状也可能各异,如可以是凹的,也可以是凸的;甚至各图形在空间上的位置也各不相同。要表示这样的图形,可用四边形有序的 4 个顶点和边来描述,由这 4 个值及它们的先后顺序就可以描述四边形在某时刻的形状及在空间上的位置,借助类似的抽象方法我们还可以描述更为复杂的树形结构和网状结构。数据结构就是使用这种抽象的概念来描述各类程序中数据的组织形式。

通过对各种数据的抽象,C 语言提供了整型、实型、字符、指针等多种数据类型,利用这些数据类型定义更高级的数据抽象,可以构造出像栈、队列、树、图等复杂的抽象数据类型。

抽象数据类型(abstract data type,ADT)是一组数据对象以及其中各元素间的结构关系和其上的一组操作的集合。ADT 可以形式化地用三元组表示为:

```
ADT 抽象数据类型名{
    数据对象 D:{对象定义}
```

数据关系 R:{关系定义}

基本操作 P:{操作定义}

}ADT 抽象数据类型名

我们熟知的数据类型与抽象数据类型从本质上看是同一个概念。例如,字符类型是定义好的字符集合,每个字符用 ASCII 码表示,并且预定义了相关的操作。当在程序设计中使用字符时,并不需要了解字符数据在计算机内如何表示,也不需要了解对字符的操作是如何实现的,只需要掌握如何使用字符的操作即可。字符的具体存储方式和操作过程如何实现与字符的使用无关。而 ADT 定义的范围更为广泛,除了在不同的语言中已经定义并实现的数据类型外,还包括设计软件系统时用户自定义的复杂数据类型。

ADT 包括定义和实现两方面,其中定义是独立于实现的,定义仅需要给出一个 ADT 的逻辑特性,不需要考虑如何在计算机中实现。

下面通过一个例子来理解数据抽象和抽象数据类型的概念。

【例 1-1】 抽象数据类型整数的定义。

完整的定义包括两个部分:数据对象和操作。其中,对象定义为整数有序子序列,从 0 开始,到计算机上的最大整数结束,不涉及整数存储。符号 x、y 表示自然数集上的两个元素,True 和 False 是布尔集上的两个元素。定义在整数集上的操作包括加、减、相等以及小于等。定义的操作如表 1-2 所示。

表 1-2 ADT 整数

数据对象 / 操作集合	x,y,最大整数		
	实现功能	输入参数	返回值
Zero()	无	0	类似构造函数,没有参数
Is_Zero(x)	x	True/False	如果 x 值为 0,则返回 False,否则返回 True
Add(x,y)	x,y	整数	自然数求和
Equal(x,y)	x,y	True/False	自然数求相等
Subtract(x,y)	x,y	整数	自然数求差
Successor(x)	x	整数	累加器,返回 x 的下一个自然数。如果不存在,即 x 已经是机器的最大整数,那么返回机器最大整数

表 1-2 说明了 ADT 定义的一般形式。我们在具体数据结构的 ADT 定义中,一般不提供类似 C 函数的操作定义。从应用的角度出发,ADT 定义的实质在于避开具体的实现细节,即在一般情况下,用自然语言的形式来解释操作的意义,用户可以选择不同的语言和机器完成物理实现。因此,抽象数据类型的定义仅仅取决于客观存在的逻辑属性,与采用何种语言以及计算机如何表示和实现无关,之所以使用 ADT,就是为实现软件的构件化和可重用性提供理论保证,进而提高软件生产率。

ADT 通常是指由用户定义且用以表示应用问题的数据模型,一般由基本的数据类型组成,并包括相关服务操作集合。本课程中将要学习的表、堆栈、队列、串、树、图等结构就是不同的抽象数据类型。

一般来说,数据类型的抽象程度越高,包含该抽象数据类型的软件复用程度就越高。

ADT定义了抽象数据类型需要包含哪些信息,并根据功能确定服务接口,使用者可以使用接口对该抽象数据类型进行操作。从应用的角度看,只要了解该抽象数据类型的规格说明,就可以利用其接口的服务来使用这个类型,而不必关心其物理实现,从而集中精力考虑如何解决实际问题。

ADT物理实现作为私有部分封装在其实现模块内,具体做法就是将描述数据对象状态的属性和数据对象固有的行为分别用数据结构和方法来加以描述,并将它们捆绑在一起形成一个可供外界访问的独立的逻辑单元,外界只能通过客体所提供的方法来对其间的数据结构加以访问,而不能直接存取。很明显,封装是实现信息隐藏的有效手段,它尽可能地隐蔽对象的内部细节,只保留有限的对外接口,使之与外部发生联系。封装保证了数据的安全性,提高了应用系统的可维护性,也有利于软件的移植与重用。

1.3 算法描述

Pascal之父、结构化程序设计的先驱Nicklaus Wirth最著名的一本书是《算法+数据结构=程序》,该书提纲挈领地表明了数据结构和算法是程序的两大组成部分,二者相辅相成,缺一不可。

问题(problem)是需要研究讨论并完成的任务或需求;算法(algorithm)是对特定问题求解步骤的一种描述,它是指令的有限序列,其中,每一条指令表示一个或多个操作;程序(program)是用具体语言对算法的实现。

【例1-2】 计算1−1/2+1/3−1/4+1/5−…+1/99−1/100。

算法1:自左至右逐项相加或相减。

算法2:将多项式写成两个多项式之差:(1+1/3+1/5+…+1/99)−(1/2+1/4+1/6+…+1/100)。

算法3:先通分,再运算。

算法是抽象的,可利用自然语言、类语言进行描述。自然语言或类语言简单明了但容易产生歧义;而程序必须是准确和严谨的,只能用程序设计语言来表示。

近年来,在计算机科学研究、嵌入式系统开发、教学和实践中,C语言的使用范围越来越广,已成为计算机专业与非计算机专业必修的高级程序设计语言。C语言类型丰富,执行效率高,本教材采用标准C语言作为描述算法的工具(假设读者在学习数据结构课程之前,都已了解和掌握了C语言)。

算法具备以下5个重要特性:

(1)有穷性。对于任意一组合法输入值,在执行有限步骤之后必须能结束,即算法中的每个步骤都能在有限时间内完成。

(2)确定性。在任意情况下所需执行的操作,在算法中都有明确的规定,使得算法的执行机构都能明确其含义及如何执行。并且在任何条件下,算法都只有一条执行路径。

(3)可行性。算法中的所有操作都必须足够精确,都可以通过已经实现的基本运算执行有限次实现。

(4)输入。算法具有若干个或0个输入。有些算法需要在执行过程中输入,而有些算法表面上可以没有输入,实际上已被嵌入算法之中。

(5)输出。它是一组与“输入”有确定关系的量值,是算法进行信息加工后得到的结果,这种确定关系即为算法的功能。

通常设计一个“好”的算法应考虑达到以下目标:

(1)正确性。首先,算法应当满足以特定的“规格说明”方式给出的需求。其次,对算法是否“正确”的理解可以有以下4个层次:

- 程序中不含语法错误。
- 程序对于几组输入数据能够得出满足要求的结果。
- 程序对于精心选择的、典型、苛刻且带有刁难性的几组输入数据能够得出满足要求的结果。
- 程序对于一切合法的输入数据都能得出满足要求的结果。

(2)可读性。要有利于人们对算法的理解;有利于程序的调试和维护、交流和移植。

(3)健壮性。当输入的数据非法时,算法应当恰当地作出反应或进行相应处理,返回一个表示错误或错误性质的值,以便在更高的抽象层次上进行处理,而不是产生莫名其妙的输出结果或者中断程序运行。

(4)高效率与低存储量需求。通常,高效率指的是算法执行时间要尽可能少;低存储量指的是算法执行过程中所需的最大存储空间要尽可能小,两者都与问题的规模有关。

1.4 算法分析

数据结构的优劣可以用实现的算法来衡量,对数据结构的分析就转化成了对算法的分析,一是要验证算法是否能正确解决问题,二是要对算法的效率作性能评价。性能评价分为算法分析和性能测量。算法分析是复杂性理论的核心内容,目的是获取与具体机器无关的时间和空间的估计。在计算机程序设计中,程序是否有效地利用存储空间,程序的执行时间是否可以接受,是对算法效率进行评价的核心标准。本节我们关注对算法的分析,即对于一个实际问题的解决,如何从若干个可行的算法中找出最有效的算法,就是以算法执行所需的机器时间和所占用的存储空间作为标准。

对一个算法的效率作出分析,就是要度量算法的执行时间,通常使用以下两种方法。

(1)事后统计法。不同的算法可通过统计实际执行时间来分辨优劣。但该方法存在缺点:一是必须先运行依据算法编制的程序;二是所得时间的统计量依赖于计算机的硬件、软件等环境因素,有时容易掩盖算法本身的优劣。

(2)事前分析估算法。求出算法的时间规模函数。

与算法执行时间相关的因素包括:

- 算法选择的策略。
- 待解决问题的规模。
- 编写程序的语言。

• 编译程序产生的机器代码的质量。

• 计算机执行指令的速度。

显然,对于相同的算法,使用不同的编译器、不同的程序语言或者在不同的机器上运行,效率均不相同。因此用绝对的时间单位衡量算法的效率是不合适的,还应考虑与计算机硬件、软件有关的因素。可以认为:一个特定算法的"运行工作量"的大小只依赖于问题的规模(通常用整数 n 表示),或者说,它是问题规模的函数,而问题规模 n 对于不同的问题其含义是不同的,对矩阵是阶数,对多项式运算是多项式项数,对图是顶点个数,对集合运算是集合中的元素个数。

1.4.1 时间复杂度

一个算法的执行时间大致上等于其编译时间和所有语句执行时间的总和,由于编译时间不依赖于问题的特征,所以对算法效率的分析主要集中在算法的执行时间上。

算法=控制结构(顺序、分支和循环)+基本操作

基本操作是指操作时间与问题规模无关的操作。例如,给变量赋值、比较两个数的大小、四则运算等,而 n 个加法操作就不是基本操作,因为它与输入规模有关。算法的时间取决于控制结构和基本操作两者的综合效果。为了便于比较同一问题的不同算法,通常的做法是,从算法中选取一种相对于所研究的问题(或算法类型)的基本操作,以该基本操作重复执行的次数作为算法的时间度量。

设 n 为求解问题的规模,基本操作执行次数的总和称为语句频度,记为 $f(n)$,算法中语句总的执行次数记为 $T(n)$。通常情况下,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。这里用 O 来表示 $T(n)$随 n 的变化情况所达到的数量级,算法的时间量度记为:

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

它表示随问题规模 n 的增大,算法执行时间的增长率和 $f(n)$的增长率相同,称做算法的渐进时间复杂度,简称时间复杂度。

由于算法的时间复杂度考虑的只是问题规模 n 的增长率,所以在难以精确计算基本操作执行次数(或语句频度)的情况下,只需求出它关于 n 的增长率或阶即可。一般情况下,随着 n 的增大,$T(n)$增长较慢的算法为最优算法。

定义:若 $O(f(n))=O(g(n))$,则当且仅当存在正的常数 c 和 n_0,使得对所有的 n,当 $n\geqslant n_0$ 时,有 $f(n)\leqslant cg(n)$。

【例 1-3】 时间复杂度分析。

```
①++x;                          O(1)      常量阶
②for(i=1;i<=n;i++)
  {++x;}                       O(n)      线性阶
③for(i=1;i<=n;i++)
     for(j=1;j<=n;j++)
    {++x;}                     O(n²)     平方阶
```

常用的时间复杂度的阶有 7 个,其复杂度从小到大依次为:$O(1)$常数阶,$O(\log_2 n)$对数阶,$O(n)$线性阶,$O(n\log_2 n)$二维阶,$O(n^2)$平方阶,$O(n^3)$立方阶,$O(2^n)$指数阶。

由图 1-4 可知，随着问题规模 n 的增大，不同阶的时间复杂度增长快慢不一，因此，应尽可能选用多项式阶算法，而避免使用指数阶的算法。

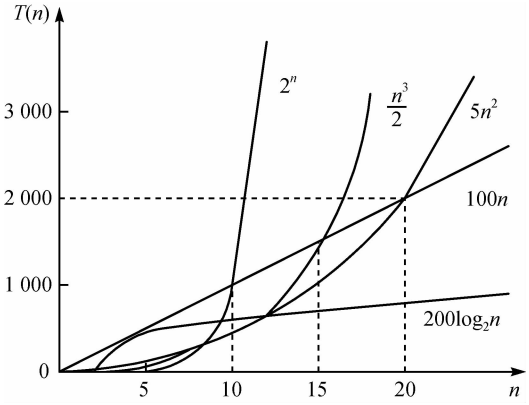


图 1-4　不同阶的时间复杂度示意图

【例 1-4】 两个矩阵相乘，判断其时间复杂度。

```
void mult(int a[],int b[],int & c[])
{
    /*以二维数组存储矩阵元素,c 是 a 和 b 的乘积*/
    for(i=1;i<=n;++i)                          /*n 次*/
        for(j=1;j<=n;++j){                     /*n² 次*/
            c[i,j]=0;                          /*n² 次*/
            for(k=1;k<=n;++k)                  /*n³ 次*/
                c[i,j]+=a[i,k]*b[k,j];         /*n³ 次*/
        }
}
```

算法的基本操作为乘法操作，语句频度为：$f(n)=n+n^2+n^2+n^3+n^3=2n^3+2n^2+n$。

时间复杂度为：$T(n)=O(f(n))=O(2n^3+2n^2+n)=O(n^3)$。

所以算法的时间代价是输入规模 n 的立方阶。

【例 1-5】 选择排序，判断其时间复杂度。

```
void select_sort(int& a[],int n)
{
    /*将 a 中整数序列重新排列成自小至大有序的整数序列*/
    for(i=0;i<n-1;++i){
        j=i;                          /*选择第 i 个最小元素*/
        for(k=i+1;k<n;++k)
            if(a[k]<a[j])j=k;
        if(j!=i)a[j]←→a[i]
    }
}
```

算法的基本操作为比较操作,时间复杂度为:$T(n)=O(f(n))=O(n^2)$。

【例 1-6】 冒泡排序,判断其时间复杂度。

```
void bubble_sort(int& a[],int n)
{
    /*将a中整数序列重新排列成自小至大有序的整数序列*/
    for(i=n-1;change=true;i>1 && change;--i){
        change=false;     /*change为元素进行交换标志*/
        for(j=0;j<i;++j)
            if(a[j]>a[j+1])
            {
                a[j]←→a[j+1];
                change=true;
            }
    }                          /*一趟冒泡*/
}
```

算法的基本操作为赋值操作,时间复杂度为:$T(n)=O(f(n))=O(n^2)$。

即便是问题的输入规模一致,算法中基本操作重复执行的次数也会随输入数据集的不同而不同。例如,在【例 1-4】中,当 a 中初始序列为递增有序时,基本操作的执行次数为 0;而当初始序列为递减有序时,基本操作的执行次数就会变为 $n(n-1)/2$,此时相应的时间复杂度应该是算法的平均时间复杂度。但是在很多情况下,分析算法的平均时间复杂度是不可行的,因为它要求掌握程序的实际输入在所有可能输入集合中的分布概率。因此,进行算法复杂度分析时,主要是讨论算法在最坏情况下的时间复杂度,即最坏时间复杂度,以估计出算法执行时间的上界。例如,冒泡排序的最坏时间复杂度为 $T(n)=O(n^2)$。在后续章节中,除非说明,所讨论算法的时间复杂度均指最坏时间复杂度。

1.4.2 空间复杂度

除了时间代价,空间代价也是我们对算法进行分析时要考虑的问题。空间复杂度是算法所需存储空间的度量,记为:

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

其中,n 是问题的规模(或大小)。

空间代价一般包括两部分:固定的空间需求和可变的空间需求。固定的空间需求是指不依赖程序输入、输出数量和大小的空间需求,包括指令存储空间,存储简单变量、固定大小的结构变量(如结构体)和常量的存储空间;可变的空间需求包括输入和输出数量、大小和值,以及结构变量所需的存储空间,这些结构变量的大小依赖问题的特定实例,同时还包括函数递归调用所需的额外存储空间。

算法执行时间的成本和所占存储空间的成本是矛盾的,难以兼顾。即算法执行时间上的节省一定是以增加存储空间为代价的,反之亦然。例如,程序对数据进行压缩以节省存储空间,代价就是压缩和解压缩需要付出额外的时间成本,即“以时间换取空间”;相反,预先组

织数据可以提高运行速度，但占用较大的存储空间，即“以空间换时间”。算法设计通常遵循空间与时间均衡的原则，但一般情况下，以算法执行时间作为算法优劣的主要衡量指标。

1.5 数据结构的C语言表示

描述一个算法可以采用不同的方法，常用的有：自然语言、流程图及改进的流程图、伪代码、N-S结构流程图、PAD图。

本书主要采用C语言描述数据结构，也采用伪码描述一些只含抽象操作的抽象算法。力图使得数据结构和算法的描述与讨论简明清晰，既利用了C语言高效和灵活的特点，又不拘泥于C语言的细节。

1）ADT的定义

在数据结构中，我们首先讨论如何用C语言来表述抽象数据类型ADT。ADT包括定义和实现两方面，其中定义独立于实现。定义仅给出一个ADT的逻辑特性，不必考虑如何在计算机中实现。

```
ADT 抽象数据类型名{
    数据对象:{对象定义}
    数据关系:{关系定义}
    基本操作:{操作定义}
}ADT 抽象数据类型名
```

其中，数据对象、数据关系用伪码描述。基本操作定义格式为：

```
基本操作名(参数表)
前置条件:<先决条件描述>
后置条件:<操作结果描述>
```

【例 1-7】 给出“串”的抽象数据类型定义。

```
ADT string
  数据对象:{Si|Si∈字符集,i=0,1,…,n-1,n≥0}
  数据关系:{R|R={<Si-1,Si>|Si-1,Si∈Dataset,i=1,3,…,n-1}}
  基本操作:设 S,S1,S2 为 string
    StrAssign(S,chars):赋值,由 chars 得到串 S;
    StrDestroy(S):销毁串;
    StrCopy(S1,S2):复制串 S2,得到 S1;
    StrCompare(S1,S2):对串 S1 和 S2 进行比较;
    StrCombine(S1,S2,S3):连接串 S1 和 S2 得到新的串 S3;
    StrReplace(S1,S2,S3):用串 S3 替换串 S1 中与 S2 相等不重叠的子串,返回新的 S1;
    …
```

可以发现，在ADT定义中，数据元素所属的数据对象没有局限于具体的数据类型，所定义的操作也是抽象的，并没有具体到何种计算机语言指令与程序编码。

2)用C语言实现ADT

用C语言实现ADT时,主要包括以下两个方面:

(1)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,以便增强程序的抽象性、简洁性和可读性。

(2)用C语言的子函数实现各个操作。

typedef可以用来为已定义好的数据类型创建别名。为创建较短、便于记忆和使用的类型名,也为了省略结构标记,程序员经常使用typedef定义结构类型。

【例1-8】 串的结构定义。

```
typedef struct string        /*串的动态数组结构定义*/
{
    char *str;               /*指向存储串的基地址*/
    int maxlength;           /*动态数组字符的最大个数*/
    int length;              /*设置当前串长值*/
}DString, *DStr;
```

typedef同时定义了两个新的数据类型名:DString和DStr。其中,DString为结构体类型,可用DString代替struct string类型名,而DStr为结构体指针类型,可用DStr代替struct string *类型名。【例1-8】也可以分成下面两步:

```
typedef struct string        /*串的动态数组结构定义*/
{
    char *str;
    int maxlength;
    int length;
}DString;
typedef DString, *DStr;
```

数据类型和变量的关系:数据类型定义的是规范,变量定义才会分配空间。在【例1-8】中虽然DString是标识符的形式,但DString并不是变量名,而是数据类型名。

3)基本操作的算法描述

算法表示形式如下:

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

函数的参数需要说明类型,但算法中使用的辅助变量可以不作变量说明,必要时对其作用给予注释,注释统一使用"/*字符串*/"形式,以提高程序的可读性。

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,除了使用值调用方式外,还使用指针参数传递方式,C语言中指针类型的参数传递可以看做是传地址方式。

1.6 习 题

1)选择题

(1)算法的时间复杂度取决于(　　)。

A. 问题的规模　　B. 待处理数据的初态

C. 程序的质量　　D. A 和 B

(2)设某数据结构的二元组形式表示为 $A=(D,R)$,$D=\{01,02,03,04,05,06,07,08,09\}$,$R=\{r\}$,$r=\{<01,02>,<01,03>,<01,04>,<02,05>,<02,06>,<03,07>,<03,08>,<03,09>\}$,则数据结构 A 是(　　)。

A. 线性结构　　B. 树形结构　　C. 物理结构　　D. 图形结构

(3)以下数据结构中(　　)是非线性结构。

A. 队列　　B. 栈　　C. 线性表　　D. 图

(4)下面程序的时间复杂度为(　　)。

```
for(i=1,s=0;i<=n;i++){t=1;for(j=1;j<=i;j++)t=t*j;s=s+t;}
```

A. $O(n)$　　B. $O(n^2)$　　C. $O(n^3)$　　D. $O(n^4)$

(5)数据的最小单位是(　　)。

A. 数据项　　B. 数据类型　　C. 数据元素　　D. 数据变量

(6)程序段"s=i=0;do{i=i+1;s=s+i;}while(i<=n);"的时间复杂度为(　　)。

A. $O(n)$　　B. $O(n\log_2 n)$　　C. $O(n^2)$　　D. $O(n^3/2)$

(7)下列说法正确的是(　　)。

A. 数据是数据元素的基本单位

B. 数据元素是数据项中不可分割的最小标识单位

C. 数据可由若干个数据元素构成

D. 数据项可由若干个数据元素构成

(8)下列程序段的时间复杂度为(　　)。

```
i=0,s=0;while(s<n){s=s+i;i++;}
```

A. $O(n^{1/2})$　　B. $O(n^{1/3})$　　C. $O(n)$　　D. $O(n^2)$

(9)某程序的时间复杂度为$(3n+n\log_2 n+n^2+8)$,其数量级表示为(　　)。

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

(10)线性表是一个具有 n 个(　　)的有限序列。

A. 表元素　　B. 字符　　C. 数据元素　　D. 数据项

(11)从逻辑上可以把数据结构分为(　　)。

A. 动态结构、静态结构　　B. 顺序结构、链式结构

C. 线性结构、非线性结构　　D. 初等结构、构造型结构

(12)关于算法的描述,不正确的是(　　)。

A. 算法最终必须由计算机程序实现

B. 所谓时间复杂度是指最坏情况下,估算算法执行时间的一个上界
C. 健壮的算法不会因非法的输入数据而出现莫名其妙的状态
D. 算法的优劣与算法描述语言无关

(13)在数据结构中,数据的基本单位是(　　)。

A. 数据项　　B. 数据元素　　C. 数据对象　　D. 数据文件

(14)有以下程序段:

```
k=1;
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        A[i][j]=k++;
```

上述程序段的时间复杂度为(　　)。

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

(15)有以下程序段:

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

上述程序段的时间复杂度为(　　)。

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

(16)从逻辑关系来看,数据元素的直接前驱为0个或1个的数据结构只能是(　　)。

A. 线性结构　　B. 树形结构
C. 线性结构和树形结构　　D. 线性结构和图形结构

(17)下列程序的时间复杂度为(　　)。

```
i=0;s=0;
while(s<n)
{
    i++;
    s=s+i;
}
```

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

(18)数据结构中所定义的数据元素是用于表示数据的(　　)。

A. 最小单位　　B. 最大单位
C. 基本单位　　D. 不可分割的单位

(19)数据的四种基本存储结构是指(　　)。

A. 顺序存储结构、索引存储结构、直接存储结构、倒排存储结构
B. 顺序存储结构、索引存储结构、链式存储结构、散列存储结构
C. 顺序存储结构、非顺序存储结构、指针存储结构、树形存储结构
D. 顺序存储结构、链式存储结构、树形存储结构、图形存储结构

(20)下列基本的逻辑结构中,结构结点间不存在任何逻辑联系的是(　　)。

A. 集合　　B. 线性结构　　C. 树形结构　　D. 图形结构

2)填空题

(1)通常从4个方面评价算法的质量:_____、_____、_____和_____。

(2)通常是以执行算法所需要的_____和所占用的_____来判断一个算法的优劣。

(3)一个算法的时间复杂度为$(n^3+n^2\log_2 n+14n)/n^2$,其数量级表示为_____。

(4)一个算法应具备的特性为_____、_____、_____、_____和_____。

(5)数据的物理结构主要包括_____和_____两种情况。

(6)数据结构从逻辑上划分的基本类型有:_____、_____和_____。

(7)for(i=1,t=1,s=0;i<=n;i++){t=t*i;s=s+t;}的时间复杂度为_____。

(8)数据结构是研究数据元素之间抽象化的相互关系和这种关系在计算机中的存储结构表示,根据数据元素之间关系的不同特性,通常有4类基本结构:集合、线性结构、_____和_____。

(9)数据常用的存储结构为_____和_____两种。

(10)数据元素都不是孤立的,它们之间总存在某种关系,通常称这种关系为_____。

3)判断题

(1)每种数据结构都应具备三种基本运算:插入、删除和搜索。( )

(2)一个程序的时间复杂度是指该程序运行时间与问题规模的对应关系。( )

(3)数据项是数据的最小单元。( )

(4)数据的基本单位是数据元素。( )

(5)数组元素之间的关系既不是线性的,也不是树形或图形的。( )

(6)算法和程序没有区别,所以在数据结构中二者是通用的。( )

(7)算法的优劣与算法描述语言无关,但与所用计算机有关。( )

4)简答题

(1)描述下列概念。

数据元素,数据,数据类型,数据结构,逻辑结构,物理结构,算法

(2)为什么说逻辑结构是数据组织的主要方面?

(3)举例描述数据结构的逻辑结构、存储结构和操作三方面的内容。

(4)描述算法的5个特性,以及对算法设计的要求。

(5)设n是正整数,求下列程序段中带@记号的语句的执行次数。

①
```
i=1;k=0;
while(i<n)
{k=k+50*i;i++;    @
}
```

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

③
```
x=y=0;
for(i=0;i<n;i++)      @
  for(j=0;j<n;j++)      @
  {x++;                 @
     for(k=0;k<n;k++)@
        y++;            @
  }
```

④
```
x=91;y=100;
while(y>0)
  if(x>100)
  {x=x-10;y--;    @
  }
  else x++;          @
```

(6)说明数据结构与数据类型的区别。

(7)举例说明两个数据结构的逻辑结构和存储方式完全相同,由于对运算的定义不同,所以两个结构具有显著不同的特性,是两个不同的结构。

(8)举例说明对相同的逻辑结构,同一种运算在不同的存储方式下实现,其运算效率不同。

第 2 章 线 性 表

线性结构是最简单和最常用的数据结构，线性表是一种典型的线性数据结构。线性结构有如下基本特点：在数据元素非空的线性表中，数据元素是有序且有限的，同时在线性结构中，有且仅有一个“开始数据元素”无直接前驱，一个“最后数据元素”无直接后继，其余的数据元素有且仅有一个直接前驱和直接后继。

本章学习要点

- 线性表的基本概念和基本运算。
- 线性表的顺序存储结构。
- 线性表的顺序存储结构中初始化、查找、插入、删除等的实现。
- 线性表的链式存储结构。
- 线性表的链式存储结构中初始化、查找、插入、删除等的实现。

2.1 线性表定义

线性表是由 $n(n\geqslant 0)$ 个类型相同的数据元素组成的有限序列。其中数据元素的个数 n 定义为表的长度。当 $n=0$ 时称为空表，非空的线性表($n>0$)一般记为：

$$(x_1,x_2,\cdots,x_{i-1},x_i,x_{i+1},\cdots,x_n)$$

这里的数据元素 $x_i(1\leqslant i\leqslant n)$ 只是一个抽象符号，其具体含义在不同情况下可以不同，但一个线性表中的数据元素必须属于同一数据对象。

从线性表的定义可以看出，它的逻辑特征是：如果 $n>0$，除了有且仅有一个开始数据元素 x_1 无直接前驱和有且仅有一个最后数据元素 x_n 无直接后继外，表中的其余每一个数据元素 $x_i(1<i<n)$ 都有且仅有一个直接前驱 x_{i-1} 和一个直接后继 x_{i+1}。线性表中相邻的数据元素之间存在序偶关系，由于该关系是线性的，因此线性表是一种线性结构。

例如，线性表可以是由英文字母组成的线性表$(A,B,C,\cdots,Z)$，也可以是由几个整数组

成的线性表(10,20,50,70),还可以是由若干个数据项组成的一个数据元素。如表 2-1 所示的学生信息表,该表中的数据元素 x_i 包含了学生的姓名、学号、性别、年龄等 4 个数据项。

表 2-1　学生信息表

姓　　名	学　　号	性　　别	年　　龄
王小凤	2012001	女	18
李芳	2012002	女	19
张斌	2012003	男	19

综上所述,线性表由同类数据元素组成,即每个 x_i 必须属于同一数据对象,线性表由有限个数据元素组成,表的长度就是表中数据元素的个数,线性表中相邻数据元素之间存在序偶关系。

2.2　线性表的抽象数据类型

线性表是一种线性数据结构,因此对线性表中的数据元素可以进行查找、插入、删除等操作。

线性表的抽象数据类型定义如下:

ADT LinearList{

Dataset:D={x_i | $x_i \in D_0$, i=0,1,…,n−1, n≥0, D_0 由 n 个数据元素组成的有限序列}

Relationset:R={<x_{i-1}, x_i> | x_{i-1}, $x_i \in D_0$, i=1,3,…,n−1}

OperationSet:

(1)InitList(L)

初始条件:L 为未初始化线性表。

操作结果:构造一个空的线性表,即将 L 初始化为空表。

(2)DestroyList(L)

初始条件:线性表 L 已存在。

操作结果:将 L 删除。

(3)ClearList(L)

初始条件:线性表 L 已存在。

操作结果:将表 L 置为空表。

(4)EmptyList(L)

初始条件:线性表 L 已存在。

操作结果:如果 L 为空表则返回真,否则返回假。

(5)ListLength(L)

初始条件:线性表 L 已存在。

操作结果:如果 L 为空表则返回 0,否则返回表中的元素个数。

(6)Locate(L,y)

初始条件:表 L 已存在,y 为合法元素值。

操作结果：如果L中存在元素y，查找成功则返回其位置，否则返回－1。

(7)GetData(L,i)

初始条件：表L已存在，且i值合法，即1≤i≤ListLength(L)。

操作结果：返回线性表L中第i个元素的值。

(8)InsList(L,i,y)

初始条件：表L已存在，y为合法元素值且1≤i≤ListLength(L)＋1。

操作结果：在L中第i个位置插入新的数据元素y，L的长度加1。

(9)DelList(L,i,&y)

初始条件：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y返回其值，L的长度减1。

}ADT LinearList

线性表的抽象类型定义中给出的各种操作定义是最基本的操作，此外对线性表还可以进行合并、分解等复杂的操作。

2.3 线性表的顺序存储结构

在计算机内存储线性表，主要有顺序存储结构和链式存储结构两种。

2.3.1 顺序存储定义

线性表的顺序存储结构，就是用一组连续的存储单元依次存放线性表中的数据元素，即线性表中逻辑结构上相邻的数据元素在物理结构上也是相邻的，用这种方式存储的线性表简称为顺序表。

由于线性表中每个数据元素的类型是相同的，因此每个数据元素所占用的存储空间的大小也是相同的。在顺序表中，每个数据元素 x_i 的存储地址是该数据元素在表中的位置的线性函数，只要知道线性表中第一个元素的存储地址和每个数据元素的大小，就可以计算出表中任意一个数据元素的存储地址。因此顺序表是一种随机存取结构。

假设顺序表中有 n 个数据元素，每个数据元素占用 c 个存储单元，设第一个数据元素 x_1 的存储地址(称为基地址)为 $\mathrm{Loc}(x_1)$，那么第 i 个数据元素的存储地址 $\mathrm{Loc}(x_i)$ 可以通过下式计算：

$$\mathrm{Loc}(x_i)=\mathrm{Loc}(x_1)+(i-1)*c$$

线性表的顺序存储结构可以用高级语言中的一维数组来表示，如用C语言的一维数组来实现。C语言的一维数组既可以是静态分配的，也可以是动态分配的。在C语言中，只要定义了一个数组，就定义了一块可供用户使用的存储空间，该存储空间的起始位置就是用数组名表示的地址常量。数组的数据类型就是顺序表中每个数据元素的数据类型，数组的大小要大于等于顺序表的长度。顺序表中的第一个数据元素被存储在数组的起始位置，即下标为0的位置上，第二个数据元素被存储在下标为1的位置上，依次类推，第 n 个数据元素被存储在下标为 $n-1$ 的位置上。

顺序存储结构在内存储中的状态如图 2-1 所示。

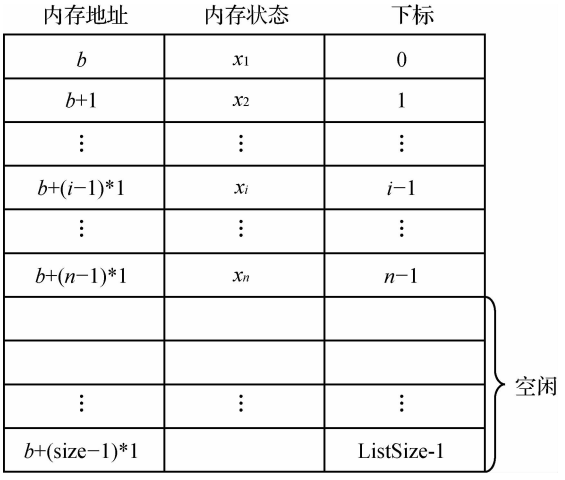


图 2-1　顺序存储结构在内存储中的状态

顺序表静态存储结构的定义如下：

```
#define Max_Size 100              /*线性表可能达到的最大长度*/
typedef int DataType;
typedef struct List{
    DataType data[Max_Size];      /*用于存放数据元素数组*/
    int length;                   /*当前表长度*/
}SeqList;
```

由于顺序表的静态数组存储结构预先设定了存储单元大小，在实际应用中有局限性，使用起来不是很灵活，如果预先定义过大则浪费存储空间，预先定义过小则会造成存储空间不足。由于线性表经常用于插入和删除操作，长度可变，所以存储空间可以用 C 语言中的动态数组分配。

顺序表动态存储结构的定义如下：

```
#define add_Size 2
typedef int DataType;
typedef struct List
{
    DataType *data;       /*指向存储空间基地址*/
    int size;             /*设置当前表长*/
    int Max_Size;         /*线性表可能达到的最大长度，当前分配的存储容量*/
}SeqList;
```

其中，data 是一个指针变量，是数组名；size 表示线性表的当前长度，必须满足 size≤Max_Size。如果线性表不空，则按照 size 分配存储空间，否则 size=0。

动态数组分配可以根据实际需要进行扩展，下面为顺序表扩展空间的定义。

```
void againMalloc(struct List *L){
    /*空间扩展为原来的 add_Size 倍，并由 p 指针指向，原内容被自动复制到 p 所指
```

```
向的存储空间*/
        DataType*p=(DataType*)realloc(L->data,add_Size*L->Max_Size*
sizeof(DataType));
        if(!p){
            /*分配失败则退出运行*/
            printf("存储空间分配失败!");
                exit(1);  /*执行此函数中止程序运行,此函数在stdlib.h中有定义*/
        }
        L->data=p;      /*使list指向新线性表空间*/
        L->Max_Size=add_Size*L->Max_Size;     /*把线性表空间大小修改为新
的长度*/
        return;
    }
```

2.3.2 顺序存储基本操作

在线性表的抽象类型定义中,介绍了线性表的基本操作,这里将根据顺序表的存储特点,对顺序表的基本运算进行具体定义。

1)初始化

初始化线性表L,在静态状态下初始化线性表。

```
void InitList(SeqList &L){
    L.length=0;
}
```

初始化线性表L,即进行动态存储空间分配并置L为一个空表(即表的当前长度为0)。

```
void initList(struct List*L,int ml){
    /*检查ml是否有效,若无效则退出运行*/
    if(ml<=0){
        printf("MaxSize非法!");
        exit(1);     /*执行此函数中止程序运行,此函数在stdlib.h中有定义*/
    }
    L->Max_Size=ml;     /*设置线性表空间大小为ml*/
    L->size=0;
    L->data=(DataType*)malloc(ml*sizeof(DataType));
    if(!L->data){
        printf("空间分配失败!");
        exit(1);  /*执行此函数中止程序运行,此函数在stdlib.h中有定义*/
    }
    return;
}
```

2)查找操作

线性表的查找有按位置查找和按内容查找两种。

(1)按位置查找:即在线性表L中查找第i个位置上的数据元素。

```
/*查找成功返回线性表L中第i个元素的值,若i超出范围,则停止程序运行*/
int GetData(struct List*L,int i)
{
    if(i<1 || i>L->size){       /*若i越界则退出运行*/
        printf("元素序号越界!");
        exit(1);
    }
    return L->data[i-1];        /*返回线性表中序号为i值的元素值*/
}
```

(2)按内容查找:即在线性表L中查找与给定值y相等的数据元素,如果查找成功,就返回该元素在表中的位置,否则返回-1。如下面的代码:

```
/*静态存储结构下进行查找操作*/
DataType Locate(SeqList L,DataType y){
  int i=0;
  while((i<=L.length)&&(L.data[i]!=y))  /*顺序搜索表中的数据元素,直到找到值等于y数据元素,或搜索到表尾而没找到为止*/
      i++;
  if(i>L.length)
      return(-1);
  else
      return(i+1);  /*数组的下标是从0开始,线性表的位置是从1开始*/
}
/*动态存储结构下进行查找操作*/
DataType Locate(struct List*L,DataType y){
/*按指定值y查找,如果找到返回该值位置,否则返回-1*/
    int i;
    for(i=0;i<L->size;i++){
        if(L->data[i]==y){
            return(i+1);  /*数组的下标是从0开始,线性表的位置是从1开始*/
        }
    }
    return -1;
}
```

在顺序表的查找操作中,从表的开始位置起,根据给定值y,逐个与表中各数据元素的值进行比较。若给定值与某个数据元素的值相等,则算法搜索成功并返回该值在表中的位置;若查遍表中所有的数据元素,没有任何一个数据元素满足要求,则算法提示搜索不成功的信息并返回-1。

查找算法的时间复杂度用数据比较次数来衡量。在查找成功的情形下，若要找的正好是表中第1个数据元素，则数据比较次数为1，这是最好情况；若要找的是表中的第 n 个数据元素，则数据比较次数为 n（设表的长度为 n），这是最坏的情况。若要计算平均数据比较次数，则需要考虑各个数据元素的查找概率 p_i 及找到该数据元素时的数据比较次数 c_i。查找的平均数据比较次数 O_{locate} 为：

$$O_{\text{locate}}=\sum_{i=1}^{n} p_i \times c_i$$

计算平均值是为了解算法对线性表操作的整体性能。若仅考虑相等概率的情形，查找各数据元素的可能性相同，有 $p_1=p_2=\cdots p_n=\frac{1}{n}$，且查找第1个数据元素的数据比较次数为1，查找第2个数据元素的数据比较次数为2，……，查找第 i 个数据元素的数据比较次数为 i，则有：

$$O_{\text{locate}}=\frac{1}{n}\sum_{i=1}^{n} i=\frac{1}{n}(1+2+\cdots+n)=\frac{1}{n}\times\frac{(1+n)\times n}{2}=\frac{1+n}{2}$$

即平均要比较 $\frac{n+1}{2}$ 个数据元素，那么平均时间复杂度为 $O_{\text{locate}}=(\frac{n+1}{2})$。

在查找不成功的情形下，需要把整个表全部检测一遍，数据比较次数达到 n 次。需要注意的是，在算法中担负检测的循环执行了 $n+1$ 次，最后一次仅检测了指针 i 已超出表的长度，没有执行数据比较，算法最坏的时间复杂度为 $O_{\text{locate}}(n)$，最好时间复杂度为 $O_{\text{locate}}(1)$。

3）插入操作

线性表的插入操作是指在线性表中的第 $i(1\leqslant i\leqslant n)$ 个位置上插入新的数据元素 y，使长度为 n 的线性表：

$$\mathrm{L}=(x_1,x_2,\cdots,x_{i-1},x_i,x_{i+1},\cdots,x_n)$$

变为长度为 $n+1$ 的线性表：

$$\mathrm{L}=(x_1,x_2,\cdots,x_{i-1},y,x_{i+1},\cdots,x_n,x_{n+1})$$

图2-2表示一个顺序表用数组表示，在进行插入操作时数据元素的位置变化前和变化后的状况。例如，为了在数组中第5个和第6个序号元素之间插入一个值为22的元素，则需要将序号为6及之后的所有的数据元素依次向后移动一个位置。

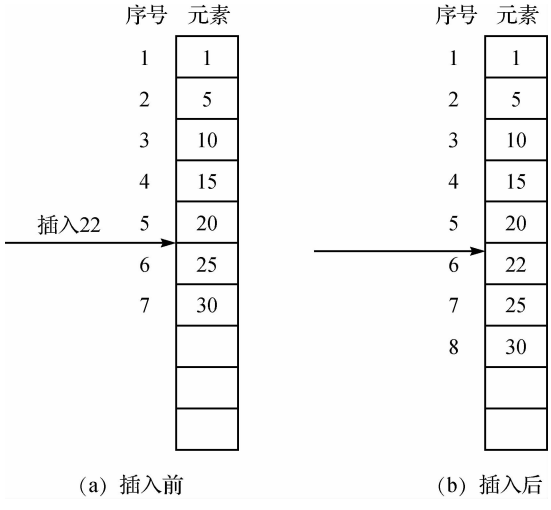


图2-2 线性表插入前后状态

在顺序表中第 i 个位置上进行插入操作时,首先判断所要插入的数据元素的位置是否越界,如果越界则插入失败;否则只需要将线性表中的第 i 个至第 n 个数据元素后移一个位置,将数据元素插入到空出 i 的位置上,再将线性表长度加 1,如果 $i=n+1$,则是在表尾插入,无需移动数据元素,而是直接将数据元素插入到空出 i 的位置上。

```
/*静态数组存储结构下进行插入操作*/
void InsList(SeqList &L,int i,DataType y)
{
    int j;
    if(i<1||i>L.length+1)  /*若i越界则插入失败*/
        exit(1);    /*执行此函数中止程序运行,此函数在stdlib.h中有定义*/
    return;
    for(j=L.length+1;j>=i;j--)
        L.data[j]=L.data[j-1];
    L.data[i-1]=y;
    L.length++;
    return;
}
/*动态数组存储结构下进行插入操作*/
void InsList(struct List *L,int i,DataType y)
{
    int k;
    if(i<1 || i>L->size+1)
    {
        /*若i越界则插入失败*/
        exit(1);  /*执行此函数中止程序运行,此函数在stdlib.h中有定义*/
    }
    if(L->size==L->Max_Size)
    {
        /*重新分配更大的存储空间*/
        againMalloc(L);
    }
    for(k=L->size-1;k>=i-1;k--)
    {
        L->data[k+1]=L->data[k];
    }
    L->data[i-1]=y;
    L->size++;
    return;
}
```

在顺序表中插入一个新的数据元素时，表的长度(设为 n)会发生改变。为了把新数据元素 y 插入到指定位置 i，必须把从 i 到 n 的所有数据元素向后移动一个数据元素的位置，以空出第 i 个位置供 y 插入，在将新数据元素插入到第 i 个数据元素($0 \leqslant i \leqslant n$)后面时，必须从后向前循环，逐个向后移动 $n-i$ 个数据元素。因此，最好的情形是在第 n 个数据元素后面追加新的数据元素，移动数据元素个数为 0；最坏的情形是在第 1 个数据元素位置插入新数据元素，移动数据元素个数为 n；平均数据移动次数 O_{insert} 在各数据元素插入概率相等时为：

$$O_{\text{insert}} = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}(n+\cdots+1+0) = \frac{1}{n+1}\times\frac{n(n+1)}{2} = \frac{n}{2}$$

即从整体性能来说，在插入时有 $n+1$ 个插入位置，平均移动 $n/2$ 个数据元素，那么平均时间复杂度为 $O_{\text{insert}}(\frac{n}{2})$。最坏时间复杂度为 $O_{\text{insert}}(n)$，最好时间复杂度为 $O_{\text{insert}}(1)$。

4)删除操作

线性表的删除操作是指将表中的第 $i(1 \leqslant i \leqslant n)$个元素删除，使长度为 n 的线性表：

$$L=(x_1, x_2 \cdots x_{i-1}, x_i, x_{i+1}, \cdots, x_n)$$

变为长度为 $n-1$ 的线性表：

$$L=(x_1, x_2 \cdots x_{i-1}, x_i, x_{i+1}, \cdots, x_{n-1})$$

如图 2-3 所示，表示一个顺序表用数组表示时，在进行删除操作时数据元素的位置变化前和变化后状况。例如，为了在数组中删除序号为 6(值为 25)的元素，则需要将序号为 7 及之后的所有数据元素依次向前移动一个位置。

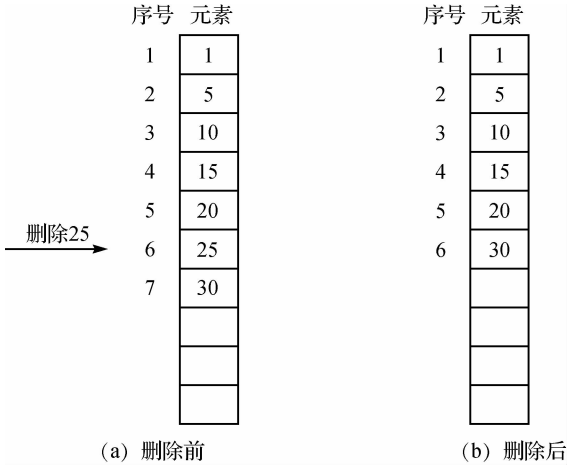


图 2-3 线性表删除前后的状态

在顺序表中删除指定位置 i 上的数据元素时，首先要判断所要删除的数据元素的位置是否越界，如果越界则删除失败，否则把第 $i+1$ 及之后的所有数据元素向前移动一个位置，将顺序表长度减 1，并返回被删除元素的值。

```
/*静态数组存储结构下进行删除操作*/
DataType DelList(SeqList &L,int i,DataType &y)
{
    int j;
```

```
    y=L.data[i-1];  /*因为数组的下标从0开始,线性表的位置是从1开始*/
    if(i<1||i>L.length)  /*若i越界则删除失败*/
        exit(1);
    for(j=i;j<L.length;j++)
        L.data[j-1]=L.data[j];
    L.length--;
    return y;
}
/*动态数组存储结构下进行删除操作*/
DataType DelList(struct List *L,int i,DataType &y)
{
    int k;
    if(i<1 || i>L->size)
    {
        /*i越界则删除失败*/
        printf("i值越界,不能进行删除操作!");
        exit(1);
    }
    y=L->data[i-1];
    for(k=i;k<L->size;k++)
        L->data[k-1]=L->data[k];
    L->size--;
    return y;
}
```

上述算法实现了删除线性表中指定位置的数据元素,并返回该数据元素的值。算法首先判断所要删除的指定线性表中数据元素的位置是否越界,如果越界,就表示删除失败;否则将线性表中的第 $i+1$ 至第 n 个数据元素前移一个位置,再将线性表长度减 1。

在顺序表中删除一个数据元素时,表的长度(设为 n)会发生改变。为了把表中的第 $i(1\leqslant i\leqslant n)$ 个指定位置的数据元素删除,必须把从 i 到 n 的所有数据元素向前移动一个数据元素的位置,在删除第 i 个数据元素($1\leqslant i\leqslant n$)时,必须从前向后循环,逐个移动 $n-i$ 个数据元素。因此,最好的情形是删去最后的第 n 个数据元素,移动数据元素个数为 0;最差情形是删去第 1 个数据元素,移动数据元素个数为 $n-1$;平均数据移动次数 O_{delete} 在各数据元素删除概率相等时为:

$$O_{\text{delete}}=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{1}{n}(n+\cdots+1+0)=\frac{1}{n}\times\frac{n(n-1)}{2}=\frac{(n-1)}{2}$$

就整体性能来说,在删除时有 n 个删除位置,平均移动 $(n-1)/2$ 个数据元素。最坏时间复杂度为 $O_{\text{delete}}(n)$,最好时间复杂度为 $O_{\text{delete}}(1)$。

2.4 线性表的链式存储结构

线性表的顺序存储结构要求占用连续的空间，其物理结构和逻辑结构保持一致，这一特点使得顺序表可以随机存取表中的任一数据元素，存取速度快，但也使得插入和删除操作要移动大量的数据元素。为了克服顺序表的缺点，可以采用链式存储结构存储线性表。链式存储结构是最常用的存储方式之一，用链式存储的线性表简称为线性链表。链表适用于插入或删除频繁，存储空间需求不定的情形。

2.4.1 链式存储定义

链式存储结构是用一组任意存储单元存储结点（该存储单元可以是连续的，也可以是不连续的），每个存储结点不仅包含所存元素本身的信息，还包含元素之间的逻辑关系，因此每个结点包括值域和指针域，其中存储数据元素值信息的域称为值域，存储结点直接后继或直接前驱位置的域称为指针域，指针域中存储的信息称为指针或链。相邻结点间用指针链接，通过结点的指针域可以访问到对应的后继或前驱结点，用链式存储结构表示的线性表称为链表。根据结点构造方法不同，链表主要有单链表、循环链表、双链表等。

2.4.2 单链表及其基本操作

单链表是指每个结点只由值域和一个指针域组成，指针域用来存储数据元素直接后继的地址（或位置）。通过每个结点的指针域将线性表的 $n(1\leqslant i\leqslant n)$ 个结点按其逻辑顺序链接在一起的表称为链表 $L=(x_1,x_2,\cdots,x_{i-1},x_i,x_{i+1},\cdots,x_n)$。如果链表的每个结点只有一个指针域，则将这种链表称为单链表。

单链表是一种最简单的链表，也叫做线性链表。用它来表示线性表时，用指针表示结点间的逻辑关系。因此单链表的一个存储结点（node）包含两个域 data 和 next，如图 2-4 所示。其中，data 称为数据域，用于存储线性表的一个数据元素，其数据类型由应用问题决定；next 称为指针域或链域，用于存放一个指针，该指针指示该链表中下一个结点的开始存储地址。

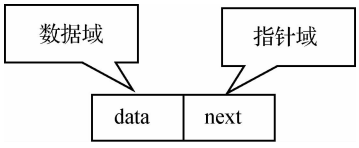


图 2-4 单链表结点结构图

单链表的存储结构定义如下：

```
typedef int DataType;
typedef struct LNode
```

```
{
    DataType data;        /*存储结点值*/
    LNode *next;          /*链表下一结点的地址*/
}LNode, *LinkList
```

例如,如图 2-5 所示的线性表(Cyan,Red,Green,Blue,Orange,Violet,Yellow)的链式存储结构。

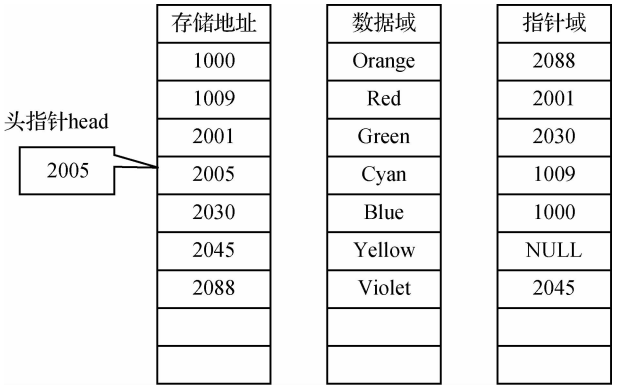


图 2-5　线性表的链式存储结构

单链表中每个结点的指针域存储的是后继结点的地址,由于线性表中的第一个结点无直接前驱,因此设置一个指针 head 指向第一个结点,指向链表第一个结点的指针称为头指针,即 head。由于最后一个结点无直接后继,因此最后一个结点指针域为"空"(NULL)。

一个线性表 $L=(x_1,x_2,\cdots,x_{i-1},x_i,x_{i+1},\cdots,x_n)$的单链表逻辑状态表示如图 2-6 所示。

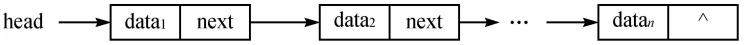


图 2-6　无头结点单链表的结构

单链表的头指针 head 是整个链表的开始,给定单链表的头指针即可访问整个链表,因此对单链表中任一结点的访问必须从头指针开始,顺着结点指针域依次进行,直到找到所需的结点。

为了便于操作,通常在第一个结点之前附设一个结点,这个结点称为头结点,头结点的数据域可以不存储任何信息,也可以存储线性表长度等附加信息。头结点指针域存储第一个结点的存储位置,头指针指向头结点。如图 2-7 所示为带头结点的单链表。如果线性表为空,则头结点指针域为空,如图 2-8 所示。

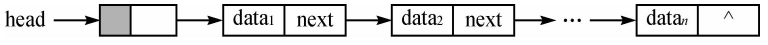


图 2-7　带头结点的单链表

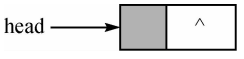


图 2-8　带头结点的空单链表

1)初始化单链表

初始化单链表 L,置单链表 L 为空,算法实现如下:

```
void initList(LinkList L)
{
    L=(LNode *)malloc(sizeof(LNode));        /*建立头结点*/
    L->next=NULL;                            /*置单链表L为空*/
}
```

说明:L 是单链表表头指针,申请一个结点,把表头指针指向该结点,即头指针指向头结点,并置链表为空。

2)单链表的创建

用带头结点的头插法建立单链表。算法思想:生成一个新结点,作为空链表的头结点,读入数据,当读入的数据大于 0 时,再生成一个新的结点,将读入的数据存放到新结点的数据域中,然后将新结点链接到当前链表的表头结点之后,直到读入数据小于等 0 时结束,建表过程如图 2-9 所示。

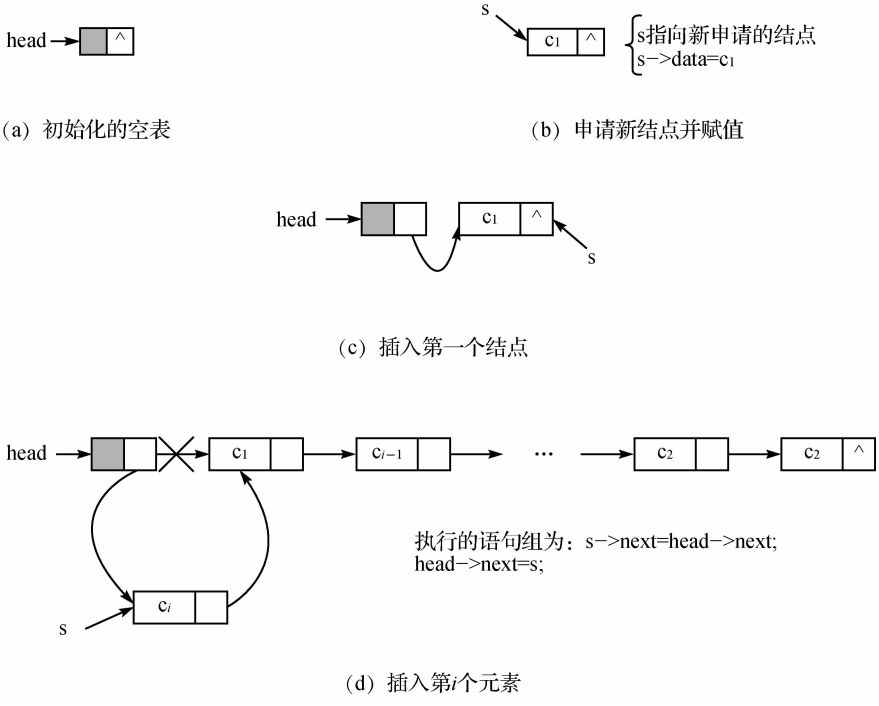


图 2-9 用带头结点的头插法建立单链表

用带头结点的头插法建立单链建表算法如下:

```
LNode *creatList(void)
{
    /*用带头结点的头插法创建单链表*/
    LNode *head;                             /*头指针*/
    LNode *s;                                /*新结点指针*/
```

```
    LNode * p;                                  /* 工作指针 */
    int c;
    head=(LNode * )malloc(sizeof(LNode));       /* 申请新结点 */
    p=head;
    p->next=NULL;
    printf("请输入链表结点数据,当输入值<=0 时建表完成,该值不接入链表\n");
    scanf(" %d",&c);
    while(c>0)      /* 输入的值大于 0 则继续,直到输入的值<=0 */
    {
        s=(LNode * )malloc(sizeof(LNode));   /* 再重申请一个结点 */
        s->data=c;
        s->next=p->next;
        p->next=s;
        scanf(" %d",&c);
    }
    printf("creatList 函数执行,创建链表成功\n");
    return head;                                /* 返回链表的头指针 */
}
```

用带头结点的尾插法建立单链表。算法思想:生成一个新结点,作为空链表的头结点,读入数据,当读入的数据大于 0 时,再生成一个新的结点,将读入的数据存放到新结点的数据域中,然后将新结点链接到当前链表的表尾,直到读入的数据小于等于 0 时结束,建表过程如图 2-10 所示。

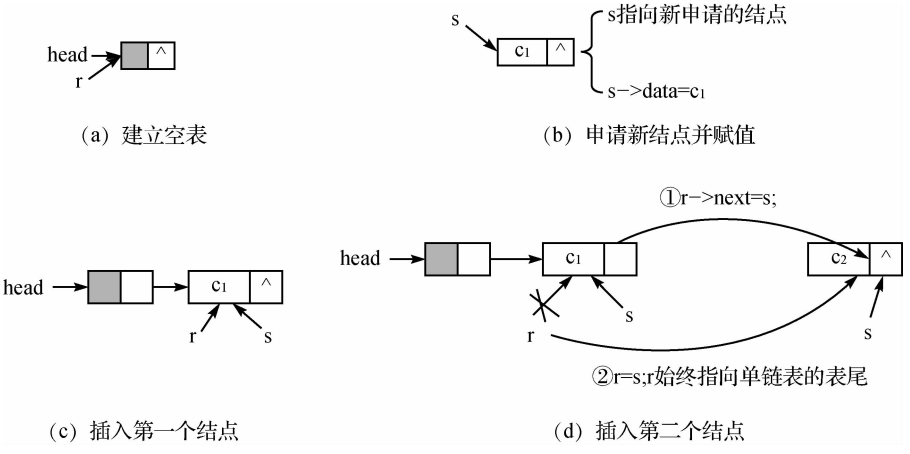


图 2-10　用带头结点的尾插法建立单链表

用带头结点的尾插法建立单链建表算法如下:

```
LNode * creatList(void)
{
    /* 用带头结点的尾插法建立单链表 */
```

```
    LNode * head;
    LNode * s;
    LNode * p;
    int c=1;;
    head=(LNode * )malloc(sizeof(LNode));        /* 申请新结点 */
    p=head;
    printf("请输入链表结点数据,当输入值<=0 时建表完成,该值不接入链表\n");
    while(c>0)  /* 输入的值大于 0 则继续,直到输入的值为负 */
    {
        s=(LNode * )malloc(sizeof(LNode));       /* 再重申请一个结点 */
        scanf(" %d",&s->data);
        c=s->data;
        p->next=s;
        p=s;
        p->next=NULL;
    }
    printf("creatList 函数执行,创建链表成功\n");
    return head;                                 /* 返回链表的头指针 */
}
```

3)单链表的查找

在单链表中,由于每个结点的存储位置都存放在该结点的前一个结点的 next 域中,因此即使知道被访问结点的位置,也要从链表的头指针出发逐个搜索,直到找到该结点为止。具体程序代码如下。

```
/* 在带头结点的单链表中,查找第 i(1≤i≤n)个结点,如果找到,就返回该结点的存储位置,否则返回 NULL */
LNode * Locate(LinkList L,int i)
{
    LNode * head;
    head=L;
    int count=0;
    if(i<=0|)
    {
        printf("Locate 函数执行,i 值非法\n");
        return NULL;
    }
    while(head->next!=NULL)
    {
        if(count<i)
        {
            ++count;
```

```
            head=head->next;
        }
        else break;
    }
    if(i==count){
        printf("\n 函数执行,链表中的第 %d 个元素的值是 %d count= %d",i,head
->data,count);
        printf("该元素在链表中的存储地址是 0x%x\n",&(head->data));
        return head;
    }
    else return NULL;
}
/*在带头结点的单链表中,查找指定值等于 key 的结点,如果找到,就返回该值位置,
否则返回 NULL,查找过程从链表的头指针出发逐个搜索,直到找到或到表尾为止*/
LNode*LocateKey(LinkList L,DataType key)
{
    LNode*head;
    head=L;
    int count=0;
    if(NULL==head)
    {
        printf("LocateKey 函数执行,链表为空\n");
        return NULL;
    }
    while(head->next!=NULL)
    {
        if(head->data!=key)
        {
            ++count;
            head=head->next;
        }
        else break;
    }
    if(head->data==key){
        printf("\n 函数执行,元素 %d 是链表中的第 %d 个元素",key,count);
        printf("该元素在链表中的存储地址是 0x%x\n",&(head->data));
    }
    else printf("链表中没有找到元素 %d",key);
    return head;
```

}

4)单链表的插入

在带头结点单链表 L 的第 i 个位置中插入一个值为 key 的新结点,首先需要在链表中找到第 $i-1$ 个结点并由指针 p 指示,然后申请一个新的结点 s,使 s->data=key,并修改指针"s->next=p->next;p->next=s;",插入过程如图 2-11 所示。

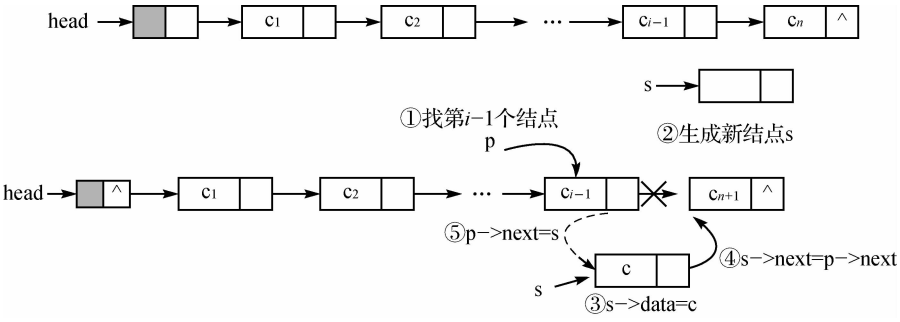


图 2-11 在链表的第 i 个结点位置插入结点 s 的情形

```
LinkList insertheadList(LinkList L,int i,DataType key)
{
    LNode * p=L, * s;
    int j=1;
    if(i<=0)
    {
        printf("insertheadList 函数执行,i 值非法,插入失败\n");
        return NULL;
    }
    while((p->next!=NULL)&&(j<i))
    {
        if(j<i)
        {
            p=p->next;
            j++;
        }
        else break;
    }
    if(j!=i)
    {
        printf("\n insertheadList 函数执行,i 值超过链表长度,插入失败\n");
        return NULL;
    }
    else
```

```
        {
            s=(LNode*)malloc(sizeof(LNode));  /*再重申请一个结点*/
            s->data=key;
            s->next=p->next;
            p->next=s;
            printf("\n insertheadList 函数执行,在第%d个位置插入%d,插入成功\n",
i,key);
        }
        return L;
    }
```

5)单链表的删除

在带头结点单链表 L 中删除第 i 个结点,首先需要在链表中找到第 $i-1$ 个结点并由指针 p 指示,然后修改指针,“q=p->next;p->next=q->next;free(q);”,过程如图 2-12 所示。

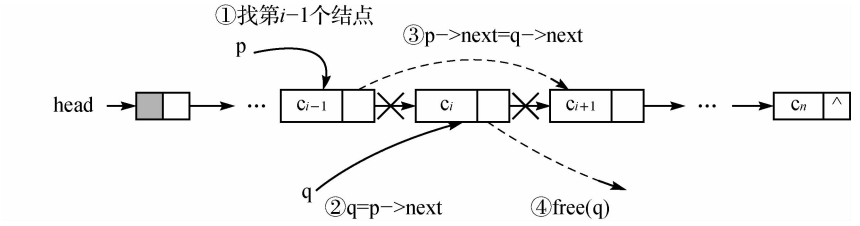


图 2-12　在单链表上删除第 i 个结点

在带头结点单链表 L 中删除第 i 个结点的算法如下:

```
LNode*DelLink(LinkList L,int i)  /*删除指定位置的结点*/
{
    LNode*p,*q;
    int j=1;
    p=L;
    if(i<1)
    {
        printf("删除结点位置不合法,删除失败\n");
        exit(1);
    }
    while(p->next!=NULL&&j<i)
    {
        p=p->next;
        j++;
    }
    if(j!=i)
```

```
    {
        printf("DelLink 函数执行,i 值超过链表长度,删除失败\n");
        return L;
    }
    else
    {
        q=p->next;
        p->next=q->next;
        printf("删除第 %d 个结点成功,删除的结点值是 %d\n",i,q->data);
        free(q);
    }
    return L;
}
```

单链表的特点是可以很方便地对长度进行扩充。当链表要增加一个新的结点时,只要可用存储空间允许,就可以为链表分配一个结点空间,供链表使用。因此,线性表中数据元素的顺序与其链表表示中结点的物理顺序可能不一致,一般通过单链表的指针将各个数据元素按照线性表的逻辑顺序链接起来。

在线性表的顺序存储中,逻辑上相邻的元素,其对应的存储位置也相邻,所以当进行插入或删除操作时,通常需要平均移动半个表的元素,这是相当费时的。在线性表的基于链表的存储表示中,逻辑上相邻的元素,其对应的存储位置是通过指针来链接的,因而可以任意安排每个结点的存储位置,不必要求它们相邻,当进行插入或删除操作时,只需修改相关结点的指针域即可,这是既方便又省时的操作。由于链表的每个结点带有指针域,因而在存储空间上比顺序存储要付出更大的代价。

2.4.3 静态链表

用一维数组描述的链表,称为静态链表,即为数组中每一个元素附加一个链接指针,就形成了静态链表结构。它允许不改变各元素的物理位置,只要重新链接就能够改变这些元素的逻辑顺序。由于是利用数组定义的,因此静态链表也必须预先分配存储空间的大小,且不能改变,因此称之为静态链表。

静态链表数据结构的定义如下:

```
typedef int DataType;
#define MAXSIZE 100
typedef struct
{
    DataType data;
    int cur;
}component,SLinkList[MAXSIZE];
```

静态链表的每个结点由两个数据成员构成:data 相当于数据域,用来存放数据;游标 cur

相当于指针域,用来存放其后继结点在数组中的相对位置,也就是数组的下标,所有结点形成一个结点数组,可将数组的第一个分量(C语言中下标为0的分量)看成头结点,其游标域指示链表的第一个结点。

静态链表中要解决的一个问题就是如何用静态链表结构模拟动态链表结构的存储空间的分配,需要时申请,不需要时释放。在动态链表中,结点的申请和释放分别借用malloc()和free()两个系统函数来实现。在静态链表中,如果要模拟动态链表中结点的申请和释放,可以通过标记数组中已被使用的分量和未被使用的分量来实现,将所有未被使用过的分量以及被删除的分量用游标链成一个链表,并附设头指针指示,申请结点时,从链表上"解下"一个结点,释放结点时,将释放后的结点链回到备用链表上。由于该链表的作用好像一个结点空间的储备仓库一样,因此称其为备用静态链表。

静态链表在进行插入和删除时不需要移动元素,仅需要修改游标cur,因此具有链式存储结构的主要优点。

1)静态链表初始化

初始化静态链表L,是指将静态链表初始化为一个备用静态链表,算法实现如下:

```
void InitList(SLinkList L)
{
    /*构造一个空的链表L,表头为L的最后一个单元L[MAX_SIZE_1],其他单元链成一个备用链表,表头为L的第一个单元L[0],"0"表示空指针*/
    int i;
    L[MAX_SIZE-1].cur=0;  /*L的最后一个单元为空链表的表头*/
    for(i=0;i<MAX_SIZE-2;i++)  /*将其他单元链接成以L[0]为表头的备用链表*/
        L[i].cur=i+1;
    L[MAX_SIZE-2].cur=0;
}
```

2)分配空间

为静态链表分配空间,是指从备用链表摘下一个结点空间,分配给一个待插入的静态链表中的一个数据元素,算法的实现如下:

```
int Malloc(SLinkList space)     /*若备用链表不空,则返回分配的结点下标*/
{
    int i=space[0].cur;         /*备用链表第1个结点的位置*/
    if(i)                       /*备用链表不空*/
        space[0].cur=space[i].cur;  /*备用链表的头结点指向原备用链表的第2个结点*/
    return i;
}
```

3)释放空间

将静态链表中的空闲结点回收到备用链表中,算法实现如下:

```
void Free(SLinkList space,int k)
{
    /*将下标为k的空闲结点回收到备用链表中*/
    space[k].cur=space[0].cur;  /*回收结点的"游标"指向备用链表的第1个结点*/
    space[0].cur=k;  /*备用链表的头结点指向新回收的结点*/
}
```

4)插入

在静态链表L的第i个元素之前,插入数据元素e的算法实现如下:

```
DataType ListInsert(SLinkList L,int i,DataType e)
{
    /*在L中第i个元素之前插入数据元素e*/
    int m,j,k=MAX_SIZE-1;       /*k指示表头结点的位序*/
    if(i<1||i>ListLength(L)+1)
        return 0;
    j=Malloc(L);                /*申请新单元*/
    if(j)                       /*申请成功*/
    {
        L[j].data=e;
        for(m=1;m<i;m++)  /*k向后移i-1个结点,使k指示第i-1个结点*/
            k=L[k].cur;         /*指向下一个结点*/
            L[j].cur=L[k].cur;  /*新结点指向第i-1个元素后的元素*/
            L[k].cur=j;         /*第i-1个元素指向新单元*/
        return 1;
    }
    return 0;
}
```

5)删除

删除静态链表L中的第i个元素,并返回其值的算法实现如下:

```
DataType ListDelete(SLinkList L,int i,DataType &e)
{
    /*删除L中第i个元素,并返回其值*/
    int j,k=MAX_SIZE-1;     /*k指示表头结点的位置*/
    if(i<1||i>ListLength(L))
        return 0;
    for(j=1;j<i;j++)        /*移动i-1个元素,使k指向第i-1个元素*/
        k=L[k].cur;         /*指向下一个元素*/
```

```
    j=L[k].cur;                /*待删除元素*/
    L[k].cur=L[j].cur;         /*使第 i-1 个元素指向待删除元素的后继元素*/
    e=L[j].data;
    Free(L,j);
    return 1;
}
```

2.4.4 循环链表

循环链表是线性表的另一种形式的表示,它是一个首尾相接的链表,它的结点结构与单链表相同。与单链表不同的是,链表中表尾结点的 next 域中不是 NULL,而是存放了一个指向链表开始结点的指针。因此,在循环链表中,只要知道表中任何一个结点的地址,就能遍历表中其他任一结点。图 2-13 是单循环链表的一个示意图,图 2-13(a)是空的单循环链表,图 2-13(b)是带头结点的单循环链表,图 2-13(c)是尾指针单循环链表。

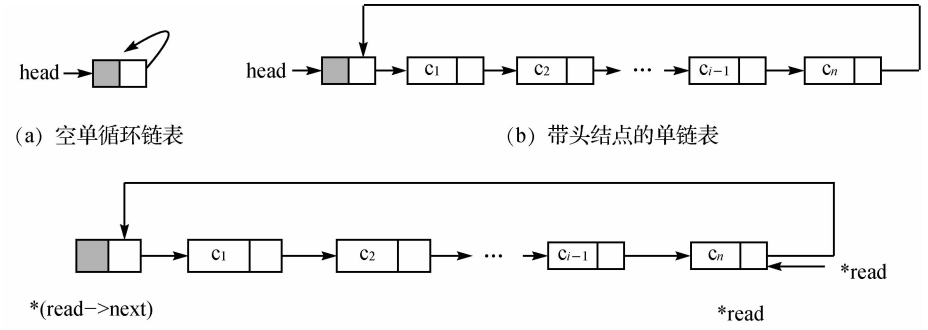


图 2-13 单循环链表

带头结点的单循环链表的各种操作的实现算法与带头结点单链表的实现算法类似,所不同的仅在于循环链表中逐个结点检测的指针,在判断 p 是否到达链表的链尾时,不是判断 p->next!=NULL 或 p!=NULL 是否成立,而是判断 head!=p->next 或 head!=p 是否成立。

在循环链表中涉及链头与链尾处理时稍有不同,如在循环链表中进行查找操作时,如果查找的是终端结点,就要从头指针开始搜索整个链表,但如果用尾指针来表示循环链表,那么搜索开始结点和终端结点都很方便,它们存储的位置分别是 read->next->next 和 read。例如,在实现循环链表的插入操作时,如果是在表的最前端插入,就必须改变链尾最后一个结点的 next 域的值,这就需要循链搜索找到最后一个结点。而如果用链尾的 rear 指针,那么在实现插入或删除操作时就会更方便,如图 2-14 所示。关键操作为:

```
s->next=rear->next;  rear->next=s;
```

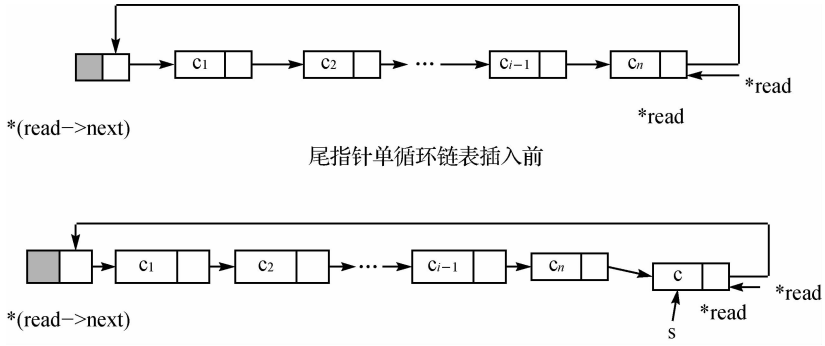


图 2-14 尾指针循环链表的最前端或尾端插入的情形

2.4.5 双向链表

双向链表又称为双链表。使用双向链表的目的是解决在链表中访问直接前驱和直接后继的问题。因为在双向链表中每个结点都有两个链指针，一个指向结点的直接前驱，一个指向结点的直接后继，这样不论是向前驱方向搜索还是向后继方向搜索，其时间开销都只有 $O(1)$。

在双向链表的每个结点中应有数据域和两个指针域：prior 指示它的前驱结点，next 指示它的后继结点，这样形成的链表中有两条方向不同的链，称为双向链表。因此，双向链表的每个结点有 3 个域，如图 2-15 所示。

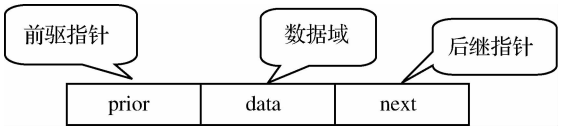


图 2-15 双向链表结点结构

双向链表存储结构定义如下：

```
typedef int DataType;
typedef struct LNode
{
    DataType data;        /*存储结点值*/
    LNode *prior;         /*前驱结点地址*/
    LNode *next;          /*后继结点地址*/
}Lnode, *LinkList
```

双向链表也是由头指针唯一确定的，增加头结点也能使双向链表的操作更方便，同时双向链表也可以有循环表，称为双向循环链表，其结构如图 2-16 所示。

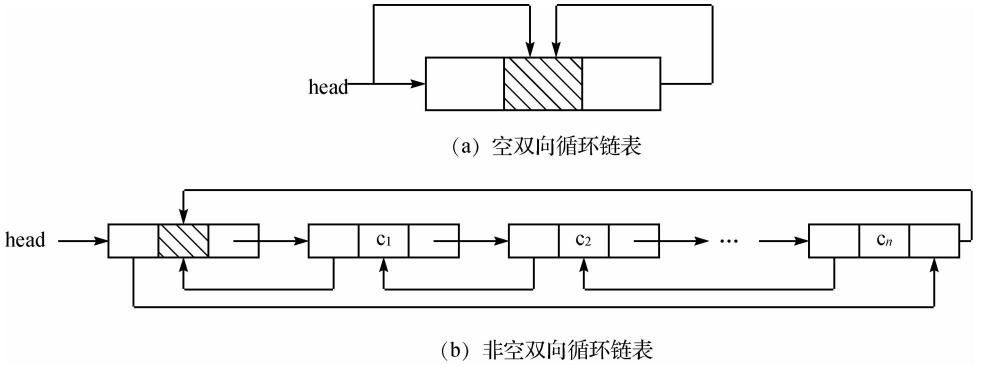


图 2-16　双向循环链表

假设指针 p 指向双向循环链表的某一结点，那么，p－＞prior 指示 p 所指结点的前驱结点，p－＞prior－＞next 中存放的是 p 所指结点前驱结点的后继结点的地址，即 p 所指结点本身；同样，p－＞next 指示 p 所指结点的后继结点，p－＞next－＞prior 也指向 p 所指结点本身。因此有 p＝＝p－＞prior－＞next＝＝p－＞next－＞prior，如图 2-17 所示。

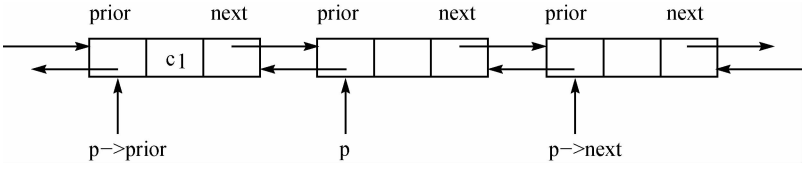


图 2-17　结点指针的指向

双向链表的插入和删除与单链表不同。在双向链表中，在 p 指针所指结点前插入一个结点 s 的语句序列如下：

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

带头结点双向循环链表 L 为空的条件为：

```
(L＝＝L－＞next)&&(L＝＝L－＞prior)
```

在双向链表中删除 p 结点时需要修改指针的语句序列为：

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

1)双向循环链表初始化

初始化双向循环链表 L 算法如下：

```
void InitList(LinkList &L)
{
    /＊产生空间的双向循环链表＊/
    L＝(LinkList)malloc(sizeof(LNode));
    if(L)
        L－＞next＝L－＞prior＝L;
```

```
    else
        exit(1);
}
```

2)求双向链表的长度

求双向循环链表 L 的长度算法如下：

```
int ListLength(LinkList L)
{
    /*返回表的长度*/
    int i=0;
    LinkList p=L->next;          /*p 指向第一个结点*/
    while(p!=L)
    {
        i++;
        p=p->next;
    }
    return i;
}
```

3)在双向链表中搜索第 *i* 个结点,并返回该结点的地址

在双向链表 L 中搜索第 *i* 个结点,并返回第 *i* 个元素的地址算法如下：

```
LinkList GetElemP(LinkList L,int i)
{
    /*在双向链表中返回第 i 个元素的地址*/
    int j;
    LinkList p=L;                /*p 指向头结点*/
    if(i<0||i>ListLength(L))
        return NULL;
    for(j=1;j<=i;j++)            /*p 指向第 i 个结点*/
        p=p->next;               /*p 指向下一个结点*/
    return p;
}
```

4)在双向链表的指定位置 *i* 插入结点

在双向链表 L 中的第 *i* 个位置插入结点的算法如下：

```
DataType ListInsert(LinkList L,int i,DataType e)
{
    /*在链表第 i 个位置上插入元素*/
    LinkList p,s;
    if(i<1||i>ListLength(L)+1)
        return 0;
```

```
    p=GetElemP(L,i-1);        /* 在 L 中确定第 i 个结点前驱的位置指针 p */
    if(! p)
        return 0;
    s=(LinkList)malloc(sizeof(LNode));    /* 生成新结点 */
    if(! s)
        return 0;
    s->data=e;                /* 将 e 赋给新的结点 */
    s->prior=p;               /* 新结点的前驱为第 i-1 个结点 */
    s->next=p->next;          /* 新结点的后继为第 i 个结点 */
    p->next->prior=s;         /* 第 i 个结点的前驱指向新结点 */
    p->next=s;                /* 第 i-1 个结点的后继指向新结点 */
    return 1;
}
```

5)双向链表的删除

在双向链表 L 中删除第 i 个位置上结点的算法如下：

```
DataType ListDelete(LinkList L,DataType &e,int i)
{
    /* 删除第 i 个结点 */
    LinkList p;
    int i;
    if(i<1)
        return 0;
    p=GetElemP(L,i);     /* 在 L 中确定第 i 个元素的位置指针 */
    if(! p)
        return 0;
    e=p->data;  /* 把第 i 个结点的元素的值赋给 e */
    p->prior->next=p->next;  /* 原第 i-1 个结点的后继指向原第 i+1
个结点 */
    p->next->prior=p->prior; /* 原第 i+1 个结点的前驱指向原第 i-1
个结点 */
    free(p);
    return 1;
}
```

2.5 顺序表与链表的优缺点

本章介绍了线性表的逻辑结构及两种线性表:顺序表和链表。通过对它们的讨论可知

它们各有其优缺点。

1)顺序表的优点

(1)方法简单,各种高级语言中都有数组,容易实现。

(2)不用为表示结点间的逻辑关系而增加额外的存储开销,存储密度大。

(3)顺序表具有按元素序号随机访问的特点,查找速度快,时间复杂度为 $O(1)$。

2)顺序表的缺点

(1)在顺序表中进行插入、删除操作时,平均移动大约表中一半的元素,因此对 n 较大的顺序表执行效率低。

(2)需要预先分配适当的存储空间,预先分配过大,可能会导致顺序表后部大量闲置;预先分配过小,又会造成溢出。

3)链表的优点

(1)插入、删除时,只要找到对应的前驱结点,修改指针即可,无需移动元素。

(2)采用动态存储分配,不会造成内存浪费和溢出。

4)链表的缺点

(1)在有些高级语言中,不支持指针,不容易实现。

(2)需要用额外空间存储线性表的关系,存储密度小。

(3)不能随机访问,查找时要从头指针开始遍历,查找元素的时间复杂度为 $O(n)$。

总之,两种线性表各有优劣,选择哪一种应由实际问题中的主要因素决定。通常"较稳定"的线性表选择顺序存储,而需要频繁插入、删除的,即动态性较强的线性表宜选择链式存储。

2.6 线性表的应用

【例 2-1】 写一个算法,删除顺序表中值相同的结点。

算法思想:从顺序表的开始结点检查它与后继结点的值是否相等,若相等,则删除后继结点,即把后继结点及之后结点全部前移一个位置,否则继续检查。

```
SeqList del(SeqList L)
{
    int i=1;
    while(i<=L.length-1)
    {
        int j=i+1;
        while(j<=L.length)
            if(L.data[i]==L.data[j])
            {
                for(int k=j+1;k<=L.length-1;k++)
                    L.data[k-1]=L.data[k];
                L.length--;
```

```
            }
            else j++;
            i++;
        }
        return L;
    }
```

【例 2-2】 写一个算法,将单链表逆置。要求逆置在原表上进行,不允许重构新表。

算法思想:先要把链表的第一个结点孤立,然后用链表的头插法把后面的结点插到孤立结点的后面。

```
    LNode * Reverse(LNode * head)        / * 对单链表进行逆序操作的函数 * /
    {
        LNode * p, * q;     / * p 将代表逆序后单链表的第一个节点,q 代表原单链表中 p 之后紧邻的结点,起交换作用 * /
        if(head!=NULL)
        {
            p=head->next;
            head->next=NULL;                    / * 将原单链表置空 * /
            while(p!=NULL)                      / * 如果 p 不为 NULL * /
            {
                q=p->next;                      / * 把当前节点的下一个结点赋给 q * /
                p->next=head->next;             / * 若当前节点为原链表中的第一个结点,则使其 next 指向 NULL,否则使其 next 指向原链表中当前节点的上一个结点,也就是正在逆序中的第一个结点 * /
                head->next=p;                   / * 使头指针指向当前结点 * /
                p=q;                            / * 把原 p 的下一个结点赋给 p * /
            }
        }
        return(head);
    }
```

【例 2-3】 写一个算法,将两个带头结点的单循环链表 LA 和 LB 合并为一个单循环链表,其头指针为 LA。

算法思想:首先分别将指针 p 和 q 指向 LA 和 LB,再找到两个链表的表尾,然后将第一个链表的表尾与第二个链表的第一个结点链接,并修改第二个链表的表尾指针,使它指向第一个链表的头结点。

```
    LinkList merge(LinkList LA,LinkList LB)
    {
        LNode * p, * q;
        p=LA;
        q=LB;
```

```
        while(p->next!=LA)p=p->next;  /*找到表LA的尾,p指向LA的表尾*/
        while(q->next!=LB)q=q->next;  /*找到表LB的尾,q指向LB的表尾*/
        q->next=LA;  /*修改表LB的尾指针,使它指向表LA的头结点*/
        p->next=LB->next;  /*修改表LA的尾指针,使它指向表LB的第一个结点*/
        free(LB);
        return LA;
    }
```

【例 2-4】 写一个算法,将双向循环链表逆置。

算法思想:因为双向循环链表的每一个结点都有左右指针,所以要逆置时,只要从头结点开始,让每一个结点的左指针指向右节点,右指针指向左结点就可以实现逆置。

```
void DoLinkReversal(DuLinkList L)
{
    LNode *p, *q;
    p=L;
    while(p!=L)
    {
        q=p->next;
        p->next=p->prior;
        p->prior=q;
        p=q;
    }
}
```

【例 2-5】 学生信息管理是学校管理的重要组成部分,其处理的信息量很大,对学生的信息管理作一个简单的模拟,用菜单选择操作方式完成下列功能:

(1)录入学生信息。

(2)浏览学生信息。

(3)查询学生信息。

(4)删除学生信息。

(5)修改学生信息。

(6)插入学生信息。

本题实质上是建立学生信息单链表,每条信息由学号、姓名、年龄、性别、出生年月、地址、电话组成,即链表中每个结点由 8 个域组成,分别为:学号、姓名、年龄、性别、出生年月、地址、电话、存放下一个结点地址的 next 域。要求完成的 6 项功能,分别实现单链表的创建、浏览、查询、插入、删除与修改等操作。

参考程序如下:

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

```
#include<string.h>
#define LEN sizeof(struct student)
#define FILE_DATA_PATH "student.txt"
struct student              /*建立一个学生的结构*/
{
    long int num;           /*学生学号*/
    char name[20];          /*学生姓名*/
    int age;                /*年龄*/
    char sex[4];            /*性别*/
    char birthday[10];      /*出生年月*/
    char address[30];       /*地址*/
    long int tele_num;      /*电话号码*/
    struct student *next;   /*指针指向学生结构*/
};
int TOTAL_NUM=0;            /*初始化总人数为0*/
struct student *head=NULL;          /*指针指向的结构的首位为空*/
void welcome();                     /*显示欢迎信息*/
void mainmenu();                    /*系统主菜单*/
void record();                      /*申请学生结点,输入学生信息并把结点插
入链表中*/
void insertLink(struct student *stu);/*插入学生信息到线性表中*/
void insertLink1();                 /*插入学生信息到线性表中*/
void display(struct student *stu);  /*浏览一个学生信息*/
void displayAll();                  /*浏览学生信息*/
void query();                       /*查询学生信息*/
void Locate_num();                  /*按学号查询学生信息*/
void Locate_name();                 /*按姓名查询学生信息*/
void readData();                    /*读取文件*/
void writeData();                   /*写入文件*/
void DelLink();                     /*删除学生信息*/
void change();                      /*修改学生信息*/
void devise(struct student *p);     /*修改学生信息菜单*/
int main()                          /*主函数*/
{
    char userName[9];
    char userPWD[7];
    int i;
    welcome();
    for(i=0;i<3;i++)
```

```
    {
        printf("\n管理员初始用户名和密码均为123\n");
        printf("请输入您的用户名:");
        scanf("%s",userName);
        printf("\n请输入您的密码:");
        scanf("%s",userPWD);
        if((strcmp(userName,"123")==0)&&(strcmp(userPWD,"123")==0))
        /*比较输入的用户名与密码和默认的用户名与密码是否相同*/
        {
            /*用户名和密码正确,显示主菜单*/
            mainmenu();
            break;
        }
        else
        {
            if(i<2)
            {
                /*用户名或密码错误,提示用户重新输入*/
                printf("用户名或密码错误,请重新输入!");
            }
            else
            {
                /*连续3次输错用户名或密码,退出系统*/
                printf("您已连续3次将用户名或密码输错,系统将退出!");
            }
        }
    }
    return 0;
}
/*显示欢迎信息*/
void welcome()
{
    printf("\t\t\t+------------------------+\n");
    printf("\t\t\t| |\n");
    printf("\t\t\t| 欢迎使用学生信息管理系统 |\n");
    printf("\t\t\t| |\n");
    printf("\t\t\t+------------------------+\n");
    printf("\n\n\n\n\n\n\n\n\n");
    printf("\t\t\t----------本程序为线性表");
```

```
}
/*系统主菜单*/
void mainmenu()
{
    int choice;
    choice=-1;
    readData();
    printf("\n温馨提示:为保证您的操作得到保存,请按正常顺序退出系统^_^\n");
    do
    {
        printf("\n\n\n");
        printf("\t\t\t----------------------------\n");
        printf("\t\t\t| 学生信息管理系统 |\n");
        printf("\t\t\t----------------------------\n");
        printf("\t\t\t| [1]----录入学生信息 |\n");
        printf("\t\t\t| [2]----浏览学生信息 |\n");
        printf("\t\t\t| [3]----查询学生信息 |\n");
        printf("\t\t\t| [4]----删除学生信息 |\n");
        printf("\t\t\t| [5]----修改学生信息 |\n");
        printf("\t\t\t| [6]----插入学生信息 |\n");
        printf("\t\t\t| [0]----退出系统 |\n");
        printf("\t\t\t|*·*·*·*·*·*·*·*·*·*·*·*·*·|\n");
        printf("\t\t\t----------------------------\n");
        printf("请输入您的选择:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 0:
                writeData();
                exit(0);
            case 1:
                record();
                break;
            case 2:
                displayAll();
                break;
            case 3:
                query();
                break;
```

```
            case 4:
                  DelLink();
                  break;
            case 5:
                  change();
                  break;
            case 6:
                  insertLink1();
                  break;
            default:
                  printf("\n 无效选项!");
                  break;
         }
    }
    while(choice!=0);
}
/* 输入学生信息 */
void record()
{
    struct student *p0;
    p0=(struct student *)malloc(LEN);  /* 为学生结构开辟一个内存空间 */
    printf("请输入学生的学号:");
    scanf("%ld",&p0->num);  /* 新建的学生指针指向学号,以下类似! */
    printf("请输入学生的姓名:");
    scanf("%s",p0->name);
    printf("请输入学生的年龄:");
    scanf("%d",&p0->age);
    printf("请输入学生的性别:");
    scanf("%s",p0->sex);
    printf("请输入学生的出生年月:");
    scanf("%s",p0->birthday);
    printf("请输入学生的地址:");
    scanf("%s",p0->address);
    printf("请输入学生的电话:");
    scanf("%ld",&p0->tele_num);
    insertLink(p0);
    printf("该学生的信息为:\n");
    printf("-------------------------------------------------------------------------");
    printf("学号\t 姓名\t 年龄\t 性别\t 出生年月\t\t 地址\t 电话\n");
```

```
    display(p0);
}
void insertLink(struct student * stu)        /* 插入学生信息到线性表中 */
{
    struct student * p0, * p1, * p2;
    p1=head;
    p0=stu;
    if(head==NULL)
    {
        head=p0;
        p0->next=NULL;
    }
    else
    {
        while((p0->num>p1->num)&&(p1->next!=NULL))
        {
            p2=p1;
            p1=p1->next;
        }
        if(p0->num<=p1->num)
        {
            if(head==p1)
                head=p0;
            else
                p2->next=p0;
            p0->next=p1;
        }
        else
        {
            p1->next=p0;
            p0->next=NULL;
        }
    }
    TOTAL_NUM++;
}
void insertLink1()
{
    record();
}
```

```
void display(struct student * p)
{
    printf("%ld\t%s\t%d\t%s\t%s\t%s\t%ld\n",p->num,p->name,p->
age,p->sex,p->birthday,p->address,p->tele_num);
}
/*浏览学生信息*/
void displayAll()
{
    struct student * p;
    printf("学生总数:%d\n",TOTAL_NUM);
    p=head;
    if(head!=NULL)
    {
        printf("\n学号\t姓名\t年龄\t性别\t出生年月\t地址\t电话\n");
        printf("-------------------------------------------------------------------------");
        do
        {
            display(p);
            p=p->next;
        }
        while(p!=NULL);
    }
    printf("\n");
}
/*查询学生信息*/
void query()
{
    int choice;
    choice=-1;
    do
    {
        printf("\n");
        printf("+-----------------+\n");
        printf("|[1]---按学号查询 |\n");
        printf("|[2]---按姓名查询 |\n");
        printf("|[0]---取消 |\n");
        printf("+-----------------+\n");
        printf("请输入您的选择:");
        scanf("%d",&choice);
```

```
            switch(choice)
            {
                case 0:
                    return;
                case 1:
                    Locate_num();
                    break;
                case 2:
                    Locate_name();
                    break;
                    default:
                    printf("\n无效选项!");
                    break;
            }
        }
        while(choice!=0);
}
/*按学号查询学生信息*/
void Locate_num()
{
    int num;
    struct student *p1;
    printf("请输入学生的学号:");
    scanf("%ld",&num);
    if(head==NULL)
    {
        printf("无学生记录! \n");
        return;
    }
    p1=head;
    while(num!=p1->num && p1->next!=NULL)
        p1=p1->next;
    if(num==p1->num)
    {
        printf("\n学号\t姓名\t年龄\t性别\t出生年月\t地址\t电话\n");
        printf("-------------------------------------------------------------------");
        display(p1);
    }
    else
```

```
        printf("没有该学生记录,请核对!");
}
/*按姓名查询学生信息*/
void Locate_name()
{
    char name[20];
    struct student *p1;
    printf("请输入学生的姓名:");
    scanf("%s",name);
    if(head==NULL)
    {
        printf("无学生记录! \n");
        return;
    }
    p1=head;
    while(strcmp(name,p1->name)&& p1->next!=NULL)
        p1=p1->next;
    if(! strcmp(name,p1->name))
    {
        printf("\n 学号\t 姓名\t 年龄\t 性别\t 出生年月\t 地址\t 电话\n");
        printf("-------------------------------------------------------------------");
        display(p1);
    }
    else
        printf("没有该学生记录,请核对!");
}
/*写入文件*/
void writeData()
{
    FILE *fp;       /*文件指针*/
    struct student *p;
    fp=fopen(FILE_DATA_PATH,"w");
    if(! fp)
    {
        printf("文件打开错误!");
        return;
    }
    fprintf(fp,"%d\n",TOTAL_NUM);
    for(p=head;p!=NULL;p=p->next)
```

```
        {
            fprintf(fp,"%ld\t%s\t%d\t%s\t%s\t%s\t%ld\n",
            p->num,p->name,p->age,p->sex,p->birthday,p->address,
p->tele_num);
        }
        fclose(fp);
    }
    /*读取文件*/
    void readData()
    {
        FILE *fp;          /*文件指针*/
        struct student *p1,*p2;
        fp=fopen(FILE_DATA_PATH,"r");
        if(!fp)
        {
            printf("文件打开错误!");
            return;
        }
        fscanf(fp,"%d\n",&TOTAL_NUM);
        head=p1=p2=(struct student *)malloc(LEN);
        fscanf(fp,"%ld\t%s\t%d\t%s\t%s\t%s\t%ld\n",
        &p1->num,p1->name,&p1->age,p1->sex,p1->birthday,p1->ad-
dress,&p1->tele_num);
        while(!feof(fp))
        {
            p1=(struct student *)malloc(LEN);
            fscanf(fp,"%ld\t%s\t%d\t%s\t%s\t%s\t%ld\n",
            &p1->num,p1->name,&p1->age,p1->sex,p1->birthday,p1->ad-
dress,&p1->tele_num);
            p2->next=p1;
            p2=p1;
        }
        p2->next=NULL;
        fclose(fp);
    }
    /*删除学生信息*/
    void DelLink()
    {
        struct student *p1,*p2;
```

```
    long int num;
    if(head==NULL)
    {
        printf("无学生记录! \n");
        return;
    }
    printf("请输入您要删除的学生的学号:");
    scanf("%ld",&num);
    p1=head;
    while(num!=p1->num && p1->next!=NULL)
    {
        p2=p1;
        p1=p1->next;
    }
    if(num==p1->num)
    {
        if(p1==head)
            head=p1->next;
        else p2->next=p1->next;
        free(p1);
        TOTAL_NUM--;
    }
    else
    printf("没有该学生记录,请核对! \n");
}
/*修改学生信息*/
void change()
{
    struct student *p1,*p2;
    long int num;
    if(head==NULL)
    {
        printf("无学生记录! \n");
        return;
    }
    printf("请输入您要修改的学生的学号:");
    scanf("%ld",&num);
    p1=head;
    while(num!=p1->num && p1->next!=NULL)
```

```
    {
        p2=p1;
        p1=p1->next;
    }
    if(num==p1->num)
        devise(p1);
    else
        printf("没有该学生记录,请核对! \n");
}
void devise(struct student * p)        /* 修改学生信息菜单 */
{
    int choice;
    choice=-1;
    do
    {
        printf("请选择您要修改的学生的信息内容:\n");
        printf("+--------------------+\n");
        printf("| 姓名 请按 1 |\n");
        printf("| 年龄 请按 2 |\n");
        printf("| 性别 请按 3 |\n");
        printf("| 出生年月 请按 4 |\n");
        printf("| 地址 请按 5 |\n");
        printf("| 电话 请按 6 |\n");
        printf("| 取消 请按 0 |\n");
        printf("+--------------------+\n");
        printf("请输入您的选择:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 0:
                return;
            case 1:
                printf("请输入新姓名:");
                scanf("%s",p->name);
                break;
            case 2:
                printf("请输入新年龄:");
                scanf("%d",&p->age);
                break;
```

```
            case 3:
                printf("请输入新性别:");
                scanf("%s",p->sex);
                break;
            case 4:
                printf("请输入新出生年月:");
                scanf("%s",p->birthday);
                break;
            case 5:
                printf("请输入新地址:");
                scanf("%s",p->address);
                break;
            case 6:
                printf("请输入新电话:");
                scanf("%ld",&p->tele_num);
                break;
            default:
                printf("\n无效选项!");
                break;
        }
    }while(choice!=0);
}
```

2.7 习 题

1)选择题

(1)顺序表是线性表的(　　)。

A. 链式存储结构　　B. 顺序存储结构

C. 索引存储结构　　D. 散列存储结构

(2)顺序表的一个存储结点仅仅存储线性表的一个(　　)。

A. 数据元素　　B. 数据项

C. 数据　　D. 数据结构

(3)以下说法错误的是(　　)。

A. 线性表的元素可以是各种各样的,逻辑上相邻的元素在物理位置上不一定相邻

B. 在线性表的顺序存储结构中,逻辑上相邻的两个元素在物理位置上不一定相邻

C. 在线性表的链式存储结构中,逻辑上相邻的元素在物理位置上不一定相邻

D. 线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素

(4)线性表 L=(a_1,a_2,…,a_i,…,a_n),下列说法正确的是(　　)。

A. 每个元素都有一个直接前驱和直接后继

B. 线性表中至少要有一个元素

C. 表中诸元素的排列顺序必须是由小到大或由大到小的

D. 除第一个元素和最后一个元素外其余每个元素都有一个且仅有直接前驱和直接后继

(5)在双向链表中,删除 p 所指的结点时,需要修改指针(　　)。

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

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

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

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

(6)已知 L 是无表头结点的单链表,且 *p* 结点既有直接前驱结点又有直接后继结点。按要求从下列语句中选择合适的语句序列。

①在 *p* 结点后插入 *s* 结点的语句序列是(　　)。

②在 *p* 结点前插入 *s* 结点的语句序列是(　　)。

③在表首插入 *s* 结点的语句序列是(　　)。

④在表尾插入 *s* 结点的语句序列是(　　)。

供选择的语句有:

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

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

E. s->next=L;　　F. s->next=NULL;

G. q=p;　　H. while(p->next!=q)　p=p->next;

I. while(p->next!=NULL)p=p->next;

J. p=q;　　K. p=L;　　L. L=s;　　M. L=p;

(7)在链表中删除尾结点和在尾结点之后插入元素,则采用(　　)最节省时间。

A. 带头指针的单向循环链表　　B. 双向链表

C. 带尾指针的单向循环链表　　D. 带头指针的双向循环链表

2)判断题

(1)线性表的逻辑顺序与存储顺序总是一致的。(　　)

(2)在线性表的顺序存储结构中,逻辑上相邻的两个元素在物理位置上并不一定相邻。(　　)

(3)线性表中每个结点都有一个直接前驱和一个直接后继。(　　)

(4)线性表的链式存储结构是用一组任意的存储单元来存储线性表中数据元素的。(　　)

(5)单链表 head 为空的判定条件是 head==NULL 或 head->next==NULL。(　　)

(6)循环单链表的最大优点是从任一结点出发都可访问到链表中的每一个元素。(　　)

(7)带头结点的双循环链表 L 中只有一个元素结点的条件是 L->next->next==L。(　　)

3)填空题

(1)线性结构的基本特征是:若至少含有一个结点,则除起始结点没有直接_____外,其他结点有且仅有一个直接_____;除终端结点没有直接_____外,其他结点有且仅有一个直接_____。

(2)所有结点按一对一的邻接关系构成的整体就是_____结构。

(3)在顺序表中插入或删除一个元素,需要平均移动_____元素,具体移动的元素个数与_____有关。

(4)单链表表示法的基本思想是用_____表示结点间的逻辑关系。

(5)为了便于实现各种操作,通常在单链表的第一个结点之前增设一个类型相同的结点,称为_____。

(6)循环链表与单链表的区别仅仅在于其尾结点的链域值不是_____,而是一个指向_____的指针。

(7)在单链表中,若在每个结点中增加一个指针域,所含指针指向前驱结点,这样构成的链表中有两个方向不同的链,称为_____。

(8)在一个长度为 n 的向量的第 i 个元素($1\leqslant i\leqslant n$)之前插入一个元素时,需向后移动_____个元素。

(9)在一个长度为 n 的向量中删除第 i 个元素($1\leqslant i\leqslant n$)时,需向前移动_____个元素。

(10)在 n 个结点的单链表中要删除已知结点 * p,需找到它的_____,其时间复杂度为_____。

4)简答题

(1)试描述头指针、头结点、开始结点的区别,并说明头指针和头结点的作用。

(2)为什么在单循环链表中设置尾指针比设置头指针更好?

(3)何时选用顺序表作为线性表的存储结构?何时选用链表作为线性表的存储结构?

5)算法设计

(1)设有一个线性表 L=(a_1,a_2,…,a_i,…,a_n),设计一个算法,将线性表逆序,要求逆序后的线性表仍占用原线性表空间,分别用顺序表和单链表两种方法表示,写出不同的处理函数。

(2)编写一个逐个输出顺序表中所有数据元素的函数。

(3)编写一个逐个输出单链表中所有数据元素的函数。

(4)将线性表 L 拆分成两个线性表 LA、LB,要求线性表 LA 中的数据元素为奇数,LB 中的数据元素为偶数,分别用顺序表和单链表两种方法表示,写出不同的处理函数。

(5)编写算法,在双向链表中查找值为 x 的结点,要求分别用从前往后和从后往前两种方法实现。

第 3 章　栈 和 队 列

栈和队列是特殊的操作受限的线性表。对于栈，所有的插入和删除操作都限制在线性表的同一端进行，是一种后进先出的线性表。对于队列，所有的插入操作限制在线性表的一端进行，所有的删除操作限制在线性表的另一端进行，是一种先进先出的线性表。

本章学习要点

- 栈和队列的基本概念和基本运算。
- 栈和队列的顺序存储结构、串的链式存储结构。
- 栈和队列的顺序存储结构中基本算法的实现。
- 栈和队列的应用。

3.1　栈 的 定 义

栈(stack)是一种最常用和最重要的数据结构。通常，栈可定义为只允许在表的末端进行插入和删除的线性表。允许插入和删除的一端叫做栈顶(top)，而不允许插入和删除的一端叫做栈底(bottom)，当栈中没有任何元素时则称为空栈。如图 3-1 所示，栈中有 n 个元素，进栈的顺序是 $a_1, a_2, \cdots, a_n$，当需要出栈时其顺序为 $a_n, \cdots, a_2, a_1$。

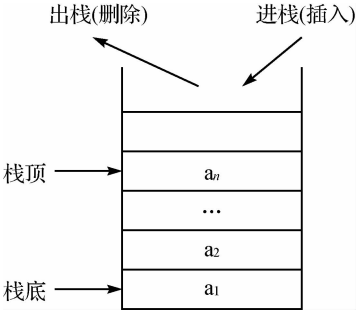


图 3-1　栈

因此，栈是后进先出(last in first out，LIFO)的线性表，简称为LIFO线性表。

3.2 栈的抽象数据类型

栈的抽象数据类型中的数据部分为具有ElemType类型的一个栈，它可以采用任一种存储结构实现；操作部分应包括元素的进栈、元素的出栈、读取栈顶元素、检查栈是否为空等。下面给出栈的抽象数据类型的具体定义：

```
ADT Stack
{
    Dataset:D={ai|ai∈ElemType,i=1,2,…,n,n≥0}
    Relationset:Rl={<ai-1,ai>|ai-1,ai∈D,i=2,…,n}
    约定an端为栈顶，a1端为栈底。
    OperationSet:
    (1)InitStack(&S)
       实现功能：构造一个空栈S。
    (2)DestroyStack(&S)
       初始条件：栈S已存在。
       实现功能：销毁栈S。
    (3)ClearStack(&S)
       初始条件：栈S已存在。
       实现功能：清空栈S。
    (4)StackEmpty(S)
       初始条件：栈S已存在。
       实现功能：判断栈S是否为空栈，若是则返回TRUE，否则返回FALSE。
    (5)StackLength(S)
       初始条件：栈S已存在。
       实现功能：返回栈S中的元素个数，即栈的长度。
    (6)GetTop(S,&e)
       初始条件：栈S已存在且非空。
       实现功能：返回S的栈顶元素，其值存放在e中。
    (7)Push(&S,e)
       初始条件：栈S已存在。
       实现功能：插入元素e作为新的栈顶元素。
    (8)Pop(&S,&e)
       初始条件：栈S已存在且非空。
       实现功能：删除S的栈顶元素，并用e返回其值。
```

(9)StackTraverse(S,visit())

初始条件:栈 S 已存在且非空,visit()为元素的访问函数。

实现功能:从栈底到栈顶依次对 S 的每个元素调用函数 visit(),一旦 visit()失败,则操作失败。

}

ADT Stack

3.3 栈的存储结构与操作

栈的存储结构有顺序存储结构和链式存储结构两种。

3.3.1 栈的顺序存储结构及实现

栈的顺序存储结构同样需要使用一个数组和一个整型变量来实现,即利用数组来顺序存储栈中的所有元素,利用整型变量来存储栈顶元素的下标位置。

栈数组用 stack[MaxStackSize]表示,指示栈顶位置的整型变量用 top 表示,则元素类型为 ElemType 的栈的顺序存储结构可定义如下:

```
typedef struct
{
    ElemType stack[MaxStackSize];
    int top;
}Stack;
```

其中,MaxStackSize 为一个整型的全局常量,需要通过 const 语句定义,由它确定顺序栈(即顺序存储的栈)的最大长度,也称为深度,即栈空间最多能够存储的元素个数,由于 top 用来指示栈顶元素的位置,所以可把它称为栈顶指针。

图 3-2 展示了一个空的顺序栈中数据元素和栈顶指针之间的对应关系。

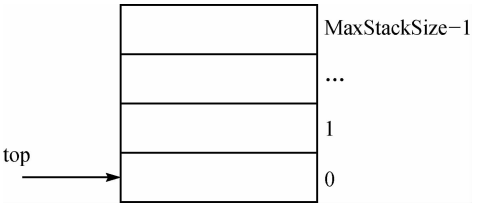


图 3-2 一个空的顺序栈

若要对存储栈的数组空间采用动态分配,则 Stake 结构类型可定义如下:

```
typedef struct
{
    ElemType * stack;
```

```
    int top;
}Stack;
```

在顺序栈中，top的值为－1时表示栈空，top的值为MaxStackSize－1时为栈满。每次向栈中压入一个新元素时，首先top的值要加1，用以指示新的栈顶位置，然后再把新元素赋值到这个空位置上。每次从栈中弹出一个元素时，首先取出栈顶元素，然后使top值减1，指示前一个元素成为新的栈顶元素。由此可知，对于顺序栈的插入和删除操作都是在顺序表的表尾进行的。

基本操作算法如下：

1)初始化栈

```
void InitStack(Stack * S)
{
    if((S=(Stack * )malloc(sizeof(Stack)))==NULL)
        exit(OVERFLOW);
    S->top=-1;
}
```

2)判断栈S是否为空

```
int StackEmpty(Stack S)
{
    if(S.top==-1)return 1;
    else return 0;
}
```

3)入栈

```
void StackPush(Stack * S,ElemType elem)
{
    if(S->top==MaxStackSize-1)
    {printf("%d","Stack is full");exit(0);}
    else S->top=elem;
}
```

4)出栈

```
void StackPop(Stack * S,ElemType * elem)
{
    if(StackEmpty( * S))
    {printf("%d","Stack is full");exit(0);}
    else * elem=S->top--;
}
```

5)获取栈顶元素内容

```
void GetTop(Stack S,ElemType * elem)
```

```
{
    if(StackEmpty(S))
    {printf("%d","Stack is full");exit(0);}
    else *elem=S.top;
}
```

【例 3-1】 建立一个顺序堆栈,首先依次输入数据元素 1,2,3,…,10,然后依次输出堆栈中的数据元素并显示。假设该顺序堆栈的数据元素个数在最坏情况下不会超过 100 个。

参考程序如下:

```
#include "stdio.h"
#include "stdlib.h"
#define MaxStackSize 100
#define OVERFLOW 1
typedef int ElemType;
#include "Stack.h"
void main(void)
{
    Stack myStack;
    int i,x;
    InitStack(&myStack);
    for(i=0;i<10;i++);
    StackPush(&myStack,i);
    GetTop(myStack,&x);
    printf("当前栈顶数据元素为:%d\n",x);
    printf("依次出栈的数据元素序列如下:\n");
    while(! StackEmpty(myStack))
    {
        StackPop(&myStack,&x);
        printf("%d  ",x);
    }
}
```

程序运行结果如图 3-3 所示。

```
当前本顶数据元素为: 10
依次出本的数据元素序列如下:
10   9   8   7   6   5   4   3   2   1   0
```

图 3-3 程序运行结果

3.3.2 栈的链式存储结构

若栈中元素的数目变化范围较大或不清楚栈元素的数目,就应该考虑使用链式存储结

构。链式存储结构表示的栈称为链栈。链栈通常用一个无头结点的单链表表示，结点结构和单链表中的结点结构相同，无需重复定义。由于栈只在栈顶作插入和删除操作，因此链栈中不需要头结点，但要注意链栈中指针的方向是从栈顶指向栈底的，这正好和单链表是相反的，如图 3-4 所示。

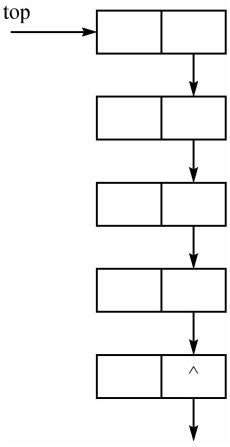


图 3-4 链栈

在一个链栈中，栈底就是链表的最后一个结点，而栈顶总是链表的第一个结点。因此，新入栈的元素即为链表的新的第一个结点，只要系统还有存储空间，就不会有栈满的情况发生。一个链栈(不带头结点)可由栈顶指针 top 唯一确定，当 top 为 NULL 时，该栈是一个空栈。

链栈的定义如下：

```
typedef struct          /*结点结构*/
{
    ElemType data;
    Struct StackNode*next;
}StackNode;
typedef struct          /*栈结构*/
{
    StackNode*top;      /*栈顶指针*/
    int length;         /*栈中元素个数*/
}Stack;
```

链栈各项基本操作的算法如下。

1)初始化栈 S

```
void InitStack(Stack*S)
{
    S->top=NULL;
}
```

2)入栈

```
void Push(Stack*S,ElemType elem)
```

```
{
    p=(StackNode * )malloc(sizeof(StackNode));
    if(! p)exit(OVERFLOW);
    else
    {
        p->data=elem;
        p->next=S->top;
        S->top=p;
    }
}
```

3)出栈

```
void Pop(Stack * S,ElemType &elem)
{
    if(StackEmpty( * S))exit("Stack is empty");
    else
    {
        elem=S->top->data;
        p=S->top;
        S->top=p->next;free(p);
    }
}
```

4)获取栈顶元素内容

```
void GetTop(Stack S,ElemType &elem)
{
    if(StackEmpty(S))exit("Stack is empty");
    else elem=S.top->data;
}
```

5)判断栈 S 是否为空

```
int StackEmpty(Stack S)
{
    if(S.top==NULL)return TRUE;
    else FALSE;
}
```

3.4 栈的应用

栈的应用非常广泛,只要问题满足后进先出原则,均可使用栈作为其数据结构。而且由

于栈结构具有后进先出的固有特性，使得它成为程序设计中常用的工具。以下是几个栈应用的例子。

【例 3-2】 将从键盘输入的字符序列逆置输出。例如，从键盘上输入“tset a si sihT”，将输出“This is a test”。

算法如下：

```
typedef char Elemtype;
void ReverseRead(     )
{
    STACK S;                          /*定义一个栈结构S*/
    char ch;  InitStack(&S);          /*初始化栈*/
    while((ch=getchar())!='\n')       /*从键盘输入字符,直到输入换行符"//"为
止*/
    Push(&S,ch);                      /*将输入的每个字符入栈*/
    while(! StackEmpty(S))            /*依次退栈并输出退出的字符*/
   {
        Pop(&S,&ch);
        putchar(ch);
    }
    putchar('\n');
}
```

【例 3-3】 十进制数转换成其他进制数。十进制数 N 和其他进制数(用 d 表示)的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法基于以下原理：

$$N=(N/d)\times d+N\%d$$

例如，将十进制数 1348 转换为八进制数的过程如下：

N	$N/8$	$N\%8$
1348	168	4
168	21	0
21	2	5
2	0	2

参考程序如下：

```
#include<stdio.h>
#include<stdlib.h>
#define StackInitSize 100
typedef int StackElementType;
typedef struct
{
    StackElementType data[StackInitSize];
    int top;
}SeqStack;
```

```
SeqStack * InitStack()
{
    SeqStack * s;
    s=(SeqStack * )malloc(sizeof(SeqStack));
    if(s!=NULL)
    {
        s->top=-1;
        return s;
    }
    else
    {
        printf("没有足够内存空间,申请失败,程序运行终止。\n");
        exit(0);
    }
}
int IsEmpty(SeqStack * s)
{
    return(s->top==-1)? 1:0;
}
void Push(SeqStack * s,StackElementType x)
{
    if(s->top==StackInitSize)
    {
        printf("栈满! 栈发生上溢,程序运行终止! \n");
        exit(0);
    }
    else
    {
        s->top++;
        s->data[s->top]=x;
    }
    return;
}
StackElementType Pop(SeqStack * s)
{
    StackElementType temp;
    if(IsEmpty(s))
    {
        printf("栈空! 栈发生下溢,程序运行终止! \n");
```

```
            exit(0);
        }
        else
        {
            temp=s->data[s->top];
            s->top--;
            return temp;
        }
}
void conversion(int N,int r)
{
    SeqStack * s;
    StackElementType x;
    s=InitStack();
    while(N)
    {
        Push(s,N % r);
        N=N/r;
    }
    while(! IsEmpty(s))
    {
        x=Pop(s);
        printf(" %d",x);
    }
    printf("\n");
}
void main()
{
    int N,r;
    printf("输入十进制整数:");
    scanf(" %d",&N);
    printf("输入数字转换基数(2~9):");
    scanf(" %d",&r);
    printf("十进制数 %d 转换为 %d 进制数结果:",N,r);
    conversion(N,r);
}
```

程序运行结果如图 3-5 所示。

图 3-5 【例 3-3】程序运行结果

【例 3-4】 三种括号的匹配问题。假设在一个算术表达式中可以包含三种括号:圆括号“( )”、方括号“[ ]”和花括号“{}”,并且这三种括号可以按任意的次序嵌套使用。例如,…[…{…}…[…]…]…[…]…(…)…。现在需要设计一个算法,用来检验在输入的算术表达式中所使用括号的合法性。

算术表达式中各种括号的使用规则为:出现左括号,必有相应的右括号与之匹配,并且每对括号之间可以嵌套,但不能出现交叉情况。我们可以利用一个栈结构保存每个出现的左括号,当遇到右括号时,从栈中弹出左括号,检验匹配情况。在检验过程中,若遇到以下几种情况之一,就可以得出括号不匹配的结论。

(1)当遇到某一个右括号时,栈已空,说明到目前为止,右括号多于左括号。

(2)从栈中弹出的左括号与当前检验的右括号类型不同,说明出现了括号交叉情况。

(3)算术表达式输入完毕,但栈中还有没有匹配的左括号,说明左括号多于右括号。

算法如下:

```
typedef char Elemtype;
int Check (     )
{
    STACK S;          /*定义栈结构 S*/
    char ch;
    InitStack(&S);  /*初始化栈 S*/
    while((ch=getchar())!='\n')       /*以字符序列的形式输入表达式*/
    {
        switch(ch)
        {
            case(ch=='('||ch=='['||ch=='{'):Push(&S,ch);break;     /*遇左括号入栈*/
            case(ch==')'):          /*在遇到右括号时,分别检测匹配情况*/
            if(StackEmpty(S))return FALSE;
            else
            {
                Pop(&S,&ch);
                if(ch!='(')return FALSE;
            }
```

```
                break;
                case(ch==']'):
                if(StackEmpty(S))return FALSE;
                else
                {
                    Pop(&S,&ch);
                    if(ch!='[')return FALSE;
                }
                break;
                case(ch=='}'):
                if(StackEmpty(S))return FALSE;
                else
                {
                    Pop(&S,&ch);
                    if(ch!='{')return FALSE;
                }
                break;
                default:break;
            }
        }
        if(StackEmpty(S))return TRUE;
        else return FALSE;
    }
```

栈的一个重要应用是在程序设计语言中实现递归过程。递归是程序设计中一个强有力的工具。一个直接调用自己或通过一系列的过程调用语句间接地调用自己的函数，称为递归函数。如众所周知的阶乘函数 $n!$ 的定义如下：

$$n!=\begin{cases}1 & n=0 \quad //\text{递归终止条件}\\ n\times(n-1) & n>0 \quad //\text{递归步骤}\end{cases}$$

根据定义可以写出如下相应的递归函数：

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

递归函数都有一个终止递归的条件，如【例 3-4】中，当 $n=0$ 时，将不再继续递归下去。有的数据结构，如二叉树、广义表等，由于结构本身固有的递归特性，使得它们的操作可递归地描述。还有一类问题，虽然问题本身没有明显的递归结构，但用递归求解比迭代求解更简单，如八皇后问题、汉诺塔问题等。

【例 3-5】 斐波那契数列。

```
public static longfib(int n)
{
    /*非递归算法*/
    if(n<3)
       return 1;
    else{
        long a=1;
        long b=1;
        for(int i=2;i<n-1;i++){
            b=a+b;
            a=b-a;
            System.out.println("a="+a+"  b="+b);
        }
        return a+b;
    }
    struct Node{
    int n,tag;
};
long Fibnacci(long N)          /*递归算法*/
{
    Stack S;
    Node w;
    long sum=0;
    do{
        while(N>1){
            w.n=N;
            w.tag=1;
            S.Push(w);
            N--;
        }
        /*向左递归到底,边走边进栈*/
        sum=sum+N;
        while(! S.IsEmpty()){
            S.Pop(w);
            if(w.tag==1){
                /*改为向右递归*/
                w.tag=2;
                S.Push(w);
```

```
                N=w.n-2;
                break;
            }
        }
    }while(! S.IsEmpty());
    return sum;
}
```

显然，斐波那契数列算法是一个递归函数，在函数的执行过程中，需多次进行自我调用。那么，这个递归函数是如何执行的呢？

在使用高级语言编写的程序中，调用函数与被调用函数之间的链接和信息交换必须通过栈进行。当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成以下三件事：

(1)将所有的实际参数、返回地址等信息传递给被调用函数保存。

(2)为被调用函数的局部变量分配存储区。

(3)将控制转移到被调用函数的入口。

从被调用函数返回调用函数之前，应该完成：

(1)保存被调函数的计算结果。

(2)释放被调函数的数据区。

(3)依照被调函数保存的返回地址将控制转移到调用函数。

多个函数嵌套调用的规则是：后调用先返回，此时的内存管理实行“栈式管理”。递归函数的调用类似于多层函数的嵌套调用，只是调用函数和被调用函数是同一个函数而已。在每次调用时，系统将属于各个递归层次的信息组成一个活动记录(activation record)，这个记录中包含本层调用的实参、返回地址、局部变量等信息，并将这个活动记录保存在系统的“递归工作栈”中，它的作用是：

(1)将递归调用时的实际参数和函数返回地址传递给下一层执行的递归函数。

(2)保存本层的参数和局部变量，以便从下一层返回时重新使用它们。

每当递归调用一次，就要在栈顶为过程建立一个新的活动记录，一旦本次调用结束，就将栈顶活动记录出栈，根据获得的返回地址信息返回到本次的调用处。

(1)递归执行过程中所占用的数据区，称为递归工作栈。

(2)每一层的递归参数等数据合成一个记录，称为活动记录。

(3)栈顶记录指示当前层的执行情况，称为当前活动记录。

(4)递归工作栈的栈顶指针，称为当前环境指针。

3.5 队列的定义

在日常生活中经常会遇到为了维护社会正常秩序而需要排队的情况，在计算机程序设计中也经常出现类似问题。数据结构“队列”与生活中的“排队”极为相似，也是按“先到先

办”的原则行事的,并且有着严格的限定:既不允许“加塞儿”,也不允许“中途离队”。

队列(queue)是限定只能在一端进行插入和在另一端进行删除操作的线性表。在表中,允许插入的一端称为“队尾(rear)”,允许删除的一端称为“队头(front)”。一个有 5 个元素的队列,入队的顺序依次为 a_1,a_2,a_3,a_4,a_5,出队的顺序依然是 a_1,a_2,a_3,a_4,a_5。显然,队列也是一种运算受限制的线性表,所以又叫先进先出(first in first out,FIFO)表。在日常生活中队列的例子很多,如排队买东西,排头的买完后走掉,新来的排在队尾。

队列的操作与栈的操作类似,不同的是插入操作在表的头部进行,删除操作在表的尾部进行。队列的示意如图 3-6 所示。

图 3-6　队列

插入端和删除端都是浮动的。通常我们将插入端用一个“队尾指针”指示,而删除端用一个“队头指针”指示。

3.6　队列的抽象数据类型

队列的抽象数据类型中的数据部分为一个具有 ElemType 类型的队列,它可以采用任一种存储结构实现;操作部分包括元素的进队、出队、读取队首元素、检查队列是否为空等。

队列的抽象数据类型的具体定义如下:

```
ADT Queue{
    Dataset:D={ai|ai∈ElemType,i=1,2,…,n,n≥0}
    Relationset:R1={<ai-1,ai>|ai-1,ai∈D,i=2,…,n}
    约定其中 a1 端为队列头,an 端为队列尾。
    OperationSet:
    (1)InitQueue(&Q)
        实现功能:构造一个空队列 Q。
    (2)DestroyQueue(&Q)
        初始条件:队列 Q 已存在。
        实现功能:队列 Q 被销毁,不再存在。
    (3)ClearQueue(&Q)
        初始条件:队列 Q 已存在。
        实现功能:将 Q 清为空队列。
    (4)QueueEmpty(Q)
        初始条件:队列 Q 已存在。
        实现功能:若 Q 为空队列,则返回 TRUE,否则返回 FALSE。
```

(5)QueueLength(Q)

初始条件:队列Q已存在。

实现功能:返回Q的元素个数,即队列的长度。

(6)GetHead(Q,&e)

初始条件:Q为非空队列。

实现功能:用e返回Q的队头元素。

(7)EnQueue(&Q,e)

初始条件:队列Q已存在。

实现功能:插入元素e为Q的新的队尾元素。

(8)DeQueue(&Q,&e)

初始条件:Q为非空队列。

实现功能:删除Q的队头元素,并用e返回其值。

}ADT Queue

3.7 队列的存储结构与操作

队列的存储结构有顺序存储结构和链式存储结构两种。

3.7.1 队列的顺序存储结构

和顺序栈类似,在队列的顺序存储结构中,除了用一组地址连续的存储单元依次存放从队列头到队列尾的元素之外,还需附设两个指针front和rear分别指示队列头元素和队列尾元素的位置。为了在C语言中描述方便起见,在此约定:初始化建空队列时,令front=rear=0,每当插入新的队列尾元素时,尾指针增1;每当删除队列头元素时,头指针增1。因此,在非空队列中,头指针始终指向队列头元素,而尾指针始终指向队列尾元素的下一个位置。如图3-7所示为头、尾指针和队列中元素之间的关系。

顺序队列的存储结构定义如下:

```
#define MAXSIZE 100                    /*最大队列长度*/
typedef struct
{
    QElemType *base;
    int front,rear;
}SqQueue;
```

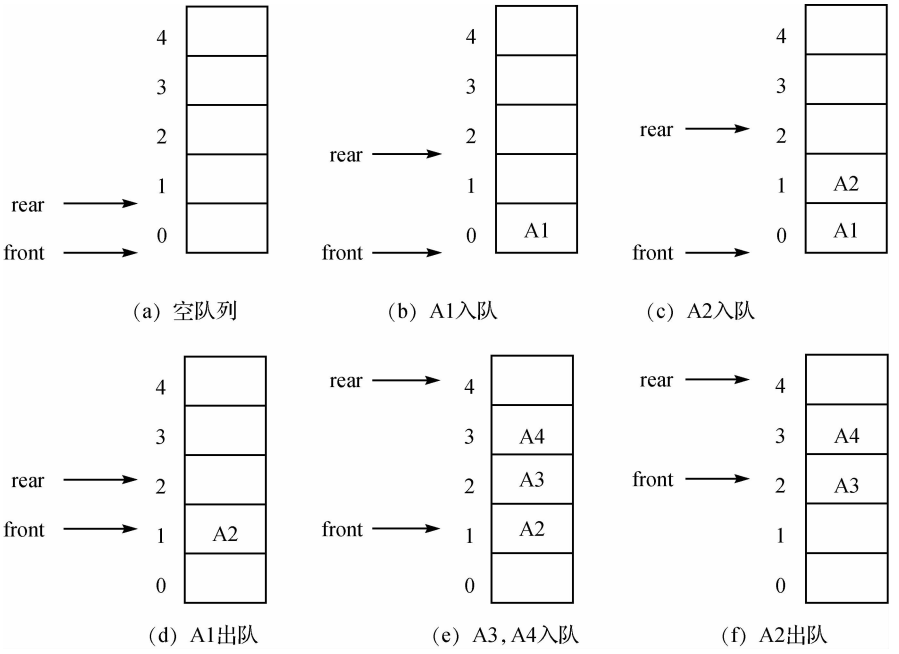


图 3-7　头、尾指针和队列中元素之间的关系

假设当前为队列分配的最大空间为 5,则当队列处于如图 3-7(f)所示的状态时不可再继续插入新的队尾元素,否则会因数组越界而导致程序代码被破坏。然而此时又不宜像顺序栈那样进行存储再分配扩大数组空间,因为队列的实际可用空间并未占满。

3.7.2　队列的链式存储结构

用链表表示的队列简称为链队列,如图 3-8 所示。它是限制仅在表头删除和表尾插入的单链表。显然一个链队列需要两个分别指示队头和队尾的指针(分别称为头指针和尾指针)才能唯一确定。这里和线性表的单链表一样,为了操作方便起见,我们也给链队列添加一个头结点,并令头指针指向头结点。由此,空链队列的判决条件为头指针和尾指针均指向头结点。

链队列的操作即为单链表的插入和删除操作的特殊情况,只是尚需修改队尾指针或队头指针,以便指示队头结点和队尾结点。

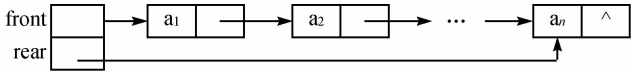


图 3-8　链队列

入队需要执行下面三条语句:

```
s->next=NULL;
rear->next=s;
rear=s;
```

下面是在C语言中,实现队列链式存储结构的类型定义:

```
type struct linklist            /*链式队列的结点结构*/
{
    Elemtype Entry;             /*队列的数据元素类型*/
    struct linklist *next;      /*指向后继结点的指针*/
}LINKLIST;
typedef struct queue            /*链式队列*/
{
    LINKLIST *front;            /*队头指针*/
    LINKLIST *rear;             /*队尾指针*/
}QUEUE;
```

下面我们给出链式队列的基本操作算法。

1)初始化队列Q

```
void InitQueue(QUEUE *Q)
{
    Q->front=(LINKLIST *)malloc(sizeof(LINKLIST));
    if(Q->front==NULL)exit(ERROR);
    Q->rear=Q->front;
}
```

2)入队

```
void EnQueue(QUEUE *Q,Elemtype elem)
{
    s=(LINKLIST *)malloc(sizeof(LINKLIST));
    if(! s)exit(ERROR);
    s->elem=elem;
    s->next=NULL;
    Q->rear->next=s;
    Q->rear=s;
}
```

3)出队

```
void DeQueue(QUEUE *Q,Elemtype *elem)
{
    if(QueueEmpty( *Q))exit(ERROR);
    else
    {
        *elem=Q->front->next->elem;
        s=Q->front->next;
        Q->front->next=s->next;
```

```
        free(s);
    }
}
```

4)获取队头元素内容

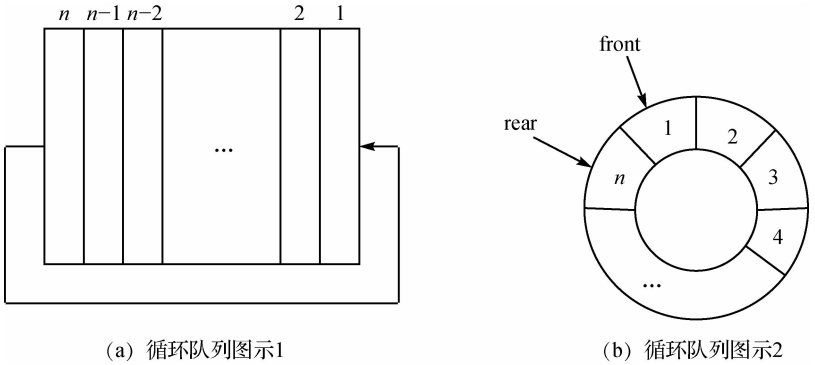
```
void GetFront(QUEUE Q,Elemtype * elem)
{
    if(QueueEmpty(Q))exit(ERROR);
    else * elem=Q->front->next->elem;
}
```

5)判断队列Q是否为空

```
int QueueEmpty(QUEUE Q)
{
    if(Q->front==Q->rear)return TRUE;
    else return FALSE;
}
```

3.8 循 环 队 列

使用顺序存储队列时,随着入队出队的进行,会使整个队列整体向后移动,这样就出现了队尾指针已经移到了最后,再有元素入队就会出现溢出。而事实上此时队中并未真的"满员",这种现象为"假溢出",这是由"队尾入队头出"这种受限制的操作所造成的。解决假溢出的方法之一,是将队列看成头尾相接的循环结构,头尾指针的关系不变,将其称为"循环队列",如图 3-9 所示。



(a) 循环队列图示1　　(b) 循环队列图示2

图 3-9　循环队列

循环队列的存储定义为:

```
#define MAX_QUEUE 10          /*队列的最大数据元素数目*/
```

```
typedef struct queue                /*假设当数组只剩下一个单元时认为队满*/
{
    Elemtype elem[MAX_QUEUE];       /*存放队列中数据元素的存储单元*/
    int front,rear;                 /*队头指针、队尾指针*/
}QUEUE;
```

假设为队列开辟的数组单元数目为 MAX_QUEUE，在 C 语言中，它的下标在 0～MAX_QUEUE－1 之间，增加队头或队尾指针，可以利用取模运算（获取一个整数数值除以另一个整数数值的余数的算法）来实现。如下所示：

```
front＝(front＋1)％MAX_QUEUE;
rear＝(rear＋1)％MAX_QUEUE;
```

当 front 或 rear 为 MAX_QUEUE－1 时，上述两个公式计算的结果就为 0。这样，就使得指针自动由后面转到前面，形成循环的效果。

解决顺序循环队列的队列满和队列空状态的判断问题通常有三种方法：

（1）少用一个存储空间。当少用一个存储空间时，以队尾 rear 加 1 等于队头 front 为队列满的判断条件，即此时队列满的判断条件为：

```
(rear＋1)％MAX_QUEUE＝＝front;
```

队列空的判断条件仍然为：

```
rear＝＝front;
```

（2）设置一个标志位。添加一个标志位。设标志位为 tag，初始设置 tag＝0；每当入队操作成功就设置 tag＝1；每当出队列操作成功就设置 tag＝0。则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。添加一个计数器。设计数器为 count，初始时置 count＝0；每当入队操作成功就使 count 加 1；每当出队操作成功就使 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，则此时队列空的判断条件为：

```
count＝＝0;
```

队列满的判断条件为：

```
count＞0 && rear＝＝front;
```

显然，上述三种方法中用设置计数器的方法判断顺序循环队列的空和满状态是最好的。

各项基本操作算法如下。

1）初始化队列 Q

```
void InitQueue(QUEUE*Q)
{
    Q－＞front＝－1;
    Q－＞rear＝－1;
}
```

2）入队

```
void EnQueue(QUEUE*Q,Elemtype elem)
```

```
{
    if((Q->rear+1)%MAX_QUEUE==Q->front)exit(OVERFLOW);
    else{Q->rear=(Q->reasr+1)%MAX_QUEUE;
    Q->elem[Q->rear]=elem;}
}
```

3)出队

```
void DeQueue(QUEUE *Q,Elemtype *elem)
{
    if(QueueEmpty(*Q))exit("Queue is empty.");
    else
    {
        Q->front=(Q->front+1)%MAX_QUEUE;
        *elem=Q->elem[Q->front];
    }
}
```

4)获取队头元素内容

```
void GetFront(QUEUE Q,Elemtype *elem)
{
    if(QueueEmpty(Q))exit("Queue is empty.");
    else *elem=Q.elem[(Q.front+1)%MAX_QUEUE];
}
```

5)判断队列Q是否为空

```
int QueueEmpty(Queue Q)
{
    if(Q.front==Q.rear)return TRUE;
    else return FALSE;
}
```

3.9 队列的应用

队列在日常生活中和计算机领域都有着广泛的应用。例如,操作系统中的各种数据缓冲区的先进先出管理,应用系统中的各种服务请求的排队管理等。

【例3-6】 构建队列逆置的程序。

参考程序如下:

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

```
#include <seqstack.h>
void proc(SeqQueue *q)
{
    SeqStack s;
    char a;
    InitStack(&s);
    while(! IsEmpty(q))
    {
        DeleteQueue(q,&a);
        Push(&s,a);
    }
    while(! IsEmptys(&s)){
        Pop(&s,&a);
        EnterQueue(q,a);
    }
}
void main()
{
    SeqQueue q;                 /*定义队列 q*/
    char str[50],temp;          /*定义字符数组 str[50],定义字符变量 temp*/
    int i;                      /*定义整型变量 i*/
    InitQueue(&q);              /*初始化队列 q*/
    printf("请输入队列内容:\0");
    gets(str);                  /*输入字符串*/
    for(i=0;str[i]!='\0';i++)
    {
        temp=str[i];            /*输出字符串的每一个字符*/
        EnterQueue(&q,temp);    /*字符串每一个字符入队 q*/
    }
    printf("\n");
    proc(&q);                   /*调用 proc 函数,传递队列 q 的地址*/
    while(! IsEmpty(&q))        /*while 循环,如果队列不为空则循环*/
    {
        DeleteQueue(&q,&temp); /*出队,出队值存放于 temp 变量中*/
        printf("%c",temp);     /*输出 temp 变量值*/
    }
    printf("\n");
}
```

上述程序实现的是将从键盘上输入的字符串逆置后输出。

【例 3-7】 编程判断一个字符串是否是“回文”。“回文”是指一个字符序列以中间字符为基准两边字符完全相同,如字符序列“ACBDEDBCA”就是“回文”。

算法思想:判断一个字符序列是否是回文,就是把第一个字符与最后一个字符进行比较,第二个字符与倒数第二个字符比较,依次类推,第 i 个字符与第 $n-i$ 个字符比较。如果每次比较都相等,则为回文;如果某次比较不相等,就不是回文。因此,可以把字符序列分别入队列和栈,然后逐个出队列和出栈并比较出队列的字符和出栈的字符是否相等,若全部相等则该字符序列就是回文,否则就不是回文。

算法中的队列和栈可以采用任意存储结构,本例采用循环顺序队列和顺序栈来实现。程序中假设输入的都是英文字符而没有其他字符,对于输入其他字符情况的处理,读者可以自己去完成。

利用【例 3-6】中的队列逆置函数(返回值、函数参数及部分语句稍作修改)编写一个程序,完成判断回文序列的程序:

```
#include <stdio.h>
#include <seqqueue.h>
#include <seqstack.h>
SeqQueue proc(SeqQueue q)
{
    SeqStack s;
    char x;
    InitStack(&s);
    while(! IsEmpty(&q))
    {
        DeleteQueue(&q,&x);
        Push(&s,x);
    }
    while(! IsEmptys(&s))
    {
        Pop(&s,&x);
        EnterQueue(&q,x);
    }
    return q;
}
void main()
{
    SeqQueue x,y;
    char str[50],temp;
    char a,b;
    int i;
    InitQueue(&x);
```

```
    InitQueue(&y);
    printf("请输入队列内容:\n");
    gets(str);
    for(i=0;str[i]!='\0';i++)
    {
        temp=str[i];
        EnterQueue(&x,temp);
    }
    printf("\n");
    y=proc(x);
    while(! IsEmpty(&x))
    {
        DeleteQueue(&x,&a);
        DeleteQueue(&y,&b);
        if(a!=b)break;
    }
    if(IsEmpty(&x))printf("yes\n");
    else printf("no\n");
}
```

3.10 习 题

1)选择题

(1)栈和队列的共同特点是(　　)。

A. 只允许在端点处插入和删除元素

B. 都是先进后出

C. 都是先进先出

D. 没有共同点

(2)允许对队列进行的操作有(　　)。

A. 对队列中的元素排序　　B. 取出最近进队的元素

C. 在队头元素之前插入元素　　D. 删除队头元素

(3)用链接方式存储的队列,在进行插入操作时(　　)。

A. 仅修改头指针　　B. 头、尾指针都要修改

C. 仅修改尾指针　　D. 头、尾指针可能都要修改

(4)设指针变量 top 指向当前链式栈的栈顶,则删除栈顶元素的操作序列为(　　)。

A. top=top+1　　B. top=top-1

C. top->next=top　　D. op=top->next

(5)队列是一种(　　)的线性表。

A. 先进先出　　B. 先进后出

C. 只能插入　　D. 只能删除

(6)设输入序列 1,2,3,…,n 经过栈作用后,输出序列中的第一个元素是 n,则输出序列中的第 i 个输出元素是(　　)。

A. $n-i$　　B. $n-1-i$

C. $n-1-i$　　D. 不能确定

(7)设输入序列为 1,2,3,4,5,6,则通过栈的作用后可以得到的输出序列为(　　)。

A. 5,3,4,6,1,2　　B. 3,2,5,6,4,1

C. 3,1,2,5,4,6　　D. 1,5,4,6,2,3

(8)队列的删除操作是在(　　)进行。

A. 队首　　B. 队尾

C. 队前　　D. 队后

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

A. f->next=s;f=s;　　B. r->next=s;r=s;

C. s->next=r;r=s;　　D. s->next=f;f=s;

(10)若用一个大小为 6 的数组来实现循环队列,且 rear 和 front 的值分别是 0 和 3。当从队列中删除一个元素,再加入两个元素后,rear 和 front 的值分别是(　　)。

A. 2 和 4　　B. 1 和 5

C. 4 和 2　　D. 5 和 1

(11)有 6 个元素 6,5,4,3,2,1 的顺序进栈,下列(　　)不是合法的出栈序列。

A. 5 4 3 6 1 2　　B. 4 5 3 1 2 6

C. 3 4 6 5 2 1　　D. 2 3 4 1 5 6

2)填空题

(1)下面程序段的功能实现数据 x 进栈,要求在横线处填上正确的语句。

```
typedef struct{int s[100];int top;}sqstack;
void push(sqstack &stack,int x)
{
    if(stack.top==m-1)printf("overflow");
    else{_____;_____;}
}
```

(2)栈的插入和删除只能在栈顶进行,后进栈的元素必定先出栈,所以又把栈称为_____表;队列的插入和删除操作分别在队列的两端进行,先进队列的元素必定先出队列,所以又把队列称为_____表。

(3)从一个栈删除元素时,首先取出_____,然后再前移一位_____。

(4)在循环队列中,存储空间为 0~$n-1$,设队头指针 front 指向队头元素前一个空闲元素,队尾指针指向队尾元素,队满标志为 front=(rear+1)%n,那么队空标志为_____。

(5)对于栈只能在_____插入和删除元素。

(6)设输入元素的顺序是 A,B,C,D,通过栈的变换,在输出端可得到各种排列。若输出

序列的第一个元素为D,则输出序列为_____。

(7)队列中允许进行删除的一端为_____。

(8)我们通常把队列中允许插入的一端称为_____。

(9)队列的原则是_____。

(10)顺序存储的队列如果不采用循环方式,则会出现_____问题。

(11)栈的原则是_____。

(12)区分循环队列的满与空,只有两种方法,它们是_____和_____。

(13)已知一循环队列的存储空间为[$m..n$],其中$n>m$,队头和队尾指针分别为front和rear,则此循环队列判满的条件是_____。

(14)设有元素序列的入栈次序为($a1,a2,\cdots,an$),其出栈的次序为($ap1,ap2,\cdots,apn$),现已知$p1=n$,则$pi=$_____。

3)判断题

(1)不论是入队列操作还是入栈操作,在顺序存储结构上都需要考虑“溢出”情况。(　　)

(2)在循环顺序队列中插入新元素不需要判断队列是否满了。(　　)

(3)入栈操作和入队列操作在链式存储结构上实现时不需要考虑栈溢出的情况。(　　)

(4)栈是一种先进后出的线性表。(　　)

(5)消除递归不一定需要使用栈。(　　)

(6)栈是实现过程和函数等子程序所必需的结构。(　　)

(7)设栈采用顺序存储结构。若已有$i-1$个元素进栈,则第i个元素进栈时,进栈算法的时间复杂度为$O(i)$。(　　)

(8)在链队列中,即使不设置尾指针也能进行入队操作。(　　)

(9)设带尾指针的循环链表表示队列,则入队和出队算法的时间复杂度均为$O(1)$。(　　)

(10)栈和队列都是线性表,只是在插入和删除时受到了一些限制。(　　)

(11)队列在程序调用时必不可少,因此递归离不开队列。(　　)

(12)栈可以作为实现程序设计语言过程调用时的一种数据结构。(　　)

(13)栈又称后进先出表,队列又称为先进先出表。(　　)

(14)栈和队列都是限制存取点的线性结构。(　　)

4)简答题

(1)简述栈和队列与线性表的关系。

(2)在一般的顺序队列中,什么是假溢出?怎样解决假溢出问题?

(3)简述栈和队列这两种数据结构的相同点和不同点。

5)算法设计题

(1)设有两个栈S_1和S_2都采用顺序栈方式,并且共享一个存储区[0..maxsize-1],为了尽量利用空间,减少溢出的可能,可采用栈顶相向,迎面增长的存储方式。试设计S_1和S_2有关入栈和出栈的操作算法。

(2)设表达式以字符形式已存入数组 $E[n]$中,“#”为表达式的结束符,试写出判断表达式中括号(“(”和“)”)是否配对的C语言描述算法:EXYX(E)。(算法中可调用栈操作的基本算法。)

(3)请利用两个栈 $S1$ 和 $S2$ 来模拟一个队列。已知栈的3个运算定义如下:PUSH(ST,x)表示元素 x 入ST栈;POP(ST,x):ST栈顶元素出栈,赋给变量 x;Sempty(ST)表示判ST栈是否为空。那么如何利用栈的运算来实现该队列的3个运算:enqueue表示插入一个元素入队列;dequeue表示删除一个元素出队列;queue_empty表示判队列为空。(请写明算法的思想及必要的注释。)

(4)已知 Q 是一个非空队列,S 是一个空栈。仅用队列和栈的ADT函数和少量工作变量,使用Pascal或C语言编写一个算法,将队列 Q 中的所有元素逆置。栈的ADT函数有:

```
makeEmpty(s:stack);                    置空栈
push(s:stack;value:datatype);          新元素 value 进栈
pop(s:stack):datatype;                 出栈,返回栈顶值
isEmpty(s:stack):Boolean;              判栈空否
```

队列的ADT函数有:

```
enQueue(q:queue:value:datatype);       元素 value 进队
deQueue(q:queue):datatype;             出队列,返回队头值
isEmpty(q:queue):boolean;              判队列空否
```

(5)已知求两个正整数 m 与 n 的最大公因子的过程用自然语言可以表述为反复执行如下动作:第一步,若 n 等于零,则返回 m;第二步,若 m 小于 n,则 m 与 n 相互交换;否则,保存 m,然后将 n 送入 m,将保存的 m 除以 n 的余数送入 n。

将上述过程用递归函数表达出来(设求 x 除以 y 的余数可以用 x MOD y 形式表示)。写出求解该递归函数的非递归算法。