第一部分

数据结构

需要全本请访问 www.ertongbook.com

第1章

绪　论

计算机科学是一门研究数据表示和数据处理的科学。数据是信息在计算机中的表示形式，它是计算机可以直接处理的最基本和最重要的对象。无论是进行科学计算或数据处理、过程控制，还是对文件的存储和检索等，都是对数据进行加工处理的过程。因此，要设计出一个结构好效率高的数据加工处理程序，必须研究数据的组织特性、数据间的相互关系及其对应的存储表示，并利用这些特性和关系设计出相应的算法和程序。数据结构这门学科就是研究这些问题的。

1.1　数据结构的概念

数据结构是计算机科学与技术专业的专业基础课，是十分重要的核心课程。所有的计算机系统软件和应用软件都要用到各种类型的数据结构。因此，要想更好地运用计算机来解决实际问题，仅掌握几种计算机程序设计语言是难以应付众多复杂的课题的。要想有效地使用计算机，充分发挥计算机的性能，还必须学习和掌握好数据结构的有关知识。打好“数据结构”这门课程的扎实基础,对于学习计算机专业的其他课程,如操作系统、编译原理、数据库管理系统、软件工程、人工智能等都是十分有益的。

1.1.1　为什么要学习数据结构

在计算机发展的初期，人们使用计算机的目的主要是处理数值计算问题。当我们使用计算机来解决一个具体问题时，一般需要经过下列几个步骤：首先要从该具体问题抽象出一个适当的数学模型,然后设计或选择一个解此数学模型的算法,最后编出程序进行调试、测试,直至得到最终的解答。例如，求解梁架结构中应力的数学模型的线性方程组，该方程组可以使用迭代算法来求解。

由于当时所涉及的运算对象是简单的整型、实型或布尔类型数据，所以程序设计者的主要精力是集中于程序设计的技巧上，而无须重视数据结构。随着计算机应用领域的扩大和软、硬件的发展，非数值计算问题越来越显得重要。据统计，当今处理非数值计算性问题占用了90%以上的机器时间。这类问题涉及到的数据结构更为复杂，数据元素之间的相互关系一般无法用数学方程式加以描述。因此，解决这类问题的关键不再是数学分析和计算方法，而是要

设计出合适的数据结构，才能有效地解决问题。下面所列举的就是属于这一类的具体问题。

【例 1】 学生信息检索系统。当我们需要查找某个学生有关情况的时候，或者想查询某个专业或年级的学生有关情况的时候，只要我们建立了相关的数据结构，按照某种算法编写了相关程序，就可以实现计算机自动检索。由此，可以在学生信息检索系统中建立一张按学号顺序排列的学生信息表和分别按姓名、专业、年级顺序排列的索引表，如图 1.1 所示。这四张表便是学生信息检索的数学模型，计算机的主要操作便是按照某个特定要求（如给定姓名）对学生信息表进行查询。

诸如此类的还有电话自动查号系统、考试查分系统、仓库库存管理系统等。在这类文档管理的数学模型中，计算机处理的对象之间通常存在着的是一种简单的线性次序关系，这类数学模型可称为线性数据结构。

学号	姓名	性别	专　业	年　级
980001	吴承志	男	计算机科学与技术	1998 级
980002	李淑芳	女	信息与计算科学	1998 级
990301	刘　丽	女	数学与应用数学	1999 级
990302	张会友	男	信息与计算科学	1999 级
990303	石宝国	男	计算机科学与技术	1999 级
000801	何文颖	女	计算机科学与技术	2000 级
000802	赵胜利	男	数学与应用数学	2000 级
000803	崔文靖	男	信息与计算科学	2000 级
010601	刘　丽	女	计算机科学与技术	2001 级
010602	魏永鸣	男	数学与应用数学	2001 级

(a)学生信息表

崔文靖	8
何文颖	6
李淑芳	2
刘　丽	3,9
石宝国	5
魏永鸣	10
吴承志	1
赵胜利	7
张会有	4

(b)姓名索引表

计算机科学与技术	1,5,6,9
信息与计算科学	2,4,8
数学与应用数学	3,7,10

(c)专业索引表

2000 级	6,7,8
2001 级	9,10
1998 级	1,2,3
1999 级	4,5

(d)年级索引表

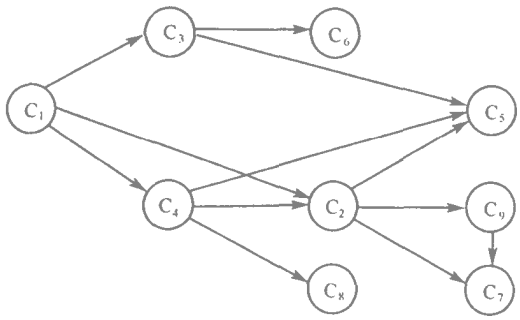
图 1.1　学生信息查询系统中的数据结构

【例 2】 教学计划编排问题。一个教学计划包含许多课程，在教学计划包含的许多课程之间，有些必须按规定的先后次序进行排列，有些则没有次序要求。即有些课程之间有先修和后续的关系，有些课程可以任意安排次序。这种各个课程之间的次序关系可用一个称作图的

数据结构来表示,如图 1.2所示。有向图中的每个顶点表示一门课程，如果从顶点 v_i 到 v_j 之间存在有向边 $<v_i, v_j>$,则表示课程 i必须先于课程 j进行。

课程编号	课程名称	先修课程
C_1	计算机导论	无
C_2	数据结构	C_1, C_4
C_3	汇编语言	C_1
C_4	C程序设计语言	C_1
C_5	计算机图形学	C_2, C_3, C_4
C_6	接口技术	C_3
C_7	数据库原理	C_2, C_9
C_8	编译原理	C_4
C_9	操作系统	C_2

(a)计算机专业的课程设置



(b)表示课程之间优先关系的有向图

图 1.2　教学计划编排问题的数据结构

由以上两个例子可见，描述这类非数值计算问题的数学模型不再是数学方程，而是诸如表、图之类的数据结构。因此，可以说数据结构课程主要是研究非数值计算的程序设计问题中所出现的计算机操作对象以及它们之间的关系和操作的学科。

学习数据结构的目的是为了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。

1.1.2　有关概念和术语

在系统地学习数据结构知识之前，首先应对一些基本概念和术语赋予确切的含义。

数据(Data)：是能够被计算机识别、存储和加工处理的信息的载体。它是计算机程序加工的原料。计算机科学中，所谓数据就是计算机加工处理的对象，它可以是数值数据，也可以是非数值数据。数值数据是一些整数、实数或复数，主要用于工程计算、科学计算和商务处理等；非数值数据包括字符、文字、图形、图像、语音等。

数据元素(Data Element)：是数据的基本单位。在不同的情况下，数据元素又可称为元素、结点、顶点、记录等。例如,学生信息检索系统中学生信息表中的一个记录,教学计划编排

需要全本请访问 www.ertongbook.com

问题中的一个顶点等，都被称为一个数据元素。

数据对象(Data Object)或数据元素类(Data Element Class)L 是具有相同性质的数据元素的集合。在某个具体问题中,数据元素都具有相同的性质(元素值不一定相等),属于同一数据对象(数据元素类),数据元素是数据元素类的一个实例。例如,在交通咨询系统的交通网中,所有的顶点是一个数据元素类,顶点 A 和顶点 B 各自代表一个城市，是该数据元素类中的两个实例,其数据元素的值分别为 A 和 B。

数据结构(Data Structure) 指的是数据之间的相互关系,即数据的组织形式。虽然至今没有一个关于数据结构的标准定义，但它一般包括以下三方面的内容：

(1)数据元素之间的逻辑关系，也称为数据的逻辑结构(Logical Structure)。

(2)数据元素及其关系在计算机存储器内的表示，也称为数据的存储结构 Storage Structure).

(3)数据的运算，即对数据施加的操作。

数据的逻辑结构是从逻辑关系上描述数据，它与数据的存储无关，是独立于计算机的。因此，数据的逻辑结构可以看作是从具体问题抽象出来的数学模型。数据的存储结构是逻辑结构在计算机存储器里的实现(亦称为映象),它是依赖于计算机的,对机器语言而言,存储结构是具体的，但我们只在高级语言的层次上来讨论存储结构。数据的运算是定义在数据的逻辑结构上的,每种逻辑结构都有一个运算的集合。例如,最常用的运算有 检索,插入,删除,更新，排序等。这些运算实际上是在抽象的数据上所施加的一系列抽象的操作，所谓抽象的操作,是指我们只知道这些操作是‘做什么”,而无须考虑‘如何做”。只有确定了存储结构之后，我们才考虑如何具体实现这些运算。本书中讨论的数据运算，均在高级语言的层次上用相应的算法来实现。

在不会产生混淆的前提下，我们常将数据的逻辑结构简称为数据结构。

数据的逻辑结构有以下两大类：

1) 线性结构

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

2) 非线性结构

非线性结构的逻辑特征：一个结点可能有多个直接前趋和直接后继。

数据的存储结构可用以下四种基本的存储方法得到：

1) 顺序存储方法：把逻辑上相邻的结点存储在物理位置上相邻的存储单元里，结点间的逻辑关系由存储单元的邻接关系来体现。

2) 链接存储方法：该方法不要求逻辑上相邻的结点在物理位置上也相邻，结点间的逻辑关系由附加的指针字段表示。

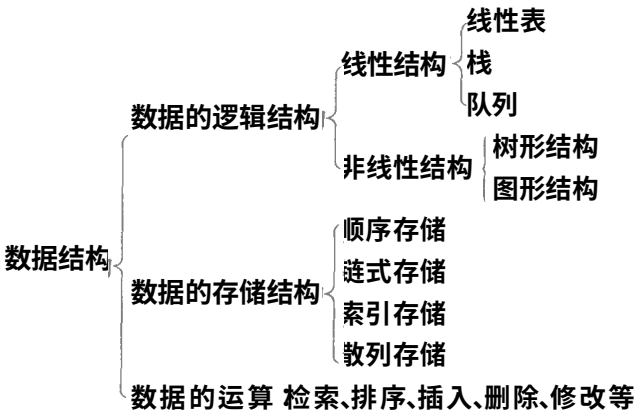
3) 索引存储方法：在存储结点信息的同时，还建立附加的素引表。

4) 散列存储方法：根据结点的关键字直接计算出该结点的存储地址。

同一逻辑结构可有不同的存储结构；同理在给定了数据的逻辑结构和存储结构之后，按定义的运算集合及其运算的性质不同，也可能导致完全不同的数据结构。例如，线性表是一种逻辑结构,若采用顺序方法的存储表示,则为顺序表 若采用链接方法的存储表示,则为链表 若采用散列方法的存储表示，则为散列表；若对线性表上的插入、删除运算限制在表的一端进行，则为栈 若对插入限制在表的一端进行,而删除限制在表的另一端进行,则为队列。

综上所述，我们可以将数据结构定义为：按某种数据关系组织起来的一批数据，应用计算机语言，可按一定的存储表示方式把它们存储在计算机的存储器中，并在该数据中定义了一个运算的集合。

上述这些基本概念的相互关系归纳如下：



1.2 算法和算法分析

算法与数据结构的关系紧密，在算法设计时，先要确定相应的数据结构，而在讨论某一种数据结构时，也必然会涉及实现一定功能的算法。下面就从算法特性、算法描述、算法性能分析与度量等三个方面对算法进行介绍。

1.2.1 算法特性

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

(1)有穷性：一个算法必须在有穷步之后结束，即必须在有限时间内完成。

(2)确定性 算法的每一步必须有确切的定义,无二义性。

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

(4)输入性：一个算法具有零个或多个输入,这些输入取自特定的数据对象集合。

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

算法的含义与程序十分相似，但又有区别。一个程序不一定满足有穷性。例如操作系统，只要整个系统不遭破坏，它将永远不会停止，即使没有作业需要处理，它仍处于动态等待中。因此，操作系统不是一个算法。另一方面，程序中的指令必须是机器可执行的，而算法中的指令则无此限制。算法代表了对问题的求解方法和步骤，而程序则是算法在计算机上的特定的实现。一个算法若用程序设计语言来描述，则它就是一个程序。

算法与数据结构是相辅相成的。解决某一特定类型问题的算法可以选定不同的数据结构，而且选择恰当与否直接影响算法的效率；反之，一种数据结构的优劣由各种算法的执行效率来体现。

要设计一个好的算法通常要考虑以下的要求：

(1)正确：算法的执行结果应当满足预先规定的功能和性能要求。

(2)可读：一个算法应当思路清晰、层次分明、简单明了、易读易懂。

(3)健壮：当输入不合法数据时，应能作适当处理，不至引起严重后果。

(4)高效：有效使用存储空间和有较高的时间效率。

1.2.2 算法描述

算法可以使用各种不同的方法来描述。

最简单的方法是使用自然语言。用自然语言来描述算法的优点是简单且便于人们对算法的阅读。缺点是不够严谨。

可以使用程序流程图、N-S 图等算法描述工具。其特点是描述过程简洁、明了。

用以上两种方法描述的算法不能够直接在计算机上执行，若要将它转换成可执行的程序还有一个编程的问题。

可以直接使用某种程序设计语言来描述算法，不过直接使用程序设计语言并不容易，而且不太直观，常常需要借助于注释才能使人看明白。

为了解决理解与执行这两者之间的矛盾，人们常常使用一种称为伪码语言的描述方法来进行算法描述。伪码语言介于高级程序设计语言和自然语言之间，它忽略高级程序设计语言中一些严格的语法规则与描述细节，因此它比程序设计语言更容易描述和被人理解，而比自然语言更接近程序设计语言。它虽然不能直接执行但很容易被转换成高级语言。

由于绝大部分读者都学过 C 语言，故本书的算法均用 C++ 来描述。

1.2.3 算法性能分析与度量

求解同一个问题，可以有许多不同的算法，那么如何来评价这些算法的好坏呢？

显然，选用的算法首先应该是“正确的”。此外，主要考虑如下三点：

(1)执行算法所耗费的时间；

(2)执行算法所耗费的存储空间，其中主要考虑辅助存储空间；

(3)算法应易于理解，易于编码，易于调试等等。

当然我们希望选用一个所占存储空间小、运行时间短、其他性能也好的算法。然而，实际上很难做到十全十美。原因是上述要求有时相互抵触。要节约算法的执行时间往往要以牺牲更多的空间为代价；而为了节省空间又可能要以更多的时间作代价。因此我们只能根据具体情况有所侧重。若该程序使用次数较少，则力求算法简明易懂，易于转换为上机的程序。对于反复多次使用的程序，应尽可能选用快速的算法。若待解决的问题数据量极大，机器的存储空间较小，则相应算法主要考虑如何节省空间。本书主要讨论算法的时间特性，偶尔也讨论空间特性。

我们可以从一个算法的时间复杂度与空间复杂度来评价算法的优劣。

1. 时间复杂度

一个算法所耗费的时间，应该是该算法中每条语句的执行时间之和，而每条语句的执行时间是该语句的执行次数(也称为频度(Frequency Count))与该语句一次执行所需时间的乘积。当我们将一个算法转换成程序并在计算机上执行时，其运行所需要的时间取决于下列因素：

(1)硬件的速度。例如使用奔Ⅲ机还是使用奔Ⅳ机。

(2)书写程序的语言。实现语言的级别越高，其执行效率就越低。

(3)编译程序所生成目标代码的质量。对于代码优化较好的编译程序其所生成的程序质量较高。

(4)问题的规模。例如,求 100 以内的素数与求 1000 以内的素数其执行时间必然是不同的。

显然，在各种因素都不能确定的情况下，很难比较出算法的执行时间。也就是说，用执行算法的绝对时间来衡量算法的效率是不合适的。因此我们假设每条语句的一次执行所需的时间均是单位时间，这样，一个算法的时间耗费就是该算法中所有语句的频度之和。于是，我们就可独立于机器的软、硬件系统来分析算法的时间耗费，这样一个特定算法的运行工作量的大小就只依赖于问题的规模（通常用正整数 n 表示),或者说它是问题规模的函数。

【例 3】 求两个 n 阶方阵的乘积 C=A×B,其算法如下：

```
        #define   n 自然数
        MATRIXMLT(float A[ ][n],B[ ][n],C[ ][n])
        {int   i,j,k;
(1)       for( i=0;i<n;i++)                                  n+1
(2)         for(j=0;j<n;j++)                                 n(n+1)
(3)           {C[i][j]=0   ;                                 n²
(4)             for(k=0;k<n;k++)                             n²(n+1)
(5)               C[i][j]= C[i][j]+ A[i][k]*B[k][j]   ;      n³
                }
        } /* MATRIXMLT */
```

其中右边列出的是各语句的频度。语句（1)的循环控制变量 i 要增加到 n,测试 i≥n 成立才会终止,故它的频度是 n+1,但是它的循环体却只能执行 n 次。语句(2)作为语句(1)循环体内的语句应该执行 n 次,但语句(2)本身要执行 n+1 次,所以语句(2)的频度是 n(n+1)。同理可得到语句(3),(4)和(5)的频度分别是 n^2,$n^2(n+1)$和 n^3。该算法中所有语句的频度之和(即算法的时间耗费)为

$$T(n)=2n^3+3n^2+2n+1$$

由此可知,算法 MATRIXMLT 的时间耗费 T(n)是矩阵阶数 n 的函数。

一般地，我们将算法所要求解问题的输入量（或初始数据量）称为问题的规模（Size,大小),并用一个整数表示。例如,矩阵乘积问题的规模是矩阵的阶数,而一个图论问题的规模则是图中的顶点数或边数。一个算法的时间复杂度（Time Complexity,也称时间复杂性)T(n)则是该算法的时间耗费，它是该算法所求解问题规模 n 的函数。许多时候要精确地计算 T(n)是困难的，我们引入渐近时间复杂度在数量上估计一个算法的执行时间，也能够达到分析算法的目的。当问题的规模 n 趋向无穷大时,我们把时间复杂度 T(n)的数量级(阶)称为算法的渐近时间复杂度。

例如,算法 MATRIXMLT 的时间复杂度 T(n),当 n 趋向无穷大时,显然有

$$\lim_{n\to\infty} T(n)/n^3=\lim_{n\to\infty}(2n^3+3n^2+2n+1)/n^3=2$$

这表明,当 n 充分大时,T(n)和 n^3 之比是一个不等于零的常数,即 T(n)和 n^3 是同阶的,

或说 $T(n)$ 和 n^3 的数量级相同，可记作 $T(n)=O(n^3)$。我们称 $T(n)=O(n^3)$是算法 MATRIXMLT 的渐近时间复杂度。其中记号“O”是数学符号，其严格的数学定义是：

定义(大O记号)如果存在三个正常数 c_1, c_2 和 n_0，使得对所有的 $n \geqslant n_0$，有：

$$c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$$

则有：

$$f(n) = O(g(n))$$

在算法分析时，往往对算法的时间复杂度和渐近时间复杂度不予区分，而经常是将渐近时间复杂度 $T(n)=O(f(n))$ 简称为时间复杂度，其中的 $f(n)$一般是算法中频度最大的语句频度。

时间复杂度相应的数量级(阶)按递增排列有：

常数阶 $O(1)$

对数阶 $O(\log_2 n)$

线性阶 $O(n)$

线性对数阶 $O(n\log_2 n)$

平方阶 $O(n^2)$

立方阶 $O(n^3)$

……

k 次方阶 $O(n^k)$

指数阶 $O(2^n)$

【例 4】分析以下程序段的时间复杂度。

```
i=1;                                  (1)
while(i<=n) i=i*2;                    (2)
```

其中语句(1) 的频度是 1，设语句(2) 的频度是 $f(n)$，则有：

$$2^{f(n)-1} \leqslant n$$

即 $f(n) \leqslant \log_2 n + 1$，取最大值 $f(n)=\log_2 n + 1$

则该程序段的时间复杂度为

$$T(n) = 1 + f(n) = 1 + 1 + \log_2 n = O(\log_2 n)$$

2. 空间复杂度

一个程序的空间复杂度(Space complexity) 是指程序运行从开始到结束所需的最大存储量。

程序的一次运行是针对所求解的问题的某一特定实例而言的。例如，求解排序问题的排序算法的每次执行是对一组特定个数的元素进行排序。对该组元素的排序是排序问题的一个实例。元素个数可视为该实例的特征。

程序运行所需的存储空间包括以下两部分：

(1)固定部分。这部分空间与所处理数据的大小和个数无关，或者称与问题的实例的特征无关。主要包括程序代码、常量、简单变量、定长成分的结构变量所占的空间。

(2)可变部分。这部分空间大小与算法在某次执行中处理的特定数据的大小和规模有关。例如 100 个数据元素的排序算法与 1000 个数据元素的排序算法所需的存储空间显然是不同的。

习　题

一、简答题

1. 简述下列术语：

数据、数据元素、数据对象、数据结构、逻辑结构、存储结构、时间复杂度、渐近时间复杂度。

2. 试举一个数据结构的例子，并叙述其逻辑结构、存储结构、运算三方面的内容。

3. 何谓算法？试叙述算法的特性及算法必须满足的条件。

二、写出下面程序段的时间复杂度

1.
```
i=1;k=0;
while(i<n)
    {k=k+10*i;i++;
    }
```

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

3.
```
x=n; /*n>1*/
while (x>=(y+1)*(y+1)) y++;
```

4.
```
x=99;y=100;
while(y>0)
    if (x>100){x=x-10;y--;}
    else x++;
```

需要全本请访问 www.ertongbook.com

第2章

线性表

线性表是最简单、最基本、也是最常用的一种线性结构。它有两种存储方法：顺序存储和链式存储，它的主要基本操作是插入、删除和检索等。

2.1 线性表的逻辑结构

2.1.1 线性表的定义

线性表是一种线性结构。线性结构的特点是数据元素之间是一种线性关系，即数据元素“一个接一个的排列”。在一个线性表中数据元素的类型是相同的，或者说线性表是由同一类型的数据元素构成的线性结构。在实际问题中线性表的例子是很多的，如学生情况信息表是一个线性表，表中数据元素的类型为学生类型；一个字符串也是一个线性表，表中数据元素的类型为字符型，等等。

综上所述，线性表定义如下：

线性表是具有相同数据类型的 $n(n\geqslant 0)$ 个数据元素的有限序列，通常记为：

$$(a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$$

其中 n 为表长，$n=0$ 时称为空表。

表中相邻元素之间存在着顺序关系。将 a_{i-1} 称为 a_i 的直接前趋，a_{i+1} 称为 a_i 的直接后继。就是说 对于 a_i 当 $i=2,\cdots,n$ 时，有且仅有一个直接前趋 a_{i-1} 当 $i=1,2,\cdots,n-1$ 时，有且仅有一个直接后继 a_{i+1}；而 a_1 是表中第一个元素，它没有前趋；a_n 是最后一个元素，它没有后继。

2.1.2 线性表的基本操作

在第一章中提到，数据结构的运算是定义在逻辑结构层次上的，而运算的具体实现是建立在存储结构上的。

线性表上的基本操作有：

（1）线性表初始化：Init_List(L)

初始条件 表 L 不存在

操作结果 构造一个空的线性表

(2) 求线性表的长度 :Length_List(L)

初始条件 表 L 存在

操作结果 :返回线性表中的所含元素的个数

(3) 取表元 :Get_List(L,i)

初始条件 表 L 存在且 1≤i≤Length_List(L)

操作结果 返回线性表 L 中的第 i 个元素的值或地址

(4) 按值查找 Locate_List(L,x),x 是给定的一个数据元素。

初始条件 线性表 L 存在

操作结果 在表 L 中查找值为 x 的数据元素，其结果返回在 L 中首次出现的值为 x 的那个元素的序号或地址,称为查找成功 否则,在 L 中未找到值为 x 的数据元素，返回一特殊值表示查找失败。

(5) 插入操作 :Insert_List(L,i,x)

初始条件 线性表 L 存在，插入位置正确(1≤i≤n+1,n 为插入前的表长)。

操作结果 在线性表 L 的第 i 个位置上插入一个值为 x 的新元素,这样使原序号为 i, i+1, …, n 的数据元素的序号变为 i+1,i+2, …, n+1,插入后表长 = 原表长 + 1。

(6) 删除操作 :Delete_List(L,i)

初始条件 线性表 L 存在,1≤i≤n。

操作结果 在线性表 L 中删除序号为 i 的数据元素,删除后使序号为 i+1, i+2,…, n 的元素变为序号为 i, i+1,…,n-1,新表长 = 原表长 -1。

需要说明的是：

(1) 某数据结构上的基本运算，不是它的全部运算，而是一些常用的基本的运算，而每一个基本运算在实现时也可能根据不同的存储结构派生出一系列相关的运算来。比如线性表的查找在链式存储结构中还会有按序号查找；再如插入运算，也可能是将新元素 x 插入到适当位置上等等，不可能也没有必要全部定义出它的运算集，读者掌握了某一数据结构上的基本运算后，其他的运算可以通过基本运算来实现，也可以直接去实现。

(2) 在上面各操作中定义的线性表 L 仅仅是一个抽象在逻辑结构层次的线性表，尚未涉及到它的存储结构，因此每个操作在逻辑结构层次上尚不能用具体的某种程序语言写出具体的算法，而算法的实现只有在存储结构确立之后。下面两节就是讨论在特定存储结构下算法的实现。

2.2 线性表的顺序存储及运算实现

2.2.1 顺序表

线性表的顺序存储是指在内存中用地址连续的一块存储空间顺序存放线性表的各元素，用这种存储形式存储的线性表称其为顺序表。因为内存中的地址空间是线性的，因此，用物理

上的相邻表示数据元素之间的逻辑相邻关系是既简单又自然的。如图 2.1 所示。设 a_1 的存储地址为 $Loc(a_1)$，每个数据元素占 d 个存储地址，则第 i 个数据元素的地址为：

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

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

在程序设计语言中，一维数组在内存中占用的存储空间就是一组连续的存储区域，因此，用一维数组来表示顺序表的数据存储区域是再合适不过的。考虑到线性表的运算有插入、删除等运算，即表长是可变的，因此，数组的容量需设计得足够大。假设用 data[MAXSIZE] 来表示，其中 MAXSIZE 是一个根据实际问题定义的足够大的整数，线性表中的数据从 data[0] 开始依次顺序存放。由于当前线性表中的实际元素个数可能未达到 MAXSIZE 多个，因此需用一个变量 last 记录当前线性表中最后一个元素在数组中的位置，即 last 起一个指针的作用，始终指向线性表中最后一个元素。表空时 last = -1。这种存储思想的具体描述可以是多样的。如可以是：

```
datatype data[MAXSIZE] ;
int last;
```

这样表示的顺序表如图 2.1 所示。表长为 last + 1，数据元素分别存放在 data[0]到 data[last]中。这样使用简单方便，但有时不便管理。

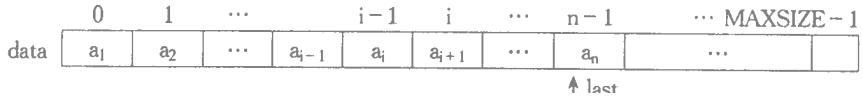


图 2.1 线性表的顺序存储示意图

从结构性上考虑，通常将 data 和 last 封装成一个结构作为顺序表的类型：

```
typedef struct
        { datatype data[MAXSIZE];
          int last;
        } SeqList;
```

定义一个顺序表：SeqList L ；

这样表示的线性表如图 2.2(a) 所示。表长 = L.last + 1，线性表中的数据元素 a_1 至 a_n 分别存放在 L.data[0]至 L.data[L.last]中。

由于我们后面的算法用 C 语言描述，根据 C 语言中的一些规则，有时定义一个指向 SeqList 类型的指针更为方便：

```
SeqList *L ;
```

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

L 中存放的是顺序表的地址，这样表示的线性表如图 2.2(b)所示。表长表示为(*L).last+1 或 L->last+1，线性表的存储区域为 L->data，线性表中数据元素的存储空间为：

$$L\rightarrow data[0] \sim L\rightarrow data[L\rightarrow last].$$

在以后的算法中多用这种方法表示，读者在读算法时注意相关数据结构的类型说明。

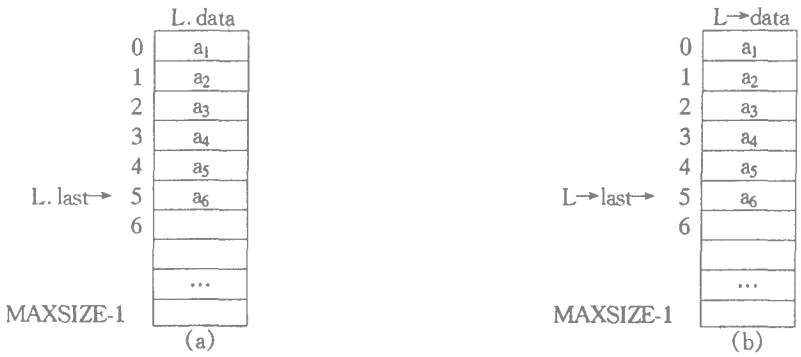


图 2.2 线性表的顺序存储示意图

2.2.2 顺序表上基本运算的实现

1.顺序表的初始化

顺序表的初始化即构造一个空表，这对表是一个加工型的运算，因此，将 L 设为指针参数,首先动态分配存储空间,然后,将表中 last 指针置为 -1,表示表中没有数据元素。

初始化算法 2.1:

```
SeqList * Init_SeqList( )
{ SeqList * L;
  L= (SeqList * )malloc(sizeof(SeqList)) ;
  L→last= -1 ;return L;
}
```

2. 插入运算

线性表的插入是指在表的第 i 个位置上插入一个值为 x 的新元素，插入后使原表长为 n 的表：

$$(a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$$

成为表长为 n+1 的表：

$$(a_1,a_2,\cdots,a_{i-1},x,a_i,a_{i+1},\cdots,a_n)。$$

i 的取值范围为 $1\leqslant i\leqslant n+1$。插入运算的示意图如图 2.3 所示。

顺序表上完成这一运算则通过以下步骤进行：

(1) 将 $a_i\sim a_n$ 顺序向下移动(移动次序是从 a_n 到 a_i),为新元素让出位置；

(2) 将 x 置入空出的第 i 个位置；

(3) 修改 last 指针(相当于修改表长),使之仍指向最后一个元素。

插入元素算法 2.2:

```
int  Insert_SeqList(SeqList * L,int i,datatype x)
{  int j;
   if (L→last= =MAXSIZE-1)
        {  printf( "表满 "); return(-1); }  /* 表空间已满,不能插入 */
   if (i<1 || i>L→last+2)       /* 检查插入位置的正确性 */
        {  print( "位置错 ");return(0) ;}
```

```
    for(j=L→last;j>=i-1;j--)
        L→data[j+1]=L→data[j];      /* 结点移动 */
    L→data[i-1]=x;                  /* 新元素插入 */
    L→last++;                       /* last仍指向最后元素 */
    return(1);                      /* 插入成功,返回 */
}
```

本算法中注意以下问题：

(1) 顺序表中数据区域有 MAXSIZE 个存储单元，所以在向顺序表中做插入时先检查表空间是否满了，在表满的情况下不能再做插入，否则产生溢出错误。

(2) 要检验插入位置的有效性,这里 i 的有效范围是：$1 \leqslant i \leqslant n+1$,其中 n 为原表长。

(3) 注意数据的移动次序和方向。



图 2.3 顺序表中的插入

插入算法的时间复杂度分析如下：

顺序表上的插入运算，时间主要消耗在数据的移动上，在第 i 个位置上插入 x,从 a_i 到 a_n 都要向下移动一个位置,共需要移动 $n-i+1$ 个元素,而 i 的取值范围为：$1 \leqslant i \leqslant n+1$,即有 $n+1$ 个位置可以插入。设在第 i 个位置上做插入的概率为 p_i,则平均移动数据元素的次数：

$$E_{in} = \sum_{i=1}^{n+1} p_i (n \quad i+1)$$

设：$p_i = 1/(n+1)$,即为等概率情况,则：

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

这说明：在顺序表上做插入操作需移动表中一半的数据元素。显然时间复杂度为 $O(n)$。

3. 删除运算

线性表的删除运算是指将表中第 i 个元素从线性表中去掉，删除后使原表长为 n 的线性表：

$$(a_1, a_2, \cdots, a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$$

成为表长为 n－1 的线性表：

$$(a_1,a_2,\cdots,a_{i-1},a_{i+1},\cdots,a_n)$$

i 的取值范围为：$1\leqslant i\leqslant n$。删除运算的示意图如图 2.4 所示。

顺序表上完成这一运算的步骤如下：

(1) 将 $a_{i+1}\sim a_n$ 顺序向上移动，a_i 元素被 a_{i+1} 覆盖；

(2) 修改 last 指针（相当于修改表长）使之仍指向最后一个元素。

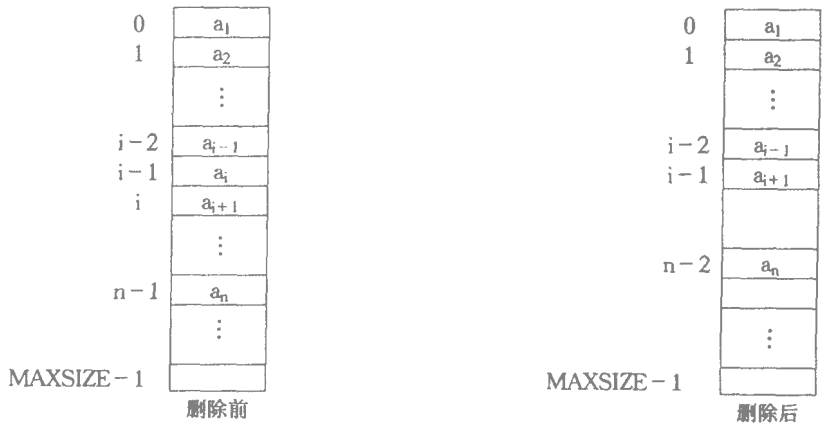


图 2.4　顺序表中的删除

删除元素算法 2.3：

```
int Delete_SeqList(SeqList *L,int i)
  { int j;
   if(i<1 || i>L→last+1) /*  检查空表及删除位置的合法性 */
          { printf("不存在第 i 个元素");return(0); }
   for(j=i;j<=L→last;j++)
          L→data[j-1]=L→data[j]; /*  向上移动 */
   L→last--;
   return(1);          /* 删除成功 */
   }
```

本算法注意以下问题：

(1) 删除第 i 个元素，i 的取值为 $1\leqslant i\leqslant n$，否则第 i 个元素不存在，因此，要检查删除位置的有效性。

(2) 当表空时不能做删除，因表空时 L→last 的值为 －1，条件(i<1 || i>L→last+1)也包括了对表空的检查。

(3) 删除 a_i 之后，该数据已不存在，如果需要，先取出 a_i，再做删除。

删除算法的时间复杂度分析如下：

与插入运算相同，其时间主要消耗在移动表中元素上，删除第 i 个元素时，其后面的元素 $a_{i+1}\sim a_n$ 都要向上移动一个位置，共移动了 n－i 个元素，所以平均移动数据元素的次数：

$$E_{de}=\sum_{i=1}^{n}p_i(n-i)$$

需要全本请访问 www.ertongbook.com