# 第1章 概　论

这一章中，我们首先给出传统的数据结构的概念，继而介绍数据抽象和抽象数据类型的概念，然后给出基于抽象数据类型的数据结构描述方法，最后介绍算法的效率和算法分析的基本方法。

## 1.1 什么是数据结构

数据结构是计算机科学与技术领域广泛使用的术语，然而，究竟什么是数据结构，在计算机科学界至今没有标准的定义。

随着计算机科学与技术的发展，计算机的应用已远远超出了单纯进行科学计算的范围。今天，信息技术作为现代技术的标志已成为世界各国经济增长的主要动力。计算机技术已渗透到国民经济的各行各业和人们日常生活的方方面面。计算机已经从传统的应用领域，如工业控制、情报检索、企业管理、商务处理、图形图像、人工智能等数据处理领域，发展到了电子政务、电子商务、办公自动化、企业资源管理系统、电子图书馆、远程教育、远程医疗等诸多领域。

现实世界各领域中的大量信息都必须转换成数据才能在计算机中存储、处理。数据是信息的载体。应用程序处理各种各样的数据。笼统地说，所谓数据(data)，就是计算机加工处理的对象。数据一般分两类：数值数据(numerical data)和非数值数据(non-numerical data)。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、表格等。

数据结构主要是为研究和解决如何使用计算机处理非数值问题而产生的理论、技术和方法。它表达数据的构造形式，即一个数据由哪些成分数据构成，以什么方式构成，具有什么结构。在这里，我们称组成数据的成分数据为数据元素(data element)。一般地，数据元素可以是简单类型的，如整数、实数、字符等，也可以是结构类型，如记录。若把每个学生的记录看成一个数据元素，它包括学号、姓名、性别等数据项(data item)，一个班的学生记录组成了图 1－1 所示的学生情况表。表是一个数据结构。

从数学概念上讲，一个数据结构(data structure)是由数据元素依据某种逻辑联系组织起来的。对数据元素间的逻辑关系的描述被称为数据的逻辑结构(logical structure)。

根据数据结构中数据元素之间的结构关系的不同特征，通常将数据结构分为如下四种基本结构：

| 学 号 | 姓 名 | 性 别 | 其他信息 |
| --- | --- | --- | --- |
| B02040101 | 王小红 | 女 | … |
| B02040102 | 林 悦 | 女 | … |
| B02040103 | 陈菁菁 | 女 | … |
| B02040104 | 张可可 | 男 | … |
| … | … | … | … |
| ⋮ | ⋮ | ⋮ | ⋮ |

图 1－1 学生情况表

(1) 集合结构(set)：数据元素的有限集合。数据元素之间除了“属于同一个集合”的关系之外没有其他关系。元素顺序是随意的。

(2)线性结构(linear)或称序列(sequence)结构：数据元素的有序集合。数据元素之间形成一对一的关系。

(3）树形结构(tree)：树是层次数据结构，树中数据元素之间存在一对多的关系。

(4)图结构(graph)：图中数据元素之间的关系是多对多的。

图 1－2给出了上述四种基本结构的示意图。

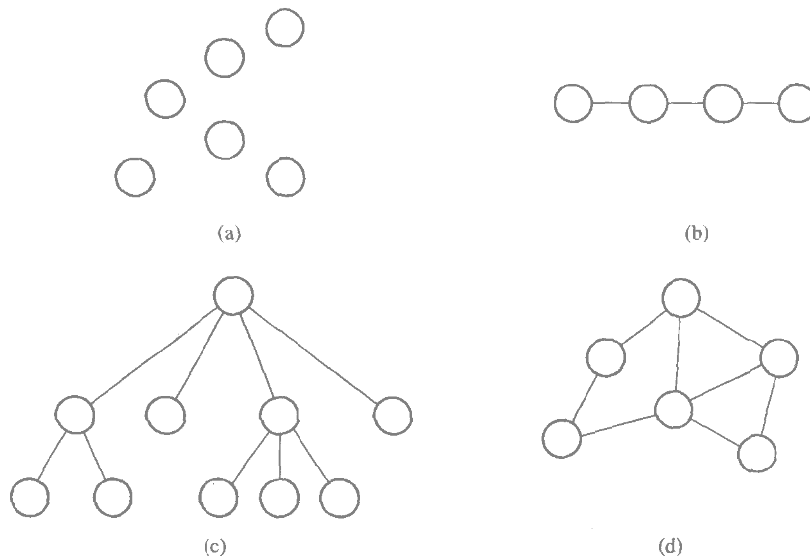


图 1－2 四种基本结构示意图

(a）集合结构； (b) 线性结构； (c) 树形结构； (d) 图结构

由于集合结构的元素间没有固有的关系，因此往往需要借助其他结构才能在计算机中实际表示该结构。

上述四种基本的结构关系可分为两类：线性结构(linear structure)和非线性结构(non-linear structure)。我们把除了线性结构以外的几种结构关系—— 树、图和集合归入非线性结构一类。

数据的逻辑结构是面向应用问题的，是从用户角度看到的数据的结构。数据必须在计

算机内存储，数据的存储结构(storage structure)是数据在计算机内的组织方式，是逻辑数据的存储映像，它是面向计算机的。

我们知道，计算机存储器（主存）是由有限个存储单元组成的一个连续的存储空间，这些存储单元或者是字节编址，或者是字编址的。从存储器角度看，存储器中是一堆二进制数据，它们可以被机器指令解释成为指令、整数、字符、布尔数等等，也可以被数据结构的算法解释成为具有某种结构的数据。

顺序(sequential)存储结构和链接(linked)存储结构是两种最基本的存储表示方法。

所谓顺序或连续(contiguous)的表示方法，也称计算的方法，需要一块连续的存储空间，并把逻辑上相邻的数据元素依次存储在连续的存储单元中。例如有四个数据元素组成的线性数据结构$(a_0, a_1, a_2, a_3)$，存储在某个连续的存储区内，设存储区的起始地址是 100，假 定每个元素占 2 个存储单元，则其顺序存储表示如图 1－3(a)所示。

在顺序存储结构表示时，可以容易地用一个数学公式来确定每个元素的位置。对于图 1－3(a) 的顺序存储结构，一个简单的地址计算公式是：

$$Loc(a_k)=102+2*k \quad (1-1)$$

公式(1－1)指明了元素 $a_k$ 的存储位置。

这种方法主要用于存储线性的数据结构。对于非线性的数据结构，如树结构，有时也可采用顺序存储的方法表示之。这将在以后讨论。

另一种基本的存储表示方法是链接存储表示。

在链接存储表示下，为了在计算机内存储一个元素，除了需要存放该元素本身的信息外，还需要存放与该元素相关的其他元素的位置信息。这两部分信息组成存放一个数据元素的结点(node)。图 1－3(b)给出了线性结构$(a_0, a_1, a_2, a_3)$的链接存储表示。其中每个结点的存储块分成两部分，一部分存放元素自身，另一部分包含该元素逻辑上的后继元素(successor)的结点的存储位置。这种关于其他结点的位置信息被称为链(link)。

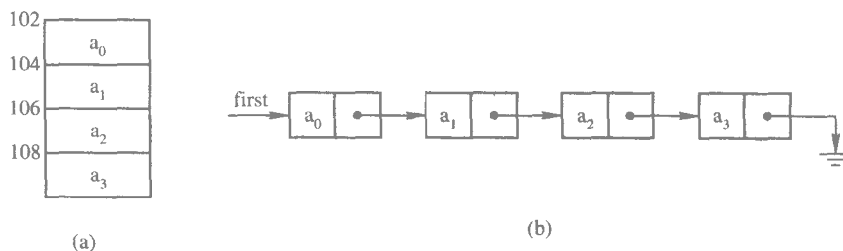


图 1－3　两种基本的存储表示方法

(a) 顺序存储结构；(b) 链接存储结构

图(b)中，电接地号表示空链(即不表示任何具体结点的单元地址)。注意：一个结点的存储位置通常是指存放该结点的存储块的起始单元的地址。

在以后的讨论中，在不会引起混淆的场合下，我们可以混合使用结点和元素这两个术语。但在必要时，我们将包括位置信息在内的存储块整体称为结点，而将其中的元素信息部分称为该结点的元素。

还可以有其他的存储数据的方法，如索引(index)的方法和散列(hash)的方法。这些基本

此为试读，需要完整PDF请访问：www.ertongbook.com

的存储表示方法，以及它们的组合，可用于实现数据的多种存储结构。

研究数据结构是为了解决应用问题，所以讨论数据结构必须同时讨论在数据结构上执行的相关运算及其算法才有意义。通过对运算及其算法的性能分析和讨论，使得我们在求解应用问题时，能选择和设计适当的数据结构，编写出高效的程序。

数据和操纵数据的运算是研究数据结构不可分割的两个方面，所以我们在讨论数据结构时，不但要讨论数据的逻辑结构，讨论数据的存储结构，还要讨论在数据结构上执行的运算(operation)，以及实现这些运算的算法(algorithm)。

## 1.2 数据抽象和抽象数据类型

抽象(abstraction)可以被理解为一种机制，其实质是抽取共同的和实质的东西，忽略非本质的细节。抽象可以使我们的求解问题过程以自顶向下的方式分步进行：首先考虑问题的最主要方面，然后再逐步细化，进一步考虑问题的某些细节，并最终实现之。

在程序设计中，抽象机制被用于两个方面：数据和过程。数据抽象(data abstraction)使程序设计者可以将数据元素间的逻辑关系与数据在计算机内的具体表示分别考虑。过程抽象(procedual abstraction) 可使程序设计者将在数据上定义的运算与实现这些运算的具体方法分开考虑。抽象的好处在于降低了问题求解的难度。

封装(encapsulation) 通常是指把数据和操纵数据的运算组合在一起的机制。使用者只能通过一组允许的运算访问其中的数据。从某种意义上说，数据的使用者只需知道这些运算的规范(定义）便可访问数据，而无需了解数据是如何组织和存储的，以及这些运算的具体算法如何等与实现有关的细节。也就是说对使用者隐藏了实现的细节。这种程序设计的策略称为信息隐蔽(information hiding)。

我们通常将数据和操纵数据的运算组成模块(module)，每个模块有一个明确定义的接口(interface)，模块内部信息只能经过这一接口被外部访问。一个模块的接口是实现运算的一组函数(functions)。这样的模块被称为黑盒子(blackbox)。一个程序使用一个采用信息隐蔽原则设计的模块被称为该模块的客户 (client)。

封装和信息隐蔽有助于降低问题求解的复杂性，提高程序的可靠性。

读者已熟悉 C 语言的基本数据类型，它们包括字符型、整型、实型、指针类型等原子类型。原子数据类型的值是不可分割的。现实世界最终可以用这些初等数据来表示。另一类是结构类型。结构类型的数据的值是由若干成分按某种结构组成的，因此是可以分解的。

数组(array)和结构(structure)是 C 语言提供的两种组织数据的机制。

例如 int list[5]定义了 5 个整数的数组，其下标范围为 0 到 4。结构需要显式定义。例如：

```
struct student{
    int student_id;
    char name[20];
    char sex;
    int age;
};
```

数据类型是数据抽象的一种方式。一个数据类型(data type)定义了一个值的集合以及作用于该值集的运算的集合。程序设计语言中，数据类型不仅规定了该类型的变量（或常量）的取值范围，还定义了该类型允许的运算。例如一个类型为 int 的变量的取值范围是 －32 768～32 767，在整型数上的运算有+、－、*、/、%，关系运算<、>、<=、>=、==、!=，赋值运算=等。图 1－4 的上半部分给出了 int 的规范，它规定了整型变量(或常量)的取值范围及允许的运算；图的下半部分表示与该类型的实现有关的方面。两者是分离的。

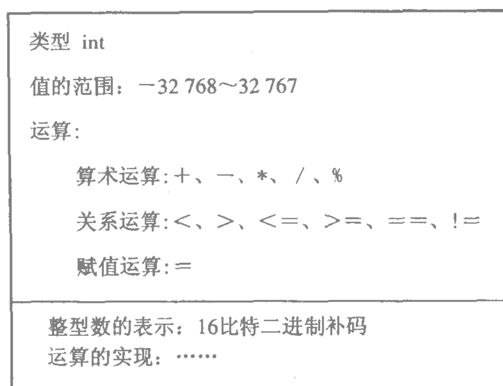


图 1－4 C 语言的数据类型 int

了解一个数据类型的数据对象(变量、常量)在计算机内的表示是有用的，但也是危险的。这使得应用程序可直接编写算法操纵该数据对象，但一旦改变该对象的存储表示，则必须改变所有使用该对象的程序。目前，普遍认为对使用者隐藏一个数据类型的对象的表示是个好的设计策略，即用户只能通过使用该类型提供的函数操纵该对象。这样，当数据对象的表示或实现函数的算法改变时，只要不改变函数的调用方式，应用程序将无需改变。

一个抽象数据类型 ADT(abstract data type)是一个数据类型，其主要特征包括该类型的数据对象及其运算的规范，它与数据对象的表示和运算的实现分离，实行封装和信息隐蔽。在一个抽象数据类型上应当定义哪些运算，这取决于应用。若一组运算是完备的，那么我们可以使用该组运算，对该数据结构实行我们希望的所有操作。

其实，C 语言的类型 int 就是一个抽象数据类型，我们只能通过类型 int 所规定的运算操纵整型变量或常量。虽然仅在面向对象程序设计语言出现后，才提供了必要的机制，使用户可以实现抽象数据类型，如 C++语言的类(class)，但这并不意味着当我们使用 C 语言描述数据结构时，不能使用抽象数据类型的原则。

对于一个数据结构，一方面要说明它的数据的逻辑结构，同时还应定义一组在该数据结构上执行的运算。逻辑结构和运算的定义组成了数据结构的规范(specification)，而数据的存储结构和运算算法的描述构成了数据结构的实现(implementation)。规范是实现的准则和依据，规范指明“做什么”，而实现解决“怎样做”。从规范和实现两方面来讨论数据结构的方式是抽象数据类型的观点。在本书中，一种数据结构被作为一个抽象数据类型来加以讨论。

## 1.3 数据结构的描述

从 1.1 节中我们知道，在概念层次上，可以根据元素间的关系将数据结构划分为四种不同的结构：集合结构、线性结构、树形结构和图结构。其中，每一种结构又可进一步划分，得到多种数据结构。有时一种数据结构需借助其他数据结构来表示，比如集合结构可以使用线性表、搜索树或散列表来表示。

如上所述，一个数据结构被看成是一个抽象数据类型。一个数据结构的 ADT 描述的是 ADT 的接口，它由 ADT 名称，对数据的逻辑结构关系的简单陈述，以及该 ADT 上定义的一组运算的规范这几部分组成。

一个运算的规范是指在概念的层次上对一个运算的精确定义，或者说是从客户使用的角度描述的运算。它既能精确描述运算又不涉及运算的实现细节。本书中我们从语法和语义两方面描述一个运算。一方面我们使用 C 语言的函数原型(function prototype)规定该运算的使用格式，包括运算名称、运算的输入参数、输出参数和返回值，另一方面使用前置条件(precondition)和后置条件(postcondition)来定义一个运算。前置条件规定了使用一个运算应当满足的先决条件。后置条件规定了运算执行后应有的结果，即运算的作用。如果某次调用满足前置条件，则该运算的任何实现应保证运算执行后得到后置条件所规定的结果。运算的调用者有责任保证前置条件成立。这里需说明一点，运算是在数据结构较抽象的层次上的概念，当我们实际用 C 语言函数实现运算时，有时还需要辅助函数，也就是说实现一个运算可能需要不止一个 C 语言函数。

下面我们以复数数据结构 Complex 为例，介绍本书中我们所采用的描述数据结构的方法和格式。复数结构 Complex 被定义为一个抽象数据类型 ADT Complex 。在 ADT 1－1 Complex 的描述中，我们使用了格式化的自然语言来描述。复数结构由一对实数(x，y)构成，x 为实部，y 为虚部。在复数上定义了构造函数(Comp)、加(Add)、减(Sub)、乘(Mul)和除(Div)四个运算。我们常常可以根据需要定义多于一个的构造函数。

ADT 1－1 Complex{

数据:

　　由一对实数(x，y)构成，x 为实部，y 为虚部。

运算：设两个复数分别为 $a=(a_1, a_2)$和 $b=(b_1, b_2)$。

　　Complex Comp(float x，float y)

　　后置条件：构造函数，函数返回复数(x，y)。

　　Complex Add(Complex a，Complex b)

　　前置条件：和的实部和虚部分别不超过实型值的允许范围。

　　后置条件：返回复数$(a_1+b_1, a_2+b_2)$。

　　Complex Sub(Complex a，Complex b)

　　前置条件：差的实部和虚部分别不超过实型值的允许范围。

　　后置条件：返回复数$(a_1-b_1, a_2-b_2)$。

　　Complex Mul(Complex a，Complex b)

前置条件：积的实部和虚部分别不超过实型值的允许范围。

后置条件：返回复数 $(a_1b_1-a_2b_2，a_1b_1+a_2b_2)$。

Complex Div(Complex a，Complex b)

前置条件 除数 b 不为 0，且商的实部和虚部分别不超过实型值的允许范围。

后置条件：返回复数 $((a_1b_1+a_2b_2)/(b_1^2+b_2^2)，(a_2b_1-a_1b_2)/(b_1^2+b_2^2))$。

}

实现一个数据结构(或 ADT)必须首先确定数据的存储表示，然后在给定的存储方式下实现相应的运算。

在本书中,我们采用 C 语言描述数据结构。在运算的规范中我们已使用了 C 函数原型，现在我们将使用 C 语言的数据类型(结构、数组和指针)描述数据的存储表示方法,并使用 C 语言语句描述实现运算的算法。可以认为，这也是一个抽象数据类型的 C 语言实现。

这里，我们不妨假定 C 语言的数组和结构都是顺序存储的：数组元素具有相同的数据类型,按照下标的次序存储在连续的存储区中 结构可以由不同类型的成分(域)组成,按照域表的次序顺序存放。借助于 C 语言的数组、结构和指针，我们可以描述和实现本书中讨论的各种数据结构。

下面我们用 C 语言的结构实现 Complex 。我们采用 typedef 定义复数类型 Complex。注意区分 Complex 和 complex。此处，complex 是结构名，不是类型名，它只是一个结构标记。当确定了以结构类型作为复数的存储表示方式后，我们便可以实现复数 ADT 上定义的运算。下面给出了 Add 的函数实现。同样可以写出其他运算的实现代码。在实现了全部运算后，我们便可在主函数中使用相应的复数运算进行复数计算。

实现复数 ADT 的代码见程序 1－1。代码可分为两部分：复数类型定义和实现复数运算的 C 函 数 。

【程序 1－1】 Complex 的实现。

```
#include <stdio.h>
#include <stdlib.h>
typedef struct complex
{
    float x, y;
}Complex;
Complex Comp(float x，float y)
{
    Complex c;
    c.x=x; c.y=y;
    return c;
}
Complex Add (Complex a，Complex b)
{
    Complex c;
    c.x=a.x+b.x;
```

此为试读,需要完整PDF请访问: www.ertongbook.com

```
        c.y=a.y+b.y;
        return c;
    }
    ……
```

当我们实现了复数抽象数据类型后，可编写相应的程序测试该 ADT 。一个简单的测试程序见程序 1－2。程序 1－2 中使用了函数 void PrintComplex(Complex a) 。从封装和信息隐蔽的角度看，这是必须的，可以将其添加到 ADT 1－1 中。

【程序 1－2】 测试复数加法运算。

```
void PrintComplex(Complex a)
{
      printf("a.x=%f, a.y=%f \n", a.x, a.y);
}
void main( )
{
     Complex a, b, c;
     a=Comp(1.0f, 2.0f); b=Comp(3.0f, 4.0f);
     c=Add(a, b);
     PrintComplex(a); PrintComplex(b);
     PrintComplex(c);
}
```

需要提醒的是，在大多数情况下，书中讨论的一个抽象数据类型 ADT 往往代表一类具有相同结构和相同运算集的数据结构，它被称为一个类属抽象数据类型(generic ADT)，它的元素类型通常并不重要。关于类属 ADT 的概念，我们在以后还会作进一步介绍。

# 1.4 算法和算法分析

## 4.1 算法及其性能标准

什么是算法？笼统地说，算法是求解一类问题的任意一种特殊的方法。较严格的说法是：一个算法(algorithm)是对特定问题的求解步骤的一种描述，它是指令的有限序列。此外，算法具有下列五个特征：

(1) 输入(input)：算法有零个或多个输入。

(2) 输出(output)：算法至少产生一个输出。

(3) 确定性(definite)：算法的每一条指令都有确切的定义，没有二义性。

(4) 能行性(effective)：算法的每一条指令都足够基本，它们可以通过执行有限次已经实现的基本运算来实现。

(5) 有穷性(terminate)：算法总能在执行有限步之后终止。

凡是算法，都必须满足以上五条特征。对于书中讨论的数据结构上定义的运算，我们将讨论实现它们的相应算法。

描述一个算法有多种方法，它可以用自然语言、流程图或程序设计语言来描述。当一个算法直接使用计算机程序设计语言描述时，该算法便成为程序。算法必须可终止，但计算机程序并没有这一限制，比如操作系统是一个程序，但不是一个算法。本书中，我们使用 C 语言描述算法，当然只有当了解了算法的足够多的实现细节后，才能写成程序，同时，这也使得对算法的性能分析往往成为对相应程序的性能分析。

衡量一个算法的性能，主要有以下几个标准：

(1) 正确性 (correctness)：算法的执行结果应当满足预先规定的功能和性能要求。

(2) 简明性(simplicity) ：一个算法应当思路清晰、层次分明、简单明了、易读易懂。

(3) 健壮性(robustness)：当输入不合法数据时，应能做适当处理，不至于引起严重后果。

(4) 效 率 (effeciency)：有效使用存储空间，并有高的时间效率。

算法的正确性是指在合法的输入下，算法应实现预先规定的功能和计算精度要求。对于大型程序，我们无法奢望它是“ 完全正确 ”的，而且这一点也往往无法证实，但我们要求它是健壮的，其含义是当程序万一遇到意外时，能按某种预定的方式作出适当的处理。正确性和健壮性是相互补充的。正确的程序并不一定是健壮的，而健壮的程序并不一定绝对正确。一个可靠的程序应当能在正常情况下正确地工作，而在异常情况下，亦能作出适当的处理，这就是程序的可靠性。

算法的效率是指算法执行的时间和所需的存储空间。算法的简明性要求算法的逻辑清晰，简单明了，是结构化的，这样使算法容易理解、容易阅读。牺牲算法的可读性换取一定的效率在现代程序设计中是不明智的做法。

### 1.4.2 算法的时间复杂度

一个程序的时间复杂度(time complexity) 是程序运行从开始到结束所需的时间。

程序的一次运行是针对所求解的问题的某一特定实例(instance)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(characteristics)，它直接影响排序算法的执行时间和所需的存储空间。因此，判断算法性能要考虑的一个基本特征是问题实例的规模 (size)。规模一般是指输入量(有时也涉及输出量)。

程序的运行时间与实例的特征有关。使用相同的排序算法对 100 个元素进行排序与对 10 000 个元素进行排序所需的时间显然是不同的。当然，算法自身的好坏直接影响程序所需的运行时间。不同的排序算法对同一组元素(即相同的实例)进行排序，程序运行的时间一般是不相同的。

程序的运行时间不仅与问题实例的特征和算法自身的优劣有关，还与运行程序的计算机软、硬件环境相关。它依赖于编译程序所产生的目标代码的效率，以及运行程序的计算机的速度及其运行环境。一般来说，我们希望能对算法（程序）作事前分析(priori analysis)，排除程序运行环境的因素来讨论算法的时间效率。当然这不是程序运行时间的实际值，而是算法运行时间的一种事前估计。算法的事后测试(posterori testing) 是测试一个程序在所选择的输入数据下运行时实际需要的时间。

为了实施算法的事前分析，通常使用程序步的概念。

一个程序步 (program step) 是指在语法上或语义上有意义的程序段，该程序段的执行时间

与问题实例的特征无关。下面我们以求一个数组元素的累加之和的程序（见程序 1－3)为例，来说明如何计算一个程序的程序步数。其中， n 个元素存放在数组 list 中，count 是全局变量，用来计算程序步数。程序中，语句 count++; 与数组求和的算法无关，只是为了计算程序步而添加的。去掉所有此语句，便是数组求和程序。可以看到，这里被计算的每一程序步均与问题实例的规模 n(数组元素的个数）无关。该程序的程序步数为 2n＋3。

【程序 1－3】 求一个数组元素的累加之和的迭代程序。

```
float Sum(float list[]，int n)
{
    float tempsum=0.0;
    count ++;                                   /*针对赋值语句*/
    for (int i=0; i<n; i++ ){
        count ++;                               /*针对 for 循环语句*/
        tempsum += list[i];
        count ++;                               /*针对赋值语句*/
    }
    count ++;                                   /*针对 for 的最后一次执行*/
    count ++;                                   /*针对 return 语句*/
    return tempsum;
}
```

我们还可以将求数组元素之和的程序写成如程序 1－4 所示的递归程序。

【程序 1－4】 求一个数组元素的累加之和的递归程序。

```
float RSum(float list[]，int n)
{
    count ++;                                   /*针对 if 条件*/
    if (n){
        count ++;                               /* 针对 RSum 调用和 return 语句*/
        return RSum(list, n-1)+list[n-1];
    }
    count++;                                    /* 针对 return 语句*/
    return 0;
}
```

为了确定这一递归程序的程序步，首先考虑当 n=0 时的情况。很明显，当 n＝0 时，执行 if 条件语句和第二句 return 语句，所需的程序步数为 2。当 n>0 时，执行 if 条件语句和第一句 return 语句。

设程序 RSum 的程序步为 T(n)，则有：

$$T(n)=\begin{cases}2 & (n=0)\\ 2+T(n-1) & (n>0)\end{cases} \tag{1-2}$$

这是一个递推公式，可以通过下面的方式计算。

$$
\begin{aligned}
T(n) &= 2+T(n-1) \\
&= 2+2+T(n-2) \\
&= 2+2+2+T(n-3) \\
&= \cdots \\
&= 2*n+T(0) \\
&= 2*n+2
\end{aligned}
$$

同样，移去程序 1－4 中的所有的 count++ 语句，便是数组求和的递归程序。

虽然计算所得到的程序 1－4的程序步为 $2*n+2$，少于程序 1－3 的程序步 $2*n+3$，但这并不意味着前者比后者快。请注意两者使用的程序步是不同的。语句 return tempsum;与语句 return RSum(list, n－1)+list[n－1]; 的时间开销显然是不同的。

鉴于对算法的事前分析的需要，我们引入了程序步的概念。正如我们看到的，不同的程序步在计算机上的实际执行时间通常是不同的，程序步数并不能确切地反映程序运行的实际时间。而且，事实上要获得一个程序的一次执行所需的程序步的精确计算往往也是困难的。那么，引入程序步的意义何在？下面定义的渐近时间复杂度使我们有望使用程序步在数量级上估计一个程序的执行时间，从而实现算法的事前分析。

### 1.4.3 渐近时间复杂度

定义：设 $f(n)$和 $g(n)$ 是定义在正整数 $n$ 上的正函数，如果存在两个正常数 $c$ 和 $n_0$，使得当 $n \geq n_0$ 时，有 $f(n) \leq cg(n)$，则记作 $f(n)=O(g(n))$，被称为大 O 记号 (bit-Oh notation)。

大 O 记号用以表达一个算法运行时间的上界。当我们说一个算法具有 $O(g(n))$ 的运行时间时，是指该算法在计算机上的实际运行时间不会超过 $g(n)$ 的一个常数倍。

例如，设一个程序的实际执行时间 $T(n)= 3.6n^3 + 2.5 n^2 + 2.8$，下面的定理表明 $T(n)=O(n^3)$。这就是说，如果我们只能够知道 $T(n)=O(n^3)$，并不能得到 $T(n)=3.6n^3 + 2.5 n^2 + 2.8$ 的计算公式，从算法事前分析的角度，我们认为已经有了满意的对算法时间复杂度的估计结果。使用大 O 记号表示的算法的时间复杂度，称为算法的渐近时间复杂度(asymptotic complexity)。渐近时间复杂度也常简称为时间复杂度。通常用 $O(1)$表示常数计算时间，即算法只需执行有限个程序步。

定理：如果 $f(n)=a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1 n+ a_0$ 是 $m$ 次多项式，则 $f(n)=O(n^m)$。

证明：取 $n_0=1$，当 $n \geq n_0$ 时，有

$$
\begin{aligned}
f(n) &= a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1 n+a_0 \\
&\leq |a_m|n^m+|a_{m-1}|n^{m-1}+\cdots+|a_1|n+|a_0| \\
&\leq (|a_m|+|a_{m-1}|/n+\cdots+|a_1|/n^{m-1}+ |a_0|/n^m)n^m \\
&\leq (|a_m|+|a_{m-1}|+\cdots+|a_1|+|a_0|)n^m
\end{aligned}
$$

可取 $c=|a_m|+|a_{m-1}|+\cdots+|a_1|+|a_0|$，定理得证。

常见的渐近时间复杂度有：$O(1)<O(\log_2 n)<O(n)<O(n\text{lb}\, n)$[①]$<O(n^2)<O(n^3)$。

我们可用大 O 记号表示的程序步来估计算法的时间复杂度，即得到算法的渐近时间复杂度。程序 1－3 和程序 1－4 的渐近时间复杂度均为 $O(n)$，即 $O(2n+3)=O(n)$ 和

① lbn 即为 $\log_2 n$。

此为试读，需要完整PDF请访问：www.ertongbook.com

$O(2n+2)=O(n)$。

很多情况下，我们可以通过考察一个程序中的关键操作（关键操作被认为是一个程序步）的执行次数来计算算法的渐近时间复杂度，有时也需要同时考虑几个关键操作，以反映算法的执行时间。例如程序 1－3 中，语句 tempsum+=list[i] 可认为是关键操作，它的执行次数为 $n$，由此得到的渐近时间复杂度也是 $O(n)$。

程序 1－5 是实现两个 $n\times n$ 矩阵相乘的程序段，每行的最右边表明该行语句执行的次数，我们将它们视为程序步。整个程序中所有语句的执行次数为

$$T(n)=2n^3+3n^2+2n+1 \tag{1－3}$$

语句 c[i][j]+=a[i][k]*b[k][j]可看成关键操作，它的执行次数是 $n^3$。所以，算法的渐近时间复杂度为 $O(n^3)$。

【程序 1－5】 矩阵乘法。

```
for(i=0; i<n; i++)                              /*  n+1    */
    for(j=0; j<n; j++){                         /* n(n+1) */
        c[i][j]=0;                              /* n^2    */
        for(k=0; k<n; k++)                      /* n^2(n+1) */
            c[i][j]+=a[i][k]*b[k][j];           /* n^3    */
    }
```

### 1.4.4 最坏、最好和平均情况时间复杂度

对于某些算法，即使问题实例的规模相同，如果输入数据不同，算法所需的时间开销也会不同。例如，在一个有 $n$ 个元素的数组中找一个给定的元素，从第一个元素开始依次检查数组元素。如果我们要找的元素是第一个元素，所需的查找时间最短，这就是所谓的算法的最好情况(best case)。如果待查的元素是最后一个元素，则是算法的最坏情况(worst case)。如果我们多次在数组中查找元素，并且假定以某种概率查找每个数组元素，最典型的是以相等的概率查找每个数组元素，这种情况下，就会发现程序需平均检索 $n/2$ 个元素，我们称之为算法时间代价的平均情况(average case)。

对应于三种不同的情况，我们有关于算法的三种时间复杂度：最好情况、最坏情况和平均情况时间复杂度。在三种情况下，我们都可以得到它们的渐近时间复杂度表示。

程序 1－5 的例子对于三种情况的时间复杂度都是一样的，三种情况下有不相同的时间复杂度的例子我们将在后面的章节中给出。

### 1.4.5 算法的空间复杂度

一个程序的空间复杂度(space complexity)是程序运行从开始到结束所需的存储量。

程序运行所需的存储空间包括两部分：

(1) 固定部分(fixed space requirement)：这部分空间与所处理数据的大小和个数无关，或者说与问题的实例的特征无关。它主要包括程序代码、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(variable space requirement)：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，将有 100 个元素的两个数组相加，与将有 10 个元素的

两个数组相加，所需的存储空间显然是不同的。这部分存储空间包括数据元素所占的空间，以及算法执行所需的额外空间，如递归栈所用的空间。

对算法的空间复杂度的讨论类似于对时间复杂度的讨论，并且一般说来，空间复杂度的计算比起时间复杂度的计算来得容易。此外，应当注意的是空间复杂度一般按最坏情况来分析。

## 小 结

本章的内容对全书来说是重要的。讨论数据结构应讨论数据的逻辑结构、存储结构以及在数据结构上定义的一组运算和实现这些运算的算法。数据结构上的运算是在逻辑结构上定义的，而只有当存储结构确定后才能给出其实现的算法。本章介绍的使用抽象数据类型描述数据结构的方式将贯穿全书。一个数据结构的 ADT 描述是用户使用数据结构的规范，它应严格定义，它的实现与使用分离，实行封装和信息隐蔽。数据结构的实现必须严格符合其 ADT 规范，这是程序设计的基本原则之一。本章最后介绍的算法效率和算法分析的基本方法将在以后各章中用以分析算法的时间和空间效率。

## 习 题 1

1-1 简述下列术语的含义：数据、数据元素、逻辑结构、存储结构、线性数据结构和非线性数据结构。

1-2 什么是数据结构？有关数据结构的讨论应包括哪些方面？

1-3 从概念上讲，有哪些基本的逻辑结构关系？

1-4 有哪两种常见的存储表示方式？

1-5 什么是抽象、数据抽象和过程抽象？

1-6 什么是封装和信息隐蔽？

1-7 什么是抽象数据类型和类属抽象数据类型？

1-8 为什么说 C 语言的类型 int 是抽象数据类型？

1-9 一个数据结构的 ADT 描述是 ADT 的接口，它包括哪几部分？

1-10 如何书写一个运算的规范？

1-11 为字符串定义一个 ADT，要求包含常见的字符串运算，每个运算定义成一个函数。请给出其 ADT 描述。

1-12 实现 ADT 1-1 Complex 除加法以外的其他运算。

1-13 什么是算法？说明算法和程序的区别。

1-14 简述衡量一个算法的主要性能标准。

1-15 什么是算法的时间复杂度和空间复杂度？

1-16 什么是程序步？引入程序步概念对算法的时间分析有何意义？

1-17 什么是算法的事前分析和事后测试？

1-18 什么是渐近时间复杂度？

1-19 确定下列各程序段的程序步，确定划线语句的执行次数，计算它们的渐近时间

**复杂度。**

```
(1) i=1; k=0;
    do {
        k=k+10*i; i++;
        } while(i<=n-1)
(2) i=1; x=0;
    do{
        x++; i=2*i;
      } while i<n;
(3) for(int i=1; i<=n; i++)
        for(int j=1; j<=i; j++)
            for (int k=1; k<=j; k++)    x++
(4) x=n; y=0;
    while(x>=(y+1)*(y+1)) y++;
```

# 第2章

# 两种基本数据结构

本书中我们将讨论多种数据结构，其中包括：线性表、栈、队列、字符串、广义表、树、优先权队列、集合、图等，它们通常被作为抽象数据类型加以讨论。实现这些数据结构最常用的存储表示方式是顺序存储和链接存储。当使用 C 语言实现它们时，我们往往使用数组实现顺序存储，而使用链表实现链接存储。这里，数组和链表作为抽象数据类型的实现形式，它们几乎是实现所有 ADT 的基础。

本章中，我们将重点讨论这两种基本数据结构。由于数组自身也可以看成一个抽象数据类型，所以在第 4章中，我们将讨论数组 ADT 和稀疏矩阵 ADT。C 语言的结构和联合在我们实现抽象数据类型时起着重要作用，我们在本章开始处先对其作简要回顾。

## 2.1 结 构 与 联 合

### 2.1.1 结构

结构(structure)是 C 语言提供的聚合数据的机制。使用结构可以将不同类型的数据组合成一个整体，便于使用。

一个结构（在许多其他程序设计语言中称为记录 record)是数据项的聚集(collection)。每个数据项有名称和类型，它们可以是不同的数据类型。

例如：

```
struct student
{
    char name[20];
    char sex;
    int age;
}
```

该语句定义了一个结构类型 struct student，可以使用它作为定义结构变量的类型，如 struct student studA;，这里，变量 studA 是一个结构。可以使用成员运算符“.”对结构变量的成员赋值。如，

```
strcpy(studA.name, "Wang");                    /*字符串赋值函数*/
studA.age=19;
```

```
studA.sex='M';
```

对成员变量可以像对普通变量一样进行其类型所允许的各种运算。

ANSI C 允许将一个结构变量整体赋值给另一个具有相同结构的结构变量，但不能将一个结构变量作为一个整体进行输入和输出，也不能直接判定两个结构是否相同。

为了能像使用 C 语言类型 int 一样使用一个结构类型，我们可以用 typedef 创建自己的结构类型 Student 如下：

```
typedef struct student
 {
     char name[20];
     char sex;
     int age;
 } Student;
```

这里，student 是结构名，Student 是结构类型名，我们可以像使用类型 int 一样用 Student 定义结构变量。定义变量的语句为 Student studA; ，无须加保留字 struct。事实上，结构名称与结构类型的名称可以相同。

### 2.1.2 联合

联 合(union) 是一个变量，它可以存放不同类型的数据对象。例如在编译程序的符号表管理中，假定常量可以是 int、float 或 char 类型。一种最简单的管理方法是不考虑它们的类型，分配相同大小的空间存放各种常量。联合的目的是使用单一变量，存放多个类型的值。显然任何时候只能存放其中之一。

定义一个联合的方法类似于结构，见下面的例子。

```
union u_tag {
     int ival;
     float fval;
     char chval;
};
```

这里， union u_tag 是一个联合类型，可以用来定义联合变量，如 union u_tag a，b;。分配给联合变量使用的存储块的大小是它的最大变量所需的存储块的大小。

联合可以放在结构或数组中，反之亦然。访问一个结构中的联合的成员等同于访问嵌套的结构变量。例如，

```
struct {
     char name[20];
     int flags;
     int utype;
     union {
          int ival;
          float fval;
          char chval;
```

```
    } u;
  } symtab [maxsize];
```

引用联合的成员 ival 的方式是：symtab[i].u.ival。

我们也可以像创建结构类型一样用 typedef 创建自己的联合类型，如下面的定义：

```
typedef union u_tag {
    int ival;
    float fval;
    char chval;
} Constval;
```

## 2.2 数　　组

数组(array)是大家熟悉的一种数据类型，几乎所有的程序设计语言都包含数组数据类型。在对数据结构的讨论中，我们通常使用数组描述数据结构的顺序表示，换句话说，使用数组实现数据的顺序存储结构。数组与结构相同，一旦定义，便不能再增加和删除元素，只能访问和修改数组元素的值，因此被称为静态数据结构(static data structures)。一个静态数据结构一旦建立，它包含的元素个数将不变。数据元素个数可变的数据结构称为动态数据结构(dynamic data structures)，可以在动态数据结构中插入或删除数据元素。数组与结构不同的是,数组的元素具有相同的数据类型,而结构的元素(域)可以是不同类型。所以,数组元素用下标(index)标识，而结构的域由域名(field name)引用。

### 2.2.1 一维数组

一维数组(one-dimensional array)常用于顺序存储的线性数据结构中。让我们先来看 C 语言中的一维数组。例如，int one[5]; 定义了 5 个整数组成的一个数组，下标从 0 到 4。数组元素可以在定义时赋值，也可以通过引用数组元素对元素赋值。例如，

```
int one[5]={0, 1, 2, 3, 4};
```

或

```
for ( i=0; i<5; i++) one[i]=i;
```

数组通常采用顺序表示，即数组中的元素按一定顺序存放在一个连续的存储区域。一个一维数组可以直接映射到一维的存储空间。由于数组元素具有相同类型，每个元素占有相同数量的存储单元，因此根据数组元素的下标可以方便地计算元素的存放地址。

设给长度为 $n$ 的一维数组 $a$ 所分配的存储块的起始地址是 $Loc(a[0])$，若已知 $a$ 的每个元素占 $k$ 个存储单元，则下标为 $i$ 的数组元素 $a[i]$ 的存放地址 $Loc(a[i])$ 是

$$Loc(a[i])= Loc(a[0])+i*k \qquad 0\leqslant i< n \tag{2-1}$$

### 2.2.2 二维数组

二维数组(two-dimensional array)的下标是二维的。可以认为二维数组是每个元素都是一维数组的一维数组。将一个二维数组映射到一维的存储空间一般有两种顺序：行优先顺序(row major order) 和列优先顺序(column major order)。大多数语言如 Pascal、Basic、C 和 C++

此为试读，需要完整PDF请访问：www.ertongbook.com