第一章 绪 论

自 1946 年美国第一台电子计算机问世以来，计算机科学和软硬件技术得到了飞速的发展，与此同时，计算机的应用领域也从最初的科学计算逐步发展到人类活动的各个领域。现在，计算机处理的对象不仅是简单的数值或字符，而且是带有不同结构的各种数据。因此，要设计出一个较好的软件，除了要掌握所用的计算机语言外，还要研究各种数据的特性和数据之间存在的关系。这就是“数据结构”这门学科形成和发展的背景。

1-1 基本术语

这一节，我们将对全书中最常用的名词和术语赋以确定的含义。

数据(data)是人们利用文字符号、数字符号以及其它规定的符号对现实世界的事物及其活动所做的描述。因此，大到一本书、一篇文章、一张图表等是数据，小到一条句子、一个单词、一个算式、以至一个数值、一个字符等都是数据。在计算机领域，人们把能够被计算机加工的对象，或者说能够被计算机输入，存储、处理和输出的一切信息都叫做数据。

数据元素(data element)是一个数据整体中相对独立的单位。如对于一个文件来说，每个记录就是它的数据元素；对于一个字符串来说，每个字符就是它的数据元素；对于一个数组来说，每一个成分就是它的数据元素。数据和数据元素是相对而言的，如对于一个记录来说，它相对于所在的文件被认为是数据元素，而它相对于所含的数据项又被认为是数据。因此，在本教材中，对数据和数据元素这两个术语的使用并不加以严格区别。

数据记录(data record)简称**记录**，它是数据处理领域组织数据的基本单位。它又由更小的单位——**数据项**(item)所组成，一个记录一般包括一个或若干个固定的数据项(当然每一个数据项还可以是记录的形式)。就拿对图书目录管理来说，每个记录表示一本图书的目录信息，如表 1-1 所示。

表 1-1 图书目录表

登录号	书 号	书 名	作者	出版社	定价
00001	ISBN 7-04-003907-9/TP・103	计算方法	唐 珍	高等教育	4.80
00002	ISBN 7-111-03247-0/TP・158	计算机辅助制造	李德庆	机械工业	3.30
00003	ISBN 7-118-00994-6/TP・126	FORTRAN90 学习指南	王文才	国防工业	11.00
00004	ISBN 7-302-00984-8/TP・363	数据结构	严蔚敏	清华大学	6.05
00005	ISBN 7-5609-0657-5/TP・65	微型计算机及其应用	周细等	华中理工大学	5.15
00006	ISBN 7-302-00860-4/TP・312	C 程序设计	谭浩强	清华大学	7.30
00007	ISBN 7-03-001460-X/TP・101	计算机图学	孟粹娟	科 学	11.30
⋮	⋮	⋮	⋮	⋮	⋮

在表 1-1 中，第一行为表目行或目录行，它给出了该表中每条记录的结构。从表目行向下的每一行为一条记录，每一列为一个数据项，每条记录都由 6 个数据项组成，其名称分别为登录号、书号、书名、作者、出版社和定价。当记录不同时，所对应的同一数据项的值可能相同，也可能不同。例如对于第 4 条和第 6 条记录来说，出版社数据项的值就相同，同为“清华大学”；对于第 1 条和第 3 条记录来说，书名数据项的值就不同，一个为《计算方法》，另一个为《FORTRAN 90 学习指南》。

在一个表或文件中，若所有记录的某个数据项的值都不同，也就是说，每个值能够唯一地标识一个记录时，则可把这个数据项作为记录的**关键数据项**，简称**关键项**(key item)，关键项中的每一个值称作为所在记录的**关键字**(key word 或 key)。在表 1-1 中，登录号数据项的值都不同，所以可把登录号作为记录的关键项，其中的每一个值都是所在记录的关键字，如 00002 为第 2 条记录的关键字，00005 为第 5 条记录的关键字等。

在一个表或文件中，能作为关键项的数据项可能没有，可能只有一个，也可能多于一个。当没有时，可把多个有关的数据项联合起来，构成一个组合关键项，用组合关键项中的每一个值来唯一地标识一个记录。

引入了记录的关键项和关键字后，为简便起见，在以后的讨论中，经常利用关键项来代替所有记录，利用关键字来代替所在的记录。

数据处理(data processing)是指对数据进行查找、插入、删除、合并、排序、统计、简单计算、输入、输出等的操作过程。在早期，计算机主要用于科学和工程计算，进入 80 年代以后，计算机主要用于数据处理。据有关统计资料表明，现在计算机用于数据处理的时间比例平均高达百分之八十以上，随着时间的推移和计算机应用的进一步普及，计算机用于数据处理的时间比例必将进一步增大。像计算机情报检索系统、经济信息管理系统、图书管理系统、物资调配系统、银行核算系统、财务管理系统等都是计算机在数据处理领域的具体应用。数据结构是数据处理的软件基础，因此，数据结构课程是计算机所有专业的主干课程之一。

数据结构(data structure)，简单地说是指数据以及数据之间的联系。上面提到，数据的描述对象是现实世界的事物及其活动，而任何事物及其活动都不是孤立存在的，都是在一定意义上相互联系、相互影响的，所以数据之间必然存在着联系。由于这种联系是内在的，或根据需要人为定义的，所以被看作为“逻辑”上的联系，因此，又把数据结构称作为数据的**逻辑结构**。数据结构在计算机存储器上的存储表示称作为数据的**物理结构**或**存储结构**。由于存储表示的方法有顺序、链接、索引、散列等多种，所以，一种数据结构可表示成一种或多种物理结构。

为了更确切地描述数据结构，通常采用二元组表示：

$$B=(K,R)$$

B 是一种数据结构，它由数据元素的集合 K 和 K 上二元关系的集合 R 所组成。其中

$$K=\{k_i \mid 1\leqslant i\leqslant n, n\geqslant 0\}$$

$$R=\{r_j \mid 1\leqslant j\leqslant m, m\geqslant 1\}$$

k_i 表示第 i 个数据元素，n 为 B 中数据元素的个数，特别地，若 $n=0$，则 K 是一个空集，因而 B 也就无结构而言，或者说它具有任何结构；r_j 表示第 j 个二元关系(以后均简称

关系),m 为 K 上关系的个数。

在本书所讨论的数据结构中,一般只讨论 m=1 的情况,即 R 中只包含一个关系(R={r})的情况。对于包含有多个关系的数据结构,可分别对每一个关系进行讨论。

K 上的一个关系 r 是序偶的集合。对于 r 中的任一序偶<x,y>(x,y∈K),我们把 x 叫做序偶的第一元素,把 y 叫做序偶的第二元素,又称序偶的第一元素为第二元素的**直接前驱**,简称**前驱**,称第二元素为第一元素的**直接后继**,简称**后继**。如在<x,y>的序偶中,x 为 y 的前驱,而 y 为 x 的后继。

数据结构还能够利用图形形象地表示出来,图形中的每个结点(或叫顶点)对应着一个数据元素,两结点之间带箭头的连线(称作有向边或弧)对应着关系中的一个序偶,其中序偶的第一元素为有向边的起始结点,第二元素为有向边的终止结点。

作为例子,下面根据表 1-2 构造一些典型的数据结构。

表 1-2 教务处人事简表

职 工 号	姓 名	性 别	出生年月	职 务	单 位
01	万明华	男	1952 年 8 月	处 长	
02	赵 宁	男	1958 年 6 月	科 长	教材科
03	张 利	女	1954 年 12 月	科 长	考务科
04	赵书芳	女	1962 年 8 月	主 任	办公室
05	刘永年	男	1949 年 8 月	科 员	教材科
06	王明理	女	1965 年 4 月	科 员	教材科
07	王 敏	女	1962 年 6 月	科 员	考务科
08	张 才	男	1957 年 3 月	科 员	考务科
09	马立仁	男	1965 年 10 月	科 员	考务科
10	邢怀常	男	1966 年 7 月	科 员	办公室

表中共有 10 条记录,每条记录都由六个数据项所组成,由于每条记录的职工号各不相同,所以可把每条记录的职工号作为该记录的关键字,并在下面的例子中,我们将用记录的关键字来代表整个记录。

例 1 一种数据结构 linearity=(K,R),其中

K={01,02,03,04,05,06,07,08,09,10}

R={r}

r={〈05,01〉,〈01,03〉,〈03,08〉,〈08,02〉,〈02,07〉,〈07,04〉,
〈04,06〉,〈06,09〉,〈09,10〉}

对应的图形如图 1-1 所示。

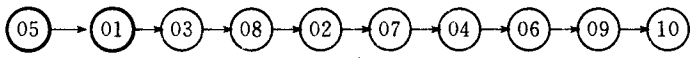

图 1-1 线性结构示意图

结合表 1-2,细心的读者不难看出:r 是按职工年龄从大到小排列的关系。

在 linearity 中,每个数据元素有且只有一个直接前驱元素(除结构中第一个元素 05

外),有且只有一个直接后继元素(除结构中最后一个元素 10 外)。这种数据结构的特点是数据元素之间的 1∶1 联系,即**线性关系**,我们把具有这种特点的数据结构叫做**线性结构**。

例 2 一种数据结构 tree=(K,R),其中

K={01,02,03,04,05,06,07,08,09,10}

R={r}

r={⟨01,02⟩,⟨01,03⟩,⟨01,04⟩,⟨02,05⟩,⟨02,06⟩,⟨03,07⟩,⟨03,08⟩,⟨03,09⟩,⟨04,10⟩}

对应的图形如图 1-2 所示。

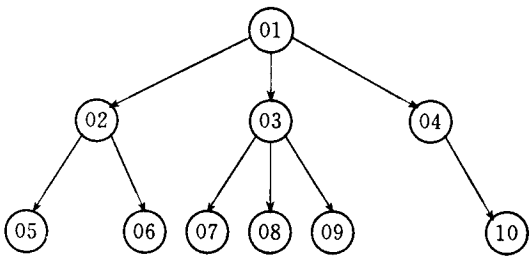

图 1-2 树结构示意图

结合表 1-2,细心的读者不难看出:r 是人员之间领导与被领导的关系。

图 1-2 象倒着画的一棵树,在这棵树中,最上面的一个没有前驱只有后继的结点叫作树根结点,最下面一层的只有前驱没有后继的结点叫作树叶结点,除树根和树叶之外的结点叫作树枝结点。在一棵树中,每个结点有且只有一个前驱结点(除树根结点外),但可以有任意多个后继结点(树叶结点可看作为具有 0 个后继结点)。这种数据结构的特点是数据元素之间的 1∶N 联系(N≥0),我们把具有这种特点的数据结构叫作**树型结构**或**树结构**。

例 3 一种数据结构 graph=(K,R),其中

K={01,02,03,04,05,06,07}

R={r}

r={⟨01,02⟩,⟨02,01⟩,⟨01,04⟩,⟨04,01⟩,⟨02,03⟩,⟨03,02⟩,⟨02,06⟩,⟨06,02⟩,⟨02,07⟩,⟨07,02⟩,⟨03,07⟩,⟨07,03⟩,⟨04,06⟩,⟨06,04⟩,⟨05,07⟩,⟨07,05⟩}

对应的图形如图 1-3 所示。

从图 1-3 可以看出,r 是 K 上的对称关系,为了简化起见,我们把⟨x,y⟩和⟨y,x⟩这两个对称序偶用一个无序对(x,y)或(y,x)来代替;在图形中,我们把 x 结点和 y 结点之间两条相反的有向边用一条无向边来代替。这样 r 关系可改写为:

r = {(01,02),(01,04),(02,03),(02,06),(02,07),(03,07),(04,06),(05,07)}

对应的图形如图 1-4 所示。

如果说 r 中每个序偶里的两个元素所代表的人员是好友的话,那么 r 关系就是人员之间的好友关系。

从图 1-3 或 1-4 可以看出,结点之间的联系是 M∶N 联系(M≥0,N≥0),也就是说,

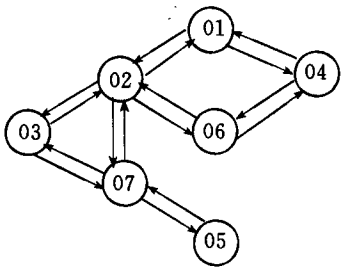

图 1-3　图型结构示意图

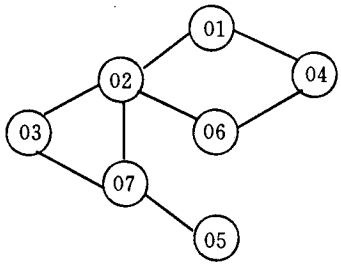

图 1-4　图型结构示意图

每个结点可以有任意多个前驱结点和任意多个后继结点。我们把具有这种特点的数据结构叫做**图型结构**。

从图型结构、树型结构和线性结构的定义可知，树型结构是图型结构的特殊情况（即 M＝1 的情况），线性结构是树型结构的特殊情况（即 N＝1 的情况）。为了区别于线性结构，我们把树型结构和图型结构统称为**非线性结构**。

例 4　一种数据结构 B＝(K，R)，其中

$K=\{k_1,k_2,k_3,k_4,k_5,k_6\}$

$R=\{r_1,r_2\}$

$r_1=\{\langle k_3,k_2\rangle,\langle k_3,k_5\rangle,\langle k_2,k_1\rangle,\langle k_5,k_4\rangle,\langle k_5,k_6\rangle\}$

$r_2=\{\langle k_1,k_2\rangle,\langle k_2,k_3\rangle,\langle k_3,k_4\rangle,\langle k_4,k_5\rangle,\langle k_5,k_6\rangle\}$

若用实线表示关系 r_1，虚线表示关系 r_2，则对应的图形如图 1-5 所示。

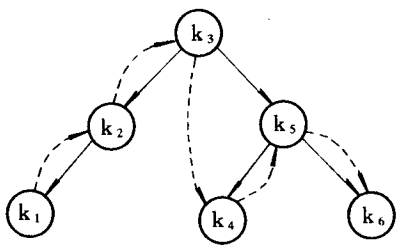

图 1-5 带有两个关系的数据结构示意图

从图 1-5 可以看出数据结构 B 是一种非线性的图型结构。但是，若只考虑关系 r_1 则为树型结构，若只考虑关系 r_2 则为线性结构。

算法(algorithm)，简单地说就是解决特定问题的方法（关于算法的严格定义，在此不作讨论）。特定的问题可分为数值的和非数值的两类。解决数值问题的算法叫做数值算法，科学和工程计算方面的算法都属于数值算法，如求解数值积分，求解线性方程组，求解代数方程，求解微分方程等。解决非数值问题的算法叫做非数值算法，数据处理方面的算法都属于非数值算法，如各种排序算法、查找算法、插入算法、删除算法、遍历算法等。数值算法和非数值算法并没有严格的区别，一般说来，在数值算法中主要进行算术运算，而在非数值算法中，则主要进行比较和逻辑运算。另一方面，特定的问题可能是递归的，也可能是非递归的，因而解决它们的算法就有递归算法和非递归算法之分。当然，从理论上讲，任何递归算法都可以通过循环、堆栈等技术转化为非递归算法。

在计算机领域，一个算法实质上是针对所处理问题的需要，在数据的逻辑结构和存储结构的基础上施加的一种运算。由于数据的逻辑结构和存储结构不是唯一的，在很大程度上可以由用户自行选择和设计，所以处理同一个问题的算法也不是唯一的。另外，即使对

于具有相同的逻辑结构和存储结构而言，其算法的设计思想和技巧不同，编写出的算法也大不相同。我们学习数据结构这门课的目的，就是要会根据数据处理问题的需要，为待处理的数据选择合适的逻辑结构和存储结构，进而设计出比较满意的算法。

1-2 算法描述

本书在进行算法描述时，采用“类 pascal 语言”作为工具。类 pascal 语言是我们在标准 pascal 语言的基础上所做的修改，它忽略了标准 pascal 语言中语法规则的一些细节，同时增加了一些功能较强的语句，这样使得描述出的算法清晰、直观、便于阅读和分析。读者根据这些算法不难编写出符合任一种 pascal 语言（如标准 pascal、IBM-PC pascal 或 Turbo pascal 等）或其它任一种语言语法规则的算法来。

类 pascal 语言所包含的语句如下：

1. 赋值语句

变量名：＝表达式；

2. 转向语句

goto 语句标号；

3. 调用过程语句

过程名(参数表)；

4. 退出循环语句

exit；

用于各种循环语句中，该语句被执行时，将立即退出当前循环，相当于 goto 到该循环语句后的第一条语句上，它是循环语句的一个非正常出口。

5. 返回语句

return；

用于过程或函数体中，该语句被执行时，将立即退出当前过程或函数，返回到调用前所保存的位置继续向下执行，它是执行过程或函数的一个非正常出口。

在函数体中使用该语句还可以书写成 return(表达式)的形式，表示把表达式的值赋给函数名后再退出当前被执行的函数，它相当于如下两条语句：

函数名：＝表达式;return；

6. 出错处理语句

error(字符串)；

在算法中为了避免非法操作，需要进行出错处理时统一使用该语句，它表示此算法的执行到此中止。在实际算法中，它可能是转去执行一个错误处理程序，也可能是简单地打印出错误信息并停止运行等。该语句括号内的字符串用以说明出错的原因，如使用字符串'overflow'表示溢出，'out of range'表示超出范围或越界等。

7. 复合语句

begin 语句 1;语句 2;…;语句 n end；

8. 条件语句

if 条件 then 语句 1[else 语句 2];

其中中括号部分可以省略。

9. 情况语句

格式 1:

```
case 变量名 of
  常量 1:语句 1;
  常量 2:语句 2;
    ⋮   ⋮
  常量 n:语句 n
end;
```

格式 2:

```
case
  条件 1:语句 1;
  条件 2:语句 2;
    ⋮   ⋮
  条件 n:语句 n
  else 语句 n+1
end;
```

10. for 循环语句

格式 1:

```
for 变量名 := 初值 to 终值 do 语句;
```

格式 2:

```
for 变量名 := 初值 downto 终值 do 语句;
```

若格式 1 中的初值大于终值,或格式 2 中的初值小于终值则都不会执行其循环体。

11. while 循环语句

```
while 条件 do 语句;
```

12. repeat 循环语句

```
repeat 一组语句 until 条件;
```

在这 12 种语句中,第 4、5、6 种语句和第 9 种的格式 2 语句是标准 pascal 语句所没有的,但在其它的 pascal 语言中具有类似的语句。

采用类 pascal 语言描述算法的书写规则如下:

1. 所有算法均以过程或函数说明的形式书写。算法的输入数据一般来自参数表,输出数据一般也由变参或函数名带回,这样就使得每个算法成为功能相对独立的一个模块。

2. 参数表中的参数一般不进行类型说明,若参数是变参,则规定为大写字母或大写字母开头的标识符表示,若参数是值参,则规定以小写字母或小写字母开头的标识符表示,当然作指定说明的除外。

3. 过程和函数体中的定义和说明部分一般被省略,语句部分通常按内容层次分别对语句进行编号,以便分析。第一层编号用(1)、(2)、(3)、……表示,第二层编号用(Ⅰ)、

(Ⅰ)、(Ⅱ)、……表示，第三层编号用(a)、(b)、(c)、……表示，并规定除第一层编号的外面不省略语句括号 begin 和 end 外，其余层次均省略。

例如，对于一维整型数组 A(1：n)，其中 1 和 n 分别表示该数组下标的下界和上界，若要求从下标 1 至 n(n≥1)的元素中查找出最大值和最小值，并把它们分别赋给变参 Max 和 Min，则算法描述为：

```
procedure find(A,n,Max,Min);
  begin
    (1) Max:=A[1]; Min:=A[1]; {赋初值}
    (2) for i:=2 to n do {查找过程}
      (Ⅰ) if A[i]>Max then Max:=A[i];
      (Ⅱ) if A[i]<Min then Min:=A[i]
  end;
```

此算法的描述完全符合上面所述的三条规则。

4. 在条件或循环语句中，出现多条简单语句时，为简单起见，也可不采用编号，而采用方括号代替 begin 和 end 语句括号。如：

```
if R[i]=T[j] then [i:=i+1; j:=j+1]
             else [i:=i-j+2; j:=1];
```

5. 当需要从若干个表达式中取其值最大者或最小者时，可简记为：

max(表达式1,表达式2,…);

min(表达式1,表达式2,…);

实际上这两个函数均可通过条件语句的嵌套或其它方法来实现。

6. 当两个变量 x 和 y 需要相互交换值时，可简记为：

x↔y;

实际上它对应下面三条赋值语句：

```
t:=x;x:=y;y:=t; {t 为临时工作变量}
```

此外，在利用类 pascal 语言描述算法时，还需要使用标准 pascal 语言中的所有数据类型、标准过程和函数等。

1-3 算法评价

对于解决同一个问题，往往能够编写出许多不同的算法。例如，对于排序问题，在第七章中将介绍多种算法。进行算法评价的目的，既在于从解决同一问题的不同算法中选择出较为合适的一种，也在于知道如何对现有算法进行改进，从而有可能设计出更好的算法。

一般从以下四个方面对算法进行评价

一、正确性

正确性是设计和评价一个算法的首要条件，如果一个算法不正确，其它方面就无从谈起。一个正确的算法是指在合理的数据输入下，能在有限的运行时间内得出正确的结果。

通过对数据输入的所有可能情况的分析和上机调试可以证明算法是否正确。当然，要从理论上证明一个算法的正确性，并不是一件容易的事，也不属于本课程所研究的范围，故不作讨论。

二、运行时间

运行时间是指一个算法在计算机上运算所花费的时间。它大致等于计算机执行一种简单操作（如赋值操作、转向操作、比较操作等）所需的时间与算法中进行简单操作次数的乘积。因为执行一种简单操作所需的时间随机器而异，它是由机器本身硬软件环境决定的，与算法无关，所以我们只讨论影响运行时间的另一个因素——算法中进行简单操作的次数。

不管一个算法是简单还是复杂，最终都是被分解成简单操作来具体执行的，因此，每个算法都对应着一定的简单操作的次数。显然，在一个算法中，进行简单操作的次数越少，其运行时间也就相对地越少；次数越多，其运行时间也就相对地越多。所以，通常把算法中包含简单操作次数的多少叫做算法的**时间复杂性**，它是一个算法运行时间的相对量度。

若解决一个问题的规模为 n，例如在排序问题中，n 表示待排元素的个数；在矩阵求逆中，n 表示矩阵的阶数；在图的遍历中，n 表示图中的顶点数。那么，算法的时间复杂性就是 n 的一个函数，通常记为 T(n)。下面通过例子来分析算法的时间复杂性。

算法1：累加求和

```
function sum(A,n):real;
                {A 表示一维实型数组,n 表示数组的大小}
  begin
    (1) s:=0; {给累加变量 s 赋初值}
    (2) for i:=1 to n do {进行累加求和}
          s:=s+A[i];
    (3) sun:=s {把 s 值(即累加和)赋给函数名}
  end;
```

计算机执行这个算法时，第(1)步和第(3)步都只需一次赋值操作，为了分析第(2)步包含多少简单操作的次数，可改写为：

```
(2)       i:=1;                          1次
      1:  if i>n then goto 2;            n+1次
          s:=s+A[i];                     n 次
          i:=i+1;                        n 次
          goto 1;                        n 次
(3)   2:  sum:=s
```

把第(2)步分解后的每一条语句的执行次数加起来，就得到了它包含的简单操作的次数，即为4n+2。因此，算法1的时间复杂性为：

$$T(n) = 4n + 4$$

算法2：矩阵相加

```
procedure matrixadd(A,B,C,n);
    {A,B,C 分别为 n 阶矩阵，A,B 表示两个加数，C 表示和}
  begin
    for i:=1 to n do
      for j:=1 to n do
        C[i,j]:=A[i,j]+B[i,j]
  end;
```

通过与算法1相似的分析过程可得到算法2的时间复杂性为：

$$T(n) = 4n^2 + 5n + 2$$

算法1和算法2的时间复杂性还比较容易计算，因为算法比较简单，同时 for 循环中的循环次数是固定的。但是，当算法较复杂，同时包含有 while 循环或 repeat 循环时，其时间复杂性的计算就相当困难了。实际上，一般也没有必要精确地计算出算法的时间复杂性，只要大致计算出相应的**数量级**(Order)即可。下面接着讨论时间复杂性 T(n)的数量级表示。

设 T(n)的一个辅助函数为 f(n)，定义为当 n 大于等于某一足够大的正整数 n_0时，存在着两个正的常数 a 和 b(a<b)，使得 $a<\frac{T(n)}{f(n)}<b$ 均成立，则称 f(n)是 T(n)的同数量级函数。把 T(n)表示成数量级的形式为

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

其中大写字母 O 为英文 order(即数量级)一词的第一个字母。这种表示的意思是指 f(n)同 T(n)只相差一个常数倍。

例如，在算法1中，当 n≥3时，$1<\frac{T(n)}{n}<6$ 均成立，则 f(n)=n；在算法2中，当 n≥2时，$1<\frac{T(n)}{n^2}<11$均成立，则 $f(n)=n^2$。由此可以推出，当 T(n)是 n 的多项式时，f(n)则为 T(n)的最高次幂。若把算法1和算法2的时间复杂性分别用数量级的形式表示，则为O(n)和 $O(n^2)$。

算法的时间复杂性采用数量级的形式表示后，将给求一个算法的 T(n)带来很大方便，这时只需要分析影响一个算法运行时间的主要部分即可，不必对每一步都进行详细的分析；同时，对主要部分的分析也可简化，一般只要分析清楚循环体内简单操作的执行次数即可。例如，对于算法1，只要根据第(2)步的循环次数 n，就可求出其时间复杂性为 O(n)；对于算法2，只要弄清楚双重循环内赋值操作的执行次数为 n^2，就可求出其时间复杂性为 $O(n^2)$。

表1-3给出了各种有代表性的 T(n)函数的算法，在不同 n 值时的运行时间。表中凡未注明的时间单位为微秒(10^{-6}秒)。因算法的实际运行时间随机器而异，所以此表上的时间主要用作相互比较。

从表中可以得出两点结论：

(1) 当 T(n)为对数函数、幂函数或它们的乘积时，算法的运行时间是可以接受的，我们称这些算法为有效的算法；当 T(n)为指数函数或阶乘函数时，算法的运行时间随着 n 而迅速增大是不可接受的，我们称这些算法是“坏”的算法或无效的算法。

(2) 随着 n 值的增大，各种 T(n)函数所对应的运行时间的增长速度大不相同，对数函数的增长速度最慢，线性函数较之快些，其余类推；因此，当 n 足够大后，各种不同数量级的 T(n)函数存在着下列关系：

$$O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<\cdots<O(2^n)<O(n!)$$

表1-3 算法的运行时间与T(n)的关系

T(n)	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	n^5	2^n	n!
n=20	4.3	20	86.4	400	8毫秒	3.2秒	1.05秒	771世纪
n=40	5.3	40	213	1600	64毫秒	1.7分	12.7天	2.59×10^{32}世纪
n=60	5.9	60	354	3600	216毫秒	13分	366世纪	2.64×10^{66}世纪

一个算法的时间复杂性除了与问题的规模 n 有关外，还与输入的具体数据以及多数据情况下的输入次序有关，当输入的具体数据与次序不同时，其算法的时间复杂性也可能不同。所以，当计算一个算法的时间复杂性时，要考虑到具体数据输入时的各种可能情况。例如，从一维数组 A(1:n)中查找其值等于给定值 x 的算法为：

```
procedure loacate(A,n,x,K);
        {K 为一变参，当查找成功时，返回对应的下标，查找失败时，返回0值}
  begin
    (1) i:=1;
    (2) while (i<=n) and (A[i]<>x) do
          i:=i+1;
    (3) if i>n then K:=0 else K:=i
  end;
```

此算法的时间复杂性主要取决于第(2)步的比较次数(即循环次数加1)，而比较次数不是固定的，它与具体的数据输入有关，当元素 A[1]等于给定值 x(即最好情况)时，只要进行一次比较，则 T(n)为 O(1)(即不随 n 的大小而改变)；当没有任何元素等于给定值 x(即最坏情况)时，表明查找失败，需进行(n+1)次比较，则 T(n)为 O(n)；当考虑到数组 A 中每个元素都有相同的概率$\left(即为\frac{1}{n+1}\right)$等于给定值 x(即平均情况)时，则所需比较的平均次数为：

$$\frac{1}{n+1}\sum_{i=1}^{n+1} i = \frac{n}{2}+1$$

所以，从平均情况看，T(n)为 O(n)。

以后在分析一个算法的时间复杂性时，一般考虑平均和最坏这两种情况。对于最坏情况，通常比较容易求出；对于平均情况，往往需要概率统计等方面的数学知识，进行严格的理论推导后才能求出，为了简化起见，我们一般不作理论上的推导，而是直接给出结论。不过，对于一个算法来说，在平均和最坏两种情况下的时间复杂性的数量级形式往往是相同的。

三、占用的存储空间

一个算法在计算机存储器上所占用的存储空间，包括存储算法本身所占用的存储空

间，算法的输入输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间这三个方面。算法的输入输出数据所占用的存储空间是由要解决的问题所决定的，它不随算法的不同而改变。存储算法本身所占用的存储空间与算法书写的长短成正比，要压缩这方面的存储空间，就必须编写出较短的算法。算法在运行过程中临时占用的存储空间随算法的不同而异，有的算法只需要占用少量的临时工作单元，而且不随问题规模的大小而改变，我们称这种算法是“就地”进行的，是节省存储的算法；有的算法需要占用的临时工作单元数随着问题规模 n 的增大而增大，当 n 较大时，将占用较多的存储单元，浪费存储空间，例如将在第七章介绍的归并排序算法就是如此。

分析一个算法所占用的存储空间要从各方面综合考虑。如对于递归算法来说，一般都比较简短，算法本身所占用的存储空间较少，但运行时需要一个附加堆栈，从而占用较多的临时工作单元；若写成非递归算法，一般可能比较长，算法本身占用的存储空间较多，但运行时将需要较少的存储单元。

算法在运行过程中临时占用的存储空间的大小被定义为算法的**空间复杂性**。算法的空间复杂性比较容易计算，它包括局部变量（即在本算法中说明的变量）所占用的存储空间和系统为实现递归（如果是递归算法的话）所使用的堆栈这两个部分。算法的空间复杂性一般也以数量级的形式给出。

四、简单性

最简单和最直接的算法往往不是最有效的，即最节省时间和空间的，但算法的简单性使得证明其正确性比较容易，同时便于编写、修改、阅读和调试，所以还是应当强调和不容忽视的。不过对于那些需要经常使用的算法来说，有效性比简单性更为重要。

上面讨论了如何从四个方面来评价一个算法的问题。这里还需要指出，除了算法的正确性之外，其余三个方面往往是相互矛盾的。如当追求较短的运行时间时，可能带来占用较多的存储空间和较繁的算法；当追求占用较少的存储空间时，可能带来较长的运行时间和较繁的算法；当追求算法的简单性时，可能带来占用较长的运行时间和较多的存储空间。所以在设计一个算法时，要从这三个方面综合考虑，还要考虑到算法的使用频率、算法的结构化和易读性以及所使用机器的硬软件环境等因素，才能设计出比较好的算法。

1-4 pascal 语言中的数据类型

在计算机领域，数据类型是与每一种计算机语言（甚至与每一种版本、每一种机器）都相关的概念。一般地说，计算机语言不同，对数据进行分类的规则也不同，因而产生出的数据类型也不完全相同。在标准 pascal 语言中，数据被划分为三大类型：简单（或叫标量）类型、结构（或叫构造）类型和指针类型。

一、简单类型

简单类型又包括整型（integer）、实型（real）、字符型（char）、布尔型（boolean）、枚举型和子域型，其中前四种类型是标准型，即由系统定义的，后两种类型由用户自行定义。每一

种简单类型都定义了一个具有相同数据特征的值的集合，如整型定义的值的集合是从－maxint 到 maxint 之间的所有整数，布尔型定义的值的集合是 false 和 true 这两个表示逻辑假和真的标识符。每个简单类型的数据都是一个无法再分割的数据“原子”，如整型数据25、实型数－3.62、字符‘a’、逻辑值 true 等都是这样的数据“原子”。存储一个简单类型的数据需要占用一个存储单元，该存储单元所包含的字节数（每个字节由八位二进制位组成）由数据的类型而决定。例如，在普通的微型机中，一个整型数据占用两个字节的存储单元，一个实型数据占用四个字节的存储单元，一个字符数据占用一个字节的存储单元（但每个汉字占用两个字节的存储单元）等。

二、结构类型

结构类型又包括数组、记录、集合和文件这四种类型。每一种结构类型中的数据都由若干个成分所组成，如数组（即数组类型中的数据）是由固定数目的数据元素所组成的。每一种结构类型中的数据都对应着一定的逻辑结构和存储结构，下面分别进行讨论。

1. 数组

数组类型的定义为：

$$\text{array}\ [T_1]\ \text{of}\ T_2;$$

其中 T_1 表示下标类型，它可以是除整型和实型外的其它任何简单类型，假定用 card(T_1) 表示 T_1 中所含不同值的个数，则该数组类型中的每个数组均含有 card(T_1) 个元素。T_2 表示成分（即元素）类型，它可以是任何类型。

数组中的元素在位置上是顺序排列的，即第 i 个元素排列在第 i－1 个元素的后面和第 i＋1 个元素的前面。这样，元素之间在位置上的排列关系就是一种线性关系，所以数组的逻辑结构是一种线性结构，可用二元组表示为：

$$\text{array} = (A, R)$$

其中

$A = \{a_i \mid 1 \leqslant i \leqslant n, n = \text{card}(T_1), a_i \in T_2\}$

$R = \{r\}$

$r = \{<a_i, a_{i+1}> \mid 1 \leqslant i \leqslant n-1\}$

对应的图形如图1-6所示。

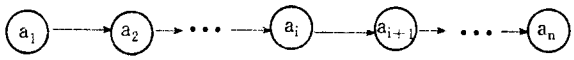

图 1-6　数组的逻辑结构示意图

数组的存储结构是一种顺序存储结构，即在存储空间上，数组的第 i＋1 个元素紧接着存储在第 i 个元素的存储位置的后面。这样，数组元素之间的线性关系通过顺序存储的方式很自然地反映出来。数组的存储结构可用图1-7表示。

由于数组中的每个元素都具有相同的类型 T_2，所以在存储空间上都占有相同的字节数（假定为 l）。若第 i 个元素 a_i 的存储地址（即对应的 l 个字节中的第一个字节的地址）用 Loc(a_i) 来表示，则第 i＋1 个元素 a_{i+1} 的存储地址为：

$$\text{Loc}(a_{i+1}) = \text{Loc}(a_i) + l$$

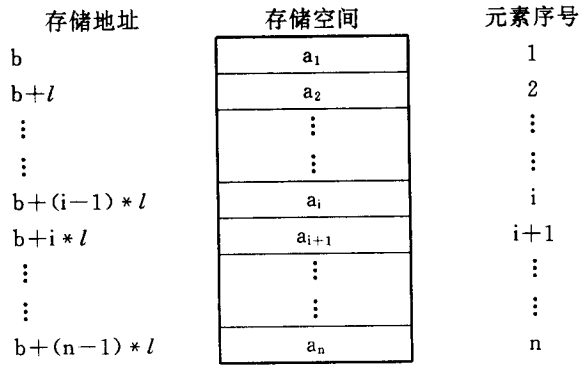

图 1-7　数组的存储结构示意图

若设 b 为数组存储空间的起始地址，亦即第一个元素 a_1 的存储地址，则数组中任一元素 a_i 的存储地址为：

$$Loc(a_i)=b+(i-1)*l$$

这样，只要给出下标 i 就可以立即计算出 a_i 的存储地址，所以可随机地访问数组中的任一元素，并且其访问时间均相等。

虽然 pascal 语言中数组类型的定义是一维的，但由于其成分类型可以是任何类型，当然也包括数组类型，所以通过数组类型的嵌套定义能够实际上定义出二维，乃至更高维结构的数组类型。如：

array [1..m] of array [1..n] of T;

就定义了一个二维数组类型。该类型的数组可一般表示为：

$$a=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

数组 a 的逻辑结构是一种嵌套的线性结构，即首先把它看作是按行号（即第一维下标）排列的具有 m 个元素的线性结构，其中第 i 个元素 a_i 为对应的第 i 行，然后再把 a_i 看作是按列号（即第二维下标）排列的具有 n 个元素的线性结构，其中的第 j 个元素为 a_{ij}。与此相应，数组 a 的存储结构也是嵌套的顺序存储结构，即第 i+1 行元素紧接着存储在第 i 行元素的存储位置的后面，而每一行中的所有元素则按照列号从小到大的顺序依次存储。假定存储数组 a 的起始地址（亦即 a_{11} 的地址）用 b 表示，每个元素占用的存储字节数用 l 表示，则任一行 a_i 所占用的存储字节数为 $n*l$，它的起始地址（即该行第一个元素 a_{i1} 的存储地址）为：

$$b+(i-1)*n*l$$

该行中任一元素 a_{ij} 的存储地址为：

$$Loc(a_{ij})=b+(i-1)*n*l+(j-1)*l$$

数组 a 的存储结构可用图1-8表示。

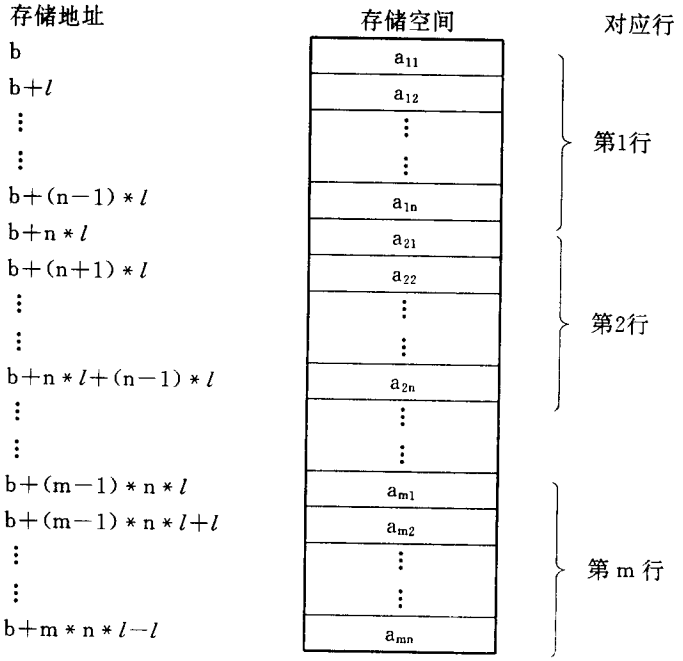

图 1-8　二维数组的存储结构示意图

对于更高维结构的数组也可进行类似地分析。例如，对于下面定义的具有三维结构的数组 a

a:array [1··s] of array [1··m] of array [1··n] of T;

或简写为：

a:array [1 ·· s,1 ·· m,1 ·· n] of T;

其中任一元素的存储地址为：

$$Loc(a_{ijk})=b+(i-1)*m*n*l+(j-1)*n*l+(k-1)*l$$

$$(1\leqslant i\leqslant s,1\leqslant j\leqslant m,1\leqslant k\leqslant n)$$

这里还需要指出，不是所有计算机语言的数组定义和存储分配都和 pascal 语言相同的，例如，在 fortran 语言中，数组的存储是按列进行的，即先存放第1列元素，再紧接着存放第2列元素，以此类推。如对于一个 m 行×n 列的数组 a，其任一元素 a_{ij} 的存储地址为：

$$Loc(a_{ij})=b+(j-1)*m*l+(i-1)*l$$

对应的存储空间分配如图1-9所示。

2. 记录

记录类型定义的一般格式为：

record

〈域名1〉:〈类型1〉;

〈域名2〉:〈类型2〉;

⋮ ⋮

〈域名 n〉:〈类型 n〉

end;

其中〈类型1〉至〈类型 n〉均可以是任何类型。

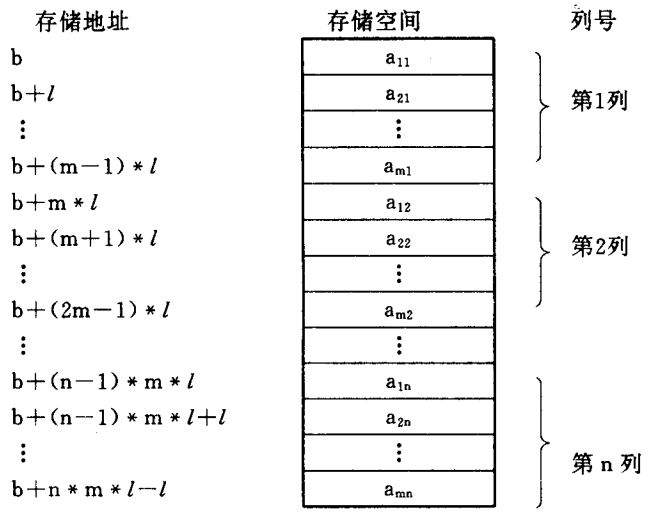

图 1-9 按列存储数组示意图

记录的成分(即域值)在位置上是顺序排列的,在存储空间上是顺序存放的,因此,记录同数组一样,其逻辑结构是线性结构,存储结构是顺序结构。记录同数组的区别为:

(1) 数组中的成分是同一类型的,而记录中的成分则允许具有不同的类型;

(2) 数组中的每个成分占有的存储单元具有相同的字节数,而记录中每个成分占有的存储单元,由于类型不同可能具有不同的字节数;

(3) 数组中的每个成分是通过下标来指明和访问的,而记录中的每个成分是通过域名来指明和访问的。

3. 集合

集合类型的定义为:

set of 〈基类型〉;

其中〈基类型〉是除整型和实型外的任何简单类型。集合类型的每一个值是一个集合。如集合类型:

set of 1..3;

所包含的全部值是下列八个集合:

[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]

一个集合只与所含的元素有关,而与元素的位置无关,如[1,2,3]和[3,2,1]表示同一个集合。因此,集合的逻辑结构是关系为空(即 r={})的结构,即元素之间不存在次序关系。

集合的存储是用一个与基类型值的个数等长的二进制位串来表示的,其中每个基类型的值与一个二进制位相对应,当一个集合中包含某个基值时,则对应的二进制位用"1"

表示，否则用“0”表示。例如有一个集合 s，其类型说明为：

s：set of 'a'..'f';

若 s 的值为['a','c','d','f']，则 s 的存储结构如图1-10所示。

二进制位序号	0	1	2	3	4	5
二进制位值	1	0	1	1	0	1
对应的基类型值	'a'	'b'	'c'	'd'	'e'	'f'

图 1-10

因此，集合的存储结构是顺序存储结构，即是以二进制位为单位，按照基类型值的序号(即图中的二进制位序号)顺序进行存储的结构。

集合同数组和记录相比，区别在于：

(1) 集合中的元素个数是不固定的，因为它可以是基类型值所组成的任何子集(包括空集)，而数组和记录中的成分个数是固定的；

(2) 集合中的元素在位置上是无序的，而数组和记录中的成分在位置上是有序的；

(3) 集合中的元素不能单独地访问和运算，而数组和记录中的成分都可以单独地访问和参加运算，单从这一点看，集合具有简单类型数据的特性，即只能作为整体进行访问和运算。

4. 文件

文件类型的定义为：

file of 〈成分类型〉;

文件的成分(一般为记录)在位置上是顺序排列的，所以单从这一点讲，文件的逻辑结构是线性结构。文件通常是被存储在外存的，其存储结构留待本书第八章专门讨论。这里需要指出的是，一个文件中成分的个数(被称为文件的长度)是可变的，它不象数组那样，其成分个数是固定的。开始建立的文件一般是一个空文件(即不包含任何成分，其长度为0)，当向一个文件写入一个新成分后，其长度就增加1；相反，当从一个文件删除一个成分后，其长度就减少1。

三、指针类型

指针类型是以存储单元的地址作为其值的一种数据类型。指针类型的定义为：

↑〈类型标识符〉;

假定 T 表示类型标识符，p 为指向 T 的指针变量，则 p 的类型说明为：

p：↑T;

那么 p↑就表示类型为 T 的一个动态变量(即结点)，p 的值就是该动态变量所对应的存储单元的首地址。

同简单变量一样，指针变量也是一种整体变量(即变量中不包含任何成分，不可再分)，系统为每个指针变量分配一个固定的存储单元，一般为两个字节。指针变量经常作为记录的成分，用来实现数据的动态链接存储。例如：

box=record