第1章 算 法

1.1 算法的基本概念

什么是算法呢？概括地说，算法是指解题方案的准确而完整的描述。

对于一个问题，如果可以通过一个计算机程序，在有限的存储空间内运行有限长的时间而得到正确的结果，则称这个问题是算法可解的。但算法不等于程序，也不等于计算方法。当然，程序也可以作为算法的一种描述，但程序通常还需考虑很多与方法和分析无关的细节问题，这是因为在编写程序时要受到计算机系统运行环境的限制。通常，程序的编制不可能优于算法的设计。

1.1.1 算法的基本特征

作为一个算法，一般应具有以下几个基本特征：

(1) 能行性(effectiveness)。算法的能行性包括以下两个方面：

① 算法中的每一个步骤必须能够实现。如在算法中不允许执行分母为 0 的操作，在实数范围内不能求一个负数的平方根等。

② 算法执行的结果要能够达到预期的目的。

针对实际问题设计的算法，人们总是希望能够得到满意的结果。但一个算法又总是在某个特定的计算工具上执行的，因此，算法在执行过程中往往要受到计算工具的限制，使执行结果产生偏差。例如，在进行数值计算时，如果某计算工具具有 7 位有效数字(如程序设计语言中的单精度运算)，则在计算下列 3 个量

$$A=10^{12},\quad B=1,\quad C=-10^{12}$$

的和时，如果采用不同的运算顺序，就会得到不同的结果，即

$$A+B+C=10^{12}+1+(-10^{12})=0$$

$$A+C+B=10^{12}+(-10^{12})+1=1$$

而在数学上，$A+B+C$ 与 $A+C+B$ 是完全等价的。因此，算法与计算公式是有差别的。在设计一个算法时，必须要考虑它的能行性，否则是不会得到满意结果的。

(2) 确定性(definiteness)。算法的确定性是指算法中的每一个步骤都必须是有明确定义的，不允许有模棱两可的解释，也不允许有多义性。这一性质也反映了算法与数学公式的明显差别。在解决实际问题时，可能会出现这样的情况：针对某种特殊问题，数学公式是正确的，但按此数学公式设计的计算过程可能会使计算机系统无所适从。这是因为根据数学公式设计的计算过程只考虑了正常使用的情况，而当出现异常情况时，此计算过程就不能适应了。

(3) 有穷性(finiteness)。算法的有穷性是指算法必须能在有限的时间内做完，即算

法必须能在执行有限个步骤之后终止。数学中的无穷级数在实际计算时只能取有限项，即计算无穷级数值的过程只能是有穷的。因此，一个数的无穷级数表示只是一个计算公式，而根据精度要求确定的计算过程才是有穷的算法。

算法的有穷性还应包括合理的执行时间的含义。因为，如果一个算法需要执行千万年，显然失去了实用价值。

(4) 拥有足够的情报。一个算法是否有效还取决于为算法所提供的情报是否足够。通常，算法中的各种运算总是要施加到各个运算对象上，而这些运算对象又可能具有某种初始状态，这是算法执行的起点或是依据。因此，一个算法执行的结果总是与输入的初始数据有关，不同的输入将会有不同的结果输出。当输入不够或输入错误时，算法本身也就无法执行或导致执行有错。一般来说，当算法拥有足够的情报时，此算法才是有效的，而当提供的情报不够时，算法并不有效。

综上所述，所谓算法，是一组严谨地定义运算顺序的规则，并且每一个规则都是有效的，且是明确的，此顺序将在有限的次数下终止。

1.1.2 算法的基本要素

一个算法通常由两种基本要素组成：一是对数据对象的运算和操作，二是算法的控制结构。

(1) 算法中对数据的运算和操作。每个算法实际上是按解题要求从环境能进行的所有操作中选择合适的操作所组成的一组指令序列。因此，计算机算法就是计算机能处理的操作所组成的指令序列。

通常，计算机可以执行的基本操作是以指令的形式描述的。一个计算机系统能执行的所有指令的集合称为该计算机系统的指令系统。计算机程序就是按解题要求从计算机指令系统中选择合适的指令所组成的指令序列。在一般的计算机系统中，基本的运算和操作有以下 4 类：

① 算术运算，主要包括加、减、乘、除等运算。

② 逻辑运算，主要包括“与”、“或”、“非”等运算。

③ 关系运算，主要包括“大于”、“小于”、“等于”、“不等于”等运算。

④ 数据传输，主要包括赋值、输入、输出等操作。

前面提到，计算机程序也可以作为算法的一种描述，但由于在编制计算机程序时通常要考虑很多与方法和分析无关的细节问题(如语法规则)，因此，在设计算法的一开始，通常并不直接用计算机程序来描述算法，而是用别的描述工具(如流程图、专门的算法描述语言，甚至用自然语言)来描述算法。但不管用哪种工具来描述算法，算法的设计一般都应从上述 4 种基本功能操作考虑，按解题要求从这些基本操作中选择合适的操作组成解题的操作序列。算法的主要特征着重于算法的动态执行，它区别于传统的着重于静态描述或按演绎方式求解问题的过程。传统的演绎数学是以公理系统为基础的，问题的求解过程是通过有限次推演来完成的，每次推演都将对问题作进一步的描述，如此不断推演，直到直接将解描述出来为止。而计算机算法则是使用一些最基本的操作，通过对已知条件一步一步的加工和变换，从而实现解题目标。这两种方法的解题思路是不同的。

(2) 算法的控制结构。一个算法的功能不仅取决于所选用的操作，而且还与各操作之间的执行顺序有关。算法中各操作之间的执行顺序称为算法的控制结构。

算法的控制结构给出了算法的基本框架，它不仅决定了算法中各操作的执行顺序，而且也直接反映了算法的设计是否符合结构化原则。描述算法的工具通常有传统流程图、N-S 结构化流程图、算法描述语言等。一个算法一般都可以用顺序、选择、循环 3 种基本控制结构组合而成。

1.2 算法设计基本方法

本节介绍工程上常用的几种算法设计方法，在实际应用时，各种方法之间往往存在着一定的联系。

1. 列举法

列举法的基本思想是，根据提出的问题，列举所有可能的情况，并用问题中给定的条件检验哪些是需要的，哪些是不需要的。因此，列举法常用于解决“是否存在”或“有多少种可能”等类型的问题，例如求解不定方程的问题。

列举法的特点是算法比较简单。但当列举的可能情况较多时，执行列举算法的工作量将会很大。因此，在用列举法设计算法时，使方案优化，尽量减少运算工作量，是应该重点注意的。通常，在设计列举算法时，只要对实际问题进行详细的分析，将与问题有关的知识条理化、完备化、系统化，从中找出规律；或对所有可能的情况进行分类，引出一些有用的信息，是可以大大减少列举量的。

下面举例说明利用列举算法解决问题时如何对算法进行优化。

例 1.1 设每只母鸡值 3 元，每只公鸡值 2 元，两只小鸡值 1 元。现要用 100 元钱买 100 只鸡，设计买鸡方案。

假设买母鸡 i 只，公鸡 j 只，小鸡 k 只。根据题意，粗略的列举算法如下：

```
procedure  baiji
for  i=0  to  100  do
for  j=0  to  100  do
for  k=0  to  100  do
  { m=i+j+k
    n=3i+2j+0.5k
    if ((m=100)and(n=100))  then
         output  i,j,k
  }
return
```

在这个算法中，共嵌套有 3 层循环，每层循环各需要循环 101 次，因此，总循环次数为 101^3。但只要对问题进行分析，发现还可以对这个算法进行优化，减少大量不必要的循环次数。

首先，考虑到母鸡为 3 元一只，因此，母鸡最多只能买 33 只，即算法中的外循环没有必要从 0 到 100，而只需要从 0 到 33 就可以了。

其次，考虑到公鸡为2元一只，因此，公鸡最多只能买50只。又考虑到对公鸡的列举是在算法的第二层循环中，此时已经买了 i 只母鸡，且买一只母鸡的价钱相当于买1.5只公鸡。因此，由第一层循环已经确定买 i 只母鸡的前提下，公鸡最多只能买 $50-1.5i$，即第二层对 j 的循环只需从0到 $50-1.5i$ 就可以了。

最后，考虑到买的总鸡数为100，而由第一层循环已确定买 i 只母鸡，由第二层循环已确定买 j 只公鸡，因此，买小鸡的数量只能是 $k=100-i-j$，即第三层循环已经没有必要。

经过以上分析，可以将上述算法改写成如下形式：

```
procedure  baiji
for  i=0  to  33  do
for  j=0  to  50-1.5i  do
  { k=100-i-j
    if  (3i+2j+0.5k=100)  then
        output  i,j,k
  }
return
```

不难分析，经修改后的算法的列举量(循环次数)为

$$\sum_{i=0}^{33}(51-1.5i)\approx 894$$

经修改后的算法的C++描述如下：

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{ int i, j, k;
  for (i=0; i<=33; i++)
  for (j=0; j<=50-1.5*i; j++)
    { k=100-i-j;
      if (3*i+2*j+0.5*k==100.0)
        cout<<setw(5)<<i<<setw(5)<<j<<setw(5)<<k<<endl;
    }
  return 0;
}
```

这个程序的运行结果如下：

```
 2   30   68
 5   25   70
 8   20   72
11   15   74
14   10   76
17    5   78
```

```
20    0    80
```

列举原理是计算机应用领域中十分重要的原理。对许多实际问题来说，采用人工列举是不可想象的。由于计算机的运算速度快，擅长重复操作，可以很方便地进行大量列举。列举算法虽然是一种比较笨拙而原始的方法，其运算量比较大，但在有些实际问题中（如寻找路径、查找、搜索等问题），局部的使用列举法却是很有效的，因此，列举算法是计算机算法中的一个基础算法。

2. 归纳法

归纳法的基本思想是，通过列举少量的特殊情况，经过分析，最后找出一般的关系。显然，归纳法要比列举法更能反映问题的本质，并且可以解决列举量为无限的问题。但是，从一个实际问题中总结归纳出一般的关系并不是一件容易的事情，要归纳出一个数学模型则更为困难。从本质上讲，归纳就是通过观察一些简单而特殊的情况，最后总结出有用的结论或解决问题的有效途径。

归纳是一种抽象，即从特殊现象中找出一般关系。但由于在归纳的过程中不可能对所有的情况进行列举，因此，最后由归纳得到的结论还只是一种猜测，还需要对这种猜测加以必要的证明。实际上，通过精心观察而得到的猜测得不到证实或最后证明猜测是错的，也是常有的事。

3. 递推

所谓递推，是指从已知的初始条件出发，逐次推出所要求的各中间结果和最后结果。其中初始条件或是问题本身已经给定，或是通过对问题的分析与化简而得到确定。递推本质上也属于归纳法，工程上许多递推关系式实际上是通过对实际问题的分析与归纳而得到的，因此，递推关系式往往是归纳的结果。

递推算法在数值计算中是极为常见的。但是，对于数值型的递推算法必须要注意数值计算的稳定性问题。

4. 递归

人们在解决一些复杂问题时，为了降低问题的复杂程度（如问题的规模等），一般总是将问题逐层分解，最后归结为一些最简单的问题。这种将问题逐层分解的过程实际上并没有对问题进行求解，而只是当解决了最后那些最简单的问题后，再沿着原来分解的逆过程逐步进行综合，这就是递归的基本思想。由此可以看出，递归的基础也是归纳。在工程实际中，有许多问题就是用递归来定义的，数学中的许多函数也是用递归来定义的。递归在可计算性理论和算法设计中占有很重要的地位。

下面用一个简单的例子来说明递归的基本思想。

例 1.2 编写一个过程，对于输入的参数 n，依次打印输出自然数 1 到 n。

这是一个很简单的问题，实际上不用递归就能解决，其算法如下：

```
PROCEDURE  WRT(n)
FOR  k=1  TO  n  DO  OUTPUT  k
RETURN
```

这个算法用 C++ 描述如下：

```
#include <iostream>
using namespace std;
void wrt(int  n)
{ int  k;
  for (k=1; k<=n; k++)  cout  <<k<<endl;
  return;
}
```

解决这个问题还可以用递归过程来实现，其算法如下：

输出自然数 1 到 n 的递归算法：

```
PROCEDURE  WRT1(n)
IF  (n≠0)  THEN
   { WRT1(n-1)
     OUTPUT  n
   }
RETURN
```

在递归算法 WRT1 中，n 是形参。在开始执行算法 WRT1 时，首先要判断形参变量值(开始时为 n)是否不等于 0，如果不等于 0，则将形参值减 1($n-1$)后作为新的实参再调用算法 WRT1；在调用函数 WRT1 时，又需判断形参值(此时已变为 $n-1$)是否不等于 0，如果不等于 0，则又将形参值减 1($n-2$)后作为新的实参再次调用算法 WRT1；……。这个过程一直进行下去，直到算法 WRT1 的形参值等于 0 为止。此时，由于在先前各层的函数调用中，算法 WRT1 实际上没有执行完，即各层中的形参值还没有被打印输出，这就需要逐层返回，以便打印输出各层中的输入参数 1，2，…，n。为此，在递归算法的执行过程中，需要记忆各层调用中的参数，以便在逐层返回时恢复这些参数继续进行处理。具体来说，在算法 WRT1 开始执行后，随着各次的递归调用，逐次记忆各层调用中的输入参数 n，$n-1$，$n-2$，…，2，1，在逐层返回时，又依次(按记忆的相反次序)将这些参数打印输出。

递归算法 WRT1 用 C++ 描述如下：

```
#include <iostream>
using namespace std;
void wrt1(int n)
{ if (n!=0)
    { wrt1(n-1);  cout<<n<<endl; }
  return;
}
```

在程序设计中，递归是一个很有用的工具。对于一些比较复杂的问题，设计成递归算法后结构清晰，可读性强。

自己调用自己的过程称为递归调用过程。

递归分为直接递归与间接递归两种。如果一个算法 P 显式地调用自己则称为直接递归。例如，上述递归算法 WRT1 是一个直接递归的算法。如果算法 P 调用另一个算法 Q，而算法 Q 又调用算法 P，则称为间接递归调用。

递归是一种很重要的算法设计方法。实际上，递归过程能将一个复杂的问题归结为若干个较简单的问题，然后将这些较简单的每一个问题再归结为更简单的问题，这个过程可以一直做下去，直到归结为最简单的问题为止。

有些实际问题既可以归纳为递推算法，又可以归纳为递归算法。但递推与递归的实现方法是大不一样的。递推是从初始条件出发，逐次推出所需求的结果；而递归则是从算法本身到达递归边界的。通常，递归算法要比递推算法清晰易读，其结构比较简练。特别是在许多比较复杂的问题中，很难找到从初始条件推出所需结果的全过程，此时，设计递归算法要比递推算法容易得多。但递归算法的执行效率比较低。

5. 减半递推技术

解决实际问题的复杂程度往往与问题的规模有着密切的关系。例如，两个 n 阶矩阵相乘，通常需要作 n^3 次乘法，两个二阶矩阵相乘需要作 8 次乘法。设两个二阶矩阵为

$$\boldsymbol{A}=\begin{bmatrix}a_{11} & a_{12}\\ a_{21} & a_{22}\end{bmatrix},\qquad \boldsymbol{B}=\begin{bmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\end{bmatrix}$$

其乘积矩阵 $\boldsymbol{C}=\boldsymbol{AB}$ 为

$$\boldsymbol{C}=\begin{bmatrix}c_{11} & c_{12}\\ c_{21} & c_{22}\end{bmatrix}=\begin{bmatrix}a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22}\\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22}\end{bmatrix}$$

能不能减少乘法次数呢？如果直接考虑一般的 n 阶矩阵相乘的问题，这是很困难的。但对于低阶的矩阵相乘问题，如二阶矩阵相乘，减少乘法次数是有可能的。实际上，若令

$$\begin{cases}x_1=(a_{11}+a_{22})(b_{11}+b_{22})\\ x_2=(a_{21}+a_{22})b_{11}\\ x_3=a_{11}(b_{12}-b_{22})\\ x_4=a_{22}(b_{21}-b_{11})\\ x_5=(a_{11}+a_{12})b_{22}\\ x_6=(a_{21}-a_{11})(b_{11}+b_{12})\\ x_7=(a_{12}+a_{22})(b_{21}+b_{22})\end{cases}\tag{1-1}$$

可以验证，乘积矩阵 $\boldsymbol{C}$ 中的各元素可用以上 7 个量的线性组合来表示，即

$$\begin{cases}c_{11}=x_1+x_4-x_5+x_7\\ c_{12}=x_3+x_5\\ c_{21}=x_2+x_4\\ c_{22}=x_1+x_3-x_2+x_6\end{cases}\tag{1-2}$$

由此可以看出，利用上述方法计算两个二阶矩阵相乘只需要 7 次乘法就够了，比通常的方法减少了 1 次乘法。由于在式(1-1)的计算中没有用到乘法的交换性(矩阵 $\boldsymbol{A}$ 中的元素始终在前面，矩阵 $\boldsymbol{B}$ 中的元素始终在后面)，因此，对于一般的 n 阶矩阵可以划分为 4 块 $n/2$ 阶的矩阵，然后利用分块矩阵相乘，可以得到 7 个 $n/2$ 阶矩阵相乘的问题。同样，对于每个 $n/2$ 阶矩阵相乘的问题，又可以化为 7 个 $n/4$ 阶矩阵相乘的问题。以此类推，最后可以归结为计算一阶矩阵相乘的问题。而一阶矩阵相乘只需要一次乘法。

根据以上分析，假设 $n=2^k$，且 $n=2^k$ 阶矩阵相乘所需要的乘法次数为 $M(k)$，则有

$$M(k) = 7M(k-1) = 7^2M(k-2) = \cdots = 7^kM(0)$$

且 $M(0)=1$。因此有

$$M(k) = 7^k = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.81} \tag{1-3}$$

显然，这个乘法次数比通常的 n^3 次要少得多，当 n 较大时更为显著。

算法设计的这种方法称为分治法，即对问题分而治之。工程上常用的分治法是减半递推技术。这个技术在快速算法的研究中有很重要的实用价值。

所谓“减半”，是指将问题的规模减半，而问题的性质不变。所谓“递推”，是指重复“减半”的过程。

下面举例说明利用减半递推技术设计算法的基本思想。

例 1.3 设方程 $f(x)=0$ 在区间$[a,b]$上有实根，且 $f(a)$与 $f(b)$异号。利用二分法求该方程在区间$[a,b]$上的一个实根。

二分法求方程实根的减半递推过程如下：

首先取给定区间的中点 $c=(a+b)/2$。

然后判断 $f(c)$是否为 0。若 $f(c)=0$，则说明 c 即为所求的根，求解过程结束；如果 $f(c)\neq 0$，则根据以下原则将原区间减半：

- 若 $f(a)f(c)<0$，则取原区间的前半部分。
- 若 $f(b)f(c)<0$，则取原区间的后半部分。

最后判断减半后的区间长度是否已经很小：

- 若 $|a-b|<\varepsilon$，则过程结束，取$(a+b)/2$ 为根的近似值。
- 若 $|a-b|\geqslant\varepsilon$，则重复上述的减半过程。

这个算法留给读者自己去描述。

6. 回溯法

前面讨论的递推和递归算法本质上是对实际问题进行归纳的结果，而减半递推技术也是归纳法的一个分支。在工程上，有些实际问题很难归纳出一组简单的递推公式或直观的求解步骤，并且也不能进行无限的列举。对于这类问题，一种有效的方法是“试”。通过对问题的分析，找出一个解决问题的线索，然后沿着这个线索逐步试探，对于每一步的试探，若试探成功，就得到问题的解，若试探失败，就逐步回退，换别的路线再进行试探。这种方法称为回溯法。回溯法在处理复杂数据结构方面有着广泛的应用。

下面举例说明回溯法的基本思想。

例 1.4 求解皇后问题。

由 n^2 个方块排成 n 行 n 列的正方形称为“n 元棋盘”。如果两个皇后位于棋盘上的同一行或同一列或同一对角线上，则称它们为互相攻击。现要求找使 n 元棋盘上的 n 个皇后互不攻击的所有布局。

n 个皇后在 n 元棋盘上的布局共有 n^n 种，为了从中找出互不攻击的布局，需要对此 n^n 种方案逐个进行检查，将有攻击的布局剔除掉。这是一种列举法。但这种方法对于较大的 n，其运算量会急剧地增加。而在实际上，逐一列举也是没有必要的。例如，如果第一行上的皇后在第一列，则第二行上的皇后就不可能在第一列。

下面用回溯法来求解皇后问题。

假设棋盘上的每一行放置一个皇后，分别用自然数进行编号为 1,2,…,n。

首先定义一个长度为 n 的一维数组 q，其中每一个元素 $q[i](i=1,2,\cdots,n)$ 随时记录第 i 行上的皇后所在的列号。

初始时，先将各皇后放在各行的第 1 列，即数组 q 的初值为

$$q[i]=1,\quad i=1,2,\cdots,n$$

第 i 行与第 j 行上的皇后在某一对角线上的条件为

$$|q[i]-q[j]|=|i-j|$$

而它们在同一列上的条件为

$$q[i]=q[j]$$

回溯法的步骤如下：

从第一行($i=1$)开始进行以下过程。

设前 $i-1$ 行上的皇后已经布局好，即它们互不攻击。现在考虑安排第 i 行上皇后的位置，使之与前 $i-1$ 行上的皇后也都互不攻击。为了实现这一目的，可以从第 i 行皇后的当前位置 $q[i]$ 开始按如下规则向右进行搜索：

① 若 $q[i]=n+1$，则说明第 i 个皇后暂时无法安排，将第 i 个皇后放在第 1 列(置 $q[i]=1$)，且回退一行，考虑重新安排第 $i-1$ 行上的皇后与前 $i-2$ 行上的皇后均互不攻击的下一个位置。此时如果已退到第 0 行(实际没有这一行)，则过程结束。

② 若 $q[i]\leqslant n$，则需检查第 i 行上的皇后与前 $i-1$ 行的皇后是否互不攻击。若有攻击，则将第 i 行上的皇后右移一个位置(置 $q[i]=q[i]+1$)，重新进行这个过程；若无攻击，则考虑安排下一行上的皇后位置，即置 $i=i+1$。

③ 若当前安排好的皇后是在最后一行(第 n 行)，则说明已经找到了 n 个皇后互不攻击的一个布局，将这个布局输出(输出 $q[i],i=1,2,\cdots,n$)。然后将第 n 行上的皇后右移一个位置(置 $q[n]=q[n]+1$)，重新进行这个过程，以便寻找下一个布局。

以上介绍了几种工程上常用的算法设计的基本方法。实际上，算法设计的方法还有很多，如数字模拟法，用于数值近似的数值法等，在此不再一一介绍了，有兴趣的读者可以参看有关算法方面的著作。

1.3 算法的复杂度分析

算法的复杂度主要包括时间复杂度和空间复杂度。

1.3.1 算法的时间复杂度

所谓算法的时间复杂度，是指执行算法所需要的计算工作量。

为了能够比较客观地反映出一个算法的效率，在度量一个算法的工作量时，不仅应该与所使用的计算机、程序设计语言以及程序编制者无关，而且还应该与算法实现过程中的许多细节无关。为此，可以用算法在执行过程中所需基本运算的执行次数来度量算法的工作量。基本运算反映了算法运算的主要特征，因此，用基本运算的次数来度量算法工作量是客观的也是实际可行的，有利于比较同一问题的几种算法的优劣。例如，在考虑两个

矩阵相乘时，可以将两个实数之间的乘法运算作为基本运算，而对于所用的加法（或减法）运算可以忽略不计。又如，当需要在一个表中进行查找时，可以将两个元素之间的比较作为基本运算。

算法所执行的基本运算次数还与问题的规模有关。例如，两个 20 阶矩阵相乘与两个 10 阶矩阵相乘所需要的基本运算（两个实数的乘法）次数显然是不同的，前者需要更多的运算次数。因此，在分析算法的工作量时，还必须对问题的规模进行度量。

综上所述，算法的工作量用算法所执行的基本运算次数来度量，而算法所执行的基本运算次数是问题规模的函数，即

$$\text{算法的工作量} = f(n)$$

其中 n 是问题的规模。例如，两个 n 阶矩阵相乘所需要的基本运算（两个实数的乘法）次数为 n^3，即计算工作量为 n^3，也就是时间复杂度为 n^3。

在具体分析一个算法的工作量时，还会存在这样的问题：对于一个固定的规模，算法所执行的基本运算次数还可能与特定的输入有关，而实际上又不可能将所有可能情况下算法所执行的基本运算次数都列举出来。例如，“在长度为 n 的一维数组中查找值为 x 的元素”，若采用顺序搜索法，即从数组的第一个元素开始，逐个与被查值 x 进行比较。显然，如果第一个元素恰为 x，则只需要比较 1 次。但如果 x 为数组的最后一个元素，或者 x 不在数组中，则需要比较 n 次才能得到结果。因此，在这个问题的算法中，其基本运算（即比较）的次数与具体的被查值 x 有关。

在同一问题规模下，如果算法执行所需的基本运算次数取决于某一特定输入时，可以用以下两种方法来分析算法的工作量。

（1）平均性态（average behavior）。所谓平均性态分析，是指用各种特定输入下的基本运算次数的带权平均值来度量算法的工作量。

设 x 是所有可能输入中的某个特定输入，$p(x)$ 是 x 出现的概率（输入为 x 的概率），$t(x)$ 是算法在输入为 x 时所执行的基本运算次数，则算法的平均性态定义为

$$A(n) = \sum_{x \in D_n} p(x)t(x)$$

其中 D_n 表示当规模为 n 时，算法执行时所有可能输入的集合。这个式子中的 $t(x)$ 可以通过分析算法来加以确定；而 $p(x)$ 必须由经验或用算法中有关的一些特定信息来加以确定，通常是不能解析地加以计算的。如果确定 $p(x)$ 比较困难，则会给平均性态的分析带来困难。

（2）最坏情况复杂性（worst-case complexity）。所谓最坏情况分析，是指在规模为 n 时，算法所执行的基本运算的最大次数。它定义为

$$W(n) = \max_{x \in D_n} \{t(x)\}$$

显然，$W(n)$ 的计算要比 $A(n)$ 的计算方便得多。由于 $W(n)$ 实际上是给出了算法工作量的一个上界，因此，它比 $A(n)$ 更具有实用价值。

下面通过一个例子来说明算法复杂度的平均性态分析与最坏情况分析。

例 1.5 采用顺序搜索法，在长度为 n 的一维数组中查找值为 x 的元素。即从数组的第一个元素开始，逐个与被查值 x 进行比较。基本运算为 x 与数组元素的比较。

首先考虑平均性态分析。

设被查项 x 在数组中的概率为 q 。当需要查找的 x 为数组中第 i 个元素时，则在查找过程中需要做 i 次比较，当需要查找的 x 不在数组中时（数组中没有 x 这个元素），则需要和数组中所有的元素进行比较，即

$$t_i = \begin{cases} i, & 1 \leqslant i \leqslant n \\ n, & i = n+1 \end{cases}$$

其中 $i=n+1$ 表示 x 不在数组中。

如果假设需要查找的 x 出现在数组中每个位置上的可能性是一样的，则 x 出现在数组中每一个位置上的概率为 q/n（因为前面已经假设 x 在数组中的概率为 q），而 x 不在数组中的概率为 $1-q$。即

$$p_i = \begin{cases} q/n, & 1 \leqslant i \leqslant n \\ 1-q, & i = n+1 \end{cases}$$

其中 $i=n+1$ 表示 x 不在数组中。

因此，用顺序搜索法在长度为 n 的一维数组中查找值为 x 的元素，在平均情况下需要做的比较次数为

$$A(n) = \sum_{i=1}^{n+1} p_i t_i = \sum_{i=1}^{n} (q/n)i + (1-q)n = (n+1)q/2 + (1-q)n$$

如果已知需要查找的 x 一定在数组中，此时 $q=1$，则 $A(n)=(n+1)/2$。这就是说，在这种情况下，用顺序搜索法在长度为 n 的一维数组中查找值为 x 的元素，在平均情况下需要检查数组中一半的元素。

如果已知需要查找的 x 有一半的机会在数组中，此时 $q=1/2$，则

$$A(n) = [(n+1)/4] + n/2 \approx 3n/4$$

这就是说，在这种情况下，用顺序搜索法在长度为 n 的一维数组中查找值为 x 的元素，在平均情况下需要检查数组中 3/4 的元素。

再考虑最坏情况分析。

在这个例子中，最坏情况发生在需要查找的 x 是数组中的最后一个元素或 x 不在数组中的时候，此时显然有

$$W(n) = \max\{t_i \mid 1 \leqslant i \leqslant n+1\} = n$$

在上述例子中，算法执行的工作量是与具体的输入有关的，$A(n)$ 只是它的加权平均值，而实际上对于某个特定的输入，其计算工作量未必是 $A(n)$，且 $A(n)$ 也不一定等于 $W(n)$。但在另外一些情况下，算法的计算工作量与输入无关，即当规模为 n 时，在所有可能的输入下，算法所执行的基本运算次数是一定的，此时有 $A(n)=W(n)$。例如，两个 n 阶的矩阵相乘，都需要做 n^3 次实数乘法，而与输入矩阵的具体元素无关。

1.3.2 算法的空间复杂度

一个算法的空间复杂度一般是指执行这个算法所需要的内存空间。

一个算法所占用的存储空间包括算法程序所占的空间、输入的初始数据所占的存储空间以及算法执行过程中所需要的额外空间。其中额外空间包括算法程序执行过程中的

工作单元以及某种数据结构所需要的附加存储空间(例如,在链式结构中,除了要存储数据本身外,还需要存储链接信息)。如果额外空间量相对于问题规模来说是常数,则称该算法是原地(in place)工作的。在许多实际问题中,为了减少算法所占的存储空间,通常采用压缩存储技术,以便尽量减少不必要的额外空间。

习　题

1.1　设给定 3 个整数 a,b,c,试写出寻找其中数的一个算法(用 C++ 描述),并分析在平均情况与最坏情况下,该算法分别要作多少次比较。

1.2　利用减半递推技术,写出求长度为 n 的数组中最大元素的递归算法(用 C++ 描述),设 $n=2^k$,其中 $k\geqslant 1$。

1.3　编写二分法求方程实根的减半递推算法(用 C++ 描述)。

1.4　编写用回溯法求解皇后问题的算法(用 C++ 描述)。

1.5　设有 12 个小球,其中 11 个小球的重量相同,称为好球;有一个小球的重量与 11 个好球的重量不同(或轻或重),称这个小球为坏球。试编写一个算法(用 C++ 描述),用一个无砝码的天平称 3 次找出这个坏球,并确定其比好球轻还是重。

第 2 章　基本数据结构及其运算

利用计算机进行数据处理是计算机应用的一个重要领域。在进行数据处理时，实际需要处理的数据元素一般有很多，而这些大量的数据元素都需要存放在计算机中，因此，大量的数据元素在计算机中如何组织，以便提高数据处理的效率，并且节省计算机的存储空间，这是进行数据处理的关键问题。

显然，杂乱无章的数据是不便于处理的。将大量的数据随意地存放在计算机中，对数据处理更是不利。

数据结构作为计算机的一门学科，主要研究和讨论以下 3 个方面的问题。

(1) 数据集合中各数据元素之间所固有的逻辑关系，即数据的逻辑结构。

(2) 在对数据进行处理时，各数据元素在计算机中的存储关系，即数据的存储结构。

(3) 对各种数据结构进行的运算。

讨论以上各问题的主要目的是为了提高数据处理的效率。所谓提高数据处理的效率，主要包括两个方面：一是提高数据处理的速度，二是尽量节省在数据处理过程中所占用的计算机存储空间。

本章主要讨论工程上实用的一些基本数据结构，它们是软件设计的基础。

2.1　数据结构的基本概念

2.1.1　两个例子

计算机已被广泛用于数据处理。实际问题中的各数据元素之间总是相互关联的。所谓数据处理，是指对数据集合中的各元素以各种方式进行运算，包括插入、删除、查找、更改等运算，也包括对数据元素进行分析。在数据处理领域中，建立数学模型有时并不十分重要，事实上，许多实际问题是无法表示成数学模型的。人们最感兴趣的是知道数据集合中各数据元素之间存在什么关系，为了提高处理效率，应如何组织它们，即如何表示所需要处理的数据元素。

本节将通过两个实例来说明对同一批数据用不同的表示方法后，对处理效率的影响。

例 2.1　无序表的顺序查找与有序表的对分查找。

图 2.1 是两个子表。从图中可以看出，在这两个子表中所存放的数据元素是相同的，但它们在表中存放的顺序是不同的。在图 2.1(a)所示的表中，数据元素的存放顺序是没有规则的；而在图 2.1(b)所示的表中，数据元素是按从小到大的顺序存放的。我们称前者为无序表，后者为有序表。

下面考虑在这两种表中进行查找的问题。

首先考虑在图 2.1(a)所示的无序表中进行查找。由于在图 2.1(a)表中数据元素的存放顺序没有一定的规则，因此，要在这个表中查找某个数时，只能从第一个元素开始，逐个将表中的元素与被查数进行比较，直到表中的某个元素与被查数相等(查找成功)或者表中所有元素与被查数都进行了比较且都不相等(查找失败)为止。这种查找方法称为顺序查找。显然，在顺序查找中，如果被查找数在表的前部，则需要比较的次数就少；如果被查找数在表的后部，则需要比较的次数就多。特别是当被查找数刚好是表中的第一个元素时(如被查数为 35)，只需要比较一次就查找成功；但当被查数刚好是表中最后一个元素(如被查数为 46)或表中根本没有被查数时(如被查数为 67)，则需要与表中所有的元素进行比较，在这种情况下，当表很大时，顺序查找是很费时间的。虽然顺序查找法的效率比较低，但由于图 2.1(a)为无序表，没有更好的查找方法，只能用顺序查找。

35
16
78
85
43
29
33
21
54
46

(a) 无序表

16
21
29
33
35
43
46
54
78
85

(b) 有序表

图 2.1　数据元素存放顺序不同的两个表

现在再考虑在图 2.1(b)所示的有序表中进行查找。由于有序表中的元素是从小到大进行排列的，在查找时可以利用这个特点，以便使比较次数大大减少。在有序表中查找一个数可以如下进行：

将被查数与表中间的这个元素进行比较：若相等，则表示查找成功，查找过程结束；若被查数大于表中间的这个元素，则表示如果被查数在表中的话，只能在表的后半部，此时可以抛弃表的前半部而保留后半部；若被查数小于表中间的这个元素，则表示如果被查数在表中的话，只能在表的前半部，此时可以抛弃表的后半部而保留前半部。然后对剩下的部分(前半部或后半部)再按照上述方法进行查找，这个过程一直做到在某一次的比较中相等(查找成功)或剩下的部分已空(查找失败)为止。例如，如果要在图 2.1(b)所示的有序表中查找 54，则首先与中间元素 35 进行比较，由于 54 大于 35，再与后半部分的中间元素 54 进行比较，此时相等，共比较了两次就查找成功。如果采用顺序查找法，在图 2.1(a)所示的无序表中查找 54 这个元素，需要比较 9 次。这种查找方法称为有序表的对分查找。

显然，在有序表的对分查找中，不论查找的是什么数，也不论要查找的数在表中有没有，都不需要与表中所有的元素进行比较，并且只需要与表中很少的元素进行比较。需要指出的是，对分查找只适用于有序表，对于无序表是无法进行对分查找的。

实际上，在日常工作和学习中也经常遇到对分查找。例如，当需要在词典中查找一个单词时，一般不是从第一页开始一页一页地往后找，而是考虑到词典中的各单词是以英文字母为顺序排列的，因此可以根据所查单词的第一个字母，直接翻到大概的位置，然后进行比较，根据比较结果再向前或向后翻，直到找到该单词为止。这种在词典中查单词的方法类似于对分查找。

由这个例子可以看出，数据元素在表中的排列顺序对查找效率是有很大影响的。

例 2.2 设有一个学生情况登记表，如表 2.1 所示。在表 2.1 中，每个学生的情况是以学号为顺序排列的。

显然，如果要在表 2.1 中查找给定学号的某学生的情况是很方便的，只要根据给定的学号就可以立即找到该学生的情况。但是，如果要在该表中查找成绩在 90 分以上的所有学生的情况，则需要从头到尾扫描全表，才能将成绩在 90 分以上的所有学生找到。在这种情况下，为了找到成绩在 90 分以上的学生情况，对于成绩在 90 分以下的所有学生情况也都要被扫描到。由此可以看出，要在表 2.1 中查找给定学号的学生情况虽然很方便，但要查找成绩在某个分数段中的学生情况时，实际上需要查看表中所有学生的成绩，其效率是很低的，尤其是当表很大时更为突出。

表 2.1 学生情况登记表

学　号	姓　名	性　别	年　龄	成　绩
970156	张小明	男	20	86
970157	李小青	女	19	83
970158	赵　凯	男	19	70
970159	李启明	男	21	91
970160	刘　华	女	18	78
970161	曾小波	女	19	90
970162	张　军	男	18	80
970163	王　伟	男	20	65
970164	胡　涛	男	19	95
970165	周　敏	女	20	87
970166	杨雪辉	男	22	89
970167	吕永华	男	18	61
970168	梅　玲	女	17	93
970169	刘　健	男	20	75

为了便于查找成绩在某个分数段中的学生情况，可以将表 2.1 中所登记的学生情况进行重新组织。例如，将成绩在 90 分以上(包括 90 分，下同)、80～89 分、70～79 分、60～69 分之间的学生情况分别登记在 4 个独立的子表中，分别如表 2.2、表 2.3、表 2.4 与表 2.5 所示。现在如果要查找 90 分以上的所有学生的情况，就可以直接在表 2.2 中进行查找，从而避免了对成绩在 90 分以下的学生情况进行扫描，提高了查找效率。

由例 2.2 可以看出，在对数据进行处理时，可以根据所做的运算不同，将数据组织成不同的形式，以便于做该种运算，从而提高数据处理的效率。

表 2.2　成绩在 90 分以上的学生情况登记表

学　号	姓　名	性　别	年　龄	成　绩
970159	李启明	男	21	91
970161	曾小波	女	19	90
970164	胡　涛	男	19	95
970168	梅　玲	女	17	93

表 2.3　成绩在 80～89 分之间的学生情况登记表

学　号	姓　名	性　别	年　龄	成　绩
970156	张 小明	男	20	86
970157	李小青	女	19	83
970162	张　军	男	18	80
970165	周　敏	女	20	87
970166	杨雪辉	男	22	89

表 2.4　成绩在 70～79 分之间的学生情况登记表

学　号	姓　名	性　别	年　龄	成　绩
970158	赵　凯	男	19	70
970160	刘　华	女	18	78
970169	刘　健	男	20	75

表 2.5　成绩在 60～69 分之间的学生情况登记表

学　号	姓　名	性　别	年　龄	成　绩
970163	王　伟	男	20	65
970167	吕永华	男	18	61

2.1.2　什么是数据结构

简单地说，数据结构是指相互有关联的数据元素的集合。例如，向量和矩阵就是数据结构，在这两个数据结构中，数据元素之间有着位置上的关系。又如，图书馆中的图书卡片目录则是一个较为复杂的数据结构，对于列在各卡片上的各种书之间，可能在主题、作者等问题上相互关联，甚至一本书本身也有不同的相关成分。

数据元素具有广泛的含义。一般来说，现实世界中客观存在的一切个体都可以是数据元素。例如：

描述一年四季的季节名

春，夏，秋，冬

可以作为季节的数据元素；

表示数值的各个数

18,11,35,23,16,…

可以作为数值的数据元素；

表示家庭成员的各成员名

父亲,儿子,女儿

可以作为家庭成员的数据元素。

甚至每一个客观存在的事件，如一次演出、一次借书、一次比赛等也可以作为数据元素。

总之，在数据处理领域中，每一个需要处理的对象都可以抽象成数据元素。数据元素一般简称为元素。

在实际应用中，被处理的数据元素一般有很多，而且，作为某种处理，其中的数据元素一般具有某些共同特征。例如，{春，夏，秋，冬}这 4 个数据元素有一个共同特征，即它们都是季节名，分别表示一年中的 4 个季节，从而这 4 个数据元素构成了季节名的集合。又如，{父亲，儿子，女儿}这 3 个数据元素也有一个共同特征，即它们都是家庭的成员名，从而构成了家庭成员名的集合。一般来说，人们不会同时处理特征完全不同且互相之间没有任何关系的各类数据元素，对于具有不同特征的数据元素总是分别进行处理。

一般情况下，在具有相同特征的数据元素集合中，各个数据元素之间存在有某种关系（联系），这种关系反映了该集合中的数据元素所固有的一种结构。在数据处理领域中，通常把数据元素之间这种固有的关系简单地用前后件关系（或直接前驱与直接后继关系）来描述。例如：

在考虑一年 4 个季节的顺序关系时，“春”是“夏”的前件（直接前驱，下同），而“夏”是“春”的后件（直接后继，下同）。同样，“夏”是“秋”的前件，“秋”是“夏”的后件；“秋”是“冬”的前件，“冬”是“秋”的后件。

在考虑家庭成员间的辈分关系时，“父亲”是“儿子”和“女儿”的前件，而“儿子”与“女儿”都是“父亲”的后件。

前后件关系是数据元素之间的一个基本关系，但前后件关系所表示的实际意义随具体对象的不同而不同。一般来说，数据元素之间的任何关系都可以用前后件关系来描述。

1. 数据的逻辑结构

前面提到，数据结构是指反映数据元素之间关系的数据元素集合的表示。更通俗地说，数据结构是指带有结构的数据元素的结合。在此，所谓结构实际上就是指数据元素之间的前后件关系。

由上所述，一个数据结构应包含以下两方面的信息：

(1) 表示数据元素的信息。

(2) 表示各数据元素之间的前后件关系。

在以上所述的数据结构中，其中数据元素之间的前后件关系是指它们的逻辑关系，而与它们在计算机中的存储位置无关。因此，上面所述的数据结构实际上是数据的逻辑

结构。

所谓数据的逻辑结构，是指反映数据元素之间逻辑关系的数据结构。

由前面的叙述可以知道，数据的逻辑结构有两个要素：一是数据元素的集合，通常记为 D；二是 D 上的关系，它反映了 D 中各数据元素之间的前后件关系，通常记为 R。一个数据结构可以表示成

$$B = (D,R)$$

其中 B 表示数据结构。为了反映 D 中各数据元素之间的前后件关系，一般用二元组来表示。例如，假设 a 与 b 是 D 中的两个数据，则二元组 (a,b) 表示 a 是 b 的前件，b 是 a 的后件。这样，在 D 中的每两个元素之间的关系都可以用这种二元组来表示。

例 2.3 一年四季的数据结构可以表示成

$$B = (D,R)$$
$$D = \{春,夏,秋,冬\}$$
$$R = \{(春,夏),(夏,秋),(秋,冬)\}$$

例 2.4 家庭成员数据结构可以表示成

$$B = (D,R)$$
$$D = \{父亲,儿子,女儿\}$$
$$R = \{(父亲,儿子),(父亲,女儿)\}$$

例 2.5 n 维向量

$$\boldsymbol{X} = (x_1,x_2,\cdots,x_n)$$

也是一种数据结构，即 $\boldsymbol{X}=(D,R)$，其中数据元素的集合为

$$D = \{x_1,x_2,\cdots,x_n\}$$

关系为

$$R = \{(x_1,x_2),(x_2,x_3),\cdots,(x_{n-1},x_n)\}$$

对于一些复杂的数据结构来说，它的数据元素可以是另一种数据结构。

例如，$m\times n$ 的矩阵

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

是一个数据结构。在这个数据结构中，矩阵的每一行

$$A_i = (a_{i1},a_{i2},\cdots,a_{in}),\quad i = 1,2,\cdots,m$$

可以看成是它的一个数据元素。这个数据结构的数据元素的集合为

$$D = \{A_1,A_2,\cdots,A_m\}$$

D 上的一个关系为

$$R = \{(A_1,A_2),(A_2,A_3),\cdots,(A_i,A_{i+1}),\cdots,(A_{m-1},A_m)\}$$

显然，数据结构 A 中的每一个数据元素 $A_i(i=1,2,\cdots,m)$ 又是另一个数据结构，即数据元素的集合为

$$D_i = \{a_{i1},a_{i2},\cdots,a_{in}\}$$

D_i上的一个关系为

$$R_i = \{(a_{i1}, a_{i2}), (a_{i2}, a_{i3}), \cdots, (a_{ij}, a_{i,j+1}), \cdots, (a_{i,n-1}, a_{in})\}$$

2. 数据的存储结构

数据处理是计算机应用的一个重要领域，在实际进行数据处理时，被处理的各数据元素总是被存放在计算机的存储空间中，并且，各数据元素在计算机存储空间中的位置关系与它们的逻辑关系不一定是相同的，而且一般也不可能相同。例如，在前面提到的一年 4 个季节的数据结构中，“春”是“夏”的前件，“夏”是“春”的后件，但在对它们进行处理时，在计算机存储空间中，“春”这个数据元素的信息不一定被存储在“夏”这个数据元素信息的前面，而可能在后面，也可能不是紧邻在前面，而是中间被其他信息所隔开。又如，在家庭成员的数据结构中，“儿子”和“女儿”都是“父亲”的后件，但在计算机存储空间中，根本不可能将“儿子”和“女儿”这两个数据元素的信息都紧邻存放在“父亲”这个数据元素信息的后面，即在存储空间中与“父亲”紧邻的只可能是其中的一个。由此可以看出，一个数据结构中的各数据元素在计算机存储空间中的位置关系与逻辑关系是有可能不同的。

数据的逻辑结构在计算机存储空间中的存放形式称为数据的存储结构（也称数据的物理结构）。

由于数据元素在计算机存储空间中的位置关系可能与逻辑关系不同，因此，为了表示存放在计算机存储空间中的各数据元素之间的逻辑关系（前后件关系），在数据的存储结构中，不仅要存放各数据元素的信息，还需要存放各数据元素之间的前后件关系的信息。

一般来说，一种数据的逻辑结构根据需要可以表示成多种存储结构，常用的存储结构有顺序、链接、索引等存储结构。采用不同的存储结构，其数据处理的效率是不同的。因此，在进行数据处理时，选择合适的存储结构是很重要的。

2.1.3 数据结构的图形表示

一个数据结构除了用二元关系表示外，还可以直观地用图形表示。在数据结构的图形表示中，对于数据集合 D 中的每一个数据元素用中间标有元素值的方框表示，一般称为数据结点，简称为结点；为了进一步表示各数据元素之间的前后件关系，对于关系 R 中的每一个二元组，用一条有向线段从前件结点指向后件结点。

例如，一年四季的数据结构可以用图 2.2 所示的图形来表示。

又如，反映家庭成员间辈分关系的数据结构可以用图 2.3 所示的图形表示。

图 2.2 一年四季数据结构的图形表示

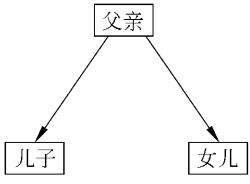

图 2.3 家庭成员间辈分关系数据结构的图形表示

显然，用图形方式表示一个数据结构是很方便的，并且也比较直观。有时在不会引起

误会的情况下，在前件结点到后件结点连线上的箭头可以省去。例如，在图 2.3 中，即使将“父亲”结点与“儿子”结点连线上的箭头以及“父亲”结点与“女儿”结点连线上的箭头都去掉，同样表示了“父亲”是“儿子”与“女儿”的前件，“儿子”与“女儿”均是“父亲”的后件，而不会引起误会。

例 2.6 用图形表示数据结构 $B=(D,R)$，其中

$$D=\{d_i \mid 1\leqslant i\leqslant 7\}=\{d_1,d_2,d_3,d_4,d_5,d_6,d_7\}$$

$$R=\{(d_1,d_3),(d_1,d_7),(d_2,d_4),(d_3,d_6),(d_4,d_5)\}$$

这个数据结构的图形表示如图 2.4 所示。

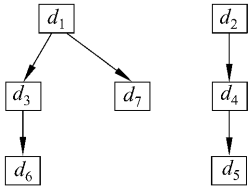

图 2.4 例 2.6 数据结构的图形表示

在数据结构中，没有前件的结点称为根结点；没有后件的结点称为终端结点（也称为叶子结点）。例如，在图 2.2 所示的数据结构中，元素“春”所在的结点（简称为结点“春”，下同）为根结点，结点“冬”为终端结点；在图 2.3 所示的数据结构中，结点“父亲”为根结点，结点“儿子”与“女儿”均为终端结点；在图 2.4 所示的数据结构中，有两个根结点 d_1 与 d_2，有 3 个终端结点 d_6、d_7、d_5。数据结构中除了根结点与终端结点外的其他结点一般称为内部结点。

通常，一个数据结构中的元素结点可能是在动态变化的。根据需要或在处理过程中，可以在一个数据结构中增加一个新结点（称为插入运算），也可以删除数据结构中的某个结点（称为删除运算）。插入与删除是对数据结构的两种基本运算。除此之外，对数据结构的运算还有查找、分类、合并、分解、复制和修改等。在对数据结构的处理过程中，不仅数据结构中的结点（数据元素）个数在动态地变化，而且，各数据元素之间的关系也有可能在动态地变化。例如，一个无序表可以通过排序处理而变成有序表；一个数据结构中的根结点被删除后，它的某一个后件可能就变成了根结点；在一个数据结构中的终端结点后插入一个新结点时，则原来的那个终端结点就不再是终端结点而成为内部结点了。有关数据结构的基本运算将在后面讲到具体数据结构时再介绍。

如果在一个数据结构中一个数据元素都没有，则称该数据结构为空的数据结构。在一个空的数据结构中插入一个新的元素后就变为非空；在只有一个数据元素的数据结构中，将该元素删除后就变为空的数据结构。

根据数据结构中各数据元素之间前后件关系的复杂程度，一般将数据结构分为两大类型：线性结构与非线性结构。

如果一个非空的数据结构满足下列两个条件：

（1）有且只有一个根结点。

（2）每一个结点最多有一个前件，也最多有一个后件。

则称该数据结构为线性结构，又称线性表。

由此可以看出，在线性结构中，各数据元素之间的前后件关系是很简单的。如例 2.3 中的一年四季这个数据结构，以及例 2.5 中的 n 维向量数据结构，它们都属于线性结构。

需要特别说明的是，在一个线性结构中插入或删除任何一个结点后还应是线性结构。