第 3 章　算法与数据结构

在计算机科学中，算法和数据结构是非常重要的概念。数据结构是计算机中存储、组织数据的方式。通常情况下，精心选择的数据结构可以带来最优效率的算法。简单地说，数据结构是研究数据及数据元素之间关系的一门学科，它包括 3 个方面的内容，即数据的逻辑结构、数据的存储结构和对数据的各种操作（也就是算法）。

3.1　算　　法

算法是程序的灵魂，对于一个需要实现特定功能的程序，实现它的算法可以有很多种，因此算法的优劣决定着程序的好坏。

3.1.1　算法的基本概念

对于算法的研究已经有数千年的历史了。计算机的出现，使得用机器自动解题的梦想成为现实，人们可以将算法编写成程序交给计算机执行，使许多原来认为不可能完成的算法变得实际可行。

算法是指对解题方案的准确而完整的描述，简单地说，就是解决问题的操作步骤。

值得注意的是，算法不等于数学上的计算方法，也不等于程序。在用计算机解决实际问题时，往往先设计算法，用某种表达方式（如流程图）描述，然后再用具体的程序设计语言描述此算法（即编程）。在编程时由于要受到计算机系统运行环境的限制，因此，程序的编制通常不可能优于算法的设计。

1．算法的基本特征

（1）可行性（Effectiveness）。算法在特定的执行环境中执行应当能够得出满意的结果，即必须有一个或多个输出。一个算法，即使在数学理论上是正确的，但如果在实际的计算工具上不能执行，则该算法也是不具有可行性的。

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 个量的和时：

$$A=10^{12}，B=1，C=-10^{12}$$

如果采用不同的运算顺序，就会得到不同的结果，例如：

$$A+B+C=10^{12}+1+（-10^{12}）=0$$

$$A+C+B=10^{12}+（-10^{12}）+1=1$$

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，必须考虑它的可行性，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。只要输入相同，初始状态相同，则无论执行多少遍，所得的结果都应该相同。如果算法的某个步骤有多义性，则该算法将无法执行。

例如，在进行汉字读音辨认时，汉字“解”在“解放”中读作 jiě，但它作为姓氏时却读作 xiè，这就是多义性。如果算法中存在多义性，计算机将无法正确地执行。

（3）有穷性（Finiteness）。算法中的操作步骤为有限个，且每个步骤都能在有限时间内完成。这包括合理的执行时间的含义。如果一个算法执行耗费的时间太长，即使最终得出了正确结果，也是没有意义的。

例如，数学中的无穷级数，当 n 趋向于无穷大时，求 $2n\times n!$，显然，这是无终止的计算，这样的算法是没有意义的。

（4）拥有足够的信息。一般来说，算法在拥有足够的输入信息和初始化信息时，才是有效的；当提供的信息不够时，算法可能无效。

例如，A=3，B=5，求 $A+B+C$ 的值，显然由于对 C 没有进行初始化，无法计算出正确的答案。所以，算法在拥有足够的输入信息和初始化信息时才是有效的。

在特殊情况下，算法也可以没有输入。因此，一个算法有 0 个或多个输入。

总之，算法是一个动态的概念，是指一组严谨地定义运算顺序或操作步骤的规则，并且每一个规则都是有效的、明确的，此顺序将在有限的次数下终止。

2．算法的基本要素

一个算法通常由两种基本要素组成：一种是对数据对象的运算和操作；另一种是算法的控制结构，即运算或操作间的顺序。

（1）算法中对数据对象的运算和操作。前面介绍了算法的一般定义和基本特征。实际上讨论的算法，主要是指计算机算法。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。算法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。不同计算机系统，其指令系统是有差异的。但一般的计算机系统中，都包括以下 4 类基本的运算和操作：

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。

计算机程序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，在设计算法之初，通常并不直接用计算机程序来描述，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四种基本操作考虑，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做进一步的描述，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地加工和变换，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、循环三种基本控制结构组合而成。

3．算法设计的基本方法

计算机解题的过程实际上是在实施某种算法，这种算法称为计算机算法。人们经过实践，总结和积累了许多行之有效的方法。常用的几种算法设计方法有列举法、归纳法、递推法、递归法、减半递推技术和回溯法。

（1）列举法。列举法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，尤其是要归纳出一个数学模型更为困难。从本质上讲，归纳就是通过观察一些简单而特殊的情况，最后总结出一般性的结论。

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）递推法。递推法是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确定。递推法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，必须注意数值计算的稳定性问题。

（4）递归法。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。

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；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。

递归是很重要的算法设计方法之一。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问题再归结为更简单的问题，这个过程可以一直进行下去，直到最简单的问题为止。

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。实际问题的复杂程度往往与问题的规模有着密切的联系。因此，利用分治法解决这类实际问题是有效的。所谓分治法，就是对问题分而治之。工程上常用的分治法是减半递推技术。

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 3-1】 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。

用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。

再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。

（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。

3.1.2 算法复杂度

一个算法的复杂度高低体现在运行该算法所需要的计算机资源的多少，所需的资源越多，就说明该算法的复杂度越高；反之，所需的资源越少，则该算法的复杂度越低。计算机的资源，最重要的是时间和空间（即存储器）资源，因此，算法复杂度包括算法的时间复杂度和算法的空间复杂度。

1．算法的时间复杂度

算法程序执行的具体时间和算法的时间复杂度并不是一致的。算法程序执行的具体时间受到所使用的计算机、程序设计语言以及算法实现过程中的许多细节影响。而算法的时间复杂度与这些因素无关。

算法的计算工作量是用算法所执行的基本运算次数来度量的，而算法所执行的基本运算次数是问题规模（通常用整数 n 表示）的函数，即算法的工作量=f（n），其中，n 为问题的规模。

所谓问题的规模就是问题的计算量的大小。如 1+2，这是规模比较小的问题，但 1+2+3+…+10000，这就是规模比较大的问题。

例如，在下列 3 个程序段中：

①{x++;s=0}

②for(i=1;i<=n;i++)

{x++;s+=x}/*一个简单的 for 循环，循环体内操作执行了 n 次*/

③for(i=1;i<=n;i++)

for(j=1;j<=n;j++)

{x++;s+=x;}/*嵌套的双层 for 循环，循环体内操作执行了 n^2 次*/

①中，基本运算“x++”只执行一次，重复执行次数为 1；

②中，由于有一个循环，所以基本运算“x++”执行了 n 次；

③中，嵌套双层循环，所以基本运算“x++”执行了 n^2 次。

则这 3 个程序段的时间复杂度分别为 O（1）、O（n）和 O（n^2）。

在具体分析一个算法的工作量时，在同一个问题规模下，算法所执行的基本运算次数还可能与特定的输入有关。即输入不同时，算法所执行的基本运算次数不同。例如，使用简单插入排序算法，对输入序列进行从小到大排序。输入序列为：

A：1　2　3　4　5

B：1　3　2　5　4

C：5　4　3　2　1

我们不难看出，序列 A 所需的计算工作量最少，因为它已经是非递减顺序排列；而序列 C 将耗费的基本运算次数最多，因为它完全是递减顺序排列的。

在这种情况下，可以用以下两种方法来分析算法的工作量：

- 平均性态；
- 最坏情况复杂性。

2．算法的空间复杂度

算法的空间复杂度是指执行这个算法所需要的内存空间。

算法执行期间所需的存储空间包括 3 个部分：

- 输入数据所占的存储空间；
- 程序本身所占的存储空间；
- 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。

如果额外空间量相对于问题规模（即输入数据所占的存储空间）来说是常数，即额外空间量不随问题规模的变化而变化，则称该算法是原地（in place）工作的。

为了降低算法的空间复杂度，主要应减少输入数据所占的存储空间以及额外空间，通常采用压缩存储技术。

3.2 数据结构的基本概念

3.2.1 数据结构的定义

数据是描述客观事物的信息符号的集合。在计算机发展的初期，由于计算机的主要功能是用于数值计算，数据就是指实数范围内的数值型数据；引入字符处理后，数据又扩展到字符型；多媒体时代，数据的类型进一步扩展，目前非数值型数据已占到数据的 90%以上。

数据类型是指具有相同特性的数据的集合。程序设计中的数据都必须归属于某个特定的数据类型。数据类型决定了数据的性质。常用的数据类型有整型、浮点型、字符型等。

对于复杂一些的数据，仅用数据类型还无法完整地描述，还需要用到数据元素的概念。数据元素是一个含义很广泛的概念。它是数据的“基本单位”，在计算机中通常作为一个整体进行考虑和处理。在数据处理领域，每一个需要处理的对象，甚至客观事物的一切个体，都可以抽象成数据元素，简称为元素。例如：

- 日常生活中一日三餐的名称——早餐、午餐、晚餐，可以作为一日三餐的数据元素；
- 在地理学中表示方向的名称——东、南、西、北，可以作为方向的数据元素；
- 在军队中表示军职的名称——连长、排长、班长、战士，可以作为军职的数据元素。

数据结构（Data Structure）的概念，在不同的书中有不同的提法。顾名思义，所谓数据结构，包含两个要素，即“数据”和“结构”。数据是指有限的需要处理的数据元素的集合，结构则是数据元素之间的关系的集合。存在着一定关系的数据元素的集合及定义在其上的操作（运算）被称为数据结构。例如：

东、南、西、北这 4 个数据元素都有一个共同的特征，它们都是地理方向名称，这 4 个数据元素构成了地理方向名称的集合；早餐、午餐、晚餐这 3 个数据元素也有一个共同的特征，即它们都是一日三餐的名称，从而构成了一日三餐名称的集合。

数据结构作为一门学科，主要研究三个方面的内容：数据的逻辑结构、数据的存储结构、对数据的各种操作（或算法）。研究的主要目的是为了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。

1．数据的逻辑结构

数据的逻辑结构就是数据元素之间的逻辑关系。在数据处理领域，通常把两两数据元素之间的关系用前、后件关系（或直接前驱与直接后继关系）来描述。实际上，数据元素之间的任何关系都可以用前、后件关系来描述。

例如，在考虑一日三餐的时间顺序关系时，“早餐”是“午餐”的前件（或直接前驱），而“午餐”是“早餐”的后件（或直接后继）。同样，在考虑军队中的上下级关系时，“连长”是“排长”的前件，“排长”是“连长”的后件；“排长”是“班长”的前件，“班长”是“排长”的后件；“班长”是“战士”的前件，“战士”是“班长”的后件。前、后件关系是数据元素之间最基本的关系。根据数据元素之间关系的不同特性，数据的逻辑结构可分为以下 4 大类：

（1）集合：数据元素之间的关系只有“是否属于同一个集合”。

（2）线性结构：数据元素之间存在线性关系，即最多只有一个前件和后件元素。

（3）树状结构：数据元素之间呈层次关系，即最多只有一个前件和多个后件元素。

（4）图状结构：数据元素之间的关系为多对多的关系。

数据的逻辑结构也可以用数学形式定义——数据结构是一个二元组：

$B=(D, R)$

其中，B 表示数据结构；D 是数据元素的集合；R 是 D 上关系的集合，它反映了 D 中各数据元素之间的前、后件关系，前、后件关系也可以用一个二元组来表示。

例如，如果把一日三餐看作一个数据结构，则可表示成：

$B=(D, R)$

D={早餐，午餐，晚餐}

R={（早餐，午餐），（午餐，晚餐）}

军队军职的数据结构可表示成：

$B=(D, R)$

D={连长，排长，班长，战士}

R={（连长，排长），（排长，班长），（班长，战士）}

2．数据的存储结构

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（也称为数据的物理结构）。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前、后件关系的信息。

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。例如，在前面提到的一日三餐的数据结构中，“早餐”是“午餐”的前件，“午餐”是“早餐”的后件，但在对它们进行处理时，在计算机存储空间中，“早餐”这个数据元素的信息不一定被存储在“午餐”这个数据元素信息的前面，可能在后面，也可能不是紧邻在前面，而是中间被其他的信息所隔开。

一般来说，数据在存储器中有顺序存储结构、链式存储结构、索引存储结构、散列存储结构 4 种基本存储方式。下面介绍两种最主要的数据存储结构方式。

（1）顺序存储结构。顺序存储结构是把逻辑上相邻的节点（也就是数据元素）存储在物理上相邻的存储单元中，节点之间的关系由存储单元的邻接关系来体现。例如，数据元素 a_1，a_2，a_3，…，a_n 的顺序存储结构如图 3-1 所示。

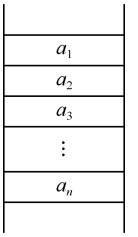


图 3-1　顺序存储结构

（2）链式存储结构。有时往往存在这样一些情况，存储器中没有足够大的连续可用空间，只有相邻的零碎小块存储单元；或者申请的内存空间不够，需要临时增加空间。这些情况下，

顺序存储就无法实现了。

链式存储结构可用一组任意的存储单元来存储数据元素，这组存储单元可以是连续的也可以是不连续的。链式存储结构因为有指针域，增加了额外的存储开销，并且在实现上也较为麻烦，但大大增加了数据结构的灵活性。

链式存储结构是将节点所占的存储单元分为两部分：一部分存放节点本身的信息，即数据域；另一部分存放该节点的后继节点所对应的存储单元的地址，即为指针域。例如，数据元素 a_1，a_2，a_3，…，a_n 的链式存储结构如图 3-2 所示。

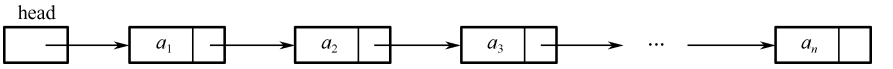


图 3-2　链式存储结构

3.2.2　数据结构的图形表示

数据元素之间最基本的关系是前、后件关系。前、后件关系，即每一个二元组，都可以用图形来表示。用中间标有元素值的方框表示数据元素，一般称之为数据节点，简称为节点。对于每一个二元组，用一条有向线段从前件指向后件。例如，一年四季的数据结构可以用如图 3-3（a）所示的图形来表示；家庭成员辈分关系数据结构可以用如图 3-3（b）所示的图形来表示。

（a）一年四季数据结构的图形表示　　（b）家庭成员辈分关系数据结构的图形表示

图 3-3　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，在前件节点到后件节点连线上的箭头可以省去。例如，在图 3-3（b）中，即使将“父亲”节点与“儿子”节点连线上的箭头以及“父亲”节点与“女儿”节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 3-2】　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 3-4 所示。

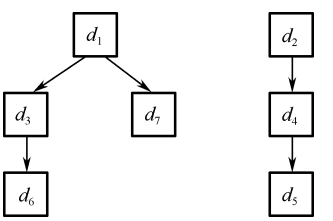


图 3-4　例 3-2 数据结构的图形表示

在数据结构中，没有前件的节点称为根节点；没有后件的节点称为终端节点（也称为叶子节点）。例如，在如图 3-3（a）所示的数据结构中，元素“春”所在的节点（简称为节点“春”，下同）为根节点，节点“冬”为终端节点；在如图 3-3（b）所示的数据结构中，节点“父亲”为根节点，节点“儿子”与“女儿”均为终端节点；在如图 3-4 所示的数据结构中，有 2 个根节点 d_1 与 d_2，有 3 个终端节点 d_6、d_7、d_5。数据结构中除了根节点与终端节点外的其他节点一般称为内部节点。

通常，一个数据结构中的元素节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节点（称为插入运算），也可以删除数据结构中的某个节点（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，不仅数据结构中的节点（即数据元素）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节点被删除后，它的某一个后件可能就变成了根节点；在一个数据结构中的终端节点后插入一个新节点后，则原来的那个终端节点就不再是终端节点而成为内部节点了。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。

3.2.3 线性结构与非线性结构

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）有且只有一个根节点；

（2）每一个节点最多有一个前件，也最多有一个后件。

则称该数据结构为线性结构。线性结构又称线性表。

不满足以上两个条件的数据结构就称为非线性结构。如图 3-3（b）所示的家庭成员辈分关系数据结构，以及如图 3-4 所示的数据结构，它们都不是线性结构，而属于非线性结构。非线性结构主要有树形结构和网状结构。显然，在非线性结构中，各数据元素之间的前、后件关系要比线性结构复杂，因此，对非线性结构的存储与处理比线性结构要复杂得多。

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。一个空的数据结构究竟是属于线性结构还是属于非线性结构，这要根据具体情况来确定。如果对该数据结构的运算是按线性结构的规则来处理的，则属于线性结构，否则属于非线性结构。

3.3 线性表及其顺序存储结构

3.3.1 线性表的基本概念

线性表是最简单最常用的一种数据结构。

1．线性表的定义

线性表是由 n（$n \geqslant 0$）个数据元素 a_1，…，a_n 组成的一个有限序列，表中的每一个数据元素，除了第一个外，有且只有一个前件；除了最后一个外，有且只有一个后件。即线性表或是一个空表，或可以表示为

$$（a_1，a_2，\cdots，a_i，\cdots，a_n）$$

其中，a_i（i=1，2，…，n）是属于数据对象的元素，通常也称其为线性表中的一个节点。

显然，线性表是一种线性结构。数据元素在线性表中的位置只取决于它们自己的序号，即数据元素之间的相对位置是线性的。

例如：

（1）英文字母表（A，B，C，…，Z）是一个长度为 26 的线性表，其中每个字母字符就是一个数据元素。

（2）地理学中的四向（东，南，西，北）是一个长度为 4 的线性表，其中的每一个方向名是一个数据元素。

（3）矩阵也是一个线性表，只不过它是一个比较复杂的线性表。在矩阵中，既可以把每一行看成一个数据元素（即每一行向量为一个数据元素），也可以把每一列看成一个数据元素（即每一列向量为一个数据元素）。其中，每一个数据元素（一个行向量或者列向量）实际上又是一个简单的线性表。

在复杂的线性表中，一个数据元素由若干数据项组成，此时，把数据元素称为记录（record），而由多个记录构成的线性表又称为文件（file）。例如，一个按照姓名的拼音字母为序排列的通信录就是一个复杂的线性表，如表 3-1 所示。表中，每个联系人的情况为一个记录，它由姓名、性别、电话号码、电子邮件和居住地址 5 个数据项组成。

表 3-1 复杂线性表

姓名	性别	电话号码	电子邮件	居住住址
张大千	男	186****2569	zdq@163.com	上海市
李永南	男	159****7463	lyn@265.com	沈阳市
白倩茹	女	130****8195	bqr@qq.com	北京市
…	…	…	…	…

2．非空线性表的特征

非空线性表有如下一些结构特征：

（1）有且只有一个根节点 a_1，它无前件。

（2）有且只有一个终端节点 a_n，它无后件。

（3）除根节点与终端节点外，其他所有节点有且只有一个前件，也有且只有一个后件。线性表中节点的个数 n 称为线性表的长度，当 n=0 时，称为空表。

3.3.2 线性表的顺序存储结构

将一个线性表存储到计算机中，可以采用许多不同的方法，其中既简单又自然的是顺序存储方法，即把线性表的节点按逻辑顺序依次存放在一组地址连续的存储单元中，用这种方

法存储的线性表简称为顺序表。

线性表的顺序存储结构具有以下两个基本特点：

（1）线性表中所有元素所占的存储空间是连续的；

（2）线性表中各数据元素在存储空间中是按逻辑顺序依次存放的。

由此可以看出，在线性表的顺序存储结构中，其前、后件两个元素在存储空间中是紧邻的，且前件元素一定存储在后件元素的前面。

在线性表的顺序存储结构中，如果线性表中各数据元素所占的存储空间（字节数）相等，则要在该线性表中查找某个元素是很方便的。

例如，长度为 n 的线性表（a_1，a_2，…，a_i，…，a_n）的顺序存储结构如图 3-5 所示，在顺序表中，第一个数据元素的存储地址（指第一个字节的地址，即首地址）为 ADR（a_1），每一个数据元素占 k 个字节，则线性表中第 i 个元素 a_i 在计算机存储空间中的存储地址为：

ADR（a_i）=ADR（a_1）+（i−1）k

存储地址	数据元素在线性表中的序号	内存状态	空间分配
ADR(a_1)	1	a_1	占k个字节
ADR(a_1)+k	2	a_2	占k个字节
…	…	…	…
ADR(a_1)+(i−1)k	i	a_i	占k个字节
…	…	…	…
ADR(a_1)+(n−1)k	n	a_n	占k个字节
…	…	…	…

图 3-5　线性表的顺序存储结构示意图

例如，在顺序表中存储数据（14，28，56，76，48，32，64），每个数据元素占 2 个存储单元，第 1 个数据元素 14 的存储地址是 300，则第 5 个数据元素 48 的存储地址是：

ADR（a_5）=ADR（a_1）+（5−1）×2=300+8=308

从这种表示方法可以看到，它是用元素在计算机内物理位置上的相邻关系来表示元素之间逻辑上的相邻关系的。只要确定了首地址，线性表内任意元素的地址都可以方便地计算出来。

在程序设计语言中，通常定义一个一维数组来表示线性表的顺序存储空间。因为程序设计语言中的一维数组与计算机中实际的存储空间结构是类似的，这就便于用程序设计语言对线性表进行各种运算处理。

3.3.3　线性表的插入运算

线性表的插入运算是指在表的第 i（1≤i≤n+1）个位置上，插入一个新节点 x，使长度为 n 的线性表变成长度为 n+1 的线性表。

在第 i 个元素之前插入一个新元素，完成插入操作主要有以下 3 个步骤：

（1）把原来第 n 个节点至第 i 个节点依次往后移一个元素位置。

（2）把新节点放在第 i 个位置上。

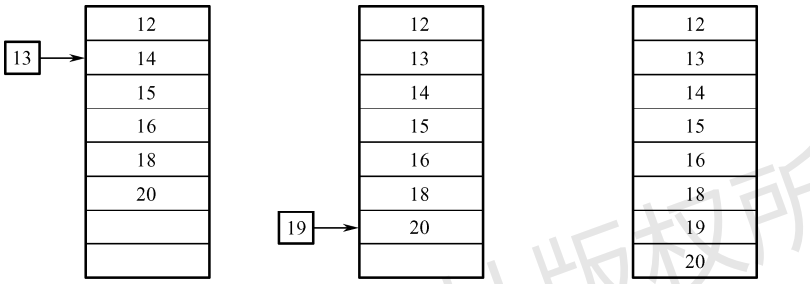
（3）修正线性表的节点个数。

例如，如图 3-6（a）所示表示一个存储空间为 8、长度为 6 的线性表。为了在线性表的

第 2 个元素（即 14）之前插入一个值为 13 的数据元素，则需将第 2～6 个数据元素，共 $n-i+1=6-2+1=5$ 个数据元素，依次往后移动一个位置，空出第 2 个元素的位置，然后将新元素 13 插入第 2 个位置。插入一个新元素后，线性表的长度增加 1，变为 7，如图 3-6（b）所示。

一般情况下，在第 i（$1\leqslant i\leqslant n$）个元素之前插入一个元素时，需将第 i 个元素之后（包括第 i 个元素）的所有元素向后移动一个位置。

再例如，在如图 3-6（b）所示的线性表的第 7 个元素（即 20）之前，再插入一个值为 19 的新元素，采用同样的步骤：将第 7 个元素之后的元素（包括第 7 个元素），共 $n-i+1=7-7+1=1$ 个元素，向后移动一个位置，然后将新元素 19 插入第 7 个位置。插入后，线性表的长度增加 1，变成 8，如图 3-6（c）所示。



（a）长度为8的线性表　（b）插入元素13后的线性表　（c）插入元素19后的线性表

图 3-6　线性表在顺序存储结构下的插入操作

一般会为线性表开辟一个大于线性表长度的存储空间，如图 3-6（a）所示，线性表长度为 6，存储空间为 8。经过线性表的多次插入运算，可能出现存储空间已满仍继续插入的错误运算，这类错误称之为“上溢”。

显然，如果插入运算在线性表的末尾进行，即在第 n 个元素之后插入新元素，则只要在表的末尾增加一个元素即可，不需要移动线性表中的元素。

如果要在第 1 个位置处插入一个新元素，则需要移动表中所有的元素。

在一般情况下，如果在第 i（$1\leqslant i\leqslant n$）个元素之前插入一个新元素，则原来第 i 个元素之后（包括第 i 个元素）的所有元素都必须移动。

线性表的插入运算，其时间主要花费在节点的移动上，所需移动节点的次数不仅与表的长度有关，而且与插入的位置有关。

3.3.4　线性表的删除运算

线性表的删除运算是指将表的第 i（$1\leqslant i\leqslant n$）个节点删除，使长度为 n 的线性表变成长度为 $n-1$ 的线性表。

删除时应将第 $i+1$ 个元素至第 n 个元素依次向前移动一个元素的位置，共移动了 $n-i$ 个元素。完成删除操作主要有以下 2 个步骤：

（1）把第 i 个元素之后（不包含第 i 个元素）的 $n-i$ 个元素依次前移一个位置。

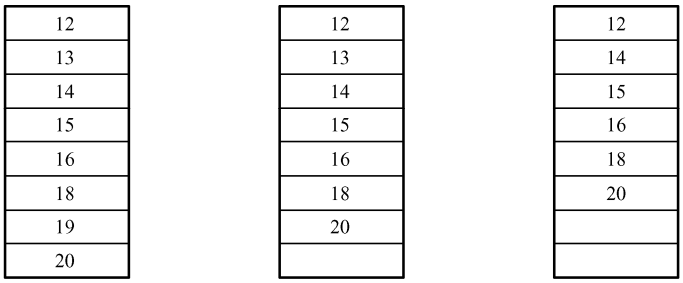
（2）修正线性表的节点个数。

例如，如图 3-7（a）所示为一个长度为 8 的线性表，将第 7 个元素 19 删除的过程如下：

把第 8 个元素 20 往前移动一个位置，此时，线性表的长度减少了 1，变成了 7，如图 3-7（b）所示。

一般情况下，要删除第 i（$1 \leqslant i \leqslant n$）个元素时，则要从第 i+1 个元素开始直到第 n 个元素之间共 $n-i$ 个元素依次向前移动一个位置。删除结束后，线性表的长度减少 1。

倘若再要删除如图 3-7（b）所示线性表的第 2 个元素 13，则采用同样的步骤：从第 3 个元素 14 开始至最后一个元素 20，共 $n-i=7-2=5$ 个元素依次往前移动一个位置。此时，线性表的长度减少了 1，变成 6，如图 3-7（c）所示。



（a）长度为8的线性表　（b）删除元素19后的线性表　（b）删除元素13后的线性表

图 3-7　线性表在顺序存储结构下的删除操作

显然，如果删除运算在线性表的末尾进行，即删除第 n 个元素，则不需要移动线性表中的元素。如果要删除第 1 个元素，则需要移动表中剩余的所有元素。

在一般情况下，如果删除第 i（$1 \leqslant i \leqslant n$）个元素，则原来第 i 个元素之后的所有元素都必须依次往前移动一个位置。

综上所述，线性表的顺序存储结构是用物理位置上的邻接关系来表示节点间的逻辑关系的，因此无须为表示节点间的逻辑关系而增加额外的存储空间，并且可以方便地随机存取表中任意节点，适合用于小线性表，或者建立之后其中元素不常变动的线性表。但插入或删除运算不方便，不适合用于需要经常进行插入和删除运算的线性表以及长度较大的线性表。

3.4　栈 和 队 列

栈和队列都是特殊的线性表，它们都有自己的特点。栈是“先进后出”的线性表；而队列是“先进先出”的线性表。本节将详细讲解栈和队列的基本运算以及它们的不同点。

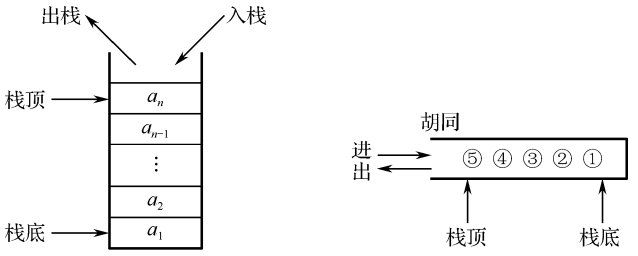
3.4.1　栈及其基本运算

1．栈的定义

栈（Stack）是一种特殊的线性表，这种线性表上的插入和删除运算限定在表的某一端进行。允许进行插入和删除的这一端称为栈顶，另一端则称为栈底。处于栈顶位置的数据元素称为栈顶元素，处于栈底位置的数据元素称为栈底元素。在如图 3-8（a）所示的顺序栈中，元素是以 a_1，a_2，…，a_n 的顺序进栈，因此栈底元素是 a_1，栈顶元素是 a_n。不含任何数据元素的栈称为空栈。

下面举例说明栈结构的特征。

假设有一个很窄的死胡同，胡同里能容纳若干人，但每次只能允许一个人进出。现有 5 个人，分别编号为①～⑤，按编号的顺序进入胡同，如图 3-8（b）所示。此时，若④要出来，必须等⑤退出后才有可能；若①要退出，则必须等到⑤④③②依次都退出后才行。



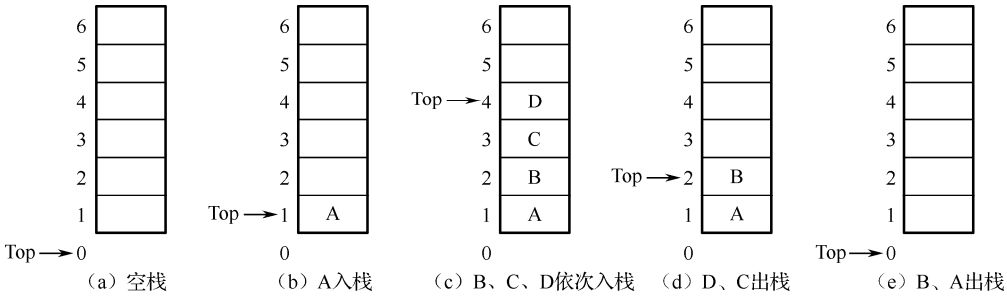
（a）顺序栈的结构　　（b）胡同进出示例

图 3-8　栈的结构示意图

栈可以比作这里的死胡同，栈顶相当于胡同口，栈底相当于胡同的另一端，进、出胡同可看作栈的插入、删除运算。插入、删除都在栈顶进行，这表明栈是“后进先出”（Last In First Out，LIFO）或“先进后出”（First In Last Out，FILO）的，因此，栈也称为“后进先出”线性表或“先进后出”线性表。

2．栈的顺序存储及其运算

通常，栈有两种实现方法：顺序实现和链接实现。栈的顺序存储结构称为顺序栈，顺序栈通常由一个一维数组和一个记录栈顶位置的变量组成。因为栈的操作仅在栈顶进行，栈底位置固定不变，所以可将栈底位置设置在数组两端的任意一个端点上，习惯上将栈底放在数组下标小的一端；另外需使用一个变量 Top 记录当前栈顶下标值，即表示当前栈顶位置，通常称 Top 为栈顶指针。如图 3-9 所示说明了在顺序栈中进行入栈和出栈运算时栈中元素和栈顶指针的变化。



（a）空栈　（b）A入栈　（c）B、C、D依次入栈　（d）D、C出栈　（e）B、A出栈

图 3-9　顺序栈入栈、出栈示意图

栈的基本运算有 3 种：入栈、出栈与读栈顶元素。

（1）入栈运算。入栈运算是指在栈顶位置插入一个新元素。这个运算有两个基本操作：首先将栈顶指针进 1（即 Top 加 1），然后将新元素插入栈顶指针指向的位置。

当栈顶指针已经指向存储空间的最后一个位置时，说明栈空间已满，不能再进行入栈操作。若此时继续入栈操作，这种情况称为栈“上溢”错误。

（2）出栈运算。出栈运算是指取出栈顶元素并赋给一个指定的变量，这个运算有两个基

本操作：首先将栈顶元素（栈顶指针指向的元素）赋给一个指定的变量，然后将栈顶指针退1（即 Top 减 1）。

当栈顶指针为 0 时，说明栈空，不能进行出栈操作。若此时继续出栈操作，这种情况称为栈“下溢”错误。

（3）读栈顶元素。读栈顶元素是指将栈顶元素赋给一个指定的变量。需要注意的是，这个运算不删除栈顶元素，只是将它的值赋给一个变量。因此，在这个运算中栈顶指针不会发生改变。

当栈顶指针为 0 时，说明栈空，读不到栈顶元素。

3.4.2 队列及其基本运算

队列也是一种运算受限的线性表，广泛应用于各种程序设计中。

1．队列的定义

队列是一种运算受限的线性表，在这种线性表中，插入限定在表的某一端进行，删除限定在表的另一端进行。允许插入的一端称为队尾，允许删除的一端称为队头。新插入的节点只能添加到队尾，被删除的只能是排在队头的节点。习惯上把往队列的队尾插入一个元素称为入队运算，从队列的队头删除一个元素称为出队运算。若有队列：

$$Q=(q_1, q_2, \cdots, q_n)$$

那么，q_1 为队头元素，q_n 为队尾元素。队列中的元素是按照 q_1，q_2，…，q_n 的顺序进入的，退出队列也只能按照这个次序依次退出。也就是说，只有在 q_1，q_2，…，q_{n-1} 都出队之后，q_n 才能退出队列。因最先进入队列的元素将最先出队，所以队列具有“先进先出”的特性。

队头元素 q_1 是最先被插入的元素，也是最先被删除的元素。队尾元素 q_n 是最后被插入的元素，也是最后被删除的元素。因此，与栈相反，队列又称为“先进先出”（First In First Out，FIFO）或“后进后出”（Last In Last Out，LILO）的线性表。

例如，火车进隧道，最先进隧道的是火车头，最后进的是火车尾，而火车出隧道的时候也是火车头先出，最后出的是火车尾。

2．队列的运算

可以用顺序存储的线性表来表示队列。为了指示当前执行出队运算的队头位置，需要一个队头指针 front；为了指示当前执行入队运算的队尾位置，需要一个队尾指针 rear。队头指针 front 总是指向队头元素的前一个位置，而队尾指针 rear 总是指向队尾元素。如图 3-10 所示是队列示意图。

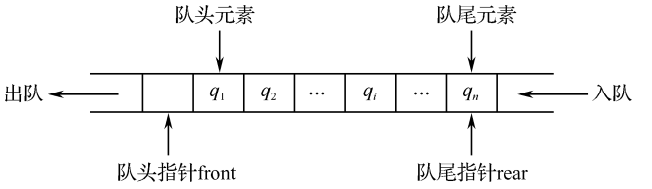


图 3-10　队列示意图

往队列的队尾插入一个元素称为入队运算，从队列的队头删除一个元素称为出队运算。

例如，如图 3-11 所示是在队列中进行插入与删除的示意图。一个大小为 10 的数组，用于表示队列，初始时队列为空，如图 3-11（a）所示；插入数据 a 后，如图 3-11（b）所示；插入数据 b 后，如图 3-11（c）所示；删除数据 a 后，如图 3-11（d）所示。

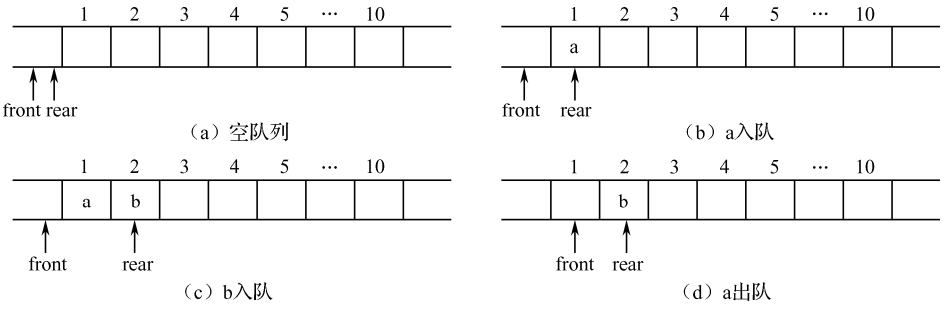


图 3-11　队列的状态示意图

3．循环队列及其运算

循环队列是队列的一种顺序存储结构，用队尾指针 rear 指向队列中的队尾元素，用队头指针指向队头元素的前一个位置，因此，从队头指针 front 指向的后一个位置直到队尾指针 rear 指向的位置之间所有的元素均为队列中的元素。

一维数组（1：*m*）最大存储空间为 *m*，数组（1：*m*）作为循环队列的存储空间时，循环队列的初始状态为空，即 front=rear=*m*。如图 3-12 所示是循环队列的初始状态示意图。

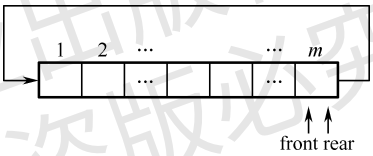


图 3-12　循环队列的初始状态示意图

循环队列的基本运算主要有两种：入队运算与出队运算。

（1）入队运算。入队运算是指在循环队列的队尾加入一个新元素。入队运算可分为两个步骤：首先队尾指针进 1（即 rear+1），然后在 rear 指针指向的位置插入新元素。特别地，当队尾指针 rear=*m*+1 时（即 rear 原值为 *m*，再进 1），置 rear=1，这表示在最后一个位置插入元素后，紧接着在第一个位置插入新元素。

（2）出队运算。出队运算是指在循环队列的队头位置退出一个元素，并赋给指定的变量。出队运算也可分为两个步骤：首先，队头指针进 1（即 front+1），然后删除 front 指针指向的位置上的元素。特别地，当队头指针 front=*m*+1 时（即 front 原值为 *m*，再进 1），置 front=1，这表示在最后一个位置删除元素后，紧接着在第一个位置删除元素。

可以看出，循环队列在队列满时和队列空时都有 front=rear。在实际应用中，通常增加一个标志量 *S*，来区分循环队列是空还是满。*S* 值的定义如下：

当 *S* = 0 时，循环队列为空，此时不能再进行出队运算，否则会发生“下溢”错误。

当 *S* =1 时，循环队列满，此时不能再进行入队运算，否则会发生“上溢”错误。

在定义了 *S* 以后，循环队列初始状态为空，表示为：*S*=0，且 front=rear。

3.5 线 性 链 表

3.5.1 线性链表的基本概念

前面主要讨论了线性表的顺序存储结构以及在顺序存储结构下的运算。线性表的顺序存储结构具有简单、运算方便等优点，特别是对于小线性表或长度固定的线性表，采用顺序存储结构的优越性更为突出。

但是，线性表的顺序存储结构在某些情况下就显得不那么方便，运算效率不那么高。实际上，线性表的顺序存储结构存在以下几个方面的缺点：

（1）在一般情况下，要在顺序存储的线性表中插入一个新元素或删除一个元素时，为了保证插入或删除后的线性表仍然为顺序存储，则在插入或删除过程中需要移动大量的数据元素。在平均情况下，为了在顺序存储的线性表中插入或删除一个元素，需要移动线性表中约一半的元素；在最坏情况下，则需要移动线性表中所有的元素。因此，对于大的线性表，特别是元素的插入或删除很频繁的情况下，采用顺序存储结构是很不方便的，插入与删除运算的效率都很低。

（2）当为一个线性表分配顺序存储空间后，如果出现线性表的存储空间已满，但还需要插入新的元素时，就会发生“上溢”错误。在这种情况下，如果在原线性表的存储空间后找不到与之连续的可用空间，则会导致运算的失败或中断。显然，这种情况的出现对运算是很不利的。也就是说，在顺序存储结构下，线性表的存储空间不便于扩充。

（3）在实际应用中，往往是同时有多个线性表共享计算机的存储空间。例如，在一个处理中，可能要用到若干个线性表（包括栈与队列）。在这种情况下，存储空间的分配将是一个难题。如果将存储空间平均分配给各线性表，则有可能造成有的线性表的空间不够用，而有的线性表的空间根本用不着或用不满。这种情况实际上使计算机的存储空间得不到充分利用。如果多个线性表共享存储空间，对每一个线性表的存储空间进行动态分配，则为了保证每一个线性表的存储空间连续且顺序分配，会导致在对某个线性表进行动态分配存储空间时，必须移动其他线性表中的数据元素。这就是说，线性表的顺序存储结构不便于对存储空间的动态分配。

由于线性表的顺序存储结构存在以上这些缺点，因此，对于大的线性表，特别是元素变动频繁的大线性表不宜采用顺序存储结构，而是采用下面要介绍的链式存储结构。

假设数据结构中的每一个数据节点对应一个存储单元，这种存储单元称为存储节点，简称节点。

在链式存储方式中，要求每个节点由两部分组成：一部分用于存放数据元素值，称为数据域；另一部分用于存放指针，称为指针域。其中，指针用于指向该节点的前一个或后一个节点（即前件或后件）。

在链式存储结构中，存储数据结构的存储空间可以不连续，各数据节点的存储顺序与数据元素之间的逻辑关系可以不一致，而数据元素之间的逻辑关系是由指针域来确定的。

链式存储方式既可用于表示线性结构，也可用于表示非线性结构。在用链式结构表示较

复杂的非线性结构时，其指针域的个数要多一些。

1．线性链表

线性表链式存储结构的特点是，用一组不连续的存储单元存储线性表中的各个元素。因为存储单元不连续，数据元素之间的逻辑关系就不能依靠数据元素存储单元之间的物理关系来表示。为了表示每个元素与其后继元素之间的逻辑关系，每个元素除了需要存储自身的信息外，还要存储一个指示其后件的信息（即后件元素的存储位置）。

线性表链式存储结构的基本单位称为存储节点，如图 3-13 所示是存储节点示意图。每个存储节点包括两个组成部分。

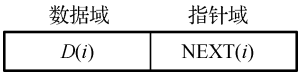


图 3-13　线性链表的一个存储节点

数据域：存放数据元素本身的信息。

指针域：存放一个指向后件节点的指针，即存放下一个数据元素的存储地址。

假设一个线性表有 n 个元素，则这 n 个元素所对应的 n 个节点就通过指针链接成一个线性链表。

所谓线性链表，是指线性表的链式存储结构，简称链表。由于这种链表中每个节点只有一个指针域，故又称为单链表。

在线性链表中，第一个元素没有前件，指向链表中的第一个节点的指针是一个特殊的指针，称为这个链表的头指针（HEAD）。最后一个元素没有后件，因此，线性链表最后一个节点的指针域为空，用 NULL 或 0 表示。

例如，如图 3-14 所示为线性表（A，B，C，D，E，F）的线性链表存储结构。头指针 HEAD 中存放的是第一个元素 A 的存储地址（即存储序号）。

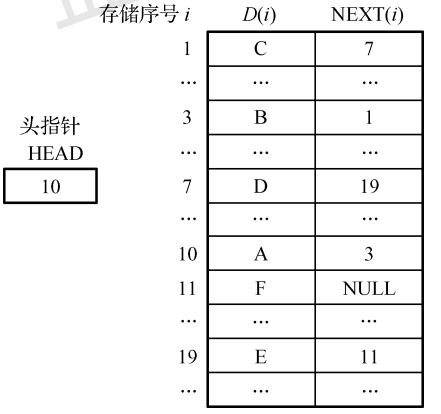


图 3-14　线性链表示例

图 3-14 中“…”的存储单元可能存有数据，也可能是空闲的。总之，线性链表的存储单元是任意的，即各数据节点的存储序号可以是连续的，也可以是不连续的。各节点在存储空间中的位置关系与逻辑关系不一致，前、后件关系由存储节点的指针来表示。指向第一个数据元素的头指针 HEAD 等于 NULL 或者 0 时，称为空表。

因为在讨论线性链表时，主要关心的只是线性表中元素的逻辑顺序，而不是每个元素在

存储器中的实际物理位置，所以，可以把如图 3-14 所示的线性链表更加直观地表示成用箭头相链接的节点序列，如图 3-15 所示。其中，每一个节点上面的数字表示该节点的存储序号（即节点号）。

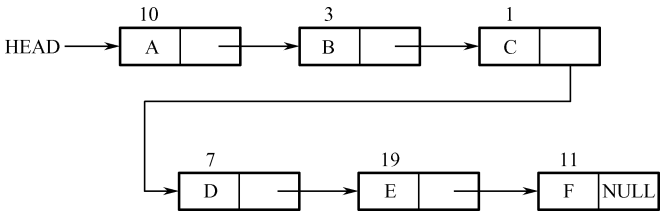


图 3-15　线性链表逻辑状态

前面提到，在这样的线性链表中，每个存储节点只有一个指针域，称为单链表。在实际应用中，有时还会用到每个存储节点有两个指针域的链表，一个指针域存放前件的地址，称为左指针（Llink）；另一个指针域存放后件的地址，称为右指针（Rlink）。这样的线性链表称为双向链表。如图 3-16 所示是双向链表的示意图。双向链表的第一个元素的左指针（Llink）为空，最后一个元素的右指针（Rlink）为空。

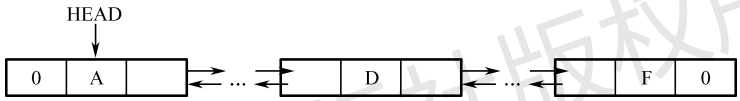


图 3-16　双向链表示意图

在单链表中，只能顺指针向链尾方向进行扫描，由某一个节点出发，只能找到它的后件。若要找出它的前件，必须从头指针开始重新寻找。

而在双向链表中，由于为每个节点设置了两个指针，从某一个节点出发，可以很方便地找到其他任意一个节点。

2．带链的栈

栈也是线性表，也可以采用链式存储结构表示，把栈组织成一个单链表。这种数据结构可称为带链的栈。

在实际应用中，带链的栈可以用来收集计算机存储空间中所有空闲的存储节点，这种带链的栈称为可利用栈。由于可利用栈链接了计算机存储空间中所有的空闲节点，因此，当计算机系统或用户程序需要存储节点时，就可以从中取出栈顶节点；当计算机系统或用户程序释放一个存储节点（该元素从表中删除）时，则要将该节点放回到可利用栈的栈顶。由此可见，计算机中的所有可利用空间都可以以节点为单位链接在可利用栈中。随着其他线性链表中节点的插入与删除，可利用栈处于动态变化之中，即可利用栈要经常进行出栈与入栈操作。

3．带链的队列

与栈类似，队列也可以采用链式存储结构表示。带链的队列就是用一个单链表来表示队列，队列中的每一个元素对应链表中的一个节点。

4．顺序表和链表的比较

线性表的顺序存储方式称为顺序表，其特点是用物理存储位置上的邻接关系来表示节点间的逻辑关系。

线性表的链接存储称为线性链表，简称链表，其特点是每个存储节点都包括数据域和指针域，用指针表示节点间的逻辑关系。

顺序表和链表的优、缺点比较如表 3-2 所示。

表 3-2 顺序表和链表的优、缺点比较

类 型	优 点	缺 点
顺序表	（1）可以随机存取表中的任意节点； （2）无须为表示节点间的逻辑关系而额外增加存储空间	（1）插入和删除运算效率很低； （2）存储空间不便于扩充； （3）不便于对存储空间的动态分配
链表	（1）在进行插入和删除运算时，只需要改变指针即可，不需要移动元素； （2）链表的存储空间易于扩充并且方便空间的动态分配	需要额外的空间（指针域）来表示数据元素之间的逻辑关系，存储密度比顺序表低

3.5.2 线性链表的基本运算

对线性链表进行的运算主要包括查找、插入、删除、合并、分解、逆转、复制和排序。本小节主要讨论线性链表的查找、插入和删除运算。

1．在线性链表中查找指定元素

查找指定元素所处的位置是插入和删除等操作的前提，只有先通过查找定位才能进行元素的插入和删除等进一步的运算。

在链表中查找指定元素必须从队头指针出发，沿着指针域 Next 逐个节点搜索，直到找到指定元素或链表尾部为止，而不能像顺序表那样，只要知道了首地址就可以计算出任意元素的存储地址，如图 3-5 所示。因此，线性链表不是随机存储结构。

在链表中，如果有指定元素，则扫描到等于该元素值的节点时停止扫描，返回该节点的位置。因此，如果链表中有多个等于指定元素值的节点，只返回第一个节点的位置；如果链表中没有元素的值等于指定元素，则扫描完所有元素后返回 NULL。

2．可利用栈的插入和删除

线性链表的存储单元是不连续的，如图 3-14 所示，这样就存在一些离散的空闲节点。如前所述，为了把计算机存储空间中空闲的存储节点利用起来，把所有空闲的节点组织成一个带链的栈，称为可利用栈。

线性链表执行删除运算时，被删除的节点可以“回收”到可利用栈，对应于可利用栈的入栈运算；线性链表执行插入运算时，需要一个新的节点，可以在可利用栈中取栈顶节点，对应于可利用栈的出栈运算。可利用栈的入栈运算和出栈运算只需要改动 Top 指针即可。

3．线性链表的插入

线性链表的插入是指在链式存储结构下的线性表中插入一个新元素。

首先，要给该元素分配一个新节点，新节点可以从可利用栈中取得。然后将存放新元素值的节点链接到线性链表中指定的位置。

要在线性链表中数据域为 M 的节点之前插入一个新元素 N，则插入过程如下所述：

（1）取可利用栈的栈顶空闲节点，生成一个数据域为 N 的节点，将新节点的存储序号存

放在指针变量 p 中。

（2）在线性链表中查找数据域为 M 的节点，将其前件的存储序号存放在变量 q 中。

（3）将新节点 p 的指针域内容设置为指向数据域为 M 的节点。

（4）将节点 q 的指针域内容改为指向新节点 p。

插入过程如图 3-17 所示。

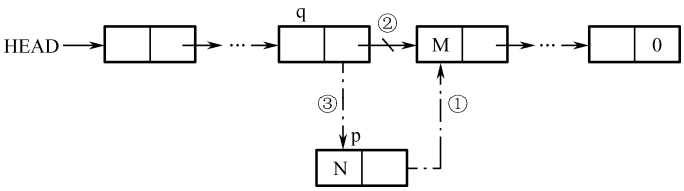


图 3-17　线性链表的插入运算

由于线性链表执行插入运算时，新节点的存储单元取自可利用栈，因此，只要可利用栈非空，线性链表总能找到存储插入元素的新节点，因而无须规定最大存储空间，也不会发生“上溢”的错误。此外，线性链表在执行插入运算时，不需要移动数据元素，只需要改动有关节点的指针域即可，插入运算效率大大提高。

4．线性链表的删除

线性链表的删除是指在链式存储结构下的线性表中删除包含指定元素的节点。

在线性链表中删除数据域为 M 的节点，其过程如下所述：

（1）在线性链表中查找包含元素 M 的节点，将该节点的存储序号存放在 p 中。

（2）把 p 节点的前件存储序号存放在变量 q 中，将 q 节点的指针修改为指向 p 节点的指针所指向的节点（即 p 节点的后件）。

（3）把数据域为 M 的节点“回收”到可利用栈。

删除过程如图 3-18 所示。

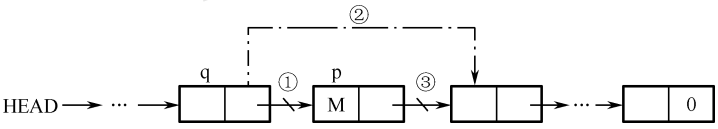


图 3-18　线性链表的删除运算

和插入运算一样，线性链表的删除运算也不需要移动元素，删除运算只需改变被删除元素前件的指针域即可。而且，删除的节点回收到可利用栈中，可供线性链表插入运算时使用。

3.5.3　循环链表及其基本运算

1．循环链表的定义

在单链表的第一个节点前增加一个表头节点，队头指针指向表头节点，最后一个节点的指针域的值由 NULL 改为指向表头节点，这样的链表称为循环链表。在循环链表中，所有节点的指针构成了一个环状链。

2．循环链表与单链表的比较

对单链表的访问是一种顺序访问，从其中某一个节点出发，只能找到它的直接后继（即后件），但无法找到它的直接前驱（即前件）。而且，对于空表和第一个节点的处理必须单独考虑，空表与非空表的操作不统一。

在循环链表中，只要指出表中任何一个节点的位置，就可以从它出发访问到表中其他所有的节点。并且，由于表头节点是循环链表所固有的节点，因此，即使在表中没有数据元素的情况下，表中也至少有一个节点存在，从而使空表和非空表的运算统一。

3.6 树与二叉树

3.6.1 树的基本概念

树（Tree）是一种简单的非线性结构，直观地看，树是以分支关系定义的层次结构。由于它呈现与自然界的树类似的结构形式，所以称它为树。树结构在客观世界中是大量存在的。

例如，一个家族中的族谱关系：A 有后代 B 和 C，B 有后代 D、E 和 F，C 有后代 G，E 有后代 H 和 I，则这个家族的成员及血统关系可用如图 3-19 所示的这样一棵倒置的树来描述。另外，像组织机构（如处、科、室），行政区（国家、省、市、县），书籍目录（书、章、节、小节）等，这些具有层次关系的数据，都可以用树这种数据结构来描述。

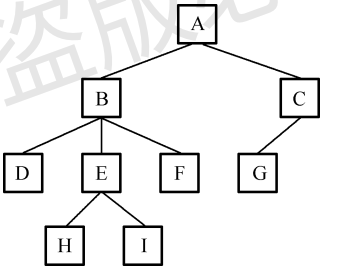


图 3-19 树的示例

在用图形表示数据结构中元素之间的前、后件关系时，一般使用有向箭头。但在树形结构中，由于前、后件关系非常清楚，即使去掉箭头也不会引起歧义，因此，图 3-19 中使用无向线段代表数据元素之间的逻辑关系（即前、后件关系）。

结合图 3-19 介绍树的相关术语如表 3-3 所示。

表 3-3 树的相关术语

术　语	定　义	示例（见图 3-19）
父节点（根）	在树结构中，每一个节点只有一个前件，称为父节点。没有前件的节点只有一个，称为树的根节点，简称树的根	节点 A 是树的根节点

续表

术　语	定　义	示例（见图 3-19）
子节点和叶子节点	在树结构中，每一个节点可以有多个后件，称为该节点的子节点。没有后件的节点称为叶子节点	节点 D、H、I、F、G 均为叶子节点
度	在树结构中，一个节点所拥有后件的个数称为该节点的度，所有节点中最大的度称为树的度	根节点 A 和节点 E 的度为 2，节点 B 的度为 3，节点 C 的度为 1，叶子节点 D、H、I、F、G 的度为 0。所以，该树的度为 3
深度	定义一棵树的根节点所在的层次为 1，其他节点所在的层次等于它的父节点所在的层次加 1，树的最大层次称为树的深度	根节点 A 在第 1 层，节点 B、C 在第 2 层，节点 D、E、F、G 在第 3 层，节点 H、I 在第 4 层。所以，该树的深度为 4
子树	在树中，以某节点的一个子节点为根构成的树称为该节点的一棵子树	节点 A 有 2 棵子树，它们分别以 B、C 为根节点。节点 B 有 3 棵子树，它们分别以 D、E、F 为根节点，其中，以 D、F 为根节点的子树实际上只有根节点一个节点。树的叶子节点度为 0，所以没有子树

3.6.2　二叉树及其基本性质

在树形结构中最常用的结构为二叉树。二叉树是一种特殊的树结构，它的每个节点最多有两个子节点，且有先后次序。

1．二叉树的定义

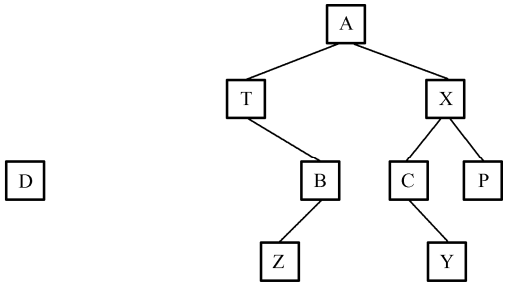
二叉树（Binary Tree）是一种很有用的非线性结构。二叉树不同于前面介绍的树结构，但它与树结构很相似，并且，树结构的所有术语都可以用到二叉树这种数据结构上。

二叉树具有以下两个特点：

（1）非空二叉树只有一个根节点；

（2）每一个节点最多有两棵子树，且分别称为该节点的左子树和右子树。

由以上特点可以看出，在二叉树中，每一个节点的度最大为 2，即所有子树（左子树或右子树）也均为二叉树；而树结构中的每一个节点的度可以是任意的。另外，二叉树中的每一个节点的子树被明显地分为左子树与右子树。在二叉树中，一个节点可以只有左子树而没有右子树，也可以只有右子树而没有左子树。当一个节点既没有左子树也没有右子树时，该节点即为叶子节点。如图 3-20（a）所示是一棵只有根节点的二叉树，如图 3-20（b）所示是一棵深度为 4 的二叉树。



（a）只有根节点的二叉树　　（b）深度为4的二叉树

图 3-20　二叉树的示例

2．二叉树的基本性质

性质 1：在二叉树的第 k 层上，最多有 2^{k-1}（$k\geqslant 1$）个节点。

例如，二叉树的第 1 层最多有 $2^{1-1}=2^0=1$ 个节点，第 3 层最多有 $2^{3-1}=2^2=4$ 个节点。满二叉树就是每层的节点数都是最大节点数的二叉树。

性质 2：深度为 m 的二叉树最多有 2^m-1 个节点。

证明：深度为 m 的二叉树是指二叉树共有 m 层，根据性质 1，只要将第 1 层到第 m 层上的最大的节点数相加，就可以得到整个二叉树中节点数的最大值，即

$$2^{1-1}+2^{2-1}+\cdots+2^{m-1}=2^m-1$$

例如，深度为 3 的二叉树，最多有 $2^3-1=7$ 个节点。

性质 3：在任意一棵二叉树中，度为 0 的节点（即叶子节点）总是比度为 2 的节点多 1 个。如果叶子节点数为 n_0，度为 2 的节点数为 n_2，则 $n_0=n_2+1$。

例如，在如图 3-20（b）所示的二叉树中，有 3 个叶子节点，有 2 个度为 2 的节点，度为 0 的节点比度为 2 的节点多 1 个。

性质 4：具有 n 个节点的二叉树，其深度至少为$[\log_2 n]+1$，其中$[\log_2 n]$表示取 $\log_2 n$ 的整数部分。

例如，有 6 个节点的二叉树中，其深度至少为$[\log_2 6]+1=2+1=3$。

3．满二叉树与完全二叉树

满二叉树与完全二叉树是两种特殊形态的二叉树。

（1）满二叉树。所谓满二叉树是指这样的一种二叉树：除最后一层外，每一层上的所有节点都有两个子节点。在满二叉树中，每一层上的节点数都达到最大值，即在满二叉树的第 k 层上有 2^{k-1} 个节点，且深度为 m 的满二叉树有 2^m-1 个节点。如图 3-21 所示分别是深度为 2、3、4 的满二叉树。

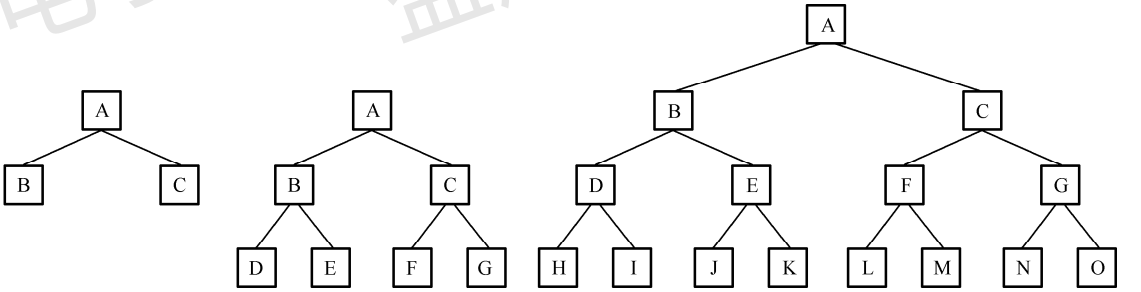
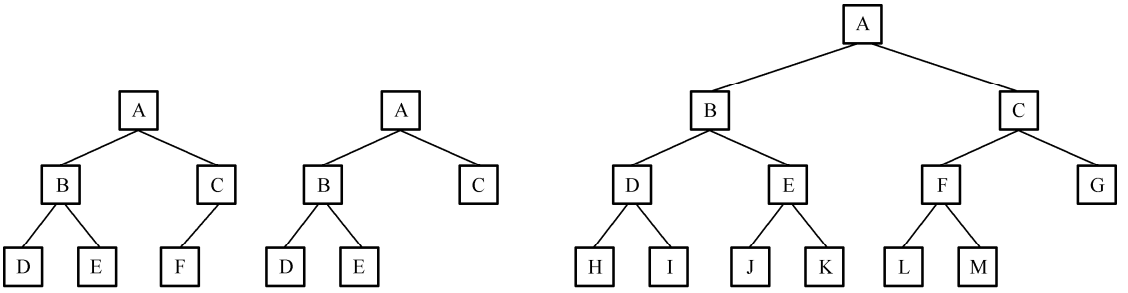


图 3-21　满二叉树

（2）完全二叉树。所谓完全二叉树是指这样的一种二叉树：除最后一层外，每一层上的节点数均达到最大值，在最后一层上只缺少右边的若干节点。

更确切地说，如果从根节点起，对二叉树的节点自上而下、自左至右用自然数进行连续编号，则深度为 m 且有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 m 的满二叉树中编号从 1 到 n 的节点一一对应时，称之为完全二叉树。如图 3-22 所示分别是深度为 3、4 的完全二叉树。



（a）深度为3的完全二叉树　　（b）深度为4的完全二叉树

图 3-22　完全二叉树

对于完全二叉树来说，叶子节点只可能在层次最大的两层上出现；对于任何一个节点，若其右分支下的子孙节点的最大层次为 p，则其左分支下的子孙节点的最大层次或为 p，或为 p+1。可以看出，满二叉树也是完全二叉树，而完全二叉树一般不是满二叉树。

完全二叉树还具有以下两个性质：

性质 5：具有 n 个节点的完全二叉树的深度为$[\log_2 n]+1$。

性质 6：设完全二叉树共有 n 个节点。如果从根节点开始，按层序（每一层从左到右）用自然数 1，2，…，n 给节点进行编号，则对于编号为 k（k=1，2，…，n）的节点有以下结论：

①若 k=1，则该节点为根节点，它没有父节点；若 k>1，则该节点的父节点编号为 $[k/2]$，其中$[k/2]$表示取 $k/2$ 的整数部分。

②若 $2k \leqslant n$，则编号为 k 的节点的左子节点编号为 $2k$，否则该节点无左子节点（也无右子节点）。

③若 $2k+1 \leqslant n$，则编号为 k 的节点的右子节点编号为 $2k$+1，否则该节点无右子节点。

根据完全二叉树的这个性质，如果按从上到下、从左到右的顺序存储完全二叉树的各节点，则很容易确定每一个节点的父节点、左子节点和右子节点的位置。

3.6.3　二叉树的存储结构

在计算机中，二叉树通常采用链式存储结构。与线性链表类似，用于存储二叉树中各元素的存储节点也由两部分组成：数据域和指针域。但在二叉树中，由于每一个元素可以有两个后件（即两个子节点），因此用于存储节点的指针域有两个：一个用于指向该节点的左子节点的存储地址，称为左指针域；另一个用于指向该节点的右子节点的存储地址，称为右指针域。二叉树存储节点的结构如图 3-23 所示。

图 3-23　二叉树存储节点的结构

由于二叉树的存储结构中每一个存储节点有两个指针域，因此，二叉树的链式存储结构也称为二叉链表。对于满二叉树与完全二叉树可以按层次进行顺序存储。

3.6.4 二叉树的遍历

遍历二叉树是二叉树的一种重要运算。所谓遍历是指沿某条搜索路径周游二叉树，对树中每个节点访问一次且仅访问一次。在遍历二叉树时，一般先遍历左子树，再遍历右子树。在先左后右的原则下，根据访问根节点的次序，二叉树的遍历分为三类：前序遍历、中序遍历和后序遍历。

（1）前序遍历（DLR）。先访问根节点，然后遍历左子树，最后遍历右子树，并且在遍历左、右子树时仍然先访问根节点，然后遍历左子树，最后遍历右子树。

（2）中序遍历（LDR）。先遍历左子树，然后访问根节点，最后遍历右子树，并且在遍历左、右子树时仍然先遍历左子树，然后访问根节点，最后遍历右子树。

（3）后序遍历（LRD）。先遍历左子树，然后遍历右子树，最后访问根节点，并且在遍历左、右子树时仍然先遍历左子树，然后遍历右子树，最后访问根节点。

例如，对于如图 3-24 所示的二叉树，前序遍历序列为 ABDEC，中序遍历序列为 BEDAC，后序遍历序列为 EDBCA。

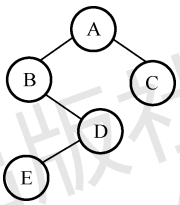


图 3-24 一棵二叉树

3.7 查 找 技 术

查找又称检索，就是在某种数据结构中找出满足指定条件的元素。查找是插入和删除等运算的基础，是数据处理的重要内容。由于数据结构是算法的基础，因此，对于不同的数据结构，应选用不同的查找算法，以获得更高的查找效率。本节将对顺序查找和二分法查找的概念进行详细说明。

3.7.1 顺序查找

顺序查找是最简单的查找方法，它的基本思想是：从线性表的第一个元素开始，逐个将线性表中的元素与被查元素进行比较，如果相等，则查找成功，停止查找；若整个线性表扫描完毕，仍未找到与被查元素相等的元素，则表示线性表中没有要查找的元素，查找失败。

例如，在一维数组（30，65，83，18，56，76，85）中查找数据元素 18，首先从第 1 个元素 30 开始进行比较，与要查找的数据不相等；接着与第 2 个元素 65 进行比较，以此类推；当进行到与第 4 个元素比较时，它们相等，所以查找成功。如果查找数据元素 100，则整个线性表扫描完毕，仍未找到与 100 相等的元素，表示线性表中没有要查找的元素。

时间复杂度和空间复杂度是衡量一个算法好坏的两个标准。在进行查找运算时，人们更关心的是时间复杂度。下面分析在长度为 n 的线性表中应用顺序查找算法的时间复杂度。

（1）最好情况：第一个元素就是要查找的元素，则比较次数为 1 次。

（2）最坏情况：最后一个元素是要查找的元素，或者在线性表中没有要查找的元素，则需要与线性表中的所有元素比较，比较次数为 n 次。

（3）平均情况：需要比较 $n/2$ 次，因此查找算法的时间复杂度为 $O(n)$

由此可以看出，对于大的线性表来说，顺序查找的效率是很低的。虽然顺序查找的效率不高，但在下列两种情况下也只能采用顺序查找：

（1）如果线性表为无序表（即表中元素的排列是无序的），则不管是顺序存储结构还是链式存储结构，都只能用顺序查找。

（2）即使是有序线性表，如果采用链式存储结构，也只能用顺序查找。

3.7.2 二分法查找

二分法查找又称折半查找，它是一种效率较高的查找方法。但是，二分法查找要求线性表是有序表，即表中节点按关键字有序，并且要求以顺序方式存储。二分法查找的基本思想是：首先将被查元素与“中间位置”的元素比较，若相等则查找成功；若被查元素大于“中间位置”的元素值，则在后半部继续进行二分法查找；否则在前半部继续进行二分法查找。

例如，假设被查找的有序表中关键字序列为：

05，13，19，21，37，56，64，75，80，88，92

当被查元素值为 21 时，进行二分法查找的过程如图 3-25 所示，图中用方括号表示当前的查找区间，用“↑”表示中间位置。

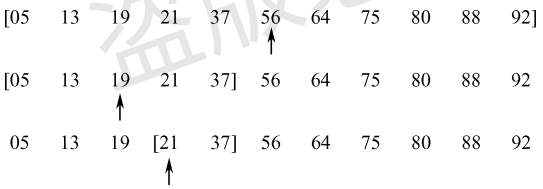


图 3-25 二分法查找过程示意图

顺序查找法的每一次比较，只将查找范围减少 1；而二分法查找每比较一次，可将查找范围减少为原来的一半，效率大大提高。可以证明，对于长度为 n 的有序线性表，在最坏情况下，二分法查找只需比较 $\log_2 n$ 次，而顺序查找需要比较 n 次。

3.8 排 序 技 术

排序也是数据处理的重要内容。所谓排序是指将一个无序序列整理成按值非递减顺序排列的有序序列。排序的方法有很多，根据待排序序列的规模以及对数据处理的要求，可以采用不同的排序方法。本节主要介绍一些常用的排序方法，其排序的对象一般认为是顺序存储的线性表，在程序设计语言中就是一维数组。

3.8.1 交换类排序法

交换类排序法是借助数据元素的“交换”来进行排序的一种方法。本小节介绍快速排序法和冒泡排序法，它们都使用交换排序方法。

1. 冒泡排序法

冒泡排序法是最简单的一种交换类排序方法。在数据元素的序列中，对于某个元素，如果其后存在一个元素小于它，则称为存在一个逆序。

冒泡排序（Bubble Sort）的基本思想就是通过两两相邻数据元素之间的比较和交换，不断地消去逆序，直到所有数据元素有序为止。

（1）冒泡排序法的思想。第一遍，在线性表中，从前往后扫描，如果相邻的两个数据元素，前面的元素大于后面的元素，则将它们交换，并称为消去了一个逆序。在扫描过程中，线性表中最大的元素不断地往后移动，最后被交换到了表的末端。此时，该元素就已经排好序了。

然后，对当前还未排好序的范围内的全部节点，从后往前扫描，如果相邻两个数据元素，后面的元素小于前面的元素，则将它们交换，也称为消去了一个逆序。在扫描过程中，最小的元素不断地往前移动，最后被换到了线性表的第一个位置，则认为该元素已经排好序了。

对还未排好序的范围内的全部节点，继续第二遍、第三遍的扫描，这样，未排好序的范围逐渐减小，最后为空，则线性表已经变为有序的了。

冒泡排序每一遍的从前往后扫描都把排序范围内的最大元素沉到了表的底部，每一遍的从后往前扫描都把排序范围内的最小元素像气泡一样浮到了表的最前面。冒泡排序的名称也由此而来。

（2）冒泡排序法的例子。如图 3-26 所示是一个冒泡排序法的例子，对（4，1，6，5，2，3）这样一个 6 个元素组成的线性表排序。图中每一遍结果中方括号“[ ]”外的元素是已经排好序的元素，方括号“[ ]”内的元素是还未排好序的元素。可以看到，方括号“[ ]”的范围在逐渐减小。具体的说明如下所述。

原始序列	4	1	6	5	2	3
第一遍（从前往后）	[1	4	5	2	3]	6
（从后往前）	1	[2	4	5	3]	6
第二遍（从前往后）	1	[2	4	3]	5	6
（从后往前）	1	2	[3	4]	5	6
最终结果	1	2	3	4	5	6

图 3-26 冒泡排序示例

第一遍的从前往后扫描：首先比较“4”和“1”，前面的元素大于后面的元素，这是一个逆序，两者交换；交换后接下来是“4”和“6”比较，不需要交换；然后“6”与“5”比较，这是一个逆序，则相互交换；“6”再与“2”比较，交换；“6”再与“3”比较，交换。这时，排序范围内（即整个线性表）的最大元素“6”已经到达表的底部，它已经到达了它在有序表中应有的位置。

第一遍的从前往后扫描的最后结果为（1，4，5，2，3，6）。

第一遍的从后往前扫描：由于数据元素“6”已经排好序，因此，现在的排序范围为（1，4，5，2，3）。先比较“3”和“2”，不需要交换；比较“2”和“5”，后面的元素小于前面的元素，这是一个逆序，互相交换；比较“2”和“4”，这是一个逆序，互相交换；比较“2”和“1”，不需要交换。此时，排序范围内（1，4，5，2，3）的最小元素“1”已经到达表头，它已经到达了它在有序表中应有的位置。

第一遍的从后往前扫描的最后结果为（1，2，4，5，3，6）。

第二遍的排序过程略。

假设线性表的长度为 n，则在最坏情况下，冒泡排序需要经过 $n/2$ 遍的从前往后的扫描和 $n/2$ 遍的从后往前的扫描，需要的比较次数为 $n(n-1)/2$。但这个工作量不是必需的，一般情况下要小于这个工作量。冒泡排序的时间效率为 $O(n^2)$。

2．快速排序法

在冒泡排序中，一次扫描只能确保最大的元素或最小的元素移到了正确位置，而未排序序列的长度可能只减少了 1。快速排序（Quick Sort）是对冒泡排序方法的一种本质的改进。

（1）快速排序法的思想。快速排序的基本思想是：在待排序的 n 个元素中取一个元素 K（通常取第一个元素），以元素 K 作为分割标准，把所有小于 K 元素的数据元素都移到 K 前面，把所有大于 K 元素的数据元素都移到 K 后面。这样，以 K 为分界线，把线性表分割为两个子表，这称为一趟排序。然后，对 K 前后的两个子表分别重复上述过程，继续下去，直到分割的子表的长度为 1 为止。这时，线性表已经是排好序的了。

第一趟快速排序的具体做法是：附设两个指针 low 和 high，它们的初值分别指向线性表的第一个元素（K 元素）和最后一个元素。首先从 high 所指的位置向前扫描，找到第一个小于 K 元素的数据元素并与 K 元素互相交换。然后从 low 所指位置向后扫描，找到第一个大于 K 元素的数据元素并与 K 元素交换。重复这两步，直到 low=high 为止。

（2）快速排序法的例子。快速排序过程如图 3-27 所示。

初始状态	45	30	61	82	74	12	26	49
	↑ low							↑ high
high向左扫描	45	30	61	82	74	12	26	49
第一次交换后	26	30	61	82	74	12	45	49
	↑ low						↑ high	
low向右扫描	26	30	61	82	74	12	45	49
第二次交换后	26	30	45	82	74	12	61	49
high向左扫描并交换后	26	30	12	82	74	45	61	49
low向右扫描并交换后	26	30	12	45	74	82	61	49
				↑ low	↑ high			
high向左扫描	26	30	12	45	74	82	61	49

（a）第一趟扫描过程

初始状态	45	30	61	82	74	12	26	49
第一趟排序后	[26	30	12]	45	[74	82	61	49]
第二趟排序后	[12]	26	[30]	45	[49	61]	74	[82]
第三趟排序后	12	26	30	45	49	[61]	74	82
排序结果	12	26	30	45	49	61	74	82

（b）各趟排序后的状态

图 3-27　快速排序示例

初始状态下，low 指针指向第一个元素 45，high 指针指向最后一个元素 49。首先，从 high 所指的位置向前（左）扫描，找到第一个比 45 小的元素，即找到 26 时，26 与 45 交换位置，此时 low 指针指向元素 26，high 指针指向元素 45；然后从 low 所指位置起向后（右）扫描，找到第一个比 45 大的元素，即找到 61 时，61 与 45 交换位置，此时 low 指针指向元素 45，high 指针指向元素 61；重复这两步，直到 low=high 为止。所以，第一趟排序后的结果为（26，30，12，45，74，82，61，49）。

以后的排序方法与第一趟的扫描过程一样，直到最后的排序结构为有序序列为止。

快速排序的平均时间效率最佳，为 $O(n\log_2 n)$；最坏情况下，即每次划分只得到一个子序列，时间效率为 $O(n^2)$。

快速排序被认为是目前所有排序算法中最快的一种。但若初始序列有序或者基本有序时，快速排序蜕化为冒泡排序。

3.8.2 插入类排序法

插入排序是每次将一个待排序元素按其元素值的大小插入到前面已经排好序的子表中的适当位置，直到全部元素插入完成为止。

1．简单插入排序法

（1）简单插入排序法的思想。简单插入排序是把 n 个待排序的元素看成是一个有序表和一个无序表，开始时有序表只包含一个元素，而无序表包含另外 $n-1$ 个元素；每次取无序表中的第一个元素插入到有序表中的正确位置，使之成为增加一个元素的新的有序表；插入元素时，插入位置及其后的记录依次向后移动。最后有序表的长度为 n，而无序表为空，此时排序完成。

（2）简单插入排序法的例子。简单插入排序过程如图 3-28 所示。图中方括号“[ ]”内为有序的子表，方括号“[ ]”外为无序的子表，每次从无序子表中取出第一个元素插入到有序子表中。

初始	[48]	37	65	96	75	12	26	49
i=2	[37	48]	65	96	75	12	26	49
i=3	[37	48	65]	96	75	12	26	49
i=4	[37	48	65	96]	75	12	26	49
i=5	[37	48	65	75	96]	12	26	49
i=6	[12	37	48	65	75	96]	26	49
i=7	[12	26	37	48	65	75	96]	49
i=8	[12	26	37	48	49	65	75	96]

图 3-28 简单插入排序过程

开始时，有序表只包含一个元素 48，而无序表包含其他 7 个元素。

当 i=2 时，即把第 2 个元素 37 插入到有序表中，37 比 48 小，所以在有序表中的序列为[37，48]；

当 i=3 时，即把第 3 个元素 65 插入到有序表中，65 比前面 2 个元素大，所以在有序表中的序列为[37，48，65]；

当 i=4 时，即把第 4 个元素 96 插入到有序表中，96 比前面 3 个元素大，所以在有序表中

的序列为[37，48，65，96]；

当 i=5 时，即把第 5 个元素 75 插入到有序表中，75 比前面 3 个元素大，比 96 小，所以在有序表中的序列为[37，48，65，75，96]；

以此类推，直到所有的元素都插入到有序序列中。

在最好情况下，即初始排序序列就是有序的情况下，简单插入排序的比较次数为 n−1，移动次数为 0。

在最坏情况下，即初始排序序列是逆序的情况下，比较次数为 n（n−1）/2，移动次数为 n（n−1）/2。假设待排序的线性表中的各种排列出现的概率相同，因此，直接插入排序算法的时间复杂度为 O（n^2）。

在简单插入排序中，每一次比较后最多消去一个逆序，因此，这种排序方法的效率与冒泡排序法相同。

2．希尔排序法

希尔排序（Shell Sort）又称为“缩小增量排序”，它也是一种插入类排序的方法，但在时间效率上较简单插入排序有较大的改进。

（1）希尔排序法的思想：将整个无序序列分割成若干小的子序列分别进行插入排序。

子序列的分割方法如下：

将相隔某个增量 d 的元素构成一个子序列。在排序过程中，逐次减小这个增量，最后当 d 减到 1 时进行一次插入排序，排序就完成了。

增量序列一般取 $d_i=n/2^i$（i=1，2，…，[$\log_2 n$]），其中，n 为待排序序列的长度。

在希尔排序过程中，虽然对于每一个子表采用的仍是插入排序，但是，在子表中每进行一次比较就有可能消去整个线性表中的多个逆序，从而改善了整个排序过程的性能。

希尔排序的效率与所选取的增量序列有关。如果选取上述增量序列，则在最坏情况下，希尔排序所需要的时间效率为 O（$n^{1.3}$）。

（2）希尔排序法的例子。希尔排序过程如图 3-29 所示。此序列共有 10 个数据，即 n=10，则增量 $d_1=10/2^1=5$，将所有距离为 5 的倍数的元素放在一组中，组成了一个子序列，即各子序列为（48，13）、（37，26）、（64，50）、（96，54）、（75，5），对各子序列进行从小到大的排序后，得到第一趟排序结果（13，26，50，54，5，48，37，64，96，75）。

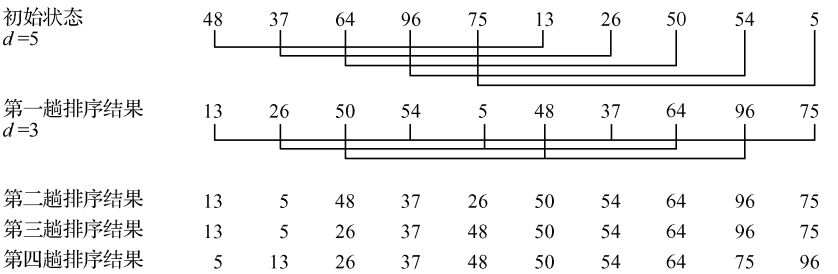


图 3-29　希尔排序过程

接着增量 $d_2=10/2^2=10/4=3$，将所有距离为 3 的倍数的元素放在一组中，组成了一个子序列，即各子序列为（13，54，37，75）、（26，5，64）、（50，48，96），对各子序列进行从小到大的排序后，得到第二趟排序结果（13，5，48，37，26，50，54，64，96，75）。

以此类推，直到得到最终结果。

3.8.3 选择类排序法

选择排序的基本思想是通过每一趟从待排序序列中选出值最小的元素，顺序放在已排好序的有序子表的后面，直到全部序列满足排序要求为止。本小节介绍简单选择排序法和堆排序法。

1．简单选择排序法

选择排序法的基本思想是：扫描整个线性表，从中选出最小的元素，将它交换到表的最前面（这是它应有的位置）；然后对剩下的子表采用同样的方法，直到子表空为止。

对于长度为 n 的序列，选择排序需要扫描 $n-1$ 遍，每一遍扫描均从剩下的子表中选出最小的元素，然后将该最小的元素与子表中的第一个元素进行交换。如图 3-30 所示是这种排序法的示意图，图中有方框的元素是刚被选出来的最小元素。

原序列	89	21	56	48	85	16	19	47
第1遍选择	[16]	21	56	48	85	89	19	47
第2遍选择	16	[19]	56	48	85	89	21	47
第3遍选择	16	19	[21]	48	85	89	56	47
第4遍选择	16	19	21	[47]	85	89	56	48
第5遍选择	16	19	21	47	[48]	89	56	85
第6遍选择	16	19	21	47	48	[56]	89	85
第7遍选择	16	19	21	47	48	56	[85]	89

图 3-30　简单选择排序法示意图

简单选择排序法在最坏情况下需要比较 $n(n-1)/2$ 次。

2．堆排序法

（1）堆的定义。若有 n 个元素的序列（h_1，h_2，…，h_n），将元素按顺序组成一棵完全二叉树，当且仅当满足下列条件时称为堆：

① $\begin{cases} h_i \geqslant h_{2i} \\ h_i \geqslant h_{2i+1} \end{cases}$　或者　② $\begin{cases} h_i \leqslant h_{2i} \\ h_i \leqslant h_{2i+1} \end{cases}$

其中，$i=1$，2，3，…，$n/2$。

情况①称为大根堆，所有节点的值大于或等于左、右子节点的值。情况②称为小根堆，所有节点的值小于或等于左、右子节点的值。本节只讨论大根堆的情况。

例如，序列（91，85，53，47，30，12，24，36）是一个堆，则它对应的完全二叉树如图 3-31（c）所示。

（2）调整建堆。在调整建堆的过程中，总是将根节点值与左、右子树的根节点进行比较，若不满足堆的条件，则将左、右子树根节点值中的大者与根节点值进行交换，这个调整过程从根节点开始一直延伸到所有叶子节点，直到所有子树均为堆为止。

假设如图 3-31（a）所示是某完全二叉树的一棵子树。在这棵子树中，根节点 47 的左、右子树均为堆，为了将整个子树调整成堆，首先将根节点 47 与其左、右子树的根节点进行比较，此时由于左子树根节点 91 大于右子树根节点 53，且它又大于根节点 47，因此，根据堆的条件，应将元素 47 与 91 交换，如图 3-31（b）所示。经过一次交换后，破坏了原来左子树

的堆结构，需要对左子树再进行调整，将元素 85 与 47 进行交换，调整后的结果如图 3-31（c）所示。

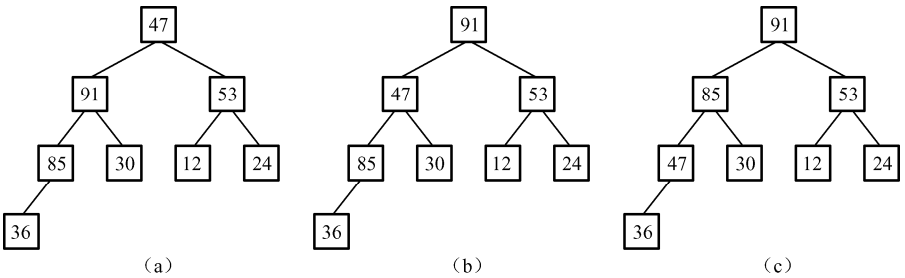


图 3-31　堆顶元素为最大的建堆过程

（3）堆排序。首先将一个无序序列建成堆，然后将堆顶元素与堆中的最后一个元素交换。不考虑已经换到最后的那个元素，将剩下的 $n-1$ 个元素重新调整为堆。重复执行此操作，直到所有元素有序为止。

对于数据元素较少的线性表来说，堆排序的优越性并不明显，但对于大量的数据元素来说，堆排序是很有效的。堆排序最坏情况需要 $O(n\log_2 n)$ 次比较。

3.8.4　排序方法比较

综合比较本节介绍的 3 类共 6 种排序方法的时间和空间复杂度，结果如表 3-4 所示。

表 3-4　常用排序方法的时间和空间复杂度比较

类　型	方　法	时间复杂度			空间复杂度	稳　定　性	复　杂　性
		平均时间	最坏情况	最好情况			
交换排序	冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
	快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
插入排序	插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
	希尔排序	$O(n^{1.3})$	与所取增量序列有关		$O(1)$	不稳定	较复杂
选择排序	选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定	较复杂

不同的排序方法各有优、缺点，可根据需要运用到不同的场合。

选取排序方法时需要考虑的因素有待排序的序列长度 n、数据元素本身的大小、关键字的分布情况、对排序的稳定性要求、语言工具的条件、辅助空间的大小等。根据这些因素，可以得出以下几点结论：

（1）如果 n 较小，可采用插入排序和选择排序。由于简单插入排序所需数据元素的移动操作比简单选择排序多，因而当数据元素本身信息量较大时，用简单选择排序方法较好。

（2）如果文件的初始状态已是基本有序，则最好选用简单插入排序或冒泡排序。

（3）如果 n 较大，则应选择快速排序或堆排序。快速排序是目前内部排序方法中性能最好的。当待排序的序列是随机分布时，快速排序的平均时间最少。但堆排序所需的辅助空间要少于快速排序，并且不会出现最坏情况。

本 章 小 结

本章介绍了算法与数据结构的基本概念。数据结构是指相互有关联的数据元素的集合。数据结构又分为数据的逻辑结构和数据的存储结构。线性表是最简单最常用的一种数据结构，栈和队列是运算受限的线性表。树是一类重要的非线性结构。排序与查找是数据处理中经常使用的重要算法。

理解和掌握算法与数据结构的基础知识，使学生拥有最基本的软件开发能力，为今后计算机知识的学习打下基础。

本章内容复习

1．填空题

（1）常用的数据类型有_____、_____、_____等。

（2）在数据结构中，从逻辑上可以把数据结构分为_____和_____两种。

（3）算法的复杂度主要包括时间复杂度和_____。

（4）数据结构包括 3 方面内容：数据的逻辑结构，数据的_____，数据的运算。

（5）对 *n* 个元素的序列进行冒泡排序时，最坏情况下，需要比较的次数是_____，其时间复杂度为_____。

（6）对长度为 *n* 的线性表进行顺序查找，在最坏情况下所需要的比较次数为_____。

（7）在长度为 *n* 的有序线性表中进行二分查找，需要的比较次数为_____。

（8）设一棵完全二叉树共有 700 个节点，则在该二叉树中有_____个叶子节点。

（9）在一个容量为 15 的循环队列中，若头指针 front=6，尾指针 rear=9，则该循环队列中共有_____个元素。

（10）_____是先进先出的线性表，_____是先进后出的线性表。

2．选择题

（1）算法的时间复杂度是指（　　）。

A．执行算法程序所需要的时间　　B．算法程序的长度

C．算法执行过程中所需要的基本运算次数　　D．算法程序中的指令条数

（2）算法的空间复杂度是指（　　）。

A．算法程序的长度　　B．算法程序中的指令条数

C．算法程序所占的存储空间　　D．算法执行过程中所需要的存储空间

（3）下列叙述中正确的是（　　）。

A．线性表是线性结构　　B．栈与队列是非线性结构

C．线性链表是非线性结构　　D．二叉树是线性结构

（4）数据结构在计算机存储空间的存放形式称为（　　）。

A．数据的存储结构　　B．数据结构

C．数据的逻辑结构　　D．数据元素之间的关系

（5）下述哪一条是顺序存储方式的优点？（　　）

A．插入运算方便

B．存储密度大

C．删除运算方便

D．可方便地用于各种逻辑结构的存储表示

（6）栈和队列的共同点是（　　）。

A．都是先进先出　　B．都是先进后出

C．只允许在端点处插入和删除元素　　D．没有共同点

（7）一个队列的入队序列是 1，2，3，4，则队列的输出序列是（　　）。

A．1，4，3，2　　B．3，2，4，1　　C．4，3，2，1　　D．1，2，3，4

（8）一个栈的入栈序列是 a，b，c，d，e，则不可能的出栈序列是（　　）。

A．edcba　　B．decba　　C．dceab　　D．abcde

（9）若进栈序列为 1，2，3，4，假设进栈和出栈可以穿插进行，则可能的出栈序列是（　　）。

A．2，4，1，3　　B．3，1，4，2　　C．3，4，1，2　　D．1，2，3，4

（10）链表不具备的特点是（　　）。

A．可随机访问任意一个节点　　B．插入和删除不需要移动任何元素

C．不必事先估计存储空间　　D．所需空间与其长度成正比

（11）深度为 5 的二叉树至多有（　　）个节点。

A．16　　B．32　　C．31　　D．10

（12）设树 T 的度为 4，其中度为 1、2、3、4 的节点个数分别为 4、2、1、1，则 T 中的叶子节点个数为（　　）。

A．8　　B．7　　C．6　　D．5

（13）某二叉树有 5 个度为 2 的节点，则该二叉树中的叶子节点个数是（　　）。

A．10　　B．8　　C．6　　D．4

（14）下列关于二叉树的叙述中，正确的是（　　）。

A．叶子节点总是比度为 2 的节点少一个

B．叶子节点总是比度为 2 的节点多一个

C．叶子节点数是度为 2 的节点数的两倍

D．度为 2 的节点数是度为 1 的节点数的两倍

（15）一棵二叉树共有 25 个节点，其中 5 个是叶子节点，则度为 1 的节点数为（　　）。

A．16　　B．10　　C．6　　D．4

（16）某二叉树共有 7 个节点，其中叶子节点有 1 个，则该二叉树的深度为（假设根节点在第 1 层）（　　）。

A．3　　B．4　　C．6　　D．7

（17）一棵二叉树的前序遍历序列为 ABDGCFK，中序遍历序列为 DGBAFCK，则后序遍历序列是（　　）。

A．AFCKDGB　　B．GDBFKCA　　C．KCFAGDB　　D．ABCDFKG

（18）对线性表进行二分查找时，要求线性表必须（　　）。

A．以顺序方式存储

B．以链接方式存储
C．以顺序方式存储，且节点按关键字有序排列
D．以链接方式存储，且节点按关键字有序排列

网上资源查找

（1）请从互联网上查找各种排序方法的相关实例。
（2）请从互联网上查找有关介绍栈和队列的应用实例。