第 3 章　复杂数据的存储与处理

复杂数据往往由大量有联系的数据组成，并且数据的存储方式也影响着处理的方法，因此，研究者应掌握必要的复杂数据的存储与处理方法。

本章主要介绍数据的逻辑结构和存储结构及算法的基本概念，并介绍常见的线性结构与非线性结构及运算，以及查找和排序算法。

3.1　算法与数据结构

3.1.1　算法

1．算法与程序

（1）算法的定义

算法是指对解题方案准确而完整的描述，即一组严谨地定义运算顺序的规则，并且每个规则都是有效的、明确的，没有二义性，同时该规则在有限次运算后可终止。

算法可以理解为由基本运算及规定的运算顺序构成的完整的解题步骤，或者将其看成按照要求设计好的、有限的、确切的计算序列，并且这样的步骤和序列可以解决一类问题。

（2）程序的定义

程序是为实现特定目标或解决特定问题而用程序设计语言描述的适合计算机执行的指令（语句）序列。一个程序应该包括以下两方面的内容。

① 对数据的描述。在程序中要指定数据的类型和数据的组织形式，即数据结构。

② 对操作的描述。即操作步骤，也就是算法。

（3）算法与程序的区别

算法不等于程序，也不是计算方法。算法是指在逻辑层面上对解决问题的方法的一种描述。一个算法可以被很多不同的程序实现，即程序可以作为算法的一种描述，但程序通常还需考虑很多与算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。算法可以被程序模拟出来，但程序只是一个手段，让计算机机械式地执行，算法才是灵魂，驱动计算机“怎么去”执行。程序的编制不可能优于算法的设计，算法并不是程序或者函数本身。程序中的指令必须是机器可执行的，而算法中的指令则无此限制。

2．算法的基本特征

（1）可行性

算法中执行的任何计算步骤都可以被分解为基本的、可执行的操作步骤，即每个计算步骤都可以在有限时间内完成（也称为有效性）。

由于算法是为了在某一个特定的计算工具上解决某一个实际的问题而设计的，因此，它总是受到计算工具的限制，从而使执行产生偏差。例如，计算机的数值有效位是有限的，往往会因为有效位的影响而产生错误。因此，在算法设计时，必须要考虑它的可行性，要根据具体的系统调整算法，否则将不会得到满意的结果。

（2）确定性

算法的设计必须每个步骤都有明确的定义，不允许有模糊的解释，也不能有多义性。

（3）有穷性

算法的有穷性是指在一定的时间内能够完成指定的步骤，即算法应该在计算有限个步骤后能够正常结束。

例如，对于数学中的无穷级数，在计算机中只能求有限项，即计算的过程是有穷的。

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）输入项

一个算法有 0 个或多个输入项，以刻画运算对象的初始情况。所谓 0 个输入项是指算法本身定出了初始条件。

（5）输出项

一个算法有一个或多个输出项，以反映对输入项加工后的结果。没有输出项的算法是毫无意义的。

3．算法的复杂度

算法的复杂度包括时间复杂度和空间复杂度。

（1）时间复杂度

时间复杂度是指实现该算法需要的计算工作量。算法的工作量用算法所执行的基本运算次数来计算。

算法的时间复杂度反映了程序执行时间随输入规模增长而增长的量级，在很大程度上能很好地反映出算法的优劣。

从数学上定义，给定算法 A，如果存在函数 $F(n)$，当 $n=k$ 时，$F(k)$表示算法 A 在输入规模为 k 的情况下的运行时间，则称 $F(n)$为算法 A 的时间复杂度。

这里首先要明确输入规模的概念。输入规模是指算法 A 所接收输入的自然独立体的大小。例如，对于排序算法来说，输入规模一般是指待排序数据元素的个数，而对于求两个同型方阵乘积的算法，输入规模可以看作单个方阵的维数。为了简单起见，在下面的讨论中，我们总是假设算法的输入规模是用大于零的整数表示的，即 $n=1,2,3,\cdots,k$。

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（即两个实数相乘）次数为 n^3，即算法的工作量为 n^3，也就是时间复杂度为 n^3。

对于同一个算法，每次执行的时间不仅取决于输入规模，还取决于输入的特性和具体的硬件环境在某次执行时的状态。所以想要得到一个统一精确的 $F(n)$是不可能的。为了解决这个问题，需要说明以下两点。

① 忽略硬件及环境因素，假设每次执行时硬件条件和环境条件是完全一致的。

② 对于输入特性的差异，操作者将从数学上进行精确分析并代入函数解析式。

【例 3.1】 时间复杂度分析示例。

问题：输入一组从 1 到 n 的整数，但其顺序是完全随机的。其数据元素个数为 n，n 为大于零的整数。输出数据元素 n 所在的位置（第一个数据元素的位置为 1）。

这个问题非常简单，下面是其解决算法之一（伪代码）：

```
LocationN(A)
{
    for(int i=1;i<=n;i++)------------------------t1
        if(A[i] == n) ----------------------------t2
           return i;  -------------------------t3
}
```

在上述代码中，t1、t2 和 t3 分别表示此行代码执行一次需要的时间。

首先，输入规模 n 是影响算法执行时间的因素之一。在 n 固定的情况下，不同的输入序列会影响其执行时间。在最好的情况下，n 排在序列的第一个位置，那么此时的运行时间= t1+t2+t3。在最坏的情况下，n 排在序列的最后一位，则运行时间为 n×t1+n×t2+t3=(t1+t2)×n+t3。

可以看到，在最好情况下的运行时间是一个常数，而在最坏情况下的运行时间是输入规模的线性函数。那么，平均情况又是如何呢？

由于输入序列的顺序是完全随机的，即 n 出现在 1～n 这 n 个位置上的可能性相等，即概率均为 1/n。而在平均情况下的执行次数即执行次数的数学期望，其解为

$$\begin{aligned} E &= p(n=1)\times 1+p(n=2)\times 2+\cdots+p(n=n)\times n \\ &= (1/n)\times(1+2+\cdots+n) \\ &= (1/n)\times((n/2)\times(1+n)) \\ &= (n+1)/2 \end{aligned}$$

即在平均情况下 for 循环要执行(n+1)/2 次，则平均运行时间为(t1+t2)×(n+1)/2+t3。

由此得出分析结论：

t1+t2+t3 <= $F(n)$ <= (t1+t2)×n+t3，在平均情况下，$F(n)$ = (t1+t2)×(n+1)/2+t3。

（2）空间复杂度

空间复杂度是对一个算法在运行过程中临时占用存储空间大小的度量。一个算法在计算机存储器中所占用的存储空间，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，是通过参数表由调用函数传递而来的，它不随本算法的不同而改变。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不同，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，这种算法被称为“就地”进行的、节省存储的算法；有的算法需要占用的临时工作单元数量与解决问题的规模 n 有关，它随着 n 的增大而增大，当 n 较大时，将占用较多的存储单元，例如，快速排序和归并排序算法就属于这种情况。

分析一个算法所占用的存储空间要从各方面综合考虑，如递归算法，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栈，从而占用较多的临时工作单元；若写成非递归算法，一般比较长，算法本身占用的存储空间较多，但运行时将占用较少的临时工作单元。

计算一个算法的空间复杂度只考虑在运行过程中为局部变量分配的存储空间的大小，它包括为参数表中形参变量分配的存储空间和为在函数体中定义的局部变量分配的存储空间两部分。若一个算法为递归算法，其空间复杂度为递归所使用的栈空间的大小，它等于一次调用所分配的临时存储空间的大小乘以被调用的次数（即递归调用的次数加 1，这个 1 表示开始进行的一次非递归调用）。算法的空间复杂度一般也以数量级的形式给出，如当一个算法的空间复杂度为一个常量，即不随被处理数据规模 n 的大小而改变时，可表示为 $O(1)$；当一个算法的空间复杂度与以 2 为底的 n 的对数成正比时，可表示为 $O(\log_2 n)$；当一个算法的空间复杂度

与 n 成线性比例关系时，可表示为 $O(n)$。若形参为数组，则只需要为它分配一个用于存储由实参传送来的一个地址指针的空间，即一个机器字长空间；若形参为引用方式，则只需要为其分配用于存储一个地址的空间，用它来存储对应实参变量的地址，以便由系统自动引用实参变量。

例如：插入排序的时间复杂度是 $O(n^2)$，空间复杂度是 $O(1)$。而一般的递归算法要有 $O(n)$ 的空间复杂度，因为每次递归都要存储返回的信息。

对于一个算法，其时间复杂度和空间复杂度往往是相互影响的。当追求一个较好的时间复杂度时，可能会使空间复杂度的性能变差，即可能导致占用较多的存储空间；反之，当追求一个较好的空间复杂度时，可能会使时间复杂度的性能变差，即可能导致占用较长的运行时间。

3.1.2 数据结构

数据结构与算法之间存在本质联系，在某一类型的数据结构上，总要涉及其上施加的运算，而只有通过对定义运算的研究，才能清楚地理解数据结构的定义和作用；在涉及运算时，总会联系到该算法处理的对象和结果的数据。

数据是指计算机可以保存和处理的信息。

数据元素是构成数据的基本单位，在计算机程序中通常作为一个整体进行处理。一个数据元素可以由一个或多个数据项组成。数据元素也称为元素、节点或记录。

数据项是指数据中不可分割的、含有独立意义的最小单位，也称为字段（Field）或域。

例如，实现对会员档案的管理，设会员档案的内容包括编号、姓名、性别、出生年月、等级和积分，如表 3.1 所示，则每个会员的编号、姓名、性别、出生年月、等级和积分就是数据项，每个会员的所有数据项就构成一个数据元素。

表 3.1　会员档案登记表

编　　号	姓　　名	性　　别	出 生 年 月	等　　级	积　　分
601110901	蒋一奇	男	1975-10-2	32	1052
601110902	刘怡婧	女	1982-01-20	38	892
601110903	张鹏	男	1980-09-23	15	165
601110904	郑晓敏	女	1978-06-12	33	1125
601110905	吴涛	男	1978-03-5	52	78

数据结构是指相互之间存在一种或多种特定关系的数据元素集合，是带有结构的数据元素的集合。它指的是数据元素之间的相互关系，即数据的组织形式。数据结构是研究数据及数据元素之间关系及其操作的实现算法的学科。它不仅是一般程序设计的基础，而且是设计和实现编译程序、操作系统、数据库系统及其他系统程序和大型应用程序的重要基础。

下面通过例 3.2 来认识数据结构。

【例 3.2】 电话是通信联络必不可少的工具，如何用计算机来实现自动查询电话号码呢？要求对于给定的任意姓名，如果该人有电话号码，则迅速给出电话号码，否则，给出查找不到该人电话号码的信息。

对于这种问题，我们可以按照用户向电信局申请电话号码的先后次序建立电话号码表，存储到计算机中。在这种情况下，由于电话号码表是没有任何规律的，只能从第一个号码开始逐一进行查找，这种逐一按顺序进行查找的方式，效率非常低。为了提高查找的效率，我们可以根据每个用户姓名的第一个拼音字母，按照 26 个英文字母的顺序进行排列，这样根据姓名的第一个字母就可以迅速地进行查找，从而极大地减少了查找所需的时间。进一步地，可以按照用户的中文姓名的汉语拼音顺序进行排序，这样就可以进一步提高查找效率了。

在例 3.2 中，要解决的另一个问题是如何提高查找效率。为了解决这个问题，就必须了解待处理对象之间的关系，以及如何存储和表示这些数据。例 3.2 中的每个电话号码就是一个要处理的数据对象，也称为数据元素。在数据结构中为了抽象地表示不同的数据元素，以及为了研究对于具有相同性质的数据元素的共同特点和操作，人们将数据元素又称为数据节点，简称为节点。电话号码经过处理并按照汉语拼音排好顺序后，每个电话号码之间的先后次序就是数据元素之间的关系。

由此可见，计算机所处理的数据并不是数据的杂乱堆积，而是具有内在联系的数据集合，如表结构（见表 3.1）、树状结构（见图 3.1）、图状结构（见图 3.2）等，这里关心的是数据元素之间的相互关系与组织方式及其上施加的运算及运算规则，并不涉及数据元素的内容具体是什么值。

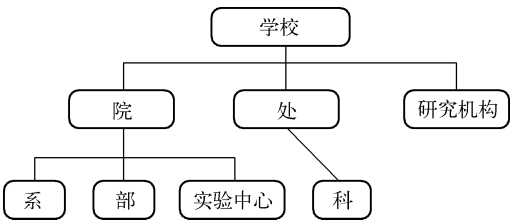


图 3.1　树状结构

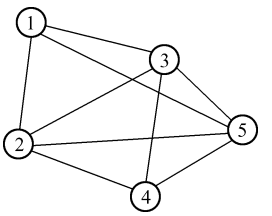


图 3.2　图状结构

例如，一维数组是向量 $\boldsymbol{A}=(a_1,a_2,\cdots,a_n)$的存储映象，使用时采用下标变量 $a[i]$的方式，关心的是其按序排列、按行存储的特性，并不关心 $a[i]$中存放的具体值。同理，二维数组 $b[i,j]$是矩阵 $\boldsymbol{B}_{m\times n}$的存储映象，我们关心的是结构关系的特性而不关心其数组元素本身的内容。

现实世界中客观存在的一切对象，在数据处理中都可以抽象成数据元素，如描述光谱颜色的名称（红、橙、黄、绿、蓝、紫）是光谱的数据元素；1～n 的各个整数可以作为整数数值的数据元素；描述物体的长、宽、高、重量等也是物体的数据元素。

在实际应用中，需要处理的数据元素一般有很多，而且，作为某种处理对象，其中的数据元素一般具有某种共同特征，如春、夏、秋、冬这四个数据元素有一个共同特征，即它们都是季节名，分别表示了一年中的四个季节，从而这四个数据元素构成了季节名的集合。

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的。

在一般情况下，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某种关系，这种关系反映了该集合中的数据元素所固有的结构。在数据处理领域中，人们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，在描述一年四个季节的顺序关系时，则“春”是“夏”的前件（即直接前驱，下同），而“夏”是“春”的后件（即直接后继，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。

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。

数据结构就是用于研究这类非数值处理的程序设计问题的。它包括三个方面的内容：数据的逻辑结构、数据的存储结构和数据的运算。

1．数据的逻辑结构

数据的逻辑结构是指数据元素之间的逻辑关系，与数据在计算机内部如何存储的无关，数据的逻辑结构是独立于计算机的。

例如，在城市交通中，两个地点之间就存在一种逻辑关系，两个地点之间的逻辑关系分为三种：第一，两个地点之间有公共汽车可以直达；第二，两个地点之间没有公共汽车可以直达，但可以通过中途换乘其他公共汽车而到达目的地；第三，两个地点之间没有公共汽车可以到达。

又如，在电话号码本中，电话号码如何进行分类，以及按照什么顺序进行排列等都是数据之间的逻辑关系。

数据的逻辑结构有如下两个要素。

① 数据元素的集合，记作 D。

② 数据之间的前后件关系，记作 R。

可得

$$数据结构=(D,R)$$

其中，D 是组成数据的数据元素的有限集合，R 是数据元素之间的关系集合。

例如，光谱颜色的数据结构可以表示为

$$D=\{红,橙,黄,绿,蓝,紫\}$$

$$R=\{(红,橙),(橙,黄),(黄,绿),(绿,蓝),(蓝,紫)\}$$

而学校成员的数据结构可以表示为

$$D=\{学校,院,处,研究机构,系,部,实验中心,科\}$$

$$R=\{(学校,院),(学校,处),(学校,研究机构),(院,系),(院,部),(院,实验中心),(处,科)\}$$

根据数据元素之间的不同特性，通常有下列四类基本的结构（见图 3.3）。

① 集合结构中的数据元素之间除同属于一个集合的关系以外，无任何其他关系。

② 线性结构中的数据元素之间存在一对一的线性关系。

③ 树状结构中的数据元素之间存在一对多的层次关系。

④ 图状结构或网状结构中的数据元素之间存在多对多的任意关系。

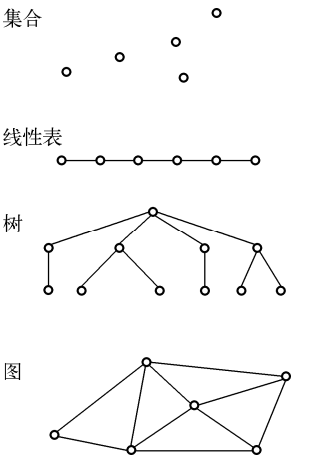


图 3.3　四类基本的结构

由于集合的关系非常松散，因此可以用其他的结构来代替它。故数据的逻辑结构可概括为

逻辑结构
- 线性结构——线性表、栈、队列、字符、串、数据、广义表
- 非线性结构——树、图

线性结构的逻辑特征是，除第一个节点和最后一个节点外，其他所有节点都有且只有一个直接前驱和一个直接后继节点。非线性结构的逻辑特征是，一个节点可能有多个直接前驱节点和多个直接后继节点。

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，或称为数据的物理结构。即数据在存储时，不仅要存储数据元素的信息，还要存储数据元素之间的前后件关系的信息。

通常的数据存储结构有顺序、链接、索引等。

例如，在城市交通的例子中，研究的是如何在计算机中表示一个地点、如何在计算机中表示两个地点之间存在一条公共汽车线路，以及该线路有多长等问题。

又如，在电话号码查询例子中，研究的是电话号码资料信息在计算机中是如何存储的，

以及如何表示资料的分类、如何表示排好顺序的电话号码之间的先后次序关系等问题。

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。数据必须在计算机中存储，数据的存储结构研究的是数据元素和数据元素之间的关系如何在计算机中表示，是逻辑数据的存储映象，它是面向计算机的。

实现从数据的逻辑结构到存储器的映象有多种不同的方式。通常，数据在存储器中的存储有如下四种基本的映象方法。

① 顺序存储结构：把逻辑上相邻的节点存储在物理位置相邻的存储单元中，节点间的逻辑关系由存储单元的邻接关系来体现。由此得到的存储形式称为顺序存储结构，它主要用于存储线性数据结构，非线性的数据结构也可以通过某种线性化的方法来实现顺序存储。

② 链式存储结构：链式存储结构把逻辑上相邻的两个数据元素存储在物理上不一定相邻的存储单元中，节点间的逻辑关系是由附加的指针字段表示的。链式存储结构的特点就是将存放每个数据元素的节点分为两部分：一部分存放数据元素（称为数据域）；另一部分存放指示存储地址的指针（称为指针域），借助指针表示数据元素之间的关系。

③ 索引存储结构：在存储节点信息的同时，建立附加的索引表。索引表中的每项称为一个索引项，索引项的一般形式是：关键字，地址。关键字是能唯一标志一个节点的数据项，地址则指向节点信息的存储位置。

④ 散列存储结构：根据节点的关键字值计算出该节点的存储地址，即在数据元素与其在存储器中的存储位置之间建立一个映象关系 F，根据关键字值和映象关系 F 就可以得到它的存储地址，即 $D=F(E)$，E 是要存放的数据元素的关键字值，D 是该数据元素的存储地址。哈希表是散列存储结构中最常用的一种。

3．数据的运算

数据的运算是定义在数据逻辑结构上的操作，每种数据结构都有一个运算的集合。常用的运算有检索、插入、删除、更新、排序等。运算的具体实现要在对应的存储结构上进行。例如，在电话号码查询问题中，就要设计如何插入一个新的电话号码信息，如何删除一个作废的电话号码信息，如何对电话号码进行快速整理排序，如何高效、快速地查找资料等算法。在城市交通问题中，就要设计求两个地点之间最短线路的算法，要能够判断从城市的任意一个地点出发乘坐公共汽车是否可以到达该城市的任何地点。

3.1.3 线性结构与非线性结构

如果在数据结构中一个数据元素都没有，则该数据结构为空数据结构；如果在空数据结构中插入一个新的数据元素，则数据结构变为非空数据结构；如果将数据结构中的所有数据元素全部删除，则该数据结构变成空数据结构。

1．线性结构

如果一个非空的数据结构满足如下条件，则该数据结构为线性结构。

- 有且只有一个根节点。
- 每个节点最多只有一个直接前驱，也最多只有一个直接后继。

线性结构的特点是：在数据元素的非空有限集合中，存在唯一的首数据元素和唯一的尾数据元素，首数据元素无直接前驱，尾数据元素无直接后继，集合中其他的每个数据元素均有唯一的直接前驱和唯一的直接后继。

注意：在线性结构表中插入或删除数据元素后，该线性表仍然应满足线性结构的条件。

线性结构的主要特征为，各个数据元素之间有明确的、唯一的“先后”顺序。线性结构包括线性表、栈和队列等。在日常生活中具有线性结构的实例非常多，例如，排队购物，队列中的每个人之间都有一个明确的先后次序关系。

2．非线性结构

如果一个数据结构不满足线性结构的条件，则为非线性结构。非线性结构包括树状结构和图状结构。

树状结构的主要特征是节点之间存在一种层次的关系，每个节点对应下一层的多个节点，也就是说，数据元素之间是“一对多”的关系。例如，学校下面有好几个院/系，每个院/系下面有好多个班，每个班下面有好多个学生。学校—院/系—班—学生，每层之间都是一种一对多的关系，这就是一个典型的树状结构。

而在图状结构中，任何两个节点之间都可能存在联系，数据元素之间存在多对多的关系。典型的图状结构就是城市交通。如果城市中有单行线路，则从城市中的一个地点 A 出发，可以到达 N 个不同的地点，从城市的 M 个不同的地点出发又可以到达地点 A。城市交通就是一个典型的“多对多”的图状结构（见图 3.2）。

3.2　线性结构的存储与处理

3.2.1　线性表的存储与处理

1．基本概念

图 3.4　线性表中数据元素之间的关系

线性表是 n 个类型相同的数据元素的有限序列，数据元素之间是一对一的关系，即每个数据元素最多有一个直接前驱和一个直接后继，如图 3.4 所示。这里的数据元素是指广义的数据元素，并不仅仅是指一个数据，如矩阵、学生记录表等。

非空线性表的结构特征如下。

- 有且只有一个根节点，它无直接前驱。
- 有且只有一个终端节点，它无直接后继。
- 除根节点和终端节点以外，所有的节点有且只有一个直接前驱和一个直接后继。线性表中节点的个数称为节点的长度 n。当 n=0 时，称为空表。

例如，英文字母表（A、B、…、Z）就是一个简单的线性表，表中的每个英文字母是一个数据元素，每个数据元素之间存在唯一的顺序关系，如在英文字母表中字母 B 的前面是字母 A，而字母 B 的后面是字母 C。在较为复杂的线性表中，数据元素可由若干数据项组成，例如，在学生成绩表中，每个学生及其各科成绩是一个数据元素，它由学号、姓名、各科成绩及平均成绩等数据项组成，常称为一个记录，含有大量记录的线性表称为文件。数据对象是性质相同的数据元素集合。

如表 3.1 所示的会员档案登记表，每个会员的相关信息由编号、姓名、性别、出生年月、等级和积分 6 个数据项组成，它是文件的一个记录（数据元素）。

综上所述，线性表的定义如下。

线性表是由 $n(n\geqslant 0)$个类型相同的数据元素组成的有限序列，记作$(a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$。这里的数据元素 $a_i(1\leqslant i\leqslant n)$只是一个抽象的符号，其具体含义在不同情况下可以不同，它既可以是原子类型，也可以是结构类型，但同一线性表中的数据元素必须属于同一数据对象。此外，

线性表中相邻数据元素之间存在着序偶关系，即对于非空的线性表（$a_1,a_2,\cdots a_{i-1},a_i,a_{i+1},\cdots,a_n$）而言，表中 a_{i-1} 领先于 a_i，称 a_{i-1} 是 a_i 的直接前驱，而称 a_i 是 a_{i-1} 的直接后继。除第一个数据元素 a_1 外，每个数据元素 a_i 有且仅有一个被称为其直接前驱的节点 a_{i-1}，除最后一个数据元素 a_n 外，每个数据元素 a_i 有且仅有一个被称为其直接后继的节点 a_{i+1}。

线性表的特点可概括为如下几点。

- 同一性：线性表由同类数据元素组成，每个 a_i 必须属于同一数据对象。
- 有穷性：线性表由有限个数据元素组成，表长度就是表中数据元素的个数。
- 有序性：线性表中相邻数据元素之间存在着序偶关系$<a_i,a_{i+1}>$。

由此可以看出，线性表是一种最简单的数据结构，因为数据元素之间是由一个前驱和一个后继的直观有序的关系确定的；线性表又是一种最常见的数据结构，因为矩阵、数组、字符串、栈、队列等都符合线性条件。

2．顺序存储结构

线性表的顺序存储结构是指用一组地址连续的存储单元依次存储线性表中的各个数据元素，使得线性表中在逻辑结构上相邻的数据元素存储在相邻的物理存储单元中，即通过数据元素物理存储的相邻关系来反映数据元素之间逻辑上的相邻关系。采用顺序存储结构的线性表通常称为顺序表。

顺序存储结构的特点如下。

- 线性表中所有的数据元素所占的存储空间是连续的。
- 线性表中各数据元素在存储空间中是按照逻辑顺序依次存放的。

假设线性表中有 n 个数据元素，每个数据元素占 k 个单元，第一个数据元素的地址为 $\mathrm{loc}(a_1)$，则可以通过如下公式计算出第 i 个数据元素的地址 $\mathrm{loc}(a_i)$：

$$\mathrm{loc}(a_i)=\mathrm{loc}(a_1)+(i-1)\times k^{*}$$

其中，$\mathrm{loc}(a_1)$称为基址。

图 3.5 所示为线性表的顺序存储结构示意。从图 3.5 中可看出，在线性表的顺序存储结构中，其前、后两个数据元素在存储空间中是紧邻的，前数据元素一定存储在后数据元素的前面，且每个节点 a_i 的存储地址是该节点在表中的逻辑位置 i 的线性函数。只要知道线性表中第一个数据元素的存储地址（基址）和表中每个数据元素所占存储单元的多少，就可以计算出线性表中任意一个数据元素的存储地址，从而实现对顺序表中数据元素的随机存取。

存储地址	内存空间状态	逻辑地址
$\mathrm{loc}(a_1)$	a_1	1
$\mathrm{loc}(a_1)+(2-1)\times k$	a_2	2
…	…	…
$\mathrm{loc}(a_1)+(i-1)\times k$	a_i	i
…	…	…
$\mathrm{loc}(a_1)+(n-1)\times k$	a_n	n

图 3.5　线性表的顺序存储结构示意

顺序存储结构可以借助高级程序设计语言中的数组来表示，一维数组的下标与数据元素在线性表中的序号相对应。在用一维数组存放线性表时，该一维数组的长度通常要定义得比线性表的实际长度长一些，以便对线性表进行各种运算，特别是插入运算。在一般情况下，如果线性表的长度在处理过程中是动态变化的，则在开辟线性表的存储空间时要考虑线性表在动态变化过程中可能达到的最大长度。如果在开始时所开辟的存储空间太小，则在线性表动态增长时可能会出现由于存储空间不够而无法再插入新的数据元素的问题；但如果在开始时所开辟的存储空间太大，而实际上又用不到那么大的存储空间，则会造成存储空间的浪费。在实际应用中，读者可以根据线性表动态变化过程中的一般规模来决定要开辟的存储空间大小。

3．线性表的基本操作

（1）线性表的查找运算

线性表有按序号查找和按内容查找两种基本的查找运算。

① 按序号查找：其结果是线性表中第 i 个数据元素。

② 按内容查找：要求查找线性表中与给定值相等的数据元素，其结果是：若在表中找到与给定值相等的数据元素，则返回该数据元素在表中的序号；若找不到，则返回一个“空序号”，表示无此数据元素。

查找运算可采用顺序查找法实现，即从第一个数据元素开始，依次将表中数据元素与给定值相比较，若相等，则查找成功，返回该数据元素在表中的序号；若给定值与表中的所有数据元素都不相等，则查找失败，可返回“无此数据元素”信息的一个值（具体可由所用语言或实际需要决定其值的形式）。

（2）线性表的插入运算

线性表的插入运算是指在表的第 i（$1\leqslant i\leqslant n+1$）个位置，插入一个新数据元素 e，使长度为 n 的线性表（$e_1,e_2,\cdots,e_{i-1},e_i,\cdots,e_n$）变成长度为 $n+1$ 的线性表（$e_1,e_2,\cdots,e_{i-1},\boldsymbol{e},e_i,\cdots,e_n$）。

用顺序表作为线性表的存储结构时，由于节点的物理顺序必须和节点的逻辑顺序保持一致，因此必须将原表中（$n,n-1,\cdots,i$）位置上的节点，依次后移到（$n+1,n,\cdots,i+1$）位置上，空出第 i 个位置，然后在该位置上插入新节点 e。当 $i=n+1$ 时，表示在线性表的末尾插入节点，所以无须移动节点，直接将 e 插入表的末尾即可。

【例 3.3】 已知一个线性表（4,7,15,28,30,32,42,51,63），现需在第 4 个数据元素之前插入一个数据元素“21”。

方法：要将数据元素“21”插入线性表中，首先需将第 4 个位置到第 9 个位置的数据元素依次向后移动一个位置，然后将数据元素“21”插入第 4 个位置，如图 3.6 所示（注意区分数据元素与数据元素的序号）。

注意：在定义线性表时，一定要定义足够的空间，否则，将不允许插入新数据元素。另外，在找到插入位置后，需要移动从插入位置开始的所有数据元素，从最后一个数据元素开始顺序后移。

显然，在线性表采用顺序存储结构时，如果插入运算在线性表的末尾进行，即在第 n 个数据元素之后（可以认为是在第 $n+1$ 个数据元素之前）插入新数据元素时，则只要在表的末尾增加一个数据元素即可，不需要移动表中的数据元素；如果要在线性表的第 1 个数据元素之前插入一个新数据元素，则需要移动表中所有的数据元素。在一般情况下，如果插入运算在第 i（$1\leqslant i\leqslant n$）个数据元素之前进行，则原来第 i 个数据元素之后（包括第 i 个数据元素）的所有数据元素都必须移动。在平均情况下，要在线性表中插入 1 个新数据元素，需要移动表中一半的数据元素。因此，在线性表采用顺序存储结构的情况下，要插入 1 个新数据元素，其效率是很低的，特别是在线性表比较大的情况下更为突出，数据元素的移动需要消耗较多的处理时间。

（3）线性表的删除运算

线性表的删除运算是指将表的第 i（$1\leqslant i\leqslant n$）个数据元素删除，使长度为 n 的线性表（$e_1,e_2,\cdots,e_{i-1},e_i,e_{i+1},\cdots,e_n$）变成长度为 $n-1$ 的线性表（$e_1,e_2,\cdots,e_{i-1},e_{i+1},\cdots,e_n$）。与插入运算类似，在顺序表上实现删除运算也必须移动节点，这样才能反映出节点间逻辑关系的变化。

【例 3.4】 将线性表（4,7,15,28,30,32,42,51,63）中的第 4 个数据元素删除。

方法：将第 5 个位置到第 9 个位置的数据元素依次向前移动一个位置（见图 3.7），即可完成任务。

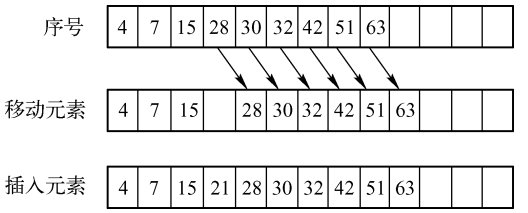


图 3.6　在线性表中插入数据元素

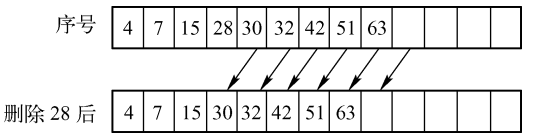


图 3.7　删除线性表中的数据元素

显然，在线性表采用顺序存储结构时，如果删除运算在线性表的末尾进行，即删除第 n 个数据元素，则不需要移动表中的数据元素；如果要删除线性表中的第 1 个数据元素，则需要移动表中所有的数据元素。在平均情况下，要在线性表中删除 1 个数据元素，需要移动表中一半的数据元素。

由线性表在顺序存储结构下的插入与删除运算可以看出，顺序存储结构对于小线性表或者其中数据元素不常变动的线性表来说是合适的，因为顺序存储的结构比较简单。但这种顺序存储的方式对于其中数据元素经常需要变动的大线性表就不太合适了，因为插入与删除运算的效率比较低。

4．链式存储结构

解决静态内存分配不足的办法就是使用动态内存分配。所谓动态内存分配，是指在程序执行的过程中动态地分配或者回收存储空间。动态内存分配不像静态内存分配那样需要预先分配存储空间，而是由系统根据程序的需要即时分配的，且分配的大小就是程序要求的大小。

每个数据节点对应一个存储单元，该存储单元称为存储节点，简称节点。

链式存储方式要求每个节点由两部分组成：一部分用于存放数据元素的值，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），如图 3.8 所示。

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

5．线性链表

线性链表用一组任意的存储单元来存放线性表中的节点，这组存储单元可以是连续的，也可以是非连续的，甚至是零散分布在内存的任意位置上的。因此，链表中节点的逻辑顺序和物理顺序不一定相同。为了正确地表示节点间的逻辑关系，必须在存储线性表中每个数据元素值的同时，存储指示其后继节点的地址（或位置）信息，这两部分信息组成的存储映象称为节点，如图 3.8 所示。

（1）单链表

在单链表中用一个专门的指针 HEAD 指向线性链表中第一个数据元素的节点（即存放第一个数据元素的地址）。线性链表中的最后一个数据元素没有后件，因此，线性链表中的最后一个节点的指针域为空（用 NULL 或 0 表示），表示链终结。

在线性链表中，各数据元素的存储序号是不连续的，数据元素间的前后件关系与位置关系也是不一致的。在线性链表中，前后件的关系依靠各节点的指针来指示，指向表的第一个数据元素的指针 HEAD 称为头指针，当 HEAD=NULL 时，表示该链表为空。

例如，线性表（A,B,C,D,E,F,G,H）的单链表存储结构如图 3.9 所示，整个链表的存取需从头指针开始进行，依次顺着每个节点的指针域找到线性表的各个数据元素。

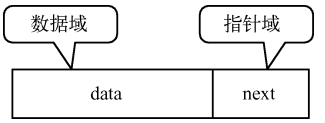


图 3.8　节点

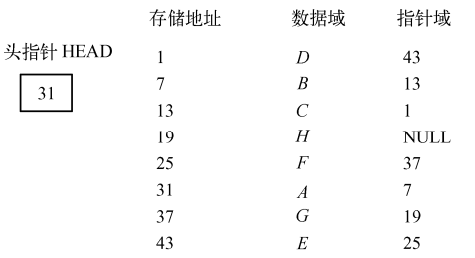


图 3.9　线性表（*A*,*B*,*C*,*D*,*E*,*F*,*G*,*H*）的单链表存储结构

在一般情况下，我们在使用链表时，只关心链表中节点间的逻辑顺序，并不关心每个节点的实际存储位置，因此通常用箭头来表示链域中的指针，于是链表就可以更直观地表示成用箭头连接起来的节点序列。图 3.9 中的单链表存储结构可表示为如图 3.10 所示的逻辑状态。

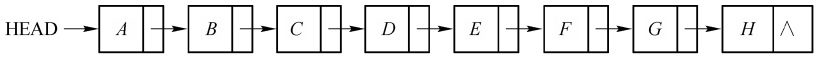


图 3.10　单链表存储结构的逻辑状态

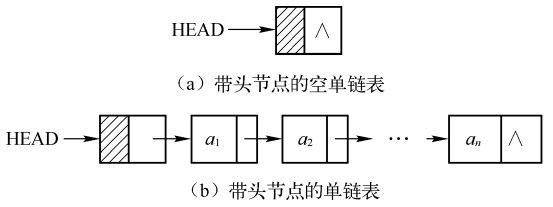


图 3.11　带头节点的单链表

有时为了操作方便，还可以在单链表的第一个节点之前附设一个头节点，头节点的数据域可以存储一些关于线性表的长度的附加信息，也可以什么都不存储；而头节点的指针域存储指向第一个节点的指针（即第一个节点的存储位置）。此时带头节点的单链表（见图 3.11）的头指针就不再指向表中的第一个节点而是指向头节点。如果线性表为空表，则头节点的指针域为空。

单链表可以从表头开始，沿着各节点的指针向后扫描链表中的所有节点，而不能从中间或表尾节点向前扫描位于该节点之前的节点。

（2）循环单链表

循环单链表是单链表的另一种形式，它是一个首尾相接的链表。其特点是将单链表最后一个节点的指针域由 NULL 改为指向头节点或线性表中的第一个节点，就得到了单链形式的循环链表，并称为循环单链表，如图 3.12 所示。

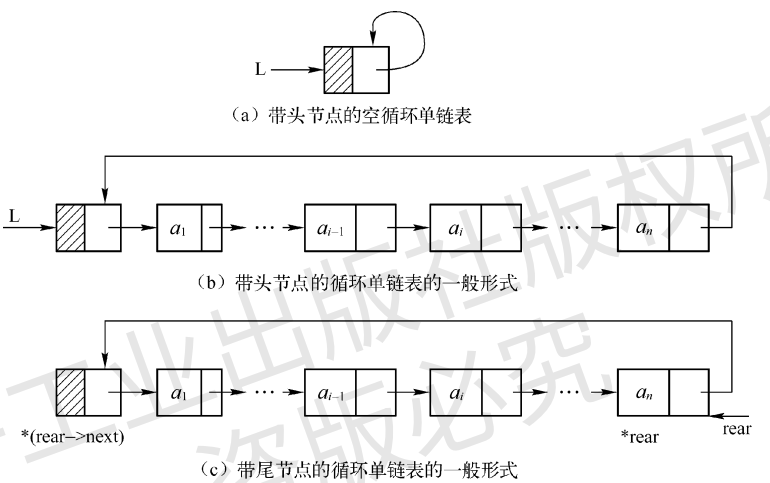


图 3.12　循环单链表

带头节点的循环单链表的各种操作的实现算法与带头节点的单链表的实现算法类似。在循环单链表中附设尾指针有时比附设头指针会使操作变得更简单。例如，在用头指针表示的循环单链表中要找到终端节点 a_n，需要从头指针开始遍历整个链表，如果用尾指针 rear 来表示循环单链表，则查找头节点和终端节点都很方便。因此，在实用中多采用尾指针表示循环单链表的形式。

（3）双向链表

单链表和循环单链表结构的缺点是不能任意地对链表中的节点按不同的方向进行扫描。通常，对链表中的节点设置两个指针域，一个为指向前件的指针域，称为前指针域（prior）；另一个为指向后件的指针域，称为后指针域（next），如图 3.13 所示。用这种节点形成的链表就是双向链表。双向链表也可以有循环表，称为双向循环链表，如图 3.14 所示。

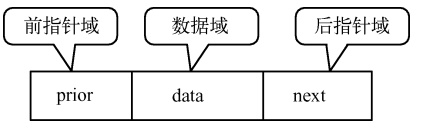


图 3.13　双向链表的节点

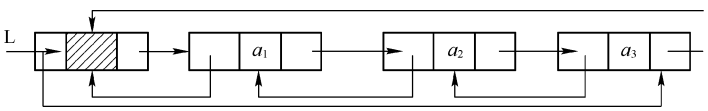


图 3.14　双向循环链表

6．线性链表的基本运算

（1）线性链表的查找运算

链表存储结构不是一种随机存取结构，如果要查找单链表中的一个节点，则必须从头指针出发，沿节点的指针域逐个往后查找，直到找到要查找的节点为止。

① 按序号查找。

在单链表中，由于每个节点的存储位置都放在其前一节点的 next 域中，所以即使知道被访问节点的序号 i，也不能像顺序表那样直接按序号 i 访问一维数组中的相应数据元素，实现随机存取，而只能从单链表的头指针出发，顺着指针域 next 逐个往下搜索，直至搜索到第 i 个节点为止。

算法描述：设带头节点的单链表的长度为 n，要查找表中第 i 个节点，则需要从单链表的头指针 L 出发，从头节点（L->next）开始顺着指针域扫描，用指针 p 指向当前扫描到的节点，初值指向头节点，设 j 为计数器，累计当前扫描过的节点数（初值为 0），当 $j = i$ 时，指针 p 所指的节点就是要找的第 i 个节点。

② 按值查找。

按值查找是指在单链表中查找是否存在一个节点的数据域的值等于要找的值，若有的话，则返回首次找到的节点的存储位置，否则返回找不到该值的信息提示。

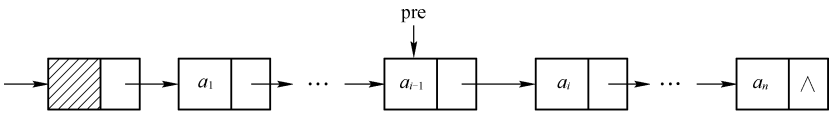
算法描述：从单链表的头指针指向的头节点出发，顺着指针域逐个将节点的数据域的值和给定的值进行比较。

（2）线性链表的插入运算

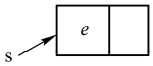
线性链表的插入运算是指在基于链式存储结构的线性表中插入一个新数据元素（节点）。

如果要在带头节点的单链表 L 中第 i 个节点之前插入一个节点 e，则需要先在单链表中找到第 i–1 个节点并由指针 pre 指示，然后申请一个新的节点并由指针 s 指示，其数据域的值为 e，并修改第 i–1 个节点的指针，使其指向指针 s 指向的节点，然后使该节点的指针域指向第 i 个节点。在单链表第 i 个节点前插入一个节点的过程如图 3.15 所示。

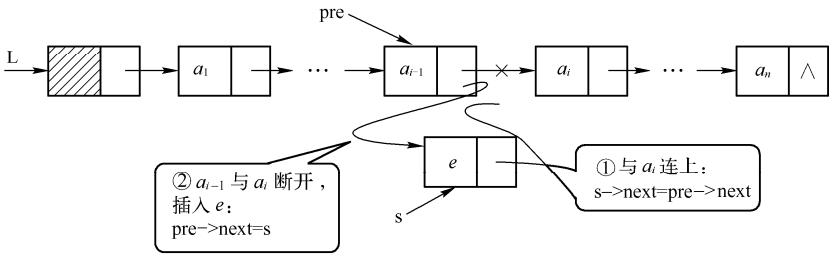
在线性链表的插入操作中，新节点是来自可利用栈的，因此不会造成线性表的溢出。同样，由于可利用栈可被多个线性表利用，因此，不会造成存储空间的浪费，大家动态地共同使用存储空间。另外，线性链表在插入过程中不会发生数据元素移动的现象，只需改变有关节点的指针即可，从而提高了插入的效率。



（a）寻找第 i–1 个节点



（b）申请新的节点



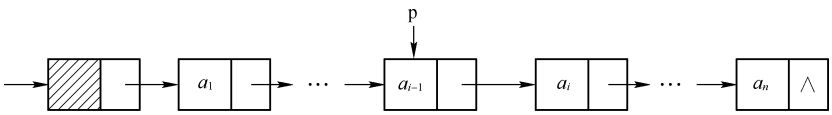
（c）插入

图 3.15　在单链表中第 i 个节点前插入一个节点的过程

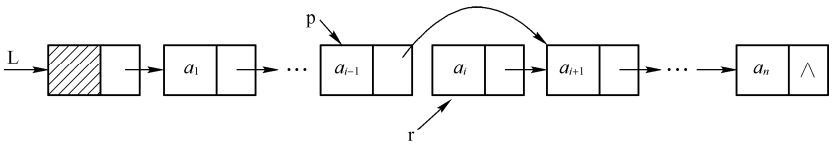
（3）线性链表的删除运算

线性链表的删除运算是指在基于链式存储结构的线性表中删除指定数据元素的节点。

如果要在带头节点的单链表 L 中删除第 i 个节点，则首先要找到第 i–1 个节点并使指针 p 指向第 i–1 个节点，如图 3.16（a）所示；然后将第 i–1 个节点的指针域的值赋给指针 r；接下来将第 i 个节点的指针域的值赋给指针 p 指向的第 i–1 个节点的指针域，这样就使第 i–1 个节点与第 i+1 个节点直接连接起来；最后用指针 r 将第 i 个节点释放，如图 3.16（b）所示，即归还给可利用栈。



（a）查找第i–1个节点并用指针p指向第i–1个节点



（b）用指针r将第i个节点释放

图 3.16　线性链表的删除过程

从线性链表的删除过程可以看出，在线性链表中删除一个数据元素后，不需要移动表中的数据元素，只需改变被删除数据元素所在节点的前一个节点的指针域即可。另外，由于可利用栈是用于收集计算机中所有的空闲节点的，因此，当从线性链表中删除一个数据元素后，该数据元素的存储节点就变为空闲，应将该空闲节点送回可利用栈。

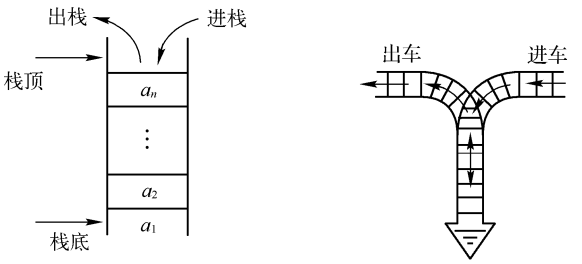
3.2.2　先进后出结构的存储与处理

1．栈

栈就是将线性表的插入和删除运算限制为仅在表的一端进行。通常将表中允许进行插入、

删除操作的一端称为栈顶（Top），因此栈顶的当前位置是动态变化的，它由一个称为栈顶指针的位置指示器指示。同时，表的另一端称为栈底（Bottom）。当栈中没有数据元素时称为空栈。栈的插入操作称为进栈或入栈，删除操作称为出栈或退栈。

根据上述定义，每次进栈的数据元素都被放在原栈顶数据元素之上而成为新的栈顶，而每次出栈的总是当前栈中“最新”的数据元素，即最后进栈的数据元素。在如图 3.17（a）所示的栈中，数据元素是以 $a_1,a_2,\cdots,a_n$ 的顺序进栈的，而退栈的顺序却是 $a_n,\cdots,a_2,a_1$。栈的修改是按照后进先出的原则进行的。因此，栈又称为先进后出的线性表，简称 FILO（First In Last Out）表。在日常生活中也可以见到很多“先进后出”的例子，如手枪子弹夹中的子弹，子弹的装入与子弹的弹出均在弹夹的最上端进行，先装入的子弹后发出，而后装入的子弹先发出。又如铁路调度站，如图 3.17（b）所示，都是栈结构的实际应用。



（a）栈的示意结构　　（b）铁路调度站的表示

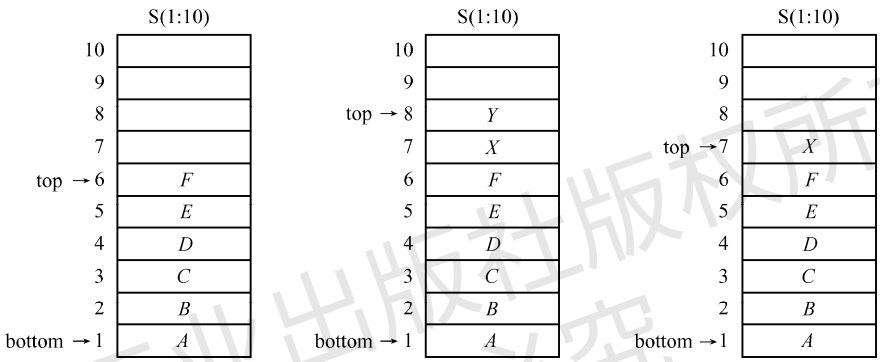
图 3.17　栈

栈的基本操作除了进栈（栈顶插入）、出栈（删除栈顶），还有建立栈（栈的初始化）、判空、判满及取栈顶数据元素等。

栈作为一种特殊的线性表，在计算机中有顺序和链式两种基本的存储结构。一般称顺序存储的栈为顺序栈，称链式存储的栈为链栈。

2．栈的顺序存储及其运算

在程序设计中，顺序栈与一般的线性表一样，用一维数组 S(1:*m*)作为栈的顺序存储空间，其中 *m* 为栈的最大容量。通常，栈底指针指向栈空间的低地址一端（即数组的起始地址这一端）。图 3.18（a）所示是容量为 10 的栈顺序存储空间，栈中已有 6 个数据元素。图 3.18（b）与图 3.18（c）所示分别为入栈与退栈后的状态。



（a）容量为10且有6个元素的栈　（b）插入 X 与 Y 后的栈　（c）退出一个元素后的栈

图 3.18　栈在顺序存储结构下的运算

在栈的顺序存储空间 S(1:*m*)中，S(bottom)通常为栈底数据元素（在栈非空的情况下），S(top)

为栈顶数据元素。top=0 表示栈空；top=m 表示栈满。

栈的基本运算有三种：入栈、退栈与读栈顶数据元素。下面分别介绍在顺序存储结构下栈的三种运算。

（1）入栈运算

入栈运算是指在栈顶位置插入一个新数据元素。其基本操作分为两步：首先将栈顶指针进一（即 top 加 1），然后将新数据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，不可以再进行入栈操作，这种情况称为栈“上溢”错误。

（2）退栈运算

退栈运算是指取出栈顶数据元素并赋给一个指定的变量。其基本操作分为两步：首先将栈顶数据元素（栈顶指针指向的数据元素）赋给一个指定的变量，然后将栈顶指针退一（即 top 减 1）。

当栈顶指针为 0 时，说明栈空，不可以进行退栈操作，这种情况称为栈“下溢”错误。

（3）读栈顶数据元素运算

读栈顶数据元素运算是指将栈顶数据元素赋给一个指定的变量。必须注意，这个运算不删除栈顶数据元素，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，读不到栈顶数据元素。

3.2.3 先进先出结构的存储与处理

1．队列

队列是另一种限定性的线性表，它只允许在表的一端插入数据元素，而在另一端删除数据元素，所以队列具有先进先出（FIFO）的特性。这与我们日常生活中的排队是一致的，最早进入队列的人最早离开，新来的人总是加入队尾。在队列中，允许插入的一端称为队尾（通常用一个队尾指针 rear 指向队尾）；允许删除的一端则称为队头（通常用一个队头指针 front 指向队头）。假设队列为 $q=(a_1,a_2,\cdots,a_n)$，那么 a_1 就是队头数据元素，a_n 则是队尾数据元素。队列中的数据元素是按照 $a_1,a_2,\cdots,a_n$ 的顺序进入的，退出队列也必须按照同样的次序依次出队，也就是说，只有在 $a_1,a_2,\cdots,a_{n-1}$ 都退出队列之后，a_n 才能退出队列。

队列在程序设计中也经常出现。一个最典型的例子就是操作系统中的作业排队。在允许多道程序运行的计算机系统中，有几个作业同时运行。如果运行的结果都需要通过通道输出，那就要按请求输出的先后次序排队。凡是请求输出的作业都从队尾进入队列。

在队列中，队尾指针 rear 与队头指针 front 共同反映了队列中数据元素动态变化的情况。图 3.19 所示是具有 8 个数据元素的队列示意。

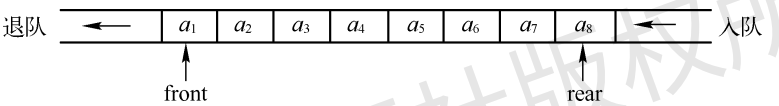
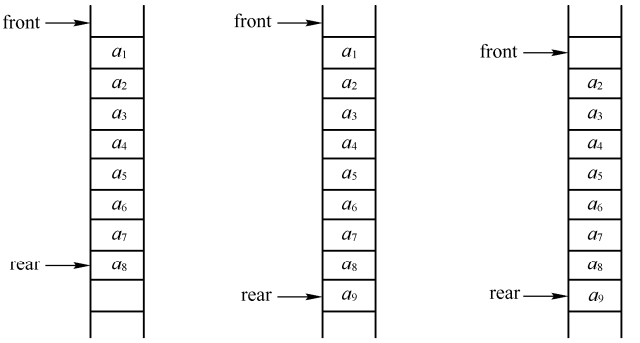


图 3.19　具有 8 个数据元素的队列示意

向队列的队尾插入一个数据元素称为入队运算，从队列的队头删除一个数据元素称为退队运算。

图 3.20 所示是在队列（$a_1,a_2,\cdots,a_8$）中进行插入与删除操作的示意。由图 3.20 可以看出，在队列的末尾插入一个数据元素 a_9（入队运算），只涉及队尾指针 rear 的变化，而要删除队列中的队头数据元素 a_1（退队运算），只涉及队头指针 front 的变化。



（a）原队列　　（b）插入 a_9 元素后的队列　　（c）删除 a_1 元素后的队列

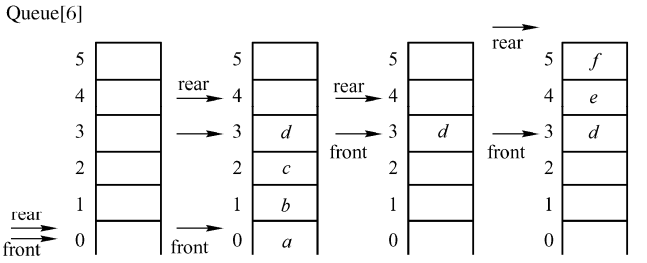
图 3.20　在队列（a_1,a_2,…,a_8）中进行插入与删除操作的示意

与此类似，在程序设计语言中，用一维数组作为队列的顺序存储空间。

队列在计算机中有顺序和链式两种基本的存储结构。

2．循环队列及其运算

循环队列是队列的一种顺序表示和实现方法。与顺序栈类似，在队列的顺序存储结构中，用一组地址连续的存储单元依次存放从队头到队尾的数据元素，如一维数组 Queue[MAXSIZE]。此外，由于队列中队头和队尾的位置都是动态变化的，因此需要附设两个指针 front 和 rear，分别指示队头数据元素和队尾数据元素在数组中的位置。在初始化队列时，令 front=rear=0，如图 3.21（a）所示；在入队时，直接将新数据元素送入尾指针 rear 所指的单元，然后尾指针增 1；在出队时，直接取出队头指针 front 所指的数据元素，然后头指针增 1。显然，在非空顺序队列中，队头指针始终指向当前的队头数据元素，而队尾指针始终指向真正队尾数据元素后面的单元。当 rear=MAXSIZE 时，认为队满。但此时不一定是真的队满，因为随着部分数据元素的出队，数组前面会出现一些空单元，如图 3.21（d）所示。由于只能在队尾入队，使得上述空单元无法使用，这种现象称为假溢出，真正队满的条件是 rear－front=MAXSIZE。

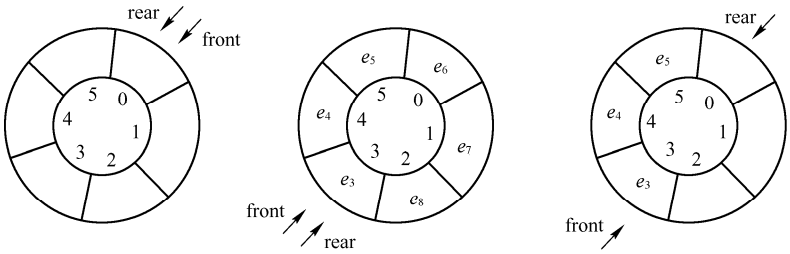


（a）空队列　　（b）a,b,c,d 入队　　（c）a,b,c 出队　　（d）e,f 入队

图 3.21　队列的基本操作

为了解决假溢出现象并使得队列空间得到充分利用，一个较巧妙的办法是将顺序队列的数组看成一个环状的空间，即规定最后一个单元的后继为第一个单元，可形象地称之为循环队列。假设队列数组为 Queue[MAXSIZE]，当 rear+1=MAXSIZE 时，令 rear=0，即可求得最后一个单元 Queue[MAXSIZE-1]的后继：Queue[0]。更简便的办法是通过数学中的取模（求余）运算来实现：rear=(rear+1)mod MAXSIZE，显然，当 rear+1=MAXSIZE 时，rear=0，同样可求得最后一个单元 Queue[MAXSIZE-1]的后继：Queue[0]。所以，借助于取模（求余）运算，可以自动实现队尾指针、队头指针的循环变化。在进队操作时，队尾指针的变化是：rear=(rear+1)mod

MAXSIZE ；而在出队操作时，队头指针的变化是：front=(front+1)mod MAXSIZE。图 3.22 所示为循环队列的几种情况。



（a）空队列　　（b）队列满　　（c）一般情况

图 3.22　循环队列的几种情况

与一般的非空顺序队列相同，在非空循环队列中，队头指针始终指向当前的队头数据元素，而队尾指针始终指向真正队尾数据元素后面的单元。在如图 3.22（c）所示的循环队列中，队头数据元素是 e_3，队尾数据元素是 e_5，当 e_6、e_7 和 e_8 相继入队后，队列空间均被占满，如图 3.22（b）所示，此时队尾指针追上队头指针，所以有：front =rear。反之，若 e_3、e_4 和 e_5 相继从图 3.22（c）的队列中删除，则得到空队列，如图 3.22（a）所示，此时队头指针追上队尾指针，所以也存在关系式：front=rear。可见，只凭 front=rear 无法判别队列的状态是“空”还是“满”。对于这个问题，可有两种处理方法。一种方法是少用一个数据元素空间，即当队尾指针所指向的空单元的后继单元是队头数据元素所在的单元时，则停止入队，这样一来，队尾指针永远追不上队头指针，所以队满时不会有 front=rear，现在队列“满”的条件为(rear+1)mod MAXSIZE=front，判断队空的条件不变，仍为 rear=front。另一种方法是增设一个标志量，以区别队列是“空”还是“满”。

3.3　数据的查找与排序

3.3.1　查找

查找是指在一个给定的数据结构中查找某个指定的数据元素。

1．顺序查找

顺序查找又称顺序搜索。一般是指在线性表中查找指定的数据元素。

基本操作方法如下。

从线性表的第一个数据元素开始，与被查找数据元素进行比较，若相等则查找成功，否则继续向后查找。如果所有的数据元素均查找完毕而未找到相等的数据元素，则表示该数据元素在指定的线性表中不存在。

顺序查找的最好情况是要查找的数据元素为线性表的第一个数据元素。如果要查找的数据元素在线性表的最后或根本不存在，则需要搜索线性表的所有数据元素，这是最差情况。

对于线性表而言，顺序查找的效率很低。但以下的线性表只能采用顺序查找的方法。

① 线性表是无序表，即表中的数据元素不是按照大小顺序进行排列的，这类线性表无论它的存储方式是顺序存储还是链式存储，都只能按顺序查找方法进行查找。

② 即使是有序线性表，如果采用链式存储方式，也只能采用顺序查找的方法。

【例 3.5】 现有线性表（7,2,1,5,9,4），整个线性表的长度为 6。要在表中查找 6，查找过程如表 3.2 所示。

表 3.2　线性表的查找过程

查找计次	操　作
1	将 6 与表中的第 1 个数据元素 7 进行比较，不相等，继续查找
2	将 6 与表中的第 2 个数据元素 2 进行比较，不相等，继续查找
3	将 6 与表中的第 3 个数据元素 1 进行比较，不相等，继续查找
4	将 6 与表中的第 4 个数据元素 5 进行比较，不相等，继续查找
5	将 6 与表中的第 5 个数据元素 9 进行比较，不相等，继续查找
6	将 6 与表中的第 6 个数据元素 4 进行比较，不相等，继续查找
7	超出线性表的长度，查找结束，该表中不存在要查找的数据元素

2．二分查找

二分查找只适用于顺序存储的线性表，即有序表。此处所述的有序表是指线性表中的数据元素按值非递减排列（即由小到大排列，但允许相邻数据元素的值相等）。

二分查找的方法如下。

将要查找的数据元素与有序序列的中间数据元素进行比较。

- 如果要查找的数据元素的值比中间数据元素的值大，则继续在线性表的后半部分（中间项以后的部分）进行查找。
- 如果要查找的数据元素的值比中间数据元素的值小，则继续在线性表的前半部分（中间项以前的部分）进行查找。

这个查找操作一直按相同的顺序进行下去，直到查找成功或子表长度为 0（说明线性表中没有要查找的数据元素）为止。

线性表的二分查找的条件是这个线性表必须是顺序存储的。它的查找效率比顺序查找要高得多，它的最坏情况的查找次数是 $\log_2 n$ 次，而顺序查找的最坏情况的查找次数是 n 次。

当然，二分查找的方法也支持基于顺序存储结构的递减序列的线性表。

【例 3.6】 有（1,2,4,5,7,9）非递减有序线性表，要查找数据元素 6。

查找的方法如下。

① 序列长度为 n=6，中间数据元素的序号 m=[(n+1)/2]=3。

② 查找计次 k=1，将 6 与中间数据元素 4 进行比较，6>4，不相等。

③ 查找计次 k=2，继续在后半部分进行查找，后半部分子表的长度为 3，计算中间数据元素的序号 m=3+[(3+1)/2]=5，将 6 与后半部分的中间数据元素 7 进行比较，6<7，不相等。

④ 查找计次 k=3，继续在后半部分序列的前半部分子表中进行查找，子表长度为 1，则中间数据元素的序号 m=3+[(1+1)/2]=4，即与第 4 个数据元素 5 进行比较，不相等，继续查找的子表长度为 0，则查找结束。

3.3.2　排序

排序是指将一个无序的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。这里主要讨论基于顺序存储结构的线性表的排序操作。

1．交换排序法

交换排序法是指通过交换逆序数据元素进行排序的方法。下面介绍的冒泡排序法和快速排序法都属于交换排序法。

（1）冒泡排序法

冒泡排序法是一种简单的交换排序法，它通过将相邻的数据元素（记录）进行交换，逐步将待排序序列变成有序序列。冒泡排序法的基本思想是：从头扫描待排序记录序列，在扫描的过程中顺序比较相邻的两个数据元素的大小。以升序为例，在第一趟排序中，对 n 个记录进行如下操作：将相邻的两个记录的关键字（能唯一进行数据元素区分的数据项或数据项集合）进行比较，逆序时就交换位置。在扫描的过程中，不断地将相邻两个记录中关键字大的记录向后移动，最后将待排序记录序列中的最大关键字记录换到了待排序记录序列的末尾，这也是最大关键字记录应在的位置。然后进行第二趟冒泡排序，对前 $n-1$ 个记录进行同样的操作，其结果是关键字大的记录被放在第 $n-1$ 个记录的位置上。如此反复，直到排好序为止（若在某一趟冒泡过程中，没有发现一个逆序，则可结束冒泡排序），所以冒泡过程最多进行 $n-1$ 趟。

【例 3.7】 现有序列（5,2,9,4,1,7,6），将该序列从小到大进行排列。

采用冒泡排序法对序列进行排序，具体操作步骤如下。

序列长度 n=7							
原序列	5	2	9	4	1	7	6
第一趟（从前往后）	5←→	2	9	4	1	7	6
	2	5	9←→	4	1	7	6
	2	5	4	9←→	1	7	6
	2	3	4	1	9←→	7	6
	2	5	4	1	7	9←→	6
第一趟结束后	2	5	4	1	7	6	9
第二趟（从前往后）	2	5←→	4	1	7	6	9
	2	4	5←→	1	7	6	9
	2	4	1	5	7←→	6	9
	2	4	1	5	6	7	9
第二趟结束后	2	4	1	5	6	7	9
第三趟（从前往后）	2	4←→	1	5	6	7	9
	2	1	4	5	6	7	9
第三趟结束	2	1	4	5	6	7	9
第四趟（从前往后）	2←→	1	4	5	6	7	9
	1	2	4	5	6	7	9
第四趟结束	1	2	4	5	6	7	9
最后结果	1	2	4	5	6	7	9

冒泡排序法最多需要扫描 $n-1$ 次，如果各数据元素已经就位，则扫描结束。测试各数据元素是否已经就位，可设置一个标志，如果该次扫描没有数据交换，则说明排序结束。

（2）快速排序法

冒泡排序法每次交换只能改变相邻两个数据元素之间的逆序，速度相对较慢。如果将两个不相邻的数据元素进行交换，则可以消除多个逆序。

快速排序法的操作过程如下。

从线性表中选取一个数据元素，设为 T，将线性表后面小于 T 的数据元素移到前面，而前面大于 T 的数据元素移到后面，结果是线性表被分为两部分（称为两个子表），T 插入其分界线的位置处，这个过程称为线性表的分割。对线性表的一次分割，就以 T 为分界线，将线性表分成前后两个子表，且前面子表中的所有数据元素均小于或等于 T，而后面的所有数据元素均大于 T。

再将前后两个子表进行相同的快速排序，将子表再进行分割，直到所有的子表均为空，则完成快速排序操作。

在快速排序过程中，随着对各个子表不断地进行分割，划分出的子表会越来越多，但一次又只能对一个子表进行分割，所以需要将暂时不用的子表记忆起来，这里可用栈来实现。

对某个子表进行分割后，可以将分割出的后一个子表的第一个数据元素与最后一个数据元素的位置压入栈中，而继续对前一个子表进行再分割；当分割出的子表为空时，可以从栈中退出一个子表进行分割。

这个过程直到栈为空为止，说明所有子表为空，没有子表需要分割，排序完成。

2．插入排序法

插入排序法是指在一个已排好序的记录子集的基础上，将下一个待排序的记录有序插入已排好序的记录子集中，直到将所有待排记录全部插入为止。

打扑克牌时的抓牌就是插入排序的一个例子，每抓一张牌，插入合适位置，直到抓完牌为止，即可得到一个有序序列。

直接插入排序是一种最基本的插入排序法。其基本操作是将第 i 个记录插入前面 $i-1$ 个已排好序的记录中，具体过程为：将第 i 个记录的关键字 K_i 顺次与其前面 $i-1$ 个记录的关键字 $K_{i-1},K_{i-2},\cdots,K_n$ 进行比较，将所有关键字大于 K_i 的记录依次向后移动一个位置，直到遇见一个关键字小于或等于 K_i 的记录（关键字 K_j），此时该记录后面必为空位置，将第 i 个记录插入空位置即可。完整的直接插入排序是从 $i=2$ 开始的，也就是说，将第 1 个记录视为已排好序的单数据元素子集合，然后将第 2 个记录插入单数据元素子集合中。i 从 2 循环到 n，即可实现完整的直接插入排序。

该方法与冒泡排序法的效率相同，最坏的情况需要进行 $n(n-1)/2$ 次比较。

【例 3.8】 现有序列（5,2,9,4,1,7,6），将该序列从小到大进行排序。

采用简单插入排序法对序列进行排序，具体操作步骤如下。

	5	2	9	4	1	7	6
		↑j=2					
	2	5	9	4	1	7	6
			↑j=3				
	2	5	9	4	1	7	6
				↑j=4			
	2	4	5	9	1	7	6
序列长度 n=7					↑j=5		
	1	2	4	5	9	7	6
						↑j=6	
	1	2	4	5	7	9	6
							↑j=7
插入排序后的结果	1	2	4	5	6	7	9

3．选择排序法

选择排序法的基本思想是在每趟排序中，在 $n-i+1$（$i=1,2,\cdots,n-1$）个记录中选取关键字最小的记录作为有序序列中第 i 个记录。下面介绍简单选择排序法。

简单选择排序法的基本思想，在第 i 趟排序中，通过 $n-i$ 次关键字的比较，从 $n-i+1$ 个记录中选出关键字最小的记录，并和第 i 个记录进行交换。共需进行 $i-1$ 趟比较，直到所有记录

排序完成为止。例如，在进行第 i 趟比较时，从当前候选记录中选出关键字最小的第 k 个记录，并和第 i 个记录进行交换。图 3.23 所示为简单选择排序法示例，图中有方框的是被选择出来的最小关键字。

原序列	89	21	56	48	85	16	19	47
第 1 趟选择	**16**	21	56	48	85	89	19	47
第 2 趟选择	16	**19**	56	48	85	89	21	47
第 3 趟选择	16	19	**21**	48	85	89	56	47
第 4 趟选择	16	19	21	**47**	85	89	56	48
第 5 趟选择	16	19	21	47	**48**	89	56	85
第 6 趟选择	16	19	21	47	48	**56**	89	85
第 7 趟选择	16	19	21	47	48	56	**85**	89

图 3.23　简单选择排序法示例

3.4　知识扩展

3.4.1　树

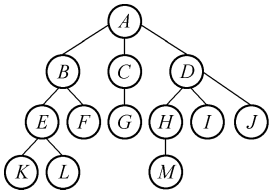


图 3.24　树的图表示

树是一种简单的非线性结构，在树的图表示中，用直线连接两端的节点，上端点为直接前驱，下端点为直接后继，如图 3.24 所示。

线性结构中节点间具有唯一前驱、唯一后继关系，而非线性结构的特征则是节点间的前驱、后继关系不具有唯一性。在树状结构中，节点间的关系是前驱唯一而后继不唯一，即节点之间是一对多的关系。直观地看，树状结构是指具有分支关系的结构（其分叉、分层的特征，类似于自然界中的树）。树状结构应用非常广泛，特别是在处理大量数据方面，如文件系统、编译系统、目录组织等。

在树状结构中，每个节点只有一个直接前驱，称为双亲节点，如图 3.24 所示的 D 节点即为节点 H、I、J 的双亲节点。没有双亲节点的节点只有一个，称为根节点，如图 3.24 所示的节点 A。每个节点可以有多个后件，它们均称为该节点的子节点，图 3.24 中的节点 E、F 是节点 B 的子节点。没有后件的节点，称为叶子节点，图 3.24 中的叶子节点有 K、L、F、G、M、I 和 J。

与树相关的术语如下。

- 节点：包含一个数据元素及若干指向其他节点的分支信息。
- 节点的度：一个节点的子树个数称为此节点的度。例如，在图 3.24 中的节点 D 的度为 3，节点 E 的度为 2。按此原则，所有叶子节点的度均为 0。
- 叶子节点：度为 0 的节点，即无后继的节点，也称为终端节点。
- 分支节点：度不为 0 的节点，也称为非终端节点。
- 孩子节点：一个节点的直接后继称为该节点的孩子节点。在图 3.24 中，B、C 是 A 的孩子节点。
- 双亲节点：一个节点的直接前驱称为该节点的双亲节点。在图 3.24 中，A 是 B、C、D 的双亲节点。
- 兄弟节点：同一双亲节点的孩子节点之间互称兄弟节点。在图 3.24 中，节点 H、I、J 互为兄弟节点。

- 祖先节点：一个节点的祖先节点是指从根节点到该节点的路径上的所有节点。在图 3.24 中，节点 *K* 的祖先节点是 *A*、*B*、*E*。
- 子孙节点：一个节点的直接后继和间接后继称为该节点的子孙节点。在图 3.24 中，节点 *D* 的子孙节点是 *H*、*I*、*J*、*M*。
- 树的度：树中所有节点的度的最大值。在图 3.24 中，所有节点中最大的度是 3，所以该树的度为 3。
- 节点的层次：从根节点开始定义，根节点的层次为 1，根节点的直接后继的层次为 2，以此类推。
- 树的深度（高度）：树中所有节点的层次的最大值。
- 有序树：在树 *T* 中，如果各子树 T_i 之间是有先后顺序的，则称为有序树。
- 森林：*m*（$m \geqslant 0$）棵互不相交的树的集合。将一棵非空树的根节点删去，树就变成一个森林；反之，给森林增加一个统一的根节点，森林就变成一棵树。
- 树分层，根节点为第一层，往下以此类推。同一层节点的所有子节点均在下一层。如图 3.24 所示：*A* 节点在第 1 层，*B*、*C*、*D* 节点在第 2 层；*E*、*F*、*G*、*H*、*I*、*J* 节点在第 3 层；*K*、*L*、*M* 节点在第 4 层。

3.4.2 二叉树

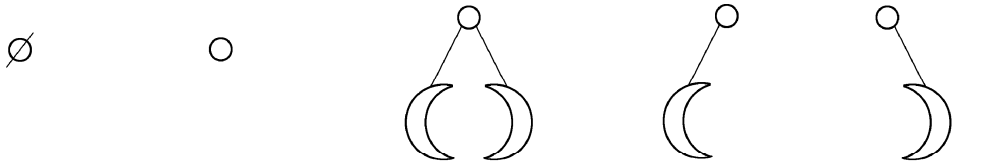
1. 二叉树的定义

二叉树是一个有限节点的集合，该集合或者为空，或者由一个根节点和两棵互不相交的称为该根节点的左子树和右子树所组成。二叉树应满足以下两个条件。

① 每个节点的度都不大于 2。

② 每个节点的孩子节点的次序不能任意颠倒。

因此，一个二叉树中的每个节点只能含有 0、1 或 2 个孩子，而且每个孩子有左右之分。我们把位于左边的孩子称为左孩子，把位于右边的孩子称为右孩子。图 3.25 所示为二叉树的 5 种基本形态。



（a）空二叉树　（b）只有一个根节点的二叉树　（c）具有左、右子树的二叉树　（d）只有左子树的二叉树　（e）只有右子树的二叉树

图 3.25　二叉树的 5 种基本形态

2. 二叉树的性质

性质 1：在二叉树的第 *i* 层上至多有 2^{i-1} 个节点（$i \geqslant 1$）。

证明：用数学归纳法进行证明。

归纳基础：当 $i=1$ 时，整个二叉树只有一个根节点，此时 $2^{i-1}=2^0=1$，结论成立。

归纳假设：假设 $i=k$ 时结论成立，即第 *k* 层上的节点总数最多有 2^{k-1} 个。

现证明当 $i=k+1$ 时，结论成立。

因为二叉树中每个节点的度最大为 2，所以第 $k+1$ 层的节点总数最多为第 *k* 层上节点最大数的 2 倍，即 $2\times 2^{k-1}=2^{(k+1)-1}$，故结论成立。

性质 2：深度为 k 的二叉树至多有 2^k-1 个节点（$k\geqslant 1$）。

证明：因为深度为 k 的二叉树，其节点总数的最大值是二叉树每层上节点的最大值之和，所以深度为 k 的二叉树的节点总数最多有

$$\sum_{i-1}^{k}\text{第}i\text{层上的最大节点个数}=\sum_{i-1}^{k}2^{i-1}=2^k-1$$

故结论成立。

性质 3：任意一棵二叉树 T，若终端节点数为 n_0，而其度为 2 的节点数为 n_2，则 $n_0=n_2+1$。

证明：设二叉树中节点总数为 n，二叉树中度为 1 的节点总数为 n_1。

因为二叉树中所有节点的度均小于或等于 2，所以有

$$n=n_0+n_1+n_2$$

设二叉树中的分支数目为 B，因为除根节点外，每个节点均对应一个进入它的分支，所以有

$$n=B+1$$

又因为二叉树中的分支都是由度为 1 和度为 2 的节点发出的，所以分支数目为

$$B=n_1+2n_2$$

整理上述两式可得

$$n=B+1=n_1+2n_2+1$$

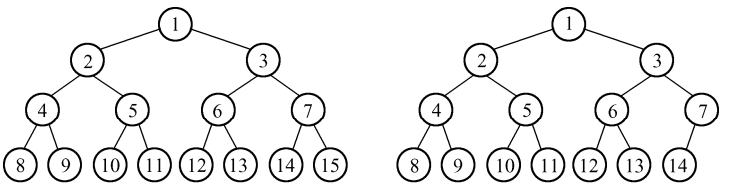
将 $n=n_0+n_1+n_2$ 代入上式得出 $n_0+n_1+n_2=n_1+2n_2+1$，整理后得 $n_0=n_2+1$，故结论成立。

下面先给出两种特殊的二叉树，然后讨论其有关性质。

满二叉树：深度为 k 且有 2^k-1 个节点的二叉树。在满二叉树中，每层节点都是满的，即每层节点都具有最大节点数。如图 3.26（a）所示的二叉树，即为一棵满二叉树。

满二叉树的顺序表示为，从二叉树的根节点开始，层间从上到下，层内从左到右，逐层进行编号（1,2,…,n）。例如，图 3.26（a）中的满二叉树的顺序表示为（1,2,3,4,5,6,7,8,9,10,11,12,13,14,15）。

完全二叉树：深度为 k，节点数为 n 的二叉树，如果其节点 1～n 的位置序号分别与满二叉树的节点 1～n 的位置序号一一对应，则为完全二叉树，如图 3.26（b）所示。



（a）满二叉树　　（b）完全二叉树

图 3.26　满二叉树与完全二叉树

满二叉树必为完全二叉树，而完全二叉树不一定是满二叉树。

性质 4：具有 n 个节点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor+1$。

证明：假设具有 n 个节点的完全二叉树的深度为 k，根据性质 2 可知，深度为 $k-1$ 的满二叉树的节点总数为

$$n_1=2^{k-1}-1$$

深度为 k 的满二叉树的节点总数为

$$n_2=2^k-1$$

显然有 $n_1<n\leqslant n_2$，进一步可以推出 $n_1+1\leqslant n<n_2+1$。

将 $n_1=2^{k-1}-1$ 和 $n_2=2^k-1$ 代入上式，可得 $2^{k-1}\leqslant n<2^k$，即 $k-1\leqslant \log_2 n<k$。

因为 k 是整数，所以 $k-1=\lfloor \log_2 n \rfloor$，$k=\lfloor \log_2 n \rfloor+1$，故结论成立。

性质 5：对于具有 n 个节点的完全二叉树，如果按照从上到下和从左到右的顺序对二叉树中的所有节点从 1 开始顺序编号，则对于任意的序号为 i 的节点有如下结论。

① 如果 $i=1$，则序号为 i 的节点是根节点，无双亲节点；如果 $i>1$，则序号为 i 的节点的双亲节点序号为 $\lfloor i/2 \rfloor$。

② 如果 $2\times i>n$，则序号为 i 的节点无左孩子；如果 $2\times i\leqslant n$，则序号为 i 的节点的左孩子节点的序号为 $2\times i$。

③ 如果 $2\times i+1>n$，则序号为 i 的节点无右孩子；如果 $2\times i+1\leqslant n$，则序号为 i 的节点的右孩子节点的序号为 $2\times i+1$。

可以用数学归纳法证明其中的②和③。

当 $i=1$ 时，由完全二叉树的定义可知，如果 $2\times i=2\leqslant n$，则说明二叉树中存在两个或两个以上的节点，所以其左孩子存在且序号为 2；反之，如果 $2>n$，则说明二叉树中不存在序号为 2 的节点，其左孩子不存在。同理，如果 $2\times i+1=3\leqslant n$，则说明其右孩子存在且序号为 3；如果 $3>n$，则说明二叉树中不存在序号为 3 的节点，其右孩子不存在。

假设对于序号为 j（$1\leqslant j\leqslant i$）的节点，当 $2\times j\leqslant n$ 时，其左孩子存在且序号为 $2\times j$；当 $2\times j>n$ 时，其左孩子不存在；当 $2\times j+1\leqslant n$ 时，其右孩子存在且序号为 $2\times j+1$；当 $2\times j+1>n$ 时，其右孩子不存在。

当 $i=j+1$ 时，根据完全二叉树的定义，如果二叉树的左孩子存在，则其左孩子节点的序号一定等于序号为 j 的节点的右孩子的序号加 1，即其左孩子节点的序号=$(2\times j+1)+1=2(j+1)=2\times i$，且有 $2\times i\leqslant n$；如果 $2\times i>n$，则左孩子不存在。如果右孩子节点存在，则其右孩子节点的序号应等于其左孩子节点的序号加 1，即右孩子节点的序号=$2\times i+1$，且有 $2\times i+1\leqslant n$；如果 $2\times i+1>n$，则右孩子不存在。

故②和③得证。

由②和③，我们可以很容易证明①。

当 $i=1$ 时，显然该节点为根节点，无双亲节点。当 $i>1$ 时，设序号为 i 的节点的双亲节点的序号为 m，如果序号为 i 的节点是其双亲节点的左孩子，根据②有 $i=2\times m$，即 $m=i/2$；如果序号为 i 的节点是其双亲节点的右孩子，根据③有 $i=2\times m+1$，即 $m=(i-1)/2=i/2-1/2$，综合这两种情况，可以得到，当 $i>1$ 时，其双亲节点的序号等于 $\lfloor i/2 \rfloor$。

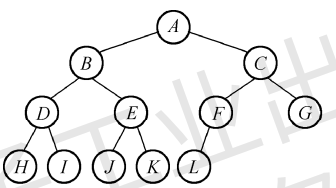
证毕。

3．二叉树的存储结构

二叉树的结构是非线性的，每个节点最多可有两个后继。二叉树有顺序存储结构和链式存储结构两种。

（1）顺序存储结构

顺序存储结构是指用一组连续的存储单元来存放二叉树的数据元素，如图 3.27 所示。



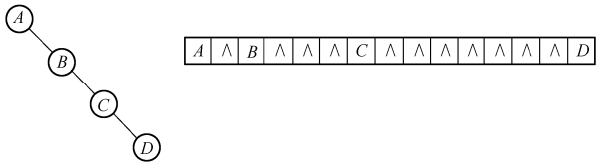
（a）二叉树

（b）二叉树的顺序存储结构

图 3.27　二叉树及其顺序存储结构

用一维数组存储，将二叉树中编号为 i 的节点存放在数组的第 i 个分量中。这样，可得节点 i 的左孩子的位置为 LChild(i)=2×i，右孩子的位置为 RChild(i)=2×i+1。

显然，这种存储方式对于一棵完全二叉树来说是非常方便的。因为此时该存储结构既不浪费空间，又可以根据公式计算出每个节点的左、右孩子的位置。但是，对于一般的二叉树而言，其必须按照完全二叉树的形式来存储，这就造成了空间的浪费。单支二叉树及其顺序存储结构如图 3.28 所示，从中可以看出，一个深度为 k 的二叉树，在最坏的情况下（每个节点只有右孩子）需要占用 2^k-1 个存储单元，而实际该二叉树只有 k 个节点，浪费的空间太大。这是顺序存储结构的一大缺点。



（a）单支二叉树　　（b）顺序存储结构

图 3.28　单支二叉树及其顺序存储结构

（2）链式存储结构

对于任意的二叉树来说，每个节点只有两个孩子，一个双亲节点。因此我们可以设计二叉树节点至少包括三个域：Data 域、LChild 域和 RChild 域，如图 3.29 所示。

其中，LChild 域指向该节点的左孩子，Data 域记录该节点的信息，RChild 域指向该节点的右孩子。

有时，为了便于找到双亲节点，可以增加一个 Parent 域，Parent 域指向该节点的双亲节点，如图 3.30 所示。

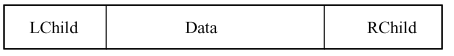


图 3.29　链式存储结构下二叉树节点示意

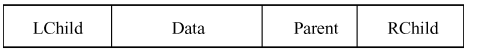
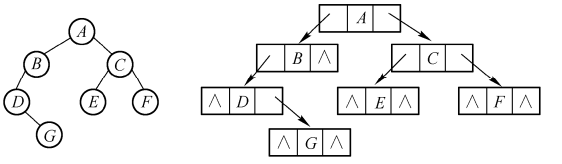


图 3.30　具有 Parent 域的二叉树节点示意

用图 3.29 节点结构形成的二叉树的链式存储结构称为二叉链表，如图 3.31 所示；用图 3.30 节点结构形成的二叉树的链式存储结构称为三叉链表。



（a）二叉树　　（b）二叉链表

图 3.31　二叉树及其二叉链表

若一个二叉树含有 n 个节点，则它的二叉链表中必含有 $2n$ 个指针域，其中必有 n+1 个空的指针域。此结论证明如下。

证明：分支数目 $B=n-1$，即非空的指针域有 $n-1$ 个，故空指针域有 $2n-(n-1)=n+1$ 个。

不同的存储结构实现二叉树的操作也不同，如要找某个节点的双亲节点，在三叉链表中很容易实现；在二叉链表中则需从根指针出发一一查找。可见，在具体应用中，要根据二叉树的形态和要进行的操作来决定二叉树的存储结构。

4．二叉树的遍历

二叉树的遍历是指按一定规律对二叉树中的每个节点进行访问且仅访问一次。其中的访

问可指计算二叉树中节点的数据信息、打印该节点的信息，以及对节点进行的任何其他操作。

为什么需要遍历二叉树呢？因为二叉树是非线性的结构，所以需要通过遍历将二叉树中的节点访问一遍，得到访问节点的顺序序列。从这个意义上说，遍历操作就是将二叉树中的节点按一定规律进行线性化的操作，目的在于将非线性化结构变成线性化的访问序列。二叉树的遍历操作是二叉树中最基本的运算。

二叉树的基本组成结构如图 3.31（a）所示，它由根节点、左子树和右子树 3 个基本单元组成，因此只要依次遍历这 3 部分，即可遍历整个二叉树。

（1）先序遍历（DLR）

若二叉树为空，则不进行操作，否则依次执行如下 3 步操作：①访问根节点；②按先序遍历左子树；③按先序遍历右子树。

（2）中序遍历（LDR）

若二叉树为空，则不进行操作，否则依次执行如下 3 步操作：①按中序遍历左子树；②访问根节点；③按中序遍历右子树。

（3）后序遍历（LRD）

若二叉树为空，则不进行操作，否则依次执行如下 3 步操作：①按后序遍历左子树；②按后序遍历右子树；③访问根节点。

显然，这种遍历是一个递归过程。

对如图 3.32 所示的二叉树进行先序、中序、后序遍历后的序列如下。

- 先序遍历：*A*,*B*,*D*,*F*,*G*,*C*,*E*,*H*。
- 中序遍历：*B*,*F*,*D*,*G*,*A*,*C*,*E*,*H*。
- 后序遍历：*F*,*G*,*D*,*B*,*H*,*E*,*C*,*A*。

最早提出的遍历问题是对存储在计算机中的表达式求值。例如，表达式(*a*+*b***c*)−*d*/*e*，用二叉树表示如图 3.33 所示。当对此二叉树进行先序、中序、后序遍历后，便可获得表达式的前缀、中缀、后缀的书写形式。

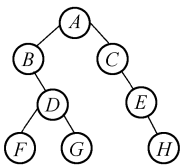


图 3.32　二叉树

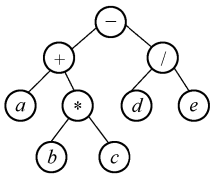


图 3.33　表达式用二叉树表示

- 前缀：−+*a***bc*/*de*。
- 中缀：*a*+*b***c*−*d*/*e*。
- 后缀：*abc**+*de*/−。

其中，中缀形式是算术表达式的一般形式，只是没有括号。前缀表达式称为波兰表达式。后缀表达式称为逆波兰表达式。在计算机中，使用后缀表达式易于求值。

习　题　3

一、填空题

1．在树状结构中，树的根节点没有_____。

2．栈顶的位置是随着_____操作而变化的。

3．顺序存储是把逻辑上相邻的节点存储在物理位置_____的存储单元中。

4．数据的逻辑结构有线性和_____两大类。

5．在算法正确的前提下，评价一个算法的两个标准是_____。

6．数据结构分为逻辑结构与存储结构，线性链表属于_____。

7．在最坏情况下，冒泡排序法的时间复杂度为_____。

8．数据的基本单位是_____。

9．算法的工作量大小和实现算法所需的存储单元的多少分别称为算法的_____。

10．在长度为 n 的基于顺序存储结构的线性表中，当在任何位置上插入一个数据元素的概率都相等时，则插入一个数据元素所需移动数据元素的平均个数为_____。

11．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_____个数据元素。

12．有序线性表能进行二分查找的前提是该线性表必须是_____存储的。

13．用树状结构表示实体类型及实体间联系的数据模型称为_____。

14．设一棵完全二叉树共有 700 个节点，则在该二叉树中共有_____个叶子节点。

15．设一棵二叉树的中序遍历结果为 *DBEAFC*，前序遍历结果为 *ABDECF*，则其后序遍历结果为_____。

二、选择题

1．用链表示线性表的优点是（　　）。

A．便于随机存取　　B．花费的存储空间比顺序存储的少

C．便于进行插入和删除操作　　D．数据元素的物理顺序与逻辑顺序相同

2．在单链表中，增加头节点的目的是（　　）。

A．方便运算的实现　　B．使单链表至少有一个节点

C．标志表中头节点的位置　　D．说明单链表是线性表的链式存储实现

3．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

4．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

5．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

6．数据的存储结构是指（　　）。

A．数据所占的存储空间　　B．数据的逻辑结构在计算机中的表示

C．数据在计算机中的顺序存储方式　　D．存储在外存中的数据

7．下列关于队列的叙述正确的是（　　）。

A．在队列中只能插入数据　　B．在队列中只能删除数据

C．队列是先进先出的线性表　　D．队列是先进后出的线性表

8．下列关于栈的叙述正确的是（　　）。

A．在栈中只能插入数据　　B．在栈中只能删除数据

C．栈是先进先出的线性表　　D．栈是先进后出的线性表

9．若进栈的输入序列是(*A*,*B*,*C*,*D*,*E*)，并且在它们进栈的过程中可以进行出栈操作，则不可能出现的出栈序列是（　　）。

A．*EDCBA*　　B．*DECBA*　　C．*DCEAB*　　D．*ABCDE*

10．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（　　）。

A．*n*+1　　B．*n*　　C．(*n*+1)/2　　D．*n*/2

11．在数据结构中，与所使用的计算机无关的是数据的（　　）。

A．存储结构　　B．物理结构　　C．逻辑结构　　D．物理和存储结构

12．一些重要的程序设计语言（如 C 语言和 Pascal 语言）允许过程的递归调用，而实现递归调用中的存储分配通常用（　　）。

A．栈　　B．堆　　C．数组　　D．链表

13．下列叙述中正确的是（　　）。

A．算法就是程序　　B．在设计算法时只需要考虑数据结构的设计

C．在设计算法时只需要考虑结果的可靠性　　D．以上三种说法都不对

14．如果进栈序列为(e_1,e_2,e_3,e_4)，则可能的出栈序列是（　　）。

A．e_3,e_1,e_4,e_2　　B．e_2,e_4,e_3,e_1　　C．e_3,e_4,e_1,e_2　　D．任意顺序

15．下列关于栈的叙述正确的是（　　）。

A．栈顶数据元素最先被删除　　B．栈顶数据元素最后才能被删除

C．栈底数据元素永远不能被删除　　D．以上三种说法都不对

16．已知二叉树的后序遍历序列是 *dabec*，中续遍历序列是 debac，则它的前序遍历序列是（　　）。

A．*acbed*　　B．*decab*　　C．*deabc*　　D．*cebda*

17．在深度为 5 的满二叉树中，叶子节点的个数为（　　）。

A．32　　B．31　　C．16　　D．15

18．设树 *T* 的度为 5，其中度为 1、2、3 的节点个数分别为 5、1、1。则树 *T* 中的叶子节点个数为（　　）。

A．8　　B．7　　C．6　　D．5

19．已知一棵树的前序遍历和中序遍历序列分别为 *ABDEGCFH* 和 *DBGEACHF*，则该二叉树的后序遍历序列为（　　）。

A．*GEDHFBCA*　　B．*DGEBHFCA*　　C．*ABCDEFGH*　　D．*ACBFEDHG*

20．树是节点的集合，它的根节点数目（　　）。

A．有且只有 1 个　　B．1 或多于 1 个　　C．0 或 1 个　　D．至少有 2 个

三、简答题

1．什么是算法？算法具有的基本特点有哪些？

2．什么是数据结构？逻辑结构和物理结构各有什么特点？

3．什么是线性结构？线性表、栈和队列各有什么特点？

4．顺序存储和链式存储各有什么优缺点？

5．二叉树有哪些基本特征？

6．试说明二叉树的三种遍历顺序的特点。

7．简述顺序比较法的基本原理。

8．常见的排序方法有哪些？各有什么特点？