数据結构简明教程

张乃孝　樊文飞　编

科　海　培　训　中　心

1987年1月

数据結构简明教程

张乃孝　樊文飞　编

科　海　培　训　中　心

1987年1月

目　录

参考书目

[1] 《数据结构》许卓群 张乃孝 杨冬青 唐世渭合编 高等教育出版社 1987年出版

[2] 《Pascal程序设计》 Peter Grogono著 蒋国南译 清华大学出版社 1981年出版

[3] 《DATA TYPES AND STRUCTURES》 C.C.Gotlieb, LeoR Gotlieb

[4] 《FUNDAMENTALS OF DATA STRUCTURES》
Ellis Horowity
Sartaj Sahni

第一章　引论

1.1 数据结构概述

1.1.1 学习数据结构的重要性

数据结构在计算机科学中的地位如何？

让我们先来考虑一个大型工厂的模型。工厂的生产过程可以看成对原材料加工操作，最后得出产品的处理过程。这个处理过程显然包括两个关键阶段：

原材料的管理。大量的原料如何在仓库中组织、储存、管理？杂乱无章地堆积的原材料将影响生产的效率。

对原料的加工操作采取什么样的工艺技术、按照什么样的操作顺序对原料进行处理？处理过程是根据不同的原料确定的。

我们可以将这个工厂模型归结为一个处理机，图示如下：

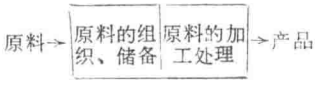


图 1-1

由此可见，原料管理与原料加工是工厂正常生产的关键。

学习计算机科学的目的是应用计算机解决问题。计算机的解题过程也是对原料进行加工操作后而得出产品的处理过程。计算机处理的原料是数据，产品是数据结果，对数据的处理是由算法决定的。

数据是对现实世界的事物采用计算机能够识别、存储和处理的形式所进行的描述。简言之，数据就是计算机化的信息。随着计算机的发展和应用领域的扩大，数据量越来越大，数据间的联系越来越复杂，对数据组织的研究也越来越受到重视。

算法是精确定义的一系列规则，指出怎样从给定的输入信息经过有限步骤得到输出信息。输入信息是要处理的数据，输出信息是得出的数据结果。要对输入信息做出正确有效的处理，就必须设计出正确有效的算法。

计算机对问题的处理可图示如下：

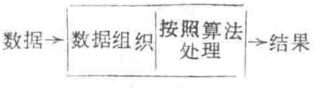


图 1-2

由此可见，计算机解题的关键是数据的组织和算法的设计。无怪乎有人将计算机科学定义为：对数据、以及利用数字计算机对数据进行表示和变换的研究。

数据结构所研究的便是数据的组织与存储、算法的设计和分析。这样看来，学习数据结构是学习计算机科学的重要基础。

应该指出，数据结构和算法之间存在着本质的联系，失去一方，另一方就没有什么意义。我们在谈到一种类型的数据结构时，总是离不开对这种类型数据结构需要施加的各种运算（又称操作，转换），例如，替换、插入、删除等等，而且只有通过对这些运算的算法的研究才能更清楚地理解这种数据结构的意义和作用。反过来，当我们谈论一种算法时，也总是自然地联系到作为该算法处理对象和结果的数据及其结构。因此，在进行数据

结构学习时，我们必须将二者联系起来看，联系起来学。

1.1.2 数据结构的定义

一般说来，数据结构是指数据之间的关系。但关于数据结构的概念至今并没有一个公认的标准定义，不过在谈到任何一种结构时，都自然地联系到对这种类型数据所需要的运算，以及为了在计算机上执行这些运算需要把这些数据如何存储在计算机中。所以数据结构概念一般包括数据之间的逻辑关系，数据在计算机中的存储方法，和数据的运算三个方面。

为了叙述上的方便，把上述数据结构的三个方面分别称为数据的逻辑结构，数据的存储结构，和数据的运算。例如一个线性表，它的逻辑结构是指对下列一些问题的回答：哪一个元素是表中的第一个元素？哪一个元素是表中的最后一个元素？哪些元素在一个给定元素之前或之后？等等。它的存储结构是指这样的内容：它的元素在存储器中是顺序地邻接存放还是用指针连在一起的，等等。数据运算可以包括：在表中找一个元素，从表中删去一个元素，向表中插入一个元素，等等。希望读者学习时，不要孤立地去理解一个方面，而要注意它们之间的联系。

例 1　某工厂职工工资表形式为：

编　号	姓　名	应发工资	附　加　费		代　扣　费		实发金额
			交通补贴	夜班费	房　租	伙　食	
1	王　一	62	2.8	0	3.8	20	41.8
2	丁　二	43	0	0	0.9	15	27.1
3	李　四	130	2	0	10	30	92
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

其中王一、丁二、李四等每个职工在表中各占一行，每行的信息说明一个职工的工资情况，是工资表的基本单位。整个工资表我们称为一个数据结构，而把表中的每一行称为一个结点。结点是数据结构中讨论的基本单位（有的情况下也把结点称作元素、记录、表目等）。上例表中每一行由编号、姓名、应发工资、附加费、代扣费、实发金额等数据项组成；而附加费和代扣费两项又分别由交通补贴、夜班费和房租、伙食等项组成，我们把编号、姓名、应发工资、实发金额等数据项称为初等项。初等项由一个或多个字符组成，是有独立含义的最小标识单位。我们把附加费、代扣费称为组合项。组合项由一个或多个初等项或组合项组成，是有独立含义的标识单位。结点由一个或多个组合项组成，构成结点的组合项又称为字段，我们把其值能唯一确定一个结点的字段或字段组合称为关键码。在上例中编号字段就是关键码，一个职工编号就唯一确定一个职工，即唯一确定工资表中的一行。在PASCAL语言中，记录相当于结点，域相当于字段。而在本书的叙述中，一般采用结点和字段这样的术语。

上述工资表就是一张线性表，表中结点和结点之间是一种简单的线性关系，表中有且有一个结点为表头（第一个结点），有且只有一个结点为表尾（最后一个结点）。对表中任一结点，与它相邻且在它前面的结点最多只有一个，这就是上述工资表的逻辑结构。当

上述工资表存入计算机时，假定结点和结点之间采用顺序存放的方式，这就是存储结构。对这样的表格，在职工调离时相应的结点要删除，在招收新职工时，要增加新的结点，调整工资时要修改，究竟如何进行插入、删除、修改，这就是数据的运算问题。弄清楚以上这些问题，工资表的结构就完全明白了。

综上所述，按某种逻辑关系组织起来的一批数据，按一定的存储表示方式把它存储在计算机的存储器中。并在这些数据上定义了一个运算的集合，就叫做一个数据结构。

下面，我们分别简介数据的逻辑结构、存储结构和数据的运算。

1.1.3 数据的逻辑结构

对数据间关系的描述是数据的逻辑结构，形式地可以用一个二元组

$$B=(K,\ R)$$

来表示，其中K是结点的有穷集合。即K是由有限个结点所构成的集合；R是K上的关系的有穷集合，即R是由有限个关系所构成的集合，而其中的每个关系都是从K到K的关系。在下面的叙述中，凡不易产生混淆之处，我们有时就把数据的逻辑结构简称为数据结构。

设B=(K, R) 是一个逻辑结构，$r\in R$，$k、k'\in K$，如果 $<k,k'>\in r$，则称 k' 是 k 的后继，k是k'的前驱，称k和k'是相邻的结点（都是对r而言）。如果不存在一个 k' 使 $<k,k'>\in r$，则称k为关于r的终端结点（即不存在后继的结点）。如果不存在 k' 使 $<k',k>\in r$，则称k为关于r的开始结点（即不存在前驱的结点）。如果k不是终端结点，也不是开始结点，则称k是内部结点。

在本书所讨论的数据结构里一般只讨论包含一个关系的R，即$R=\{r\}$。对于R中包含多个关系的情况，可以用类似的方法进行讨论。

数据结构可以根据对前驱、后继约束程度的不同划分为线性结构和非线性结构。

本书后面的章节主要内容分布如下：第二章和第三章介绍线性结构的概念和实例；第四章介绍一种重要的非线性结构——树形结构；第五章讨论另一种重要的非线性结构——图；第六章和第七章介绍两种重要的运算——排序和检索；第八章介绍文件部分。

上面我们讨论的逻辑结构是从逻辑关系的角度来考察数据的，它与数据的存储无关，是独立于计算机的。

1.1.4 数据的存储结构

数据的存储结构是逻辑结构在计算机里的表示，它是依赖于计算机的。

对数据的逻辑结构B= 〈K, R〉 的存储必须满足两个条件：

(1) 保证结点集K中的每一个结点k都存储在存储器中。

(2) 这种存储必须明显或隐含地体现结点间的联系，即关系R。

有四种基本的存储方法：

顺序的方法，

键接的方法，

索引的方法，

散列的方法。

我们在后面的章节中，对这四种方法都进行了详细的讨论，在此不再赘述。

1.1.5 数据的运算

研究数据结构是为了更有效地处理数据。对数据的处理一般称为数据的运算。数据的

运算是定义在数据的逻辑结构上的，但运算的具体实现要在存储结构上进行。数据的各种逻辑结构有相应的各种运算。运算的选择是根据数据的逻辑结构的特点而确定的，每种逻辑结构都有一个运算的集合。

在此，我们列出几种常见的运算，作一简要介绍。

（1）检索　检索就是在数据结构里查找满足一定条件的结点。一般是给定某字段的值，找具有该字段值的结点。检索又称为查找。

（2）插入　往数据结构里增加新的结点。

（3）删除　把指定的结点从数据结构里去掉。

（4）更新　改变指定结点的一个或多个字段的值。

插入、删除、更新运算都包括着一个检索运算，以确定插入、删除、更新的确切位置。

（5）排序　保持结构的结点集合里结点数不变，把结点按某种指定的顺序排列。例如，按结点中某一字段的值由小到大对结点进行排列。又如将字符串按字典顺序排列，整数序列按上升顺序排列等等。排序是一种非常有用的运算，第六章中我们将介绍排序的若干算法。

数据处理过程往往由一系列运算共同完成。这一系列运算依据执行的顺序确定了一个运算序列。这个运算序列一般称为算法，关于算法，我们在下一节再加以详细讨论。

1.1.6　数据类型与数据结构

下面，我们讨论两个容易混淆的概念——数据类型和数据结构。

数据类型是指在程序设计语言中各个变量可“包含”的数据种类。一般分为两大类型：初等类型和组合类型。

初等类型是只定义了一个初等项的类型。在PASCAL语言中，初等类型有整数类型、实数类型、布尔类型、字符类型、指针类型等。

组合类型是由初等类型按某种方式组合而成的。不同的组合方式形成不同的类型，如数组类型，记录类型等。

本书中，结点的类型都采用PASCAL中的数据类型。关于PASCAL的数据类型，我们在附录中再详细介绍。

数据结构的概念不仅要描述结点的集合，而且要描述结点之间相互联系的方式，以及可以合法地应用于结点集合K中各个结点的操作集合。也就是说，我们必须对操作集合作出规定，说明它们是如何工作的。数据的逻辑结构、存储结构和运算互相联系，缺一不可，从不同的方面描述了数据结构的概念。

1.2　算法简介

我们讲过，学习数据结构是要培养组织数据的能力和设计算法的能力。上一节我们介绍了数据结构的一些概念，这一节我们讨论算法。

1.2.1　算法的定义

我们知道，在计算机科学这个领域中，“算法”是一个基本的概念。那么，什么是算法的确切定义呢？

算法是一个指令的有限集合，如果遵循它就会完成一项特定的任务。此外，每个算法都必须符合下列准则：

(1) 有穷性。一个算法必须总是在执行有穷步后结束。

(2) 确定性。算法的每一步，必须是确切定义的，无二义性。对于每种情况，要执行的动作必须严格和清楚地规定。

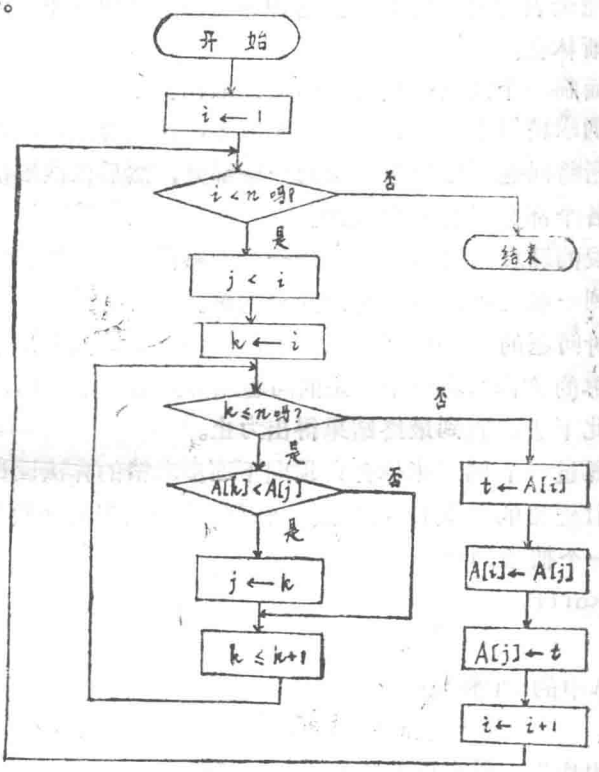
(3) 输入。一个算法有0个或多个输入。它们是在算法开始前对算法最初始的量。这些输入取自特定的对象集合，如取自于整数集。

(4) 输出。一个算法有一个或多个输出。

(5) 有效性。每条指令都必须充分基本，以便原则上可由一个人仅用笔和纸就能实现。

我们给出了算法的定义。下面我们给出一个例子。

例1 数组A中放有n个整数a_1，a_2，…，a_n（$n\geqslant 1$），整数a_i放在数组的第i个元素位置A［i］（$1\leqslant i\leqslant n$）中。如果我们用流程图来表示对n个整数的排序过程，即：把每个处理步骤置于一个框中，并用箭头指出下一步，用不同形状的框代表不同的操作，则对n个整数排序可表示为图1-3。



图中j<i 改为J←i，k≤k+1改为k←k+1

图 1-3

其中+，i，j，k是整型变量。

判断一下，这是不是算法？

算法和程序是有区别的。一份程序未必满足算法准则。一份程序，如计算机的操作系统，可以是永不终止的。

我们已经知道了什么是算法，但是，如何描述一个算法呢？

1.2.2 算法的描述

算法可以用多种方式描述。在本书中，我们使用PASCAL语言来描述算法。但是，

我们并不使用PASCAL语言的各种特异性，因此，一个用PASCAL语言描述的算法可以很方便地翻译成其它语言的程序。为了描述算法的方便，我们这增添了几个PASCAL语言中没有的语句，称为“类PASCAL”语句。我们在附录1中介绍了PASCAL语言。希望不熟悉PASCAL语言的同志在继续学习之前先读一读附录1。

本书中所有算法稍加变动(去掉类PASCAL语句)，都是在IBM-PC上可运行的程序。

下面，我们介绍一种算法的设计方法。

1.2.3 算法的设计

我们向大家建议一种有效的算法设计方法：自顶向下，逐步求精的设计方法。

我们将算法的设计看成多层次结构。高层次表述问题的解的总的和抽象的全貌，低层次包含实现的细节。我们建议从层次结构的顶上（最抽象的一层）入手，即首先提出一个很抽象的解，然后不断将它精细化，直到层次结构的最底下（最详细的一层），能用所选的程序设计语言完全表述出来时为止。这种方法便是自顶向下，逐步求精的方法。这种思想须在实践中逐渐体会。

现在我们又面临一个问题：如何使问题精细化？

有三种主要的求精技术，它们是：分而治之，作出有限进展和情况分析。

分而治之是指将问题划分为不相交的一些部分，然后依次解决每一部分。所谓“不相交”是指问题的各个部分是相互独立的。

作出有限进展的思想是这样的：采用一个朝解的方向得到有限进展的方法，重复应用它，每一次都得到一些进展，最终达到完全的解。

情况分析是对问题的各种情况给以分析，给出更详细的处理描述。

求精方法的总的方向是将一个复杂的问题划分为若干子问题，再将子问题精细化为更简单的问题，如此下去，直到最终结果得出为止。

下面，我们通过一个例子来体会自顶向下逐步求精的算法设计方法。

例2 解决例1提出的整数排序问题。排好序的数列仍放在数组A中。

我们先提出一个抽象的解：

```
procedure sort;
begin
    将数组A中的n个整数排序
  end;
```

然后，我们用作出有限进展的方法精细一步：

```
procedure sort;
  begin
```

(1) 从n个整数中选出一个最小整数放到一个已排好序的数列中，作为该数列中的第一个数；

(2) 反复进行下述处理：从剩下的未排序的整数中选出最小的整数，放到已排好序的数列中，接在该数列中最后一个数的后面，直到没有未排序的数剩下为止；

```
    end;
```

不难看出，我们已将n个数的排序简化了一步，并且，反复执行步骤2，就可将所有整数都放到已排好序的数列中。

现在，我们用情况分析这一技术再使问题变得简单明确一些。

不难看出，步骤1实际上是步骤2的特例。若用步骤2来代替步骤1，则此时“剩下的未排序的整数”即为“所有整数”，而已排好序的数列此时为空。因而步骤1可以取消，只要重复执行几次步骤2即可。

现在把这一粗略的思路进行细化。我们可以认为：共执行n次重复，i从1开始，每执行一次重复，i就加1。当执行完第i－1次重复时，A［1］到A［i－1］为已排好序的数列，而A［i］到A［N］为剩下的未排序的整数。我们从A［i］到A［N］中选出最小整数放在A［i］处。这样可以写出稍细一些的思路：

```
procedure    sort;
 begin
    for i:=1 to n do
          begin
           (1) 检查A［i］到A［n］，并选出最小整数，设为A［j］；
           (2) 把A［i］与A［j］进行交换；
          end
 end;
```

下面，我们再用分而治之的技术，对步骤（1）和步骤（2）进行细化处理。

在上述思路中，实际上第n次重复不必执行，因为此时只剩下了一个未排序的整数，因而不必再执行选出最小整数等操作了。现在对上述思路中步骤(1)和(2)进一步细化。显然，步骤（2）是容易实现的，只要引进一个中间变量，暂时存放A［i］即可；而步骤（1）可以这样来完成：先把A［i］作为最小整数，然后把A［i］与A［i+1］，A［i+2］，…进行比较，一旦出现比A［i］小的整数，譬如说A［k］，则就把A［k］作为新的最小整数，然后再把A［k］与A［k+1］，A［k+2］，…进行比较等等。这样我们可以写出此例的最后算法。

数组A和变量n（存放整数个数）定义如下：

```
VAR A: ARRAY［1…n］OF integer;
```

n是一个常量，可据据实际情况确定。

算法描述如下：

算法1.1　算法示例

```
procedure SORT;
  var  i, j, k, t: integer;
  begin
    for i:=1 to n-1 do
    begin
      j:=i;
    for k:=i to n do
      if A［k］〈A［j］
        then j:=k;
    t:=A［i］;
    A［i］:=A［j］;
    A［i］:=t
  end;
  return
end;
```

说明：删去return，上述算法便成为一个标准PASCAL过程。

掌握设计算法的技术，光靠这一个例子的讲述是不够的。大家必须在实践中体会、熟悉。设计几个算法，动手试一试，可能对你大有裨益。不要怕出错，由设计过程中出错到写出正确的算法，是要有一个过渡过程的。

1.3 数据结构的选择和评价

我们已经知道，有各种不同的数据结构。关于同一个问题的数据集合可以组织不同的数据结构。那么，如何评价不同的数据结构？怎样去选择解决一个应用问题的最佳数据结构呢？

简单说来有两条标准。一条是存储开销，处理同一个问题时设计的不同数据结构，占用存储单元越少的越好。另一条是时间代价，处理该数据结构的算法使用的时间越少越好。

这两者往往是矛盾的，也是可以互相转换的。经常地，我们用牺牲存储效率为代价来提高时间效率，或者用增加执行时间为代价来降低存储开销。情况往往很复杂。我们应根据具体情况，选择出最佳的数据结构。

习题

1．设有数据结构为

$B=(K, r)$，$K=\{k_1, k_2, \cdots, k_9\}$，

$r=\{<K_1, K_3>, <K_1, K_8>, <K_2, K_3>, <K_2, K_4>, <K_2, K_5>, <K_3, K_9>, <K_5, K_6>, <K_8, K_9>, <K_9, K_7>, <K_4, K_7>, <K_4, K_6>\}$

画出这个逻辑结构的图示，并确定相对于关系r哪些结点是开始结点，哪些结点是终端的结点？

2．设字符集为字母和数字的集合，字符的顺序为A，B，C，…，t，0，1，…，9，请将下列字符串按字典顺序排列。

PAB，5C，PABC，CXY

7，B899，B9，CRSI

3．有一个包括100个元素的数组，每个元素的值是个实数，请写出求最大元素的值及其下标的算法。

4．举一个数据结构的例子，叙述其逻辑结构，存储结构，运算等三方面的内容。

第二章　向量、栈和队列

从这一章开始，我们讨论各种具体的数据结构。

概论中已经讲到，根据对结点的前驱、后继约束程度的不同，数据结构可以划分为线性结构和非线性结构。在线性结构中有且仅有一个开始结点，它没有前驱，仅有一个后继；有且仅有一个终端结点，它没有后继，仅有一个前驱；其它所有结点都是内部结点，并且都只有一个前驱和一个后继。线性结构中的所有结点按它们之间的关系可以排成一个线性序列：

$$k_1, k_2, k_3, \cdots\cdots k_n$$

其中k_1是开始结点，k_n是终端结点，k_i是k_{i+1}的前驱，而k_{i+1}是k_i的后继（对所有i=1，2，……n-1成立）。

在讨论线性结构时，通常把上述线性序列称为“线性表”，把线性结构中的结点k_i=(i=1，2…n)称为线性表的“表目”。[注] 在线性表中，我们常说某表目是第i个表目，就是指在上述线性序列中它是第i个，即k_i。i称为该表目的索引，任给一个索引i($1 \leq i \leq n$)。就能在线性表中唯一确定一个表目k_i。

在这一章中，我们讨论几种最常用的线性表——向量、栈和队列。介绍它们存储的顺序表示和链接表示。最后，我们简单介绍可利用空间表及其管理。

[注]：结点是数据结构中讨论的基本单位，它具有广泛的意义，但在不同的结构中，它可以有不同的名称。在线性表的讨论中，我们把结点和表目作为同义词来混用，根据上下文的连贯性，有时采用结点，有时又采用表目。类似的情况后面还会出现，例如在向量中，我们习惯于使用“元素”；图中我们习惯地使用“顶点”；而在文件中又习惯地使用“记录”。

2.1 向量

如果线性表中的所有表目都是同一类型的结点，我们便称之为“向量”，向量中的表目又称为“元素”，元素的索引又称为“下标”。

2.1.1 向量的存储表示

向量常用的二种存储方法是：顺序存储和链接存储。

1. 向量的顺序存储方法

顺序存储方法是将逻辑上相邻的结点存储在物理上相邻的存储单元里，由存储单元的邻接关系来体现结点之间的关系的一种方法。假设我们用loc(k)表示结点k对应的存储单元的地址，k所占存储的第一个单元为Z，最后一个单元为Z_1，k的后继为k′，则Z_1的下一个单元的地址为loc(k′)，即有loc(k′) $=Z_1+1$（结点的地址用存储该结点的第一个单元的地址表示）。

在一般情况下，每个结点所占的存储单元并不只一个，而且所占单元的个数不一定相符，这时顺序存储时仍然一个接一个地填满了整个区域。

顺序方法主要用来存储线性结构，但有时也采用局部线性化的方法来存储非线性结构。

存储向量最常用的方法是顺序存储方法。

由于向量的所有元素属于同一类型。所以每个元素在存储器中占用的空间大小相同。假设向量的第一个元素存放的位置为loc (k_1)，并且为简单起见，设每个元素占用一个存储单元，即占用的空间长度为1，则元素k_i的存放位置为：

$$loc(k_i) = loc(k_1) + 1 * (i - 1)$$

任给一个i，利用上式就可以很快地算出loc(k_i)。

下面，我们给出顺序存储的向量k的形式证明。设向量的元素个数n始终不大于某一整数n。向量中的元素类型是一个已定义的类型datatype。在解决实际问题时，datatype可换作具体类型，我们定义向量k如下：

```
TYPE  vtype = ARRAY [1…n0] oF datatype;
VAR   k:vtype;
```

即用数组k来存储向量k，k [1]，k [2]，…，k [n] 存放向量中各元素。k [n + 1]，…，k [n0] 暂时为空，是备用的结点空间，它们的用途将在向量运算时讨论中表现出来。

向量k的顺序存储的示意图如图2-1所示。

存储向量k的每个元素的存储单元全部用来存放结点的数据，也就是说，用这种存储方法可以将结点的内部信息稠密地放在一起。

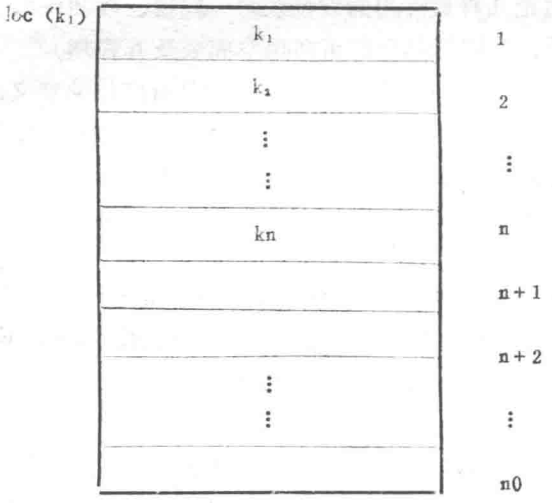


图 2-1　向量的顺序存储

2. 链接的方法

向量的另一种存储方法是链接的方法。

这种方法是给结点附加上指针字段。即将结点所占的存储单元分为两部分，一部分存放结点本身的信息，称数据项；另一部分存放此结点的后继结点所对应的存储单元的地址，称指针项。指针项可以包括一个或多个指针，以指向结点的一个或多个后继，或记录其他信息。（当然，一般来说，指针是用来表示某结点的地址的，并不是只能表示后继结点的地址）。

我们用info表示结点的数据项，用Link表示结点的指针项。那么如果k′是k的后继结点，则有

$oc(k') = k \uparrow .link$。

不难看出，在链接存储表示中，一部分存储单元用来存储结点的数据，另一部分存储空间里存放的是表示结点之间关系的附加的信息——指针。

可以用三种不同的链接方式来存储向量。

(1) 单链表

在单链表里分配给每个结点的存储单元分为两部分，一部分存放结点的数据，称作info字段，另一部分存放的是指向结点后继的指针，称为link字段。终端结点没有后继，它的link字段值为nil（在图中用∧表示），在计算机内部可表示成零或负数，另外还需要一个表头变量head指向表的第一个结点。

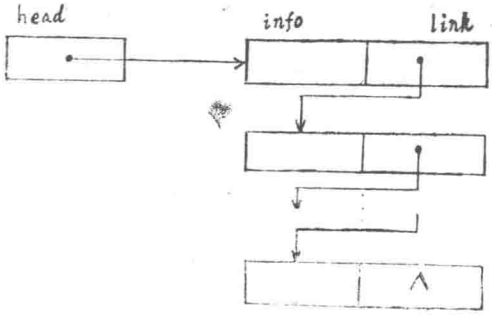


图 2-2 单链表的存储结构

单链表的结点类型及变量head说明如下：

```
TYPE  pointer = ↑node;
      node = RECORD
        info: datatype;
        link: pointer
          END;
VAR   head: pointer;
```

其中datatype是结点的数据所要求的某种已定义过的数据类型，↑node是指向类型node的结点的指针类型。head定义为一个pointer类型的变量，它的值便是一个代表某node类型的结点的地址。

(2) 循环表

将单链表的形式稍作改变，最后一个结点的指针不让它为nil，而让它指向第一个结点，改用表尾变量rear来代替head，rear指向表尾结点，这样就得到了循环表，如图2-3所示。

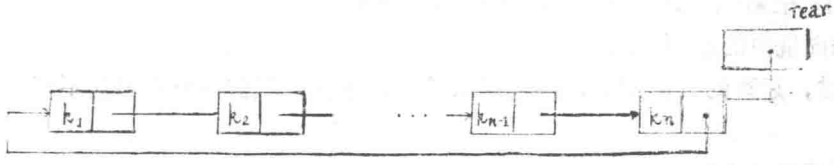


图 2-3 循环表

k_i表示第i个结点info字段的值。循环表的类型定义同单链表，变量rear定义为：

```
VAR  rear:pointer;
```

(3) 双链表

用单链表表示线性表，从任何一个结点都能通过link字段找到它的后继，但不能找出它的前驱，如果在每个结点中增加一个指向前驱的指针，处理就灵活的多。我们用rlink表示指向后继的指针，llink表示指向前驱的指针，用变量head指向第一个结点，用变量rear指向最后一个结点，使得到图2-4所示的双链表。

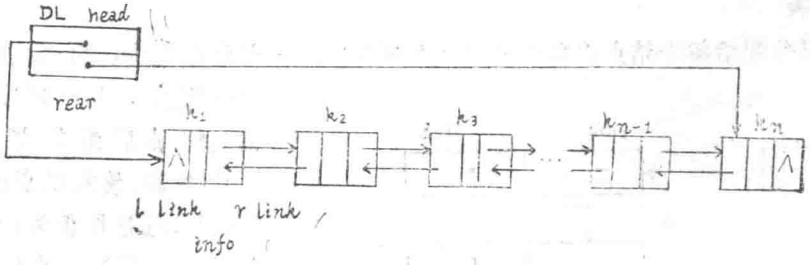


图 2-4 双链表

描述双链表及其结点类型的说明为：

```
TYPE pointer = ↑node;
     node = RECORD
            info: datatype;
            llink, rlink: pointer
            END;
     double = RECORD
            head, rear: pointer
            END;
```

引进一个双链表只要定义一个double型的变量即可。

VAR de double;

我们介绍了几种不同的存储向量的方式。我们为什么设计这些不同的存储方式？这些方式各有什么特点？处理一个实际问题时，我们应该选择什么样的存储方式？这些问题，我们将在下面给出回答。

2.1.2 向量的运算

我们讲过，数据结构由逻辑结构，存储结构和运算三个方面构成。我们已讨论了向量的逻辑结构、存储结构，下面我们讨论向量的三种主要运算：

1）查找向量的第i个元素；

2）在向量中第i个元素的后面插入一个新元素；

3）删除向量中的第i个元素。

我们讲过，运算的具体实现依赖于存储结构。下面，我们便分别根据不同的存储结构来讨论运算。

1．在顺序结构上运算的实现

对于顺序存储的向量k，如果已知$loc(k_1)$，那么对于任意一个i，我们可以公式：

$$loc(k_i) = loc(k_1) + 1 * (i-1)$$

求出k_i的地址，也即找到k_i，假设每个元素占一个存储单元，由此可见，在向量的顺序存储结构中查找第i个元素是很容易的。这里的i实际上起的是索引的作用。

想一想，i的大小要满足什么条件？

下面，我们研究向量的插入。

对顺序存储表的向量，插入的示意图如图2-5所示。

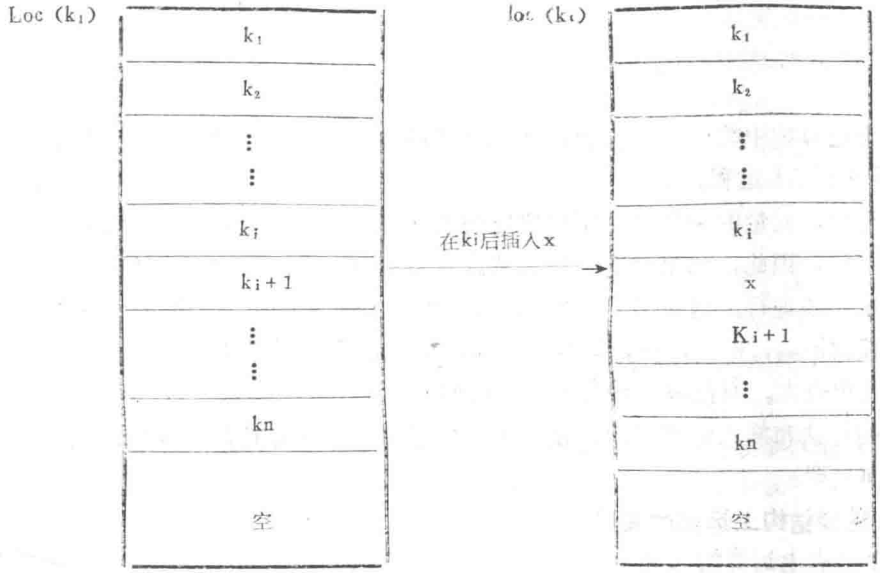


图 2-5　向量的插入运算示意图

其中，x是要插入的数据，其类型和向量元素的类型相同。k_{i+1}，…，k_n向后顺序移动一个单元，一个原来为空的单元被占用。这时，我们可以看出k[n+1]，…，k[n]这些空单元的作用。

向量插入算法中用到的类型和变量说明如下：

```
TYPE  position=1…n0;    {表示向量下标}
```

vtype的说明前面已经给出。

算法描述如下：

算法2.1　向量插入

```
Procedure INSERT (x:datatype;i:position;var n:position;var k:vtype) ;
    var j:position;
    begin
        {检查参数}
        if i> n then begin
                     writeln (' error' ) ;
                     return
                     end;
        if n=n0 then begin
                     writeln (' overflow' ) ;
                     return
                     end;
        {后移}
```