TURING

计算机程序设计竞赛权威指导书

程序设计中实用的数据结构

王建德 吴永辉 编著

人民邮电出版社
POSTS & TELECOM PRESS

TURING

程序设计中实用的数据结构

王建德 吴永辉 编著

人民邮电出版社
北　京

绪　言

最近几年，我们长期从事程序设计研究和竞赛辅导，并在此基础上精心编写了ACM/ICPC竞赛的系列教程，其中包括《新编实用算法分析与程序设计》（人民邮电出版社2008年出版）、《程序设计中常用的计算思维方式》（人民铁道出版社2009年出版）、《程序设计中常用的解题策略》（人民邮电出版社2009年出版）和《程序设计中实用的数据结构》（即本书）。按照“数据结构＋算法＝程序”的定义，《新编实用算法分析与程序设计》和《程序设计中实用的数据结构》属于程序设计的知识范畴，而《程序设计中常用的计算思维方式》和《程序设计中常用的解题策略》分别从思维方式和行为特征这两个角度给出了解决问题的基本方法，属于基于算法和数据结构知识的编程能力范畴。上述所有著作贴近实际，结合实例进行阐释，使读者对什么是程序设计，为什么要学习程序设计，怎样进行程序设计等诸多问题产生浓厚的兴趣和探索意识。因此可以说，这4部著作从知识、能力和情感体验3个维度，建立了程序设计学习的基本框架。

数据结构是计算机软件专业的核心课程，也是编程人员必学的知识。现在关于数据结构的书很多，章节结构大致雷同，具体描述也大同小异。为了避免本书落入“千篇一律”的俗套，我们按照数据结构知识的分类，将全书分为3篇。

上篇“讨论线性表”根据线性结构存储方式的不同，讲解了顺序存储结构和链式存储结构，并根据元素存取方式的不同，讲解了栈和队列两种特殊的线性结构，另外还介绍了线性结构的如下3个重要的应用。

(1) 排序与顺序统计。探讨了贪心策略中的序关系和分治策略在排序中的应用，并在排序知识的基础上阐释了顺序统计的有关方法。

(2) 散列表。介绍了如何使用散列函数确定关键字存储位置，并介绍了在多个关键字的运算结果相同的情况下，如何采用链式存储结构的分离链接法和采用顺序存储结构的开放定址法解决“冲突”。

(3) 后缀数组。介绍了一种专用于字符串处理的重要工具。

中篇“讨论树型问题”详细阐释了什么是树，树有哪些类型，如何通过不同的遍历规则将非线性结构的树转化为线性序列，程序中可采用哪些数据类型存储树，这些数据类型与树操作的效率之间有什么样的关系，怎样通过树形式的并查集实现集合间的合并，以及在哪个集合中查找元素。这一篇着重阐释了结构简单、操作方便且便于二分处理数据的二叉树，介绍了普通有序树转化为等价二叉树的基本方法和如下4种特殊类型的二叉树。

(1) 用二叉堆实现元素入队和按照优先级出队的优先队列。

(2) 在数据通信等方面有广泛用途的最优二叉树。

(3) 用于区间处理和数据统计的线段树。

(4) 用于高效查找数据的3种二叉查找树，即二叉排序树、静态二叉排序树和平衡树。

下篇“讨论图型问题”详尽地阐释了图的基本概念和简单分类，介绍了图的4种存储方式（相邻矩阵、邻接表、关联矩阵和邻接压缩表）和2种遍历规则（宽度优先遍历和深度优先遍历），深入讨论了图的连通性问题，分别给出了在有向图中计算强连通分量和传递闭包以及在无向图中计算割点、桥和双连通分量的基本思路，并重点讲解了计算最小生成树和最短路径的策略。二分图和引入流量因素的单源单汇简单有向图（网络流图）是两类重要的图，我们给出了求解二分图最大匹配以及在加权二分图中计算权和最大匹配的基本方法，并分别叙述了应对不同网络流问题的算法。

在上述3篇的论述中，我们尽力凸显如下特色。

(1) *与时俱进*。尽可能将近年来数据结构研究上涌现的新知识、新成果纳入本书，并且以贴近实际、深入浅出的形式展现给读者。

(2) *注重实际应用能力*。通过实例教会读者，怎样寻找问题中各对象之间的关系，确定数学模型所使用的逻辑结构；怎样实现对各个对象的操作，即确定数据所采用的存储结构；怎样融数据类型与定义在该类型上的运算于一体，为面向对象的程序设计方法奠定基础。

(3) *掌握并联两种数据结构的映射方法*。教会读者在运算对象需要同时采用两种数据结构时，怎样将两种数据结构的元素一一对应起来，使得当一种数据结构中的关键元素发生变化时，另一种数据结构的相应元素也随之变化。

(4) *辩证地看问题*。使读者既能了解线性结构这种基本数据结构的局限性，又可以掌握采用基本数据结构实现复杂非线性结构的基本方法，还进一步探索了用基本数据结构取代非线性结构的简化途径。如果说基本数据结构到非线性结构是一种提高，那么从非线性结构回到基本数据结构却不是一种简单的回归，而是一种螺旋式发展。对于基本数据结构的再次运用，是基于对整个数据结构知识的一种深刻领悟，在进一步理解和运用基本数据结构的过程中，可使我们的知识与思想得到升华。

(5) *精益求精*。随着试题难度和问题规模的不断提升，对算法时空效率的要求愈来愈高，数据结构对算法效率的制约力也愈来愈大。因此，本书旨在引导读者充分考虑数据结构对程序效率所产生的影响，在选择数据结构时，尽可能采用“扬长避短”的原则并结合“取长补短”的方法，更加有效地优化算法。

为了使选用本教程的辅导教师容易教，参与竞赛培训的学生容易学，读者在解题查阅时容易懂，本书在表述上采取了如下4条措施。

(1) 对于每个比较难懂的概念，都举实例加以说明；对于每个比较抽象的定理，都提供具体的应用例证帮助理解；对于每个经典的算法，都有清晰的程序流程给予示范。

(2) 大量运用图表，使得概念、定理和算法的由来更具体、形象和直观。

(3) 定理证明尽量采用初等数学知识，公式推导尽量浅显和详细。

(4) 为了使程序样例不受单一程序设计语言的限制，不让某种特定语言的特殊性掩盖算法的本质内容，书中采用了一种结构清晰、移植性强且贴近自然语言的类程序设计语言。无论你具备Pascal、C、true.bascal等哪一种语言的基础，都可以比较轻松地读懂其语义。本书之所以未给出可直接编译运行的程序代码，除了考虑到语言的通用性外，还有一个重要的原因，就是想给读者留出动手编程的空间，避免无意义的“依葫芦画瓢”。

最后，由衷感谢复旦大学ACM队的舒天民同学。他花费了数月时间，字斟句酌地查找书中的瑕疵，反复测试所有的程序示例，并编程解答了其中部分试题。当我们向他表示感谢时，他却说，这本书使他学到了许多过去闻所未闻的数据结构新知识，编程解题的视野更加开阔了。当然我们自知，这本书不可能是完美的，更不可能涵盖全部的数据结构知识和应用，但在数据结构知识的开发上，可为初学者或浅尝过这门课程的读者提供十分新颖和有用的信息。但愿你能够像舒天民同学一样，通过本书的学习拓展新知，获取灵感，开发睿智，提升能力。

王建德　吴永辉

2011/1/10

目　录

上篇　讨论线性表

中篇　讨论树型问题

下篇 讨论图型问题

Part 1 上 篇

讨论线性表

本篇内容

- 第1章 数组
- 第2章 链式存储结构
- 第3章 两种存取方式特殊的线性表
- 第4章 散列技术
- 第5章 后缀数组

线性结构是由有限个数据元素组成的有序集合，其中每个数据元素都有一个数据项或者多个数据项。这种数据结构是最简单、最常用的，其特征有以下几点。

- **均匀性**。同一线性表中各数据元素的数据类型一致且数据项数相同。例如，字符串是一个线性结构，该结构中每一个数据元素都为单个字符；又如，学生成绩表中的每个数据元素都含学生姓名、学号、若干学科成绩等数据项，因此也是一个线性结构。
- **有序性**。表中数据元素之间的相对位置是线性的，即存在唯一的“第一个”和“最后一个”数据元素。除第一个和最后一个外，其他元素前后均只有一个数据元素（直接前趋和直接后继）。例如，字符串中的字符和学生成绩表中的前后元素间就存在着这种“一一对应”的关系。

本篇根据线性结构存储方式的不同，讲解了以下两种存储结构。

(1) *顺序存储结构*，即所谓的数组。

(2) *链式存储结构*，简称链表，包括单链表、双链表或循环链表。

根据元素存取方式的不同，本篇还讲解了以下两种特殊的线性结构。

(1) 栈，即“先进后出”的线性表。

(2) 队列，即“先进先出”的线性表。

这两种数据结构通常用数组实现。另外，本篇还介绍了线性结构的如下3个重要的应用。

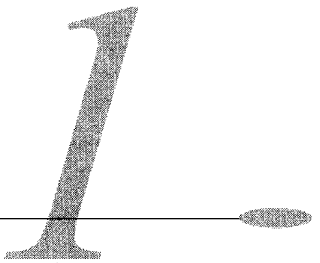
(1) 排序与顺序统计。本篇介绍了两种特别重要的排序方法：计数排序和快速排序，探讨了贪心策略中的序关系和分治策略在排序中的应用，并在排序知识的基础上阐释了顺序统计的有

关方法。

(2) 散列表。一种对关键字做某种运算后直接确定其存储位置，并可在常数平均时间内执行插入、删除和查找的技术。对于多个关键字运算结果相同的情况，即产生了所谓的“冲突”，本篇介绍了两种方法解决“冲突”：采用链式存储结构的分离链接法和采用顺序存储结构的开放定址法。

(3) 后缀数组。一种专用于处理字符串这种特殊类型数组的重要工具。

第 1 章 数 组



数组是程序设计中使用最多的数据结构，无论是线性结构的还是非线性结构的数据对象，都可用数组存储。

本章将揭示数组的物理地址与逻辑地址之间的线性对应关系，介绍使用数组存储各类数据对象的一般方法，并针对 3 种特殊的二维矩阵（规则矩阵、稀疏矩阵和 01 矩阵）提出压缩存储的优化方式。

在数组的应用中，我们顺便介绍了计数排序、采用二分策略的快速排序和合并排序，以及包含排序过程的贪心策略，阐释了顺序统计的基本方法。这些算法一般都是以数组为其存储结构的。

1.1 数组的基本概念

我们称有限个同类型数据元素的序列为数组，它是一种定长的线性表。若数据元素不再有分量，则称该序列为一维数组；若数据元素为数组，则称该序列为多维数组。例如，下列代码中，A 为一维数组，表长为 $d-c+1$，元素类型为 atype；B 为二维数组，表长为 $(d_1-c_1+1)(d_2-c_2+1)$，元素类型为 btype。

```
var
  A:array[c..d] of atype;
  B:array[c1..d1,c2..d2] of btype;
```

一维数组和多维数组的元素类型和元素个数需在程序的说明部分预先定义，编译系统按照数组说明预留静态数据区，执行过程中不能增减其内存空间。

1.1.1 数组是一种顺序存储结构

在数组存储结构中，元素依次用一组地址连续的存储单元存储，每个元素在序列中的逻辑位置由数组下标指明，映射了元素之间在物理地址上的相邻关系，因此数组亦称*顺序存储结构*。例如，一维数组 a 的长度为 $maxlen$，$a[i]$的内存地址为 $loc(a[i])$ $(1 \leqslant i \leqslant maxlen)$，元素的存储单位长度为 la。数组元素 $a[1]..a[n]$的存储情况如图 1-1 所示。

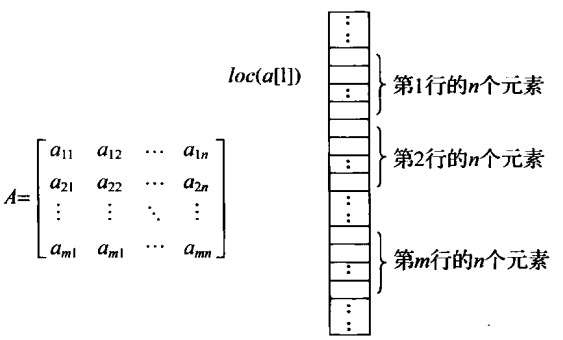
显然，$a[i]$的内存地址为 $loc(a[i]) = loc(a[1]) + (i-1)la$，其中 $loc(a[1])$为存储区的首地址，$(i\text{-}1)la$ 为偏移量。一维数组按照下标递增的顺序访问表中元素，即 $a[1] \rightarrow a[2] \rightarrow \cdots \rightarrow a[n]$。

存储地址	数组元素	元素在序列中的逻辑位置
$loc(a[1])$	$a[1]$	1
$loc(a[1]) + la$	$a[2]$	2
…		…
$loc(a[1]) + (i-1)la$	$a[i]$	i
…		…
$loc(a[1]) + (n-1)la$	$a[n]$	n
$loc(a[1]) + nla$		
…		
$loc(a[1]) + (maxlen-1)la$		

空闲

图 1-1 一维数组 $a[1..n]$的存储情况

二维数组一般是按照以行为主的顺序存储在计算机的连续存储空间中的，即先存储数组的第1行再存储数组的第2行，依次类推，每一行的元素按从左到右的顺序存储。例如，图 1-2a 所示为一个 $m \times n$ 的数组 A，图 1-2b 所示为数组 A 在计算机中的存储形式。



(a) $m \times n$的数组A (b) A的存储情况

图 1-2 数组 A 及其在计算机中的存储形式

设 $loc(a[i,j])$为元素 $a[i,j]$的内存地址（$1 \leqslant i \leqslant m$，$1 \leqslant j \leqslant n$），存储单位长度为 la，则

$$loc(a[i,j]) = loc(a[1,1]) + [(i-1)n + j - 1]\ la$$

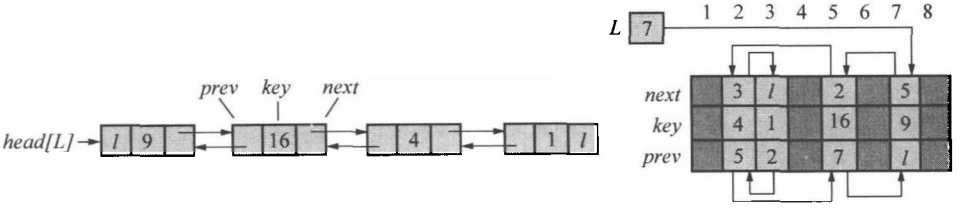
其中 $loc(a[1,1])$为存储区的首地址，$[(i-1)n + j - 1]la$ 为偏移量。二维数组按照先行后列的顺序访问表中元素，即 $a[1,1] \rightarrow a[1,2] \rightarrow \cdots \rightarrow a[1,n] \rightarrow \cdots \rightarrow a[i-1,n] \rightarrow a[i,1] \rightarrow \cdots \rightarrow a[m,n-1] \rightarrow a[m,n]$。

在数组中，元素的下标间接反映了元素的存储地址，而计算机内存是随机存取的装置，在数组中存取一个元素只要通过下标计算它的存储地址就行了，因此在数组中存取任意一个元素的时间都为 $O(1)$。从这个意义上讲，数组的存储结构是一种随机存取的结构。

1.1.2 数组是程序设计中使用频率最高的数据类型

数组在程序中几乎“无处不在”，无论是线性结构的还是非线性结构的数据对象，都可用数

组存储。例如，可以使用数组构造链表中的数据元素和指针。图 1-3a 所示为一个存储动态集合{1, 4, 9, 16}的双链表 L，表头为 $head[L]$，数据元素含关键字字段 *key*、前驱指针字段 *prev* 和后继指针字段 *next*，首元素的 *prev* 值和末元素的 *next* 值为 nil（用“/”表示）。双链表 L 也可以用记录类型数组 L 存储，其中 $L[i].key$ 存储节点 i 的关键字，$L[i].prev$ 和 $L[i].next$ 分别存储节点 i 的前驱指针和后继指针；或者分别用 3 个一维数组 *key*[]、*prev*[]和 *next*[]存储链表 L 中每个节点的关键字字段、前驱的数组下标和后继的数组下标。图 1-3b 给出了用数组 L 实现图 1-3a 中所示双链表 L 的情形，表头的下标为 7。有关链表的知识将在第 2 章中详细介绍。



(a) 动态指针实现的双链表　　(b) 数组实现的双链表

图 1-3　动态指针实现的双链表与数组实现的双链表

数组亦可存储非线性结构的数据对象。例如，一棵含 n 个节点的完全二叉树可用一维数组 $A[1..n]$存储，其中节点 $A[1]$为根，节点 $A[i]$（$2 \leqslant i \leqslant \left\lfloor \frac{n}{2} \right\rfloor$）的父节点为 $A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$，左儿子为 $A[2i]$，右儿子为 $A[2i+1]$，节点 $A\left[\left\lfloor \frac{n}{2} \right\rfloor+1\right]$至 $A[n]$为叶节点。图 1-4 给出了一棵含 14 个节点的完全二叉树。

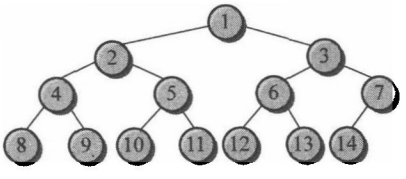


图 1-4　一棵含 14 个节点的完全二叉树的下标定义

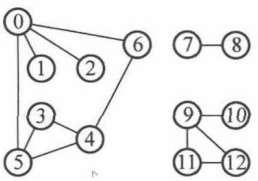
在完全二叉树中增删节点，可以在一维数组的基础上“原地操作”，不必另辟数组，不过其操作必须维护完全二叉树的特征。有关这方面的内容请参阅第 10 章。

数组也可以用来存储图，例如，用二维数组 $A[1..n, 1..n]$存储图的相邻矩阵是最典型的应用。图 1-5a 给出了一个无向图 G，其对应的相邻矩阵如图 1-5b 所示。

有关相邻矩阵的知识，请参阅 14.3.1 节。

如果一个图是稠密图（$|E|$比较接近 n^2）或需要及时判别任一对节点间是否存在连接边，通常采用相邻矩阵，但是如果一个图为稀疏图（$|E|$远小于 n^2）则通常采用邻接表。邻接表由一个包含 n 个列表的一维数组 *Adj* 组成，每个列表对应于图中的一个节点和一条含所有相邻点的链表，即对于每一个节点 $u \in V$，$Adj[u]$链表包含了图中所有与 u 相邻的节点。例如，图 1-6a 所示无向图

对应的邻接表如图 1-6b 所示。

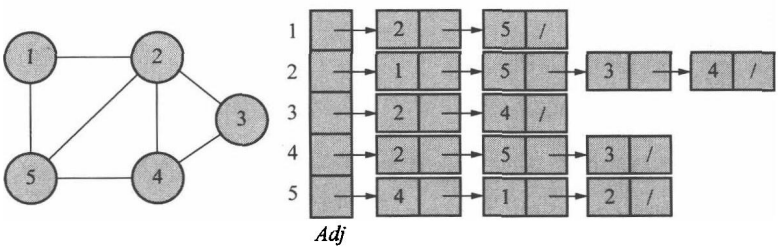


(a) 无向图

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

(b) 对应的相邻矩阵

图 1-5 示例无向图及其对应的相邻矩阵



(a) 无向图 (b) 邻接表

图 1-6 示例无向图及其对应的邻接表

由图 1-6 可见，每个邻接表为一个单链表，当然这个单链表亦可用数组取而代之。设 *d*[*u*]为节点 *u* 的度，*Adj*[*u*,1..*d*[*u*]]存储与节点 *u* 相邻的 *d*[*u*]个节点的序号。虽然这种方法的内存开销与相邻矩阵相同，但编程比单链表形式的邻接表方便一些，且运行速度比相邻矩阵要好一些。因为如果在相邻矩阵的基础上计算与 *u* 节点相连的所有边，则需要遍历 *u* 行的 *n* 个元素，而邻接表仅需遍历 *u* 行的前 *d*[*u*]个元素即可。有关相邻矩阵和邻接表的知识请参阅 14.3.2 节。

1.2 优化数组的存储方式

优化数组存储方式的目的是为了节省内存。下面我们分别针对二维数组中的以下 3 种特殊情况提出压缩存储的优化方式：

- 规则矩阵（元素分布有一定规律）；
- 稀疏矩阵（元素一半以上为无用 0）；
- 01 矩阵（元素值为 0 和 1）。

1.2.1 规则矩阵的压缩存储

规则矩阵是指矩阵中的非零元素都按照如下两种规律分布：

(1) 三角矩阵（非零元素仅在左下角（或右上角）出现，而右上角（或左下角）的元素均为 0）；

(2) 对称矩阵（左下角与右上角的元素对称，即 $a_{ij}=a_{ji}$）。

现实生活中的许多问题都可以抽象为规则矩阵。例如，由于存储无向图的相邻矩阵对称，仅需存储下三角矩阵（或上三角矩阵）即可，因此该矩阵为规则矩阵。这方面的知识详见 14.3 节。

规则矩阵的优化存储方式即为压缩存储，因为非零元素有规律地分布在规则矩阵中，因此只需压缩存储其中某个三角矩阵即可，不必存储另一个三角矩阵中的零元素或者因对称而重复的非零元素，从而节省了存储空间。一个 $n\times n$ 的规则矩阵仅需存储某个三角矩阵中的 $\frac{n(n+1)}{2}$ 个元素。当然这种优化的实施前提是，规则矩阵被压缩成三角矩阵后必须能够准确、方便地访问原规则矩阵中的每一个元素。设 n 阶的下三角矩阵为图 1-7a 中的矩阵，n 阶的对称矩阵为图 1-7b 中的矩阵，我们用一个长度为 $\frac{n(n+1)}{2}$ 的一维数组 b 来存储这两种规则矩阵中的下三角子矩阵，存储结构如图 1-7c 所示。

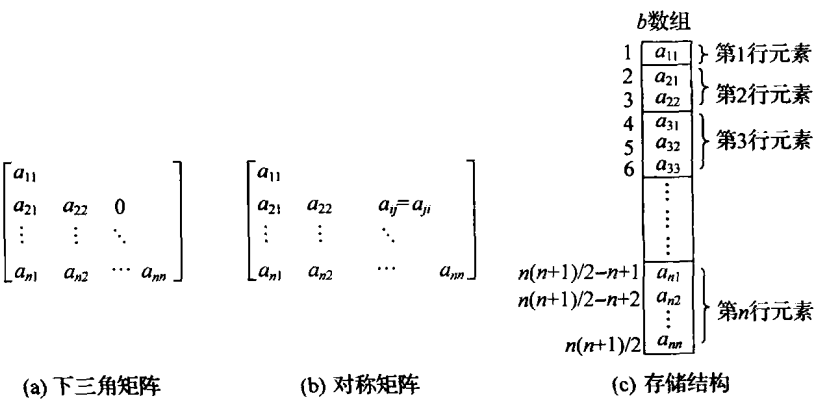


图 1-7 下三角矩阵、对称矩阵及其存储结构

若 a_{ij} 为下三角部分的元素（$j\leqslant i$），则 a_{ij} 存储在 $b[\frac{i(i-1)}{2}+j]$ 中（其中 $\frac{i(i-1)}{2}$ 为下三角部分前 $i-1$ 行的元素个数，j 为第 i 行的元素个数）；若 a_{ij} 为非下三角部分的元素（$j>i$），则 a_{ij} 未存入数组 b，a_{ij} 为 0。

$$a_{ij}=\begin{cases} b\left[\dfrac{i(i-1)}{2}+j\right] & (j\leqslant i) \\ 0 & (j>i) \end{cases}$$

规则矩阵是一种“用时间换空间”的策略，因为需要通过上述数学公式将 a 数组的二维下标转化为 b 数组的一维下标，所以需要一定的时间代价。

1.2.2 稀疏矩阵的压缩存储

若数组中绝大多数的元素值为零，仅极少数的元素值非零，则该矩阵为稀疏矩阵。例如，一

个7×8的稀疏矩阵中仅有8个非零元素，其余为零元素（图1-8）。

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 2 & 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

图1-8 7×8的稀疏矩阵

显然，若这些零元素也存入计算机，则会浪费大量的存储空间。因此，在方便访问每个元素（包括零元素）的前提下，仅存储少量的非零元素，这就是稀疏矩阵的压缩存储。

与规则矩阵的压缩存储不同的是，稀疏矩阵中的非零元素没有规律，不能采用一个一维数组来依次存放，因此稀疏矩阵的压缩存储方式比规则矩阵稍复杂一些。设数组有 m 行 n 列，其中非零元素的个数为 t，我们建立一个长度为 t+1 的记录类型数组 d。

```
type                                        /*d 数组的元素类型*/
  node=record                               /*非零元素的类型*/
    i,j,v:integer;                          /*非零元素的行列序号为 i 和 j，值为 v*/
  end;
  dtype=array[1..⌊(m*n)/2⌋]of node;         /*d 数组的元素类型定义*/
  arr=array[1..m] of integer;               /*pos 和 num 序列的类型定义*/
  arrtype= array[1..m,1..n] of integer;     /*稀疏矩阵的类型定义*/
var
  d:dtype;          /*非零元素序列*/
  pos,num:arr;      /*稀疏矩阵第 k 行中非零元素的个数为 num[k]。若该行有非零元素的话，则第 1 个非零
                      元素在 d 数组的序号为 pos[k]，即为 d[pos[k]]*/
```

数组 d 为压缩后的非零元素序列。其中 k>1 时，$d[k]$给出了第 k–1 个非零元素 a_{ij}的信息，即 $d[k].i=i$，$d[k].j=j$，$d[k].v=a_{ij}$；k=1 时，$d[1].i$ 和 $d[1].j$ 为数组的行和列数，$d[1].v$ 为非零元素的个数，即 $d[1].i=m$，$d[1].j=n$，$d[1].v=t$。

这样做的目的是为了确定唯一性，因为在稀疏矩阵的下方添加全0的若干行或在右方添加全0的若干列，得出的 d 序列也是相同的，而指明稀疏矩阵的规模和非零元素个数，则 d 对应的稀疏矩阵是唯一的。例如，用 d[1..9]存储图1-8所示的稀疏矩阵：

$d[1].i=7$ $d[1].j=8$ $d[1].v=8$
$d[2].i=1$ $d[2].j=1$ $d[2].v=3$
$d[3].i=1$ $d[3].j=8$ $d[3].v=1$
$d[4].i=3$ $d[4].j=1$ $d[4].v=9$
$d[5].i=4$ $d[5].j=5$ $d[5].v=7$
$d[6].i=5$ $d[6].j=7$ $d[6].v=6$
$d[7].i=6$ $d[7].j=4$ $d[7].v=2$
$d[8].i=6$ $d[8].j=6$ $d[8].v=3$
$d[9].i=7$ $d[1].j=3$ $d[1].v=3$

为了避免因访问稀疏矩阵中的某元素而遍历整个 d 序列的盲目性，附设了两个长度与稀疏矩阵行数相同的一维整数数组 pos 和 num，其中 $num[k]$给出了稀疏矩阵第 k 行中非零元素的个数。若该行有非零元素的话，则 $pos[k]$给出了其中第1个非零元素在 d 数组中的序号，即 $d[pos[k]]$存储 k 行第1个非零元素的信息。显然