第五章　数组和广义表

线性表：$L=(a_1, a_2, ..., a_n)$， a_i是同类型的数据元素，$1 \leqslant i \leqslant n$

数组：　$A=(a_1, a_2, ..., a_n)$

若a_i是同类型的数据元素，A是一维数组，$1 \leqslant i \leqslant n$

若a_i是同类型的定长线性表，A是多维数组，$1 \leqslant i \leqslant n$

广义表：$Ls=(a_1, a_2, ..., a_n)$

a_i可以是同类型的数据元素或广义表，$1 \leqslant i \leqslant n$

1

数组和广义表的特点：一种推广的线性表

① 数据元素的值并非原子类型，可以再分解，表中数据元素也是一个线性表（即广义的线性表）。

② 所有基本数据元素仍属同一数据类型。

2

5.1 数组的定义

数组是相同类型的数据的有限的、有序的组合。其特征有：

- 数组中各元素具有统一的类型；
- 数组一旦被定义，其维数和维界不再改变，有固定结构；
- 数组的基本操作比较简单。

5.1.1 数组的递归定义

1. 一维数组：

是一个定长线性表$(a_1, a_2, ..., a_n)$。

其中：a_i为数据元素，i为下标/序号，$1 \leqslant i \leqslant n$

$(a_1, a_2, ..., a_n)$又称为向量。

特征：1个下标，a_i 是a_{i+1}的直接前驱。

2. 二维数组

是一个定长线性表$(\alpha_1, \alpha_2, ..., \alpha_m)$，其中:$\alpha_i=(a_{i1}, a_{i2}, ..., a_{in})$为行向量，$1 \leqslant i \leqslant m$ ，由m个行向量组成，记作：

$$A_{m*n}=\begin{pmatrix} (\ a_{11} & a_{12} & ... & a_{1n}\ ) \\ (\ a_{21} & a_{22} & ... & a_{2n}\ ) \\ & .... & & \\ (\ a_{m1} & a_{m2} & ... & a_{mn}\ ) \end{pmatrix} \qquad A_{mxn}=\begin{pmatrix} a_{11} & a_{12} & \vdots & a_{1n} \\ a_{21} & a_{22} & \vdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \vdots & a_{mn} \end{pmatrix}$$

即 $A_{m*n}=((a_{11}\ a_{12} ... a_{1n}), (a_{21}\ a_{22} ... a_{2n}), ..., (a_{m1}\ a_{m2} ... a_{mn}))$

或由n个列向量组成，记作上右图的形式。

特征：

2个下标，每个元素$a_{i,j}$受两个关系（行关系和列关系）约束。

3. 三维数组

是一个定长线性表($\beta_1, \beta_2, \ldots, \beta_p$)，

其中：β_k=($\alpha_1, \alpha_2, \ldots, \alpha_m$)为定长二维数组，$1 \leqslant k \leqslant p$

例　三维数组A[1..3,1..4,1..2]，p=3，m=4，n=2

$$A_{3*4*2} = \underbrace{\begin{pmatrix} a_{111} & a_{112} \\ a_{121} & a_{122} \\ a_{131} & a_{132} \\ a_{141} & a_{142} \end{pmatrix}}_{\text{第1页}} \quad \underbrace{\begin{pmatrix} a_{211} & a_{212} \\ a_{221} & a_{222} \\ a_{231} & a_{232} \\ a_{241} & a_{242} \end{pmatrix}}_{\text{第2页}} \quad \underbrace{\begin{pmatrix} a_{311} & a_{312} \\ a_{321} & a_{322} \\ a_{331} & a_{332} \\ a_{341} & a_{342} \end{pmatrix}}_{\text{第3页}}$$

N维数组的特点：n个下标，每个元素受到n个关系约束。

一个**n**维数组可以看成是由若干个**n－1**维数组组成的线性表。

5

---

n维数组的抽象数据类型定义

ADT Array {

数据对象：$\mathbf{j_i = 0, \ldots, b_i\text{-}1}$，($1 \leqslant i \leqslant n$) $\mathbf{b_i}$为第**i**维的长度

$\mathbf{D = \{ a_{j_1, j_2, \ldots, j_n} \mid j_i}$为数组元素的第**i** 维下标，

$\mathbf{a_{j_1, j_2, \ldots, j_n} \in Elemset \}}$

数据关系：$\mathbf{R = \{ R_1, R_2, \ldots R_n \}}$，其中

$\mathbf{R_i = \{ < a_{j_1, j_2, \ldots, j_i, \ldots, j_n}, a_{j_1, j_2, \ldots, j_i+1, \ldots, j_n} > \mid}$

$\mathbf{a_{j_1, j_2, \ldots, j_i \ldots j_n}, a_{j_1, j_2, \ldots, j_i+1, \ldots, j_n} \in D \}}$

基本操作：

构造数组、销毁数组、读数组元素、写数组元素
（具体定义见p90）

}ADT Array

6

5.1.2 数组的类型定义和变量说明

例1
```
int a[10];          //10个整数的一维数组
char b[4][5];       //4行5列个字符的二维数组
float c[3][4][2];   //3*4*2个实数的三维数组
```

例2
```
#define m 4        //定义符号常量m
#define n 5        //定义符号常量n
int a[m];          //m个整数的一维数组
char b[m][n];      //m行n列个字符的二维数组
```

7

例3
```
#define m 4                  //定义符号常量m
#define n 5                  //定义符号常量n
typedef int ara[m];          //一维数组类型ara
typedef char arb[m][n];      //二维数组类型arb
ara a;                       //ara类型的变量a
arb b;                       //arb类型的变量b
```

C语言早期版本中定义静态数组时，元素数目必须是常量

错例1
```
int m=4,n=5;
int a[m][n];        //m,n是变量
```

错例2
```
int p;
scanf("%d",&p);
int c[p];           //p是变量
```

8

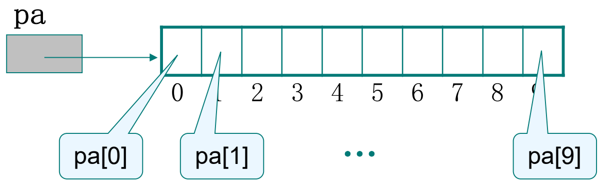
5.1.3 程序设计举例

例1
```
main()
  { int i,a[10];                    //生成一维数组a
    for (i=0; i<10; i++)
      scanf("%d",&a[i]);            //输入元素
    for (i=0; i<10; i++)
      printf("%d ",a[i]*a[i]);  //输出元素的平方
  }
```

退出时释放a

例2 生成动态的10个整数的一维数组

```
int  *pa;                                   //指针变量pa
pa=(int *)malloc(10*sizeof(int));   //动态数组pa
```



```
main()
{ int i,n,*pa;
  scanf("%d",&n);                      //动态输入n
  pa=(int *)malloc(n*sizeof(int)); //生成动态数组*pa
  for (i=0; i<n; i++)
    *(pa+i)=2*i;                 //指针法引用数组元素,赋值
  for (i=0; i<n; i++)
    printf("%d,",*(pa+i));       //输出数组元素0,2,4,6,...
  for (i=0; i<n; i++)
    scanf("%d",&pa[i]);          //下标法引用数组元素, 输入
  for (i=0; i<n; i++)
    printf("%d,",pa[i]);         //输出数组元素
  free(pa);                      //释放(销毁)数组空间
}
```

11

n维数组ADT复习

ADT Array {

数据对象: $j_i = 0, ..., b_i-1$, $(1 \leq i \leq n)$ b_i为第i维的长度

$D = \{ a_{j_1, j_2, ..., j_n} \mid j_i$为数组元素的第i 维下标 ,

$a_{j_1, j_2, ..., j_n} \in Elemset \}$

数据关系: $R = \{ R_1, R_2, ... R_n \}$, 其中

$R_i = \{ \langle a_{j_1, j_2, ..., j_i, ..., j_n}, a_{j_1, j_2, ..., j_i+1, ..., j_n} \rangle \mid$

$a_{j_1, j_2, ..., j_i, ..., j_n}, a_{j_1, j_2, ..., j_i+1, ..., j_n} \in D \}$

基本操作:

构造数组、销毁数组、读数组元素、写数组元素
(具体定义见p90)

}ADT Array

12

5.2 数组的顺序表示和实现

- ❑ 讨论：物理存储结构是一维的，怎样存放多维的数组？
  - ✓事先约定按某种次序将数组元素排成一个序列，然后将这个线性序列存入存储器中。
  - ✓在二维数组中，我们既可以规定按行存储，也可以规定按列存储。

注意：

- ❑ 若规定了次序，则数组中任一元素的存放地址便具有规律，可形成地址计算公式；
- ❑ 约定的次序不同，则计算元素地址的公式也有所不同；
- ❑ C和PASCAL中一般采用行优先顺序；FORTRAN采用列优先。

13

5.2.1 顺序表示(顺序存储结构)

1. 以行序为主序的顺序存储方式

   左边的下标后变化，右边的下标先变化

2. 以列序为主序的顺序存储方式

   左边的下标先变化，右边的下标后变化

例1 二维数组a[1..3,1..2],b是首地址,L是元素所占的单元数

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \\ a31 & a32 \end{pmatrix}$$

逻辑结构

序号	内存	地址
1	a11	b
2	a12	b+L
3	a21	b+2*L
4	a22	b+3*L
5	a31	b+4*L
6	a32	b+5*L

以行序为主序

序号	内存	地址
1	a11	b
2	a21	b+L
3	a31	b+2*L
4	a12	b+3*L
5	a22	b+4*L
6	a32	b+5*L

以列序为主序

14

例2 三维数组a[1..2,1..3,1..2]

$$\begin{pmatrix} a111 & a112 \\ a121 & a122 \\ a131 & a132 \end{pmatrix} \quad \begin{pmatrix} a211 & a212 \\ a221 & a222 \\ a231 & a232 \end{pmatrix}$$

第1页 第2页

逻辑结构

序号	内存	地址
1	a111	b
2	a112	b+L
3	a121	b+2*L
4	a122	b+3*L
5	a131	b+4*L
6	a132	b+5*L
7	a211	b+6*L
8	a212	
9	a221	
10	a222	
11	a231	
12	a232	b+11*L

以行序/高维为主序

序号	内存	地址
1	a111	b
2	a211	b+L
3	a121	b+2*L
4	a221	b+3*L
5	a131	b+4*L
6	a231	b+5*L
7	a112	b+6*L
8	a212	
9	a122	
10	a222	
11	a132	
12	a232	b+11*L

以列序/低维为主序

15

5.2.2 数组的映象函数

数组元素的存储地址公式。

例1 一维数组a[0..n-1]

a_0	a_1	a_2	. . .	a_i	. . .	a_{n-1}	

下标	0	1	2	i	n-1
地址	b	b+L	b+2*L	b+i*L	b+(n-1)*L

设：b为首地址，L为每个元素所占的存储单元数

则：元素a[i]的存储地址：

$$Loc(i)=Loc(0)+i*L=b+i*L \qquad 0 \leqslant i \leqslant n-1$$

16

例2 一维数组a[1..n]

	a_1	a_2	a_3	...	a_i	...	a_n	
下标	1	2	3		i		n	
地址	b	b+L	b+2L		b+(i-1)L		b+(n-1)L	

元素a[i]的存储地址

Loc(i)=Loc(1)+(i-1)*L=b+(i-1)*L $1\leqslant i\leqslant n$

17

例3 二维数组a[0..m-1,0..n-1]（有零行零列）

$$A_{m\times n}=\begin{pmatrix} a_{00} & \cdots & a_{0j} & \cdots & a_{0n-1} \\ a_{10} & \cdots & a_{1j} & \cdots & a_{1n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{i0} & \cdots & a_{ij} & \cdots & a_{in-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m-10} & \cdots & a_{m-1j} & \cdots & a_{m-1n-1} \end{pmatrix}$$

共i行

共j列

(1) 以行序为主序，a[i][j]的地址为

Loc(i, j)=Loc(0,0)+(n*i+j)*L

=b+(n*i+j)*L $0\leqslant i\leqslant m-1, 0\leqslant j\leqslant n-1$

a_{00}, …, $a_{0,n-1}$, a_{10}, …, $a_{1,n-1}$, …, $a_{m-1,0}$, …, $a_{m-1,n-1}$

18

例4 二维数组a[0..m-1,0..n-1]（有零行零列）

$$A_{mxn}=\begin{pmatrix} a_{00} & a_{01} \ldots & a_{0j} & \ldots & a_{0n-1} \\ a_{10} & a_{11} \ldots & a_{1j} & \ldots & a_{1n-1} \\ \ldots\ldots & \ldots & & \ldots\ldots & \\ a_{i0} & a_{i1} \ldots & a_{ij} & \ldots & a_{in-1} \\ \ldots\ldots & \ldots & \ldots\ldots & & \\ a_{m-10} & a_{m-11} \ldots & a_{m-1j} & \ldots & a_{m-1n-1} \end{pmatrix}$$

共i行

共j列

(2) 以列序为主序，a[i][j]的地址为

Loc(i, j)=Loc(0,0)+(m*j+i)*L

=b+(m*j+i)*L　　　0≤i≤m-1, 0≤j≤n-1

a_{00}, …, $a_{m-1,0}$, a_{01}, …, $a_{m-1,1}$, …, $a_{0,n-1}$, … $a_{m-1,n-1}$

19

例5 三维数组a[0..p-1,0..m-1,0..n-1],

(1) 以行序/高维为主序，a[k][i][j]的地址为

Loc(k, i, j)=Loc(0,0,0)+(m*n*k+n*i+j)*L

=b+(m*n*k+n*i+j)*L

0≤k≤p-1, 0≤i≤m-1, 0≤j≤n-1

其中：

p---页数　n--列数　m-行数

(2) 以列序/低维为主序，a[k][i][j]的地址为

Loc(k, i, j)=Loc(0,0,0)+(p*m*j+p*i+k)*L

=b+(p*m*j+p*i+k)*L

0≤k≤p-1, 0≤i≤m-1, 0≤j≤n-1

其中：

p---页数　n--列数　m-行数

a[1..2,1..3,1..2]

序号	内存	地址
1	a111	b
2	a211	b+L
3	a121	b+2*L
4	a221	b+3*L
5	a131	b+4*L
6	a231	b+5*L
7	a112	b+6*L
8	a212	
9	a122	
10	a222	
11	a132	
12	a232	b+11*L

以列序为主序

例6： 设数组a[1…60, 1…70]的基地址为2048，每个元素占2个存储单元，若以列序为主序顺序存储，则元素a[32,58]的存储地址为___8950___。

请注意审题！

根据列优先公式 $Loc(a_{ij})=Loc(a_{11})+[(j-1)*m+(i-1)]*L$

得： $LOC(a_{32,58})=2048+[(58-1)*60+(32-1)]*2 = 8950$

若数组是a[0…59, 0…69]，结果是否仍为8950？

维界虽未变，但此时的a[32,58]不再是原来的a[32,58]

21

对于 *n* 维数组，其中任一元素的地址该如何计算？

某一元素对应下标$(j_1, j_2, \dots, j_n)$，$0\le j_i \le b_i-1$，b_i为第i维的长度。

以行序为主序。

$LOC(j_1, j_2, \dots, j_n)$

$= LOC(0,0,\dots,0) + (b_n \times \dots \times b_2 \times j_1 + b_n \times \dots b_3 \times j_2 + \dots + b_n j_{n-1} + j_n)\, L$

$= LOC(0,0,\dots,0) + b_n \times \dots \times b_2 L \times j_1 + b_n \times \dots b_3 L \times j_2 + \dots + b_n L\, j_{n-1} + L j_n$

该式称为n维数组的映像函数。

若记： $c_n = L$，$c_{i-1} = b_i \times c_i$，$i=n, n-1, \dots, 2$, 则：

$$LOC(j_1, j_2, \dots, j_n) = LOC(0,0,\dots,0) + \sum_{i=1}^{n} c_i j_i$$

三维数组且列优先时的元素地址要会计算！

22

n维数组的顺序存储表示

```
#define MAX_ARRAY_DIM   8   //假设最大维数为8
 typedef struct{
     ELemType *base;        //数组元素基址
     int          dim;           //数组维数
     int         *bound;       //数组各维长度信息保存区基址
     int        *constants;    //数组映像函数常量的基址
  }Array;
```

即C_i信息保存区

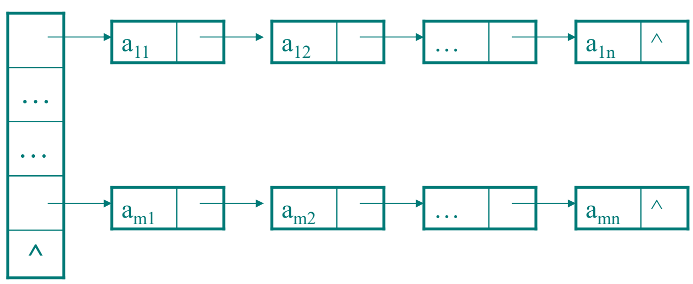
数组的基本操作函数说明（有5个）

（参阅严教材P93-95）

注意： 本章所讨论的数组与高级语言中的数组有所区别：高级语言中的数组只是顺序结构；而本章的数组既可以是顺序的，也可以是链式结构，用户可根据需要选择。

补充：二维数组的链式存储—用带行指针向量的单链表来表示。

行指针向量



注：链式数组的运算请参见“稀疏矩阵的转置”

5.3 矩阵的压缩存储

- 在某些大规模矩阵中，存在许多相同元素或零元素，如何实现压缩存储？
  - ✓ 多个值相同的元只分配一个存储空间；
  - ✓ 零元不分配存储空间。
- 特殊矩阵：值相同的元素或零元素的分布具有一定规律。如对称矩阵， 三角矩阵等。
- 稀疏矩阵：零元素占较大比例且分布不具有规律。

25

5.3.1 特殊矩阵的压缩存储

1. n阶对称矩阵

上三角

$$A_{nxn}=\begin{pmatrix} a_{11} & & a_{1n} \\ & a_{ji} & \\ & a_{ij} & \\ a_{n1} & & a_{nn} \end{pmatrix}$$

下三角

$a_{ij}==a_{ji}$

$1\leqslant i, j\leqslant n$

（1）只存储对称矩阵中上三角或下三角中的元素（含对角元），

（2）将n^2个元素压缩存储到$n(n+1)/2$个元素的空间中，以一个一维数组作为A的存储空间。

26

假定以行序为主，顺序存储下三角元素到SA[0..maxleng -1]

a_{11}	a_{21}	a_{22}	a_{31}	...	a_{i1}	...	a_{ij}	...	a_{ii}	...	a_{n1}	...	a_{nn}

k=0　1　2　3　...　　i(i-1)/2+j-1　...　　n(n+1)/2-1

如何求a_{ij}在SA中的位置，即序号k？

(1) 设a_{ij}在下三角，$i \geqslant j$

∵ 第1~i-1行共有元素

$1+2+3+\ldots+(i-1)=i(i-1)/2$ (个)

$a_{i1} \sim a_{ij}$共有j个元素

∴ a_{ij}的序号为:

$k=i(i-1)/2+j-1$

(2) 设a_{ij}在上三角，$i<j$

∵ 上三角的a_{ij} = 下三角的a_{ji}

下三角的a_{ji}的序号为

$k=j(j-1)/2+i-1$　　$i<j$

∴ 上三角的a_{ij}的序号为

$k=j(j-1)/2+i-1$　　$i<j$

由(1)和(2)，任意a_{ij}在SA中的序号，为

$$k(i,j)=\begin{cases} i(i-1)/2+j-1 & i \geqslant j \\ j(j-1)/2+i-1 & i<j \end{cases}$$

称为在SA中的映象函数，或下标转换公式。

2. 三对角矩阵

$$A_{nxn}=\begin{pmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & \text{全}0 & \\ & a_{32} & a_{33} & a_{34} & & \\ & & \cdots & a_{ij} & \cdots & \\ \text{全}0 & & & & a_{n-1n-1} & a_{n-1n} \\ & & & & a_{nn-1} & a_{nn} \end{pmatrix}$$

假定以行序为主，顺序存储非0元素到SA[1..maxleng]:

a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	...	a_{ij}	...	a_{nn-1}	a_{nn}
k=1	2	3	4	5	6	...	?	...		3n-2

三对角线上任意元素a_{ij}，在SA中的序号(从1开始）：

k= (3*(i-1)-1)+(j-i+2) = 2(i-1)+j

29

5.3.2 稀疏矩阵的压缩存储

问题：

如果只存储稀疏矩阵中的非零元素，那这些元素的位置信息该如何表示？

解决思路：

对每个非零元素增开若干存储单元，用来存放其所在的行号和列号，便可准确反映该元素所在位置。

实现方法：

将每个非零元素用一个三元组（i，j，a_{ij}）来表示，则每个稀疏矩阵可用一个三元组表来表示。

应用实例：关联规则挖掘中事务数据库可表示成稀疏布尔矩阵

30

例1： 三元素组表中的每个结点对应于稀疏矩阵的一个非零元素，它包含有三个数据项，分别表示该元素的 行下标 、 列下标 和 元素值 。

例2：写出右图所示稀疏矩阵的压缩存储形式。

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix}$$

解：至少有3种存储形式。

法1：用三元组顺序表表示

1：用三元组顺序表表示：

	i	j	value
0	6	6	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	5	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix}$$

注意：为方便处理，通常再加一行“总体”信息：即总行数、总列数、非零元素总个数。

稀疏矩阵压缩存储的缺点：

将失去随机存取功能 ！

2：用带辅助向量的三元组表表示。

用途：便于高效访问稀疏矩阵中任一非零元素。

方法：增加2个辅助向量：

① 记录每行非0元素个数，用NUM（i）表示；

② 记录稀疏矩阵中每行第一个非0元素在三元组中的行号，用POS（i）或 cpot（i）表示。

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix}$$

	i	j	v
0	6	6	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	5	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

i	1	2	3	4	5	6
NUM(i)	2	0	2	1	1	2
POS(i)	1	3	3	5	6	7

POS(i)如何计算？ 用途后续

POS(1)＝1

POS(i)＝POS(i-1)+NUM(i-1)

33

3：表示为 a sequence of sequences.

A sequence is a mathematical notion representing an ordered list of elements, such as ‹a， b， c， d›.

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix}$$

‹‹(2,12) ,(3,9)›, ‹ ›, ‹(1,-3) ,(5,14)›, ‹(3,24)›, ‹(2,18)›, ‹(1,15) ,(4,-7)››

- In this representation, each subsequence represents a row.
- Each of these rows contains only the non-zero elements each of which is stored as the non-zero value along with the column index in which it belongs.
- Hence the name compressed sparse row.
- Now lets consider how we do a vector matrix multiply using this representation.

34

$$\underbrace{\begin{pmatrix} 3 & 0 & 0 & -1 \\ 0 & -2 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 0 & -3 & 1 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} 2 \\ 1 \\ 3 \\ 1 \end{pmatrix}}_{\mathbf{x}}$$

```
A = <<(0,3),(3,-1)>,<(1,-2),(2,2)>,<(2,1)>,<(0,2),(2,-3),(3,1)>

x = < 2, 1, 3, 1 >
```

- If we assume an array implementation of sequences then this routine will take O(nz(A)) work and O(logn) span.
- To calculate the work we note that we only multiply each non-zero element once and each non-zero element is only involved in one reduction.
- For the span we note that the sum requires a reduction but otherwise everything.

35

4：用十字链表表示

用途：方便稀疏矩阵的**加减**运算

方法：每个非**0**元素占用**5个域**

i	j	e
down		right

同一列中下一非零元素的指针

同一行中下一非零元素的指针

十字链表的特点：

①每行非零元素链接成单链表或带表头结点的循环链表；

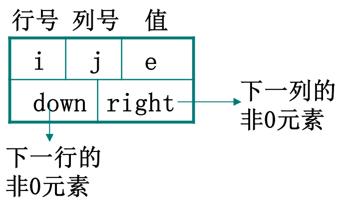
②每列非零元素也链接成单链表或带表头结点的循环链表。

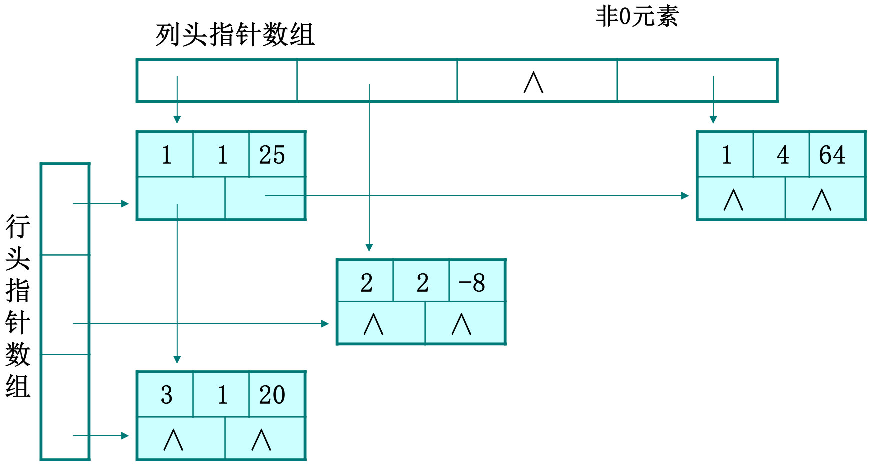
则每个非零元素既是行（循环）链表中的一个结点；

又是列（循环）链表中的一个结点，即呈十字链状。

36

例 稀疏矩阵 $M=\begin{pmatrix} 25 & 0 & 0 & 64 \\ 0 & -8 & 0 & 0 \\ 20 & 0 & 0 & 0 \end{pmatrix}$





37

三元组顺序表的存储表示

```
#define MAXSIZE 125000   //设非零元素最大个数125000
typedef struct{
 Triple data[MAXSIZE+1]; //三元组表，以行为主序存入一维
                           向量data[ ]中， data[0]未用
 int mu;            //矩阵总行数
 int nu;            //矩阵总列数
 int tu;            //矩阵中非零元素总个数
}TSMatrix;
```

```
typedef struct{
 int i;              //元素行号
 int j;              //元素列号
 ElemType e;         //元素值
}Triple;
```

对表中每个结点的结构定义

38

二、稀疏矩阵的操作（以转置运算为例，加减用十字链表）

目的：

$$M=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix} \xrightarrow{\text{转置后}} \begin{pmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} T$$

已知三元组表 a.data
(1, 2, 12)
(1, 3, 9)
(3, 1, -3)
(3, 5, 14)
(4, 3, 24)
(5, 2, 18)
(6, 1, 15)
(6, 4, -7)

求三元组表 b.data
(1, 3, -3)
(1, 6, 15)
(2, 1, 12)
(2, 5, 18)
(3, 1, 9)
(3, 4, 24)
(4, 6, -7)
(5, 3, 14)

提问： 若采用三元组顺序表存储稀疏矩阵，是否只要把每个元素的行下标和列下标互换，就完成了对该矩阵的转置运算吗？

答：不正确！

除了：（1）每个元素的行下标和列下标互换（即三元组中的i和j互换）；

还需要：（2）T的总行数mu和总列数nu也要互换；

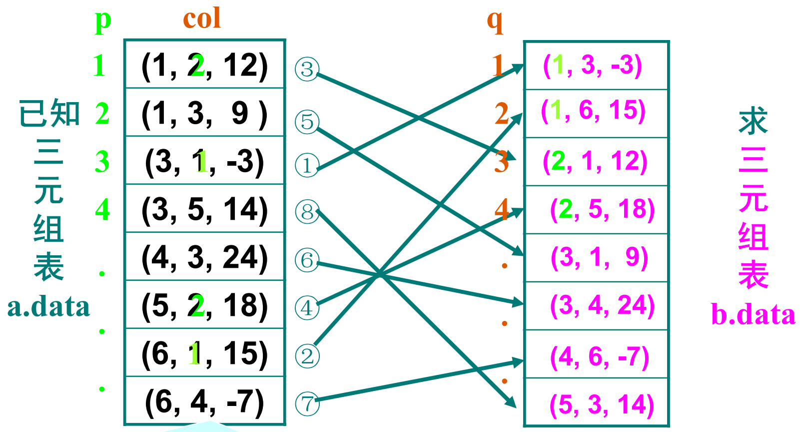
（3）重排三元组内各元素顺序，使转置后的三元组也按行（或列）为主序有规律的排列。

上述（1）和（2）容易实现，难点在（3）。

有两种实现转置的方法：
- 压缩转置
- 快速(压缩)转置

方法1：压缩转置

思路：反复扫描a表（记为a.data）中的列序，
从j=1～n依次进行转置。



每个元素的列分量表示为：**a.data[p].j**

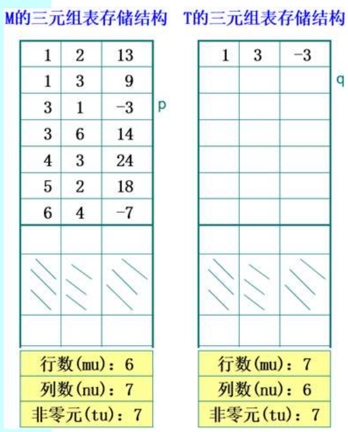
三元组顺序表的存储表示复习

```
#define MAXSIZE 125000   //设非零元素最大个数125000
typedef struct{
 Triple data[MAXSIZE+1]; //三元组表，以行为主序存入一维
                          向量data[ ]中， data[0]未用
 int mu;          //矩阵总行数
 int nu;          //矩阵总列数
 int tu;          //矩阵中非零元素总个数
}TSMatrix;
```

```
typedef struct{
 int i;           //元素行号
 int j;           //元素列号
 ElemType e;      //元素值
}Triple;
```

对表中每个结点的结构定义

```
T.mu=M.nu;T.nu=M.Mu; T.tu=M.tu;
if (T.tu)  {
  q=1;       /*指示向T写时的位置*/
  for(col=1;col<=M.nu; ++col)
    for(p=1;p<=M.tu;++p) //扫描三元组表
      if(M.data[p].j==col) /*当前行*/
          {T.data[q].i=M.data[p].j;
           T.data[q].j=M.data[p].i;
           T.data[q].e=M.data[p].e;
           q++; }
    }
return OK;
```

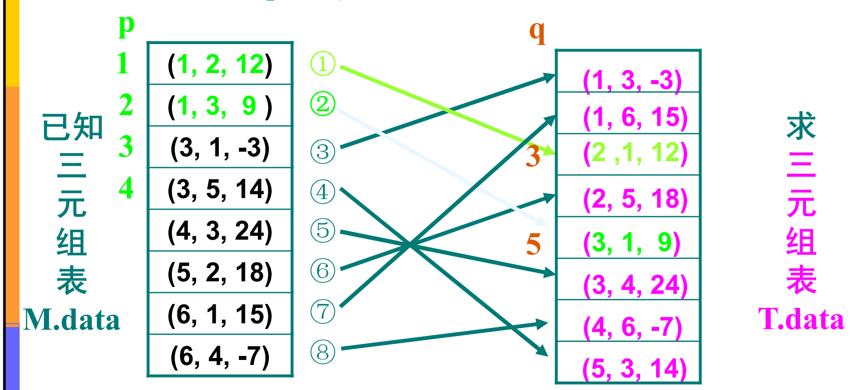


算法5.1:
时间复杂度:
O(nu*tu)

43

方法2 快速转置

思路：依次把M.data中的元素直接送入T.data的恰当位置上（即M三元组的p指针不回溯）。



关键：怎样寻找T.data的"恰当"位置？

44

设计思路:

如果能预知M矩阵每一列(即T的每一行)的非零元素个数，
又能很快得知第一个非零元素在T.data中的位置，
则扫描M.data时便可以将每个元素准确定位(因已知若干参考点)

请注意M.data特征：每列首个非零元素必定先被扫描到。

技巧：为实现转置运算，应当按列生成 M 矩阵三元组表的两个辅助向量，让它表示每列非零元素个数 NUM(j) 以及每列的第一个非零元素在T的三元组表中的位置POS(j) 等信息。

辅助向量的样式：

i	1	2	3	4	5	6
NUM(i)	2	0	2	1	1	2
POS(i)	1	3	3	5	6	7

计算式： POS(1)＝1

POS(i)＝POS(i-1)+NUM(i-1)

45

令: M矩阵中的列变量用col表示;
num[col]: 存放M中第col 列中非0元素个数
cpos[col]: 存放M中第col列第一个非0元素转置后的位置
（即T.data中待计算的“恰当”位置所需参考点）

col	1	2	3	4	5	6
num[col]	2	2	2	1	1	0
cpot[col]	1	3	5	7	8	9

$$
\begin{array}{c|cccccc}
\text{col} & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
M & \multicolumn{6}{c}{\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix}}
\end{array}
$$

计算式： cpos(1)＝1

cpos[col] ＝ cpos[col-1] + num[col-1]

讨论：求出按列优先的辅助向量后，如何实现快速转置？

由M.data中每个元素的列信息，可以直接从辅助向量cpos[col]中查出在T.data中的“基准”位置，进而得到当前元素的位置。

46

快速转置算法描述：

```
Status FastTransposeSMatrix（TSMatirx M, TSMatirx &T）
{             //M是顺序存储的三元组表，求M的转置矩阵T
T.mu = M.nu ;T .nu = M.mu ; T.tu = M.tu ;
 if ( T.tu ) {
for(col = 1; col <=M.nu; col++) num[col] =0;  //每列非零元素个数初始化
for( i = 1; i <=M.tu; i ++) {col =M.data[i] .j ; ++num [col] ;}
cpos[ 1 ] =1;                            //再生成每列首元位置辅助向量
for(col = 2; col <=M.nu; col++)  cpos[col ]=cpos[col-1]+num [col-1 ] ;
for( p =1; p <=M.tu ; p ++ )       //p指向a.data，循环次数为非0元素总个数tu
 { col =M.data[ p ]. j ; q =cpos [ col ];  //查辅助向量得q，即T中位置
        T.data[q].i = M.data[p]. j;
        T.data[q].j = M.data[p]. i;
        T.data[q]. value = M.data[p]. value;
        + + cpos[col] ;
  } //for
} //if
return OK;
} //FastTranposeSMatrix;
```

前3个for循环用来产生两个辅助向量

元素转置

重要！修改辅助向量内容，预备给同列的下一非零元素定位之用。

47

快速转置算法的效率分析：

1. 与常规算法相比，附加了生成辅助向量的工作。增开了2个长度为列长的数组(num[ ]和cpos[ ]）。
2. 从时间上，此算法用了4个并列的单循环，而且其中前3个单循环都是用来产生辅助向量的。

```
for(col = 1; col <=M.nu; col++){ };    循环次数 = nu（列数）;
for( i = 1; i <=M.tu; i ++) { };       循环次数 = tu （非0元素个数）;
for(col = 2; col <=M.nu; col++) { };   循环次数 = nu;
for( p =1; p <=M.tu ; p ++ ) { };      循环次数 = tu;
```

该算法的时间复杂度 = nu+tu+nu+tu=O(nu+tu)

讨论：最坏情形是矩阵中全为非零元，此时tu=nu*mu 时间复杂度是O(mu*nu)，未超过传统转置(未压缩)算法的复杂度。

小结：传统转置：O(mu*nu)　　压缩转置：O(nu*tu)
压缩快速转置：O(nu+tu)

增设辅助向量，牺牲空间效率换取时间效率。

5.4 广义表的定义

1、定义：

广义表是线性表的推广，也称为列表（lists）

记为： $LS = (\underbrace{a_1}_{\text{表头(Head)}}, \underbrace{a_2, \ldots\ldots, a_n}_{\text{表尾 (Tail)}})$

LS：广义表名

n是表长

在广义表中约定：

① 第一个元素是表头，而其余元素组成的表称为表尾；

② 用小写字母表示原子类型，用大写字母表示列表。

讨论：广义表与线性表的区别和联系？

广义表中元素既可以是原子类型，也可以是列表；

当每个元素都为原子且类型相同时，就是线性表。

49

2、特点：

- 有次序性　一个直接前驱和一个直接后继
- 有长度　＝表中元素个数
- 有深度　＝表中括号的重数
- 可递归　自己可以作为自己的子表
- 可共享　可以为其他广义表所共享

特别提示：

任何一个非空表，表头可能是原子，也可能是列表；

但表尾一定是列表！

50

例1： 求下列广义表的长度。

1） A =( )　　n=0，因为A是空表

2） B = (e)　　n=1，表中元素e是原子

3） C =(a ,(b , c , d))　　n=2，a 为原子，(b, c, d)为子表

4） D=(A , B ,C)　　n=3，3个元素都是子表

5） E=(a, E)　　n=2，a 为原子，E为子表

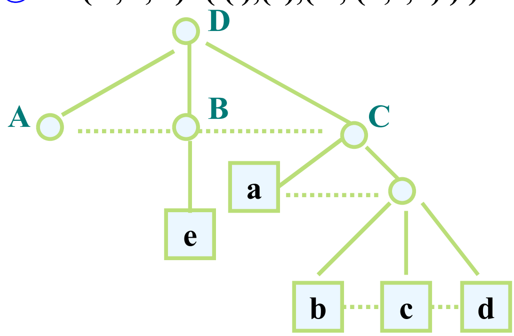
D=(A,B,C)=(( ),(e),(a,(b,c,d)))，共享表

E=(a,E)=(a,(a,E))= (a,(a,(a,…….)))，E为递归表

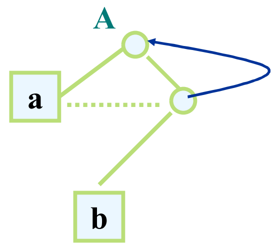
51

例2： 试用图形表示下列广义表.
（设○代表子表，e 代表元素）

① D=(A,B,C)=(( ),(e),(a, (b,c,d)))



② A=(a ，(b, A))



①的长度为3，深度为3

深度＝括号的重数＝○结点的层数

②的长度为2，深度为∞

52

广义表的抽象数据类型定义见教材

介绍两种特殊的基本操作：

GetHead (L) ——取表头(可能是原子或列表)

GetTail (L) ——取表尾(一定是列表)

例：求下列广义表操作的结果

1. GetTail (b, k, p, h)＝___(k, p, h)___;
2. GetHead ((a,b), (c,d))＝___(a,b)___ ;
3. GetTail ((a,b), (c,d))＝___((c,d))___ ;
4. GetTail （ GetHead((a,b),(c,d)) ）＝___(b)___ ;
5. GetTail（e）＝___( )___;
6. GetHead （( ))＝___( )___ .
7. GetTail (( )) ＝___( )___ .

(a,b)

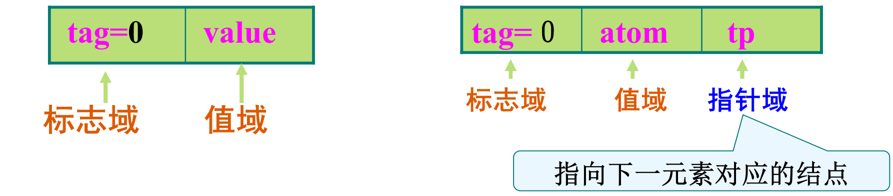
5.5 广义表的存储结构

由于广义表的元素可以是不同结构（原子或列表），难以用顺序存储结构表示 ，通常用链式结构，每个元素用一个结点表示。

注意：广义表的“元素”还可是列表，所以结点可能有两种形式

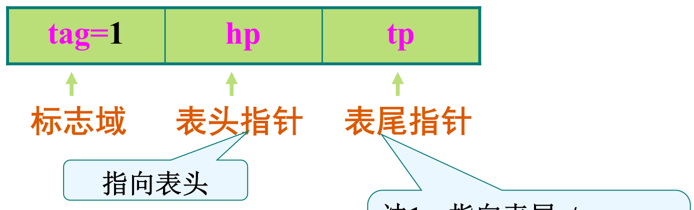
1. 原子结点: 表示原子，有两种表示法，可设2个域或3个域。

法1：标志域，数值域 　　　　法2：标志域、值域、指针域



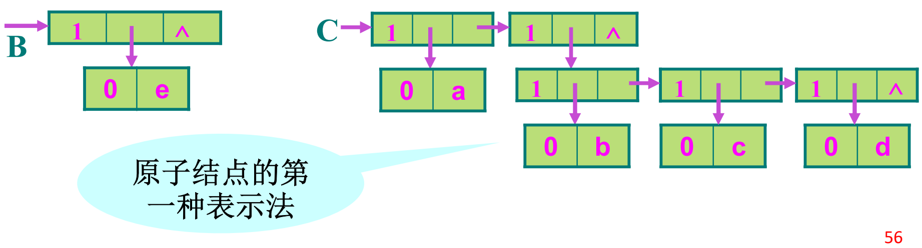
55

2. 表结点：表示列表，若表不空，则可分解为表头和表尾，用3个域表示：标志域，表头指针，表尾指针。



例： ① A =( ) 　　A=NULL

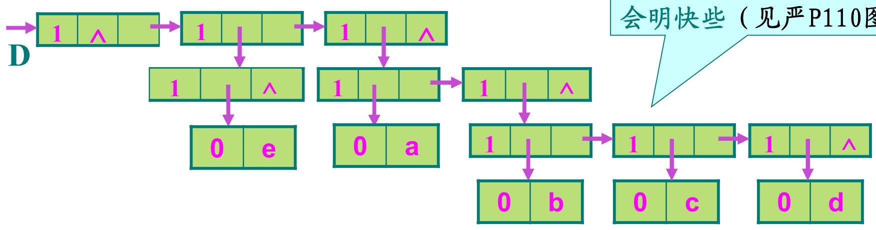
② B=(e) 　　③ C =(a ,(b , c , d))



56

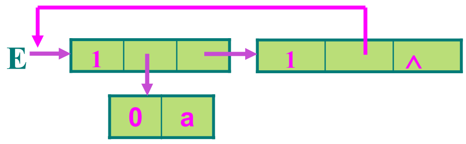
④ D=(A , B ,C)=(( ),(e),(a,(b,c,d)))

原子结点的第一种表示法比较麻烦，若采用第二种会明快些（见严P110图）



（参见严教材P109图）

⑤ E=(a, E)



57

广义表的头尾链表存储结构（法1）：

原子结点:

tag=0	atom(元素)

列表结点:

tag=1	hp(表头）	tp(表尾)

```
typedef  enum{ATOM, LIST}ElemTag;
                    //ATOM==0:原子, LIST==1:子表
typedef  struct GLNode{
    ElemTag  tag;
    union { AtomType atom;
            struct { struct GLNode    *hp,*tp; }ptr;
          }
} *GList;
```

第二种表示法与第一种表示法有哪些不同点？

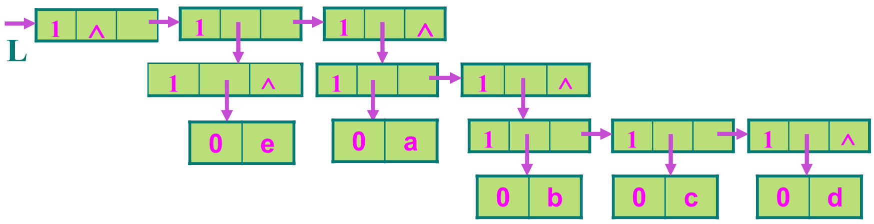
58

基于广义表头尾链表存储结构（法1）的元素存取：

```
typedef  struct GLNode{
   ElemTag  tag;
   union { AtomType atom;
           struct { struct GLNode    *hp, *tp; }ptr;
          }
} *GList;
```

- 广义表 $LS=(a_1, a_2,\cdots, a_i,\cdots,a_n)$（$1\leqslant i\leqslant n$）
- LS 的链表头指针为GList L，如何存取元素a_2？

LS=(( ),(e),(a,(b,c,d)))



广义表操作的递归算法

- 求广义表的深度：int GLisitDepth(Glist L)
  - ✓广义表 $LS=(a_1,\cdots,a_i,\cdots,a_n)$（$1\leqslant i\leqslant n$）
  - ✓深度定义为广义表括号的重数
  - ✓递归思想：全局问题化为子问题求解（每个元素a_i对应的广义表深度最大值+1）
  - ✓递归出口：一个空广义表的深度为 1；
    或当子问题为原子结点时，深度为 0；

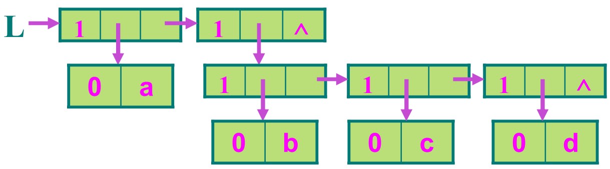
$$GListDepth(L)=\begin{cases}1 & L\text{是空广义表；}\\ 0 & L\text{是原子节点；}\\ 1+max(a_i\text{的深度}) & L\text{是非空广义表；}\end{cases}$$

广义表操作的递归算法

```
typedef struct GLNode{
   ElemTag  tag;
   union { AtomType atom;
           struct{ struct GLNode *hp,*tp;
                  }ptr;
         }
   } *GList;
```

```
int GLisitDepth(Glist L)
{
   if (!L) return 1;
   if (L->tag==0) return 0;
   for (max=0,p=L; p; p=p->ptr.tp)
   {
      depth=GLisitDepth(p->ptr.hp);
         //每层表尾的表头对应元素ai结点
      if (depth>max) max=depth;
   }
   return max+1;
}
```

L =(a ,(b , c , d))



61

本章小结

1. 数组可视为一种广义线性表；
2. 数组的存储有行/低址优先和列/高址优先两种不同的顺序；
3. 稀疏矩阵的压缩存储（三元组表与十字链表）和运算方法；
4. 广义表（列表）是线性表的推广，也是一种线性结构；
5. 任何一个非空表，表头可能是原子，也可能是列表；但表尾一定是列表。
6. 广义表的链式存储结构（掌握方法1：头尾链表存储结构）。
7. 广义表的递归算法（有兴趣的同学可学习MOOC 5.7对应内容）

62

课外思考：

严数据结构题集中

5.1　　5.18

5.23（课外思考题）