中国科学技术大学
University of Science and Technology of China

数据结构

数组与广义表（4 学时）

October 28, 2022

目录

认识数组

认识

- 可以看成是线性表的推广
- 几乎所有的程序设计语言都支持这种数据结构或将这种数据结构设定为语言的固有类型
- 数组是一组 (下标值，数据元素值) 构成的集合
- 在数组中，对于一组有意义的下标，都存在一个与其对应的值。一维数组对应着一个下标值，二维数组对应着两个下标值，如此类推

定义

- 数组是由 $n(n>1)$ 个具有相同数据类型的数据元素 $a_1, a_2, \ldots, a_n$ 组成的有序序列，且该序列必须存储在一块地址连续的存储单元中。
  - 数组中的数据元素具有相同数据类型。
  - 数组是一种随机存取结构，即：给定一组下标，就可以直接访问与其对应的数据元素。
  - 数组中的数据元素个数是固定的。

数组的 ADT

```
ADT Queue{
    数据对象: j_i ∈ {0, 1, ..., b_{i-1}}, i = 1, 2, ..., n
              D = {a_{j1 j2 ... jn} | n(> 0)为数组的维数,
              b_i为第i维的长度, j_i是数组第i维下标, a_{j1 j2 ... jn} ∈ ElemSet}

    数据关系: R = {R_1, R_2, ..., R_n}, R_i = {< a_{j1 ... ji ... jn}, a_{j1 ... ji+1 ... jn} >}
              0 ≤ j_k ≤ b_k − 1, 1 ≤ k ≤ n, k ≠ i, 0 ≤ j_i ≤ b_i − 2
              a_{j1 ... ji ... jn}, a_{j1 ... ji+1 ... jn} ∈ D

    基本操作:
        InitArray(&A,n,bound1,···,boundn) //初始化
        DestroyArray(&A)      //销毁

        Value(A,&e,index1,···,indexn) //读数组元素
        Assign(&A,e,index1···,indexn) //写数组元素
}ADT Array
```

数组与数组的下标

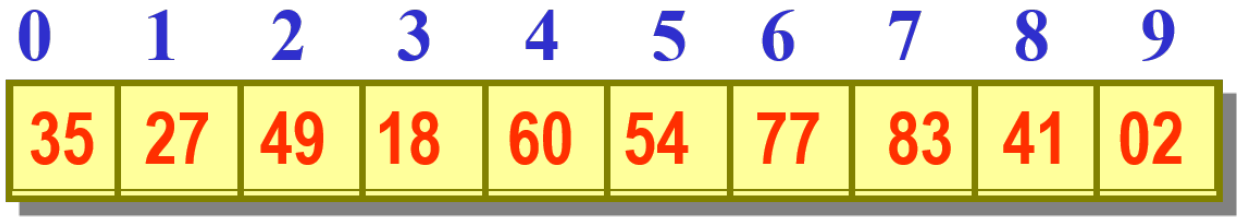
基本性质

- 数据元素个数为 $\Pi_{i=1}^{n} b_i, n(>3)$ 时可想象成超长方体
- 每个元素都受着 n 个关系的约束，即每一维都存在序偶关系
- $n=1$ 时，n 维数组退化为定长的线性表
- n 维数组也可看为线性表的推广，如：
  - 二维数组视为一个定长线性表，其每个数据元素是一个定长线性表
  - 三维数组可看为一个定长线性表，其每个数据元素是一个二维数组
  - ...
  - n 维数组视为一个定长线性表，其每个数据元素是一个 $n-1$ 维数组

我们关心多维数组在线性内存中的存储方式

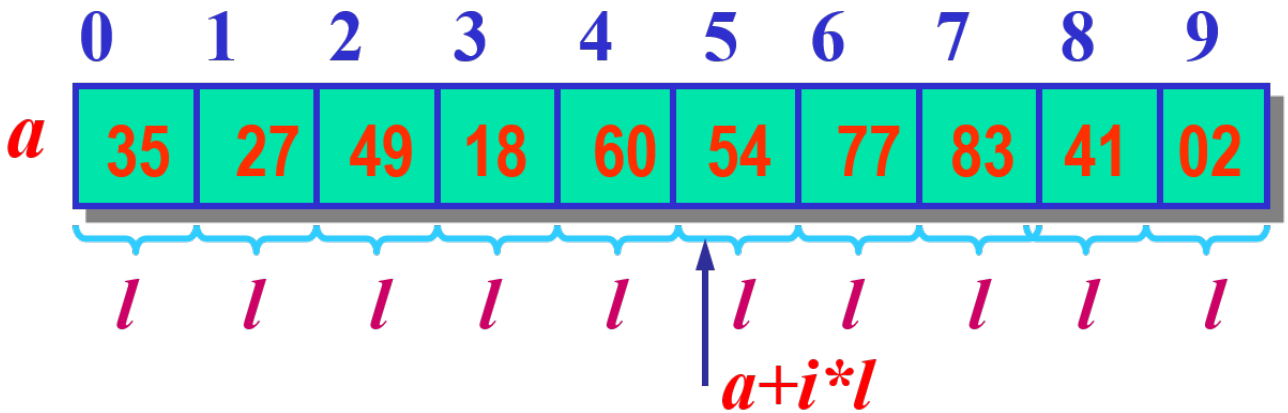
一维数组

例子



存储方式

$$LOC(i) = \begin{cases} a, & i = 0 \\ LOC(i-1)+l = a+i*l, & i > 0 \end{cases}$$



$$LOC(i) = LOC(i-1)+l = a+i*l$$

二维数组

存储方式

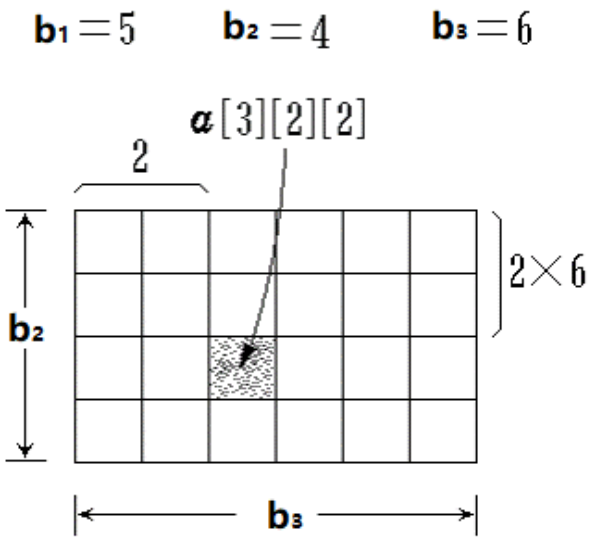
- 行优先存放：设数组开始存放位置 $LOC(0,0) = a$, 每个元素占用 L 个存储单元
- $LOC(i,j) = a + (n * i + j) * L$

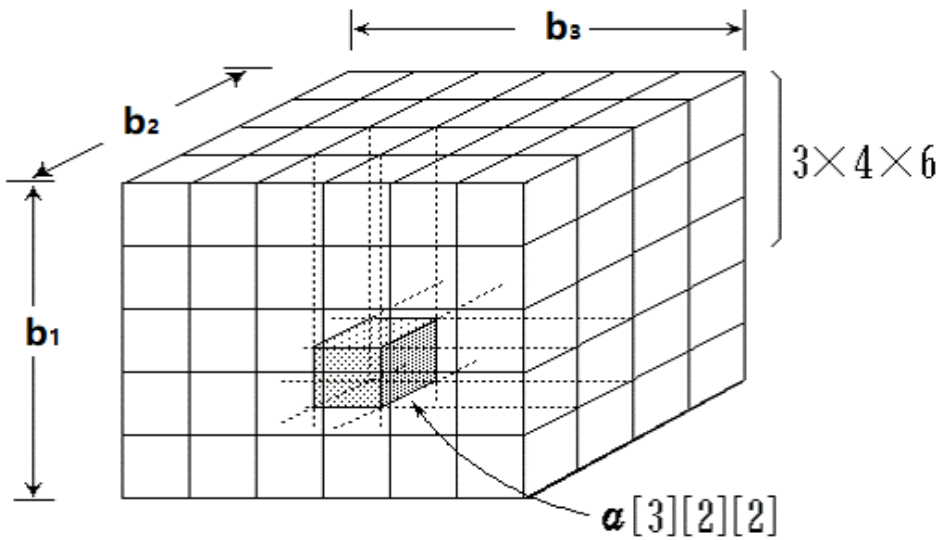
$$
\boldsymbol{A}=\begin{pmatrix}
\boldsymbol{a[0][0]} & \boldsymbol{a[0][1]} & \cdots & \boldsymbol{a[0][n-1]} \\
\boldsymbol{a[1][0]} & \boldsymbol{a[1][1]} & \cdots & \boldsymbol{a[1][n-1]} \\
\boldsymbol{a[2][0]} & \boldsymbol{a[2][1]} & \cdots & \boldsymbol{a[2][n-1]} \\
\vdots & \vdots & \ddots & \vdots \\
\boldsymbol{a[m-1][0]} & \boldsymbol{a[m-1][1]} & \cdots & \boldsymbol{a[m-1][n-1]}
\end{pmatrix}
$$

三维数组

存储方式

- 各维元素个数为 b_1, b_2, b_3
- 下标为 j_1, j_2, j_3 的数组元素的存储地址：（按页/行/列存放）
- $LOC(j_1, j_2, j_3) = a + (b_2 * b_3 * j_1 + b_3 * j_2 + j_3) * L$，其中 $b_2 * b_3 * j_1$ 是前 j_1 页元素的总个数，$b_3 * j_2$ 是第 j_1 页前 j_2 行元素的总个数
- 右下图给出了三维数组的存储结构，左下图是其第 3 页的切面，是三维数组的一个元素（二维数组）





n 维数组

存储结构

- 映象函数：给定 n 维数组的下标，计算存储位置。计算方式如下：
- $LOC(j_1, j_2, \ldots, j_n) =$
  $a + (b_2 * b_3 * \ldots * b_n * j_1 + b_3 * b_4 * \ldots * b_n * j_2 + \ldots + b_n * j_{n-1} + j_n) * L$
- 令 $c_n = L, c_{i-1} = c_i * b_i, i = n-1, n-2, \ldots, 2$，则映象函数改写成：
- $LOC(j_1, j_2, \ldots, j_n) = a + \Sigma_{i=1}^{n} c_i j_i$

思考

- 为什么要引入 c_i？(提示：从实现的时间开销考虑)。我们将 c_i 取个名字“第 i 维的地址系数”
- 理解“随机存取”与存储结构的关系

数组的顺序实现

数组在内存中的表示：静态结构

```
1    Typedef struct{
2        ElemType *base; //数组元素基址
3        int dim;              //数组维数
4        int * bounds;     //数组维界基址
5        int * constants; //数组映象函数常量基址
6    }Array;
```

补充知识

C 语言实现参数个数可变的函数

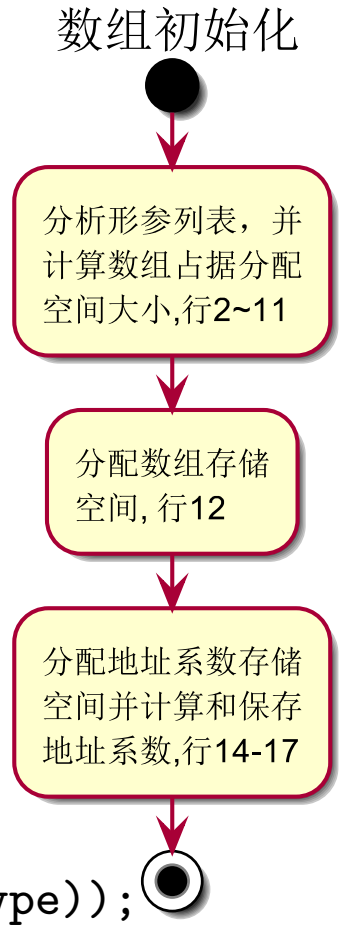
- 例如：printf(), scanf()，函数参数个数可变，如何编码实现？
- 利用三个 macros：va_start(), va_arg(), va_end()，参考 http://www.cplusplus.com/reference/cstdarg/

```
1  #include <stdio.h>
2  #include <stdarg.h>
3  //利用宏定义：va_list, va_start, va_arg, va_end
4  //例子：实现打印n个实数的功能，n可变
5  void PrintFloats(int n,...){//0个或多个固定参数+省略号表示的变参
6  //通常'...'表示的变参列表放在最后,其前用个固定参数n来表示变参个数,方便处理
7    int i; double val;
8    va_list vl;
9    va_start(vl,n);//从本函数的参数中读取变参表到vl中,va_start(vl, last_arg)
10   for (i=0;i<n;i++) { //用循环依次读取n个参数
11     val=va_arg(vl,double);//将变参表中下一个参数以double类型读取
12     printf ("[%.2f]",val); //打印输出
13   }
14   va_end(vl);//与va_start配对使用，相当于右括号
15 }//这些宏是怎么回事？用gcc -E 完成预编译，查看宏定义展开后的代码
```

数组的顺序实现

数组的顺序实现：初始化（变参函数）

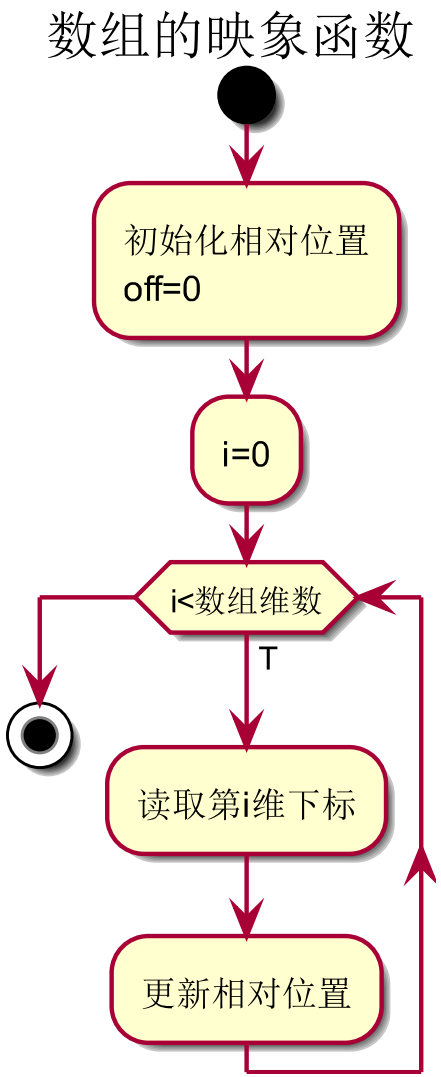
```
1  Status InitArray(Array &A,int dim,…){
2        A.dim=dim;
3        A.bounds =(int *) malloc(dim* sizeof(int));
4       //计算并分配数组元素空间
5       elemtotal=1;
6       va_start(ap,dim);
7       for (i=0;i<dim;++i){
8             A.bounds[i]=va_arg(ap,int);
9             elemtotal *= A.bounds[i];
10      }
11      va_end(ap);
12      A.base = (ElemType *)malloc(elemtotal*sizeof(ElemType));
13      //求映象函数的常数ci,并存入A.constants[i-1]
14      A.constants = (int *) malloc (dim * sizeof(int));
15      A.constants[dim-1]=1;    // cn=L
16      for (i =dim-2; i>=0;--i)   // ci-1=bi*ci,
17           A.constants[i] = A.bounds[i+1]*A.constants[i+1];
18      return OK;
19 }
```



数组的顺序实现

数组的顺序实现：映象函数 (变参函数)

```
1  Status Locate(Array A,va_list ap,int & off){
2  //根据ap指示的各下标值，求出对应元素相对地址off
3    off=0;
4    for ( i=0;i<A.dim;++i){
5      ind = va_arg(ap,int);
6      //用映象函数求和值
7      off += A.constants[i]* ind;
8    }
9      return OK;
10 }
```



思考题

多维数组该如何存储，算法效率更高？

- 考虑二维数组：行优先？列优先？
- 矩阵乘法？
- 矩阵加法？
- 映象函数计算的方法和速度，决定了效率

特殊矩阵

定义

- 指非零元素或零元素的分布有一定规律的矩阵。

特殊矩阵的压缩存储

- 核心思想：针对阶数很高的特殊矩阵，为节省存储空间，对可以不存储的元素，如零元素或对称元素，不再存储。

典型的特殊矩阵

- 对称矩阵
- 对角矩阵、三对角矩阵等

对称矩阵

定义与特点

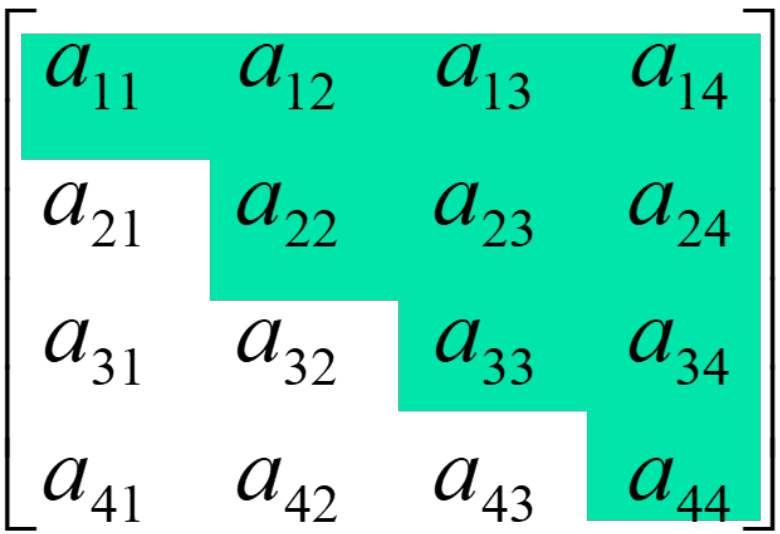
- $n \times n$ 的方阵，满足 $a_{ij} = a_{ji}$
- 例子如下：

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

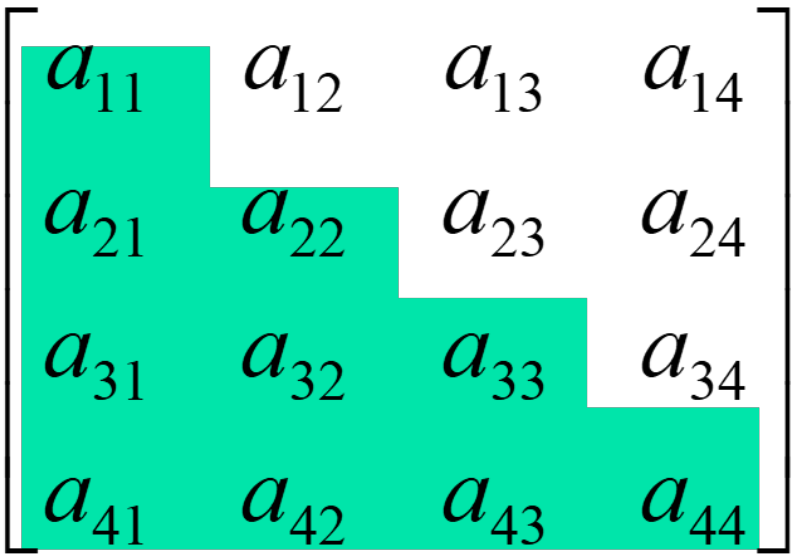
对称矩阵

压缩存储

- 只存对角线及对角线以上的元素，或者只存对角线及对角线以下的元素。前者称为上三角矩阵，后者称为下三角矩阵。
- 如图所示



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

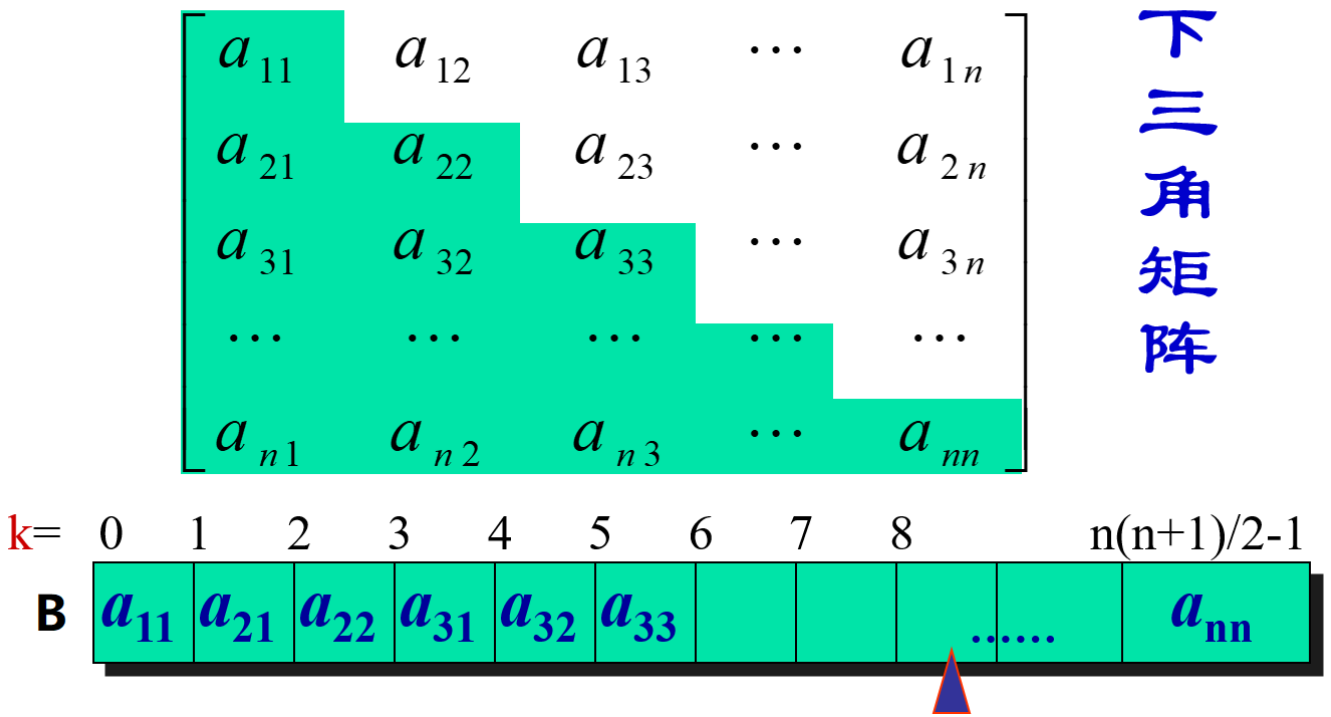


$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

对称矩阵

压缩存储的实现

- 将元素按行存放于一维数组 B 中，称为对称矩阵 A 的压缩存储方式。
- 数组 B 共有 $n+(n-1)+\ldots+1=n*(n+1)/2$ 个元素，如图所示。



对称矩阵

压缩存储的实现：映象函数的计算

- 若 $i \geq j$，数组元素 a_{ij} 在数组 B 中的存储位置
  $k = 1 + 2 + \ldots + (i-1) + (j-1) = (i-1) * i/2 + j - 1$
- 若 $i < j$，数组元素 a_{ij} 在矩阵的上三角部分，在数组 B 中没有存放，可以找它的对称元素 $a_{ji} = j * (j-1)/2 + i - 1$

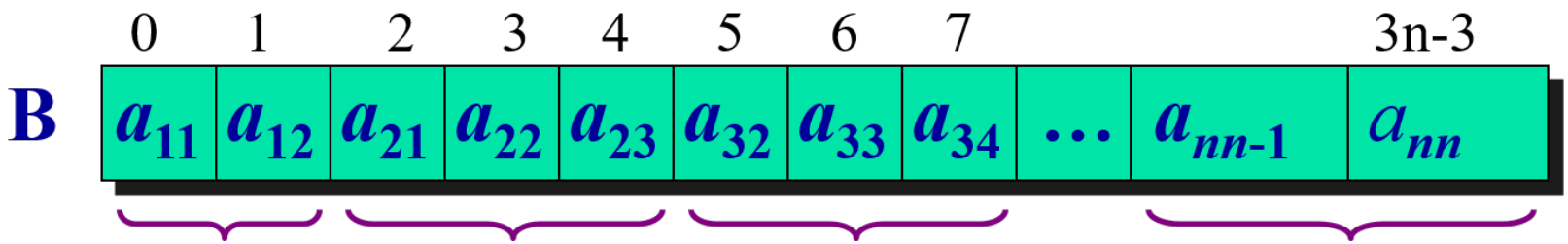
综上，得到下标 k 与 i, j 之间的关系

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

例题：三对角阵及其压缩存储

下标 k 与 i, j 之间的关系?

$$
\mathrm{A} = \begin{bmatrix}
a_{11} & a_{12} & 0 & 0 & 0 & 0 \\
a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\
0 & a_{32} & a_{33} & a_{34} & 0 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} & a_{n-1n} \\
0 & 0 & 0 & 0 & a_{nn-1} & a_{nn}
\end{bmatrix}
$$



练习题

二对角阵的压缩存储

- 如图所示二对角阵，压缩存储在一维矩阵 B 中
- 求下标 k 与 i, j 之间的关系

$$
\mathrm{A}=\begin{bmatrix}
a_{11} & a_{12} & 0 & 0 & 0 & 0 \\
0 & a_{22} & a_{23} & 0 & 0 & 0 \\
0 & 0 & a_{33} & a_{34} & 0 & 0 \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & 0 & a_{n-1n-1} & a_{n-1n} \\
0 & 0 & 0 & 0 & 0 & a_{nn}
\end{bmatrix}
$$

求解思路

- 方法一：手工“硬找”规律，先猜测表达式，再验证；
- 方法二：若存在这种表达式形式的规律，通常为“线性形式”，故，不妨设为 $k = f(i, j) = a * i + b * j + c$，然后代入几个特殊元素，解出系数 a, b, c

稀疏矩阵

定义

- 非零元素个数远远少于矩阵元素个数：非零元素比例/稀疏因子 $\delta \leq 0.05$。
- 稀疏矩阵常见且有用。一个稀疏矩阵的例子：

$$\mathbf{A}_{6\times 7} = \begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵的 ADT

```
ADT SparseMatrix{
  数据对象: D = {a_ij | i = 1,2,...,m; j = 1,2,...,n;
                a_ij ∈ Elemset, m, n分别称为行数和列数}
    数据关系:R={Row,Col}
             ……
    基本操作:
             ……
             MultSMatrix(M,N,&Q);     //矩阵乘积
             TransposeSMatrix(M,&T); //求转置矩阵
} ADT SparseMatrix
```

稀疏矩阵的压缩存储

三元组表：如图所示

- 从左侧稀疏矩阵转换为右侧仅保留非零元素及其行列位置的形式（转换了逻辑结构形式）
- 三元组表的按行优先存储：行号小先存；行号相同，列号小先存

$$M=\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

	行 (row)	列 (col)	值 (value)
[0]	1	4	22
[1]	1	7	15
[2]	2	2	11
[3]	2	6	17
[4]	3	4	−6
[5]	4	6	39
[6]	5	1	91
[7]	6	3	28

稀疏矩阵的存储实现

稀疏矩阵存储实现步骤

- 稀疏矩阵（逻辑结构）⇒ 三元组表 (逻辑结构)
- 三元组表 (逻辑结构)⇒ 三元组表的顺序实现（存储结构）
- 结构体数组来实现三元组表的存储结构

三元组表的内存表示：静态部分

```
1  #define MAXSIZE 12500
2  Typedef struct{
3      int row,col;          //非零元的行下标和列下标
4      ElemType value;
5  }Triple;
6
7  Typedef struct{
8      Triple data[MAXSIZE +1]; //非零元三元组表，以行序为主序
9      int mu,nu,tu;      //矩阵的行数、列数、非零元个数
10 }TSMatrix;
```

稀疏矩阵的存储实现

三元组表的基本操作

- 矩阵的运算包括矩阵的转置、矩阵求逆、矩阵加减、矩阵乘除等
- 讨论在这种压缩存储结构下的求矩阵的转置运算
  - 一个 $m \times n$ 的矩阵 A，它的转置 B 是一个 $n \times m$ 的矩阵，且 $b[i][j] = a[j][i], 0 \leq i \leq n-1, 0 \leq j \leq m-1$
  - 设稀疏矩阵 A 是按行优先顺序压缩存储在三元组表 a.data 中，若仅仅是简单地交换 a.data 中 i 和 j 的内容，得到三元组表 b.data，b.data 将是一个按列优先顺序存储的稀疏矩阵 B，要得到按行优先顺序存储的 b.data，就必须重新排列三元组表 b.data 中元素的顺序

矩阵转置算法一：思想

- 将矩阵的行、列下标值交换。即将三元组表中的行、列位置值 i, j 相互交换
- 重排三元组表中元素的顺序。即交换后仍然是按行优先顺序排序的
- 算法细节略

矩阵转置的实现二

伪代码思想:

- 从稀疏矩阵 A 的三元组表 a.data 中寻找未处理的、列号最小、同列号时行号最小的三元组，修改并存入 b.data 中
- 找每个未处理的编号最小“三元组”，如何找？(利用三元组表的行列下标分布特征)
- 时间复杂度 $O(a.nu \times tu)$，一般矩阵的转置时间复杂度 $O(mu \times nu)$。当非零元素的个数 tn 和 $mu \times nu$ 同量级时，算法时间复杂度为 $O(mu \times nu^2)$，时间增加，时间换空间，适合稀疏矩阵

```
1  void TransMatrix(TSMatrix a,TSMatrix &b)
2  {  int p, q, col; b.mu=a.nu; b.nu=a.mu; b.tu=a.tu;
3  /*置三元组表b.data的行、列数和非0元素个数*/
4  if (b.tu==0) printf("The Matrix A=0\n");
5  else{ q=0;
6       for(col=1;col<=a.nu;col++)//从1开始，依次指定列号
7          for (p=0;p<a.tu; p++) //遍历三元组表找指定列号
8            if (a.data[p].col==col){ //找到了就将其复制到转置后结果b
9               b.data[q].row=a.data[p].col; //一定是先找到行号最小的
10              b.data[q].col=a.data[p].row;
11              b.data[q].value=a.data[p].value;
12              q++;
13           }
14     }
15 }
```

矩阵转置的实现三

改进的快速算法：算法思想

- 按照稀疏矩阵 A 三元组表 a.data 的次序，直接依次转换，将 a.data 的元素转换后的三元组直接放置在转置后三元组表 b.data 的恰当位置
- 想象成算法一的改进，交换下标 i, j 后，直接将三元组放入“恰当的位置”。元素在转置后，其“恰当的位置”的计算成为关键

“恰当的位置”计算：独立、快速的过程，如右图所示

- 通过求得 A 中每一列的非 0 元素个数，就能算出原矩阵 A 中每一列的 (即 B 中每一行) 第一个非 0 元素在 b.data 中的“恰当的位置”，则在转置时就可直接放在 b.data 中该位置；
- 其他元素的恰当位置：如何计算？

$$A=\begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

非零个数： 1 1 1 2 0 2 1

k: 0 1 2 3 ? 5 7

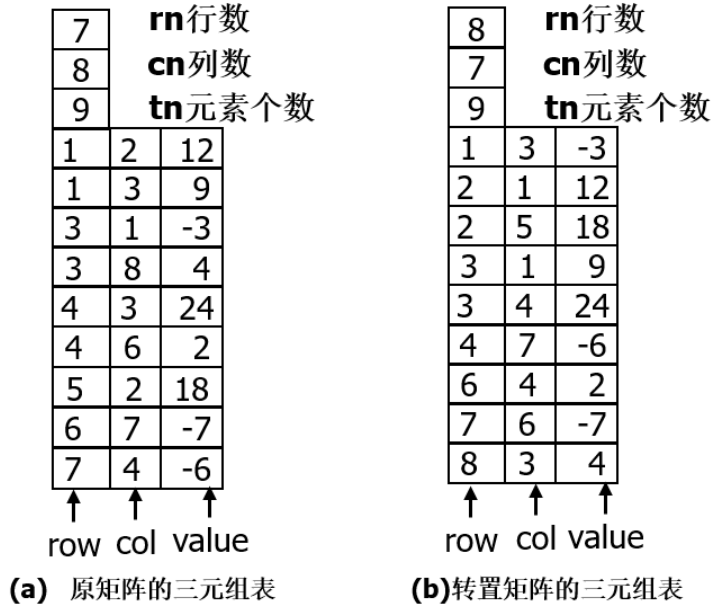
$B=A^T$ B[k]=A[i][j]

红色k指出A每列（B每行的第一个非零元素的下标）

矩阵转置的快速算法：示例

$$A=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵示例



稀疏矩阵及其转置矩阵的三元组顺序表

附设两个辅助向量 num[ ] 和 cpot[ ]

- num[col]：统计 A 中第 col 列中非 0 元素的个数；
- cpot[col]：指示 A 中第 col 列中第一个非 0 元素在 b.data 中的恰当位置。

矩阵转置的快速算法：示例

$$\begin{cases} \text{cpot}[0] = 0 \\ \text{cpot}[col] = \text{cpot}[col-1] + \text{num}[col-1] & 1 \leq col < a.cn \end{cases}$$

$$A = \begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{pmatrix}$$

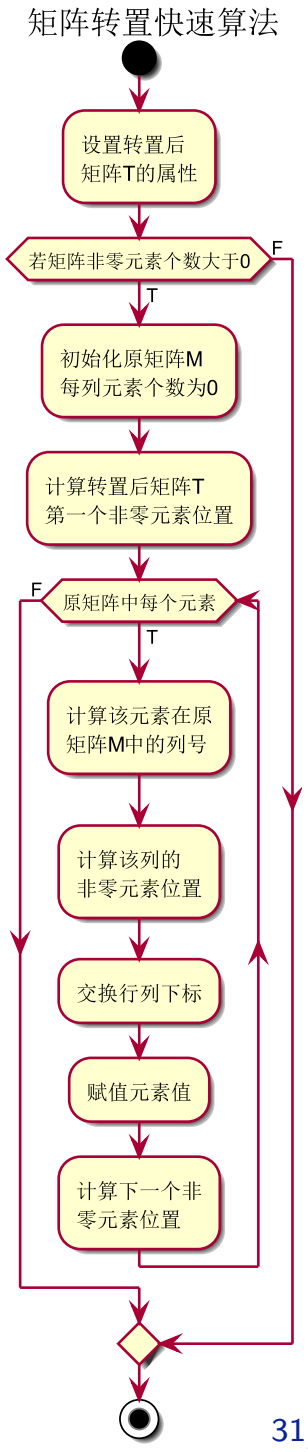
col	0	1	2	3	4	5	6	7
num[col]	1	2	2	1	0	1	1	1
cpot[col]	0	1	3	5	6	6	7	8

每列的第 2，3，...，个非零元素怎么办？

- 第 i 列的第一个非零元素总是最先放入转置后的矩阵 B 中，第二非零元素第 2 个放入 B 中，所以第 2 个元素的位置就是 ++cpot[i]
- 故第 i 列的某个元素放入 B 中，++cpot[i] 即可得到下一个非零元素的“恰当位置”

矩阵转置快速算法伪代码

```
1  Status FastTransMatrix(TSMatrix M, TSMatrix &T){
2    T.mu=M.nu; T.nu=M.mu; T.tu=M.tu;
3    if (T.tu) {
4      for(col=0;col<M.nu;++col) num[col]=0;
5      for(t=0;t<M.tu;++t)//求M中每一列非零元个数
6        ++num[M.data[t].col];
7      cpot[0]=0;
8      //求第col列中第一个非零元在T.data中的序号
9      for(col=1;col<=M.nu;++col)
10       cpot[col]=cpot[col-1]+num[col-1];
11     for(p=0;p<=M.tu;++p){
12       col=M.data[p].j; q=cpot[col];
13       T.data[q].row=M.data[p].col;
14       T.data[q].col=M.data[p].row;
15       T.data[q].value=M.data[p].value;
16       ++cpot[col];
17     }//for
18   }//if
19   return OK;
20 } // 时间复杂度 O(nu + tu)
```



稀疏矩阵的链式存储

十字链表

- 对于稀疏矩阵，当非 0 元素的个数和位置在操作过程中变化较大时，采用链式存储结构表示比三元组的线性表更方便。
- 矩阵中非 0 元素的结点所含的域有：行、列、值、行指针 (指向同一行的下一个非 0 元)、列指针 (指向同一列的下一个非 0 元)。其次，十字链表还有一个头结点，结点的结构如下图所示。

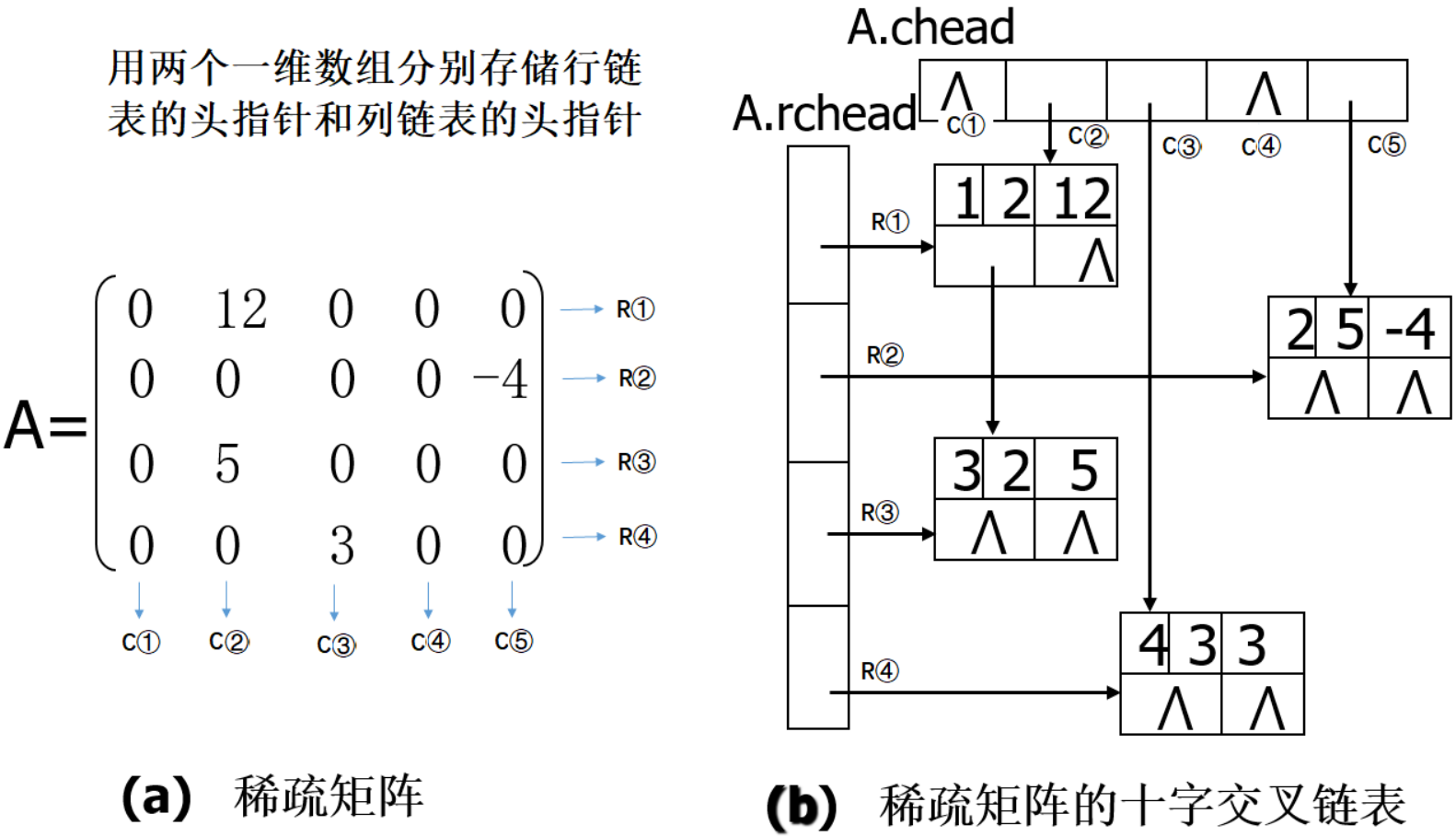
(a) 结点结构　　**(b)** 头结点结构

图　十字链表结点结构

- 稀疏矩阵中同一行的非 0 元素由 right 指针域链接成一个行链表，由 down 指针域链接成一个列链表；每个非 0 元素既是某个行链表中的一个结点，同时又是某个列链表中的一个结点，所有的非 0 元素构成一个十字交叉的链表。称为十字链表。

十字链表

例子如图所示


用两个一维数组分别存储行链表的头指针和列链表的头指针
A=
0 12 0 0 0
0 0 0 0 −4
0 5 0 0 0
0 0 3 0 0
R①
R②
R③
R④
C①
C②
C③
C④
C⑤
A.chead
A.rchead
1 2 12
2 5 -4
3 2 5
4 3 3
(a) 稀疏矩阵
(b) 稀疏矩阵的十字交叉链表

十字链表的实现

十字链表在内存中的表示：静态部分

```
1  typedef struct Clnode
2  {   int row , col ; /* 行号和列号 */
3      elemtype value ; /* 元素值 */
4      struct Clnode *down , *right ;
5  }OLNode ; /* 非0元素结点 */
6
7  typedef struct Clnode
8  {   int  rn;        /* 矩阵的行数 */
9      int  cn;        /* 矩阵的列数 */
10     int  tn;        /* 非0元素总数 */
11     OLNode *rhead ;
12     OLNode *chead ;
13 } CrossList ;
```

练习题

稀疏矩阵

- 设有稀疏矩阵 A 如下图所示，请画出该稀疏矩阵的三元组表和十字链表存储结构。

$$A=\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 5 & 0 \\ 0 & 0 & -3 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

广义表

定义

- 广义表是线性表的推广，在人工智能领域中应用十分广泛。(lisp 语言)
- 第二章中线性表被定义为 $n(\geq 0)$ 个元素 $a_1, a_2, \ldots, a_n$ 的有穷序列，该序列中的所有元素具有相同的数据类型且只能是原子项 (Atom)。
  - 所谓原子项可以是一个数或一个结构，是指结构上不可再分的。若放松对元素的这种限制，容许它们具有其自身结构，就产生了广义表。
  - 广义表 (Lists，简称列表)：是由 $n(\geq 0)$ 个元素组成的有穷序列：$LS = (a_1, a_2, \ldots, a_n)$，其中 a_i 或者是原子项，或者是一个广义表。LS 是广义表的名字，n 为它的长度。若 a_i 是广义表，则称为 LS 的子表。

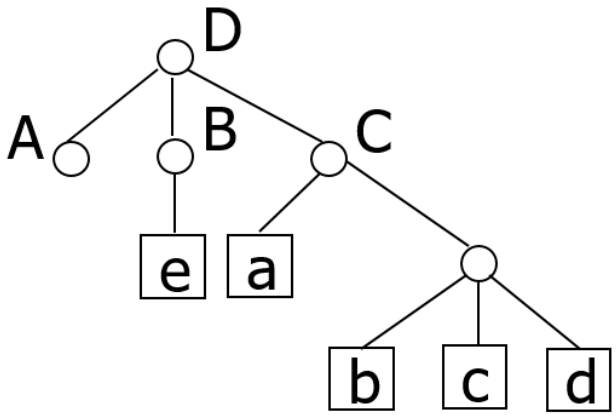
更多说明

- 习惯上：原子项用小写字母，子表用大写字母。
- 若广义表 LS 非空时：
  - a_1(表中第一个元素) 称为表头；
  - 其余元素组成的子表称为表尾；$(a_2, \ldots, a_n)$
  - 广义表中所包含的元素 (包括原子和子表) 的个数称为表的长度。
  - 广义表中括号的最大层数称为表深 (度)。

广义表的例子

广义表及其示例

广　义　表	表长n	表深h
A=()	0	1
B=(e)	1	1
C=(a,(b,c,d))	2	2
D=(A,B,C)	3	3
E=(a,E)	2	∞
F=(())	1	2



广义表的图形表示

(+ 3 (- 4 (* 5 9)))

Lisp程序设计语言例子

理解广义表

更多解释

- 广义表的元素可以是原子，也可以是子表，子表的元素又可以是子表，…。即广义表是一个多层次的结构。
- 广义表可以被其它广义表所共享，也可以共享其它广义表。广义表共享其它广义表时通过表名引用。
- 广义表本身可以是一个递归表。
- 根据对表头、表尾的定义，任何一个非空广义表的表头可以是原子，也可以是子表，而表尾必定是广义表。

广义表的实现

链式存储结构

- 由于广义表中的数据元素具有不同的结构，通常用链式存储结构表示，每个数据元素用一个结点表示。因此，广义表中就有两类结点：
  - 一类是表结点，用来表示广义表项，由标志域，表头指针域，表尾指针域组成；
  - 另一类是原子结点，用来表示原子项，由标志域，原子的值域组成。
  - 只要广义表非空，都是由表头和表尾组成。即一个确定的表头和表尾就唯一确定一个广义表。

(a) 原子结点　　**(b)** 表结点

图 广义表的链表结点结构示意图

广义表的实现

广义表在内存中的表示：静态结构

```
1       typedef struct GLNode
2       { int  tag ;  /*标志域，为1：表结点;为0 ：原子结点*/
3         union{
4               elemtype value; /*原子结点的值域*/
5               struct {
6                 struct GLNode *hp;
7                 struct GLNode *tp;
8               }
9             };
10      } GLNode ;    /* 广义表结点类型 */
```

广义表的实现

例子

- 对 A=()，B=(e)，C=(a, (b, c, d))，D=(A, B, C)，E=(a, E) 的广义表的存储结构如下图所示

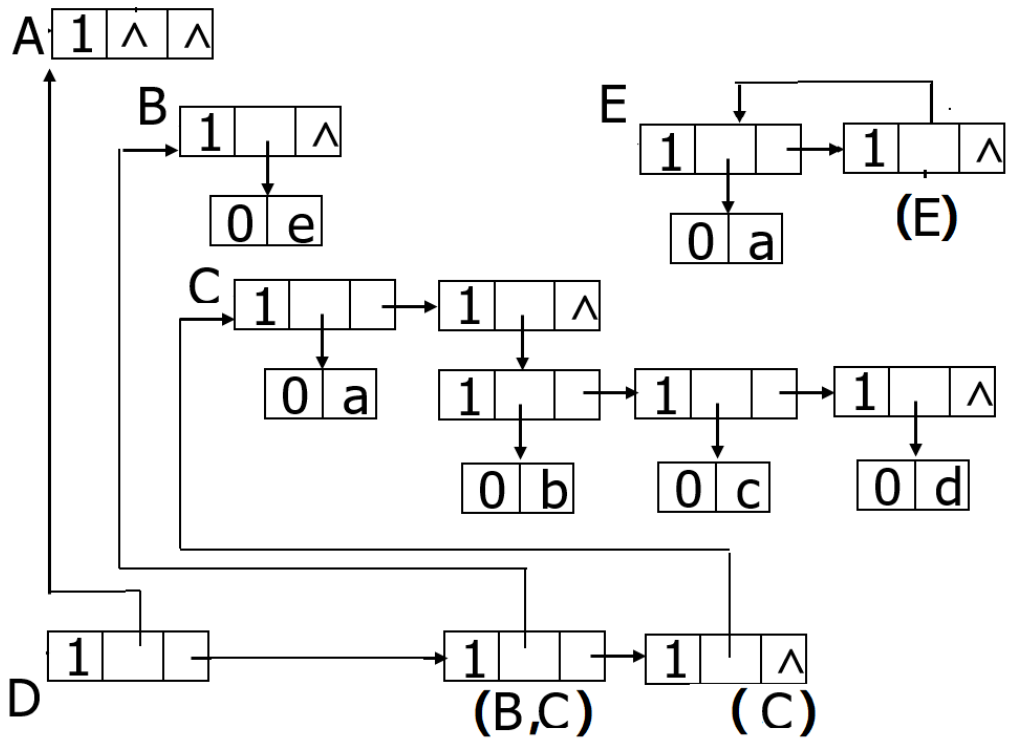


图 广义表的存储结构示意图

广义表的实现

广义表的存储结构的特点

- 若广义表为空，表头指针为空；否则，表头指针总是指向一个表结点，其中 hp 指向广义表的表头结点 (或为原子结点，或为表结点)，tp 指向广义表的表尾 (表尾为空时，指针为空，否则必为表结点)。
- 这种结构求广义表的长度、深度、表头、表尾的操作十分方便。
- 表结点太多，造成空间浪费。

练习题

广义表

- 什么是广义表？请简述广义表与线性表的区别？
- 一个广义表是 (a, (a, b), d, e, (a, (i, j), k))，请画出该广义表的链式存储结构。

二维数组

- 设有二维数组 a[6][8]，每个元素占相邻的 4 个字节，存储器按字节编址，已知 a 的起始地址是 1000，试计算：
  - 数组 a 的最后一个元素 a[5][7] 起始地址；
  - 按行序优先时，元素 a[4][6] 起始地址；
  - 按行序优先时，元素 a[4][6] 起始地址。