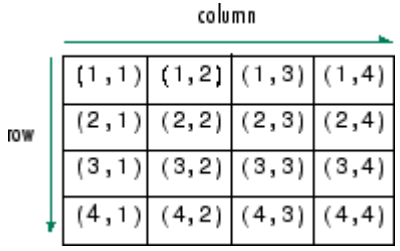
Ch 5　数组和广义表

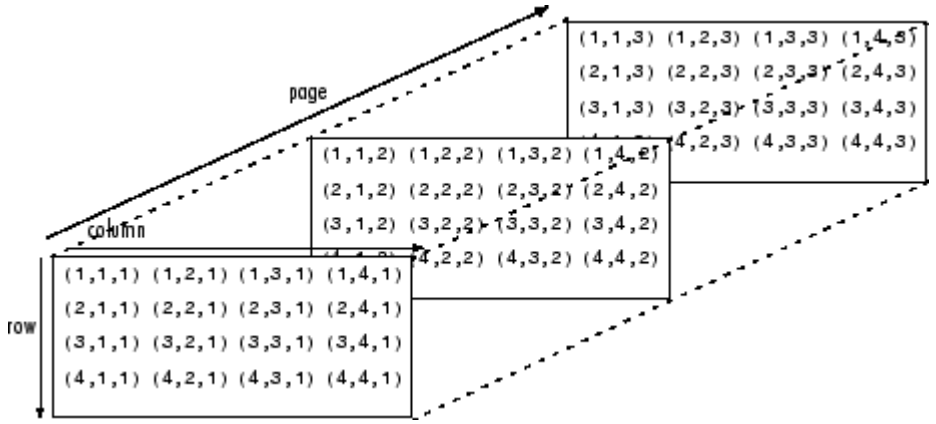
优势：
1. 数组是一种数据容器，便于数据管理
2. 数组元素的类型可按需定义，便于处理不同问题
3. 支持直接访问方式，便于数据操作 -> O(1)
（静态）数组限制了一些基本运算

- 数组(An array)的基本知识

数组是一组存储在地址连续的存储空间中的类型相同的数据元素的集合.

二维数组 & 多维数组





数组分量是指数组中的一个数据元素. 它通常用下标(subscript)来标识(index). 下标可以从0或1或任意值n开始.下标的个数取决于数组维度, 且每个下标都具有确定的下界和上界, 每一维的长度/大小为该维的[上界–下界+1].

使用数组的优势(前已述).

任意一个数组元素的地址 = 基地址 + 偏移量

为什么默认从0开始：使得数组实现更为简单高效（若从1开始，额外增加了CPU的计算负担）

基本操作: (1)初始化即定义一个数组 (2)访问或存取数组元素的值
但一般不进行插入、删除运算(why? Once we create an array, its size is fixed.)

- 数组(array)的抽象数据类型ADT

n-Array = (D/数据对象, R/数据关系, O/基本操作) ➔ 描述: P90, 其中:

数据对象 n维数组$A_{b1b2...bn}$中含有 $\prod_{i=1}^{n} b_i$ (b_i是数组第i维的长度)个数据元素,
其中每个数组分量记为$a_{j1j2...jn}$, j_i表示n维中的第i维(i=1,2,...,n)
且j_i=1,2,…,b_i (或记作1..b_i) 或 j_i=0,1,…,b_i–1(或记作0..b_i–1)
如: A_6 --- 一维数组A(即n=1), 长度为6(即b_1=6), a_1~a_6或a_0~a_5
$A_{4\times5}$ --- 二维数组A(即n=2), 长度分别为4和5(即b_1=4, b_2=5),
a_{11}~a_{45}或a_{00}~a_{34}

数据关系 n维数组每个数据元素均受n个关系的约束. 如2维数组的行、列关系: <$a_{i,j}$, $a_{i,j+1}$>、 <$a_{i,j}$, $a_{i+1,j}$>, 3维数组中的每个元素考虑行、列及页关系则最多分别有3个可能的前驱和3个可能的后继, 依次类推m维数组中的每个元素则最多分别有m个可能的前驱和m个可能的后继. 因此, 二维或多维数组(n>2)属于非线性结构.

基本操作: …

- 数组的顺序存储

考虑到数组一旦建立, 存储结构中的元素个数和元素之间的关系就不再发生变化. 因此, 一般都只采用顺序存储的方法来表示数组.

如何用线性的(即一维的)存储结构来表示非线性的二维或多维数组?
先按某种次序将数组元素排成一个线性序列, 然后再将其依次存放至RAM.

- 行优先顺序(Row-Major Ordering)
- 列优先顺序(Column-Major Ordering)

} 一个例子

注: C/C++, JAVA, VB中的数组均默认是按行优先顺序存储的.

p=m行优先
p=n 列优先

n=1时, 1维数组便退化为一个定长的线性表/顺序表
n=2时, 2维数组可看作一个扩展的定长的线性表: $A_{m\times n} = (\alpha_1, \alpha_2, \cdots, \alpha_p)$

它的每个数据元素又均是一个定长的线性表
即$\alpha_i = (a_{i1}, a_{i2}, \ldots, a_{in},)$ 或 $\alpha_i = (a_{1i}, a_{2i}, \ldots, a_{mi},)$

因此, 一个n维数组可定义成其数据元素为n–1维数组类型的一维数组类型.

```
typedef ElemType A[m][n];  //m×n的二维数组
或typedef ElemType A1[n]; //含n个元素的一维数组
  typedef A1 A[m];  //含m个元素的一维数组
                    //A中每个元素是A1类型
```

$$A_{m\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{bmatrix}$$

- 数组的顺序存储 — 一个例子

以二维数组为例

$$A_{m\times n}=\begin{bmatrix}(a_{11} & a_{12} & \cdots & \cdots & a_{1n})\\ (a_{21} & a_{22} & \cdots & \cdots & a_{2n})\\ (\cdots & \cdots & \cdots & \cdots & \cdots)\\ (a_{m1} & a_{m2} & \cdots & \cdots & a_{mn})\end{bmatrix}$$

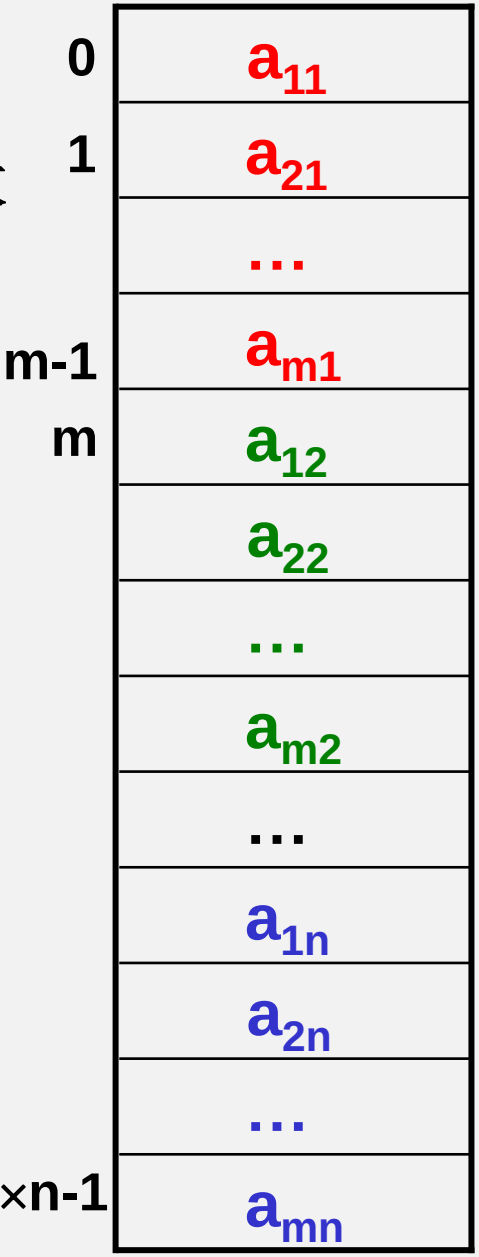
上述的行、列优先顺序可推广到维数更高的数组: 行优先规定为先按最右的下标排, 从右向左, 最后按最左下标排. 列优先则先按最左的下标排, 从左向右, 最后按最右下标排.

如: 三维数组$A_{m\times n\times p}$按行优先, 则:

$a_{1,1,1}\ a_{1,1,2}\ \cdots\ a_{1,1,p}\ a_{1,2,1}\ a_{1,2,2}\ \cdots\ a_{1,2,p}\ \cdots\ a_{1,n,1}\ \cdots\ a_{1,n,p}\ a_{2,1,1}\ \cdots\ a_{2,1,p}\ a_{2,2,1}\ \cdots a_{2,2,p}\ \cdots a_{m,1,p}\ \cdots a_{m,n,p}$

$Loc(a_{i,j,k})=Loc(a_{1,1,1})+[(i-1)\times n\times p+(j-1)\times p+k-1]\times L$

以列为主序存放



$Loc(a_{i,j})=Loc(a_{1,1})+[(j-1)\times m+(i-1)]\times L$

- 数组的顺序存储

可将 $Loc(a_{i,j,k}) = Loc(a_{1,1,1}) + [(i-1)\times n\times p + (j-1)\times p + k-1]\times L$ 推广至一般情况:

$$Loc(a_{j1,j2,\ldots,jn}) = Loc(a_{1,1,\ldots,1}) + [(j_1-1)\times b_2\times b_3\times \ldots \times b_n + (j_2-1)\times b_3\times b_4\times \ldots \times b_n + \ldots + (j_{n-1}-1)\times b_n + j_n-1] \times L$$

上述的讨论, 数组$A_{b1b2\ldots bn}$中数组元素各维下标均从1开始计算, 即$j_i=1,2,\ldots,b_i$. 若元素各维下标均从0开始计算, 即$j_i=0,1,\ldots,b_i-1$, 则上述所有地址公式变为:

二维数组$A_{m\times n}$: $Loc(a_{i,j})=Loc(a_{0,0}) + (i\times n + j) \times L$ (按行优先)

$Loc(a_{i,j})=Loc(a_{0,0}) + (j\times m + i) \times L$ (按列优先)

三维数组$A_{m\times n\times p}$: $Loc(a_{i,j,k})=Loc(a_{0,0,0}) + (i\times n\times p + j\times p + k) \times L$ (按行优先)

n维数组$A_{b1b2\ldots bn}$: $Loc(a_{j1,j2,\ldots,jn}) = Loc(a_{0,0,\ldots,0}) + (j_1\times b_2\times b_3\times \ldots \times b_n + j_2\times b_3\times b_4\times \ldots \times b_n + \ldots + j_{n-1}\times b_n + j_n) \times L$

几个例子 ➔

- 数组的运算

C/C++允许大于2维的数组, 维数的限制依赖于具体的编译程序. 考虑高维数组由于占用存储空间太大, 一般多为专业计算使用.

因此, 有关数组的运算只需掌握一维、二维、三维数组运算即可. P93~94略去

例1.设2维数组$A_{5\times6}$的每个元素占4个字节, 已知 $Loc(a_{00}) = 1000$, A共占多少字节? A的元素a_{45}的起始地址为多少? 按行和按列优先存储时, a_{24}的起始地址分别为多少?

$A_{5\times6}$中共有5×6=30个数组元素, 共占30×4=120字节,又元素下标地址从0开始计算. 则知a_{45}是A中最末一个元素, 其 $Loc(a_{56}) = Loc(a_{0,0}) + 120 - 4 = 1116$.
若按行优先, 则:$Loc(a_{24}) = Loc(a_{0,0}) + (i\times n + j)\times L = 1000+(2\times6 + 4)\times4=1064$.
若按列优先, 则:$Loc(a_{24}) = Loc(a_{0,0}) + (j\times m + i)\times L = 1000+(4\times5 + 2)\times4=1088$.

例2. 三维数组$A_{4\times5\times6}$按行优先存储在内存中, 若每个元素占2个字节存储单元, 且数组中第一个元素的存储地址为120, 则元素$a_{3,4,5}$的存储地址为( ).
A. 356　B. 358　C. 360　D. 362

按题目并不知道数组元素下标是从0还是1开始计算下标即第一个元素是$a_{0,0,0}$还是$a_{1,1,1}$, 且$a_{3,4,5}$的各维下标值无论是从0还是1开始计算都合法, 因此只能分别按下标从0和1开始来计算$a_{3,4,5}$的存储地址, 然后与答案比较而选择结果. [否则, 自己约定或按考试指定的参考用书中的说明. 一般地, 若有数组A[r..s, t..u], 则表示一个二维数组, $a_{r,t}$为其第一个元素且两维大小分别为s-r+1,u-t+1.]
对$A_{b1,b2,b3}$, $Loc(a_{i,j,k})=Loc(a_{0,0,0}) + (i\times b_2\times b_3 + j\times b_3 + k) \times L$　(按行优先)
$Loc(a_{i,j,k})=Loc(a_{1,1,1}) + [(i-1)\times b_2\times b_3 + (j-1)\times b_3 + (k-1)] \times L$
对$A_{4\times5\times6}$, 则: $Loc(a_{3,4,5}) = 120 + (3\times5\times6 + 4\times6 + 5) \times 2 = 358$
$Loc(a_{3,4,5}) = 120 + (2\times5\times6 + 3\times6 + 4) \times 2 = 284$
因此, 选择B.

例3. 按行及按列优先顺序列出4维数组$A_{2\times3\times2\times3}$的所有元素在RAM中的存储次序, 设第一个数组元素是a_{0000}.

该4维数组共有2×3×2×3=36个元素. 按多维数组的行、列优先次序分别得:

a_{0000} a_{0001} a_{0002}
a_{0010} a_{0011} a_{0012}
a_{0100} a_{0101} a_{0102} a_{0110} a_{0111} a_{0112}
a_{0200} a_{0201} a_{0202} a_{0210} a_{0211} a_{0212}
a_{1000} a_{1001} a_{1002}
a_{1010} a_{1011} a_{1012}
a_{1100} a_{1101} a_{1102} a_{1110} a_{1111} a_{1112}
a_{1200} a_{1201} a_{1202} a_{1210} a_{1211} a_{1212}

行优先次序

---

a_{0000} a_{1000}
a_{0100} a_{1100}
a_{0200} a_{1200}
a_{0010} a_{1010}
a_{0110} a_{1110}
a_{0210} a_{1210}

a_{0001} a_{1001}
a_{0101} a_{1101}
a_{0201} a_{1201}
a_{0011} a_{1011}
a_{0111} a_{1111}
a_{0211} a_{1211}

a_{0002} a_{1002}
a_{0102} a_{1102}
a_{0202} a_{1202}
a_{0012} a_{1012}
a_{0112} a_{1112}
a_{0212} a_{1212}

列优先次序

- 特殊矩阵(二维数组)的存储结构

1. 特殊矩阵的定义

值相同(0或非零)的数据元素在矩阵中分布有规律的方阵(Square Matrix)(n阶矩阵).

1)n阶($M_{n\times n}$)下(上)三角矩阵(Lower/Upper Triangular Matrix)

$$M=\begin{pmatrix} x & & & & \\ x & x & & 0 & \\ x & x & x & & \\ & & \cdots & & \\ x & x & \cdots & x & x \end{pmatrix}_n \qquad M=\begin{pmatrix} x & x & \cdots & x & x \\ & \cdots & & & \\ & & x & x & x \\ 0 & & & x & x \\ & & & & x \end{pmatrix}_n$$

主对角线以上(以下)的数据元素均为0.

2)n阶三对角矩阵
(Tri-diagonal Matrix)

$$M=\begin{pmatrix} x & x & & & \\ x & x & x & & \\ & x & x & x & 0 \\ & & \cdots & & \\ 0 & & x & x & x \\ & & & x & x \end{pmatrix}_n$$

主对角线及其直接上下方元素为非0. 即除了首尾行外其余行的元素均为3个

3)n阶对称矩阵
(Symmetrical Matrix)

$$M=\begin{pmatrix} a_{ij}=a_{ji} \\ i,j\in[0,n-1] \end{pmatrix}_n$$

2. 特殊矩阵的压缩存储

压缩存储就是只存储上述特殊矩阵中的所有非0元素.

将非零元素分布有规律的特殊矩阵M"压缩"到一维数组A中, 并确定M和A两者之间的一一对应关系, 即若已知矩阵中的非0元素a_{ij}, 则能找到它在一维数组中的位置k; 反之亦然.

约定: 矩阵M和数组A的下标均从0开始并且按行优先存储.

➢对n阶下(上)三角矩阵

$$k = \begin{cases} \dfrac{i\times(i+1)}{2} + j & \text{If } i \geq j \text{ (下三角矩阵)} \\ i + \dfrac{j\times(j+1)}{2} & \text{If } i \leq j \text{ (上三角矩阵)} \end{cases}$$

其中：$i, j\in[0, n-1]$

$k\in[0, n(n+1)/2-1]$

$$Loc(a_{ij}) = Loc(a_{00}) + [\frac{i\times(i+1)}{2} + j] \times L$$

…

➢对n阶三对角矩阵

$k = 2 \times i + j$ (?如何推得的)

其中：$i, j\in[0, n-1]$; $k\in[0, 3n-3]$

$$Loc(a_{ij}) = Loc(a_{00}) +2 \times i + j$$

➢对n阶对称矩阵 ➔ 同n阶下(上三角)矩阵

$$k = \begin{cases} \dfrac{(i-1)\times i}{2} + j - 1 + 1 \\ \dfrac{(j-1)\times j}{2} + i - 1 + 1 \end{cases}$$

$k = 2 \times (i - 1) + (j-1) + 1$

$= 2 \times i + j - 2$

其中：$i, j\in[1, n]$

$k\in[1, n(n+1)/2]$

$k\in[1, 3n-2]$

- 三元组表—稀疏矩阵(Sparse Matrix)的一种压缩存储方法

1. 三元组(表)的表示

只有少量非零元素且在其中分布无规律(稀疏因子δ)

(i, j, a_{ij}) 其中$a_{ij} \neq 0$. 若干个这样的三元组构成三元组表.

三元组表中每个元组和稀疏矩阵中每个非0元素一一对应.

```
typedef struct tuple { int i, j;   //非0元素的行下标、列下标
                       ElemType e; } Triple;
typedef struct smatrix { int mu, nu, tu;  //矩阵的行数、列数、非0元素个数
                         Triple data[MaxSize]; } SMatrix;
```

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & 15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}_{6\times6} \qquad T = \begin{pmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 28 \\ 22 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}_{6\times6}$$

p	M.data[p].i	M.data[p].j	M.data[p].e		T.data[q].i	T.data[q].j	T.data[q].e
1	1	1	15	❶→	❶ 1	1	15
2	1	4	22	❷	❼ 1	5	91
3	1	6	15	❸	❹ 2	2	11
4	2	2	11	❹	❺ 3	2	3
5	2	3	3	❺	❽ 3	6	28
6	3	4	6	❻	❷ 4	1	22
7	5	1	91	❼	❻ 4	3	6
8	6	3	28	❽	❸ 6	1	15

2. 通过三元组表对稀疏矩阵进行运算 --- 转置[从三元组表到三元组表的运算]

方法一: 对矩阵$M_{m\times n}$的每一列(col=1 to nu), 均扫描一趟三元组表(共n趟), 每趟均需比较(M.data[p].j, col)(共p=1 to tu次). O(nu×tu). 具体算法描述P98-99/A.5.1.

```
……
q = 1;   //q为存放转置矩阵T的三元组的标识位置
for ( col = 1; col <= M.nu; ++col )  //趟数控制
  for ( p = 1; p <= M.tu; ++p )  //每趟的扫描
    if ( M.data[p].j == col ) {  //生成新的三元组表
      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; }
……
```

p	M.data[p].i	M.data[p].j	M.data[p].e		T.data[q].i	T.data[q].j	T.data[q].e
1	1	1	15	❶→❶	1	1	15
2	1	4	22	❹→❶	1	5	91
3	1	6	15	❻→❷	2	2	11
4	2	2	11	❷→❸	3	2	3
5	2	3	3	❸→❸	3	6	28
6	3	4	6	❹→❹	4	1	22
7	5	1	91	❶→❹	4	3	6
8	6	3	28	❸→❻	6	1	15

❺→❺ 其中第5趟的扫描结果为空

2. 通过三元组表对稀疏矩阵进行运算 --- 转置[从三元组表到三元组表的运算]

方法二: 先计算出$M_{m\times n}$的每一列中非0元素的个数num[col]:

```
for (col=1;col<=M.nu; ++col) num[col] = 0;
for ( p=1; p<=M.tu; ++p) ++num[M.data[p].j]; //求M中每一列非0个数
```

再求得每一列的第一个非0元素在T.data中应有的位置cpot[col].

这两者之间的关系如下:

cpot[1] = 1;

cpot[col] = cpot[col-1] + num[col-1] 其中: $2 \le col \le M.nu$

对上例，则有:

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

```
for (col=2;col<=M.nu; ++col)
    cpot[col]=cpot[col-1]+num[col-1];
```

最后, 这时只需扫描一趟三元组表: 根据M.data[p].j的列值col, 按cpot[col]的位置值q, 将M.data[p]放在T.data[q]中即可. O(nu+tu). 完整算法P100/A.5.2. ◀

$$M = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & 15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}_{6\times 6}$$

↓ Transpose

$$T = \begin{pmatrix} 15 & 0 & 0 & 0 & 91 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 28 \\ 22 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}_{6\times 6}$$

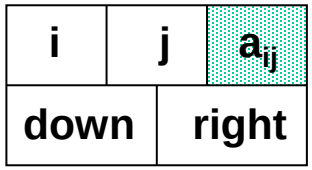
```
for (p=1;p<=M.tu; ++p) {
  col=M.data[p].j; q=cpot[col];
  T.data[q].i=M.data[p].j;T.data[q].j=T.data[p].i;
  T.data[q].e=M.data[p].e; ++cpot[col]; }
```

对稀疏矩阵当然还有其它的运算, 如修改、两个矩阵的相加和相乘等.

- 十字链表(cross/orthogonal linked list)—稀疏矩阵的另一种压缩存储方法

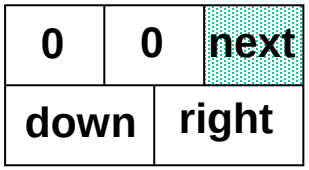
1. 十字链表的组成元素

1) m×n阶稀疏矩阵中的非零元素结点, 它含5个域:

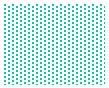


其中: i, j域为a_{ij} ($a_{ij} \neq 0$)的行、列位置值,
指针域right指向该行(i行)的下一个非零元素,
指针域down指向该列(j列)的下一个非零元素.

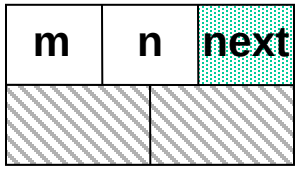
2) 有s(s = Max(m, n))个行列表头结点: [行列 ➔ 行、列]



其中: 指针域next指向下一个行列表头结点,
指针域right指向该行(i行)的第一个非零元素,
指针域down指向该列(j列)的第一个非零元素.

注:  域是一个联合(union)类型, 表现为value或pointer

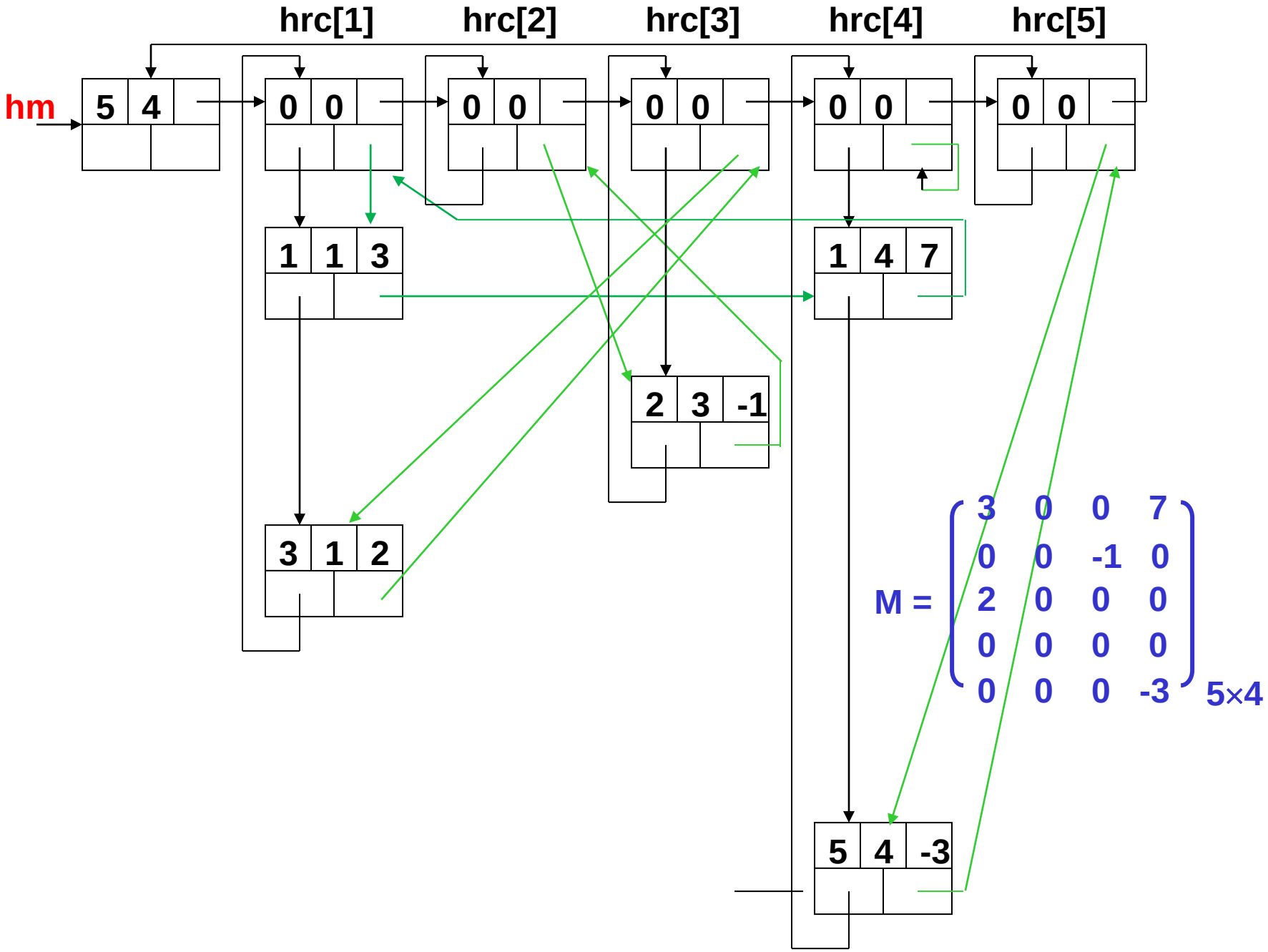
3) 有一个附加的总表头结点:



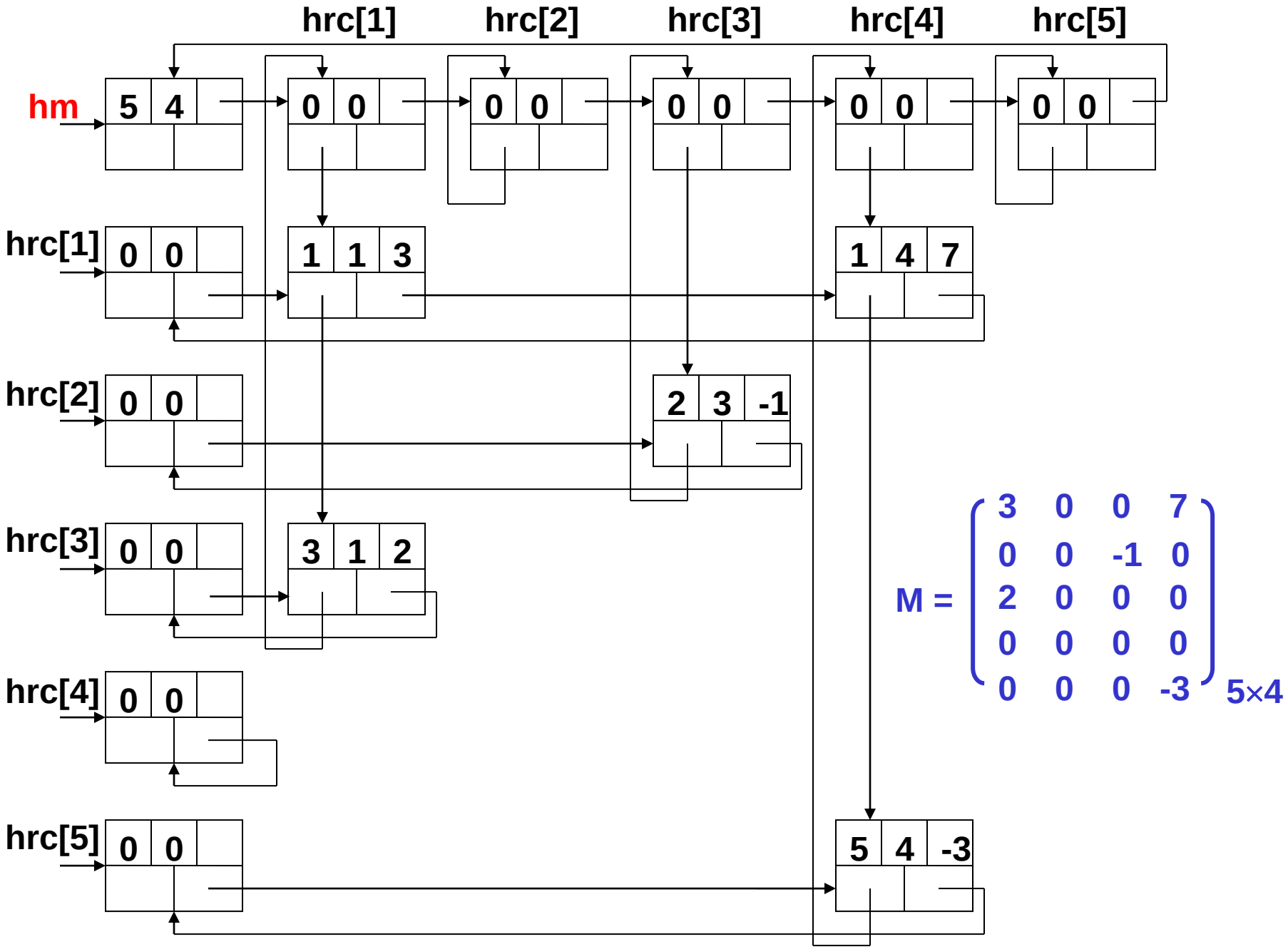
其中: m, n域为矩阵的行、列数,指针域next指向第一个行列表头结点, right和down域为空.

稀疏矩阵中的每个非零元素既是某个行链表中的一个结点, 又是某个列链表中的一个结点, 整个矩阵构成一个十字交叉的链表, 故称之为~.
其中, 总表头结点+行列表头结点组成一个循环链表, 只要给出指向总表头结点的指针值hm(头指针), 便可存取整个稀疏矩阵.

2. 十字链表的物理存储结构



2. 十字链表的物理存储结构

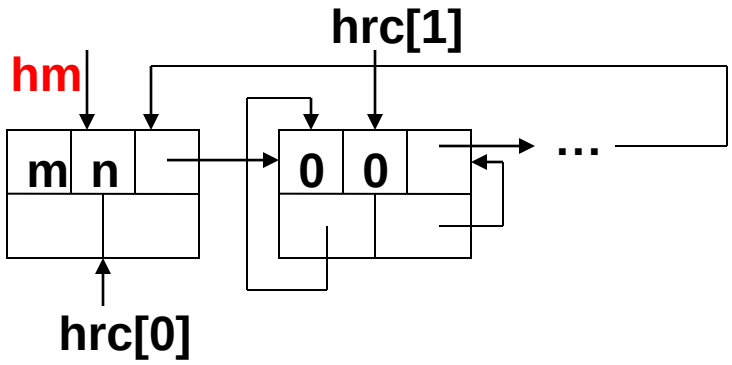


3. 稀疏矩阵的十字链表的创建算法　参见P104~105/A.5.4

```
typedef struct crosslistnode {
    int row, col;
    struct crosslistnode *down, *right;
    union { struct crosslistnode *next;
            ElemType val; }
 } CLNode;
typedef struct crosslist {
   CLNode *hm, *hrc[ ];
   int m, n, t;
 } CrossList;

void CreateSM_CrossLinked(CrossList *SM)
{  s = Max(m, n);
   p = (CLNode *)malloc(sizeof(CLNode));
   p->row = m; p->col = n; //input m & n
  SM->hm = p; SM->hrc[0] = p;
   for ( i = 1; i <= s; i++ ) {
      p = (CLNode *)malloc(sizeof(CLNode));
      p->row = 0; p->col = 0; SM->hrc[i] = p;
      p->right = p; p->down = p;
      SM->hrc[i-1]->next = p; }  //初始化行列头结点
  SM->hrc[s]->next = SM->hm;
```

```
scanf(&r, &c, &v);
while (r != rend) {
   p = (CLNode *)
         malloc(sizeof(CLNode));
    p->row=r; p->col=c; p->val=v;
    q=SM->hrc[r];  //准备插入到行中
    while((q->right != SM->hrc[r])
    &&(q->right->col<c)) q=q->right;
    p->right=q->right;  q->right=p;
    q=SM->hrc[c]; //准备插入到列中
    while((q->down != SM->hrc[c])
    &&(q->down->col<r)) q=q->down;
    p->down=q->down;  q->down=p;
    scanf(&r, &c, &v);
  }
}
```



4. 稀疏矩阵的十字链表的输出/遍历算法

```
void OutputSM_CrossLinked(CrossList *SM)   //traverse by rows
  { for ( i = 1, p = SM->hm->next; p != SM->hm; i++ ) {
                        // p指向第一个行列头结点
                        // i控制行值
        for ( j = 1, q = p->right; q != p; j++ )
                              // q指向第i行的第一个非零元素结点
                              // j控制列值
            if ( j == q->col ) {
               printf( q-> val );  q = q->right; // q指向该行下一个非0元素
            }
            else printf( "\t");  //格式控制
        printf( "\n");   //格式控制
        p = p->next;   // p指向下一个行列头结点
    }
  }
```

Exercises (After class)

1. 在一个大小为**n**的整型数组中, 找出最大值最小值需要比较的次数是多少?

2. 编写如下在十字链表中查找元素的算法: 若已知**i**、**j**, 查找a_{ij}; 或已知**x**的值, 查找它在第几行列. 若找不到, 则给出相应的提示信息.

- 广义表(Lists / Generalized List)的概念和特征

广义表是n(n≥0)个数据元素的有限序列, 记作LS = (α_1, α_2, ···, α_n)

其中
- LS — 广义表的名称
- n是它的长度 (n = 0 时称为空表, 记作: LS = ())
- α_i(1≤i≤n)是广义表中的一个元素, 它或是称之为原子项(atom), 或是称为子表(sublist)的广义表[通常原子用小写字母表示, 广义表用大写字母表示]

注: 若广义表中的元素均为原子时, 即为线性表. ➜ 线性表是广义表的特例.

当广义表为非空(n≥1)时, 称表中第一个元素α_1为表头(head), 称其余元素组成的表[即(α_2, α_3, ···, α_n)]为表尾(tail). 因此,

特征
- 表头可能是一个原子项或一个广义表
- 表尾则一定是一个广义表
- 一对确定的表头和表尾能确定一个唯一的广义表
- 广义表是一种“复杂的”线性表 , 仍具有线性结构特征
- 从元素本身的异构性来看(多层次性), 广义表又属非线性结构

例子

Lists	A = ( )	B = (e)	C = (a, (b, c, d))	D = (A, B, C)	E = (a, E)	F = (())
len(LS)	0	1	2	3	2	1
head(LS)	--	e	a	A	a	( )
tail(LS)	--	( )	((b, c, d))	(B, C)	(E)	( )

- 广义表(Lists / Generalized List)ADT的基本运算

基本运算

- length(LS) – 广义表的长度
- depth(LS) – 广义表的深度
- head(LS) & tail(LS) – 广义表的表头和表尾
- initialize(LS) & create(LS) – 广义表的初始化和创建
- search(LS) / traverse(LS) – 广义表的查找或遍历
- …

例如 利用Head(LS)和Tail(LS)运算, 写出从广义表B = (a, (b, c, d)) 中分离出原子项c的过程:

Tail(B) = ((b, c, d))

Head(Tail(B)) = (b, c, d)

Tail(Head(Tail(B))) = (c, d)

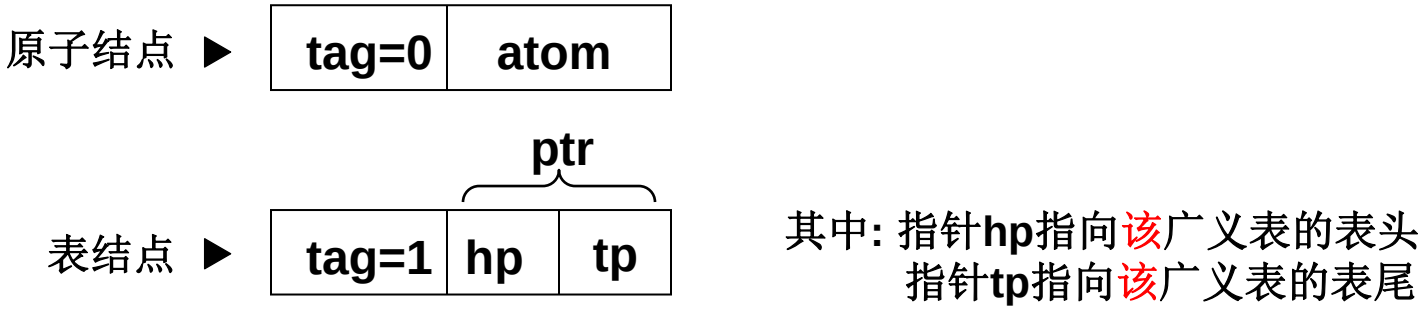
∴ Head(Tail(Head(Tail(B)))) = c

- 广义表的存储结构 – 链式存储结构(Generalized Linked List)

① 表头表尾的存储方式

② 线性链表的存储方式

- 广义表的存储结构 – 表头表尾的链式结构(P109)



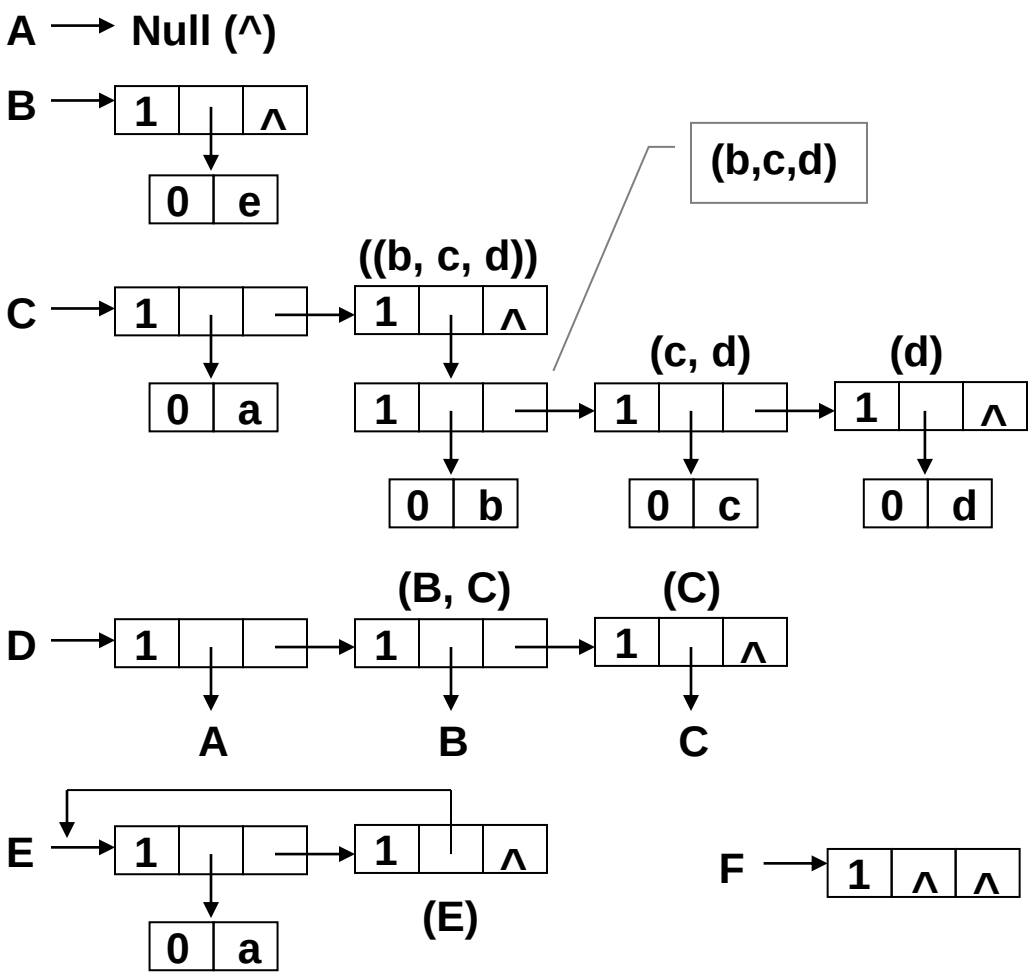
其中: 指针**hp**指向该广义表的表头
指针**tp**指向该广义表的表尾

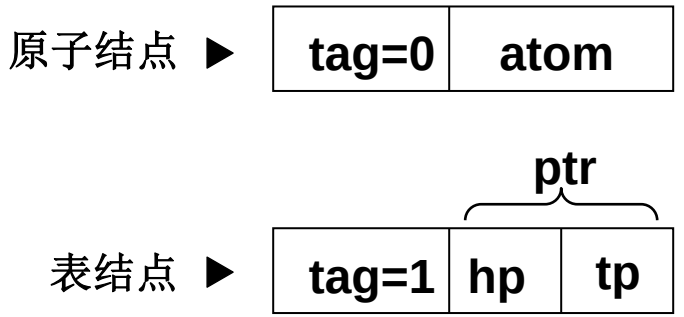
```
typedef enum { ATOM, LISTS } ElemTag;
typedef struct listsnode {
    ElemTag  tag;  //common part
    union {
        AtomType atom;
        struct { struct listsnode *hp, *tp;  } ptr;
     }  //private part
 } ListsNode, *Lists;
```

广义表的存储结构 – 表头表尾的链式结构(P109)

这种链式存储结构均由一个表头指针指向: 除空表的表头指针为空外, 对任何非空广义表, 其表头指针均指向一个表结点, 且该表结点中的hp指针指向广义表表头(原子结点/表结点), tp指针指向广义表表尾(若表尾为空表, 则tp为空; 否则tp必指向一个表结点).

画出下列广义表的基于表头表尾的链式存储结构(设表头指针用广义表的名称表示):
A = ( ) B = (e) C = (a, (b, c, d)) D = (A, B, C) E = (a, E) F = (())

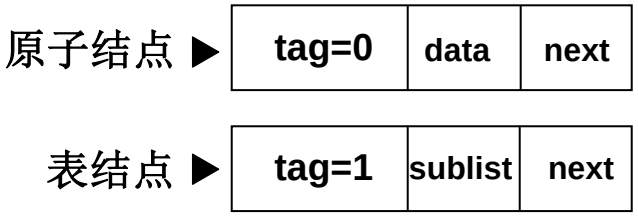




其中: 指针hp指向该广义表的表头
指针tp指向该广义表的表尾

```
typedef enum { ATOM, LISTS } ElemTag;
typedef struct listsnode {
   ElemTag  tag;  //common part
   union {
      AtomType atom;
      struct { struct listsnode *hp, *tp;  } ptr;
   }  //private part
} ListsNode, *Lists;
```

- 广义表的存储结构 – 线性链表的链式结构(P110)

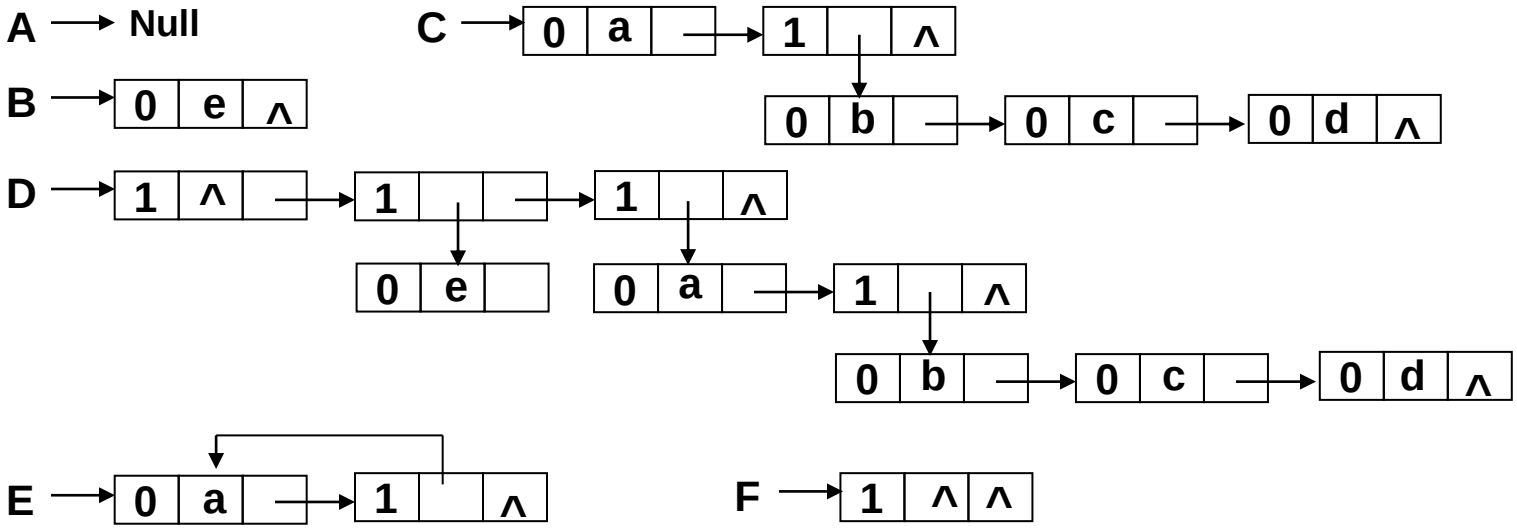


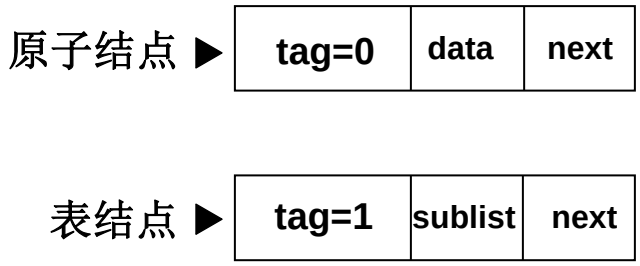

```
typedef struct listsnode {
    ElemTag  tag;  //标识原子结点或表结点
    union {
        AtomType data; //原子结点的值域
        struct listsnode *sublist; //指向子表结点的表头指针
       } val;
    struct listsnode *next; //指向该结点的后继结点的指针
} ListsNode, *Lists;
```

广义表的存储结构 – 线性链表的链式结构(P110)

广义表不带头结点的线性链式存储结构: 除空表的表头指针为空外, 对任何非空广义表, 其表头指针指向该广义表的表头结点(原子结点/表结点), 该表头结点的val或为原子值data或为一个指向子表的指针值sublist, 该表头结点的next指向其后继元素结点(空/原子结点/表结点),.

画出下列广义表的不带头结点的线性链式的存储结构(设表头指针用广义表的名称表示):
A = ( ) B = (e) C = (a, (b, c, d)) D = (A, B, C) E = (a, E) F = (())



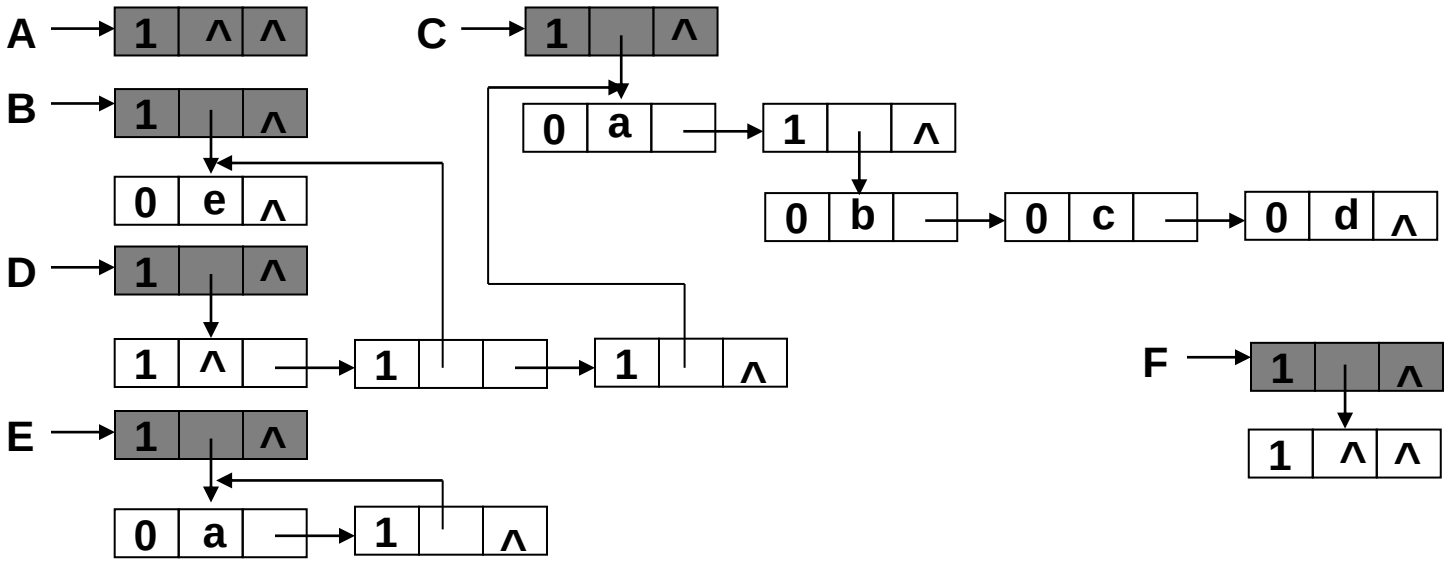


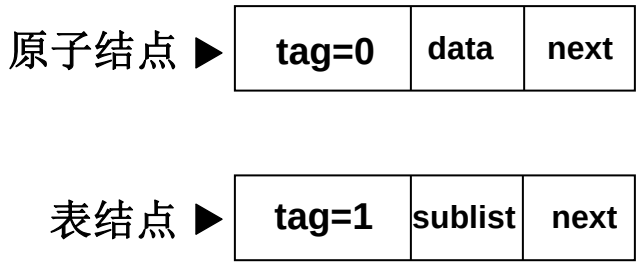

```
typedef struct listsnode {
   ElemTag  tag;  //标识原子结点或表结点
   union {
      AtomType data; //原子结点的值域
      struct listsnode *sublist; //指向子表结点的表头指针
     } val;
   struct listsnode *next; //指向该结点的后继结点的指针
} ListsNode, *Lists;
```

广义表的存储结构 – 线性链表的链式结构(P110)

广义表带有附加的头结点的线性链式存储结构: 任何广义表均由一个表头指针指向其附加的头结点. 其中, 空广义表的头结点的sublist指针域和next指针域均为Null; 非空广义表的sublist的头结点的指针域指向该广义表表头(原子结点/表结点)、 next指针域为Null.

画出下列广义表的带有头结点的线性链式的存储结构(设表头指针用广义表的名称表示):
A = ( ) B = (e) C = (a, (b, c, d)) D = (A, B, C) E = (a, E) F = (())






```
typedef struct listsnode {
   ElemTag  tag;  //标识原子结点或表结点
   union {
      AtomType data; //原子结点的值域
      struct listsnode *sublist; //指向子表结点的表头指针
     } val;
   struct listsnode *next; //指向该结点的后继结点的指针
} ListsNode, *Lists;
```

- 广义表的运算 – 基于表头表尾的存储结构

1. 广义表L的两个基本运算: GetHead(L) / Head(L) 和 GetTail(L) / Tail(L)

```
ListsNode *GetHead(Lists L) {
  if (!L) return Error; //空表无表头
  else return L->ptr.hp;
}

ListsNode *GetTail(Lists L) {
  if (!L) return Error; //空表无表尾
  else return L->ptr.tp;
}
```

- 广义表的运算 – 基于表头表尾的存储结构

2. 计算广义表L的长度

```
// LS = (α1, α2, ···, αn), n是广义表LS的长度.
// 当广义表为非空时, 称表中第一个元素α1为表头, 称其余元素组成的表(α2, α3, ···, αn)为表尾.
int GetLength(Lists L) { //recursive version
  if (!L) return 0;
  else return 1+ GetLength(L->ptr.tp); //表头元素(1个)+表尾中元素的个数
}
 ----------------------------------------------------------------------------------------------
int GetLength(Lists L) { //iterative version
  if (!L) return 0;
  while (L) { c++; L = L->ptr.tp; }
  return c;
} //for (c = 0, L; c++; L = L->ptr.tp); return c;
```

- 广义表的运算 – 基于表头表尾的存储结构

3. 广义表的查找

```
ListsNode *SearchLists(Lists L, AtomType x) {
//查找广义表L中有无原子项元素x
  if (!L) return Null;
  if (L->tag == ATOM && L->atom != x)  return Null;
  if (L->tag == ATOM && L->atom == x)  return L;
  if ( !SearchLists(L->ptr.hp, x) )  //查找广义表L的表头中有无原子项x
      return SearchLists(L->ptr.tp, x); //若广义表L表头中无原子项x, 则继续在其表尾找
}
```

```
void TraverseLists(Lists L) {
//遍历广义表L; 或者是输出广义表中的全部原子项, 比如对例子中的广义表C, 则输出a b c d
  if (!L) return;
  if (L->tag == ATOM) { printf(L->atom); return; }
  else {
     TraverseLists(L->ptr.hp);  //遍历广义表L的表头
     TraverseLists(L->ptr.tp);  //遍历广义表L的表尾
   }
}
```

- 广义表的运算 – 基于表头表尾的存储结构

4. 输出广义表

```
void OutputLists(Lists L, int ht)
//表头指针L指向广义表Lists的表结点, 控制变量ht用来标识当前输出表头(=0)还是表尾 (=1)
//base case: 1. 当L指向空表时, 输出一对括号
//base case: 2. 当L指向原子结点时, 输出该原子
//general case: 当L指向广义表时, 则先输出其表头, 再输出其表尾
{ if (!L) printf("( )");
   else if (L->tag == ATOM)  printf(L->atom);
          else {
              if (ht == 0) printf('(');
              OutputLists(L->ptr.hp, 0);
              if (!L->ptr.tp) printf(')');
              else { printf(','); OuputLists(L->ptr.tp, 1); }
          }
}
```

- 广义表的运算 – 基于表头表尾的存储结构

5. 计算广义表的深度P113/A.5.5

```
int GetDepth(Lists L) //广义表的深度可非形式化的定义为广义表中括号的重数
//广义表L的深度可递归地定义成：
//base case 1. 当L指向空表时, Depth(L)=1;  2. 当L指向原子结点时, Depth(L)=0
//general case: 当L指向非空表时, Depth(L) = 1 + Max{ Depth(αi) } (1≤i≤n)
{ if (!L) return 1;
  if (L->tag == ATOM)  return 0;
  for (max = 0, p = L; p; p = p->ptr.tp) {
     depth = GetDepth(p->ptr.hp);
      if (depth > max) max = depth; }
  return 1 + max; } //?
```

例 计算广义表A = ((), (a, (b)))深度的过程

Depth(A) = 1+ Max{Depth(α_1), Depth(α_2)} = 3

Depth(α_1) = Depth(()) = 1

Depth(α_2) = Depth((a, (b))) = 1 + Max{Depth(a), Depth((b))} = 2

Depth(a) = 0

Depth((b)) = 1 + Max{Depth(b)} = 1

Depth(b) = 0

- 广义表的运算 – 基于表头表尾的存储结构

  6. 复制一个广义表**P114/A.5.6**
  7. 创建一个广义表**P116~117/A.5.7**

- 广义表的应用 **(**略**)**

  → **(**门户**)**网站的栏目菜单;
  → 多元多项式;
  → **… …**

Exercises (After class)

1. 已知下列广义表, 画出其基于表头表尾的链式存储结构, 并计算其长度和深度, 表**A**和表**B**的表头和表尾各是什么**?**同时根据计算广义表深度的递归算法分析表**C**的深度, 且给出从表**C**中分离出原子项**a**的运算**(**运算符为**Head**和**Tail).**

① A=(a, (b, c, d), e, (f, g))

② B=((a))

③ C=(y, (z, w), (a, (x, z, w)))

2. 现假设广义表采用的是带有头结点的线性链式的存储结构, 现要求:

①写出判断该广义表为空的条件表达式;

②写出算法 **GetHead(Lists L)**来获得广义表**L**的表头.

3. 编写算法**CompareLists(Lists L1, Lists L2)**比较两个广义表是否相等**(**假设广义表的数据结构是基于表头表尾的链式结构**).**