数据结构与算法

数组和广义表

数组和广义表

5.1 数组
- 5.1.1 数组的基本概念
- 5.1.2 数组的顺序存储方式

5.2 矩阵
- 5.2.1 矩阵的定义和操作
- 5.2.2 特殊矩阵
- 5.2.3 稀疏矩阵

5.3 广义表
- 5.3.1 基本概念
- 5.3.2 广义表的存储结构

数组和广义表

数组和广义表可看成是一种特殊的线性表，其特殊在于，表中的数据元素本身也是一种线性表。

5.1 数组

- 数组是我们最熟悉的数据类型，在早期的高级语言中，数组是唯一可供使用的数据类型。
- 由于数组中各元素具有统一的类型，并且数组元素的下标一般具有固定的上界和下界，因此，数组的处理比其它复杂的结构更为简单。
- 多维数组是向量的推广。

5.1 数组

- 数组的特点是每个数据元素可以又是一个线性表结构。
- 因此，数组结构可以简单地定义为：若线性表中的数据元素为非结构的简单元素，则称为一维数组，即为向量；若一维数组中的数据元素又是一维数组结构，则称为二维数组；依次类推，若二维数组中的元素又是一个一维数组结构，则称作三维数组。

5.1.1 数组的基本概念

- 数组：由一组类型相同的数据元素构成，每一个元素可直接按序号寻址的线性表。
- 在n维数组中，每个元素受n个线性关系的约束。
- 通常很少在数组中进行插入和删除操作。

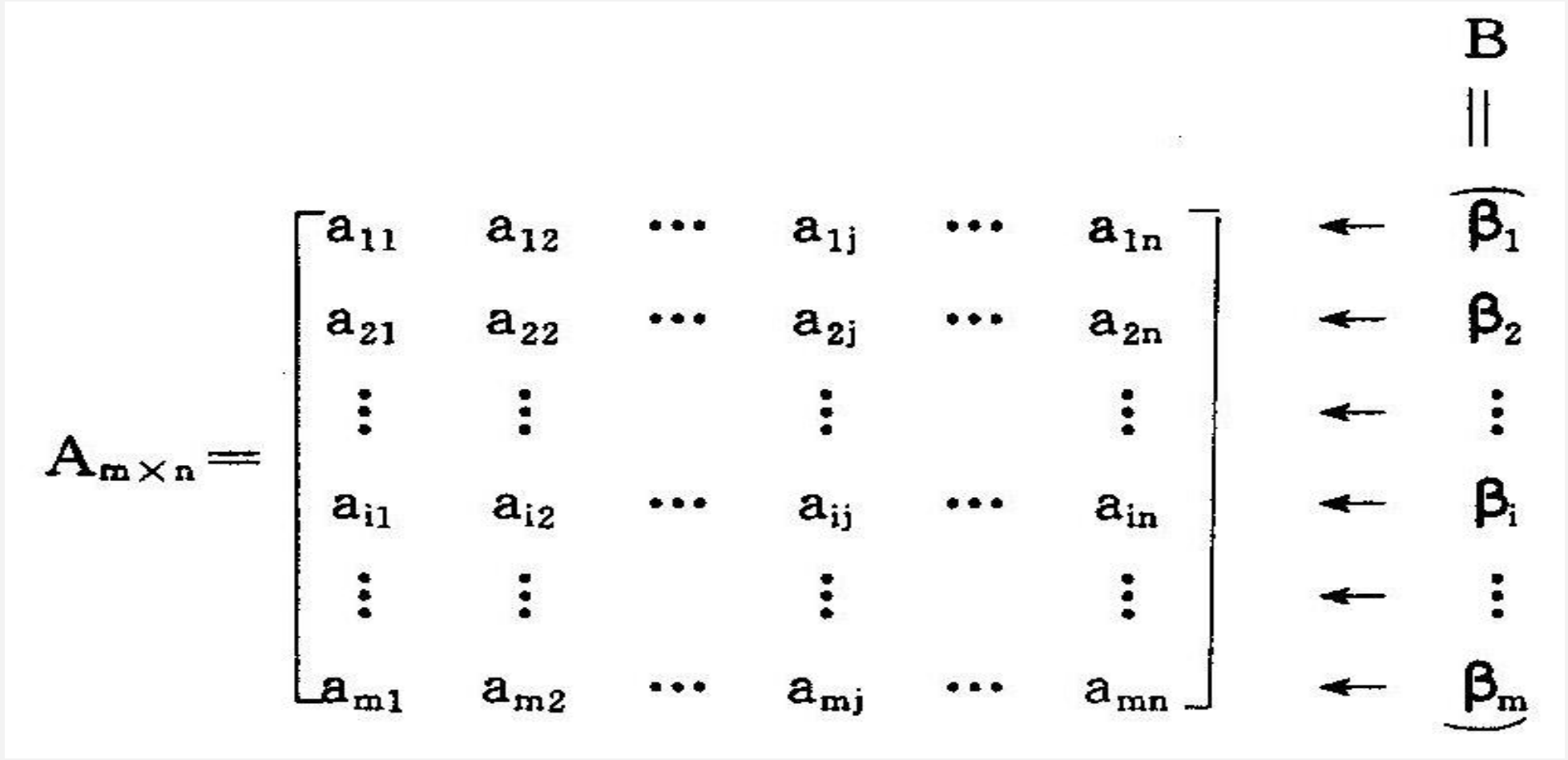
5.1.1 数组的基本概念

$$A=(\alpha_1\ \alpha_2\ \cdots\ \alpha_j\ \cdots\ \alpha_n)$$

$$A_{m\times n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1j} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2j} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mj} & \cdots & a_{mn} \end{bmatrix}$$

矩阵$A_{m\times n}$看成n个列向量的线性表

5.1.1 数组的基本概念



矩阵$A_{m\times n}$看成m个行向量的线性表

5.1.1 数组的基本概念

- 我们以二维数组为例介绍了数组的结构特性，实际上数组是一组有固定个数的元素的集合。
- 也就是说，一旦定义了数组的维数和每一维的上下限，数组中元素的个数就固定了。 例如二维数组$A_{3\times4}$，它有3行、4列，即由12个元素组成。

5.1.1 数组的基本概念

- 由于这个性质，使得对数组的操作不像对线性表的操作那样可以在表中任意一个合法的位置插入或删除一个元素。对于数组的操作一般只有两类。
  - 存取元素
  - 修改元素

5.1.2 数组的顺序存储方式

- 从理论上讲，数组结构也可以使用两种存储结构，即顺序存储结构和链式存储结构。
- 然而，由于数组结构很少进行插入、删除元素的操作，所以使用顺序存储结构更为适宜。
- 换句话说，一般的数组结构不使用链式存储结构。

5.1.2 数组的顺序存储方式

- 由于计算机的内存结构是一维的，因此用一维内存来表示多维数组，就必须按某种次序将数组元素排成一列序列，然后将这个线性序列存放在存储器中。

5.1.2 数组的顺序存储方式

- 通常有两种顺序存储方式
  - 行优先顺序(以行序为主序)——将数组元素按行排列，第i+1个行向量紧接在第i个行向量后面。
    - 以二维数组为例，按行优先顺序存储的线性序列为： $a_{11}, a_{12}, \cdots, a_{1n}, a_{21}, a_{22}, \cdots a_{2n}, \cdots\cdots, a_{m1}, a_{m2}, \cdots, a_{mn}$ 。
    - 在PASCAL、C语言中，数组就是按行优先顺序存储的。

5.1.2 数组的顺序存储方式

- 通常有两种顺序存储方式
  - 列优先顺序(以列序为主序)——将数组元素按列向量排列，第j+1个列向量紧接在第j个列向量之后.
    - A的m*n个元素按列优先顺序存储的线性序列为： $a_{11}, a_{21}, \cdots, a_{m1}, a_{12}, a_{22}, \cdots a_{m2}, \cdots\cdots, a_{n1}, a_{n2}, \cdots, a_{nm}$。
    - 在FORTRAN语言中，数组就是按列优先顺序存储的。

5.1.2 数组的顺序存储方式

$$A_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

5.1.2 数组的顺序存储方式

- 以上规则可以推广到多维数组的情况：行优先顺序可规定为先排最右的下标，从右到左，最后排最左下标；
- 列优先顺序与此相反，先排最左下标，从左向右，最后排最右下标。

5.1.2 数组的顺序存储方式

- 按上述两种方式顺序存储的序组，只要知道开始结点的存放地址（即基地址或基址），维数和每维的上、下界，以及每个数组元素所占用的单元数，就可以将数组元素的存放地址表示为其下标的线性函数。
- 因此，数组中的任一元素可以在相同的时间内存取，即顺序存储的数组是一个随机存取结构。

5.1.2 数组的顺序存储方式

- 以二维数组$A_{m\times n}$为例，假设每个元素只占一个存储单元，“以行为主”存放数组，下标从1开始，首元素a_{11}的地址为Loc［1，1］，求任意元素a_{ij}的地址。a_{ij}是排在第i行，第j列，并且前面的第i-1行有$n\times(i-1)$个元素，第 i 行第 j 个元素前面还有 j－1 个元素。由此得到如下地址计算公式：

  Loc［i，j］=Loc［1，1］+$n\times(i-1)+(j-1)$

5.1.2 数组的顺序存储方式

- 根据计算公式，可以方便地求得a_{ij}的地址是Loc［i，j］。如果每个元素占size个存储单元，则任意元素a_{ij}的地址计算公式为：

  Loc［i，j］=Loc［1，1］+(n×(i-1)+j-1)×size

5.1.2 数组的顺序存储方式

- 在C语言中，数组各维下标的下界是0，因此在C语言中，二维数组的地址计算公式为：

$$LOC(i,j)=LOC(0,0)+(i*b_2+j)*L$$

- 推广到一般情况，可以得到n维数组的数据元素存储位置的计算公式：

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

5.2 矩阵

- 矩阵是一个二维数组，它是很多科学与工程计算问题中研究的数学对象。
- 矩阵可以用行优先或列优先方法顺序存放到内存中，但是，当矩阵的阶数很大时将会占较多存储单元。
- 当里面的元素分布呈现某种规律时，这时，从节约存储单元出发，可考虑若干元素共用一个存储单元，即进行压缩存储。

5.2 矩阵

- 特殊矩阵：值相同的元素或者零元素在矩阵中的分布有一定规律。
- 稀疏矩阵：非零元较零元少，且分布没有一定规律的矩阵。

5.2.1 矩阵的定义和操作

- 矩阵可描述为二维数组。
- 矩阵的下标通常从1开始，而不是C语言的数组那样从0开始。为了统一，可以定义一个矩阵类来实现。
- 运用熟练以后，可以不需要特别的封装。

5.2.2 特殊矩阵

- 值相同的元素或零元素在矩阵中按一定的规律分布。
- 可以用特殊的方法进行存储和处理，以便提高空间和时间效率。
- 对称矩阵、三角矩阵、对角矩阵

5.2.2 特殊矩阵

- 对称矩阵:
- 在一个n阶方阵A中，若元素满足下述性质:
  $a_{ij}=a_{ji}$　　$0 \leqslant i, j \leqslant n-1$。则称A为对称矩阵。

$$A=\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 4 \\ 3 & 4 & 6 \end{bmatrix}$$

5.2.2 特殊矩阵

对称矩阵：

•对称矩阵中的元素关于主对角线对称，故只要存储矩阵中上三角或下三角中的元素，让每两个对称的元素共享一个存储空间，这样，能节约近一半的存储空间。

•不失一般性，我们按“行优先顺序”存储主对角线（包括对角线）以下的元素，元素总数为：n(n+1)/2。

5.2.2 特殊矩阵

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & & a_{2n} \\ & & & \cdots & & \\ a_{n1} & a_{n2} & \cdots & & & a_{nn} \end{bmatrix}$$

按行序为主序：

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	……	a_{n1}	……	a_{nn}
k=0	1	2	3	4		n(n-1)/2		n(n+1)/2-1

5.2.2 特殊矩阵

对称矩阵：

•将这些元素存放在一个向量s[0..n(n+1)/2-1]中。为了便于访问对称矩阵A中的元素，我们必须在a_{ij}和s[k]之间找一个对应关系。

•若i≧j，则a_{ij}在下三角形中。a_{ij}之前的i行（从第1行到第i-1行）一共有1+2+…+i-1=i(i-1)/2个元素，在第i行上，a_{ij}之前恰有j-1个元素（即$a_{i1}, a_{i2}, \cdots, a_{ij-1}$），因此：k=i(i-1)/2+j-1。

•若i<j，则a_{ij}是在上三角矩阵中。因为$a_{ij}=a_{ji}$，所以只要交换上述对应关系式中的i和j即可得到： k=j(j-1)/2+i-1

5.2.2 特殊矩阵

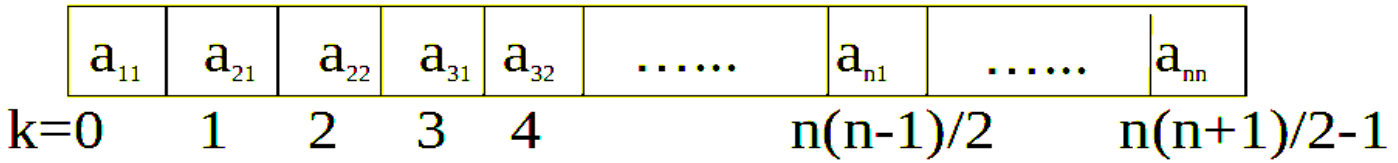
三角矩阵

- 以主对角线划分，三角矩阵有上三角和下三角两种。
- 下三角矩阵如图所示，它的上三角（不包括主对角线）中的元素均为常数。上三角矩阵正好相反，它的主对角线下方均为常数。
- 三角矩阵中的重复元素c可共享一个存储空间，其余的元素正好有n(n+1)/2个，因此，三角矩阵可压缩存储到向量s[0..n(n+1)/2]中，其中c存放在向量的最后一个分量中。
- 对称矩阵的存储方式可以应用于三角矩阵。

5.2.2 特殊矩阵

$$\begin{bmatrix} a_{11} & 0 & 0 & \cdots\cdots & 0 \\ a_{21} & a_{22} & 0 & \cdots\cdots & 0 \\ & & \cdots\cdots\cdots\cdots & & 0 \\ a_{n1} & a_{n2} & a_{n3} & \cdots\cdots & a_{nn} \end{bmatrix}$$

按行序为主序：



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

5.2.2 特殊矩阵

对角矩阵

- 我们讨论三对角矩阵的压缩存储，五对角矩阵，七对角矩阵等可以作类似分析。
- 非零元素仅出现在主对角(a_{ii}, $0 \leqq i \leqq n-1$)上，紧邻主对角线上面的那条对角线上($a_{i\,i+1}$, $0 \leqq i \leqq n-2$)和紧邻主对角线下面的那条对角线上($a_{i+1\,i}$, $0 \leqq i \leqq n-2$)。显然，当$|i-j|>1$时，元素$a_{ij}=0$。

5.2.2 特殊矩阵

对角矩阵

- 由此可知，一个k对角矩阵(k为奇数)A是满足下述条件的矩阵：若 | i-j | >(k-1)/2 ，则元素 $a_{ij}=0$。
- 对这种三对角矩阵，我们也可按行优序为主序来存储。除第1行和第n行是2个元素外，每行的非零元素都要是3个，因此，需存储的元素个数为3n-2。

5.2.2 特殊矩阵

$$
\begin{bmatrix}
a_{11} & a_{12} & 0 & \cdots\cdots\cdots\cdots & 0 \\
a_{21} & a_{22} & a_{23} & 0 \cdots\cdots\cdots\cdots & 0 \\
0 & a_{32} & a_{33} & a_{34} \; 0 \cdots\cdots & 0 \\
\multicolumn{5}{c}{\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots} \\
0 & 0 & \cdots & a_{n-1,n-2} \; a_{n-1,n-1} & a_{n-1,n} \\
0 & 0 & \cdots & \cdots a_{n,n-1} & a_{nn}
\end{bmatrix}
$$

按行序为主序：

a11	a12	a21	a22	a23	……	……	ann-1	ann
k=0	1	2	3	4		n(n-1)/2		n(n+1)/2-1

5.2.2 特殊矩阵

总结：上述的三种特殊矩阵，其非零元素的分布都是有规律的，因此总能找到一种方法将它们压缩存储到一个向量中，并且一般都能找到矩阵中的元素与该向量的对应关系，通过这个关系，仍能对矩阵的元素进行随机存取。

5.2.3 稀疏矩阵

简单说，设矩阵A中有s个非零元素，若s远远小于矩阵元素的总数（即s≦m×n），则称A为稀疏矩阵。

5.2.3 稀疏矩阵

- 设在的矩阵A中，有s个非零元素。令 e=s/(m*n)，称e为矩阵的稀疏因子。通常认为e≦0.05时称之为稀疏矩阵。
- 在存储稀疏矩阵时，为了节省存储单元，很自然地想到使用压缩存储方法。

5.2.3 稀疏矩阵

以常规方法，即以二维数组表示高阶的稀疏矩阵时产生的问题

- 零值元素占了很大空间
- 计算中进行了很多和零值的运算，遇除法还需判别除数是否为零

5.2.3 稀疏矩阵

解决问题的原则
- 尽可能少存或不存零值元素
- 尽可能减少没有实际意义的运算
- 操作方便，即能尽可能快地找到与下标值(i，j)对应的元素

5.2.3 稀疏矩阵

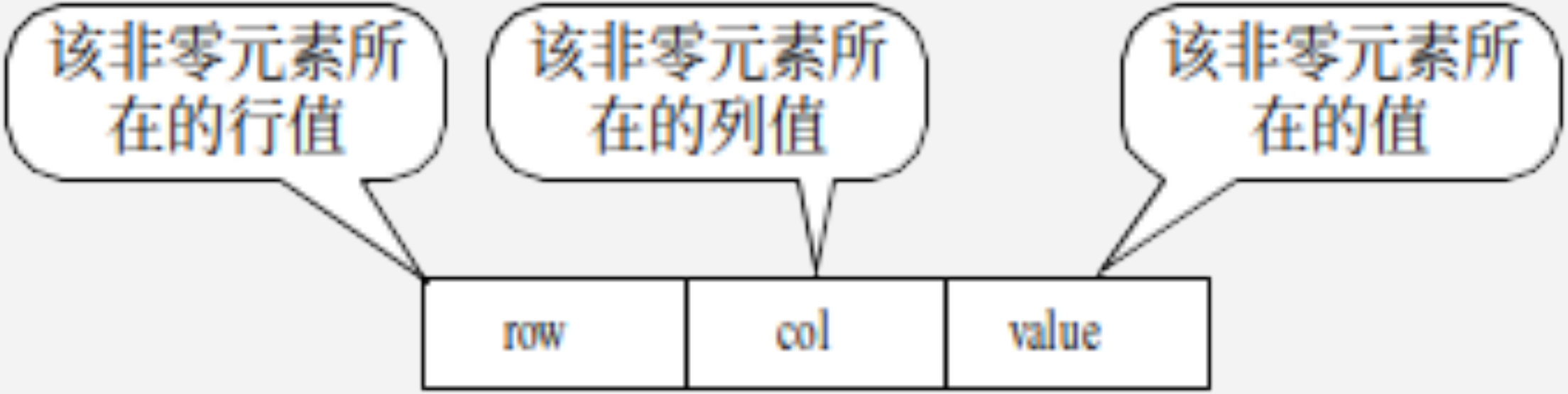
- 由于非零元素的分布一般是没有规律的，因此在存储非零元素的同时，还必须同时记下它所在的行和列的位置（i，j）。
- 一个三元组(i，j，a_{ij})唯一确定了矩阵A的一个非零元。因此，稀疏矩阵可由表示非零元的三元组及其行列数唯一确定。

5.2.3 稀疏矩阵

$$M_{6\times7}=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}$$

$$N_{7\times6}=\begin{bmatrix} 0 & 0 & 3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

5.2.3 稀疏矩阵



5.2.3 稀疏矩阵

1	1	2	12
2	1	3	9
3	3	1	−3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	−7

(a) 矩阵 M 的三元组表

1	1	3	3
2	1	6	15
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7	4	6	−7
8	6	3	14

(b) 矩阵 N 的三元组表

5.2.3 稀疏矩阵 - 基本操作

(1) int GetRows() const
初始条件: 稀疏矩阵已存在。
操作结果: 返回稀疏矩阵行数。
(2) int GetCols() const
初始条件: 稀疏矩阵已存在。
操作结果: 返回稀疏矩阵列数。
(3) int GetNum() const
初始条件: 稀疏矩阵已存在。
操作结果: 返回稀疏矩阵非零元素个数。

5.2.3 稀疏矩阵 - 基本操作

(4) bool Empty() const
  初始条件：稀疏矩阵已存在。
  操作结果：如矩阵为空，则返回true，否则返回false
(5) StatusCode SetElem(int r,int c,const ElemType &v)
  初始条件：稀疏矩阵已存在。
  操作结果：设置指定位置的元素值。
(6) StatusCode GetElem(int r, int c, ElemType &v)
  初始条件：稀疏矩阵已存在。
  操作结果：求指定位置的元素值。

5.2.3 稀疏矩阵 – 基本操作

- 稀疏矩阵的顺序压缩存储
  三元组顺序表
- 稀疏矩阵的链式压缩存储
  十字链表

5.2.3.1 三元组顺序表

- 假设以顺序存储来表示三元组表，则可得到稀疏矩阵的一种压缩存储方法——三元组顺序表。
- 三元组顺序表重点介绍矩阵的转置运算。包括2种方法，一种是以转置后的矩阵三元组为出发点的算法（简单转置算法），一种是以原矩阵三元组为出发点的算法（快速转置算法）。

5.2.3.1 三元组顺序表

```
// 三元组类模板
template<class ElemType>
struct Triple
{
    // 数据成员:
    int row, col;       // 非零元素的行下标与列下标
    ElemType value;     // 非零元素的值

    // 构造函数模板:
    Triple();           // 无参数的构造函数模板
    Triple(int r, int c, ElemType v);
                        // 已知数据域建立三元组
};
```

5.2.3.1 三元组顺序表

```
// 稀疏矩阵三元组顺序表类模板
template<class ElemType>
class TriSparseMatrix
{
protected:
    // 稀疏矩阵三元组顺序表的数据成员:
    Triple<ElemType> *triElems;    // 存储稀疏矩阵的三元组表
    int maxSize;                   // 非零元素最大个数
    int rows, cols, num;           // 稀疏矩阵的行数,列数及非零元个数
```

5.2.3.1 三元组顺序表

```
public:
    // 抽象数据类型方法声明及重载编译系统默认方法声明:
    TriSparseMatrix(int rs = DEFAULT_SIZE, int cs = DEFAULT_SIZE, int size =
DEFAULT_SIZE);  // 构造一个rs行cs列非零元素最大个数为size的空稀疏矩阵
    ~TriSparseMatrix();  // 析构函数
    int GetRows() const; // 返回稀疏矩阵行数
    int GetCols() const; // 返回稀疏矩阵列数
    int GetNum() const;  // 返回稀疏矩阵非零元个数
    StatusCode SetElem(int r, int c, const ElemType &v);
                                        // 设置指定位置的元素值
    StatusCode GetElem(int r, int c, ElemType &v);
                                       // 求指定位置的元素值
    TriSparseMatrix<ElemType> &operator =(const TriSparseMatrix<ElemType> &copy);
                                       // 赋值……
};
```

5.2.3.1 三元组顺序表

一个m×n的矩阵A，它的转置B是一个n×m的矩阵，且a[i][j]=b[j][i]，0≦i≦m，0≦j≦n，即A的行是B的列，A的列是B的行。将A转置为B，就是将A的三元组表a.data置换为表B的三元组表b.data。

5.2.3.1 三元组顺序表

- 一个矩阵的转置
  - 将矩阵的行列值相互交换。
  - 将每一个三元组中的i，j相互调换。
  - 重新排列三元组之间的次序。
- 如何实现上面的3条，有2种处理方法。

简单转置算法

- 算法的基本思想：第一次从转置前稀疏矩阵source中取出应该放置到转置后的稀疏矩阵dest中第一个位置的元素，行列号互换后，放于dest中第一个位置；第二次从source中选取应该放到dest中的第二个位置的元素，……，如此进行，依次生成dest中的各元素。

简单转置算法

- 按照这种思想设计的算法：对source中的每一列col(0≦col≦n-1)，通过从头至尾扫描三元表source.data，找出所有列号等于col的那些三元组，将它们的行号和列号互换后依次放入dest.data中，即可得到dest的按行优先的压缩存储表示。
- 由于转置后列号变行号，所以转置后元素的行序排列实质上是原矩阵元素的列序排列。

简单转置算法

实现算法可形式化描述为:

```
destPos = 0; // 稀疏矩阵dest的下一个三元组的存放位置
for(col = 最小列号; col <=最大列号; col++)
{
   在source中从头查找有无列号为col的三元组;
   若有，则将其行、列号交换后，依次存入dest
   中destPos所指位置，同时destPos加1;
}
```

简单转置算法

- 分析这个算法，主要的工作是在source和col的两个循环中完成的，故算法的时间复杂度为O(nu*tu)，即矩阵的列数和非零元的个数的乘积成正比。
- 而一般传统矩阵的转置算法其时间复杂度为O(nu*mu)。当非零元素的个数tu和mu*nu同数量级时，算法transmatrix的时间复杂度为$O(mu*nu^2)$。
- 三元组顺序表虽然节省了存储空间，但时间复杂度比一般矩阵转置的算法还要复杂，同时还有可能增加算法的难度。因此，此算法仅适用于tu<<mu*nu的情况。

快速转置算法

- 依次按三元组表A的次序进行转置，转置后直接放到三元组表B的正确位置上。这种转置算法称为快速转置算法。
- 为了能将待转置三元组表A中元素一次定位到三元组表B的正确位置上，需要预先计算以下数据：
  - 待转置矩阵每一列中非零元素的个数（即转置后矩阵每一行中非零元素的个数）。
  - 待转置矩阵每一列中第一个非零元素在三元组表B中的正确位置（即转置后矩阵每一行中第一个非零元素在三元组B中的正确位置）。

快速转置算法

- 为此，需要设置两个一维数组num[0..n]和cpot[0..n]。
- num[0..n]：统计M中每列非零元素的个数。
- cpot[0..n]：由递推关系得出M中的每列第一个非零元素在B中的位置。

  算法通过cpot数组建立位置对应关系：

  cpot[1]=0

  cpot[col]=cpot[col-1]+num[col-1]

  2<=col<=a.nu

快速转置算法

矩阵M和相应的三元组A可以求得num[col]和 cpot[col]的值如下

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

快速转置算法

- 快速转置算法的时间主要耗费在多个并列的单循环上，每个单循环分别执行了nu，tu，因而总的时间复杂度为O（nu+tu）。当待转置矩阵M中非零元素个数接近于mu×nu 时，其时间复杂度接近于经典算法的时间复杂度O(mu×nu)。
- 快速转置算法在空间耗费上除了三元组表所占用的空间外，还需要两个辅助向量空间。可见，算法在时间上的节省，是以更多的存储空间为代价的。

5.2.3.2 十字链表

- 与用二维数组存储稀疏矩阵比较，用三元组表表示的稀疏矩阵不仅节约了空间，而且使得矩阵某些运算的运算时间比经典算法还少。
- 但是在进行矩阵加法、减法和乘法等运算时，有时矩阵中的非零元素的位置和个数会发生很大的变化。如A=A+B，将矩阵B加到矩阵A上，此时若还用三元组表表示法，势必会为了保持三元组表“以行序为主序”而大量移动元素。

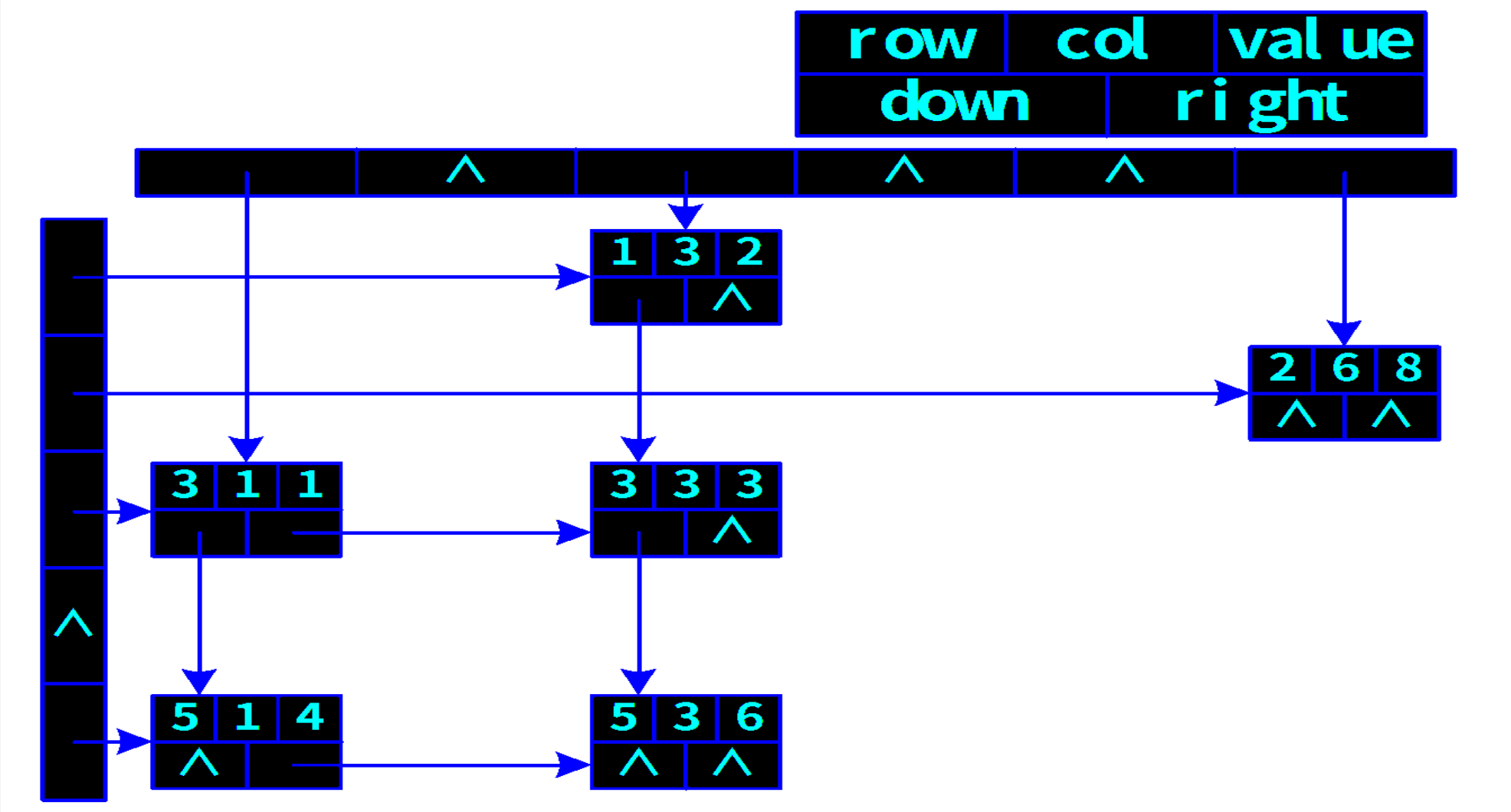
5.2.3.2 十字链表

- 当矩阵中非零元素的个数和位置经过运算后变化较大时，就不宜采用三元组顺序表，而应采用链式存储结构来表示三元组。
- 由于稀疏矩阵的链式存储表示最终形成了一个十字交叉的链表，所以这种存储结构叫做十字链表。

5.2.3.2 十字链表

- 在十字链表中，矩阵的每一个非零元素用一个结点表示，该结点除了（row，col，value）以外，还要有以下两个链域：
  right：用于链接同一行中的下一个非零元素；
  down：用于链接同一列中的下一个非零元素。
- 为了能够快速找到各个行列链表，可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。

5.2.3.2 十字链表



5.3 广义表

- 广义表（Lists，又称列表）是线性表的推广。
- 在第2章中，我们把线性表定义为n>=0个元素 $a_1, a_2, a_3, \cdots, a_n$ 的有限序列。线性表的元素仅限于原子项，原子是作为结构上不可分割的成分，它可以是一个数或一个结构。
- 若放松对表元素的这种限制，容许它们具有其自身结构，这样就产生了广义表的概念。

5.3.1 广义表的基本概念

- 广义表通常简称为表，由n(>=0)个表元素组成的有限序列，记作 GL = $(a_1, a_2, a_3, \cdots, a_n)$，GL是表名，$a_i$为表元素,简称为元素，它可以是表(称为子表元素，简称为子表)，可以是数据元素(称为原子元素，简称为原子)。

5.3.1 广义表的基本概念

- n为表的长度。n=0的广义表为空表。
- n>0时，表的第一个表元素称为广义表的表头(head)，除此之外，其它表元素组成的表称为广义表的表尾(tail)。

5.3.1 广义表的基本概念

- 在线性表的定义中a_i仅限于单个元素，而在广义表中它即可以是单个元素也可以是广义表。分别称为广义表的原子和子表。习惯上用大写表示广义表，用小写表示原子。
- 广义表的定义是一个递归的定义，因为在描述广义表的时候又用到了广义表的概念。
- 由于广义表中的元素又可以是广义表，因此对于广义表有深度的概念。

广义表举例

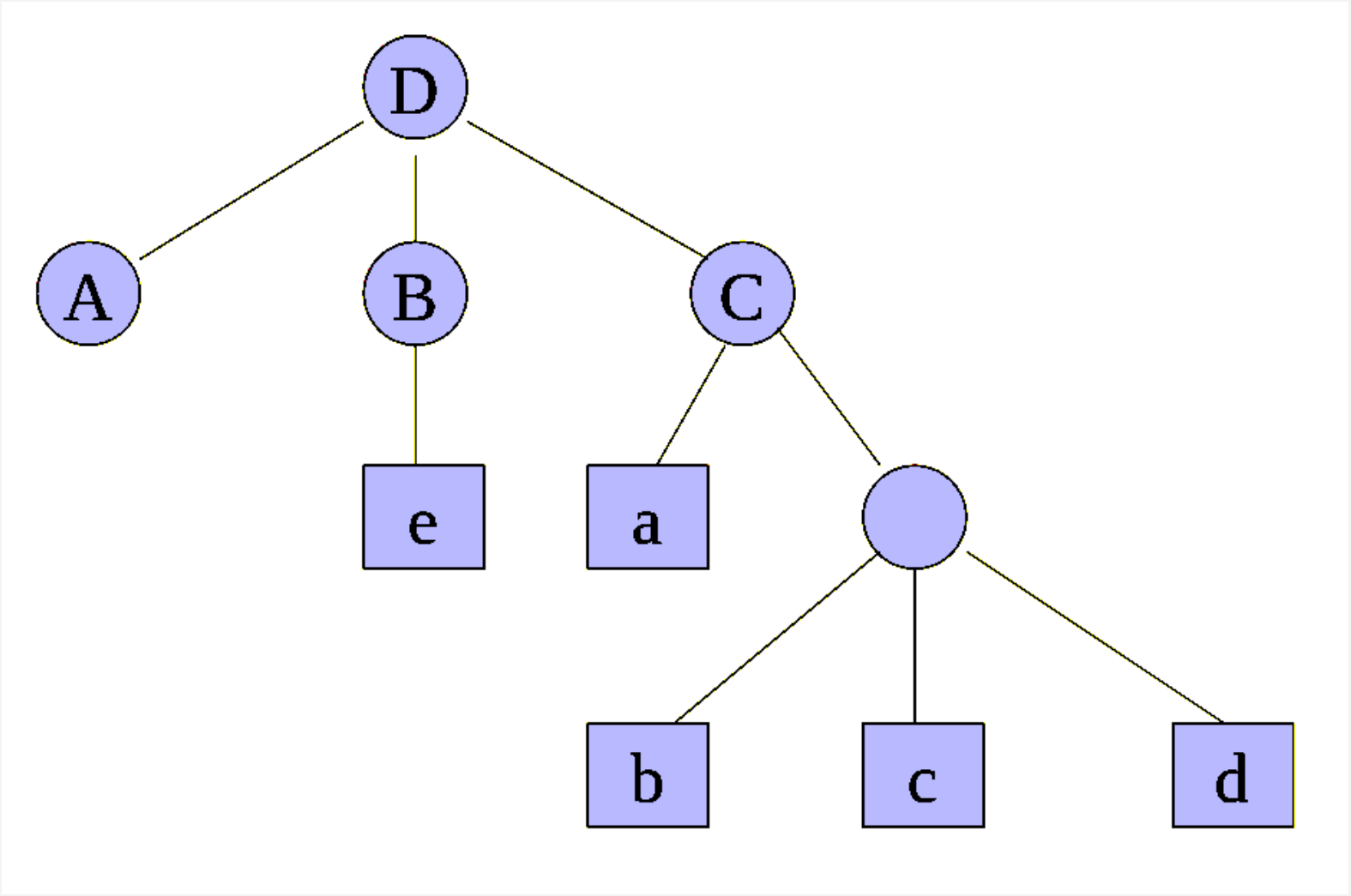
- A=( )——A是一个空表，它的长度为0。
- B=(e)——广义表B只有一个原子，长度为1。
- C=(a,(b,c,d))——列表C的长度为2，两个元素分别为原子a和子表(b,c,d)。
- D=(A,B,C)——广义表D的长度为3，3个元素都是列表。
- E=(a,E)——这是一个递归的表，长度为2。E相当于一个无限的列表。

5.3.1 广义表的基本概念

定义和例子可以得到广义表的3个重要结论

- 广义表的元素可以是子表，而子表的元素还可以是子表。由此，广义表是一个多层次的结构，可以用图形象地表示。
- 广义表可以为其他广义表所共享
- 广义表可以是一个递归的表，即广义表也可以是其本身的一个子表。

5.3.1 广义表的基本概念



5.3.1 广义表的基本概念

根据广义表表头和表尾的定义还有如下性质

- 任何一个非空列表其表头可能是原子，也可能是广义表。
- 任何一个非空列表的表尾必定为广义表。

例如：

GetHead(B)=e, GetTail(B)=( );

GetHead(D)=A, GetTail(D)=(B,C);

GetHead(B,C)=B, GetTail(B,C)=(C);

5.3.1 广义表的基本概念

- 广义表的深度本质上就是广义表表达式中括号的最大嵌套层数，与长度概念不同。
- 广义表GL的深度Depth(GL)定义如下

$$Depth(GL)=\begin{cases}0 & GL\text{为原子元素}\\ 1 & GL\text{为空表}\\ 1+Max(Depth(a_i)\,|\,1\le i\le n) & \text{其它情况}\end{cases}$$

5.3.1 广义表的基本概念

例如：

2	深度为0(括号层数为0)
( )	深度为1(括号层数为1)
(2, (3, 6))	深度为2(括号层数为2)

广义表的基本操作

1. GenListNode<ElemType> *First() const
初始条件：广义表已存在。
操作结果：返回广义表的第一个元素。
2. GenListNode<ElemType> *Next(GenListNode<ElemType> *elemPtr) const
初始条件：广义表已存在，elemPtr指向广义表的元素。
操作结果：返回elemPtr指向的广义表元素的后继
3. bool Empty() const
初始条件：广义表已存在。
操作结果：如广义表为空，则返回true，否则返回false

广义表的基本操作

4. void Push(const ElemType &e)
初始条件：广义表已存在。
操作结果：将元子元素e作为表头加入到广义表最前面。
5. void Push(GenList<ElemType> &subList)
初始条件：广义表已存在。
操作结果：将子表subList作为表头加入到广义表最前面。
6. int Depth()
初始条件：广义表已存在。
操作结果：返回广义表的深度。

5.3.2 广义表的存储结构

- 由于广义表中的数据元素可以具有不同的结构（或是原子，或是列表），因此难以用顺序表的存储结构来表示，通常采用链式存储结构，每个数据元素可用一个结点来表示。
- 这里主要介绍常用的借助引用数链式存储结构——引用数法广义表。

5.3.2 广义表的存储结构

- 广义表一般需要两种结构的结点：一种是表结点，用来表示列表；一种是原子结点，用来表示原子。
- 为了方便起见，引用数法广义表还在链表的前面加上头结点，形成第三种节点。

5.3.2 广义表的存储结构

引用数法广义表：在这种方法中，每一个表结点由三个域组成，结点可分为三类

- 头结点：用标志域tag=HEAD标识，数据域ref用于存储引用数，广义表的引用数表示能访问此广义表的广义表或指针个数。引用数可用于是否删除表的判断。
- 原子结点：用标志域tag=ATOM标识，原子元素用原子结点存储，数据域atom用于存储原子元素的值。
- 表结点：用标志域tag=LIST标识，指针域subLink用于存储指向子表头结点的指针。

5.3.2 广义表的存储结构

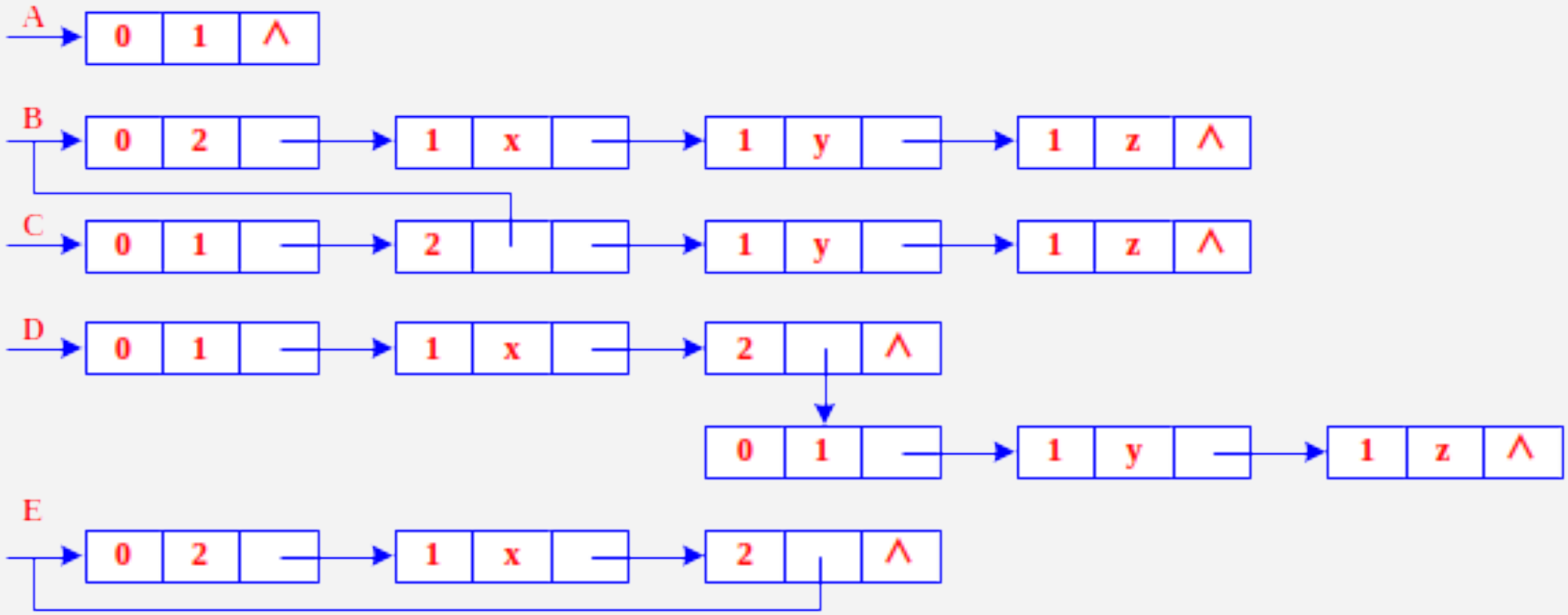
(a) 头结点

(b) 原子结点

(c) 表结点

5.3.2 广义表的存储结构

A=()，B=(x，y，z)，C=(B，y，z)，D=(x,(y，z))，E=(x，E)



5.3.2 广义表的存储结构

引用数法广义表的特点

•广义表中的所有表都带有头结点，空表也不例外。优点是便于操作。

- 特别是广义表被共享时，当删除该表的第一个元素，则要删除此元素对应的节点，头结点的存在，从而不用修改任何指向该子表的指针。
- 便于释放广义表。

5.3.2 广义表的存储结构

引用数法广义表的特点

- 表中节点的层次分明。
- 容易计算表的长度。注意与深度的区别。沿着nextLink指针即可计算长度。

练习

已知广义表LS＝((a,b,c),(d,e,f)),运用head和tail函数取出LS中原子e的运算是(C　)。

A. head(tail(LS))
B. tail(head(LS))
C. head(tail(head(tail(LS)))
D. head(tail(tail(head(LS))))

已知广义表L=（（x,y,z），a，（u，t，w））,从L表中取出原子项t的运算是（ head（tail(head（tail（tail（L））））））

练习

设对称矩阵 $A=\begin{bmatrix}1&0&0&2\\0&3&0&0\\0&0&0&5\\2&0&5&0\end{bmatrix}$

（1）若将 A 中包括主对角线的下三角元素按列的顺序压缩到数组 S 中，即 S:

1	0	0	2	3	0	0	0	5	0
下标：1	2	3	4	5	6	7	8	9	10

试求出A中任一元素的行列下标[i，j](1<=i，j<=4)与S中元素的下标K之间的关系.

（2）若将 A 视为稀疏矩阵时，画出其三元组表形式压缩存储表。

（1）k=(2n-j+2)(j-1)/2+i-j+1　　（当i≥j时，本题n=4）

　　k=(2n-i+2)(i-1)/2+j-i+1　　（当i<j时，本题n=4）

（2）稀疏矩阵的三元组表为：s=((4,4,6),(1,1,1),(1,4,2),(2,2,3),(3,4,5),(4,1,2),(4,3,5)。其中第一个三元组是稀疏矩阵行数、列数和非零元素个数。其它三元组均为非零元素行值、列值和元素值。