第6章 数组和广义表

关联的知识点

- 理解数组和一般线性表之间的差异。
- 重点掌握数组的顺序存储结构和元素地址计算方法。
- 掌握各种特殊矩阵如对称矩阵、上、下三角矩阵和对角矩阵的压缩存储方法。
- 掌握稀疏矩阵的各种存储结构以及基本运算实现算法。
- 灵活运用数组这种数据结构解决一些综合应用问题。

6.1 数　组

关联的知识点

- 数组的逻辑结构特性
- 数组的顺序存储结构及其特点
- 对称矩阵、上三角矩阵、下三角矩阵和 三对角矩阵的压缩存储

6.1.1 数组的基本概念

从逻辑结构上看，数组A是n（$n>1$）个相同类型数据元素a_1、a_2、…、a_n构成的有限序列，其逻辑表示为：

$$A=(a_1, a_2, \ldots, a_n)$$

其中，a_i（$1\leq i\leq n$）表示数组A的第i个元素。

一个二维数组可以看作是每个数据元素都是相同类型的一维数组的一维数组。以此类推，任何多维数组都可以看作一个线性表，这时线性表中的每个数据元素也是一个线性表。

推广到d（d≥3）维数组，不妨把它看作一个由$d-1$维数组作为数据元素的线性表；或者这样理解，它是一种较复杂的线性表结构，由简单的数据结构即线性表——辗转合成而得。

由此看出，多维数组是线性表的推广。

数组抽象数据类型＝逻辑结构＋基本运算（运算描述）

数组的基本运算如下：

- Value(A,$index_1$,$index_2$,…,$index_d$)：A是已存在的d维数组，$index_1$，$index_2$，…，$index_d$是指定的d个下标值，且这些下标均未越界。其运算结果是返回由上述下标指定的A中的对应元素的值。
- Assign(A,e,$index_1$,$index_2$,…,$index_d$)：A是已存在的d维数组，e为元素变量，$index_1$，$index_2$，…，$index_d$ 是指定的d个下标值，且这些下标均未越界。其运算结果是将e的值赋给由上述下标指定的A中的对应元素。
- ADisp(A,b_1,b_2,…,b_d)：输出d维数组A的所有元素值。

…

6.1.2　数组的存储结构

从存储结构上看，数组的所有元素存储在一块地址连续的内存单元中，这是一种顺序存储结构。

几乎所有的计算机语言都支持数组类型，以C/C++语言为例，其中数组数据类型具有以下性质：

（1）数组中的数据元素数目固定。一旦定义了一个数组，其数据元素数目不再有增减变化。

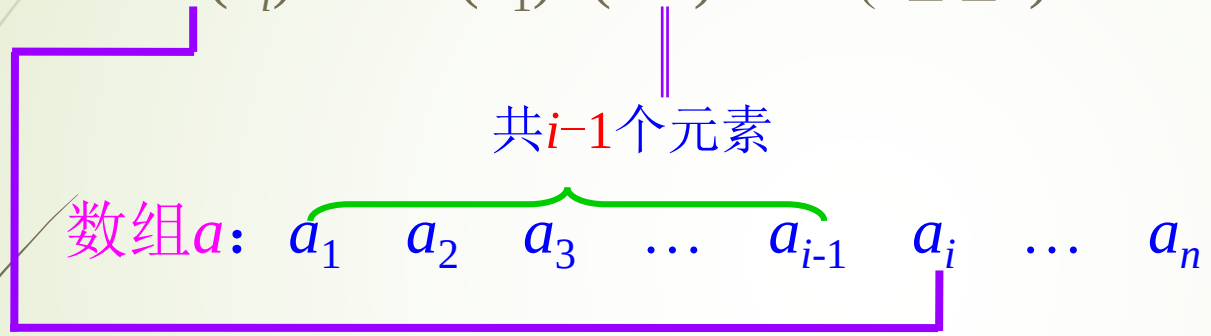
（2）数组中的数据元素具有相同的数据类型。

（3）数组中的每个数据元素都和一组唯一的下标值对应。

（4）数组是一种随机存储结构。可随机存取数组中的任意数据元素。

在一维数组中，一旦a_1的存储地址LOC(a_1)确定，并假设每个数据元素占用k个存储单元，则任一数据元素a_i的存储地址LOC(a_i)就可由以下公式求出：

$$LOC(a_i)=LOC(a_1)+(i-1)*k \quad (0\le i\le n)$$



该式说明，一维数组中任一数据元素的存储地址可直接计算得到，即一维数组中任一数据元素可直接存取，因此一维数组是一种随机存储结构。同样，二维及多维数组也满足随机存储特性。

对于一个m行n列的二维数组$A_{m\times n}$,有：

$$A_{m\times n}=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \cdots & & & \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

将$A_{m\times n}$简记为，A是这样的一维数组：

$A=(a_1,a_2,\ldots,a_i\ldots,a_m)$

其中，$a_i=(a_{i,1},a_{i,2},\ldots,a_{i,n})$ （$1\le i\le m$）

对于二维数组来说，由于计算机的存储结构是线性的，如何用线性的存储结构存放二维数组元素就有一个行 / 列次序排放问题。

以行序为主序的存储方式：即先存储第1行,然后紧接着存储第m行。此时二维数组的线性排列次序为：

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

$$\underbrace{a_{1,1},\ a_{1,2},\ \ldots,a_{1,n}}_{\text{第1行的元素}},\ \underbrace{a_{2,1},\ a_{2,2},\ \ldots,a_{2,n}}_{\text{第2行的元素}},\ \ldots,\ \underbrace{a_{m,1},\ a_{m,2},\ \ldots,\ a_{m,n}}_{\text{第}m\text{行的元素}}$$

以行序为主序的存储方式：

$a_{1,1}$，$a_{1,2}$，…，$a_{1,n}$，…，$a_{i,1}$，$a_{i,2}$，…，$a_{i,j}$，…$a_{i,n}$，…

第1行的元素　　第i行的元素

$a_{i,j}$元素前有1~i−1行，每行n个元素，计$n\times(i-1)$个元素

在第i行中，$a_{i,j}$元素前有j-1个元素

则$a_{i,j}$元素前共有$n\times(i-1)+j-1$个元素

所以，当二维数组第一个数据元素$a_{1,1}$的存储地址LOC($a_{1,1}$)和每个数据元素所占用的存储单元k确定后，则该二维数组中任一数据元素$a_{i,j}$的存储地址可由下式确定：

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

同理可推出在以列序为主序的计算机系统中有：

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

其中m为行数。

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

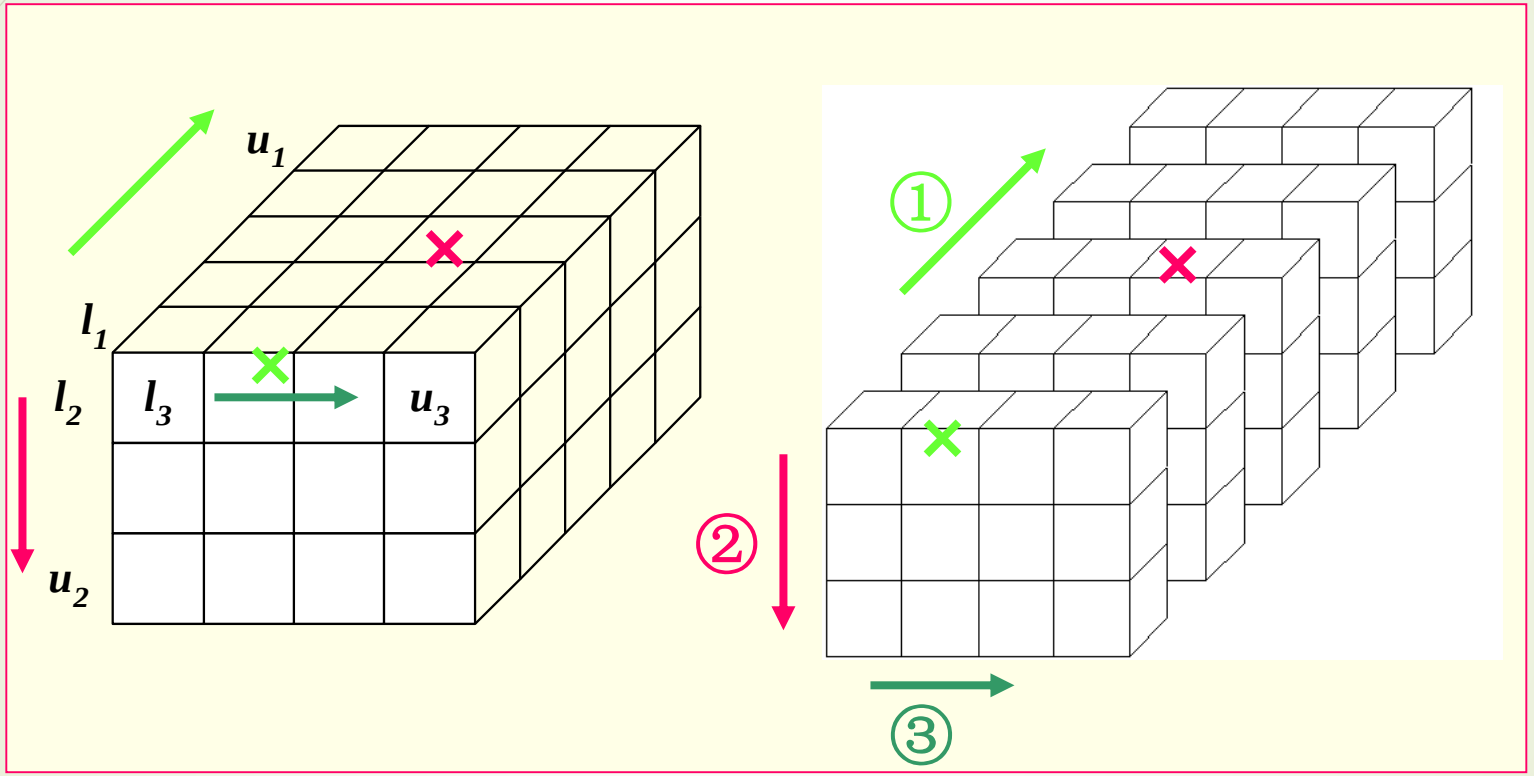
这一过程可以推广到d（$d>2$）的多维数组。

所以，数组采用顺序存储结构时，具有随机存取特性。

是指给定序号i，可以在时间复杂度为O(1)的时间内找到相应的元素。

三维数组

按页/行/列存放，页优先的顺序存储



三维数组

- a[m1][m2] [m3] 各维元素个数为 m_1, m_2, m_3
- 下标为 i_1, i_2, i_3的数组元素的存储位置：

$$LOC(i_1, i_2, i_3) = a + (i_1-1)* m_2 * m_3 + (i_2-1)* m_3 + i_3-1$$

前i_1-1页总元素个数

第i_1页的前i_2-1行总元素个数

第 i_2 行前 i_3-1 列元素个数

n维数组

- 各维元素个数为 $m_1\ m_2, m_3, \ldots, m_n$
- 下标为 $i_1, i_2, i_3 \ldots, i_n$ 的数组元素的存储位置：

$$LOC(i_1, i_2, \cdots, i_n) = a + (i_1 - 1) * m_2 * m_3 * \cdots * m_n +$$

$$+ (i_2 - 1) * m_3 * m_4 * \cdots * m_n + \cdots\cdots + (i_{n-1} - 1) * m_n + i_n - 1$$

$$= a + \left(\sum_{j=1}^{n-1} i_j * \prod_{k=j+1}^{n} m_k \right) + i_n - 1$$

【例6.1：p135】C/C++中二维数组float $a[5][4]$的起始地址为2000，且每个数组元素长度为32位（即4个字节），求数组元素$a[3][2]$的内存地址。

解：由于C/C++中数组的行、列下界均为0，又由于C/C++数组采用行序为主序的存储方式，有：

$$LOC(a_{3,2})=LOC(a_{0,0})+(i\times 4+j)\times k$$

k表示是4个字节

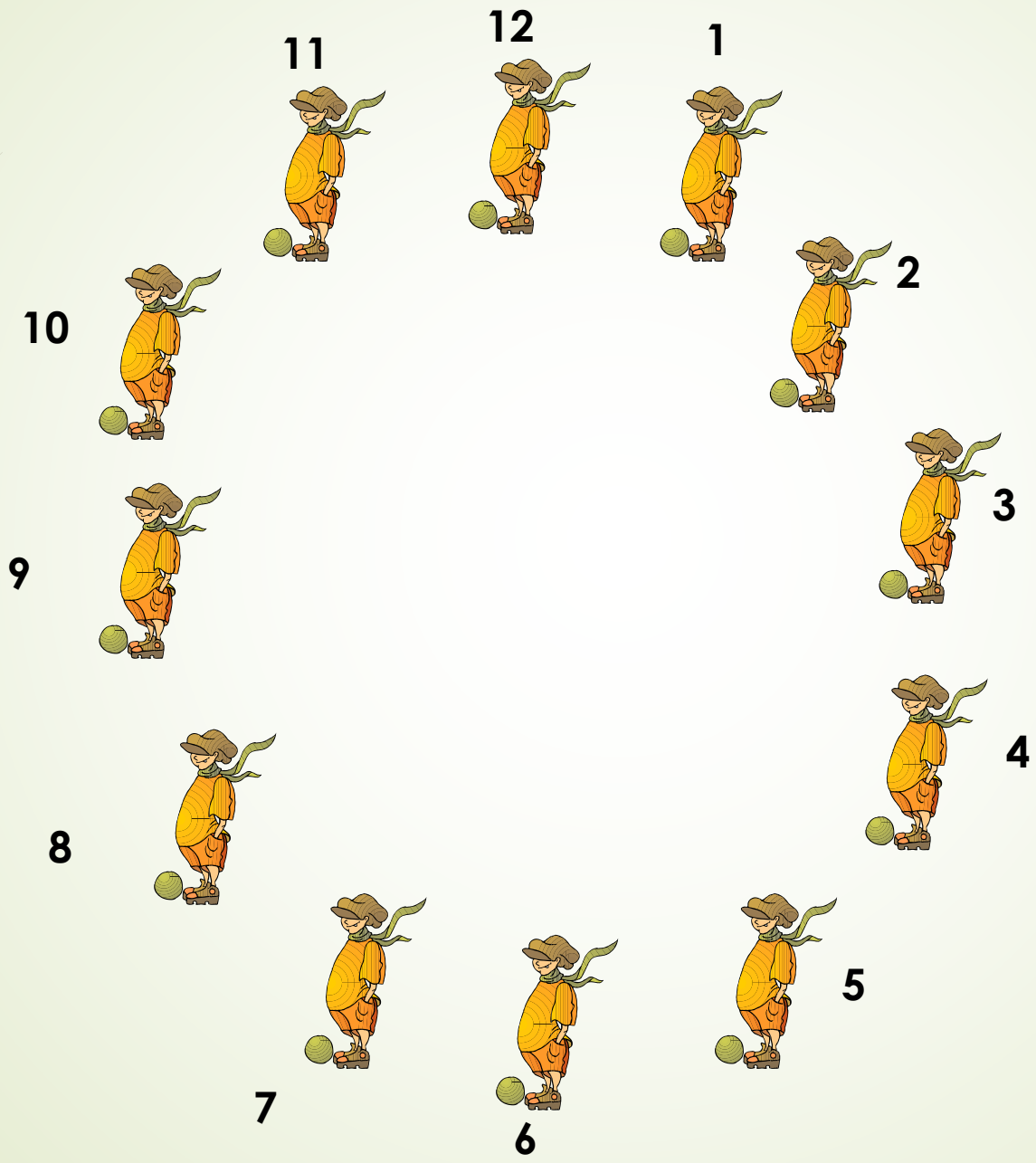
$$=2000+(3\times 4+2)\times 4=2056。$$

从中看到，对于C/C++的数组$a[m][n]$，求$a[i][j]$元素的地址为：

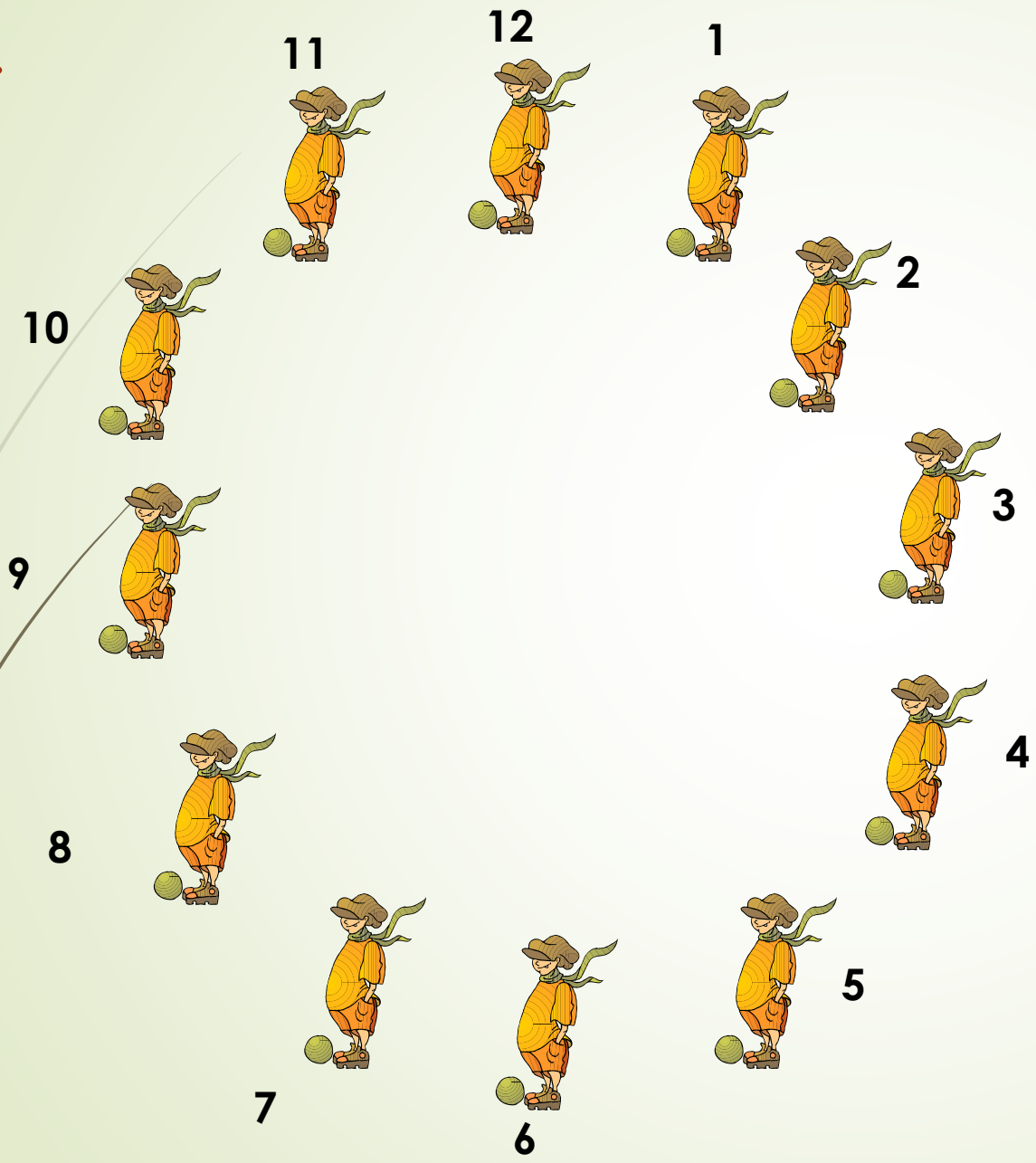
$$LOC(a_{i,j})=LOC(a_{0,0})+(i\times n+j)\times k$$

【例6.2：p135】求解约瑟夫问题：设有n个人站成一圈，其编号为1~n，从编号为1的人开始按顺时针方向“1，2，3，4，…”循环报数，数到m的人出列，然后从出列者的下一人开始重新报数，数到m的人又出列，如此重复进行，直到n个人都出列为止。要求输出这n个人的出列顺序。

M=4 N=12

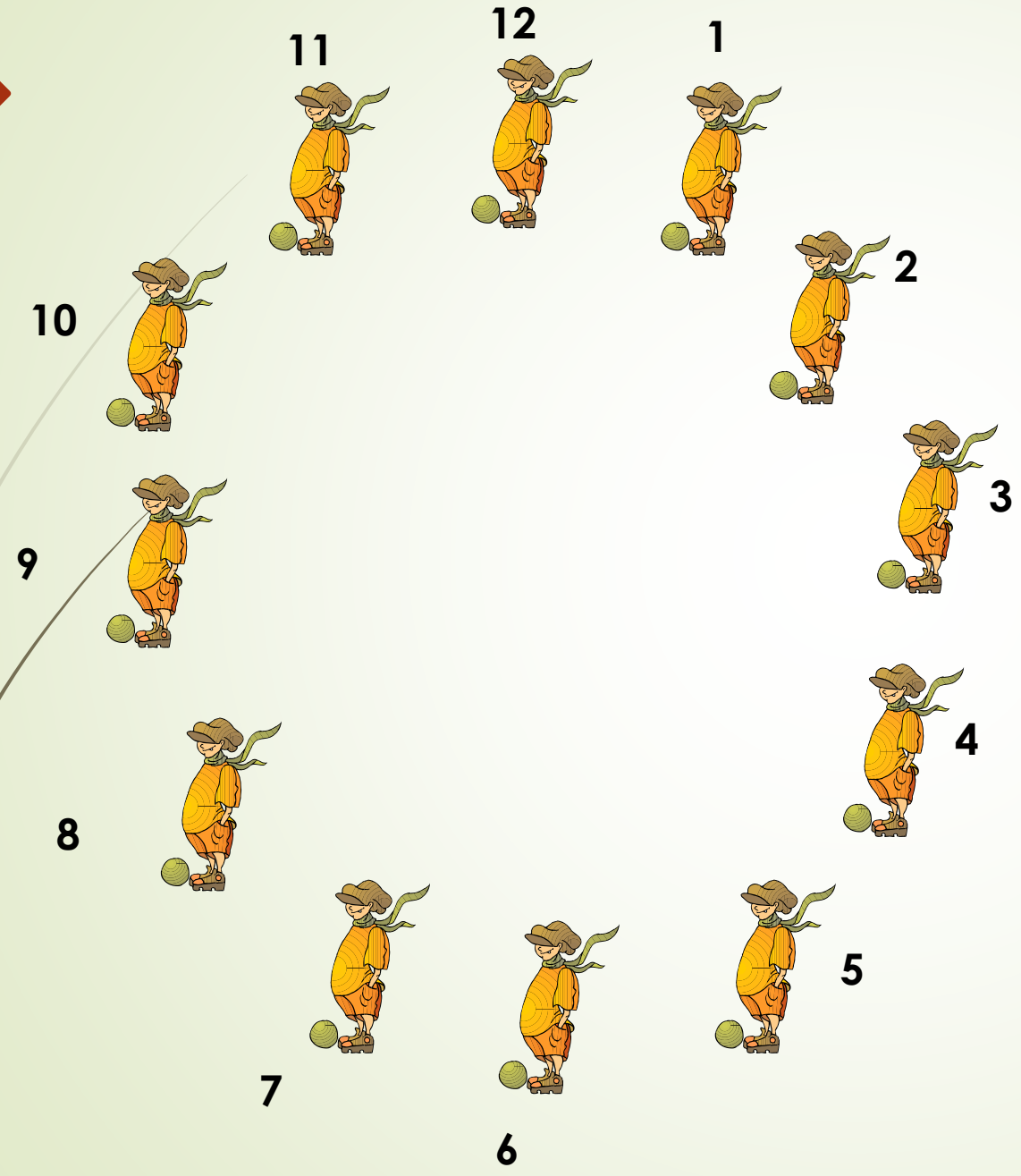


M=4 N=12

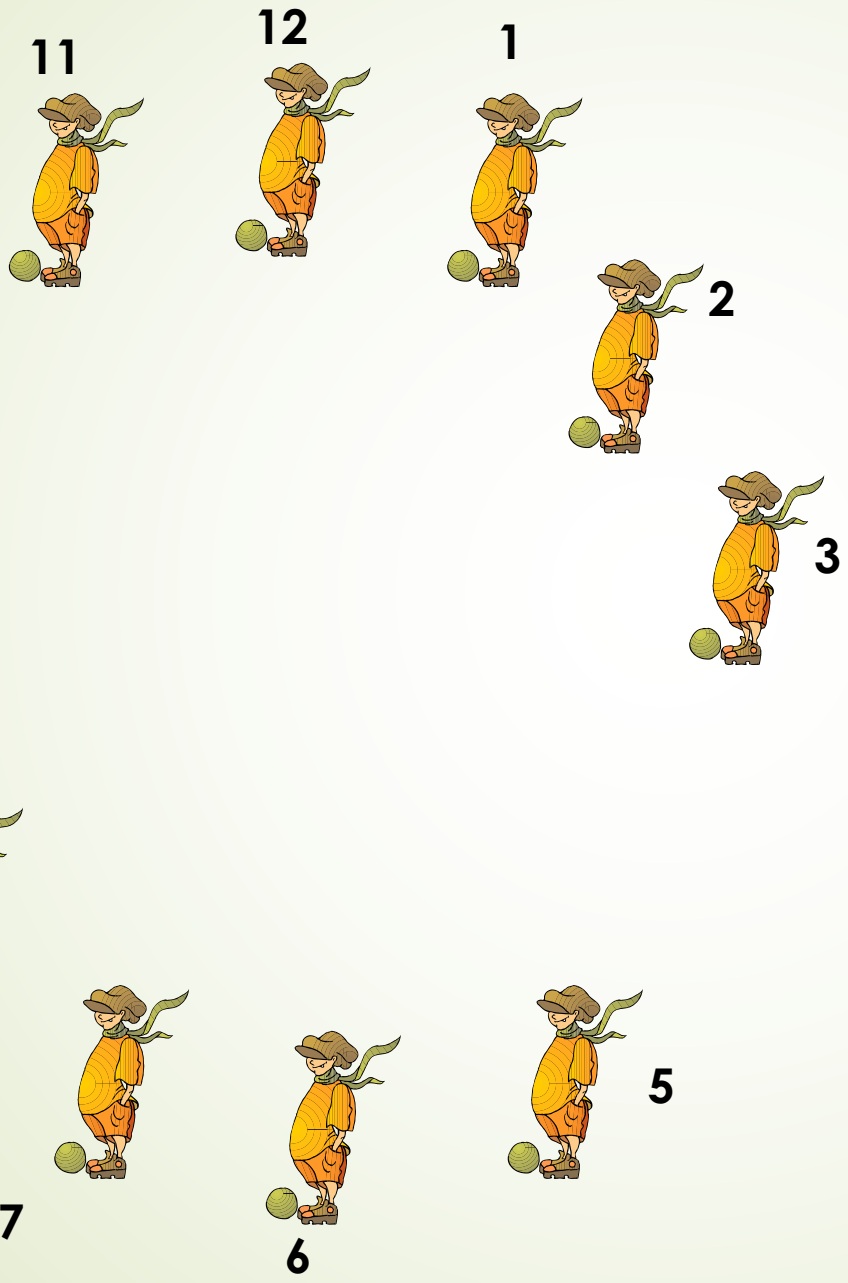


M=4　N=12

出列学生

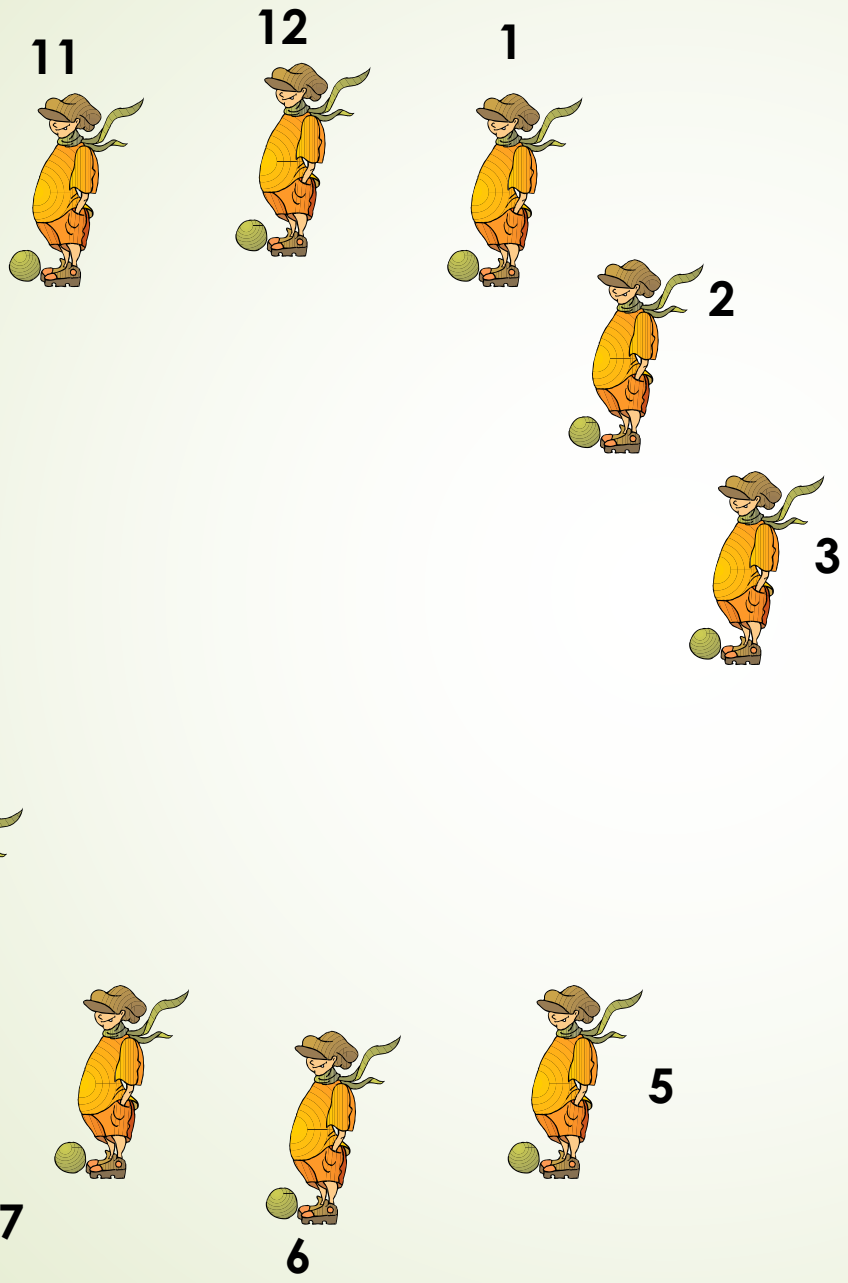


M=4 N=12



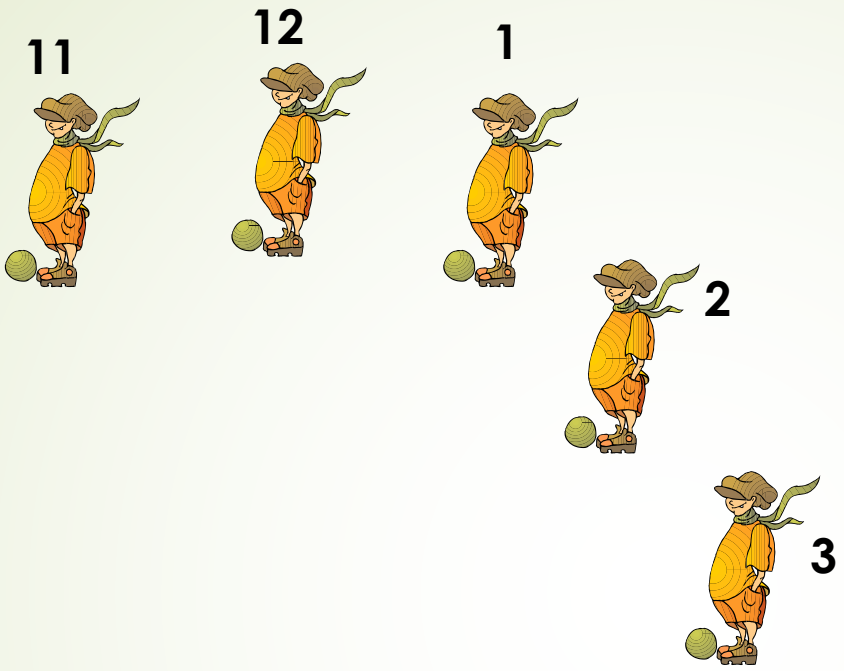
出列学生

M=4 N=12



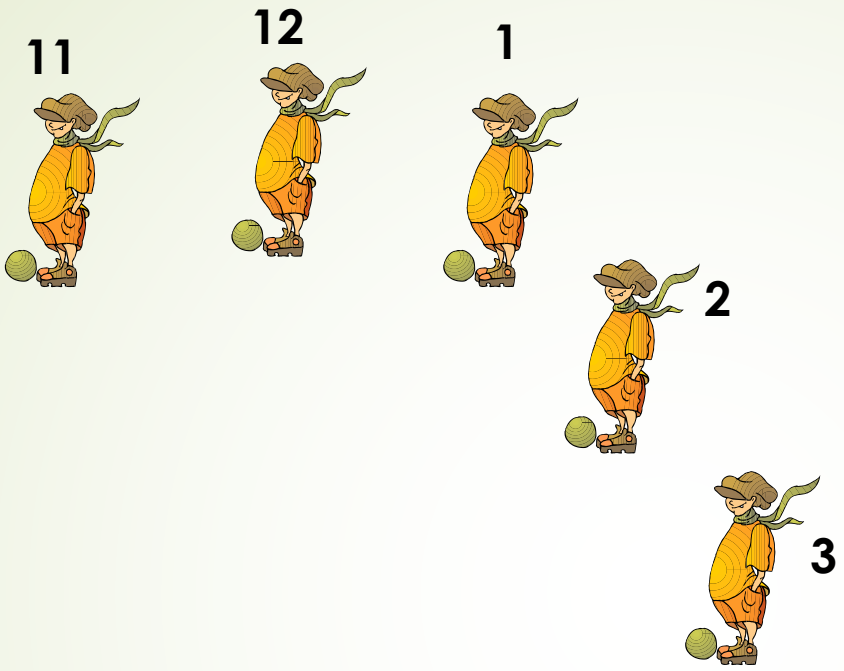
出列学生

M=4 N=12



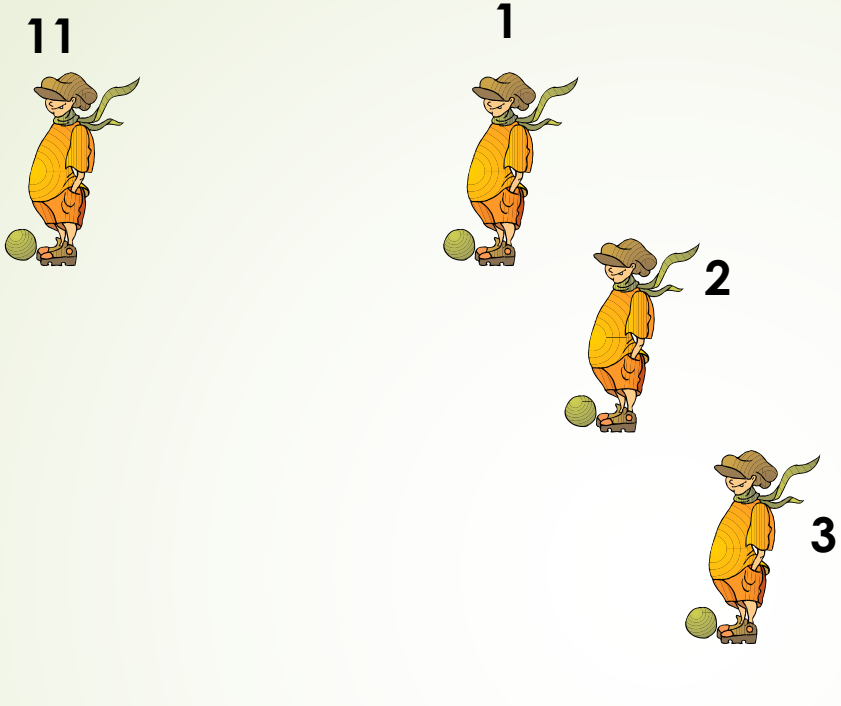
出列学生

M=4 N=12



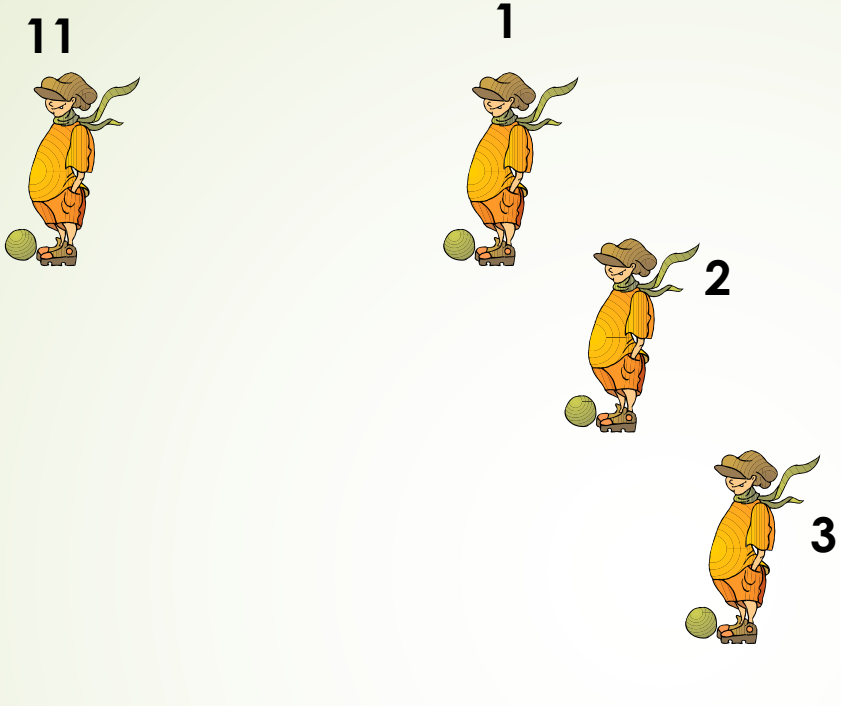
出列学生

M=4 N=12



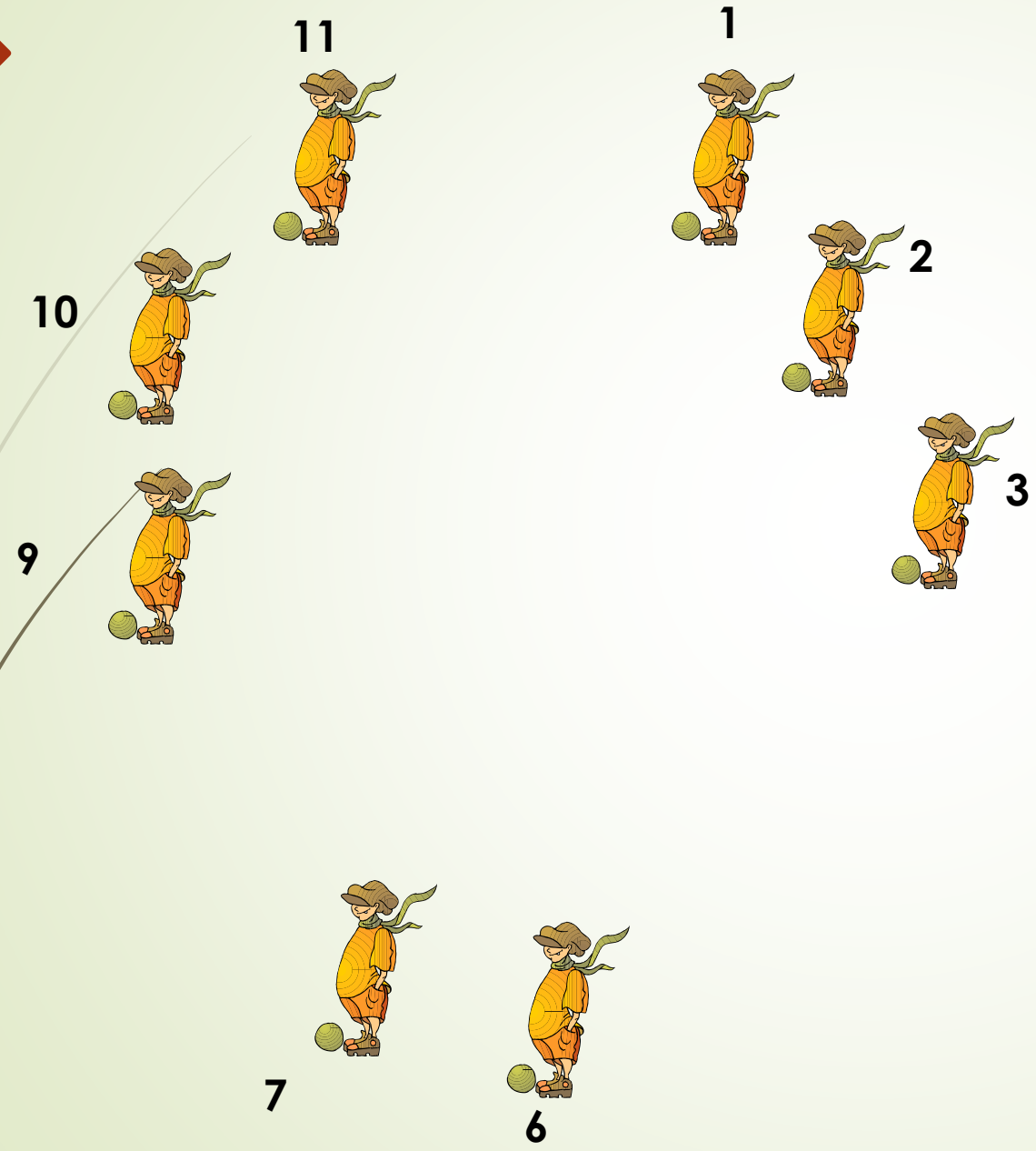
出列学生

M=4 N=12



出列学生

M=4 N=12



出列学生

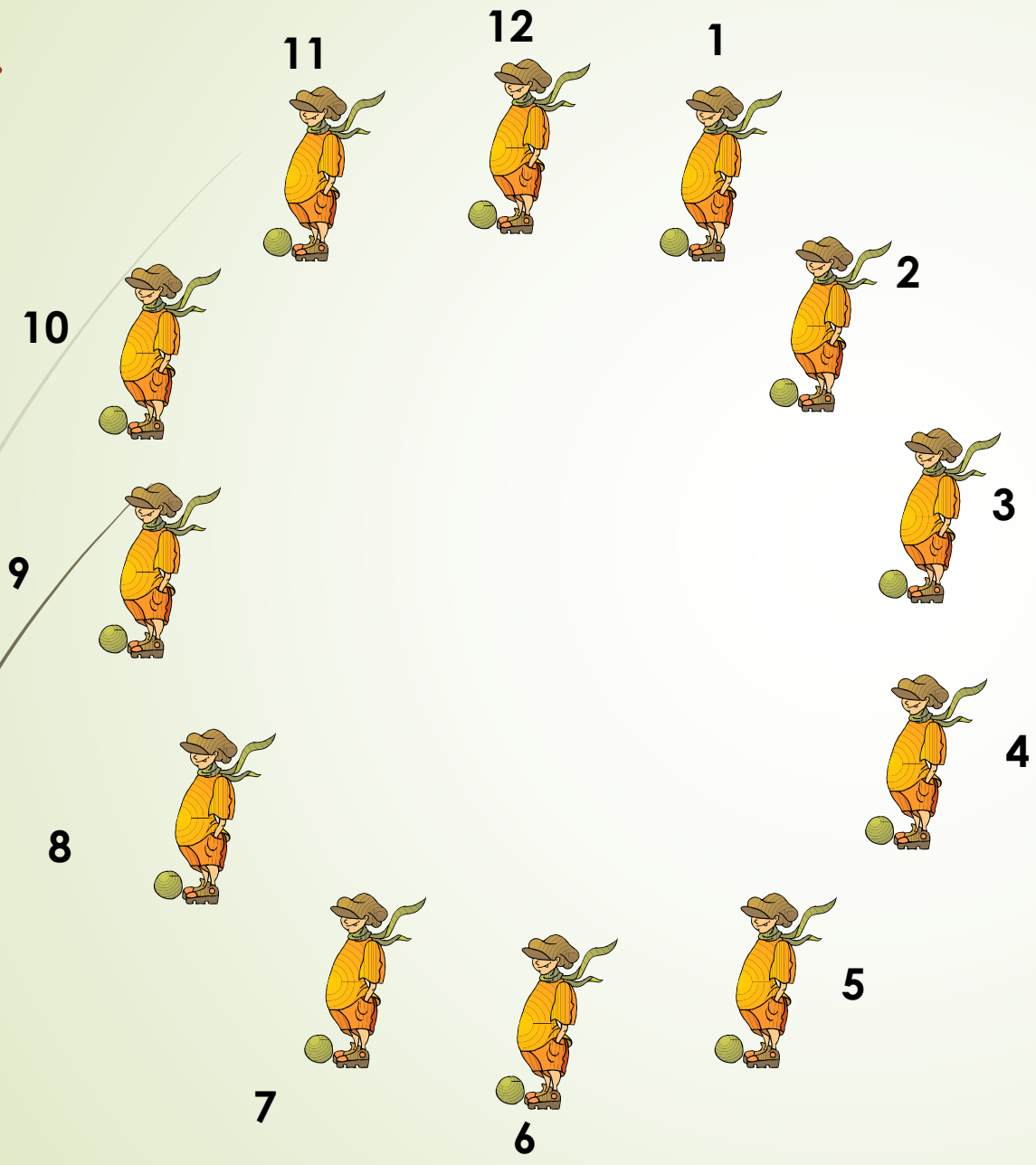
4

8

12

5

要解决的几个问题



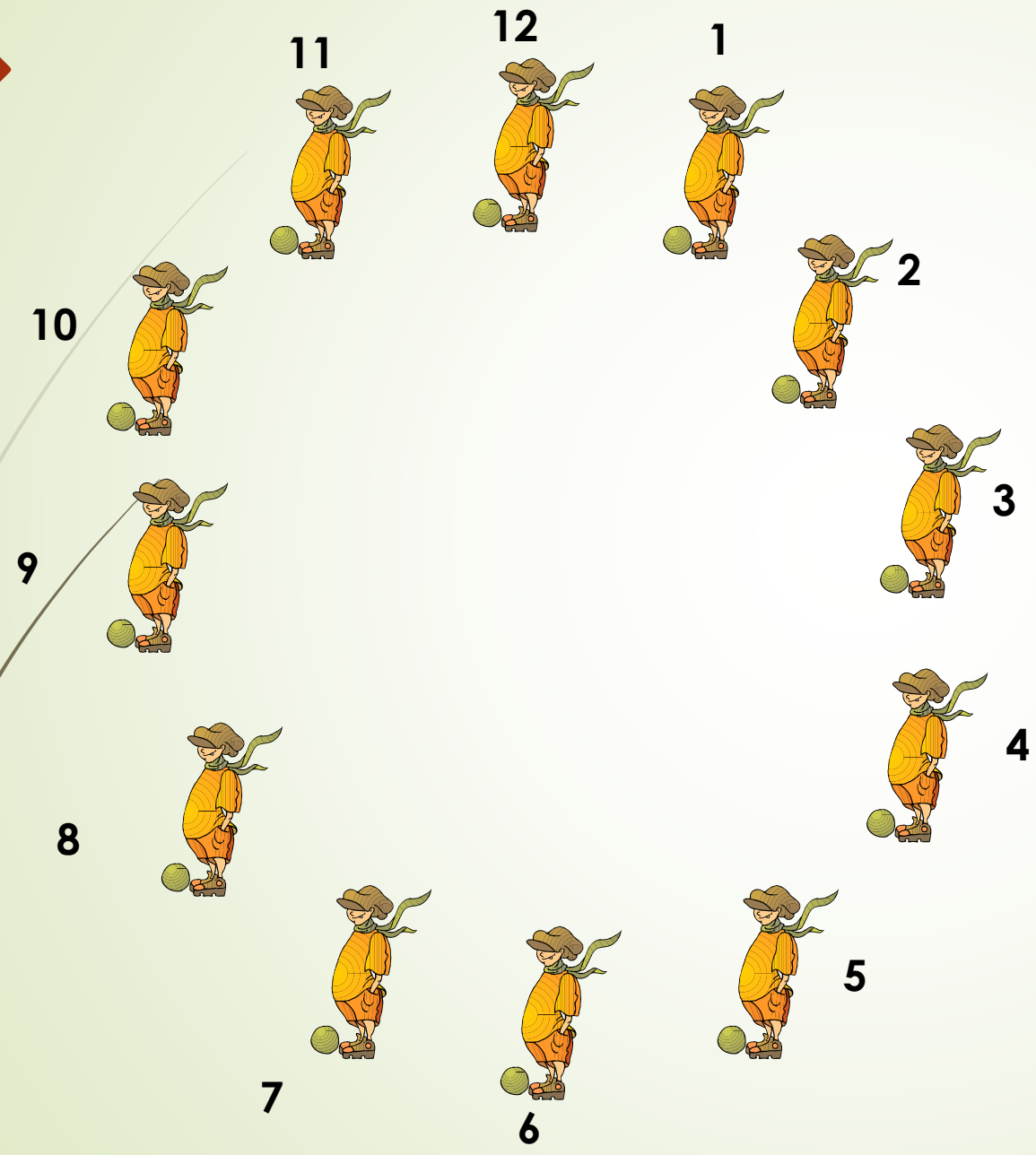
1. 人如何存放？

2. 如何报数？

3. 出列的人如何表示？

4. 何时报数停止？

要解决的几个问题



1. 人如何存放？

采用一维数组进行存储

初始时，

person[0]=1

person[1]=2

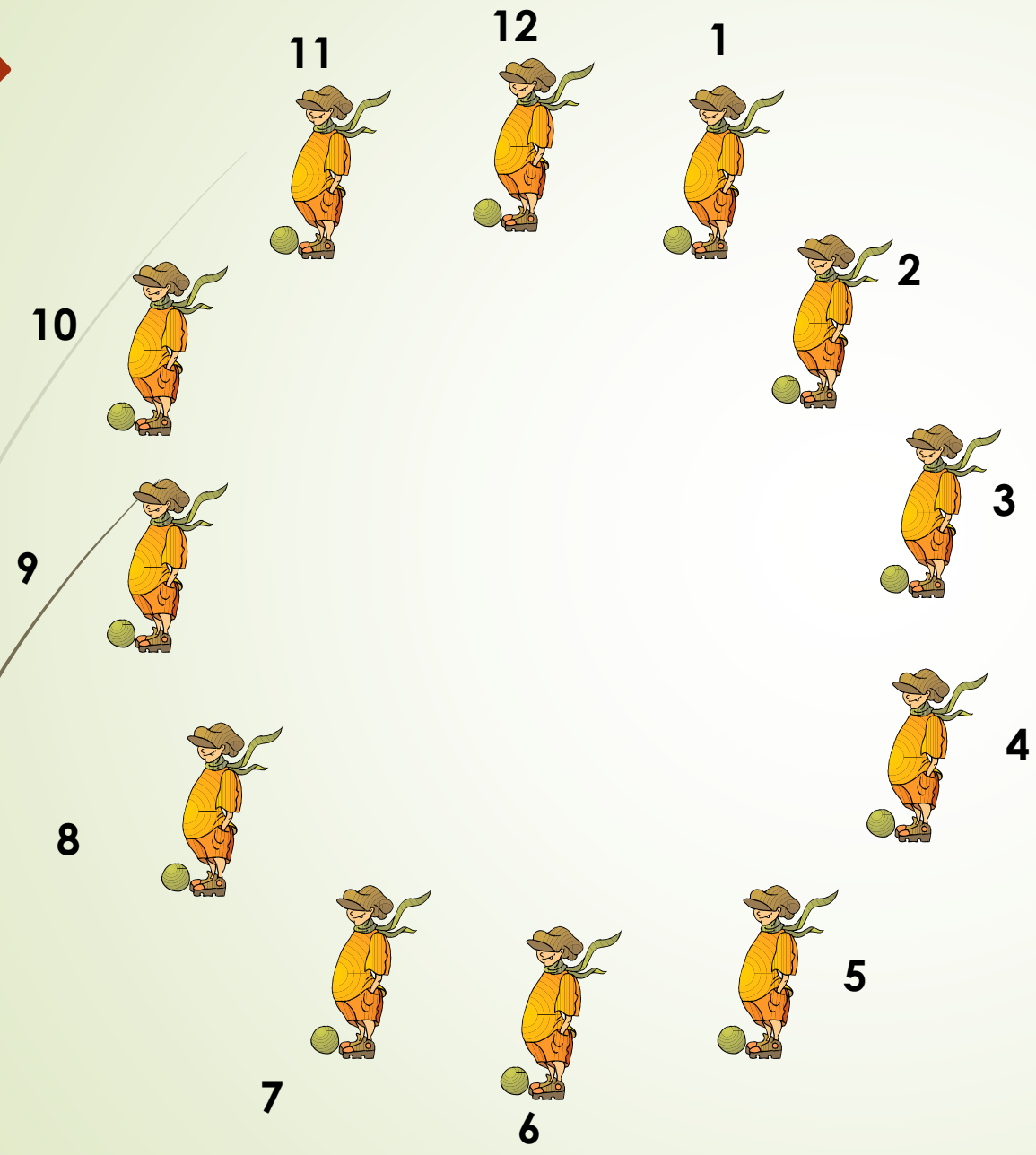
…

person[i]=i+1

…

Person[n-1]=n

要解决的几个问题



M=4 N=12

2 如何报数？

报到m的人，就是下标为t=（t+m-1）%i

t的初始值为0

i是数组中的人数

3 出列的人如何表示

报到m的人出列，就从数组中删除，后面的人依次往前移。这样数组的人就少一个。

4 报数何时停止

报数n次就可以停止了。

```
Void Josephus(int n,int m)
{
      int p[maxsize];
      int i,j,t;
      for(i=0;i<n;i++)
        p[i]=i+1;
      t=0;
      printf("出列顺序:");
      for(i=n;i>=1;i--)
      {
            t=(t+m-1)%i;
            printf("%d",p[t]);
            for(j=t+1;j<=i-1;j++)
              p[j-1]=p[j];
      }
      printf("\n");
}
```

6.1.3 特殊矩阵的压缩存储

1. 什么是压缩存储?

若多个数据元素的值都相同，则只分配一个元素值的存储空间，且零元素不占存储空间。

2. 什么样的矩阵能够压缩?

一些特殊矩阵，如：对称矩阵，对角矩阵，三角矩阵，稀疏矩阵等。

3. 什么叫稀疏矩阵?

矩阵中非零元素的个数较少（一般小于5%）

特殊矩阵的主要形式有：

（1）对称矩阵

（2）上三角矩阵 / 下三角矩阵

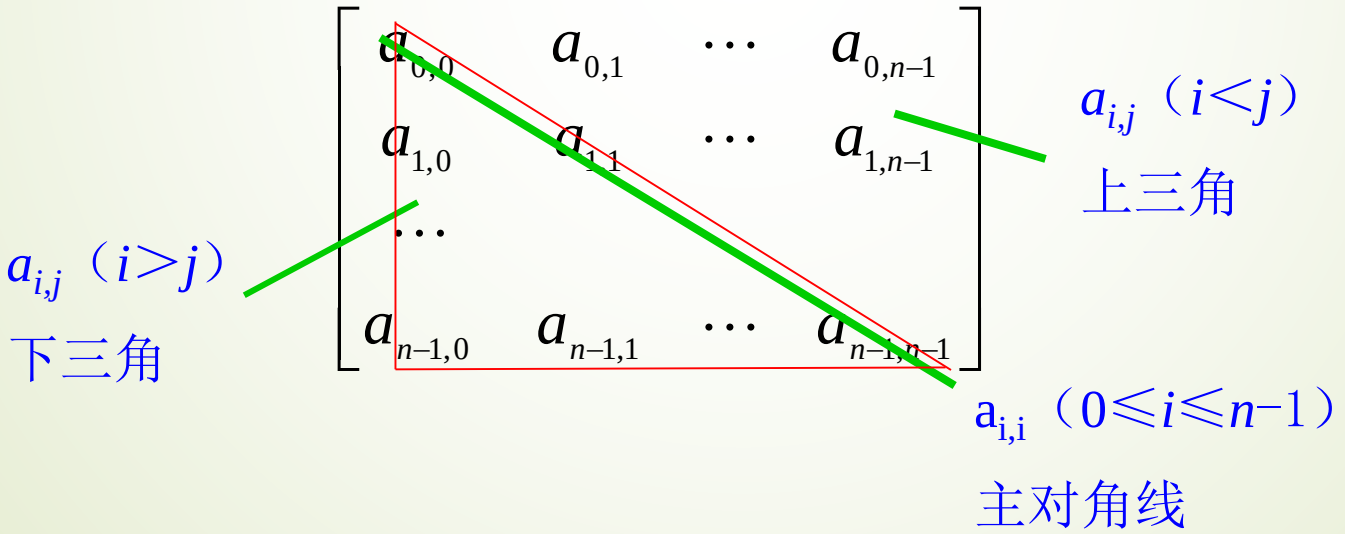
（3）对角矩阵

它们都是方阵，即行数和列数相同。

1. 对称矩阵的压缩存储

若一个n阶方阵$A[n][n]$中的元素满足$a_{i,j}=a_{j,i}$（$0\le i, j\le n-1$），则称其为n阶对称矩阵。

由于对称矩阵中的元素关于主对角线对称，因此在存储时可只存储对称矩阵中上三角或下三角中的元素，使得对称的元素共享一个存储空间。这样，就可以将n^2个元素压缩存储到$n(n+1)/2$个元素的空间中。以行序为主序存储其下三角＋对角线的元素。

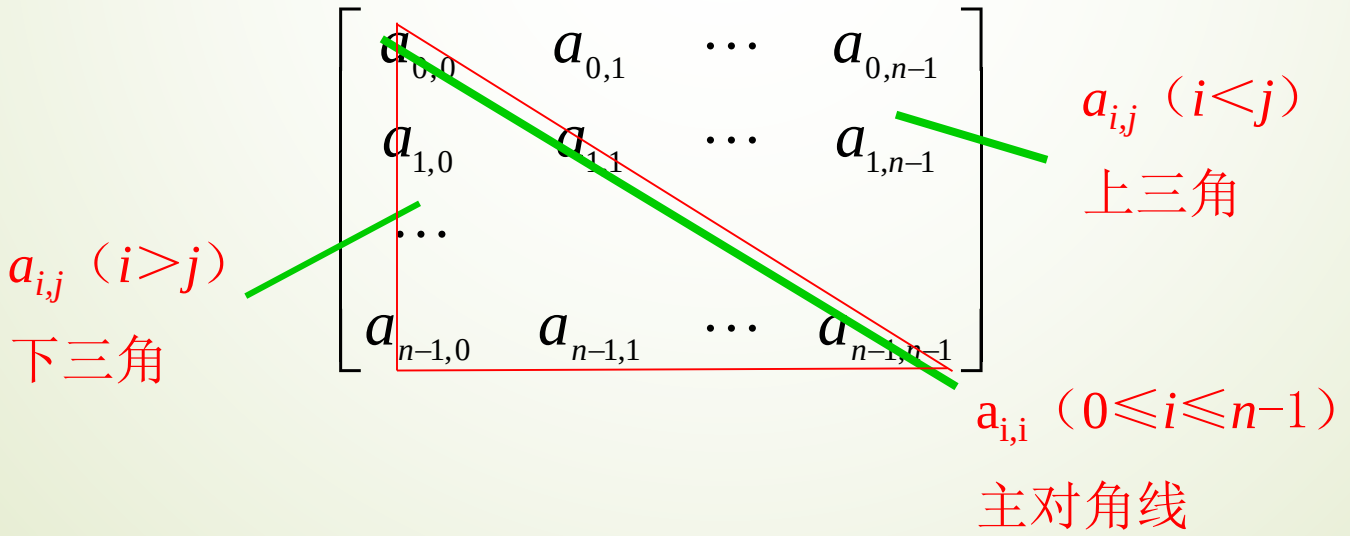


n^2个元素←→ $n(n+1)/2$个元素

A[0..$n-1$,0..$n-1$] ←→ B[0..$n(n+1)/2-1$]

$a[i][j]$ ←→ $b[k]$

$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & \text{当 } i\geq j\text{时}\\[2ex] \dfrac{j(j+1)}{2}+i & \text{当}i<j\text{时}\end{cases}$$



2. 三角矩阵的压缩存储

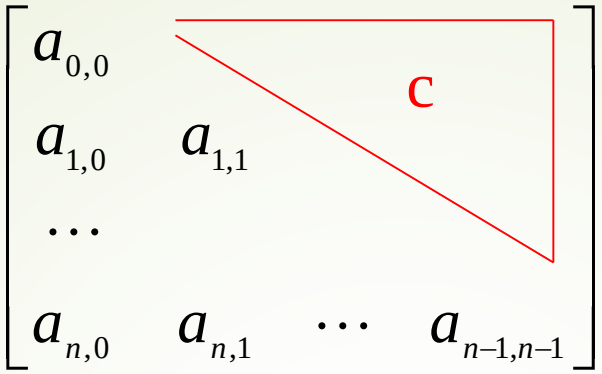
- 上三角矩阵：

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ & a_{1,1} & \cdots & a_{1,n-1} \\ & & c & \\ & & & a_{n-1,n-1} \end{bmatrix}$$

$$k=\begin{cases} \dfrac{i*(2n-i+1)}{2}+j-i & \text{当}i\le j\text{时} \\ \dfrac{n(n+1)}{2} & \text{当}i>j\text{时} \end{cases}$$

存放常量c

- 下三角矩阵



$$k=\begin{cases}\dfrac{i*(i+1)}{2}+j & \text{当}i\geq j\text{时}\\ \dfrac{n(n+1)}{2} & \text{当}i<j\text{时}\end{cases}$$

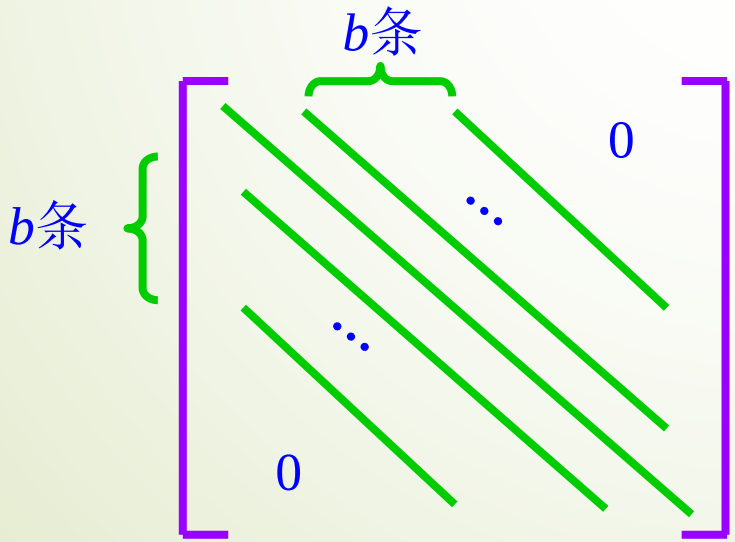
存放常量c

3. 对角矩阵的压缩存储

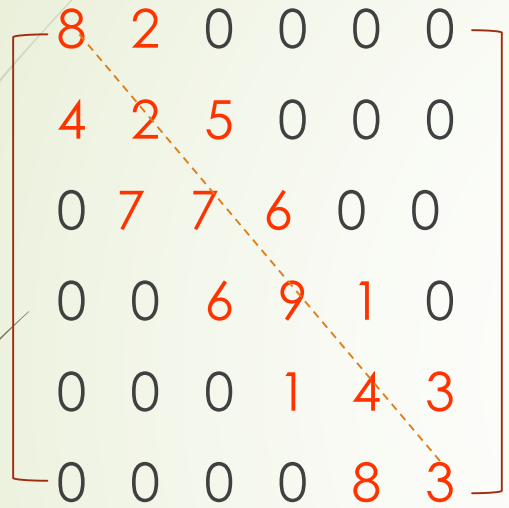
若一个n阶方阵A满足其所有非零元素都集中在以主对角线为中心的带状区域中，则称其为n阶对角矩阵（带状矩阵）。

其主对角线上下方各有b条次对角线，称b为矩阵半带宽，$(2b+1)$为矩阵的带宽。对于半带宽为b（$0 \leqslant b \leqslant (n-1)/2$）的对角矩阵，其$|i-j| \leqslant b$的元素$a_{i,j}$不为零，其余元素为零。

下图所示是半带宽为b的对角矩阵示意图。



半带宽为b的对角矩阵



b=1的三对角矩阵

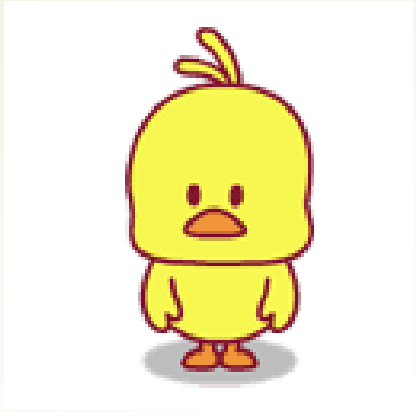
$$A \longleftrightarrow B$$

$$a[i][j] \longleftrightarrow b[k]$$

当$b=1$时称为三对角矩阵。

其压缩地址计算公式如下：

$$k=2i+j$$



思考题：

为什么采用压缩存储，需要解决什么问题？

6.2 稀疏矩阵

关联的知识点

- 稀疏矩阵的特点
- 稀疏矩阵的三元组和十字链表存储结构

稀疏矩阵的定义

一个阶数较大的矩阵中的非零元素个数s相对于矩阵元素的总个数t十分小时，即s<<t时,称该矩阵为稀疏矩阵。例如一个100×100的矩阵，若其中只有100个非零元素，就可称其为稀疏矩阵。

6.2.1 稀疏矩阵的三元组表示

稀疏矩阵的压缩存储方法是只存储非零元素。

由于稀疏矩阵中非零元素的分布没有任何规律，所以在存储非零元素时还必须同时存储该非零元素所对应的行下标和列下标。

稀疏矩阵中的每一个非零元素需由一个三元组：

$$(i, j, a_{i,j})$$

唯一确定，稀疏矩阵中的所有非零元素构成三元组线性表。

稀疏矩阵三元组表示的动画演示

一个6×7阶稀疏矩阵A的三元组线性表示

$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

一个稀疏矩阵A

(0,2,1)　(1,1,2)　(2,0,3)　(3,3,5)　(4,4,6)　(5,5,7)　(5,6,4)

((0,2,1)，(1,1,2)，(2,0,3)，(3,3,5)，(4,4,6)，(5,5,7)，(5,6,4))

若把稀疏矩阵的三元组线性表按顺序存储结构存储，则称为稀疏矩阵的三元组顺序表。则三元组顺序表的数据结构可定义如下：

```
#define MaxSize  100   //矩阵中非零元素最多个数
typedef struct
{   int r;             //行号
    int c;             //列号
    ElemType d;        //元素值
} TupNode;             //三元组定义
typedef struct
{   int rows;          //行数值
    int cols;          //列数值
    int nums;          //非零元素个数
    TupNode data[MaxSize];
} TSMatrix;            //三元组顺序表定义
```

通常约定：data域中表示的非零元素通常以行序为主序顺序排列，它是一种下标按行有序的存储结构。

这种有序存储结构可简化大多数矩阵运算算法。

下面的讨论假设data域按行有序存储。

（1）从一个二维矩阵创建其三元组表示

以行序方式扫描二维矩阵A，将其非零的元素插入到三元组t的后面。

$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

t:

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

```
void CreatMat(TSMatrix &t,ElemType A[M][N])
{ int i,j;    t.rows=M;t.cols=N;t.nums=0;
    for (i=0;i<M;i++)
    {  for (j=0;j<N;j++)
        if (A[i][j]!=0)          //只存储非零元素
        {  t.data[t.nums].r=i;
           t.data[t.nums].c=j;
           t.data[t.nums].d=A[i][j];
           t.nums++;
        }
     }
}
```

（2）三元组元素赋值：执行$A[i][j]=x$ 运算

分为两种情况：①将一个非0元素修改为另一个非0值，如A[5][6]=8。

$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}\Rightarrow A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&8\end{bmatrix}$$

修改元素

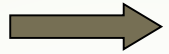
i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

⇒

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	**8**

②将一个0元素修改为非0值。如A[3][5]=8

$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$



$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&8&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

增加元素

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
3	5	8
4	4	6
5	5	7
5	6	8

算法如下：

```
bool Value(TSMatrix &t,ElemType x,int i,int j)
{   int k=0,k1;
    if (i>=t.rows || j>=t.cols)
      return false;          //失败时返回false
    while (k<t.nums && i>t.data[k].r) k++;//查找行
    while (k<t.nums && i==t.data[k].r
      && j>t.data[k].c) k++;      //查找列
```

```
    if (t.data[k].r==i && t.data[k].c==j) //存在这样的元素
       t.data[k].d=x;
    else       //不存在这样的元素时插入一个元素
    {    for (k1=t.nums-1;k1>=k;k1--)
         {  t.data[k1+1].r=t.data[k1].r;
            t.data[k1+1].c=t.data[k1].c;
            t.data[k1+1].d=t.data[k1].d;
         }
         t.data[k].r=i;t.data[k].c=j;t.data[k].d=x;
         t.nums++;
     }
     return true;          //成功时返回true
}
```

（3）将指定位置的元素值赋给变量　执行$x=A[i][j]$

先在三元组t中找到指定的位置，再将该处的元素值赋给x。

```
bool Assign(TSMatrix t,ElemType &x,int i,int j)
{   int k=0;
    if (i>=t.rows || j>=t.cols)
        return false;              //失败时返回false
    while (k<t.nums && i>t.data[k].r) k++; //查找行
    while (k<t.nums && i==t.data[k].r
          && j>t.data[k].c) k++; //查找列
    if (t.data[k].r==i && t.data[k].c==j)
       x=t.data[k].d;
    else
       x=0;          //在三元组中没有找到表示是零元素
    return true;       //成功时返回true
}
```

（4）输出三元组

从头到尾扫描三元组t，依次输出元素值。算法如下：

```
void DispMat(TSMatrix t)
{   int i;
    if (t.nums<=0) return;
    printf("\t%d\t%d\t%d\n",t.rows,t.cols,t.nums);
    printf("  -------------------\n");
    for (i=0;i<t.nums;i++)
        printf("\t%d\t%d\t%d\n",
            t.data[i].r,t.data[i].c, t.data[i].d);
}
```

（5）矩阵转置

对于一个$m\times n$的矩阵$A_{m\times n}$，其转置矩阵是一个$n\times m$的矩阵$B_{n\times m}$,满足$b_{i,j}=a_{j,i}$，其中$0\le i\le m-1$，$0\le j\le n-1$。

$$A_{6\times 7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

$$B_{7\times 6}=\begin{bmatrix}0&0&3&0&0&0\\0&2&0&0&0&0\\1&0&0&0&0&0\\0&0&0&5&0&0\\0&0&0&0&6&0\\0&0&0&0&0&7\\0&0&0&0&0&4\end{bmatrix}$$

i	j	a_{ij}
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

i	j	a_{ij}
0	2	3
1	1	2
2	0	1
3	3	5
4	4	6
5	5	7
6	5	4

```
void TranTat(TSMatrix t,TSMatrix &tb)
{  int p,q=0,v;              //q为tb.data的下标
   tb.rows=t.cols;tb.cols=t.rows;tb.nums=t.nums;
   if (t.nums!=0)            //当存在非零元素时执行转置
   {  for (v=0;v<t.cols;v++) //tb.data[q]中记录以列序排列
       for (p=0;p<t.nums;p++) //p为t.data的下标
        if (t.data[p].c==v)
        {   tb.data[q].r=t.data[p].c;
            tb.data[q].c=t.data[p].r;
            tb.data[q].d=t.data[p].d;
            q++;
        }
    }
}
```

以上算法的时间复杂度为O(t.cols×t.nums)，而将二维数组存储在一个m行n列矩阵中时，其转置算法的时间复杂度为O($m\times n$)。

最坏情况是当稀疏矩阵中的非零元素个数t.nums和$m\times n$同数量级时，上述转置算法的时间复杂度就为O($m\times n^2$)。

【P140例6.3】采用三元组存储稀疏矩阵，设计两个稀疏矩阵相加的运算算法。

实现相加运算的两个稀疏矩阵a和b的行数、列数都必须相同。用i和j两个变量扫描三元组a和b，按行序优先方式进行处理，并将结果存放在c中。当a的当前元素（简称为a元素）和b的当前元素（简称为b元素）的行号、列号均相等时，将它们的值相加，只有在相加值不为0时，才在c中添加一个新的元素表示想加后的结果。

$$A_{6\times7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

$$B_{6\times7}=\begin{bmatrix}0&0&2&0&0&0&0\\0&0&0&0&3&0&0\\8&0&0&0&0&0&0\\0&0&0&9&0&0&0\\0&6&0&0&0&0&0\\0&0&0&0&0&2&4\end{bmatrix}$$

```
bool MatAdd(TSMatrix a,TSMatrix b,TSMatrix &c)
{  int i=0,j=0,k=0;
   ElemType v;
   if(a.rows!=b.rows||a.cols=b.cols)
       return false; //行数或列数不等时不能进行相加运算
   c.rows=a.rows;c.cols=a.cols;//c的行列数与a，b相同
   while(i<a.nums && j<b.nums)//处理a和b中的每个元素
    {  if (a.data[i].r==b.data[j].r)  //行号相等时
       {  if (a.data[i].c<b.data[j].c) //a元素的列号小于b元素的列号
          {   c.data[k].r=a.data[i].r;//将a元素添加到c中
              c.data[k].c=a.data[i].c;
              c.data[k].d=a.data[i].d;
              k++;i++;
          }
          else if(a.data[i].c>b.data[j].c)//a元素的列号大于b元素的列号
          {   c.data[k].r=b.data[j].r; //将b元素添加到c中
              c.data[k].c=b.data[j].c;
              c.data[k].d=b.data[j].d;
              k++;j++;
          }
```

```
            else //a元素的列号等于b元素的列号
            {   v= a.data[i].r+b.data[j].r;
                if(v!=0) //只将不为0的结果添加到c中
                {  c.data[k].r=a.data[i].r;
                   c.data[k].c=a.data[i].c;
                   c.data[k].d=v;
                   k++;
                 }
                 i++;j++;
             }
        }// end of if (a.data[i].r==b.data[j].r)
        else if (a.data[i].r<b.data[j].r)//a元素的行号小于b元素的行号
        {  c.data[k].r=a.data[i].r; //将a元素添加到c中
           c.data[k].c=a.data[i].c;
           c.data[k].d=a.data[i].d;
           k++;i++;
        }
        else //a元素的行号大于b元素的行号
        {  c.data[k].r=b.data[j].r; ;//将b元素添加到c中
           c.data[k].c=b.data[j].c;
           c.data[k].d=b.data[j].d;
           k++;j++;
        }
    }//end of while
```

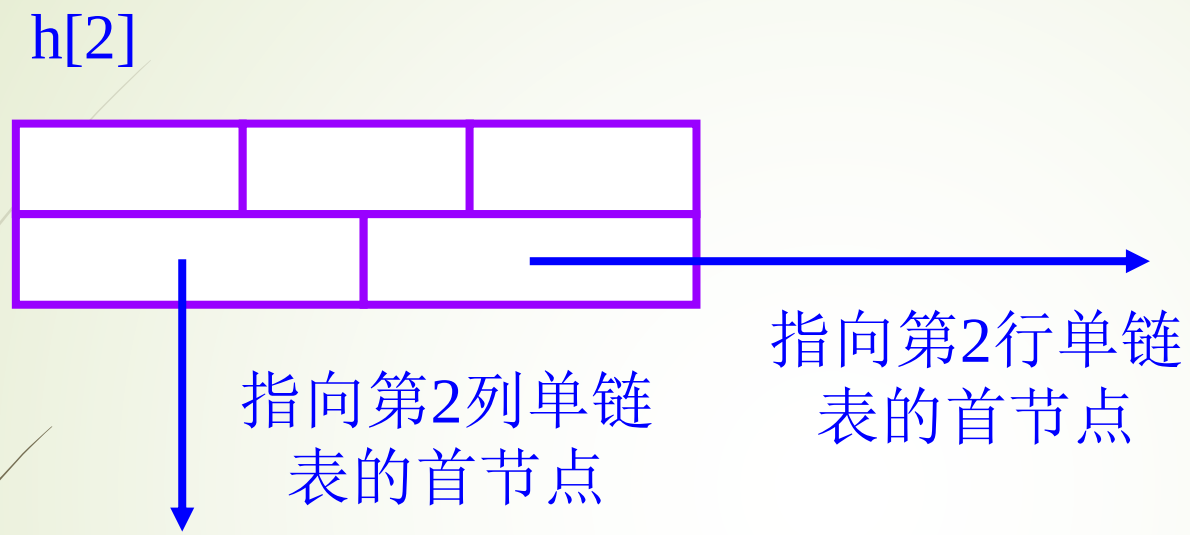
```
    if (i<a.nums&&j=b.nums) //b里面已经没有非0元素了
       while(i<a.nums)
       {  c.data[k].r=a.data[i].r; //将剩下的a元素全部添加到c中
          c.data[k].c=a.data[i].c;
          c.data[k].d=a.data[i].d;
          k++;i++;
        }
     if(i=a.nums&&j<b.nums) //a里面已经没有非0元素了
       while(j<b.nums)
       {  c.data[k].r=b.data[j].r;//将剩下的b元素全部添加到c中
          c.data[k].c=b.data[j].c;
          c.data[k].d=b.data[j].d;
          k++;j++;
       }
      c.nums=k;
  return true;
}
```

6.2.2 稀疏矩阵的十字链表表示

一个m行n列的十字链表的存储结构如下：

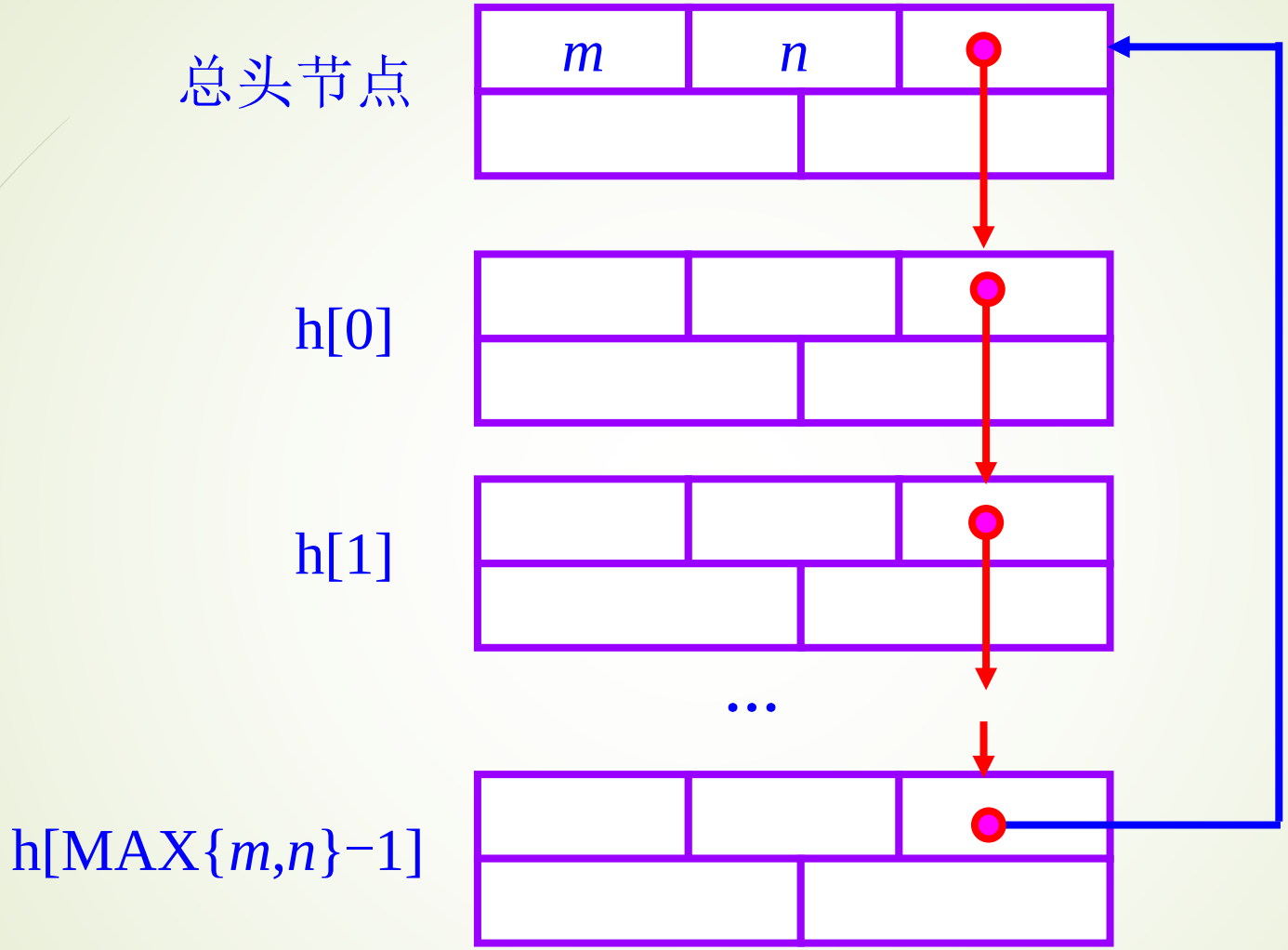
- 每个非零元素对应一个节点。
- 每行的所有节点链起来构成一个带行头节点的循环单链表。以h[i]（$0 \le i \le m-1$）作为第i行的头节点。
- 每列的所有节点链起来构成一个带列头节点的循环单链表。以h[j]（$0 \le j \le n-1$）作为第j列的头节点。这样每一个非零元素同时包含在所在行的行链表中和所在列的列链表中。方便算法中行方向和列方向的搜索，因而大大降低了算法的时间复杂度。

行列头节点可以共享，如：



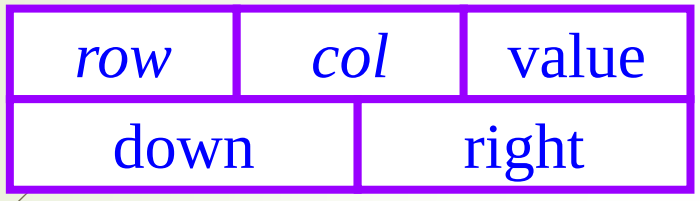
这样行列头节点的个数=MAX{m,n}

所有行列头结点又链起来构成一个带总头节点的循环单链表：

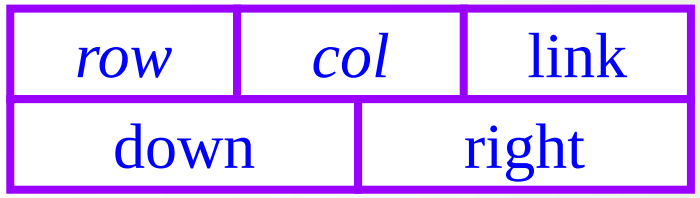


因此，总的头节点个数=MAX{m,n}+1

为了统一，设计节点类型如下：



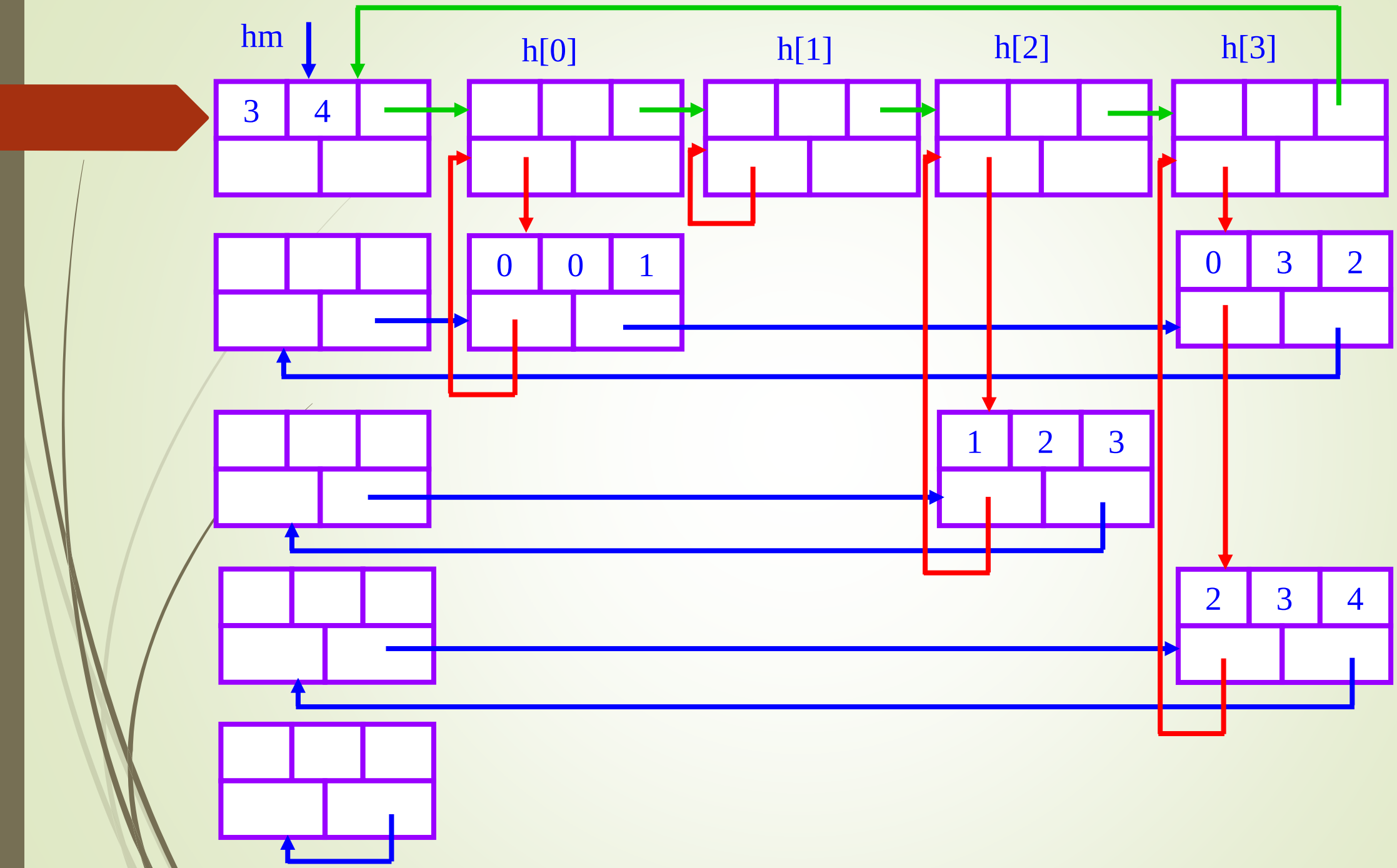
（a）节点结构



（b）头节点结构

$$B_{3\times4}=\begin{bmatrix}1 & 0 & 0 & 2\\ 0 & 0 & 3 & 0\\ 0 & 0 & 0 & 4\end{bmatrix}$$

一个稀疏矩阵


hm
h[0]
h[1]
h[2]
h[3]
3
4
0
0
1
0
3
2
1
2
3
2
3
4

十字链表节点结构和头节点的数据结构可定义如下：

```
#define M 3                              //矩阵行
#define N 4                              //矩阵列
#define Max ((M)>(N)?(M):(N))  //矩阵行列较大者
typedef struct mtxn
{   int row;                    //行号
    int col;                  //列号
    struct mtxn *right,*down;   //向右和向下的指针
    union
    {   int value;
        struct mtxn *link;
    } tag;
 } MatNode;                     //十字链表类型声明
```

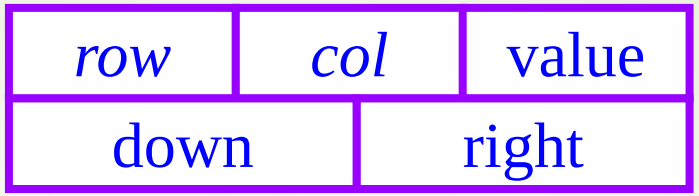
（1）对一个二维矩阵创建其十字链表表示

先建立十字链表头节点的循环链表，然后以行序扫描二维矩阵A，将非零的元素插入到十字链表中，插入操作如下：

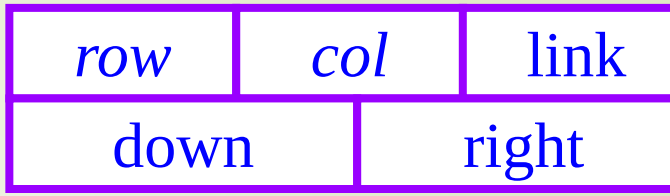
1.创建一个节点*p；

2.根据行号找到在行表中的插入位置并在行表中插入该节点；

3.根据列号找到在列表中的插入位置并在列表中插入该节点。



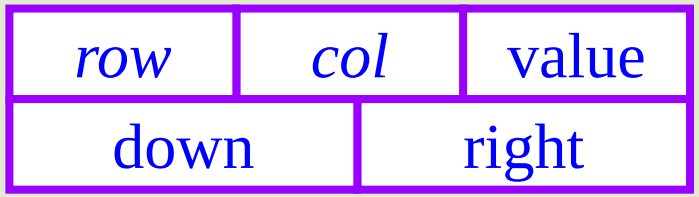
(a) 节点结构



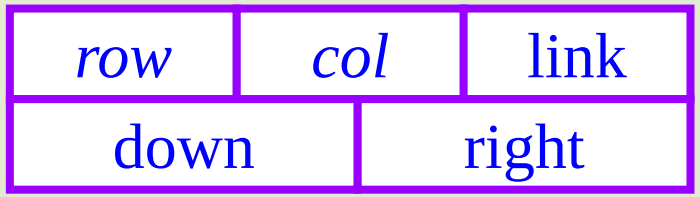
(b) 头节点结构

```
Void CreateMat(MatNode *&mh,ElemType a[M][N])
{  int i;
   MatNode *h[Max],*p,*q,*r;
   mh=(MatNode*)malloc(sizeof(MatNode));//创建十字链表的头节点
   mh->row=M;mh->col=N;
   r=mh;     //r指向尾节点
   for(i=0;i<Max;i++) //采用尾插法创建头节点h[0],h[1], …循环链表
     {  h[i]=(MaxNode*)malloc(sizeof(MatNode));
        h[i]->down=h[i]->right=h[i]; //将down和right方向置为循环的
        r->tag.link=h[i];//将h[i]加入链表中
        r=h[i];
     }
    r->tag.link=mh;      //置为循环链表
```

```
for(i=0;i<M;i++)   //处理每一行
 {   for (j=0;j<n;j++)   //处理每一列
      {    if (a[i][j]!=0)//处理非零元素
           { p=(MaxNode *)malloc(sizeof(MaxNode));//创建一个新节点
             p->row=i;p->col=j;p->tag.value=a[i][j];
             q=h[i];//查找在行表中的插入位置
             while(q->right! =h[i]&&q->right->col<j)
                q=q->right;
             p->right=q->right;q->right=p;//完成行表的插入
             q=h[j];//查找在列表中的插入位置
             while(q->down!=h[j]&&q->sown->row<i)
                q=q->down;
             p->down=q->down;q->down=p;//完成列表的插入
           }
      }
 }
}
```

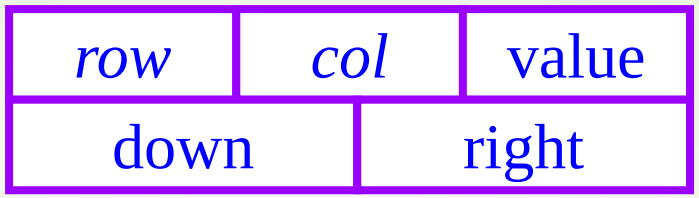


(a) 节点结构

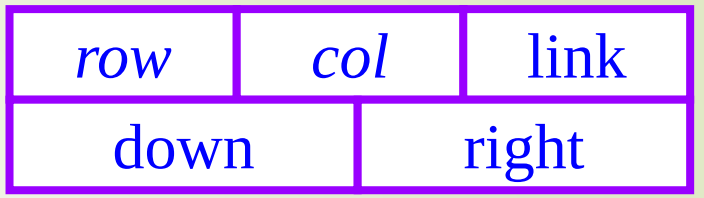


(b) 头节点结构

（2）输出十字链表矩阵



（a）节点结构



（b）头节点结构

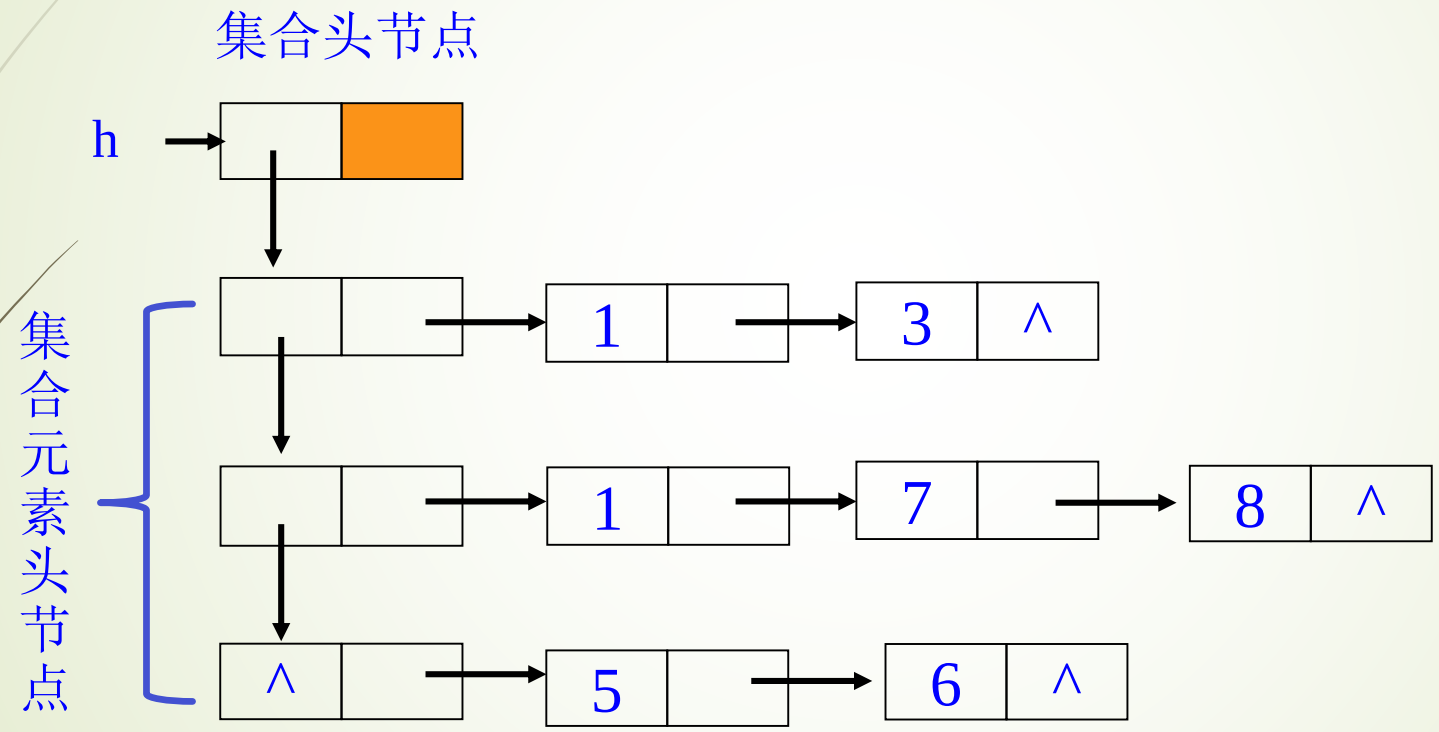
以行序方式从头到尾扫描十字链表h，依次输出元素值。

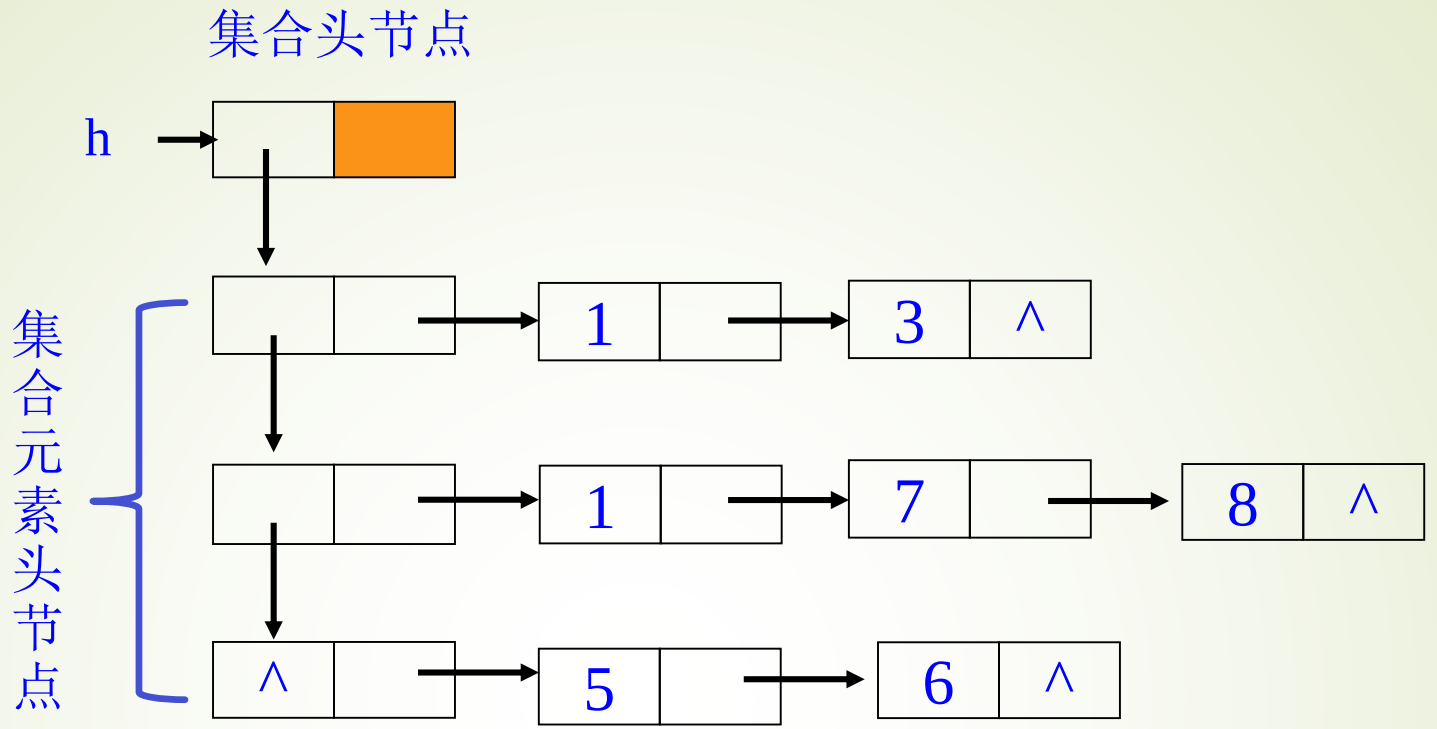
```
Void DispMat(MatNode *mh)
 {  MatNode *p,*q;
    printf(“行=%d列=%d\n”,mh->row;mh->col);
    P=mh->tag.link; //p指向第一个行（列）头节点
    while(p!=mh)
    {   q=p->right;//q指向当前行的第一个数据节点
        while(p!=q)
        {  printf(“%d\t%d\t%d\n”,q->row,q->col,q->tag.value);
           q=q->right;
        }
        p=p->tag.link; //转到下一行（列）的头节点
     }
}
```

【例6.4：p144】设计一个用于存储双层集合的存储结构，所谓双层集合是指这样的集合，其中的每个元素也是一个集合（称为集合元素），且由普通元素构成，例如，s={{1，3}，{1，7，8}，{5，6}}。

解：采用类似于十字链表的思路，将每个集合元素设计成带头节点的单链表，再将这些集合元素头节点串起来构成一个单链表，设置*h作为集合头节点。

s={{1，3}，{1，7，8}，{5，6}}。





数据节点的类型定义如下：

```
typedef struct dnode
{    Elemtype data;
     struct dnode  *next;
} DType
```

集合元素头节点的类型定义如下：

```
typedef struct dnode
{   Dtype *next;
     struct hnode  *link;
} DType
```

实验六 稀疏矩阵的加法和乘法问题

- 【基本要求】
  - 两个稀疏矩阵的三元组存储结构生成；
  - 实现稀疏矩阵的转置；
  - 实现两个稀疏矩阵的加运算；
  - 实现两个稀疏矩阵的乘法运算。

6.3 广义表

关联的知识点

- 广义表的定义及其特点
- 广义表的链式存储结构及其递归特性
- 广义表的基本运算算法设计
- 广义表的递归算法设计方法

6.3.1 广义表的定义

广义表简称表，它是线性表的推广。一个广义表是n（$n\geq 0$）个元素的一个序列，若$n=0$时则称为空表。设a_i为广义表的第i个元素,则广义表GL的一般表示与线性表相同：

$$GL=(a_1,a_2,\ldots,a_i,\ldots,a_n)$$

其中n表示广义表的长度，即广义表中所含元素的个数，$n\geq 0$。如果a_i是单个数据元素，则a_i是广义表GL的原子；如果a_i是一个广义表，则a_i是广义表GL的子表。

广义表具有如下重要的特性：

（1）广义表中的数据元素有相对次序；

（2）广义表的长度定义为最外层包含元素个数；

（3）广义表的深度定义为所含括弧的重数。其中原子的深度为0，空表的深度为1；

（4）广义表可以共享；一个广义表可以为其他广义表共享；这种共享广义表称为再入表；

（5）广义表可以是一个递归的表。一个广义表可以是自己的子表。这种广义表称为递归表。递归表的深度是无穷值，长度是有限值；

（6）任何一个非空广义表GL均可分解为表头head(GL) = a_1和表尾tail(GL) = (a_2,…,a_n) 两部分。

广义表	表头	表尾
(#)	无表头	无表尾
((#))	(#)	(#)
(*a*)	*a*	(#)
((*a*))	(*a*)	(#)
(*a*,(*b*),((*c*)))	*a*	((*b*),((*c*)))

显然，一个广义表的表尾始终是一个广义表，空表无头无尾。是递归的数据结构。

为了简单起见，下面讨论的广义表不包括前面定义的再入表和递归表，即只讨论一般的广义表。另外，我们规定用小写字母表示原子，用大写字母表示广义表的表名。例如：

A=(#)　　空表，其长度为0

B=(e)　　只含有单个原子e，长度为1

C=(a,(b,c,d))　有两个元素，一个是原子，另一个是子表，C的长度为2

D=(A,B,C)=((),(e),(a,(b,c,d)))　有三个元素，每个元素又是一个表，D的长度为3

E=((a,(a,b),((a,b),c)))　只含有一个元素，该元素是一个表，E的长度为1

如果把每个表的名字（若有的话）写在其表的前面，则上面的5个广义表可相应地表示如下：

A(#)

B(e)

C(a,•(b,c,d))

D(A(),B(e),C(a, •(b,c,d)))

E(•(a, •(a,b), •(•(a,b),c)))

若用圆圈和方框分别表示表和单元素，并用线段把表和它的元素（元素节点应在其表节点的下方）连接起来,则可得到一个广义表的图形表示。

广义表的图形表示动画演示

A(#)

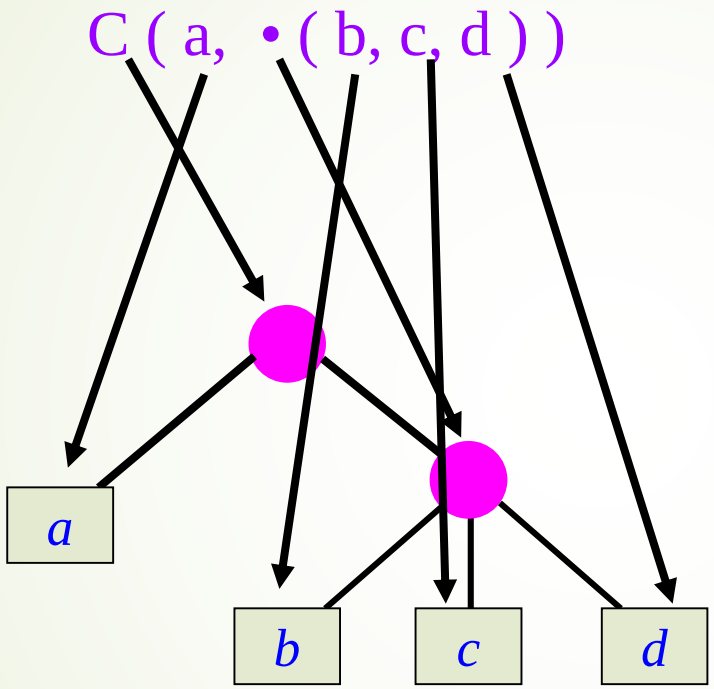
生成广义表图形表示

广义表的图形表示动画演示

B(e)

生成广义表图形表示

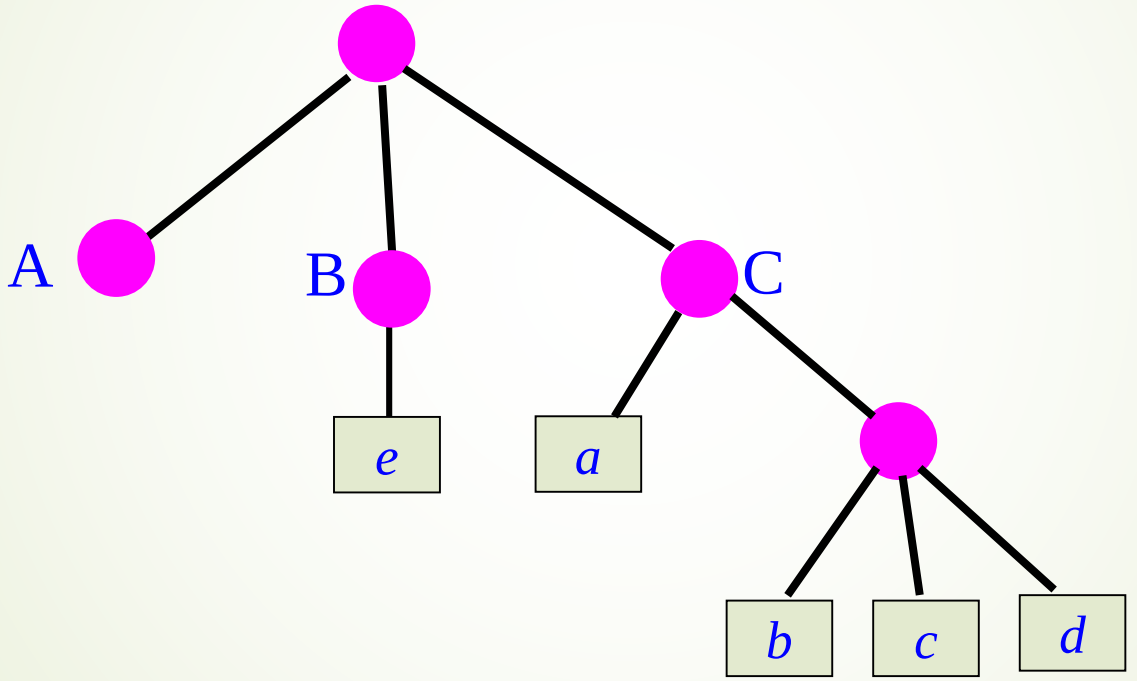
广义表的图形表示动画演示



生成广义表图形表示

广义表的图形表示动画演示

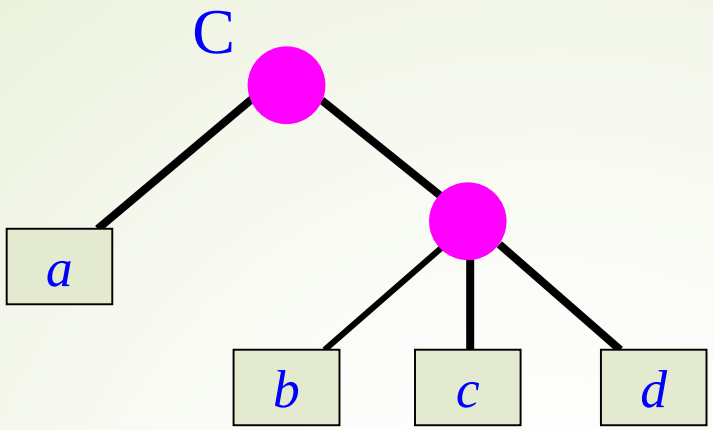
D(A(),B(e),C(a, •(b,c,d)))

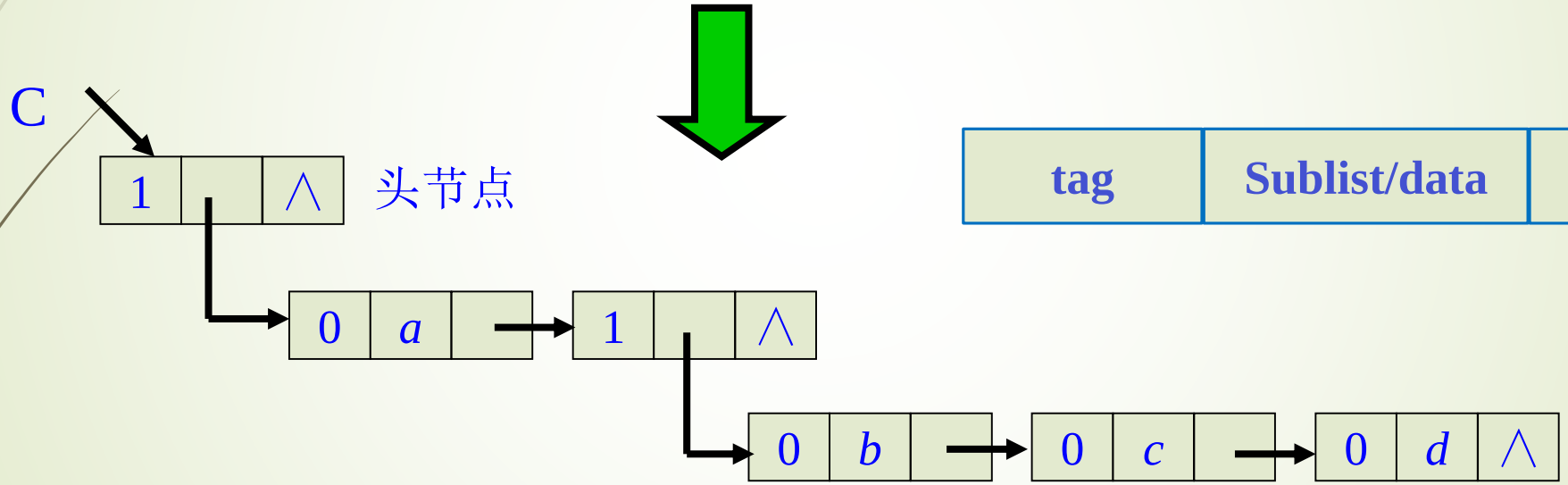


生成广义表图形表示

6.3.2 广义表的存储结构

广义表是一种递归的数据结构，因此很难为每个广义表分配固定大小的存储空间，所以其存储结构只好采用动态链式结构。





tag	Sublist/data	link

广义表的链式存储结构

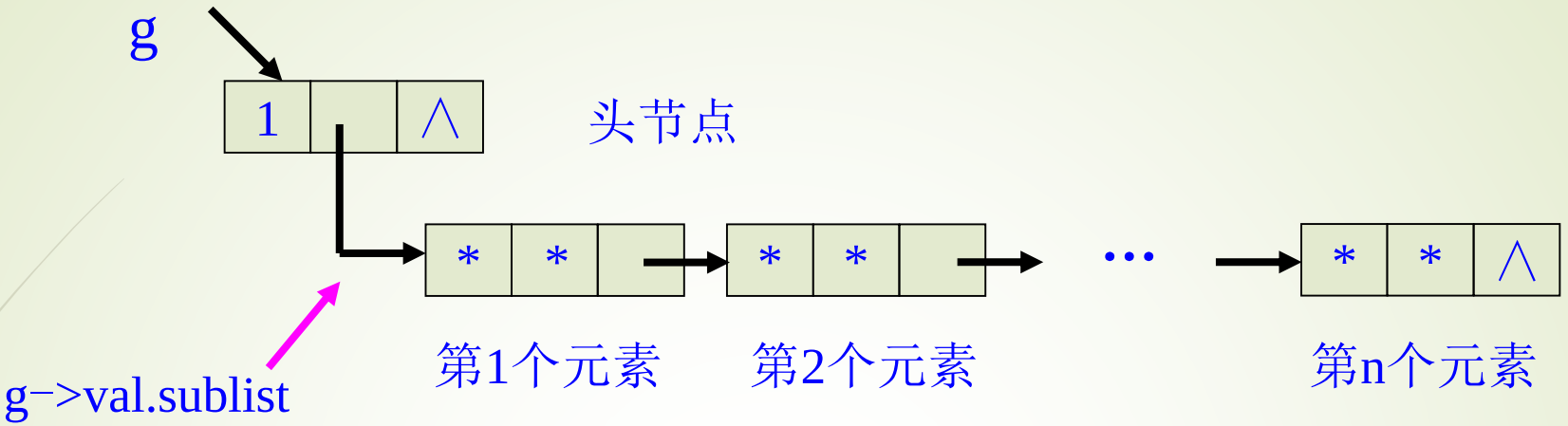
广义表节点类型定义：

```
typedef struct lnode
{  int tag;              //节点类型标识
   union
   {    ElemType data;
        struct lnode *sublist;
   }  val;
   struct lnode *link;   //指向下一个元素
} GLNode;               //广义表节点类型定义
```

6.3.3 广义表的运算

（1）求广义表的长度

在广义表中，同一层次的每个节点是通过link域链接起来的，所以可把它看做是由link域链接起来的单链表。这样,求广义表的长度就是求单链表的长度，可以采用以前介绍过的求单链表长度的方法求其长度。



求广义表长度的非递归算法如右：

```
int GLLength(GLNode *g)
//g为一个广义表头节点的指针
{   int n=0;
    g=g->val.sublist; //g指向广义表的第一个元素
    while (g!=NULL)
    {   n++;
        g=g->link;
    }
    return n;
}
```

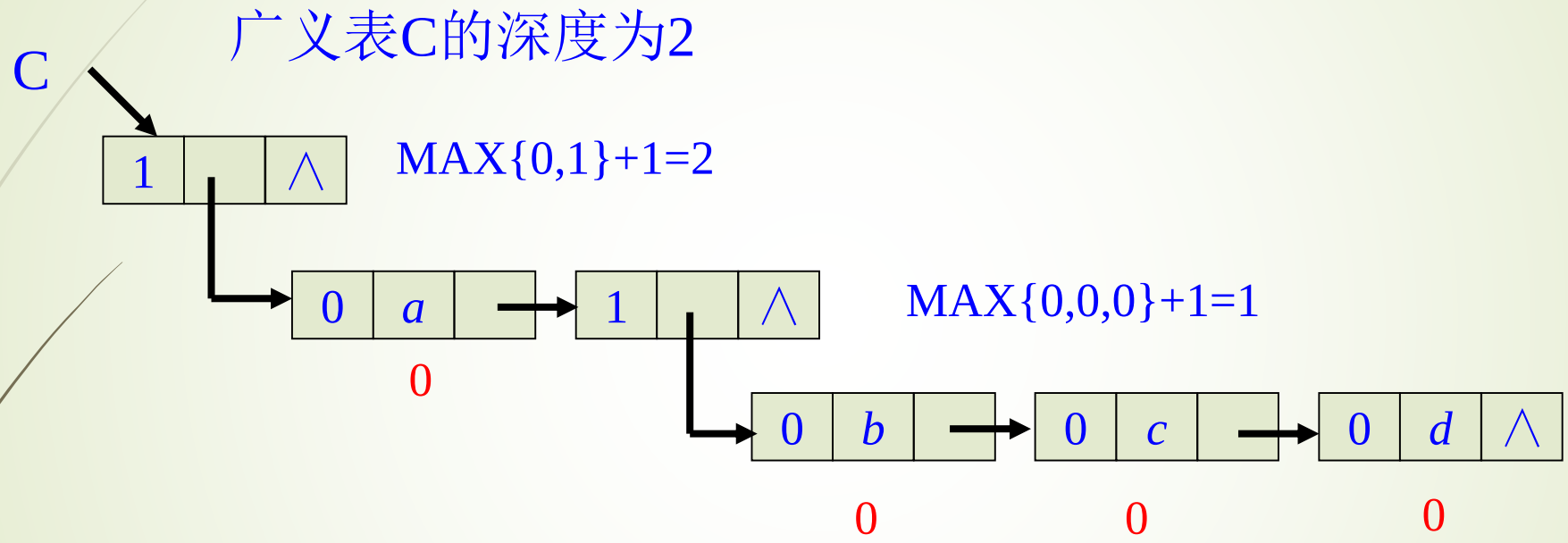
（2）求广义表的深度

对于带头节点的广义表g，广义表深度的递归定义是它等于所有子表中表的最大深度加1。若g为原子，其深度为0。

求广义表深度的递归模型f()如下：

$$f(g)=\begin{cases}0 & \text{若g为原子}\\ 1 & \text{若g为空表}\\ MAX\{f(subg)\}+1 & \text{其他情况，subg为g的子表}\end{cases}$$

求广义表深度的动画演示



```
int GLDepth(GLNode *g) //求带头节点的广义表g的深度
{   int max=0,dep;
    if (g->tag==0) return 0;    //为原子时返回0
    g=g->val.sublist;           //g指向第一个元素
    if  (g==NULL) return  1;    //为空表时返回1
    while (g!=NULL)             //遍历表中的每一个元素
    {   if (g->tag==1)          //元素为子表的情况
        {   dep=GLDepth(g); //递归调用求出子表的深度
            if (dep>max) max=dep;
               //max为同一层所求过的子表中深度的最大值
        }
        g=g->link;      //使g指向下一个元素
    }
    return(max+1);      //返回表的深度
}
```

（3）输出广义表

以g作为头节点的广义表的表头指针，打印输出该广义表时，需要对子表进行递归调用。

```
void DispGL(GLNode *g)   //输出广义表g
{   if (g!=NULL)        //表不为空判断
    {  //先输出g的元素
       if (g->tag==0) //g的元素为原子时
         printf("%c", g->val.data);  //输出原子值
       else          //g的元素为子表时
       {    printf("("); //输出'('
            if (g->val.sublist==NULL) //为空表时
               printf("#");
            else    //为非空子表时
               DispGL(g->val.sublist);//递归输出子表
             printf(")"); //输出')'
       }
       if (g->link!=NULL)
       {    printf(",");
            DispGL(g->link);    //递归输出g的兄弟
       }
     }
}
```

（4）建立广义表的链式存储结构

假定广义表中的元素类型ElemType为char类型，每个原子的值被限定为单个英文字母。并假定广义表是一个正确的表达式，其格式为：元素之间用一个逗号分隔，表元素的起止符号分别为左、右圆括号，空表在其圆括号内不包含任何字符。

例如“(a,(b,c,d),(#))”就是一个符合上述规定的广义表格式。

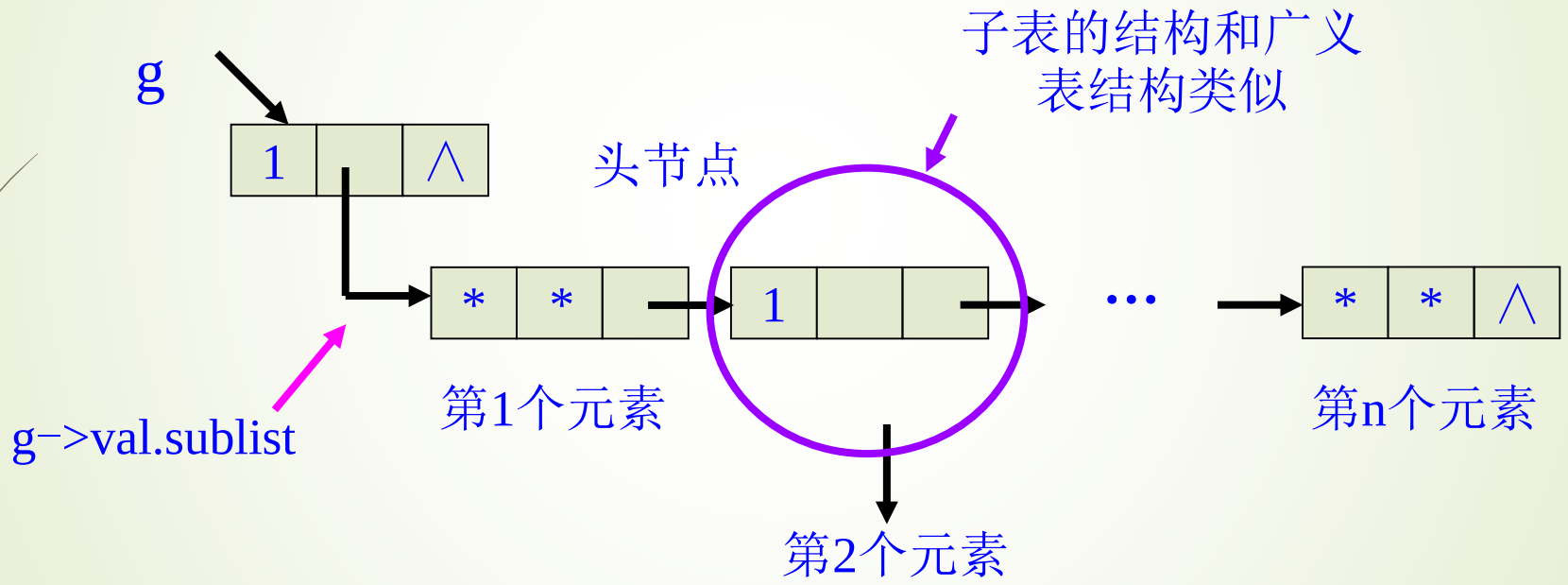
```
GLNode *CreateGL(char *&s)
//返回由括号表示法表示s的广义表链式存储结构
{  GLNode *g;
    char ch=*s++;       //取一个字符
    if (ch!='\0')       //串未结束判断
    {   g=(GLNode *)malloc(sizeof(GLNode));
                //创建一个新节点
        if (ch=='(')    //当前字符为左括号时
        { g->tag=1;     //新节点作为表头节点
          g->val.sublist=CreateGL(s);
             //递归构造子表并链到表头节点
        }
        else if (ch==')')
          g=NULL;       //遇到')'字符,g置为空
        else if (ch=='#')//遇到'#'字符,表示空表
          g=NULL;
        else            //为原子字符
        {   g->tag=0; //新节点作为原子节点
            g->val.data=ch;
        }
     }
    else            //串结束,g置为空
       g=NULL;
    ch=*s++;        //取下一个字符
```

```
    if (g!=NULL)        //串未结束，继续构造兄弟节点
      if (ch==',')      //当前字符为','
        g->link=CreateGL(s); //递归构造兄递节点
      else      //没有兄弟了,将兄弟指针置为NULL
        g->link=NULL;
    return g;     //返回广义表g
}
```

6.3.4 广义表的递归算法设计（补充）

广义表的上述链式存储结构具有递归性，可以从两个方面来理解这种递归性，从而得到广义表的两种递归算法方法。

解法1：一个非空广义表的基本存储结构如下图所示，它是由若干元素构成的，每个子表元素的存储结构和整个广义表的存储结构是相似的。因此，这些子表的处理和整个广义表的处理是相似的。



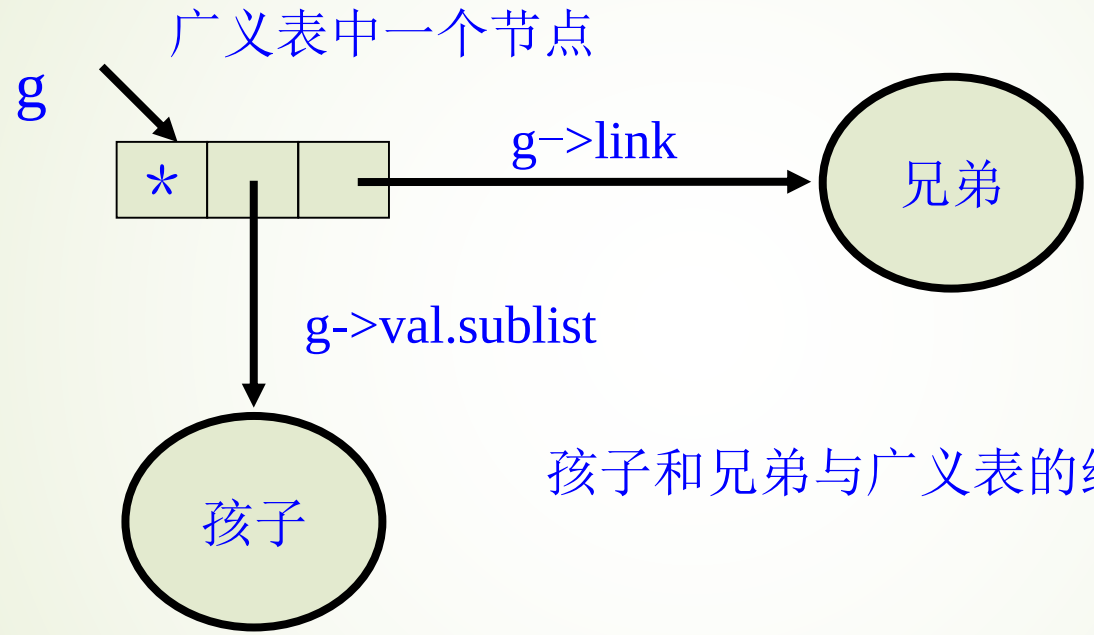
从这个角度出发设计求解广义表递归算法的一般格式如下：

```
void fun1(GLNode *g) //g为广义表头结点指针
{   GLNode *g1=g->val.sublist; //g1指向第一个元素
    while (g1!=NULL)  //元素未处理完循环
    {
      if (g1->tag==1) //为子表时
        fun1(g1);       //递归处理子表
      else              //为原子时
        原子处理语句;    //实现原子操作
    g1=g1->link;    //处理兄弟
    }
}
```

先处理一个元素

再处理后继元素

解法2：一个非空广义表存储结构中每个结点的基本结构如下图所示，每个原子结点的data域为原子值（没有元素）、link域指向其兄弟，每个表/子表结点的sublist指向它元素、link指向兄弟。



因此，对于广义表中的任一个结点，如果它是表/子表结点，其元素和兄弟的处理与整个广义表的处理是相似的，如果它是原子结点，其兄弟的处理与整个广义表的处理是相似的。

```
void fun2(GLNode *g)      //g为广义表结点指针
{   if (g!=NULL)
    {
      if (g->tag==1)      //为子表时
         fun2(g->val.sublist);//递归处理其元素
      else                //为原子时
         原子处理语句;      //实现原子操作

      fun2(g->link);
    }
}
```

先递归处理孩子

递归处理兄弟

【例6.5：p151】 设计一个算法求给定的广义表g中的原子个数。

解法1的算法：

```
int Count1(GLNode *g)
{   int n=0;
    GLNode *g1=g->val.sublist;
    while (g1!=NULL) //对每个元素进行循环处理
    {  if (g1->tag==0)  //为原子时
         n++;       //原子个数增1
       else         //为子表时
         n+=Count1(g1);//累加元素的原子个数
      g1=g1->link;    //累加兄弟的原子个数
    }
    return n;       //返回总原子个数
}
```

解法2的算法：

```
int Count2(GLNode *g)
{   int n=0;
    if (g!=NULL)          //对每个元素进行循环处理
    {   if (g->tag==0) //为原子时
          n++;          //原子个数增1
        else            //为子表时
          n+=Count2(g->val.sublist);   //累加元素的原子个数
        n+=Count2(g->link);//累加兄弟的原子个数
    }
     return n;          //返回总原子个数
}
```

解法3的算法（教程中的解法）：

设f(g)表示广义表g中的原子个数，根据广义表的递归特性得到递归模型如下：

f(g) = 0　　当g=NULL

f(g) = 1+f(g−>link)　　当g为原子节点时

f(g) = f(g−>val.sublist)+f(g−>link)　　当g为表或子表节点时

```
int Count3(GLNode *g)    //求广义表g中的原子个数
{   if (g!=NULL)
     {  if (g->tag==0)
           return 1+Count3(g->link);
        else
           return Count3(g->val.sublist)+Count3(g->link);
      }
    else return 0;
}
```

实验五 稀疏矩阵的加法和乘法问题

【基本要求】

- 两个稀疏矩阵的三元组存储结构生成；
- 实现稀疏矩阵的转置；
- 实现两个稀疏矩阵的加运算；
- 实现两个稀疏矩阵的乘法运算。

$$a_{4\times 4} = \begin{bmatrix} 1 & 0 & 3 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

ta:

i	j	a_{ij}
0	0	1
0	2	3
1	1	1
2	2	1
3	2	1
3	3	1

$$b_{4\times 4} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

tb:

i	j	b_{ij}
0	0	3
1	1	4
2	2	1
3	3	2

矩阵的加法运算

$$a_{4\times 4}=\begin{bmatrix}1&0&3&0\\0&1&0&0\\0&0&1&0\\0&0&1&1\end{bmatrix}$$

ta:

m →

i	*j*	a_{ij}
0	0	1
0	2	3
1	1	1
2	2	1
3	2	1
3	3	1

$$b_{4\times 4}=\begin{bmatrix}3&0&0&0\\0&4&0&0\\0&0&1&0\\0&0&0&2\end{bmatrix}$$

tb:

n →

i	*j*	b_{ij}
0	0	3
1	1	4
2	2	1
3	3	2

ta+tb：行数和列数不相等时不能相加。

当ta.m[i]=tb.n[i]时：

如果 ta.m[j]<tb.n[j],将ta.m[a_{ij}]放入结果中

如果ta.m[j]>tb.n[j],将tb.m[b_{ij}]放入结果中

如果ta.m[j]=tb.n[j],将ta.m[a_{ij}] + tb.m[b_{ij}], 放入结果中

当ta.m[i]<tb.n[i]时：

将ta.m[a_{ij}]放入结果中

当ta.m[i]>tb.n[i]时：

将tb.m[b_{ij}]放入结果中

矩阵的乘法运算

$$a_{4\times 4}=\begin{bmatrix}1&0&3&0\\0&1&0&0\\0&0&1&0\\0&0&1&1\end{bmatrix}$$

ta:

m →

i	*j*	a_{ij}
0	0	1
0	2	3
1	1	1
2	2	1
3	2	1
3	3	1

$$b_{4\times 4}=\begin{bmatrix}3&0&0&0\\0&4&0&0\\0&0&1&0\\0&0&0&2\end{bmatrix}$$

tb:

n →

i	*j*	b_{ij}
0	0	3
1	1	4
2	2	1
3	3	2

ta×tb：a的列数和b的行数不相等时不能相乘。

getvalue（TSMatrix c，int i，int j）在三元组c中找到位置为（i，j）的元素值。

```
for( i=0;i<a.rows,i++)
    for(j=0;j<b.cols;j++)
    {   s=0;
        for (k=0;k<a.cols; k++)
          s=s+getvalue(ta,i,k)*getvalue(tb,k,j)
        ……
    }
```

本章小结

本章基本学习要点如下：

（1）理解数组和一般线性表之间的差异。

（2）重点掌握数组的顺序存储结构和元素地址计算方法。

（3）掌握各种特殊矩阵如对称矩阵、上、下三角矩阵和对角矩阵的压缩存储方法。

（4）掌握稀疏矩阵的各种存储结构以及基本运算实现算法。

（5）掌握广义表的定义和广义表的链式存储结构。

（6）掌握广义表的基本运算，包括创建广义表、输出广义表、求广义表的长度和深度。

—本章完—