数据结构

北京邮电大学 信息安全中心
武 斌 杨 榆

上章内容

上一章（串）内容：

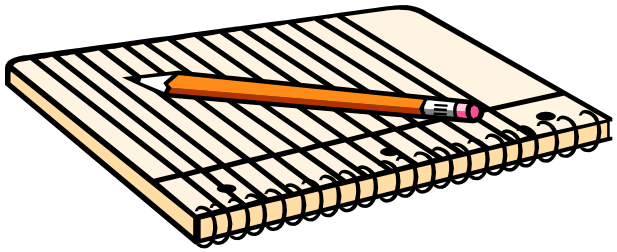
- 理解“串”类型定义中各基本操作的特点
- 能正确利用这些特点进行串的其它操作
- 理解串类型的各种存储表示方法
- 理解串匹配的各种算法

本次课程学习目标

学习完本次课程，您应该能够：

- 理解数组类型的特点及其在高级编程语言中的存储表示和实现方法
- 掌握数组在"以行为主"的存储表示中的地址计算方法
- 掌握特殊矩阵的存储压缩表示方法
- 理解稀疏矩阵的两类存储压缩方法的特点及其适用范围
- 掌握广义表的结构特点及其存储表示方法

矩阵的压缩存储



- 5.1 数组的类型定义
- 5.2 数组的顺序存储表示
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构

矩阵的压缩存储——行逻辑链接的顺序表

●2、行逻辑链接的顺序表

- 已知三元组顺序表按行序有序存放矩阵的非零元，便于按**行序顺序处理**矩阵元素；但因为各行非零元个数无规律，若需处理某行的非零元，则必须从三元组顺序表第一元素开始查找。
- 为了便于**随机存取任意一行**的非零元，需要知道每行首个非零元素所在位置。
- 矩阵快速转置算法使用M中每列首个非零元素位置数组 cpot[M.nu+1]来降低操作复杂度。
- 可借鉴快速转置算法思路，创建指示行信息的辅助数组cpot，固定在稀疏矩阵存储结构中。
- 可将 cpot 中的值视作指向每行第一个非零元的“指针”，故称这种表示方法为**“行逻辑链接”**的顺序表。

矩阵的压缩存储——行逻辑链接的顺序表

●行逻辑链接的三元组顺序表的结构定义

```
const MAXRC = 500;                  // 矩阵行数和列数的最大值
const MAXSIZE=12500;                // 假设非零元个数的最大值为12500
typedef struct {
    Triple data[MAXSIZE + 1];       // 非零元三元组表，data[0] 未用
    int rpos[MAXRC + 1];            // 指示各行第一个非零元在data中的位置
    int mu, nu, tu;                 // 矩阵的行数、列数和非零元的个数
} RLSMatrix;                        // 行逻辑链接顺序表
```

矩阵的压缩存储——行逻辑链接的顺序表

- 以行逻辑链接顺序表来表示稀疏矩阵时，在部份矩阵操作中具有优势。以矩阵乘法分析。
  - 已知$Q_{m_1\times n_2} = M_{m_1\times n_1} \times N_{m_2\times n_2}$。以二维数组表示矩阵时，矩阵乘法如下：

$$Q(i,j)=\sum_{k=1}^{n_1} M(i,k)\times N(k,j) \qquad \begin{matrix} 1\le i\le m_1 \\ 1\le j\le n_2 \end{matrix}$$

  - **void** multiplication(**double** M[m][n], **double** N[n][p], **double** Q[m][p])

```
{
   for (i=0; i<m; i++)
      for (j=0; j<p; j++){
         Q[i][j]=0;
         for (k=0; k<n; k++)
            Q[i][j]+=M[i][k]*N[k][j];
      }
}
```

矩阵的压缩存储——行逻辑链接的顺序表

- 以行逻辑链接顺序表来表示稀疏矩阵时，在部份矩阵操作中具有优势。以**矩阵相乘**分析。
  - 对于稀疏矩阵，以实例分析上述方法的效率：
  - $M=\begin{bmatrix}3 & 0 & 0 & 5\\ 0 & -1 & 0 & 0\\ 2 & 0 & 0 & 0\end{bmatrix}$，$N=\begin{bmatrix}0 & 2\\ 1 & 0\\ -2 & 4\\ 0 & 0\end{bmatrix}$，$Q_{3\times2}=M_{3\times4}\times N_{4\times2}=\begin{bmatrix}0 & 6\\ -1 & 0\\ 0 & 4\end{bmatrix}$
  - 按照定义，不论$M(i,k)$和$N(k,j)$是否为零，都要进行一次乘法操作，对于稀疏矩阵而言，这些操作是无效、应避免的。例如：
  - $Q(1,1)=M(1,1)\times N(1,1)+M(1,2)\times N(2,1)+M(1,3)\times N(3,1)+M(1,4)\times N(4,1)=3\times\mathbf{0}+\mathbf{0}\times1+\mathbf{0}\times(-2)+5\times0$

矩阵的压缩存储——行逻辑链接的顺序表

- 总结，对于稀疏矩阵，只要找$M(i,k)$和$N(k,j)$**都非零**的元素相乘即可，也就是说，在三元组顺序表表示的$M.data$和$N.data$中找到**相应**元素（$(i,k,M(i,k))$和$(k,j,N(k,j))$）相乘即可。
- 例如：
- $M=\begin{bmatrix}3&0&0&5\\0&-1&0&0\\2&0&0&0\end{bmatrix}$，$N=\begin{bmatrix}0&2\\1&0\\-2&4\\0&0\end{bmatrix}$，$Q_{3\times2}=M_{3\times4}\times N_{4\times2}=\begin{bmatrix}0&6\\-1&0\\0&4\end{bmatrix}$
- 对于M的非零元$M(1,1)$而言，$M(1,1)$应与$N(1,1)$和$N(1,2)$相乘，而$N(1,1)$为0，所以只需计算M的$(1,1,3)$应与N的$(1,2,2)$即可。
- **$M(i,k)$对应的非零元在N的第k行**，比起带有列起始位置的辅助存储，显然带行逻辑链接的数据结构更适合该项操作。

矩阵的压缩存储——行逻辑链接的顺序表

- 一般情况 $Q_{m_1\times n_2} = H_{m_1\times n_1} \times T_{m_2\times n_2}, m_2 = n_1$
- $[q_{i,1}, q_{i,2}, \cdots q_{i,n_2}] =$
- $$\left[[h_{i,1}, h_{i,2}, \cdots h_{i,n_1}] \times \begin{bmatrix} t_{1,1} \\ t_{2,1} \\ \vdots \\ t_{n_1,1} \end{bmatrix}, [h_{i,1}, h_{i,2}, \cdots h_{i,n_1}] \times \begin{bmatrix} t_{1,2} \\ t_{2,2} \\ \vdots \\ t_{n_1,2} \end{bmatrix}, \cdots, [h_{i,1}, h_{i,2}, \cdots h_{i,n_1}] \times \begin{bmatrix} t_{1,n_2} \\ t_{2,n_2} \\ \vdots \\ t_{n_1,n_2} \end{bmatrix} \right] =$$
- $$\begin{bmatrix} h_{i,1} \times [t_{1,1}, t_{1,2}, \cdots t_{1,n_2}] + \\ h_{i,2} \times [t_{2,1}, t_{2,2}, \cdots t_{2,n_2}] + \\ \cdots \\ h_{i,n_1} \times [t_{n_1,1}, t_{n_1,2}, \cdots t_{n_1,n_2}] \end{bmatrix}$$

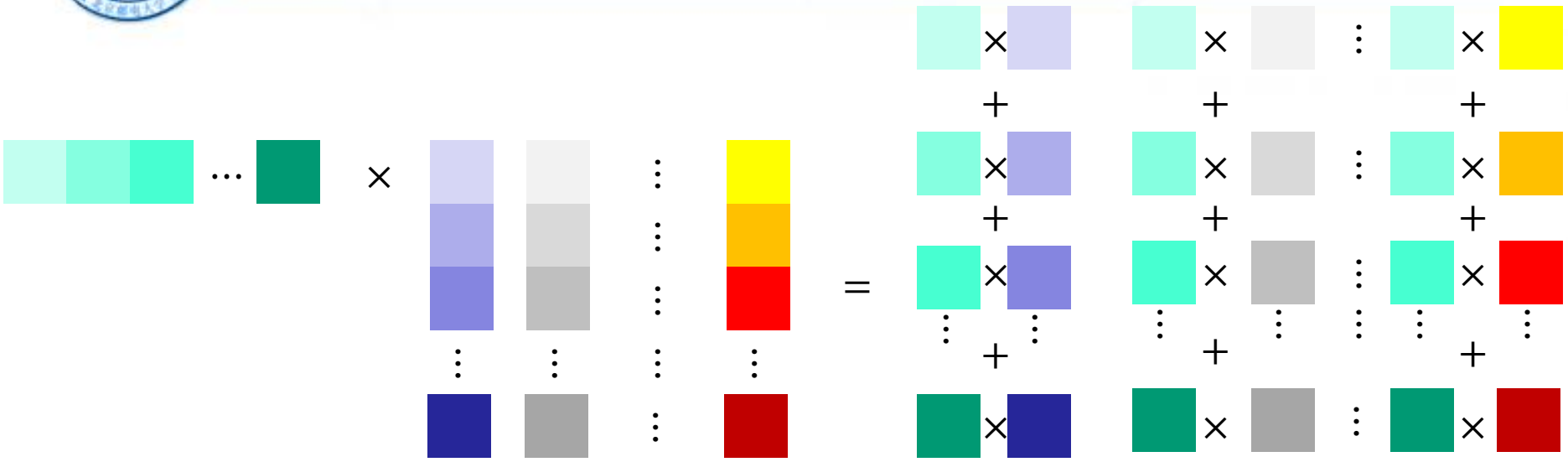
无论行逻辑还是列逻辑链接顺序表都难以简单地同时顺序访问H第i行和T第j列。

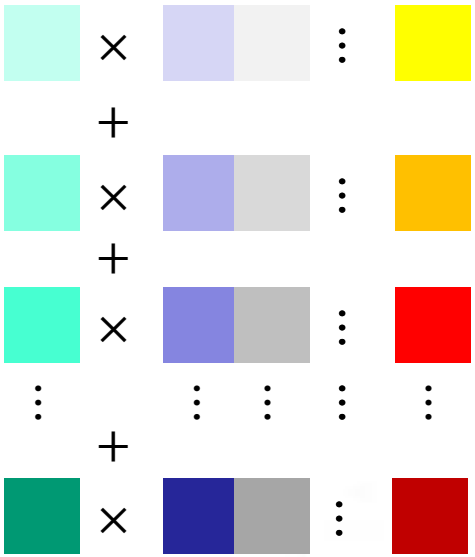
$h_{i,k}$与T第k行相乘。

Q的第i行是H第i行中各个元素$h_{i,k}$与T第k行相乘的累加和。对于矩阵T和H而言，操作都是按行顺序访问，显然行逻辑链接顺序表方式有优势。

左矩阵第i行第k列元素h(i,k)加权右矩阵第k行所有元素t(k,:)，累加得到矩阵Q的第i行。

矩阵的压缩存储——行逻辑链接的顺序表





$$\begin{bmatrix} h_{i,1} \times [t_{1,1}, t_{1,2}, \cdots t_{1,n_2}] + \\ h_{i,2} \times [t_{2,1}, t_{2,2}, \cdots t_{2,n_2}] + \\ \cdots \\ h_{i,n_1} \times [t_{n_1,1}, t_{n_1,2}, \cdots t_{n_1,n_2}] \end{bmatrix}$$

总结：每次计算积矩阵的一行效率更高。积矩阵第i行是左矩阵第i行第k列元素h(i,k)加权右矩阵第k行所有元素t(k,:)，累加得到矩阵Q的第i行。

矩阵 $\begin{bmatrix} m_{i,1} \times [n_{1,1}, n_{1,2}, \cdots n_{1,n_2}] + \\ m_{i,2} \times [n_{2,1}, n_{2,2}, \cdots n_{2,n_2}] + \\ \cdots \\ m_{i,n_1} \times [n_{n_1,1}, n_{n_1,2}, \cdots n_{n_1,n_2}] \end{bmatrix}$ 辑链接的顺序表

- 例如：上述算法矩阵表示
- $M = \begin{bmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix}$,
- $N = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix}$
- $Q(1,:) = [q_{1,1}, q_{1,2}] = \begin{bmatrix} (3 \times 0) & (3 \times 2) \\ + & + \\ (0 \times 1) & (0 \times 0) \\ + & + \\ (0 \times -2) & (0 \times 4) \\ + & + \\ (5 \times 0) & (5 \times 0) \end{bmatrix}$

- 例如：上述算法稀疏表示

M

1	1	3
1	4	5
2	2	-1
3	1	2

N

1	2	2
2	1	1
3	1	-2
3	2	4

Q第1行：

1.第一次（$Q(1,:)^{(1)}$）：M(1,1,3)与N第1行非零元(1,2,2)相乘。

$$Q(1,:)^{(1)} = (0,3 \times 2) = (0,6)$$

2.第二次（$Q(1,:)^{(2)}$）：M(1,4,5)应与N第4行相乘非零元相乘，而N第4行无非零元，所以 $Q(1,:)^{(2)} = Q(1,:)^{(1)} + 5 \times N(4,:) = (0,6)$，无变化。

3.M第1行所有非零元处理完毕，Q第1行值为(0,6)

矩阵的压缩存储——行逻辑链接的顺序表

- 练习：稀疏矩阵M和N表示如下，请计算$M \times N$

M(mu:100,nu:40)

2	15	3
2	38	5
40	4	-1
40	19	2
40	27	7
100	1	5
100	2	7
100	9	-1
100	15	-4

N(mu:40,nu:75)

1	2	2
1	3	-3
2	2	1
9	5	4
15	3	-2
15	15	4
38	5	6
38	15	-7

Q(mu:100,nu:75)

2	3	-6
2	5	30
2	15	-23
100	2	17
100	3	-7
100	5	-4
100	15	-16

矩阵的压缩存储——行逻辑链接的顺序表

- 对行逻辑链接的三元组顺序表表示的稀疏矩阵，矩阵M、N相乘得Q，
- Q第i行计算方法描述为：

设累加器 ctemp 的容量为 p(p 为 Q 中列数，即 N 的列数，保存、累加Q第i行中间结果，即Q(i,:)=0;Q(i,:)=Q(i,:)+M(i,k)*N(k,:))

初始化累加器 ctemp[ ]=0;

for (M 中第 i 行的所有非零元 M.data[p])

{

```
    brow=M.data[p].j;      // 该非零元在 M 中的列号
    for (N中第 brow 行的非零元 N.data[q])
    {
        ccol= N.data[q].j;   // 该非零元在N中的列号
       ctemp[ccol]+=M.data[p].e * N.data[q].e;
    }
```

}

矩阵的压缩存储——行逻辑链接的顺序表

- 容易看出上述运算的结果 ctemp 中所有非零分量即为 Q 中第 i 行的所有非零元。
- 实际上，乘积 Q 中哪些元为零哪些元非零，难以直接根据 M 和 N 的单个非零元判断。
- 需要通过上述运算得到 ctemp，将其非零元按列号从小到大依次存入 Q.data 中。

矩阵的压缩存储——行逻辑链接的顺序表

- 例如：

$$M=\begin{bmatrix}0&1&0\\2&0&-4\\-1&0&0\\0&-2&3\end{bmatrix},N=\begin{bmatrix}2&0&0&6\\0&-4&0&0\\0&-1&0&3\end{bmatrix},Q=M\times N=\begin{bmatrix}0&-4&0&0\\4&4&0&0\\-2&0&0&-6\\0&5&0&9\end{bmatrix}$$

M.data

	i	j	e
1	1	2	1
2	2	1	2
3	2	3	-4
4	3	1	-1
5	4	2	-2
6	4	3	3

p →

q →

N.data

	i	j	e
1	1	1	2
2	1	4	6
3	2	2	-4
4	3	2	-1
5	3	4	3

Q.data

	i	j	e
1	1	2	-4
2			
3			
4			
5			
6			

- M.rpos的值为：

1	2	3	4	（5）
1	2	4	5	7

- N.rpos的值为：

1	2	3	（4）
1	3	4	6

矩阵的压缩存储——行逻辑链接的顺序表

- ctemp（Q中第二行）

1	2	3	4
0+ 2×2 =4	0+ (-4) ×(-1) =4	0	0+ 2×6+ (-4) ×3 =0

M.data

p →

	i	j	e
1	1	2	1
2	2	1	2
3	2	3	-4
4	3	1	-1
5	4	2	-2
6	4	3	3

- M.rpos的值为：

行号	1	2	3	4	(5)
M.rpos	1	2	4	5	7

- N.rpos的值为：

行号	1	2	3	(4)
N.rpos	1	3	4	6

N.data

q →

	i	j	e
1	1	1	2
2	1	4	6
3	2	2	-4
4	3	2	-1
5	3	4	3

矩阵的压缩存储——行逻辑链接的顺序表

- 由此，**两个稀疏矩阵相乘（Q＝M ×N）的过程**可大致描述如下：

```
Q初始化;
  If（ Q是非零矩阵 ）{                                  // 逐行求积
      for (arow=1; arow<=M.mu; ++arow)      // 处理M的每一行
      {
          ctemp[] = 0;                                    // 累加器清零
          计算 Q 中第 arow 行的积并存入 ctemp[] 中；
          将 ctemp[] 中非零元压缩存储到 Q.data；
      } // for arow
  } // if
```

矩阵的压缩存储——行逻辑链接的顺序表

- 其中：
  - 对矩阵Q进行初始化的操作如下：

```
Q.mu = M.mu;   // Q的行数和M相同
Q.nu = N.nu;    // Q的列数和N相同
Q.tu = 0;        // Q的非零元个数初始化为零
```

  - 求得Q中一行非零元的操作为：

```
ctemp[] = 0;        // 当前行各元素累加器清零

 for(p=M.rpos[arow];p<M.rpos[arow+1];++p)
{   // 处理 M 当前行中每一个非零元
      brow=M.data[p].j;        // 找到对应元在N中的行号
      for(q=N.rpos[brow];q<N.rpos[brow+1];++q)
      {// M.data[p] 乘以N当前行中每一个非零元
            ccol = N.data[q].j;      // 乘积元素在Q中列号
            ctemp[ccol]+=M.data[p].e * N.data[q].e;
      } // for q
 } // for p
```

矩阵的压缩存储——行逻辑链接的顺序表

● 算法 5.3

```
bool MultSMatrix(RLSMatrix M, RLSMatrix N, RLSMatrix &Q)
{
// 采用行逻辑链接存储表示，求矩阵乘积Q=M×N。
if (M.nu!= N.mu)   return ERROR;
Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;
                                    // M的列数和N的行数不等，不能相乘
if (M.tu*N.tu != 0) {          // M和N中均含有非零元，对矩阵Q进行初始化;
  for (arow=1; arow<=M. mu; ++arow)
  {
    //处理M(即Q)的每一行，求得 Q 中第crow( =arow)行的非零元
```

矩阵的压缩存储——行逻辑链接的顺序表

```
ctemp[] = 0;      // 当前行各元素累加器清零
Q.rpos[arow] = Q.tu+1; // 当前行在Q中的起始位置
for(p=M.rpos[arow];p<M.rpos[arrow+1];++p)
{   // 处理 M 当前行中每一个非零元
  brow=M.data[p].j; // 找到对应元在N中的行号
  for(q=N.rpos[brow];q<N.rpos[brow+1];++q)
  {
        ccol = N.data[q].j; // 乘积元素在Q中列号
        ctemp[ccol]+=M.data[p].e * N.data[q].e;
  } // for q
} // for p
```

矩阵的压缩存储——行逻辑链接的顺序表

```
    for (ccol=1; ccol<=Q.nu; ++ccol)   // 压缩存储该行非零元
        if (ctemp[ccol]) {
            if (++Q.tu > MAXSIZE)  return ERROR;
            Q.data[Q.tu] = (arow, ccol, ctemp[ccol]);
        } // if
    } // for arow
} // if
return OK;
} // MultSMatrix
```

- 该算法的时间复杂度为

- 累加器ctemp初始化的复杂度O(M.mu×N.nu)，求Q的所有非零元的复杂度O（M.tu×N.tu/N.mu），Q压缩存储的复杂度O(M.mu×N.nu)。

  O (M.mu×N.nu+M.tu×N.tu/N.mu)。

- 其中，N.tu/N.mu 表示N中每一行非零元个数的平均值。

矩阵的压缩存储——行逻辑链接的顺序表

- 如何建立稀疏矩阵的行逻辑链接的顺序表？

  - 首先应该输入该矩阵的行数、列数以及非零元的个数，然后依次输入各个非零元的行号、列号和它的值，并在输入每一行的第一个非零元的同时为 **rpos** 中相应分量赋值。显然，对于顺序结构应尽少进行“移动元素”的操作，因此非零元的输入次序应对行有序，且同一行的非零元按列有序。

  - 算法中需要注意的是，可能存在某一行或连续几行都没有非零元的情况，则这些行的“第一个非零元在顺序表中的位置”应该和当前输入的那个非零元所在行相同。

  - 虽然，表面上 rpos 中各分量的值只是指示每一行的第一个非零元在 data 中的位置，实际上它还隐含着一个信息，即"第 k 行的非零元的个数=rpos[k+1]-rpos[k]"。为了使这个公式也适用于最后一行，在 rpos 中增添一个下标为(行数+1)的分量。

矩阵的压缩存储——行逻辑链接的顺序表

● 算法 5.2

```
void Create_SM(RLSMatrix& M)
{// 以行序为主序的次序逐个输入非零元，建立稀疏矩阵的行逻辑链接顺序表
    cin >> M.mu>>M.nu>>M.tu;
    k = 1; curRow = 0;
    for (i=1; i<= M.tu; i++) {
        cin>>M.data[k].i >> M.data[k].j >> M.data[k].e;
        while (curRow < M.data[k].i)
            M.rpos[++curRow]=k;
        ++k;
    } // for
    while (curRow < M.mu+1)          // 剩余的没有非零元的行
        M.rpos[++curRow]=k;
}
```

❊上述算法的控制结构只有一个for循环，显然它的时间复杂度为 ***O(M.tu)***。

矩阵的压缩存储——十字链表

3、十字链表

- 以上讨论的矩阵运算都不改变参与运算的矩阵本身，即在它的存储结构中没有进行插入、删除之类的操作。
- 如果矩阵运算的结果将增加或减少已知矩阵中的非零元的个数，则显然不宜采用顺序存储结构，而应以链式映象作为三元组线性表的存储结构。

矩阵的压缩存储——十字链表

- 每个非零元以**含5个域的结点**表示，除了表示**非零元信息的三元组(i,j,e)**之外，添加了两个链域：一个是链接同一行下一个非零元结点的 **right 域**，另一个是链接同一列下一个非零元结点的 **down 域**。
- 每个非零元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，整个矩阵构成了一个十字交叉的链表，故称为**十字链表**，以两个分别存放行链表的头指针和列链表的头指针的一维数组表示。

矩阵的压缩存储——十字链表

➔ 稀疏矩阵的十字链表存储表示

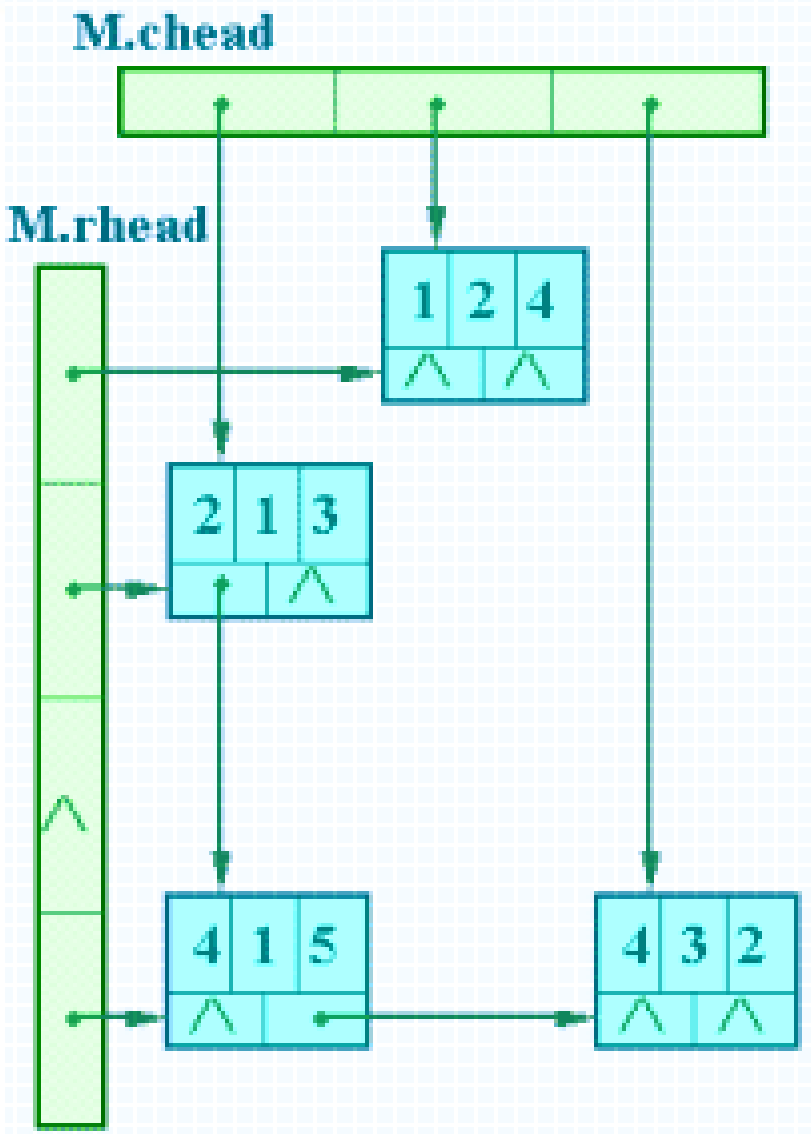
```
typedef struct OLNode {        // 结点结构定义
    int i, j;        // 该非零元的行和列下标
    ElemType e;
    struct OLNode *right, *down;     // 该非零元所在行表和列表的后继链域
} *OLink;
typedef struct {                    // 链表结构定义
    OLink *rhead, *chead;    // 行和列链表头指针向量基址由CreateSMatrix分配
    int mu, nu, tu;          // 稀疏矩阵的行数、列数和非零元个数
} CrossList;
```

矩阵的压缩存储——十字链表

●例如，矩阵

$$M=\begin{bmatrix}0 & 4 & 0\\ 3 & 0 & 0\\ 0 & 0 & 0\\ 5 & 0 & 2\end{bmatrix}$$

的十字链表如图所示。



矩阵的压缩存储——十字链表

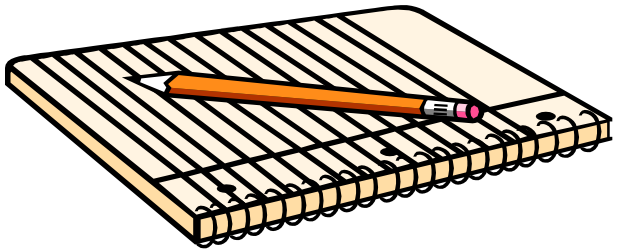
● 算法 5.4

```
Status CreateSMatrix_OL (CrossList &M)
{   // 创建稀疏矩阵M的十字链表存储结构,若存储分配失败，则返回FALSE
    cin >> M.mu >> M.nu >> M.tu;        // 输入M的行数、列数和非零元个数
    if (!(M.rhead = (OLink*)malloc((M.mu+1)*sizeof(OLink))))
        exit(OVERFLOW); // 存储分配失败
    if (!(M.chead = (OLink*)malloc((M.nu+1)*sizeof(OLink))))
        exit(OVERFLOW); // 存储分配失败
    M.rhead[ ] = M.chead[ ] = NULL ;// 初始化行列头指针向量；各行列链表为空链表
    for ( scanf(&i, &j, &e);  i!=0;  scanf(&i, &j, &e))
    {                                                  // 按任意次序输入非零元
        if (!(p = (OLNode*)malloc(sizeof(OLNode )))) exit(OVERFLOW);
          p->i = i; p->j = j; p->e = e;                // 生成结点
        if (M.rhead[i]==NULL || j < M.rhead[i]->j) {
          p->right = M.rhead[i]; M.rhead[i] = p;
        }
```

矩阵的压缩存储——十字链表

```
        else {                                  // 寻查在行表中的插入位置
            for( q = M.rhead[i]; q->right && q->right->j < j; q=q->right; );
            p->right = q->right; q ->right = p;
        }                                       // 完成行插入
        if (M.chead[j] ==NULL || i < M.chead[j]->i) {
            p->down = M.chead[j];  M.chead[j] = p;
         }
        else {                                  // 寻查在列表中的插入位置
            for( q = M.chead[j]; q->down && q->down->i < i; q = q->down; );
            p->down = q->down; q->down = p;
        }                                       // 完成列插入
    } // for
    return OK;
} // CreateSMatrix_OL
```

广义表的定义



- 5.1 数组的类型定义
- 5.2 数组的顺序存储表示
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构

广义表的定义

- 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$a_1,a_2,a_3,\ldots,a_n$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。

- **广义表是n(n>=0)个元素$a_1,a_2,a_3,\ldots,a_n$的有限序列**，其中a_i或者是原子项，或者是一个广义表。通常记作LS=（$a_1,a_2,a_3,\ldots,a_n$)。LS是广义表的名字，n为它的长度。若a_i是广义表，则称它为LS的子表。

广义表的定义

- 例1：

  D=(E，F)

  E=(a, (b,c))

  F=(d, (e))

- 例2

  A=( )

  B=(a, B)=(a,(a,(a, ……,)))

  C=(A, D, F)

广义表的定义

- 抽象数据类型广义表的定义如下：

ADT GList {

数据对象：

D＝{ e_i | i=1,2,..,n; n≥0; e_i ∈AtomSet 或 e_i ∈GList, AtomSet为某个数据对象 }

数据关系：

R1＝{< e_{i-1} , e_i >| e_{i-1} , e_i ∈D, 2≤i≤n }

广义表的定义

- 从上述定义可知，广义表LS=（$a_1,a_2,a_3,\ldots,a_n$)兼有线性结构和层次结构的特性，归纳如下：
  - 1. 广义表中的数据元素有固定的相对次序；
  - 2. 广义表的长度定义为最外层括弧中包含的数据元素个数；
    - 例如广义表E=(a, (b,c))，及其子表（b,c）的长度都是2。
  - 3. 广义表的深度定义为广义表书写形式中括弧的最大重数；
    - 例如广义表E=(a, (b,c))，D=((),(e),(a,(b,c,d)))的深度分别是2和3。
  - 4. 广义表可被其它广义表所共享；
  - 5. 广义表可以是一个递归的表，递归表的深度可以是无穷，但长度有限；
    - 例如广义表E=(a, E)=(a,(a,(a,(…)))) 的长度都是2，深度是无穷。
  - 6. 对于任意一个非空广义表LS=（a1,a2,a3,…,an) ，它的第一个数据元素被定义为广义表的“表头”，Head(LS)=a1，而由其余数据元素构成的广义表被定义为广义表的“表尾”，Tail(LS)= (a2,a3,…,an) 。

广义表的定义

- 已知： A=( )， D=(E，F)， E=(a, (b,c))， F=(d, (e))

  LS = (A, D) = (( ), (E, F)) = (( ), ((a, (b, c)), F))

求：

Head(LS) ,Tail(LS), Head(D) , Tail(D), Head(E) , Tail(E),
Head(((b,c))), Tail (((b,c))), Head((b,c)), Tail((b,c)),
Head((c)) ,Tail((c)),

Head(LS) = A	Tail(LS) = (D)
Head(D) = E	Tail(D) =(F)
Head(E) = a	Tail(E) = ((b,c))
Head(((b,c))) = (b,c)	Tail (((b,c))) = ( )
Head((b,c)) = b	Tail((b,c)) = (c)
Head((c)) = c	Tail((c)) = ( )

广义表的定义

基本操作：

{结构初始化}

InitGList(&L);

操作结果：创建空的广义表 L。

CreateGList(&L, S);

初始条件：S是广义表的书写形式串。

操作结果：由S创建广义表 L。

CopyGList(&T, L);

初始条件：广义表 L 存在。

操作结果：由广义表 L 复制得到广义表 T。

{销毁结构}

DestroyGList(&L);

初始条件：广义表 L 存在。

操作结果：销毁广义表 L。

广义表的定义

{引用型操作}

GListLength(L);

初始条件：广义表 L 存在。

操作结果：求广义表 L 的长度，即元素个数。

GListDepth(L);

初始条件：广义表 L 存在。

操作结果：求广义表 L 的深度。

GListEmpty(L);

初始条件：广义表 L 存在。

操作结果：判定广义表 L 是否为空表。

GetHead(L);

初始条件：广义表 L 存在且非空。

操作结果：返回广义表 L 的表头。

GetTail(L);

初始条件：广义表 L 存在且非空。

操作结果：返回广义表 L 的表尾。

广义表的定义

{加工型操作}

InsertFirst_GL(**&**L, e);

初始条件：广义表 L 存在。

操作结果：插入元素 e 作为广义表 L 的第一个元素。

DeleteFirst_GL(**&**L, **&**e);

初始条件：广义表 L 存在。

操作结果：删除广义表 L 中第一个元素，并用 e 返回其值。

Traverse_GL(L, Visit());

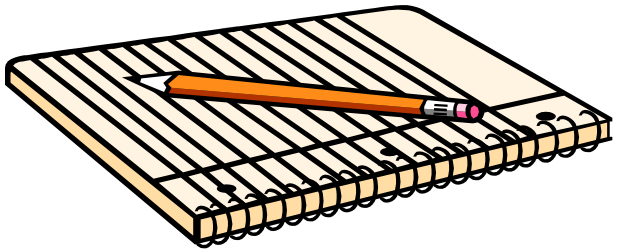
初始条件：广义表 L 存在。

操作结果：遍历广义表 L，用函数 Visit 处理每个元素。

} ADT GList

广义表的存储结构

- 5.1 数组的类型定义
- 5.2 数组的顺序存储表示
- 5.3 矩阵的压缩存储
- 5.4 广义表的定义
- 5.5 广义表的存储结构



广义表的存储结构

- 由于广义表中的数据元素可以是原子，也可以是广义表，显然**难以用顺序存储结构表示**之，并且为了在存储结构中便于分辨原子和子表，令表示广义表的链表中的结点为“异构”结点，结点中设有一个“标志域tag”， 并约定 tag=0 表示原子结点，tag=1 表示表结点。原子结点中的 data 域存储原子，表结点中指针域的两个值分别指向表头和表尾。
- 空表：ls=Nil
- 非空表：ls指向表节点

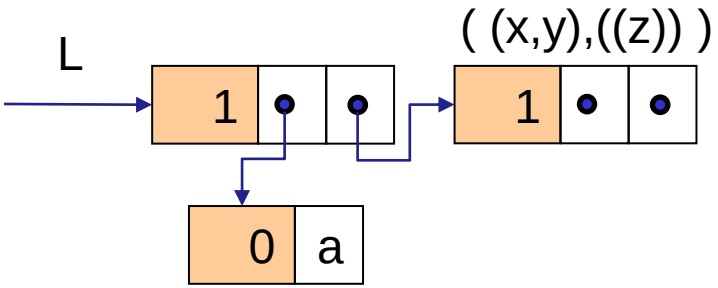
广义表的存储结构

➔ // 广义表的存储表示

```
typedef enum {ATOM=0, LIST=1} ElemTag;        // ATOM(=0)标志原子，
                                                 LIST(=1)标志子表
typedef struct GLNode {
    ElemTag tag;              // 公共部分，用于区分原子结点和表结点
    union {                    // 原子结点和表结点的联合部分
        AtomType data;        // data是原子结点的值域，AtomType由用户
                                 定义
        struct {struct GLNode *hp, *tp;} ptr;
                  // ptr是表结点的指针域，ptr.hp和ptr.tp分别指向表头和
                     表尾
    }
} *GList;                 // 广义表类型
```

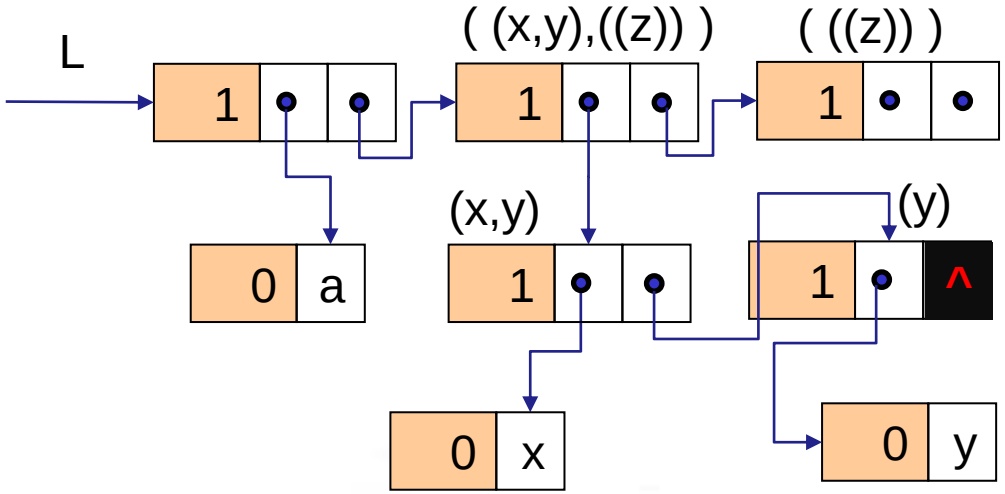
广义表的存储结构

- 例如，广义表 L=(a,(x,y),((z))) 的存储结构如下图所示。它可由对L进行表头和表尾的分析得到。
- Head(L)=a;Tail(L)=((x,y),((z)))



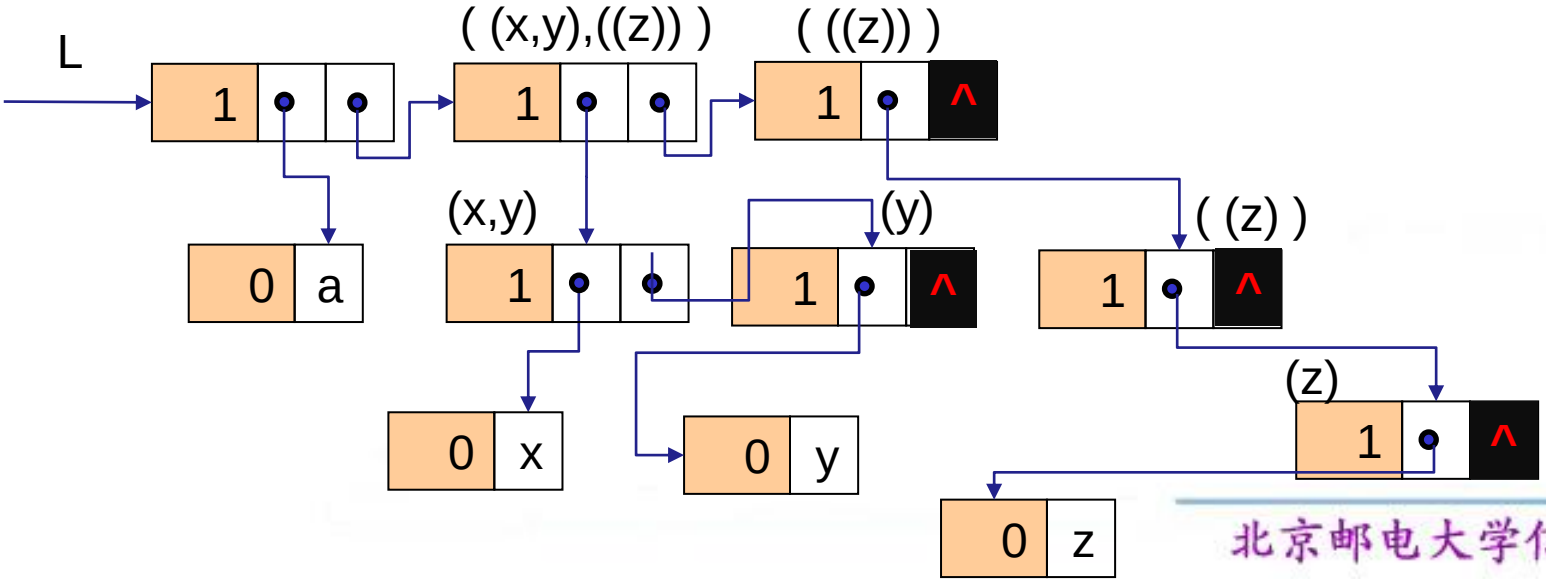
广义表的存储结构

- 例如，广义表 L=(a,(x,y),((z))) 的存储结构如下图所示。它可由对L进行表头和表尾的分析得到。
- Head(((x,y),((z))))=(x,y);Tail(((x,y),((z))))=(((z)))
- Head((x,y))=x;Tail((x,y))=(y)
- Head((y))=y;Tail((y))=()



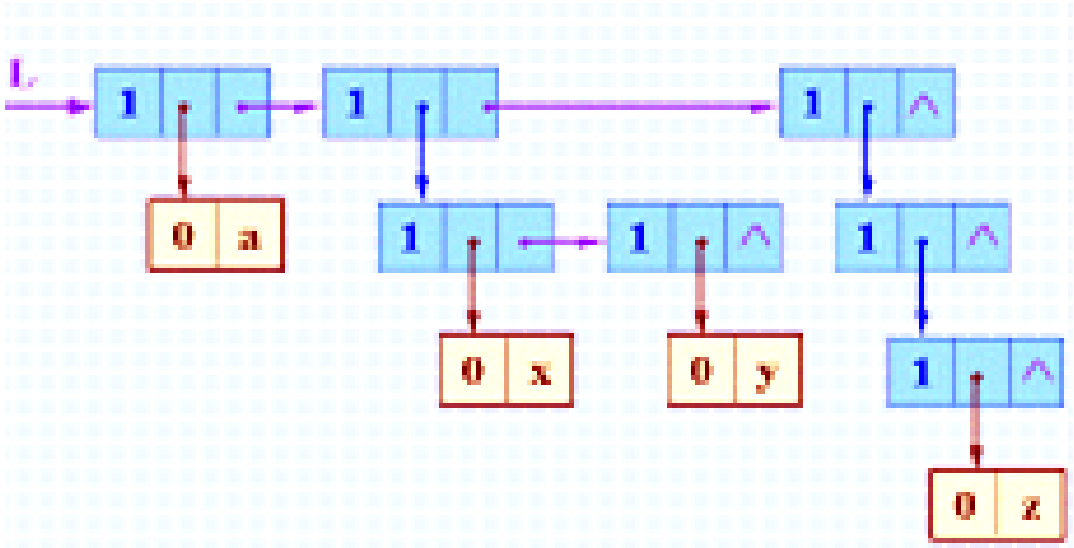
广义表的存储结构

- 例如，广义表 L=(a,(x,y),((z))) 的存储结构如下图所示。它可由对L进行表头和表尾的分析得到。
- Head((((z))))=((z));Tail((((z))))=()
- Head(((z)))=(z);Tail(((z)))=()
- Head((z))=z;Tail((y))=()



广义表的存储结构

- 例如，广义表 L=(a,(x,y),((z))) 的存储结构如下图所示。它可由对L进行表头和表尾的分析得到。



- 演示8-2-1.swf

本章小结

- 通过这一章的学习，主要是了解**数组的类型定义**及其在高级语言中实现的方法。数组作为一种数据类型，它的特点是一种多维的线性结构，并只进行存取或修改某个元素的值，因此它只需要采用顺序存储结构。
- 介绍了**随机稀疏矩阵的三种表示方法**。至于在具体应用问题中采用哪一种表示法这取决于该矩阵主要进行什么样的运算。
- 从结构本身特性而言，**广义表**归属于线性结构，但实现广义表操作的算法和树的操作的算法更为相近，这正是广义表这种数据结构的特点。由于广义表是一种递归定义的线性结构，因此它兼有线性结构和层次结构的特点。

本章知识点与重点

- **知识点**

数组的类型定义、数组的存储表示、特殊矩阵的压缩存储表示方法、随机稀疏矩阵的压缩存储表示方法

- **重点和难点**

本章重点是学习数组类型的定义及其存储表示 。