# 第五章 数组和广义表

- 数组的类型定义（5.1 ）
- 数组的顺序表示和实现（5.2）
- 稀疏矩阵的压缩存储 （5.3）
- 广义表的类型定义（5.4）
- 广义表的表示方法（5.15）
- 广义表操作的递归函数（5.6）

# 4.1 数组的类型定义

数组是几乎所有程序设计语言都设定为固有类型的一种数据类型。从线性结构看，数组类型可以看作是数据元素本身还是线性结构的一个数据结构。n维数组的特点是每一个数据元素受n个线性关系的约束，在每个关系中都有唯一的直接前驱和后继。

ADT Array {

数组一旦被定义，其维数和阶数都不再改变

数据对象：

$D=\{a_{j1,j2,...,ji,jn} \mid a_{j1,j2,...,ji,jn} \in ElemeSet,$

$j_i=0,...,b_i-1,\ i=1,2,..,n$；n为数组的维数，

$b_i$ 是数组第i维阶数(长度)；$j_i$ 是数组元素第i维下标}

数据关系：

$R=\{R1, R2, ..., Rn\}$

$Ri=\{<a_{j1,...ji,...jn}, a_{j1,...ji+1,...jn}> \mid 0 \le j_k \le b_k-1,$

$1 \le k \le n$ 且$k \ne i,\ 0 \le j_i \le b_i-2, i=2,...,n\}$

基本操作：

} ADT Array;

# 基本操作：

**InitArray(&A, n, $bound_1$, ..., $bound_n$);**

**DestroyArray(&A);**

**Value(A, &e, $index_1$, ..., $index_n$);**

**Assign(&A, e, $index_1$, ..., $index_n$);**

# 5.2 数组的顺序表示和实现

## 数组类型特点：

一旦建立了数组，结构中的数据元素个数和元素之间的关系固定。因此，采用**顺序存储结构表示数组是自然**；**用顺序存储表示多维数组，就必须按某种次序将数组元素排列到一个序列中。**

## 有两种顺序映象的方式：

- 以行序为主序;
- 以列序为主序;

例如：

| | | |
|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ |
| $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ |

以行为先

| | | | | | |
|---|---|---|---|---|---|
| $a_{0,0}$ | $a_{0,1}$ | $a_{0,2}$ | $a_{1,0}$ | $a_{1,1}$ | $a_{1,2}$ |

以列为先

| | | | | | |
|---|---|---|---|---|---|
| $a_{0,0}$ | $a_{1,0}$ | $a_{0,1}$ | $a_{1,1}$ | $a_{0,2}$ | $a_{1,2}$ |

一旦确定数组元素的下标，我们就可以根据数组的维数和各维阶数以及存储顺序确定该元素的存储位置

二维数组A中任一元素$a_{i,j}$的存储位置(以行为先)：

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

二维数组A中任一元素$a_{i,j}$的存储位置(以行为先)：

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

# n 维数组数据元素以行序为主序存储位置的映象关系

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

$$= LOC(0,0,...0) + (\sum_{i=1}^{n-1} j_i \prod_{k+i+1}^{n} b_k + j_n)L$$

$$= LOC(0,0,...0) + \sum_{i=1}^{n} c_i j_i$$

其中 $c_n = L$，$c_{i-1} = b_i \times c_i$，$1 < i \le n$

称为 n 维数组的映象函数。数组元素的存储位置是其下标($j_1$, $j_2$, ⋯$j_n$)的线性函数。一旦确定了数组的各维的阶数，$c_i$就是常数。所以存取数组中任一元素的时间相等。我们称具有这一特点的存储结构为随机存储结构。

# 5.3 矩阵的压缩存储（特殊矩阵）

- 特殊矩阵是指非零元素或零元素的分布有一定规律的矩阵。
- 假若值相同的元素或者零元素在矩阵中的分布有一定规律，则我们称此类矩阵为**特殊矩阵**；反之，称为**稀疏矩阵**。
- 特殊矩阵的压缩存储主要是针对阶数很高的特殊矩阵。为节省存储空间，对可以不存储的元素，如零元素或对称元素，不再存储。

  - 对称矩阵
  - 三对角矩阵

# 对称矩阵

$$A=\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

- 对称矩阵中的元素关于主对角线对称，$a_{ij}=a_{ji}$，$0\leqslant i,j\leqslant n\text{-}1$
- 为节约存储，只存对角线及对角线以上的元素，或者只存对角线或对角线以下的元素。前者称为上三角矩阵，后者称为下三角矩阵。

# 行序为主序的下三角的存储

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1n-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2n-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} \end{bmatrix}$$

下三角矩阵

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | n(n+1)/2-1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| B | $a_{00}$ | $a_{10}$ | $a_{11}$ | $a_{20}$ | $a_{21}$ | $a_{22}$ | $a_{30}$ | $a_{31}$ | $a_{32}$ | …… | $a_{n-1n-1}$ |

对任一组下标(i，j)，都能找到相应的存储位置k

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

- 若已知某矩阵元素位于数组 B 的第 $k$个位置（$k \geq 0$），可寻找满足

$$i(i+1)/2 \leq k < (i+1)*(i+2)/2$$

的 ***i***, 此即为该元素的行号。

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

此即为该元素的列号

# 对角矩阵

$$A=\begin{bmatrix} a_{00} & a_{01} & 0 & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & a_{n-2n-3} & a_{n-2n-2} & a_{n-2n-1} \\ 0 & 0 & 0 & 0 & a_{n-1n-2} & a_{n-1n-1} \end{bmatrix}$$

三对角矩阵

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| B | $a_{00}$ | $a_{01}$ | $a_{10}$ | $a_{11}$ | $a_{12}$ | $a_{21}$ | $a_{22}$ | $a_{23}$ | … | $a_{n-1n-2}$ | $a_{n-1n-1}$ |

- 三对角矩阵中除主对角线及在主对角线上 下最临近的两条对角线上的元素外，所有其它元素均为0。总共有$3n-2$个非零元素。
- 在三条对角线上的元素$a_{ij}$满足

$$0 \le i \le n-1, \quad i-1 \le j \le i+1$$

- 在一维数组 B 中 A[$i$][$j$] 在第 $i$ 行，它前面有 $3*i-1$ 个非零元素，在本行中第 $j$ 列前面有 $j-i+1$ 个，所以元素 A[$i$][$j$] 在 B 中位置为 $k = 2*i + j$。
- 若已知三对角矩阵中某元素 A[$i$][$j$] 在数组 B[ ] 存放于第 $k$ 个位置，则有

$$i = \lfloor (k+1)/3 \rfloor$$

$$j = k - 2 * i$$

# 稀疏矩阵

假设 m 行 n 列的矩阵含 s个非零元素，令 e = s/(m*n)，称为**稀疏因子**通常认为 e ≤ 0.05 的矩阵为稀疏矩阵

$$A_{6\times7}=\begin{pmatrix}0&0&0&22&0&0&15\\0&11&0&0&0&17&0\\0&0&0&-6&0&0&0\\0&0&0&0&0&39&0\\91&0&0&0&0&0&0\\0&0&28&0&0&0&0\end{pmatrix}$$

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**:

1) 零值元素占了很大空间；

2) 计算中进行了很多和零值的运算。

如遇除法，则需判别除数是否为零；

**解决问题的原则:**

在存储稀疏矩阵时，为节省存储空间，应只存储非零元素。但虽然非零元素的分布一般没有规律，元素之间的关系是存在的，所以存储时必须保持元素之间的关系。

每一个三元组 $(i, j, a_{ij})$ 唯一确定了矩阵A的一个非零元素。因此，稀疏矩阵可由表示非零元的一系列三元组及其行列数唯一确定。

基于三元组表示的稀疏矩阵的压缩存储方法

一、三元组顺序表

二、行逻辑联接的顺序表

三、 十字链表

# 一、三元组顺序表

```
#define MAXSIZE 12500
typedef struct {
    int i, j;       //该非零元的行下标和列下标
    ElemType e;     // 该非零元的值
} Triple; // 三元组类型


typedef struct {
    Triple data[MAXSIZE + 1];
    int    mu, nu, tu; // 矩阵的行数、列数和非零元素个数
} TSMatrix; // 稀疏矩阵类型
```

## 三元组表表示的稀疏矩阵

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

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

# 用三元组表表示的稀疏矩阵及其转置

## 原矩阵三元组表

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

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

## 转置矩阵三元组表

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

| | 行 (row) | 列 (col) | 值 (value) |
|---|---|---|---|
| [0] | 0 | 4 | 91 |
| [1] | 1 | 1 | 11 |
| [2] | 2 | 5 | 28 |
| [3] | 3 | 0 | 22 |
| [4] | 3 | 2 | -6 |
| [5] | 5 | 1 | 17 |
| [6] | 5 | 3 | 39 |
| [7] | 6 | 0 | 16 |

# 稀疏矩阵转置算法思想

转置矩阵：a 矩阵的行列阶数互换　b 将每个矩阵元素的行、列号互换 c 按次序排定转置矩阵的元素

```
Status TransposeSmatrix(TSMarix M,   TSMatrix &T) {
    //求M矩阵的转置，结果由B返回
    T.mu = M.nu;  T.nu = M.mu;  T.tu = M.tu
    if (T.tu) {
       q =1;//转置矩阵的元素号
      for (col = 0; col < m.nu; col++)    //以转置矩阵的行序为先
          for (p = 0; p < M.tu; p++)     //对原矩阵按行序进行遍历
              if (M.data[p].j== col) {
                 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++;                }
    return OK
}
```

原矩阵是以行序为先的，保证了扫描时转置矩阵的行序次序。

设矩阵三元组表总共有 $t$ 项，上述算法的时间代价为 O（$n*t$）。当 $t$ 和 $m*n$ 同数量级时，算法transmatrix的时间复杂度为$O(m*n^2)$。

# 快速转置算法

- 快速转置算法的思想：预先求得原矩阵M 每一列（即T中每一行）的第一个非零元在T中 的位置，那么，对M转置扫描时，立即可以确定在转置矩阵 T 三元组表中的位置，并装入它。
- 为加速转置速度，建立辅助数组 num和 cpot：
  - num[col]：记录矩阵转置前各列（即转置矩阵各行）非零元素个数；
  - cpot[col]：记录各列非零元素在转置三元组表中开始存放位置。

```
• for (col=1; col<=M.nu; ++col)  num[col] = 0;
  for (t=1; t<=M.tu; ++t)  ++num[M.data[t].j];
• cpot[1] = 1;
  for (col=2; col<=M.nu; ++col)
        cpot[col] = cpot[col-1] + num[col-1];
```

例：

| | | |
|---|---|---|
| 1 | 2 | 15 |
| 1 | 5 | -5 |
| 2 | 2 | -7 |
| 3 | 1 | 36 |
| 3 | 4 | 28 |

| col | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **Num[col]** | 1 | 2 | 0 | 1 | 1 |
| **Cpot[col]** | **1** | **2** | **4** | **4** | **5** |

```
Status FastTransposeSMatrix(TSMatrix M, TSMatrix &T){
    T.mu = M.nu;  T.nu = M.mu;  T.tu = M.tu;
    if (T.tu) {
      for (col=1; col<=M.nu; ++col)  num[col] = 0;
      for (t=1; t<=M.tu; ++t)  ++num[M.data[t].j];
      cpot[1] = 1;
      for (col=2; col<=M.nu; ++col)
          cpot[col] = cpot[col-1] + num[col-1];
      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 = M.data[p].i;
            T.data[q].e = M.data[p].e;      ++cpot[col]
      }
    } // if
    return OK;
} // FastTransposeSMatrix
```

时间复杂度为: O(M.nu+M.tu)

# 二、行逻辑联接的顺序表

- 三元组顺序表又称**有序的双下标法**，它的特点是，非零元在表中按行序有序存储，因此**便于进行依行顺序处理的矩阵运算**。然而，若需随机存取某一行中的非零元，则需从头开始进行查找。

- 修改前述的稀疏矩阵的结构定义，增加一个数据成员rpos，指示各行第一个非零元素的位置。

```
#define  MAXMN  500
typedef struct {
        Triple  data[MAXSIZE + 1];
        int     rpos[MAXMN + 1]; //各行第一个非零元的位置表
        int     mu, nu, tu;
} RLSMatrix;  // 行逻辑链接顺序表类型
```

例如：两个稀疏矩阵相乘（$Q=M_{m1xn1}\times N_{n1xn2}$）

矩阵乘法的精典算法:

```
for (i=1; i<=m1; ++i)
    for (j=1; j<=n2; ++j) {
        for (k=1; k<=n1; ++k)
            Q[i][j] += M[i][k] * N[k][j];
    }
```

例如：

| | | |
|---|---|---|
| 1 | 2 | 2 |
| 1 | 5 | 3 |
| 2 | 2 | -1 |
| 2 | 3 | 5 |
| 3 | 1 | 4 |
| 3 | 4 | 7 |
| 3 | 5 | 6 |
| 4 | 3 | -3 |

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

矩阵乘法的精典算法:

```
for (i=1; i<=m1; ++i)
    for (j=1; j<=n2; ++j) {
        for (k=1; k<=n1; ++k)
            Q[i][j] += M[i][k] * N[k][j];
    }
```

其时间复杂度为: O(m1×n2×n1)

N是以列序访问的。

矩阵乘法的精典算法:

```
for (i=1; i<=m1; ++i)
    for (j=1; j<=n1; ++j) {
        for (k=1; k<=n2; ++k)
            Q[i][k] += M[i][j] * N[j][k];
    }
```

其时间复杂度为: O(m1×n1×n2)

N是以行序访问的。

```
Status MultSMatrix(RLSMatrix M, RLSMatrix N, RLSMatrix &Q) {
  if (M.nu != N.mu) return ERROR;
  Q.mu = M.mu; Q.nu = N.nu; Q.tu = 0;
  if (M.tu*N.tu != 0) {  // Q是非零矩阵
    for (arow=1; arow<=M.mu; ++arow) {
      // 处理M的每一行
      ctemp[    ] = 0;              // 当前行各元素累加器清零
      Q.rpos[arow] = Q.tu+1;
      for (p=M.rpos[arow]; p<M.rpos[arow+1];++p) {
        //对当前行中每一个非零元
        brow=M.data[p].j;
        if (brow < N.nu )  t = N.rpos[brow+1];
        else  { t = N.tu+1 }
        for (q=N.rpos[brow];  q< t;  ++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
      } // 求得Q中第crow( =arow)行的非零元
      for (ccol=1; ccol<=Q.nu; ++ccol)
        if (ctemp[ccol]) {
          if (++Q.tu > MAXSIZE) return ERROR;
          Q.data[Q.tu] = {arow, ccol, ctemp[ccol]};
        } // if
    } // for arow
  } // if
  return OK;
} // MultSMatrix
```

# 分析上述算法的时间复杂度

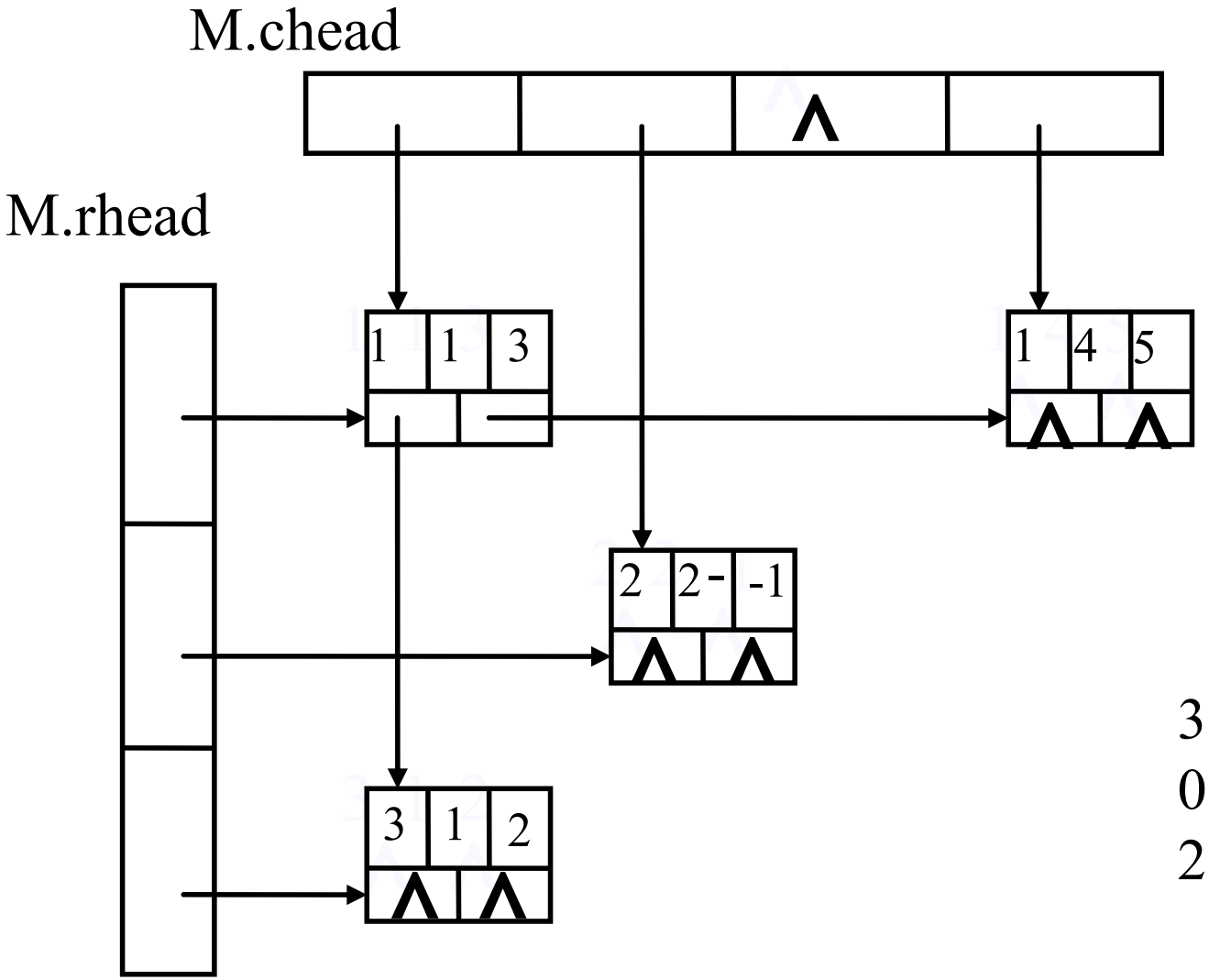
累加器ctemp初始化的时间复杂度为O(M.mu×N.nu)，
求Q的所有非零元的时间复杂度为O(M.tu×N.tu/N.mu)，
进行压缩存储的时间复杂度为O(M.mu×N.nu)，
总的时间复杂度就是O(M.mu×N.nu+M.tu×N.tu/N.mu)。

若M是m行n列的稀疏矩阵，N是n行p列的稀疏矩阵，
则M中非零元的个数 $M.tu = \delta_M \times m \times n$，
N中非零元的个数 $N.tu = \delta_N \times n \times p$，
相乘算法的时间复杂度就是 $O(m \times p \times (1+n\delta_M\delta_N))$ ，
当$\delta_M<0.05$ 和$\delta_N<0.05$及 $n<1000$时，
相乘算法的时间复杂度就相当于 $O(m \times p)$。

以上两种稀疏矩阵的顺序存储结构实际上适合稀疏矩阵非零元素个数变化不大的情况。当矩阵的非零元个数和位置在操作中变化较大时[例如矩阵的加法A=A+B]，由于顺序存储，就需要插入操作，就不宜采用顺序存储结构来表示三元组的线性表，而是采用链式存储结构表示三元组的线性表。

# 三、 十字链表

例：




3 0 0 5
0 -1 0 0
2 0 0 0

每个非零元由一个含5个域的节点表示（i，j，e, right, down）

```
typedef struct OLNode{
    int                 i, j;
    ElemType            e;
    struct OLNODE  *right, *down
} OLNode; *Olink;

typedef struct{
     Olink   *rhead, *chead;//行和列链表头指针向量基址
     in mu, nu, tu;
} CrossList
```

例： 将矩阵B加到矩阵A上

$$a'_{ij} = a_{ij} + b_{ij},$$

$$a'_{ij} \neq 0, \text{当且仅当} \begin{cases} a_{ij} + b_{ij}; \begin{cases} =0; & \text{删除该节点;} \\ \neq 0; & \text{改变节点的}val\text{域值;} \end{cases} \\ a_{ij}; \quad \text{不对A做任何操作;} \\ b_{ij}; \quad \text{插入一个新的结点，值为} b_{ij} \end{cases}$$

整个算法过程可从矩阵的第一行起逐行进行。对每行都从行表头出发分别找到A和B在该行中的第一个非零元节点后开始比较，然后按上述4种情况分别处理。

① 若pa==NULL或者pa->j>pb->j,
② 若pa->j<pb->j,
③ 若pa->j==pb->j且pa->e+pb->e!=0,
④ 若pa->j ==pb->j且pa->e+pb->e==0

# 总结

- 了解数组的两种存储表示方法，并掌握数组在以行为主的存储结构中的地址计算方法。
- 掌握对特殊矩阵进行压缩存储时的下标变换公式。
- 了解稀疏矩阵的两类压缩存储方法的特点和适用范围，领会以三元组表示稀疏矩阵时进行矩阵运算采用的处理方法。

# 5.4 广义表 (General Lists )

是线性表的推广。

**ADT Glist {**

数据对象：$D=\{e_i \mid i=1,2,..,n;\ n\geq 0;\ e_i \in AtomSet$ 或 $e_i \in GList,$ AtomSet为某个数据对象 $\}$

数据关系：$LR=\{<e_{i-1}, e_i> \mid e_{i-1}, e_i \in D,\ 2\leq i\leq n\}$

基本操作：

**} ADT** Glist

- 广义表是递归定义的线性结构

一般情况下，广义表可以表示为 $LS(a_1, a_2, a_3, \cdots, a_n)$

- ai 是表元素，可以是广义表（称为子表），也可以是数据元素（称为原子）
- *LS* 是表名；n为表的长度。n＝0 的广义表为空表。

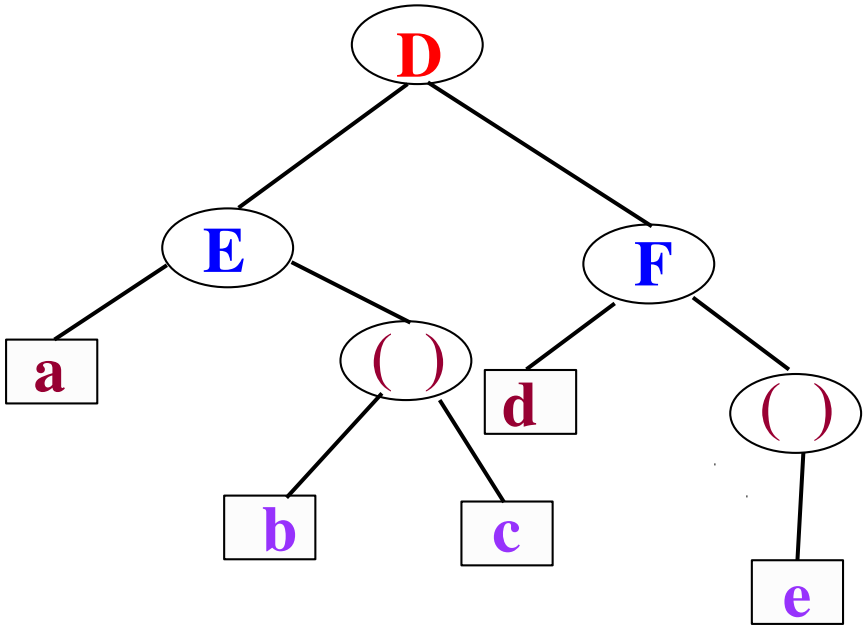
- 广义表是一个多层次的线性结构，甚至可以是递归

例如：**D=(E, F)**

其中：

**E=(a, (b, c))**

**F=(d, (e))**



其它如:

A=( )

B=(**a**, B)=(a, (a, (a, …)))

C=(**A**, D，F)

# 广义表 LS = ( α1, α2, …, αn )的结构特点:

- 广义表中的数据元素有相对次序;
- 广义表有有限的长度的，广义表的长度定义为最外层包含元素个数;
- 广义表是有深度的，广义表的深度定义为所含括弧的重数;

  注意:“原子”的深度为 0 ，不是广义表

  “空表”的深度为 1 ，长度为0。

- 广义表可以共享;
- 广义表可以是一个递归的表;

  递归表的深度是无穷值，长度是有限值。

- 任何一个非空广义表 LS = ( α1, α2, ⋯, αn)均可分解为

  表头 Head(LS) = α1 和

  表尾 Tail(LS) = ( α2, …, αn) 两部分

任何一个非空的广义表的表尾必定是一个广义表

例：

A( )　　Length(A) =0, Head(A) 和 Tail(A) 不存在

B( 6, 2 )　　Length(A) =2, Head(B) = 6, tail(B) = (2)

C( ‘*a*’, ( 5, 3, ‘*x*’ ) )

Length(C) =2, Head(C) =‘*a*’ , Tail(C) = ((5,3,’x’))

D( B, C, A )

Length(A) =3, Head(D) = B, Tail(D) = (C, A)

F( 4, F )

Length(A) =2, Head(F) = 4, Tail(F) = (F);

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

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 )** ) = ( )

# 5.5 广义表的存储结构

- 广义表的数据元素具有不同的结构，且是一个带深度的层次结构。因此难以用顺序存储结构来存放，而是采用链式存储结构。

- 广义表中元素可以为原子或者广义表，

表结点<br>原子结点 } Union类型

构造存储结构的两种分析方法:

1) 表头、表尾分析法

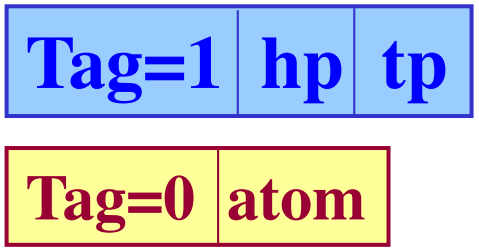
2) 子表分析法

# 1) 表头、表尾分析法

若广义表不空，则可分解为表头和表尾；反之，一对确定的表头和表尾可唯一确定广义表

空表A( )　　A=NIL

非空表A( )



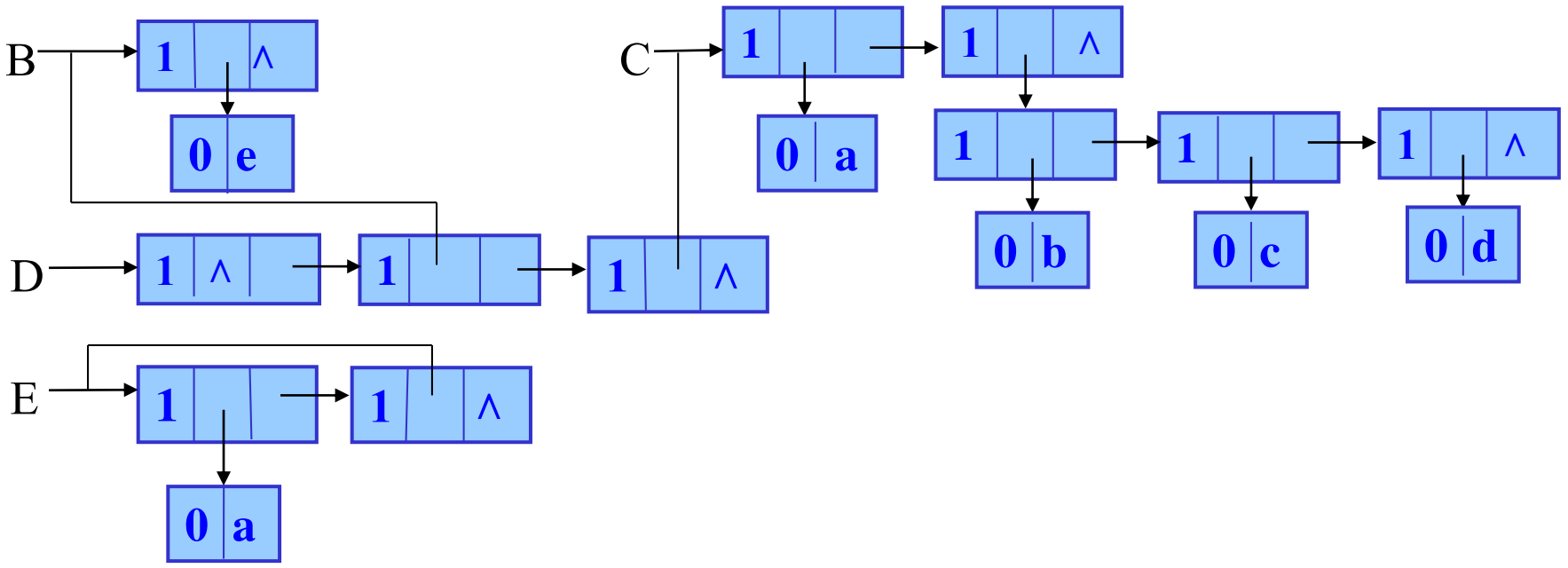
例： A=NIL

B( e )

C(a, (b, c, d))

D( A, B, C)

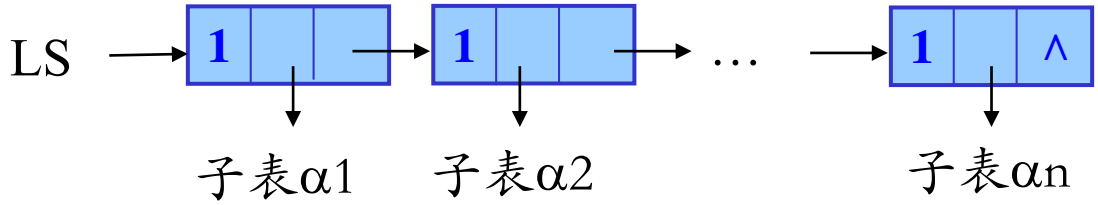
E( A, E )



# 2) 子表分析法

空表A( )　　A=NIL

非空表A( )　　LS = ( α1, α2, …, αn )




若子表为原子，则为 


否则，依次类推。

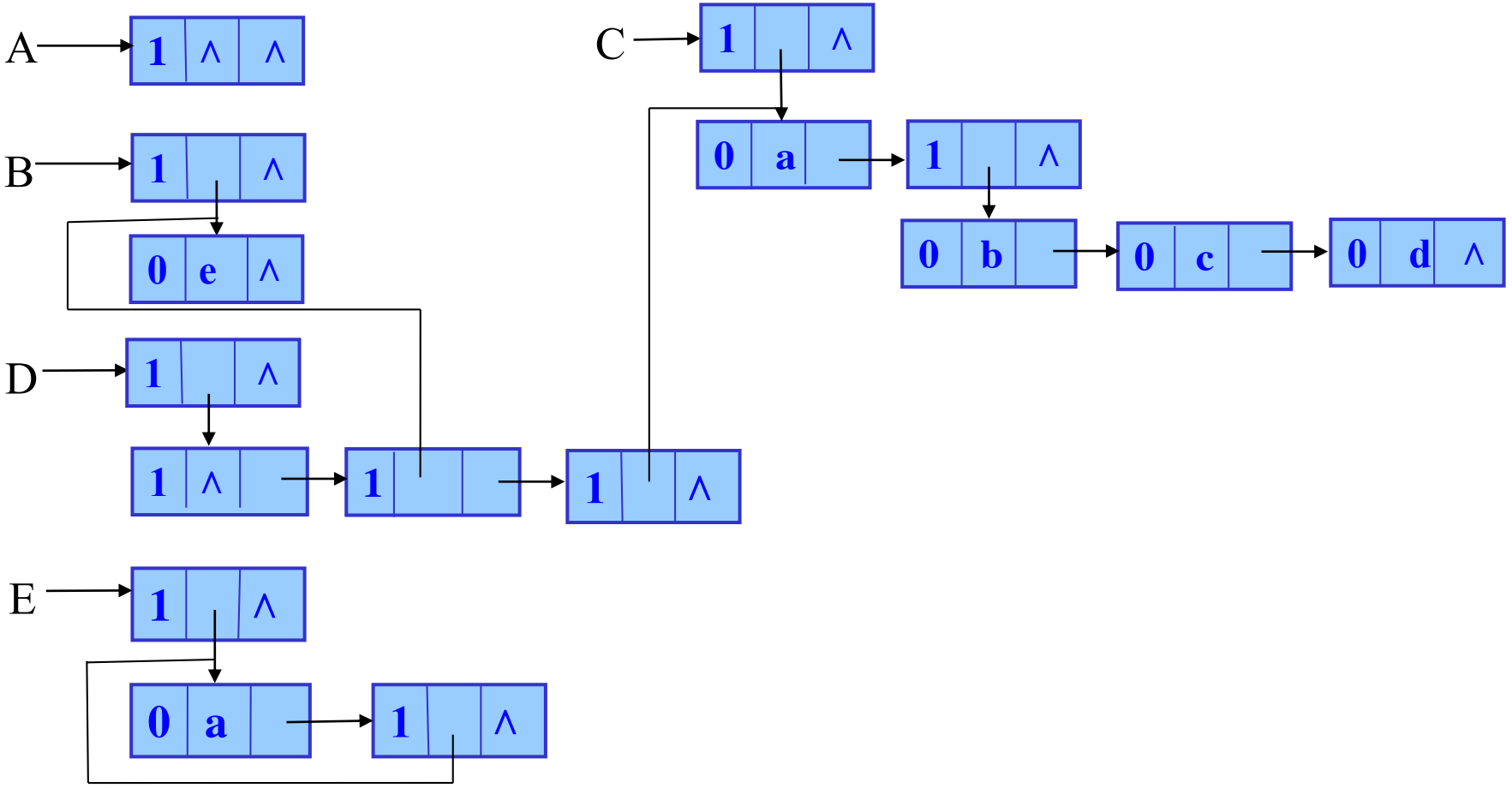
例： A( )

B( e )

C(a, (b, c, d))

D( A, B, C)

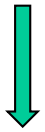
E( A, E )



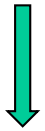
# 5.6 m元多项式的表示



$$P(x,y,z) = x^{10}y^3z^2 + 2x^6y^3z^2 + 3x^5y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz + 15$$



$$P(x,y,z) = ((x^{10} + 2x^6)y^3 + 3x^5y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z + 15$$

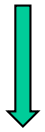


$$P(z) = A(y)z^2 + B(y)z + 15$$

$$A(y) = C(x)y^3 + D(x)y^2 \qquad B(y) = E(x)y^4 + F(x)y$$

$$C(x) = x^{10} + 2x^6 \qquad E(x) = x^4 + 6x^3$$

$$D(x) = 3x^5 \qquad F(x) = 2$$



$$P = z((A,2),(B,1),(15,0))$$

$$A = y((C,3),(D,2)) \qquad B = y((E,4),(F,1))$$

$$C = x((1,10),(2,6)) \qquad E = x((1,4),(6,3))$$

$$D = x((3,5)) \qquad F = x((2,0))$$



```
typedef struct MPNode {
    ElemTag  tag;
    int          exp; //指数域
    union {          //原子结点和表结点的联合部分
        float   coef;   //系数域
        struct GLNodex *hp;
    };
    struct MPNode *tp; //相当于线性链表的next
} *Mplist;      //m元多项式广义表类型定义
```

$P = z((A,2),(B,1),(15,0))$

$A = y((C,3),(D,2)$

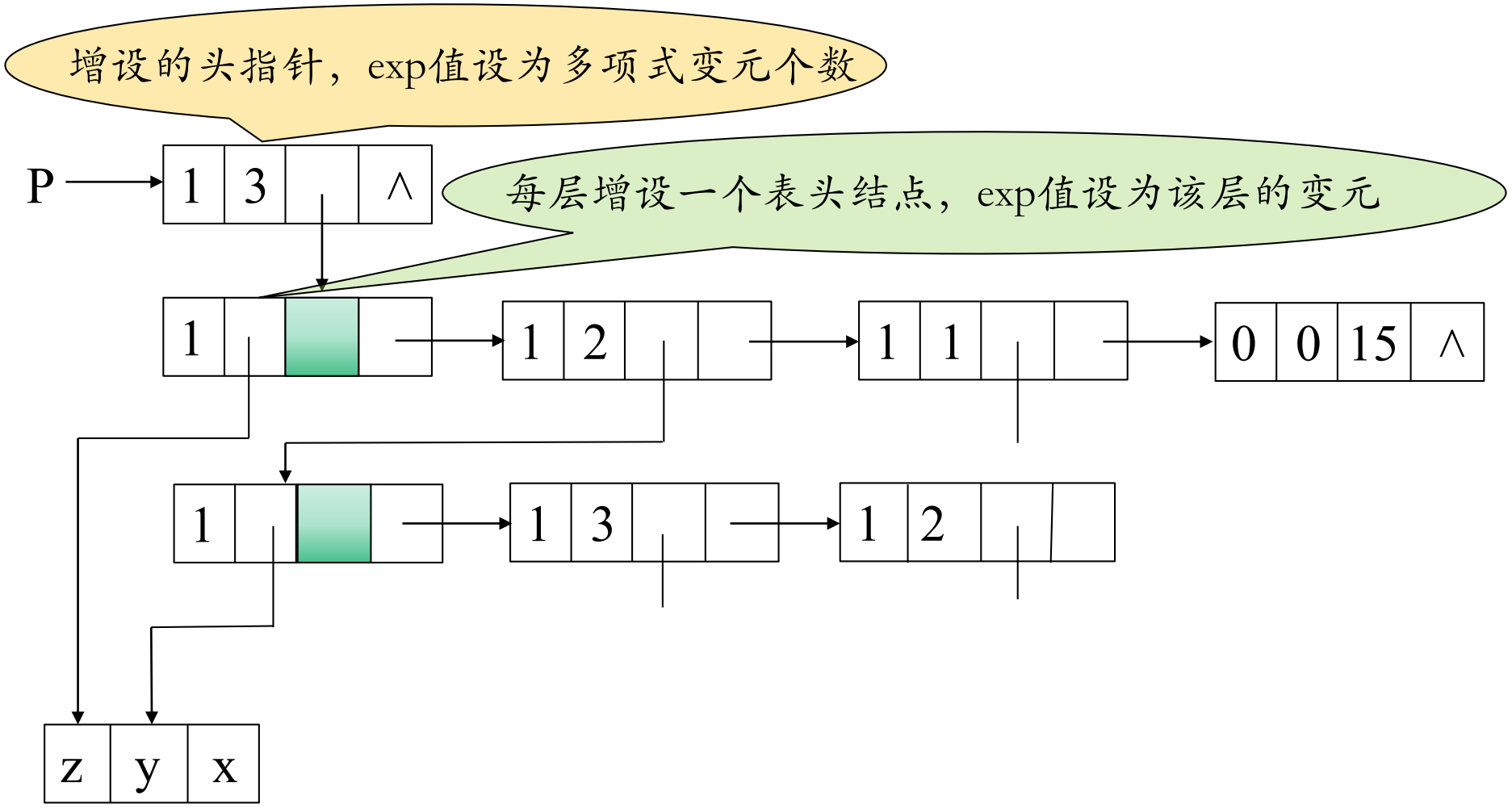
$B = y((E,4),(F,1))$

$C = x((1,10),(2,6))$

$E = x((1,4),(6,3))$

$D = x((3,5))$

$F = x((2,0))$



# 5.6 广义表操作的递归函数



## 递归函数

一个含直接或者间接调用自身函数的函数被称为递归函数。

递归是算法减而治之、分而治之思想的重要体现

- 为求解一个大规模问题，可以将其划分为多个相互不关联的子问题，且这些子问题的规模缩减。分别求子问题，再用某种方法把它们组合成原来问题的解。若子问题的规模仍然相当大，则可以反复使用分治法，直至最后所分得的子问题足够小。

- 在利用分治法求解问题时，所得的子问题的类型如果和原问题相同，很自然就导致了递归求解。

例如：梵塔问题：Hanoi(n, x, y, z)

可以分为三个子问题：

① Hanoi(n-1, x, z, y); ——递归

② move(x, n, z);

③ Hanoi(n-1, y, x, z); ——递归

例如：计算任意n个整数之和

```
int SumI ( int A[], int n) {
    int sum=0;      //o(1)
        for (i=0;i<n; i++)     //o(n)
            sum +=A[n];     //o(1)
   return(sum); //o(1)
}
```

```
int Sum (int A[],  int n) {
        return
            (n<1)? 0:sum(A, n-1)+A[n-1];
}
```

- 有效的递归调用函数(或过程) 必须满足一下两个条件:

  ① 在每次调用自己时，必须是（在某种意义上）更接近于解;

  ② 必须有一个终止处理或计算的准则。

例如:

```
int bad(int n) {
    if ( n == 0)    return 0;
    else
        return bad(n/3 + 1) + n-1;
}
```

广义表从结构上可以分解为

广义表=表头+表尾

或者

广义表=子表1+子表2+…+子表n

求广义表的深度

复制广义表

创建广义表的存储结构

# 例一： 求广义表的深度

将广义表$LS(a_1, a_2, a_3, \cdots, a_n)$分解成 n 个子表，分别(递归)求得每个子表的深度，

基本项:

depth(LS)=1 当LS为空表

depth(LS)=0 当LS为原子

递归项：

$$depth(LS)=1+\max_{1\le i\le n}\{depth(a_i)\}$$

```
int GlistDepth(Glist L) {
// 返回指针L所指的广义表的深度
    if (!L) return 1;
    if (L->tag == ATOM) return 0;
    for (max=0, pp=L; pp; pp=pp->ptr.tp){
      dep = GlistDepth(pp->ptr.hp);
      if (dep > max) max = dep;
    }
    return max + 1;
} // GlistDepth
```

# 例二 复制广义表

将广义表分解成表头和表尾两部分，分别(递归)复制求得新的表头和表尾，新的广义表由新的表头和表尾构成。

基本项:

InitGList(NEWLS) 当LS为空表

递归项:

COPY(表头) {复制表头}

COPY(表尾) {复制表尾}

```
Status CopyGList(Glist &T, Glist L) {
  if (!L) T = NULL; // 复制空表
  else {
    if ( !(T = new GLNode) )
      exit(OVERFLOW);
    T->tag = L->tag;
    if (L->tag == ATOM)
      T->atom = L->atom;
    else { 分别复制表头和表尾 }
  } // else
  return OK;
} // CopyGList
```

```
CopyGList(T->ptr.hp, L->ptr.hp);
   // 复制求得表头T->ptr.hp的一个副本L->ptr.hp
CopyGList(T->ptr.tp, L->ptr.tp);
   // 复制求得表尾T->ptr.tp 的一个副本L->ptr.tp
```

# 例三 创建广义表的存储结构

假设以字符串 S = '($\alpha_1, \alpha_2, \cdots, \alpha_n$)' 的形式定义广义表 L，建立相应的存储结构。

基本项：置空广义表　　当 S 为空串

建立原子结点的子表　　当S为单字符串时

递归项：

假设sub为脱去S最外层括弧的子串，为‘$\alpha_1, \alpha_2, \cdots, \alpha_n$’。为每个$\alpha_1$构造一个表结点 *L, 对应 创建子广义表 的L->ptr.hp，并将上一个子表的尾指针指向当前新建立的表结点。依次类推，直至剩余串为空串止。

```
Status  sever(Sstring &str, Sstring &&hstr){
//将非空串str分割为两部分：hstr为第一个最外层‘，’之前的子串，str为之后
  的子串
  n = StrLength(str);   i=0;   k=0;
  do{
       ++i;
       SubString(ch, str, i, 1);
       if ( ch==‘（’)  ++k;
       else ( ch==‘)’ )   --k;
  }while ( i<n && (ch!=‘,’ || k!=0) );
  if (i<n){
      SubString(hstr, str, 1, i-1);
      SubString(str, str, i+1, n-i);
  }
  else {  StrCopy(hstr, str);   ClearString(str); }
}//sever
```

```
void CreateGList(Glist &L, String S) {
 if (空串) L = NULL; // 创建空表
 else {
   L = new GLNode; // 生成表结点
   if (Strlength(L)==1){建立单原子表}
   else{
       L->tag=List;   p=L;
       // 脱去串 S 的外层括弧
       sub=SubString(S,2,StrLength(S)-1);
       由sub中所含n个子串建立n个子表;
    } // else
  }//else
}
```

```
do {
    sever(sub, hsub); //分离出子表串hsub=αi
    CreateGList(p->ptr.hp, hsub); q=p:
    if (!StrEmpty(sub) { //余下的表不为空
      p=new(sizeof(GLNode));
      // 建下一个子表的表结点*(p->ptr.tp)
      p->tag = LIST;    q->ptr.tp=p;
    }
} while (!StrEmpty(sub));
q->ptr.tp = NULL;    // 表尾为空表
```

# 总结

- 掌握广义表的结构特点及其存储表示方法，读者可根据自己的习惯熟练掌握任意一种结构的链表，学会对非空广义表进行分解的两种分析方法：即可将一个非空广义表分解为表头和表尾两部分或者分解为n个子表。
- 学习利用分治法的算法设计思想编制递归算法的方法。