第四章 数组、串与广义表

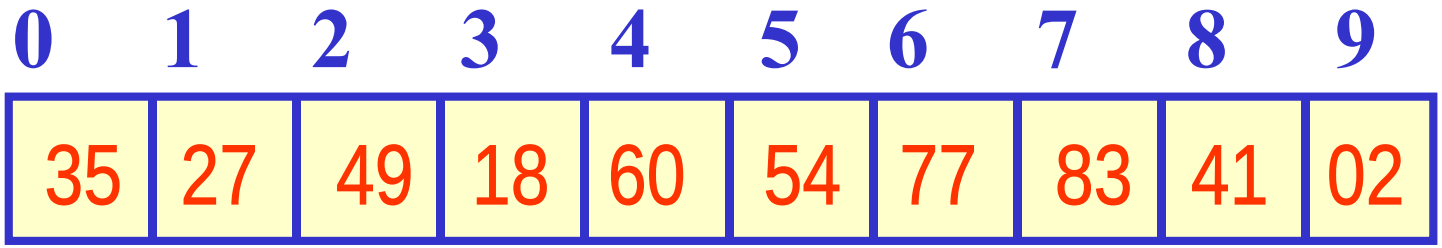
- 一维数组与多维数组
- 特殊矩阵
- 稀疏矩阵
- 字符串
- 广义表

一维数组

- 定义

  数组是相同类型的数据元素的集合，而一维数组的每个数组元素是一个序对，由下标（**index**）和值（**value**）组成。

- 一维数组的示例



- 在高级语言中的一维数组只能按元素的下标直接存取数组元素的值。

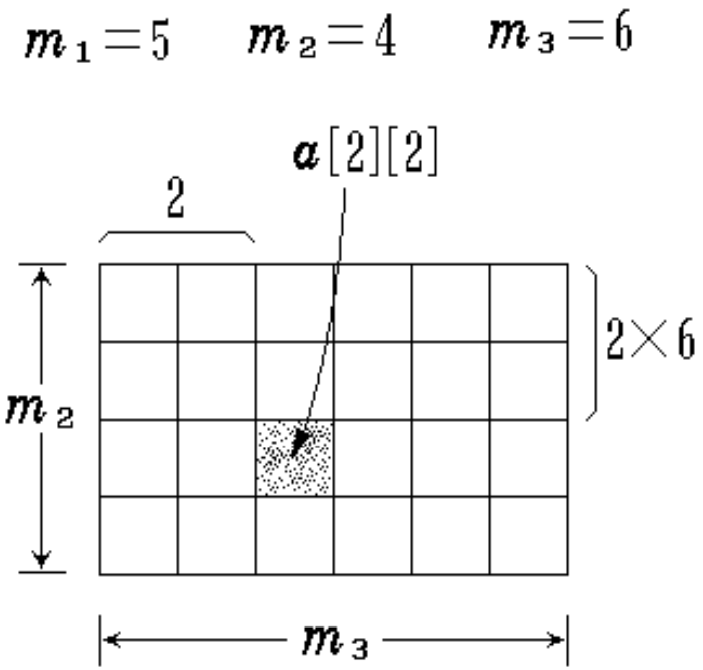
一维数组的定义和初始化

```
#include <iostream.h>
void main ( ) {
    int a[3] = { 3, 5, 7 }, *elem, i;          //静态数组
    for (i = 0; i < 3; i++)
        cout << a[i] << endl;
    elem = new int[3];                         //动态数组
    for (i = 0; i < 3; i++)
        cin >> elem[i];
    int * temp = elem;
    for (i = 0; i < 3; i++)
      { cout << *temp<< endl;  temp++; }
     delete [] elem;
}
```

多维数组

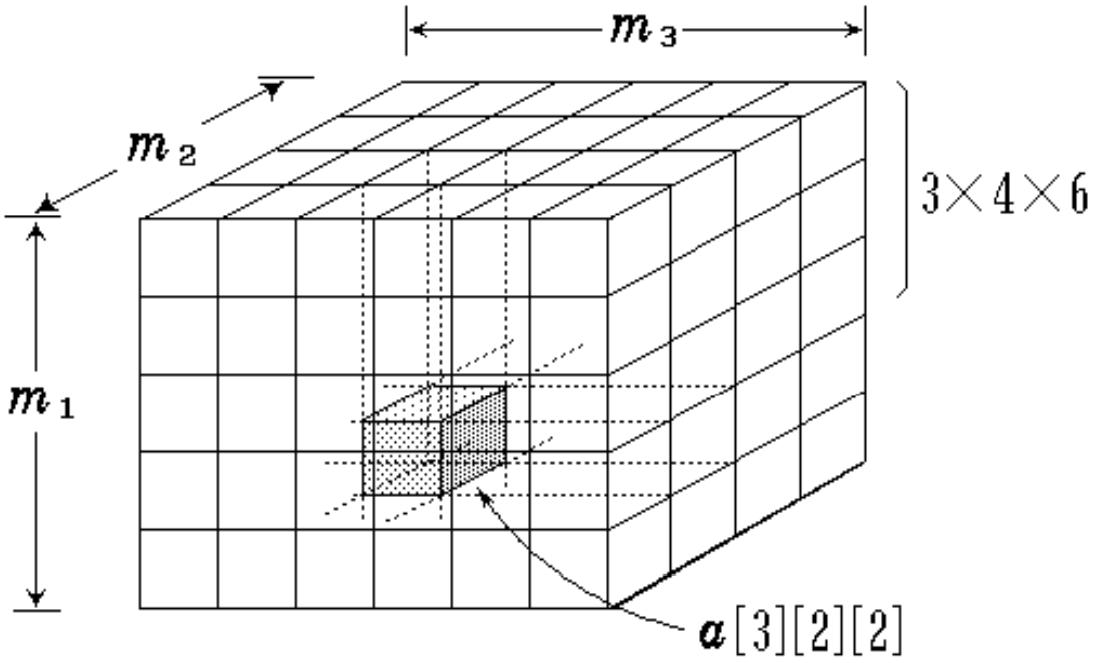
- 多维数组是一维数组的推广。
- 多维数组的特点是每一个数据元素可以有多个直接前驱和多个直接后继。
- 数组元素的下标一般具有固定的下界和上界，因此它比其他复杂的非线性结构简单。
- 例如二维数组的数组元素有两个直接前驱，两个直接后继，必须有两个下标（行、列）以标识该元素的位置。

二维数组



行向量 下标 *i*
列向量 下标 *j*

三维数组

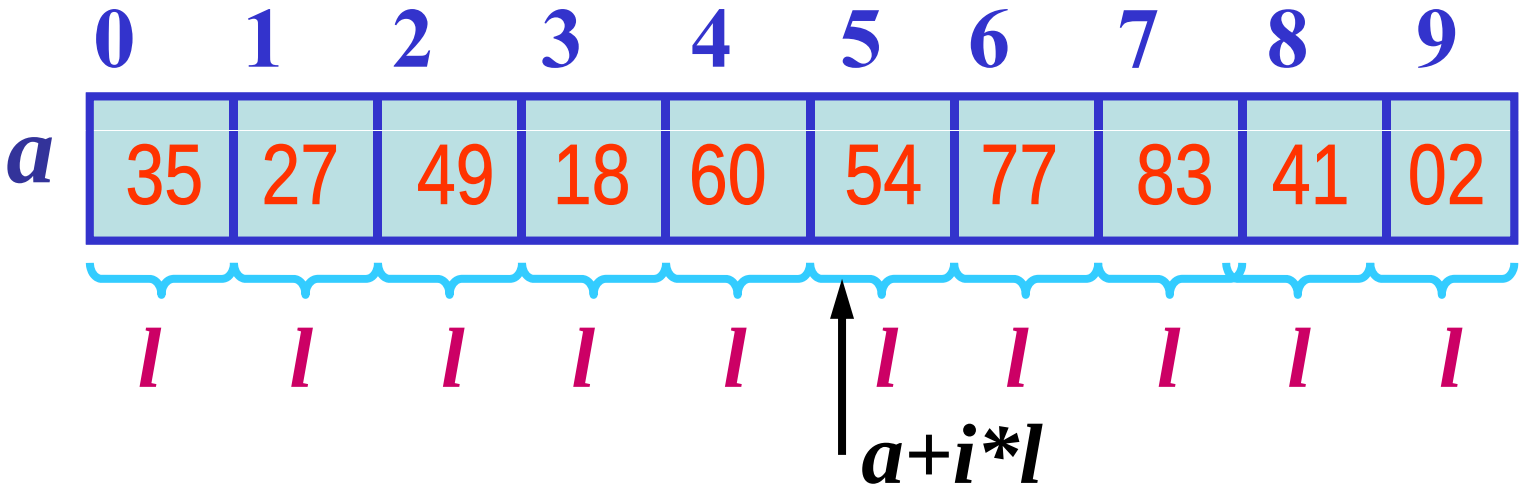


页向量 下标 *i*
行向量 下标 *j*
列向量 下标 *k*

数组的连续存储方式

一维数组

$$LOC(i)=\begin{cases} a, & i=0 \\ LOC(i-1)+l = a+i*l, & i>0 \end{cases}$$



二维数组

- 一维数组常被称为向量（Vector）。
- 二维数组 **A[m][n]** 可看成是由 **m** 个行向量组成的向量，也可看成是由 **n** 个列向量组成的向量。
- 一个二维数组类型可以定义为其分量类型为一维数组类型的一维数组类型：

```
typedef T array2[m][n];      //T为元素类型
```

等价于：

```
typedef T array1[n];         //列向量类型
typedef array1 array2[m];    //二维数组类型
```

- 同理，一个三维数组类型可以定义为其数据元素为二维数组类型的一维数组类型。
- 静态定义的数组，其维数和维界不再改变，在编译时静态分配存储空间。一旦数组空间用完则不能扩充。
- 动态定义的数组，其维界不在说明语句中显式定义，而是在程序运行中创建数组对象时通过 **new** 动态分配和初始化，在对象销毁时通过 **delete** 动态释放。
- 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列到一个序列中。

二维数组的动态定义和初始化

```
#include <iostream.h>
……
int **A;
int row =  3, col = 3;   int i, j;
A = new int *[row];
for (i = 0; i < row; i++)
    A[i] = new int [col];
for (i = 0; i < row; i++)
    for (j = 0; j < col; j++)  cin >> A[i][j];
……
```

二维数组中数组元素的顺序存放

$$a = \begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & \cdots & a[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & \cdots & a[n-1][m-1] \end{pmatrix}$$

- 行优先存放：
  设数组开始存放位置 LOC(0, 0) = a, 每个元素占用 l 个存储单元
- $$LOC(j,k) = a + (j * m + k) * l$$

$$a = \begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & \cdots & a[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & \cdots & a[n-1][m-1] \end{pmatrix}$$

列优先存放：

设数组开始存放位置 LOC(0, 0) = *a*，每个元素占用 *l* 个存储单元

$$LOC(j,k) = a + (k * n + j) * l$$

三维数组

- 各维元素个数为 m_1, m_2, m_3
- 下标为 i_1, i_2, i_3的数组元素的存储地址：

（按页 / 行 / 列存放）

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

$i_1 * m_2 * m_3$	$i_2 * m_3$	i_3
前i_1页总元素个数	第i_1页前i_2行总元素个数	第 i_2 行前 i_3 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 $i_1, i_2, i_3, \ldots, i_n$ 的数组元素的存储地址：

$$\begin{aligned} \mathrm{LOC}(i_1, i_2, \ldots, i_n) = a + \\ (i_1*m_2*m_3*\ldots*m_n + i_2*m_3*m_4*\ldots*m_n + \\ + \ldots\ldots + i_{n-1}*m_n + i_n) * l \end{aligned}$$

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

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。
- 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

对称矩阵的压缩存储

- 设有一个 **n×n** 的对称矩阵 **A**。

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

- 对称矩阵中的元素关于主对角线对称，

  $a_{ij} = a_{ji}$，　$0 \le i, j \le n-1$

- 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。

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

下三角矩阵

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

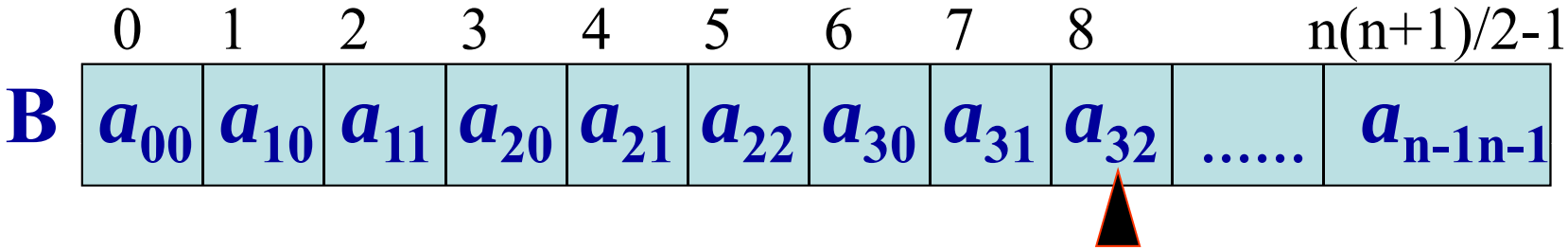
上三角矩阵

- 把它们按行存放于一个一维数组 **B** 中，称之为对称矩阵 **A** 的压缩存储方式。
- 数组 **B** 共有 **n + (n - 1) + … + 1 =**

  n*(n+1) / 2 个元素。

下三角矩阵

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



若 $i \geq j$, 数组元素A[i][j]在数组B中的存放位置为 $1 + 2 + \cdots + i + j = (i + 1) * i / 2 + j$

前i行元素总数　第i行第j个元素前元素个数

- 若 $i < j$，数组元素 A[i][j] 在矩阵的上三角部分，在数组 B 中没有存放，可以找它的对称元素A[j][i]：$= j*(j+1)/2 + i$
- 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

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

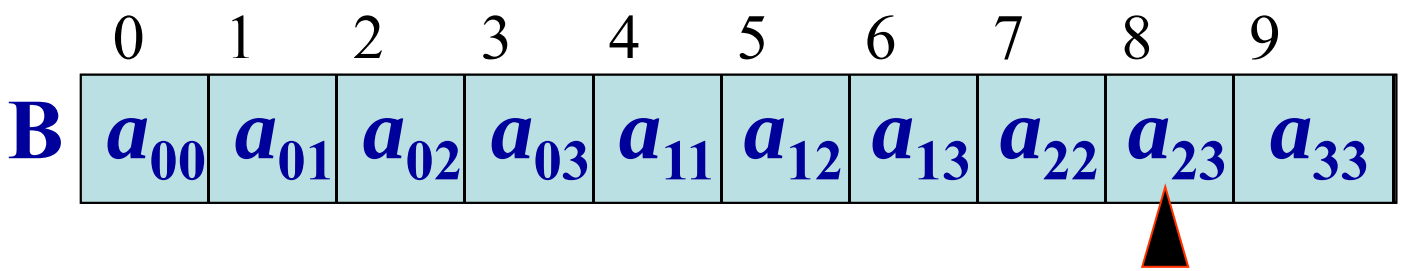
的 i, 此即为该元素的行号。

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

此即为该元素的列号。

- 例，当 $k = 8$，$3*4/2 = 6 \leq k < 4*5/2 = 10$，取 $i = 3$。则 $j = 8 - 3*4/2 = 2$。

$n=4$ $\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$ 上三角矩阵



若$i \le j$，数组元素A[i][j]在数组B中的存放位置为 $n+(n-1)+(n-2)+\cdots+(n-i+1)+j-i$

前i行元素总数　第i行第j个元素前元素个数

- 若 $i \le j$，数组元素A[*i*][*j*]在数组B中的存放位置为

$$n + (n-1) + (n-2) + \cdots + (n-i+1) + j-i =$$

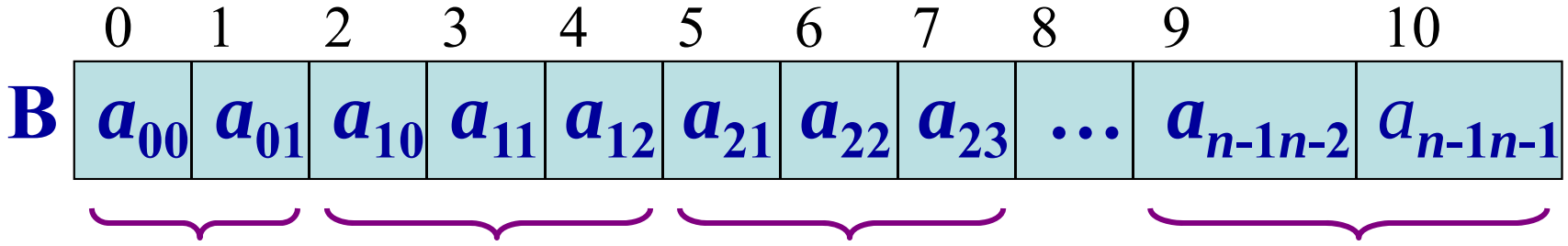
$$= (2*n-i+1) * i / 2 + j-i =$$

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

- 若$i > j$，数组元素A[*i*][*j*]在矩阵的下三角部分，在数组 B 中没有存放。因此，找它的对称元素A[*j*][*i*]。A[*j*][*i*]在数组 B 的第 $(2*n-j-1) * j / 2 + i$ 的位置中找到。

三对角矩阵的压缩存储

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



- 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。
- 将三对角矩阵A中三条对角线上的元素按行存放在一维数组 B 中，且a_{00}存放于B[0]。
- 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

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

$$j = k - 2 * i$$

- 例如，当 $k = 8$ 时，

$$i = \lfloor (8+1) / 3 \rfloor = 3,\ j = 8 - 2*3 = 2$$

当 $k = 10$ 时，

$$i = \lfloor (10+1) / 3 \rfloor = 3,\ j = 10 - 2*3 = 4$$

稀疏矩阵 (Sparse Matrix)

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

- 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$s \ll m\times n$），则称 A 为稀疏矩阵。

- 设矩阵 **A** 中有 **s** 个非零元素。令 **e = s/(m×n)**，称 **e** 为矩阵的稀疏因子。
- 有人认为 **e≤0.05** 时称之为稀疏矩阵。
- 在存储稀疏矩阵时，为节省存储空间，应只存储非零元素。但由于非零元素的分布一般没有规律，故在存储非零元素时，必须记下它所在的行和列的位置 **(*i*, *j*)**。
- 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。

稀疏矩阵的定义

```
const int drows = 6,  dcols = 7, dterms = 9;
template<class E>
struct Triple {                  //三元组
    int row, col;                //非零元素行号/列号
    E value;                     //非零元素的值
    void operator = (Triple<E>& R)      //赋值
      { row = R.row;  col = R.col;  value = R.value; }
};
```

```
template <class E>
class SparseMatrix {
  public:
    SparseMatrix (int Rw = drows, int Cl = dcols,
        int Tm = dterms);                    //构造函数
    void Transpose(SparseMatrix<E>& b);    //转置
    void Add (SparseMatrix<E>& a,
        SparseMatrix<E>& b);          //a = a+b
    void Multiply (SparseMatrix<E>& a,
        SparseMatrix<E>& b);          //a = a*b
  pvivate:
    int Rows, Cols, Terms;         //行 / 列 / 非零元素数
    Triple<E> *smArray;            //三元组表
};
```

稀疏矩阵的构造函数

```
template <class E>
SparseMatrix<E>::SparseMatrix (int Rw, int Cl, int
    Tm) {
    Rows = Rw;   Cols = Cl;   Terms = Tm;
    smArray = new Triple[Terms];          //三元组表
    if (smArray == NULL) {
        cerr <<"存储分配失败！" << endl;  exit(1);
    }
};
```

稀疏矩阵的转置

- 一个 **m×n** 的矩阵 **A**，它的转置矩阵 **B** 是一个 **n×m** 的矩阵，且 **A[*i*][*j*] = B[*j*][*i*]**。即
  - 矩阵 **A** 的行成为矩阵 **B** 的列
  - 矩阵 **A** 的列成为矩阵 **B** 的行。
- 在稀疏矩阵的三元组表中，非零矩阵元素按行存放。当行号相同时，按列号递增的顺序存放。
- 稀疏矩阵的转置运算要转化为对应三元组表的转置。

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

转置矩阵

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

	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

用三元组表表示的稀疏矩阵及其转置

原矩阵三元组表

	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

	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

- 设矩阵列数为 **Cols**，对矩阵三元组表扫描 **Cols** 次。

- 第 **k** 次扫描找寻所有列号为 **k** 的项，将其行号变列号、列号变行号，顺次存于转置矩阵三元组表。

稀疏矩阵的转置

```
template <class E>
void SparseMatrix<E> ::
Transpose (SparseMatrix<E>& B) {
//转置this矩阵，转置结果由B返回
    B.Rows = Cols;   B.Cols = Rows;
    B.Terms = Terms;
      //转置矩阵的列数,行数和非零元素个数
    if (Terms > 0) {
        int CurrentB = 0;   //转置三元组表存放指针
        int i, k;
```

```
        for (k = 0; k < Cols; k++)      //处理所有列号
            for (i = 0; i < Terms; i++)
                if (smArray[i].col == k) {
                    B.smArray[CurrentB].row = k;
                    B.smArray[CurrentB].col =
                                    smArray[i].row;
                    B.smArray[CurrentB].value=
                                    smArray[i].value;
                    CurrentB++;
                }
    }
};
```

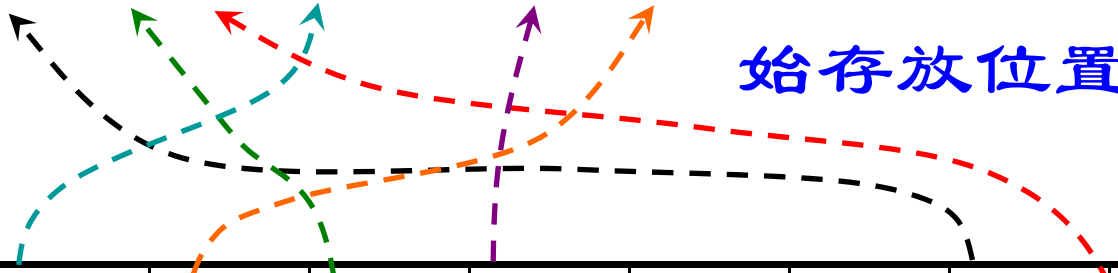
快速转置算法

- 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*)**。当非零元素的个数 *t* 和 ***m*n*** 同数量级时，算法**Transpose**的时间复杂度为$O(m*n^2)$。
- 若矩阵有 **200** 行，**200** 列，**10,000** 个非零元素，总共有 **2,000,000** 次处理。
- 快速转置算法的思想为：对原矩阵**A** 扫描一遍（实际为两遍，建辅助数组需扫描一遍），按 **A** 中每一元素的列号，立即确定在转置矩阵 **B** 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 **rowSize** 和 **rowStart**：
  - **rowSize**记录矩阵转置前各列，即转置矩阵各行非零元素个数；
  - **rowStart**记录转置矩阵各行非零元素在转置三元组表中开始存放位置。

- 扫描矩阵三元组表，根据某项列号，确定它转置后的行号，查 **rowStart** 表，按查到的位置直接将该项存入转置三元组表中。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	语 义
rowSize	1	1	1	2	0	2	1	矩阵 A 各列非零元素个数
rowStart	0	1	2	3	5	5	7	矩阵 B 各行开始存放位置



A三元组	(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)
行row	0	0	1	1	2	3	4	5
列col	3	6	1	5	3	5	0	2
值value	22	15	11	17	-6	39	91	28

稀疏矩阵的快速转置算法

```
template <class E>
void SparseMatrix<E>::
    FastTranspose (SparseMatrix<E>& B) {
  int *rowSize = new int[Cols];       //列元素数数组
  int *rowStart = new int[Cols];      //转置位置数组
  B.Rows = Cols;   B.Cols = Rows;
  B.Terms = Terms;
  if (Terms > 0) {
    int i, j;
    for (i = 0; i < Cols; i++) rowSize[i] = 0;
```

```
        for (i = 0; i < Terms; i++)
            rowSize[smArray[i].col]++;
        rowStart[0] = 0;
        for (i = 1; i < Cols; i++)
            rowStart[i] = rowStart[i - 1]+rowSize[i - 1];
        for (i = 0; i < Terms; i++) {
            j = rowStart [smArray[i].col];
            B.smArray[j].row = smArray[i].col;
            B.smArray[j].col = smArray[i].row;
            B.smArray[j].value = smArray[i].value;
            rowStart [smArray[i].col]++;
        }
    }
    delete [ ] rowSize;   delete [ ] rowStart;
}
```

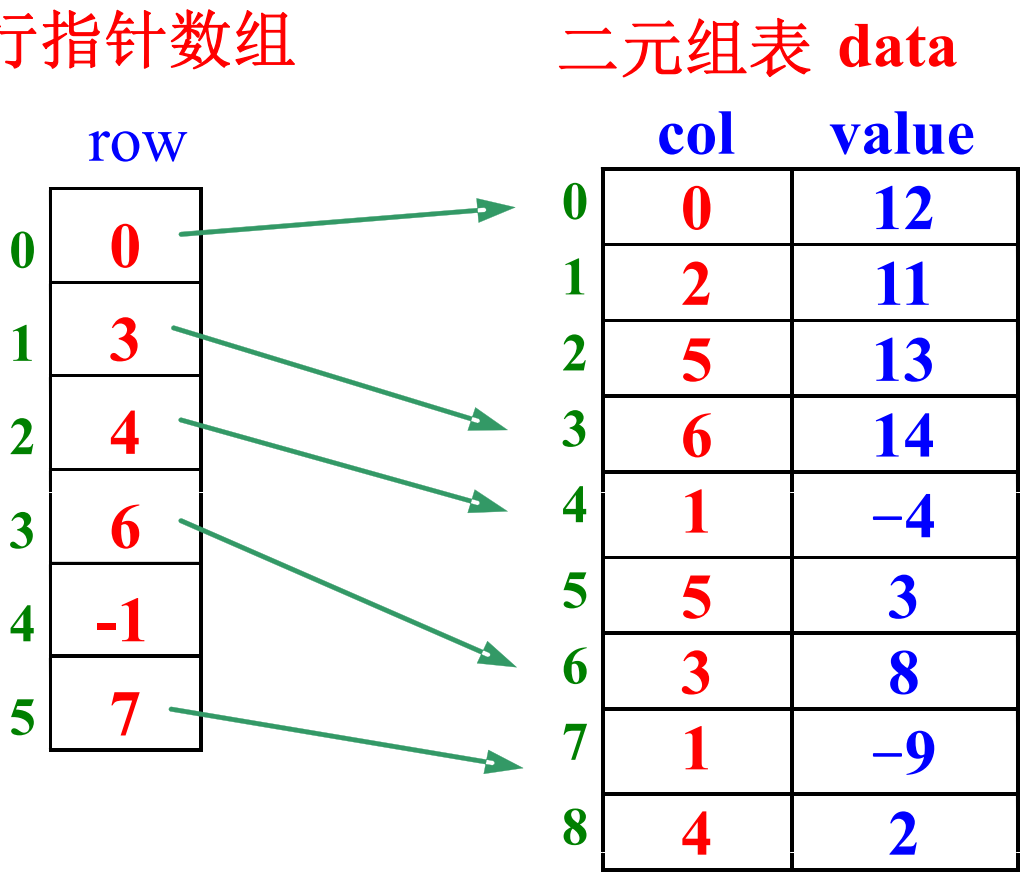
带行指针数组的二元组表

- 稀疏矩阵的三元组表可以用带行指针数组的二元组表代替。

- 在行指针数组中元素个数与矩阵行数相等。第 *i* 个元素的下标 *i* 代表矩阵的第 *i* 行，元素的内容即为稀疏矩阵第 *i* 行的第一个非零元素在二元组表中的存放位置。

- 二元组表中每个二元组只记录非零元素的列号和元素值，且各二元组按行号递增的顺序排列。

$$\begin{pmatrix} 12 & 0 & 11 & 0 & 0 & 13 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$



用正交链表来表示稀疏矩阵

- 优点：适应矩阵操作(+、-、*)时矩阵非零元素的动态变化
- 稀疏矩阵表示为行链表与列链表的十字交叉——称为正交链表（十字链表）

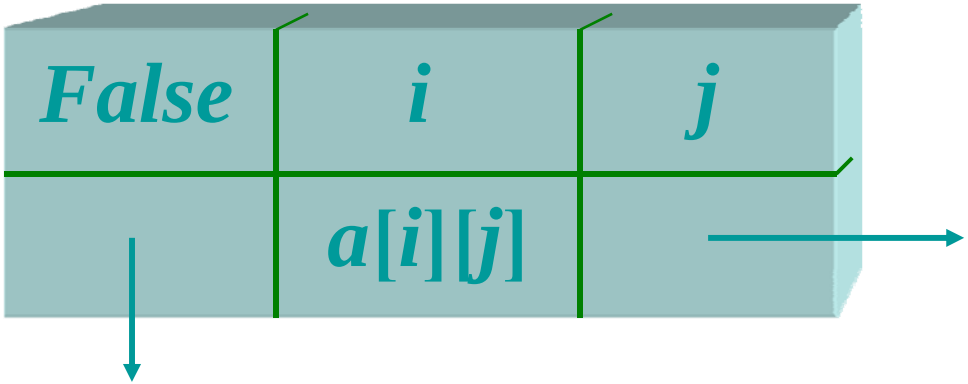
- 稀疏矩阵的结点

head	*next*
down	*right*

(a) 表头结点

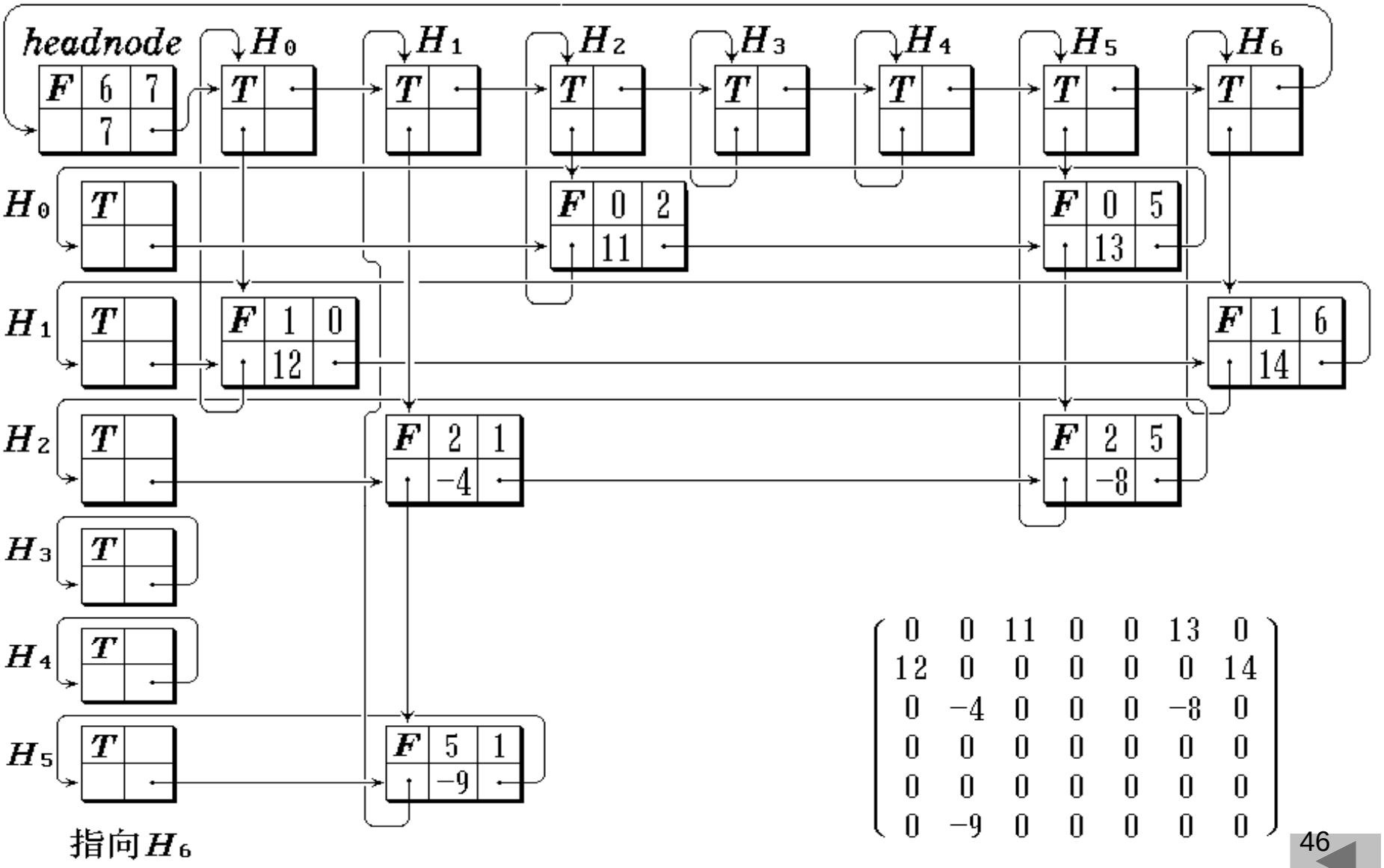
head	*row*	*col*
down	*value*	*right*

(b) 非零元素结点



(c) 建立*a*[*i*][*j*]结点

稀疏矩阵的正交链表表示的示例



$$\begin{pmatrix} 0 & 0 & 11 & 0 & 0 & 13 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

字符串 (String)

- 字符串是 n (≥ 0) 个字符组成的有限序列，

  记作 S : “$c_0c_1c_2\ldots c_{n-1}$”

  其中，S 是串名字

  “$c_0c_1c_2\ldots c_{n-1}$”是串值

  （默认包含串结束符‘\0’）

  c_i 是串中字符

  n 是串的长度，$n = 0$ 称为空串。

- 例如，S = “Nanjing University”。

- 注意：空串和空白串不同，例如“ ”和“”分别表示长度为1的空白串和长度为0的空串。

- 串中任意个连续字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。

- 设A和B分别为

  A = “This is a string”　B = “is”

  则 **B** 是 **A** 的子串，**A** 为主串。**B** 在 **A** 中出现了两次，首次出现所对应的主串位置是**2**（从**0**开始）。因此，称 **B** 在 **A** 中的位置为**2**。

- 特别地，空串是任意串的子串，任意串是其自身的子串。

字符串的类定义

```
#ifndef ASTRING_H      //定义在文件“Astring.h”中
#define ASTRING_H
#define defaultSize = 128;     //字符串的最大长度
class AString {
//对象: 零个或多个字符组成的一个有限序列。
private:
    char *ch;                  //串存放数组
    int curLength;             //串的实际长度
    int maxSize;               //存放数组的最大长度
```

```
public:
   AString(int sz = defaultSize);      //构造函数
   AString(const char *init );         //构造函数
   AString(const AString& ob);         //复制构造函数
   ～AString() {delete [ ]ch; }        //析构函数
   int Length() const { return curLength; }   //求长度
   Astring& operator() (int pos, int len);   //求子串
   bool operator == (AString& ob) const
      { return strcmp (ch, ob.ch) == 0; }
      //判串相等. 若串相等, 则函数返回true
   bool operator != (AString& ob) const
      { return strcmp (ch, ob.ch) != 0; }
     //判串不等. 若串不相等, 则函数返回true
```

```
    bool operator ! () const { return curLength == 0; }
      //判串空否。若串空, 则函数返回true
    AString& operator = (AString& ob);      //串赋值
    AString& operator += (AString& ob);    //串连接
    char& operator [ ] (int i);             //取第 i 个字符
    int Find (AString& pat, int start) const;    //串匹配
};
```

字符串的构造函数

```
AString::AString(int sz) {
//构造函数：创建一个空串
    maxSize = sz;
    ch = new char[maxSize+1];        //创建串数组
    if (ch == NULL)
       { cerr << "存储分配错!\n"; exit(1); }
    curLength = 0;
    ch[0] = '\0';
};
```

字符串的构造函数

```
AString::AString(const char *init) {
//复制构造函数: 从已有字符数组*init复制
    int len = strlen(init);
    maxSize = (len > defaultSize) ? len : defaultSize;
    ch = new char[maxSize+1];     //创建串数组
    if (ch == NULL)
      { cerr << "存储分配错！\n";  exit(1); }
    curLength = len;                    //复制串长度
    strcpy(ch, init);                   //复制串值
};
```

字符串的复制构造函数

```
AString :: AString(const AString& ob) {
//复制构造函数：从已有串ob复制
    maxSize = ob.maxSize;              //复制串最大长度
    ch = new char[maxSize +1];     //创建串数组
    if (ch == NULL)
       { cerr << "存储分配错! \n";  exit(1); }
    curLength = ob.curLength;          //复制串长度
    strcpy(ch, ob.ch);                 //复制串值
};
```

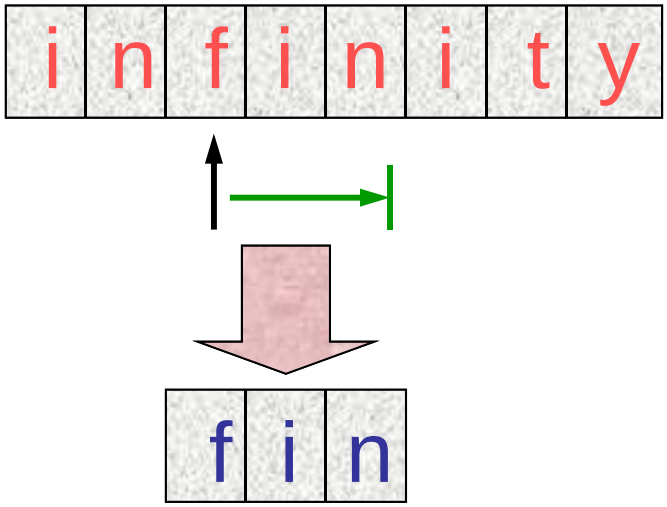
字符串重载操作的使用示例

序号	重载操作	操作	使用示例 (设使用操作的当前串为S：‘good’)
1	( ) (int pos, int len)	取子串	S1 = S(1, 3), //S1结果为‘ood’
2	== (const AString& ob)	判两串相等	S == S1 , //若S与S1相等，结果为true，否则为false
3	!= (const AString& ob)	判两串不等	S != S1 , //若S与S1不等，结果为true，否则为false
4	! ()	判串空否	!S , //若串S为空，结果为true，否则为false

序号	重载操作	操作	使用示例 (设使用操作的当前串为S：‘good’)
5	= (const AString& ob)	串赋值	S1 = S, //S1结果为‘good’
6	+= (const AString& ob)	串连接	若设S1为‘ morning’, 执行S += S1, //S结果为‘good morning’
7	[ ] (int i)	取第 i 个字符	S[3], //取出字符为‘d’

提取子串的算法示例

pos = 2, len = 3

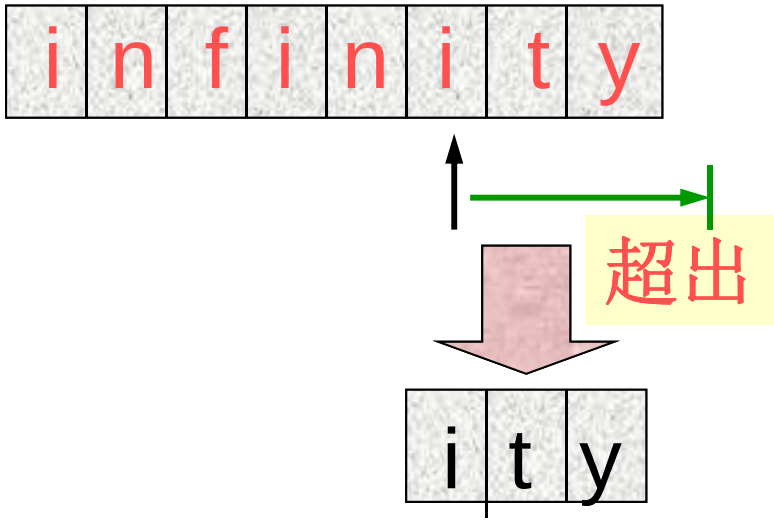


pos+len−1

≤ curLength−1

可以全部提取

pos = 5, len = 4



pos+len −1

≥ curLength

只能从**pos**取到串尾

串重载操作：提取子串

```
AString  AString::operator () (int pos, int len) {
//从串中第 pos 个位置起连续提取 len 个字符形成
//子串返回
    AString  temp;                          //建立空串对象
    if (pos >= 0 && pos+len-1 < maxSize && len > 0)
    {                                       //提取子串
        if (pos+len-1 >= curLength)
            len = curLength  -  pos;        //调整提取字符数
        temp.curLength = len;               //子串长度
```

```
        for (int i = 0, j = pos; i < len; i++, j++)
            temp.ch[i] = ch[j];          //传送串数组
        temp.ch[len] = '\0';             //子串结束
    }
    return  temp;
};
```

- 例：串 st = “university”，pos = 3，len = 4

  使用示例　subSt = st(3, 4)

  提取子串　subSt = “vers”

串重载操作: 串赋值

```
AString& AString::operator = (const AString& ob) {
    if (&ob != this) {        //若两个串相等为自我赋值
        delete []ch;
        ch = new char[maxSize+1];      //重新分配
        if (ch == NULL)
           { cerr << "存储分配失败!\n "; exit(1); }
        curLength = ob.curLength;  strcpy(ch,ob.ch);
    }
    else cout << "字符串自身赋值出错!\n";
    return this;
};
```

串重载操作：取串的第i个字符

```
char AString::operator [ ] (int i) {
//串重载操作：取当前串*this的第i个字符
    if (i < 0 || i >= curLength)
       { cout << "字符串下标超界!\n "; exit(1); }
    return ch[i];
};
```

- 例：串 st = “university”,
  使用示例　newSt = st;　newChar = st[1];
  数组赋值结果　newSt = “university”
  提取字符结果　newChar = ‘n’

串重载操作: 串连接

```
AString& AString::operator += (const AString& ob)
{
    char *temp = ch;                    //暂存原串数组
    int n = curLength + ob.curLength;       //串长度累加
    int m = (maxSize >= n) ? maxSize : n; //新空间大小
    ch = new char[m];
    if (ch == NULL)
       { cerr << "存储分配错!\n "; exit(1); }
    maxSize = m;  curLength = n;
    strcpy(ch, temp);                   //拷贝原串数组
    strcat(ch, ob.ch);                  //连接ob串数组
```

```
    delete []temp;
    return this;
};
```

- 例：串 **st1 = “Nanjing ”,**
  st2 = “university”,
  使用示例 **st1 += st2;**
  连接结果 **st1 = “Nanjing university”**
  st2 = “university”

串的模式匹配

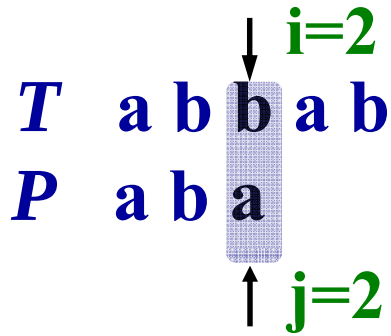
- 定义 在主串中寻找子串（第一个字符）在串中的位置
- 词汇 在模式匹配中，子串称为模式，主串称为目标。
- 示例 目标 **T : “Nanjing”**

  模式 P : “jin”

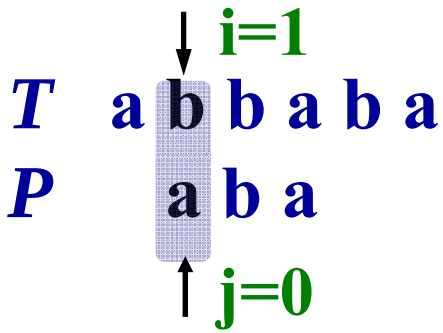
  匹配结果 = 3

朴素的模式匹配

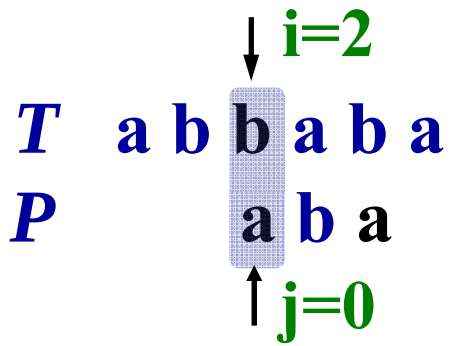
第1趟



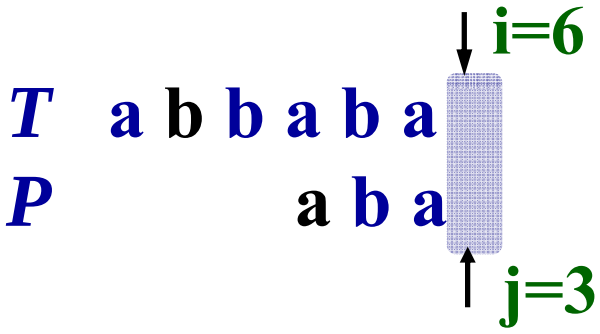
第2趟



第3趟



第4趟



朴素的模式匹配算法

```
int AString::Find(AString& pat, int start) const {
//在当前串中从第 start 个字符开始寻找模式 pat 在当
//前串中匹配的位置。若匹配成功, 则函数返回首
//次匹配的位置, 否则返回 -1。
    int i, j, n = curLength, m = pat.curLength;
    if (m == 0) return -1;    //pat为空
    for (i = start; i <= n-m; i++) {
        for (j = 0; j < m; j++){
            if (ch[i+j] != pat.ch[j]) break;   //本次失配
        }
        if (j == m) return i;   //pat扫描完, 匹配成功
    }
  return -1;          //在*this中找不到子串
};
```

- 在最坏情况下，若设 **n** 为目标串长度，**m** 为模式串长度，则匹配算法最多比较 **n−m+1** 趟，每趟比较都在比较到模式串尾部才出现不等，要做 **m** 次比较，总比较次数将达到 **(n-m+1)*m**。在多数场合下 **m** 远小于 **n**，因此，算法的运行时间为**O(n*m)**。

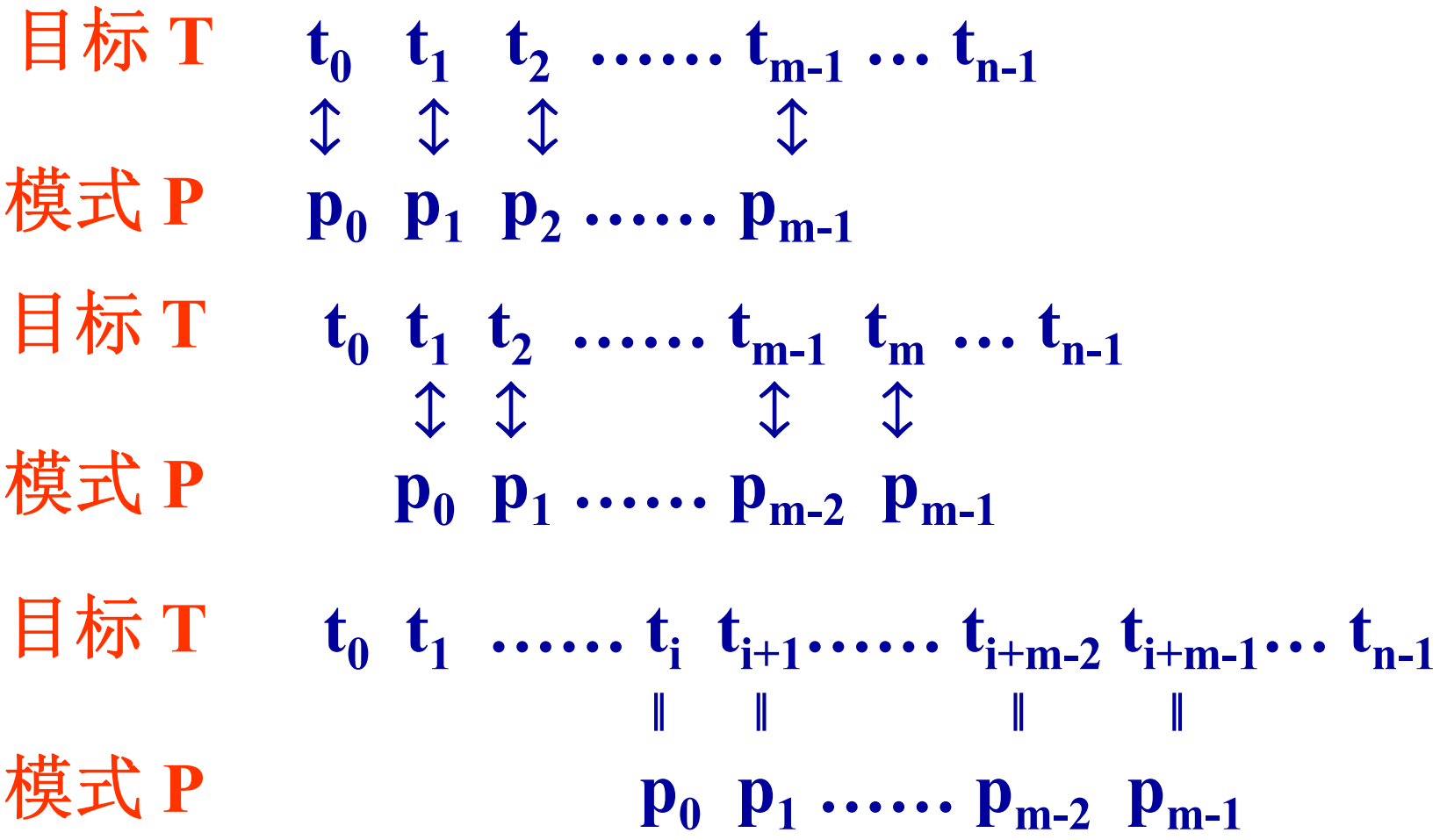
- 低效的原因在于每趟重新比较时，目标串的检测指针要回退。

改进的模式匹配

- 只要消除了每趟失配后为实施下一趟比较时目标指针的回退，就可以提高模式匹配效率。

- 这种处理思想是由**D.E.Knuth**、**J.H.Morris**和**V.R.Pratt**同时提出来的，故称为**KMP**算法。

改进的模式匹配


目标 T
t_0 t_1 t_2 …… t_{m-1} … t_{n-1}
模式 P
p_0 p_1 p_2 …… p_{m-1}
目标 T
t_0 t_1 t_2 …… t_{m-1} t_m … t_{n-1}
模式 P
p_0 p_1 …… p_{m-2} p_{m-1}
目标 T
t_0 t_1 …… t_i t_{i+1}…… t_{i+m-2} t_{i+m-1}… t_{n-1}
模式 P
p_0 p_1 …… p_{m-2} p_{m-1}

$T \quad t_0\ t_1 \dots t_{s-1}\ t_s\ \ t_{s+1}\ t_{s+2} \dots t_{s+j-1}\ t_{s+j}\ t_{s+j+1} \dots t_{n-1}$

$\qquad\qquad\qquad\quad \parallel \quad \parallel \quad \parallel \quad \parallel \quad \parallel \quad \times$

$P \qquad\qquad\qquad\quad p_0\ \ p_1\ \ p_2\ \ \dots\ p_{j-1}\ \ p_j$

则有 $t_s\ t_{s+1}\ t_{s+2} \dots t_{s+j-1} = p_0\ p_1\ p_2 \dots p_{j-1}$ (1)

下一趟，为使模式 P 与目标 T 匹配，必须满足

$p_0\ p_1\ p_2 \dots p_{j-1} \dots p_{m-1} = t_{s+1}\ t_{s+2}\ t_{s+3} \dots t_{s+j} \dots t_{s+m}$

如果 $p_0\ p_1 \dots p_{j-2} \neq p_1\ p_2 \dots p_{j-1}$ (2)

则立刻可以断定

$p_0\ p_1 \dots p_{j-2} \neq t_{s+1}\ t_{s+2} \dots t_{s+j-1}$

下一趟必不匹配 $p_0\ \ p_1\ \ \dots\ p_{j-2}$

同样，若 $p_0\,p_1\cdots p_{j-3} \neq p_2\,p_3\cdots p_{j-1}$

则再下一趟也不匹配，因为有

$$p_0\,p_1\cdots p_{j-3} \neq t_{s+2}\,t_{s+3}\cdots t_{s+j-1}$$

直到对于某一个“k”值，使得

$$p_0\,p_1\cdots p_{k+1} \neq p_{j-k-2}\,p_{j-k-1}\cdots p_{j-1}$$

且 $p_0\,p_1\cdots p_k = p_{j-k-1}\,p_{j-k}\cdots p_{j-1}$

则 $p_0\,p_1\cdots p_k = t_{s+j-k-1}\,t_{s+j-k}\cdots t_{s+j-1}$

$\|\qquad\|\qquad\|$

$p_{j-k-1}\;p_{j-k}\cdots\;p_{j-1}$

下一趟可以直接用 p_{k+1} 与 t_{s+j} 继续比较。

k 的确定方法

- **Knuth** 等人发现，对于不同的 **j**（失配位置），**k** 的取值不同，它仅依赖于模式 **P** 本身前 **j** 个字符的构成，与目标无关。

- 可以用一个 **next** 特征向量来确定：当模式 **P** 中第 **j** 个字符与目标 **T** 中相应字符失配时，模式 **P** 中应当由哪个字符（设为第**k+1**个）与目标中刚失配的字符重新继续进行比较。

- 设模式 $P = p_0p_1\cdots p_{m-2}p_{m-1}$，next特征向量定义如下：

$$next(j)=\begin{cases}-1, & j=0\\ k+1, & 0\le k<j-1\text{且使得}p_0p_1\cdots p_k=p_{j-k-1}p_{j-k}\cdots p_{j-1}\text{的最大整数}\\ 0, & \text{其他情况}\end{cases}$$

- 示例

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
next(j)	-1	0	0	1	1	2	0	1

利用next特征向量进行匹配处理

- 若设在进行某一趟匹配比较时在模式 **P** 的第 **j** 位失配：
  - 如果 **j > 0**，那么在下一趟比较时模式串 **P** 的起始比较位置是 $p_{next(j)}$，目标串 **T** 的指针不回溯，仍指向上一趟失配的字符；
  - 如果 **j = 0**，则目标串指针 **T** 进一，模式串指针 **P** 回到 p_0，继续进行下一趟匹配比较。

运用KMP算法的匹配过程

第1趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=1 ⇨ next(1) = 0， 下次p_0

第2趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=0 ⇨ 下次p_0, 目标指针进 1

第3趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=5 ⇨ next(5) = 2,

下次p_2

第4趟 目标 a c a b a a b a a b c a c a a b c

模式 (a b) a a b c a c √

用KMP算法实现快速匹配算法

```
int AString::fastFind(AString& pat, int start,
        int next[]) const {
//从 start 开始寻找 pat 在 this 串中匹配的位置。若找
//到，函数返回 pat 在 this 串中开始位置，否则函
//数返回-1。数组next[ ] 存放 pat 的next[j] 值。
    int posP = 0,  posT = start;          //两个串的扫描指针
    int lengthP = pat.curLength;          //模式串长度
    int lengthT = curLength;              //目标串长度
    while (posP < lengthP && posT < lengthT)
     {   if (posP == -1 || pat.ch[posP] == ch[posT])
            { posP++;  posT++; }          //对应字符匹配
         else posP = next[posP];   //求pat下趟比较位置
     }
    if (posP < lengthP) return -1;     //匹配失败
    else return posT-lengthP;                  //匹配成功
};
```

- 此算法的时间复杂度取决于 **while** 循环。由于是无回溯的算法，执行循环时，目标串字符比较有进无退，要么执行 **posT** 和 **posP** 进 **1**，要么查找 **next[ ]** 数组进行模式位置的右移，然后继续向后比较。

- 若设 **n** 为目标串长度，**m** 为模式串长度，字符的比较次数最多为 **O(n)**，不超过目标串的长度。

next特征向量的计算

- 设模式 $P = p_0 p_1 p_2 \cdots p_{m-1}$ 由 m 个字符组成，而next特征向量为next $= n_0 n_1 n_2 \ldots n_{m-1}$，表示了模式的字符分布特征。
- next特征向量从0, 1, 2, …, $m-1$逐项递推计算:

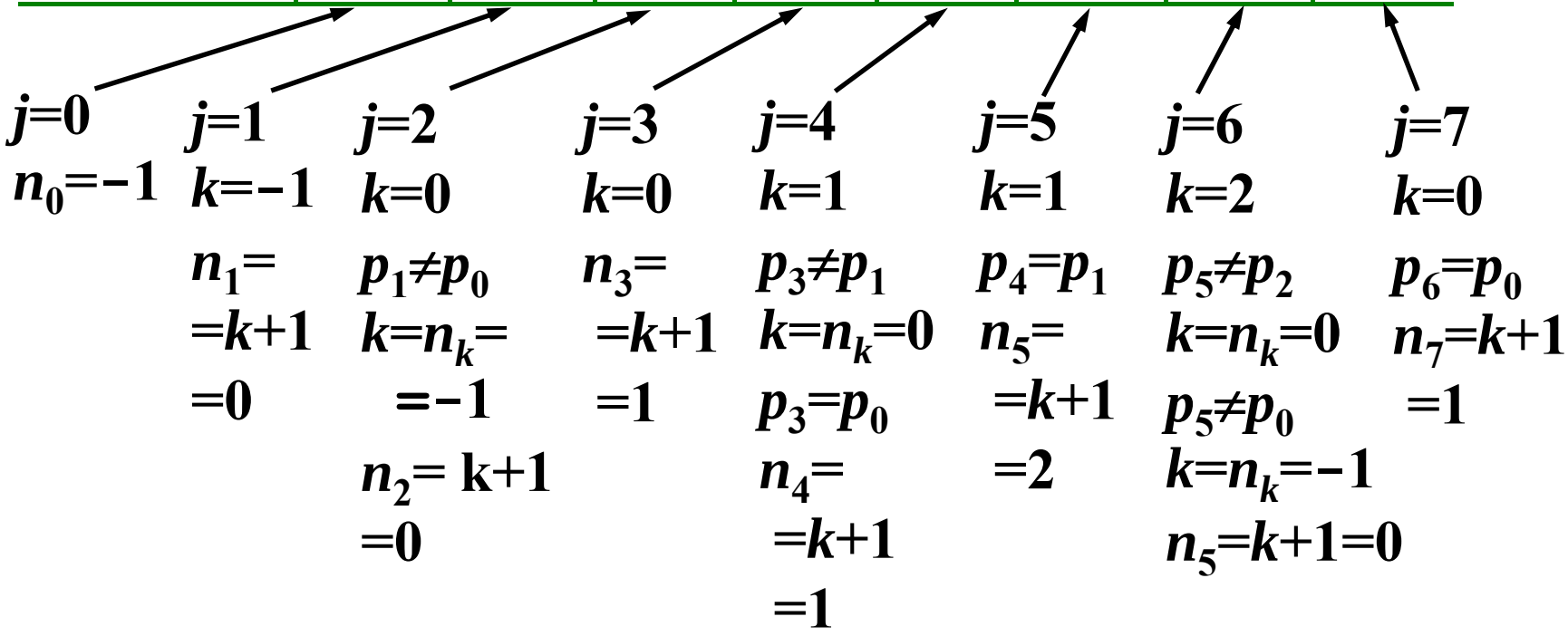
❶ 当 $j = 0$时，$n_0 = -1$。设 $j > 0$ 时 $n_{j-1} = = k$:

❷ 当 $k == -1$或 $(j > 0$且 $p_{j-1} == p_k)$，则 $n_j = k+1$。

❸ 当 $p_{j-1} \neq p_k$ 且 $k \neq -1$，令 $k = n_k$，并让❸循环直到条件不满足，$n_j = k+1$

- 以前面的例子说明：

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
next [j]	-1	0	0	1	1	2	0	1



对当前串计算next特征向量的算法

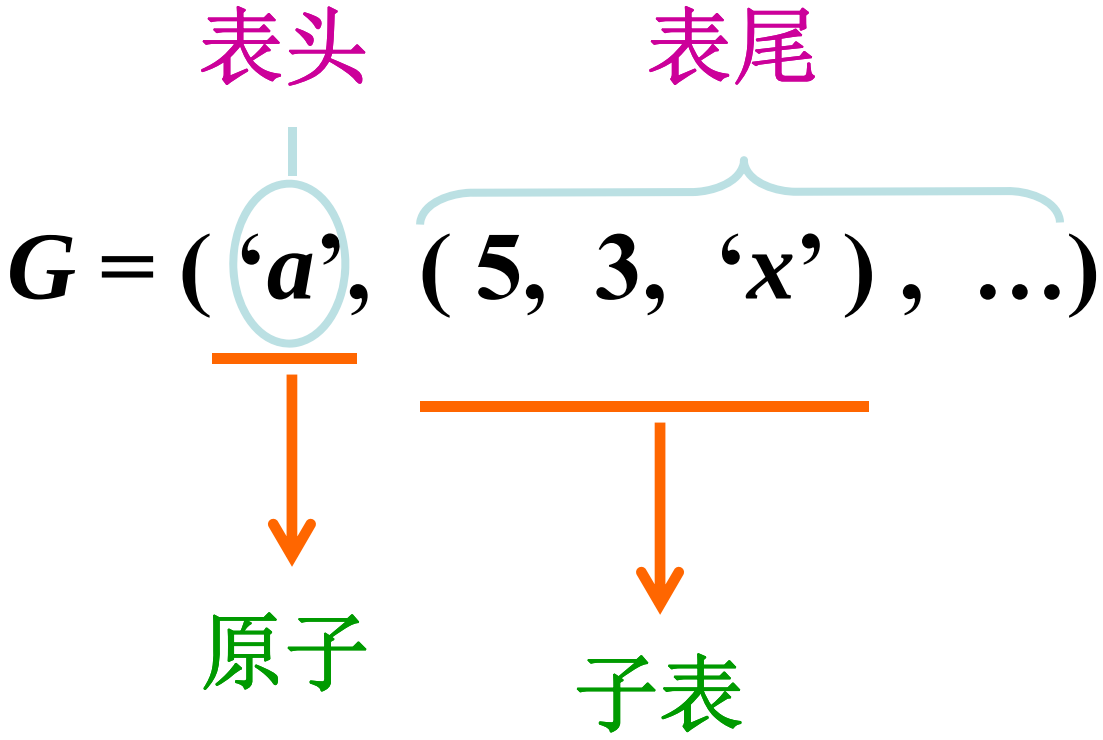
```
void AString::getNext(int next[]) {
    int j = 0, k = -1,  lengthP = curLength;
    next[0] = -1;  j = 1;
    while (j < lengthP) {                    //计算next[j]
        k = next[j-1];
        while(1) {
            if ( k == -1 || ch[j-1] == ch[k]) {
                k++;
                next[j] = k;
                j++;
                break;
            }
            else k = next[k];
        }
    }
};
```

广义表 (General Lists)

- 广义表是 n (≥0) 个表元素组成的有限序列，记作

$$LS\ (a_1, a_2, a_3, \ldots, a_n)$$

- LS 是表名，a_i 是表元素，可以是表（称为子表），可以是数据元素（称为原子）。
- 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）。**

广义表的特性

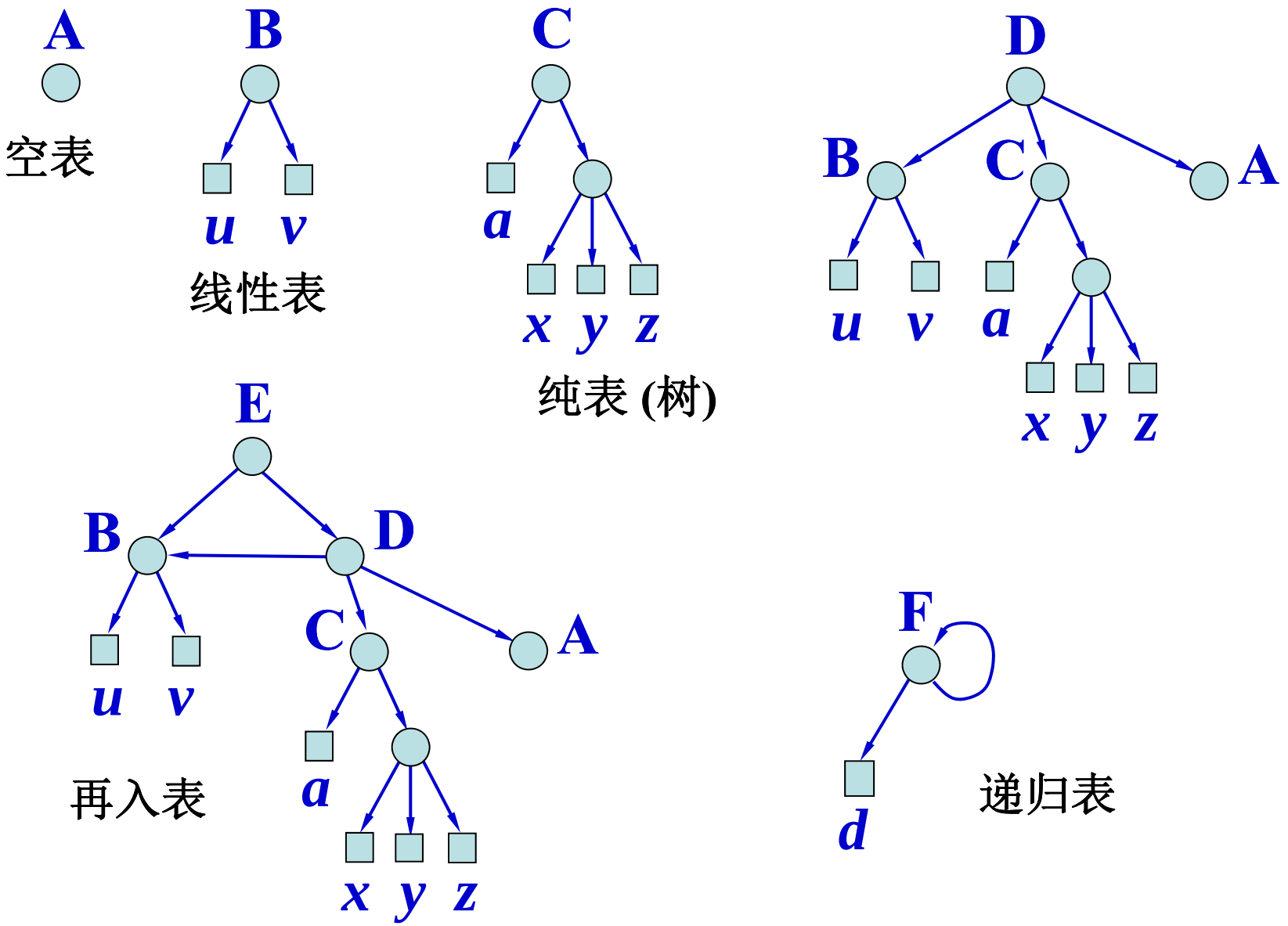
- 有次序性
- 有长度
- 有深度
- 可共享
- 可递归

A( )	**A长度为0，深度为1**
B(6, 2)	**B长度为2，深度为1**
C(‘*a*’, (5, 3, ‘*x*’))	**C长度为2，深度为2**
D(B, C, A)	**D长度为3，深度为3**
E(B, D)	**E长度为？ 深度为？**
F(4, F)	**F长度为？ 深度为？**

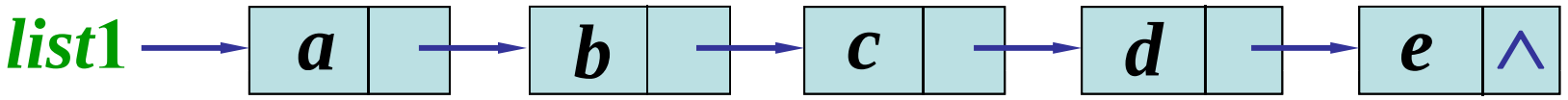
广义表的表头与表尾

- 广义表的第一个表元素即为该表的表头，除表头元素外其他表元素组成的表即为该表的表尾。

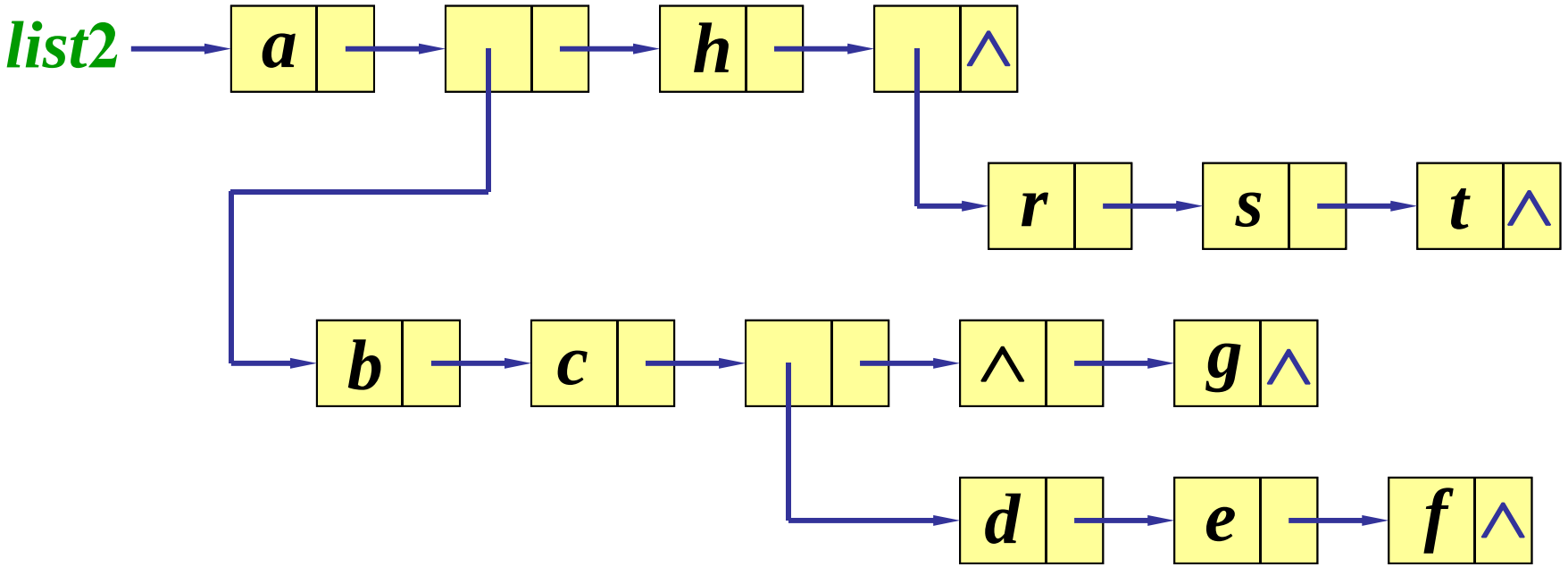
A( )	**head(A) 和 tail(A) 不存在**
B(6, 2)	**head(B) = 6, tail(B) = (2)**
C(‘*a*’, (5, 3, ‘*x*’))	**head(C) =‘*a*’**
D(B, C, A)	**tail(C) = ((5,3, ‘x’))**
E(B, D)	**head(((5,3, ‘x’))) = (5,3, ‘x’)**
F(4, F)	**tail(((5,3, ‘x’))) = ( )**


A
空表
B
u v
线性表
C
a
x y z
纯表 (树)
D
B C A
u v a
x y z
E
B D
u v
C A
a
x y z
再入表
F
d
递归表

广义表的表示



*list*1 = (*a*, *b*, *c*, *d*, *e*)



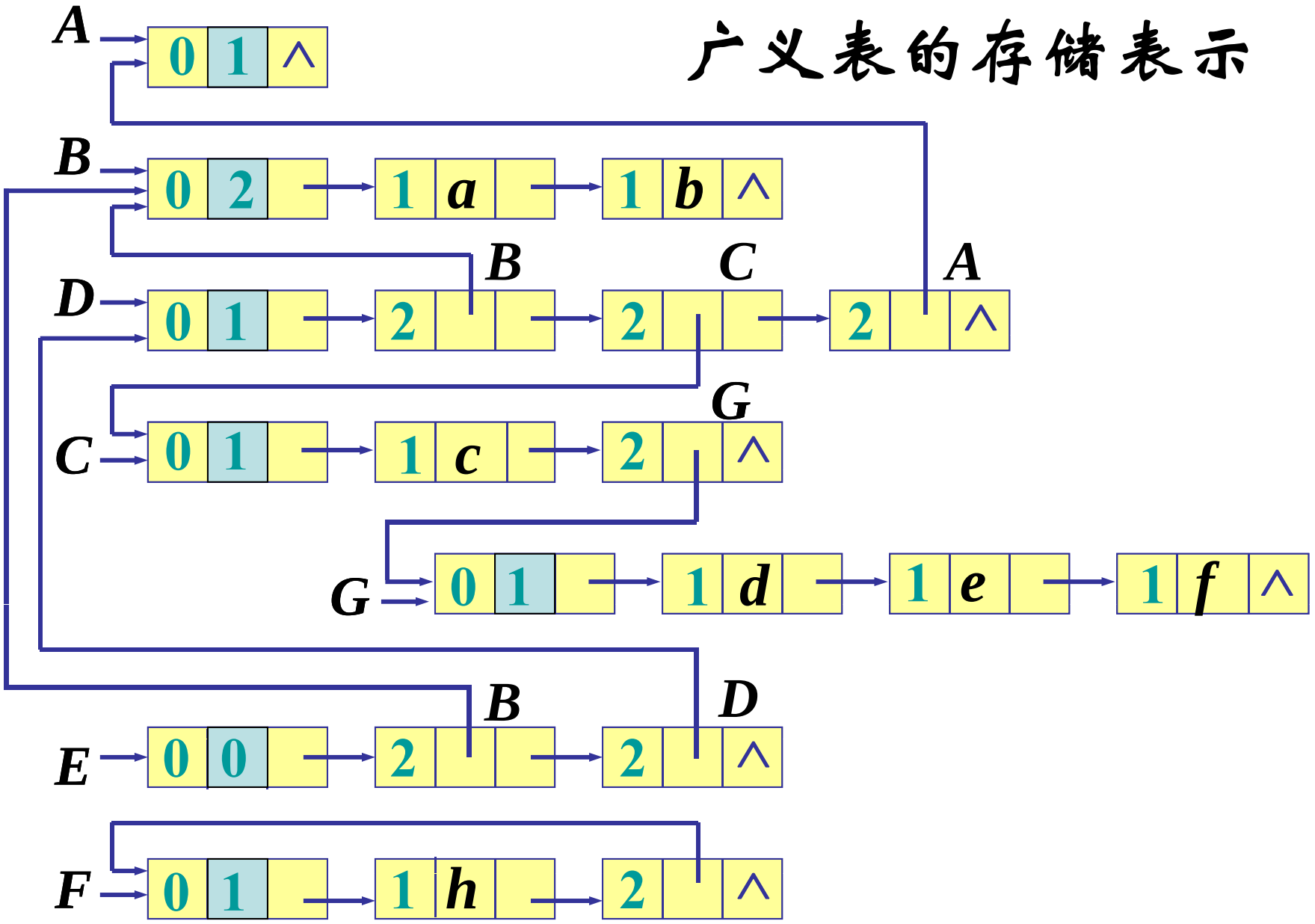
*list*2 = (*a*, (*b*, *c*, (*d*, *e*, *f*), (), *g*), *h*, (*r*, *s*, *t*))

广义表结点定义

utype	info	tlink

- 结点类型 **utype**：＝**0**, 表头；＝**1**, 原子数据；＝**2**, 子表
- 信息**info**：**utype** ＝ **0**时, 存放引用计数**(ref)**；**utype** ＝ **1**时, 存放数据值**(value)**；**utype** ＝ **2**时, 存放指向子表表头的指针**(hlink)**
- 尾指针**tlink**：**utype** ＝ **0**时, 指向该表第一个结点；**utype** ≠ **0**时, 指向同一层下一个结点

广义表的存储表示



广义表的类定义

```
template <class T>
struct GenListNode {          //广义表结点类定义
    int utype;                 //＝0 / 1 / 2
    GenListNode<T> *tlink;   //同层下一结点指针
    union {                                 //等价变量
        int ref;              //存放引用计数
        T value;              //存放数值
        GenListNode<T> *hlink;          //存放子表指针
    } info;
    GenListNode()                       //构造函数
        : utype(0), tlink(NULL), info.ref(0) {}
    GenListNode(GenListNode<T>& R) {
    //复制构造函数
        utype = R.utype;   tlink = R.tlink;
        info = R.info;
    }
};
```

```
template <class T>
class GenList {                                    //广义表类定义
   public:
     Genlist();                                    //构造函数
     ～GenList();                                  //析构函数
     bool Head (GenListNode<T>& x);   //x 返回表头元素
     bool Tail (GenList<T>& lt);          //lt 返回表尾
     GenListNode<T> *First();   //返回第一个元素地址
     GenListNode<T> *Next (GenListNode<T> *elem);
          //返回表元素elem的直接后继元素
     void Copy ( const GenList<T>& R);
          //广义表的复制
     int Length();                                 //计算广义表长度
     int Depth();                                  //计算非递归表深度
```

```
private:
    GenListNode<T> *first;       //广义表头指针
    GenListNode<T> *Copy (GenListNode<T> *ls);
      //复制一个ls指示的无共享非递归表
    int Length (GenListNode<T> *ls);
      //求由ls指示的广义表的长度
    int Depth (GenListNode<T> *ls);
      //计算由ls指示的非递归表的深度
    bool Equal (GenListNode<T> *s,
          GenListNode<T> *t);
     //判以s和t为表头的两个表是否相等
    void Remove (GenListNode<T> *ls);
     //释放以ls为附加头结点的广义表
```

```
    void CreateList (istream &in,  GenListNode<T> *&
ls, SeqList<T>& L1,  SeqList <GenListNode<T> *>&
L2);
        //从输入流对象输入广义表的字符串描述,
        //建立一个带头结点的广义表结构

  friend istream& operator >> (istream& in,
          GenList<T>& L);
};
```

广义表类的构造和访问成员函数

```
template <class T>
Genlist<T>::GenList() {                    //构造函数
    GenListNode<T> * first = new GenListNode;
    if (first == NULL)   { cerr << "存储分配失败！ \n";  exit(1); }
};

template <class T>
bool GenList<T>::Head (GenListNode <T>& x) {
//若广义表非空，则通过x返回其第一个元素的值
//否则函数没有定义
   if (first->tlink == NULL) return false;     //空表
   else {                                                  //非空表
        x.utype = first->tlink->utype;
        x.info = first->tlink->info;
        return true;                              //x返回表头的值
   }
};
```

```
template <class T>
bool GenList<T>::Tail(GenList<T>& lt) {
//若广义表非空，则通过lt返回广义表除表头元素
//以外其他元素组成的表，否则函数没有定义
    if (first->tlink == NULL) return false;       //空表
    else {                                        //非空表
        lt.first->utype = 0;                      //设置头结点
        lt.first->info.ref = 0;
        lt.first->tlink = Copy(first->tlink->tlink);
        return true;
    }
};
```

```
template <class T>
GenListNode<T> *GenList<T>::First() {
//返回广义表的第一个元素（若表空，则返回一个
//特定的空值NULL）
   if (first->tlink == NULL) return NULL;    //空表
   else return first->tlink;                 //非空表
};
```

```
template <class T>
GenListNode<T> *GenList<T>::
Next(GenListNode<T> *elem) {
//返回表元素elem的直接后继元素
    if (elem->tlink == NULL) return NULL;
    else return elem->tlink;
};
```

广义表的递归算法

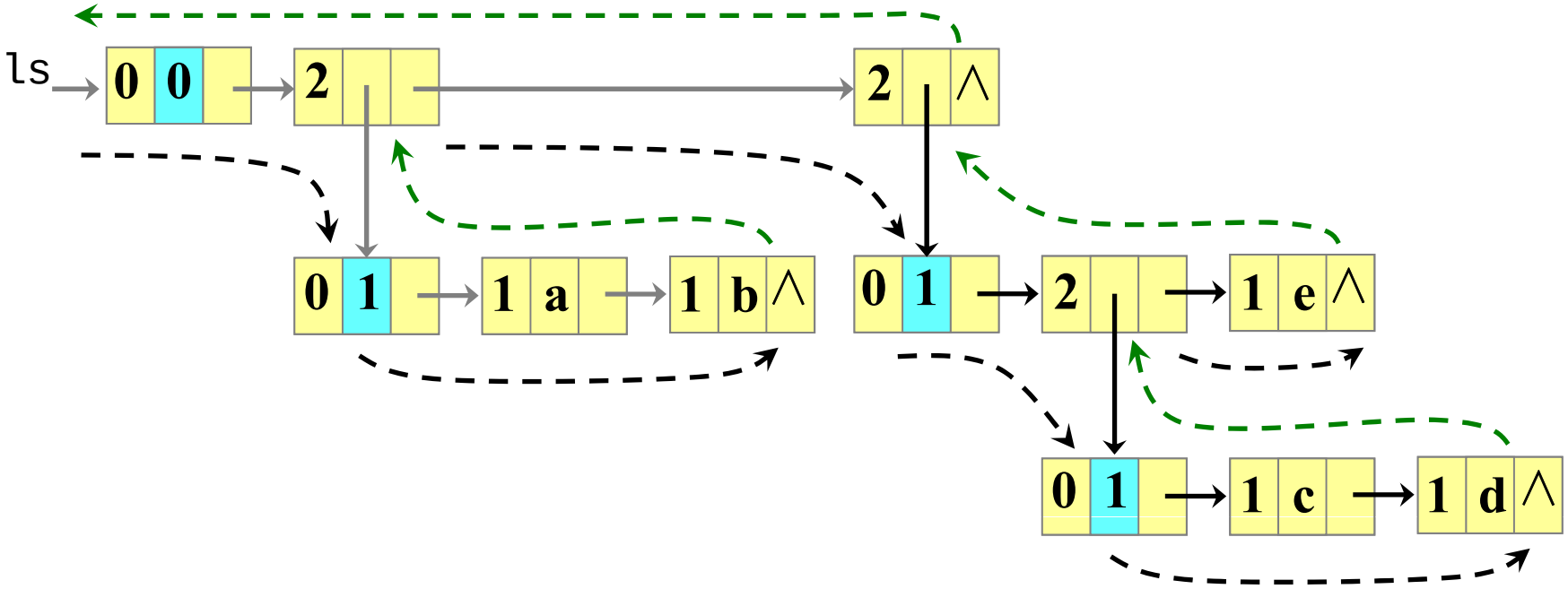
- 一个递归算法有两种：一个是递归函数的外部调用；另一个是递归函数的内部调用。

- 通常，把外部调用设置为公有函数，把内部调用设置为私有函数。

广义表的复制算法

```
template <class T>                    //公有函数
void GenList<T>::Copy(const GenList<T>& R)
    first = Copy(R.first);            //调用私有函数
};
```

```
template <class T>                    //私有函数
GenListNode<T>* GenList<T>::Copy(GenListNode <T> *ls) {
   //复制一个 ls 指示的无共享子表的非递归表
    GenListNode<T> *q = NULL;
    if (ls != NULL) {
        q = new GenListNode<T>;  //处理当前的结点q
        q->utype = ls->utype;    //复制结点类型
          switch (ls->utype) {     //根据utype传送信息
            case 0: q->info.ref = ls->info.ref;  break;
            case 1: q->info.value = ls->info.value;  break;
            case 2: q->info.hlink = Copy(ls->info.hlink);
                    break;
           }
        q->tlink = Copy(ls->tlink);  //处理同层下一结点
   }
    return q;
};
```

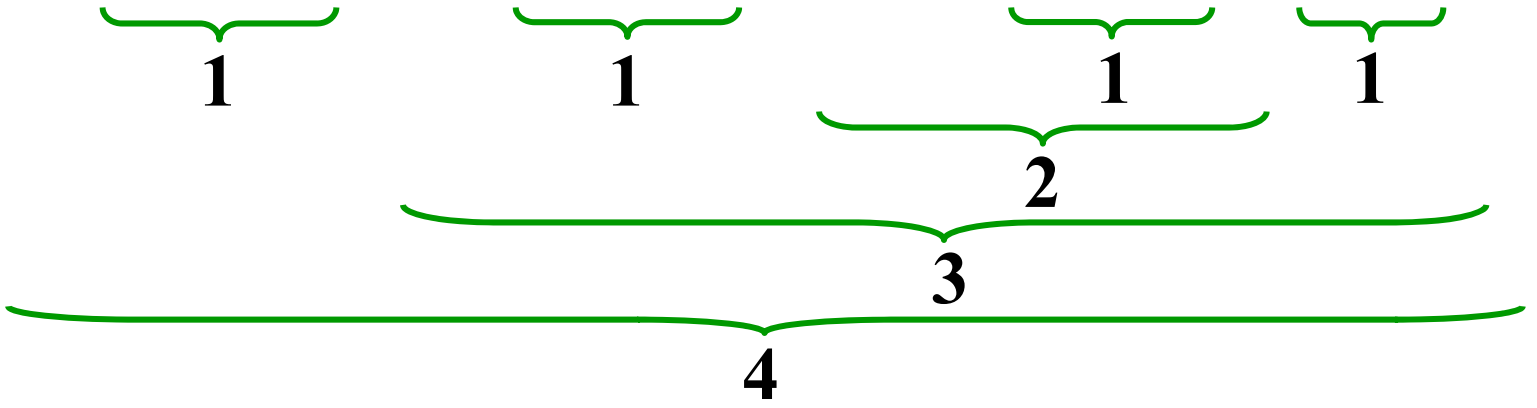

ls
0 0
2
2 ∧
0 1
1 a
1 b ∧
0 1
2
1 e ∧
0 1
1 c
1 d ∧

求广义表深度的算法

$$\text{Depth}(LS)=\begin{cases}1, & \text{当LS为空表时}\\ 0, & \text{当LS为原子时}\\ 1+\max\limits_{0\le i\le n-1}\{\text{Depth}(\alpha_i)\}, & \text{其他},n\ge 1\end{cases}$$

- 例如，对于广义表

E (B (a, b), D (B (a, b), C (u, (x, y, z)), A ( )))

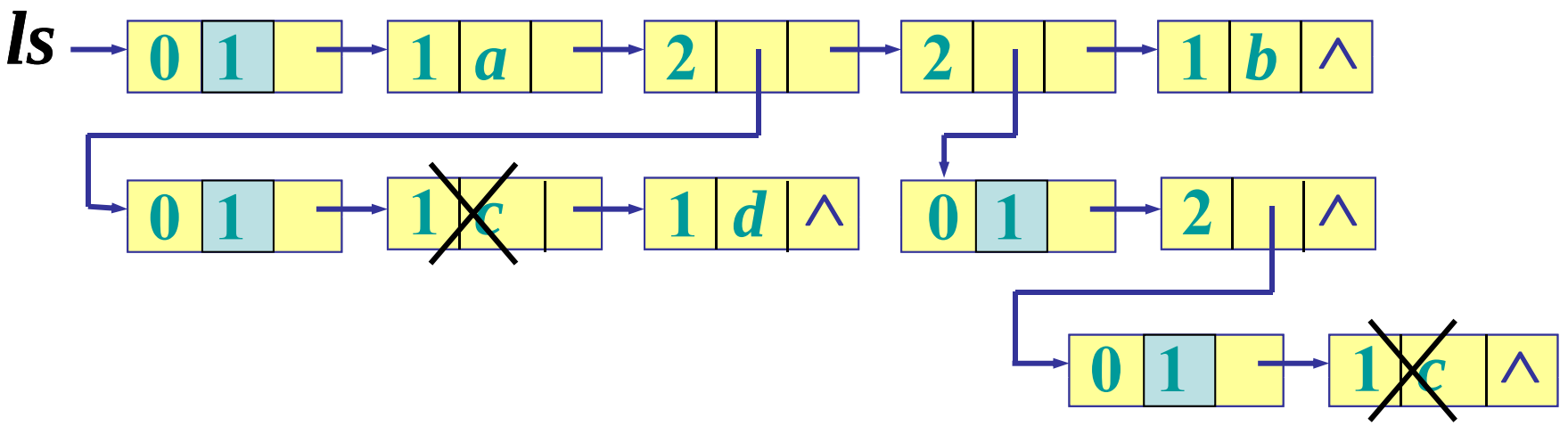


```
template <class T>
int GenList<T>::Depth() {              //公有函数
//计算一个非递归表的深度
    return Depth(first);
};

template <class T>                     //私有函数
int GenList<T>::Depth(GenListNode<T> *ls) {
    if (ls->tlink == NULL) return 1;
     // ls->tlink ==NULL, 空表，深度为1
    GenListNode<T> *temp = ls->tlink;
    int m = 0, n;
```

```
    while (temp != NULL) {      //在广义表顶层横扫
        if (temp->utype == 2) {     //扫描到表结点
            n = Depth(temp->info.hlink);
             //递归计算以子表深度
           if (m < n) m = n;            //取最大深度
        }
        temp = temp->tlink;
    }
    return m+1;                         //返回深度
};
```

广义表的删除算法



- 扫描子链表
  - 若结点数据为*c*, 删除。可能做循环连续删。
  - 若结点数据不为*c*，不执行删除。
  - 若结点为子表，递归在子表执行删除。

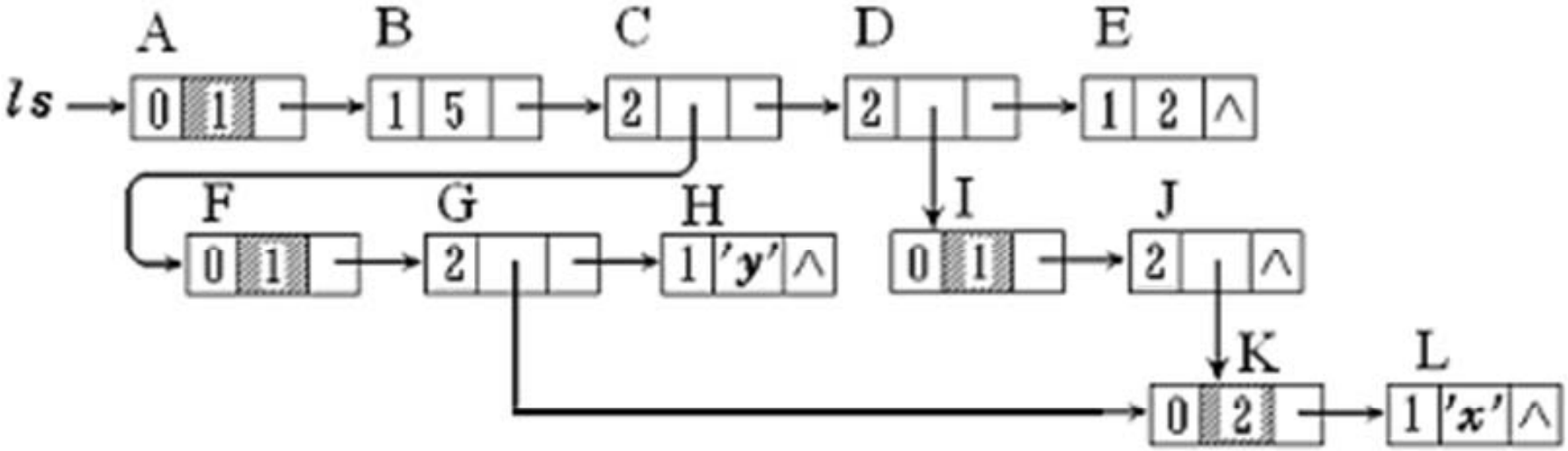
```
template <class T>
void delvalue(GenListNode<T> *ls, T x) {
    if (ls->tlink != NULL) {                    //非空表
        GenListNode<T> * p = ls->tlink;  //第一个结点
        while (p != NULL && (p->utype == 1 &&
            p->info.value == x)) {
            ls->tlink = p->tlink;  delete p;
            p = ls->tlink;              //p指向同层下一结点
        }
        if (p != NULL) {
            if (p->utype == 2)        //递归在子表中删除
                delvalue(p->info.hlink, x);
                delvalue(p, x);
                //在以p为表头的链表中递归删除
        }
    }
};
```

- 删除子表：
  - 对于共享表来说，如果一个表元素有多个地方使用它，贸然删去它会造成其他地方使用出错。因此，当要做删除时，先把该表的头结点中的引用计数 **ref** 减**1**，当引用计数减到 **0** 时才能执行结点的真正释放。

```
template <class T>
GenList<T>::～GenList() {
//广义表的析构函数,每个头结点都有引用计数
    Remove(first);
};
```

```
template <class T>
void GenList<T>::Remove(GenListNode<T> *ls) {
//私有函数：释放以ls为表头指针的广义表
    ls->info.ref--;                  //头结点的引用计数减1
    if (ls->info.ref <= 0) {                  //如果减到0
        GenListNode<T> *q;
        while (ls->tlink != NULL) {    //横扫表顶层
            q = ls->tlink;                          //到第一个结点
            if (q->utype == 2) {           //递归删除子表
                Remove(q->info.hlink);
                if (q->info.hlink->info.ref <= 0)
                    delete q->info.hlink;  //删子表头结点
            }
            ls->tlink = q->tlink;  delete q;
        }
    }
};
```

- 例：给出下面的广义表进行删除时的结点删除的顺序



建立广义表的链表表示

设广义表的字符串描述如下

***S*(A (‘*b*’, ‘c’), B(‘#’), ‘*d*’);**

据字符串描述生成广义表链表结构

- 假设一个广义表的元素类型 **T** 是字符类型 **char**，每一个原子元素的值为英文字母，并假定广义表从输入流对象 **istream** 输入。

- 在算法的执行过程中，检测从输入流对象输入的一个字符，如果遇到表名（用大写字母表示），首先检查这个表名是否已经存在，如果是，说明该表是共享表，只要将相应头结点的引用计数加一即可；如果不是，保存该表名并建立相应广义表。表名后面一定是左括号‘(’，不是则输入错，是则递归建立广义表结构。

- 如果遇到用小写字母表示的原子，则建立原子结点；如果遇到右括号‘)’，子表链收尾并退出递归。

- 注意在空表情形括号里应夹入一个非英文字母，如字符'#'，不能一个字符也没有。整个广义表描述字符串以';'结束。

由字符串建立广义表的链表表示算法

```
#include <string.h>
template <class T>
void Genlist<T>::CreateList(istream &in,
    GenListNode<T> *& ls, SeqList<T>& L1,
    SeqList <GenListNode<T> *>& L2) {
//从广义表的字符串描述 s 出发, 建立一个带头结
//点的广义表，要求T为char类型。在表L1存储大
//写字母的表名, 在表L2存储表名对应子表结点的
//地址。
     T chr;
     in >> chr;
```

```
 //读入一个字符，只可能读入#、左括号和字母
if (isalpha(chr) && isupper(chr) || chr == '(') {
 //大写字母或'('
    ls = new GenListNode<T>;        //建子表结点
    ls->utype = 2;
    if (isalpha(chr) && isupper(chr)) {  //表名处理
        int n = L1.Length();
        int m = L1.Search(chr);
        if (m != 0) {                  //该表已建立
            GenListNode<T> *p = L2.Locate(m);
             //查子表地址
            ls->info.hlink = p;
            p->ref++;                  //引用计数加一
            CreateList(in, ls->tlink, L1, L2);
            //递归建后继表
        }
```

```
        else {      //该表未建立
            L1.Insert(n, chr);  L2.Insert(n, ls);
                    //保存表名及地址
            in >> chr;
            if (chr != '(') exit(1); //表名后必跟'('
            ls->info.hlink = new GenListNode<T>;
            ls->info.hlink->utype = 0;     //建头结点
            ls->info.hlink->ref = 1;
            CreateList(in, ls->info.hlink->tlink, L1,
L2);
        //递归建子表
         }
```

```
    }
    else if (isalpha(chr) && islower(chr)) {
        //建原子结点
        ls = new GenListNode<T>;
        ls->utype = 1;  ls->info.value = chr;
        CreateList(in, ls, L1, L2);
    }
    else if (chr == ', ')                    //建后继结点
        CreateList(in, ls->tlink, L1, L2);
    else if (chr == ')') ls->tlink = NULL;  //链收尾
    else if (chr == '#') ls == NULL;          //空表, 链收尾
};
```

- 该算法需要扫描输入广义表中的所有字符，且处理每个字符都是简单比较或赋值操作，其时间复杂度为**O(1)**，所以整个算法的时间复杂度为**O(n)**，**n**是广义表中所有字符数。

- 算法中既包含向子表的递归调用，也包含向后继表的递归调用，所以递归调用的最大深度不会超过生成的广义表中所有结点数，其空间复杂度也为**O(n)**。

```
template <class T>
istream& operator >> (istream& in, GenList<T>& L)
{
    int n;
    cout << "输入广义表串的字符个数：" << endl;
    in >> n;
    T *Ls1 = new T[n];          //创建辅助数组并初始化
    GenListNode<T> *Ls2 = new GenListNode<T>[n];
    CreateList (in, L.first, Ls1, Ls2);    //建立存储结构
    delete [ ]Ls1;  delete [ ]Ls2;
};
```