

第四章 数组、串与广义表

- 多维数组的概念与存储
- 特殊矩阵
- 稀疏矩阵
- 字符串
- 广义表
- 本章小结

4.1 多维数组的概念与存储

- 定义

相同类型的数据元素的集合。

- 一维数组的示例

一
维
数
组

0	1	2	3	4	5	6	7	8	9
35	27	49	18	60	54	77	83	41	02

- 在高级语言中的一维数组只能按元素的下标直接存储和访问数组元素。

数组的定义和初始化

```
#include <iostream.h>  
class Szcl {  
    int e;  
public:  
    Szcl ( ) { e = 0; }  
    Szcl ( int value ) { e = value; }  
    int Get_Value ( ) { return e; }  
};
```

```
int main ( ) {  
    Szcl a1[3] = { 3, 5, 7 }, *elem;  
    for ( int i = 0; i < 3; i++ )  
        cout << a1[i].Get_Value ( ) << “\n” ; //静态  
    elem = a1;  
    for ( int i = 0; i < 3; i++ ) {  
        cout << elem->Get_Value( ) << “\n” ; //动态  
        elem++;  
    }  
    return 0;  
}
```

一维数组(Array)类的定义

```
#include <iostream.h>
#include <stdlib.h>
const int DefaultSize = 30;
template <class Type> class Array {
    Type *elements; //数组存放空间
    int ArraySize; //当前长度
    void GetArray ( ); //建立数组空间
public:
    Array( int size = DefaultSize );
    Array( const Array <Type> &x );
```

```
~Array( ) { delete [ ] elements; }  
Array <Type> & operator = //数组复制  
    ( const Array <Type> &A );  
Type & operator [ ] ( int i ); //取元素值  
int Length ( ) const { return ArraySize; }  
//取数组长度  
void ReSize ( int sz ); //扩充数组  
};
```

一维数组公共操作的实现

```
template <class Type>
void Array <Type> :: GetArray ( ) {
//私有函数：创建数组存储空间
    elements = new Type [ArraySize];
    if ( elements == NULL ) {
        ArraySize = 0;
        cerr << “存储分配错！” << endl;
        return;
    }
}
```

```
template <class Type>  
Array <Type> :: Array ( int sz ) {  
//构造函数  
    if ( sz <= 0 ) {  
        ArraySize = 0;  
        cerr << “非法数组大小” << endl;  
        return;  
    }  
    ArraySize = sz;  
    GetArray ( );  
}
```



```
template <class Type> Array <Type> ::  
Array ( Array <Type> &x ) { //复制构造函数  
    int n = ArraySize = x.ArraySize;  
    elements = new Type [n];  
    if ( elements == NULL ) {  
        ArraySize = 0;  
        cerr << “存储分配错” << endl;  
        return;  
    }  
    Type *srcptr = x.elements;  
    Type *destptr = elements;  
    while ( n-- ) *destptr++ = *srcptr++;  
}
```

```
template <class Type>
Type & Array <Type> :: operator [ ] ( int i ) {
    //按数组名及下标 i , 取数组元素的值
    if ( i < 0 || i > ArraySize-1 ) {
        cerr << “数组下标超界” << endl;
        return NULL;
    }
    return elements[i];
}
```

使用该函数于赋值语句

$\text{Pos} = \text{Position}[i] + \text{Number}[i]$

```
template <class Type>  
void Array <Type> :: Resize ( int sz ) {  
    if ( sz >= 0 && sz != ArraySize ) {  
        Type *newarray = new Type [sz];  
        //创建新数组  
        if ( newarray == NULL ) {  
            cerr << “存储分配错” << endl;  
            return;  
        }  
        int n = ( sz <= ArraySize ) ? sz : ArraySize;  
        //按新的大小确定传送元素个数
```

```
Type *srcptr = elements; //源数组指针
Type *destptr = newarray; //目标数组指针
while ( n-- ) *destptr++ = *srcptr++;
//从源数组向目标数组传送
delete [ ] elements;
elements = newarray;
ArraySize = sz;
    }
}
```

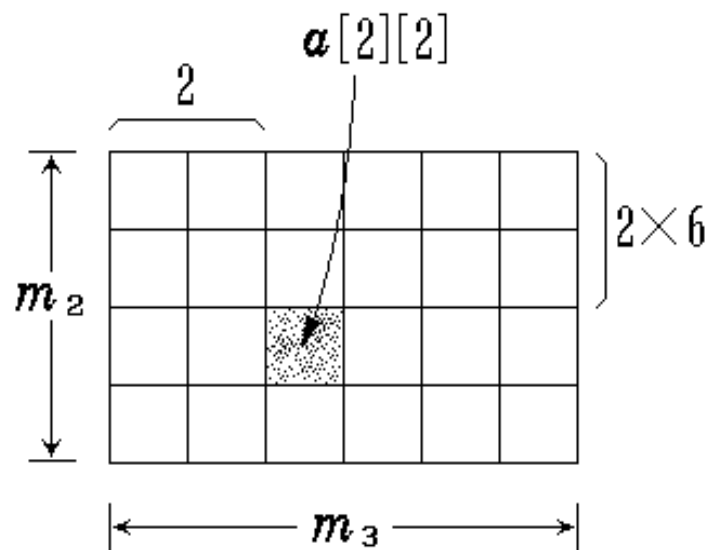
多维数组

- 多维数组是一维数组的推广。
- 从逻辑关系上讲，多维数组是一种非线性结构。其特点是**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，因此它比其它复杂的非线性结构简单。

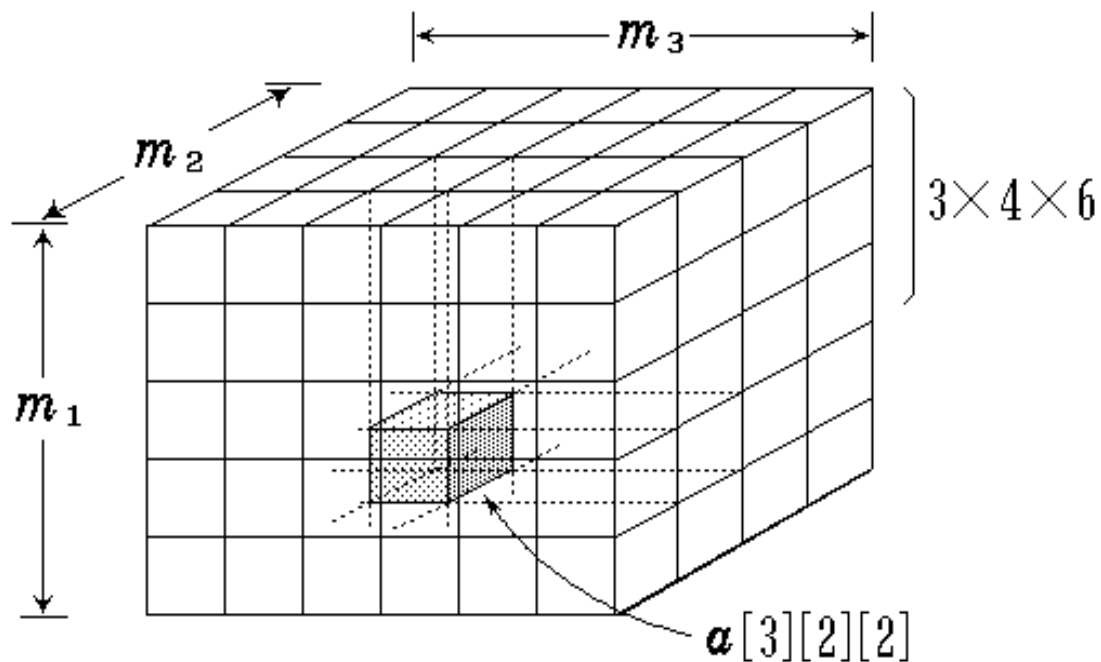
二维数组

三维数组

$$m_1 = 5 \quad m_2 = 4 \quad m_3 = 6$$



行向量 下标 i
列向量 下标 j

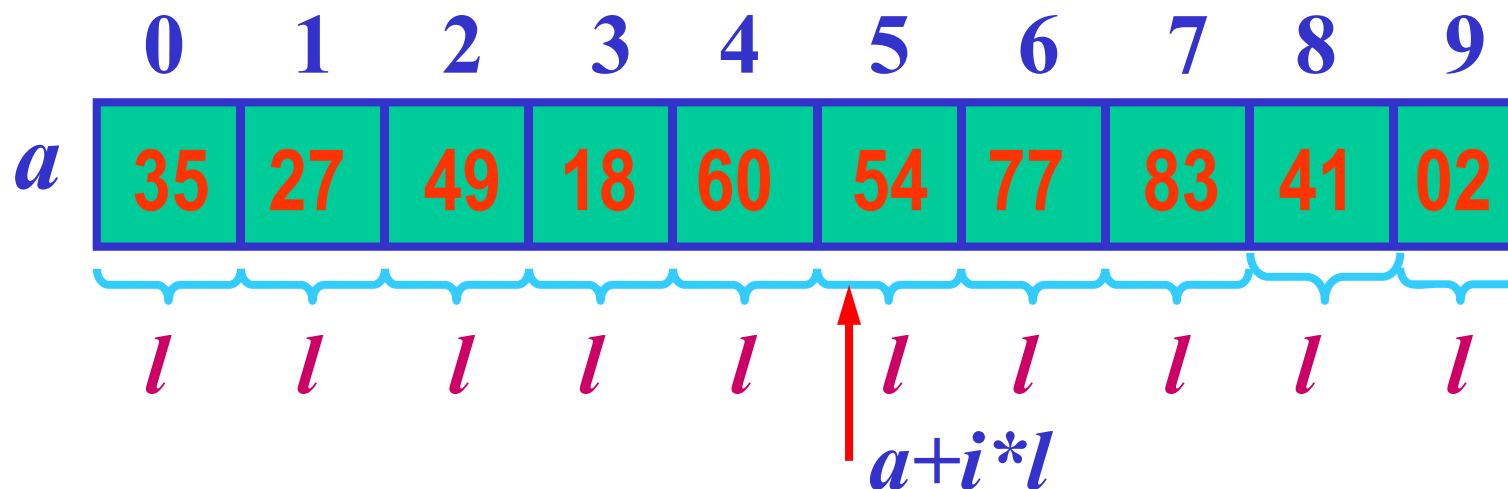


页向量 下标 i
行向量 下标 j
列向量 下标 k

数组的连续存储方式

■ 一维数组

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



$$\text{LOC}(i) = \text{LOC}(i-1) + l = a + i * l$$

二维数组

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

行优先存放:

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

$$\text{LOC}(j, k) = a + (j * m + k) * l$$

二维数组

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

列优先存放:

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

$$\text{LOC}(j, k) = a + (k * n + j) * l$$

■ 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)

$$\text{LOC} (i_1, i_2, i_3) = a + (i_1 * m_2 * m_3 + i_2 * m_3 + i_3) * l$$

前 i_1 页总
元素个数

第 i_1 页的
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 $m_1, m_2, m_3, \dots, m_n$ 。

➡ 下标为 $i_1, i_2, i_3, \dots, i_n$ 的数组元素的存储地址：

$$\begin{aligned} \text{LOC} (i_1, i_2, \dots, i_n) &= a + \\ &\quad (i_1 * m_2 * m_3 * \dots * m_n + i_2 * m_3 * m_4 * \dots * m_n \\ &\quad + \dots + i_{n-1} * m_n + i_n) * l \\ &= a + \left(\sum_{j=1}^{n-1} i_j * \prod_{k=j+1}^n m_k + i_n \right) * l \end{aligned}$$



随堂练习

例1：二维数组A的每个元素是由6个字符组成的串，其行下标 $i=0, 1, \dots, 8$ ，列下标 $j=1, 2, \dots, 10$ 。若A按行先存储，元素A[8, 5]的起始地址与当A按列先存储时的元素的哪一个元素的起始地址相同。设每个字符占一个字节。

例2：若三维数组M[2..3, -4..2, -1..4]中，每个元素占用2个存储单元，起始地址为100，则如果按页优先顺序存储，M[3][-3][3]的存储地址为_____。

例1：二维数组A的每个元素是由6个字符组成的串，其行下标*i*=0, 1, ..., 8，列下标*j*=1, 2, ..., 10。若A按行先存储，元素A[8, 5]的起始地址与当A按列先存储时的元素的哪一个元素的起始地址相同。设每个字符占一个字节。

设二维数组A[c1..d1, c2..d2]，每个数据元素占L个字节，则：

行优先存储地址计算公式：

$$\text{Loc}(a_{ij}) = \text{Loc}(a_{c_1c_2}) + [(i - c_1) * (d_2 - c_2 + 1) + (j - c_2)] * L$$

列优先存储地址计算公式：

$$\text{Loc}(a_{ij}) = \text{Loc}(a_{c_1c_2}) + [(j - c_2) * (d_1 - c_1 + 1) + (i - c_1)] * L$$

$$(8 - 0) * (10 - 1 + 1) + (5 - 1) = (j - 1) * (8 - 0 + 1) + (i - 0)$$

$$\text{即 } 9(j - 1) + (i - 0) = 84$$

推出A[3, 10]。

例2：若三维数组M[2..3, -4..2, -1..4]中，每个元素占用2个存储单元，起始地址为100，则如果按页优先顺序存储，M[3][-3][3]的存储地址为(204)。

设三维数组A[c1..d1, c2..d2, c3..d3]，每个数据元素占L个字节，则：

页优先存储地址计算公式：

$$\text{Loc}(aijk) = \text{Loc}(ac_1c_2c_3) + [(i-c_1)*(d_2-c_2+1)(d_3-c_3+1) + (j-c_2)*(d_3-c_3+1) + (k-c_3)] * L$$

行优先存储地址计算公式：

$$\text{Loc}(aijk) = \text{Loc}(ac_1c_2c_3) + [(j-c_2)*(d_1-c_1+1)(d_3-c_3+1) + (i-c_1)*(d_3-c_3+1) + (k-c_3)] * L$$

列优先存储地址计算公式：

$$\text{Loc}(aijk) = \text{Loc}(ac_1c_2c_3) + [(k-c_3)*(d_1-c_1+1)(d_2-c_2+1) + (i-c_1)*(d_2-c_2+1) + (j-c_1)] * L$$



4.2 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 $n \times n$ 的对称矩阵 A :

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

在矩阵中, $a_{ij} = a_{ji}$

- 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 $n + (n-1) + \cdots + 1 = n*(n+1)/2$ 个元素。

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

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}

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，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 $j = 8 - 3 * 4 / 2 = 2$ 。

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

	0	1	2	3	4	5	6	7	8	9
B	a_{00}	a_{01}	a_{02}	a_{03}	a_{11}	a_{12}	a_{13}	a_{22}	a_{23}	a_{33}

若 $i \leq j$, 数组元素 $A[i][j]$ 在数组 B 中的存放位置为 $\underbrace{n + (n-1) + (n-2) + \cdots + (n-i+1)}_{\text{前 } i \text{ 行元素总数}} + \underbrace{j-i}_{\text{第 } i \text{ 行第 } j \text{ 个元素前元素个数}}$

前 i 行元素总数

第 i 行第 j 个元素前元素个数

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

$$\begin{aligned} & n + (n-1) + (n-2) + \cdots + (n-i+1) + j - i = \\ & = (2 * n - i + 1) * i / 2 + j - i \\ & = (2 * n - i - 1) * i / 2 + j \end{aligned}$$

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

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

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}	\dots	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。
- 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
$$0 \leq i \leq n-1, i-1 \leq j \leq i+1$$
- 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,

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



随堂练习

例1：将一个 $A[1..100, 1..100]$ 的三对角矩阵，按行优先存入一维数组 $B[1..298]$ 中， A 中元素 $A[66,65]$ （即该元素下标 $i=66$ ， $j=65$ ），在 B 数组中的位置 K 为_____。

例2：五对角矩阵中 k 与 i 、 j 的关系。

例1： 将一个 **$A[1..100, 1..100]$** 的三对角矩阵，按行优先存入一维数组 **$B[1..298]$** 中， **A** 中元素 **$A[66,65]$** （即该元素下标 **$i=66$** ， **$j=65$** ），在 **B** 数组中的位置 **K** 为 **195**。

例2： 五对角矩阵中 **k** 与 **i** 、 **j** 的关系。

$$k = \begin{cases} 4(i-1) + j & i = 1 \\ 4(i-1) + j - 1 & 1 < i < n \\ 4(i-1) + j - 2 & i = n \end{cases}$$

4.3 稀疏矩阵 (Sparse Matrix)

$$\mathbf{A}_{6 \times 7} = \begin{pmatrix} 0 & 0 & 0 & 22 & 0 & 0 & 15 \\ 0 & 11 & 0 & 0 & 0 & 17 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 39 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 & 0 \end{pmatrix}$$

非零元素个数远远少于矩阵元素个数

稀疏矩阵的定义

- 设矩阵 $A_{m \times n}$ 中有 t 个非零元素，若 t 远远小于矩阵元素的总数 $m \times n$ ，则称矩阵 A 为稀疏矩阵。
- 为节省存储空间，应只存储非零元素。
- 非零元素的分布一般没有规律，应在存储非零元素时，同时存储该非零元素的行下标 row 、列下标 col 、值 $value$ 。
- 每一个非零元素由一个三元组唯一确定：
(行号 row , 列号 col , 值 $value$)

稀疏矩阵类定义

```
template <class Type> class SparseMatrix;  
template <class Type> class Trituple { //三元组  
    friend class SparseMatrix;  
    private:  
        int row, col; //非零元素行号/列号  
        Type value; //非零元素的值  
    public:  
        Trituple <Type> & operator =  
            ( Trituple <Type> &x )  
        { row = x.row; col = x.col; value = x.value; }  
};
```

```
template <class Type> class SparseMatrix {  
    int Rows, Cols, Terms; //行/列/非零元素数  
    Trituple <Type> *sm_Array;  
    int maxTerms;  
public: //三元组表  
    SparseMatrix ( int maxSize = DefaultSize );  
    SparseMatrix ( SparseMatrix <Type> &x );  
    ~SparseMatrix ( ) { delete [ ] smArray; }  
    SparseMatrix <Type> Transpose  
        ( SparseMatrix <Type> & ); //转置  
    SparseMatrix <Type> Add ( SparseMatrix <Type> &b ); //相加  
    SparseMatrix <Type> Multiply ( SparseMatrix <Type> &b );  
    //相乘  
};
```

稀疏矩阵的转置

- 一个 $m \times n$ 的矩阵 A ，其转置矩阵 B 是一个 $n \times m$ 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

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

[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]

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

转置矩阵

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

	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

	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

	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 次检测列号为 k 的项。
- 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 Type> SparseMatrix <Type>
SparseMatrix <Type> :: Transpose ( ) {
    SparseMatrix <Type> b(maxTerms);
    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
```

```
for ( int k = 0; k < Cols; k++ )  
    //对所有列号处理一遍  
    for ( int 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++;  
        }  
    }  
return b;  
}
```

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 $O(n * t)$ 。
- 若矩阵有 200 行、200 列、10,000 个非零元素，总共有 2,000,000 次处理。
- 为加速转置速度，建立辅助数组 `rowSize` 和 `rowStart`，记录矩阵转置后各行非零元素个数和各行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

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

```
template <class Type> SparseMatrix <Type>  
SparseMatrix <Type> :: FastTranspos ( ) {  
    int *rowSize = new int [Cols];  
    int *rowStart = new int [Cols];  
    SparseMatrix <Type> b(maxTerms);  
    b.Rows = Cols; b.Cols = Rows;  
    b.Terms = Terms;  
    if ( Terms > 0 ) {  
        for ( int 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++ ) {  
    int 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;  
return b;  
}
```

矩阵相加算法

- 前提——两矩阵大小相同，行数和列数对应相等。
- 算法主要思想

- 比较矩阵元素 $a.smArray[i]$ 和 $b.smArray[j]$ 在原矩阵中的位置。

$index_a = a.smArray[i].row * Cols + a.smArray[i].col$

$index_b = b.smArray[j].row * Cols + b.smArray[j].col$

- 若 $index_a > index_b$ ，则往结果矩阵中添加 $b.smArray[j]$ ；若 $index_a < index_b$ ，则往结果矩阵中添加 $a.smArray[i]$ ；若 $index_a = index_b$ ，则相加之后非零，往结果矩阵中添加和值 $(a.smArray[i].value + b.smArray[j].value)$ 。

稀疏矩阵的相加算法

```
template <class Type> SparseMatrix <Type>
SparseMatrix <Type> :: Add ( SparseMatrix <Type> b ) {
    Sparsematrix <Type> Result(Rows*Cols);
    if ( Rows != b.Rows || Cols != b.Cols ) {
        cout << “Imcompatible matrices” << endl;
        return result;
    }
    int i = 0, j = 0, index_a, index_b;
    Result.Terms = 0;
```

```
while ( i < Terms && j < b.Terms ) {  
    index_a = Cols * smArray[i].row + smArray[i].col;  
    index_b = Cols * b.smArray[j].row + b.smArray[j].col;  
    if ( index_a < index_b ) {  
        Result.smArray[Result.Terms] = smArray[i];  
        i++;    }  
    if ( index_a > index_b ) {  
        Result.smArray[Result.Terms] = b.smArray[j];  
        j++;    }  
    if ( index_a == index_b ) {  
        if ( ( smArray[i].value + b.smArray[j].value ) != 0 ) {  
            Result.smArray[Result.Terms] = smArray[i];  
            Result.smArray[Result.Terms].value =  
                smArray[i].value + b.smArray[j].value;    }  
        i++; j++;    }  
    Result.Terms++;  
}
```

```
for ( ; i < Terms; i++ ) {  
    Result.smArray[Result.Terms] = smArray[i];  
    Result.Terms++;  
}  
for ( ; j < b.Terms; j++ ) {  
    Result.smArray[Result.Terms] = b.smArray[j];  
    Result.Terms++;  
}  
}
```

带行指针数组的二元组表

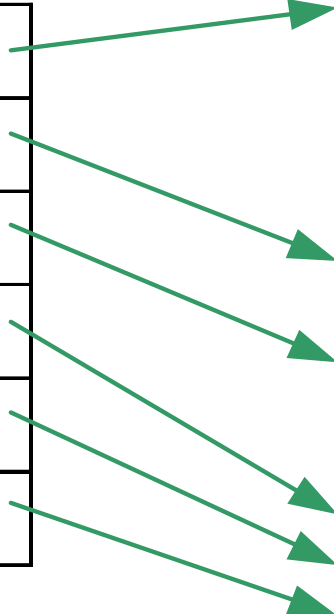
- 稀疏矩阵的三元组表可以用带行指针数组的二元组表代替。
- 在行指针数组中元素个数与矩阵行数相等。
第 i 个元素的下标 i 代表矩阵的第 i 行，
元素的内容即为稀疏矩阵第 i 行的第一个
非零元素在二元组表中的存放位置。

- 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。

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

行指针数组

	row
0	0
1	3
2	4
3	6
4	7
5	7



二元组表 data

	col	value
0	0	12
1	2	11
2	5	13
3	6	14
4	1	-4
5	5	3
6	3	8
7	1	-9
8	4	2



4.4 字符串 (String)

字符串是 n (≥ 0) 个字符的有限序列。

记作 $S : "c_1c_2c_3 \dots c_n"$

其中, S 是串名字

$"c_1c_2c_3 \dots c_n"$ 是串值

c_i 是串中字符

n 是串的长度

例如, $S = \text{"Tsinghua University"}$

串的定义

- 假设 V 是程序设计语言所采用的字符集，由字符集 V 上的字符所组成的任何有限序列，称为字符串（或简称为串）：

$$s = "a_1 a_2 \dots a_n" \quad (n \geq 0)$$

其中：

- s 是串的名；
- 两个双引号之间的字符序列 " $a_1 a_2 \dots a_n$ " 是串的值；
- $a_i \in V \ (1 \leq i \leq n)$ 是字符集上的字符。
- 串中字符的数目 n 称为串的长度。长度为 0 的串称为空串。
- 一个串的子串是这个串中的任一连续子序列。包含子串的串相应地称为主串。
- 通常称字符在序列中出现的序号为该字符在串中的位置。相应地，子串在主串中的位置则以该子串的第一个字符在主串中的位置来表示。

串的逻辑结构和基本操作

- 在逻辑结构方面，串与线性表极为相似，区别在于串的数据对象约束为字符集。
- 字符串的逻辑表示：
 - 数据对象： $D=\{a_i|a_i\in\text{字符集}, i=1, 2, \dots, n, n\geq 0\}$
 - 数据关系： $R=\{<a_{j-1}, a_j>|(a_{j-1}\in D)\wedge(a_j\in D), 2\leq j\leq n\}$

- 在基本操作方面，串与线性表差别很大。
 - 线性表：大多以"单个元素"为操作对象，如：在线性表中查找某个元素、在某个位置上插入一个元素或删除一个元素等；
 - 串：通常以"串的整体"作为操作对象，如：在串中查找某个子串、在串的某个位置上插入一个子串或删除一个子串等。
- 字符串数据结构需支持的基本操作：
 1. **Assign** 字符串的初始化，用字符串常量为当前字符串初始化。
 2. **GetLen** 获得字符串的长度。
 3. **IsEmpty** 判断当前字符串是否为空串。
 4. **Empty** 清空当前字符串，即使当前串为空串。
 5. **Comp** 比较两个字符串是否相等。
 6. **Concat** 将两个字符串拼接在一起。
 7. **SubString** 获得位置 $npos$ 开始，长度为 $nCount$ 的子串。
 8. **Find** 获得字符串中子串的出现位置。
- 除上述基本操作外，字符串上的操作还包括**Insert**、**Delete**、**Replace**等。相对串上的基本操作，这些操作相对比较复杂。

串的存储结构

1. 数组存储
2. 块链存储

串的数组存储表示

(1) 静态数组存储方法

- 为字符串变量分配一个固定长度的存储空间
- 一般用定长数组加以实现

```
const int nMaxLen=1024; //字符串的最大长度
```

```
class SString {
```

```
private:
```

```
    int nLen; //字符串的当前长度
```

```
    char ch[nMaxLen+1]; //字符串存储空间
```

```
};
```

(2) 动态数组存储方法

- 使用C++提供的**new**操作符分配一块连续的堆空间

```
class Dstring {
```

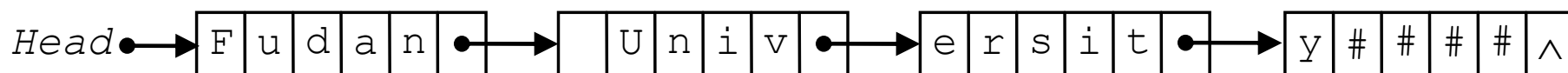
```
private:
```

```
    char *ch; //指向当前字符串的指针
```

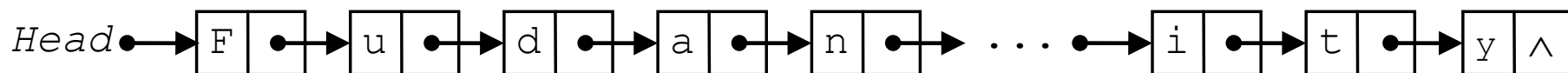
```
    int nLen; //字符串长度
```

```
};
```

串的块链存储表示



(a) 结点大小为 5



(b) 结点大小为 1

图3-1 字符串"*Fudan University*"的块链存储方式

- 结点大小的选择和子串存储方式是影响效率的重要因素

$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

- 块链式存储结构对字符串的连接等操作的实现比较方便，但总体来说操作复杂，占用的存储空间大。
- 块链存储结构字符串的操作实现，与线性表的链表实现类似。

字符串类定义

```
const int maxLen = 128;  
class AString {  
    int curLength; //串是当前长度  
    char *ch; //串的存储数组  
    int maxSize;  
public:  
    AString ( const AString &ob );  
    AString ( const char *init );  
    AString ( int sz = defaultSize );  
    ~AString ( ) { delete [ ] ch; }
```

```
int Length ( ) const { return curLength; }  
//求当前串 *this 的实际长度  
AString & operator ( ) ( int pos, int len );  
//取 *this 从 pos 开始 len 个字符组成的子串  
int operator == ( const AString &ob ) const  
    { return strcmp ( ch, ob.ch ) == 0; }  
//判当前串 *this 与对象串 ob 是否相等  
int operator != ( const AString &ob ) const  
    const { return strcmp ( ch, ob.ch ) != 0; }  
//判当前串 *this 与对象串 ob 是否不等
```

```
int operator ! ( )  
    const { return curLength == 0; }  
//判当前串 *this 是否空串  
AString & operator = ( AString &ob );  
//将串 ob 赋给当前串 *this  
AString & operator += ( AString &ob );  
//将串 ob 连接到当前串 *this 之后  
char & operator [ ] ( int i );  
//取当前串 *this 的第 i 个字符  
int Find ( AString &pat ) const;  
};
```

字符串部分操作的实现

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

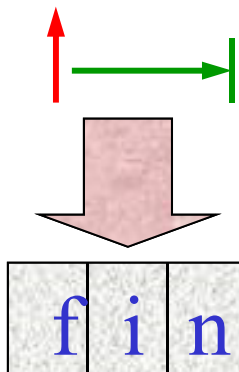
```
AString :: AString ( const char *init ) {  
  //构造函数：从已有字符数组*init复制  
  ch = new char [maxSize]; //创建串数组  
  if ( ch == NULL ){  
    cerr << “存储分配错 ! \n”;  
    exit(1);  
  }  
  curLength = strlen ( init ); //复制串长度  
  strcpy ( ch, init ); //复制串值  
}
```

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

提取子串的算法示例

pos = 2, len = 3

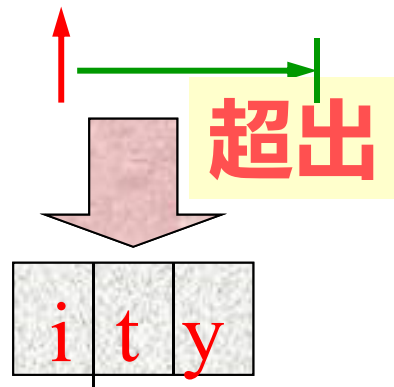
i	n	f	i	n	i	t	y
---	---	---	---	---	---	---	---



$\text{pos} + \text{len} - 1$
 $\leq \text{curLength} - 1$

pos = 5, len = 4

i	n	f	i	n	i	t	y
---	---	---	---	---	---	---	---



$\text{pos} + \text{len} - 1$
 $\geq \text{curLength}$


```
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; //建立对象  
    if ( pos < 0 || pos+len-1 >= maxSize || len < 0 ) {  
        temp.curLength = 0; //返回空串  
        temp.ch[0] = '\0';  
    }  
    else { //提取子串  
        if ( pos+len-1 >= curLength )  
            len = curLength - pos;
```

```
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”

```
String & String :: operator = ( String &ob ) {  
    //串赋值：从已有串 ob 复制  
    if ( &ob != this ) {  
        delete [ ] ch;  
        ch = new char [maxSize+1]; //重新分配  
        if ( ch == NULL )  
            { cerr << “内存不足！ \n”; exit (1); }  
        curLength = ob.curLength; //串复制  
        strcpy ( ch, ob.ch );  
    }  
    else cout << “字符串自身赋值出错！ \n”;  
    return *this;  
}
```

```
char & String::operator [ ] ( int i ) {  
//按串名提取串中第 i 个字符  
    if ( i < 0 && i >= curLen )  
        { cout << “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 newSt = “university”

提取字符 newChar = ‘n’

```
AString & AString :: operator += ( String &ob ) {
```

```
//串连接
```

```
    char *temp = ch; //暂存原串数组
```

```
    curLength += ob.curLenth; //串长度累加
```

```
    ch = new char [maxSize+1];
```

```
    if ( ch == NULL )
```

```
        { cerr << “存储分配错! \n”; exit (1); }
```

```
    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 = “beijing ”`,
 `st2 = “university”`,

使用示例 `st1 += st2;`

连接结果 `st1 = “beijing university”`
 `st2 = “university”`

随堂练习

例1：设**S**为一个长度为**n**的字符串，其中的字符各不相同，求解**S**中互异的非平凡子串（非空且不同于**S**本身）的个数。

例2：若串**S="software"**，求解其子串（不含空串）数目。

例3：如果字符串的一个子串（其长度大于1）的各个字符均相同，则称之为等值子串。试设计一个算法，输入字符串**S**，以"!"作为结束标志。如果串**S**中不存在等值子串，则输出信息"无等值子串"，否则求出（输出）一个长度最大的等值子串。

例4：编写一个算法**frequency**，统计在一个输入字符串中各个不同字符出现的频度。用适当的测试方法来验证这个算法。

例1：设**S**为一个长度为**n**的字符串，其中的字符各不相同，求解**S**中互异的非平凡子串（非空且不同于**S**本身）的个数。

除长度为**n**的子串外，长度为**n-1**的不同子串个数为**2**，长度为**n-2**的不同子串个数为**3**，依此类推，直至长度为**1**的不同子串个数为**n**，即**S**的非平凡子串个数为： $2+3+\dots+n=n(n+1)/2-1$ 。

例2：若串**S="software"**，求解其子串（不含空串）数目。

已知**S**串长度为**8**，则其子串个数为 $1+2+3+\dots+8=8(8+1)/2=36$ 。

例3:

先从键盘上接受字符串并送入字符串数组 S，然后扫描字符串数组 S。设变量 head 指向当前发现的最长等值子串的串头，max 记录此子串的长度。扫描过程中，若发现等值子串则用 count 变量记录其长度，如果它的长度大于原最长等值子串的长度，则对 head 和 max 进行更新。重复上述过程直到 S 的末尾。最后，根据扫描所得的结果输出最长等值子串或输出等值子串不存在信息。

```
void Equstring (char s[ ])
{
    for(k=0; ; k++)
    {
        scanf("%c", &S[k]);
        if (S[k]=='!')
            break;
    }
    for (i=0, j=1, head=0, max=1; S[i]!='!' && S[j]!='!'; i=j, j++)
    {
        count=1;
        while (S[i]==S[j])
        {
            j++;
            count++;
        }
        if (count>max)
        {
            head=i;
            max=count;
        }
    }
    if (max>1)
        for (k=head; k<(head+max); k++)
            printf("%c", S[k]);
    else
        printf("There is no equivalent substring in S!");
}
```

例4:

```
void frequency(String &s) {  
    int LenOfChars=0; //不相同的字符个数  
    int freqs= new int[maxLen+1];  
    char *temp= new char[maxLen+1];  
    for (int j=0; j<s->length( ); j++) {  
        for (int k=0; k<LenOfChars; k++){  
            if (temp[k] == s[j]){  
                freqs[k]++; break;  
            }  
        }  
        if ( k == LenOfChars) {  
            temp[LenOfChars] = s[j];  
            freqs[LenOfChars++] = 1;  
        }  
    }  
    cout << "Results are: " << endl;  
    for( int j=0; j<LenOfChars; j++)  
        cout << temp[j] << " is: " << freqs[j] << endl;  
    cout << "End of Print!" << endl;  
}
```

串的模式匹配

- **定义** 在串中寻找子串（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**，串称为**目标**。
- **示例** 目标 T：“Beijing”
模式 P：“jin”
匹配结果 = 3

模式匹配的哲理

- 词汇 目标: “外面的世界”
模式: “小小的我”
- 定义 在外面的世界中寻找
小小的我的“位置”。

第1趟

T a b b a b a
P a b a

穷举模式
匹配过程

第2趟

T a b b a b a
P a b a

第3趟

T a b b a b a
P a b a

第4趟

T a b b a b a
P a b a

✓

用*P*中的字符依次与

***T*中的字符做比较**

```
int AString::Find ( AString &pat ) const {  
    //穷举的模式匹配  
    char *p = pat.ch, *s = ch; int i = 0;  
    while ( *p != '\0' && *s != '\0' )  
        //当两串未检测完  
        if ( *p++ != *s++ ) //比较串字符, 不等  
            { i++; s = ch + i; p = pat.ch; }  
        //对应字符不相等, 对齐目标的下一位置  
        //继续比较  
    if ( *p == '\0' ) return i; //相等  
    else return -1;  
}
```

■ 穷举的模式匹配算法分析（时间代价）

设模式串 P 有 m 个字符，而目标串 T 有 n 个字符：

最好情况— T 的前 m 个字符与 P 匹配，可在 m 次比较后找到匹配结果，算法复杂度为 $O(m)$ 。

最坏情况—假设未进行比较最后一个字符之类的优化，且第一个字符总是能匹配上，但却永远没有匹配上的模式。如： $P=\text{"abc"}$ ， $T=\text{"aaaaaaaaa"}$ ($m=3$ ， $n=8$)，则模式 $P=\text{"abc"}$ 的 m ($m=3$)个字符必须与 T 中的文本块一共比较 $n-m+1$ 次，一般情况下必须比较 m 个字符共 $(n-m+1)$ 次，即一共进行 $m(n-m+1)$ 次。

而 $m(n-m+1) \leq m(n-m+m) = mn$
算法复杂度的估计值为 $O(mn)$ 。

■ 算法速度慢的原因在于回溯

每趟重新比较时，目标串的检测指针要回退。

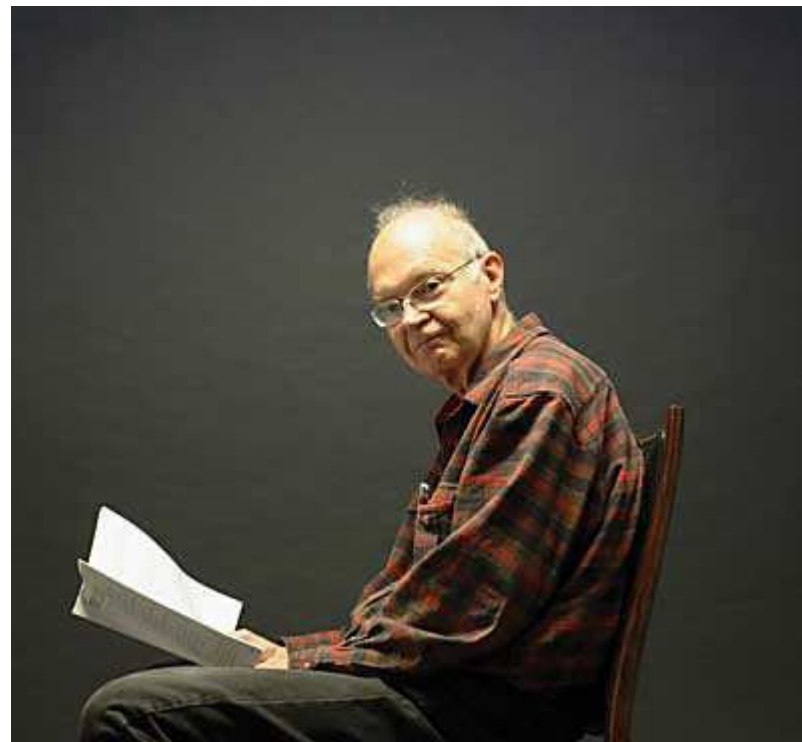
BF算法的特点总结

- (1) 不需要预处理过程;
- (2) 只需固定的存储空间;
- (3) 滑动窗口的移动每次都为**1**;
- (4) 字符串的比对可按任意顺序进行（从左到右、从右到左、或特定顺序均可）;
- (5) 算法的时间复杂性为 **$O(m \times n)$** ;
- (6) 最大比较次数为 **$(n-m+1) \times m$** 。

KMP (Knuth-Morris-Pratt)算法

- **BF**和**KR**算法的一个共同点是每次比较的窗口向右滑动的距离均为**1**。
- **KMP**算法，旨在利用**pattern**的内容指导滑动窗口向右滑动的距离。

- **Knuth-Morris-Pratt**算法（简称KMP）是最常用的字符串模式匹配算法之一，1968年。
- 以三个发明者命名，起头的那个K就是著名科学家**Donald Knuth**。




BBC ABCDAB ABCDABCDABDE
ABCDABD

1) 目标串"BBC ABCDAB ABCDABCDABDE"的第一个字符与模式串"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以模式串后移一位。

BBC ABCDAB ABCDABCDABDE
ABCDABD


2) 因为B与A不匹配，模式串再往后移。

BBC ABCDAB ABCDABCDABDE
ABCDABD

A red dashed box highlights the first character 'A' of the target string 'ABCDABD' and the first character 'A' of the pattern string 'ABCDAB' in the text 'BBC ABCDAB ABCDABCDABDE'.

3) 就这样，直到目标串有一个字符，与模式串的第一个字符相同为止。

BBC ABCDAB ABCDABCDABDE
ABCDABD

A red dashed box highlights the first two characters 'AB' of the target string 'ABCDABD' and the first two characters 'AB' of the pattern string 'ABCDAB' in the text 'BBC ABCDAB ABCDABCDABDE'.

4) 接着比较目标串和模式串的下一个字符，还是相同。

BBC ABCDAB ABCDABCDABDE
ABCDABD

5) 直到目标串有一个字符，与模式串对应的字符不相同为止。

BBC ABCDAB ABCDABCDABDE
ABCDABD

6) 这时，最自然的反应是，将模式串整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为要把"搜索位置"移到已经比较过的位置，重比一遍。

BBC ABCDAB ABCDABCDABDE
ABCDABD

7) 当空格与D不匹配时，已知前面六个字符是"ABCDAB"。KMP算法设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，由此提高效率。

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

8) 《部分匹配表》 (Partial Match Table)

BBC ABCDAB ABCDABCDABDE
ABCDABD

9) 已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符B对应的"部分匹配值"为2，按照下面公式算出向后移动的位数：

移动位数 = 已匹配的字符数 - 对应的部分匹配值

因为 $6 - 2$ 等于4，所以将模式串向后移动4位。

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

BBC ABCDAB ABCDABCDABDE
ABCDABD

10) 因为空格与C不匹配，模式串还要继续往后移。
这时，已匹配的字符数为2 ("AB")，对应的"部分匹配值"为0。所以，移动位数 = 2 - 0，结果为2，于是将模式串向后移2位。

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

BBC ABCDAB ABCDABCDABDE
ABCDABD

11) 因为空格与A不匹配，继续后移一位。

BBC ABCDAB ABCDABCDABDE
ABCDABD

12) 逐位比较，直到发现C与D不匹配。于是，移动位数 = $6 - 2$ 。继续将模式串向后移动4位。

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

BBC ABCDAB ABCDABCDABDE

ABCDABD

13) 逐位比较，直到模式串的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = $7 - 0$ ，再将模式串向后移动7位，这里就不再重复。

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

《部分匹配表》 (Partial Match Table)

字符串： "bread"

前缀： b , br , bre , brea

后缀： read , ead , ad , d

- "前缀"指除了最后一个字符以外，一个字符串的全部头部组合；
- "后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

- "部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以"ABCDABD"为例,
 - ✓ "A"的前缀和后缀都为空集, 共有元素的长度为0;
 - ✓ "AB"的前缀为[A], 后缀为[B], 共有元素的长度为0;
 - ✓ "ABC"的前缀为[A, AB], 后缀为[BC, C], 共有元素的长度0;
 - ✓ "ABCD"的前缀为[A, AB, ABC], 后缀为[BCD, CD, D], 共有元素的长度为0;
 - ✓ "ABCDA"的前缀为[A, AB, ABC, ABCD], 后缀为[BCDA, CDA, DA, A], 共有元素为"A", 长度为1;
 - ✓ "ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA], 后缀为[BCDAB, CDAB, DAB, AB, B], 共有元素为"AB", 长度为2;
 - ✓ "ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB], 后缀为[BCDABD, CDABD, DABD, ABD, BD, D], 共有元素的长度为0。

BBC ABCDAB ABCDABCDABDE
ABCDABD

"部分匹配"的实质是，有时候，字符串头部和尾部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是2（"AB"的长度）。模式串移动的时候，第一个"AB"向后移动4位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

简单模式匹配的缺点：无谓比较

目标 *T* *S T U D E N S T U D E N T.....*
 || || || ×

模式 *pat* *S T U D E N T*

目标 *T* *S T U D E N S T U D E N T.....*
 ×

模式 *pat* *S T U D E N T.....*

目标 *T* *S T U D E N S T U D E N T.....*
 || || || ||

模式 *pat* *S T U D E N T*

改进的模式匹配:

寻找最大“跳跃”

目标 T

t_0	t_1	t_2	$\dots\dots$	t_{j-1}	t_j	$\dots\dots$	t_{n-1}
\parallel	\parallel	\parallel		X			

模式 pat $p_0 p_1 p_2 \cdots p_{j-1} p_j \cdots p_{m-1}$

目标 T t_0 t_1 \dots t_k \dots \dots t_{n-1}

\updownarrow \updownarrow \updownarrow \updownarrow

模式 pat $p_0 \ p_1 \ \cdots \cdots \ p_{m-2} \ p_{m-1}$

设目标 $T = "t_0 t_1 \dots t_{n-1}"$ ，模式 $P = "p_0 p_1 \dots p_{m-1}"$ ，

用穷举模式匹配法做第 $s+1$ 趟比较时——

从目标 T 的第 s 位置 t_s 与模式 P 的第 0 位置 p_0 开始进行比较，直到目标 T 第 t_{s+j} “失配”。

$$\begin{array}{cccccccccccc}
 \textcolor{red}{T} & t_0 & \cdots & t_{s-1} & t_s & t_{s+1} & t_{s+2} & \cdots & t_{s+j-2} & t_{s+j-1} & t_{s+j} & \cdots & t_{n-1} \\
 & & & & \parallel & \parallel & \parallel & & \parallel & \parallel & \times & & \\
 \textcolor{red}{P} & & & & \{p_0 & [p_1 & p_2 & \cdots & p_{j-2}] & p_{j-1} &] p_j
 \end{array}$$

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

为使模式 $\textcolor{red}{P}$ 与目标 $\textcolor{red}{T}$ 匹配，必须满足

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

如果 $p_0 p_1 \cdots p_{j-2} \neq p_1 p_2 \cdots p_{j-1} \quad (\textcolor{red}{r=2}) \quad (2)$
 则立刻可以断定

$p_0 p_1 \cdots p_{j-2} \neq t_{s+1} t_{s+2} \cdots t_{s+j-1}$
 下一趟必不匹配。

同样，若 $p_0 p_1 \cdots p_{j-3} \neq p_2 p_3 \cdots p_{j-1}$ ($r=3$)
 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 “ r ” 值，使得

$$p_0 p_1 \cdots p_{j-r} \neq p_r p_{r+1} \cdots p_{j-1}$$

且 $p_0 p_1 \cdots p_{j-(r+1)} = p_{r+1} p_r \cdots p_{j-1}$
 用 k 替换 $j-(r+1)$ ，即 $k=j-(r+1)$ ，则 $j-r=k+1$ ， $r+1=j-k$ 。

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

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

则

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

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

在第 s 趟比较“失配”时的模式 P 从当时位置直接向右“滑动” $j-k-1$ 位。

因为目标 T 中 t_{s+j} 之前已经与模式 P 之前的字符匹配，则可以直接从 T 中的 t_{s+j} （即上一趟失配的位置）与模式中的 p_{k+1} 开始，继续向下进行匹配比较。

改进算法中，目标 T 在第 s 趟比较失配时，扫描指针 s 不必回溯。算法下一趟继续从此处开始向下进行匹配比较；而在模式 P 中，扫描指针应退回到 p_{k+1} 位置。

k 的确定方法

当比较到模式第 j 个字符失配时, k 的值与模式的前 j 个字符有关, 与目标无关。

目标 T $F I F I F I Y U D E N T \dots$

|| || || || ×

模式 pat $F I F I Y$

启发: 要更好地了解外面的世界,
先要了解小小的我。

失效函数

可以利用失效函数 *next(j)* 描述。

失效函数又名失败链接值。

哲理：在当前位置遇到失败后，
重新开始的位置。

当模式 P 中的第 j 个字符与目标 T 中相应字符失配时，模式 P 中应该由哪个字符与目标中刚失配的字符对齐继续进行比较。

若设 模式 $P = p_0 p_1 \cdots p_{m-2} p_{m-1}$

$$next(j) = \begin{cases} -1, & \text{当 } j = 0 \\ k + 1, & \text{当 } 0 \leq k < j - 1 \text{ 且使得 } p_0 p_1 \cdots p_k = p_{j-k-1} p_{j-k} \cdots p_{j-1} \text{ 的最大整数} \\ 0, & \text{其它情况} \end{cases}$$

示例：确定失效函数 $next(j)$

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

利用失效函数 $next(j)$ 的匹配处理

如果 $j = 0$ ，则目标指针加 1，
模式指针回到 p_0 。

如果 $j > 0$ ，则目标指针不变，
模式指针回到 $p_{next(j)}$ 。

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*

× $j = 1 \Rightarrow j = \text{next}(j) = 0$

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*

× $j = 0$ 目标指针加 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*

× $j = 5$

$\Rightarrow j = \text{next}(j) = 2$

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* ✓


```
int AString :: FastFind ( AString pat ) const {  
    //带失效函数的KMP匹配算法  
    int posP = 0, posT = 0;  
    int lengthP = pat.curLength, lengthT = curLength;  
    while ( posP < lengthP && posT < lengthT )  
        if ( pat.ch[posP] == ch[posT] ) {  
            posP++; posT++; //相等继续比较  
        }  
        else if ( posP == 0 ) posT++; //不相等  
            else posP = pat.next [posP];  
    if ( posP < lengthP ) return -1;  
    else return posT - lengthP;  
}
```

```
int AString :: FastFind ( AString &pat,  
                          int k, int next[ ] ) const {  
    //用模式串 pat 从 k 开始寻找当前串中匹配的位置  
    int posP = 0, posT = k;  
    int lengthP = pat.curLength, lengthT = curLength;  
    while ( posP < lengthP && posT < lengthT )  
        if ( posP == -1 || pat.ch[posP] == ch[posT] ) {  
            posP++; posT++; //相等继续比较  
        }  
        else posP = pat.next [posP];  
    if ( posP < lengthP ) return -1;  
    else return posT - lengthP;  
}
```

$next(j)$ 的正确计算方法

在串 $p_0 p_1 p_2 \cdots p_{j-1}$ 中找出最长的相等的前缀子串 $p_0 p_1 \cdots p_k$ 和后缀子串 $p_{j-k-1} p_{j-k} \cdots p_{j-1}$ 。

设 $next(j) = k$, 则有

$$0 \leq k < j-1 \text{ 且 } p_0 p_1 \cdots p_k = p_{j-k-1} p_{j-k} \cdots p_{j-1}$$

若设 $next(j+1) = \max\{k+1 \mid 0 \leq k+1 < j\}$, 使得 $p_0 p_1 \cdots p_{k+1} = p_{j-k-1} p_{j-k} \cdots p_j$ 成立。

若 $p_{k+1} = p_j$, 则 $next(j+1) = k+1 = next(j)+1$ 。

若 $p_{k+1} \neq p_j$, 则在 $p_0 p_1 \cdots p_k$ 中寻找使得

$$p_0 p_1 \cdots p_h = p_{k-h} p_{k-h+1} \cdots p_k$$

的 h 。

(1) 找到 h , $next(k) = h$, 则有

$$p_0 p_1 \cdots p_h = p_{k-h} p_{k-h+1} \cdots p_k = p_{j-h-1} p_{j-h} \cdots p_{j-1}$$

即在 $p_0 p_1 \cdots p_{j-1}$ 中找到了长度为 $h+1$ 的相等的前缀子串和后缀子串。

若 $p_{h+1} = p_j$, 则 $next(j+1) = h+1 = next(k)+1$
 $= next(next(j))+1$ 。

若 $p_{h+1} \neq p_j$, 则在 $p_0 p_1 \cdots p_h$ 中寻找更小的 h' 。
如此递推, 以同样方式再缩小寻找范围, 直到 $next(t) = -1$ 才算是失败。

(2) 找不到 h , $next(k) = -1$ 。

```
void AString :: GetNext ( int next[ ] ) {  
    //计算失效函数  
    int j = 0, k = -1;  
    int lengthP = curLength;  
    next [0] = -1; //直接赋值  
    while ( j < lengthP ) { //依次求 next[j]  
        if ( k == -1 || ch[j] == ch[k] ) {  
            j++; k++;  
            next[j] = k;  
        }  
        else k = next[k];  
    }  
}
```

- 改进的模式匹配算法可使目标串的检测指针每趟不回退。
- 改进的模式匹配(KMP)算法的时间代价
 - ◆在带失效函数的匹配算法中，字符比较次数最多为目标串的长度 n ，则其时间复杂度为 $O(n)$ ；
 - ◆在失效函数计算过程中，循环次数至多不能超过 $m-1$ ，则其时间复杂度为 $O(m)$ ；
 - ◆在包括计算失效函数的整个模式匹配过程中，时间复杂度为 $O(m+n)$ 。



■ KMP算法

```
1. //预处理：计算Next数组
2. //带优化
3. void preprocessing(char *p, int
   m, int Next[ ]) {
4.     for (int i=0, j=Next[0]=-1;
       i<m-1; ) {
5.         for ( ; j>-1 && p[i]!=p[j]; )
6.             j=Next[j];
7.         i++; j++;
8.         if (p[i]==p[j])
9.             Next[i]=Next[j];
10.        else
11.            Next[i]=j;
12.    }
13. }
```

```
1. int KMP(char *p, int m, char *t,
   int n)
2. {
3.     int i, j, Next[SIZE];
4.     //预处理
5.     preprocessing(p, m, Next);
6.     //查找
7.     for (i=j=0; j<n; ) {
8.         for ( ; i>-1 && p[i]!=t[j]; )
9.             i=Next[i];
10.        i++; j++;
11.        if (i==m)
12.            return j-i;
13.    }
14.    return -1;
15. }
```

■ KMP算法

1. //预处理：计算Next数组
2. //带优化
3. **void preprocessing(char *p, int m, int Next[])** {
4. **for (int i=0, j=Next[0]=-1;**
 i<m-1;) {
5. **for (; j>-1&& p[i]!=p[j];)**
6. **j=Next[j];**
7. **i++; j++;**
8. **if (p[i]==p[j])**
9. **Next[i]=Next[j];**
10. **else**
11. **Next[i]=j;**
12. **}**
13. }

j	0	1	2	3	4	5
P _j	a	b	a	c	a	b
N _j	-1	0	0	1	0	1

j	0	1	2	3	4	5
P _j	a	b	a	c	a	b
N _j	-1	0	-1	1	-1	0

■ KMP算法

1. //预处理：计算Next数组
2. //带优化
3. **void preprocessing(char *p, int m, int Next[])** {
4. **for (int j=0, k=Next[0]=-1;**
 j<m-1;) {
5. **for (; k>-1&& p[j]!=p[k];)**
6. **k=Next[k];**
7. **j++; k++;**
8. **if (p[j]==p[k])**
9. **Next[j]=Next[k];**
10. **else**
11. **Next[j]=k;**
12. **}**
13. **}**

j	0	1	2	3	4	5
P _j	a	b	a	c	a	b
N _j	-1	0	0	1	0	1

j	0	1	2	3	4	5
P _j	a	b	a	c	a	b
N _j	-1	0	-1	1	-1	0

■ KMP算法

1. //预处理：计算Next数组

2. //未带优化

3. void preprocessing(char *p, int m, int
Next[]) {

4. for (int j=0, k=Next[0]=-1; j<m-1;)

5. {

6. for (; k>-1 && p[j]!=p[k];)

7. k=Next[k];

8. j++; k++;

9. Next[j]=k;

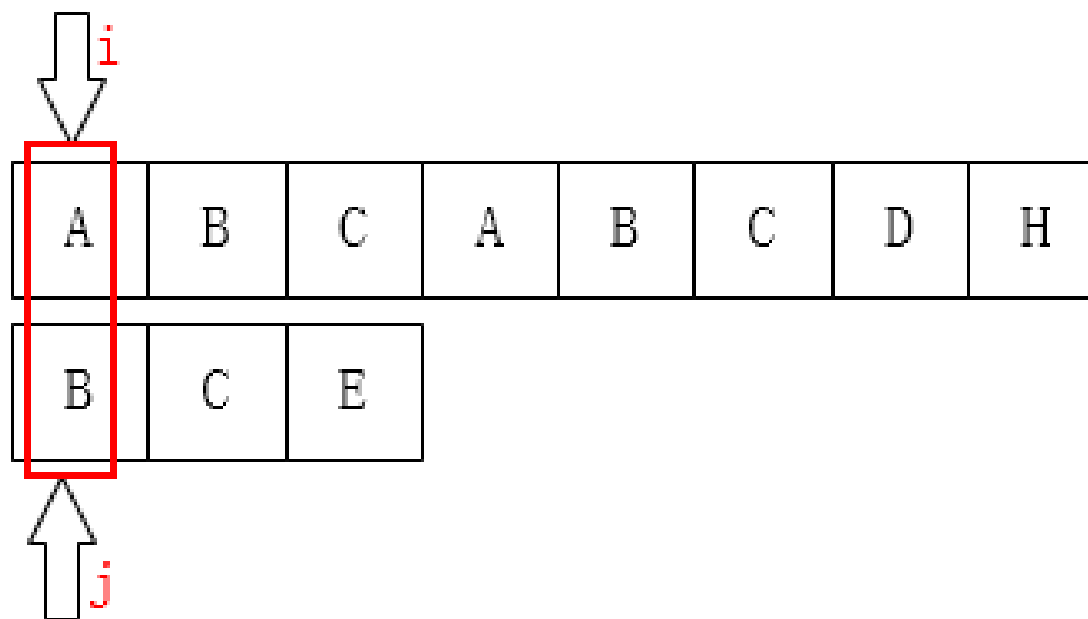
10. }

11. }

j	0	1	2	3	4	5
P _j	a	b	a	c	a	b
N _j	-1	0	0	1	0	1

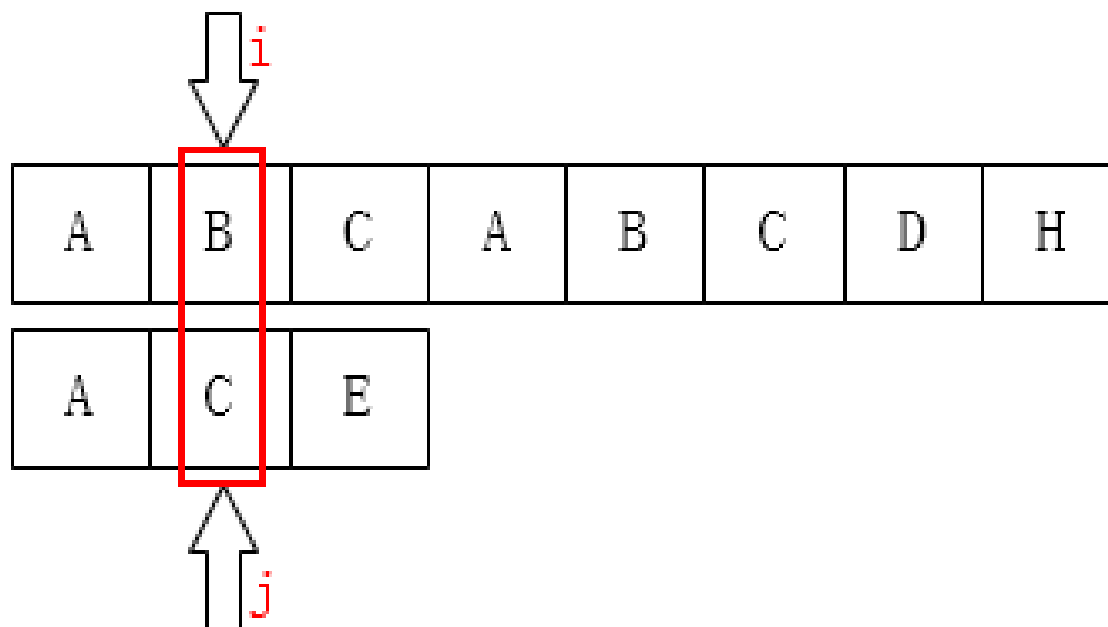
//next[j]的值（也就是k）表示，当 $P[j] \neq T[i]$ 时，j指针的下一步移动位置。

1) 当j为0时，如果这时候不匹配，怎么办？

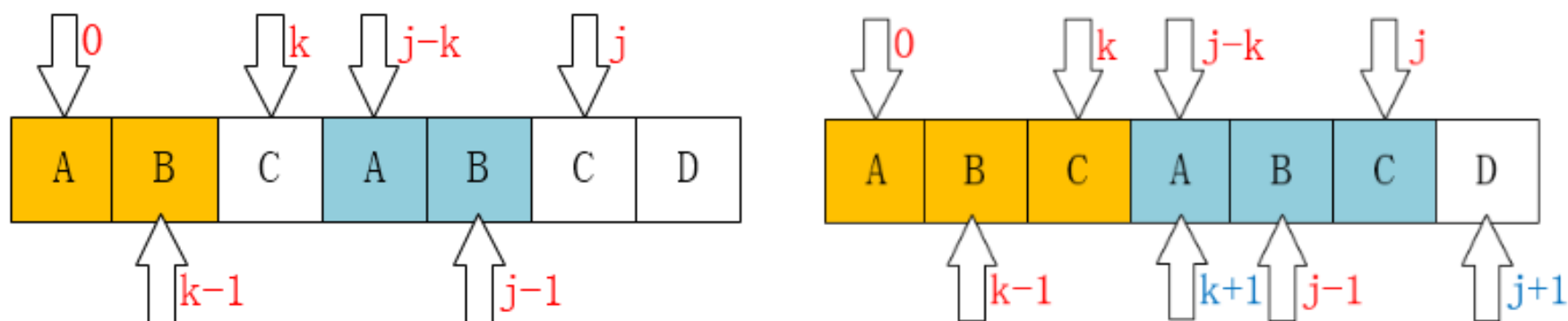


j已经在最左边，不可能再移动，这时候要应该是i指针后移。所以在代码中才会有 $\text{next}[0] = -1$;这个初始化。

2) 如果是当j为1的时候呢？



显然，j指针一定是移动到0位置的，因为它前面也就只有这一个位置。



3) 发现一个规律：

当 $P[j] == P[k]$ 时，有 $next[j+1] == next[j] + 1$

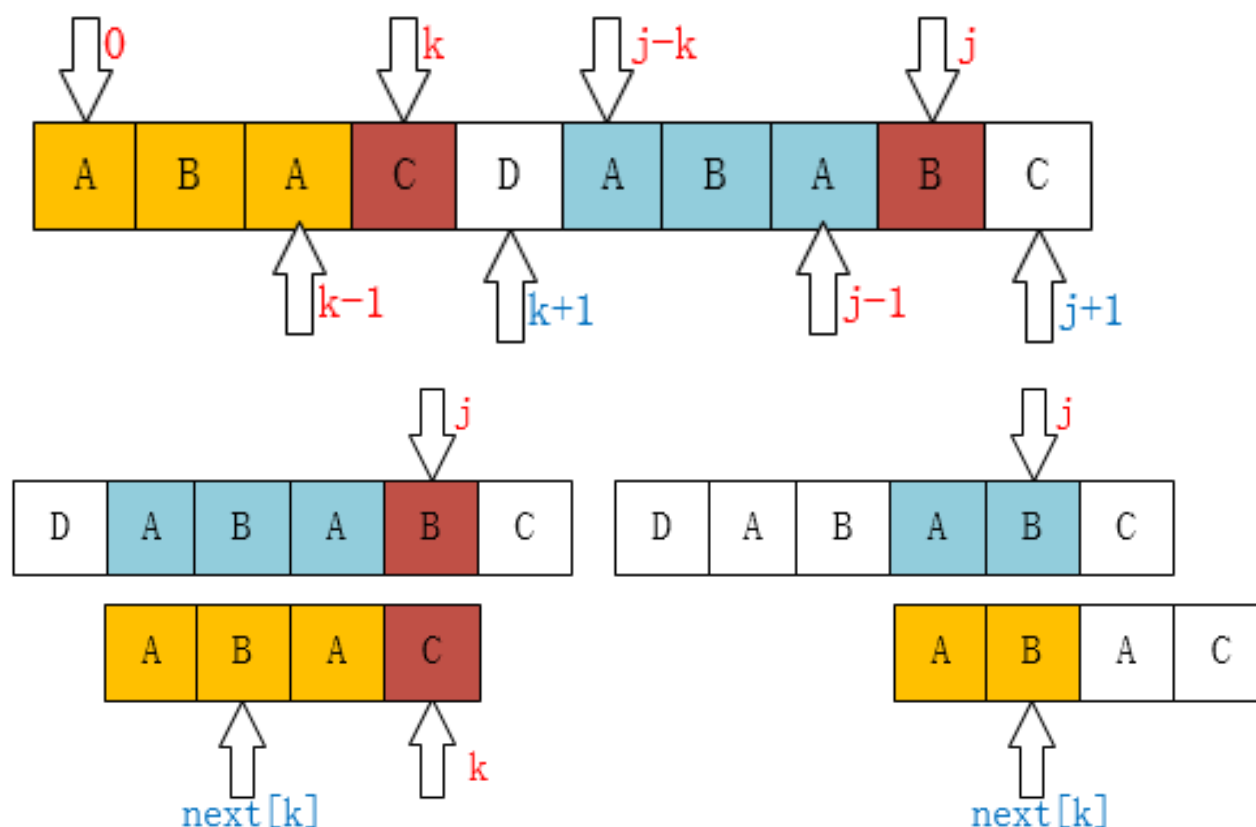
从代码上看应该是 $j++$; $k++$; $Next[j] = k$;

这个是可以证明的：

因为在 $P[j]$ 之前已经有 $P[0 \sim k-1] == p[j-k \sim j-1]$ ，即 $next[j] == k$ 。

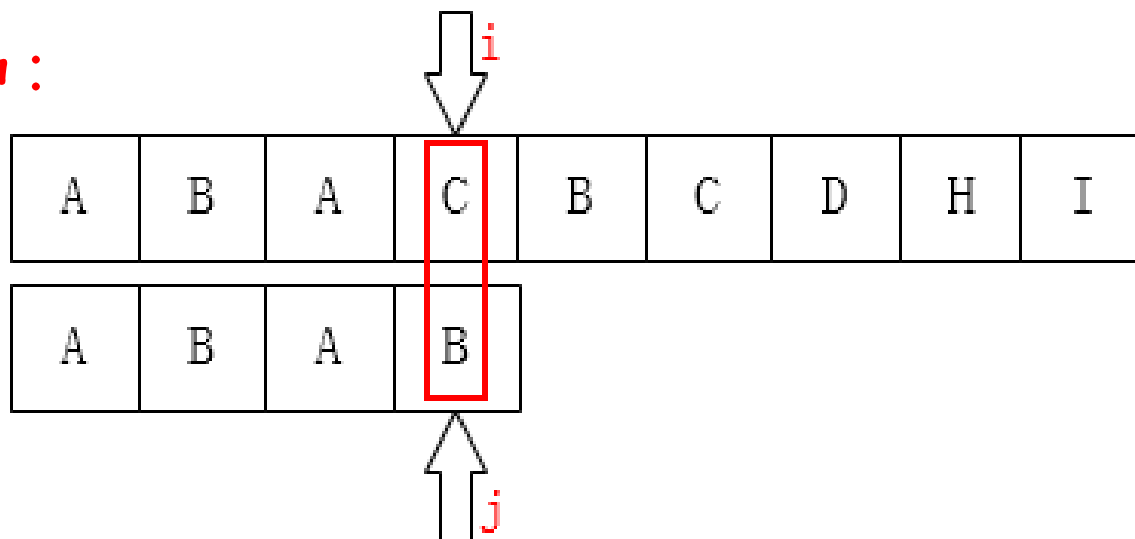
这时候有 $P[k] == P[j]$ ，可得到 $P[0 \sim k-1] + P[k] == p[j-k \sim j-1] + P[j]$ ，则 $P[0 \sim k] == P[j-k \sim j]$ ，即 $next[j+1] == k+1 == next[j] + 1$ 。

4) 如果 $P[j] \neq P[k]$ ，从代码上看应该是 $k = \text{next}[k]$ 。

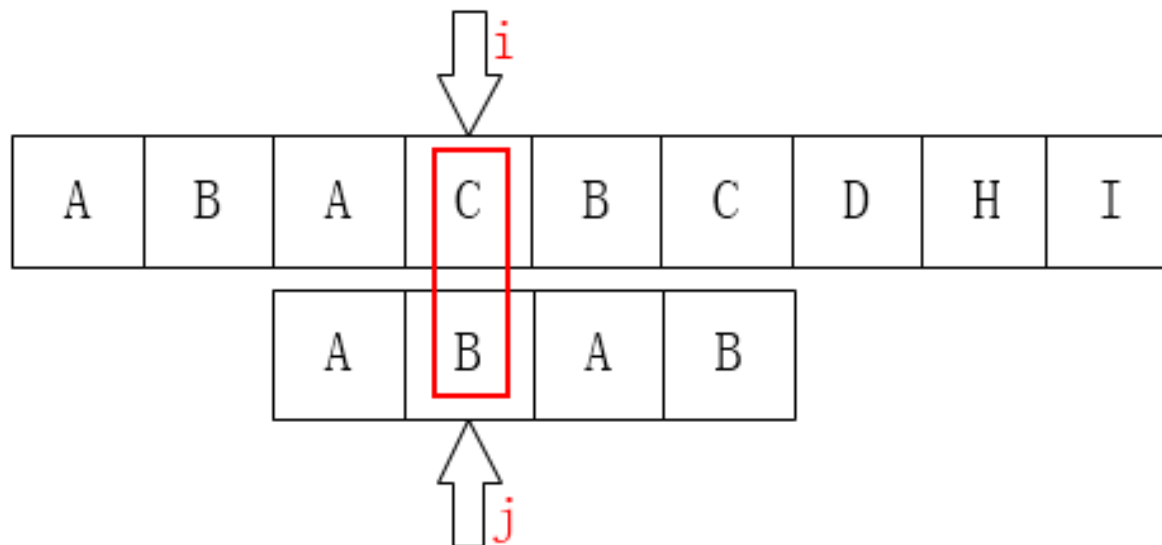


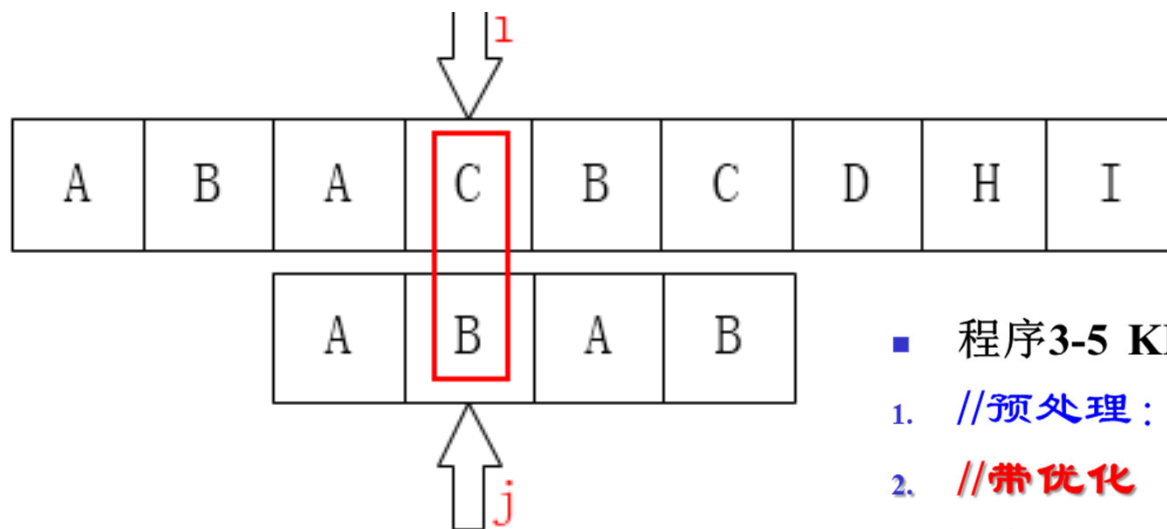
已经不可能找到ABAB这个最长的后缀串，但还是可能找到AB、B这样的前缀串。则这个过程像在定位ABAC这个串，当C和主串不一样（也就是k位置不一样），那当然是把指针移动到 $\text{next}[k]$ 。

5) 缺陷:



当前续算法得到的next数组应该是 $[-1, 0, 0, 1]$, 则下一步应该把j移动到第1个元素。





这一步完全没有意义，
因为后面的B已经不匹配，
那前面的B也一定是不匹配，
同样的情况其实还发生在第2个
元素A上。

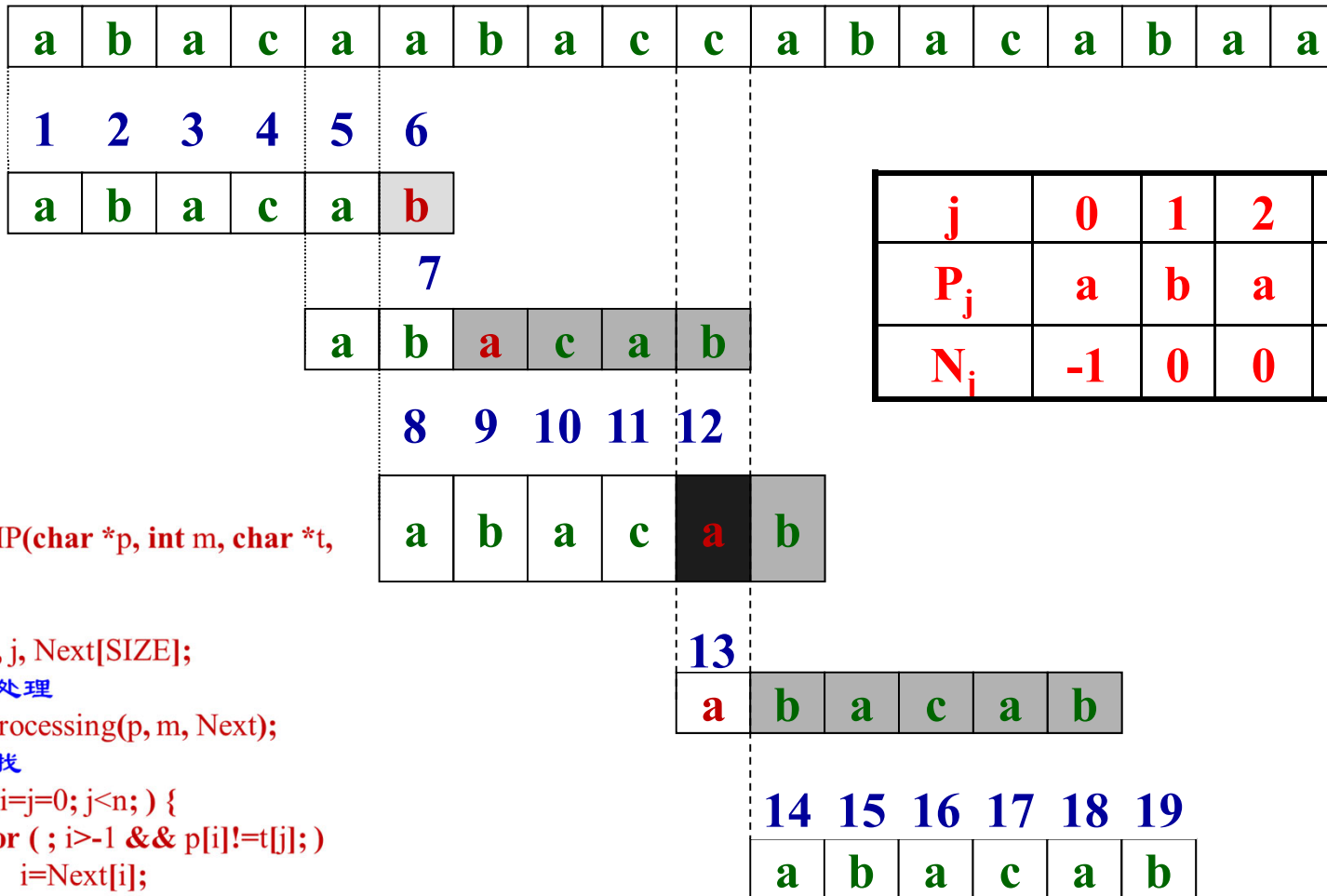
显然，发生问题的原因
在于 $P[j] == P[next[j]]$ 。

■ 程序3-5 KMP算法

```

1. //预处理：计算Next数组
2. //带优化
3. void preprocessing(char *p, int
   m, int Next[ ]) {
4.     for (int j=0, k=Next[0]=-1;
       j<m-1;) {
5.         for (; k>-1 && p[j]!=p[k];)
6.             k=Next[k];
7.         j++; k++;
8.         if (p[j]==p[k])
9.             Next[j]=Next[k];
10.        else
11.            Next[j]=k;
12.    }
13. }
```


KMP模式匹配：示例



```

1. int KMP(char *p, int m, char *t,
2.   int n)
3. {
4.   int i, j, Next[SIZE];
5.   //预处理
6.   preprocessing(p, m, Next);
7.   //查找
8.   for (i=j=0; j<n; ) {
9.     for ( ; i>=0 && p[i]!=t[j]; )
10.       i=Next[i];
11.     i++; j++;
12.     if (i==m)
13.       return j-i;
14.   }
15.   return -1;

```

KMP模式匹配： 示例

a	b	a	c	a	a	b	a	c	c	a	b	a	c	a	b	a	a
1	2	3	4	5	6												
a	b	a	c	a	b												
						7	8	9	10	11							
						a	b	a	c	a	b						

j	0	1	2	3	4	5
P _j	a	b	a	c	a	b
N _j	-1	0	-1	1	-1	0

```

1. int KMP(char *p, int m, char *t,
   int n)
2. {
3.     int i, j, Next[SIZE];
4.     //预处理
5.     preprocessing(p, m, Next);
6.     //查找
7.     for (i=j=0; j<n; ) {
8.         for ( ; i>=0 && p[i]!=t[j]; )
9.             i=Next[i];
10.        i++; j++;
11.        if (i==m)
12.            return j-i;
13.    }
14.    return -1;
15. }
```

12 13 14 15 16 17

a	b	a	c	a	b
---	---	---	---	---	---

预处理过程

i	0	1	2	3	4	5	6	7	8
$x[i]$	C	G	T	C	T	C	T	C	
$Next[i]$	-1	0	0	-1	1	-1	1	-1	1

查找过程

第一轮:

y

C	G	T	A	G	C	G	T	C	T	C	T	C	A	T	A	T	G	T	C	A	T	G	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1 2 3 4

x

C	G	T	C	T	C	T	C
---	---	---	---	---	---	---	---

窗口右移 4 ($i - Next[i] = 3 - (-1)$)

第二轮:

y

C	G	T	A	G	C	G	T	C	T	C	T	C	A	T	A	T	G	T	C	A	T	G	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1

x

C	G	T	C	T	C	T	C
---	---	---	---	---	---	---	---

窗口右移 1 ($i - Next[i] = 0 - (-1)$)

第三轮:

y

C	G	T	A	G	C	G	T	C	T	C	T	C	A	T	A	T	G	T	C	A	T	G	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1 2 3 4 5 6 7 8

x

C	G	T	C	T	C	T	C
---	---	---	---	---	---	---	---

找到子串, 程序结束

图3-4 KMP算法示例

KMP算法的特点

- (1) 与BF算法类似，按照从左到右的顺序进行比较；
- (2) 预处理需要 $O(m)$ 的时间和存储空间；
- (3) 算法时间复杂度为 $O(m+n)$ ；
- (4) 查找阶段的最大比较次数为 $2n-1$ 。

随堂练习

例 1： 设串 s 为 "aaab"，串 t 为 "abcabaa"，串 r 为 "abcaabbabcabaacbacba"，试分别计算它们的失效函数 $f(j)$ 的值。

例 2： 给定一模式串 $p = \text{"abcabaa"}$ 与目标串 $t = \text{"abcaabbabcabaacbacba"}$ ，求解模式串的失效函数值，并给出利用 **KMP** 算法进行模式匹配的每一趟匹配过程的实现。

KR(Karp-Rabin)算法（略）

- 作为最朴素的字符串匹配算法，**BF**算法的效率并不理想。其主要原因有二：
 - 其一是子串与滑动窗口内的子串逐个字符匹配所引发的效率问题；
 - 其二是该算法太健忘，前一次匹配的信息其实可以有部分应用到后一次匹配中的，而**BF**算法只是简单的把这个信息扔掉，重头再来。
- **KR**算法优化"滑动窗口内容逐一匹配"
- **KR**算法：将滑动窗口内 m 个字符的比较变为一个哈希值的比较。
 - 通过对字符串进行哈希运算，然后比较子串哈希值与滑动窗口内子串的哈希值；仅当这两个哈希值相等时，再来比较窗口内的子串是否相等。

■ 程序3-3 Karp-Rabin算法框架

```
1.  function RabinKarp(string s[0..n-1], string sub[0..m-1])
2.      hsub=hash(sub[0..m-1]); //计算子串的哈希值
3.      hs=hash(s[0..m-1]); //计算窗口内子串的哈希值
4.      for i from 0 to n-m
5.          //依次比较窗口内子串与给定子串是否相同
6.              if hs=hsub
7.                  if s[i..i+m-1]=sub
8.                      return i; //如果相同则记录位置并退出函数
9.                  hs=hash(s[i+1..i+m]); //否则，当前窗口右移
10.     return not found;
11.     //没有发现，记录没有发现给定子串，退出
```

哈希算法

- **KR**算法的效率取决于哈希函数的选取。通常，**KR**算法使用 $h(x)=x \bmod q$ 作为哈希函数。

- 滑动窗口的哈希值计算公式：

$$\text{hash}(w[0..m-1]) = (w[0] \times 2^{m-1} + w[1] \times 2^{m-2} + \dots + w[m-1] \times 2^0) \bmod q$$

其中， q 为一个大整数。

- 当滑动窗口右移时，需要对当前窗口内的子串重新利用哈希函数计算，其计算公式为：

$$\text{rehash}(a, b, h) = ((h - a \times 2^{m-1}) \times 2 + b) \bmod q$$

其中， h 为上一滑动窗口的哈希值； a 为即将移出滑动窗口的字符； b 是即将移入滑动窗口的字符。

■ 程序3-4 Karp-Rabin算法（窗口长度为 m ， q 选取 2^{m-1} ）

```

1.  #define Rehash(a, b, h) (((h)-(a)*d)<<1)+(b))
2.  int KR(char *x, int m, char *y, int n) {
3.      int d, hx, hy, i, j;
4.      //预处理，计算 $d=2^{(m-1)}$ ，这里的d为算法描述中的q
5.      for (d=i=1; i<m; i++)
6.          d=d<<1;
7.      //分别计算字符串x和y的哈希值
8.      for (hx=hx=i=0; i<m; i++) {
9.          hx=(hx<<1)+x[i];
10.         hy=(hy<<1)+y[i];
11.     }
12.     //查找过程
13.     for (j=0; j<=n-m; j++) {
14.         //如果哈希值相等，再判断字符串是否相等
15.         if (hx==hy && memcmp(x, y+j, m)==0)
16.             return j; //如字符串相等，返回位置
17.         hy=Rehash(y[j], y[j+m], hy);
18.         //否则，对y上的子串重新进行哈希
19.     }
20.     return -1;
21. }

```

第一轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C

x C G T C T C T C

$\text{Hash}(y[0..7])=17910$

第二轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C

x C G T C T C T C

$\text{Hash}(y[1..8])=18735$

第三轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C

x C G T C T C T C

$\text{Hash}(y[2..9])=19378$

...

第六轮比较:

y C G T A G C G T C T C T C A T A T G T C A T G C

1 2 3 4 5 6 7 8

x C G T C T C T C

$\text{Hash}(y[5..12])=\text{Hash}(x)=18055$

找到子串, 函数结束

图3-3 Karp-Rabin算法示例

KR算法的特点


- (1) 利用哈希的方法;
- (2) 预处理需要 $O(m)$ 的时间和常数的存储空间;
- (3) 最坏情况下, 算法时间复杂度为 $O(m \times n)$;
- (4) 算法时间复杂性的预期为 $O(m+n)$ 。

BM(Boyer-Moore)算法（略）

- **KMP**算法并不是效率最高的算法，实际采用并不多。
- 各种文本编辑器的"查找"功能(Ctrl+F)，大多采用**Boyer-Moore**算法。
- **Boyer-Moore**算法不仅效率高，而且构思巧妙，容易理解。
- 1977年，德克萨斯大学的**Robert S. Boyer**教授和**J Strother Moore**教授发明这种算法。

- **BM**算法与**KMP**算法的差别在于：
 - (1) 在进行匹配比较时，不是自左向右进行，而是自右向左进行；
 - (2) 预先计算出正文中可能的字符在模式中出现的位置相关信息，利用这些信息来减少比较次数。
- 根据字符串匹配的思想，窗口向右移动的动作一定发生在匹配失效或完全匹配时。为了右移更大的距离，**BM**算法使用两个预计算函数来指导窗口向右移动的距离，这两个函数分别被称为好后缀移动 (*Good-Suffix Shift*) 和 坏字符移动 (*Bad-Character Shift*)。

HERE IS A SIMPLE EXAMPLE
EXAMPLE



- 1) "目标串"与"模式串"头部对齐，从尾部开始比较。
这是一个很聪明的想法，因为如果尾部字符不匹配，那么只要一次比较，就可以知道前7个字符（整体上）肯定不是要找的结果。
- 2) "S"与"E"不匹配。这时，"S"就被称为"坏字符" (bad character)，即不匹配的字符。
- 3) 还发现，"S"不包含在模式串"EXAMPLE"之中，这意味着可以把模式串直接移到"S"的后一位。

HERE IS A SIMPLE EXAMPLE
EXAMPLE



4) 依然从尾部开始比较，发现"P"与"E"不匹配，所以"P"是"坏字符"。但是，"P"包含在模式串"EXAMPLE"之中。所以，将模式串后移两位，两个"P"对齐。

HERE IS A SIMPLE EXAMPLE
EXAMPLE




5) "坏字符规则"

后移位数=坏字符的位置-模式串中的上一次出现位置

如果"坏字符"不包含在模式串之中，则上一次出现位置为 -1。

以"P"为例，它作为"坏字符"，出现在模式串的第6位（从0开始编号），在模式串中的上一次出现位置为4，所以后移 $6 - 4 = 2$ 位。再以前面第二步的"S"为例，它出现在第6位，上一次出现位置是-1（即未出现），则整个模式串后移 $6 - (-1) = 7$ 位。

HERE IS A SIMPLE EXAMPLE
EXAMPLE



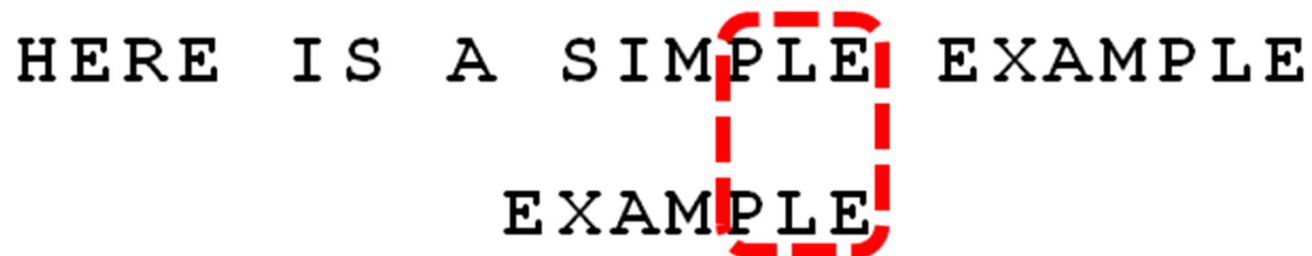
6) 依然从尾部开始比较, "E"与"E"匹配。

HERE IS A SIMPLE EXAMPLE
EXAMPLE



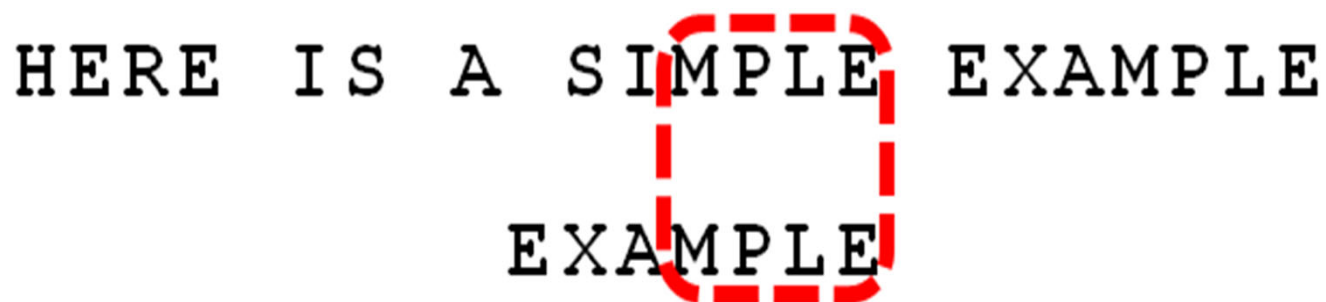
7) 比较前面一位, "LE"与"LE"匹配。

HERE IS A SIMPLE EXAMPLE
EXAMPLE

A diagram illustrating a string matching step. The text 'HERE IS A SIMPLE EXAMPLE' is on the top line, and 'EXAMPLE' is on the bottom line, shifted to the right. A red dashed rectangular box is drawn around the 'SIMPLE' part of the top line and the 'XAMPLE' part of the bottom line, specifically highlighting the 'PLE' suffix of both words.

8) 比较前面一位, "PLE"与"PLE"匹配。

HERE IS A SIMPLE EXAMPLE
EXAMPLE

A diagram illustrating a string matching step. The text 'HERE IS A SIMPLE EXAMPLE' is on the top line, and 'EXAMPLE' is on the bottom line, shifted to the right. A red dashed rectangular box is drawn around the 'SIMPLE' part of the top line and the 'XAMPLE' part of the bottom line, specifically highlighting the 'MPLE' suffix of both words.

9) 比较前面一位, "MPLE"与"MPLE"匹配, 把这种情况称为"好后缀" (good suffix), 即所有尾部匹配的字符串。注意, "MPLE"、"PLE"、"LE"、"E"都是好后缀。

HERE IS A SIMPLE EXAMPLE

EXAMPLE

10) 比较前一位，发现"I"与"A"不匹配。所以，"I"是"坏字符"。

HERE IS A SIMPLE EXAMPLE

EXAMPLE

11) 根据"坏字符规则"，此时模式串应该后移 $2 - (-1) = 3$ 位。问题是，此时有没有更好的移法？

HERE IS A SIMPLE EXAMPLE
EXAMPLE



12) 此时存在"好后缀", 可采用"好后缀规则":

后移位数=好后缀的位置-模式串中的上一次出现位置

举例来说, 如果目标串"ABCDAB"的后一个"AB"是"好后缀"。那么它的位置是5 (从0开始计算, 取最后的"B"的值), 在"模式串中的上一次出现位置"是1 (第一个"B"的位置), 所以后移 $5 - 1 = 4$ 位, 前一个"AB"移到后一个"AB"的位置。

再举一个例子，如果目标串"ABCDEF"的"EF"是好后缀，则"EF"的位置是5，上一次出现的位置是 -1（即未出现），所以后移 $5 - (-1) = 6$ 位，即整个目标串移到"F"的后一位。

这个规则有三个注意点：

- (1) "好后缀"的位置以最后一个字符为准。假定"ABCDEF"的"EF"是好后缀，则它的位置以"F"为准，即5（从0开始计算）。
- (2) 如果"好后缀"在模式串中只出现一次，则它的上一次出现位置为-1。比如，"EF"在"ABCDEF"之中只出现一次，则它的上一次出现位置为-1（即未出现）。

这个规则有三个注意点：

(3) 如果"好后缀"有多个，则除了最长的那个"好后缀"，其他"好后缀"的上一次出现位置必须在头部。

比如，假定"BABCDAB"的“好后缀”是"DAB"、"AB"、"B"，请问这时"好后缀"的上一次出现位置是什么？回答是，此时采用的好后缀是"B"，它的上一次出现位置是头部，即第0位。这个规则也可以这样表达：如果最长的那个"好后缀"只出现一次，则可将模式串改写成如下形式进行位置计算"(DA)BABCDAB"，即虚拟加入最前面的"DA"。

HERE IS A SIMPLE EXAMPLE

EXAMPLE

9') 所有"好后缀" (MPLE、PLE、LE、E)中, 只有"E"在"EXAMPLE"还出现在头部, 所以后移 $6 - 0 = 6$ 位。

HERE IS A SIMPLE EXAMPLE

EXAMPLE

10') "坏字符规则"只能移3位, "好后缀规则"可以移6位。所以, Boyer-Moore算法的基本思想是, 每次后移这两个规则之中的较大值。

两个规则的移动位数, 只与模式串有关, 与目标串无关。因此, 可预先计算生成《坏字符规则表》和《好后缀规则表》。

HERE IS A SIMPLE EXAMPLE

EXAMPLE

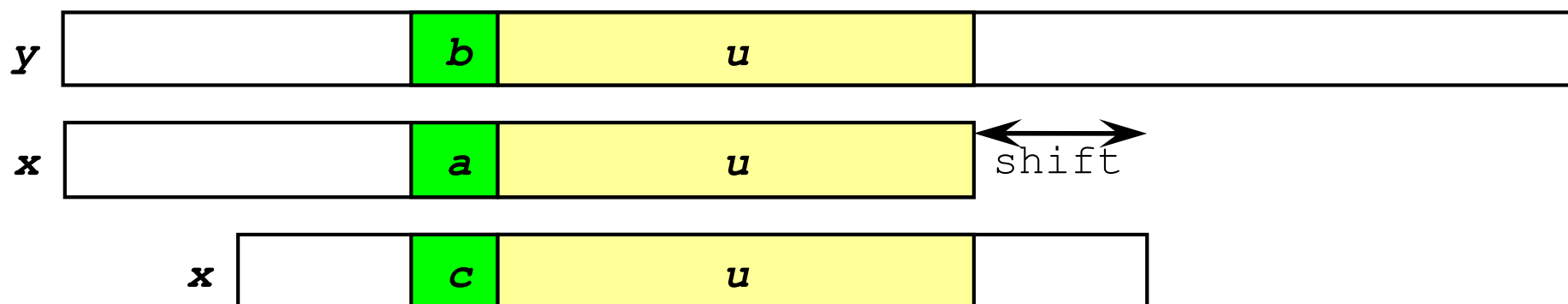
11') 继续从尾部开始比较, "P"与"E"不匹配, 因此
"P"是"坏字符"。根据"坏字符规则", 后移 $6 - 4 = 2$ 位。

HERE IS A SIMPLE EXAMPLE

EXAMPLE

12') 从尾部开始逐位比较, 发现全部匹配, 于是搜索结束。如果还要继续查找 (即找出全部匹配), 则根据"好后缀规则", 后移 $6 - 0 = 6$ 位, 即头部的"E"移到尾部的"E"的位置。

- 好后缀移动 假定当匹配失效发生时，窗口的左侧位于字符串的第 j 个位置，且失效发生在第 i 个位置。由于BM算法按照从右向左的顺序对窗口中的字符串进行比对，这时 $x[i+1..m-1]=y[j+i+1..j+m-1]=u$ ，且 $a=x[i]\neq y[j+i]=b$ 。好后缀移动的基本思想是将窗口右移，使 x 中片段 u 除 $x[i+1..m-1]$ 之外的最右出现移动至当前的比对位置（如图3-5所示）。



- 图3-5 好后缀移动（后缀重复出现）

BM算法

```
1.  const int ASIZE=200;
2.  const int XSIZE=100;
3.  "
4.  //预计算bmBc数组（存放坏字符信息）
5.  //BmBc数组的下标是字符，而不是数字
6.  void preBmBc(char *x, int m, int bmBc[ ])
7.  {
8.      int i;
9.      for (i=0; i<ASIZE; ++i)
10.         bmBc[i]=m;
11.     for (i=0; i<m-1; ++i)
12.         bmBc[x[i]]=m-i-1;
13. }
```

```

1. //在计算BmGc数组时，为提高效率，先计算辅助数组Suff[ ]
2. void suffixes(char *x, int m, int *suff)
3. {
4.     int f, g, i; //f为匹配区间的右端标号
5.     //g为匹配区间的左端标号-1 (g位置的字符不匹配)
6.     suff[m-1]=m; //模式串中的最末位置的匹配的区间为整个字符串
7.     g=m-1; //将匹配区间的左端标号赋值为m-1
8.     for (i=m-2; i>=0; --i) { //循环计算suff数组
9.         //若i>g，则i落在了当前覆盖最远的匹配区间当中
10.        //可以利用已经计算好的suff[i]值
11.        if (i>g && suff[i+m-1-f]<i-g)
12.            suff[i]=suff[i+m-1-f];
13.        //若i<g，则i落在了当前覆盖最远的匹配区间之外
14.        //需要从第i个字符向前检验，直到不能匹配的字符q为止。
15.        //则suff[i]=f-g.
16.        //该过程相当于右端标号f保持不变，
17.        //每计算一次suff，则更新一次f。
18.        else {
19.            if (i<g)
20.                g=i;
21.            f=i;
22.            while (g>=0 && x[g]==x[g+m-1-f])
23.                --g; //直到匹配结束时为止
24.            suff[i]=f-g;
25.        }
26.    }
27. }

```

```
1. //预计算bmGs数组 (存放好后缀信息)
2. //BmGs数组的下标是数字, 表示字符在模式串中的位置
3. void preBmGs(char *x, int m, int bmGs[ ])
4. {
5.     int i, j=0, suff[XSIZE];
6.     suffixes(x, m, suff);
7.     for (i=0; i<m; ++i)
8.         bmGs[i]=m;
9.     for (i=m-1; i>=-1; --i)
10.        if (i== -1 || suff[i]==i+1)
11.            for ( ; j<m-1-i; ++j)
12.                if (bmGs[j]==m)
13.                    bmGs[j]=m-1-i;
14.     for (i=0; i<=m-1; ++i)
15.         bmGs[m-1-suff[i]]=m-1-i;
16. }
```

```
1. int BM(char *x, int m, char *y, int n)
2. {
3.     int i, j, bmGs[XSIZE], bmBc[ASIZE];
4.     //预处理
5.     preBmGs(x, m, bmGs);
6.     preBmBc(x, m, bmBc);
7.     //查找
8.     j=0;
9.     while (j<=n-m) { //计算字符串是否匹配到了尽头
10.         for (i=m-1; i>=0&& x[i]==y[i+j]; --i);
11.         if (i<0)
12.             return j; //找到匹配, 函数结束
13.         else
14.             j+=MAX(bmGs[i], bmBc[y[i+j]]-m+1+i);
15.         //右移窗口
16.     }
17. }
```

预处理

c	A	C	G	T
$bmBCs[i]$	8	2	6	1

i	0	1	2	3	4	5	6	7
$x[i]$	C	G	T	C	T	C	T	C
$suff[i]$	1	0	0	2	0	4	0	8
$bmGs[i]$	7	7	7	2	7	4	7	1

查找

第一轮:

y
C G T A G C G T
C T C T C A T A T G T C A T G C

1

x
C G T C T C T C

右移 1 ($bmGs[7] = bmBc[A] - 7 + 7$)

第二轮:

y
C G T A G C G T C T C T C A T A T G T C A T G C

3 2 1

x
C G T C T C T C

右移 4 ($bmGs[7] = bmBc[A] - 7 + 7$)

第三轮:

y
C G T A G C G T C T C T C A T A T G T C A T G C

8 7 6 5 4 3 2 1

x
C G T C T C T C

找到子串, 程序结束

BM算法示例

BM算法的特点

- (1) 按照从右到左的顺序进行比较;
- (2) 预处理需要 $O(m+\delta)$ 的时间和存储空间;
- (3) 匹配时间的复杂性为 $O(m \times n)$;
- (4) 最好情况下, 时间复杂性为 $O(n/m)$;
- (5) 对非周期的pattern而言, 最坏情况下需要 $3n$ 次字符比较。



4.5 广义表 (General Lists)

- 广义表的概念 $n (\geq 0)$ 个表元素组成的有限序列，记作：

$$LS(a_0, a_1, a_2, \dots, a_{n-1})$$

- LS 是表名， a_i 是表元素，它可以是表（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*)。

广义表的一些例子

$A = ()$

空表，长度为0，无表头和表尾。

$B = (6, 2)$

线性表（只包括原子），长度为2，表头为6，表尾为(2)。

$C = ('a', (5, 3, 'x'))$

长度为2，表头为'a'，表尾为((5, 3, 'x'))（子表）。

$D = (B, C, A)$

长度为3，表头为B，表尾为(C, A)。

$E = (B, D)$

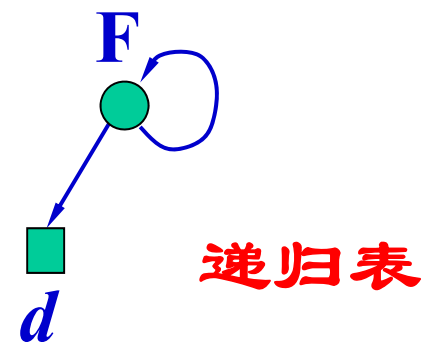
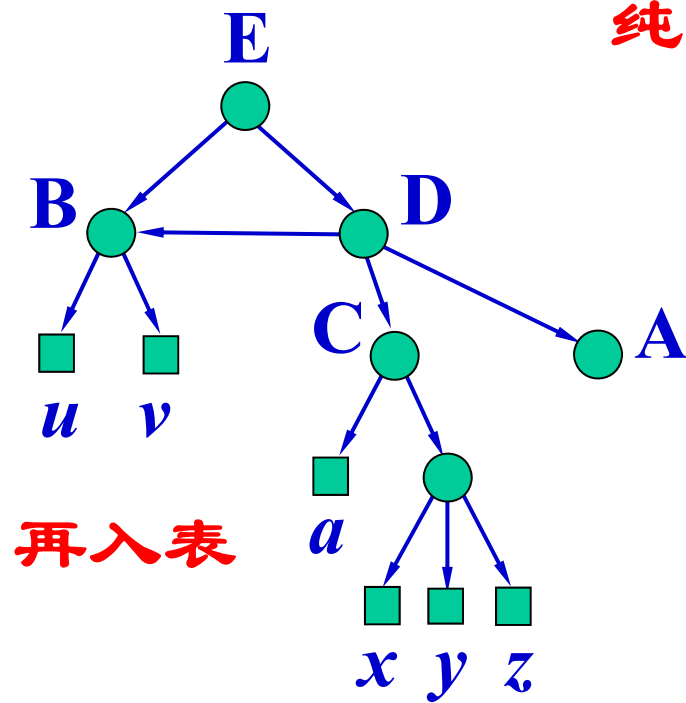
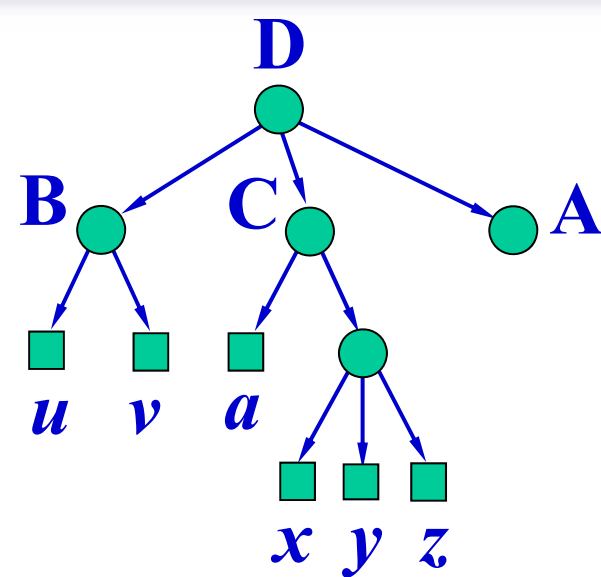
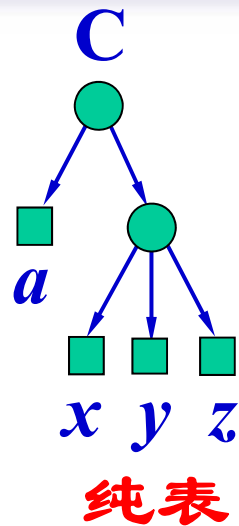
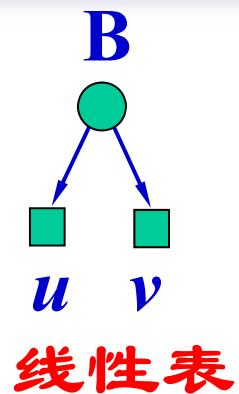
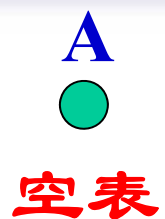
长度为2，表头为B，表尾为(D)。

$F = (4, F)$

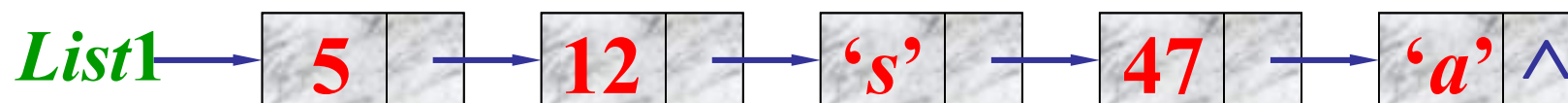
长度为2，表头为4，表尾为(F)。

广义表的特性

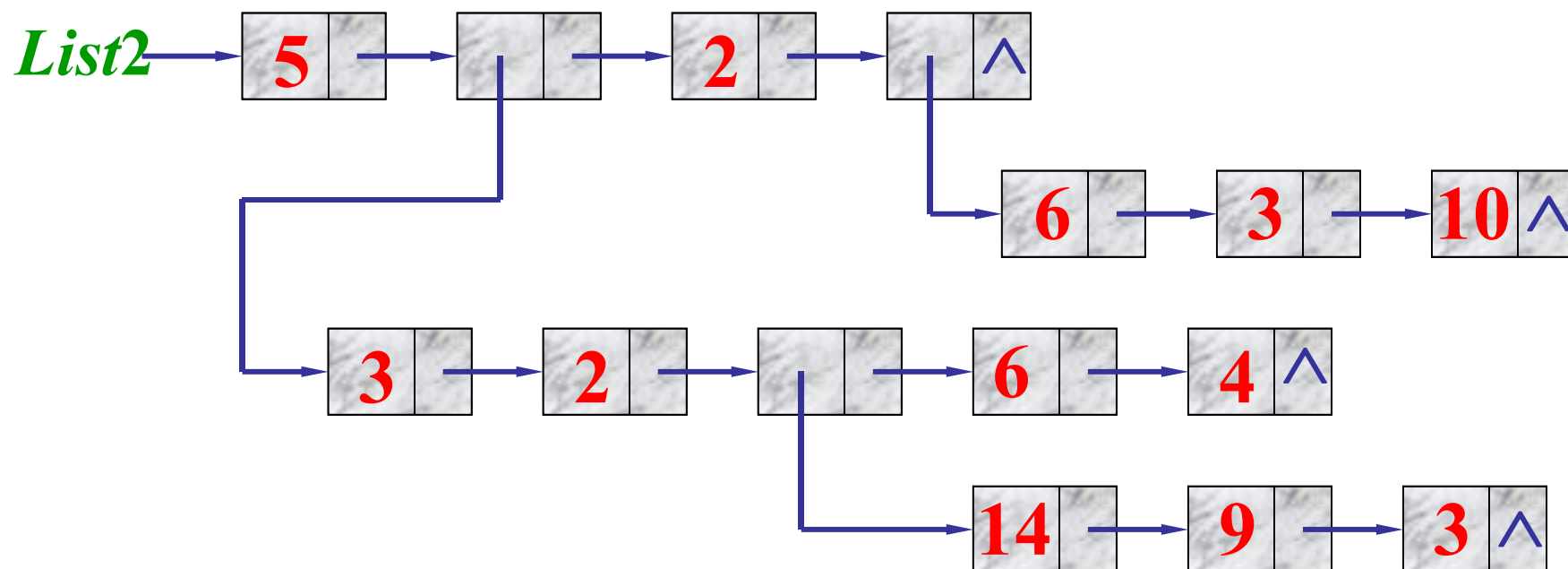
- **有次序性**
 - 在广义表中，各表元素在表中以线性序列排列，每个元素至多一个直接前驱，一个直接后继，且次序不能交换。
- **有长度**
 - 广义表中元素个数一定，不能是无限的，可以是空表。
- **有深度**
 - 广义表是多层次结构，表元素可以是原表的子表，子表的表元素还可以是子表等等。
- **可递归**
 - 广义表可以是自己的子表（如***F***）。
- **可共享**
 - 广义表可以为其它广义表共享—共享表（如***B***）。



广义表的表示



只包括整数和字符型数据的广义表链表表示



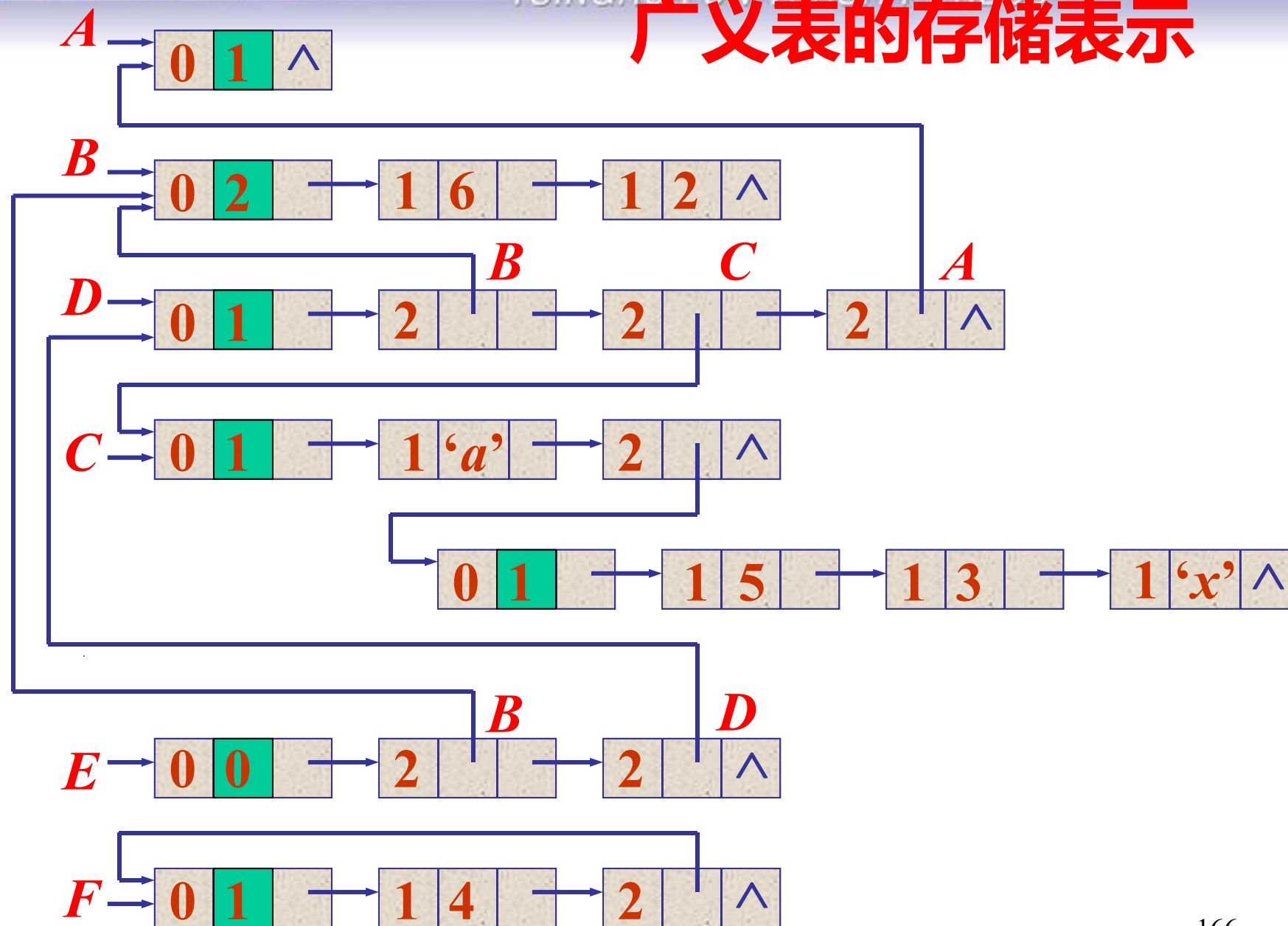
表中套表情形下的广义表链表表示

广义表结点定义

utype	info	tlink
-------	------	-------

- **标志域utype**——**结点类型**, utype = 0, 表头; = 1, 原子; = 2, 子表。
- **信息域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)。
- **尾指针域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



- **存储表示的特点**

- 广义表中的所有表（包括空表），不论哪一层的子表，都带有一个表头结点。
 - 优点——便于操作。
- **表中结点的层次分明**
 - 所有位于同一层的表元素，在其存储表示中也在同一层。
- 最高一层的表结点个数（除表头结点外）即为表长度。

广义表的类定义

```
template <class Type> class GenList;

template <class Type> class GenListNode { //广义表结点类定义
friend class Genlist;
private:
    int utype; // = 0/1/2
    GenListNode <Type> *tlink; //指向同一层下一结点的指针
    union { //联合
        int ref; // utype = 0, 存放引用计数
        Type value; // utype = 1, 存放数值
        GenListNode <Type> *hlink; // utype = 2, 存放指向子表的指针
    } info;
public:
    GenListNode ( ) : utype (0), tlink (NULL), info.ref (0) { }
    GenListNode ( GenListNode <Type> &RL ) {
        utype = RL.utype; tlink = RL.tlink; info = RL.info; }
};
```



```
template <class Type> class GenList { //广义表类定义
private:
    GenListNode <Type> *first; //广义表头指针
    GenListNode <Type> * Copy ( GenListNode <Type> *ls );
    //复制一个 ls 指示的无共享非递归表
    int Depth ( GenListNode <Type> *ls );
    //计算由 ls 指示的非递归表的深度
    int Length ( GenListNode <Type> *ls );
    //计算由 ls 指示的非递归表的长度
    bool Equal ( GenListNode <Type> *s, GenListNode <Type> *t );
    //比较以 s 和 t 为表头的两个表是否相等
    void Remove ( GenListNode <Type> *ls );
    //释放以 ls 为表头结点的广义表
    void CreateList ( istream &in, GenListNode <Type> *&ls,
        SeqList <Type> &L1, SeqList <GenListNode <Type> *> &L2 );
    //从输入流对象输入广义表的字符串描述
    //建立一个带附加头结点的广义表结构
```

public:

Genlist (); //构造函数

~GenList (); //析构函数

bool Head (info &x); //返回表头元素

bool Tail (GenList <Type> <); //返回表尾

GenListNode <Type> * First (); //返回第一个元素

GenListNode <Type> * Next (GenListNode <Type> *elem);

//返回表元素 elem 的直接后继元素

void Copy (const GenList <Type> &R);

//返回一个以 x 为头, list 为尾的新广义表

int Length (); //计算广义表的长度

int Depth (); //计算广义表的深度

friend istream & operator >> (istream &in,
GenList <type> &L);

};

广义表的访问算法

广义表结点类的存取成员函数

```
template <class Type> GenListNode & GenListNode <Type> ::
```

```
Info ( ) { //返回表元素的值
```

```
    GenListNode *pitem = new GenListNode;
```

```
    pitem->utype = utype;
```

```
    pitem->info.value = info.value;
```

```
    return *pitem;
```

```
}
```

```
template <class Type> void GenListNode <Type> ::
```

```
SetInfo ( GenListNode <Type> *elem,
```

```
        GenListNode <Type> &x ) { //修改表元素的值为 x
```

```
    elem->utype = x.utype;
```

```
    elem->info.value = x.info.value;
```

```
}
```

广义表类的构造和访问成员函数

```
template <class Type> Genlist <Type> ::  
GenList ( ) { //构造函数  
    GenListNode *first = new GenListNode;  
    first->utype = 0;  
    first->info.ref = 1;  
    first->tlink = NULL;  
    assert ( first!=NULL );  
}
```

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

```
template <class Type> bool GenList <Type> ::  
Tail ( GenList <Type> &lt ) {  
//若广义表非空， 则返回广义表除第一个元  
//素外其它元素组成的表， 否则函数没有定义  
if ( first->tlink == NULL ) return false;  
else {  
    lt.first->utype = 0;  
    lt.first->info.ref = 0;  
    lt.first->tlink = Copy ( first->tlink->tlink );  
}  
return true;  
}
```

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

```
template <class Type> GenListNode * GenList <Type> ::  
Next ( GenListNode <Type> *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

广义表的复制算法

- 任何一个非空的广义表均可分为表头、表尾两个部分，则一对确定的表头和表尾可唯一确定一个广义表。
- 复制一个广义表时，需要分别复制其表头和表尾，然后再合成，其前提是广义表不是共享表或递归表。


```
template <class Type> void GenList <Type> ::
```

```
Copy ( const GenList <Type> &R ) { //公有函数
```

```
    first = Copy ( R.first );
```

```
}
```

```
template <class Type> GenListNode * GenList <Type> ::
```

```
Copy ( GenListNode <Type> *ls ) { //私有函数
```

```
    GenListNode <Type> *q = NULL;
```

```
    if ( ls != NULL ) {
```

```
        q = new GenListNode <Type>;
```

```
        q->utype = ls->utype;
```

```
        switch ( ls->utype ) {
```

```
            case 0: q->info.ref = ls->info.ref; break;
```

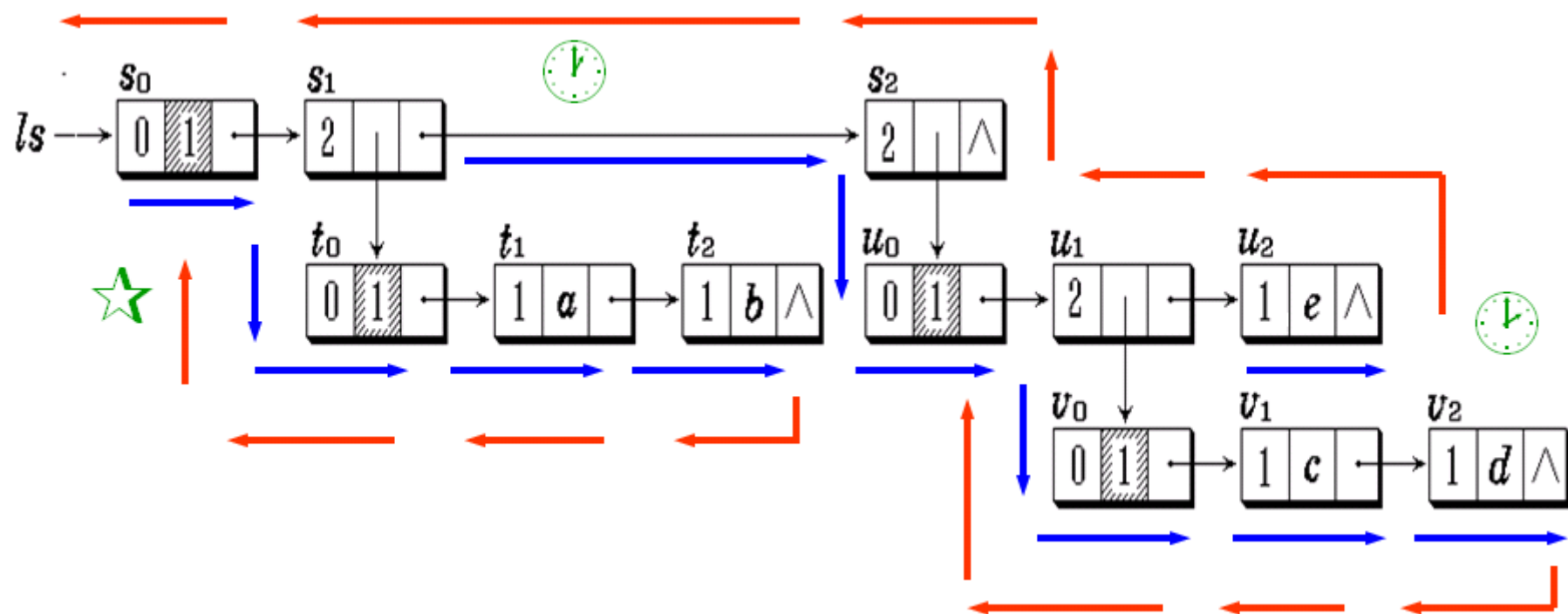
```
            case 1: q->info.value = ls->info.value; break;
```

```
            case 2: q->info.hlink = Copy ( ls->info.hlink ); break; }
```

```
        q->tlink = Copy ( ls->tlink );    }
```

```
    return q;
```

```
}
```



求广义表的长度

```
template <class Type> int GenList <Type> ::  
Length ( ) { //公有函数  
    return Length ( first->tlink );  
}
```

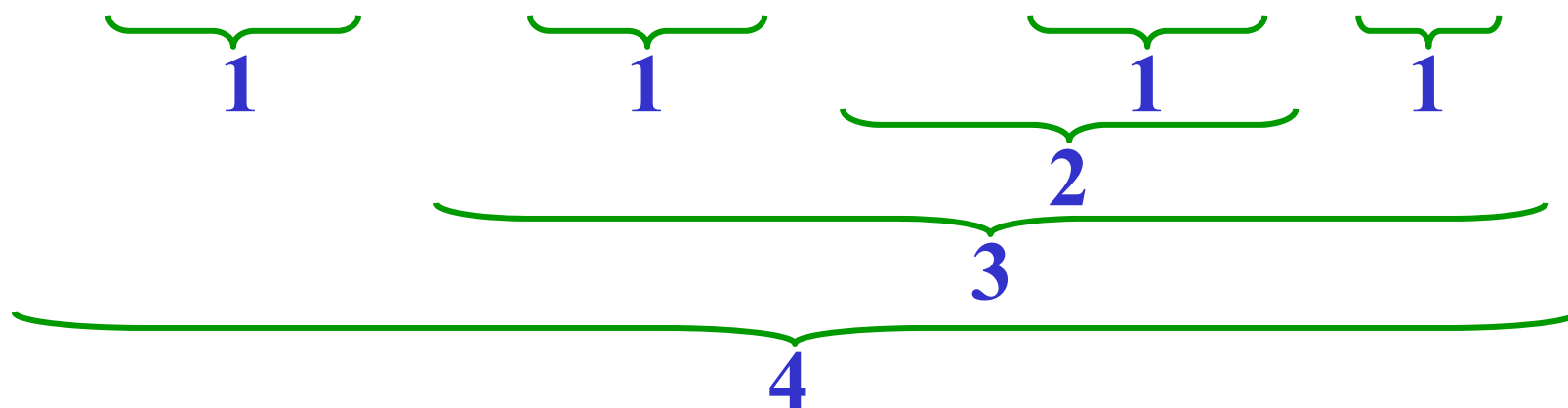
```
template <class Type> int GenList <Type> ::  
Length ( GenListNode <Type> *ls ) { //私有函数  
    if ( ls != NULL ) return 1+Length ( ls->tlink );  
    else return 0;  
}
```

求广义表的深度

$$\text{Depth}(LS) = \begin{cases} 1, & \text{当LS为空表时} \\ 0, & \text{当LS为原子时} \\ 1 + \max_{0 \leq i \leq n-1} \{\text{Depth}(a_i)\}, & \text{其他, } n \geq 1 \end{cases}$$

例如，对于广义表

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



```
template <class Type> int GenList <Type> :: Depth ( ) {  
    return Depth ( first );  
}  
  
template <class Type> int GenList <Type> ::  
Depth ( GenListNode <Type> *ls ) {  
    if ( ls->tlink == NULL ) return 1; //空表  
    GenListNode <Type> *temp = ls->tlink; int m = 0, n;  
    while ( temp != NULL ) { //在表顶层横扫  
        if ( temp->utype == 2 ) { //结点为表结点  
            n = Depth ( temp->info.hlink );  
            if ( m < n ) m = n; // m 记最大深度 }  
            temp = temp->tlink;  
        }  
        return m+1;  
    }  
}
```



```
template <class Type> void GenList <Type> ::  
DelValue ( GenListNode <Type> *ls, Type x )  
{ //在广义表中删除所有含 x 的结点  
    if ( ls->tlink != NULL ) { //非空表  
        GenListNode <Type> *p = ls->tlink;  
        while ( p != NULL && //横扫链表  
                ( ( p->utype == 1 && p->info.value == x ) ) {  
            ls->tlink = p->tlink; delete p; //删除  
            p = ls->tlink; //指向同一层后继结点 }  
        if ( p != NULL ) {  
            if ( p->utype == 2 ) //在子表中删除  
                DelValue ( p->info.hlink, x );  
            DelValue ( p, x ); //在后续链表中删除 }  
        }  
    }  
}
```

- **对于共享表**

- 如果想删除某一个子表，要看它是否为几个表共享。
- 如果一个表元素有多个地方使用，删除操作可能会造成其它地方使用出错。
- 在表头结点中设置一个引用计数，当要做删除时先把该引用计数减1，只有引用计数减到0时才能执行结点的真正释放。


```
template <class Type> GenList <Type> :: ~GenList ( ) {
```

```
//析构函数，每个表头结点都有一个引用计数
```

```
    Remove ( first );
```

```
}
```

```
template <class Type> void GenList <Type> ::
```

```
Remove ( GenListNode <type> *ls ) {
```

```
    ls->info.ref --; //引用计数减一
```

```
    if ( ls->info.ref <= 0 ) { //引用计数减至0才能删除
```

```
        GenListNode <Type> *q;
```

```
        while ( ls->tlink != NULL ) { //横扫链表
```

```
            q = ls->tlink;
```

```
            if ( q->utype == 2 ) { //遇到子表
```

```
                Remove ( q->info.hlink ); //递归删除子表
```

```
                if ( q->info.hlink->info.ref <= 0 )
```

```
                    delete q->info.hlink; }
```

```
            ls->tlink = q->tlink; delete q; }
```

```
        }
```

```
    }
```

判断两个广义表是否相等的算法

- **判断两个广义表是否相等**
 - **不但两个广义表具有相同的结构，而且对应的数据成员具有相等的值。**
 - 如果两个广义表都是空表，则相等。
 - 如果两个广义表的对应结点都是原子结点，对应项的值相等，再递归比较同一层后面的表元素。
 - 如果两个广义表中对应项是子表结点，则递归比较相应的子表。

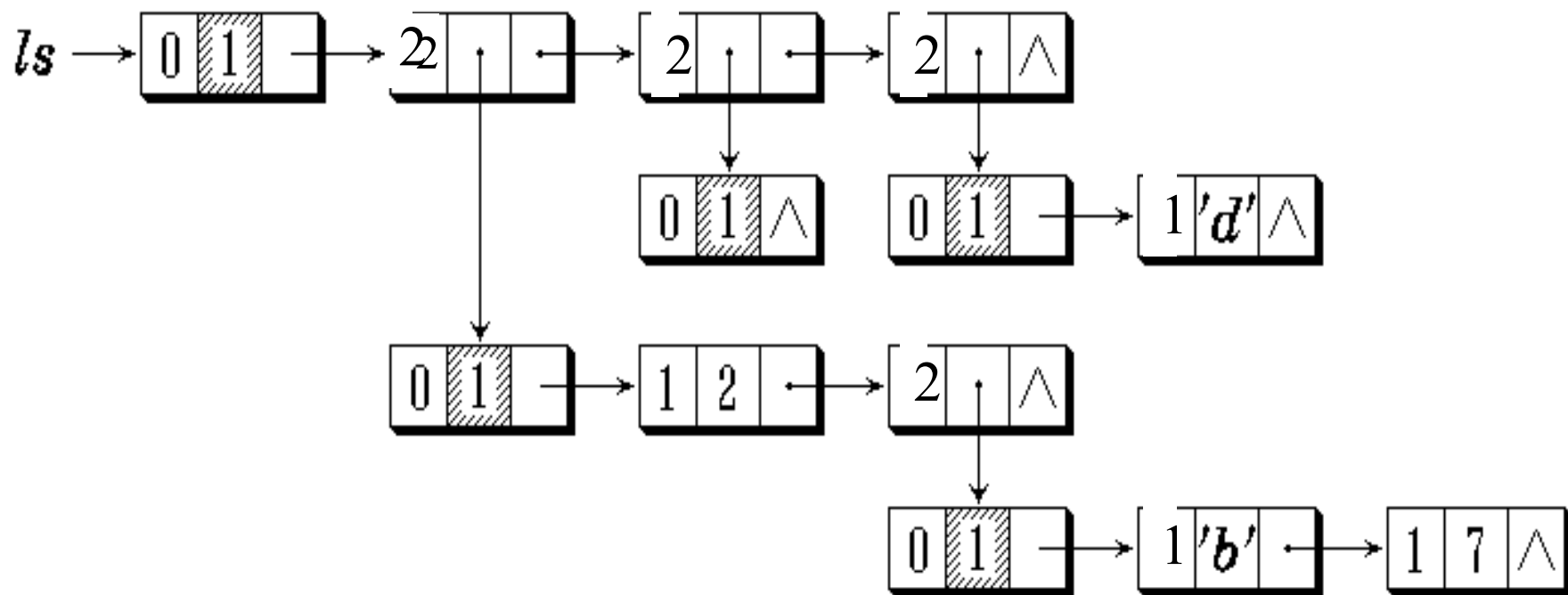
```
template <class Type> bool GenList <Type> ::  
Equal ( GenListNode <Type> *s, GenListNode <Type> *t ) {  
    int x;  
    if ( s->tlink == NULL && t->tlink == NULL )  
        return true; //表 s 与表 t 都是空表或者所有结点都比较完  
    if ( s->tlink != NULL && t->tlink != NULL &&  
        s->tlink->utype == t->tlink->utype )  
    { //两表都非空且结点标志相同  
        if ( s->tlink->utype == 1 ) //原子结点, 比较对应数据  
            x = (s->tlink->info.value == t->tlink->info.value) ? 1 : 0;  
        else if ( s->tlink->utype == 2 ) //原子结点, 比较对应数据  
            x = Equal ( s->tlink->info.hlink, t->tlink->info.hlink );  
        if ( x == 1 ) return Equal ( s->tlink, t->tlink );  
    }  
    return false;  
}
```

从字符串 s 建立广义表的链表表示 ls

- 对于从输入流对象输入的字符，检测其内容。
 - 大写字母表示的表名，首先检查其是否存在，如是则说明是共享表，将相应附加头结点的引用计数加一；如果不是，则保存该表名并建立相应广义表。
 - 表名后一定是左括号‘(’，不是则输入错，是则递归建立广义表结构。
 - 如果遇到小写字母表示的原子，则建立原子元素结点；如果遇到右括号‘)’，子表链收尾并退出递归。
 - 注意，空表情形括号里应加入一个非英文字母，如‘#’，不能一个字符也没有。

设 $str = “((2, ('b', 7)), (), ('d'))”$

得到的广义表链表结构



“ $D(B(a, b), C(u, (x, y, z)), A(\#));$ ”

```
template <class Type> void GenList <Type> ::  
CreateList ( istream in, GenListNode <Type> *&ls,  
            SeqList <Type> &L1, SeqList <GenListNode <Type> *> L2 ) {  
//在表 L1 存储大写字母的表名  
//在表 L2 中存储表名对应子表结点的地址  
    Type chr;  
    in >> chr;  
    if ( isalpha(chr) && isupper(chr) || che == '(' ) {  
        ls = new GenListNode <Type>; //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 <Type> *p = L2.Locate ( m ); //查子表地址  
                p->ref++; //表引用计数加 1 }  
            else { L1.Insert ( n, chr ); L2.Insert ( n, ls ); } //保存表名及地址
```

```
    in >> chr;
    if ( chr != '(' ) eixt(1); //表名后必跟 '('
}
ls->info.hlink = new GenListNode <Type>; //建附加头结点
ls->info.hlink->utype = 0; ls->info.hlink->ref = 1;
CreateList ( in, ls->info.hlink->tlink, L1, L2 ); //递归建子表
CreateList ( in, ls, L1, L2 ); //递归建后继表
}
else if ( isalpha(chr) && islower(chr) ) { //建原子结点
    ls = new GenListNode <Type>; 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; //右括号, 链收尾
else if ( chr == '#' ) ls == NULL; //空表, 链收尾
}
```



随堂练习

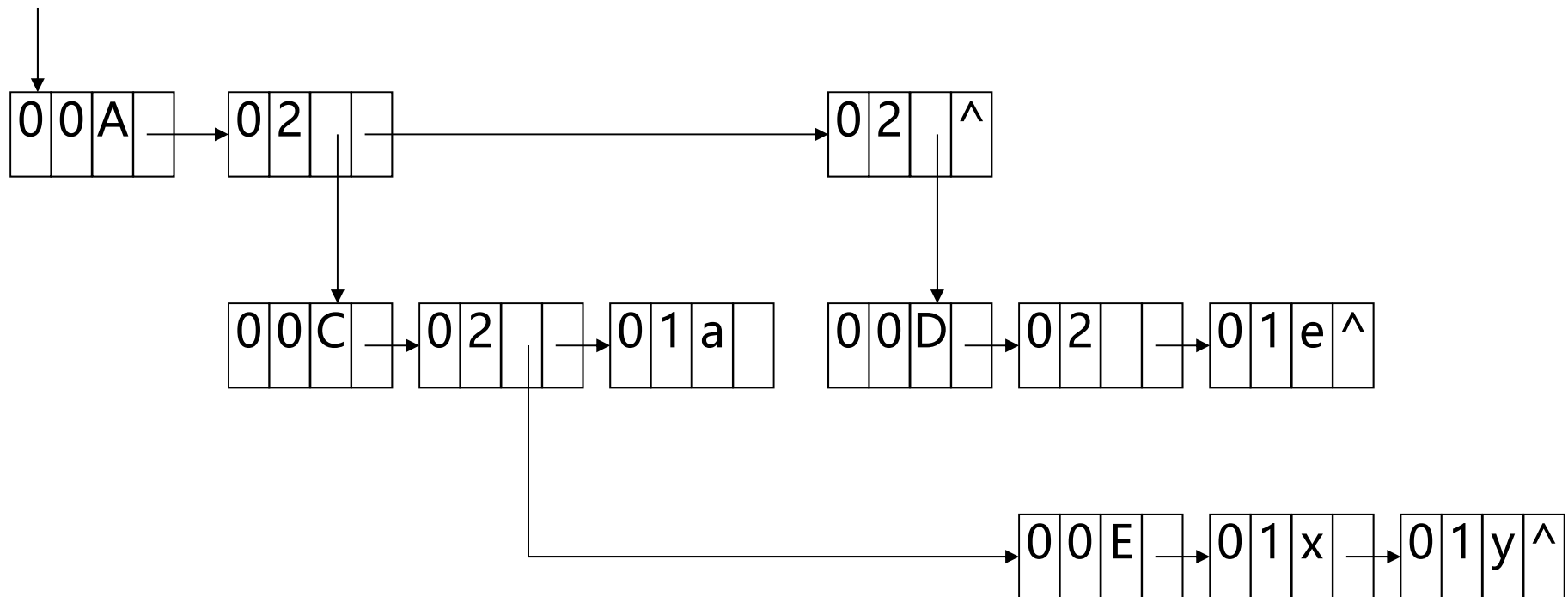
例1：若广义表满足 $\text{Head}(A)=\text{Tail}(A)$ ，则A为 $(())$ ，其长度和深度为_____和_____。

例2：广义表具有可共享性，因此在遍历一个广义表时必须为每一个结点增加一个标志域mark，以记录该结点是否访问过。一旦作了mark，以后就不再访问该结点。

- (1) 定义该广义表的类结构；
- (2) 采用递归算法对一个非递归的广义表进行遍历；
- (3) 试使用一个栈，实现一个非递归算法，对一个非递归广义表进行遍历。

例1：若广义表满足 $\text{Head}(A)=\text{Tail}(A)$ ，则A为 $(())$ ，其长度和深度为 1 和 2。

例2：



三种结点类型：表头结点，原子结点，子表结点。

```
class GenList;  
class GenListNode  
{  
    friend class GenList;  
    private:
```

```
        int mark, utype;
```

```
        //utype=0/1/2, mark是访问标记
```

```
        GenListNode* tlink; //指向同层下一结点
```

```
        union{
```

```
            char listname; //utype=0,表头结点存表名
```

```
            char atom; //utype=1,存原子结点数据
```

```
            GenListNode* hlink; //指向子表
```

```
        }value;
```

简化约定：

a. 原子结点数据为字符型，并且不用大写字母

b. 表名用大写字母表示，存于表头结点

public:

GenListNode(int tp,char info)

: mark(0),utype(tp),tlink(NULL) {
 if (utype==0) value.listname=info;
 else value.atom=info;

} //表头或原子结点构造函数

GenListNode(GenListNode * hp)//子表构造函数

: mark(0), utype(2), value.hlink(hp) {}

char Info(GenListNode * elem) {

 //返回表元素elem的值

 return(utype==0)? elem->value.listname
 : elem->value.atom;

}

};

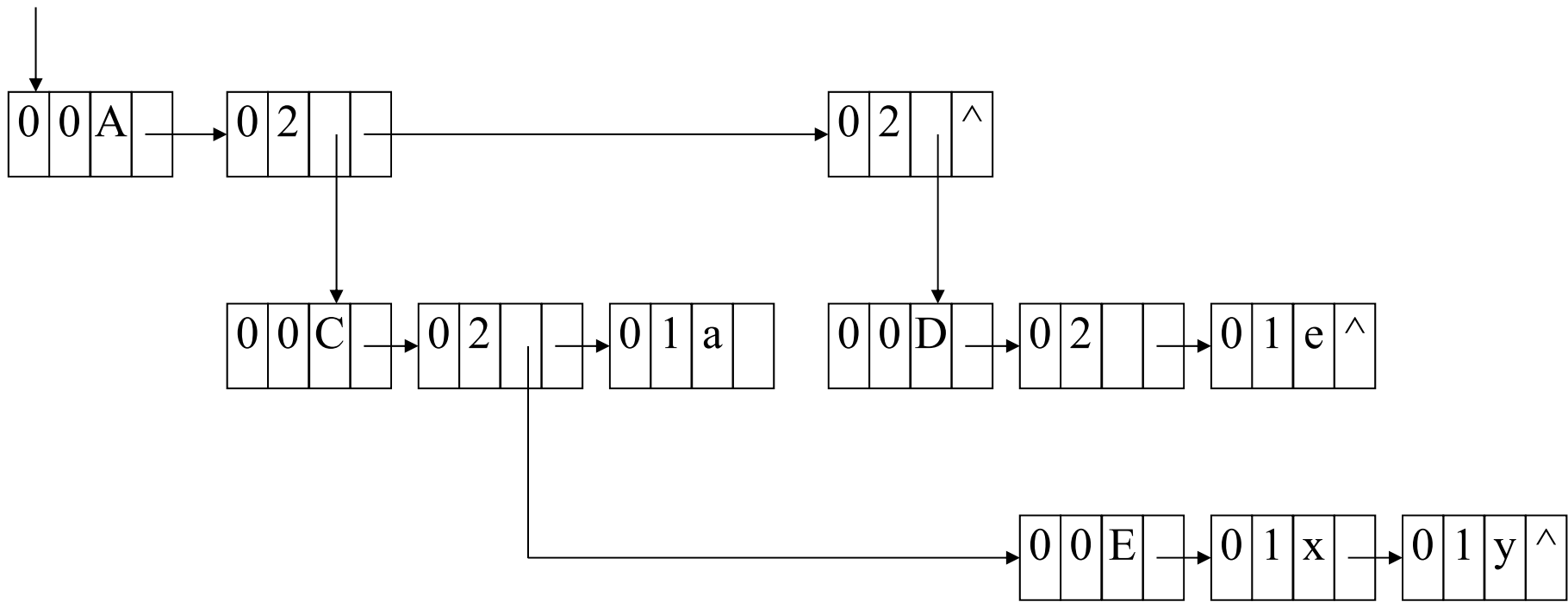
```
class GenList { //广义表类定义
private:
    GenListNode *first;
    void traverse(GenListNode * ls);
    //遍历广义表
    void Remove(genListNode * ls);
    //释放广义表
public:
    GenList( char& value);
        //value是指定的停止建表标志数据
    ~GenList();
    void traverse();
}
```

递归遍历广义表

```
void GenList::traverse() {  
    traverse(first);  
}
```

```
void GenList::traverse(GenListNode *ls)
{
    if(ls!=NULL) {
        ls->mark=1;
        if(ls->utype==0) cout<<
            ls->value.listname<<'(';
        //表头结点
        else if (ls->utype==1) //原子结点
        {
            cout<<ls->value.atom;
            if(ls->tlink!=NULL) cout<<',';
        }
    }
}
```

```
else if(ls->utype==2) { //子表结点
    if(ls->value.hlink->mark==0)
        traverse(ls->value.hlink);
    else cout<<ls->value.hlink
        ->value.listname;
    if(ls->tlink!=NULL) cout<<',';
}
traverse(ls->tlink);
} //endif(ls!=NULL)
else cout<<')';
} //end function
```



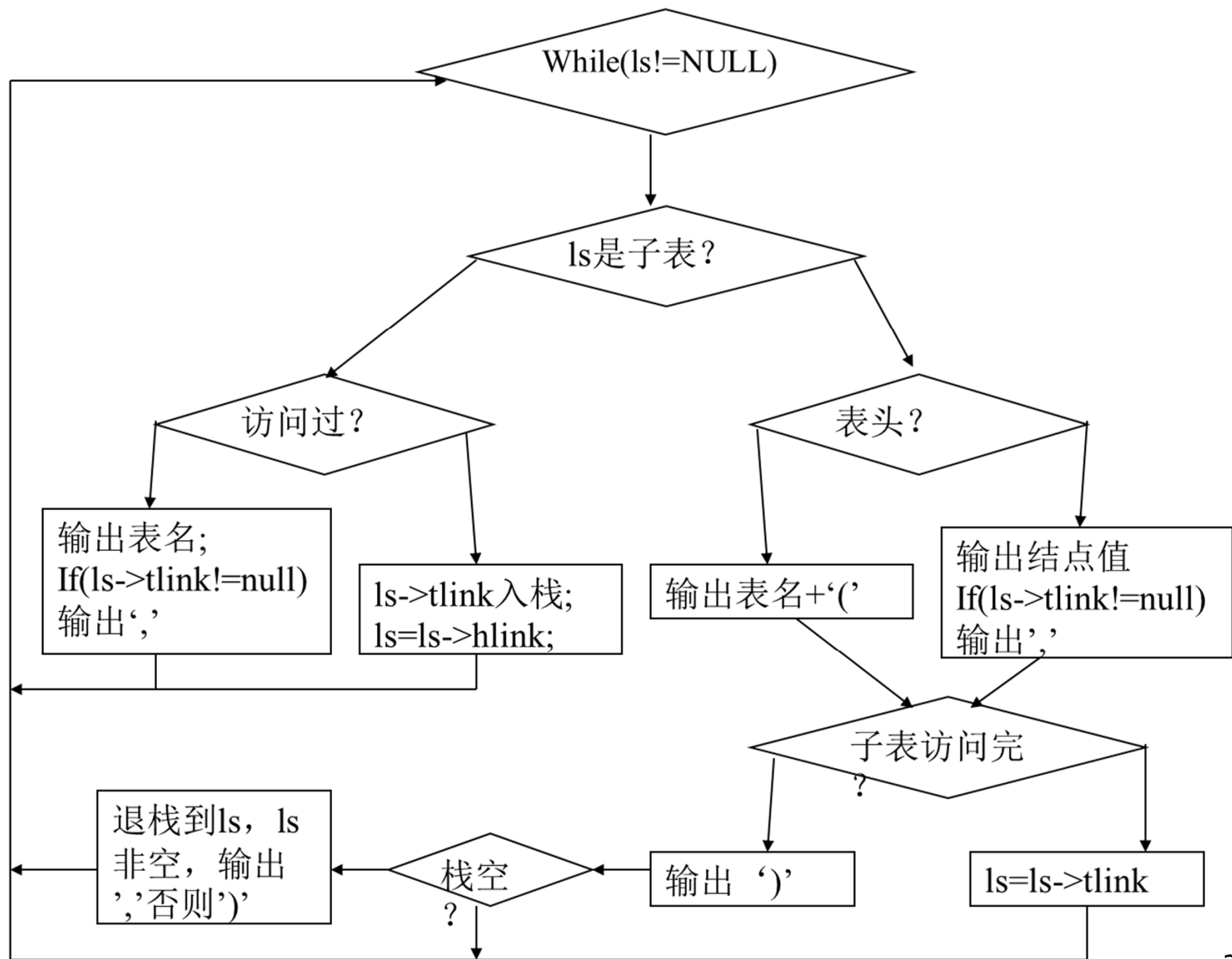
$A(C(E(x, y), a), D(E, e))$

非递归遍历

利用栈实现

栈中存放回退时下一将访问的结点地址tlink.

```
Void GenList::traverse(GenListNode *ls)
```



```

void GenList :: traverse ( GenListNode *ls ) {
    Stack <GenListNode<Type> *> st;
    while ( ls != NULL ) {
        ls->mark = 1;
        if ( ls->utype == 2 ) { //子表结点
            if ( ls->value.hlink->mark == 0 )
                //该子表未访问过
            {
                st.Push( ls->tlink ); ls = ls->value.hlink;
                //暂存下一结点地址, 访问子表
            }
        }
        else {
            cout << ls->value.hlink->value.listname;
            //该子表已访问过, 仅输出表名
            if ( ls->tlink != NULL )
            {
                cout << ','; ls = ls->tlink;
            }
        }
    }
}

```

```

else { //非子表结点
    if ( ls->utype == 0 )
        cout << ls->value.listname << '('; //表头结点
    else if ( ls->utype == 1 ) { //原子结点
        cout << ls->value.atom;
        if ( ls->tlink != NULL ) cout << ',';
    }
    if ( ls->tlink == NULL ) { //子表访问完, 子表结束处理
        cout << ')';
        if ( st.IsEmpty( ) == 0 ) { //栈不空
            ls = st.GetTop ( ); st.Pop ( ); //退栈
            if ( ls != NULL ) cout << ',';
            else cout << ')';
        }
    }
    else ls = ls->tlink; //向表尾搜索
}
}

```



本章小结

- 知识点
 - 数组及其压缩存储方式
 - 字符串及**KMP**算法
 - 广义表及其递归算法

- **课程习题**

- **笔做题——4.3, 4.8, 4.12, 4.15, 4.16**
(以作业形式提交)
- **上机题——4.13, 4.17**
- **思考题——4.6, 4.14**