北京航空航天大學
BEIHANG UNIVERSITY

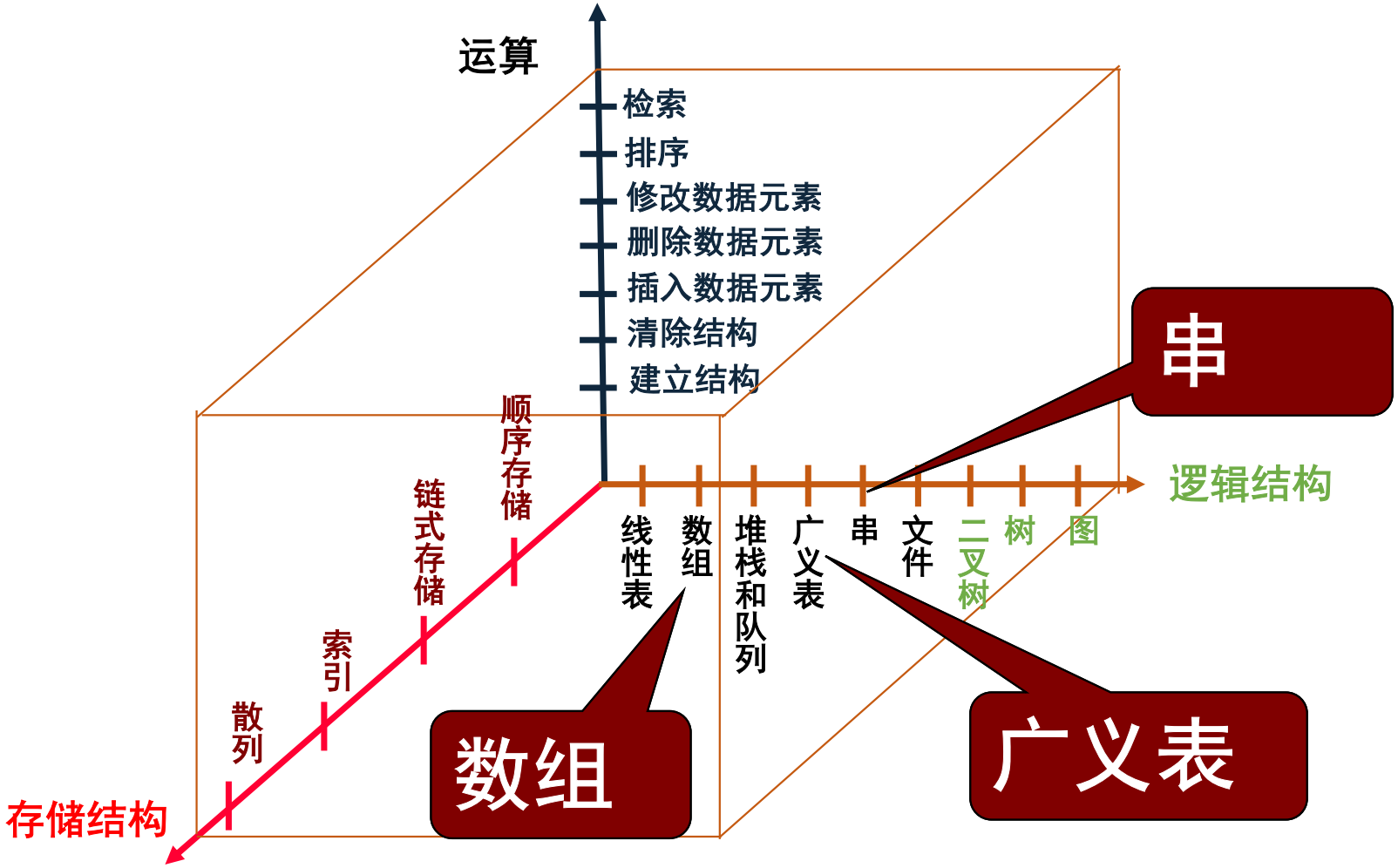
数据结构与程序设计
(Data Structure and Programming)

数据结构

广义表、矩阵与串

北航计算机学院 晏海华

数据结构的基本问题空间




数组*

本章内容

数组的基本概念
数组的存储结构
特殊矩阵的压缩存储
稀疏矩阵的三元组表表示
稀疏矩阵的十字链表表示
数组的应用示例

数组的基本概念

（一）数组的定义

数组是下标与值组成的偶对的有穷集合。

（二） 数组的基本操作

对数组的操作是基于下标进行的

1. 给定一组下标，存取或修改相应元素的值。
2. 检索满足条件的数组元素。
3. 对数组元素进行排序。

一般情况下，没有“改变数组规模”的插入、删除运算。
数组的规模是固定的，为静态结构，需要预先分配内存。

数组的存储结构

（一）一维数组A[1..n]

$A[1..n] = (\ a_1,\ a_2,\ a_3,\ \ldots,\ a_{n-1},\ a_n\ )$

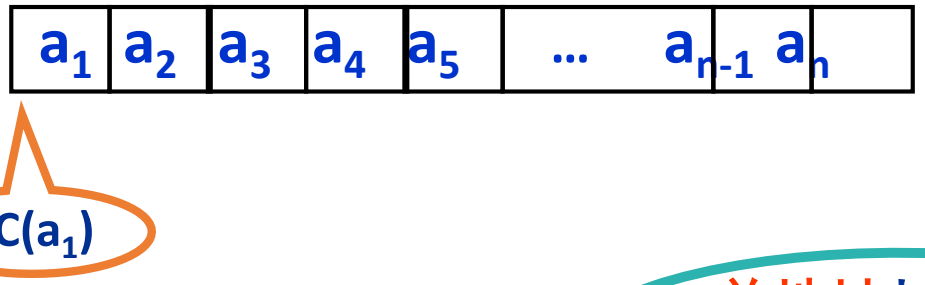
典型的线性结构

(1) 一维数组的特点

除了第一个元素外，其他每个元素有且仅有一个直接前驱元素；

除了最后那个元素外，其他每个元素有且仅有一个直接后继元素。

(2) 一维数组的存储



首地址加上被求元素前面的所有元素占用的单元数

若已知每个元素占k个存储单元，并且第一个元素的存储地址$LOC(a_1)$，则

$$LOC(a_i) = LOC(a_1) + (i-1) \times k$$

（二） 二维数组A[1..m,1..n]

$$A[1..m,1..n] = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ & & \cdots & \cdots & & \\ & & \cdots & \cdots & & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & \cdots & a_{mn} \end{pmatrix}$$

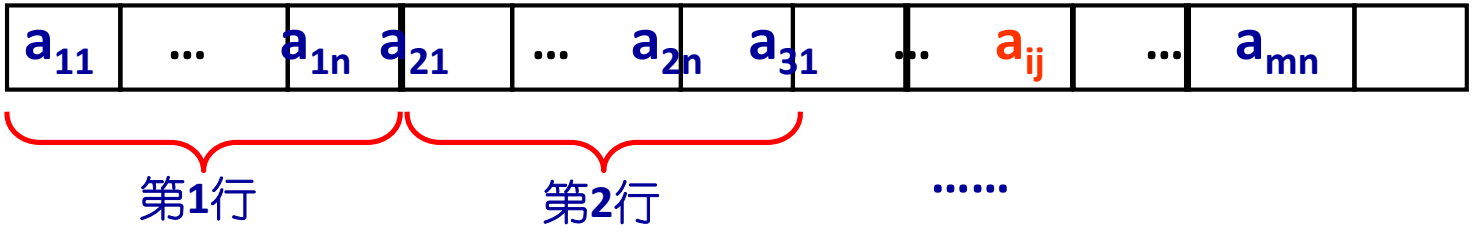
二维数组的存储

行序为主序分配方式

列序为主序分配方式

(1) 行序为主序分配方式

$$A[1..m,1..n] = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ & & \cdots & \cdots & & \\ & & \cdots & \cdots & & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & \cdots & a_{mn} \end{pmatrix}$$



特点

前一行最后一个元素的存储位置与后一行的第一个元素的存储位置相邻。

对于二维数组

$$A[1..m,1..n] = \begin{matrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & \cdots & a_{3n} \\ \cdots & \cdots & & & & \\ \cdots & \cdots & & a_{ij} & & \\ \cdots & \cdots & & & & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & \cdots & a_{mn} \end{matrix}$$

i-1行

j-1个元素

首地址加上被求元素前面的所有元素占用的单元数

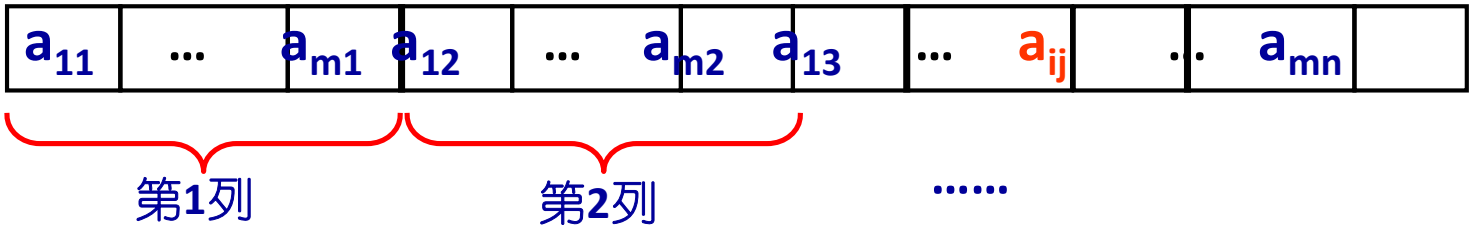
若已知每个元素占k个存储单元，并且第一个元素的存储地址$LOC(a_{11})$，则

$$LOC(a_{ij}) = LOC(a_{11}) + (i-1)\times n\times k + (j-1)\times k$$

$$= LOC(a_{11}) + [\,(i-1)\times n+(j-1)\,]\times k$$

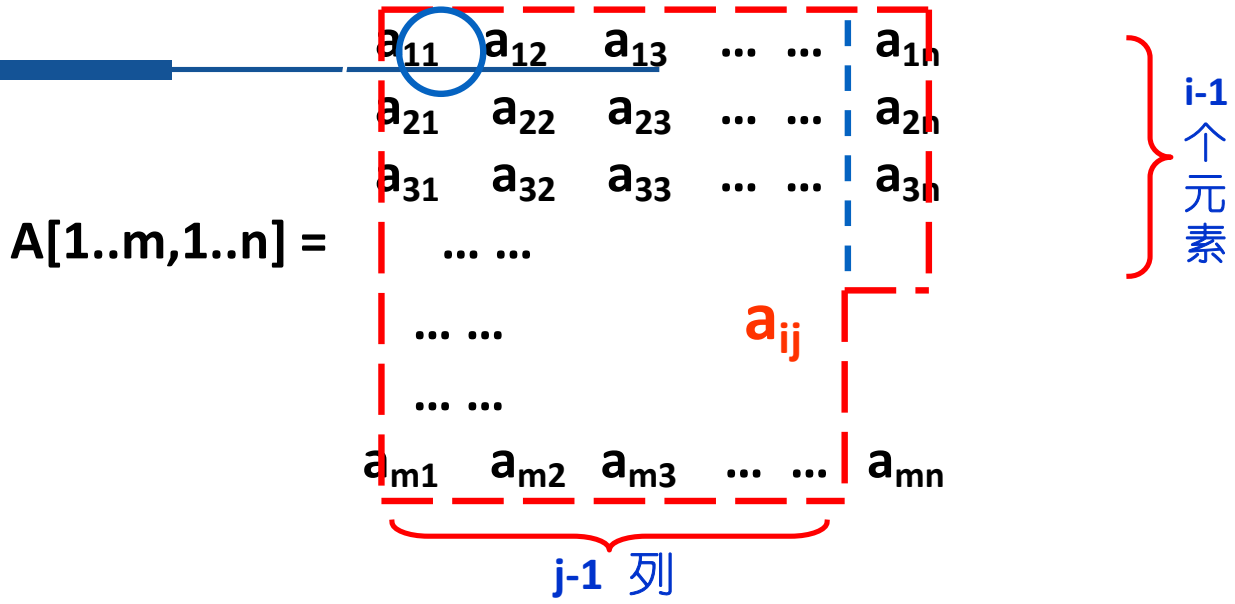
（2）列序为主序分配方式

$$A[1..m,1..n] = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ & & \cdots & \cdots & & \\ & & \cdots & \cdots & & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & \cdots & a_{mn} \end{pmatrix}$$



特点

前一列最后一个元素的存储位置与后一列的第一个元素的存储位置相邻。



若已知每个元素占k个存储单元，并且第一个元素的存储地址$LOC(a_{11})$，则

$$LOC(a_{ij}) = LOC(a_{11}) + (j-1)\times m\times k + (i-1)\times k$$

$$= LOC(a_{11}) + [\,(j-1)\times m+(i-1)\,]\times k$$

特殊矩阵的压缩存储

目的：节省存储空间

所谓**压缩存储**是指为多个值相同的元素，或者位置分布有规律的元素分配尽可能少的存储空间，而对0元素一般情况下不分配存储空间。

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ & \cdots & \cdots & & \\ & \cdots & \cdots & & \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}$$

A[0..m−1][0..n−1]

传统做法

（一）对称矩阵的压缩存储

一个n阶矩阵A的元素满足性质

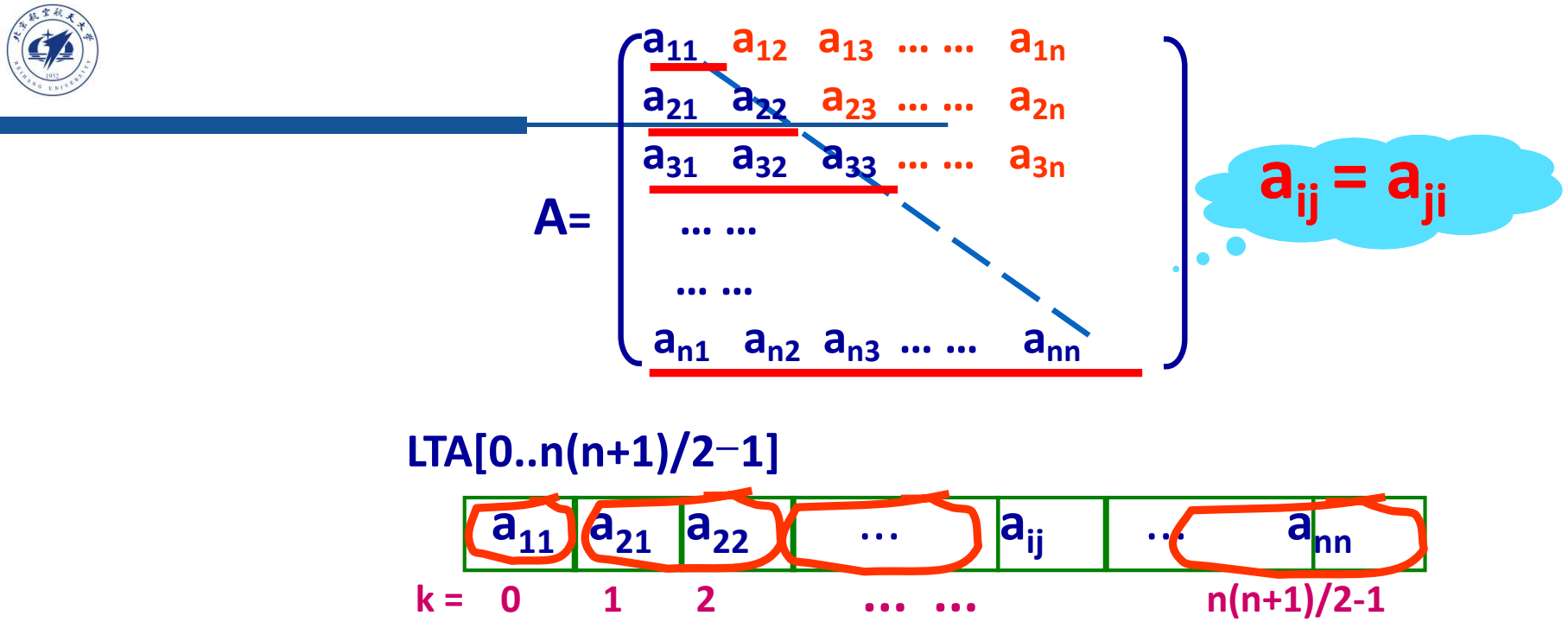
$a_{ij} = a_{ji}$　　$1 \le i, j \le n$

则称矩阵A为n阶**对称矩阵**。

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & \cdots & a_{3n} \\ \cdots & \cdots & & & & \\ \cdots & \cdots & & & & \\ a_{n1} & a_{n2} & a_{n3} & \cdots & \cdots & a_{nn} \end{pmatrix}$$

传统做法

定义一个二维数组A[0..n−1][0..n−1]



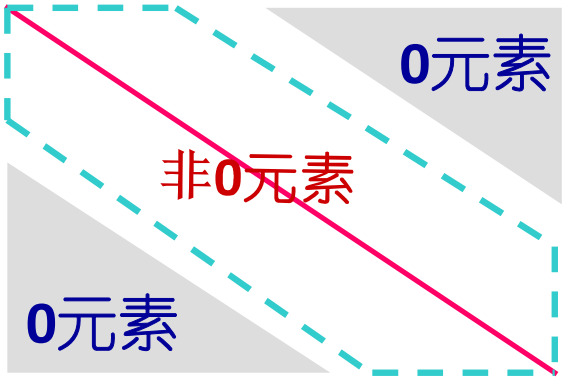
A中任意一元素a_{ij}与LTA[k] 之间存在对应关系:

$$k=\begin{cases} i\times(i-1)/2+j-1 & \text{当}i\geqslant j\text{时} \\ j\times(j-1)/2+i-1 & \text{当}i<j\text{时} \end{cases}$$

（二）对角矩阵的压缩存储

若一个矩阵中，值非0的元素对称地集中在主对角线两旁的一个带状区域中（该区域之外的元素都为0元素），称这样的矩阵为 **对角矩阵**。

B =



n × n

传统做法

定义一个二维数组B[0..n−1][0..n−1]

例. 三对角矩阵的压缩存储

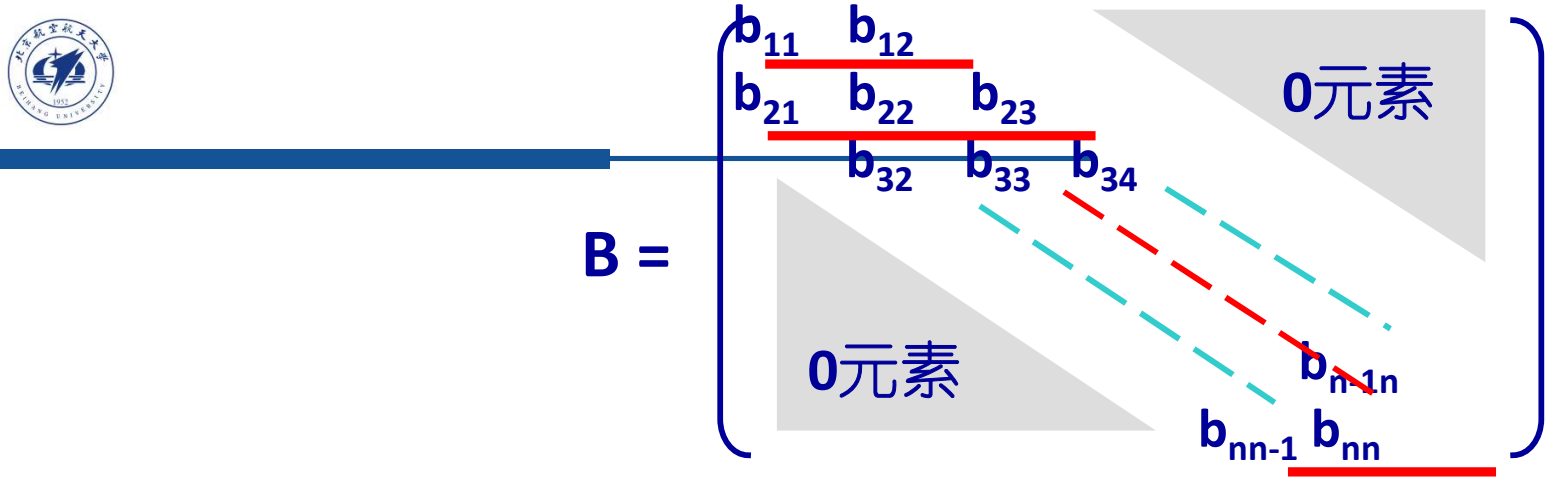
$$B=\begin{pmatrix} b_{11} & b_{12} & & & & \\ b_{21} & b_{22} & b_{23} & & & \\ & b_{32} & b_{33} & b_{34} & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-1n-1} & b_{n-1n} & \\ & & & b_{nn-1} & b_{nn} \end{pmatrix}$$

0元素

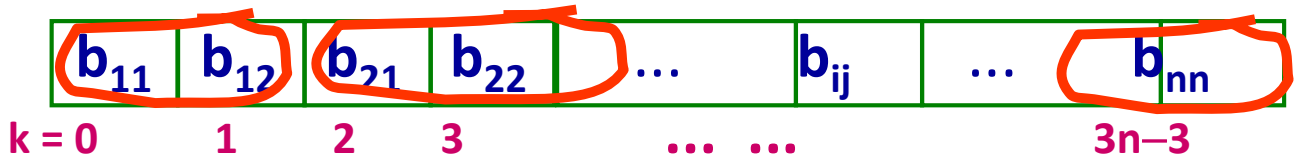
0元素

有多少个非零元素？

有3n−2个非零元素



LTB[0..3n–3]



B中任一非零元素 b_{ij} 与LTB[K]之间存在对应关系

$$k = 2 \times i + j - 3$$

《计算机与数字工程》 2001年06期， 杨文茂 刘明杰

主对角线两边非对称分布的带状稀疏矩阵的压缩存储通用寻址公式

稀疏矩阵的三元组表表示

（一）什么是稀疏矩阵?

一个较大的矩阵中，零元素的个数相对于整个矩阵元素的总个数所占比例较大时，可以称该矩阵为一个**稀疏矩阵**。

$$A = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

传统做法

定义一个二维数组B[0..5][0..5]

（二）稀疏矩阵的三元组表表示

三元组 (i, j, value)

$$\begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{pmatrix}$$

例如

(1,1,15) 表示第1行、第1列、值为15的元素；

(1,4,22) 表示第1行、第4列、值为22的元素；

(1,6,−15) 表示第1行、第6列、值为−15的元素；

(2,2,11) (2,3,3) (3,4,−6)

(5,1,91) (6,3,28)

一个特殊的三元组

(m, n, t)

其中，**m, n, t** 分别表示稀疏矩阵的总的行数、总的列数与非零元素的总个数。

三元组表存储方法：

若一个**m×n**阶稀疏矩阵具有**t**个非零元素，则用**t+1**个三元组来存储，其中第一个三元组分别用来给出稀疏矩阵的总行数**m**、总列数**n**以及非零元素的总个数**t**；第二个三元组到第 **t+1** 个三元组按行序为主序的方式依次存储**t**个非零元素。

例

$$A=\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

A[0..5][0..5]

传统做法

三元组表

LTMB=

6	6	8	0
1	1	15	1
1	4	22	2
1	6	-15	3
2	2	11	4
2	3	3	5
3	4	-6	6
5	1	91	7
6	3	28	8

LTMB[0..8][0..2]

三元组表法

思考题

1．有人说，数组除了插入和删除操作以外，主要操作还有存取、修改、检索和排序，你认为如何？

2. 对于一个具有t个非零元素的 m×n 阶矩阵，若采用三元组表方法存储，则当t 满足什么条件，这样做才有意义？

答案

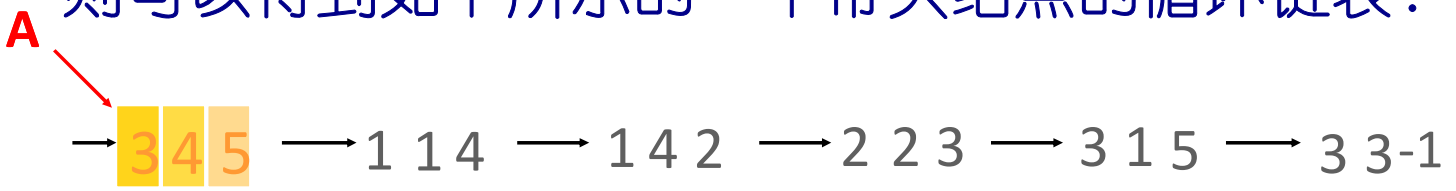
$$t < \frac{m \times n}{3} - 1$$

稀疏矩阵的链表表示 十字 链表表示*

例如，如下一个稀疏矩阵：

$$A = \begin{pmatrix} 4 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 5 & 0 & -1 & 0 \end{pmatrix}$$

若以行序为主序依次将所有非零元素链接起来
则可以得到如下所示的一个带头结点的循环链表：



缺点

要确定一个元素比较麻烦，导致相关操作效率低

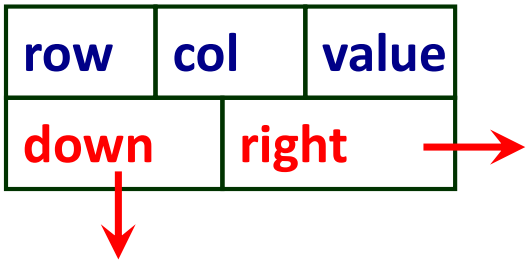
十字链表

为稀疏矩阵的每一行设置一个单独的循环链表，同样为每一列设置一个单独的循环链表。矩阵中每一个非零元素同时包含在两个循环链表中，即包含在它所在的行链表与所在的列链表中，即两个链表的交汇处。

链结点的结构会长成什么样？

对于一个m×n的稀疏矩阵，分别建立m个行的循环链表与n个列的循环链表，每个非零元素用一个链结点存储。

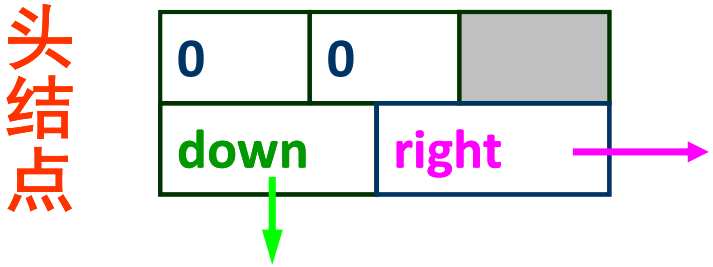
链结点的构造为：



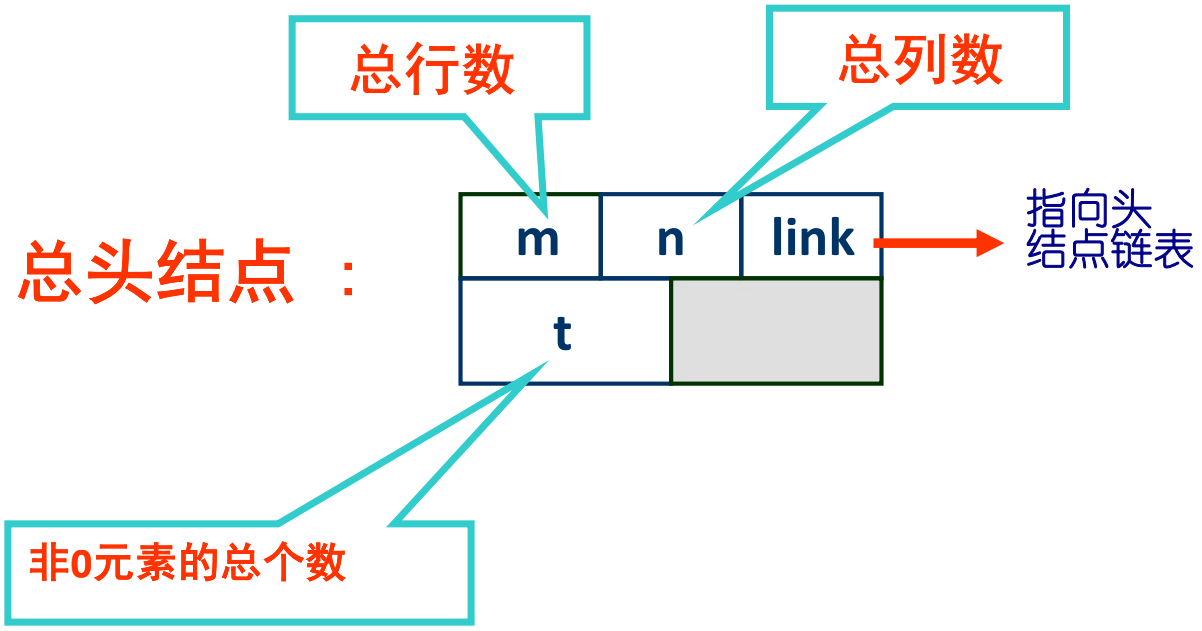
其中，**row, col, value** 分别表示某个非零元素所在的行号、列号和元素的值；**down** 和 **right** 分别为向下与向右指针，分别用来链接同一列中的与同一行中的所有非零元素对应的链结点。

对m个行链表，分别设置m个行链表**表头结点**。 表头结点的构造与链表中其他链结点一样， 只是令**row**与**col** 的值分别为**0**, **right**域指向相应行链表的第一个结链点. 同理, 对**n**个列链表，分别设置**n**个列链表表头结点指向相应列链表的第一个结链点。

一共设置**MAX(m,n)**个头结点，头结点构造为.



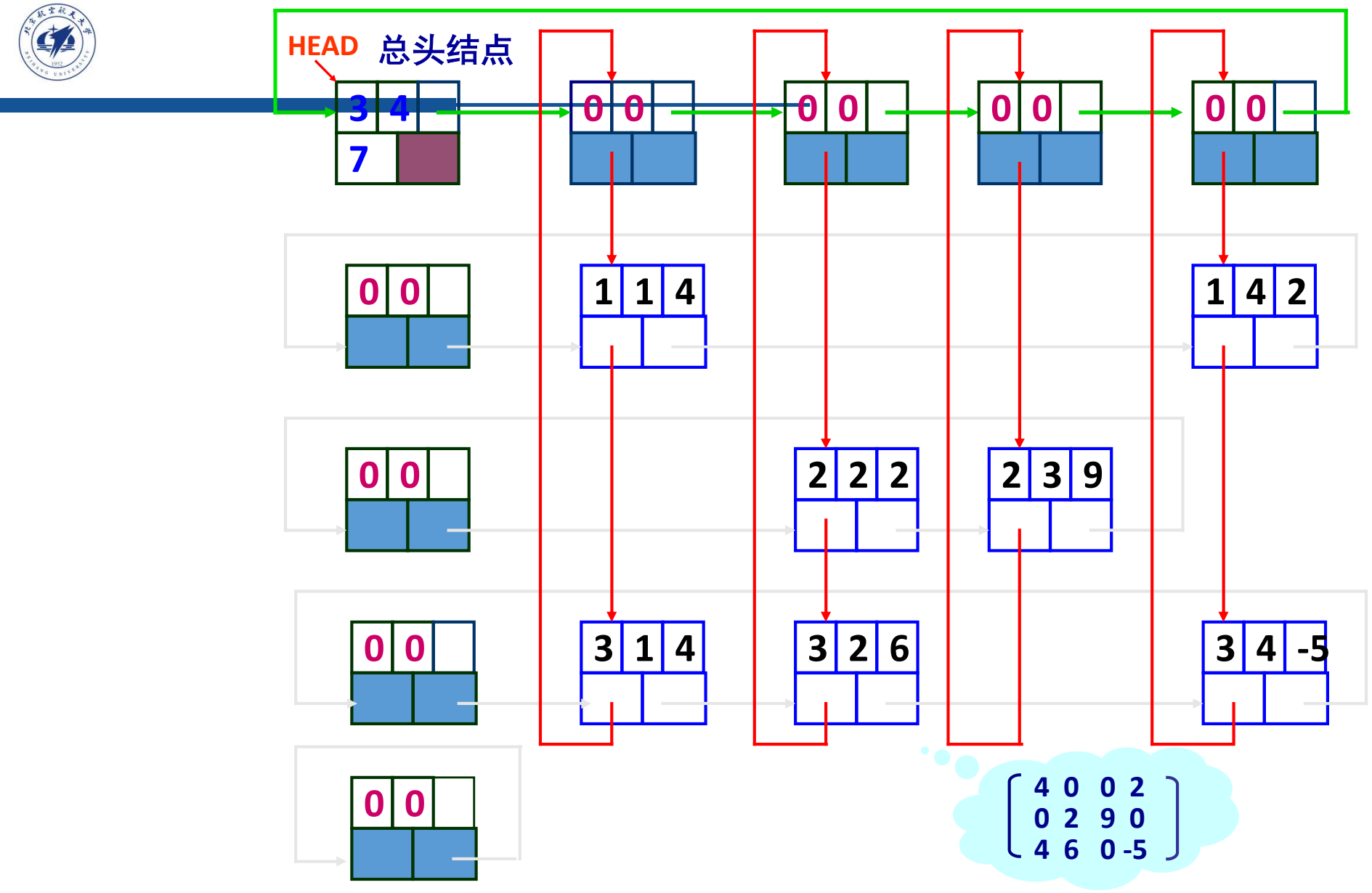
再设置一个总头结点(如下所示)，通过Value(即下图的Link)域把所有头结点也链接成一个循环链表。



对于如下稀疏矩阵B，

$$B = \begin{pmatrix} 4 & 0 & 0 & 2 \\ 0 & 2 & 9 & 0 \\ 4 & 6 & 0 & -5 \end{pmatrix}$$

十字链表表示如下：


HEAD
总头结点
3 4
7
0 0
1 1 4
1 4 2
2 2 2
2 3 9
3 1 4
3 2 6
3 4 -5
4 0 0 2
0 2 9 0
4 6 0 -5

思考题

要编写一款可供第三方独立使用的矩阵运算软件，应该从哪下手？需要提供哪几类文件？

(1) 筛选拟支持的矩阵运算的种类：如有相同行数和列数的矩阵间的加法、减法；符合矩阵乘法规则要求的矩阵间的乘法；方阵间的除法；方阵的求逆；矩阵的求转置矩阵等基本功能。

对称正定矩阵分解与行列式求值；矩阵的三角分解；实数矩阵的奇异值分解等高级功能。

(2) 采用适当的数据结构实现上述多种运算程序。

(3) 提供.h文件，.c或.cpp或.lib或.dll文件，软件工程要求的各类文档，至少应有用户手册或帮助文件。

数组的应用举例*

（一）一元n阶多项式的数组表示

一个标准的一元n阶多项式的各项若按降幂排列，可以表示为如下形式：

$$A_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \quad (a_n \neq 0)$$

一个标准的6阶多项式

例如

$A(x)=2x^6-4x^5+10x^4-7x^3+6x^2-4x+1$

$B(x)=2x^6-4x^5+6x^2+1$

$C(x)=x^{2000}-5$

非标准多项式

定义一个一维数组A[0..n+1]来存储多项式，其中，
A[0] 用来存放多项式的阶数n;
A[1]~A[n+1] 依次用来存放多项式的n+1项的系数。

对于多项式 $A(x)=10x^6-8x^5+3x^2-1$, 有

A[0..7]

6	10	−8	0	0	3	0	−1
0	1	2	3	4	5	6	7

对于多项式 $B(x)=x^{2000}-5$, 有

B[0..2001]

2000	1	0	0	… …	0	−5

1999项为0

定义一个一维数组A[0..2m]来存储多项式，其中，

A[0] 存放系数非零项的总项数m；

A[1]~A[2m] 依次存放系数非零项各项的系数与指数偶对（一共m个这样的偶对）。

对于多项式 $A(x)= 10x^6 - 8x^5 + 3x^2 - 1$, 有

A[0..8]

4	10	6	−8	5	3	2	−1	0

第1项　第2项　第3项　第4项

对于多项式 $C(x)=x^{2000} - 5$, 有

C[0..4]

2	1	2000	−5	0

第1项　第2项

对于那些**标准的**或者**基本标准的**多项式，宜采用方法**1**。

缺项很少

例

$A(x)= 10x^6- 8x^5+2x^4-7x^3+3x^2+x-1$

$B(x)= 10x^6-2x^4-7x^3+3x^2-1$

对于那些**缺项很多的**多项式，宜采用方法**2**。

例

$C(x)= x^{2000}-5$

（二）n 阶“魔方”（n为任意奇数）*

游戏

将1~9不重复地填在3行3列的9个方格中，分别使得每一行、每一列、两个对角线上的元素之和都等于15。

3阶魔方

6	1	8
7	5	3
2	9	4

一个3 阶魔方

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

一个5 阶魔方

……

以n=5阶魔方为例

分别以i和j表示行与列的位置

(i, j)	(i, j)	(i, j)	(i, j)	(i, j)	
(i, j)	15	8	1	24	17	0
(i, j)	16	14	7	5	23	1
(i, j)	22	20	13	6	4	
(i, j)	3	21	19	12	10	
	9	2	25	18	11	4

1 ~ 25

初始 i=0, j=⌊n/2⌋

i--; j--;

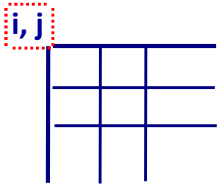
i+=2; j++;

i	j
0	2
4	1
3	0
4	3
3	4
4	3
3	4
2	3
	2

i	j
1	1
0	0
1	0
4	4
2	2
1	1
0	1
4	0
3	4
2	3
1	2
0	4
4	3
3	2
2	

一个5阶魔方

规律



1. 将用做“魔方”的二维数组的所有元素清0；

2. 第一个数填在第一行居中的位置上(i=0，j=n/2)；

3. 以后每填一个数后，将位置移到当前位置(i,j)的左上角，即做动作i=i−1，j=j−1；

4. 根据不同情况对位置进行修正：
   （1）若位置(i,j)上已经填数，或者i,j同时小于0，将位置修改为i=i+2，j=j+1；
   （2）若i小于0，但j不小于0，修改i为n−1；
   （3）若j小于0，但i不小于0，修改j为n−1。

重复(循环)


```
void  magic(int a[], int n )
{
    int  i, j, num;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            a[i][j]=0;              /* 魔方清0 */
    i=0;
    j=n/2;                          /* 确定i与j的初始位置 */
    for(num=1; num<=n*n; num++){
        if(i<0 && j<0 || a[i][j]!=0){
            i+=2;
            j++;
        }
        a[i--][j--]=num;        /* 填数，并且左上移一个位置 */
        if(i<0 && j>=0)
            i=n-1;              /* 修正i的位置 */
        if(j<0 && i>=0)
            j=n-1;              /* 修正j的位置 */
    }
}
```

延伸阅读*

建议有兴趣的同学自已看看有关基于稀疏矩阵的矩阵加法和乘法等运算。

本节内容小结

数组

数组的基本概念

- 数组的定义
- 数组的基本操作

本章内容不作为重点要求

数组的存储方法

- 一维数组的存储
- 二维数组的存储

  行序为主序、列序为主序方式(地址计算公式)

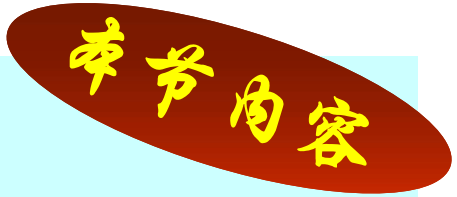
特殊矩阵的压缩存储

- 对称矩阵、(三)对角矩阵的压缩存储
- 稀疏矩阵的压缩存储

  三元组表表示、十字链表表示

数组的应用举例

广义表*



广义表的基本概念

广义表的存储结构

多元多项式的广义表表示

广义表的概念

一. 广义表的定义

一个长度为$n \geq 0$ 的广义表是一个数据结构

$LS = (a_1, a_2, \ldots\ldots, a_{n-1}, a_n)$

其中，LS为广义表的名字，a_i为表中元素；a_i可以是原子元素，也可以是一个子表。n为表的长度，长度为0的表称为空表。

若a_i为不可再分割的具体信息，则称a_i为**原子元素**；若a_i为一个子表，则称a_i为**表元素**。这里，用小写字母表示原子元素，用大写字母表示表元素。

$(a_1, a_2, a_3, \ldots\ldots, a_{n-1}, a_n)$

二、广义表的例子

A=( ) ——长度为0的空表。

B=(a) ——长度为1,且只有一个原子元素的广义表。

C=(a, (b,c)) ——长度为2的广义表。

D=(A, B, c)
E=(A, B, C) } ——长度为3的广义表。

F=(a, F) ——长度为2的递归的广义表。

……

F=(a, F)=(a, (a, (a, ...)))

(1)广义表是多层结构的
(2)广义表可为其他广义表所共享
(3)广义表可以是嵌套的

广义表的深度
----括号嵌套的重数。

二、广义表的例子

广义表的深度 ----括号嵌套的重数

A1 = (A) = (())是空表吗？长度是 1 深度是 2

A = ((a))　　B = (a,(b,c,d),e,())

C = (x,((y),B,A))

问：C的长度　　C的深度

广义表的存储结构

顺序存储？还是链式存储？

$$LS = (a_1, a_2, a_3, \ldots\ldots, a_{n-1}, a_n)$$

广义表一般采用链式存储结构，链结点的构造可以为

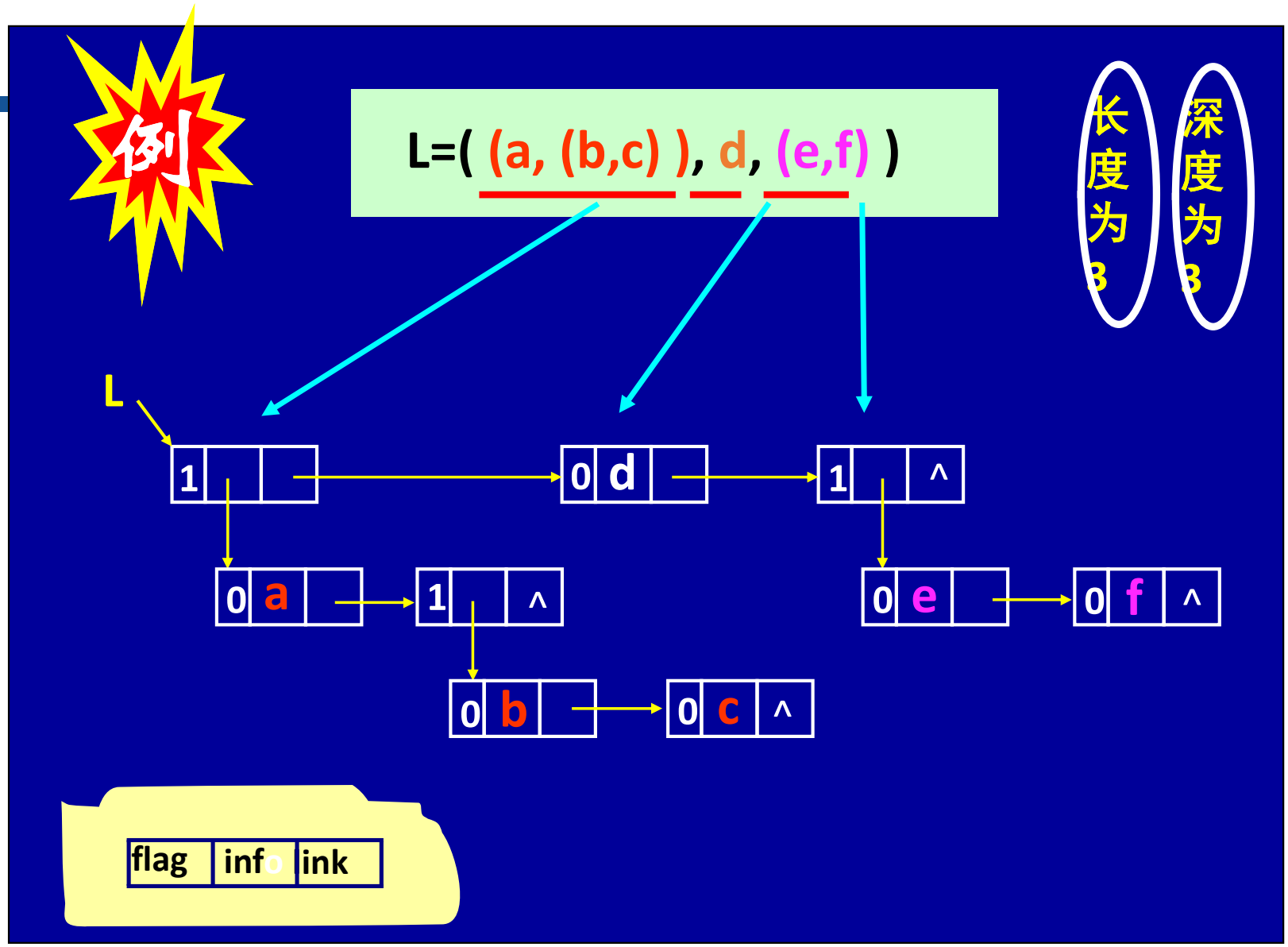
其中，**flag**为标志位，令

$$flag = \begin{cases} 1 & \text{表示本结点为表结点} \\ 0 & \text{表示本结点为原子结点} \end{cases}$$

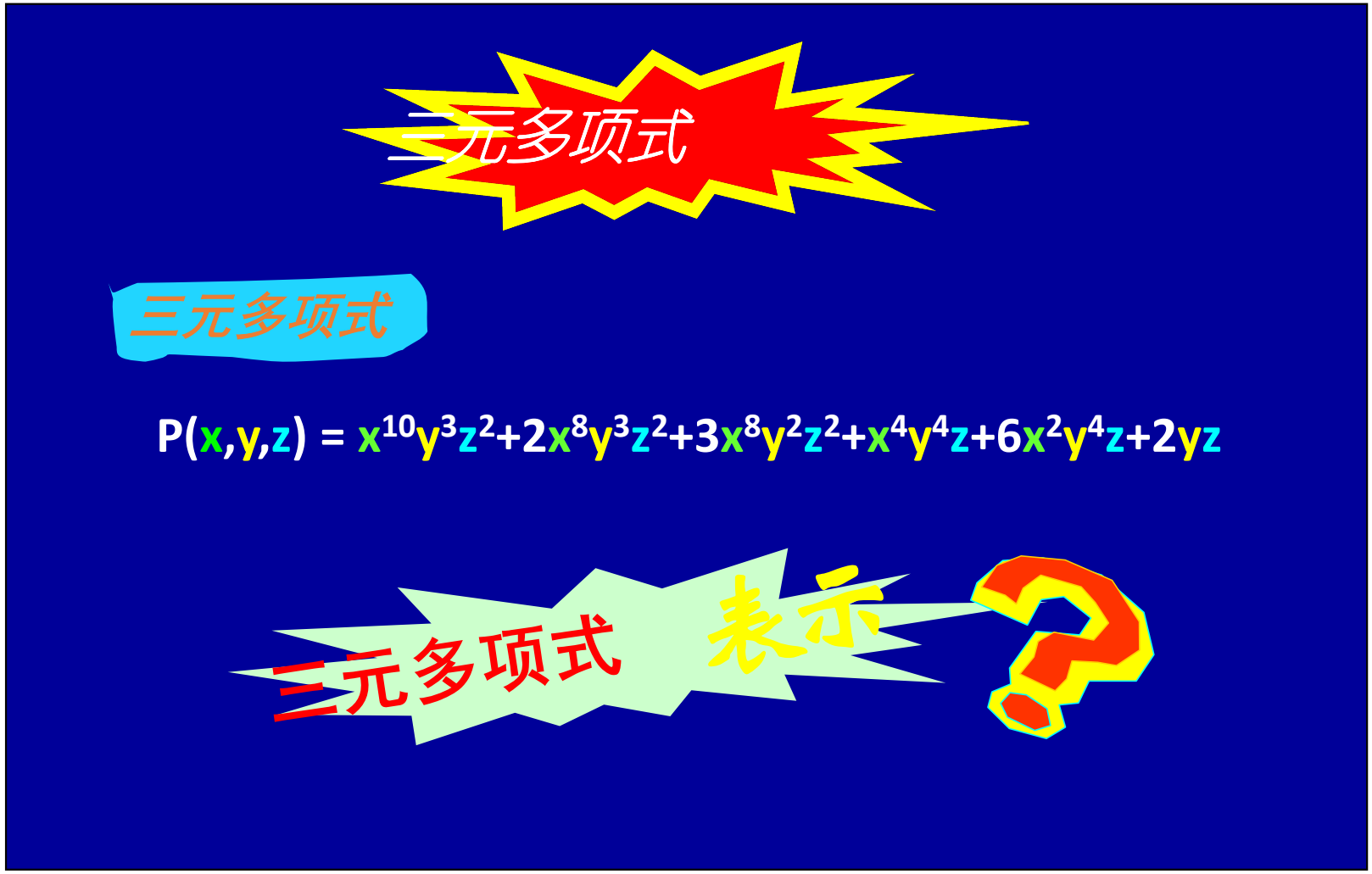
当**flag=0**时，**info**域存放相应原子元素的信息；
当**flag=1**时，**info**域存放子表第一个元素对应的链结点的地址；
link域存放元素同一层的下一个元素所在链结点的地址，当本元素为所在层的最后一个元素时，**link**域为**NULL**。

类型定义

```
typedef  struct node{
     int  flag;
     union{
          DataType data;
          struct node *pointer;
     } info;
     struct node *link;
}BSNode, *BSLinkList;
```


例
L=((a, (b,c)), d, (e,f))
长度为3
深度为3
L
1
0 d
1 ^
0 a
1 ^
0 e
0 f ^
0 b
0 c ^
flag info link

5.3 多元多项式的广义表表示



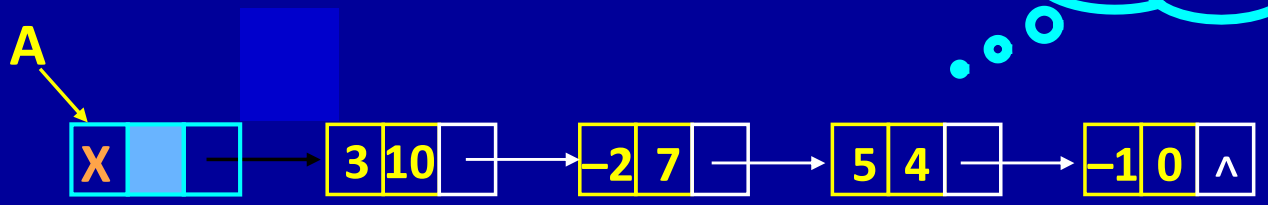
回忆

一元多项式

$$A(x) = 3x^{10} - 2x^{7} + 5x^{4} - 1$$

一个链结点：

一元多项式的链表表示



三元多项式

方法1

coef	expx	expy	expz	link

3个指数域

$3x^8y^2z^2$

3	8	2	2	

$-5x^4z^7$

-5	4	0	7	

$6x^7$

6	7	0	0	

缺点

- 链结点中域的个数取决于表达式中变量的个数；
- 给存储管理和操作带来困难。

方法2

链结点的构造设计为

其中，**coef** 表示多项式的某一项的**系数**，
exp 表示多项式的某一项的指数，
link 为链接多项式中同一层各链结点的指针。

三元多项式

例

$$P(x,y,z) = x^{10}y^3z^2+2x^8y^3z^2+3x^8y^2z^2+x^4y^4z+6x^2y^4z+2yz$$

$$= ((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^2)y^4+2y)z$$

$$P(z) = Az^2+Bz$$

其中：

$$A(x,y) = (x^{10}+2x^8)y^3+3x^8y^2$$

$$A(y) = Cy^3+Dy^2$$

$$C(x) = x^{10}+2x^8$$

$$D(x) = 3x^8$$

$$B(x,y) = (x^4+6x^2)y^4+2y$$

$$B(y) = Ey^4+Fy$$

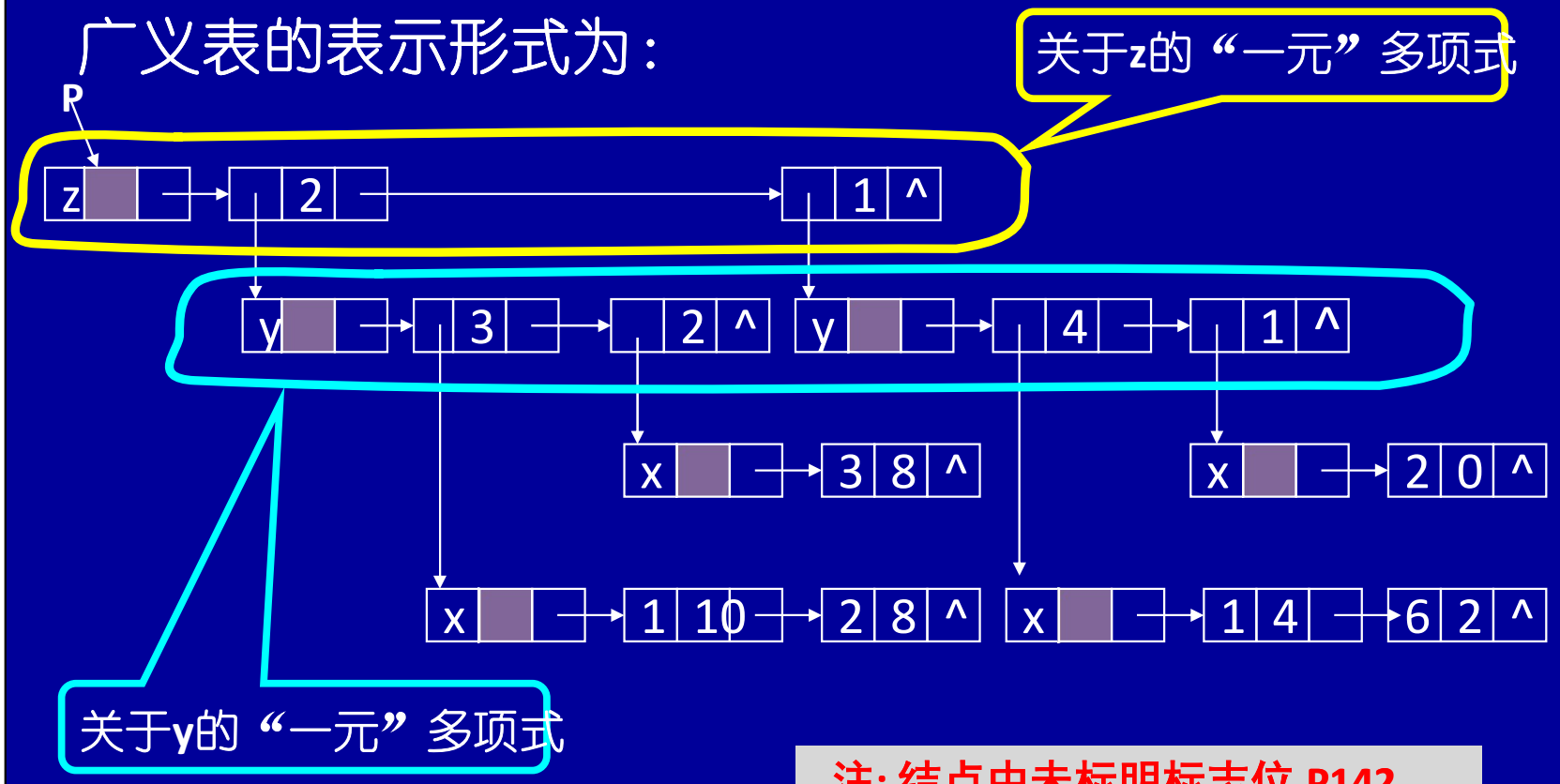
$$E(x) = x^4+6x^2$$

$$F(x) = 2x0$$

一元多项式

$$P(x,y,z)=x^{10}y^3z^2+2x^8y^3z^2+3x^8y^2z^2+x^4y^4z+6x^2y^4z+2yz$$
$$=((x^{10}+2x^8)y^3+3x^8y^2)z^2+((x^4+6x^2)y^4+2y)z$$

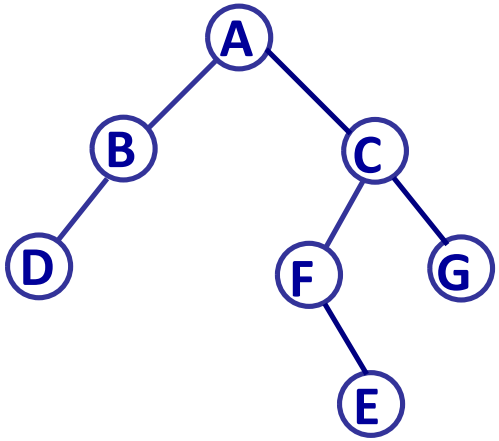
广义表的表示形式为：

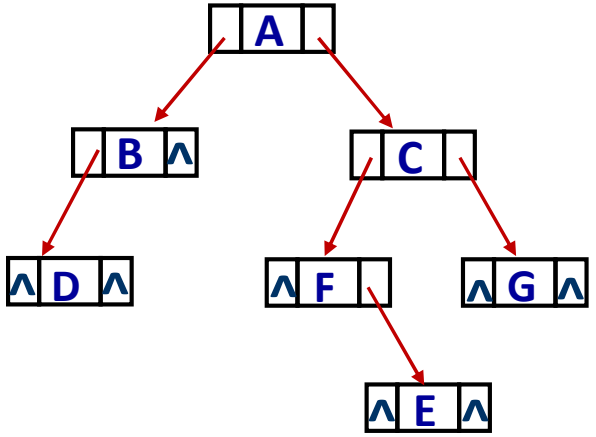


注: 结点中未标明标志位,P142

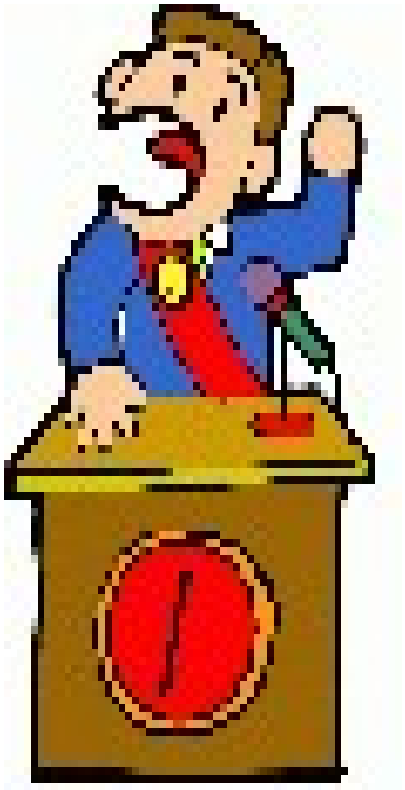
后续章节中的二叉树的广义表表示法

A(B(D), C(F(, E), G))





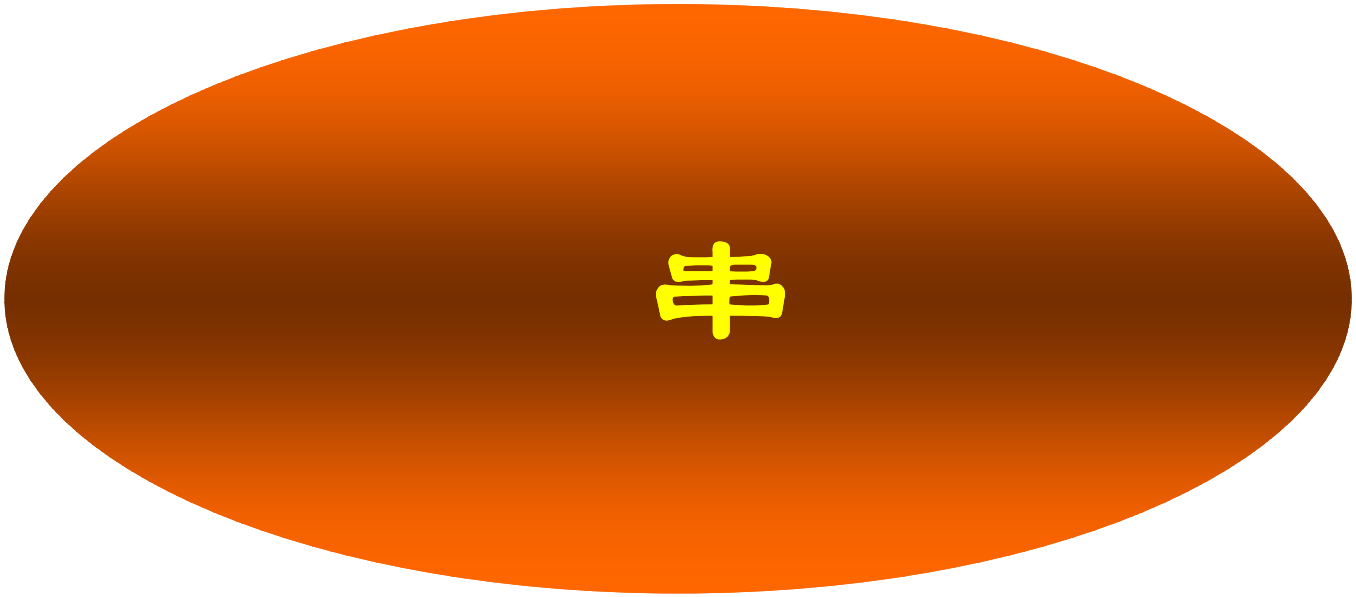
本节内容小结



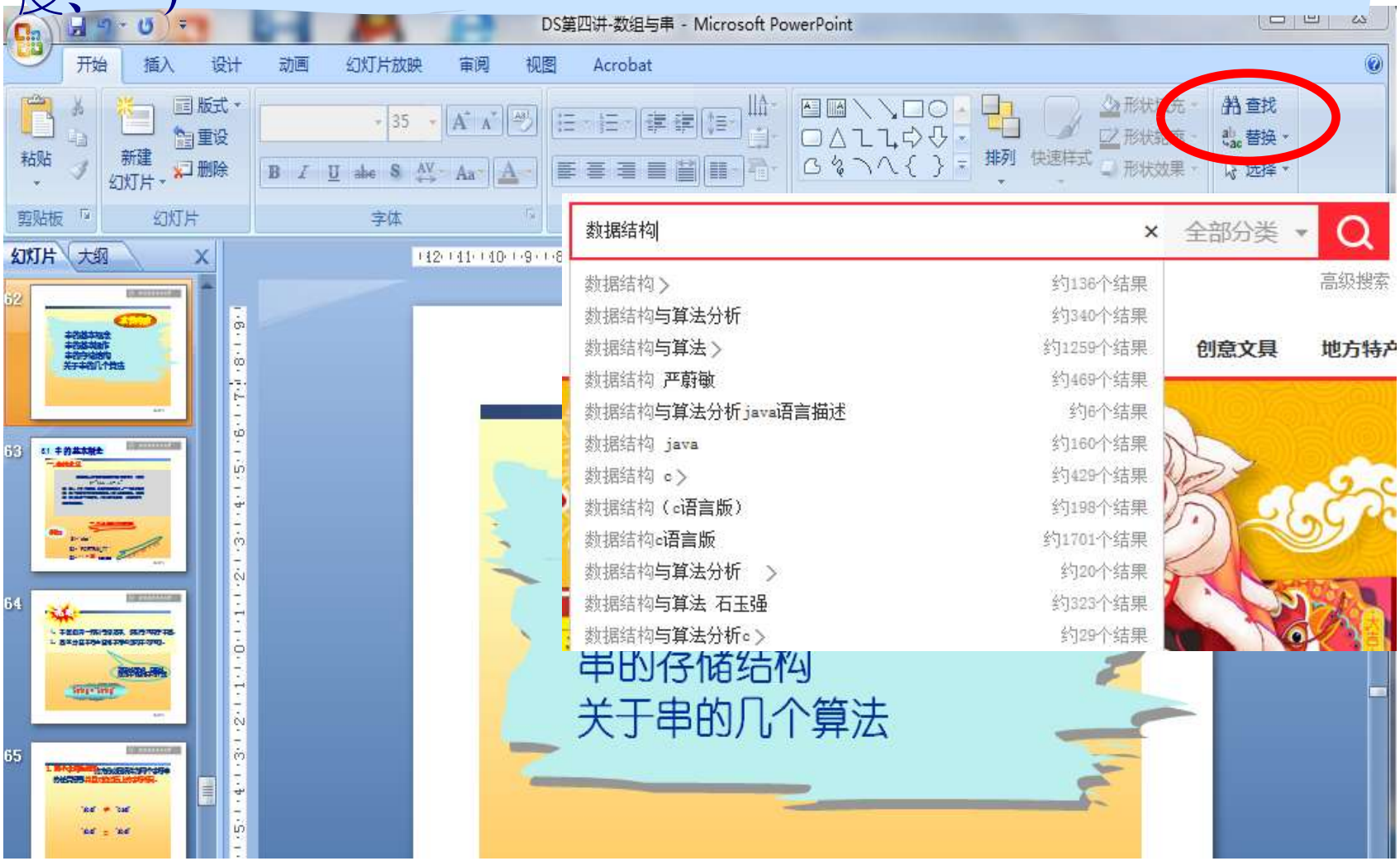
广义表

- 广义表的基本概念
  - 广义表的定义
  - 基本的名词概念
- 广义表的存储方法
  - 链表表示法
- 三元多项式的广义表表示

本章内容
不作为重点


串

串的应用非常广，几乎所有的办公软件都有关于串的操作（如OFFIC软件），也是所有搜索引擎的基本功能（如Google、百度、…）



问题4.1：实现一个简化的Linux命令grep*

Linux系统中**grep**命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。grep全称是Global Regular Expression Print。其格式为：

% grep [-acinv] [--color=auto] '搜寻字符串' filename

实现一个简化版的grep命令，其格式为：

 # grep [-i] [-n] [-v] '搜寻字符串' filename

选项与参数：

-i ：忽略大小写的不同，所以大小写视为相同

-n ：顺便输出行号

-v ：反向选择，亦即显示出没有'搜寻字符串'内容的那一行！

不支持正则表达式搜索

例如：

将文件/etc/passwd中有root出现的行取出来，同时显示这些行在/etc/passwd中的行号

```
# grep -n root /etc/passwd
```

输出：

```
1:root:x:0:0:root:/root:/bin/bash
30:operator:x:11:0:operator:/root:/sbin/nologin
```

一个**UNIX/Linux**下非常实用的工具

本节内容

串的基本概念*
串的基本操作*
串的存储结构*
关于串的几个算法*
串的模式匹配(Pattern Matching)

串的基本概念

一. 串的定义

串是由n≥0个字符组成的有限序列，记为

$S = ' a_1 a_2 a_3 \ldots a_{n-1} a_n '$

其中，S表示串名（也称串变量），一对引号括起来的字符序列称为串值，a_i可以是字母、数字或其他允许的字符。n为串的长度，长度为0的串称为空串。

例如

一个长度为3的字符串

S1= ′abc′

S2= ′FORTRAN_77′

S3= ′ ′ = Φ (空串)

一个长度为10的字符串

说明

1. 串值须用一对引号括起来，但引号不属于串值。
2. 要区分空串与由空格字符组成的串的不同。

前者长度为0，后者长度为串中空格字符个数

String = ′String′

3.两个字符串相等 的充分必要条件为两个字符串的长度相等，并且对应位置上的字符相同。

′abcd′ ≠ ′bacd′

′abcd′ = ′abcd′

串的基本操作

C函数

1.给串变量赋值 ASSIGN(S1,S2) strcpy(S1,S2)

2.判断两个串是否相等 EQUAL(S1,S2) strcmp(S1,S2)

3.两个字符串连接 CONCAT(S1,S2) strcat(S1,S2)

4.求字符串的长度 LEN(S) strlen(S)

5.求子串 SUBSTR(S,i,k) c++:substr(S,start,len)

6.求子串在主串中的位置 INDEX(S1,S2) strstr(S1,S2)

7.串的替换 REPLACE(S,S1,S2)

8.串的复制 COPY(S1,S2) strcpy(S2,S1)

9.串的插入 INSERTS(S1,i,S2)

10.串的删除 DELETES(S,i,k)

……

模式匹配

串的存储结构

一. 串的顺序存储结构

1.非紧缩格式（设每个字有4个字节）

例如: S = 'DATA STRUCTURE'

D			
A			
T			
A			
S			
T			
R			
U			
C			
T			
U			
R			
E			
@			

2.紧缩格式

'\0'

D	A	T	A
	S	T	R
U	C	T	U
R	E	@	

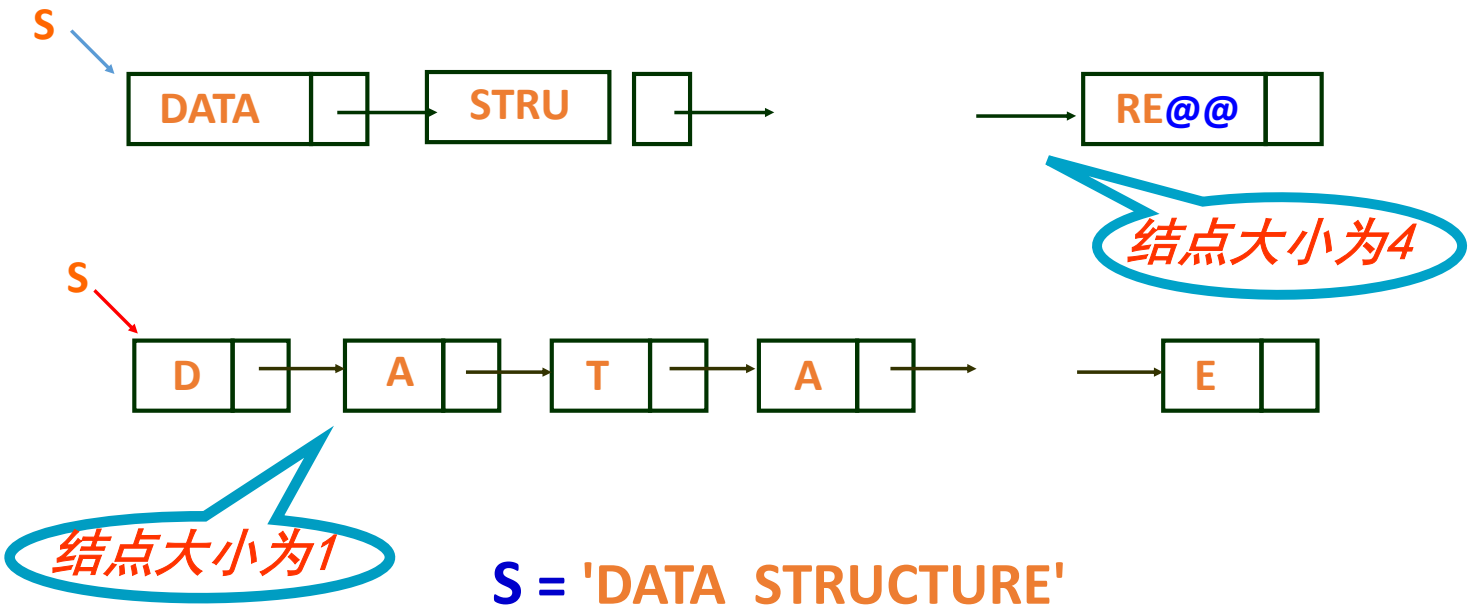
3. 单字节方式

D	A	T	A		S	T	R	U	C	T	U	R	E			@

二. 串的链式存储结构

说明

所谓**链结点大小**是指每个链结点的数据域中存放字符的个数。



S = 'DATA STRUCTURE'

三. C语言中串的存储与表示

说明

1. 在C中，串是按单字节方式存储，串中每个字符的类型为char，串的结束标志为'\0'。
2. 在C中，字符串常量是以双引号括起来的，如"C Language"，其内存中存放形式如下：

'C'	' '	'L'	'a'	'n'	'g'	'u'	'a'	'g'	'e'	'\0'

3. 在C中，串可通过数组和指针来存储和访问；如：

```
char array[32], *str;
strcpy(array, "Data Structure");
strptr = (char *)malloc(strlen(array)+1);
strcpy(strprt, array);
```

4. 在C中，存储串的空间必须比串中字符个数多1（用以存放串结束标志）

串的几个算法

一. 判断两个字符串是否相等

功能

两个字符串分别存放于数组S1与S2中，判断两个串是否相等，若相等返回信息1，否则，返回信息0。

S1 = “a_1 a_2 a_3 a_4 a_5 … a_n”

S2 = “b_1 b_2 b_3 b_4 b_5 … b_m”

算法

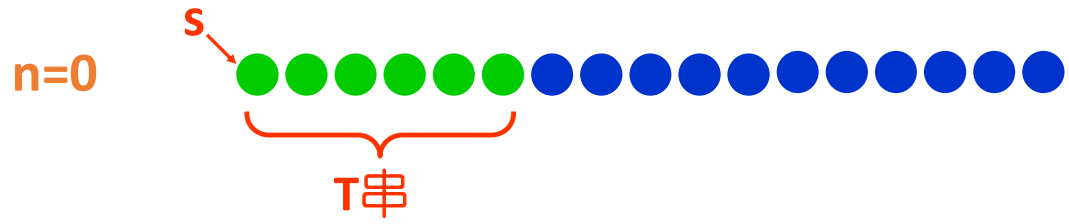
```
int  equal(char s1[ ],char s2[ ])
{    int i=0;
     for(i=0; s1[i]!=´\0´  &&  s2[i]!=´\0´; i++)
          if(s1[i]!=s2[i])
               return  0;       /* 两个串不相等 */
     if(s1[i]==´\0´  && s2[i]==´\0´)
          return 1;             /* 两个串相等 */
     return 0;
}
```

二. 串的插入

功能：在字符串S的第n个字符后面插入字符串T。

前提：字符串S与T分别采用数组形式存储。

约定：1. 当n=0时，将T 插在S 的最前面。
2. 结果串由S 指出。



几个名词概念

1. 子串：串中若干个连续的字符组成的子序列。

例如：　S= ´Beijing&Shanghai´
　　T= ´jing´

T是S的子串，
S是T的主串

2. 主串：包含子串的串。

3. 位置： **(1)** 单个字符在主串中的位置被定义为该字符在串中的序号。

(2) 子串在主串中的位置 被定义为主串中首次出现的该子串的第一个字符在主串中的位置。

例如：　S= ´Beijing&Nanjing&Shanghai´
　　T= ´jing´

位置为4

算法

```
int  insert(char s[ ],char t[ ], int n)
{    int i=0;
     char *p;
     p = (char *)malloc(strlen(s)+1);
     for(i=0; s[i+n] != '\0'; i++)
         p[i] = s[i+n];
     for(i=n,j=0; t[j] != '\0'; i++,j++)
         s[i] = t[j];
     for(j=0; p[j] != '\0'; i++,j++)
         s[i] = p[j];
     free(p);
     return n;
}
```

```
int  insert(char s[ ] , char t[ ],
int n)
{
    strcat(t, s+n);
    s[n] = '\0';
    strcat(s, t);
    return n;
}
```

三. 串的模式匹配(Pattern Matching)

功能：字符串的定位。给定一个主字符串S和一个子串T(又称模式串)，长度分别为n和m。在主串S中，从起始位置开始查找，若在主串S中找到一个与子串T相等的子串，则返回T的第一个字符在主串中的位置序号。

例如： S=´Beijing&Nanjing&Shanghai´
T=´jing´

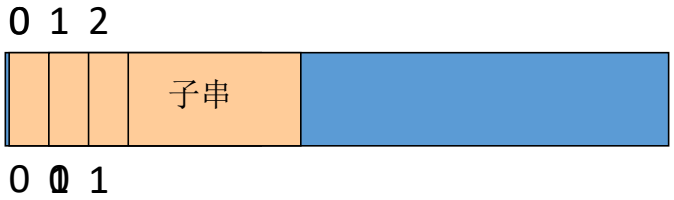
返回4

串的简单模式匹配算法Brute-Force（布鲁特-福斯，又称朴素的模式匹配算法）算法：

- 将主串S的第一个字符和模式串T的第1个字符比较
  若相等，继续逐个比较后续字符；
  若不等，从主串S的下一字符起，重新与T第一个字符比较。
- 直到主串S的一个连续子串字符序列与模式T相等。返回值为S中与T匹配的子序列第一个字符的序号，即匹配成功。否则，匹配失败，返回值 -1。

算法设计

- 设int index(char s[ ], char t[ ])函数用来在字符串s中查找字符串t。若找到则返回t在s中出现的位置，否则返回-1。其主要查找算法如下：



主串

遍历主字符串中每个字符

```
在字符串s中查找字符串t ：
for(i=0; s[i] != ‘\0’; i++)
    for(j=i,k=0; t[k] != ‘\0; j++,k++)
        s[j]和t[k]进行比较
```

依次与子串中每个字符比较。j为s中每次开始比较的位置。

主要算法分析

一个经典的朴素字符串查找算法（Brute-Force）

算法

```
a b b a b a    a b b a b a  …  a b b a b a
a b a            a b a        …      a b a
```

```
int index(char s[ ], char t[ ])
{
    int i, j, k;
    for(i =0; s[i] != '\0'; i++){
        for(j=i,k=0;t[k]!='\0'&&s[j]==t[k]; j++,k++)
            ;
        if(t[k] == '\0')
            return ( i);
    }
    return ( -1);
}
```

O(n*m)

另一种实现方式

```
int index(char S[ ], char T[ ])
{
    int i = 0,  j=0;

    while ( S[i]!='\0'  &&  T[j]!='\0') {
        if (S [i] == T[j] ) {
            i++;
            j++
        }
        else {
            i = i-j +1;
            j = 0;
        }
    }
    if ( T[j] == '\0')
        return  i-j;
    else
        return -1;
}
```

最坏情况下，主串前面n-m个位置都部分匹配到子串的最后一位，即这n-m位比较了m次，最后m位也各比较了一次，加上m，所以总次数为：(n-m)*m+m ＝(n-m+1)*m

算法改进

```
int index(char s[ ], char t[ ])
{
        int i, j, k,n,m;
         n = strlen(s);
        m = strlen(t);
        for(i =0; n-i >= m; i++){
                for(j=i,k=0;t[k]!='\0'&&s[j]==t[k];  j++,k++)
                        ;
                if(t[k] == '\0')
                        return ( i);
        }
        return ( -1);
}
```

当s中剩余字符数小于t中字符数时，停止查找

问题4.1：实现一个简化的Linux命令grep*

Linux系统中**grep**命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。grep全称是Global Regular Expression Print。其格式为：

% grep [-acinv] [--color=auto] '搜寻字符串' filename

实现一个简化版的grep命令，其格式为：

 # grep [-i] [-n] [-v] '搜寻字符串' filename

选项与参数：

-i ：忽略大小写的不同，所以大小写视为相同

-n ：顺便输出行号

-v ：反向选择，亦即显示出没有'搜寻字符串'内容的那一行！

不支持正则表达式搜索

例如：

将文件/etc/passwd中有root出现的行取出来，同时显示这些行在/etc/passwd中的行号

```
# grep -n root /etc/passwd
```

输出：

```
1:root:x:0:0:root:/root:/bin/bash
30:operator:x:11:0:operator:/root:/sbin/nologin
```

问题4.1：算法分析

该问题的算法关键是在一个字符串（行）中查找另一个字符串。设：

```
int index(char s[], char t[], int status)
```

返回0表示没有找到；返回非0正整数时，表示模式匹配成功次数。当status为0时表示严格匹配；为1时表示大小写无关匹配。

问题4.1：代码实现

```
int index(char s[ ], char t[ ], int status ) //status为0：大小写敏感；status为1大小写无关
{
      int i, j, k,n,m,count=0;
       n = strlen(s);
      m = strlen(t);
      for(i =0; n-i >= m; i++){
            for(j=i,k=0;t[k]!='\0'&&issame(s[j], t[k], status); j++,k++)
                  ;
            if(t[k] == '\0')
                  count++; //找到后，次数加1，继续查找
      }
      return  count;
}
int  issame(char a, char b, int s)
{
      return s?(tolower(a) == tolower(b)) : (a == b);
}
```

```
int  main(int argc, char *argv[])
{
   char  line[MAXLEN], *pattern, *filename;
   int ai=0, an=0, av=0,lnum = 0, pos; //对应参数-i,-n,-v
   FILE  *fp;
   if( argc < 3 || argc > 6 ) error("usage:  grep  [-i] [-n] [-v] pattern filename\n");
   else {
      pattern = argv[argc-2]; filename = argv[argc-1];
      for( ; argc>3 ; argc--) {
         if(argv[argc-3][0] != '-') error("usage:grep  [-i] [-n] [-v] pattern filename\n");
         else
            switch(argv[argc-3][1]) {
               case  'i': ai = 1; break;
               case  'n': an = 1; break;
               case 'v': av = 1; break;
               default: error("usage:  grep [-i] [-n] [-v] pattern filename\n");
            }
      }
   }
   if((fp = fopen(filename, "r")) == NULL)   error("Can open file\n");
   else {
      while(fgets(line, MAXLEN-1, fp) !=NULL){
         lnum++;
         pos =  index(line, pattern, ai);
         print(line,lnum,pos,an,av);
      }
   }
   return 0;
}
```

问题4.1：思考

参考问题4.1设计并实现一个程序用来在一个文件中查找某个字符串并替换成另一个串。命令格式为：

```
replace [-i] str1 str2 filename
```

要求：

- 支持大小写无关
- 支持简单的模式查找，如？

模式匹配的KMP(Knuth-Morris-Pratt)算法

功能：字符串的匹配。给定一个主字符串S和一个子串T(又称模式串)，长度分别为n和m。若在主串S中找到一个与子串T相等的子串，则返回T的第一个字符在主串中的位置序号。

O(n*m) ➡ **O(n+m)**

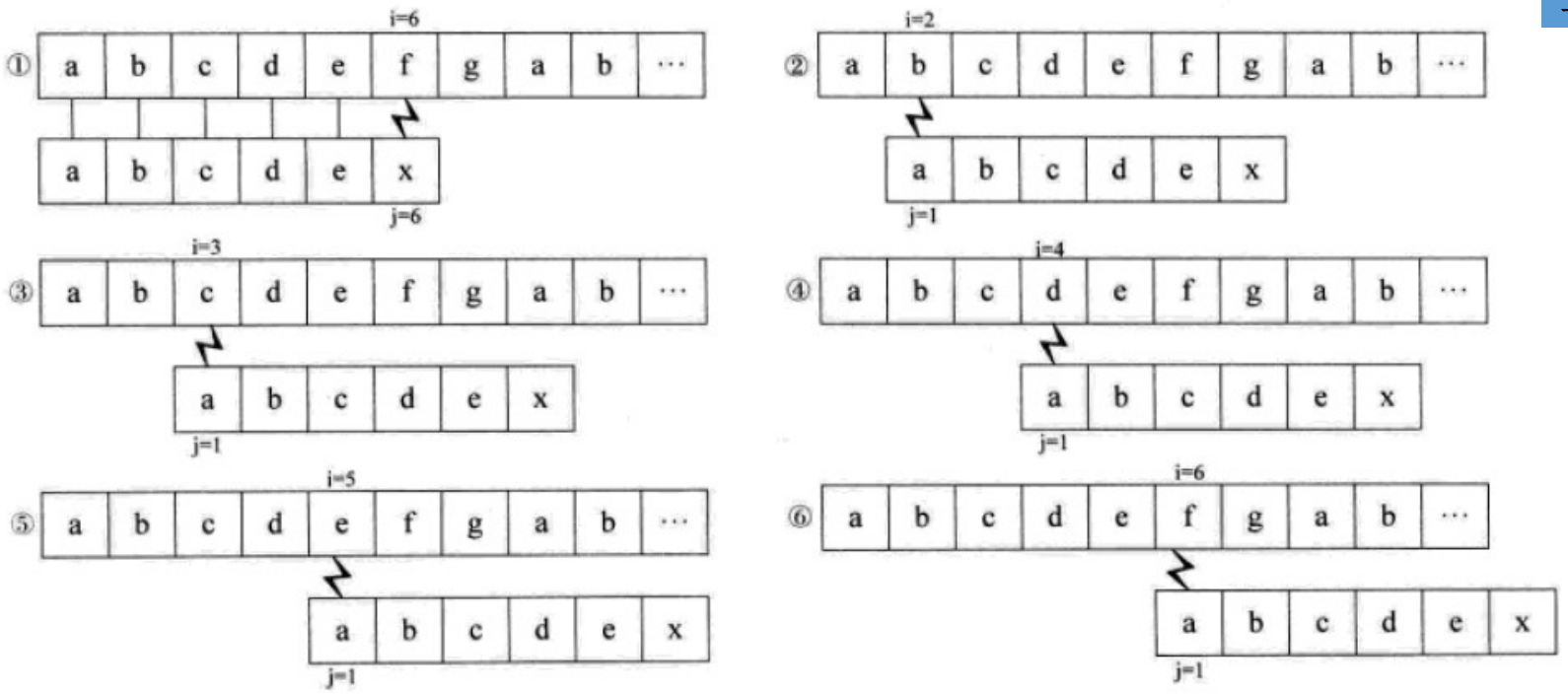
朴素(Brute-Force)字符串匹配算法C实现

```
int index(char S[ ], char T[ ])
{
    int i = 0,  j=0;

    while ( S[i]!='\0'  &&  T[j]!='\0') {
       if (S [i] == T[j] ) {
          i++;
          j++
       }
       else {
          i = i-j +1;          //i回退到上次匹配起始位置的下一个位置
          j = 0;               //j回退到起始位置重新开始查找
       }
    }
    if ( T[j] == '\0')         //匹配成功，返回匹配位置
       return  i-j;
   else
       return -1;
}
```

朴素字符串匹配算法存在的问题

在串“abcdefgab…”中查找子串“abcdex”



朴素字符串匹配算法存在的问题：

① 当前匹配在找到不匹配的字符后，要将主串中下一次匹配开始位置移动一个位置（即要回溯），而不管当前匹配情况，如上图。

KMP算法核心思想

源串称为主串，定义为S，当前匹配位置为i；目标串称为子串，定义为T，当前匹配位置为j。当前匹配在找到不匹配的字符后，重新开始匹配时：

1. 主串当前位置i不回溯，即不重置为上次匹配开始位置的一下位置；
2. 子串当前位置j视情况回溯至起始串位置（0），或子串中某一位置。

如何计算子串回溯的位置？

KMP算法核心思想 – 计算子串回溯位置

根据子串T当前匹配的规律：　“$T_0 \ldots T_{k-1}$”=“$T_{j-k} \ldots T_{j-1}$”
由当前失配位置j(已知)，可以归纳计算下次匹配起点 k的表达式。

令k = next[j]（函数next用子串当前位置j来计算下次开始匹配位置k，k与j 显然具有函数关系），则

$$next[j] = \begin{cases} -1 & \text{当 } j = 0 \text{ 时} \\ \max\{k \mid 0<k<j \text{ 且 } `T_0 \ldots T_{k-1}' = `T_{j-k} \ldots T_{j-1}'\} & \\ 0 & \text{其他情况} \end{cases}$$

取当前位置j前序串首尾最大的相同子串长度

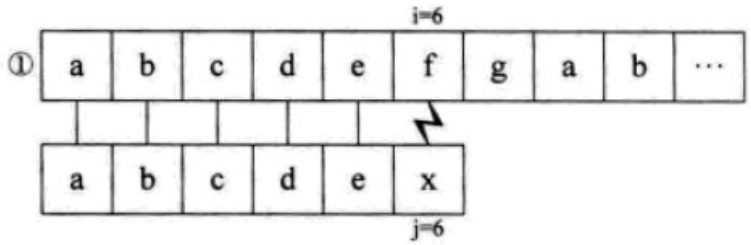
注意：

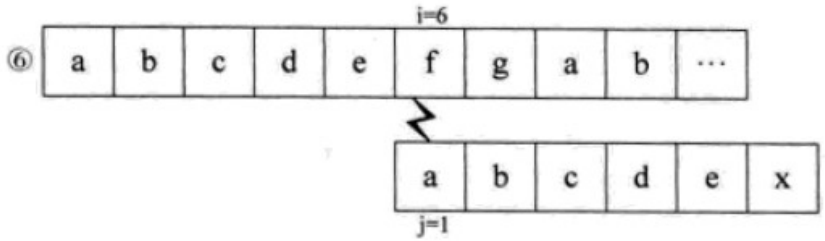
（1）k值仅取决于子串本身而与相匹配的主串无关。

（2）k值为子串从头向后及从j向前的两部分子串最大相同子串的长度。

（3）这里的两部分子串可以有部分重叠的字符，但不可以全部重叠，即k最大为j-1。

KMP算法核心思想

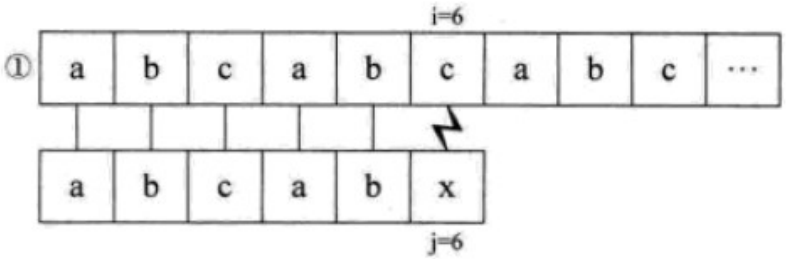


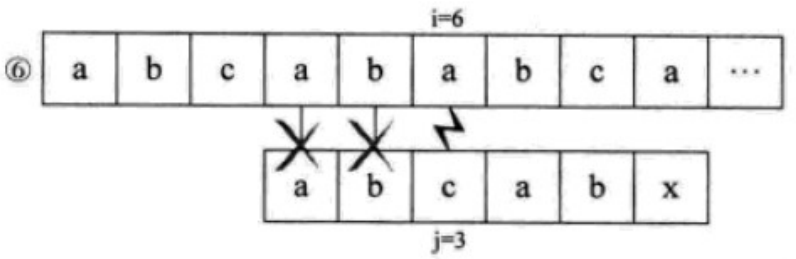


j	0 1 2 3 4 5 6 7 8
子串T	a b c d e f g h i
next[j]	-1 0 0 0 0 0 0 0 0

T串中无重复子串

KMP算法核心思想





j	0 1 2 3 4 5 6 7 8
子串T	a b c a b c a b c
next[j]	-1 0 0 0 1 2 3 4 5

T串中有重复子串

j	0 1 2 3 4 5 6 7 8
子串T	a a a a a a a a i
next[j]	-1 0 1 2 3 4 5 6 7

T串中有重复子串

KMP算法C代码实现


```
int KMPindex(char S[ ], char T[ ])
{
    int i = 0,  j=0, *next;

    next = (int *)malloc(sizeof(int)*(strlen(T)+1));
    getnext(T, next);
    while ( S[i]!='\0'  &&  T[j]!='\0') {
       if (S [i] == T[j] ) {
          i++;
          j++ ;
       }
       else
          (j == 0) ? i++ :( j = next[j]);              //j回退到相应位置开始匹配，i值不变
    }
    free(next);
    if ( T[j] == '\0') //匹配成功，返回匹配位置
       return  i-j;
   else
       return -1;
}
```

KMP算法：由于下标i无须回溯，比较次数仅为n,即使加上计算next[j]时所用的比较次数m，比较总次数也仅为**n+m=O(n+m)**，大大快于朴素的Brute-Force算法。

KMP算法C代码实现-计算next

```
void getnext(char T[], int next[])
{
    int i=0, j=-1;
    next[0] = -1;
    while(T[i]!= '\0' ){
      if(j==-1 || T[i]==T[j]){ //i为后缀位置；j为前缀位置
        i++;
        j++;
        next[i]=j;
      }
      else
        j = next[j]; //若字符不同，则j值回溯
   }
}
```

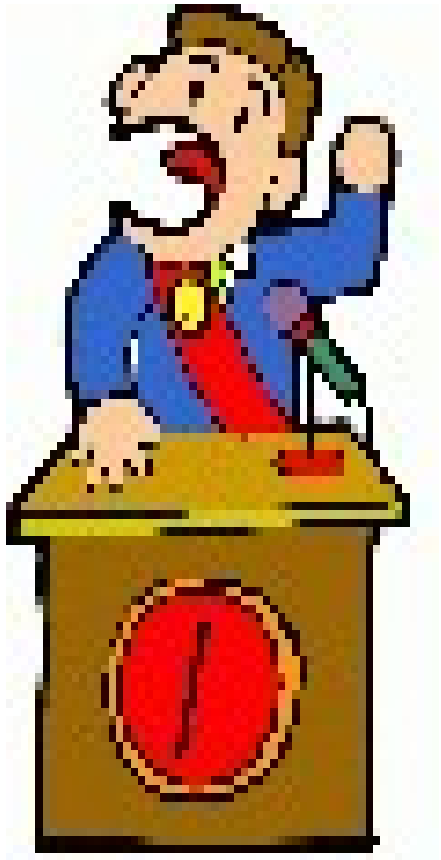
KMP算法C代码实现 – 示例

```
int main( )
{
   char filename[64], t[81], line[MAXLINE],*s;
   FILE *fp;
   int n,pos=0;
   scanf("%s %s", filename,t);
   if((fp = fopen(filename, "r")) == NULL){
      printf("Can't open file %s!\n", filename);
      return 1;
   }
   while(fgets(line, MAXLINE-1, fp) != NULL){
      s = line;
      while((n=KMPindex(s, t)) >= 0){
         pos += n;
         printf("%d\n", pos);
         s = s+(n+1);
         pos++;
      }
   }
   return 0;
}
```

关于字符串匹配还有.....*

延伸阅读*：

1. 正则表达式匹配
2. 模糊匹配
3. 最长共有字符串匹配



本章内容小结

字符串

1. 字符串的基本概念

- 字符串的定义
- 基本的名词概念
  - 子串、主串、位置、两个串相等

2. 字符串的存储结构

- 顺序存储结构
  - 紧缩格式、非紧缩格式、单字节格式
- 链式存储结构
  - 关于链结点大小

3. 关于字符串的几个基本算法

- 判断两串相等
- 串的插入
- 模式匹配

结束！