第四章 数组、串与广义表

内容提要：

本章学习另外三种特殊的线性表：

- 数组：特殊在元素受多个线性关系约束
  - 操作—该数据结构上定义的操作很少
  - 在高级语言中都已物理实现为数组数据类型。
- 串：特殊在数据元素是字符
  - 操作—操作对象一般为一组元素（子串）
  - 在有些高级语言中已经物理实现为字符串类型
- 广义表：特殊在数据元素又可以是线性表（略）
  - 表嵌套

4.1 数组

4.1.1 数组ADT的定义

1. 数组逻辑结构

[数组 Array]：简单说是数据类型相同的一组数据元素的有序集合。元素之间可以具有多个线性关系。元素在集合中的位置是由元素在每一个线性关系上的位置共同决定的。

特点：*参与多个线性关系*，在每个关系上都有前驱、后继！

[下标 Index]：元素在某个关系上处的位置。

[维数 Dimensions]：元素参与线性关系的个数。

4.1 数组

4.1.1 数组ADT的定义

1. 数组逻辑结构

数组数据结构的形式化定义:

1-ARRAY=(D,R)

$D=\{ a_i \mid i=c_1..d_1 \quad a_i \in D_0 \}$

$R=\{ R_1 \}$

$R_1=\{ <a_i,a_{i+1}> \mid a_i,a_{i+1} \in D \quad c_1 \le i \le d_1-1 \}$

哇!
与一般线性表完全一样啊!

一维数组: a_{i-1} a_i a_{i+1}

4.1 数组

4.1.1 数组ADT的定义

1. 数组逻辑结构

2-ARRAY=(D,R)

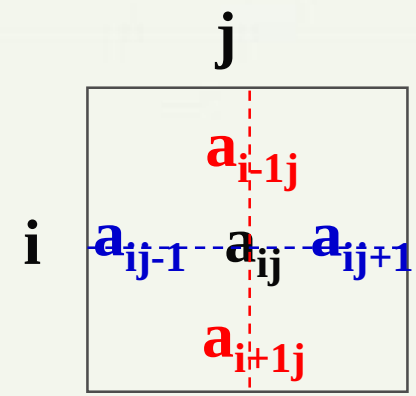
D={ a_{ij} | i=c_1..d_1,j=c_2..d_2 a_{ij}∈D_0 }

R={ ROW,COL}

ROW={ <a_{ij},a_{ij+1}> | a_{ij},a_{ij+1} ∈D c_1≤ i ≤d_1 c_2≤ j ≤d_2-1 }

COL ={ <a_{ij},a_{i+1j}> | a_{ij},a_{i+1j} ∈D c_1≤ i ≤d_1-1 c_2≤ j ≤d_2 }

二维数组：

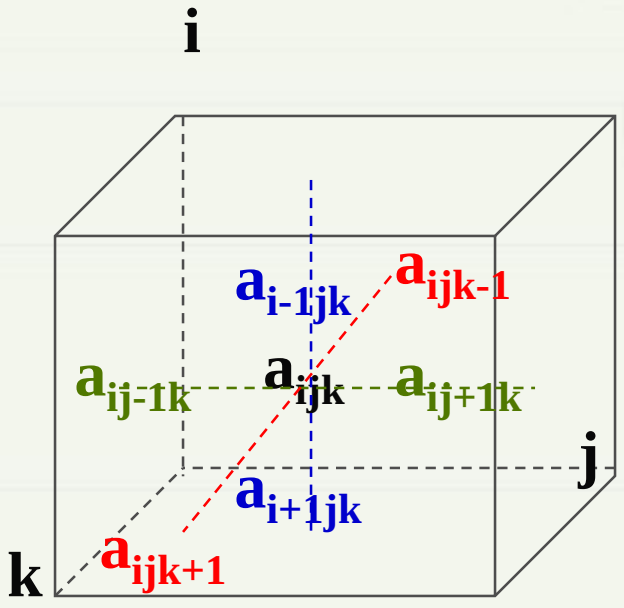


4.1 数组

4.1.1 数组ADT的定义

1. 数组逻辑结构

三维数组：



3-ARRAY=(D,R)

$D=\{a_{ijk} \mid i=c_1..d_1, j=c_2..d_2\ k=c_3..d_3\ \ a_{ijk} \in D_0\}$

$R=\{R_1,R_2,R_3\}$

$R_1=\{<a_{ijk},a_{ijk+1}> \mid a_{ijk},a_{ijk+1} \in D, c_1 \le i \le d_1, c_2 \le j \le d_2, c_3 \le k \le d_3-1\}$

$R_2=\{<a_{ijk},a_{ij+1k}> \mid a_{ijk},a_{i+1jk} \in D, c_1 \le i \le d_1, c_2 \le j \le d_2-1, c_3 \le k \le d_3\}$

$R_3=\{<a_{ijk},a_{i+1jk}> \mid a_{ijk},a_{i+1jk} \in D, c_1 \le i \le d_1-1, c_2 \le j \le d_2, c_3 \le k \le d_3\}$

4.1 数组

4.1.1 数组ADT的定义

1. 数组逻辑结构

n维数组:

n-ARRAY=(D,R)

$D=\{a_{j1j2...jn} \mid j_i=c_i..d_i \quad i=1,2...n\ a_{j1j2...jn}\in D_0\}$

$R=\{R_1,R_2,...,R_n\}$

$R_i=\{<a_{j1j2...ji...jn},a_{j1j2...ji+1...jn}> \mid a_{j1j2...jn}\in D$

$c_k\le j_k\le d_k \quad 1\le k\le n \quad k\neq i,\ c_i\le j_i\le d_i-1 \quad \}$

$i=1,2,\ldots,n$

4.1 数组

4.1.1 数组ADT的定义

2. 数组上定义的操作

由于数组中的数据元素受多个线性关系的约束，“牵一发而动全身”，因此一般做静态数据处理，不进行诸如插入、删除这样的操作！

（1）初始化：为数组分配存储空间；

（2）元素的存取：已知下标访问元素；

4.1 数组

4.1.1 数组ADT的定义

3. 数组ADT

ADT n-Array

data structure:

n-ARRAY=(D,R)

$D=\{a_{j1j2...jn} \mid ji=ci..di \quad i=1,2...n,\ a_{j1j2...jn} \in D_0\}$

$R=\{R_1,R_2,...,R_n\}$

$R_i=\{<a_{j1j2...ji...jn},a_{j1j2...ji+1...jn}> \mid a_{j1j2...jn} \in D$

$c_k \le j_k \le d_k \quad 1 \le k \le n \quad k \ne i \quad c_i \le j_i \le d_i-1 \quad \}$

operations:

Create(A);

Store(A, Index, value);

Get(A,Index)

END

4.1 数组

4.1.2 数组ADT的实现

由于数组这种线性数据结构的特殊性（多个线性关系约束元素），其上定义的操作很简单，所以数组ADT实现一般都采用了顺序存储结构。

几乎在所有的语言中，都已经把数组ADT物理实现了，即我们可以直接使用数组数据类型来求解问题。

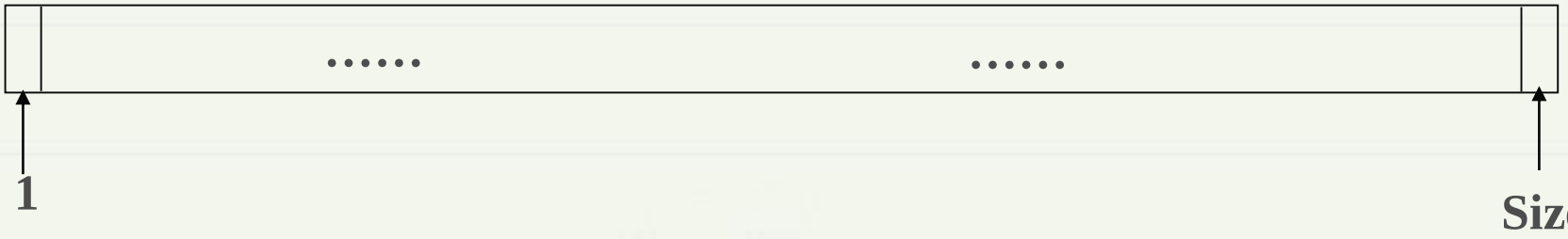
本节简单了解一下数组ADT是如何实现的，以便能更深刻地理解和使用数组。

4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

（1）存储方式：顺序存储，即用地址连续的一段存储空间依次存放数组的各个元素。



- 开辟多大的空间？
- 如何把元素“依次”存放到空间中去而且把关系也表示出来

4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

显然，一维数组很简单，按照线性关系顺序存放就可以了！即，第1个元素存放在空间的第1个位置，第2个元素存放在空间的第2个位置，……。关系通过物理相邻表示出来。

二维数组时，元素如何存放进去？ 即第i个存储空间存放那个元素？此时，逻辑上元素的位置是由行和列位置共同决定的，即按行数出的元素位置和按列数出的元素位置不一样！

一般地，多维数组中，数据元素的位置是由数据元素在各个关系上的位置确定的，即下标值$(i_1,i_2,...i_k)$；那么，那个在关系先，那个关系在后？

4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

通常，把线性关系正向或反向顺序排列，例如对二维数组，两个线性关系，顺序有“行-列”或“列-行”。分别称为行主序和列主序。

$$R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow \cdots\cdots \rightarrow R_{k-1} \rightarrow R_k$$

$$R_k \rightarrow R_{k-1} \rightarrow R_{k-2} \rightarrow \cdots\cdots \rightarrow R_2 \rightarrow R_1$$

关系的优先次序约定好后，所有元素就可以唯一的排出一个次序。

假设k维数组，其各维的界为: ($c_1..d_1$, $c_2..d_2$, ..., $c_k..d_k$)

则数组的元素个数

$$n=(d_1-c_1+1)*(d_2-c_2+1)...*(d_k-c_k+1)$$

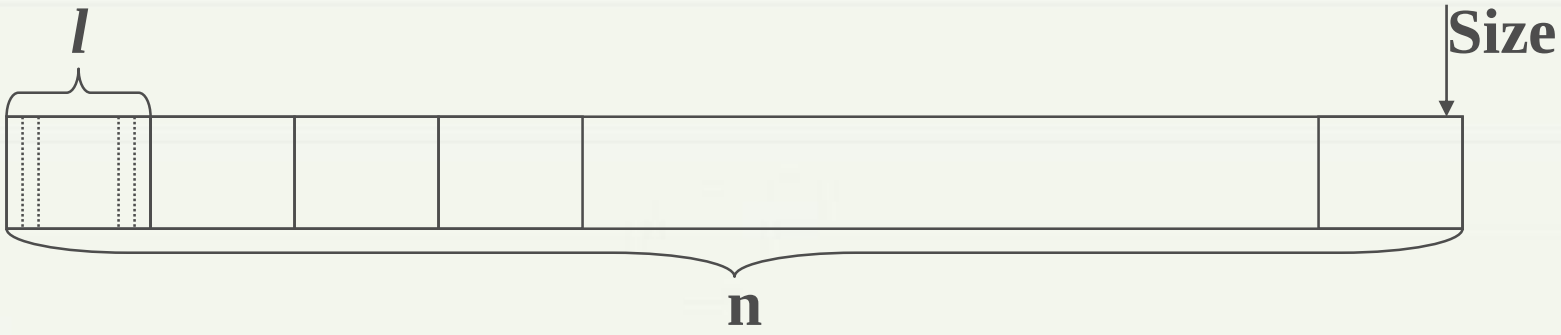
4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

每个数据元素占用各 *l* 空间，则占用空间为：

$$Size=n*l==(d_1-c_1+1)*(d_2-c_2+1)...*(d_k-c_k+1)*l$$



然后，元素按照线性关系的“顺序”就有一个排列顺序，从第1~n个。则序号为*i*的元素存放到第*i*个空间中去。

4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

例如，二维数组 A[1..4,1..5]

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{matrix}$$

空间大小=4*5*l=20l

元素的存储：

行主序： $a_{11}a_{12}a_{13}a_{14}a_{15}a_{21}a_{22}a_{23}a_{24}a_{25}a_{31}a_{32}a_{33}a_{34}a_{35}a_{41}a_{42}a_{43}a_{44}a_{45}$

列主序： $a_{11}a_{21}a_{31}a_{41}a_{12}a_{22}a_{32}a_{42}a_{13}a_{23}a_{33}a_{43}a_{14}a_{24}a_{34}a_{44}a_{15}a_{25}a_{35}a_{45}$

4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

再如：数组 A[0..3, 1..2, 3..6]　空间大小=4*2*4*l=32l

元素的存储：

R_1-R_2-R_3

a[0,1,3]	a[0,1,4]	a[0,1,5]	a[0,1,6]	a[0,2,3]	a[0,2,4]	a[0,2,5]	a[0,2,6]
a[1,1,3]	a[1,1,4]	a[1,1,5]	a[1,1,6]	a[1,2,3]	a[1,2,4]	a[1,2,5]	a[1,2,6]
a[2,1,3]	a[2,1,4]	a[2,1,5]	a[2,1,6]	a[2,2,3]	a[2,2,4]	a[2,2,5]	a[2,2,6]
a[3,1,3]	a[3,1,4]	a[3,1,5]	a[3,1,6]	a[3,2,3]	a[3,2,4]	a[3,2,5]	a[3,2,6]

R_3-R_2-R_1

a[0,1,3]	a[1,1,3]	a[2,1,3]	a[3,1,3]	a[0,2,3]	a[1,2,3]	a[2,2,3]	a[3,2,3]
a[0,1,4]	a[1,1,4]	a[2,1,4]	a[3,1,4]	a[0,2,4]	a[1,2,4]	a[2,2,4]	a[3,2,4]
a[0,1,5]	a[1,1,5]	a[2,1,5]	a[3,1,5]	a[0,2,5]	a[1,2,5]	a[2,2,5]	a[3,2,5]
a[0,1,6]	a[1,1,6]	a[2,1,6]	a[3,1,6]	a[0,2,6]	a[1,2,6]	a[2,2,6]	a[3,2,6]

4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

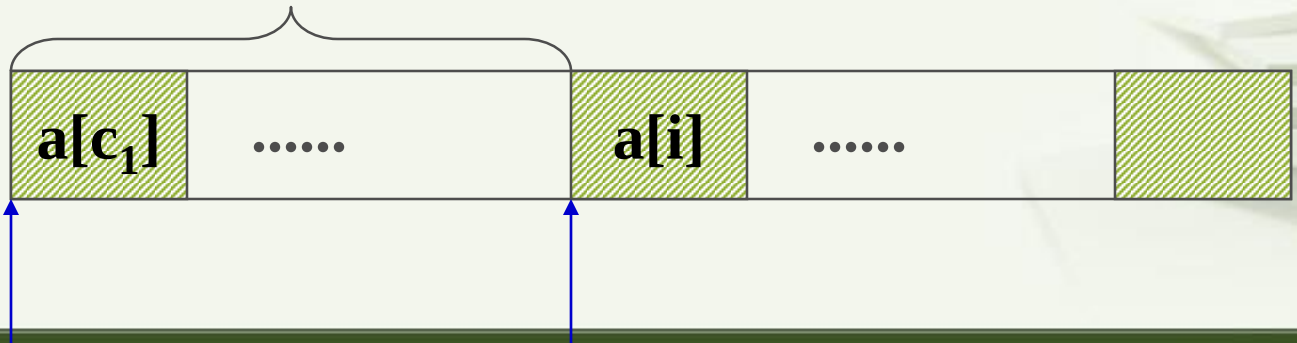
元素依次存放，关系是用物理上相邻来表示的。那么如何表示出多个线性关系呢？ 即逻辑关系和物理地址的对应如何？

一维数组： 逻辑与物理完全一致。

$A[c_1..d_1]$

行主序：$LOC(a[i])=LOC(a[c_1])+(i-c_1)*l$

列主序：$LOC(a[i])=LOC(a[c_1])+(i-c_1)*l$



4.1 数组

4.1.2 数组ADT的实现

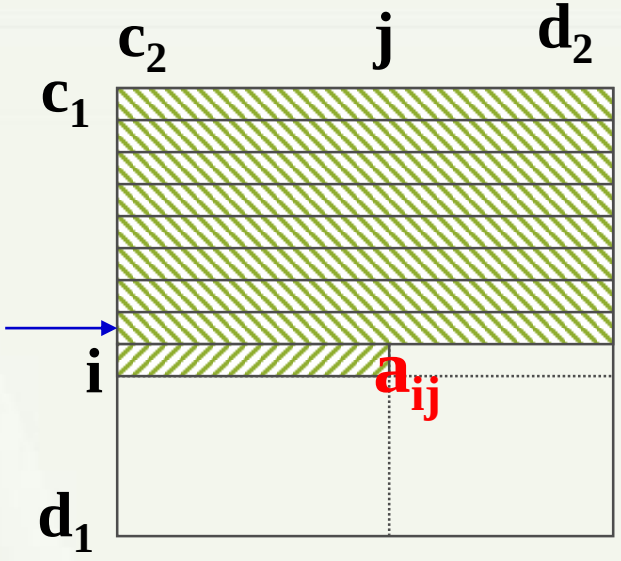
1. 数组的存储结构

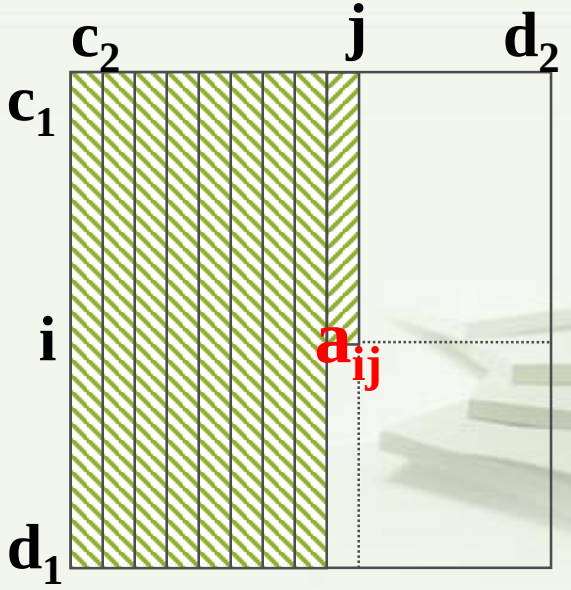
二维数组： 逻辑上(i,j)与一个物理位置对应。

$A[c_1..d_1,c_2..d_2]$

行主序：$LOC(a[i,j])=LOC(a[c_1,c_2])+(d_2-c_2+1)*(i-c_1)*l+(j-c_2)*l$

列主序：$LOC(a[i,j])=LOC(a[c_1,c_2])+(d_1-c_1+1)*(j-c_2)*l+(i-c_1)*l$





4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

三维数组： 逻辑上(i,j,k)与一个物理位置对应。

$A[c_1..d_1,c_2..d_2,c_3..d_3]$

行主序：$LOC(a[i,j,k])=LOC(a[c_1,c_2,c_3])$ //基地址

$+(d_2-c_2+1)*(d_3-c_3+1)*(i-c_1)*l$ //体

$+(d_3-c_3+1)*(j-c_2)*l$ //面

$+(k-c_3)*l$ //线

列主序：$LOC(a[i,j,k])=LOC(a[c_1,c_2,c_3])$

$+(d_1-c_1+1)*(d_2-c_2+1)*(k-c_3)*l$

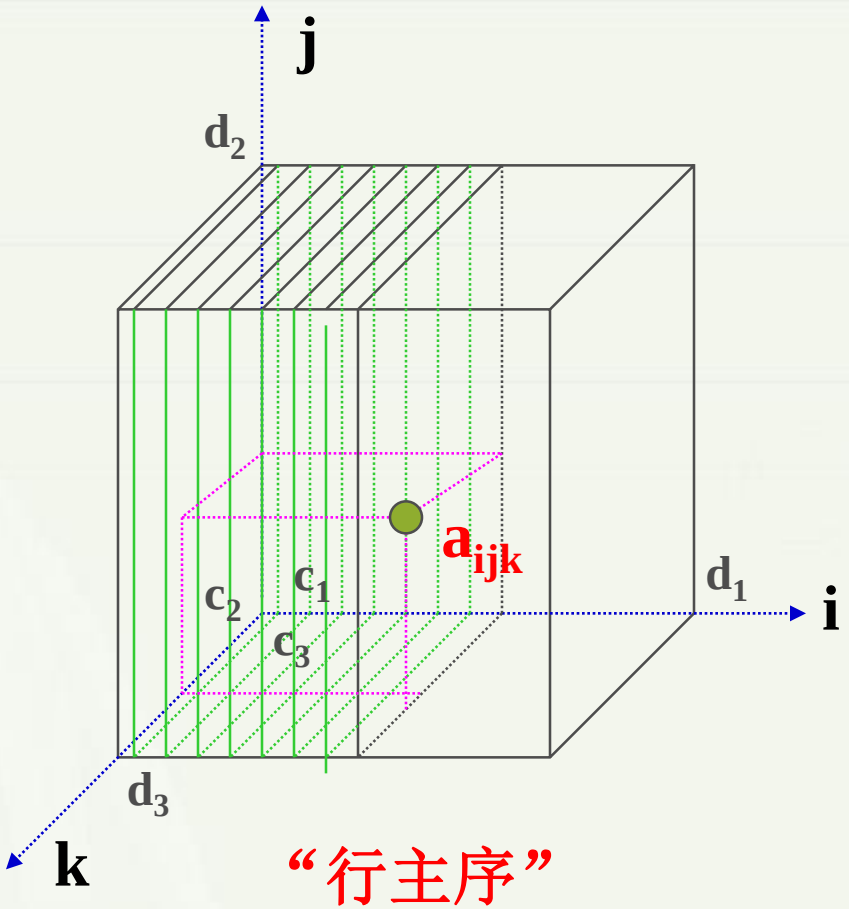
$+(d_1-c_1+1)*(j-c_2)*l$

$+(i-c_1)*l$

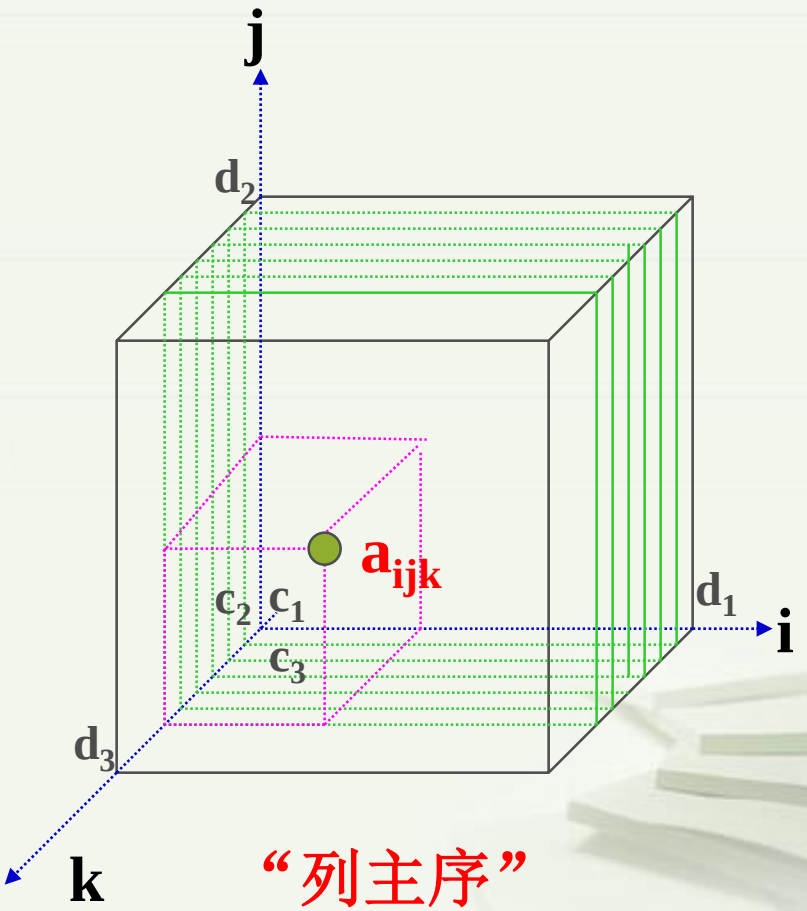
4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构



“行主序”



“列主序”

4.1 数组

4.1.2 数组ADT的实现

1. 数组的存储结构

特点：随机存取，即存取任何元素花费的时间相同。

实现：

根据数组说明，得到数组的元素个数，及元素类型（每个元素占用空间），然后，分配连续空间，空间的首地址存储在数组名中。

2. 数组操作的实现

已知要访问的元素的下标，则根据存储方式可以计算出该元素的存储地址（根据前面的公式），于是可以存取该元素。

4.1 数组

4.1.3 矩阵及特殊矩阵

1. 矩阵ADT定义

ADT Matrix

data structure:

Matrix=(D,R)

D={a_{ij} | i=1..m,j=1..n $a_{ij}\in D_0$ }

R={ROW,COL}

ROW={<a_{ij},a_{ij+1}> | a_{ij},a_{ij+1} $\in$D 1≤ i ≤m 1≤ j ≤n-1 }

COL ={<a_{ij},a_{i+1j}> | a_{ij},a_{i+1j} $\in$D 1≤ i ≤m-1 1≤ j ≤n }

operations:

转置

加法

减法

乘法

输出

END

4.1 数组

4.1.3 矩阵及特殊矩阵

2. 矩阵ADT实现

存储结构：二维数组

操作实现：很简单，同学们自己写出。

3. 特殊矩阵的存储及操作的实现

对于一个矩阵结构显然用一个二维数组来存储是非常恰当的，但在有些情况下，比如常见的一些特殊矩阵，如三角矩阵、对称矩阵、带状矩阵、稀疏矩阵等，从节约存储空间的角度考虑，这种存储是不太合适的。下面从这一角度来考虑这些特殊矩阵的存储方法。

节省空间，但是带来了矩阵操作上的复杂！

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（1） 对称矩阵、上三角矩阵、下三角矩阵

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

a[i,j]=a[j,i]
0≤i≤n-1
0≤j≤n-1

对称矩阵

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

当 i>j 时 a[i,j]=0
0≤i≤n-1
0≤j≤n-1

上三角矩阵

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

当 i<j 时 a[i,j]=0
0≤i≤n-1
0≤j≤n-1

下三角矩阵

4.1 数组

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（1） 对称矩阵、上三角矩阵、下三角矩阵

一般存储方式：可以用二维数组存放，需要 n^2个存储单元。

优点：简单，矩阵的运算实现方便

缺点：空间效率不高，为什么？

为了节省空间，可以采用其他特殊存储方式，对相同值只存放一个，或者零元素不存储。——矩阵的压缩存储

4.1.3 矩阵及特殊矩阵

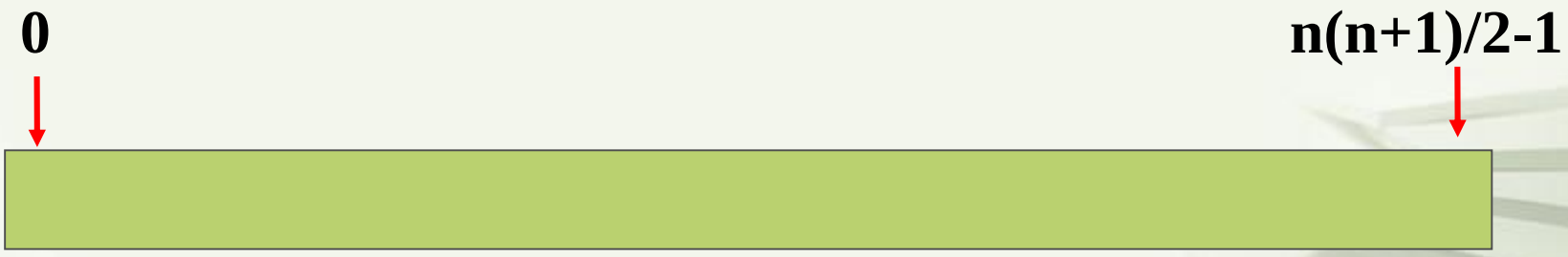
3. 特殊矩阵的存储及操作的实现

（1） 对称矩阵、下三角矩阵、上三角矩阵

分析：这类矩阵，真正需要存储的元素有 $\frac{n\times(n+1)}{2}$ 个。

$$n+(n-1)+(n-2)+\ldots+2+1=\frac{n\times(n+1)}{2}$$

因此，只要开辟 $\frac{n\times(n+1)}{2}$ 个连续空间就可以存放这样的矩阵！



4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

压缩存储方式：对下三角（或上三角）矩阵元素排成一个顺序，排在第i的元素存储在第i个位置。

显然：也有按行和列两种顺序！

按行：

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

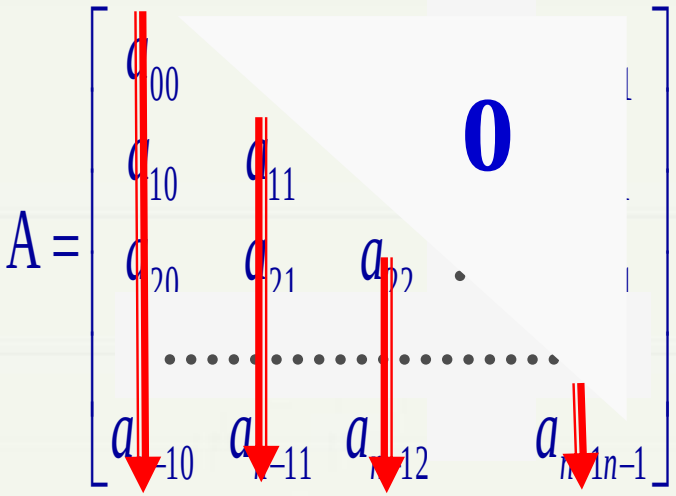
0	1	2	3	4	5	6	7	8		n(n+1)/2-1
a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}	……	a_{n-1n-1}

4.1 数组

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

按列：



0	1	2	3	4	5	6	7	8		n(n+1)/2-1
a_{00}	a_{10}	a_{20}	a_{30}	.....	a_{n-10}	a_{11}	a_{21}	a_{31}	.....	a_{n-1n-1}

4.1 数组

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

我们以下三角、行主序为例，可以推出其地址计算公式：

对于下三角矩阵a中的任何一个元素a_{ij}，当 $i \geq j$ 时，元素位于下三角部分，因此要在空间存放，其存储位置为：

$$LOC(a_{ij})=1+2+3+\ldots+i+j=(i+1)*i/2+j \qquad i \geq j$$

而对下三角矩阵a中的任何一个元素a_{ij}，当 $i<j$ 时，元素位于上三角部分，没有在空间中存放，但是其对称元素a_{ji}存放了，因此，也可以计算出其存储位置：

$$LOC(a_{ij})=LOC(a_{ji})=1+2+3+\ldots+j+i$$
$$=(j+1)*j/2+i \qquad i<j$$

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

相反，对于存储空间中的一个元素b_k，它唯一对应下三角矩阵中的一个元素a_{ij}。

有兴趣的同学可以自己推导出：

若已知某矩阵元素位于连续存储空间的第 k个位置（$k \geq 0$），可寻找满足 $i\,(i+1)/2 \leq k < (i+1)*(i+2)/2$ 的 i，
此即为该元素的行号。

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

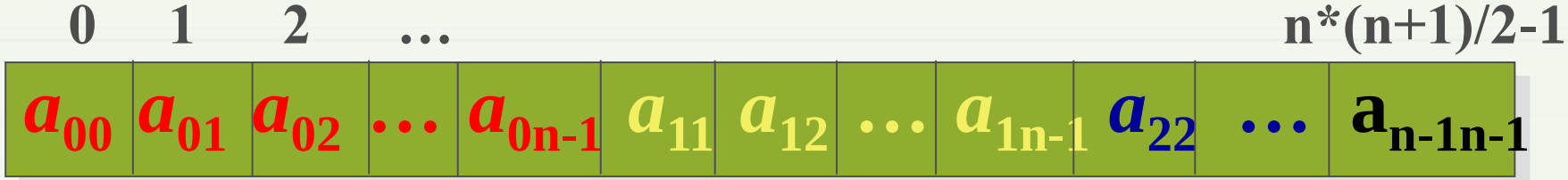
此即为该元素的列号。

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

类似，对于上三角矩阵，也可以采用相同的存储策略来存储（也分行主序和列主序）。

对于上三角矩阵中的任何一个元素a_{ij}，假设采用行主序，其存储地址计算公式如下：



$$LOC(a_{ij}) = n + (n-1) + (n-2) + \cdots + (n-i+1) + j - i$$
$$= (2*n - i - 1) * i / 2 + j \qquad i \le j$$

$$LOC(a_{ij}) = (2*n - j - 1) * j / 2 + i \qquad i \ge j$$

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（2） 三对角矩阵、带状矩阵

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。

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

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

总共有$3n\text{-}2$个非零元素。

一般存储方式：

可以用二维数组存放，需要 n^2个存储单元。有$n^2\text{-}3n\text{+}2$个存放0。空间效率低。

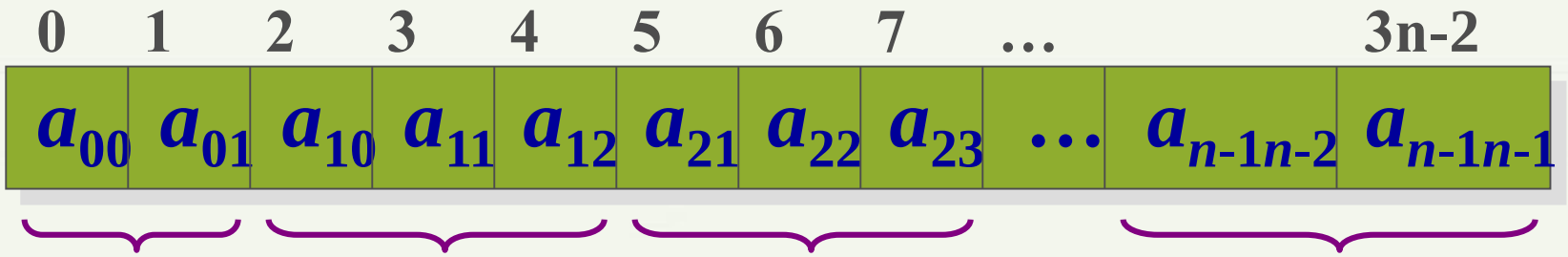
4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（2） 三对角矩阵、带状矩阵

压缩存储方式：

思想同上，即用连续存储空间只存放非0元素。即开辟3n-2个连续空间单元。



则，对于带状矩阵a中的任何一个元素a_{ij}，当 $0 \le i \le n-1$，$i-1 \le j \le i+1$时，在存储空间中有存放，存储位置可以计算出来：

$$LOC(a_{ij})=3*i-1+j-i+1=2*i+j$$

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（2） 三对角矩阵、带状矩阵

相反，若已知三对角矩阵中某元素 a_{ij} 在空间中的存储位置为k，也可以推出其在原三对角矩阵中的逻辑位置：

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

$$j=k-2*i$$

对于带状矩阵，处理方式类似。（略）

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

稀疏矩阵：零元素比较多，且分布不均匀(没有规律)。

- 设矩阵 A 中有 s 个非零元素，若 s 远远小于矩阵元素的总数（即s<<m×n），则称 A 为稀疏矩阵。

- 令 e = s/(m*n),称 e 为矩阵的稀疏因子。
- 有人认为 e≤0.05 时称之为稀疏矩阵。

4.1 数组

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

一般存储方式：可以用二维数组存放，需要 n^2个存储单元。但是大部分存放的是0。空间效率低。

*分析：*可以考虑只存放非0元素，但如果仅仅存放一个元素值是不够的，因为不能确定它在矩阵中的位置，所以必须还要存储其在矩阵中的位置（下标）。

*这样：*每个非0元素可以用一个三元组(i,j,v) 表示。于是一个稀疏矩阵就是这样一些三元组的集合，考虑它们在矩阵中的位置，稀疏矩阵就可以抽象为一个以三元组为数据元素的线性表（线性关系可以是行或列）。

4.1 数组

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

例如有稀疏矩阵：

$$\begin{matrix} 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{matrix}$$

可抽象为线性表：

按行： ((1,3,1), (1,5 2), (3,1,1), (4,3,1), (4,5,-1))

按列： ((3,1,1), (1,3,1), (4,3,1), (1,5,2), (4,5,-1))

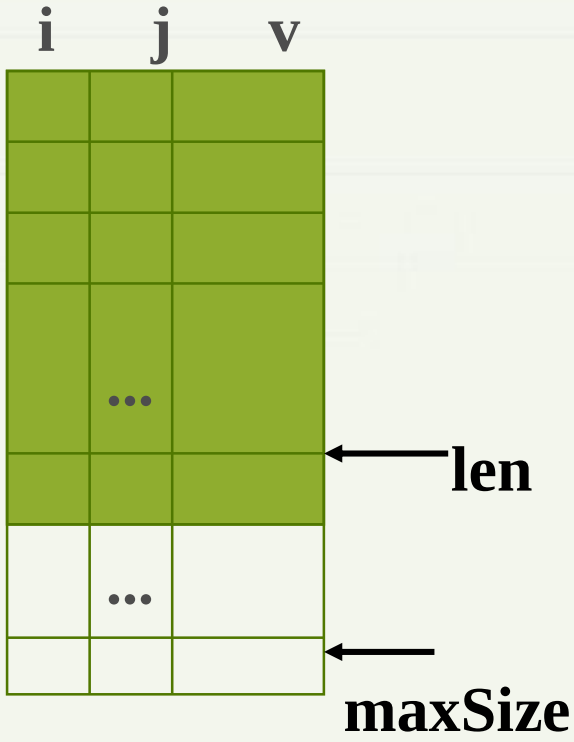
线性表ADT的实现前面我们已经学习。存储线性表有顺序和链式两种，所以稀疏矩阵也有两种存储形式！

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

稀疏矩阵压缩存储方式之一：三元组顺序存储



4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

```
typedef 矩阵元素类型 ElemType;
struct Triple   //三元组定义，即线性表元素类型
{
    int  row, col;                  //非零元素行号/列号
    ElemType value;                 //非零元素的值
    void operator = (Triple & R)          //赋值
      { row = R.row;  col = R.col;  value = R.value; }
};
```

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

```
class SparseMatrix   //稀疏矩阵三元组类定义
{  public:
      SparseMatrix (int Rw = drows, int Cl = dcols,
        int Tm = dterms);                          //构造函数
      void Transpose(SparseMatrix& b);             //转置
      void Add (SparseMatrix& a, SparseMatrix& b);  //a = a+b
      void Multiply (SparseMatrix& a, SparseMatrix& b);  //a = a*b
   private:
      int Rows, Cols, Terms;        //矩阵的行、列，非零元素数
      Triple   *smArray;                    //三元组表，顺序存储
};
```

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

各个操作函数的实现：

稀疏矩阵采用三元组存储后，存储空间节省了，但是矩阵的运算采用传统的方法不可以了，需要设计新的实现算法。

输入、输出、转置、加、乘、……

见教材P143-152 （略）

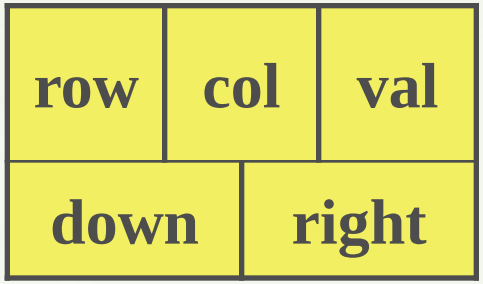
4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

稀疏矩阵压缩存储方式之二：三元组链式存储—十字链表

每个三元组用一个如下的结点存储：



- row,col :非0元素的行、列值
- val : 非0元素的值
- down : 指向同列的下一个非0元素结点
- right : 指向同行的下一个非0元素结点

单选题 1分

设置

数组是一种特殊的线性表。它特殊在：

A 数据元素

B 定义的操作

C 元素受非线性关系约束

D 元素受多个线性关系约束

提交

单选题 1分

设置

特殊矩阵如果采用二维数组存储，空间利用率很低。采用所谓“压缩”存储的目的是为了节省空间，不同的特殊矩阵，有不同的压缩存储方案。对于带宽是k的nn矩阵，如果采用只存储带上的元素，则节省的空间是：

A n*(2k+1)-2k

B n*(2k+1)

C n*n- n*(2k+1)+2k

D n*n-n*(2k+1)

提交

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

（3） 稀疏矩阵

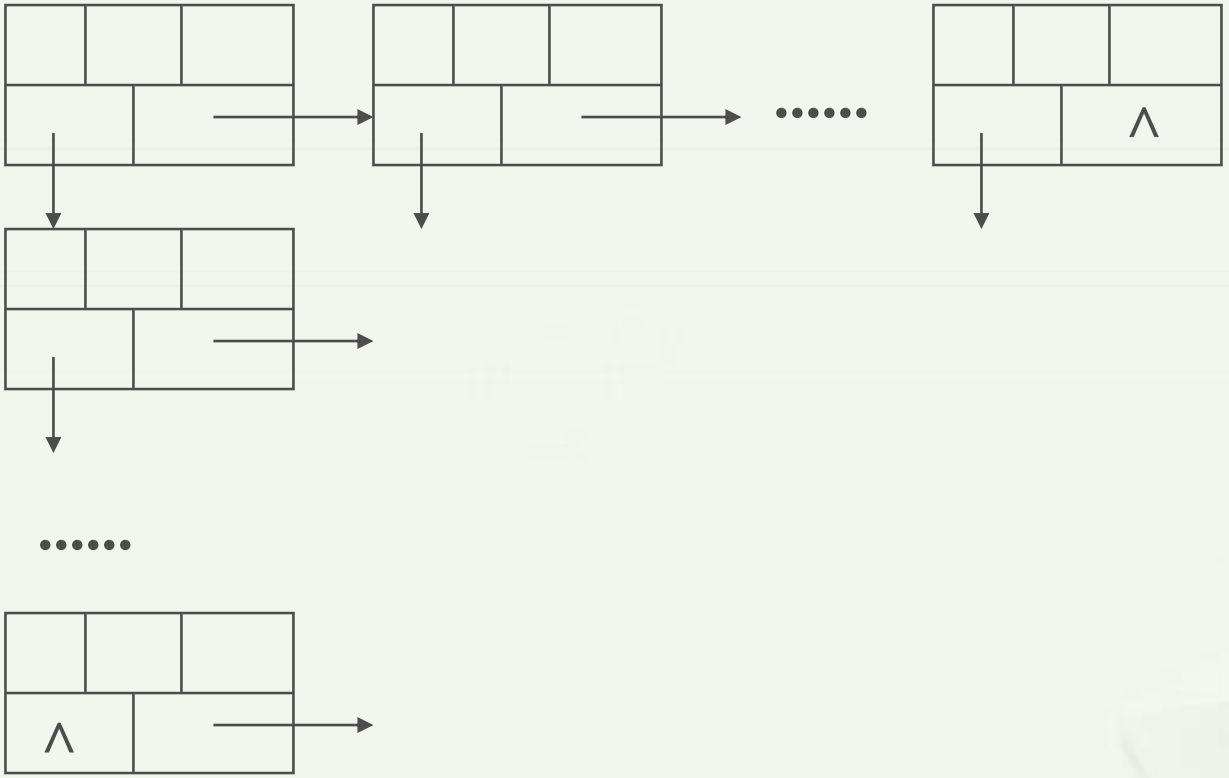
结点的结构定义如下:

```
typedef  struct   node
{     int  row, col;
      struct node *down , *right;
      union  v_next
         {  ElemType  val;
            struct node  *next;
          }
} MNode，  *MLink;
```

4.1.3 矩阵及特殊矩阵

3. 特殊矩阵的存储及操作的实现

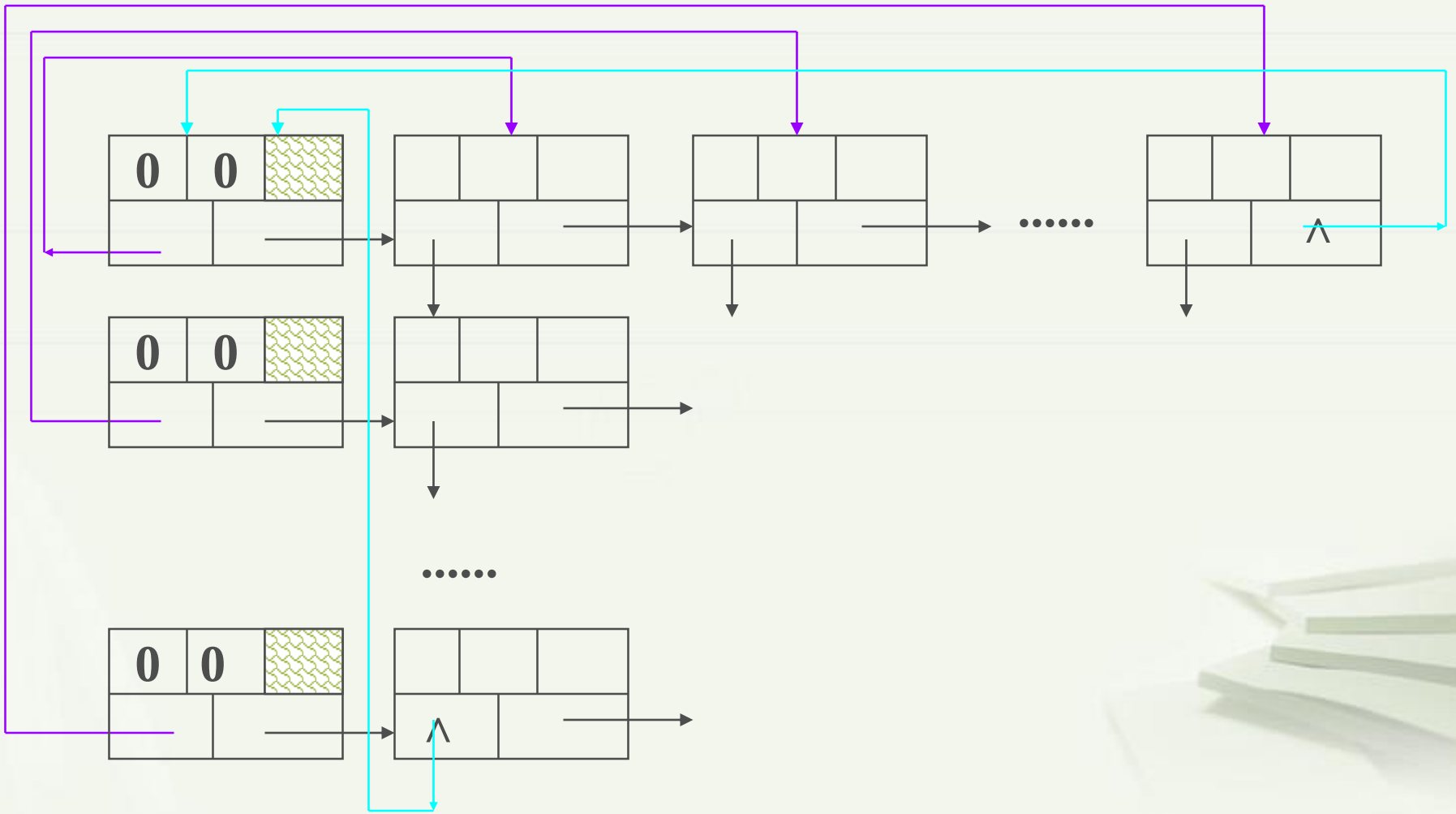
所以：所有非0元素的结点按行看或按列看都是一些单链表！



4.1 数组

4.1.3 矩阵及特殊矩阵

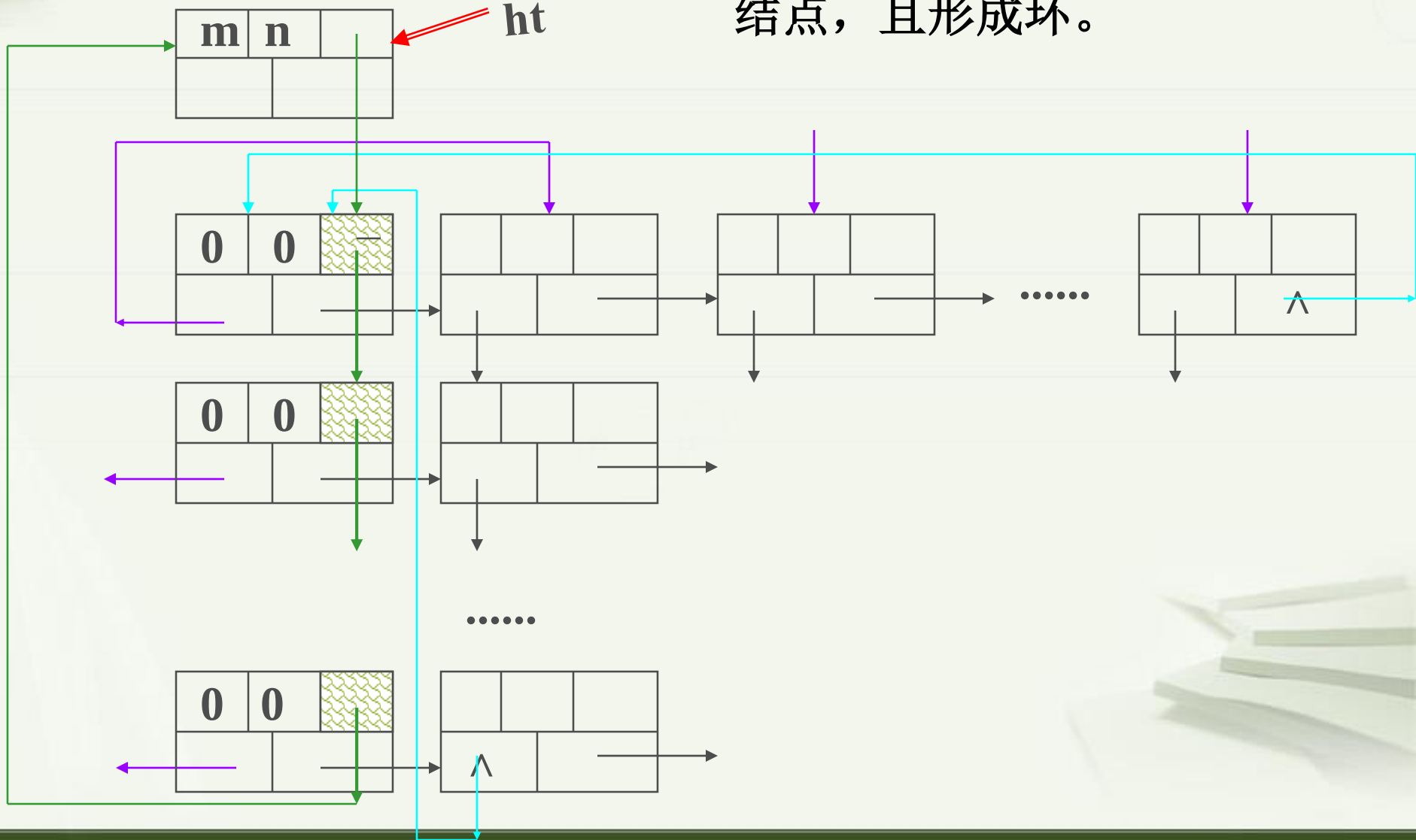
每个单链表加上头结点（行、列），且构成环



4.1 数组

4.1.3 矩阵及特殊矩阵

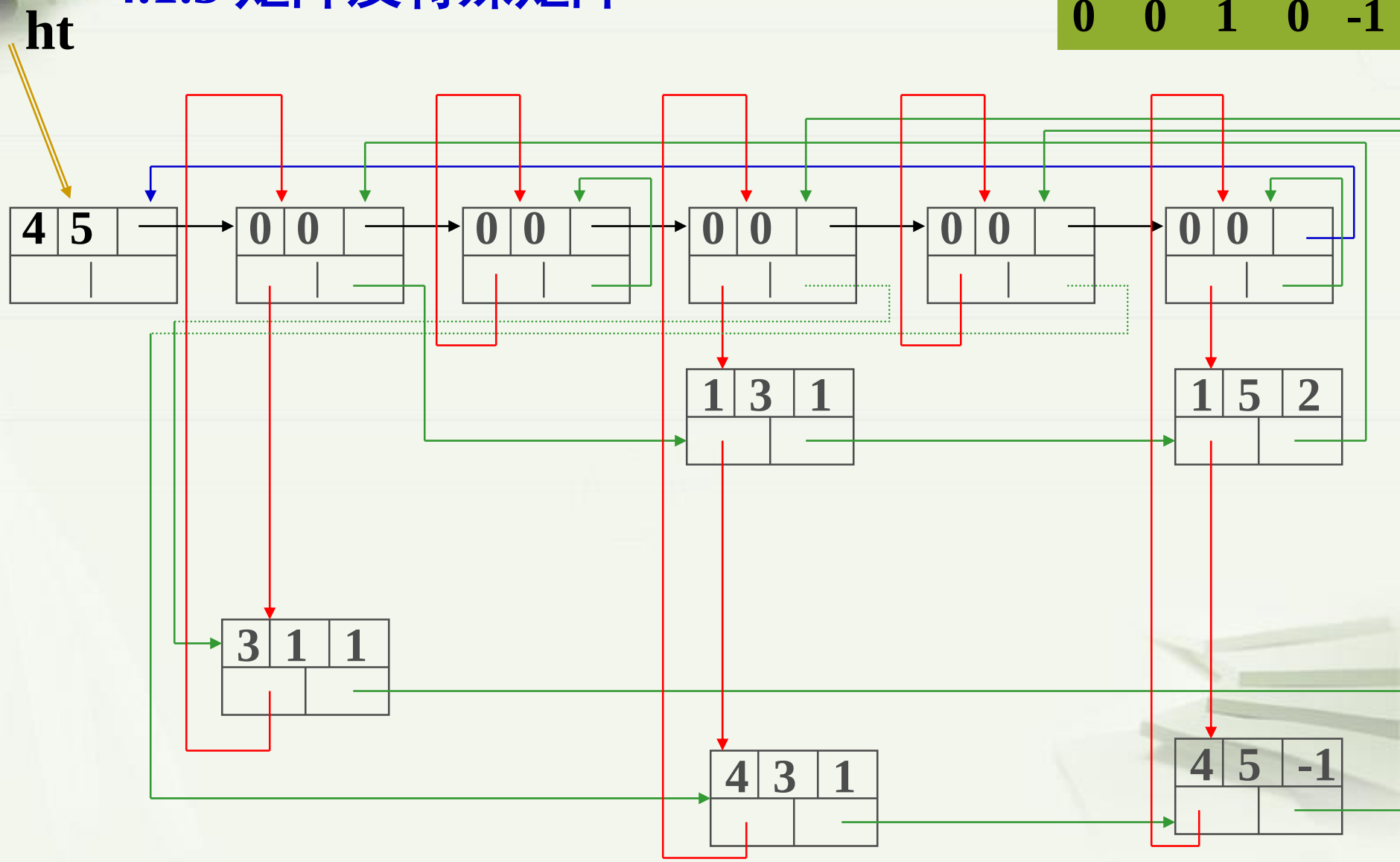
头结点形成链表，加一个头结点，且形成环。



4.1 数组

4.1.3 矩阵及特殊矩阵

0	0	1	0	2
0	0	0	0	0
1	0	0	0	0
0	0	1	0	-1



4.1 数组

4.1.3 矩阵及特殊矩阵

矩阵操作肯定比原来复杂了！

请同学们自己学习、练习！

建立

输出

……

4.2 字符串

4.2.1 字符串ADT定义

1. 字符串逻辑结构

[串 String]:简单说，它是有限字符集中的零个或多个*字符*组成的有限序列。它是一种特殊的线性表（数据集为字符集，元素之间具有线性关系）。

String=(D,R)

D={c_i | $c_i \in D_0$ D_0=CHARACTER i=1,2,...,n n>=0}

R= {<c_{i-1},c_i> | c_{i-1},c_i $\in D_0$ i=2,3,...,n}

特点：数据元素都是字符，使得它的操作的对象一般不再是单个数据元素，而是一组数据元素（子串）。

4.2 字符串

4.2.1 字符串ADT定义

1. 字符串逻辑结构

空串	长度为零的字符串，n=0
空格串	数据元素都是空格的字符串；
子串	串中连续的任意个字符组成的子序列，称为该串的子串；
主串	包含子串的串；
字符在串中的位置	字符在串中的序号（即第几个数据元素）
子串在串中的位置	子串的第一个字符在主串中的位置；
串相等	两个串的长度相等，且各对应位置处的字符都相等；

4.2 字符串

4.2.1 字符串ADT定义

2. 字符串结构上定义的操作

- 串置空
- 串赋值(创建)
- 串复制
- 判断串比较
- 求串长度
- 串联结
- 取子串
- 定位
- 串置换
- 串插入
- 串删除
- 判断是否空串
- 串销毁

其中下面5个操作称为最小操作子集：

- 串赋值，StrAssign
- 串比较，StrCompare
- 求串长，StrLength
- 串联接，Concat
- 求子串，SubString

注：这5种操作不可能利用其他串操作来实现，但其他串操作均可在这个最小操作子集上实现。

4.2 字符串

4.2.1 字符串ADT定义

3. 字符串ADT

ADT String

data structure:

$D=\{c_i \mid c_i \in D_0 \quad i=1,2,\ldots\ldots \quad n>=0\}$

$R=\{<c_{i-1},c_i> \mid c_{i-1},c_i \in D_0\ D_0=\text{CHARACTER}\ ,\quad i=2,3,4,\ldots\ldots\}$

operations:

Setnull(S);

StrAssign(s,t);

StrCompare(s, t);

StrLength(s);

StrConcat(s,t);

Substr(s,start,len) ;

Index(s,t);

StrRepacle(s,t,v);

StrInsert(s,pos,t);

StrDelete(s,pos,len);

END

4.2 字符串

4.2.2 字符串ADT的实现

在很多计算机程序设计语言中，已经把字符串ADT物理实现为字符串类型(string)。即我们可以直接使用语言提供的字符串数据类型来求解问题。

例如C++ 对字符串ADT的实现，函数库名为"string.h"，要使用串操作，只要加载该头文件即可。具体实现的操作见p154-156:

strcpy, strncpy, strcat, strncat, strlen, strstr, ……

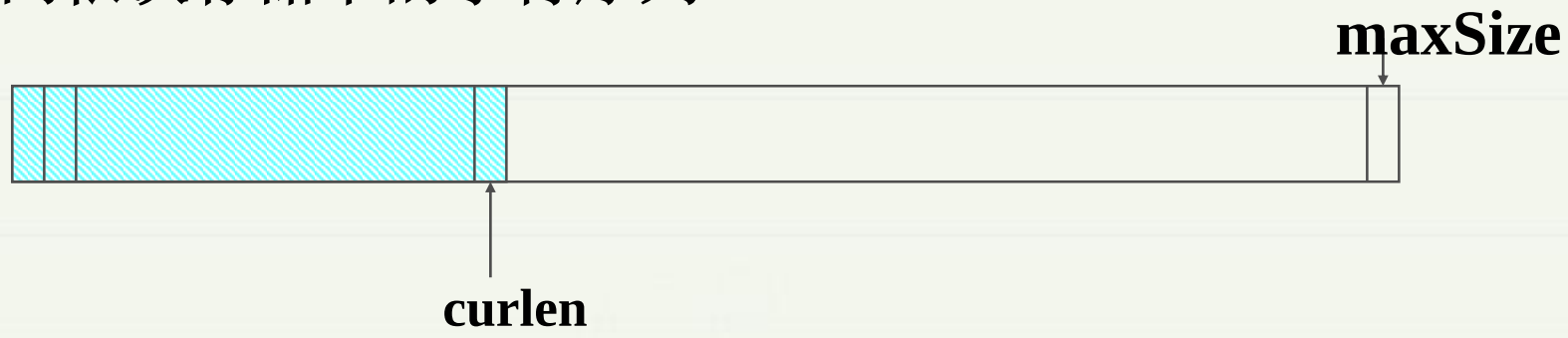
本节简单了解一下字符串ADT是如何实现的，以便能更深刻地理解和求解问题时更好地使用字符类型。其中重点应该是串的操作的实现。

4.2 字符串

4.2.2 字符串ADT的实现

1. 字符串的存储结构

存储方式：采用顺序存储结构。即用一组地址连续的存储空间依次存储串的字符序列。




```
#define  maxSize    串的最大长度
typedef char ElemType;  //定义数据元素类型为CHAR
 struct SeqString
{  ElemType  ch[maxSize];      //表示存放串值的连续空间
   int       curlen;           //curlen当前串的长度
}String;
```

4.2 字符串

4.2.2 字符串ADT的实现

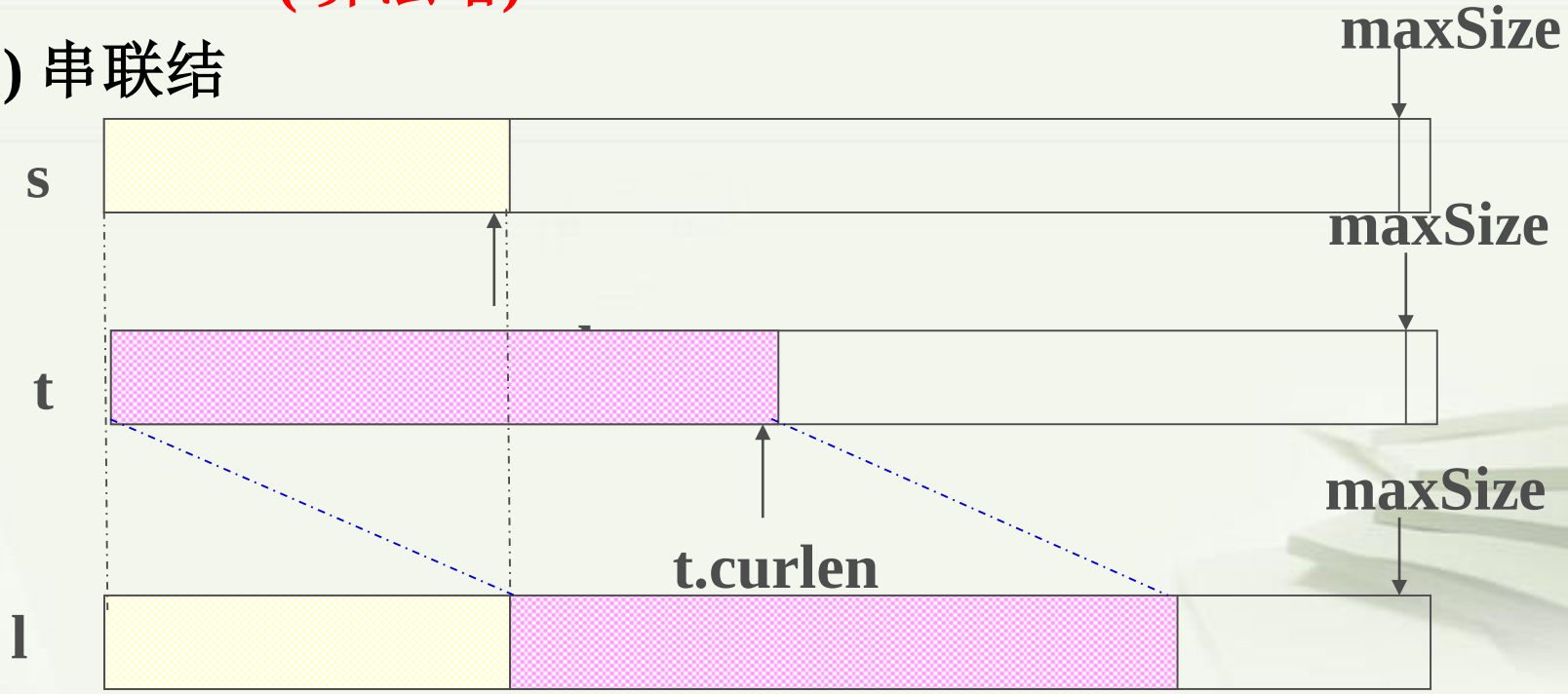
2. 字符串的操作的实现

(1) 判断串相等

长度相等，逐个字符比较

(算法略)

(2) 串联结



4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

分几种情况：

s.curlen+t.curlen<=maxSize: 正常联结

s.curlen+t.curlen>maxSize)

AND(s.curlen<maxSize): 没有正常联结

但联接了部分

s.curlen=maxSize: 没有正常联结

(3) 取子串

有两种取法：其一：已知子串起点和长度；

其二：已知子串起点和终点；

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

(4) 子串定位

模式匹配：在一个主串中，查找子串是否存在，存在返回子串的位置；不存在返回0。

子串称为模式。

实现它的方法也很多，如BF算法、KMP算法、BM算法、KR算法等等。

我们只介绍最常见的两种：

BF算法——最简单的匹配算法；

KMP算法——效率较高的匹配算法；

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

朴素的字符串匹配算法-**Brute Force**算法，简称BF算法、蛮力匹配算法。

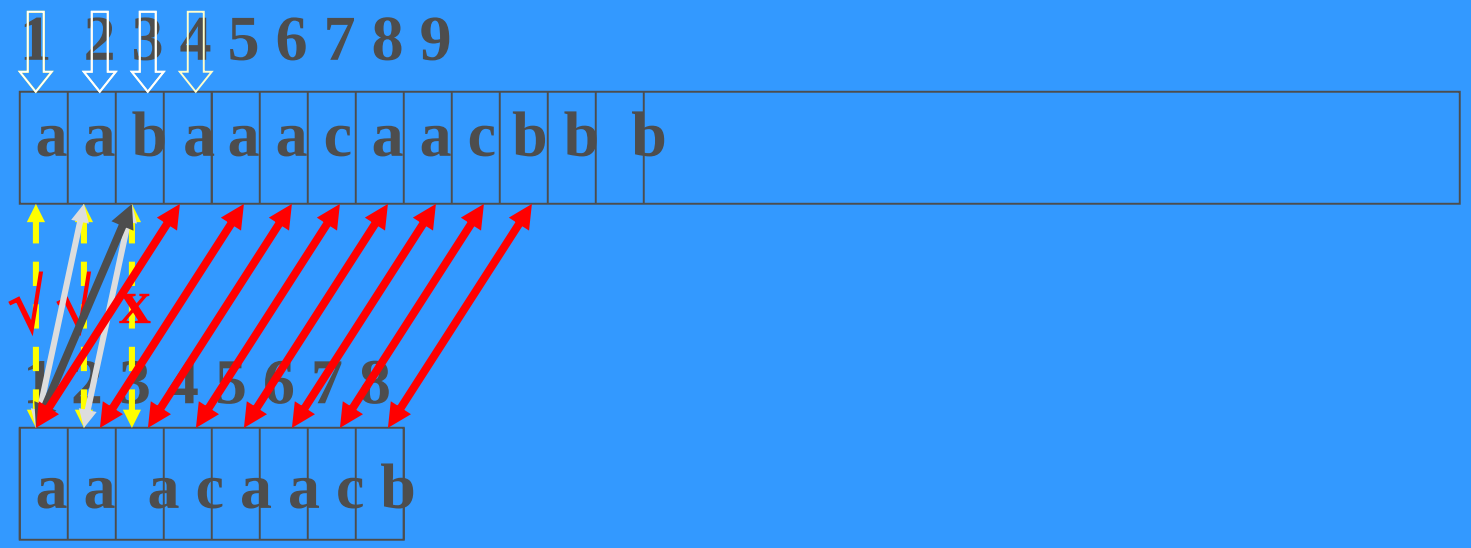
BF算法的基本思想：回溯

从主串的第i个位置开始，与子串的每个字符逐个比较，若均相等，则找到；否则，即在某个位置出现了不等，则说明从该起点开始的子串不是模式串，换一个新起点，重新开始！

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现



每次：

主串回溯到上次起点的下一个位置；
模式串回到1

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

算法如下：

```
int    INDEX  (String  s, String t, int pos)
//返回子串t在主串s中第pos个字符之后的位置，
//若不存在返回0
 {   i=pos;  j=1;
      while    ( i<=s.curlen )&&(j<=t.curlen)      //i<=s.curlen-t.curlen+1
          if  (s.ch[i]==t.ch [j])
            {  i++;   j++ ; }
          else
            { i=i-j+2 ;  j=1;}                  //将i指示器回溯
       if  ( j>t.curlen ) return  (i-t.curlen);
          else   return (0);                    //匹配失败
 }
```

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

教材P163 程序4.5

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

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

算法效率：

设主串长度为 m ， 模式串长度为 n ;

找到模式串（无回溯）：

最好：开始就是模式串, O(n)

aaaaabbbbbb

aaa

最坏：最后是模式串， O(m)

cbbbcbcbaaa

aaa

找不到模式串（无回溯）: O(m)

aaaaaaaaaaa

xyz

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

找到模式串（有回溯）：

最好：回溯很少，或没回溯 O(Index+n) O(m+n)

abcabcabcaab

aab

最坏：回溯很多，或每次都有回溯 O(Index*n)

O(m*n)

aaaaaaaaaaaa

aaab

*找不到模式串（有回溯）: O(m*n)*

aaaaaaaaaaaa

aaac

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

BF算法的特点：

*a、时间复杂性最坏为O(m*n) ，但一般情况下为O(m+n)，所以还是一个常用算法。*

b、由于有回溯，所以主串输入后必须保存。

下面介绍一种改进的模式匹配算法——KMP算法

D.E.Knuth

J.H.Morris

V.R.Pratt

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

I、KMP算法的基本思想：

按照BF算法：当比较到某个位置，不匹配时，不管比过的字符串是什么情况，都是进行回溯！那么，有没有可能在比较过的串中存在部分匹配的信息呢？？

假设主串为 s=‘ $s_1s_2s_3...s_n$ ‘
模式串为 t=‘ $t_1t_2t_3...t_m$ ’ m<<n
子串‘ $s_is_{i+1}...s_j$’ ‘$t_it_{i+1}...t_j$’ 分别用 s[i..j] 和 t[i..j] 表示
则模式匹配可描述为：

求最小的 i， $0 \le i \le n-m$，使得
t[1..m]=s[i+1,...i+m]

4.2 字符串

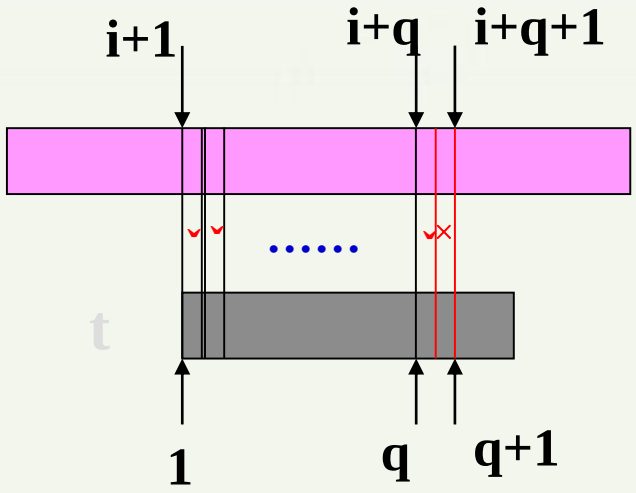
4.2.2 字符串ADT的实现

2. 字符串的操作的实现

假设在 s 中从 s_{i+1}开始匹配，遇到一个不完全的匹配，即得到一个 q (1≤q<m)，使得：

t[1..q]=s[i+1..i+q] 但

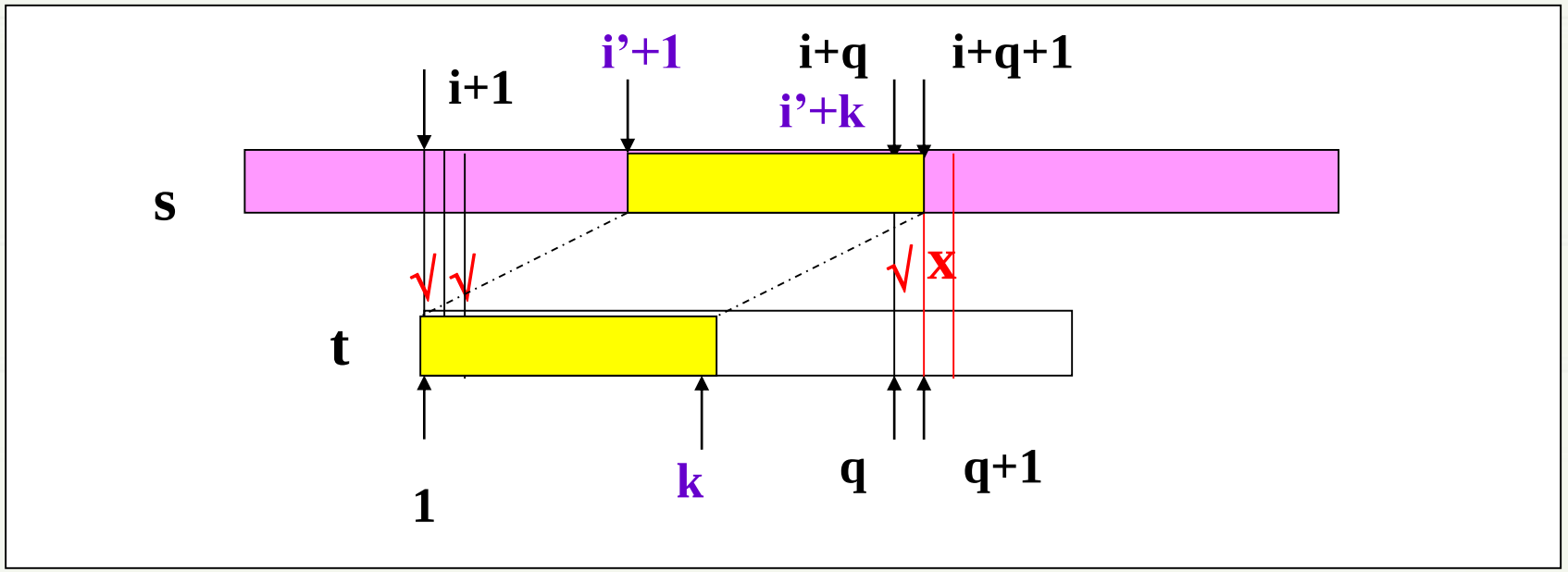
t[1..q+1]≠s[i+1..i+q+1]



4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现



如果能确定一个最小的整数 i’ ∈{i+1,i+2,...,i+q} 使得 t[1..k]=s[i’+1...i’+k]，其中 k=i+q-i’

那么，哪些测试和比较是多余的？？

4.2 字符串

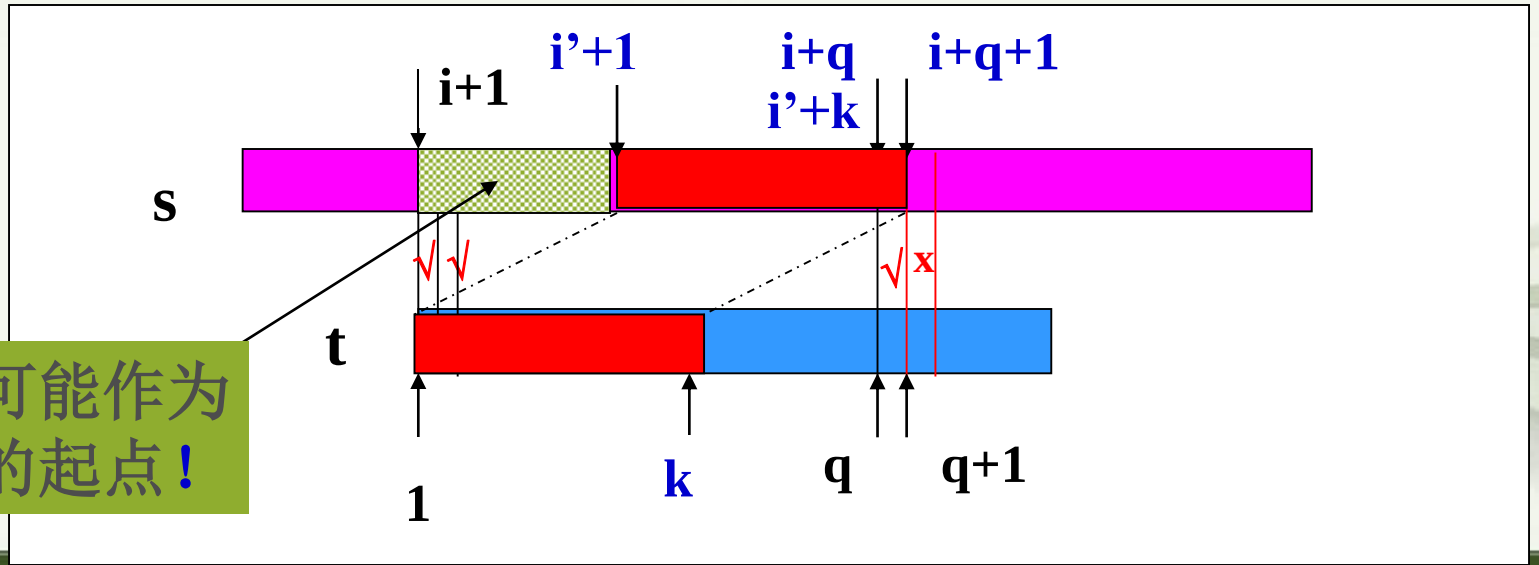
4.2.2 字符串ADT的实现

2. 字符串的操作的实现

A、由上面可知：对于 任意 *l*，$i<l<i'$ 有：

$t[1..k] \neq s[l+1..l+k]$ 从而可以肯定有

$t[1..m] \neq s[l+1..l+m]$ （部分不匹配，整体肯定不匹配！！），即下一步匹配可以跳到 $s_{i'+1}$



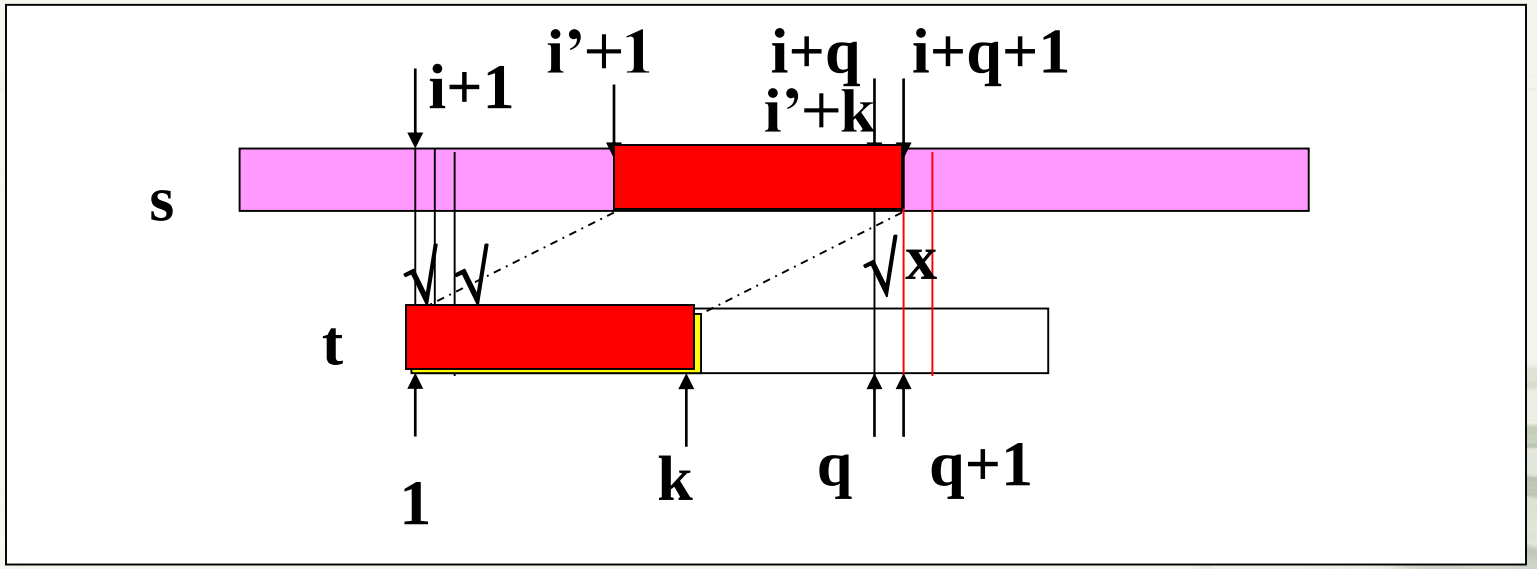
都不可能作为匹配的起点！

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

B、由于 t[1..k]=s[i’+1..i’+k] 所以从 i’ 开始的比较可以从 i’+k+1 开始，即从 t[k+1] 与 s[i’+k+1] 比较开始。

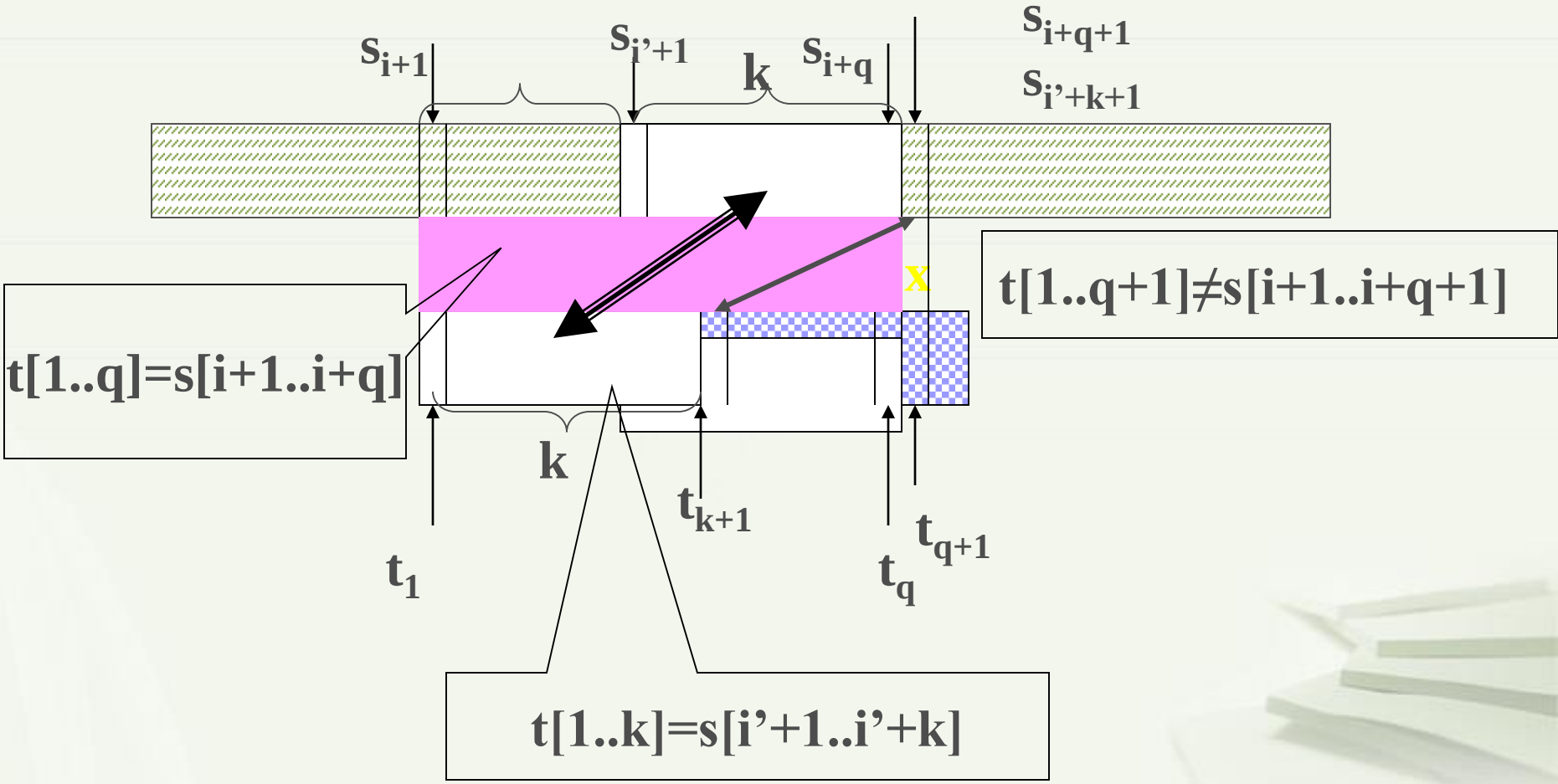


那么如何求 i’ 呢？？

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现



4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

假设 k=i+q-i’，所以，确定 i’ 就等价于确定 k,
即在{i+1,i+2,...i+q}中找最小的整数 i’使得
t[1..k]=s[i’+1...i’+k] 等价于在 {0,1,2,...,q-1}找最大的整数 k，使得t[1..k]=s[i’+1...i’+k]！！

从分析可知：t[1..k]是 t[1..q]的最长的既是真前缀又是真后缀的子串（即长度小于q）

所以，求 k 就是求 t[1..q] 中最长的既是真前缀又是真后缀的子串的长度。
k与主串无关！！

单选题 1分

设置

假设有主串为s=‘0000000010000110’,采用朴素的BF算法进行模式匹配。查找模式串t1=‘0011’和模式串t2=‘0101’,分别进行的比较次数是：

A 40，31

B 35，28

C 35，31

D 40，28

提交

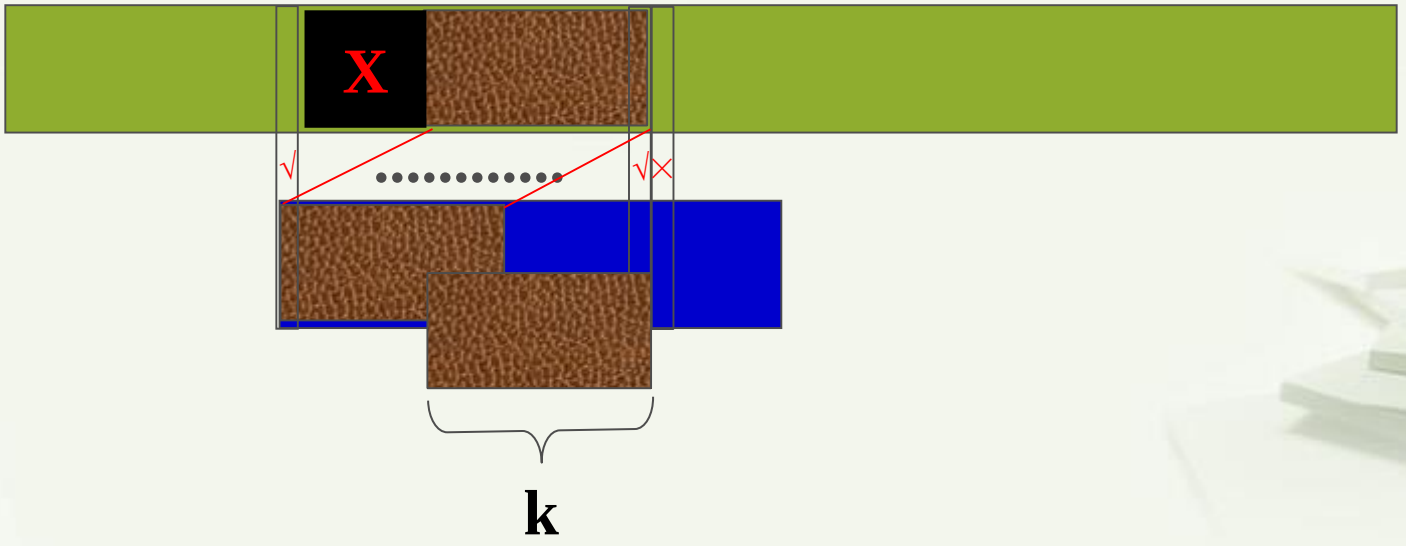
KMP

模式匹配之 KMP算法

- 基本思想：

是对BF算法的改进，利用“部分不匹配，则整体就不匹配”这一性质

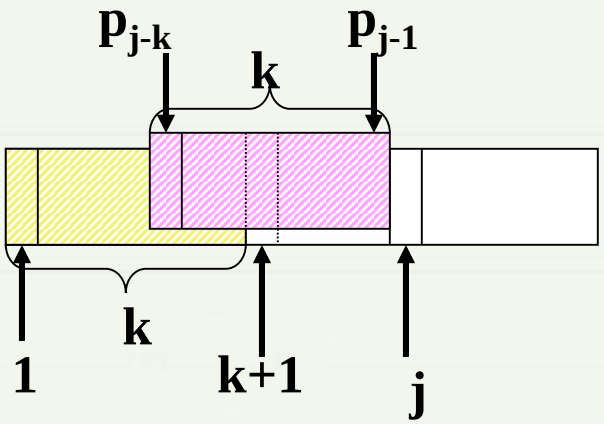
把不可能的匹配点直接跳过去，即找到部分匹配的新起点。



KMP

模式匹配之 KMP算法

- 关键： 对模式串，求每个字符的失败函数值



$$next[j]=\begin{cases} 0 & j=1 \\ Max\{k+1 \mid 1<k+1<j \quad `p_1p_2\cdots p_k' = `p_{j-k}\cdots p_{j-1}' \quad \text{非空} \\ 1 & \text{其他情况} \end{cases}$$

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

II、KMP算法

假设如何求 k 的问题已经解决，即对于每个t[j]都求出了最长的前缀串长度k,那么就有：与 t[j]匹配失败时，应继续与 t[k+1]比较， k+1称为t[j]的失败函数值,记作 next[j]=k+1 。

于是，KMP算法如下：

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

II、KMP算法

```
int    index_KMP(String  s, String t, int pos)
//返回子串t在主串s中第pos个字符之后的位置，
//若不存在返回0
  {   i=pos;  j=1;
       while    ( i<=s.curlen )&&(j<=t.curlen)
           if  (j=0||s.ch[i]==t.ch [j])
              {  i++;   j++ ; }
           else  {   j=next[j]; }
        if  ( j>t.curlen ) return  (i-t.curlen);
           else   return (0);                    //匹配失败
  }
```

4.2.2 字符串ADT的实现

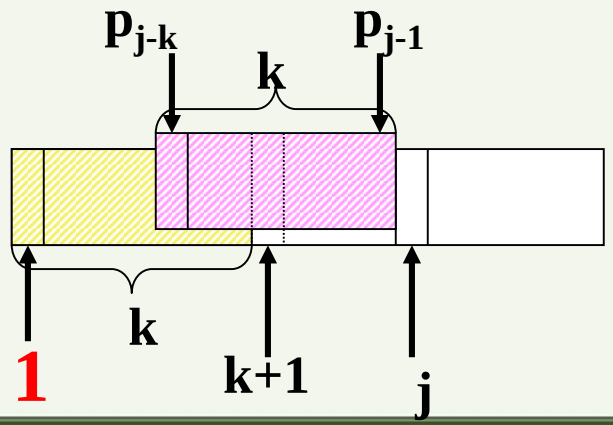
2. 字符串的操作的实现

III、失败函数的计算

根据前面的分析，假设模式串为 $p=‘p_1p_2\cdots p_m’$

失败函数可以定义为：

$$next[j]=\begin{cases}0 & j=1\\ Max\{k+1 \mid 1<k+1<j \ \ ‘p_1p_2\cdots p_k’=‘p_{j-k}\cdots p_{j-1}’\} & \text{非空}\\ 1 & \text{其他情况}\end{cases}$$



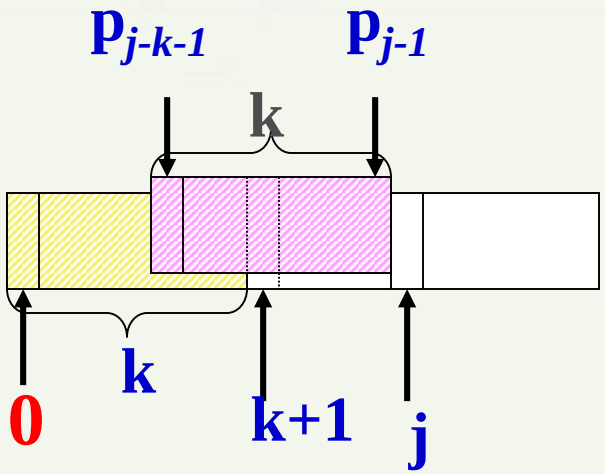
4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

教材上的定义，最大长度前缀串的长度为k,失败函数为：

$$next[j]=\begin{cases}-1 & \text{当 } j=0 \text{ 时}\\ Max\{k+1 \mid 0\le k<j-1 \text{ 且 } `p_0p_2\cdots p_k`=`p_{j-k-1}\cdots p_{j-1}`\} & \text{非空}\\ 0 & \text{其他情况}\end{cases}$$



4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

计算方法：

方法1：根据定义计算，即求最长的既是真前缀又是真后缀的子串的长度k,则next[j]=k+1。

方法2：利用递推方式

已知 next[1]=0　　或者 next[0]=-1

假设 next[j]=k

求 next[j+1]=?

因为next[j]=k 所以有

$$‘p_1p_2\cdots p_{k-1}’=‘p_{j-k+1}\cdots p_{j-1}’$$

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

若 $p_k=p_j$ 则有

‘$p_1p_2\cdots p_{k-1}p_k$’= ‘$p_{j-k+1}\cdots p_{j-1}p_j$’

于是 next[j+1]=k+1= next[j]+1

否则，即 $p_k<>p_j$ 那么运用KMP思想，p_j应与谁匹配？ 显然应该是 p_k，k’=next[k]

同样：

若 $p_{k'}=p_j$ 则有

‘$p_1p_2\cdots p_{k'-1}p_{k'}$’ = ‘$p_{j-k'+1}\cdots p_{j-1}p_j$’

于是 next[j+1]=k’+1=next[k]+1

否则，即 $p_{k'}<>p_j$ 那么运用KMP思想，p_j应与谁匹配？ 显然应该是 $p_{k''}$，k”=next[k’]

4.2.2 字符串ADT的实现 按定义：

2. 字符串的操作的实现

依此类推下去，直到：

p_j 与某个匹配成功 则 next[j+1]=k"… +1

或 不存在满足条件的 k"…(1<k"…<j)，

则 next[j+1]=1

例如：p='abaabcac'

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

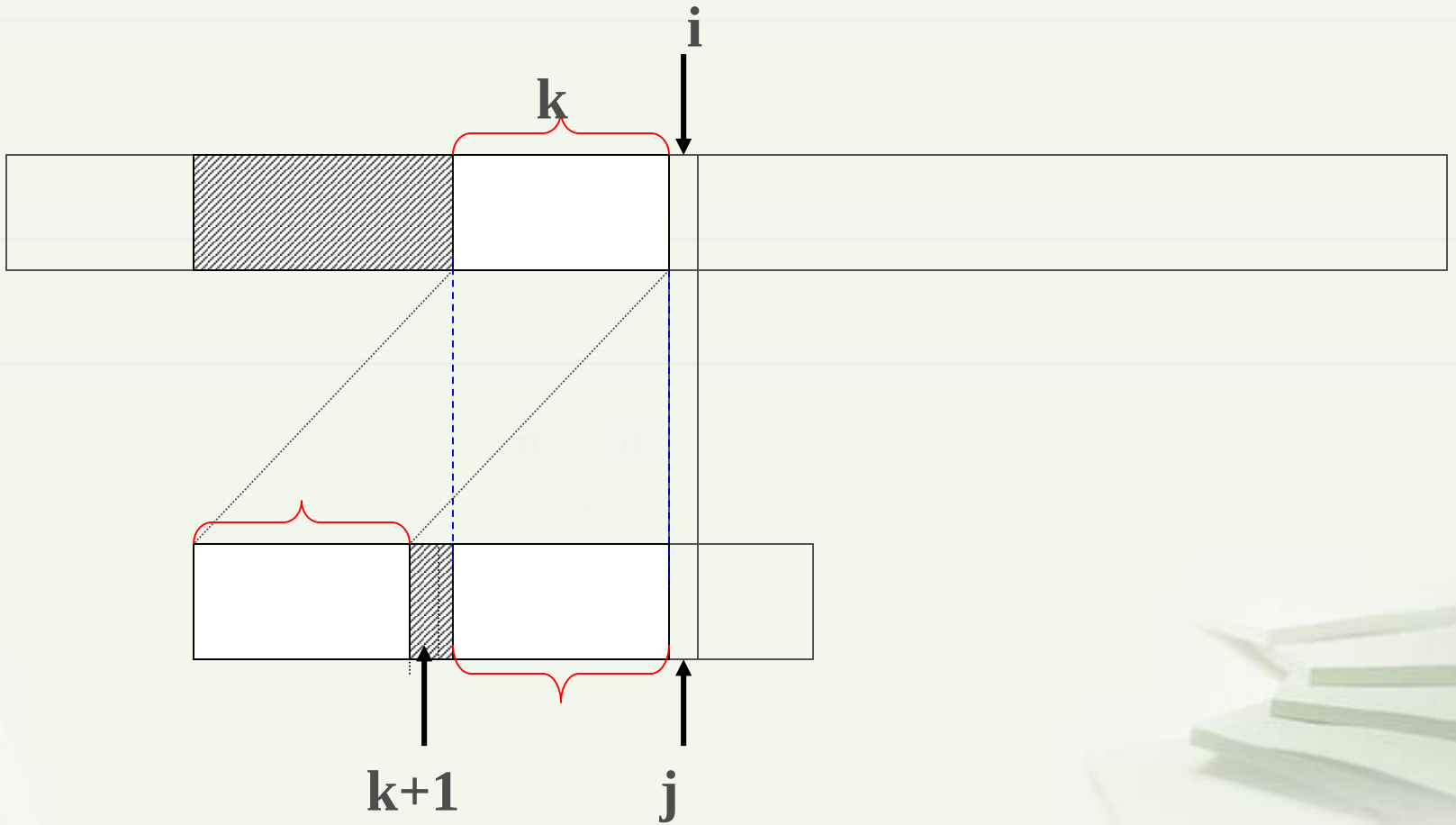
或

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

4.2 字符串

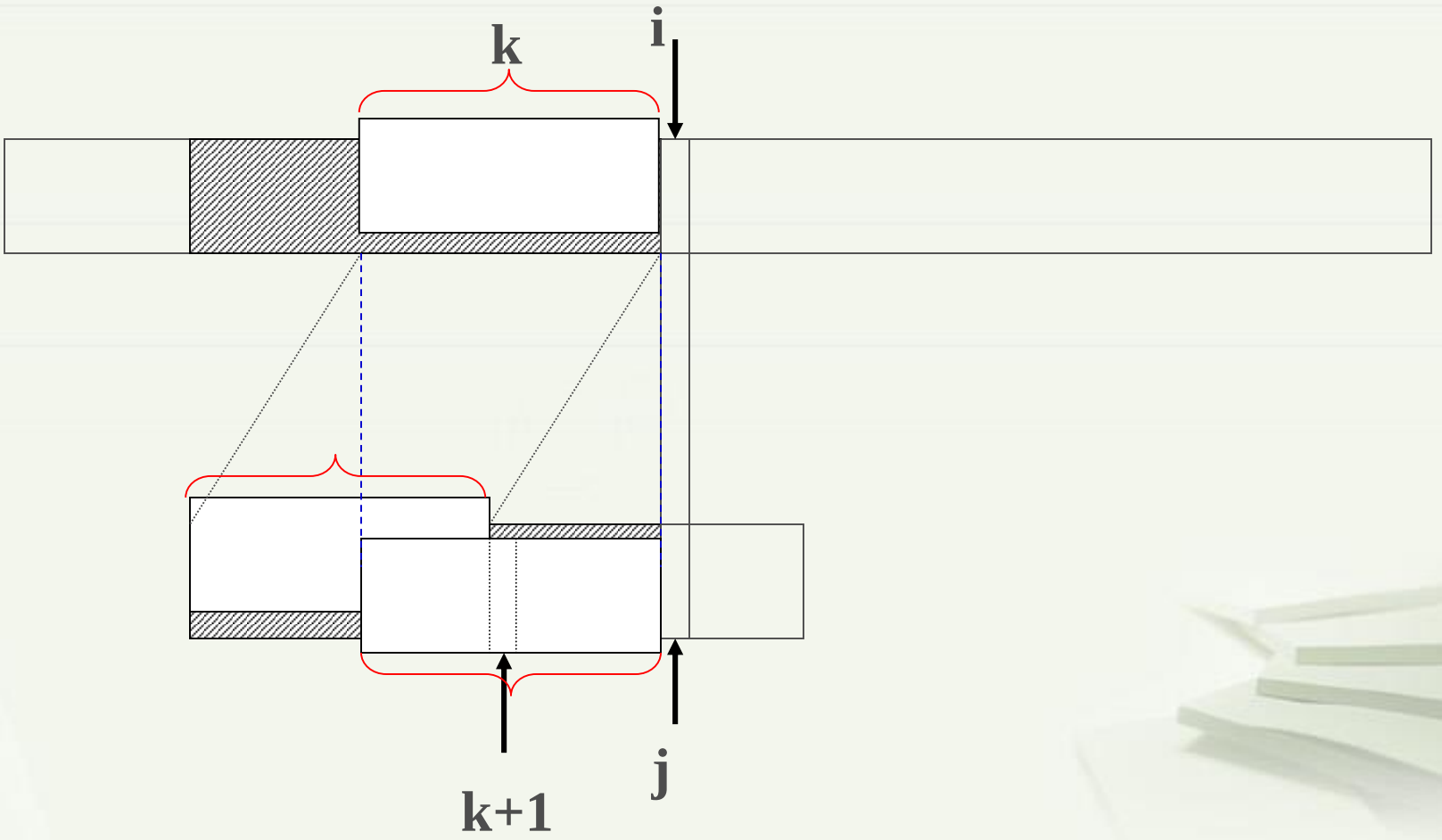
4.2.2 字符串ADT的实现

2. 字符串的操作的实现



4.2.2 字符串ADT的实现

2. 字符串的操作的实现



4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

求NEXT的算法：

```
void  get_next(String t, int &next[])
 //求模式串t的NEXT函数值并存入数组next
 {   j=1;   next[1]=0;  k=0;
     while(j<t.curlen)
       if(k==0)||(t.ch[j]==t.ch[k])
         { j++; k++; next[j]=k;  }
       else  k=next[k]
   }
```

为什么？

时间复杂度 O（m）

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

注意两点：

A. KMP算法的特点：效率较高；不回溯，对主串仅需要从头到尾扫描一遍（可以边读入边匹配）。

B. 改进NEXT值：　next[j]=k　即 s_i 与 p_j 比较不相等时，继续与 p_k 比较，但是若 $p_k=p_j$ 显然肯定还要失败，所以 next[j] 可以改进为 next[k] .....

算法如下：

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

```
void  get_nextval(String  t, int &nextval[])
//求模式串t的NEXT函数修正值并存入数组nextval
{   j=1;   nextval[1]=0;  k=0;
    while(j<t.curlen)
      if(k==0)||(t.ch[j]==t.ch[k])
        { j++; k++;
          if(t.ch[j]!=t.ch[k]) nextval[j]=k;
          else nextval[j]=nextval[k]  }
      else  k=nextval[k]
 }
```

为什么？

时间复杂度 O（m）

4.2.2 字符串ADT的实现

KMP算法进行模式匹配举例：

对于给定的主串s='aabcabaabbaabacaa'，模式串， p='aaba'
写出KMP算法匹配的过程，统计共进行了多少次比较。

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7
s_i:	a	a	b	c	a	b	a	a	b	b	a	a	b	a	c	a	a

	1	2	3	4
p_j:	a	a	b	a
next	0	1	2	1

$s_1=p_1$, $s_2=p_2$, $s_3=p_3$, $s_4\neq p_4$, next[4]=1

$s_4\neq p_1$, next[1]=0　　没有部分匹配的！

$s_5=p_1$, $s_6\neq p_2$, next[2]=1

$s_6\neq p_1$, next[1]=0　　没有部分匹配的！

$s_7=p_1$, $s_8=p_2$, $s_9=p_3$, $s_{10}\neq p_4$, next[4]=1

$s_{10}\neq p_1$, next[1]=0　　没有部分匹配的！

$s_{11}=p_1$, $s_{12}=p_2$, $s_{13}=p_3$, $s_{14}=p_4$

4+1+2+1+4+1+4=17

按BF算法，比较多少次？

对下面的主串和模式呢？
s='aaaaaaaaaabaa'
p='aaab'

4.2 字符串

4.2.2 字符串ADT的实现

2. 字符串的操作的实现

(5) 串的置换 Replace(s,t,v)

将串 s 中所有子串 t 的出现都置换为 v

在 s 中定位 t，然后根据 t 和 v 二者的的长度：

t.curlen<v.curlen : 后移动 v.curlen-t.curlen

t.curlen=v.curlen : 替换

t.curlen>v.curlen : 前移动 t.curlen-v.curlen

重复，直到 s 中不存在 t 为止。

(6) 串的插入 Insert(s,pos,t)

移动或用取子串、联结运算实现（有溢出问题）

(7) 串的删除 Delete(s,pos,len)

4.2 字符串

4.2.2 字符串ADT的实现

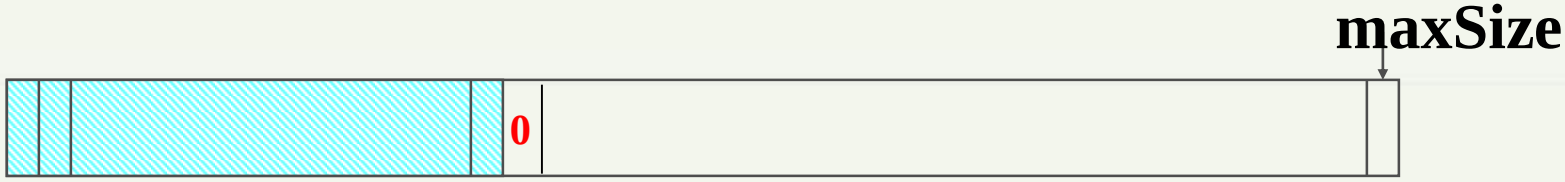
字符串ADT基于链式存储的实现

略

4.2 字符串

4.2.2 字符串ADT的实现

而实际上，在C语言中存储字符串并没有这么复杂，怎么存的？



用在字符串最后存放一个“空字符”-'\0'来表示串结束。

因此，所有string.h中的函数在使用时，要求必须是以串形式存储的字符数组，不能是一般的字符数组。
在字符串操作函数中，使用一般的字符数组，会有什么后果？

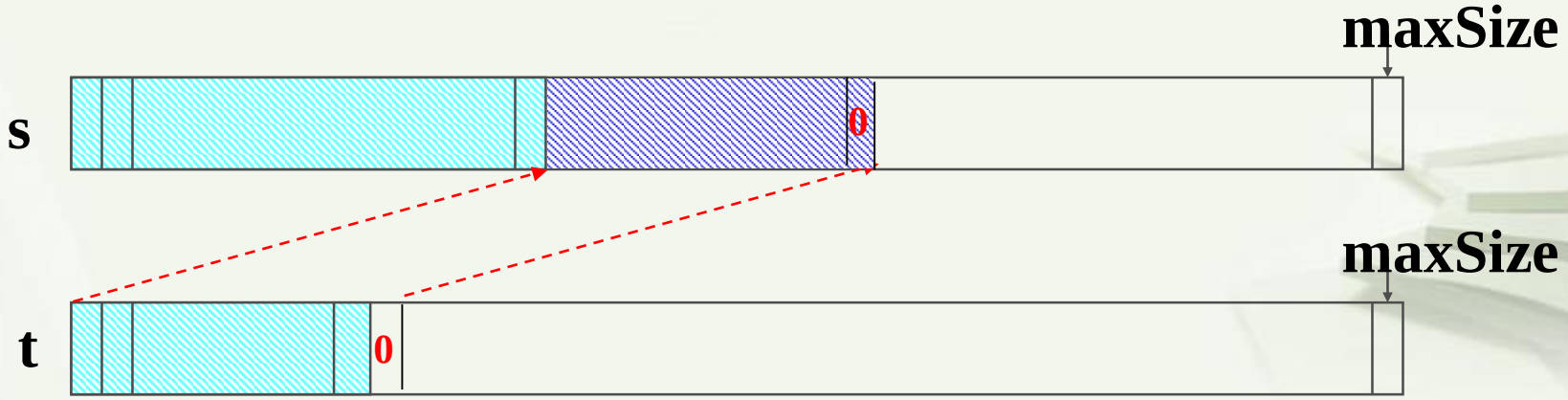
4.2 字符串

4.2.2 字符串ADT的实现

gets(s): s是字符数组，在输入有效串字符后，再存入一个’\0’

puts(s): s必须是字符串形式存储的数组，即必须保证串的最后存储着’\0’。

strcat(s,t): s,t必须是字符串形式存储的数组，即必须保证每个串的最后存储着’\0’。



4.3 广义表

略

本章小结

重点和难点：

1.	数组逻辑结构的定义：相对于一般线性表，它特殊在哪里？
	■下标的含义、维数的含义
	■ 数组数据结构的形式化定义；
2.	**数组ADT的定义**
	■ 为什么数组数据结构上定义的操作很少？
3.	**数组ADT的实现**
	■ 存储结构：顺序存储；如何存储的？受多线性关系约束的数据元素如何顺序存储到一个物理位置（多个线性关系的优先顺序：“行主序”、“列主序”）。
	■ 存储的特点：随机存取（寻址公式的推导）

本章小结

重点和难点：

4.	特殊矩阵ADT的实现
	■ 特殊矩阵如何存储？——目的是节省空间；
	■ 逻辑位置(I,j) 与物理存储位置k的映射关系；
5.	串逻辑结构的特点：相对于一般线性表，它特殊在哪里？
	■ 串操作的对象；串的重要操作；
6.	串ADT的定义
7.	串ADT的实现
	■ 存储结构：顺序存储；如何存储的？
	■ 重要串操作的实现：赋值、定位、取子串、串联结、求串长度

实验 5

1. 问题描述：基于十字链表存储的稀疏矩阵ADT实现

在现实应用中，一些规模很大的特殊矩阵具有重要的地位，稀疏矩阵就是其中一类。这类矩阵直接采用二维数组类型存储时，其操作实现都比较简单，但是空间利用率很低。为了节省空间，可以只存储非零元素，即把稀疏矩阵抽象为非零元素的三元组的线性表，而线性表可以采用顺序存储，也可以采用链式存储，但是很显然，矩阵的操作实现变复杂了！

假设稀疏矩阵采用三元组形式的“压缩”存储（顺序或十字链式），实现矩阵ADT。

实验5

2. 要求

（1）输入包括：矩阵的行数、列数、非零元素个数，各个非零元素的行、列、值

（2）实现创建、输出操作

（3）实现矩阵的加运算、转置、乘运算。

（4）输出可以是简单的三元组形式，也可以是人们喜欢的一般矩阵的格式。


END