第4章　串、数组和广义表

王迪

wangd@sdas.org

- 第2章　线性表
- 第3章　栈和队列
- 第4章　串、数组和广义表

线性结构

可表示为：（a_1，a_2，……，a_n）

教学内容

4.1　串

4.2　数组

4.3　广义表

教学目标

1. 了解串的存储方法，理解串的两种模式匹配算法，重点掌握BF算法。
2. 明确数组和广义表这两种数据结构的特点，掌握数组地址计算方法，了解几种特殊矩阵的压缩存储方法。
3. 掌握广义表的定义、性质及其GetHead和GetTail的操作。

4.1 串的定义

串(String)-----零个或多个字符组成的有限序列

$$s = 'a_1 a_2 \cdots a_n'$$

串名

串值

串长 → n

空串 ← n=0

a=‘BEI’,

b=‘JING’

c=‘BEIJING’

d=‘BEI JING’

子串

主串

字符位置

子串位置

串相等

空格串

4.2 案例引入

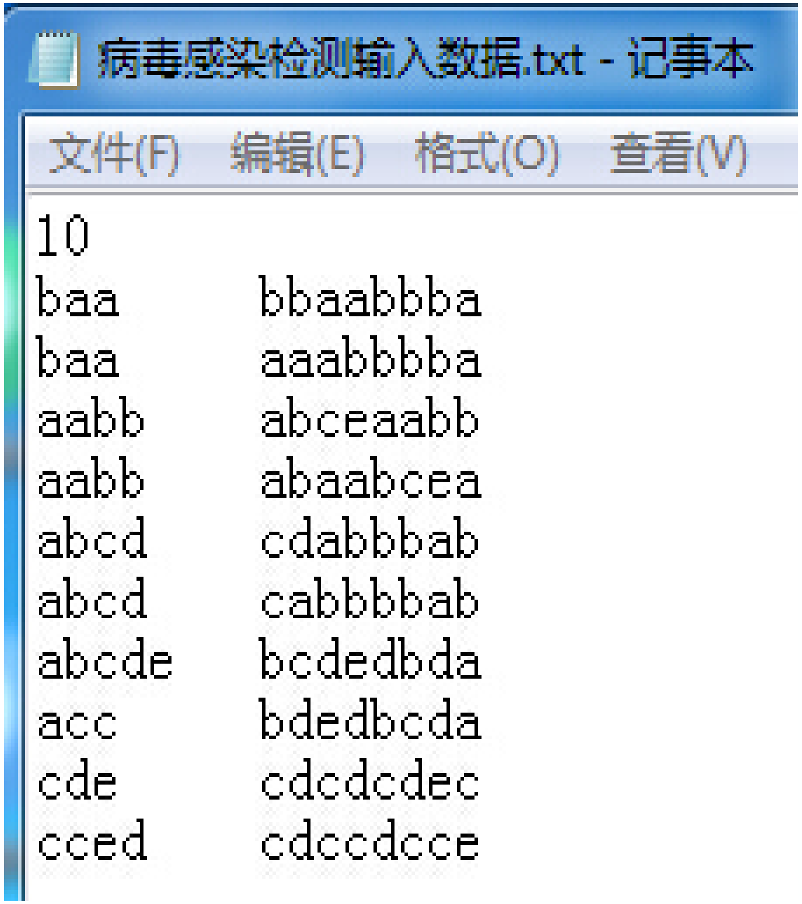
案例4.1 ：病毒感染检测

研究者将人的DNA和病毒DNA均表示成由一些字母组成的字符串序列。

然后检测某种病毒DNA序列是否在患者的DNA序列中出现过，如果出现过，则此人感染了该病毒，否则没有感染。

例如，假设病毒的DNA序列为baa，患者1的DNA序列为aaabbba，则感染，患者2的DNA序列为babbba，则未感染。

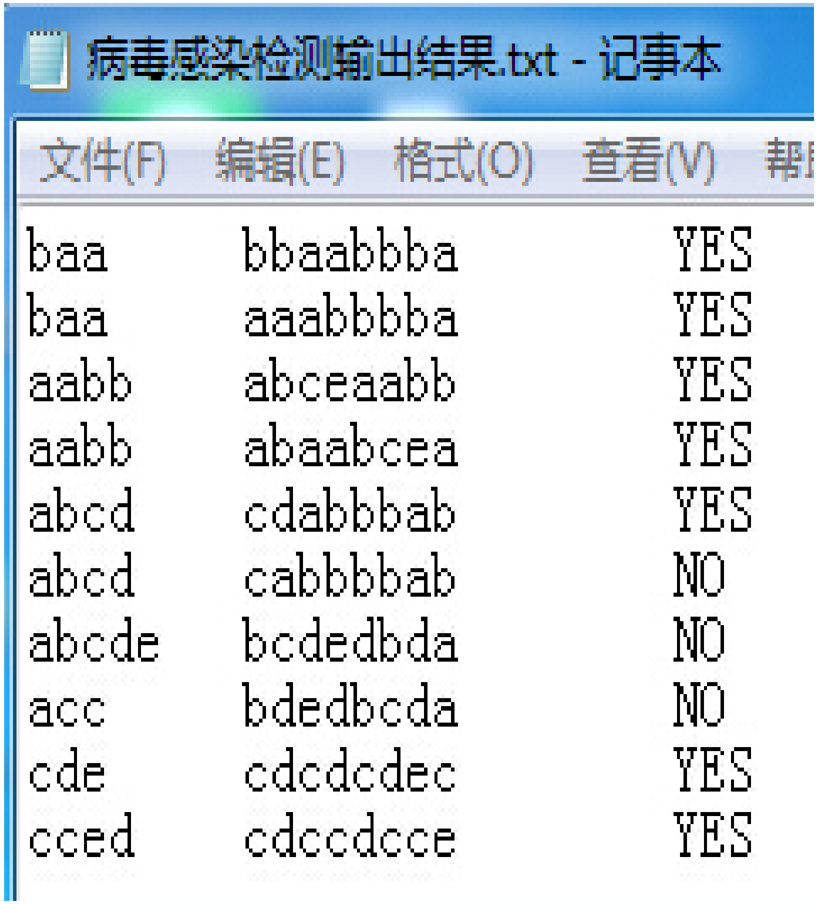
（注意，人的DNA序列是线性的，而病毒的DNA序列是环状的）



病毒感染检测输入数据.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V)

10	
baa	bbaabbba
baa	aaabbbba
aabb	abceaabb
aabb	abaabcea
abcd	cdabbbab
abcd	cabbbbab
abcde	bcdedbda
acc	bdedbcda
cde	cdcdcdec
cced	cdccdcce



病毒感染检测输出结果.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮

baa	bbaabbba	YES
baa	aaabbbba	YES
aabb	abceaabb	YES
aabb	abaabcea	YES
abcd	cdabbbab	YES
abcd	cabbbbab	NO
abcde	bcdedbda	NO
acc	bdedbcda	NO
cde	cdcdcdec	YES
cced	cdccdcce	YES

4.3 串的类型定义、存储结构及运算

ADT String {

数据对象：$D = \{a_i \mid a_i \in CharacterSet, i = 1,2,\cdots,n,\ \mathrm{n} \geq 0\}$

数据关系：$R_1 = \{< a_{i-1}, a_i > \mid a_{i-1}, a_i \in D, i = 1,2,\cdots,n\}$

基本操作：

(1) StrAssign (&T,chars) //串赋值

(2) StrCompare (S,T) //串比较

(3) StrLength (S) //求串长

(4) Concat(&T,S1,S2) //串联

(5) SubString(&Sub,S,pos,len)   //求子串

(6) StrCopy(&T,S)   //串拷贝

(7) StrEmpty(S)   //串判空

(8) ClearString (&S)   //清空串

(9) Index(S,T,pos)   //子串的位置

(11) Replace(&S,T,V)   //串替换

(12) StrInsert(&S,pos,T)   //子串插入

(12) StrDelete(&S,pos,len)   //子串删除

(13) DestroyString(&S)   //串销毁

}ADT String

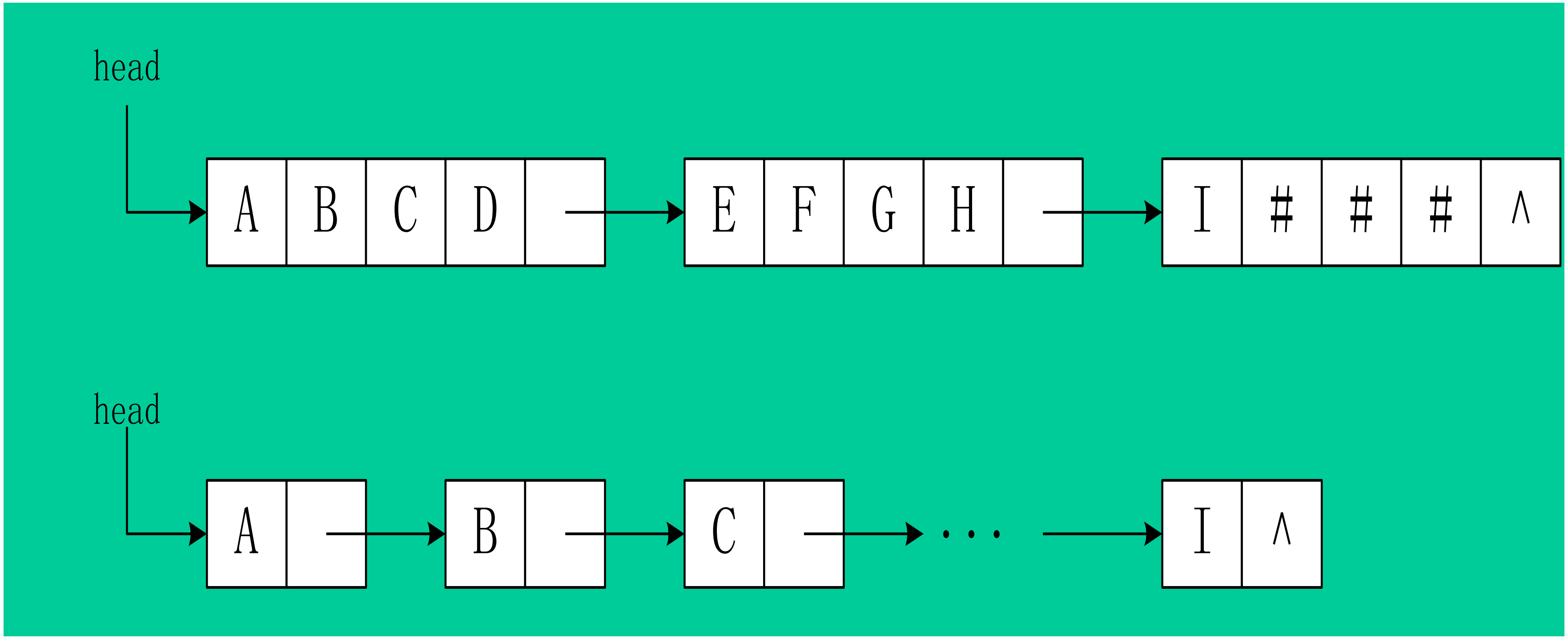
串的存储结构

- 顺序存储
- 链式存储

顺序存储表示

```
typedef struct{
    char *ch;       //若串非空,则按串长分配存储区,
                    //否则ch为NULL
    int  length;    //串长度
}HString;
```

链式存储表示



链式存储表示

```
#define CHUNKSIZE 80        //可由用户定义的块大小
typedef struct Chunk{
   char  ch[CHUNKSIZE];
   struct Chunk *next;
}Chunk;

typedef struct{
   Chunk *head,*tail;      //串的头指针和尾指针
   int curlen;             //串的当前长度
}LString;
```

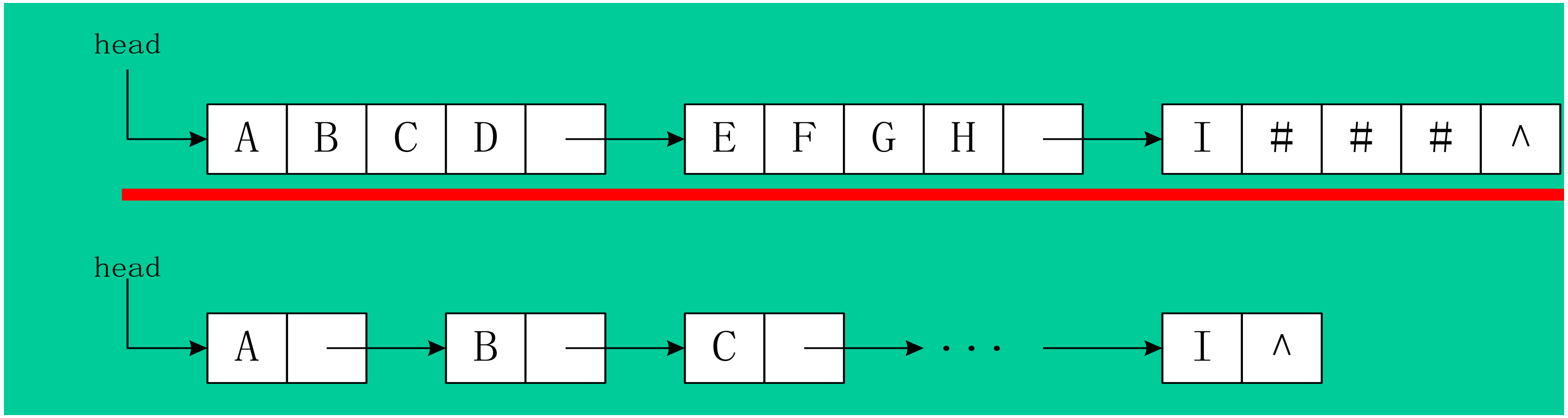
链式存储表示

优点：操作方便

缺点：存储密度较低

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

可将多个字符存放在一个结点中，以克服其缺点



串的模式匹配算法

算法目的：

确定主串中所含子串第一次出现的位置（定位）

算法种类：

- BF算法（又称古典的、经典的、朴素的、穷举的）
- KMP算法（特点：速度快）

BF算法设计思想

i

S : a b a b c a b c a c b a b

T : a b c

j

i指针回溯

S : a b a b c a b c a c b a b

T : a b c

S : a b a b c a b c a c b a b

T : a b c

BF算法设计思想

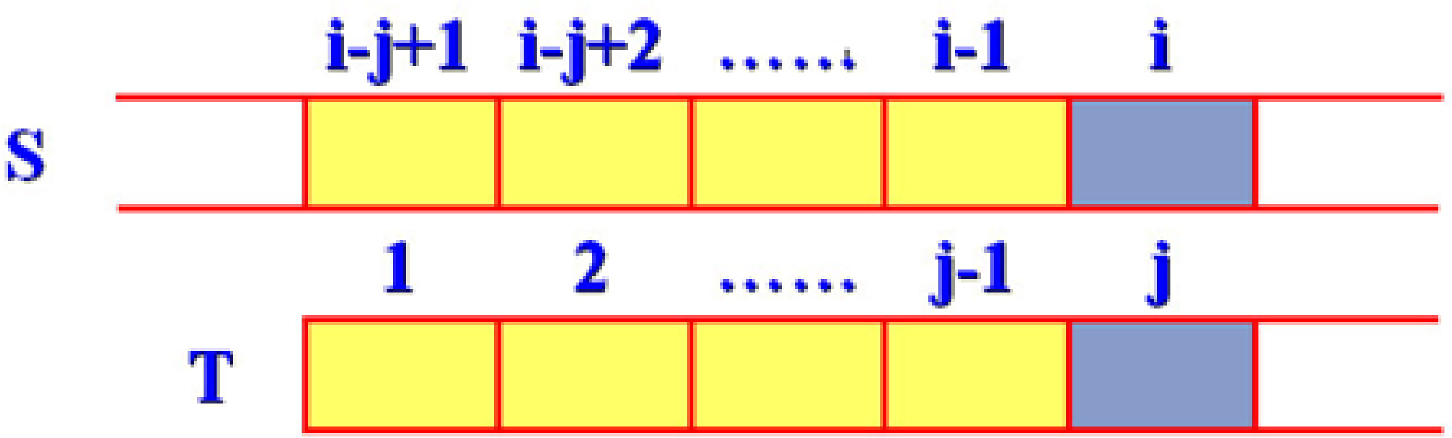
Index(S,T,pos)

- 将主串的第pos个字符和模式的第一个字符比较，
  若相等，继续逐个比较后续字符；
  若不等，从主串的下一字符起，重新与模式的第一个字符比较。

- 直到主串的一个连续子串字符序列与模式相等 。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 0

BF算法描述（算法4. 1）

```
int  Index(Sstring S,Sstring T,int pos){
    i=pos;   j=1;
   while (i<=S[ 0 ] && j <=T[ 0 ]){
      if ( S[ i ]=T[ j ]) {++i;  ++j; }
     else{ i=i-j+2;    j=1; }
   if ( j>T[ 0 ])   return   i－T[0];
   else return 0;
}
```



BF算法时间复杂度

例：　S='0000000001'，T='0001'，pos=1

若n为主串长度，m为子串长度，最坏情况是

- ✓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位各比较了m次
- ✓最后m位也各比较了1次

总次数为：$(n-m)*m+m = (n-m+1)*m$

若m<<n，则算法复杂度$O(n*m)$

KMP（Knuth Morris Pratt）算法

《计算机程序设计艺术 第1卷 基本算法》
《计算机程序设计艺术 第2卷 半数值算法》
《计算机程序设计艺术 第3卷 排序与查找》

http://www-cs-faculty.sta

KMPKMP（Knuth Morris Pratt）算法

利用已经部分匹配的结果而加快模式串的滑动速度？
且主串S的指针i不必回溯！可提速到O(n+m)！

S=‘a b a b c a b c a c b a b’　　　S=‘a b a b c a b c a c b a b’

T=‘a b c a c’　　　　　　　　　　T=‘a b c a c’

S=‘a b a b c a b c a c b a b’

T=‘a b c a c’

第 1 次匹配　　s=abacaba　　**i=4**

p=abab　　**j=4**　　**失败**

p=abab　　**j=2**

因$p_1 \neq p_2$, $s_2=p_2$, 必有$s_2 \neq p_1$, 又因$p_1=p_3$, $s_3=p_3$, 所以必有$s_3=p_1$。因此, 第二次匹配可直接从i=4, j=2开始。

改进：每趟匹配过程中出现字符比较不等时，不回溯主指针i，利用已得到的“部分匹配”结果将模式向右滑动尽可能远的一段距离，继续进行比较。

$$
\begin{array}{cccccccccc}
s_1 & s_2 & s_3\cdots & s_{i-j+1} & s_{i-j+2} & \cdots & s_{i-2} & s_{i-1} & s_i & s_{i+1} \\
 & & & \| & \| & & \| & \| & \neq & \\
 & & & p_1 & p_2 & \cdots & p_{j-2} & p_{j-1} & p_j & p_{j+1} \\
 & & & & & \| & & \| & & \\
 & & & & & p_1 & \cdots & p_{k-1} & p_k & p_{k+1}
\end{array}
$$

① "$p_1p_2\ldots p_{k-1}$" = "$s_{i-k+1}s_{i-k+2}\ldots s_{i-1}$"

② "$p_{j-k+1}p_{j-k+2}\ldots p_{j-1}$" = "$s_{i-k+1}s_{i-k+2}\ldots s_{i-1}$" （部分匹配）

③ "$p_1p_2\ldots p_{k-1}$" = "$p_{j-k+1}p_{j-k+2}\ldots p_{j-1}$" (真子串）

为此,定义next[j]函数，表明当模式中第j个字符与主串中相应字符“失配”时，在模式中需重新和主串中该字符进行比较的字符的位置。

$$
next[j]=\begin{cases}
\max\{\, k \mid 1<k<j, \text{且 “} p_1 \ldots p_{k-1} \text{”=“} p_{j-k+1} \ldots p_{j-1} \text{”} \} & \text{当此集合非空时} \\
0 & \text{当 } j=1 \text{ 时} \\
1 & \text{其他情况}
\end{cases}
$$

- 如何求next函数值

1. next[1] = 0;表明主串从下一字符s_{i+1}起和模式串重新开始匹配。i = i+1; j = 1;

2. 设next[j] = k，则next[j+1] = ?

①若$p_k=p_j$，则有“$p_1…p_{k-1}p_k$”=“$p_{j-k+1}…p_{j-1}p_j$”，如果在

j+1发生不匹配，说明next[j+1] = k+1 = next[j]+1。

②若$p_k≠p_j$，可把求next值问题看成是一个模式匹配问题，整个模式串既是主串，又是子串。

$$
\begin{array}{llllllll}
p_1 & p_2 \cdots p_{j-k+1} \cdots p_{j-1} & & p_j & p_{j+1} & & \text{next}[j]=k \\
 & \quad \| \qquad\qquad \| & & \neq & & & \\
 & p_1 \ \cdots \ p_{k-1} & & p_k & p_{k+1} & & \text{next}[k]=k' \\
 & \quad p_1 \cdots\cdots & & p_{k'} & p_{k'+1} & & \text{next}[k']=k'' \\
 & \qquad p_1 \cdots & & p_{k''} & p_{k''+1} & & \text{next}[k''']=k'''
\end{array}
$$

- 若 $p_{k'}=p_j$，则有“$p_1 \ldots p_{k'}$”=“$p_{j-k'+1} \ldots p_j$”，
  $\text{next}[j+1]=k'+1=\text{next}[k]+1=\text{next}[\text{next}[j]]+1.$
- 若 $p_{k''}=p_j$，则有“$p_1 \ldots p_{k''}$”=“$p_{j-k''+1} \ldots p_j$”，
  $\text{next}[j+1]=k''+1=\text{next}[k']+1=\text{next}[\text{next}[k]]+1.$
- $\text{next}[j+1]=1.$

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
模式串	a	b	c	a	a	b	b	c	a	b	c	a	a	b	d	a	b
next[j]	0	1	1	1	2	2	3	1	1	2	3	4	5	6	7	1	2

- void get_next(SString T, int &next[])

```
{
    i= 1; next[1] = 0; j = 0;
    while( i<T[0]){
        if(j==0 || T[i] == T[j]){
            ++i; ++j;
            next[i] = j;
        }
        else
            j = next[j];
    }
}
```

```
int Index_KMP (SString S,SString T, int pos)
{
    i= pos,j =1;
    while (i<S[0] && j<T[0]) {
        if (j==0 || S[i]==T[j]) {   i++;j++;  }
        else
            j=next[j];        /*i不变,j后退*/
    }
    if (j>T[0])  return i-T[0];  /*匹配成功*/
    else   return 0;              /*返回不匹配标志*/
}
```

- KMP算法的时间复杂度

设主串s的长度为n，模式串t长度为m，在KMP算法中求next数组的时间复杂度为O(m)，在后面的匹配中因主串s的下标不减即不回溯，比较次数可记为n，所以KMP算法总的时间复杂度为O(n+m)。

next函数的改进

j	1 2 3 4 5
模式	a a a a b
next[j]	0 1 2 3 4
nextval[j]	0 0 0 0 4

i=4

a a a b a a a a b

a a a a

j=4

① a a a

j=3

② a a

j=2

③ a

j=1

a a a a b

i = 5; j = 1

next[j] = k，而p_j=p_k，则 主串中s_i和p_j不等时，不需再和p_k进行比较，而直接和$p_{next[k]}$进行比较。

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
模式串	a	b	c	a	a	b	b	c	a	b	c	a	a	b	d	a	b
next[j]	0	1	1	1	2	2	3	1	1	2	3	4	5	6	7	1	2
nextval[j]	0	1	1	0	2	1	3	1	0	1	1	0	2	1	7	0	1

- void get_nextval(SString T, int &nextval[])

```
{
    i= 1; nextval[1] = 0; j = 0;
    while( i<T[0]){
        if(j==0 || T[i] == T[j]){
            ++i; ++j;
            if(T[i] != T[j]) nextval[i] = j;
            else  nextval[i] = nextval[j];
        }
        else  j = nextval[j];
    }
}
```

next[i] = j;

4.4 数组

数组是由一组个数固定，类型相同的数据元素组成的阵列。

一维数组：线性表中的数据元素为非结构的简单元素。是定长的线性表。

以二维数组为例：

二维数组中的每个元素都受两个线性关系的约束，即行关系和列关系。

在每个关系中，每个元素都有且仅有一个直接前驱，有且只有一个直接后继。

数组的抽象数据类型

ADT Array {

数据对象:

$$j_i = 0, \cdots b_i - 1, \mathrm{i} = 1, 2, \cdots, \mathrm{n}$$

$$D = \{\mathrm{a}_{\mathrm{j}\,\mathrm{j}_2 \cdots \mathrm{j}_\mathrm{n}} \mid \mathrm{a}_{\mathrm{j}_1 \mathrm{j}_2 \cdots \mathrm{j}_\mathrm{n}} \in ElemSet$$

数据关系:

$$R_1 = \{< a_{j_1 \cdots j_i \cdots j_n}, a_{j_1 \cdots j_i + 1 \cdots j_n} >|$$

$$0 \le \mathrm{j}_\mathrm{k} \le b_k - 1,\ \ 1 \le \mathrm{k} \le \mathrm{n},\ \text{且}\,\mathrm{k} \ne \mathrm{i},$$

$$0 \le \mathrm{j}_\mathrm{i} \le b_k - 2,$$

$$a_{j_1 \cdots j_i \cdots j_n}, a_{j_1 \cdots j_i + 1 \cdots j_n} \in D, i = 2, \cdots, n\}$$

基本操作:

(1) InitArray (&A,n,bound1, …boundn)

//构造数组A

(2) DestroyArray (&A) **// 销毁数组A**

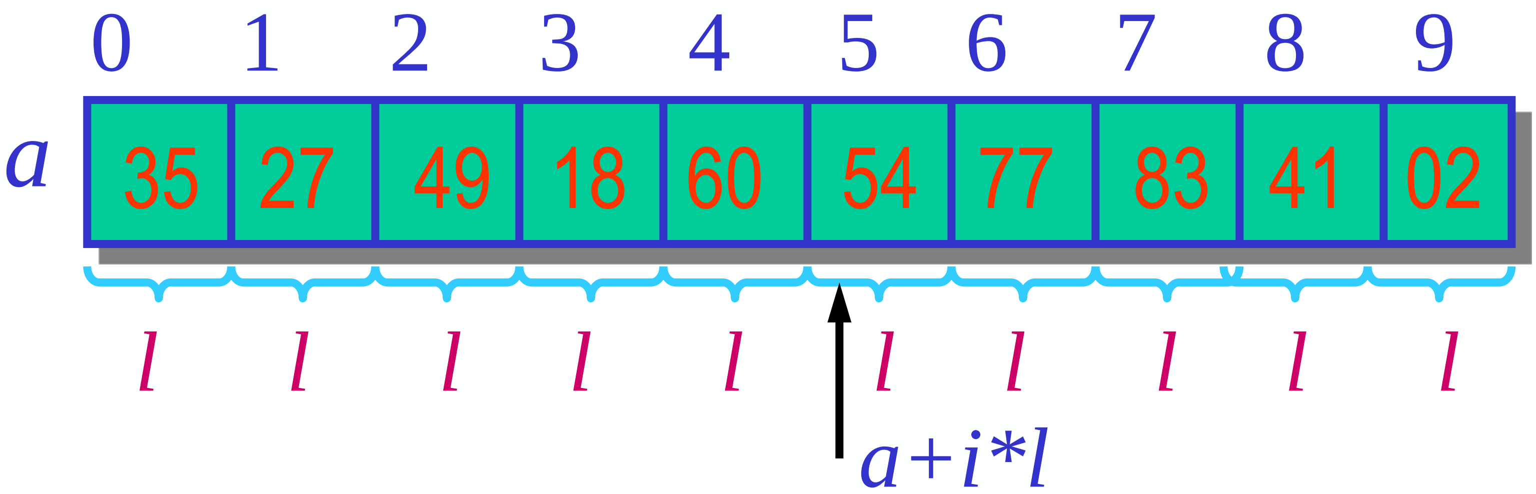
(3) Value(A,&e,index1,…,indexn) **//取数组元素值**

(4) Assign (A,&e,index1,…,indexn) **//给数组元素赋值**

}ADT Array

一维数组

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



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

二维数组

$A=(\alpha_1,\alpha_2,\cdots,\alpha_p)\quad (\mathrm{p}=\mathrm{m}或\mathrm{n})$

$$A_{m\times n}=\begin{bmatrix} [a_{11} & a_{12} & \cdots & a_{1n}] \\ [a_{21} & a_{22} & \cdots & a_{2n}] \\ \vdots & \vdots & & \vdots \\ [a_{m1} & a_{m2} & \cdots & a_{mn}] \end{bmatrix}$$

$\alpha_i=(a_{i1},a_{i2},\cdots,a_{in})\quad 1\le i\le m$

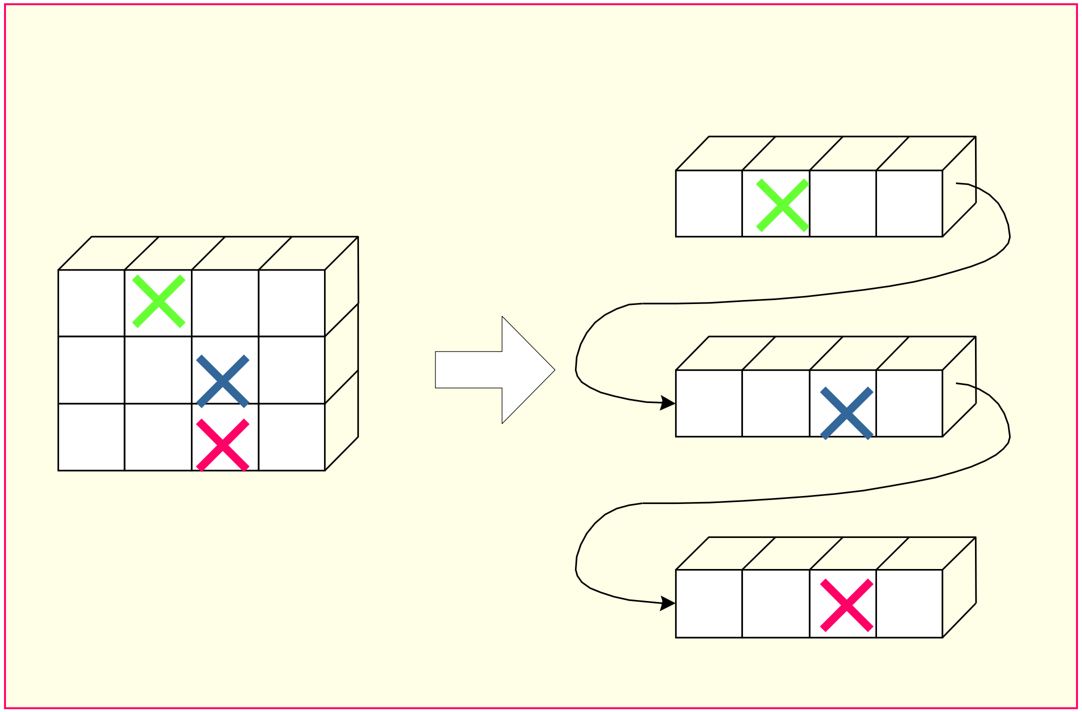
$\alpha_j=(a_{1j},a_{2j},\cdots,a_{mj})\quad 1\le \mathrm{j}\le n$

$$A_{m\times n}=\begin{bmatrix} \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} & \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} & \begin{matrix} \cdots \\ \cdots \\ \\ \cdots \end{matrix} & \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} \end{bmatrix}$$

数组的顺序存储

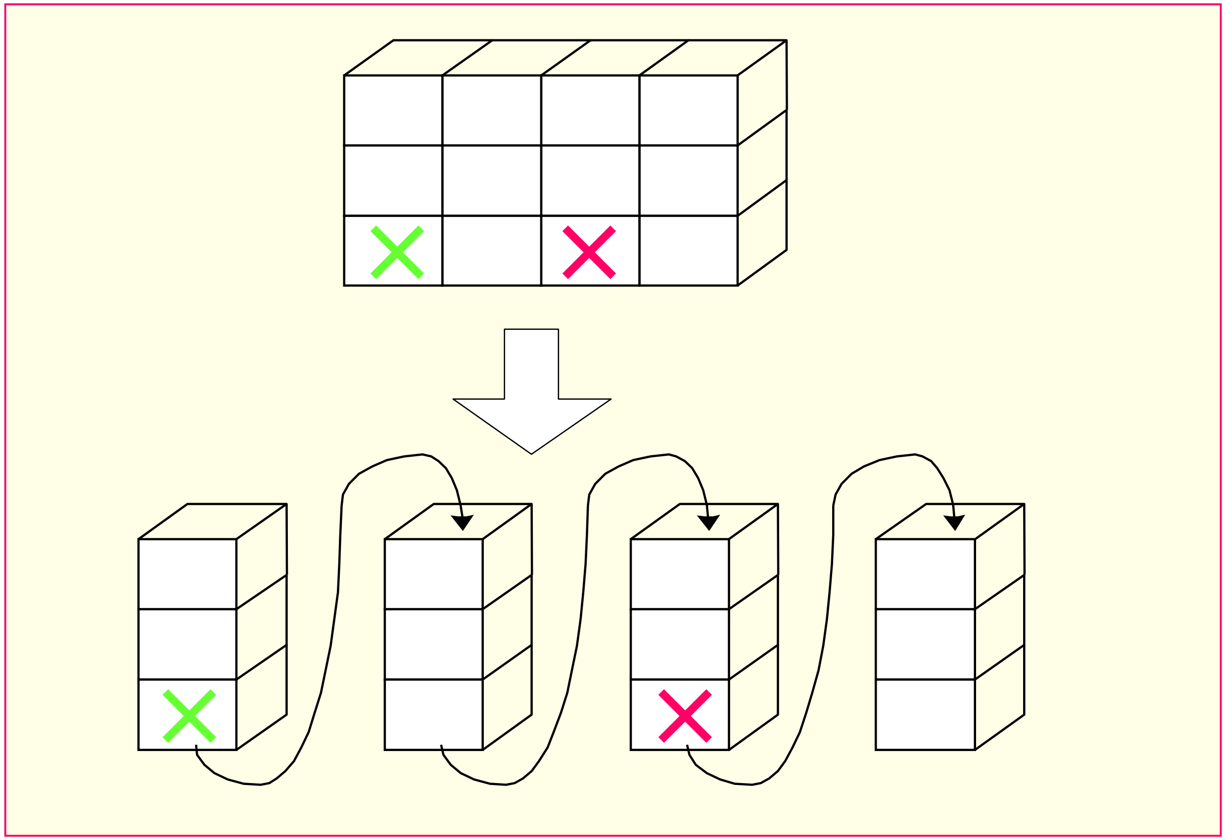
- 以行序为主序

  C, PASCAL



- 以列序为主序

FORTRAN



二维数组的行序优先表示

a[n][m]

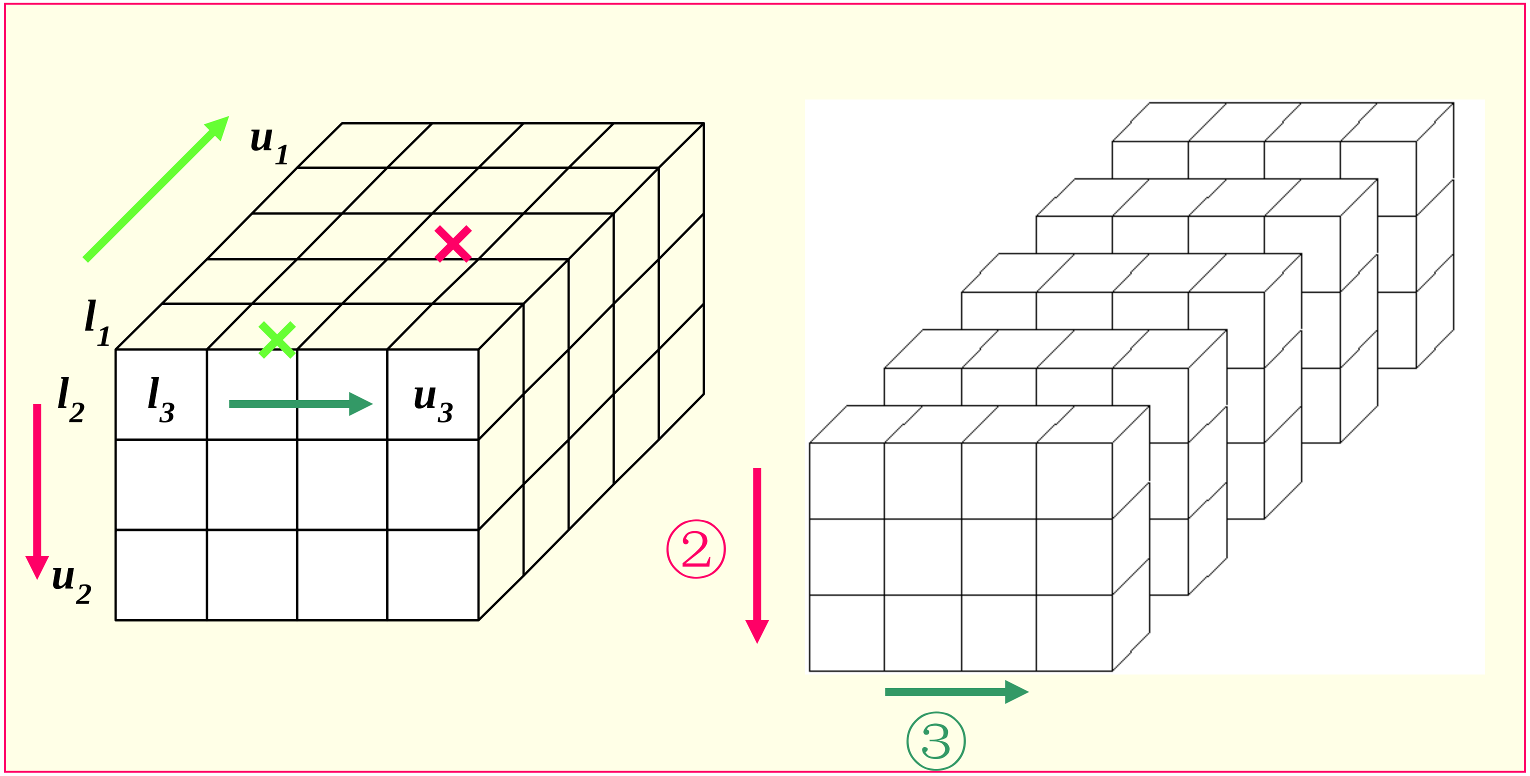
$$\boldsymbol{a}=\begin{pmatrix} a[0][0] & a[0][1] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & \cdots & a[2][m-1] \\ \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & \cdots & a[n-1][m-1] \end{pmatrix}$$

设数组开始存放位置 LOC(0, 0) = a

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

三维数组

按页/行/列存放，页优先的顺序存储



三维数组

☞ **a[m1][m2] [m3]** 各维元素个数为 m_1, m_2, m_3

☞ 下标为 i_1, i_2, i_3的数组元素的存储位置：

$$\mathrm{LOC}(i_1, i_2, i_3) = a + \underbrace{i_1 * m_2 * m_3}_{\text{前}i_1\text{页总元素个数}} + \underbrace{i_2 * m_3}_{\text{第}i_1\text{页的前}i_2\text{行总元素个数}} + i_3$$

第 i_2 行前 i_3 列元素个数

n维数组

☞ 各维元素个数为 $m_1, m_2, m_3, \ldots, m_n$

☞ 下标为 $i_1, i_2, i_3, \ldots, i_n$ 的数组元素的存储位置：

$$LOC(i_1, i_2, \cdots, i_n) = \mathrm{a} + i_1 * m_2 * m_3 * \cdots * m_n +$$

$$+ i_2 * m_3 * m_4 * \cdots * m_n + \cdots\cdots + i_{n-1} * m_n + i_n$$

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

n维数组

$$LOC[j_1, j_2, \cdots, j_n] = LOC[0,0,\cdots,0] + \left(\sum_{i=1}^{n} c_i j_i\right) L$$

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

练习

设有一个二维数组$A[m][n]$按行优先顺序存储，假设$A[0][0]$存放位置在$644_{(10)}$，$A[2][2]$存放位置在$676_{(10)}$，每个元素占一个空间，问$A[3][3]_{(10)}$存放在什么位置？脚注$_{(10)}$表示用10进制表示。

设数组元素A[i][j]存放在起始地址为Loc (i, j) 的存储单元中

$\because$ Loc (2, 2) = Loc (0, 0) + 2 * n + 2 = 644 + 2 * n + 2 = 676.

$\therefore$ n = (676 - 2 - 644) / 2 = 15

$\therefore$ Loc (3, 3) = Loc (0, 0) + 3 * 15 + 3 = 644 + 45 + 3 = 692.

练习

设有二维数组A[10,20]，其每个元素占两个字节，A[0][0]存储地址为100，若按行优先顺序存储，则元素A[6,6]的存储地址为______ 352 按列优先顺序存储，元素A[6,6]的存储地址为______ 232

$$(6*20+6)*2+100=352$$

$$(6*10+6)*2+100=232$$

特殊矩阵的压缩存储

1. 什么是压缩存储？

若多个数据元素的值都相同，则只分配一个元素值的存储空间，且零元素不占存储空间。

2. 什么样的矩阵能够压缩？

一些特殊矩阵，如：对称矩阵，对角矩阵，三角矩阵，稀疏矩阵等。

3. 什么叫稀疏矩阵？

矩阵中非零元素的个数较少（一般小于5%）

数组下标(i,j) 确定 存储地址

1. 对称矩阵

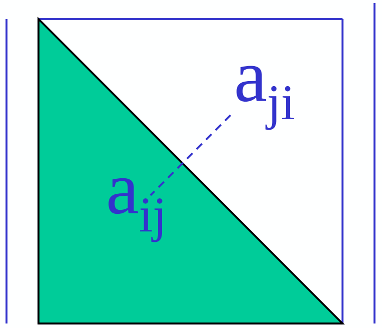
[特点] 在n×n的矩阵a中，满足如下性质：

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

[存储方法] 只存储下(或者上)三角(包括主对角线)的数据元素。共占用n(n+1)/2个元素空间。

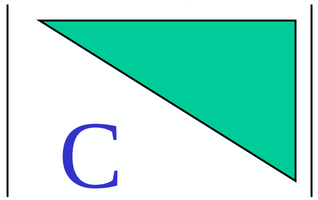
sa	a_{11}	a_{21}	a_{22}	a_{31}		$a_{ij}(a_{ji})$		a_{nn}
k	1	2	3	4				n(n+1)/2

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

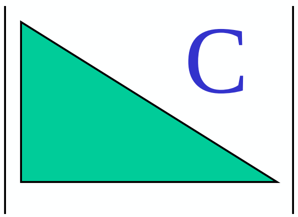


2. 三角矩阵

[特点] 对角线以下(或者以上)的数据元素(不包括对角线)全部为常数c。



上三角矩阵



下三角矩阵

[存储方法] 重复元素c共享一个元素存储空间，共占用 n(n+1)/2+1个元素空间: sa[1.. n(n+1)/2+1]

上三角矩阵

$$k=\begin{cases}(i-1)\times(2n-i+2)/2+j-i+1 & i\leq j\\ n(n+1)/2+1 & i>j\end{cases}$$

下三角矩阵

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

3. 对角矩阵（带状矩阵）

[特点] 在n×n的方阵中，非零元素集中在主对角线及其两侧共L(奇数)条对角线的带状区域内 — L对角矩阵。

[存储方法]

- **以对角线的顺序存储**

$$
\begin{pmatrix}
8 & 2 & 3 & 0 & 0 & 0 \\
4 & 2 & 0 & 3 & 0 & 0 \\
5 & 7 & 7 & 6 & 8 & 0 \\
0 & 9 & 6 & 9 & 1 & 5 \\
0 & 0 & 6 & 1 & 4 & 2 \\
0 & 0 & 0 & 2 & 8 & 3
\end{pmatrix}
\Rightarrow
$$

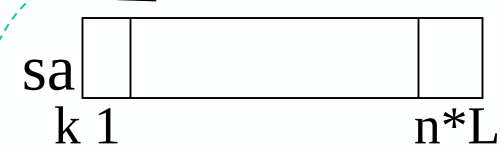
五对角矩阵

	1	2	3	4	5	6
-2			3	3	8	5
-1		2	0	6	1	2
0	8	2	7	9	4	3
1	4	7	6	1	8	
2	5	9	6	2		

s[-2..2; 1..6]

$$\begin{cases} i1=i-j \\ j1=j \end{cases}$$

|i-j|≤(L-1)/2



k=(i1+2)*n+j1=(i-j+2)*n+j

|i-j|≤(L-1)/2

- **只存储带状区内的元素**

除首行和末行，按每行 L个元素，共(n-2)L+(L+1)个元素。sa[1..(n-1)L+1]

$k=(i-1)L+1+(j-i) \qquad |i-j| \le (L-1)/2$

$$
\begin{bmatrix}
8 & 2 & 3 & 0 & 0 & 0 \\
4 & 2 & 0 & 3 & 0 & 0 \\
5 & 7 & 7 & 6 & 8 & 0 \\
0 & 9 & 6 & 9 & 1 & 5 \\
0 & 0 & 6 & 1 & 4 & 2 \\
0 & 0 & 0 & 2 & 8 & 3
\end{bmatrix}
$$

sa	8	2	3		4	2	0	3	5	7
k	1	2	3	4	5	6	7	8	9	10
	7	6	8	9	6	9	1	5	6	1
	11	12	13	14	15	16	17	18	19	20
	4	2		2	8	3				
	21	22	23	24	25	26				

稀疏矩阵

[特点] 大多数元素为零。

[常用存储方法] 只记录每一非零元素(i,j,a_{ij})

节省空间，但丧失随机存取功能

- 顺序存储：三元组表
- 链式存储：十字(正交)链表

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

稀疏矩阵的顺序存储：三元组表

1、三元组顺序表

	i	j	v
0	6	6	8
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	5	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 \end{pmatrix}$$

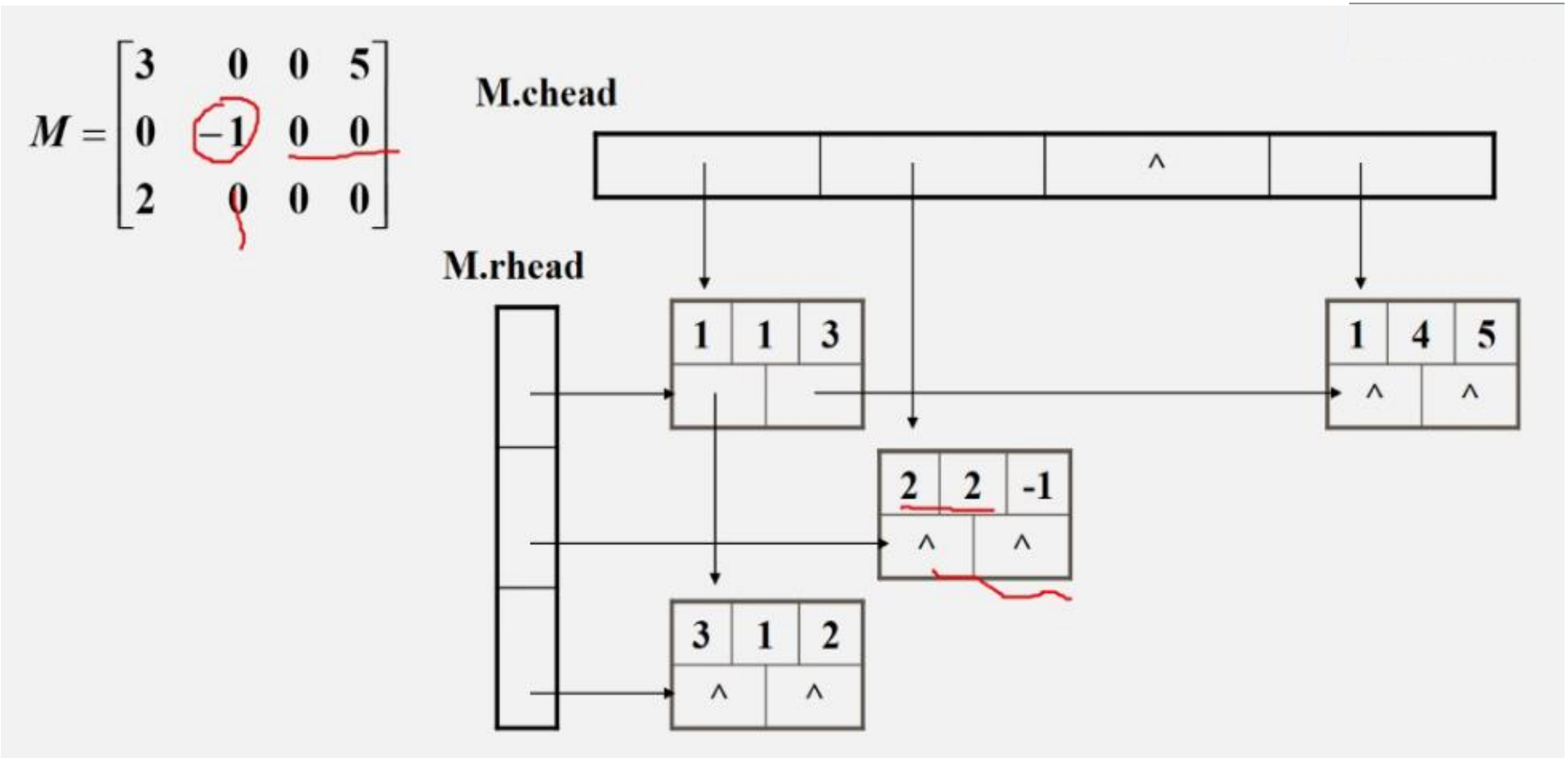
注意：为更可靠描述，通常再加一个“总体”信息：即总行数、总列数、非零元素总个数。

稀疏矩阵的链式存储：十字(正交)链表

- 优点：它能够灵活地插入因运算而产生的新的非零元素，
  删除因运算而产生的新的零元素，实现矩阵的各种运算。
- 在十字链表中，矩阵的每一个非零元素用一个结点表示，
  该结点除了（row，col，value）以外，还要有两个域：
  - right：用于链接同一行中的下一个非零元素；
  - down：用以链接同一列中的下一个非零元素。
- 十字链表中结点的结构示意图：

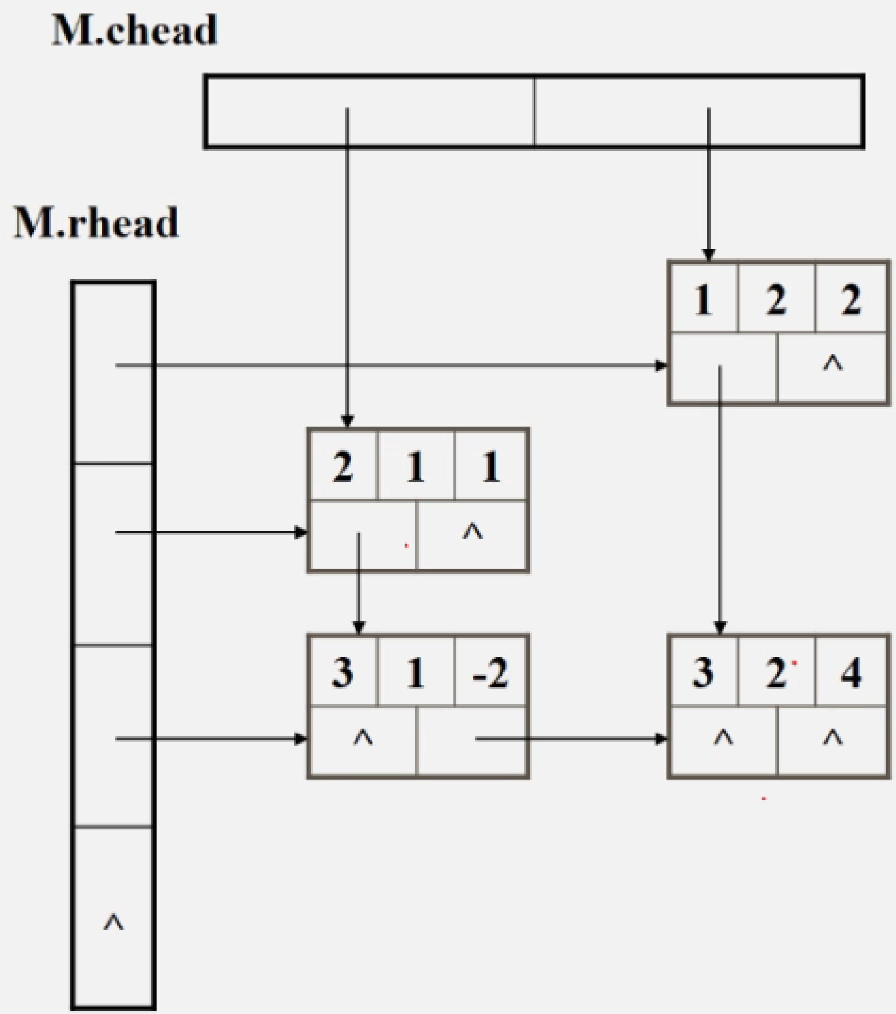
<table>
<tr><td colspan="2">row</td><td colspan="2">col</td><td>value</td></tr>
<tr><td>down</td><td colspan="4">right</td></tr>
</table>

稀疏矩阵的链式存储：十字(正交)链表



稀疏矩阵的链式存储：十字(正交)链表

$$N = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ -2 & 4 \\ 0 & 0 \end{bmatrix}$$



4.5 广义表

- 广义表（列表）： $n\ (\geq 0\ )$个表元素组成的有限序列，

记作$LS = (a_0, a_1, a_2, \ldots, a_{n-1})$

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。

广义表与线性表的区别？

- 线性表的成分都是结构上不可分的单元素
- 广义表的成分可以是单元素，也可以是有结构的表
- 线性表是一种特殊的广义表
- 广义表不一定是线性表，也不一定是线性结构

广义表的基本运算

- （1）求表头GetHead(L)：非空广义表的第一个元素，可以是一个单元素，也可以是一个子表
- （2）求表尾GetTail(L)：非空广义表除去表头元素以外其它元素所构成的表。表尾一定是一个表

广义表的基本运算

- （1）A=()　空表，长度为0
- （2）B=(( )) 长度为1，表头、表尾均为()。
- （3）C=(a,(b,c)) 长度为2，由原子a和子表(b,c)构成。

  表头为a；表尾为((b,c))。
- （4）E=(C,D) 长度为2，每一项都是子表。

  表头为C; 表尾为(D)。

练习

A=( )	GetHead和GetTail均无定义
A=(a,b)	GetHead(A)=a GetTail(A)=(b)
A=(a)	GetHead(A)=a GetTail(A)=( )
A=((a))	GetHead(A)=(a) GetTail(A)=( )

广义表的特点

- 有次序性:一个直接前驱和一个直接后继
- 有长度＝表中元素个数（最外层所包含元素的个数）
- 有深度＝表展开后所含括号的层数;
- 可递归:自己可以作为自己的子表
- 可共享可以为其他广义表所共享

练习：求下列广义表的长度

1）A =( )　　n=0，因为A是空表

2）B = (e)　　n=1，表中元素e是原子

3）C =(a ,(b , c , d))　　n=2，a 为原子，(b,c,d)为子表

4）D=(A , B ,C)　　n=3，3个元素都是子表

5）E=(a, E)　　n=2，a 为原子，E为子表

D=(A,B,C)=(( ),(e),(a,(b,c,d)))，共享表

E=(a,E)=(a,(a,E))= (a,(a,(a,…….)))，E为递归表

4.6 案例分析与实现

案例4.1 ：病毒感染检测

【案例分析】

●因为患者的DNA和病毒DNA均是由一些字母组成的字符串序列，要检测某种病毒DNA序列是否在患者的DNA序列中出现过，实际上就是字符串的模式匹配问题。

●可以利用BF算法，也可以利用更高效的KMP算法。

●但与一般的模式匹配问题不同的是，此案例中病毒的DNA序列是环状的。

●这样需要对传统的BF算法或KMP算法进行改进。

【案例实现】

●对于每一个待检测的任务，假设病毒DNA序列的长度是m，因为病毒DNA序列是环状的，为了线性取到每个可行的长度为m的模式串，可将存储病毒DNA序列的字符串长度扩大为$2m$，将病毒DNA序列连续存储两次。

●然后循环m次，依次取得每个长度为m的环状字符串，将此字符串作为模式串，将人的DNA序列作为主串，调用BF算法进行模式匹配。

●只要匹配成功，即可中止循环，表明该人感染了对应的病毒；否则，循环m次结束循环时，可通过BF算法的返回值判断该人是否感染了对应的病毒。

【算法步骤】

① 从文件中读取待检测的任务数*num*。

② 根据*num*个数依次检测每对病毒DNA和人的DNA是否匹配，循环*num*次，执行以下操作：

- 从文件中分别读取一对病毒DNA序列和人的DNA序列；
- 设置标志性变量flag，用来标识是否匹配成功，初始为0，表示未匹配；
- 病毒DNA序列的长度是*m*，将存储病毒DNA序列的字符串长度扩大为2*m*，将病毒DNA序列连续存储两次；
- 循环m次，重复执行以下操作：
  - 依次取得每个长度为*m*的病毒DNA环状字符串；
  - 将此字符串作为模式串，将人的DNA序列作为主串，调用BF算法进行模式匹配，将匹配结果返回赋值给flag；
  - 若flag非0，表示匹配成功，中止循环，表明该人感染了对应的病毒。
- 退出循环时，判断flag的值，若flag非0，输出“YES”，否则，输出“NO”。

小结

1. 了解串的存储方法，理解串的两种模式匹配算法，重点掌握BF算法。
2. 明确数组和广义表这两种数据结构的特点，掌握数组地址计算方法，了解几种特殊矩阵的压缩存储方法。
3. 掌握广义表的定义、性质及其GetHead和GetTail的操作。