

数据结构与算法

Data Structure and Algorithms

西安交通大学自动化系

蔡忠闽 周亚东

第四章 串

4.1 串类型的定义

4.2 串的实现

4.2.1 定长顺序存储表示

4.2.2 堆分配存储表示

4.2.3 串的块链存储表示

4.3 串的模式匹配算法

4.1 串类型的定义

- 问题:

- 在如下文本中查询关键词:

As one of the oldest universities in China, Xi'an **Jiaotong University** is one of the nine universities in the elite C9 League of Chinese premier universities (China's 'Ivy League').

查找关键词 Jiaotong University.

- 线性的数据结构

- 以字符序列为处理对象



串

4.1 串类型的定义

4.1.1 串及其基本概念

- **串的定义：**串 (String) 是零个或多个字符组成的有限序列。
一般记作 $S = \text{“}a_1a_2a_3\dots a_n\text{”}$ ，其中 S 是串名，双引号括起来的字符序列是串值；其中元素 a_i ($1 \leq i \leq n$) 可以是字母、数字或其它字符。
- **串的长度：**串中所包含的字符个数称为该串的长度。
- **空串：**长度为零的串称为空串 (Empty String)，它不包含任何字符。
- **空格串：**将仅由一个或多个空格组成的串称为空格串。注意：空串和空格串的不同，例如 “ ” 和 “ ” 分别表示长度为1的空格串和长度为0的空串。

4.1 串类型的定义

- **子串与主串：**串中任意个连续字符组成的子序列称为该串的**子串**，包含子串的串相应地称为**主串**。通常将子串在主串中首次出现时的该子串的首字符对应的主串中的序号，定义为子串在主串中的**位置**。

例如，设A=“This is a string” B=“is”则B是A的子串，A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是3。因此，称B在A中的位置为3。

- **相等：**当且仅当两个串的长度相等，并且每个对应位置的字符都相等时，称两个串相等。例如，A= “BEIJING”
B= “BEIJING”，则串A与串B相等。

4.1 串类型的定义

4.1.2 串的抽象数据定义

ADT String{

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

数据关系: $R = \{\langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2, \dots, n\}$

基本操作: StrAssign(&T, chars) \\生成值为常量char的串

StrEmpty(S) 判断S是否为空

StrCompare(S, T) 比较两字符串

Concat(&T, S1, S2) 将S1和S2连接起来赋值给T

SubString(&Sub, S, pos, len) 返回子串

Index(S, T, pos) 在S中pos位置之后寻找与T相同的子串

Replace(&S, T, V) 用V替换S中出现的所有与T相等且不重叠子串

StrInsert(&S, pos, T) 在串S中的第pos字符之前插入T

StrDelete(&S, pos, len) 在S中删除第pos字符起长为len的串

..... }

4.2 串的实现和表示

4.2.1 定长顺序存储表示

定长顺序存储结构，是直接使用定长的字符数组来存放串中的字符序列，数组的上界预先给出：

```
#define maxstrlen 255
```

```
typedef char SString[maxstrlen+1]; // 0分量存放串长
```

```
Status SubString( SString &Sub, SString S, int pos, int len )
```

```
{ if( pos < 1 || pos > S[0] || len < 0 || len > S[0]-pos+1 )
```

```
    return ERROR;
```

```
    Sub[1..len] = S[pos..pos+len-1];
```

```
    Sub[0] = len;
```

```
    return OK;
```

```
}
```

4.2 串的实现和表示

4.2.1 定长顺序存储表示

定长顺序存储结构，是直接使用定长的字符数组来存放串中的字符序列，数组的上界预先给出：

```
#define maxstrlen 255
```

```
typedef char SString[maxstrlen+1]; // 0分量存放串长
```

```
Status Concat(SString &T, SString S1, SString S2)
```

如何在定长顺序存储表示的物理结构中完成该算法

4.2 串的实现

Status Concat(SString &T, SString S1, SString S2)

{ if(S1[0] + S2[0] <= maxstrlen) // 未截断, T=S1+S2

{ T[1..S1[0]] = S1[1..S1[0]];

T[S1[0]+1..S1[0]+S2[0]] = S2[1..S2[0]];

T[0] = S1[0] + S2[0]; return TRUE;

}

else if (S1[0] < maxstrlen) // 截断, T=S1+部分S2

{ T[1..S1[0]] = S1[1..S1[0]];

T[S1[0]+1..maxstrlen] = S2[1..maxstrlen-S1[0]];

T[0] = maxstrlen; return FALSE;

}

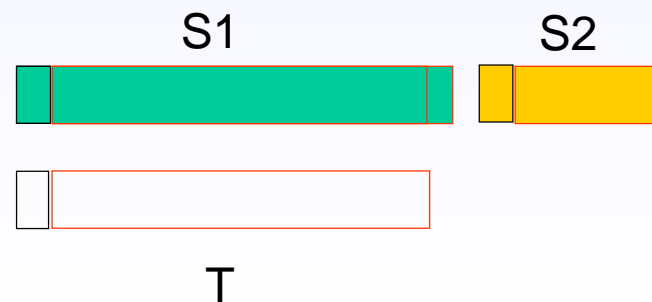
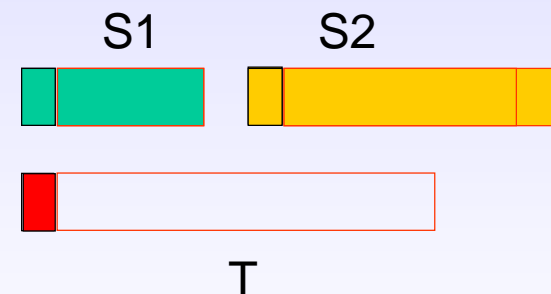
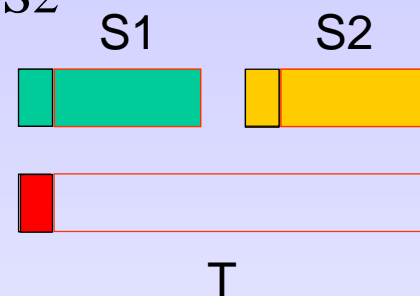
else // 截断, T=S1

{ T[0..maxstrlen] = S1[0..maxstrlen];

return FALSE;

}

}



4.2 串的实现和表示

4.2.2 堆分配存储表示

以一组地址连续的存储单元存放串值字符序列，但它们的存储空间是在程序执行过程中动态分配而得，所以也称为动态存储分配的顺序表。在C语言中，利用`malloc`和`free`等动态存储管理函数，来根据实际需要动态分配和释放字符数组空间。

4.2 串的实现和表示

//串的堆分配存储表示

```
typedef struct{  
    char *ch;  
    int length;  
} HString;
```

//基本操作函数说明

Status StrAssign(HString &t, char *chars) //声称值为chars的串

int StrLength(HString s) // 求串s的长度

void ClearString(HString &s) // 将串s清为空串

int StrCompare(HString s, HString t) // 串的比较

Status Concat(HString &t, HString s1, HString s2) // 串的联结

Status SubString(HString &sub, HString s, int pos, int len)

//求子串

4.2 串的实现和表示

```
Status StrAssign(HString &t, char *chars)
```

```
{ //生成一个其值等于串常量chars的串t
```

```
    if(t.ch) free(t.ch);           //先清空已有数据
```

```
    for(i=0, c=chars; *c; ++i, ++c); //求chars的长度
```

```
    if(!i) { t.ch=NULL; t.length=0; }
```

```
    else{
```

```
        t.ch=(char *)malloc(i*sizeof(char));
```

```
        if(t.ch==NULL) exit(OVERFLOW);
```

```
        t.ch[0..i-1] = chars[0..i-1];
```

```
        t.length=i;
```

```
    }
```

```
    return OK;
```

```
}
```

4.2 串的实现和表示

```
int StrLength(HString s) // 求串s的长度
```

```
{ return s.length; }
```

```
void ClearString(HString &s) // 将串s清为空串
```

```
{ if(s.ch) { free(s.ch); s.ch=NULL;}
```

```
    s.length=0;
```

```
}
```

```
int StrCompare(HString s, HString t) // 串的比较
```

```
{ for(i=0; i<s.length && i<t.length; ++i)
```

```
    if(s.ch[i] != t.ch[i]) return(s.ch[i] - t.ch[i]);
```

```
    return s.length-t.length;
```

```
}
```

4.2 串的实现和表示

```
Status SubString(HString &sub, HString s, int pos, int len)
```

```
{ if( pos<1 || pos>s.length || len<0 || len>s.length-pos+1 )
```

```
    return ERROR;
```

```
    if(sub.ch) free(sub.ch); //先清空sub的内容
```

```
    if(!len) { sub.ch=NULL; sub.length=0; }
```

```
    else
```

```
    { sub.ch=(char *)malloc(len*sizeof(char));
```

```
      sub.ch[0..len-1]=s[pos-1..pos+len-2];
```

```
      s.length=len;
```

```
    }
```

```
    return OK;
```

```
}
```

4.2 串的实现和表示

定长顺序存储

```
Status SubString( SString &Sub, SString S, int pos, int len )  
{ if( pos < 1 || pos > S[0] || len < 0 || len > S[0]-pos+1 )  
    return ERROR;  
    Sub[1..len] = S[pos..pos+len-1];  
    Sub[0] = len;  
    return OK;  
}
```

4.2 串的实现和表示

Status Concat(HString &t, HString s1, HString s2) // 串联结

{ if(t.ch) free(t.ch); **//先清空t的内容**

t.ch=(char*)malloc((s1.length+s2.length)*sizeof(char));

if(t.ch==NULL) exit(overflow);

t.ch[0..s1.length-1]=s1.ch[0..s1.length-1];

t.length=s1.length+s2.length;

t.ch[s1.length..t.length-1]=s2.ch[0..s2.length-1];

return OK;

}

4.2 串的实现

Status Concat(SString &T, SString S1, SString S2)

{ if(S1[0] + S2[0] <= maxstrlen) // 未截断, T=S1+S2

{ T[1..S1[0]] = S1[1..S1[0]];

T[S1[0]+1..S1[0]+S2[0]] = S2[1..S2[0]];

T[0] = S1[0] + S2[0]; return TRUE;

}

else if (S1[0] < maxstrlen) // 截断, T=S1+部分S2

{ T[1..S1[0]] = S1[1..S1[0]];

T[S1[0]+1..maxstrlen] = S2[1..maxstrlen-S1[0]];

T[0] = maxstrlen; return FALSE;

}

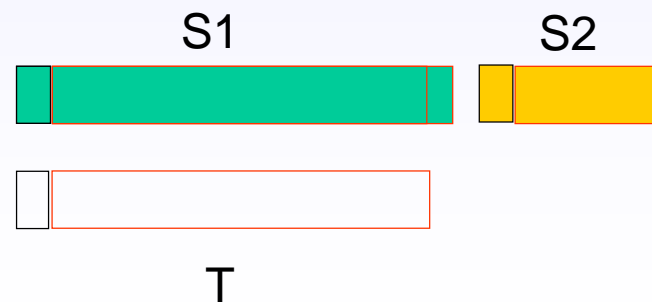
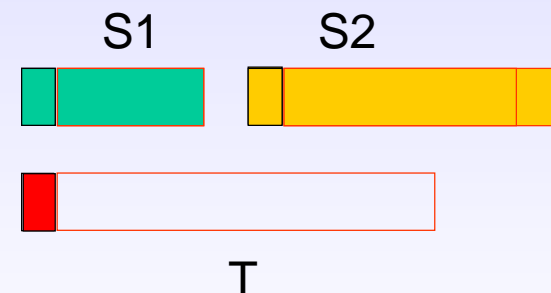
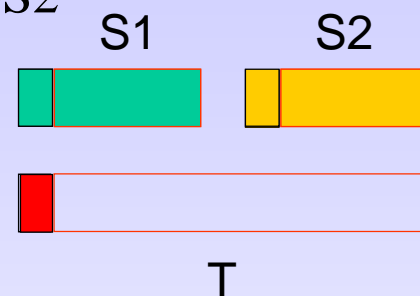
else // 截断, T=S1

{ T[0..maxstrlen] = S1[0..maxstrlen];

return FALSE;

}

}



4.2 串的实现和表示

4.2.3 串的块链存储表示

顺序串上的插入和删除操作不方便，需要移动大量的字符。因此，可用单链表方式来存储串值，串的这种链式存储结构简称为链串。一个链串由头指针唯一确定。

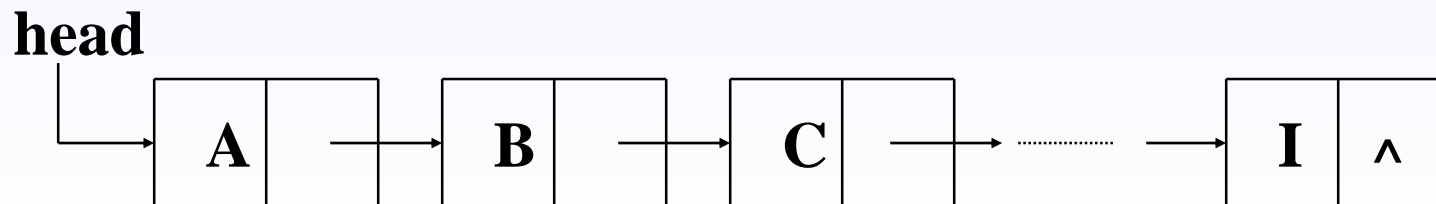
优点：便于进行插入和删除运算

缺点：存储空间利用率低

```
# define CHUNKSIZE 80
```

```
typedef struct Chunk {  
    Char ch[CHUNKSIZE ];  
    struct Chunk *next;  
}Chunk;
```

```
typedef struct{  
    Chunk *head, *tail;  
    int  curlen;  
} LString;
```



4.3 串的模式匹配算法

串的模式匹配 (pattern matching) 运算的应用非常广泛。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。

4.1 串类型的定义

- 问题:

- 在如下文本中查询关键词:

As one of the oldest universities in China, Xi'an **Jiaotong University** is one of the nine universities in the elite C9 League of Chinese premier universities (China's 'Ivy League').

查找关键词 Jiaotong University.

- 线性的数据结构

- 以字符序列为处理对象



串

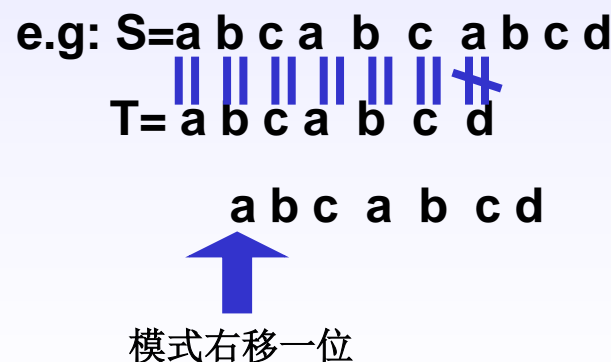
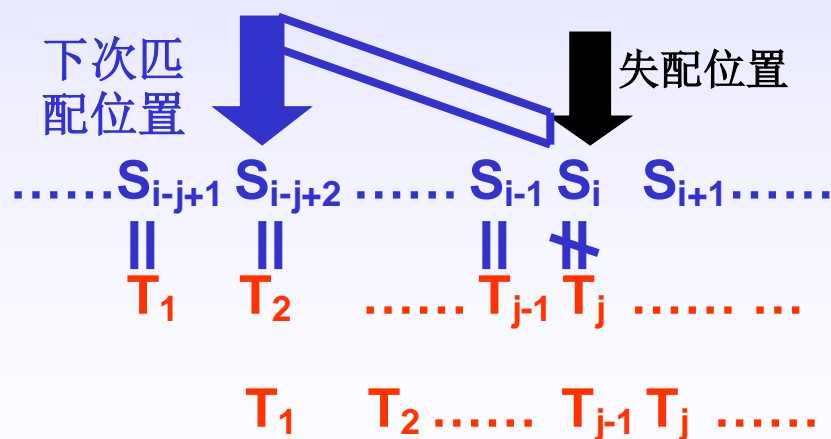
4.3 串的模式匹配算法

- 设有两个串 S 和 T : $S = s_0s_1\dots s_{n-1}$, $T = t_0t_1\dots t_{m-1}$
其中 $1 < m \leq n$ (通常 $m \ll n$)
- 模式匹配的目的是在 S 中找出和 T 相同的子串。
- 此时, S 称为“目标”, 而 T 称为“模式”。
- 模式匹配的结果有两种:
 - 匹配成功: S 中存在等于 T 的子串, 返回子串在 S 中的位置
 - 匹配失败: 返回一个特定的标志 (如 -1) 。

4.3 串的模式匹配算法

4.3.1 朴素模式匹配算法--Brute-Force (BF)算法:

基本方法: 从指定位置开始逐个比较主串与模式的字符，一旦发现出现字符不匹配，则整个模式相对于原来的位置右移一位。如下图所示:



4.3 串的模式匹配算法

```
int Index( SString S, SString T, int pos ) // S 目标, T 模式
{
    i = pos ; j = 1;
    while ( i <= S[0] && j <= T[0] ) // 0号单元存放串长
    {
        if ( S[i] == T[j] ) { ++i ; ++j ; } // 继续比较
        else { i = i - j + 2; j = 1; } // 指针后退重新开始匹配
    }
    if ( j > T[0] ) return i - T[0]; // T在S中的匹配起始位置
    else return 0;
}
```

4.3 串的模式匹配算法

下标i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
目标S	a	a	b	c	b	a	b	c	a	a	b	c	a	a	b	a	b	c	
模式T	a	b	c	a	a	b	a	b	c										
		a	b	c	a	a	b	a	b	c									
			a	b	c	a	a	b	a	b	c								
				a	b	c	a	a	b	a	b	c							
					a	b	c	a	a	b	a	b	c						
						a	b	c	a	a	b	a	b	c					
							a	b	c	a	a	b	a	b	c				
								a	b	c	a	a	b	a	b	c			
									a	b	c	a	a	b	a	b	c		
										a	b	c	a	a	b	a	b	c	
											a	b	c	a	a	b	a	b	c

4.3 串的模式匹配算法

时间复杂度分析:

- 朴素模式匹配算法一旦比较不等，T右移一个字符并且下次从t0开始重新进行比较，对于目标S，存在回溯现象。
- 匹配失败的最坏情况：每趟匹配皆在最后一个字符不等，且有 **$n-m+1$** 趟匹配(每趟比较m个字符)，共比较 $m*(n-m+1)$ 次，由于 $m \ll n$ ，因此最坏时间复杂度 $O(n*m)$ 。
- 匹配失败的最好情况： $n-m+1$ 次比较[每趟只比较第一个字符]。
- 匹配成功的最好情况： m 次比较。
- 匹配成功的最坏情况：与匹配失败的最坏情况相同。
- 综上所述：朴素模式匹配算法的时间复杂度为 $O(m*n)$

4.3 串的模式匹配算法


4.3.2 Knuth-Morris-Pratt 模式匹配算法 (KMP 算法)

说明BF模式匹配算法在最坏情况下时间复杂度为 $O(n*m)$ 的实例（ n 为主串长度， m 为模式长度）。每比较 m 次，移动模式一次。最后在主串的 $n-m+1$ 找到主串，比较 $(n-m+1)*m$ 次。



4.3 串的模式匹配算法

说明 **KMP** 算法的实例：

失配点

 S = abcabcabcd
 T = abcabcd

S = abcabcabcd
 T = abcabcd
 右移一位，仍失配

S = abcabcabcd
 T = abcabcd
 又右移一位，仍失配

S = abcabcabcd
 T = abcabcd

三次比较省去？

本次比较省去？

本次比较省去？

问题：能否省去上述五次比较，直接进行 **S7** 和 **T4** 之间的比较呢？

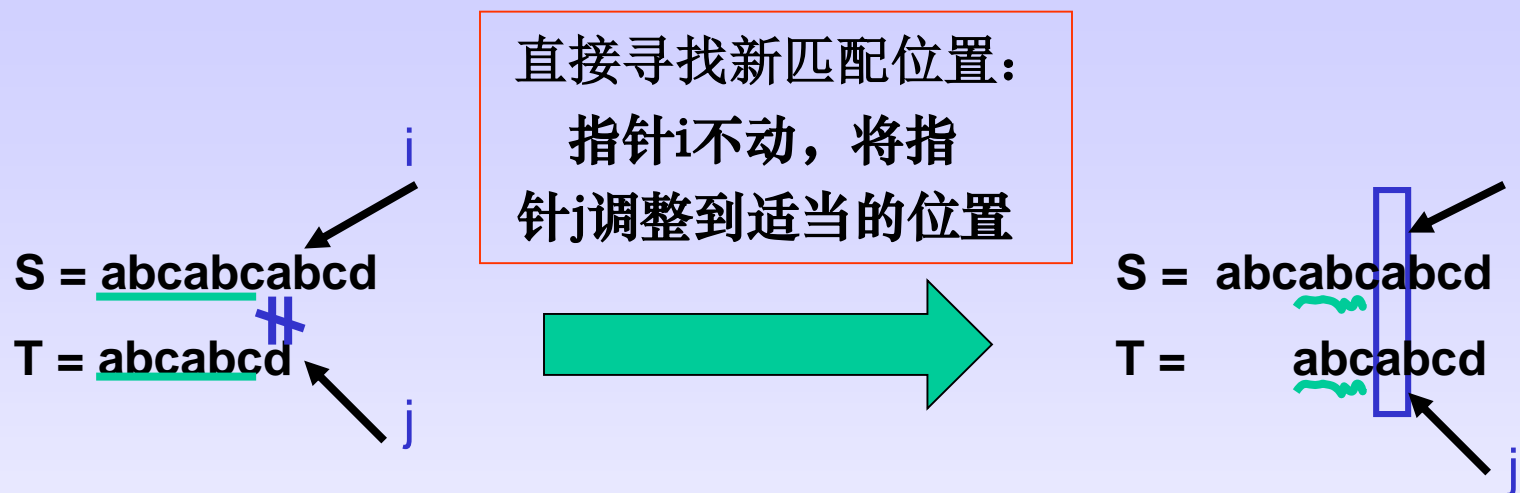
再右移一位，三次比较之后，再进行断点处的比较，比较成功！

4.3 串的模式匹配算法

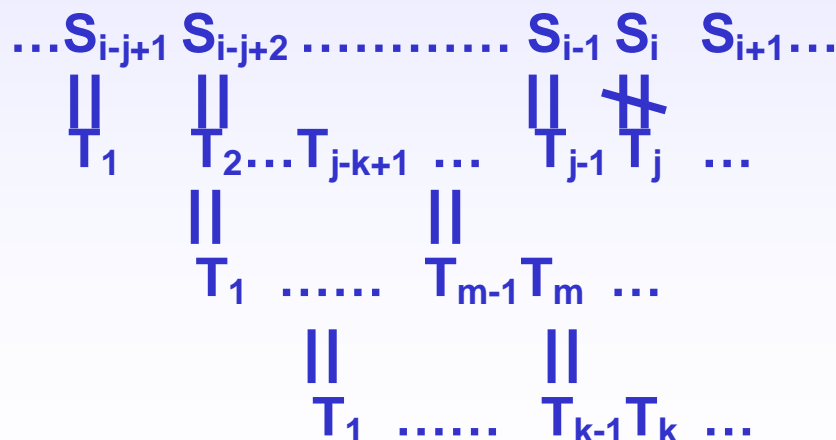
4.3.2 Knuth-Morris-Pratt 模式匹配算法（KMP 算法）

- 1、朴素模式匹配算法中目标串的指针*i*不需要回溯
- 2、模式串是当前比较位置围绕*i*跳动
- 3、**next[j]**记录了模式串*j*位置与目标串*i*位置不匹配后，下一个与*i*位置进行比较的位置
- 4、失配时按照**next[j]**的两种情况进行移动：**0**和大于**0**
- 5、找到一个匹配或到达目标串的末尾

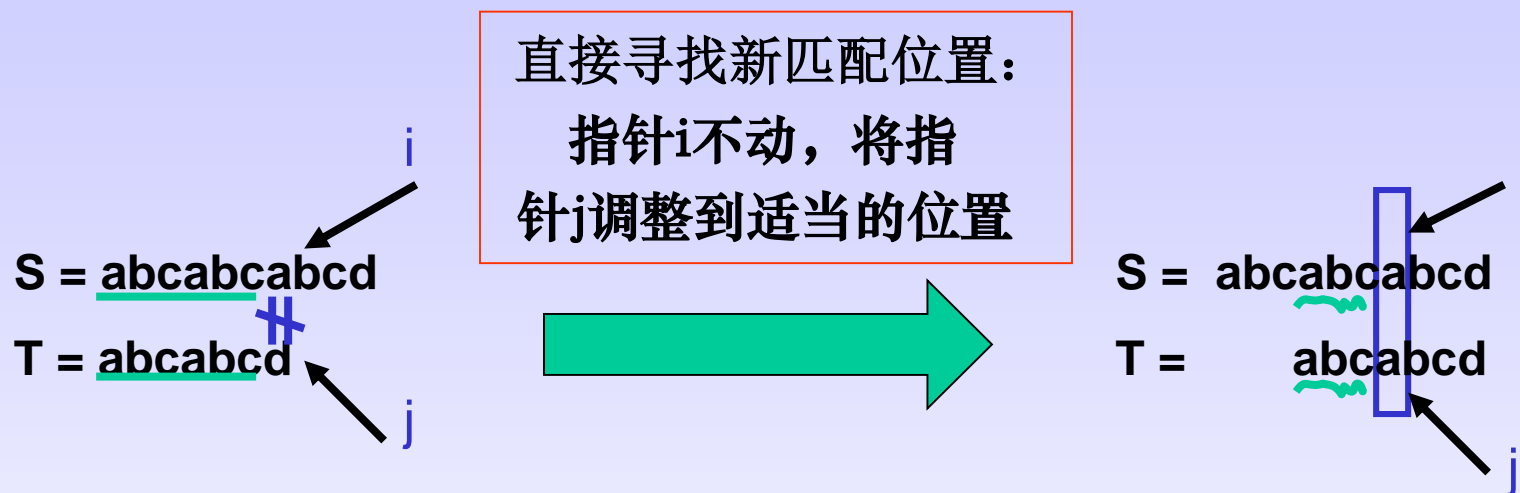
4.3 串的模式匹配算法



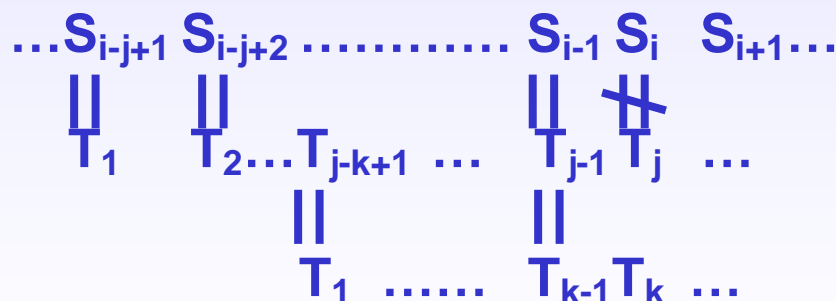
分析：当 S_i 和 T_j 发生失配时，如果要将 S_i 和 T_k 匹配（如下图所示）



4.3 串的模式匹配算法



分析：当 S_i 和 T_j 发生失配时，如果要将 S_i 和 T_k 匹配（如下图所示）



则应满足条件： $T_1 T_2 \dots T_{k-1} = T_{j-k+1} T_{j-k+2} \dots T_{j-1}; \ (k < j)$

4.3 串的模式匹配算法

引进一个数组**next**， $next[j]$ 的含义是当模式中第 j 个字符与主串中第 i 个字符匹配失败时，在模式中需要重新和主串中第 i 个字符进行比较的字符的位置。

$$next[j] = \begin{cases} 0 & j=1 \\ \text{Max}\{k \mid 1 < k < j \text{ 且 } T_1 \dots T_{k-1} = T_{j-k+1} \dots T_{j-1}\}, \text{此集合非空} & \\ 1 & \text{其它} \end{cases}$$

例：

j	1	2	3	4	5	6	7
T	A	B	C	A	B	C	D
next[j]	0	1	1	1	2	3	4
T	A	B	A	A	B	C	A
next[j]	0	1	1	2	2	3	1
T	A	A	A	A	A	A	A
next[j]	0	1	2	3	4	5	6

4.3 串的模式匹配算法

```
int Index( SString S, SString T, int pos ) // S 目标, T 模式
{
    i = pos ; j = 1;
    while ( i <= S[0] && j <= T[0] ) // 0号单元存放串长
    {
        if ( S[i] == T[j] ) { ++i ; ++j ; } // 继续比较
        else { i = i - j + 2; j = 1; }      // 指针后退重新开始匹配
    }
    if ( j > T[0] ) return i-T[0];          // T在S中的匹配起始位置
    else return 0;
}
```

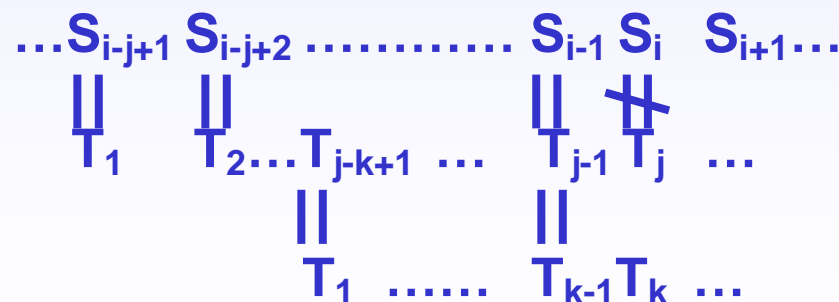
4.3 串的模式匹配算法

- 利用 `next[j]` 函数值的模式匹配（KMP）算法：

```

int Index_KMP( SString S, SString T, int pos )
// 已知 next 函数值，T 非空，1<=pos<=Strlength(S)
{   i = pos ; j = 1;
    while ( i <= S[0] && j <= T[0] )
    {   if ( j == 0 || S[i] == T[j] ) { ++i ; ++j ; }
        else j = next[ j ];
    }
    if ( j > T[0] ) return i-T[0];
    else return 0;
} // Index_KMP

```



4.3 串的模式匹配算法

- 利用 `next[j]` 函数值的模式匹配（KMP）算法：

```
int Index_KMP( SString S, SString T, int pos )
// 已知 next 函数值，T 非空，1<=pos<=Strlength(S)
{   i = pos ; j = 1;
    while ( i <= S[0] && j <= T[0] )
    {   if ( j == 0 || S[i] == T[j] ) { ++i ; ++j ; }
        else j = next[ j ];
    }
    if ( j > T[0] ) return i-T[0]; // T在S中的匹配起始位置
    else return 0;
} // Index_KMP
```

• Next[]的求解

1) Next[]的求解只与模式串本身有关;

2) $\text{next}[j] = k$ 表明在模式串中存在:

$$'T_1 \cdots T_{k-1}' = 'T_{j-k+1} \cdots T_{j-1}'$$

其中 $1 < k < j$, 且不存在 $k' > k$

3) 在 $\text{next}[j] = k$ 基础上求 $\text{next}[j+1]$:

若 $T_k = T_j$, 则 $'T_1 \cdots T_{k-1} T_k' = 'T_{j-k+1} \cdots T_{j-1} T_j'$

故 $\text{next}[j+1] = k+1 = \text{next}[j]+1$

若 $T_k \neq T_j$ 则 $'T_1 \cdots T_{k-1} T_k' \neq 'T_{j-k+1} \cdots T_{j-1} T_j'$

问题转化为串T和它本身间的模式匹配问题:

$$T_{j-k+1} = T_1, T_{j-k+2} = T_2, \cdots, T_{j-1} = T_{k-1}, T_j \neq T_k$$

$$T_{j-k+1} = T_1, T_{j-k+2} = T_2, \dots, T_{j-1} = T_{k-1}, T_j \neq T_k$$



在该问题中，需要将下面的T向右移动使得 $T_{k'}$ 与 T_j 比较，即 $\text{next}[k]=k'$ 。
若 $T_{k'} = T_j$ ，则

$$'T_1 \dots T_{k'-1} T_{k'}' = 'T_{j-k'+1} \dots T_{j-1} T_j'$$

故 $\text{next}[j+1] = k' + 1 = \text{next}[k] + 1$

若 $T_{k'} \neq T_j$ ，则用第 $\text{next}[k']$ 个元素与 T_j 比较，以此类推，直至有某个字符匹配成功，否则 $\text{next}[j+1] = 1$ 。

```

void get_next(SString 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[0]){
    if (k == 0 || T[j] == T[k])
      { ++j; ++k; next[j] = k;}
    else k = next[k];
  }
}

```

T_1	T_2	...	T_{i-j+1}	T_{i-j+2}	...	T_{i-1}	T_i	...
\parallel	\parallel		\parallel	\parallel		\parallel	\nparallel	
T_1	T_2	...	T_{j-1}	T_j	...			

4.3 串的模式匹配算法

KMP算法复杂度分析:

```
int Index_KMP( SString S, SString T, int pos )
// 已知 next 函数值, T 非空, 1<=pos<=Strlength(S)
{   i = pos ; j = 1;
    while ( i <= S[0] && j <= T[0] )
    {   if ( j == 0 || S[i] == T[j] ) { ++i ; ++j ; }
        else j = next[ j ];
    }
    if ( j > T[0] ) return i-T[0]; // T在S中的匹配起始位置
    else return 0;
} // Index_KMP
```

4.3 串的模式匹配算法

KMP算法复杂度分析:

- 算法中 i 值只增不减, 由于 i 的初值为 pos , 循环过程中保持 $i < n$, 所以循环体中 $i++$ 语句的执行次数不超过 n , 从而 $j++$ 的执行次数也不超过 n 。
- 另外, j 的初始值为1, 唯一使 j 减少的语句是 $j = next[j]$, 由于 $next[j]$ 在 $[0, j]$ 范围内, 一旦 $j = next[j]$ 使 $j=0$, 那么下步必定执行 $\{i++, j++\}$ 。由于“ $j = next(j);$ ”每执行一次必然使得 j 减少(至少减1), 而使得 j 增加的操作只有“ $j++$ ”, 由上面的讨论知道, “ $j++$ ”执行的次数不超过 n 次。那么, 如果“ $j = next(j);$ ”的执行次数超过 n 次, 最终的结果必然使得 j 为负数。因此, $j = next[j]$ 的执行次数也必定不超过 $i++$ 的执行次数加1。
- 因此, 在已经计算出 $next$ 数组的前提下, 算法的时间效率为 $O(n)$ 。

- Next[]的修正

aaaab:

J = 1 2 3 4 5

next[j] 0 1 2 3 4

aaabaaaab

... S_{i-j+1} S_{i-j+2} S_{i-1} S_i S_{i+1} ...

⊥

T_1 T_2 ... T_{j-k+1} T_{j-k+2} ... T_{j-1} T_j T_{j+1} ...

|| || || ||
 T_1 T_2 ... T_{k-1} T_k T_{k+1} ...

?

T_1 T_2 ... $T_{k'}$ $T_{k'+1}$...

- Next[]的修正

```
void get_next(SString 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[0]){  
    if (k == 0 || T[j] == T[k])  
      { ++j; ++k;  
        if (T[j] != T[k]) nextval[j] = k;  
        else nextval[j] = nextval[k]  
      }  
    else k = nextval[k];  
  }  
}
```

第四章 串

4.1 串类型的定义

4.2 串的实现

4.2.1 定长顺序存储表示

4.2.2 堆分配存储表示

4.2.3 串的块链存储表示

4.3 串的模式匹配算法

数据结构与算法

【课程内容】

- ☞ 数据的各种逻辑结构和物理结构（存储结构），以及它们之间的相应关系
- ☞ 并对每种结构定义相适应的各种运算
- ☞ 设计出相应的算法
- ☞ 分析算法的效率

常用数据结构类型： 线性表、栈和队列、串、数组和特殊矩阵、树和二叉树、图。

第五章 数组和广义表

- 5.1 数组的定义
- 5.2 数组的顺序表示和实现
- 5.3 矩阵的压缩存储
 - 5.3.1 特殊矩阵
 - 5.3.2 稀疏矩阵
- 5.4 广义表的定义
- 5.5 广义表的存储结构

5.1 数组的定义

数组的定义

数组是由一组**类型相同**的数据元素构成的**有序**集合，每个数据元素称为一个数组元素（简称为元素），每个元素受 $n(n \geq 1)$ 个**线性关系**的约束，每个元素在 n 个线性关系中的序号 i_1 、 i_2 、...、 i_n 称为该元素的下标，并称该数组为 n 维数组。

数组的特点

- 元素本身可以具有某种结构，属于同一数据类型；
- 数组是一个具有固定格式和数量的数据集合。

5.1 数组的定义

数组的抽象数据类型

ADT Array{

数据对象: $j_i = 0, \dots, b_i - 1, i = 1, 2, \dots, n,$

$D = \{a_{j_1 j_2 \dots j_n} | n(>0) \text{ 称为数据的维数, } b_i \text{ 是数组第 } i \text{ 维的长度,}$

$j_i \text{ 是数组元素的第 } i \text{ 维下标, } a_{j_1 j_2 \dots j_n} \in \text{ElemSet}\}$

数据关系: $R = \{R_1, R_2, \dots, R_n\}$

$R_i = \{ \langle a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_i + 1 \dots j_n} \rangle | \quad 0 \leq j_k \leq b_k - 1, 0 \leq k \leq n, \text{ 且 } k \neq i,$

$0 \leq j_i \leq b_i - 2,$

$a_{j_1 \dots j_i \dots j_n}, a_{j_1 \dots j_i + 1 \dots j_n} \in D \}$

基本操作:

InitArray(&A, n, bound1, ... , boundn)

//若维数n和各维长度合法, 则构造相应数组A, 并返回ok;

DestroyArray(&A)

//销毁数组A;

Value(A, &e, index1, ... , indexn)

//indexi表示第i个下标值, 若各下标不出界, 则将e赋值为对应元素

Assign(&A, e, index1, ... , indexn)

//indexi表示第i个下标值, 若各下标不出界, 则将e赋值给对应元素

Transpose(&A, B)

5.1 数组的定义

数组示例

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \mathbf{a_{22}} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

例如，元素 a_{22} 受两个线性关系的约束，在行上有一个行前驱 a_{21} 和一个行后继 a_{23} ，在列上有一个列前驱 a_{12} 和一个列后继 a_{32} 。

5.1 数组的定义

数组——线性表的推广

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



$$A = (A_1, A_2, \dots, A_n)$$

其中:

$$A_i = (a_{1i}, a_{2i}, \dots, a_{mi}) \\ (1 \leq i \leq n)$$

二维数组是数据元素为线性表的线性表。

5.1 数组的定义

数组的基本操作

- (1) 存取：给定一组下标，读出对应的数组元素；
- (2) 修改：给定一组下标，存储或修改与其相对应的数组元素。

存取和修改操作本质上只对应一种操作——**寻址**

① 数组应该采用何种方式存储？

数组没有插入和删除操作，所以，不用预留空间，适合采用顺序存储。

数组的顺序存储方式

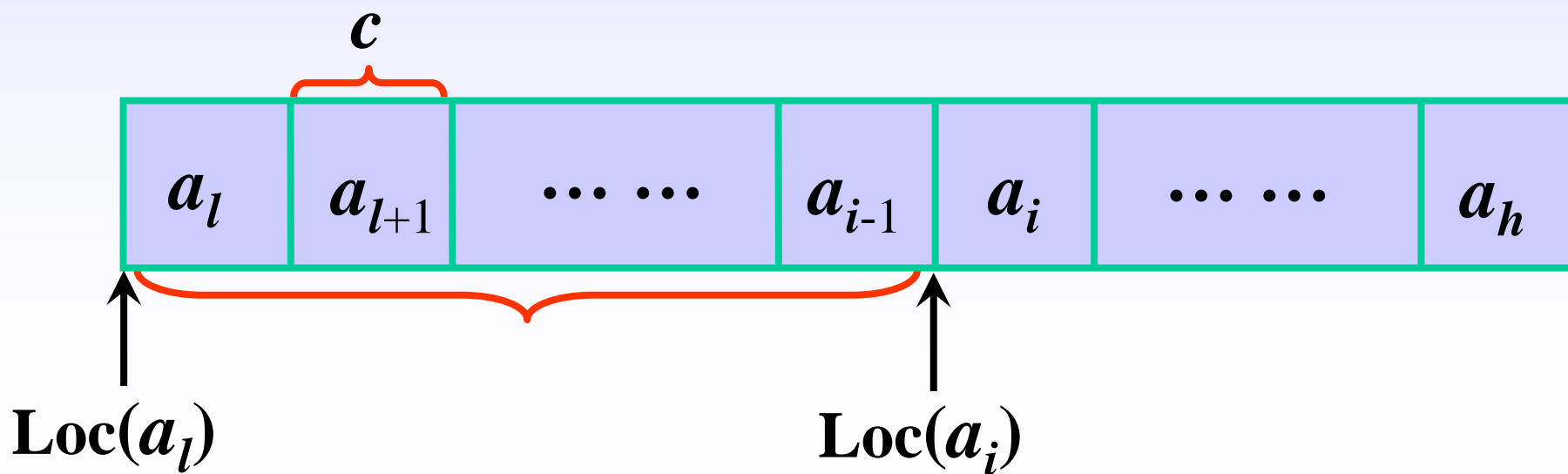
```
# define MAX_ARRAY_DIM
typedef struct {
    ElemType * base;    //数组元素基址;
    int      dim;       //.....维数;
    int      *bounds;   //.....维界基址;
    int      *constants //.....映像函数常量基址.
} Array
```

5.2 数组的表示与实现

数组的存储结构与寻址——一维数组

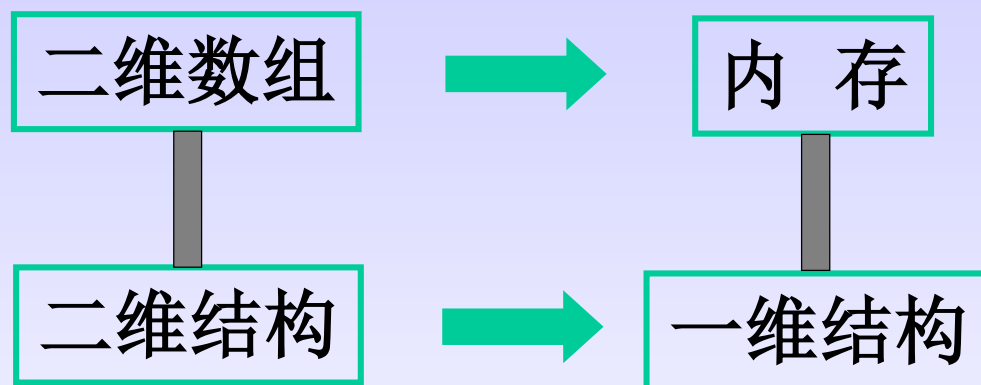
设一维数组的下标的范围为闭区间 $[l, h]$ ，每个数组元素占用 c 个存储单元，则其任一元素 a_i 的存储地址可由下式确定：

$$\text{Loc}(a_i) = \text{Loc}(a_l) + (i - l) \times c$$



5.2 数组的表示与实现

数组的存储结构与寻址——二维数组

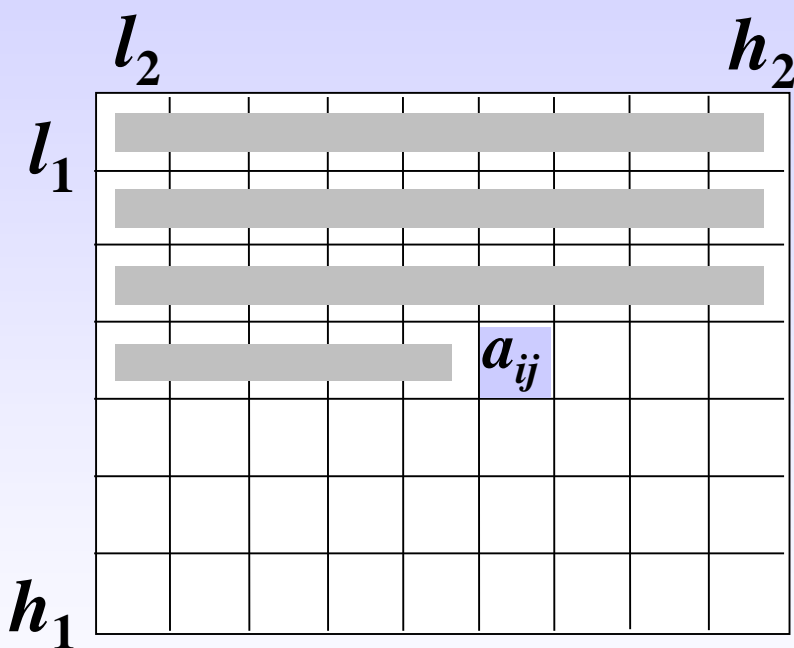


常用的映射方法有两种：

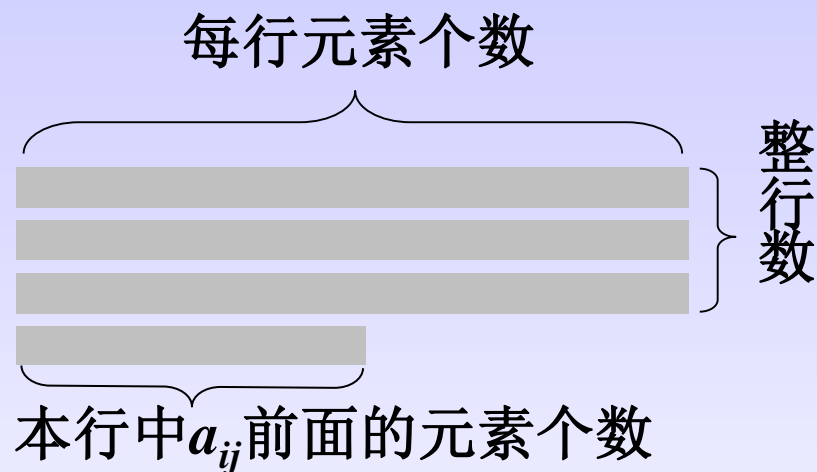
- 按**行**优先：**先行后列**，先存储行号较小的元素，行号相同者先存储列号较小的元素。
- 按**列**优先：**先列后行**，先存储列号较小的元素，列号相同者先存储行号较小的元素。

5.2 数组的表示与实现

按行优先存储的寻址



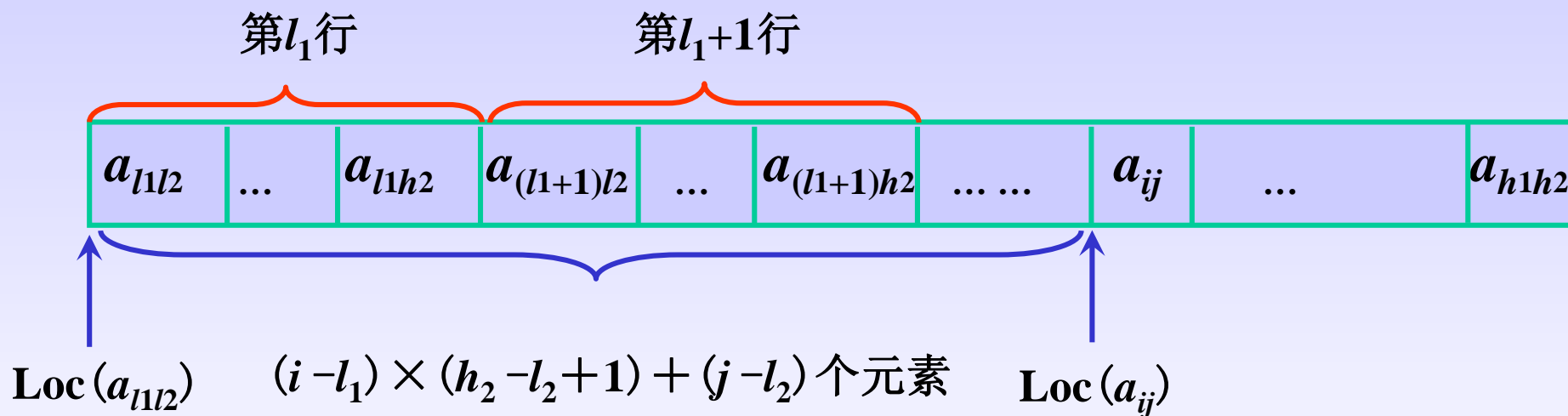
(a) 二维数组



$$\begin{aligned}
 & a_{ij} \text{ 前面的元素个数} \\
 &= \text{阴影部分的面积} \\
 &= \text{整行数} \times \text{每行元素个数} + \text{本行中} \\
 & a_{ij} \text{ 前面的元素个数} \\
 &= (i - l_1) \times (h_2 - l_2 + 1) + (j - l_2)
 \end{aligned}$$

5.2 数组的表示与实现

按行优先存储的寻址



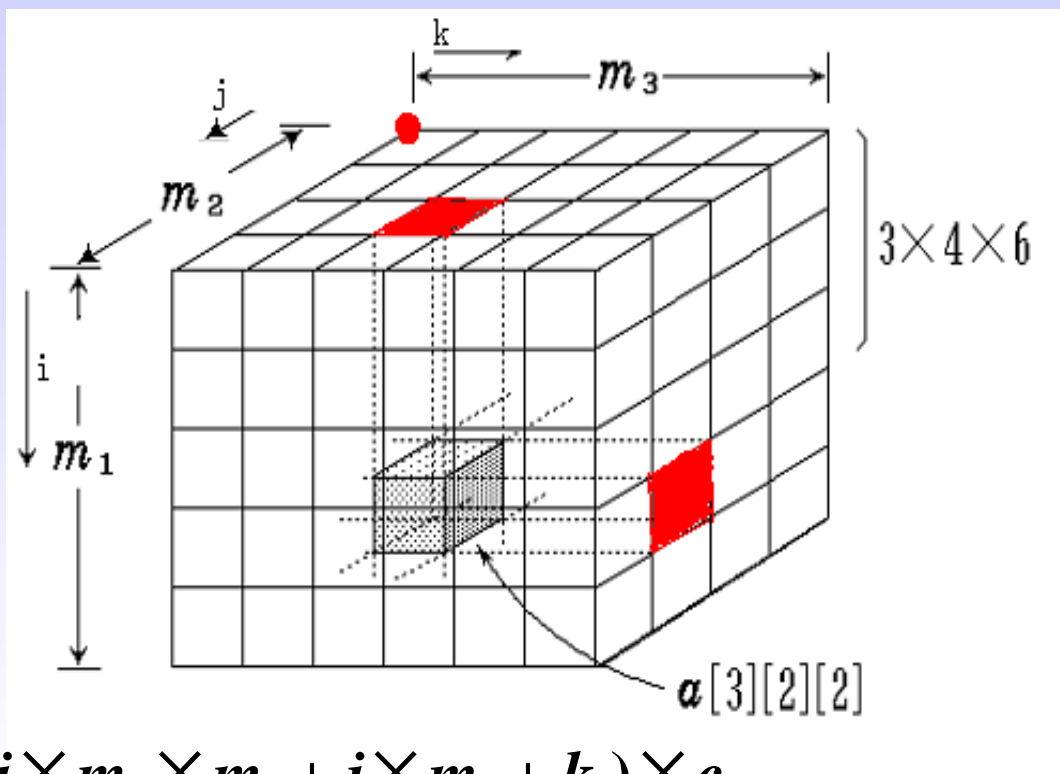
$$\text{Loc}(a_{ij}) = \text{Loc}(a_{l_1l_2}) + ((i - l_1) \times (h_2 - l_2 + 1) + (j - l_2)) \times c$$

按列优先存储的寻址方法与此类似。

5.2 数组的表示与实现

数组的存储结构与寻址——多维数组

n ($n > 2$) 维数组一般也采用按行优先和按列优先两种存储方法。请自行推导任一元素存储地址的计算方法。



$$\text{Loc}(a_{ijk}) = \text{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

$$\begin{aligned} \text{Loc}(j_1, j_2, \dots, j_n) = & \text{Loc}(0, 0, \dots, 0) + (j_1 \times m_2 \times \dots \times m_n \\ & + j_2 \times m_3 \times \dots \times m_n \\ & + \dots + j_n) \times c \end{aligned}$$

- 数组的基本操作实现

```
status InitArray(Array &A, int dim, ...)
```

```
{  
    if (dim < 1) || dim > MAX_ARRAY_DIM) return Error;  
    A.dim = dim;  
    A.bounds = (int *)malloc(dim*sizeof(int));  
    if (!A.bounds) exit(Overflow)  
    elemtotal = 1;  
    va_list ap;  
    va_start(ap, dim);  
    for (i = 0; i < dim; ++i) {  
        A.bounds[i] = va_arg(ap, int);  
        if (A.bounds[i] < 0) return Underflow;  
        elemtotal *= A.bounds[i];  
    }  
    va_end(ap);
```

VA_LIST 是在C语言中解决变参问题的一组宏

```
A.base = (ElemType *)malloc(elemtotal * sizeof(ElemType ));  
if (!A.base ) exit (Overflow);  
A.constants = (int *) malloc(dim*sizeof (int));  
if (!A.constants) exit(Overflow);  
A.constants[dim-1] = 1;  
for (i = dim-2; i >= 0; --i)  
    A.constants[i] = A.bounds[i+1] * A.constants[i+1] ;  
return OK;  
}
```

```
status DestroyArray(Array &A)
{
    if (!A.base)      return Error;
    free(A.base);     A.base = NULL;
    if (!A.bounds)    return Error;
    free(A.bounds);   A.bounds = NULL;
    if (!A.constants) return Error;
    free(A.constants); A.constants = NULL;
    return OK;
}
```

```
Status Locate(Array, va_list ap, int &off)
{ // 若ap指示的各下标值合法，求该元素在A中的相对地址off
  off = 0;
  for (i = 0; i < A.dim; ++i)
  { ind = va_arg(ap, int); //ind 表示元素的某个下标值;
    if (ind < 0 || ind >= A.bounds[i]) return Overflow;
    off += A.constants[i] * ind;
  }
  return OK;
}
```

```
status value(Array A, ElemType &e, ...) {  
    // 可变参数是n个下标值;  
    // 若各下标不超界, 则为e赋值, 并返回OK;  
    va_start(ap, e);  
    if ((result = Locate(A, ap, off)) <= 0 return result;  
    e = *off;  
    return OK;  
}
```

```
status value(Array A, ElemType &e, ...) {  
    // 可变参数是n个下标值;  
    // 若各下标不超界, 则为e赋值, 并返回OK;  
    va_start(ap, e);  
    if ((result = Locate(A, ap, off)) <= 0 return result;  
    e = *off;  
    return OK;  
}
```

```
status value(Array A, ElemType &e, ...) {  
    // 可变参数是n个下标值;  
    // 若各下标不超界, 则为e赋值, 并返回OK;  
    va_start(ap, e);  
    if ((result = Locate(A, ap, off)) <= 0 return result;  
    e = *(A.base + off*sizeof(ElemType ));  
    return OK;  
}
```

5.3 矩阵的压缩存储

特殊矩阵和稀疏矩阵

特殊矩阵：矩阵中很多值相同的元素并且它们的分布有一定的规律。

稀疏矩阵：矩阵中有很多零元素或值相同的元素，其分布没有规律。

压缩存储的基本思想是：

- (1) 为多个值**相同**的元素只分配**一个**存储空间；
- (2) 对**零**元素**不分配**存储空间。

5.3 矩阵的压缩存储——特殊矩阵

1 特殊矩阵的压缩存储——对称矩阵

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

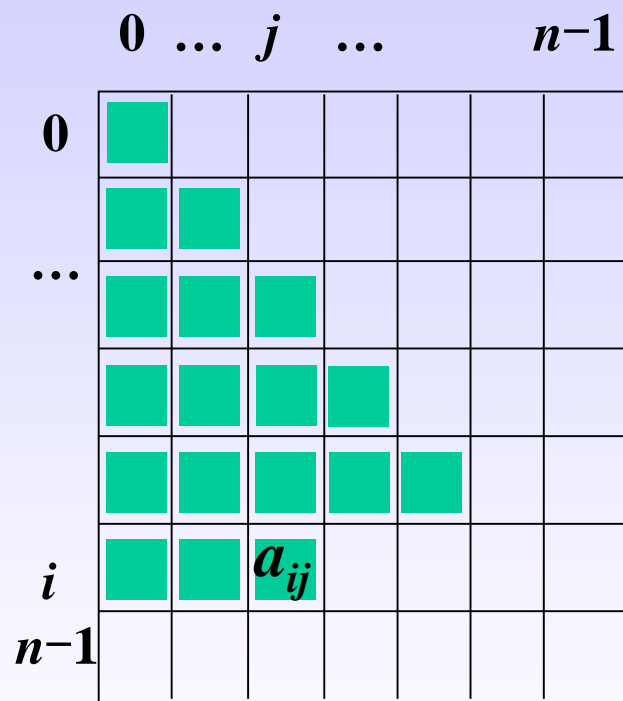
对称矩阵特点: $a_{ij}=a_{ji}$

① 如何压缩存储?

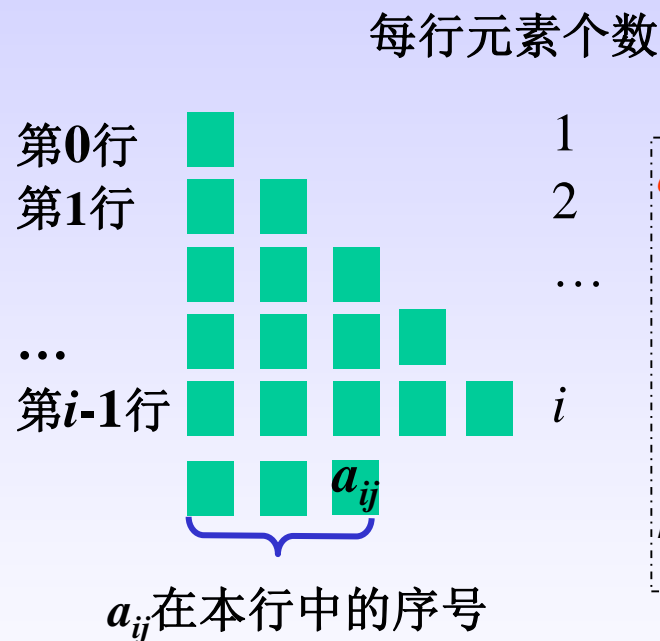
只存储下三角部分的元素（含对角线）。

5.3 矩阵的压缩存储—特殊矩阵

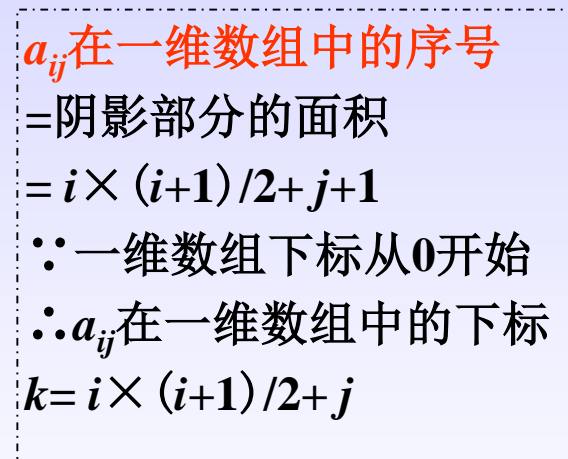
对称矩阵的压缩存储



(a) 下三角矩阵



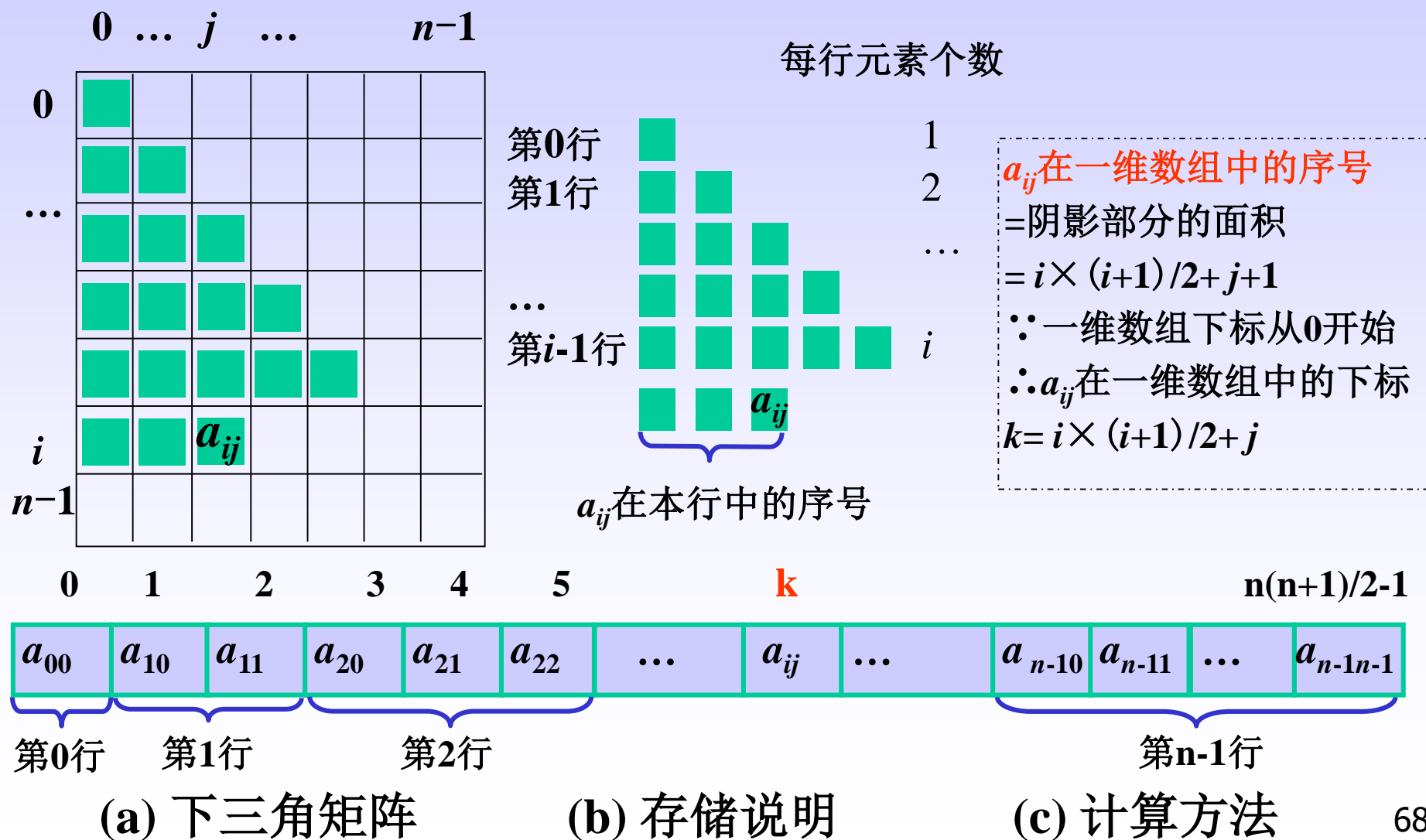
(b) 存储说明



(c) 计算方法

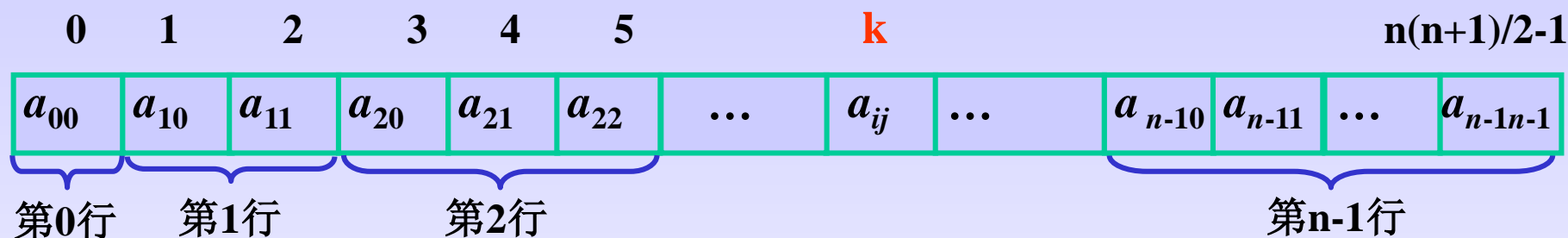
5.3 矩阵的压缩存储—特殊矩阵

对称矩阵的压缩存储



5.3 矩阵的压缩存储——特殊矩阵

对称矩阵的压缩存储



对于下三角中的元素 a_{ij} ($i \geq j$)，在数组SA中的下标 k 与 i 、 j 的关系为： $k = i \times (i+1)/2 + j$ 。

上三角中的元素 a_{ij} ($i < j$)，因为 $a_{ij} = a_{ji}$ ，则访问和它对应的元素 a_{ji} 即可，即： $k = j \times (j+1)/2 + i$ 。

5.3 矩阵的压缩存储——特殊矩阵

2 特殊矩阵的压缩存储——三角矩阵

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

(a) 下三角矩阵

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

(b) 上三角矩阵

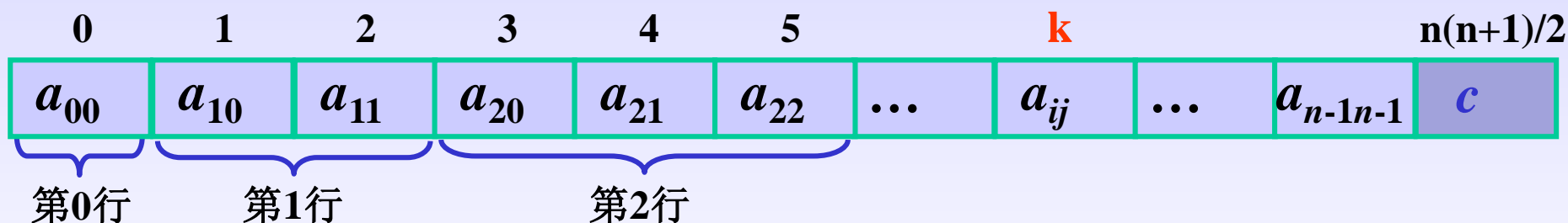
① 如何压缩存储？

只存储上三角（或下三角）部分的元素（含对角线）。

5.3 矩阵的压缩存储——特殊矩阵

下三角矩阵的压缩存储

存储 { 下三角元素
对角线上方的常数——只存一个



矩阵中任一元素 a_{ij} 在数组中的下标 k 与 i 、 j 的对应关系:

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

5.3 矩阵的压缩存储——特殊矩阵

上三角矩阵的压缩存储

存储 { 上三角元素
对角线下方的常数——只存一个

矩阵中任一元素 a_{ij} 在数组中的下标 k 与 i 、 j 的对应关系:

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

5.3 矩阵的压缩存储——特殊矩阵

上三角矩阵的压缩存储

存储 { 上三角元素
对角线下方的常数——只存一个

矩阵中任一元素 a_{ij} 在数组中的下标 k 与 i 、 j 的对应关系:

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

5.3 矩阵的压缩存储——特殊矩阵

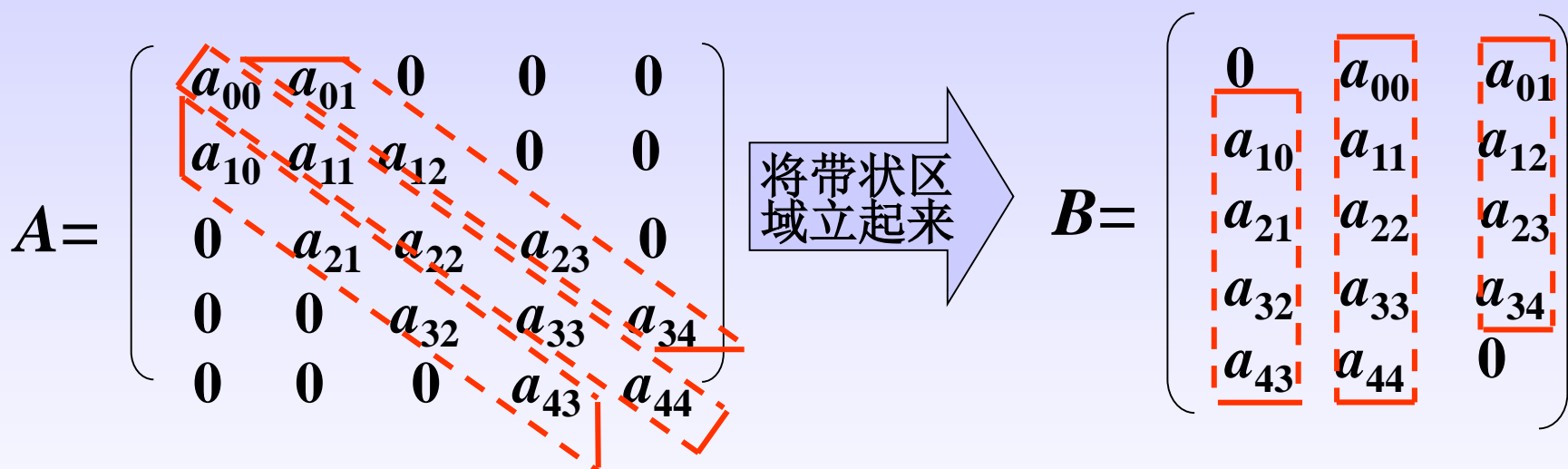
3 特殊矩阵的压缩存储——对角矩阵

对角矩阵：所有非零元素都集中在以主对角线为中心的带状区域中，除了主对角线和它的上下方若干条对角线的元素外，所有其他元素都为零。

$$A = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

5.3 矩阵的压缩存储——特殊矩阵

对角矩阵的压缩存储



还有什么方法？

映射到二维数组B中，映射关系 $\begin{cases} t=i \\ s=j-i+1 \end{cases}$

5.3 矩阵的压缩存储——特殊矩阵

对角矩阵的压缩存储

$$A = \begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & 0 \\ 0 & 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{43} & a_{44} \end{pmatrix}$$

(a) 三对角矩阵



0	1	2	3	4	5	6	7	8	9	10	11	12
a_{00}	a_{01}	a_{10}	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	a_{43}	a_{44}

(c) 压缩到一维数组中

元素 a_{ij} 在一维数组中的序号
 $= 2 + 3(i-1) + (j-i+2)$
 $= 2i + j + 1$

∵ 一维数组下标从0开始

∴ 元素 a_{ij} 在一维数组中的下标
 $= 2i + j$

(b) 寻址的计算方法

还有什么方法?