

第四章 字符串和数组

本章的基本内容是：

4.1 串

4.2 模式匹配

4.3 多维数组

4.4 矩阵的压缩存储

4.1 字符串 (string)

- **字符串**：零个或多个**字符**组成的有限**序列**（简称串）。
- **串长度**：串中所包含的字符个数。
- **空串**：长度为0的串，记为：""。
- **特殊的线性表**，即元素为字符的线性表。
- **非空串**通常记为：

$$S = " s_1 s_2 \dots s_n "$$

其中：S是串名，双引号是**定界符**，双引号引起来的部分是串值， s_i ($1 \leq i \leq n$) 是一个任意字符。

字符/符号

- **字符(char)**：组成字符串的基本单位
- 取值依赖于**字符集** Σ （结点的有限集合）
 - 二进制字符集： $\Sigma = \{0,1\}$
 - 生物信息中DNA字符集： $\Sigma = \{A,C,G,T\}$
 - 英语语言： $\Sigma = \{26\text{个字符}, \text{标点符号}, \dots\}$
 - 简体中文标准字符集 GB2312： $\Sigma = \{6763\text{个汉字}, \text{标点符号}, \dots\}$
 -

字符编码

□ASCII编码

- 单字节（8 bits）
- 对128个符号（字符集charset）进行编码
- 在C和C++中均采用

□其他编码方式

- ANSI编码（本地化，GB2312、BIG5、JIS等，不同ANSI编码间互不兼容）
- UNICODE（国际化，各种语言中的每一个字符具有唯一的数字编号，便于跨平台的文本转换）

字符串

- 子串：串中任意个连续的字符组成的子序列。
- 主串：包含子串的串。
- 子串的位置：子串的第一个字符在主串中的序号。

$S1 = "ab12cd"$

$S2 = "ab12"$

$S3 = "ab13"$

$S4 = "ab12\varphi"$

$S5 = ""$

$S6 = "\varphi\varphi\varphi"$

字符串

□串的比较：通过组成串的字符之间的比较来进行的。

给定两个串： $X="x_1x_2\cdots x_n"$ 和 $Y="y_1y_2\cdots y_m"$ ，则：

1. 当 $n=m$ 且 $x_1=y_1, \cdots, x_n=y_m$ 时，称 $X=Y$ ；

2. 当下列条件之一成立时，称 $X<Y$ ：

(1) $n<m$ 且 $x_i=y_i$ ($1\leq i\leq n$) ；

(2) 存在 $k\leq\min(m, n)$ ，使得 $x_i=y_i$ ($1\leq i\leq k-1$) 且 $x_k<y_k$ 。

例： $S1="ab12cd"$ ， $S2="ab12"$ ， $S3="ab13"$

字符串



与其他数据结构相比，串的操作对象有什么特点？

串的操作通常以串的整体作为操作对象。

字符串

求子串操作SubStr(s, i, len)示例

$i = 3, \text{len} = 3$

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> | <i>e</i> | <i>f</i> | <i>g</i> | <i>e</i> |
|----------|----------|----------|----------|----------|----------|----------|----------|



| | | |
|----------|----------|----------|
| <i>c</i> | <i>d</i> | <i>e</i> |
|----------|----------|----------|

$i = 7, \text{len} = 4$

| | | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|----------|
| <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> | <i>e</i> | <i>f</i> | <i>g</i> | <i>e</i> |
|----------|----------|----------|----------|----------|----------|----------|----------|



| | |
|----------|----------|
| <i>g</i> | <i>e</i> |
|----------|----------|

空串

字符串长度

- 理论上，一个字符串的长度是任意且有限的，但在实际的语言中总有一定的长度
 - 定长：具有一个固定的最大长度，所用内存量始终如一
 - 变长：根据实际需要伸缩。尽管命名为变长，但实际长度也有限（取决于可用的内存量）

字符串数据类型

□ 因语言而不同

- 简单类型
- 复合类型

□ 字符串常数和变量

- 字符串常数 (**string literal**)
 - 例如: “\n”, “a”, “student”...
- 字符串变量

C++的标准string

□ **标准字符串：** 将C/C++的<string.h>函数库作为字符串数据类型的方案

- 例如： `char S[M];`定义了字符串变量
e.g., `char s1[7] = "value";`

□ **串的结束标记：** `'\0'`

- `'\0'`是ASCII码中8位BIT全0码，又称为**NULL**符，专门用于结束标志
- 字符串的实际长度为 **M-1**

C++标准string

❑ 串长函数

```
int strlen(char *s);
```

❑ 串复制

```
char *strcpy(char *s1, char*s2);
```

❑ 串拼接

```
char *strcat(char *s1, char *s2);
```

❑ 串比较

```
int strcmp(char *s1, char *s2);
```

❑ 输入和输出函数

```
cin>>      cout<<
```

❑ 定位函数

```
char *strchr(char *s, char c);
```

❑ 右定位函数

```
char *strrchr(char *s, char c);
```

String抽象数据类型

□字符串类（**class String**）：

- 适应字符串长度动态变化的复杂性
- 不再以字符数组**char S[M]**的形式出现，而采用一种动态变长的存储结构

C++ String 部分操作列表

| 操作类别 | 方法 | 描述 |
|-------|--------------------|------------------------|
| 子串 | substr () | 返回一个串的子串 |
| 拷贝/交换 | swap () | 交换两个串的内容 |
| | copy () | 将一个串拷贝到另一个串中 |
| 赋值 | assign () | 把一个串、一个字符、一个子串赋值给另一个串中 |
| | = | 把一个串或一个字符赋值给另一个串中 |
| 插入/追加 | insert() | 在给定位置插入一个字符、多个字符或串 |
| | += | 将一个字符或串追加到另一个串后 |
| | append () | 将一个或多个字符、或串追加在另一个串后 |
| 拼接 | + | 通过将一个串放置在另一个串后面来构建新串 |
| 查询 | find () | 找到并返回一个子序列的开始位置 |
| 替换/清除 | replace () | 替换一个指定字符或一个串的字串 |
| | clear () | 清除串中的所有字符 |
| 统计 | size () | 返回串中字符的数目 |
| | length () | 返回 size () |
| | max_size () | 返回串允许的最大长度 |

字符串的存储结构和实现

- 字符串的顺序存储
- 字符串类class String的存储结构
- C++标准串的运算实现
- String类的运算实现

字符串的顺序存储

□ 对于串长变化不大的字符串，可以有三种处理方案：

(1) 用S[0]作为记录串长的存储单元 (Pascal)

- 缺点：限制了串的最大长度不能超过256

(2) 为存储串的长度，另辟一个存储的地方

- 缺点：串的最大长度一般是静态给定的，不是动态申请数组空间

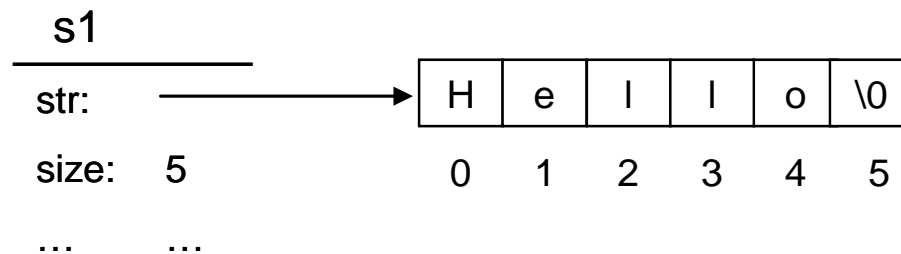
(3) 用一个特殊的末尾标记'\0' (C/C++)

- 例如：C++语言的string函数库 (`#include <string.h>`) 采用这一存储结构

字符串类class String的存储结构

```
private:           // 具体实现的字符串存储结构
    char *str;      // 字符串的数据表示
    int size;       // 串的当前长度
```

例如，
String s1 = "Hello";



字符串运算的算法实现

1. 串长函数

```
int strlen(char *s);
```

2. 串复制

```
char *strcpy(char *d, char*s);
```

3. 串拼接

```
char *strcat(char *s1, char *s2);
```

4. 串比较

```
int strcmp(char *s1, char *s2);
```

5. 寻找字符

```
char * strchr(char *d, char ch)
```

```
char * strrchr(char *d, char ch)
```

C++标准串运算的实现

● 字符串的复制

```
char *strcpy(char *d, char *s)
{ //这个程序的毛病是，如果字符串s比字符串d要
  长，
  //这个程序没有检查拷贝出界，没有报告错误。
  //可能会造成d的越界
    int i = 0;
    while (s[i] != '\0') {
        d[i] = s[i];  i++;
    }
    d[i] = '\0';
    return d;
}
```

C++标准串运算的实现(续)

- 字符串的比较

```
int strcmp( char *d, char *s)  
{  
    int i =0;  
    while (s[i] != '\0' && d[i] != '\0' )  
    {  
        if (d[i] > s[i])  
            return 1;  
        else if (d[i] < s[i])  
            return -1;
```

C++标准串运算的实现(续)

```
    i++;  
}  
if( d[i] == '\0' && s[i] != '\0')  
    return -1;  
else if (s[i] == '\0' && d[i] != '\0')  
    return 1;  
return 0;  
}
```

C++标准串运算的实现(续)

- 求字符串的长度

```
int strlen(char d[])  
{  
    int i =0;  
    while (d[i] != 0)  
        i++;  
    return i;  
}
```

C++标准串运算的实现(续)

● 寻找字符

char * strchr(char *d , char ch)

{

//按照数组指针d依次寻找字符ch,

//如果找到ch, 则将指针位置返回,

//如果没有找到ch, 则为0值。

i = 0;

C++标准串运算的实现(续)

- 反向寻找字符

```
char * strrchr(char *d , char ch)
```

```
{
```

```
    //按照数组指针d，从其尾部反着寻找字符ch，
```

```
    //如果找到ch，则将指针位置返回，
```

```
    //如果没有找到ch，则为0值。
```

```
    i = 0;
```

```
        //找串尾
```

```
    while (d[i] != '\0' )
```

```
        i++;
```


C++标准串运算的实现(续)

```
//循环跳过那些不是ch的字符
while (i >= 0 && d[i] != ch )
    ;
    //当本串不含字符ch，则在串尾结束；
    //当成功寻找到ch，返回该位置指针
if (i < 0)
    return 0;
else
    return &d[i] ;
}
```

C++标准串运算的实现(续)

//循环跳过那些不是ch的字符

```
while (d[i] != 0 && d[i] != ch )      i++;
```

//当本串不含字符ch，则在串尾结束；

//当成功寻找到ch，返回该位置指针

```
if (d[i] == 0)
```

```
    return 0;
```

```
else
```

```
    return &d[i] ;
```

```
}
```

C++标准string

比如, 字符串s :

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|----|----|
| s | H | e | l | l | o | | w | o | r | l | d | \0 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| | | | | | ↑ | | | ↑ | | | | |

strchr(s,'o')

strrchr(s, 'o')

寻找字符o, strchr(s,'o')结果返回4;

反方向寻找r, strrchr(s,'o')结果返回7

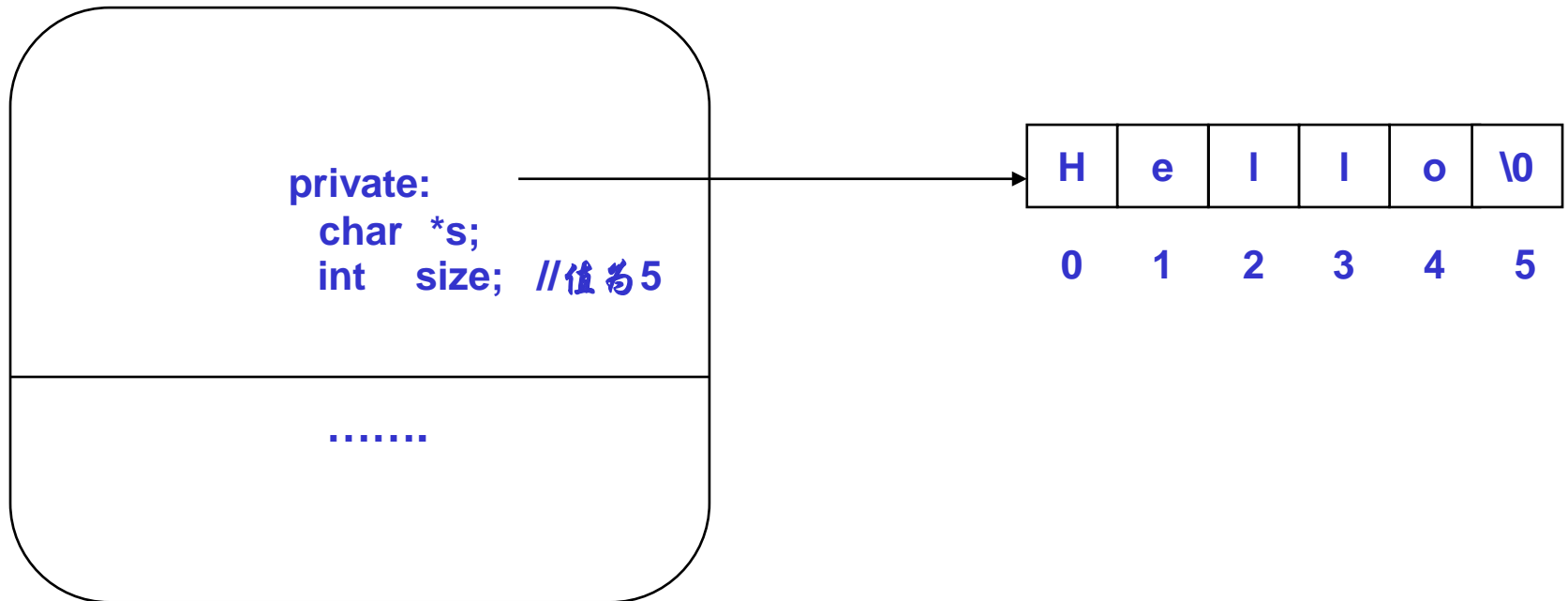
String串运算的实现

```
String::String(char *s) {  
    // 先要确定新创字符串实际需要的存储空间，s的类型为(char *)，  
    // 作为新创字符串的初值。确定s的长度，用标准字符串函数  
    // strlen(s)计算长度  
    size = strlen(s);  
  
    // 然后，在动态存储区域开辟一块空间，用于存储初值s，把结束  
    // 字符也包括进来  
    str = new char [size+1];  
    // 开辟空间不成功时，运行异常，退出  
    assert(str != '\0');  
  
    // 用标准字符串函数strcpy，将s完全复制到指针str所指的存储空间  
    strcpy(str, s);  
}
```

String 串的创建运算

String s1 = "Hello" ;

s1



String串运算的实现

// 析构函数

```
String::~String() {  
    // 必须释放动态存储空间  
    delete [] str;  
}
```

String串运算的实现

// 赋值算子

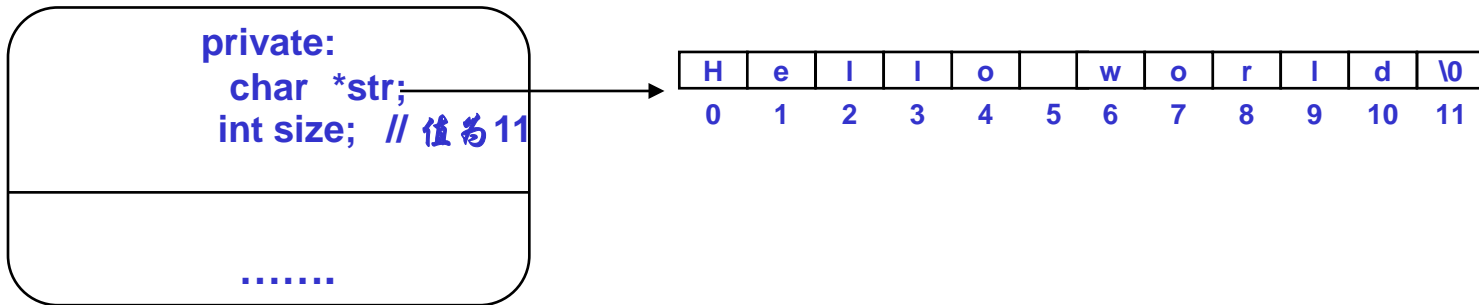
```
String String::operator= (String& s) {  
    // 参数 s 将被赋值到本串。若本串的串长和s的串长不同，则应该释放本串的  
    // str存储空间，并开辟新的空间  
    if (size != s.size) {  
        delete [] str ;           // 释放原存储空间  
        str = new char [s.size+1];  
        // 若开辟动态存储空间失败，则退出正常运行  
        assert(str != 0);  
        size = s.size;  
    }  
    strcpy(str , s.str );  
    // 返回本实例，作为String类的一个实例  
    return *this;  
}
```

String的赋值运算

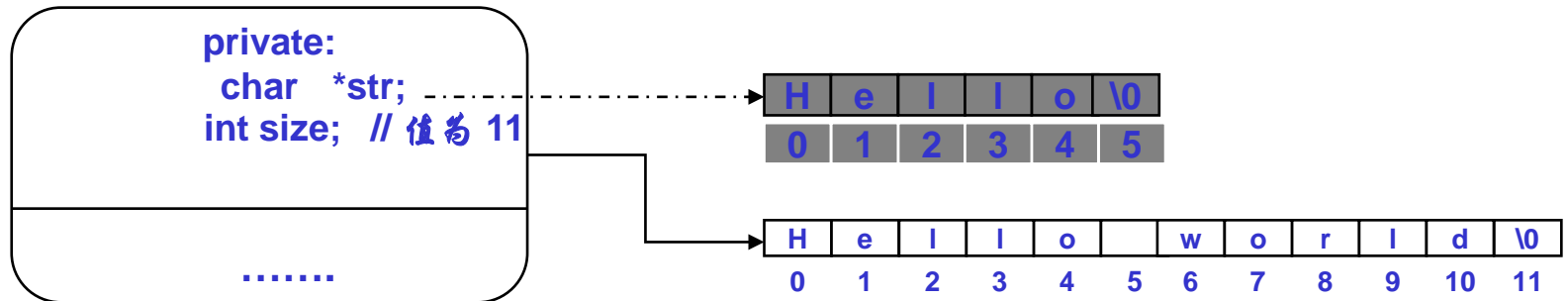
String s2 = "Hello world";

并通过赋值语句: s1 = s2;

s2



s1



String串运算的实现

// 抽取子串函数

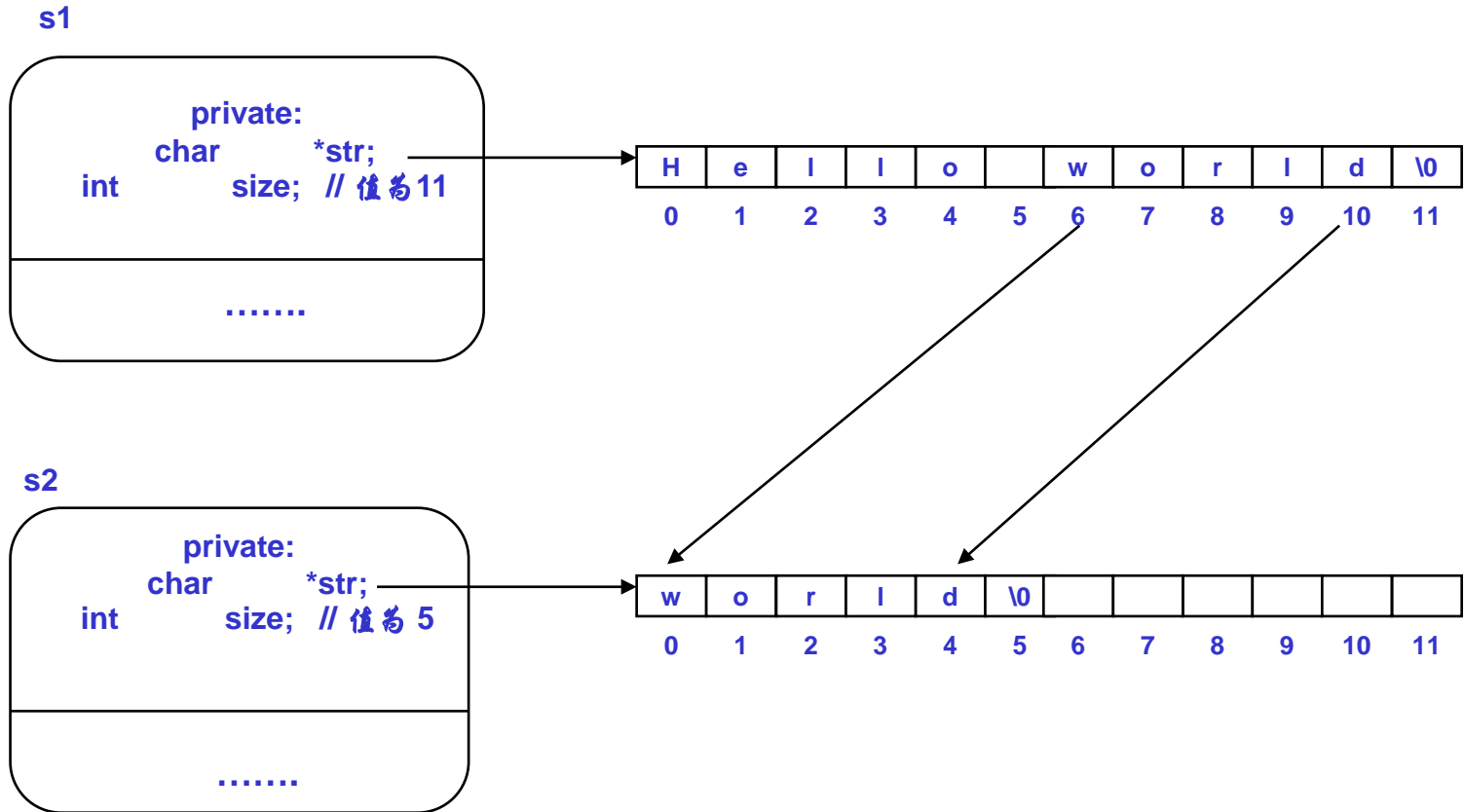
```
String String::Substr(int index , int count ) {  
    // 取出一个子串返回，自下标index开始，长度为count  
  
    int i;  
    // 本串自下标index开始向右数直到串尾，长度为left  
    int left = size - index ;  
    String temp;  
    char *p, *q;  
    // 若下标index值太大，超过本串实际串长，则返回空串  
    if (index >= size)  
        return temp;  
  
    // 若count超过自index以右的实际子串长度，则把count变小  
    if (count > left )  
        count = left;  
    // 释放原来的存储空间  
    delete [] temp.str;
```

String串运算的实现

```
// 若开辟动态存储空间失败，则退出
temp.str = new char [count+1];
assert(temp.str != 0);
// p的内容是一个指针，指向目前暂无内容的字符数组的首字符处
p = temp.str;
// q的内容是一个指针，指向本实例串的str数组的下标index字符
q = &str[index];
// 用q指针取出它所指的字符内容后，指针加1
// 用p该指针所指的字符单元接受拷贝，该指针也加1
for (i = 0; i < count; i++)
    *p++ = *q++;
// 循环结束后，让temp.str的结尾为' \0'
*p = 0;
temp.size = count;
return temp;
}
```

String抽取子串

`s2 = s1.Substr(6, 5);`



字符串模式匹配

□ 模式匹配(pattern matching)

- 一个目标对象T（字符串）
- 一个模式（pattern）P（字符串）

在目标T中寻找一个给定的模式P的过程

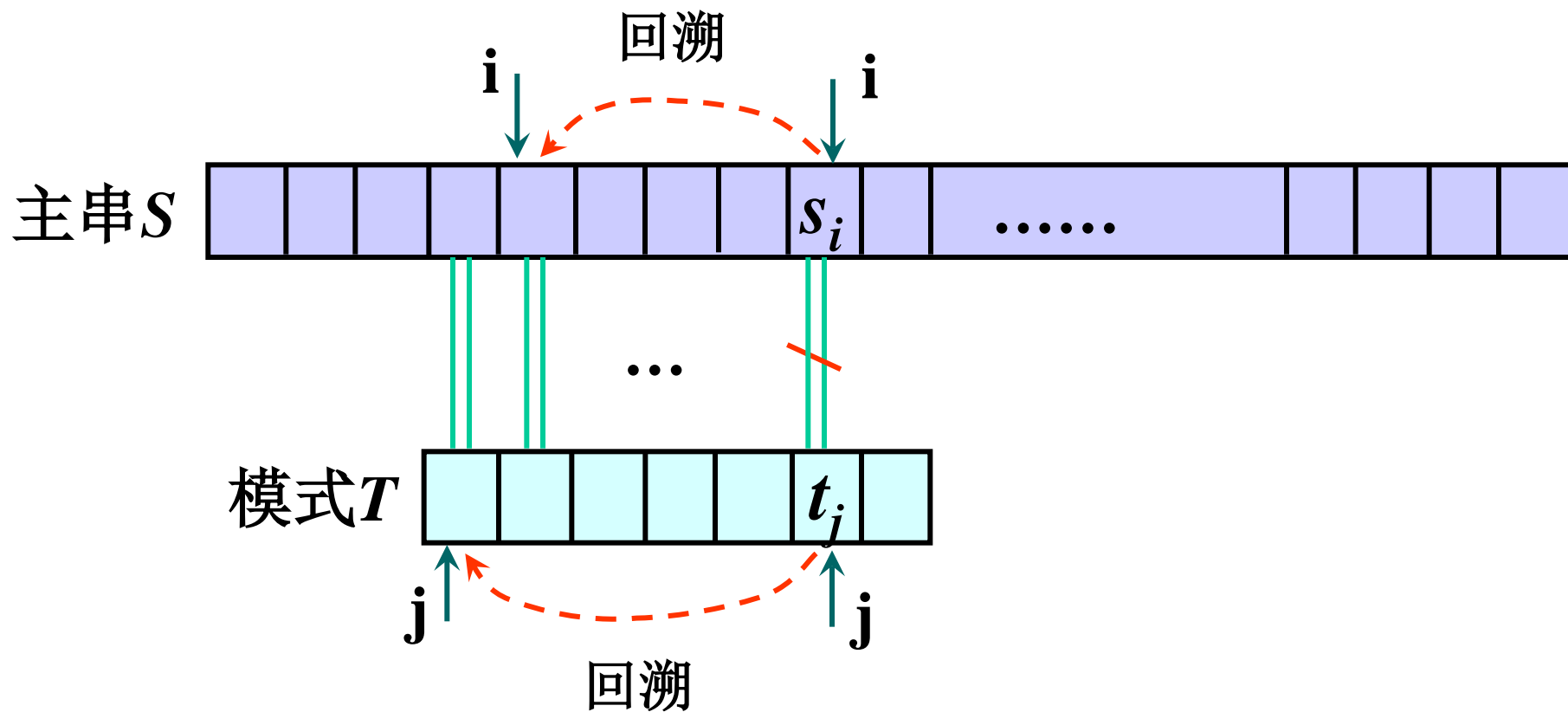
□ 应用

- 文本编辑时的特定词、句的查找
- DNA信息的提取
- 确认是否具有某种结构
- ...

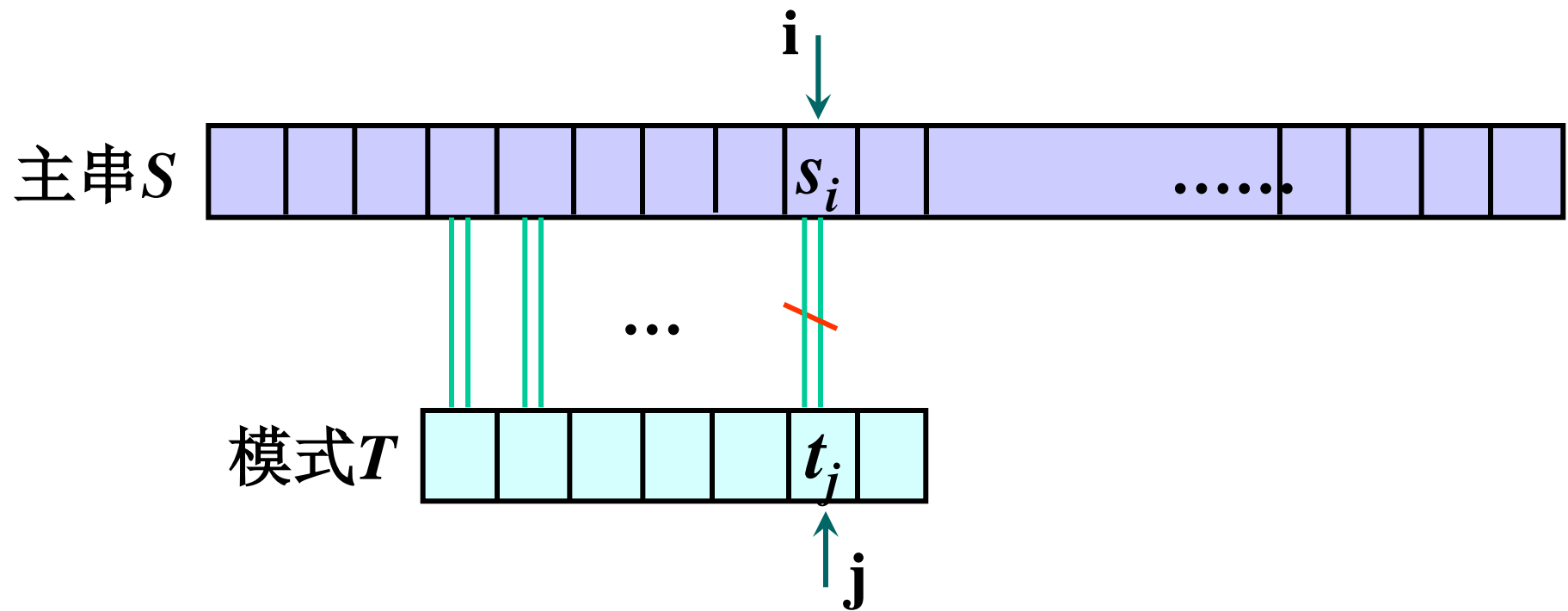
模式匹配目标

- 在大文本（诸如，句子、段落，或书本）中定位（查找）特定的模式
 - 对于大多数的算法而言，匹配的主要考虑在于其速度和效率
 - 有相当数目的算法用于解决模式匹配问题，将介绍朴素(**Brute Force**)、 **Knuth-Morris-Pratt (KMP)** 算法

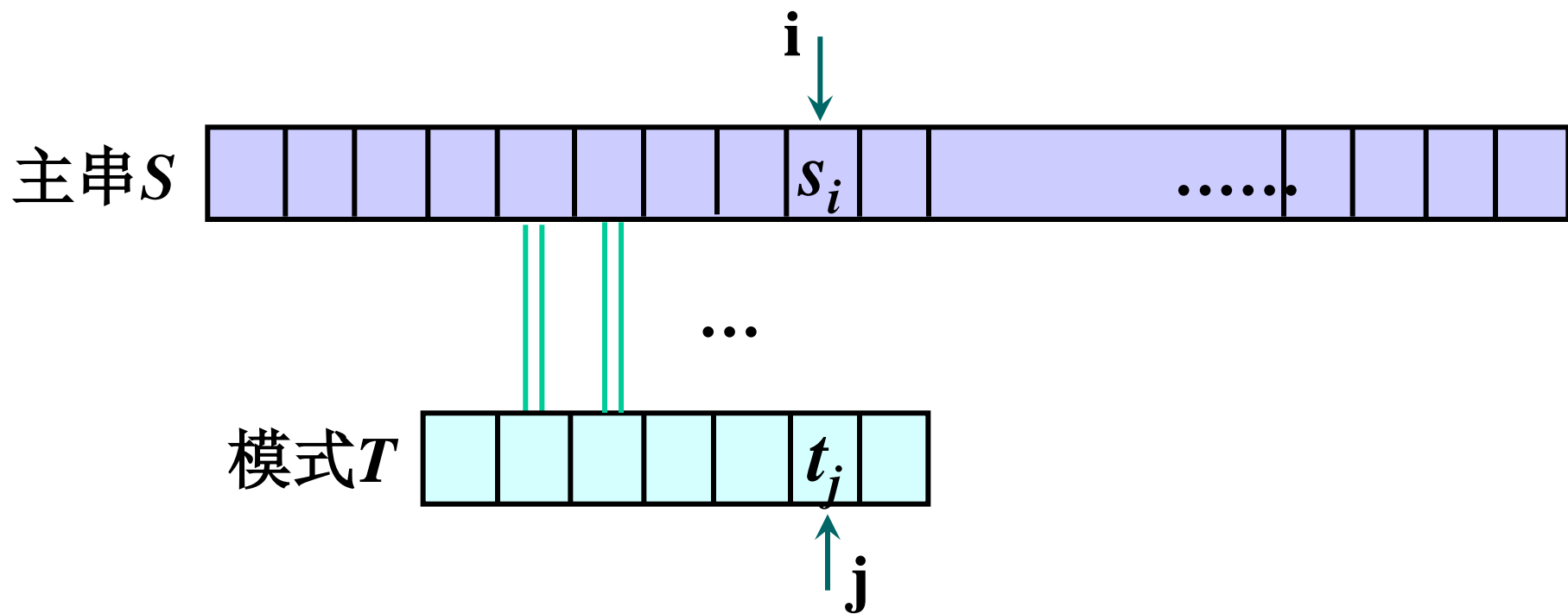
模式匹配算法



模式匹配算法



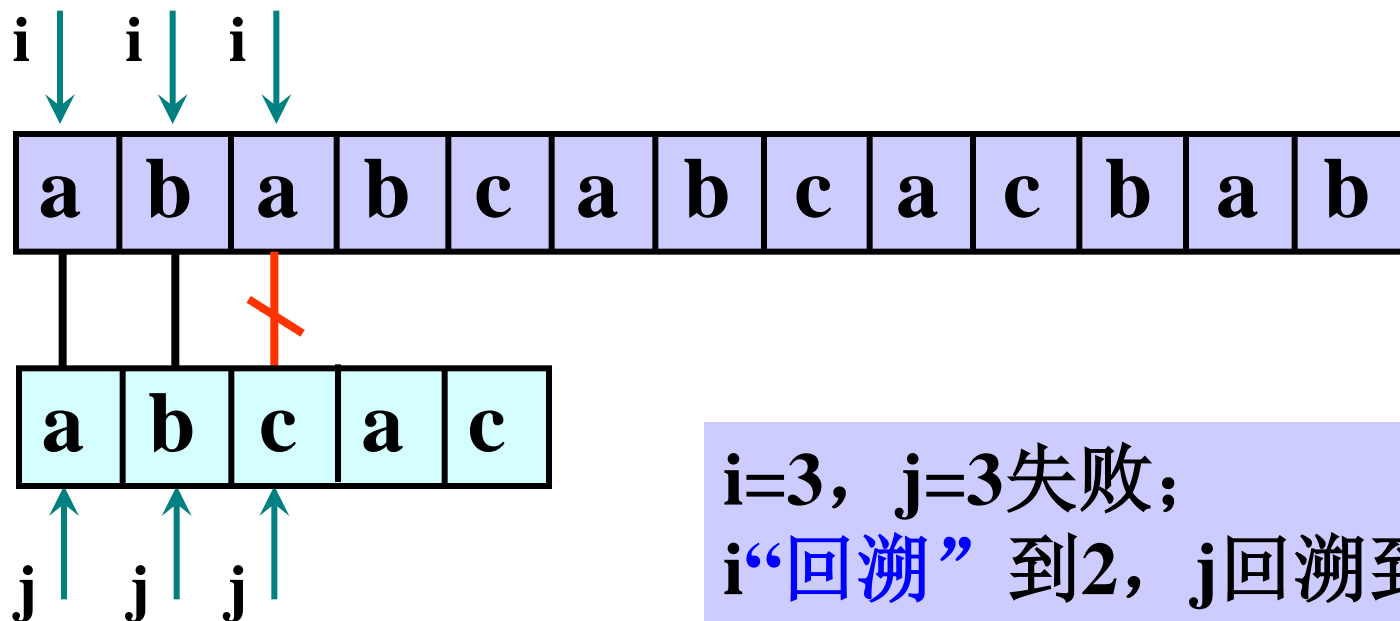
模式匹配算法



模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第
1
趟

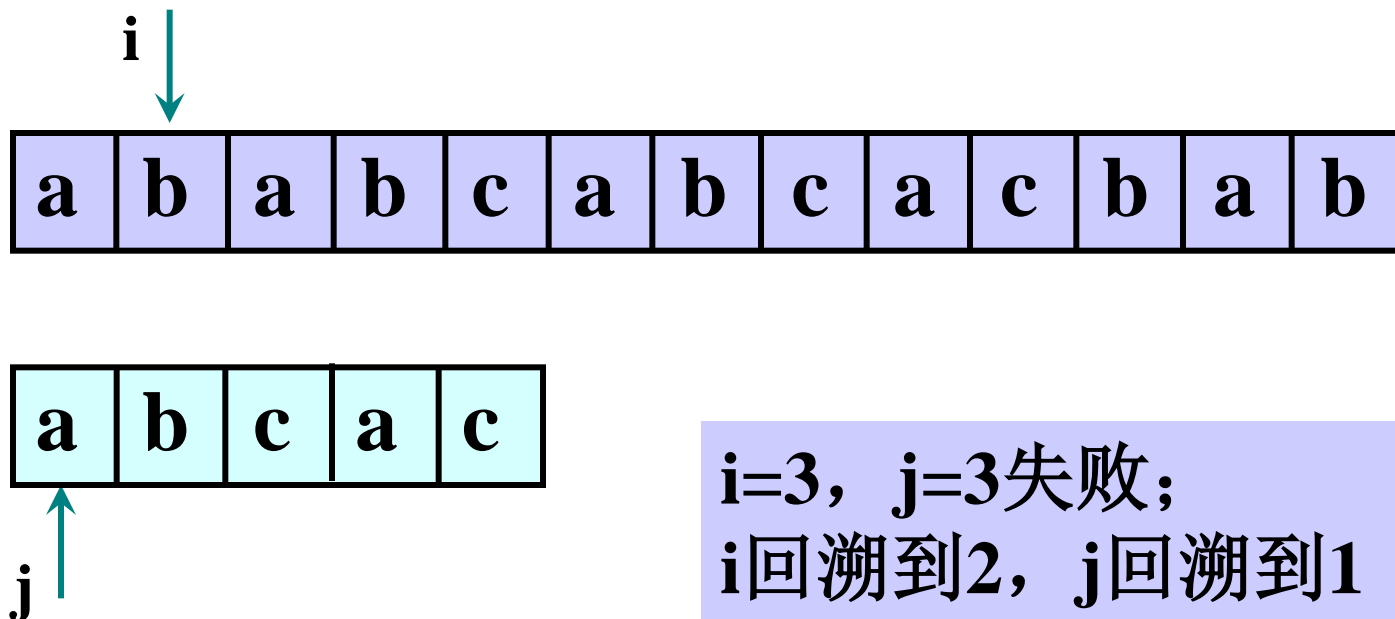


i=3, j=3失败;
i“回溯”到2, j回溯到1

模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

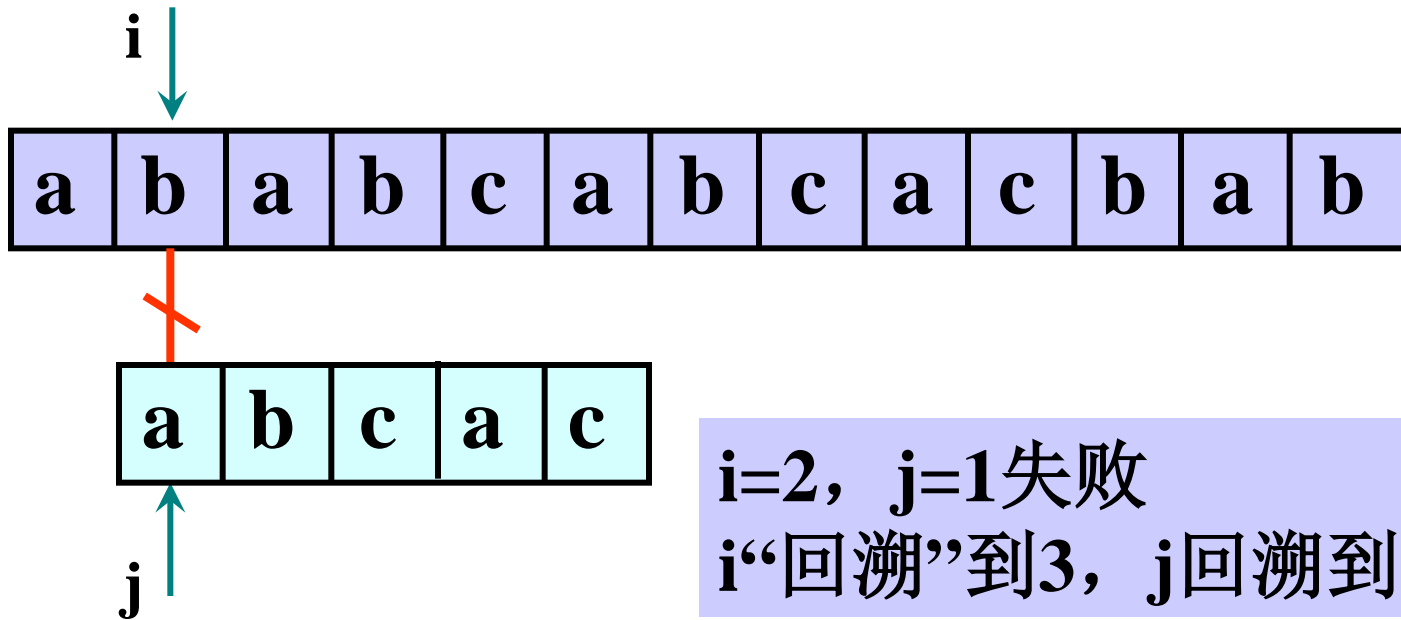
第
1
趟



模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

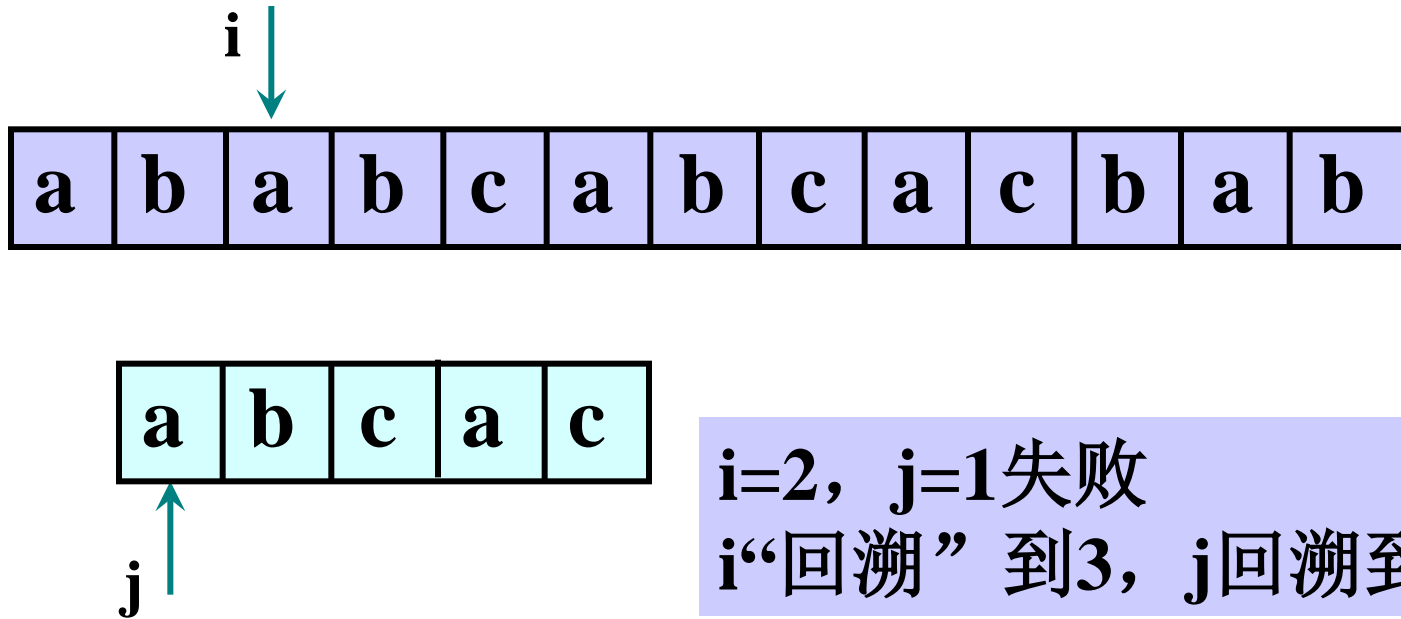
第
2
趟



模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

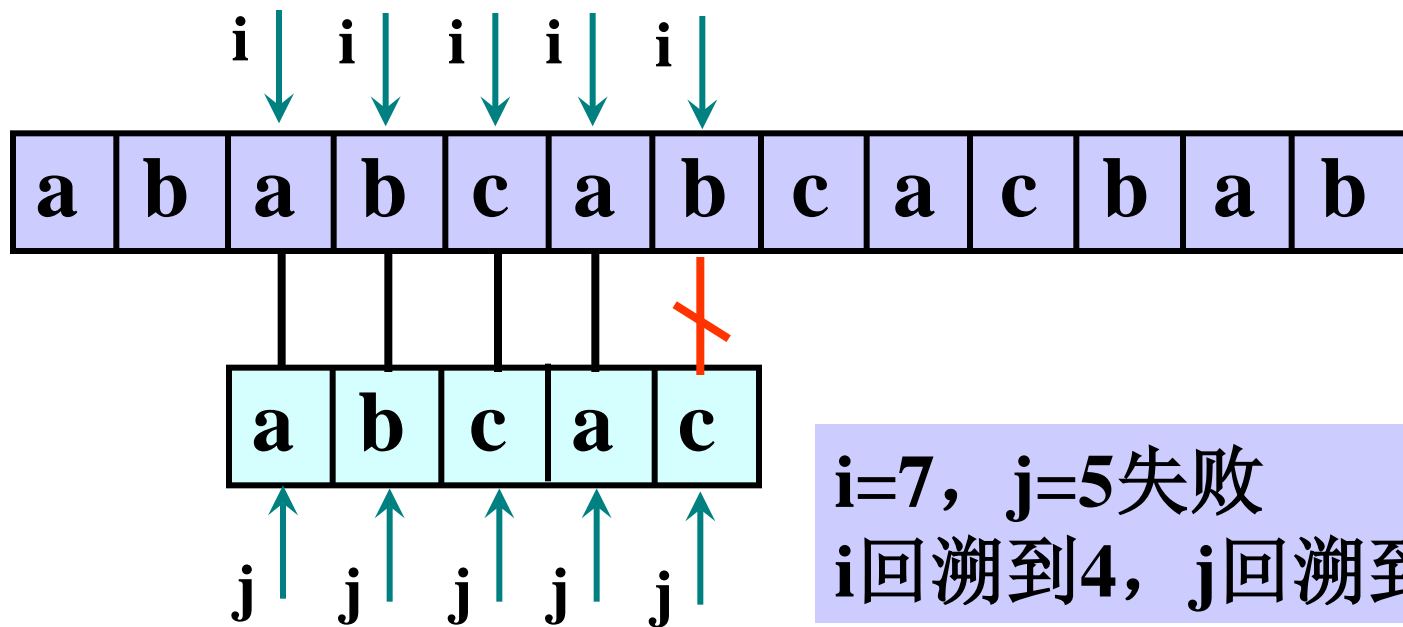
第
2
趟



模式匹配算法

例：主串S="ababcabcacbab", 模式T="abcac"

第
3
趟

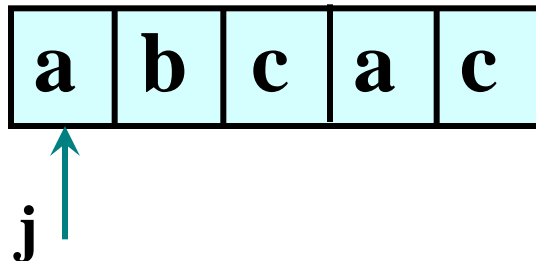
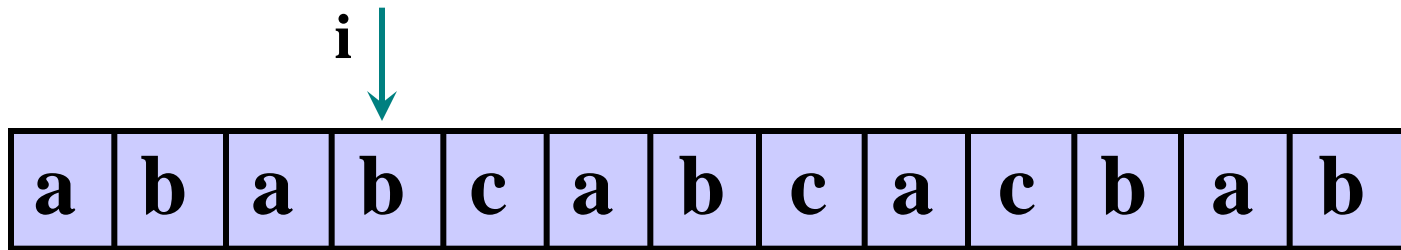


i=7, j=5失败
i回溯到4, j回溯到1

模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第
3
趟

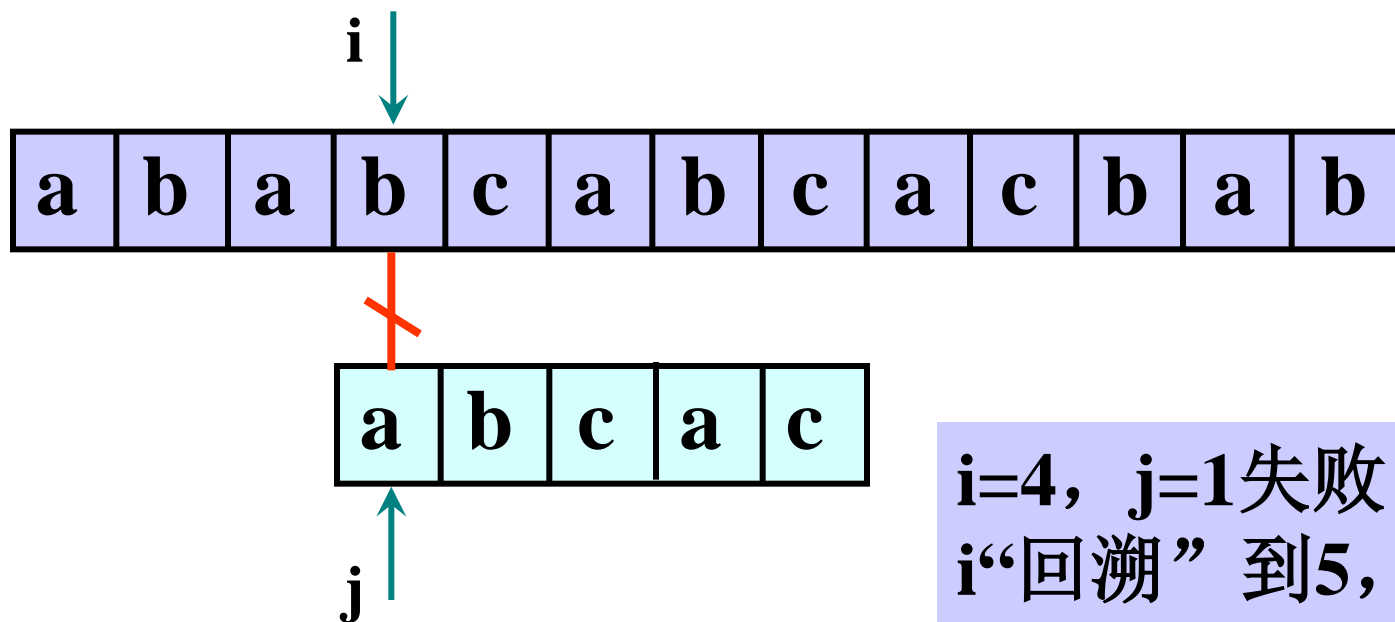


i=7, j=5失败
i回溯到4, j回溯到1

模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第
4
趟

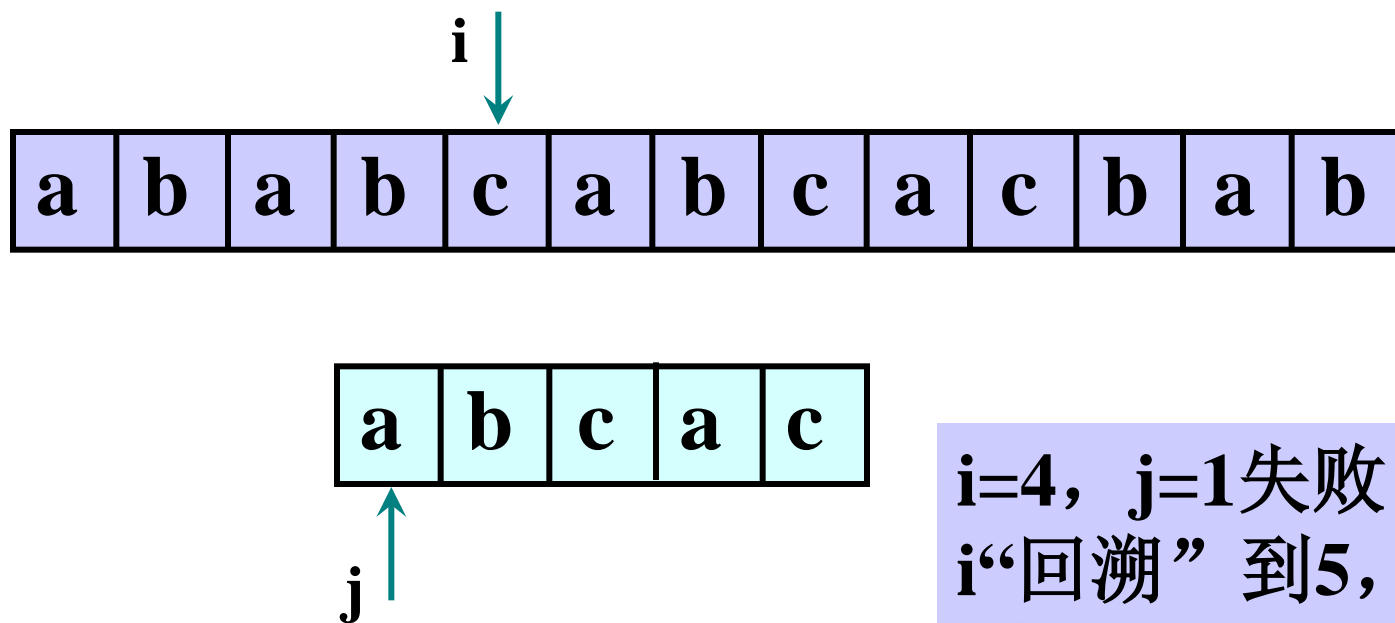


i=4, j=1失败
i“回溯”到5, j回溯到1

模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第
4
趟

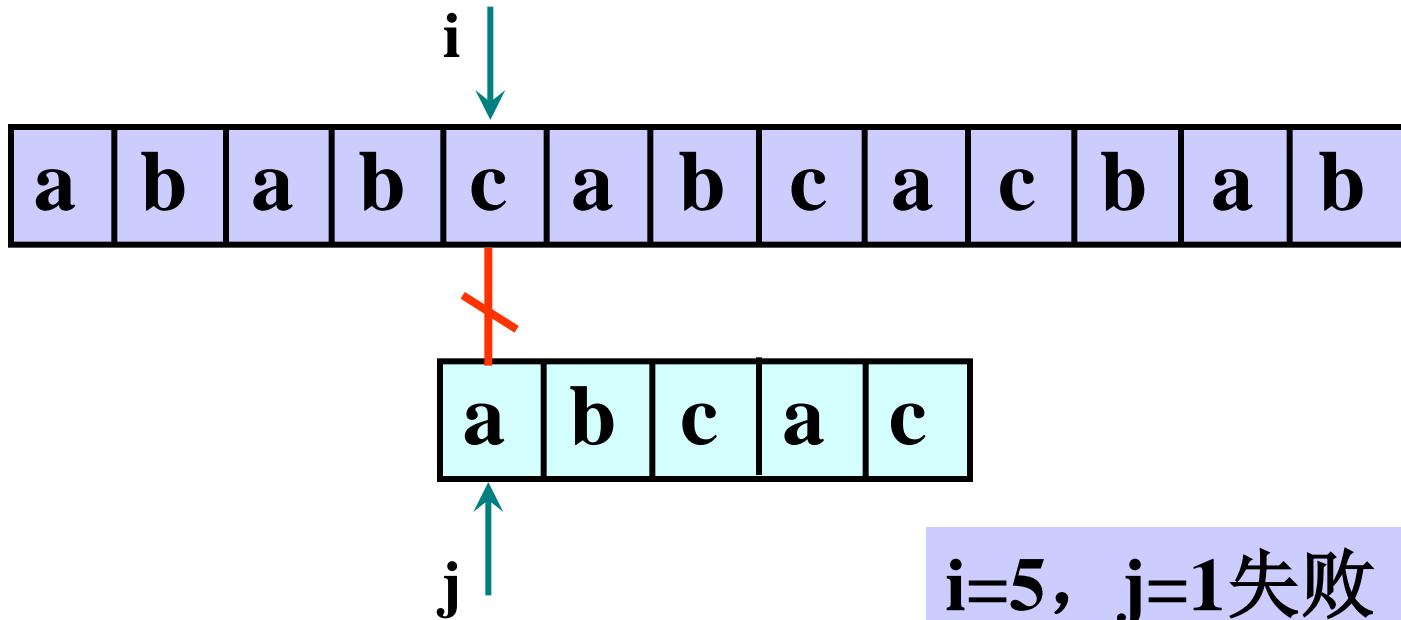


i=4, j=1失败
i“回溯”到5, j回溯到1

模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第
5
趟

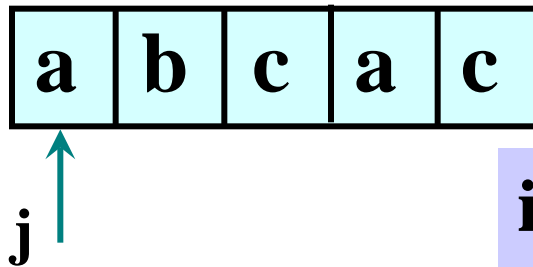
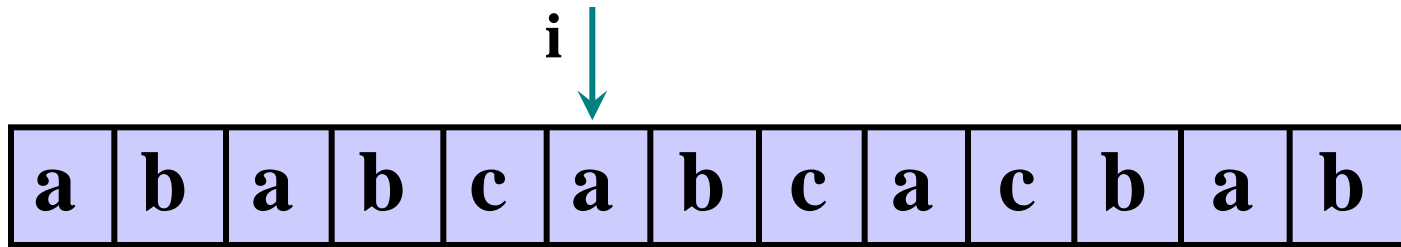


i=5, j=1失败
i“回溯”到6, j回溯到1

模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第
5
趟

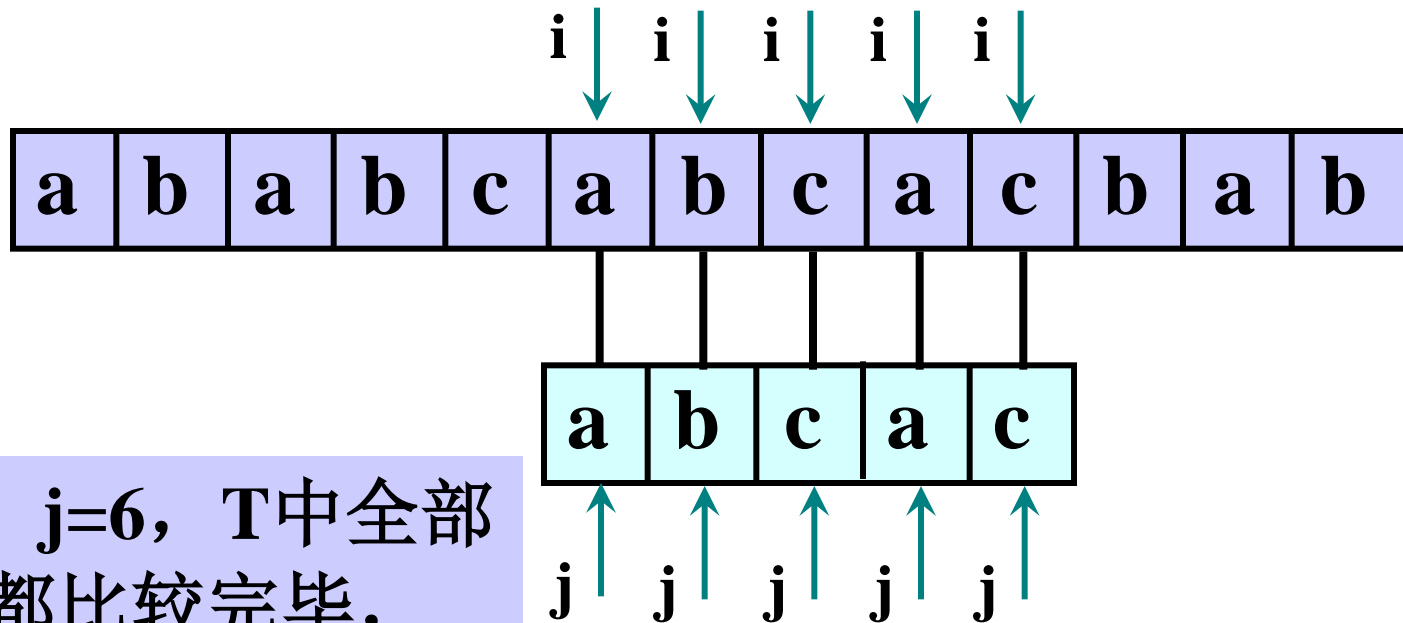


i=5, j=1失败
i“回溯”到6, j回溯
到1

模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第
6
趟



i=11, j=6, T中全部
字符都比较完毕，
匹配成功。

模式匹配算法

1. 在串S和串T中设比较的起始下标i和j;
2. 循环直到S或T的所有字符均比较完;
 - 2.1 如果 $S[i]=T[j]$, 继续比较S和T的下一个字符;
 - 2.2 否则, 将i和j回溯, 准备下一趟比较;
3. 如果T中所有字符均比较完, 则匹配成功, 返回匹配的起始比较下标; 否则, 匹配失败, 返回0;

匹配算法实现

```
#include "String.h"
#include <assert.h>
int NaiveStrMatching (String T, String P) {
    int i = 0;                // 模式的下标变量
    int j = 0;                // 目标的下标变量
    int pLen = P.length( );   // 模式的长度
    int tLen = T.length( );   // 目标的长度
    if (tLen < pLen)           // 如果目标比模式短，匹配无法成功
        return (-1);
    while ( i < pLen && j < tLen) // 反复比较对应字符来开始匹配
        if (T[j] == P[i])
            i++, j++;
        else {
            j = j - i + 1;
            i = 0;
        }
    if ( i >= pLen)
        return (j - pLen + 1);
    else return (-1);
}
```

模式匹配算法

设串 S 长度为 n ，串 T 长度为 m ，在匹配成功的情况下，考虑两种极端情况：

(1) **最好**：不成功的匹配都发生在串 T 的第一个字符。

例如： $S = \text{"aaaaaaaaaaaa}bcdccccc\text{"}$

$T = \text{"bcd"}$

模式匹配算法

设串 S 长度为 n ，串 T 长度为 m ，在匹配成功的情况下，考虑两种极端情况：

最好情况：不成功的匹配都发生在串 T 的第一个字符。

设匹配成功发生在 s_i 处，则在 $i-1$ 趟不成功的匹配中共比较了 $i-1$ 次，第 i 趟成功的匹配共比较了 m 次，所以总共比较了 $i-1+m$ 次，所有匹配成功的可能情况共有 $n-m+1$ 种，则：

$$\sum_{i=1}^{n-m+1} p_i (i-1+m) = \frac{(n+m)}{2} = O(n+m)$$

模式匹配算法

设串 S 长度为 n ，串 T 长度为 m ，在匹配成功的情况下，考虑两种极端情况：

最坏情况：不成功的匹配都发生在串 T 的最后一个字符。

例如： $S = \text{"aaaaaaaaa} \textcolor{blue}{aaab} \text{cccccc"}$

$T = \text{"aaab"}$

模式匹配算法

设串 S 长度为 n ，串 T 长度为 m ，在匹配成功的情况下，考虑两种极端情况：

最坏情况：不成功的匹配都发生在串 T 的最后一个字符。

设匹配成功发生在 s_i 处，则在 $i-1$ 趟不成功的匹中共比较了 $(i-1) \times m$ 次，第 i 趟成功的匹配共比较了 m 次，所以总共比较了 $i \times m$ 次，因此（一般地， $m \ll n$ ）

$$\sum_{i=1}^{n-m+1} p_i (i \times m) = \frac{m(n-m+2)}{2} = O(n \times m)$$

4.3 多维数组

数组(array)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。

数组示例

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

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

数组——线性表的推广

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



$$\mathbf{A} = (\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_n)$$

其中:

$$\mathbf{A}_i = (\mathbf{a}_{1i}, \mathbf{a}_{2i}, \dots, \mathbf{a}_{mi}) \\ (1 \leq i \leq n)$$

二维数组是数据元素为线性表的线性表。

数组的基本操作

① 在数组中插入（或）删除一个元素有意义吗？

将元素 x 插入
到数组中第1行第2列。

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

删除数组中
第1行第2列元素。

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

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**

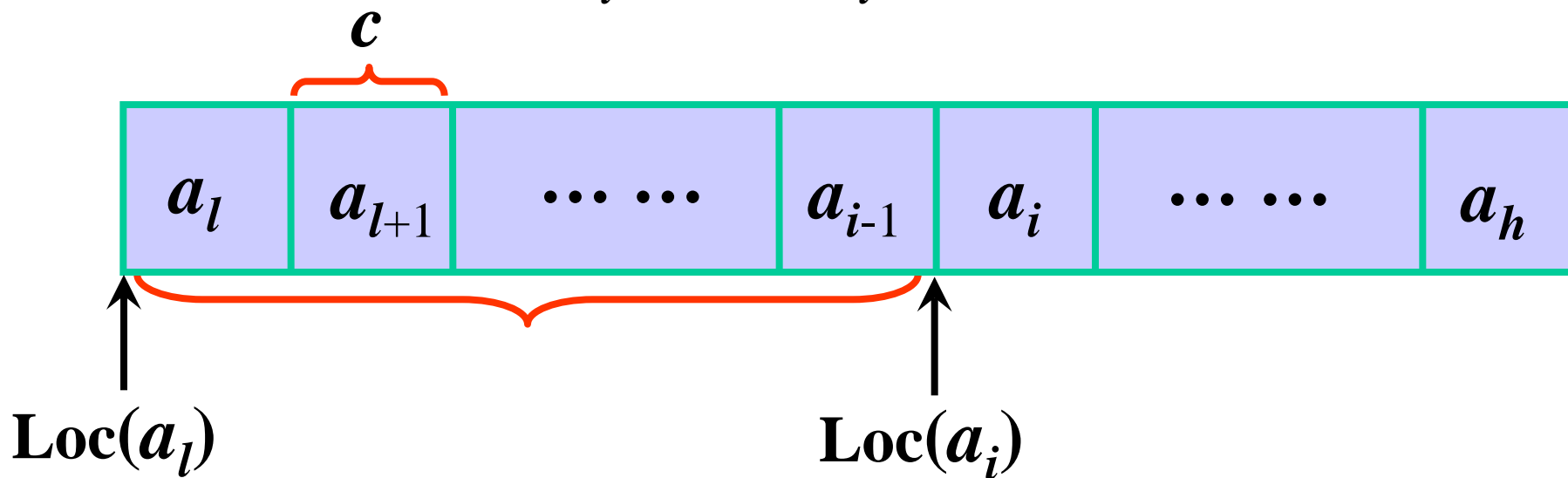
① **数组应该采用何种方式存储？**

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

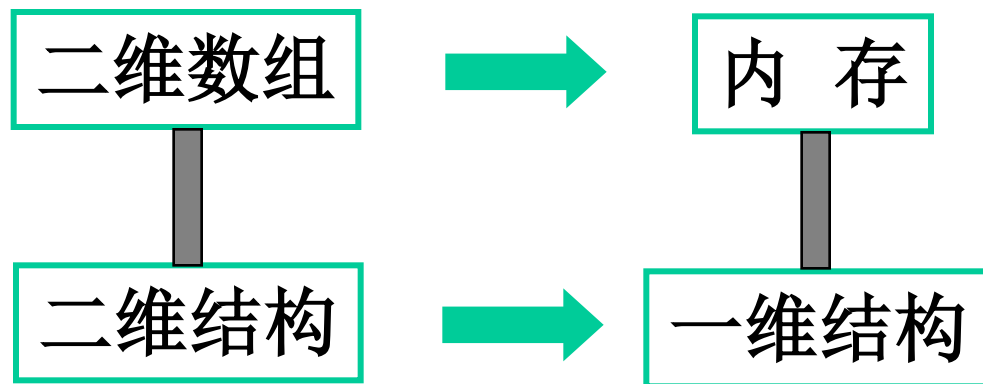
设一维数组的下标的范围为闭区间 $[l, h]$ ，每个数组元素占用 c 个存储单元，则其任一元素 a_i 的存储地址可由下式确定：

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



⑦ 在C语言中，如何计算 $\text{Loc}(a_i)$ ？

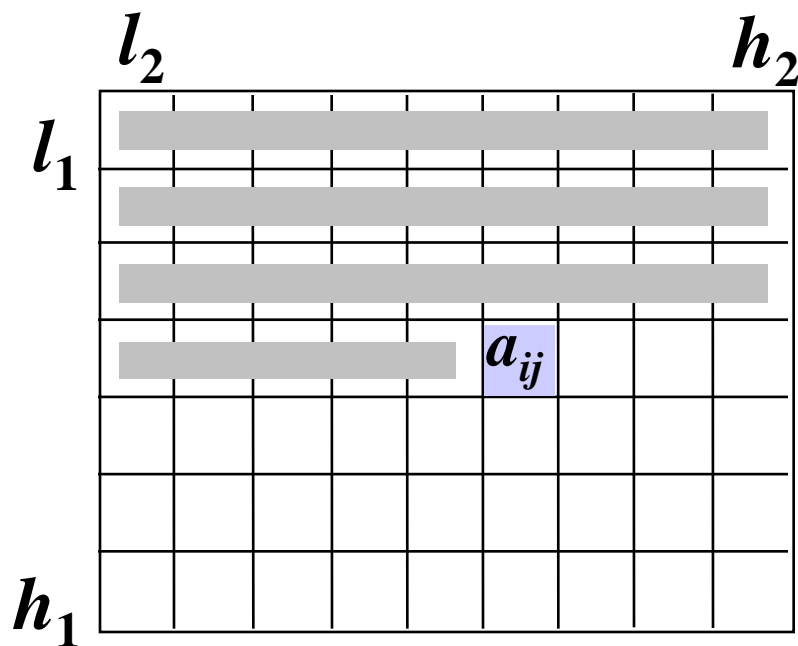
数组的存储结构与寻址——二维数组



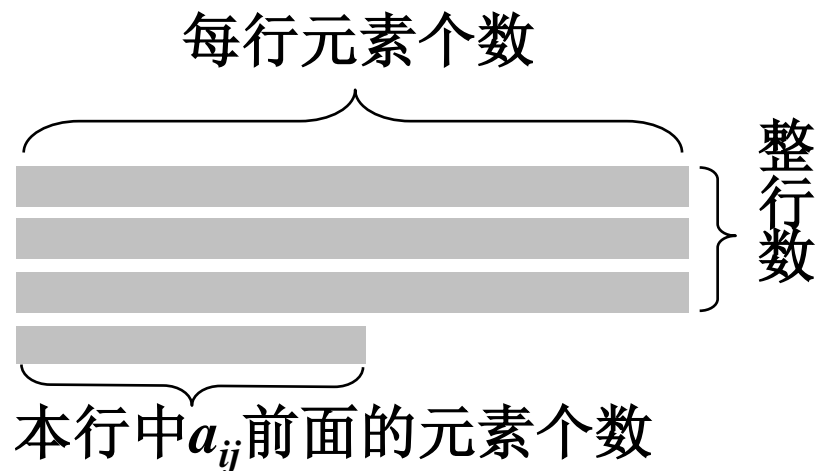
常用的映射方法有两种：

- 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。

按行优先存储的寻址



(a) 二维数组



a_{ij} 前面的元素个数

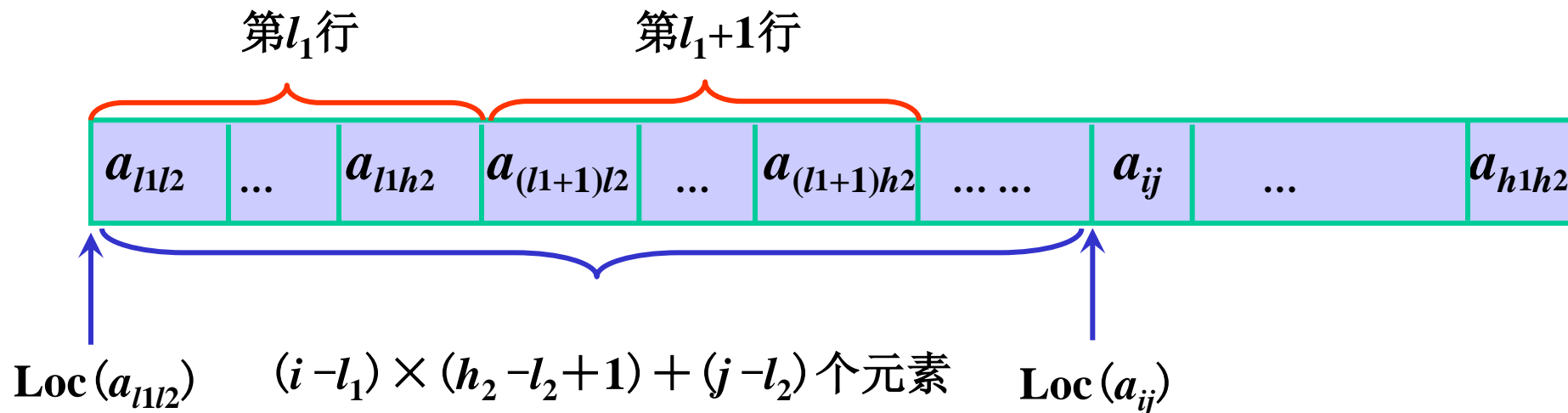
=阴影部分的面积

=整行数 \times 每行元素个数 + 本行中

a_{ij} 前面的元素个数

$$= (i - l_1) \times (h_2 - l_2 + 1) + (j - l_2)$$

按行优先存储的寻址



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

练习：习题4，1（4）

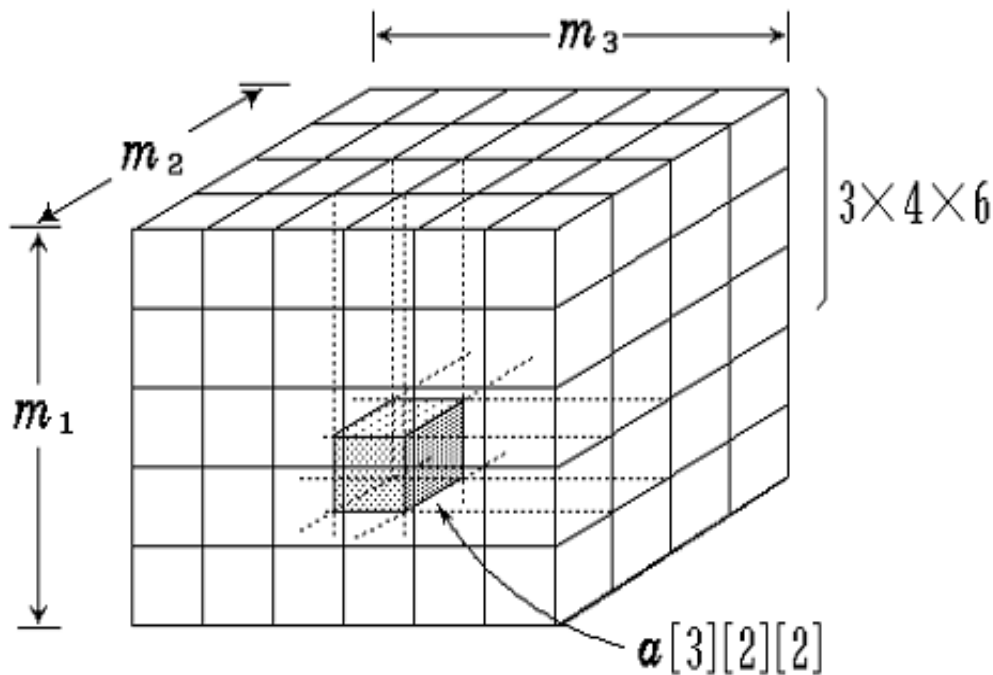
按列优先存储的寻址方法与此类似。

① 在C语言中，如何计算 $\text{Loc}(a_{ij})$ ？

数组的存储结构与寻址

——多维数组(multi-array)

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。

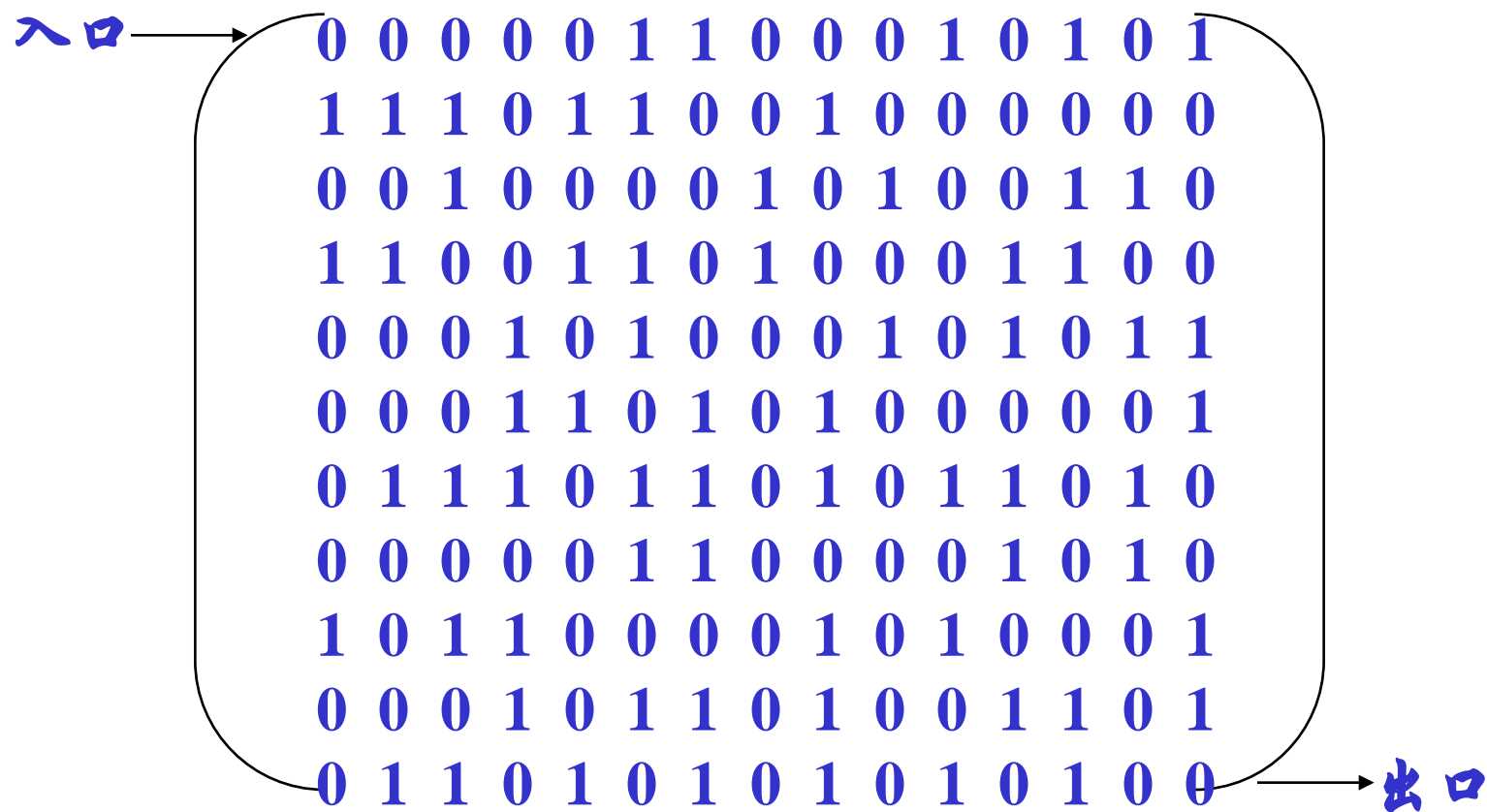


$$\text{Loc}(a_{ijk}) = \text{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

数组的应用——问题 迷宫

在计算机模拟实现中，可以用一个较大的数组表示迷宫，其中元素0表示走得通，元素1表示走不通（受阻），行走路径只考虑水平和垂直两个方向（上、下、左、右）。

一般来说，我们用一个 m 行 n 列的矩阵`maze`表示迷宫，并且假设老鼠从`maze[0][0]`（左上角）进入迷宫，而迷宫的唯一出口在`maze[m-1][n-1]`处（右下角）。任一时刻老鼠在迷宫中的位置用行、列号 $[i][j]$ 来表示，这时它有四个方向可以进行试探，即从图上看是上下左右，设下一位置是 $[g][h]$ ，显然 $[g][h]$ 的值与走的方向有关。



一个迷宫示意图

走迷宫的步骤

- (1) 令老鼠处在迷宫入口，此为当前位置；
- (2) 在当前位置上从右方开始，然后依下、左、上的顺序探测前进方向；
- (3) 向可以进入的方向前进，即目标位置的maze和mark值全为0。前进一步后，目标位置为当前位置，将mark矩阵的当前位置赋值为1，并且将前一位置坐标及进入当前位置的方向入栈；
- (4) 重复步骤(2)和(3)；
- (5) 若找不到前进通路，从原路后退一步（退栈），改变探测方向，再重复步骤(2)、(3)，以寻找另一条新的通路。
- (6) 重复步骤(2)-(5)，直到走出迷宫或宣布迷宫无出路为止。

走迷宫的C++算法

```
void mazepath(maze)          // maze为扩大的迷宫矩阵
{
    // 初始化，老鼠进入迷宫
    mark[1][1] = 1;
    top = 0;
    i = 1; j = 1; d = 0;
    do
    {
        g = i+move[0][d];
        h = j+move[1][d]; // 进行试探
        if ( (maze[g][h] == 0)&&(mark[g][h] == 0) )
        {
            mark[g][h] = 1;    // 进入新位置
            top = top+1;
            stack[top].i = i;
            stack[top].j = j;
            stack[top].d = d;
            i = g; j = h; d = 0;
        }
    };
}
```

```

else
{
    if(d<3)
        d = d+1; // 换新方向再试探
    else
    {
        if(top>0)
        {
            // 后退一步再试
            i = stack[top].i ;
            j = stack[top].j;
            d = stack[top].d;
        }
        else
        {
            cout<<"Have no path"; // 迷宫无通路
            return;
        }
    }
}
}while((g!=m)||(h!=n))
cout <<"Success!"; // 走出迷宫
}

```


4.4 矩阵的压缩存储

特殊矩阵(special matrix): 矩阵中有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(sparse matrix): 矩阵中有很多零元素。

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。

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：

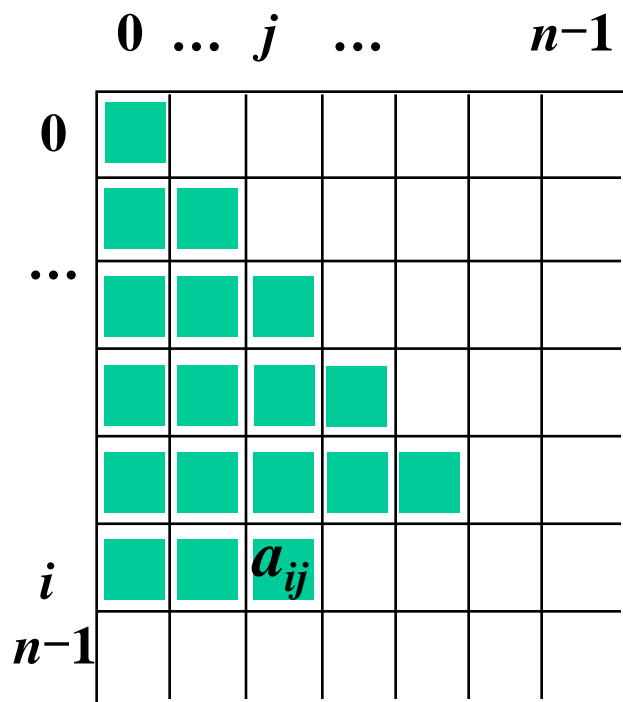
$$a_{ij} = a_{ji}$$



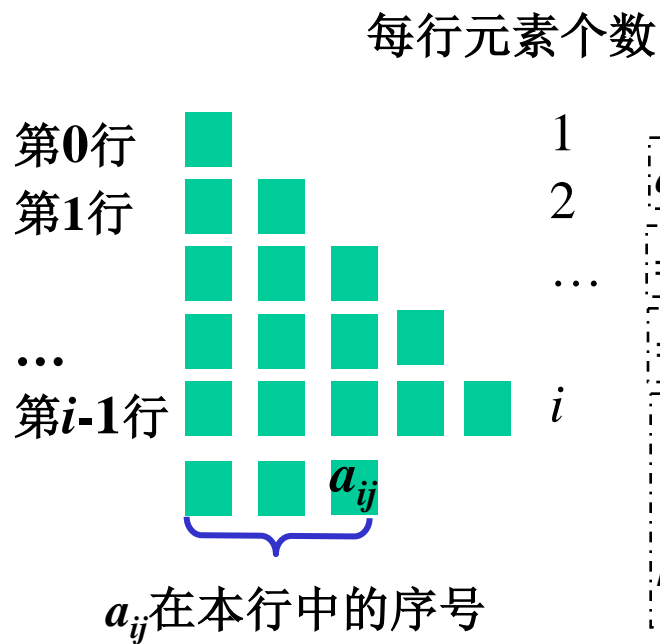
如何压缩存储？

只存储下三角部分的元素。

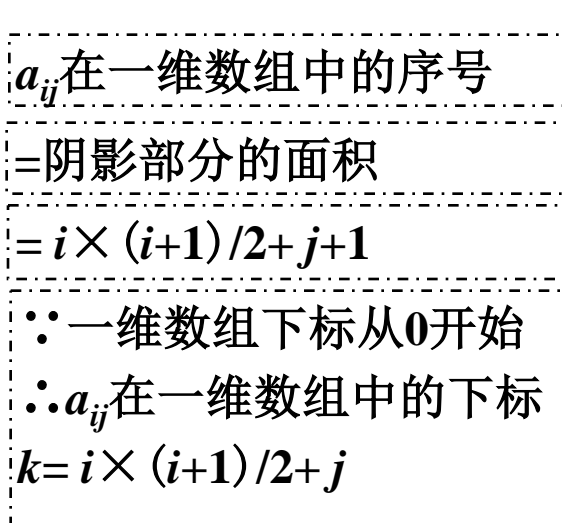
对称矩阵的压缩存储:



(a) 下三角矩阵

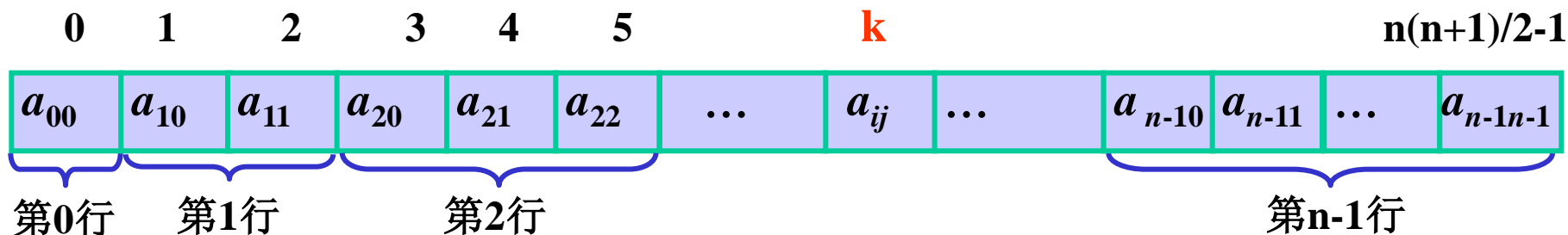


(b) 存储说明



(c) 计算方法

对称矩阵的压缩存储



对于下三角中的元素 a_{ij} ($i \geq j$)，在一维数组中的下标 k 与 i 、 j 的关系为： $k = i \times (i+1)/2 + j$ 。

对于上三角中的元素 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： $k = j \times (j+1)/2 + i$ 。

2、特殊矩阵的压缩存储——三角矩阵

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

(a) 下三角矩阵

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

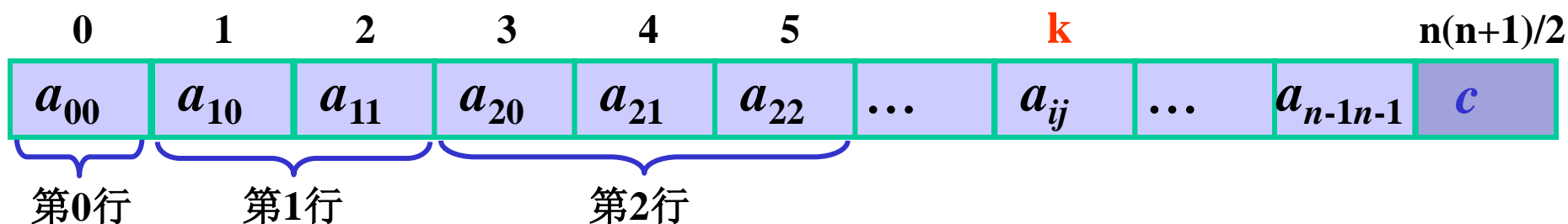
(b) 上三角矩阵

① 如何压缩存储?

只存储上三角（或下三角）部分的元素。

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:

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

上三角矩阵的压缩存储

存储 { 上三角元素
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:

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

3、稀疏矩阵的压缩存储

$$A = \begin{pmatrix} 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

① 如何只存储非零元素？

注意：稀疏矩阵中的非零元素的分布没有规律。

稀疏矩阵的压缩存储

将稀疏矩阵中的每个非零元素表示为：

(行号，列号，非零元素值)——三元组

稀疏矩阵的三元组表示：

```
typedef int dataType;      // 数组元素为整型  
struct element  
{  
    int row, col;          //行号，列号  
    dataType item;       //非零元素值  
};
```

稀疏矩阵的压缩存储

三元组表：将稀疏矩阵的非零元素对应的三元组所构成的集合，按行优先的顺序排列成一个线性表。

$$A = \begin{pmatrix} 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

三元组表 = ((0,0,15), (1,1,11), (2,3,6), (4,0,9))

① 如何存储三元组表？

稀疏矩阵的压缩存储——三元组顺序表

采用顺序存储结构存储三元组表

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



三元组顺序表是否需要预留存储空间？

稀疏矩阵的修改操作



三元组顺序表的插入/删除操作

稀疏矩阵的压缩存储——三元组顺序表

采用顺序存储结构存储三元组表

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

① 是否对应唯一的稀疏矩阵？

| | row | col | item |
|-----------|-----------|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| | 7 (非零元个数) | | |
| | 5 (矩阵的行数) | | |
| | 6 (矩阵的列数) | | |

稀疏矩阵的压缩存储——三元组顺序表

存储结构定义：

```
const int MaxTerm=100;  
typedef int dataType;  
struct SparseMatrix  
{  
    dataType data[MaxTerm]; //存储非零元素  
    int mu, nu, tu;        //行数，列数，非零元个数  
};
```

三元组顺序表操作——转置操作

例：

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

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

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | 1 | 1 | 11 |
| 3 | 2 | 1 | 3 |
| 4 | 3 | 0 | 22 |
| 5 | 3 | 2 | 6 |
| 6 | 5 | 0 | -15 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

三元组顺序表转置算法

基本思想：直接取，顺序存。

即在A的三元组顺序表中依次找第0列、第1列、...直到最后一列的三元组，并将找到的每个三元组的行、列交换后顺序存储到B的三元组顺序表中。

设置矩阵B的行数、列数、非零元个数

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

在矩阵A中查找第0列非零元，顺序存储到矩阵B中

| | row | col | item |
|-----------|-----|-----|------|
| → 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| → 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

在矩阵A中查找第1列非零元，顺序存储到矩阵B中

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| → 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | 1 | 1 | 11 |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

在矩阵A中查找第2列非零元， 顺序存储到矩阵B中

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| → 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | 1 | 1 | 11 |
| 3 | 2 | 1 | 3 |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

在矩阵A中查找第3列非零元， 顺序存储到矩阵B中

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| → 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| → 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | 1 | 1 | 11 |
| 3 | 2 | 1 | 3 |
| 4 | 3 | 0 | 22 |
| 5 | 3 | 2 | 6 |
| 6 | | | |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

在矩阵A中查找第4列非零元， 顺序存储到矩阵B中

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | 1 | 1 | 11 |
| 3 | 2 | 1 | 3 |
| 4 | 3 | 0 | 22 |
| 5 | 3 | 2 | 6 |
| 6 | | | |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

在矩阵A中查找第5列非零元，顺序存储到矩阵B中

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| → 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | 1 | 1 | 11 |
| 3 | 2 | 1 | 3 |
| 4 | 3 | 0 | 22 |
| 5 | 3 | 2 | 6 |
| 6 | 5 | 0 | -15 |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

在矩阵A中查找第6列非零元，顺序存储到矩阵B中

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 3 | 22 |
| 2 | 0 | 5 | -15 |
| 3 | 1 | 1 | 11 |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 6 |
| 6 | 4 | 0 | 91 |
| | 空 | 空 | 空 |
| MaxTerm-1 | 闲 | 闲 | 闲 |
| 5（矩阵的行数） | | | |
| 6（矩阵的列数） | | | |
| 7（非零元个数） | | | |

| | row | col | item |
|-----------|-----|-----|------|
| 0 | 0 | 0 | 15 |
| 1 | 0 | 4 | 91 |
| 2 | 1 | 1 | 11 |
| 3 | 2 | 1 | 3 |
| 4 | 3 | 0 | 22 |
| 5 | 3 | 2 | 6 |
| 6 | 5 | 0 | -15 |
| | | | |
| MaxTerm-1 | | | |
| 6（矩阵的行数） | | | |
| 5（矩阵的列数） | | | |
| 7（非零元个数） | | | |

三元组顺序表转置算法——伪代码

1. 设置转置后矩阵**B**的行数、列数和非零元个数;
2. 在**B**中设置初始存储位置**pb**;
3. **for** (**col**=最小列号; **col**<=最大列号; **col**++)
 - 3.1 在**A**中查找列号为**col**的三元组;
 - 3.2 交换其行号和列号, 存入**B**中**pb**位置;
 - 3.3 **pb**++;

稀疏矩阵的压缩存储——十字链表

采用**链接**存储结构存储三元组表，每个非零元素对应的三元组存储为一个链表结点，结构为：

| row | col | item |
|------|-----|-------|
| down | | right |

row: 存储非零元素的行号

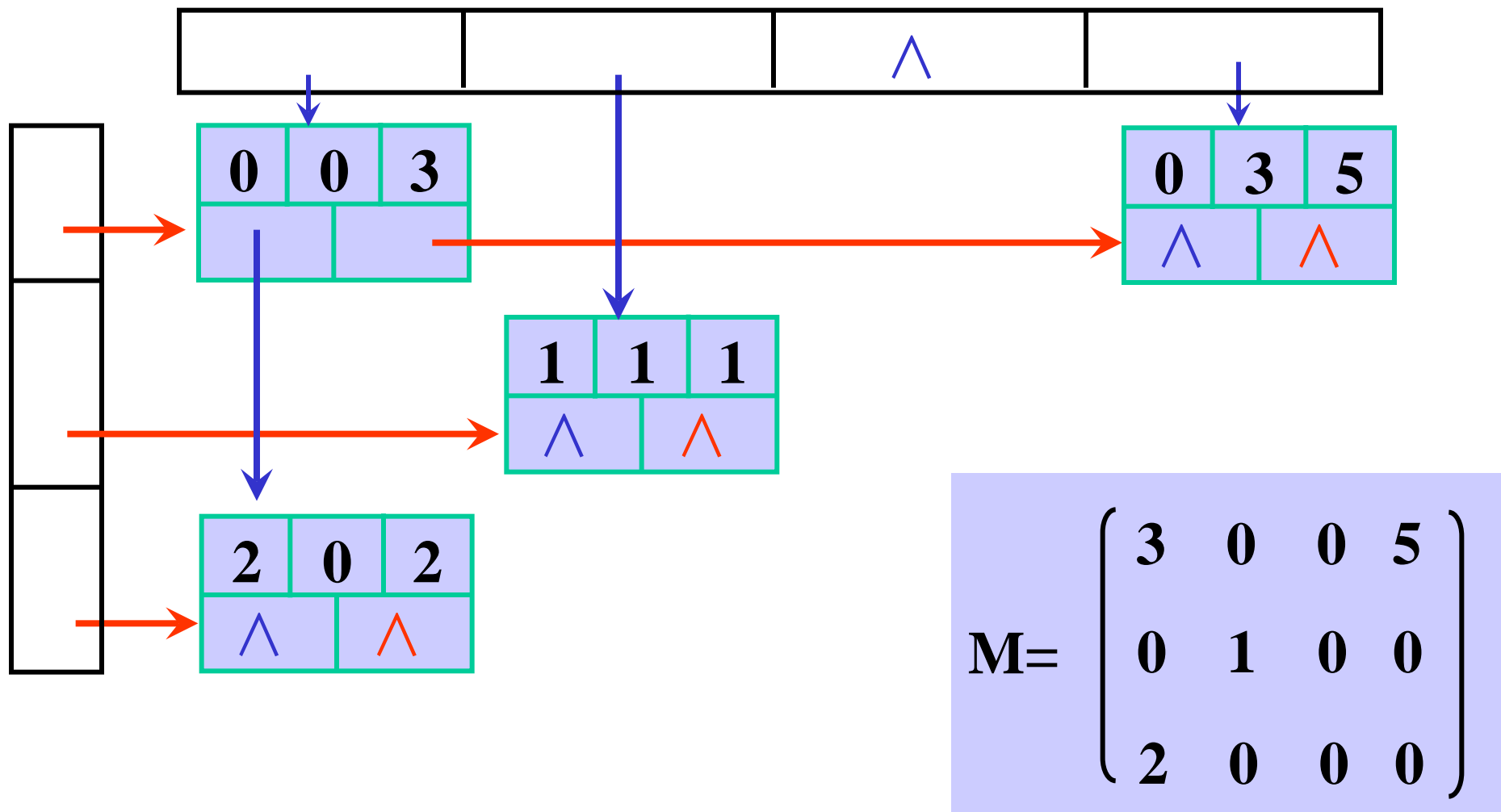
col: 存储非零元素的列号

item: 存储非零元素的值

right: 指针域，指向同一行中的下一个三元组

down: 指针域，指向同一列中的下一个三元组

稀疏矩阵的压缩存储——十字链表



本章作业

习题4 (P96) : 4, 5