第四章 字符串和数组

本章的基本内容是：

4.1 字符串

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：

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

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（$1 \le i \le n$）是一个任意字符。

一、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：Σ = {0,1}
  - 生物信息中DNA字符集：Σ = {A,C,G,T}
  - 英语语言： Σ = {26个字符，标点符号，…}
  - 简体中文标准字符集 GB2312 ： Σ = {6763个汉字，标点符号，… }
  - ……

二、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）

三、与字符串相关的基本概念

- **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。

*S*1="*ab*12*cd* "

*S*2="*ab*12"

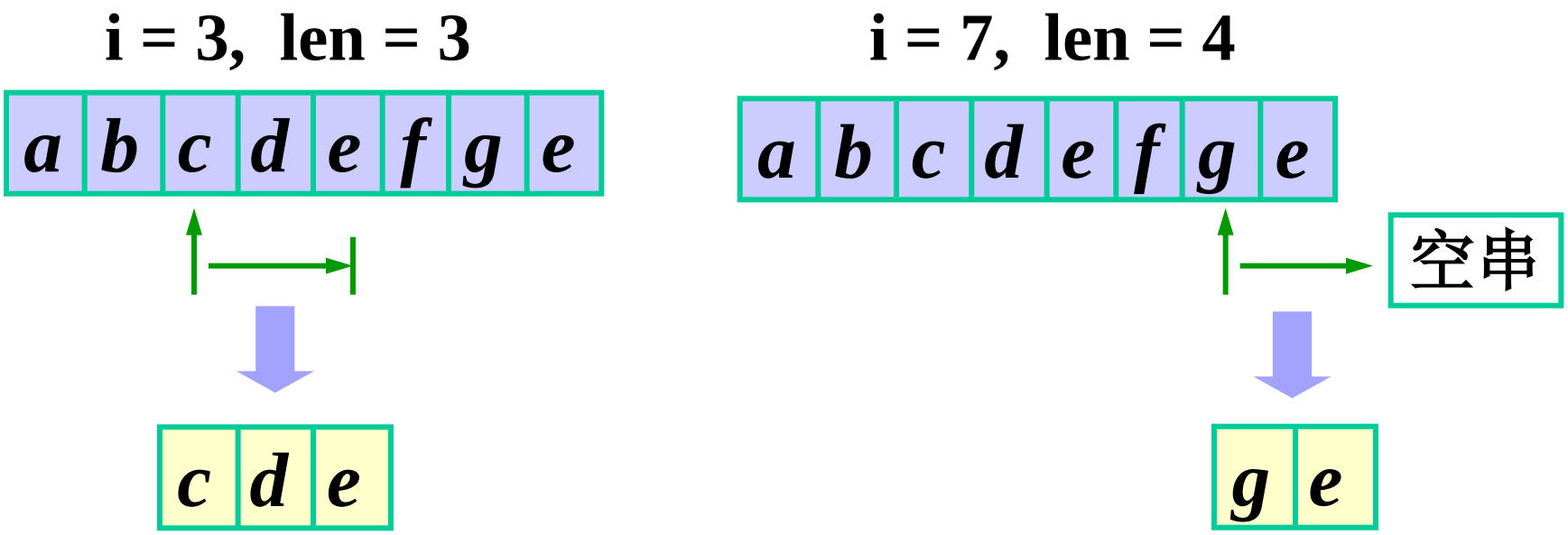
*S*3="*ab*13"

*S*4="*ab*12*φ*"

*S*5=" "

*S*6="*φφφ* "

例如：求子串操作SubStr(s, i, len)



□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：$X="x_1x_2\ldots x_n"$和$Y="y_1y_2\ldots y_m"$，则：

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

2. 当下列条件之一成立时，称$X<Y$：

⑴ $n<m$且$x_i=y_i$（$1\le i\le n$）；

⑵存在$k\leqslant \min(m,n)$，使得$x_i=y_i$（$1\le i\le k-1$）且$x_k<y_k$。

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

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++标准字符串

- **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++标准字符串函数

- 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

四、字符串的存储结构和实现

- 字符串的顺序存储
- C++标准字符串函数的实现
- 字符串类class String的存储结构
- String类的运算实现

1、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

2、C++标准字符串函数的实现

- 串长函数：int strlen(char *s);
- 串复制：char *strcpy(char *d, char*s);
- 串拼接：char *strcat(char *s1, char *s2);
- 串比较：int strcmp(char *s1, char *s2);
- 寻找字符：

  char * strchr(char *d, char ch)

  char * strrchr(char *d, char ch)

- 求字符串的长度

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

● 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;
}
```

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

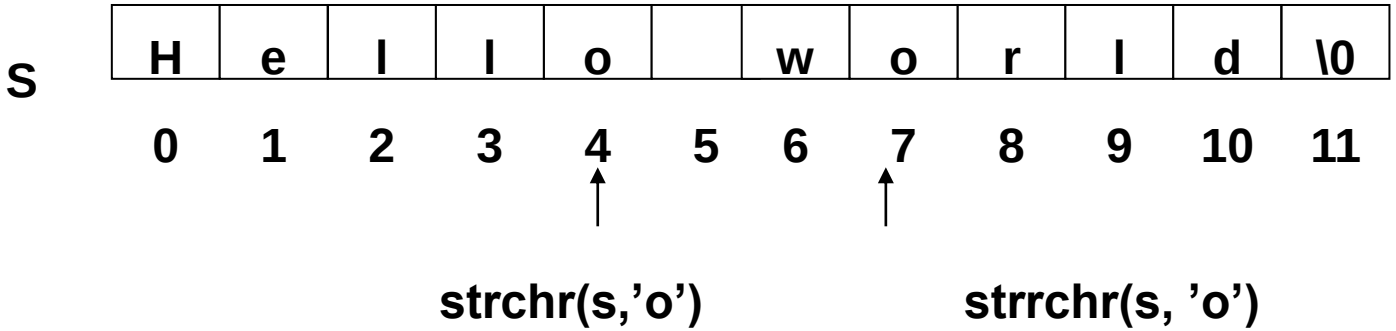
● 寻找字符

```
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
//如果没有找到ch，则为0值。
    while(*s != '\0' && *s != c) {
        ++s;
    }
    return *s==c ? s : NULL;
}
```

● 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++;
 //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] ;
}
```

例如，　字符串s ：



寻找字符o，strchr(s,'o')结果返回4；

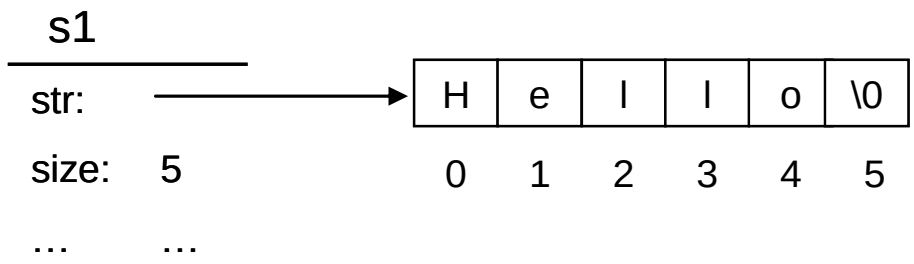
反方向寻找r， strrchr(s,'o')结果返回7

3、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";



4、String串类运算的实现

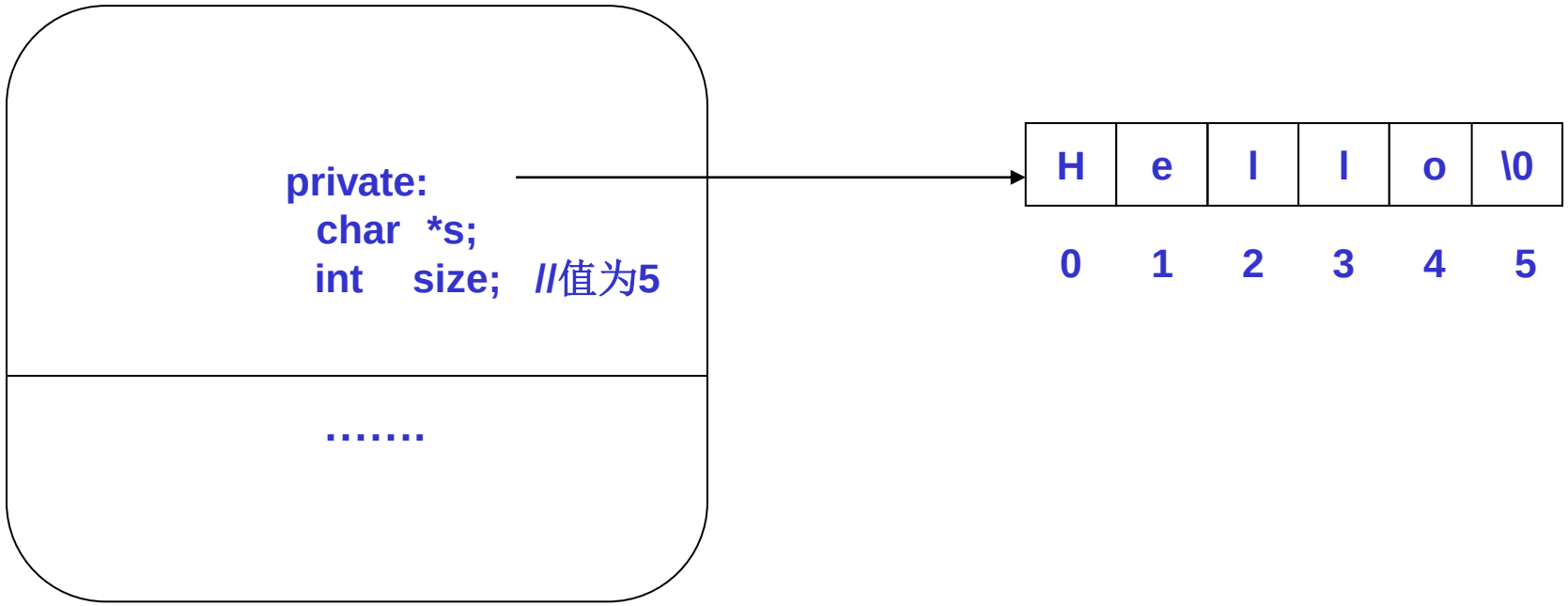
```
String::String(char *s)  {
    // 先要确定新创字符串实际需要的存储空间，s的类型为(char *)，
    // 作为新创字符串的初值。确定s的长度，用标准字符串函数
    //strlen(s)计算长度
    size = strlen(s) ;

    // 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 s1 = "Hello" ;

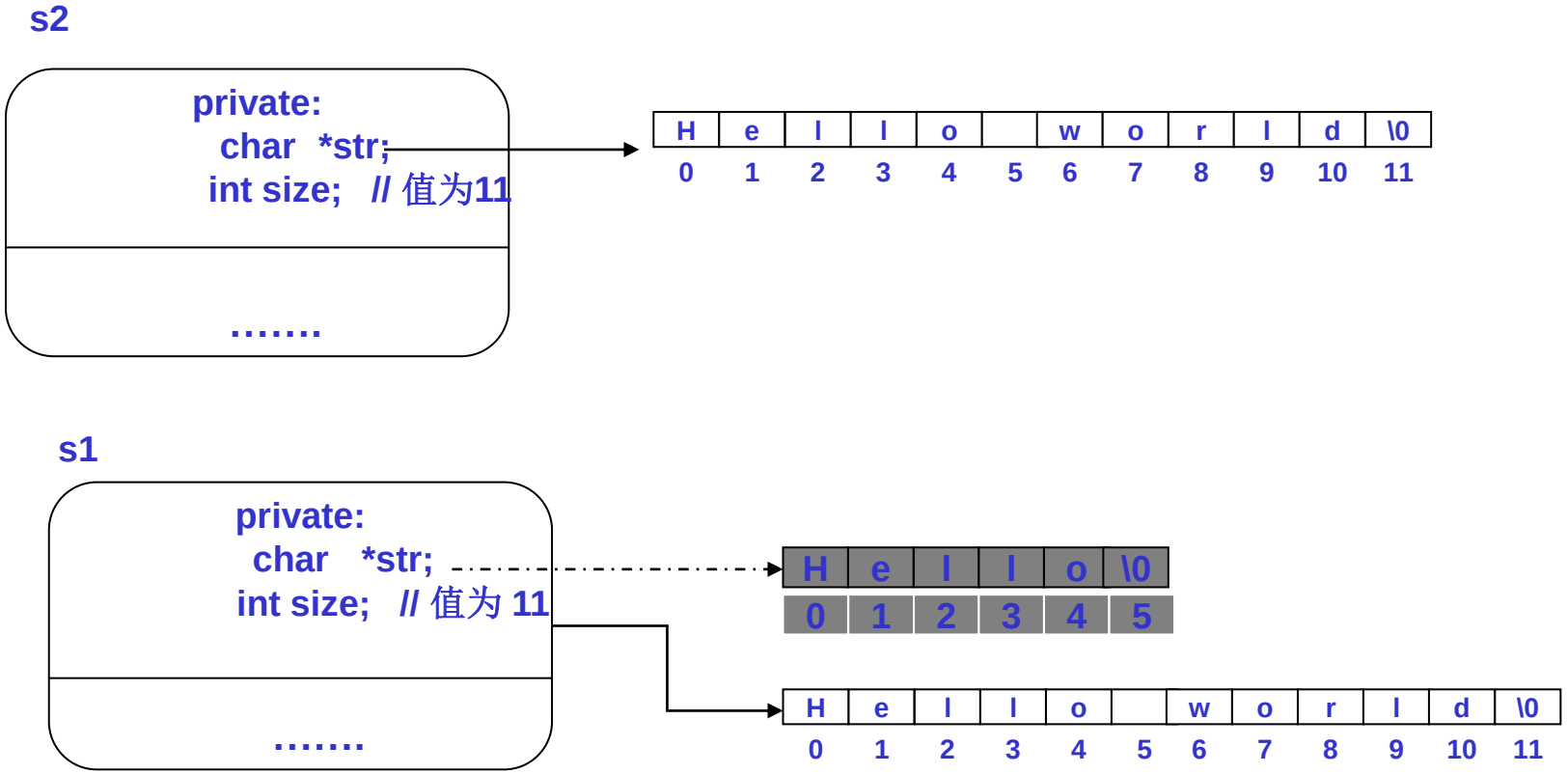
s1



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

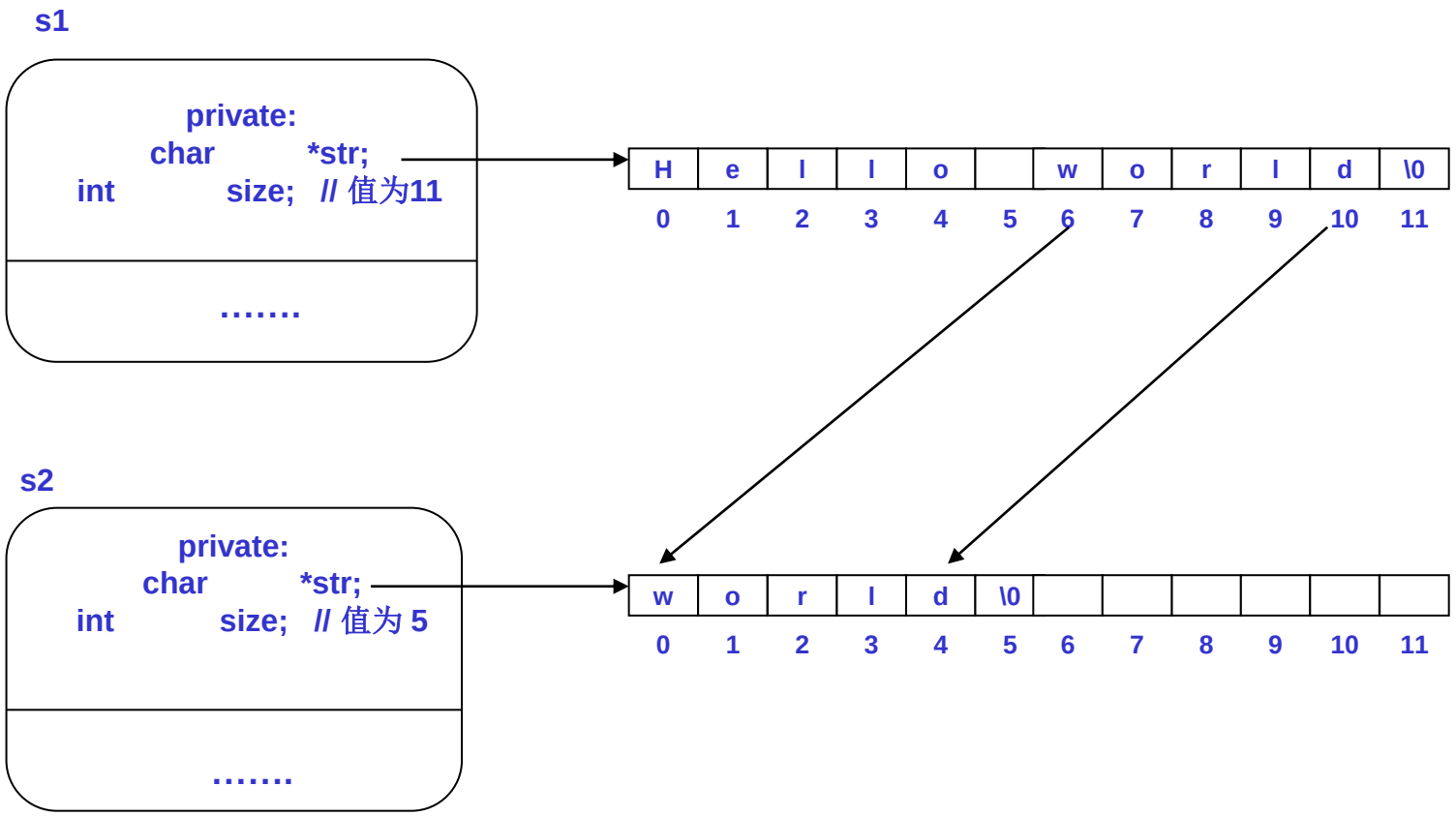
```
// 赋值运算
String  String::operator= (String&  s) {
  // 参数 s 将被赋值到本串。 若本串的串长和s的串长不同，
  //则应该释放本串的str存储空间，并开辟新的空间
  if (size != s.size)  {
        delete [] str ;            // 释放原存储空间
    str = new char [s.size+1];
    // 若开辟动态存储空间失败，则退出正常运行
    assert(str != 0);
    size = s.size;
  }
  strcpy(str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

String s2 = "Hello world";
并通过赋值语句：s1 = s2;



```
// 取子串运算
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;
   //  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;
}
```

s2 = s1.Substr(6, 5) ;



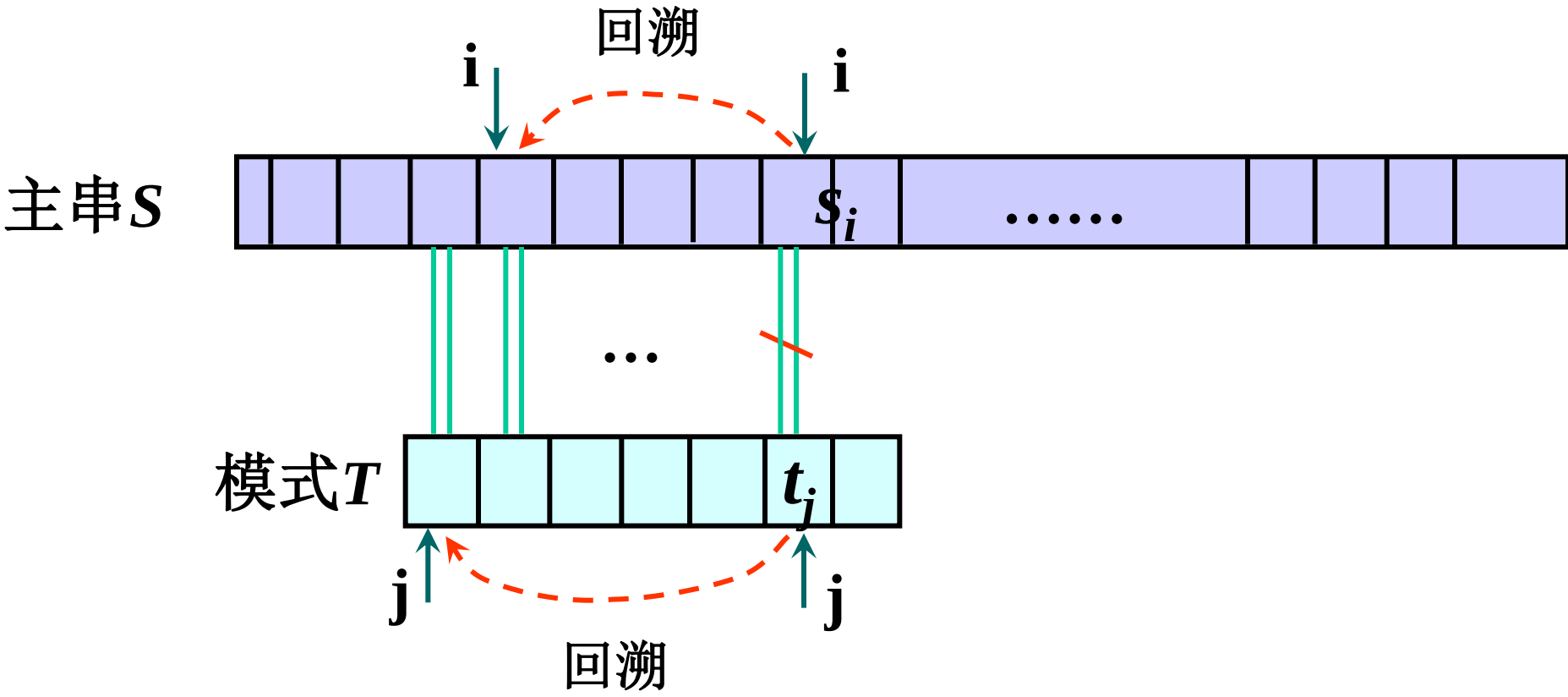
4.2 模式匹配

- 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）

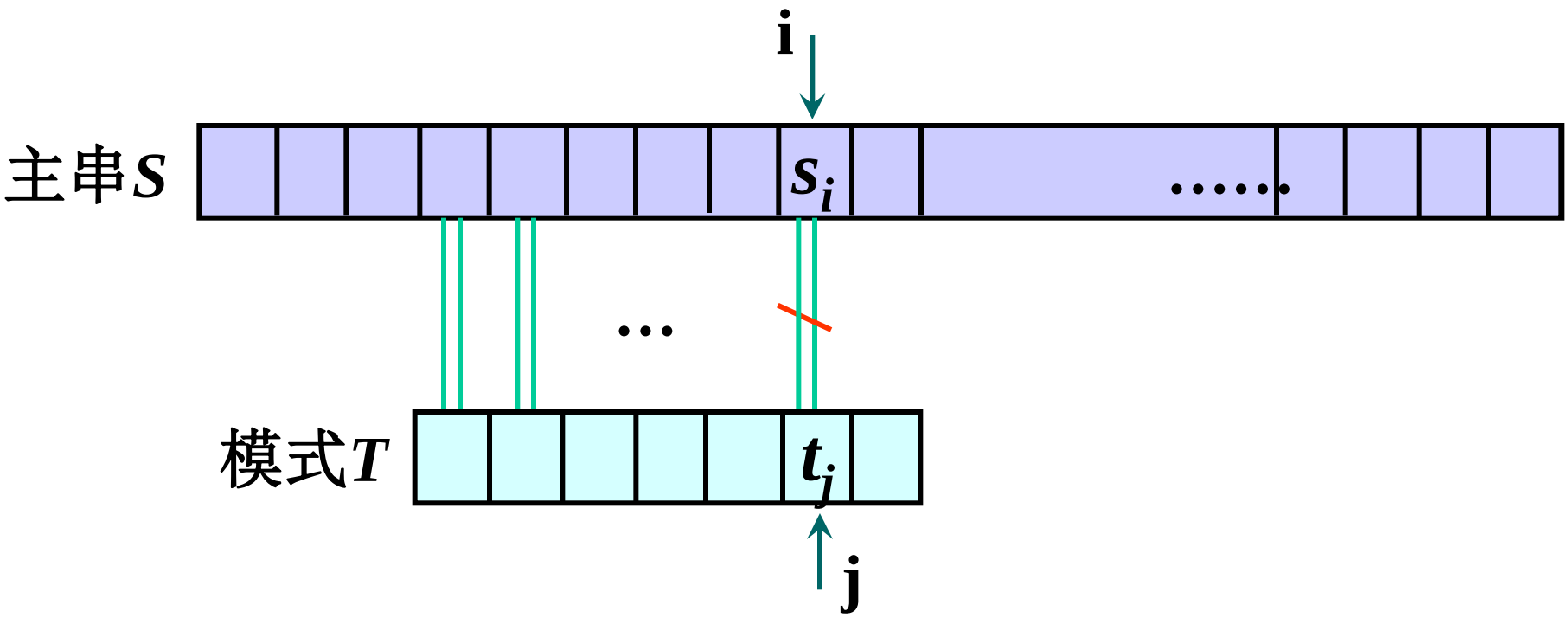
  所谓模式匹配就是在目标T中寻找一个给定的模式P的过程 。
- 模式匹配的应用
  - 文本编辑时的特定词、句的查找
  - DNA信息的提取
  - 确认是否具有某种结构
  - …

- 在大文本（诸如句子、段落，或书本）中定位（或查找）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，这里主要介绍朴素(**Brute Force**)模式匹配算法和**Knuth-Morris-Pratt (KMP)** 模式匹配算法。

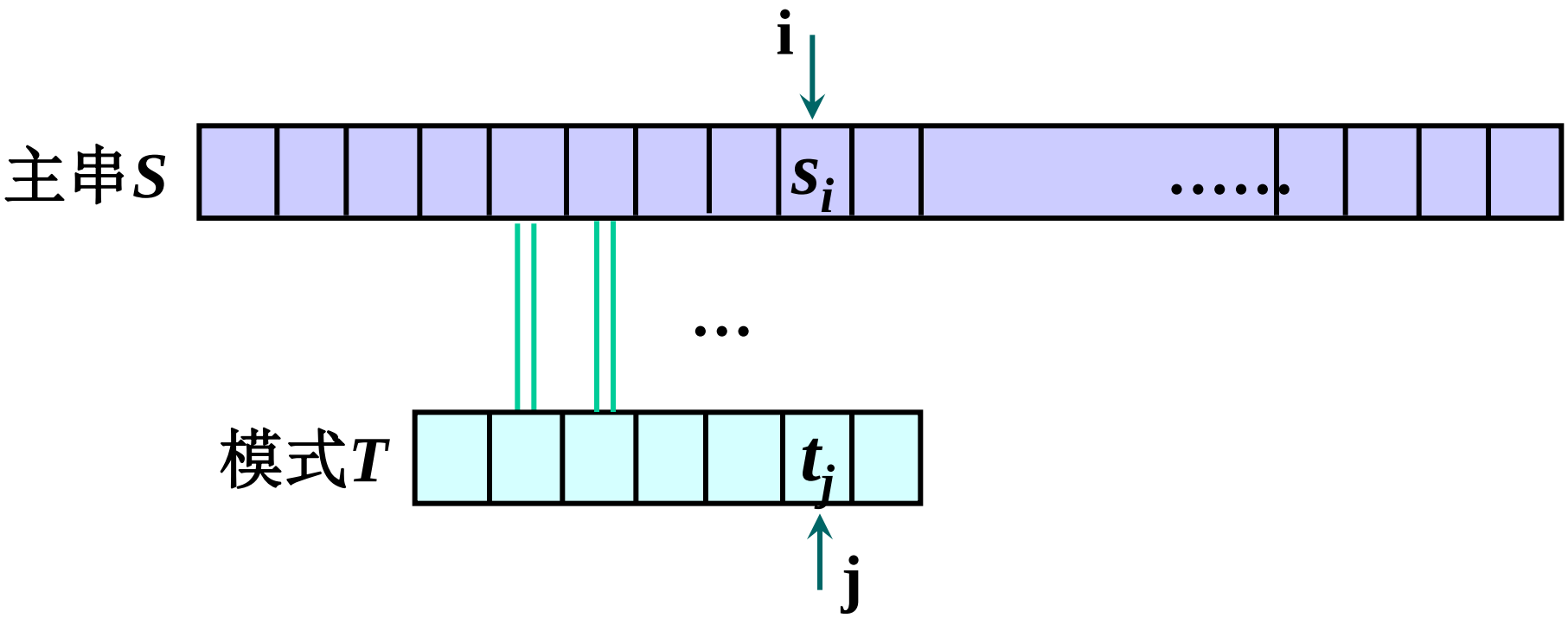
一、BF模式匹配算法



BF模式匹配算法

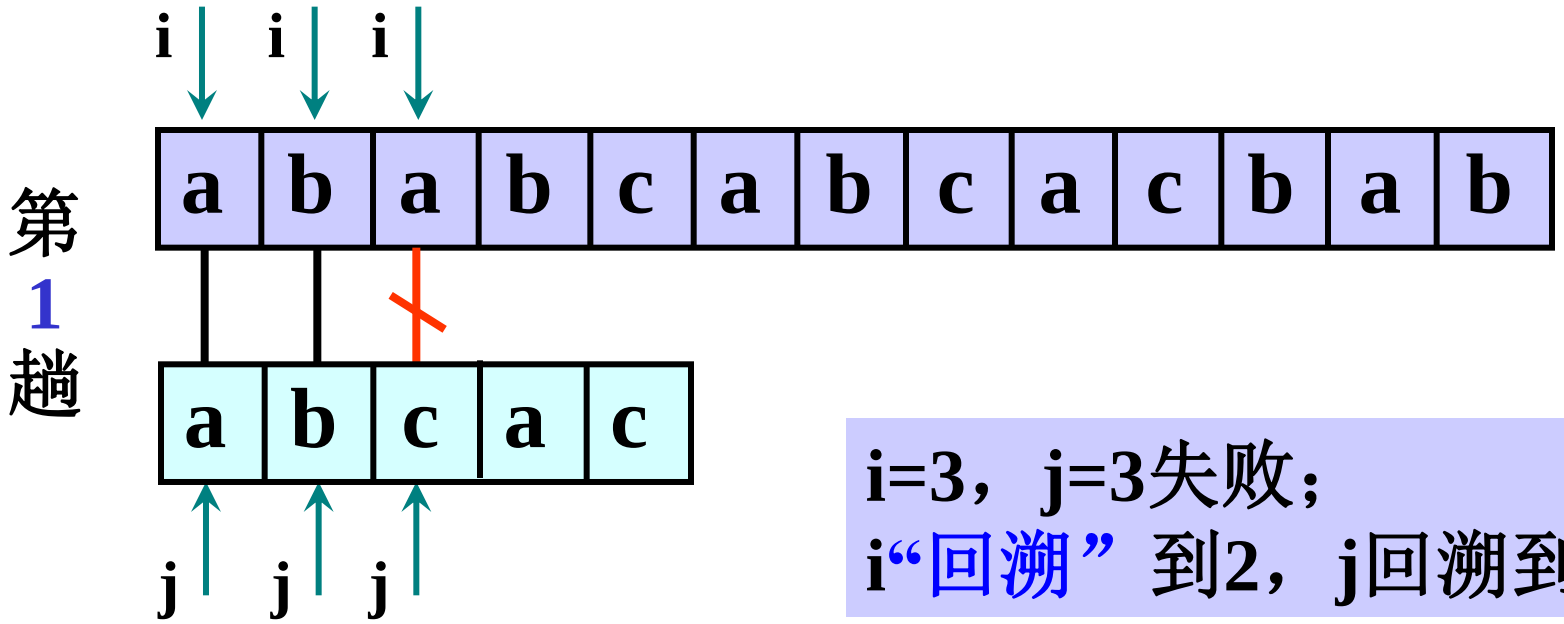


BF模式匹配算法



BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

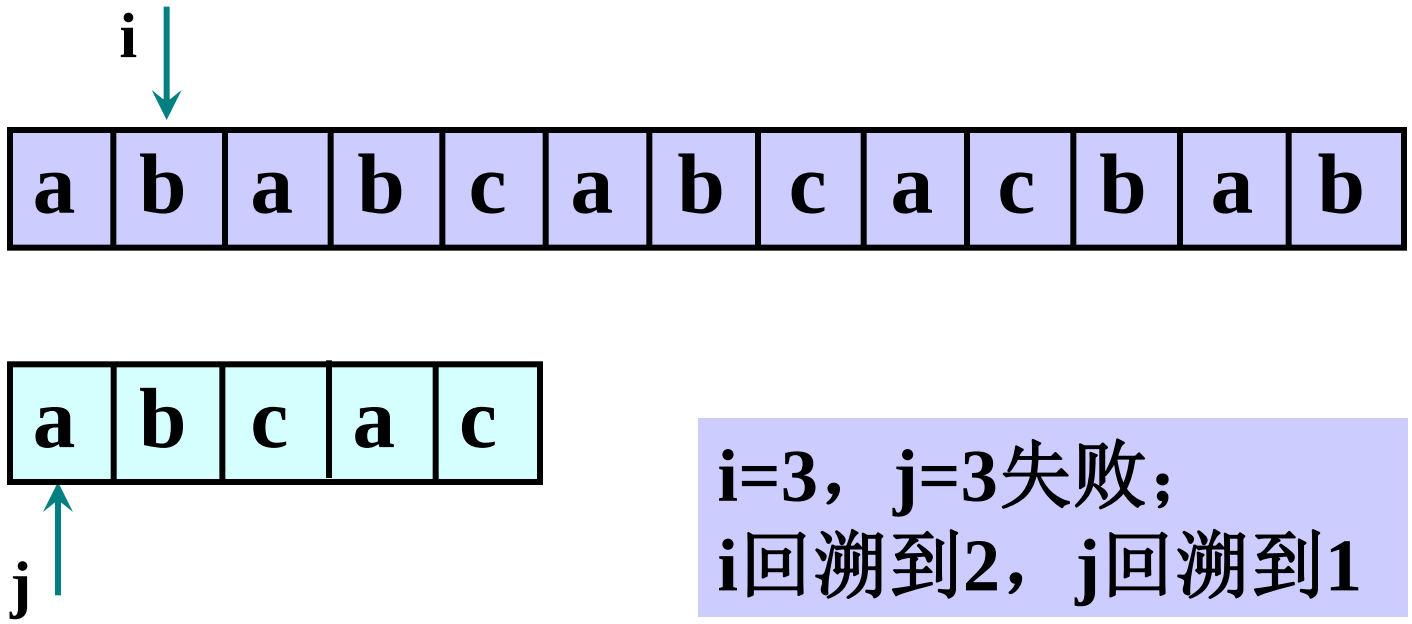


i=3，j=3失败；
i“回溯”到2，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第1趟

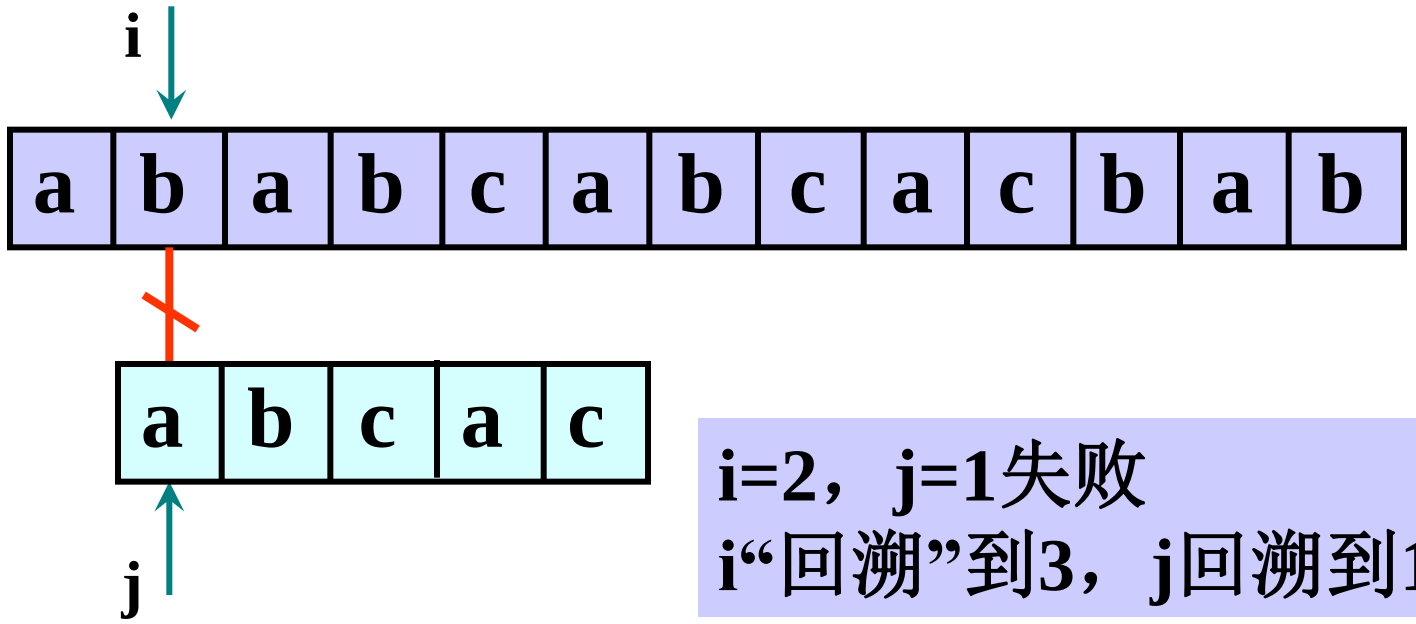


i=3，j=3失败；
i回溯到2，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第2趟

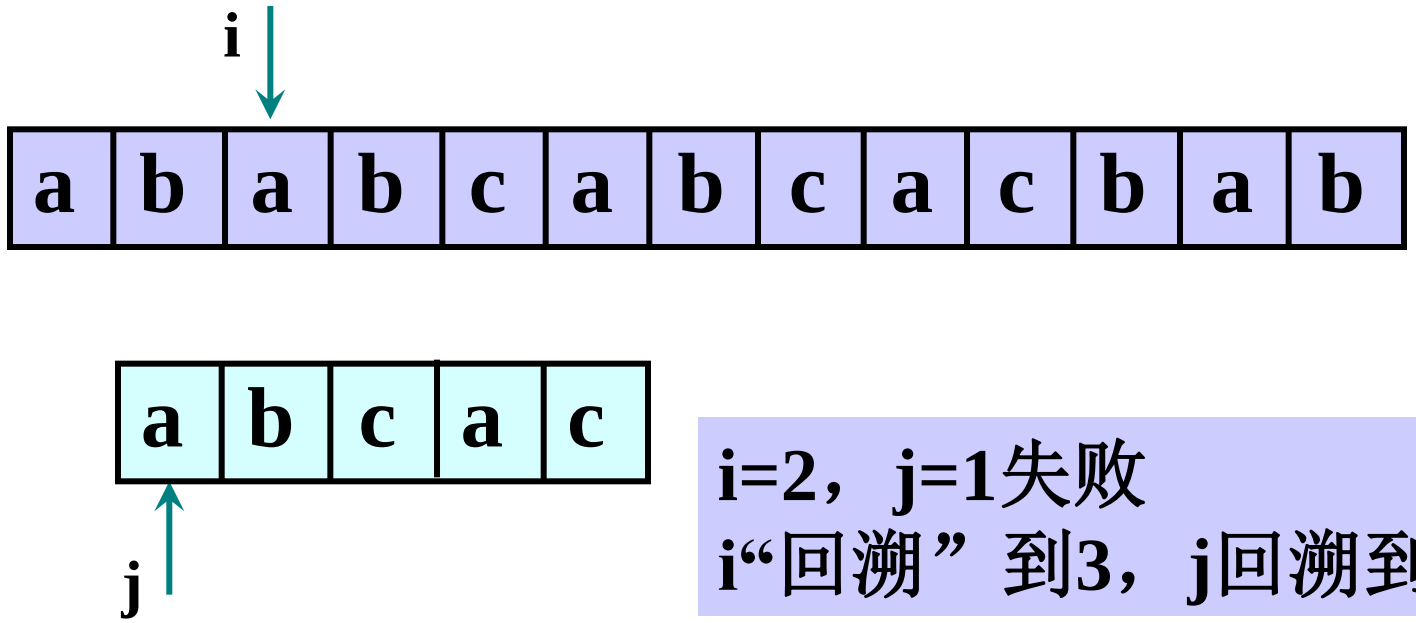


i=2，j=1失败
i“回溯”到3，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第2趟

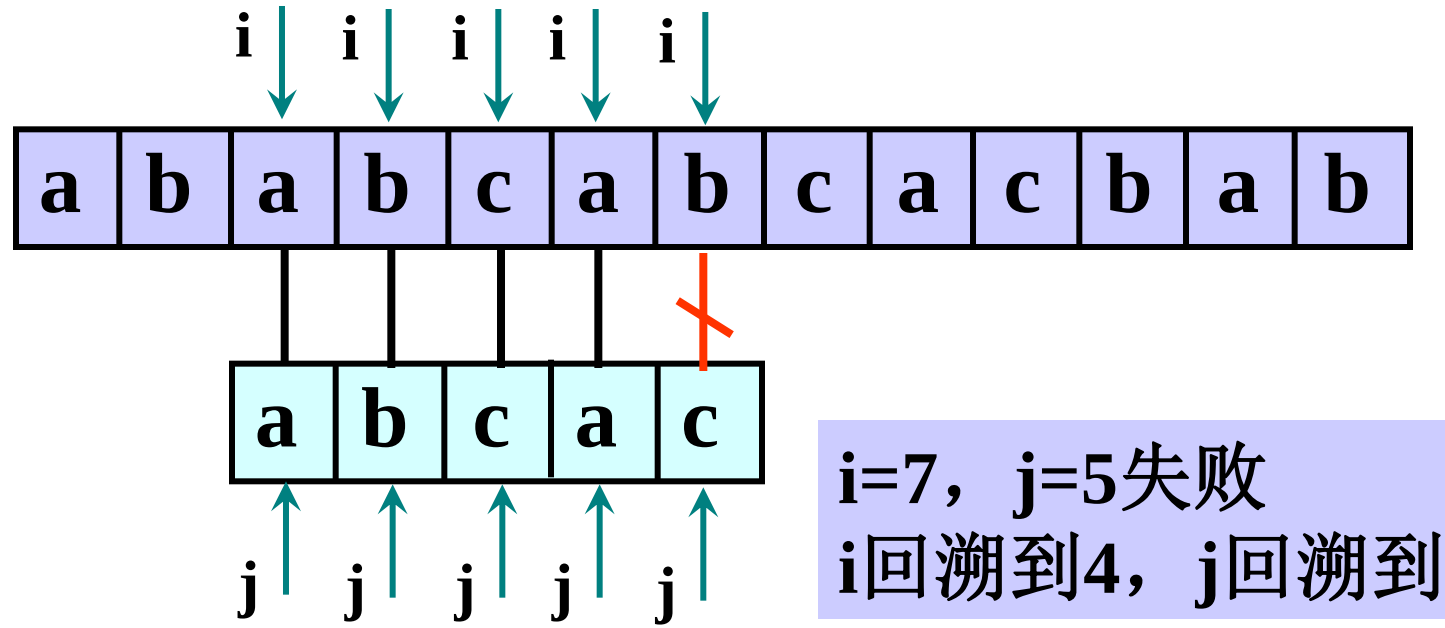


i=2，j=1失败
i“回溯”到3，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第3趟

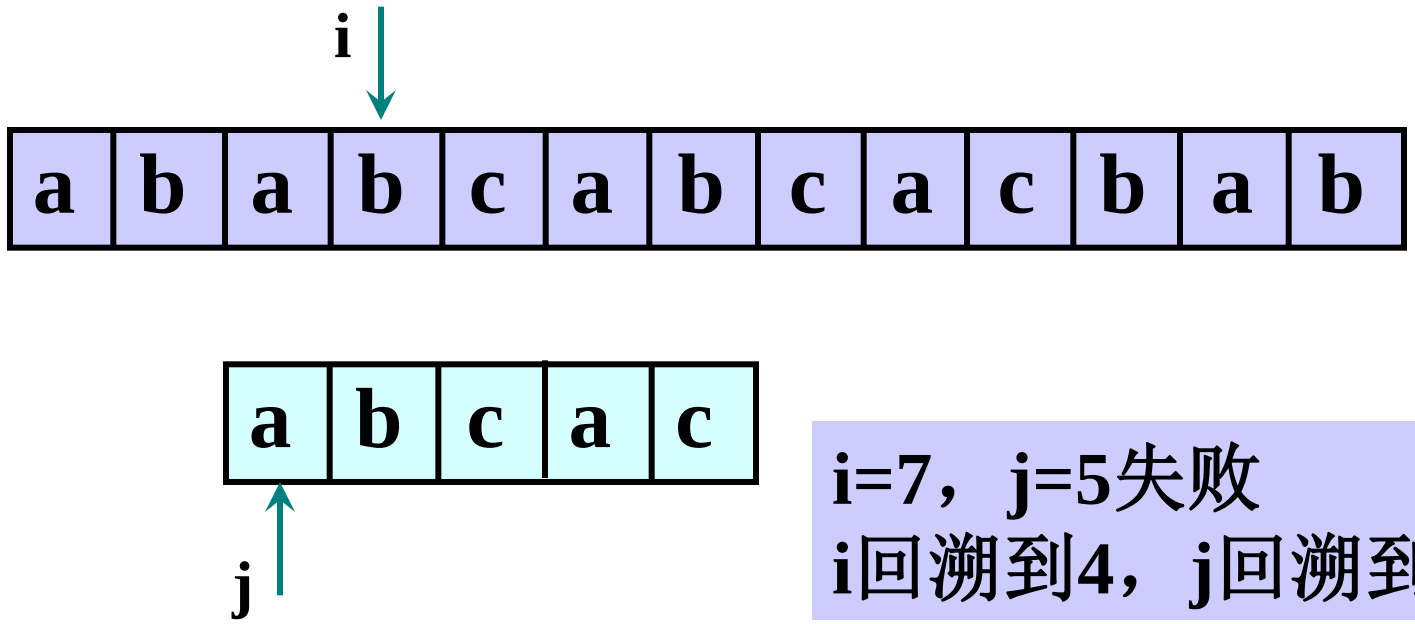


i=7，j=5失败
i回溯到4，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第3趟

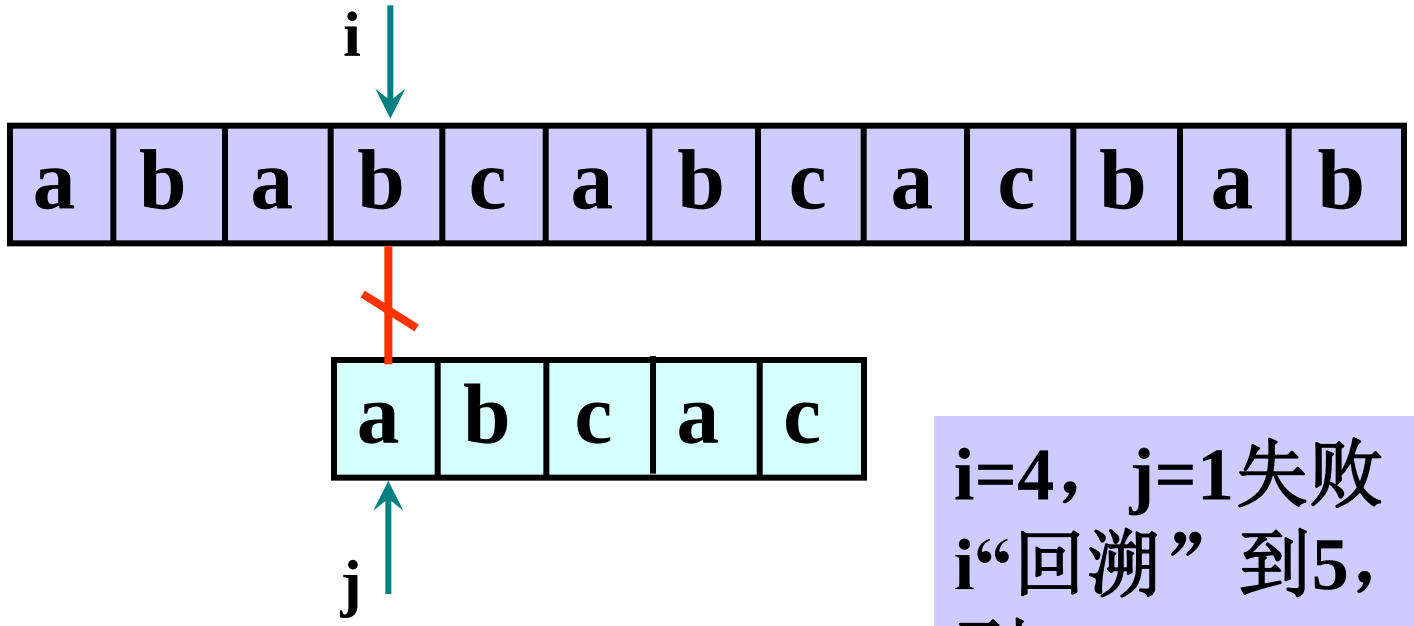


i=7，j=5失败
i回溯到4，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第4趟

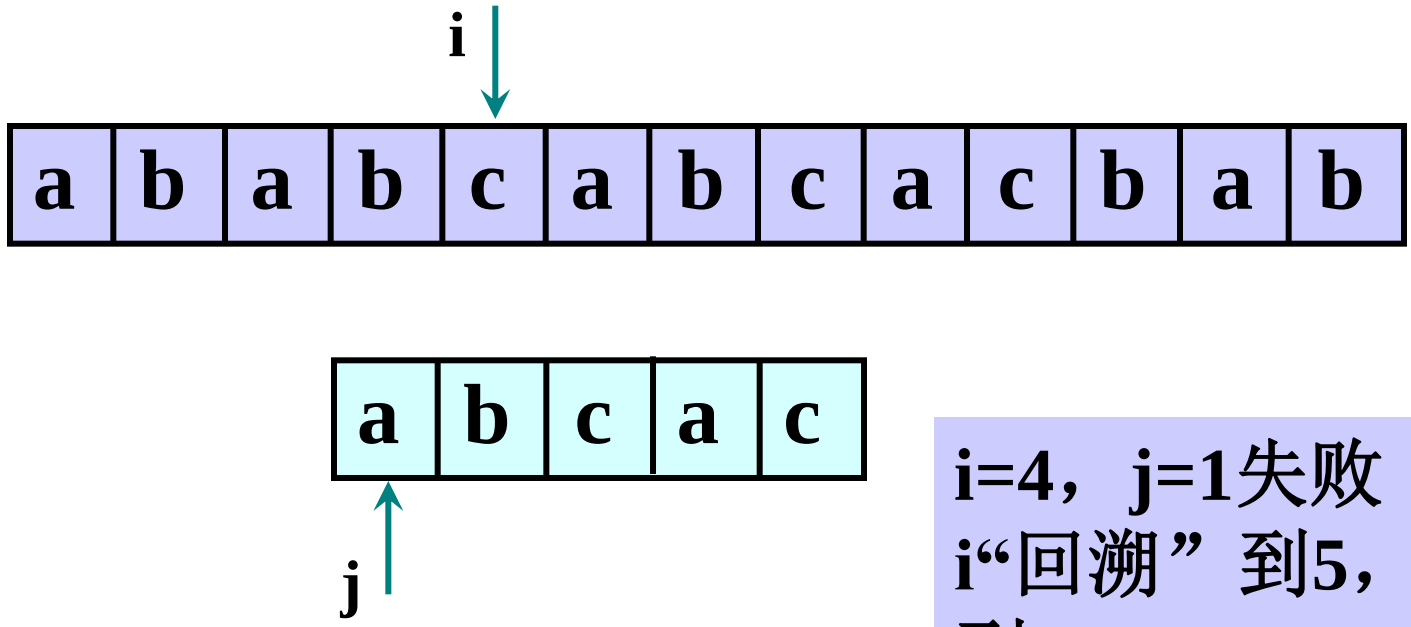


i=4，j=1失败
i“回溯”到5，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第4趟

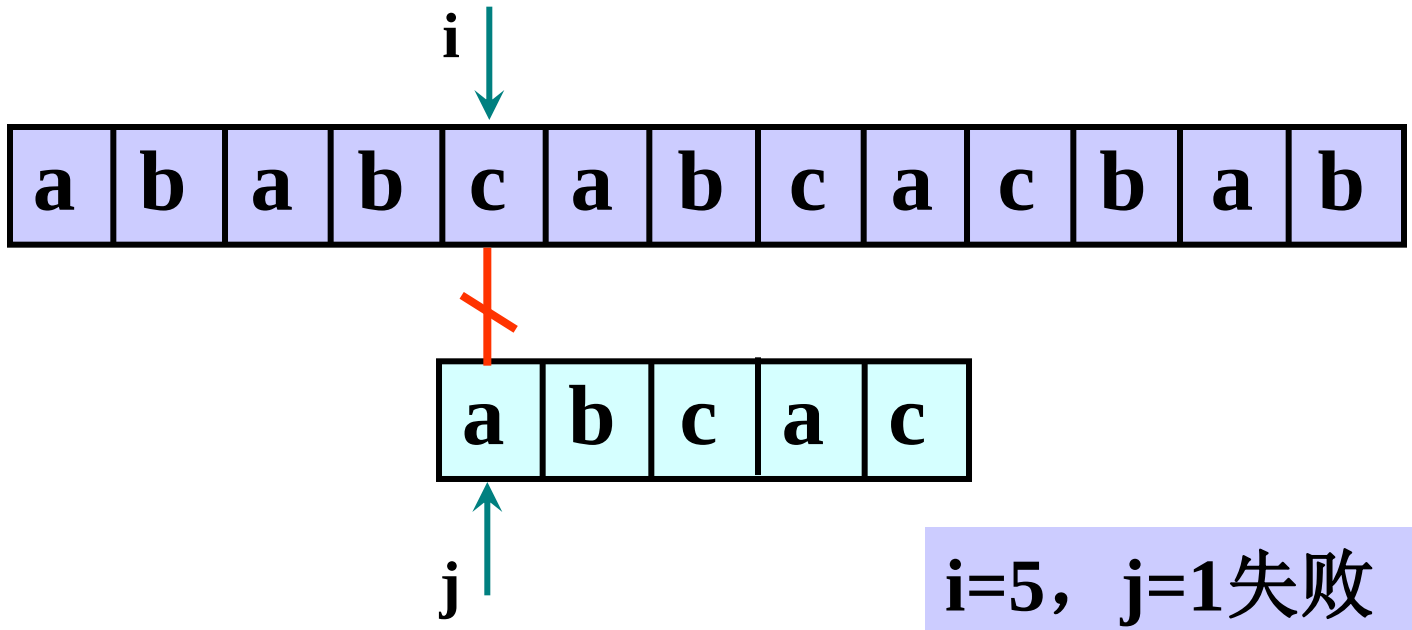


i=4，j=1失败
i“回溯”到5，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第5趟

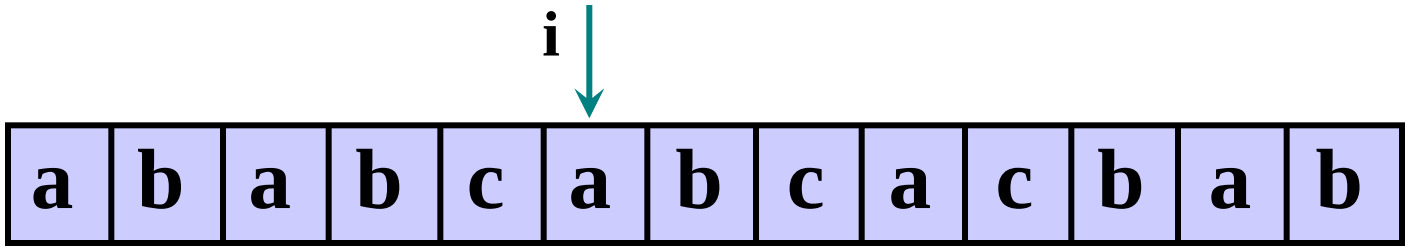


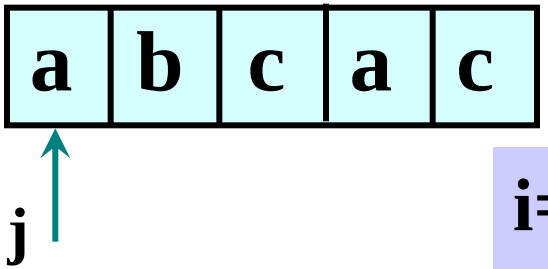
i=5，j=1失败
i“回溯”到6，j回溯到1

BF模式匹配算法

例：主串S="ababcabcacbab"，模式T="abcac"

第5趟



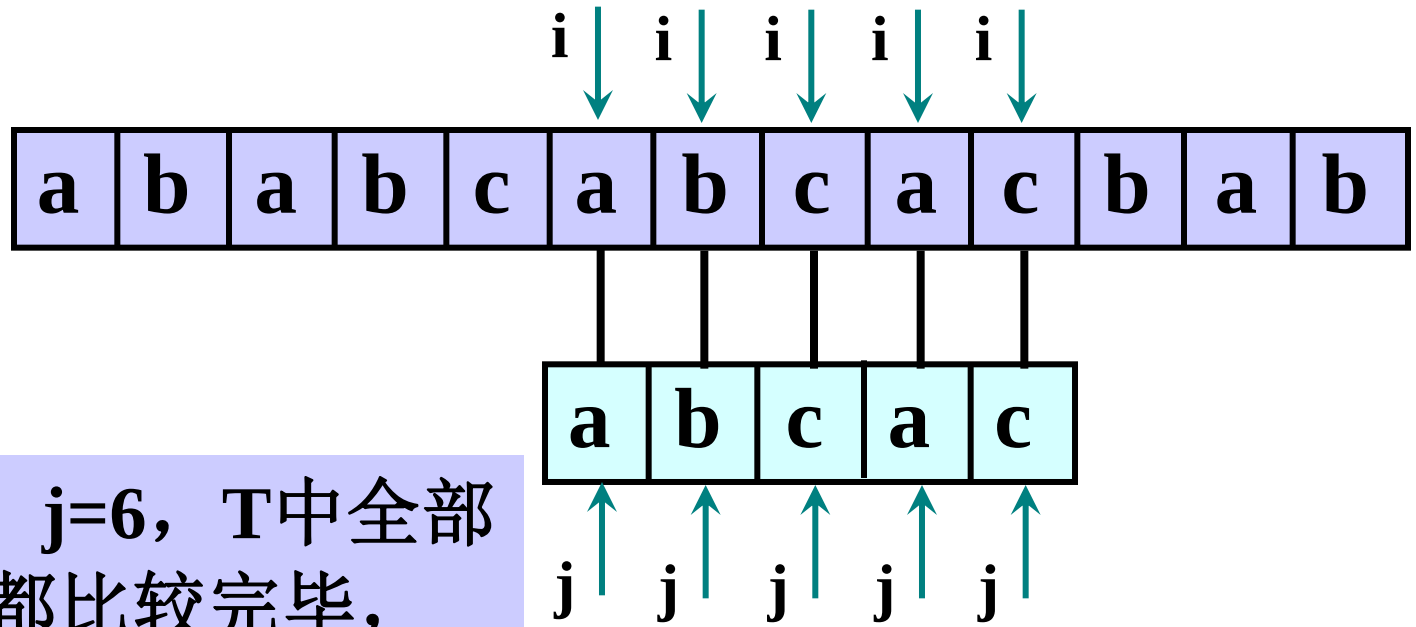


i=5，j=1失败
i“回溯”到6，j回溯到1

BF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。

BF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；

BF匹配算法实现

```
#include “String.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
    nt 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);
 }
```

BF模式匹配算法分析

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*="*aaaaaaaaaabcd*ccccc"

T="*bcd* "

BF模式匹配算法分析

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：

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

BF模式匹配算法分析

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*="*aaaaaaaaaaaabccccc*"

T="*aaab*"

BF模式匹配算法分析

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)×*m*次，第*i*趟成功的匹配共比较了*m*次，所以总共比较了*i*×*m*次，因此（一般地，m<<n）

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

二、KMP 模式匹配算法

例1， a a a a a a a a a a b

a a a a a a b

×

a a a a a a b

⋮

a a a a a a b

例2， a b c d e f a b c d e f f

a b c d e f f

×

a b c d e f f

1、KMP模式匹配算法思想

S $s_0\ s_1 \cdots s_{i-j-1}\ s_{i-j}\ s_{i-j+1}\ s_{i-j+2} \cdots s_{i-2}\ s_{i-1}\ s_i \cdots s_{n-1}$

$\|\ \ \|\ \ \|\ \ \|\ \ \|\ \ \times$

P $p_0\ p_1\ p_2 \cdots p_{j-2}\ p_{j-1}\ p_j$

则有 $s_{i-j}\ s_{i-j+1}\ s_{i-j+2} \cdots s_{i-1} = p_0\ p_1\ p_2 \cdots p_{j-1}$ (1)

如果 $p_0\ p_1 \cdots p_{j-2} \neq p_1\ p_2 \cdots p_{j-1}$ (2)

则立刻可以断定

$p_0\ p_1 \cdots p_{j-2} \neq s_{i-j+1}\ s_{i-j+2} \cdots s_{i-1}$

(朴素匹配的)下一趟一定不匹配，可以跳过去

同样，若 $p_0\,p_1\cdots p_{j-3} \neq p_2\,p_3\cdots p_{j-1}$

则再下一趟也不匹配，因为有

$$p_0\,p_1\cdots p_{j-3} \neq s_{i-j+2}\,s_{i-j+3}\cdots s_{i-1}$$

直到对于某一个“k”值(首尾串长度)，使得

$$p_0\,p_1\cdots p_k \neq p_{j-k-1}\,p_{j-k}\cdots p_{j-1}$$

且 $p_0\,p_1\cdots p_{k-1} = p_{j-k}\,p_{j-k+1}\cdots p_{j-1}$

则 $p_0\,p_1\cdots p_{k-1} = s_{i-k}\ s_{i-k+1}\ \cdots\ s_{i-1}\ s_i$

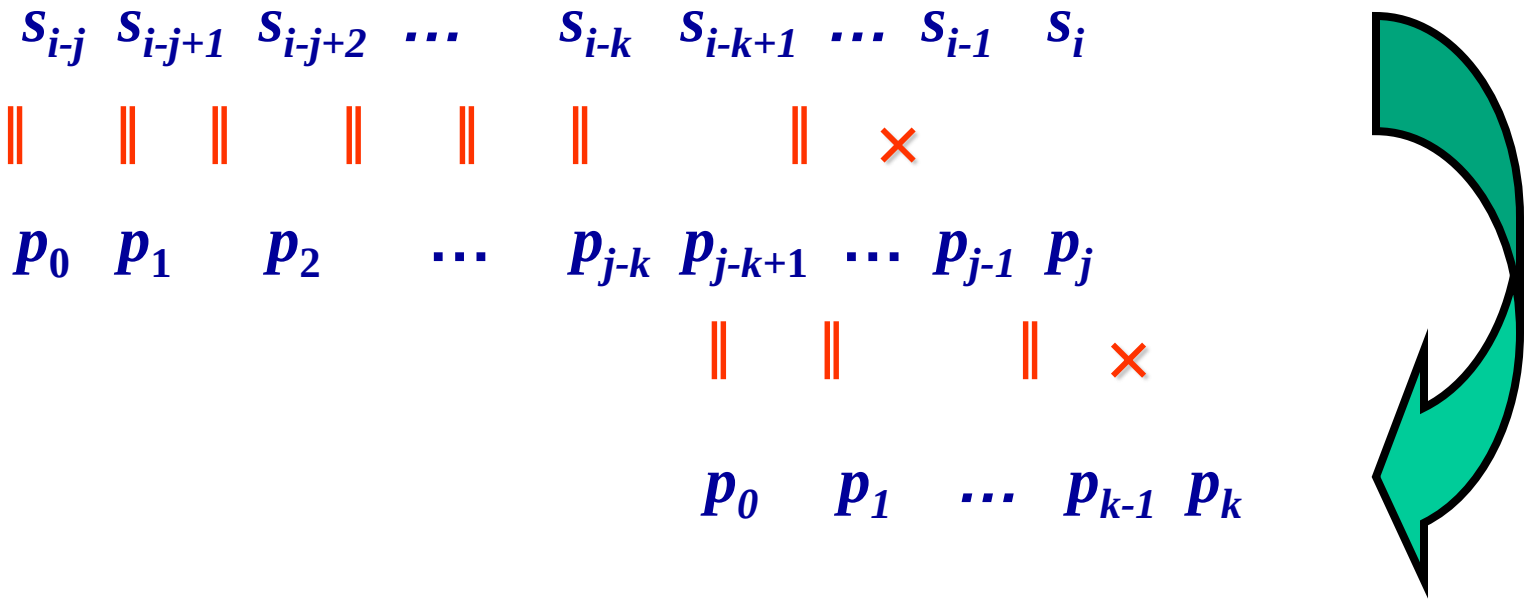
$\parallel\ \parallel\ \parallel\ \times$

$p_{j-k}\ p_{j-k+1}\ \cdots\ p_{j-1}\ p_j$

$\parallel\ \parallel\ \parallel\ \times$

模式右滑 j−k位 $p_0\ \ p_1\ \ \cdots\ p_{k-1}\ p_k$

模式右滑j-k位


s_{i-j} s_{i-j+1} s_{i-j+2} … s_{i-k} s_{i-k+1} … s_{i-1} s_i
p_0 p_1 p_2 … p_{j-k} p_{j-k+1} … p_{j-1} p_j
p_0 p_1 … p_{k-1} p_k

2、字符串的特征向量N

- 设模板P由m个字符组成：

  记为$P = q_0\, q_1 q_2 q_3 \cdots\cdots q_{m-1}$

- 令特征向量N用于表示模板P的字符分布特征，并简称N向量。它和P同长，由m个特征数$n_0 \cdots n_{m-1}$非负整数组成：

  记为$N = n_0\, n_1 n_2 n_3 \cdots\cdots n_{m-1}$

- 下面说明n_i的含义和它的递归定义：

  列出模板P开头的任意t个字符，把它称为P的前缀子串。

  $q_0q_1q_2\cdots q_{t-1}$

  在P的第i位置的左边，也取出t个字符，称为i位置的左子串。

  $q_{i-t+1}\cdots q_{i-2}q_{i-1}q_i$

- 计算特征数n_i
  - 设法求出最长的（t最大的）能够与前缀子串匹配的左子串（简称第i位的最长前缀串）。
  - t就是要求的特征数n_i。

- 特征数n_i（ $0 \leqslant n_i \leqslant i$ ）是递归定义的，定义如下：

  ① $n_0=0$，对于$i > 1$的n_i，假定已知前一位置的特征数n_{i-1}，并且$n_{i-1}= k$；

  ② 如果$q_i = q_k$，则$n_i = k+1$；

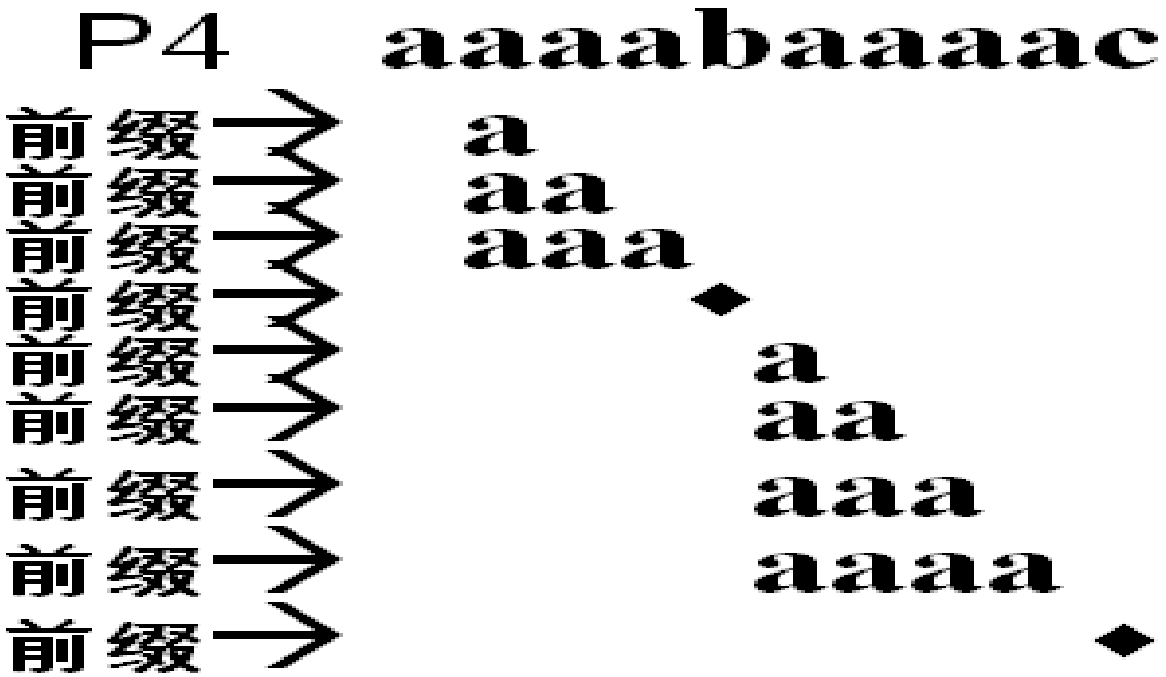
  ③ 当$q_i \neq q_k$ 且 $k \neq 0$时，则 令$k = n_{k-1}$；

  让③循环直到条件不满足；

  ④当$q_i \neq q_k$ 且 $k = 0$时，则 $n_i = 0$;

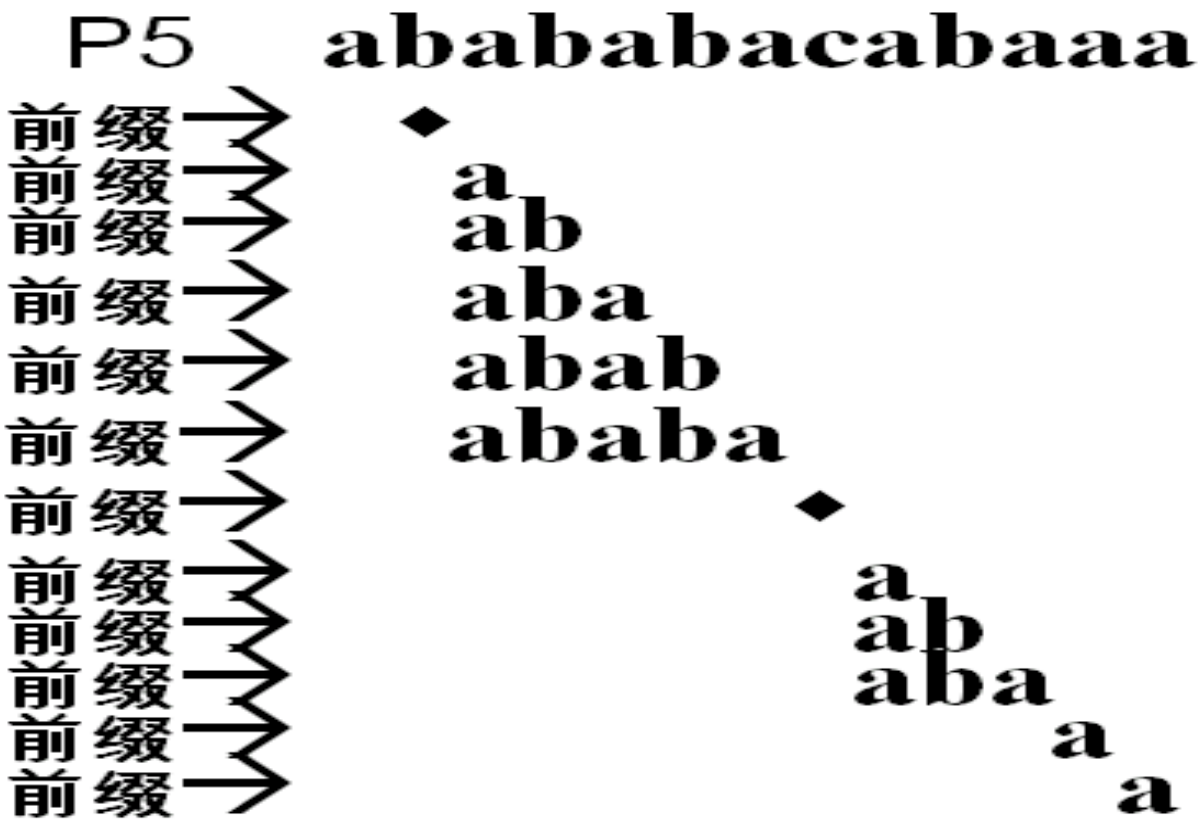
- 例如：

P4 = "aaaabaaaaac"
N = “0123012340"



P5 = "abababacabaaa"

N = “001234501231 1"



- 计算向量N的算法

```
int *Next(String P){
   int m = P.strlen();     //m为模板P的长度
   assert( m > 0);     //若m＝0，退出
   int *N = new  int[m];  // 动态存储区开辟整数数组
   assert( N != 0);    //若开辟存储区域失败，退出
   N[0] = 0;
   for( int i =1 ; i < m ;  i++ ){ //分析P的每个位置i
      int k = N[i-1];  //第(i-1)位置的最长前缀串长度
      //以下while语句递推决定合适的前缀位置k
      while( k > 0 && P[i] != P[k] ) k = N[k-1];
      //根据P[i]比较第k位置前缀字符，决定N[i]
      if(P[i] == P[k])  N[i] = k+1;
      else              N[i] = 0 ;
   }
   return N;
}
```

● KMP模式匹配算法

```
int KMP_FindPat(String S, String P, int *N, int
    startindex) {
  //假定事先已经计算出P的特征数组N，作为输入参数
  // S末尾再倒数一个模板长度位置
   int LastIndex = S.size - P.size;
   if ((LastIndex - startindex) < 0)
     return (-1);     //startindex过大，匹配无法成功
   int i;                // i 是指向S内部字符的游标，
   int j = 0;            // j 是指向P内部字符的游标，
 // S游标i循环加1
   for (i = startindex; i < S.size; i++) {
        //若当前位置的字符不同，则用N循环求当前的j，
       //用于将P的恰当位置与S的i位置对准
        while (P.str[j] != S.str[i]  &&  j > 0)   j = N[j-1];
        //P[j]与S[i]相同，继续下一步循环
         if (P.str[j] == S.str[i]) j++;
       //匹配成功，返回该S子串的开始位置
          if( j == P.size)  return (i - j +1);
    };
    return (-1);   //P和S整个匹配失败，函数返回值为负
}
```

讨论：如果“i < LastIndex”，
那么后面的就匹配不到。
例如，aaaaaaaaaab
aaaaaab
S.size=11，P.size=7，
LastIndex = 4;
“i = 4，j = 0”， 匹配‘a’，
接着，“i = 5，j = 1”就进行不了。

KMP模式匹配示例（一）

```
     0  1  2  3  4  5  6
P =  a  b  a  b  a  b  b
N =[0  0 1  2  3  4  0]

    0  1  2  3  4  5  6  7  8  9 10 11 12
S= a  b  a  b  a  b  a  b  a  b  a  b  b…
    a  b  a  b  a  b  b
                      X  i=6, j=6, N[j-1]=4
          a  b  a  b  a  b  b
                         X i=8, j=6, N[j-1] =4
             a  b  a  b  a  b  b
                            X i=10, j=6, j'=4
                a  b  a  b  a  b  b
                                  √
```

```
     0  1  2  3  4  5  6  7  8  9
P =  a  a  a  a  b  a  a  a  a  c
N =[0  1  2  3  0  1  2  3  4  0]
```

```
     0 1 2 3 4  5  6 7 8 9 10 11 12  13  14
S= a a b a a  a  a a a b a  a  a  a  c b c a a b a b c
     a a a a b  a  a a a c
          X  i=2, j=2, N[j-1]=1
       a a a a  b  a a a a c
          X i=2, j=1, N[j-1]=0
            a a  a  a b a a a a c
                        X i=7, j=4, N[j-1]=3
                  a  a  a a b a a  a  a  c
                              X i=8, j=4, N[j-1]=3
                     a  a  a a b a  a  a  a  c
```

√

0 1 2 3 4 5 6 7 8 9

P = a a a a b a a a a c

N =[0 1 2 1 0 1 2 3 4 0]

X (不是最长的，应该是3)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

S= a a b a a a a a b a a a a c b c a a b a b c

a a a a b a a a a c

X i=2, j=2, N[j-1]=1

a a a a b a a a a c

X i=2, j=1, N[j-1]=0

a a a a b a a a a c

X i=7, j=4, N[j-1]=1

a a a a b a a a a c

…… （错过了！）

KMP算法分析

- 两重循环
  - for循环最多执行n＝S.size次
    - 其内部的while循环，最长循环次数是m＝P.size次。
- 初看起来其时间开销也可能达到O(n×m)。

- 循环体中“j=N[j-1];”语句的执行次数不能超过n次。否则，
  - 由于“j= N[j-1];”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++”
  - 那么，如果“j= N[j-1];”的执行次数超过n次，最终的结果必然使得j为负数。这是不可能的。
- 同理可以分析出求next数组的时间为O(m)
- 因此，KMP算法的时间为O(n+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

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

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}和和一个列后继a_{32}。

数组——线性表的推广

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

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

其中：

$A_i=(a_{1i}, a_{2i}, \ldots\ldots, a_{mi})$

$(1\le i\le n)$

二维数组是数据元素为线性表的线性表。

数组的基本操作

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。

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

x

删除数组中
第1行第2列元素。

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

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

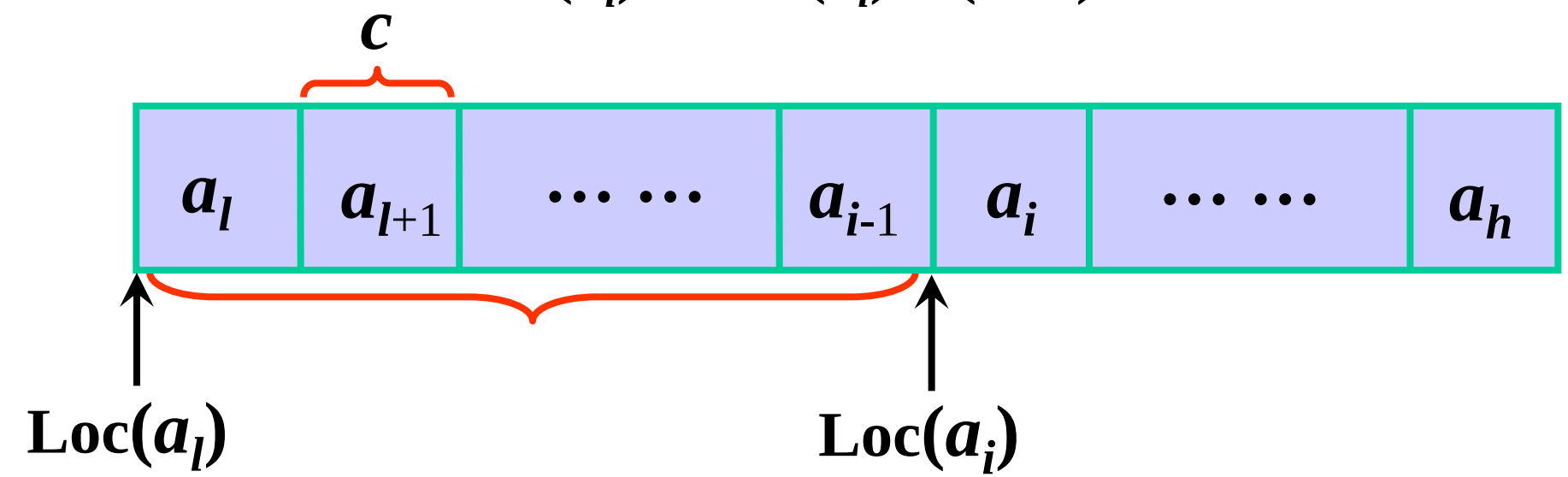
数组应该采用何种方式存储？

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

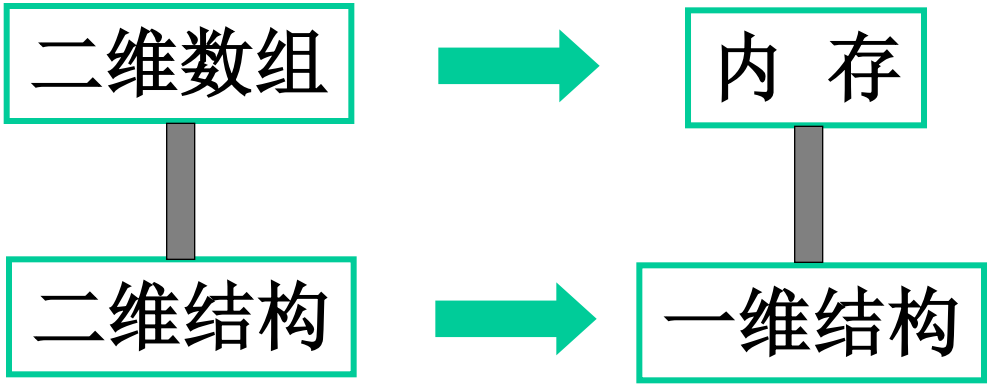
设一维数组的下标的范围为闭区间［l，h］，每个数组元素占用 c 个存储单元，则其任一元素 a_i 的存储地址可由下式确定：

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



在C语言中，如何计算Loc(a_i)？

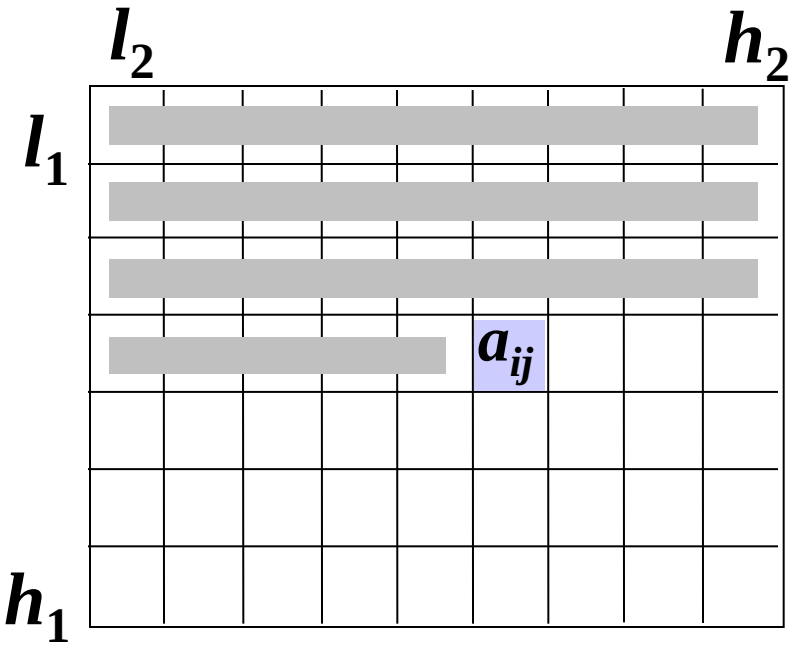
数组的存储结构与寻址——二维数组



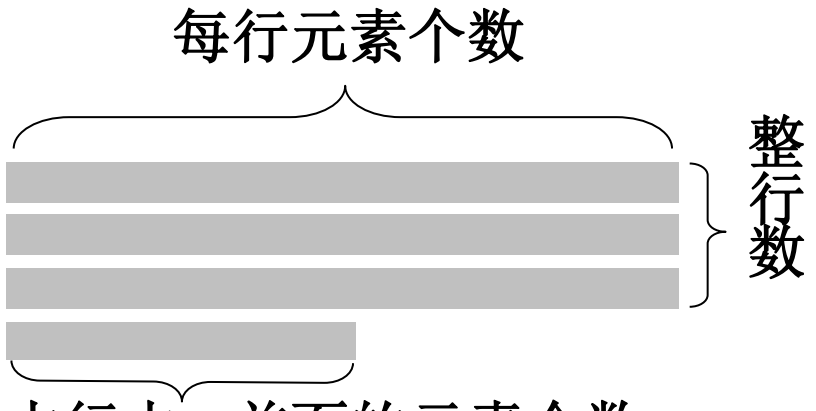
常用的映射方法有两种：

- 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



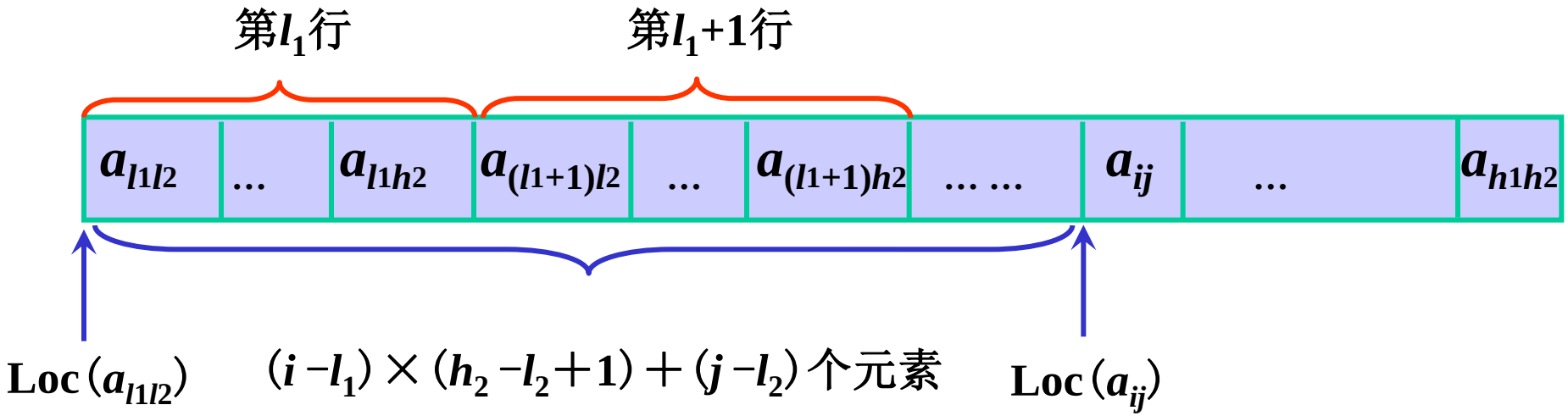
a_{ij}前面的元素个数

=阴影部分的面积

=整行数×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



$$\mathrm{Loc}(a_{ij})=\mathrm{Loc}(a_{l1l2})+((i-l1)\times(h2-l2+1)+(j-l2))\times c$$

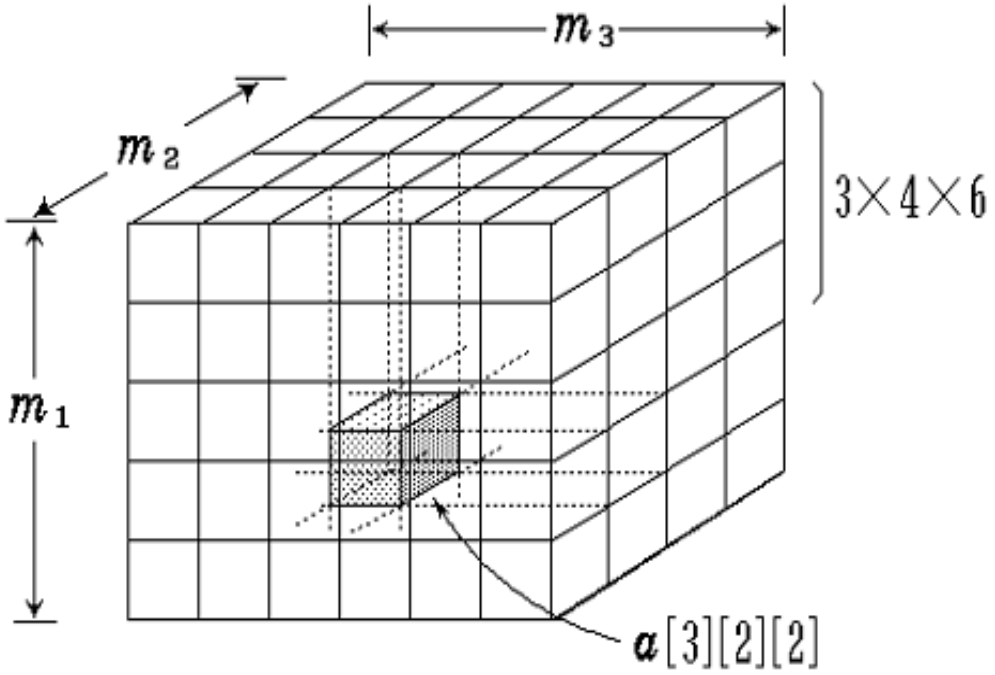
练习：习题4，1（4）

按列优先存储的寻址方法与此类似。

在C语言中，如何计算$\mathrm{Loc}(a_{ij})$？

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。

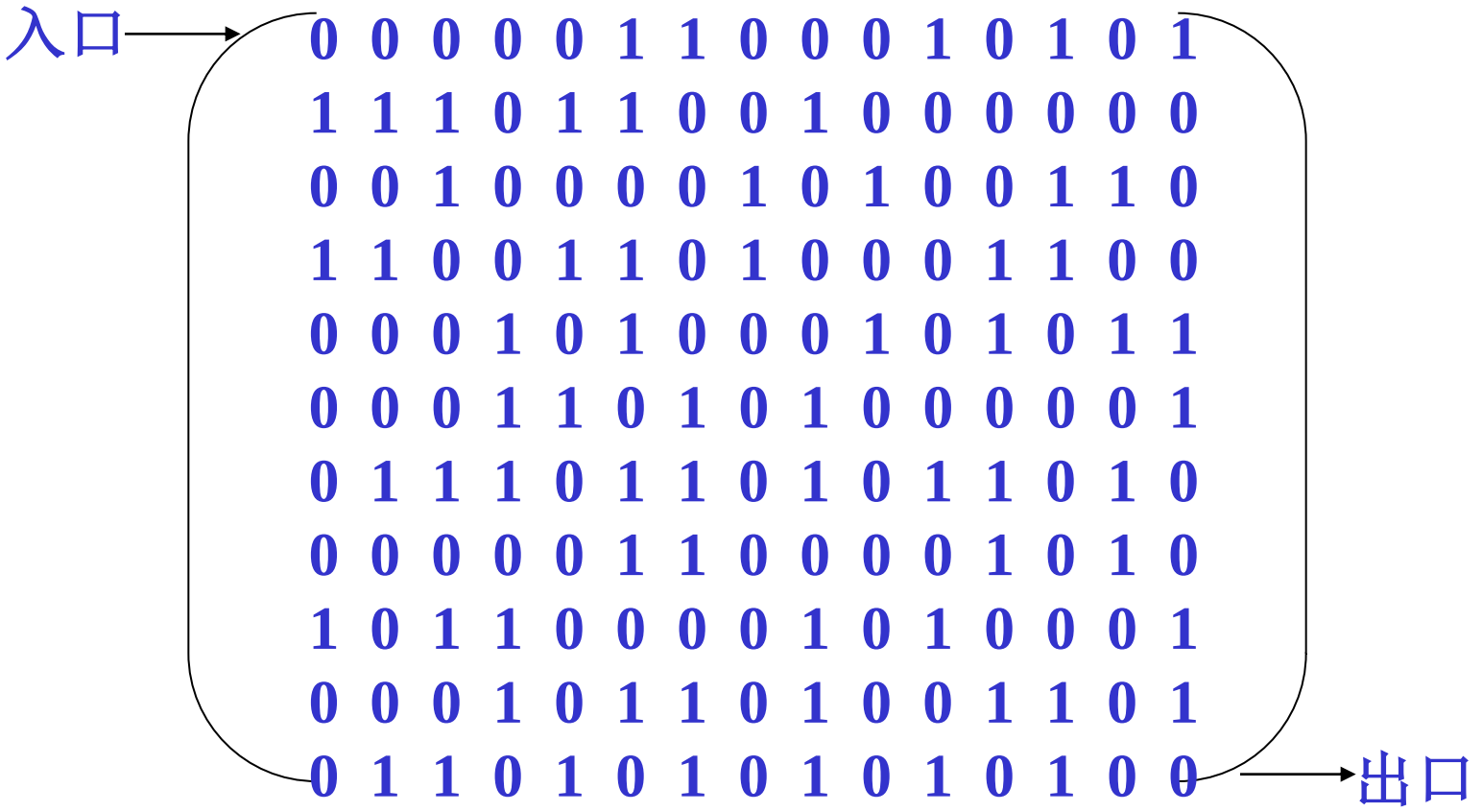


$$\mathrm{Loc}(a_{ijk}) = \mathrm{Loc}(a_{000}) + (i \times m_2 \times m_3 + j \times m_3 + k) \times c$$

数组的应用——迷宫问题

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。


入口
0 0 0 0 0 1 1 0 0 0 1 0 1 0 1
1 1 1 0 1 1 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 1 0 1 0 0 1 1 0
1 1 0 0 1 1 0 1 0 0 0 1 1 0 0
0 0 0 1 0 1 0 0 0 1 0 1 0 1 1
0 0 0 1 1 0 1 0 1 0 0 0 0 0 1
0 1 1 1 0 1 1 0 1 0 1 1 0 1 0
0 0 0 0 0 1 1 0 0 0 0 1 0 1 0
1 0 1 1 0 0 0 0 1 0 1 0 0 0 1
0 0 0 1 0 1 1 0 1 0 0 1 1 0 1
0 1 1 0 1 0 1 0 1 0 1 0 1 0 0
出口

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为扩大了的迷宫矩阵
{// 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

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

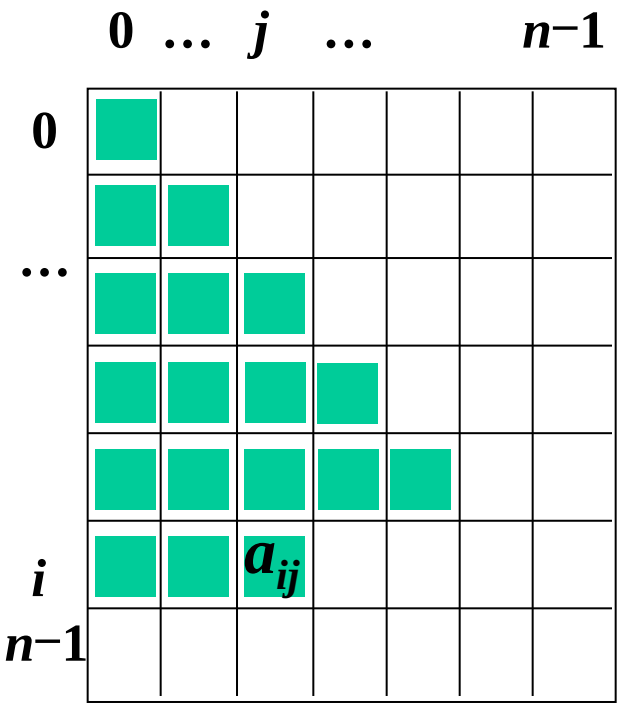
对称矩阵特点：

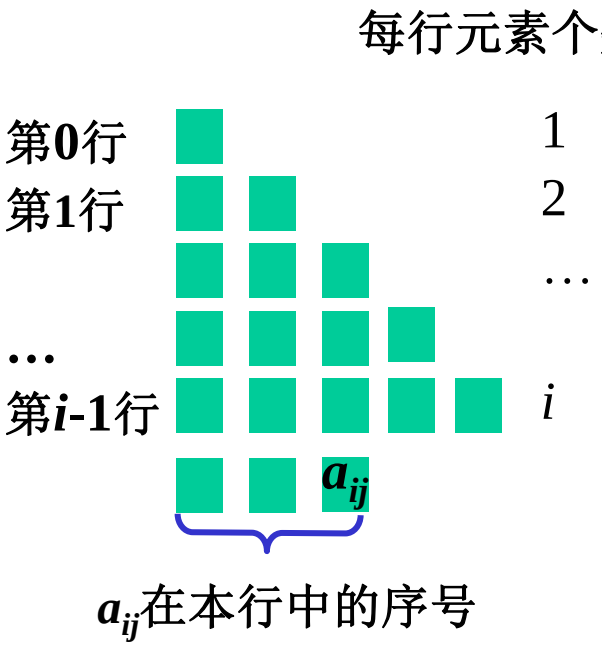
$a_{ij}=a_{ji}$

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_{ij}在一维数组中的序号

=阴影部分的面积

$= i \times (i+1)/2 + j + 1$

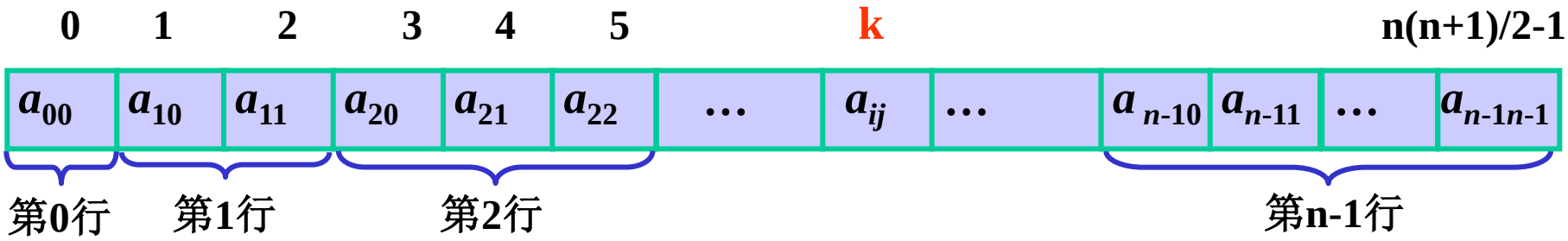
∵一维数组下标从0开始

∴a_{ij}在一维数组中的下标

$k = i \times (i+1)/2 + j$

(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

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

(a) 下三角矩阵

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

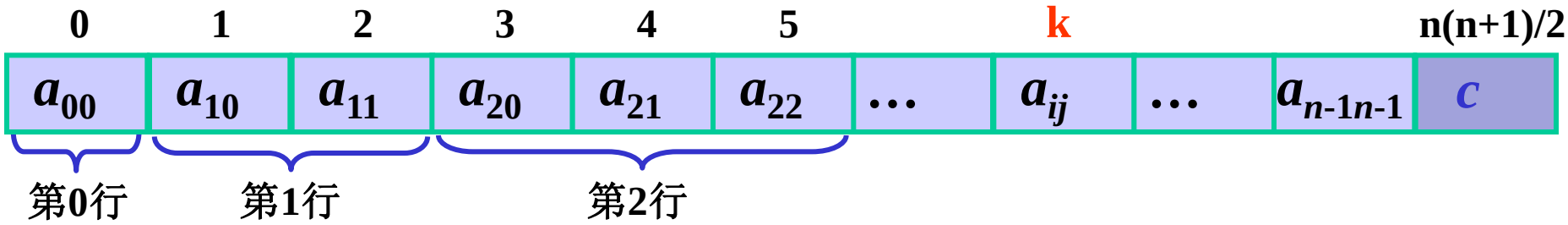
(b) 上三角矩阵

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：

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

3、稀疏矩阵的压缩存储

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

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))

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

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

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：

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

	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）

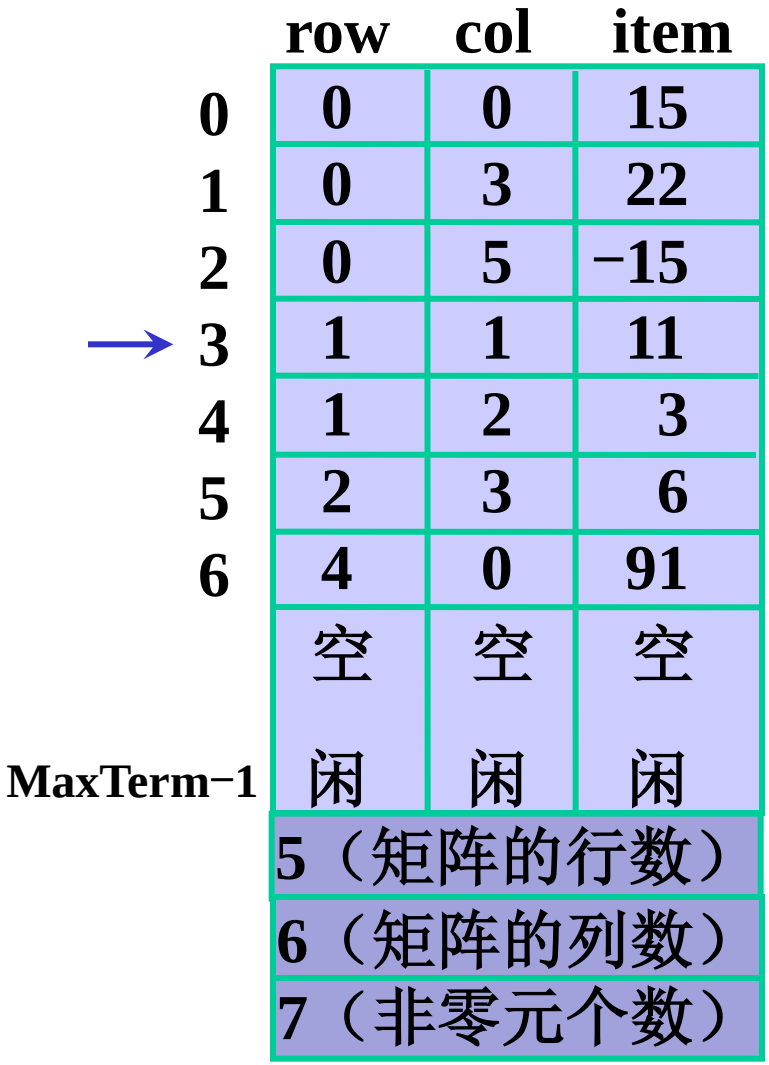
7（非零元个数）

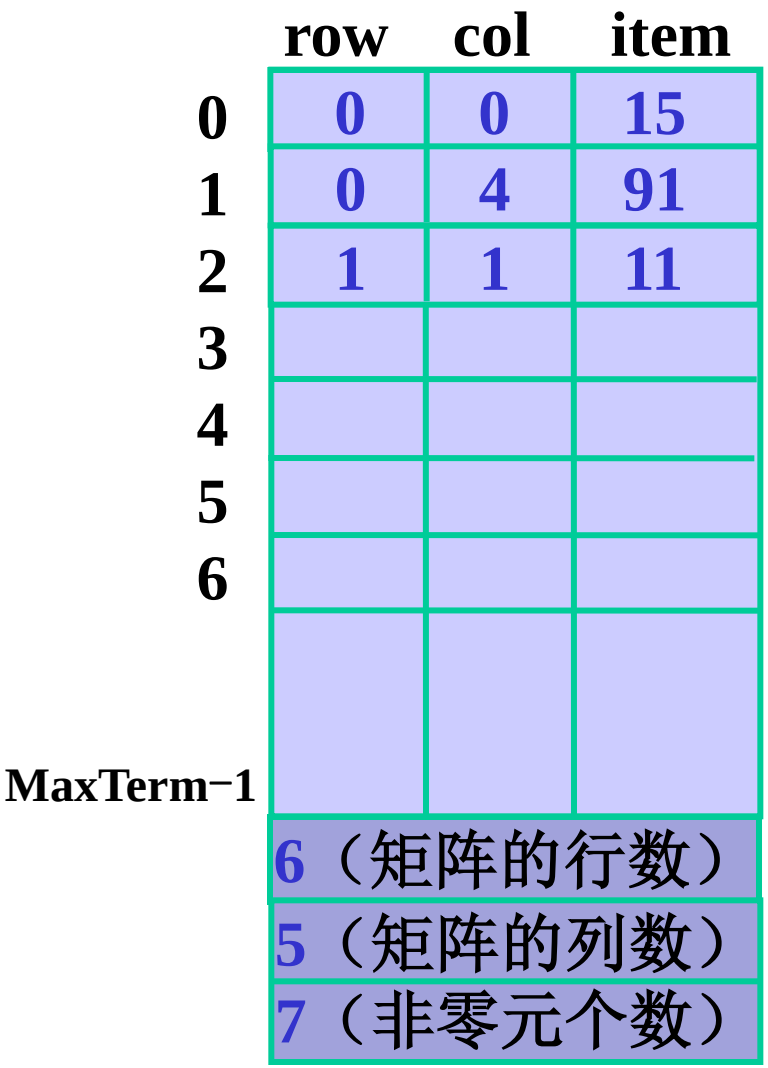
在矩阵A中查找第0列非零元，顺序存储到矩阵B中

	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



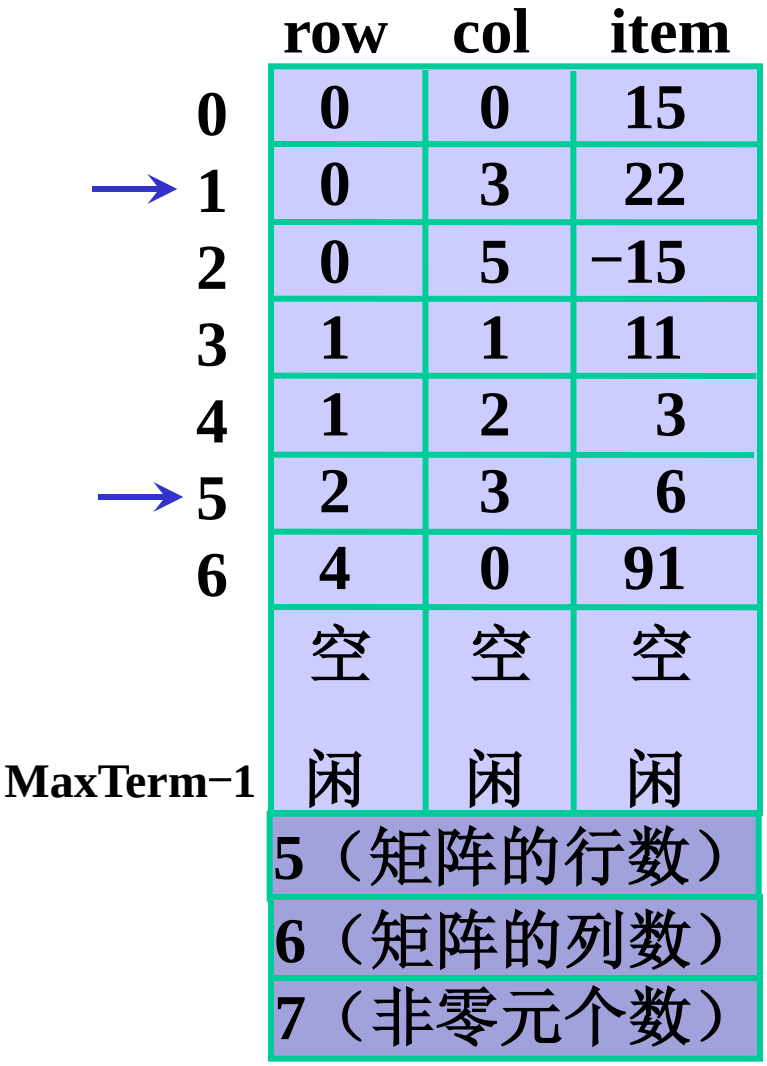


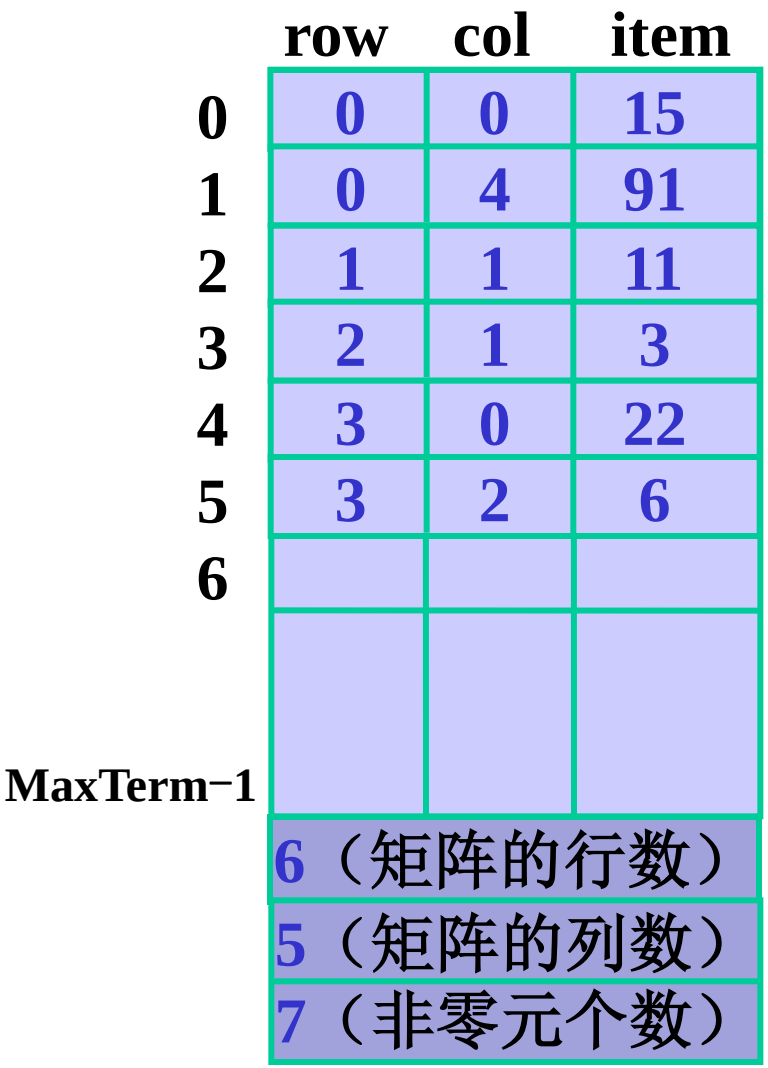
在矩阵A中查找第2列非零元，顺序存储到矩阵B中

	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
MaxTerm−1	空闲	空闲	空闲
	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





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
MaxTerm−1	空闲	空闲	空闲
	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
MaxTerm−1	空闲	空闲	空闲
	5（矩阵的行数）		
	6（矩阵的列数）		
	7（非零元个数）		

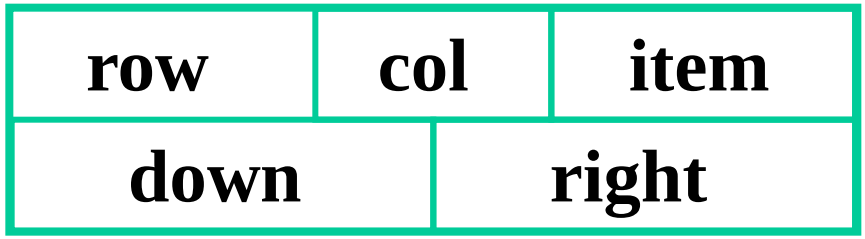
	row	col	item
0	0	0	15
1	0	4	91
2	1	1	11
3	2	1	3
4	3	0	22
5	3	2	6
6	5	0	-15
MaxTerm−1			
	6（矩阵的行数）		
	5（矩阵的列数）		
	7（非零元个数）		

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++；

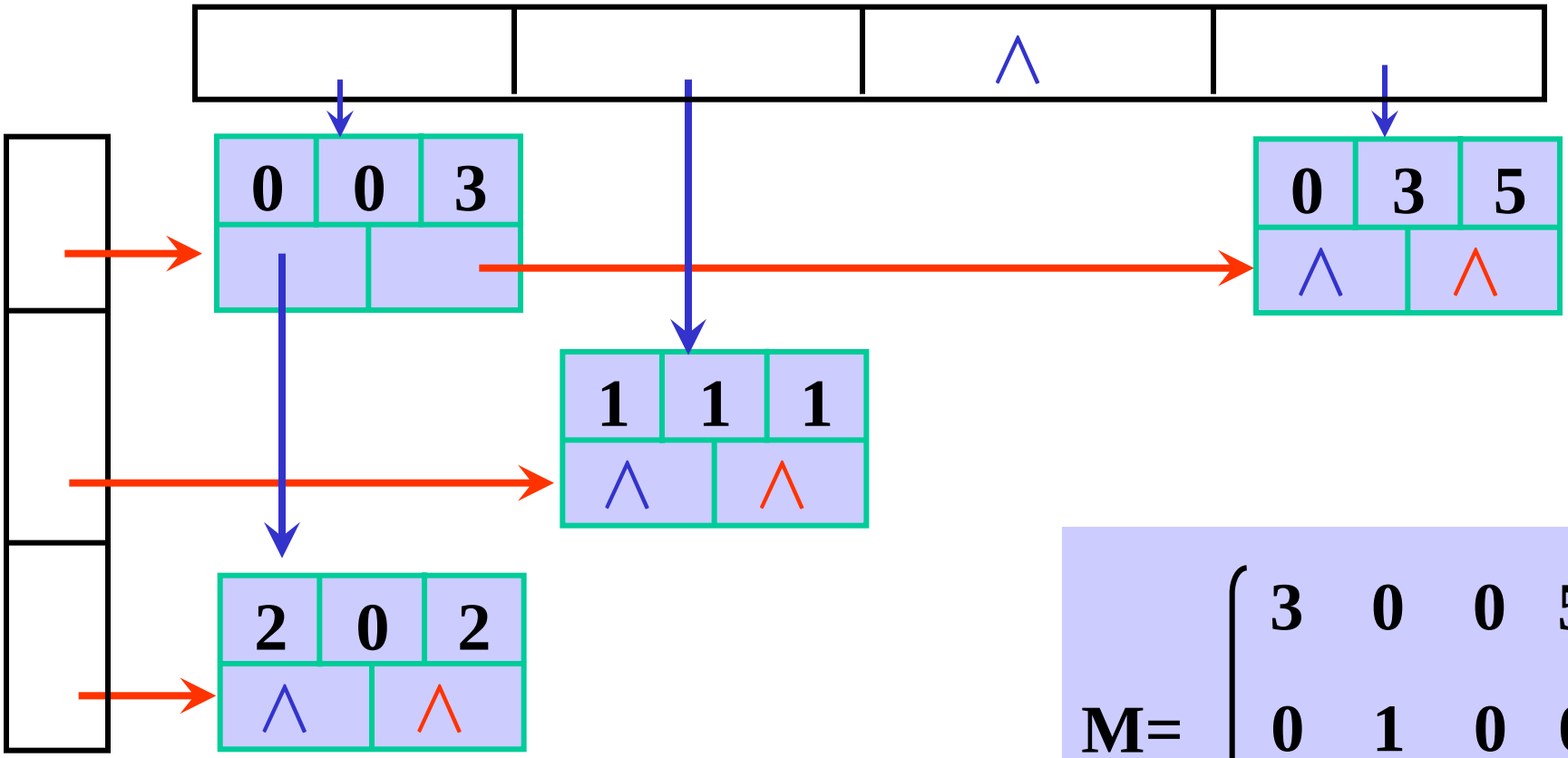
稀疏矩阵的压缩存储——十字链表

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：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



$$M=\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

本章作业

习题4（P96）：4,5