

# 第三章 字符串

## 本章主要内容：

- 字符串及其运算
- 字符串的存储表示（顺序、链接）
- 模式匹配
  - 朴素的模式匹配
  - 无回溯的模式匹配（KMP算法）
- 串操作应用示例
  - 文本编辑
  - 词索引表建立

## 3.1 字符串及其运算

### 1. 基本概念

- **字符串**：简称为串，是零个或多个字符组成的有限序列。  
一般记为： $s = "s_0s_1 \dots s_{n-1}"$  ( $n \geq 1$ )，其中 $s$ 为串名，每个字符 $s_i$  ( $0 \leq i \leq n-1$ )可以是字母、数字或其它字符
- **串长**：字符串中字符个数
- **空串**：长度为零的字符串，记为 $s = ""$  [注意与**空格串**的区别]。
- **主串与子串**：字符串 $s_1$ 中连续字符组成的子序列 $s_2$ 被称为 $s_1$ 的**子串**， $s_1$ 为 $s_2$ 的**主串**。

- **字符在串中的位置**：该字符在串中第一次出现的位置[下标]。
- **子串在主串中的位置**：该子串的第一个字符在主串中的位置。
- **两个字符串相等的充分必要条件**：长度相等，对应位置上字符相同。

## 2. 字符串的基本运算

➤ 创建一个空串；

`s = ""; strcpy(s, "");`

➤ 判断是否为空串；

`strlen(s) == 0 (s[0] == 0);`

➤ 求串长；

`strlen(s)`

➤ 删除子串；

➤ 插入子串；

- 字符串的联结（多个串联结成一个新串）； **strcat(s1, s2)**
- 取子串[求字符串中从第i个字符开始的连续m个字符构成的子串]； **strncpy(s2, &s1[i-1], m)**
- 定位[子串s2在主串s1中第一次出现的位置]；  
**strstr(s1, s2);**
- 字符串的比较运算； **strcmp(s1, s2), strncmp(s1, s2, n)**
- 字符串拷贝运算； **strcpy(s1, s2), strncpy(s1, s2, n)**

**\_strupr(s): 转换为大写**

**\_strlwr(s): 转换为小写**

## 3.2 字符串的存储表示

串是一种**特殊的线性表**，其每个结点仅有一个字符组成，线性表的存储方法同样适用于字符串的存储。但是在选择存储结构时应考虑基本运算的实现。例如，插入、删除时，采用顺序结构不方便，但访问字符时（尤其连续多个）非常容易。

**顺序存储：**串的字符顺序地存在连续的单元中；

**链接存储：**单链表方式实现。

下面讨论两种存储结构下基本运算的实现。

## 3.2.1 字符串的顺序存储及其基本运算

### 1. 顺序存储

字符串中的字符顺序存储在一组地址连续的存储单元中，最后一个字符后面可加一个特定的结束标志（如：‘\0’等），该结束标志占用存储空间但不计入串长。

#### C语言描述：

```
#define MAXNUM <串允许的最大字符个数>
struct SeqString
{   char c[MAXNUM];           /* 字符数组           */
    int  n;                   /* 串长 < MAXNUM */
};
typedef struct SeqString *PSeqString;
```

#### 示例：

s = “abcdefg”

s.n = 7  
s.c数组  
下标

a	b	c	d	e	f	g	\0		
0	1	2	3	4	5	6	7		

## 2. 顺序存储结构基本运算

- 创建空串
- 联结两个串
- 取子串
- 删除子串
- 插入子串
- 子串定位[模式匹配中详细讨论]
- 求串长

### 基本运算实现算法

## 3.2.2 字符串的链接存储及其基本运算

### 1. 链接存储

每个结点包含两个域：  
字符域、指针域。

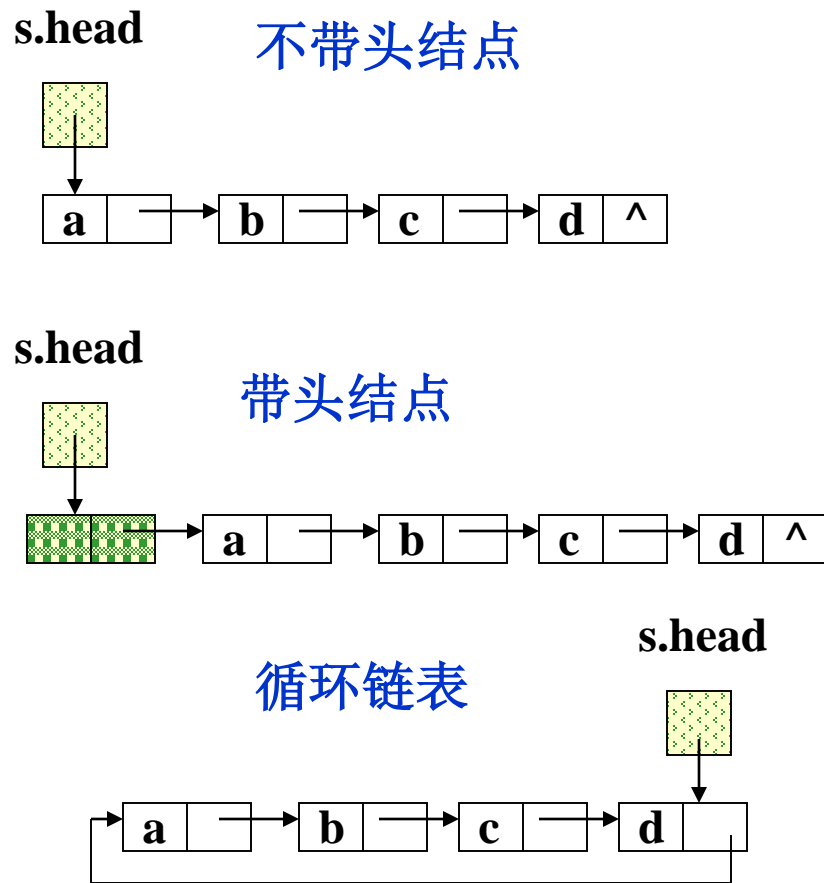
C语言描述：

// 结点结构

```
struct StrNode
{
    char c;
    struct StrNode *link;
};
typedef struct StrNode *PStrNode;
```

// 链串结构

```
struct LinkString
{
    PStrNode head;
};
typedef struct LinkString *PLinkString;
```





## 2. 链接存储结构基本运算

- 创建空串
- 联结两个串
- 取子串
- 删除子串
- 插入子串
- 子串定位[模式匹配中详细讨论]
- 求串长

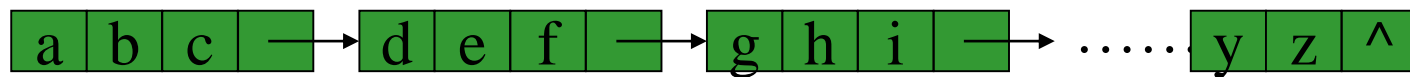
### 基本运算实现算法[带头结点的单链表]

**需要注意的是：**上述链接存储结构虽然操作实现简单，但存储密度太低。改进办法：每个结点存放多个字符。改进后虽然提高了存储密度，但操作实现变的复杂。每个结点存放多少个字符？最后结点存放字符的个数与前面的不同，操作实现需要特殊处理。

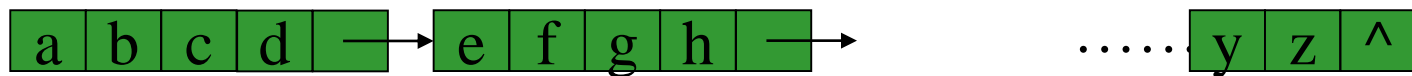
双字符



三字符



四字符



## 3.3 模式匹配

设有两个串 $t$ 和 $p$ :  $t = t_0t_1\dots t_{n-1}$  ,  $p = p_0p_1\dots p_{m-1}$   
其中  $1 < m \leq n$  (通常  $m \ll n$ )

**模式匹配**的目的是在 $t$ 中找出和 $p$ 相同的子串。  
此时， $t$ 称为“**目标**”，而 $p$ 称为“**模式**”。

模式匹配的结果有两种：

**匹配成功**： $t$ 中存在等于 $p$ 的子串，返回子串在 $t$ 中的位置

**匹配失败**：返回一个特定的标志（如-1）。

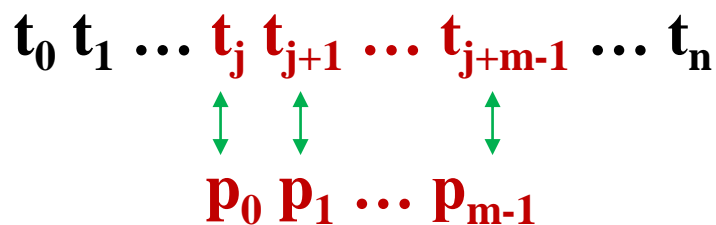
**模式匹配**是一个比较复杂的字符串操作，下面的讨论是基于字符串的**顺序存储**结构进行。

- 朴素的模式匹配方法
- **KMP**无回溯的模式匹配方法

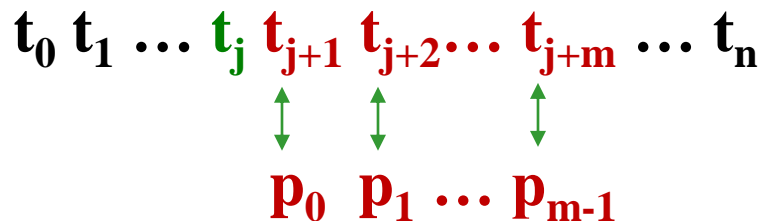
### 3.3.1 朴素的模式匹配（BF算法）

#### 1. 匹配思想：

一趟匹配：选择 $t$ 中的连续 $m$ 个字符与模式 $p$ 的 $m$ 个字符一一对应比较。BF算法的选择是从左往右一组一组地进行。







对于上述状态，如果对于所有的 $i$  ( $0 \leq i \leq m-1$ )，皆有 $t_{j+i} = p_i$ ，则匹配成功，直接返回位置 $j$ ；否则，如果存在某个 $i$ 满足 $t_{j+i} \neq p_i$ ，则此趟匹配失败，这时将 $p$ 右移一个字符，进行下一趟匹配：



直到匹配成功或无法移动 $p$ 与 $t$ 继续比较为止（此时，匹配完全失败）。

## 2. 匹配示例： t=“abbaba”，p=“aba”

t	a b b a b a	a b b a b a	a b b a b a	a b b a b a
p	 a b a	 a b a	 a b a	 a b a
	p <sub>2</sub> , t <sub>2</sub> 不等	p <sub>0</sub> , t <sub>1</sub> 不等	p <sub>0</sub> , t <sub>2</sub> 不等	匹配成功

## 3. 匹配算法（基于字符串的顺序存储结构）

```
int index(SeqString t, SeqString p)
```

```
{
    int i=0, j=0;                                //初始化
    while ( i < p.n && j < t.n)                  //反复比较
    {
        if ( p.c[i] == t.c[j]) { i++; j++; }    //继续匹配下一个字符
        else
        { j = j-i+1; i = 0; }                    //主串、子串i、j值回溯，重新开始下一次匹配
    }
    if ( i >= p.n)    return (j-p.n);            //匹配成功，返回位置【下标】
    else              return -1;                 //匹配失败【下标： -1】
}
```

## 4. 算法时间效率分析:

朴素模式匹配算法简单, 容易理解, 但效率不高。主要原因是: 一旦比较不等,  $p$  右移一个字符并且下次从  $p_0$  开始重新进行比较, 对于目标  $t$ , 存在回溯现象。(也做了许多重复比较)

**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  $O(m*n)$ , 基本运算为比较运算。

## 5. 进一步分析:

造成朴素模式匹配效率低的主要原因是一趟匹配失败后的目标串**回溯操作**。实质上这些回溯操作是可以避免的。

改进办法有很多，关键的问题是如何减少或避免回溯，进而减少总的比较次数。**试想：当某次匹配失败时，下次匹配时是否可以利用前面已经比较的结果？**

$t_i t_{i+1} t_{i+2} \dots t_{i+k} \dots$

$\neq$

$t_i t_{i+1} t_{i+2} \dots t_{i+k} \dots$

?

$p_0 p_1 p_2 \dots p_k \dots$

$p_0 \dots p_k' \dots p_k \dots$

例如：前面的例子中， $t = \text{“abbaba”}$ ， $p = \text{“aba”}$ 。

第一趟匹配时有 $p_0 = t_0$ ， $p_1 = t_1$ ， $p_2 \neq t_2$ ；由于 $p_0 \neq p_1$ 可以推出 $p_0 \neq t_1$ ，所以 $p$ 右移一位后的比较一定不等；

再有 $p_0 = p_2$ ，可以推出 $p_0 \neq t_2$ ，所以将 $p$ 继续右移一位后的比较也一定不等，因此**可以由第一趟匹配直接跳过2、3趟匹配进入第4趟匹配。**

a b b a b a  
 $\updownarrow \updownarrow \nearrow$   
 a b a

a b b a b a  
 $\nearrow$   
 a b a

a b b a b a  
 $\nearrow$   
 a b a

a b b a b a  
 $\updownarrow \updownarrow \updownarrow$   
 a b a

$p_0 = t_0$   
 $p_1 = t_1$   
 $p_2 \neq t_2$

$p_0 \neq p_1$   
 $p_0 \neq t_1$

$p_0 = p_2$   
 $p_0 \neq t_2$

a b b a b a  
 $\updownarrow \updownarrow \nearrow$   
 a b a

a b b a b a  
 $\updownarrow \updownarrow \updownarrow$   
 a b a



j不后退，通过前面已经做的比较，直接确定下一次j（或j+1）与p中的那个字符对齐比较。

此时消除了回溯问题，提高了模式匹配的时间效率。  
 上面只是个例分析，到底如何消除回溯操作？

—————> **KMP无回溯模式匹配方法**



### 3.3.2 无回溯的模式匹配（KMP算法）

#### 1. 基本思想：

要找到一个无回溯的模式匹配算法，关键在于当匹配过程中，一旦 $t_j$ 与 $p_i$ 比较不等，即：

$t_0 t_1 t_2 \dots t_{j-i+1} \dots t_{j-1} \textcolor{red}{t_j} \dots$   
 $p_0 \dots p_{i-1} \textcolor{red}{p_i} \dots$

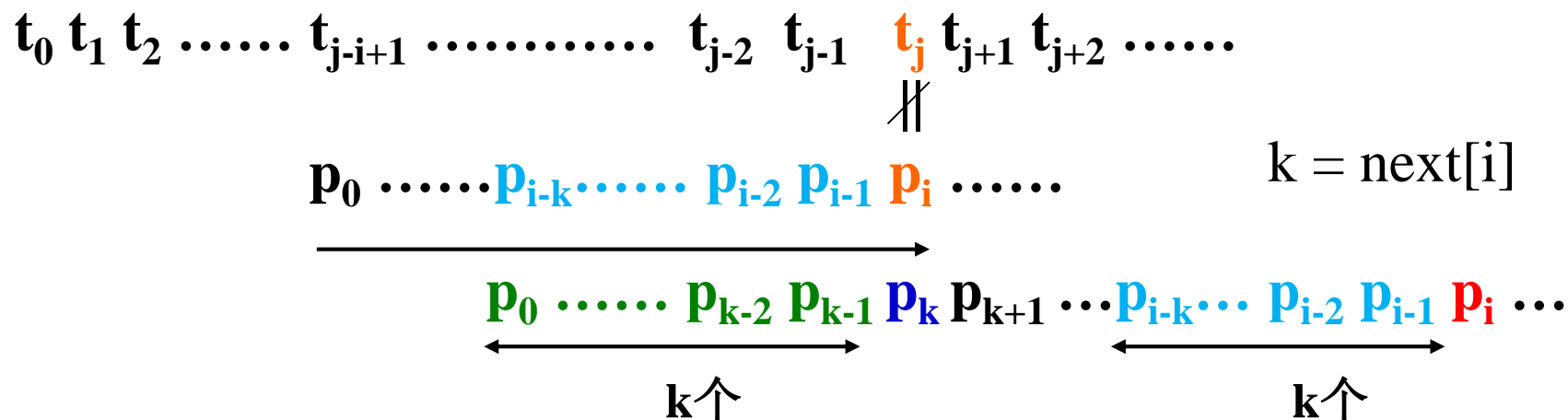
$t_0 t_1 t_2 \dots t_{j-i+1} \dots t_{j-1} \textcolor{red}{t_j} \dots$   
 $p_0 \dots \textcolor{green}{p_k} \dots p_{i-1} \textcolor{red}{p_i} \dots$

$$\begin{cases} t[j-i+1, i-1] = p[0, i-1] & \text{【前面}i\text{个字符相同】} \\ t_j \neq p_i \end{cases}$$

要能立即确定 $p$ 右移的位数和继续（无回溯）比较的字符，也就是说应该用 $p$ 中的哪个字符和 $t_j$ 进行比较？把这个字符记为 $p_k$ ，显然有  $k < i$ ，并且对于不同的 $i$ ， $k$ 值也不同。

D.E.Knuth、J.H.Morris 和 V.R.Pratt 同时发现：这个 $k$ 值仅依赖于模式 $p$ 本身前 $i$ 个字符的组成，而与目标 $t$ 无关。用 $\text{next}[i]$ 表示与 $i$ 对应的 $k$ 值，其意义在于：

- 若 $\text{next}[i] \geq 0$ ，表示一旦匹配过程中 $p_i$ 与 $t_j$ 比较不等，可用 $p$ 中以 $\text{next}[i]$ 为下标的字符与 $t_j$ 进行比较。



- 若 $\text{next}[i] = -1$ ，则表示 $p$ 中任何字符都不必再与 $t_j$ 进行比较，下次比较从 $t_{j+1}$ 与 $p_0$ 开始。

Text string  $t$ :  $t_0 t_1 t_2 \dots t_{j-1} t_j t_{j+1} \dots$

Pattern string  $p$ :  $p_0 \dots p_{i-1} p_i \dots$

对于任意模式 $p$ ，只要确定 $\text{next}[i](i=0, 1, \dots, m-1)$ 的值，就可以加速匹配过程，避免回溯问题。当 $t_j \neq p_i$ 时，直接右移模式串 $i - \text{next}[i]$ 个字符，并从 $t_j$ (或 $t_{j+1}$ )继续下去。

## 2. 无回溯的模式匹配算法:

已知next数组（该数组的计算方法在下面讨论），无回溯的模式匹配算法如下：

```
int pMatch( SeqString t, SeqString p, int *next )
{   int i, j;
    i = j = 0;                                //初始化
    while ( i < p.n && j < t.n )              //反复比较
    {
        if (i == -1 || p.c[i] == t.c[j])
        {   i++; j++; }                        //继续匹配下一个字符
        else   i = next[i];                   //j不变, i后退
    }
    //匹配成功, 返回p第一个字符在t中的位置（下标）
    if ( i >= p.n) return (j-p.n)
    else          return -1;                  //匹配失败
}
```

### 3. 匹配算法的时间效率分析:

由于 $i$ 变量的增减次数不定，循环次数不好估计。

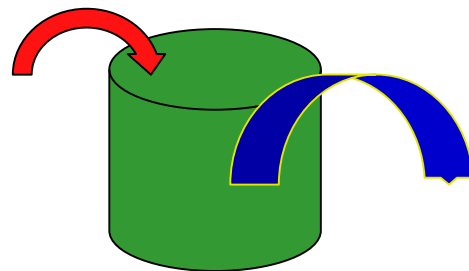
$i$ 增加的次数：上述算法中 $j$ 值只增不减，由于 $j$ 的初值为0，循环过程中保持 $j < n$ ，所以循环体中 $j++$ 语句的执行次数不超过 $n$ ，从而 $i++$ 的执行次数也不超过 $n$ 。

$i$ 减少的次数（退）： $i$ 的初始值为0，唯一使 $i$ 减少的语句是 $i = \text{next}[i]$ ，由于 $\text{next}[i]$ 在 $[-1, i)$ 范围内， $i = \text{next}[i]$ 的执行次数也必定不超过 $i++$ 的执行次数。（如果超过，会出现什么状况？）

【“退”的快（每次至少1），“加”的慢。既然“加”的次数不超过 $n$ ，“退”的次数也不会超过 $n$ 】

综合上述情况，在已经计算出 $\text{next}$ 的前提下，算法的时间效率为 $O(n)$ 。

$\text{next}[i]$ 如何计算？

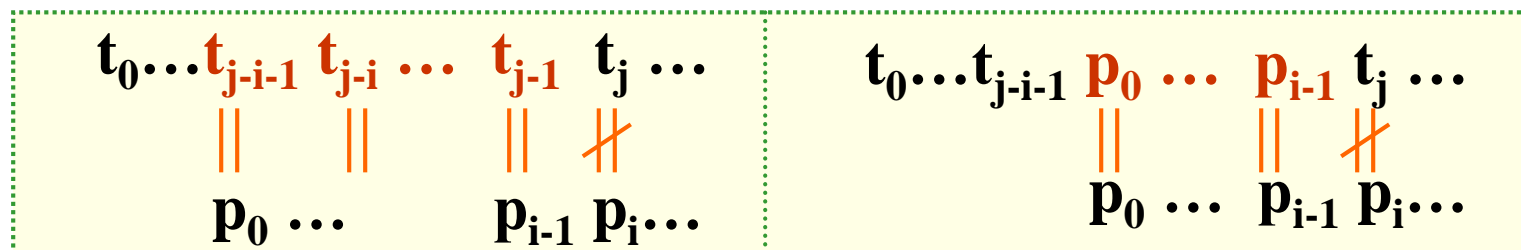


## 4. Next数组计算

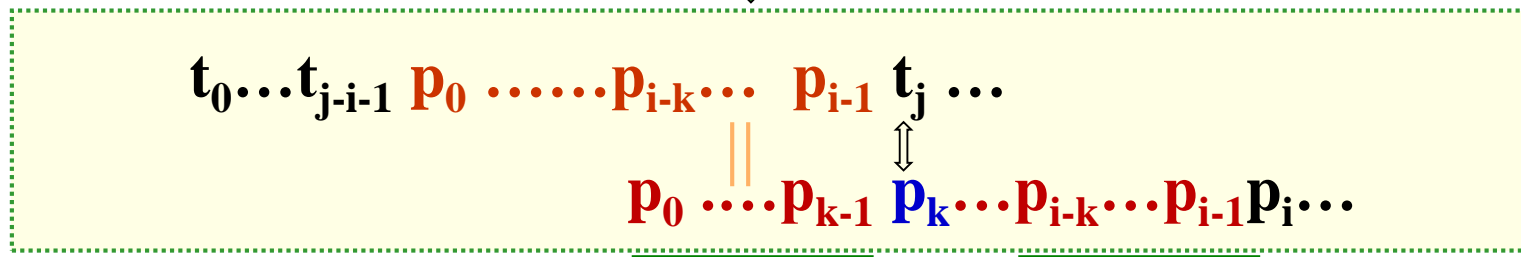
对于任意的模式串  $p=p_0p_1\cdots p_{m-1}$ , 存在一个由模式串本身唯一确定的与目标串无关的数组  $next$ 。

$next$ 数组的计算方法。

在  $p$  与任意的目标串  $t$  匹配时, 若发现  $t_j \neq p_i$ , 则意味着  $p_0$ 、 $p_1$ 、 $\dots$ 、 $p_{i-1}$  已经与  $t$  中对应的字符进行过比较, 而且是相等的, 否则轮不到  $t_j$  与  $p_i$  的比较, 因此下面左右两个图是等价的。

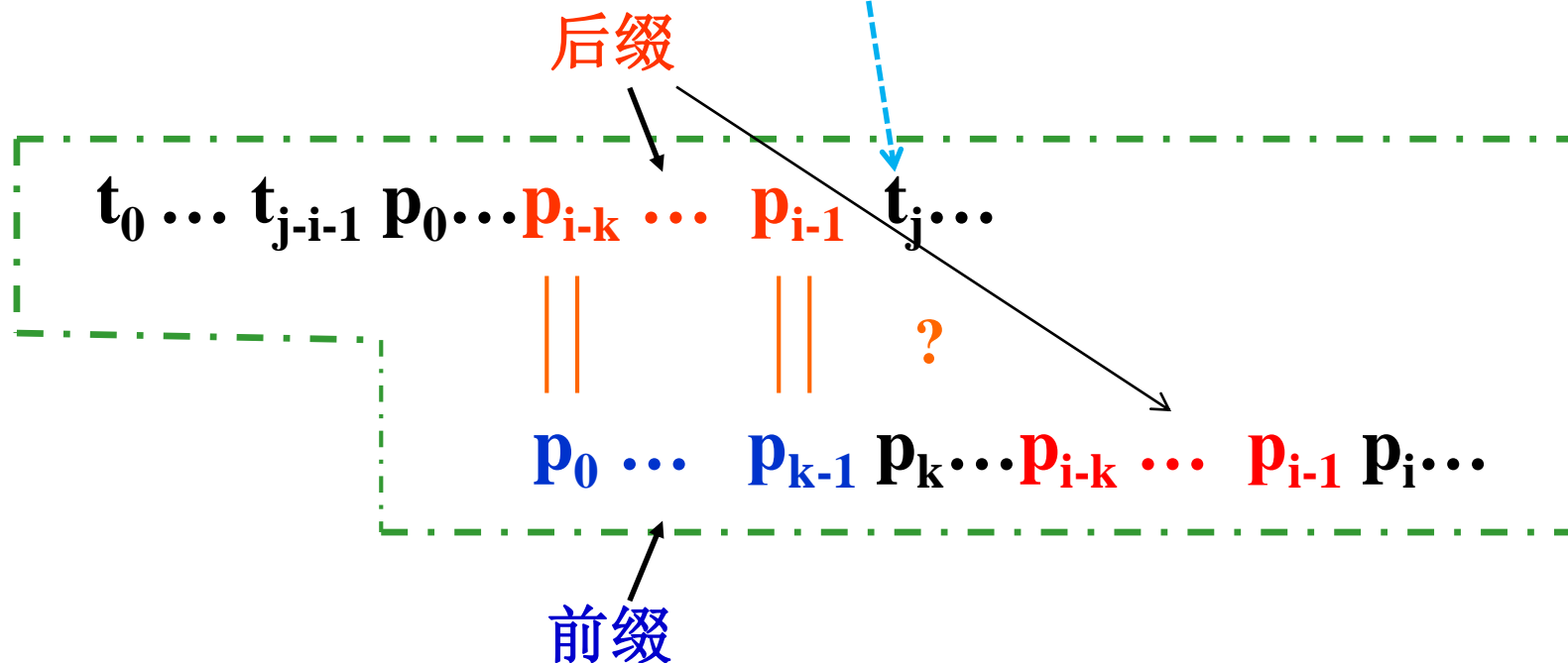


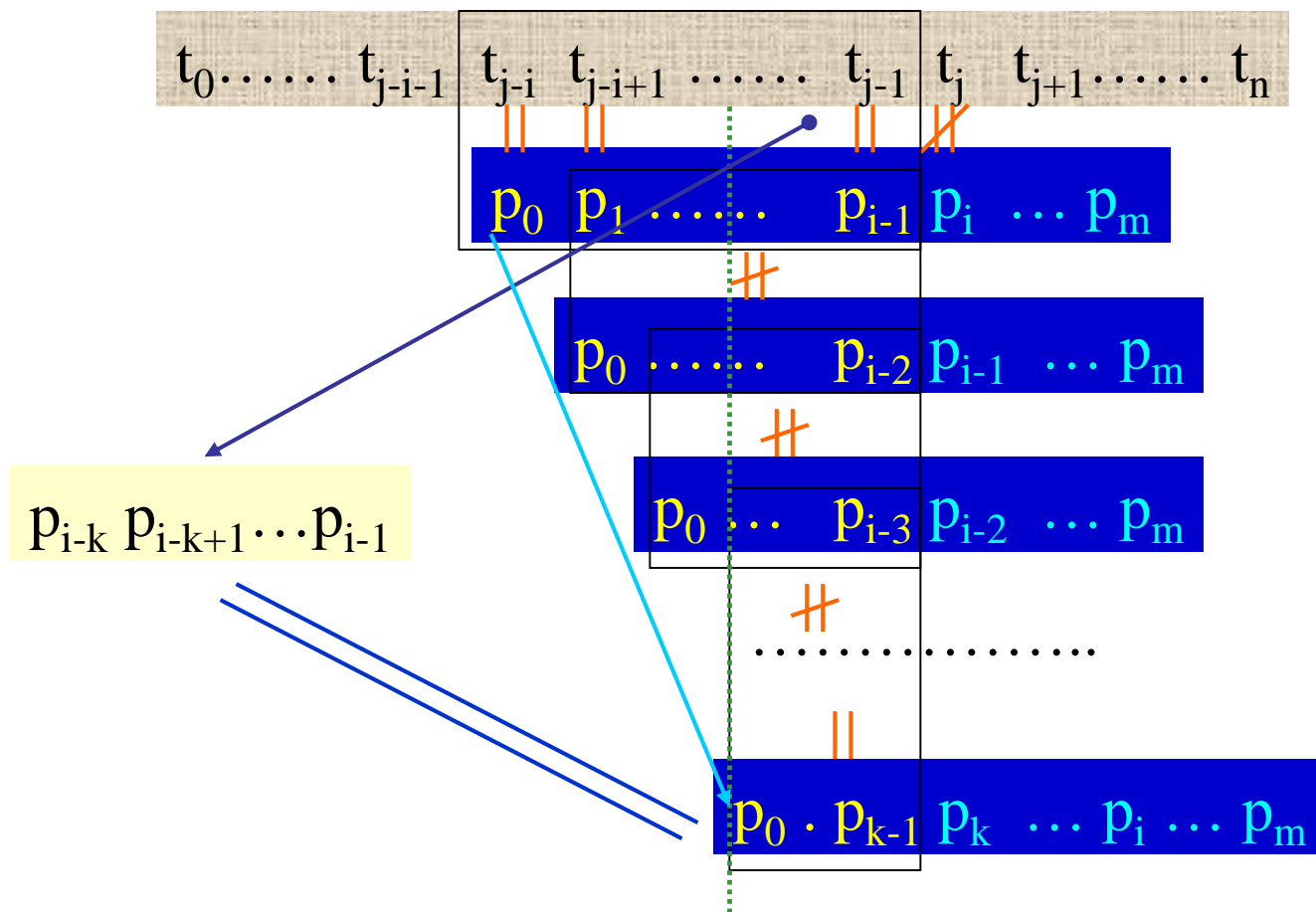
下一步,  $j$  不回溯, 与模式那个字符继续比较?  
即模式往右移动多少? (假定  $p_k$  与  $t_j$  对齐比较)  
如果存在这样的  $k$ , 意味着什么?



如**k**存在:  $p$ 右移若干位至 $p_k$ ,  $t_j$ 以前的工作相当于用 $p_0 \dots p_{i-1}$ 的一个前缀( $p_0 \dots p_{k-1}$ )与它的一个长度相同的后缀( $p_{i-k} \dots p_{i-1}$ )进行比较。显然比较的结果由 $p$ 本身和 $p_i$ 决定, 而与 $t$ 无关。

如**k**不存在: 下一步只能是 $t_{j+1}$ 与 $p_0$ 对齐继续下一趟匹配。





如相等：

$\text{next}[i] = i-1$

$\text{next}[i] = i-2$

.....

$\text{next}[i] = k$

$\text{next}[i]$

这样，可以在 $p_0 \dots p_{i-1}$ 中求出相同并且最大的前缀和后缀的长度 $k(0 \leq k < i-1)$ （不包括 $p_0 \dots p_{i-1}$ 本身，但允许空串）。

当模式串 $p$ 的右移量为 $i-k$ 时，相同的前后缀恰好对齐， $p_k$ 与 $t_j$ 也

- 用不着去比较这一对前后缀，因为已经知道它们是相等的；
- 当右移量少于 $i-k$ 时，也用不着比较，因为此时出现长度大于 $k$ 的前后缀，与前面的最大前后缀矛盾。因此，二者比较肯定不等。

$$\begin{array}{ccccccc} \mathbf{t}_0 & \mathbf{t}_1 & \dots & \dots & \dots\dots\dots & \dots & \dots & \mathbf{t}_j & \mathbf{t}_{j+1} & \mathbf{t}_{j+2}\dots\dots\dots \\ & & & & & & & \mathbf{p}_0 & \mathbf{p}_1\dots\dots\mathbf{p}_{k-1} & \mathbf{p}_k & \dots & \mathbf{p}_v\dots\dots\dots\mathbf{p}_{i-1} & \mathbf{p}_i & \dots \end{array}$$

- 当右移量大于 $i-k$ 时，可能漏掉匹配成功的机会。

由此可见，当发现 $p_i \neq t_j$ 时，即可直接把 $p$ 右移 $i-k$ 位，并且只从 $p_k$ 和 $t_j$ 开始向右比较。这样做并没有丢掉任何匹配成功的机会。



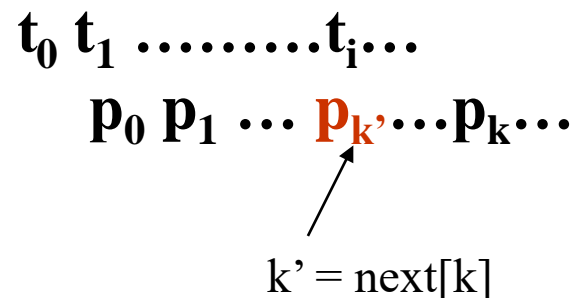
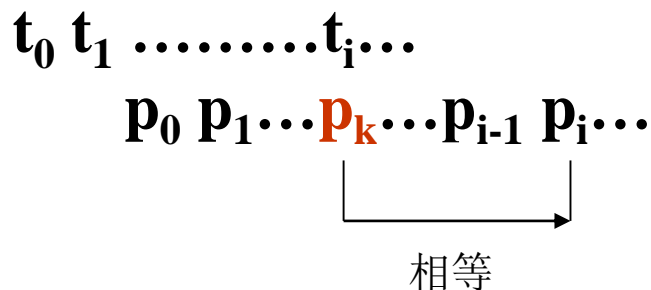
通过上面分析，得到了初步的next计算方法：

- (1) 求 $p_0 \dots p_{i-1}$ 中最大相同的前缀和后缀的长度 $k$ ;
- (2)  $\text{next}[i] = k$ ;

作为特殊情况，当 $i=0$ 时，令 $\text{next}[i] = -1$ ;  
显然，对于任意 $i(0 \leq i < m)$ ，有 $\text{next}[i] < i$ ;

按照上述方法求得的next数组已经可以用于无回溯模式匹配中，但还可进一步改进。

若求得 $k$ 后，且 $p_k = p_i$ ，则当 $p_i \neq t_j$ 时， $p_k$ 与 $t_j$ 的比较必然不等，应该继续右移 $p$ ，用 $p_{\text{next}[k]}$ 与 $t_j$ 比较。



修改next计算中的步骤(2)：

```
if ( $p_k \neq p_i$ ) next[i] = k;  
else      next[i] = next[k];  
          ( $\text{next}[k]$ 肯定小于 $\text{next}[i]$ )
```

如此求出的next数组中，不仅 $\text{next}[0]=-1$ ，还可能由其它 $\text{next}[i] < 0$ 。如果存在 $\text{next}[i]=-1$ ，表示当发现 $p_i \neq t_j$ 时，则右移 $i+1$ 位，用 $p_0$ 与 $t_{j+1}$ 开始比较。  
最大前后缀计算：

$p_0, p_1, \dots, p_{i-4} p_{i-3} p_{i-2}$	$\Leftrightarrow$	$p_1, p_2, p_3, \dots, p_{i-1}$	[长度 $i-1$ ]
$p_0, p_1, \dots, p_{i-4} p_{i-3}$	$\Leftrightarrow$	$p_2, p_3, \dots, p_{i-1}$	[长度 $i-2$ ]
$p_0, p_1, \dots, p_{i-4}$	$\Leftrightarrow$	$p_3, \dots, p_{i-1}$	[长度 $i-3$ ]
.....			
$p_0, p_1, \dots, p_{k-1}, p_k$	$\Leftrightarrow$	$p_{i-k-1}, p_{i-k}, \dots, p_{i-1}$	[长度 $k+1$ ]
$p_0, p_1, \dots, p_{k-1}$	$\Leftrightarrow$	$p_{i-k}, p_{i-k+1}, \dots, p_{i-1}$	[长度 $k$ ]

对于任意 $p$ ,  $\text{next}[0] = -1$ 。表示：如果 $p_0 \neq t_j$ , 下一趟只能用 $p_0$ 与 $t_{j+1}$ 进行比较。

对于每个 $i$ , 采用前面的由长到小一个个比较实现, 非常麻烦!  
可以巧妙地通过下面的递推方法实现模式整体 $\text{next}$ 数组计算。

next计算过程: 初始 $\text{next}[0] = -1$ , 然后计算 $\text{next}[1], \text{next}[2], \dots, \text{next}[i], \text{next}[i+1], \dots, \text{next}[m-1]$ .

假定已经计算得到 $\text{next}[i]$ , 那么如何计算 $\text{next}[i+1] = ?$

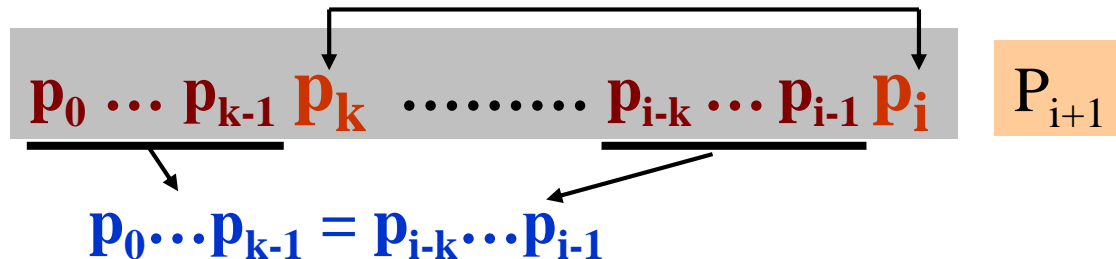
直观地理解: 已知 $\text{next}[i]=k$ , 则 $\text{next}[i+1]$ 就是在

$\underline{p_0, \dots, p_{k-1}}, \textcolor{red}{p_k}, p_{k+1}, \dots, \underline{p_{i-k}, \dots, p_{i-1}}, p_i \text{ 【} p_{i+1} \text{】}$   
-----

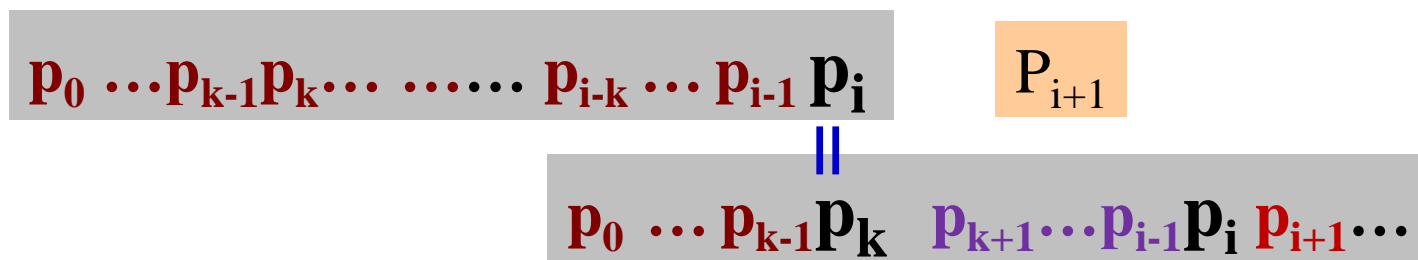
中找最大前后缀。显然,  $\text{next}[i+1]$ 不超过 $k+1$ 。

下面, 讨论 $\text{next}[i+1]$ 的计算过程。

next[i+1] 的计算:

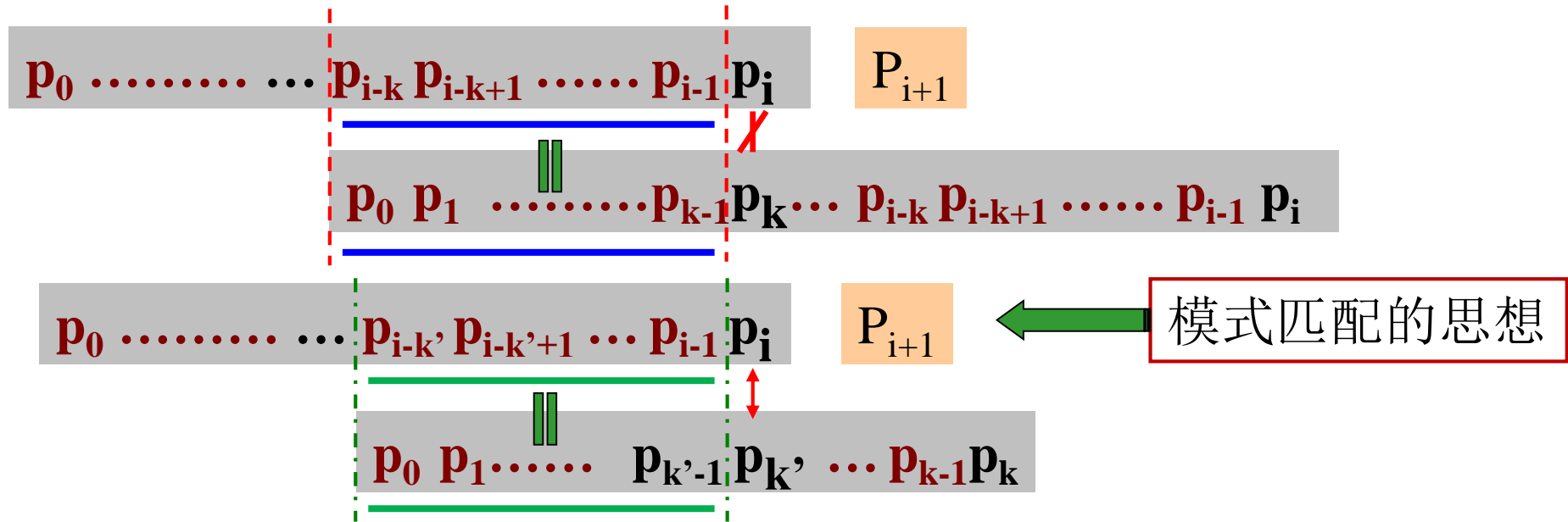


(1) 如果 $p_k = p_i$ , 则有:  $p_0 \dots p_{k-1} p_k = p_{i-k} \dots p_{i-1} p_i$   
 $\text{next}[i+1] = k+1 = \text{next}[i]+1$



(2) 如果 $p_k \neq p_i$ , 与模式匹配的思路一样, 应当将模式串往右滑动到模式串的第 $k' = \text{next}[k]$ 个字符与 $p_i$ 进行比较。此时, 模式串本身既是目标又是模式。

由长到短, 寻找 $k'$ 。满足:  $p_0 \dots p_{k'-1} p_{k'} = p_{i-k'} \dots p_{i-1} p_i$



假定  $k' = \text{next}[k]$ , 说明模式中  $k'$  字符前存在一个长度为  $k'$  ( $\text{next}[k]$ ) 的最长子串, 满足:  $p_0 \dots p_{k'-1} = p_{i-k'} \dots p_{i-1}$

➤ 若  $p_{k'} = p_i$ , 则有:  $\text{next}[i+1] = k' + 1 = \text{next}[k] + 1$

➤ 若  $p_{k'} \neq p_i$ , 则将模式串继续往右滑动直至将模式中第  $\text{next}[k']$  个字符与  $p_i$  对齐, 依次类推, 直至模式中某个 **匹配成功** 或不存在任何  $k'$  满足:  $p_0 \dots p_{k'-1} = p_{i-k'} \dots p_{i-1}$ , 此时  $\text{next}[i+1] = 0$

## next[i+1]的计算

next[0] = -1;

next[1] = .....

.....

next[i] = .....

next[i+1] =  $p_{0...i}$ 中最大前后缀

**k = next[i];**

**while (k >= 0)**

**{**

**if ( $p_k == p_i$ ) { break; }**

**k=next[k];**

**}**

**next[i+1] = k+1;**

**【 再加一次修正 】**

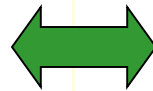
**【为什么只要1次即可? 】**（前面的都是已经修正的。）

.....

next[m-1]=.....

## 计算next数组的算法:

```
makeNext1( SeqString p, int *next )
{
    int i = 0, k = -1;
    next[0] = -1;
    while ( i < p.n-1 ) //计算next[i+1]
    {
        while (k >= 0 && p.c[i] != p.c[k])
            k = next[k]; //求最大前后缀
        i++; k++;
        if (p.c[i] != p.c[k])
            next[i] = k;
        else next[i] = next[k];
    }
}
```



//与前面的KMP算法类似

```
makeNext2(SeqString p, int *next)
{
    int i = 0, k = -1;
    next[0] = -1;
    while ( i < p.n-1 ) //计算next[i+1]
    {
        if ( k == -1 || p.c[i] == p.c[k])
        { i++; k++;
          if ( p.c[i] != p.c[k])
              next[i] = k;
          else next[i] = next[k];
        }
        else k = next[k];
    }
}
```

计算next数组的时间复杂度为 $O(m)$  [ $m$ 为模式串长度]，这样KMP无回溯模式匹配的时间复杂度为： $O(m+n)$ 。

## 5. 示例

**t = “aabcbabcaabcaababc” , n = 18**

**s = “abcaababc”** , **m = 9**

## 计算next数组

下标i	0	1	2	3	4	5	6	7	8
$p_i$	a	b	c	a	a	b	a	b	c
$P_0 \dots p_{i-1}$ 中最大相同的前后缀长度 k		0	0	0	1	1	2	1	2
$P_k$ 与 $p_i$ 比较		$\neq$	$\neq$	$=$	$\neq$	$=$	$\neq$	$=$	$=$
next[i]	-1	0	0	-1	1	0	2	0	0

## 利用next数组进行与目标t的匹配

序号i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
-----	---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----

目标t    a a b c b a b c a a b c a a b a b c

模式p    **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**

**b: next[1] = 0**

**a: next[3] = -1**

**a: next[6] = 2**

# 成功



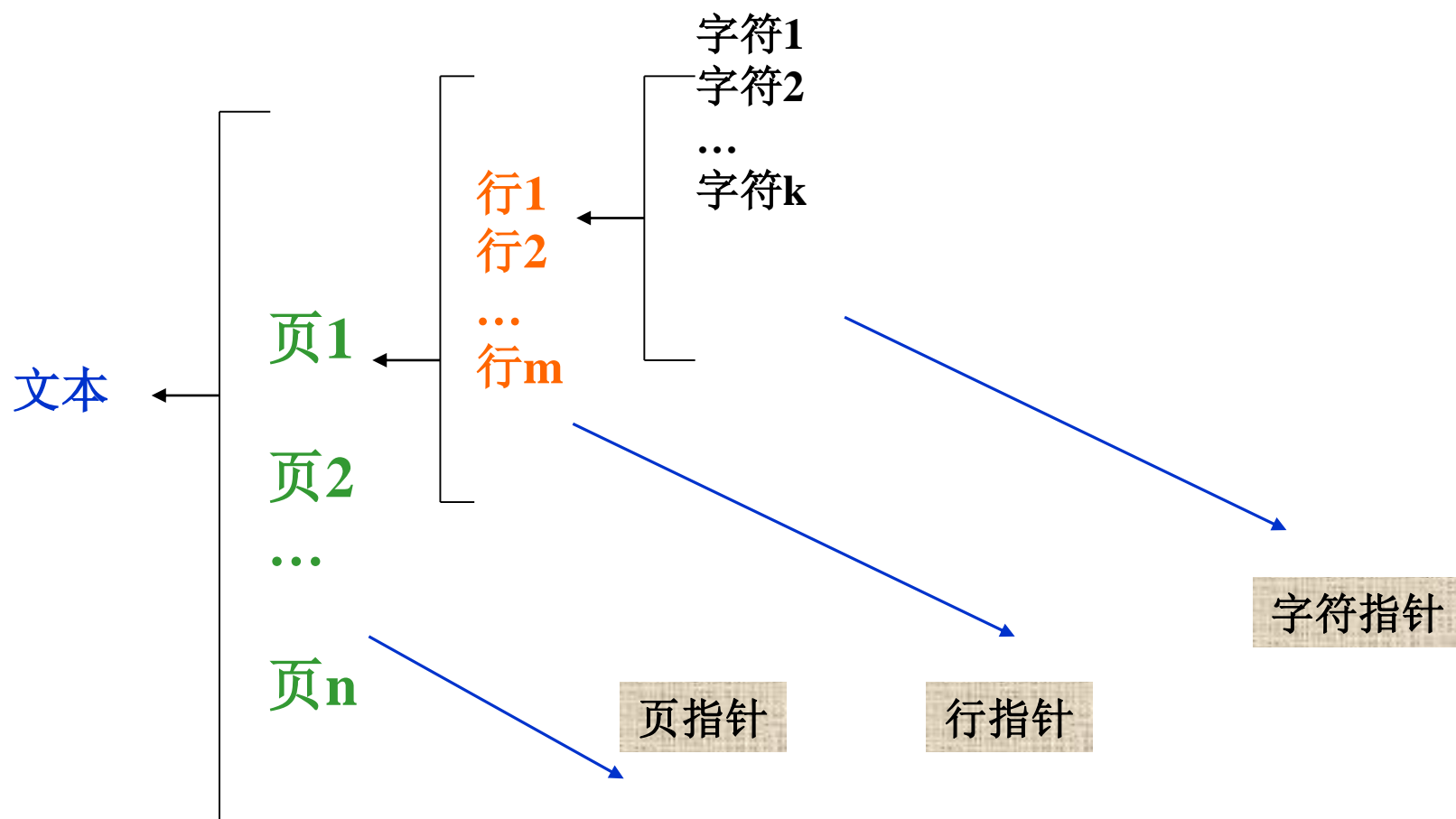
## 3.4 串操作应用

### 1. 文本编辑程序

文本编辑程序实质上就是一个字符串的操作应用。可以把整个文本看成是一个字符串，这样文本的插入、删除、查找、替换等就是字符串的各种操作的实现。

文本编辑程序通常设立页索引表、行索引表进行管理，并且附设页指针、行指针和字符指针指示当前正在编辑的页、行和字符。


例如：插入、删除行，首先需要通过页索引表确定插入、删除操作的页，然后对该页索引表进行操作。如果是插入、删除字符，则除了确定页之外，还需要通过行索引表确定操作的行，然后通过字符指针完成插入、删除字符的操作，当然此时还需要修改行的长度等。



## 2. 词索引表建立

图书检索中，通常要建立“关键词-书号索引”：

书号	书名	关键词	书号索引
005	Computer Data Structures	algorithms	034
010	Introduction to Data Structures	analysis	034 050 067
023	Fundamental of Data Structures	computer	005 034
034	The Design and Analysis of Computer Algorithms	data	005 010 023
050	Introduction to Numerical Analysis	design	034
067	Numerical Analysis	fundamentals	023
		introduction	010 050
		numerical	050 067
		structures	005 010 023



### 步骤：

- (1) 输入一本书名，提取关键词（需要与常用词表对比）
- (2) 在索引表中查找该关键词是否存在。如不存在，则加入该关键词（按照字典有序原则进行，并建立书号索引；如存在，这在该关键词项中加入书号索引。
- (3) 继续输入下一本书名，重复上述操作。

## 数据结构:

(1) **词表**: 存放书名中的各个关键词, 数量有限, 采用顺序结构存储, 每个词是一个字符串;

(2) **索引表**: 虽然动态生成, 并且需要频繁插入, 但考虑到该表主要用于查询, 因此采用顺序存储结构。表中每个索引项包含两方面内容: 关键词和书号索引。书号索引由于是动态生成的, 且数目不同, 宜采用链表结构存储。

```
struct WordList
```

```
{  char *item[]; //字符串数组
   int last;     //词表长度
};
```

```
struct LinkList
```

```
{  int No;        //书号
   struct LinkList *next; //下一本书
}
```

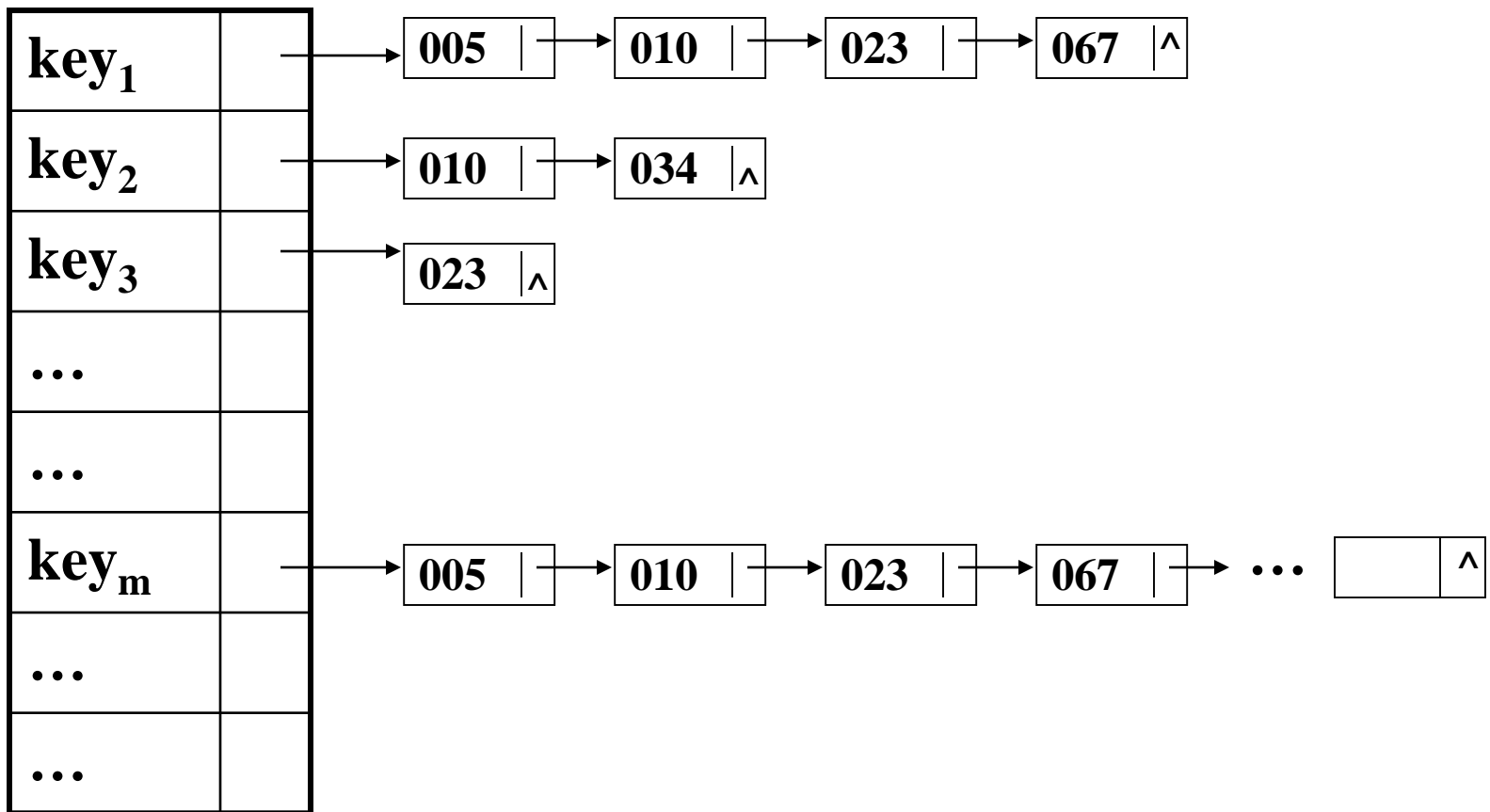
```
typedef struct LinkList *PLinkList;
```

```
struct KeyItem
```

```
{  char Key[64];      //关键词
   PLinkList booklist; //书号索引表
};
```

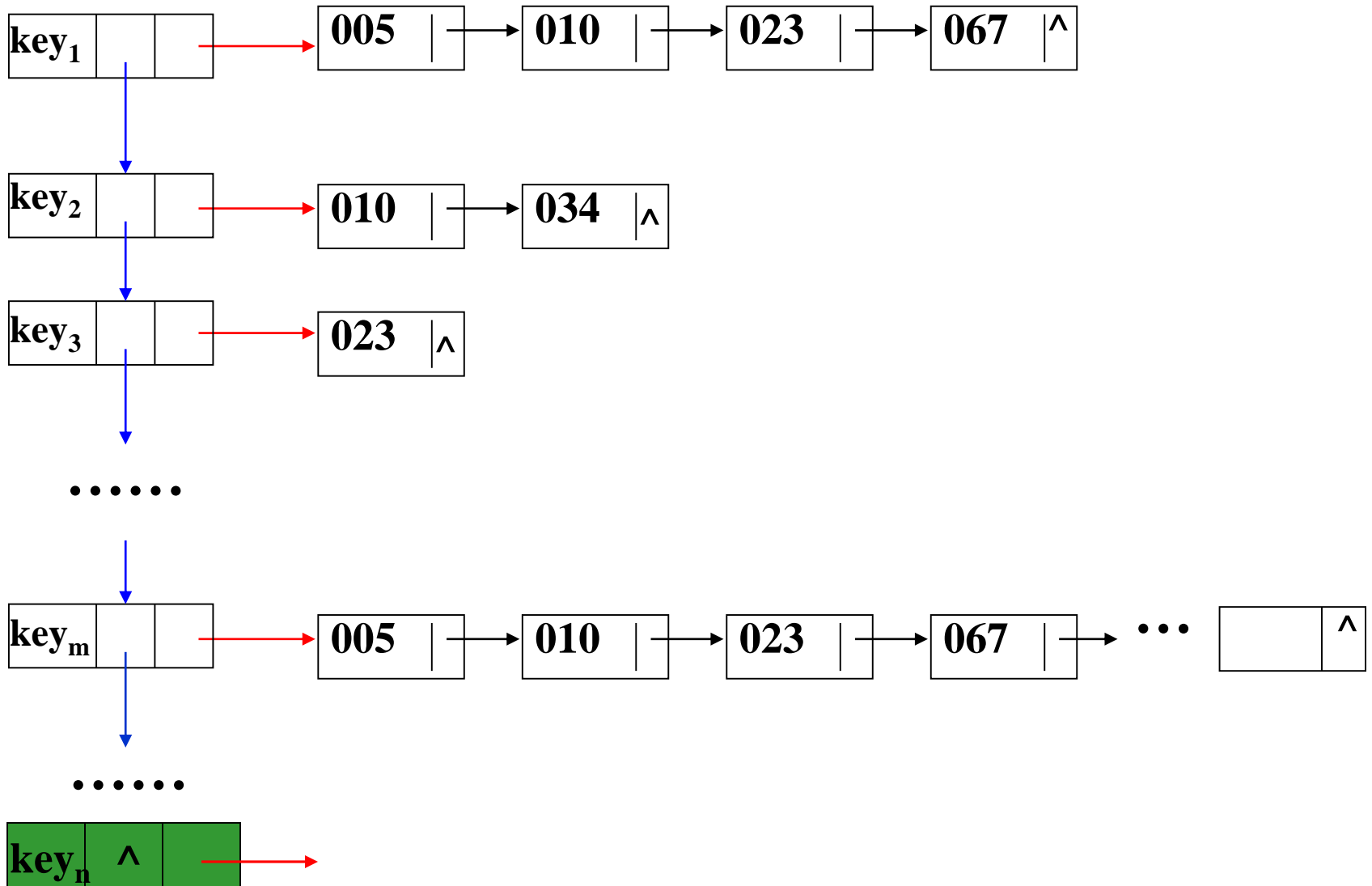
```
struct IndexList
```

```
{  struct KeyItem item[MAXNUM_KEY];
   int last;                //关键词数目
}
```



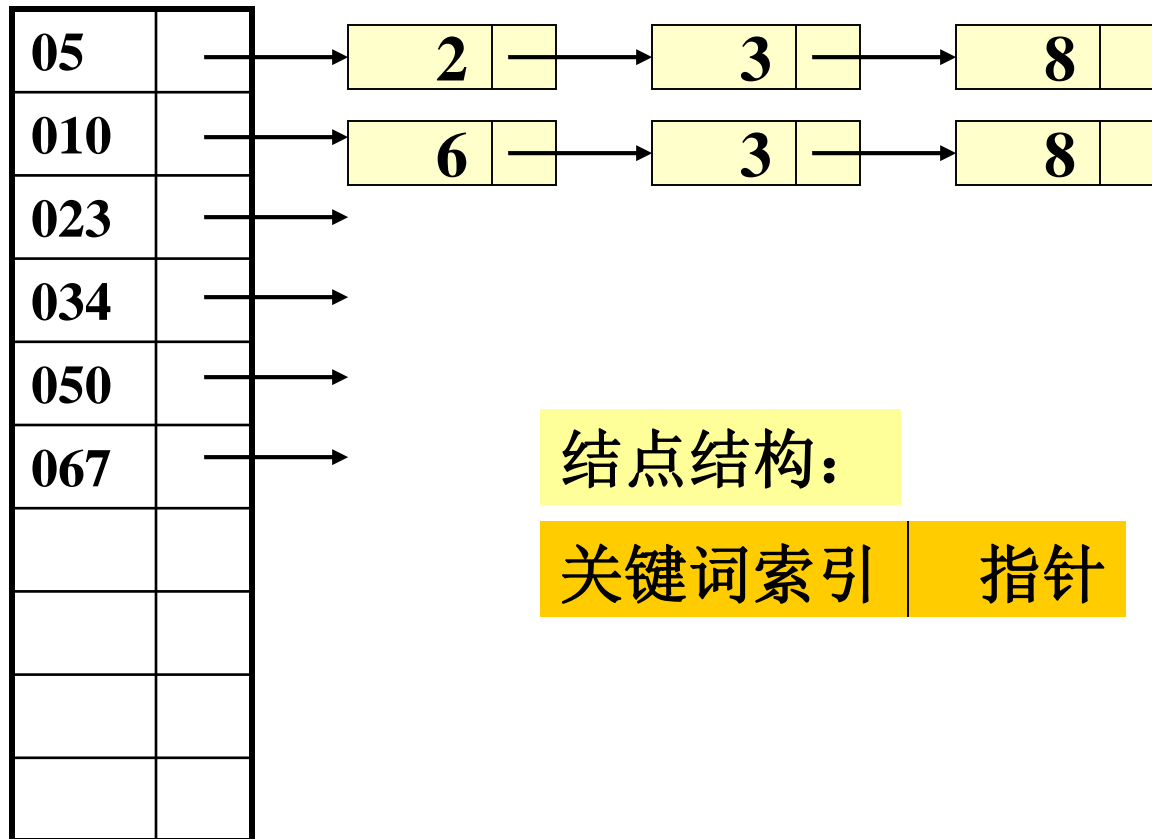
构造算法.....

# 词索引表采用链表存储



**大家再进一步思考：**前面的结构非常方便根据“关键字”查“书号”，如果要给定“书号”查“关键字”如何操作？

如果可以改变结构，如何改变？换成下面的结构（书号关键词索引表）如何？



结点结构：

关键词索引

指针

关键词表

algorithms  
analysis  
computer  
data  
design  
fundamentals  
introduction  
numerical  
structures

/\* STRTOK.C: In this program, a loop uses strtok to print all the tokens (separated by commas or blanks) in the string \*/

```
#include <string.h>
```

```
#include <stdio.h>
```

```
char string[] = "A string\tof ,,tokens\nand some more tokens";
```

```
char seps[] = " ,\t\n", *token;
```

```
void main( void )
```

```
{ printf( "%s\n\nTokens:\n", string );
```

```
/* Establish string and get the first token: */
```

```
token = strtok( string, seps );
```

```
while( token != NULL )
```

```
{ /* While there are tokens in "string" */
```

```
printf( " %s\n", token ); /* Get next token: */
```

```
token = strtok( NULL, seps );
```

```
}
```

```
}
```



## **Output**

**A string of „tokens  
and some more tokens**

**Tokens:**

**A  
String  
of  
Tokens  
and  
some  
more  
tokens**

# 小结

本章主要讨论了字符串的概念、存储表示以及基本运算的实现。字符串是一种线性结构，它是零或多个字符组成的有限序列。常用存储方式：**顺序、链接存储结构**；

应该**重点掌握**字符串在顺序存储结构和链接存储结构中基本操作的实现。

另外，**模式匹配**是一个比较复杂的操作。**朴素模式匹配**算法简单，但由于回溯操作使得算法效率低[ $O(n*m)$ ]。改进方法是**KMP无回溯模式匹配**方法，该方法中首先根据模式本身计算得到next数组[ $O(m)$ ]，然后通过使用next数组避免匹配过程中的回溯操作，从而提高了匹配速度，减少了时间复杂度。KMP算法的时间效率为 $O(m+n)$ ，但空间效率比朴素模式匹配方法降低。

# 上机作业

- (1) 实现KMP无回溯模式匹配算法（上机调试完成）
- (2) 实现关键词索引表建立程序（上机调试完成）

思考：某一数据文件，由于输入错误，数值间既有空格又有逗号，如何读取？

例如：

```
122  100 300, 200 800 400, 200
998, 234 128  176, 111, 555 666
988  777 222  456, 789, 124  333
```

# 字符串字符大小写转换

#include <string.h>

## **\_strupr(string)**

将string中的字符转换为大写，如“abcAB123”转换后：  
“ABCAB123”

## **\_strlwr(string)**

将string中的字符转换为小写，如“abcAB123”转换后：  
“abcab123”

例如：判断两个文件名是否相同

```
if (strcmp(_strupr(filename1), _strupr(filename2)) == 0)
{
    printf(“相同文件\n”);
}
```

或

```
if (strcmp(_strlwr(filename1), _strlwr(filename2)) == 0)
{
    printf(“相同文件\n”);
}
```