



信息与软件工程学院

# 程序设计与算法基础II

主讲教师：陈安龙

# 第4章 串

- 串的基本概念
- 串的存储实现
- 定长顺序串
- 堆串
- 块链串
- **Brute-Force**模式匹配算法
- **KMP**模式匹配算法

## 4.1 串的基本概念

串（或字符串）是由零个或多个字符组成的有限序列。

串  $\subset$  线性表

串中所含字符的个数称为该串的长度，含0个字符的串称为空串，用 $\Phi$ 表示。

串的逻辑表示， $a_i$  ( $1 \leq i \leq n$ ) 代表一个字符：

“ $a_1 a_2 \dots a_n$ ”



双引号不是串的内容，起标识作用

**串相等**：当且仅当两个串的长度相等并且各个对应位置上的字符都相同时，这两个串才是相等的。

如：

*“abcd”*  $\neq$  *“abc”*

*“abcd”*  $\neq$  *“abcde”*

所有空串是相等的。

**子串**：一个串中任意个连续字符组成的子序列（含空串）称为该串的子串。

例如，“*abcde*”的子串有：

“”、“*a*”、“*ab*”、“*abc*”、“*abcd*”和“*abcde*”等

**真子串**是指不包含自身的所有子串。

## 串抽象数据类型=逻辑结构+基本运算（运算描述）

- (1) StrAssign(S,chars):** 初始条件:chars是字符串常量  
操作结果:生成一个值等于chars的串S
- (2) StrInsert(S,pos,T):** 初始条件:串S和T存在, $1 \leq \text{pos} \leq \text{StrLength}(S) + 1$   
操作结果:在串S的第pos个字符之前插入串T
- (3) StrDelete(S,pos,len):** 初始条件: 串S存在, $1 \leq \text{pos} \leq \text{StrLength}(S) - \text{len} + 1$   
操作结果: 从串S中删除第pos个字符起长度为len的子串
- (4) StrCopy(S,T):** 初始条件: 串S存在  
操作结果:由串T复制得串S

(5) **StrEmpty(S)** : 始条件: 串**S**存在

操作结果:若串**S**为空串,则返回**TRUE**,否则返回**FALSE**

(6) **StrCompare(S,T)**: 初始条件: 串**S**和**T**存在

操作结果:若**S>T**,则返回值**>0**;若**S=T**,则返回值**=0**;若**S<T**, 则返回值**<0**

(7) **StrLength(S)**: 初始条件: 串**S**存在

操作结果:返回串**S**的长度,即串**S**中的元素个数

(8) **StrClear(S)**: 初始条件: 串**S**存在

操作结果:将**S**清为空串

(9) **StrCat(S,T)**: 初始条件: 串**S**和**T**存在

操作结果:将串**T**的值连接在串**S**的后面

### (10) SubString(Sub,S,pos,len)

初始条件: 串S存在,  $1 \leq \text{pos} \leq \text{StrLength}(S)$  且  $1 \leq \text{len} \leq \text{StrLength}(S) - \text{pos} + 1$

操作结果: 用Sub返回串S的第pos个字符起长度为len的子串

### (11) StrIndex(S,T,pos) : 初始条件: 串S和T存在,T是非空串, $1 \leq \text{pos} \leq \text{StrLength}(S)$

操作结果: 若串S中存在与串T相同的子串, 则返回它在串S中第pos个字符之后第1次出现的位置; 否则返回0

### (12) StrReplace(S,T,V): 初始条件: 串S,T和V存在, 且T是非空串

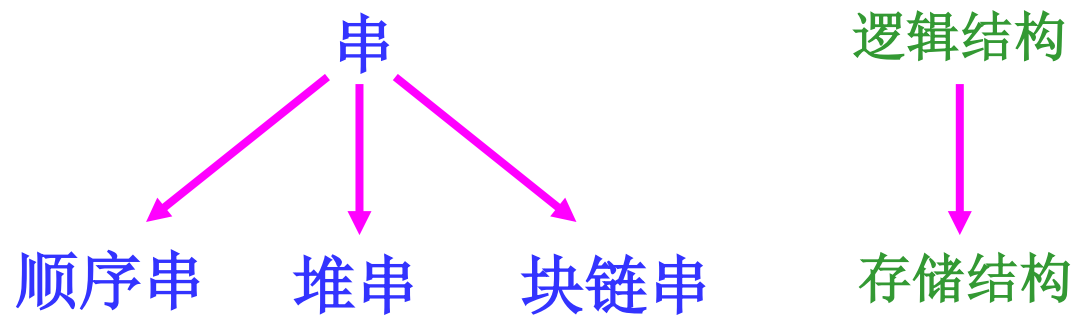
操作结果: 用V替换串S中出现的所有与T相等的不重叠子串

### (13) StrDestroy(S): 初始条件: 串S存在; 操作结果: 销毁串S



## 4.2 串的存储结构

串中元素逻辑关系与线性表的相同，串可以采用与线性表类似的存储结构。



## 4.2.1 串的顺序存储及其基本操作实现

串的顺序存储（顺序串）有两种方法：

- 每个单元(如4个字节)只存一个字符，称为**非紧缩格式**（其存储密度小）。
- 每个单元存放多个字符，称为**紧缩格式**（其存储密度大）。

1001	A			
1002	B			
1003	C			
1004	D			
1005	E			
1006	F			
1007	G			
1008	H			
1009	I			
100a	J			
100b	K			
100c	L			
100d	M			
100e	N			

非紧缩格式示例

1001	A	B	C	D
1002	E	F	G	H
1003	I	J	K	L
1004	M	N		

紧缩格式示例

一个单元

对于非紧缩格式的顺序串，其类型定义如下：

```
#define MAXLEN 100
```

```
typedef struct
```

```
{    char ch[MAXLEN];
```

```
    int len;
```

```
} SeqString;
```

用来存储字符串长度

用来存储字符串

## 【算法4-1】：顺序串的插入算法

请大家完成：如果 $s \rightarrow \text{len} + t \rightarrow \text{len} \leq \text{MAXLEN}$ ，插入后需要保存s和t的所有字符，如何改写算法？

```
int StrInsert(SeqString *s,int pos, SeqString t)
{ //在字符串s的下标为pos位置插入字符串t
  if(pos<=0 || pos>s->len) return 0;
  if(s->len+t.len<=MAXLEN) // 插入后长度<=MAXLEN， t串和s串的字符都能保存
  { for (i=s->len+t.len-1; i>=t.len+pos;i--) s->ch[i]=s->ch[i-t.len];
    for (i=0; i<t.len; i++) s->ch[i+pos]=t->ch[i];
    s->len=s->len+t.len;
  }
  else if (pos+t.len<=MAXLEN) { //插入后能保存t串的所有字，但不能存储s串所有字符
    for (i=MAXLEN-1; i>t.len+pos-1;i--) s->ch[i]=s->ch[i-t.len];
    for (i=0;i<t.len; i++) s->ch[i+pos] = t.ch[i];
    s->len=MAXLEN;
  }
  else { for (i=0;i<MAXLEN-pos;i++) s->ch[i+pos]=t.ch[i]; //插入后只能保存t串的部分字
    s->len = MAXLEN;      }
  return 1;
}
```

## 【算法4-2】：顺序串的删除函数

```
int StrDelete(SeqString *s, int pos, int k) /*在串s中删除从序号pos起k个字符*/  
{ int i;  
  if (pos<0 || pos>(s->len-k)) return 0;  
  for (i=pos+k;i<s->len;i++)  
    s->ch[i-k]=s->ch[i];  
  s->len=s->len - k;  
  return 1;  
}
```

**【算法4-3】** 串比较运算StrCompare (s, t)的算法。例如：

"ab" < "abcd"      "abcd" < "abd"

解：算法思路如下：

(1) 比较s和t两个串共同长度范围内的对应字符：

- ① 若s的字符 > t的字符，返回1；
- ② 若s的字符 < t的字符，返回-1；
- ③ 若s的字符 = t的字符，按上述规则继续比较。

(2) 当(1)中对应字符均相同时，比较s和t的长度：

- ① 两者相等时，返回0；
- ② s的长度 > t的长度，返回1；
- ③ s的长度 < t的长度，返回-1。

```

int StrCompare (SeqString s, SeqString t)
{
    int i, comlen;
    if (s.length<t.length) comlen=s.length; //求s和t的共同长度
    else comlen=t.length;
    for (i=0;i<comlen;i++) //在共同长度内逐个字符比较
        if (s.data[i]>t.data[i])
            return 1;
        else if (s.data[i]<t.data[i])
            return -1;

```

```

if (s.length==t.length)    //s==t
    return 0;
else if (s.length>t.length) //s>t
    return 1;
else return -1;          //s<t

```

}



所有共同长度内的字符相同，哪个长哪个大

# 4.2.2 堆串

以一组地址连续的存储单元存放串值字符序列，存储空间在程序执行过程中动态分配。系统将一个地址连续、容量很大的存储空间作为字符串的可用空间，每当建立新串时，系统就从该空间中分配大小和字符串长度相同的空间存储新串值。

所有串名的存储映像构成一个符号表，在串名和串值之间建立对应关系，称为串名的存储映像。

堆串的存储映象示例：a='a program', b='string ', c='process', free=23。

Heap[MAXSIZE]

free=23

a		p	r	o	g	r	a	m	s	t	r	i	n	g	
p	r	o	c	e	s	s									

符号表

符号名	len	start
a	9	0
b	7	9
c	7	16



用C语言中的“堆”实现堆串,其定义为:

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

其中: len域指示串的长度, ch域指示串的起始地址。

# 堆串的基本操作

## (1) 串赋值函数

**StrAssign(HString \*s; char \*tval) /\*将字符常量tval的值赋给串s \*/**

**{ int len, i=0;**

**if (s->ch!=NULL) free(s->ch);**

释放原来的串

**while (tval[i]!='\0') i++;**

**len=i;**

**if (len) { s->ch=(char \*)malloc(len);**

**if (s->ch==NULL) return(0);**

**for (i=0;i<len;i++) s->ch[i]=tval[i];**

**} else s->ch=NULL; s->len=len;**

**return(1);**

**}**

## (2) 串插入函数

**StrInsert(HString \*s, HString \*t, int pos) /\*在串s中序号为pos的字符之前插入串t \*/**

```
{ int i; char *temp;
```

```
if (pos<0 || pos>s->len || s->len==0) return(0);
```

```
temp=(char *)malloc(s->len + t.len);
```

```
if (temp==NULL) return(0);
```

```
for (i=0;i<pos;i++) temp[i]=s->ch[i];
```

```
for (i=0;i<t.len;i++) temp[i+pos]=t.ch[i];
```

```
for (i=pos;i<s->len;i++) temp[i + t.len]=s->ch[i];
```

```
s->len+=t.len;
```

```
free(s->ch);
```

```
s->ch=temp;
```

```
return(1);
```

```
}
```

### (3) 串删除函数

StrDelete(HString \*s, int pos, int len) /\*在串s中删除从序号pos起len个字符 \*/

```
{ int i; char *temp;
```

```
    if (pos<0 || pos>(s->len - len)) return(0);
```

```
    temp=(char *)malloc(s->len - len);
```

```
    if (temp==NULL) return(0);
```

```
    for (i=0;i<pos;i++) temp[i]=s->ch[i];
```

要删除区间前的字符复制过来

```
    for (i=pos;i<s->len - len;i++) temp[i]=s->ch[i+len];
```

```
    s->len=s->len-len; free(s->ch);s->ch=temp;
```

```
    return(1);
```

```
}
```

## (4) 串复制函数

**StrCopy(HString \*s, HString \*t) /\*将串t的值复制到串s中 \*/**

**{ int i;**

**s->ch=(char \*)malloc(t.len);**

**if (s->ch==NULL) return(0);**

**for (i=0;i<t.len;i++) s->ch[i]=t.ch[i];**

**s->len=t.len;**

**return(1);**

**}**

## (5) 判空函数

**StrEmpty(HString s) /\*若串s为空(即串长为0), 则返回1, 否则返回0 \*/**

**{**

**if (s.len==0) return(1);**

**else return(0);**

**}**

## (6) 串比较函数

```
StrCompare(HString s, HString t) /*若串s和t相等返回0， 若s>t返回1， 若s<t返回-1 */  
{  
    int i;  
    for (i=0;i<s.len&& i<t.len;i++)  
        if (s.ch[i]!=t.ch[i]) return(s.ch[i] - t.ch[i]);  
    return(s.len - t.len);  
}
```

## (7) 求串长函数

```
StrLength(HString s) /*返回串s的长度 */  
{  
    return(s.len);  
}
```



## (8) 清空函数

**StrClear(HString \*s) /\*将串s置为空串 \*/**

**{ if (s->ch!=NULL) free(s->ch);**

**s->ch=NULL;**

**s->len=0;**

**return(1);**

**}**

## (9) 连接函数

**StrCat(HString \*s, HString \*t) /\*将串t联接在串s的后面 \*/**

**{ int i; char \*temp;**

**temp=(char \*)malloc(s->len + t.len);**

**if (temp==NULL) return(0);**

**for (i=0;i<s->len;i++) temp[i]=s->ch[i];**

**for (i=s->len;i<s->len + t.len;i++) temp[i]=t.ch[i-s->len];**

**s->len+=t.len; free(s->ch);s->ch=temp;**

**return(1);**

**}**

## (10) 求子串函数

SubString(HString \*sub, HString \*s, int pos, int len) /\*将串s中序号pos起len个字符复制到sub中 \*/

```
{ int i;
  if (sub->ch!=NULL) free(sub->ch);
  if (pos<0 || pos>s.len || len<1 || len>s.len-pos) {
    sub->ch=NULL;
    sub->len=0;
    return(0);
  }
  else {sub->ch=(char *)malloc(len);
    if (sub->ch==NULL) return(0);
    for (i=0;i<len;i++) sub->ch[i]=s.ch[i+pos];
    sub->len=len; return(1);
  }
}
```

## (11) 定位函数

**StrIndex(HString s, int pos, HString t) /\*求串t在串s中的位置 \*/**

**{ int i=pos; int j=0;**

**if (s.len==0 || t.len==0) return(0);**

**while (i<s.len && j<t.len)**

**if (s.ch[i]==t.ch[j]) { i++;**

**j++;**

**}**

**else { i=i-j+1;**

**j=0;**

**}**

**if (j>=t.len) return(i-j);**

**else return(0);**

**}**

## 4.2.3 块链串

`#define BLOCK_SIZE <每结点存放字符个数>`

`typedef struct Block{`

`char ch[BLOCK_SIZE];`

`struct Block *next;`

`} Block;`

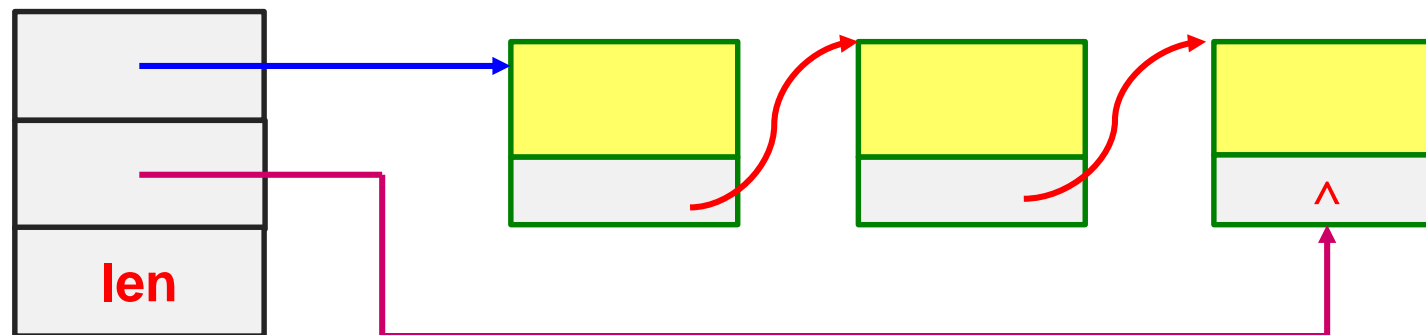
`typedef struct {`

`Block *head;`

`Block *tail;`

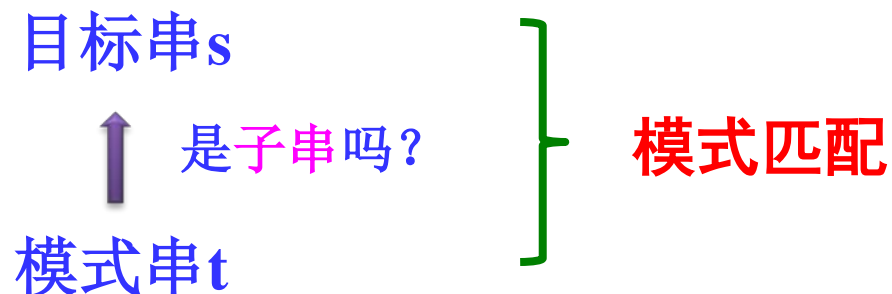
`int len;`

`} BLString;`



请同学们模仿堆串实现相应的算法

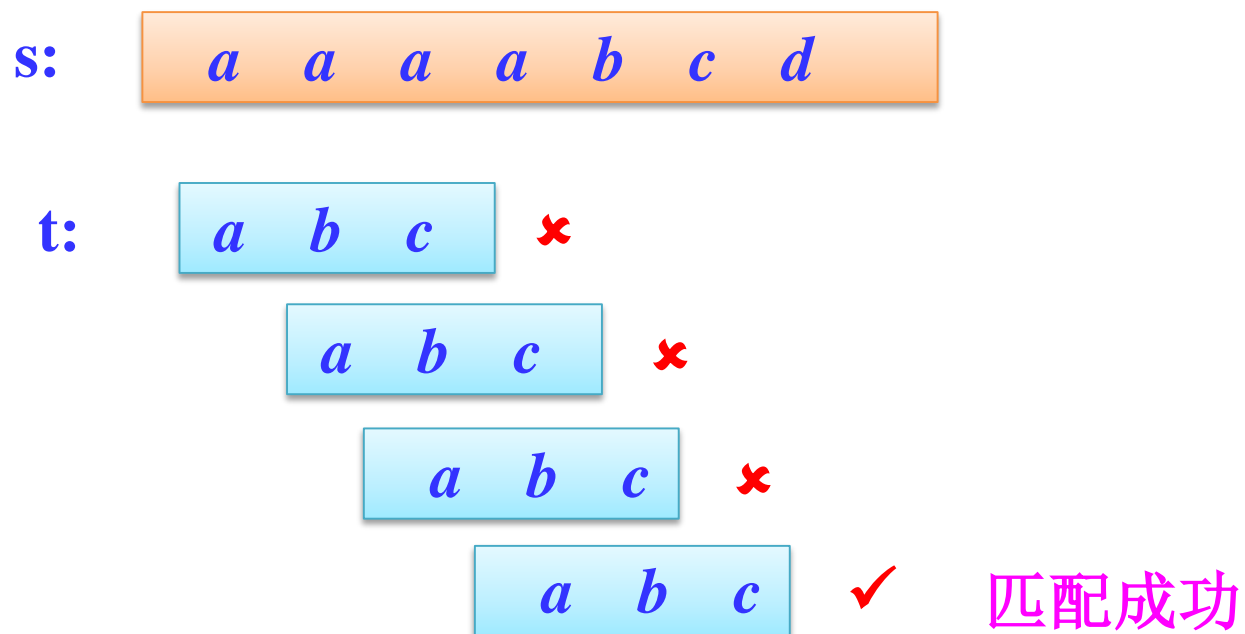
## 4.3 串的模式匹配



- ✓ **成功**是指在目标串s中找到一个模式串t——t是s的子串，返回t在s中的位置。
- ✓ **不成功**则指目标串s中不存在模式串t——t不是s的子串，返回-1。

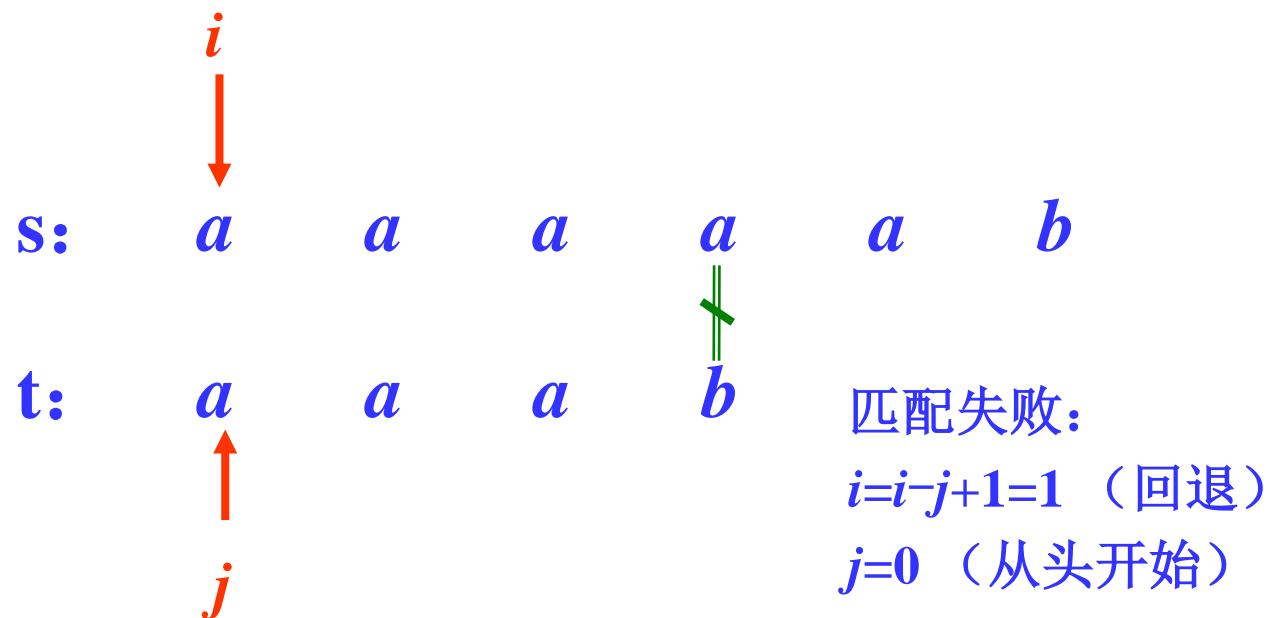
### 4.3.1 Brute-Force算法

Brute-Force简称为**BF算法**，亦称简单匹配算法。采用穷举的思路。



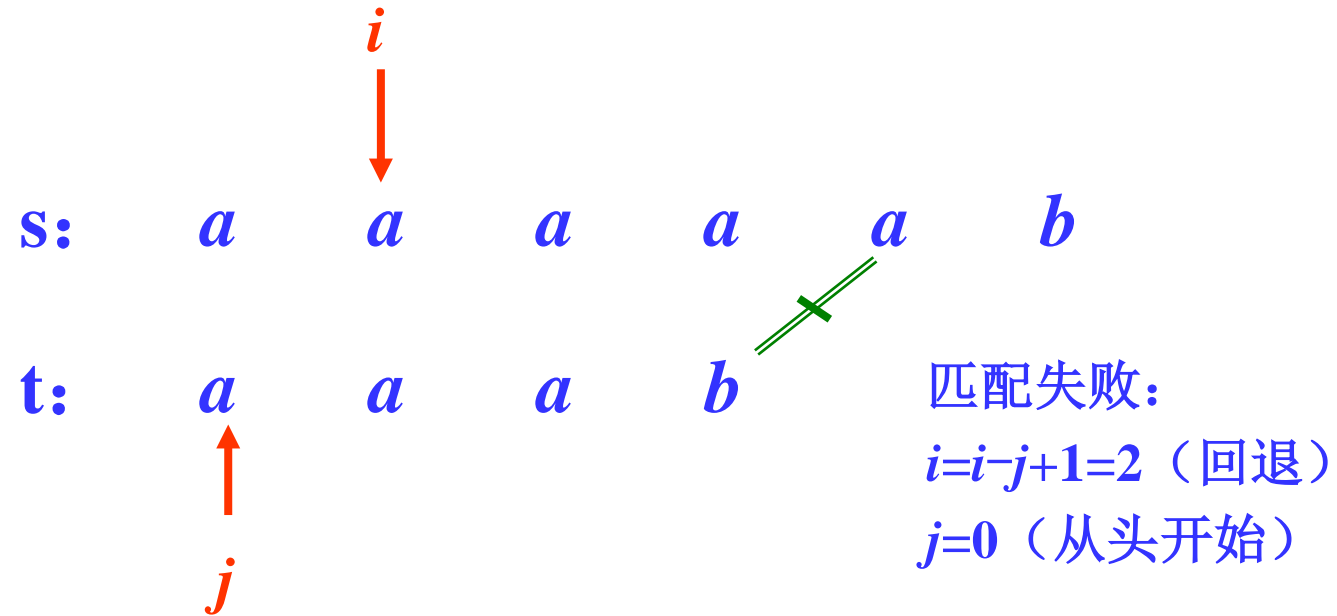
算法的思路是从s的每一个字符开始依次与t的字符进行匹配。

例如，设目标串 $s = \text{"aaaaab"}$ ，模式串 $t = \text{"aaab"}$ 。s的长度为 $n$  ( $n=6$ )，t的长度为 $m$  ( $m=4$ )。BF算法的匹配过程如下。





$i=1, j=0$



$i=2, j=0$

$s:$      $a$      $a$      $a$      $a$      $a$      $b$

$t:$      $a$      $a$      $a$      $b$

$i$  ↓

↑  $j$

匹配成功:

$i=6, j=4$

返回  $i-t.len=2$

对应的BF算法如下：

```
int StrIndex(SeqString s, int pos, SeqString t)
{   int i , j, start ;
    if(t.len==0) return(0);
    start = pos; i=start; j=0;
    while (i<s.len && j<t.len)
    {   if (s.ch[i]==t.ch[j])           //继续匹配下一个字符
        {   i++;                       //主串和子串依次匹配下一个字符
            j++;
        }
        else                           //主串、子串指针回溯重新开始下一次匹配
        {   i=i-j+1;                   //主串从下一个位置开始匹配
            j=0;                       //子串从头开始匹配
        }
    }
    if (j>=t.len) return(i-t.len);      //返回匹配的第一个字符的下标
    else
    return(-1);                         //模式匹配不成功
}
```

## BF算法分析:

- ✓ 算法在字符比较不相等, 需要回溯 (即 $i=i-j+1$ ): 即退到s中的下一个字符开始进行继续匹配。
- ✓ 最好情况下的时间复杂度为 $O(m)$ 。
- ✓ 最坏情况下的时间复杂度为 $O(n \times m)$ 。
- ✓ 平均的时间复杂度为 $O(n \times m)$ 。

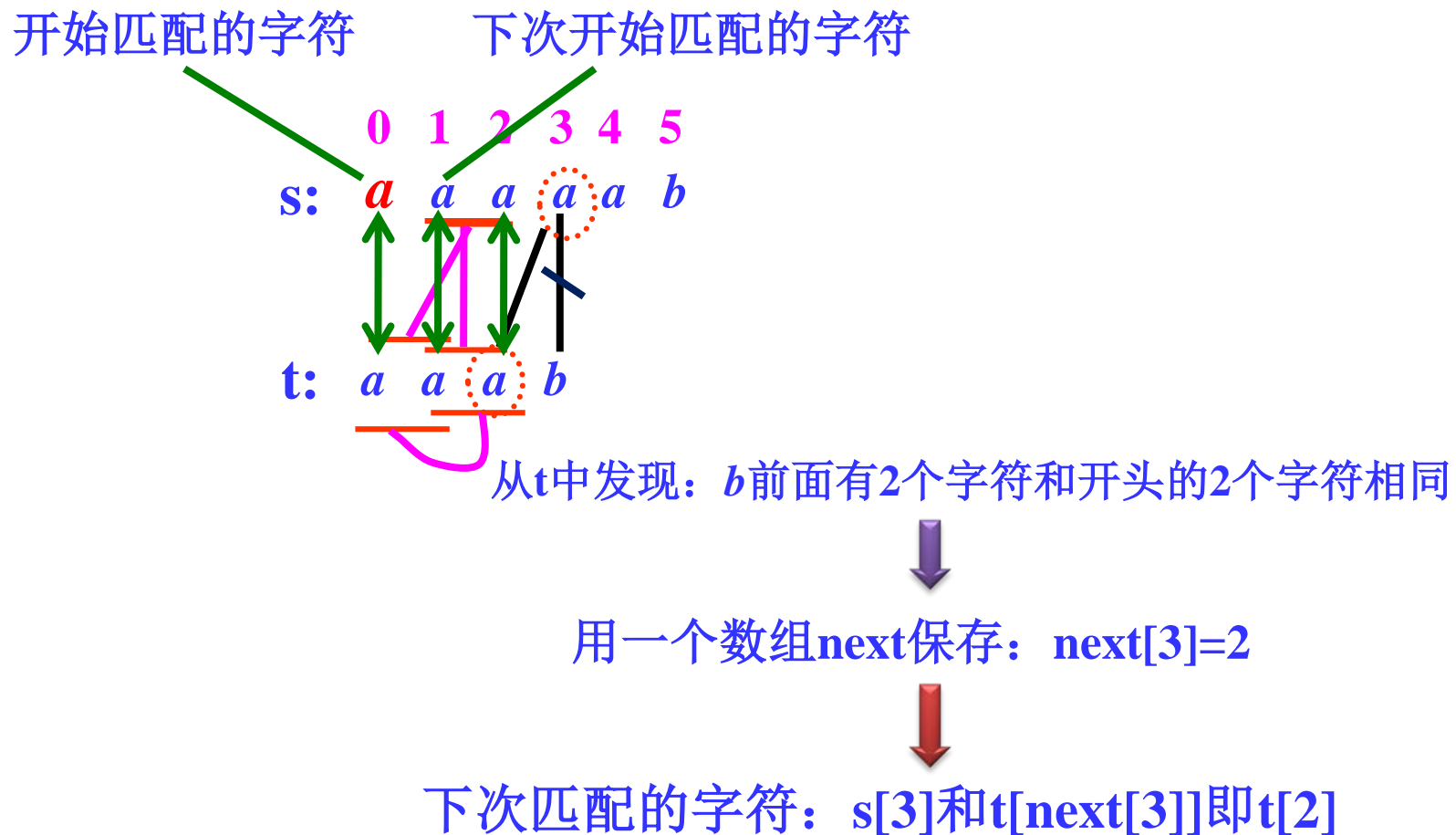
### 4.3.2 KMP算法

KMP算法是D.E.Knuth、J.H.Morris和V.R.Pratt共同提出的，简称**KMP**算法。

该算法较BF算法有较大改进，主要是消除了主串指针的回溯，从而使算法效率有了某种程度的提高。

## KMP算法用next数组保存部分匹配信息的演示

目标串s= “aaaaab”，模式串t= “aaab”。



# KMP算法部分匹配时的两个特征

$S = \text{"a b a b c a a b c c b a b"}$   
 $T = \text{"a b c a a b a b c"}$

$i$  ↓  
↑  $j$

(1) 模式滑动到第 $k$ 个字符, 有  $t_{j-k} \sim t_{j-1} = s_{i-k} \sim s_{i-1}$

(2) 再观察失配时, 有  $t_0 \sim t_{k-1} = t_{i-k} \sim t_{i-1}$

两式联立可得:  $t_0 \sim t_{k-1} = s_{j-k} \sim s_{j-1}$

**next[j]**是指**t[j]**字符前有多少个字符与**t**开头的字符相同。

模式串**t**存在某个**k** ( $0 < k < j$ )，使得以下成立：

$$\underbrace{\text{"}t_0t_1\cdots t_{k-1}\text{"}}_{\text{开头的}k\text{个字符}} = \underbrace{\text{"}t_{j-k}t_{j-k+1}\cdots t_{j-1}\text{"}}_{t[j]\text{前面的}k\text{个字符}}$$

例如， $t = \overset{0}{a} \overset{1}{b} \overset{2}{a} \overset{3}{b} \overset{4}{c}$  考虑  $t[4] = 'c'$



有  $t_0t_1 = t_2t_3 = "ab"$   $\Rightarrow k=2$

所以  $\text{next}[4] = k = 2$ 。



归纳起来，定义next[j]数组如下：

$$\text{next}[j] = \begin{cases} -1 & \text{当 } j=0 \text{ 时} \\ 0 & \text{其他情况} \end{cases}$$

$\text{MAX}\{ k \mid 0 < k < j, \text{ 且 } \overbrace{t_0 t_1 \dots t_{k-1}}^{\text{开头的 } k \text{ 个字符}} = \overbrace{t_{j-k} t_{j-k+1} \dots t_{j-1}}^{\text{后面的 } k \text{ 个字符}} \}$  当此集合非空时

t = “aaab” 对应的next数组如下：

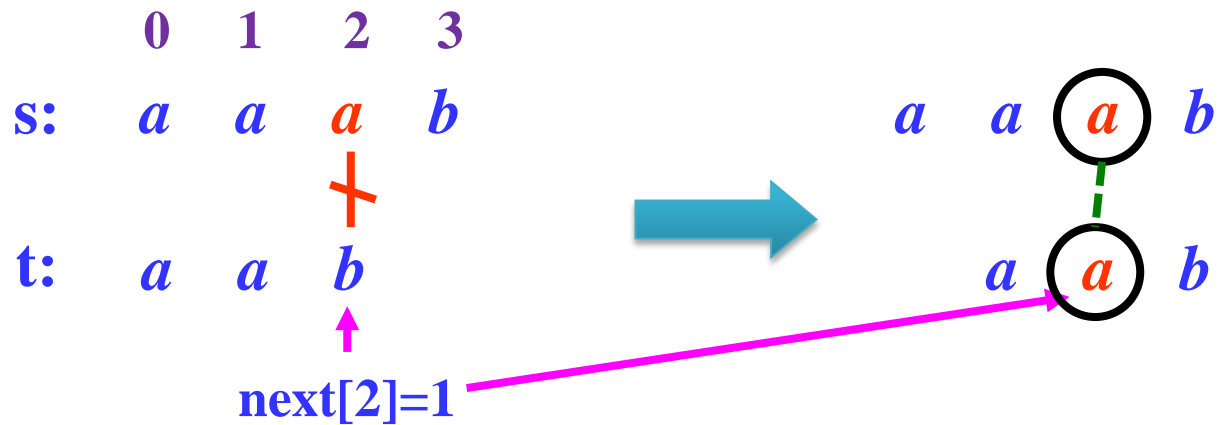
j	0	1	2	3
t[j]	a	a	a	b
next[j]	-1	0	1	2

$$t_0 = t_1 = "a" \quad t_0 t_1 = t_1 t_2 = "aa"$$

## next[j]的含义

### (1) $\text{next}[j]=k$ 表示什么信息？

说明模式串 $t[j]$ 之前有 $k$ 个字符已成功匹配，下一趟应从 $t[k]$ 开始匹配。



### (2) $\text{next}[j]=-1$ 表示什么信息？

说明模式串 $t[j]$ 之前没有任何用于加速匹配的信息，下一趟应从 $t$ 的开头即 $j++ \Rightarrow j=0$ 开始匹配。

如 $t = "abcd"$ ， $\text{next}[0]=-1; \text{next}[1]=\text{next}[2]=\text{next}[3]=0$ 。

## 由模式串t求next值的算法:

```
void GetNext(SeqString t, int next[])
{   int j, k;
    j=0; k=-1; next[0]=-1;
    while (j<t.len-1)
    {   if (k==-1 || t.ch[j]==t.ch[k])
        {   j++; k++;
            next[j]=k;
        }
        else k=next[k];
    }
}
```

## KMP算法:

```
int KMPIndex(SeqString s, SeqString t)
```

```
{  int next[MAXLEN], i=0, j=0;
```

```
    GetNext(t, next);
```

```
    while (i<s.len && j<t.len)
```

```
    {  if (j==-1 || s.ch[i]==t.ch[j])
```

```
        {  i++;
```

```
            j++;
```

//i、j各增1

```
        }
```

```
    }  else j=next[j];
```

//i不变, j后退

```
    if (j>=t.len)
```

```
        return(i-t.len);
```

//返回匹配模式串的首字符下标

```
    else
```

```
        return(-1);
```

//返回不匹配标志

```
}
```

主串位置不变, 子串重新定位 (右移)

没有有用信息或两个字符相等时, 继续比较后面的字符

## KMP算法分析

设串s的长度为 $n$ ，串t长度为 $m$ 。

在KMP算法中求next数组的时间复杂度为 $O(m)$ ，在后面的匹配中因主串s的下标不减即不回溯，比较次数可记为 $n$ ，所以KMP算法平均时间复杂度为 $O(n+m)$ 。

最坏的时间复杂度为 $O(n \times m)$ 。

【例（补充）】 已知字符串S为“*abaabaabacacaabaabcc*”，模式串t为“*abaabc*”，采用KMP算法进行匹配，第一次出现“失配” ( $s[i] \neq t[j]$ )时， $i=j=5$ ，则下次开始匹配时， $i$ 和 $j$ 的值分别是\_\_\_\_\_。

A. $i=1, j=0$     B. $i=5, j=0$     C. $i=5, j=2$     D. $i=6, j=2$

说明：本题为2015年全国考研题

$j$	0	1	2	3	4	5
$t[j]$	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>c</i>
$next[j]$	-1	0	0	1	1	2

选C

## 思考题

上述KMP算法仍然存在什么缺陷？

设目标串s=“*aaabaaaab*”，模式串t=“*aaaab*”。KMP模式匹配过程。

求t的next:

j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
next[j]	-1	0	1	2	3



<b>j</b>	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>
<b>t[j]</b>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
<b>next[j]</b>	-1	0	1	2	3

**0 1 2 3 4 5 6 7 8**  
**s:**    *a a a b a a a a b*  
       ↑↑↑↑  
**t:**    *a a a a b*  
       **0 1 2 3 4**

失败:  
*i=3*  
*j=3, j=next[3]=2*

j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
next[j]	-1	0	1	2	3

$i=3$   
 $j=2$

0 1 2 3 4 5 6 7 8

$s:$  *a a a b a a a a b*

$t:$  *a a a a b*

0 1 2 3 4

$\times$

失败:

$i=3$   
 $j=2, j=\text{next}[2]=1$

2020年3月24日

Design By Chen Anlong

50

j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
next[j]	-1	0	1	2	3

*i*=3

*j*=1

012345678

*s:*   *a a a b a a a a b*

*t:*   *a a a a b*

01234

失败:

*i*=3

*j*=1, *j*=next[1]=0

j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
next[j]	-1	0	1	2	3

*i*=3

*j*=0

*s*:

012345678

*a a a b a a a a b*

*t*:

*a a a a b*

01234

失败:

*i*=3

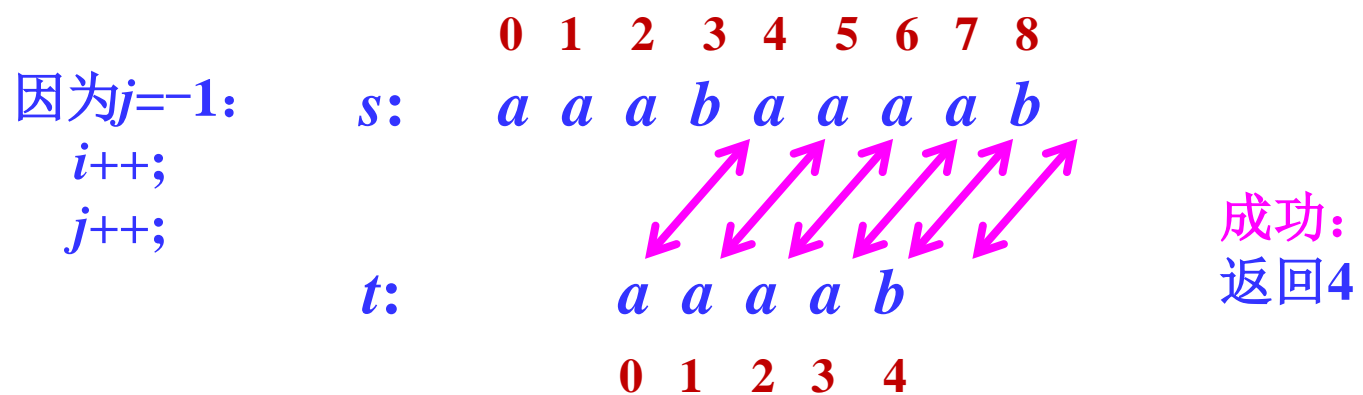
*j*=0, *j*=next[0]=-1

2020年3月24日

Design By Chen Anlong

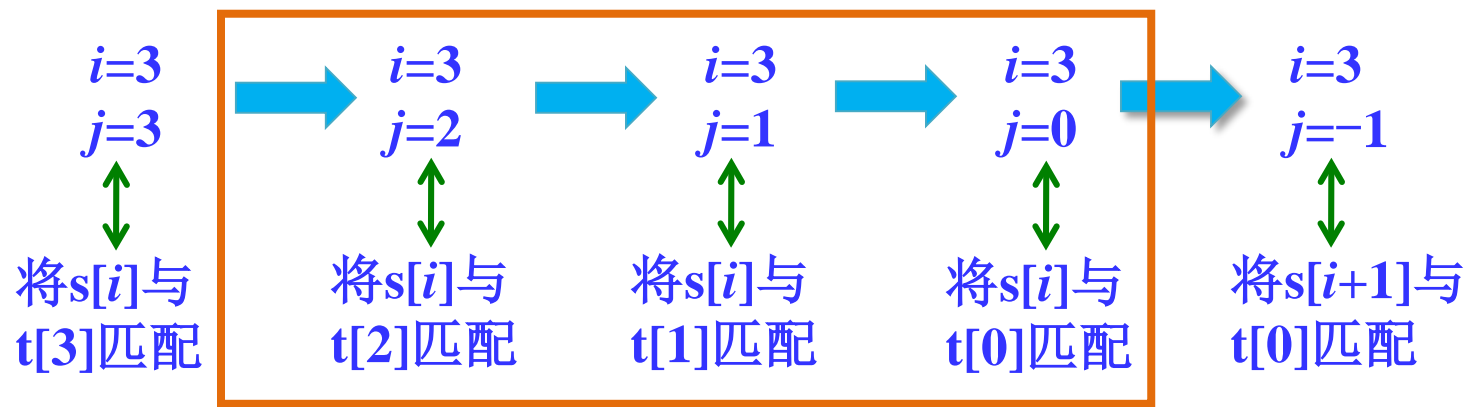
52

j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
next[j]	-1	0	1	2	3

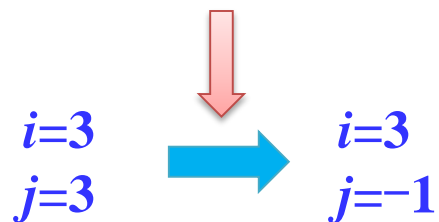


j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
next[j]	-1	0	1	2	3

前面的匹配过程：



因为 $t[3]=t[2]=t[1]=t[0]='a'$  是不必要的




## 4.2.3 改进的KMP算法

将next改为  
nextval:

j	0	1	2	3	4
t[j]	a	a	a	a	b
next[j]	-1	0	1	2	3
nextval[j]	-1	-1	-1	-1	3

$\text{next}[1]=0$   
 $t[1]=t[\text{next}[1]]=t[0]='a'$   
 $\therefore \text{nextval}[1]=\text{nextval}[0]=-1$

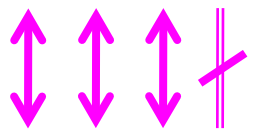
$t[4]='b' \neq t[\text{next}[4]]=t[3]='a'$   
 $\therefore \text{nextval}[4]=\text{next}[4]$

- 
- ✓  $\text{nextval}[0]=-1$
  - ✓ 当  $t[j]=t[\text{next}[j]]$  时:  $\text{nextval}[j]=\text{nextval}[\text{next}[j]]$
  - ✓ 否则:  $\text{nextval}[j]=\text{next}[j]$

用nextval取代next, 得到改进的KMP算法。

## 使用改进后的KMP算法示例：

j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
nextval[j]	-1	-1	-1	-1	3

0 1 2 3 4 5 6 7 8  
*s:*    *a a a b a a a a b*  
         
*t:*    *a a a a b*  
       0 1 2 3 4

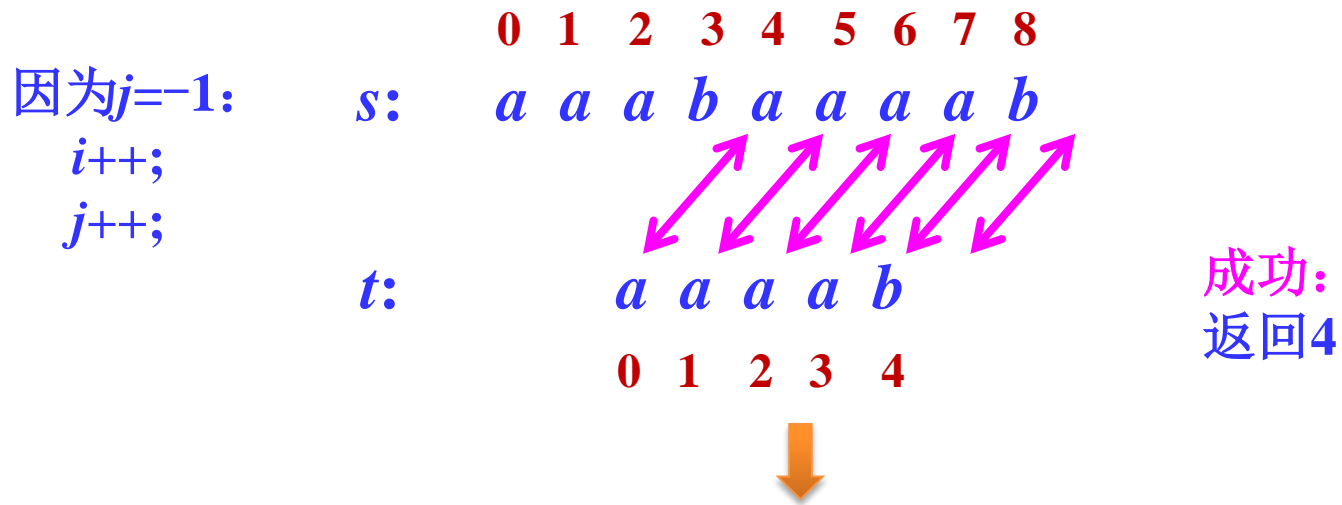
失败:

$i=3$

$j=3, j=\text{nextval}[3]=-1$



j	0	1	2	3	4
t[j]	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
nextval[j]	-1	-1	-1	-1	3



改进后的KMP算法进一步提高模式匹配的效率。

## 改进后求nextval的值

```
void GetNextval(SeqString t,int nextval[]) //由模式串t求出nextval值
{
    int j=0,k=-1;
    nextval[0]=-1;
    while (j<t.len)
    {
        if (k == -1 || t.ch[j]==t.ch[k])
        {
            j++;k++;
            if (t.ch[j] != t.ch[k]) nextval[j]=k;
            else nextval[j]=nextval[k];
        }
        else k=nextval[k];
    }
}
```

## 改进后的KMP算法

```
int KMPIndex1(SeqString s,SeqString t)  //修正的KMP算法
{  int nextval[MaxSize],i=0,j=0;
  GetNextval(t,nextval);
  while (i<s.len && j<t.len)
  {
    if (j==-1 || s.ch[i]==t.ch[j])
    { i++; j++; }
    else j=nextval[j];
  }
  if (j>=t.len) return(i-t.len);
  else return(-1);
}
```

本章结束