



北京交通大学
BEIJING JIAOTONG UNIVERSITY



第四章 串



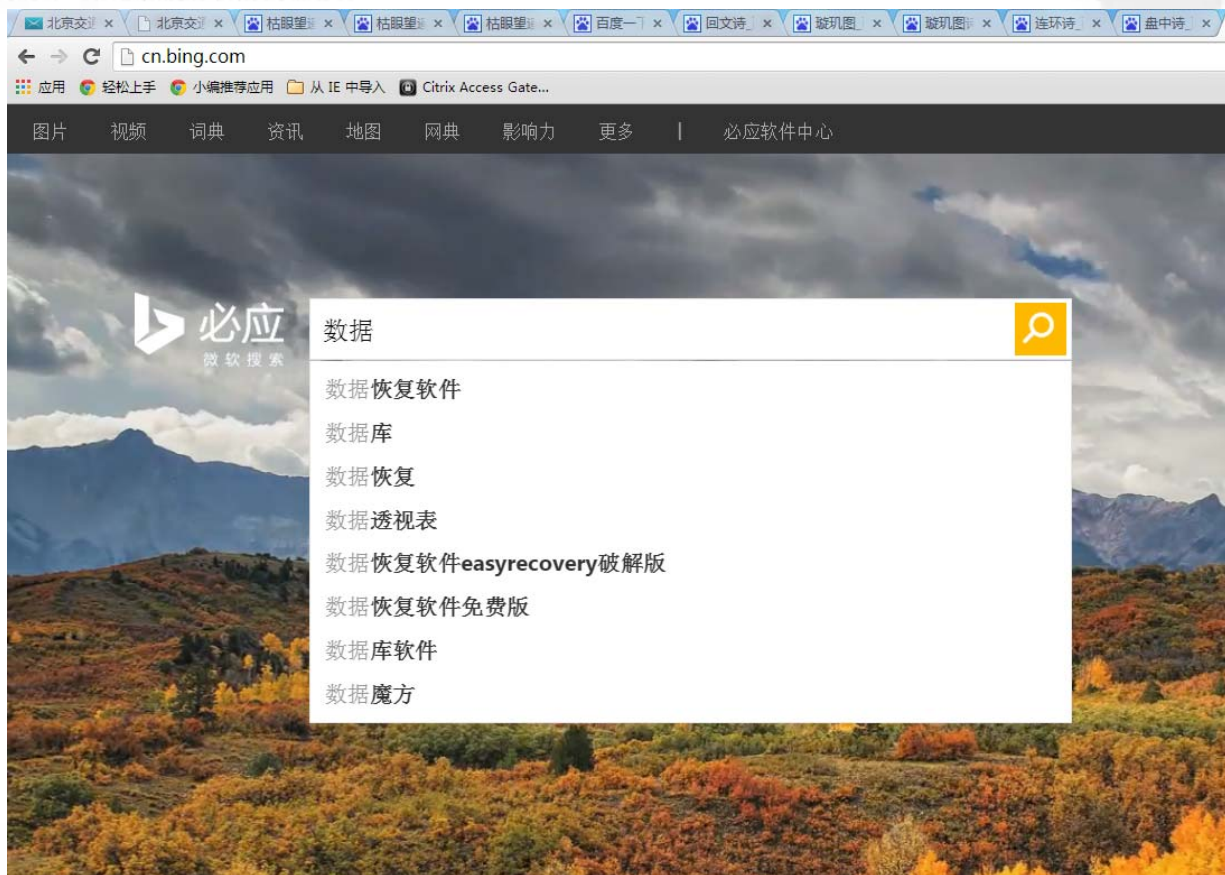


- ④ 4.1 串的定义
- ④ 4.2 串的抽象数据类型
- ④ 4.3 串的存储结构
- ④ 4.4 串的模式匹配算法



4.1 串的定义

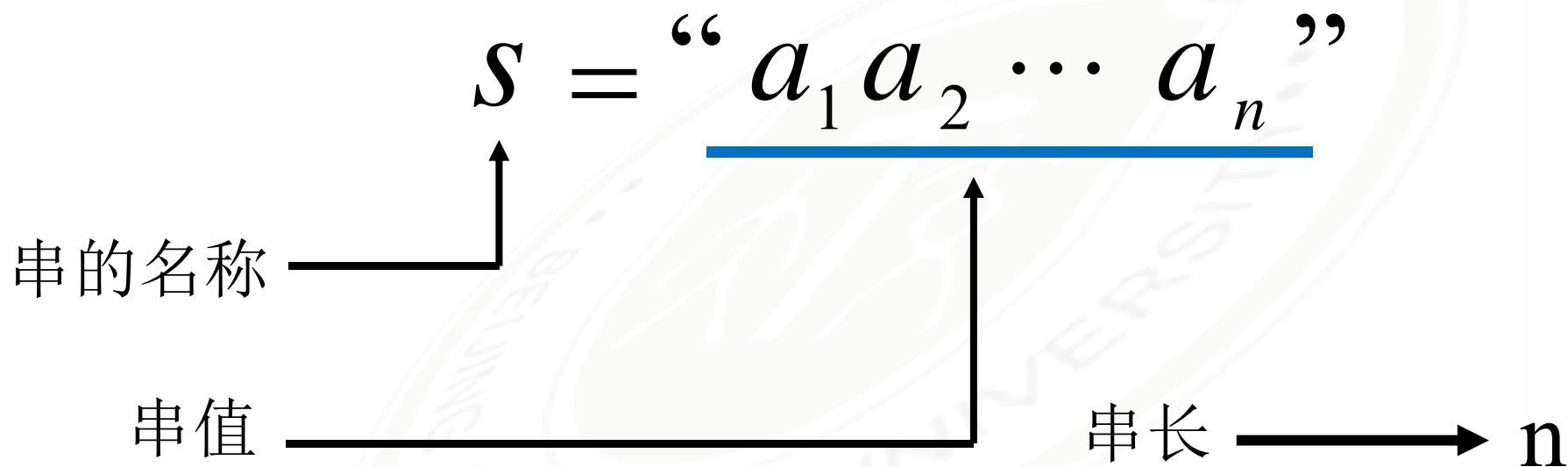
早先的计算机在被发明时，主要作用是做一些科学和工程的计算工作，也就是现在我们理解的计算器，只不过它比小小计算器功能更强大、速度更快一些。后来发现，在计算机上作非数值处理的工作越来越多，使得我们不得不需要引入对字符的处理。于是就有了字符串的概念。





4.1 串的定义

- 串(String): 零个或多个字符组成的有限序列。
- 一般记作:



- 零个字符的串称为空串，它的长度为0，可以直接用双引号 “ ” 表示，也可以用希腊字母 Φ 表示。



4.1 串的定义

a='BEI',

b='JING'

c='BEIJING'

d='BEI JING'

子串

字符位置

主串

子串位置

空格串

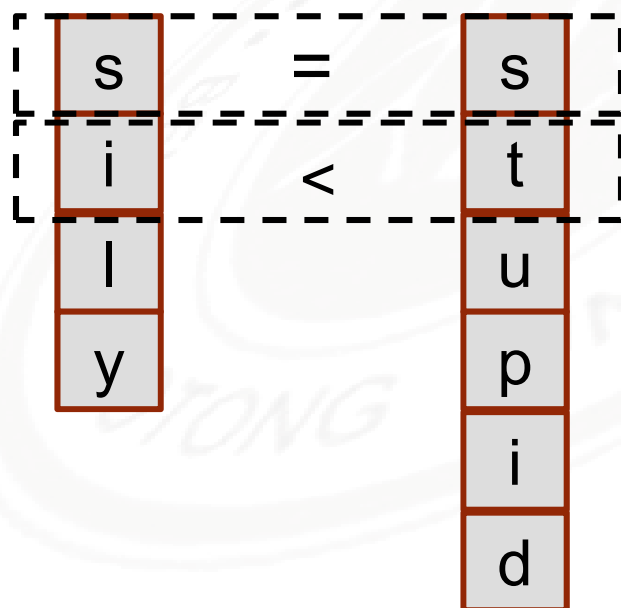


4.1 串的定义



串的比较

- 我们知道，数字很容易比较大小。那么，字符串如何比较大小呢？
- 规定：字符串的大小取决于它们挨个字符的**前后顺序**。
- 例如，“sily”和“stupid”比较大小。



“sily” < “stupid”



4.1 串的定义

复习ASCII编码

事实上，串的比较是通过组成串的字符之间的编码来进行的，而字符的编码指的是字符在对应字符集中的序号。

计算机中的常用字符是使用标准的 ASCII 编码，更准确一点，由 7 位二进制数表示一个字符，总共可以表示 128 个字符。后来发现一些特殊符号的出现，128 个不够用，于是扩展 ASCII 码由 8 位二进制数表示一个字符，总共可以表示 256 个字符，这已经足够满足以英语为主的语言和特殊符号进行输入、存储、输出等操作的字符需要了。可是，单我们国家就有除汉族外的满、回、藏、蒙古、维吾尔等多个少数民族文字，换作全世界估计要有成百上千种语言与文字，显然这 256 个字符是不够的，因此后来就有了 Unicode 编码，比较常用的是由 16 位的二进制数表示一个字符，这样总共就可以表示 2^{16} 个字符，约是 65 万多个字符，足够表示世界上所有语言的所有字符了。当然，为了和 ASCII 码兼容，Unicode 的前 256 个字符与 ASCII 码完全相同。



复习ASCII编码

H/L	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		☺	☹	♥	♦	♣	♠	•	◼	◯	⊕	♂	♀	♠	♠	✳
1	▶	◀	↕	!!	¶	§	—	‡	↑	↓	→	←	└	↔	▲	▼
2		!	"	#	\$	%	&	'	()	*+,-./	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	Δ



4.1 串的定义

串的比较

- 串的比较规则

那么对于两个串不相等时，如何判定它们的大小呢。我们这样定义：

给定两个串： $s = "a_1a_2\cdots a_n"$ ， $t = "b_1b_2\cdots b_m"$ ，当满足以下条件之一时， $s < t$ 。

1. $n < m$ ，且 $a_i = b_i$ ($i=1, 2, \cdots, n$)。

例如当 $s = "hap"$ ， $t = "happy"$ ，就有 $s < t$ 。因为 t 比 s 多出了两个字母。

2. 存在某个 $k \leq \min(m, n)$ ，使得 $a_i = b_i$ ($i=1, 2, \cdots, k-1$)， $a_k < b_k$ 。

例如当 $s = "happen"$ ， $t = "happy"$ ，因为两串的前 4 个字母均相同，而两串第 5 个字母 (k 值)，字母 e 的 ASCII 码是 101，而字母 y 的 ASCII 码是 121，显然 $e < y$ ，所以 $s < t$ 。



4.1 串的定义



串的比较

- 串相等的条件

所以如果我们要在 C 语言中比较两个串是否相等，必须是它们串的长度以及它们各个对应位置的字符都相等时，才算是相等。即给定两个串： $s = "a_1a_2\cdots a_n"$ ， $t = "b_1b_2\cdots b_m"$ ，当且仅当 $n=m$ ，且 $a_1=b_1$ ， $a_2=b_2$ ， \cdots ， $a_n=b_m$ 时，我们认为 $s=t$ 。



- ④ 4.1 串的定义
- ④ 4.2 串的抽象数据类型
- ④ 4.3 串的存储结构（定长顺序）
- ④ 4.4 串的模式匹配算法



ADT String {

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

数据关系: $R_1 = \{ \langle a_{i-1}, a_i \mid a_{i-1}, a_i \in D, i = 1, 2, \dots, n \rangle \}$

串的逻辑结构和线性表很相似，不同之处在于串针对的是字符集，也就是串中的元素都是字符，哪怕串中的字符是“123”这样的数字组成，或者“2010-10-10”这样的日期组成，它们都只能理解为长度为 3 和长度为 10 的字符串，每个元素都是字符而已。

基本操作:

因此,对于串的基本操作与线性表是有很大差别的。线性表更关注的是单个元素的操作,比如查找一个元素,插入或删除一个元素,但串中更多的是查找子串位置、得到指定位置子串、替换子串等操作。

- √ (1) **StrAssign (&T, *chars)** //生成一个其值等于字符串常量chars的串T
- √ (2) **StrCompare (S,T)** //若 $S>T$, 返回值 >0 , $S=T$, 返回0, $S<T$, 返回值 <0
- √ (3) **StrLength (S)** //求串长
- √ (4) **Concat(&T,S1,S2)** //串联接, 用T返回S1和S2联接而成的新串
- √ (5) **SubString(&Sub,S,pos,len)** //求子串, 用Sub返回串S的第pos个字符起长度为len的子串

基本操作:

- (6) StrCopy(&T,S) //串拷贝
 - (7) StrEmpty(S) //串判空
 - (8) ClearString (&S) //清空串
 - (9) StrInsert(&S,pos,T) //子串插入
 - (10) StrDelete(&S,pos,len) //子串删除
 - (11) **Index(S,T,pos)** //求子串的位置
 - (12) **Replace(&S,T,V)** //串替换
- }ADT String



- ④ 4.1 串的定义
- ④ 4.2 串的抽象数据类型
- ④ 4.3 串的存储结构
- ④ 4.4 串的模式匹配算法



4.3 串的存储结构



顺序存储结构

- 定长顺序表示

```
#define MAXSTRLEN 255
```

```
typedef unsigned char SString[MAXSTRLEN+1];
```

//0号单元存放串长

```
SString T;
```



↑
T[0]

↑
MAXSTRLEN+1



- (1) **StrAssign (&T, *chars)** //生成一个其值等于字符串常量chars的串T
- (2) **StrCompare (S,T)** //若 $S>T$, 返回值 >0 , $S=T$, 返回0, $S<T$, 返回值 <0
- (3) **StrLength (S)** //求串长
- (4) **Concat(&T,S1,S2)** //串联接, 用T返回S1和S2联接而成的新串
- (5) **SubString(&Sub,S,pos,len)** //求子串



(1) 串赋值- 定长顺序

Status StrAssign(SString T,char *chars)

```
{ // 生成一个其值等于chars的串T
```

```
    if(strlen(chars)>MAXSTRLEN)
```

```
        return ERROR;
```

```
    else
```

```
    {
```

```
        T[0]=strlen(chars);
```

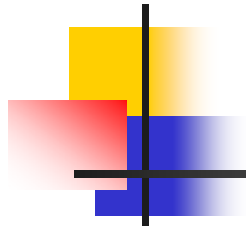
```
        for(i=1;i<=T[0];i++)
```

```
            T[i]=*(chars+i-1);
```

```
        return OK;
```

```
    }
```

```
} //StrAssign
```



(2) 串比较- 定长顺序

```
int StrCompare(SString S,SString 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 $< 0$ 
```

```
    for(i=1;i<=S[0]&& i<=T[0];++i)
```

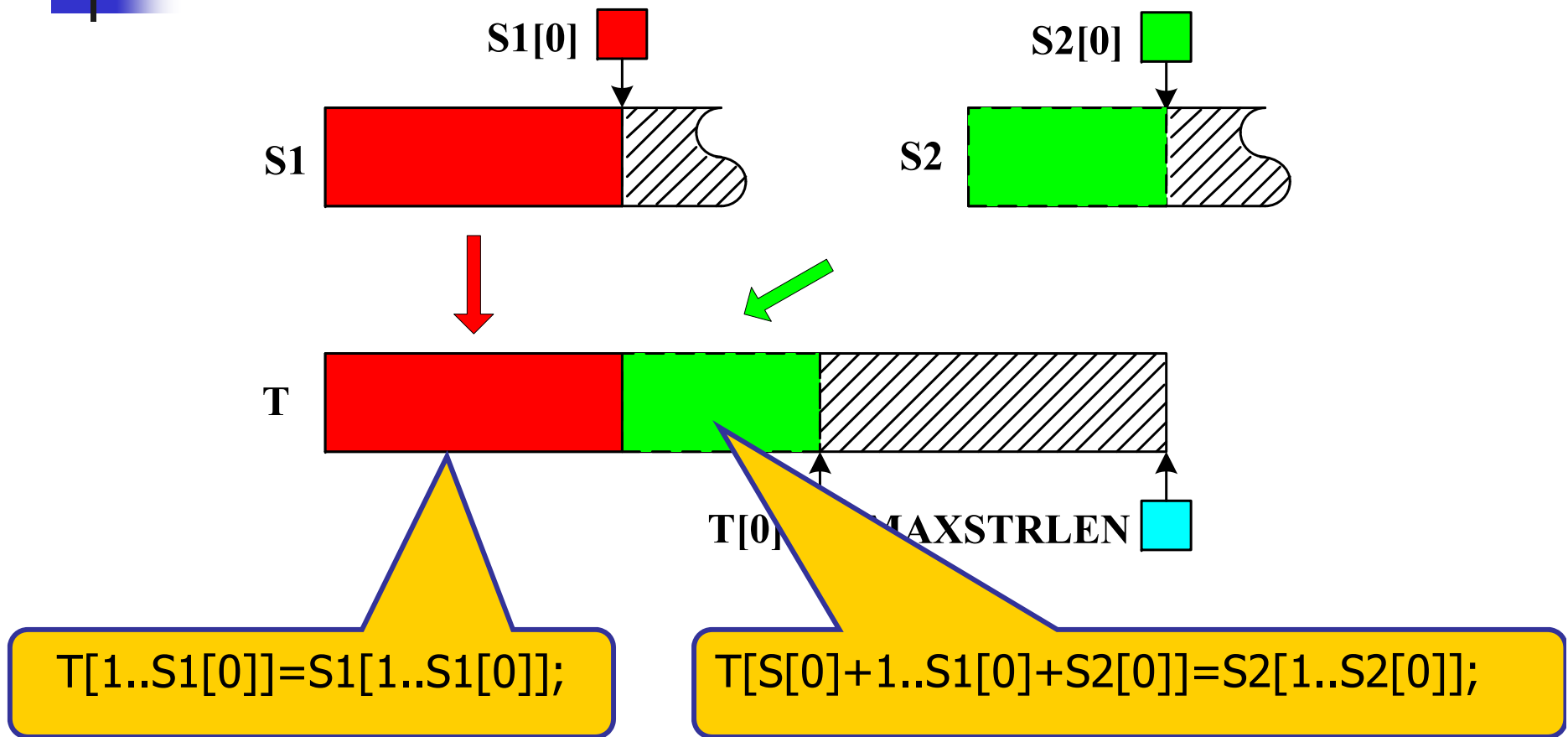
```
        if(S[i]!=T[i])
```

```
            return S[i]-T[i];
```

```
    return S[0]-T[0];
```

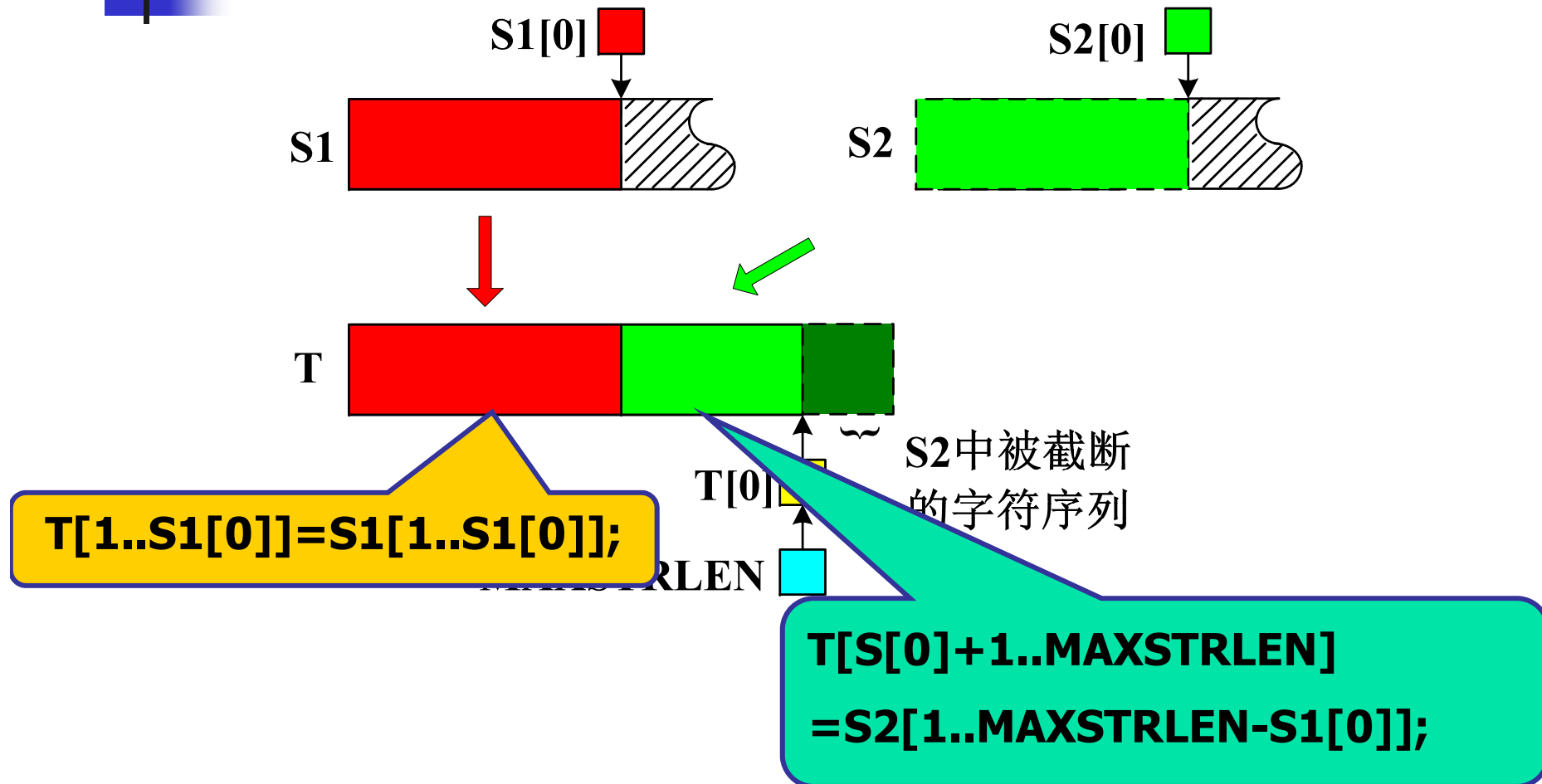
```
}// StrCompare
```

(4) 串连接- 定长顺序



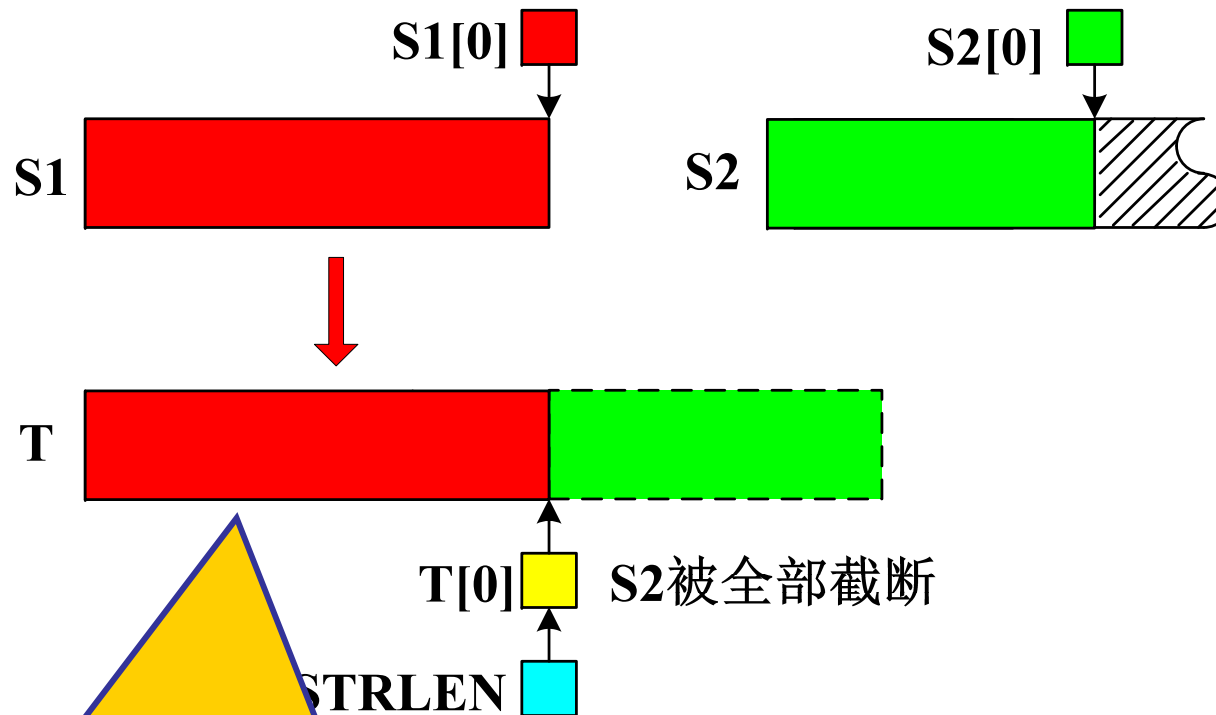
(a) $S1[0] + S2[0] \leq MAXSTRLEN$

(4) 串连接- 定长顺序



(b) $S1[0] < MAXSTRLEN$ 而
 $S1[0] + S2[0] > MAXSTRLEN$

(4) 串连接- 定长顺序



(c) $S1[0] \leq MAXSTRLEN$

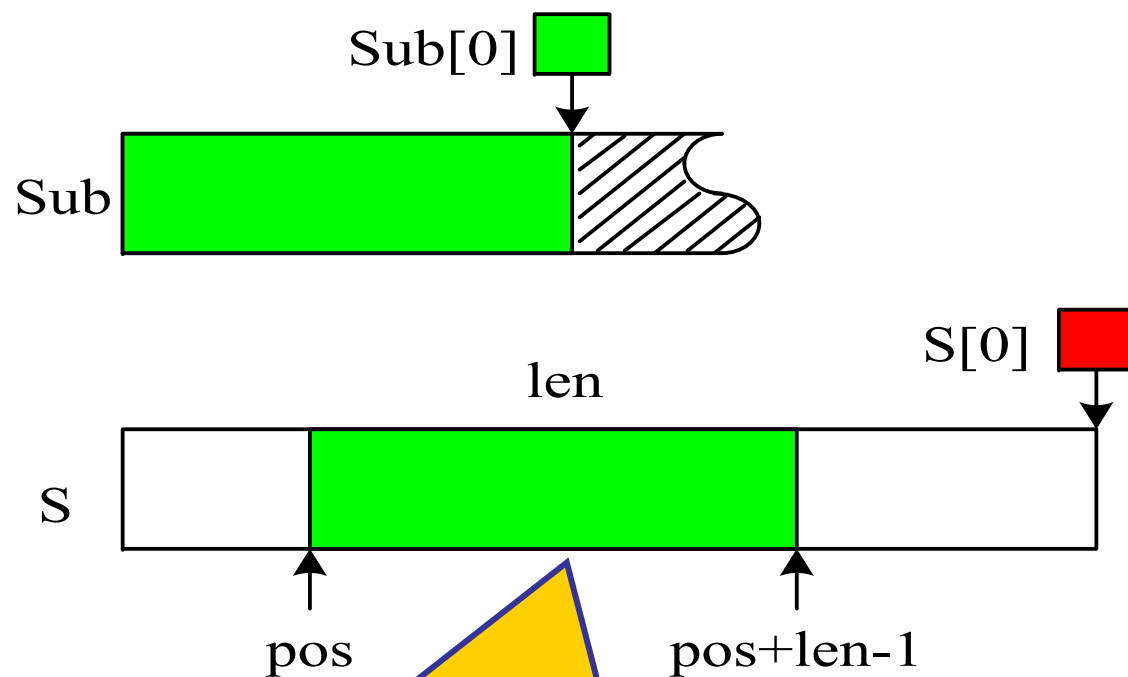
```

Status Concat(SString &T,SString S1,SString S2){
    //用T返回由S1和S2联接而成的新串.若未截断,则返回TRUE,否则返回FALSE
    if(S1[0]+S2[0]<=MAXSTRLEN){          //未截断
        T[1..S1[0]]=S1[1..S1[0]];
        T[S1[0]+1..S1[0]+S2[0]]=S2[1..S2[0]];
        T[0]=S1[0]+S2[0];  uncut=TRUE;
    }
    else if(S1[0]<MAXSTRLEN){              //截断
        T[1..S1[0]]=S1[1..S1[0]];
        T[S1[0]+1..MAXSTRLEN]=S2[1..MAXSTRLEN-S1[0]];
        T[0]=MAXSTRLEN;uncut=FALSE;
    }
    else{                                  //截断(仅取S1)
        T[0..MAXSTRLEN]=S1[1..MAXSTRLEN];
        uncut=FALSE;
    }
    return uncut;
} //Concat

```

(5) 求子串- 定长顺序

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



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

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

```
//用Sub串返回串S的第pos个字符起长度为len的子串
```

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

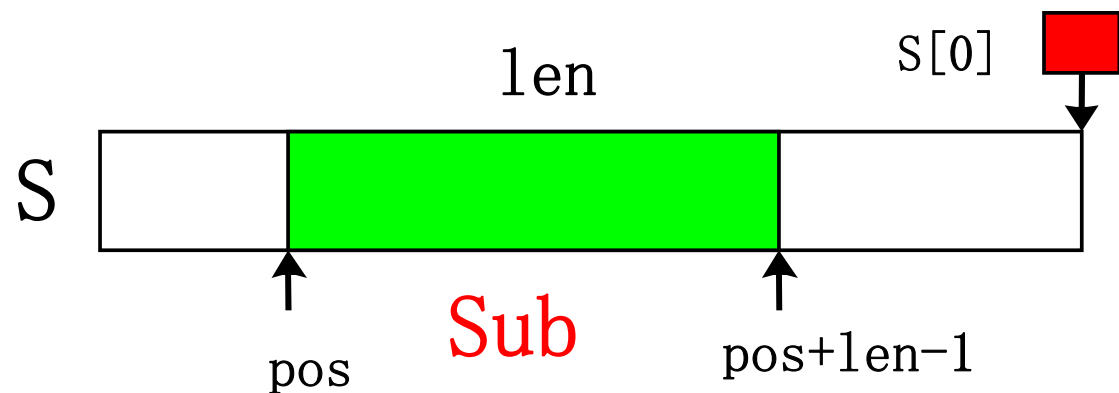
```
    return ERROR;
```

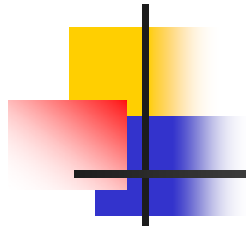
```
Sub[1..len]=S[pos..pos+len-1];
```

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

```
return OK;
```

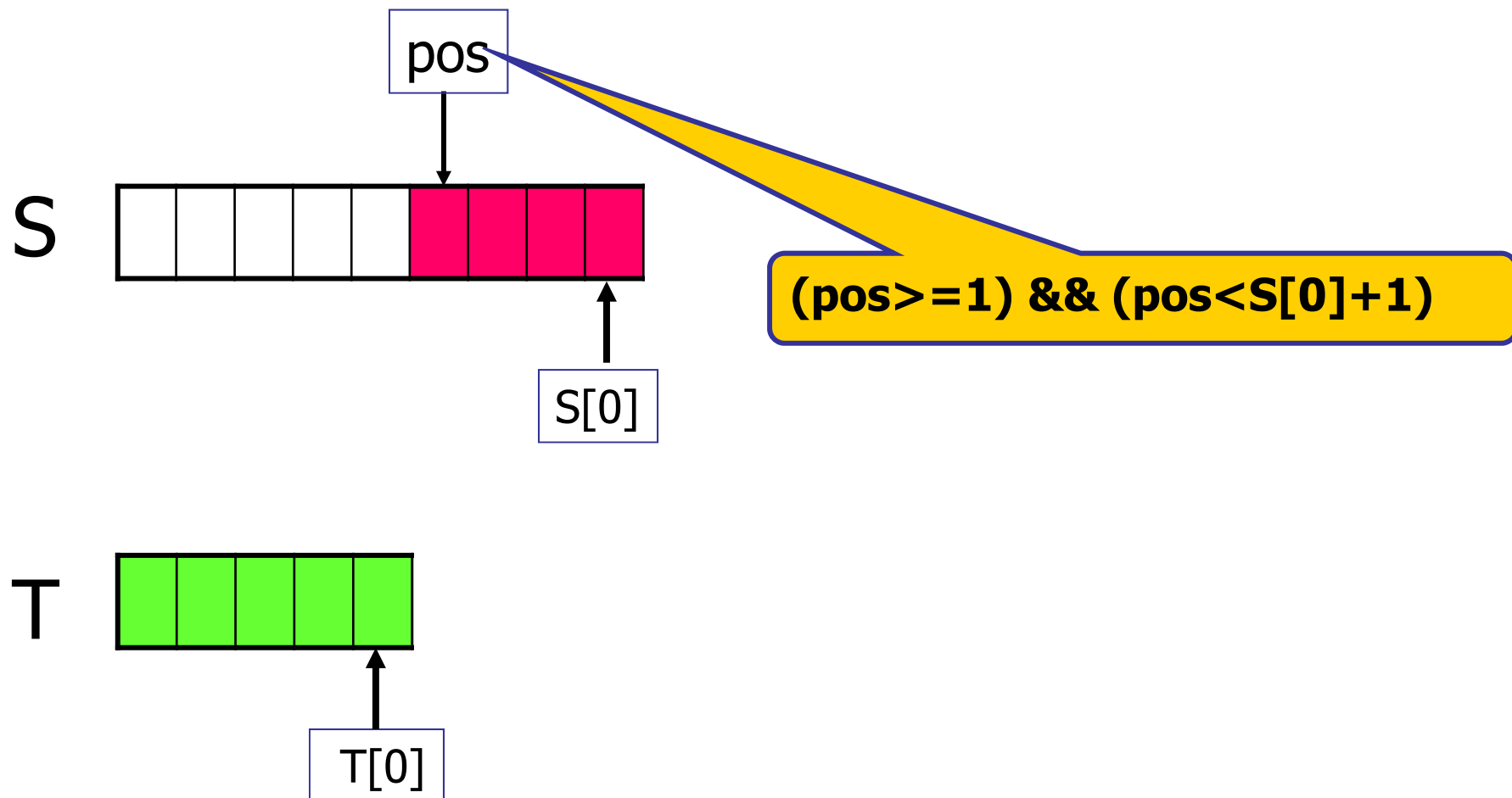
```
}//SubString
```





(9) 串插入- 定长顺序

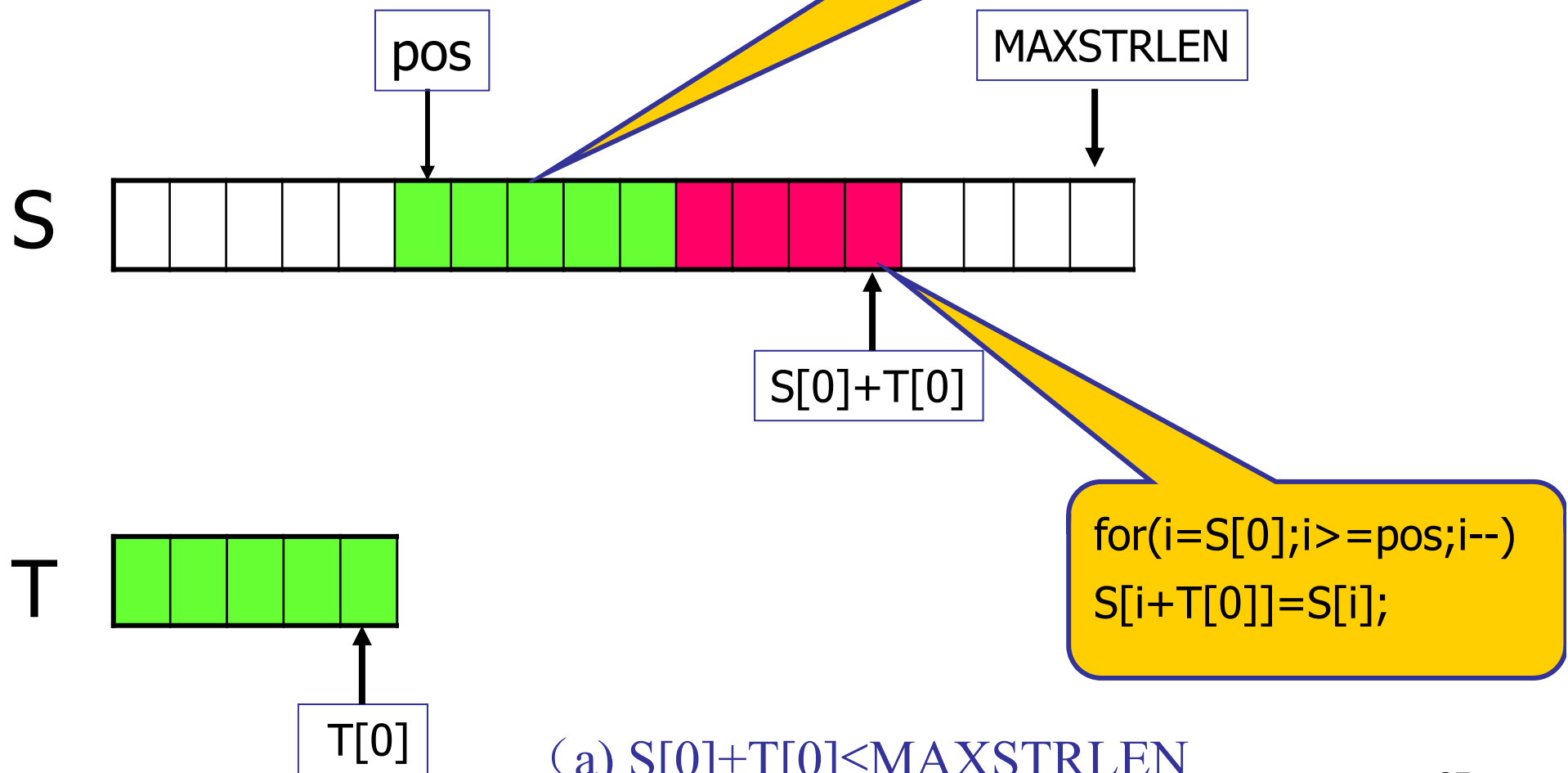
Status StrInsert(SString S,int pos,SString T)



(9) 串插入-定

```
for(i=pos;i<pos+T[0];i++)  
S[i]=T[i-pos+1];
```

Status StrInsert(SString S,int pos,SString T)

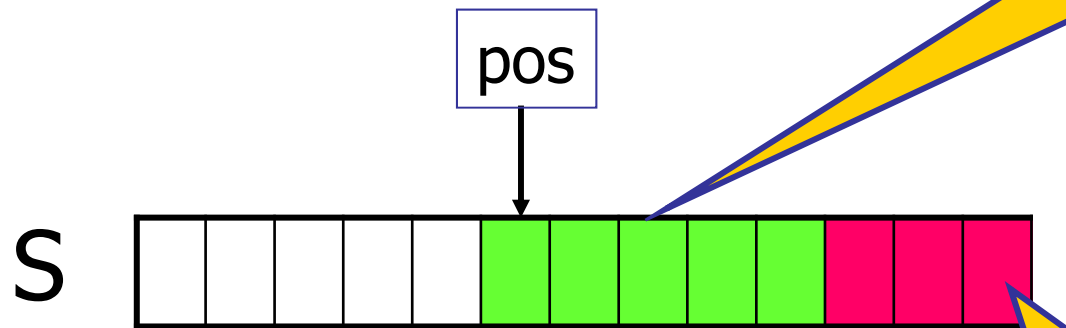


(a) $S[0] + T[0] \leq \text{MAXSTRLEN}$

(9) 串插入-定

```
for(i=pos;i<pos+T[0];i++)  
S[i]=T[i-pos+1];
```

Status StrInsert(SString S,int pos,SString T)



MAXSTRLEN



T[0]

```
for(i=MAXSTRLEN;i>=pos+T[0];i--)  
S[i]=S[i-T[0]]
```

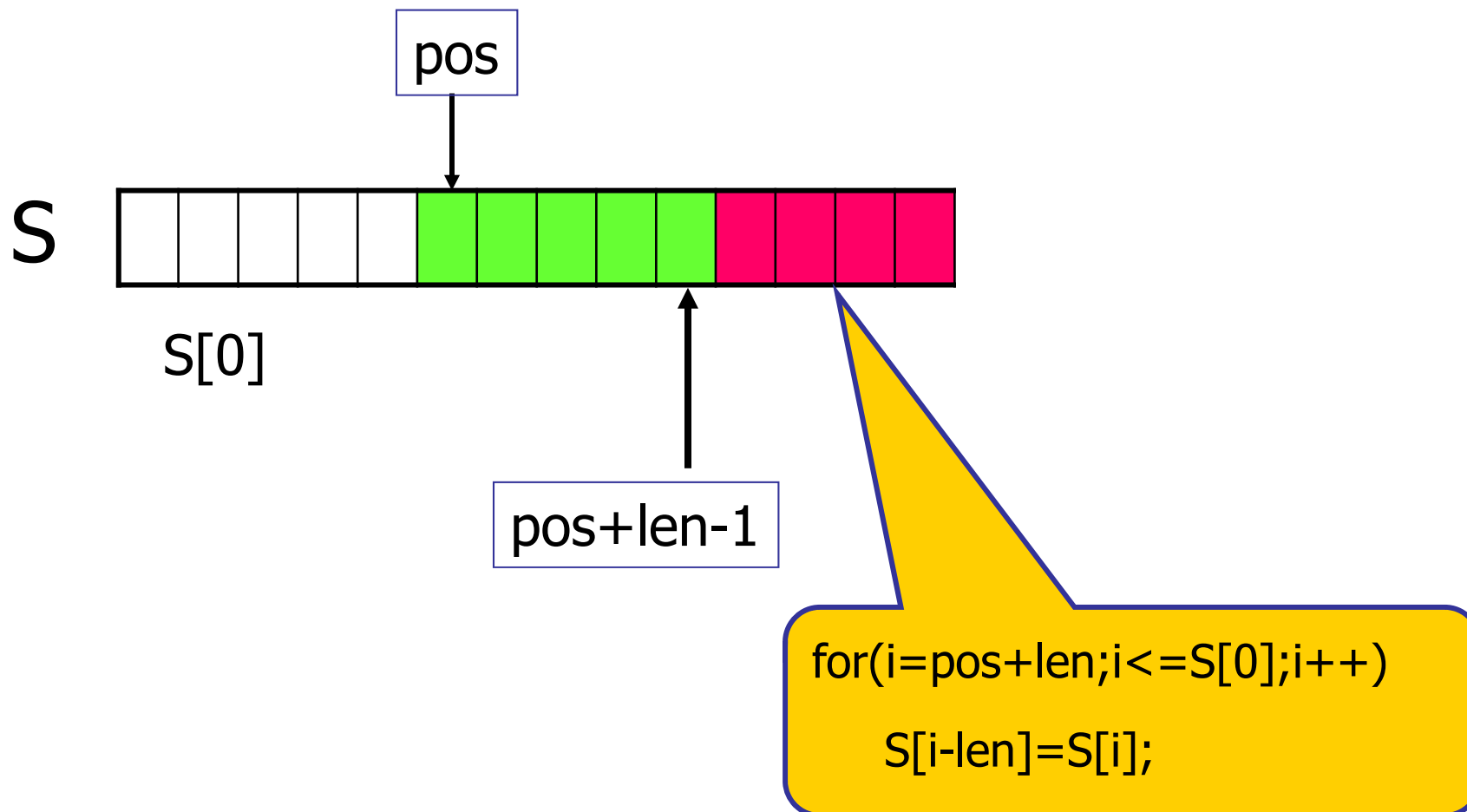
(b) $S[0] < \text{MAXSTRLEN}$ 而
 $S[0] + T[0] > \text{MAXSTRLEN}$

Status StrInsert(SString &S,int pos,SString T)

```
{
    if(pos<1||pos>S[0]+1)    return ERROR;
    if(S[0]+T[0]<=MAXSTRLEN) { // 完全插入
        for(i=S[0];i>=pos;i--)  S[i+T[0]]=S[i];
        for(i=pos;i<pos+T[0];i++) S[i]=T[i-pos+1];
        S[0]=S[0]+T[0];
        return TRUE;
    }
    else { // 部分插入
        for(i=MAXSTRLEN;i>=pos+T[0];i--)  S[i]=S[i-T[0]];
        for(i=pos;i<pos+T[0];i++)  S[i]=T[i-pos+1];
        S[0]=MAXSTRLEN;
        return FALSE;
    }
}
```

(10) 串删除- 定长顺序

Status StrDelete(HString &S,int pos,int len){





(10) 串删除

Status StrDelete(SString S,int pos,int len)

{ //从串S中删除第pos个字符起长度为len的子串

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

return ERROR;

for(i=pos+len;i<=S[0];i++)

S[i-len]=S[i];

S[0]-=len;

return OK;

}



4.3 串的存储结构



顺序存储结构

- 堆分配存储表示

```
typedef struct {  
    char *ch;           // 若串非空, 则按串长分配存储区,  
                        // 否则ch为NULL  
    int  length;        // 串长度  
} HString;
```




(1) 串赋值- 定长顺序

Status StrAssign(SString T,char *chars)

{ // 生成一个其值等于chars的串T

~~if(strlen(chars)>MAXSTRLEN)~~

~~return ERROR;~~

~~else~~

~~{~~

~~T[0]=strlen(chars);~~

for(i=1;i<=T[0];i++)

T[i]=*(chars+i);

return OK;

}

} //StrAssign



(1) 串赋值-堆分配

```
Status StrAssign(HString &T,char *chars){  
    if(T.ch) free(T.ch);           //释放T原有空间  
    for(i=0,c=chars;*c!='\0';++i,++c); //求chars的长度i  
    if(!i){ T.ch=NULL;T.length=0;}  
    else{  
        if(!(T.ch=(char*) malloc(i*sizeof(char))))  
            return ERROR;  
        T.ch[0..i-1]=chars[0..i-1];  
        T.length=i;  
    }  
    return OK;  
} //StrAssign
```



(2) 串比较- 定长顺序

```
int StrCompare(SString S,SString 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 $<0$ 
```

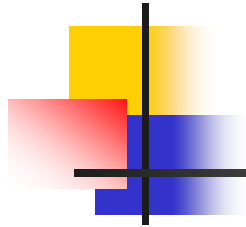
```
    for( $i=1;i\leq S[0]\&\&i\leq T[0];++i$ )
```

```
        if( $S[i]\neq T[i]$ )
```

```
            return  $S[i]-T[i]$ ;
```

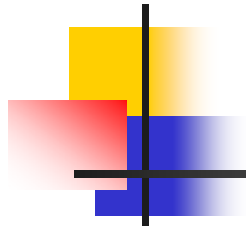
```
    return  $S[0]-T[0]$ ;
```

```
}// StrCompare
```



(2) 串比较- 堆分配

```
int StrCompare(HString S,HString T){  
    //若S>T,返回值>0;若S=T,返回值=0;若S<T,返回值<0;  
    for(i=0;i<S.length && i<T.length;++i)  
        if(S.ch[i]!=T.ch[i] return (S.ch[i]-T.ch[i]));  
    return S.length-T.length;  
} //StrCompare
```



(4) 串联接- 堆分配

```
Status Concat(HString &T,HString S1,HString S2){
```

```
//用T返回由S1和S2联接而成的新串
```

```
if(T.ch) free(T.ch);    //释放旧空间
```

```
if(!(T.ch=(char *)malloc(S1.length+S2.length)*sizeof(char))))
```

```
    return ERROR;
```


```
T.ch[0..S1.length-1]=S1.ch[0..S1.length-1];
```

```
T.length=S1.length+S2.length;
```

```
T.ch[S1.length..T.length-1]=S2.ch[0..S2.length-1];
```

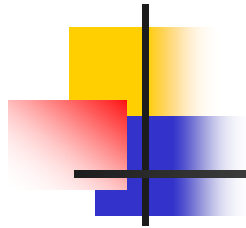
```
return OK;
```

```
}//Concat
```



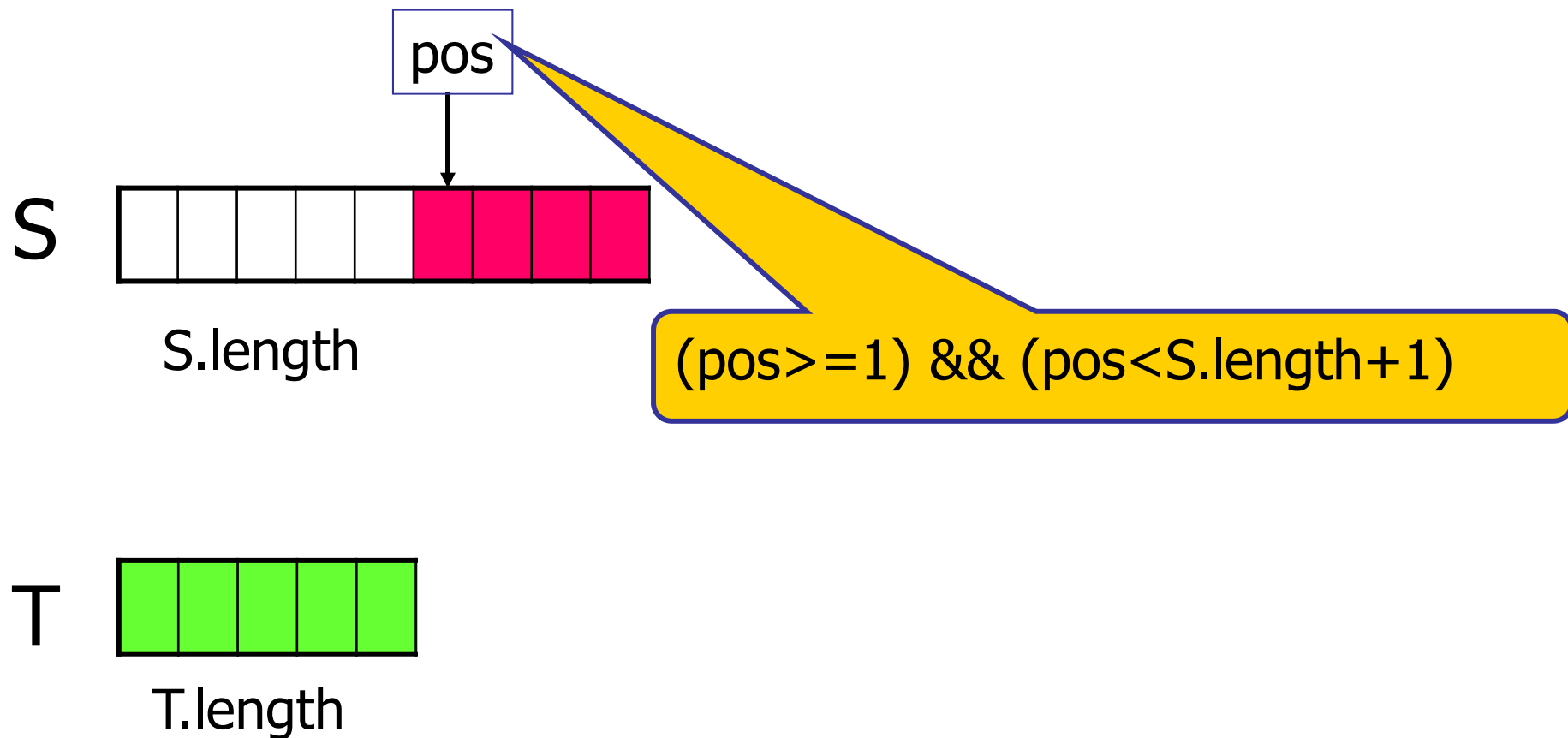
(5) 求子串- 堆分配

```
Status SubString(HString &Sub,HString S, int pos,int len){  
    if(pos<1 || pos>S.length || len<0 || len>S.length-pos+1)  
        return ERROR;  
    if(Sub.ch) free(Sub.ch);  
    Sub.ch =(char *)malloc(len*sizeof(char));  
    if(!Sub.ch) return ERROR;  
    Sub.ch[0..len-1]=S[pos-1..pos+len-2];  
    Sub.length=len;  
    return OK;  
} //SubString
```



(9) 串插入- 堆分配

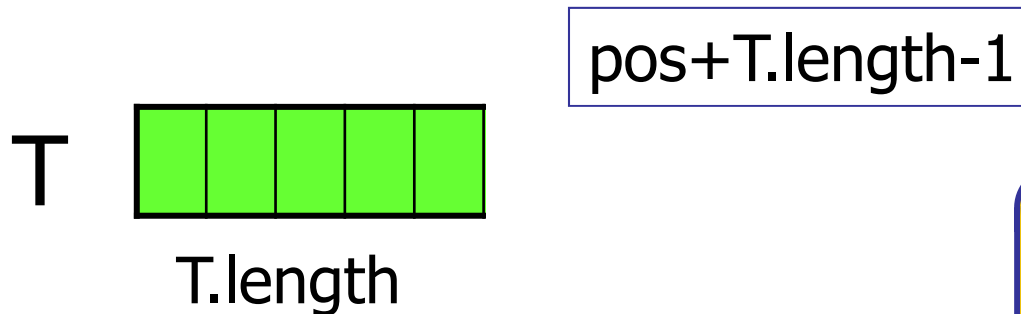
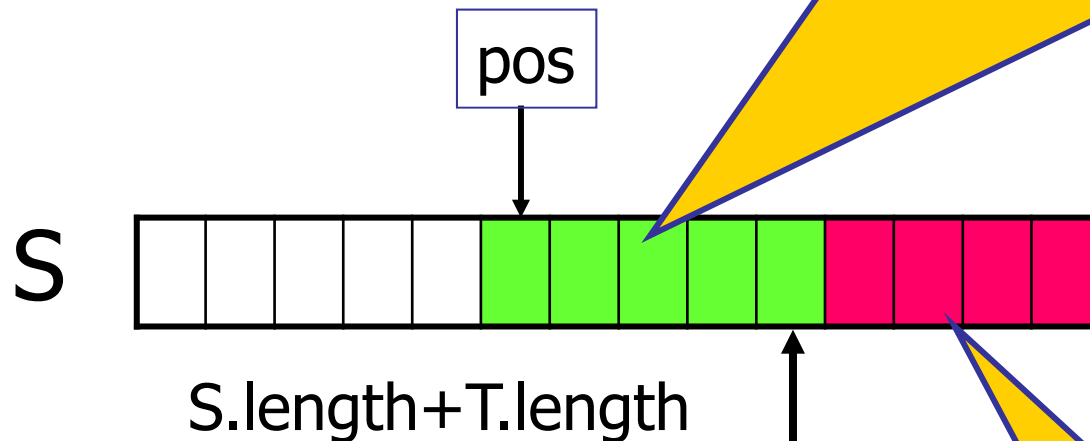
```
Status StringInsert(HString &S,int pos,HString T){
```



(9) 串插入-堆分配

Status StringInsert(H

$S.ch[pos-1..pos+T.length-2]=T.ch[0..T.length-1];$



for($i=S.length-1; i \geq pos-1; --i$)
 $S.ch[i+T.length]=S.ch[i];$


```
Status StringInsert(HString &S,int pos,HString T){
```

```
    if(pos<1||pos>S.length+1) return ERROR; //pos不合法
```

```
    if(T.length){ //T非空,则重新分配空间,插入T
```

```
        if(!(S.ch=(char *)realloc(S.ch,(S.length+T.length)*sizeof(char))))
```

```
            return ERROR;
```

```
        for(i=S.length-1;i>=pos-1;--i) //为插入T而腾出位置
```

```
            S.ch[i+T.length]=S.ch[i];
```

```
        S.ch[pos-1..pos+T.length-2]=T.ch[0..T.length-1]; //插入T
```

```
        S.length+=T.length;
```

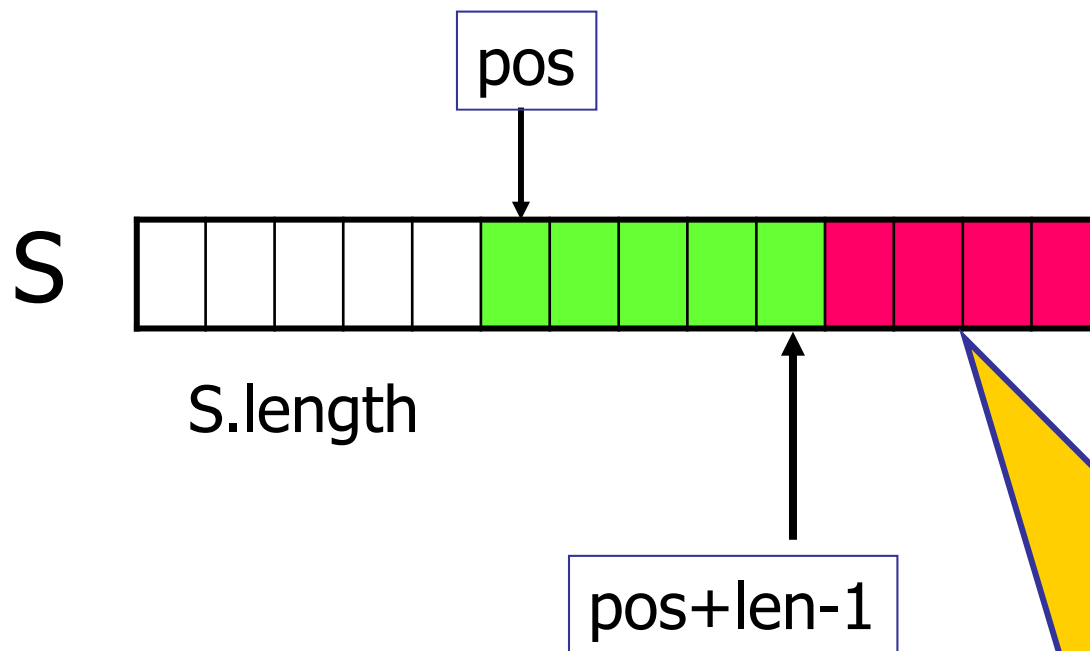
```
    }
```

```
    return OK;
```

```
}//StringInsert
```

(10) 串删除- 堆分配

Status StrDelete(HString &S,int pos,int len){



```
for(i= pos+len-1;i< S.length; i++)  
S.ch[i-len]=S.ch[i];
```



(10) 串删除- 堆分配

```
Status StrDelete(HString &S,int pos,int len)
{ // 从串S中删除第pos个字符起长度为len的子串
  if(pos+len-1> S.length)
    return ERROR;
  for(i=pos+len-1;i<S.length;i++)
    S.ch[i-len]=S.ch[i];
  S.length-=len;
  S.ch=(char*)realloc(S.ch,S.length*sizeof(char));
  return OK;
}
```



实验5：病毒感染检测



带有遗传讯息的DNA片段称为基因。
基因在染色体上的位置称为**座位**，
每个基因都有自己特定的座位。



基因查找

- ✓ 如何确定某人是否含有某种特殊的基因？
- ✓ 类似的，如何确定某人否感染了某病毒（基因/DNA）？

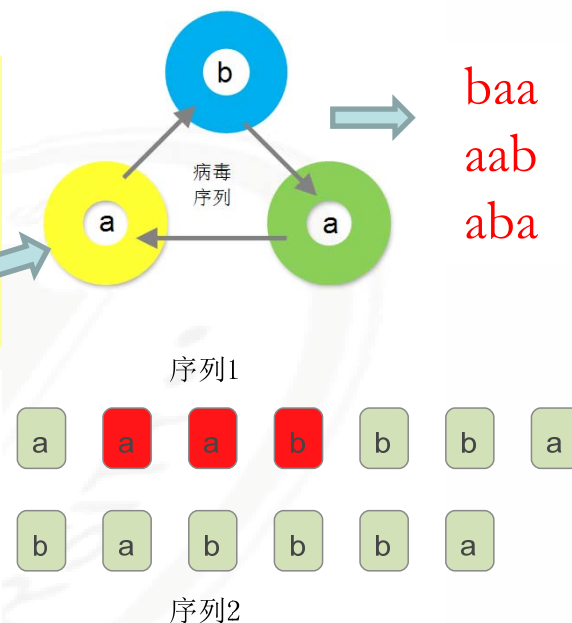




实验5：病毒感染检测

研究者将人的DNA和病毒DNA均表示成由一些字母组成的字符串序列，然后检测某种病毒DNA序列是否在患者的DNA序列中出现过：如果出现过，则此人感染了该病毒，否则没有感染。

例如，假设病毒的DNA序列为**baa**，患者1的DNA序列为a**a**bbba，则感染，患者2的DNA序列为babbbba，则未感染。注意，人的DNA序列是线性的，而病毒的DNA序列是**环状的**）





实验5：病毒感染检测



输入文件：num+1行

输出文件：num行

num

具体待检测任务

病毒感染检测输入数据.txt - 记事本		
文件(F) 编辑(E) 格式(O) 查看(V)		
10		
baa	bbaabbba	
baa	aaabbbba	
aabb	abceaabb	
aabb	abaabcea	
abcd	cdabbbab	
abcd	cabbbbab	
abcde	bcdedbda	
acc	bdedbda	
cde	cdcdcdec	
cced	cdccdcce	
病毒DNA	人的DNA	

病毒感染检测输出结果.txt - 记事本		
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)		
baa	bbaabbba	YES
baa	aaabbbba	YES
aabb	abceaabb	YES
aabb	abaabcea	YES
abcd	cdabbbab	YES
abcd	cabbbbab	NO
abcde	bcdedbda	NO
acc	bdedbda	NO
cde	cdcdcdec	YES
cced	cdccdcce	YES
病毒DNA	人的DNA	检测结果

✓ 要处理的对象是字符串，将病毒的DNA序列看作子串，人的DNA序列看作主串，检测任务的实质是：看子串是否在主串出现过。

✓ 字符串模式匹配算法（chap 4.3），该案例分析（chap 4.6）。



串的模式匹配算法

算法目的:

确定主串中所含子串第一次出现的位置（定位）

int Index(S, T, pos)

算法种类:

- **BF算法**（又称古典的、经典的、朴素的、穷举的）
- **KMP算法**（特点：速度快，复杂）



串的模式匹配算法

子串的定位操作:

确定主串中所含子串第一次出现的位置 (定位)

int Index(S, T, pos)

正文串 **$S = s_1 s_2 \cdots s_n$**

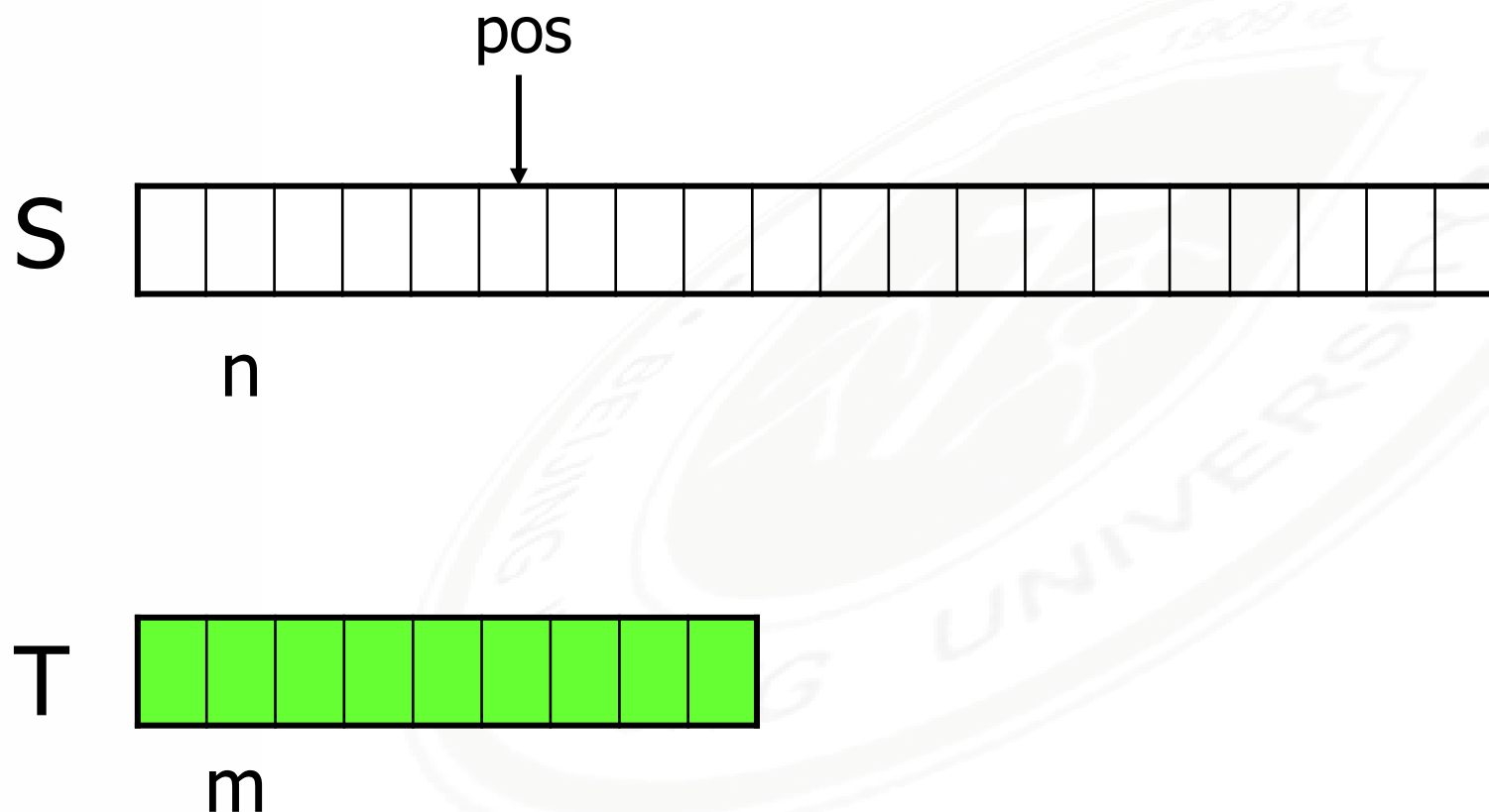
模式串 **$T = p_1 p_2 \cdots p_m$**

$$s_i s_{i+1} \cdots s_{i+m-1} = p_1 p_2 \cdots p_m$$



例：Index操作的实现

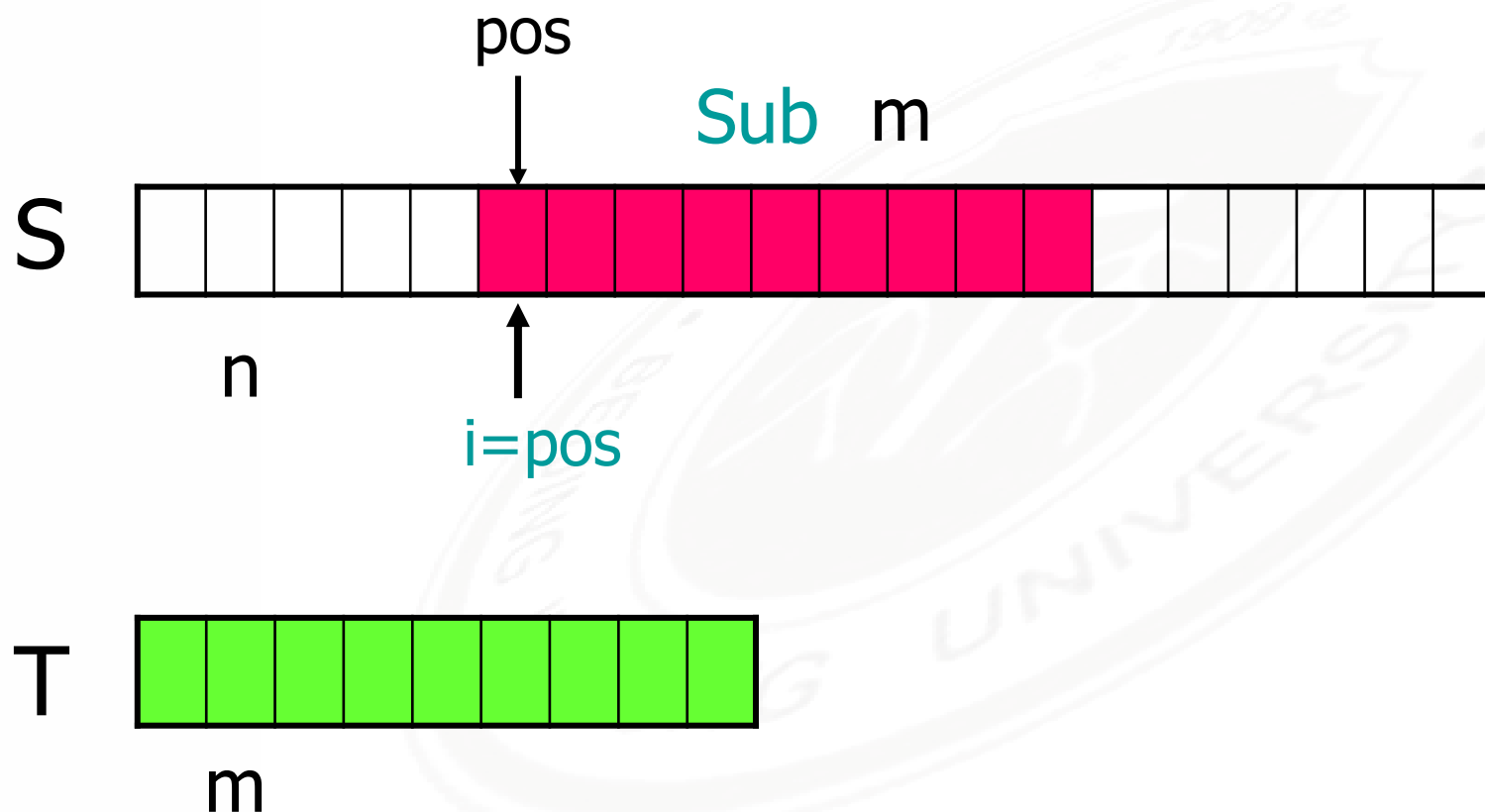
```
int Index(String S, String T, int pos){
```





例：Index操作的实现

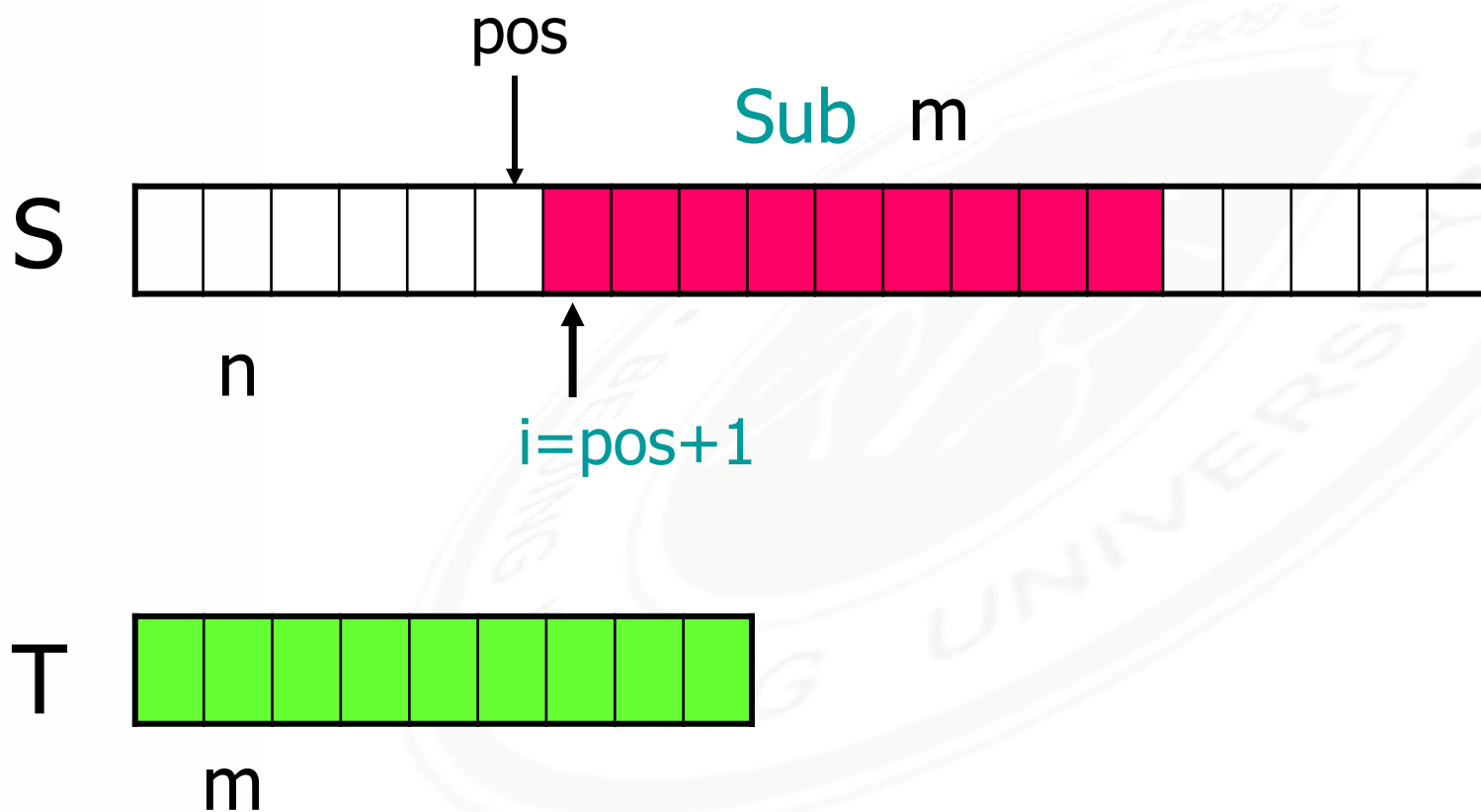
```
int Index(String S, String T, int pos){
```





例：Index操作的实现

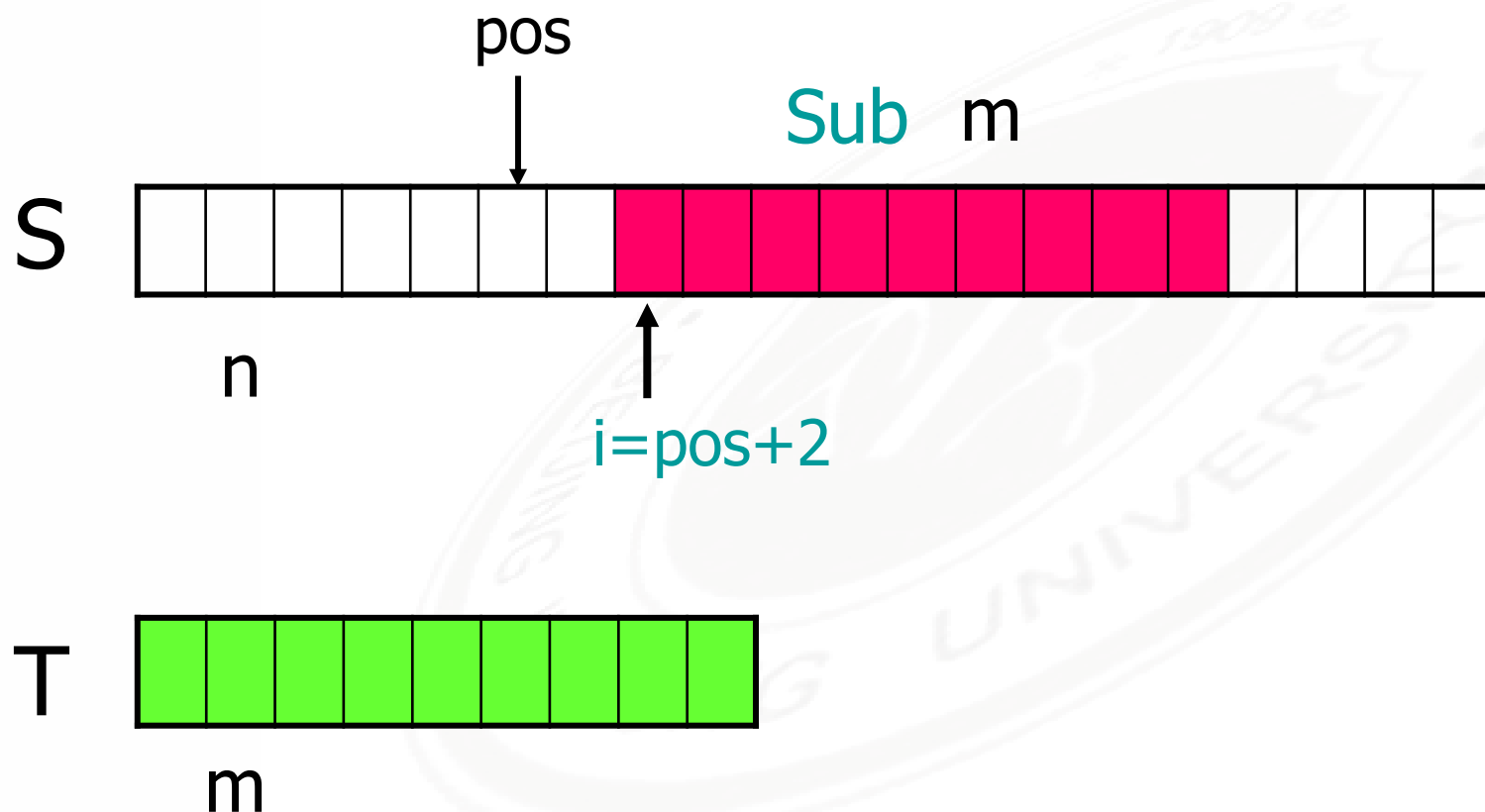
```
int Index(String S, String T, int pos){
```





例：Index操作的实现

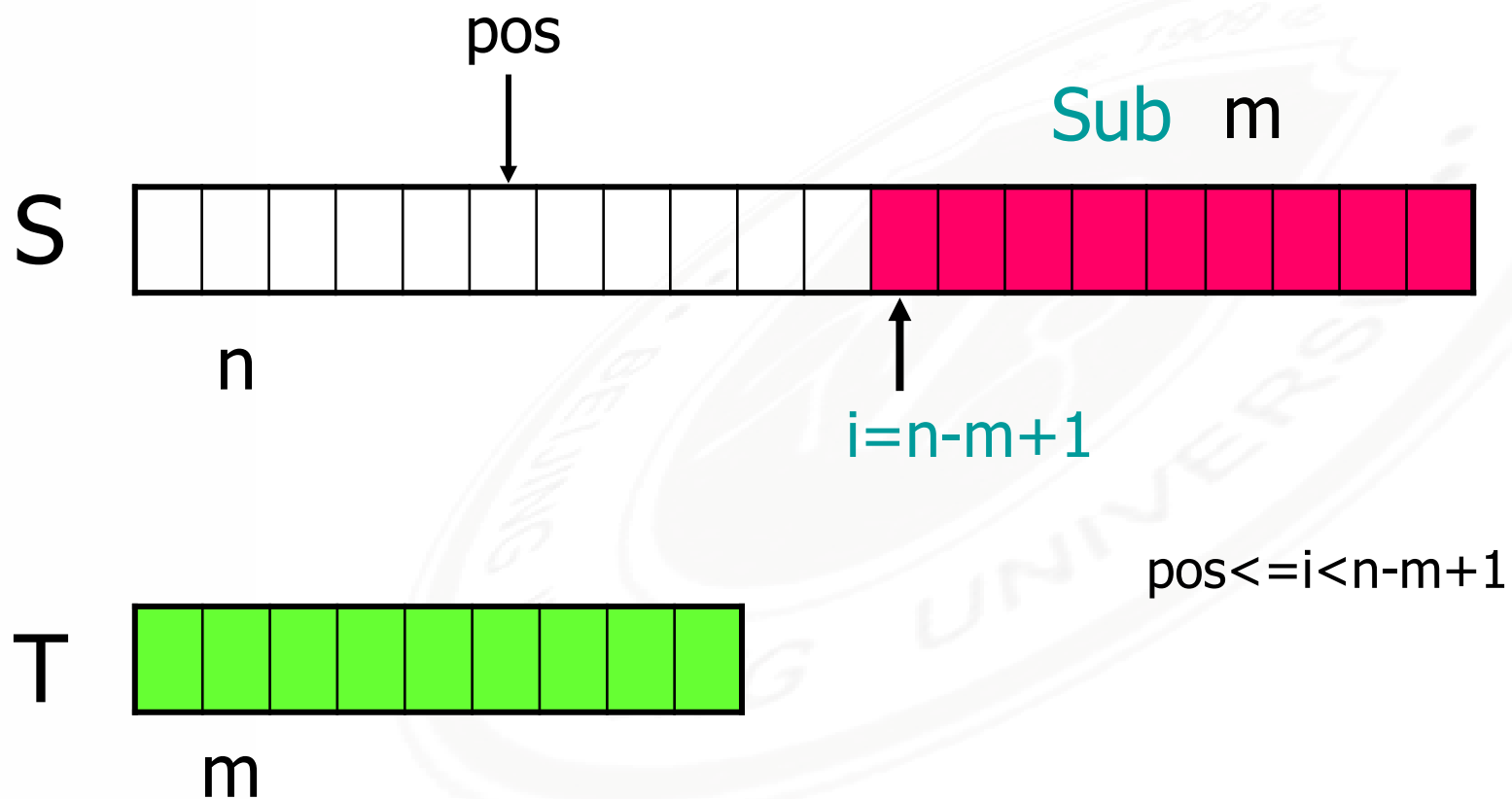
```
int Index(String S, String T, int pos){
```





例：Index操作的实现

```
int Index(String S, String T, int pos){
```





例：Index操作的实现

```
int Index(String S, String T, int pos){  
    if(pos>0){  
        n=StringLength(S); m=StringLength(T); i=pos;  
        while(i<=n-m+1){  
            SubString(sub,S,i,m);  
            if(StrCompare(sub,T)!=0) ++i;  
            else return i; //返回子串在主串中的位置  
        }//while  
    }//if  
    return 0; // S中不存在与T相等的子串  
}//Index
```



北京交通大学
BEIJING JIAOTONG UNIVERSITY

↓ $i=1$

1 ababcabcacbab

abcac

↑ $j=1$





北京交通大学
BEIJING JIAOTONG UNIVERSITY

↓ $i=2$

1 **ab**abcabcacbab

abcac

↑ $j=2$





北京交通大学
BEIJING JIAOTONG UNIVERSITY

↓ $i=3$

1 **ab**abcabcacbab

abcac

↑ $j=3$





北京交通大学
BEIJING JIAOTONG UNIVERSITY

↓ $i=3$
1 **ab**abcabcacbab
abcac
↑ $j=3$

↓ $i=2$
2 **ab**abcabcacbab
abcac
↑ $j=1$



↓ $i=3$
1 **ab**abcabcacbab
abcac
↑ $j=3$

↓ $i=2$
2 **ab**abcabcacbab
abcac
↑ $j=1$

↓ $i=3$
3 **ab**abcabcacbab
abcac
↑ $j=1$



↓ $i=3$
1 **a****b**abcabcacbab
a**b**cac
↑ $j=3$

↓ $i=2$
2 **a****b**abcabcacbab
a**b**cac
↑ $j=1$

↓ $i=4$
3 **a****b****a****b**cabcacbab
a**b**cac
↑ $j=2$



北京交通大学

BEIJING JIAOTONG UNIVERSITY

↓ $i=3$
1 **ab**abcabcacbab
abcac
↑ $j=3$

↓ $i=2$
2 **ab**abcabcacbab
abcac
↑ $j=1$

↓ $i=7$
3 **abab**ca**bc**acbab
abca**c**
↑ $j=5$



北京交通大学

BEIJING JIAOTONG UNIVERSITY

↓ $i=3$
1 ab**a**bcabcacbab
ab**a**cac
↑ $j=3$

↓ $i=2$
2 abab**a**bcabcacbab
ab**a**cac
↑ $j=1$

↓ $i=7$
3 abab**a**bc**a**bcacbab
ab**a**ca**c**
↑ $j=5$

↓ $i=4$
4 abab**a**bcabcacbab
ab**a**bc
↑ $j=1$



↓ $i=3$
1 ab**a**bcabcacbab
ab**a**cac
↑ $j=3$

↓ $i=2$
2 abab**a**bcabcacbab
ab**a**cac
↑ $j=1$

↓ $i=7$
3 abab**a**bc**a**bcacbab
ab**a**cac
↑ $j=5$

↓ $i=4$
4 abab**a**bcabcacbab
ab**a**bc
↑ $j=1$

↓ $i=5$
5 abab**a**bcabcacbab
ab**a**cac
↑ $j=1$



↓ $i=3$
1 ab**a**bcabcacbab
ab**a**cac
↑ $j=3$

↓ $i=2$
2 abab**a**bcabcacbab
ab**a**cac
↑ $j=1$

↓ $i=7$
3 abab**a**bc**a**bcacbab
ab**a**ca**c**
↑ $j=5$

↓ $i=4$
4 abab**a**bcabcacbab
ab**a**bc
↑ $j=1$

↓ $i=5$
5 abab**a**bcabcacbab
ab**a**ca**c**
↑ $j=1$

↓ $i=6$
6 abab**a**bcabcacbab
ab**a**ca**c**
↑ $j=1$



↓ $i=3$
1 ab**a**bcabcacbab
ab**a**cac
↑ $j=3$

↓ $i=2$
2 abab**a**bcabcacbab
ab**a**cac
↑ $j=1$

↓ $i=7$
3 abab**a**bc**a**bcacbab
ab**a**ca**c**
↑ $j=5$

↓ $i=4$
4 abab**a**bcabcacbab
ab**a**bc
↑ $j=1$

↓ $i=5$
5 abab**a**bcabcacbab
ab**a**ca**c**
↑ $j=1$

↓ $i=6$
6 abab**a**bcabcacbab
ab**a**ca**c**
↑ $j=1$



北京交通大学

BEIJING JIAOTONG UNIVERSITY

↓ $i=3$
1 ab**a**bcabcacbab
ab**a**cac
↑ $j=3$

↓ $i=2$
2 abab**a**bcabcacbab
ab**a**cac
↑ $j=1$

↓ $i=7$
3 abab**a**bc**a**bcacbab
ab**a**cac
↑ $j=5$

↓ $i=4$
4 abab**a**bcabcacbab
ab**a**bc
↑ $j=1$

↓ $i=5$
5 abab**a**bcabcacbab
ab**a**cac
↑ $j=1$

↓ $i=11$
6 ababcb**a**bcacbab
ab**a**cac
↑ $j=6$



↓ $i=3$
1 ab**a**bcabcacbab
ab**a**cac
↑ $j=3$

↓ $i=2$
2 ab**a**bcabcacbab
a**a**bcac
↑ $j=1$

.....

↓ $i=7$ ↓ $i=4$
3 abab 每次从模式串的**第一个字母**开始比较，效率较低
ab
↑ $j=5$ ↑ $j=1$

.....

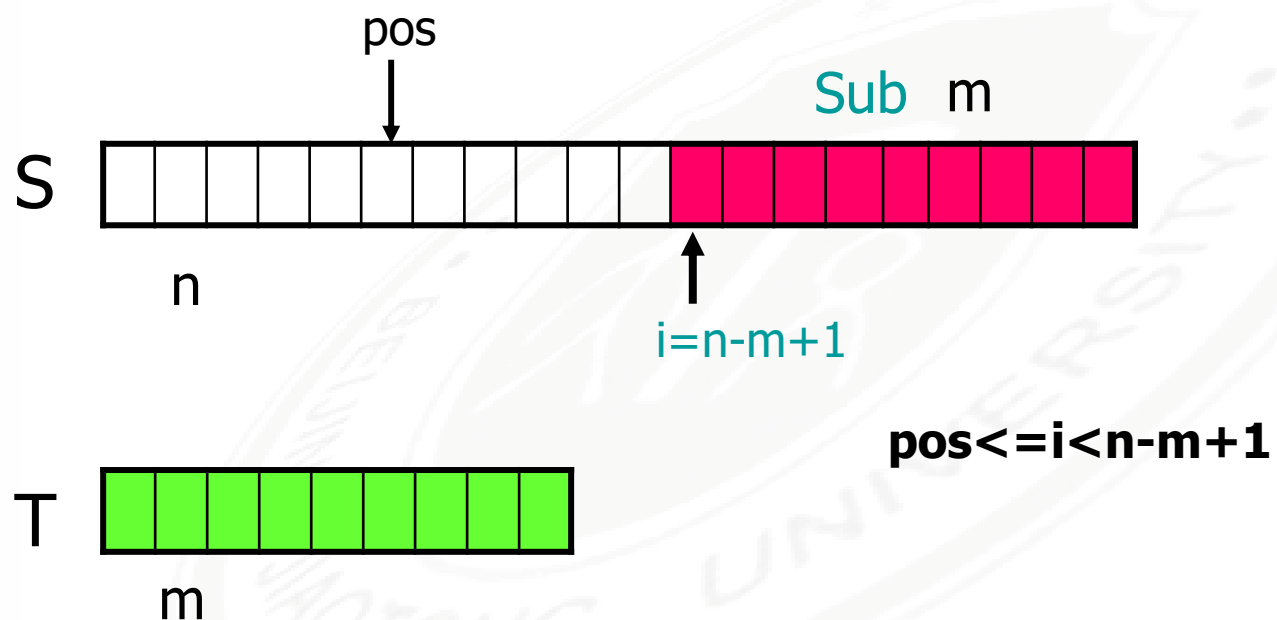
↓ $i=5$
5 abab**a**bcabcacbab
a**a**bcac
↑ $j=1$

↓ $i=11$
6 ababcb**a**bcacbab
ab**a**cac
↑ $j=6$



Index操作的实现2

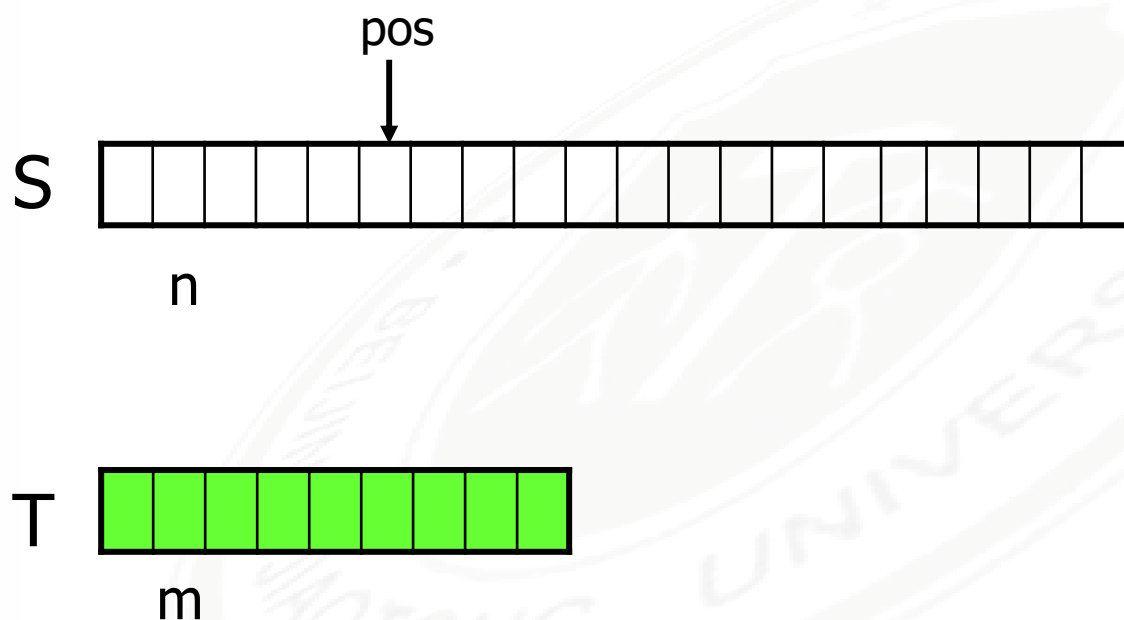
```
int Index(String S, String T, int pos){
```





Index操作的实现2

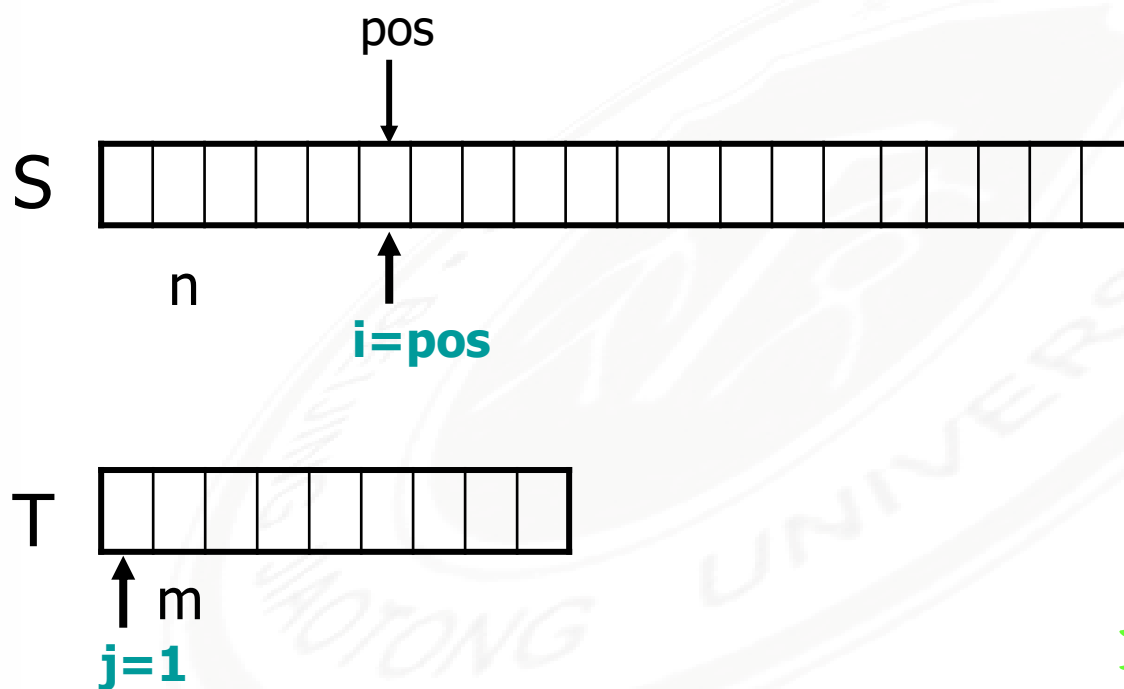
```
int Index(String S, String T, int pos){
```





Index操作的实现2

```
int Index(String S, String T, int pos){
```

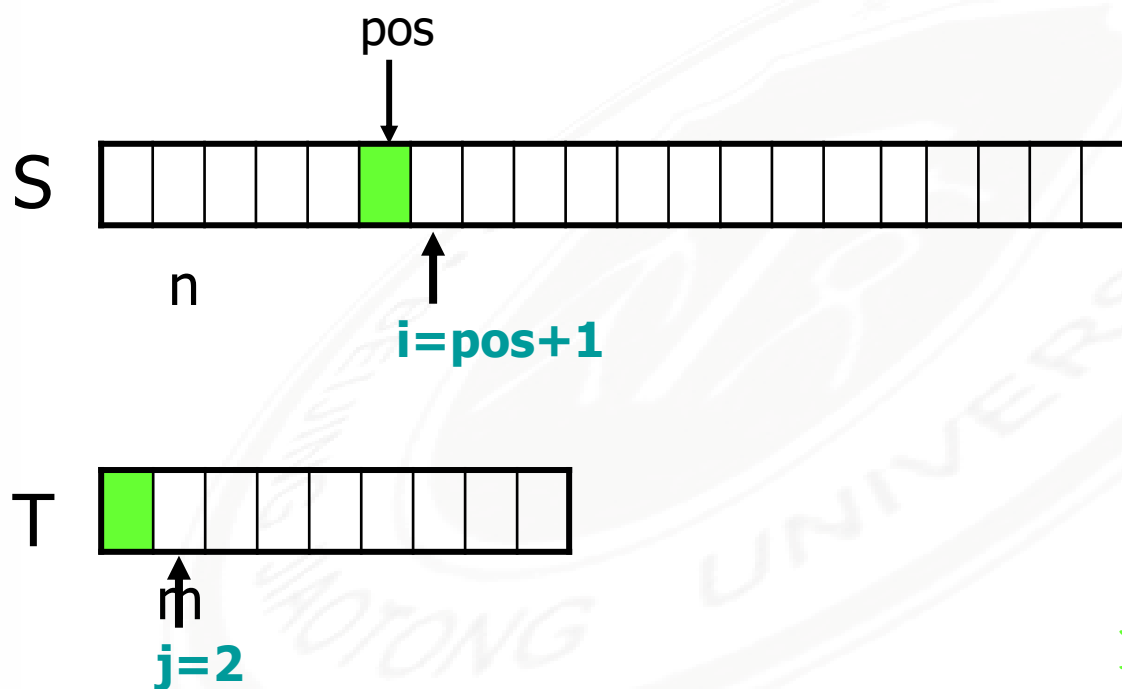


第 **1** 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

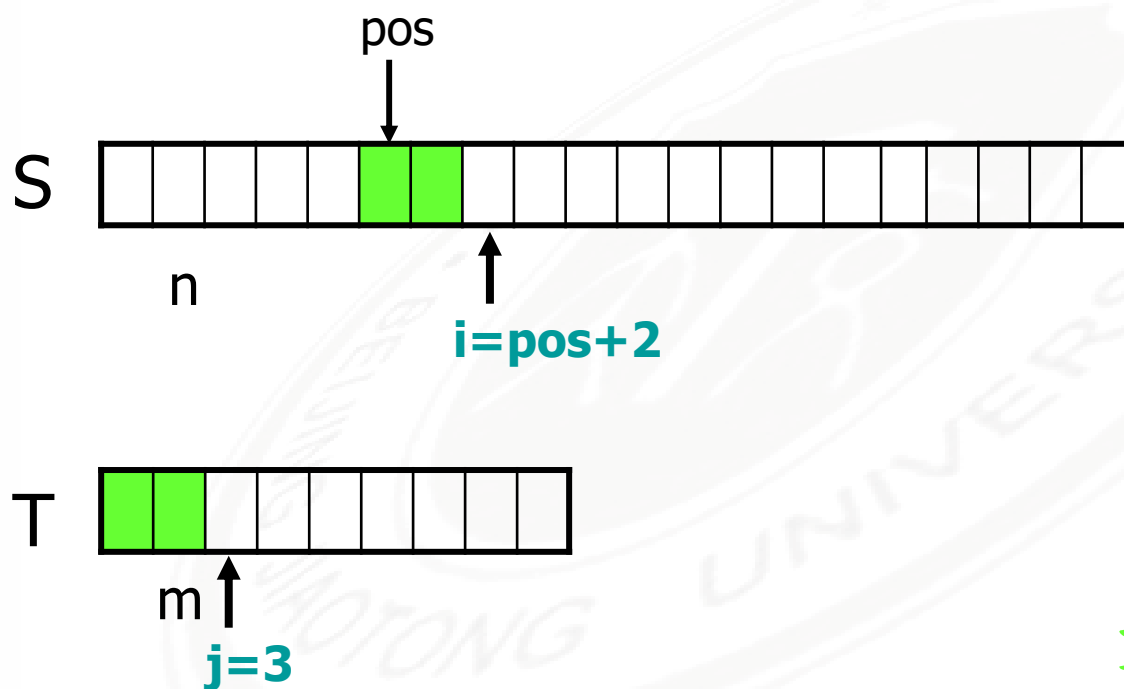


第 1 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

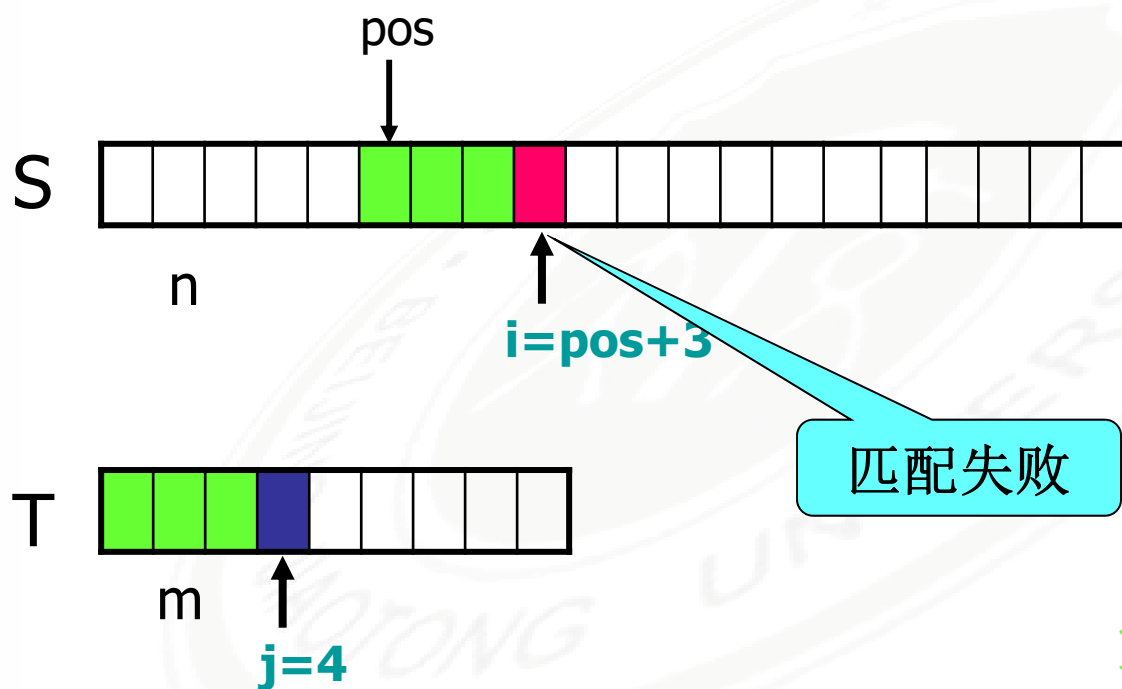


第 1 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

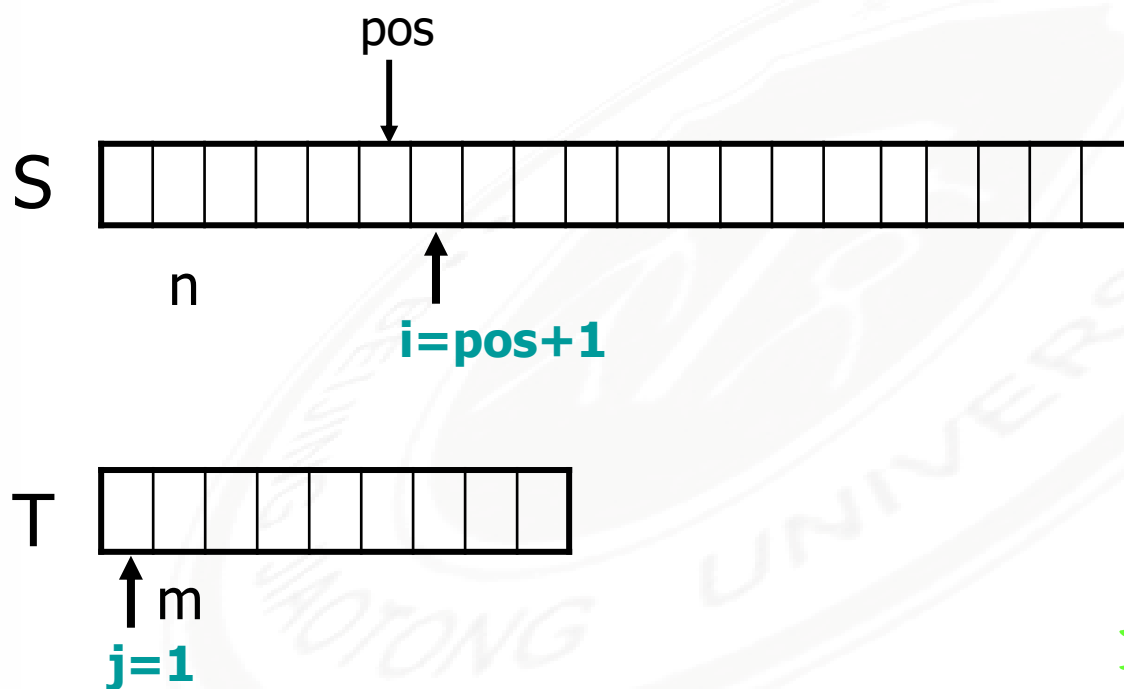


第 1 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

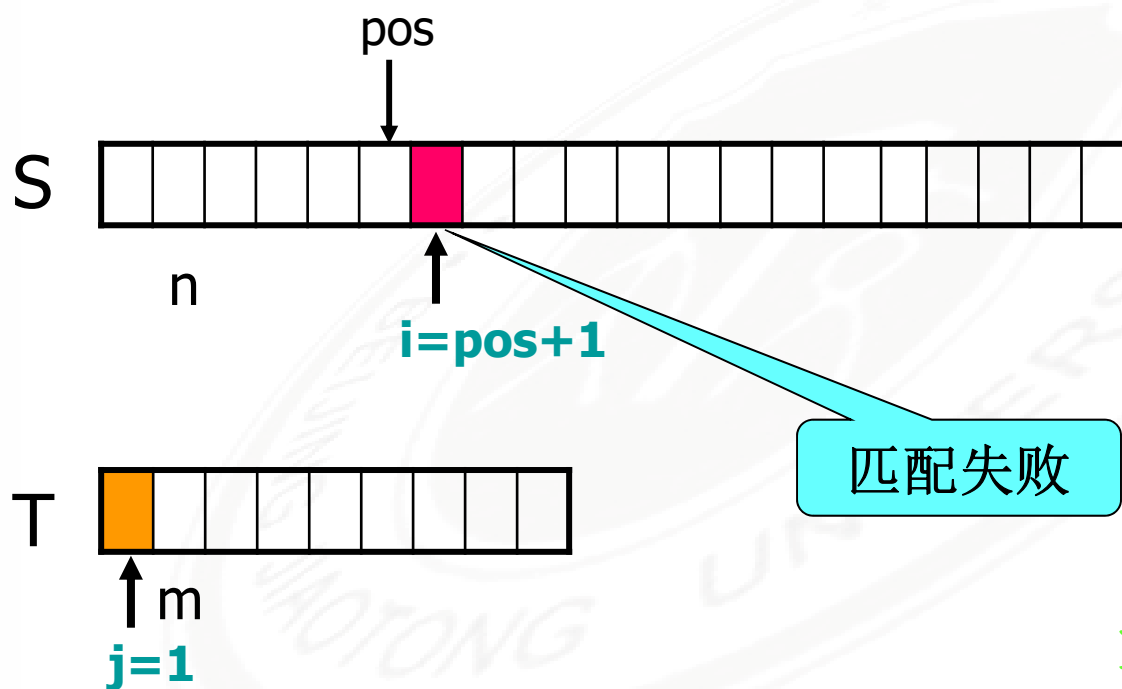


第 2 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

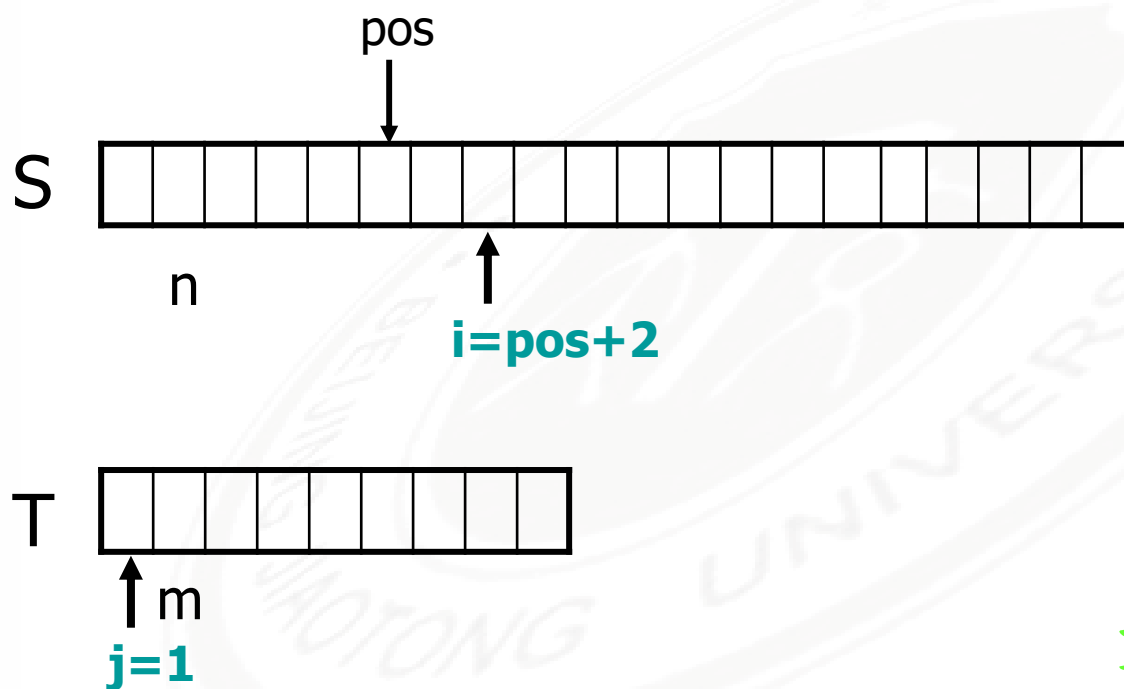


第 2 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

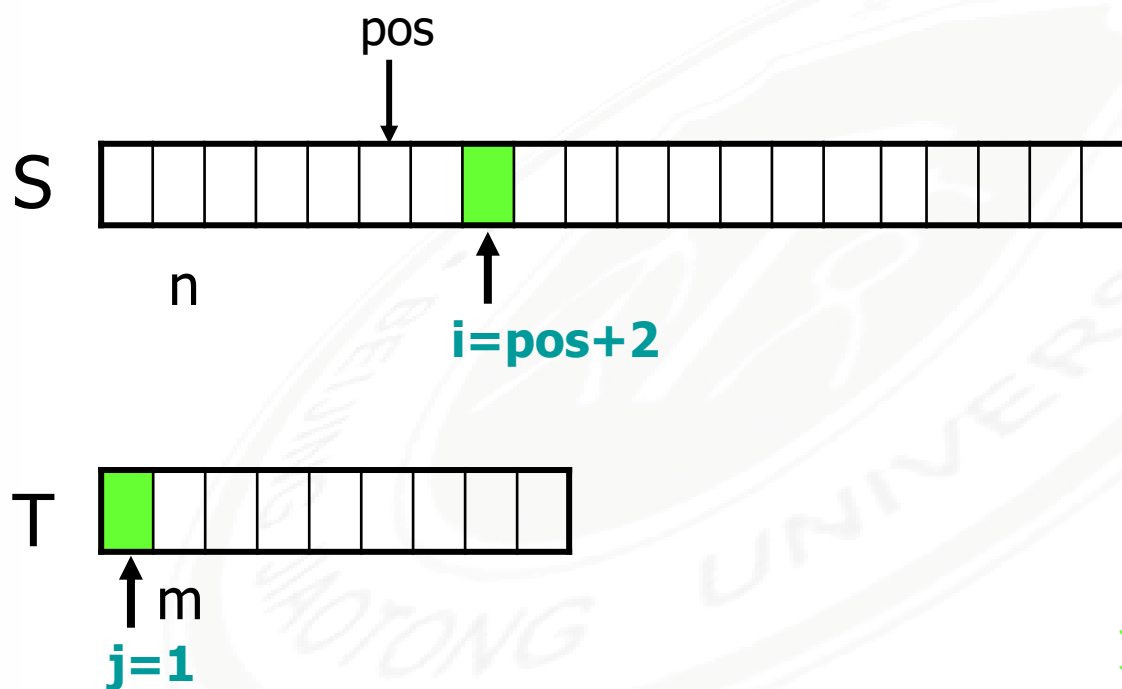


第 3 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

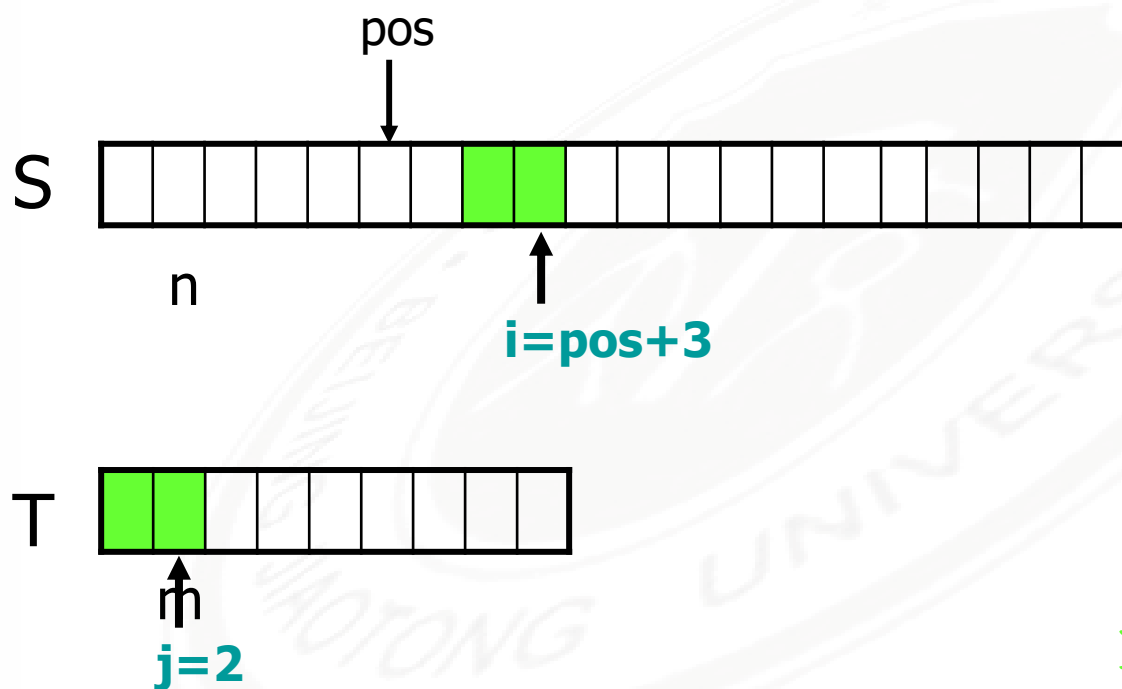


第 3 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

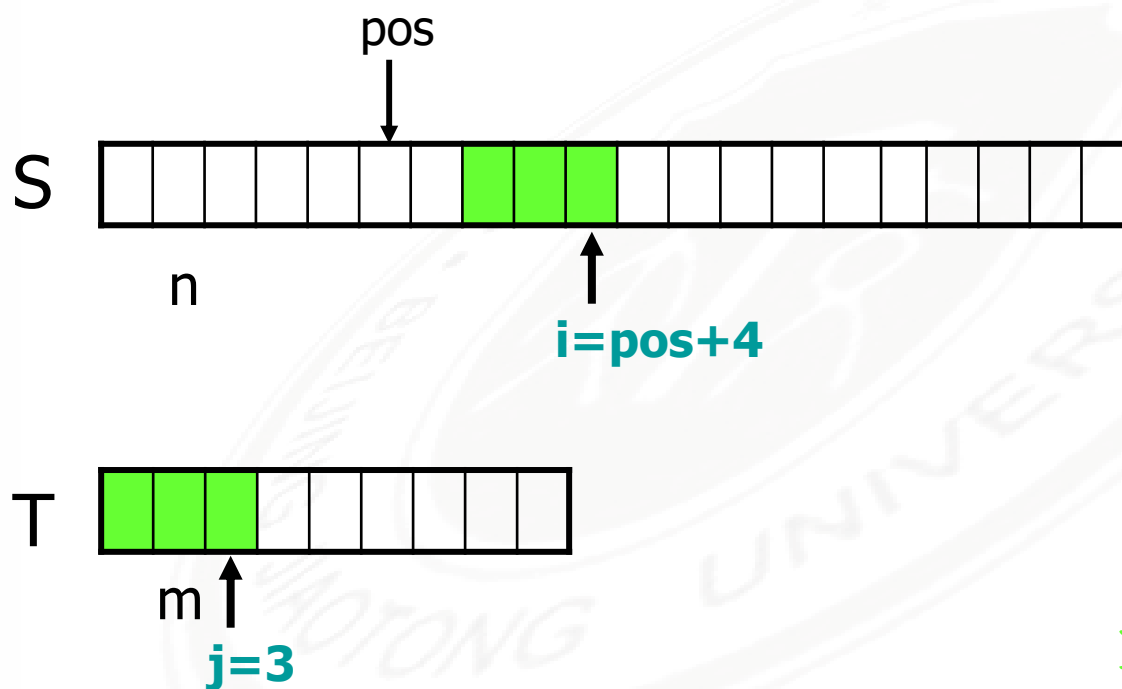


第 3 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```

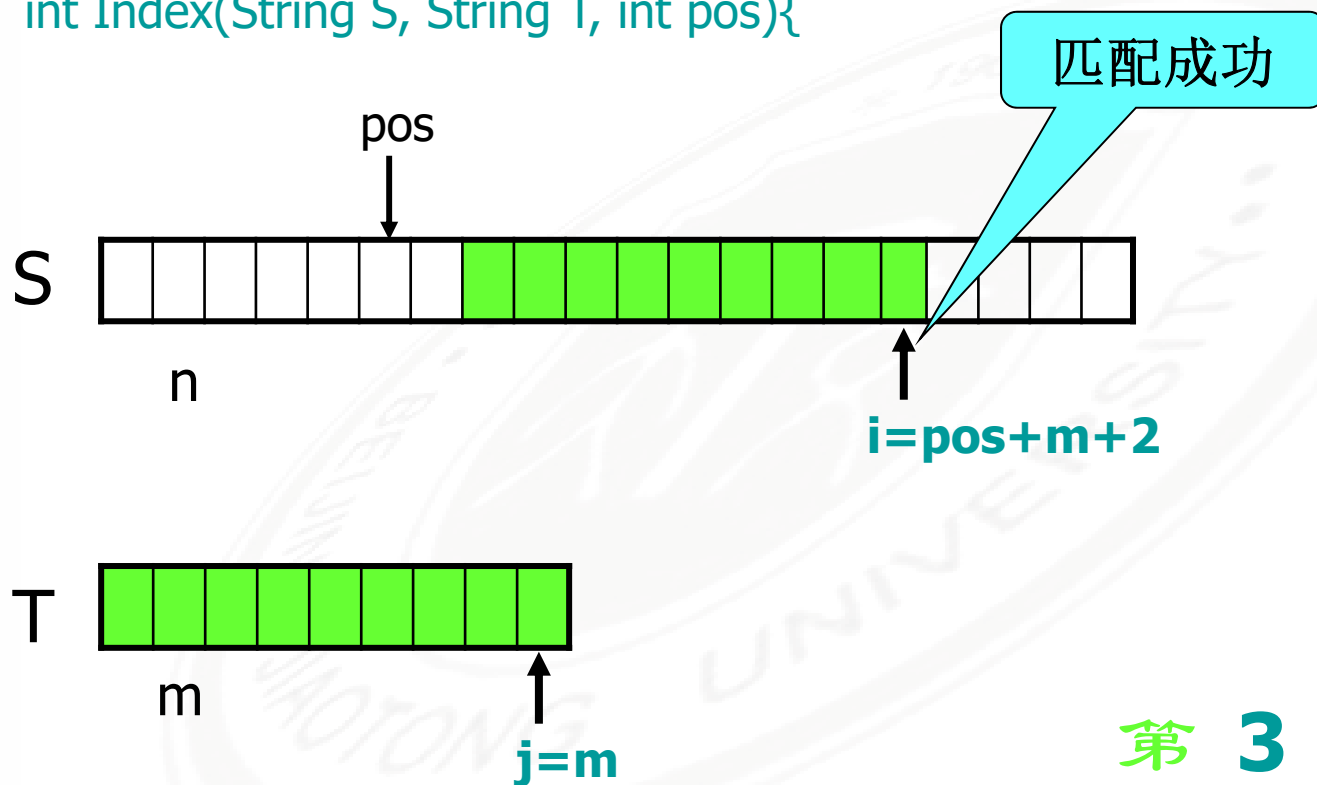


第 3 趟



Index操作的实现2

```
int Index(String S, String T, int pos){
```



第 3 趟



Index操作的实现2

```
int Index(SString S,SString T,int pos){
```

```
    if(pos>=1&&pos<=S[0]){
```

```
        k=i=pos; j=1; //i正文试配起点, j:模式指针, k:正文指针
```

```
        last=S[0]-T[0]+1; //last:正文最后一个起点
```

```
        while(k<=last && j<=T[0])
```

```
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符
```

```
            else{ i=++k; j=1; } //指针后退重新开始匹配
```

```
            if(j>T[0]) return i;
```

```
            else return 0;
```

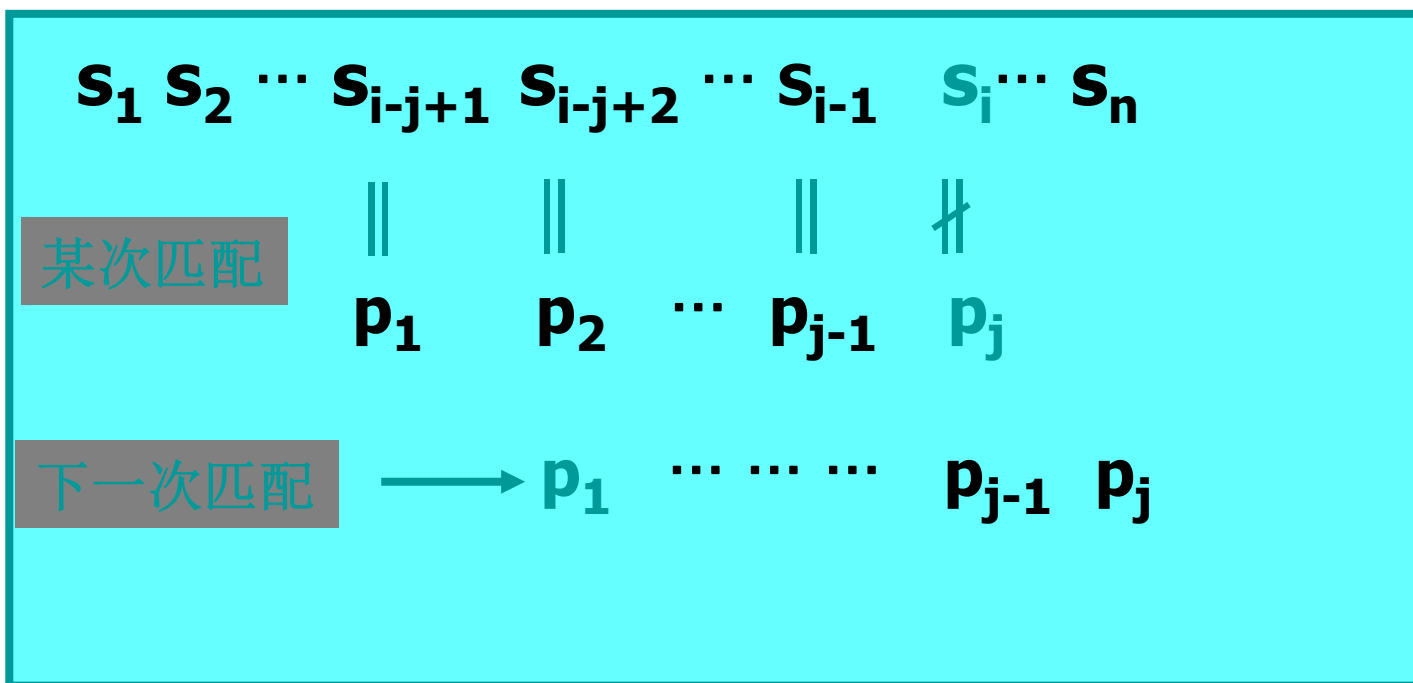
```
        }
```

```
    else return 0;
```

```
}
```



模式匹配的一般过程



$$i = i - j + 2, \quad j = 1$$



Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0]){  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
            if(j>T[0]) return i-T[0];  
            else return 0;  
        }  
    }  
    else return 0;  
}
```

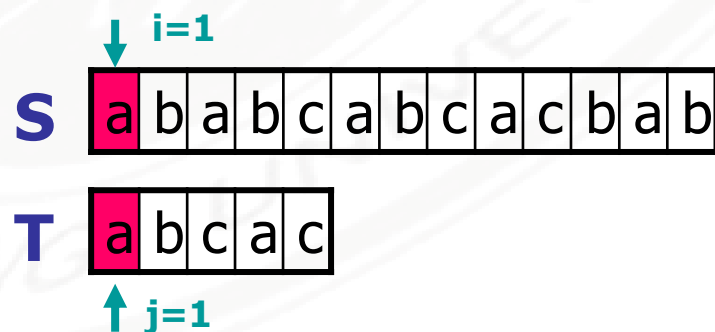
“蛮力” 算法



Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1;} // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 1 趟

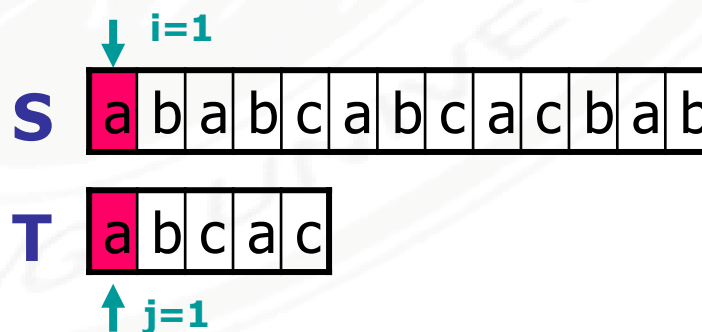




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0]){  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1;} // 指针后退重新开始匹配  
            if(j>T[0]) return i-T[0];  
            else return 0;  
        }  
    }  
    else return 0;  
}
```

第 1 趟

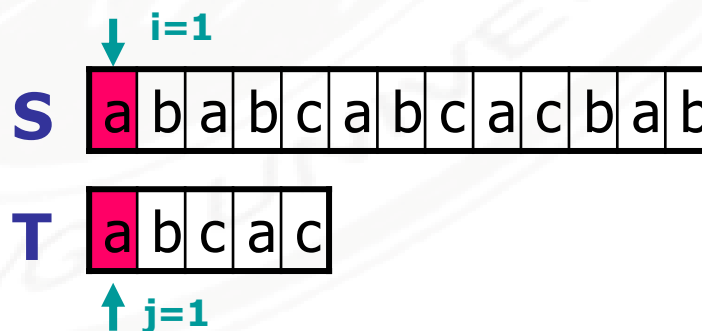




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1;} //指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 1 趟

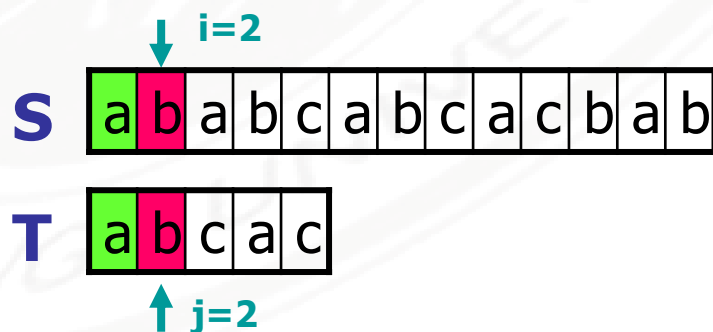




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 1 趟

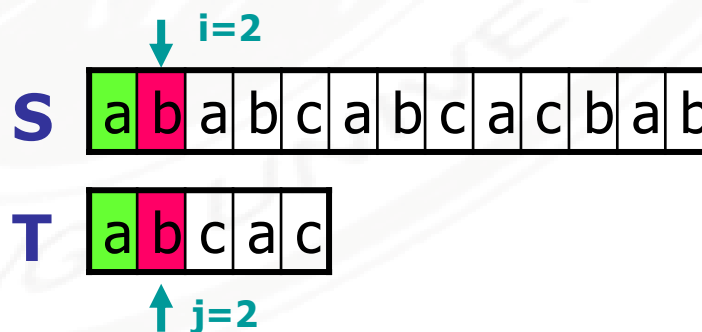




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 1 趟

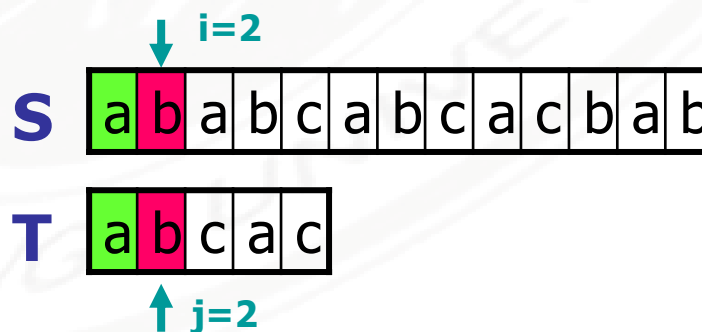




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 1 趟





Brute-Force 算法

```
int Index(SString S,SString T,int pos){
```

```
    if(pos>=1&&pos<=S[0]){
```

```
        i=pos; j=1;
```

```
        while(i<=S[0] && j<=T[0])
```

```
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符
```

```
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配
```

```
        if(j>T[0]) return i-T[0];
```

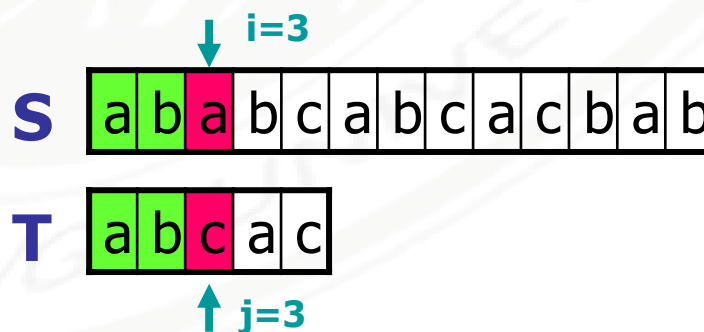
```
        else return 0;
```

```
    }
```

```
    else return 0;
```

```
}
```

第 1 趟





Brute-Force 算法

```
int Index(SString S,SString T,int pos){
```

```
    if(pos>1&&pos<=S[0]){
```

```
        i=pos; j=1;
```

```
        while(i<=S[0] && j<=T[0])
```

```
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符
```

```
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配
```

```
        if(j>T[0]) return i-T[0];
```

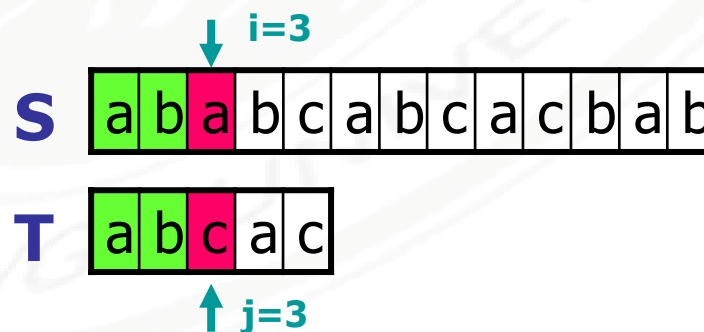
```
        else return 0;
```

```
    }
```

```
    else return 0;
```

```
}
```

第 1 趟





Brute-Force 算法

```
int Index(SString S,SString T,int pos){
```

```
    if(pos>=1&&pos<=S[0]){
```

```
        i=pos; j=1;
```

```
        while(i<=S[0] && j<=T[0])
```

```
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符
```

```
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配
```

```
        if(j>T[0]) return i-T[0];
```

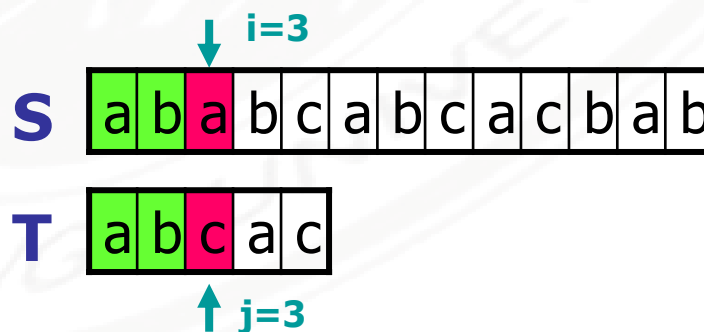
```
        else return 0;
```

```
    }
```

```
    else return 0;
```

```
}
```

第 1 趟

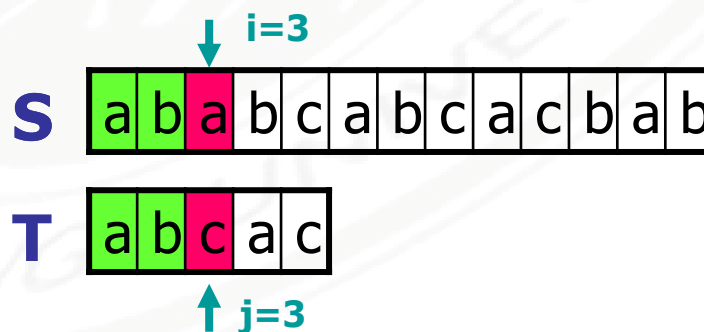




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 1 趟

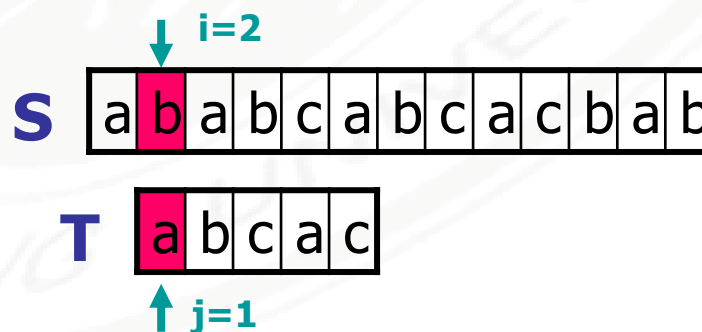




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1;} // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 2 趟

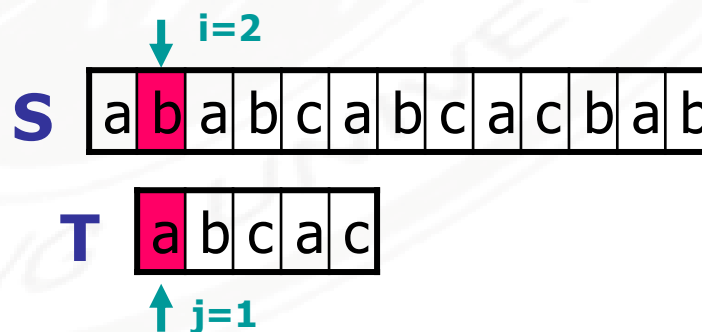




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0]){  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
            if(j>T[0]) return i-T[0];  
            else return 0;  
        }  
    }  
    else return 0;  
}
```

第 2 趟





Brute-Force 算法

```
int Index(SString S,SString T,int pos){
```

```
    if(pos>=1&&pos<=S[0]){
```

```
        i=pos; j=1;
```

```
        while(i<=S[0] && j<=T[0])
```

```
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符
```

```
            else{ i=i-j+2; j=1;} // 指针后退重新开始匹配
```

```
        if(j>T[0]) return i-T[0];
```

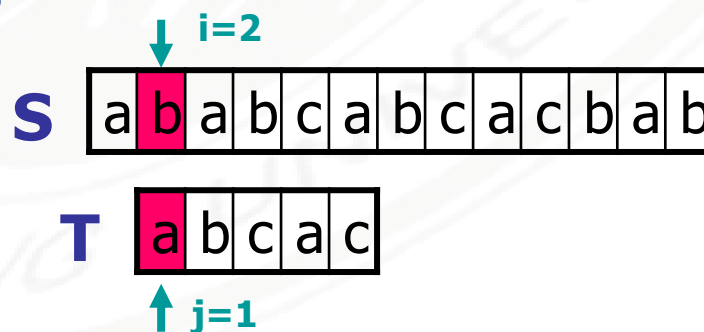
```
        else return 0;
```

```
    }
```

```
else return 0;
```

```
}
```

第 2 趟

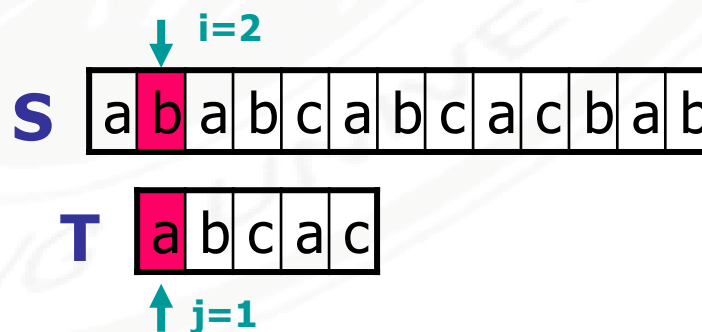




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 2 趟

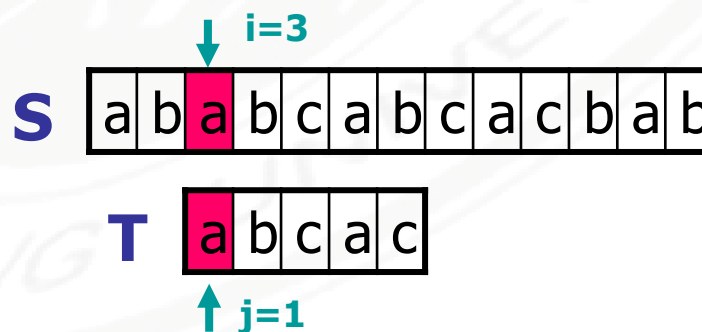




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else { i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 3 趟

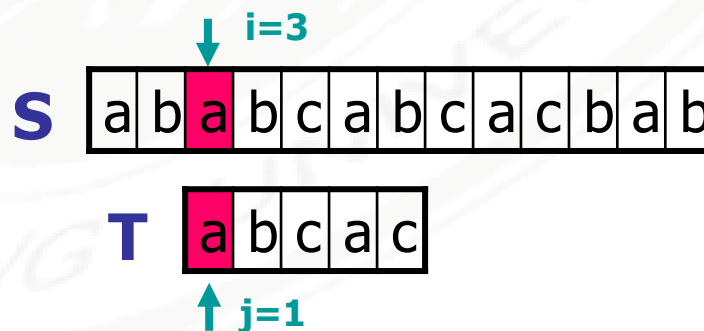




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 3 趟





Brute-Force 算法

```
int Index(SString S,SString T,int pos){
```

```
    if(pos>=1&&pos<=S[0]){
```

```
        i=pos; j=1;
```

```
        while(i<=S[0] && j<=T[0])
```

```
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符
```

```
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配
```

```
        if(j>T[0]) return i-T[0];
```

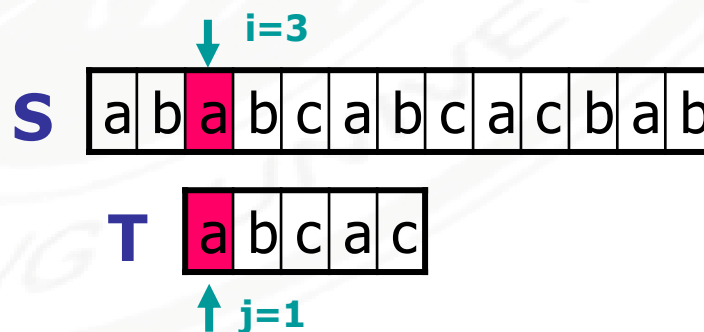
```
        else return 0;
```

```
    }
```

```
    else return 0;
```

```
}
```

第 3 趟

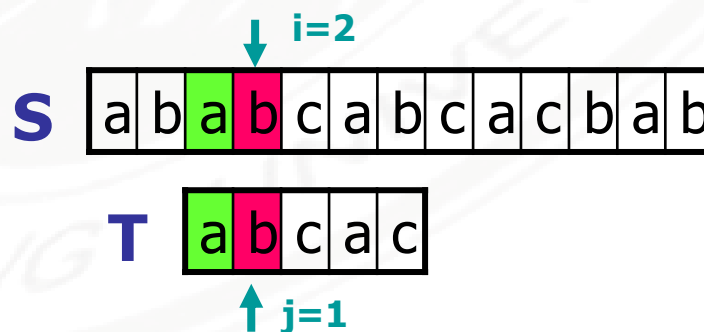




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 3 趟

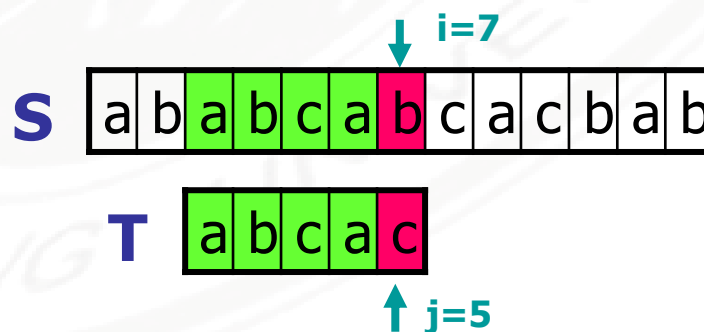




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 3 趟

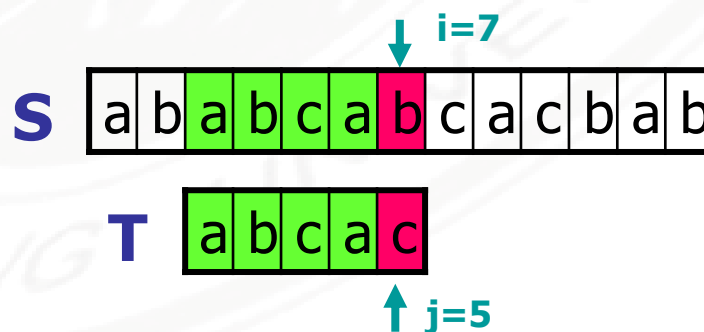




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0]){  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1;} // 指针后退重新开始匹配  
            if(j>T[0]) return i-T[0];  
        }  
        else return 0;  
    }  
}
```

第 3 趟

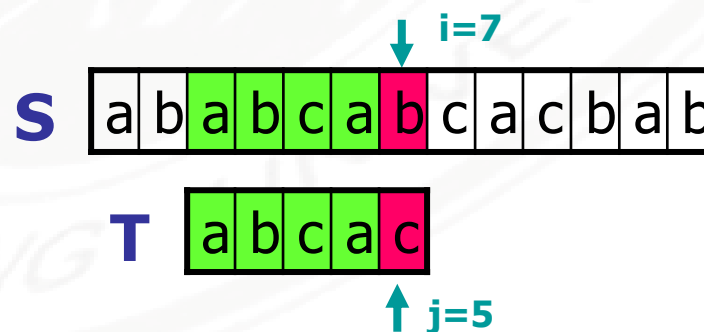




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 3 趟

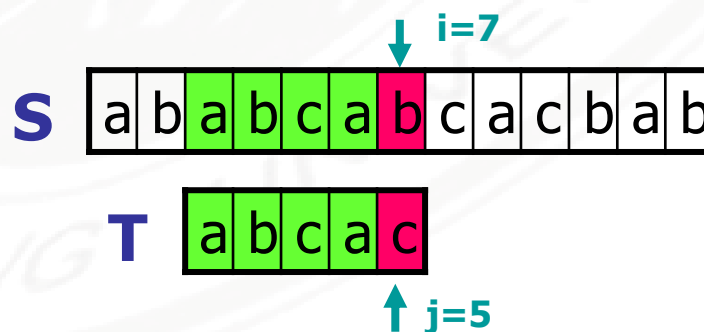




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 3 趟

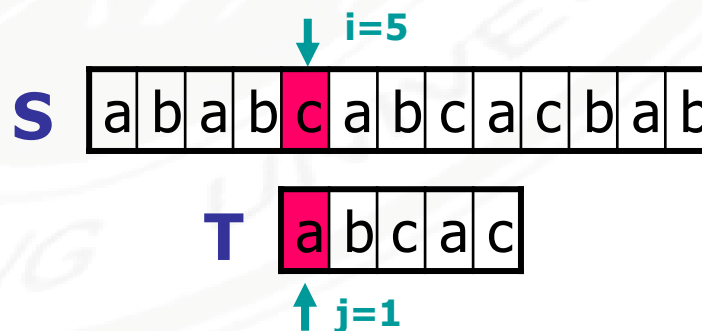




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else { i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 5 趟

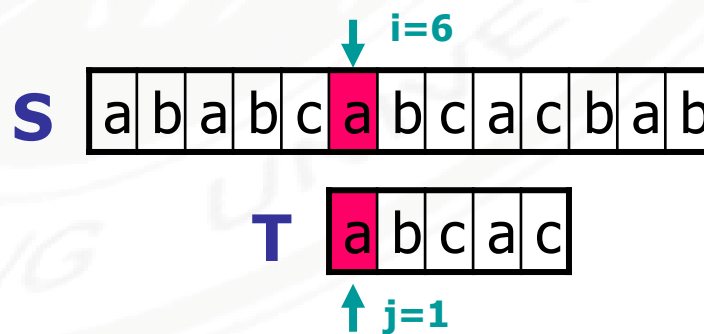




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0]){  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else { i=i-j+2; j=1; } // 指针后退重新开始匹配  
        }  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 6 趟

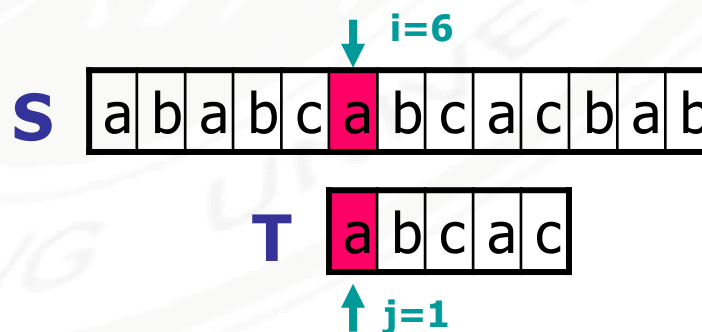




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0]){  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配  
            if(j>T[0]) return i-T[0];  
        }  
        else return 0;  
    }  
    else return 0;  
}
```

第 6 趟

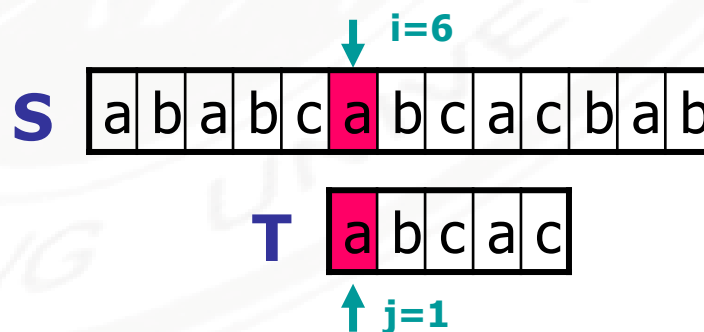




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 6 趟

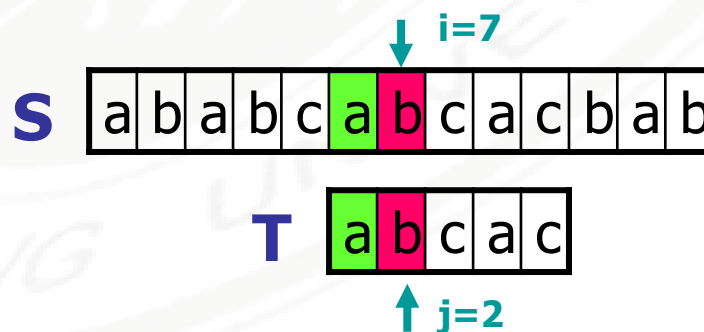




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 6 趟

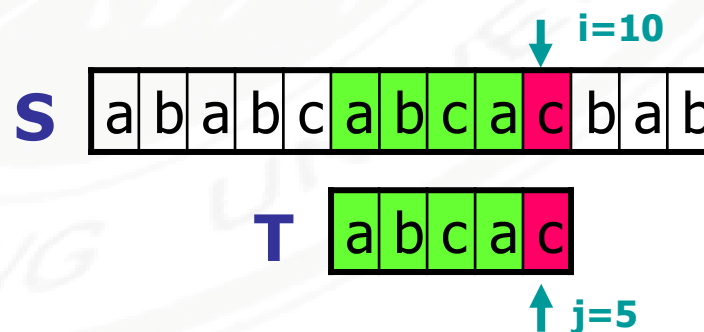




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1; } //指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 6 趟

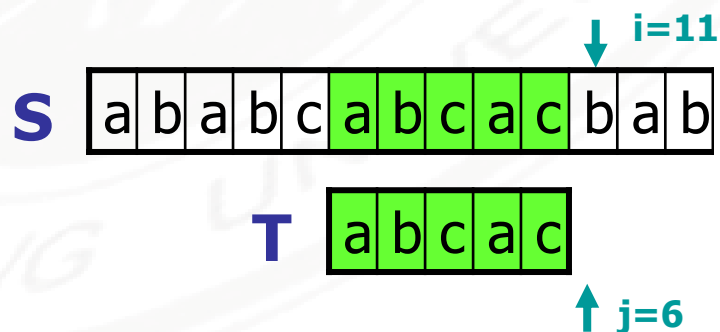




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j;} // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 6 趟

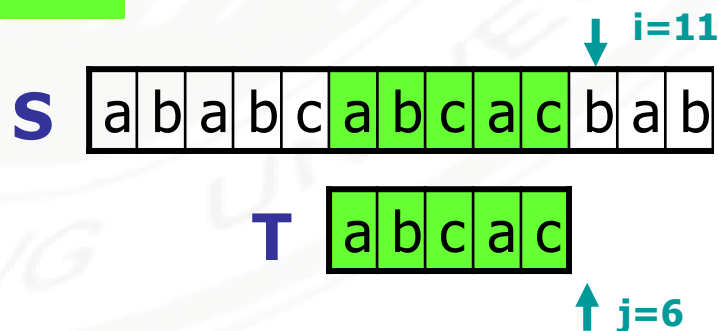




Brute-Force 算法

```
int Index(SString S,SString T,int pos){  
    if(pos>=1&&pos<=S[0]){  
        i=pos; j=1;  
        while(i<=S[0] && j<=T[0])  
            if(S[i]==T[j]) {++i; ++j; } // 继续比较后继字符  
            else{ i=i-j+2; j=1; } // 指针后退重新开始匹配  
        if(j>T[0]) return i-T[0];  
        else return 0;  
    }  
    else return 0;  
}
```

第 6 趟





1

$s_1 s_2 \cdots s_i \cdots s_m s_{m+1} \cdots s_{n-m+1} s_{n-m+1} \cdots$

s_n || $\#$

$p_1 p_2 \cdots p_i \cdots p_m$





北京交通大学

BEIJING JIAOTONG UNIVERSITY

1

$s_1 s_2 \cdots s_i \cdots s_m s_{m+1} \cdots s_{n-m+1} s_{n-m+1} \cdots s_n$

$\parallel \parallel \#$

$p_1 p_2 \cdots p_i \cdots p_m$

2

$s_1 s_2 \cdots s_i \cdots s_m s_{m+1} \cdots s_{n-m+1} s_{n-m+1} \cdots s_n$

$\parallel \parallel \#$

$p_1 p_2 \cdots p_i \cdots p_m$



北京交通大学

BEIJING JIAOTONG UNIVERSITY

1

$s_1 s_2 \cdots s_i \cdots s_m s_{m+1} \cdots s_{n-m+1} s_{n-m+1} \cdots s_n$

$\parallel \parallel \nparallel$

$p_1 p_2 \cdots p_i \cdots p_m$

2

$s_1 s_2 \cdots s_i \cdots s_m s_{m+1} \cdots s_{n-m+1} s_{n-m+1} \cdots s_n$

$\parallel \parallel \nparallel$

$p_1 p_2 \cdots p_i \cdots p_m$

n-m

$s_1 s_2 \cdots s_i \cdots s_m s_{m+1} \cdots s_{n-m+1} s_{n-m+1} \cdots$

$p_1 p_2 \cdots p_i \cdots p_m$

$O(m(n-m+1)) \rightarrow O(mn)$



北京交通大学
BEIJING JIAOTONG UNIVERSITY

课堂练习

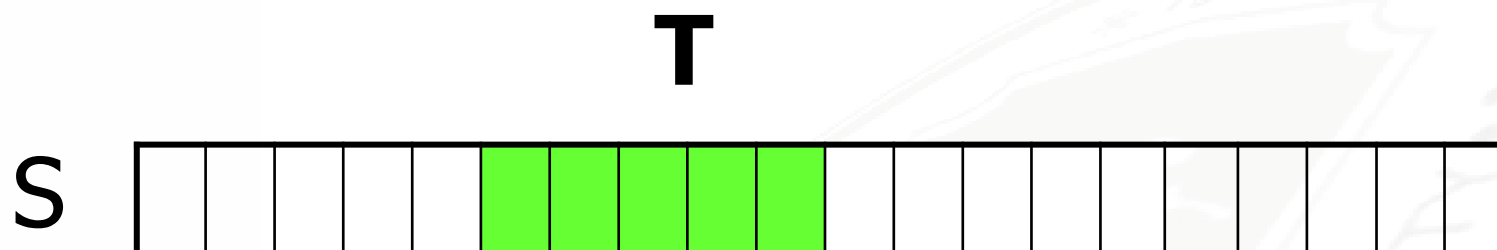


串替换操作的实现

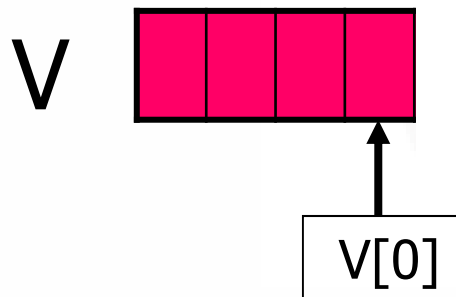




Status Replace(SString S, SString T, SString V)



S[0]

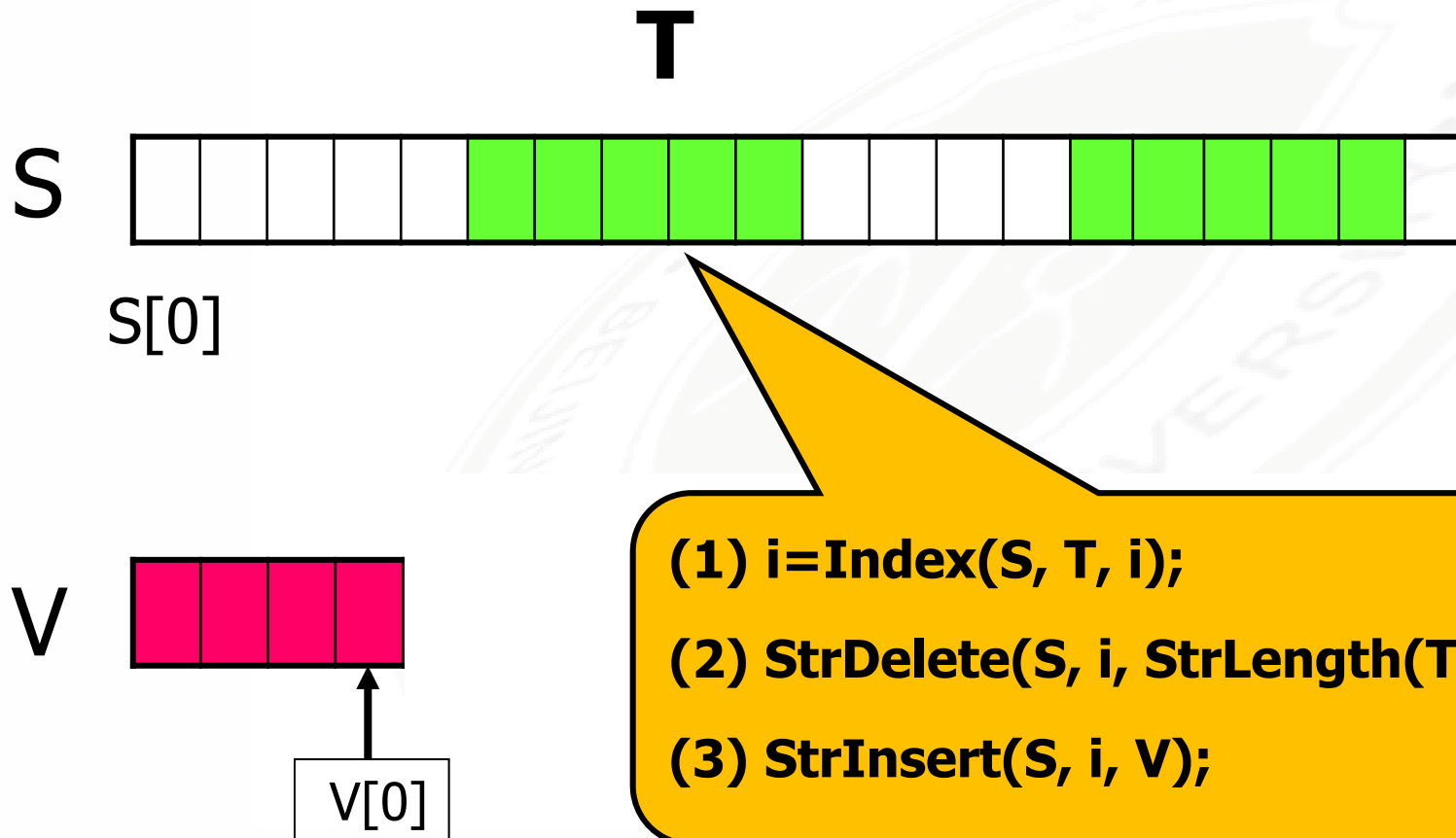


V[0]

- (1) $i = \text{Index}(S, T, i);$
- (2) $\text{StrDelete}(S, i, \text{StrLength}(T));$
- (3) $\text{StrInsert}(S, i, V);$



Status Replace(SString S, SString T, SString V)





Status Replace(SString S, SString T, SString V)

```
{ // 用V替换主串S中出现的所有与T相等的不重叠的子串
  int i=1; // 从串S的第一个字符起查找串T
  if(StrEmpty(T)) return ERROR; // T是空串
  do{ i=Index(S,T,i); // 结果i为从上一个i之后找到的子串T的位置
    if(i){ // 串S中存在串T
      StrDelete(S,i,StrLength(T)); // 删除该串T
      StrInsert(S,i,V); // 在原串T的位置插入串V
      i+=StrLength(V); // 在插入的串V后面继续查找串T
    }
  }while(i);
  return OK;
}
```