

第4章 串

4.1 串的基本概念

4.2 串的存储结构

4.3 串的模式匹配

4.1 串的基本概念

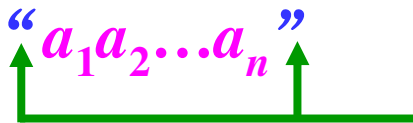
串（或字符串）是由零个或多个**字符**组成的**有限序列**。

串 \subset 线性表

串中所含字符的个数称为该**串的长度**（或串长），含零个字符的串称为空串，用 Φ 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）

串的基本运算如下：

- ① **StrAssign(&s,cstr)**: 将字符串常量cstr赋给串s，即生成其值等于cstr的串s。
- ② **StrCopy(&s,t)**: 串复制。将串t赋给串s。
- ③ **StrEqual(s,t)**: 判串相等。若两个串s与t相等则返回真；否则返回假。
- ④ **StrLength(s)**: 求串长。返回串s中字符个数。
- ⑤ **Concat(s,t)**: 串连接:返回由两个串s和t连接在一起形成的新串。
- ⑥ **SubStr(s,i,j)**: 求子串。返回串s中从第 i ($1 \leq i \leq n$) 个字符开始的、由连续 j 个字符组成的子串。

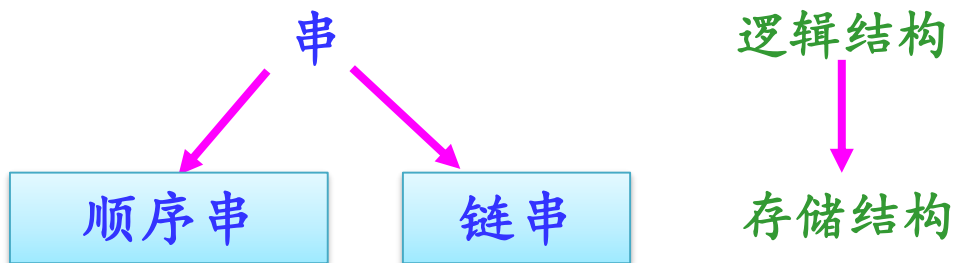
- ⑦ **InsStr(s1,i,s2):** 插入。将串s2插入到串s1的第 i ($1 \leq i \leq n+1$) 个字符中, 即将s2的第一个字符作为s1的第 i 个字符, 并返回产生的新串。
- ⑧ **DelStr(s,i,j):** 删除。从串s中删去从第 i ($1 \leq i \leq n$) 个字符开始的长度为 j 的子串, 并返回产生的新串。
- ⑨ **RepStr(s,i,j,t):** 替换。在串s中, 将第 i ($1 \leq i \leq n$) 个字符开始的 j 个字符构成的子串用串t替换, 并返回产生的新串。
- ⑩ **DispStr(s):** 串输出。输出串s的所有元素值。

思考题

串和线性表有什么异同？

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相同的存储结构。



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 MaxSize 100
```

```
typedef struct
```

```
{   char data[MaxSize];
```

```
    int length;
```

```
} SqString;
```

用来存储字符串

用来存储字符串长度

顺序串中实现串的基本运算与顺序表的基本运算类似。详细算法实现参见教材。

【例4-1】 设计顺序串上实现串比较运算Strcmp(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 Strcmp(SqString s, SqString 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 //s<t
        return -1;
}
```

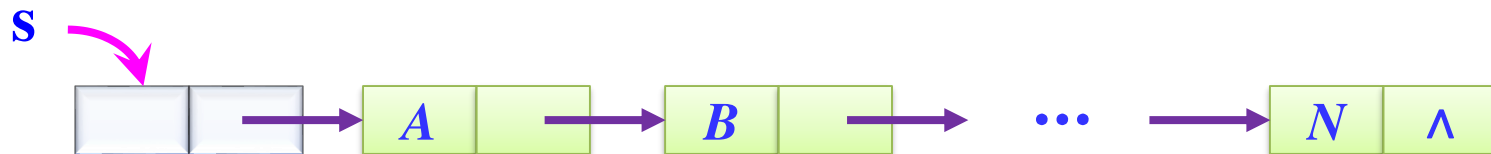
4.2.2 串的链式存储及其基本操作实现

链串的组织形式与一般的链表类似。

链串中的一个节点可以存储多个字符。通常将链串中每个节点所存储的字符个数称为节点大小。



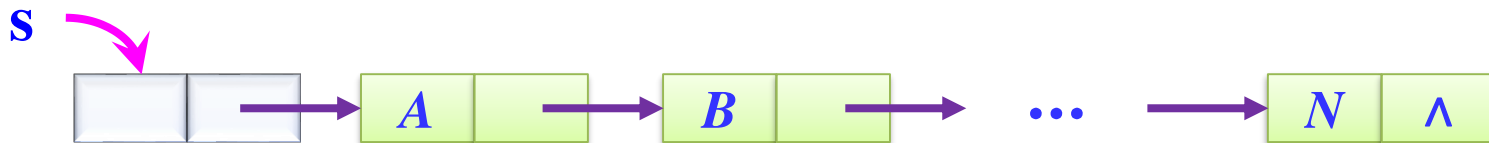
节点大小为4的链串



节点大小为1的链串

链串节点大小1时，链串的节点类型定义如下：

```
typedef struct snode
{
    char data;
    struct snode *next;
} LiString;
```



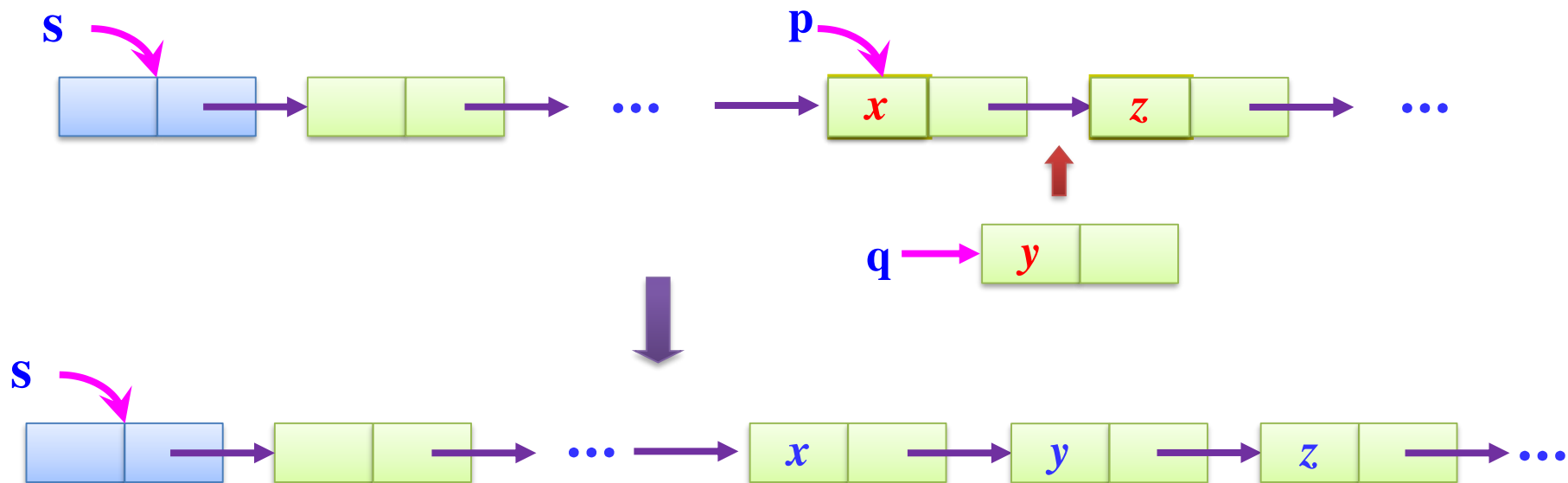
链串中实现串的基本运算与单链表的基本运算类似。详细算法实现参见教材。

【例4-2】 在链串中，设计一个算法把最先出现的子串“*ab*”
改为“*xyz*”。

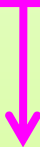
❶ 查找： $p \rightarrow \text{data} = 'a' \ \&\& \ p \rightarrow \text{next} \rightarrow \text{data} = 'b'$



② 替换



```
void Repl(LiString *&s)
{   LiString *p=s->next,*q;
    int find=0;
    while (p->next!=NULL && find==0)           //查找ab子串
    {
        if (p->data==' a' && p->next->data=='b')
        {   p->data='x'; p->next->data='z';
            q=(LiString *)malloc(sizeof(LiString));
            q->data='y'; q->next=p->next; p->next=q;
            find=1;
        }
        else p=p->next;
    }
}
```



替换为xyz

算法的时间复杂度为 $O(n)$ 。

——本讲完——

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。

4.4.1 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。

s:

a	a	a	a	b	c	d
---	---	---	---	---	---	---

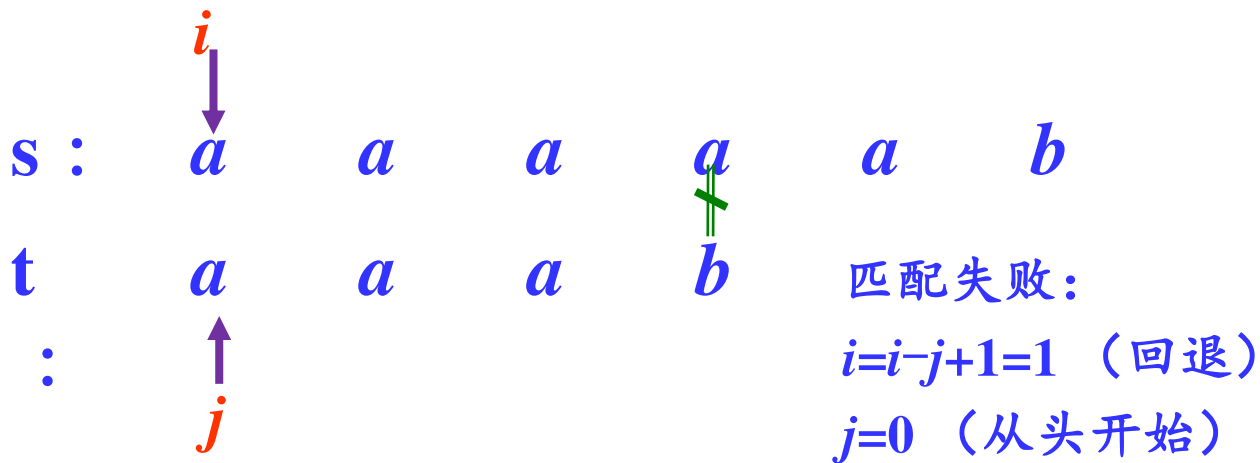
t:

a	a	a	b	c	c
---	---	---	---	---	--------------

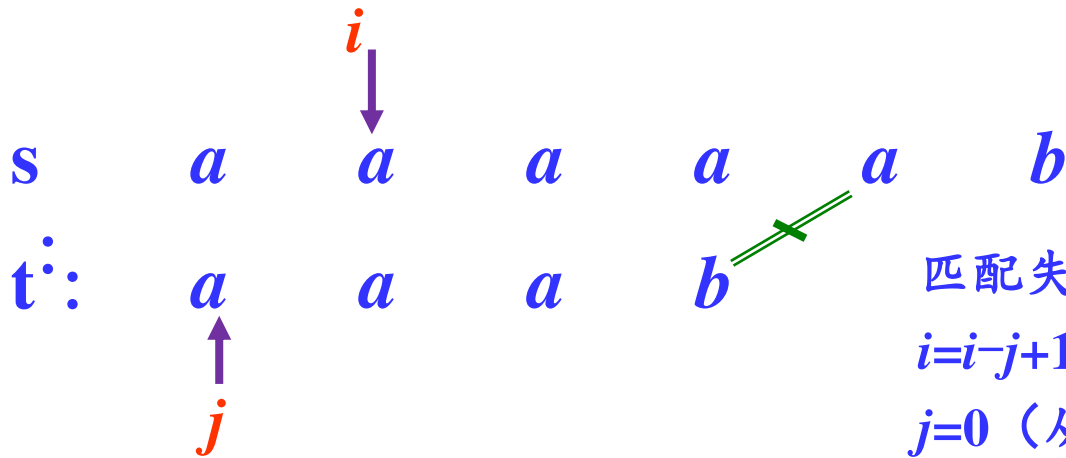
 ✓

匹配成功

例如，设目标串 s ="aaaaab"，模式串 t ="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。



$i=1, j=0$

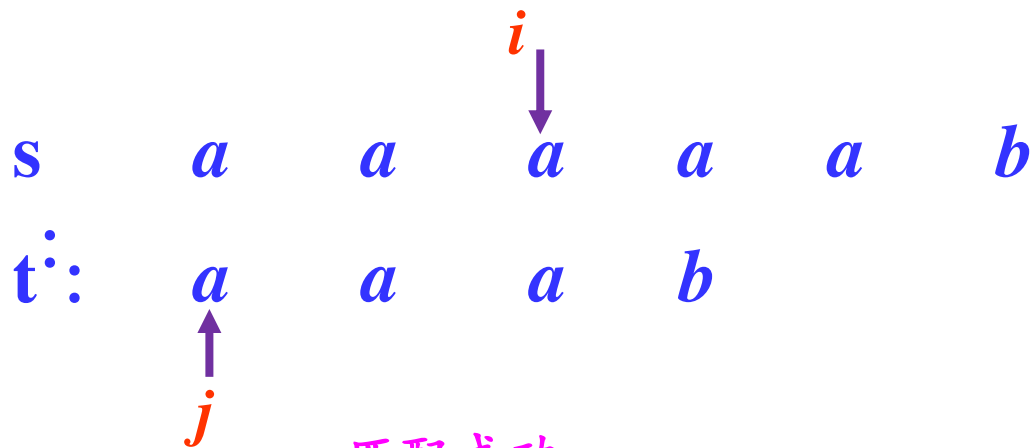


匹配失败:

$i=i-j+1=2$ (回退)

$j=0$ (从头开始)

$i=2, j=0$



匹配成功:

$i=6, j=4$

返回 $i-t.length=2$

对应的BF算法如下：

```
int index(SqString s,SqString t)
{   int i=0, j=0;
    while (i<s.length && j<t.length)
    {   if (s.data[i]==t.data[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.length)
        return(i-t.length);               //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**。

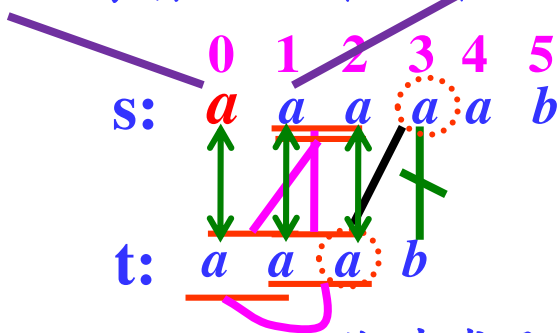
该算法较BF算法有较大改进，主要是**消除了主串指针的回溯**，从而使算法效率有了某种程度的提高。

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”。

开始匹配的字符

下次开始匹配的字符



从t中发现: *b*前面有2个字符和开头的2个字符相同



用一个数组next保存：next[3]=2



下次匹配的字符: `s[3]`和`t[next[3]]`即`t[2]`

$\text{next}[j]$ 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：

$$\underbrace{t_0 t_1 \dots t_{k-1}}_{\text{开头的}k\text{个字符}} = \underbrace{t_{j-k} t_{j-k+1} \dots t_{j-1}}_{t[j]\text{前面的}k\text{个字符}}$$

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



有 $t_0 t_1 = t_2 t_3 = "ab"$ $\Rightarrow k=2$

所以 $\text{next}[4] = k = 2$ 。

归纳起来，定义next[j]数组如下：

$$\text{next}[j] = \begin{cases} \text{MAX}\{k \mid 0 < k < j, \text{ 且 } \overbrace{t_0 t_1 \dots t_{k-1}}^{\text{开头的}k\text{个字符}} = \overbrace{t_{j-k} t_{j-k+1} \dots t_{j-1}}^{\text{后面的}k\text{个字符}}\} & \text{当此集合非空时} \\ -1 & \text{当} j=0 \text{ 时} \\ 0 & \text{其他情况} \end{cases}$$

t = "aaab" 对应的next数组如下：

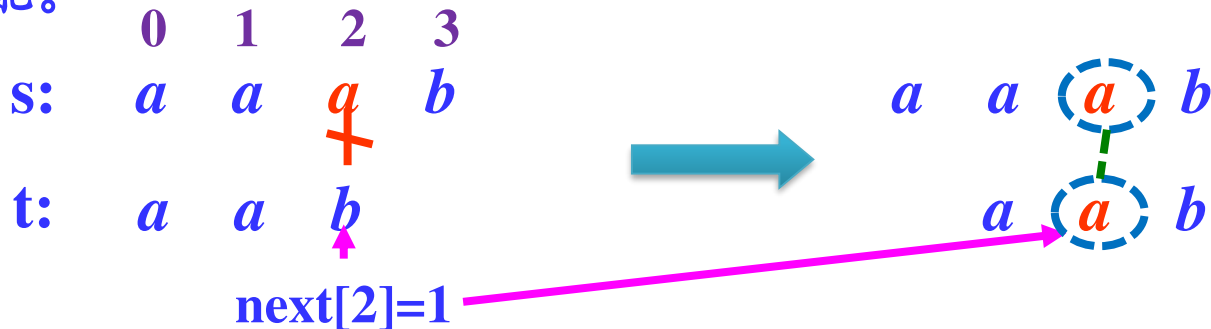
j	0	1	2	3
t[j]	a	a	a	b
next[j]	-1	0	1	2

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

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

由模式串t求next值的算法：

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

KMP算法:

```
int KMPIIndex(SqString s,SqString t)
```

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

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

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

```
    {
```

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

```
        {  i++;
```

```
            j++;
```

//i、j各增1

```
        }
```

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

//i不变, j后退

```
    }
```

没有有用信息或两个字符相等时, 继续比较后面的字符

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

//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)

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4-3】 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

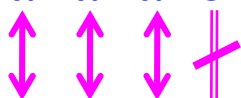
上述KMP算法仍然存在什么缺陷？

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

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

s: **0 1 2 3 4 5 6 7 8**
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
<i>s</i> :	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
<i>t</i> :	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>				
	0	1	2	3	4				

A pink 'X' is drawn over the character 'a' at index 3 of string *s* and the character 'a' at index 3 of string *t*.

失败:
i=3
j=2, *j*=next[2]=1

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

	0	1	2	3	4	5	6	7	8
<i>s</i> :	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>
<i>t</i> :	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>b</i>				
	0	1	2	3	4				

A pink 'X' is drawn over the character 'b' at index 3 of string *s* and the character 'a' at index 2 of string *t*.


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

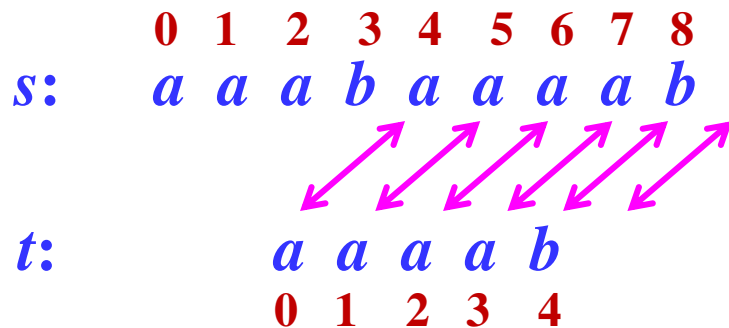
t: *a a a a b*
 0 1 2 3 4



失败:
i=3
j=0, *j*=next[0]=-1

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

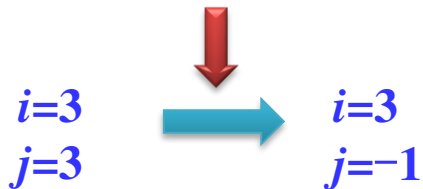
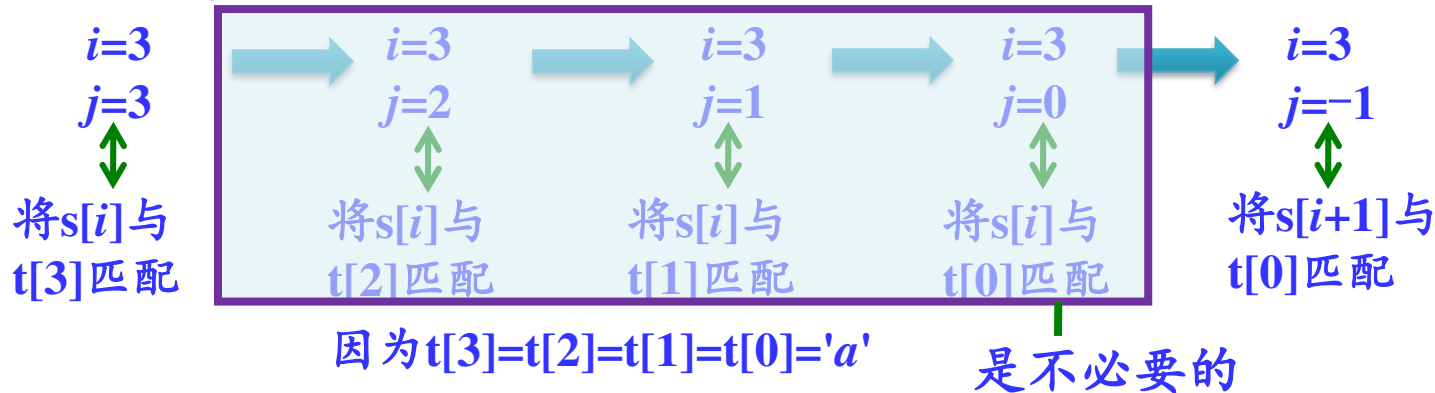
因为 $j = -1$:
 $i++$;
 $j++$;



成功:
返回4

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:



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]	a	a	a	a	b
nextval[j]	-1	-1	-1	-1	3

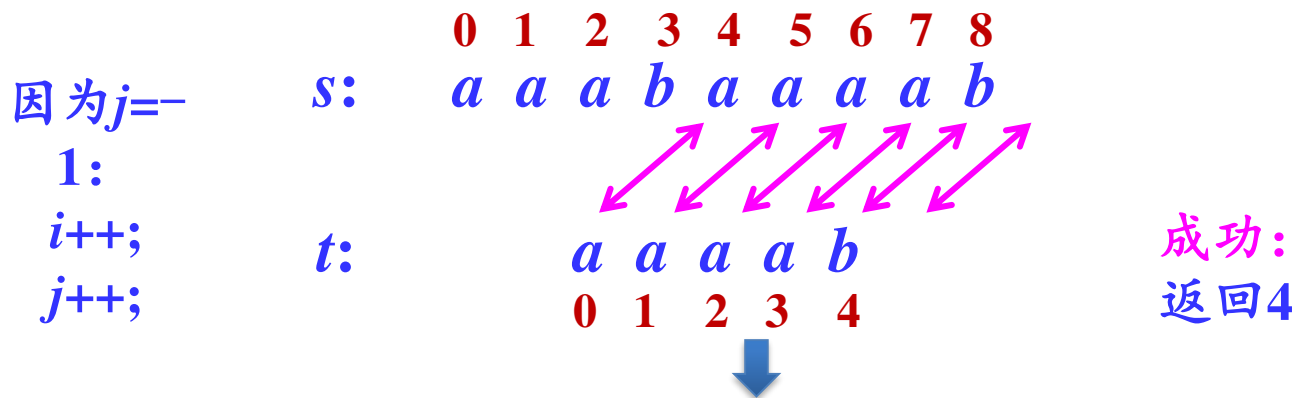
	0	1	2	3	4	5	6	7	8
s:	a	a	a	b	a	a	a	a	b
	↕	↕	↕	⋈					
t:	a	a	a	a	b				
	0	1	2	3	4				

失败：

$i=3$

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

j	0	1	2	3	4
t[j]	a	a	a	a	b
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。

数据结构经典算法的启示

BF算法



利用模式串中部分匹配信息

KMP算法



——本章完——



第5周小结

1

串的存储结构

① 串是一种特殊的线性表，是线性表的一个子集。



- 顺序串

- 链串

② 链串只能采用单链表吗？

- 不一定。需要根据需要情况而定。
- 如果需要从某个节点出发前后查找，可以采用双链表。
- 如果需要快速查找尾节点，可以采用循环双链表。

2

串的算法设计

① 串的基本算法设计

借鉴线性表的算法设计方法。

● 顺序串 \Leftrightarrow 顺序表

● 链 串 \Leftrightarrow 单链表

② 串的模式匹配算法设计

- BF算法
- KMP算法



- ④ 为什么KMP算法平均性能更高？
- ④ 是不是任何情况下KMP算法都好于BF算法？



假设串采用顺序结构存储。设计一个算法求串
 s 中出现的第一个最长重复子串的下标和长度。

解： (i, len) 记录当前重复子串， $(maxi, maxlen)$ 记录第一个最
长重复子串。

										$maxi=0, maxlen=0$		
										$i=0, len=1$	$maxi=0, maxlen=1$	
										$i=1, len=2$	$maxi=1, maxlen=2$	
i ↓	a	a	b	a	b	c	a	b	c	d	$i=2, len=1$	$maxi=1, maxlen=2$
		↑									$i=3, len=3$	$maxi=3, maxlen=3$
		j										



$maxi=3, maxlen=3$, 即 “ abc ”

算法如下：

```
void maxsubstr(SqString s, SqString &t)
{   int maxi=0, maxlen=0, len, i, j, k;
    i=0;
    while (i<s.length)    //从下标为i的字符开始
    {   j=i+1;            //从i的下一个位置开始找重复子串
```



```

while (j<s.length)
{
    if (s.data[i]==s.data[j]) //找一个子串，其起始下标为i，长度为len
    {
        len=1;
        for (k=1;s.data[i+k]==s.data[j+k];k++)
            len++;
        if (len>maxlen) //将较大长度者赋给maxi与maxlen
        {
            maxi=i;
            maxlen=len;
        }
        j+=len;
    }
    else j++;
}
i++; //继续扫描第i字符之后的字符
}

```

```
t.length=maxlen;
```

```
//将最长重复子串赋给t
```

```
for (i=0;i<maxlen;i++)
```

```
    t.data[i]=s.data[maxi+i];
```

```
}
```

```
int main()
{
    SqString s,t;
    StrAssign(s,"aababcbcd");
    printf("s:"); DispStr(s);
    maxsubstr(s,t);
    printf("t:");DispStr(t);
    return 1;
}
```

A screenshot of a Windows command prompt window. The title bar shows the file path "F:\清华大学数据结...". The window contains the following text:

```
s:aababcbcd
t:abc
Press any key to continue
```