

# 第4章 串

4.1 串类型的定义

4.2 串的实现和表示

4.3 串的模式匹配算法

## 4.1 串类型的定义

**串**是一种线性结构，可以看作是特殊的线性表，是由多个或零个字符组成的有限序列，记作

$$S = 'c_1c_2c_3\cdots c_n' \quad (n \geq 0)$$

其中，**S**是串名字，**'c<sub>1</sub>c<sub>2</sub>c<sub>3</sub>⋯c<sub>n</sub>'**是串值

**c<sub>i</sub>**是串中字符，**n**是串的**长度**，表示串中字符的数目。

**空串**：零个字符的串称为空串记作“ $\emptyset$ ”

**子串**：串中任意个连续的字符组成的子序列

**主串**：包含子串的串

**字符在串中的位置**：字符在序列中的序号

**子串在串中的位置**：子串的第一个字符在主串中的位置

例：  $a$ ='BEI'，  $b$ ='JING'，  $c$ ='BEIJING'，  
 $d$ ='BEI JING'

串长度？ 子串？ 子串在串中的位置？

- **串相等：** 当且仅当两个串长度相同，并且各个对应位置的字符都相同
- **空格串：** 由一个或多个空格组成的串
- **串的代表：** 用一对单引号括起来

# 串的抽象数据类型

ADT String {

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

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

    基本操作： 13个

}ADT String

## StrAssign (&T, chars)

初始条件： chars是字符串常量

操作结果： 串赋值，将串值chars赋值给串T

## StrCopy(&T, S)

初始条件： 串S存在

操作结果： 串复制，将一个串S赋给串T。

## StrEmpty(S)

初始条件：串S存在

操作结果：判串空，判断串S是否为空。若S为空串，则返回TRUE, 否则返回FALSE。

## StrCompare(S, T)

初始条件：串S和串T存在

操作结果：串比较，若 $S > T$ ，则返回值 $> 0$ ；若 $S = T$ ，则返回值 $= 0$ ；若 $S < T$ ，则返回值 $< 0$ 。

## StrLength(S)

初始条件：串S存在

操作结果：求串长，返回串S的长度。

## ClearString(&S)

初始条件：串S存在

操作结果：将串S清为空串

**Concat(&T, S1, S2)**

初始条件：串S1和串S2存在

操作结果：用T返回由S1和S2联接而成的新串。

**SubString(&Sub, S, pos, len)**

初始条件：串S存在,  $1 \leq \text{pos} \leq \text{StrLength}(S)$  且  $0 \leq \text{len} \leq \text{StrLength}(S) - \text{pos} + 1$

操作结果：用Sub返回串S的第pos个字符起长度为len的子串。



## Index(S, T, pos)

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

操作结果：返回子串T在主串S中第pos个字符之后第一次出现的位置；若子串T不在主串S中，则函数值为0。

## Replace(&S, T, V)

初始条件：串S，T和V存在，T是非空串

操作结果：用V替换主串S中出现的所有与T相等的  
不重叠的子串。

## StrInsert(&S, pos, T)

初始条件：串S和串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

## 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。

## DestroyString(S)

初始条件：串S存在

操作结果：串S被销毁。

- **基本操作集：**

- 串赋值**StrAssign**、串比较**StrCompare**、求串长**StrLength**、串联接**Concat**、求子串**SubString**, 5种操作构成串类型的最小基本操作集。

- 其他串操作（除了串清除和串销毁）均可在这个最小操作子集上实现。如可用求串长和求子串等操作实现定位函数**Index**。

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

{
    if (pos>0) {
        n = Strlength(S); m = Strlength(T); i= pos;
        while (i <= n-m+1) {
            SubString(sub, S, i, m);
            if (StrCompare(sub, T) != 0) i++;
            else return i;;
        }//while
    }//if
    return 0;
}
```

**串的基本操作集可以有不同的定义方法，依具体的程序设计语言而定。**

- **C语言中提供了丰富的字符串函数**

- **strcat (str1, str2):** 把str2接到str1后面，str1后面的 ‘\0’被取消；

- **strcmp (str1, str2):** 比较两个字符串

- **Strcpy(str1, str2):** 把str2字符串拷贝到str1中去

- **Strlen(str):** 求字符串的长度

## 串的特点：

- 串的逻辑结构：**与线性表极为相似，但串的数据对象约束为字符集

- 串的操作：**与线性表（“单个元素”作为操作对象）有很大差别，以“串的整体”为操作对象

例如，在串中查找某个子串、求取一个子串、插入一个子串、删除一个子串

## 4.2 串的实现和表示

- 三种机内表示:
  - 1. 定长顺序存储表示
  - 2. 堆分配存储表示
  - 3. 串的块链存储表示
- 串的基本操作的实现



# 1. 定长顺序存储表示

- **静态分配**

- 类似于线性表的顺序存储结构，用一组地址连续的存储单元存储串值的字符序列。
- 每个串预先分配一个固定长度的存储区域。

用定长数组描述：

```
#define MAXSTRLEN 255 //最大串长
```

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

//0号单元存放串的长度

- 串长表示方法：
  - 以下标为0的数组分量存放串的实际长度——PASCAL;
  - 在串值后加入” \0” 表示结束，此时串长为隐含值——C
- 实际串长可在所分配的固定长度区域内变动，超过预定义长度的串值则被舍去，称之为“**截断**”。

# 串联结

```
Status Concat(sstring &t,sstring s1,sstring s2) {  
    if (s1[0]+s2[0])<=MAXSTRLEN {  
        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 {  
        t.[1..s1[0]]=s1[1..s1[0]];  
        t.[s1[0]+1..MAXSTRLEN]=s2[1.. MAXSTRLEN-s1[0]];  
        t.[0]= MAXSTRLEN; uncut=FALSE; }  
    else {  
        t.[1.. MAXSTRLEN]=s1[1..MAXSTRLEN];  
        t.[0]= MAXSTRLEN ;uncut=FALSE; }  
    return uncut;  
}
```

# 求子串

```
Status Substring(sstring &sub,sstring s,int pos,int len)
{
    if(pos<1 || pos>s[0] || len<0 || len>s.[0]-pos+1)
        return ERROR;
    sub[1..len]=s[pos..pos+len-1];
    sub[0]=len;
    return OK;
}
```

## 2. 堆分配存储表示

- 以一组地址连续的存储单元存放串值字符序列；
- 存储空间动态分配，用malloc()和free()来管理

//串的堆分配存储表示

```
typedef struct {
```

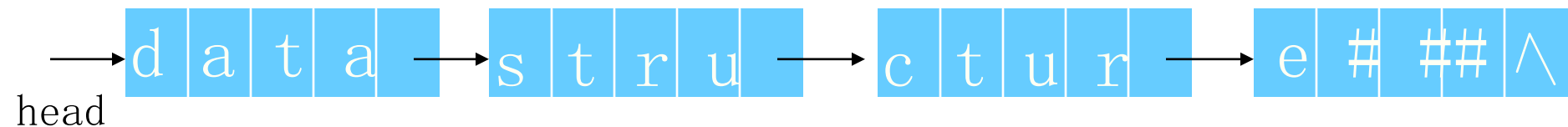
```
    char *ch; // 若为非空串，按实际长度分配  
    存储区，否则ch为NULL
```

```
    int length; //串长度
```

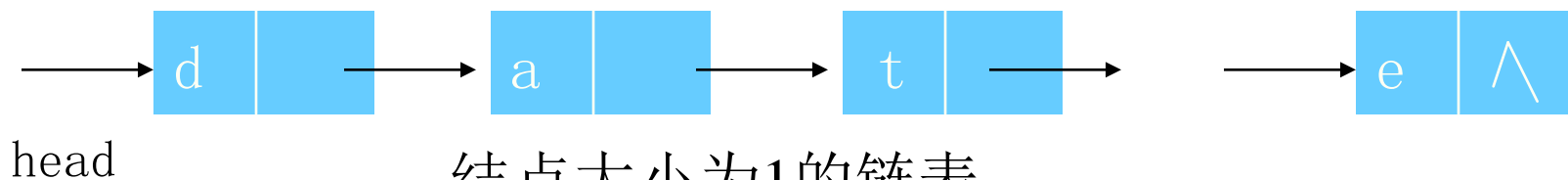
```
}HString;
```

### 3. 串的块链存储表示

- 串的链式存储方式
- 结点大小：一个或多个字符
  - ✱ 串长不一定是结点大小的整倍数，导致最后一个结点不一定全被串值占满，用“#”补上。
  - ✱ 存储密度 = 串值所占的存储位 / 实际分配的存储位
  - ✱ 存储密度小，运算处理方便，然而，存储占用量大。
- P78图4.2 (a) (b)



结点大小为4的链表



结点大小为1的链表

### 串值的链表存储方式

有时，为了便于对串进行连接等操作，在链串中可设置尾指针，指向最后一个结点，并给出当前串的长度。这种定义的存储结构称为**块链结构（P78）**。

## 4. 串的基本操作

- 串插入     `Status StrInsert(HString &S, int pos, HString T)`
- 串赋值     `Status StrAssign(HString &S, char *chars)`
- 求串长     `int StrLength(HString S)`
- 串比较     `int StrCompare(HString S, HString T)`
- 串联接     `Status Concat(HString &S, HString S1, HString S2)`
- 求子串     `Status SubString(HString &Sub, HString S, int pos, int len)`
- 串清空     `Status ClearString(HString &S)`
- 串定位
- 删除
- 置换

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

//在串S的第pos个位置前插入串T

```
{ int i;
  if (pos<1 || pos>S.length+1) return ERROR;
  if (T.length) {
    if (!(S.ch=(char*) realloc(S.ch, (S.length+T.length)*sizeof(char))))
      exit(OVERFLOW);
    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]; }
    for (i=0; i<=T.length-1; i++)
      S.ch[pos-1+i]=T.ch[i];
    S.length+=T.length;
  } return OK;
}
```



Status StrAssign(HString &S, char \*chars)  
生成一个值等于chars的串S

```
{ int i, j;  char *c;
  for (i=0, c=chars;*c;++i, ++c); //求chars的长度
  if (!i) {S.ch=NULL;  S.length=0;}
  else {
    if (!(S.ch=(char *)malloc(i * sizeof(char))))
      exit(OVERFLOW);
    for (j=0; j<=i-1; j++) {
      S.ch[j]=chars[j];}
    S.length=i;
  }
  return OK;
}
```

```
int StrLength(HString S)
```

求串的长度

```
{  
    return S.length;  
}
```

int StrCompare(HString S, HString T)  
比较两个串，若相等返回0

```
{  
    int i;  
    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;  
}
```

Status Concat(HString &S, HString S1, HString S2)

用S返回由S1和S2联接而成的新串

```
{ int j;
  if (!(S.ch = (char*)malloc((S1.length+S2.length)*sizeof(char))))
    exit(OVERFLOW);
  for (j=0; j<=S1.length-1; j++)
    { S.ch[j]=S1.ch[j]; }
  S.length=S1.length+S2.length;
  for (j=0; j<=S2.length-1; j++)
    { S.ch[S1.length+j]=S2.ch[j]; }
  return OK;
}
```

Status SubString(HString &Sub, HString S, int pos, int len)

用Sub返回串S的第pos个字符开始长度为len的子串

```
{
    if (pos<1 || pos>S.length || len<0 || len>S.length-
        pos+1)
        return ERROR;
    if (!len) { Sub.ch=NULL; Sub.length=0;}
    else {
        Sub.ch=(char *)malloc(len*sizeof(char));
        for (int j=0;j<=len-1;j++) {
            Sub.ch[j]=S.ch[pos-1+j];}
        Sub.length=len;
    }
    return OK;
}
```

Status ClearString(HString &S)  
将S清为空串

```
{  
    if (S.ch) { free(S.ch); S.ch=NULL; }  
    S.length=0;  
    return OK;  
}
```

## 4.3 串的模式匹配算法

- \* **定义** 在串中寻找子串（第一个字符）在串中的位置
- \* **词汇** 在模式匹配中，子串称为**模式**，串称为**目标**。
- \* **示例** 目标 S : “Beijing”  
模式 P : “jin”  
匹配结果 = 4

# 1. 穷举模式匹配

**基本思想：** 从源串S的第一个字符开始对模式串T进行匹配，若匹配失败，则从串S的下一位置的字符开始进行匹配，直到匹配成功或到达S的串尾。



# 1. 穷举模式匹配

- 设 $S=s_1, s_2, \dots, s_n$  (主串)  $P=p_1, p_2, \dots, p_m$  (模式串)  
 $i$ 为指向 $S$ 中字符的指针,  $j$ 为指向 $P$ 中字符的指针

匹配失败:  $s_i \neq p_j$ 时,

$$(s_{i-j+1} \dots s_{i-1}) = (p_1 \dots p_{j-1})$$

回溯:  $i=i-j+2$  ;  $j=1$

重复回溯太多,  $O(m*n)$

第1趟	S	a	b	b	a	b	a
	P	a	b	a			

## 穷举的模式 匹配过程

第2趟	S	a	b	b	a	b	a
	P		a	b	a		

第3趟	S	a	b	b	a	b	a
	P			a	b	a	

第4趟	S	a	b	b	a	b	a
	P				a	b	a

√

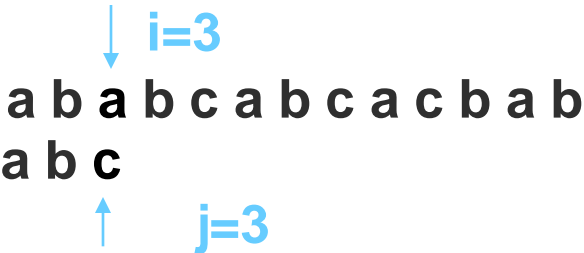
# 求子串位置的定位函数

```
int Index(SString S, SString T, int pos) {  
    //穷举的模式匹配  
    int i=pos;    int j=1;  
    while (i<=S[0] && j<=T[0]) {  
        //当两串未检测完,  
        //S[0]、T[0]为串长  
        if (S[i]==T[j]) {++i; ++j;}  
        else {i=i-j+2; j=1;}  
    }  
    if (j>T[0]) return i-T[0]; //匹配成功  
    else return 0;  
}
```

# 匹配过程:

第一趟匹配

$i=1$



$S[3] \neq T[3]$ , 该趟匹配失败

$i=i-j+2, j=1$  进入下趟

第二趟匹配

$i=2$

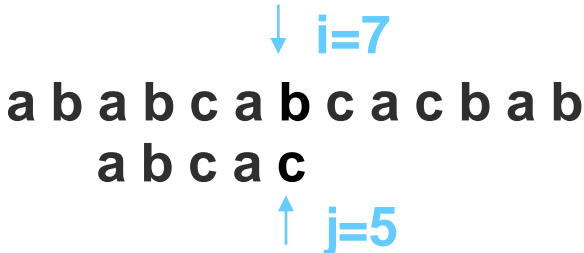


$S[2] \neq T[2]$ , 该趟匹配失败

$i=i-j+2, j=1$  进入下趟

第三趟匹配

$i=3$



$S[7] \neq T[5]$ , 该趟匹配失败

$i=i-j+2, j=1$  进入下趟

# 匹配过程:

第四趟匹配

i=4

↓ i=4

a b a b c a b c a c b a b

a

↑ j=1

第五趟匹配

i=5

↓ i=5

a b a b c a b c a c b a b

a

↑ j=1

第六趟匹配

i=6

↓ i=11

a b a b c a b c a c b a b

a b c a c

↑ j=6

匹配成功

S [4]!=T[1], 该趟匹配失败  
i=i- j+2, j=1 进入下趟

S [5]!=T[1],该趟匹配失败  
i=i- j+2, j=1 进入下趟

j 等于T的串长+1, 该趟匹配成功,  
返回i-T[0]

## 算法分析：

优点：简单易懂，实现容易。

缺点：算法的执行效率较低，因为该算法实际上是对串S中所有位置进行穷举，直到匹配成功或扫描完S中全部位置。

算法时间复杂度：

设串S和T的长度分别为m和n，在最坏的情况下的时间复杂度为 $O(m*n)$ 。

$s = \text{'aaaaaaaaaaaaaaaaaba'}$   
 $t = \text{'aaaaaaaba'}$



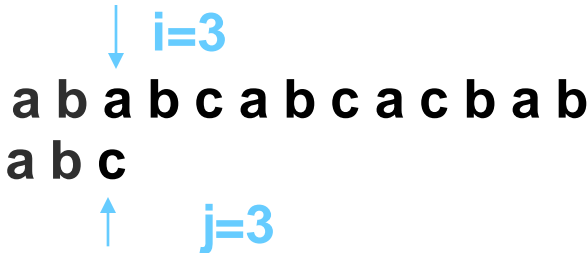
## 2. KMP快速模式匹配

- D. E. Knuth, J. H. Morris, V. R. Pratt同时发现
- 无回溯的模式匹配
- **改进：** 每当一趟匹配过程中出现字符比较不等时，不需回溯i指针，而是利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离后，继续进行比较。

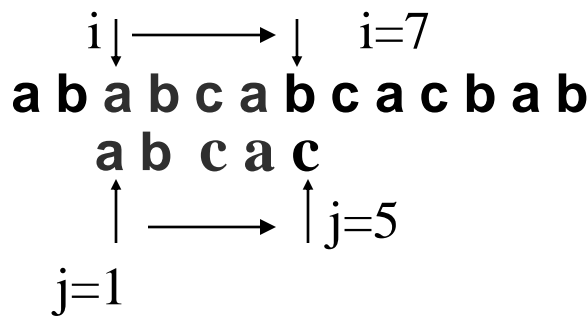


匹配过程示例:

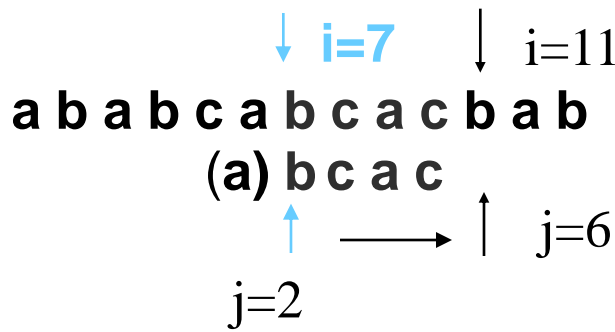
第一趟匹配



第二趟匹配



第三趟匹配



一般情况：

$$\begin{array}{cccccccccccccccc}
 \textcolor{red}{S} & s_1 & \cdots & s_{i-j-1} & s_{i-j} & s_{i-j+1} & s_{i-j+2} & \cdots & s_{i-1} & s_i & s_{i+1} & \cdots & s_n \\
 & & & & & \parallel & \parallel & \parallel & \parallel & \times & & & \\
 \textcolor{red}{P} & & & & & p_1 & p_2 & \cdots & p_{j-1} & p_j & p_{j+1} & \cdots & p_m
 \end{array}$$

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

为使模式  $\textcolor{red}{P}$  与目标  $\textcolor{red}{S}$  匹配，必须满足

$$p_1 p_2 \cdots p_{j-1} p_j \cdots p_m = s_{i-j+1} s_{i-j+2} \cdots s_{i-1} s_i \cdots s_{i-j+m}$$

$$\text{如果 } p_1 \cdots p_{j-2} \neq p_2 p_3 \cdots p_{j-1} \quad (2)$$

由(1) (2)则立刻可以断定

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

下一趟必不匹配

同样，若  $p_1 p_2 \cdots p_{j-3} \neq p_3 p_4 \cdots p_{j-1}$

则再下一趟也不匹配，因为有

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

直到对于某一个“ $k$ ”值，使得

$$p_1 \cdots p_k \neq p_{j-k} p_{i-k+1} \cdots p_{j-1}$$

且  $p_1 \cdots p_{k-1} = p_{j-k+1} p_{j-k+2} \cdots p_{j-1}$

则 
$$\begin{array}{ccccccc}
 p_1 & \cdots & p_{k-1} & = & s_{i-k+1} & s_{i-k+2} & \cdots & s_{i-1} \\
 & & & & \parallel & \parallel & & \parallel \\
 & & & & p_{j-k+1} & p_{j-k+3} & \cdots & p_{j-1}
 \end{array}$$

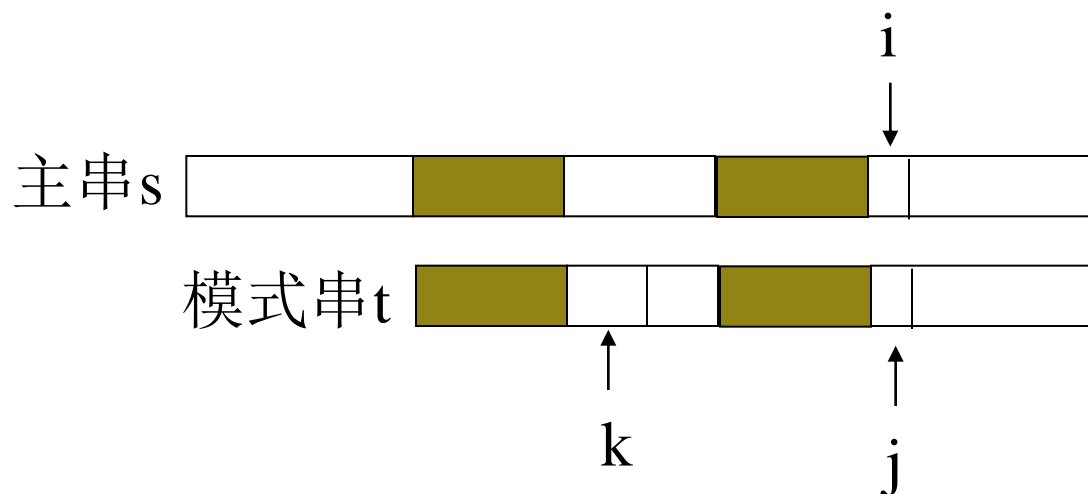
模式右滑  $j-k$  位

# next数组值

- 假设当模式中第j个字符与主串中相应字符“失配”时，可以拿第k个字符来继续比较，则令 $\text{next}[j]=k$

next函数定义：

$$\text{next}[j] = \begin{cases} 0 & \text{当 } j=1 \text{ 时} \\ \text{Max} \{k \mid 1 < k < j \text{ 且 } p_1 \cdots p_{k-1} = p_{j-k+1} \cdots p_{j-1}\} & \text{当此集合不空时} \\ 1 & \text{其他情况} \end{cases}$$



# 手工求next数组的方法

• 序号j	1	2	3	4	5	6	7	8
• 模式P	a	b	a	a	b	c	a	c
• k		1	1	2	2	3	1	2
• $P_k == P_j$		$\neq$	$=$	$\neq$	$=$	$\neq$	$=$	$\neq$
• next[j]	0	1	1	2	2	3	1	2
• Nextval[j]	0	1	0	2	1	3	0	2

在求得了 $\text{next}[j]$ 值之后，KMP算法的思想是：

设目标串(主串)为 $s$ ，模式串为 $t$ ，并设 $i$ 指针和 $j$ 指针分别指示目标串和模式串中正待比较的字符，设 $i$ 和 $j$ 的初值均为1。若有 $s_i=t_j$ ，则 $i$ 和 $j$ 分别加1。否则， $i$ 不变， $j$ 退回到 $j=\text{next}[j]$ 的位置，再比较 $s_i$ 和 $t_j$ ，若相等，则 $i$ 和 $j$ 分别加1。否则， $i$ 不变， $j$ 再次退回到 $j=\text{next}[j]$ 的位置，依此类推。直到下列两种可能：

- (1)  $j$ 退回到某个下一个 $[j]$ 值时字符比较相等，则指针各自加1继续进行匹配。
- (2) 退回到 $j=0$ ，将 $i$ 和 $j$ 分别加1，即从主串的下一个字符 $s_{i+1}$ 模式串的 $t_1$ 重新开始匹配。

匹配过程举例如下：

# 运用KMP算法的匹配过程

第1趟 目标 a c a b a a b a a b c a c a a b c  
模式 a b a a b c a c  
 $\times \text{next}(2) = 1$

第2趟 目标 a c a b a a b a a b c a c a a b c  
模式 a b a a b c a c  $\text{next}(1)=0$

第3趟 目标 a c a b a a b a a b c a c a a b c  
模式 a b a a b c a c  
 $\times \text{next}(6) = 3$

第4趟 目标 a c a b a a b a a b c a c a a b c  
模式 (a b) a a b c a c  
 $\checkmark$

# KMP算法

```
int Index_KMP(SString S, SString T, int
    *next) { int i, j;
    i=1; j=1;
    while (i<=S[0] && j<=T[0]) {
        if (j==0 || S[i]==T[j]) {++i; ++j;}
        else j=next[j];
    }
    if (j>T[0]) return i-T[0];
    else return 0;
}
```



很显然，KMP\_index函数是在已知下一个函数值的基础上执行的，以下讨论如何求next函数值？

求模式串的next[j]值与主串s无关，只与模式串t本身的构成有关，则可将求next函数值的问题看成是一个模式匹配问题。由next函数定义可知：

当 $j=1$ 时： $\text{next}[1]=0$ 。

设 $\text{next}[j]=k$ ，即在模式串中存在： $t_1t_2\cdots t_{k-1}=t_{j-(k-1)}t_{j-k}\cdots t_{j-1}$ ，其中下标 $k$ 满足 $1<k<j$ 的某个最大值，此时求 $\text{next}[j+1]$ 的值有两种可能：

(1) 若有 $t_k=t_j$ ：则表明在模式串中有：

$t_1t_2\cdots t_{k-1}t_k=t_{j-(k-1)}t_{j-k}\cdots t_{j-1}t_j$ ，且不可能存在 $k'>k$ 满足上式，即： $\text{next}[j+1]=\text{next}[j]+1=k+1$

(2) 若有  $t_k \neq t_j$  : 则表明在模式串中有:  $t_1 t_2 \dots t_{k-1} t_k \neq t_{j-(k-1)} t_{j-k} \dots t_{j-1} t_j$ , 当  $t_k \neq t_j$  时应将模式向右滑动至以模式中的第  $\text{next}[k]$  个字符和主串中的第  $j$  个字符相比较。  
若  $\text{next}[k] = k'$ , 且  $t_j = t_{k'}$ , 则说明在主串中第  $j+1$  字符之前存在一个长度为  $k'$  (即  $\text{next}[k]$ ) 的最长子串, 与模式串中从第一个字符起长度为  $k'$  的子串相等。即  
 $\text{next}[j+1] = k' + 1$

同理, 若  $t_j \neq t_k$ , 应将模式继续向右滑动至将模式中的第  $\text{next}[k']$  个字符和  $t_j$  对齐, …… , 依此类推, 直到  $t_j$  和模式串中的某个字符匹配成功或者不存在任何  $k'$  ( $1 < k' < j$ ) 满足等式:  $t_1 t_2 \dots t_{k-1} t_{k'} = t_{j-(k'-1)} t_{j-k'} \dots t_{j-1} t_j$

则:  $\text{next}[j+1] = 1$

# 求next数组的步骤

(1)  $\text{next}[1]=0$

$i=1; \quad j=0;$

(2) 设  $\text{next}[i]=j$

若  $j=0$ ,  $\text{next}[i+1]=1$

若  $P_i=P_j$ ,  $\text{next}[i+1]=j+1=\text{next}[i]+1$

若  $P_i \neq P_j$ ,  $\text{next}[i+1]=\text{next}[j]+1$

参看教材p82~83递推过程

举例：

## 求next数组的函数

```
void get_next(SString S, int *next) {  
    int i, j;  
    i=1;    next[1]=0; j=0;  
    while (i<S[0]) {  
        if (j==0 || S[i]==S[j]) {++i; ++j;  
            next[i]=j;}  
        else j=next[j];  
    }  
}
```

# 改进的求next数组方法

设 $\text{next}[i]=j$

若 $P[i]=P[j]$ , 则 $\text{nextval}[i]=\text{nextval}[j]$

例:

• 序号j	1	2	3	4	5
• 模式P	a	a	a	a	b
• $\text{next}[j]$	0	1	2	3	4
• $\text{Nextval}[j]$	0	0	0	0	4

# 改进的求next数组的函数

```
void get_nextval(SString S, int *nextval) {  
    int i, j;  
    i=1;    nextval[1]=0;    j=0;  
    while (i<S[0]) {  
        if (j==0 || S[i]==S[j]) {  
            ++i; ++j;  
            if (S[i]!=S[j]) nextval[i]=j;  
            else nextval[i]=nextval[j];  
        }  
        else j=nextval[j];  
    }  
}
```

- 穷举的模式匹配算法时间代价：
  - 最坏情况比较 $n-m+1$ 趟，每趟比较 $m$ 次，总比较次数达 $(n-m+1)*m$
- 原因在于每趟重新比较时，目标串的检测指针要回退。改进的模式匹配算法可使目标串的检测指针每趟不回退。
- 改进的模式匹配(KMP)算法的时间代价：
  - ◆ 若每趟第一个不匹配，比较 $n-m+1$ 趟，总比较次数最坏达 $(n-m)+m = n$
  - ◆ 若每趟第 $m$ 个不匹配，总比较次数最坏亦达到  $n$
  - ◆ 求next函数的比较次数为 $m$ ，所以总的时间复杂度是 $O(n+m)$

# 小 结

## \* 重点:

- \* 串的基本概念：空串与空白串、主串和子串、目标串和模式串
- \* 串的基本运算：求串长、串复制、串联接、串比较、串的字符定位
- \* 串的两种存储结构：定长顺序存储、堆分配存储
- \* 串的朴素的模式匹配算法