

第四章

串

※ 教学内容:

串的逻辑结构，存储结构；串操作的实现

※ 教学重点:

串的七种基本运算的定义，利用这些基本运算来实现串的其它各种运算的方法；在顺序存储结构和在堆存储结构上实现串的各种操作的方法；KMP算法，熟悉NEXT函数和改进NEXT函数的定义和计算。

※ 教学难点:

KMP算法，手工计算NEXT函数和改进NEXT函数的值。

第四章 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

一、串的定义

二、串的抽象数据类型的定义

一、串的定义

串 (string) (或字符串)，是由零个或多个字符组成的有限序列。一般记为

$$S = 'a_1 a_2 a_3 \dots a_n' \quad (n \geq 0)$$

其中，S是**串名**，单引号括起来的字符序列是**串的值**；

a_i 可以是字母，数字或其他字符；

串中的字符个数称为**串的长度**；

零个字符的串称为**空串** (Null string)，其长度为零；

串中任意个连续的字符组成的子序列称为该串的**子串**；

包含子串的串相应地称为**主串**；

字符在序列中的序号为该字符在串中的**位置**；

子串在主串中的位置则以子串的第一个字符在主串中的位置来表示；

由一个或多个空格字符组成的串称为**空格串**；

例： 串a, b, c, d, e五个串如下：

a="very difficult";

b=" ";

c="";

d="very";

e="difficult";

串a的长度为___;

串b是长度为___的空格串;

串c是空串, 长度为___;

串d和e是串a的子串, 串d在串a中的位置是___, 串e在串a中的位置是___。

- **串相等**：当且仅当两个串的值相等，即只有当两个串的**长度**相等，并且各个**对应位置**的字符都相等时才相等；
- 空串是任意串的子串；
- 任意串又是自己的子串

注意：

- (1)串值必须用一对单引号括起来，但单引号本身不属于串；
- (2)空串与空格串截然不同，空串不包含任何字符。

二、串的抽象数据类型的定义如下:

ADT String {

数据对象:

$$D = \{ a_i \mid a_i \in \text{CharacterSet}, \\ i=1,2,\dots,n, \quad n \geq 0 \}$$

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:

StrAssign (&T, chars)

DestroyString(&S)

StrCopy (&T, S)

StrLength(S)

StrCompare (S, T)

Concat (&T, S1, S2)

StrEmpty (S)

SubString (&Sub, S, pos, len)

ClearString (&S)

Index (S, T, pos)

Replace (&S, T, V)

StrInsert (&S, pos, T)

StrDelete (&S, pos, len)

} ADT String

StrAssign (&T, chars)

初始条件: chars 是字符串常量。

操作结果: 把 chars 赋为 T 的值。

StrCopy (&T, S)

初始条件： 串 S 存在。

操作结果： 由串 S 复制得串 T。

DestroyString (&S)

初始条件： 串 S 存在。

操作结果： 串 S 被销毁。

StrEmpty (S)

初始条件： 串S存在。

操作结果： 若 S 为空串，
则返回TRUE；
否则返回 FALSE。

“ ” 表示空串， 空串的长度为零。

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 。

StrLength (S)

初始条件： 串 S 存在。

操作结果： 返回 S 的元素个数，
称为串的长度。

Concat (&T, S1, S2)

初始条件：串 S1 和 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。

子串为“串”中的一个字符子序列

例如：

SubString(sub, 'command', 4, 3)

求得 sub = **'man'**

SubString(sub, 'command', 1, 7)

求得 sub = **'command'**

SubString(sub, 'command', 7, 1)

求得 sub = **'d'**

SubString(sub, 'commander', 4, 7)

sub = ?

SubString(sub, 'beijing', 7, 2) = ?

sub = ?

- 起始位置和子串长度之间存在约束关系

SubString('student', 5, 0) = ""

- 长度为 0 的子串为 “合法” 串

Index (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： 若主串 S 中存在和串 T 值相同的子串，则返回T在主串 S 中第pos个字符之后第一次出现的位置；否则函数值为0。

“子串在主串中的位置” 意指子串中的第一个字符在主串中的**位序**。

假设 $S = \text{'abcaabcaabc'}$, $T = \text{'bca'}$

$$\text{Index}(S, T, \mathbf{1}) = 2$$

$$\text{Index}(S, T, \mathbf{3}) = 6$$

$$\text{Index}(S, T, \mathbf{8}) = 0$$

Replace (&S, T, V)

初始条件： 串S, T和 V 均已存在，
且 T 是非空串。

操作结果： 用V替换主串S中出现
的所有与（模式串）T
相等的**不重叠**的子串。

例如 Replace (&S, T, V) :

假设 $S = \text{'abcaabcaabca'}$, $T = \text{'bca'}$

若 $V = \text{'x'}$, 则经置换后得到

$S = \text{'axaxaax'}$

若 $V = \text{'bc'}$, 则经置换后得到

$S = \text{'abcabcaabc'}$

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。

例如： S = 'chater', T = 'rac',

则执行 StrInsert(S, 4, T)之后得到

S = 'character'

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。

ClearString (&S)

初始条件： 串S存在。

操作结果： 将S清为空串。

对于串的基本操作集可以有不同的定义方法，在使用高级程序设计语言中的串类型时，应以该语言的参考手册为准。

例如：C语言函数库中提供下列串处理函数：

gets(str) 输入一个串；

puts(str) 输出一个串；

strcat(str1, str2) 串联接函数；

strcpy(str1, str2, k) 串复制函数；

strcmp(str1, str2) 串比较函数；

strlen(str) 求串长函数；

在上述抽象数据类型定义的13种操作中，

串赋值StrAssign、串复制Strcopy、

串比较StrCompare、求串长StrLength、

串联接Concat以及求子串SubString

等六种操作构成串类型的最小操作子集。

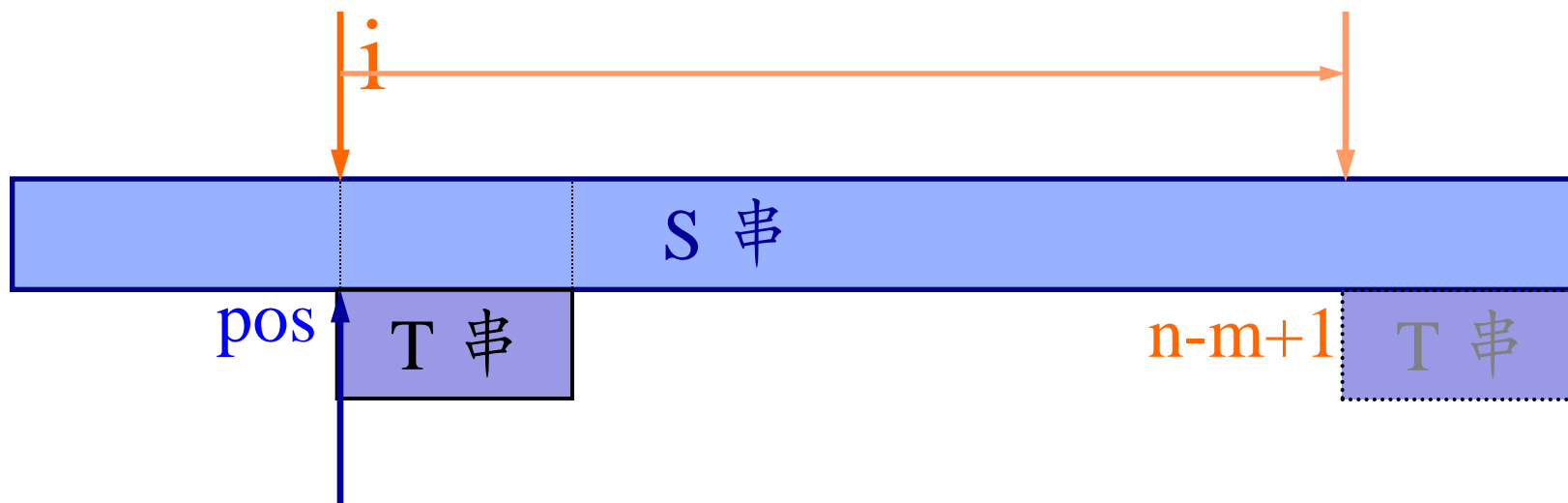
即：这些操作不可能利用其他串操作来实现，
反之，其他串操作（除串清除ClearString和串
销毁DestroyString外）可在这个最小操作子集
上实现。

例如，定位函数**Index(S,T,pos)**如何实现？

可利用串比较、求串长和求子串操作实现。

实现方法：

$\text{StrCompare}(\text{SubString}(S, i, \text{StrLength}(T)), T)$



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

```
    // T为非空串。若主串S中第pos个字符之后存在与T相等的子串，  
    // 则返回第一个这样的子串在S中的位置，否则返回0
```

```
    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 ;           //返回子串在主串中的位置
```

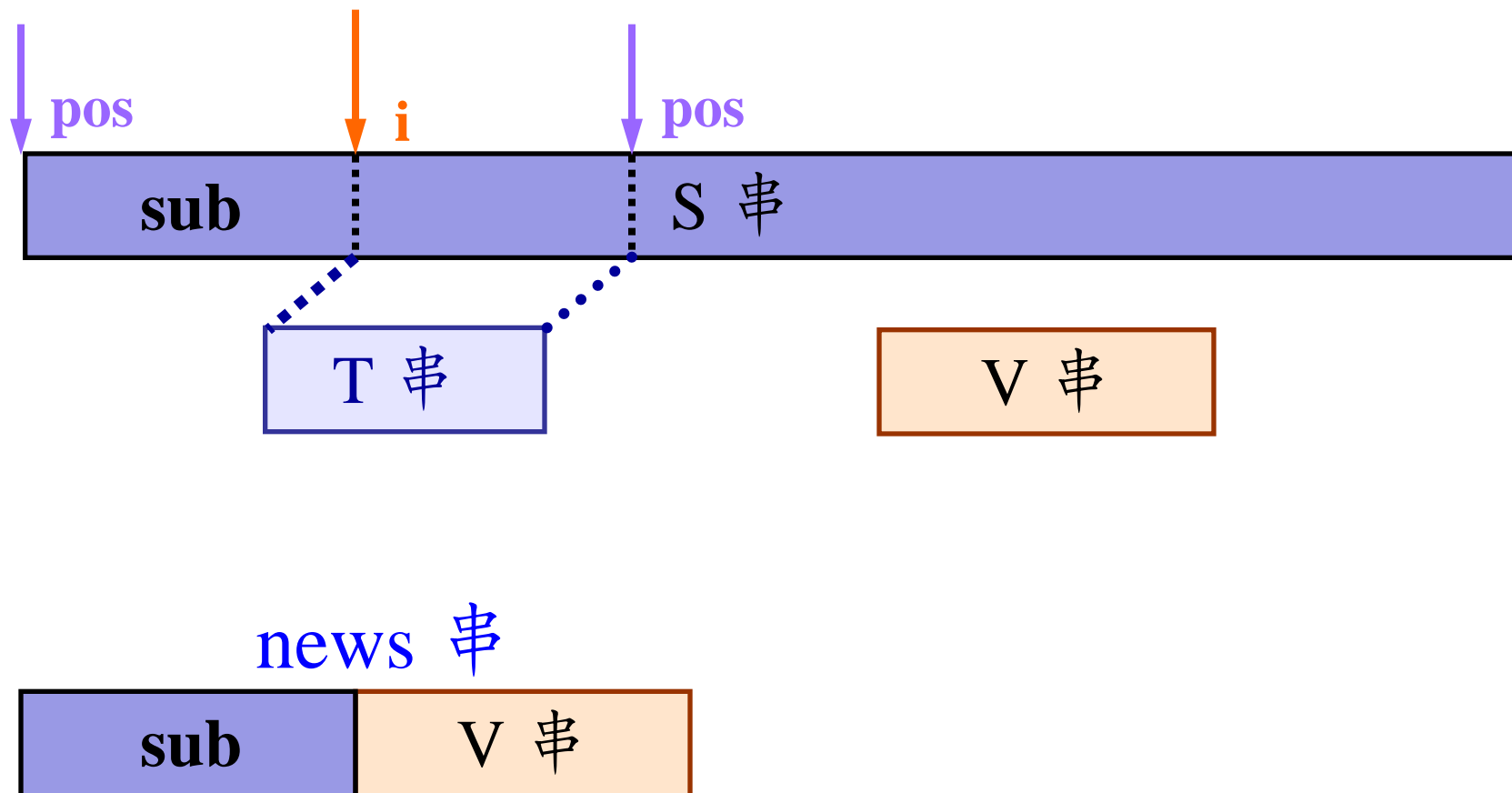
```
      } // while
```

```
    } // if
```

```
    return 0;           // S中不存在与T相等的子串
```

```
} // Index
```

又如串的置换函数 **Replace**(&S,T,V):



串的**逻辑结构**和**线性表**极为**相似**，**区别**仅在于串的**数据对象**约束为**字符集**。

串的**基本操作**和**线性表**有**很大差别**。

在线性表的基本操作中，大多以“**单个元素**”作为操作对象；

在串的基本操作中，通常以“**串的整体**”作为操作对象。

4.2 串的实现和表示

在程序设计语言中，串只是作为输入或输出的常量出现，则只需存储此串的串值，即字符序列即可。但在多数非数值处理的程序中，串也以变量的形式出现。

串有三种机内表示方法：

一、串的定长顺序存储表示

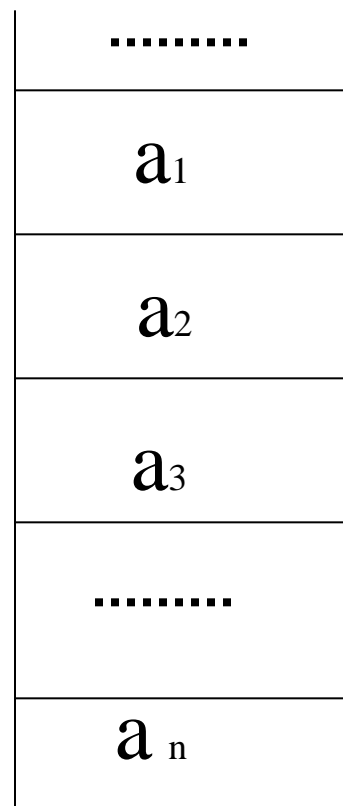
二、串的堆分配存储表示

三、串的块链存储表示

一、串的定长顺序存储表示

将串中的字符顺序地存放在内存一片连续的存储单元中。

设串 $S = \langle a_1 a_2 \dots a_n \rangle$, 则顺序存储结构在内存贮器中的存储示意图如图所示:



串的顺序存贮结构的两种方式

- 非紧缩存贮格式

即一个字的存贮单元存放一个字符。

- 紧缩存贮格式

即一个字的存贮单元中放满多个字符，
然后在往下一个字存贮单元存放

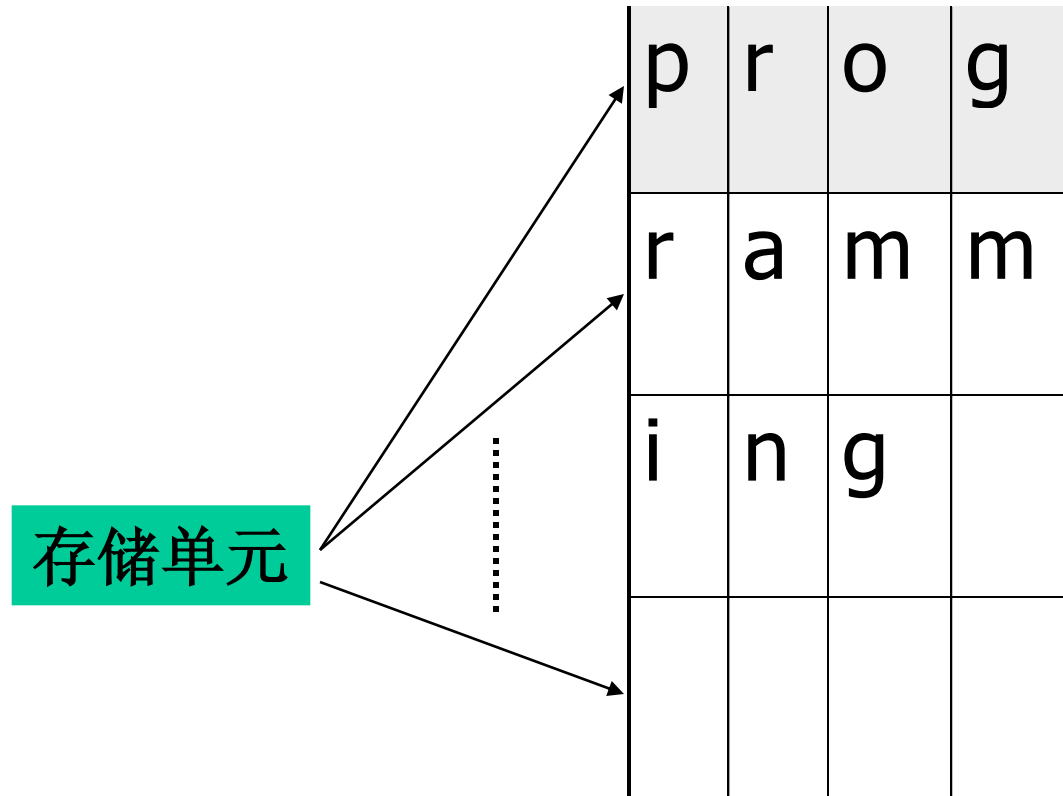
非紧缩存储结构

存储单元

⋮

p			
r			
o			
g			
r			
a			
m			
m			
i			
n			
g			

紧缩存贮结构



与非紧缩存贮格式相比，紧缩存贮格式节省了存贮空间

Status Concat(SString S1, SString S2, SString &T) {

// 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[0..MAXSTRLEN];
```

```
    // T[0] == S1[0] == MAXSTRLEN
```

```
    uncut = FALSE; }
```

```
return uncut;
```

```
} // Concat
```


2.求子串 SubString(&Sub,S,pos,len)

求子串的过程即为复制字符序列的过程，将串S中从第pos个字符开始长度为len的字符序列复制到串Sub中。显然，本操作不会有需截断的情况，但有可能产生用户给出的参数不符合操作的初始条件，当参数非法时，返回ERROR。

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

```
// 用Sub返回串S的第pos个字符起长度为len的子串.其中,  
1≤pos≤StrLength(S)且0≤len≤StrLength(S)-pos+1
```

```
    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;
```

```
} // SubString
```

综上两个操作可见，在顺序存储结构中，实现串操作的**原操作**为“**字符序列的复制**”，操作的时间复杂度基于复制的字符序列的长度。

另一操作**特点**是，如果在操作中出现串值序列的长度**超过上界**MAXSTRLEN时，约定用**截尾法**处理，这种情况下不仅在求联接串时可能发生，在串的其它操作中，如插入、置换等也可能发生。**克服这个弊病**唯有不限定串长的最大长度，即**动态分配串值的存储空间**。

二、串的堆分配存储表示

仍以一组地址连续的存储单元存放串值字符序列，但它们的存储空间是在程序执行过程中动态分配而得。在C语言中，存在一个称之为“**堆**”的**自由存储区**，并由C语言的动态分配函数`malloc()`和`free()`来管理。利用**函数**`malloc()`为每个新产生的串**分配一块实际串长所需的存储空间**，若分配成功，则**返回**一个指向起始地址的**指针**，作为串的基址。

例：`ch=(char*)malloc(len*sizeof(char))`

申请分配一个串长度为`len`的存储空间。

串的堆分配存储表示:

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

这类串操作**实现的算法**为：

先为新生成的串分配一个存储空间，然后进行串值的复制。

例如：串复制操作**StrCopy(&T,S)**的实现算法是，若T已存在，则先释放串T所占空间，当串S不空时，首先为串T分配大小和串S长度相等的存储空间，然后将串S的值复制到串T中；

又如串插入操作**StrInsert(&S, pos, T)**的实现算法是，为串S重新分配大小等于串S和串T长度之**和**的存储空间，然后进行串值复制，如下述算法：

Status SubInsert (HString &S, int pos, Hstring T) {

// $1 \leq \text{pos} \leq \text{StrLength}(S)+1$ 。在串S的第pos个字符之前插入串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))))

exit(OVERFLOW);

for (i = S.length - 1; i >= pos - 1; i--) // 为插入T而腾出位置，数组下标
// 从0--- S.length-1。

S.ch[i + T.length] = S.ch[i]; // 后移T.length个位置

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

S.length += T.length; } return OK;

} // SubInsert

以上两种存储表示通常为高级程序设计语言所采用。由于堆分配存储结构的串既有顺序存储结构的特点，处理方便，操作中对串长又没有任何限制，更显灵活，因此，在串处理的应用程序中也常被选用。

以下所示为只含最小操作子集的Hstring串类型的模块说明：

基本操作的函数原型说明:

Satus StrAssign(Hstring &T,char *chars);

//生成一个其值等于串常量chars的串T

int StrLength(Hstring S);

//返回S的元素个数，称为串的长度

int Strcompare(Hstring S, Hstring T);

//若S>T，则返回值>0;

//若S=T，则返回值=0;

//若S<T，则返回值<0;

Status ClearString(Hstring &S);

//将S清为空串，并释放S所占空间

Status Concat (Hstring &T,Hstring S1,Hstring S2);

//用T返回S1和S2联接而成的新串

Hstring SubString(Hstring S, int pos,int len);

// $1 \leq \text{pos} \leq \text{StrLength}(S)$, 且 $0 \leq \text{len} \leq \text{StrLength}(S) - \text{pos} + 1$

//返回串S的第pos个字符起长度为len的子串

```
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))))  
        exit (OVERFLOW); //分配失败时退出  
    T.ch[0..S1.length-1] = S1.ch[0..S1.length-1];  
    T.ch[S1.length..T.length-1] = S2.ch[0..S2.length-1];  
    T.length = S1.length + S2.length;  
    return OK;  
} // Concat
```

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

```
    //用Sub返回串S的第pos个字符起长度为len的子串
```

```
if (pos < 1 || pos > S.length || len < 0 || len > S.length-pos+1)
```

```
    return ERROR;           //位置不合适
```

```
if (Sub.ch) free (Sub.ch);    //释放sub的旧空间
```

```
if (!len)           //长度为0
```

```
    { Sub.ch = NULL; Sub.length = 0; } //空子串
```

```
else { Sub.ch = (char *)malloc(len*sizeof(char));
```

```
    Sub.ch[0..len-1] = S.ch [pos-1..pos+len-2];
```

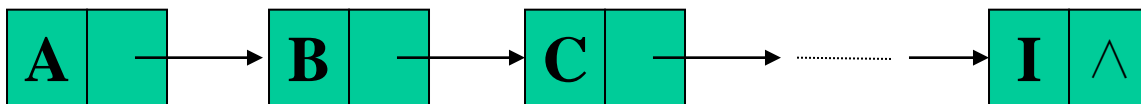
```
    Sub.length = len; }      //完整子串
```

```
    return OK;
```

```
} // SubString
```

三、串的块链存储表示

也可用链表来存储串值，由于串结构的特殊性——结构中的每个数据域元素是一个字符，则用链表存储串值时，存在一个“**结点大小**”的问题，即每个结点可以存放一个字符，也可以存放多个字符。



结点大小为1的链表



结点大小为4的链表

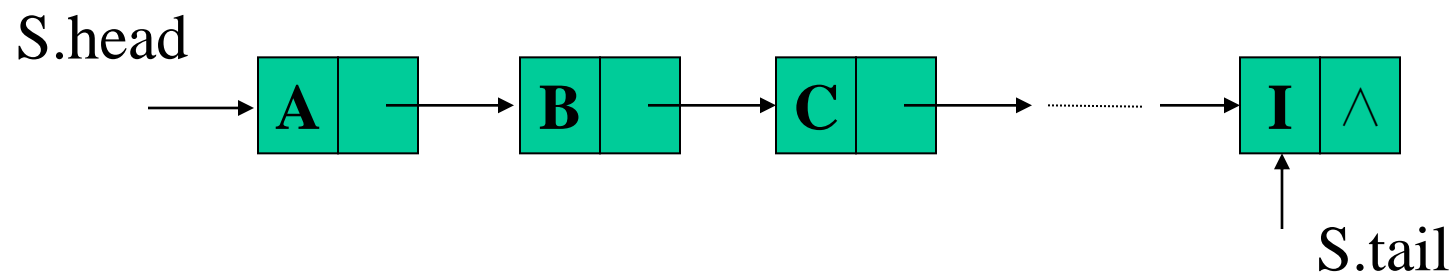
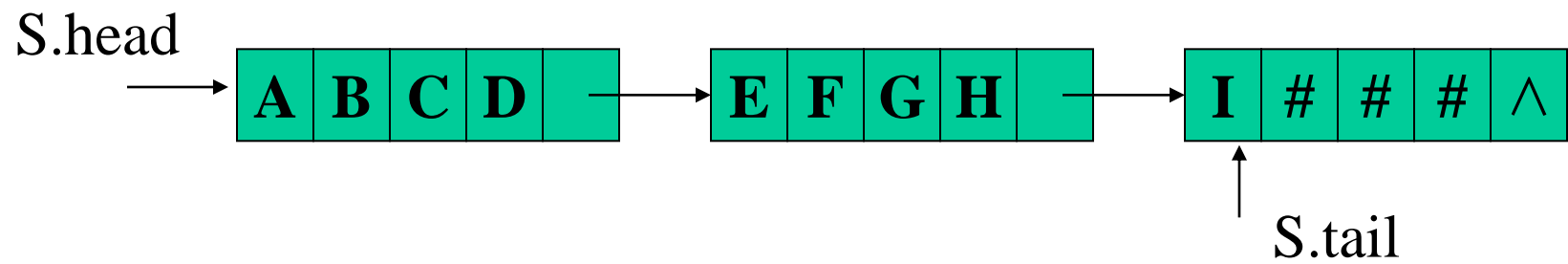
当结点大小大于1时。由于串长不一定是结点大小的整数倍，则链表中的最后一个结点不一定全被串值占满，此时通常补上“#”或其它的非串值字符。

为了便于进行串的操作，当以链表存储串值时，除**头指针**外，还可附设一个**尾指针**指示链表中的最后一个结点，并给出当前串的长度。称如此定义的串存储结构为**块链结构**，说明如下：

```
#define CHUNKSIZE 80 //可由用户定义的块大小

typedef struct Chunk { // 结点结构
    char ch[CHUNKSIZE];
    struct Chunk *next;
} Chunk;

typedef struct { // 串的链表结构
    Chunk *head, *tail; // 串的头和尾指针
    int curlen; // 串的当前长度
} LString;
```



在链式存储方式中，结点大小的选择直接影响着串处理的效率。这要求我们考虑串值的**存储密度**。

$$\text{存储密度} = \frac{\text{数据元素所占存储位}}{\text{实际分配的存储位}}$$

显然，存储密度小（如结点大小为1时），运算处理方便，然而，存储占用量大。实际应用时，可以根据问题所需来设置结点的大小。

串值的链式存储结构对某些串操作，如**拼接操作**等有一定**方便**之处，但总的说来不如另外两种存储结构灵活，它占用**存储量大且操作复杂**。实际应用很少。

4.3 串的模式匹配算法

子串的**定位操作**通常称作串的**模式匹配**，其中，子串被称作模式串。串的模式匹配是串的一种重要操作，很多软件，若有**“编辑”**菜单项的话，则其中必有**“查找”**子菜单项。

首先，回忆一下**串匹配**(查找)的定义：

INDEX (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： 若主串S中存在和串T值相同的子串返回它的主串S中第pos个字符之后第一次出现的位置；否则函数值为0。

```
int Index (String S, String T, int pos) {  
    // T为非空串。若主串S中第pos个字符之后存在与 T相等的子串，  
    // 则返回第一个 这样的子串在S中的位置，否则返回0  
  
    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 ; //返回子串在主串中的位置  
        } // while  
    } // if  
  
    return 0;  
} // Index
```

下面讨论以**定长顺序结构**表示串时的几种INDEX (S, T, pos) 算法。

一、简单算法

二、首尾匹配算法

三、KMP(D.E.Knuth, V.R.Pratt, J.H.Morris) 算法

一、简单算法

算法的基本思想：

从主串S的第pos个字符起和模式T的第一个字符比较之，若相等，则继续逐个比较后续字符，否则从主串的下一个字符起再重新和模式的字符比较之。依次类推，直至模式T中的每个字符依次和主串S中的一个**连续**的字符序列相等，则称**匹配成功**，函数值为和模式T中第一个字符相等的字符在主串S中的序号，否则称**匹配不成功**，函数值为零。

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

// 返回子串T在主串S中第pos个字符之后的位置。若不存在，
// 则函数值为0。其中，T非空， $1 \leq \text{pos} \leq \text{StrLength}(S)$ 。

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

```
    while (i <= S[0] && j <= T[0]) { //S[0]存储串的长度
```

```
        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
```

```
    }                                     //i指向比较完成后的下一个字符
```

```
    if (j > T[0]) return i-T[0];    //说明j已经比较完成
```

```
    else return 0;
```

算法时间复杂度为：

```
} // Index     $O(m \times n)$   n为主串的长度，  m为子串的长度，
```


第一趟匹配

↓ $i=3$
a b a b c a b c a c b a b
a b c
↑ $j=3$

第二趟匹配

↓ $i=2$
a b a b c a b c a c b a b (主串S)
a
↑ $j=1$

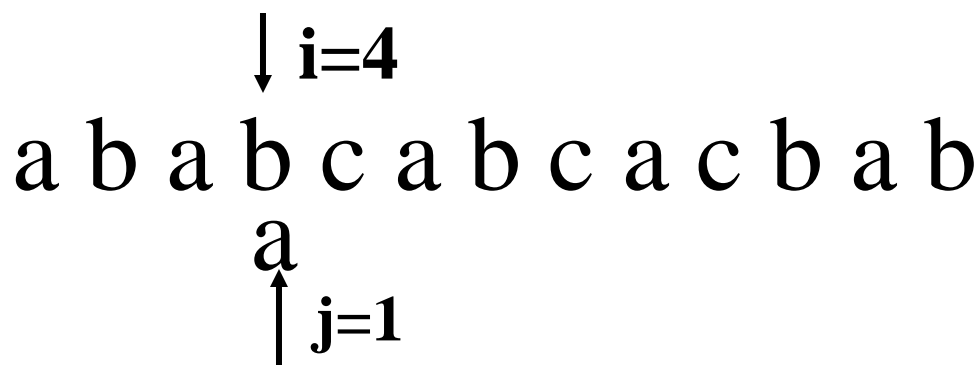
第三趟匹配

↓ i —————> ↓ $i=7$
a b a b c a b c a c b a b
a b c a c
↑ $j=1$ —————> ↑ $j=5$

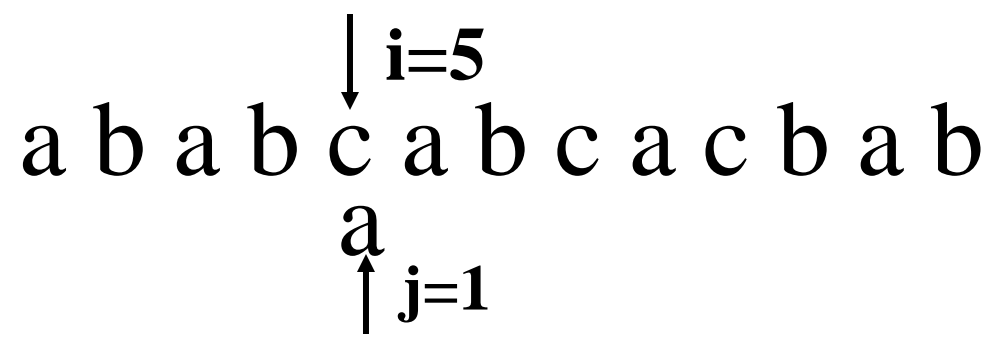
简单算法的匹配过程示例(模式串T='abcac')

简单算法的匹配过程示例
(T='ab
cac')

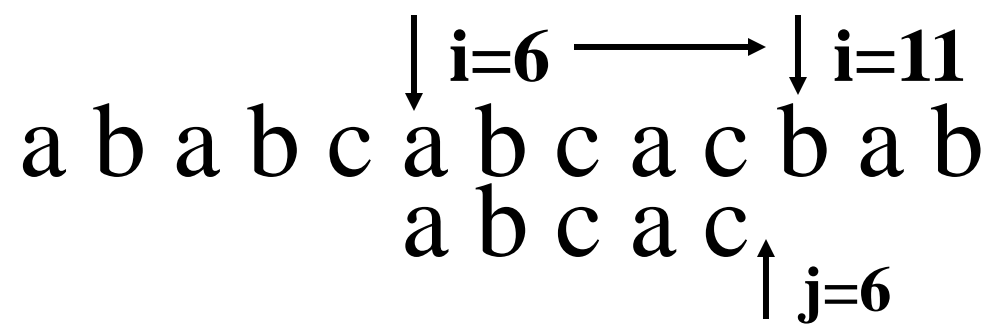
第四趟匹配



第五趟匹配



第六趟匹配



二、首尾匹配算法

先比较模式串的~~第一个~~字符，

再比较模式串的~~最后一个~~字符，

~~最后~~比较模式串中从第二个到
第 $n-1$ 个字符。

```
int Index_FL(SString S, SString T, int pos) {//首尾匹配

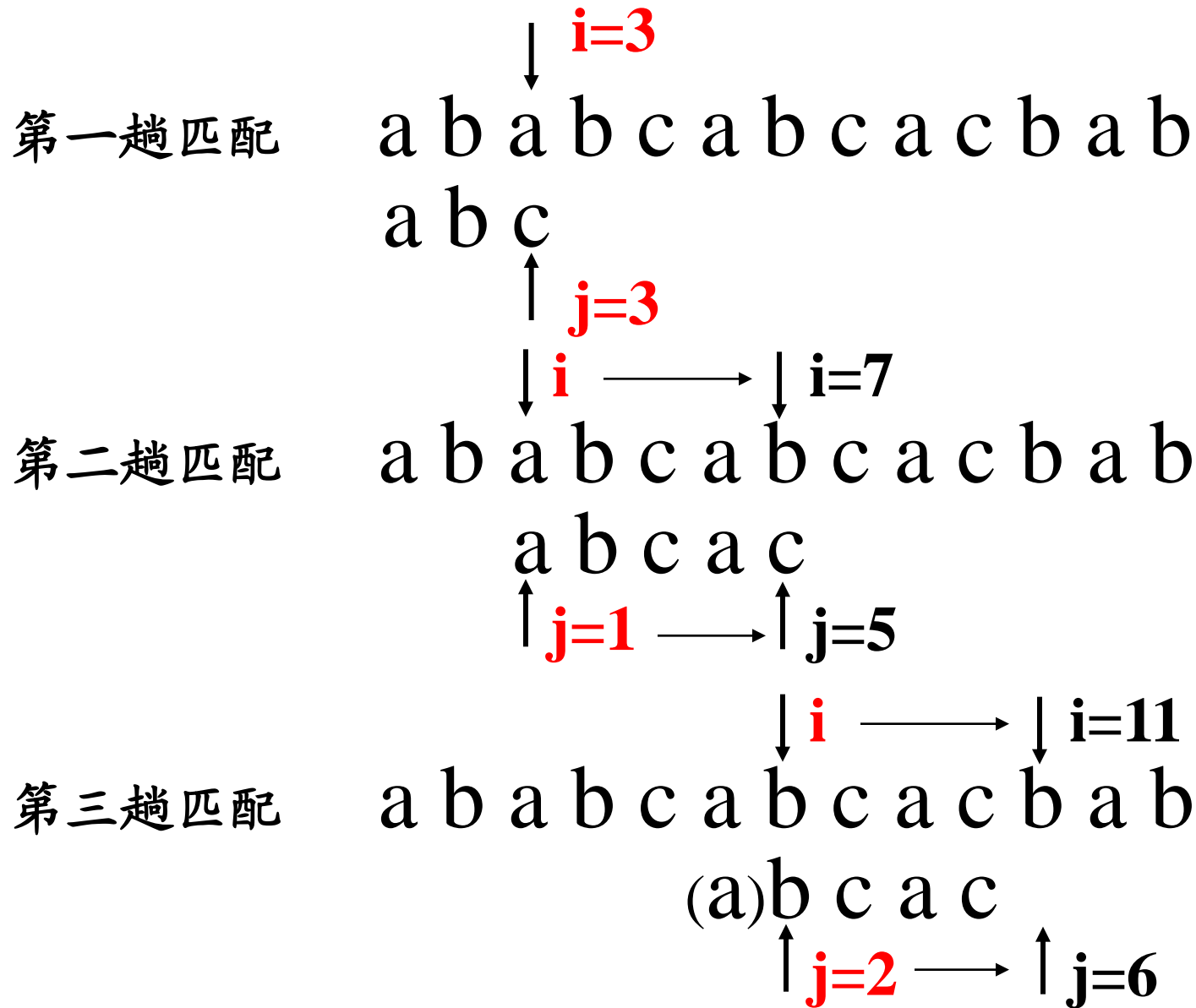
    sLength = S[0]; tLength = T[0]; i = pos;
    patStartChar = T[1]; patEndChar = T[tLength];
    while (i <= sLength - tLength + 1) {
        if (S[i] != patStartChar) ++i; //重新查找匹配起始点
        else if (S[i+tLength-1] != patEndChar) ++i;
            //模式串的“尾字符”不匹配
        else {k = 1; j = 2; //检查中间字符的匹配情况
            while ( j < tLength && S[i+k] == T[j])
                { ++k; ++j; }
            if ( j == tLength ) return i;
            else ++i; //重新开始下一次的匹配检测 } }
    return 0;
}
```

三、KMP (D. E. Knuth, V. R. Pratt, J. H. Morris) 算法

KMP算法的时间复杂度可以达到 $O(m+n)$

其改进在于：

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。



改进算法的匹配过程示例(T='abcac')

当 $S[i] \neq T[j]$ 时,
已经得到的结果:

$$S[i-j+1..i-1] == T[1..j-1]$$

若已知

$$T[1..k-1] == T[j-k+1..j-1]$$

则有

$$S[i-k+1..i-1] == T[1..k-1]$$

即模式中头 $k-1$ 个字符的子串 $T[1..k-1]$ 与主串中第 i 个字符之**前**长度为 $k-1$ 的子串 $S[i-k+1..i-1]$ 相等, 由此, 匹配仅需从模式中第 k 个字符与主串中第 i 个字符起继续进行比较。

若令 $next[j]=k$ ，则 $next[j]$ 表明当模式中第 j 个字符与主串中相应字符“失配”时，在模式中需要重新和主串中该字符进行比较的字符的位置。由此，

定义：模式串的 $next$ 函数

$$next[j] = \left\{ \begin{array}{ll} 0 & \text{当 } j = 1 \text{ 时} \\ \text{Max} \{k \mid 1 < k < j \\ \text{且 } 'p_1 p_2 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}' \} & \\ 1 & \text{其它情况} \end{array} \right\}$$

$$next[j] = \begin{cases} 0 & \text{当 } j = 1 \text{ 时} \\ \text{Max} \{k \mid 1 < k < j \\ \text{且 } 'p_1 p_2 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\} & \\ 1 & \text{其它情况} \end{cases}$$

根据上述定义，计算下列模式串的next函数的值

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
Next[j]	0	1	1	2	2	3	1	2

在求得模式的next函数之后，匹配可如下进行：
假设以指针i和j分别指示主串S和模式P中正待比较的字符，令i的初值为pos，j的初值为1。

若在匹配过程中， $S_i=p_j$ ，则i和j分别增1；
否则，i不变，而j退到next[j]的位置再比较，
若相等，则指针各自增1，否则j再退到下一个next值的位置，依次类推，直至下列两种可能：
一种是j退到某个next值时字符比较相等，则指针各自增1继续进行匹配；
另一种是j退到值为零（即模式的第一个字符“失配”），则此时需将模式继续向右滑动一个位置，即从主串的下一个字符 S_{i+1} 起和模式重新开始匹配。

T='abaabcac'

第一趟 主串
 模式

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

 ↓ i=2

 a b

 ↑ j=2 next[2]=1

第二趟 主串
 模式

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

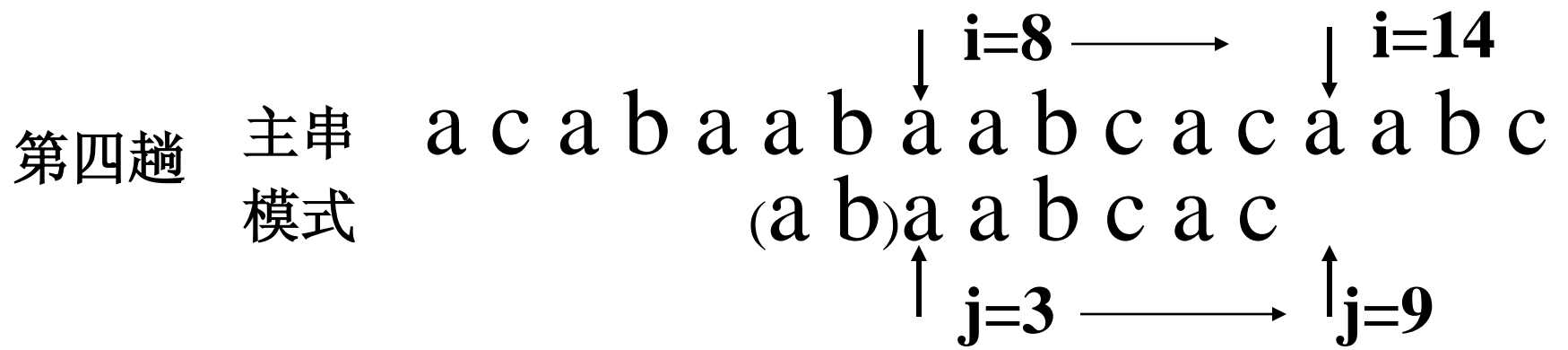
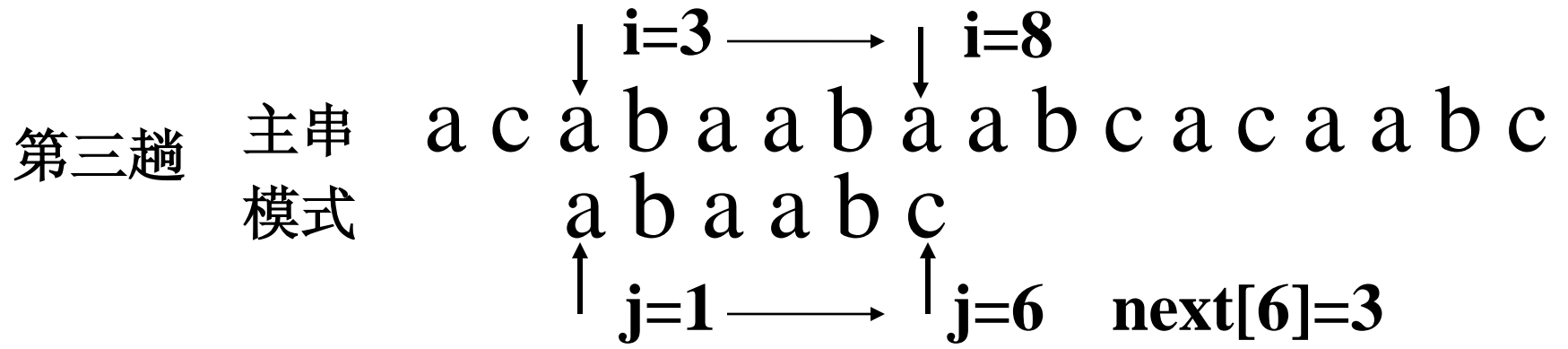
 ↓ i=2

 a

 ↑ j=1 next[1]=0

利用模式的next函数进行的匹配过程示例

T='abaabcac'



利用模式的next函数进行的匹配过程示例

此算法和第一种简单算法极为相似。不同之处仅在于：

当匹配过程中产生“失配”时，指针 **i 不变**，指针 **j 退回**到 `next[j]` 所指示的位置上重新进行比较，并且当指针 **j 退至零**时，指针 **i** 和指针 **j** 需同时增 1。即若主串的第 **i** 个字符和模式的第 1 个字符不等，应从主串的第 **i+1** 个字符起重新进行匹配。

```
int Index_KMP(SString S, SString T, int pos) {  
    //  $1 \leq \text{pos} \leq \text{StrLength}(S)$   
    i = pos; j = 1;  
    while (i <= S[0] && j <= T[0]) {  
        if (j = 0 || S[i] == T[j]) { ++i; ++j; }  
            // 继续比较后继字符  
        else j = next[j]; // 模式串向右移动  
    }  
    if (j > T[0]) return i-T[0]; // 匹配成功  
    else return 0;  
} // Index_KMP
```


算法时间复杂度为：

$O(m+n)$

改进的 NEXT: (还有一种特殊情况需要考虑)

例如: 主串 $S = \text{'aaabaaabaaabaaabaaab'}$

模式 $T = \text{'aaaab'}$



当 $i=4$, $j=4$ 时, $S_i \neq T_j$, 由 $\text{next}[j]$ 的指示还需进行 $i=4$, $j=3$, $i=4$, $j=2$, $i=4$, $j=1$ 等三次比较。实际上, 因为模式中第 1、2、3 个字符和第 4 个字符都相等, 因此不需要再和主串中第 4 个字符相比较, 而可以将模式一气向右滑动 4 个字符的位置直接进行 $i=5$, $j=1$ 时的字符比较。

$\text{next}[j] = 01234$

$\text{nextval}[j] = 00004$

第四章串总结

1. 熟悉串的六种基本操作的定义，并能利用这些基本操作来实现串的其它各种操作的方法。
2. 熟练掌握在串的定长顺序存储结构上实现串的各种操作的方法。
3. 了解串的堆存储结构以及在其上实现串操作的基本方法。
4. 理解串匹配的KMP算法，熟悉NEXT函数的定义，学会手工计算给定模式串的NEXT函数值和改进的NEXT函数值。