第4章 串

4.1 串类型的定义

4.2 串的表示和实现

4.3 串的模式匹配算法

本章重点难点

重点: (1)ADT串的设计、实现方法和基本操作；(2)串的简单模式匹配算法，KMP算法。

难点:串的模式匹配算法中的KMP算法。

第4章 串

4.1 串类型的定义

4.2 串的表示和实现

4.3 串的模式匹配算法

4.1 串类型的定义

□ 串(String)的定义

串是由零个或多个字符组成的有限序列。

记为：s=”a1a2…an” (n≥0)

其中，s是串的名，用双引号括起来的字符序列是串的值。

□ 串的有关术语

(1) 串的长度:串中字符的数目n。

(2) 空串(Null string):长度为零的串。

(3) 子串:串中任意个连续的字符组成的子序列。

串的有关术语

(4) 主串

包含子串的串相应地称为主串。

(5) 串相等

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，称两串相等。

(6) 空格串(空白串)(blank string)

由一个或多个空格组成的串。要和“空串”区别，空格串有长度就是空格的个数。

4.1 串类型的定义

□ 串与一般线性表的区别

(1) 串数据对象约束为字符集。

(2) 基本操作的对象不同，线性表以“单个元素”为操作对象；串以“串的整体”为操作对象，操作的一般都是子串。

4.1 串类型的定义

□ 串的ADT定义

ADT String {

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

数据关系：$R_1=\{ < a_{i-1}, a_i > \mid a_{i-1}, a_i \in D, i=2,...,n \}$

基本操作：见下页

} ADT String

4.1 串类型的定义

基本操作：

StrAssign (&T, chars) //根据串常量chars生成串T

DestroyString(&S) //销毁串S

StrCopy (&T, S) //把串S中内容拷贝到T串

StrLength(S) //求串长

StrCompare (S, T) //比较串S和T

Concat (&T, S1, S2) //连接串

StrEmpty (S) //判断串是否空

4.1 串类型的定义

- 基本操作：

SubString (&Sub, S, pos, len) //求子串

ClearString (&S) //清空串S

Index (S, T, pos) //子串定位

Replace (&S, T, V) //把串S中符合T的子串替换

StrInsert (&S, pos, T) //插入子串

StrDelete (&S, pos, len) //删除子串

4.2 串的表示和实现

4.2.1、定长顺序存储表示

4.2.2、堆分配存储表示

4.2.3、串的块链存储表示

串的顺序存储C语言实现

```
#define  MAXSTRLEN  255
     // 用户可在255以内定义最大串长
typedef unsigned char  Sstring[MAXSTRLEN+1];
      // 0号单元存放串的长度

Sstring  S;
```

串的连接算法

```
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;      }
          ……
  return uncut;
} // Concat
```

□ 串的连接算法

```
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。
    ……
       else if (S1[0] <MAXSTRSIZE) { // 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;  }
   return uncut;
 } // Concat
```

串的连接算法

```
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。
    ……
        else { // 截断(仅取S1)

          T[0..MAXSTRLEN] = S1[0..MAXSTRLEN];
                        // T[0] == S1[0] == MAXSTRLEN
            uncut = FALSE;          }

  return uncut;
} // Concat
```

子串的删除算法

```
Status StrDelete (SSstring &S, int pos, int len) {
  if (pos<1 || pos>S[0]||len<=0)
    return error;
  if (pos+len-1>=S[0])
    S[0]=pos-1;
  else{
   for(i=pos+len;i<=S[0];i++)
      {S[i-len]=S[i];
       S[0]=S[0]-len}
    }
 return OK;
}
```

4.2 串的表示和实现

4.2.1、定长顺序存储表示

4.2.2、堆分配存储表示

4.2.3、串的块链存储表示

4.2.2 堆分配存储表示

□ 堆分配存储表示的C语言实现

```
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;
```

4.2.2 堆分配存储表示

□ 堆分配存储表示的串连接算法

```
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);
  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
```

4.2.2 堆分配存储表示

□ 堆分配存储表示的求子串算法

```
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);                    // 释放旧空间
  if (!len)
    { Sub.ch = NULL;  Sub.length = 0; } // 空子串
  else {          ……..          }                 //  完整子串
  return OK;
} // SubString
```

- 堆分配存储表示的求子串算法

接上页

```
Sub.ch = (char *)malloc(len*sizeof(char));
Sub.ch[0..len-1] = S[pos-1..pos+len-2];
Sub.length = len;
```

4.2.3 串的块链存储表示

链表存储字符串的讨论

字符串本身就是一个线性表，可以用链表存储。

如果每个结点存储一个字符，如采用32位地址，字符按8位记，则存储密度是多少？

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

$$\text{存储密度} = \frac{8}{40} = 20\%$$

4.2.3 串的块链存储表示

□ 链表存储字符串的讨论

结论：采用普通链表存储字符串，存储密度非常低，浪费空间严重。

解决办法：一个结点存储多个字符。这就是串的块链存储。

4.2.3 串的块链存储表示

□ 串的块链存储的C语言实现

```
#define  CHUNKSIZE  80
 typedef  struct Chunk {  // 结点结构
   char  ch[CUNKSIZE];
   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;
```

4.3 串的模式匹配算法

4.3.1 模式匹配简单算法

4.3.2 模式匹配KMP算法

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≤pos≤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。

4.3.1 模式匹配简单算法

```
int Index(SString S, SString T, int pos) {
    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;
} // Index
```

4.3.1 模式匹配简单算法

□ 时间复杂性分析

讨论下面这种情况的时间复杂性，设S串长为n，T串长为m。

S：a b c d e f g h j k l l k c d e

T：j k l

假设从第i个位置匹配成功，前i-1趟共比较了i-1次。

第i趟比较了m次,共比较了i+m-1次。

i从1到n-m+1,共(n+m)(n-m+1)/2

平均需比较(n+m)/2

最好的情况平均时间复杂度为O(n+m)

4.3.1 模式匹配简单算法

□ 时间复杂性分析

讨论下面这种情况的时间复杂性，设S串长为n，T串长为m。

S：0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

T：0 0 0 0 0 0 1

若第i趟比较成功，共比较了多少次？

前i-1趟比较每次都比较m次，第i趟也比较m次

共im次， i从1到n-m+1

共比较了(n-m+2)(n-m+1)m/2

平均比较次数(n-m+2)m/2

最坏的情况时间复杂度为O(n×m)

4.3 串的模式匹配算法

4.3.1 模式匹配简单算法

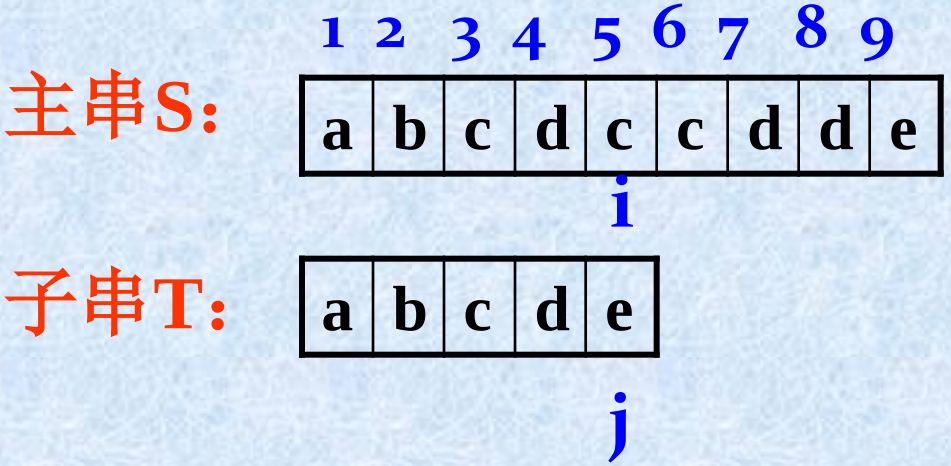
4.3.2 模式匹配KMP算法

4.3.1 模式匹配KMP算法

事例讨论

例

下图中主串游标i指向5，子串中游标j指向5，按照简单模式匹配算法两处不相等时j回到1，i回到2，继续比较，分析在这种情况下，这样做有没有意义？



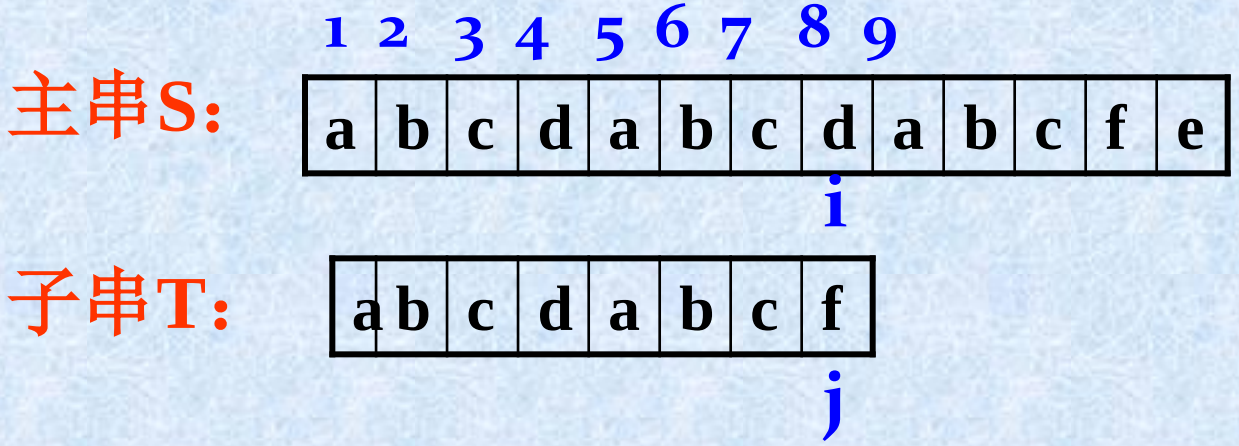
结论：没有意义。

4.3.1 模式匹配KMP算法

事例讨论

例

下图中主串游标i指向8，子串中游标j指向8，按照简单模式匹配算法两处不相等时j回到1，i回到2，继续比较，分析在这种情况下，这样做有没有意义？在什么情况下才有意义？

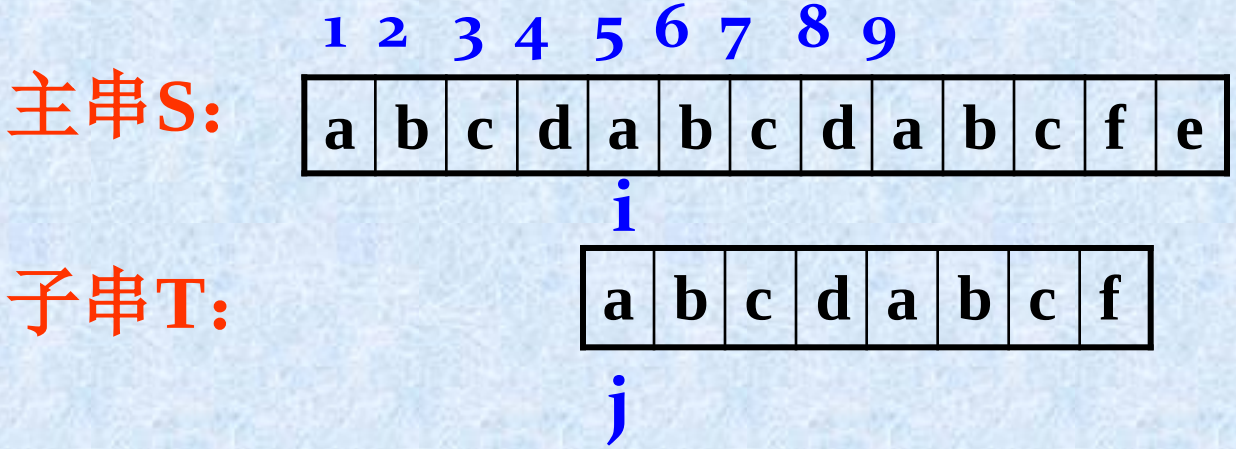


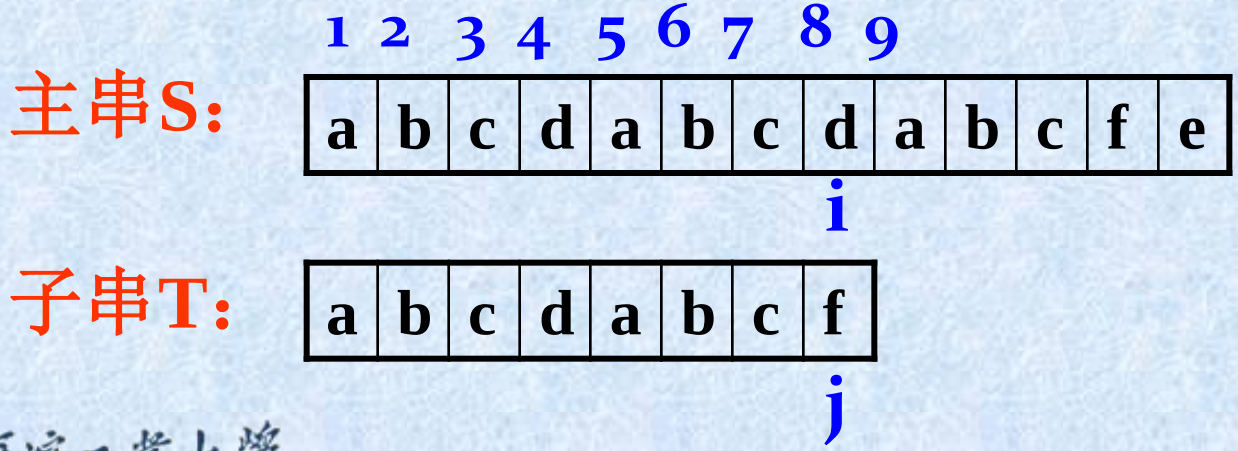
结论：j回到1，i回到2没有意义。

4.3.1 模式匹配KMP算法

事例讨论

结论：只有i回到5，j回到1才有意义。如下图。



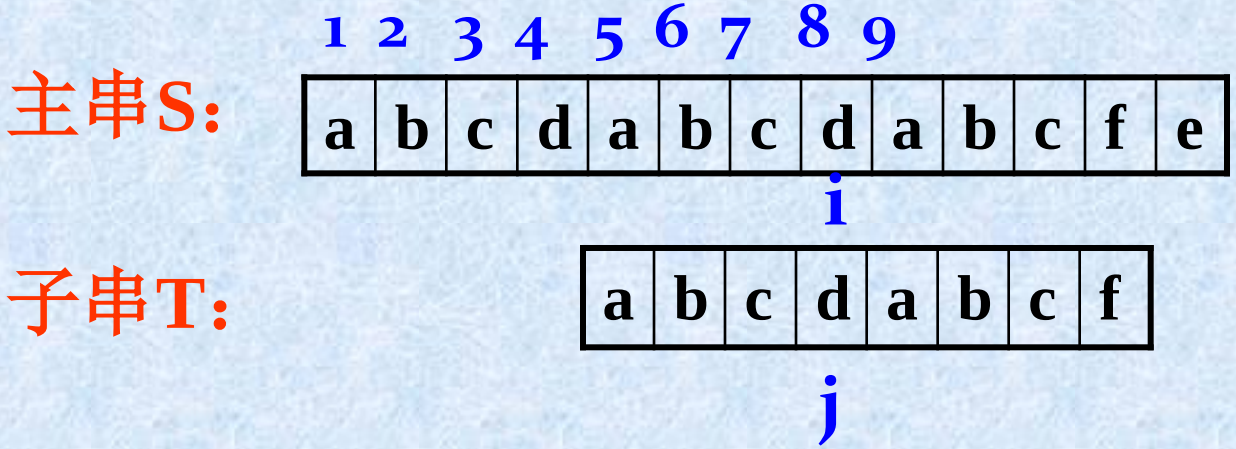


与原来的比较图进行对比，看有什么发现？

4.3.1 模式匹配KMP算法

事例讨论

结论：可以i不动，j回到4。如下图。



KMP算法的思想

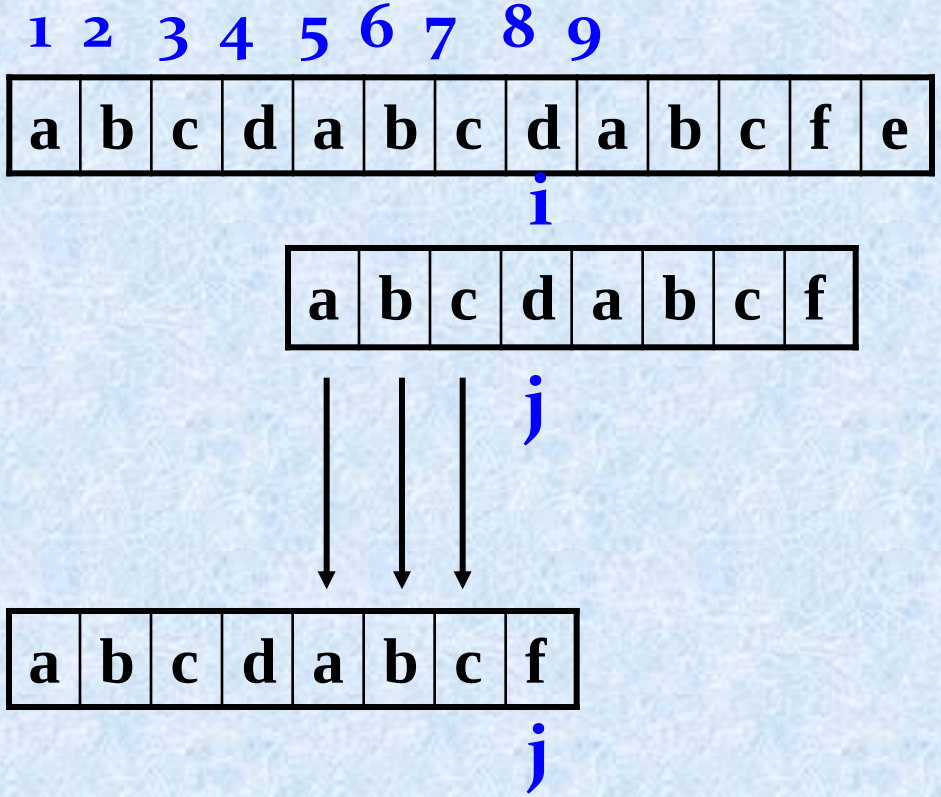
主串指针不回溯，模式串向后滑动至某个位置上。

KMP算法的时间复杂度可以达到O(m+n)

4.3.1 模式匹配KMP算法

- 子串游标滑动到k必须满足的条件

结论：可以i不动，j回到4。如下图。



与原来的比较图进行对比，看有什么发现？

4.3.1 模式匹配KMP算法

□ 子串游标滑动到k必须满足的条件

主串S: …S_{i-j} S_{i-j+1} S_{i-j+2} … S_{i-2} S_{i-1} S_i S_{i+1}…

子串T: t_1 t_2 t_{j-2} t_{j-1} t_j

假如j滑动到k,即从t的k处和S_i比较有意义:必须满足:

主串S: …S_{i-j} S_{i-j+1} S_{i-j+2} …S_{i-k+1} S_{i-k+2} … S_{i-2} S_{i-1} S_i S_{i+1}…

子串T: t_1 t_2 … t_{j-k+1} t_{j-k+2} … t_{j-2} t_{j-1} t_j

子串T: t_1 t_2 … t_{k-2} t_{k-1} t_k t_{k+1}…

即 “$t_1t_2…t_{k-1}$” = “$t_{j-k+1}t_{j-k+2}…t_{j-1}$” (真子串)

具体含义是$t_{1...j-1}$的前k-1个字符构成的前缀真子串和后k-1个字符构成的后缀真子串相等，不过$t_{1...j-1}$的相等的前后缀真子串可能有多对，k取最长的那对的长度+1

next[j]函数

表明当模式中第j个字符与主串中相应字符“失配”时，在模式中需重新和主串中该字符进行比较的字符的位置。

$$\text{next}[j]=\begin{cases}\max\{k \mid 1<k<j, \text{且“}t_1\cdots t_{k-1}\text{”}=\text{“}t_{j-k+1}\cdots t_{j-1}\text{”}\} & \text{当此集合非空时}\\ 0 & \text{当 } j=1 \text{ 时}\\ 1 & \text{其他情况}\end{cases}$$

注：$j=1$时，$T_{1\ldots j-1}$不存在。

这个时候i，j都要后移

其他情况是指：表示j>o时候不存在相同的最长前后缀串，下一次j从1号位置开始，i不变。

4.3.1 模式匹配KMP算法

□ KMP算法

```
int Index_KMP (SString S,SString T, int pos)
{     i= pos,j =1;
      while (i<=S[0] && j<=T[0])
         if (j==0 || S[i]==T[j])  { i++;j++; }
           else
              j=next[j];                    //i不变,j后退
         if (j>t[0])  return i-t[0];  //匹配成功
      else   return 0;                      //返回不匹配标志
}
```

若next[j] = o，则表示T中任何字符都不必在与s_j进行比较，下次比较从s_{i+1}与T_1开始。

求next函数值

(1) next[1] = 0;表明主串从下一字符s_{i+1}起和模式串重新开始匹配。i = i+1; j = 1;

(2) 设next[j] = k，则next[j+1] = ?

① 若$t_k=t_j$，则有“$t_1 \cdots t_{k-1} t_k$”=“$t_{j-k+1} \cdots t_{j-1} t_j$”，如果在j+1发生不匹配:

说明next[j+1] = k+1 = next[j]+1。

② 若$t_k \neq t_j$，可把求next值问题看成是一个模式匹配问题，整个模式串既是主串，又是子串。

求next函数值

求next[j+1],如果t[j]=t[k];则next[j+1]=k+1;

如果t[j] ≠t[k]

若$t_{k'}=t_j$，则有“$t_1 \ldots t_{k'}$”=“$t_{j-k'+1} \ldots t_j$”，

next[j+1]=k'+1=next[k]+1=next[next[j]]+1.

若$t_{k''}=t_j$，则有“$t_1 \ldots t_{k''}$”=“$t_{j-k''+1} \ldots t_j$”，

next[j+1]=k''+1=next[k']+1=next[next[k]]+1.

next[j+1]=1.

4.3.1 模式匹配KMP算法

□ 求next函数算法

```
void get_next(SString T,int next[])
  {  int j=1,k=0;
     next[1]=0;
     while(j<T[0]){
        if(k==0||T[j]==T[k])
          {++j; ++k; next[j]=k;}/*next[++j]=++k*/
        else  k=next[k] ;/*T[j]≠T[k]时，k跳转到next[k]*/
      }
  }
```

if中代码的含义就是在next[j]=k的情况下,如果T[j]==T[k],则next[j+1]=k+1或者k=0时，让next[j+1]=1，表示j>1时候不存在相同的最长前后缀串即++k后k=1即next公式的其他情况

初值j=1，k=0：表示j=1时之前不存在最长的前后缀串，即next[1]=0，这种情况下表示一开始就匹配不上，是迭代计算next值的基础。

4.3.1 模式匹配KMP算法

□ 求next函数的改进算法

还有一种特殊情况需要考虑：

例如：

S = 'aaabaaaabaaaabaaaabaaaab'

T = 'aaaab'

next[j]= 01234

nextval[j]=00004

4.3.1 模式匹配KMP算法

□ 求next函数的改进算法(续)

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

单选题 1分

设置

给出字符串abaabab的next值(未经优化，下标从1开始计数)

A 0112234

B 0001123

C 0112233

D 0223345

提交

本章小结

熟练掌握：

(1)串的定义、性质和特点；

(2)ADT串的设计、实现方法和基本操作；

(3)朴素模式匹配算法；