第四章 字符串/串(string)

“串”的学习目标

- 熟悉串的有关概念，串和线性表的关系。
- 掌握串的各种存储结构，比较它们的优、缺点。
- 熟练掌握串的七种基本运算，并能利用这些基本运算实现串的其它各种运算。

4.1 串的定义与操作

1.术语

(1)串：由零个或多个字符组成的有限序列。

n个字符$C_1, C_2, \ldots, C_n$组成的串记作：

$s='C_1C_2\ldots C_n'$　　$n>=0$

其中：s为串名，　$C_1C_2\ldots C_n$为串值，　n为串长

例 PASCA语言：　$s_1='data1234'$　　$s_2='123*abc'$

C语言：　　$s_1="data1234"$　　$s_2="123*abc"$

'A' 为字符　　"A"为字符串

%c为字符格式　　%s为字符串格式

(2) 空串：不含字符的串/长度为零的串。

PASCA语言： s=’’

C语言： s=""

(3) 空格串：仅含空格字符’ ’的串。

例 s_1="Φ" s_2="ΦΦ"

s_1=" " s_2=" "

(4) 子串：串s中任意个连续的字符组成的子序列称为串s的子串。

主串---包含某个子串的串。

例 st="ABC123A123CD"

s_1="ABC" s_3="123A" s_4="ABCA"

s_2="ABC12" s_5="ABCE" s_6="321CBA"

特别地：空串是任意串的子串，任意串是其自身的子串。

(5) 子串的位置：

子串t 在主串s 中的位置是指主串s 中第一个与t 相同的子串的首字母在主串中的位置;

例：s="ababcabcac"，t="abc"， 子串t 在主串s 中的位置为3

(6) 串相等：

两个串相等，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。

- 串是一种特殊的线性表
- 串与一般线性表的区别：
  - 串的数据元素约束为字符集；
  - 串的基本操作通常针对串的整体或串的一个部分进行。
- 为何要单独讨论“串”类型？
  - 字符串操作比其他数据类型更复杂（如拷贝、连接操作）
  - 程序设计中，处理对象很多都是串类型，应用很广。

串的抽象数据类型定义

ADT String{

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

数据关系：$S=\{< a_{i-1}, a_i > | a_{i-1},\ a_i \in D,\ i = 2,\ldots,n\}$

基本操作：

StrAssign(&T,chars)	*Index(S,T,pos)*
StrLength(S)	*StrEmpty(S)*
SubString(&Sub,S,pos,len)	*Replace(&S,T,V)*
StrCopy(&T,S)	*StrInsert(&S,pos,T)*
StrCompare(S,T)	*StrDelete(&S,pos,len)*
Concat(&T,S1,S2)	……

} ADT String

在上述抽象数据类型定义的操作中，其中

串赋值StrAssign、　　串比较StrCompare、

求串长StrLength、　　串联接Concat、

以及求子串SubString

等五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直接来实现，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。

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。

SubString (&Sub, S, pos, len)

初始条件：串 S 存在，1≤pos≤StrLength(S)
且0≤len≤StrLength(S)-pos+1。

操作结果：用 Sub 返回 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，

则返回它在主串 S 中第pos个字符之后

第一次出现的位置；

否则函数值为0。

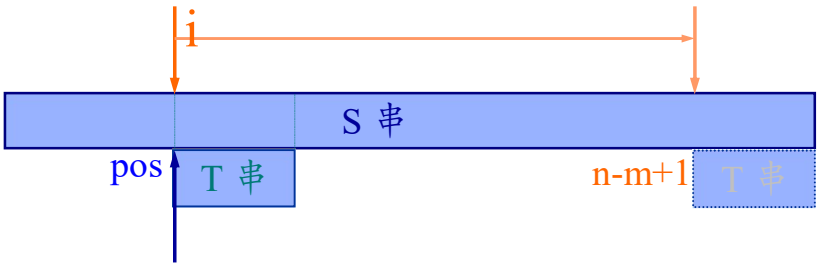
可利用串比较、求串长和求子串等操作实现定位函数

Index(S,T,pos)。

算法的基本思想为：

SubString(sub,S, i, StrLength(T))

StrCompare(sub,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 ;
   } // while
  } // if
  return 0;                          // S中不存在与T相等的子串
} // Index
```


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)	**串复制函数；**
strncpy(str1, str2, n)	**串复制函数；**
strcmp(str1, str2)	**串比较函数；**
strlen(str)	**求串长函数；**

2.C语言串变量、字符变量的定义与使用

例1 串变量

```
char st[]="abc\'*123";
gets(st);  scanf("%s",st);
strcpy(st,"data");
puts(st);  printf("st=%s\n",st);
```

例2 字符变量

```
char ch='A';
ch=' B' ;  ch=getchar();
scanf("%c",&ch);  printf("ch=%c\n",ch);
```


3.串的基本操作与串函数(C语言标准库函数，string.h)

(1)strcpy(t,s)---s的串值复制到t中。

执行：strcpy(t,"data"); 有：t="data"

(2)strlen(s)----求s的长度

strlen("data*123")=8 strlen("")=0

(3)strcat(s1,s2)----s2的值接到s1的后面。

设 s1="data" ， s2="123"

执行：strcat(s1,s2);

有： s1="data123" ， s2="123"

(4) strcmp(s1,s2)---比较s1和s2的大小

若s1<s2,返回负整数　　如："ABC"<"abc"

若s1=s2,返回0如：　　如："abc"="abc"

若s1>s2,返回正整数　　如："ABCD">"ABC"

(5) strstr(s1,s2)----若s2是s1的子串，则指向s2在s1中第1次出现时的第1个字符的位置；否则返回NULL。

设 s1="ABE123*DE123bcd"

s2="E123"

有 strstr(s1,s2)=3

s1="ABE123*DE123"

(6) Replace(s,t,v)----置换

用v代替主串s中出现的所有与t相等的不重叠的子串。

设 s="abc123abc*ABC", t="abc", v="OK"

执行：Replace(s,t,v);

有：　s="OK123OK*ABC", t="abc", v="OK"

设 A="abcaaaaaABC", B="aa", C="aaOK"

执行：Replace(A,B,C);

有：　A="abcaaOKaaOKaABC", B="aa", C="aaOK"

(7) StrInsert(s,i,t)----将t插入s中第i个字符之前。

设　　s="ABC123"

执行: StrInsert(s,4,"**");

有:　　s="ABC**123"

(8)用Replace(s,t,v)实现删除

设　　s="ABC//123"

执行：Replace(s,"//","")

有：　s="ABC123"

(9)用Replace(s,t,v)实现插入

设　　s="ABC123"

执行：Replace(s,"123","**123")

有：　s="ABC**123"

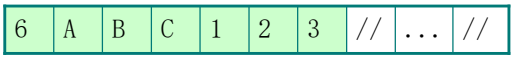
4.2 串的表示和实现

串的物理表示方法：

- 顺序存储：
  - 定长顺序存储表示（静态数组结构）
  - 堆分配存储表示（动态数组结构）
- 链式存储：串的块链存储表示

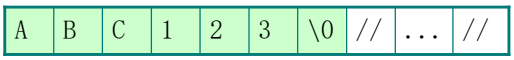
4.2.1 串的定长顺序存储表示

PASCAL: 下标为0的分量存放串的实际长度



0 1 2 255

C: 在串值后加串结束标记‘\0’,串长为隐含值



0 1 2 255

顺序存储(C语言方式)----用一维字符数组表示一个串的值。

例1 char st_1[80]=”ABC123”;

0 1 2 3 4 5 6 ... 79

例2 char st_2[]=”ABC123”;

0 1 2 3 4 5 6

例3 char st[20];

0 1 2 3 4 5 6 ... 19

串的定长顺序存储：给每个定义的串分配一个固定长度的存储区。

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

特点：

- 串的实际长度可在这个预定义长度的范围内随意设定，超过预定义长度的串值则被舍去，称之为“截断” 。
- 按这种串的表示方法实现的串的运算时，其基本操作为“字符序列的复制”，是pascal语言的串的方式。

2. 串运算实现举例

例 联接运算：给定串s_1, s_2, 将s_1, s_2联接为串t, 记作：

Concat(&t, s_1, s_2)

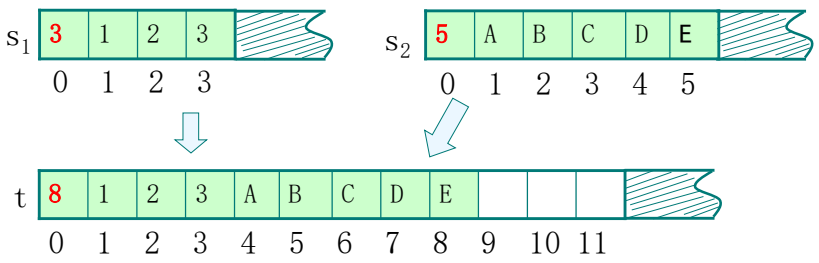
设 s_1=”123”, s_2=”ABCDE”

执行： Concat(t, s_1, s_2)； 有： t=”123ABCDE”

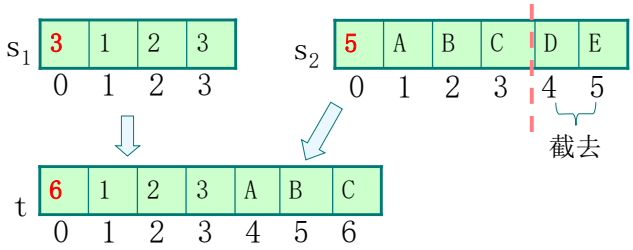
实现Concat(&t, s_1, s_2)的方法（Pascal语言方式）：

(1) s_1的长度+s_2的长度≤t的最大长度

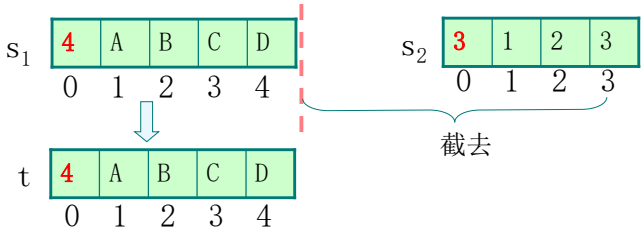
（注意：统一串空间最大长度为MAXSTRLEN）



(2) s_1的长度≤t的最大长度≤s_1的长度+s_2的长度：



(3) t的最大长度=s_1的长度=MAXSTRLEN

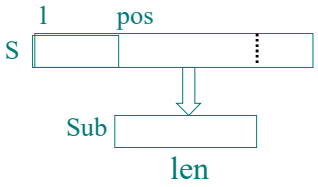


基于统一类型*SString*实现Concat的算法4.2(p73)


```
Status Concat(SString &T,SString S1,SString S2){
  if (S1[0]+S2[0]<=MAXSTRLEN){   // 两个字符串的长度之和小于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){     /*两个字符串的长度之和大于MAXSTRLEN */
       T[1 …S1[0]]=S1[1 …S1[0]];
       T[S1[0]+1 …MAX]=S2[1…MAX-S1[0]];
       T[0]=MAXSTRLEN; uncut=FALSE; }
    else {  T[0...MAXSTRLEN]=S1[0…MAXSTRLEN]; uncut=FALSE;}
                                  // T[0] == S1[0] == MAXSTRLEN
  return OK;
  }
```

求子串算法：复制字符序列

```
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;
}//SubString
```



4.2.2 堆分配存储和实现

特点：

用一组连续的存储单元来存放串，但存储空间是在程序执行过程中动态分配而得，堆空间为多个串共享。

串长无限制，更灵活。

思路：

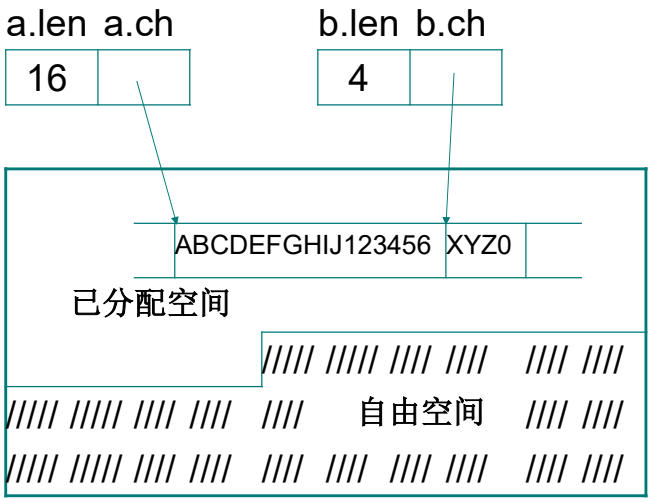
利用malloc函数合理预设串长空间。

串值的确定是通过串在堆中的起始位置和串长实现的。

串操作实现算法为：

先为新生成的串分配一个存储空间，然后进行串值的复制。

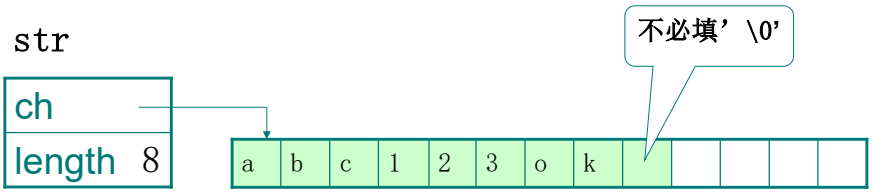
示例



堆存储结构描述,定义将串长作为存储结构的一部分：

```
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;
HString  str;
```

用以表示字符串“abc123ok”



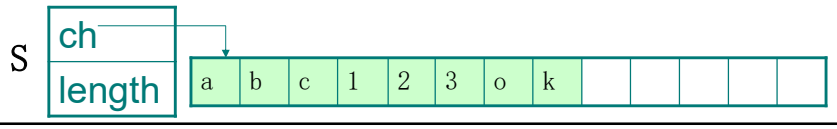
基本操作：

串的赋值算法：生成一个其值等于串常量chars的串T

```
Status StrAssign(HString &T,char *chars){
   if (T.ch)  free (T.ch);
   for (i=0,c=chars;c;++i,++c);
       // c指针自增至null,求chars的长度i
   if (!i) {T.ch=NULL; T.length=0;} //当chars为空串时
   else {
     T.ch=(char*)malloc(i*sizeof(char));
     if (!T.ch)  exit OVERFLOW;
     T.ch[0...i-1]=chars[0...i-1]; //T.ch[0]没有用来表示串长
     T.length=i;
        }
    return OK;
}
```

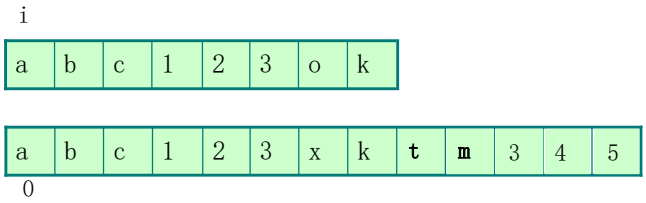

用“堆”方式编写串插入函数

```
Status StrInsert ( HString &S,  int pos,  HString T )
{   //在串S的第pos个字符之前（包括尾部）插入串T
if (pos<1||pos>S.length+1) return ERROR;   //pos不合法则告警
  if(T.length){              //只要串T不空，就需要重新分配S空间，以便插入T
     if (!（S.ch=(char*)realloc(S.ch, (S.length+T.length)*sizeof(char))))
          exit(OVERFLOW);        //若开不了新空间，则退出
     for ( i=S.length-1;  i>=pos-1; --i ) S.ch[i+T.length] = S.ch[i];
          // 为插入T而腾出pos之后的位置，即从S的pos位置起全部字符均后移
     S.ch[pos-1…pos+T.length-2] = T.ch[0…T.length-1];  //插入T
     S.length + = T.length;          //刷新S串长度
   } return OK;
} //StrInsert
```



串的比较算法

```
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
  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;
}
```



串的连接算法:

```
Status Concat(HString &T,HString S1,HString S2)
{                    /* 用T返回由S1和S2联接成的新串 */
  if (T.ch) free(T.ch);                  //释放旧空间
  T.ch=(char*)malloc((S1.length+S2.length)*sizeof(char))
  if (!T.ch) 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;
}
```

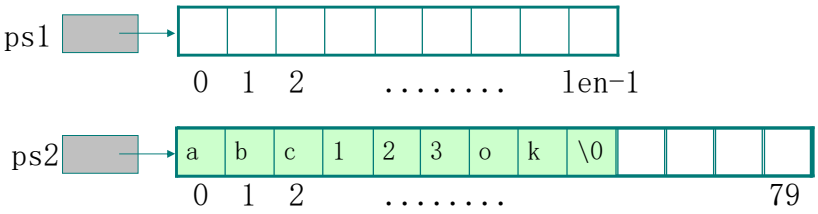
取子串算法:

```
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leng的子串
    if (pos<1 ||pos>S.length||len<0||len>S.length-pos+1)
        return ERROR;
    if (Sub.ch) free(Sub.ch);
    if (!len) {Sub.ch=NULL;Sub.length=0;}  //空串
    else {
       Sub.ch=(char *)malloc(len*sizeof(char));
       Sub.ch[0...len-1]=S[pos-1...pos+len-2];
       Sub.length=len;
    }
    return OK;
}
```


- C语言中提供的串类型就是利用函数malloc( )和 free( )进行串值空间的动态管理，为每一个新产生的串分配一个存储区，称串值共享的存储空间为“堆”。
- 串用字符指针表示。
- C语言中的串以空字符‘\0’为结束符，串长是隐含值。

例1：C语言中利用动态分配使用系统堆。

```
{ char *ps1,*ps2; int len;
  scanf("%d",&len);                  //输入长度值
  ps1=(char *)malloc(len);           //ps1指向分配的存储空间
  gets(ps1);  puts(ps1);             //输入一个串，再输出
  ps2=(char *)malloc(80);            //ps2指向分配的存储空间
  strcpy(ps2,"abc123ok") ;           //赋值，再输出
  puts(ps2);
  free(ps1); free(ps2);              //释放存储空间
 }
```



例2：基于堆串表示输出字符串

```
void StrPrint(HString T)
{
 int i;
 for(i=0;i<T.length;i++)
     putchar(T.ch[i]);
 }

void  main(void)
{
  HString str;
  StrAssign(&str,” abcd123”);
  StrPrint(str);
 }
```

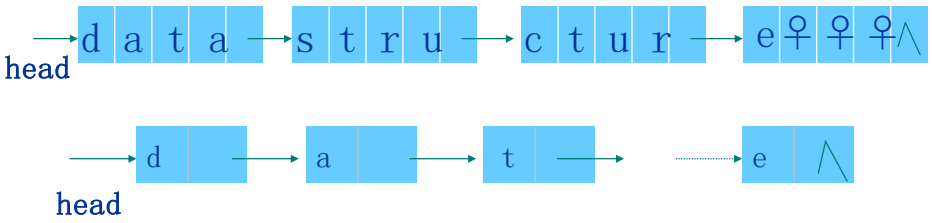
4.2.3 串的单链表表示

链式存储结构

链式存储结构类似线性链表，由于串结构的特殊性,要考虑每个结点是存放一个字符还是多个字符。

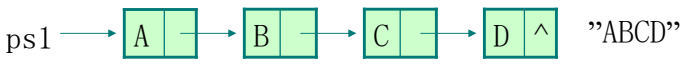
一个字符的（单字符结点链），插入、删除、求长度非常方便，但存储效率低。

多个字符的（块链结构），改善了效率，在处理不定长的大字符串时很有效，但插入、删除不方便，可用特殊符号来填满未充分利用的结点。



例1 一个结点只放1个字符

```
struct node1
 { char data;            //为一个字符
   struct node1 *next;   //为指针
  }*ps1;
```



存储密度=串值所占存储位/实际分配存储位

存储密度为 0.33

例2 一个结点放4个字符

```
struct node4
{ char data[4];              //为4个字符的串
  struct node4 *next;       //为指针
 }*ps2;
```

存储密度为 0.67

串的块链存储表示：

```
#define CHUNKSIZE 80          // 可由用户定义的块大小
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

求子串位置的定位函数Index(S, T, pos)

- 模式匹配：子串的定位操作通常称作串的模式匹配。
  - ✓目标串：主串S。
  - ✓模式串：子串T。
  - ✓匹配成功：若存在T的每个字符依次和S中的一个连续字符序列相等，则称匹配成功。返回T中第一个字符在S中的位置。
  - ✓匹配不成功：返回0。
- 算法目的：确定主串中所含子串第一次出现的位置。
- 典型算法：
  - ✓*Brute-Force (BF)*算法：穷举法，带回溯，速度慢。
  - ✓*KMP*算法：避免回溯，匹配速度快。

BF算法的实现：Index(S, T, pos)函数

例： S='ababcabcacbab'，T='abcac'，pos=1，

求：串T在串S中第pos个字符之后的位置。

BF算法设计思想：

- 将主串S的第pos个字符和模式T的第1个字符比较，
  - ✓ 若相等，继续逐个比较后续字符；
  - ✓ 若不等，从主串S的下一字符（pos+1）起，重新与T第一个字符比较。
- 直到主串S的一个连续子串字符序列与模式T相等。
  - ✓返回值为S中与T匹配的子序列第一个字符的序号，即匹配成功。
- 否则，匹配失败，返回值 0 .

利用演示系统看BF算法执行过程。

下面算法以定长的顺序串类型作为存储结构

```
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≤pos≤StrLength(S)。
    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
```

BF算法的时间复杂度

讨论：

若n为主串长度，m为子串长度，则串的BF匹配算法最坏的情况下需要比较字符的总次数为：　(n-m+1)*m＝O(n*m)

最好的情况是：一配就中！　只比较了m次。

最坏的情况是：主串前面(n-m)个位置都部分匹配到子串的最后一位，即这n-m位比较了m次，最后m位也各比较了一次，因此总次数为：

(n-m)*m+m ＝(n-m+1)*m

平均时间复杂度是：O(n+m)

依次从最好到最坏情况统计总的比较次数，然后取平均。

BF算法时间复杂度的具体分析（设pos=1）

主串长n； 子串长m。可能匹配成功的位置（1 ~ n-m+1)。

①最好的情况下，

第i个位置匹配成功，比较了（i-1+m）次，平均比较次数：

最好情况下算法的平均时间复杂度O(n+m)。

②最坏的情况下，

第i个位置匹配成功，比较了（i*m）次，平均比较次数：

设n>>m，最坏情况下的平均时间复杂度为O(n*m)。

$$\sum_{i=1}^{n-m+1} p_i(i-1+m) = \frac{1}{n-m+1}\sum_{i=1}^{n-m+1}(i-1+m) = \frac{1}{2}(m+n)$$

$$\sum_{i=1}^{n-m+1} p_i(i\times m) = \frac{m}{n-m+1}\sum_{i=1}^{n-m+1} i = \frac{1}{2}m(n-m+2)$$

如何优化模式匹配的速度？

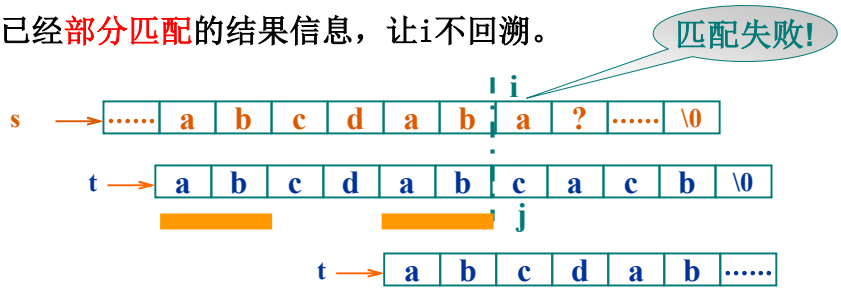
利用已部分匹配过的信息使主串S的指针i不回溯，最坏情况也能达到O(n+m)。

- 注意：
  - ✓ 当遇到 $s_i \neq t_j$,主串要回退到$i = i-j+2$的位置，而模式串要回到第一个位置（即 $j=1$ 的位置);
  - ✓ 但当一次比较出现 $s_i \neq t_j$ 时，则应该有：

    "$s_{i-j+1}s_{i-j+2}\cdots\cdots s_{i-1}$"="$t_1t_2\ \cdots\cdots t_{j-2}t_{j-1}$"
- 改进：
  - ✓ 每当一趟匹配过程出现 $s_i \neq t_j$ 时，主串指示器i 不用回溯，而是利用已经得到的“部分匹配”结果，将模式串向右“滑动”到适当位置后，继续进行比较。
- KMP算法

KMP算法设计思想：

利用已经部分匹配的结果信息，让i不回溯。



当匹配过程“失配”时，模式串“向右滑行” 的距离应是多远？

讨论一般情况:

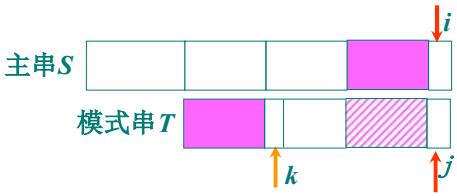
设: 主串S=″$s_1s_2…s_i…s_n$″

模式串T=″$p_1p_2…p_j…p_m$″

问：当某趟比较发生“失配”（即$s_i \neq p_j$）时，模式串应该向“右”滑动的可行距离为多长？

- 设某趟匹配发生$s_i \neq p_j$时
  - ✓ S_i应该与 p_k（$k < j$）继续比较
  - ✓ 根据：″$p_1p_2…p_{k-1}$″ ＝″$s_{i-k+1}s_{i-k+2}…s_{i-1}$″

    ″$p_{j-k+1}p_{j-k+2}…p_{j-1}$″ ＝″$s_{i-k+1}s_{i-k+2}…s_{i-1}$″
  - ✓ 可得： “$p_1p_2…p_{k-1}$” ＝ “$p_{j-k+1}p_{j-k+2}…p_{j-1}$”
- 图示如下：



如何确定模式向右滑动的新比较起点k？

令：$next(j)=k$

$$next(j)=\begin{cases}0 & \text{当 } j=1; \quad \text{// } i\text{后移, } j\leftarrow 1\\ Max\{k \mid 1<k<j \text{ 且 } \text{“}p_1p_2\cdots p_{k-1}\text{”}=\text{“}p_{j-k+1}p_{j-k+2}\cdots p_{j-1}\text{”}\} & \text{当此集合非空; } \text{// } i \text{ 不回溯, } j\leftarrow k\\ 1 & \text{其他情况。} \quad \text{// } i \text{ 不回溯, } j\leftarrow 1\end{cases}$$

例1：计算如下模式串的*next*函数值。

j	1	2	3	4	5
P_j	a	b	c	a	c
next(j)	0	1	1	1	2

例2：计算如下模式串的*next*函数值。

j	1	2	3	4	5	6	7	8
P_j	a	b	a	a	b	c	a	c
next(j)	0	1	1	2	2	3	1	2

讨论：

（1） next[j]的物理意义是什么？

（2） next[j]具体怎么求？

next[j]函数表征着模式串中最大相同前缀子串和后缀子串（真子串）的长度（+1）。

计算Next[j]的方法：

•当j=1时，Next[j]=0；

//Next[j]=0表示根本不进行字符比较

•当j>1时，Next[j]的值为：模式串的位置从1到j-1构成的串中所出现的首尾相同的子串的最大长度加1。

无首尾相同的子串时Next[j]的值为1。

// Next[j]=1表示从模式串头部开始进行字符比较

怎样计算模式串所有可能的失配点 j 所对应的 next[j]?

模式串的next函数值的求解

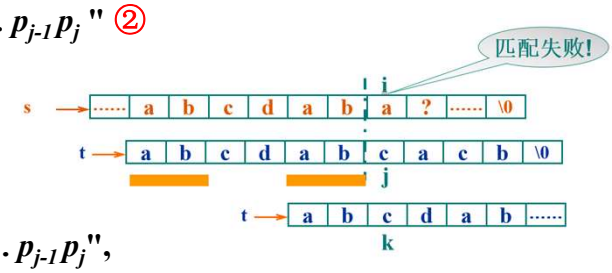
- 由定义知：*next*[1]=0，
- 设*next*[*j*]=*k*
- 表明："$p_1p_2\cdots p_{k-1}$"="$p_{j-k+1}p_{j-k+2}\cdots p_{j-1}$"

  （其中$1<k<j$, 且不存在$k'(k'>k)$满足上式）
- 问：*next*[*j*+1]=？

如何用递推方式来求出最大相同子串的长度?

case1：

- 若$p_K=p_j$,即 "$p_1p_2\cdots p_{k-1}p_k$"= "$p_{j-k+1}p_{j-k+2}\cdots p_{j-1}p_j$ " ②
- 则*next[j+1]=next[j]+1=k+1*；



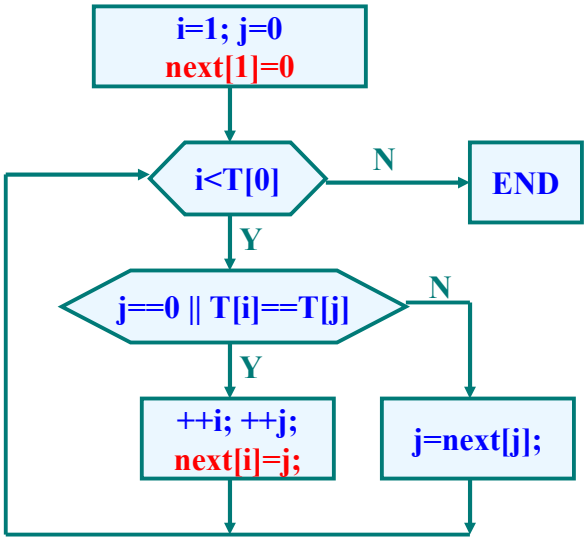
case2：

- 若$p_K\neq p_j$,即"$p_1p_2\cdots p_{k-1}p_k$" ≠ "$p_{j-k+1}p_{j-k+2}\cdots p_{j-1}p_j$",
- 但 "$p_1p_2\cdots p_{k-1}$ "= "$p_{j-k+1}p_{j-k+2}\cdots p_{j-1}$ ",
- 则应将模式向右滑动至模式中的*next[k]=k'*个字符比较，重复上述过程直至P_j和模式中的某个字符匹配成功或不存在任何*k'*(1<*k'*<*j*)满足 ②，则令*next[j+1]*=1。

算法4.7 (参见严教材P83或袁教材P87程序)

```
void get_next(SString T, int  &next[ ] ){
      //求模式串T的next函数值并存入数组next[ ]。
j=1;  next[1]=0; k=0;
while(j<T[0] ){
  while(k>= 0&&T[j] !=T[k]) k=next[k];    //case2
  ++j; ++k; next[j]=k;                    //case1
 }
}// get_next
```

求解next[j]流程图（递推）



next函数的改进算法见严教材P84算法4.8，nextval [j]

KMP算法（ 见教材P82算法4.6）

Step1: 先把模式T所有可能的失配点j 所对应的next[j]计算出来；

Step2: 执行定位函数Index_kmp （与BF算法模块很相似）

```
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;}     //不失配则继续比较后续字符
      else  j=next[ j ];                   //S的i指针不回溯，且从T的k位置开始匹配
    }
   if (j>T[0]) return i -T[0];             //子串结束，说明匹配成功
   else return 0;
}// Index_KMP
```

KMP算法的时间复杂度

BF的最坏情况：S与T之间存在大量的部分匹配，比较总次数为：

$$(n-m+1)*m = O(n*m)$$

KMP的情况是：由于指针i无须回溯，主串只扫描一趟，字符比较的复杂度为O(n),即使加上计算next[j]时所用的比较次数O(m)，比较总次数也仅为 O(n+m)，优于BF算法。

注意：

由于**BF**算法在一般平均情况下的时间复杂度也近似于**O(n+m)**，所以至今仍被广泛采用。

4.4 串的应用举例

- 文本编辑
  - 文本编辑程序是一个面向用户的系统服务程序，广泛应用于文件的起草、录入、修改、存储、打印。
  - 在文本编辑器中，处理的对象主要是针对字符串的，所以文本编辑器的基本操作，一般包括串的插入、删除、查找、替换、存储及打印等功能。
  - 利用块链结构时整个文本编辑区可看成是一个串，每一行是一个子串，构成一个结点。即：同一行的串用定长结构(80个字符)，行和行之间用指针相连接。
- 建立词索引表（信息检索）

小　结

- 串
  - 逻辑结构　$s =$“$a_1a_2 \ldots\ldots a_n$”
  - 存储结构
    - 定长顺序存储结构
    - 堆存储结构
    - 块链存储结构
  - 操 作
    - 若干函数的实现
    - 模式匹配算法
      - *BF*算法——古典
      - *KMP*算法— 快速

课外思考

严：4.6　4.20