CHAPTER 5

第5章 串

视频讲解

基本知识点 串的概念、串的存储结构和串的基本运算。

重点 在顺序串和链串上实现串的基本运算，模式匹配过程。

难点 串的模式匹配算法的设计和应用。

5.1 知识点1：串的基本概念

5.1.1 要点归纳

1. 串的定义

串是字符串的简称。**串**是由 $n(n\geqslant 0)$ 个字符组成的有限序列，$n=0$ 时为**空串**，用 Φ 或()表示。其他串称为非空串。任何串中所含字符的个数称为该**串的长度**(或串长)。空串的长度为0。

通常将一个串表示成"$a_1a_2\cdots a_n$"的形式。其中，最外边的双引号本身不是串的内容，它们是串的标志，以便将串与标识符(如变量名等)加以区别。每个 $a_i(1\leqslant i\leqslant n)$ 代表一个字符，在一般情况下，合法字符有英文字母、数字(0～9)和空格符等。

两个串相等：当且仅当它们的长度相等并且各个对应位置上的字符都相同。一个串中任意 $m(0\leqslant m\leqslant n)$ 个连续字符组成的子序列称为该串的**子串**，该串称为它的所有子串的**主串**，除主串自身之外的所有子串称为**真子串**。空串是任何串的子串。

说明：串中元素之间的逻辑关系为线性关系，串可以看成是一个特殊的线性表，其特殊性体现在串中每个元素均为单个字符。

2. 串的基本运算

串的基本运算如下所述。

(1) StrAssign(&s,chars)：将一个字符串常量赋给串 s，即生成一个值等于 chars 的串 s。

(2) StrCopy(&s,t)：串复制。将串 t 赋给串 s。

(3) StrLength(s)：求串长。返回串 s 中的字符个数。

(4) 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。

(5) SubStr(s,i,j)：求子串。返回串 s 中从第 $i(1\leqslant i\leqslant n)$ 个字符开始的、由连续 j 个字符组成的子串。

(6) InsStr($s1$,i,$s2$)：串插入。将串 $s2$ 插入串 $s1$ 的第 $i(1\leqslant i\leqslant n)$ 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。

(7) DelStr(s,i,j)：子串删除。从串 s 中删去从第 $i(1\leqslant i\leqslant n)$ 个字符开始的长度为 j 的子串，并返回产生的新串。

(8) DispStr(s)：串输出。输出串 s 的所有字符值。

3. 串的存储结构

串有两种主要的存储结构，即顺序串和链串。前者采用顺序存储结构实现串，后者采用链式存储结构实现串，它们之间的区别与本书第 2 章介绍的顺序表和链表类似。

5.1.2 例题解析

1. 单项选择题

【例 5-1-1】 串是任意有限个_____。

A. 符号构成的集合　　B. 符号构成的序列

C. 字符构成的集合　　D. 字符构成的序列

答：串是任意有限个字符构成的序列。本题答案为 D。

【例 5-1-2】 串是一种特殊的线性表，其特殊性体现在_____。

A. 可以顺序存储　　B. 数据元素是单个字符

C. 可以链接存储　　D. 数据元素可以是多个字符

答：串中每个元素都是单个字符。本题答案为 B。

【例 5-1-3】 以下_____是"abcd321ABCD"串的子串。

A. abcd　　B. 321AB　　C. "abcABC"　　D. "21AB"

答：子串是由主串中若干个连续的字符组成的。本题答案为 D。

【例 5-1-4】 两个串相等必有串长度相等且_____。

A. 串的各位置字符任意　　B. 串中各位置对应字符均相等

C. 两个串含有相同的字符　　D. 两个所含字符任意

答：B。

【例 5-1-5】 若串 s="software"，其子串的个数是_____。

A. 8　　B. 37　　C. 36　　D. 9

答：s 的长度为 8，长度为 8 的子串有 1 个，长度为 7 的子串有两个，长度为 6 的子串有 3 个，长度为 5 的子串有 4 个，……，长度为 1 的子串有 8 个，另有一个空串，所以 s 的子串共有 1＋2＋…＋8＋1＝37 个。本题答案为 B。

说明：s 的真子串的个数为 36 个，不含自身。

2. 填空题

【例 5-1-6】 空串是__(1)__，其长度等于__(2)__。

答：(1)0 个字符的串；(2)0。

【例 5-1-7】 一个串中_____称为该串的子串。

答：任意连续字符组成的子序列。

【例 5-1-8】 两个串相等的充分必要条件是_____。

答：两个串的长度相等且对应位置的字符相同。

3. 判断题

【例 5-1-9】 判断以下叙述的正确性。

(1) 串是由有限个字符构成的序列,子串是主串中任意字符构成的有限序列。

(2) 串长度为串中不同字符的个数。

(3) 串通常有顺序存储和链式存储两种存储结构。

(4) 空串就是由空格构成的串。

答：(1) 错误。子串是主串中任意个连续字符构成的有限序列。

(2) 错误。串长度为串中字符的个数。

(3) 正确。

(4) 错误。空串中不含有任何字符(包括空格字符)。

4. 简答题

【例 5-1-10】 简述一个字符串中子串的构成。

答：一个字符串中任意个连续字符组成的子序列称为该字符串的子串。

【例 5-1-11】 简述空串和空格串的区别。

答：空串是指长度为 0 的串,其中不包含任何字符,它是任何串的子串。空格串是指仅由空格字符构成的串,其长度大于或等于 1。

【例 5-1-12】 设 s 为一个长度为 n 的串,其中的字符各不相同,则 s 中的互异非平凡子串(非空且不同于 s 本身)的个数是多少?

答：由串 s 的特性可知,1 个字符的子串有 n 个,两个字符的子串有 $n-1$ 个,3 个字符的子串有 $n-2$ 个,……,$n-2$ 个字符的子串有 3 个,$n-1$ 个字符的子串有两个。所以,非平凡子串的个数$=n+(n-1)+(n-2)+\cdots+2=\dfrac{n(n+1)}{2}-1$。

【例 5-1-13】 若 $s1$ 和 $s2$ 为串,给出使 $s1//s2=s2//s1$ 成立的所有可能的条件(其中,// 为两个串连接运算符)。

答：所有可能的条件如下。

(1) $s1$ 和 $s2$ 为空串。

(2) $s1$ 或 $s2$ 其中之一为空串。

(3) $s1$ 和 $s2$ 为相同的串。

(4) 若两串长度不等,则长串由整数个短串组成。

5.2 知识点 2：顺序串的算法

5.2.1 要点归纳

1. 顺序串的定义

采用顺序存储结构存储的串称为**顺序串**,顺序串中的字符被依次存放在一组连续的存储单元里,并假设每个存储单元只存放一个字符。顺序串的类型定义如下。

```
typedef struct
{   char data[MaxSize];                    //存放串字符
```

```
    int length;                          //存放串长
} SqString;                              //声明顺序串类型
```

2. 顺序串的基本运算算法

(1) 创建串

将一个字符串常量赋给串 str,即生成一个其值等于 cstr 的串 s,对应算法如下。

```
void StrAssign(SqString &str,char cstr[])
{   int i;
    for (i=0;cstr[i]!='\0';i++)
        str.data[i]=cstr[i];
    str.length=i;
}
```

本算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$,其中 n 为 cstr 串中的字符个数。

(2) 复制串

将串 t 复制给串 s(不改变 t),对应算法如下。

```
void StrCopy(SqString &s,SqString t)
{   int i;
    for (i=0;i<t.length;i++)
        s.data[i]=t.data[i];             //逐个字符复制
    s.length=t.length;
}
```

本算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$,其中 n 为 t 串中的字符个数。

(3) 求串长

返回串 s 中的字符个数,对应算法如下。

```
int StrLength(SqString s)
{
    return s.length;
}
```

(4) 连接串

返回由两个串 s 和 t 连接在一起形成的新串(不改变 s 和 t),对应算法如下。

```
SqString Concat(SqString s,SqString t)
{   SqString str;
    int i;
    str.length=s.length+t.length;
    for (i=0;i<s.length;i++)             //将 s.data[0..s.length-1]复制到 str
        str.data[i]=s.data[i];
    for (i=0;i<t.length;i++)             //将 t.data[0..t.length-1]复制到 str
        str.data[s.length+i]=t.data[i];
    return str;
}
```

本算法的时间复杂度为 $O(m+n)$,空间复杂度为 $O(m+n)$,其中 m、n 为 s 和 t 串中的字符个数。

(5) 求子串

返回串 s 中从第 $i(1\leqslant i\leqslant n)$ 个字符开始的由连续 j 个字符组成的子串(不改变 s),对应算法如下。

```
SqString SubStr(SqString s,int i,int j)
{   SqString str;
    int k;
    str.length=0;
    if (i<=0 || i>s.length || j<0 || i+j-1>s.length)
        return str;                          //参数不正确时返回空串
    for (k=i-1;k<i+j-1;k++)                  //将 s.data[i..i+j]复制到 str
        str.data[k-i+1]=s.data[k];
    str.length=j;
    return str;
}
```

本算法的时间复杂度为 $O(n)$,空间复杂度为 $O(n)$,其中 n 为 s 串中的字符个数。

(6) 插入串

将串 $s2$ 插入串 $s1$ 的第 $i(1\leqslant i\leqslant n)$ 个字符前,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(不修改 $s1$ 和 $s2$),对应算法如下。

```
SqString InsStr(SqString s1,int i,SqString s2)
{   int j;
    SqString str;
    str.length=0;
    if (i<=0 || i>s1.length+1)               //参数不正确时返回空串
        return s1;
    for (j=0;j<i-1;j++)                      //将 s1.data[0..i-2]复制到 str
        str.data[j]=s1.data[j];
    for (j=0;j<s2.length;j++)                //将 s2.data[0..s2.length-1]复制到 str
        str.data[i+j-1]=s2.data[j];
    for (j=i-1;j<s1.length;j++)              //将 s1.data[i-1..s1.length-1]复制到 str
        str.data[s2.length+j]=s1.data[j];
    str.length=s1.length+s2.length;
    return str;
}
```

本算法的时间复杂度为 $O(m+n)$,空间复杂度为 $O(m+n)$,其中 m、n 分别为 $s1$、$s2$ 串中的字符个数。

(7) 删除子串

从串 s 中删去第 $i(1\leqslant i\leqslant n)$ 个字符开始的长度为 j 的子串,并返回产生的新串(不改变 s),对应算法如下。

```
SqString DelStr(SqString s,int i,int j)
{   int k;
    SqString str;
    str.length=0;
    if (i<=0 || i>s.length || i+j>s.length+1) //参数不正确时返回空串
        return str;
    for (k=0;k<i-1;k++)                      //将 s.data[0..i-2]复制到 str
```

```
    str.data[k]=s.data[k];
  for (k=i+j-1;k<s.length;k++)          //将 s.data[i+j-1..s.length-1]复制到 str
    str.data[k-j]=s.data[k];
  str.length=s.length-j;
  return str;
}
```

本算法的时间复杂度为 $O(n)$,空间复杂度为 $O(n)$,其中 n 为 s 串中的元素个数。

(8) 输出串

输出串 s 的所有字符值,对应算法如下。

```
void DispStr(SqString s)
{  int i;
  if (s.length>0)
  {  for (i=0;i<s.length;i++)
       printf("%c",s.data[i]);
    printf("\n");
  }
}
```

5.2.2 例题解析

1. 单项选择题

【例 5-2-1】 设 s="abcd",$s1$="123",则执行语句 s2=InsStr(s,2,s1)后,s2=_____。

A. "123abcd"　　B. "a123bcd"　　C. "ab123cd"　　D. "abc123d"

答:B。

【例 5-2-2】 设 s="abcd",则执行语句 s2=DelStr(s,2,2)后,s2=_____。

A. "abcd"　　B. "abc"　　C. "ad"　　D. "ab"

答:C。

2. 填空题

【例 5-2-3】 对于顺序串 s,将其初始化为空串的操作是_____。

答:s.length=0。

【例 5-2-4】 设 s="abcd",则执行语句 s2=SubStr(s,4,2)后,s2=_____。

答:空串或""。

3. 算法设计题

【例 5-2-5】③ 有一个仅含小写英文字母的顺序串 s,设计一个算法判断其中所有字母是否都是唯一的。

解:小写英文字母为 a～z,共 26 个,设置一个辅助数组 flag[26],其中 flag[i]表示它的 ASCII 码减'a'ASCII 码为 i 的字符出现的次数,初始化所有元素为 0。用 i 遍历串 s,循环将数组元素 flag[s[i]-'a']增 1,若有元素大于或等于 2,返回 0,表示不是所有字母都唯一;循环结束后返回 1,表示所有字母都唯一。对应的算法如下。

```
int Unique(SqString s)
{  int i,j;
  int flag[26];
```

```
    for (i=0;i<26;i++)                    //初始化 flag 所有元素为 0
        flag[i]=0;
    for (i=0;i<s.length;i++)              //遍历 s 的所有字符
    {   j=s.data[i]-'a';                  //求出当前字符对应的数组下标
        flag[j]++;
        if (flag[j]>=2)                   //不是唯一的返回 0
            return 0;
    }
    return 1;
}
```

【例 5-2-6】③ 设计一个算法,计算一个顺序串 s 中每个字符出现的次数。

解:设计一个结构体类型用于存放求解结果,即存放字符及其出现次数的数组元素类型,如下。

```
typedef struct
{    char ch;                         //存放字符
     int count;                       //存放字符出现的次数
} Cch;
```

用 i 扫描串 s,判断 s.data[i]是否在 C[]数组(其元素类型为 Cch)中,若在其中,只需将相应的 count 域增 1 即可;若不在其中,则将该字符添加到 C[]中,并将相应的 count 域置 1。对应算法如下。

```
int Count(SqString s,Cch C[])             //返回不同字符个数
{   int i,j,n=0;
    for (i=0;i<s.length;i++)
    {   for (j=0;j<=n;j++)                //在 C[]中查找等于 s.data[i]的字符
            if (C[j].ch==s.data[i])       //若找到,累计值增 1
            {   C[j].count++;
                break;
            }
        if (j>n)                          //若未找到,在 C 中增加一个元素
        {   C[n].ch=s.data[i];
            C[n].count=1;
            n++;
        }
    }
    return n;
}
```

【例 5-2-7】② 设计一个算法,判断顺序串 s 是否为回文(所谓回文,是一个字符串从前向后读和从后向前读的结果相同)。

解:用 i 从前向后扫描顺序串 s,用 j 从后向前扫描顺序串 s。当 $i<j$ 时,若 i、j 所指字符不相同,返回 0,否则循环扫描过程;循环结束返回 1。算法如下。

```
int Palindrome(SqString s)
{   int i=0,j=s.length-1;
    while (i<j)
    {   if (s.data[i]!=s.data[j])
            return 0;
```

```
        i++; j--;
    }
    return 1;
}
```

【例 5-2-8】③ 设计一个算法 RepStr(s,i,j,t),在串 s 中将第 i($1\leqslant i\leqslant n$)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(不修改 s 和 t)。

解:先判断参数 i、j 是否正确,若不正确,则返回一个空串;否则,新建一个空串 str,将 s.data[0..i−2]元素复制到 str 中,再将 t.data[0..t.length−1]元素复制到 str 中,再将 s.data[i+j−1..s.length−1]元素复制到 str 中,最后返回这个新串 str。对应算法如下。

```
SqString RepStr(SqString s,int i,int j,SqString t)
{   int k;
    SqString str;
    str.length=0;                                   //置 str 为空串
    if (i<=0 || i>s.length || i+j-1>s.length)       //参数不正确时返回空串
        return str;
    for (k=0;k<i-1;k++)                             //将 s.data[0..i-2]复制到 str
        str.data[k]=s.data[k];
    for (k=0;k<t.length;k++)                        //将 t.data[0..t.length-1]复制到 str
        str.data[i+k-1]=t.data[k];
    for (k=i+j-1;k<s.length;k++)                    //将 s.data[i+j-1..s.length-1]复制到 str
        str.data[t.length+k-j]=s.data[k];
    str.length=s.length-j+t.length;                 //设置 str 的长度
    return str;
}
```

【例 5-2-9】② 设计一个算法 RepChar(s,x,y),将串 s 中的所有字符 x 替换成字符 y。要求空间复杂度为 $O(1)$。

解:因要求算法空间复杂度为 $O(1)$,所以只能对串 s 直接替换,从头开始遍历串 s,一旦找到字符 x 便将其替换成 y,对应算法如下。

```
void RepChar(SqString &s,char x,char y)
{   int i;
    for (i=0;i<s.length;i++)
        if (s.data[i]==x)
            s.data[i]=y;
}
```

【例 5-2-10】③ 设计一个 Strcmp(s,t)算法,按照词典顺序比较两个顺序串 s 和 t 的大小。

解:本算法思路如下。

(1) 比较 s 和 t 两个串共同长度范围内的对应字符,若 s 的字符>t 的字符,返回 1;若 s 的字符<t 的字符,返回−1;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,两者相等时,返回 0;s 的长度>t 的长度时,返回 1;s 的长度<t 的长度时,返回−1。

以上思路对应的算法如下。

```
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 return -1;                           //s<t
}
```

【例 5-2-11】④ 设计一个递归算法,利用串的基本运算 SubStr()判断字符 x 是否在串 s 中。

解:设串 s="$a_1a_2\cdots a_n$",设 Find(s,x)的值表示 x 是否为串 s 的元素,若是,返回 1,否则返回 0。本题的递归模型如下。

Find(s,x)=0　　若 s 为空串

Find(s,x)=1　　若 $a_1=x$

Find(s,x)=Find("$a_2\cdots a_n$",x)　　其他情况

对应的递归算法如下。

```
int Find(SqString s,char x)
{   SqString s1;
    if (s.length==0)
        return 0;
    else if (s.data[0]==x)                    //a1=x
        return 1;
    else
    {   s1=SubStr(s,2,s.length-1);            //s1="a2…an"
        return(Find(s1,x));
    }
}
```

【例 5-2-12】③ 假设顺序串 s 中包含数字和字母字符,设计一个算法,将其中所有数字字符存放到顺序串 $s1$ 中,将其中所有字母字符存放到顺序串 $s2$ 中。要求不破坏顺序串 s,并且 $s1$、$s2$ 中的字符保持原来的相对次序不变。

解:用 i 扫描串 s,用 j、k 分别表示串 $s1$、$s2$ 的字符个数,初值均为 0。当 $i<s$.length 时,若 s.data[i]为数字,将其复制到 $s1$ 中,置 j++;若 s.data[i]为字母,将其复制到 $s2$ 中,置 k++,如此循环;循环结束,置 $s1$、$s2$ 的长度分别为 j、k。算法如下。

```
void Split(SqString s,SqString &s1, SqString &s2)
{   int i=0,j=0,k=0;
    while (i<s.length)
    {   if (s.data[i]>='0' && s.data[i]<='9')        //数字字符
        {   s1.data[j]=s.data[i];
            j++;
        }
        else if ((s.data[i]>='a' && s.data[i]<='z') ||
            (s.data[i]>='A' && s.data[i]<='Z'))     //字母字符
        {   s2.data[k]=s.data[i];
            k++;
```

```
        }
        i++;
    }
    s1.length=j; s2.length=k;
}
```

【例 5-2-13】④ 设计一个递归算法，利用串的基本运算算法 StrLength()、SubStr()和Concat()将非空串 s 的所有字符逆置。

解： 设串 s="$a_1a_2\cdots a_n$"，设 reverse(s)函数用于将串 s 直接逆置(逆置后 s="$a_na_{n-1}\cdots a_1$")，其递归模型如下。

reverse(s) ≡ 不做任何事件　　如果 s 只含一个字符

reverse(s) ≡ reverse("$a_2\cdots a_n$")与"a_1"连接　　其他情况

对应的递归算法如下。

```
void reverse(SqString &s)
{   SqString s1,s2;
    if (StrLength(s)>1)
    {   s1=SubStr(s,1,1);                    //s1="a1"
        s2=SubStr(s,2,s.length-1);           //s2="a2…an"
        reverse(s2);                         //s2=reverse("a2…an")
        s=Concat(s2,s1);                     //连接
    }
}
```

【例 5-2-14】④ 设顺序串 s 存放的是一个采用科学计数法正确表示的数值字符串，设计一个算法将其转换成对应的实数。例如将“1.345e-2”转换成 0.01345。

解： 先跳过顺序串 s 前面的空格，考虑其符号，正数用 sign=1 表示，负数用 sign=0 表示；考虑整数和小数部分，产生一个实数 val，跳过 e 或 E，对于正指数，val 乘以相应个数的 10，对于负指数，val 除以相应个数的 10；最后返回 val * sign。算法如下。

```
double Atoe(SqString s)
{   double val,power,e;
    int sign,i=0,j;
    char c;
    for (;s.data[i]==' '||s.data[i]=='\n'
        ||s.data[i]=='\t';i++);                  //跳过空格
    sign=1;
    if (s.data[i]=='+'||s.data[i]=='-')          //数的符号处理
        sign=(s.data[i++]=='+')?1:-1;
    for (val=0;s.data[i]>='0'&&s.data[i]<='9';i++)  //数字字符转换
         val=val*10+(s.data[i]-'0');
    if (s.data[i]=='.')                          //考虑小数部分
    {   i++;
        for (power=1;s.data[i]>='0'&&s.data[i]<='9';i++)
        {    val=val*10+s.data[i]-'0';
             power*=10;
        }
        val=val/power;
    }
    if (s.data[i]=='e'||s.data[i]=='E')          //考虑指数部分
```

```
    {  i++;
       if (s.data[i]=='+'|| s.data[i]=='-')              //取指数符号
          c=s.data[i++];
       else
          c='+';
       for (e=0;s.data[i]>='0'&& s.data[i]<='9';i++)
            e=e*10+(s.data[i]-'0');                      //取指数值
       if (c=='+')
          for (j=e;j>0;j--)
               val*=10;
       else
          for (j=e;j>0;j--)
               val/=10;
    }
    return (val*sign);
}
```

5.3 知识点3：链串的算法

5.3.1 要点归纳

1. 链串的定义

链串采用链表作为串的存储结构，通常用单链表表示链串，链串中的结点存储的只是字符值。将链串中每个结点所存储的字符个数称为**结点大小**。显然，结点越大，存储密度越大。但如果存储密度太大，一些操作(如插入、删除、替换等)将有所不便，且可能引起大量字符移动，因此它适合于在串基本保持静态使用方式时采用。反之，结点越小(如结点大小为1，即每个结点存放一个字符时)，操作处理越方便，但存储密度下降。为简便起见，这里规定链串结点大小均为1。

链串的结点类型定义如下。

```
typedef struct snode
{  char data;                                            //存放单个字符
   struct snode *next;
} LiString;                                              //声明链串结点类型
```

2. 链串的基本运算算法

(1) 创建链串

将一个字符串常量 t 赋给串 s，即生成一个其值等于 t 的串 s，采用尾插法对应的算法如下。

```
void StrAssign(LiString *&s,char t[])
{  int i;
   LiString *r,*p;                                       //r始终指向尾结点
   s=(LiString *)malloc(sizeof(LiString));
   r=s;
   for (i=0;t[i]!='\0';i++)
    {  p=(LiString *)malloc(sizeof(LiString));
       p->data=t[i];                                     //复制结点
       r->next=p;r=p;
```

```
    }
    r->next=NULL;                                    //尾结点的 next 域置为 NULL
}
```

本算法的时间复杂度为 $O(n)$,空间复杂度为 $O(n)$,其中 n 为 t 串中的字符个数。

(2) 复制串

将串 t 复制给串 s(不改变 t),采用尾插法建立链串 s,对应算法如下。

```
void StrCopy(LiString *&s,LiString *t)
{   LiString *p=t->next, *q, *r;
    s=(LiString *)malloc(sizeof(LiString));          //创建头结点
    r=s;                                             //r 始终指向尾结点
    while (p!=NULL)                                  //将 t 的所有结点复制到 s
     {  q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;                             //复制结点
        r->next=q;r=q;
        p=p->next;
    }
    r->next=NULL;                                    //尾结点的 next 域置为 NULL
}
```

本算法的时间复杂度为 $O(n)$,空间复杂度为 $O(n)$,其中 n 为 t 串中的字符个数。

(3) 求串长

返回串 s 中的字符个数,对应算法如下。

```
int StrLength(LiString *s)
{   int i=0;
    LiString *p=s->next;
    while (p!=NULL)
     {  i++;
        p=p->next;
    }
    return i;
}
```

(4) 连接串

返回由两个串 s 和 t 连接在一起形成的新串(不改变 s 和 t),采用尾插法建立结果链串,对应算法如下。

```
LiString *Concat(LiString *s,LiString *t)
{   LiString *str, *p=s->next, *q, *r;
    str=(LiString *)malloc(sizeof(LiString));
    r=str;                                           //r 始终指向尾结点
    while (p!=NULL)                                  //将 s 的所有结点复制到 str
     {  q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;q->next=NULL;
        r->next=q;r=q;
        p=p->next;
    }
    p=t->next;
    while (p!=NULL)                                  //将 t 的所有结点复制到 str
    {   q=(LiString *)malloc(sizeof(LiString));
```

```
        q->data=p->data;q->next=NULL;
        r->next=q;r=q;
        p=p->next;
    }
    r->next=NULL;                                    //尾结点的 next 域置为 NULL
    return str;
}
```

本算法的时间复杂度为 $O(m+n)$,空间复杂度为 $O(m+n)$,其中 m、n 分别为 s、t 串中的字符个数。

(5) 求子串

返回串 s 中从第 $i(1\leqslant i\leqslant n)$个字符开始的由连续 j 个字符组成的子串(不改变 s),采用尾插法建立结果子串,对应算法如下:

```
LiString *SubStr(LiString *s,int i,int j)
{   int k;
    LiString *str,*p=s->next,*q,*r;
    str=(LiString *)malloc(sizeof(LiString));
    str->next=NULL;                                  //建立空串
    r=str;                                           //r 始终指向尾结点
    if (i<=0 || i>StrLength(s) || j<0 || i+j-1>StrLength(s))
        return str;                                  //参数不正确时返回空串
    for (k=0;k<i-1;k++)                              //移动到第 i 个结点
        p=p->next;
    for (k=1;k<=j;k++)                               //将 s 的第 i 个结点开始的 j 个结点复制到 str
    {   q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;q->next=NULL;
        r->next=q;r=q;
        p=p->next;
    }
    r->next=NULL;                                    //尾结点的 next 域置为 NULL
    return str;
}
```

本算法的时间复杂度为 $O(n)$,空间复杂度为 $O(n)$,其中 n 为 s 串中的字符个数。

(6) 插入串

将串 t 插入到串 s 的第 $i(1\leqslant i\leqslant n)$个字符前,即将 t 的第一个字符作为 s 的第 i 个字符,并返回产生的新串(不改变 s 和 t),采用尾插法建立结果链串,对应算法如下。

```
LiString *InsStr(LiString *s,int i,LiString *t)
{   int k;
    LiString *str,*p=s->next,*p1=t->next,*q,*r;
    str=(LiString *)malloc(sizeof(LiString));
    str->next=NULL;                                  //建立空串
    r=str;                                           //r 始终指向尾结点
    if (i<=0 || i>StrLength(s)+1)                    //参数不正确时返回空串
        return str;
    for (k=1;k<i;k++)                                //将 s 的前 i-1 个结点复制到 str
    {   q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        r->next=q;r=q;
```

```
    p=p->next;
  }
  while (p1!=NULL)                          //将 t 的所有结点复制到 str
  {  q=(LiString *)malloc(sizeof(LiString));
     q->data=p1->data;
     r->next=q;r=q;
     p1=p1->next;
  }
  while (p!=NULL)                           //将结点 p 及其后的结点复制到 str
  {  q=(LiString *)malloc(sizeof(LiString));
     q->data=p->data;
     r->next=q;r=q;
     p=p->next;
  }
  r->next=NULL;                             //尾结点的 next 域置为 NULL
  return str;
}
```

本算法的时间复杂度为 $O(m+n)$，空间复杂度为 $O(m+n)$，其中 m、n 分别为 s、t 串中的字符个数。

(7) 删除子串

从串 s 中删去从第 $i(1\leqslant i\leqslant n)$ 个字符开始的长度为 j 的子串，并返回产生的新串(不改变 s 串)，采用尾插法建立新链串，对应算法如下。

```
LiString *DelStr(LiString *s,int i,int j)
{  int k;
   LiString *str,*p=s->next,*q,*r;
   str=(LiString *)malloc(sizeof(LiString));
   str->next=NULL;                          //建立空串
   r=str;                                   //r 始终指向尾结点
   if (i<=0 || i>StrLength(s) || j<0 || i+j-1>StrLength(s))
      return str;                           //参数不正确时返回空串
   for (k=0;k<i-1;k++)                      //将 s 的前 i-1 个结点复制到 str
   {  q=(LiString *)malloc(sizeof(LiString));
      q->data=p->data;
      r->next=q;r=q;
      p=p->next;
   }
   for (k=0;k<j;k++)                        //让 p 沿 next 跳 j 个结点
      p=p->next;
   while (p!=NULL)                          //将结点 p 及其后的结点复制到 str
   {  q=(LiString *)malloc(sizeof(LiString));
      q->data=p->data;
      r->next=q;r=q;
      p=p->next;
   }
   r->next=NULL;                            //尾结点的 next 域置为 NULL
   return str;
}
```

本算法的时间复杂度为 $O(n)$，空间复杂度为 $O(n)$，其中 n 为 s 串中的字符个数。

(8) 输出串

输出串 s 的所有字符值,对应算法如下。

```
void DispStr(LiString *s)
{   LiString *p=s->next;
    while (p!=NULL)
    {   printf("%c ",p->data);
        p=p->next;
    }
    printf("\n");
}
```

5.3.2 例题解析

1. 单项选择题

【例 5-3-1】 对于一个链串 s,查找第一个元素值为 x 的算法的时间复杂度为_____。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. 以上都不对

答:B。

【例 5-3-2】 对于一个链串 s,查找第 i 个元素的算法的时间复杂度为_____。

A. $O(1)$　　B. $O(n)$　　C. $O(n^2)$　　D. 以上都不对

答:B。

【例 5-3-3】 串采用结点大小为 1 的链表作为其存储结构,是指_____。

A. 链表的长度为 1

B. 链表中只存放一个字符

C. 链表中每个结点的数据域中只存放一个字符

D. 以上都不对

答:C。

2. 填空题

【例 5-3-4】 对于带头结点的链串 s,串为空的条件是_____。

答:s->next==NULL。

【例 5-3-5】 若要对串进行高效的模式匹配,在顺序串和链串中应选择_____存储结构。

答:顺序串具有随机存取特性,更适合于进行高效的模式匹配。本题答案为:顺序串。

3. 算法设计题

【例 5-3-6】④ 设计一个算法,在链串 s 中,将第 $i(1\leqslant i\leqslant n)$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,当参数不正确时返回空串。

解:采用尾插法建立结果链串 str。先创建一个空链串 str,再将 s 的前 $i-1$ 个结点复制到 str,再将 t 的所有结点复制到 str,然后在 s 中跳过 j 个结点至 p 结点,将结点 p 及其后的结点复制到 str,最后返回 str。对应算法如下:

```
LiString *RepStr(LiString *s,int i,int j,LiString *t)
{   int k;
    LiString *str,*p=s->next,*p1=t->next,*q,*r;
```

```
    str=(LiString *)malloc(sizeof(LiString));          //创建头结点
    str->next=NULL;
    r=str;                                             //r 指向 str 串的尾结点
    if (i<=0 || i>StrLength(s) || j<0 || i+j-1>StrLength(s))
        return str;                                    //参数不正确时返回空串
    for (k=0;k<i-1;k++)                                //将 s 的前 i-1 个结点复制到 str
    {   q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        r->next=q;r=q;
        p=p->next;
    }
    for (k=0;k<j;k++)                                  //让 p 沿 next 跳 j 个结点
        p=p->next;
    while (p1!=NULL)                                   //将 t 的所有结点复制到 str
    {   q=(LiString *)malloc(sizeof(LiString));
        q->data=p1->data;
        r->next=q;r=q;
        p1=p1->next;
    }
    while (p!=NULL)                                    //将结点 p 及其后的结点复制到 str
    {   q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        r->next=q;r=q;
        p=p->next;
    }
    r->next=NULL;                                      //尾结点的 next 域置为 NULL
    return str;
}
```

【例 5-3-7】④ 设计一个算法将一个链串 s 中的所有子串"abc"删除。

解：用 p、q、r、t 指针分别指向链串 s 的连续 4 个结点，p 首先指向头结点。当 q、r、t 所指结点的值分别为'a'、'b'和'c'时，删除这 3 个结点并释放其空间，再后移 q、r 和 t 指针；否则 p、q、r 和 t 指针同步后移一个结点。对应算法如下。

```
int Delsub(LiString *&s)
{   int n=0;
    LiString *p=s->next, *q, *r, *t;
    while (p!=NULL)                                    //求串 s 的长度 n
    {    p=p->next;
         n++;
    }
    if (n<3) return 0;                                 //少于 3 个字符时返回 0
    p=s;q=p->next;
    r=q->next;t=r->next;
    while (q!=NULL && r!=NULL && t!=NULL)
    {   if (q->data=='a' && r->data=='b' && t->data=='c')
        {   p->next=t->next;                           //找到"abc",删除它
            free(q);free(r);free(t);
            q=p->next;                                 //q、r、t 分别指向后面的 3 个结点
            if (q!=NULL)
                {   r=q->next;
                    if (r!=NULL) t=r->next;
```

```
            }
        }
        else                                          //不为"abc"时 p、q、r、t 同步后移一个结点
        {   p=p->next; q=p->next;
            r=q->next; t=r->next;
        }
    }
    return 1;
}
```

【例 5-3-8】③ 设计一个算法判断链串 s 中是否所有字符递增排列。

解：用 p 和 q 指向链串 s 的两个相邻结点，p 先指向首结点，当 q->data≥p->data 时，p 和 q 同步后移一个结点，否则返回 0；当所有元素是递增排列时返回 1，对应算法如下。

```
int increase(LiString *s)
{   LiString *p=s->next, *q;
    if (p!=NULL)
     {  while (p->next!=NULL)
        {   q=p->next;                                //q 指向 p 结点的后继结点
            if (q->data>=p->data)
                p=q;
            else                                      //逆序时返回 0
              return 0;
        }
    }
    return 1;
}
```

【例 5-3-9】④ 设计一个算法，计算一个链串 s 中每个字符出现的次数。

解：本题与例 5-2-6 的功能相同，只是这里改为链串，设计思路也相似，这里也将统计结果存放在一个带头结点的单链表 t 中，其结点类型定义如下。

```
typedef struct node
{   char ch;                                          //存放字符
    int count;                                        //存放字符出现的次数
    struct node *next;
} Cch;
```

本题对应算法如下。

```
void Count1(LiString *s,Cch *&t)
{   LiString *p=s->next;
    Cch *q, *r, *q1;
    t=(Cch *)malloc(sizeof(Cch));                     //建立头结点
    t->next=NULL;                                     //置为一个空链表
    r=t;                                              //r 始终指向尾结点
    while (p!=NULL)                                   //遍历 s 的所有结点
    {   q1=t->next;
        while (q1!=NULL && q1->ch!=p->data)
            q1=q1->next;                              //在 t 中查找等于 p->data 的字符
        if (q1!=NULL)                                 //在 t 中找到等于 p->data 的字符
            q1->count++;                              //累计值增 1
```

```
        else                                          //在 t 中没有找到等于 p->data 的字符
        {   q=(Cch *)malloc(sizeof(Cch));
            q->next=NULL;
            q->ch=p->data;
            q->count=1;
            r->next=q; r=q;
        }
        p=p->next;
    }
}
```

【例 5-3-10】③ 假设串采用链串存储,设计一个算法判断链串 s 是否具有对称性,并要求算法的时间复杂度为 $O(n)$,其中 n 为串 s 的长度。

解法 1:利用一个链栈实现算法,不改变链串 s。采用不带头结点的单链表表示链栈。先将 s 的所有结点复制后进栈,然后依次比较字符是否相等,若都相等,则 s 具有对称性,否则不具有对称性,对应的算法如下。

```
int Match1(LiString *s)                           //判断串 s 是否对称
{   LiString *st=NULL;                            //初始化链栈 st
    LiString *p=s->next, *q;
    while (p!=NULL)                               //将 s 的所有结点复制进栈
    {   q=(LiString *)malloc(sizeof(LiString));
        q->data=p->data;
        q->next=NULL;
        if (st==NULL)                             //原来为空栈
            st=q;
        else                                      //原来不为空栈
        {   q->next=st;                           //q 所指结点作为开始结点
            st=q;
        }
        p=p->next;
    }
    p=s->next;                                    //p 指向第一个字符结点
    while (p!=NULL && p->data==st->data)          //依次比较字符是否相等
    {   p=p->next;
        q=st;                                     //退栈操作
        st=st->next;
        free(q);
    }
    if (p!=NULL) return 0;
    else return 1;
}
```

上述算法的时间复杂度为 $O(n)$,空间复杂度为 $O(n)$,调用算法后链串 s 没有发生改变。

解法 2:不用栈实现算法,改变链串 s。先将链串 s 以中间位置断开,前半部分为带头结点的链串 s,后半部分是不带头结点的链串 $s1$。将 $s1$ 逆置,再判断两串对应位置的字符是否相等,若都相等,则原链串 s 具有对称性,否则不具有对称性,对应的算法如下。

```
LiString *Middle(LiString *s)                     //求中间位置结点
{   LiString *p=s, *q=s;
    while (q!=NULL && q->next!=NULL)
```

```
    {    p=p->next;
         q=q->next->next;
    }
    //当结点个数 n 为偶数时,p 指向前一个中间位置结点,n 为奇数时 p 指向中间位置结点
    return p;
}
void Reverse(LiString *&s)                          //逆置不带头结点的单链表 s
{   LiString *p=s, *q;
    s=NULL;
    while (p!=NULL)
    {   q=p->next;
        if (s==NULL)
        {   s=p;
            s->next=NULL;
        }
        else
        {   p->next=s;
            s=p;
        }
        p=q;
    }
}
int Match2(LiString *s)                             //判断串 s 是否对称
{   LiString *p, *s1, *q;
    if (s==NULL) return 1;                          //空表返回 1
    p=Middle(s);
    s1=p->next;                                     //将单链表 s 断开为两个部分
    if (s1==NULL) return 1;                         //后半部分为空,返回 1
    p->next=NULL;
    Reverse(s1);                                    //逆置不带头结点的单链表 s1
    p=s->next;                                      //p 指向前半部分首结点
    q=s1;                                           //q 指向后半部分首结点
    while (p!=NULL && q!=NULL)
    {   if (p->data!=q->data)
            return 0;
        p=p->next;
        q=q->next;
    }
    return 1;
}
```

上述算法的时间复杂度为 $O(n)$,空间复杂度为 $O(1)$,但调用算法后链串 s 发生了改变。

【例 5-3-11】③ 假设串采用链串存储,设计一个将串 t 插入串 s 中某个字符 c(第一次出现)之前的算法(若串 s 中不存在此字符,则将串 t 连接在串 s 的末尾)。要求算法的空间复杂度为 $O(1)$。

解:先找到串 t 的尾结点 q,再在串 s 中查找字符 c,用 p 指向该结点,pre 指向其前驱结点。若 p 结点存在,将 t 插入 pre 和 p 结点之间,否则将串 t 插入 pre 结点之后,对应的算法如下。

```
void Strinsert(LiString * &s,LiString * t,char c)
{   LiString * p=s->next, * pre=s, * q=t->next;
    while (q->next!=NULL)                    //找到串 t 的尾结点 q
        q=q->next;
    while (p!=NULL && p->data!=c)            //在串 s 中查找字符 c
    {   pre=p;                               //pre 指向 p 的前一个结点
        p=p->next;
    }
    if (p!=NULL)                             //在 s 中找到 c 字符
    {   pre->next=t->next;                   //将串 t 插入 pre 和 p 结点之间
        q->next=p;
    }
    else if (p==NULL)                        //在 s 中没有找到 c 字符,将串 t 插入 pre 结点之后
        pre->next=t->next;
    free(t);                                 //释放 t 的头结点
}
```

【例 5-3-12】③ 假设串采用链串存储,设计一个算法求串 s 中最长平台的长度,所谓平台,是指连续相同字符构成的子串。

解:用 maxcount 存放最大平台长度(初始值为 0),扫描串 s,计算一个平台的长度 count,若 count 大于 maxcount,则置 maxcount 为 count,最后返回 maxcount。对应的算法如下。

```
int Maxlength(LiString * s)
{   int count,maxcount=0;
    LiString * p=s->next, * q;
    while (p!=NULL)
    {   count=1;
        q=p;p=p->next;
        while (p!=NULL && p->data==q->data)
        {   count++;                         //求以结点 q 开头的平台长度
            p=p->next;
        }
        if (count>maxcount)                  //比较求更长的平台长度
            maxcount=count;
    }
    return maxcount;
}
```

5.4 知识点 4:模式匹配的算法

5.4.1 要点归纳

1. 串模式匹配的定义

设有串 s 和串 t,在串 s 中找到一个与串 t 相等的子串称为串定位,通常把串 s 称为**目标串**,把串 t 称为**模式串**,因此定位也称作**模式匹配**。模式匹配成功,是指在目标串 s 中找到一个模式串 t;不成功则指目标串 s 中不存在模式串 t。

说明:为了算法设计方便,本节中子串在主串中的位置均指物理位置,例如,主串为"*aababcd*",子串为"*abc*",则子串在主串中的位置记为 3(其逻辑位置应为 4)。

2. Brute-Force 算法

Brute-Force 算法(简称 BF 算法)是一种简单的暴力匹配算法,其思路是从目标串 s="$s_0s_1\cdots s_{n-1}$"的第一个字符开始和模式串 t="$t_0t_1\cdots t_{m-1}$"中的第一个字符比较,若相等,则继续逐个比较后续字符,否则从目标串 s 的第二个字符开始重新与模式串 t 的第一个字符比较,以此类推,若存在模式串 t 中的每个字符依次和目标串中的一个连续字符序列相等,则匹配成功,函数返回模式串 t 中第一个字符在主串 s 中的位置;否则匹配失败,函数返回-1。BF 算法如下。

视频讲解

```
int index(SqString s,SqString t)
{   int i=0,j,k;
    while (i<s.length)
    {   for (j=i,k=0;j<s.length && k<t.length
             && s.data[j]==t.data[k];j++,k++);
        if (k==t.length)            //子串比较完毕
            return i;               //返回在主串中的位置
        i++;                        //继续匹配
    }
    return(-1);
}
```

上述算法中,用 i 扫描 s,用 j 从 i 开始扫描 s,k 从头开始扫描 t,当串 s 和串 t 的当前比较字符相等时,依次比较下去,当 t 的所有字符比较完毕,则说明 t 是 s 的子串。

现只用 i 扫描 s,j 扫描 t,当 s 和 t 的当前比较字符相等时,依次比较下去,即 $i++$,$j++$;当 s 和 t 的当前比较字符不相等时进行回溯,即 $i=i-j+1$,t 从头开始比较,即 $j=0$,修改后的 BF 算法如下。

```
int index1(SqString s,SqString t)
{   int i=0,j=0;
    while (i<s.length && j<t.length)
    {   if (s.data[i]==t.data[j])           //若当前比较的字符相等
        {   i++;                            //主串和子串依次匹配后续字符
            j++;
        }
        else                                //若当前比较的字符不相等
        {   i=i-j+1;                        //主串从下一个位置重新开始匹配
            j=0;                            //子串从头开始匹配
        }
    }
    if (j>=t.length)
        return i-t.length;                  //返回主串中相匹配的第一个字符的位置
    else
        return -1;                          //模式匹配不成功
}
```

上述两个算法是等价的,只不过修改后的算法减少了一个变量 k,所以需要 i 回溯。它们的时间复杂度均为 $O(n\times m)$,其中 n 和 m 分别为串 s 和 t 的长度。

说明:BF 算法采用穷举思路,依次从串 s 的每个字符开始与串 t 的字符进行匹配,每次匹配都是独立的。

例如,设主串 s="abababababca",模式串 t="ababc",采用上述 index1(s,t)函数求子串位置的过程如图 5.1 所示,最后匹配成功时 $i=9$、$j=5$(与 t.length 相等),返回 i-t.length=4,表示 t 是 s 的子串,其位置是 4。

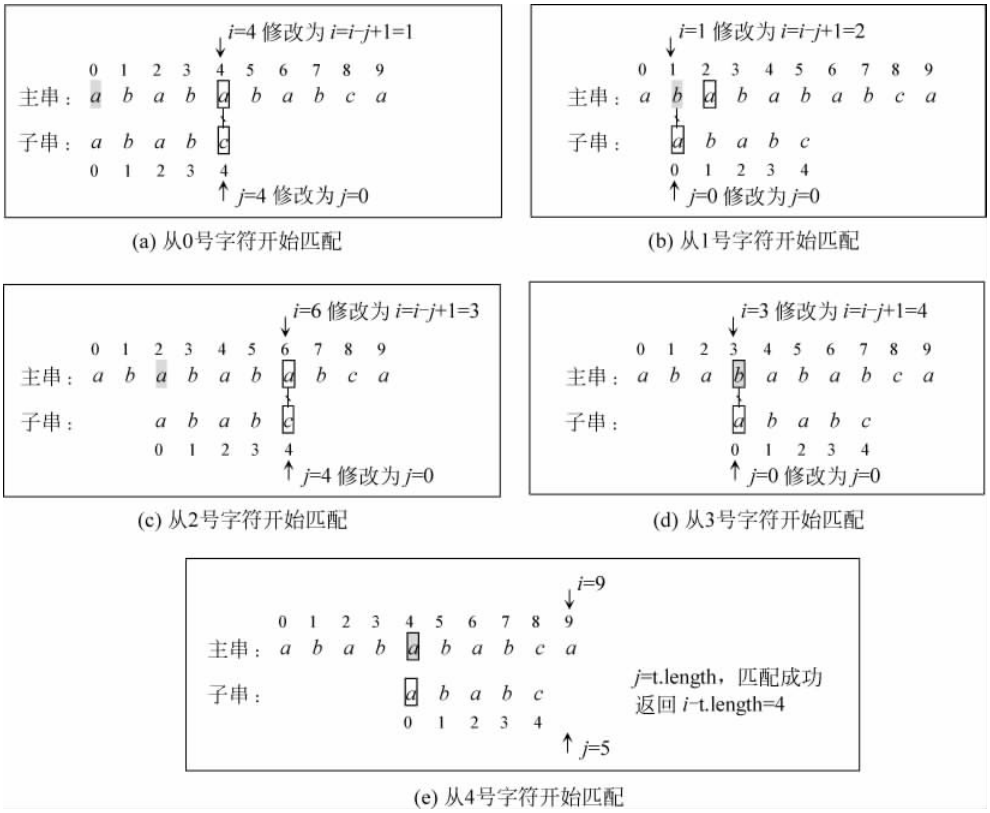


图 5.1 BF 算法的模式匹配过程示例

3. KMP 算法

视频讲解

KMP 算法是由 D. E. Knuth、J. H. Morris 和 V. R. Pratt 等人共同提出，该算法分析了模式串中隐含的用于模式匹配的信息，这种信息就是模式串中的“部分匹配”信息。

对于模式串 t 的每个字符 t_j，存在一个整数 $k(k<j)$，使得模式串 t 中的 k 个字符($t_0t_1\cdots t_{k-1}$)依次与 t_j 前面的 k 个字符($t_{j-k}t_{j-k+1}\cdots t_{j-1}$)相同，并与主串 s 中 i 所指字符之前的 k 个字符相等。那么，下次匹配仅需要从模式串 t 中的 k 所指字符与主串 s 的 i 所指字符开始继续比较。

说明：模式串中隐含什么信息？如图 5.2 所示，模式串为"$xaxb$"，假设 x、y 均表示一个字符序列(这里作为变量)，当匹配到字符 b 时，说明 x 和主串中字符序列 y 匹配成功，也说明字符 b 前有若干字符与主串中的 y 字符序列是相匹配的，下次不必进行重复比较，这种信息需要记录在字符 b 中，即 next[b 字符下标]=x 的字符个数。

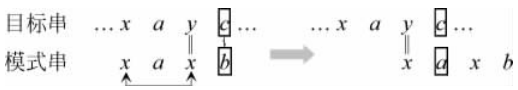


图 5.2 匹配过程

对于图 5.1 所示的例子，模式串 t="$ababc$"，用 next 数组存放这些“部分匹配”信息，对于 0 号字符 a，规定 next[0]=−1(固定)；对于 1 号字符 b，规定 next[1]=0(固定)；对于 2 号字

符 a,前一个字符 b 不等于模式 t 的开头字符,即 next[2]=0;对于 3 号字符 b,前面子串有"a"、"ba",其中"a"与模式 t 的开头字符匹配,它含一个字符,所以 next[3]=1;对于 4 号字符 c,前面的子串有"b"、"ab"和"bab",其中"ab"与模式串 t 的开头字符匹配,它含两个字符,则 next[4]=2。所以模式串 t 对应的 next 数组如表 5.1 所示。

表 5.1　模式串 t 的 next 值

j	0	1	2	3	4
$t[j]$	a	b	a	b	c
next[j]	−1	0	0	1	2

归纳起来,求模式串 t 的 next[j]数组(称为失效函数)的公式如下:

$$\text{next}[j]=\begin{cases}\text{MAX}\{k \mid 0<k<j \mid "t_0t_1\cdots t_{k-1}"="t_{j-k}t_{j-k+1}\cdots t_{j-1}"\} & \text{当此集合非空时}\\ -1 & \text{当 } j=0 \text{ 时}\\ 0 & \text{其他情况}\end{cases}$$

说明:其中,k 表示模式 t 中 j 号字符前有多少个字符与开头相应个字符相同,如果有多个 k,next[j]取值最大。

当有了 next 数组之后,图 5.1 所示的例子采用 KMP 算法的模式匹配过程如图 5.3 所示。可以看到,由于利用了 next 数组中的信息,只是从主串的 0、2 和 4 号字符开始比较,明显提高了匹配效率。

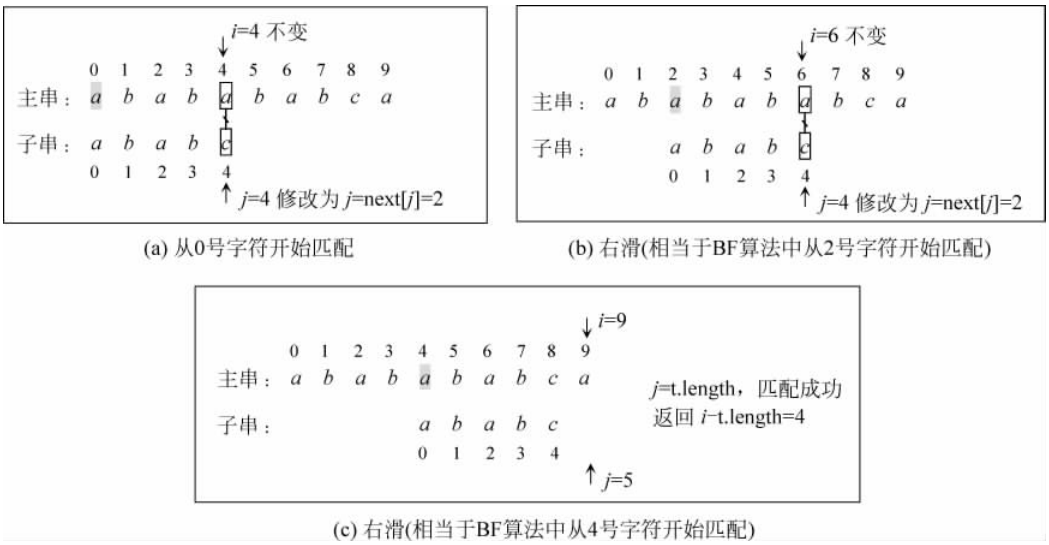


图 5.3　KMP 算法的模式匹配过程示例

下面讨论一般情形,设目标串 $s="s_0s_1\cdots s_{n-1}"$,模式串 $t="t_0t_1\cdots t_{m-1}"$,在进行第 i 趟匹配时,出现以下情况(从 s_{i-j} 开始比较到 $s_i\neq t_j$ 失败):

$$\begin{array}{lccccccccc} s: & s_0s_1 & \cdots & s_{i-j}s_{i-j+1} & \cdots & s_{i-1}s_is_{i+1} & \cdots & s_{n-1} \\ & & & \mid\ \mid & & \mid\ \backslash & & \\ t: & t_0 & t_1 & \cdots & t_{j-1}t_jt_{j+1} & \cdots & s_{m-1} & & \end{array}$$

这时,应有:

$$"t_0t_1\cdots t_{j-1}"="s_{i-j}s_{i-j+1}\cdots s_{i-1}" \qquad \text{(式 5.1)}$$

如果在模式 t 中有：

$$"t_0t_1\cdots t_{j-1}" \neq "t_1t_2\cdots t_j" \qquad (式\ 5.2)$$

则回溯到 s_{i-j+1} 开始与 t 匹配必然“失配”，由式 5.1 和式 5.2 综合可知：

$$"t_0t_1\cdots t_{j-1}" \neq "s_{i-j+1}s_{i-j+2}\cdots s_i"$$

既然如此，回溯到 s_{i-j+1} 开始与 t 匹配可以不做。那么，回溯到 s_{i-j+2} 开始与 t 匹配又怎么样？由以上推理可知，如果：

$$"t_0t_1\cdots t_{j-2}" \neq "t_2t_3\cdots t_j"$$

仍然有：

$$"t_0t_1\cdots t_{j-2}" \neq "s_{i-j+2}s_{i-j+3}\cdots s_i"$$

这样的比较仍然“失配”。以此类推，直到对于某一个值 k，使得：

$$"t_0t_1\cdots t_{k-2}" \neq "t_{j-k+1}t_{j-k+2}\cdots t_{j-1}" \quad 且 \quad "t_0t_1\cdots t_{k-1}" = "t_{j-k}t_{j-k+1}\cdots t_{j-1}"$$

则有：

$$"t_{j-k}t_{j-k+1}\cdots t_{j-1}" = "s_{i-k}s_{i-k+1}\cdots s_{i-1}" = "t_0t_1\cdots t_{k-1}"$$

上述推理过程说明下一次可以从 s_i 和 t_k 开始比较，这样就直接把从 s_{i-j} 开始比较“失配”时的模式串 t 从当前位置直接右滑 $j-k$ 位，这里的 k 即为 next[j]。也就是说，BF 算法中总是从目标串 s 的 s_0，s_1，s_2，…字符开始依次与模式串 t 匹配，匹配趟数是连续的，即 $i=0,1,2,\cdots$；而 KMP 算法会根据 next 数组值跳过一些不必要匹配，从而提高了匹配效率。

总结前面的讨论，可以得到求 next 数组的算法和 KMP 算法如下。

```
void GetNext(SqString t,int next[])                    //由模式串 t 求出 next 值
{   int j,k;
    j=0;k=-1;next[0]=-1;
    while (j<t.length-1)
    {   if (k==-1 || t.data[j]==t.data[k])              //如果 k 为-1 或比较的两字符相等
        {   j++;k++;
            next[j]=k;
        }
        else k=next[k];
    }
}
int KMPIndex(SqString s,SqString t)                    //KMP 算法
{   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
    }
    if (j>=t.length)                                    //返回匹配模式串的首字符下标
        return i-t.length;
    else                                                //返回匹配失败标志
        return -1;
}
```

设主串 s 的长度为 n，子串 t 的长度为 m，在 KMP 算法中求 t 的 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 i 不减，即不回溯，比较次数可记为 n，所以 KMP 算法总的时间复杂度为 $O(n+m)$。

上述定义的 next 数组在某些情况下尚有缺陷。例如，主串 $s=$"*aaabaaaab*"，模式串 $t=$

"*aaaab*",采用KMP算法的匹配过程如图5.4所示,在这个匹配过程中,当$i=3$、$j=3$时,$s_3 \neq t_3$,由next[j]的指示还需进行$i=3/j=2$、$i=3/j=1$、$i=3/j=0$三次比较,实际上,因为模式串中的t_0、t_1、t_2字符和t_3都相等,由于$s_3 \neq t_3$,所以一定有$s_3 \neq t_0$,$s_3 \neq t_1$,$s_3 \neq t_2$,因此,不需要再将t_0、t_1、t_2字符和主串中s_3字符比较,可以将模式一次向右滑动4个字符的位置直接进行$i=4/j=0$时的字符比较。

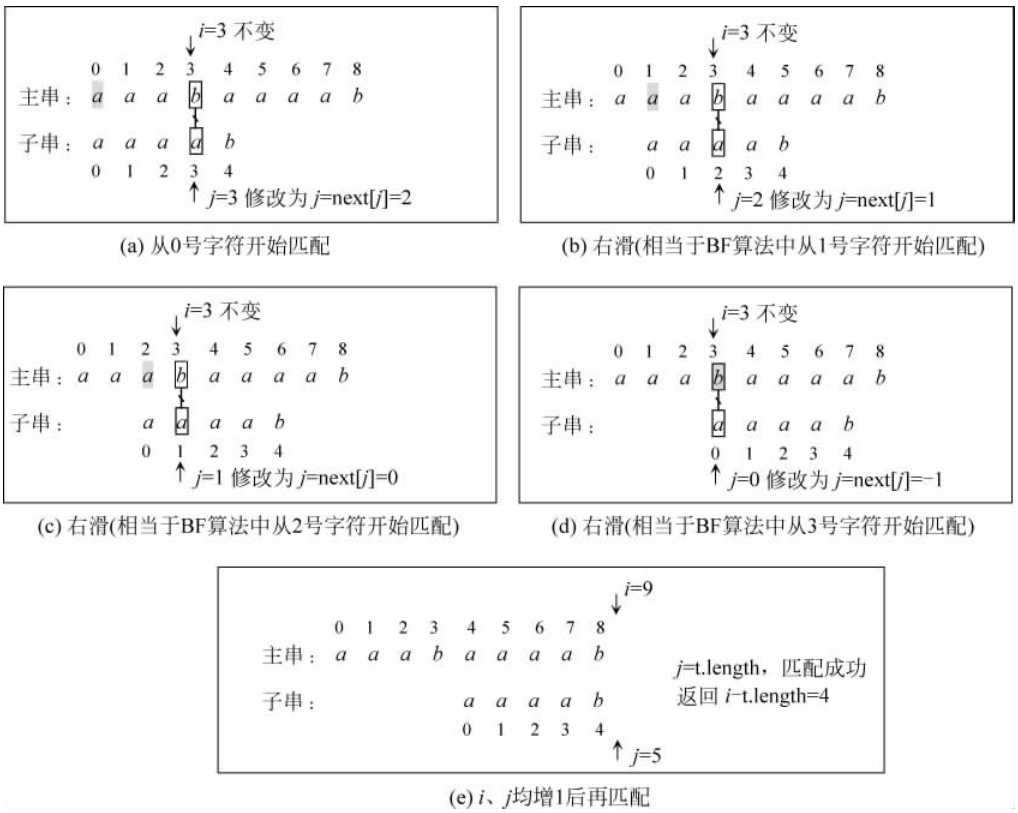


图5.4　KMP算法的模式匹配过程示例

也就是说,当s和t匹配时有$s_i \neq t_j$,若按next定义得到next[j]$=k$,KMP算法实际是进行s_i与t_k的比较;但如果模式串中有$t_j=t_k$,则不需要再将s_i与t_k进行比较。为此将next[j]修正为nextval[j],求nextval数组的方法如下。

(1) nextval[0]$=-1$。

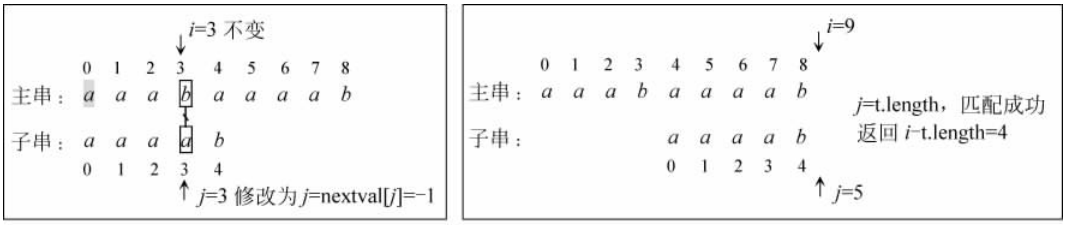
(2) 若$k=$next[j],并且$t_j=t_k$,则nextval[j]$=$nextval[k]。

(3) 若$k=$next[j],并且$t_j \neq t_k$,则nextval[j]$=$next[j]。

这样,模式串t的nextval数组值如表5.2所示,与主串$s=$"*aaabaaaab*"的匹配过程如图5.5所示,可以看到匹配效率也得到了提高。

表5.2　模式串t的nextval值

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



(a) 从0号字符开始匹配　　(b) i、j均增1后再匹配

图 5.5　修正 KMP 算法的模式匹配过程示例

修正后的 KMP 算法如下。

```
void GetNextval(SqString t,int nextval[])          //由模式串 t 求出 nextval 值
{   int j=0,k=-1;
    nextval[0]=-1;
    while (j<t.length)
    {   if (k==-1 || t.data[j]==t.data[k])
        {   j++;k++;
            if (t.data[j]!=t.data[k])
                nextval[j]=k;
            else
                nextval[j]=nextval[k];
        }
        else k=nextval[k];
    }
}
int KMPIndex1(SqString s,SqString t)               //修正的 KMP 算法
{   int nextval[MaxSize],i=0,j=0;
    GetNextval(t,nextval);
    while (i<s.length && j<t.length)
    {   if (j==-1 || s.data[i]==t.data[j])
        { i++;j++; }
        else j=nextval[j];
    }
    if (j>=t.length)
        return i-t.length;
    else
        return -1;
}
```

与改进前的 KMP 算法一样，本算法的时间复杂度也为 $O(n+m)$。

5.4.2　例题解析

1. 单项选择题

【例 5-4-1】 设有两个串 p 和 q，求 q 在 p 中首次出现的位置的运算称为_____。

A. 连接　　B. 模式匹配　　C. 求子串　　D. 求串长

答：B。

【例 5-4-2】 已知 t="*abcaabbcabcaabdab*"，该模式串的 next 数组值为_____。

A. −1,0,0,0,1,1,2,0,0,1,2,3,4,5,6,0,1

B. 0,1,0,0,1,1,2,0,0,1,2,3,4,5,6,0,1

C. −1,0,0,0,1,1,2,0,0,1,2,3,4,5,6,7,1

D. −1,0,0,0,1,1,2,3,0,1,2,3,4,5,6,0,1

答：next 数组的求解过程如表 5.3 所示。本题答案为 A。

表 5.3　计算 next 数组

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$t[j]$	a	b	c	a	a	b	b	c	a	b	c	a	a	b	d	a	b
next[j]	−1	0	0	0	1	1	2	0	0	1	2	3	4	5	6	0	1

【例 5-4-3】 在 BF 算法中，模式串位 j 与目标串位 i 比较，如果两字符不相等，则 j 的位移方式是_____。

A. j++　　B. $j=0$　　C. $j=i-j+1$　　D. $j=j-i+1$

答：B。

【例 5-4-4】 在 BF 算法中，模式串位 j 与目标串 i 比较，如果两字符相等，则 i 的位移方式是_____。

A. i++　　B. $i=j+1$　　C. $i=i-j+1$　　D. $i=j-i+1$

答：A。

【例 5-4-5】 在 BF 算法中，模式串位 j 与目标串 i 比较，如果两字符相等，则 j 的位移方式是_____。

A. j++　　B. $i=j+1$　　C. $i=i-j+1$　　D. $i=j-i+1$

答：A。

【例 5-4-6】 在 KMP 算法中，用 next 数组存放模式串的部分匹配信息，模式串位 j 与目标串位 i 比较，如果两字符不相等，则 i 的位移方式是_____。

A. $i=$next[j]　　B. i 不变　　C. j 不变　　D. $j=$next[j]

答：B。

【例 5-4-7】 在 KMP 算法中，用 next 数组存放模式串的部分匹配信息，模式串位 j 与目标串位 i 比较，如果两字符不相等，则 j 的位移方式是_____。

A. $i=$next[j]　　B. i 不变　　C. j 不变　　D. $j=$next[j]

答：D。

【例 5-4-8】 在 KMP 算法中，用 next 数组存放模式串的部分匹配信息，模式串位 j 与目标串 i 比较，如果两字符相等时，则 i 的位移方式是_____。

A. i++　　B. $i=j+1$　　C. $i=i-j+1$　　D. $i=$next[i]

答：A。

【例 5-4-9】 在 KMP 算法中，用 next 数组存放模式串的部分匹配信息，模式串位 j 与目标串 i 比较，如果两字符相等时，则 j 的位移方式是_____。

A. j++　　B. $i=j+1$　　C. $i=i-j+1$　　D. $i=$next[i]

答：A。

【例 5-4-10】 在 KMP 算法中，用 next 数组存放模式串的部分匹配信息，next[j]=−1 的含义是_____。

A. 表示 $j=-1$　　B. 表示下一趟从 $j=0$ 位置开始比较

C. 表示两字符比较相等　　D. 表示两串匹配成功

答：B。

【例 5-4-11】 设目标串为 s，模式串为 t，在 KMP 模式匹配中，next[4]＝2 的含义是_____。

A. 表示目标串匹配失败的位置是 i＝4

B. 表示模式串匹配失败的位置是 j＝2

C. 表示 t_4 字符前面最多有 2 个字符和 t 开头的 2 个字符相同

D. 表示 s_4 字符前面最多有 2 个字符和 s 开头的 2 个字符相同

答：C。

2. 填空题

【例 5-4-12】 模式串 t＝"*abaabcac*"的 next 数组值为__(1)__，nextval 数组值为__(2)__。

答：next 数组和 nextval 数组的求解过程如表 5.4 所示。本题答案为(1)－1,0,0,1,1,2,0,1；(2)－1,0,－1,1,0,2,－1,1。

表 5.4 计算 next 数组和 nextval 数组

j	0	1	2	3	4	5	6	7
$t[j]$	*a*	*b*	*a*	*a*	*b*	*c*	*a*	*c*
next[j]	－1	0	0	1	1	2	0	1
nextval[j]	－1	0	－1	1	0	2	－1	1

【例 5-4-13】 模式串 t＝"*abcaabbabcab*"，对应的 next 数组值为__(1)__，nextval 数组值为__(2)__。

答：next 数组和 nextval 数组的求解过程如表 5.5 所示。本题答案为(1)－1,0,0,0,1,1,2,0,1,2,3,4；(2)－1,0,0,－1,1,0,2,－1,0,0,－1,4。

表 5.5 计算 next 数组和 nextval 数组

j	0	1	2	3	4	5	6	7	8	9	10	11
$t[j]$	*a*	*b*	*c*	*a*	*a*	*b*	*b*	*a*	*b*	*c*	*a*	*b*
next[j]	－1	0	0	0	1	1	2	0	1	2	3	4
nextval[j]	－1	0	0	－1	1	0	2	－1	0	0	－1	4

3. 简答题

【例 5-4-14】 在 KMP 算法中，计算模式串的 next 数组时，当 j＝0 时，为什么要取 next[0]＝－1？


视频讲解

答：在 KMP 算法中，当目标串 s 与模式串 t 匹配时，若 $s_i=t_j$，执行 i＋＋，j＋＋(称为情况 1)；若 $s_i\neq t_j$(失配处)，i 位置不变，置 j＝next[j]（称为情况 2)。若失配处是 j＝0，即 $s_i\neq t_0$，那么从 s_i 开始的子串与 t 匹配一定不成功，下一趟匹配应该从 s_{i+1} 与 t_0 开始比较，即 i＋＋，j＝0，为了与情况 1 统一，置 next[0]＝－1 即 j＝next[0]＝－1，这样再执行 i＋＋，j＋＋→j＝0，从而保证下一趟从 s_{i+1} 开始与 t_0 进行匹配。

【例 5-4-15】 在串的模式匹配中，KMP 算法是很有用的算法，回答以下问题。

(1) KMP 算法的基本思想是什么？

(2) 对模式串 $t(t=t_0,t_1,\cdots,t_{m-1})$,求 next 数组时,next[$j$]为$-1$或满足什么条件的 k 的最大值。

答:(1) KMP 匹配算法的基本思想是每当一趟匹配过程中出现字符比较不等时,不需回溯目标串的 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。

(2) next[j]为-1或满足"$t_0\cdots t_{k-1}$"="$t_{j-k}\cdots t_{j-1}$"条件的 k 的最大值。

例如,模式串 t="$abcabcaaa$",求 t 的 next 数组如表 5.6 所示,计算过程说明如下。

当 $j=0$ 时,next[0]$=-1$(固定值)。

当 $j=1$ 时,next[1]$=0$(固定值)。

当 $j=2$ 时,$t_0\neq t_1$,next[2]$=0$。

当 $j=3$ 时,$t_0\neq t_2$,$t_0t_1\neq t_1t_2$,next[3]$=0$。

当 $j=4$ 时,$t_0=t_3=$"a",$k=j-3=1$,next[4]$=k=1$。

当 $j=5$ 时,$t_0t_1=t_3t_4=$"ab",$k=j-3=2$,next[5]$=k=2$。

当 $j=6$ 时,$t_0t_1t_2=t_3t_4t_5=$"abc",$k=j-3=3$,next[6]$=k=3$。

当 $j=7$ 时,$t_0t_1t_2t_3=t_3t_4t_5t_6=$"$abca$",$k=j-3=4$,next[7]$=k=4$。

当 $j=8$ 时,$t_0t_1\neq t_6t_7$,$t_0=t_7=$"a",$k=j-7=1$,next[8]$=k=1$。

表 5.6 模式 t 对应的 next 数组值

j	0	1	2	3	4	5	6	7	8
$t[j]$	a	b	c	a	b	c	a	a	a
next[j]	-1	0	0	0	1	2	3	4	1

【例 5-4-16】 设目标串为 s="$abcaabbabcabaacbacba$",模式串 t="$abcabaa$"。

(1) 计算模式串 t 的 nextval 数组值。

(2) 不写算法,只画出利用 KMP 算法进行模式匹配时每一趟的匹配过程。

答:(1) 先计算 next 数组,在此基础上求 nextval 数组,如表 5.7 所示。

表 5.7 计算 next 数组和 nextval 数组

j	0	1	2	3	4	5	6
$t[j]$	a	b	c	a	b	a	a
next[j]	-1	0	0	0	1	2	1
nextval[j]	-1	0	0	-1	0	2	1

(2) 采用 KMP 算法求子串位置的过程如下(开始时 $i=0$,$j=0$)。

第 1 趟匹配:s="$abca\ \boxed{a}\ bbabcabaacbacba$"

t="$abca\ \boxed{b}\ aa$"

此时 $i=4$,$j=4$,匹配失败,则 $i=4$,j 修改为 $j=$nextval[4]$=0$。

第 2 趟匹配:s="$abcaab\ \boxed{b}\ abcabaacbacba$"

t="$ab\ \boxed{c}\ abaa$"

此时 $i=6$,$j=2$,匹配失败,则 $i=6$,j 修改为 $j=$nextval[2]$=0$。

第 3 趟匹配：s="$abcaab\ \boxed{b}\ abcabaacbacba$"

t="$\boxed{a}\ bcabaa$"

此时 $i=6,j=0$，匹配失败，则 $i=6,j$ 修改为 $j=$nextval[0]$=-1$。因 $j=-1$，执行 $i=i+1=7,j=j+1=0$。

第 4 趟匹配：s="$abcaabb\underline{abcabaa}cbacba$"

t="$\underline{abcabaa}$"

此时 $i=14,j=7$，匹配成功，返回 $i-t$.length$=14-7=7$。

4. 算法设计题

【例 5-4-17】③ 用顺序结构存储串，设计一个算法在串 str 中查找子串 substr 最后一次出现的位置(不能使用任何字符串标准函数)。

解：在主串 str 中通过简单匹配一直查找子串 substr，返回最后找到的位置，对应算法如下。

视频讲解

```
int LastPos(SqString str,SqString substr)
{   int i,j,k,idx=-1;
    for (i=0;i<str.length;i++)
    {   for (j=i,k=0;j<str.length && k<substr.length && str.data[j]==substr.data[k];j++,k++);
        if (k==substr.length)
            idx=i;
    }
    return idx;
}
```

【例 5-4-18】③ 用顺序结构存储串，设计一个算法在串 s 中从后向前查找子串 t 最后一次出现的位置(不能使用任何字符串标准函数)。

解：采用简单模式匹配算法，如果串 s 的长度小于串 t 的长度，直接返回 -1；否则 i 从 s.length$-t$.length 到 0 循环，对于 i 的每次取值置 $j=i$、$k=0$，若 s.data[j]$==t$.data[k]，则 $j++,k++$，循环扫描，当 $k==t$.length，表示找到子串，返回最后一个子串的下标 i；所有循环结束后，表示串 t 不是串 s 的子串时，返回 -1。算法如下。

```
int LastPos(SqString s,SqString t)
{   int i,j,k;
    if (s.length-t.length<0)
        return -1;
    for (i=s.length-t.length;i>=0;i--)
    {   for (j=i,k=0;j<s.length && k<t.length && s.data[j]==t.data[k];j++,k++);
        if (k==t.length)
            return i;
    }
    return -1;
}
```

【例 5-4-19】③ 用顺序结构存储串，设计一个实现主串 s 和子串 t 进行通配符匹配的算法 Match(s,t)，其中的通配符只有?，它可以和任一字符匹配成功，例如，Match("there are","? re")返回的结果是 2。

解：采用简单匹配方法，只需增加通配符 '?'的处理功能，对应算法如下。

```
int Match(SqString s,SqString t)
{   int i,j,k;
    for (i=0;i<s.length;i++)
    {   for (j=i,k=0;j<s.length && k<t.length && s.data[j]==t.data[k]
            || t.data[k]=='?';j++,k++);
        if (k==t.length)
            return i;
    }
    return -1;
}
```

【例 5-4-20】④ 用顺序结构存储串,设计一个算法求串 t 在串 s 中不重叠出现的次数,如果不是子串,则返回 0。例如"aa"在"aaaab"中计为出现 2 次。

解法 1:用 count 累计 t 在串 s 中不重叠出现的次数(初始值为 0),采用 BF 方法,在找到子串后不是退出,而是 count 增加 1,i 增加 t 的长度并继续查找,直到整个字符串查找完毕。对应的算法如下。

视频讲解

```
int StrCount1(SqString s,SqString t)              //利用 BF 算法求 t 在 s 中出现的次数
{   int i=0,j,k,count=0;
    while (i<s.length-t.length)
    {   for (j=i,k=0; j<s.length && k<t.length &&
            s.data[j]==t.data[k]; j++,k++);
        if (k==t.length)                          //找到一个子串
        {   count++;                              //累加次数
            i+=t.length;                          //i 从 j 开始
        }
        else i++;                                 //i 增加 1
    }
    return(count);
}
```

解法 2:用 count 累计 t 在串 s 中不重叠出现的次数(初始值为 0),采用 KMP 方法,当匹配成功时,count 增加 1,并且置 j 为 0 重新开始比较。对应的算法如下。

```
void GetNext(SqString t,int next[])               //由模式串 t 求出 next 值
{   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++;                              //k 为-1 或比较的两字符相等的情况
            next[j]=k;
        }
        else k=next[k];
    }
}
int StrCount2(SqString s,SqString t)              //利用 KMP 算法求 t 在 s 中出现的次数
{   int i=0,j=0,count=0;
    int next[MaxSize];
    GetNext(t,next);
    while (i<s.length && j<t.length)
    {   if (j==-1 || s.data[i]==t.data[j])
```

```
        {   i++;
            j++;                                    //i 和 j 各增 1
        }
        else j=next[j];                             //i 不变,j 后退
        if (j>=t.length)                            //成功匹配 1 次
        {   count++;
            j=0;                                    //j 设置为 0,继续匹配
        }
    }
    return count;
}
```

【例 5-4-21】⑤ 采用顺序结构存储串，设计一个算法求串 s 中出现的第一个最长重复子串(不重叠)。

解：采用简单匹配算法的思路，先给最长重复子串的下标 maxi 和长度 maxlen 均赋值为 0。设 $s=$"$a_1\cdots a_n$"，扫描通过串 s，对于当前字符 a_i，判定其后是否有相同的字符，若有记为 a_j，再判定 a_{i+1}是否等于 a_{j+1}，a_{i+2}是否等于 a_{j+2}，……，直至找到一个不同的字符为止，即找到一个重复出现的子串，把其下标 i 与长度 len 记下来，将 len 与 maxlen 相比较，保留较长的子串 maxi 和 maxlen；再从 $a_{j+\text{len}}$之后查找重复子串，然后对于 a_{i+1}之后的字符采用上述函数，最后的 maxi 与 maxlen 即记录下最长重复子串的下标与长度，将其复制到串 t 中并返回。对应算法如下。

视频讲解

```
SqString Maxsubstr(SqString s)
{   int maxi=0,maxlen=0,len,i=0,j,k;
    SqString t;
    while (i<s.length)
    {   j=i+1;
        while (j<s.length)
        {   if (s.data[i]==s.data[j])               //找一个子串,其序号为 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];
    return t;
}
```

【例 5-4-22】⑤ 采用顺序结构存储串，两个串 s 和 t 的长度分别为 m 和 n，设计一个算法求这两个串的最大公共子串。给出其时间复杂度 $T(m,n)$，说明最好情况下的时间复杂度是

多少。

解：采用简单匹配算法，其思路与例5-4-21类似，返回值为 s、t 的最大公共子串，不妨假设 $m>n$，对应算法如下：

```
SqString maxcomstr(SqString s,SqString t)
{   SqString str;
    int midx=0,mlen=0,tlen,i=0,j,k;              //用(midx,mlen)保存最大公共子串
    while (i<s.length)                           //用i扫描串s
    {   j=0;                                     //用j扫描串t
        while (j<t.length)
        {    if (s.data[i]==t.data[j])           //找到一个子串,在s中下标为i,长度为tlen
             {     tlen=1;
                   for (k=1;i+k<s.length && j+k<t.length
                            && s.data[i+k]==t.data[j+k];k++)
                        tlen++;
                   if (tlen>mlen)                //将较大长度者赋给midx与mlen
                   {   midx=i;
                       mlen=tlen;
                   }
                   j+=tlen;                      //继续扫描t中第j+tlen字符之后的字符
              }
              else j++;
        }
        i++;                                     //继续扫描s中第i字符之后的字符
    }
    for (i=0;i<mlen;i++)                         //将最大公共子串复制到str中
         str.data[i]=s.data[midx+i];
    str.length=mlen;
    return str;                                  //返回最大公共子串
}
```

该算法的时间复杂度 $T(m,n)$ 为 $O(m\times n)$。最好情况是串 t 恰好是串 s 的首字符开头的子串，此时算法时间复杂度为 $O(n)$。