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

1. 基本概念

- 串(字符串) String：是零个或多个字符组成的有限序列。一般记为：S= '$a_1a_2...a_n$' (n≥0)

  栈、队列：操作受限的线性表。

  串：取值范围受限的线性表，数据对象约束为字符集。

  其中S是串的名字, 用单引号括起来的字符序列是串的值，a_i(1≤i≤n)可以是字母、数字或其它字符。

- 串的长度：串中字符的个数n。

- 空串(Null String)： n=0时的串称为空串，用符号 ∅ ”表示 。

- 子串：串中任意个连续的字符组成的子序列称为该串的子 串。“∅”为任意串的子串。
- 主串：包含子串的串相应地称为主串。

- 位置：字符在串中的序号称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。

  A=‘China Beijing’, B=‘Beijing’, C=‘China’, 则它们的长度分别为13、7和5。B和C是A的子串, B在A中的位置是7，C在A中的位置是1。

- 串相等：当且仅当两个串的值相等时,称这两个串是相等的，即只有当两个串的长度相等，并且每个对应位置的字符都相等时才相等。

- 空格串：由一个或多个空格组成的串。与空串不同。

串的逻辑结构和线性表相似，故看作一种线性表。

$$s = \text{‘}a_1 a_2 \cdots a_n\text{’} \qquad (n \geq 0)$$

串的基本操作和线性表区别很大。

线性表：大多以“单个元素”为操作对象

例，查找某个元素；插入某个元素；删除某个元素

串：通常以“串的整体”为操作对象

例，查找某个子串；截取某个子串；

在某个位置插入、删除某个子串

2. 串的基本运算:

- 串赋值StrAssign(&T, chars)
  初始条件: chars是字符串常量。
  操作结果: 生成一个值等于chars的串T。

- 串比较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;如S=T, 则返回值=0;
  若S<T, 则返回值<0。

- 求串长StrLength(S)
  初始条件: 串S存在。
  操作结果: 返回串S的长度, 即串S中的元素个数。

- 串联接Concat(&T, S1, S2)

  初始条件：串S1和S2存在。

  操作结果: 将T返回由S1和S2联接而成的新串。

- 求子串SubString(&Sub, S, pos, len)

  初始条件：串S存在，1≤pos≤StrLength(S)且

  1≤len≤StrLength(S)-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。

4.2 串的表示和实现

- 定长顺序存储表示——顺序存储
- 堆分配存储表示——顺序存储
- 块链存储表示——链式存储

1. 定长顺序存储表示（定长顺序串）

```
# define MAXSTRLEN 255

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

- 定长顺序串的存储分配是在编译时完成的。与前面所讲的线性表的顺序存储结构类似, 用一组地址连续的存储单元存储串的字符序列。
- 超出予定义长度的串值被舍去，称之为“截断”。

- **串长的表示：**
  - **①以下标为0的数组分量存放串的实际长度；**
  - **②串值后加入一个不计入串长的结束标记字符，C中“\0”表串值的终结，其ASCII码值为0。**

算法4.2 串联接 strcat(&T，S1，S2)

要求：顺序联接串 S1 和串 S2 得到新串 T 。

思想：

基于串S1和S2长度的不同情况，串T可能出现3种情况：

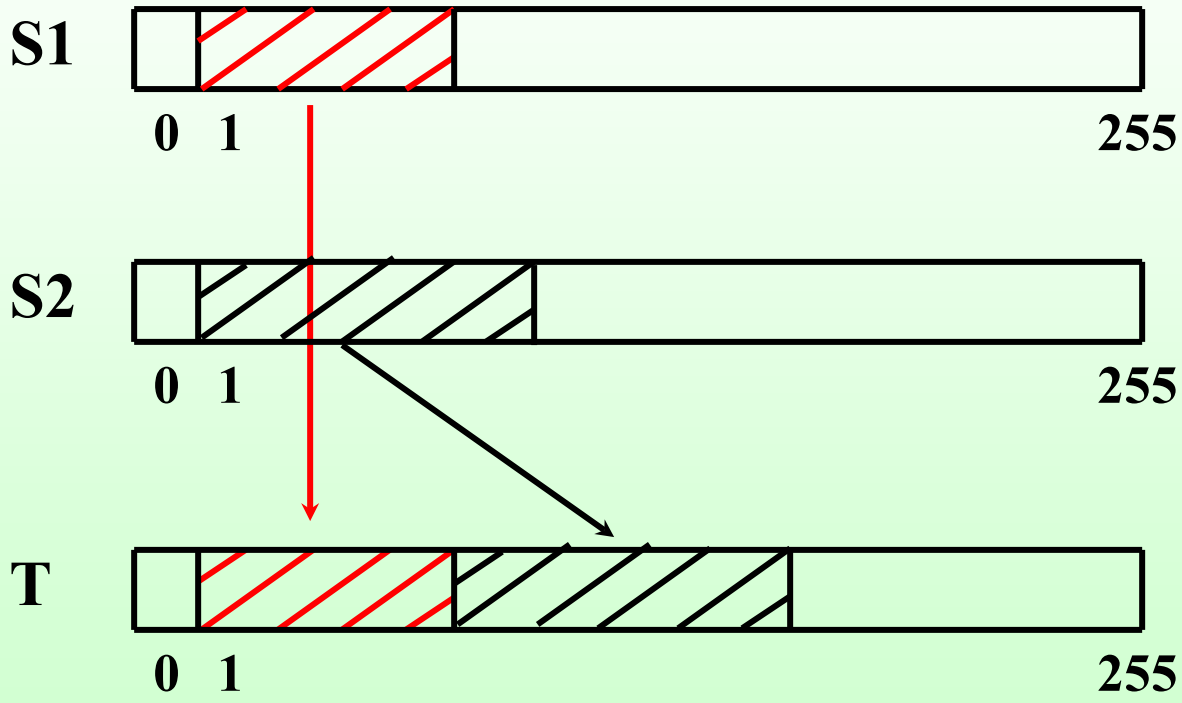
S1[0]+S2[0]≤MAXSTRLEN， 直接联接，T[0]≤MAXSTRLEN ；

S1[0]+S2[0]＞MAXSTRLEN

S1[0]＜MAXSTRLEN，截断S2，联接，T[0]＝MAXSTRLEN；

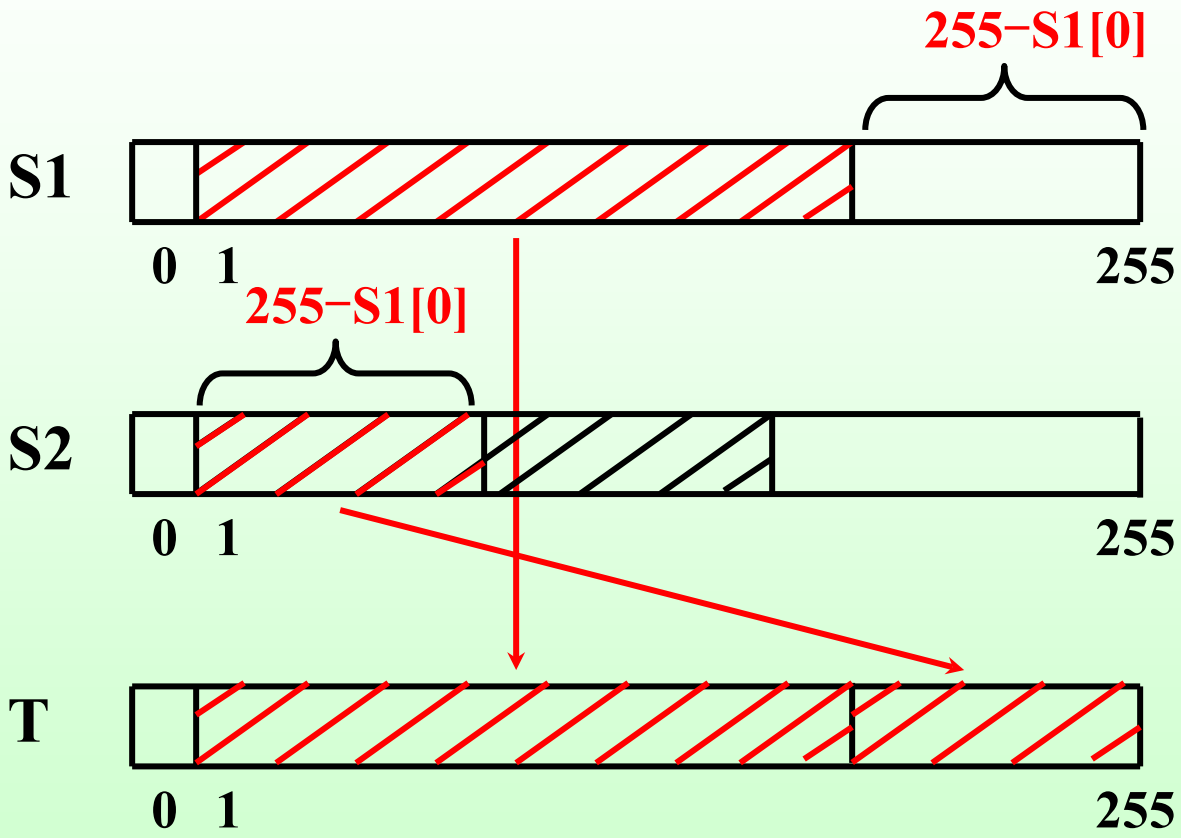
S1[0]＝MAXSTRLEN，T ＝ S1；

S1[0]+S2[0]≤MAXSTRLEN



T[0] = S1[0] + S2[0]

S1[0]+S2[0]＞MAXSTRLEN，S1[0]＜MAXSTRLEN



T[0]＝MAXSTRLEN

2. 堆分配存储表示（堆串）

在C语言中，已经有一个称为“堆”的自由存储空间，并可用malloc（）和free（）函数完成动态存储管理。

```
typedef struct{
    char  * ch;
    int   length;
} HString;
```

应用程序用到的内存分配：栈分配和堆分配。

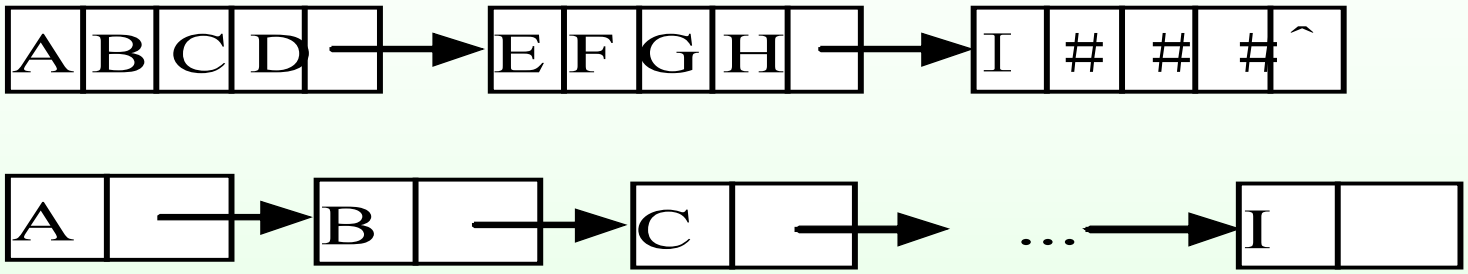
堆：用户程序动态申请的地址空间。

栈：保存函数参数和块内局部变量的内存区。

```
void Fun1() {
   int i;
   char p[10];….;
}
```

```
void Fun2(){
    int *p=new int[10];
    delete [] p;…..;
}
```

3. 串的块链存储表示（链串）




```
#define  CHUNKSIZE  <大小>
typedef struct Chunk{
      char    ch[CHUNKSIZE];
      struct  Chunk   *next;
}Chunk;
```

```
typedef struct {
      Chunk   *head;
      Chunk   *tail;
      int       curlen;
}LString;
```

$$存储密度=\frac{串值所占的存储位}{实际分配的存储位}$$

- 存储密度大，一些操作（如插入，删除）有所不便，引起大量字符移动，适合于在串基本保持静态使用方式时采用；
- 存储密度小，运算处理方便，但存储空间量大，若处理过程中，需大量内外存交换，会影响总效率；
- 串的字符集的大小，也会影响串值存储方式的选取。

- 作业：

  4.1

  4.18

4.3 串的模式匹配算法

1. 朴素模式匹配算法(Brute-Force算法)

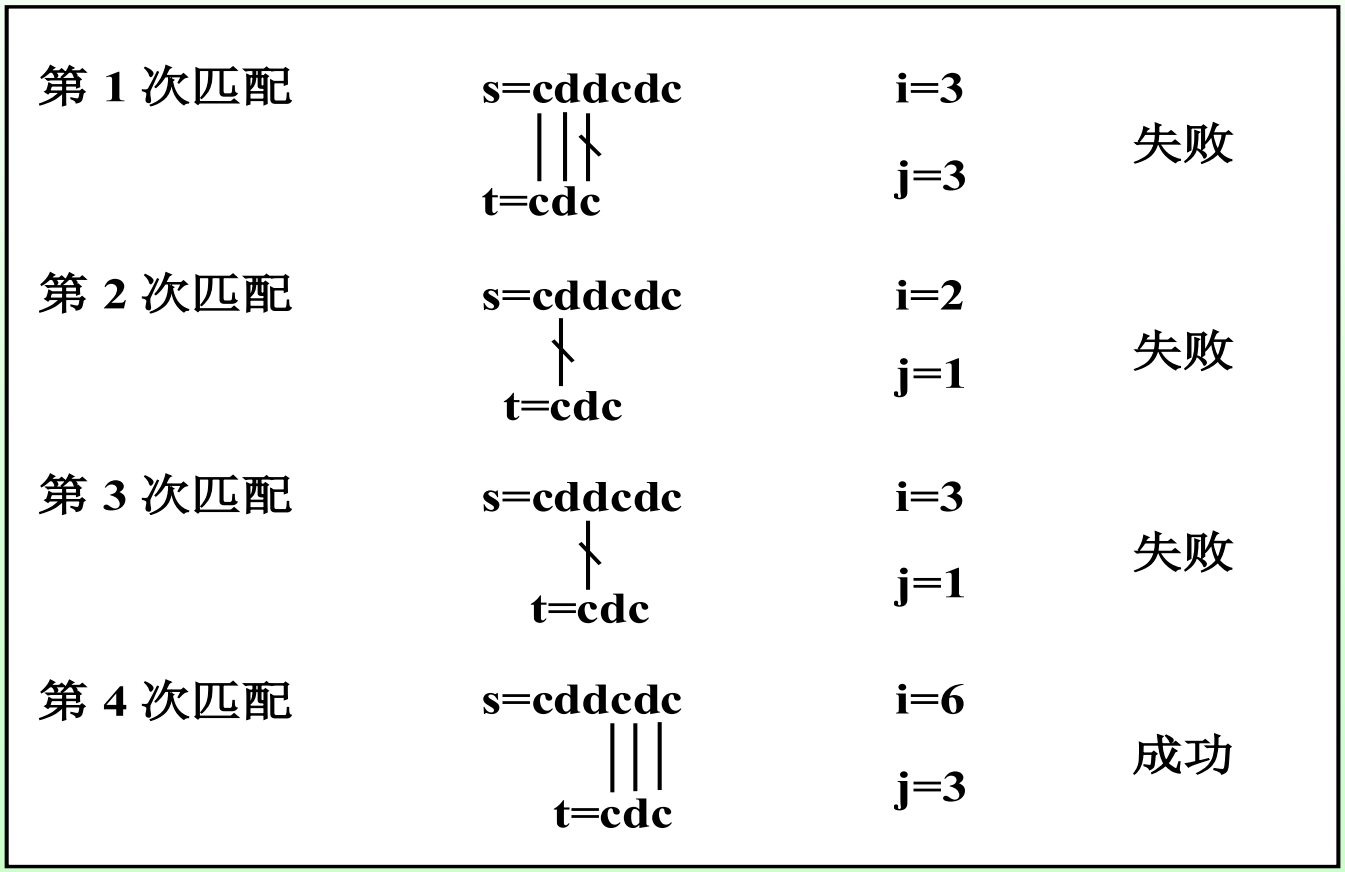
求子串位置的定位函数Index(S, T, pos).

- 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。

- **Brute-Force简称为BF算法,亦称简单匹配算法,其基本思路是:**

  从目标串s="$s_1s_2\ldots s_n$"的第一个字符开始和模式串t="$t_1t_2\ldots t_m$"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的第一个字符进行比较。依次类推,若从目标串s的第i个字符开始,每个字符依次和模式串t中的对应字符相等,则匹配成功,该算法返回i；否则,匹配失败,函数返回0。

例如,设目标串s="cddcdc",模式串t="cdc"。s的长度为n(n=6),t的长度为m(m=3)。用指针i指示目标串s的当前比较字符位置,用指针j指示模式串t的当前比较字符位置。BF模式匹配过程如下所示。



i = i –j +2;

j = 1;

- 子串定位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; }
         else{ i = i-j+2;   j =1; }
     }
     if(j>T[0]) return i-T[0];
     else return 0;
}
```

- 朴素模式匹配算法的时间复杂度

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)。

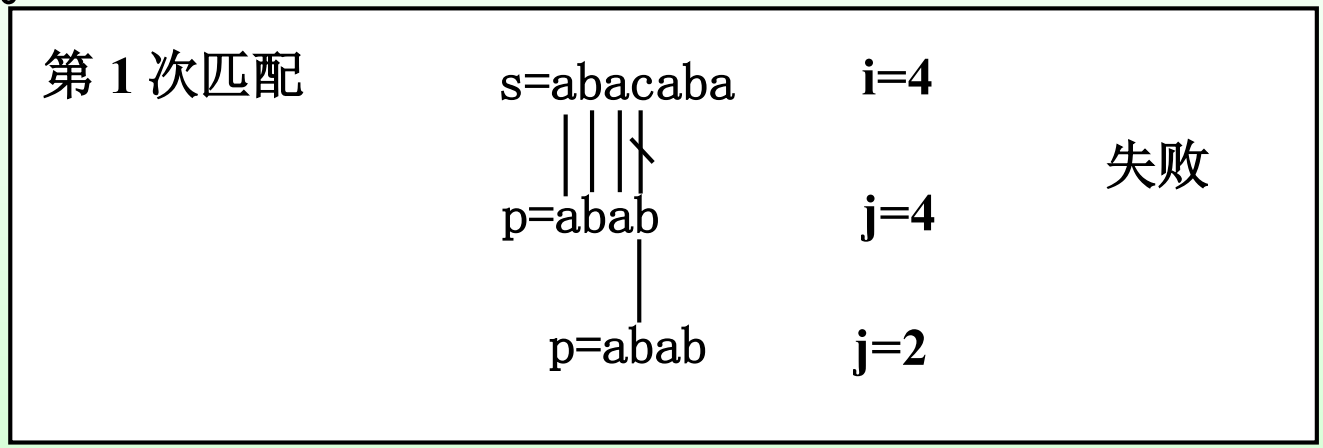
②最坏的情况下，

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)。

2. 模式匹配的改进算法-KMP算法

KMP算法是D.E.Knuth、J.H.Morris和V.R.Pratt共同提出的,简称KMP算法。该算法较BF算法有较大改进,主要是消除了主串指针的回溯,从而使算法效率有了某种程度的提高。



因$p_1 \neq p_2$, $s_2=p_2$, 必有$s_2 \neq p_1$, 又因$p_1=p_3$, $s_3=p_3$, 所以必有$s_3=p_1$。因此, 第二次匹配可直接从i=4, j=2开始。

改进：每趟匹配过程中出现字符比较不等时，不回溯主指针i，利用已得到的“部分匹配”结果将模式向右滑动尽可能近的一段距离，继续进行比较。

$$
\begin{array}{cccccccccc}
s_1 & s_2 & s_3 \cdots & s_{i-j+1} & s_{i-j+2} \cdots & s_{i-2} & s_{i-1} & s_i & s_{i+1} \\
 & & \| & & \| & \| & \| & \neq & \\
 & & p_1 & & p_2 \cdots & p_{j-2} & p_{j-1} & p_j & p_{j+1} \\
 & & & & \| & & \| & & \\
 & & & & p_1 \cdots & & p_{k-1} & p_k & p_{k+1}
\end{array}
$$

① “$p_1p_2\cdots p_{k-1}$” = “$s_{i-k+1}s_{i-k+2}\cdots s_{i-1}$”

②“$p_{j-k+1}p_{j-k+2}\cdots p_{j-1}$” = “$s_{i-k+1}s_{i-k+2}\cdots s_{i-1}$”（部分匹配）

③ “$p_1p_2\cdots p_{k-1}$” = “$p_{j-k+1}p_{j-k+2}\cdots p_{j-1}$”（真子串）

为此,定义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。

$$
\text{next}[j]=\begin{cases}\max\{\,k \mid 1<k<j,\text{且“}p_1\cdots p_{k-1}\text{”}=\text{“}p_{j-k+1}\cdots p_{j-1}\text{”}\,\} & \text{当此集合非空时}\\ 0 & \text{当 } j=1 \text{ 时}\\ 1 & \text{其他情况}\end{cases}
$$

```
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*/
}
```

- 如何求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$p_k=p_j$，则有“$p_1 \cdots p_{k-1}p_k$”=“$p_{j-k+1} \cdots p_{j-1}p_j$”，如果在

j+1发生不匹配，说明next[j+1] = k+1 = next[j]+1。

②若$p_k \neq p_j$，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。

$p_1\ p_2 \cdots p_{j-k+1} \cdots p_{j-1}\ p_j\ p_{j+1}$ next[j]=k

$\|\qquad\qquad\|\quad \neq$

$p_1 \cdots p_{k-1}\ p_k\ p_{k+1}$ next[k]=k'

$p_1 \cdots\cdots p_{k'}\ p_{k'+1}$ next[k']=k"

$p_1 \cdots p_{k"}\ p_{k"+1}$ next[k"]=k"'

- 若$p_{k'}=p_j$，则有“$p_1 \cdots p_{k'}$”=“$p_{j-k'+1} \cdots p_j$”，
  next[j+1]=k'+1=next[k]+1=next[next[j]]+1.
- 若$p_{k"}=p_j$，则有“$p_1 \cdots p_{k"}$”=“$p_{j-k"+1} \cdots p_j$”，
  next[j+1]=k"+1=next[k']+1=next[next[k]]+1.
- next[j+1]=1.

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
模式串	a	b	c	a	a	b	b	c	a	b	c	a	a	b	d	a	b
next[j]	0	1	1	1	2	2	3	1	1	2	3	4	5	6	7	1	2

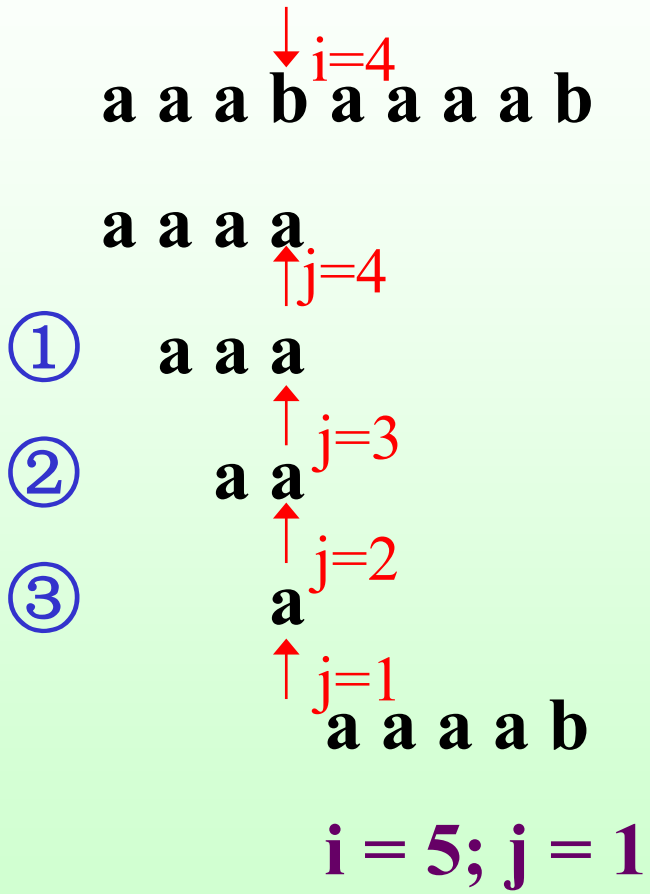
```
void get_next(SString T, int &next[])
{
    i= 1; next[1] = 0; j = 0;
    while( i<T[0]){
        if(j==0 || T[i] == T[j]){
            ++i; ++j;
            next[i] = j;
        }
        else
            j = next[j];
    }
}
```

●KMP算法的时间复杂度

设主串s的长度为n,模式串t长度为m,在KMP算法中求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)。

KMP算法最大的特点就是主串的指针不需要回溯，整个匹配过程只需从头到尾扫描一遍，这对于处理需要从外设输入的庞大数据很有效，可以一边读入一边匹配。

- **next函数的改进**



j	1	2	3	4	5
模式	a	a	a	a	b
next[j]	0	1	2	3	4
nextval[j]	0	0	0	0	4

next[j] = k，而$p_j=p_k$，则 主串中s_i和p_j不等时，不需再和p_k进行比较，而直接和$p_{next[k]}$进行比较。

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
模式串	a	b	c	a	a	b	b	c	a	b	c	a	a	b	d	a	b
next[j]	0	1	1	1	2	2	3	1	1	2	3	4	5	6	7	1	2
nextval[j]	0	1	1	0	2	1	3	1	0	1	1	0	2	1	7	0	1

- void get_nextval(SString T, int &nextval[])

```
{
    i= 1; nextval[1] = 0; j = 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];
    }
}
```

next[i] = j;

本章小结

本章基本学习要点如下:

(1) 理解串和一般线性表之间的差异。

(2) 重点掌握在顺序串上和链串上实现串的基本运算算法。

(3) 掌握串的模式匹配算法。

(4) 灵活运用串这种数据结构解决一些综合应用问题。