第四章

串

4.1 串的抽象数据类型的定义

4.2 串的表示和实现

4.3 串的模式匹配算法

4.1 串的抽象数据类型的定义如下：

串是有限长的字符序列，由一对单引号相括，如: 'a string'

ADT String {

数据对象：

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

数据关系：

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

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。

例如：StrCompare('data', 'state') < 0

StrCompare('cat', 'case') > 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。

例如： Concate(T, 'man', 'kind')

求得 T = 'mankind'

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, 'commander', 4, 3)

求得 sub = 'man' ；

SubString(sub, 'commander', 1, 9)

求得 sub = 'commander'；

SubString(sub, 'commander', 9, 1)

求得 sub = 'r'；

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，

$1 \le pos \le StrLength(S)$。

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。

“**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 = 'abcaabcaaabc'，T = 'bca'

Index(S, T, **1**) = 2；

Index(S, T, **3**) = 6；

Index(S, T, **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：

假设 S = 'abcaabcaaabca'，T = 'bca'

若 V = 'x'，则经置换后得到

S = 'axaxaax'

若 V = 'bc'，则经置换后得到

S = 'abcabcaabc'

StrInsert (&S, pos, T)

初始条件：串S和T存在，

$1 \le pos \le StrLength(S) + 1$。

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≤pos≤StrLength(S)-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，

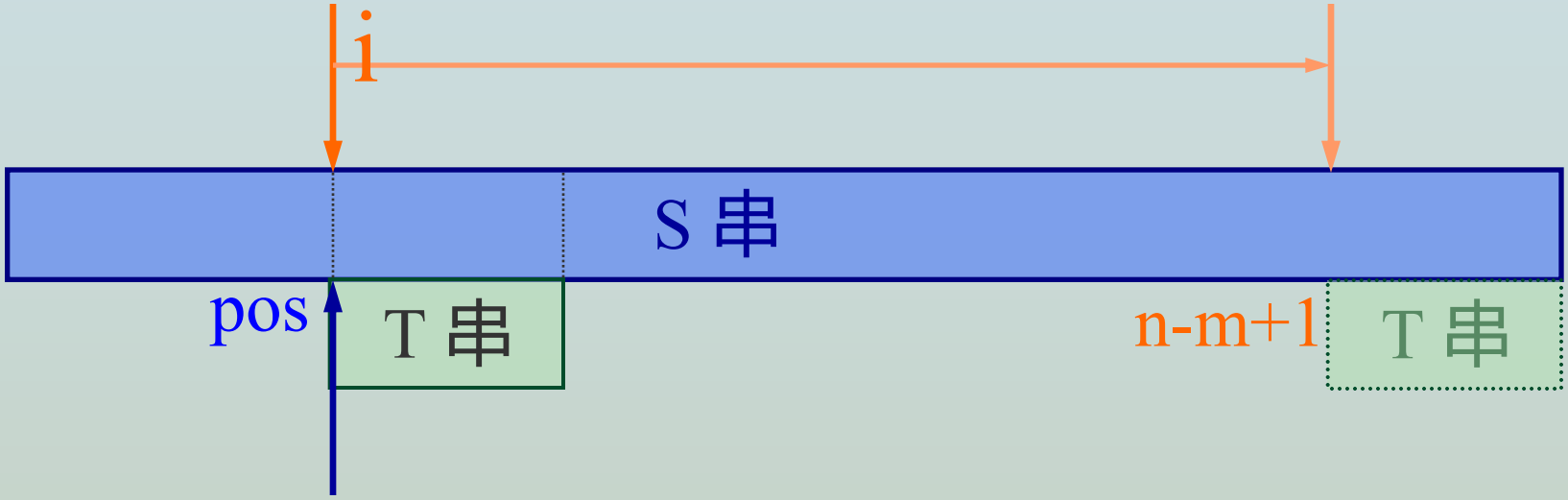
串赋值StrAssign、串复制Strcopy、串比较StrCompare、求串长StrLength、串联接Concat以及求子串SubString等六种操作构成串类型的最小操作子集。

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，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)。

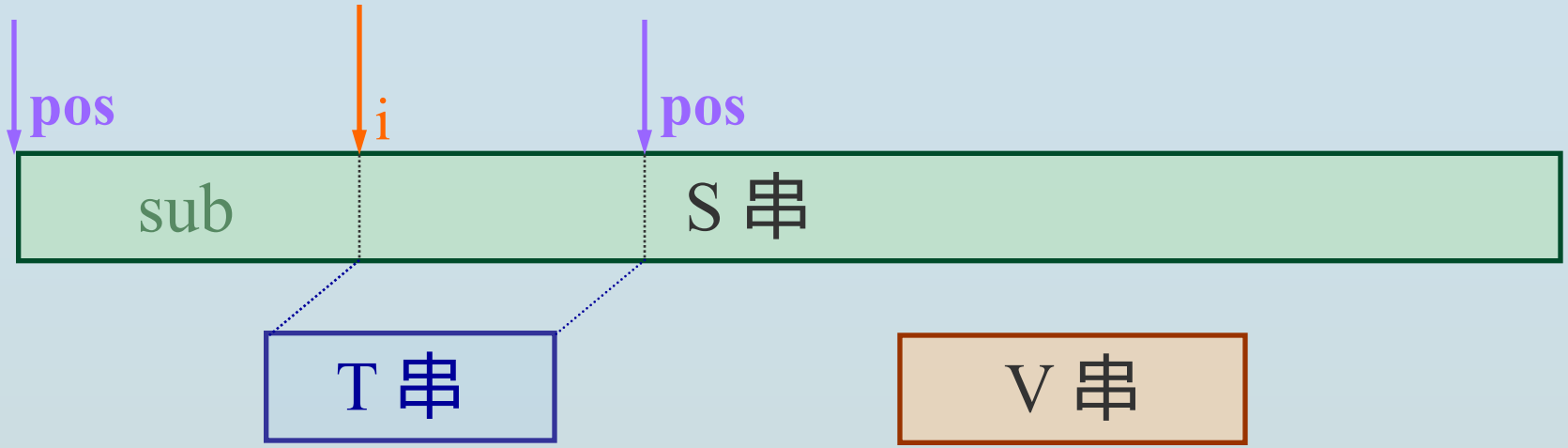
算法的基本思想为：

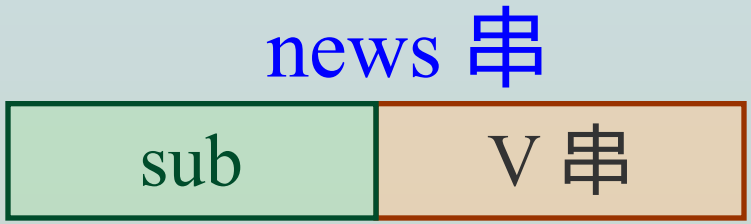
StrCompare(SubString(S, i, StrLength(T)),T) $\stackrel{?}{=}0$



```
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 ;
     } // 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：





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 串的表示和实现

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。

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

```
#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
```

特点:

- **串**的实际长度可在这个予定义长度的范围内随意设定，超过予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”**。

例如：串的联接算法中需分三种情况处理：

```
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]];
else if (S1[0]<MAXSTRSIZE) { // 截断
  T[1..S1[0]] = S1[1..S1[0]];
  T[S1[0]+1..MAXSTRLEN]
    T[0..MAXSTRLEN] = S1[0..MAXSTRLEN];
                  // T[0] == S1[0] == MAXSTRLEN
      uncut = FALSE;          }
return uncut;
} // Concat
```

二、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;
```

通常，C语言中提供的串类型就是以这种存储方式实现的。系统利用函数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**”。

C语言中的串以一个空字符为结束符，串长是一个隐含值。

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。

```
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
```

```
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;
  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
```

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

三、串的块链存储表示

也可用链表来存储串值，由于串的数据元素是一个字符，它只有 8 位二进制数，因此用链表存储时，通常一个结点中存放的不是一个字符，而是一个子串。

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

```
#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;
```

实际应用时，可以根据问题所需来设置结点的大小。

例如: 在编辑系统中，整个文本编辑区可以看成是一个串，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联接。

4.3 串的模式匹配算法

这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，
$1 \le pos \le StrLength(S)$。

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。

下面讨论以定长顺序结构表示串时的几种算法。

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

```
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
```

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) {
 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 { // 检查中间字符的匹配情况   }
 }
 return 0;
}
```

```
k = 1;  j = 2;
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
```

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)

当 S[i] <> 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]

定义：模式串的*next*函数

$$next[j]=\begin{cases} 0 & \text{当}j=1\text{时} \\ \mathrm{Max}\{k \mid 1<k<j \\ \text{且}'p_1p_2\cdots p_{k-1}'='p_{j-k+1}\cdots p_{j-1}'\} \\ 1 & \text{其它情况} \end{cases}$$

```
int Index_KMP(SString S, SString T, int pos) {
    //  1≤pos≤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
```

求*next*函数值的过程是一个递推过程，分析如下:

已知：next[1] = 0；

假设：next[j] = k；又 T[j] = T[k]

则： next[j+1] = k+1

若： T[j] ≠ T[k]

则需往前回朔，检查 T[j] = T[？]

这实际上也是一个匹配的过程，

不同在于：主串和模式串是同一个串

```
void get_next(SString &T, int &next[] ) {
  // 求模式串T的next函数值并存入数组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];
  }
} // get_next
```

还有一种特殊情况需要考虑：

例如：

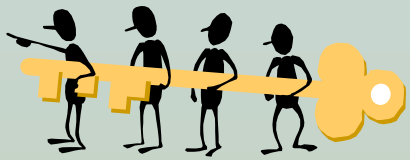
S = 'aaabaaaabaaaabaaaabaaaab'

T = 'aaaab'

next[j]=01234

nextval[j]=00004

```
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])  next[i] = j;
            else  nextval[i] = nextval[j];
        }
        else  j = nextval[j];
    }
} // get_nextval
```



本章学习要点

1. 熟悉串的七种基本操作的定义，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。

5. 了解串操作的应用方法和特点。