第四章 串

- 串类型的定义
- 串的表示和实现
- 串的模式匹配算法

串类型的定义

- 串即字符串，是由零个或多个字符组成的有限序列，是数据元素为单个字符的特殊线性表。

记为：　　$s =$ ‘$a_1 a_2 \ldots a_n$’　　(n≥0)

串名　　串值

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。

串类型的定义

- 相关术语：

  串长：串中字符个数（n≥0）， n=0 时称为空串。

  空白串：由一个或多个空格符组成的串。

  子串：串s中任意个连续的字符序列叫s的子串;

  s 叫主串。

  子串位置：子串的第一个字符的序号。

  字符位置：字符在串中的序号。

  串相等：串长度相等，且对应位置上字符相等。

串类型的定义

- a =‘BEI’ b =‘JING’

  c = ‘BEIJING’ d = ‘BEI JING’

  问：① 他们各自的长度？

  ② a是哪个串的子串？在主串中的位置是多少？

  答： a 、 b 、 c 、 d 长度分别为3、4、 7、8

- 空串和空白串有无区别？

  答：有区别。空串(Null String)是指长度为零的串；而空白串(Blank String),是指包含一个或多个空白字符‘ ’(空格键)的字符串。

串类型的定义

ADT String {

数据对象：D＝{ a_i ｜ a_i ∈CharacterSet, i=1,2,...,n, n≥0 }

数据关系：R1＝{ < a_{i-1} , a_i > | a_{i-1} , a_i ∈D, i=2,...,n }

基本操作：

StrAssign (&T, chars)

初始条件：chars 是串常量。

操作结果：赋于串T的值为 chars。

StrCopy (&T, S)

初始条件：串 S 存在。

操作结果：由串 S 复制得串 T。

串类型的定义

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。

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。

串类型的定义

StrLength (S)

初始条件：串 S 存在。

操作结果：返回串 S 序列中的字符个数，即串的长度。

ClearString (&S)

初始条件：串 S 存在。

操作结果：将 S 清为空串。

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。

串类型的定义

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。

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。

串类型的定义

Replace (&S, T, V)

初始条件：串 S，T 和 V 存在，T 是非空串。

操作结果：用V替换主串S中出现的所有与T相等的不重叠的子串。

StrInsert (&S, pos, T)

初始条件：串 S 和 T 存在，1≤pos≤StrLength(S)＋1。

操作结果：在串 S 的第 pos 个字符之前插入串 T。

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。

} **ADT** String

串的表示和实现

- 串与线性表的运算有所不同，是以“串的整体”作为操作对象，例如查找某子串，在主串某位置上插入一个子串等。
- 串有三种机内表示方法：
  - 定长顺序存储表示
  - 堆分配存储表示
  - 串的块链存储表示

串的表示和实现 --定长顺序存储表示

- 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在运算时，其基本操作为“字符序列的复制”。

串的表示和实现 --定长顺序存储表示

```
Status Concat(SString &T, SString S1, SString S2) {
  // 算法4.2。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
    for (i=1; i<=S1[0]; i++) T[i] = S1[i];
    for (i=1; i<=S2[0]; i++) T[i+S1[0]] = S2[i];
    T[0] = S1[0]+S2[0];
    uncut = TRUE;
  } else if (S1[0] < MAXSTRLEN) { // 截断
    for (i=1; i<=S1[0]; i++) T[i] = S1[i];
```


串的表示和实现 --定长顺序存储表示

```
    for (i=S1[0]+1; i<=MAXSTRLEN; i++)
      T[i] = S2[i-S1[0]];
      T[0] = MAXSTRLEN;
      uncut = FALSE;
    } else {                        // 截断(仅取S1)
      for (i=0; i<=MAXSTRLEN; i++) T[i] = S1[i];
      uncut = FALSE;
    }
    return uncut;
} // Concat
```


串的表示和实现 --堆分配存储表示

- 堆分配存储表示

```
typedef struct {
  char *ch; // 若是非空串，则按串实用长度分配
            //存储区，否则 ch 为NULL
  int  length;  // 串长度
} HString;
```


串的表示和实现 --堆分配存储表示

```
Status StrAssign(HString &T, char *chars){
        if (T.ch) free(T.ch);
        for (i=0, c=chars; c; ++i, ++c);       //求串长度
        if (!i) {T.ch = NULL; T.length = 0;}
        else{
            if (!(T.ch = (char*)malloc(i*sizeof(char))))
            exit(OVERFLOW);
           T.ch[0..i-1] = chars[0..i-1];
           T.length =i;
        }
        return OK;
}//StrAssign
```


串的表示和实现 --堆分配存储表示

```
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)  delete(T.ch);        // 释放旧空间
  if (!(T.ch =new char[S1.length+S2.length]))
      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
```


串的表示和实现 --堆分配存储表示

```
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)  delete (Sub.ch); // 释放旧空间
  if (!len)
    { Sub.ch = NULL;  Sub.length = 0; } // 空子串
  else {
   if(!(Sub.ch = new char[len])) return ERROR;
   Sub.ch[0..len-1] = S.ch[pos-1..pos+len-2];
   Sub.length = len;} // 完整子串
  return OK;} // SubString
```


串的表示和实现 --串的块链存储表示

☑ 链式存储特点：用链表存储串值，易插入和删除。

方法1：链表结点（数据域）大小取1

1/2

方法 2：链表结点（数据域）大小取n(例如n=4)

9/15=3/5

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

串的表示和实现 --串的块链存储表示

- 若数据元素很多，用法2存储更优—称为块链结构
- 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;
```


串的模式匹配算法

- 子串定位运算又称为模式匹配(Pattern Matching)或串匹配(String Matching)，它是串处理系统中最重要的操作之一。
- 算法种类：
  - BF算法 （又称古典或经典的、朴素的、穷举的）
  - KMP算法（特点：速度快）

串的模式匹配算法

- BF算法设计思想：
  - 将主串的第pos个字符和模式的第1个字符比较，
    - 若相等，继续逐个比较后续字符；
    - 若不等，从主串的下一字符（pos+1）起，重新与第一个字符比较。
  - 直到主串的一个连续子串字符序列与模式相等 。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 。

串的模式匹配算法

- BF算法实现：

```
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}   //继续比较后续字符
    else {i=i-j+2; j=1;} //指针回溯到下一首位,重新开始匹配
    }
  if(j>T[0]) return i-T[0];  //子串结束，说明匹配成功
  else return0;
}//Index
```


串的模式匹配算法

- BF算法性能分析

一般情况下BF算法的时间复杂度为O(n+m)；

BF匹配算法的最坏时间复杂度O(n*m)。

- KMP算法（特点：速度快）
  - KMP算法设计思想
  - KMP算法的推导过程
  - KMP算法的实现 （关键技术:计算next[j]）
  - KMP算法的时间复杂度

串的模式匹配算法

- 思想：
  - 匹配过程中指针 i 没有回溯。
  - KMP算法的核心思想是利用已经得到的部分匹配信息来进行后面的匹配过程 。

串的模式匹配算法

$s_1 s_2 \ldots s_{i-1}\ s_i \ldots s_n$

$p_1 p_2 \ldots p_{j-1} p_j \ldots p_m$

$s_i \neq p_j$ 此时主串的i应该与模式串的第k个字符相比较，即

$p_1 p_2 \ldots p_{k-1} = s_{i-k+1} s_{i-k+2} \ldots s_{i-1}$ 而

$p_{j-k+1} p_{j-k+2} \ldots p_{j-1} = s_{i-k+1} s_{i-k+2} \ldots s_{i-1}$ 所以

$p_1 p_2 \ldots p_{k-1} = p_{j-k+1} p_{j-k+2} \ldots p_{j-1}$

串的模式匹配算法

$$next[j]=\begin{cases}0 & \text{当}j=1\text{时}\\ \text{Max}\{k\mid 1<k<j\text{且}'p_1p_2\cdots p_{k-1}'='p_{j-k+1}\cdots p_{j-1}'\} & \text{当此集合不空时}\\ 1 & \text{其他情况}\end{cases}$$

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
Next[j]	0	1	1	2	2	3	1	2

串的模式匹配算法

```
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
    }//while
   if (j > T[0])  return  i-T[0];    // 匹配成功
   else return 0;
} // Index_KMP
```


串的模式匹配算法

```
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
```


串的模式匹配算法

- KMP算法的时间复杂度可以达到O(m+n)
- 注意：由于BF算法在一般情况下的时间复杂度也近似于O(n+m)，所以至今仍被采用。