

# 第四章 串

- 4.1 串的基本概念
- 4.2 串存储和实现
- 4.3 串的匹配算法

## 基本内容

- 1 串的有关概念 串的基本操作
- 2 串的定长顺序存储结构，堆分配存储结构；
- 3 串的基本操作算法；
- 4 串的模式匹配算法；

## 学习要点

- 1 了解串的基本操作，了解利用这些基本操作实现串的其它操作的方法；
- 2 掌握在串的堆分配存储结构下，串的基本操作算法；
- 3 掌握串的模式匹配算法；

## 4.1 串类型的定义

串 (String) 是零个或多个字符组成的有限序列。一般记作

$$S = 'a_1 a_2 a_3 \dots a_n'$$

其中S 是串名，单引号括起来的字符序列是串值；  
 $a_i$  ( $1 \leq i \leq n$ ) 可以是字母、数字或其它字符；

串的应用非常广泛，许多高级语言中都把串作为基本数据类型。在事务处理程序中，顾客的姓名、地址, 货物的名称、产地可作为字符串处理，文本文件中的每一行字符等也可作为字符串处理。

下面是一些串的例子：

(1) `a = 'LIMING'`

(2) `b = '12345'`

(3) `c = 'DATA STRUCTURES'`

(4) `d = ''`

(5) `e = ' '`

说明：

- ◆ 串中所包含的字符个数称为该串的长度。
- ◆ 长度为零的串称为空串 (Empty String)，它不包含任何字符。
- ◆ 通常将仅由一个或多个空格组成的串称为空格串 (Blank String)

**注意：** 空串和空格串的不同，例如上面(4)中的串d和(5)中的串e 分别表示长度为0的空串和长度为1的空格串。

# 串的有关术语

## 1) 子串、主串

串中任意连续的字符组成的子序列称为该串的子串，包含子串的串相应地称为主串。

## 2) 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。

## 3) 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；

例如      A='This is a string'    B='is'

则B是A的子串，A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是3。因此，称B在A中的序号（或位置）为3。

特别地，空串是任意串的子串，任意串是其自身的子串。

## 串的基本操作

串的逻辑结构与线性表一样，都是线性结构。但由于串的应用与线性表不同，串的基本操作与线性表有很大差别。

### 1) 串赋值操作 `StrAssign( &T, chars)`

功能：将串常量chars的值赋给串变量T；

### 2) 复制串操作 `StrCopy(&T,S)`

功能：由串变量S复制得到串变量T；

### 3) 判空操作 `StrEmpty(S)`

功能：若S为空串，则返回TRUE，否则返回FALSE

### 4) 串比较操作 `StrCompare( S, T)`

功能：若 $S > T$ ，则返回值 $> 0$ ；若 $S = T$ ，则返回值 $= 0$ ；若 $S < T$ ，则返回值 $< 0$

### 5) 串置空操作 `ClearString( &S)`

功能：将S清为空串

## 6) 求串长操作 StrLenght( &S)

功能：返回S的元素个数，称为串的长度。

## 7) 串连接操作 Concat( &T, S1, S2)

功能：由S1和S2连接组成新串，用T返回新串；

## 8) 求子串操作 SubString( &Sub, S, pos, len)

功能：用Sub返回串S的第pos个字符起长度为len 子串；

## 9) 求子串位置操作 Index( S, T, pos )

功能：返回S中第pos个字符之后与T相同的子串的位置，若不存在返回0；

## 10) 串插入操作 StrInsert( &S, pos , T)

功能：将串T插入到串S的第pos个字符之前；

## 11) 串删除操作 StrDelete( &S, pos , len)

功能：从串S中删除第pos个字符起长度为len 的子串。

## 4.2 串的实现和表示

### 一、串的存储

#### 1、定长顺序存储结构

定长顺序存储表示,也称为静态存储分配的顺序表。定长顺序存储结构类似于C语言的字符数组,以一组地址连续的存储单元存放串值字符序列,其类型说明如下:

```
#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
```



## 2、堆分配存储

堆分配存储类似于线性表的顺序存储结构，以一组地址连续的存储单元存放串值字符序列，其存储空间是在程序执行过程中动态分配的。所以也称为动态存储分配的顺序表。

在C语言中，存在一个称之为“堆”的自由存储区，并由C语言的动态分配函数管理。利用函数`malloc()`为每个新产生的串分配一块实际串长所需的存储空间，若分配成功，则返回一个指向起始地址的指针，作为串的基址，同时，为了以后处理方便，将串长也作为存储结构的一部分。串的这种存储结构本教材中称为堆分配存储。

## 堆分配存储的类型说明

```
Typedef struct {
```

```
    char    *ch;    //指针域，指向存放串值的存储空间基址
```

```
    // 若是非空串，则按串长分配存储区，否则ch为NULL
```

```
    int     length; // 整型域：存放串长
```

```
}Hstring;
```

在这种存储结构下，串操作仍是基于“**字符序列的复制**”进行的。例如，如串插入操作StrInsert(&S,pos,T)（将串T插入到串S的第pos字符之前）的算法是，为**串S重新分配大小等于串S和串T长度之和的存储空间**，然后进行将S和T串值复制到新分配存储空间中**算法4.4**。

由于堆分配存储结构的串既有顺序存储结构的特点，处理方便，操作中对串长又没有任何限制，更显灵活，因此在串处理的应用程序中常被采用。下面我们给出在堆分配存储结构下，串的部分基本操作算法。

## 串插入算法

```
StrInsert(HString &S,int pos ,HString T) {
```

```
    //为串S重新分配大小等于串S和串T长度之和的存储空间，将S和  
    //T串值复制到新分配存储空间中;
```

```
if (pos <1 || pos>S.length+1) return ERROR;           //参数不合法
```

```
if (T.length) {    // 若T非空，为串S重新分配存储空间,并插入T
```

```
    if (!(S.ch=(char * ) realloc ((S.ch,S.length+T.length) * sizeof(char))));
```

```
        exit(OVERFLOW);
```

```
    for(i=S.length-1;i>=pos-1; --i)
```

```
        S.ch[i+T.length]=S.ch[i];    //将S的第pos个字符及后面的字符后  
                                     // 移，为插入T腾出位置
```

```
    S.ch[pos-1..pos+T.length-2]=T.ch[0..T.length-1];    //插入T
```

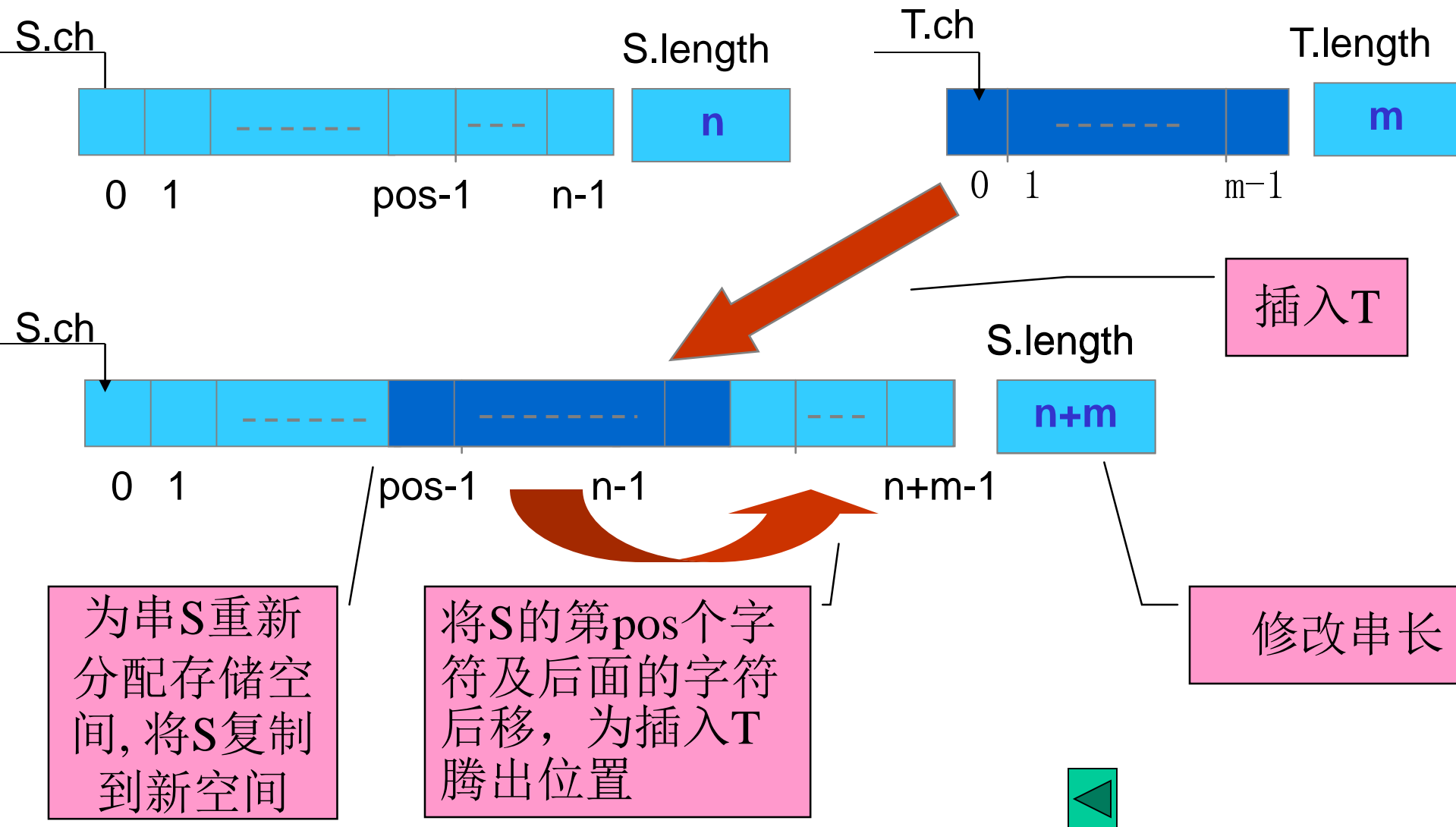
```
    S.length+=T.length;
```

```
}
```

```
    return OK;
```

```
}SubString
```



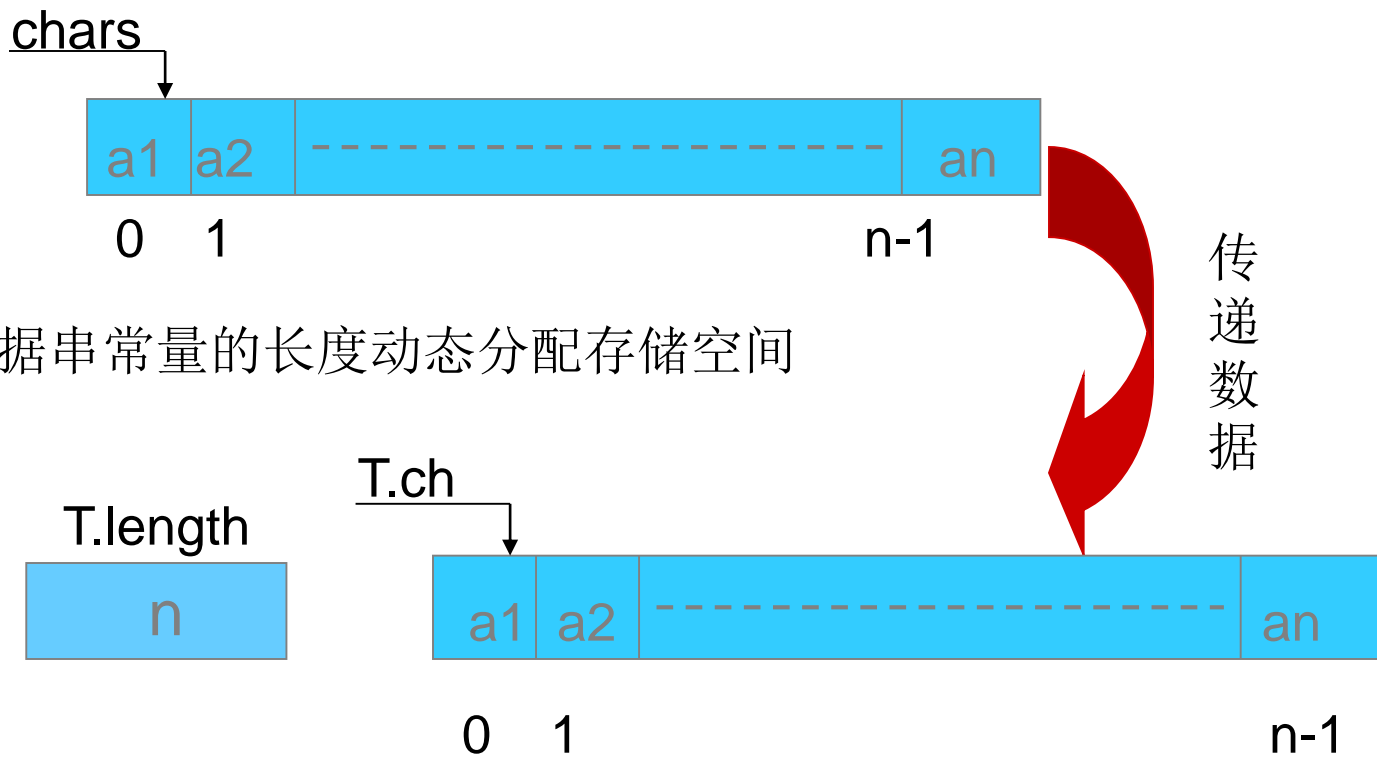


## 二 串基本操作算法

### 串赋值算法

```
Status StrAssign(HString &T, char *chars) {  
    //生成一个其值等于串常量chars 的串T,其中chars采用的是标准C  
    //的表示法, T采用是数据结构课程中的表示法。  
    if (T.ch) free(T.ch);    //若T.ch非空, 释放T.ch所指向的存储空间  
    for (i=0, c=chars; c; ++i, ++c);  
        //求chars 的长度i, 没有使用strlen( chars)  
    if (!i) {T.ch =NULL; T.length=0;}    //若i=0,生成空串T  
    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
```

串常量



串赋值操作图示

## 串比较算法

```
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;  
} //StrCompare
```

## 串置空算法

```
Status ClearString(HString &S) {  
    //将S清为空串。  
    if (S.ch) {free(S.ch); S.ch=NULL;} //若S.ch非空, 释放S.ch所指  
                                         //存储空间, 并且S.ch=null  
    S.length = 0;  
    return OK;  
} //ClearString
```



## 串连接算法

```
Status Concat(HString &T, Hstring S1, HString S2){
```

```
    //用T返回由S1和S2连接而成的新串。
```

```
    if (T.ch) free(T.ch);    //若T.ch非空，释放T.ch所指向的存储空间
```

```
    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;
```

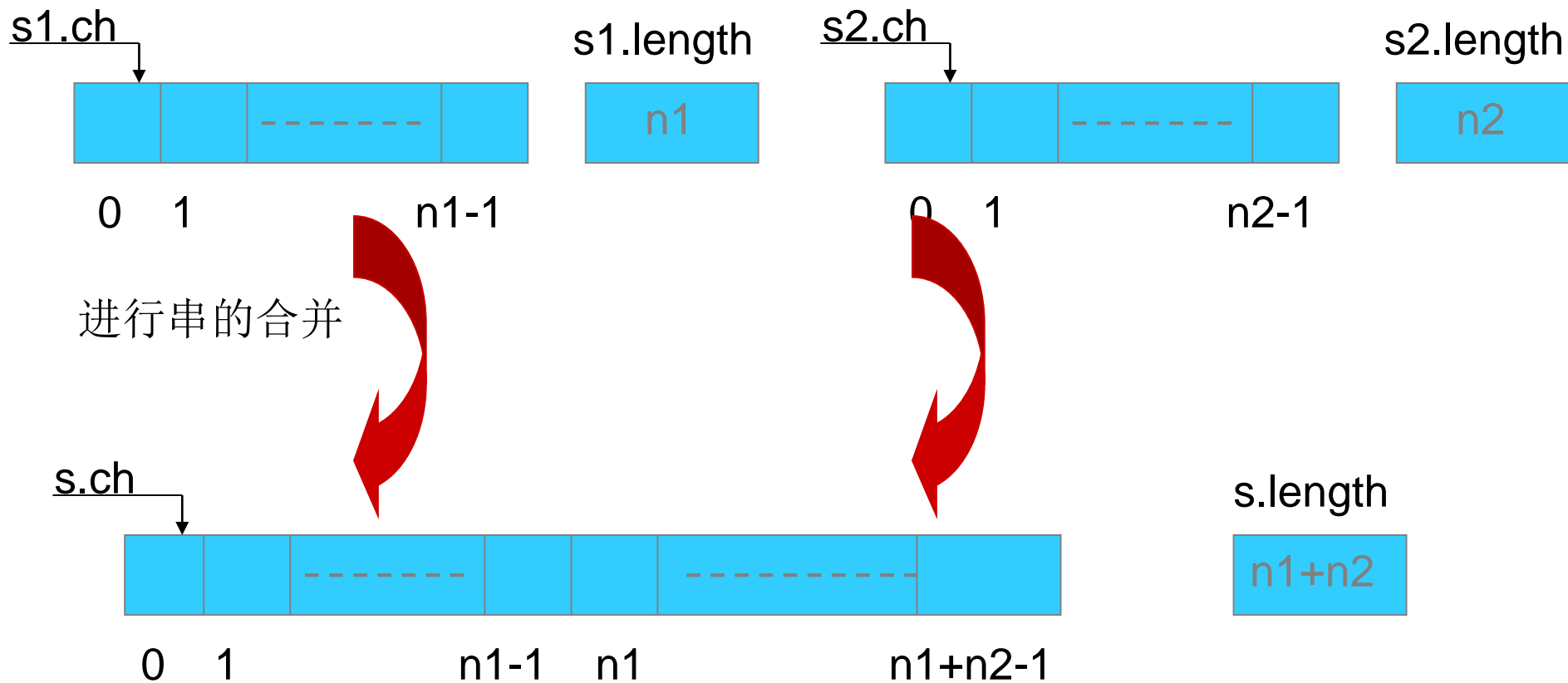
```
    T.ch[S1.length..T.length-1]=S2.ch[0..S2.length-1];
```

```
    return OK;
```

```
}//Concat
```

串s1

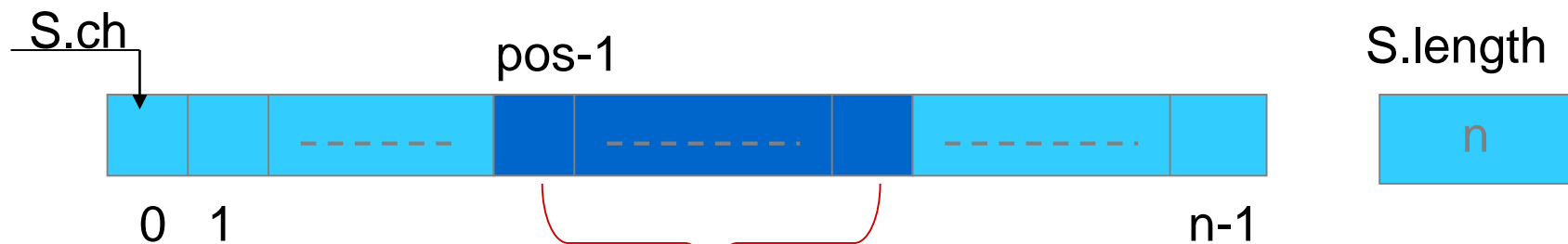
串s2



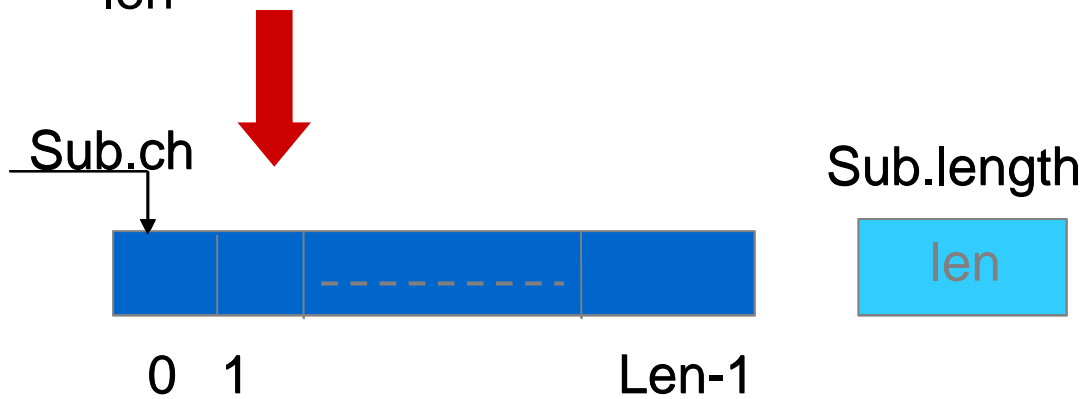
串连接图示

## 求子串算法

```
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len的子串。  
    //其中,  $1 \leq \text{pos} \leq \text{StrLength}(S)$  且  $0 \leq \text{len} \leq \text{StrLength}(S) - \text{pos} + 1$   
if (pos < 1 || pos > S.length || len < 0 || len > S.length - pos + 1) return ERROR;  
    //参数不合法  
if (Sub.ch) free (Sub.ch); //若Sub.ch非空, 释放Sub.ch所指向存储  
if (!len) {Sub.ch=NULL; Sub.length=0;} //若len=0, Sub为空子串  
    else{ //复制子串  
        Sub.ch=(char * ) malloc (len * sizeof(char));  
        Sub.ch[0..len-1] =S[pos-1..pos+len-2];  
        Sub.length=len;  
    }  
    return OK;  
}SubString
```



动态分配存储空间



### 4.2.3 串的块链式存储结构

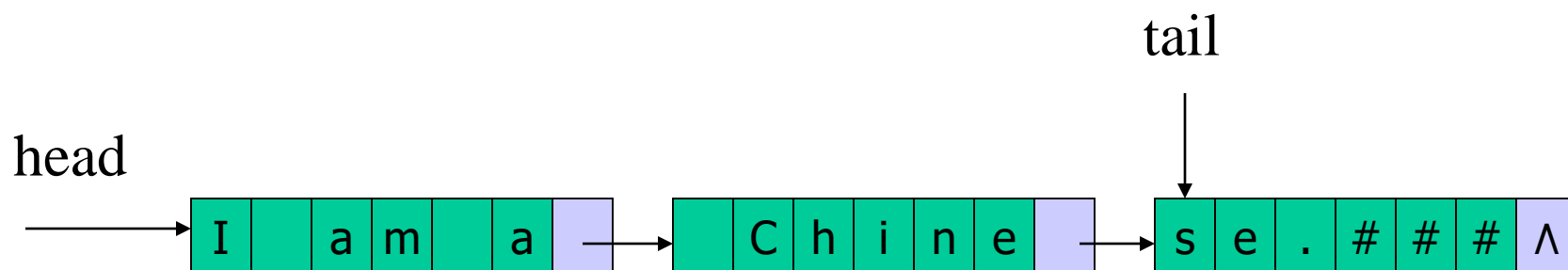
顺序串上的插入和删除操作不方便，需要移动大量的字符。因此，我们可用单链表方式来存储串值，串的这种链式存储结构简称为链串。

```
typedef struct Chunk{  
    char    data;  
    struct Chunk *next;  
} Chunk ;
```

一个链串由头指针唯一确定。

这种结构便于进行插入和删除运算，但存储空间利用率太低。

为了提高存储密度，可使每个结点存放多个字符。通常将结点数据域存放的字符个数定义为“**结点的大小**”，显然，当结点大小大于 1 时，串的长度不一定正好是结点的整数倍，因此要用特殊字符来填充最后一个结点，以表示串的终结。**还可设一尾指针，并给出当前串的长度**，此串存储结构为**块链结构**，对于结点大小不为 1 的链串，其类型定义只需对上述的结点类型做简单的修改即可。



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

typedef struct Chunk { // 结点结构
    char ch[CHUNKSIZE];
    struct Chunk *next;
} Chunk;

typedef struct { // 串的链表结构
    Chunk *head, *tail; // 串的头和尾指针
    int curlen; // 串当前长度
} LString;
```

## 4.3 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)，此运算的应用非常广泛。例如，在文本编辑程序中，我们经常要**查找某一特定单词在文本中出现的位置**。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。

子串定位函数的定义：

**INDEX (S, T, pos)**

**初始条件：**串S和T存在，T是非空串， $1 \leq \text{pos} \leq \text{StrLength}(S)$ 。

**操作结果：**若主串S中存在和串T值相同的子串，则返回它为主串S中第pos个字符之后第一次出现的位置；否则函数值为0。

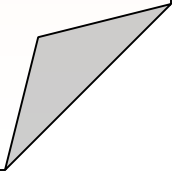


利用串的基本操作实现,参见P72 算法4.1:

利用判等、求串长和求子串等操作实现定位函**Index(S,T,pos)**

**算法的基本思想为:** 在主串S中取从第i(i的初值为pos)个字符起、长度和串T相等的子串和串T比较,若相等,则求得函数值为i,否则i值增1直至串S中不存在和串T相等的子串为止。

```
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
```



下面讨论以**定长顺序结构**表示串时的算法。

## 1.简单算法

算法的基本思想为：利用**计数指针i和j**指示主串S和模式串T中**当前正待比较的字符位置**。从主串S的第pos个字符起和模式T的第一个字符比较，若相等，则继续逐个比较后续字符，否则从主串的下一个字符起再重新和模式的字符比较之。依次类推，直至模式T中的每个字符依次和主串S中的一个连续的字符序列相等，则称**匹配成功**，函数值为模式T中第一个字符相等的字符在主串S中的序号，否则称**匹配不成功**，函数值为零。图4.3展示了模式和主串S的匹配过程。

在匹配的过程中，一旦发现出现字符不匹配，则整个模式相对于原来的位置右移一位，如算法演示所示。

```
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， $1 \leq \text{pos} \leq \text{StrLength}(S)$ 。

```
    i = pos;    j = 1; // i=pos, 主串匹配的起始位置
```

```
    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i后退(至当前匹配起始位置的  
                                // 下一位置) 重新开始匹配
```

```
    }
```

```
    if (j > T[0]) return i-T[0]; // 匹配成功，返回子串T的位置
```

```
    else return 0;
```

```
} // Index
```

例S='ababcabcac'    T='abc'    pos=4    index(S, T, pos) 返回值为6

**E.g:** 说明最坏情况下时间复杂性的  $O(n * m)$  的实例。

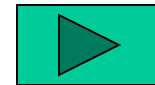
每比较  $m = 3$  次，移动模式一次。最后在主串的  $n-m+1$  找到主串，比较  $(n-m+1) * m$  次



## 二、Knuth-Morris-Pratt 模式匹配算法（KMP 算法）

**起因：**降低时间代价，从最坏情况下的  $O(n * m)$  降低到  $O(n + m)$ ；

**改进：**每当一趟匹配过程中出现字符不等时，不需回溯  $i$  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。



**NEXT[j]** 的定义：当失配点发生在  $P_j$  处时，主串中的失配点将和模式中的哪一个字符进行比较。那一个字符的位置定义为 **NEXT[j]**。

$$\text{NEXT}[j] = \begin{cases} 0 & \text{如果 } j=1 \\ \text{MAX} \{ k \mid 1 < k < j \text{ 且 } P_1 \dots P_{k-1} = P_{j-k+1} \dots P_{j-1} \} & \\ 1 & \end{cases}$$



```
int Index_KMP(SString S, SString T, int pos) {  
// 利用模式串T的next函数求T在主串S中第pos个 字符之后的位  
//置的KMP算法。其中， T非空，  $1 \leq \text{pos} \leq \text{StrLength}(S)$   
    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
```



## NEXT[j] 函数值的求法:

1、由定义,  $\text{next}[1] = 0$

2、若  $\text{next}[j] = k$ , 求  $\text{next}[j+1] = ?$

由已知可得:

$$P_1 P_2 \dots P_{k-1} = P_{j-k+1} P_{j-k+2} \dots P_{j-1}$$

①、若  $P_k = P_j$ ; 则

$$P_1 P_2 \dots P_{k-1} P_k = P_{j-k+1} P_{j-k+2} \dots P_{j-1} P_j$$

所以,  $\text{next}[j+1] = k + 1$

②、若  $P_k \neq P_j$ ;

```
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
```

```
void get_nextval(SString &T, int &nextval[]) {  
    // 求模式串T的next函数修正值并存入数组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
```

# 第四章 习题一

P27      4.3

4.6