



数据结构

第四章 串

主讲：陈锦秀

厦门大学信息学院计算机系

第四章 串

4.1 串类型的定义

4.2 串的实现和表示

4.2.1 定长顺序存储表示

4.2.2 堆分配存储表示

4.2.3 串的块链存储表示

4.3 串的模式匹配算法

4.3.1 求子串位置的定位函数

4.3.2 模式匹配的一种改进算法

4.4 串操作应用举例

4.1 串类型的定义

4.1.1 串和基本概念

- 串 (**String**) 是零个或多个字符组成的**有限序列**。一般记作 **S** = “**a₁a₂a₃...a_n**”
 - 其中S是串名，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**空白串** (**Blank String**)

注意：空串和空白串的不同，例如 “ ” 和 “ ” 分别表示长度为1的空白串和长度为0的空串。

4.1.1 串和基本概念

- 串中任意个连续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**。

- 通常将子串在主串中**首次出现时**的该子串的首字符对应的主串中的序号，定义为子串在主串中的序号（或位置）。
- 例如，设A和B分别为

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。

4.1.1 串和基本概念

- 通常在程序中使用的串可分为：**串变量**和**串常量**。
- **串常量**和整常数、实常数一样，在程序中只能被引用但不能改变其值，即**只能读不能写**。
 - 通常串常量是由直接量来表示的，例如语句 `Error(“OVERFLOW”)` 中 “OVERFLOW” 是直接量。
 - 但有的语言允许对串常量命名，以使程序易读、易写。如C++中，可定义

```
const char path[] = “dir/bin/appl”;
```

这里path是一个串常量，对它只能读不能写。
- **串变量**和其它类型的变量一样，其**取值是可以改变的**。

4.1.2 串的抽象数据定义

串的抽象数据类型定义：书P₇₁

ADT String {

数据对象：

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

数据关系：

$R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, \\ i=2,\dots,n \}$

基本操作：

.....

} ADT String

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)

串和线性表的异同：

- 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。
- 然而，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，例如：在线性表中查找某个元素等。
 - 在串的基本操作中，通常以“串的整体”作为操作对象。例如：在串中查找某个子串、插入一个子串等。

4.1.2 串的抽象数据定义

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。

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；

4.1.3 串的基本操作

- 定义下列几个变量:

```
char s1[20]="dirtreeformat";
```

(1) 求串长(length)

```
int strlen(char *s); //求串的长度
```

例如: **printf("%d",strlen(s1));** 输出**13**

4.1.3 串的基本操作

- `char s1[20]="dirtreeformat";`
`char s3[30];`

(2) 串复制(copy)

`char *strcpy(char *to,char *from);`

//该函数将串from复制到串to中，并且返回一个指向串to的开始处的指针。

例如: `strcpy(s3,s1); //s3="dirtreeformat"`

4.1.3 串的基本操作

- `char s1[20]="dirtreeformat",s2[20]="file.mem";`
`char s3[30];`

(3) 联接(concatenation)

`char * strcat(char * to,char * from)`

//该函数将串from复制到串to的末尾，并且返回一个指向串to的开始处的指针。

例如: `strcat(s3,"/")`

`strcat(s3,s2); //s3="dirtreeformat/file.mem"`

4.1.3 串的基本操作

(4) 串比较 (compare)

int strcmp(char *s1,char *s2);

//该函数比较串s1和串s2的大小，当返回值小于0，等于0或大于0时分别表示s1<s2，s1=s2或s1>s2

例如: **result=strcmp("baker","Baker") result>0**

result=strcmp("12","12"); result=0

result=strcmp("Joe","Joseph"); result<0

4.1.3 串的基本操作

■ `char s2[20]="file.mem";`

`char *p;`

(5) 字符定位(index)

`char* strchr(char * s,char c);`

//该函数是找c在字符串中第一次出现的位置，若找到则返回该位置的地址指针，否则返回NULL。

例如: `p=strchr(s2, ".");` p 指向 “file”之后的位置。

`s2="file.mem"`

`if(p) strcpy(p, ".cpp"); s2="file.cpp"`

4.1.3 串的基本操作

- 上述串的操作是最基本的，其中后四个还有变种形式：
strncpy, strncat, strncmp。串的其余操作可由这些基本操作组合而成。

- 例1、求子串

求子串的过程即为复制字符序列的过程，将串s中的第pos个字符开始长度为len的字符复制到串sub中。

```
void substr(char * sub, char * s, int pos, int len)
```

```
{  
    if(pos<0 || pos>strlen(s)-1 || len<0)  
        error("parameter error")  
    strncpy(sub,&s[pos],len);  
}
```


• 子串为“串” 中的一个字符子序列

例如：

substr (sub, “commander”, 4, 3)

求得 sub = “man” ;

substr(sub, “commander”, 1, 9)

求得 sub = “commander”;

substr (sub, “commander”, 9, 1)

求得 sub = “r”;

4.1.3 串的基本操作

■ 例2、串的定位 $\text{index}(s,t,pos)$

- 在主串 s 中取从第 pos 个字符起、长度和串 t 相等的子串和 T 比较,
- 若相等, 则求得函数值为 pos ,
- 否则值增1直至 s 中不存在和串 t 相等的子串为止。

```
int index(char * s, char * t, int pos)
{
    char *sub;
    if(pos>0){
        n=strlen(s); m=strlen(t); i=pos;
        while(i<=n-m+1){
            substr(sub,s,i,m);
            if(strcmp(sub,t)!=0) ++i;
            else return(i); //返回子串在主串中的位置
        }
    }
    return(0); //s中不存在于t相等的子串
}
```

“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。

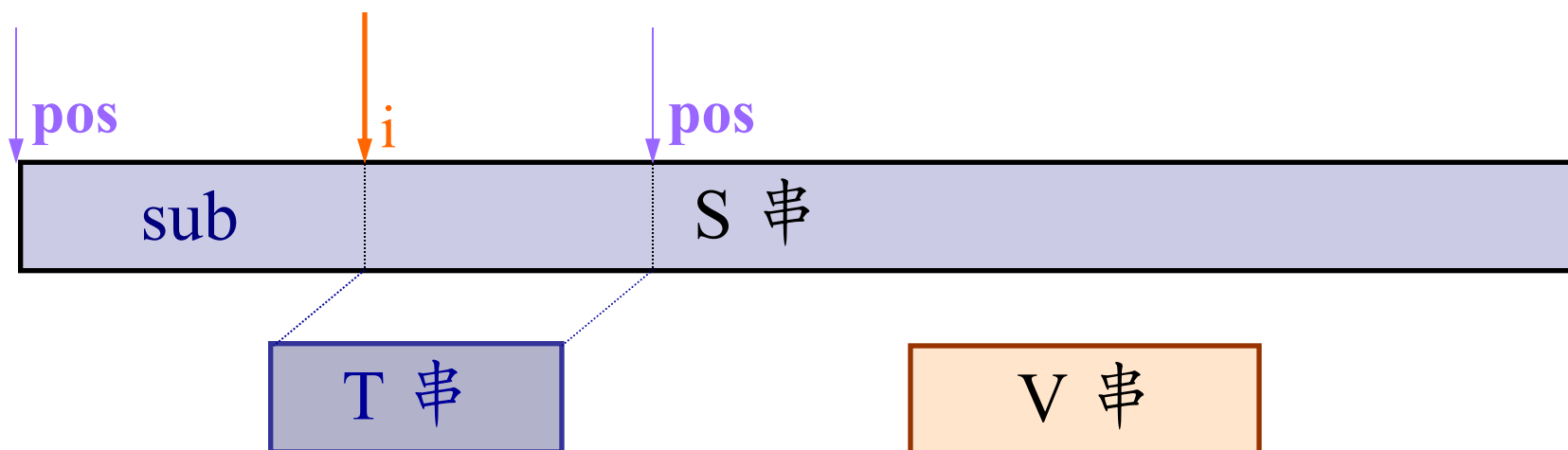
假设 $S = \text{“abcaabcaabc”}$, $T = \text{“bca”}$

$\text{Index}(S, T, 1) = 2;$

$\text{Index}(S, T, 3) = 6;$

$\text{Index}(S, T, 8) = 0;$

- 例3 串的置换函数 $\text{Replace}(\&S, T, V)$
- 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。



news 串



例如：

假设 $S = \text{'abcaabcaabca'}$, $T = \text{'bca'}$

若 $V = \text{'x'}$, 则经置换后得到

$S = \text{'axaxaax'}$

第四章 串

4.1 串类型的定义

4.2 串的实现和表示

4.2.1 定长顺序存储表示

4.2.2 堆分配存储表示

4.2.3 串的块链存储表示

4.3 串的模式匹配算法

4.3.1 求子串位置的定位函数

4.3.2 模式匹配的一种改进算法

4.4 串操作应用举例

4.2 串的实现

- 因为串是特殊的线性表，故其存储结构与线性表的存储结构类似。只不过组成串的结点是单个字符。
- 串有三种机内表示方法，下面分别介绍。
 - 定长顺序存储表示
 - 堆分配存储表示
 - 串的链式存储结构

4.2.1 定长顺序存储表示

- 定长顺序存储表示, 也称为静态存储分配的顺应表。它是用一组地址连续的存储单元来存放串中的字符序列。
- 定长顺序存储结构可以直接使用定长的字符数组来定义, 数组的上界预先给出:

```
#define MAXSTRLEN 255
```

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

```
SString s; //s是一个可容纳255个字符的顺序串
```

- 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”。

串长的计算??

4.2.1 定长顺序存储表示

- 0号单元存放串的长度。
- 在串值后面加一个不计入串长的结束标记字符：
 - 例如，C语言中以字符 `'\0'` 表示串值的终结，这就是为什么在上述定义中，串空间最大值`MAXSTRLEN+1`为256，但最多只能存放255个字符的原因，因为必须留一个字节来存放 `'\0'` 字符。
- 若不设终结符，也可用一个整数来表示串的长度，那么该长度减1的位置就是串值的最后一个字符的位置。此时顺序串的类型定义和顺序表类似：

```
typedef struct{  
    char ch[MAXSTRLEN];  
    int length;  
}SString; //其优点是涉及到串长操作时速度快。
```

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]];
```

```
    T[0] = S1[0]+S2[0]; uncut = TRUE; }
```

```
else if (S1[0] < MAXSTRLEN) { // 截断
```

```
    T[1..S1[0]] = S1[1..S1[0]];
```

```
    T[S1[0]+1..MAXSTRLEN] =
```

```
        S2[1..MAXSTRLEN - S1[0]];
```

```
    T[0] = MAXSTRLEN; uncut = FALSE; }
```

```
else { //截断（仅取S1），T[0] == S1[0] == MAXSTRLEN
```

```
    T[0..MAXSTRLEN] = S1[0..MAXSTRLEN];
```

```
    uncut = FALSE; }
```

```
return uncut;
```

```
} // Concat
```

4.2.2 堆分配存储表示

- **特点：**仍以一组地址连续的存储单元存放串值字符序列，但它们的存储空间是在程序执行过程中动态分配而得。所以也称为**动态存储分配的顺序表**。
- 在C语言中，利用**malloc()**和**free()**等动态存储管理函数，来根据实际需要动态分配和释放字符数组空间。称串值共享的存储空间为“**堆**”。这样定义的顺序串类型也有两种形式。

```
typedef char *string;
```

//c中的串库相当于此类型定义，或者再额外加入一个长度变量。

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

4.2.2 堆分配存储表示

- 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 StrInsert(HString &s, int pos, HString t)

```
{ //在字符串s的第pos个字符之前插入串t
  if(pos<1 || pos>s.length+1)    return ERROR; //检查pos位置的合法性
  if(t.length)
  { //t非空，则重新分配空间，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)    //移动s中pos位置之后的字符
      s.ch[i+t.length]=s.ch[i];
    /*插入字符串t并修改字符串s的长度*/
    s.ch[pos-1..pos+t.length-2]=t.ch[0..t.length-1];
    s.length+=t.length;
  }
  return OK;
}
```

内存动态分配函数说明:

- **malloc函数**: 从堆上获得指定字节的内存空间。

函数声明如下: **Void *malloc(int n);**

- 如果函数执行成功, 返回获得内存空间的首地址, 失败则返回**null**。
- 由于**malloc**函数值的类型为**void**型指针, 因此, 可以将其值**类型转换后附给任意类型指针**, 这样就可以通过操作该类型指针来操作从堆上获得的内存空间。
- 需要注意的是, **malloc**函数分配得到的内存空间是**未初始化的**。因此, 一般使用时要调用**memset**来将其初始化为全0。程序结束之前必须调用**free**释放空间。

- **calloc函数**: 与**malloc**函数的功能相似, 都是从堆分配内存, 不同的是**calloc**函数得到的内存空间是**经过初始化的**, 其内容全为0。分配的内存也需要自行释放。

- **realloc函数**的功能比**malloc**和**calloc**的功能更丰富, 可以实现内存的分配和内存释放的功能。

函数声明如下: **void *realloc(void *p,int n);**

- 其中, 指针**p**必须为指向堆内存空间的指针, **realloc**函数将指针**p**指向的内存块的大小改变为**n**字节。
- 如果**n**小于或等于**p**之前指向的空间大小, 那么, 保持原有状态不变。如果**n**大于原来**p**之前指向的空间大小, 那么系统将重新为**p**从堆上分配一块大小为**n**的内存空间, 同时, 将原来指向空间的内容依次复制到新的内存空间上, **p**之前指向的空间被释放。**realloc**函数分配的空间也是未初始化的。

Status StrAssign(HString &t, char *chars)

```
{ //生成一个其值等于串常量chars的串t
    if(t.ch) free(t.ch); //释放字符串t原有的空间
    /*计算字符串chars的长度*/
    for(i=0, c=chars; c ; ++i, ++c);
    if(!i) { //若chars长度为0，则设置字符串t为空字符串
        t.ch=NULL; t.length=0;
    }
    else{ //否则，分配空间并拷贝chars到字符串t中
        if(!(t.ch=(char *)malloc(i*sizeof(char))))
            exit(OVERFLOW);
        t.ch[0..i-1]=chars[0..i-1];
        t.length=i;
    }
    return OK;
}
```




```
int StrLength(HString s)
```

```
{ //求字符串s的长度
```

```
    return s.length;
```

```
}
```

```
int StrCompare(HString s, HString t)
```

```
{ //比较字符串s和t的大小
```

```
    //在s和t的长度范围内逐个比较字符，找到一个不相等字符，  
    返回该字符的比较结果
```

```
    for(i=0; i<s.length && i<t.length; ++i)
```

```
        if(s.ch[i]!=t.ch[i])
```

```
            return(s.ch[i]-t.ch[i]);
```

```
    //若超出长度范围，则返回s和t的长度比较结果
```

```
    return s.length-t.length;
```

```
}
```

Status ClearString(HString &s)

```
{ //清除字符串s  
    if(s.ch){ free(s.ch); s.ch=NULL;}  
    s.length=0;  
    return ok;  
}
```



Status Concat(HString &t, HString s1, HString s2)

{ //将s1和s2连接成的字符串拷贝到t中

//给目标字符串t分配空间

if(!(t.ch)=(char*)malloc(s1.length+s2.length)*sizeof(char)))

exit(OVERFLOW);

/*拷贝s1和s2的内容，并修改字符串的长度*/

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;

}

Status SubString(HString &sub, HString s, int pos, int len)

```
{ //返回串s中的第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; //检查位置pos和长度len的合法性
    if(sub.ch) free(sub.ch); //释放sub的空间
    if( ! len ){ /*若长度len为0，则设置sub为空串*/
        sub.ch=NULL;
        sub.length=0;
    }
    else{ //否则分配空间，拷贝第pos个字符起长度为len的子串，并设置长度
        sub.ch=(char *)malloc(len*sizeof(char));
        sub.ch[0..len-1]=s[pos-1..pos+len-2];
        sub.length=len;
    }
    return OK;
}
```

4.2.3 串的链式存储结构

- 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 node{  
    char data;  
    struct node *next;  
}LString;
```

- 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。

4.2.3 串的链式存储结构

- 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大小的整数倍，因此要用特殊字符来填充最后一个结点，以表示串的终结。
- 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 data[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
```

4.2.3 串的链式存储结构

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。

4.2.3 串的链式存储结构

- 在处理串的联结操作时，要注意处理第一个串尾的无效字符
- 使用块链来表示串，要考虑串值的存储密度
$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$
可能浪费的空间包含头结点、指针域和无效字符。
- 块的大小如果设得太大，则浪费空间。块的大小如果设得太小，则访问效率低下。因此，要针对具体的应用场合，选择合适的大小。例如扇区大小为512个字节。
- 块链的插入和删除，同样要涉及到移动元素，其实也具有顺序串的缺点。所以，块链方式不大实用。
- 块链的操作，与链表的操作类似，不作详细讨论。

第四章 串

4.1 串类型的定义

4.2 串的实现和表示

4.2.1 定长顺序存储表示

4.2.2 堆分配存储表示

4.2.3 串的块链存储表示

4.3 串的模式匹配算法

4.3.1 求子串位置的定位函数

4.3.2 模式匹配的一种改进算法

4.4 串操作应用举例

4.3 串的模式匹配算法

- 子串的定位操作又称为模式匹配（**Pattern Matching**）或串匹配（**String Matching**），其中子串T被称为模式串。
- 此操作的应用非常广泛。很多软件，若有“编辑”菜单项的话，则其中必有“查找”子菜单项。
 - 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。

4.3 串的模式匹配算法

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

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

4.3.1 朴素的模式匹配算法（穷举法）

- **基本思想（同算法4.1）**：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，则继续逐个比较后继字符，否则从主串的下一个字符起重新和模式**T**的字符比较。依次类推，直到找到匹配成功，或匹配失败。

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;

while (i <= S[0] && j <= T[0]) {

if (S[i] == T[j]) { ++i; ++j; } // 继续比较后继字符

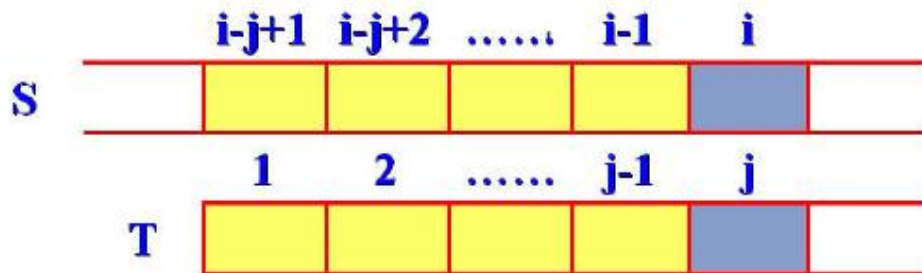
else { i = i-j+2; j = 1; } // 指针后退重新开始匹配

}

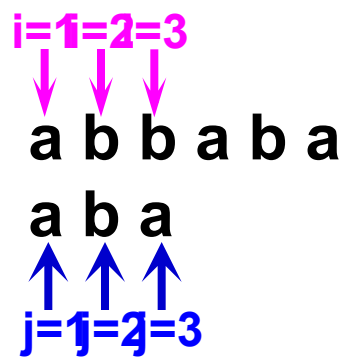
if (j > T[0]) return i-T[0];

else return 0;

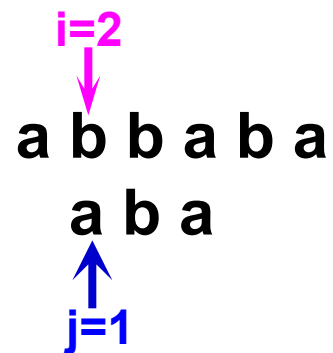
} // Index



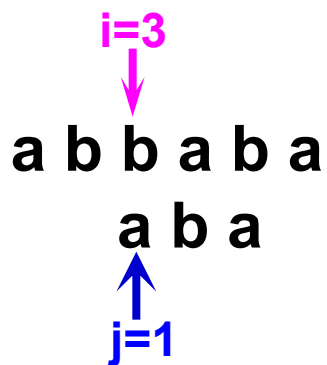
第1趟 S
T



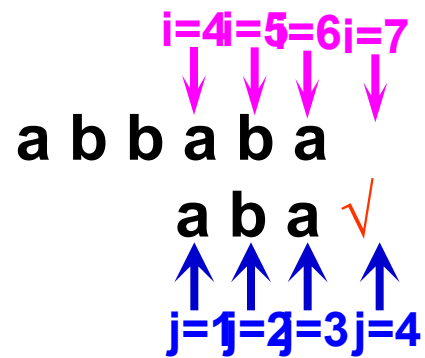
第2趟 S
T



第3趟 S
T



第4趟 S
T



返回 $i=4$

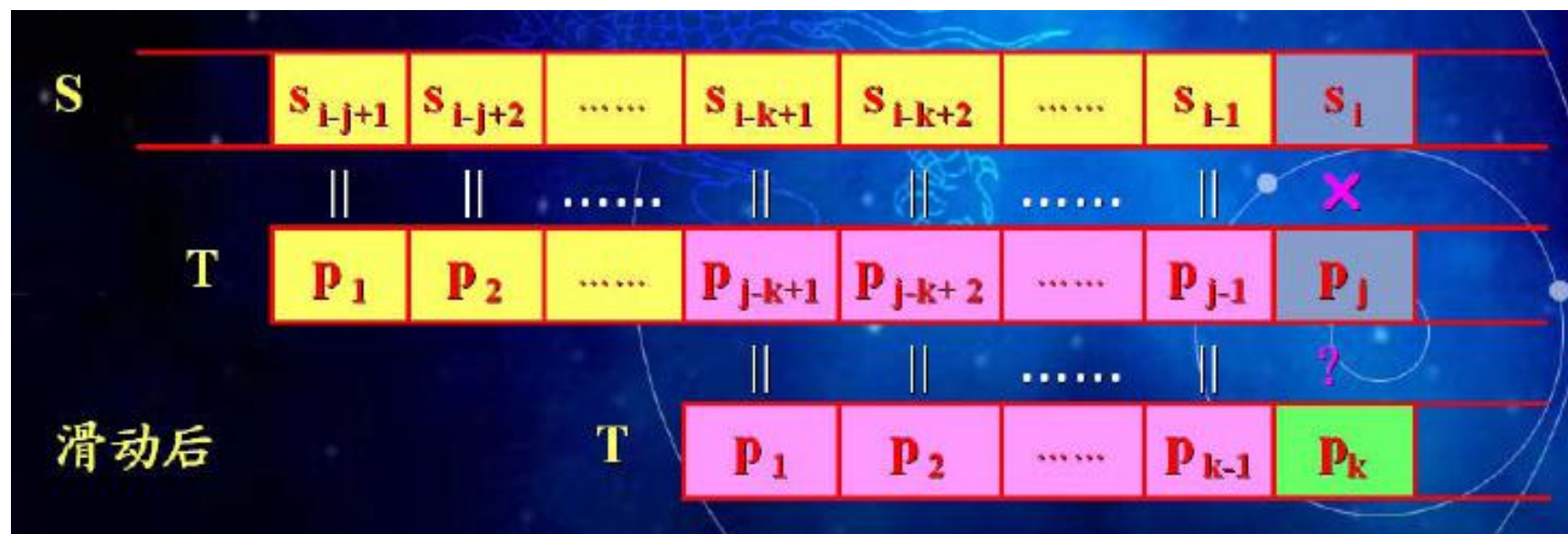
4.3.1 朴素的模式匹配算法（穷举法）

- [illegible]

4.3.2 改进的模式匹配算法——KMP算法

- D.E.Knuth与V.R.Pratt和J.H.Morris同时发现的，故简称为**KMP**算法
- 每当出现失配时，**i指针不回溯**，而是利用已经得到的“部分匹配”结果将模式向右“滑动”**尽可能远的一段距离**后，继续比较。
- 假设主串 ‘ $s_1s_2\cdots s_n$ ’，模式串 ‘ $p_1p_2\cdots p_m$ ’，当主串中第*i*个字符与模式串中第*j*个字符“失配”时，主串中第*i*个字符应与模式串中第*k*个字符再比较。

4.3.2 改进的模式匹配算法——KMP算法



实质： $k-1$ 为 ' $p_1p_2\dots p_{j-1}$ ' 的最大相同真前缀，即模式中头 $k-1$ 个字符的子串 ' $p_1p_2\dots p_{k-1}$ ' 必定与主串中第 i 个字符之前长度为 $k-1$ 的子串 ' $s_{i-k+1}s_{i-k+2}\dots s_{i-1}$ ' 相等，由此，匹配仅需从模式中第 k 个字符与主串中第 i 个字符比较起继续进行。

第1趟 S
T

$i=1=2=3$
a b b a b a
a b a
 $j=1=2=3$

第2趟 S
T

$i=3$
a b b a b a
a b a
 $j=1$

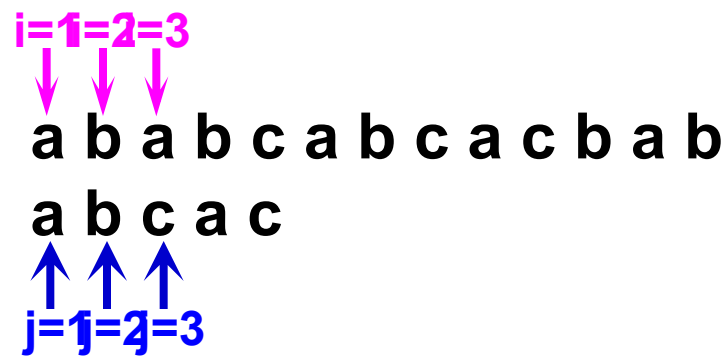
i指针不回溯

第3趟 S
T

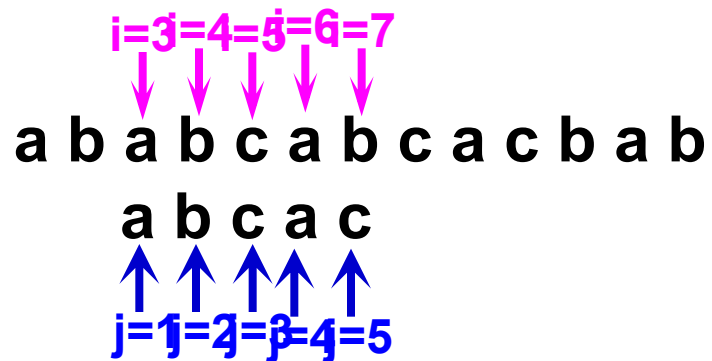
$i=4=5=6=7$
a b b a b a
a b a ✓
 $j=1=2=3=4$

返回 $i=4$

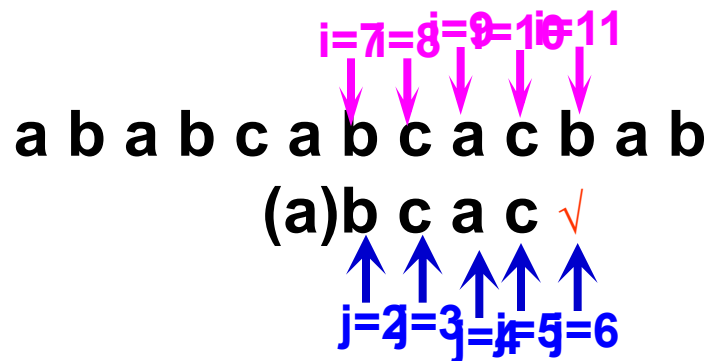
第1趟 S
T



第2趟 S
T



第3趟 S
T



返回 $i=6$

4.3.2 改进的模式匹配算法——KMP算法

- 定义模式串的**next函数**：若令 $next[j]=k$,则 $next[j]$ 表明当模式串中第 j 个字符与主串中第 i 个字符“失配”时，在模式串中需重新和主串中该字符进行比较的字符的位置。
- **next函数**有时也称为**失效函数**，其值仅取决于模式串本身而和想匹配的主串无关——即**与 i 值无关**！

$$next[j] = \begin{cases} 0 & \text{当 } j = 1 \text{ 时} \\ \text{Max} \{k \mid 1 < k < j \text{ 且 } 'p_1 p_2 \cdots p_{k-1}' = 'p_{j-k+1} \cdots p_{j-1}'\} & \text{当此集合不空时} \\ 1 & \text{其它情况} \end{cases}$$

4.3.2 改进的模式匹配算法——KMP算法

【例1】

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

【例2】

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

```
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
```

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： $next[j] = k$;

1) 若： $p_j = p_k$

则： $next[j+1] = k+1$

2) 若： $p_j \neq p_k$ ，且 $next[k]=k'$ ， 则：

a) 若 $p_j = p_{k'}$ ，则： $next[j+1] = next[k]+1$;

b) 若 $p_j \neq p_{k'}$ ，则： 将模式串继续向右滑动直至将模式中的第 $next[k']$ 个字符和 p_j 对齐，.....，以此类推，直至 p_j 和模式串中某个字符匹配成功或者不存在任何 k' ($1 < k' < j$) 满足

$'p_1 \dots p_k' = 'p_{j-k'+1} \dots p_j'$ ， 则 $next[j+1]=1$ 。



这实际上也是一个模式匹配的过程，
不同在于：主串和模式串是同一个

■ 例子：模式串的next函数值

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

已知前6个字符的next函数值，求next[7]？

- 由next[6]=3, 又 $p_6 \neq p_3$, 则需比较 p_6 和 p_1 (因为next[3]=1), 这相当于将子串模式向右滑动。
- 由于 $p_6 \neq p_1$, 而且next[1]=0, 所以next[7]=1;

- 因为 $p_7 = p_1$, 则next[8]=2。

求next函数值的算法:

```
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
```

- 时间复杂度为 $O(n)$
- 通常，模式串的长度 m 比主串的长度要小得多，故对整个匹配算法来说，所增加的这点时间是值得的。

4.3.3 KMP算法 VS. 朴素算法

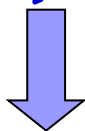
■ 时间复杂度

- 朴素算法的时间复杂度为 $O(n*m)$ ，但一般情况下，其实际执行时间爱你近似于 $O(n+m)$ ，故至今仍被采用。
- **KMP**算法仅当模式与主串之间存在许多“部分匹配”的情况下才显得比朴素算法快得多。

■ 对主串的扫描

- **KMP**算法的最大特点：指示主串的指针不需要回溯，整个匹配过程对主串仅需从头到尾扫描一遍，这对处理从外设输入的庞大文件很有效，可以边读边匹配，无需回头重读。

朴素算法的不足



■ S: a b c a b c d a b c

■ T: a b c a b c a (1)

■ T: a b c a b c a (2)

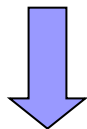
■ T: a b c a b c a (3)

■ T: a b c a b c a (4)

■ 其实, (1) 可以直接到 (4) 。

■ next[7]=4;

改进后的KMP



- S: a b c a b c d a b c
- T: a b c a b c a (1)
- 因为next[7]=4;
- T: a b c a b c a (2)
- 因为next[4]=1;
- T: a b c a b c a (4)
- 因为next[1]=0;这时候比较下一位。i=8,j=1

KMP的不足



- S: a b a b a b d
- T: a b a b a b a
- T: a b a b a b a
- T: a b a b a b a
- T: a b a b a b a
- T: a b a b a b a (下移一位)
- 一步到位：因为所比较的T的当前值都是a，所以
 $\text{next}[7]=\text{next}[5]=\text{next}[3]=\text{next}[1]=0$ 。

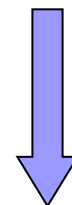
next函数的改进: 例如:

$S = \text{'aaabaaabaaabaaab'}$

$T = \text{'aaaab'}$

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

- 当 $i=4$ 、 $j=4$ 时, $S[4] \neq T[4]$, 由 $\text{next}[j]$ 的指示还需进行 $S[4]$ 和 $T[3]$ 、 $S[4]$ 和 $T[2]$ 、 $S[4]$ 和 $T[1]$ 三次的比较。
- 但是由 $T[1]=T[2]=T[3]=T[4]$, 可知不需要再和主串的 $S[4]$ 比较, 可以将模式一气向右滑动4个字符直接进行 $S[5]$ 和 $T[1]$ 字符的比较。



若 $\text{next}[j]=k$, 而模式中 $p_k=p_j$, 则当主串中字符 s_i 和 p_j 比较不等时, 不需要再和 p_k 进行比较, 而直接和 $p_{\text{next}[k]}$ 进行比较, 即使 $\text{next}[j]$ 和 $\text{next}[k]$ 相同。

next函数的改进算法:

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

第四章 串

4.1 串类型的定义

4.2 串的实现和表示

4.2.1 定长顺序存储表示

4.2.2 堆分配存储表示

4.2.3 串的块链存储表示

4.3 串的模式匹配算法

4.3.1 求子串位置的定位函数

4.3.2 模式匹配的一种改进算法

4.4 串操作应用举例

4.4 串操作应用举例—— 文本编辑

- 用户可利用换页符和换行符把文本划分成若干页，每页有若干行。
- 将文本看成一个字符串，称为文本串，页则是文本串的子串，行又是页的子串。

m	a	i	n	()	✓		[e	h	a	r		r
[N]	,	t	[N]	,	*	s	=	"	"	;
	i	n	t			p	o	s	;	✓			g	e
t	s	(r)	;	g	e	t	s	(t)	;	
		s	c	a	n	f	c	"	%	d	"	,	&	p
o	s)	;	✓			s	t	r	i	n	s	e	r
t	(&	s	,	p	o	s	,	r	,	t)	;	p
u	t	s	(s)	;	✓)	✓					

4.4 串操作应用举例—— 文本编辑


- 为管理文本串的页和行，在进入文本编辑时，编辑程序先为文本串建立相应的页表和行表，即**建立各子串的存储映像**。
 - 页表的每一项给出了页号和该页的起始行号。
 - 行表的每一项则指示了每一行的行号、起始地址和该行子串的长度（含换行符在内）。

列 号	起始地址	长 度
10	1000	7
20	1007	34
30	1041	39
40	1080	33
50	1113	2

- 文本编辑程序中设立页指针、行指针和字符指针分别指示当前操作的页、行和字符。

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。

- 
- 基础知识题自己做
 - 习题4.12,4.17,4.24
 - 4.7, 4.8
 - 实验选做： 3.5 程序分析