数据结构

北京邮电大学 网络空间安全学院

武 斌

上章内容

上一章（栈和队列）内容：

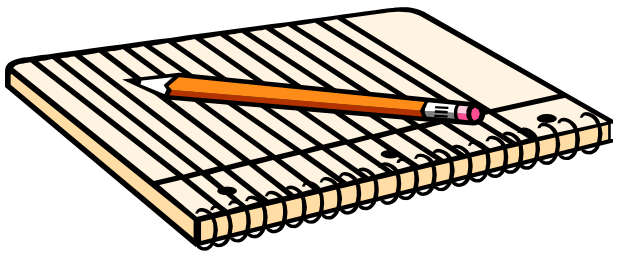
- 掌握栈和队列这两种抽象数据类型的特点
- 熟练掌握栈类型的两种实现方法
- 熟练掌握循环队列和链队列的基本操作实现算法
- 理解递归算法执行过程中栈的状态变化过程

本次课程学习目标

学习完本次课程，您应该能够：

- 理解“串”类型定义中各基本操作的特点
- 能正确利用这些特点进行串的其它操作
- 理解串类型的各种存储表示方法
- 理解串匹配的各种算法

本章课程内容（第四章 串）

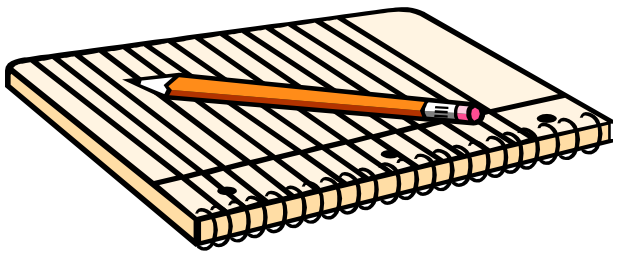


- 4.1 串类型的定义
- 4.2 串的表示和实现
- 4.3 串的模式匹配算法
- 4.4 串操作应用举例

第四章 串

- 串即字符串，是计算机非数值处理的主要对象之一。在早期的程序设计语言中，串仅作为输入和输出的常量出现。
- 随着计算机应用的扩展，需要在程序中进行对“串”的操作，从而使众多编程语言增加了串类型，以便程序员可以在程序中对“串变量”进行操作。
- 现今使用的计算机的硬件结构主要是面向数值计算的需要，基本上没有提供对串进行操作的指令。
- 因此需要用软件来实现串数据类型。而且，在不同的应用中，所处理的串具有不同的特点，为有效地实现串的操作，需要根据具体情况使用合适的存储结构。

串的类型定义



- 4.1 串的类型定义
- 4.2 串的表示和实现
- 4.3 串的模式匹配算法
- 4.4 串操作应用举例

串的类型定义

- 串(String)是零个或多个字符组成的有限序列。一般记作S="$a_1a_2a_3\ldots a_n$"，其中S是串名，双引号括起来的字符序列是串值；a_i (1≦i≦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格串(Blank String)。

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

串的类型定义

- 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。

串的类型定义

- 通常在程序中使用的串可分为两种：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。
- 串值必须用一对双引号括起来，但双引号本身不属于串，它的作用只是为了避免与变量或数的常量混淆而已。
- 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。

串的类型定义

- 串的类型定义如下：

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。

DestroyString (&S)

初始条件：串 S 存在。

操作结果：串 S 被销毁。

回顾线性表的数据对象和数据关系的定义可见，作为数据结构，串和线性表的差别仅在于串中的数据元素限定为"字符"。

串的类型定义

StrCopy (&T, S)

初始条件：串 S 存在。

操作结果：由串 S 复制得串 T。

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。

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。

串的类型定义

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。

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。

串的类型定义

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

串的基本操作

- “串值大小”是按“词典次序”进行比较的，如：
  - **StrCompare("data","Stru")<0**
  - **StrCompare("cat"," case")>0**
- 显然，只有在两个串的长度相等且每个字符一一对等的情况下称两个串相等。
- “串联接”操作的结果是生成一个新的串，其值是“将第二的串的第一个字符紧接在第一个串的最后一个字符之后”得到的字符序列。如操作：
  - **Concat(T,“man”,“kind”)**得到的结果是：
  - **T =“mankind”**。

串的基本操作

- “子串”指串中任意个连续的字符组成的子序列。如操作：
  - **SubString(Sub,“commander”,4,3)**得到的结果是**Sub=“man”**。
- 显然必须在满足初始条件中规定的“起始位置”和“长度”之间的约束关系时才能求得一个合法的子串。允许 len 的下限为0是由于空串也是合法串，但实际上求长度为0的子串是没有意义的。
  - “子串在主串中的位置”指的是子串中的第1个字符在主串中的位序。**Index(S, T, pos)操作类似于线性表的Locate操作**，在S中查询和T值相同的子串，pos为查询的起始位置。如：
  - 子串“man”在主串“commander”中的位置为4。
  - **Index(“This is a pen”,“is”,pos)**，若pos=1，则查询得结果为3；若 pos=4，则得结果为6，若pos=6，则得查询结果为0。

串的基本操作

- **Replace (&S, T, V)**

假设S=“abcacabcaca”， T=“abca”和V=“x”， 则置换之后的S=“xcxca”。注意定义中“不重叠”三个字，若上例中的V=“ab”时，则置换后的结果应该是：

➔**S=“ abcabca”**，而不是**“abbca”**。

- **StrInsert (&S, pos, T)**

例如：S=“chater”，T=“rac”，pos=4，则插入后的结果为：

➔**S="character"**。

串的基本操作

- **思考**：串的基本操作和线性表一样，无非也就是查找、插入和删除等，那么它们能否用线性表的操作来替代呢？
  - ➔ **回答**：串的基本操作和线性表**有很大的区别**，同样是查找、插入和删除，但对**线性表**而言操作对象是“**单个数据元素**”，如在线性表中查找某一个特定的数据元素，或者插入/删除一个数据元素。

    而对**串**而言，是**以串的整体作为操作对象**，如将两个串联接在一起，在串中查找一个子串，插入/删除一个串等等。由此可以体会到**串类型不能和线性表类型混为一谈**。

串的基本操作

- 补充：对于串的基本操作，许多高级语言均提供了相应的运算或标准库函数来实现。下面介绍几种**在C语言中常用的串运算**。

  ➔ 定义下列几个变量：

  char s1[20]=“dirtreeformat”,s2[20]=“file.mem”;

  char s3[30],*p;

  int result;

  ➢ 求**串长(length)**

  int strlen(char s); //求串的长度

  例如：printf(“%d”,strlen(s1)); // 输出13

串的基本操作

- **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”

- **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”

串的基本操作

- 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

- 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，否则返回NULL。

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

if(p) strcpy(p,”.cpp”);　// s2=“file.cpp”

串的基本操作

- 实现Index(S,T,pos)算法的基本思想为：从主串**S**中取“第 **i** 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**直至找到和串**T**相等的子串或者串**S**中不存在和**T**相等的子串为止。

  - 即求使下列等式

    StrCompare(SubString(S,i,StrLength(T)),T)**==0**

    成立的 i 值。**i** 的初值应为 **pos**，在找不到的情况下，**i** 的终值应该是 **n-m+1**，其中，**n** 为**S**串的长度，**m** 为**T**串的长度。

  - 演示 4-1-1

串的基本操作

- 算法4.1

```
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); // 求得串长
        i = pos;
        while ( i <= n-m+1) {
            SubString (sub, S, i, m); // 取得从第 i 个字符起长度为 m 的子串
            if (StrCompare(sub,T) != 0) ++i ;
            else return i ;              // 找到和 T 相等的子串
        } // while
    } // if
    return 0;                    // S 中不存在满足条件的子串
} // Index
```

串的基本操作

- 实现“置换”Replace (&S, T, V)操作的基本思想为：由S和V生成一个新的串 news。首先将它初始化为一个空串，然后重复下列两步直至“查找不成功”为止：

➔1) 自起始位置起，在串S中查找和T相同的子串；

➔2) 将S中不被置换的子串（即从 pos 起到和T相同子串在S中的位置之前的字符序列）和V相继连接到news 串上。

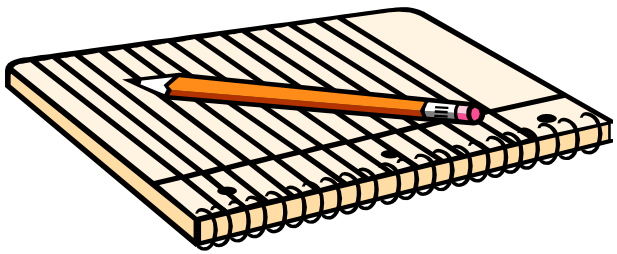
➔演示 4-1-2

串的基本操作

● 算法4.2 // 以串V替代串S中出现的所有和串T相同的子串

```
void replace(String& S, String T, String V)
{   n=StrLength(S); m=StrLength(T); pos = 1;
    StrAssign(news, NullStr);              // 初始化 news 串为空串
    i=1;
    while ( pos <= n-m+1 && i )
    {   i=Index(S, T, pos);                // 从pos指示位置起查找串T
        if (i!=0) {
            SubString(sub, S, pos, i-pos);   // 不置换子串
            Concat(news, news, sub);         // 联接S串中不被置换部分
            Concat(news,news, V);            // 联接V串
            pos = i+m;                       // pos 移至继续查询的起始位置
        } // if
    } // while
    SubString(sub, S, pos, n-pos+1);   // 剩余串
    Concat( S, news, sub );            // 联接剩余子串并将新的串赋给S
}
```

串的表示和实现



- 4.1 串的类型定义
- 4.2 串的表示和实现
- 4.3 串的模式匹配算法
- 4.4 串操作应用举例

串的表示和实现

- 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：
  - ➔ **一、串的定长顺序存储表示**

    定长顺序存储表示,也称为**静态存储分配的顺序表**。它是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是**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 256
typedef  char  sstring[maxstrlen];
sstring  s;   //s是一个可容纳255个字符的顺序串，0号单元放串的长度。
```

串的表示和实现

- 一般可使用一个不会出现在串中的特殊字符在串值的尾部来表示串的结束。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，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。

- 在这种表示方法下，实现串操作的基本操作是"字符序列的复制"。如下列算法4.3和算法4.4分别为串的联接和求子串。

串的表示和实现

- 算法 4.3　// 以T返回由S1和S2联接而成的新串

```
Status Concat(SString &T,SString S1,SString S2)
{    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{
         T[0.. MAXSTRLEN] = S1[0.. MAXSTRLEN];
         uncut  = FALSE; }
     Return uncut;
} // Concat_Sq
```

串的表示和实现

❊**注意**：这种表示方法中，存储串值的数组空间都是**静态分配**的，此算法中串T的存储空间必须在调用此函数的程序中予以分配，即在该程序中**必须为T分配足够的空间**，否则将会出现操作不成功 。

串的表示和实现

●算法 4.4

```
Status SubString (SString &Sub,SString S,int pos,int len)
{   // 若参数合法(即1≤pos≤StrLength(S) 且
    // 0≤len≤StrLength(S)-pos+1),则以Sub带回串S中第pos个
    // 字符起长度为len的子串，并返回TRUE，否则返回FALSE
    if (pos < 1 || pos > S[0] || len < 0 || len > S[0] -pos+1)
        return FALSE;
    Sub[1..len] = S[pos..pos+len-1];
    Sub[0] = len;
    return TRUE;
} // SubString
```

串的表示和实现

- 从上面这两个例子可见，用字符数组表示字符串时，串的操作容易进行，实现串操作的**原操作为“字符序列的复制”**，操作的**时间复杂度**基于复制的字符序列的长度。
- 但由于在此用的是**C语言中的静态数组，串值的存储空间必须进行预分配，致使它的应用受到一定限制**。
- 如果在操作中出现串值序列的长度超过预分配的上界MAXSTRLEN时，约定用截尾法处理。
- 克服此弊病的方法:不限定串长的最大长度，动态分配串值的存储空间。

串的表示和实现

二、串的堆分配存储表示

这种存储表示的特点是，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。堆分配存储结构的串定义如下：

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

串的表示和实现

- 由于串仍然是以数组存储的字符序列表示，因此串的操作仍基于“字符序列的复制”实现。在此仅举“插入”操作一例，其算法如算法4.5所示。
  - 分析：插入操作的基本思想是构造一个新的串。因此首先要为被插入串S重新分配存储空间，然后将S串中插入位置之前的子串、T串以及S串中插入位置之后的子串依次复制到这个新分配的存储空间中去。

串的表示和实现

- 算法4.5

```
Status StrInsert (Hstring& S, int pos, Hstring T)
{   // 若1≤pos≤StrLength(S)＋1，则改变串S，在串S的第pos个
    // 字符之前插入串T，并返回TRUE,否则串S不变，并返回FALSE
    if (pos < 1 || pos > S.length+1)  return FALSE;     // 插入位置不合法
    if (T.length){      // T 非空，则为S重新分配空间并插入 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];
        S.ch[pos-1..pos+T.length-2] = T.ch[0..T.length-1];
        S.Length += T.length;
    }
    return OK;
}
```

串的表示和实现

➔ 三、串的块链存储表示

- 采用链表方式存储串值。
- 结点大小：每个结点可以存放一个字符，也可以存放多个字符。
- 由于在一般情况下，串的操作都是从前往后进行的，因此串的链表通常不设双链，也不设头结点，但为了便于进行诸如串的联接等操作，链表中还附设有尾指针，并且由于串的长度不一定是结点大小的整数倍（链表中最后一个结点中的字符非都是有效字符），因此还需要一个指示串长的域。

串的表示和实现

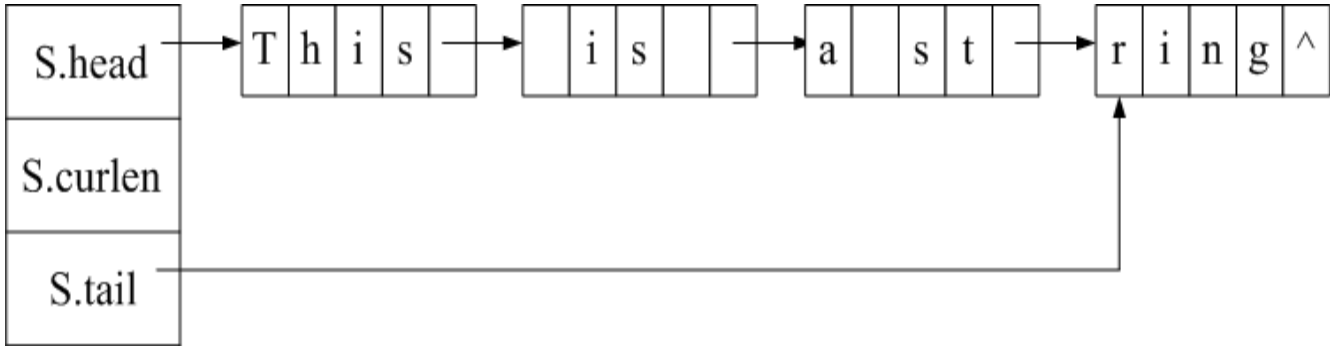
称如此定义的存储结构为串的块链存储结构，其定义如下：

```
const CHUNKSIZE = 80;          // 可由用户定义的块（结点）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;        // 串的头指针和尾指针
    int curlen;                // 串的当前长度
} LString;
```

串的表示和实现

- 由于串的结构的特殊性，即串的数据元素是一个字符，它只有8位二进制数，因此用链表存储串值时通常采用的办法是在一个结点中存放多个字符（如下图所示），图中的 “结点大小” CHUNKSIZE=4。



- 在以链表存储串值时，定义串的存储密度为

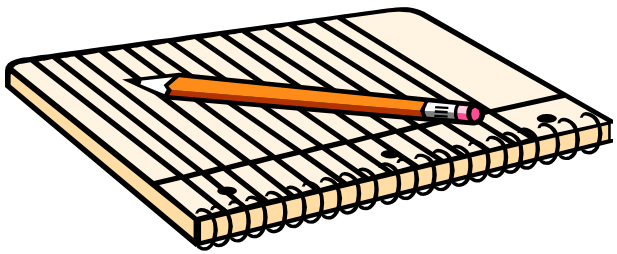
$$\text{存储密度} = \frac{\text{串值所占的存储位}}{\text{实际分配的存储位}}$$

  - 假设指针域 next 占16"位(bit)"，则上图中串的存储密度为2/3。

串的表示和实现

- 显然，**以块链作存储结构时实现串的操作很不方便**，如在串中插入一个子串时可能需要分割结点，联接两个串时，若第一个串的最后一个结点没有填满时还需要添加其它字符等等。
- 但在应用程序中，**可将串的链表存储结构和串的定长结构结合使用**。例如在正文编辑系统中，整个“正文”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），而行和行之间用指针相链。

串的模式匹配算法



- 4.1 串的类型定义
- 4.2 串的表示和实现
- 4.3 串的模式匹配算法
- 4.4 串操作应用举例

串的模式匹配算法

- 在计算机所处理的各类数据中，有很大一类属于正文数据，也常称为文本型数据，如各种文稿资料、源程序、上网浏览页面的HTML文件等。几乎在所有对正文进行编辑的软件中，都提供有“查找”的功能，即要求在正文串中，查询有没有和一个“给定的串”相同的子串，若存在，则屏幕上的光标移动到这个子串的起始位置。这个操作即为串的定位操作，通常称为正文模式匹配。例如：
  - 若 S =“concatenation”，T =“cat”
  - 则称主串S中存在和模式串T相同的子串，起始位置为4，即Index(S,T,1)=4 。

串的模式匹配算法

●1、串的模式匹配的简单算法

➔ 分析：此算法的思想是直截了当的，正如算法4.1中所描述的那样，将主串S中某个位置i起始的子串和模式串T相比较。即从 j=0 起比较 S[i+j] 与 T[j]，若相等，则在主串 S 中存在以 i 为起始位置匹配成功的可能性，继续往后探索(j逐步增 1)，直至T串中最后一个字符比较相等为止，否则改从主串的下一个字符起重新开始进行下一轮的“匹配”，即将串T向后滑动一位，即 i 增1，而 j 退回至0，重新开始新一轮的匹配。

➔ 演示 4-3-1

串的模式匹配算法

● 算法4.6

```
int Index (SString S,SString T,int pos)
{    // 若串 S 中从第pos(1≤pos≤StrLength(S))个字符起存在和串 T 相同的子串，
     // 则称匹配成功，返回第一个这样的子串在串 S 中的位置，否则返回 0
     i = pos; j = 1;
     while ( i<=s[0]&&j<=T[0]) {
          if ( S[i] == T[j] )
           { ++i;++j}              // 继续比较后一字符
          else { i=i-j+2; j = 1; }                  // 重新开始新的一轮匹配
     }
     if(j>T[0]) return i-T[0];//匹配成功
     else return 0;    // 串S中(第pos个字符起)不存在和串T相同的子串
} // Index
```

● 此算法的**时间复杂度**为***O*(m×n)**，其中 m 和 n 分别为S串和T串的长度。

串的模式匹配算法

❊**注意**：匹配成功时，函数返回的值是"子串在主串中的位置"，即**子串中第1个字符在主串字符序列中的 "位置"**，而**非该字符在存储结构数组中的下标**。

- 一般情况下，算法4.6的**实际执行效率**与字符 T[0] 在串S中**是否频繁出现密切相关**。对于一般文稿中串的匹配，算法4.6的**时间复杂度可降为*O*(m+n)**，因此在多数的实际应用场合下被应用。

串的模式匹配算法

- 为了便于说明问题，先将算法4.6改写为算法4.7的形式，并观看一个实例的匹配过程。
- 算法4.7

```
int Index_BF1 ( char S [ ], char T [ ], int pos )
{ // 若串 S 中从第pos(1≤pos≤StrLength(S))个字符起存在和串 T 相同的子串
  // 则称匹配成功，返回第一个这样的子串在串 S 中的位置，否则返回 0
    i = pos-1; j = 0;
    while ( S[i] != '\0'&& T[j] != '\0') {
        if ( S[i] == T[j] )
            { i++; j ++; }                          // 继续比较后一字符
        else { i = i-j+1; j = 0; }                  // 重新开始新的一轮匹配
    }
    if ( T[j] == '\0') return (i-j);                // 匹配成功
    else return 0;          // 串S中(第pos个字符起)不存在和串T相同的子串
} // Index_BF1
```

串的模式匹配算法

- **例如**：按此算法进行模式串

  T = “abcac” 和主串S =“ababcabcabcacabca” 在 pos=0 的情况下匹配的过程如演示一所示。 4-3-2

- 在此，你可以对比一下另一种匹配过程，即这两个串的匹配也可以这样进行，如演示二所示。 4-3-3

  - **思考**：你有没有看出这两个匹配过程中，最明显的差别是什么？
  - **回答**：在第2个匹配过程中，指针 i 没有“回溯”！

串的模式匹配算法

- 从上页同样一个实例的两个匹配过程可见，简单算法之所以时间复杂度会在最坏情况下达到“二次级” 是因为，当进行“两个子串的比较”过程中，不管是 T 串的第几个字符发生“不等”的情况，指针 i 都必须 “回溯” 到S串中原来开始进行比较的字符的下一个位置。
- 而在第二个匹配过程中，当发生两个串中字符比较不等时指针 i 不需要回溯，而只要将 T 串向右滑动到一定位置继续进行字符间的比较。如上例中，S串中除了第3、7和10个字符和T串中的字符比较了两次之外，其它字符和T串中的字符均只进行了一次比较。

串的模式匹配算法

- 前面第二个动画演示的匹配过程正是本节要讨论的模式匹配的改进算法，它是由三位计算机学者 D.E.Knuth 与 V.R.Pratt 和 J.H.Morris 同时发现的，因此人们通常简称它为 **KMP 算法**。可以证明它的时间复杂度为***O*(m+n)**，直观地看，是因为在匹配过程中指针 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

- 回顾前面所举实例，假设主串为$S_0\,S_1\,S_2\,S_3\,S_4\,S_5\,S_6$，模式串为$t_0\,t_1\,t_2\,t_3\,t_4$，当 $S_2 \neq t_2$ 时，按简单算法接着应进行 S_1 和 t_0 的比较，由于我们在这之前的匹配过程中已经得到了“ $S_1 = t_1$ ”的信息，而从模式串本身又得到了 “ $t_1 \neq t_0$ ” 的信息，由此可得结论：$S_1 \neq t_0$ ，也就没有必要进行 S_1和 t_0 的比较，而直接进行 S_2 和 t_0 的比较。
- 同理当 $S_6 \neq t_4$ 时，只要直接进行 S_6 和 t_1 的比较即可。换句话说，在进行了 S_6 和 t_4 的比较并且知道两者不等之后，可将T串直接向右滑动到和 S_5 对齐的起始位置进行下一轮的匹配，这是由于**T串本身的特性和之前的匹配过程决定**的。

串的模式匹配算法

- 从失配的匹配过程得到 “ $t_0\ t_1 \ldots\ t_{j-k}\ t_{j-k+1} \ldots\ t_{j-1}$ ”=“ $S_{i-j}\ S_{i-j+1} \ldots\ S_{i-k}\ S_{i-k+1} \ldots\ S_{i-1}$ ” 和 ‘ t_j ’ ≠ ‘ S_i ’，
- 从当前匹配过程得到结果 “ $t_0\ t_1 \ldots\ t_{k-1}$ ”=“ $S_{i-k}\ S_{i-k+1} \ldots\ S_{i-1}$ ” 并且不可能存在 k’>k 满足此式，那么下一步只需要继续进行 ‘ t_k ’ 和 ‘ S_i ’ 的比较即可。
- 一般情况下，若 “ $t_0\ t_1 \ldots\ t_{k-1}$ ”=“ $t_{j-k}\ t_{j-k+1} \ldots\ t_{j-1}$ ”，并且不可能存在 k‘>k 满足此式，称 “k”为模式串中字符 ‘ t_j ’ 的 next 函数值。
- 定义：

$$next[j]=\begin{cases} -1 & \text{当j=0时} \\ Max\{k \mid 0<k<j \text{且} "t_0t_1\cdots t_{k-1}" = "t_{j-k}t_{j-k+1}\cdots t_{j-1}"\} & \\ 0 & \text{其它情况} \end{cases}$$

- next 函数值的含义是：当出现$S_i \neq t_j$ 时，下一次的比较应该在 S_i 和 $t_{next[j]}$ 之间进行。

串的模式匹配算法

- 例如，下列所示为模式串的 next 函数值的两个例子。

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

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

串的模式匹配算法

● 算法 4.8

```
int Index_KMP(char S[], char T[], int pos)
{
    // 利用模式串T的next函数求T在主串S中第pos个字符之后第一次
    // 出现的位置的KMP算法。其中1≤pos≤StrLength(S)
    i = pos-1; j = 0;
    while ( S[i] != '\0' && T[j] != '\0' ) {
        if (j = = -1 || 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 ( T[j] == '\0' ) return (i-j);    // 匹配成功
        else return 0;
} // Index_KMP
```

串的模式匹配算法

- 仔细比较算法4.8和4.7，你一定会发现这两个算法只有两个地方不同，一是当 **S[i] ≠T[j] 时**，在算法4.7中指针 i 回溯到 i-j+1 的位置，指针 j 重又指向第 1 个字符，而在算法4.8中，指针i不变，而指针 j=next[j]；二是 **if 语句的条件中多了一种情况(j == -1)**，那么这是一种什么情况呢？
  - 从 next 函数值的定义看，next[0] = -1，因此while 循环中出现 j = -1的情况，只能是在 S[i] ≠ T[0] 之后，即T串中的第一个字符和S串中某个字符比较不等，显然，此时应将T串向右滑动。

串的模式匹配算法

- 下一个问题是如何求模式串的 next 函数值？
  - 求 next 函数的过程是一个递推的过程：
    - 1.首先由定义得 next[0]=-1，next[1]=0;
    - 2.假设已知 next[j]=k，又 T[j] = T[k]，则显然有 next[j+1]=k+1；
    - 3.如果 T[j] !=T[k]，则令 k=next[k]，直至 T[j] 等于 T[k] 为止。
  - 由此可得下页求 next 函数值的**算法4.9**，它和上述的KMP**算法4.8**非常相似。

串的模式匹配算法

- 算法4.9

```
void get_next(char T[], int next[])
{
    // 求模式串T的next函数值并存入数组 next。
    j = 0; next[0] = -1; k = -1;
    while ( T[j+1] != '\0' ) {
        if (k = = -1 || T[j] = = T[k])
        { ++j; ++k; next[j] = k; }
        else k = next[k];
    }// while
} // get_next
```

串的模式匹配算法

- 但是还有一种特殊情况需要考虑：
  - 例如：S = ‘aaabaaabaaabaaabaaab‘，T = 'aaaab‘
    - 此时因为T串的next函数值依次为-1,0,1,2,3,虽然匹配过程中指针 i 没有回溯，但对 i=3、7、11和15重复进行了多次的‘b’是否等于‘a’的操作。
    - 换句话说，如果 next[j]=k，那么此时应该看一下T[j]是否等于 T[k]，如果不等，则 next[j]=k，否则next[j]就应该取值 next[k]。例如上述T串的 next 函数值依次为-1,-1,-1,-1,3。由此改写求模式串的 next 函数值的算法如**算法4.10**。
    - 算法4.10的执行过程及由此进行的KMP算法匹配的过程见动画演示。
      - 演示一 4-3-4-1
      - 演示二 4-3-4-2

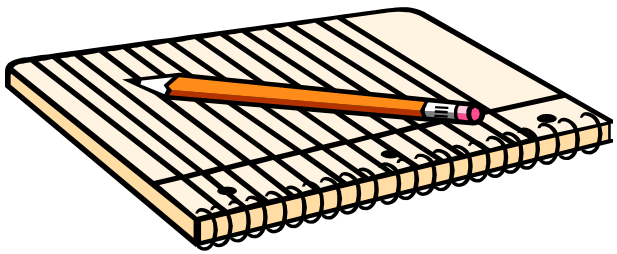
串的模式匹配算法

- 算法 4.10

```
void get_nextval(char T[], int next[])
{// 求模式串T的next函数值并存入数组 next
    j = 0; next[0] = -1; k = -1;
    while ( T[j+1] != '\0' ) {
      if (k = = -1 || T[j] = = T[k]) {
        ++j; ++k;
        if (T[j]!=T[k]) next[j] = k;
        else next[j] = next[k];
        }   // if
      else k = next[k];
    }   // while
}   // get_nextval
```

- algo4-1.cpp

串操作应用举例



- 4.1 串的类型定义
- 4.2 串的表示和实现
- 4.3 串的模式匹配算法
- 4.4 串操作应用举例

串操作应用举例

- 文本编辑程序是一个面向用户的系统服务程序，广泛用于源程序的输入和修改，甚至用于报刊和书籍的编辑排版以及办公室的公文书信的起草和润色等。**文本编辑的实质是修改字符数据的形式或格式**。无论是 Microsoft word 还是 WPS，其工作的基础原理都是正文编辑。虽然各种正文编辑程序的功能强弱不同，但其基本功能大致相同，一般都包括串的查找、插入、删除和修改等基本操作。
- 为了编辑方便起见，用户可以通过换页符和换行符将文本划分为若干页和若干行(或直接划分为若干行)。在编辑程序中，则可**将整个文本看成是一个"正文串"，"页"是正文串的子串，而行则是页的子串**。

串操作应用举例

- 假设有下列一段C的源程序

```
main() {
    float a,b,max;
    scanf("%f,%f",&a,&b);
    if a>b  max=a;
    else max=b;
};
```

- 我们将此源程序看成是一个正文串，输入内存后如图所示，图中"↙"为换行符。

串操作应用举例

201

m	a	i	n	(	)	{	↙			f	l	o	a	t		a	,	b	,
m	a	x	;	↙			s	c	a	n	f	(	“	%	f	,	%	f	“
,	&	a	,	&	b	)	;	↙			i	f		a	>	b			m
a	x	=	a	;	↙			e	l	s	e			m	a	x	=	b	;
↙	}	;	↙																

- 为了管理文本串中的页和行，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，页表的每一项列出页号和该页的起始行号，行表的每一项则指示每一行的行号、起始地址和该行子串的长度。假设此例文本串只占一页，起始行号为100，则该文本串的行表如下页表1所示。

串操作应用举例

表1

行号	起始地址	长度
100	201	8
101	209	17
102	226	24
103	250	17
104	267	15
105	282	3

- 在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。如果在某行内插入或删除若干字符，则要修改行表中该行的长度，若该行长度因插入而超出了原分配给它的存储空间，则要为该行重新分配存储空间，并修改该行的起始位置。

串操作应用举例

- 例如，对上述源程序进行编辑后，其中的103行修改成

if (a>b) max=a;

105行修改成

}

- 修改后的行表如右表所示。

行号	起始地址	长度
100	201	8
101	209	17
102	226	24
103	**285**	**19**
104	266	15
105	**282**	**2**

串操作应用举例

- 当插入或删除一行时，必须同时对行表也进行插入和删除，若被删除的“行”是所在页的起始行，则还要修改页表中相应页的起始行号（应修改成下一行的行号）。

- 为了查找方便，行表是按行号递增的顺序安排的，因此对行表进行插入或删除时需移动操作之后的全部表项。页表的维护与行表类似，在此不再赘述。由于对文本的访问是以页表和行表作为索引的，因此在删除一页或一行时，可以只对页表或行表作相应修改，不必删除所涉及的字符，这可以节省不少时间。

本章小结

- 在这一章向读者介绍了串类型的定义及其实现方法，并重点讨论了串操作中最常用的“串定位操作（又称模式匹配）”的两个算法。
- 串的两个显著特点：
  - 其一、它的数据元素都是字符，因此它的存储结构和线性表有很大不同，例如多数情况下，实现串类型采用的是“堆分配”的存储结构，而当用链表存储串值时，结点中数据域的类型不是“字符”，而是“串”，这种块链结构通常只在应用程序中使用；
  - 其二、串的基本操作通常以“串的整体”作为操作对象，而不像线性表是以“数据元素”作为操作对象。

本章小结

- **“串匹配”的简单算法**（**算法4.6**）其算法思想直截了当，简单易懂，适用于在一般的文档编辑中应用，但在某些特殊情况，例如只有0和1两种字符构成的文本串中应用时效率就很低。

- **KMP算法**是它的一种改进方法，其**特点**是利用匹配过程中已经得到的主串和模式串对应字符之间"等与不等"的信息以及T串本身具有的特性来决定之后进行的匹配过程，从而减少了简单算法中进行的"本不必要再进行的"字符比较。

本章知识点与重点

●知识点

串的类型定义、串的存储表示、串匹配、KMP算法

●重点和难点

相对于其它各个知识点而言，本章非整个课程的重点，鉴于串已是多数高级语言中已经实现的数据类型，因此本章重点仅在于了解串类型定义中各基本操作的定义以及串的实现方法，并学会利用这些基本操作来实现串的其它操作。

本章的难点是理解实现串匹配的KMP算法的思想，希望有能力的同学能够基本掌握该算法。