CHAPTER 3

第3章

串与文本编辑

计算机已经被大量用来处理非数值计算问题，如文本编辑、自然语言翻译等。这些问题中所涉及的处理对象多数是字符串数据。字符串(String)一般简称串，是数据元素为字符的一种线性表。对线性表的操作通常是以线性表的数据元素作为操作的基本单位，与线性表的操作特点不同，串的元素是特定的字符型，用途广泛，常常被作为一种独立的类型来研究。

本章主要讨论串的基本概念、抽象数据类型和两种存储结构，并给出了串的一种实际应用——文本编辑。

3.1 串的类型定义

1. 串的相关术语

串是由零个或多个字符组成的有限序列，记为：$s="s_1 s_2 \cdots s_n"$。其中 s 是串名；双引号内的字符序列 $s_1 s_2 \cdots s_n$ 是串值；$n(n \geqslant 0)$表示串的长度。

例如：

```
s1 = "data structure"                    //串,长度为 14
```

串长度为零的串称为空串。

例如：

```
s = ""                                   //空串,长度为 0
```

组成串的字符均为空格的串称为空格串或空白串。

例如：

```
s = "    "                               //空格串,长度为 4
```

一个串中任意连续的字符组成的子序列称为该串的子串。空串是任何串的子串。

例如：

```
s1 = "data structure"
s2 = "data"                              //s2 是 s1 的子串
```

```
s3 = "structure"                    //s3 是 s1 的子串
s4 = "datastructure"                //s4 不是 s1 的子串
```

包含子串的串称为主串。上例中 s1 为主串。

串中每一个字符的位置都用一个自然数表示,称为字符在串中的序号(index)。串的第一个字符的序号为 1。子串的序号是该子串的第一个字符在主串中的序号。在上例中,子串 s2 在 s1 中的序号为 1,s3 在 s1 中的序号为 6。s4 不是 s1 的子串,也可以说,s4 在 s1 中的序号为 0。

当且仅当串的长度相等并且对应位置上的字符都相同时,称这两个字符串是相等的。

例如:

```
s1 = "data structure"
s2 = "data structure"
s3 = "datastructure"                //s1 与 s2 相等,s3 与 s1 和 s2 均不相等
```

按照串中字符的次序,逐一比较两个字符串中字符的大小,以确定两个串的大小关系的操作,称为串的比较。

例如:s5="data",s6="DATA",则有 s5>s6 的比较结果为 1,s5<s6 的比较结果为 0。

2. 串的 ADT 定义

串是一种特殊的线性表,可以看成是运算对象受限的线性表,其数据及数据之间的关系与线性表的描述相同。但串的基本操作和线性表有很大差别,在线性表中,大多以单个元素作为操作对象,而在串的基本操作中,通常以串的整体作为操作对象。

在引入串的 ADT 定义前先来看一个字符串应用的例子。

【例 3-1】 有一个字符串"live on no evil",检查它是否为"回文"。当一个字符串顺读和逆读都一样,就称这个字符串是回文。

英文中的回文具有广义和狭义之分,广义的回文是指串中的空格字符不计入内,比如串"ten animals I slam in a net"去掉空格字符后是一个回文。狭义的回文是指将空格字符计入在内,比如题目中的"live on no evil"不过滤掉空格就是回文字符串。单个英文单词的回文符合狭义回文。例如:eye,mum,refer,level 等。

判断一个字符串 s 是否为回文(狭义的),需要进行如下操作。

(1) 存储串 s,并以相反顺序存储为串 t。

(2) 比较 s 与 t。

(3) 得出字符串 s 是否为回文串的判断。

(4) 输出回文串 s。

例 3-1 是一个串的实际应用问题,为解决问题所需要的有关串的操作,即串类型应该提供的应用接口都是以串为单位,而不是串中的单个字符为单位。下面给出串的 ADT 定义:

```
ADT String
{
Data:
    D = {a_i | a_i ∈ ElemSet, i = 1,2,…,n,n >= 0}
Structure:
    S = {< a_i-1,a_i > | a_i-1,a_i ∈ D, i = 2,3,…,n}
```

```
Operations:
    ConstructString()
    //操作结果: 创建一个空的串 s
    DestructString()
    //操作条件: 已有串 s
    //操作结果: 销毁当前串 s
    StringLen()
    //操作条件: 已有串 s
    //操作结果: 得到当前串 s 的实际长度
    StringCpy(t)
    //操作条件: 已有串 s 和参数串 t
    //操作结果: 将 t 复制到当前串 s 中
    OutputString()
    //操作条件: 已有串 s
    //操作结果: 输出当前串 s
    SubString(pos, len, &t)
    //操作条件: 已有串 s
    //操作结果: 取串 s 中从第 pos 个字符开始的,长度为 len 的子串,由 t 返回
    DelSubString(pos, len, &t)
    //操作条件: 已有串 s 和一个参数串 t
    //操作结果: 删除当前串从第 pos 个字符开始,长度为 len 的子串
                并由 t 返回被删除的子串
    InsertSubString(pos, t)
    //操作条件: 已有串 s 和参数串 t
    //操作结果: 将 t 插入到当前串第 pos 个位置前
    ConnectString (t)
    //操作条件: 已有串 s 和参数串 t
    //操作结果: 将 t 连接到当前串 s 之后
    ClearString()
    //操作条件: 已有串 s
    //操作结果: 将当前串 s 清空
    ReplaceString(pos, len, t)
    //操作条件: 已有串 s 和参数串 t
    //操作结果: 将当前串 s 中第 pos 个字符开始的长度为 len 的子串,替换为 t
}ADT String;
```

串的逻辑结构与线性表极为相似,区别仅在于串的数据对象约束为字符类型,所以串的ADT 定义中的数据对象和数据关系与线性表基本相同。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,在线性表中常用的查找某个元素,在某个位置上插入一个元素和删除一个元素等;而在串的基本操作中,通常以“串的整体”作为操作对象,例如在串中查找某个子串,在串的某个位置上插入一个子串以及删除一个子串,即使长度为 1,即单个字符时,也称之为子串,而非单个元素。

3.2 串的存储表示

从串的定义可以看到,串在本质上仍是一种线性结构。故串的存储结构亦可分为顺序存储方式和链式存储方式。两种存储方式各有优缺点,具体选用哪种存储方式要依据实际问题来选取。下面分别介绍这两种存储方式。

3.2.1　串的顺序存储

串的顺序存储结构与线性表的顺序存储结构相似，可以用数组来存储串。在一组地址连续的存储单元中，将串中的字符依次存储在数组的相邻单元中。采用顺序存储结构的串称为顺序串。借助 C++中的数组类型来表示顺序存储结构，由于顺序存储结构是预先分配空间，其大小与表中实际元素个数不一定相等，所以将串所占用空间的大小称为串容量，实际存在的元素个数称为串长，如图 3-1 所示。

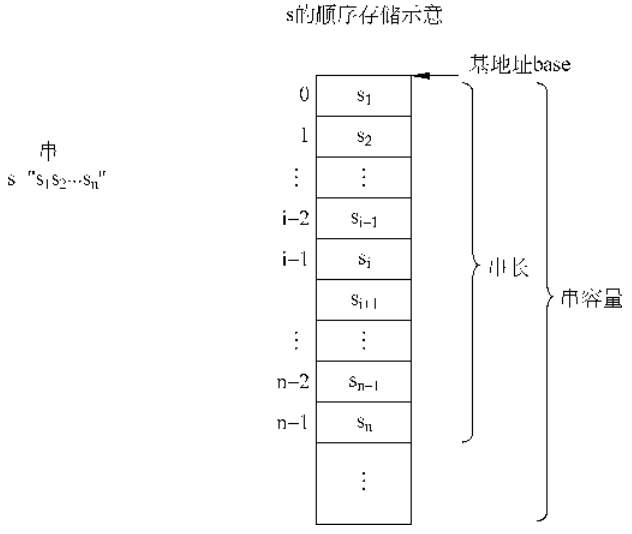


图 3-1　串的顺序存储结构

为了表示串的结束，可在串内容最后一个有效字符后，再多开辟一个存储空间，存放结束标识'\0'(C/C++语言中的字符串就采用这种方法)，如图 3-2 所示。

d	a	t	a		s	t	r	u	c	t	u	r	e	\0

(a) 串"data structure"的存储结构

	\0

(b) 空格串(长度为 1 时)的存储结构

图 3-2　带结束符的串顺序存储结构

在某些时候，会频繁用到串的长度，用末尾添加结束符的方法不利于快速地获得串长，因此可以单独设置一个表示串长的变量，由该变量隐含地指定串的结束。借助于顺序存储时数组的 0 号下标存储串长，既有效地利用了空间，又使得串中字符的位序与其存放位置(下标)一致，如图 3-3 所示。

14	d	a	t	a		s	t	r	u	c	t	u	r	e

图 3-3　带串长度的存储结构

将串定义成字符数组，串的存储空间分配在编译时完成，不能更改，这是一种静态存储方式。考虑到串的操作过程中对空间的需求是不确定的，不宜直接采用数组的声明方式，与顺序表相仿，采用串基址（即顺序存储串时，为其分配的空间的首地址）取而代之，另设表示串存储空间容量（maxlen）的变量，以便灵活地进行顺序串存储空间的管理。因此，本书采用以下方式定义顺序串：

```
#define MAX 100
class SqString
{
public:
    char *base;                          //存储串的字符数组
    //base[0]表示串的实际长度,不另设结束标识
    int maxlen;                          //表示串的最大长度
public:
    SqString();                          //构造函数①
    SqString(char *s);                   //构造函数②
    SqString(SqString &t);               //构造函数③
    ~SqString();                         //析构函数

    bool InsertString(int pos,SqString &t);          //在串的指定位置 pos 插入一个子串 t
    bool DelSubString(int pos,int len,SqString &t);  //删除当前串的子串,并由 t 返回
    void OutputString();                             //输出串中所有字符
    void ConnectString(SqString &t);                 //在当前串尾连接串 t
    bool SubString(int pos,int len,SqString &t);     //求当前串从 pos 开始长度为 len 的子串
    int Indexof(SqString &t);                        //求模式串 t 在当前串中第一次出现的位置
    int Indexof_KMP(SqString &t);              //KMP 法求模式串 t 在当前串中第一次出现的位置
    void GetNext(int next[]);                        //KMP 模式匹配算法辅助函数
    //其他操作
};
```

下面分析其中几个基本操作的实现。

1. 顺序串的构造与析构

串的构造有 3 种方法，分别是构造空串、由基本类型的字符串构造一个新串以及使用串对象来构造串。下面给出 3 种方法构造串及析构串的实现过程。

（1）构造空的顺序串

【算法 3-1-1】

```
SqString::SqString(){

    maxlen = MAX;
    base = new char[maxlen + 1];                     //0 下标留作记录长度
    base[0] = 0;
}
```

（2）由基本字符串构造一个新串

【算法 3-1-2】

```
SqString::SqString(char *s)                          //由机内标准串构造
```

```
{
    maxlen = MAX;
    base =  new char[maxlen + 1];
    base[0] = strlen(s);
    for(int i = 0;s[i]!= '\0';i++)
        base[i + 1] = s[i];
}
```

(3) 使用串对象来构造串

【算法 3-1-3】

```
SqString::SqString(SqString &t)
{
    maxlen = t.maxlen;
    base = new char[maxlen + 1];
    base[0] = t.base[0];
    for(int i = 1;i <= base[0];i++)
        base[i] = t.base[i];
}
```

(4) 析构函数

【算法 3-1-4】

```
SqString::~SqString()
{
    delete[] base;
}
```

2. 顺序串的插入操作

顺序串的插入操作是指将一个指定的串插入到当前串中的指定位置之前，以 s 串为待插入串，t 串为 pos 指定的插入位置，则插入前后 s 串与 t 串的状态如图 3-4(a)和图 3-4(b)所示。

图 3-4　顺序串的插入

顺序串插入操作的实现算法如下。

(1) 检查插入位置的合法性，即当插入位置 pos＜1 或 pos＞base[0]，或 base[0]＋t.base[0]＞maxlen(没有足够空间插入 t)时，提示错误信息，终止程序。

(2) 从 pos 指向的位置开始,一直到最后的字符,每个字符都要向后移动,移动的长度为 t 串的长度。

(3) 插入 t 串,修改 s 的串长,操作成功,结束。

【算法 3-2】

```
bool SqString::InsertString(int pos,SqString &t)
{

    if(pos<1||pos>base[0]+1||pos-1+t.base[0]>maxlen)
    {
        cout<<"插入失败"<<endl;
        return false ;
    }
    else
    {
        for(int i=base[0];i>=pos;i--)
            base[i+t.base[0]]=base[i];          //元素后移
        for(i=1;i<=t.base[0];i++)
            base[pos-1+i]=t.base[i];            //插入元素
        base[0]+=t.base[0];
        return true;
    }
}
```

3. 顺序串的删除操作

顺序串删除操作的功能是删除 s 串中从第 pos 个位置开始的长度为 len 的子串,如图 3-5 所示。

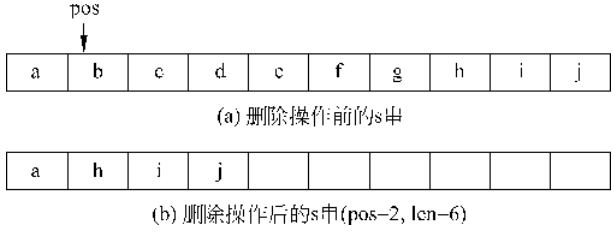


图 3-5 顺序串的删除

顺序串删除操作的实现算法如下。

(1) 检查参数的合法性,有两种不合法的操作条件:一是 pos 的值不在串 s 的长度范围内,即 pos<1 或 pos>base[0];二是从串 s 第 pos 个位置开始不存在长度为 len 的子串,即 pos+len−1>base[0]。

(2) 将待删除的子串复制给 t。

(3) 在 s 中删除指定的子串,修改 s 的串长,操作成功,结束。

【算法 3-3】

```
bool SqString::DelSubString(int pos,int len,SqString &t)
```

```
{
    if((pos<1)||(pos>base[0])||pos-1+len>base[0])
    {
        cout<<"删除失败"<<endl;
        return false;
    }
    else
    {
        for(int i=pos;i<=pos-1+len;i++)        //将删除的元素复制给 t
            t.base[i-pos+1]=base[i];
        t.base[0]=len;
        for(i=pos+len;i<=base[0];i++)          //元素前移
        {
            base[i-len]=base[i];
        }
        base[0]-=len;
        return true;
    }
}
```

4. 输出顺序串的操作

顺序串输出操作的功能是将串中的字符全部输出。顺序串输出操作的实现算法如下。

(1) 检查串是否为空串,若为空,输出空串信息。

(2) 若串非空,则利用循环输出串的内容。

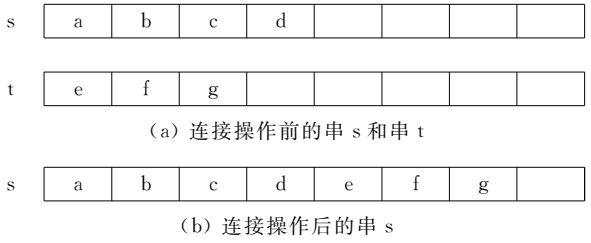
(3) 操作成功,结束。

【算法 3-4】

```
void SqString::OutputString()
{
    if(base[0]==0)                             //判断串是否为空串
        cout<<"空串"<<endl;
    else
    {
        for(int i=1;i<=base[0];i++)
            cout<<base[i];
        cout<<endl;
    }
}
```

5. 串的连接

串的连接,顾名思义,是指将两个已有的串连接成为一个串。例如,有两个串 s 和 t 都是类 SqString 的对象,那么两个串连接后为 s=s+t,如图 3-6 所示。由于顺序串存储空间固定,因此在连接时要检验是否会溢出,为增加操作的灵活性,还要实现溢出时重新分配空间的功能。



(a) 连接操作前的串 s 和串 t

(b) 连接操作后的串 s

图 3-6 顺序串的连接

顺序串连接操作实现算法如下。

(1) 计算连接后的串长,如果超出顺序串的 maxlen,重新分配空间。

(2) 否则,从当前串的第 base[0]+1 个位置起,依次将 t 串中每一个字符复制到 s。

(3) 更新当前串长,操作成功,返回。

【算法 3-5】

```
void SqString::ConnectString(SqString &t)
{
    if(base[0] + t.base[0]> maxlen)
    {
        char *p = new char[base[0] + t.base[0]];
        for(int i = 1;i<= base[0];i++)
            p[i] = base[i];
        delete []base;
        base = p;
        maxlen = base[0] + t.base[0];
    }
    for(int i = 1;i<= t.base[0];i++)
        base[base[0] + i] = t.base[i];
    base[0] += t.base[0];
}
```

6. 求子串(非空子串)

求子串的定义为将串 s 中的第 pos 个字符开始长度为 len 的子串,复制到串 t 中,如图 3-7 所示。

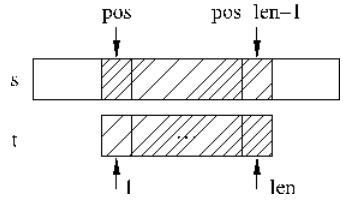


图 3-7 求子串示意图

顺序串中求子串的实现算法如下。

(1) 检查参数的合法性,当 pos<1 或 pos>base[0],或 len<1 或 pos+len−1>base[0]时,操作失败。

(2) 将当前串从 pos 指向位置开始的长度为 len 的子串复制到串 t 中。

(3) 操作成功,结束。

【算法 3-6】

```
bool SqString::SubString(int pos,int len,SqString &t)
{
    if((pos<1)||(pos>base[0])||pos-1+len>base[0])
```

```
    {
        cout <<"操作失败"<< endl;
        return false;
    }
    else
    {
        for(int i = pos;i <= len;i++)
            t.base[i - pos + 1] = base[i];
        t.base[0] = len;
        return true;
    }
}
```

3.2.2　串的链式存储

串的顺序存储方式节约了系统开销，但是如果需要经常对串执行插入、删除子串等操作，就需要频繁移动串中的字符，因此引入串的另一种存储方式——链式存储，又称动态存储。这样就可以避免频繁的插入、删除操作带来的效率低下等问题。

在链式存储结构下，存储空间被分成一系列大小相同的结点，每个结点包含两个域：字符域 data 和指针域 next。其中，字符域用来存放字符，指针域则用来存放指向下一个结点的指针。

一个串可用一个单链表来存储。链表中的结点数目等于串的长度。例如，一个字符串 s="ABCDEFGHI"，那么它的单链表存储方式如图 3-8 所示。

图 3-8　结点大小为 1 的链串存储方式

链式存储结构的最大优点是插入、删除操作方便，但是我们发现，由于串结构的特殊性，存在这样一个问题，每个结点仅存放一个字符，而指针域所占空间比字符域所占空间要大数倍。为了提高串的链式存储的存储密度，节省空间，可以将链串的结点大小设置为 4。那么串 s="ABCDEFGHI"在结点大小为 4 的链串存储结构如图 3-9 所示。链串的最后一个结点的 data 域不一定全被串的字符占满，一般采用特殊字符填充(本书采用"#"号)。这种存储串的方法称为块链式存储。

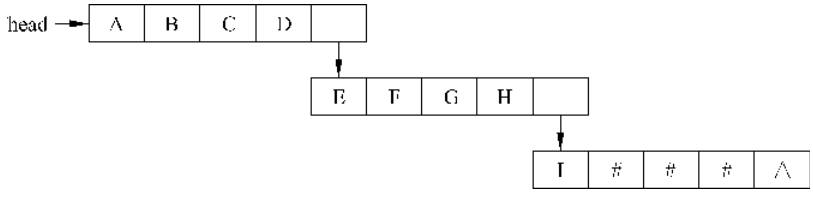


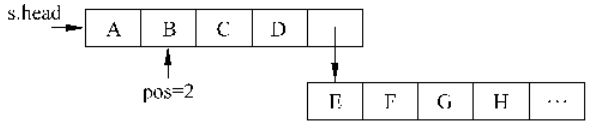
图 3-9　结点为 4 的链串存储方式

链串的存储结构可定义如下：

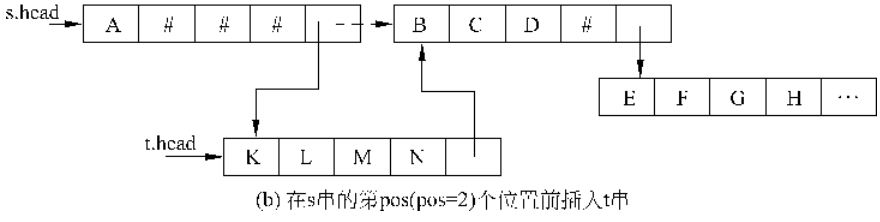
```
#define N 4
```

```
struct LStringNode
{
    char data[N];
    struct LStringNode * next;
};
class LinkString
{
    LStringNode * head;
    int length;
public:
    LinkString();
    LinkString(char * t);
    LinkString(LinkString &t);
    ~LinkString();
    bool InsertString(int pos,LinkString &t);
    bool DelSubString(int pos,int len,LinkString &t) ;
    void OutputString();
    //其他操作
};
```

下面仅分析一下链串的插入操作。链串的插入操作与单链表的插入过程相似，但又有明显的区别。单链表中每一个结点都是一个单独的元素，而块链式的串中每一块有若干个独立的元素，如图 3-10(a)所示，当插入位置不是刚好位于每一块的起始处时，插入子串的处理相对要复杂。为尽量减少插入时字符的移动，可采用牺牲一定存储空间的办法，将插入点所在块的串拆分成两个块，无效字符的位置用“#”填充，如图 3-10(b)所示，这样待插入的串就可以直接进行链接。



(a) 插入前的s串



(b) 在s串的第pos(pos=2)个位置前插入t串

图 3-10　链式串的插入

链串插入操作的实现算法如下。

(1) 判断插入位置是否有效，无效立即结束；否则继续。

(2) 找到插入位置，以指针 p 指向 pos 所在块或其前一块。若 pos 对块长取余不为 0，p 指向 pos 所在块，生成新结点，对该块进行拆分；否则，p 指向 pos 所在块的前一块。

(3) 将 t 串链接到 s 串中。

(4) 操作成功，结束。