第 4 章 串

教材中练习题及参考答案

1. 串是一种特殊的线性表，请从存储和运算两方面分析它的特殊之处。

答：从存储方面看，串中每个元素是单个字符，在设计串存储结构时可以每个存储单元或者结点只存储一个字符。从运算方面看，串有连接、判串相等、求子串和子串替换等基本运算，这是线性表的基本运算中所没有的。

2. 为什么模式匹配中，BF 算法是有回溯算法，而 KMP 算法是无回溯算法？

答：设目标串为 *s*，模式串为 *t*。在 BF 算法的匹配过程中，当 *t*[*j*]=s[*i*]时，置 *i*++，*j*++；当 *t*[*j*]≠*s*[*i*]时，置 *i*=*i*−*j*+1，*j*=0。从中看到，一旦两字符不等，目标串指针 *i* 会回退，所以 BF 算法是有回溯算法。在 KMP 算法的匹配过程中，当 *t*[*j*]=*s*[*i*]时，置 *i*++，*j*++；当 *t*[*j*]≠*s*[*i*]时，*i* 不变，置 *j*=*next*[*j*]。从中看到，目标串指针 *i* 不会回退，只会保持位置不变或者向前推进，所以 KMP 算法是无回溯算法。

3. 在 KMP 算法中，计算模式串的 *next* 时，当 *j*=0 时，为什么要置 *next*[0]=−1？

答：当模式串中 t_0 字符与目标串中某字符 s_i 比较不相等时，此时置 *next*[0]=−1 表示模式串中已没有字符可与目标串的 s_i 比较，目标串当前指针 *i* 应后移至下一个字符，再和模式串的 t_0 字符进行比较。

4. KMP 算法是简单模式匹配算法的改进，以目标串 *s*="*aabaaabc*"、模式串 *t*="*aaabc*"为例说明的 *next* 的作用。

答：模式串 *t*="*aaabc*"的 *next* 数组值如表 4.1 所示。

表 4.1 模式串 t 对应的 next 数组

j	0	1	2	3	4
t[*j*]	*a*	*a*	*a*	*b*	*c*
next[*j*]	−1	0	1	2	0

从 *i*=0，*j*=0 开始，当两者对应字符相等时，*i*++，*j*++，直到 *i*=2，*j*=2 时对应字符不相等。如果是简单模式匹配，下次从 *i*=1，*j*=0 开始比较。

KMP 算法已经获得了前面字符比较的部分匹配信息，即 *s*[0..1]=*t*[0..1]，所以 *s*[0]=*t*[0]，而 *next*[2]=1 表明 *t*[0]=*t*[1]，所以有 *s*[0]=t[1]，这说明下次不必从 *i*=1，*j*=0 开始比较，而只需保持 *i*=2 不变，让 *i*=2 和 *j*=*next*[*j*]=1 的字符进行比较。

i=2，*j*=1 的字符比较不相等，保持 *i*=2 不变，取 *j*=*next*[*j*]=0。

i=2，*j*=0 的字符比较不相等，保持 *i*=2 不变，取 *j*=*next*[*j*]=−1。

当 *j*=−1 时 *i*++、*j*++，则 *i*=3，*j*=0，对应的字符均相等，一直比较到 *j* 超界，此时表示匹配成功，返回 3。

从中看到，*next*[*j*]保存了部分匹配的信息，用于提高匹配效率。由于是在模式串的 *j* 位置匹配失败的，*next* 也称为失效函数或失配函数。

5. 给出以下模式串的 *next* 值和 *nextval* 值：

（1）*ababaa*

（2）*abaabaab*

答：（1）求其 *next* 和 *nextval* 值如表 4.2 所示。

表 4.2 模式串"*ababaa*"对应的 next 数组

j	0	1	2	3	4	5
t[*j*]	*a*	*b*	*a*	*b*	*a*	*a*
next[*j*]	−1	0	0	1	2	3
nextval[*j*]	−1	0	−1	0	−1	3

（2）求其 *next* 和 *nextval* 值如表 4.3 所示。

表 4.3 模式串"*abaabaab*"对应的 next 数组

j	0	1	2	3	4	5	6	7
t[*j*]	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*
next[*j*]	−1	0	0	1	1	2	3	4
nextval[*j*]	−1	0	−1	1	0	−1	1	0

6. 设目标为 *s*="*abcaabbabcabaacbacba*"，模式串 *t*="*abcabaa*"。

（1）计算模式串 *t* 的 *nextval* 数组。

（2）不写算法，给出利用改进的KMP算法进行模式匹配的过程。

（3）问总共进行了多少次字符比较？

解：（1）先计算*next*数组，在此基础上求*nextval*数组，如表4.4所示。

表 4.4 计算 next 数组和 nextval 数组

j	0	1	2	3	4	5	6
t[*j*]	*a*	*b*	*c*	*a*	*b*	*a*	*a*
next[*j*]	−1	0	0	0	1	2	1
nextval[*j*]	−1	0	0	−1	0	2	1

（2）改进的 KMP 算法进行模式匹配的过程如图 4.2 所示。

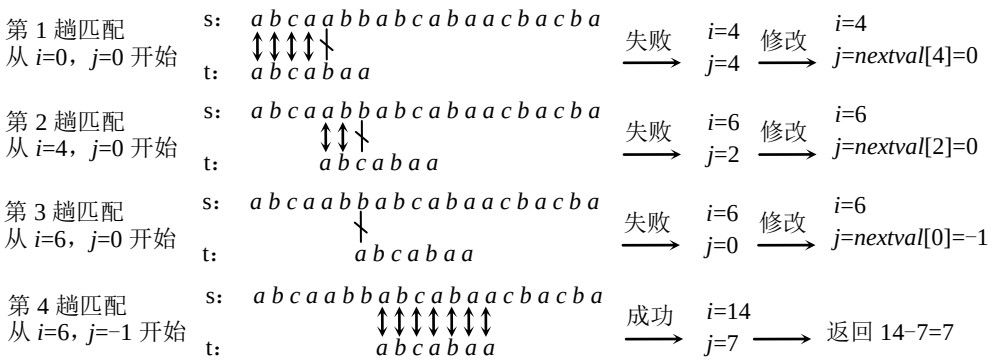

图 4.2 改进的 KMP 算法模式匹配的过程

（3）从上述匹配过程看出：第 1 趟到第 4 趟的字符比较次数分别是 5、3、1、7，所以总共进行了 16 次字符比较。

7. 有两个顺序串 *s*1 和 *s*2，设计一个算法求一个顺序串 *s*3，该串中的字符是 *s*1 和 *s*2 中公共字符（即两个串都包含的字符）。

解：扫描 *s*1，对于当前字符 *s*1.*data*[*i*]，若它在 *s*2 中出现，则将其加入到串 *s*3 中。最后返回 *s*3 串。对应的算法如下：

```
SqString CommChar(SqString s1,SqString s2)
{   SqString s3;
    int i,j,k=0;
    for (i=0;i<s1.length;i++)
    {   for (j=0;j<s2.length;j++)
            if (s2.data[j]==s1.data[i])
                break;
        if (j<s2.length)            //s1.data[i]是公共字符
        {   s3.data[k]=s1.data[i];
            k++;
        }
    }
    s3.length=k;
    return s3;
}
```

8. 采用顺序结构存储串，设计一个实现串通配符匹配的算法 pattern_index()，其中的通配符只有‘？’，它可以和任一个字符匹配成功。例如，pattern_index("？*re*"，"*there are*") 返回的结果是 2。

解：采用 BF 算法的穷举法的思路，只需要增加对‘?’字符的处理功能。对应的算法如下：

```
int index(SqString s,SqString t)
{   int i=0,j=0;
    while (i<s.length && j<t.length)
    {   if (s.data[i]==t.data[j] || t.data[j]=='?')
        {   i++;
            j++;
```

```
            }
            else
            {   i=i-j+1;
                j=0;
            }
        }
        if (j>=t.length)
            return(i-t.length);
        else
            return(-1);
    }
```

9. 设计一个算法，在顺序串 *s* 中从后向前查找子串 *t*，即求 *t* 在 *s* 中最后一次出现的位置。

解：采用简单模式匹配算法。如果串 *s* 的长度小于串 *t* 的长度，直接返回-1。然后 *i* 从 *s.length*-*t.length* 到 0 循环：再对于 *i* 的每次取值循环：置 *j*=*i*，*k*=0，若 *s.data*[*j*]==*t.data*[*k*]，则 *j*++，*k*++。循环中当 *k*==*t.length* 为真时，表示找到子串，返回物理下标 *i*。所有循环结束后都没有返回，表示串 *t* 不是串 *s* 的子串则返回-1。对应的算法如下：

```
int LastPos1(SqString s,SqString t)
{   int i,j,k;
    if (s.length-t.length<0)
        return -1;
    for (i=s.length-t.length;i>=0;i--)
    {   for (j=i,k=0;j<s.length && k<t.length && s.data[j]==t.data[k];j++,k++);
        if (k==t.length)
            return i;
    }
    return -1;
}
```

10. 设计一个算法，判断一个字符串 *s* 是否形如"序列 1@为序列 2"模式的字符序列，其中序列 1 和序列 2 都不含有'@'字符，且序列 2 是序列 1 的逆序列。例如"*a*+*b*@*b*+*a*"属于该模式的字符序列，而"1+3@3-1"则不是。

解：建立一个临时栈 *st* 并初始化为空，其元素为 char 类型。置匹配标志 *flag* 为 true。扫描顺序串 *s* 的字符，将'@'之前的字符进栈。继续扫描顺序串 *s* 中'@'之后的字符，每扫描一个字符 *e*，退栈一个字符 *x*，若退栈时溢出或 *e* 不等于 *x*，则置 *flag* 为 false。循环结束后，若栈不空，置 *flag* 为 false。最后销毁栈 *st* 并返回 *flag*。对应的算法如下：

```
bool symm(SqString s)
{   int i=0; char e,x;
    bool flag=true;
    SqStack *st;
    InitStack(st);
    while (i<s.length)      //将'@'之前的字符进栈
    {   e=s.data[i];
        if (e!='@')
            Push(st,e);
        else
```

```
            break;
        i++;
    }
    i++;                    //跳过@字符
    while (i<s.length &&flag)
    {   e=s.data[i];
        if (!Pop(st,x)) flag=false;
        if (e!=x) flag=false;
        i++;
    }
    if (!StackEmpty(st)) flag=false;
    DestroyStack(st);
    return flag;
}
```

11. 采用顺序结构存储串，设计一个算法求串 *s* 中出现的第一个最长重复子串的下标和长度。

解：采用简单模式匹配算法的思路，先给最长重复子串的起始下标 *maxi* 和长度 *maxlen* 均赋值为 0。用 *i* 扫描串 *s*，对于当前字符 s_i，判定其后是否有相同的字符，若有记为 s_j，再判定 s_{i+1} 是否等于 s_{j+1}，s_{i+2} 是否等于 s_{j+2}，…，直至找到一个不同的字符为止，即找到一个重复出现的子串，把其起始下标 *i* 与长度 *len* 记下来，将 *len* 与 *maxlen* 相比较，保留较长的子串 *maxi* 和 *maxlen*。再从 s_{j+len} 之后查找重复子串。然后对于 s_{i+1} 之后的字符采用上述过程。循环结束后，*maxi* 与 *maxlen* 保存最长重复子串的起始下标与长度，将其复制到串 *t* 中。对应的算法如下：

```
void maxsubstr(SqString s,SqString &t)
{   int maxi=0,maxlen=0,len,i,j,k;
    i=0;
    while (i<s.length)              //从下标为 i 的字符开始
    {   j=i+1;                      //从 i 的下一个位置开始找重复子串
        while (j<s.length)
        {   if (s.data[i]==s.data[j]) //找一个子串,其起始下标为 i,长度为 len
            {   len=1;
                for(k=1;s.data[i+k]==s.data[j+k];k++)
                    len++;
                if (len>maxlen)     //将较大长度者赋给 maxi 与 maxlen
                {   maxi=i;
                    maxlen=len;
                }
                j+=len;
            }
            else j++;
        }
        i++;                        //继续扫描第 i 字符之后的字符
    }
    t.length=maxlen;                //将最长重复子串赋给 t
    for (i=0;i<maxlen;i++)
        t.data[i]=s.data[maxi+i];
}
```

12. 用带头结点的单链表表示链串，每个结点存放一个字符。设计一个算法，将链串 *s* 中所有值为 *x* 的字符删除。要求算法的时间复杂度均为 O(*n*)，空间复杂度为 O(1)。

解：让 *pre* 指向链串头结点，*p* 指向首结点。当 *p* 不为空时循环：当 *p*->*data*==*x* 时，通过 *pre* 结点删除 *p* 结点，再让 *p* 指向 *pre* 结点的后继结点；否则让 *pre*、*p* 同步后移一个结点。对应的算法如下：

```
void deleteall(LinkStrNode *&s,char x)
{   LinkStrNode *pre=s,*p=s->next;
    while (p!=NULL)
    {   if (p->data==x)
        {   pre->next=p->next;
            free(p);
            p=pre->next;
        }
        else
        {   pre=p;          //pre、p 同步后移
            p=p->next;
        }
    }
}
```