第四章 字符串

- 字符串的概念
- 字符串的实现
- 字符串的模式匹配

42-2

字符串的概念

- 字符串是 n (≥ 0) 个字符的有限序列，
  记作 S : “$c_1c_2c_3…c_n$”
  其中，S 是串名字
  “$c_1c_2c_3…c_n$”是串值
  c_i 是串中字符
  n 是串的长度，n = 0 称为空串。
- 例如，S = “Tsinghua University”。
- 注意：空串和空白串不同，例如 “ ” 和 “” 分别表示长度为1的空白串和长度为0的空串。

42-3

- 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，该子串首字符对应的主串中的序号，定义为子串在主串中的位置。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，首次出现所对应的主串位置是2（从0开始）。因此，称 B 在 A 中的位置为2。
- 特别地，空串是任意串的子串，任意串是其自身的子串。

42-4

- 通常在程序中使用的串可分为两种：串变量和串常量。
- 串常量在程序中只能被引用但不能改变它的值，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
  char path[] = “dir/bin/appl”;
- 这里path存储的是一个串常量。串变量和其它类型的变量一样，其取值可以改变。

42-5

在C中常用的字符串操作

- 字符串初始化
  - char name[12] = “Tsinghua”;
  - char name[ ] = “Tsinghua”;
  - char name[12] = {‘T’,‘s’,‘i’,‘n’,‘g’,‘h’,‘u’,‘a’};
  - char name[ ] = {‘T’,‘s’,‘i’,‘n’,‘g’,‘h’,‘u’,‘a’,‘\0’};
  - char *name = “Tsinghua”;
  - char name[12];
    name = “Tsinghua”; × 因数组名是地址常量

42-6

- 单个字符串的输入函数 gets (str)
  例 char name[12];
  gets (name);
- 字符串输出函数 puts (str)
  例 char name[12];
  gets (name);
  puts (name);
- 字符串求长度函数 strlen(str)
  字符串长度不包括 "\0"和分界符
  例 int m = strlen ("University");
  printf ("%d\n", m); //输出10

42-7

- 字符串连接函数 strcat (str1, str2)

例 str1	"Tsinghua\0"	//连接前
str2	"University\0"	//连接前
str1	"TsinghuaUniversity\0"	//连接后
str2	"University\0"	//不变

- 字符串比较函数 strcmp (str1, str2)
  //从两个字符串第 1 个字符开始，逐个对应字符进
  //行比较，全部字符相等则函数返回0，否则在不
  //相等字符处停止比较，函数返回其差值
  // (比较基于 ASCII代码)
  例 str1 "University" i 的代码值105
  str2 "Universal" a 的代码值97，差8

42-8

字符串的实现

- 除 C 语言提供的字符串库函数外，可以自定义字符串。适用于自定义字符串数据类型的有三种存储表示：定长顺序存储表示、堆分配存储表示、块链存储表示。

定长顺序存储表示

- 即顺序串，使用静态分配的字符数组存储字符串中的字符序列。
- 字符数组的长度预先用 maxSize 指定，一旦空间存满不能扩充。

42-9

- 有两种实现方法：
  - 字符存放于字符数组的 0～maxSize 单元，另外用整数n记录串中实际存放的字符个数；
  - 字符存放于字符数组的 1～maxSize 单元，用 0 号单元记录串中实际存放的字符个数。
- 本教材采用前者。
- 按照 C 语言规定，在字符串值最后有一个特殊的"\0"表示串值的结束。因此，在存放串值时要求为它留一个位置。
- 定长存储表示的定义如下：

42-10

```
#define maxSize 256            //顺序串的预设长度
Typedef struct {               //顺序串的定义
    char ch[maxSize+1];        //存储数组
    int n;                     //串中实际字符个数
} SeqString;
```

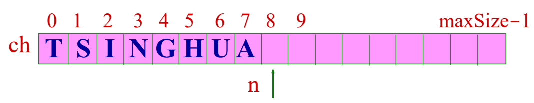
堆分配存储表示

- 即堆式串。字符数组的存储空间是动态分配的。串的最大空间数 maxSize 和串中实际字符个数 n 保存在串的定义中。
- 可以根据需要，随时改变字符数组的大小。

42-11

堆分配字符串的结构定义

```
#define defaultSize 256;
typedef struct {
    char *ch;          //串的存储数组
    int maxSize;       //串数组的最大长度
    int n;             //串的当前长度
} HString;
```



42-12

堆式串部分操作的实现

```
void initStr ( HString& S ) {
//初始化：创建字符串 S 的存储空间并置空串
    S.ch = ( char * ) malloc ((defaultSize+1)*sizeof
        (char));                    //分配字符数组空间
    if ( S.ch == NULL ) exit (1);   //判断分配成功与否
    S.ch[0] = '\0';                 //置空串
    S.maxSize = defaultSize;        //置串的最大字符数
    S.n = 0;                        //实际字符数置0
}
```

42-13

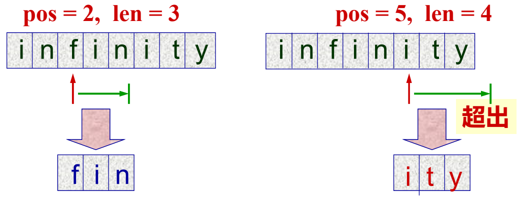
```
void createStr ( HString& s, char *init ) {
//从字符数组 init 构造串 s, 要求 s 已存在并初始化
    s.maxSize = defaultSize;
    s.ch = ( char *) malloc (( defaultSize+1 )*sizeof
        ( char ));
    s.n = strlen ( init );  strcpy ( s.ch, init );
}
void copyStr ( HString& s, HString& t ) {
//把串 t 复制给串 s , 要求 s 已存在并初始化
    s.maxSize = defaultSize;
    s.ch = ( char *) malloc (( defaultSize+1 )*sizeof
        ( char ));
```

42-14

```
    s.n = t.n;  strcpy ( s.ch, t.ch );
}
void printStr ( HString& s ) {
//打印字符串 s
    printf ("串长度=%d, 最大长度=%d\n",
        s.n, maxSize);
    for ( int i = 0; i < s.n; i++ )
        if ( s.ch[i] == '\0' ) break;
        else printf ( "%c", s.ch[i] );
    printf ( "\n");
}
```

42-15

提取子串的算法示例



pos+len−1 ≤ curLen−1
可以全部提取

pos+len −1 ≥ curLen
只能从pos取到串尾

42-16

串操作：提取子串

```
HString subStr ( HString& s, int pos, int len ) {
//在串 s 中连续取从 pos 开始的 len 个字符，构成子串
//返回。若提取失败则函数返回NULL
    HString tmp;
    tmp.ch = ( char *) malloc (( defaultSize+1 )*sizeof
        ( char ));                  //创建子串空间
    tmp.maxSize = defaultSize;
    if ( pos < 0 || len < 0 || pos+len-1 >= s.maxSize )
                                    //参数不合理，返回空串
        { tmp.n = 0;  tmp.ch[0] = '\0'; }
    else {
```

42-17

```
        if ( pos+len−1 >= s.n ) len = s.n−pos;
                    //若提取个数超出串尾，修改个数
        for ( int i = 0, j = pos; i < len; i++, j++ )
            tmp.ch[i] = s.ch[j];    //复制子串的字符
        tmp.n = len;  tmp.ch[len] = '\0';
    }
    return tmp;                     //返回复制的子串
}
```

- 例：串 st = “university”, pos = 3, len = 4
  使用示例 HString t = subStr(st, 3, 4)
  提取子串 t = “vers”

42-18

串操作: 串连接

```
void concatStr ( HeapString& s, HeapString& t ) {
//函数将串 t 复制到串 s 之后，通过串 s 返回结果，
//串 t 不变。
  if ( s.n+t.n <= s.maxSize ) {
                      //原空间可容纳连接后的串
     for ( int i = 0; i < t.n; i++ )
        s.ch[s.n+i] = t.ch[i];    //串 t 复制到串 s 后
     s.n = s.n+t.n;  s.ch[s.n] = '\0';
  }
  else {              //原空间容不下连接后的串
     char *tmp = s.ch;  s.maxSize = s.n+t.n;
```



```
     s.ch = ( char* ) malloc (( s.maxSize+1 )*sizeof
        ( char ));    //按新的大小分配存储空间
     strcpy ( s.ch, tmp );      //复制原串 s 数组
     strcat ( s.ch, t.ch );     //连接串 t 数组
     s.n = s.n+t.n;  free ( tmp );
  }
}
```

- 例：串 st1 = “beijing ”,
  st2 = “university”,

  使用示例　concatStr (st1, st2);

  连接结果　st1 = “beijing university”

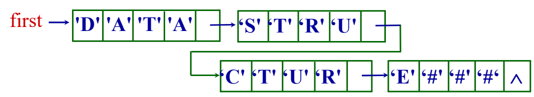
  st2 = “university”

块链存储表示

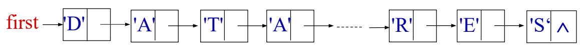
- 使用单链表作为字符串的存储表示，此即字符串的链接存储表示。
- 链表的每个结点可以存储 1 个字符，称其“块的大小”为 1，也可以存储 n 个字符，称其“块的大小”为 n。
- 定义存储密度为：

$$存储密度 = \frac{该串的串值占用的存储空间大小}{为该串分配的存储空间总大小}$$

- 显然，存储密度越高，存储利用率越高。



(a) 结点大小为4



(b) 结点大小为1

- 结点大小为 4 时，存储利用率高，但操作复杂，需要析出单个字符；结点大小为 1 时，存储利用率低，但操作简单，可直接存取字符。
- 块链存储表示一般带头结点，设置头、尾指针。

块链存储表示的结构定义

```
#define blockSize 1        //由使用者定义的结点大小
typedef struct block {     //链表结点的结构定义
   char ch[blockSize];
   struct block *link;
} Chunk;
typedef struct {           //链表的结构定义
   Chunk *first, *last;    //链表的头指针和尾指针
   int n;                  //串的当前长度
} LinkString;
```


字符串的模式匹配

- 定义　在字符串中寻找子串（第一个字符）在串中的位置
- 词汇　在串的模式匹配中，子串称为模式，主串称为目标。
- 示例　目标 T : “Beijing”

  模式 P : “jin”

  匹配结果 = 3 （匹配位置从 0 开始）
- 讨论两种匹配算法：BF 算法和 KMP 算法。

朴素的模式匹配算法（BF算法）

- 初始时让目标T的第 0 位与模式P的第 0 位对齐；
- 顺序比对目标T与模式P中的对应字符：
  - 若 P 与 T 比对发现对应位不匹配，则本趟失配。将 P 右移一位与 T 对齐，进行下一趟比对；
  - 若 P 与 T 对应位都相等，则匹配成功，返回 T 当前比较指针停留位置减去 P 的长度，即目标 T 中匹配成功的位置，算法结束。
  - 若 P 与 T 比对过程中，T 后面所剩字符个数少于 P 的长度，则模式匹配失败。

BF 算法匹配过程的示例

第1趟　T　a b b a b a　↓i=2
　　　 P　a b a　↑j=2

第3趟　T　a b b a b a　↓i=2
　　　 P　　　a b a　↑j=0

第2趟　T　a b b a b a　↓i=1
　　　 P　　a b a　↑j=0

第4趟　T　a b b a b a　↓i=6
　　　 P　　　　a b a　↑j=3

- 这是最简单的模式匹配算法。


```
int Find ( HString& T, HString& P ) {
//在 T 中从第 0 个字符开始寻找 P 在 T 中匹配的位
//置。若在 T 中找不到与 P 匹配的子串, 则函数返回
//-1, 否则返回 P 在 T 中第一次匹配的位置。
   int i, j, k;          //T.n-P.n为在T中最后可比对位置
   for ( i = 0; i <= T.n – P.n; i++) {      //逐趟比对
      for ( k = i, j = 0; j < P.n; k++, j++ )
         if ( T.ch[k] != P.ch[j] ) break;   //比对不等
      if ( j == P.n ) return i;             //匹配成功
   }
   return -1;                               //匹配失败
};
```


- 若设 n 为目标 T 的长度，m 为模式 P 的长度，匹配算法最多比较 $n-m+1$趟。若每趟比较都比较到模式 P 尾部才出现不等，要做 m 次比较，则在最坏情况下，总比较次数 $(n-m+1)*m$。在多数场合下 m 远小于 n，因此，算法的运行时间为$O(n*m)$。
- 低效的原因在于每趟重新比较时，目标 T 的检测指针要回退。
- 如果消除了每趟失配后为实施下一趟比较时目标指针的回退，可以提高模式匹配效率。

T角标j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
目标字符串（T）	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
	a	b	c	a	b	c	a	c	a	b																
		a	b	c	a	b	c	a	c	a	b															
			a	b	c	a	b	c	a	c	a	b														
				a	b	c	a	b	c	a	c	a	b													
					a	b	c	a	b	c	a	c	a	b												
						a	b	c	a	b	c	a	c	a	b											
							a	b	c	a	b	c	a	c	a	b										
模式字符串（P）								a	b	c	a	b	c	a	c	a	b									
									a	b	c	a	b	c	a	c	a	b								
										a	b	c	a	b	c	a	c	a	b							
											a	b	c	a	b	c	a	c	a	b						
												a	b	c	a	b	c	a	c	a	b					
													a	b	c	a	b	c	a	c	a	b				
														a	b	c	a	b	c	a	c	a	b			
															a	b	c	a	b	c	a	c	a	b		
																a	b	c	a	b	c	a	c	a	b	

无回溯的模式匹配 KMP 算法

- KMP是指D.E.Knuth、J.H.Morris和V.R.Pratt。
- 实施KMP算法，若一趟匹配过程比对失配，在做下一趟匹配比对时，目标 T 的检测指针不回退，模式 P 右移，与 T 的检测指针对齐 再开始比对过程，算法的时间代价：
  - 若每趟第一个不匹配，比较$n-m+1$趟，总比较次数最坏达$(n-m)+m = n$。
  - 若每趟第 m 个不匹配，总比较次数最坏亦达到 n。

KMP 算法匹配过程的示例

目标 T　$t_0\ t_1\ t_2\ \ldots\ldots\ t_{m-1}\ \cdots\ t_{n-1}$

模式 P　$p_0\ p_1\ p_2\ \ldots\ldots\ p_{m-1}$　若失配，右移

目标 T　$t_0\ t_1\ t_2\ \ldots\ldots\ t_{m-1}\ t_m\ \cdots\ t_{n-1}$

模式 P　$p_0\ p_1\ \ldots\ldots\ p_{m-2}\ p_{m-1}$　若失配，右移

目标 T　$t_0\ t_1\ \ldots\ldots\ t_i\ t_{i+1}\ldots\ldots\ t_{i+m-2}\ t_{i+m-1}\cdots\ t_{n-1}$

模式 P　$p_0\ p_1\ \ldots\ldots\ p_{m-2}\ p_{m-1}$

T　$t_0\ t_1\ \cdots\ t_{s-1}\ t_s\ \ t_{s+1}\ t_{s+2}\ \cdots\ t_{s+j-1}\ t_{s+j}\ t_{s+j+1}\ \cdots\ t_{n-1}$

P　$p_0\ p_1\ \ p_2\ \ \cdots\ \ p_{j-1}\ \ p_j$

则有　$t_s\ t_{s+1}\ t_{s+2}\ \cdots\ t_{s+j-1} = p_0\ p_1\ p_2 \cdots p_{j-1}$　(1)

为使模式 P 与目标 T 匹配，必须满足

$p_0\ p_1\ p_2 \cdots p_{j-1} \cdots p_{m-1} = t_{s+1}\ t_{s+2}\ t_{s+3}\ \cdots\ t_{s+j}\ \cdots\ t_{s+m}$

如果　$p_0\ p_1\ \cdots\ p_{j-2} \neq p_1\ p_2 \cdots p_{j-1}$　(2)

则立刻可以断定

$p_0\ p_1 \cdots p_{j-2} \neq t_{s+1}\ t_{s+2}\ \cdots\ t_{s+j-1}$

下一趟必不匹配

同样，若　$p_0\ p_1 \cdots p_{j-3} \neq p_2\ p_3 \cdots p_{j-1}$

则再下一趟也不匹配，因为有

$p_0\ p_1 \cdots p_{j-3} \neq t_{s+2}\ t_{s+3}\ \cdots\ t_{s+j-1}$

直到对于某一个 “k”值，使得

$p_0\ p_1 \cdots p_{k+1} \neq p_{j-k-2}\ p_{j-k-1} \cdots p_{j-1}$

且　$p_0\ p_1 \cdots p_k = p_{j-k-1}\ p_{j-k} \cdots p_{j-1}$

则　$p_0\ p_1 \cdots p_k = t_{s+j-k-1}\ t_{s+j-k}\ \cdots\ t_{s+j-1}$

$p_{j-k-1}\ \ p_{j-k}\ \cdots\ p_{j-1}$

下一趟可以直接用 p_{k+1} 与 t_{s+j} 继续比较。

k 的确定方法

- Knuth 等人发现，对于不同的 j（P 中的失配位置），k 的取值不同，它仅依赖于模式 P 本身前 j 个字符的构成，与目标无关。
- 可以用一个 next [ ] 失配函数来确定：当模式 P 中第 j 个字符与目标 T 中相应字符失配时，模式 P 中应当由哪个字符（设为第k+1个）与目标中刚失配的字符重新继续进行比较。
- 设模式 $P = p_0p_1 \ldots p_{m-2}p_{m-1}$，next [ ] 失配函数定义如下：

$$
\text{next}(j) = \begin{cases} -1, & j = 0 \\ k+1, & 0 \le k < j-1 \text{ 且使得 } p_0p_1\cdots p_k = p_{j-k-1}p_{j-k}\cdots p_{j-1} \text{ 的最大整数} \\ 0, & \text{其他情况} \end{cases}
$$

j	0	1	2	3	4	5	6	7
P	a	b	a	a	b	c	a	c
next(j)	-1	0	0	1	1	2	0	1

利用 next 失配函数进行匹配处理

- 若设若在进行某一趟匹配比较时在模式 P 的第 j 位失配：
  - 若 j > 0，那么在下一趟比较时模式 P 的起始比较位置是 $p_{next(j)}$，目标 T 的检测指针不回溯，仍指向上一趟失配的字符；
  - 若 j = 0，则目标 T 检测指针进一，模式 P 检测指针回到 p_0，进行下一趟匹配比较。
- 运用 KMP 算法的匹配过程如下图。

第1趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=1 ⇨ next(1) = 0，下次p_0

第2趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=0 ⇨ 下次p_0, 目标指针进 1

第3趟 目标 a c a b a a b a a b c a c a a b c

模式 a b a a b c a c

× j=5 ⇨ next(5) = 2, 下次p_2

第4趟 目标 a c a b a a b a a b c a c a a b c

模式 (a b) a a b c a c √

用KMP算法实现的快速匹配算法

```
int fastFind ( HString& T, HString& P, int next[ ] ) {
//在目标 T 中寻找模式 P 的匹配位置。若找到，则函
//数返回 P 在 T 中开始字符下标，否则函数返回-1。
//数组next存放 P 的失配函数next[j]值
   int j = 0, i = 0;                //P与T的扫描指针
   while ( j < P.n && i < T.n ) {   //对两串扫描
     if ( j == -1 || P.ch[j] == T.ch[i] ) { j++;  i++; }
                                   //对应字符匹配，比对位置加一
     else j = next[j];  //第 j 位失配，找下一对齐位置
   }
```



```
   if ( j < P.n ) return -1; //j 未比完失配，匹配失败
   else return i-P.n;        //匹配成功
}
```

- 此算法的时间复杂度取决于 while 循环。由于是无回溯的算法，执行循环时，目标 T 字符比较有进无退，要么执行 i++ 和 j++（对应位相等），要么查找 next[ ] 数组进行模式 P 位置的右移，然后继续向后比较。字符的比较次数最多为 $O(n)$，n 是目标 T 的长度。

next 失配函数的计算

- 设模式 $P = p_0 p_1 p_2 \ldots p_{m-1}$由 m 个字符组成，而 next 失配函数为next = $n_0\, n_1\, n_2 \ldots n_{m-1}$，表示了模式的字符分布特征。
- Next 失配函数从0, 1, 2, …, m-1逐项递推计算:

① 当 j = 0时，$n_0 = -1$。设 $j > 0$ 时 $n_{j-1} = k$:

② 当 $k = -1$或 $j > 0$且 $p_{j-1} = p_k$，则 $n_j = k+1$。

③ 当 $p_{j-1} \neq p_k$ 且 $k \neq -1$，令 $k = n_k$，并让③循环直到条件不满足。

④ 当 $p_{j-1} \neq p_k$ 且 $k = -1$，则 $n_j = 0$。

- 以前面的例子说明：

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

j=0: n_0=-1

j=1: k=-1; n_1= =k+1 =0

j=2: k=0; $p_1 \neq p_0$; $k=n_k$= =-1; n_2= =k+1 =0

j=3: k=0; n_3= =k+1 =1

j=4: k=1; $p_3 \neq p_1$; $k=n_k$=0; $p_3=p_0$; n_4= =k+1 =1

j=5: k=1; $p_4=p_1$; n_5= =k+1 =2

j=6: k=2; $p_5 \neq p_2$; $k=n_k$=0; $p_5 \neq p_0$; $k=n_k$=-1; $n_5=k+1=0$

j=7: k=0; $p_6=p_0$; $n_7=k+1$ =1

计算next 失配函数的算法

```
void getNext ( HString& P, int next[] ) {
   int j = 0, k = -1;
   next[0] = -1;              //初始值
   while ( j < P.n )  {//计算next[j]
      while ( k >= 0 && P.ch[j] != P.ch[k] )
        k = next[k];         //缩小前缀、后缀子串长度
      j++;  k++;
      next[j] = k;
   }
};
```

- 设 T = "aaacaaab", P = "aaab"， naxt[]函数为
- 使用 next[] 函数对串 T 执行 KMP 算法的匹配过程如下:

j	0	1	2	3
模式P	a	a	a	b
next[j]	−1	−1	−1	2

	i	0	1	2	3	4	5	6	7
	目标 T	a	a	a	c	a	a	a	b
第一趟	模式 P	a	a	a	b	next[3] = 2			
第二趟			a	a	a	b	next[2] = −1		
第三趟						a	a	a	b

42-43