# 第4章　串

# 4.1 串类型的定义

- 串是由多个或零个字符组成的有限序列，记作
  S＝ ‘$c_1c_2c_3\cdots c_n$’ ($n >=0$)

其中，S是串名字，‘$c_1c_2c_3\cdots c_n$’是串值

$c_1$是串中字符，$n$是串的长度，表示串中字符的数目。

- 空串：零个字符的串称为空串,记作∅
- 子串：串中任意个连续的字符组成的子序列
- 主串：包含子串的串
- 字符在串中的位置：字符在序列中的序号
- 子串在串中的位置：子串的第一个字符在主串中的位置

- **串相等**：当且仅当两个串的值相等
- **空格串**：由一个或多个空格组成的串
- **串的表示**：用一对单引号括起来
- **串的操作**：以“串的整体”为操作对象
- 串的抽象数据类型

**ADT String{**

数据对象：D={$a_i | a_i$ ∈字符集, $i = 1,2,\ldots,n, n$>=0}

数据关系：R={<$a_{i-1}$, $a_i$>|$a_{i-1}$, $a_i$ ∈ D，$i = 1,2,\ldots,n$}

基本操作：见下页

**}ADT String**

# 串的基本操作

- StrAssign(&T,chars) //赋值
- StrCompare(S,T) //比较
- StrLength(S)//串长
- Concat(&S,S1,S2) //连接
- SubString(&Sub,S,pos,len)//子串
- StrCopy(&T,S) //复制
- StrInsert(&S,pos,T)//插入
- StrDelete(&S,pos,len)//删除
- ClearString(&S)//清空
- Index(S,T,pos)//查找子串
- Replace(&S,T,V)//替换
- DestroyString(&S)//销毁
- StrEmpty(S) //判串是否为空

# 4.2 串的表示和实现

- 1. 定长顺序存储表示
- 2. 堆分配存储表示
- 3. 串的块链存储表示
- 4. 串的基本操作实现

# 1.定长顺序存储表示

- 静态分配
  - 每个串预先分配一个固定长度的存储区域。
  - 实际串长可在所分配的固定长度区域内变动
    - 以下标为0的数组分量存放串的实际长度——PASCAL;
    - 在串值后加入”\0”表示结束，此时串长为隐含值——C语言
- 用定长数组描述:

```
#define MAXSTRLEN 255 //最大串长
typedef unsigned char SString[MAXSTRLEN + 1]
  //0号单元存放串的长度
```

# 2.堆分配存储表示

- 以一组地址连续的存储单元存放串值字符序列;
- 存储空间动态分配，用malloc()和free()来管理
- typedef struct{

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

# 3. 串的块链存储表示

- 串的链式存储方式， 结点大小：一个或多个字符

```
#define CHUNKSIZE 80
typedef struct Chunk{
    char ch[CHUNKSIZE];
    struct Chunk *next;
  }Chunk;
typedef struct{
    Chunk* head,*tail;
    int curlen;
  }Lstring;
```

- 存储密度=串值所占的存储位/实际分配的存储位

# 4. 串的基本操作

- 串插入：Status StrInsert(HString &S,int pos, HString T)
- 串赋值：Status StrAssign(HString &S,char *chars)
- 求串长：int StrLength(HString S)
- 串比较：int StrCompare(HString S,HString T)
- 串联接：Status Concat(HString &S,HString S1, HString S2)
- 求子串：Status SubString(HString &Sub, HString S, int pos, int len)
- 串清空：Status ClearString(HString &S)
- 串定位、删除、置换、销毁

```
Status StrInsert(HString &S, int pos, HString T)
//在串S的第pos个位置前插入串T
{ int i;
  if (pos<1||pos>S.length+1) return ERROR;
  if (T.length){
        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];}
        for (i=0; i<=T.length-1;i++)
               S.ch[pos-1+i]=T.ch[i];
        S.length+=T.length;
  } return OK;
}
```

Status StrAssign(HString &S,char *chars)
//生成一个值等于串常量chars的串S

```
{ int i,j;  char *c;
  for (i=0,c=chars;*c;++i,++c);//求chars长度
  if (!i) {S.ch=NULL;  S.length=0;}
  else {
        if (!(S.ch=(char *)malloc(i * sizeof(char))))
                exit(OVERFLOW);
        for (j=0;j<=i-1;j++)
                S.ch[j]=chars[j];
        S.length=i;
       }
  return OK;
}
```

```
int StrLength(HString S)
//求串的长度
{
  return S.length;
}
```

```
int StrCompare(HString S,HString T)
//比较两个串，若相等返回0
 {
   int i;
   for (i=0;i<S.length && i<T.length; ++i)
        if (S.ch[i] != T.ch[i]) return S.ch[i]-T.ch[i];
        return S.length-T.length;
 }
```

Status Concat(HString &S,HString S1,HString S2)
//用S返回由S1和S2联接而成的新串

```
{ int j;
  if (!(S.ch =
  (char*)malloc((S1.length+S2.length)*sizeof(char))))
        exit(OVERFLOW);
  for (j=0;j<=S1.length-1;j++)
     { S.ch[j]=S1.ch[j]; }
  S.length=S1.length+S2.length;
  for (j=0;j<=S2.length-1;j++)
     { S.ch[S1.length+j]=S2.ch[j]; }
  return OK;
}
```

```
Status SubString(HString &Sub, HString S, int pos, int len)
{//用Sub返回串S的第pos个字符开始长度为len的子串
  if (pos<1 || pos>S.length ||
        len<0 || len>S.length-pos+1)
      return ERROR;
  if (!len) { Sub.ch=NULL; Sub.length=0;}
  else {
      Sub.ch=(char*)malloc(len*sizeof(char));
      for (int j=0;j<=len-1;j++){
          Sub.ch[j]=S.ch[pos-1+j];}
      Sub.length=len;
  }   return OK;
}
```

```
Status ClearString(HString &S)
//将S清为空串
{
  if (S.ch) { free(S.ch); S.ch=NULL;}
  S.length=0;
  return OK;
}
```

# 4.3 串的模式匹配算法

- **定义：在串中寻找子串（第一个字符）在串中的位置，称作串的模式匹配**

  词汇：在模式匹配中，子串称为模式，串称为目标。

  示例：目标 S: “Beijing”

  模式 P: “jin”

  匹配结果 = 4

# 1.穷举模式匹配

- 设S=s1,s2,···,sn(主串) P=p1,p2,···,pm(模式串)

  i为指向S中字符的指针，j为指向P中字符的指针

匹配失败：si≠pj时，

$$(s_{i-j+1} \cdots s_{i-1})=(p_1 \cdots p_{j-1})$$

回溯： i=i-j+2 ; j=1

重复回溯太多，O(m*n)

**i=1,2,3 j= 1,2,3**

第1趟 S a b b a b a　　　　<u>穷举的模式</u>

P a b a　　　　<u>匹配过程</u>

**i=2 j= 1**

第2趟 S a b b a b a

P  a b a

**i=3 j= 1**

第3趟 S a b b a b a

P   a b a

**i=4,5,6 j= 1,2,3**

第4趟 S a b b a b a

P    a b a

√

# 求子串位置的定位函数

```
int Index(SString S, SString T,int pos) {
//穷举的模式匹配
  int i=pos;   int j=1;
  while (i<=S[0] && j<=T[0]) {//当两串未检测完，S[0]、S[0]为串长
        if (S[i]==T[j]) {++i; ++j;}
        else {i=i-j+2;  j=1;}
  }
  if (j>T[0]) return i-T[0]; //匹配成功
  else return 0;
}
```

# 2. （课堂不讲）KMP快速模式匹配

- D.E.Knuth, J.H.Morris, V.R.Pratt同时发现无回溯的模式匹配

$$
\begin{array}{ccccccccccccc}
S & s_1 & \cdots & s_{i-j-1} & s_{i-j} & s_{i-j+1} & s_{i-j+2} & \cdots & s_{i-1} & s_i & s_{i+1} & \cdots & s_n \\
 & & & & & \| & \| & \| & \| & \times & & & \\
P & & & & & p_1 & p_2 & \cdots & p_{j-1} & p_j & p_{j+1} & \cdots & p_m
\end{array}
$$

则有 $s_{i-j+1}\, s_{i-j+2} \cdots s_{i-1} = p_1\, p_2 \cdots p_{j-1}$ (1)

为使模式 **P** 与目标 **S** 匹配，必须满足

$p_1\, p_2 \cdots p_{j-1}\, p_j \cdots p_m = s_{i-j+1}\, s_{i-j+2} \cdots s_{i-1}\, s_i \cdots s_{i-j+m}$

如果 $p_1 \cdots p_{j-2} \neq p_2\, p_3 \cdots p_{j-1}$ (2)

由(1)(2)则立刻可以断定

$p_1 \cdots p_{j-2} \neq s_{i-j+2}\, s_{i-j+3} \cdots s_{i-1}$

下一趟必不匹配

同样，若 $p_1 p_2 \cdots p_{j-3} \neq p_3 p_4 \cdots p_{j-1}$

则再下一趟也不匹配，因为有

$$p_1 \cdots p_{j-3} \neq s_{i-j+3} \cdots s_{i-1}$$

直到对于某一个“k”值，使得

$$p_1 \cdots p_k \neq p_{j-k}\, p_{i-k+1} \cdots p_{j-1}$$

且 $p_1 \cdots p_{k-1} = p_{j-k+1}\, p_{j-k+2} \cdots p_{j-1}$

则

$$\begin{array}{ccccc} p_1 \cdots p_{k-1} = & s_{i-k+1} & s_{i-k+2} & \cdots & s_{i-1} \\ & \| & \| & & \| \\ & p_{j-k+1} & p_{j-k+3} & \cdots & p_{j-1} \end{array}$$

模式右滑j-k位

# next数组值

- 假设当模式中第j个字符与主串中相应字符“失配”时，可以拿第k个字符来继续比较，则令next[j]=k

next函数定义：

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

# 手工求next数组的方法

- 序号j 1 2 3 4 5 6 7 8
- 模式P a b a a b c a c
- k 1 1 2 2 3 1 2
- Pk==Pj ≠ = ≠ = ≠ = ≠
- next[j] 0 1 1 2 2 3 1 2
- Nextval[j] 0 1 0 2 1 3 0 2

运用KMP算法的匹配过程

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

× next(2) = 1

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　next(1)=0

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

×next(6) = 3

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

√

# KMP算法

```
int Index_KMP(SString S, SString T, int *next){        int
  i,j;
  i=1;  j=1;
  while (i<=S[0] && j<=T[0]){
        if (j==0 || S[i]==T[j]){++i;++j;}
        else j=next[j];
  }
  if (j>T[0]) return i-T[0];
  else return 0;
}
```

# 求next数组的步骤

(1) $next[1]=0$

$i=1;\ j=0;$

(2) 设 $next[i]=j$

若 $j=0$, $next[i+1]=1$

若 $P_i=P_j$, $next[i+1]=j+1=next[i]+1$

若 $P_i \neq P_j$, $next[i+1]=next[j]+1$

参看教材p82~83递推过程

# 求next数组的函数

```
void get_next(SString S, int *next){
  int i,j;
  i=1; next[1]=0;    j=0;
  while (i<S[0]) {
        if (j==0 || S[i]==S[j]) {++i; ++j; next[i]=j;}
        else j=next[j];
  }
}
```

# 改进的求next数组方法

设next[i]=j

若P[i]==P[j], 则nextval[i]=nextval[j]

# 改进的求next数组的函数

```
void get_nextval(SString S, int *nextval) {
   int i,j;
   i=1;  nextval[1]=0;  j=0;
   while (i<S[0])  {
         if (j==0 || S[i]==S[j]) {
               ++i;++j;
               if (S[i]!=S[j]) nextval[i]=j;
               else nextval[i]=nextval[j];
         }
         else j=nextval[j];
   }
}
```

- 穷举的模式匹配算法时间代价：
  最坏情况比较n-m+1趟，每趟比较m次，总比较次数达(n-m+1)*m
- 原因在于每趟重新比较时，目标串的检测指针要回退。改进的模式匹配算法可使目标串的检测指针每趟不回退。
- 改进的模式匹配(KMP)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
  - 求next函数的比较次数为m, 所以总的时间复杂度是O(n+m)