* 串即字符串
* 是计算机非数值处理的主要对象之一

**4.1串的类型定义**

* 串（String，或称字符串）是n个字符的有限序列。通常记作s=”a1,a2...an”(n>=0)
* S是串的名
* 串中字符的数目n称为串的长度
* 含零个字符的串称为空串（null string）
* 由一个或多个空格组成的串称为空格串（blank string），用符号“Φ”表示“空格串”。

**4.1.1串的抽象数据类型**

ADT String{  
 数据对象：D＝{ai|ai∈CharacterSet,i=1,2,...,n,n≥0}  
 数据关系：R1＝{<ai-1,ai>|ai-1,ai∈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。  
  
 StrCopy(&T,S)  
 初始条件：串S存在。  
 操作结果：由串S复制得串T。

DestroyString(&S)  
 初始条件：串S存在。  
 操作结果：串S被销毁。  
  
 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。  
  
 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。

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

串赋值StrAssign、串复制StrCopy、串比较StrCompare、求串长StrLength、串联接Concat以及求子串SubString等6种操作构成串类型的最小操作子集。

**应用**

例如，可利用判等、求串长和求子串等操作实现串的定位

函数Index(S,T,pos)和串的置换操作Replace(S,T,V)。

int Index(String S,String T,int pos)  
{  
 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  
 }  
 }  
 return 0; //S中不存在满足条件的子串  
}

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  
 }  
 }  
 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  
}

**4.2串的表示和实现**

**4.2.1串的定长顺序存储表示**

* 用一组地址连续的存储单元存储串值的字符序列
* #define MAXSTRLEN 255
* Typedef unsigned char SString[MAXSTRLEN]

void Concat(char S1[],char S2[],char T[])  
{  
 //以T返回由S1和S2联接而成的新串  
 j=0;k=0;  
 while(S1[j]!='\0')T[k++]=S1[j++]; //复制串S1  
 j=0;  
 while(S2[j]!='\0')T[k++]=S2[j++]; //紧接着复制串S2  
 T[k]='\0'; //置结果串T的结束标志  
}

bool SubString(char Sub[],char 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  
 slen=StrLength(S); //求串S的长度  
 if(pos<1||pos>slen||len<0||len>slen-pos+1)  
 return FALSE;  
 for(j=0;j<len;j++) Sub[j]=S[pos+j-1];  
 //向子串Sub复制字符  
 Sub[len]='\0'; //置串Sub的结束标志  
 return TRUE;  
}

**4.2.1串的堆分配存储表示**

* 串的堆分配存储的特点是，程序中出现的所有串变量的存储空间都是在程序执行过程中动态分配而得的。
* 堆分配存储结构的串定义如下：

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

bool 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  
 char S1[S.length]; //S1作为辅助串空间用于暂存S.ch  
 if(T.length)  
 { //T非空，则为S重新分配空间并插入T  
 p=S.ch;i=0;  
 while(i<S.length)  
 S1[i++]=\*(p+i); //暂存串S  
 S.ch=new char[S.length+T.length]; //为S重新分配串值存储空间

for(i=0,k=0;i<pos-1;i++)  
 S.ch[k++]=S1[i]; //保留插入位置之前的子串  
 j=0;  
 while(j<T.length)  
 S.ch[k++]=T.ch[j++]; //插入T  
 while(i<S.length)  
 S.ch[k++]=S1[i++]; //复制插入位置之后的子串  
 S.length+=T.length; //置串S的长度  
} //if  
return TRUE;

}

typedef struct{

char\*str;

int length;

} STRING;

（1）串的赋值

int StringAssign(STRING\*s,char\*string\_constant)

{

if(s->str)free(s->str);

//若s已经存在，将它占据的空间释放掉

for(len=0,ch=string\_constant;ch;len++,ch++);

//求string\_constant串的长度

if(!len) {

s->str=(char\*)malloc(sizeof(char));

s->str[0]=’\0’;s->length=0;

} //空串

else{

s->str=(char\*)malloc((len+1)\*sizeof(char));

//分配空间

if(!s->str)return ERROR;

s->str[0..len]=string\_constant[0..len];

//对应的字符赋值

s->length=len;

//赋予字符串长度

}

return OK;

}

（2）判断串是否为空

int StringEmpty(STRING s)

{

if(!s.length) return TRUE;

else return FALSE;

}

（3）求串的长度

int Length(STRING s)

{

return s.length;

}

（4）串连接

int Concat(STRING\*s1,STRING s2)

{

STRING s;

StringAssign(&s,s1->str);

//将s1原来的内容保留在s中

len=Length(s1)+Length(s2);

//计算s1和s2的长度之和

free(s1->str);

//释放s1原来占据的空间

s1->str=(char\*)malloc((len+1)\*sizeof(char));

//重新为s1分配空间

if(!s1)return ERROR;

else{ //连接两个串的内容

s1->str[0..Length(s)-1]=s.str[0..Length(s)-1)];

s1->str[Length(s)..len+1]=s2.str[0..Length(s2)];

s1->length=len;

free(s->str); //释放为临时串s分配的空间

return OK;

}

}

（5）求子串

int SubStr(STRING\*s1,STRING s2,int start,int len)

{

len2=Length(s2); //计算s2的长度

if(start<1||start>len2||len2<=0||len>len2-start+1){

//判断start和len的合理性

s1->str=(char\*)malloc(sizeof(char));

s1->str[0]=’\0’;s1->length=0;return ERROR;} //if

s1->str=(char\*)malloc((len+1)\*sizeof(char));

if(!s1.str)return ERROR;

s1->str[0..len-1]=s2.str[start-1..start+len-2];

s1->str[len]=’\0’;

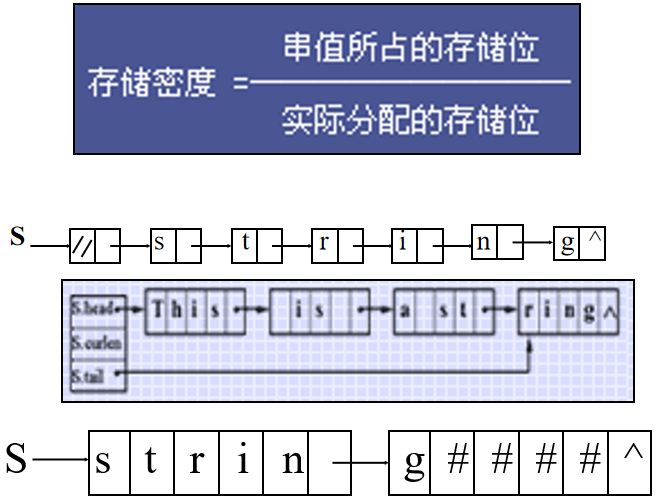
s1->length=len;

return OK;

}

**4.2.2串的块链存储表示**

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;  


**4.3匹配模式**

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。

**串的模式匹配的简单算法**

int Index\_BF(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+j]!=‘\0’&&T[j]!=‘\0’)  
 if(S[i+j]==T[j]) j++; //继续比较后一字符  
 else{i++;j=0;} //重新开始新的一轮匹配  
 if(T[j]=='\0') return(i+1); //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  
}

**串的模式匹配的改进算法**

按此算法进行模式串T="abcac"和主串S="ababcabcabcacabca"在pos=0的情况

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  
}

改进后的

**4.4KMP算法**

* 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**
* s1s2...si-1 si…sn
* p1p2…pj-1pj…pm
* si≠pj此时主串的i应该与模式串的第k个字符相比较,即
* p1p2…pk-1=si-k+1si-k+2...si-1而
* pj-k+1pj-k+2…pj-1=si-k+1si-k+2...si-1所以
* p1p2…pk-1=pj-k+1pj-k+2…pj-1

0 当j=1

Next[j]= Max{k|1<k<j且p1p2…pk-1= pj-k+1pj-k+2…pj-1 } 当集合不空

1. 其它情况

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

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;j=1;  
 while(i<=S[0]&&j<=T[0])  
 {

if(j==0||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  
 }  
 if(j>T[0]) return(i-T[0]); //匹配成功  
 else return 0;  
}

**求S的逆序**

void String\_Reverse(Stringtype s,Stringtype&r)  
{  
 StrAssign(r,''); //初始化r为空串  
 for(i=Strlen(s);i;i--)  
 {  
 StrAssign(c,SubString(s,i,1));  
 StrAssign(r,Concat(r,c)); //把s的字符从后往前添加到r中  
 }  
}

从串S中删除所有与t相同的子串，并返回删除次数

int Delete\_SubString(Stringtype&s,Stringtype t)  
{  
 for(n=0,i=1;i<=Strlen(s)-Strlen(t)+1;i++)  
 if(!StrCompare(SubString(s,i,Strlen(t)),t))  
 {  
 StrAssign(head,SubString(S,1,i-1));  
 StrAssign(tail,SubString(S,i+Strlen(t),Strlen(s)-i-Strlen(t)+1));  
 StrAssign(S,Concat(head,tail)); //把head,tail连接为新串  
 n++;  
 }  
 return n,  
}

将串S中所有子串T替换为V，并返回置换次数

int Replace(Stringtype&S,Stringtype T,Stringtype V)  
{  
 for(n=0,i=1;i<=Strlen(S)-Strlen(T)+1;i++) //注意i的取值范围  
 if(!StrCompare(SubString(S,i,Strlen(T)),T))  
 { //分别把T的前面和后面部分保存为head和tail  
 StrAssign(head,SubString(S,1,i-1));  
 StrAssign(tail,SubString(S,i+Strlen(T),Strlen(S)-i-Strlen(T)+1));  
 StrAssign(S,Concat(head,V));  
 StrAssign(S,Concat(S,tail)); //把head,V,tail连接为新串  
 i+=Strlen(V); //当前指针跳到插入串以后  
 n++;  
 }  
 return n;  
}