软件技术基础

第六讲

上讲主要内容

- 队列
- 链队列
- 循环队列

本讲主要内容

串类型定义

串的表示和实现

串

定义

串是由0个或多个字符组成的有限序列。一般记为 S="a_1 a_2…a_n"。

- S是串名；
- 用两个双引号括起来的字符序列是串值；
- a_i可以是字母、数字或是其他字符；
- 串所包含的字符个数成为该串的长度。

长度为零的串为空串"Ø"

子串和主串

- 串中任意个连续的字符组成的子序列称为该串的子串。
- 包含子串的串成为主串。
- 子串的第一个字符是在主串中的序号，定义为子串在子串中的位置（或序号）。

特别的，空串是任意串的子串，任意串是其自身的子串 。

子串和主串

- 串相等两个串的长度相等，且对应位置的字符相等

以“串的整体”为操作对象

串举例

例

两个串A和B

A=‘ This is a string’

B=‘string’

它们的长度分别为16和6

B是A的子串，在A中的位置是11

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

定长顺序存储表示

用地址连续的存储单元存放串值。
静态分配存储空间，用定长数组描述
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

顺序串的类型描述如下

```
# define  MAXSTRLEN 255
typedef  unsigned char
      SString[MAXSTRLEN
      +1];
```

✓以下标为0的数组分量存放串的实际长度；
✓在串值后加入"\0"表示结束，此时串长为隐含值

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

堆分配存储的串的基本操作

- 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

Status StrAssign(HString &S,char *chars)

```
//生成一个值等于chars的串S
{    int i,j;  char *c;
     for (i=0,c=chars;*c;++i,++c);
     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 ClearString(HString &S)

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

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;
}
```

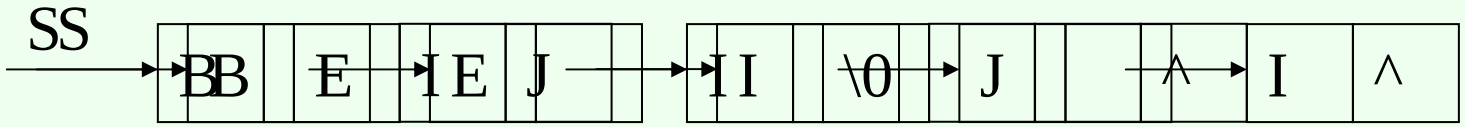
Status StrInsert(HString &S,int pos,HString T)

```
//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;
}
```

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
- ✓ 存储密度=串值所占的存储位/实际分配的存储位

例：S="BEIJING"



（a）结点大小为1的链表

（b）结点大小为4的链表

串 的块链存储表示

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

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

[词汇]

在模式匹配中，子串称为**模式**，串称为**目标**。

[例]

目标 S：“Beijing”

模式 P：“jin”

匹配结果 = 4

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：$s_i \neq p_j$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)

第1趟

S a b b a b a

P a b a

第2趟

S a b b a b a

P a b a

第3趟

S a b b a b a

P a b a

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;
}
```

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

S s_1 … s_{i-j-1} s_{i-j} s_{i-j+1} s_{i-j+2} … s_{i-1} s_i s_{i+1} … s_n

‖ ‖ ‖ ‖ ×

P p_1 p_2 … p_{j-1} p_j

则有 s_{i-j+1} s_{i-j+2} … s_{i-k+1} … s_{k-1} … s_{i-1}

‖ ‖ ‖ ‖ ‖ ‖ ‖ ‖

p_1 p_2 … $p_{j-(k-1)}$ … p_{j-k+1} … p_{j-1}

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

设因该与模式串中的第k（$k<j$）个字符比较，则须满足

$p_1\, p_2 \cdots p_{k-1} = s_{i-k+1}\, s_{i-k+2} \cdots s_{i-1}$

则 $p_1 \cdots p_{k-1} = p_{j-(k-1)} \cdots p_{j-1}$ (2)

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：

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

根据定义求next数组

序号j	1	2	3	4	5	6	7	8
模式P	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	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

√

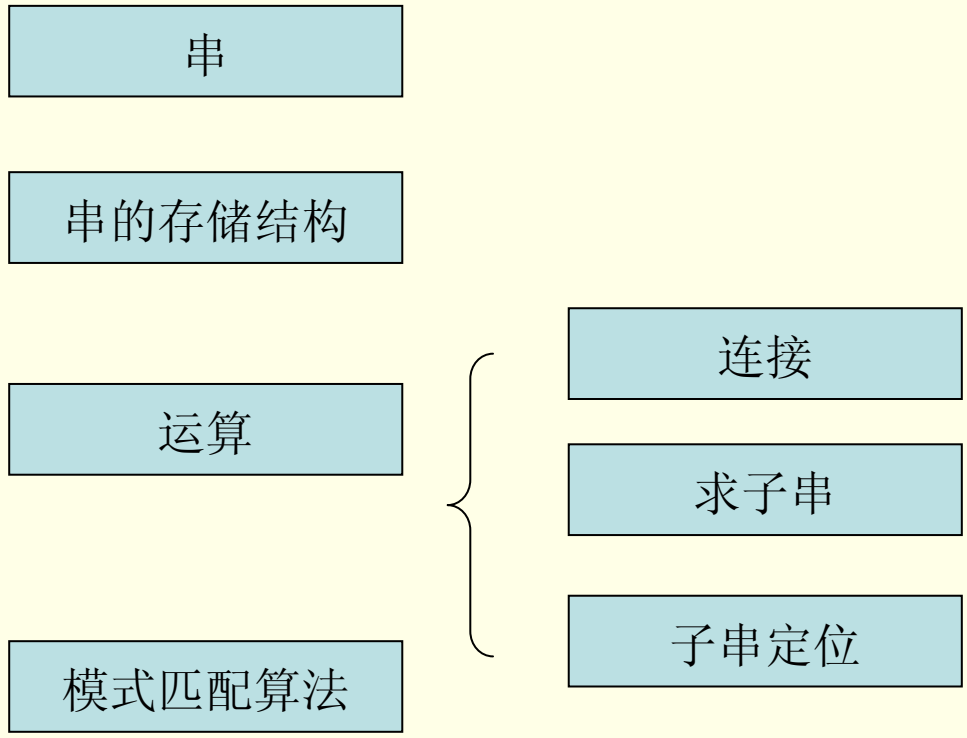
KMP算法

```
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];
   }
}
```

小结



作业