电话：15229238541

算法之一：

对两个多项式链表进行相加，生成一个新的相加后的结果多项式链表，原来两个多项式链表依然存在，不发生任何改变，如果要再对原来两个多项式进行其它操作也不影响。

QQ：871729782

算法描述

```
Ploy *add_ploy(ploy *La， ploy *Lb)
  /* 将以La，Lb为头指针表示的一元多项式相加，生成一个新的结果多项式 */
  {  ploy *Lc,*pc,*pa,*pb,*p;  float x;
     Lc=pc=(ploy *)malloc(sizeof(ploy));
     pa=La->next; pb=Lb->next;
     while (pa!=NULL&&pb!=NULL)
       { if (pa->expn<pb->expn)
           { p=(ploy *)malloc(sizeof(ploy));
             p->coef=pa->coef; p->expn=pa->expn;
```

```
        /* 生成一个新的结果结点并赋值 */
        pc->next=p；pc=p；pa=pa->next；
    }
    /* 生成的结点插入到结果链表的最后，pa指向下一个结点 */
if（pa->expn>pb->expn)
  {  p=(ploy *)malloc(sizeof(ploy))；
     p->coef=pb->coef；p->expn=pb->expn；
     p->next=NULL；
        /* 生成一个新的结果结点并赋值 */
      pc->next=p；pc=p；pb=pb->next；
    }   /* 生成的结点插入到结果链表的最后，pb
```

```
if (pa->expn==pb->expn)
   {  x=pa->coef+pb->coef ;
      if  (abs(x)<=1.0e-6)
            /*  系数和为0，pa, pb分别直接后继结点 */
            {  pa=pa->next ; pb=pb->next ;  }
      else    /* 若系数和不为0，生成的结点插入到结果链表的最后， pa, pb分别直接后继结点  */
            {  p=(ploy *)malloc(sizeof(ploy)) ;
               p->coef=x ; p->expn=pb->expn ;
```

```
            }
        }
    }    /* end of while */
 if  (pb!=NULL)
    while(pb!=NULL)
      {   p=(ploy *)malloc(sizeof(ploy)) ;
          p->coef=pb->coef ; p->expn=pb-
      >expn ;
          p->next=NULL ;
              /*  生成一个新的结果结点并赋值  */
            pc->next=p ; pc=p ; pb=pb-
      >next ;
      }
```

电话：15229238541

```
  if (pa!=NULL)
     while(pa!=NULL)
       {   p=(ploy *)malloc(sizeof(ploy)) ;
           p->coef=pb->coef ; p->expn=pa-
       >expn ;
           p->next=NULL ;
                /*  生成一个新的结果结点并赋值  */
             pc->next=p ; pc=p ; pa=pa-
       >next ;
       }
   return (Lc) ;
}
```


QQ：871729782

习题二

1　简述下列术语：线性表，顺序表，链表。

2　何时选用顺序表，何时选用链表作为线性表的存储结构合适?各自的主要优缺点是什么?

3　在顺序表中插入和删除一个结点平均需要移动多少个结点?具体的移动次数取决于哪两个因素?

4　链表所表示的元素是否有序?如有序，则有序性体现于何处?链表所表示的元素是否一定要在物理上是相邻的?有序表的有序性又如何理解?

5　设顺序表L是递增有序表，试写一算法，将x插入到L中并使L仍是递增有序表。

6　写一求单链表的结点数目ListLength(L)的算法。

7　写一算法将单链表中值重复的结点删除，使所得的结果链表中所有结点的值均不相同。

8　写一算法从一给定的向量A删除值在x到y(x≤y)之间的所有元素(注意：x和y是给定的参数，可以和表中的元素相同，也可以不同)。

9　设A和B是两个按元素值递增有序的单链表，写一算法将A和B归并为按按元素值递减有序的单链表C，试分析算法的时间复杂度。

第3章 栈和队列

栈和队列是两种应用非常广泛的数据结构，它们都来自线性表数据结构，都是“操作受限”的线性表。

栈在计算机的实现有多种方式：

- 硬堆栈：利用CPU中的某些寄存器组或类似的硬件或使用内存的特殊区域来实现。这类堆栈容量有限，但速度很快；
- 软堆栈：这类堆栈主要在内存中实现。堆栈容量可以达到很大。在实现方式上，又有动态方式和静态方式两种。

本章将讨论栈和队列的基本概念、存储结构、基本操作以及这些操作的具体实现。

3.1 栈

3.1.1 栈的基本概念

1 栈的概念

栈(Stack)：是限制在表的一端进行插入和删除操作的线性表。又称为后进先出LIFO (Last In First Out)或先进后出FILO (First In Last Out)线性表。

栈顶(Top)：允许进行插入、删除操作的一端，又称为表尾。用栈顶指针(top)来指示栈顶元素。

栈底(Bottom)：是固定端，又称为表头。

空栈：当表中没有元素时称为空栈。

设栈$S=(a_1, a_2, \ldots a_n)$，则a_1称为栈底元素，a_n为栈顶元素，如图3-1所示。

栈中元素按$a_1, a_2, \ldots a_n$的次序进栈，退栈的第一个元素应为栈顶元素。即栈的修改是按后进先出的原则进行的。

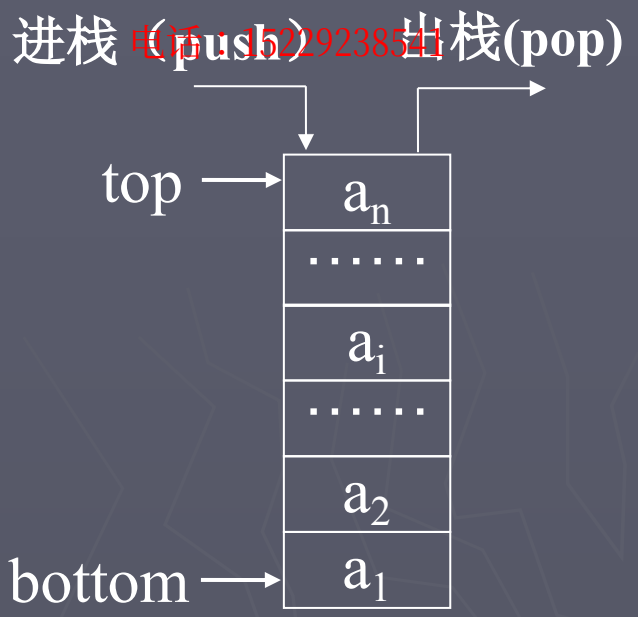


图3-1　顺序栈示意图

2　栈的抽象数据类型定义

ADT Stack{

数据对象：D ={ $a_i | a_i \in$ ElemSet, i=1,2,…,n， n≥0 }

数据关系：R ={$<a_{i-1}, a_i> | a_{i-1}$， $a_i \in D$, i=2,3,…,n }

基本操作：初始化、进栈、出栈、取栈顶元素等

} ADT Stack

3.1.2 栈的顺序存储表示

栈的顺序存储结构简称为顺序栈，和线性表相类似，用一维数组来存储栈。根据数组是否可以根据需要增大，又可分为静态顺序栈和动态顺序栈。

- 静态顺序栈实现简单，但不能根据需要增大栈的存储空间；
- 动态顺序栈可以根据需要增大栈的存储空间，但实现稍为复杂。

3.1.2.1 栈的动态顺序存储表示

采用动态一维数组来存储栈。所谓动态，指的是栈的大小可以根据需要增加。

- 用bottom表示栈底指针，栈底固定不变的；栈顶则随着进栈和退栈操作而变化。用top（称为栈顶指针）指示当前栈顶位置。
- 用top=bottom作为栈空的标记，每次top指向栈顶数组中的下一个存储位置。
- 结点进栈：首先将数据元素保存到栈顶(top所指的当前位置)，然后执行top加1，使top指向栈顶的下一个存储位置;

◆ **结点出栈**：首先执行top减1，使top指向栈顶元素的存储位置，然后将栈顶元素取出。

图3-2是一个动态栈的变化示意图。

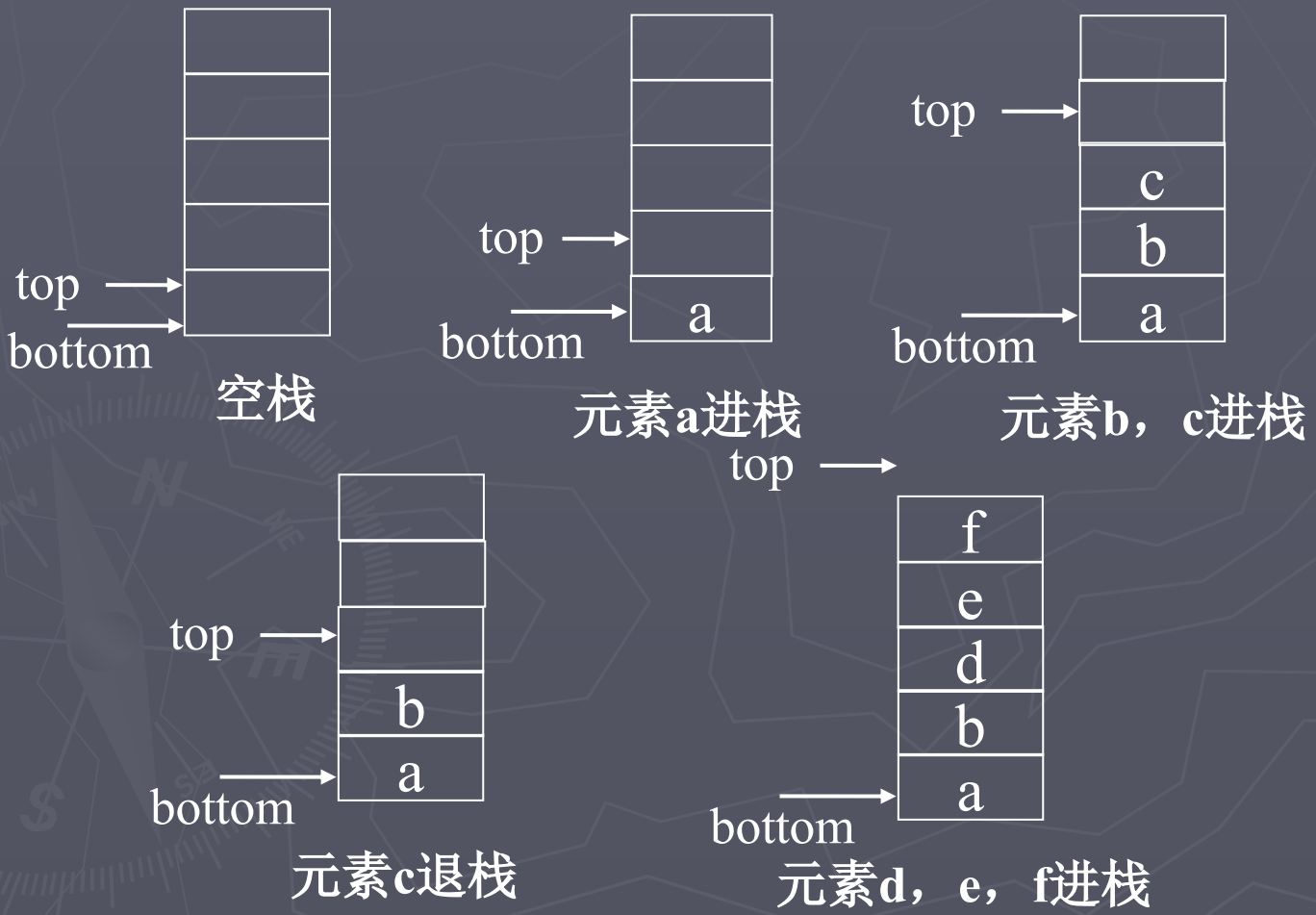


图3-2 (动态)堆栈变化示意图

基本操作的实现

1 栈的类型定义

```
#define  STACK_SIZE  100    /* 栈初始向量大小 */
#define STACKINCREMENT 10   /* 存储空间分配增量 */
#typedef  int  ElemType ;
typedef struct sqstack
   {   ElemType  *bottom;     /* 栈不存在时值为NULL */
       ElemType  *top;        /* 栈顶指针 */
       int   stacksize ;      /* 当前已分配空间，以元素为单位 */
   }SqStack ;
```

2 栈的初始化

```
Status Init_Stack(void)
  {  SqStack  S ;
     S.bottom=(ElemType *)malloc(STACK_SIZE
     *sizeof(ElemType));
     if (! S.bottom) return  ERROR;
     S.top=S.bottom ;    /*  栈空时栈顶和栈底指针相同
     */
     S. stacksize=STACK_SIZE;
     return OK ;
  }
```

3 压栈(元素进栈)

```
Status push(SqStack S , ElemType  e)
   {  if  (S.top-S.bottom>=S. stacksize-1)
        {   S.bottom=(ElemType *)realloc((S.
        STACKINCREMENT+STACK_SIZE)
        *sizeof(ElemType));   /* 栈满，追加存储空间
        */
          if (! S.bottom)  return  ERROR;
          S.top=S.bottom+S. stacksize ;
          S. stacksize+=STACKINCREMENT ;
        }
     *S.top=e;  S.top++ ; /*  栈顶指针加1，e成为新的栈顶
     */
```

电话：15229238541

4 弹栈(元素出栈)

```
Status pop( SqStack   S, ElemType  *e )
   /*弹出栈顶元素*/
   {   if ( S.top== S.bottom )
          return ERROR ;     /*  栈空，返回失败标志  */
      S.top-- ; e=*S. top ;
      return  OK ;
   }
```


QQ：871729782

3.1.2.2 栈的静态顺序存储表示

采用静态一维数组来存储栈。

栈底固定不变的，而栈顶则随着进栈和退栈操作变化的，

- 栈底固定不变的；栈顶则随着进栈和退栈操作而变化，用一个整型变量top（称为栈顶指针）来指示当前栈顶位置。
- 用top=0表示栈空的初始状态，每次top指向栈顶在数组中的存储位置。
- 结点进栈：首先执行top加1，使top指向新的栈顶位置，然后将数据元素保存到栈顶(top所指的当前位置)。

◆ 结点出栈：首先把top指向的栈顶元素取出，然后执行top减1，使top指向新的栈顶位置。

若栈的数组有Maxsize个元素，则top=Maxsize-1时栈满。图3-3是一个大小为5的栈的变化示意图。

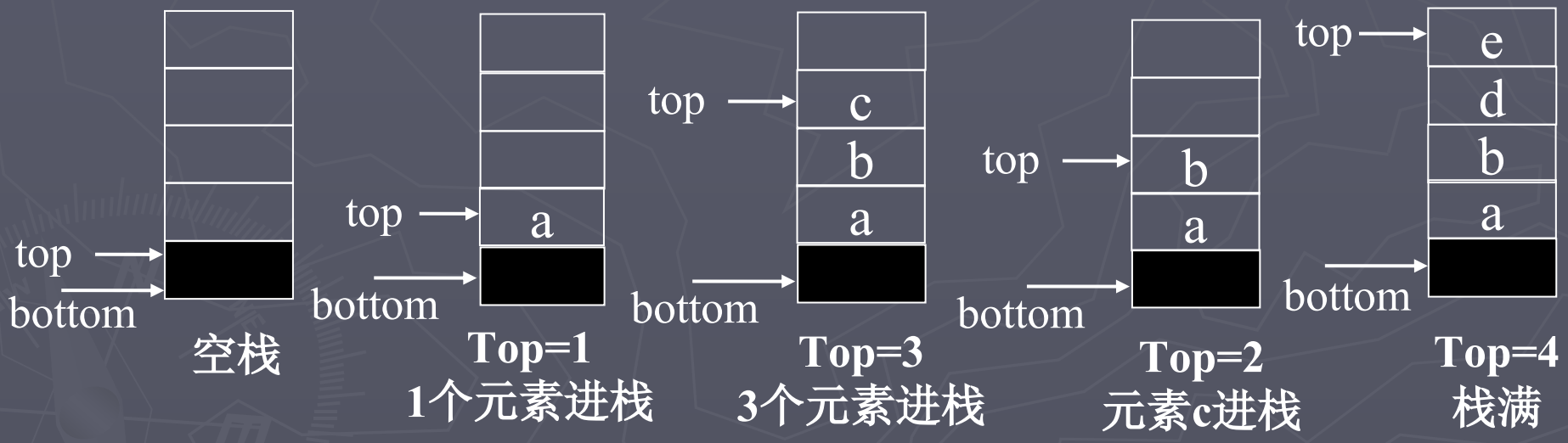


图3-3 静态堆栈变化示意图

基本操作的实现

1　栈的类型定义

```
# define  MAX_STACK_SIZE  100      /*  栈向量大小 */
# typedef  int  ElemType ;
typedef struct  sqstack
   {  ElemType
   stack_array[MAX_STACK_SIZE] ;
      int  top;
   }SqStack ;
```

2 栈的初始化

```
SqStack Init_Stack(void)
  {   SqStack  S ;
    S.bottom=S.top=0 ;  return(S) ;
  }
```

3 压栈(元素进栈)

```
Status push(SqStack S , ElemType  e)
   /*   使数据元素e进栈成为新的栈顶   */
  {  if  (S.top==MAX_STACK_SIZE-1)
        return  ERROR;      /* 栈满，返回错误标志    */
     S.top++ ;      /*   栈顶指针加1  */
     S.stack_array[S.top]=e ;  /* e成为新的栈顶
     */
     return OK;          /* 压栈成功    */
  }
```

4 弹栈(元素出栈)

```
Status  pop( SqStack   S, ElemType  *e )
    /*弹出栈顶元素*/
  {  if ( S.top==0 )
        return ERROR ;      /* 栈空，返回错误标志   */
     *e=S.stack_array[S.top] ;
     S.top-- ;
     return OK ;
  }
```

当栈满时做进栈运算必定产生空间溢出，简称“上溢”。上溢是一种出错状态，应设法避免。

当栈空时做退栈运算也将产生溢出，简称“下溢”。下溢则可能是正常现象，因为栈在使用时，其初态或终态都是空栈，所以下溢常用来作为控制转移的条件。

3.1.3 栈的链式存储表示

1 栈的链式表示

栈的链式存储结构称为链栈，是运算受限的单链表。其插入和删除操作只能在表头位置上进行。因此，链栈没有必要像单链表那样附加头结点，栈顶指针top就是链表的头指针。图3-4是栈的链式存储表示形式。

链栈的结点类型说明如下：

```
typedef  struct  Stack_Node
   {  ElemType   data ;
      struct Stack_Node  *next ;
   } Stack_Node ;
```

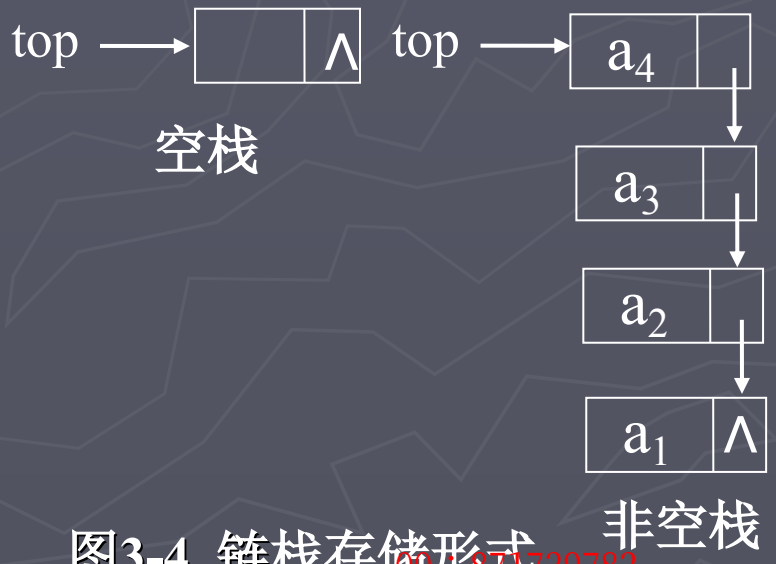


图3-4 链栈存储形式

2 链栈基本操作的实现

(1) 栈的初始化

```
Stack_Node  *Init_Link_Stack(void)
  {   Stack_Node  *top ;
     top=(Stack_Node
     *)malloc(sizeof(Stack_Node )) ;
     top->next=NULL ;
     return(top) ;
  }
```

(2) 压栈(元素进栈)

```
Status push(Stack_Node *top , ElemType  e)
  {  Stack_Node  *p ;
     p=(Stack_Node
     *)malloc(sizeof(Stack_Node)) ;
     if (!p)  return  ERROR;
        /*  申请新结点失败，返回错误标志 */
     p->data=e ;
     p->next=top->next  ;
     top->next=p ;   /*  钩链  */
     return OK;
  }
```

(3) 弹栈(元素出栈)

```
Status pop(Stack_Node  *top , ElemType *e)
  /*  将栈顶元素出栈  */
  {  Stack_Node  *p ;
     ElemType  e ;
     if  (top->next==NULL )
        return ERROR ;   /* 栈空，返回错误标志  */
     p=top->next ; e=p->data ;   /*  取栈顶元素
     */
     top->next=p->next ;    /*  修改栈顶指针  */
     free(p) ;
     return OK ;
  }
```

3.2 栈的应用

由于栈具有的“后进先出”的固有特性，因此，栈成为程序设计中常用的工具和数据结构。以下是几个栈应用的例子。

3.2.1 数制转换

十进制整数N向其它进制数d(二、八、十六)的转换是计算机实现计算的基本问题。

转换法则：该转换法则对应于一个简单算法原理：

n=(n div d)*d+n mod d

其中：div为整除运算, mod为求余运算

例如 $(1348)_{10}= (2504)_8$，其运算过程如下：

n	n div 8	n mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

电话：15229238541

采用静态顺序栈方式实现

```
void conversion(int  n , int  d)
  /*将十进制整数N转换为d(2或8)进制数*/
  {  SqStack  S ;    int k, *e ;
     S=Init_Stack();
     while  (n>0)  {  k=n%d ; push(S , k) ; n=n/d ;   }
       /*  求出所有的余数，进栈  */
     while  (S.top!=0)     /*  栈不空时出栈，输出  */
        {   pop(S, e) ;
            printf(“%1d” , *e) ;
        }
  }
```


QQ：871729782

3.2.2 括号匹配问题

在文字处理软件或编译程序设计时，常常需要检查一个字符串或一个表达式中的括号是否相匹配?

匹配思想：从左至右扫描一个字符串(或表达式)，则每个右括号将与最近遇到的那个左括号相匹配。则可以在从左至右扫描过程中把所遇到的左括号存放到堆栈中。每当遇到一个右括号时，就将它与栈顶的左括号(如果存在)相匹配，同时从栈顶删除该左括号。

算法思想：设置一个栈，当读到左括号时，左括号进栈。当读到右括号时，则从栈中弹出一个元素，与读到的左括号进行匹配，若匹配成功，继续读入；否则匹配失败，返回FLASE。

电话：15229238541

算法描述

```
#define TRUE  0
#define  FLASE  -1
SqStack  S ;
S=Init_Stack() ;  /*堆栈初始化*/
int Match_Brackets( )
   {   char ch , x ;
       scanf(“%c” , &ch) ;
        while (asc(ch)!=13)
```


QQ：871729782

```
{  if ((ch=='(')||(ch=='['))  push(S, ch);
   else if  (ch==']')
      {  x=pop(S);
         if (x!='[')
               {  printf("'['括号不匹配");
                  return FLASE ; } }
   else if  (ch==')')
      {  x=pop(S);
         if (x!='(')
            { printf("'('括号不匹配");
              return FLASE ;}
      }
```

```
    if (S.top!=0)
      {   printf("括号数量不匹配！");
          return FLASE ;
      }
    else  return TRUE ;
}
```

3.2.2 栈与递归调用的实现

栈的另一个重要应用是在程序设计语言中实现递归调用。

递归调用：一个函数(或过程)直接或间接地调用自己本身，简称**递归**(**Recursive**)。

递归是程序设计中的一个强有力的工具。因为递归函数结构清晰，程序易读，正确性很容易得到证明。

为了使递归调用不至于无终止地进行下去，实际上有效的递归调用函数(或过程)应包括两部分：**递推规则(方法)**，**终止条件**。

例如：求n!

$$Fact(n)=\begin{cases}1 & \text{当n=0时} \quad \text{终止条件}\\ n*fact(n-1) & \text{当n>0时} \quad \text{递推规则}\end{cases}$$

为保证递归调用正确执行，系统设立一个“递归工作栈”，作为整个递归调用过程期间使用的数据存储区。

每一层递归包含的信息如：参数、局部变量、上一层的返回地址构成一个“工作记录”。每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。

从被调函数返回调用函数的一般步骤：

(1) 若栈为空，则执行正常返回。

(2) 从栈顶弹出一个工作记录。

(3) 将“工作记录”中的参数值、局部变量值赋给相应的变量；读取返回地址。

(4) 将函数值赋给相应的变量。

(5) 转移到返回地址。

3.3 队 列

3.3.1 队列及其基本概念

1 队列的基本概念

队列(Queue)：也是运算受限的线性表。是一种先进先出(First In First Out，简称FIFO)的线性表。只允许在表的一端进行插入，而在另一端进行删除。

队首(front)：允许进行删除的一端称为队首。

队尾(rear)：允许进行插入的一端称为队尾。

例如：排队购物。操作系统中的作业排队。先进入队列的成员总是先离开队列。

队列中没有元素时称为空队列。在空队列中依次加入元素$a_1, a_2, \ldots, a_n$之后，a_1是队首元素，a_n是队尾元素。显然退出队列的次序也只能是$a_1, a_2, \ldots, a_n$，即队列的修改是依先进先出的原则进行的，如图3-5所示。

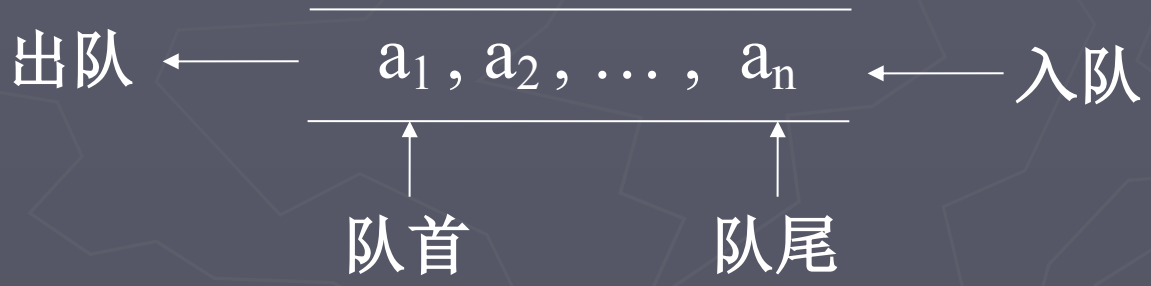


图3-5 队列示意图

2 队列的抽象数据类型定义

ADT Queue{

数据对象：D ={ a_i|a_i∈ElemSet, i=1, 2, …, n, n >= 0 }

数据关系：$R = \{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D,\ i=2,3,\ldots,n\}$

约定a_1端为队首，a_n端为队尾。

基本操作：

Create()：创建一个空队列；

EmptyQue()：若队列为空，则返回true，否则返回flase；

……

InsertQue(x)：向队尾插入元素x；

DeleteQue(x)：删除队首元素x；

} ADT Queue

3.3.2 队列的顺序表示和实现

利用一组连续的存储单元(一维数组) 依次存放从队首到队尾的各个元素，称为顺序队列。

对于队列，和顺序栈相类似，也有动态和静态之分。本部分介绍的是静态顺序队列，其类型定义如下：

```
#define  MAX_QUEUE_SIZE   100
typedef  struct  queue
   {  ElemType   Queue_array[MAX_QUEUE_SIZE] ;
      int   front ;
      int  rear ;
   }SqQueue;
```

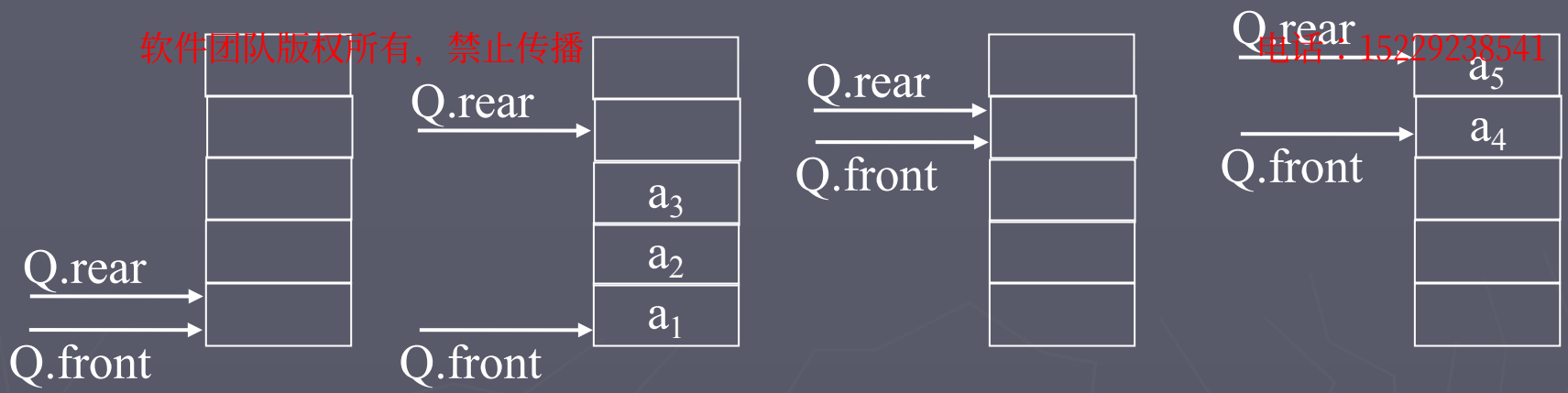
3.3.2.1 队列的顺序存储结构

设立一个队首指针front，一个队尾指针rear，分别指向队首和队尾元素。

- 初始化：front=rear=0。
- 入队：将新元素插入rear所指的位置，然后rear加1。
- 出队：删去front所指的元素，然后加1并返回被删元素。
- 队列为空：front=rear。
- 队满：rear=MAX_QUEUE_SIZE-1或front=rear。

在非空队列里，队首指针始终指向队头元素，而队尾指针始终指向队尾元素的下一位置。

顺序队列中存在“假溢出”现象。因为在入队和出队操作中，头、尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。因此，尽管队列中实际元素个数可能远远小于数组大小，但可能由于尾指针已超出向量空间的上界而不能做入队操作。该现象称为假溢出。如图3-6所示是数组大小为5的顺序队列中队首、队尾指针和队列中元素的变化情况。



(a) 空队列　(b) 入队3个元素　(c) 出队3个元素　(d) 入队2个元素

图3-6　队列示意图

3.3.2.2 循环队列

为充分利用向量空间，克服上述“假溢出”现象的方法是：将为队列分配的向量空间看成为一个首尾相接的圆环，并称这种队列为循环队列(Circular Queue)。

在循环队列中进行出队、入队操作时，队首、队尾指针仍要加1，朝前移动。只不过当队首、队尾指针指向向量上界(MAX_QUEUE_SIZE-1)时，其加1操作的结果是指向向量的下界0。

这种循环意义下的加1操作可以描述为：

```
if  (i+1==MAX_QUEUE_SIZE)   i=0;
else     i++ ;
```

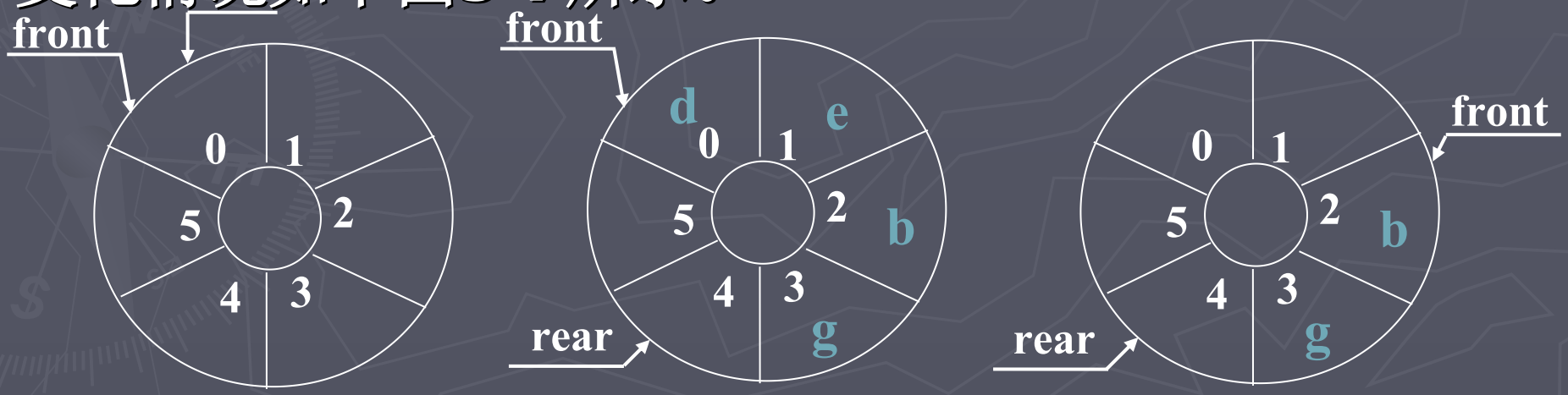
其中：i代表队首指针(front)或队尾指针(rear)

用模运算可简化为：

i=(i+1)%MAX_QUEUE_SIZE；

显然，为循环队列所分配的空间可以被充分利用，除非向量空间真的被队列元素全部占用，否则不会上溢。因此，真正实用的顺序队列是循环队列。

例：设有循环队列**QU[0，5]**，其初始状态是**front=rear=0**，各种操作后队列的头、尾指针的状态变化情况如下图**3-7**所示。



(a) 空队列　(b) d, e, b, g入队　(c) d, e出队

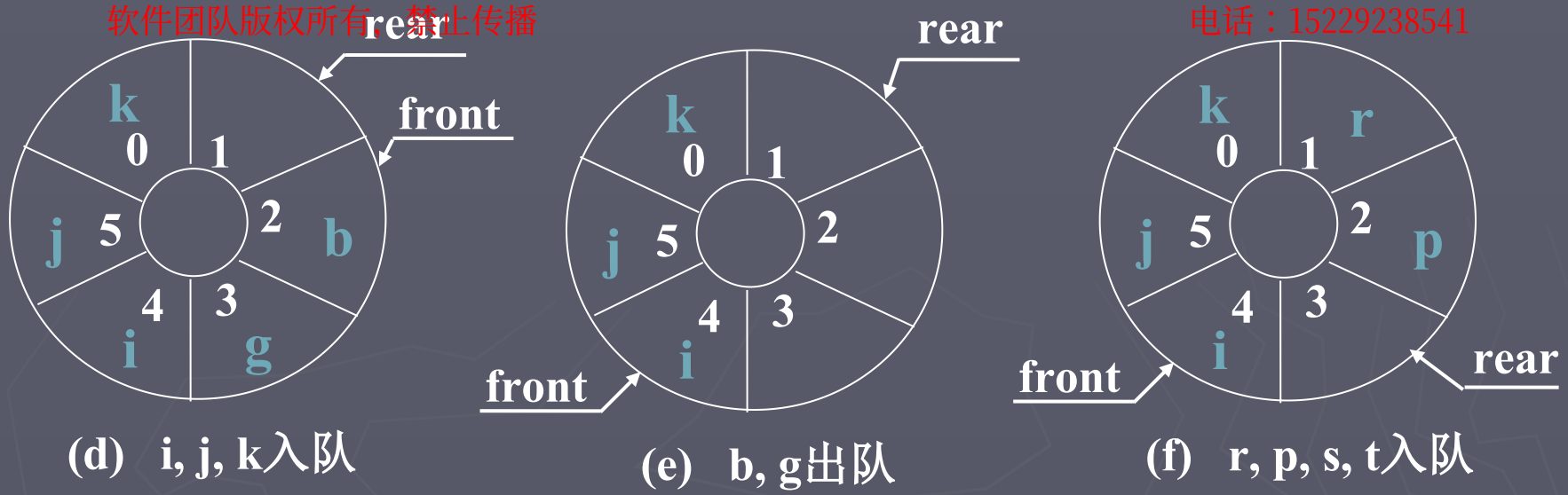


图3-7 循环队列操作及指针变化情况

入队时尾指针向前追赶头指针，出队时头指针向前追赶尾指针，故队空和队满时头尾指针均相等。因此，无法通过front=rear来判断队列“空”还是“满”。解决此问题的方法是：约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队满。即：

- rear所指的单元始终为空。

- 循环队列为空：front=rear 。
- 循环队列满：(rear+1)%MAX_QUEUE_SIZE =front。

循环队列的基本操作

1 循环队列的初始化

```
SqQueue Init_CirQueue(void)
  { SqQueue  Q ;
    Q.front=Q.rear=0;  return(Q) ;
  }
```

2 入队操作

```
Status Insert_CirQueue(SqQueue  Q , ElemType  e)
  /*   将数据元素e插入到循环队列Q的队尾   */
  {  if  ((Q.rear+1)%MAX_QUEUE_SIZE== Q.front)
     return  ERROR;     /*  队满，返回错误标志    */
     Q.Queue_array[Q.rear]=e ;   /*   元素e入队    */
     Q.rear=(Q.rear+1)% MAX_QUEUE_SIZE ;
          /*  队尾指针向前移动  */
     return OK;       /*  入队成功    */
  }
```

3 出队操作

```
Status Delete_CirQueue(SqQueue  Q, ElemType  *x )
    /*   将循环队列Q的队首元素出队   */
   {   if  (Q.front+1== Q.rear)
           return ERROR ;      /*  队空，返回错误标志    */
       *x=Q.Queue_array[Q.front] ;  /*  取队首元素  */
       Q.front=(Q.front+1)% MAX_QUEUE_SIZE ;
           /*  队首指针向前移动  */
       return OK ;
   }
```

3.3.3 队列的链式表示和实现

1 队列的链式存储表示

队列的链式存储结构简称为链队列，它是限制仅在表头进行删除操作和表尾进行插入操作的单链表。

需要两类不同的结点：数据元素结点，队列的队首指针和队尾指针的结点，如图3-8所示。

数据元素结点类型定义：

```
typedef struct Qnode
  { ElemType   data ;
    struct Qnode  *next ;
  }QNode ;
```

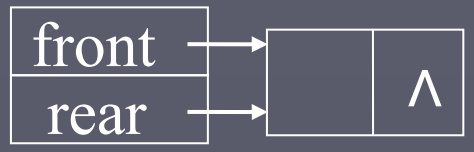
图3-8 链队列结点示意图

指针结点类型定义：

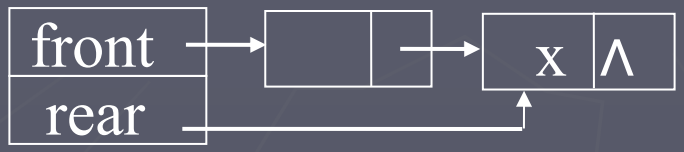
```
typedef struct link_queue
   {   QNode  *front , *rear ;
   }Link_Queue ;
```

2　链队运算及指针变化

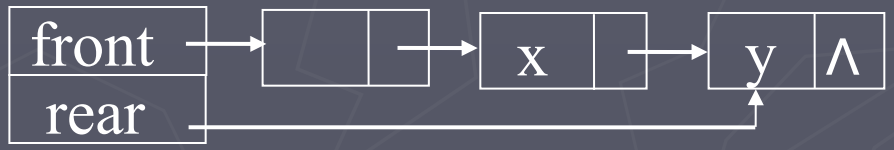
链队的操作实际上是单链表的操作，只不过是删除在表头进行，插入在表尾进行。插入、删除时分别修改不同的指针。链队运算及指针变化如图3-9所示。



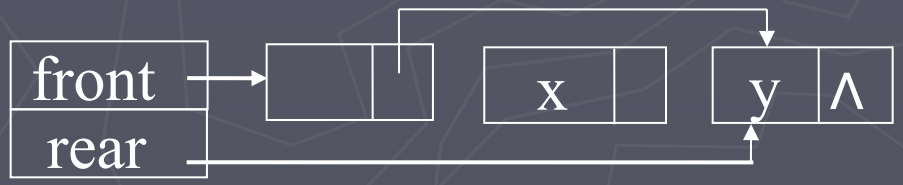
(a) 空队列



(b) x入队



(c) y再入队



(d) x出队

图3-9 队列操作及指针变化

3 链队列的基本操作

(1) 链队列的初始化

```
LinkQueue *Init_LinkQueue(void)
  { LinkQueue  *Q ;  QNode  *p ;
    p=(QNode *)malloc(sizeof(QNode)) ; /* 开辟头结点 */
    p->next=NULL ;
    Q=(LinkQueue  *)malloc(sizeof(LinkQueue)) ;
          /*  开辟链队的指针结点  */
    Q.front=Q.rear=p ;
    return(Q) ;
  }
```

(2) 链队列的入队操作

在已知队列的队尾插入一个元素e，即修改队尾指针(Q.rear)。

```
Status  Insert_CirQueue(LinkQueue  *Q , ElemType  e)
        /*   将数据元素e插入到链队列Q的队尾   */
  {   p=(QNode *)malloc(sizeof(QNode)) ;
      if (!p)  return  ERROR;
         /*  申请新结点失败，返回错误标志 */
      p->data=e ; p->next=NULL ;      /*  形成新结点 */
      Q.rear->next=p ;  Q.rear=p ;  /*  新结点插入到队尾  */
      return OK;
  }
```

(3) 链队列的出队操作

```
Status  Delete_LinkQueue(LinkQueue  *Q, ElemType *x)
    {   QNode *p ;
        if  (Q.front==Q.rear)  return ERROR ;   /*  队空  */
        p=Q.front->next ;   /*    取队首结点    */
        *x=p->data ;
        Q.front->next=p->next ;      /*  修改队首指针  */
        if  (p==Q.rear)  Q.rear=Q.front ;
                /*  当队列只有一个结点时应防止丢失队尾指针  */
          free(p) ;
        return OK ;
    }
```

电话：15229238541

(4) 链队列的撤消

```
void  Destroy_LinkQueue(LinkQueue  *Q )
  /*  将链队列Q的队首元素出队  */
   {   while  (Q.front!=NULL)
          {  Q.rear=Q.front->next;
                /*  令尾指针指向队列的第一个结点   */
             free(Q.front);     /*  每次释放一个结点  */
               /*  第一次是头结点，以后是元素结点  */
             Q.ront=Q.rear;
          }
   }
```


QQ：871729782

电话：15229238541

习 题 三

1 设有一个栈，元素进栈的次序为a, b, c。问经过栈操作后可以得到哪些输出序列？

2 循环队列的优点是什么?如何判断它的空和满?

3 设有一个静态顺序队列，向量大小为**MAX**，判断队列为空的条件是什么？队列满的条件是什么？

4 设有一个静态循环队列，向量大小为**MAX**，判断队列为空的条件是什么？队列满的条件是什么？

5 利用栈的基本操作，写一个返回栈**S**中结点个数的算法**int StackSize(SeqStack S)**，并说明S为何不作为指针参数的算法?

QQ：871729782

6　一个双向栈S是在同一向量空间内实现的两个栈，它们的栈底分别设在向量空间的两端。试为此双向栈设计初始化InitStack(S)，入栈Push(S,i,x)，出栈Pop(S,i,x)算法，其中i为0或1，用以表示栈号。

7　设Q[0,6]是一个静态顺序队列，初始状态为front=rear=0，请画出做完下列操作后队列的头尾指针的状态变化情况，若不能入对，请指出其元素，并说明理由。

a, b, c, d入队

a, b, c出队

i , j , k , l , m入队

d, i出队

n, o, p, q, r入队

8 假设Q[0,5]是一个循环队列，初始状态为front=rear=0，请画出做完下列操作后队列的头尾指针的状态变化情况，若不能入对，请指出其元素，并说明理由。

d, e, b, g, h入队

d, e出队

i , j , k , l , m入队

b出队

n, o, p, q, r入队

电话：15229238541

第4章 串

在非数值处理、事务处理等问题常涉及到一系列的字符操作。计算机的硬件结构主要是反映数值计算的要求，因此，字符串的处理比具体数值处理复杂。本章讨论串的存储结构及几种基本的处理。

鸣谢 软件学院研究生会
QQ：871729782

4.1 串类型的定义

4.1.1 串的基本概念

串(字符串)：是零个或多个字符组成的有限序列。记作： S="$a_1a_2a_3$…"，其中S是串名，$a_i(1 \leqq i \leqq n)$是单个，可以是字母、数字或其它字符。

串值：双引号括起来的字符序列是串值。

串长：串中所包含的字符个数称为该串的长度。

空串(空的字符串)：长度为零的串称为空串，它不包含任何字符。

空格串(空白串)：构成串的所有字符都是空格的串称为空白串。

注意：空串和空白串的不同，例如“ ”和“”分别表示长度为1的空白串和长度为0的空串。

子串（**substring**）：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。

子串的序号：将子串在主串中首次出现时的该子串的首字符对应在主串中的序号，称为子串在主串中的序号（或位置）。

例如，设有串A和B分别是：

A=“这是字符串”，B=“是”

则B是A的子串，A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是3。因此，称B在A中的序号为3 。

特别地，空串是任意串的子串，任意串是其自身的子串。

串相等：如果两个串的串值相等(相同)，称这两个串相等。换言之，只有当两个串的长度相等，且各个对应位置的字符都相同时才相等。

通常在程序中使用的串可分为两种：串变量和串常量。

串常量和整常数、实常数一样，在程序中只能被引用但不能不能改变其值，即只能读不能写。通常串常量是由直接量来表示的，例如语句错误(“溢出”)中“溢出”是直接量。

串变量和其它类型的变量一样，其值是可以改变。

4.1.2 串的抽象数据类型定义

ADT String{

数据对象： $D = \{ a_i | a_i \in CharacterSet,\ i=1,2,\ldots,n,\ n \geq 0 \}$

数据关系： $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D,\ i=2,3,\ldots,n \}$

基本操作：

StrAssign(t , chars)

初始条件： chars是一个字符串常量。

操作结果： 生成一个值为chars的串t 。

StrConcat(s, t)

初始条件： 串s, t 已存在。

操作结果：将串t联结到串s后形成新串存放到s中。

StrLength(t)

初始条件：字符串t已存在。

操作结果：返回串t中的元素个数，称为串长。

SubString (s, pos, len, sub)

初始条件：串s, 已存在, 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。

……

} ADT String

4.2 串的存储表示和实现

串是一种特殊的线性表，其存储表示和线性表类似，但又不完全相同。串的存储方式取决于将要对串所进行的操作。串在计算机中有3种表示方式：

- **定长顺序存储表示**：将串定义成字符数组，利用串名可以直接访问串值。用这种表示方式，串的存储空间在编译时确定，其大小不能改变。
- **堆分配存储方式**：仍然用一组地址连续的存储单元来依次存储串中的字符序列，但串的存储空间是在程序运行时根据串的实际长度动态分配的。
- **块链存储方式**：是一种链式存储结构表示。

4.2.1 串的定长顺序存储表示

这种存储结构又称为串的顺序存储结构。是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构，是直接使用定长的字符数组来定义，数组的上界预先确定。

定长顺序存储结构定义为：

```
#define MAX_STRLEN  256
typedef  struct
   {  char  str[MAX_STRLEN] ;
      int  length;
   } StringType ;
```

电话：15229238541

1 串的联结操作

```
Status  StrConcat ( StringType  s, StringType t)
   /*   将串t联结到串s之后，结果仍然保存在s中    */
   {  int i,  j ;
      if ((s.length+t.length)>MAX_STRLEN)
         Return ERROR ;   /*  联结后长度超出范围  */
      for (i=0 ; i<t.length ; i++)
         s.str[s.length+i]=t.str[i] ;   /*  串t联结到串s之后  */
      s.length=s.length+t.length ;  /* 修改联结后的串长度 */
      return OK ;
   }
```


QQ：871729782

2 求子串操作

```
Status SubString (StringType s, int pos, int len,
StringType *sub)
  {  int k,  j ;
     if (pos<1||pos>s.length||len<0||len>(s.length-pos+1))
        return ERROR ;   /*  参数非法  */
     sub->length=len-pos+1 ;   /*  求得子串长度  */
     for (j=0, k=pos ; k<=leng ; k++, j++)
        sub->str[j]=s.str[i] ;   /*  逐个字符复制求得子串  */
     return OK ;
  }
```

4.2.2 串的堆分配存储表示

实现方法：系统提供一个空间足够大且地址连续的存储空间(称为“堆”)供串使用。可使用C语言的动态存储分配函数malloc()和free()来管理。

特点是：仍然以一组地址连续的存储空间来存储字符串值，但其所需的存储空间是在程序执行过程中动态分配，故是动态的，变长的。

串的堆式存储结构的类型定义

```
typedef  struct
   {  char *ch;    /*  若非空，按长度分配，否则为NULL */
      int length;      /*  串的长度  */
   } HString;
```

1 串的联结操作

```
Status  Hstring  *StrConcat(HString  *T, HString *s1,
HString *s2)
   /*  用T返回由s1和s2联结而成的串  */
   {   int k,  j , t_len ;
       if (T.ch)  free(T);      /*  释放旧空间   */
       t_len=s1->length+s2->length ;
       if ((p=(char *)malloc(sizeof((char)*t_len))==NULL)
          {   printf("系统空间不够，申请空间失败！\n") ;
              return ERROR  ;    }
       for (j=0 ; j<s->length; j++)
          T->ch[j]=s1->ch[j] ;    /*  将串s复制到串T中 */
```

```
    for (k=s1->length, j=0 ; j<s2->length; k++, j++)
        T->ch[j]=s1->ch[j] ;    /* 将串s2复制到串T中 */
    free(s1->ch) ;
    free(s2->ch) ;
    return OK ;
}
```

4.2.3 串的链式存储表示

串的链式存储结构和线性表的串的链式存储结构类似，采用单链表来存储串，结点的构成是：

- data域：存放字符，data域可存放的字符个数称为结点的大小；
- next域：存放指向下一结点的指针。

若每个结点仅存放一个字符，则结点的指针域就非常多，造成系统空间浪费，为节省存储空间，考虑串结构的特殊性，使每个结点存放若干个字符，这种结构称为块链结构。如图4-1是块大小为3的串的块链式存储结构示意图。

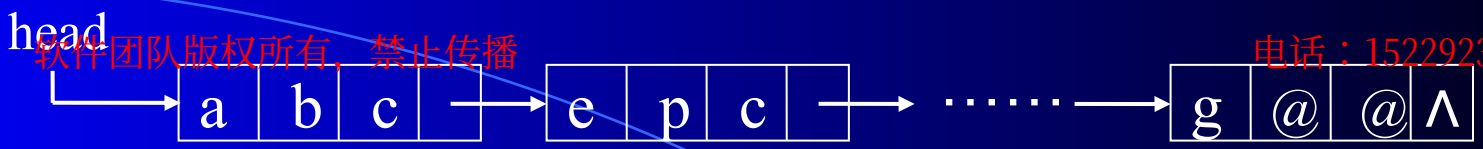


图4-1　串的块链式存储结构示意图

串的块链式存储的类型定义包括：

(1) 块结点的类型定义

```
#define BLOCK_SIZE  4
typedef  struct  Blstrtype
   {  char  data[BLOCK_SIZE] ;
      struct  Blstrtype  *next;
   }BNODE ;
```

电话：15229238541

(2) 块链串的类型定义

```
typedef  struct
  {  BNODE  head;     /*  头指针  */
     int  Strlen ;       /*  当前长度  */
  } Blstring ;
```

在这种存储结构下，结点的分配总是完整的结点为单位，因此，为使一个串能存放在整数个结点中，在串的末尾填上不属于串值的特殊字符，以表示串的终结。

当一个块(结点)内存放多个字符时，往往会使操作过程变得较为复杂，如在串中插入或删除字符操作时通常需要在块间移动字符。

4.3 串的模式匹配算法

模式匹配(模范匹配)：子串在主串中的定位称为模式匹配或串匹配(字符串匹配）。模式匹配成功是指在主串S中能够找到模式串T，否则，称模式串T在主串S中不存在。

模式匹配的应用在非常广泛。例如，在文本编辑程序中，我们经常要查找某一特定单词在文本中出现的位置。显然，解此问题的有效算法能极大地提高文本编辑程序的响应性能。

模式匹配是一个较为复杂的串操作过程。迄今为止，人们对串的模式匹配提出了许多思想和效率各不相同的计算机算法。介绍两种主要的模式匹配算法。

4.3.1 Brute-Force模式匹配算法

设S为目标串，T为模式串，且不妨设：

S="$s_0s_1s_2\ldots s_{n-1}$"，　T="$t_0t_1t_2\ \ldots t_{m-1}$"

串的匹配实际上是对合法的位置$0 \leqq i \leqq n-m$依次将目标串中的子串$s[i\ldots i+m-1]$和模式串$t[0\ldots m-1]$进行比较：

- 若$s[i\ldots i+m-1]=t[0\ldots m-1]$：则称从位置i开始的匹配成功，亦称模式t在目标s中出现；
- 若$s[i\ldots i+m-1]\neq t[0\ldots m-1]$：从i开始的匹配失败。

位置i称为位移，当$s[i\ldots i+m-1]=t[0\ldots m-1]$时，i称为有效位移；当$s[i\ldots i+m-1]\neq t[0\ldots m-1]$时，i称为无效位移。

这样，串匹配问题可简化为找出某给定模式T在给定目标串S中首次出现的有效位移。

算法实现

```
int IndexString(StringType  s , StringType  t , int pos )
  /*  采用顺序存储方式存储主串s和模式t，    */
  /*  若模式t在主串s中从第pos位置开始有匹配的子串，  */
  /*  返回位置，否则返回-1   */
   {  char *p , *q ;
      int  k , j ;
      k=pos-1 ;   j=0 ; p=s.str+pos-1 ; q=t.str ;
         /*  初始匹配位置设置  */
          /*  顺序存放时第pos位置的下标值为pos-1 */
```

电话：15229238541

```
while (k<s.length)&&(j<t.length)
    {  if (*p==*q)   { p++ ; q++ ; k++ ; j++ ; }
       else { k=k-j+1 ; j=0 ; q=t.str ; p=s.str+k ;  }
       /*  重新设置匹配位置    */
       }
 if (j==t.length)
     return(k-t.length) ;   / *   匹配，返回位置    */
 else return(-1) ;      /*   不匹配，返回-1   */
}
```


QQ：871729782

该算法简单，易于理解。在一些场合的应用里，如文字处理中的文本编辑，其效率较高。

该算法的时间复杂度为O(n*m)，其中n、m分别是主串和模式串的长度。通常情况下，实际运行过程中，该算法的执行时间近似于O(n+m)。

理解该算法的关键点

当第一次$s_k \neq t_j$时：主串要退回到k-j+1的位置，而模式串也要退回到第一个字符（即j=0的位置）。

比较出现$s_k \neq t_j$时：则应该有$s_{k-1}=t_{j-1}$，…，$s_{k-j+1}=t_1$，$s_{k-j}=t_0$。

4.3.2 模式匹配的一种改进算法

该改进算法是由D.E.Knuth，J.H.Morris和V.R.Pratt提出来的，简称为KMP算法。其改进在于：

每当一趟匹配过程出现字符不相等时，主串指示器不用回溯，而是利用已经得到的“部分匹配”结果，将模式串的指示器向右“滑动”尽可能远的一段距离后，继续进行比较。

例：设有串s=“abacabab”，t=“abab”。则第一次匹配过程如图4-2所示。

```
s=“a  b  cbb”      i=3
    ‖  ‖ ≠                  匹配失败
t=“a  b  b”        j=3
```

图4-2 模式匹配示例

在i=3和j=3时，匹配失败。但重新开始第二次匹配时，不必从i=1，j=0开始。因为$s_1=t_1$，$t_0\neq t_1$，必有$s_1\neq t_0$，又因为$t_0=t_2$，$s_2=t_2$，所以必有$s_2=t_0$。由此可知，第二次匹配可以直接从i=3、j=1开始。

总之，在主串s与模式串t的匹配过程中，一旦出现$s_i\neq t_j$，主串s的指针不必回溯，而是直接与模式串的t_k($0\leqq k<j$进行比较，而k的取值与主串s无关，只与模式串t本身的构成有关，即从模式串t可求得k值。)

不失一般性，设主串s=“$s_1s_2\ldots s_n$”，模式串t=“$t_1t_2\ldots t_m$”。

当$s_i \neq t_j$($1 \leqslant i \leqslant n-m$，$1 \leqslant j < m$，$m < n$)时，主串s的指针i不必回溯，而模式串t的指针j回溯到第k(k<j)个字符继续比较，则模式串t的前k-1个字符必须满足4-1式，而且不可能存在k’>k满足4-1式。

$$t_1t_2\ldots t_{k-1} = s_{i-(k-1)}\ s_{i-(k-2)}\ \cdots\ s_{i-2}\ s_{i-1} \qquad (4\text{-}1)$$

而已经得到的“部分匹配”的结果为：

$$t_{j-(k-1)}\ t_{j-k}\cdots\ t_{j-1} = s_{i-(k-1)}\ s_{i-(k-2)}\ \cdots\ s_{i-2}\ s_{i-1} \qquad (4\text{-}2)$$

由式(4-1)和式(4-2)得：

$$t_1t_2\ldots t_{k-1} = t_{j-(k-1)}\ t_{j-k}\cdots\ t_{j-1} \qquad (4\text{-}3)$$

该推导过程可用图4-3形象描述。实际上，式(4-3)描述了模式串中存在相互重叠的子串的情况。

电话：15229238541

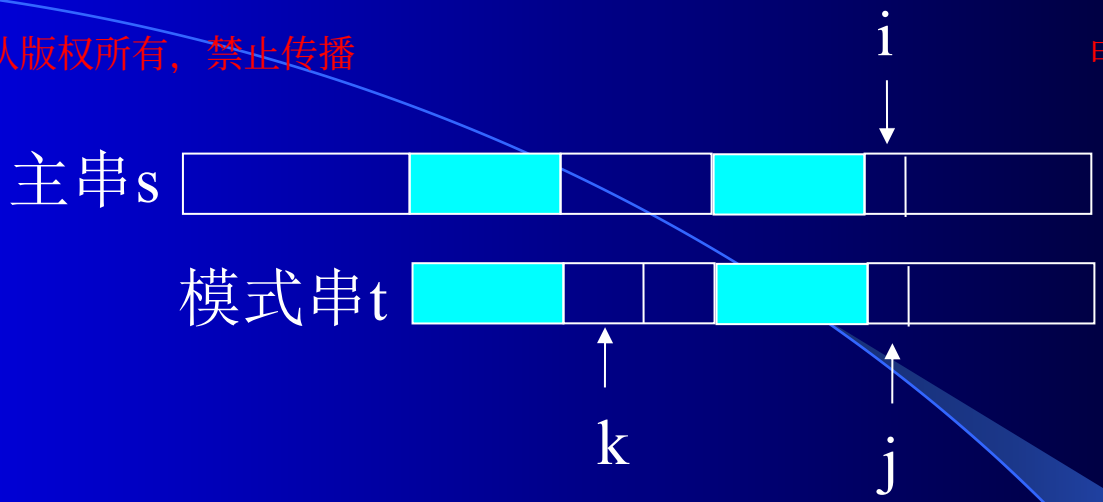


图4-3　KMP算法示例

定义next[j]函数为

$$next[j]=\begin{cases}0 & \text{当}j=1\text{时}\\ Max\{k\mid 1<k<j \wedge t_1t_2\ldots t_{k-1}=t_{j-(k-1)}\ t_{j-k}\ldots\ t_{j-1}\} & \text{该集合不空时}\\ 1 & \text{其它情况}\end{cases}$$

QQ：871729782

在求得了next[j]值之后，KMP算法的思想是：

设目标串(主串)为s，模式串为t，并设i指针和j指针分别指示目标串和模式串中正待比较的字符，设i和j的初值均为1。若有$s_i=t_j$，则i和j分别加1。否则，i不变，j退回到j=next[j]的位置，再比较s_i和t_j，若相等，则i和j分别加1。否则，i不变，j再次退回到j=next[j]的位置，依此类推。直到下列两种可能：

(1) j退回到某个下一个[j]值时字符比较相等，则指针各自加1继续进行匹配。

(2)退回到j=0，将i和j分别加1，即从主串的下一个字符s_{i+1}模式串的t_1重新开始匹配。

KMP算法如下

电话：15229238541

```
#define Max_Strlen 1024
int next[Max_Strlen];
int KMP_index (StringType  s , StringType  t)
  /* 用KMP算法进行模式匹配，匹配返回位置，否则返回-1 */
  /*用静态存储方式保存字符串， s和t分别表示主串和模式串  */
   {   int  k=0 , j=0 ;      /*初始匹配位置设置 */
       while (k<s.length)&&(j<t.length
          {  if ((j==-1)|| (s. str[k]==t.str[j]))  {  k++ ; j++ ; }
              else j=next[j] ;
          }
       if (j>= t.length)  return(k-t.length) ;
       else  return(-1) ;
   }
```


QQ：871729782

很显然，KMP_index函数是在已知下一个函数值的基础上执行的，以下讨论如何求next函数值?

由式(4-3)知，求模式串的next[j]值与主串s无关，只与模式串t本身的构成有关，则可把求next函数值的问题看成是一个模式匹配问题。由next函数定义可知：

当j=1时： next[1]=0。

设next[j]=k，即在模式串中存在：$t_1t_2\ldots t_{k-1}=t_{j-(k-1)}t_{j-k}\ldots t_{j-1}$，其中下标k满足1<k<j的某个最大值，此时求next[j+1]的值有两种可能：

(1) 若有$t_k=t_j$：则表明在模式串中有：

$t_1t_2\ldots t_{k-1}t_k=t_{j-(k-1)}t_{j-k}\ldots t_{j-1}t_j$，且不可能存在k'>k满足上式，即：next[j+1]=next[j]+1=k+1

(2) 若有$t_k \neq t_j$：则表明在模式串中有：$t_1\ t_2 \ldots t_{k-1}\ t_k \neq t_{j-(k-1)}\ t_{j-k} \ldots\ t_{j-1}\ t_j$，当$t_k \neq t_j$时应将模式向右滑动至以模式中的第next[k]个字符和主串中的第j个字符相比较。若next[k]= k’，且$t_j = t_{k'}$，则说明在主串中第j+1字符之前存在一个长度为k’(即next[k])的最长子串，与模式串中从第一个字符起长度为k’的子串相等。即　　next[j+1]=k’+1

同理，若$t_j \neq t_k$，应将模式继续向右滑动至将模式中的第next[k’]个字符和t_j对齐，……，依此类推，直到t_j和模式串中的某个字符匹配成功或者不存在任何k’(1< k’<j)满足等式：$t_1\ t_2 \ldots t_{k-1}\ t_{k'} = t_{j-(k'-1)}\ t_{j-k'} \ldots\ t_{j-1}\ t_j$

则：　next[j]+1=1

根据上述分析，求next函数值的算法如下：

```
void  next(StringType  t , int next[])
  /*  求模式t的next串t函数值并保存在next数组中   */
   {   int  k=1 , j=0 ; next[1]=0;
       while (k<t.length)
          {   if ((j==0)|| (t.str[k]==t.str[j]))
                 {   k++ ; j++ ;
                     if ( t.str[k]!=t.str[j] )  next[k]=j;
                     else  next[k]=next[j];
                 }
              else next[j]=j ;
          }
   }
```

电话：15229238541

习 题 四

(1) 解释下列每对术语的区别：空串和空白串；主串和子串；目标串和模式串。

(2) 若x和y是两个采用顺序结构存储的串，写一算法比较这两个字符串是否相等。

(3) 写一算法void StrRelace(char *T, char *P, char *S)，将T中第一次出现的与P相等的子串替换为S，串S和P的长度不一定相等，并分析时间复杂度。

QQ：871729782

第5章 数组和广义表

数组是一种人们非常熟悉的数据结构，几乎所有的程序设计语言都支持这种数据结构或将这种数据结构设定为语言的固有类型。数组这种数据结构可以看成是线性表的推广。

科学计算中涉及到大量的矩阵问题，在程序设计语言中一般都采用数组来存储，被描述成一个二维数组。但当矩阵规模很大且具有特殊结构(对角矩阵、三角矩阵、对称矩阵、稀疏矩阵等)，为减少程序的时间和空间需求，采用自定义的描述方式。

广义表是另一种推广形式的线性表，是一种灵活的数据结构，在许多方面有广泛的应用。

5.1 数组的定义

数组是一组偶对(下标值，数据元素值)的集合。在数组中，对于一组有意义的下标，都存在一个与其对应的值。一维数组对应着一个下标值，二维数组对应着两个下标值，如此类推。

数组是由n(n>1)个具有相同数据类型的数据元素a_1，a_2，…，a_n组成的有序序列，且该序列必须存储在一块地址连续的存储单元中。

- 数组中的数据元素具有相同数据类型。
- 数组是一种随机存取结构，给定一组下标，就可以访问与其对应的数据元素。
- 数组中的数据元素个数是固定的。

5.1.1 数组的抽象数据类型定义

1 抽象数据类型定义

ADT Array{

数据对象：$j_i= 0,1,\ldots,b_i-1 , 1,2, \ldots,n$;

D = { $a_{j_1j_2\ldots j_n}$ | n>0称为数组的维数，b_i是数组第i维的长度，j_i是数组元素第i维的下标，$a_{j_1j_2\ldots j_n}\in$ElemSet }

数据关系：$R = \{R_1, R_2, \ldots, R_n\}$

$R_i=\{<a_{j_1j_2\ldots j_i\ldots j_n}, a_{j_1j_2\ldots j_{i+1}\ldots j_n}>|0\leqq j_k\leqq b_k-1$ ，
$1\leqq k\leqq n$且$k\neq i$，$0\leqq j_i\leqq b_i-2$，$a_{j_1j_2\ldots j_{i+1}\ldots j_n}\in D\}$

基本操作： ……

} ADT Array

由上述定义知，n维数组中有$b_1 \times b_2 \times \ldots \times b_n$个数据元素，每个数据元素都受到n维关系的约束。

2 直观的n维数组

以二维数组为例讨论。将二维数组看成是一个定长的线性表，其每个元素又是一个定长的线性表。

设二维数组$A=(a_{ij})_{m\times n}$，则

$A=(\alpha_1, \alpha_2, \ldots, \alpha_p)$ (p=m或n)

其中每个数据元素α_j是一个列向量(线性表)：

$\alpha_j=(a_{1j}, a_{2j}, \ldots, a_{mj})$ $1 \leqslant j \leqslant n$

或是一个行向量：

$\alpha_i=(a_{i1}, a_{i2}, \ldots, a_{in})$ $1 \leqslant i \leqslant m$

如图3-1所示。

$$A=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

(a)　矩阵表示形式

$$A=\begin{pmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \end{bmatrix} \\ \begin{bmatrix} a_{21} & a_{22} & \cdots & a_{2n} \end{bmatrix} \\ \cdots \cdots \cdots \cdots \cdots \\ \begin{bmatrix} a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \end{pmatrix}$$

(b)　列向量的一维数组形式

$$A=\begin{pmatrix} \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} \cdots \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} \end{pmatrix}$$

(c)　行向量的一维数组形式

图5-1　二维数组图例形式

5.2 数组的顺序表示和实现

数组一般不做插入和删除操作，也就是说，数组一旦建立，结构中的元素个数和元素间的关系就不再发生变化。因此，一般都是采用顺序存储的方法来表示数组。

问题：计算机的内存结构是一维(线性)地址结构，对于多维数组，将其存放(映射)到内存一维结构时，有个次序约定问题。即必须按某种次序将数组元素排成一列序列，然后将这个线性序列存放到内存中。

二维数组是最简单的多维数组，以此为例说明多维数组存放(映射)到内存一维结构时的次序约定问题。