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，…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，…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。

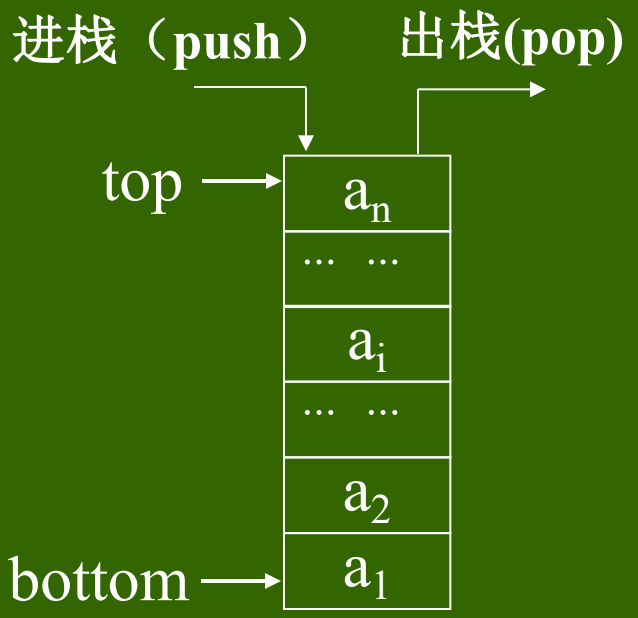


图3-1　顺序栈示意图

2　栈的抽象数据类型定义

ADT Stack{

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

数据关系：R ={<a_{i-1}, a_i>|a_{i-1}，a_i∈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。

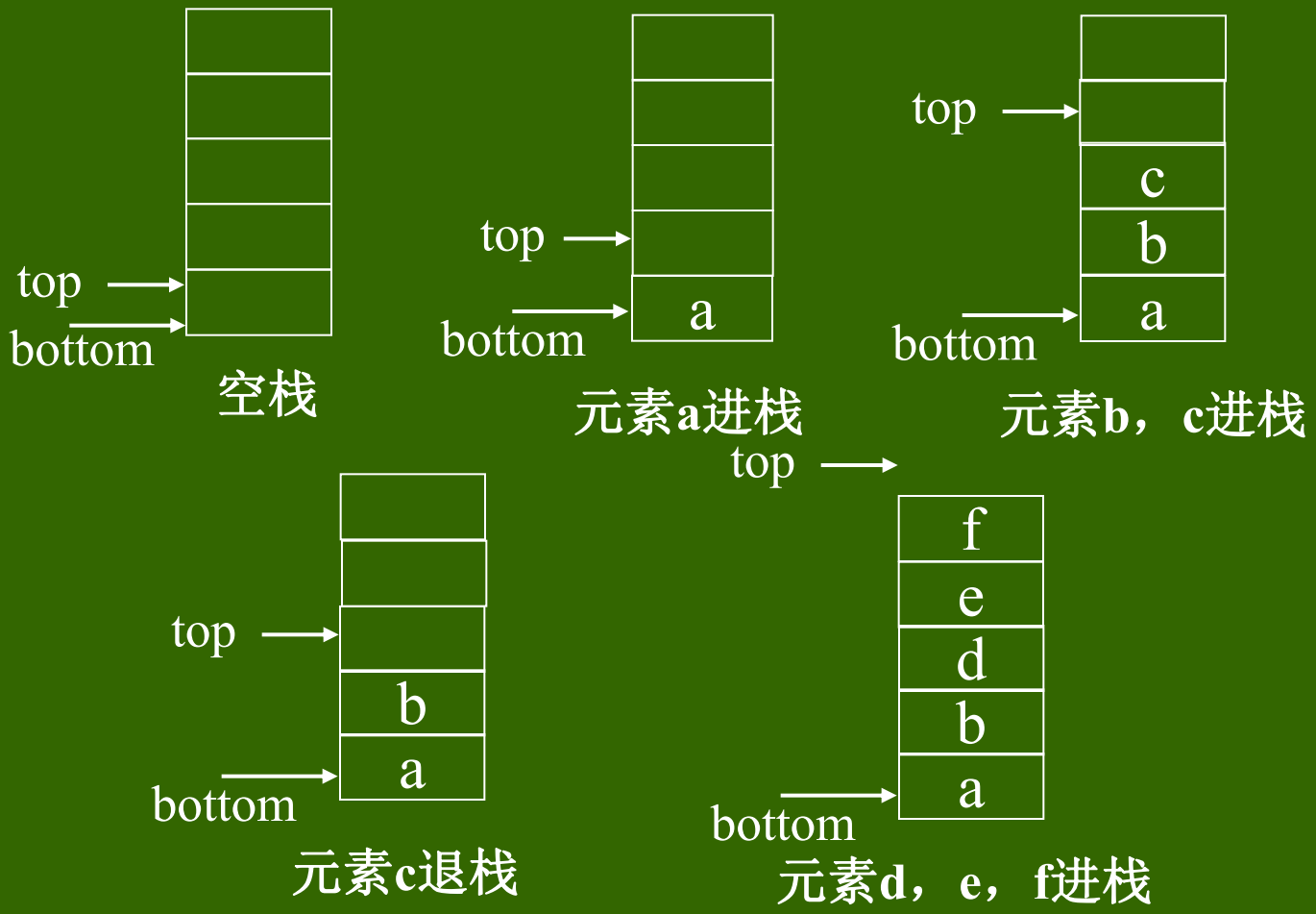


图3-2 (动态)堆栈变化示意图

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  */
      return OK;
```

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 ;
   }
```

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。

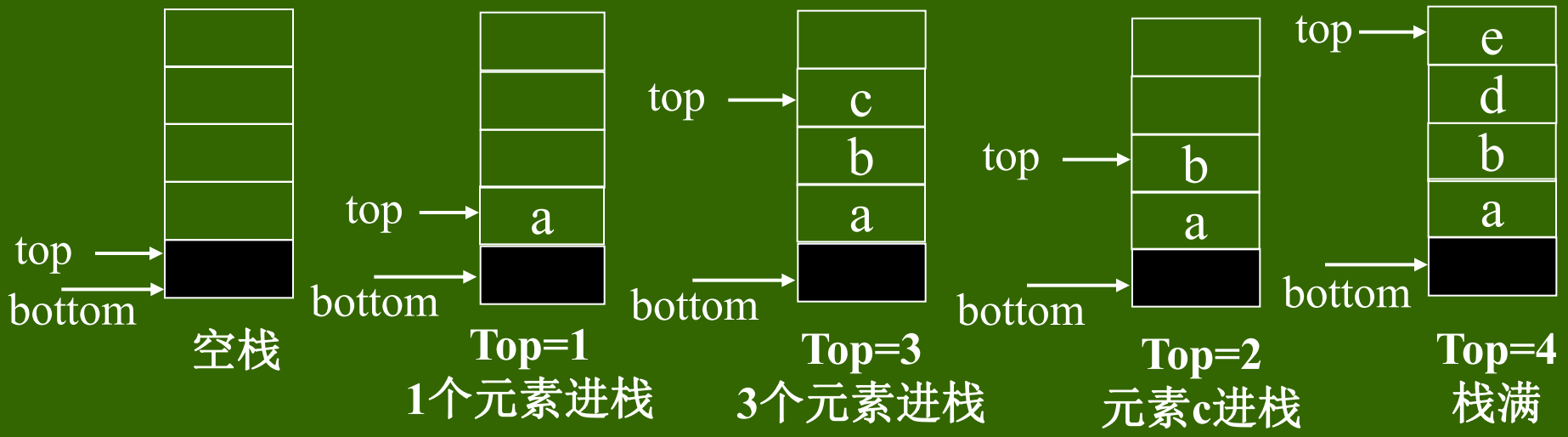


图3-3　静态堆栈变化示意图

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 ;
```

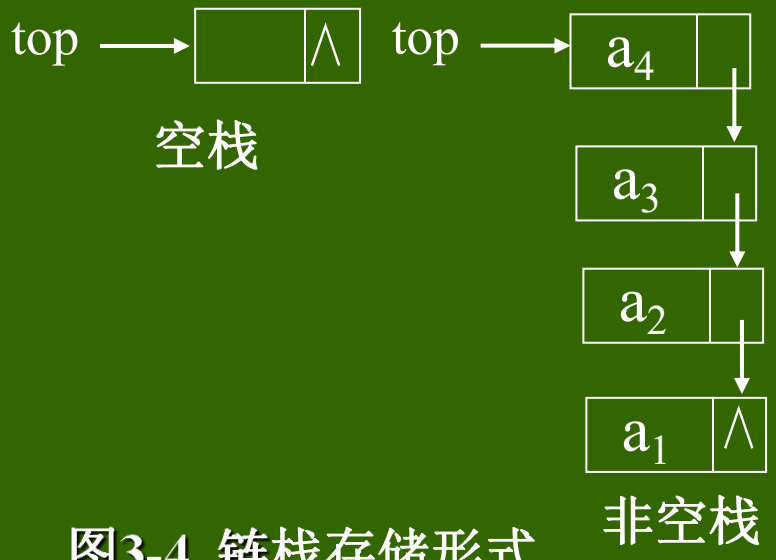


图3-4 链栈存储形式

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

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) ;
        }
  }
```

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。

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)
```

```
{   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。

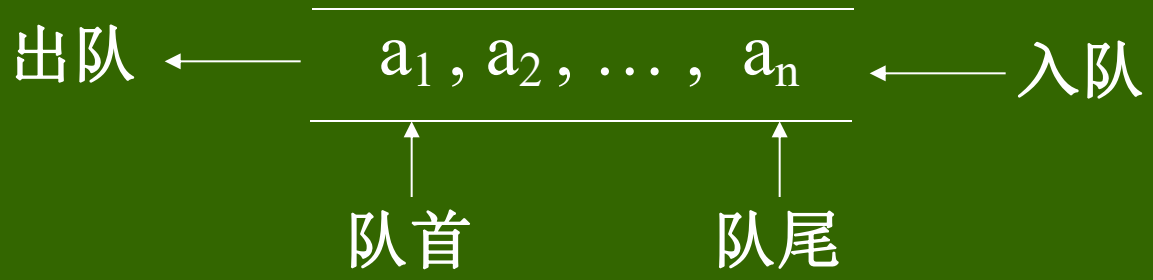


图3-5 队列示意图

2 队列的抽象数据类型定义

ADT Queue{

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

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;
```

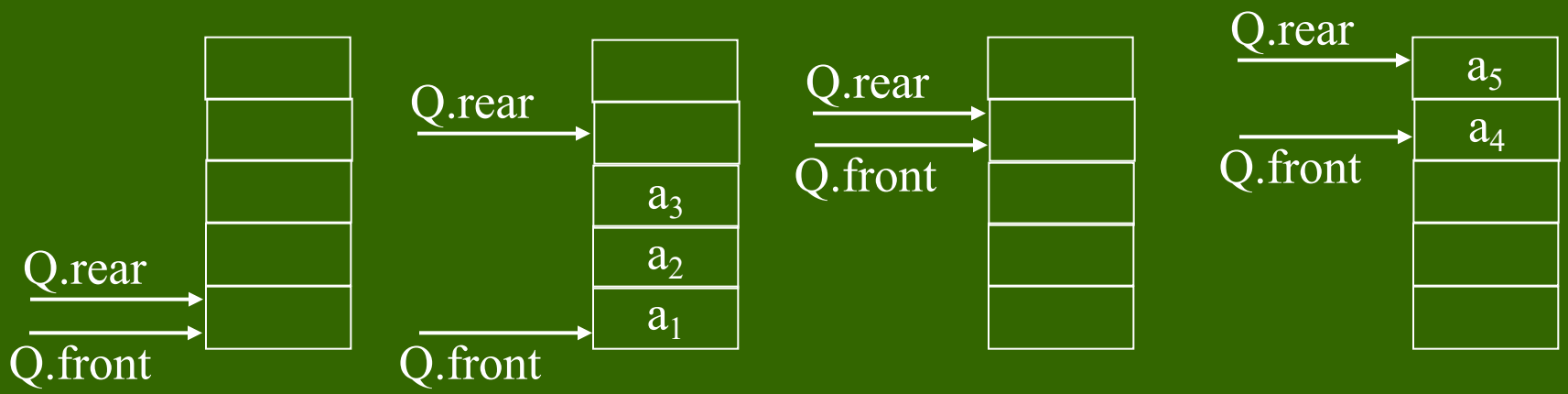
3.3.2.1 队列的顺序存储结构

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：

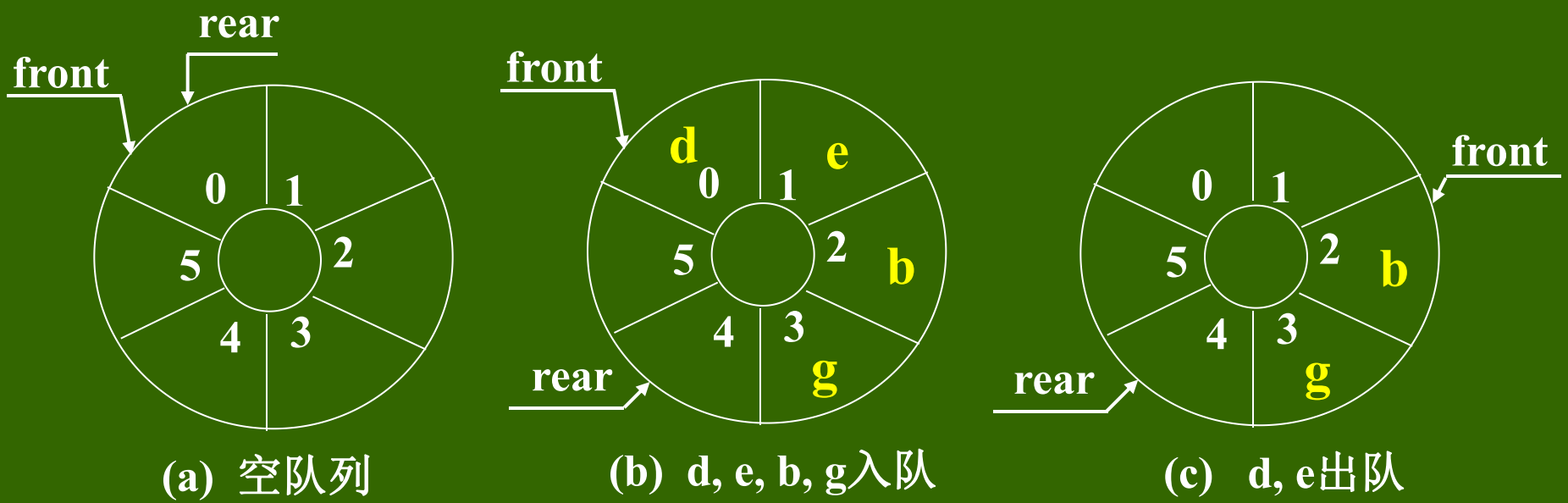
```
if (i+1==MAX_QUEUE_SIZE)  i=0;
else     i++ ;
```

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

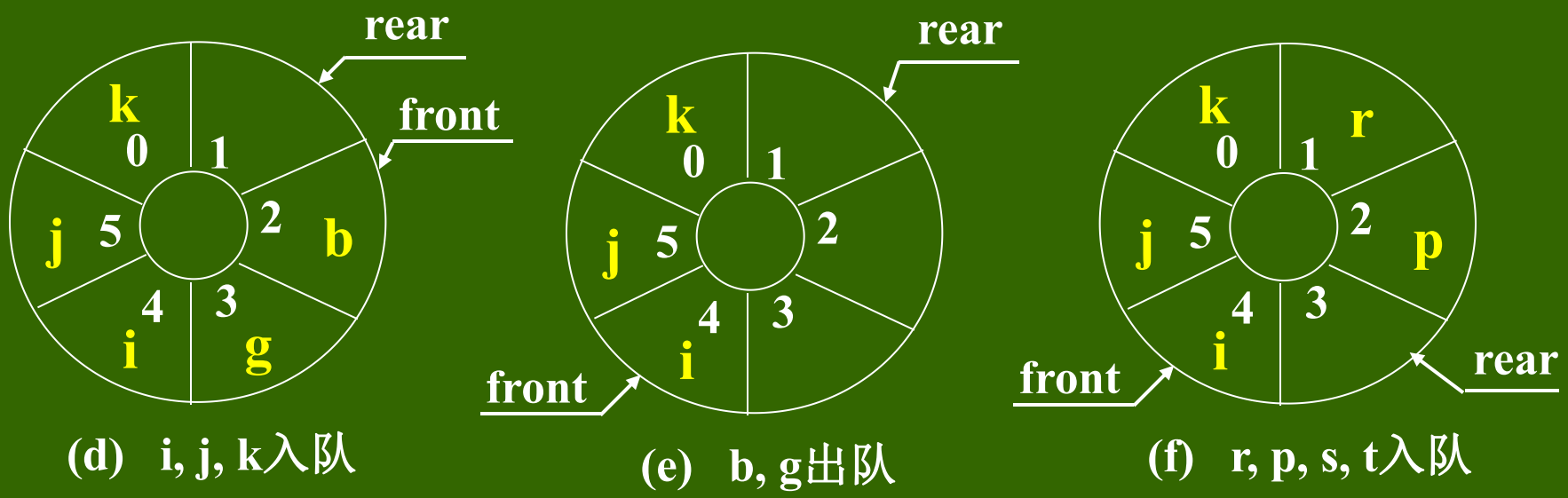


图3-7 循环队列操作及指针变化情况

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 ;
```

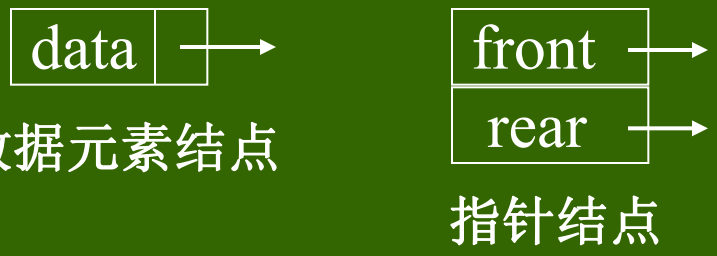


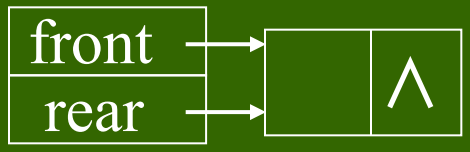
图3-8 链队列结点示意图

指针结点类型定义：

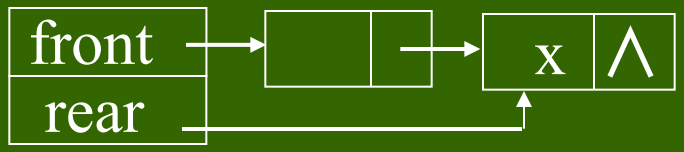
```
typedef struct link_queue
   {   QNode  *front , *rear ;
   }Link_Queue ;
```

2　链队运算及指针变化

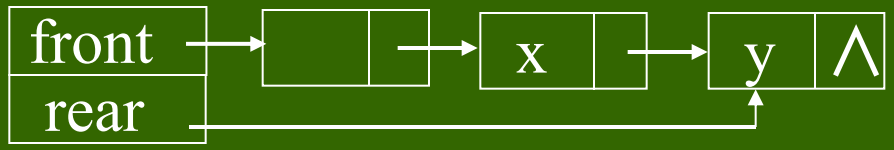
链队的操作实际上是单链表的操作，只不过是删除在表头进行，插入在表尾进行。插入、删除时分别修改不同的指针。链队运算及指针变化如图3-9所示。



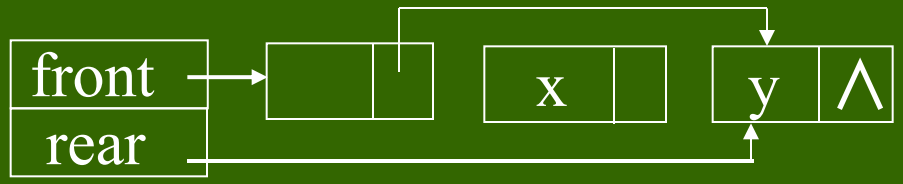
(a) 空队列



(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 ;
    }
```

(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;
         }
   }
```

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?

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