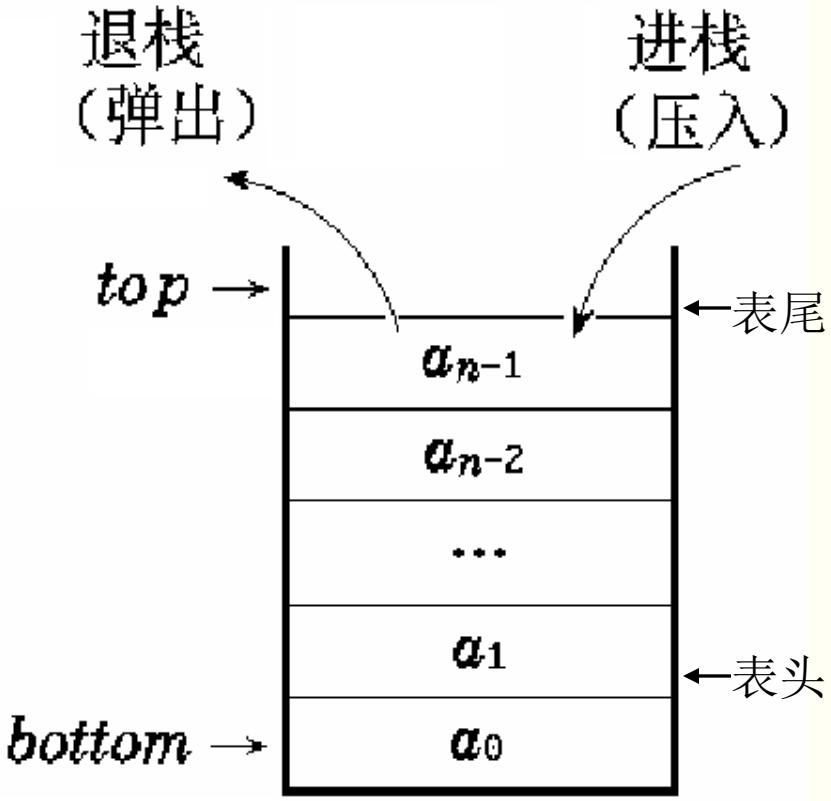
第3章　栈和队列

3.1 栈（ Stack ）

1. 定义：限定只在表的一端(表尾)进行插入和删除操作的线性表

- 特点:后进先出(LIFO)
- 允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)



栈的抽象数据类型定义：

ADT Stack {

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

数据关系：$R1=\{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,\ldots,n\}$

约定a_n端为栈顶，a_1端为栈底

基本操作：

}ADT Stack

栈的基本操作

INITSTACK (&s) 构造一个空栈s。

EMPTYSTACK(s) 判断s是否为空栈。若s为空栈，返回1，否则返回0。

PUSH (&s, x) 进栈操作。在栈s顶部插入一个新的元素x。

POP(&s, x) 退栈操作。若s非空，删除s中的栈顶元素，并返回该元素。

GETTOP(s) 取栈s的栈顶元素。与**POP(&s, x)**的区别是， **GETTOP(s)**不改变栈的状态。

CLEASTACK（&s) 将栈s清为空栈。

STACKLENGTH(s) 求栈的长度，返回栈s中的元素个数。

2. 栈的表示和实现

1）顺序栈—栈的顺序存储结构

2）链栈—栈的链式存储结构

3）静态分配整型指针

1）顺序栈——栈的顺序存储结构

限定在表尾进行插入和删除操作的顺序表

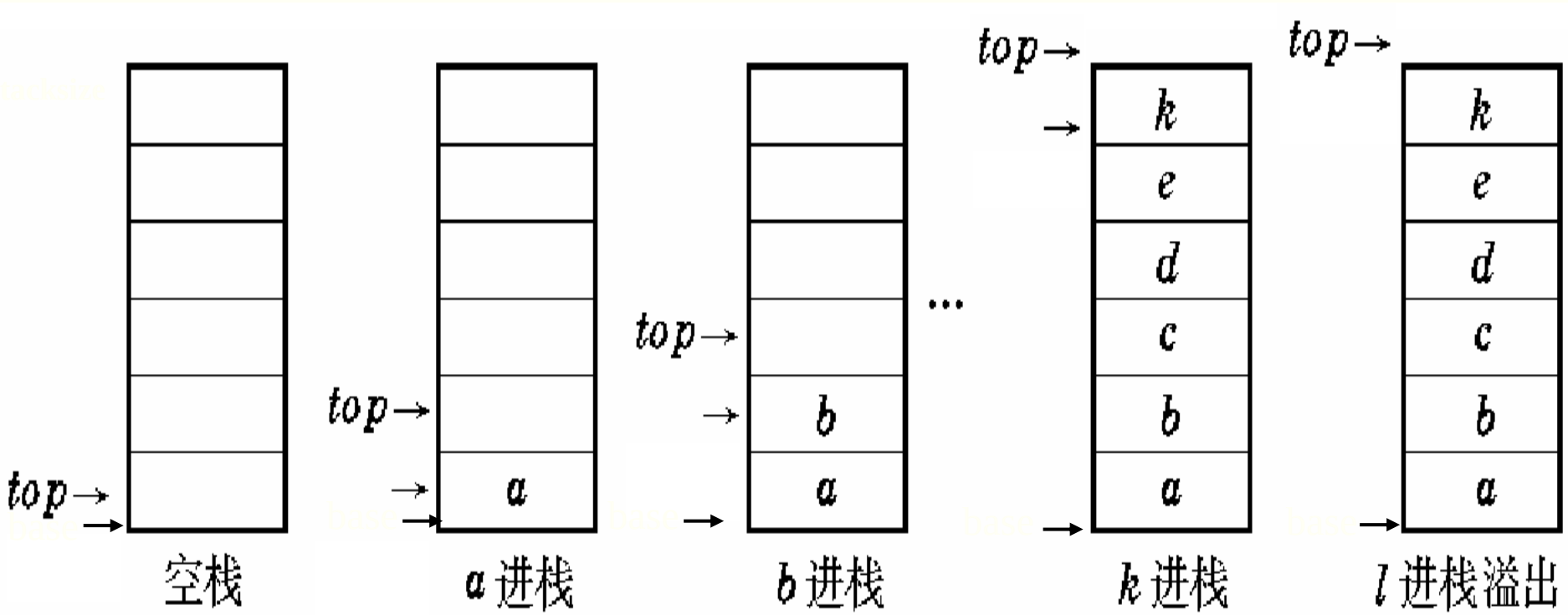
类型定义：　p46

```
typedef struct {
        SElemType   *base;
        SElemType   *top;
        int   stacksize;
        } SqStack;
SqStack   s;
```

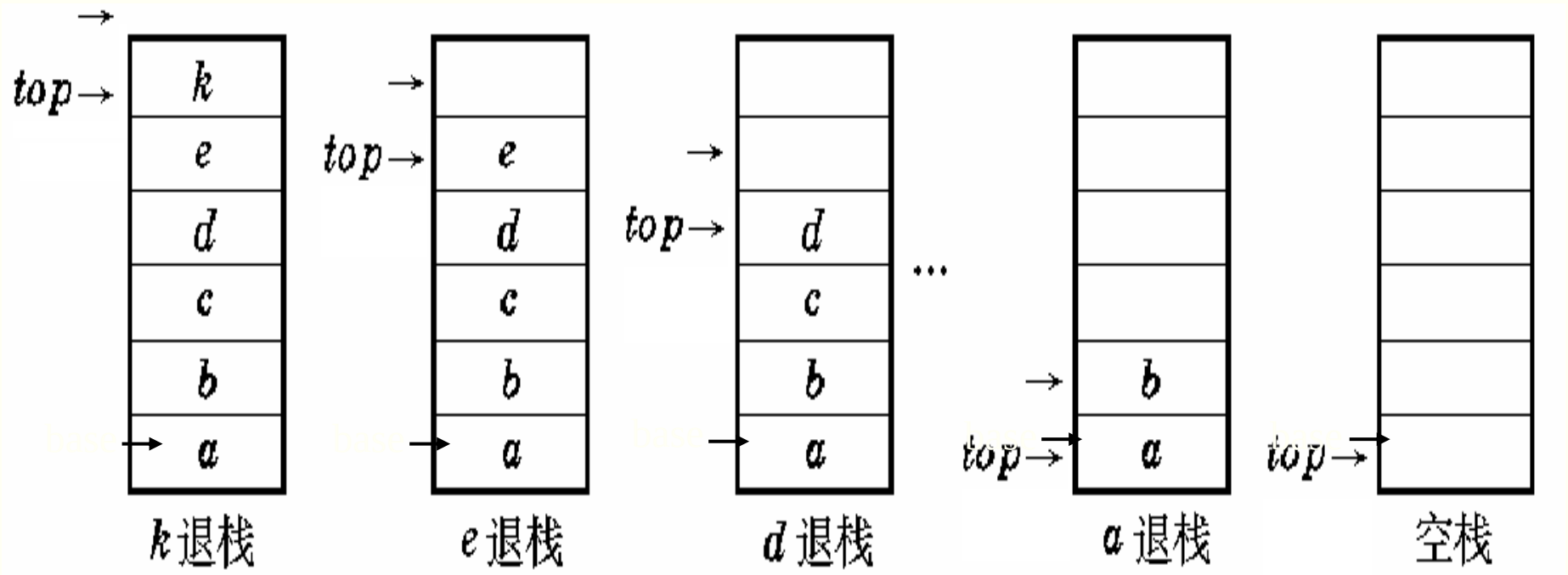
说明：

- base称为栈底指针，始终指向栈底；
  当base == NULL时，表明栈结构不存在。
- top为栈顶指针
  a. top的初始值指向栈底，即top=base
  b. 空栈：当top=base时为栈空的标记
  c. 当栈非空时，top的位置：指向当前栈顶元素的下一个位置
- stacksize ——当前栈可使用的最大容量

进栈示例



退栈示例



几点说明：

- 栈空条件：s.top ==s.base　此时不能出栈
- 栈满条件：s.top-s.base>=s.stacksize
- 进栈操作：*s.top++=e;　或*s.top=e; s.top++;
- 退栈操作：e=*--s.top;　或s.top--; e=*s.top;
- 当栈满时再做进栈运算必定产生空间溢出，简称“上溢”；
- 当栈空时,再做退栈运算也将产生溢出，简称为“下溢”。

基本操作的实现

* 栈的初始化操作 p47

  Status InitStack(SqStack &S)

* 取栈顶元素 p47

  Status GetTop(SqStack S, SElemType &e)

* 进栈操作 p47

  Status Push(SqStack &S, SElemType e)

* 退栈操作 p47

  Status Pop(SqStack &S, SElemType &e)

栈的初始化操作　p47

```
Status InitStack (SqStack &S) {
    S.base = (SElemType )malloc(STACK_INIT_SIZE * sizeof(ElemType));
  if (!S.base) return (OVERFLOW);
  S.top=S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
}
```

取栈顶元素　　p47

```
Status  GetTop(SqStack S, SElemType  &e)
{
  if (S.top == S.base) return ERROR;
  e = *(S.top-1);
  return OK;
}
```

进栈操作　　p47

```
Status  Push(SqStack  &S, SElemType  e)
{
  if (S.top-S.base>=S.stacksize)
     { S.base=(SElemType*)realloc(S.base,
  (S.stacksize+STACKINCREMENT)*sizeof(ElemType));
        if (!S.base) return (OVERFLOW);
        S.top = S.base +S.stacksize;
        S.stacksize += STACKINCREMENT;
     }
 *S.top++ = e;  /*   *S.top = e; S.top = S.top+1;
 return OK;
}
```

退栈操作　　p47

```
Status Pop(SqStack &S, SElemType  &e)
{
  if (S.top == S.base) return ERROR;
  e=*--S.top;  /* S.top=S.top-1;e=*S.top;
  return OK;
}
```

思考题

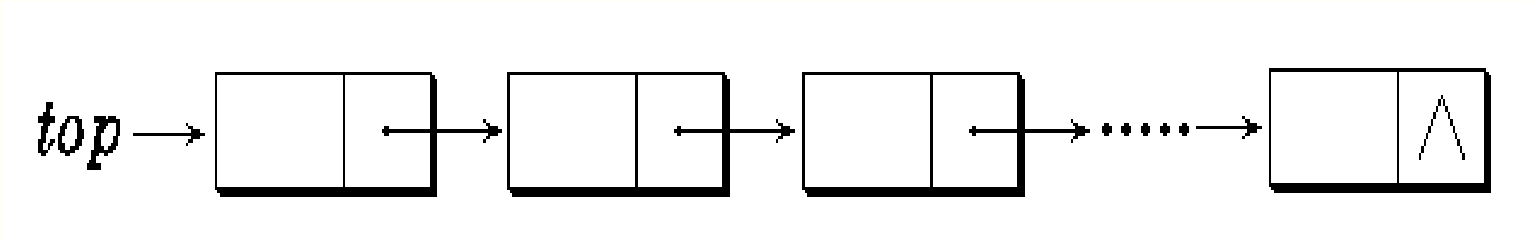
如果进栈的数据元素序列为A、B、C、D，则可能得到的出栈序列有多少种？写出全部可能的序列。

2）链栈——栈的链式存储结构

- 不带头结点的单链表，其插入和删除操作仅限制在表头位置上进行。链表的头指针即栈顶指针。
- 类型定义：

```
typedef struct SNode{
        SElemType  data;
        struct SNode  *next;
        }SNode, *LinkStack;
LinkStack s;
```

- 链栈示意图　　p47 图3.3



- 栈空条件：　　s=NULL
- 栈满条件：　　无　/ 无Free Memory可申请

进栈操作：

```
Status  Push_L (LinkStack &s, SElemType e)
    {  p=(LinkStack)malloc(sizeof(SNode));
       if (!p) exit(Overflow);
       p->data = e;   p->next = s;  s=p;
       return OK;
     }
```

退栈操作

```
Status  Pop_L (LinkStack &s, SElemType &e)
    {  if (!s) return  ERROR;
       e=s->data;  p=s;  s=s->next;
       free(p);
       return OK;
     }
```

3）静态分配整型指针

* 定义

```
#define MAXSIZE 100
 typedef struct {
        SElemType  base[MAXSIZE];
        int top;
 }SqStack;
 SqStack  s;
```

初始化

```
status InitStack(SqStack &s)
{   s.top = 0;
    return OK;
 }
```

进栈

```
Status Push(SqStack &s,SElemType e)
   {  if (s.top == MAXSIZE) return
  ERROR;
      s.base[s.top] = e;  s.top++;
      return OK;
    }
```

退栈

```
Status Pop(SqStack &s,SElemType &e)
    {  if (s.top ==0) return ERROR;
       s.top--;
       e=s.base[s.top];
       return OK;
     }
```

取栈顶元素

```
Status  GetTop(SqStack s, SElemType  &e)
{
  if (s.top == 0) return ERROR;
  e=s.base[s.top-1];
  return OK;
}
```

3.2　栈的应用

1. 数制转换　　p48 算法3.1
2. 行编辑程序　p50 算法3.2
3. 表达式求值 p52 ～ p54

1. 数制转换 p48

十进制N和其它进制数d的转换是计算机实现计算的基本问题，基于下列原理：

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	低位 4
168	21	0
21	2	5
2	0	高位 2

算法3.1

要求：输入一个非负十进制整数，输出任意进制数

```
void Conversion()
{  InitStack(s);
    scanf("%d,%d",&N,&base);
    N1=N;
    while (N1)
          {  Push(s,N1%base);
              N1 = N1/base;    }
    while (!(StackEmpty(s))
          {  Pop(s,e);
              if (e>9) printf("%c",e+55);
              else printf("%c",e+48);   }
    printf("\n");
}
```

2. 行编辑程序　p50 算法3.2

* 　　简单行编辑程序的**功能**：接受用户从终端输入的程序或数据，并存入用户的数据区。
* **做法**：设立一个输入缓存区，用以接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入时出差错。

  如：可用一个退格符“#”，删除前一个字符；

  　　可用一个退行符“@”，删除一行。
* **实现**：设这个缓存区为一个栈结构，每当从终端接受一个字符后作如下判别：

  如果既不是退格符也不是退行符，则压栈；

  如果是一个退格符，则从栈顶删去一个字符；

  如果是一个退行符，则将字符栈清为空栈。

2. 行编辑程序　p50 算法3.2

```
void lineedit( )
{  initstack(s);
   ch=getchar( );
   while(ch!=eof)
     { while(ch!=eof && ch!= '\n' )
        {  switch(ch)
                          {    case  '#'  : pop(s,c);
                              case  '@'  : clearstack(s);
                                default : push(s,ch);
                          }
                    ch=getchar( );
                 }
           把从栈底到栈顶的栈内字符传送到调用过程的数据区;
           clearstack(s);
           if(ch!=eof)      ch=getchar( );
        }
         destroystack(s);
 }
```

3. 表达式求值 p52 ~p54

- 算符优先法
- 运算：只考虑加、减、乘、除运算
- 运算符：+、—、*、/、（、）、#
- 算术四则运算的规则：
  - 先乘除，后加减；
  - 从左算到右
  - 先括号内，后括号外。

这样， A/B * C+D * E-A * C 应理解为：
(((A/B) * C)+(D * E))-(A * C)

- 左括号：比括号内的算符的优先级低
  比括号外的算符的优先级高
- 右括号：比括号内的算符的优先级低
  比括号外的算符的优先级高
- #：表达式的结束符，优先级总是最低

* 算符间的优先级关系 ： p53 表3.1

Θ_1 \ Θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

$\Theta_1<\Theta_2$： Θ_1的优先权低于 Θ_2

$\Theta_1>\Theta_2$： Θ_1的优先权高于 Θ_2

$\Theta_1=\Theta_2$ ： Θ_1的优先权等于 Θ_2

3. 表达式求值 p52～ p54

- 为实现算符优先算法，可使用两个工作栈：

  OPND栈：存数据或运算结果

  OPTR栈：存运算符

算法思想：

1. 初态： 置OPND栈为空；将“#”作为OPTR栈的栈底元素
2. 依次读入表达式中的每个字符
   1）若是操作数，则进入OPND栈；
   2）若是运算符，则与OPTR栈的栈顶运算符进行优先权（级）的比较：
   - 若读入运算符的优先权高，则进入OPTR栈；
   - 若读入运算符的优先权低，则OPTR退栈（退出原有的栈顶元素），OPND栈退出两个元素（先退出b，再退出a），中间结果再进入OPND栈；
   - 若读入“）”，OPTR栈的原有栈的栈顶元素若为“（”，则OPTR退出“（”；
   - 若读入“#”，OPTR栈栈顶元素也为“#”，则OPTR栈退出“#”，结束。

   例：3*（7-2）

例：3*（7-2）求解过程如下：

步骤	OPTR栈	OPND栈	输入字符	主要操作
1	#		3*（7-2）#	PUSH（OPND, ‘3’）
2	#	3	*（7-2）#	PUSH（OPTR, ‘*’ ）
3	#*	3	（7-2）#	PUSH（OPTR, ‘（’ ）
4	#*（	3	7-2）#	PUSH（OPND, ‘7’）
5	#*（	3 7	-2）#	PUSH（OPTR, ‘-’）
6	#*（-	3 7	2）#	PUSH（OPND, ‘2’）
7	#*（-	3 7 2	）#	operate(‘7’, ‘-‘, ’ 2’)
8	#*(	3 5	)#	POP(OPTR)
9	#*	3 5	#	operate(‘3’, ‘*’, ‘5’)
10	#	15	#	RETURN(GETTOP(OPND))

算术表达式求值算法

```
OperandType EvaluateExpression() {
    InitStack(OPTR); Push(OPTR,'#');
    InitStack(OPND); c=getchar();
    while (c!='#' || GetTOp(OPTR)!= '#' ) {
    if (! In(c,OP)) { Push(OPND,c); c=getchar();}
      else
           switch (Precede(GetTOp(OPTR),c)) {
             case '<' ; Push(OPTR,c); c=getchar(); break;
            case '=' ; Pop(OPTR,x); c=getchar(); break;
            case '>' ; Pop(OPTR,theta); Pop(OPND,b);
                     Push(OPND,Operate(a,theta,b)); break;
         }
     }
     return GetTop(OPND);
   }
```

3.4 队列

1. 定义

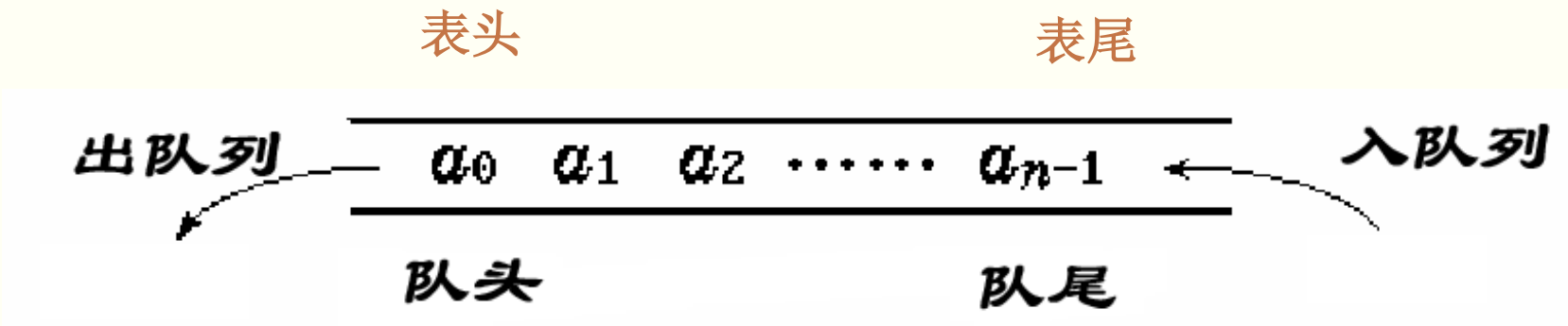
2. 链队列——队列的链式存储结构

3. 循环队列——队列的顺序存储结构

1. 定义

- 队列是限定在表的一端进行删除，在表的另一端进行插入操作的线性表。
- 允许删除的一端叫做队头(front)，允许插入的一端叫做队尾(rear)。
- 特性：FIFO(First In First Out)

图示 p59



队列的表示和实现

1）链队列——队列的链式存储结构

2）循环队列——队列的顺序存储结构

2. 链队列——队列的链式存储结构

- 实质是带头结点的线性链表
- 两个指针：
  - 队头指针Q.front指向头结点
  - 队尾指针Q.rear指向尾结点
- 初始态：队空条件
  - 头指针和尾指针均指向头结点

    Q.front = Q.rear

1）链队列的类型定义　p61

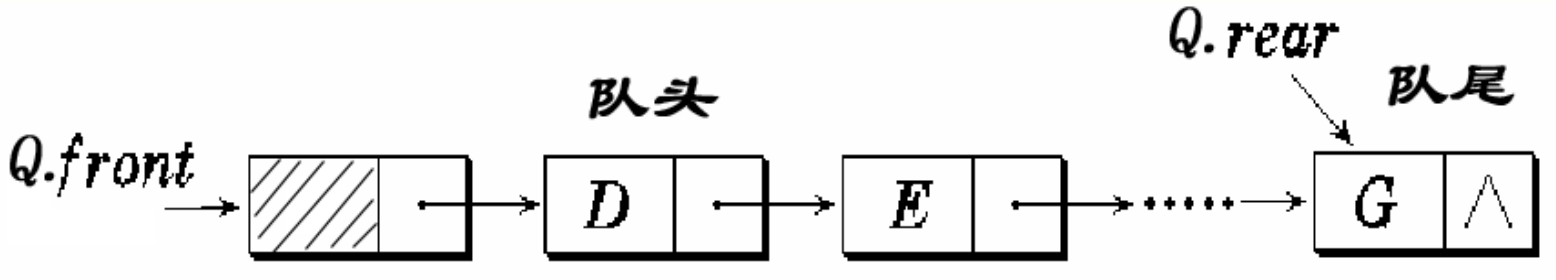
```
typedef  struct QNode {    //元素结点
   QElemType    data;
   struct QNode *next;
   }QNode, *QueuePtr;

typedef struct{                 //特殊结点
   QueuePtr  front;          //队头指针
   QueuePtr  rear;           //队尾指针
  }LinkQueue;
LinkQueue  Q;
```

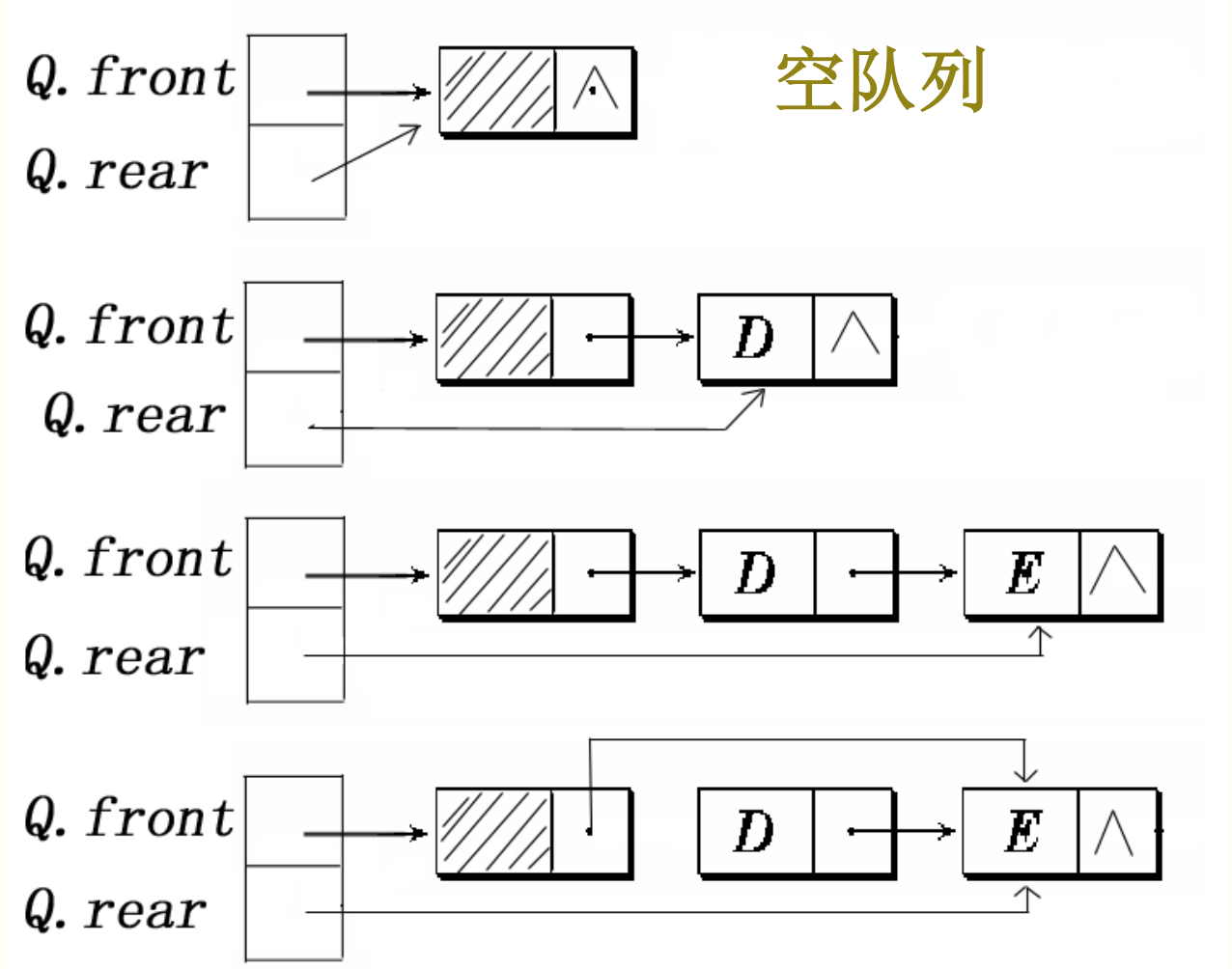
Q.front——指向链头结点

Q.rear ——指向链尾结点

2）链队列示意图　p61图3.10



队列运算指针变化状况


Q. front
Q. rear
空队列
Q. front
Q. rear
D
Q. front
Q. rear
D
E
Q. front
Q. rear
D
E

3）基本操作与实现

* 初始化　p62
  Status InitQueue (LinkQueue &Q)
* 销毁队列　p62
  Status DestroyQueue (LinkQueue &Q)
* 入队　p62
  Status EnQueue(LinkQueue &Q, QElemType e)
* 出队　p62
  Status DeQueue(LinkQueue &Q, QElemType &e)
* 判队空
  Status QueueEmpty(LinkQueue Q)
* 取队头元素
  Status GetHead(LinkQueue Q,QElemType &e)

链队列初始化

```
Status InitQueue (LinkQueue &Q)
{
  Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
  if (!Q.front) exit(OVERFLOW);
  Q.front->next=NULL;
  return OK;
}
```

链队列的销毁

```
Status DestroyQueue (LinkQueue &Q)
{ while (Q.front)
        {  Q.rear=Q.front->next;
           free(Q.front);
           Q.front=Q.rear;
        }
  return OK;
}
```

链队列的插入（入队）

```
Status EnQueue (LinkQueue &Q, QElemType e)
  {  p=(QueuePtr)malloc(sizeof(QNode));
    if (!p) exit(OVERFLOW);
    p->data = e;  p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return OK;
}
```

链队列的删除（出队）

```
Status DeQueue (LinkQueue &Q, ElemType &e)
{  if (Q.front==Q.rear) return ERROR;
  p=Q.front->next;
  e=p->data;
  Q.front->next=p->next;
  if (Q.rear == p) Q.rear=Q.front;
  free(p);
  return OK;
}
```

判断链队列是否为空

```
Status QueueEmpty(LinkQueue Q)
{
  if (Q.front==Q.rear) return TRUE;
  return FALSE;
}
```

取链队列的第一个元素结点

```
Status GetHead(LinkQueue Q,QElemType &e)
{
  if (QueueEmpty(Q)) return ERROR;
  e=Q.front->next->data;
  return OK;
}
```

3. 循环队列——队列的顺序存储结构

顺序队列：

——用一组地址连续的存储单元依次存放从队列头到队列尾的元素

设两个指针：

——Q.front 指向**队列头元素**；

——Q.rear 指向**队列尾元素的下一个位置**

初始状态（空队列）：

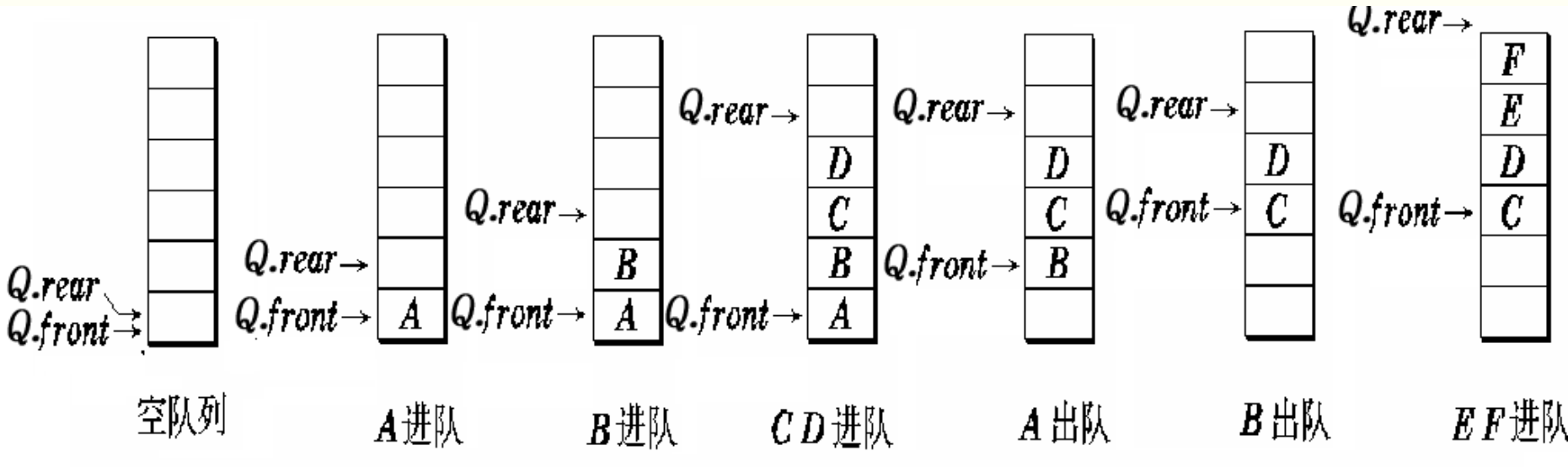
Q.front = Q.rear=0

队列的真满与假满

类型定义 p64

```
#define MAXSIZE 100
typedef struct {
    QElemType  *base;
    int   front;
    int   rear;
  }SqQueue;
SqQueue  Q;
```

队列的进队和出队

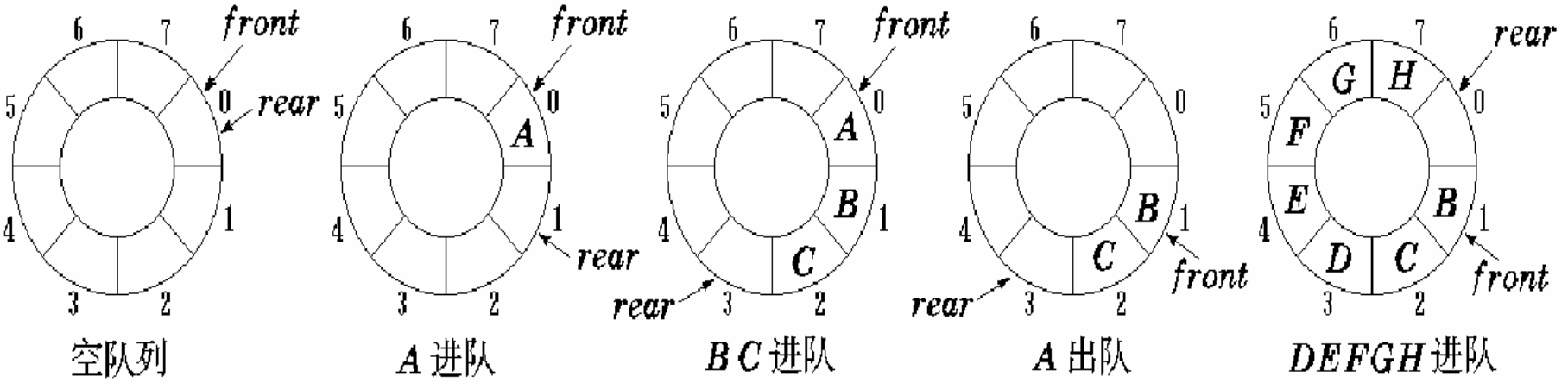


- 进队时，将新元素按Q.rear 指示位置加入，再将队尾指针增1，Q.*rear* = Q.*rear* + 1。
- 出队时，将下标为Q.front 的元素取出，再将队头指针增1，Q.*front* = Q.*front* + 1。
- 队满时再进队将溢出出错；队空时再出队作队空处理。上图为“假满”

循环队列（Circular Queue）

- 存储队列的数组被当作首尾相接的表处理。
- 队头、队尾指针加1时从maxsize -1直接进到0，可用语言的取模(余数)运算实现。
- 队头指针进1： Q.front = (Q.front + 1)% MAXSIZE
  队尾指针进1： Q.rear = (Q.rear + 1)% MAXSIZE;
- 队列初始化：Q.front = Q.rear = 0;
- 队空条件：Q.front == Q.rear;
- 队满条件：(Q.rear + 1) % MAXSIZE == Q.front
- 队列长度：(Q.rear-Q.front+MAXSIZE)%MAXSIZE

循环队列的进队和出队



说明

- 不能用动态分配的一维数组来实现循环队列，初始化时必须设定一个最大队列长度。
- 循环队列中要有一个元素空间浪费掉
  ——p63　约定队列头指针在队列尾指针的下一位置上为“满”的标志
- 解决 Q.front=Q.rear不能判别队列“空”还是“满”的其他办法：
  - 使用一个计数器记录队列中元素的总数（即队列长度）
  - 设一个标志变量以区别队列是空或满

基本操作

- 初始化　p64

Status InitQueue (SqQueue &Q)

- 求队列的长度　p64

int QueueLength(SqQueue Q)

- 入队　p65

Status EnQueue (SqQueue &Q, QElemType e)

- 出队　p65

Status DeQueue(SqQueue &Q, QElemType &e)

- 判队空

Status QueueEmpty(SqQueue Q)

- 取队头元素

Status GetHead(SqQueue Q,QElemType &e)

Status InitQueue (SqQueue &Q)

```
{
  Q.base=(QElemTye )malloc(MAXQSIZE*sizeof(QElemType));
  if (!Q.base) exit(OVERFLOW);
  Q.front=Q.rear=0;
  return OK;
}
```

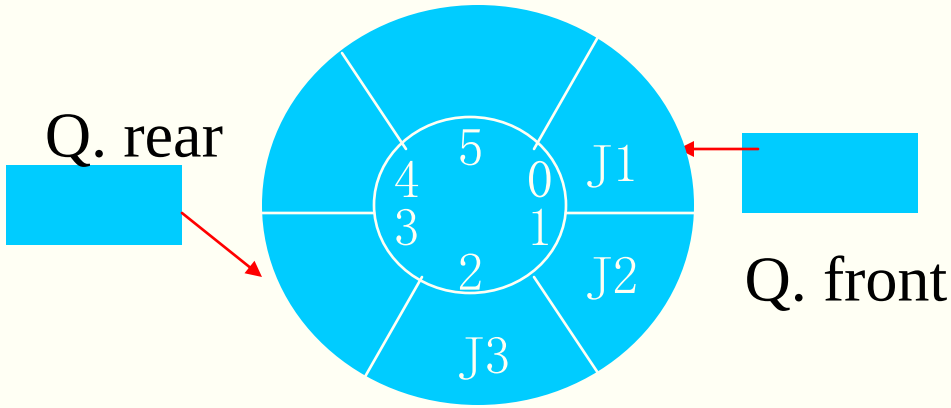
int QueueLength(SqQueue Q)

```
{
  return (Q.rear-Q.front+MAXQSIZE)%MAXQSIZE;
}
```

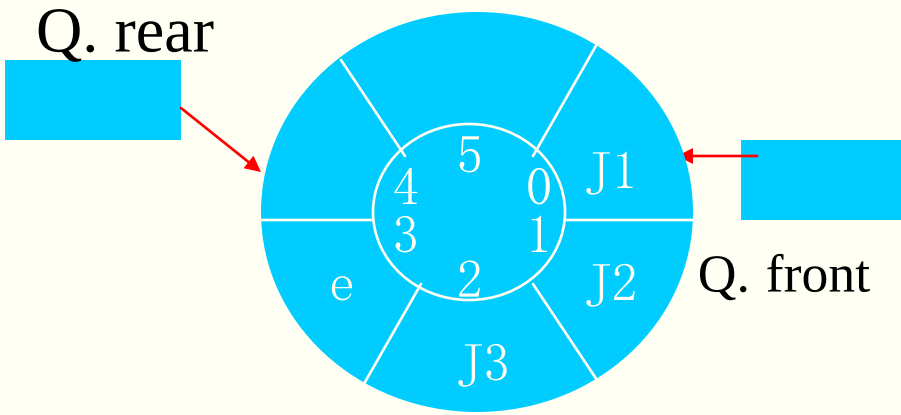
```
Status EnQueue (SqQueue &Q, QElemType e)

{
  if ((Q.rear+1) % MAXQSIZE ==Q.front)
     return ERROR;
  Q.base[Q.rear]=e;
  Q.rear = (Q.rear+1)%MAXQSIZE;
  return OK;
}
```

入队操作：将元素 e 插入队尾



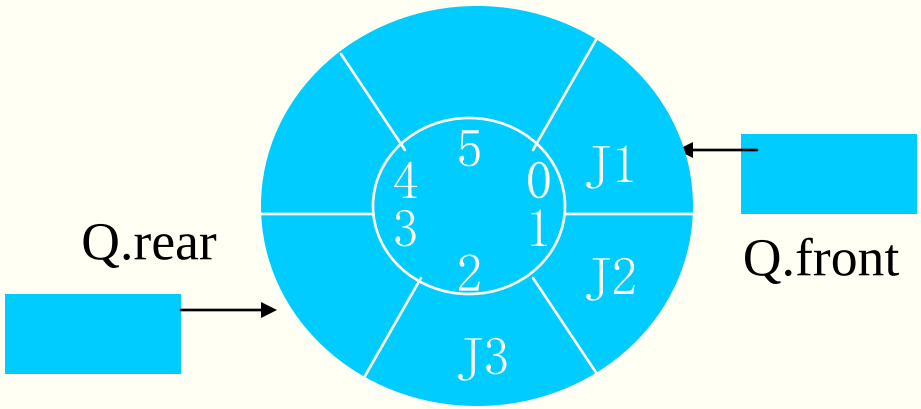
元素 e 入队前



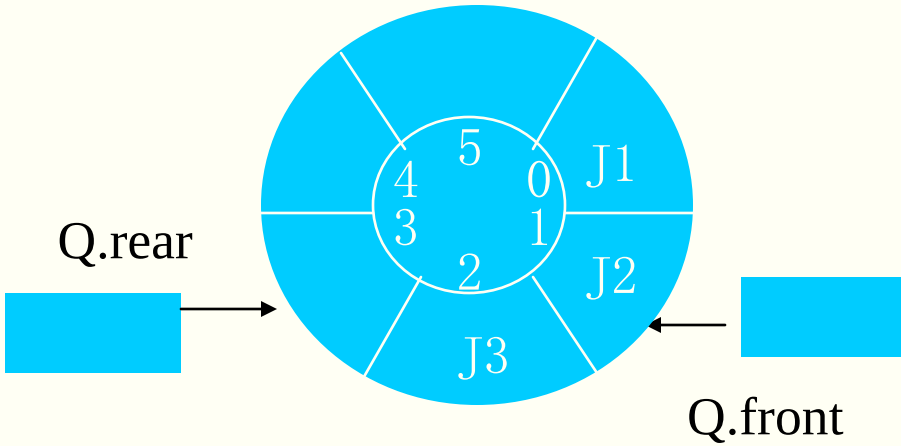
元素 e 入队后

```
Status DeQueue(SqQueue &Q, QElemType &e)
{
  if (Q.rear==Q.front) return ERROR;
  e=Q.base[Q.front];
  Q.front=(Q.front+1)%MAXQSIZE;
  return OK;
}
```

出队操作：删除队头元素



出队操作前



出队操作后

```
Status QueueEmpty(SqQueue Q)
{
  if (Q.front==Q.rear) return TRUE;
  return FALSE;
}
```

```
Status GetHead(SqQueue Q,QElemType &e)
{
  if QueueEmpty(Q) return ERROR;
  e=Q.base[Q.front];
  return OK;
}
```

非循环队列

- 类型定义：与循环队列完全一样
- 关键：修改队尾/队头指针

  Q.rear = Q.rear + 1;　　Q.front = Q.front+1;
- 在判断时，有%MAXQSIZE为循环队列，否则为非循环队列
- 队空条件：　Q.front = Q.rear
- 队满条件：　Q.rear>= MAXQSIZE
- 注意“假上溢”的处理
- 长度：　　Q.rear - Q.front

4. 队列的应用举例

1）解决计算机主机与外设不匹配的问题，如解决高速CPU与低速打印设备之间速度不匹配问题；

2）解决由于多用户引起的资源竞争问题；

在操作系统课程中会讲到

3）离散事件的模拟----模拟实际应用中的各种排队现象；

p65

4）用于处理程序中具有先进先出特征的过程。

- 本章重点：
  - 线性表、栈、队列的异同。
  - 顺序栈和链栈的进栈、退栈算法，栈的“上溢”、“下溢”概念及其判别条件。
  - 顺序队列（循环队列）和链队列的入队、出队算法，队列的“上溢”、“下溢”概念及其判别条件。
  - 循环队列对边界条件的处理方法