栈

考察重点：栈的定义、特点、顺序与链式存储表示、基本算法；栈的应用；队列的定义、特点；链队列、循环队列相关的定义、特点、基本算法；栈与递归的实现

栈的定义

栈是一种只能在**一端**进行插入或删除操作的**线性表**。其中允许进行插入或删除操作的一端称为栈顶（top）。栈顶由一个称为栈顶指针的位置指示器（其实就是一个变量。对于顺序栈，就是记录栈顶元素所在数组位置标号的一个整型变量，对于链式栈就是记录栈顶元素所在结点地址的指针）来指示，它是动态变化。表的另一端称为栈底，栈底是固定不变的。栈的插入和删除操作一般被称为**入栈**和**出栈**。

栈的特点

由栈的定义可以看出栈的特点，一句话概括就是**先进后出（FILO）**。栈中的元素就好比开进一个死胡同的多辆汽车，最先开进去的汽车只能等后进来的汽车都出去了，才能出来。

栈的存储结构

栈有两种主要的存储结构，**顺序栈和链式栈**。因为在栈的定义中已经说明，栈是一种在操作上稍加限制的线性表，即栈本质上是线性表，而线性表有两种主要存储结构，顺序表和链表，因此栈也同样有对应的两种存储结构。

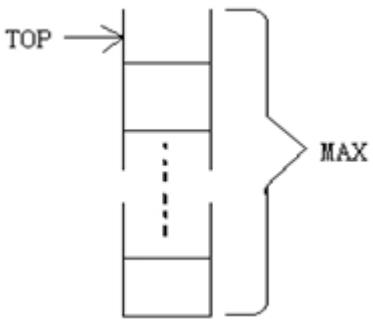
队列的定义

队列简称队，它也是一种运算受限的**线性表**，其限制为允许在表的一端进行插入，而在表的另一端进行删除。进行插入的一端称作**队尾**（rear），进行删除的一端称为**队头**（front）。向队列中插入新元素称为**进队**，新元素进队后就成为新的队尾元素；从队列中删除元素称为**出队**，元素出队后，其后继元素就成为新的队头元素。

队列的特点

队列的特点概括一句话就是**先进先出（FIFO）**。打个比方说，队列就好像开进隧道的一列火车，各节车厢就是队中的元素，最先开进隧道的车厢总是最先驶出隧道。

顺序栈的定义

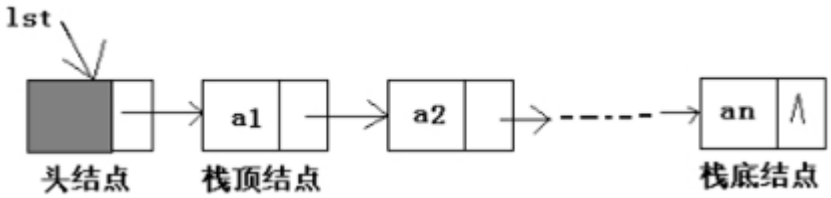


```
typedef struct {
    int data[MAX]; //存放栈中元素
    int top;    //栈顶指针
} SqStack;  //顺序栈类型定义
```

链栈节点的定义

```
typedef struct LNode {
    int data;   //数据项
    struct LNode *next; //指针域
}LNode;
```

链栈就是采用链表来存储栈。这里我们用带头结点的单链表来作为存储体，示意图如下：



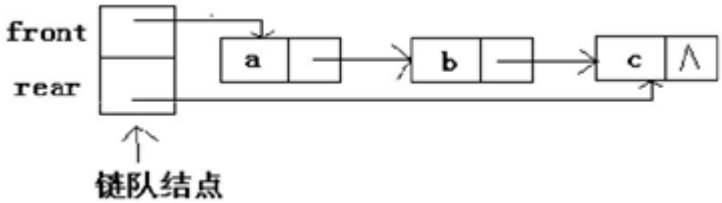
顺序队列定义

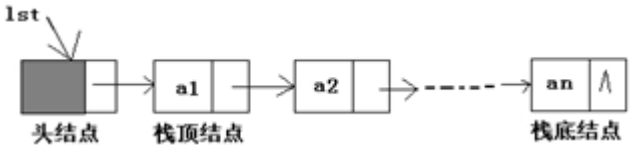
```
typedef struct {
    int data[MaxSize];
    int front;
    int rear;
}SqQueue;
```

链队定义

```
// 队节点类型定义
typedef struct QNode {
    int data;
    struct QNode *next;
}QNode;

// 链队类型定义
typedef struct {
    QNode *front;
    QNode *rear;
}LiQueue;
```





顺序栈的基本算法操作

1. 顺序栈的要素

对于顺序栈 st，一共有 4 个要素，包括两个特殊状态，两个操作。（1）两状态：

①栈空状态 `st.top==-1`（有的书上规定 `st.top==0` 为栈空条件，这样会浪费一个元素大小的空间，本书统一规定栈空状态为 `st.top==-1`。考试中有时会出现其他规定，其实大同小异稍加注意即可）

②栈满状态 `st.top==MAX-1`（MAX 为栈中最大元素个数，则 MAX-1 为栈满时栈顶元素在数组中的位置，因为数组下标从 0 号开始。规定栈顶指针 top 为-1 时栈空，即 `top==0` 的数组位置也可以存有数据元素，这与之前顺序表中规定 0 号数组位置不存数据元素是不同的，考生应注意这一点）

（2）两操作：

①元素 x 进栈操作 `st.top++;st.data[st.top]=x;`（既然规定了 top 为-1 时栈为空，则元素进栈操作必须是先移动指针，再进入元素，因为数组下表不存在-1。在其他书中因有不同规定，会有先元素进栈再栈顶指针加 1 的进栈操作，其实本质一样，考生注意即可）

②元素 x 出栈操作 `x=st.data[st.top];st.top--;`（进栈操作次序决定了出栈操作次序，为什么这样说呢，如果进栈操作是先变动栈顶指针，再存入元素，则出栈操作必须为先取出元素，再变动指针。如果在上述进栈操作不变的情况下先变动指针，再取出元素，

则栈顶元素丢失，取出的是栈顶下边的元素。

2. 初始化栈的算法

初始化一个栈只需将栈顶指针置为 -1 即可。对应算法如下：

```
void initStack(Sqstack &st) {
    st.top=-1;  //只需将栈顶指针设置为-1。
}
```

3. 判断栈空的算法

栈 st 为空的时候返回 1，否则返回 0。对应算法如下：

```
int StackEmpty(SqStack st) {
    if (st.top==-1)
        return 1;
    else
        return 0;
}
```

4. 进栈算法

```
int Push(SqStack &st,int x) {
    if(st.top==MAX-1)   //这里要注意，栈满不能进栈。
        return 0;
    st.top++; //先移动指针再进栈。
    st.data[st.top]=x;
    return 1;
}
```

5. 出栈算法

```
int Pop(SqStack &st,int &x) {
    if(st.top==-1)  //注意，如果栈空则不能出栈
        return 0;
    x=st.data[st.top]; //先取出元素，再移动指针
    st.top--;
    return 1;
}
```

> 说明：在考试中，栈常常作为一个工具来解决其他问题，因此一般情况下，栈的声明以及操作可以写的很简单，不必要调用以上函数。上述函数只作为标准操作来参考，使用价值不高。在考题中比较实用的栈的操作的写法如下：
>
> 1．声明一个栈并且初始化，假设元素是 int 型，可以这么写： int stack[MAX]; int top=-1; 这两句话连声明带初始化都有了。
>
> 2．元素 x 进栈：`stack[++top]=x;` 这一句话就搞定进栈操作。
>
> 3．元素 x 出栈：`x=stack[top--];` 这一句话就搞定出栈操作。
>
> 2 与 3 点需注意的地方是，当前栈是否为空，空时不出；是否为满，满时不进。这些判断根据题目需要来决定写还是不写，不必要像标准操作那样每次都判断（比如题目中入栈元素不多，而 MAX 足够大，就无需考虑入栈操作会产生溢出）。通过 2 与 3 两点，来稍微复习一下 C 语言基础。`top++` 与 `++top`（`--top` 与 `top--` 的情况类似）的不同可以用几句话来说清楚。前者是先将 top 赋值给接收它的变量，然后 top 自增 1，而后者是先 top 自增之后，再把值赋给接收它的变量。比如 `a=top++;` 中 a保存了自增前的 top 值，而 `a=++top;` 中 a 保存了自增后的 top 值。同样 `stack[++top]=x;` 中 x 存放到 top 变化之后所指示的位置上（一下看不懂可以拆开看。第一步：top 先自增1。第二步：自增后的 top 把自己的数值放在 stack[] 的括号内而指出了将要保存元素的位置。第三步：x 存储在 top 所指的位置上）。这里看懂了，就很容易理解 `stack[++top]=x;` 等价于 `top++; stack[top]=x;` 而 `x=stack[top--];` 等价于 `x=stack[top];top--;`。

例题1：如果1.2.3依次进栈，会有哪些出栈次序?

例题2：c 语言里算术表达式中的括号只有小括号。编写算法，判断一个表达式中的括号是否正确配对，表达式已经存入字符数组 exp[] 中（元素从数组下标 1 开始存储），表达式中的字符个数为 n。

分析：

本题可以用栈来解决，为什么可以用栈来解决呢，这是关键。一个表达式，用目测怎么判断括号是否匹配？可以这样做，从左往右看这个表达式中的括号，看到一个“(”，就记住它（这里可以理解为入栈），如果下一个括号是“)”，则划掉这两个括号（这里可以理解为出栈），表示一对括号处理完毕继续往后看。如果前边所有的括号都被划掉，而下一个括号却是“)”，则括号一定不匹配，因为“)”之前已经没有括号和它匹配了。如果下一个括号是“(”，则暂时不管前一个“(”，先把它放在那里，等后边的“(”处理掉后再来处理它（后边的“(”处理掉才能回来处理先前的“(”，这里体现了栈的先进后出）。以后看到的括号要么是“(”要么是“)”，就用前边说的方法来处理。如果到最后所有括号都被划掉，则匹

配，否则就不匹配。由此可以看到，一个问题中如果出现这种情况，即在解决问的过程中出现了一个状态，但凭现有条件不能判断当前的状态是否可以解决，需要记下，等待以后出现可以解决当前状态的条件后返回来再解决之。这种问题需要用栈来解决，栈具有记忆的功能，这是栈的`FILO`特性所延伸出来的一种特性。

通过以上分析可知，此题应该用栈来解决，代码如下：

```
int match(char exp[],int n) {
    char stack[MAX]; int top=-1; //两句完成栈的声明和初始化，考试中的这种简写可以节省时间。
    int i;
    for(i=1;i<=n;i++) {
        if(exp[i]=='(') //如果遇到“(”则入栈等待以后处理。
        stack[++top]='('; //一句话完成入栈操作。
        if(exp[i]==')') {
            if(top==-1)//如果当前遇到的括号是“)”且栈已空，则不匹配，返回 0。
                return 0;
            else
                top--;  //如果栈不空则出栈，这里相当于完成了以上分析中的
        }   //划掉两个括号的动作。
    }   //栈空，即所有括号都被处理掉则说明括号是匹配的。
    if(top==-1)
        return 1;   //否则括号不匹配。
    else
        return 0;
}
```

例题3：编写一个函数，求后缀式的数值，其中后缀式存于一个字符数组 exp 中，exp中最后一个字符为`\0`，作为结束符，并且假设后缀式中的数字都只有一位。

分析：

这里首先要复习一下算术表达式的三种形式：前缀式、中缀式、后缀式。中缀式是我们所熟悉的表达式。比如 $(a+b+c\times d)/e$ 是一个中缀式，转化为前缀式为：$/++ab\times cde$，转化为后缀式为：$abcd\times++e/$。

注意：中缀表达式转化成后缀或者是前缀，结果并不一定唯一。比如 $ab+cd\times+e/$ 同样是 $(a+b+c\times d)/e$ 的后缀式。后缀式和前缀式都只有唯一的一种运算次序，而中缀式却不一定，后缀式和前缀式是由中缀式按某一种运算次序而生成的，因此对于一个中缀式可能有多种后缀式或者前缀式。比如 $a+b+c$ 可以先算 $a+b$ 也可以先算 $b+c$，这样就有两种后缀式与其对应，分别是 $ab+c+$ 和 $abc++$。

回到本题，后缀式的求值可以用栈来解决，为什么呢？对于一个后缀式，当从左往右扫描到一个数值的时候，具体怎么运算，此时还不知道，需要扫描到后边的运算符才知道，因此必须先存起来，这符合例题2中所描述的情形，因此可以用栈来解决。

执行过程：当遇到数值的时候入栈，当遇到运算符的时候，连续两次出栈，将两个出栈元素结合运算符进行运算。将结果当成新遇到的数值入栈。如此往复，直到扫描到终止符'\0',此时栈底元素值即为表达式的值。

由此可以写出以下代码：

```
int op(int a,char Op,int b) { //本函数是运算函数，来完成算式 a Op b 的运算
    if(Op=='+')
        return a+b;
    if(Op=='-')
        return a-b;
    if(Op=='*')
```

```
        return a*b;
    if(Op=='/') {
        if(b==0) { //这里需要判断，如果除数为 0 则输出错误标记，这种题目中的小陷阱
            cout<<"ERROR"<<endl; return 0;
        } else
            return a/b;
    }
}   //后缀式计算函数。
int com(char exp[])  {  //a,b 为操作数，c 来保存结果
    int i,a,b,c;
    int stack[MAX];
    int top=-1;//栈的初始化和声明，注意元素类型必须为 int 型，不能是char 型。因为虽然题目中说操作数都只有一位，但是在运算过程中可能产生多位的数字，因此要用整型。
    char Op;    //Op 用来取运算符。
    for(i=0;exp[i]!='\0';i++) {
        if(exp[i]>='0'&&exp[i]<='9')//如果遇到操作数，则入栈等待处理，体现了栈的记忆功能。
            stack[++top]=exp[i]-'0'; //注意：字符型和整形的转换（后边讲解）。
        else {  //如果遇到运算符，则说明前边待处理的数字的处理条件已经具备，开始运算。
            Op=exp[i];
            b=stack[top--];    //取第二个操作数（因为第二个操作数后入栈，所以先出栈的是第二个操作数）。
            a=stack[top--];    //取第一个操作数。
            c=op(a,Op,b);   //将两个操作数结合运算符 Op 进行运算，结果保存在 c 中。
            stack[++top]=c; //运算结果入栈。
        }
    }
    return stack[top];
}
```

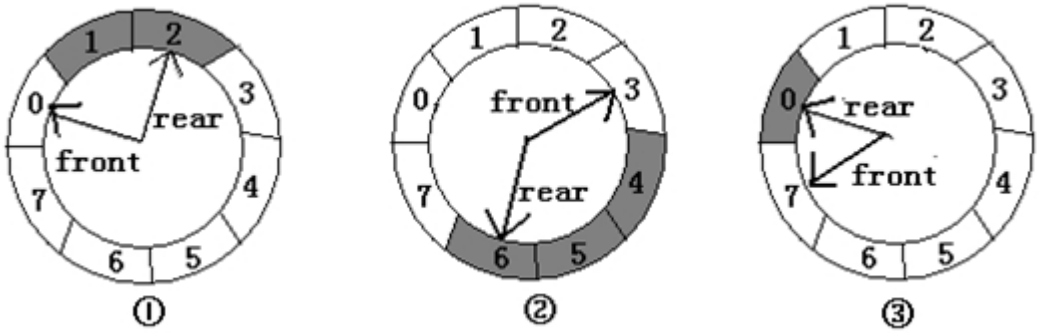
假设有一个字符‘5’，如果定义一个整型变量 a，执行 a=‘5’;此时 a 里边保存了 5 的 ASCII 码，而不是数字 5。如何将‘5’这个字符代表的真正意义，即 5 这个整数保存于 a 中呢？只需要执行 `a=‘5’-‘0’;` 即可。同理，如果把一个整型数字(假设为 a)，转化为对应的字符型数字存储在字符变量(假设为 b)中，只需执行 `b=a+‘0’;` 即可。此时 b 中保存的是 a 这个数字的字符，但是这种转化只适用于 0~9 这 10 个数字。这个小技巧在程序设计题目中应用的比较多。

队列的算法应用：

1.循环队列

在顺序队中，通常让队尾指针 `rear` 指向刚进队的元素位置，让队首 `front` 指针指向刚出队的元素位置。因此元素进队的时候，`rear` 要向后移动，元素出队的时候，`front` 也要向后移动，这样经过一系列的出队和进队操作以后，两个指针最终会达到数组末端 MAX-1处，虽然队中已经没有元素，但仍然无法让元素进队，这就是所谓的“假溢出”。要解决这个问题，我们可以把数组弄成一个环，让 `rear` 和 `front` 沿着环走，这样就永远不会出现两者来到数组尽头无法继续往下走的情况，这样就产生了循环队列，循环队列是改进的顺序队列。示意图如下：

`Front` 指向一个空位置，`rear` 指向一个位置



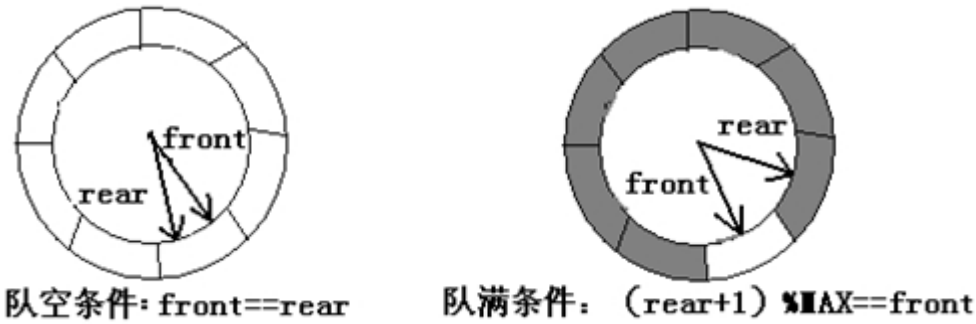
①由空队进队两个元素，此时 front 指向 0，rear 指向 2。

②进队 4 个元素，出对 3 个元素，此时 front 指向 3，rear 指向 6。

③进队 2 个元素，出队 4 个元素，此时 front 指向 7，rear 指向 0。

从上图中由①到③的变化过程可以看出，经过元素的进进出出，即便是 `rear` 和 `front` 都到了数组尾端（③图所示），依然可以让元素继续入队，因为两指针不是沿着数组下表递增的直线行走，而是沿着一个环行走，走到数组尽头的时候自动返回数组起始位置。怎样实现指针在递增的过程中沿着环形道路行走呢，有一个方法，就上图 3.1 中的例子，拿 `front` 指针来说，可以循环执行语句 `front=(front+1)%8` ，`front` 初值为 0 的话，在一个无限循环中，`front` 的取值为 0,1,2,3,4,5,6,7,0,1,2……，即以 0 到 7为周期的无限循环数，也就是 `front` 沿着上图的环在行走。对于一般情况，上述语句可写为 `front=(front+1)%MAX`(MAX 是数组长度)。

队满和队空（rear 的情况和 front 类似）。



2.循环队列的要素

（1）两状态：

①队空状态 `qu.rear==qu.front`

②队满状态 `(qu.rear+1) %MAX==qu.front`

（2）两操作：

①元素 x 进队操作（移动队尾指针） `qu.rear=(qu.rear+1)%MAX;qu.data[qu.rear]=x;`

②元素 x 出队操作（移动队首指针） `qu.front=(qu.front+1)%MAX;x=qu.data[qu.front]`

说明：元素入队时，先移动指针，后存入元素；元素出队时，也是先移动指针再取出元素。其他地方可能有不同的次序，其实本质是一样的，对于程序设计题目已经足够。对于选择题，则可根据题目描述确定是先存取元素，再移动指针，还是其他处理顺序。

3.始化队列算法

```
void InitQueue(SqQueue &qu) {
    qu.front=qu.rear=0; //队首队尾指针重合，且指向 0。
}
```

4．判断队空

```
int QueueEmpty(SqQueue qu) {
    if(qu.front==qu.rear) //不论队首队尾指针指向数组中的哪个位置，
        return 1; //只要两者重合，即为队空。
    else
        return 0;
}
```

5．元素进队

```
int enQueue(SqQueue &qu,int x) {
    if((qu.rear+1)%MAX==qu.front)    //队满的判断条件，满则不能入队。
        return 0;
    qu.rear=(qu.rear+1)%MAX;     //先移动指针。
    qu.data[qu.rear]=x; //再存入元素。
    return 1;
}
```

6．元素出队

```
int deQueue(SqQueue &qu,int &x) {
    if(qu.front==qu.rear)    //队空则不能出队 。
        return 0;
    qu.front=(qu.front+1)%MAX;   //先移动指针。
    x=qu.data[qu.front];     //再取出元素。
    return 1;
}
```

练习题：

1.假设以 I 和 O 分别表示入栈和出栈操作。栈的初态和终态均为空，入栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列，称可以操作的序列为合法序列，否则称为非法序列。

（1）试指出判别给定序列是否合法的一般规则。

（2）两个不同合法序列（对同一输入序列）能否得到相同的输出元素序列？如能得到，请举列说明。

2.有 5 个元素，其入栈次序为：A,B,C,D,E，在各种可能的出栈次序中，以元素 C,D 最先出栈（即 C 第一个且 D 第二个出栈）的次序有哪几个？

3.如果允许在循环队列的两端都可以进行插入和删除操作。要求：（1）写出循环队列的类型定义；（2）写出“从队尾删除”和“从队头插入”的算法。

4.设计一个循环队列，用 front 和 rear 分别作为队头和队尾指针，另外用一个标志 tag 表示队列是空还是不空，约定当 tag 为 0 时队空，当 tag 为 1 时队不空，这样就可以用 front==rear 作为队满的条件。要求设计队列的结构和相关基本运算算法（队列元素 int 型）

答案：

1．分析：

（1）通常有两条规则。第一是给定序列中 S 的个数和 X 的个数相等；第二是从给定序列的开始，到给定序列中的任一位置，S 的个数要大于或等于 X 的个数。

（2）可以得到相同的输出元素序列。例如:输入元素为 A,B,C 则两个输入的合法序列 A,B,C 和 B,A,C 均可得到输出元素序列 A,B,C。对于合法序列 A,B,C，我们使用本题约定的 SXSXSX 操作序列；对于合法序列 B,A,C，我们使用 SSXXSX 操作序列。

（3）由（1）中分析可以写出以下代码：

```
int Judge(char ch[]) {
    /*判断字符数组 ch 中序列是否是合法序列。如是，返回 1 否则返回 0。*/
    int i=0;
    int I=0,O=0; //I 和 O 分别为字符'I'和'O'的个数。
    while(ch[i]!='\0') {
        if(ch[i]=='I')
            I++;
        if(ch[i]=='O')
            O++;
        if(O>I)
            return 0;   //扫描过程中出现 O 的个数大于 I 的情况，则一定不合法。
        i++;
    }
    if(I!=O)    //I 的总数和 O 不相等，不合法，返回 0。
        return 0;
    else
        return 1;   //合法返回 1。
}
```

2.分析：

C,D 最先出栈，且 C 先与 D，则 A,B 的相对顺序已经确定，必为 B 先与 A。E 的位置可以在 B 前，B 与 A 中间，或者 A 后。因此所得序列有三种，如下：

C,D,E,B,A C,D,B,E,A C,D,B,A,E

3.分析：

用一维数组 `data[0……MAX-1]` 实现循环队列，其中 MAX 是队列长度。设置队头指针 front 和队尾指针 rear，约定 front 指向队头元素的前一位置，rear 指向队尾元素。定义满足 `front==rear` 时为队空。从队尾删除元素，则 rear 向着下标减小的方向行走，从队头插入元素，front同样向着下标减小的方向行走，因 此 当 满 足 `rear==(front-1+MAX)%MAX` 时队满。

（1）队列的结构定义

```
typedef struct {
    int data[MAX]; //假设 MAX 为已定义常量。
    int front,rear;
}cycqueue;
```

（2）算法实现：

①出队算法

```
int dequeue(cycqueue &Q,int &x) {
    /*本算法实现“从队尾删除”，若删除成功用 x 接纳删除元素，返回 1，否则返回 0。*/
    if(Q.front==Q.rear) //队空无法出队，返回 0。
        return 0;
    else {
        x=Q.data[Q.rear];
        Q.rear=(Q.rear-1+MAX)%MAX;  //修改队尾指针。
        return 1;   //出队成功，返回 1。
    }
}
```

②入队算法

```
int enqueue(cycqueue &Q,int x) {
    /*本算法实现“从队头插入”元素 x。*/
    if (Q.rear==(Q.front-1+MAX)%MAX) //队满
        return 0;
    else {
        Q.data[Q.front]=x; //x 入队列。
        Q.front=(Q.front-1+MAX)%MAX; //修改队头指针。
    }
}
```

说明：本题算法中用到了一个操作:`Q.front=(Q.front-1+MAX)%MAX;` 如果把这一句放在一个循环中，front 指针则沿着 Max-1,Max-2, ……2,1,0,Max-1,Max-2…… 的无限循环数行走，这个操作和 `Q.front=(Q.front+1)%MAX;` 实现的效果正好相反。这两个操作在程序设计题目中是很常用的。

4.分析：

本题为循环队列基本算法操作的扩展。在队列结构定义中加入 tag。用 tag 判断队列是否为空，用 front==rear 判断队满。具体过程如下：

队列的结构定义：

```
typedef struct {
    int data[MAX];//假设 MAX 为已定义的常量。
    int front,rear;
    int tag;
}Queue;
```

定义一个队列 Queue qu;

队列的各要素：

初始时 `qu.tag=0; qu.front=qu.rear;`

队空条件 `qu.front=qu.rear&&qu.tag==0;`

队满条件 `qu.front=qu.rear&&qu.tag==1;`

算法描述：

```
void InitQueue(Queue &qu) { //初始化队列。
    qu.front=qu.rear=0;
    qu.tag=0;
}
int QueueEmpty(Queue qu) { //判断队是否为空。
    if(qu.front==qu.rear&&qu.tag==0)
        return 1;
    else
        return 0;
}
int QueueFull(Queue qu) { //判断队是否为满。
    if(qu.tag==1&&qu.front==qu.rear)
        return 1;
    else
        return 0;
}
int enQueue(Queue &qu,int x) { //元素进队。
    if(QueueFull(qu)==1)
        return 0;
    else {
        qu.rear=(qu.rear+1)%MAX;
        qu.data[qu.rear]=x;
        qu.tag=1;   //只要进队就把 tag 设置为 1。
        return 1;
    }
}
int deQueue(Queue &qu,int &x) { //元素出队
    if(QueueEmpty(qu)==1)
        return 0;
    else {
        qu.front=(qu.front+1)%MAX;
        x=qu.data[qu.front];
        qu.tag=0;   //只要有元素出队，就把 tag 设置为 0。
        return 1;
    }
}
```

说明：对于 tag 值的设置，初始时一定为 0。插入成功后应设置为 1，删除成功后应设置为 0；因为只有在插入操作后队列才有可能满，在删除操作后队列才有可能空。tag 的值再配合上 `front==rear` 这一句的判断就能正确区分队满与队空。