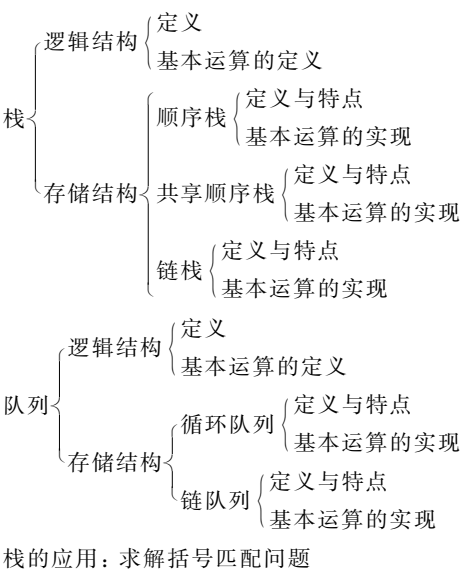
第3章 栈和队列

基本知识点：栈和队列的逻辑结构；栈和队列两种存储结构以及在两种存储结构下基本运算的实现和分析；使用栈求解括号匹配问题。

学习重点：栈和队列的定义和特点；顺序栈和链栈基本运算的实现和分析；循环队列和链队列基本运算的实现和分析。

学习难点：循环队列基本运算的实现；栈和队列的应用。

3.1 本章知识体系结构



栈的应用：求解括号匹配问题

3.2 本章主要数据结构类模板的声明

3.2.1 顺序栈

栈的顺序存储结构简称为顺序栈(sequential stack)，它是操作受限的顺序表。类似于顺序表的定义，顺序栈也用一维数组来实现。因为栈底位置是固定不变的，所以可以将栈底位置设置在数组两端的任意一个端点。在栈的使用过程中，栈顶位置是随着进栈和退栈操作而变化的，故需用一个整型量 top 来指示当前栈顶位置，通常称 top 为栈顶指针。因此，

顺序栈的类型定义只需将顺序表的类型定义中的长度属性改为 top 即可。

```
//顺序栈类模板声明
const int MaxStackSize = 20;                    //栈最大可容纳元素的个数
template < typename T >
class SeqStack
{
    public:
        SeqStack(T a[ ], int n = 0);
        ~SeqStack();
        int IsEmpty();                          //判断栈是否为空
        int IsFull();                           //判断栈是否已满
        SeqStack < T > & Push(T x);             //进栈
        SeqStack < T > & Pop(T& x);             //退栈
        T GetTop();                             //取栈顶元素
        void ShowStack();                       //输出栈
    private:
        T data[MaxStackSize];                   //存放栈中元素的数组
        int top;                                //栈顶指针
};
```

3.2.2 共享顺序栈

栈的使用非常广泛，常常会出现在一个程序中需要同时使用多个栈的情形。为了不因栈上溢而产生错误中断，必须给每个栈顶分配一个较大空间。但这并不容易做到，因为各个栈实际所用最大空间很难估计。而另一方面，各个栈的实际容量在使用期间是在变化的，往往会有这样的情况，即其中某一个栈发生上溢，而另一个栈还是空的。试设想：若令多个栈共享空间，则将提高空间的使用效率，并减少发生栈上溢的可能性。

假设程序同时需要编号为 1 和 2 的两个栈。我们可以将两个栈存放于一个数组空间 data[MaxStackSize]中，栈底分别处于数组的两端，下面给出了这种双栈结构的类模板声明。

```
//双栈类模板的声明
const int MaxStackSize = 15;                    //栈的总容量
template < typename T>
class DSeqStack
{
    public:
        DSeqStack(T a1[ ],T a2[ ], int n1 = 0,int n2 = 0);
        //初始化双栈,两个栈的初始长度分别为 n1 和 n2
        ~DSeqStack();                           //清空栈的内容
        bool IsEmpty(int i = 0);                //判断栈 i 空否,空返回 1,否则返回 0,i = 0 表示双栈
        bool IsFull();                          //判断栈满否,满返回 1,否则返回 0
        DSeqStack < T > & Push(T x,int i);      //把 x 插到栈 i 的栈顶
        DSeqStack < T > & Pop(int i,T& x);      //退掉栈 i 位于栈顶的元素
        T GetTop(int i);                        //返回栈 i 位于栈顶的元素
        void ShowStack();                       //输出双栈中的元素
    private:
        int top[2];                             //栈顶指针
```

```
    T data[MaxStackSize];              //栈数组
};
```


3.2.3 链栈

栈的链式存储结构称为链栈(linked stack),它是操作受限的单链表,其插入和删除操作仅限制在表头位置上进行。由于只能在链表头部进行操作,故链栈没有必要像单链表那样附加头结点。栈顶指针就是链表的头指针。

```
//链栈类模板的声明
template<typename T>
class LinkStack
{
    public:
        LinkStack(T a[],int n=0);
        ~LinkStack();
        bool IsEmpty();
        LinkStack<T>& Push(T x);
        LinkStack<T>& Pop(T& x);
        T GetTop();
        void ShowStack();
    private:
        Node<T> *top;
};
```

3.2.4 循环队列

队列的顺序存储结构称为顺序队列,顺序队列实际上是操作受限的顺序表,和顺序表一样,顺序队列也是用一个大小为 MaxQueueSize 的一维数组来存放当前队列中的元素。由于队列的队头和队尾的位置是变化的,因而要设置两个整型变量 front 和 rear 分别指示队头元素和队尾元素在数组空间中的位置,并称为队头指针和队尾指针。

解决顺序队列假溢出的方法是把数组的前端和后端连接起来,形成一个环形的顺序表,即把存储队列元素的表从逻辑上看成一个环,队列的这种头尾相接的顺序存储结构称为循环队列(circular queue)。“队满”和“队空”的条件都是 front==rear,故无法通过此条件判断队列是“空”还是“满”。

可以用下面介绍的两种方法之一来解决此问题。

- 使用一个计数器 count 记录队列中元素的总数(即队列长度)。当 count==0 时队空,当 count==MaxQueueSize 时为队满。
- 少用一个元素空间,即把如图 3.1 所示的情况就视为队满,此时的状态是队尾指针加 1 就会从后面赶上队头指针,这种情况下队满和队空的条件分别为:

队满条件:(rear+1)%MaxQueueSize==front

队空条件:front==rear

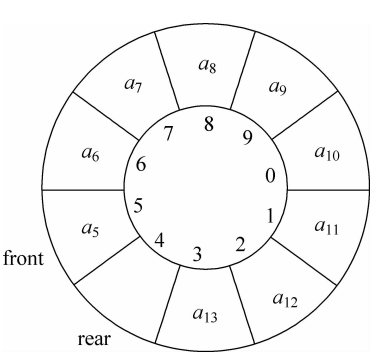


图 3.1 队满示意图

循环队列利用第一种方法判断队空和队满的类模板声明如下。

```
//顺序循环队列利用第一种方法判断队空和队满的类模板声明
const int MaxQueueSize = 20;
template < typename T >
class SeqQueue
{
    public:
        SeqQueue(T a[ ], int n = 0);          //构造函数,创建初始队列
        ~SeqQueue();                          //析构函数,清空队列
        int IsEmpty();                        //判断队列空否
        int IsFull();                         //判断队列满否
        SeqQueue < T > & EnQueue(T x);        //入队
        SeqQueue < T > & DeQueue(T& x);       //出队
        T GetQueueFront();                    //取队头元素
        void ShowQueue();                     //显示队列中的元素
    private:
        T data[MaxQueueSize];
        int front;
        int rear;
        int count;
};
```

循环队列利用第二种方法判断队空和队满的类模板声明时,不再需要计数器 count。

```
//顺序循环队列利用第二种方法判断队空和队满的类模板声明
const int MaxQueueSize = 20;
template < typename T >
class SeqQueue
{
    public:
        SeqQueue(T a[ ], int n = 0);          //构造函数,创建初始队列
        ~SeqQueue();                          //析构函数,清空队列
        int IsEmpty();                        //判断队列空否
        int IsFull();                         //判断队列满否
        SeqQueue < T > & EnQueue(T x);        //入队
        SeqQueue < T > & DeQueue(T& x);       //出队
        T GetQueueFront();                    //取队头元素
        void ShowQueue();                     //显示队列中的元素
    private:
        T data[MaxQueueSize];
        int front;
        int rear;
};
```

3.2.5 链队列

队列的链接存储结构称为链队列(linked queue)。链队列是在单链表的基础上做了简单的修改。为了使空队列和非空队列的操作一致,链队列也加上头结点。根据队列的先进先出原则,为了操作上的方便,设置队头指针指向链队列的头结点,队尾指针指向终端结点,

即队尾结点。

```
//链队列的类模板声明
template<typename T>
class LinkQueue
{
    public:
        LinkQueue(T a[],int n=0);          //构造函数,创建初始队列
        ~LinkQueue();                      //析构函数,清空队列
        bool IsEmpty();                    //判断队列空否
        LinkQueue<T>& EnQueue(T x);        //入队
        LinkQueue<T>& DeQueue(T& x);       //出队
        T GetQueueFront();                 //取队头元素
        void ShowQueue();                  //显示队列中的元素
    private:
        Node<T> *front, *rear;             //队头和队尾指针,分别指向头结点和尾结点
};
```

3.3 主教材中的全部习题和参考解答

一、名词解释

栈 顺序栈 链栈 队列 顺序队列 循环队列 链队列

【问题解答】

栈是限定只在表的端进行插入、删除操作的特殊线性表。

栈的顺序存储结构简称为顺序栈。

栈的链式存储结构称为链栈。

队列也是一种特殊的线性表,是限制在表的一端进行插入和在另一端进行删除的线性表。

队列的顺序存储结构称为顺序队列。

把存储队列元素的表从逻辑上看成一个环,队列的这种头尾相接的顺序存储结构称为循环队列。

队列的链接存储结构称为链队列。

二、填空题

1. 线性表、栈和队列都是_____结构,可以在线性表的_____位置插入和删除元素;对于栈只能在_____插入和删除;对于队列只能在_____分别插入元素和删除元素。

【问题解答】线性　任意　表的端　表尾和表头

2. 同一栈内的各元素的类型_____。

【问题解答】必须一致

3. 对于一个顺序栈来说,进行进栈运算时,应先判别栈是否为_____,进行退栈运算时,应先判别栈是否为_____,当栈中元素个数为 n 时,进行进栈运算时发生上溢,则说明栈的可用最大容量为_____。为了增加内存空间的利用率和减少发生上溢的可能性,由

两个栈共享一片连续的内存空间时，应将两个栈的_____分别设在这片内存空间的两端，这样只有当_____时才产生上溢。

【问题解答】满　空　n　栈底　两个栈顶相遇

4. 设有一个空栈，现有输入序列 1,2,3,4,5，执行 Push,Push,Pop,Push,Pop,Push,Push 的操作，当执行 Pop 操作时，立即输出所 Pop 出来的元素，则输出的序列是_____。

【问题解答】2,3

【结果分析】由于栈具有先进后出的特点，首先 Push,Push,Pop 后，2 出栈，同时被输出，接着 Push,Pop，此时 3 出栈，同时被输出，最后 Push,Push，没有输出。所以输出的序列是 2,3。

5. 设有元素序列的进栈次序为：$(a_1,a_2,\cdots,a_n)$，其退栈次序为：$(a_{p1},a_{p2},\cdots,a_{pm})$，现已知 $p_1=n$，则 $p_i=$_____。

【问题解答】$n-i+1$

【结果分析】由于栈具有先进后出的特点，当 $p_1=n$，即 a_n 是最先退栈的，则 a_n 必定是最后入栈的，那么元素序列的进栈次序为$(a_1,a_2,\cdots,a_n)$，则退栈次序为$(a_n,a_{n-1},\cdots,a_1)$。

6. 无论对于顺序存储还是链式存储的栈和队列来说，进行插入或删除运算的时间复杂度相同，均为_____。

【问题解答】$O(1)$

【结果分析】栈只能在栈顶进行插入或删除运算，队列只能在队尾和队头分别进行插入和删除运算，这样均不需要查找，故时间复杂度均为 $O(1)$。

7. 队列是限制插入只能在表的一端，而删除在表的另一端进行的线性表，其特点是_____。

【问题解答】先进先出

8. 队列元素个数是_____，元素进入队列的那端是_____，元素离开队列的那端是_____。

【问题解答】可变的　队尾　队头

9. 对于循环队列，在进行进队操作之前要判断_____，在进行出队操作之前要判断_____。

【问题解答】队满否　队空否

10. 在带头结点的链队 lq 中，指向队头元素的指针是_____，指向队尾元素的指针是_____，队空的条件是_____。

【问题解答】lq->front->next　lq->rear　lq->front= =lq->rear

三、选择题

1. 判定一个顺序栈 ST 为(元素最多为 MaxStackSize)空的条件是(　　)。

A. ST. top = =-1　　B. ST. top ! = -1

C. ST. top ! = MaxStackSize　　D. ST. top = = MaxStackSize

【问题解答】A

2. 往顺序栈中压入一个元素时，栈顶指针是(　　)。

A. 加 1　　B. 减 1　　C. 不变　　D. 清 0

【问题解答】A

3. 当栈中的元素为 n 个,进行进栈操作时发生上溢,则说明该栈的最大容量是(　　)。

A. $n/2$　　B. n　　C. $n+1$　　D. $n-1$

【问题解答】B

4. 若进栈序列为 1,2,3,4,进栈的过程中可以退栈,则(　　)不可能是一个退栈序列。

A. 3421　　B. 2431　　C. 1423　　D. 3214

【问题解答】C

【结果分析】由于栈具有先进后出的特点,若进栈序列为 1234,退栈序列不可能是 1423,这是因为若 4 在 2、3 前退,当时 2、3 必定还在栈中,且 2 在 3 之前进栈,则 3 应在 2 之前出栈。

5. 若栈的输入序列为 a,b,c,d,e,f,g,进栈的过程中可以退栈,则当栈空时,输出的元素序列可以是下列(　　)。

A. d,e,f,b,c,g,a　　B. f,e,g,d,a,c,b

C. e,f,d,g,b,c,a　　D. c,d,b,e,f,a,g

【问题解答】D

【结果分析】由于栈具有先进后出的特点。若进栈序列为 abcdefg,退栈序列不可能是选项 ABC,这是因为若 d 在 a、b、c 前退栈,当时 a、b、c 必定还在栈中,则它们的退栈序列必定是 cba。

6. 链栈与顺序栈相比,有一个比较明显的优点是(　　)。

A. 插入操作更加方便　　B. 通常不会出现栈满的情况

C. 不会出现栈空的情况　　D. 删除操作更加方便

【问题解答】B

7. 如果以链表作为栈的存储结构,则退栈操作时(　　)。

A. 需要判别栈是否为空　　B. 对栈不做任何判别

C. 需要判别栈是否为满　　D. 判别栈元素的类型

【问题解答】A

8. 队列结构属于下列结构中的(　　)。

A. 集合　　B. 线性　　C. 树状　　D. 网状

【问题解答】B

9. 栈和队列的共同点是(　　)。

A. 都是先进后出　　B. 都是先进先出

C. 只允许在端点处插入和删除元素　　D. 没有共同点

【问题解答】C

【结果分析】栈和队列都是受限的线性表,所谓“受限”指的是只允许在端点处插入和删除元素。

10. 队列是限定在(　　)处进行插入操作、在(　　)处进行删除操作的线性表。

A. 任意端点　　B. 队头　　C. 队尾　　D. 中间

【问题解答】C B

11. 4 个元素按 A、B、C、D 顺序连续进入队列 Q,队列的头元素是(　　),尾元素是(　　)。

A. A　　B. B　　C. C　　D. D

【问题解答】A D

12. 循环队列占用的空间(　　)。

A. 必须连续　　B. 可以不连续　　C. 不能连续　　D. 不必连续

【问题解答】A

13. 一个循环队列一旦说明,其占用空间的大小(　　)。

A. 已固定　　B. 可以改变　　C. 不能固定　　D. 动态变化

【问题解答】A

14. 判定一个循环队列 Q(存放元素位置 0～MaxQueueSize－1)队满的条件是(　　)。

A. Q. front = = Q. rear

B. Q. front +1 = =Q. rear

C. Q. front = = (Q. rear + 1) % MaxQueueSize

D. Q. rear = = (Q. front +1) % MaxQueueSize

【问题解答】C

【结果分析】根据循环队列的结构,很快可以排除 A 和 B,因为它们与 MaxQueueSize 无关,而为了将循环队列队满和队空的条件区分开,一般采用少用一个空间,即假定队尾指针加 1 就会从后面赶上队头指针时为队满。

15. 容量是 10 的循环队列的头指针的位置为 2,则队列的头元素的位置是(　　),若尾指针的位置为 2,则队列的尾元素的位置是(　　)。

A. 2　　B. 3　　C. 1　　D. 0

【问题解答】A C

【结果分析】主教材中约定:在非空队列里,队头指针指向队头元素,而队尾指针 rear 指向队尾元素的下一个位置。

16. 对链队列进行入队、出队操作时,其尾指针 rear(　　)。

A. 始终不改变　　B. 有时改变　　C. 入队时改变　　D. 出队时改变

【问题解答】C

【结果分析】队列限定从队尾入队,队头出队,所以入队时需要改变尾指针。

17. 关于链队列,下列说法中(　　)是正确的。

A. 存在队满的情况　　B. 不存在队空的情况

C. 出队之前先判断空否　　D. 入队之前必须判断满否

【问题解答】C

18. 设计一个判别表达式中左右括号是否配对出现的算法,采用(　　)数据结构最好。

A. 线性表　　B. 队列　　C. 二叉树　　D. 栈

【问题解答】D

19. 设长度为 n 的链队列用单循环链表表示,若只设头指针,则入队操作的时间复杂度是(　　)。

A. $O(n)$　　B. $O(n\log_2 n)$　　C. $O(1)$　　D. $O(\log_2 n)$

【问题解答】A

【结果分析】队列限定只能从队尾入队,若只设头指针,则首先需要找到队尾,而单循环链表查找的时间复杂度为 $O(n)$。

20. 递归过程或函数调用时,处理参数及返回地址,要用一种称为(　　)的数据结构。

A. 队列　　B. 多维数组　　C. 栈　　D. 线性表

【问题解答】C

四、简答题

1. 将内存中一片连续空间(不妨设地址从 1 到 n),提供给两个栈 S1 和 S2 共享。如何分配这部分空间,使得对任何一个栈,只有当这部分空间全满时才发生溢出。

【问题解答】为这两个栈分配空间的适当的方案是:S1 的栈底位置设为 1,S2 的栈底位置设为 n,即两个栈的栈顶相向而对。

2. 假设以 S 和 X 分别表示进栈和退栈操作,则对初态和终态均为空的栈操作可由 S 和 X 组成的序列表示(如 SXSX),试指出判别给定序列是否合法的一般规则。

【问题解答】合法的栈运算序列必须满足下列两个条件。

(1) 对于运算序列的任何前缀序列(从开始到任何一个操作时刻),S 的个数不得少于 X 的个数。

(2) 整个操作序列中 S 和 X 的数目相等。

3. 何谓队列的上溢现象和假溢出现象?解决它们有哪些方法?

【问题解答】队列的上溢现象是指所使用的存储空间已满,这时再有新元素入队的情况。队列的假溢出指的是由于队列运算算法原因造成虽然队列空间还有空闲的元素空间,但根据算法已无法继续使用这些空间在队列中插入新元素。

避免这一现象发生的方法可以有以下几种。

① 每一次出队列操作后将队列元素向左平移一位,并使 front 始终指向队列的存储区的起始位置、rear 指向队尾。

② 每当溢出发生时,判断 front 的值是否指向首位,如果否,则将队列中元素依次左移,并使得 front 指向起始位置、rear 指向队尾。

③ 采取循环队列。将队列的存储区视为首尾相接的环。

4. 设长度为 n 的链队列用单循环链表表示,若设头指针,则入队出队操作的时间复杂度为何?若只设尾指针呢?

【问题解答】队列限定只能从队尾入队,若只设头指针,首先需要找到队尾,而单循环链表查找的时间复杂度为 $O(n)$,则入队出队操作的时间复杂度为 $O(n)$,显然若只设尾指针,则入队出队操作的时间复杂度为 $O(1)$。

5. 指出下列程序段的功能是什么。

(1)
```
void Test_1(SeqStack *S)
{
int i, arr[64], n=0;
while(!S->IsEmpty())  S->Pop(&arr[n++]);
for(i=0, i<n; i++)  S->Push(arr[i]);
}
```

【问题解答】当栈不为空时,将栈中元素逆置。

(2)
```
void Test_2(SeqStack *S,int m)
{
int a[20],x;
SeqStack T(a);
```

```
while(!S->IsEmpty())
{
S->Pop(x);
if((x!= m)  T.Push(x);
}
while(!T.IsEmpty())
{
T.Pop(x);
S->Push(x);
}
}
```

【问题解答】删除栈 S 中的元素 m。

(3)
```
void Test_3 (SeqQueue *Q)
{
int x,a[20];
SeqStack S(a);
while (!Q->IsEmpty())
{
Q->DeQueue(x);
S.Push(x);
}
while (!S.IsEmpty())
{
S.Pop(x);
Q->EnQueue(x );
}
}
```

【问题解答】将队列 Q 中的元素倒排。

(4)
```
SeqQueue Q1, Q2;          //队列元素为 int 型
int x, i, n = 0;
…                         //初始化 Q2,Q1 元素入队
while (!Q1.IsEmpty())
{
Q1.DeQueue(x);
Q2.EnQueue(x);
n++;
}
for (i = 0; i< n; i++)
{
Q2. DeQueue(x);
Q1. EnQueue(x);
Q2. EnQueue(x);
}
```

【问题解答】复制队列 $Q1$ 到 $Q2$。

五、算法设计题

1. 写一个函数,利用栈运算,读入一行文本,并以相反的次序输出。

【问题解答】下面的 Invert()函数是算法的实现。函数 Invert()使用堆栈 s,栈中元素类型 T 定义为 char 类型。

```
//XiTi3_5_1.cpp
void Invert()
{
    char a[120];
    SeqStack<char> s(a,0);                          // 栈的初始长度为 0
    cout<<"Input a string,'#' for the end."<<endl;
    for(char e=getchar();e!='#';e=getchar())        //输入一串文本,以字符'#'结束
        s.Push(e);                                  //每输入一个字符,立即将该字符进栈
    cout<<"The Inverted string is as below.\n";
    while(!s.IsEmpty())                             //输出栈中字符,直到堆栈为空
        {
        s.Pop(e);
        cout<<e;
        }
    cout<<endl;
}
```

2. 利用栈的基本操作,写一个将栈 S 中所有结点均删去的算法。

【问题解答】下面的 ClearStack()函数是算法的实现。

```
//XiTi3_5_2.cpp
template<typename T>
void ClearStack(SeqStack<T>& S)
{
    T x;
    while(!S.IsEmpty())
        S.Pop(x);
}
```

3. 用第二种方法,即少用一个元素空间的方法来区别循环队列的队空和队满,试为其设计置空队、判队空、判队满、出队、入队及取队头元素 6 个基本操作的算法。

【问题解答】下面是 6 个基本操作的算法实现。

```
//XiTi3_5_3.cpp
//置空队
template<typename T>
SeqQueue<T>::SeqQueue()
{
    front=0;
    rear=0;
}

//判队空
template<typename T>
int SeqQueue<T>::IsEmpty()
{
    return (front==rear);
}

//判队满
template<typename T>
```

```
int SeqQueue<T>::IsFull()
{
    return ((rear+1)%MaxQueueSize==front);
}

//入队
template<typename T>
SeqQueue<T>& SeqQueue<T>::EnQueue(T x)
{
    if(IsFull())
    {
        cout<<"队满,终止运行!"<<endl;
        exit(1);
    }
    data[rear]=x;                                    //新元素插入队尾
    rear=(rear+1)%MaxQueueSize;                      //循环意义下将队尾指针加1
    return *this;
}

//出队
template<typename T>
SeqQueue<T>& SeqQueue<T>::DeQueue(T& x)
{
    if(IsEmpty())
    {
        cout<<"队空,终止运行!"<<endl;
        exit(1);
    }
    x=data[front];
    front=(front+1)%MaxQueueSize;                    //循环意义下将队头指针加1
    return *this;
}

//取队头元素
template<typename T>
T SeqQueue<T>::GetQueueFront()
{
    if(IsEmpty())
    {
        cout<<"队空,终止运行!"<<endl;
        exit(1);
    }
    return data[front];
}
```

4. 假设以带头结点的循环链表表示队列,并且只设一个指针指向队尾元素结点(注意不设头指针),试编写相应的置空队、判队空、入队和出队等算法。

【问题解答】首先要声明一个带头结点的循环链表表示队列的类模板 RearCLinkQueue,然后实现相应的算法。

```
//XiTi3_5_4.cpp
//定义一个带头结点的循环链表表示队列的类模板
struct Node
{
    T data;
    Node * next;
};
template<typename T>
class RearCLinkQueue
{
    public:
        RearCLinkQueue();                          //构造函数,置空队
        ~RearCLinkQueue();                         //析构函数,清空队列
        bool IsEmpty();                            //判断队列空否
        RearCLinkQueue<T>& EnQueue(T x);           //入队
        RearCLinkQueue<T>& DeQueue(T& x);          //出队
        T GetQueueFront();                         //取队头元素
        void ShowQueue();                          //显示队列中的元素
    private:
        Node<T> * rear;                            //队尾指针,指向尾结点
};

//构造函数,置空队
template<typename T>
RearCLinkQueue<T>::RearCLinkQueue()
{
    rear = new Node<T>;
    rear->next = rear;
}

//判断队列空否
template<typename T>
bool RearCLinkQueue<T>::IsEmpty()
{
    return (rear == rear->next);
}

//入队
template<typename T>
RearCLinkQueue<T>&RearCLinkQueue<T>::EnQueue(T x)
{
        Node<T> * p;
        p = new Node<T>;                           //申请新结点
        p->data = x;                               //新结点的数据域
        p->next = rear->next;                      //新结点的指针域
        rear->next = p;                            //新结点插入到队尾
        rear = p;                                  //队尾指针指向新结点
        return * this;                             //返回新队列
}

//出队
```

```
template<typename T>
RearCLinkQueue<T>&RearCLinkQueue<T>::DeQueue(T& x)
{
    if(IsEmpty())
    {
        cout<<"队空,不能进行出队操作!"<<endl;
        exit(1);
    }
    Node<T> *p;
    p=rear->next->next;
    x=p->data;                                  //保存结点中的数据
    rear->next->next=p->next;                   //将队头元素摘链
    if (p==rear) rear=rear->next;               //是否只剩最后一个元素
    delete p;
    return *this;
}
```

5. 设计链式队列的链表结构,要求只设置一个队列指针 q,且能使得入队和出队运算的时间复杂度均为 $O(1)$。简述该链式队列结构,给出其类型定义,并实现入队和出队函数。

【问题解答】 根据题意,这里的链式队列可以复用上面第 4 题的 RearCLinkQueue,只需将队尾指针 rear 改为 q 即可。

```
//XiTi3_5_5.cpp
//定义只设置一个队列指针 q 的链式队列的类模板
template<typename T>
struct Node
{
    T data;
    Node *next;
};
template<typename T>
class QCLinkQueue
{
    public:
        QCLinkQueue();                          //构造函数,置空队
        ~QCLinkQueue();                         //析构函数,清空队列
        bool IsEmpty();                         //判断队列空否
        QCLinkQueue<T>& EnQueue(T x);           //入队
        QCLinkQueue<T>& DeQueue(T& x);          //出队
        T GetQueueFront();                      //取队头元素
        void ShowQueue();                       //显示队列中的元素
    private:
        Node<T> *q;                             //队尾指针,指向尾结点
};

//入队
template<typename T>
QCLinkQueue<T>&QCLinkQueue<T>::EnQueue(T x)
{
        Node<T> *p;
```

```
        p = new Node<T>;                          //申请新结点
        p->data = x;                              //新结点的数据域
        p->next = q->next;                        //新结点的指针域
        q->next = p;                              //新结点插入到队尾
        q = p;                                    //队尾指针指向新结点
        return *this;                             //返回新队列
}

//出队
template<typename T>
QCLinkQueue<T> &QCLinkQueue<T>::DeQueue(T& x)
{
    if(IsEmpty())
    {
        cout<<"队空,不能进行出队操作!"<<endl;
        exit(1);
    }
    Node<T> *p;
    p = q->next->next;
    x = p->data;                                  //保存结点中的数据
    q->next->next = p->next;                      //将队头元素摘链
    if (p==q) q = q->next;                        //是否只剩最后一个元素
    delete p;
    return *this;
}
```