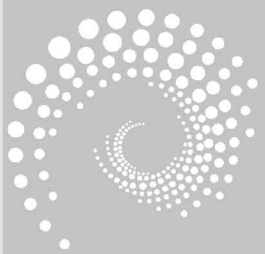
第3章

栈和队列

CHAPTER 3



本章学习目标

- 理解栈和队列的基本概念
- 掌握顺序栈和链栈的各种操作实现
- 掌握链队列和循环队列的各种操作实现
- 学会利用栈和队列解决应用问题

栈和队列是非常重要的两种数据结构，在软件设计中的应用很多。栈和队列也是线性结构，线性表、栈和队列这 3 种数据结构的数据元素以及数据元素间的逻辑关系完全相同，区别在于线性表的操作不受限制，而栈和队列的操作受到限制。栈的操作只能在表的一端进行；队列的插入操作在表的一端进行，而其他操作在表的另一端进行。因此，将栈和队列称为操作受限的线性表。

视频讲解

3.1 栈的基本概念

3.1.1 栈的相关定义

1. 栈

栈(stack)是只允许在表尾端进行插入和删除的线性表。

在插入数据元素时,新插入的数据元素 e 只能处于线性表的表尾,如图 3-1 所示。

在删除数据元素时,只能删除线性表的表尾元素,如图 3-2 所示。

$$(a_1,a_2,\cdots,a_n)\Longrightarrow(a_1,a_2,\cdots,a_n,e)$$

图 3-1 插入数据元素

$$(a_1,a_2,\cdots,a_{n-1},a_n)\Longrightarrow(a_1,a_2,\cdots,a_{n-1})$$

图 3-2 删除数据元素

2. 栈顶和栈底

表尾端为栈顶(top),表头端为栈底(bottom)。

3. 进栈和出栈

栈的插入操作称为入栈或进栈(push),栈的删除操作称为出栈或退栈(pop)。

4. LIFO

最后入栈的数据元素最先出栈,最先入栈的数据元素最后出栈。因此,栈也被称为“后进先出”(Last In First Out,LIFO)的线性表。

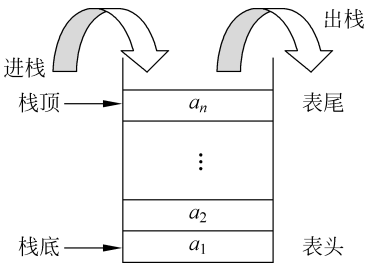


图 3-3 栈的图示

图 3-3 为栈的图示。其中,a_n 是最后入栈的数据元素,a_n 最先出栈;a_1 是最先入栈的数据元素,a_1 最后出栈。

在实际生活中有许多类似于栈的例子。例如,刷洗盘子,将洗净的盘子一个接一个地往上放(相当于将元素入栈);在取用盘子时,则从最上面一个接一个地往下拿(相当于将元素出栈)。

3.1.2 栈的抽象数据类型

ADT Stack{
数据对象: $D=\{a_i \mid a_i\in \text{ElemSet}, i=1,2,\cdots,n, n\geqslant 0\}$
数据关系: $R=\{<a_{i-1},a_i> \mid a_{i-1},a_i\in D, i=2,\cdots,n\}$
基本操作:
(1) int Count(): 求栈的长度。
操作结果: 返回栈中数据元素的个数。
(2) boolean IsEmpty(): 判断栈是否为空。
操作结果: 如果栈为空,则返回 TRUE,否则,返回 FALSE。
(3) Clear(): 清空操作。
操作结果: 使栈为空。
(4) Push(*T* item): 入栈操作。
操作结果: 将值为 item 的新的数据元素添加到栈顶,栈发生变化。
(5) *T* Pop(): 出栈操作。
初始条件: 栈不为空。

操作结果：将栈顶元素从栈中取出，栈发生变化。
(6) T GetTop()：取栈顶元素。
初始条件：栈不为空。
操作结果：返回栈顶元素的值，栈不发生变化。
}

关于栈抽象数据类型中 D 的说明：D 是 n 个数据元素的集合，D 是 ElemSet 的子集。ElemSet 表示某个集合，集合中的数据元素类型相同，如整数集、自然数集等。

关于栈抽象数据类型中 R 的说明：R 是数据元素之间关系的集合，$<a_{i-1},a_i>$表示 a_{i-1} 和 a_i 之间互为前驱和后继的逻辑关系。约定 a_n 端为栈顶，a_1 端为栈底。

栈的数据对象和数据关系与线性表相同。

关于栈抽象数据类型中基本操作的说明：

(1) 栈的数据元素类型用 T 表示，T 可以是原子类型，也可以是结构类型。整型用 int 表示，逻辑型用 boolean 表示。

(2) 栈的基本操作是定义于逻辑结构上的基本操作，是向使用者提供的使用说明。基本操作只有在存储结构确定之后才能实现。如果栈采用的存储结构不同，则栈的基本操作实现算法也不相同。

(3) 基本操作的种类和数量可以根据实际需要决定。但是，栈是操作受限的线性表，不能任意地定义基本操作。例如，不能在栈的第 i 个位置插入数据元素。

(4) 基本操作名称，形式参数数量、名称、类型，返回值类型等由设计者决定。使用者根据设计者的设计规则使用基本操作。

在 Java 程序设计语言中，用接口表示栈的抽象数据类型如下。

```
interface IStack<T> {
    int Count();              // 求栈的长度
    boolean IsEmpty();        // 判断栈是否为空
    void Clear();             // 清空操作
    void Push(T item);        // 入栈操作
    T Pop();                  // 出栈操作
    T GetTop();               // 取栈顶元素
}
```

与线性表相同，栈也有两种存储方式：顺序存储和链式存储。

视频讲解

3.2　栈的顺序存储

3.2.1　栈的顺序存储定义

将栈的数据元素存放在一组地址连续的存储单元中。假设每个元素占用 l 个存储单元，栈中第一个元素(即栈底元素)的存储地址是 $\mathrm{LOC}(a_1)=b$，那么栈中最后一个元素(即栈顶元素)的存储地址是多少？

栈中第一个元素的存储地址是 b，第二个元素的存储地址是 $b+l$，第三个元素的存储地址是 $b+2l$，…，以此类推，第 n 个元素的存储地址是 $b+(n-1)l$，如图 3-4 所示。栈的顺序

存储结构简称顺序栈。

在 Java 语言的层面上讨论时,栈的顺序存储是将栈的数据元素自栈底至栈顶放在一维数组中。同时,附设一个 top 指示器指向栈顶元素,如图 3-5 所示。top 的值就是栈顶元素在数组中的下标。也可以让 top 指示器指向栈顶元素的下一个位置。

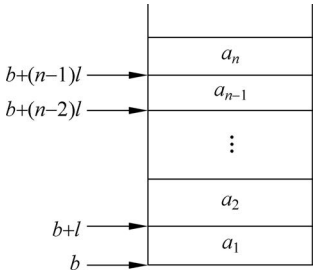


图 3-4 栈中元素的存储地址

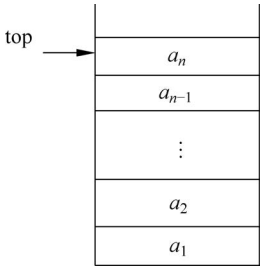


图 3-5 top 指示器

3.2.2 顺序栈基本操作分析

1. 进栈操作

进栈操作即插入元素 e 为新的栈顶元素。栈的插入操作是往栈顶位置插入(即表尾位置),所以不需要移动数据元素,直接插入即可。

进栈操作的步骤如下:

(1) 令 top 值加 1。

(2) 令栈顶元素等于 e。

进栈操作如图 3-6 所示。

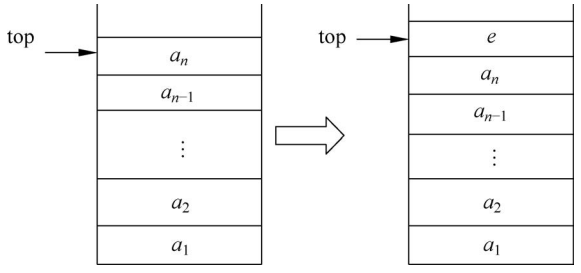


图 3-6 进栈操作

2. 出栈操作

出栈操作即删除 S 的栈顶元素并返回其值。栈的删除操作是删除栈顶元素,所以不需要移动数据元素,直接删除即可。

出栈操作的步骤如下:

(1) 令变量=栈顶元素。

(2) 令 top 值减 1,并返回变量的值。

出栈操作如图 3-7 所示。

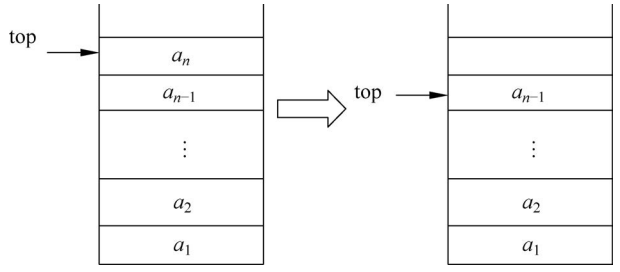


图 3-7　出栈操作

3.2.3　顺序栈源码实现

1. 顺序栈类的实现

(1) 定义顺序栈类 SeqStack < T >,实现接口 IStack < T >。

```
public class SeqStack<T> implements IStack<T>
```

(2) 创建顺序栈类 SeqStack < T >中的属性成员。

```
public class SeqStack<T> implements IStack<T> {
    public T[] data;                    //数组,用于存储顺序栈中的数据元素
    public int maxsize;                 //顺序栈的容量
    public int top;                     //指示顺序栈的栈顶,值是栈顶元素在数组中的下标
}
```

关于属性成员的几点说明:

① Java 语言中的数组在内存中占用的存储空间是一组地址连续的存储单元。因此,在 Java 虚拟处理器中考虑问题时,认为栈的顺序存储就是将栈的数据元素存放到数组中。

② maxsize 表示数组的容量。数组的容量可以用 data 的 Length 属性即 data.length 来表示,但为了说明顺序栈的最大长度(顺序栈的容量),在 SeqStack < T >类中用字段 maxsize 来表示。

③ top 指示顺序栈的栈顶,值是栈顶元素在数组中的下标。

(3) 创建顺序栈类 SeqStack < T >的构造方法。

```
public SeqStack(int size){
    data = (T[])new Object[size];;   //类型为 T、长度为 size 的数组存放栈中元素
    maxsize = size;                  //栈的容量等于数组长度
    top = -1;                        //空栈 top 的值为 -1
}
```

2. 基本操作的实现

(1) 求顺序栈的长度。

由于数组是 0 基数组,即数组的最小索引为 0,因此,顺序栈的长度就是栈中最后一个元素的索引 top 加 1。

求顺序栈长度的算法实现如下。

```
public int Count(){
     return top + 1;
}
```

(2) 清空操作。

清除顺序栈中的数据元素是指使顺序栈为空,此时 top 等于-1。

清空顺序栈的算法实现如下。

```
public void Clear(){
   top = -1;
}
```

(3) 判断顺序栈是否为空。

如果顺序栈的 top 为-1,则顺序栈为空,返回 true; 否则,返回 false。

判断顺序栈是否为空的算法实现如下。

```
public boolean IsEmpty(){
    if (top == -1)
       return true;
    else
       return false;
    }
```

(4) 判断顺序栈是否为满。

如果顺序栈为满,top 等于 maxsize-1,则返回 true; 否则,返回 false。

判断顺序栈是否为满的算法实现如下。

```
public boolean IsFull(){
  if (top == maxsize - 1)
    return true;
  else
    return false;
}
```

(5) 入栈操作。

入栈操作是指在顺序栈未满的情况下,先使栈顶指示器 top 加 1,然后在栈顶添加一个新元素。

入栈操作的算法实现如下。

```
public void Push(T item){
   if(top == maxsize - 1){                  //栈满
      System.out.println("Stack is full");  //提示信息
      return;
   }
   data[++top] = item;                  //先使栈顶指示器 top 加 1,然后在栈顶添加一个新元素
}
```

(6) 出栈操作。

顺序栈的出栈操作是指在栈不为空的情况下,使栈顶指示器 top 减 1。

出栈操作的算法实现如下。

```
public T Pop(){
    T tmp ;                                 //T 类型 tmp
    if (IsEmpty()){
        throw new RuntimeException("栈为空,无法删除" );
      }
  tmp = data[top];                          //栈非空, tmp 等于栈顶元素
  top--;                                    //栈顶指示器减 1
```

```
  return tmp;                                    //返回 tmp
}
```

(7) 取栈顶元素。

如果顺序栈不为空,则返回栈顶元素的值；否则,返回特殊值,表示栈为空。

取栈顶元素操作的算法实现如下。

```
public T Getop(){
   T tmp ;                                       //T 类型 tmp
   if (IsEmpty()){
      throw new RuntimeException("栈为空,无法删除" );
   }
   tmp = data[top];                              //栈非空, tmp 等于栈顶元素
   //没有栈顶指示器减 1
   return tmp;                                   //返回 tmp
}
```

3.2.4　Java 基础类库中的顺序栈

Java 语言的推出者设计开发了顺序栈类 java. util. Stack,供应用程序员使用。在开发软件时,可以利用继承和覆盖技术在 java. util. Stack 的基础上开发新的顺序栈类,以便切合软件的实际功能需求。java. util. Stack 中的部分构造和普通方法如表 3-1 和表 3-2 所示,读者可以结合 java. util. Stack 源代码与本书中的 SeqStack 栈比较。

表 3-1　构造方法及功能说明

构 造 方 法	功 能 说 明
Stack()	创建一个空堆栈

表 3-2　普通方法及功能说明

返回值类型	普 通 方 法	功 能 说 明
boolean	empty()	测试堆栈是否为空
T	peek()	查看堆栈顶部的对象,但不从堆栈中移除它
T	pop()	移除堆栈顶部的对象,并作为此函数的值返回该对象
T	push(*T* item)	将数据元素压入堆栈顶部
int	search(Object *o*)	返回对象在堆栈中的位置,以 1 为基数

3.3　栈的链式存储

3.3.1　栈的链式存储定义

栈的另一种存储方式是链式存储,即将栈中的数据元素存放在一组任意的存储单元中,简称为链栈(linked stack)。链栈通常用单链表来表示,它的实现是单链表的简化。因此,链栈结点的结构与单链表结点的结构相同,如图 3-8 所示。

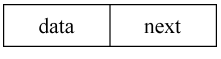


图 3-8　链栈结点

由于链栈的操作只是在一端进行,因此为了操作方便,将栈顶设在链表的头部,并且不需要头结点。栈(a_1,a_2,a_3,a_4,a_5,a_6)的链式存储结构如图 3-9 所示。

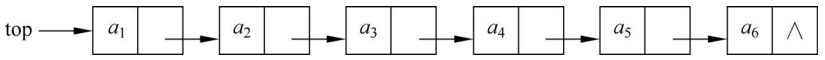


图 3-9 链栈

3.3.2 链栈源码实现

链栈用单链表来表示,其结点结构与单链表结点结构相同,因此这里仍然采用单链表中定义的结点类(Node<T>)作为链栈中的结点类。

定义链栈泛型类 LinkStack<T>,实现接口 IStack<T>。在 LinkStack<T>类中,用字段 top 表示栈顶结点地址。由于链栈的栈顶指示器不能指示栈的数据元素的个数,因此,在求链栈的长度时,必须将栈中的数据元素一个一个计数,每前进一步就将计数器增加 1,直至到达栈底。这种算法的时间复杂度比较高,为此需要在 LinkStack<T>类中增设字段 num 表示链栈中结点的个数,以牺牲空间效率换取求长度等操作时间效率的提高。

链栈泛型类 LinkStack<T>中的属性成员和构造方法如下。

```
public class LinkStack<T> implements IStack<T>{
    public Node<T> top;                    // 栈顶指示器
    public int num;                        // 栈中结点的个数
     // 构造器
    public LinkStack(){
       top = null;
       num = 0;
    }
    // 其他方法
}
```

在 Java 语言中,一个基本操作表现为类中的一个方法。LinkList<T>类除了必须实现接口 IListDS<T>中的方法外,还可添加一些另外的成员方法。链栈基本操作的实现如下。

(1) 求链栈的长度。

num 的大小表示链栈中数据元素的个数,通过返回 num 的值可以求链栈的长度。求链栈长度的算法实现如下。

```
public int GetLength() {
    return num;
}
```

(2) 清空操作。

清空操作是指清除链栈中的结点,使链栈为空。此时,栈顶指示器 top 等于 null,并且 num 等于 0。清空链栈的算法实现如下。

```
public void Clear(){
    top = null;
    num = 0;
}
```

(3) 判断链栈是否为空。

如果链栈的栈底指示器为 null 且 num 等于 0,则链栈为空,返回 true;否则,返回 false。判断链栈是否为空的算法实现如下。

```
public boolean IsEmpty(){
```

```
    if ((top == null) && (num == 0))
        return true;
    else
        return false;
}
```

(4) 入栈操作。

链栈的入栈操作是指在栈顶添加一个新结点，top 指向新的结点，num 加 1，栈发生变化。入栈操作的算法实现如下。

```
public void Push(T item) {
    Node<T> q = new Node<T>(item);
    if (top == null)
        top = q;
    else {
        q.next = top;
        top = q;
    }
    ++num;
}
```

(5) 出栈操作。

出栈操作是指在栈不为空的情况下，先取出栈顶结点的值，然后将栈顶指示器指向栈顶结点的直接后继结点，使其成为新的栈顶结点，num 减 1，栈发生变化。

出栈操作的算法实现如下。

```
public T Pop() {
    if (IsEmpty()) {
        throw new RuntimeException("Stack is empty!");
    }
    Node<T> p = top;
    top = top.next;
    --num;
    return p.data;
}
```

(6) 取栈顶元素。

如果链栈不为空，则返回栈顶结点的值；否则，抛出异常，栈不发生变化。

取栈顶元素操作的算法实现如下。

```
public T GetTop() {
    if (IsEmpty()) {
        throw new RuntimeException("Stack is empty!");
    }
    return top.data;
}
```

3.4　栈的应用举例

由于栈结构具有后进先出的固有特性，因此栈成为程序设计的有力工具。本节讨论两个栈应用的典型示例。

视频讲解

3.4.1 数制转换

十进制与其他数制的转换是计算机实现计算的基本问题,下面以十进制和八进制为例讲解如何转换。

1. 八进制转换为十进制

$(2504)_8 \Rightarrow (1348)_{10}$

该数制转换的运算过程如下。

$$(2504)_8 = 4\times8^0 + 0\times8^1 + 5\times8^2 + 2\times8^3 = 4 + 0 + 5\times64 + 2\times512 = 4 + 320 + 1024$$
$$= (1348)_{10}$$

怎样利用计算机来完成这个过程?

要求:输入一个任意八进制整数,打印输出与其等值的十进制数。

该过程实现较为简单,可以考虑利用堆栈实现。将八进制的各个数位依次入栈并出栈,出栈时乘以各个数位的权值,然后求加权和即可得到与其等值的十进制数。八进制转换为十进制的代码如下。

```
public static void OtoD() throws IOException {
    System.out.println("请输入八进制数,按 Enter 键结束:");
    char c;
    int n = 0;
    int dec = 0, i = 0;
    SeqStack < Integer > s = new SeqStack < Integer >(15);
    while ((c = (char) System.in.read()) != '\r') {
    // 从键盘读取字符,遇到 Enter 键则跳出循环
        s.Push(c - '0');                    // 数字字符转换为整型数入栈
    }
    while (!s.IsEmpty()) {
        n = s.Pop();
        dec = dec + n * (int) Math.pow(8, i);
        i = i + 1;
    }
    System.out.println("转换的十进制数是:" + dec);
}
```

2. 十进制转换为八进制

$(1348)_{10} \Rightarrow (2504)_8$

该进制转换的运算过程如下。

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

怎样利用计算机来完成这个过程?

要求:输入一个任意十进制整数,打印输出与其等值的八进制数。

计算过程:从低位到高位顺序产生八进制的各个数位 4、0、5、2。

输出:从高位到低位输出各个数位 2、5、0、4。

这个过程符合后产生先输出的原则，可以考虑利用堆栈实现，具体代码如下。

```
public static void DtoO() {
    int n;
    System.out.println("请输入要转换的十进制数:");
    Scanner reader = new Scanner(System.in);
    // 实例化 Scanner 类对象 reader
    // 调用 reader 对象的相应方法,读取键盘数据
    n = reader.nextInt();
    SeqStack < Integer > s = new SeqStack < Integer >(15);
    // 产生的各个余数位依次入栈
    while (n > 0) {          // 商等于 0 时退出循环
        s.Push(n % 8);       // 余数位入栈
        n = n / 8;           // 整型数相除,结果只保留整数
    }
    // 输出栈中元素,最后产生的余数位最先输出
    System.out.println("转换的八进制数是:");
    while (!s.IsEmpty()) {
        n = s.Pop();
        System.out.print(n);
    }
    reader.close();
}
```

3.4.2　表达式求值

表达式求值是程序设计语言编译中的一个基本问题，其实现是栈应用的又一典型示例。

在 Java 语言中，任何一个表达式都是由操作数(operand)、运算符(operator)和界限符(delimiter)组成的。运算符和界限符合称算符。例如，算术表达式(4＋2)×3－9/5 由操作数 4、2、3、9、5，运算符＋、－、×、/，界限符(、)组成。

为了叙述简洁，本书仅讨论简单算术表达式的求值问题，这种算术表达式只含加、减、乘、除 4 种运算符，操作数是一位整数，读者不难由此推广到更一般的表达式上。简单算术表达式的运算规则如下。

(1) 先乘除，后加减。

(2) 同级运算时先左后右。

(3) 先括号内，后括号外。

根据上述 3 条规则，任意两个相继出现的运算符 θ_1 和 θ_2 之间的优先关系如表 3-3 所示，其中“＃”是表达式结束符。

表 3-3　运算符间的优先关系

θ_2 \ θ_1	＋	－	×	/	(	)	＃
＋	＞	＞	＜	＜	＜	＞	＞
－	＞	＞	＜	＜	＜	＞	＞
×	＞	＞	＞	＞	＜	＞	＞
/	＞	＞	＞	＞	＜	＞	＞
(	＜	＜	＜	＜	＜	＝	无
)	＞	＞	＞	＞	无	＞	＞
＃	＜	＜	＜	＜	＜	无	＝

θ_1 和 θ_2 之间的优先关系通常为如下 3 种关系。

(1) $\theta_1<\theta_2$：θ_1 的优先权低于 θ_2。

(2) $\theta_1=\theta_2$：θ_1 的优先权等于 θ_2。

(3) $\theta_1>\theta_2$：θ_1 的优先权高于 θ_2。

在表 3-3 中，θ_1 和 θ_2 是相继出现的，先出现 θ_1，后出现 θ_2。θ_1 和 θ_2 可以是同一运算符。如果 θ_1 和 θ_2 都是两个加号，则第一个加号的优先权高。先做第一个加法运算，再做第二个加法运算。

在表 3-3 中，“#”是表达式结束符，为了算法简洁，在表达式的最左边也虚设一个“#”构成表达式的一对括号。表中的“(”=“)”表示当左、右括号相遇时，括号内的运算已完成，同理“#”=“#”表示整个表达式求值完毕。表中的 3 处“无”表示在表达式中不允许它们相继出现，一旦遇到这种情况，表示出现了语法错误。在下面的讨论中，假设输入的表达式不会出现语法错误。

在处理表达式前，需要先设置两个栈。

(1) 操作数栈(OPRD)存放处理表达式过程中的操作数。

(2) 运算符栈(OPTR)存放处理表达式过程中的运算符。开始时，在运算符栈中的栈底压入一个表达式的结束符“#”。

在处理表达式时，从左到右依次读出表达式中的各个符号(操作数或运算符)，每读出一个符号后，根据运算规则进行如下处理。

(1) 假如是操作数，则将其压入 OPRD，并依次读下一个符号。

(2) 假如是运算符，则令 θ_1=OPTR 栈顶运算符，θ_2=读出的运算符。根据 θ_1 和 θ_2 关系的不同，分以下 3 种情况分别处理。

① $\theta_1<\theta_2$：θ_2 入 OPTR。

② $\theta_1=\theta_2$：θ_1 出 OPTR，处理 θ_2 之后的下一运算符。

③ $\theta_1>\theta_2$：θ_1 出 OPTR，操作数出 OPRD，计算 $a\theta_1 b$，计算结果入操作数栈，重新处理 θ_2。其中，b 为先出 OPRD 的操作数，a 为后出 OPRD 的操作数。

(3) OPRD 中剩下的数就是计算结果。

【例 3-1】 写出求解表达式#4+3×5#时的堆栈变化情况。

分析：表达式的求解步骤如下。

(1) # 入 OPTR，剩余表达式部分为 4+3×5#。

(2) 4 入 OPRD，剩余表达式部分为+3×5#。

(3) 因#<+，+入 OPTR，剩余表达式部分为 3×5#。

(4) 3 入 OPRD，剩余表达式部分为×5#。

(5) 因+<×，×入 OPTR，剩余表达式部分为 5#。

(6) 5 入 OPRD，剩余表达式部分为#。

(7) 因×>#，×出 OPTR，5 和 3 出 OPRD，计算 3×5=15，15 入 OPRD，继续处理#。

(8) 因+>#，+出 OPTR，15 和 4 出 OPRD，计算 4+15=19，19 入 OPRD，继续处理#。

(9) 因#=#，#出 OPTR。此时，表达式求值完毕。返回 Getop(OPRD)，即 19。

堆栈变化情况如表 3-4 所示。

表 3-4　求解表达式＃4＋3×5＃时的堆栈变化情况

步　骤	OPTR	OPRD	剩余表达式部分
(1)	＃		4＋3×5＃
(2)	＃	4	＋3×5＃
(3)	＃,＋	4,3,5	3×5＃
(4)	＃,＋	4,3	×5＃
(5)	＃,＋,×	4,3	5＃
(6)	＃,＋,×	4,3,5	＃
(7)	＃,＋	4,15	＃
(8)	＃	19	＃
(9)		19	

算术表达式求值的算法实现如下。

```
public static int EvaluateExpression() {
    SeqStack < Character > optr = new SeqStack < Character >(20);
    SeqStack < Integer > opnd = new SeqStack < Integer >(20);
    optr.Push('#');
    char c = (char) System.in.read();
    char theta = ' ';
    int a = 0;
    int b = 0;
    while (c != '#') {
        if ((c != '+') && (c != '-') && (c != '*') && (c != '/') && (c != '(') && (c != ')')) {
            optr.Push(c);
        } else {
            switch (Precede(optr.Getop(), c)) {
            case '<':
                optr.Push(c);
                c = (char) System.in.read();
                break;
            case '=':
                optr.Pop();
                c = (char) System.in.read();
                break;
            case '>':
                theta = optr.Pop();
                a = opnd.Pop();
                b = opnd.Pop();
                opnd.Push(Operate(a, theta, b));
                break;
            }
        }
    }
    return opnd.Getop();
}
```

上述算法调用了两个方法。其中，Precede 是判定 optr 栈顶运算符与读入运算符之间的优先级关系的方法，Operate 是进行二元运算的方法。这两个方法读者可作为练习自行实现。

视频讲解

3.5 队列的基本概念

3.5.1 队列的相关定义

1. 队列

队列(queue)是一种只允许在表尾端进行插入,在表头端进行删除的线性表。

插入数据元素时,新插入的数据元素 e 只能处于线性表的表尾,如图 3-10 所示。

删除数据元素时,只能删除线性表的表头元素,如图 3-11 所示。

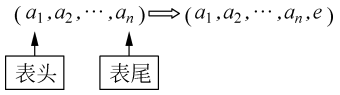


图 3-10 插入数据元素

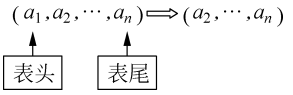


图 3-11 删除数据元素

2. 队头和队尾

队头(front)即表头,队尾(end)即表尾。

3. 入队列和出队列

入队列是队列的插入操作,出队列是队列的删除操作。

4. FIFO

队头元素 a_1 是最先进队列的,也是最先出队列的;队尾元素 a_n 是最后进队列的,因而也是最后出队列的。因此,队列也被称为“先进先出”(First In First Out,FIFO)线性表。

3.5.2 队列的抽象数据类型

ADT Queue{
数据对象: $D=\{a_i \mid a_i \in \text{ElemSet}, i=1,2,\cdots,n, n\geqslant 0\}$
数据关系: $R=\{<a_{i-1},a_i> \mid a_{i-1},a_i \in D, i=2,\cdots,n\}$
基本操作 P:
(1) int Count(): 求队列的长度。
操作结果: 返回队列中数据元素的个数。
(2) boolean IsEmpty(): 判断队列是否为空。
操作结果: 如果队列为空,则返回 TRUE; 否则,返回 FALSE。
(3) Clear(): 清空操作。
操作结果: 使队列为空。
(4) In(T item): 入队列操作。
操作结果: 将值为 item 的新数据元素添加到队尾,队列发生变化。
(5) T Out(): 出队列操作。
初始条件: 队列不为空。
操作结果: 将队头元素从队列中取出,队列发生变化。
(6) T GetFront(): 取队头元素。
初始条件: 队列不为空。
操作结果: 返回队头元素的值,队列不发生变化。
}

关于队列抽象数据类型中 D 的说明:D 是 n 个数据元素的集合,D 是 ElemSet 的子集。ElemSet 表示某个集合,集合中的数据元素类型相同,如整数集、自然数集等。

关于队列抽象数据类型中 R 的说明：R 是数据元素之间关系的集合，$<a_{i-1},a_i>$表示 a_{i-1} 和 a_i 之间互为前驱和后继的逻辑关系。约定 a_n 端为队尾，a_1 端为队头。

队列的数据对象和数据关系与线性表是相同的。

关于队列抽象数据类型中基本操作的说明：

(1) 队列中的数据元素类型用 T 表示，T 可以是原子类型，也可以是结构类型。整型用 int 表示，逻辑型用 boolean 表示。

(2) 队列的基本操作是定义于逻辑结构上的基本操作，是向使用者提供的使用说明。基本操作只有在存储结构确定之后才能实现。如果队列采用的存储结构不同，则队列的基本操作实现算法也不相同。

(3) 基本操作的种类和数量可以根据实际需要决定。但是，队列是操作受限的线性表，不能任意地定义基本操作。例如，不能在队列的第 i 个位置插入数据元素。

(4) 基本操作名称，形式参数数量、名称、类型，返回值类型等由设计者决定。使用者根据设计者的设计规则使用基本操作。

在 Java 程序设计语言中，用接口表示队列的抽象数据类型如下。

```
interface IQueue<T> {
    int Count();          // 求队列的长度
    boolean IsEmpty();    // 判断队列是否为空
    void Clear();         // 清空操作
    void In(T item);      // 入队列操作
    T Out();              // 出队列操作
    T GetFront();         // 取队头元素
}
```

与线性表相同，队列也有两种存储方式：顺序存储和链式存储。

3.6　队列的链式存储

3.6.1　队列的链式存储定义

队列的链式存储是指将队列的数据元素存放在一组任意的存储单元中(编号可以不连续)。队列的链式存储结构简称链队列。

由于数据的存储没有规律，无法根据当前元素的地址算出它的后继元素的地址，因此无法确定它的后继元素。必须在数据元素后面附设一个引用，作为后继元素的地址。

队列 $q=(a_1,a_2,\cdots,a_n)$的链式存储结构如图 3-12 所示。

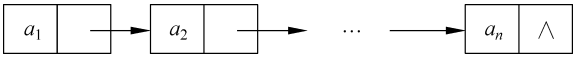


图 3-12　链队列

与线性表相同，为了操作方便，一般在第一个结点之前增加一个头结点。带头结点的链队列如图 3-13 所示。

头结点的类型与其他结点一样，分为数据部分和引用部分。头结点的数据部分为空，引

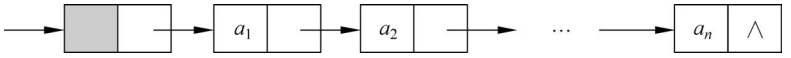


图 3-13 带头结点的链队列

用部分指向链队列的第一个数据元素结点。

只要知道头结点的地址,就能“顺藤摸瓜”确定队列的每一个数据元素。头结点的地址用 front 来表示,称 front 为队头引用。

因为队列的插入操作都是在队尾进行的,为了提高插入操作的效率,附设一个引用 rear 指示队尾元素的地址,称 rear 为队尾引用。以数据的冗余换取入队操作时间效率的提高。带尾引用的链队列如图 3-14 所示。

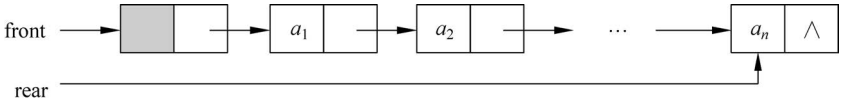


图 3-14 带尾引用的链队列

本教材链队列基本操作的实现指的是带尾引用的链队列基础上实现的。

视频讲解

3.6.2 链队列基本操作分析

1. 出队列

出队列即删除队列的第一个数据元素,令头结点的引用指向第二个结点,如图 3-15 所示。

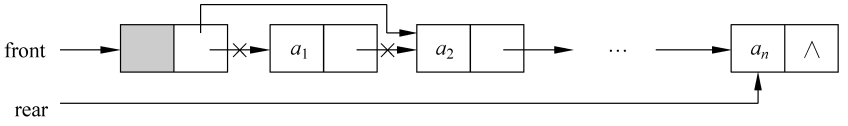


图 3-15 出队列操作

2. 入队列

入队列即插入 e 为队列新的队尾元素,其操作步骤如下。

(1) 建立新结点 s,数据部分是 e,引用部分为 NULL。

(2) a_n 引用指向结点 s。

(3) 队尾引用 rear 指向结点 s。

入队列操作如图 3-16 所示。

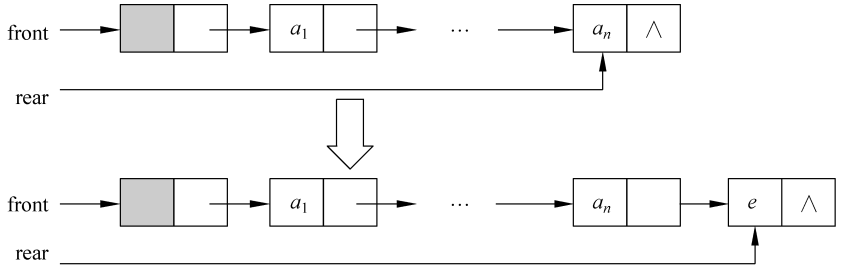


图 3-16 入队列操作

3.6.3　链队列源码实现

链队列用单链表来表示，其结点结构与单链表结点结构相同，因此这里仍然采用单链表中定义的结点类(Node < T >)作为链队列中的结点类。

定义链队列泛型类 LinkQueue < T >，实现接口 IQueue < T >。

本书用队头引用和队尾引用的组合来表示队列，即 LinkQueue < T >类中有两个引用 front 和 rear。

链队列泛型类 LinkQueue < T >中的属性成员和构造方法如下。

```
public class LinkQueue < T > implements IQueue < T > {
    // 链队列类 LinkQueue < T >
    public Node < T > front;        // 队头指示器
    public Node < T > rear;         // 队尾指示器
    // 无参构造器 ,构造了一个只有头结点的空队列
    public LinkQueue() {
        front = rear = new Node < T >();
        // 头结点数据域等于 T 类型默认值,引用域为空
    }
    //其他方法
}
```

在 Java 语言中，一个基本操作表现为类中的一个方法。LinkList < T >类除了必须实现接口 IQueue < T >中的方法外，还可添加一些另外的成员方法。链队列基本操作的实现如下。

(1) 求链队列的长度。

从队头引用开始，一个结点一个结点地计数，直到队尾。求链队列长度的代码如下。

```
public int Count() {
    Node < T > p = front;      // 新建结点 p 等于头引用
    int len = 0;               // len 值初始化为 0
    while (p != null) {
        ++len;
        p = p.next;
    }                          // 此循环执行完毕后, len 值为链表结点的个数(包含头结点)
    return len - 1;            // 链表结点的个数减 1 即为队列的元素个数
}
```

(2) 清空操作。

清空操作是指清除队列中的结点，使队列为空。当队列为空时，头引用 front.next 等于 null，rear 等于 front。清空操作如图 3-17 所示，具体代码如下。

```
public void Clear() {
    front.next = null;         // 令头引用 front.next 等于 null
    rear = front;
}
```

(3) 判断链队列是否为空。

队列非空和空的状态如图 3-18 所示。如果链队列的队头指示器等于队尾指示器，则表示链队列为空，返回 true；否则，返回 false。判断链队列是否为空的代码如下。

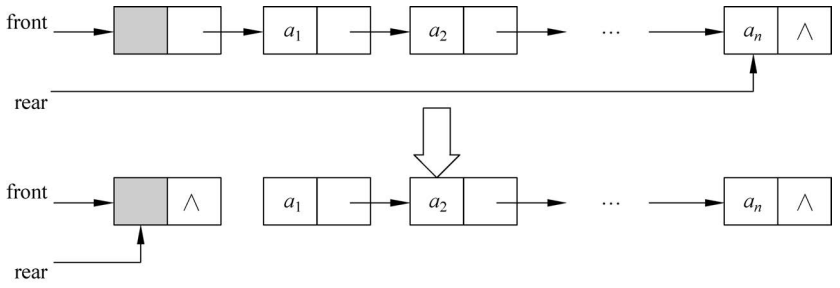
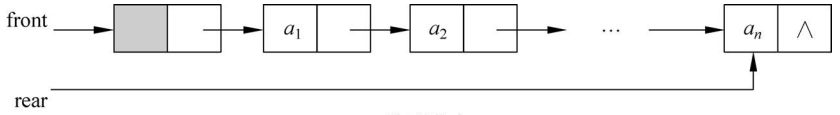
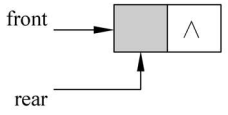


图 3-17 清空操作

```
public boolean IsEmpty() {
    if (front == rear)
        return true;
    else
        return false;
}
```



(a) 队列非空



(b) 队列为空

图 3-18 队列非空和空状态示意图

当队列为空时,队头指示器 front 的引用域为空,代码也可以写成如下形式。

```
public boolean IsEmpty() {
    if (front.next == null)
        return true;
    else
        return false;
}
```

(4) 入队操作。

链队列的入队操作是指在队尾添加一个新结点,队尾指示器 rear 指向新的结点。入队操作的算法实现如下。

```
public void In(T item) {
    Node<T> q = new Node<T>(item);    // 建立新结点 q
    rear.next = q;                    // 最后一个结点的引用域指向新结点
    rear = q;                         // rear 指向新的结点
}
```

(5) 出队操作。

出队操作是指在链队列不为空的情况下,将第 1 个元素出队列。链队列头结点的引用域指向第二个结点,队列发生改变。

如果队列中只有一个元素,则出队列后队列为空,此时需要修改尾引用指向头结点。

出队操作的算法实现如下。

```
public T Out() {
    if (IsEmpty()) {
        throw new RuntimeException("队列为空,无法删除");
    }
    Node<T> p = front.next;              // p 指向第一个结点
    front.next = p.next;                 // 头引用的 next 域指向第二个结点
    if(Count() == 0) rear = front;       //如果元素出队列后,队列变为空状态,则修改尾引用
    return p.data;                       // 返回第一个结点的数据
}
```

(6) 获取链队列头结点的值。

如果链队列不为空,则返回链队列第一个结点的值；否则,返回提示信息,表示队列为空,队列不发生改变。获取链队列头结点的值的算法实现如下。

```
public T GetFront() {
    if (IsEmpty()) {
        throw new RuntimeException("队列为空,无法获取");
    }
    Node<T> p = front.next;              // p 指向第一个结点
    return p.data;                       // 返回第一个结点的数据
}
```

3.7　队列的顺序存储

3.7.1　队列的顺序存储定义

队列的顺序存储是指将队列的数据元素存放在一组地址连续的存储单元中,在 Java 语言中就是将数据元素存放在一维数组中。由于队列的删除位置只能是队列的第一个位置,因此为了避免删除数据元素时大量地移动数据元素,附设一个 front 指示队头元素在数组中的下标,同时附设一个整型的 rear 指示队尾元素在数组中的位置,其值等于队尾元素在数组中的下标加 1。队列的顺序存储结构简称为顺序队列。

队列($J1$,$J2$,$J3$)的顺序存储结构如图 3-19 所示,其中 front＝0,rear＝3。

空队列如图 3-20 所示,其中 rear＝front＝0。

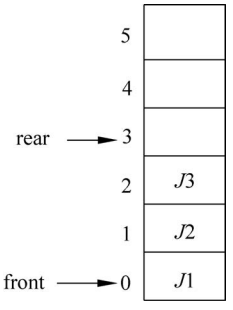


图 3-19　队列($J1$,$J2$,$J3$)的顺序存储结构

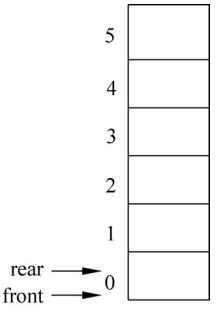


图 3-20　空队列

视频讲解

3.7.2 顺序队列基本操作分析

1. 入队操作

顺序队列入队操作是指在 rear 处插入元素,然后将 rear 值加 1,如图 3-21 所示。

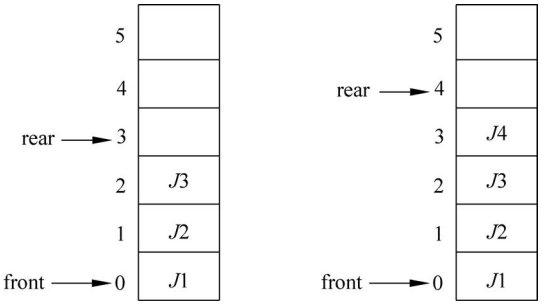


图 3-21 入队列

2. 出队操作

顺序队列出队操作是指删除 front 处的队头元素,即队头指示器 front 加 1,如图 3-22 所示。经过若干插入、删除操作之后,某一时刻队列的状态如图 3-23 所示。

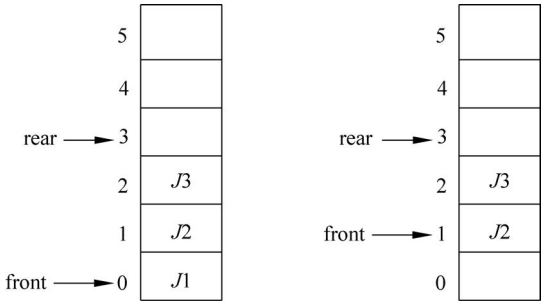


图 3-22 出队列

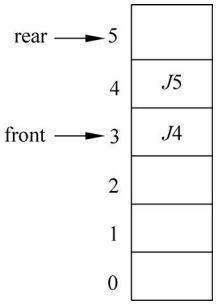


图 3-23 某一时刻队列的状态

此时,队尾 rear 已经指向了数组的最后一个位置,若再有元素入列,则会导致“溢出”,即元素下标超出队尾 rear 的最大值。在图 3-23 中,虽然队尾 rear 已经指向了最后一个位置,但事实上数组中还有空位置。也就是说,数组的存储空间并没有满,但队列却发生了溢出,将这种现象称为假溢出。如何处理假溢出?正确的做法是:将数组的空间想象成一个环状的空间,将第 0 个位置看成是最后一个位置的下一位置。在第 5 个位置插入数据元素 $J6$ 后,rear 指向第 0 个位置,如图 3-24 所示。

如果 $J7$ 需要入队列,则在第 0 个位置插入数据元素 $J7$ 后,rear 后移并指向第 1 个位置,如图 3-25 所示。此时,队列的逻辑结构是($J4$,$J5$,$J6$,$J7$)。

顺序队列中的 rear 和 front 达到最大值后都可以从 0 重新开始循环,所以顺序队列又称为循环顺序队列,简称为循环队列。

循环队列存在一个问题:队满和队空的标志均为 front=rear,如图 3-26 所示,如何区分队满和队空?

解决方案:少用一个空间,当队列中的元素个数达到数组的长度减 1 时,就标志队列已满,数据元素不能再入队列了,如图 3-27 所示。

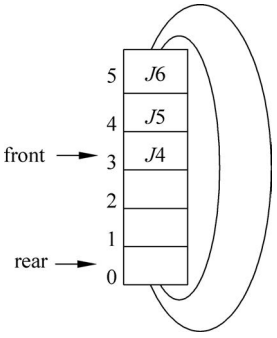


图 3-24　rear 指向第 0 个位置

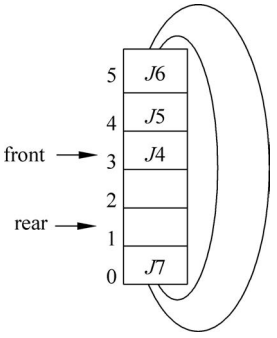


图 3-25　rear 指向第 1 个位置

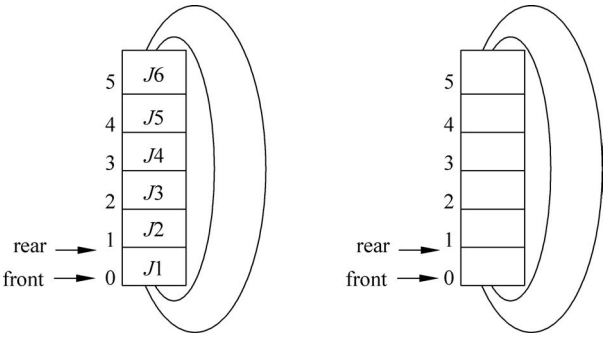


图 3-26　队满和队空(无法区分)

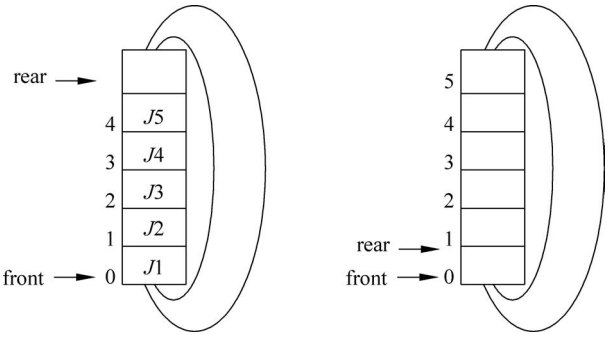


图 3-27　队满和队空(可区分)

3.7.3　循环顺序队列源码实现

1. 循环顺序队列类的实现

(1) 定义循环顺序队列类 CSeqQueue < T >,实现接口 IQueue < T >。

```
public class CSeqQueue < T > implements IQueue < T >
```

(2) 创建循环顺序队列类 CSeqQueue < T > 中的属性成员。

```
public class CSeqQueue < T > implements IQueue < T > {
   public T[ ] data;
     //数组,用于存储队列中的数据元素
    public int maxsize;
    public int front;      //队头,front 的变化范围是 0～maxsize－1
```

```
    public int rear;        //队尾,rear 的变化范围也是 0～maxsize－1
}
```

关于属性成员的几点说明:

① Java 语言中的数组在内存中占用的存储空间就是一组地址连续的存储单元,因此,在 Java 虚拟处理器中考虑问题时,认为队列的顺序存储就是将队列的数据元素存放到数组中。

② maxsize 表示数组的容量。

③ 队头 front 的变化范围是 0～maxsize－1,队尾 rear 的变化范围也是 0～maxsize－1。

(3) 创建循环顺序队列类 CSeqQueue＜T＞的构造方法。

```
public CSeqQueue(int size) {
    data = (T[]) new Object[size];
    maxsize = size;
    front = rear = 0;
}
```

2. 基本操作的实现

(1) 求顺序队列的长度。

① rear＞front,如图 3-28(a)所示。此时,循环顺序队列的长度为 rear－front。

② 当 rear 到达数组的上限后又从数组的底端开始,rear＜front,如图 3-28(b)所示。此时,循环顺序队列的长度为 rear－front＋maxsize。

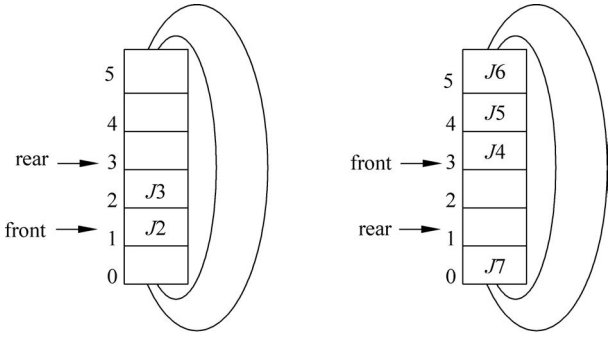


图 3-28　求队列长度

综合以上两种情况,求循环顺序队列长度的代码如下。

```
public int Count() {
    return (rear - front + maxsize) % maxsize;
}
```

(2) 清空操作。

清空操作是指使循环顺序队列为空,rear 和 front 相等即表示循环队列为空,代码如下。

```
public void Clear() {
    front = rear;
}
```

(3) 判断循环顺序队列是否为空。

如果循环顺序队列的 rear 和 front 相等,则循环顺序队列为空,返回 true; 否则,返回 false。

```
public boolean IsEmpty() {
    if (front == rear)
        return true;
    else
        return false;
}
```

(4) 判断循环顺序队列是否为满。

① 当 rear＞front 时(见图 3-29(a)),如果循环顺序队列为满,则有(rear＋1)％ maxsize＝＝front 成立。

② 当 rear＜front 时(见图 3-29(b)),如果循环顺序队列为满,则有 rear＋1＝＝front 成立,且(rear＋1)％ maxsize＝＝front 也成立。

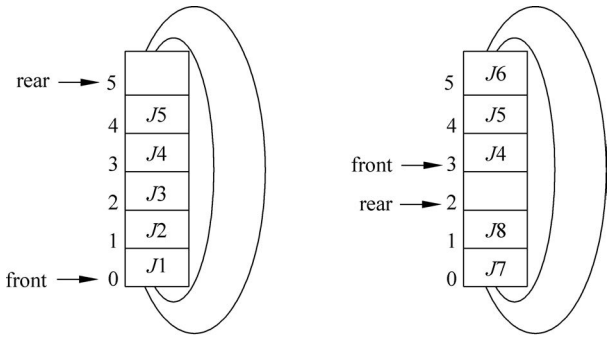


图 3-29　判断队列状态是否为满

综合以上两种情况,队列满的标志为(rear＋1)％ maxsize＝＝front。

当循环顺序队列为满,即(rear＋1)％ maxsize＝＝front 时,则返回 true; 否则,返回 false。

判断循环顺序队列是否为满的代码如下。

```
public boolean IsFull(){
   return (rear + 1) % maxsize == front;
}
```

(5) 入队操作。

入队操作通常是指在循环顺序队列未满的情况下,在 rear 处插入元素,然后使循环顺序队列的 rear 加 1。特别地,如果 rear 已是数组的最后一个位置,则插入元素后,rear 由 maxsize－1 变为 0。

入队操作的代码如下。

```
public void In(T item) {
    if (IsFull()) {
        System.out.println("Queue is full");
        return;
    }
    data[rear] = item;
    if (rear == maxsize - 1)
        rear = 0;
    else
        rear = rear + 1;
}
```

综合以上两种情况,rear 的变化表达式为

$$\text{rear}=(\text{rear}+1)\%\text{maxsize}=\begin{cases}0, & \text{当 rear}=\text{maxsize}-1\text{ 时}\\ \text{rear}+1, & \text{当 rear}<\text{maxsize}-1\text{ 时}\end{cases}$$

因此,入队操作是指在循环顺序队列未满的情况下,在 rear 处插入元素,使得 rear=(rear+1)%maxsize。代码也可写成如下形式。

```
public void In1(T item) {
    if (IsFull()) {
        System.out.println("Queue is full");
        return;
    }
    data[rear] = item;
    rear = (rear + 1) % maxsize;
}
```

(6) 出队操作。

循环顺序队列的出队操作通常是指在队列不为空的情况下,使队头指示器 front 加 1。特别地,如果 front 已是数组的最后一个位置,则元素出队后,front 由 maxsize-1 变为 0。

出队操作的代码如下。

```
public T Out() {
    T tmp;
    // 判断队列是否为空
    if (IsEmpty()) {
        throw new RuntimeException("队列为空");
    }
    tmp = data[front];
    if (front == maxsize - 1)
        front = 0;
    else
        front = front + 1;
    return tmp;
}
```

综合以上两种情况,front 的变化表达式为

$$\text{front}=(\text{front}+1)\%\text{maxsize}=\begin{cases}0, & \text{当 front}=\text{maxsize}-1\text{ 时}\\ \text{front}+1, & \text{当 front}<\text{maxsize}-1\text{ 时}\end{cases}$$

因此,循环顺序队列的出队操作是指在队列不为空的情况下,使得 front=(front+1)%maxsize。代码也可写成如下形式。

```
public T Out1() {
    T tmp;
    // 判断队列是否为空
    if (IsEmpty()) {
        throw new RuntimeException("队列为空");
    }
    tmp = data[front];
    front = (front + 1) % maxsize;
    return tmp;
}
```

(7) 获取队头元素。

如果循环顺序队列不为空,则返回队头元素的值。

```
public T GetFront() {
    if (IsEmpty()) {
        throw new RuntimeException("队列为空");
    }
    return data[front];
}
```

3.8　Java 基础类库中的队列

Java 语言的推出者设计开发了 java.util.Queue，供应用程序员使用。在 Java 基础类库中，实现该接口的类有 AbstractQueue、ArrayBlockingQueue、ArrayDeque、ConcurrentLinkedQueue、DelayQueue、LinkedBlockingDeque、LinkedBlockingQueue、LinkedList、PriorityBlockingQueue、PriorityQueue、SynchronousQueue。在开发软件时，也可以开发新类实现该接口，以便切合软件的实际功能需求。java.util.Queue 中的部分接口方法如表 3-5 所示。

表 3-5　接口方法及功能说明

返回值类型	接 口 方 法	功 能 说 明
boolean	add(*T e*)	将指定的元素插入此队列(如果立即可行且不会违反容量限制)。如果成功，则返回 true；如果当前没有可用的空间，则抛出 IllegalStateException
T	element()	获取但不移除此队列的头
boolean	offer(*T e*)	将指定的元素插入此队列(如果立即可行且不会违反容量限制)。当使用有容量限制的队列时，此方法通常要优于 add(*T*)，后者可能无法插入元素而只是抛出一个异常
T	peek()	获取但不移除此队列的头。如果此队列为空，则返回 null
T	poll()	获取并移除此队列的头。如果此队列为空，则返回 null
T	remove()	获取并移除此队列的头

3.9　队列的应用举例

视频讲解

队列的典型应用是通过编程判断一个字符串是否为回文。回文是指一个字符序列以中间字符为基准，其两边字符完全相同，如字符序列“ACBDEDBCA”是回文。

算法思想：判断一个字符序列是否为回文，即将第一个字符与最后一个字符相比较，第二个字符与倒数第二个字符比较，以此类推，直至第 i 个字符与第 $n-i$ 个字符比较。如果每次比较都相等，则该序列为回文；如果某次比较不相等，则不是回文。因此，可以将字符序列分别入队列和栈，然后逐个出队列和出栈，并比较出队列的字符和出栈的字符是否相等。如果字符比较全部相等，则该字符序列是回文；否则，该序列不是回文。

算法中的队列和栈可以采用任意存储结构，本例采用循环顺序队列和顺序栈来实现，其他的情况读者可作为习题。算法中假设输入的都是英文字符而没有其他字符，对于输入其他字符情况的处理，读者可以自行实践。使用循环顺序队列和顺序栈来判断一个字符串是否为回文的代码如下。

```
public static void main(String[] args) throws IOException {
    // TODO Auto-generated method stub
    SeqStack<Character> s = new SeqStack<Character>(50);     // 建立堆栈 s
    CSeqQueue<Character> q = new CSeqQueue<Character>(50);   // 建立队列 q
    System.out.println("请输入字符串,按 Enter 键结束");
    char c;
    while ((c = (char) System.in.read()) != '\r') {
        // 从键盘读入字符,遇到 Enter 键则跳出循环
        s.Push(c);                                           // 字符入栈
        q.In(c);                                             // 字符入队列
    }
    while (!s.IsEmpty() && !q.IsEmpty()) {
        // 队列和栈均不空,进行以下比较
        if (s.Pop() != q.Out()) {
            // 出栈和出队列元素不相等,退出循环
            break;
        }
    }
    if (!s.IsEmpty() && !q.IsEmpty()) {
        // 栈或队列非空,是从 break 语句退出循环
        System.out.println("这不是回文!");
    } else {
        // 栈和队列均为空,不是从 break 语句退出循环
        System.out.println("这是回文!");
    }
}
```

本章小结

本章内容分成两大部分：栈和队列。栈和队列均是操作受限的线性表。第一部分首先介绍了栈的基本概念和抽象数据类型栈；然后介绍了栈的两种存储结构：顺序栈和链栈，基于顺序栈和链栈实现了抽象数据类型栈中的基本操作，并介绍了 Java 基础类库中的顺序栈 java.util.Stack 的部分代表性方法的功能，供读者分析比较；最后讲了栈的应用，利用栈实现了数制转换和表达式求值。第二部分首先介绍了队列的基本概念和抽象数据类型队列；然后介绍了队列的两种存储结构：带尾引用的单向链队列和循环顺序队列，基于链队列和循环顺序队列实现了抽象数据类中队列中的基本操作，并介绍了 Java 基础类库中 java.util.Queue 接口的部分方法及实现该接口的类，供读者分析比较；最后讲了队列的应用，综合利用队列和栈判断一个字符串是否为回文。

在线测试

习题 3

一、选择题

1. 若一个栈的输入序列是 1,2,3,…,n,且输出序列的第一个元素是 n,则第 k 个输出元素是(　　)。

A. k　　B. $n-k-1$　　C. $n-k+1$　　D. 不确定

2. 栈与队列都是(　　)。

A. 链式存储的线性结构　　B. 链式存储的非线性结构

C. 限制存取点的线性结构　　D. 限制存取点的非线性结构

3. 在解决计算机主机与打印机之间速度不匹配问题时,通常设置一个打印数据缓冲区,主机将要输出的数据依次写入该缓冲区,而打印机则从该缓冲区中取走数据打印。该缓冲区应该是一个(　　)结构。

A. 堆栈　　B. 队列　　C. 数组　　D. 线性表

4. 循环队列的队头和队尾引用分别为 front 和 rear,判断一个循环队列 Q(最多 n 个元素)为满的条件是(　　)。

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

C. Q.front==(Q.rear+1)%n　　D. Q.front==(Q.rear−1)%n

5. 假设用一个大小为 6 的数组来实现循环队列,且当前 rear 和 front 的值分别为 0 和 3。当从队列中删除一个元素,再加入两个元素后,rear 和 front 的值分别为(　　)。

A. 1 和 5　　B. 2 和 4　　C. 4 和 2　　D. 5 和 1

6. 队列的插入操作是在(　　)。

A. 队尾　　B. 队头

C. 队列任意位置　　D. 队头元素后

7. 循环队列的队头和队尾引用分别为 front 和 rear,则判断循环队列为空的条件是(　　)。

A. front==rear　　B. front==0

C. rear==0　　D. front=rear+1

8. 假设有一个顺序栈 S,其栈顶引用为 top,则将元素 e 入栈的操作是(　　)。

A. S.top=e;S.top++　　B. S.top++;S.top=e

C. S.top=e　　D. S.top=e

9. 栈的插入和删除操作在(　　)。

A. 栈底　　B. 栈顶　　C. 任意位置　　D. 指定位置

10. 判定一个顺序栈 S(栈空间大小为 n)为空的条件是(　　)。

A. S.top==0　　B. S.top!=0

C. S.top==n　　D. S.top!=n

11. 在一个链队列中,front 和 rear 分别为头引用和尾引用,则插入一个结点 s 的操作为(　　)。

A. front=front.next　　B. s.next=rear;rear=s

C. rear.next=s;rear=s　　D. s.next=front;front=s

12. 若一个队列的入队序列是 1,2,3,4,则队列的出队序列是(　　)。

A. 1,2,3,4　　B. 4,3,2,1

C. 1,4,3,2　　D. 3,4,1,2

13. 当用大小为 N 的数组存储顺序循环队列时,该队列的最大长度为(　　)。

A. N　　B. N+1　　C. N−1　　D. N−2

14. 队列的删除操作是在(　　)。

A. 队首　　B. 队尾　　C. 队前　　D. 队后

15. 循环队列用数组 $A[0..m-1]$存放其元素值,已知其头尾引用分别是 front 和 rear,则当前队列中的元素个数是(　　)。

A. (rear−front+m)%m　　B. rear−front+1

C. rear−front−1　　D. rear−front

16. 在一个链队列中,假设 front 和 rear 分别为队头引用和队尾引用,则删除一个结点的操作是(　　)。

A. front=front.next　　B. rear=rear.next

C. rear.next=front　　D. front.next=rear

17. 队列和栈的主要区别是(　　)。

A. 逻辑结构不同　　B. 存储结构不同

C. 所包含的运算个数不同　　D. 限定插入和删除的位置不同

二、填空题

1. 栈是限定在_____一端进行插入或删除操作的线性表。在栈中,允许插入和删除操作的一端称为_____,而另一端称为_____。不含元素的栈称为_____。

2. 在栈的运算中,栈的插入操作称为_____,栈的删除操作称为_____。

3. 根据栈的定义,每一次进栈的元素都在原栈顶元素之上,并成为新的_____;每一次出栈的元素总是当前的_____,最后进栈的元素总是最先出栈。因此,栈也称为后进先出线性表,简称为_____表。

4. 栈是一种操作受到限制的线性表,是一种特殊的线性表。因此,栈也有_____和_____两种存储结构,分别称为_____和_____。

5. 队列也是一种特殊的线性表。在队列中,允许插入的一端称为_____,允许删除的一端称为_____。

6. 已知栈的输入序列为 1,2,3,…,n,输出序列为 $a_1,a_2,\cdots,a_n$,符合 $a_2=\ =n$ 的输出序列的个数为_____。

7. 设栈 S 和队列 Q 的初始状态为空,元素 $e1,e2,e3,e4,e5,e6$ 依次通过栈 S,一个元素出栈后即进入队列 Q。若 6 个元素出队的序列是 $e2,e4,e3,e6,e5,e1$,则栈的容量至少应该是_____。

8. 设循环队列的容量为 70,现经过一系列的入队和出队操作后,front 为 20,rear 为 11,则队列中元素的个数为_____。

三、判断题

1. 栈和队列都是限制存取点的线性结构。(　　)

2. 不同的入栈和出栈组合可能得到相同的输出序列。(　　)

3. 循环队列是顺序存储结构。(　　)

4. 当循环队列满时,rear==front。(　　)

5. 在对链队列(带头结点)进行出队操作时,不会改变 front 引用的值。(　　)

四、综合题

1. 设有 4 个元素 A、B、C 和 D 进栈,试给出它们所有可能的出栈顺序。

2. 假设以带头结点的循环链表表示队列,只设一个引用指向队尾结点,且不设头引用,请写出图 3-30 相应的入队列算法。

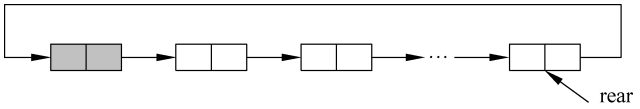


图 3-30　习题 4.2 图

五、实验题

1. 编写程序，从键盘输入一个十进制数，输出与其等值的八进制数。
2. 编写程序，利用栈和队列实现判断一个字符串是否为回文。