第3章 栈与队列

在各类数据结构中，栈（Stack）和队列（Queue）是除线性表以外，另外两种应用非常广泛且极为重要的线性结构。例如，递归函数调用之间的链接和信息交换、编译器对程序的语法分析过程，操作系统实现对各种进程的管理等，都要涉及栈或队列的应用。它们与线性表之间的不同之处在于：栈和队列可被看成是两种操作受限的特殊线性表，其特殊性体现在它们的插入和删除操作都是控制在线性表的一端或两端进行。

本章主要知识点：

- 栈的概念及其抽象数据类型描述；
- 顺序栈类和链栈类的描述与实现；
- 栈的应用；
- 队列的概念及其抽象数据类型描述；
- 顺序循环队列类和链队列类的描述与实现；
- 队列的应用。

3.1 栈

3.1.1 栈的概念

栈是一种特殊的线性表，栈中的数据元素以及数据元素间的逻辑关系和线性表相同，两者之间的差别在于：线性表的插入和删除操作可以在表的任意位置进行，而栈的插入和删除操作只允许在表的尾端进行。其中，栈中允许进行插入和删除操作的一端称为栈顶（top），另一端称为栈底（bottom）。假设栈中的数据元素序列为$\{a_0,a_1,a_2,\cdots,a_{n-1}\}$，则$a_0$称为栈底元素，$a_{n-1}$称为栈顶元素，$n$为栈中数据元素的个数（当$n=0$时，栈为空）。通常，人们将栈的插入操作称为入栈（push），而将删除操作称为出栈（pop），如图3.1所示。

从栈的概念可知，每次最先入栈的数据元素总是被放在栈的底部，成为栈底元素；而每次最先出栈的总是那个放在栈顶位置的数据元素，即栈顶元素。因此，栈是一种后进先出（Last In First Out，LIFO），或先进后出（First In Last Out，FILO）的线性表。

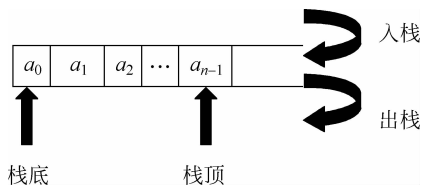


图3.1 栈及其操作的示意图

在现实生活中有许多具有栈的特性（运算受限）

的应用实例。例如,一叠盘子可被看作是一个栈,因为取出盘子和添加盘子的操作满足“后进先出”或“先进后出”的原则,还有火车调度也可被视为是一个栈的模型。

尽管栈的特性降低了栈的插入与删除操作的灵活性,但这种特性使栈的操作更为有效、更易实现。栈在计算机应用中也到处可见,例如,浏览器对用户当前访问过地址的管理、键盘缓冲区中对键盘输入信息的管理等都采用了栈式结构。

思考:假设有编号为 a、b、c 和 d 的 4 辆列车,顺序进入一个栈式结构的站台,这 4 辆列车开出车站的所有可能的顺序有多少种?

3.1.2 栈的抽象数据类型描述

栈也是由 $n(n\geqslant 0)$个数据元素所构成的有限序列,其数据元素的类型可以任意,但只要是同一种类型即可。根据栈的特性,定义在栈的抽象数据类型中的基本操作如下。

(1) 置栈空操作 clear():将一个已经存在的栈置成空栈。

(2) 判栈空操作 isEmpty():判断一个栈是否为空,若栈为空,则返回 true;否则,返回 false。

(3) 求栈中数据元素个数操作 length():返回栈中数据元素的个数。

(4) 取栈顶元素操作 peek():读取栈项元素并返回其值,若栈为空,则返回 null。

(5) 入栈操作 push(x):将数据元素 x 压入栈顶。

(6) 出栈操作 pop():删除并返回栈顶元素。

栈的抽象数据类型用 Java 接口描述如下:

```
public interface IStack {
    public void clear();
    public boolean isEmpty();
    public int length();
    public Object peek();
    public void push(Object x) throws Exception;
    public Object pop();
}
```

下面分别从顺序和链式两种不同的存储结构介绍栈的 Java 接口的两种实现方法,其中采用顺序存储结构的栈称为顺序栈,采用链式存储结构的栈称为链栈。

3.1.3 顺序栈及其基本操作的实现

1. 顺序栈类的描述

与顺序表一样,顺序栈也是用数组来实现的。假设数组名为 stackElem。由于入栈和出栈操作只能在栈顶进行,所以需再加上一个变量 top 来指示栈顶元素的位置。top 有两种定义方式,一种是将其设置为指向栈顶元素存储位置的下一个存储单元的位置,则空栈时,top=0;另一种是将 top 设置为指向栈顶元素的存储位置,则空栈时,top=-1。本书中采用前一种方式来表示栈顶。下面是实现接口 IStack 的顺序栈类的 Java 语言描述。

```
package ch03;
public class SqStack implements IStack {
```

```
    private Object[] stackElem;              // 对象数组
    private int top; // 在非空栈中,top 始终指向栈顶元素的下一个存储位置; 当栈为空时,top 值为 0
    // 栈的构造函数,构造一个存储空间容量为 maxSize 的空栈
    public SqStack(int maxSize) {
        top = 0;                             // 初始化 top 为 0
        stackElem = new Object[maxSize]; // 为栈分配 maxSize 个存储单元
    }
    // 栈置空
    public void clear() {
        top = 0;
    }
    // 判栈是否为空
    public boolean isEmpty() {
        return top == 0;
    }
    // 求栈中数据元素个数
    public int length() {
        return top;
    }
    // 取栈顶元素
    public Object peek() {
        if (!isEmpty())                      // 栈非空
            return stackElem[top-1];     // 返回栈顶元素
        else
            return null;
        }
    // 入栈
    public void push(Object x) throws Exception
        { … }
    // 出栈
    public Object pop()
        { … }
    // 输出栈中所有数据元素(从栈顶元素到栈底元素)
    public void display() {
        for (int i = top - 1; i >= 0; i--)
            System.out.print(stackElem[i].toString() + " ");  // 输出
  }
} // 顺序栈类的描述结束
```

根据上述描述,对于栈(34,12,25,61,30,49)的顺序存储结构可以用图 3.2 表示。

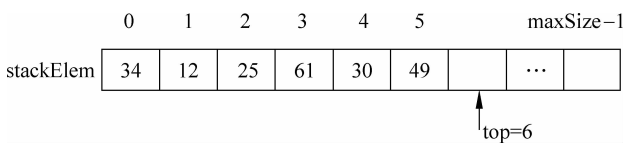


图 3.2 顺序栈的存储结构图

2. 顺序栈基本操作的实现

上面的顺序栈类中已经给出了对顺序栈的置空、判空、求长度和取栈顶元素操作的实现方法。由于 top 指向的是栈顶元素存储位置的下一个存储单元的位置，其值就是栈顶元素在数组中的存储位置编号加 1(数组中位置的编号是从 0 开始)，所以对于顺序栈的置空、判空、求长度和取栈顶元素操作的实现，只要抓住以下几个关键问题，理解起来就非常简单。

(1) 顺序栈为空的条件是 top == 0。

(2) 顺序栈为满的条件是 top == stackElem.length。

(3) 栈的长度为 top。

(4) 栈顶元素就是以 top－1 为下标的数组元素 stackElem[top－1]。

对于顺序栈的入栈和出栈操作的实现分析如下。

1) 顺序栈的入栈操作

入栈操作的基本要求是将数据元素 x 插入顺序栈中，使其成为新的栈顶元素，其中 x 的类型为 Object。完成此处理的主要步骤归纳如下：

(1) 判断顺序栈是否为满，若为满，则抛出异常后结束操作，否则转(2)。

(2) 将新的数据元素 x 存入 top 所指向的存储单元，使其成为新的栈顶元素。

(3) 栈顶指针 top 加 1。

完成(2)和(3)所对应的 Java 语句为：stackElem[top＋＋] ＝ x。

图 3.3 显示了在顺序栈上执行入栈操作时，栈顶元素和栈顶指针的变化情况。

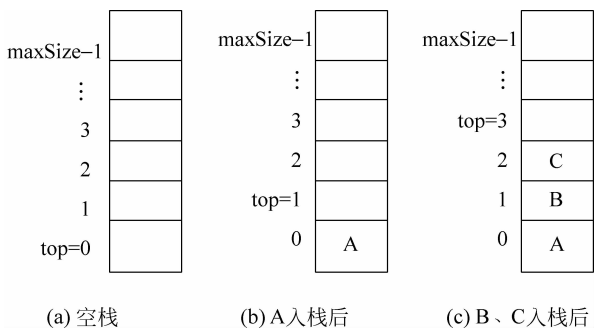


图 3.3　执行入栈操作时栈顶元素和栈顶指针的变化

【算法 3.1】 顺序栈的入栈操作算法。

```
public void push(Object x) throws Exception {
        if (top == stackElem.length)       // 栈满
            throw new Exception("栈已满"); // 抛出异常
        else
            stackElem[top++] = x;          // 先将新的数据元素 x 压入栈顶,再 top 增 1
    } // 算法 3.1 结束
```

2) 顺序栈的出栈操作

出栈操作的基本要求是将栈顶元素从栈中移去，并返回被移去的栈顶元素的值。完成此处理的主要步骤如下：

(1) 判断顺序栈是否为空,若为空,则返回 null,否则转(2)。

(2) 先将 top 减 1,再使栈顶指针指向栈顶元素。

(3) 返回 top 所指示的栈顶元素的值。

完成(2)和(3)所对应的 Java 语句为:

```
return stackElem[ -- top];
```

图 3.4 显示了图 3.3(c)中顺序栈的出栈操作时,栈顶元素和栈顶指针的变化情况。

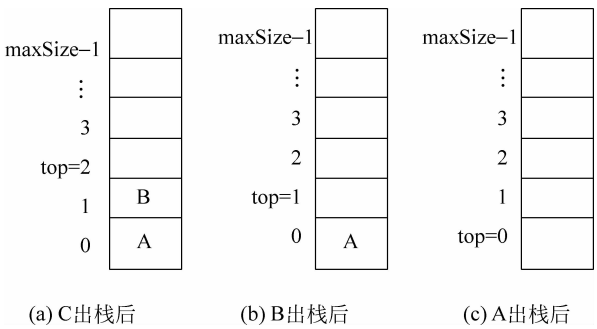


图 3.4 执行出栈操作时栈顶元素和栈顶指针的变化

【算法 3.2】 顺序栈的出栈操作算法。

```
public Object pop() {
        if (isEmpty())                    // 栈空
            return null;
        else                              // 栈非空
            return stackElem[ -- top];
    } // 算法 3.2 结束
```

所有有关顺序栈操作算法的时间复杂度都为 $O(1)$。

3.1.4 链栈及其基本操作的实现

1. 链栈的存储结构

链栈的存储结构可以用不带表点结点的单链表来实现。由于在栈中,入栈和出栈操作只能在栈顶进行,不存在在单链表的任意位置进行插入和删除操作的情况,所以,在链栈中不需要设置头结点,直接将栈顶元素放在单链表的首部成为首结点。图 3.5 给出了链栈的存储结构的示意图,其中,指针 top 指向栈顶元素结点,每一个结点的 next 域存储指向其后继结点的指针。

图 3.5 链栈的存储结构示意图

2. 链栈类的描述

链栈中的结点类引用了前面第 2 章中所讨论过的 Node 类,下面是实现接口 IStack 的链栈类的 Java 语言描述。

```
package ch03;
import ch02.Node;
public class LinkStack implements IStack {
    private Node top;                          // 栈顶元素的引用
    // 将栈置空
    public void clear() {
        top = null;
    }
    // 判链栈是否为空
    public boolean isEmpty() {
        return top == null;
    }
    // 求链栈的长度
    public int length()
        { … }
    // 取栈顶元素并返回其值
    public Object peek() {
        if (!isEmpty())                        // 栈非空
            return top. data;                  // 返回栈顶元素的值
        else
            return null;
    }
     // 入栈
    public void push(Object x) {
        { … }
    // 出栈
    public Object pop()
        { … }
     // 输出栈中所有数据元素(从栈顶元素到栈底元素)
    public void display() {
        Node p = top;                          // 初始化, p 指向栈顶元素
        while (p != null) {                    // 输出所有非空结点的数据元素值
            System.out.print((p.data.toString() + " "));
            p = p.next;                        // p 指针向后移
        }
    }
} // 链栈类的描述结束
```

3. 链栈基本操作的实现

下面开始求链栈的长度、入栈和出栈操作的实现方法。

1) 求链栈的长度操作

求链栈长度操作的基本要求是计算出链栈中所包含的数据元素的个数并返回其值。此操作的基本思想与求单链表的长度相同：引进一个指针 p 和一个计数变量 length，p 的初始状态指向栈顶元素，length 的初始值为 0；然后逐个进行计数，即 p 沿着链栈中的后继指针进行逐个移动，同时 length 逐个加 1，直到 p 指向空为止，此时 length 值即为链栈的长度值。具体的实现算法描述如下：

【算法 3.3】 求链栈的长度操作算法。

```
public int length() {
        Node p = top;                    // 初始化,p 指向栈顶元素,length 为计数器
        int length = 0;
        while (p != null) {              // 从栈顶元素开始向后查找,直到 p 指向空
            p = p.next;                  // p 指向后继结点
            ++length;                    // 长度增加 1
        }
        return length;
    } // 算法 3.3 结束
```

2) 链栈的入栈操作

链栈的入栈操作的基本要求是将数据域值为 x 的新结点插入到链栈的栈顶,使其成为新的栈顶元素,其中,x 的类型为 Object。此操作的基本思想与不带头结点的单链表上的插入操作类似,不相同的仅在于插入的位置对于链栈来说,是限制在表头(栈顶)进行的。链栈的入栈操作的主要步骤归纳如下:

(1) 构造数据域值为 x 的新结点。

(2) 将新结点直接链接到链栈的头部(栈顶),并使其成为新的首结点(栈顶结点)。

图 3.6 显示了链栈的入栈操作后状态的变化情况。

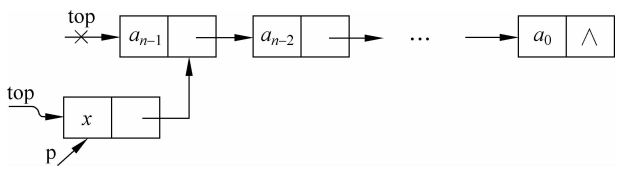


图 3.6 链栈的入栈操作示意图

【算法 3.4】 链栈的入栈操作算法。

```
public void push(Object x) {
        Node p = new Node(x);            // 构造一个新结点
        p.next = top;
        top = p;                         // 新结点成为当前的栈顶结点
    } // 算法 3.4 结束
```

3) 链栈的出栈操作

链栈的出栈操作的基本要求是将首结点(栈顶结点)从链栈中移去,并返回该结点的数据域的值。此操作的基本思想与不带头结点的单链表上的删除操作类似,不相同的在于待删除的结点仅限制为链栈的栈顶结点。链栈的出栈操作的主要步骤归纳如下:

(1) 判断链栈是否为空,若为空,则结束操作并返回 null; 否则,转(2)。

(2) 确定被删结点为栈顶结点。

(3) 修改相关指针域的值,使栈顶结点从链栈中移去,并返回被删的栈顶结点的数据域的值。

图 3.7 显示了链栈出栈操作后状态的变化情况。

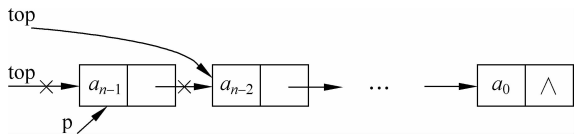


图 3.7　链栈的出栈操作示意图

【算法 3.5】 链栈的出栈操作算法。

```
public Object pop() {
        if (isEmpty()) {
            return null;
          }
        else {
            Node p = top;                    // p 指向被删结点(栈顶结点)
            top = top.next;                  // 修改链指针,使栈顶结点从链栈中移去
            return p.data;                   // 返回栈顶结点的数据域的值
          }
   } // 算法 3.5 结束
```

说明：

(1) 链栈置空、判栈空、取栈顶元素、入栈与出栈操作的时间复杂度都为 $O(1)$，求栈的长度和栈的输出操作的时间复杂度为 $O(n)$，其中 n 为栈的长度。

(2) 顺序栈类与链栈类都实现了接口 IStack 接口，故链栈类的外部接口和顺序栈类的外部接口是完全一样的。

3.1.5　栈的应用

栈是各种软件系统中应用最广泛的数据结构之一，只要涉及先进后出处理特征的问题都可以使用栈式结构。例如：函数递归调用中的地址和参数值的保存、文本编辑器中 undo 序列的保存、网页访问历史的记录保存、在编译软件设计中的括号匹配及表达式求值等问题。下面通过讨论几个栈式结构的具体应用来说明栈在解决实际问题中的运用。

【例 3.1】 分隔符匹配问题：编写判断 Java 语句中分隔符是否匹配的程序。

【问题分析】 分隔符的匹配是任意编译器的一部分，若分隔符不匹配，则程序就不可能正确。Java 程序中有以下分隔符：圆括号“(”和“)”，方括号“[”和“]”，大括号“{”和“}”以及注释分隔符“/*”和“*/”。以下是一些正确使用分隔符的例子：

```
a = b + (c + d) * (e - f);
s[4] = t[a[2]] + u/((i + j) * k);
if (i!= (n[8] + 1)) {p = 7; /* initialize p */ q = p + 2;}
```

以下是一些分隔符不匹配的例子：

```
a = (b + c/(d * e) * f;                          // 左括号多余
s[4] = t[a[2]] + u/(i + j) * k);                 // 右括号多余
while (i!= (n[8] + 1)] {p = 7; /* initialize p */ q = p + 2;}  // 左右括号不匹配
```

一个分隔符和它所匹配的分隔符可以被其他的分隔符分开，即分隔符允许嵌套。因此，

一个给定的右分隔符只有在其前面的所有右分隔符都被匹配上后才可以进行匹配。例如：条件语句 if (i!=(n[8]+1))中，第一个左圆括号必须与最后一个右圆括号相匹配，而且这只有在第二个左圆括号与倒数第二个右圆括号相匹配后才能进行；依次地，第二个括号的匹配也只有在第三个左方括号与倒数第三个右方括号匹配后才能进行。可见，最先出现的左分隔符在最后才能进行匹配，这个处理与栈式结构的先进后出的特性相吻合。分隔符匹配的算法归纳如下：

从左到右扫描 Java 语句，从语句中不断地读取字符，每次读取一个字符，若发现它是左分隔符，则将它压入栈中；当从输入中读到一个右分隔符时，则弹出栈顶的左分隔符，并且查看它是否和右分隔符匹配，若它们不匹配，则匹配失败，程序报错；若栈中没有左分隔符与右分隔符匹配(即栈为空)，或者一直存在没有被匹配的左分隔符，则匹配失败，程序报错，左分隔符没有被匹配，表现为把所有的字符都读入后，栈中仍留有左分隔符；若所有的字符读入结束后，栈为空(即所有左分隔符都已经匹配)，则表示匹配成功。

【程序代码】

```
package ch03;
import java.util.Scanner;
public class Example3_1 {
    private final int LEFT = 0;              // 记录分隔符为"左"分隔符
    private final int RIGHT = 1;             // 记录分隔符为"右"分隔符
    private final int OTHER = 2;             // 记录其他字符
    // 判断分隔符的类型,有 3 种:"左"、"右"、"非法"
    public int verifyFlag(String str) {
        if ("(".equals(str) || "[".equals(str) || "{".equals(str)
            || "/*".equals(str))             // 左分隔符
            return LEFT;
        else if (")".equals(str) || "]".equals(str) || "}".equals(str)
            || "*/".equals(str))             // 右分隔符
            return RIGHT;
        else                                 // 其他的字符
            return OTHER;
    }
    // 检验左分隔符 str1 和右分隔符 str2 是否匹配
    public boolean matches(String str1, String str2) {
        if (("(".equals(str1) && ")".equals(str2))
                || ("[".equals(str1) && "]".equals(str2))
                || ("{".equals(str1) && "}".equals(str2))
                || ("/*".equals(str1) && "*/".equals(str2)))  // 匹配规则
            return true;
        else
            return false;
    }
    private boolean isLegal(String str) throws Exception {
        if (!"".equals(str) && str != null) {
            SqStack S = new SqStack(100); // 新建最大存储空间为 100 的顺序栈
            int length = str.length();
            for (int i = 0; i < length; i++) {
                char c = str.charAt(i);   // 指定索引处的 char 值
                String t = String.valueOf(c);       // 转化成字符串型
                if (i != length) {                  // c 不是最后一个字符
```

```
                if (('/' == c && '*' == str.charAt(i + 1))|| ('*' == c &&
                        '/' == str.charAt(i + 1))) {  // 是分隔符"/*"或"*/"
                    t = t.concat(String.valueOf(str.charAt(i + 1)));
                                                  // 与后一个字符相连
                    ++i;                          // 跳过一个字符
                }
            }
            if (LEFT == verifyFlag(t)) {          // 为左分隔符
                S.push(t);                        // 压入栈
            }
            else if (RIGHT == verifyFlag(t)) {    // 为右分隔符
                if (S.isEmpty() || !matches(S.pop().toString(), t)) {
                                                  // 右分隔符与栈顶元素不匹配
                    throw new Exception("错误：Java 语句不合法!");   // 抛出异常
                }
            }
        }
        if (!S.isEmpty())                         // 栈中存在没有匹配的字符
            throw new Exception("错误：Java 语句不合法!");          // 抛出异常
        return true;
        }
    else
        throw new Exception("错误：Java 语句为空!");                  // 抛出异常
}

public static void main(String[] args) throws Exception {
    Example3_1 e = new Example3_1();
    System.out.println("请输入分 Java 语句：");
    Scanner sc = new Scanner(System.in);
    if (e.isLegal(sc.nextLine()))
        System.out.println("Java 语句合法!");
    else
        System.out.println("错误：Java 语句不合法!");
}
}
```

【运行结果】

运行结果分别如图 3.8 和图 3.9 所示。

图 3.8　例 3.1 程序的运行结果 I

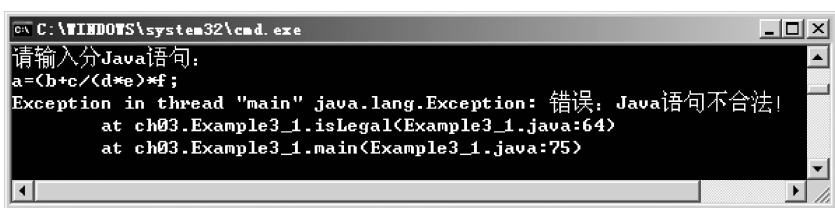


图 3.9　例 3.1 程序的运行结果 II

【例 3.2】 大数加法问题：编程实现两个大数的加法运算。

【问题分析】 整数是有最大上限的。所谓大数是指超过整数最大上限的数，例如 18 452 543 389 943 209 752 345 473 和 8 123 542 678 432 986 899 334 就是两个大数，它们是无法用整型变量来保存的，更不用说保存它们相加的和了。为解决两个大数的求和问题，可以把两个加数看成是数字字符串，将这些数的相应数字存储在两个堆栈中，并从两个栈中弹出对应位的数字依次执行加法却可得到结果。图 3.10 显示了以 784 和 8465 为例进行加法的计算过程。

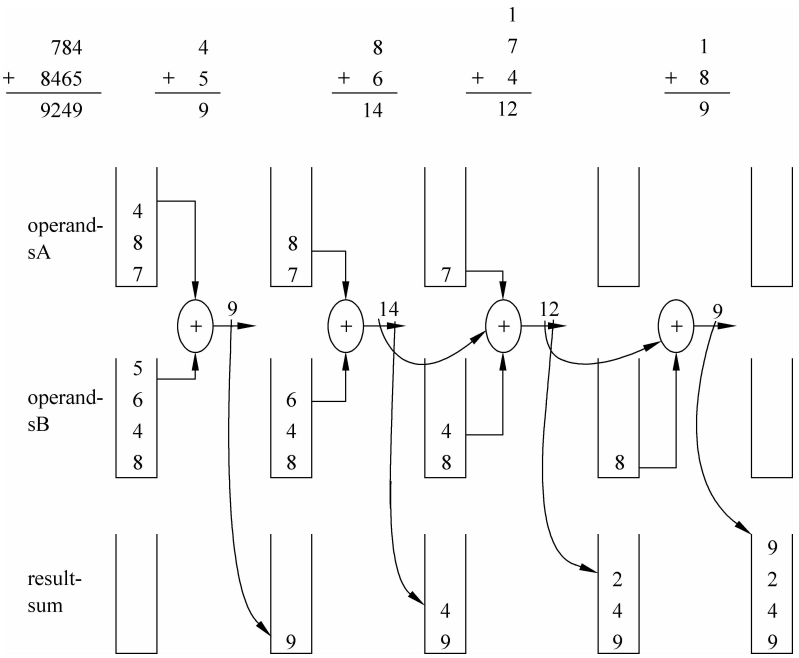


图 3.10 使用堆栈将 784 和 8465 相加

对于两个大数的加法，其操作步骤归纳如下：

(1) 将两个加数的相应位从高位到低位依次压入栈 sA 和 sB 中。

(2) 若两个加数栈都非空，则依次从栈中弹出栈顶数字相加，和存入变量 partialSum 中；若和有进位，则将和的个位数压入结果栈 sum 中，并将进位数加到下一位数字相加的和中；若和没有进位，则直接将和压入结果栈 sum 中。

(3) 若某个加数堆栈为空，则将非空加数栈中的栈顶数字依次弹出与进位相加，和的个位数压入结果栈 sum 中，直到此该栈为空为止。若最高位有进位，则最后将 1 压入栈 sum 中。

(4) 若两个加数栈都为空，则栈 sum 中保存的就是计算结果。注意栈顶是结果中的最高位数字。

【程序代码】

```
package ch03;
public class Example3_2 {
// 求两个大数的和,加数和被加数以字符串的形式输入(允许大数中出现空格),计算的结果也以字
```

```
// 符串的形式返回
    public String add(String a, String b) throws Exception {
        LinkStack sum = new LinkStack();          // 大数的和
        LinkStack sA = numSplit(a);               // 加数字符串以单个字符的形式放入栈中
        LinkStack sB = numSplit(b);               // 被加数字符串以单个字符的形式放入栈中
        int partialSum;                           // 对于两个位的求和
        boolean isCarry = false;                  // 进位标示
        while (!sA.isEmpty() && !sB.isEmpty()) {  // 加数和被加数栈同时非空
            partialSum = (Integer) sA.pop() + (Integer) sB.pop();
                      // 对于两个位求和,并在栈中去除加数和被加数中的该位
            if (isCarry) {                        // 低位进位
                partialSum++;                     // 进位加到此位上
                isCarry = false;                  // 重置进位标示
            }
            if (partialSum >= 10) {               // 需要进位
                partialSum -= 10;
                sum.push(partialSum);
                isCarry = true;                   // 标示进位
             }
             else {                               // 位和不需要进位
                sum.push(partialSum);             // 和放入栈中
             }
        }
        LinkStack temp = !sA.isEmpty() ? sA : sB;// 引用指向加数和被加数中非空栈
        while (!temp.isEmpty()) {
            if (isCarry) {                        // 最后一次执行加法运算中需要进位
                int t = (Integer) temp.pop();     // 取出加数或被加数没有参加的位
                ++t; // 进位加到此位上
                if (t >= 10) {                    // 需要进位
                    t -= 10;
                    sum.push(t);
                }
                else {
                    sum.push(t);
                    isCarry = false;              // 重置进位标示
                }
            }
            else
                // 最后一次执行加法运算中不需要进位
                sum.push(temp.pop());             // 把加数或被加数中非空的值放入和中
        }
        if (isCarry) {                            // 最高位需要进位
            sum.push(1);                          // 进位放入栈中
        }
        String str = new String();
        while (!sum.isEmpty())
            // 把栈中元素转化成字符串
            str = str.concat(sum.pop().toString());
```

```
        return str;
    }
    // 字符串以单个字符的形式放入栈中,并去除字符串中空格,返回以单个字符为元素的栈
    public LinkStack numSplit(String str) throws Exception {
        LinkStack s = new LinkStack();
        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);              // 指定索引处的 char 值
            if (' ' == c)                        // 去除空格
                continue;
            else if ('0'<= c && '9'>= c)         // 数字放入栈中
                s.push(Integer.valueOf(String.valueOf(c)));
            else                                 // 非法数字字符
                throw new Exception("错误: 输入了非数字型字符!");
        }
        return s;
    }
    public static void main(String[] args) throws Exception {
        Example3_2 e = new Example3_2();
        System.out.println("两个大数的和为: "
                + e.add("18 452 543 389 943 209 752 345 473",
                        "8 123 542 678 432 986 899 334"));   // 输出运算结果
    }
}
```

【运行结果】

运行结果如图 3.11 所示。

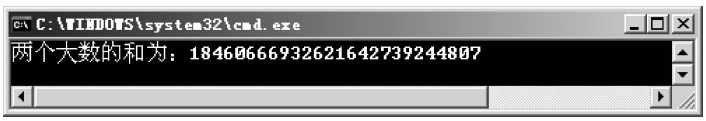


图 3.11 例 3.2 程序的运行结果

【例 3.3】 表达式求值问题：编程实现算术表达式求值。

【问题分析】 算术表达式是由操作数、算术运算符和分隔符所组成的式子。为了方便，下面的讨论仅限于含有二元运算符且操作数是一位整数的算术表达式的运算。

表达式一般有中缀表达式、后缀表达式和前缀表达式共 3 种表示形式，其中，中缀表达式是将运算符放在两个操作数的中间，这正是人们平时书写算术表达式的一种描述形式；后缀表达式(也称逆波兰表达式)是将运算符放在两个操作数之后，而前缀表达式是将运算符放在两个操作数之前。例如：中缀表达式 A＋(B－C/D)＊E，对应的后缀表达式为 ABD/－E＊＋，对应的前缀表达式为＋A＊－B/CDE。

由于运算符有优先级，所以在计算机内部使用中缀表达式描述时，对计算是非常不方便的，特别是带括号时更麻烦。而后缀表达式中既无运算符优先级又无括号的约束问题，因为在后缀表达式中运算符出现的顺序正是计算的顺序，所以计算一个后缀表达式的值要比计算一个中缀表达式的值简单得多。由此，求算术表达式的值可以分成两步来进行，第一步先

将原算术表达式转换成后缀表达式，第二步再对后缀表达式求值。下面分别给出这两个步骤的实现方法。

1. 将原算术表达式转换成后缀表达式

由于原算术表达式与后缀表达式中的操作数所出现的先后次序是完全一样的，只是运算符出现的先后次序不一样，所以转换的重点放在运算符的处理上。首先设定运算符的优先级如表 3.1 所示。

表 3.1　运算符的优先级

运算符	（（左括号）	+（加）、-（减）	*（乘）、/（除）、%（取模）	^（幂）
优先级	0	1	2	3

表 3.1 中的优先级别从高到低依次用 0～3 的数字来表示，数字越大，表示其运算符的优先级越高。

要使运算符出现的次序与真正的算术运算顺序一致，就要使优先数高的以及括号内的运算符出现在前，所以在把算术表达式转换成后缀表达式时，要使用一个栈来保留还未送往后缀表达式的运算符，此栈称为运算符栈。实现算法的基本思想如下：

(1) 初始化一个运算符栈。

(2) 从算术表达式输入的字符串中从左到右读取一个字符。

(3) 若当前字符是操作数，则直接送往后缀表达式。

(4) 若当前字符是左括号“(”时，将其压进运算符栈。

(5) 若当前字符为运算符时，则：

① 当运算符栈为空，则将其压入运算符栈。

② 当此运算符的优先数高于栈顶运算符，则将此运算符压入运算符栈；否则，弹出栈顶运算符送往后缀式，并将当前运算符压栈，重复步骤(5)。

(6) 若当前字符是右括号“)”时，反复将栈顶符号弹出，并送往后缀表达式，直到栈顶符号是左括号为止，再将左括号出栈并丢弃。

(7) 若读取还未完毕，则跳转到(2)。

(8) 若读取完毕，则将栈中剩余的所有运算符弹出并送往后缀表达式。

利用上述的转换规则，将算术表达式(A + B) * (C - D)/E^F + G%H，转换成后缀表达式的过程如表 3.2 所示。

表 3.2　算术表达式转换成后缀表达式的过程

步骤	算术表达式	运算符栈	后缀表达式	规　则
1	(A+B) * (C-D)/E^F+G%H	(		是左括号，压栈
2	A+B) * (C-D)/E^F+G%H	(	A	是操作数，送往后缀表达式
3	+B) * (C-D)/E^F+G%H	(+	A	是运算符且优先级高于栈顶运算符，压栈
4	B) * (C-D)/E^F+G%H	(+	AB	是操作数，送往后缀表达式

续表

步骤	算术表达式	运算符栈	后缀表达式	规　　则
5	)*(C－D)/E^F＋G％H		AB＋	是右括号,将栈中左括号之前的所有运算符送往后缀表达式并将栈中左括号弹出
6	*(C－D)/E^F＋G％H	*	AB＋	是运算符且栈为空,压栈
7	(C－D)/E^F＋G％H	*(	AB＋	是左括号,压栈
8	C－D)/E^F＋G％H	*(	AB＋C	是操作数,送往后缀表达式
9	－D)/E^F＋G％H	*(－	AB＋C	是运算符且优先级高于栈顶运算符,压栈
10	D)/E^F＋G％H	*(－	AB＋CD	是操作数,送往后缀表达式
11	)/E^F＋G％H	*	AB＋CD－	是右括号,将栈中左括号之前的所有运算符弹出送往后缀表达式并将栈中左括号弹出
12	/E^F＋G％H	/	AB＋CD－*	是运算符且优先级等于栈顶运算符,则弹出栈顶运算符送往后缀式,并将当前运算符压栈
13	E^F＋G％H	/	AB＋CD－*E	是操作数,送往后缀表达式
14	^F＋G％H	/^	AB＋CD－*E	是运算符且优先级高于栈顶运算符,压栈
15	F＋G％H	/^	AB＋CD－*EF	是操作数,送往后缀表达式
16	＋G％H	＋	AB＋CD－*EF^/	是运算符且优先级低于栈顶运算符,则重复弹出优先级更高的栈顶运算符送往后缀式,再将当前运算符压栈
17	G％H	＋	AB＋CD－*EF^/G	是操作数,送往后缀表达式
18	％H	＋％	AB＋CD－*EF^/G	是运算符且优先级高于栈顶运算符,压栈
19	H	＋％	AB＋CD－*EF^/GH	是操作数,送往后缀表达式
20	结束		AB＋CD－*EF^/GH％＋	弹出栈中剩余项并送往后缀表达式

2. 计算后缀表达式的值

要计算后缀表达式的值比较简单,只要先找到运算符,再去找前面最后出现的两个操作数,从而构成一个最小的算术表达式进行运算,在计算过程中也需利用一个栈来保留后缀表达式中还未参与运算的操作数,此栈也称为操作数栈。计算后缀表达式值的算法的基本思想如下:

(1) 初始化一个操作数栈。

(2) 从左到右依次读入后缀表达式中的字符:

① 若当前字符是操作数，则压入操作数栈。

② 若当前字符是运算符，则从栈顶弹出两个操作数并分别作为第 2 个操作数和第 1 个操作数参与运算，再将运算结果压入操作数栈内。

(3) 重复步骤(2) 直到读入的后缀表达式结束为止，则操作数栈中的栈顶元素即为后缀表达式的计算结果。

按照以上的算法，可以在计算机内首先完成表达式的转换，然后计算表达式的值，这样计算机在计算时相对比较节省系统资源。

【程序代码】

```
package ch03;
public class Example3_3 {
    //将算术表达式转换为后缀表达式的函数,结果以字符串的形式返回
    public String convertToPostfix(String expression) throws Exception {
        LinkStack st = new LinkStack();     // 初始化一个运算符栈
        String postfix = new String();      // 用于存放输出的后缀表达式
        for (int i = 0; expression != null && i < expression.length(); i++) {
            char c = expression.charAt(i); // 从算术表达式中读取一个字符
            if ('' != c) {                  // 字符 c 不为空格
                if (isOpenParenthesis(c)) {
                    st.push(c);             // 为开括号,压栈
                    }
                else if (isCloseParenthesis(c)) {     // 为闭括号
                         char ac = (Character) st.pop();   // 弹出栈顶元素
                         while (!isOpenParenthesis(ac)) {   // 一直到为开括号为止
                            postfix = postfix.concat(String.valueOf(ac));
                                        // 串联到后缀表达式的结尾
                            ac = (Character) st.pop();
                         }
                 else if (isOperator(c)) {  // 为运算符
                   if (!st.isEmpty()) { // 栈非空,取出栈顶优先级高的运算符送往后缀表达式
                        char ac = (Character) st.pop();
                        while (ac != null&& priority(ac.charValue()) >= priority(c)) {
                               postfix = postfix.concat(String.valueOf(ac));
                            ac = (Character) st.pop();
                           }
                        if (ac != null) {  // 若最后一次取出的优先级低的操作符,重新压栈
                            st.push(ac);
                        }
                   }
                   st.push(c);              // 压栈
                  }
                else {                      // 为操作数,串联到后缀表达式的结尾
                    postfix = postfix.concat(String.valueOf(c));
                 }
             }
        }
        while (!st.isEmpty())               // 栈中剩余的所有操作符串联到后缀表达式的结尾
            postfix = postfix.concat(String.valueOf(st.pop()));
        return postfix;
```

```
}

// 对后缀表达式进行求值计算的函数
public double numberCalculate(String postfix) throws Exception {
    LinkStack st = new LinkStack();
    for (int i = 0; postfix != null && i < postfix.length(); i++) {
        char c = postfix.charAt(i);    // 从后缀表达式中读取一个字符
        if (isOperator(c)) {           // 当为操作符时
            // 取出两个操作数
            double d2 = Double.valueOf(st.pop().toString());
            double d1 = Double.valueOf(st.pop().toString());
            double d3 = 0;
            if ('+' == c) {            // 加法运算
                d3 = d1 + d2;
            } else if ('-' == c) {     // 减法运算
                d3 = d1 - d2;
            } else if ('*' == c) {     // 乘法运算
                d3 = d1 * d2;
            } else if ('/' == c) {     // 除法运算
                d3 = d1 / d2;
            } else if ('^' == c) {     // 幂运算
                d3 = Math.pow(d1, d2);
            } else if ('%' == c) {
                d3 = d1 % d2;
            }
            st.push(d3);
        } else {                       // 当为操作数时
            st.push(c);
        }
    }
    return (Double) st.pop();          // 返回运算结果
}

// 判断字符串是否为运算符
public boolean isOperator(char c) {
    if ('+' == c || '-' == c || '*' == c || '/' == c || '^' == c
            || '%' == c) {
        return true;
    } else {
        return false;
    }
}

// 判断字符串是否为开括号
public boolean isOpenParenthesis(char c) {
    return '(' == c;
}
```

```
    // 判断字符串是否为闭括号
    public boolean isCloseParenthesis(char c) {
        return ')' == c;
    }

    // 判断运算法的优先级
    public int priority(char c) {
        if (c == '^') {                          // 为幂运算
            return 3;
          }
        if (c == '*' || c == '/' || c == '%') { // 为乘、除、取模运算
            return 2;
          }
        else if (c == '+' || c == '-') {         // 为加、减运算
              return 1;
            }
            else {                               // 其他
                return 0;
          }
    }
public static void main(String[] args) throws Exception {
    Example3_3 p = new Example3_3();
    String postfix = p.convertToPostfix("(1 + 2) * (5 - 2)/2 ^ 2 + 5 % 3");
                                                 // 转化为后缀表达式
    System.out.println("表达式的结果为: " + p.numberCalculate(postfix));
                                                 // 对后缀表达式求值后,并输出
    }
}
```

【运行结果】

运行结果如图 3.12 所示。

图 3.12　例 3.3 程序的运行结果

【例 3.4】 栈与递归问题：编程实现汉诺塔(Hanoi)问题的求解。

n 阶汉诺塔问题：假设有 3 个分别命名为 X、Y 和 Z 的塔座，在塔座 X 上插有 n 个直径大小各不相同，且从小到大编号为 1、2、…、n 的圆盘。现要求将塔座 X 上的 n 个圆盘借助塔座 Y 移至塔座 Z 上，并仍按同样顺序叠排。圆盘移动时必须遵循下列规则：

(1) 每次只能移动一个圆盘。

(2) 圆盘可以插在 X、Y 和 Z 中的任何一个塔座上。

(3) 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

【问题分析】 当 $n=1$ 时，问题比较简单，只要将编号为 1 的圆盘从塔座 X 直接移动到

塔座 Z 上即可；当 $n>1$ 时，需利用塔座 Y 作辅助塔座，若能先设法将压在编号为 n 的圆盘上的 $n-1$ 个圆盘从塔座 X 移到塔座 Y 上，则可将编号为 n 的圆盘从塔座 X 移至塔座 Z 上，然后将塔座 Y 上的 $n-1$ 个圆盘移至塔座 Z 上。而如何将 $n-1$ 个圆盘从一个塔座移至另一个塔座是一个和原问题具有相同特征属性的问题，只是问题规模小于 1，因此可以用同样的方法求解。由此可知，求解 n 阶汉诺塔问题可以用递归分解的方法来进行。

【程序代码】

```
package ch03;
public class Example3_4 {
    private int c = 0;                          // 全局变量,对搬动计数
    // 将塔座 x 上按直径由小到大且自上而下的编号为 1 至 n 的 n 个圆盘按规则移到塔座 z 上,y
    // 用作辅助塔座
    public void hanoi(int n, char x, char y, char z) {
        if (n == 1) {
            move(x, 1, z);                      // 将编号为 1 的圆盘从 x 移到 z
         }
         else {
            hanoi(n - 1, x, z, y);              // 将 x 上编号为 1 至 n-1 的圆盘移到 y,z 作辅助塔
            move(x, n, z);                      // 将编号为 n 的圆盘从 x 移到 z
            hanoi(n - 1, y, x, z);              // 将 y 上编号为 1 至 n-1 的圆盘移到 z,x 作辅助塔
        }
    }
    // 移动操作,将编号为 n 的圆盘从 x 移到 z
    public void move(char x, int n, char z) {
        System.out.println("第" + ++c + "次移动: " + n + "号圆盘," + x + "->" + z);
    }
    public static void main(String[] args) {
        Example3_4 h = new Example3_4();
        h.hanoi(3, 'x', 'y', 'z');              // 对于圆盘数量为 3 进行移动
    }
}
```

【运行结果】

运行结果如图 3.13 所示。

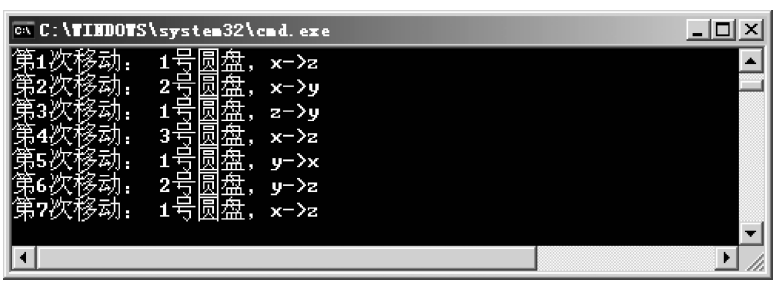


图 3.13　例 3.4 程序的运行结果

图 3.14 给出了 Hanoi(3,X,Y,Z)运行过程中圆盘的移动情况。

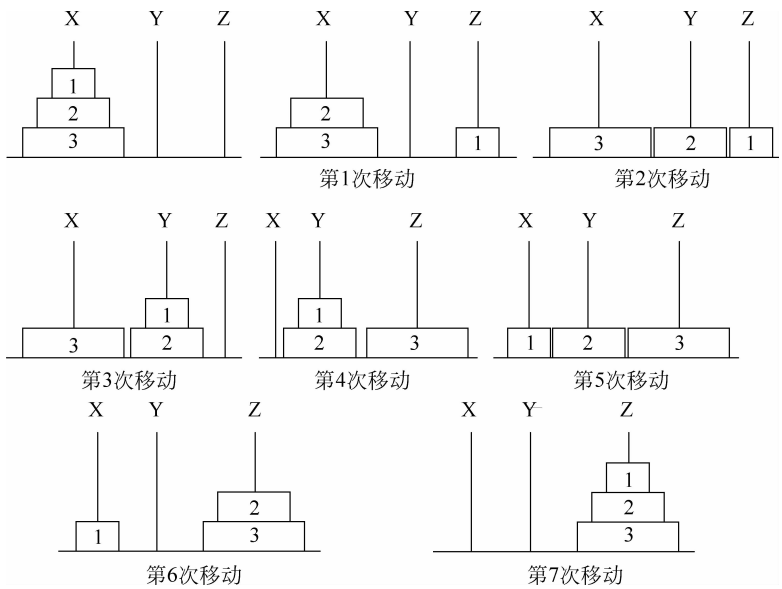


图 3.14　Hanoi(3,X,Y,Z)运行过程中圆盘的移动情况示意图

3.2　队列

3.2.1　队列的概念

队列是另一种特殊的线性表,它的特殊性体现在队列只允许在表尾插入数据元素,在表头删除数据元素,所以队列也是一种操作受限的特殊的线性表,它具有先进先出(First In First Out, FIFO)或后进后出(Last In Last Out,LILO)的特性。

允许进行插入的一端称为是队尾(rear),允许进行删除的一端称为是队首(front)。假设队列中的数据元素序列为$\{a_0,a_1,a_2,\cdots,a_{n-1}\}$,则其中$a_0$称为队首元素,$a_{n-1}$称为队尾元素,$n$为队列中数据元素的个数,当$n=0$时,称为空队列。队列的插入操作通常称为入队操作,而删除操作通常称为出队操作,如图 3.15 所示。

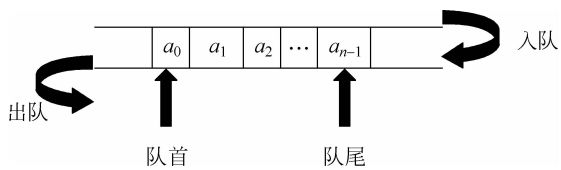


图 3.15　队列及其操作的示意图

队列在现实生活中处处可见,例如,人在食堂排队买饭、人在车站排队上车;汽车排队进站等。这些排队都有一个规则就是按先后顺序,后来的只能在队列的最后排队,先来的先处理再离开,不能插队。在生产建设中也有队列的应用,例如:生产计划的调度就是根据一

个任务队列进行。队列也经常应用于计算机领域中,例如:操作系统中存在各种队列,有资源等待队列、作业队列等。

3.2.2 队列的抽象数据类型描述

队列也是由 $n(n\geqslant 0)$ 个具有相同类型的数据元素所构成的有限序列。队列的基本操作与栈类似,有以下 6 种:

(1) 清空队列操作 clear():将一个已经存在的队列置成空队列。

(2) 判空操作 isEmpty():判断一个队列是否为空,若为空,则返回 true;否则,返回 false。

(3) 求队列长度操作 length():返回队列中数据元素的个数。

(4) 取队首元素操作 peek():读取队首元素并返回其值。

(5) 入队操作 offer(x):将数据元素 x 插入到队列中使其成为新的队尾元素。

(6) 出队操作 poll():删除队首元素并返回其值,若队列为空,则返回 null。

队列的抽象数据类型用 Java 接口描述如下:

```
public interface IQueue {
    public void clear();
    public boolean isEmpty();
    public int length();
    public Object peek();
    public void offer(Object x) throws Exception;
    public Object poll();
    }
```

同栈一样,队列也有顺序和链式两种存储结构。顺序存储结构的队列称为顺序队列,链式存储结构的队列称为链队列。

3.2.3 顺序队列及其基本操作的实现

1. 顺序队列的存储结构

与顺序栈类似,在顺序队列的存储结构中,需要分配一块地址连续的存储区域来依次存放队列中从队首到队尾的所有元素。这样也可以使用一维数组来表示,假设数组名为 queueElem,数组的最大容量为 maxSize,由于队列的入队操作只能在当前队列的队尾进行,而出队操作只能在当前队列的队首进行,所以需加上变量 front 和 rear 来分别指示队首和队尾元素在数组中的位置,其初始值都为 0。在非空栈中,front 指向队首元素,rear 指向队尾元素的下一个存储位置。图 3.16 是一个在具有 6 个连续存储空间的顺序队列上进行入队、出队操作后的动态示意图。

图 3.16 中描述了一个从空队列开始,先后经过 A、B、C 入队列;A、B 出队列;E、F、G 入队列操作后,队列的顺序存储结构状态。

初始化队列时,令 front=rear=0;入队时,直接将新的数据元素存入 rear 所指的存储单元中,然后将 rear 值加 1;出队时,直接取出 front 所指的存储单元中数据元素的值,然后将 front 值加 1。

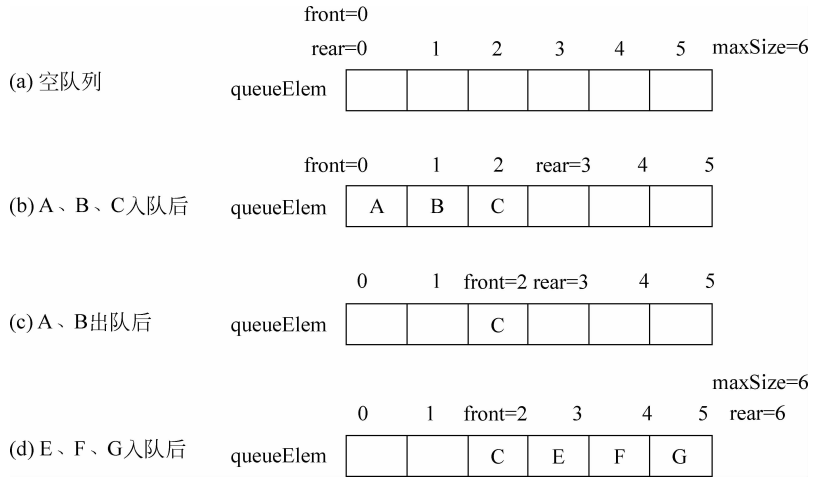


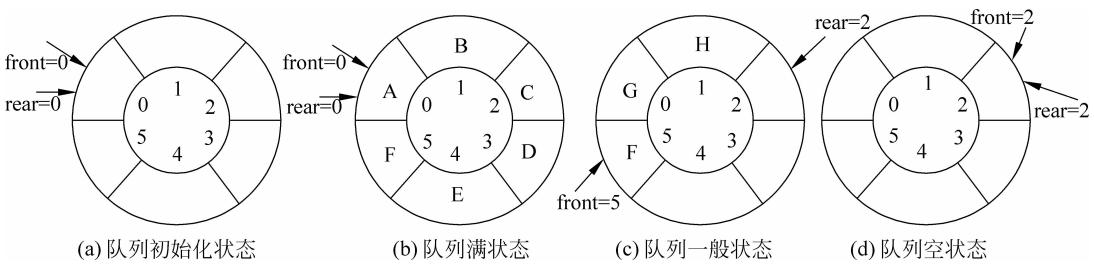
图 3.16　顺序队列的入队、出队操作的动态示意图

从图 3.16(d)可以看出，若此时还需要将数据元素 H 入队，H 应该存放于 rear＝6 的位置处，顺序队列则会因数组下标越界而引起“溢出”，但此时顺序队列的首部还空出了两个数据元素的存储空间。因此，这时的“溢出”并不是由于数组空间不够而产生的溢出。这种因顺序队列的多次入队和出队操作后出现有存储空间，但不能进行入队操作的溢出现象称为“假溢出”。

要解决“假溢出”问题，最好的办法就是把顺序队列所使用的存储空间看成是一个逻辑上首尾相连的循环队列。当 rear 或 front 到达 maxSize－1 后，再加 1 就自动到 0。这种转换可利用 Java 语言中对整型数据求模（或取余）运算来实现，即令 rear＝(rear＋1)％maxSize。显然，当 rear＝maxSize－1 时，rear 加 1 后，rear 的值就为 0。这样，就不会出现顺序队列数组的头部有空的存储空间，而队尾却因数组下标越界而引起的假溢出现象。

2. 循环顺序队列类的描述

假设 maxSize＝6，循环顺序队列的初始化状态如图 3.17(a)所示，此时有 front == rear == 0；当 A、B、C、D、E、F 分别入队后，循环顺序队列为满，如图 3.17(b)所示，此时有 front == rear；当 A、B、C、D、E 出队，而 G、H 又入队后，循环顺序队列的状态如图 3.17(c)所示，此时 front＝5，rear＝2；再将 F、G、H 出队后，循环顺序队列为空，如图 3.17(d)所示，此时有 front == rear。为此，在循环顺序队列中就引发出一个新的问题：无法区分队空和队满的状态，这是因为循环顺序队列的判空和判满的条件都是 front == rear。



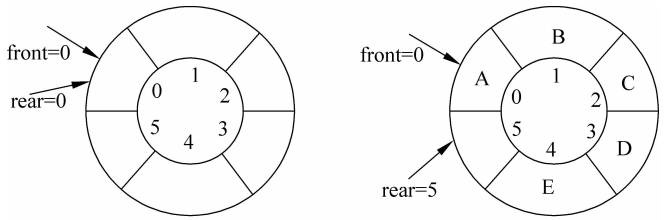
(a) 队列初始化状态　(b) 队列满状态　(c) 队列一般状态　(d) 队列空状态

图 3.17　循环顺序队列的 4 种状态图

解决循环顺序队列的队空和队满的判断问题通常可采用下面3种方法：

1）少用一个存储单元

当顺序存储空间的容量为maxSize时，只允许最多存放maxSize－1个数据元素。图3.18为这种情况下的队空和队满的两种状态图。此时，队空的判断条件为：front == rear，而队满的判断条件为：front == (rear＋1)％maxSize。



(a) 队列空状态　　(b) 队列满状态

图3.18　少用一个存储单元时循环顺序队列的两种状态图

2）设置一个标志变量

在程序设计过程中引进一个标志变量flag，其初始值置为0，每当入队操作成功后就置flag＝1；每当出队操作成功后就置flag＝0，则此时队空的判断条件为：front == rear && flag == 0，而队满的判断条件为：front == rear && flag == 1。

3）设置一个计数器

在程序设计过程中引进一个计数变量num，其初始值置为0，每当入队操作成功后就将计数变量num的值加1；每当出队操作成功后就将计数变量num的值减1，则此时队空的判断条件为：num == 0，而队满的判断条件为：num＞0 && front == rear。

本书中采用第一种方法来判断队空和队满的状态，下面是实现接口IQueue的循环顺序队列类的Java语言的描述。

```
public class CircleSqQueue implements IQueue {
    private Object[] queueElem;           // 队列存储空间
    private int front;                    // 队首的引用,若队列不空,指向队首元素
    private int rear;                  // 队尾的引用,若队列不空,指向队尾元素的下一个存储位置
    // 循环队列类的构造函数
    public CircleSqQueue(int maxSize) {
        front = rear = 0;                 // 队首、队尾初始化为0
        queueElem = new Object[maxSize]; // 为队列分配maxSize个存储单元
    }
    // 队列置空
    public void clear() {
        front = rear = 0;

    }
    // 判队列是否为空
    public boolean isEmpty() {
        return front == rear;
    }
    // 求队列的长度
    public int length() {
```

```
        return (rear - front + queueElem.length) % queueElem.length;
    }
    // 读取队首元素
    public Object peek() {
        if (front == rear)                  // 队列为空
            return null;
        else
            return queueElem[front];        // 返回队首元素
    }
    // 入队
    public void offer(Object x) throws Exception
        { … }
    // 出队
    public Object poll()
        { … }
    // 输出队列中的所有数据元素(从队首到队尾)
    public void display() {
        if (!isEmpty()) {
            for (int i = front; i != rear; i = (i + 1) % queueElem.length)
                System.out.print(queueElem[i].toString() + " ");
        } else {
            System.out.println("此队列为空");
        }
    }
} // 循环顺序队列类的描述结束
```

3. 循环顺序队列基本操作的实现

下面仅给出循环顺序队列的入队和出队操作的实现方法,其他操作的实现算法比较简单,所以在上述的类 CircleSqQueue 中已直接给出。

1) 循环顺序队列的入队操作

入队操作的基本要求是将新的数据元素 x 插入到循环顺序队列的尾部,使其成为新的队尾元素,其中 x 的类型为 Object。实现此操作的主要步骤归纳如下:

(1) 判断循环顺序队列是否为满,若满,则抛出异常后结束操作;否则转(2)。

(2) 先将新的数据元素 x 存入 rear 所指示的数组存储位置中,使其成为新的队尾元素,再将 rear 值循环加 1,使 rear 始终指向队尾元素的下一个存储位置。对应的 Java 语句为:

```
queueElem[rear] = x ;                    // 将 x 存入 rear 所指的数组存储位置中
rear = (rear + 1) % queueElem.length;    // rear 值循环加 1
```

【算法 3.6】 循环顺序队列的入队操作算法。

```
public void offer(Object x) throws Exception {
        if ((rear + 1) % queueElem.length == front)// 队列满
            throw new Exception("队列已满");        // 抛出异常
        else {
            queueElem[rear] = x;
                                  // x 存入 rear 所指的数组存储位置中,使其成为新的队尾元素
         rear = (rear + 1) % queueElem.length;    // 修改队尾指针
```

```
        }
    } // 算法 3.6 结束
```

2) 循环顺序队列的出队操作

出队操作的基本要求是将队首元素从循环顺序队列中移去,并返回被移去的队首元素的值。实现此操作的主要步骤归纳如下:

(1) 判断循环顺序队列是否为空,若为空,则返回空值;否则转(2)。

(2) 先取出 front 所指的队首元素的值,再将 front 值循环加 1。对应的 Java 语句为:

```
t = queueElem[front];                    // 取出队首元素
front = (front + 1) % queueElem.length; // front 值循环加 1
```

【算法 3.7】 循环顺序队列的出队操作算法。

```
public Object poll() {
    if (front == rear)                   // 队列为空
        return null;
    else {
        Object t = queueElem[front];
        front = (front + 1) % queueElem.length;
        return t;                        // 返回队列的队首元素
    }
} // 算法 3.7 结束
```

入队与出队操作算法的时间复杂度都为 $O(1)$,其他基本操作算法的时间复杂度也都为 $O(1)$。

3.2.4 链队列及其基本操作的实现

1. 链队列的存储结构

队列的链式存储结构也用不带头结点的单链表来实现。为了便于实现入队和出队操作,需要引进两个指针 front 和 rear 来分别指向队首元素和队尾元素的结点。图 3.19 为队列 $a_0, a_1, \cdots, a_{n-1}$ 的链式存储结构图。

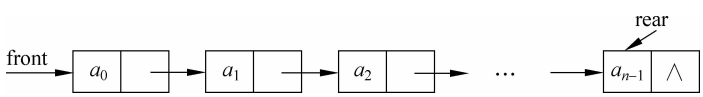


图 3.19 链队列的存储结构示意图

2. 链队列类的描述

链队列中的结点类也引用了前面所讨论过的 Node 类,下面是实现接口 IQueue 的链队列类的 Java 语言描述。

```
package ch03;
import cho2.Node;
public class LinkQueue implements IQueue {
```

```
    private Node front;                    // 队首指针
    private Node rear;                     // 队尾指针
    // 链队列类的构造函数
    public LinkQueue() {
        front = rear = null;
    }
    // 队列置空
    public void clear() {
        front = rear = null;
    }
    // 队列判空
    public boolean isEmpty() {
        return front == null;
    }
    // 求队列的长度
    public int length() {
            Node p = front;
            int length = 0;
            while (p != null) {
                p = p.next;                // 指针下移
                ++length;                  // 计数器加 1
        } return length;
    }
    // 取队首元素
    public Object peek() {
        if (front != null)                 // 队列非空
            return front.data;             // 返回队首结点的数据域值
        else
            return null;
    }
    // 入队
    public void offer(Object x)
        { … }

    // 出队
    public Object poll() {
        if (front != null) {               // 队列非空
            Node p = front;                // p 指向队首结点
            front = front.next;            // 队首结点出列
            if (p == rear)                 // 被删除的结点是队尾结点时
                    rear = null;
            return p.data;                 // 返回队首结点的数据域值
            }
        else
            return null;
    }
}
```

3. 链队列基本操作的实现

由于链队列的置空、判空、求长度和出队操作与链栈相应操作的实现方法相同，所以下

面仅介绍链队列入队操作的实现方法。

链队列入队操作的基本要求是将数据元素 x 所对应的结点插入到链队列的尾部,使其成为新的队尾结点,其中 x 的类型为 Object。此操作的基本思想也与不带头结点的单链表上的插入操作类似,不相同之处为链队列的插入位置是限制在表尾进行。实现此操作的主要步骤归纳如下:

(1) 创建数据域值为 x 的新结点。

(2) 判断链队列是否为空,若为空,则直接将新结点置为队首和队尾结点,否则就将新结点链接到队列的尾部并使其成为新的队尾结点。

图 3.20 显示了链队列入队操作后状态的变化情况。

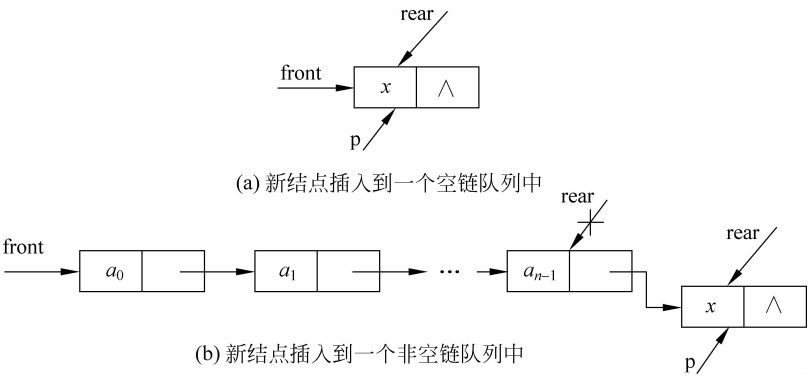


图 3.20 链队列的入队操作示意图

【算法 3.8】 链队列的入队操作算法。

```
public void offer(Object x) {
    Node p = new Node(x);                  // 初始化新结点
    if (front != null) {                   // 队列非空
        rear.next = p;
        rear = p;                          // 改变队尾的位置
    }
    else
        front = rear = p;
} // 算法 3.8 结束
```

3.2.5 队列的应用

由于队列是一种具有先进先出特性的线性表,所以在现实世界中,当求解具有先进先出特性的问题时可以使用队列。例如:操作系统中各种数据缓冲区的先进先出管理;应用系统中各种任务请求的排队管理;软件设计中对树的层次遍历和对图的广度遍历过程等,都需用使用队列。队列的应用非常广泛,下面只通过讨论求解素数环问题和实现键盘输入循环缓冲区的管理两个例子来说明队列在解决实际问题中的应用。

【例 3.5】 编程实现求解素数环问题。

【问题描述】 将 1～n 的 n 个自然数排列成环形,使得每相邻两数之和为素数,从而构成一个素数环。

【问题提示】

(1) 先引入顺序表类 Sqlist 和链队列类 LinkQueue,再创建 Sqlist 类的一个对象 L 作为顺序表,用于存放素数环的数据元素;创建 LinkQueue 类的一个对象 Q,作为队列用于存放还未加入到素数环中的自然数。

(2) 初始化顺序表 L 和队列 Q:将 1 加入到顺序表 L 中,将 2～n 的自然数全部加入到 Q 队列中。

(3) 将出队的队首元素 p 与素数环最后一个数据元素 q 相加,若两数之和是素数并且 p 不是队列中的最后一个数据元素(或队尾元素),则将 p 加入到素数环中;否则说明 p 暂时无法处理,必须再次入队等待,再重复此过程。若 p 为队尾元素,则还需要判断它与素数环的第一个数据元素相加的和数是否为素数,若是素数,则将 p 加入到素数环,求解结束;若不是素数,则重复(3),直到队列为空或已对队列中每一个数据元素都遍历了一次且未能加入到素数环为止。

【程序代码】

```
package ch03;
import ch02.SqList;                          // 引入第 2 章中的顺序表类 SqList
public class Example3_5 {
    // 判断正整数是否为素数
    public boolean isPrime(int num) {
        if (num == 1)                        // 整数 1 返回 false
            return false;
        Double n = Math.sqrt(num);           // 求平方根
        for (int i = 2; i <= n.intValue(); i++)
            if (num % i == 0 )               // 模为 0 返回 false
                return false;
        return true;
    }
    // 求 n 个正整数的素数环,并以顺序表返回
    public SqList makePrimeRing(int n) throws Exception {
        if (n % 2 != 0)                      // n 为奇数则素数环不存在
            throw new Exception("素数环不存在!");
        SqList L = new SqList(n);            // 构造一个顺序表
        L.insert(0, 1);                      // 初始化顺序表的首结点为 1
        LinkQueue Q = new LinkQueue();       // 构造一个链队列
        for (int i = 2; i <= n; i++)         // 初始化链队列
            Q.offer(i);
        return insertRing(L, Q, 2, n);       // 返回素数环
    }
    // 在一个顺序表中插入第 m 个数,使其与顺序表中第 m - 1 个数的和为素数,若 m 等于 n,则还
    // 要满足第 m 个数与 1 的和也为素数,程序返回顺序表
    public SqList insertRing(SqList L, LinkQueue Q, int m, int n)
            throws NumberFormatException, Exception {
        int count = 0;          // 记录遍历队列中的数据元素的个数,防止在一次循环中重复遍历
        while (!Q.isEmpty() && count <= n - m) {       // 队列非空,且未重复遍历队列
            int p = (Integer) Q.poll();
            int q = (Integer) L.get(L.length() - 1);  // 取出顺序表中的最后一个数据元素
            if (m == n) {                    // 为队列中的最后一个元素
```

```
                if (isPrime(p + q) && isPrime(p + 1)) {  // 满足素数环的条件
                    L.insert(L.length(),p);插入到顺序表尾
                    return L;
                }
                else                    // 不满足素数环条件
                    Q.offer(p);
            }
            else if (isPrime(p + q)) {// 未遍历到队列的最后一个数据元素,且满足素数环条件
                    L.insert(L.length(),p);插入到顺序表尾
                    if (insertRing(L, Q, m + 1, n) != null)// 递归调用函数,若返回值不为
                                                           // 空,即已成功找到素数环,返回
                        return L;
                    L.remove(L.length() );// 移除顺序表表尾位置的数据元素
                    Q.offer(p);
            }
            else
                Q.offer(p);             // 加入的队列尾部
            ++count;                    // 遍历次数增 1
        }
        return null;
    }
    public static void main(String[] args) throws Exception {
        Example3_5 r = new Example3_5(); // 构造素数环对象
        SqList L = r.makePrimeRing(6);  // 求素数环
        for (int i = 0; i < L.length(); i++)
            System.out.print(L.get(i) + " ");
    }
} // 例 3.5 程序代码结束
```

【运行结果】

运行结果如图 3.21 所示。

图 3.21　例 3.5 程序的运行结果

3.2.6　优先级队列

优先级队列是一种带有优先级的队列,它是一种比栈和队列更为专用的数据结构。与普通队列一样,优先级队列有一个队首和一个队尾,并且也是从队首删除数据元素,但不同的是优先级队列中数据元素按关键字的值有序排列。由于在很多情况下,需要访问具有最小关键字值的数据元素(例如要寻找最便宜的方法或最短的路径去做某件事),因此,约定关键字最小的数据元素(或者在某些实现中是关键字最大的数据元素)具有最高的优先级,并且总是排在队首。在优先级队列中数据元素的插入也不仅仅限制在队尾进行,而是顺序插

入到队列的合适位置，以确保队列的优先级顺序。

优先级队列在很多情况下都很有用。一方面，程序员经常采用优先级队列。例如：在第 5 章中，构造哈夫曼树算法中就应用了优先级队列；另一方面，在某些计算机系统中优先级队列也有很多应用。例如：在抢先式多任务操作系统中，程序被排列在优先级队列中，这样优先级最高的程序就会先得到时间片并得以运行。

1. 优先级队列类的描述

优先级队列也可以采用顺序和链式两种存储结构。考虑到在优先级队列中，既要保证快速地访问到优先级高的数据元素，又要保证可以实现较快的插入操作，所以通常以链式存储结构实现优先级队列。数据元素的优先级别的高低依据优先数的大小来鉴定，优先数越小，优先级别就越大。优先队列中结点的 data 类描述如下：

```
package ch03;
public class PriorityQData {
    public Object elem;                    // 结点的数据元素值
    public int priority;                   // 结点的优先数
    // 构造函数
    public PriorityQData(Object elem, int priority) {
        this.elem = elem;
        this.priority = priority;
    }

}                                          // 优先队列中结点的数据域 data 类描述结束
```

实现 IQueue 接口的类描述如下：

```
import ch02.Node;
public class PriorityQueue implements IQueue {
    private Node front;                    // 队首的引用
    private Node rear;                     // 队尾的引用
    // 优先队列类的构造函数
    public PriorityQueue() {
        front = rear = null;
    }
    // 队列置空
    public void clear() {
        front = rear = null;
    }
    // 队列判空
    public boolean isEmpty() {
        return front == null;
    }
    // 求队列长度
    public int length() {
        Node p = front;
        int length = 0;                    // 队列的长度
        while (p != null) {                // 一直查找到队尾
            p = p.next;
```

```
            ++length;                    // 长度增加 1
        }
        return length;
    }
    //入队
    public void offer(Object x) {
        PriorityQData pn = (PriorityQData) x;
        Node s = new Node(pn);           // 构造一个新结点
        if (front == null)               // 队列为空
            front = rear = s;            // 修改队列的首尾结点
        else {
            Node p = front, q = front;
        while (p != null&& pn.priority >= ((PriorityQData) p.data).priority) {
                                         // 新结点的数据域值与队列结点的数据域值相比较
                q = p;
                p = p. next;
            }
            if (p == null) {             // p 为空,表示遍历到了队列尾部
                rear.next = s;           // 将新结点加入到队尾
                rear = s;                // 修改队尾指针
            }
            else if (p == front) {       // p 的优先级大于队首结点的优先级
                s.next = front;          // 将新结点加入到队首
                front = s;               // 修改队首结点
            }
            else {                       // 新结点加入队列中部
                q.next = s;
                s.next = p;
            }
        }
    }
    // 读取队首元素
    public Object peek() {
        if (front == null)               // 队列为空
            return null;
        else                             // 返回队首结点的数据域值
            return front.data;
    }
    // 出队
    public Object poll() {
        if (front == null)               // 队列为空
            return null;
        else {                                  // 返回队首结点的数据域值,并修改队首指针
            Node p = front;
            front = p.next;
            return p.data;
        }
    }
    // 输出所有队列中的所有数据元素(从队首到队尾)
    public void display() {
        if (!isEmpty()) {
```

```
            Node p = front;
            while (p != rear.next) {      // 从队首到队尾
                PriorityQData q = (PriorityQData) p.data;
                System.out.println(q.elem + " " + q.priority);
                p = p.next;
            }
        }
        else {
            System.out.println("此队列为空");
        }
    }
} // 优先队列类描述结束
```

2. 优先级队列的应用

优先级队列的应用也很广泛。例如，计算机操作系统用一个优先队列来实现进程的调度管理。在一系列等待执行的进程中，每一个进程可以用一个数值来表示它的优先级，优先级越高，这个值越小。优先级高的进程应该最先获得处理器。另一个例子是打印机的输出任务队列。对于先后到达的打印几百页和只有几页的任务，一个合理的方法是先打印页数少的任务，后打印页数多的任务，这样做就是按照文件的大小来排列打印任务的优先顺序。下面以第一个例子为例模拟优先级队列的实现过程。

【例 3.6】 设计一个程序模仿操作系统的进程管理问题。要求按进程服务优先级高的先服务、优先级相同的按先到先服务的原则进行管理。

【问题描述】 操作系统中采用一个优先队列来管理进程。当优先级队列中有多个进程排队等待系统响应时，只要 CPU 空闲，进程管理系统就会从优先队列中找出优先级最高的进程首先出队并占用 CPU 资源，即按进程服务的优先级，优先级高的先服务、优先级相同的按先到先服务的原则管理。

【问题提示】 操作系统中每个进程的模仿数据由进程号和进程优先级两部分组成，进程号是每个不同进程的唯一标识，优先级通常是一个 0 到 40 的数值，规定 0 为优先级最高，40 为优先级最低。下面为一组模拟数据：

进程号	进程优先级
1	20
2	40
3	0
4	10
5	40

此问题只要通过优先级队列的入队和出队操作即可得到解决，所以只要直接引用前面的优先级队列类 PriorityQueue 中的相应方法即可。

【程序代码】

```
package ch03;
public class Example3_6 {
        public static void main(String[] args) {
```

```
        PriorityQueue pm = new PriorityQueue();        // 构造一个对象
        pm.offer(new PriorityQData(1, 20));            // 插入优先级队列
        pm.offer(new PriorityQData(2, 30));
        pm.offer(new PriorityQData(3, 20));
        pm.offer(new PriorityQData(4, 20));
        pm.offer(new PriorityQData(5, 40));
        pm.offer(new PriorityQData(6, 10));
        System.out.println("进程服务的顺序: ");
        System.out.println("进程 ID 进程优先级");
        while (!pm.isEmpty()) {                  // 从队首到队尾,输出结点的数据域值和优先级
            PriorityQData p = (PriorityQData) pm.poll();   // 移除队首结点,并返回其数据域值
            System.out.println(" " + p.elem + "\t" + p.priority);  // 输出
        }
    }
}
```

【运行结果】

运行结果如图 3.22 所示。

图 3.22 例 3.6 程序的运行结果

优先级队列可以用多种方法加以实现,例如采用无序线性表或有序线性表。上例中就是用有序线性表的链式存储结构来实现。虽然都是查找到优先级最小(或最大)的数据元素,但是插入或删除的时间代价很大。堆是一种很好的优先级队列的实现方法。因为最小堆建成之后,其堆顶元素满足了关键码最小的要求,这就为快速查找和删除优先级最高的数据元素创造了条件。有关堆的知识将在后面章节中进行介绍。

3.3 栈与队列的比较

栈与队列既是两种重要的线性结构,也是两种操作受限的特殊线性表。为了让读者能够更好地掌握它们的使用特点,在此对栈和队列做一个比较。

1. 栈与队列的相同点

(1) 都是线性结构,即数据元素之间具有“一对一”的逻辑关系。
(2) 插入操作都是限制在表尾进行。
(3) 都可在顺序存储结构和链式存储结构上实现。
(4) 在时间代价上,插入与删除操作都需常数时间;在空间代价上,情况也相同。

（5）多链栈和多链队列的管理模式可以相同。在计算机系统软件中，经常会出现同时管理和使用两个以上栈或队列的情况，若采用顺序存储结构实现栈和队列，将会给处理带来极大的不便，因而一般采用多个单链表来实现多个栈或队列。图 3.23、图 3.24 是多链栈和多链队列的存储结构的示意图。它们将多个链栈的栈顶指针或链队列的队首、队尾指针分别存放在一个一维数组中，从而很方便地实现了统一管理和使用多个栈或队列。

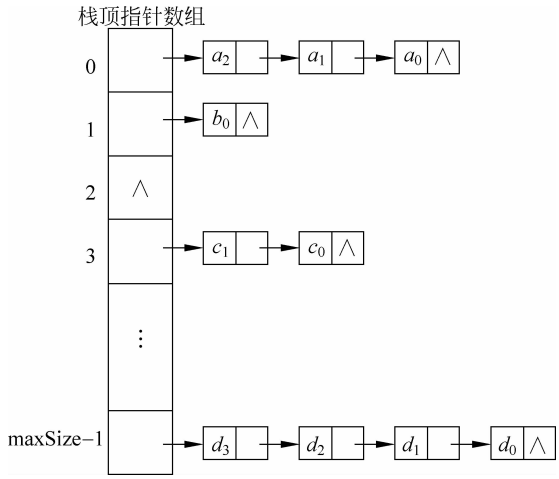


图 3.23　多链栈的存储结构示意图

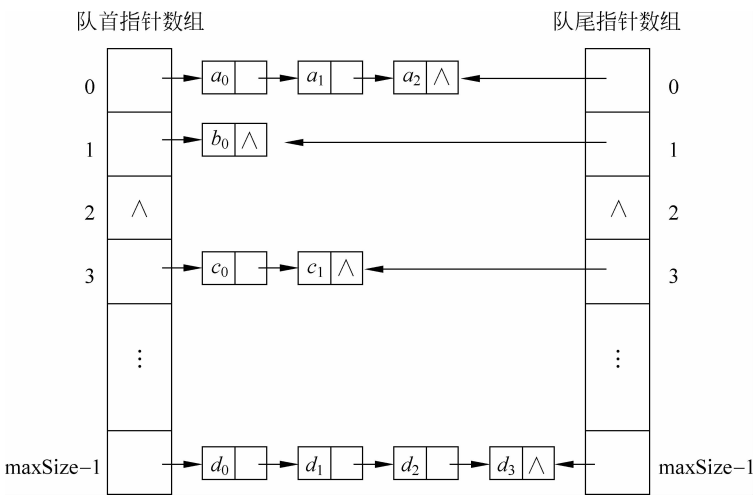


图 3.24　多链队列的存储结构示意图

2. 栈与队列的不同点

（1）删除数据元素操作的位置不同。栈的删除操作控制在表尾进行，而队列的删除操作控制在表头进行。

（2）两者的应用场合不相同。具有后进先出（或先进后出）特性的应用需求，可使用栈式结构进行管理。例如：递归调用中现场信息、计算的中间结果和参数值的保存；图与树

的深度优先搜索遍历都采用栈式存储结构加以实现。而具有先进先出(后进后出)特性的应用需求,则要使用队列结构进行管理。例如:消息缓冲器的管理;操作系统中对内存、打印机等各种资源进行管理,都使用了队列,并且可以根据不同优先级别的服务请求,按优先类别把服务请求组成多个不同的队列;队列也是图和树在广度搜索遍历过程中采用的数据结构。

(3) 顺序栈可实现多栈空间共享,而顺序队列则不同。实际应用中经常会出现在一个程序中需要同时使用两个栈或队列的情况。若采用顺序存储,就可以使用一个足够大的数组空间来存储多个栈,即让多个栈共享同一存储空间。图 3.25 是两个栈共享空间的示意图。其中,把数组的两端设置为两栈各自的栈底,两栈的栈顶从两端开始向中间延伸。可以充分利用顺序栈单向延伸的特性,使两个栈的空间形成互补,从而提高存储空间的利用率。然而,对于顺序队列就不能像顺序栈那样在同一个数组中存储两个队列,除非总有数据元素从一个队列转入另一个队列。

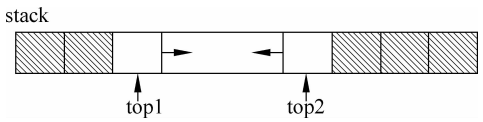


图 3.25 两个栈共享同一个数组空间

3.4 栈与队列的综合应用举例

栈和队列这两种特殊的线性表与基本线性表一样,在实际工作和生活中有着广泛的应用。本节通过一个实例再一次给出栈与队列的综合应用,使读者能更进一步地区分栈与队列的特性和它们各自的实现技巧。

【例 3.7】 停车场管理问题。

【问题描述】 假设停车场是一个可停放 n 辆车的狭长通道,并且只有一个大门可供汽车进出。在停车场内,汽车按到达的先后次序,由北向南依次排列(假设大门在最南端)。若车场内已停满 n 辆车,则后来的汽车需在门外的便道上等候,当有车开走时,便道上的第一辆车即可驶入。当停车场内某辆车要离开时,在它之后进入的车辆必须先退出车场为它让路,待该辆车开出大门后,其他车辆再按原次序返回车场。每辆车离开停车场时,应按其停留时间的长短交费(在便道上停留的时间不收费)。

试编写程序,模拟上述管理过程。

【问题分析】 由于停车场中某辆车的离开是按在它之后进入的车辆必须先退出车场为它让路的原则下进行,显然满足了“后进先出”的特性,所以可以用栈式结构来模拟停车场,而对于指定停车场,它的车位是相对固定的,因而本题采用顺序栈来加以实现。又由于便道上的车是按先到先开进停车场的原则进行,即满足“先进先出”的特性,所以可以用队列来模拟便道,而对于便道上停放车辆的数目是不固定的,因而本题采用链队列来加以实现。

为了能更好地模拟停车场管理,可以从终端读入汽车到达或离去的数据,每组数据应该包括三项:①是“到达”还是“离去”;②汽车牌照号码;③“到达”或“离去”的时刻。与每组输入信息相应的输出信息为:若是到达的车辆,则输出其在停车场中或便道上的位置;若是离去的车辆,则输出其在停车场中停留的时间和应交纳的费用。

【程序代码】

```
import java.text.DecimalFormat;
import java.util.GregorianCalendar;
import java.util.Scanner;
public class Example3_7 {
    private SqStack S = new SqStack(100);          // 顺序栈存放停车场内的车辆信息
    private LinkQueue Q = new LinkQueue();         // 链队列存放便道上的车辆信息
    private double fee = 2;                        // 每分钟停车费用
    public final static int DEPARTURE = 0;         // 标识车辆离开
    public final static int ARRIVAL = 1;           // 标识车辆到达
    // 内部类用于存放车辆信息
    public class CarInfo {
        public int state;                          // 车辆状态,离开/到达
        public GregorianCalendar arrTime;          // 车辆达到时间
        public GregorianCalendar depTime;          // 车辆离开时间
        public String license;                     // 车牌号
    }
    // 停车场管理,参数 license 表示车牌号码,action 表示此车辆的动作到达或离开
    public void parkingManag(String license, String action) throws Exception {
        if ("arrive".equals(action)) {             // 车辆到达
            CarInfo info = new CarInfo();          // 构造一个车辆信息实例
            info.license = license;                // 修改车辆状态
            if (S.length() < 100) {                // 停车场未满
                info.arrTime = (GregorianCalendar) GregorianCalendar
                        .getInstance();            // 当前时间初始化到达时间
                info.state = ARRIVAL;
                S.push(info);
                System.out.println(info.license + "停放在停车场第" + S.length()
                        + "个位置!");
            }
            else {                                 // 停车场已满
                Q.offer(info);                     // 进入便道队列
                System.out.println(info.license + "停放在便道第" + Q.length()
                        + "个位置!");
            }
        }
        else if ("depart".equals(action)) {        // 车辆离开
            CarInfo info = null;
            int location = 0;                      // 车辆的位置
            SqStack S2 = new SqStack(S.length());
            // 构造一个新栈用于存放因车辆离开而导致的其他车辆暂时退出车场
            for (int i = S.length(); i > 0; i--) {
                info = (CarInfo) S.pop();
                if (info.license.equals(license)) {  // 将离开的车辆
                    info.depTime = (GregorianCalendar) GregorianCalendar.getInstance();
                    // 当前时间来初始化离开时间
                    info.state = DEPARTURE;
                    location = i;                  // 取得车辆位置信息
                    break;
                }
                else                               // 其他车辆暂时退出车场
                    S2.push(info);
            }
```

```
            while (!S2.isEmpty())                    // 其他车辆重新进入停车场
                S.push(S2.pop());
            if (location != 0) {                     // 停车场内存在指定车牌号码的车辆
                double time = (info.depTime.getTimeInMillis() - info.arrTime
                .getTimeInMillis())/(1000 * 60); // 计算停放时间,并把毫秒换算成分钟
                DecimalFormat df = new DecimalFormat("0.0");  // 对 double 进行格式化,保留
                                                              // 两位有效小数
                System.out.println(info.license + "停放:" + df.format(time)
                        + "分钟,费用为:" + df.format(time * fee));   // 输出
            }
            if (!Q.isEmpty()) {                      // 便道上的第一辆车进入停车场
                info = (CarInfo) Q.poll();
                info.arrTime = (GregorianCalendar) GregorianCalendar
                        .getInstance();              // 当前时间来初始化离开时间
                info.state = ARRIVAL;
                S.push(info);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        Example3_8 pm = new Example3_8();            // 构造对象
        for (int i = 1; i <= 12; i++)
        // 初始化 12 辆车,车牌号分别为 1、2、…、12,其中有 10 辆车停在停车场内,两辆车停放在便道上
            pm.parkingManag(String.valueOf(i), "arrive");
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入车牌号:");
        String license = sc.next();
        System.out.println("arrive or depart ?");
        String action = sc.next();
        pm.parkingManag(license, action);            // 调用停车场管理函数
    }
}
```

【运行结果】

运行结果如图 3.26 所示。

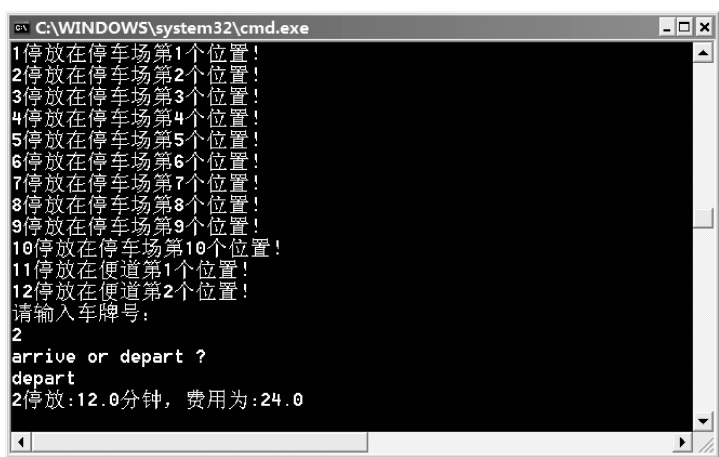


图 3.26　例 3.7 程序的运行结果

小结

本章介绍了两种特殊的线性表：栈和队列。栈是插入和删除操作限制在表尾一端进行，无论是在栈中插入数据元素还是删除栈中数据元素，都只能固定在线性表的表尾进行。通常将进行插入和删除操作的这一端称为“栈顶”；而另一端称为“栈底”，它是一种具有“后进先出”或“先进后出”特性的线性表。队列是插入操作只限制在表尾进行，而删除操作只限制在表头进行。通常将允许进行插入操作的一端称为“队尾”，而将允许进行删除操作的另一端称为“队首”，它是一种具有“先进先出”或“后进后出”的线性表。

栈的基本操作主要包括判栈是否为空、出栈、入栈和取栈顶元素等。队列的基本操作主要包括判队列是否为空、出队、入队和取队首元素等。这些操作的时间复杂度都是 $O(1)$。

栈与队列都可以用顺序和链式两种存储方式加以实现，为此有顺序栈和链栈、顺序队列和链队列之分。其中，顺序栈和顺序队列用数组实现；链栈和链队列则是用单链表进行存储，读者要注意的是，链栈结点中指针的指向是从栈顶开始依次向后链接的，也就是说链栈中结点的指针不像单链表那样是指向它在线性表中的后继，而是指向其在线性表中的前驱。链队列结点的链接方向与单链表相同，但为了便于实现插入和删除操作，链队列除了引进一个队首指针外，还引进了队尾指针来指向队尾元素，并将两者封装在一个类中。

顺序栈比链栈的使用更为广泛。在顺序栈中，注意掌握入栈和出栈操作，特别是在入栈操作前要进行的判满条件和在出栈操作前要进行的判空条件。在链栈中的操作与单链表操作类似，而且由于入栈与出栈操作都是固定在链栈的栈顶位置进行，所以实现起来比单链表相应操作更为简单。

循环队列比非循环队列使用更为广泛。在顺序存储方式下，也要注意掌握队列的入队和出队操作、队列判空与判满的条件以及队列“假溢出”的处理方法。循环顺序队列就是为了避免“假溢出”现象而提出的一种队列。它是一个假想的环，是通过模运算来使其首尾相连。特别要注意的是，在循环顺序队列中的入队和出队操作实现与在非循环顺序队列中的入队和出队操作实现的不同点在于队首和队尾指针的变化不再是简单的加 1 或减 1，而需加 1 或减 1 后再取模运算。在循环顺序队列中为了区分队列的判空和判满条件，特别提出了 3 种解决方法：第一种是少用一个存储单元；第二种是设置一个标志变量；第三种是设置一个计数变量。链队列中的操作也与单链表中的操作类似，而且由于入队操作总是固定在链队列的队尾进行，而出队操作总是固定在链队列的队首进行，所以链队列的入队和出队操作实现起来非常简单。

栈与队列是两种十分重要的，并且应用非常广泛的数据结构。常见的栈的应用包括括号匹配问题的求解、表达式的转换和求值、函数调用和递归实现和深度优先搜索遍历等。凡是遇到对数据元素的读取顺序与处理顺序相反，都可考虑使用栈将读取到而又未处理的数据元素保存在栈中。常见队列的应用包括计算机系统中各种资源的管理、消息缓冲器的管理和广度优先搜索遍历等。凡是遇到对数据元素的读取顺序与处理顺序相同，都可考虑使用队列来保存读取到而未处理的数据元素。

优先级队列是带有优先级的队列。优先级队列中的每一个数据元素都有一个优先权，优先权可以比较大小，它既可以在数据元素被插入到优先级队列时被人为赋予，也可以是数

据元素本身所具有的某一属性。优先权的大小决定着该对象接受服务的先后顺序,所以也将其称为优先级。优先级队列不同于一般的队列,优先级队列是按照数据元素优先级的高低来决定出队的次序,而不是按照数据元素进入队列的次序来决定的。一般队列也可以被看作是一种特殊的优先级队列。在一般的队列中,数据元素的优先级是由其进入队列的时间确定的,时间越长,优先级越高。优先级队列的实现方法可以采用有序、无序线性表或堆来实现。本章中只介绍了在有序线性表上实现的优先级队列。在这种情况下,入队操作是进行数据元素的有序插入,即将待插入的数据元素插入到队列中的适当位置,并使插入后的队列仍按照优先级从大到小的顺序排放。入队操作的时间复杂度为 $O(n)$。出队操作只要将队首元素(即优先级最高的元素)从队列中删除即可,其时间复杂度为 $O(1)$。在一些实际应用中,若需要采用一种数据结构来存储数据元素,对这种数据结构的要求是:数据元素加入的次序是无关紧要的,但每次取出的数据元素应是具有最高优先级的数据元素,这时就可以采用优先级队列来解决问题。

其实利用栈和队列的思想还可以设计出其他一些变种的栈和队列结构。例如:双端队列、双端栈、超队列和超栈等,这些都是根据插入与删除操作位置受限的不相同而得名的。双端队列是指插入和删除操作限制在线性表的两端进行;双栈是指两个底部相连的栈,它是一种添加限制的双端队列,并且规定从一端插入的数据元素只能从这一端删除;超队列是一种删除受限的双端队列,删除操作只允许在一端进行,而插入操作可在两端进行;超栈是一种插入受限的双端队列,插入操作只限制在一端,而删除操作允许在两端进行。这些变种的栈和队列在某些特定情况下具有很好的应用价值。

习题 3

一、选择题

1. 在栈中存取数据的原则是(　　)。

A. 先进先出　　B. 先进后出

C. 后进后出　　D. 没有限制

2. 若将整数 1、2、3、4 依次进栈,则不可能得到的出栈序列是(　　)。

A. 1234　　B. 1324　　C. 4321　　D. 1423

3. 在链栈中,进行出栈操作时(　　)。

A. 需要判断栈是否满　　B. 需要判断栈是否为空

C. 需要判断栈元素的类型　　D. 无须对栈作任何差别

4. 在顺序栈中,若栈顶指针 top 指向栈顶元素的下一个存储单元,且顺序栈的最大容量是 maxSize,则顺序栈的判空条件是(　　)。

A. top == 0　　B. top == -1

C. top == maxSize　　D. top == maxSize−1

5. 在顺序栈中,若栈顶指针 top 指向栈顶元素的下一个存储单元,且顺序栈的最大容量是 maxSize。则顺序栈的判满的条件是(　　)。

A. top == 0　　B. top == -1

C. top == maxSize　　D. top == maxSize－1

6. 在队列中存取数据元素的原则是(　　)。

A. 先进先出　　B. 先进后出　　C. 后进后出　　D. 没有限制

7. 在循环顺序队列中,假设以少用一个存储单元的方法来区分队列判满和判空的条件,front 和 rear 分别为队首和队尾指针,它们分别指向队首元素和队尾元素的下一个存储单元,队列的最大存储容量为 maxSize,则队列的判空条件是(　　)。

A. front == rear　　B. front！＝rear

C. front == rear＋1　　D. front == (rear＋1)％ maxSize

8. 在循环顺序队列中,假设以少用一个存储单元的方法来区分队列判满和判空的条件,front 和 rear 分别为队首和队尾指针,它们分别指向队首元素和队尾元素的下一个存储单元,队列的最大存储容量为 maxSize,则队列的判满条件是(　　)。

A. front == rear　　B. front！＝rear

C. front == rear＋1　　D. front == (rear＋1)％ maxSize

9. 在循环顺序队列中,假设以少用一个存储单元的方法来区分队列判满和判空的条件,front 和 rear 分别为队首和队尾指针,它们分别指向队首元素和队尾元素的下一个存储单元,队列的最大存储容量为 maxSize,则队列的长度是(　　)。

A. rear－front

B. rear－front＋1

C. (rear－front＋maxSize)％maxSize

D. (rear－front＋1)％maxSize

10. 设长度为 n 的链队列采用单循环链表加以表示,若只设一个头指针指向队首元素,则入队操作的时间复杂度为(　　)。

A. $O(1)$　　B. $O(n)$　　C. $O(\log_2 n)$　　D. $O(n^2)$

二、填空题

1. 栈是一种操作受限的特殊线性表,其特殊性体现在其插入和删除操作都限制在_____进行。允许插入和删除操作的一端称为_____,而另一端称为_____。栈具有_____的特点。

2. 栈也有两种存储结构,一种是_____ ,另一种是_____;以这两种存储结构存储的栈分别称为_____和_____。

3. 在顺序栈中,假设栈顶指针 top 是指向栈顶元素的下一个存储单元,则顺序栈判空的条件是_____;栈顶元素的访问形式是_____。

4. 在不带表头结点的链栈中,若栈顶指针 top 直接指向栈顶元素,则将一个新结点 p 入栈时修改链的两个对应语句为_____; _____。

5. 在不带表头结点的链栈中,若栈顶指针 top 直接指向栈顶元素,则栈顶元素出栈时的修改链的对应语句为_____。

6. 队列也是一种操作受限的线性表,它与栈不同的是,队列中所有的插入操作均限制在表的一端进行,而所有的删除操作都限制在表的另一端进行,允许插入的一端称为_____,允许删除的一端称为_____ 。队列具有_____的特点。

7. 由于队列的删除和插入操作分别在队首和队尾进行,因此,在链式存储结构描述中分别需要设置两个指针分别指向_____和_____,这两个指针又分别称为_____和_____。

8. 循环顺序队列是将顺序队列的存储区域看成是一个首尾相连的环,首尾相连的状态是通过数学上的_____运算来实现的。

9. 在循环顺序队列中,若规定当 front == rear 时,循环队列为空;当 front == (rear+1)%maxSize 时,循环队列为满,则入队操作时的队尾指针变化的相应语句是_____;出队操作时的队首指针变化的相应语句是_____。

10. 无论是顺序栈还是顺序队列,插入元素时必须先进行_____判断,删除元素时必须先进行_____判断;而链栈或链队列中,插入元素无须进行栈或队列是否为满的判断,只要在删除元素时先进行_____判断。

三、算法设计题

1. 编写一个函数,要求借助一个栈把一个数组中的数据元素逆置。

2. 编写一个函数判断一个字符序列是否为回文序列,所谓回文序列就是正读与反读都相同的字符序列,例如,abba 和 abdba 均是回文序列。要求只使用栈来实现。

3. 假设以一个数组实现两个栈:一个栈以数组的第一个存储单元作为栈底,另一个栈以数组的最后一个存储单元作为栈底,这种栈称为顺序双向栈。试编写一个顺序双向栈类 DuSqStack,类中要求编写 3 个方法。一个是构造方法 DuDuSqStack(int maxSize),此方法实现构造一个容量为 maxSize 的顺序双向空栈;一个是实现入栈操作的方法 push(Object X,int i),此方法完成将数据元素 X 压入到第 i(i=0 或 1)号栈中的操作;一个是实现出栈操作的方法 pop(int i),此方法完成将第 i 号栈的栈顶元素出栈的操作。

4. 循环顺序队列类采用设置一个计数器的方法来区分循环队列的判空和判满。试分别编写顺序循环队列中入队和出队操作的函数。

5. 假设采用带头结点的循环链表来表示队列,并且只设一个指向队尾元素的指针(不设队首指针),试编写相应的队列置空、队列判空、入队和出队操作的函数。

四、上机实践题

1. 设计一个测试类,使其实际运行来测试顺序栈类中各成员函数的正确性。

2. 设计一个测试类,使其实际运行来测试链队列类中各成员函数的正确性。

3. 设计一个循环顺序队列类。要求:

(1) 循环顺序队列类采用设置标志位的方法来区分循环队列的判空和判满。

(2) 循环顺序队列类除构造函数外,成员函数还应包括入队、出队和判队列是否为空的函数。

(3) 设计一个测试程序进行测试,并给出测试结果。

4. 设计一个数制转换类。要求:

(1) 编写一个将十进制数转换成二进制数的方法。

(2) 设计一个测试程序进行测试,并给出测试结果。