第3章 栈与队列

线性表是线性结构，栈与队列也是线性结构，它们有着密切的关系。栈和队列可以看成两种特殊的线性表，在进行插入和删除操作时都是在表的一端或者两端进行。栈和队列的应用非常广泛，例如，操作系统实现对各种进程的管理。

本章学习目标：

(1) 掌握栈和队列这两种数据结构的特点。

(2) 熟悉栈(队列)和线性表的关系。

(3) 重点掌握顺序栈和链栈的基本操作实现。

(4) 重点掌握循环队列和链队列的基本操作实现。

(5) 熟悉栈和队列的下溢和上溢的概念以及消除假溢的方法。

(6) 能够应用栈和队列解决实际问题。

3.1 栈

3.1.1 栈的基本概念

栈是一种特殊的线性表，其特殊性在于栈的数据元素的插入和删除只能在表尾进行，而线性表的数据元素插入和删除可以在表的任意位置进行。允许进行插入、删除操作的一端称为栈顶，另一端称为栈底(如图 3-1(a)所示)。

日常生活中有不少类似于栈的例子。假设有一个很窄的死胡同，其宽度只能容纳一辆车，现有 5 辆车，分别编号为①～⑤，这 5 辆车按编号顺序依次进入此胡同，如图 3-1(b)所示，若要退出④，必须先退出⑤；若要退出③，必须将⑤、④依次都退出才行。这个死胡同就是一个栈。越是先入栈的数据元素越是后出栈，这就是“先进后出”或“后进先出”原则。现实中有很多栈的例子，如叠放的碗碟。

定义栈的抽象数据类型基本操作如下。

(1) StackEmpty()：判断一个栈是否为空，返回值为布尔型。若栈为空，则返回 true，否则返回 false。

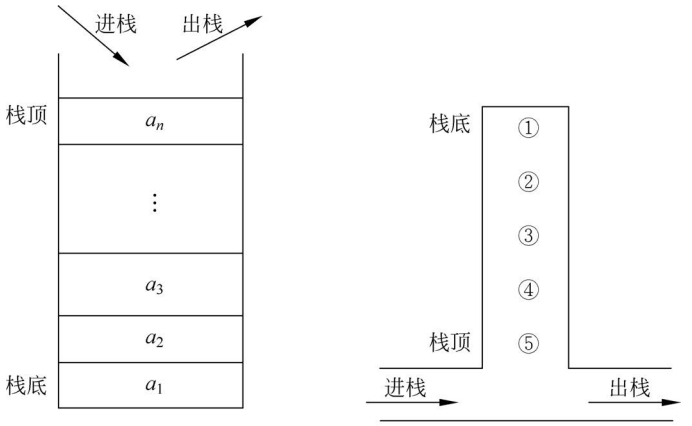
(2) clear()：栈的置空操作，即清除栈中全部元素。

(3) getSize()：求栈的数据元素个数，返回栈的数据元素个数。

(4) Top()：返回栈顶元素。若操作成功，则返回栈顶元素，否则抛出异常。

(5) Push()：在栈顶插入元素 e(入栈)，无返回值。

(6) Pop()：从栈中删除栈顶元素(出栈)，无返回值。



(a) 栈示例　　(b) 胡同及车示例

图 3-1　栈及其操作示意图

(7) display()：输出栈中各个数据元素。输出顺序是从栈顶到栈底。

使用 Java 接口描述上述基本操作如下。

```
public interface Stack {
    public boolean StackEmpty();                          //判断一个栈是否为空
    public void Clear();                                  //清除栈中全部元素
    public int getSize();                                 //求栈中数据元素个数
    public Object Top();                                  //返回栈顶元素
    public void Push(Object e) throws Exception;          //入栈
    public void Pop()throws Exception;                    //出栈
    public void display();                                //输出栈
}
```

3.1.2　栈的顺序存储结构

栈的顺序存储与顺序表类似，按顺序依次存储，可以通过数组来实现，假设数组名称为 stackElement，给数组分配 space 个存储空间。设置一个整型变量 top 用来表示栈顶元素的位置。当 top=0 时，表示栈是一个空栈；当 top=space 时，表示栈满，即所有存储空间中都有数据元素；当 top 为 1 到 space 中的任意值时，表示此时栈既不是空栈也不是栈满状态，如图 3-2 所示。

栈只有一个出入口，出栈与入栈都在栈顶进行，top 值随着入栈、出栈操作而动态变化，一直指向栈顶元素位置。再结合图 3-2 就可以找到顺序栈的栈满、栈空、清空、求栈中数据元素个数、求栈顶元素的实现条件或方法，如下。

(1) 判断一个栈是否为空的条件是 top==0。

(2) 判断一个栈是否为满的条件是 top==space。

(3) 栈的清空，使 top=0。

(4) 栈的元素个数即为 top 的值。

(5) 栈顶元素，即 top 对应的那个数据元素。

以上操作具体的实现请查看顺序栈类 SqStack 对应的各个方法。顺序栈类 SqStack 的描述如下。

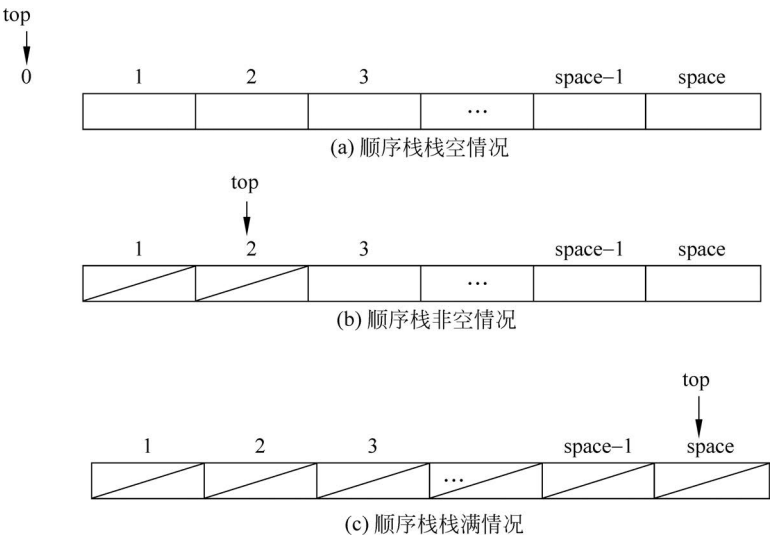


图 3-2　顺序栈存储情况

```
public class SqStack implements Stack {
    private Object[] stackElement;
    private int space;
    private int top;
    public SqStack(int space) {            //构造一个空栈
        this.space=space;
        top=0;
        stackElement =new Object[space];
    }
    public boolean StackEmpty() {          //判断一个栈是否为空
        return top==0;
    }
    public boolean StackFull () {          //判断一个栈是否为满
         return top==space;
     }
    public void Clear() {                  //清除栈中全部元素
         top=0;
     }
    public int getSize() {                 //返回栈的数据元素个数
         return top;
     }
    public Object Top() {                  //返回栈顶元素
         if (!StackEmpty())                //栈非空
             return stackElement[top − 1];      //栈顶元素
         else                              //栈为空
             return null;
}
    public void Push(Object e) throws Exception {      //入栈,入栈不成功,提示栈满; 成功,返回 1
        ...}
     //从栈中删除栈顶元素(出栈)。若操作成功,则返回 1; 否则返回 0
    public void Pop() throws Exception {
        ...}

     public void display() {
          ...}
}
```

顺序栈中的构造空栈、入栈、出栈、输出操作的实现过程分析与程序见算法 3.1～算法 3.4。

【算法 3.1】 构造一个空栈。

使用带一个参数的构造函数构造一个空栈，为该栈分配 space 个存储空间，给 top 赋初始值，值为 0。

```
public SqStack(int space) {                    //构造一个空栈
      top=0;                                   //top 赋初始值
      stackElement =new Object[space];         //为栈分配存储单元
}
```

【算法 3.2】 顺序栈的入栈操作算法。

```
public void Push(Object e) throws Exception {    //入栈
        if (top==stackElement.length)             //栈满
            {throw new Exception("栈满");         //提示栈满
            }
        else                                      //栈不满
            {stackElement[top++]=e;               //e 入栈
            }
}
```

【算法 3.3】 顺序栈的出栈操作算法。

```
public void Pop() throws Exception {//从栈中删除栈顶元素(出栈)
    if (top ==0)                    //栈为空
        throw new Exception("栈空");
    else {                          //栈非空
        --top;                      //top 后移
        System.out.println("出栈操作成功! ");
    }
}
```

【算法 3.4】 顺序栈的输出。

```
public void display() {
         for (int i =top - 1; i >=0; i--)
                System.out.print(stackElement[i].toString() + " ");      //输出
}
```

【例 3.1】 顺序栈的测试，测试数据：1，4，5，8，10。

```
import java.util.Scanner;
public class Example3_1 {
public static void main(String[] args) throws Exception{
        SqStack sq=new SqStack(5);
        Scanner sc=new Scanner(System.in);
        System.out.println("请依次输入栈中的元素(从栈底到栈顶): ");
        for(int i=0;i<5;i++) {
            String s=sc.next();
            sq.Push(s);
        }
        System.out.print("输出栈中各元素为(栈顶到栈底): ");
        sq.display();           //打印栈中元素(栈底到栈顶)

        System.out.println(" ");
        System.out.println("栈满: "+sq.StackFull());
```

```
            sq.Pop();                //从栈中删除栈顶元素(出栈)。若操作成功,则返回 1,否则返回 0
            System.out.print("重新输出栈中各元素为(栈顶到栈底): ");
            sq.display();            //重新打印栈中元素(栈底到栈顶)
            System.out.println(" ");
            System.out.println("输出栈顶元素: "+sq.Top());      //返回栈顶元素
            System.out.println("输出栈的元素个数: "+sq.getSize());
        }
    }
```

运行结果如图 3-3 所示。

```
请依次输入栈中的元素(从栈底到栈顶):
4
8
9
10
15
输出栈中各元素为(栈顶到栈底): 15 10 9 8 4
栈满: true
出栈操作成功!
重新输出栈中各元素为(栈顶到栈底): 10 9 8 4
输出栈顶元素: 10
输出栈的元素个数: 4
```

图 3-3　例 3.1 运行结果

3.1.3　栈的链式存储结构

栈的链式存储结构是先把栈的数据元素存储在内存空间,然后采用链表的形式连接起来。采用链式存储结构的栈称为链栈。但是,请注意它与普通单链表不同,栈的插入、删除操作都只能在栈顶进行,因此可以不用设置头结点,但是需要设置 top 指针,使其始终指向栈顶元素所在结点。top 指针类似于单链表中的 head 指针。当 top 指针为空时,代表指针指向空结点,即栈为空栈,如图 3-4 所示。链栈不需要判断栈满情况,如果有新加入的数据元素,给该数据元素创建结点,把这个新创建的结点连到当前栈的栈顶即可。所以链栈是无法处理栈满的情况的,是根据需要申请空间存放结点,当然是在有可用的空间的前提下。

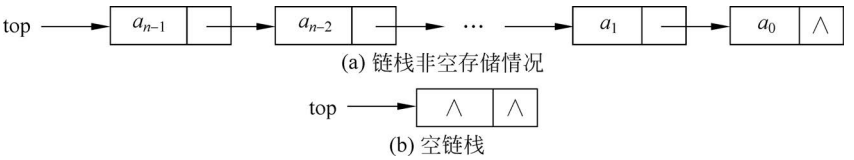


图 3-4　栈的链式存储结构

1. 链栈的结点类

链栈的结点中含有两个域,一个是数据域,另一个是指针域。与单链表类似,数据域存放该结点的数据元素,指针域存放指向下一个结点的 next 指针。链栈的结点类如下。

```
public class LinkStNode {
    public Object data;
    public LinkStNode next;
    public LinkStNode() {}
    public LinkStNode(Object data) {
        this.data=data;
    }
```

```
    public LinkStNode(Object data, LinkStNode next) {
        this.data=data;
        this.next=next;
    }
}
```

2. 链栈类

建立链栈类 LinkStack 实现栈类 Stack 接口，先定义一个私有的 top 结点，此结点类似于单链表的 head 结点。因为栈的操作只能在栈顶进行，所以定义一个 top 指针就够用了，不再需要尾指针。

```
public class LinkStack implements Stack {
    private LinkStNode top;
    private Object data;
    public boolean StackEmpty() {                    //判断一个栈是否为空
        return top==null;
    }
    public void Clear() {                            //清除栈中全部元素
        top=null;
    }
    public int getSize() {                           //求栈中数据元素个数
        LinkStNode p=top;int size=0;
        while(p!=null) {
            p=p.next;
            size++;
        }
        return size;
    }
    public Object Top() {                            //返回栈顶元素
        if(!StackEmpty())
            return top.data;
        else
            return null;
    }

    public void Push(Object e) throws Exception{     //在栈顶插入元素 e(入栈)
        LinkStNode p=new LinkStNode(e);
        p.next=top;
        top=p;

    }
    public void Pop() throws Exception {             //从栈中删除栈顶元素(出栈)
        if(StackEmpty())
            {throw new Exception("栈是空栈");
            }
            else
            {top=top.next;
            System.out.println("出栈操作成功!");
            }
    }
    public void display() {                          //输出栈
        LinkStNode p=top;
        while(p!=null) {
            System.out.print(p.data+" ");
```

```
            p=p.next;
        }
    }
}
```


3. 实现链栈的基本操作

1) 链栈的入栈

链栈的入栈操作是指将新的数据元素 e 所在结点连接到链栈中。由于栈操作只能在栈顶进行,所以把 p 结点连接到当前的 top 指针所指结点即可。具体操作可以分为以下三步。

(1) 给新数据元素建立新结点 p,LinkStNode p=new LinkStNode(e)。

(2) 新数据元素结点 p 的 next 指针指向当前 top 所指结点,即 p.next=top。

(3) 移动 top 指针,使其指向 p 结点。

【算法 3.5】 链栈的入栈。

```
public void Push(Object e) throws Exception{      //在栈顶插入元素 e(入栈)
        LinkStNode p=new LinkStNode(e);
        p.next=top;
        top=p;
    }
```

2) 链栈的出栈

链栈的出栈是指将栈顶结点从栈中移出。若栈是空栈则抛出异常,显示“栈是空栈”提示信息;若栈不是空栈,则将栈顶结点移出,top 指针移到下一个结点位,显示“出栈操作成功”。

【算法 3.6】 链栈的出栈。

```
public void Pop() throws Exception {      //从栈中删除栈顶元素(出栈)
        if(StackEmpty())                  //栈空
            {throw new Exception("栈是空栈");
            }
        else                              //栈满
            {top=top.next;                //top 指针后移
             System.out.println("出栈操作成功! ");
            }
}
```

3) 求链栈的元素个数

栈只能在栈顶进行操作,所以求链栈的元素个数需要从栈顶开始移动指针挨个点数直到栈底。具体步骤如下。

(1) 定义一个 p 结点使其指向栈顶,即 LinkStNode p=top。

(2) 定义整型变量 size,并赋初值为 0,用来计数。

(3) size 加 1,p 指针后移。

(4) 循环操作(3),直到 p 移动到指向栈底下一位置,即指向空。

【算法 3.7】 求链栈的元素个数。

```
public int getSize() {      //求栈中数据元素个数
        LinkStNode p=top;int size=0;
        while(p!=null) {
                size++;
```

```
            p=p.next;
        }
        return size;
}
```

4）链栈的输出

链栈的输出操作步骤如下。

（1）定义一个 p 结点使其指向栈顶，即 LinkStNode p=top。

（2）输出 p 结点的数据元素，然后移动 p 指针。

（3）只要 p 没有到达栈底的下一个位置，即 p!=null，一直循环步骤（2）。

【算法 3.8】 链栈的输出。

```
public void display() {      //输出栈
        LinkStNode p=top;
        while(p!=null) {
            System.out.print(p.data+" ");
            p=p.next;
        }
}
```

【例 3.2】 链栈的测试。

【程序实现】

```
public class Example3_2 {
    public static void main(String[] args) throws Exception{
        LinkStack lin=new LinkStack();
        lin.Push("a");
        lin.Push("b");
        lin.Push("c");
        lin.Push("d");
        System.out.print("输出栈中各元素为(从栈顶到栈底): ");
        lin.display();                  //打印栈中元素(栈底到栈顶)
        System.out.println(" ");
        System.out.println("栈的元素个数: "+lin.getSize());
        System.out.println("输出栈顶元素: "+lin.Top());       //返回栈顶元素
        lin.Pop();                      //从栈中删除栈顶元素(出栈)
        System.out.println("再次输出栈顶元素: "+lin.Top()); //返回栈顶元素
        System.out.print("重新输出栈中各元素为(从栈顶到栈底): ");
        lin.display();                  //重新打印栈中元素(栈底到栈顶)
        System.out.println(" ");
    }
}
```

运行结果如图 3-5 所示。

```
输出栈中各元素为(从栈顶到栈底): d c b a
栈的元素个数: 4
输出栈顶元素: d
出栈操作成功!
再次输出栈顶元素: c
重新输出栈中各元素为(从栈顶到栈底): c b a
```

图 3-5　例 3.2 运行结果

3.2 栈的应用

3.2.1 数制转换

数制转换可以通过辗转相除得到，例如，十进制数 2021 转换成相应的八进制数，用 2021 对 8 进行不断地求余运算，余数依次为 5，4，7，3，然后将其反序输出，就可以得到 2021 的八进制数是 3745。

【例 3.3】 将十进制数转换成相应的八进制数。

【程序实现】

```
public class Example3_3 {
    public static void main(String[] args) throws Exception {
        LinkStack lin=new LinkStack();
        Scanner sc=new Scanner(System.in);
        System.out.print("请输入十进制数 x=");
        System.out.println(" ");
        int x=sc.nextInt();
        while(x>0) {
            lin.Push(x%8);
            x=x/8;
        }
        System.out.print("输出八进制数: ");
        lin.display();
    }
}
```

运行结果如图 3-6 所示。

```
请输入十进制数x=
2021
输出八进制数: 3 7 4 5
```

图 3-6 例 3.3 运行结果

3.2.2 栈在括号匹配问题中的应用

括号包括花括号“{”和“}”、方括号“[”和“]”、圆括号“(”和“)”。括号匹配是指括号要成对出现且可以任意相互嵌套。成对出现就是指有左括号就必须有相应的右括号，有右括号也必须要有相应的左括号。下面是一些正确使用括号的例子。

```
i=[c-(a+b)]*d;
k=f-{2+[(a+b)*c-d]-5};
while(p!=null){p=p.next;q=q+1;}.
```

下面是一些不正确使用括号的例子：

```
s=c-(a+b)*d)-5;                          //右括号多余
if(stackempty(p=q())                     //左括号多余
while(p!=2*(a-b]){p=p.next;q=q+1;}       //左右括号不匹配
```

【例 3.4】 给出一段 Java 语句，判断此语句中的括号是否匹配。

【程序实现】

```
public class Example3_4 {                                    //符号匹配
    //判断左分隔符 str1 和右分隔符 str2 是否匹配
    public boolean matches(char str1, char str2) {
            if((str1=='('&&str2==')')||(str1=='['&&str2==']')||(str1=='{'&&str2
==')')//匹配
             return true;
            else//不匹配
                return false;
    }

    public boolean isRight(String str) throws Exception {
        SqStack Sq=new SqStack(100);                         //新建最大存储空间为 100 的顺序栈
            int length =str.length();                        //字符串长度
            for(int i =0; i<length;i++) {
                char c =str.charAt(i);                       //指定索引处的 char 值
                if (c=='('||c=='['||c=='{')                  //c 是左括号
                    Sq.Push(c);                              //压入栈
                else if(c==')'||c==']'||c=='}')              //c 是右括号
                    {if (Sq.StackEmpty())
                        { System.out.println("缺少左括号(");return false;}
                    else if (matches((char)Sq.Top(), c))
                        { Sq.Pop();}
                    else
                        {System.out.println("左右括号不匹配");return false; }
                     }
                else                                         //c 是其他字符
                    continue;
            }
            if (Sq.StackEmpty())                             //栈中存在没有匹配的字符
                return true;
            else
                {System.out.println("缺少右括号)");return false; }
            }

            public static void main(String[] args) throws Exception {
                Example3_4 e =new Example3_4();
                System.out.println("请输入 Java 语句:");
                Scanner sc =new Scanner(System.in);
                String ss=sc.nextLine(); //输入的 Java 语句需要括号是英文状态的括号才能匹配
                System.out.println(e.isRight(ss));       //输出
            }
}
```

运行结果如图 3-7 所示。

```
请输入Java语句:
3+(5-2
缺少右括号)
false
```

图 3-7　例 3.4 运行结果

3.2.3 汉诺塔问题

汉诺塔问题是源于印度一个古老传说的益智游戏。传说中大梵天创造世界的时候做了三根金刚石柱子，在第一根柱子上从下往上按照大小顺序摞着 64 片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。

如图 3-8 所示，从左到右有 A、B、C 三根柱子，其中，A 柱子上面有从小叠到大的 n 个圆盘，现要求将 A 柱子上的圆盘移到 C 柱子上去，其间只有一个原则：一次只能移动一个盘子且大盘子不能在小盘子上面，求移动的步骤和移动的次数。其中，圆盘子自上而下编号为 $1,2,\cdots,n$。

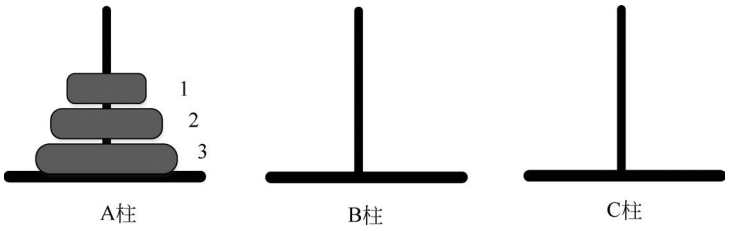


图 3-8 汉诺塔

当 $n=1$ 时，只需要移动 1 次，即直接从 A 柱移动到 C 柱。

当 $n=2$ 时，上面只有 1 号和 2 号两个圆盘，需要移动 3 次。先把 1 号盘从 A 柱移动到 B 柱，再把 2 号盘从 A 柱移动到 C 柱，最后把 1 号盘从 B 柱移动到 C 柱。

当 $n=3$ 时，需要移动 7 次，移动的具体情况如图 3-9 所示。

```
(1) n == 1
        第1次 1号盘 A----->C     sum = 1 次
(2) n == 2
        第1次 1号盘 A----->B
        第2次 2号盘 A----->C
        第3次 1号盘 B----->C     sum = 3 次
(3) n == 3
        第1次 1号盘 A----->C
        第2次 2号盘 A----->B
        第3次 1号盘 C----->B
        第4次 3号盘 A----->C
        第5次 1号盘 B----->A
        第6次 2号盘 B----->C
        第7次 1号盘 A----->C     sum = 7 次
```

图 3-9 $n=3$ 时汉诺塔问题移动次数

【例 3.5】 汉诺塔问题-递归算法。

【程序实现】

```
public class Example3_5 {
    private int s =0;                              //全局变量,对搬动计数
      //将塔座 a 上圆盘按规则移到塔座 c 上,b 作为辅助塔座
```

```
    public void hanoiRecursion(int n, char a, char b, char c) {
        if (n ==1) {
            move(a, 1, c);                    //将最上面(编号为 1)的圆盘从 a 移到 c
else {
            hanoiRecursion(n - 1, a, c, b); //将 a 上编号为 1～n－1 的圆盘移到 b,c 作辅助塔
            move(a, n, c);                    //将编号为 n 的圆盘从 a 移到 c
            hanoiRecursion(n - 1, b, a, c); //将 b 上编号为 1～n－1 的圆盘移到 c,a 作辅助塔
            }
        }

    //移动操作,将编号为 n 的圆盘从 a 移到 c
    public void move(char a, int n, char c) {
        System.out.println("第"+ ++s +"次移动:把 "+n+"号圆盘从"+a+"柱－>移动到
"+c+"柱");
        }

    public static void main(String[] args) {
        Example3_5 x =new Example3_5();
        System.out.println("－－－－－－－－－汉诺塔问题－－－－－－－－－");
        x.hanoiRecursion(3, 'a', 'b', 'c');   //对于圆盘数量为 3 时进行移动
    }
}
```

运行结果如图 3-10 所示。

```
--------汉诺塔问题--------
第1次移动: 把 1号圆盘从a柱->移动到c柱
第2次移动: 把 2号圆盘从a柱->移动到b柱
第3次移动: 把 1号圆盘从c柱->移动到b柱
第4次移动: 把 3号圆盘从a柱->移动到c柱
第5次移动: 把 1号圆盘从b柱->移动到a柱
第6次移动: 把 2号圆盘从b柱->移动到c柱
第7次移动: 把 1号圆盘从a柱->移动到c柱
```

图 3-10　例 3.5 运行结果

3.3　队　　列

3.3.1　队列的基本概念

队列是一种特殊的线性表,与栈类似,队列的数据元素的插入、删除操作也有限制,限定在表的一端只能插入,另一端只能删除。

队列的基本操作如下。

(1) queueEmpty(): 判断一个队列是否为空。返回值是布尔型,若为空,则返回 1; 否则返回 0。

(2) clear(): 清除队列中全部元素。

(3) getSize(): 求队列中数据元素个数。

(4) frontElem(): 取队首元素。

(5) inQueue(Object e): 入队,返回值是布尔型,若入队成功,则返回 1; 否则返回 0。

(6) outQueue() throws Exception: 出队,返回值是出队元素。

(7) display(): 输出。

队列基本操作的接口描述如下。

```
public interface Queue {
    public boolean queueEmpty(); //判断一个队列是否为空。若为空,则返回 1,否则返回 0
    public void clear();                                    //清除队列中全部元素
    public int getSize();                                   //求队列中数据元素个数
    public Object frontElem();                              //取队首元素
    public void inQueue(Object e) throws Exception;         //入队
    public Object outQueue() ;                              //出队
    public void display();                                  //输出
}
```

3.3.2 队列的顺序存储

1. 顺序队列

顺序队列就是采用顺序存储方式存储队列数据元素的队列。由于队列的操作在队尾与队首进行,设置两个变量指向队首和队尾位置。front 变量指向队首数据元素所在位置,rear 变量指向队尾数据元素所在位置的下一位置。顺序存储采用数组形式存储,数组下标从 0 开始计数,队列中数据元素个数可以用 rear－front 表示。入队时,front 代表队首位置,所以值不变; rear 代表队尾的下一位置,每入队一个数据元素,值增加 1。出队时,rear 值不变; 每出队一个数据元素,front 值增加 1。

图 3-11 展示了一队列采用顺序存储方式,从空队列开始,数据元素依次入队、出队、入队的情况。具体过程分析如下。

(1) 初始化队列,令 front＝rear＝0。此时,队列为空队列,如图 3-11(a)所示。

(2) 让数据元素 a、b、c、d 依次入队。有 4 个数据元素入队,rear 值增加 4,此时 rear＝4,front 值不变,如图 3-11(b)所示。

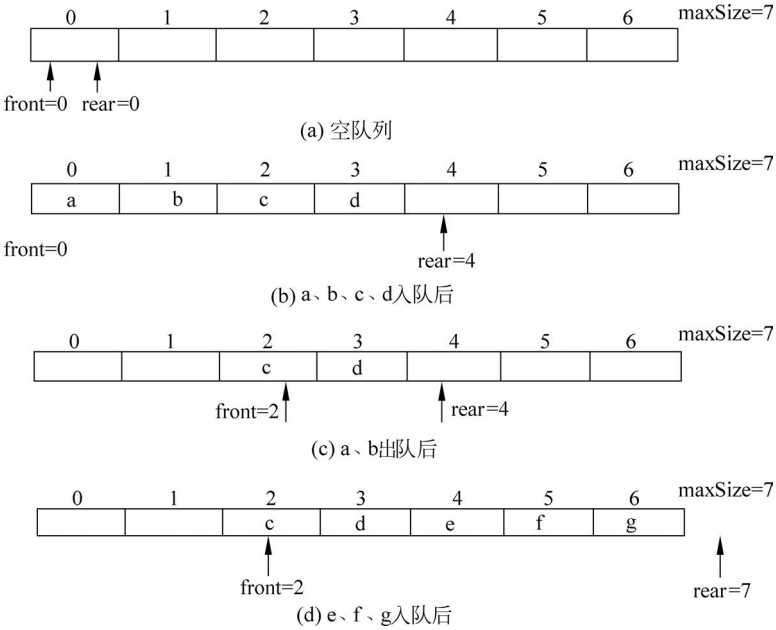


图 3-11　顺序队列

(3) 再让数据元素 a、b 依次出队。此时，front 值增加 2，front＝2，rear 值不变，rear＝4，如图 3-11(c)所示。

(4) 再让数据元素 e、f、g 入队。又有 3 个数据元素入队，rear 值增加 3，此时 rear＝7，front 值不变，front＝2，如图 3-11(d)所示。

如果还有数据元素 h 没有入队，需要入队，h 应该存放于 rear＝7 的位置，但是这时已经越界，超出了顺序存储的最大存储空间 maxSize，引起了“溢出”现象，使人误以为存储空间已满，实际上在顺序队列的前面还有两个空间可以存储数据元素。因此，把这种溢出现象称为假溢出。要解决这种假溢出现象，就需要把顺序队列的存储空间看成一个能够首尾相连的循环队列。当 rear 值到达表尾后，再增加 1 时，自动跳到 0 位置，如图 3-11(d)所示，e、f、g 入队后，rear 不再指向 7 位置，而是指向 0 位置，即 rear＝0。

2. 循环顺序队列

循环顺序队列是将顺序队列看成一个逻辑上首尾相连的队列。这里以实际例子来具体说明循环顺序队列的不同状态。循环顺序队列的最大可存储空间为 maxSize，假设 maxSize＝7，那么队列的变化情况及其不同状态展示如图 3-12 所示。

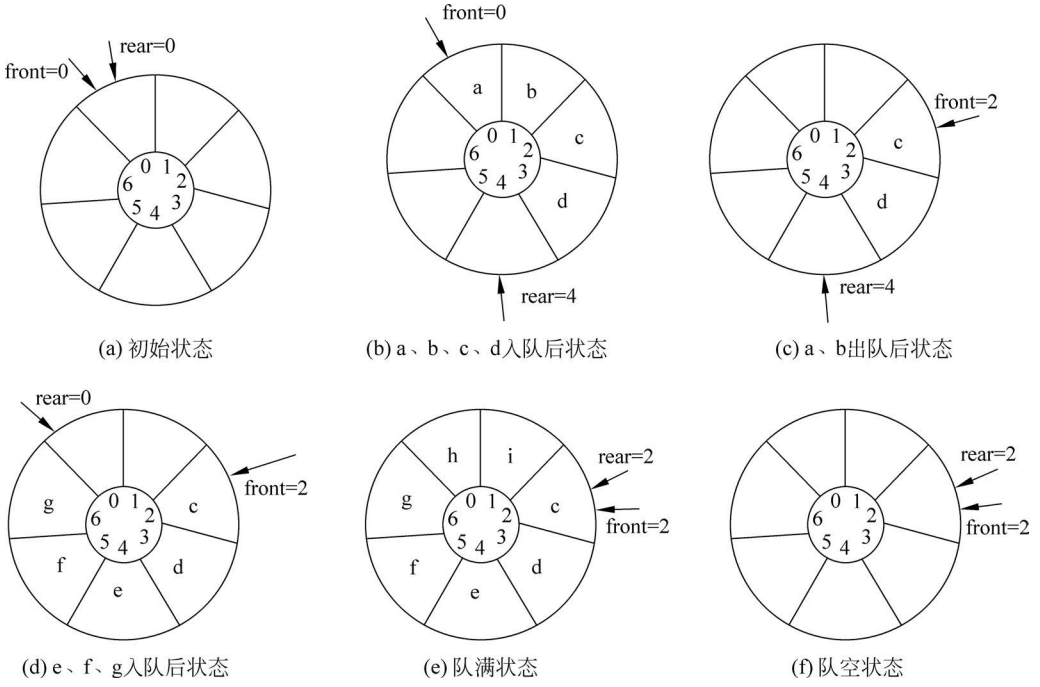


图 3-12　循环队列变化及其不同状态

(1) 队列的初始状态为空队列，如图 3-12(a)所示。

(2) 数据元素 a、b、c、d 依次入队，分别存储在 0、1、2、3 位置，此时 front＝0，rear 值随着入队的数据元素个数改变，每次加 1，变为 4，如图 3-12(b)所示。

(3) 数据元素 a、b 依次出队，0、1 位置空置，没有数据元素，front 值随着改变，变为 2，如图 3-12(c)所示。

(4) 数据元素 e、f 入队，当 rear 值随着改变，从原来的 4 每次加 1，变为 5，再变到 6。数据元素 g 入队，rear 值从 6 位置跳到下一位置，是 0 位置，所以此时 rear＝0，front 值不变，

如图 3-12(d)所示。

(5) 数据元素 h、i 入队后，此时 rear＝2，front＝2，如图 3-12(e)所示。

(6) 所有数据元素依次出队，rear 值不变，rear＝2，front 值随着数据元素出队发生变化，由原来的 2 变为 3、4、5、6，再循环回到 0、1、2，即 front＝2，此时 front＝rear＝2，如图 3-12(f)所示。

图 3-12(e)为队满状态、图 3-12(f)为队空状态，当队列是队满状态时 front＝rear＝2，当队列是队空状态时，front 与 rear 值还是 2，这表示不能仅根据 front 与 rear 值是否相等来判断队列是队满状态还是队空状态。如何判断队列的队满与队空状态，有几种解决方法，本书采用增加一个标识变量的方式来解决。设标识变量 flag 的初始值为 0，每一次入队成功，flag 值置为 1；每一次出队成功，flag 值置为 0。这时判断队列队满状态的条件是 front＝rear && flag＝1；判断队列队空状态的条件是 front＝rear && flag＝0。

下面分析循环队列的基本操作算法。

1) 循环顺序队列入队操作

入队操作是将新的数据元素插到队尾，使其变为新的队尾，分为两步实现。当队列是队满状态时，输出异常。当队列不是队满状态时，将新数据元素值 e 插到队尾 rear 位置，然后 rear 值后移一位，若 rear 值已经指向存储空间的末尾位置，则循环移动到存储空间的 0 位，即 rear＝(rear ＋ 1) % queueSpace.length。

【算法 3.9】 循环顺序队列入队操作。

```
public void inQueue(Object e) throws Exception {
        if (queueFull())                                  //队列满
        throw new Exception("队列已满");                   //输出异常
        else {                                            //队列未满
            queueSpace[rear] =e;                          //e 赋给队尾元素
            rear =(rear + 1) % queueSpace.length;         //修改队尾指针
            flag=1;
        }
}
```

2) 循环顺序队列出队操作

循环顺序队列出队操作是将队首元素从队列中移出，此元素不再是队列中的数据元素，将 front 下移一位，使用 front＝(front ＋ 1) % queueSpace.length 语句，循环顺序队列出队操作如算法 3.10 所示。

【算法 3.10】 循环顺序队列出队操作。

```
public Object outQueue() {
        if (queueEmpty())                                 //队列为空
            return null;
        else {
            Object x =queueSpace[front];                  //取出队首元素
            front =(front + 1) % queueSpace.length;       //更改队首的位置
            flag=0;
            return x;                                     //返回队首元素
        }
}
```

3) 循环顺序队列输出数据元素

【算法 3.11】 循环顺序队列输出数据元素操作。

```
public void display() {
        System.out.print("从队首到队尾队列数据元素分别为:");
        if (queueEmpty())
            System.out.print("此队列为空");
        else if(queueFull()) {//队满
            for (int i =0; i<queueSpace.length; i++)//输出队列中所有数据元素
                System.out.print(queueSpace[i].toString() + " ");
            }
        else//队列非满非空状态
            for (int i =front; i !=rear; i =(i + 1) % queueSpace.length)
                    //从队首到队尾输出数据元素
                System.out.print(queueSpace[i].toString() + " ");

        System.out.println("");
}
```

4）求循环顺序队列数据元素个数

【算法 3.12】 求循环顺序队列数据元素个数操作。

```
public int getSize() {
        if(rear-front>0)
            return rear-front ;
        else
            return (rear-front +queueSpace.length);
}
```

3. 循环顺序队列类

```
public class SqQueue implements Queue {
    private Object[] queueSpace;                    //队列存储空间
    int maxSize;
    private int front;                              //队首元素位置
    private int rear;                               //队尾元素的下一个位置
    private int flag;
    //顺序循环队列类的构造函数

    public SqQueue(int maxSize) {
        front =rear =0;                             //队首、队尾初始化为 0
        queueSpace =new Object[maxSize];            //为队列分配 maxSize 个存储单元
        flag=0;
    }

    //将一个已经存在的队列置成空
    public void clear() {
        front =rear =0;
        flag=0;
    }

    //测试队列是否为空
    public boolean queueEmpty() {
        return front ==rear&&flag==0;
    }
    //测试队列是否为满
    public boolean queueFull() {
        return front==rear&&flag==1;
```

```
    }

    public int getSize() {                                  //求队列中的数据元素个数
        ...
    }
    public void inQueue(Object e) throws Exception {        //入队
        ...
    }
    //返回队首元素,如果此队列为空,则返回 null
    public Object frontElem() {
        if (queueEmpty())                                   //队列为空
            return null;
        else
            return queueSpace[front];                       //返回队首元素
    }

    public Object outQueue() {                              //出队操作
        ...
    }

    public void display() {                                 //输出队列
        System.out.print("从队首到队尾队列数据元素分别为:");
        if (queueEmpty())
            System.out.print("此队列为空");
        else if(queueFull()) {                              //队满
            for (int i =0; i<queueSpace.length; i++)   //输出队列中所有数据元素
                System.out.print(queueSpace[i].toString() + " ");
            }
        else                                                //队列非满非空状态
            for (int i =front; i !=rear; i =(i + 1) % queueSpace.length)
                    //从队首到队尾输出数据元素
                System.out.print(queueSpace[i].toString() + " ");

        System.out.println("");
    }
}
```

3.3.3 链队列

由于队列的插入、删除操作只允许在队首位置插入,队尾位置删除,因此在进行队列的链式存储时可以采用不带头结点的单链表来实现。在实现时一般设置两个指针,一个指针是 front 指针,指向队首元素结点;另一个是队尾 rear 指针,指向队尾元素结点,如图 3-13 所示。

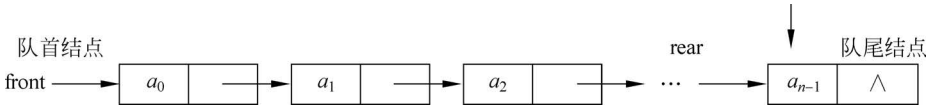


图 3-13 非循环链队列

1. 链队列的基本操作

1) 链队列入队操作

链队列入队操作算法如算法 3.13 所示,具体步骤如下。

(1) 将待入队的数据元素 e 进行初始化,生成新的结点 p,程序语句为: LinkStNode p=new LinkStNode(e)。

(2) 如果队列非空,队尾指针指向 p,然后队尾指针移动到 p 结点位置; 如果队列为空,则队首、队尾指针都指向 p 结点。

【算法 3.13】 链队列入队操作。

```
public void inQueue(Object e) {
        LinkStNode p =new LinkStNode(e);          //初始化新入队的结点
        if (front !=null) {                       //队列非空
            rear.next=p;                          //队尾指针指向 p
            rear =p;                              //队尾指针移动到 p
        } else                                    //队列为空
            front =rear =p;                       //队首队尾指针相等
}
```

2) 链队列出队操作

如果队列为空,则返回空值; 如果队列非空,则按以下步骤进行操作,具体如算法 3.14 所示。

(1) 定义 p 结点,且使 p 结点指向队首结点,即 LinkStNode p=front。

(2) 如果队首结点也是队尾结点,即 p 指向的也是队尾结点,则队尾指针为空。

(3) front 指针后移,即 front=front.next。

(4) 返回 p 的数据域,即 return p.data。

【算法 3.14】 链队列出队操作。

```
public Object outQueue() {
        if (front !=null) {                       //队列非空
            LinkStNode p =front;                  //定义 p 结点,且指向队首结点
            if (p==rear)                          //被删的结点是队尾结点
                rear=null;                        //rear 值为空
            front =front.next;                    //front 后移
            return p.data;                        //返回队首结点数据
        } else                                    //队列空
            return null;                          //返回空
}
```

3) 取队首元素操作

如果队列为空,则返回空值; 如果队列非空,输出队首元素 front.data,具体如算法 3.15 所示。

【算法 3.15】 取队首元素。

```
public Object frontElem() {
        if (front !=null)                 //队列非空
            return front.data;            //返回队首结点数据元素
        else                              //队列为空
            return null;                  //返回空
}
```

4) 求队列数据元素个数操作

求队列数据元素个数操作的算法如算法 3.16 所示,具体步骤如下。

(1) 定义 p 结点,且使 p 结点指向队首结点,即 LinkStNode p=front。

(2) 定义变量 size 且赋初值为 0。

(3) 只有 p 非空,则 p 指针后移,size 自增。

(4) 返回 size 值。

【算法 3.16】 求队列数据元素个数算法。

```
public int getSize() {
        LinkStNode p =front;                    //p 指向队首结点 front
        int size =0;                            //对变量 size 赋初值为 0
        while (p !=null) {                      //循环查找,一直查找到队尾
            p =p.next;                          //p 后移
            ++size;                             //长度增 1
        }
        return size;
}
```

2. 链队列类描述

```
public class LinkQueue implements Queue{
    private LinkStNode front;                   //队首指针
    private LinkStNode rear;                    //队尾指针
    //链队列类的无参构造函数
    public LinkQueue() {
        front =rear =null;
    }
    //将一个已经存在的队列置成空队列
    public void clear() {
        front =rear =null;
    }
    //判断一个队列是否为空
    public boolean queueEmpty() {
        return front ==null;
    }
    //求队列中的数据元素个数
    public int getSize() {
        LinkStNode p =front;                    //p 指向队首结点 front
        int size =0;                            //对变量 size 赋初值为 0
        while (p !=null) {                      //循环查找,一直查找到队尾
            p =p.next;                          //p 后移
            ++size;                             //长度增 1
        }
        return size;
    }
    //入队操作
    public void inQueue(Object e) {
        LinkStNode p =new LinkStNode(e);        //初始化新入队的结点
        if (front !=null) {                     //队列非空
            rear.next=p;                        //队尾指针指向 p
            rear =p;                            //队尾指针移动到 p
        } else                                  //队列为空
            front =rear =p;                     //队首队尾指针相等
    }
    public Object frontElem() {                 //取队首元素
        if (front !=null)                       //队列非空
            return front.data;                  //返回队列元素
```

```
        else                                    //队列为空
            return null;                        //返回空
    }
    public Object outQueue() {                  //出队操作
        if (front !=null) {                     //队列非空
            LinkStNode p =front;                //定义 p 结点,且指向队列首结点
            if (p==rear)                        //被删的结点是队尾结点
                rear=null;                      //rear 值为空
            front =front.next;                  //front 后移
            return p.data;                      //返回队首结点数据
        } else                                  //队列空
            return null;                        //返回空
    }
    //输出函数,输出所有队列中的元素(从队首到队尾)
    public void display() {
        if (!queueEmpty()) {                    //队列非空
            LinkStNode p =front;                //定义 p 结点,且指向队列首结点
            while (p !=rear.next) {             //从队首到队尾
                System.out.print(p.data.toString() + " ");
                p =p.next;                      //p 后移
            }
        } else {                                //队列空
            System.out.println("此队列为空");
        }
    }
}
```

3.4 队列的应用

3.4.1 回文判定

回文是汉语中的一种回文语法,即把相同的词汇或句子,在文中调换位置或颠倒过来,产生首尾回环的情况,也叫作回环。

判定字符串是否是回文的基本思想就是从左至右读字符串和从右至左读字符串是一样的。例如,ABCDCBA 是回文,ABCEBA 不是回文。如何判断字符串是否是回文?这里使用队列和栈的特性来设计算法。由于队列是先进先出,栈是先进后出,只需要将字符串中的字符依次存放在一个队列和一个栈内,然后出队和出栈,依次比较出队和出栈字符是否相等,若全部都相等,则字符串是回文,否则不是回文。

【例 3.6】 从键盘输入任意字符串,判断字符串是否是回文。

【程序实现】

```
import java.util.Scanner;
public class Example3_6 {
    public static boolean paCheck(String str) throws Exception{
        int n=str.length();                     //字符串 str 的长度
        SqStack Sq=new SqStack(1000);           //建立一个顺序栈
        SqQueue SQ=new SqQueue(1000);           //建立一个顺序队列
        for(int i=0;i<n;i++) {
            Sq.Push(str.charAt(i));             //字符串第 i 个字符入栈
```

```
            SQ.inQueue(str.charAt(i));          //字符串第 i 个字符入队
        }
        while (!Sq.StackEmpty()&&!SQ.queueEmpty()) {//栈 Sq 非空并且队列 SQ 非空时,依
                                                    //次判断栈顶与队首元素是否相等
            if( Sq.Top()!=SQ.frontElem()) //栈顶与队首元素不相等
            {return false;}
            Sq.Pop();                           //栈顶元素出栈
            SQ.outQueue();                      //队首元素出队
        }
        return true;
    }
    public static void main(String[] args)throws Exception {
        System.out.println("请输入字符串:");
        Scanner sc =new Scanner(System.in);
        String ss=sc.next();
        if(Example3_6.paCheck(ss))
            System.out.println(ss+"是回文");
        else
            System.out.println(ss+"不是回文");
    }
}
```

运行结果如图 3-14 所示。

图 3-14　例 3.6 运行结果

3.4.2　打印杨辉三角

杨辉三角是形如图 3-15 所示的有规律排列的数字。它实际上就是牛顿二项式的系数。除第一、第二行外,其他行数字的特点是,除第一个和最后一个数字是 1 之外,其余各数字都是上一行中位于其左、右方的两个数之和。

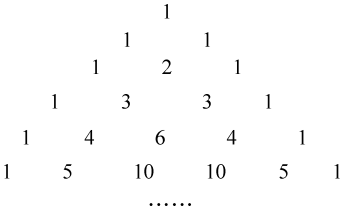


图 3-15　杨辉三角

【例 3.7】 从键盘输入杨辉三角的行数,输出杨辉三角。

【程序实现】

```
import java.util.Scanner;
public class Example3_7 {
    public static void YangHuiTriangle(int n) {
        LinkQueue q=new LinkQueue();                    //建立链队列 q
        q.inQueue(1);                                   //1 入队
        for(int rowIndex=2;rowIndex<=n+1;rowIndex++ ) {  //从对 2 行开始循环操作
         q.inQueue(1);                                  //每行的第一个数字 1,入队
```

```
            for(int j=1;j<rowIndex-1;j++) {          //用第 rowIndex-1 行计算第 rowIndex 的值
                System.out.printf("%4d",q.frontElem());    //输出队首
                int x=(int)q.outQueue();                  //队首出队并把值赋给 x
                x=x+(int)q.frontElem();                   //x 等于原队首加上现在队首的值
                q.inQueue(x);                             //x 入队
            }
            System.out.printf("%4d",q.outQueue());
            System.out.println("");                       //换行
            q.inQueue(1);                                 //每行的最后一个数字 1,入队
        }
        while (!q.queueEmpty()) {
            System.out.printf("%4d",q.outQueue());
        }
    }

    public static void main(String[] args) {
        System.out.println("请输入杨辉三角的行数:");     //输出
        Scanner sc =new Scanner(System.in);
        int row=sc.nextInt();
        YangHuiTriangle(row);
    }
}
```

习　　题

一、选择题

1. 栈是一种特殊的线性表,其特殊性体现在其插入和删除操作都限制在栈顶进行。因此,栈具有(　　)特点。

A. 先进后出　　B. 先进先出　　C. 后进后出　　D. 没有限制

2. 若将数据元素 a、b、c、d 依次进栈,则不可能得到的出栈序列是(　　)。

A. a、b、c、d　　B. d、c、b、a　　C. d、c、a、b　　D. a、d、c、b

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. 无须对栈做任何判定

4. 队列中数据元素具有(　　)特点。

A. 先进先出　　B. 先进后出　　C. 后进先出　　D. 没有限制

5. 一个队列元素的进队顺序为 1234,则其出队顺序是(　　)。

A. 1234　　B. 4321　　C. 3124　　D. 4123

6. 向一个栈顶指针为 top 的非空链栈插入一个结点 s,则插入的程序语句是(　　)。

A. top.next=s,top=s　　B. s.next=top,top=s

C. top.next=s,s.next=top　　D. top.next=s

7. 一个顺序栈的栈顶指针为 top,则它的判空条件是(　　)。

A. top=MaxSize　　B. top=MaxSize-1

C. top=-1　　D. top=0

8. 一个顺序队列(非循环)的队尾指针为 rear,队首指针为 front,则它的判空条件是(　　)。

A. front=rear=null　　B. front=null

C. rear＝null　　D. front＝rear＝－1

9. 假设一个非空链队列(非循环)的队尾指针为 rear,队首指针为 front,在队列中插入一个结点 s,则插入的程序语句是(　　)。

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

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

10. 一个循环顺序队列的队尾指针为 rear,队首指针为 front,采用增加一个标识变量的方式来解决区分循环队列的判空和判满,则它的判空条件是(　　),判满条件是(　　)。

A. front＝rear＝null　　B. front＝rear && flag＝0

C. front＝rear && flag＝1　　D. (front＋1)％ MaxSize＝＝rear

二、填空题

1. 若用大小为 6 的数组来实现循环队列,且当前 front 和 rear 的值分别为 0 和 4,当从队列中删除两个元素,再加一个元素后,front 和 rear 的值分别为_____和_____。

2. 引入循环队列的目的是防止_____现象。

3. 顺序队列在插入数据元素时必须先进行_____判断,删除元素时必须先进行_____判断;而在链队列的插入操作时无须进行队列是否为满的判断,只要在删除元素时先进行_____判断。

4. 循环顺序队列的 rear 指针后移一位对应的 Java 语句是_____。

5. 链队列的数据元素 e 入队,生成新的结点 p 的 Java 程序语句是_____。

6. 链队列进行判空的 Java 程序语句是_____。

三、判断题

1. 队列中数据元素具有后进后出的特点。(　　)

2. 栈共有三个数据元素,分别是 a、b、c,假设进栈顺序是 abc,则最先出栈的数据元素一定是 a。(　　)

3. 队列是一种特殊的线性表。(　　)

4. 循环队列在插入数据元素时必须先进行判空操作。(　　)

5. 顺序队列在删除数据元素时必须先进行判空操作。(　　)

6. 数据元素 b,c,d 依次入队,则出队顺序也是 b,c,d。(　　)

7. 栈只有一个出入口,即栈的出口与入口相同。(　　)

8. 栈和队列都可以采用链式存储方式存储数据元素。(　　)

四、算法设计题

1. 设计一个算法,使用一个整数栈把数组中的所有数据元素逆置过来,并进行测试。

2. 设计一个程序,利用一个栈模拟汽车停车场,停车场只有一个出口,未入场汽车在外排队等候。

3. 假设采用少用一个元素空间的方法解决循环顺序队列的判满和判空问题,试着编写入队操作和出队操作函数。

4. 约瑟夫问题:设有 n 个人站成一圈,其编号为 1～n,编号由入圈的顺序决定,第一个入圈的人编号为 1,最后一个为 n,从第 1 个人开始报数,数到 m ($1 \leqslant m \leqslant N$)的人将出圈,然后下一个人继续从 1 开始报数,直至所有人全部出圈,求依次出圈的编号。请采用循环队列的方法解决约瑟夫问题。