第5章　栈 和 队 列

栈和队列都属于线性表，但由于对它们操作的简易性和特殊性，并且在计算机科学和日常生活中非常有用，所以专门用一章内容来讨论。

5.1　栈的定义和运算

1．栈的定义

栈（stack）又称堆栈，它是一种运算受限的线性表，其限制是仅允许在线性表的固定一端（表尾）进行插入、删除和读取元素等运算，不允许在其他任何位置进行运算，比一般线性表运算更加简单和方便。人们把对栈进行运算的一端称为**栈顶**（top），栈顶位置的元素被称为**栈顶元素**，相对地，把另一端称为**栈底**（bottom）。向一个栈插入新元素又称为**进栈**、**压栈**或**入栈**（push），它是把新元素放到当前栈顶元素的上面位置，使之成为新的栈顶元素；从一个栈删除元素又称为**出栈**或**退栈**（pop），它是把当前栈顶元素删除掉，使其下面位置的元素成为新的栈顶元素。

在日常生活中有许多类似栈的例子，例如，刷洗盘子时，依次把每个洗净的盘子放到洗好的一摞盘子之上，相当于进栈操作；取用盘子时，从一摞盘子上一个接一个地往下拿，相当于出栈操作。又如，包装商品时，一层一层地从里向外包装材料，相当于进栈操作；打开包装时，又一层一层地从外向里去掉包装，相当于出栈操作。

由于栈的插入和删除运算仅在栈顶一端进行，后进栈的元素必定先出栈，后被摞上的盘子必定先被取用，所以又把栈称为**后进先出表**（Last In First Out，LIFO 表）。

例如，假定一个栈 S 被抽象为(a, b, c)，其中，表尾的一端为栈顶，字符 c 为栈顶元素。若向 S 压入一个元素 d，则 S 变为(a, b, c, d)，此时字符 d 为栈顶元素；若接着从栈 S 中依次删除两个元素，则首先删除的是元素 d，接着删除的是元素 c，栈 S 变为(a, b)，此时栈顶元素为 b。

由于栈操作只限定在一端进行，对它做插入（进栈）和删除（出栈）等运算时不需要比较和移动任何元素，所以其时间复杂度均为 O(1)。

2．栈的抽象数据类型

栈的抽象数据类型也同样包含数据和操作（运算）两个部分。数据部分为采用任何存储方法存储的一个数据栈；操作部分应包括元素进栈、元素出栈、读取栈顶元素、检查栈是否为空等，它比对线性表的操作要少得多。下面给出栈的抽象数据类型的具体定义。

```
ADT Stack is
```

```
    Data:
        采用任何存储方法存储的一个数据栈
    Operation:
        initStack();          //创建一个栈并初始化为空
        push(obj);            //向栈顶插入一个新元素 obj
        pop();                //从栈中删除栈顶元素并返回
        peek();               //返回栈顶元素的值
        isEmpty();            //判断栈是否为空栈
        clear();              //清除栈中的所有元素使之为一个空栈
end Stack
```

栈的抽象数据类型也需要采用 Java 语言中的接口来描述，假定采用的接口名仍为 Stack。在栈接口 Stack 中，只需要包含抽象数据类型中的操作声明部分（创建和初始化栈的操作除外，因为它与数据存储结构有关），不需要包含其数据部分，栈的数据部分和对其进行初始化的操作，以及对其他所有操作的具体定义，都应包含在实现栈接口 Stack 的相应子类中。

栈接口 Stack 的具体定义如下：

```
public interface Stack
{      //栈的抽象接口的定义
    void push(Object obj);   //向栈顶插入一个新元素 obj
    Object pop();            //从栈中删除栈顶元素并返回
    Object peek();           //返回栈顶元素的值
    boolean isEmpty();       //判断栈是否为空，若是则返回 true，否则返回 false
    void clear();            //清除栈中的所有元素使之成为一个空栈
}
```

3．栈的运算举例

假定一个整型数组 a={20,16,38,42,29}，整型变量 x=80，待操作的当前栈 stack 为空，则对 stack 进行的一组运算如下：

```
for(int i=0; i<a.length; i++) stack.push(a[i]);
    //向栈 stack 依次插入数组 a 中每个元素，栈的内容为(20,16,38,42,29)
stack.push(18);                        //元素 18 进栈，当前栈为(20,16,38,42,29,18)
int a=(Integer)stack.pop();            //栈顶元素 18 退栈并赋给整型变量 a
System.out.println(stack.peek());      //输出返回当前的栈顶元素 29
stack.pop();                           //栈顶元素 29 出栈，当前栈变为(20,16,38,42)
stack.isStack();                       //返回 false，因为当前栈非空
stack.push(x/3);      //将 x/3 的值 26 压入当前栈中，当前栈变为(20,16,38,42,26)
```

5.2 栈的顺序存储结构和操作实现

与线性表的情况相同，栈的顺序存储结构同样需要使用一个数组和一个整型变量，利用数组来顺序存储栈中的所有元素，利用整型变量来存储栈顶元素的下标位置。假定存储

栈的数组用 stackArray[minSize]表示，指示栈顶元素位置的整型变量用 top 表示，通常又称 top 为**栈顶指针**，这些对象的具体定义形式为：

```
final int minSize=10;            //假定存储栈的一维数组的初始长度为 10
private Object[] stackArray;     //定义存储栈的数组引用
private int top;                 //定义数组中所保存栈的栈顶元素的下标位置
```

在顺序存储的栈中，top 的值为–1 表示栈空，每次向栈中压入一个元素时，首先使 top 增 1，用以指示新的栈顶位置，然后再把元素赋值到这个空位置上，每次从栈中弹出（删除）一个元素时，首先取出栈顶元素，然后使 top 减 1，指示出前一个元素成为新的栈顶元素。由此可知，对顺序栈的插入和删除运算相当于是在顺序存储的线性表的表尾进行的，其时间复杂度为 O(1)。

在一个顺序栈中，若 top 已经指向了最后一个下标位置，则表示栈满，若再向其插入新元素时就需要重新分配更大的数组空间，以满足继续添加新元素的要求；相反，若 top 的值已经等于–1，则表示栈空，则不能再进行读取和删除元素的操作，通常利用栈空作为循环结束的条件，表明栈中数据已经处理完毕。

设一个栈 S 为(a,b,c,d,e)，对应的顺序存储结构如图 5-1(a)所示。若向 S 中插入一个元素 f，则栈 S 对应如图 5-1(b)所示。若接着执行两次出栈操作后，则栈 S 对应如图 5-1(c)所示。若依次使栈 S 中的所有元素出栈，最后使 S 变为空，则如图 5-1(d)所示。在图 5-1 中栈是垂直画出的，并且使下标编号向上递增，这样的栈顶在上、栈底在下的表示，使得栈操作更加形象化。

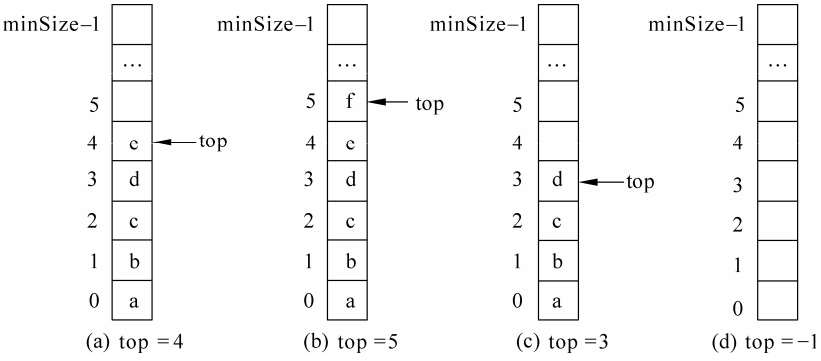


图 5-1　栈的顺序存储结构和操作过程

像对集合与线性表的处理一样，需要把顺序存储的栈定义为一个类，该类包括数据定义部分和相应构造方法的定义部分，以及实现栈接口 Stack 中所有操作方法的定义部分。假定采用顺序存储结构实现的栈类命名为 SequenceStack，该类的具体定义格式如下，每个方法的具体定义将在后面陆续讨论。

```
public class SequenceStack implements Stack    //顺序栈类的定义
{
    final int minSize=10;            //假定存储栈的一维数组的初始长度为 10
```

```
    private Object[] stackArray;     //定义存储栈的数组引用
    private int top;                 //定义数组中所保存栈的栈顶元素的下标位置

    public SequenceStack() {}        //无参的构造方法的定义
    public SequenceStack(int n) {}   //带初始长度参数的构造方法的定义
    public void push(Object obj) {} //元素进栈
    public Object pop() {}           //返回栈顶元素并退栈
    public Object peek() {}          //返回栈顶元素，栈状态不变
    public boolean isEmpty() {}      //判栈空
    public void clear() {}           //清除栈使之变为空
}
```

1．初始化顺序栈为空

在顺序栈类的构造方法中，需要初始化栈为空，并为保存栈的数组 stackArray 分配存储空间。该构造方法有两个，一个为无参构造方法，另一个为带有数组初始长度参数的构造方法。这两个构造方法的具体定义如下：

```
public SequenceStack()
{    //无参的构造方法的定义
    top=-1;                          //栈的初始为空，置 top 值为-1
    stackArray=new Object[minSize]; //数组初始长度为 minSize 的值 10
}
public SequenceStack(int n)
{    //带初始长度参数的构造方法的定义
    if(n<minSize) n=minSize;         //条件成立时将 n 修改为固定值 minSize
    top=-1;                          //置 top 的值为-1 表示空栈
    stackArray=new Object[n];        //给数组创建具有 n 大小的存储空间
}
```

2．向栈顶插入元素

```
public void push(Object obj)
{    //向栈顶插入一个新元素 obj
    if(top==stackArray.length-1) {       //对数组空间用完情况进行再分配
       Object[] p=new Object[top*2];     //新数组空间为原来的近二倍
       for(int i=0; i<=top; i++) p[i]=stackArray[i]; //复制原内容
       stackArray=p;                     //使 stackArray 指向新数组空间
    }
    top++;                               //栈顶指针加 1 表示进栈
    stackArray[top]=obj;                 //将新元素写入到新的栈顶位置
}
```

3．删除栈顶元素并返回

```
public Object pop()
{    //从栈中删除栈顶元素并返回
```

```
    if(top==-1) return null;     //栈为空时返回空值
    top--;                       //栈顶指针减 1 表示退栈
    return stackArray[top+1];    //返回原栈顶元素的值
}
```

注意：做出栈操作时，栈顶指针下移，但原栈顶位置中保存的元素依然存在，仍可以被利用，只是不属于当前栈中的元素而已。当前栈中的元素为从栈顶到栈底之间的所有元素。

4．读取栈顶元素的值

```
public Object peek()
{    //返回栈顶元素的值
    if(top==-1) return null;     //栈为空时返回空值
    return stackArray[top];      //返回栈顶元素的值
}
```

此算法只返回栈顶元素，而不改变栈的状态，即不修改栈顶元素和栈顶位置。

5．判断栈是否为空

```
public boolean isEmpty()
{    //判断栈是否为空栈，若为空则返回 true，否则返回 false
    return top==-1;
}
```

6. 清除栈中的所有内容使之成为空栈

```
public void clear()
{    //清除栈中的所有元素使之成为一个空栈
    top=-1;
}
```

可以采用下面程序 Example5_1.java 来调试上面对栈的各种操作的算法。

```
public class Example5_1
{
    public static void main(String[] args)
    {
        Stack sck=new SequenceStack();                    //定义并创建空栈 sck
        int []a={3,8,5,17,9,30,15,22};                    //定义数组 a
        for(int i=0; i<a.length; i++) sck.push(a[i]);     //数组 a 元素进栈
        System.out.println(sck.pop()+", "+sck.pop()+", "+sck.pop());
        sck.push(68);
        System.out.println(sck.peek()+", "+sck.pop()+", "+sck.pop());
        while(!sck.isEmpty()) System.out.print(sck.pop()+" ");
        System.out.println();
        sck.clear();
```

```
    }
}
```

得到的编译和运行结果如下：

```
D:\tsinghua>javac Stack.java
D:\tsinghua>javac SequenceStack.java
D:\tsinghua>javac Example5_1.java
D:\tsinghua>java  Example5_1
22, 15, 30
68, 68, 9
17 5 8 3
```

5.3 栈的链接存储结构和操作实现

栈的链接存储结构与线性表的链接存储结构完全相同，是通过由结点构成的单链表实现的，此时表头指针被称为栈顶指针，由栈顶指针指向的元素结点被称为栈顶结点，整个单链表被称为**链栈**，即链接存储的栈。当向一个链栈插入新元素时，是把它插入到栈顶，即使新元素结点的指针域指向原来的栈顶结点，而栈顶指针则修改为指向此新元素结点，使之成为新的栈顶结点。当从一个链栈中删除元素时，是把栈顶元素结点删除掉，即取出栈顶元素后，使栈顶指针指向其后继（下一个）结点。由此可知，对链栈的插入和删除操作是在单链表的表头进行的，其时间复杂度为 O(1)。

设一个栈为(a,b,c)，当采用链接存储时，对应的存储结构示意图如图 5-2(a)所示，其中，top 表示栈顶指针，其值为存储元素 c 结点的引用。当向这个栈插入一个元素 d 后，对应如图 5-2(b)所示。当从这个栈依次删除两个元素后，对应如图 5-2(c)所示。当链栈中的所有元素全部出栈后，栈顶指针 top 的值为空（null）。

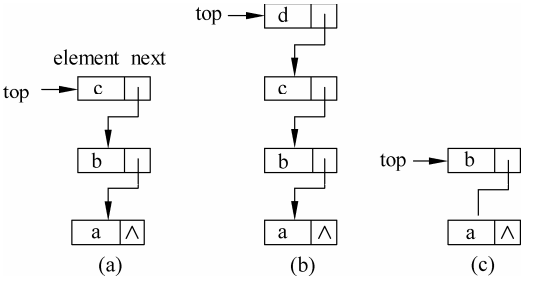


图 5-2　栈的链接存储结构及操作过程

因为只需要在链接存储的栈的顶端插入和删除结点，不会在其他任何位置进行插入和删除，所以使用一般的单链表存储栈即可。假定采用标识符 LinkStack 作为链接存储的栈类名，该类通常只需要包含一个私有数据成员，它是指向栈顶结点的引用 top，其类型为在第 2 章所定义的结点类型 Node；在该类中还要定义相应的无参构造函数，以及定义在 Stack 栈接口中已经声明的每种具体的栈操作。

链栈类的定义和每个操作方法的定义都很简单，整个类的具体定义如下：

```
public class LinkStack implements Stack
{    //实现 Stack 接口的链接栈类的定义
   private Node top;                  //定义 top 为栈顶指针(引用)

   public LinkStack()
   {    //无参的构造函数的定义
      top=null;                       //栈的初始为空，即把空值 null 赋给 top
   }

   public void push(Object obj)
   {    //向栈顶插入一个新元素 obj
      top=new Node(obj, top);         //相当于在单链表的表头插入结点
   }

   public Object pop()
   {    //从栈中删除栈顶元素并返回
      if(top==null) return null;      //返回空表示栈已空，无元素删除
      Node p=top;                     //栈顶结点的引用暂存到 p 中
      top=top.next;                   //栈顶指针指向下一个结点，即删除栈顶结点
      return p.element;               //返回原栈顶结点的值
   }

   public Object peek()
   {    //返回栈顶元素的值
      if(top==null) return null;      //栈空时返回空值
      return top.element;             //返回栈顶结点的值
   }

   public boolean isEmpty()
   {    //判断栈是否为空栈，若为空则返回 true，否则返回 false
      return top==null;
   }

   public void clear()
   {    //清除栈中的所有元素使之成为一个空栈
      top=null;
   }
}
```

为了验证链栈类 LinkStack 中每种运算方法的正确性，可以仍使用上面的对顺序栈类的调试程序 Example5_1.java，只要把主函数中的 SequenceStack()替换为 LinkStack()，然后编译 LinkStack.java 和 Example5_1.java 这两个类后，再运行 Example5_1.class 文件，将得到与上面完全相同的运行结果。

5.4 栈的简单应用举例

例 5-1 用键盘输入一批整数，然后按照相反的次序打印出来。

根据题意可知，后输入的整数将先被打印出来，这正好符合栈的后进先出的特点。所以此题很容易通过使用栈来解决。

由于此算法中需要用到键盘输入和对每行用空格分隔的整数进行处理，所以需要引入 java.io 包和 java.util 包。在 java.io 包中包含对使用文件（包括使用键盘和显示器的标准输入和输出文件）所采用的有关流类，当用键盘输入数据时，在新一行的开始按下 Ctrl+Z 组合键为文件结束符，表示键盘输入结束；在一般的文件中，文件结束符是用 ASCII 为 255 的特定字符表示的，对应的整数值为–1。在 java.util 包中包含对字符串处理有用的 StringTokenizer 类，把一个字符串定义为该类后，可将字符串中的内容按分隔符分开的每个子串依次进行处理，常用的分隔符为空格、回车、换行、制表（Tab）等。例如，假定字符串“25 48 36 75”被定义为 StringTokenizer 类对象后，可分解为 4 个整数子串，通过该类的相应方法可以读取每个子串，再转换成相应的整数后使用。关于 StringTokenizer 类的详细介绍，请参考 Java 的有关书籍。

此算法是栈的简单应用，其复杂性主要体现在文件操作、字符串处理和异常处理上。

```
public static void reverse()
{    //把用键盘输入的若干行整数利用栈的功能按照相反次序输出
   Stack sck=new SequenceStack();       //定义并创建一个空的顺序栈，链栈也可
   BufferedReader fin;                  //定义 fin 为带缓冲的文件读入器引用对象
   try {
      fin=new BufferedReader(new InputStreamReader(System.in));
         //fin 指向带缓冲的键盘读入器对象，通过它实现用键盘输入数据
      for(String s=fin.readLine(); s!=null; s=fin.readLine())
      {     //每次循环从键盘读入一行数据并赋给 s 字符串中，当读入文件结
            //束符时，s 为空指针，从而结束循环处理过程
         StringTokenizer r=new StringTokenizer(s);
           //通过字符串类 String 的对象 s 来创建对象 r，r 中保存的字符串
           //值与 s 中完全相同，但能够利用特定方法对 r 进行更有效的处理
         while(r.hasMoreTokens())
         {     //当 r 中存在被分隔符隔开的数据时，此循环条件为真
            String t=r.nextToken();       //取出 r 中的一个整数子串赋给 t
            int x=Integer.parseInt(t);    //将 t 的内容转换为整型数赋给 x
            sck.push(x);                  //将一个整数值 x 压入栈
         }  //此循环能够把一行中用空格分隔的所有整数依次保存到栈中
      }  //当读到新的一行数据只是文件结束符时才结束 for 循环
      fin.close();                        //关闭与键盘输入相关联的 fin 流
    }
    catch(IOException e) {                //对文件操作异常进行处理
       System.out.print("发生文件访问异常:!"+e);
```

```
            e.printStackTrace();   //显示出发生异常错误的跟踪信息，以便参考
        }
        while(!sck.isEmpty())      //依次输出栈中的数据，它与键盘输入次序相反
            System.out.print(sck.pop()+" ");
        System.out.println();
    }
```

假定键盘输入为：

```
78 63 45 82 91
34 55
^Z
```

则输出为：

```
55 34 91 82 45 63 78
```

例 5-2　堆栈在计算机语言的编译过程中用来进行语法检查，试编写一个算法，用来检查一个 Java 语言程序中的花括号、方括号和圆括号是否配对，若能够全部各自配对则返回逻辑真，否则返回逻辑假。

分析：在这个算法中，需要扫描待检查程序中的每一个字符，当扫描到每个花、中、圆左括号时，令其进栈，当扫描到每个花、中、圆右括号时，则检查栈顶是否为相应的左括号，若是则作退栈处理，若不是则表明出现了语法错误，应返回假（false）。当扫描到程序文件结尾后，若栈为空则表明没有发现括号配对错误，应返回真（true），否则表明栈中还有未配对的括号，应返回假。

根据分析，编写出算法如下：

```
public static boolean bracketsCheck(String fname)
{     //对由字符串类对象 fname 的内容为文件名的文本文件进行括号配对检查
    Stack sck=new SequenceStack();   //定义并创建一个暂存左括号的顺序栈或链栈
    try {
        BufferedReader br=new BufferedReader(new FileReader(fname));
            //建立带缓存的读入器对象 br，用于打开文本文件 fname
        int ch;
        for(ch=br.read(); ch!=-1; ch=br.read())
        {    //从头开始依次读取文件中的每个字符并进行处理，-1 为文件尾标志
          if(ch==39) {   //对单引号内字符不做配对检查
            while(true) {           //循环查找后一个单引号字符
                if((ch=br.read())==-1) return false;    //文件结束返回假
                if(ch==39)          //找到配对的单引号，读取下一字符并退出循环
                    {if((ch=br.read())==-1} return false;break;}
            }
          }
          else if(ch==34) {      //对双引号内字符不做配对检查
            while(true) {        //循环查找后一个双引号字符
              if((ch=br.read())==-1) return false;      //文件结束返回假
              if(ch==34) {       //找到配对的双引号,读取下一字符并退出循环
```

```
                    if((ch=br.read())==-1) return false;
                    break;
                 }
              }
           }
           switch ((char)ch) {          //对扫描到的当前字符进行分类处理
              case '{':
              case '[':
              case '(':
                sck.push((char)ch);     //出现以上 3 种左括号则进栈
                break;
              case '}':                 //读到右花括号时的处理情况
                if(sck.peek()==null) return false;    //若栈为空则返回假
                if((Character)sck.peek()=='{')
                    sck.pop();          //栈顶的左花括号出栈
                else return false;      //若不配对则返回假
                break;
              case ']':                 //读到右中括号时的处理情况
                if(sck.peek()==null) return false;    //若栈为空则返回假
                if((Character)sck.peek()=='[')
                    sck.pop();          //栈顶的左中括号出栈
                else return false;      //若不配对则返回假
                break;
              case ')':                 //读到右圆括号时的处理情况
                if(sck.peek()==null) return false;    //若栈为空则返回假
                if((Character)sck.peek()=='(')
                    sck.pop();          //栈顶的左圆括号出栈
                else return false;      //若不配对则返回假
              }                         //对其他字符不做任何处理
           }                            //for 循环结束
           br.close();
        }
        catch(IOException e) {          //对文件操作异常进行处理
           System.out.print("发生文件访问异常:!"+e);
           e.printStackTrace();
        }
        if(sck.isEmpty()) return true;  //括号配对成功返回真
        else return false;              //存在不配对的括号返回假
    }
```

例 5-3 把十进制整数转换为二至九之间的任一进制数并输出。

分析：由计算机基础知识可知，把一个十进制整数 x 转换为任一种 r 进制数得到的是一个 r 进制的整数，假定为 y，转换方法是逐次除基数 r 取余法。具体做法是：首先用十进制整数 x 除以基数 r，得到的整余数是 r 进制数 y 的最低位 y_0，接着以 x 除以 r 的整数商作为被除数，用它除以 r 得到的整余数是 y 的次最低位 y_1，依次类推，直到商为 0 时得到的

整余数是 y 的最高位 y_m，假定 y 共有 m+1 位。这样得到的 y 与 x 等值，y 的按权展开式为：

$$y=y_0+y_1r+y_2r^2+\cdots+y_mr^m$$

假定一个十进制整数为 3425，则把它转换为八进制数的过程如图 5-3 所示。

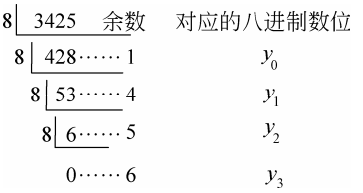


图 5-3　十进制整数 3425 转换为八进制数 6541 的过程

最后得到的八进制数为$(6541)_8$，对应的十进制数为 $6\times8^3+5\times8^2+4\times8+1=3425$，即为被转换的十进制数，证明转换过程是正确的。

从十进制整数转换为 r 进制数的过程中，由低到高依次得到 r 进制数中的每一位数字，而输出时又需要由高到低依次输出每一位。所以此问题适合利用栈来解决，具体算法描述为：

```
public static void transform(long num, int r)
{       //把一个长整型数 num 转换为一个 r 进制数输出
    Stack sck=new LinkStack();       //定义并创建一个暂存余数的链栈或顺序栈
    while(num!=0) {                  //由低到高求出 r 进制数的每一位并入栈
        int k=(int)(num % r);        //求出 r 进制中的一位赋给 k
        sck.push(k);                 //k 的值进栈
        num/=r;                      //num 对 r 的整数商仍保存在 num 中
    }
    while(!sck.isEmpty())            //由高到低输出 r 进制数的每一位数字
        System.out.print(sck.pop());
    System.out.println();
}
```

假定用下面的程序 Example5_2.java 来调试上面的 3 个算法。

```
import java.io.*;                    //进行文件打开和访问操作所必须包含的语句
import java.util.StringTokenizer;    //使用此类能够利用分隔符处理字符串

public class Example5_2
{
    public static void transform(long num, int r){}     //定义同上
    public static void reverse(){}                       //定义同上
    public static boolean bracketsCheck(String fname){} //定义同上

    public static void main(String[] args)
    {
        boolean b=bracketsCheck("Example5_1.java");
        if(b) System.out.println("程序文件中括号配对正确!");
```

```
        else System.out.print("程序文件中括号配对错误!");
        System.out.println();
        System.out.println("数制转换调试");
        transform(3425,8);
        transform(3425,6);
        transform(3425,4);
        transform(3425,2);
        System.out.println("输入若干行整数，在新行开始按下 Ctrl+Z 后回车结束!");
        reverse();      //按照与输入相反的次序输出整数
    }
}
```

编译和运行此程序后得到的结果如下：

```
D:\tsinghua>javac Example5_2.java
D:\tsinghua>java  Example5_2
程序文件中括号配对正确!

数制转换调试
6541
23505
311201
110101100001
输入若干行整数，在新行开始按下 Ctrl+Z 后回车结束!
78 63 45 82 91
34 55 19
^Z
19 55 34 91 82 45 63 78
```

5.5 算术表达式的计算

在计算机中进行算术表达式的计算是通过栈来实现的。本节首先讨论算术表达式的两种表示方法，即中缀表示法和后缀表示法，接着讨论后缀表达式求值的算法，最后讨论中缀表达式转换为后缀表达式的算法。

1．算术表达式的两种表示

通常书写的算术表达式是由操作数（又叫运算对象或运算量）和运算符以及改变运算次序的圆括号连接而成的式子。操作数可以是常量、变量和函数，同时还可以是表达式。运算符包括单目运算符和双目运算符两类，单目运算符只要求有一个操作数，并被放在该操作数的前面，双目运算符要求有两个操作数，并被放在这两个操作数的中间。单目运算符为取正（+）和取负（–），双目运算符有加（+）、减（–）、乘（*）和除（/）等。为了简便起见，在我们的讨论中只考虑双目运算符，并且仅限于+、–、*、/这 4 种运算。

例如，对于一个算术表达式 2+5*6，乘法运算符（*）的两个操作数是它两边的 5 和 6；而加法运算符（+）的两个操作数，一个是它前面的 2，另一个是它后面的 5*6 的结果即 30。把双目运算符出现在两个操作数中间的这种习惯表示叫做算术表达式的**中缀表示**，这种算术表达式被称为**中缀算术表达式**或**中缀表达式**。

中缀表达式的计算比较复杂，它必须遵守以下 3 条规则：

（1）先计算括号内，后计算括号外。

（2）在无括号或同层括号内，先进行乘除运算，后进行加减运算，即乘除运算的优先级高于加减运算的优先级。

（3）同一优先级运算，从左向右依次进行。

从这 3 条规则可以看出，在中缀表达式的计算过程中，既要考虑括号的作用，又要考虑运算符的优先级，还要考虑运算符出现的先后次序。因此，各运算符在计算过程中实际的运算次序往往同它们在表达式中先后出现的次序是不一致的。当然在手工计算时，直观判别出一个中缀表达式中哪个运算符最先算，哪个次之，……，哪个最后算并不困难，但通过计算机处理将麻烦得多。

那么，能否把中缀算术表达式转换成另一种形式的算术表达式，使手工和计算机的计算都简单化呢？回答是肯定的。波兰科学家卢卡谢维奇（Lukasiewicz）很早就提出了算术表达式的另一种表示，即**后缀表示**，又称**逆波兰表示法**，其定义是把运算符放在两个运算对象的后面。采用后缀表示的算术表达式被称为**后缀算术表达式**或**后缀表达式**。在后缀表达式中，不存在括号，也不存在运算符优先级的差别，计算过程完全按照运算符出现的先后次序进行，整个计算过程仅需一遍扫描便可完成，显然比中缀表达式的计算要简单得多。例如，对于后缀表达式“12 4 – 5.0 /”，因减法运算符在前，除法运算符在后，所以应先做减法，后做除法；减法的两个操作数是它前面的 12 和 4，其中，第一个数 12 是被减数，第二个数 4 是减数；除法的两个操作数是它前面的 12 减 4 的差（即 8）和 5.0，其中，8 是被除数，5.0 是除数，得到的运算结果为 1.6。

中缀算术表达式转换成对应的后缀算术表达式的规则是：把每个运算符都移到它的两个运算对象的后面，并用空格填补原来运算符位置，然后删除掉所有的括号即可。

例如，对于下列各中缀表达式：

（1）3/5+6

（2）16–9*(4+3)

（3）2*(x+y)/(1–x)

（4）(25+x)*(a*(a+b)+b)

对应的后缀表达式分别为：

（1）3 5/ 6+

（2）16 9 4 3+*–

（3）2 x y+* 1 x–/

（4）25 x+ a a b+* b+*

从以上实例可以看出，转换前后每个数据元素的前后次序没有改变，改变的只是表达式中每个运算符的位置和次序。

2．后缀表达式求值的算法

后缀表达式的求值比较简单，扫描一遍即可完成。它需要使用一个栈，假定用 sck 表示，其元素类型应为操作数的类型，假定为浮点型 double，用此栈存储后缀表达式中的操作数、计算过程中的中间结果以及最后结果。假定一个后缀算术表达式以一个字符串的方式提供，后缀表达式求值算法的基本思路是：对于一个以后缀算术表达式为内容的字符串，每次从该字符串中读入一个字符，若它是空格则不做任何处理，若它是运算符，则表明它的两个操作数已经在栈 sck 中，其中栈顶元素为运算符的后一个操作数，栈顶元素的前一个元素为运算符的前一个操作数，把它们弹出后进行相应运算并保存到一个双精度变量（假定为 x）中，否则，扫描到的字符必为数字或小数点，应把从此开始的双精度浮点数字符串转换为一个浮点数并存入 x 中，然后把计算或转换得到的浮点数（即 x 的值）压入到栈 sck 中。依次向下扫描每一个字符并进行上述处理，当处理后缀表达式结束后，最终结果保存在栈中，并且栈中仅存这一个值，把它弹出返回即可。整个算法描述为：

```
public static double compute(String str)
{    //计算字符串 str 中的后缀表达式的值
  //用 sck 栈存储操作数和中间计算结果
    Stack sck=new SequenceStack();
  //定义 x,y 用于保存浮点数，定义 a 用于保存由 str 转换得到的字符数组
    double x,y;
    char[] a=str.toCharArray();          //字符串的内容被赋给字符数组 a 中
    int i=0;
  //扫描后缀表达式中的每个字符，并进行相应处理
    while(i<a.length) {
        while(a[i]==' ') i++;            //扫描到空格字符不做任何处理
        switch(a[i]) {                   //对其他字符分情况处理
          case '+':                      //做栈顶两个元素的加法，和赋给 x
            x=(Double)sck.pop()+(Double)sck.pop();
            i++; break;
          case '-':       //做栈顶两个元素的减法，差赋给 x
            x=(Double)sck.pop();         //弹出减数
            x=(Double)sck.pop()-x;       //弹出被减数并做减法
            i++; break;
          case '*':                      //做栈顶两个元素的乘法，积赋给 x
            x=(Double)sck.pop()*(Double)sck.pop();
            i++; break;
          case '/':                      //做栈顶两个元素的除法，商赋给 x
            x=(Double)sck.pop();         //弹出除数
            if(Math.abs(x)>1e-6) x=(Double)sck.pop()/x;  //弹出被除数并计算
            else {      //除数为 0 时终止运行
                System.out.println("除数为 0，退出运行!");
                System.exit(1);
            }
```

```
            i++; break;
          default:  //扫描到的若是浮点数字符串，生成对应的浮点数
            if((a[i]<'0' || a[i]>'9') && a[i]!='.') {
                System.out.println("非法字符，退出运行!");
                System.exit(1);
            }
            x=0.0;  //利用 x 保存扫描到的整数部分的值
            while(a[i]>=48 && a[i]<=57) {     //48 是字符 0 的 ASCII 码
                x=x*10+a[i]-48; i++;
            }
            if(a[i]=='.') {       //转换小数部分
                i++;
                y=0.0;            //利用 y 保存扫描到的小数部分的值
                double j=10.0;    //用 j 作为相应小数位的权值
                while(a[i]>=48 && a[i]<=57) {
                    y=y+(a[i]-48)/j;
                    i++; j*=10;
                }
                x+=y;             //把小数部分合并到整数部分 x 中
            }                     //if 语句结束
        }                         // switch 语句结束
       //把扫描转换后或进行相应运算后得到的一个浮点数压入栈中
        sck.push(x);
    }  //while 语句结束
  //若计算结束后栈为空则中止运行
    if(sck.isEmpty()) {
        System.out.println("栈为空，退出运行!");
        System.exit(1);
    }
  //若栈中仅有一个元素，则它就是后缀表达式的值，否则为出错
    x=(Double)sck.pop();
    if(!sck.isEmpty())  {
        System.out.println("表达式格式错,退出运行!");
        System.exit(1);
    }
    return x;  //返回后缀表达式的值
}
```

此算法的运行时间主要花在 while(i<a.length)循环上，它从头到尾扫描后缀表达式中的每一个字符，若后缀表达式的字符串长度为 n，则此算法的时间复杂度为 O(n)。此算法在运行时所占用的临时空间主要取决于栈 sck 的大小，显然，它的最大深度不会超过表达式中所含操作数的个数，因为操作数的个数远比 n 小，所以此算法的空间复杂度不超过 O(n)。

假定参数字符串对象 str 中保存的后缀表达式为：

```
"10 3.5- 4.3 2.4+* 5/";
```

对应的中缀算术表达式为(10–3.5)*(4.3+2.4)/5，则使用如下两条语句调用上述函数得到的输出结果为 8.71。

```
double x=compute(str);
System.out.println((int)(x*100+0.5)/100.0);
```

在后一条输出语句中，使输出的 *x* 值能够保留两位小数，并对第 3 位小数进行四舍五入处理。如果 *x* 的值为 3.2452，则输出的结果为 3.25。

在进行这个后缀算术表达式求值的过程中，每处理一个操作数或运算符后，栈 sck 中保存的操作数和中间结果的变化情况如图 5-4 所示。

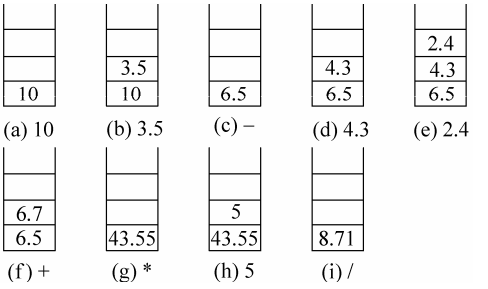


图 5-4　栈 sck 中数据的变化过程

3．把中缀表达式转换为后缀表达式的算法

设中缀算术表达式已经保存在具有 String 类型的 s 字符串中，转换后得到的后缀算术表达式拟存于可改变的字符串类型（StringBuffer）对象 sb 中，算法结束时再转换为一般的字符串类型（String）的对象返回。由中缀表达式转换为后缀表达式的规则可知：转换前后，表达式中的数值项（运算对象）的次序不变，而运算符的次序发生了变化，由处在两个运算对象的中间变为处在两个运算对象的后面，同时去掉了所有的括号。为了使转换正确，必须设定一个运算符栈，并在栈底开始放入一个特殊运算符，假定为'@'字符，让它具有最低的运算符优先级，假定其优先级设定为数值 0，此栈用来保存扫描中缀表达式所得到的暂不能放入后缀表达式中的运算符，待它的两个运算对象都放入到后缀表达式以后，再令其出栈并写入到后缀表达式中。

把中缀表达式转换为后缀表达式算法的思路是：从头到尾扫描中缀表达式中的每个字符（以'@'字符作为结束标记），对于不同类型的字符按不同情况进行处理。若遇到的是空格则认为是分隔符，不需要进行任何处理；若遇到的是数字或小数点，则直接写入到 s2 中，并在每个数值的后面写入一个空格；若遇到的是中缀表达式结束符'@'，则退出循环处理过程，接着进行结束和返回处理；若遇到的是左括号，则应把它压入到运算符栈中，待以它开始的括号内的表达式转换完毕后再出栈；若遇到的是右括号，则表明括号内的中缀表达式已经扫描完毕，把从栈顶直到保存着的对应左括号之间的运算符依次退栈并写入到 sb 串的结尾处；若遇到的是运算符，当该运算符的优先级大于栈顶运算符的优先级（加减运算符的优先级设定为 1，乘除运算符的优先级设定为 2，在栈底保存的特殊运算符'@'和左括号运算符'('的优先级设定为 0）时，表明该运算符的后一个运算对象还没有被扫描并放

入到 sb 串中，应把它暂存于运算符栈中，待它的后一个运算对象从 s 串中读出并写入到 sb 串之后，再令其出栈并写入 sb 串中；若遇到的运算符的优先级小于等于栈顶运算符的优先级，表明栈顶运算符的两个运算对象已经被保存到 sb 串中，应将栈顶运算符退栈并写入到 sb 串中，对于新的栈顶运算符仍继续进行比较和处理，直到被处理的运算符的优先级大于栈顶运算符的优先级为止，然后再将该运算符进栈。

按照以上过程扫描到中缀表达式字符串的结尾符'@'时，应把栈中剩余的运算符依次退栈并写入到后缀表达式中，整个转换过程就处理完毕，在 sb 中就得到了转换成的后缀表达式，把它转换成一般的字符串对象后返回即可。

将中缀算术表达式转换为后缀算术表达式的算法描述如下：

```
public static String change(String s)
{      //将字符串 s 中保存的中缀表达式(以'@'结尾)转换为后缀表达式并返回
  //定义用于暂存运算符的栈 sck，它可以为顺序，也可以为链接
    Stack sck=new SequenceStack();
  //给栈底放入'@'字符，它具有最低优先级 0
    sck.push('@');
  //将 s 中保存的中缀表达式转换为字符数组 a 的内容，以方便处理
    char[] a=s.toCharArray();
  //定义和创建一个可变化的字符串类型对象 sb，初始内容为空
    StringBuffer sb=new StringBuffer();
  //定义 i 和 ch 分别用来指示字符数组 a 中被处理元素的下标和元素值
    int i=0;
    char ch=a[i];
  //依次处理中缀表达式中的每个字符
    while(i<a.length) {
      //对于空格字符不做任何处理，循环读取下一个字符
        while(ch==' ') ch=a[++i];
      //如果访问到表达式结束符'@'则退出循环，做算法的结束处理
        if(ch=='@') break;
      //对于左括号，直接进栈
        if(ch=='(') {
            sck.push(ch); ch=a[++i];
        }
      //对于右括号，使括号内的仍停留在栈中的运算符依次出栈并写入 sb 结尾
        else if(ch==')') {
            while((Character)sck.peek()!='(') sb.append(sck.pop());
            sck.pop();        //删除栈顶的左括号
            ch=a[++i];
        }
      //对于运算符，使栈顶且不低于 ch 优先级的运算符依次出栈并写入 sb 结尾
        else if(ch=='+' || ch=='-' || ch=='*' || ch=='/') {
            char w=(Character)sck.peek();
            while(precedence(w)>=precedence(ch))
            {    // precedence(w)函数返回运算符形参 ch 的优先级
```

```
                sb.append(sck.pop());      //出栈
                w=(Character)sck.peek();  //读取新的栈顶元素
            }
            sck.push(ch);  //把 ch 运算符写入栈中
            ch=a[++i];
        }
      //此处必然为数字或小数点字符，否则为中缀表达式错误
        else {
          //若 ch 不是数字或小数点字符则退出运行
          if((ch<'0' ||ch>'9') && (ch!='.')) {
              System.out.println("中缀表达式表示错误!");
              System.exit(1);
          }
          //把一个数值中的每一位依次写入到 sb 的字符串的末尾
          while((ch>='0' && ch<='9') || ch=='.') {
              sb.append(ch);
              ch=a[++i];
          }
          //在 sb 的每个数值后面放入一个空格字符
          sb.append(' ');
        }
    }
  //做算法的结束处理，把暂存在栈中的运算符依次退栈并写入到 sb 末尾
    ch=(Character)sck.pop();
    while(ch!='@') {
        if(ch=='(') {System.out.println("表达式错!"); System.exit(1);}
        else {
            sb.append(ch);
            ch=(Character)sck.pop();
        }
    }
  //把可变化的字符串对象 sb 转换为 String 对象并返回
    return new String(sb);
}
```

在上面的算法中用到了 precedence(char op)方法，它返回运算符 op 的优先级，该方法的具体定义为：

```
public static int precedence(char op) {  //返回运算符 op 所对应的优先级
    switch(op) {
      case '+':
      case '-':
          return 1;  //定义加减运算的优先级为 1
      case '*':
      case '/':
          return 2;  //定义乘除运算的优先级为 2
```

```
        case '(':
        case '@':
        default:
            return 0;  //定义在栈中的左括号和栈底字符的优先级为 0
    }
}
```

在上面的中缀转后缀的算法中，中缀算术表达式中的每个字符均需要扫描一遍，对于从 s 中扫描得到的每个运算符，最多需要进行入 sck 栈、出 sck 栈和写入 sb 后缀表达式这 3 次操作，对于从 s 中扫描得到的每个数字或小数点，只需要把它直接写入到 sb 后缀表达式即可。所以，此算法的时间复杂度为 O(n)，n 为后缀表达式中字符的个数。该算法需要使用一个运算符栈，需要的深度不会超过中缀表达式中运算符的个数，所以此算法的空间复杂度至多也为 O(n)。

利用表达式的后缀表示和堆栈技术只需要两遍扫描就可完成中缀算术表达式的计算，显然比直接进行中缀算术表达式计算的扫描次数要少得多，在直接进行中缀算术表达式的计算时，原则上包含多少个运算符就需要进行多少次扫描过程才能够完成。

4．程序举例

可以采用下面的程序 Example5_3.java 来调试算术表达式求值的有关算法。

```
public class Example5_3
{
    public static int precedence(char op) {}    //方法定义同上
    public static double compute(String str) {} //方法定义同上
    public static String change(String s) {}    //方法定义同上

    public static void main(String[] args)
    {
        String s1="(10-3.5)*(4.3+2.4)/5@";
        String s2=change(s1);
        double x=compute(s2);
        System.out.println("中缀表达式: "+s1);
        System.out.println("后缀表达式: "+s2);
        System.out.println("计算结果: "+Math.rint(x*100)/100); //输出两位小数
    }
}
```

程序的编译和运行情况如下：

```
D:\tsinghua>javac  Example5_3.java
D:\tsinghua>java   Example5_3
中缀表达式: (10-3.5)*(4.3+2.4)/5@
后缀表达式: 10 3.5 -4.3 2.4 +*5 /
计算结果: 8.71
```

5.6 栈与递归

递归是一种非常重要的解决问题的方法，在计算机科学和数学等领域有着广泛的应用。使用递归往往能够使复杂的问题简单化，使相应的算法简明扼要、结构清晰。当求解一个问题时，若是通过求解与它具有同样解法的、相对简化的子问题而得到的，这就是递归。一个递归的求解问题必然包含终止递归的条件，当满足一定条件时就终止继续向下递归，从而使最小的问题不通过递归而直接得到解决，然后再依次返回解决较大的问题，最后解决整个问题。解决递归问题的算法称为递归算法，在递归算法中需要根据递归条件直接或间接地调用算法本身，当满足终止条件时结束递归调用。当然对于一些简单的递归问题，很容易把它转换为循环问题来解决，从而使编写出的算法更为有效。

例 5-4 采用递归算法求解正整数 n 的阶乘（$n!$）。

由数学知识可知，n 阶乘的递归定义为：它等于 n 乘以 $n-1$ 的阶乘，即 $n!=n*(n-1)!$，并且规定 1 和 0 的阶乘为 1。设函数 $f(n)=n!$，则 $f(n)$可表示为：

$$f(n)=\begin{cases}1 & (n=1 \text{ 或 } n=0)\\ n\times f(n-1) & (n>1)\end{cases}$$

在这里(n==1 || n==0)为递归终止条件，使函数返回 1，n>1 时以 $n-1$ 的值作为新的参数值进行递归调用，由 n 的值乘以 $f(n-1)$的递归调用的返回值求出 $f(n)$的值。

用 Java 语言编写出求解 $n!$的递归函数为：

```
public static long f(int n)
{
   if(n==1 || n==0) return 1;
   else return n*f(n-1);
}
```

从主函数或其他函数中调用此阶乘函数属于非递归调用，通过递归函数内部的调用表达式调用自身属于递归调用。当非递归调用此处的递归函数 $f(n)$时，首先把实参的值传送给形参 n，同时把调用后的返回地址保存起来，以便调用结束之后返回之用；接着执行循环体，当 n 等于 1 或 0 时则直接返回函数值 1，结束本次非递归调用或递归调用，并按返回地址返回到进行本次调用的调用函数的位置继续向下执行，当 n 大于 1 时，则以实参 $n-1$ 的值去调用本递归函数，返回 n 的值与本次递归调用所求值的乘积。因为进行一次递归调用，传送给形参 n 的值就减 1，所以最终必然导致 n 的值为 1，从而结束递归调用，接着不断地执行与递归调用相对应的返回操作，最后返回到进行非递归调用的调用函数的位置向下执行。

假定用 $f(4)$去调用 $f(n)$函数，该函数返回 $4\times f(3)$的值，因返回表达式中包含函数 $f(3)$，所以接着进行递归调用，返回 $3\times f(2)$的值，依次类推，当最后进行 $f(1)$递归调用，返回函数值 1 后，结束本次递归调用，返回到调用函数 $f(1)$的位置，从而计算出 $2\times f(1)$的值 2，即 $2\times f(1)=2\times 1=2$，2 作为调用函数 $f(2)$的返回值，返回到 $3\times f(2)$的表达式中，计算出 $f(3)$函数的返回值 6，即 $3\times f(2)=3\times 2=6$，接着返回到 $4\times f(3)$表达式中，计算出 $f(4)$的返回值

24，即 4×*f* (3)=4×6=24，从而结束整个调用过程，返回到调用函数 *f*(4)的位置继续向下执行。

上述调用和返回过程可形象地用图 5-5 表示出来。

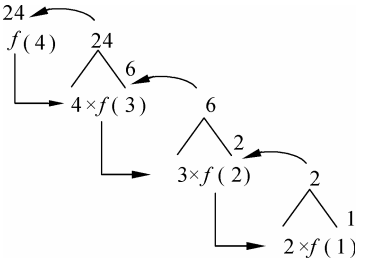


图 5-5　利用 *f*(4)调用 *f*(*n*)递归函数的执行过程

在计算机系统内，执行递归函数是通过自动使用栈来实现的，栈中的每个元素包含递归函数的每个参数域、函数体内的每个局部变量域和调用后的返回地址域。每次进行函数调用时，都把相应的值压入栈，每次调用结束时，都按照本次返回地址返回到指定的位置执行，并且自动作一次退栈操作，使得上一次调用所使用的参数成为新的栈顶，继续被使用。

例如，对于求 *n* 阶乘的递归函数 *f*(*n*)，当调用它时系统自动建立一个栈，该栈中的元素包含值参 *n* 的域和返回地址域（假定用标识符 *r* 表示），假定用 *f*(4)去调用 *f*(*n*)函数，调用后的返回地址用 r_1 表示，在 *f*(*n*)函数中，每次进行 *f*(*n*–1)调用的返回地址（即 *f*(*n*–1)所在的位置）用 r_2 表示，则系统所使用栈的数据变化情况如图 5-6 所示。

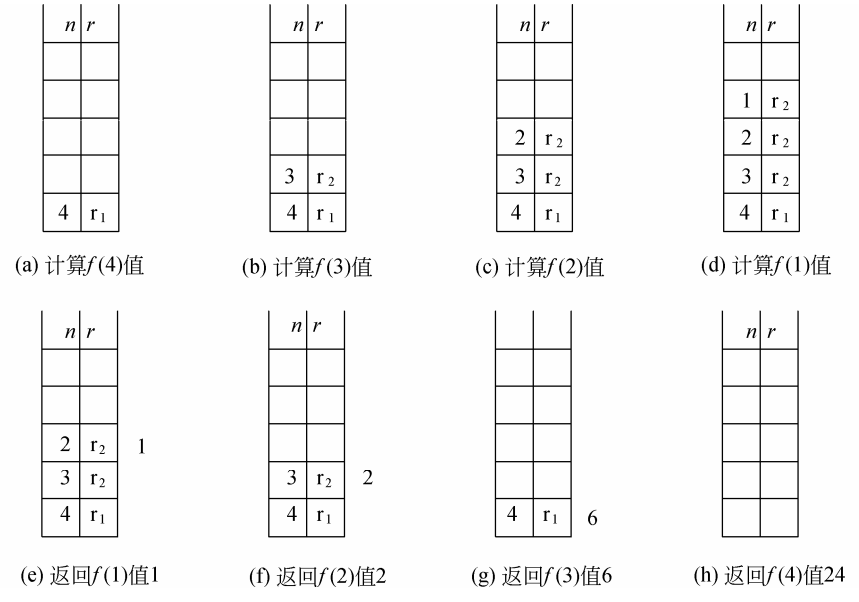


图 5-6　进行 *f*(4)调用时系统工作栈的变化过程

在图 5-6 中，每个栈的栈顶元素中的 *n* 域就是调用 *f*(*n*)函数时为值参 *n* 所分配的存储空间，*r* 域就是为保存当前一次调用结束后的返回地址所分配的存储空间。例如，进行 *f*(4)调

用时，栈顶元素中的值参 *n* 域保存的值为 4，返回地址域保存的值为 r_1，当执行 *f*(4)调用结束（即执行完返回语句或函数体最后的右花括号结束符）时，就返回到 r_1 的位置执行。又如，当执行 *f*(3)调用时，栈顶元素中的值参 *n* 域保存的值为 3，返回地址域保存的值为 r_2，当调用 *f* (3)结束后，就返回到 r_2 的位置（即上一层返回表达式中乘号后面的位置）执行。

当调用 *f*(*n*)算法时，系统所使用栈的最大深度为 *n*，*n* 为首次调用时传送来的实参的值，所以其空间复杂度为 O(*n*)。又因为每执行一次递归调用就是执行一个条件语句，其时间复杂度为 O(1)，执行整个算法求出 *n*!的值需要进行 *n* 次调用，所以其时间复杂度也为 O(*n*)。由于采用循环方法求解 *n*!的问题，其空间复杂度为 O(1)，时间复杂度为 O(*n*)，并且省去进出栈的烦琐操作，显然比采用递归算法更为有效。这里对求 *n* 阶乘采用递归算法，只是为了详细说明系统对递归算法的处理过程，以便读者能够理解更复杂的递归算法。

例 5-5 编写一个算法输出 *n* 个二进制位的所有可能的编码值。

分析：每个二进制位取 0 和 1 两个值。根据题意，当 *n* 为 1 时有两个编码值 0 和 1，当 *n* 为 2 时有 4 个编码值，依次为 00、01、10 和 11，当 *n* 为 3 时有 8 个编码值，依次为 000、001、010、011、100、101、110 和 111。总之，对于 *n* 个二进制位，所有可能的编码值为 2^n 个，每个编码值都有 *n* 位。

n 个二进制位的 2^n 种不同的编码值可以写做 $2\times2^{n-1}$，其中，2^{n-1} 种是 *n*–1 个二进制位的所有编码值，每种包含 *n*–1 个二进制位。*n* 个二进制位的每一种编码值是在 *n*–1 个二进制位的每个编码值的前面分别加上 0 或 1 而得到的结果，合起来正好是 $2\times2^{n-1}=2^n$ 种编码。由此可以看出它是一个递归的计算过程。

设 *n* 个二进制位用一个整型数组 b[*n*]来表示，要得到 b [0]~ b [*n*–1]这 *n* 个二进制位的每一种可能的编码，则要首先在 b [0]被置 0 的情况下得到 b [1]~ b [*n*–1]这 *n*–1 个二进制位的每一种可能的编码，然后在 b [0]被置 1 的情况下得到 b[1]~b[*n*–1]这 *n*–1 个二进制位的每一种可能的编码；同理，要得到 b[1]~b[*n*–1]这 *n*–1 个二进制位的每一种可能的编码，则要首先在 b[1]被置 0 的情况下得到 b[2]~b[*n*–1]这 *n*–2 个二进制位的每一种可能的编码，然后在 b[1]被置 1 的情况下得到 b[2]~b[*n*–1]这 *n*–2 个二进制位的每一种可能的编码；依次类推，直到最后一个二进制位 b[*n*–1]被置 0 后输出整个数组值和被置 1 后输出整个数组值为止。

下面是对 b[0]~b[*n*–1]之间的 *n* 个二进制位输出其所有编码的递归算法，初始非递归调用时应把实参值 0 传送给形参 *k*。

```
public static void coding(int b[], int k, int n)
{
    if (k==n)
    {    //终止递归，输出在 b 数组中排列好的一个二进制数
        for (int i=0; i<n; i++)
            System.out.print(b[i]);
        System.out.print("   ");
    }
    else
    { //把下标为 k 的元素置 0 后，从下标 k+1 起递归调用
        b[k]=0; coding(b, k+1, n);
```

```
        //把下标为 k 的元素置 1 后，从下标 k+1 起递归调用
          b[k]=1; coding(b, k+1, n);
      }
  }
```

此算法的每一次调用都要引起两次递归调用，当 $k==n$ 时结束本次调用，返回到原来调用函数的位置继续执行。因第一次非递归调用时传送给 k 的值为 0，所以共需要进行 $2^{n+1}-1$ 次递归调用（含最开始一次的非递归调用）。对于 n 个二进制位共需要输出 2^n 种所有不同的编码值，因此在 $2^{n+1}-1$ 次递归调用中共有 2^n 次递归调用输出数组 b 的值。例如当 n=3 时，整个递归调用的次数为 15 次，输出 8 种不同的编码；当 n=4 时，整个递归调用的次数为 31 次，输出 16 种不同的编码。

当用户调用这个算法时，系统自动建立一个栈，该栈包含整型数组 b 的引用域，整型参数 k 和 n 的值域，以及返回地址 r 的域。假定最开始非递归调用后的返回地址为 r_1，该算法中第一条递归调用语句执行后的返回地址（即为执行 b[k]=1 语句的开始地址）为 r_2，第二条递归调用语句执行后的返回地址（即为 else 语句块结束的地址，亦即算法的结束地址）为 r_3，并假定用 coding(a,0,3)去调用这个算法，其中，a 是一个元素个数大于等于 3 的整型数组，感兴趣的读者可以画出系统栈和数组 a 在算法执行过程中的变化状态，由于参数 b 和 n 的值为 a 和 3，始终保持不变，在栈中不用给出它们所对应的域，只需给出 k 值域和返回地址 r 域即可。

此算法在执行过程中需要被调用 $2^{n+1}-1$ 次，其中有 2^n 次需要调用输出数组 b 中 n 个元素的值，所以算法的时间复杂度为 $O(n\times 2^n)$，该算法所使用的系统栈的最大深度为 n+1，所以其空间复杂度为 $O(n)$，n 为待编码的二进制位的个数。

例 5-6　求解迷宫问题。

分析：一个迷宫包含 m 行×n 列个小方格，每个方格用 0 表示可通行，用 1 表示墙壁，即不可通行。迷宫中通常有一个入口和一个出口，设入口点的坐标为(1,1)，出口点的坐标为(m, n)，当然入口点和出口点的值应均为 0，即均可通行。从迷宫中的某一个坐标位置向东、南、西、北任一方向移动一步（即一个方格）时，若前面的小方格为 0，则可前进一步，否则通行受阻，不能前进，应按顺时针改变为下一个方向移动。求解迷宫问题是从入口点出发寻找一条通向出口点的路径，并打印出这条路径，即经过的每个小方格的坐标。如图 5-7(a)所示是一个 6×8 的迷宫，入口点坐标为(1,1)，出口点坐标为(6,8)，其中的一条路径为(1,1),(1,2),(2,2),(2,3),(3,3),(3,4),(3,5),(3,6),(4,6),(4,7),(5,7),(6,7),(6,8)。

在一个迷宫中，中间的每个方格位置都有 4 个可选择的移动方向，而在 4 个顶点只有两个方向，并且每个顶点的两个方向均有差别，每条边线上除顶点之外的每个位置只有 3 个方向，并且也都有差别。为了在求解迷宫的算法中避免判断边界条件和进行不同处理的麻烦，使每一个方格都能够试着按 4 个方向移动，可在迷宫的周围镶上边框，在边框的每个方格里填上 1，作为墙壁，如图 5-7(b)所示。这样需要用一个[m+2][n+2]大小的二维整型数组（假定用 maze 表示数组名）来存储迷宫数据。

	1	2	3	4	5	6	7	8
1	0	0	1	1	0	1	0	1
2	1	0	0	1	1	0	0	0
3	0	0	0	0	0	0	1	1
4	1	1	0	1	1	0	0	0
5	0	0	0	0	0	1	0	1
6	1	0	1	0	0	0	0	0

(a) 一个 6×8 的迷宫

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	0	1	0	1	1
2	1	1	0	0	1	1	0	0	0	1
3	1	0	0	0	0	0	0	1	1	1
4	1	1	1	0	1	1	0	0	0	1
5	1	0	0	0	0	0	1	0	1	1
6	1	1	0	1	0	0	0	0	0	1
7	1	1	1	1	1	1	1	1	1	1

(b) 带四周墙壁的迷宫

图 5-7　迷宫阵列图

当从迷宫中的一个位置（称它为当前位置）前进到下一个位置时，下一个位置相对于当前位置的位移量（包括行位移量和列位移量）随着前进方向的不同而不同，东、南、西、北（即右、下、左、上）各方向的行和列位移量依次为(0, 1)、(1, 0)、(0, –1)和(–1, 0)。假定用一个 4×2 的整型数组 move 来存储位移量数据，则 move 数组的内容为{{0,1},{1,0},{0,–1},{–1,0}}，每个元素 move[0]~move[3]依次存储向东、南、西、北每个方向移动一步的位移量。例如，move[1][0]和 move[1][1]分别为从当前位置向南移动一步的行位移量和列位移量，其值分别为 1 和 0。

在求解迷宫问题时，还需要使用一个与存储迷宫数据的 maze 数组同样大小的辅助数组，假定用标识符 mark 表示，用它来标识迷宫中对应位置是否被访问过。该数组每个元素的初始值为 0，表示迷宫中的所有位置均没有被访问过。每访问迷宫中一个可通行的位置时，都使 mark 数组中对应元素置 1，表示该位置已经被访问过，以后不会再访问到，这样才能够探索新的路径，避免重走已经走不通的老路。

为了寻找从入口点到出口点的一条通路，首先从入口点出发，按照东、南、西、北各方向的次序试探前进，若向东可通行，同时没有被访问过，则向东前进一个方格；否则表明向东没有通向出口的路径，接着应向南方向试着前进，若向南可通行同时没有被访问过，应向南前进一步；否则依次向西和向北试探。如果试探完当前位置上的所有方向后都没有通路，则应退回一步，从到达该当前位置的上一个方格位置的下一个方向试探着前进，如果到达该当前位置的方向为东，则上一个方格位置的下一个方向为南。因此，每前进一步都要记录其上一步的坐标位置以及前进到此步的方向，以便退回之用，这正好需要用栈来解决，每前进

一步时，都把当前位置和前进方向进栈，接着使向前一步后的新位置成为当前位置，若从当前位置无法继续前进时，就做一次退栈操作，从上一次位置的下一个方向试探着前进。若当前位置是出口点时，则表明找到了一条从入口点到出口点的路径，应结束算法执行，此时路径上的每个方格坐标（除出口坐标外）均被记录在栈中。若在算法执行过程中做退栈操作时栈为空，则表明入口点也已经退栈，并且其所有方向都已访问过，没有通向出口点的路径，此时应结束算法，返回假，表示无通路。

栈和递归是可以相互转换的，当编写递归算法时，虽然表面上没有使用栈，但系统执行时会自动建立和使用栈。求解迷宫问题也是一个递归问题，适合采用递归算法来解决。若迷宫中的当前位置（初始为入口点）就是出口位置，则表示找到了通向出口的一条路径，应返回 true 结束递归；若当前位置上的所有方向都试探完毕，表明从当前位置出发没有寻找到通向出口点的路径，应返回 false 结束递归；若从当前位置按东、南、西、北方向的次序前进到下一个位置时，若该位置可通行且没有被访问过，则应以该位置为参数进行递归调用，若返回 true 的话，表明从该位置到出口点有通路，输出该位置坐标，或者把该位置坐标保存到一个栈后，继续向上一个位置返回 true 结束递归。

根据以上分析思路，我们可以试着编写出求解迷宫问题的递归算法，该算法需要带有 maze 和 mark 的二维整型数组参数，用来分别保存带四周墙壁的迷宫数据和访问标记，它们的行数和列数应均为 m+2 和 n+2，其中，m 和 n 是一个迷宫矩阵的行数和列数；还需要带有 move 二维整型数组，用来保存向每个方向前进一步的行和列的位移量数据，该二维数组的行和列数应分别为 4 和 2；还要带有表示当前试探位置的行坐标 x 和列坐标 y，它的初始值（即开始调用该算法时所传入的实参值）应为入口点的行列坐标 1 和 1；另外，还需要一个存储路径上所有坐标点的数据栈参数，当迷宫中存在着一条通路时，该通路上所有坐标点将按照从出口到入口的先后次序压入栈中，调用该算法结束后，再依次弹出栈中的坐标点并输出，就得到了从入口到出口的坐标路径。

下面给出此算法的具体描述。

```
public static boolean seekPath(Stack sck, int [][]maze, int [][]mark,
                               int [][]move, int x, int y)
{       //从迷宫中坐标点(x,y)的位置寻找通向终点(m,n)的路径，若找到则返回 true，
        //否则返回 false，同时在 sck 中记录该路径
  //i 作为循环变量，代表从当前位置移到下一个位置的方向
    int i;
  //g 和 h 作为下一个位置的行坐标和列坐标
    int g,h;
  //若到达出口点返回 true 结束递归，判 x 和 y 的坐标值是否分别等于 m 和 n
    if((x==maze.length-2)&&(y==maze[0].length-2)) return true;
  //依次按每个方向寻找通向终点的路径，i=0、1、2、3 分别表示东、南、西、北方向
    for(i=0; i<4; i++) {
      //求出下一个位置的行坐标和列坐标
        g=x+move[i][0];
        h=y+move[i][1];
      //若下一位置可通行同时没有被访问过，则从该位置起递归查找
        if((maze[g][h]==0)&&(mark[g][h]==0)) {
```

```
            //置 mark 数组中对应位置为 1，表明已访问过
              mark[g][h]=1;
            //当条件语句中的递归调用返回 true，则表明从(g,h)到终点存在通路，
            //该位置坐标进栈，同时返回 true，否则进入下一轮 for 循环
              if(seekPath(sck,maze,mark, move, g, h)) {
                 String s="("+String.valueOf(g)+","+String.valueOf(h)+") ";
                 sck.push(s);   //将表示方格坐标的 s 字符串压入栈
                 return true;
              } //end if
           } //end if
        } //end for
      //从当前位置(x,y)没有通向终点的路径，应返回 false
        return false;
    }
```

可以用下面的 Example5_4.java 程序调试上面的求解迷宫的算法。

```
public class Example5_4
{
    public static boolean seekPath(Stack sck, int [][]maze, int [][]mark,
                                int [][]move, int x, int y) {} //同上

    public static void main(String[] args)
    {
      //定义并创建一个顺序栈 sck，用来记录路径上的坐标点
        Stack sck=new SequenceStack();
        int [][]maze={{1,1,1,1,1,1,1,1,1,1},{1,0,0,1,1,0,1,0,1,1},
                      {1,1,0,0,1,1,0,0,0,1},{1,0,0,0,0,0,0,1,1,1},
                      {1,1,1,0,1,1,0,0,0,1},{1,0,0,0,0,0,1,0,1,1},
                      {1,1,0,1,0,0,0,0,0,1},{1,1,1,1,1,1,1,1,1,1}};
      //定义并创建一个二维整型数组 mark，每个元素被初始化为 0
        int [][]mark=new int[8][10];
        for(int i=0; i<8; i++) for(int j=0; j<10; j++) mark[i][j]=0;
      //定义并创建一个二维整型数组 move，用 4 个方向的位移量坐标进行初始化
        int [][]move={{0,1},{1,0},{0,-1},{-1,0}};
      //设置入口点坐标对应的访问标记值为 1，表明已访问
        mark[1][1]=1;
      //从入口点(1,1)开始调用求解迷宫的递归算法
        if(seekPath(sck,maze,mark,move,1,1)) {  //调用返回真则输出路径
           System.out.println("从迷宫入口到出口的一条搜索路径为:");
           System.out.print("("+1+","+1+") "); //首先输出入口点坐标
           while(!sck.isEmpty()) System.out.print(sck.pop());
           System.out.println();
        }
        else System.out.println("此迷宫不存在从入口到出口的任何通路!");
    }
```

```
}
```

当按图 5-7 输入迷宫数据，则得到如下编译和运行结果。

```
D:\tsinghua>javac Example5_4.java
D:\tsinghua>java  Example5_4
从迷宫入口到出口的一条搜索路径为:
 (1,1) (1,2) (2,2) (2,3) (3,3) (3,4) (3,5) (3,6) (4,6) (4,7) (5,7) (6,7) (6,8)
```

例 5-7　求解汉诺塔（Tower of Hanoi）问题。大意是：有 3 个台柱，分别编号为 A、B、C 或 1、2、3，在 A 柱上穿有 n 个圆盘，每个圆盘的直径均不同，并且按照直径从大到小的次序叠放在柱子上；要求把 A 柱上的 n 个圆盘搬移到 C 柱上，B 柱此时可以作为过渡柱使用，并且每次只能搬动一个圆盘，同时必须保证在任何柱子上的圆盘在任何时候都要按序码放，即大的在下，小的在上；当把若干个圆盘从一个柱子搬到另一个柱子时，剩余的第三个柱子作为过渡柱使用；题目要求编写出一个算法，输出搬动圆盘的过程。

分析：若一个柱子上只有一个圆盘，则不需要使用过渡台柱，直接把它放到目的柱上即可。若一个柱子上有两个圆盘，则先把一个（只能是上面一个）放到过渡柱子上，再把另一个放到目的柱上，最后把过渡柱上的一个圆盘放到目的柱上，到此完成搬动过程。若一个柱子上有 3 个、4 个、…，又如何解决呢？必须找出适用于任意多个（即大于等于 2 个）情况的通用方法才行。由此可能想到递归，即先把原柱子上的 n–1 个圆盘设法搬到过渡柱上，再把原柱子上剩下的最后一个圆盘直接搬到目的柱上，最后设法把过渡柱上的 n–1 个圆盘搬到目的柱上，从而完成全部搬动过程。当把 n–1 个圆盘从一个柱子搬动到另一个柱子时，若它的值不是一个，又需要使用第三个柱子作为过渡。此递归就是把 n 的问题化解为两个 n–1 的问题，当 n 等于 1 时不需要再向下递归，只需要直接移动到目的柱即可。

例如，当 A 柱上有 3 个圆盘，要求把它移动到 C 柱上，则需要以下 3 步完成：

第 1 步：把 A 柱上的两个圆盘移到过渡柱 B 上；

第 2 步：把 A 柱上剩下的一个圆盘直接移到目的柱 C 上；

第 3 步：把过渡柱 B 上的两个圆盘移到目的柱 C 上。

对于上面的第 1 步还需要递归完成，具体又细分为以下 3 步：

第 1.1 步：把 A 柱上的一个圆盘直接移到此时的过渡柱 C 上；

第 1.2 步：把 A 柱上剩余的一个圆盘直接移到此时的目的柱 B 上；

第 1.3 步：把此时的过渡柱 C 上的一个圆盘直接移到此时的目的柱 B 上。

对于上面的第 3 步也需要递归完成，具体又细分为以下 3 步：

第 3.1 步：把 B 柱上的一个圆盘直接移到此时的过渡柱 A 上；

第 3.2 步：把 B 柱上剩余的一个圆盘直接移到此时的目的柱 C 上；

第 3.3 步：把此时的过渡柱 A 上的一个圆盘直接移到此时的目的柱 C 上。

上述整个移动过程为 7 个直接的移动步骤，依次为：

A→C; A→B; C→B; A→C; B→A; B→C; A→C

或用数字编号写为：

1→3; 1→2; 3→2; 1→3; 2→1; 2→3; 1→3

根据以上分析，设把 n 个盘子由字符参数 a 所表示的柱子搬到由字符参数 c 所表示的柱

子，用字符参数 b 所表示的柱子作为过渡，则编写出的递归算法如下：

```
public static void HanoiTower(int n, char a, char b, char c )
{
  //当只有一个盘子时，直接由 a 柱搬到 c 柱后结束当前调用
    if(n==1) {System.out.print(a+"→"+c+" "); return;}
  //当多于一个盘子时，向下递归
    else {
      //首先把 n-1 个盘子由参数 a 所表示的柱子搬到由参数 b 所表示
      //的柱子上，用参数 c 所表示的柱子作为过渡
        HanoiTower(n-1,a,c,b);
      //把由参数 a 所表示的柱子上的最后一个盘子搬到由参数 c 所表示的柱子上
        System.out.print(a+"→"+c+" ");
      //最后把 n-1 个盘子由参数 b 所表示的柱子搬到由参数 c 所表示
      //的柱子上，用参数 a 所表示的柱子作为过渡
        HanoiTower(n-1,b,a,c);
    }
}
```

假定采用 HanoiTower(3,'A','B','C')去调用该递归算法，则得到的整个递归调用关系如图 5-8 所示，它是一棵树结构，每个树叶结点下面的输出是执行 if(n==1)子句中输出语句的结果，每个树枝结点下面的输出是执行 else 子句中输出语句的结果。图中函数名简记为 H。

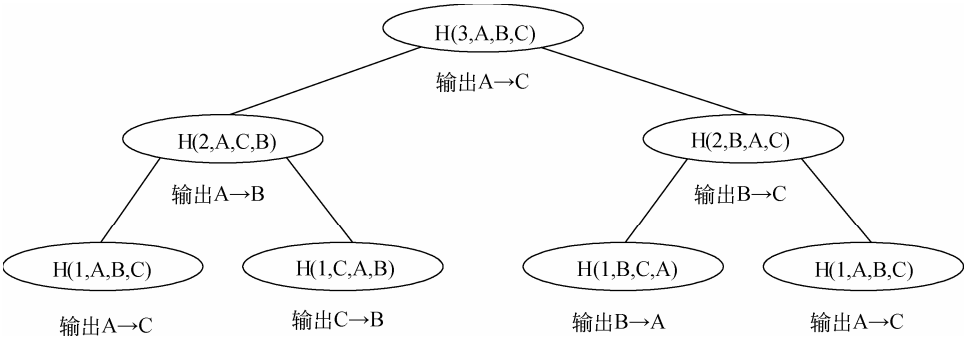


图 5-8　执行 HanoiTower(3,'A','B','C')时的递归调用关系树

调用上述递归算法时，若实参 n 的值为 1 则算法被执行 1 次，若值为 2 则被执行 3 次，若为 3 则被执行 7 次，依次类推，总之被执行 2^n-1 次。所以，此算法的时间复杂度为 $O(2^n)$。算法在执行时系统需要自动建立工作栈，栈的深度等于对应递归调用关系树的深度（即层数），该深度等于 n。所以，此算法的空间复杂度为 $O(n)$。

若采用 HanoiTower(4,'A','B','C')调用上述递归函数，则得到的输出结果如下所示：

A→B A→C B→C A→B C→A C→B A→B A→C B→C B→A C→A B→C A→B A→C B→C

5.7　队列

5.7.1　队列的定义和运算

1．队列的定义

队列（queue）简称**队**，它也是一种运算受限的线性表，其限制是仅允许在表的一端进行插入，而在表的另一端进行删除。把进行插入的一端称做**队尾**（rear），进行删除的一端称做**队首**（front）。向队列中插入新元素称为**进队**或**入队**，新元素进队后就成为新的队尾元素；从队列中删除元素称为**离队**或**出队**，元素离队后，其后继元素就成为队首元素。由于队列的插入和删除操作分别是在各自的一端进行，每个元素必然按照进入的先后次序离队，所以又把队列称为**先进先出表**（First In First Out，FIFO 表）。

在日常生活中，人们为购物或等车时所排的队形就是一个队列，新来购物或等车的人排到队尾（即进队），站在队首的人购得物品或上车后离开（即出队），当最后一人离队后，则队列为空。

假定有 a、b、c、d 这 4 个元素依次进队，则得到的队列为(a, b, c, d)，其中，字符 a 为队首元素，字符 d 为队尾元素。若从此队列中删除一个元素，则字符 a 出队，字符 b 成为新的队首元素，此队列变为(b, c, d)；若接着向该队列插入一个字符 e，则 e 成为新的队尾元素，此队列变为(b, c, d, e)；若接着做 3 次删除操作，则队列变为(e)，此时只有一个元素 e，它既是队首元素又是队尾元素，当它被删除后队列变为空。

2．队列的抽象数据类型

队列的抽象数据类型中的数据部分为采用任何存储方法存储的一个队列，操作部分包括元素进队、出队、读取队首元素、检查队列是否为空等方法。下面给出队列的抽象数据类型的具体定义：

```
ADT Queue is
    Data:
        采用任何存储方法存储的一个队列
    Operation:
        initQueue();            //创建一个队列并初始化队列为空
        enter(obj);             //元素进队，即向队列尾部添加一个新元素 obj
        leave();                //元素出队，即从队列首部删除队首元素并返回
        peek();                 //返回队首元素的值
        isEmpty();              //判断队列是否为空
        clear();                //清除队列中的所有元素使之变为一个空队
end Queue
```

队列的抽象数据类型也需要采用 Java 语言中的接口来描述，假定采用的接口名仍为 Queue。在队列接口 Queue 中，只需要包含抽象数据类型中的操作声明部分（创建和初始化

队列的操作除外），不需要包含其数据部分，队列的数据部分和对其进行初始化的操作，以及对其他操作方法的具体定义，应包含在实现队列接口 Queue 的相应子类中。

3．队列的接口定义

队列接口 Queue 的具体定义如下：

```
public interface Queue
{     //队列的抽象接口的定义
   void enter(Object obj);      //元素进队，即向队列尾部添加一个新元素 obj
   Object leave();              //元素出队，即从队列首部删除队首元素并返回
   Object peek();               //返回队首元素的值
   boolean isEmpty();           //判断队列是否为空
   void clear();                //清除队列中的所有元素使之变为一个空队
}
```

4．队列的运算举例

假定有一个初始为空的队列 q，其元素类型为整型，下面给出对队列 q 进行运算的一些例子。

```
q.enter(35);                    //元素 35 进队
int x=12; q.enter(2*x+3);       //元素 2*x+3 的值 27 进队
q.enter(-16);                   //元素-16 进队，此时的队列为(35,27,-16)
System.out.println(q.peek());   //读取并输出队首元素的值 35
q.leave(); q.leave();           //依次从队列中删除元素 35 和 27
while(!q.isEmpty())System.out.print(q.leave()+" ");
//依次输出队列 q 中的所有元素，因 q 中只有一个元素-16，所以此循环只输出它
```

5.7.2 队列的顺序存储结构和操作实现

队列的顺序存储结构需要使用一个数组和两至三个整型变量来实现，利用数组来顺序存储队列中的所有元素，利用第一个整型变量存储队首元素的位置（通常存储队首元素的前一个位置），利用第二个整型变量存储队尾元素的位置，利用第三个整型变量（若使用的话）存储队列的长度，即队列中当前已有的元素个数。把指向队首元素前一个位置的变量称为**队首指针**，由它加 1 就得到队首元素的下标位置；把指向队尾元素位置的变量称为**队尾指针**，由它可直接得到队尾元素的下标位置。假定存储队列的数组用 queueArray[]表示，队首指针和队尾指针分别用 front 和 rear 表示，存储队列长度的变量用 len 表示，则元素类型为 Object 的队列的顺序存储结构可通过下列一组变量的定义来实现：

```
final int minSize=10;               //假定存储队列的一维数组的初始长度为 10
private Object[] queueArray;        //定义存储队列的数组引用
private int front, rear, len;       //分别定义队首指针和队尾指针及队列长度
```

每次向队列插入一个元素，需要首先使队尾指针后移一个位置，然后再向这个位置写入新元素。当队尾指针指向数组空间的最后一个位置（即 queueArray.length–1）时，若队首元

素的前面仍存在空闲的位置，则表明队列未占满整个数组空间，下一个存储位置应是下标为 0 的空闲位置，因此，首先要使队尾指针指向下标为 0 的位置，然后再向该位置写入新元素。通过赋值表达式 rear=(rear+1)%queueArray.length 可使存储队列的整个数组空间变为首尾相接的一个环，所以顺序存储的队列又称为**循环队列**。在循环队列中，其存储空间是首尾循环利用的，当 rear 指向最后一个存储位置时，下一个所求的位置自动为数组空间的开始位置（即下标为 0 的位置）。

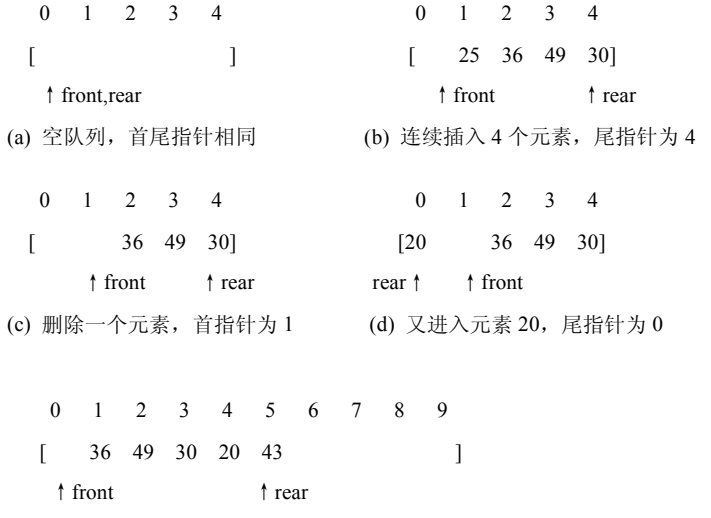
每次从队列中删除一个元素时，若队列非空，则首先把队首指针后移，使之指向队首元素，然后再返回该元素的值。使队首指针后移也必须采用取模运算，该计算表达式为 front=(front+1)%ququeArray.length，这样才能够实现存储空间的首尾相接，即循环利用。

当一个顺序队列中的长度域 len 的值为 0 时，表明该队列为空，则不能进行出队和读取队首元素的操作，当 len 域的值等于 queueArray.length 时，表明队列已满，即存储空间已被用完，此时应动态扩大存储空间，接着才能插入新元素。

在顺序存储的队列类型的定义中，若省略长度域 len 也是可行的，但此时的队列长度只能为 queueArray.length–1，也就是说必须有一个位置空闲着。这是因为若使用全部 queueArray.length 个位置存储队列，则当队首和队尾指针指向同一个位置时，也可能为空队，也可能为满队，就存在二义性，无法进行判断。为了解决这个矛盾，只有牺牲一个位置的存储空间，利用队首和队尾指针是否相等作为判断空队的条件，而利用队尾指针加 1 并对 queueArray.length 取模后是否等于队首指针（即队尾是否从后面又追上了队首）作为判断满队的条件。

由于对队列的插入和删除操作分别在两端进行，并且通过使用循环队列后，不需要比较和移动任何元素，所以其时间复杂度均为 O(1)。

图 5-9 给出了在顺序存储的循环队列中进行插入和删除元素的过程。



(a) 空队列，首尾指针相同　(b) 连续插入 4 个元素，尾指针为 4

(c) 删除一个元素，首指针为 1　(d) 又进入元素 20，尾指针为 0

(e) 当再插入元素 43 时，因队列满，扩大空间后继续插入，首指针为 0，尾指针为 5

图 5-9　在循环队列中插入和删除元素的过程

与对栈的情况一样，也要把顺序存储的队列定义为一个类，该类包括数据部分的定义和

相应构造方法的定义，以及实现队列接口 Queue 中所有操作的方法定义。假定采用顺序存储结构实现的队列类的名称为 SequenceQueue，该类的具体定义如下，每个操作方法的定义将随后介绍。

```
public class SequenceQueue implements Queue    //顺序队列类的定义
{
    final int minSize=10;              //假定存储队列的一维数组的初始长度为 10
    private Object queueArray[];       //定义存储队列的数组引用
    private int front, rear;           //定义队首和队尾指针

    public SequenceQueue() {}          //无参的构造方法的定义
    public SequenceQueue(int n) {}     //带初始长度参数的构造方法的定义
    public void enter(Object obj){}    //元素进队，即向队尾添加元素 obj
    public Object leave() {}           //元素出队，即从队首删除元素并返回
    public Object peek() {}            //返回队首元素的值
    public boolean isEmpty() {}        //判断队列是否为空
    public void clear() {}             //清除队列中的所有元素使之变为空队
}
```

下面给出顺序队列类中各成员方法的具体定义。

1. 利用构造方法初始化队列为空

在顺序队列类的构造方法中，需要初始化队列为空，并为保存队列的数组 queueArray 分配存储空间。该构造方法有两个，一个为无参构造方法，另一个为带有数组初始长度参数的构造方法。这两个构造方法的具体定义如下：

```
public SequenceQueue()                 //无参的构造函数的定义
{
    front=rear=0;                      //队列的初始为空，置队首和队尾指针值为 0
    queueArray=new Object[minSize]; //数组初始长度为 minSize 的值 10
}

public SequenceQueue(int n)            //带初始长度参数的构造函数的定义
{
    front=rear=0;                      //置队首和队尾指针值为 0,初始队列为空
    if(n<=minSize) n=minSize;          //将 n 的值最小设置为 minSize
    queueArray=new Object[n];          //给数组创建具有 n 大小的存储空间
}
```

2. 向队列插入元素

当向队列插入一个元素时，是把它写入到当前队尾元素的后面，为此，要首先让队尾指针循环后移一个位置，若队列已满，则需要重新分配更大的数组存储空间，通常是原数组空间的二倍，此时还需要把原数组的内容复制到新数组空间中，然后才能正常插入元素。

```
public void enter(Object obj)
{     //元素进队，即向队列尾部添加一个新元素 obj，使之成为新的队尾
    if((rear+1)%queueArray.length==front)
    {      //对队列满（即存储空间用完）的情况进行处理
        Object[] p=new Object[queueArray.length*2]; //扩大二倍的存储空间
        if(rear==queueArray.length-1)
        {      //把原队列内容复制到新空间中的对应位置，此时 front 值为 0
            for(int i=1; i<=rear; i++) p[i]=queueArray[i];
        }
        else { //先复制原队列的前面内容，再复制其后面内容
            int i, j=1;
            for(i=front+1; i<queueArray.length; i++,j++)
                p[j]=queueArray[i];
            for(i=0; i<=rear; i++,j++)
                p[j]=queueArray[i];
            front=0; rear=queueArray.length-1;    //复制后修改首尾指针
        }
        queueArray=p;                          //使 queueArray 指向新队列的存储空间
    }
    rear=(rear+1)%queueArray.length;     //求出队尾的下一个循环位置
    queueArray[rear]=obj;                //把 obj 的值赋给新的队尾位置
}
```

3. 从队列中删除元素

从队列中删除元素就是删除队首元素并返回该元素的值，首先要检查队列是否为空，即检查队首指针和队尾指针的值是否相等，当队列非空时，让队首指针循环后移一个元素位置，然后返回该位置的元素值；当队列为空时，返回空值，表示删除失败。

```
public Object leave()
{     //元素离队，即从队列首部删除队首元素并返回
    if(front==rear) return null;         //若队列为空则返回空值
    front=(front+1)%queueArray.length;  //使队首指针指向下一个位置
    return queueArray[front];            //返回队首元素的值
}
```

4. 读取队首元素

此算法很简单，当队列为空时返回空值，非空时返回队首指针的下一个位置的元素值。此算法不改变队首和队尾指针的值，即不影响队列的当前状态。

```
public Object peek()                                    //返回队首元素的值
{
    if(front==rear) return null;                        //若队列为空则返回空值
    return queueArray[(front+1)%queueArray.length]; //返回队首元素
}
```

5. 检查一个队列是否为空

若队列中队首和队尾指针指向了同一个下标位置，则队列为空，应返回 true，否则应返回 false。

```
public boolean isEmpty()                    //判断队列是否为空
{
    return front==rear;
}
```

6. 清除队列内容成为空队列

```
public void clear()                         //清除队列中的所有元素使之变为一个空队
{
    front=rear=0;
}
```

7. 调试程序举例

```
public class Example5_5
{
    public static void main(String[] args)
    {
        Queue que=new SequenceQueue();
        int []a={3,8,5,17,9,30,15,22,20,13,35,26};
        int i;
        for(i=0; i<a.length; i++) que.enter(a[i]);
        System.out.print(que.leave()+" ");
        System.out.print(que.leave()+" ");
        que.enter(68);
        System.out.print(que.peek()+" ");
        System.out.println(que.leave());
        while(!que.isEmpty()) System.out.print(que.leave()+" ");
        System.out.println();
        que.clear();
    }
}
```

程序编译和运行结果如下：

```
D:\tsinghua>javac Queue.java
D:\tsinghua>javac SequenceQueue.java
D:\tsinghua>javac Example5_5.java
D:\tsinghua>java  Example5_5
3 8 5 5
17 9 30 15 22 20 13 35 26 68
```

5.7.3　队列的链接存储结构和操作实现

队列的链接存储结构也是通过由结点构成的单链表实现的，此时只允许在单链表的表头进行删除和在单链表的表尾进行插入，因此它需要使用两个指针：队首指针 front 和队尾指针 rear。用 front 指向队首（即表头）结点，用 rear 指向队尾（即表尾）结点。用于存储队列的单链表简称链接队列或链队。假定链队中的结点类型仍为第 2 章已经定义过的单链表结点类型 Node，则队首和队尾指针的类型为 Node。一个链接队列的存储结构如图 5-10 所示。

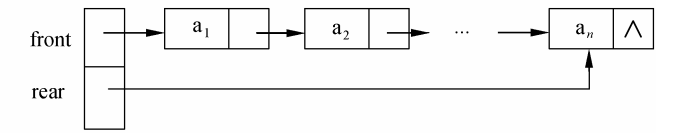


图 5-10　链接队列的存储结构

在链接存储的队列类中，其私有数据成员就是 front 和 rear，而操作方法就是对 Queue 接口中定义的所有抽象方法的实现。假定链接队列类的名称用 LinkQueue 表示，该类的具体定义如下：

```
public class LinkQueue implements Queue     //链接队列类的定义
{
    private Node front, rear;               //定义队首和队尾指针(引用)

    public LinkQueue()
    {     //无参构造方法的定义
        front=rear=null;      //队列初始为空，即使队首和队尾指针值为空
    }

    public void enter(Object obj)
    {     //向队列插入一个新元素 obj，即插入到队列尾部，成为新的队尾
        if(rear==null)    //若链队为空，则新结点既是队首结点又是队尾结点
            front=rear=new Node(obj, null);
        else              //若链队非空，则把新结点链接到队尾并修改队尾指针
            rear=rear.next=new Node(obj, null);
    }

    public Object leave()
    {     //从队列中删除队首元素并返回
        if(front==null) return null;      //对队列为空时的处理情况
        Node x=front;                     //队首结点的引用暂存 x
        front=front.next;                 //删除队首结点，相当于删除表头结点
        if(front==null) rear=null;        //若队列变为空，则队尾指针也须变为空
        return x.element;                 //返回原队首结点的值
    }
```

```
    public Object peek()
    {    //返回队首元素的值
        if(front==null) return null;    //对队列为空时的处理情况
        return front.element;          //返回队首结点的值
    }

    public boolean isEmpty()
    {    //判断队列是否为空，若为空则返回 true，否则返回 false
        return front==null;
    }

    public void clear()
    {    //清除队列中的所有元素使之为一个空队列
        front=rear=null;
    }
}
```

由于对链接队列的插入和删除运算都不需要进行结点的比较和移动，只是简单地根据队尾指针插入结点和根据队首指针删除结点，所以其时间复杂度均为 O(1)，当然，其他在队列上进行运算的时间复杂度也为 O (1)。

若修改上面调试顺序队列的程序 Example5_5.java 的主函数中的第 1 条语句，使之变为“Queue que=new LinkQueue();”，就可以直接调试链接队列了，将会得到相同的输出结果。当然要事先输入和编译好 LinkQueue.java 类。

在计算机科学中，除了上面介绍的一般队列外，还有一种特殊的队列叫做**优先级队列**。这种队列中的每个元素都带有一个优先级号，用以表示其优先级别。在优先级队列中，优先级最高的元素必须处在队首位置，因此，每次向它插入元素时，都要按照一定次序调整元素位置，确保把优先级最高的元素调整到队首，每次从中删除队首元素（即优先级最高的元素）时，也都要按照一定次序调整队列中的有关元素，确保把优先级最高的元素调整到队首。优先级队列在操作系统的各种调度算法中应用广泛，它需要使用以后将要介绍的堆结构来实现。