CHAPTER 3

第3章 栈和队列

栈和队列是两种常用的数据结构，它们的数据元素的逻辑关系也是线性关系，但在运算上不同于线性表。

本章主要学习要点如下：

（1）栈、队列和线性表的异同，栈和队列抽象数据类型的描述方法。

（2）顺序栈的基本运算算法设计。

（3）链栈的基本运算算法设计。

（4）顺序队的基本运算算法设计。

（5）链队的基本运算算法设计。

（6）双端队列和优先队列的应用。

（7）Java 中的 Stack < E >、Queue < E >、Deque < E >和 PriorityQueue < E >集合及其使用。

（8）综合运用栈和各种类型的队列解决一些复杂的实际问题。

3.1 栈

本节先介绍栈的定义，然后讨论栈的存储结构和基本运算算法设计，最后通过两个综合实例说明栈的应用。

3.1.1 栈的定义

先看一个示例，假设有一个老鼠洞，口径只能容纳一只老鼠，有若干只老鼠依次进洞（如图 3.1 所示），当到达洞底时，这些老鼠只能一只一只地按与原来进洞时相反的次序出洞，如图 3.2 所示。在这个例子中，老鼠洞就是一个栈，由于其进出洞口只能容纳一只老鼠，所以不论洞中有多少只老鼠，它们只能是一只一只地排列，从而构成一种线性关系。再看老鼠洞的主要操作，显然有进洞和出洞，进洞只能从洞口进，出洞也只能从洞口出。

抽象起来，栈是一种只能在一端进行插入或删除操作的线性表。在表中允许进行插入、删除操作的一端称为**栈顶**。栈顶的当前位置是动态的，由一

个称为栈顶指针的位置指示器来指示。表的另一端称为**栈底**。当栈中没有数据元素时称为**空栈**。栈的插入操作通常称为**进栈**或**入栈**,栈的删除操作通常称为**退栈**或**出栈**。

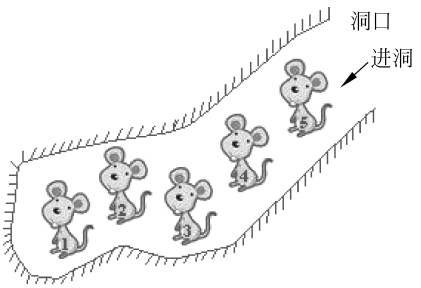


图 3.1　老鼠进洞的情况

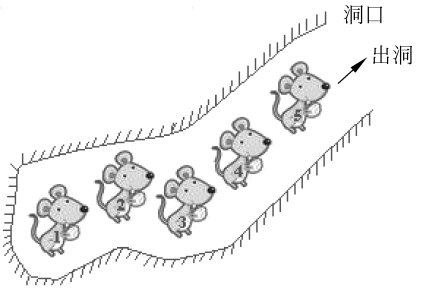


图 3.2　老鼠出洞的情况

说明:对于线性表,可以在中间和两端的任何地方插入和删除元素,而栈只能在同一端插入和删除元素。

栈的主要特点是"后进先出",即后进栈的元素先出栈。每次进栈的元素都放在原当前栈顶元素之前成为新的栈顶元素,每次出栈的元素都是原当前栈顶元素。栈也称为**后进先出表**。

抽象数据类型栈的定义如下:

ADT Stack
{
数据对象:
$D=\{a_i \mid 0\leqslant i\leqslant n-1, n\geqslant 0$,元素 a_i 为 E 类型$\}$
数据关系:
$R=\{r\}$
$r=\{<a_i, a_{i+1}> \mid a_i, a_{i+1}\in D, i=0,\cdots,n-2\}$
基本运算:
boolean empty():判断栈是否为空,若栈空返回真,否则返回假。
void push(E e):进栈操作,将元素 e 插入栈中作为栈顶元素。
E pop():出栈操作,返回栈顶元素。
E peek():取栈顶操作,返回当前的栈顶元素。
}

视频讲解

【例 3.1】 若元素进栈的顺序为 1234,能否得到 3142 的出栈序列?

解:为了让 3 作为第一个出栈元素,1、2 先进栈,第一次出栈 3,接着要么 2 出栈,要么 4 进栈后出栈,第二次出栈的元素不可能是 1,所以得不到 3142 的出栈序列。

【例 3.2】 用 S 表示进栈操作,X 表示出栈操作,若元素进栈的顺序为 1234,为了得到 1342 的出栈顺序,给出相应的 S 和 X 操作串。

解:为了得到 1342 的出栈顺序,其操作过程是 1 进栈,1 出栈,2 进栈,3 进栈,3 出栈,4 进栈,4 出栈,2 出栈,因此相应的 S 和 X 操作串为 SXSSXSXX。

【例 3.3】 设 n 个元素的进栈序列是 1,2,3,…,n,通过一个栈得到的出栈序列是 p_1,p_2,p_3,…,p_n,若 $p_1=n$,则 $p_i(2\leqslant i\leqslant n)$的值是什么?

解:当 $p_1=n$ 时,说明进栈序列的最后一个元素最先出栈,此时出栈序列只有一种——n,$n-1$,…,2,1,即 $p_1=n$,$p_2=n-1$,…,$p_{n-1}=2$,$p_n=1$,也就是说 $p_i+i=n+1$,推出

$p_i = n - i + 1$。

3.1.2　栈的顺序存储结构及其基本运算算法的实现

由于栈中元素的逻辑关系与线性表的相同，所以可以借鉴线性表的两种存储结构来存储栈。

当采用顺序存储时，分配一块连续的存储空间，用 data＜E＞数组来存放栈中元素，并用一个整型变量 top(栈顶指针)指向当前的栈顶，以反映栈中元素的变化。采用顺序存储的栈称为**顺序栈**。

顺序栈的存储结构如图 3.3 所示，capacity 为 data 数组的容量，由于将 data[0]端作为栈底，所以 top＋1 恰好表示栈中实际元素的个数。

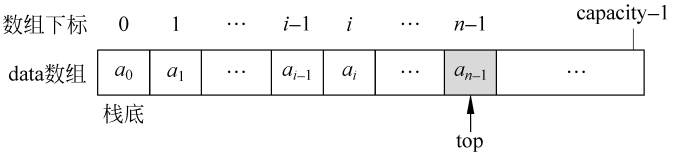


图 3.3　顺序栈的示意图

视频讲解

图 3.4 是一个栈的动态示意图，图 3.4(a)表示一个空栈，top＝－1；图 3.4(b)表示元素 a 进栈以后的状态；图 3.4(c)表示元素 b、c、d 进栈以后的状态；图 3.4(d)表示出栈元素 d 以后的状态。

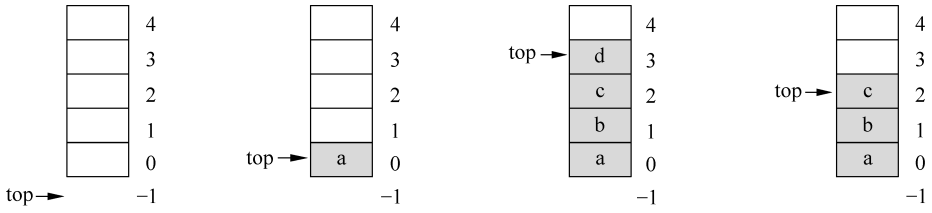


图 3.4　栈操作示意图

从中看到，初始时置栈顶指针 top＝－1。顺序栈的四要素如下：

(1) 栈空的条件为 top＝＝－1。

(2) 栈满(栈上溢出)的条件为 top＝＝capacity－1，这里采用动态扩展容量的方式，即栈满时将 data 数组的容量扩大两倍。

(3) 元素 e 进栈操作是先将栈顶指针 top 增 1，然后将元素 e 放在栈顶指针处。

(4) 出栈操作是先将栈顶指针 top 处的元素取出，然后将栈顶指针减 1。

说明：在进栈操作中，栈中元素始终是向一端伸展的。在采用 data 数组存放栈元素时，既可以将 data[0]端作为栈底，也可以将 data[capacity－1]端作为栈底，但不能将 data 数组的中间位置作为栈底，这里的顺序栈设计采用前一种方式。

顺序栈泛型类 SqStackClass＜E＞设计如下：

```
public class SqStackClass＜E＞                    //顺序栈泛型类
{  final int initcapacity=10;                     //顺序栈的初始容量(常量)
```

```
    private int capacity;                          //存放顺序栈的容量
    private E[] data;                              //存放顺序栈中的元素
    private int top;                               //存放栈顶指针
    public SqStackClass()                          //构造方法,实现 data 和 top 的初始化
    {   data = (E[])new Object[initcapacity];      //强制转换为 E 类型数组
        capacity=initcapacity;
        top=-1;
    }
    private void updatecapacity(int newcapacity)   //改变顺序栈的容量为 newcapacity
    {   E[] newdata = (E[])new Object[newcapacity];
        for(int i=0;i<top;i++)                     //复制原来的元素
            newdata[i]=data[i];
        capacity=newcapacity;                      //设置新容量
        data=newdata;                              //仍由 data 标识数组
    }
    //栈的基本运算算法
}
```

顺序栈的基本运算算法如下。

1) 判断栈是否为空：empty()

若栈顶指针 top 为-1,表示空栈。对应的算法如下：

```
public boolean empty()                             //判断栈是否为空
{ return top==-1;  }
```

2) 进栈：push(e)

元素进栈只能从栈顶进,不能从栈底或中间位置进,如图 3.5 所示。

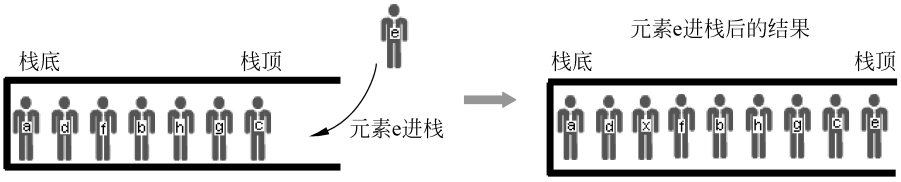


图 3.5　元素进栈示意图

在进栈中,当栈满时倍增容量,再将栈顶指针 top 增 1,然后在该位置上插入元素 e。对应的算法如下：

```
public void push(E e)                              //元素 e 进栈
{ if(top==capacity-1)                              //顺序栈空间满时倍增容量
      updatecapacity(2*(top+1));
  top++;                                           //栈顶指针增 1
  data[top]=e;
}
```

3) 出栈：pop()

元素出栈只能从栈顶出,不能从栈底或中间位置出,如图 3.6 所示。

在出栈中,当栈空时抛出异常,否则先将栈顶元素赋给 e,然后将栈顶指针 top 减 1,若栈容量大于初始容量并且实际元素个数仅为当前容量的 1/4,则当前容量减半,最后返回 e。

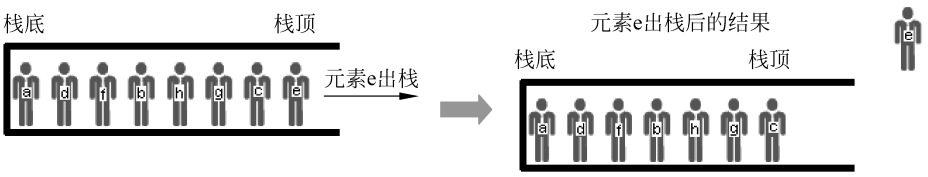


图 3.6　元素出栈示意图

对应的算法如下：

```
public E pop()                                          //出栈操作
{ if(empty())
      throw new IllegalArgumentException("栈空");
  E e=(E)data[top];
  top--;
  if(capacity>initcapacity && top+1==capacity/4)        //满足要求则容量减半
      updatecapacity(capacity/2);
  return e;
}
```

4）取栈顶元素：peek()

在栈不为空的条件下将栈顶元素赋给 e，不移动栈顶指针。对应的算法如下：


视频讲解

```
public E peek()                                         //取栈顶元素操作
{ if(empty())
      throw new IllegalArgumentException("栈空");
  return (E)data[top];
}
```

从以上看出，栈的各种基本运算算法的时间复杂度均为 $O(1)$。

3.1.3　顺序栈的应用算法设计示例

【例 3.4】 设计一个算法，利用顺序栈检查用户输入的表达式中的括号是否配对（假设表达式中可能含有小括号、中括号和大括号），并用相关数据进行测试。

解： 因为各种括号匹配过程遵循任何一个右括号与前面最靠近的同类左括号匹配的原则，所以采用一个栈来实现匹配过程。

用 str 存放含有各种括号的表达式，建立一个字符顺序栈 st，用 i 遍历 str，当遇到各种类型的左括号时进栈，当遇到右括号时，若栈空或者栈顶元素不是匹配的左括号，则操作失败，返回 false（如图 3.7 所示），否则退栈一次。当 str 遍历完毕，若栈 st 为空，返回 true，否则返回 false。

对应的完整程序如下：


视频讲解

```
public class Exam3_4
{ public static boolean isMatch(String str)             //判断算法
  {   int i=0;
      char e,x;
      SqStackClass<Character> st=new SqStackClass<Character>();   //建立一个顺序栈
      while(i<str.length())
```

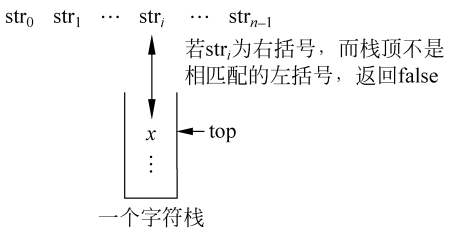


图 3.7　用一个栈判断 str 中的括号是否匹配

```
    {   e=str.charAt(i);
        if(e=='('||e=='['||e=='{')
            st.push(e);                              //将左括号进栈
        else
        {   if(e==')')
            {   if(st.empty()) return false;         //栈空则返回 false
                if(st.peek()!='(') return false;     //栈顶不是匹配的'('则返回 false
                st.pop();
            }
            if(e==']')
            {   if(st.empty()) return false;         //栈空则返回 false
                if(st.peek()!='[') return false;     //栈顶不是匹配的'['则返回 false
                st.pop();
            }
            if(e=='}')
            {   if(st.empty()) return false;         //栈空则返回 false
                if(st.peek()!='{') return false;     //栈顶不是匹配的'{'则返回 false
                st.pop();
            }
        }
        i++;                                         //继续遍历 str
    }
    if(st.empty()) return true;                      //栈空则返回 true
    else return false;                               //栈不空则返回 false
  }
  public static void main(String[] args)
  {   System.out.println("测试 1");
      String str="([)]";
      if(isMatch(str))   System.out.println(str+"中括号是匹配的");
      else System.out.println(str+"中括号不匹配");
      System.out.println("测试 2");
      str="([])";
      if(isMatch(str))   System.out.println(str+"中括号是匹配的");
      else System.out.println(str+"中括号不匹配");
      }
}
```

上述程序的执行结果如下：

```
测试 1
([)]中括号不匹配
```

测试 2
(□)中括号是匹配的

【例 3.5】 设计一个算法，利用顺序栈判断用户输入的字符串表达式是否为回文，并用相关数据进行测试。

解：用 str 存放表达式，其中含 n 个字符，建立一个字符型顺序栈 st，可以将 str 中的 n 个字符 $str_0, str_1, \cdots, str_{n-1}$ 依次进栈再连续出栈，得到反向序列 $str_{n-1}, \cdots, str_1, str_0$，若 str 与该反向序列相同，则是回文，否则不是回文。当然可以改为更高效的方法，若 str 的前半部分的反向序列与 str 的后半部分相同，则是回文，否则不是回文。判断过程如下：

(1) 用 i 从头开始遍历 str，将前半部分字符依次进栈。

(2) 若 n 为奇数，i 跳过中间的字符，i 继续遍历其他后半部分字符，每访问一个字符，则出栈一个字符，两者进行比较(如图 3.8 所示)，若不相等返回 false，当 str 遍历完毕时返回 true。

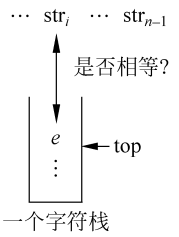


图 3.8　用一个栈判断 str 是否为回文

对应的完整程序如下：

```
public class Exam3_5
{  public static boolean isPalindrome(String str)          //例 3.5 的算法
   {  SqStackClass<Character> st=new SqStackClass();//建立一个顺序栈
      int n=str.length();
      int i=0;
      while(i<n/2)                                         //使 str 的前半部分字符进栈
      {  st.push(str.charAt(i));
         i++;                                              //继续遍历 str
      }
      if(n%2==1)                                           //n 为奇数时
         i++;                                              //跳过中间的字符
      while(i<n)                                           //遍历 str 的后半部分字符
      {  if(st.pop()!=str.charAt(i))
            return false;                                  //若 str[i]不等于出栈字符,则返回 false
         i++;
      }
      return true;                                         //是回文则返回 true
   }
   public static void main(String[] args)
   {  System.out.println("测试 1");
      String str="abcba";
      if(isPalindrome(str))    System.out.println(str+"是回文");
      else System.out.println(str+"不是回文");
      System.out.println("测试 2");
      str="1221";
      if(isPalindrome(str))    System.out.println(str+"是回文");
      else System.out.println(str+"不是回文");
   }
}
```

上述程序的结果如下：

测试 1
abcba 是回文
测试 2
1221 是回文

【例 3.6】 有 1～n 的 n 个元素，通过一个栈可以产生多种出栈序列，设计一个算法判断序列 b 是否为一个合法的出栈序列，并给出操作过程，要求用相关数据进行测试。

解：先建立一个整型顺序栈 st，将进栈序列 1～n 存放到数组 a 中，判断 a 序列通过一个栈算是否得到出栈序列 b，如图 3.9 所示。


视频讲解

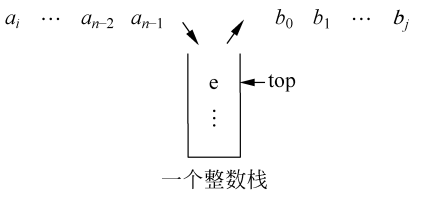


图 3.9 用一个栈判断 str 是否为合适的出栈序列

令 i、j 分别遍历 a、b 数组(初始值均为 0)，反复执行以下操作，直到 a 或者 b 数组遍历完为止：

(1) 若栈空，$a[i]$进栈，i++。

(2) 若栈不空，如果栈顶元素≠$b[j]$，将 $a[i]$进栈，i++。

(3) 否则出栈一个元素，j++。

上述过程简单地说就是当栈顶元素＝b 序列的当前元素时出栈一次，否则将 a 序列的当前元素进栈。

当该过程结束，再将栈中与 b 序列相同的元素依次出栈。由于 a 序列的每个元素最多进栈一次、出栈一次，而只有出栈时 j 才后移，所以如果 b 序列遍历完(j＝＝n)，说明 b 序列是合法的出栈序列，返回 true，否则不是合法的出栈序列，返回 false。其中用字符串 op 记录所有的栈操作。

当上述过程返回 true 时输出 op，否则输出提示信息。对应的完整程序如下：

```
import java.util.*;
public class Exam3_6
{ static String op="";
  static int cnt;
  public static boolean isSerial(int[] b)
  {  int i,j,n=b.length;
     Integer e;
     int[] a=new int[n];
     SqStackClass<Integer> st=new SqStackClass<Integer>();
     for (i=0;i<n;i++)
        a[i]=i+1;                                   //将 1～n 放入数组 a 中
     i=0; j=0;
     while (i<n && j<n)                             //a 和 b 均没有遍历完
     {  if (st.empty() || (st.peek()!=b[j]))        //栈空或者栈顶元素不是 b[j]
        {  st.push(a[i]);                           //a[i]进栈
           op+="  元素"+a[i]+"进栈\n";
```

```
            i++;
        }
        else                                    //否则出栈
        {   e=st.pop();
            op+="  元素"+e+"出栈\n";
            j++;
        }
    }
    while (!st.empty() && st.peek()==b[j])      //使栈中与 b 序列相同的元素出栈
    {   e=st.pop();
        j++;
    }
    if (j==n) return true;                      //是出栈序列时则返回 true
    else return false;                          //不是出栈序列时则返回 false
}
public static void solve(int[] b)               //求解算法
{   for (int i=0;i<b.length;i++)
        System.out.print(" "+b[i]);
    if (isSerial1(b))
    {   System.out.println("是合法的出栈序列");
        System.out.println(op);
    }
    else System.out.println("不是合法的出栈序列");
}
public static void main(String[] args)
{   System.out.println("测试 1");
    int[] b={1,3,2,4};
    solve(b);
    System.out.println("测试 2");
    int[] c={4,3,1,2};
    solve(c);
}
}
```

本程序的执行结果如下：

```
测试 1
 1 3 2 4 是合法的出栈序列
  元素 1 进栈
  元素 1 出栈
  元素 2 进栈
  元素 3 进栈
  元素 3 出栈
  元素 2 出栈
  元素 4 进栈
  元素 4 出栈
测试 2
 4 3 1 2 不是合法的出栈序列
```

上述过程可以进一步简化，同样用 i、j 分别遍历 a、b 数组(初始值均为 0)，在 a 数组没有遍历完时：

(1) 将 $a[i]$进栈,$i++$。

(2) 栈不空并且栈顶元素与 $b[j]$相同时循环,即出栈元素 e,$j++$(改为连续判断)。

在上述过程结束后,如果栈空,返回 true,否则返回 false。

对应的简化算法如下:

```
public static boolean isSerial1(int[] b)                    //简化的算法
{ int i,j,n=b.length;
  Integer e;
  int[] a=new int[n];
  SqStackClass<Integer> st=new SqStackClass<Integer>();
  for (i=0;i<n;i++)
      a[i]=i+1;                                             //将 1~n 放入数组 a 中
  i=0; j=0;
  while (i<n)                                               //a 没有遍历完
  {  st.push(a[i]);
     op+="  元素"+a[i]+"进栈\n";
     i++;
     while (!st.empty() && st.peek()==b[j])                 //b[j]与栈顶匹配的情况
     {  e=st.pop();
        op+="  元素"+e+"出栈\n";
        j++;
     }
  }
  return st.empty();                                        //栈空返回 true,否则返回 false
}
```

说明:上述两个算法中可以不使用 a 数组,直接将 $a[i]$的地方用$(i+1)$代替,但使用 a 数组时稍微修改一下,可以判断任何进栈序列 a 是否可以得到合法的出栈序列 b。

【例 3.7】 设有两个栈 S1 和 S2,它们都采用顺序栈存储,并且共享一个固定容量的存储区 $s[0..M-1]$,为了尽量利用空间,减少溢出的可能,请设计这两个栈的存储方式。

解:为了尽量利用空间,减少溢出的可能,可以采用栈顶相向、进栈元素迎面增长的存储方式,如图 3.10 所示。

视频讲解

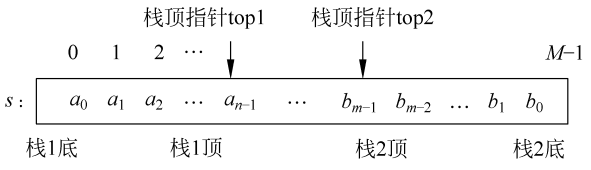


图 3.10 两个顺序栈的存储结构

两个栈的栈顶指针分别为 top1 和 top2。

栈 S1 空的条件是 top1=−1;栈 S1 满的条件是 top1=top2−1;元素 e 进栈 S1(栈不满时)的操作是 top1++;s[top1]=e;元素 e 出栈 S1(栈不空时)的操作是 e=s[top1];top1−−。

视频讲解

栈 S2 空的条件是 top2=M;栈 S2 满的条件是 top2=top1+1;元素 e 进栈 S2(栈不满时)的操作是 top2−−;s[top2]=e;元素 e 出栈 S2(栈不空时)的操作是 e=s[top2];top2++。

3.1.4　栈的链式存储结构及其基本运算算法的实现

采用链式存储的栈称为链栈，这里采用单链表实现。链栈的优点是不需要考虑栈满上溢出的情况。用带头结点的单链表 head 表示的链栈如图 3.11 所示，首结点是栈顶结点，尾结点是栈底结点，栈中元素自栈顶到栈底依次是 a_0、a_1、……、a_{n-1}。

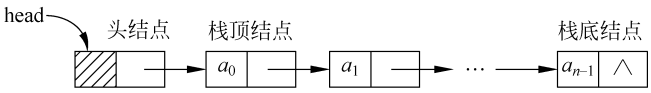


图 3.11　链栈的存储结构


视频讲解

从该链栈的存储结构看到，初始时只含有一个头结点 head 并置 head.next 为 null。链栈的四要素如下：

(1) 栈空的条件为 head.next==null。

(2) 由于只有在内存溢出才会出现栈满，通常不考虑这种情况。

(3) 元素 e 进栈操作是将包含该元素的结点 s 插入作为首结点。

(4) 出栈操作返回首结点值并且删除该结点。

和单链表一样，链栈中每个结点的类型 LinkNode<E>定义如下：

```
class LinkNode<E>                          //链栈结点泛型类
{ E data;
  LinkNode<E> next;
  public LinkNode()                        //构造方法
  {   next=null;   }
  public LinkNode(E d)                     //重载构造方法
  {   data=d;
      next=null;
  }
}
```

链栈类模板 LinkStackClass<E>的设计如下：

```
public class LinkStackClass<E>             //链栈泛型类
{ LinkNode<E> head;                        //存放头结点
  public LinkStackClass()                  //构造方法
  {   head=new LinkNode<E>();              //创建头结点
      head.next=null;                      //设置为空栈
  }

                                           //栈的基本运算算法
}
```

在链栈中实现栈的基本运算的算法如下。

1) 判断栈是否为空：empty()

若 head.next 为 null 表示空栈，即单链表中没有任何数据结点。对应的算法如下：

```
public boolean empty()                     //判断栈是否为空
{ return head.next==null;   }
```

2) 进栈：push(e)

新建包含数据元素 e 的结点 p，将 p 结点插入头结点之后。对应的算法如下：

```
public void push(E e)                          //元素 e 进栈
{ LinkNode<E> s=new LinkNode<E>(e);            //新建结点 s
  s.next=head.next;                            //将结点 s 插入表头
  head.next=s;
}
```

3) 出栈：pop()

在链栈空时抛出异常，否则将首结点的数据域赋给 e，然后将其删除。对应的算法如下：

```
public E pop()                                 //出栈操作
{ if (empty())
      throw new IllegalArgumentException("栈空");
  E e=(E)head.next.data;                       //取首结点值
  head.next=head.next.next;                    //删除原首结点
  return e;
}
```

4) 取栈顶元素：peek()

在链栈空时抛出异常，否则将首结点的数据域赋给 e，但不删除该结点。对应的算法如下：


视频讲解

```
public E peek()                                //取栈顶元素操作
{ if (empty())
      throw new IllegalArgumentException("栈空");
  E e=(E)head.next.data;                       //取首结点值
  return e;
}
```

3.1.5 链栈的应用算法设计示例

【例 3.8】 设计一个算法，利用栈的基本运算将一个整数链栈中的所有元素逆置。例如链栈 st 中的元素从栈底到栈顶为(1,2,3,4)，逆置后为(4,3,2,1)。

解：这里要求利用栈的基本运算来设计算法，所以不能直接采用单链表逆置方法。先出栈 st 中的所有元素并保存到一个数组 a 中，再将数组 a 中的所有元素依次进栈。对应的算法如下：


视频讲解

```
public static LinkStackClass<Integer> Reverse(LinkStackClass<Integer> st)
{ int[] a=new int[MaxSize];
  int i=0;
  while (!st.empty())                          //将出栈的元素放到数组 a 中
  {   a[i]=st.pop();
      i++;
  }
  for (int j=0;j<i;j++)                        //将数组 a 的所有元素进栈
      st.push(a[j]);
```

```
    return st;
}
```

【例 3.9】 假设用不带头结点的单链表作为链栈，请设计栈的 4 个基本运算算法。

解：设计原理与带头结点的单链表作为链栈相同，结点类 LinkNode＜E＞不改变。对应的链栈泛型类 LinkStackClass1＜E＞如下：

```
class LinkStackClass1＜E＞                          //链栈泛型类
{  LinkNode＜E＞ head;                              //存放首结点
   public LinkStackClass1()                         //构造方法
   {   head=null;    }                              //初始时将 head 设置为 null 表示链栈
                                                    //栈的基本运算算法
   public boolean empty()                           //判断栈是否为空
   {   return head==null;   }
   public void push(E e)                            //元素 e 进栈
   {   LinkNode＜E＞ s=new LinkNode＜E＞(e);         //新建结点 s
       s.next=head;                                 //将结点 s 插入表头
       head=s;
   }
   public E pop()                                   //出栈操作
   {   if (empty())
           throw new IllegalArgumentException("栈空");
       E e=(E)head.data;                            //取首结点值
       head=head.next;                              //删除原首结点
       return e;
   }
   public E peek()                                  //取栈顶元素操作
   {   if (empty())
           throw new IllegalArgumentException("栈空");
       E e=(E)head.data;                            //取首结点值
       return e;
   }
}
```

3.1.6　Java 中的栈容器——Stack＜E＞

Java 语言中提供了栈容器 Stack＜E＞，它是从 Vector＜E＞继承的，除了继承的许多方法外，作为栈的主要方法如下。

（1）boolean empty()：判断栈是否为空。

（2）int size()：返回栈中元素的个数。

（3）E push(E item)：把对象压入栈顶，即进栈操作。

（4）E pop()：移除栈顶对象，并作为此函数的值返回该对象，即出栈操作。

（5）E peek()：查看栈顶对象，但不从栈中移除它，即返回栈顶元素操作。

（6）int search(Object o)：返回元素 o 在栈中的位置，该位置从栈顶开始往下算，栈顶为 1。

（7）boolean contains(Object o)：如果栈中包含指定的元素 o，返回 true，否则返回 false。

视频讲解

例如,以下程序说明了 Stack<E>的基本使用方法。

```
import java.util.*;
public class Stackapp
{ public static void main(String[] args)
  {  Stack<String> st=new Stack<String>();                    //建立 String 栈对象 st
     st.push("a");                                            //进栈顺序: a,b,c,d,e
     st.push("b");
     st.push("c");
     st.push("d");
     st.push("e");
     System.out.println("empty():" + st.empty());            //输出:false
     System.out.println("peek():" + st.peek());              //输出:e
     System.out.println("search(Object o):" + st.search("a"));  //输出:5
     System.out.println("search(Object o):" + st.search("e"));  //输出:1
     System.out.println("search(Object o):" + st.search("no")); //输出:-1
     while(!st.isEmpty())                                     //出栈顺序: e,d,c,b,a
          System.out.println(st.pop() + " ");
     System.out.println("empty():" + st.empty());            //输出:true
  }
}
```

3.1.7 栈的综合应用

本节通过用栈求解简单表达式求值问题和用栈求解迷宫问题两个示例来说明栈的应用。

视频讲解

1. 用栈求解简单表达式求值问题

1) 问题描述

这里限定的简单表达式求值问题是,用户输入一个仅包含+、-、*、/、正整数和小括号的合法数学表达式,计算该表达式的运算结果。

2) 数据组织

简单算术表达式采用字符数组 exp 表示,其中只含有+、-、*、/、正整数和小括号。为了方便,假设该表达式是合法的数学表达式,例如 exp="1+2*(4+12)"。在设计相关算法时用到两个栈——一个运算符栈 opor 和一个运算数栈 opand,均采用 Java 的 Stack 容量,这两个栈对象定义如下:

```
Stack<Character> opor;                                       //运算符栈
Stack<Double> opand;                                         //运算数栈
```

3) 设计运算算法

在简单算术表达式中,运算符位于两个运算数中间的表达式称为**中缀表达式**,例如 1+2*3 就是一个中缀表达式,中缀表达式是最常用的一种表达式方式。对中缀表达式的运算一般遵循"从左到右计算,先乘除,后加减,有括号时先括号内,后括号外"的规则。因此中缀表达式不仅要依赖运算符的优先级,而且要处理括号。

所谓**后缀表达式**,就是运算符在运算数的后面,例如 1+2*3 的后缀表达式为 123*+。后缀表达式有这样的特点:已经考虑了运算符的优先级,不包含括号,只含运算数和运算符。对后缀表达式求值的过程是从左到右遍历后缀表达式,若遇到一个运算数,就将它进运

算数栈；若遇到一个运算符 op，就从运算数栈中连续出栈两个运算数，假设为 a 和 b，计算 b op a 之值，并将计算结果进运算数栈；对整个后缀表达式遍历结束后，栈顶元素就是计算结果。

假设给定的简单表达式 exp 是正确的，其求值过程分为两步，即先将表达式 exp 转换成后缀表达式 postexp，然后对该后缀表达式求值。

(1) 中缀表达式转换成后缀表达式。

中缀表达式 exp 和后缀表达式 postexp 均用 String 对象存放。设计求表达式值的类 ExpressClass 如下：

```
class ExpressClass                          //求表达式值的类
{ String exp;                               //存放中缀表达式
  String postexp="";                        //存放后缀表达式
  public void Setexp(String str)            //设置 exp
  {    exp=str;   }
  public String getpostexp()                //取 postexp
  {   return postexp;   }
  public void Trans()                       //将 exp 转换成后缀表达式
  public double getValue()                  //计算 postexp 的值
  { …}
}
```

将表达式 exp 转换成后缀表达式 postexp 用到运算符栈 opor，转换过程是遍历 exp，遇到数字符，将其直接放到 postexp 中；遇到'('，将其进栈到 opor；遇到')'，退栈 opor 运算符并放到 postexp 中，直到退栈的是'('为止(该左括号不放到 postexp 中)；遇到运算符 op_2，将其跟栈顶运算符 op_1 的优先级进行比较，只有当 op_2 的优先级高于 op_1 的优先级时才将 op_2 进栈，否则将栈中'('(如果有)以前的优先级等于或大于 op_2 的运算符均退栈并放到 postexp 中(如图 3.12 所示)，再将 op_2 进栈。其转换过程可以简化如下：

```
while (若 exp 未读完)
{ 从 exp 读取字符 ch;
  ch 为数字:将后续的所有数字依次存放到 postexp 中,并以字符'#'标识数值串结束;
  ch 为左括号'(':将'('进栈到 opor;
  ch 为右括号')':将 opor 栈中'('以前的运算符依次出栈并存放到 postexp 中,再将'('退栈;
  若 ch 的优先级高于栈顶运算符的优先级,则将 ch 进栈;否则退栈并存入 postexp 中,直到遇到'('
或者 ch 的优先级高于栈顶运算符优先级,再将 ch 进栈;
}
若字符串 exp 扫描完毕,则退栈 opor 的所有运算符并存放到 postexp 中。
```

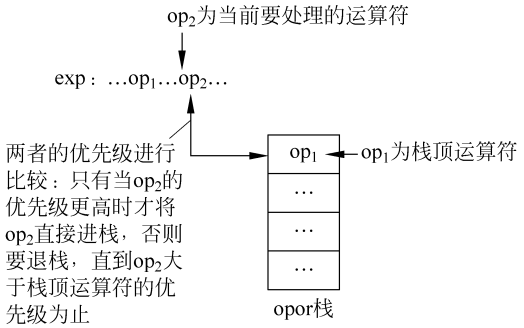


图 3.12　当前运算符的操作

在简单算术表达式中,只有'*'和'/'运算符的优先级高于'+'和'-'运算符的优先级,所以上述过程进一步简化为:

while(若 exp 未读完)
{ 从 exp 读取字符 ch;
  ch 为数字:将后续的所有数字依次存放到 postexp 中,并以字符'#'标志运算数串结束;
  ch 为左括号'(':将'('进栈到 opor;
  ch 为右括号')':将 opor 栈中'('以前的运算符依次出栈并存放到 postexp 中,再将'('退栈;
  ch 为'+'或'-':将 opor 栈中'('以前的所有运算符出栈并存放到 postexp 中,再将'+'或'-'进栈;
  ch 为'*'或'/':将 opor 栈中'('以前的'*'或'/'运算符出栈并存放到 postexp 中,再将'*'或'/'进栈;
}
若字符串 exp 扫描完毕,则退栈 opor 的所有运算符并存放到 postexp 中。

对于算术表达式"(56-20)/(4+2)",其转换成后缀表达式的过程如表 3.1 所示。

表 3.1 表达式"(56-20)/(4+2)"转换成后缀表达式的过程

ch	操 作	postexp	opor 栈
(	将'('进 opor 栈		(
56	将 56 存入 postexp 中,并插入一个字符'#'	56#	(
-	由于 opor 中'('以前没有字符,则直接将'-'进 opor 栈	56#	(-
20	将 20# 存入 postexp 中	56#20#	(-
)	将 opor 栈中'('以前的运算符依次出栈并存入 postexp,然后将'('出栈	56#20#-	
/	将'/'进 opor 栈	56#20#-	/
(	将'('进 opor 栈	56#20#-	/(
4	将 4# 存入 postexp	56#20#-4#	/(
+	由于 opor 中'('以前没有运算符,则直接将'+'进 opor 栈	56#20#-4#	/(+
2	将 2# 存入 postexp	56#20#-4#2#	/(+
)	将 opor 栈中'('以前的运算符依次出栈并存入 postexp,然后将'('出栈	56#20#-4#2#+	/
	exp 扫描完毕,将 opor 栈中的所有运算符依次出栈并存入 postexp,得到最后的后缀表达式	56#20#-4#2#+/	

根据上述原理得到的中缀表达式转后缀表达式的算法如下:

```
public void Trans()                          //将算术表达式 exp 转换成后缀表达式 postexp
{ Stack<Character> opor=new Stack<Character>();
  int i=0;                                   //i 作为 exp 的下标
  char ch,e;
  while(i<exp.length())                      //exp 表达式未扫描完时循环
  {  ch=exp.charAt(i);;
     if(ch=='(') opor.push(ch);              //判定为左括号,将左括号进栈
     else if(ch==')')                        //判定为右括号
     {  while(!opor.empty() && opor.peek()!='(')
        {                                    //将栈中最近'('之前的运算符退栈并存入 postexp
           e=opor.pop();
           postexp+=e;
        }
```

```
            opor.pop();                          //将'('退栈
        }
        else if(ch=='+'||ch=='-')                //判定为加号或减号
        {   while(!opor.empty() && opor.peek()!='(')
            {                                    //将栈中不低于 ch 的所有运算符退栈并存入 postexp
                e=opor.pop();
                postexp+=e;
            }
            opor.push(ch);                       //再将'+'或'-'进栈
        }
        else if(ch=='*'||ch=='/')                //判定为'*'或'/'
        {   while(!opor.empty() && opor.peek()!='(' && (opor.peek()=='*'||opor.peek()=='/'))
            {                                    //将栈中不低于 ch 的所有运算符退栈并存入 postexp
                e=opor.pop();
                postexp+=e;
            }
            opor.push(ch);                       //再将'*'或'/'进栈
        }
        else                                     //处理数字字符
        {   while(ch>='0' && ch<='9')            //判定为数字
            {   postexp+=ch;
                i++;                             //将连续的数字放入 postexp
                if (i<exp.length())
                    ch=exp.charAt(i);
                else
                    break;
            }
            i--;                                 //退一个字符
            postexp+='#';                        //用#标识一个数值串结束
        }
        i++;                                     //继续处理其他字符
    }
    while (!opor.empty())                        //此时 exp 扫描完毕,栈不空时循环
    {   e=opor.pop();                            //将栈中的所有运算符退栈并放入 postexp
        postexp+=e;
    }
}
```

(2) 后缀表达式求值。

在后缀表达式求值算法中用到运算数栈 opand,后缀表达式求值的过程如下:

```
while (若 postexp 未读完)
{ 从 postexp 读取字符 ch;
  ch 为'+':从 opand 栈出栈两个运算数 a 和 b,计算 c=b+a;将 c 进栈 opand;
  ch 为'-':从 opand 栈出栈两个运算数 a 和 b,计算 c=b-a;将 c 进栈 opand;
  ch 为'*':从 opand 栈出栈两个运算数 a 和 b,计算 c=b*a;将 c 进栈 opand;
  ch 为'/':从 opand 栈出栈两个运算数 a 和 b,若 a 不为零,计算 c=b/a;将 c 进栈 opand;
  ch 为数字字符:将连续的数字串转换成运算数 d,将 d 进栈 opand;
}
opand 栈中唯一的运算数即为表达式值。
```

后缀表达式“56＃20＃－4＃2＃＋/”的求值过程如表 3.2 所示。

表 3.2　后缀表达式“56＃20＃－4＃2＃＋/”的求值过程

ch 序列	说　　明	st 栈
56＃	遇到 56＃,将 56 进栈	56
20＃	遇到 20＃,将 20 进栈	56,20
－	遇到'－',出栈两次,将 56－20＝36 进栈	36
4＃	遇到 4＃,将 4 进栈	36,4
2＃	遇到 2＃,将 2 进栈	36,4,2
＋	遇到'＋',出栈两次,将 4＋2＝6 进栈	36,6
/	遇到'/',出栈两次,将 36/6＝6 进栈	6
	postexp 扫描完毕,算法结束,栈顶运算数 6 即为所求	

根据上述原理得到的计算后缀表达式值的算法如下：

```
public double getValue()                    //计算后缀表达式 postexp 的值
{  Stack<Double> opand=new Stack<Double>();
   double a,b,c,d;
   int i=0;
   char ch;
   while (i<postexp.length())               //遍历 postexp 串
   {  ch=postexp.charAt(i);                 //从后缀表达式中取一个字符 ch
      switch (ch)
      {
      case '+':                             //判定为'+'
         a=opand.pop();                     //退栈得到运算数 a
         b=opand.pop();                     //退栈得到运算数 b
         c=b+a;                             //计算 c
         opand.push(c);                     //将计算结果进栈
         break;
      case '-':                             //判定为'-'
         a=opand.pop();                     //退栈得到运算数 a
         b=opand.pop();                     //退栈得到运算数 b
         c=b-a;                             //计算 c
         opand.push(c);                     //将计算结果进栈
         break;
      case '*':                             //判定为'*'
         a=opand.pop();                     //退栈得到运算数 a
         b=opand.pop();                     //退栈得到运算数 b
         c=b*a;                             //计算 c
         opand.push(c);                     //将计算结果进栈
         break;
      case '/':                             //判定为'/'
         a=opand.pop();                     //退栈得到运算数 a
         b=opand.pop();                     //退栈得到运算数 b
         if (a!=0)
         {  c=b/a;                          //计算 c
            opand.push(c);                  //将计算结果进栈
         }
```

```
            else
                throw new ArithmeticException("运算错误:除零");
            break;
        default:                                    //处理数字字符
            d=0;                                    //将连续的数字字符转换成运算数存放到 d 中
            while(ch>='0' && ch<='9')               //判定为数字字符
            {   d=10*d+(ch-'0');
                i++;
                ch=postexp.charAt(i);
            }
            opand.push(d);                          //将数值 d 进栈
            break;
        }
        i++;                                        //继续处理其他字符
    }
    return opand.peek();                            //栈顶元素即为求值结果
}
```

4）设计主函数

设计以下包含主函数的类 Express 求简单算术表达式“(56－20)/(4＋2)”的值：

```
public class Express
{ public static void main(String[] args)
  {   double v;
      ExpressClass obj=new ExpressClass();
      String str="(56-20)/(4+2)";
      obj.Setexp(str);
      System.out.println("中缀表达式: "+str);
      obj.Trans();
      System.out.println("后缀表达式: "+obj.getpostexp());
      System.out.println("求值结果:    "+obj.getValue());
   }
}
```

5）运行结果

本程序的执行结果如下：

```
中缀表达式: (56-20)/(4+2)
后缀表达式: 56#20#-4#2#+/
求值结果:    6.0
```

上述简单表达式求值是先转换为后缀表达式，再对后缀表达式求值，也可以将两步合并起来，同样需要设置运算符栈 opor 和运算数栈 opand，合并后扫描表达式 exp：

(1)遇到数字字符将后续的所有数字字符合起来转换为运算数，进栈到 opand。
(2)遇到左括号，进栈到 opor。
(3)遇到右括号，将 opor 栈中'('以前的运算符 op 出栈并执行 op 计算，再将'('退栈。
(4)遇到运算符，只有优先级高于 opor 栈顶运算符才直接进栈 opor，否则出栈 op 并执行 op 计算。
若简单表达式扫描完毕，退栈 opor 的所有运算符并执行 op 计算。

其中执行 op 计算的过程是出栈 opand 两次得到运算数 a 和 b，执行 $c=b$ op a，然后 c 进栈

opand。最后 opand 栈的栈顶运算数就是简单表达式值。


视频讲解

2. 用栈求解迷宫问题

1) 问题描述

给定一个 $M\times N$ 的迷宫图,求一条从指定入口到出口的路径。假设迷宫图如图 3.13 所示(其中 $M=6,N=6$,含外围加上的一圈不可走的方块,这样做的目的是避免在查找时出界),迷宫由方块构成,空白方块表示可以走的通道,带阴影方块表示不可走的障碍物。要求所求路径必须是简单路径,即在求得的路径上不能重复出现同一空白方块,而且从每个方块出发只能走向上、下、左、右 4 个相邻的空白方块。

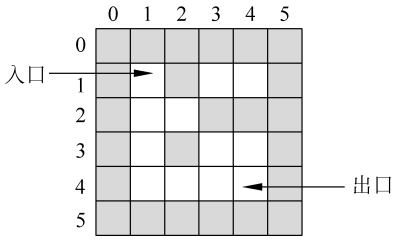


图 3.13 迷宫示意图

2) 数据组织

为了表示迷宫,设置一个数组 mg,其中每个元素表示一个方块的状态,为 0 时表示对应方块是通道,为 1 时表示对应方块不可走。为了使算法方便,在迷宫外围加了围墙。图 3.13 所示的迷宫对应的迷宫数组 mg(由于迷宫四周加了围墙,故数组 mg 的外围元素均为 1)如下:

```
int[][]mg= { {1,1,1,1,1,1},{1,0,1,0,0,1}, {1,0,0,1,1,1},
             {1,0,1,0,0,1}, {1,0,0,0,0,1},{1,1,1,1,1,1} };
```

另外,在算法中用到的栈 st 采用 Java 中的 Stack 容器表示,需要将搜索中的方块进栈,每个方块的 Box 类定义如下:

```
class Box                                   //方块类
{ int i;                                    //方块的行号
  int j;                                    //方块的列号
  int di;                                   //di 是下一个可走相邻方位的方位号
  public Box(int i1,int j1,int di1)         //构造方法
  {  i=i1;
     j=j1;
     di=di1;
  }
}
```

3) 设计运算算法

求迷宫问题就是在一个指定的迷宫中求出从入口到出口的路径。在求解时通常用的是“穷举求解”的方法,即从入口出发,顺某一方向向前试探,若能走通,则继续往前走;否则进入死胡同,沿原路退回,换一个方位再继续试探,直到所有可能的通路都试探完为止。

为了保证在任何位置上都能沿原路退回(称为回溯),需要用一个后进先出的栈来保存从入口到当前位置的路径。

对于迷宫中的每个方块,有上、下、左、右 4 个方块相邻,如图 3.14 所示。第 i 行第 j 列的方块的位置记为(i,j),规

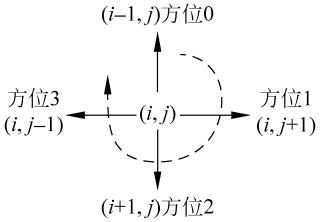


图 3.14 方位图

定上方方块为方位 0，并按顺时针方向递增编号。在试探过程中，假设按从方位 0 到方位 3 的方向查找下一个可走的方块，即从上方位开始按顺时针方向试探。为了便于回溯，对于可走的方块都要进栈，并试探它的下一个可走的方位，将这个可走的方位保存到栈中。

求解迷宫中从(xi,yi)到(xe,ye)路径的过程是先将入口进栈(其初始方位设置为－1)，在栈不空时循环，也就是取栈顶方块(不退栈)，若该方块是出口，则输出栈中所有方块，即为一条迷宫路径；否则找下一个可走的相邻方块，若不存在这样的方块，说明当前路径不可能走通(进入死胡同)，原路回退(回溯)，也就是恢复当前方块后退栈。若存在这样的方块，则将其方位保存到栈顶元素中，并将这个可走的相邻方块进栈(其初始方位设置为－1)。

求迷宫路径的回溯过程如图 3.15 所示，从前一方块找到一个可走相邻方块(即当前方块)后，再从当前方块找可走相邻方块，若没有这样的方块，说明当前方块不可能是从入口到出口路径上的一个方块，则从当前方块回溯到前一方块，继续从前一方块找另一个可走的相邻方块。

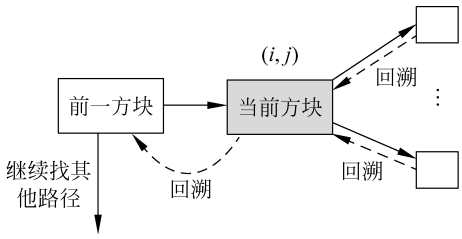


图 3.15　求迷宫路径的回溯过程

为了保证试探的可走相邻方块不是已走路径上的方块，如(i,j)已进栈，在试探(i＋1,j)的下一个可走方块时又试探到(i,j)，这样可能会引起死循环，为此在一个方块进栈后将对应的 mg 数组元素值改为－1(变为不可走的相邻方块)，当退栈时(表示该栈顶方块没有可走相邻方块)，将其恢复为 0。求图 3.13 中入口(1,1)到出口(4,4)迷宫路径的搜索过程如图 3.16 所示，图 3.16 中带“×”的方块是死胡同方块，走到这样的方块后需要回溯，找到出口后，栈中方块对应迷宫路径。

求解迷宫问题的 MazeClass 类定义如下：

```
class MazeClass                          //用栈求解一条迷宫路径类
{ final int MaxSize=20;
  int[][] mg;                            //迷宫数组
  int m,n;                               //迷宫行/列数
  public MazeClass(int m1,int n1)        //构造方法
  {   m=m1;
      n=n1;
      mg=new int[MaxSize][MaxSize];
  }
  public void Setmg(int[][] a)           //设置迷宫数组
  {   for(int i=0;i<m;i++)
          for(int j=0;j<n;j++)
              mg[i][j]=a[i][j];
  }
  boolean mgpath(int xi,int yi,int xe,int ye)   //求一条从(xi,yi)到(xe,ye)的迷宫路径
```

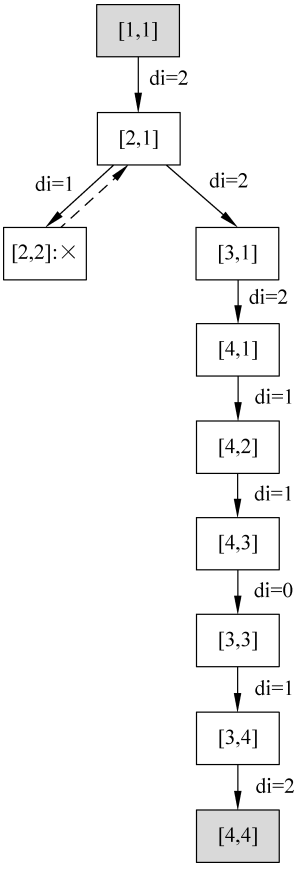


图 3.16　用栈求(1,1)到(4,4)迷宫路径的搜索过程

```
{   int i,j,di,i1=0,j1=0;
    boolean find;
    Box box,e;
    Stack<Box> st=new Stack<Box>(); //建立一个空栈
    st.push(new Box(xi,yi,-1));     //入口方块进栈
    mg[xi][yi]=-1;                  //为避免来回找相邻方块,将进栈的方块置为-1
    while(!st.empty())              //栈不空时循环
    {   box=st.peek();              //取栈顶方块,称为当前方块
        i=box.i; j=box.j; di=box.di;
        if(i==xe && j==ye)          //找到了出口,输出栈中所有方块构成一条路径
        {   int cnt=1;
            while(!st.empty())
            {   e=st.pop();
                System.out.print("["+e.i+","+e.j+"]  ");
                if(cnt%5==0)        //每行输出 5 个方块
                    System.out.println();
                cnt++;
            }
            System.out.println();
            return true;            //找到一条路径后返回 true
        }
```

```
            find=false;                          //否则继续找路径
            while(di<3 && !find)                 //找下一个相邻可走方块
            {   di++;                            //找下一个方位的相邻方块
                switch(di)
                {
                case 0:i1=i-1; j1=j; break;
                case 1:i1=i; j1=j+1; break;
                case 2:i1=i+1; j1=j; break;
                case 3:i1=i; j1=j-1; break;
                }
                if(mg[i1][j1]==0)
                    find=true;                   //找到下一个可走相邻方块(i1,j1)
            }
            if(find)                             //找到了下一个可走方块
            {   e=st.pop();
                e.di=di;
                st.push(e);                      //修改栈顶元素的 di 成员为 di
                st.push(new Box(i1,j1,-1));      //下一个可走方块进栈
                mg[i1][j1]=-1;                   //为避免来回找相邻方块,将进栈的方块置为-1
            }
            else                                 //没有路径可走,则退栈
            {   mg[i][j]=0;                      //恢复当前方块的迷宫值
                st.pop();                        //将栈顶方块退栈
            }
        }
        return false;                            //没有找到迷宫路径,返回 false
    }
}
```

其中 mgpath(int xi,int yi,int xe,int ye)方法用于求一条从入口(xi,yi)到出口(xe,ye)的迷宫路径,当成功找到出口后,栈 st 中从栈顶到栈底恰好是一条从出口到入口的迷宫逆路径,输出该迷宫逆路径,并返回 true,否则说明找不到迷宫路径,返回 false。

说明:从图 3.16 的过程看出,当找到一个出口后,可以进一步回溯找到从入口到出口的其他迷宫路径。如果只找一条迷宫路径,当栈顶方块(i,j)回退出栈时可以不置 mg[i][j]为 0,但如果找多条迷宫路径就必须置 mg[i][j]为 0 以恢复其迷宫值。

4) 设计主函数

设计以下包含主函数的 Maze 类,求如图 3.13 所示的迷宫图中从(1,1)到(4,4)的一条迷宫路径:

```
public class Maze
{ public static void main(String[] args)
  {  int[][] a= { {1,1,1,1,1,1},{1,0,1,0,0,1},
                  {1,0,0,1,1,1},{1,0,1,0,0,1},
                  {1,0,0,0,0,1},{1,1,1,1,1,1} };
     MazeClass mz=new MazeClass(6,6);    //M=6,N=6
     mz.Setmg(a);
     if (!mz.mgpath(1,1,4,4))            //(1,1)->(4,4)
        System.out.println("不存在迷宫路径");
  }
}
```

5) 运行结果

本程序的执行结果如下:

```
[4,4]  [3,4]  [3,3]  [4,3]  [4,2]
[4,1]  [3,1]  [2,1]  [1,1]
```

这里的迷宫路径是按从栈顶到栈底的顺序输出的,是一条迷宫逆路径,很容易转换为正向路径。该路径如图 3.17 所示(迷宫路径方块上的箭头指示路径上下一个方块的方位),显然这个解不是最优解,即不是最短路径。在使用队列求解时可以找出最短路径,这将在后面介绍。

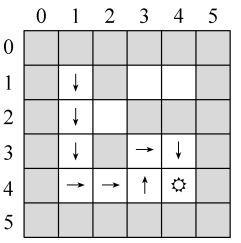


图 3.17 用栈找到的一条迷宫路径

3.2 队列

本节先介绍队列的定义,然后讨论队列存储结构和基本运算算法设计,最后通过迷宫问题的求解说明队列的应用。

3.2.1 队列的定义

同样先看一个示例,假设有一座独木桥,桥右侧有一群小兔子要过桥到桥左侧去,桥宽只能容纳一只兔子,那么这群小兔子怎么过桥呢? 结论是只能一只接一只地过桥,如图 3.18 所示。在这个例子中,独木桥就是一个队列,由于其宽度只能容纳一只兔子,所以不论有多少只兔子,它们只能是一只一只地排列着过桥,从而构成一种线性关系。再看看独木桥的主要操作,显然有上桥和下桥两种,上桥表示从桥右侧走到桥上,下桥表示离开桥。

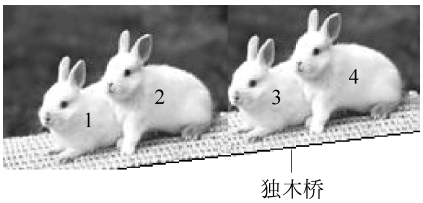


图 3.18 一群小兔子过独木桥

归纳起来,队列(简称为队)是一种操作受限的线性表,其限制为仅允许在表的一端进行插入,而在表的另一端进行删除。通常把进行插入的一端称为**队尾**(rear),把进行删除的一端称为**队头**或**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**出队**或**离队**,元素出队后,其直接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表的一端进行的,每个元素必然按照进入的次序出队,所以又把队列称为**先进先出表**。

抽象数据类型队列的定义如下:

```
ADT Queue
{
数据对象:
```

$D=\{a_i \mid 0\leqslant i\leqslant n-1, n\geqslant 0, a_i$ 为 E 类型$\}$
数据关系：
$R=\{r\}$
$r=\{<a_i, a_{i+1}> \mid a_i, a_{i+1}\in D,\ i=0,\cdots,n-2\}$
基本运算：
boolean empty()：判断队列是否为空，若队列为空，返回真，否则返回假。
void push(E e)：进队，将元素 e 进队作为队尾元素。
E pop()：出队，从队头出队一个元素。
E peek()：取队头，返回队头元素值而不出队。
}

【例 3.10】 若元素进队的顺序为 1234，能否得到 3142 的出队序列？

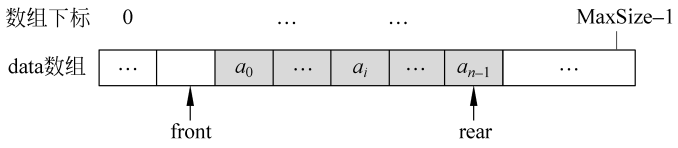
解：进队顺序为 1234，则出队顺序只能是 1234(先进先出)，所以不能得到 3142 的出队序列。

3.2.2　队列的顺序存储结构及其基本运算算法的实现

由于队列中元素的逻辑关系与线性表的相同，所以用户可以借鉴线性表的两种存储结构来存储队列。

当队列采用顺序存储结构存储时，分配一块连续的存储空间，用 data<E>数组来存放队列中的元素，另外设置两个指针，队头指针为 front(实际上是队头元素的前一个位置)，队尾指针为 rear(正好是队尾元素的位置)。

为了简单，这里使用固定容量的数组 data(容量为常量 MaxSize)，如图 3.19 所示，队列中从队头到队尾为 a_0、a_1、……、a_{n-1}。采用顺序存储结构的队列称为顺序队。



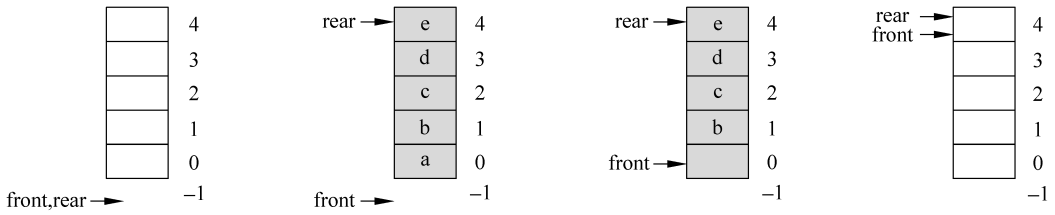

视频讲解

图 3.19　顺序队示意图

顺序队分为非循环队列和循环队列两种方式，这里先讨论非循环队列，并通过说明该类型队列的缺点引出循环队列。

1. 非循环队列

图 3.20 是一个非循环队列的动态变化示意图(MaxSize=5)。图 3.20(a)表示一个空队；图 3.20(b)表示进队 5 个元素后的状态；图 3.20(c)表示出队一次后的状态；图 3.20(d)表示再出队 4 次后的状态。



(a) 空队　(b) a、b、c、d、e元素进队　(c) 出队一次　(d) 出队4次

图 3.20　队列操作的示意图

从中可以看到,初始时置 front 和 rear 均为-1(front==rear)。非循环队列的四要素如下:

(1) 队空的条件为 front==rear,图 3.20(a)和(d)满足该条件。

(2) 队满(队上溢出)的条件为 rear==MaxSize-1(因为每个元素进队都让 rear 增 1,当 rear 到达最大下标时不能再增加),图 3.20(d)满足该条件。

(3) 元素 e 进队的操作是先将队尾指针 rear 增 1,然后将元素 e 放在该位置(进队的元素总是在尾部插入的)。

(4) 出队操作是先将队头指针 front 增 1,然后取出该位置的元素(出队的元素总是从头部出来的)。

说明:为什么让 front 指向队列中当前队头元素的前一个位置呢?因为在 front 增 1 后,该位置的元素已经出队(被删除)了。

非循环队列的泛型类 SqQueueClass<E>定义如下:

```
class SqQueueClass<E>                    //非循环队列的泛型类
{  final int MaxSize=100;                //假设容量为 100
   private E[] data;                     //存放队列中的元素
   private int front,rear;               //队头、队尾指针
   public SqQueueClass()                 //构造方法
   {  data = (E[])new Object[MaxSize];
      front=-1;
      rear=-1;
   }
   //队列的基本运算算法
}
```

非循环队列中实现队列的基本运算如下。

1) 判断队列是否为空:empty()

若满足 front==rear 条件,返回 true,否则返回 false。对应的算法如下:

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

2) 进队运算:push(e)

元素 e 进队只能从队尾插入,不能从队头或中间位置进队,仅仅改变队尾指针,如图 3.21 所示。

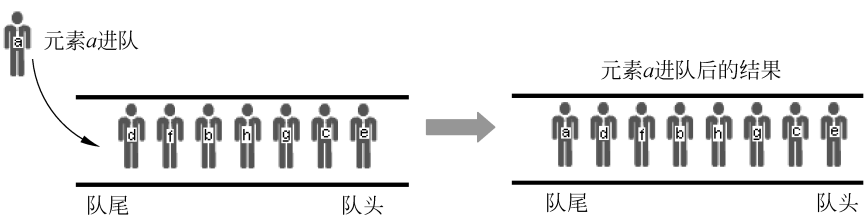


图 3.21 元素进队的示意图

在进队中,当队不满时,先将队尾指针 rear 增 1,然后将元素 e 放到该位置处,否则抛出异常。对应的算法如下:

```
public void push(E e)                          //元素 e 进队
{ if(rear==MaxSize-1)                          //队满
      throw new IllegalArgumentException("队满");
  rear++;
  data[rear]=e;
}
```

3）出队：pop()

元素出队只能从队头删除，不能从队尾或中间位置出队，仅仅改变队头指针，如图 3.22 所示。

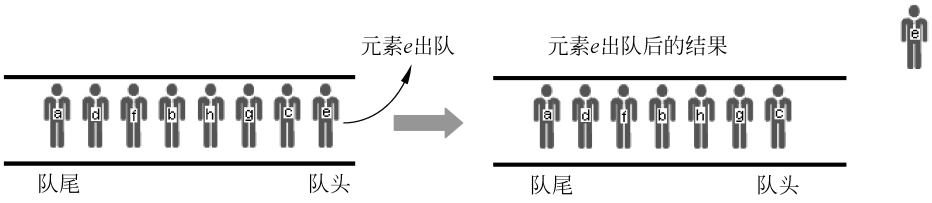


图 3.22　元素出队的示意图

在出队中，当队列不空时，将队头指针 front 增 1，并返回该位置的元素值，否则抛出异常。对应的算法如下：

```
public E pop()                                 //出队元素
{ if(empty())                                  //队空
      throw new IllegalArgumentException("队空");
  front++;
  return (E)data[front];
}
```

4）取队头元素：peek()

与出队类似，但不需要移动队头指针 front。对应的算法如下：

```
public E peek()                                //取队头元素
{ if(empty())                                  //队空
      throw new IllegalArgumentException("队空");
  return (E)data[front+1];
}
```

上述算法的时间复杂度均为 $O(1)$。

在上述非循环队列中，元素进队时队尾指针 rear 增 1，元素出队时队头指针 front 增 1，当进队 MaxSize 个元素后，满足设置的队满条件，即 rear==MaxSize-1 成立，此时即使出队若干元素，队满条件仍然成立（实际上队列中有空位置），这种队列中有空位置但仍然满足队满条件的上溢出称为**假溢出**。也就是说，非循环队列存在假溢出现象。

2. 循环队列

为了克服非循环队列的假溢出，充分使用数组中的存储空间，可以把 data 数组的前端和后端连接起来，形成一个循环数组，即把存储队列元素的表从逻辑上看成一个环，称为**循环队列**（也称为**环形队列**）。


视频讲解

循环队列首尾相连,当队尾指针 rear=MaxSize-1 时再前进一个位置就应该到达 0 位置,这可以利用数学上的求余运算(%)实现。

(1) 队首指针循环进 1:front=(front+1)%MaxSize

(2) 队尾指针循环进 1:rear=(rear+1)%MaxSize

循环队列的队头指针和队尾指针初始化时都置 0,即 front=rear=0。在进队元素和出队元素时,队头和队尾指针都循环前进一个位置。

那么循环队列的队满和队空的判断条件是什么呢?若设置队空条件是 rear==front,如果进队元素的速度快于出队元素的速度,队尾指针很快就赶上了队头指针,此时可以看出循环队列在队满时也满足 rear==front,所以这种设置无法区分队空和队满。

实际上循环队列的结构与非循环队列相同,也需要通过 front 和 rear 标识队列状态,一般是采用它们的相对值(即|front-rear|)实现的。若 data 数组的容量为 m,则队列的状态有 $m+1$ 种,分别是队空、队中有 1 个元素、队中有 2 个元素、……、队中有 m 个元素(队满)。front 和 rear 的取值范围均为 0~$m-1$,这样|front-rear|只有 m 个值,显然 $m+1$ 种状态不能直接用|front-rear|区分,因为必定有两种状态不能区分。为此让队列中最多只有 $m-1$ 个元素,这样队列恰好只有 m 种状态,就可以通过 front 和 rear 的相对值区分所有状态了。

在规定队列中最多只有 $m-1$ 个元素时,设置队空条件仍然是 rear==front。当队列有 $m-1$ 个元素时一定满足(rear+1)%MaxSize==front。

这样循环队列在初始时置 front=rear=0,其四要素如下:

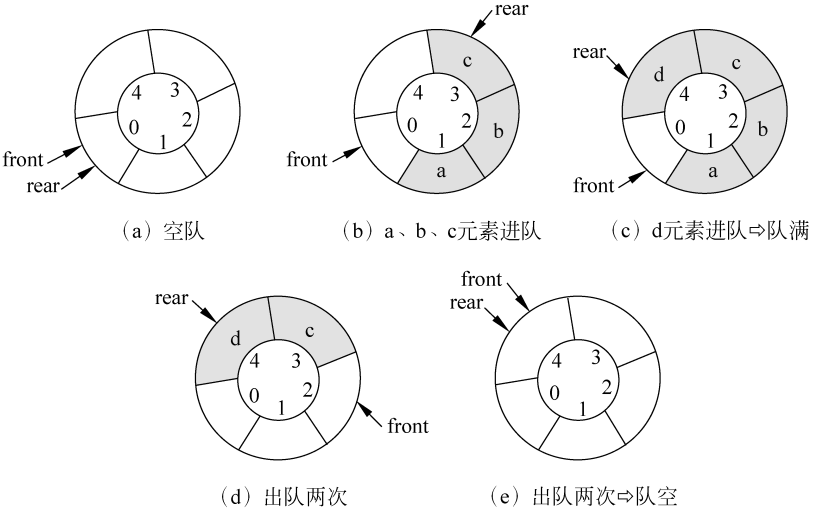
(1) 队空条件为 rear==front。

(2) 队满条件为(rear+1)%MaxSize==front(相当于试探进队一次,若 rear 达到 front,则认为队满了)。

(3) 元素 e 进队时,rear=(rear+1)%MaxSize,将元素 e 放置在该位置。

(4) 元素出队时,front=(front+1)%MaxSize,取出该位置的元素。

图 3.23 说明了循环队列的几种状态,这里假设 MaxSize 等于 5。图 3.23(a)为空队,此时 front=rear=0;图 3.23(b)中有 3 个元素,当进队元素 d 后队中有 4 个元素,此时满足队满的条件。



(a) 空队　(b) a、b、c元素进队　(c) d元素进队⇨队满

(d) 出队两次　(e) 出队两次⇨队空

图 3.23　循环队列进队和出队操作示意图

循环队列的泛型类 CSqQueueClass＜E＞定义如下：

```
class CSqQueueClass＜E＞                    //循环队列的泛型类
{ final int MaxSize=100;                    //假设容量为 100
  private E[] data;                         //存放队列中的元素
  private int front,rear;                   //队头、队尾指针
  public CSqQueueClass()                    //构造方法
  {  data = (E[])new Object[MaxSize];
     front=0;
     rear=0;
  }
  //队列的基本运算算法
}
```

在这样的循环队列中，实现队列的基本运算算法如下。

1）判断队列是否为空：empty()

若满足 front==rear 条件，返回 true，否则返回 false。对应的算法如下：

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

2）进队：push(e)

在队列不满时，先将队尾指针 rear 循环增 1，然后将元素 e 放到该位置处，否则抛出异常。对应的算法如下：

```
public void push(E e)                       //元素 e 进队
{ if((rear+1)%MaxSize==front)               //队满
      throw new IllegalArgumentException("队满");
  rear=(rear+1)%MaxSize;
  data[rear]=e;
}
```

3）出队：pop()

在队列不为空的条件下，将队头指针 front 循环增 1，并返回该位置的元素值，否则抛出异常。对应的算法如下：

```
public E pop()                              //出队元素
{ if(empty())                               //队空
      throw new IllegalArgumentException("队空");
  front=(front+1)%MaxSize;
  return (E)data[front];
}
```

4）取队头元素：peek()

与出队类似，但不需要移动队头指针 front。对应的算法如下：

```
public E peek()                             //取队头元素
{ if(empty())                               //队空
      throw new IllegalArgumentException("队空");
  return (E)data[(front+1)%MaxSize];
}
```

上述算法的时间复杂度均为 $O(1)$。

3.2.3 循环队列的应用算法设计示例

【例 3.11】 在循环队列泛型类 CSqQueueClass<E>中增加一个求元素个数的算法 size()。对于一个整数循环队列 qu,利用队列基本运算和 size()算法设计进队和出队第 k($k\geqslant 1$,队头元素的序号为 1)个元素的算法。

解:对于前面的循环队列,队头指针 front 指向队中队头元素的前一个位置,队尾指针 rear 指向队中的队尾元素,可以求出队中元素个数=(rear-front+MaxSize)%MaxSize。为此在循环队列泛型类 CSqQueueClass<E>中增加 size()算法如下:

视频讲解

```
public int size()                                   //返回队中元素个数
{ return((rear-front+MaxSize)%MaxSize);  }
```

在队列中并没有直接取 k($k\geqslant 1$)个元素的基本运算,进队第 k 个元素 e 的算法思路是出队前 $k-1$ 个元素,边出边进,再将元素 e 进队,将剩下的元素边出边进。该算法如下:

```
public static boolean pushk(CSqQueueClass<Integer> qu,int k,Integer e) //进队第 k 个元素 e
{ Integer x;
  int n=qu.size();
  if(k<1 || k>n+1)
     return false;                                  //参数 k 错误,返回 false
  if(k<=n)
  {  for(int i=1;i<=n;i++)                          //循环处理队中的所有元素
     {  if(i==k)
           qu.push(e);                              //将 e 元素进队到第 k 个位置
        x=qu.pop();                                 //出队元素 x
        qu.push(x);                                 //进队元素 x
     }
  }
  else qu.push(e);                                  //k=n+1 时直接进队 e
  return true;
}
```

出队第 k($k\geqslant 1$)个元素 e 的算法思路是出队前 $k-1$ 个元素,边出边进,出队第 k 个元素 e,e 不进队,将剩下的元素边出边进。该算法如下:

```
public static Integer popk(CSqQueueClass<Integer> qu,int k)   //出队第 k 个元素
{ Integer x,e=0;
  int n=qu.size();
  if(k<1 || k>n)                                    //参数 k 错误
     throw new IllegalArgumentException("参数错误");
  for(int i=1;i<=n;i++)                             //循环处理队中的所有元素
  {  x=qu.pop();                                    //出队元素 x
     if(i!=k)
        qu.push(x);                                 //将非第 k 个元素进队
     else e=x;                                      //取第 k 个出队的元素
  }
  return e;
}
```

【例 3.12】 对于循环队列来说，如果知道队头指针和队中元素个数，则可以计算出队尾指针，也就是说可以用队中元素个数代替队尾指针。设计出这种循环队列的判队空、进队、出队和取队头元素的算法。

解：本例的循环队列包含 data 数组、队头指针 front 和队中元素个数 count。初始时 front 和 count 均置为 0。队空条件为 count==0；队满的条件为 count==MaxSize；元素 e 进队的操作是先根据队头指针和元素个数求出队尾指针 rear1，将 rear1 循环增 1，然后将元素 e 放置在 rear1 处；出队操作是先将队头指针循环增 1，然后取出该位置的元素。设计对应的循环队列泛型类 CSqQueueClass1<E>如下：


视频讲解

```
class CSqQueueClass1<E>                    //本例的循环队列泛型类
{ final int MaxSize=100;                   //假设容量为 100
  private E[] data;                        //存放队列中的元素
  private int front;                       //队头指针
  private int count;                       //元素个数
  public CSqQueueClass1()                  //构造方法
  {  data = (E[])new Object[MaxSize];
     front=0;
     count=0;
  }
                                           //队列的基本运算算法
  public boolean empty()                   //判断队列是否为空
  {   return count==0;  }
  public void push(E e)                    //元素 e 进队
  {  int rear1;
     rear1=(front+count)%MaxSize;
     if(count==MaxSize)                    //队满
        throw new IllegalArgumentException("队满");
     rear1=(rear1+1) % MaxSize;
     data[rear1]=e;
     count++;                              //元素个数增 1
  }
  public E pop()                           //出队元素
  {  if(empty())                           //队空
        throw new IllegalArgumentException("队空");
     count--;                              //元素个数减 1
     front=(front+1)%MaxSize;
     return (E)data[front];
  }
  public E peek()                          //取队头元素
  {  if(empty())                           //队空
        throw new IllegalArgumentException("队空");
     return (E)data[(front+1)%MaxSize];
  }
}
```

说明：本例设计的循环队列中最多可保存 MaxSize 个元素。

从上述循环队列的设计看出，如果将 data 数组的容量改为可以扩展的，新建更大容量的数组 newdata 后，不能像顺序表、顺序栈那样简单地将 data 中的元素复制到 newdata 中，

需要按队列操作,将 data 中的所有元素出队后进队到 newdata 中,这里不再详述。

3.2.4 队列的链式存储结构及其基本运算算法的实现

队列的链式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首进行删除操作和在单链表的表尾进行插入操作,这里的单链表是不带头结点的,需要使用两个指针(即队首指针 front 和队尾指针 rear)来标识,front 指向队首结点,rear 指向队尾结点。用于存储队列的单链表简称为**链队**。


视频讲解

链队存储结构如图 3.24 所示,链队中存放元素的结点类型 LinkNode<E>定义如下:

```
class LinkNode<E>                          //链队结点泛型类
{ E data;
  LinkNode<E> next;
  public LinkNode()                        //构造方法
  {  next=null;  }
  public LinkNode(E d)                     //重载构造方法
  {  data=d;
     next=null;
  }
}
```

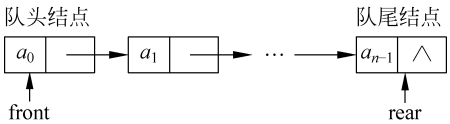
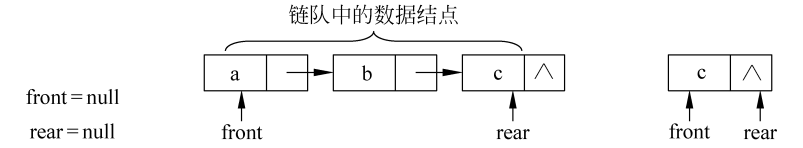


图 3.24 链队存储结构的示意图

设计链队泛型类 LinkQueueClass<E>如下:

```
public class LinkQueueClass<E>             //链队泛型类
{ LinkNode<E> front;                       //首结点指针
  LinkNode<E> rear;                        //尾结点指针
  public LinkQueueClass()                  //构造方法
  {  front=null;
     rear=null;
  }
  //队列的基本运算算法
}
```

图 3.25 说明了一个链队的动态变化过程。图 3.25(a)是链队的初始状态,图 3.25(b)是在链队中进队 3 个元素后的状态,图 3.25(c)是从链队中出队两个元素后的状态。



(a) 链队的初态　(b) 元素a、b、c进队　(c) 出队两个元素

图 3.25 一个链队的动态变化过程

从图 3.25 可以看到，初始时置 front＝rear＝null。链队的四要素如下：

（1）队空的条件为 fronr＝rear＝＝null，用户不妨仅以 front＝＝null 作为队空条件。

（2）由于只有在内存溢出时才出现队满，通常不考虑这样的情况。

（3）元素 e 进队的操作是在单链表尾部插入存放 e 的 s 结点，并让队尾指针指向它。

（4）出队操作是取出队首结点的 data 值，并将其从链队中删除。

对应队列的基本运算算法如下。

1）判断队列是否为空：empty()

若链队结点的 rear 域值为 null，表示队列为空，返回 true，否则返回 false。对应的算法如下：

```
public boolean empty()                          //判断队是否为空
{    return front==null;   }
```

2）进队：push(e)

创建存放 e 的结点 s。若原队列为空，则将 front 和 rear 指向 s 结点，否则将 s 结点链到单链表的末尾，并让 rear 指向它。对应的算法如下：

```
public void push(E e)                           //元素 e 进队
{ LinkNode<E> s=new LinkNode<E>(e);             //新建结点 s
  if(empty())                                   //原链队为空
     front=rear=s;
  else                                          //原链队不空
  {  rear.next=s;                               //将 s 结点链接到 rear 结点的后面
     rear=s;
  }
}
```

3）出队：pop()

若原队为空，抛出异常；若队中只有一个结点（此时 front 和 rear 都指向该结点），取首结点的 data 值赋给 e，并删除之，即置 front＝rear＝null；否则说明有多个结点，取首结点的 data 值赋给 e，并删除之。最后返回 e。对应的算法如下：

```
public E pop()                                  //出队操作
{ E e;
  if(empty())                                   //原链队不空
     throw new IllegalArgumentException("队空");
  if(front==rear)                               //原链队只有一个结点
  {  e=(E)front.data;                           //取首结点值
     front=rear=null;                           //置为空
  }
  else                                          //原链队有多个结点
  {  e=(E)front.data;                           //取首结点值
     front=front.next;                          //front 指向下一个结点
  }
  return e;
}
```

4) 取队头元素：peek()

与出队类似，但不需要删除首结点。对应的算法如下：

```
public E peek()                                    //取队头元素操作
{ if(empty())
      throw new IllegalArgumentException("队空");
  E e=(E)front.data;                               //取首结点值
  return e;
}
```

上述算法的时间复杂度均为 $O(1)$。

3.2.5 链队的应用算法设计示例

【例 3.13】 采用一个不带头结点只有一个尾结点指针 rear 的循环单链表存储队列，设计出这种链队的进队、出队、判队空和求队中元素个数的算法。

解：如图 3.26 所示，用只有尾结点指针 rear 的循环单链表作为队列存储结构，其中每个结点的类型为 LinkNode<E>(为 3.2.4 节的链队数据结点类型)。

视频讲解

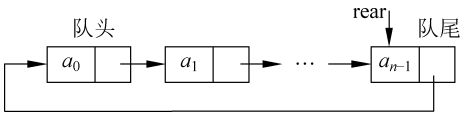


图 3.26 用只有尾结点指针的循环单链表作为队列存储结构

在这样的链队中，没有结点时队列为空，即 rear=null，进队在链表的表尾进行，出队在链表的表头进行。本例包含各种运算的链队类模板 LinkQueueClass1<E>定义如下：

```
public class LinkQueueClass1<E>                    //本例链队泛型类
{ LinkNode<E> rear;                                //尾结点指针
  public LinkQueueClass1()                         //构造方法
  {  rear=null;  }

                                                   //队列的基本运算算法
  public boolean empty()                           //判断队是否为空
  {  return rear==null;  }
  public void push(E e)                            //元素 e 进队
  {  LinkNode<E> s=new LinkNode(e);                //创建新结点 s
     if(rear==null)                                //原链队为空
     {  s.next=s;                                  //构成循环单链表
        rear=s;
     }
     else
     {  s.next=rear.next;                          //将 s 结点插入 rear 结点之后
        rear.next=s;
        rear=s;                                    //让 rear 指向 s 结点
     }
```

```
    }
    public E pop()                                    //出队操作
    {   E e;
        if(empty())                                   //原链队不空
            throw new IllegalArgumentException("队空");
        if(rear.next==rear)                           //原链队只有一个结点
        {   e=(E)rear.data;                           //取该结点值
            rear=null;                                //置为空队
        }
        else                                          //原链队有多个结点
        {   e=(E)rear.next.data;                      //取队头结点值
            rear.next=rear.next.next;                 //删除队头结点
        }
        return e;
    }
    public E peek()                                   //取队头元素操作
    {   E e;
        if(empty())                                   //原链队不空
            throw new IllegalArgumentException("队空");
        if(rear.next==rear)                           //原链队只有一个结点
            e=(E)rear.data;                           //取该结点值
        else                                          //原链队有多个结点
            e=(E)rear.next.data;                      //取队头结点值
        return e;
    }
}
```

3.2.6　Java 中的队列接口——Queue<E>

在 Java 语言中提供了队列接口 Queue<E>，提供了队列的修改运算，但不同于 Stack<E>，由于它是接口，所以在创建时需要指定元素类型，例如 LinkedList<E>。Queue<E>接口的主要方法如下。


视频讲解

(1) boolean isEmpty()：返回队列是否为空。

(2) int size()：队列中元素的个数。

(3) boolean add(E e)：将元素 e 进队(如果立即可行且不会违反容量限制)，在成功时返回 true，如果当前没有可用的空间，则抛出异常。

(4) boolean offer(E e)：将元素 e 进队(如果立即可行且不会违反容量限制)，当使用有容量限制的队列时，此方法通常要优于 add(E e)，后者可能无法插入元素，而只是抛出一个异常。

(5) E peek()：取队头元素，如果队列为空，则返回 null。

(6) E element()：取队头元素，它对 peek()方法进行简单封装，如果队头元素存在，则取出但并不删除，如果不存在则抛出异常。

(7) E poll()：出队，如果队列为空，则返回 null。

(8) E remove()：出队，直接删除队头的元素。

在 Queue<E>的使用中，主要操作是进队方法 offer()、出队方法 poll()、取队头元素方法 peek()，它们的时间复杂度均为 $O(1)$。

例如，以下程序说明了 Queue<E>的基本使用方法。

```
public class tmp
{ public static void main(String[] args)
  {  Queue<Integer> qu=new LinkedList<Integer>();
     qu.offer(1);
     qu.offer(2);
     qu.offer(3);
     while(!qu.isEmpty())
        System.out.print(qu.poll()+" ");      //输出:1 2 3
     System.out.println();
  }
}
```

3.2.7 队列的综合应用

本节通过用队列求解迷宫问题来讨论队列的应用。

视频讲解

1. 问题描述

参见 3.1.7 节。

2. 数据组织

用队列解决求迷宫路径问题，使用 Queue<Box>接口对象作为队列 qu，qu 队列中存放搜索的方块，其 Box 类型如下：

```
class Box                              //队列中的方块类
{ int i;                               //方块的行号
  int j;                               //方块的列号
  Box pre;                             //前驱方块
  public Box(int i1,int j1)            //构造方法
  {  i=i1;
     j=j1;
     pre=null;
  }
}
```

在设计相关算法时用到一个队列 qu，其定义如下：

```
Queue<Box>  qu;
```

3. 设计运算算法

在求迷宫路径中，从入口开始试探，将所有试探的方块均进队，如图 3.27 所示。假设当

前方块为(i,j)，对应的队列元素(Box 对象)为 b，然后找它所有的相邻方块，假设有 4 个相邻方块可走(实际上最多 3 个)，则这 4 个相邻可走的方块均进队，它们对应的队列元素分别为 $b_1 \sim b_4$，将方块(i,j)称为它们的前驱方块，它们称为方块(i,j)的后继方块，需要设置这样的关系，也就是置每个 b_i 的 pre 成员(即前驱方块)为 b，对应 b_i.pre=b。

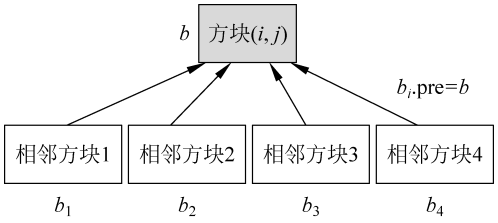


图 3.27　当前方块 b 和相邻方块

当找到出口后，此时队列 qu 中的方块可能有很多，不像用栈求解迷宫问题，此时栈中恰好保存一条迷宫路径上的所有方块。假设出口方块对应的队列元素为 b，通过 b.pre 向前查找前驱方块，直到入口方块为止，这些方块构成一条迷宫逆路径。

查找从(xi,yi)到(xe,ye)的路径的过程是首先建立(xi,yi)的 Box 对象 b，将 b 进队，在队列 qu 不为空时循环：出队一次，称该出队的方块 b 为当前方块。

(1) 如果 b 是出口，则从 b 出发查找一条迷宫逆路径，输出该路径后结束。

(2) 否则按顺时针方向一次性查找方块 b 的 4 个方位中的相邻可走方块，每个相邻可走方块均建立一个 Box 对象 b_1，置 b_1.pre=b，将 b_1 进 qu 队列。与用栈求解一样，一个方块进队后，其迷宫值置为−1，以避免回过来重复搜索。

如此操作，如果队列为空都没有找到出口，表示不存在迷宫路径，返回 false。

在图 3.13 所示的迷宫图中求入口(1,1)到出口(4,4)迷宫路径的搜索过程如图 3.28 所示，图 3.28 中带“×”的方块表示没有相邻可走方块，每个方块旁的数字表示搜索顺序，找到出口后，通过虚线(即 pre)找到一条迷宫逆路径。

用队列求解迷宫问题的 MazeClass 类定义如下：

```
class MazeClass                          //用队列求解一条迷宫路径类
{ final int MaxSize=20;
  int[][] mg;                            //迷宫数组
  int m,n;                               //迷宫行/列数
  Queue<Box> qu;                         //队列
  public MazeClass(int m1,int n1)        //构造方法
  {   m=m1;
      n=n1;
      mg=new int[MaxSize][MaxSize];
      qu=new LinkedList<Box>();          //创建一个空队 qu
  }
  public void Setmg(int[][] a)           //设置迷宫数组
  {   for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
```

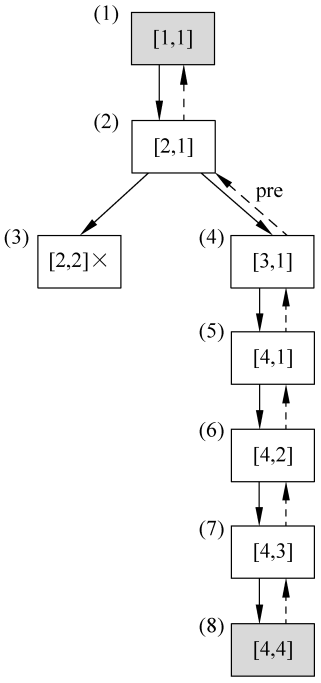


图 3.28　用队列求(1,1)到(4,4)迷宫路径的搜索过程

```
            mg[i][j]=a[i][j];
}
public void disppath(Box p)                    //从 p 出发找一条迷宫路径
{   int cnt=1;
    while(p!=null)                             //找到入口为止
    {   System.out.print("["+p.i+","+p.j+"]  ");
        if(cnt%5==0)                           //每行输出 5 个方块
            System.out.println();
        cnt++;
        p=p.pre;
    }
    System.out.println();
}
boolean mgpath(int xi,int yi,int xe,int ye)    //求(xi,yi)到(xe,ye)的一条迷宫路径
{   int i,j,i1=0,j1=0;
    Box b,b1;
    b=new Box(xi,yi);                          //建立入口结点 b
    qu.offer(b);                               //结点 b 进队
    mg[xi][yi]=-1;                             //进队方块的 mg 值置为-1
    while(!qu.isEmpty())                       //队不空时循环
    {   b=qu.poll();                           //出队一个方块 b
        if(b.i==xe && b.j==ye)                 //找到了出口,输出路径
        {   disppath(b);                       //从 b 出发回推导出迷宫路径并输出
            return true;                       //找到一条路径时返回 true
```

```
            }
            i=b.i; j=b.j;
            for(int di=0;di<4;di++)              //循环扫描每个方位,把每个可走的方块进队
            {   switch(di)
                {
                case 0:i1=i-1; j1=j;     break;
                case 1:i1=i;     j1=j+1; break;
                case 2:i1=i+1; j1=j;     break;
                case 3:i1=i;     j1=j-1; break;
                }
                if(mg[i1][j1]==0)                //找相邻可走方块
                {   b1=new Box(i1,j1);           //建立后继方块结点 b1
                    b1.pre=b;                    //设置其前驱方块为 b
                    qu.offer(b1);                //b1 进队
                    mg[i1][j1]=-1;               //将进队的方块置为-1
                }
            }
        }
        return false;                            //未找到任何路径时返回 false
    }
}
```

4. 设计主函数

设计以下包含主函数的 Maze 类,用于求图 3.13 所示的迷宫图中从(1,1)到(4,4)的一条迷宫路径:

```
public class Maze
{  public static void main(String[] args)
   {   int[][] a= { {1,1,1,1,1,1},{1,0,1,0,0,1},
                    {1,0,0,1,1,1},{1,0,1,0,0,1},
                    {1,0,0,0,0,1},{1,1,1,1,1,1} };
       MazeClass mz=new MazeClass(6,6);
       mz.Setmg(a);
       if(!mz.mgpath(1,1,4,4))
           System.out.println("不存在迷宫路径");
   }
}
```

5. 运行结果

本程序的执行结果如下:

```
[4,4]  [4,3]  [4,2]  [4,1]  [3,1]
[2,1]  [1,1]
```

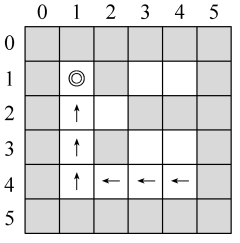


图 3.29　用队列求出的一条迷宫路径

该路径如图 3.29 所示,迷宫路径上方块的箭头表示其前驱方块的方位。显然这个解是最优解,也就是最短路径。至于为什么用栈求出的迷宫路径不一定是最短路径,而用队列求出的迷宫

路径一定是最短路径,这个问题会在第 8 章中说明。

* 3.2.8 双端队列

视频讲解

双端队列是在队列的基础上扩展而来的,其示意图如图 3.30 所示。双端队列与队列一样,元素的逻辑关系也是线性关系,但队列只能在一端进队,在另外一端出队,而双端队列可以在两端进行进队和出队操作,因此双端队列的使用更加灵活。

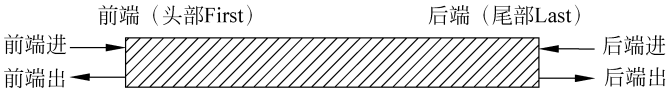


图 3.30 双端队列示意图

在 Java 中提供了双端队列接口 Deque<E>,它继承自 Queue<E>。Deque<E>接口的主要方法如下。

(1) boolean offer(E e):将元素 e 插入双端队列的尾部(与队列进队操作相同)。如果成功,返回 true,如果当前没有可用的空间,则返回 false。

(2) boolean offerFirst(E e):将元素 e 插入双端队列的开头。如果成功,返回 true,如果当前没有可用的空间,则返回 false。

(3) boolean offerLast(E e):将元素 e 插入双端队列的尾部(与队列进队操作相同)。如果成功,返回 true,如果当前没有可用的空间,则返回 false。与 offer(E e)相同。

(4) E poll():从双端队列中出队队头元素(与队列出队操作相同)。如果双端队列为空,则返回 null。

(5) E pollFirst():从双端队列中出队队头元素(与队列出队操作相同)。如果双端队列为空,则返回 null。与 poll()相同。

(6) E pollLast():从双端队列中出队队尾元素。如果双端队列为空,则返回 null。

(7) E peek():取双端队列的队头元素(与队列取队头元素操作相同)。如果双端队列为空,则返回 null。

(8) E peekFirst():取双端队列的队头元素(与队列取队头元素操作相同)。如果双端队列为空,则返回 null。用 peek()相同。

(9) E peekLast():取双端队列的队尾元素。如果双端队列为空,则返回 null。

从中可以看出,如果规定 Deque<E>只在一端进队,另外一端出队列,即只使用 offer(E e)/offerLast(E e)、poll()/pollFirst()、peek()/peekFirst()(后端进,前端出),或者 offerLast(E e)、pollFirst()、peekFirst()(前端进,后端出),此时 Deque<E>相当于队列。

如果规定 Deque<E>只在同端进队和出队,即只使用 offerFirst(E e)、pollFirst()、peekFirst()(前端进,前端出),或者 offerLast(E e)、pollLast()、peekLast()(后端进,后端出),此时 Deque<E>相当于栈。

3.2.9 优先队列

视频讲解

所谓优先队列,就是指定队列中元素的优先级,优先级越大越优先出队。普通队列中按进队的先后顺序出队,可以看成进队越早越优先。优先队列按照根的大小分为大根堆和小根堆,大根堆的元素越大越优先出队(即元素越大优先级也越大),小根堆的元素越小越优先

出队(即元素越小优先级越大)。

在 Java 中提供了优先队列类 PriorityQueue<E>,已实现的接口有 Queue<E>。创建优先队列的主要方式如下。

1) 默认方式

```
public PriorityQueue()
```

使用默认的初始容量(11)创建一个 PriorityQueue,并根据其自然顺序(例如整数按从小到大排列)对元素进行排序。

2) 指定优先队列的初始容量

```
public PriorityQueue(int initialCapacity)
```

使用指定的初始容量创建一个 PriorityQueue,并根据其自然顺序对元素进行排序。参数 initialCapacity 指定此优先队列的初始容量,如果 initialCapacity 小于 1,则会抛出 IllegalArgumentException 异常。

3) 指定比较器

```
public PriorityQueue(int initialCapacity,Comparator<? super E> comparator)
```

使用指定的初始容量创建一个 PriorityQueue,并根据指定的比较器对元素进行排序。参数 initialCapacity 指定此优先队列的初始容量,comparator 是用于对此优先队列进行排序的比较器,如果该参数为 null,则将使用元素的自然顺序排列。

PriorityQueue<E>类的主要方法如下。

(1) isEmpty():返回优先队列是否为空。

(2) int size():返回此优先队列中的元素数。

(3) void clear():从此优先队列中移除所有元素。

(4) boolean add(E e):将指定的元素插入此优先队列。

(5) boolean offer(E e):将指定的元素插入此优先队列。

(6) E poll():获取并移除此队列的头,如果此队列为空,则返回 null。

(7) E peek():获取但不移除此队列的头,如果此队列为空,则返回 null。

(8) boolean contains(Object o):如果此优先队列包含指定的元素,则返回 true。

说明:优先队列的最大优点是进队和出队运算的时间复杂度均为 $O(\log_2 n)$。

【例 3.14】 以下程序创建了 4 个优先队列,每个队列插入若干元素并出队所有元素。给出程序的执行结果。

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
class Stud                                          //学生类
{  private int id;                                  //学号
   private String name;                             //姓名
   public Stud(int id1,String na1)                  //构造方法
   {   id=id1;
       name=na1;
   }
```

```
    public int getid()                                          //id 属性
    {   return id;    }
    public String getname()                                     //name 属性
    {   return name;   }
}
public class Exam3_14
{ public static Comparator<Stud> idComparator                   //定义 idComparator
    = new Comparator<Stud>()
    {   public int compare(Stud o1,Stud o2)                     //用于创建 id 小根堆
        {
            return (int)(o1.getid()-o2.getid());
        }
    };
    public static Comparator<Stud> nameComparator               //定义 nameComparator
    = new Comparator<Stud>()
    {   public int compare(Stud o1,Stud o2)                     //用于创建 name 大根堆
        {
            return (int)(o2.getname().compareTo(o1.getname()));
        }
    };
    public static void main(String[] args)
    {   System.out.println("(1)创建 intminqu 队");
        PriorityQueue<Integer> intminqu = new PriorityQueue<>(7);
        System.out.println("    向 intminqu 队中依次插入 3,1,2,5,4");
        intminqu.offer(3);
        intminqu.offer(1);
        intminqu.offer(2);
        intminqu.offer(5);
        intminqu.offer(4);
        System.out.print("    intminqu 出队顺序: ");
        while(!intminqu.isEmpty())
            System.out.print(intminqu.poll()+" ");
        System.out.println();
        //*********************************************************
        System.out.println("(2)创建 intmaxqu 队");
        PriorityQueue<Integer> intmaxqu
        = new PriorityQueue<Integer>(10, new Comparator<Integer>()
        {   public int compare(Integer o1,Integer o2)
            {
                return o2-o1;
            }
        });
        System.out.println("    向 intmaxqu 队中依次插入 3,1,2,5,4");
        intmaxqu.offer(3);
        intmaxqu.offer(1);
        intmaxqu.offer(2);
        intmaxqu.offer(5);
        intmaxqu.offer(4);
        System.out.print("    intmaxqu 出队顺序: ");
        while(!intmaxqu.isEmpty())
            System.out.print(intmaxqu.poll()+" ");
```

```
        System.out.println();
        //*********************************************************
        System.out.println("(3)创建 stidminqu 队");
        PriorityQueue<Stud> stidminqu=new PriorityQueue<Stud>(10, idComparator);
        System.out.println("    向 stidminqu 队中依次插入 4 个对象");
        stidminqu.offer(new Stud(3,"Mary"));
        stidminqu.offer(new Stud(1,"David"));
        stidminqu.offer(new Stud(4,"Johm"));
        stidminqu.offer(new Stud(2,"Smith"));
        System.out.print("    stidminqu 出队顺序: ");
        Stud e;
        while(!stidminqu.isEmpty())
        {   e=stidminqu.poll();
            System.out.print("["+e.getid()+","+e.getname()+"]  ");
        }
        System.out.println();
        //*********************************************************
        System.out.println("(4)创建 stnamaxqu 队");
        PriorityQueue<Stud> stnamaxqu=new PriorityQueue<Stud>(10, nameComparator);
        System.out.println("    向 stnamaxqu 队中依次插入 4 个对象");
        stnamaxqu.offer(new Stud(3,"Mary"));
        stnamaxqu.offer(new Stud(1,"David"));
        stnamaxqu.offer(new Stud(4,"Johm"));
        stnamaxqu.offer(new Stud(2,"Smith"));
        System.out.print("    stnamaxqu 出队顺序: ");
        while(!stnamaxqu.isEmpty())
        {   e=stnamaxqu.poll();
            System.out.print("["+e.getid()+","+e.getname()+"]  ");
        }
        System.out.println();
    }
}
```

解：intminqu 和 intmaxqu 两个优先队列中的元素都是基本数据类型 int，intminqu 采用整数自然顺序，即小根堆，而 intmaxqu 为大根堆，需要重写 Integer 包装类的 Comparator 比较器。stidminqu 和 stnamaxqu 两个优先队列中的元素都是 Stud 类型，前者按 id 越小越优先出队（按 id 的小根堆），后者按 name 越大越优先出队（按 name 的大根堆）。上述程序的执行结果如下：

```
(1)创建 intminqu 队
   向 intminqu 队中依次插入 3,1,2,5,4
   intminqu 出队顺序: 1 2 3 4 5
(2)创建 intmaxqu 队
   向 intmaxqu 队中依次插入 3,1,2,5,4
   intmaxqu 出队顺序: 5 4 3 2 1
(3)创建 stidminqu 队
   向 stidminqu 队中依次插入 4 个对象
   stidminqu 出队顺序: [1,David]  [2,Smith]  [3,Mary]  [4,John]
(4)创建 stnamaxqu 队
   向 stnamaxqu 队中依次插入 4 个对象
   stnamaxqu 出队顺序: [2,Smith]  [3,Mary]  [4,John]  [1,David]
```

3.3 练习题

3.3.1 问答题

1. 简述线性表、栈和队列的异同。

2. 有 5 个元素,其进栈次序为 a、b、c、d、e,在各种可能的出栈次序中,以元素 c、d 最先出栈(即 c 第一个出栈且 d 第二个出栈)的次序有哪几个?

3. 一个栈用固定容量的数组 data 存放栈中元素,假设该容量为 n,则最多只能进栈 n 次,这句话正确吗?

4. 假设以 I 和 O 分别表示进栈和出栈操作,则初态和终态为栈空的进栈和出栈的操作序列可以表示为仅由 I 和 O 组成的序列,称可以实现的栈操作序列为合法序列(例如 IIOO 为合法序列,IOOI 为非法序列)。试给出区分给定序列为合法序列或非法序列的一般准则。

5. 若采用数组 data[1..m]存放栈元素,回答以下问题:

(1) 只能以 data[1]端作为栈底吗?

(2) 为什么不能以 data 数组的中间位置作为栈底?

6. 解决顺序队列的“假溢出”现象有哪些方法?

7. 假设循环队列的元素存储空间为 data[0..m-1],队头指针 f 指向队头元素,队尾指针 r 指向队尾元素的下一个位置(例如 data[0..5],队头元素为 data[2],则 front=2,队尾元素为 data[3],则 rear=4),则在少用一个元素空间的前提下表示队空和队满的条件各是什么?

8. 在算法设计中有时需要保存一系列临时数据元素,如果先保存的后处理,应该采用什么数据结构存放这些元素?如果先保存的先处理,应该采用什么数据结构存放这些元素?

9. 栈和队列的特性正好相反,栈先进后出,队列先进先出,所以任何一个问题的求解要么采用栈,要么采用队列,不可能用两种数据结构来求解同一个问题。这句话正确吗?

10. 简述普通队列和优先队列的差别。

3.3.2 算法设计题

1. 假设以 I 和 O 分别表示进栈和出栈操作,栈的初态和终栈均为空,进栈和出栈的操作序列 str 可表示为仅由 I 和 O 组成的字符串。设计一个算法判定 str 是否合法。

2. 给定一个字符串 str,设计一个算法采用顺序栈判断 str 是否为形如“序列 1@序列 2”的合法字符串,其中序列 2 是序列 1 的逆序,在 str 中恰好只有一个@字符。

3. 设计一个算法,利用一个整数栈将一个整数队列中的所有元素倒过来,队头变队尾,队尾变队头。

4. 有一个整数数组 a,设计一个算法将所有偶数位的元素移动到所有奇数位的元素的前面,要求它们的相对次序不改变。例如,a={1,2,3,4,5,6,7,8},移动后 a={2,4,6,8,1,3,5,7}。

5. 设计一个循环队列,用 data[0..MaxSize-1]存放队列元素,用 front 和 rear 分别作为队头和队尾指针,另外用一个标志 tag 标识队列可能空(false)或可能满(true),这样加上

front==rear 可以作为队空或队满的条件。要求设计队列的相关基本运算算法。

6. 用两个整数栈实现一个整数队列。

7. 用两个整数队列实现一个整数栈。

3.4　实验题

3.4.1　上机实验题

1. 编写一个程序求 n 个不同元素通过一个栈的出栈序列的个数，输出 n 为 1～7 的结果。

2. 用一个一维数组 S(设固定容量 MaxSize 为 5，元素类型为 int)作为两个栈的共享空间。编写一个程序，采用例 3.7 的共享栈方法设计其判栈空运算 empty(i)、栈满运算 full()、进栈运算 push(i,x)和出栈运算 pop(i)，其中 i 为 1 或 2，用于表示栈号，x 为进栈元素。采用相关数据进行测试。

3. 改进 3.1.7 节中的用栈求解迷宫问题的算法，累计图 3.13 所示的迷宫的路径条数，并输出所有迷宫路径。

4. 用循环队列求解约瑟夫问题：设有 n 个人站成一圈，其编号为从 1 到 n。从编号为 1 的人开始按顺时针方向 1,2,…循环报数，数到 m 的人出列，然后从出列者的下一个人重新开始报数，数到 m 的人又出列，如此重复进行，直到 n 个人都出列为止。要求输出这 n 个人的出列顺序，并用相关数据进行测试。

3.4.2　在线编程题

1. HDU1237——简单计算器

时间限制：2000ms；空间限制：65 536KB。

问题描述：读入一个只包含＋、－、*、/的非负整数运算的表达式，计算该表达式的值。

输入格式：测试输入包含若干测试用例，每个测试用例占一行，每行不超过 200 个字符，整数和运算符之间用一个空格分隔；没有非法表达式；当一行中只有 0 时输入结束，相应的结果不要输出。

输出格式：对每个测试用例输出一行，即该表达式的值，精确到小数点后两位。

输入样例：

```
1 + 2
4 + 2 * 5 - 7 / 11
0
```

输出样例：

```
3.00
13.36
```

2. POJ1363——铁轨问题

时间限制：1000ms；空间限制：65 536KB。

问题描述：A 市有一个著名的火车站，那里的山非常多。该站建于 20 世纪，不幸的是该火车站是一个死胡同，并且只有一条铁轨，如图 3.31 所示。

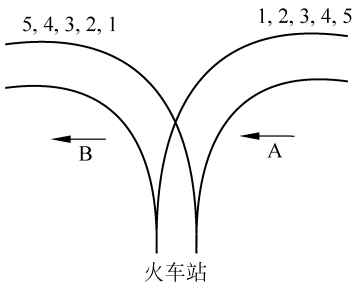


图 3.31　铁轨问题示意图

当地的传统是从 A 方向到达的每列火车都继续沿 B 方向行驶，需要以某种方式进行车厢重组。假设从 A 方向到达的火车有 n($n \leqslant 1000$)个车厢，按照递增的顺序 1，2，…，n 编号。火车站负责人必须知道是否可以通过重组得到 B 方向的车厢序列。

输入格式：输入由若干行块组成；除了最后一个之外，每个块描述了一个列车以及可能更多的车厢重组要求；在每个块的第一行中为上述的整数 n，下一行是 1，2，…，n 的车厢重组序列；每个块的最后一行仅包含 0；最后一个块只包含一行 0。

输出格式：输出包含与输入中具有车厢重组序列对应的行。如果可以得到车厢对应的重组序列，输出一行“Yes”，否则输出一行“No”。此外，在输入的每个块之后有一个空行。

输入样例：

```
5
1 2 3 4 5
5 4 1 2 3
0
6
6 5 4 3 2 1
0
0
```

输出样例：

```
Yes
No

Yes
```

3. HDU1702——ACboy 请求帮助问题

时间限制：1000ms；空间限制：32 768KB。

问题描述：ACboy 被怪物绑架了！他非常想念母亲并且十分恐慌。ACboy 呆的房间非常黑暗，他非常可怜。作为一个聪明的 ACMer，你想让 ACboy 从怪物的迷宫中走出来。但是当你到达迷宫的大门时，怪物说：“我听说你很聪明，但如果不能解决我的问题，你会和 ACboy 一起死。”

墙上显示了怪物的问题：每个问题的第一行是一个整数 N(命令数)和一个单词“FIFO”或“FILO”(你很高兴，因为你知道“FIFO”代表“先进先出”，“FILO”的意思是“先进后出”)。以下 N 行，每行是“IN M”或“OUT”(M 代表一个整数)。问题的答案是门的密码，所以如果你想拯救 ACboy，请仔细回答问题。

输入格式：输入包含多个测试用例。第一行有一个整数，表示测试用例的数量，并且每

个子问题的输入像前面描述的方式。

输出格式：对于每个命令“OUT”，应根据单词是“FIFO”还是“FILO”输出一个整数，如果没有任何整数，则输出单词“None”。

输入样例：

```
4
4 FIFO
IN 1
IN 2
OUT
OUT
4 FILO
IN 1
IN 2
OUT
OUT
5 FIFO
IN 1
IN 2
OUT
OUT
OUT
5 FILO
IN 1
IN 2
OUT
IN 3
OUT
```

输出样例：

```
1
2
2
1
1
2
None
2
3
```

4. POJ2259——团队队列问题

时间限制：2000ms；空间限制：65 536KB。

问题描述：队列和优先队列是大多数计算机科学家都知道的数据结构，然而团队队列(Team Queue)并不那么出名，尽管它经常出现在人们的日常生活中。在团队队列中，每个队员都属于一个团队，如果一个队员进入队列，他首先从头到尾搜索队列，以检查其队友(同一团队的队员)是否已经在队列中，如果在，他就会进到该队列的后面；如果不在，他进入当

前尾部的队列并成为该队列的一个队员(运气不好)。和在普通队列中一样,队员按照他们在队列中出现的顺序从头到尾进行处理。

请编写一个模拟此团队队列的程序。

输入格式:输入将包含一个或多个测试用例。每个测试用例以团队数量 t($1 \leqslant t \leqslant 1000$)开始,然后是团队描述,每个团队描述由该团队的队员数和队员本身组成,每个队员是一个 0～999 999 的整数,一个团队最多可包含 1000 个队员,最后是一个命令列表,有以下 3 种不同的命令。

(1) ENQUEUE x:将队员 x 进入团队队列。

(2) DEQUEUE:处理第一个元素并将其从队列中删除。

(3) STOP:测试用例结束。

当输入的 t 为 0 时终止。注意,测试用例最多可包含 200 000(20 万)个命令,因此团队队列的实现应该是高效的,即队员进队和出队应该只占常量时间。

输出格式:对于每个测试用例,首先输出一行"Scenario $\sharp k$",其中 k 是测试用例的编号,然后对于每个 DEQUEUE 命令,输出一行指出出队的队员。在每个测试用例后输出一个空行,包括最后一个测试用例。

输入样例:

```
2
3 101 102 103
3 201 202 203
ENQUEUE 101
ENQUEUE 201
ENQUEUE 102
ENQUEUE 202
ENQUEUE 103
ENQUEUE 203
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
STOP
2
5 259001 259002 259003 259004 259005
6 260001 260002 260003 260004 260005 260006
ENQUEUE 259001
ENQUEUE 260001
ENQUEUE 259002
ENQUEUE 259003
ENQUEUE 259004
ENQUEUE 259005
DEQUEUE
DEQUEUE
ENQUEUE 260002
ENQUEUE 260003
```

```
DEQUEUE
DEQUEUE
DEQUEUE
DEQUEUE
STOP
0
```

输出样例：

```
Scenario #1
101
102
103
201
202
203

Scenario #2
259001
259002
259003
259004
259005
260001
```

5. POJ3984——迷宫问题

时间限制：1000ms；空间限制：65 536KB。

问题描述：定义一个二维数组如下。

```
int maze[5][5] = {
  0, 1, 0, 0, 0,
  0, 1, 0, 1, 0,
  0, 0, 0, 0, 0,
  0, 1, 1, 1, 0,
  0, 0, 0, 1, 0,
}
```

它表示一个迷宫，其中的 1 表示墙壁，0 表示可以走的路，注意只能横着走或竖着走，不能斜着走，要求编程找出从左上角到右下角的最短路线。

输入格式：一个 5×5 的二维数组，表示一个迷宫。数据保证有唯一解。

输出格式：从左上角到右下角的最短路径，格式如样例所示。

输入样例：

```
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

输出样例：

```
(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(3, 4)
(4, 4)
```

6. POJ1028——Web 导航

时间限制：1000ms；空间限制：10 000KB。

问题描述：标准 Web 浏览器包含有在最近访问过的页面之间前后移动的功能。实现此功能的一种方法是使用两个栈来跟踪可以通过前后移动到达的页面。在此问题中，系统会要求编程实现此功能。

程序需要支持以下命令。

(1) BACK：将当前页面进入前向栈作为栈顶，从后向栈中出栈栈顶的页面，使其成为新的当前页面。如果后向栈为空，则忽略该命令。

(2) FORWARD：将当前页面进入后向栈作为栈顶，从前向栈中出栈栈顶的页面，使其成为新的当前页面。如果前向栈为空，则忽略该命令。

(3) VISIT：将当前页面进入后向栈作为栈顶，并将 URL 指定为新的当前页面。清空前向栈。

(4) QUIT：退出浏览器。

假设浏览器最初加载的页面的 URL 为“http://www.acm.org/”。

输入格式：输入是一系列命令；命令关键字 BACK、FORWARD、VISIT 和 QUIT 都是大写的；URL 中没有空格，最多包含 70 个字符；可以假设任何实例在任何时候每个栈中的元素不会超过 100；由 QUIT 命令指示输入结束。

输出格式：对于除 QUIT 之外的每个命令，如果不忽略该命令，则在执行命令后打印当前页面的 URL，否则打印“Ignored”。每个命令输出一行，QUIT 命令没有任何输出。

输入样例：

```
VISIT http://acm.ashland.edu/
VISIT http://acm.baylor.edu/acmicpc/
BACK
BACK
BACK
FORWARD
VISIT http://www.ibm.com/
BACK
BACK
FORWARD
FORWARD
```

```
FORWARD
QUIT
```

输出样例：

```
http://acm.ashland.edu/
http://acm.baylor.edu/acmicpc/
http://acm.ashland.edu/
http://www.acm.org/
Ignored
http://acm.ashland.edu/
http://www.ibm.com/
http://acm.ashland.edu/
http://www.acm.org/
http://acm.ashland.edu/
http://www.ibm.com/
Ignored
```

7. HDU1873——看病要排队

时间限制：3000ms；空间限制：32 768KB。（HDU1509 与之类似）

问题描述：看病要排队是地球人都知道的常识，不过细心的 0068 经过观察，他发现医院里的排队还是有讲究的。0068 所去的医院有 3 个医生同时在看病，而看病的人病情有轻重，不能根据简单的先来先服务的原则，所以医院对每种病情规定了 10 种不同的优先级，级别为 10 的优先权最高，级别为 1 的优先权最低。医生在看病时，会在他的病人里面选择一个优先权最高的进行诊治，如果遇到两个优先权一样的病人，则选择最早来排队的病人。

请编程帮助医院模拟这个看病过程。

输入格式：输入数据包含多组测试，请处理到文件结束。每组数据的第一行有一个正整数 n（$0<n<2000$），表示发生事件的数目，接下来有 n 行，分别表示发生的事件。一共有以下两种事件。

（1）IN A B：表示有一个拥有优先级 B 的病人要求医生 A 诊治（$0<A\leqslant 3$，$0<B\leqslant 10$）。

（2）OUT A：表示医生 A 进行了一次诊治，诊治完毕后病人出院（$0<A\leqslant 3$）。

输出格式：对于每个“OUT A”事件，请在一行里面输出被诊治人的编号 ID。如果该事件中无病人需要诊治，则输出“EMPTY”。诊治人的编号 ID 的定义为：在一组测试中，“IN A B”事件发生第 k 次时进来的病人的 ID 即为 k。ID 从 1 开始编号。

输入样例：

```
7
IN 1 1
IN 1 2
OUT 1
OUT 2
IN 2 1
OUT 2
OUT 1
```

```
2
IN 1 1
OUT 1
```

输出样例：

```
2
EMPTY
3
1
1
```