第3章

栈和队列

CHAPTER 3

栈和队列是两种特殊的线性结构，从元素之间的逻辑关系上看，它们都属于线性结构，其特殊性在于：插入和删除操作会限制在表的一端进行。因此，栈和队列被称为操作受限的线性表。在实际生活中，栈和队列被广泛应用于软件开发过程。本章主要介绍栈和队列的基本概念、存储结构、基本运算及典型应用。

本章主要内容

- 栈
- 栈与递归
- 队列
- 双端队列

视频讲解

3.1 栈

栈是一种只能在表的一端进行插入和删除操作的线性表,但其应用非常广泛,如算术表达式求值、括号匹配和递归算法等。本节主要介绍栈的定义、栈的存储结构及应用。

视频讲解

3.1.1 栈的基本概念

栈(stack),也称为堆栈,它是仅在表尾进行插入和删除操作的线性表。允许插入、删除操作的一端称为栈顶(stack top),另一端称为栈底(stack bottom)。栈顶是动态变化的,通常由一个称为栈顶指针(top)的变量指示。当表中没有元素时,称为空栈。

栈的插入操作称为入栈或进栈,删除操作称为出栈或退栈。

将元素序列 $a_1,a_2,\cdots,a_n$ 依次顺序进栈后,a_1 成为栈底元素,a_n 成为栈顶元素,由栈顶指针 top 指示,如图 3-1 所示。最先进栈的元素位于栈底,最后进栈的元素成为栈顶元素,每次出栈的元素也是栈顶元素。因此,栈是一种后进先出(Last In First Out,LIFO)的线性表。

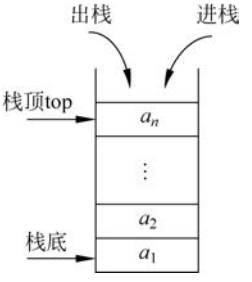

图 3-1 栈

若将元素 a,b,c,d 依次入栈,最后将栈顶元素出栈,则栈顶指针 top 的变化情况如图 3-2 所示。

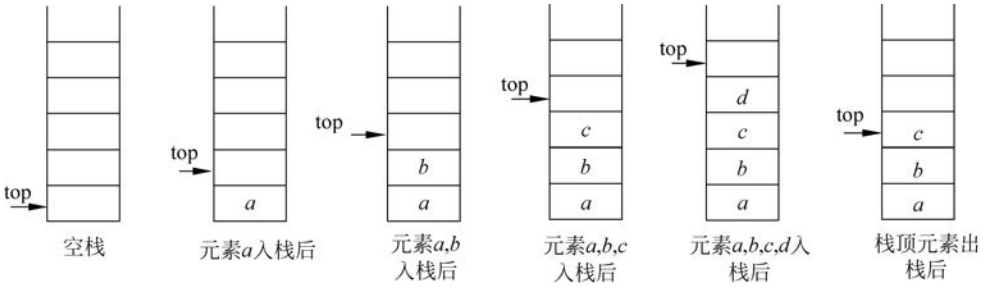

图 3-2 栈的插入和删除过程

【例 3-1】 若 a,b,c 为一个入栈序列,则(　　)不可能是出栈序列。

【分析】 根据栈的"后进先出"特性,出栈序列有 5 种可能:abc,acb,bac,bca 和 cba,而 cab 不可能是出栈序列。这是因为若 c 最先出栈,则说明 c 位于栈顶,且 a 和 b 已经进栈而未出栈,根据入栈顺序和栈的性质,b 一定在 a 的上面。因此,出栈序列可能是 cba,但不可能是 cab。

【例 3-2】 假设一个栈的输入序列为 $P_1,P_2,P_3,\cdots,P_n$,输出序列为 1、2、3、…、n。如果 $P_3=1$,则 P_1 的值(　　)。

A. 可能是 2　　B. 一定是 2　　C. 不可能是 2　　D. 不可能是 3

【分析】 因为 $P_3=1$ 且 1 是第一个出栈的元素,说明栈中还有 P_2、P_1 两个元素,其中 P_2 为新的栈顶元素,P_1 位于栈底。若 $P_2=2$、$P_1=3$,则出栈序列为 1,2,3,…;若 $P_2=3$、$P_1=2$,则出栈序列为 1,3,2,…。因此,P_1 的值可能是 3,但不可能是 2,故选项 C 是正确的,具体如图 3-3 所示。

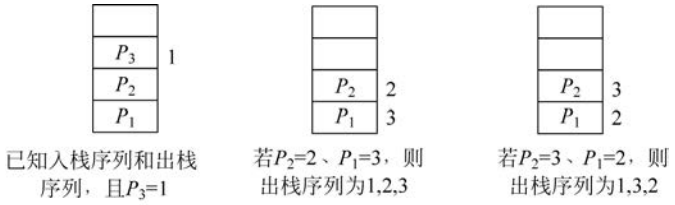

图 3-3　P_1、P_2 和 P_3 在栈中的情况

3.1.2　栈的抽象数据类型


视频讲解

栈的抽象数据类型描述如表 3-1 所示。

表 3-1　栈的抽象数据类型描述

数据对象	栈的数据对象集合为$\{a_1,a_2,\cdots,a_n\}$，每个元素都有相同的类型	
数据关系	栈的数据元素之间是一对一的关系。栈具有线性表的特点：除了第一个元素 a_1 外，每个元素有且只有一个直接前驱元素；除了最后一个元素 a_n 外，每个元素有且只有一个直接后继元素	
基本操作	InitStack(&S)	初始化操作，建立一个空栈 S。这就像准备好了一个空箱子，还没有往里面放盘子
	StackEmpty(S)	判断栈是否为空。若栈 S 为空，则返回 1，否则返回 0。栈空就像准备好了箱子，箱子还是空的，里面没有盘子；栈不空，说明箱子里已经放进了盘子
	GetTop(S,&e)	返回栈 S 的栈顶元素给 e。栈顶元素就像箱子里最上面的那个盘子
	PushStack(&S,e)	在栈 S 中插入元素 e，使其成为新的栈顶元素。这就像在箱子里新放入了一个盘子，这个盘子成为一摞盘子中最上面的一个
	PopStack(&S,&e)	删除栈 S 的栈顶元素，并用 e 返回其值。这就像把箱子里最上面那个盘子取出来
	StackLength(S)	返回栈 S 的元素个数。这就像清点放在箱子里的盘子总共有多少个
	ClearStack(S)	清空栈 S。这就像把箱子里的盘子全部取出来

3.1.3　栈的顺序表示与实现


视频讲解

栈有两种存储结构：顺序存储和链式存储。本节主要介绍栈的顺序存储结构及基本运算实现。

1. 顺序栈的类型定义

采用顺序存储结构的栈称为顺序栈。顺序栈利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，可利用 Java 语言中的数组作为顺序栈的存储结构，同时附设一个栈顶指针 top，用于指向顺序栈的栈顶元素。当 top=0 时，表示空栈。栈的顺序存储结构类型在类中进行定义，具体由栈的初始化实现。

栈的顺序存储结构类型及基本运算通过自定义类实现，顺序栈的类名定义为

SeqStack,SeqStack 中的类成员变量描述如下。

```
public class SeqStack {
       int top;
       final int MAXSIZE = 50;
       char stack[];
}
```

其中,stack[]用于存储栈中的数据元素,top 为栈顶指针,MAXSIZE 为栈的最大容量。

当栈中元素个数为 MAXSIZE 时,称为栈满。如果继续进栈操作则会产生溢出,称为上溢。对空栈进行删除操作,就会产生下溢。

顺序栈的结构如图 3-4 所示。元素 a、b、c、d、e、f、g、h 依次进栈后,a 为栈底元素,h 为栈顶元素。在实际操作中,栈顶指针指向栈顶元素的下一个位置。

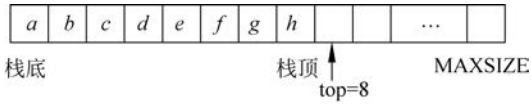

图 3-4 顺序栈结构

在执行进栈操作前,要先判断栈是否已满。若栈未满,则将元素压入栈中,即 S. stack[S. top]=e,然后使栈顶指针加 1,即 S. top++=1。在执行出栈操作前,要先判断栈是否为空。若栈为空,则使栈顶指针减 1,即 S. top--=1,然后元素出栈,即 e=S. stack[S. top]。判断顺序栈为空的条件:top==0,判断顺序栈是否已满的条件:top>=MAXSIZE。

2. 顺序栈的基本运算

顺序栈的基本操作及类方法名称如表 3-2 所示。

表 3-2 SeqStack 类的成员函数

基 本 操 作	基本操作的类方法名称
栈的初始化	SeqStack()
判断栈是否为空	StackEmpty()
入栈	PushStack(*e*)
出栈	PopStack()
取栈顶元素	GetTop()
求栈的长度	StackLength()
清空栈	ClearStack()

(1) 初始化栈。

```
SeqStack() {
     top = 0;
     stack = new char[MAXSIZE];
}
```

(2) 判断栈是否为空。

```
public boolean StackEmpty() {
    if (top == 0)
        return true;
    else
```

```
        return false;
}
```

(3) 取栈顶元素。在取栈顶元素前,先判断栈是否为空。如果栈为空,则抛出异常表示取栈顶元素失败;否则,将栈顶元素返回。取栈顶元素的算法实现如下。

```
public char GetTop() throws Exception {
      if (StackEmpty()) {
           System.out.println("栈为空,取栈顶元素失败!");
           throw new Exception("栈为空!");
      } else
           return stack[top - 1];
}
```

(4) 将元素 *e* 入栈。在将元素 *e* 入栈前,需要先判断栈是否已满。若栈已满,则返回 false 表示入栈操作失败;否则,将元素 *e* 压入栈中,然后将栈顶指针 top 加 1,并返回 ture 表示入栈操作成功。入栈操作的算法实现如下。

```
public boolean PushStack(char e)
{
      if(top >= MAXSIZE) {
            System.out.println("栈已满!");
            return false;
      }
      else {
            stack[top] = e;
            top++;
            return true;
      }
}
```

(5) 将栈顶元素出栈。在将元素出栈前,需要先判断栈是否为空。若栈为空,则抛出异常;若栈不为空,则先使栈顶指针减 1,然后将栈顶元素返回。出栈操作的算法实现如下。

```
public char PopStack() throws Exception {
      if (StackEmpty()) {
            System.out.println("栈为空,不能进行出栈操作!");
            throw new Exception("栈为空!");
      }
      else
      {
            top--;
            char x = stack[top];
            return x;
      }
}
```

(6) 求栈的长度。

```
public int StackLength()
{
    return top;
}
```

(7) 清空栈。

```
public void ClearStack()
{
     top = 0;
}
```

3. 顺序栈应用示例

【例 3-3】 任意给定一个数学表达式如{5×(9－2)－[15－(8－3)/2]}＋3×(6－4),试设计一个算法判断表达式的括号是否匹配。

【分析】 检验括号是否匹配可以设置一个栈,依次读入括号进行判断,如果是左括号,则直接进栈。

(1) 如果读入的是右括号:

① 且与当前栈顶的左括号是同类型的,则说明这对括号是匹配的,则将栈顶的左括号出栈;否则为不匹配。

② 且栈已经为空,则说明缺少左括号,该括号序列不匹配。

(2) 如果输入序列已经读完,而栈中仍然有等待匹配的左括号,则说明缺少右括号,该括号序列不匹配。

(3) 如果读入的是数字字符,则不进行处理,直接读入下一个字符。

当输入序列和栈同时变为空时,则说明括号完全匹配。

程序实现如下。

```
public static boolean Match(char e,char ch) {
        //判断左右两个括号是否为同类型,同类型则返回 true,否则返回 false
        if (e == '('&& ch==')')
            return true;
        else if(e == '['&& ch == ']')
            return true;
        else if(e == '{'&& ch == '}')
            return true;
        else
            return false;
    }
    public static void main(String args[]) throws Exception {
        SeqStack S = new SeqStack();
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入算术表达式:");
        String str = sc.nextLine();
        int i = 0;
        while(i<str.length()) {
            if(str.charAt(i) == '('|| str.charAt(i) =='['|| str.charAt(i) =='{')
                                    //如果是左括号,将括号进栈
            {
                S.PushStack(str.charAt(i));
                i++;
            }
            else if(str.charAt(i) ==')'|| str.charAt(i) ==']'|| str.charAt(i) =='}')
            {
                if(S.StackEmpty()) {
                    System.out.println("缺少左括号");
```

```
                    throw new Exception("缺少右括号");
                }
                else
                {
                    char e = S.GetTop();  //如果栈不空且读入的是右括号,则取出栈顶的括号
                    if(Match(e, str.charAt(i)))     //如果栈顶括号与读入的右括号匹配,则
                                                    //将栈顶的括号出栈
                    {
                        e = S.PopStack();
                        i++;
                    }
                    else                    //否则
                    {
                        System.out.println("左右括号不匹配");
                        throw new Exception("缺少右括号");
                    }
                }
            }
            else                            //如果是其他字符,则不处理,直接将 p 指向下一个字符
                i++;
        }
        if (S.StackEmpty())                 //如果字符序列读入完毕且栈已空,说明括号序列匹配
            matching("括号匹配");
        else                                //如果字符序列读入完毕且栈不空,说明缺少右括号
            System.out.println("缺少右括号");
    }
}
```

程序的运行结果如下。

请输入算术表达式：

[9－6]＋[5－(6＋3)]

括号匹配

3.1.4　栈的链式表示与实现


视频讲解

在顺序栈中，由于顺序存储结构需要事先静态分配，而存储规模往往又难以确定，如果栈空间分配过小，可能会造成溢出；如果栈空间分配过大，又造成存储空间浪费。因此，为了克服顺序栈的缺点，可以采用链式存储结构表示栈。本节主要介绍链式栈的存储结构及基本运算实现。

1. 链栈的存储结构

栈的链式存储结构用一组不一定连续的存储单元来存放栈中的数据元素。一般来说，当栈中数据元素的数目变化较大或不确定时，使用链式存储结构作为栈的存储结构是比较合适的。人们将用链式存储结构表示的栈称为链栈或链式栈。

链栈通常用单链表示。插入和删除操作都在栈顶指针的位置进行，这一端称为栈顶，通常由栈顶指针 top 指示。为了操作方便，通常在链栈中设置一个头结点，用栈顶指针 top 指向头结点，头结点的指针指向链栈的第一个结点。例如，元素 a、b、c、d 依次入栈的链栈如图 3-5 所示。

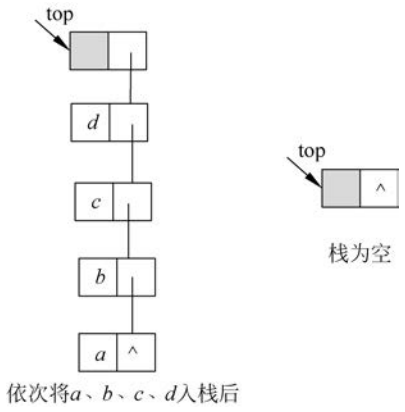

图 3-5 带头结点的链栈

栈顶指针 top 始终指向头结点,最先入栈的元素在链栈的栈底,最后入栈的元素成为栈顶元素。由于链栈的操作都是在链表的表头位置进行的,因而链栈的基本操作的时间复杂度均为 $O(1)$。

链栈的结点类型描述如下。

```
class LinkStackNode
{
    int data;
    LinkStackNode next;
    LinkStackNode(int data)
    {
        this.data = data;
        this.next = next;
    }
}
```

对于带头结点的链栈,在初始化链栈时,有 top.next = null,判断栈空的条件为 top.next==null。对于不带头结点的链栈,在初始化链栈时,有 top=null,判断栈空的条件为 top ==null。

采用链式存储的栈不必事先估算栈的最大容量,只要系统有可用的空间,就能随时为结点申请空间,不用考虑栈满的情况。

2. 链栈的基本运算

链栈的基本运算通过 LinkStack 类实现,在该类中定义相关基本运算,结点的分配需要调用 LinkStackNode 类。链栈的基本操作及 MyLinkStack 类的成员方法名称如表 3-3 所示。

表 3-3 MyLinkStack 类的成员函数

基 本 操 作	基本操作的类方法名称
链栈的初始化	LinkStack ()
判断链栈是否为空	StackEmpty()
入栈	PushStack(e)

续表

基 本 操 作	基本操作的类方法名称
出栈	PopStack()
取栈顶元素	GetTop()
求栈的长度	StackLength()
链栈的创建	CreateStack()

(1) 初始化链栈。初始化链栈由构造方法 LinkStack()实现,即初始化栈顶指针 top。初始化链栈的算法实现如下。

```
public class LinkStack {
    LinkStackNode top;
    LinkStack()
    {
        top = new LinkStackNode(0);
    }
}
```

(2) 判断链栈是否为空。如果头结点指针域为空,说明链栈为空,则返回 true; 否则,返回 false。判断链栈是否为空的算法实现如下。

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

(3) 将元素 e 入栈。先动态生成一个结点,用 pnode 指向该结点,再将元素 e 值赋给 pnode 结点的数据域,然后将新结点插入链表的第一个结点之前。把新结点插入链表中分为两个步骤: ①pnode.next=top.next; ②top.next=pnode。入栈操作如图 3-6 所示。

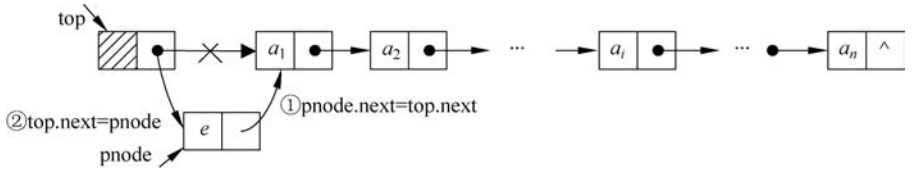

图 3-6　入栈操作

注意　在插入新结点时,需要注意插入结点的顺序不能颠倒。

将元素 e 入栈的算法实现如下。

```
public void PushStack(int e)
{
    LinkStackNode pnode = new LinkStackNode(e);
    pnode.next = top.next;
    top.next = pnode;
}
```

(4) 将栈顶元素出栈。先判断栈是否为空,若栈为空,则抛出异常表示出栈操作失败; 否则,将栈顶元素值赋给 x,将该结点从链表中删除,并将栈顶元素返回。出栈操作

如图 3-7 所示。

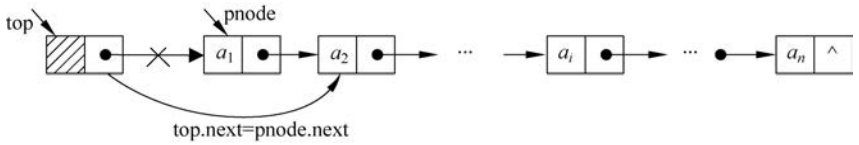

图 3-7 出栈操作

将栈顶元素出栈的算法实现代码如下。

```
public int PopStack() throws Exception {
    if(StackEmpty()) {
        throw new Exception("栈为空,不能进行出栈操作!");
    }
    else {
        LinkStackNode pnode = top.next;
        top.next = pnode.next;
        int x = pnode.data;
        return x;
    }
}
```

(5) 取栈顶元素。在取栈顶元素前要判断链栈是否为空,如果为空,则抛出异常;否则,将栈顶元素返回。取栈顶元素的算法实现如下。

```
public int GetTop() throws Exception {
    if (StackEmpty()) {
        throw new Exception("栈为空!");
    }
    else
        return top.next.data;
  }
```

(6) 求栈的长度。栈的长度就是链栈的元素个数。从栈顶元素开始,通过 next 域找到下一个结点,并使用变量 len 计数,直到栈底为止,len 的值就是栈的长度,将 len 返回即可。求链栈长度的时间复杂度为 $O(n)$。求链栈长度的算法实现如下。

```
public int StackLength() {
    LinkStackNode p = top.next;
    int len = 0;
    while(p!= null) {
        p = p.next;
        len = len + 1;
    }
    return len;
}
```

(7) 创建链栈。创建链栈主要利用了链栈的插入操作思想,根据用户输入的元素序列,将该元素序列存入 arr 中,然后依次取出每个元素,将其插入链栈中,即将元素依次入栈。创建链栈的算法实现如下。

```
public void CreateStack() {
    System.out.print("请输入要入栈的整数:");
```

```
        Scanner input = new Scanner(System.in);
        String s = input.nextLine();
        String[] arr = s.split(" ");
        for(int i = 0;i < arr.length;i++) {
            LinkStackNode pnode = new LinkStackNode(Integer.parseInt(arr[i]));
            pnode.next = top.next;
            top.next = pnode;
        }
    }
```

测试代码如下：

```
    public static void main(String args[]) throws Exception {
        LinkStack S = new LinkStack();
        S.CreateStack();
        System.out.println("栈顶元素:" + S.GetTop());
        System.out.println("长度:" + S.StackLength());
        while(!S.StackEmpty()) {
            int e = S.PopStack();
            System.out.println("出栈元素:" + e);
        }
    }
```

程序运行结果如下。

```
请输入要入栈的整数：6 2 3 1 4
栈顶元素：4
长度：5
出栈元素：4
出栈元素：1
出栈元素：3
出栈元素：2
出栈元素：6
```

3. 链栈应用示例

【例 3-4】 利用链表模拟栈实现将十进制数 5678 转换为对应的八进制数。

【分析】 进制转换是计算机实现计算的基本问题，可以采用辗转相除法实现将十进制数转换为八进制数。将 5678 转换为八进制数的过程如图 3-8 所示。

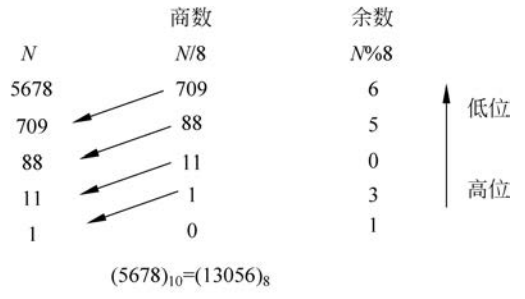

图 3-8　十进制数 5678 转换为八进制数的过程

转换后的八进制数为$(13056)_8$。观察图 3-8 的转换过程,每次不断利用被除数除以 8 得到商数后,记下余数,又将商数作为新的被除数继续除以 8,直到商数为 0 为止,将得到的余数排列起来就是转换后的八进制数。十进制数 N 转换为八进制的算法如下:

(1) 将 N 除以 8,记下其余数;

(2) 判断商是否为零,如果为零,则结束程序;否则,将商送入 N,转到(1)继续执行。

将得到的余数逆序排列就是转换后的八进制数,得到的位序正好与八进制数的位序相反,这正好可以利用栈的"后进先出"特性,先把得到的余数序列放入栈保存,最后依次出栈即可得到八进制数。

在利用链表实现将十进制数转换为八进制数时,可以将每次得到的余数按照头插法插入链表,即将元素入栈,然后从链表的头指针开始依次输出结点的元素值,就得到了八进制数,即将元素出栈。这正好是元素的入栈与出栈操作。也可以利用栈的基本操作实现栈的进制转换。

十进制转换为八进制的算法描述如下。

```
public class Convert10to8 {
    public static int covert10to8(int x,int num[]) {
       LinkStackNode top = null,p;
     while(x != 0) {
         p = new LinkStackNode(x % 8);
         p.next = top;
         top = p;
         x = x / 8;
     }
     int k = 0;
     while(top!= null)
     {
         p = top;
         num[k++] = p.data;
         top = top.next;
     }
     return k;
    }
    public static void main(String args[]) {
     Scanner input = new Scanner(System.in);
     System.out.println("请输入一个十进制整数:");
     int num[] = new int[20],count;
     int x = input.nextInt();
     count = covert10to8(x,num);
     System.out.print("转换后的八进制数是:");
     for(int i = 0;i < count;i++)
         System.out.print(num[i]);
     }
}
```

程序运行结果如下。

```
请输入一个十进制整数:5678
转换后的八进制数是:13056
```

3.1.5　栈的典型应用

【例 3-5】　通过键盘输入一个表达式，如 6＋(7－1)×3＋9/2，要求将其转换为后缀表达式，并计算该表达式的值。

【分析】　表达式求值是程序设计编译中的基本问题，它正是利用了栈的“后进先出”思想把人们便于理解的表达式翻译成计算机能够正确理解的表示序列。

一个算术表达式是由操作数、运算符和分界符组成的。为了简化问题求解，假设算术运算符仅由加、减、乘、除 4 种运算符和左、右圆括号组成。

例如，一个算术表达式为 6＋(7－1)×3＋9/2。

这种算术表达式中的运算符总是出现在两个操作数之间，称为中缀表达式。计算机编译系统在计算一个算术表达式之前，要将中缀表达式转换为后缀表达式，然后对后缀表达式进行计算。后缀表达式的算术运算符出现在操作数之后，并且不含括号。

计算机在求解算术表达式的值时分为以下两个步骤：

(1) 将中缀表达式转换为后缀表达式；

(2) 依据后缀表达式计算表达式的值。

1. 将中缀表达式转换为后缀表达式

要将一个算术表达式的中缀形式转换为后缀形式，首先需要了解算术四则运算规则。算术四则运算的规则是：

(1) 先乘除，后加减；

(2) 同级别的运算从左到右依次进行计算；

(3) 先括号内，后括号外。

上面的算术表达式可转换为后缀表达式 6 7 1 － 3 × ＋ 9 2 / ＋。

不难看出，转换后的后缀表达式具有以下两个特点：

(1) 后缀表达式与中缀表达式的操作数出现顺序相同，只是运算符先后顺序改变了；

(2) 后缀表达式不出现括号。

在利用后缀表达式进行算术运算时，编译系统不必考虑运算符的优先关系，仅需要从左到右依次扫描后缀表达式的各个字符。当系统遇到运算符时，直接对运算符前面的两个操作数进行运算即可。

如何将中缀表达式转换为后缀表达式呢？可设置一个栈，用于存放运算符。这里约定'#'作为中缀表达式的结束标志，并假设 θ_1 为栈顶运算符，θ_2 为当前扫描的运算符。运算符的优先关系如表 3-4 所示。

表 3-4　运算符的优先关系

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

续表

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

依次读入表达式中的每个字符,根据读取的当前字符进行以下处理:

(1) 初始化栈,并将'#'入栈。

(2) 若当前读入的字符是操作数,则将该操作数输出,并读入下一字符。

(3) 若当前字符是运算符,记作 θ_2,将 θ_2 与栈顶的运算符 θ_1 比较。若 θ_1 优先级低于 θ_2,则将 θ_2 进栈;若 θ_1 优先级高于 θ_2,则将 θ_1 出栈并将其作为后缀表达式输出。然后继续比较新的栈顶运算符 θ_1 与当前运算符 θ_2 的优先级,若 θ_1 的优先级与 θ_2 相等,且 θ_1 为"(",θ_2 为")",则将 θ_1 出栈,继续读入下一个字符。

(4) 如果 θ_2 的优先级与 θ_1 相等,且 θ_1 和 θ_2 都为'#',则将 θ_1 出栈,栈为空。此时中缀表达式转换为后缀表达式,算法结束。

重复执行步骤(2)~(4),直到所有字符读取完毕。

中缀表达式 6+(7−1)×3+9/2#转换为后缀表达式的具体过程如图 3-9 所示(为了转换方便,在要转换表达式的末尾加一个'#'作为结束标记)。

2. 求后缀表达式的值

将中缀表达式转换为后缀表达式后,就可以计算后缀表达式的值了。计算后缀表达式的值的规则为:依次读入后缀表达式中的每个字符,如果是操作数,则将操作数进入栈;如果是运算符,则将处于栈顶的两个操作数出栈,然后利用当前运算符进行运算,将运行结果入栈,直到整个表达式处理完毕。

利用上述规则,后缀表达式的 6 7 1 − 3 × + 9 2 / +的值的运算过程如图 3-10 所示。

3. 算法实现

在算法实现时,设置两个字符数组 str、exp 及一个栈 S1,其中,str 用于存放中缀表达式的字符串,exp 用于存放转换后的后缀表达式字符串,S1 用于存放转换过程中遇到的运算符。

(1) 将中缀表达式转换为后缀表达式的方法是:依次扫描数组 str 中的每个字符,如果遇到的是数字,则将其直接存入数组 exp 中。如果遇到的是运算符,则将 S1 的栈顶运算符与当前运算符比较,若当前运算符的优先级高于栈顶运算符的优先级,则将当前运算符入栈 S1;若栈顶运算符的优先级高于当前运算符的优先级,则将 S1 的栈顶运算符出栈,并保存到 exp 中。

(2) 求后缀表达式的值时,依次扫描后缀表达式中的每个字符,如果是数字字符,则将其转换为数字(数值型数据),并将其入栈;如果是运算符,则将栈顶的两个数字出栈,进行加、减、乘、除运算,并将结果入栈。当后缀表达式对应的字符串处理完毕后,将栈顶元素返回给被调用函数,即为所求表达式的值。

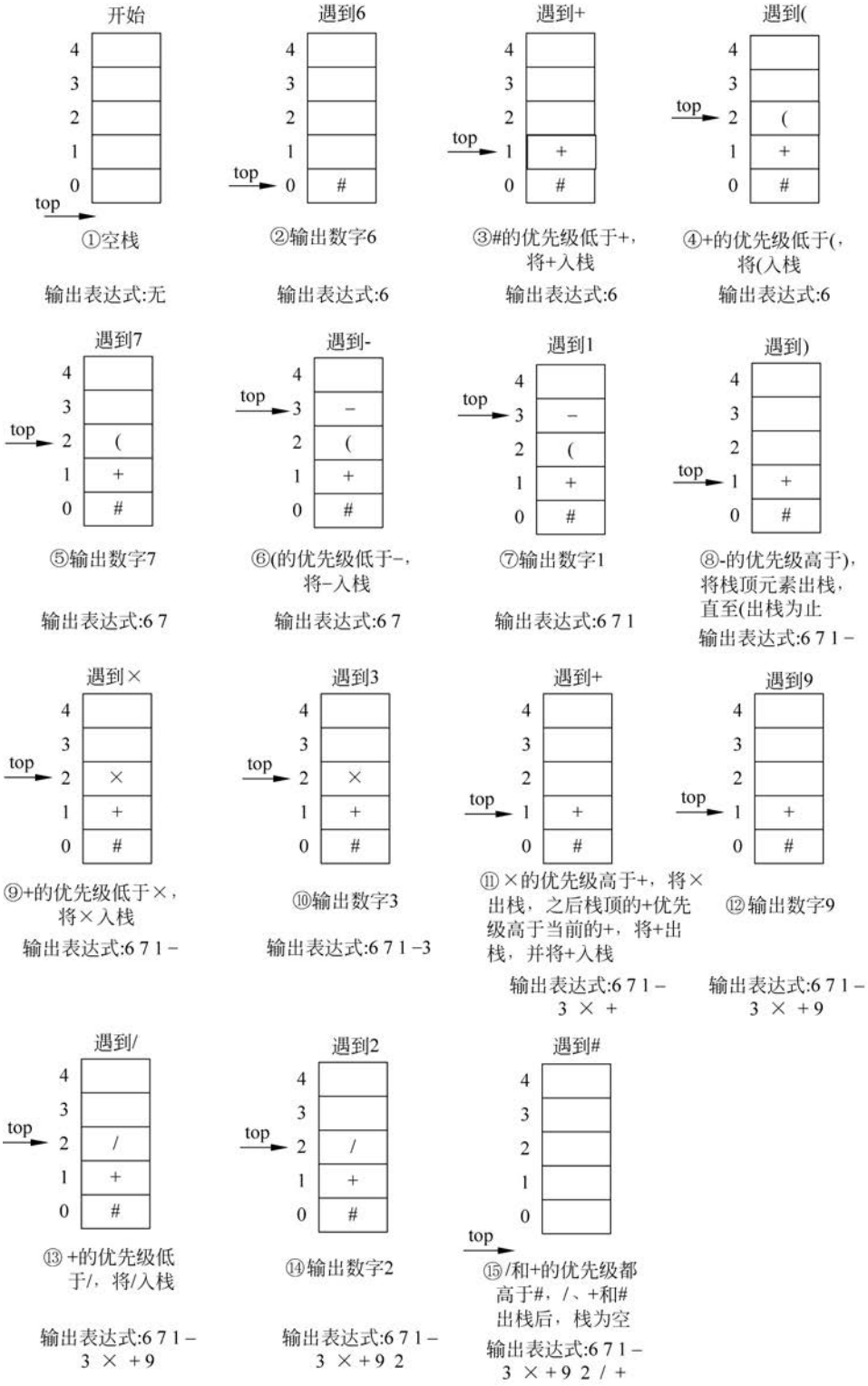

图 3-9　中缀表达式 6＋(7－1)×3＋9/2 转换为后缀表达式的过程

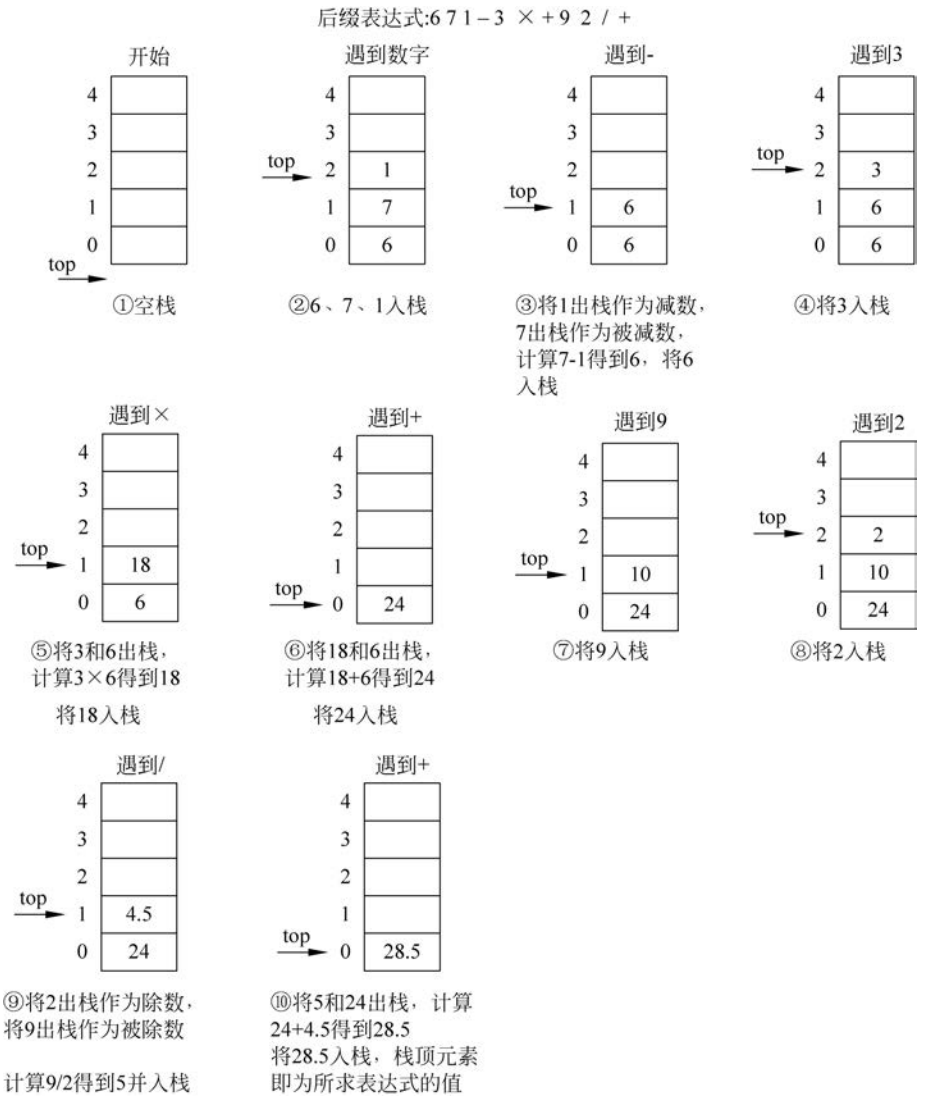

图 3-10　后缀表达式 6 7 1－3 × ＋ 9 2 / ＋的运算过程

利用栈求解算术表达式的值的算法实现如下。

```
import java.util.Scanner;
public class CalculateExpress {
    public static void TranslateExpress(char str[],char exp[]) throws Exception
    //中缀表达式转换为后缀表达式
    {
        int i,j;
        i = 0;
        j = 0;
        boolean end = false;
        char ch = str[i],e;
        SeqStack S = new SeqStack();
        i = i + 1;
        while (i <= str.length &&!end)
        {
```

```
            if (ch == '(')                  //如果当前字符是左括号,则将其进栈
                push(ch);
            else if(ch == ')')              //如果是右括号,则将栈中的运算符出栈,并将其存入
                                            //数组 exp 中
            {
                while (S.GetTop() != '(') {
                    e = S.PopStack();
                    exp[j] = e;
                    j = j + 1;
                }
                e = S.PopStack();           //将左括号出栈
            }
            else if(ch == '+'|| ch == '-')     //如果遇到的是 '+' 和 '-',因为其优先级低
于栈顶运算符的优先级,所以先将栈顶字符出栈,并将其存入 exp 中,然后将当前运算符进栈
            {
                while (!S.StackEmpty() && S.GetTop() != '(') {
                    e = S.PopStack();
                    exp[j] = e;
                    j = j + 1;
                }
                S.PushStack(ch);            //当前运算符进栈
            }
            else if(ch == '*'|| ch == '/')     //如果遇到 '*' 和 '/', 先将同级运算符出栈,
                                               //并存入 exp 中,然后将当前的运算符进栈
            {
                while (!S.StackEmpty() && S.GetTop() == '/'|| S.GetTop() == '*')
                {
                    e = S.PopStack();
                    exp[j] = e;
                    j = j + 1;
                 }
                S.PushStack(ch);            //当前运算符进栈
            }
            else if(ch == ' ')              //如果遇到空格,则忽略
                break;
            else                            //若遇到操作数,则将操作数直接送入 exp 中
            {
                while (ch >= '0' && ch <= '9') {
                    exp[j] = ch;
                    j = j + 1;
                    if (i < str.length)
                        ch = str[i];
                    else {
                        end = true;
                        break;
                    }
                    i = i + 1;
                }
                i = i - 1;
            }
            ch = str[i];                    //读入下一个字符,准备处理
            i = i + 1;
        }
        while (!S.StackEmpty())             //将栈中所有剩余的运算符出栈,送入 exp 中
```

```
    {
        e = S.PopStack();
        exp[j] = e;
        j = j + 1;
    }
}

public static float ComputeExpress(char a[]) throws Exception {
    int i = 0;
    float x1 = 0.0f,x2 = 0.0f,result = 0.0f;
    LinkStack S = new LinkStack();
    while (i < a.length) {
        if (a[i] >= '0' && a[i] <= '9') {
            S.PushStack(a[i] - '0');   //处理之后将数字进栈
        }
        else {
            if (a[i] == '+') {
                x1 = S.PopStack();
                x2 = S.PopStack();
                result = x1 + x2;
                S.PushStack(result);
            } else if (a[i] == '-') {
                x1 = S.PopStack();
                x2 = S.PopStack();
                result = x2 - x1;
                S.PushStack(result);
            } else if (a[i] == '*') {
                x1 = S.PopStack();
                x2 = S.PopStack();
                result = x1 * x2;
                S.PushStack(result);
            } else if (a[i] == '/') {
                x1 = S.PopStack();
                x2 = S.PopStack();
                result = x2 / x1;
                S.PushStack(result);
            }
        }
        i = i + 1;
    }
    if (!S.StackEmpty())                //如果栈不空,则将结果出栈,并返回
        result = S.PopStack();
    if (S.StackEmpty())
        return result;
    else
        System.out.println("表达式错误");
    return result;
}
public static void main(String args[]) throws Exception {
    SeqStack S =  new SeqStack();
    Scanner sc = new Scanner(System.in);
    System.out.println("请输入一个算术表达式:");
    String inputstring = sc.nextLine();
    char str[] = inputstring.toCharArray();
```

```
        char exp[] = new char[str.length];
        TranslateExpress(str, exp);
        System.out.println("后缀表达式:");
        for(int i = 0;i < exp.length;i++)
            System.out.print(exp[i] + " ");
        System.out.print("\n 表达式的值 = " + ComputeExpress(exp));
    }
}
```

程序运行结果如下。

请输入一个算术表达式：

6＋(7－1)＋×3＋9/2

后缀表达式：

6 7 1－3×＋9 2/＋

表达式的值＝28.5

注意　(1) 在将中缀表达式转换为后缀表达式的过程中，如果遇到连续的数字字符，则需要将连续的数字字符作为一个数字处理，而不是作为两个或多个数字，这可以在函数 ComputeExpress 或 TranslateExpress 中进行处理。

(2) 在 ComputeExpress()函数中，当遇到-运算符时，先出栈的为减数，后出栈的为被减数。对于/运算也一样。

【思考】 能否在求解算术表达式的值时，不输出转换的后缀表达式而直接进行求值？

【分析】 求解算术表达式的值也可以将中缀表达式转换为后缀表达式和利用后缀表达式求值同时进行，这需要定义两个栈：运算符栈和操作数栈。只是要将原来操作数的输出变成入操作数栈操作，在运算符出栈时，需要将操作数栈中的元素输出并进行相应运算，将运算后的操作数入操作数栈。

算法主要代码如下：

```
public static Float CalExpress(char str[]) throws Exception     //计算表达式的值
{
    OptStack < String > Optr = new OptStack < String >();
    OptStack < Float > Opnd = new OptStack < Float >();
    OptStack < Integer > TempStack = new OptStack < Integer >();
    Optr.Push("#");
    int n = str.length;
    int i = 0, k = 0;
    int base = 1;
    Integer res = 0;
    char a[] = new char[20];
    System.out.println("运算符栈和操作数栈的变化情况如下:");
    while (i < n || Optr.GetTop() != null) {
        if (i < n && IsOptr(!str[i]))                            //是操作数
        {
            while (i < n &&! IsOptr(str[i]))                     //读入的是数字
            {
                TempStack.Push(str[i] - '0');                    //将数字字符转换为数字并暂存
                i += 1;
                while (!TempStack.StackEmpty())                  //将暂存的数字序列转换为一
                                                                 //个完整的数字
```

```
                {
                    Integer evalue = TempStack.Pop();
                    res += evalue * base;
                    base * = 10;
                }
            }
        }
        base = 1;
        if (res != 0) {
            Opnd.Push(res * 1.0f);                          //将运算结果压入 Opnd 栈
            DispStackStatus(Optr, Opnd);
        }
        res = 0;
        if (IsOptr(str[i]))                                 //是运算符
        {
            if (Precede((String) Optr.GetTop().data, str[i]) == '<') {
                Optr.Push(String.valueOf(str[i]));
                i += 1;
                DispStackStatus(Optr, Opnd);
            } else if (Precede((String) Optr.GetTop().data, str[i]) == '>') {
                String theta = Optr.Pop();
                Float rvalue = Opnd.Pop();
                Float lvalue = Opnd.Pop();
                Float exp = GetValue(theta, lvalue, rvalue);
                Opnd.Push(exp);
                DispStackStatus(Optr, Opnd);
            } else if (Precede((String) Optr.GetTop().data, str[i]) == '=') {
                String theta = Optr.Pop();
                i += 1;
                DispStackStatus(Optr, Opnd);
            }
         }
    }
    return (Float) Opnd.GetTop().data;
}

public static Float GetValue(String ch, Float a, Float b) throws Exception     //求值
{
    if(ch.equals("+"))
        return (Float)(a +b);
    else if(ch.equals("-"))
        return (Float)(a-b);
    else if(ch.equals("*"))
        return (Float)(a*b);
    else if(ch.equals("/"))
        return (Float)(a/b);
    else
        throw new Exception("运算符异常");
}
public static void main(String args[]) throws Exception {
    Scanner sc = new Scanner(System.in);
    System.out.println("请输入算术表达式串:");
    String str = sc.nextLine();
    Float res = CalExpress(str.toCharArray());
```

```
        System.out.print("表达式" + str + "的运算结果为:" + res);
    }
```

程序运行结果如下。

```
请输入算术表达式串:
6+(7-1)×3+9/2#
运算符栈和操作数栈的变化情况如下:
运算符栈:# ,操作数栈:6.0
运算符栈:# + ,操作数栈:6.0
运算符栈:# + ( ,操作数栈:6.0
运算符栈:# + ( ,操作数栈:6.0 7.0
运算符栈:# + ( - ,操作数栈:6.0 7.0
运算符栈:# + ( - ,操作数栈:6.0 7.0 1.0
运算符栈:# + ( ,操作数栈:6.0 6.0
运算符栈:# + ,操作数栈:6.0 6.0
运算符栈:# + × ,操作数栈:6.0 6.0
运算符栈:# + × ,操作数栈:6.0 6.0 3.0
运算符栈:# + ,操作数栈:6.0 18.0
运算符栈:# ,操作数栈:24.0
运算符栈:# + ,操作数栈:24.0
运算符栈:# + ,操作数栈:24.0 9.0
运算符栈:# + / ,操作数栈:24.0 9.0
运算符栈:# + / ,操作数栈:24.0 9.0 2.0
运算符栈:# + ,操作数栈:24.0 4.5
运算符栈:# ,操作数栈:28.5
运算符栈:,操作数栈:28.5
表达式 6+(7-1)×3+9/2# 的运算结果为:28.5
```

【思考】 若遇到连续字符串表示的多位数,如“123+16×20”中的“123”、“16”和“20”,要将这些字符串转换为对应的整数,如何不使用栈进行处理呢?

3.2　栈与递归

栈的“后进先出”的思想在递归函数中同样有所体现。本节主要介绍栈与递归调用的关系、递归利用栈的实现过程、递归与非递归的转换。

3.2.1　设计递归算法


视频讲解

递归是指在函数的定义中,在定义自己的同时又出现了对自身的调用。如果一个函数在函数体中直接调用自己,称为直接递归函数。如果一个函数经过一系列的中间调用,间接调用自己,称为间接递归函数。

1. 斐波那契数列

【例 3-6】 如果兔子在出生两个月后就有繁殖能力,以后一对兔子每个月能生出一对兔子,假设所有兔子都正常存活,那么一年以后可以繁殖多少对兔子呢?

不妨拿新出生的一对小兔子来分析下。第一、二个月小兔子没有繁殖能力,共有 1 对兔子;两个月后,生下一对小兔子,共有 2 对兔子;三个月后,老兔子又生下一对,因为小兔子还没有繁殖能力,所以一共是 3 对兔子;以此类推,可以得出如表 3-5 所示的每月兔子的对数。

表 3-5 每月兔子的对数

经过的月数	1	2	3	4	5	6	7	8	9	10	11	12
兔子对数	1	1	2	3	5	8	13	21	34	55	89	144

从表 3-5 中不难看出,数字 1、1、2、3、5、8…构成了一个数列,这个数列有一个十分明显的特征,即前面相邻两项之和构成后一项,可用数学函数表示如下。

$$\mathrm{Fib}(n)=\begin{cases}0, & \text{当 } n=0 \text{ 时}\\ 1, & \text{当 } n=1 \text{ 时}\\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2), & \text{当 } n>1 \text{ 时}\end{cases}$$

求斐波那契数列的非递归算法实现如下。

```
public static int fib(int f[],int n)
{
    int i = 2;
    f[0] = 1;
    f[1] = 1;
    while(i < n)
    {
        f[i] = f[i - 1] + f[i - 2];
        i++;
    }
    return i;
}
```

如果用递归实现,代码结构会更加清晰。

```
public static int fib2(int n)                    //使用递归方法计算斐波那契数列
{
    if(n == 0)                                   //若是第 0 项
        return 0;                                //则返回 0
    else if(n == 1)                              //若是第 1 项
        return 1;                                //则返回 1
    else                                         //其他情况
        return fib2(n - 1) + fib2(n - 2);        //第三项为前两项之和
}
```

当 $n=4$ 时,递归函数执行过程如图 3-11 所示。

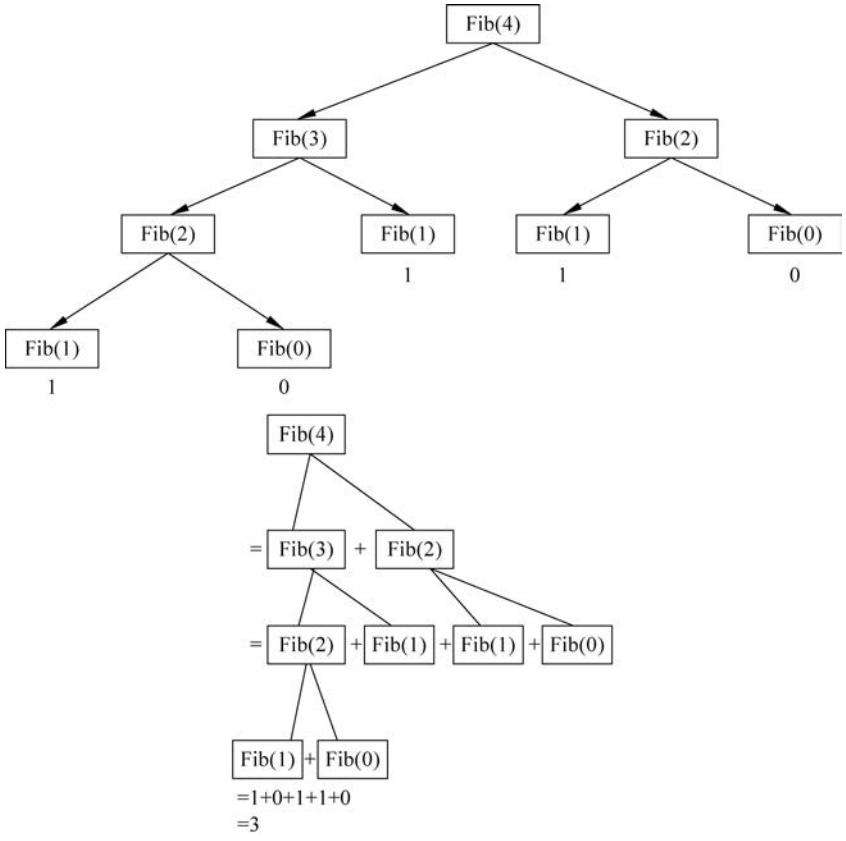

图 3-11　斐波那契数列的执行过程

2. n 的阶乘

【例 3-7】 求 n 的阶乘的递归函数定义如下。

$$\text{fact}(n)=\begin{cases}1, & \text{当 } n=1 \text{ 时}\\ n\times\text{fact}(n-1), & \text{当 } n>1 \text{ 时}\end{cases}$$

n 的阶乘递归算法实现如下。

```
public static int fact(int n)                    //n 的阶乘
{
    if(n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

3. Ackermann 函数

【例 3-8】 Ackermann 函数定义如下。

$$\text{Ack}(m,n)=\begin{cases}n+1, & \text{当 } m=0 \text{ 时}\\ \text{Ack}(m-1,1), & \text{当 } m\neq 0, n=0 \text{ 时}\\ \text{Ack}(m-1),\text{Ack}(m,n-1), & \text{当 } m\neq 0, n\neq 0 \text{ 时}\end{cases}$$

Ackermann 递归函数算法实现如下。

```
public static long Ack(long m,long n)              //Ackermann 递归算法实现
{
    if(m == 0)
        return n + 1;
    else if(n == 0)
        return Ack(m - 1, 1);
    else
        return Ack(m - 1, Ack(m, n - 1));
}
```


视频讲解

3.2.2 分析递归调用过程

递归问题可以被分解成规模小、性质相同的问题加以解决。在之后将要介绍的广义表、二叉树等都具有递归的性质,它们的操作可以用递归实现。下面以著名的汉诺塔问题为例分析递归调用的过程。

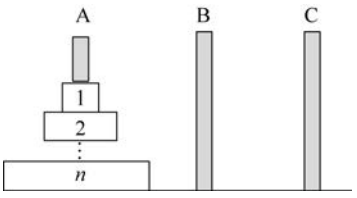

图 3-12 *n* 阶汉诺塔初始状态

n 阶汉诺塔问题。假设有 3 个塔座 A、B、C,在塔座 A 上放置有 *n* 个直径大小各不相同、从小到大编号为 1,2,…,*n* 的圆盘,如图 3-12 所示。要求将塔座 A 上的 *n* 个圆盘移动到塔座 C 上,并要求按照同样的叠放顺序排列。圆盘移动时必须遵循以下规则:

(1) 每次只能移动一个圆盘。

(2) 圆盘可以放置在 A、B 和 C 中的任意一个塔座上。

(3) 任何时候都不能将一个较大的圆盘放在较小的圆盘上。

如何实现将放在塔座 A 上的圆盘按照规则移动到塔座 C 上呢? 当 $n=1$ 时,直接将编号为 1 的圆盘从塔座 A 移动到 C 即可。当 $n>1$ 时,需利用塔座 B 作为辅助塔座,先将放置在编号为 *n* 之上的 $n-1$ 个圆盘从塔座 A 移动到 B,然后将编号为 *n* 的圆盘从塔座 A 移动到 C,最后将塔座 B 上的 $n-1$ 个圆盘移动到塔座 C 上。那现在将 $n-1$ 个圆盘从一个塔座移动到另一个塔座又成为与原问题类似的问题,只是规模减小了 1,故可用同样的方法解决。显然这是一个递归的问题,汉诺塔的递归算法描述如下。

```
public static void Hanoi(int n,String A,String B,String C)
//将塔座 A 上的编号为 1～n 的圆盘按照规则移动到塔座 C 上,B 可以作为辅助塔座
{
    if(n == 1)
        move(1, A, C);              //将编号为 1 的圆盘从 A 移动到 C
    else {
        Hanoi(n - 1, A, C, B);  //将编号为 1～n-1 的圆盘从 A 移动到 B,C 作为辅助塔座
        move(n, A, C);              //将编号为 n 的圆盘从 A 移动到 C
        Hanoi(n - 1, B, A, C);  //将编号为 1～n-1 的圆盘从 B 移动到 C,A 作为辅助塔座
    }
}
public static void move (int n, String tempA, String tempB) {
    System.out.println("move plate" + n + " from column " + tempA + " to column " +  tempB);
}
```

下面以 $n=3$ 为例，观察一下汉诺塔递归调用的具体过程。在函数体中，当 $n>1$ 时，需要 3 个过程移动圆盘。第 1 个过程，将编号为 1 和 2 的圆盘从塔座 A 移动到 B；第 2 个过程，将编号为 3 的圆盘从塔座 A 移动到 C；第 3 个过程，将编号为 1 和 2 的圆盘从塔座 B 移动到 C。递归调用过程如图 3-13 所示。

(1) 第 1 个过程通过调用 Hanoi(2,A,C,B)实现。Hanoi(2,A,C,B)调用自己，完成将编号为 1 的圆盘从塔座 A 移动到 C，编号为 2 的圆盘从塔座 A 移动到 B，编号为 1 的圆盘从塔座 C 移动到 B，如图 3-14 和图 3-15 所示。

(2) 第 2 个过程完成将编号为 3 的圆盘从塔座 A 移动到 C，如图 3-16 所示。

第 3 个过程通过调用 Hanoi(2,B,A,C)实现。通过再次递归完成将编号为 1 的圆盘从塔座 B 移动到 A，将编号为 2 的圆盘从塔座 B 移动到 C，将编号为 1 的圆盘从塔座 A 移动到 C，如图 3-16 和图 3-17 所示。

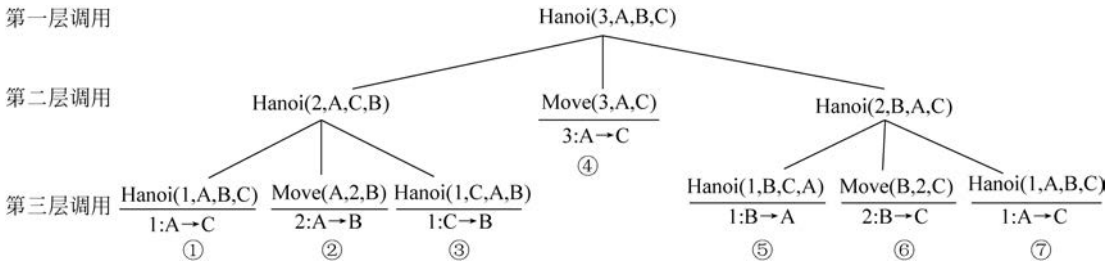

图 3-13　汉诺塔递归调用过程

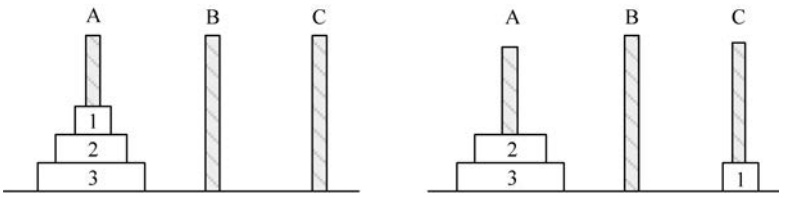

图 3-14　将编号为 1 的圆盘从塔座 A 移动到 C 上

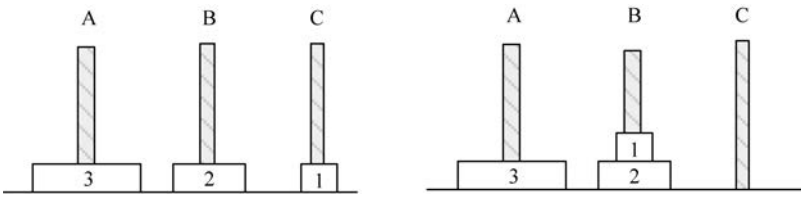

图 3-15　将编号为 2 的圆盘从塔座 A 移动到 B 上，编号为 1 的圆盘从塔座 C 移动到 B 上

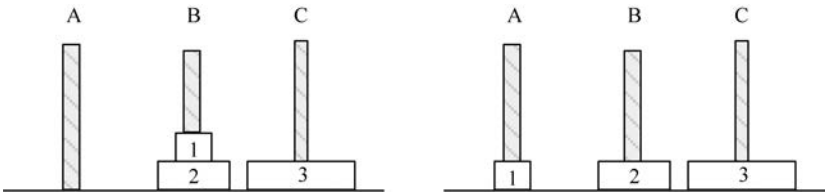

图 3-16　将编号为 3 的圆盘从塔座 A 移动到 C 上，编号为 1 的圆盘从塔座 B 移动到 A 上

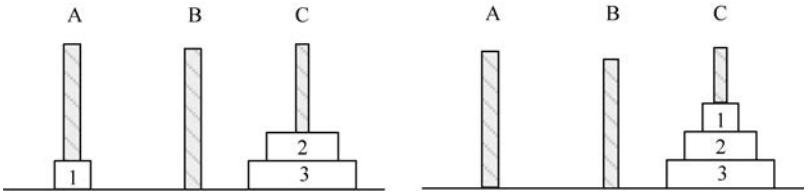

图 3-17 将编号为 2 的圆盘从塔座 B 移动到 C 上,编号为 1 的圆盘从塔座 A 移动到 C 上

递归的实现本质上就是把嵌套调用变成栈实现。在递归调用过程中,被调用函数在执行前系统要完成以下 3 个任务。

(1) 将所有参数和返回地址传递给被调用函数保存。

(2) 为被调用函数的局部变量分配存储空间。

(3) 将控制转到被调用函数的入口。

当被调用函数执行完毕后,在结果返回给调用函数前,系统同样需要完成以下 3 个任务。

(1) 保存被调用函数的执行结果。

(2) 释放被调用函数的数据存储区。

(3) 将控制转到调用函数的返回地址处。

在有多层嵌套调用时,后调用的先返回,这刚好满足后进先出的特性,因此递归调用是通过栈实现的。在函数递归调用过程中,在递归结束前,每调用一次,就进入下一层。当一层递归调用结束时,返回到上一层。

为了保证递归调用能正确执行,系统设置了一个工作栈作为递归函数运行期间使用的数据存储区。每一层递归包括实际参数、局部变量及上一层的返回地址等,这些数据构成一个工作记录。每进入下一层,新的工作栈记录被压入栈顶。每返回到上一层,就从栈顶弹出一个工作记录。因此,当前层的工作记录是栈顶工作记录,也称为活动记录。递归过程产生的栈由系统自动管理,类似用户自己定义的栈。


视频讲解

3.2.3 消除递归

用递归编写的程序结构清晰,算法容易理解与实现,但递归算法的执行效率比较低,这是因为递归需要反复入栈,时间和空间开销都比较大。

为了避免这种开销,就需要消除递归。消除递归的方法通常有两种:一种是对于简单的递归可以通过迭代消除;另一种是利用栈的方式实现。例如,n 的阶乘就是一个简单的递归,可以直接利用迭代消除递归。n 的阶乘的非递归算法如下。

```
public static long fact(int n)      //n 的阶乘的非递归算法实现
{
    long f = 1;
    int i;
    for(i = 1;i < n + 1;i++)        //直接利用迭代消除递归
        f = f * i;
    return f;
}
```

当然,利用栈结构也可以实现 n 的阶乘。

【例 3-9】　编写求 n 的阶乘的递归算法与利用栈实现的非递归算法。

【分析】　利用栈模拟实现求 n 的阶乘。在利用 Java 实现时，可以通过定义一个嵌套的 $n\times 2$ 的数组存储临时变量和每层返回的中间结果，第一维用于存放本层参数 n，第二维用于存放本层要返回的结果。

当 $n=3$ 时，递归调用过程如图 3-18 所示。

在递归函数调用的过程中，各参数入栈情况如图 3-19 所示。为便于描述，用 f 代替 fact 表示函数。

当 $n=1$ 时，递归调用开始逐层返回，参数开始出栈，如图 3-20 所示。

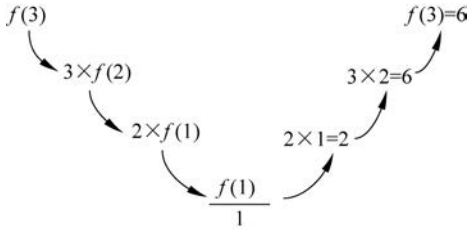

图 3-18　递归调用过程

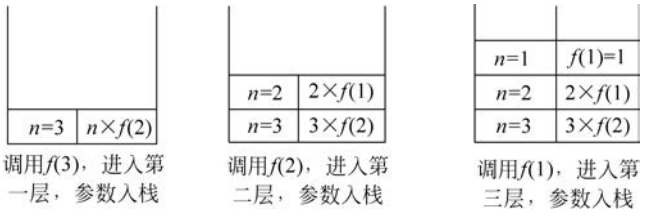

图 3-19　递归调用入栈过程

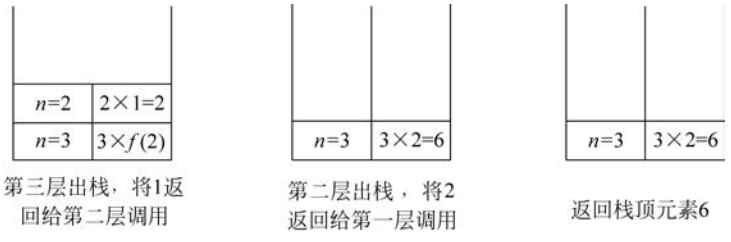

图 3-20　递归调用出栈过程

n 的阶乘的递归与非递归算法实现如下。

```
public static long fact2(int n)                    //n 的阶乘非递归实现
{
    final int MAXSIZE = 50;
    long s[][] = new long[MAXSIZE][2];             //定义一个二维数组用于存储临时变量及返回结果
    int top = -1;                                  //将栈顶指针置为 -1
    top = top + 1;                                 //栈顶指针加 1,将工作记录入栈
    s[top][0] = n;                                 //记录每层的参数
    s[top][1] = 0;                                 //记录每层的结果返回值
    do{
        if (s[top][0] == 1)                        //递归出口
        {
            s[top][1] = 1;
```

```
            System.out.println("n = " + s[top][0] + ",fact = " + s[top][1]);
        }
        if (s[top][0] > 1 && s[top][1] == 0) //通过栈模拟递归的递推过程,将问题依次入栈
        {
            top = top + 1;
            s[top][0] = s[top - 1][0] - 1;
            s[top][1] = 0;                  //将结果置为 0,还没有返回结果
            System.out.println("n = " + s[top][0] + ",fact = " + s[top][1]);
        }
        if (s[top][1] != 0)                 //模拟递归的返回过程,将每层调用的结果返回
        {
            s[top - 1][1] = s[top][1] * s[top - 1][0];
            System.out.println("n = " + s[top - 1][0] + ", fact = " + s[top - 1][1]);
            top = top - 1;
        }
    }while(top > 0);
    return s[0][1];                         //返回计算的阶乘结果
}
public static void main(String args[])
{
    int n;
    Scanner sc = new Scanner(System.in);
    System.out.println("请输入一个正整数(n < 15):");
    n = sc.nextInt();
    System.out.println("递归实现 n 的阶乘:");
    long f = fact(n);                       //调用 n 的阶乘递归实现函数
    System.out.println("n!= " + f);
    System.out.println("利用栈非递归实现 n 的阶乘:");
    f = fact2(n);                           //调用 n 的阶乘非递归实现函数
    System.out.print("n!= " + f);
}
```

程序运行结果如下。

请输入一个正整数(n<15):

5

递归实现 n 的阶乘:

n! =120

利用栈非递归实现 n 的阶乘:

n=4,fact=0

n=3,fact=0

n=2,fact=0

n=1,fact=0

n=1,fact=1

n=2,fact=2

n=3,fact=6

n=4,fact=24

n=5,fact=120

n!=120

利用栈实现的非递归过程可分为以下几个步骤。

(1) 设置一个工作栈,用于保存递归工作记录,包括实参、返回地址等。

(2) 将调用函数传递过来的参数和返回地址入栈。

(3) 利用循环模拟递归分解过程,逐层将递归过程的参数和返回地址入栈。当满足递归结束条件时,依次逐层退栈,并将结果返回给上一层,直到栈空为止。

思政元素
在栈的基本操作实现过程中和利用栈将递归转换为非递归时,都需要用到栈的“后进先出”原理,在利用栈模拟递归的过程中还要保存每一步的参数和返回结果,并且不能出现任何差错。差之毫厘,谬以千里。因此,在算法实现过程中,不仅要遵守规范,而且要养成一丝不苟、精益求精的职业素养。

3.3　队列

与栈类似,队列也是一种操作受限的线性表。队列遵循的是“先进先出”的原则,这一特点决定了队列的操作需要在两端进行。

3.3.1　队列的定义及抽象数据类型


视频讲解

队列只允许在表的一端进行插入操作,在另一端进行删除操作。

1. 队列的定义

队列(queue)是一种先进先出(First In First Out,FIFO)的线性表,它只允许在表的一端进行插入,在另一端进行删除。这与日常生活中的排队是一致的,最早进入队列的元素最早离开。在队列中,允许插入的一端称为队尾(end),允许删除的一端称为队头(front)。

假设队列为 $q=(a_1, a_2, \cdots, a_i, \cdots, a_n)$,则 a_1 为队头元素,a_n 为队尾元素。元素在进入队列时是按照 $a_1, a_2, \cdots, a_n$ 的顺序进入的,退出队列时也是按照这个顺序退出的。当先进入队列的元素都退出后,后进入队列的元素才能退出,即只有当 $a_1, a_2, \cdots, a_{n-1}$ 都退出队列后,a_n 才能退出队列。图 3-21 是队列的示意图。

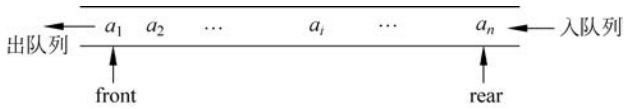

图 3-21　队列

例如,在日常生活中,人们在医院排队挂号就是一个队列。新来挂号的人到队尾排队,形成新的队尾,即入队;在队首的人挂完号离开,即出队。在程序设计中也经常会遇到排队等待服务的问题,一个典型的例子就是操作系统中的多任务处理。在计算机系统中,同时有多个任务等待输出,此时要按照请求输出的先后顺序进行输出。

2. 队列的抽象数据类型

队列的抽象数据类型定义描述如表 3-6 所示。

表 3-6 队列的抽象数据类型定义描述

数据对象	队列的数据对象集合为$\{a_1, a_2, \cdots, a_n\}$,每个元素都具有相同的数据类型		
数据关系	队列中的数据元素之间是一对一的关系。除第一个元素 a_1 外,每个元素有且只有一个直接前驱元素;除最后一个元素 a_n 外,每个元素有且只有一个直接后继元素。这些元素只能在队列的特定端进行相应的操作		
基本操作	操作名称	操作说明	举例
	InitQueue(&Q)	初始化操作,建立一个空队列 Q	这就像医院新增一个挂号窗口,前来看病的人可以排队在这里挂号看病
	QueueEmpty(Q)	若 Q 为空队列,则返回 true,否则返回 false	这就像挂号窗口前是否有人排队挂号
	EnQueue(&Q,e)	在队列 Q 的队尾插入元素 e	这就像前来挂号的人都要到队列的最后排队挂号
	DeQueue(&Q,&e)	删除 Q 的队首元素,并用 e 返回其值	这就像排在最前面的人挂完号离开队列
	Gethead(Q,&e)	用 e 返回 Q 的队首元素	这就像询问排队挂号的人的相关信息
	ClearQueue(&Q)	将队列 Q 清空	这就像所有排队的人都挂完号并离开队列


视频讲解

3.3.2 队列的顺序存储及实现

队列的存储表示有两种,分别为顺序存储和链式存储。采用顺序存储结构的队列称为顺序队列,采用链式存储结构的队列称为链式队列。

1. 顺序队列的表示

顺序队列通常采用一维数组依次存放从队首到队尾的元素。使用两个指针分别指示数组中存放的第一个元素和最后一个元素的位置,指向第一个元素的指针称为队首指针 front,指向最后一个元素的指针称为队尾指针 rear。

元素 a、b、c、d、e、f、g 依次进入队列后的状态如图 3-22 所示。元素 a 存放在数组下标为 0 的存储单元中,g 存放在下标为 6 的存储单元中,队首指针 front 指向第一个元素 a,队尾指针 rear 指向最后一个元素 g 的下一位置。

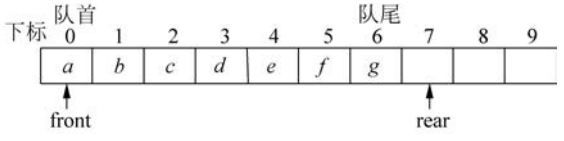

图 3-22 顺序队列

在使用队列前,要先初始化队列,此时队列为空,队首指针 front 和队尾指针 rear 都指向队列的第一个位置,即 front=rear=0,如图 3-23 所示。

当一个元素进入队列时,队尾指针 rear 加 1。若元素 a、b、c 依次进入空队列,则 front

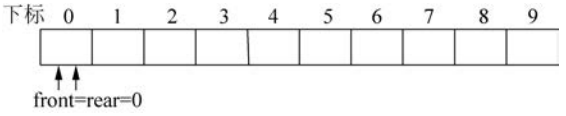

图 3-23　顺序队列为空

指向第一个元素，rear 指向下标为 3 的存储单元，如图 3-24 所示。

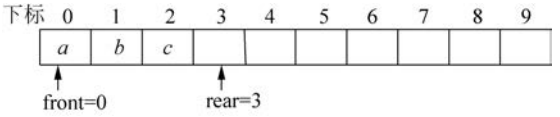

图 3-24　插入 3 个元素后的顺序队列

当一个元素退出队列时，队首指针 front 加 1。队首元素 a 出队后，front 向后移动一个位置，指向下一个位置，rear 不变，如图 3-25 所示。

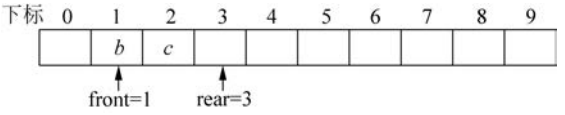

图 3-25　删除队首元素 a 后的顺序队列

注意　在非空队列中，队首指针 front 指向队首元素的位置，队尾指针 rear 指向队尾元素的下一个位置；队满指的是元素占据了队列中的所有存储空间，没有空闲的存储空间可以插入元素；队空指的是队列中没有一个元素，也叫空队列。

2. 顺序队列的“假溢出”

在对顺序队列进行插入和删除操作的过程中，可能会出现“假溢出”现象。经过多次插入和删除操作后，实际上队列还有存储空间，但是又无法向队列中插入元素，将这种溢出称为“假溢出”。

例如，将图 3-25 所示的队列进行一次出队操作，在依次将元素 d、e、f、g、h、i 入队后，若再将元素 j 入队，队尾指针 rear 将越出数组下界，从而造成“假溢出”，如图 3-26 所示。

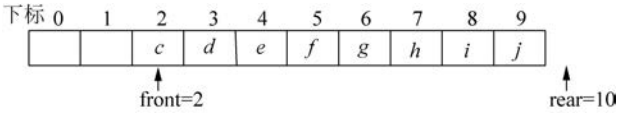

图 3-26　顺序队列的“假溢出”

3. 顺序循环队列的表示与基本运算

为了避免出现顺序队列的“假溢出”，通常采用顺序循环队列实现队列的顺序存储。

1）顺序循环队列的表示

为了充分利用存储空间，消除这种“假溢出”现象，当队尾指针 rear 和队首指针 front 到达存储空间的最大值（假定队列的存储空间为 QUEUESIZE）时，将队尾指针和队首指针转换为 0，这样就可以将元素插入队列还没有利用的存储单元中。例如，在图 3-27 中插入元素 j 后，rear 将变为 0，可以继续将元素插入下标为 0 的存储单元中。这样，顺序队列使用的存

储空间就可以构造成一个逻辑上首尾相连的循环队列。

当队尾指针 rear 达到最大值 QUEUESIZE－1 时,若队列中还有存储空间且要插入元素,则要将队尾指针 rear 变为 0;当队头指针 front 达到最大值 QUEUESIZE－1 时,若要将队首元素出队,则要将队首指针 front 变为 0。通过取余操作可以实现队列的首尾相连。例如,假设 QUEUESIZE＝10,当队尾指针 rear＝9 时,若要将新元素入队,则先令 rear＝(rear＋1)%10＝0,然后将元素存入队列的第 0 号单元,通过取余操作实现队列逻辑上的首尾相连。

2) 顺序循环队列的队空和队满判断

在顺序循环队列队空和队满的情况下,队首指针 front 和队尾指针 rear 会同时指向同一个位置,即 front＝＝rear,如图 3-27 所示。在队空时,有 front＝0、rear＝0,则 front＝＝rear;在队满时,也有 front＝0、rear＝0,因此 front＝＝rear。

为了区分是队空还是队满,通常采用以下两个方法。

(1) 增加一个标志位。设这个标志位为 flag,初始时,flag＝0;当进入队列成功时,flag＝1;当退出队列成功时,flag＝0。则队空的判断条件为 front＝＝rear&&flag＝＝0,队满的判断条件为 front＝＝rear&&flag＝＝1。

(2) 少用一个存储单元。队空的判断条件为 front＝＝rear,队满的判断条件为 front＝＝(rear＋1)% QUEUESIZE。那么,入队的操作语句为 rear＝(rear＋1)%QUEUESIZE,Q[rear]＝x;出队的操作语句为 front＝(front＋1)%QUEUESIZE。少用一个存储单元的顺序循环队列队满情况如图 3-28 所示。

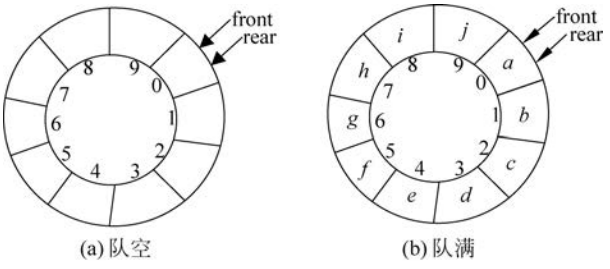

图 3-27 顺序循环队列队空和队满状态

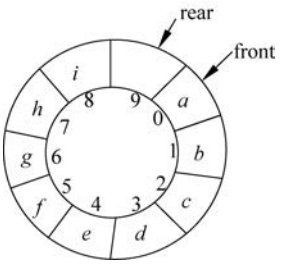

图 3-28 少用一个存储单元的顺序循环队列队满状态

顺序循环队列 SQ 的主要操作说明如下。

(1) 初始时,设置 SQ. front＝SQ. rear＝0。

(2) 循环队列队空的条件为 SQ. front＝＝SQ. rear,队满的条件为 SQ. front＝＝(SQ. rear＋1)%QUEUESIZE。

(3) 在执行入队操作时,要先判断队列是否已满。若队列未满,则将元素值 e 存入队尾指针指向的存储单元,然后将队尾指针加 1 后取模。

(4) 在执行出队操作时,要先判断队列是否为空。若队列不空,则将队首指针指向的元素值赋给 e,即取出队头元素,然后将队首指针加 1 后取模。

(5) 循环队列的长度为(SQ. rear＋QUEUESIZE－SQ. front)%QUEUESIZE。

注意 对于顺序循环队列中的入队操作和出队操作,在 front 和 rear 移动时都要进行取模运算,以避免“假溢出”。

3）顺序循环队列的基本运算

顺序循环队列的基本操作及基本操作的类方法名称如表 3-7 所示。

表 3-7　SeqQueue 类的成员函数

基 本 操 作	基本操作的类方法名称
顺序循环队列的初始化	SeQueue()
判断顺序循环队列是否为空	IsEmpty()
将元素 x 入队	EnQueue(x)
将队首元素出队	DeQueue()
取队首元素	GetHead()
求队列的长度	SeqLength()
顺序循环队列的创建	CreateSeqQueue()

（1）初始化队列。

```
public class SeQueue < T > {
    final int QUEUESIZE = 20;
    T s[];
    int front, rear;
    SeQueue() {
    //顺序循环队列的初始化
    s = (T[]) new Object[QUEUESIZE];
    front = 0;                                  //将队首指针置为 0
    rear = 0;                                   //将队尾指针置为 0
    }
}
```

（2）判断队列是否为空。若队首指针与队尾指针相等，则队列为空；否则，队列不为空。判断队列是否为空的算法实现如下。

```
public boolean IsEmpty()                        //判断顺序循环队列是否为空
{
    if (front == rear)                          //当顺序循环队列为空时
        return true;                            //返回 true
    else                                        //否则
        return false;                           //返回 false
}
```

（3）将元素 x 入队。在将元素入队（即将元素插入队尾）之前，要先判断队列是否已满。如果队列未满，则执行插入运算，然后队尾指针加 1，将队尾指针向后移动一个位置。入队操作的算法实现如下。

```
public boolean EnQueue(T x)
 //元素 e 插入顺序循环队列中，插入成功则返回 true;否则，返回 false
{
     if ((rear + 1) % QUEUESIZE != front)       //在插入新元素前，判断队尾指针是否到达队列的
                                                //最大值，即是否上溢
     {
          s[rear] = x;                          //在队尾插入元素 e
          rear = (rear + 1) % QUEUESIZE;        //将队尾指针向后移动一个位置
          return true;
     }
     else
```

```
        {
            System.out.println("当前队列已满!");
            return false;
        }
    }
```

(4) 将队首元素出队。在队首元素出队(即删除队首元素)之前,要先判断队列是否为空。若队列不空,则删除队首元素,然后将队首指针向后移动,使其指向下一个元素。出队操作的算法实现如下。

```
public T DeQueue() throws Exception
{
    //将队首元素出队,并将该元素赋值给 e
    if (front == rear)                          //判断队列是否为空
    {
        System.out.println("队列为空,出队操作失败!");
        throw new Exception("队列为空,不能进行出队操作");
    }
    else
    {
        T e = s[front];                         //将待出队的元素赋值给 e
        front = (front + 1) % QUEUESIZE;        //将队首指针向后移动一个位置,指向新的队首
        return e;                               //返回出队的元素
    }
}
```

(5) 取队首元素。先判断顺序循环队列是否为空,如果队列为空,则抛出异常表示取队首元素失败;否则,将队首元素返回,表示取队首元素成功。取队首元素的算法实现如下。

```
public T GetHead() throws Exception
{
    //取队首元素,并将该元素返回
    if(!IsEmpty())                              //若顺序循环队列不为空
        return (T)s[front];                     //返回队首元素
    else                                        //否则
    {
        throw new Exception("队列为空");
    }
}
```

(6) 获取队列的长度。

```
public int SeqLength() {
    return (rear - front + QUEUESIZE) % QUEUESIZE;
}
```

(7) 创建队列。

```
public void CreateSeqQueue()
{
    System.out.println("请输入元素(-1 作为输入结束):");
    Scanner sc = new Scanner(System.in);
    Integer data = sc.nextInt();
    while(data!= -1) {
```

```
            EnQueue(data);
            data = sc.nextInt();
        }
    }
```

4. 顺序循环队列应用示例

【例 3-10】 假设在周末舞会上，男士们和女士们进入舞厅时各自排成一队。在跳舞开始时，依次从男队和女队的队首各出一人配成舞伴。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。试编写算法模拟上述舞伴配对问题。

【分析】 根据舞伴配对原则，先入队的男士或女士先出队配成舞伴。因此该问题具有典型的先进先出特性，可用队列作为算法的数据结构。

在算法实现时，假设男士和女士的记录存放在一个数组中作为输入，然后依次扫描该数组的各元素，并根据性别来决定是进入男队还是女队。当这两个队列构造完成后，依次将两队当前的队首元素出队以配成舞伴，直至某队列变空为止。此时，若某队仍有等待配对者，算法输出此队列中等待者的人数及排在队首的等待者的名字，他（或她）将是下一轮舞曲开始时第一个获得舞伴的人。

舞伴配对问题实现代码如下。

```
import java.util.Scanner;
public class DancePartner                          //舞伴结构类型定义
{
    String name;
    String gender;
    DancePartner() {
    }
    DancePartner(String n, String g) {
        this.name = n;
        this.gender = g;
    }

    public String GetName() {
        return name;
    }

    public String GetGender() {
        return gender;
    }
    public static void DispQueue(SeQueue Q) throws Exception     //输出舞池中正在排队的男士
                                                                 //或女士
    {
        if (!Q.IsEmpty()) {
            DancePartner d = (DancePartner)Q.GetHead();
            if (d.gender.equals("男"))
                System.out.println("舞池中正在排队的男士:");
            else
                System.out.println("舞池中正在排队的女士:");
        }
```

```
        int f = Q.front;
        while (f != Q.rear) {
            DancePartner d = (DancePartner)Q.s[f];
            System.out.print(d.GetName() + " ");
            f = f + 1;
        }
        System.out.println();
    }
    public static void main(String args[]) throws Exception {
        SeQueue<DancePartner> Q1 = new SeQueue<DancePartner>();
        SeQueue<DancePartner> Q2 = new SeQueue<DancePartner>();
        System.out.println("请输入舞池中排队的人数:");          //输入舞池中排队的人数
        Scanner sc = new Scanner(System.in);
        int i,n = sc.nextInt();
        for(i = 0;i < n;i++) {
            DancePartner dancer = new DancePartner();
            System.out.println("姓名:");
            dancer.name = sc.next();        //输入姓名
            System.out.println("性别:");
            dancer.gender = sc.next();
            if (dancer.gender.equals("男"))
                Q1.EnQueue(dancer);
            else
                Q2.EnQueue(dancer);
        }
        DispQueue(Q1);
        DispQueue(Q2);
        System.out.println("舞池中的舞伴配对方式:");
        while(!Q1.IsEmpty() && !Q2.IsEmpty()) {
            DancePartner dancer1 = Q1.DeQueue();
            DancePartner dancer2 = Q2.DeQueue();
            System.out.println("(" + dancer1.GetName() + "," + dancer2.GetName() + ")"
+ " ");
        }
        if(!Q1.IsEmpty())
            DispQueue(Q1);
        if(!Q2.IsEmpty())
            DispQueue(Q2);
    }
}
```

程序的运行结果如下。

```
请输入舞池中排队的人数:
5
姓名:吴女士
性别:女
姓名:张先生
性别:男
姓名:赵先生
性别:男
姓名:刘女士
```

性别：女
姓名：郭女士
性别：女
舞池中正在排队的男士：
张先生 赵先生
舞池中正在排队的女士：
吴女士 刘女士 郭女士
舞池中的舞伴配对方式：
(张先生,吴女士)
(赵先生,刘女士)
舞池中正在排队的女士：
郭女士

3.3.3　队列的链式存储及实现


视频讲解

采用链式存储的队列称为链式队列或链队列。链式队列在插入和删除过程中,不需要移动大量的元素,只需要改变指针的位置即可。本节主要介绍链式队列的表示、实现及应用。

1. 链式队列的表示

顺序队列在插入和删除操作过程中需要移动大量元素,算法的效率会比较低,为了避免该问题,可采用链式存储结构表示队列。

(1) 链式队列。

链式队列通常用链表实现。一个链队列显然需要两个分别指示队首和队尾的指针(分别称为队首指针和队尾指针)才能唯一确定。与单链表类似,为了操作方便,这里给链队列添加一个头结点,并令队首指针 front 指向头结点,用队尾指针 rear 指向最后一个结点。一个不带头结点的链式队列和带头结点的链队列分别如图 3-29 和图 3-30 所示。

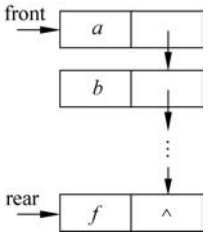

图 3-29　不带头结点的链式队列

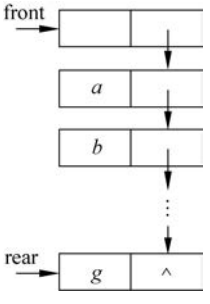

图 3-30　带头结点的链式队列

对于带头结点的链式队列,当队列为空时,队首指针 front 和队尾指针 rear 都指向头结点,如图 3-31 所示。

在链式队列中,插入和删除操作只需要移动队首指针和队尾指针,这两种操作的指针变

化如图 3-32、图 3-33 和图 3-34 所示。图 3-32 表示在队列中插入元素 a 的情况,图 3-33 表示队列中插入了元素 a、b、c 之后的情况,图 3-34 表示元素 a 出队列的情况。

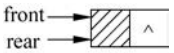

图 3-31 带头结点的空链式队列

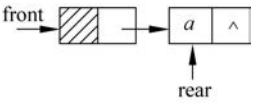

图 3-32 在链式队列中插入一个元素 *a*

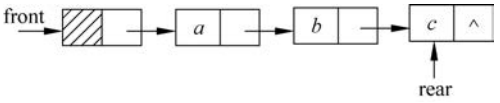

图 3-33 在链式队列中插入一个元素 *c*

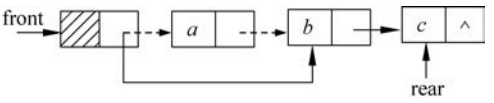

图 3-34 在链式队列中删除一个元素 *a*

链式队列的结点类型描述如下。

```
class QueueNode
{
   char data;
   QueueNode next;
   QueueNode(char data)
   {
      this.data = data;
      this.next = null;
   }
}
```

对于带头结点的链式队列,在初始时需要生成一个结点 QueueNode myQueueNode=new QueueNode('a'),然后令 front 和 rear 分别指向该结点。

(2) 链式循环队列。

将链式队列的首尾相连就构成了链式循环队列。在链式循环队列中,可以只设置队尾指针,如图 3-35 所示。当队列为空时,如图 3-36 所示,队列 LQ 为空的判断条件为 LQ.rear.next==LQ.rear。

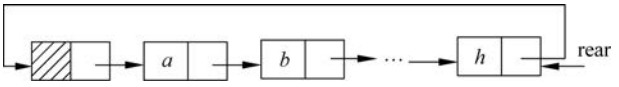

图 3-35 链式循环队列

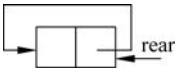

图 3-36 空链式循环队列

2. 链式队列的基本运算

链式队列的基本运算算法实现如下。

(1) 初始化队列。先生成一个 QueueNode 类型的结点,然后使 front 和 rear 分别指向该结点。

```
public class LinkQueue
{
    QueueNode front,rear;
    LinkQueue()//初始化队列
```

```
    {
        QueueNode QNode = new QueueNode('0');
        front = QNode;
        rear = QNode;
    }
}
```

（2）判断队列是否为空。

```
public boolean QueueEmpty()
//判断链式队列是否为空,队列为空则返回 true,否则返回 false
{
    if(front == rear)                    //若链式队列为空时
        return true;                     //则返回 true
    else                                 //否则
        return false;                    //返回 false
}
```

（3）将元素 *e* 入队。先生成一个新结点 pNode，再将 *e* 赋给该结点的数据域，使原队尾元素结点的指针域指向新结点，最后让队尾指针指向新结点，从而将结点加入队列中。操作过程如图 3-37 所示。

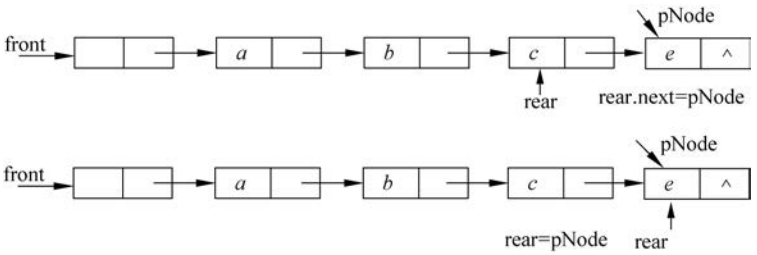

图 3-37　将元素 *e* 入队的操作过程

将元素 *e* 入队的算法实现如下。

```
public void EnQueue(char e)
//将元素 e 入队
{
    QueueNode pNode = new QueueNode(e);      //生成一个新结点,将元素值赋值给结点的数据域
    rear.next = pNode;                       //将原队列的队尾结点的指针指向新结点
    rear = pNode;                            //将队尾指针指向新结点
}
```

（4）将队首元素出队。删除队首元素时，应首先通过队首指针和队尾指针是否相等判断队列是否已空。若队列非空，则删除队首元素，然后将指向队首元素的指针向后移动，使其指向下一个元素。将队首元素出队的算法实现如下。

```
public char DeQueue() throws Exception
//将链式队列中的队首元素出队返回该元素,若队列为空,则抛出异常
{
        if(QueueEmpty())                     //在出队前,判断链式队列是否为空
        {
            throw new Exception("队列为空,不能出队操作");
        }
        else
```

```
        {
            QueueNode pNode = front.next;      //使 pNode 指向队首元素
            front.next = pNode.next;           //使头结点的 next 指向 pNode 的下一个结点
            nodal point(pNode == rear)         //如果要删除的结点是队尾,则使队尾指针指向
                                               //队头
                rear = front;
            return pNode.data;                 //返回出队元素
        }
    }
```

(5) 取队首元素。在取队首元素之前,先判断链式队列是否为空。取队首元素的算法实现如下。

```
public int getHead()
//取链式队列中的队首元素
{
    if (!QueueEmpty())                         //若链式队列不为空
        return front.next.data;                //返回队首元素
}
```

3. 链式队列应用示例

【例 3-11】 编写一个算法,判断任意给定的字符序列是否为回文。所谓回文是指一个字符序列以中间字符为基准,两边字符完全相同,即顺着看和倒着看是相同的字符序列。例如,字符序列“XYZMTATMZYX”为回文,而字符序列“XYZMYZX”不是回文。

【分析】 这个题目是典型的考查栈和队列的应用,可以通过构造栈和队列实现。具体思想:分别把字符串序列入队和入栈,根据队列的“先进先出”和栈的“后进先出”特点,依次将队列中的元素出队和出栈,出队列的元素序列仍然是原来的顺序,而出栈的字符序列刚好与原字符序列的顺序相反。这样逐个比较,若全部字符都相等,则表明该字符序列是回文;若有字符不相等,则表明该字符序列不是回文。

在具体实现时,可采用链栈和链式队列作为存储结构,算法实现如下。

```
public static void Huiwen() throws Exception {
    LinkQueue LQ1 = new LinkQueue();
    LinkQueue LQ2 = new LinkQueue();
    LinkStack2 LS1 = new LinkStack2();
    LinkStack2 LS2 = new LinkStack2();
    String str1 = new String("XYZMTATMZYX");  //回文字符序列 1
    String str2 = new String("ABCBCAB");      //回文字符序列 2
    for(int i = 0;i < str1.length();i++) {
        LQ1.EnQueue(str1.charAt(i));
        LS1.PushStack(str1.charAt(i));
    }
    for(int i = 0;i < str2.length();i++) {
        LQ2.EnQueue(str2.charAt(i));
        LS2.PushStack(str2.charAt(i));        //依次把字符序列 2 进栈
    }
    System.out.println("字符序列 1:" + str1);
    System.out.println("出队序列 出栈序列");
    while (!LS1.StackEmpty())                 //判断堆栈 1 是否为空
    {
```

```
            char q1 = LQ1.DeQueue();            //字符序列依次出队,并把出队元素赋值给 q
            char s1 = LS1.PopStack();           //字符序列出栈,并把出栈元素赋值给 s
            System.out.println(q1 + ":" + s1);
            if (q1 != s1) {
                System.out.println("字符序列 1 不是回文!");
                return;
            }
        }
        System.out.println("字符序列 1 是回文!");
        System.out.println("字符序列 2:" + str2);
        System.out.println("出队序列 出栈序列");
        while(!LS2.StackEmpty()) {
            char q2 = LQ2.DeQueue();            //字符序列依次出队,并把出队元素赋值给 q
            char s2 = LS2.PopStack();           //字符序列出栈,并把出栈元素赋值给 s
            System.out.println(q2 + ":" + s2); //输出字符序列
            if (q2 != s2) {
                System.out.println("字符序列 2 不是回文!");     //输出提示信息
                return;
            }
        }
        System.out.println("字符序列 2 是回文!");//输出提示信息
    }

    public static void main(String args[]) throws Exception {
        Huiwen();
    }
```

程序运行结果如下。

```
字符序列 1：XYZMTATMZYX
出队序列　出栈序列
   X    :    X
   Y    :    Y
   Z    :    Z
   M    :    M
   T    :    T
   A    :    A
   T    :    T
   M    :    M
   Z    :    Z
   Y    :    Y
   X    :    X
字符序列 1 是回文!
字符序列 2：ABCBCAB
出队序列　出栈序列
   A    :    B
字符序列 2 不是回文!
```

思政元素
队列“先进先出”的特点就像在日常生活中排队买票、排队上车一样,人们需要养成遵守规则、规范的良好习惯,只有这样,一切才会有章可循,社会才会井然有序。例如,尽管近年新冠肺炎病毒的肆虐给人们的生活和工作带来了诸多不便,但是在党和国家的正确领导下,全国人民严格遵守各项防疫措施,我国已经实现了新冠肺炎的阶段性胜利。在算法实现、软件开发过程中,同样需要严格遵循软件编码规范和准则,并具有精益求精、勤学精技的实践精神。只有这样,开发出的软件才会更加可靠、安全。


视频讲解

3.4 双端队列

双端队列与栈、队列类似,也是一种操作受限的线性表。本节主要介绍双端队列的定义及应用。

3.4.1 双端队列的定义

双端队列是限定插入和删除操作在表两端进行的线性表。双端队列的两端分别称为端点 1 和端点 2。双端队列可以在队列的任何一端进行插入和删除操作,而一般的队列要求在一端插入元素,在另一端删除元素。双端队列如图 3-38 所示。

图 3-38 双端队列

在图 3-38 中,可以在队列的左端或右端插入元素,也可以在队列的左端或右端删除元素。其中,end1 和 end2 分别是双端队列的指针。

在实际应用中,还有输入受限和输出受限的双端队列。所谓输入受限的双端队列是指只允许在队列的一端插入元素,而两端都能删除元素的队列。所谓输出受限的双端队列是指只允许在队列的一端删除元素,而两端都能输入元素的队列。

3.4.2 双端队列的应用

采用一个一维数组作为双端队列的数据存储结构,并编写入队算法和出队算法。双端队列为空的状态如图 3-39 所示。

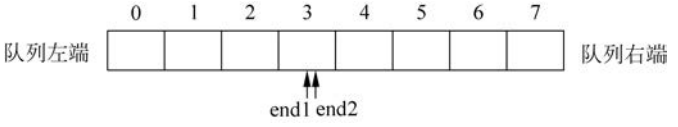

图 3-39 双端队列的初始状态(队列为空)

在实际操作过程中，用循环队列实现双端队列的操作是比较恰当的。元素 a、b、c 依次进入右端的队列，元素 d、e 依次进入左端的队列，如图 3-40 所示。

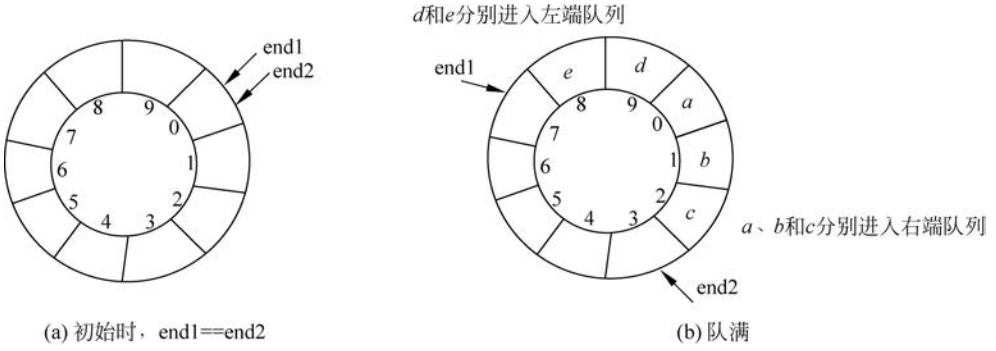

图 3-40　双端队列插入元素之后

注意　虽然双端队列是两个队列共享一个存储空间，但是每个队列只有一个指针。在算法实现过程中，需要判断入队操作和出队操作是在哪一端进行的，然后再进行插入和删除操作。

思考　栈具有“后进先出”特性，队列具有“先进先出”特性，你能举出生活中具有这些性质的例子吗？你觉得一名合格的程序员除了具备必要的专业知识外，还应该具备哪些职业素养？

3.5　实验

3.5.1　基础实验

1. 基础实验 1：实现顺序栈的基本运算

实验目的：理解顺序栈的存储结构，并能熟练掌握基本操作。

实验要求：创建一个 MySeqStack 类，该类应至少包含以下基本运算。

(1) 栈的初始化；

(2) 判断顺序栈是否为空；

(3) 入栈和出栈；

(4) 取栈顶元素；

(5) 创建栈；

(6) 输出栈中的元素。

2. 基础实验 2：实现链式栈的基本运算

实验目的：理解链式栈的存储结构，并能熟练掌握基本操作。

实验要求：创建一个 MyLinkStack 类，该类应至少包含以下基本运算。

(1) 链式栈的初始化；

(2) 判断链式栈是否为空;
(3) 入栈和出栈;
(4) 取栈顶元素;
(5) 创建栈;
(6) 销毁栈;
(7) 输出栈中元素。

3. 基础实验 3:实现顺序循环队列的基本运算

实验目的:考察是否掌握顺序队列的存储结构和基本运算。
实验要求:创建一个 MySeqQueue 类,该类应至少包含以下基本运算。
(1) 顺序循环队列的初始化;
(2) 判断队列是否为空;
(3) 入队和出队;
(4) 求队列的长度;
(5) 取队首元素。

4. 基础实验 4:实现双端链式队列的基本运算

实验目的:考察对链式队列的存储结构、双端队列的基本操作理解与掌握情况。
实验要求:创建一个 MyDLinkQueue 类,该类应至少包含以下基本运算。
(1) 双端队列的初始化;
(2) 判断队列是否为空;
(3) 入队和出队;
(4) 双端队列的创建;
(5) 双端队列的销毁。

5. 基础实验 5:利用栈将递归程序转换为非递归程序

实验目的:考察对栈和递归的理解,以及对递归程序消除的掌握。
实验要求:任意输入 n 和 m 的值,求组合数 C_n^m,其定义如下。
当 $n \geqslant 0$ 时,有 $C(n,0)=1, C(n,n)=1$。
当 $n>m$, $n \geqslant 0$, $m \geqslant 0$ 时,有 $C(n,m)=C(n-1,m)+C(n-1,m-1)$。
(1) 编写一个求解 $C(n,m)$ 的递归函数;
(2) 利用栈的基本运算,编写求解 $C(n,m)$ 的非递归算法。

3.5.2 综合实验

1. 综合实验 1:迷宫求解

实验目的:深入理解栈的存储结构,熟练掌握栈的基本操作。
实验背景:求迷宫中从入口到出口的路径是经典的程序设计问题。通常采用穷举法分析,即从入口出发,沿某一个方向向前探索,若能走通,则继续往前走;否则沿原路返回,换

另一个方向继续探索，直到探索到出口为止。为了保证在任何位置都能原路返回，显然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。

可以用如图 3-41 所示的方块表示迷宫。其中，空白方块为通道，带阴影的方块为墙。

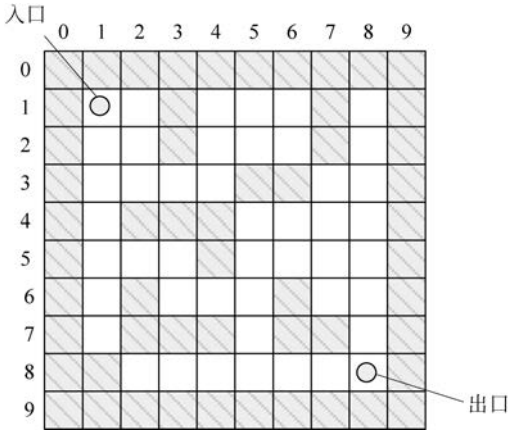

图 3-41　迷宫

所求路径必须是简单路径，即求得的路径上不能重复出现同一通道块。求迷宫中一条路径的算法的基本思想是：如果当前位置“可通”，则纳入“当前路径”，并继续朝下一个位置探索，即切换下一个位置为当前位置，如此重复直至到达出口；如果当前位置不可通，则应沿“来向”退回到前一通道块，然后朝“来向”之外的其他方向继续探索，如果该通道块的四周 4 个方块均不可通，则应从当前路径上删除该通道块。所谓下一个位置指的是当前位置四周（东、南、西、北）4 个方向上相邻的方块。

假设入口位置为（1，1），出口位置为（8，8），则根据以上算法搜索出来的一条路径如图 3-42 所示。

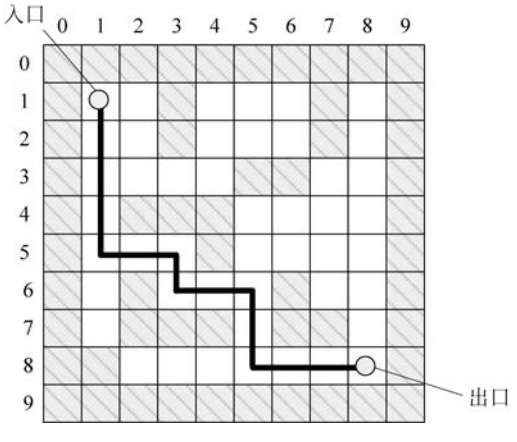

图 3-42　迷宫中的一条可通路径

实验内容：在图 3-43 所示的迷宫中，编写算法求一条从入口到出口的路径，具体要求如下。

（1）使用数组表示迷宫中的各个位置；

（2）在向前试探的过程中，利用栈保存当前的通路；

(3) 从入口到出口按照 1～n 进行增量输出,试探过的位置用－1 表示,如图 3-43 所示。

```
0  0  0  0  0  0  0  0  0  0
0  1  2  0 -1 -1 -1  0  1  0
0  1  3  0 -1 -1 -1  0  1  0
0  5  4 -1 -1  0  0  1  1  0
0  6  0  0  0  1  1  1  1  0
0  7  8  9  0  1  1  1  1  0
0  1  0 10 11 12  0  1  1  0
0  1  0  0  0 13  0  0  1  0
0  0  1  1  1 14 15 16 17  0
0  0  0  0  0  0  0  0  0  0
```

图 3-43　迷宫求解输出结果

2. 综合实验 2：模拟停车场管理

实验目的：深入理解栈、队列的存储结构,熟练掌握栈和队列的基本运算。

实验背景：停车场是一个可停放 n 辆汽车的狭长通道,且只有一个大门可供汽车进出。汽车在停车场内按车辆到达时间的先后顺序,依次由北向南排列(大门在最南端,最先到达的第一辆车停放在车场的最北端)。若停车场内已经停满 n 辆车,那么后来的车只能在门外的便道上等候。一旦有车开走,则排在便道上的第一辆车即可开入。当停车场内某辆车要离开时,在它之后进入的车辆必须先退出车场为它让路,待该辆车开出大门外,其他车辆再按原次序进入车场。每辆停放在车场的车在它离开停车场时必须按它停留的时间长短缴纳费用。

实验内容：试为停车场编制按上述要求进行管理的模拟程序。

实验提示：根据栈的后进先出和队列的先进先出的性质,需要用栈模拟停车场,用队列模拟便道,当停车场停满车后,再进入的汽车需要停在便道上。当有汽车准备停车时,判断栈是否已满,如果栈未满,则将汽车信息入栈;如果栈满,则将汽车信息入队列。当有汽车离开时,先依次将栈中的元素出栈,暂存到另一个栈中,等该车辆离开后,再将暂存栈中的元素依次存入停车场栈,并将停在便道上的汽车入栈。

设 $n=2$,输入数据为('A',1,5),('A',2,10),('D',1,15),('A',3, 20),('A',4,25),('A',5,30),('D',2,35),('D',4,40),('E',0,0)。每一组输入数据包括汽车“到达”或“离开”信息、汽车牌照号码、到达或离开的时刻 3 个数据项,其中,'A'表示到达,'D'表示离去,'E'表示输入结束。例如,('A',1,5)表示 1 号牌照车在时刻 5 到达,而('D',1,15)表示 1 号牌照车在时刻 15 离开。

小结

栈和队列是限定性线性表。栈只允许在线性表的一端进行插入和删除操作。

与线性表类似,栈也有顺序存储和链式存储两种存储方式。采用顺序存储结构的栈称为顺序栈,采用链式存储结构的栈称为链栈。

栈的后进先出特性使栈在编译处理等方面发挥了极大的作用。例如,数制转换、括号匹配、表达式求值、迷宫求解等可利用栈的后进先出特性解决。

递归的调用过程是系统借助栈的特性实现的。因此,可利用栈模拟递归调用过程,可以设置一个栈,用于存储每一层递归调用的信息,包括实际参数、局部变量及上一层的返回地

址等。每进入一层，将工作记录压入栈顶；每退出一层，将栈顶的工作记录弹出。这样就可以将递归转化为非递归，从而消除递归。

队列是只允许在表的一端进行插入操作，在另一端进行删除操作的线性表。

队列有顺序存储和链式存储两种存储方式。采用顺序存储结构的队列称为顺序队列，采用链式存储结构的队列称为链式队列。

顺序队列存在“假溢出”的问题，该问题不是因为存储空间不足而产生的。为了避免“假溢出”，可以用循环队列表示顺序队列。

为了区分循环队列是队空还是队满，通常有两种方式：设置一个标志位和少用一个存储单元。


视频讲解

习题

本书提供在线测试习题，扫描下面的二维码，可以获取本章习题。


在线测试