模块 2　栈——歌曲播放器

技能目标

- 理解栈的定义，掌握栈的特征和基本运算。
- 会使用栈的顺序存储结构解决问题。
- 能实现栈的各种基本运算。
- 会使用栈的链式存储结构解决问题。
- 能实现链栈的各种基本运算。

思维导图

本模块思维导图请扫描右侧二维码。

栈思维导图

栈是一种特殊形式的线性表，由于它们应用十分广泛，人们早已把它们单列为新的数据结构。在实际的软件开发中，栈的应用也是无处不在。比如在浏览网页时，不管什么样的浏览器都有一个“后退”键，单击后可按访问顺序的逆序加载浏览过的网页，这就是利用栈的后进先出的特点，将用户浏览的网页依次放入栈中，需要时再从栈中依次取出。此外在程序设计开发阶段，栈的应用也非常广泛，比如常见的函数调用就是通过“栈”这种数据结构来实现的。

本模块将介绍栈的基本概念、运算以及常见的实现方法，并通过一个有趣的案例介绍栈的应用。

2.1　项目描述

某公司研发的播放器软件程序，其中一个功能是：从播放列表中顺序播放歌曲。设计一个程序，模拟播放器的播放顺序功能。播放列表示意图如图 2-1 所示。

图 2-1　播放列表示意图

1. 功能描述

该播放器程序包括三个功能按钮和两个展示待播歌曲和已播歌曲的文本框。队列长度默认为 20 首歌曲。当用户单击“更新长度”按钮，可以更新歌曲的总数。单击“添加歌曲”按钮后，将把第 n 首歌曲推入栈中，并放在第 $n-1$ 首歌曲的上面，此过程显示在待播歌曲一栏；单击

“播放歌曲”按钮，则从待播歌曲栏的最上方移动一条歌曲进入已播歌曲栏，如图 2-2 所示。

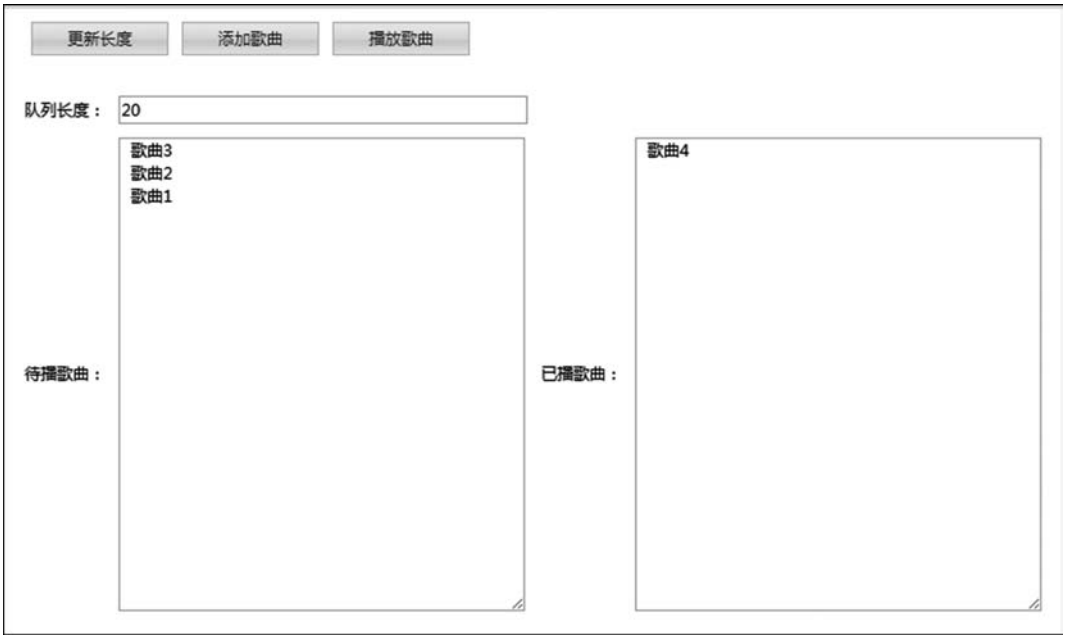

图 2-2　播放器程序界面

具体系统功能需求如下。

（1）可在待播放歌曲列表添加歌曲。

（2）当前单击播放的歌曲因为是最想听的，应先播放，即后加入歌曲列表的歌曲先播放。

（3）待播放列表设定最大长度作为任务 1，不设定最大长度作为任务 2。

（4）输入错误提示：最大长度设定值只能为数字，如输入字符，系统自动将其设置为默认值 20。

2. 设计思路

本项目利用栈结构存储待播歌曲。“添加歌曲”功能对应入栈操作，新加的歌曲放在栈顶（待播歌曲栏的最上方），“播放歌曲”功能对应出栈操作，即从栈顶取出最后放入的歌曲，添加到已播歌曲列表。

2.2　相关知识

让我们观察一下餐饮店里盘子的堆放和取用操作，可以发现以下一些特点：盘子一个个地叠放成一摞，可以看作一个由盘子组成的线性表。每次将洗净的盘子放入盘叠，总是放在最顶部，而每次用盘子时，也总是先取用盘叠最上方的那个盘子。

当我们交考试试卷时，也可以发现这种情况的存在：第一个交卷的同学卷子放在最底下，而最后一个交卷的同学卷子放在最上面。当老师改卷时，总是先从最上面的试卷开始

批阅。

从上述两个例子可以看出,在对事物的组织和管理上,采用的是同一机制,即使用一个线性表,且仅在表的一端允许插入和删除,这就是栈的概念。

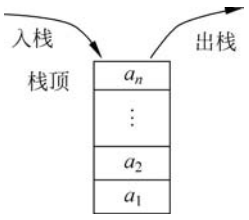

图 2-3 堆栈

2.2.1 栈的定义

栈是一种特殊的线性表,它仅允许在表的一端进行运算。在表中,允许插入和删除的一端称为“栈顶”,另一端称为“栈底”,将元素插入栈顶的操作成为“进栈”,称删除栈顶元素的操作为“出栈”,如图 2-3 所示。因为出栈操作时后进栈的元素先出,所以栈也被称为是一种“后进先出”表,简称为 LIFO(last in first out)。

2.2.2 栈的基本运算

根据实际应用,通常认为,栈应该包含了以下一些基本运算。

(1) 栈初始化:置栈为空栈。

(2) 判断栈是否为空:若栈为空,则返回 true,否则返回 false。

(3) 求栈的长度:返回栈的元素个数。

(4) 进栈:将一个元素下推进栈。

(5) 出栈:将栈顶元素托出栈。

(6) 读栈顶:返回栈顶元素。

抽象数据类型(abstract data type,ADT)是带有一组操作的一些对象的集合,在 ADT 的定义中只定义了一些基本的操作,没有提到关于这组操作是如何实现的任何解释。在 Java 中,我们用栈的接口 IStack 表示栈这些功能操作的集合。在后面的例子中,我们将实现这个接口,通过展示不同的实现代码,详细解释顺序栈和链栈的不同之处。不管怎样,只要这些类实现了栈的接口,就可以将其称为栈。

栈的接口代码如下。

【代码 2-1】

```
public interface IStack {
    public void push(Object obj) throws Exception;          //进栈
    public Object pop() throws Exception;                   //出栈
    public Object getTop() throws Exception;                //取栈顶
    public boolean isEmpty();                               //判断栈是否为空
    public int getSize();                                   //求栈长
}
```

2.2.3 顺序栈

与线性表类似,栈的存储结构也分为顺序存储结构和链式存储结构。顺序存储结构的栈简称为顺序栈,链式存储结构的栈称为链栈。

1. 顺序存储定义

与顺序线性表类似，顺序栈也需要通过一个一维数组存储元素，同时设置栈顶元素的位置下标，如图 2-4 所示。

顺序栈＝一维数组＋栈顶指示

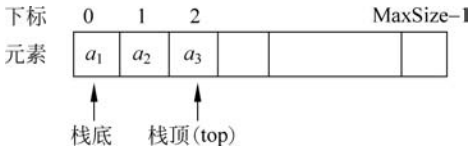

图 2-4　顺序栈的存储结构

具体地说，顺序栈用一个类 SeqStack 实现，其数据类型描述如下。

【代码 2-2】

```
public class SeqStack implements IStack {
    final int defaultSize = 10;
    int maxSize;
    int top;                        //栈顶指示
    Object[ ] stack;                //一维数组
}
```

顺序栈用一维数组 stack 存放数据，序号为 i 的元素对应数组的下标是 $i-1$，即用 stack[$i-1$]表示，栈顶用 stack[top]表示。

此外，在栈的上述存储表示下，不难得到以下栈空及栈满条件：

- 栈空条件　top＝－1；
- 栈满条件　top＝MaxSize－1。

2. 顺序栈的基本运算

根据顺序栈的运算定义，可实现顺序栈的以下操作。

1）栈初始化

栈的初始化实现比较简单，通过添加一个 initiate 方法，在该方法中将栈顶 top 的值设置为－1 即可，同时创建一个用于存储栈中元素的一维数组 stack，参数 sz 表示栈的大小。然后在构造函数中调用此方法。代码如下。

【代码 2-3】

```
public class SeqStack implements IStack {
    final int defaultSize = 10;
    int maxSize;
    int top;                        //栈顶指示
    Object[ ] stack;                //一维数组

    public SeqStack() {
        initiate(defaultSize);
    }
    public SeqStack(int sz) {
        initiate(sz);
```

```
    }
    private void initiate(int sz) {
        maxSize = sz;
        top = -1;
        stack = new Object[sz];
    }
}
```

2) 判断栈是否为空

此处实现接口中的 isEmpty 方法。在判断栈是否为空时,只需将栈顶指示 top 值与－1 相比即可,若 top 值为－1,则表示顺序栈中不包含任何元素。代码如下。

【代码 2-4】

```
public boolean isEmpty() {
    return top == -1;
}
```

3) 求栈的长度

此处实现接口中的 getSize 方法。栈的长度即为栈中数组的元素个数,因为 top 值总是指向最后一个元素,考虑到当 top 值为 0 时,已经有一个元素存在,则元素的个数为 top＋1。代码如下。

【代码 2-5】

```
public int getSize() {
    return top + 1;
}
```

4) 进栈操作

此处实现接口中的 push()方法。假设顺序栈中包含元素(a_1,a_2,a_3),当将元素 e 入栈时,实际就是要在栈顶位置插入该元素。相关过程如图 2-5 所示,具体步骤如下。

(1) 栈顶指示 top 朝栈的增长方向前进一步(即 top 值增 1)。

(2) 将元素放入栈中由当前栈顶 top 指向的位置上。

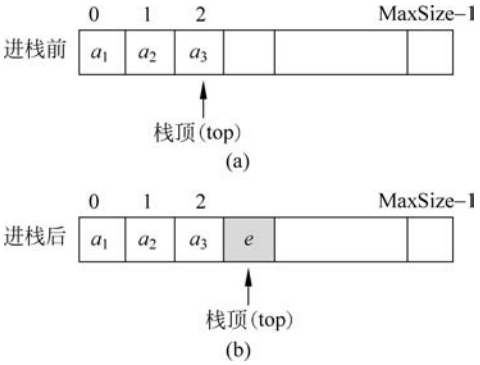

图 2-5　将元素入栈

微课 2-1　堆栈的压入和弹出

需要注意的是，在栈的这种静态实现中，进行进栈运算时，必须先进行栈满检查，以避免错误。代码如下。

【代码 2-6】

```
public void push(Object obj) throws Exception {
    if(top == maxSize - 1)
        throw new Exception("栈满,无法进栈!");
    else {
        top++;
        stack[top] = obj;
    }
}
```

5）出栈操作

此处实现接口中的 pop 方法。同样假设顺序栈中包含元素（a_1，a_2，a_3），现将 a_3 元素出栈，只需将栈顶指示 top 后退一步（即 top 值减 1）即可，如图 2-6 所示。同时若需在出栈的同时返回该出栈元素，还需通过一个临时变量获取 a_3 并返回。应该注意的是，出栈前应进行栈空检查。

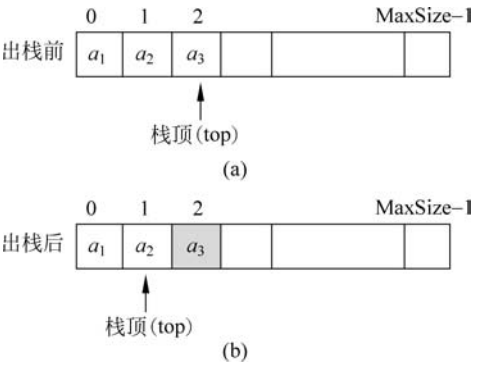

图 2-6　将元素出栈

代码如下。

【代码 2-7】

```
public Object pop() throws Exception {
    if(top == -1)
        return null;
    else {
        Object e = stack[top];
        top--;
        return(e);
    }
}
```

6）获取栈顶元素

此处实现接口中的 getTop 方法。根据栈顶（top）指示，可以直接获取最后入栈的元素。

应该注意的是,在进行读取之前,也要进行栈空检查。代码如下。

【代码 2-8】

```
public Object getTop() throws Exception {
    if(top == -1)
        return null;
    return stack[top];
}
```

7) 打印栈中所有元素

此方法非接口中定义的功能,但可以在 SeqStack 类中进行扩展。代码如下。

【代码 2-9】

```
public void print() {
    for(int i = 0;i <= top;i++)
        System.out.print(stack[i] + " ");
    System.out.println();
}
```

要测试上述这些方法,可以编写测试端代码 2-10,相关结果如图 2-7 所示。

【代码 2-10】

```
public static void main(String[] args) throws Exception{
    //创建一个栈
    SeqStack ss = new SeqStack();
    //判断是否为空
    System.out.println("是否为空:" + ss.isEmpty());
    //进栈操作,1~10 依次进栈
    for(int i = 0;i < 10;i++)
        ss.push(i + 1);
    //打印栈中元素
    ss.print();
    //显示栈顶元素
    System.out.println("当前栈顶元素为" + ss.getTop());
    //出栈 5 次,并显示每一次的出栈元素
    for(int i = 0;i < 5;i++)
        System.out.println("出栈元素为" + ss.pop());
    //显示栈长
    System.out.println("当前栈长为" + ss.getSize());
}
```

大家可能注意到,这些栈的运算都很简单,因此,在实际编程中,有时并不将这些操作设计为方法,而是直接以语句的方式操作。不过,当涉及的栈较多,或栈的元素较为复杂,或要在多个地方进行栈的操作,还是应该采用类封装的方式,将栈运算设计为方法函数,这既符合结构化程序设计的要求,也利于阅读。

图 2-7　堆栈程序运行结果

3. 顺序栈的动手实践

1）实训目的

掌握顺序栈的进栈、出栈等操作，学会较为复杂问题的求解。

2）实训内容

给定一个只包括 '('、')'、'{'、'}'、'['、']' 的字符串 s，判断字符串是否有效。有效字符串需满足：

（1）左括号必须用相同类型的右括号闭合；

（2）左括号必须以正确的顺序闭合。

相关测试用例如表 2-1 所示。

表 2-1　测试用例示例

示例 1	示例 2	示例 3	示例 4	示例 5
输入：s = "( )" 输出：true	输入：s = "( )[ ]{ }" 输出：true	输入：s = "( ]" 输出：false	输入：s = "([ )]" 输出：false	输入：s = "{ [ ] }" 输出：true

请利用栈的特性，编写相关函数实现该功能。

3）实训思路

当开始接触题目时，我们会不禁想到如果计算出左括号的数量和右括号的数量，如果每种括号左、右数量相同，会不会就是有效的括号了呢？

事实上不是的，假如输入[{]}，每种括号的左右数量分别相等，但不是有效的括号。这是因为结果还与括号的位置有关。

我们仔细分析后发现，对于有效的括号，它的部分子表达式仍然是有效的括号，比如 {()[()]} 是一个有效的括号，()[{}] 是有效的括号，[()] 也是有效的括号。并且当每次删除一个最小的括号对时，会逐渐将括号删除完，如图 2-8 所示。

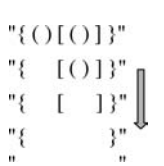

图 2-8　删除最小的括号对

这个思考的过程其实就是栈的实现过程。因此使用栈并当

遇到匹配的最小括号对时,将这对括号从栈中删除(即出栈)。如果最后栈为空,那么它是有效的括号,反之不是。

图 2-9 演示了使用栈进行括号匹配的过程。

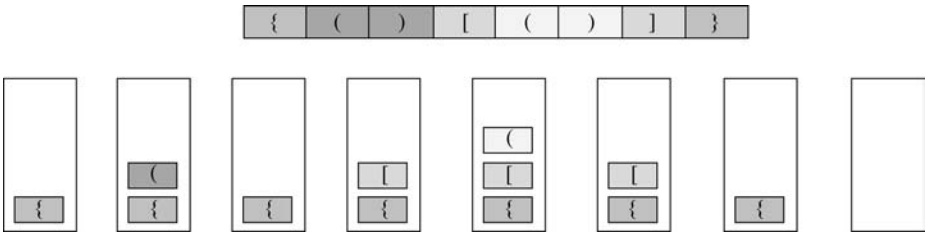

图 2-9 括号匹配栈的原理图

4) 关键代码

请读者理解如下代码并填空,运行得到相应结果。

【代码 2-11】

```
public static boolean isValid(String s) throws Exception {
    SeqStack stk = new SeqStack(s.length());
    for (char c : s.toCharArray())
        //如果 c 是“({[”,则入栈
        if (c == '(' || c == '{' || c == '[')          ①
        //如果 c 是“)}]”且栈不为空,则判断栈顶是否为对应的左括号,是则出栈,否则返回 fasle
        else if (c == ')' && !stk.isEmpty() && (char) stk.getTop() == '(') {
                                    ②
        } else if (c == '}' && !stk.isEmpty() && (char) stk.getTop() == '{') {
                                    ③
        } else if (c == ']' && !stk.isEmpty() && (char) stk.getTop() == '[') {
                                    ④
        } else {
            //如果 c 是“)}]”,栈为空,那么返回 false
            //如果 c 是“)}]”,栈不为空,但是栈顶不是与 c 对应的左括号,那么返回 false
            return false;
        }
    //如“(){}[”,如果最后栈不为空,那么就是有多余的左括号了
    return stk.isEmpty();
}
```

参考答案:①stk.push(c);②stk、pop();③stk.pop();④stk.pop();

测试端代码如下。

【代码 2-12】

```
public static void main(String[] args) throws Exception {
    Scanner input = new Scanner(System.in);
    System.out.println("请输入一个字符串:");
    String text = input.next();
    boolean flag = isValid(text);
```

```
        if (flag)
            System.out.println(text + "括号匹配");
        else
            System.out.println(text + "括号不匹配");
    }
```

5）运行结果

括号匹配程序运行结果如图 2-10 所示。

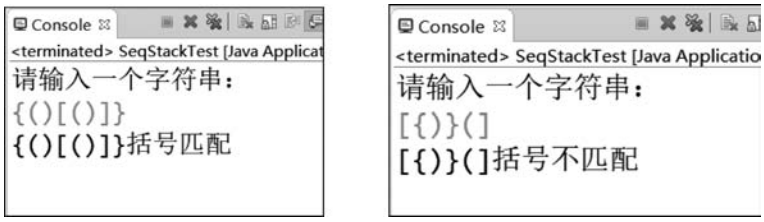

图 2-10　括号匹配程序运行结果

2.2.4　链栈

前面已经讨论了栈的顺序存储实现。通常，在顺序存储实现下，每个栈都需要按最大需求留足存储空间，这必将造成存储空间的大量浪费。解决此问题的方法之一就是采取栈的链式存储实现。

1. 栈的链式存储结构

栈的链式存储表示也称为链栈，它实际上是一个单链表，并以其链头指针作为栈顶指针。图 2-11 给出了链栈的结构示意图。因此，进出栈的运算都只能在链头进行。即

链栈 = 单链表 + 栈顶指针

具体地说，链栈 LinkStack 类同样实现了 IStack 接口，其数据类型描述如下。

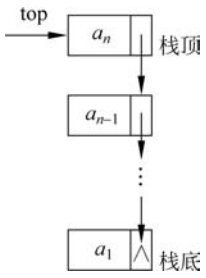

图 2-11　链栈结构示意图

【代码 2-13】

```
public class LinkStack implements IStack {
    Node top;              //栈顶结点
    int size;              //结点个数
}
```

其中结点类 Node 符合单链表中结点类的设计思路，即由两个类成员组成，一个表示元素本身，一个存储下一个结点的引用，其类型描述如下。

【代码 2-14】

```
public class Node{
    Object data;                  //数据元素
```

```
    Node next;                          //表示下一个结点的对象引用
    Node(Node nextval){                 //用于头结点的构造函数 1
        next = nextval;
    }
    Node(Object obj,Node nextval){  //用于其他结点的构造函数 2
        data = obj;
        next = nextval;
    }
}
```

此外,在链栈的上述存储表示情况下,不难得到以下栈空条件。

```
top == null;
```

2. 链栈的基本运算

根据链栈的运算定义,可实现链栈的以下操作。

1) 栈初始化

栈的初始化实现比较简单,通过添加一个构造函数即可实现,代码如下。

【代码 2-15】

```
public class LinkStack implements IStack {
     Node top;                  //栈顶结点
     int size;                  //结点个数
    //构造函数
    public LinkStack() {
        top = null;             //空栈
        size = 0;
    }

}
```

2) 判断栈是否为空。

此处实现接口中的 isEmpty 方法。在判断栈是否为空时,只需将栈顶结点 top 与 null 相比即可,代码如下。

【代码 2-16】

```
public boolean isEmpty() {
    return top == null;
}
```

3) 求栈的长度

此处实现接口中的 getSize()方法,代码如下。

【代码 2-17】

```
public int getSize() {
    return size;
}
```

4）进栈操作

此处实现接口中的 push()方法。假设元素 e 要进栈，进栈过程如图 2-12 所示。

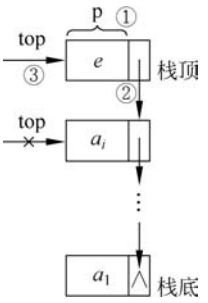

图 2-12　将元素入栈

相关的操作可按以下步骤进行。

（1）形成元素 e 对应的结点 p。

```
Node p = new Node(e,null);
```

（2）p 结点指向原栈顶结点 top。

```
p.next = top;
```

（3）栈顶结点 top 指向 p 结点。

```
top = p;
```

此外，当进栈时，表示栈长的 size 也要加 1。代码如下。

【代码 2-18】

```
public void push(Object e) throws Exception {
    Node p = new Node(e,null);
    p.next = top;
    top = p;
    size++;
}
```

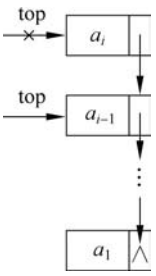

图 2-13　将元素出栈

5）出栈操作

此处实现接口中的 pop 方法。假设栈顶结点 top 要出栈，出栈过程如图 2-13 所示。

出栈操作可按照以下步骤进行。

（1）先判断栈是否为空，为空则无法进行出栈操作，抛出一个异常：

```
if(size == 0)
    throw new Exception("堆栈已空");
```

（2）获取当前栈顶结点 top 的元素值，用于返回该出栈元素：

```
Object obj = top.data;
```

（3）栈顶结点 top 沿链指向下一结点；释放之前 top 的存储空间：

```
top = top.next;
```

此外，当出栈时，表示栈长的 size 也要减 1。代码如下。

【代码 2-19】

```
public Object pop() throws Exception {
    if(size == 0)
        throw new Exception("堆栈已空");
    Object obj = top.data;
    top = top.next;
    size -- ;
    return obj;
}
```

6) 获取栈顶元素

此处实现接口中的 getTop 方法。此处直接返回栈顶结点 top 的数值域部分即可,代码如下。

【代码 2-20】

```
public Object getTop() throws Exception {
    return top.data;
}
```

7) 打印栈中所有元素

此方法非接口中定义的功能,可利用单链表的特点进行栈的元素遍历,实现代码如下。

【代码 2-21】

```
public void print() {
    Node curr = top;
    while(curr!= null) {
        System.out.print(curr.data + " ");
        curr = curr.next;
    }
    System.out.println();
}
```

要测试上述这些方法,可以编写测试端代码如下,相关结果如图 2-14 所示。

【代码 2-22】

```
public static void main(String[ ] args) throws Exception{
    //创建一个栈类
    LinkStack ls = new LinkStack();
    //判断是否为空
    System.out.println("是否为空:" + ls.isEmpty());
    //进栈操作,10 个随机数进栈
    Random rnd = new Random();
    for(int i = 0;i < 10;i++)
        ls.push(rnd.nextInt(100));
    //显示栈中元素
    ls.print();
```

```
        //显示栈顶元素
        System.out.println("当前栈顶元素为" + ls.getTop());
        //出栈 5 次,并显示每一次的出栈元素
        for(int i = 0;i < 5;i++)
            System.out.println("出栈元素为" + ls.pop());
        //显示栈长
        System.out.println("当前栈长为" + ls.size());
    }
```

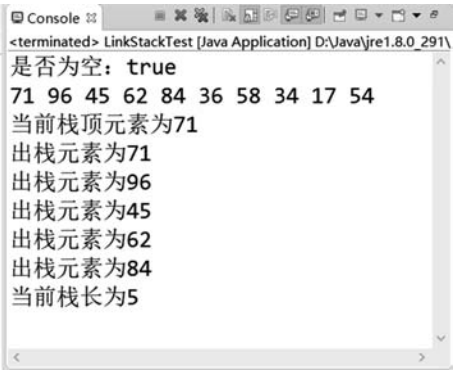

图 2-14　运行结果

需要注意的是,图 2-13 与图 2-14 显示栈中元素的效果有所区别。在顺序栈的遍历中,是从下标为 0 的数组进行遍历,因此出栈元素为遍历序列的最后一个元素。而链栈的遍历中,从 top 栈顶开始进行遍历,因此出栈元素为遍历序列的第一个元素。

3. 链栈的动手实践

1）实训目的

掌握链栈的进栈、出栈等操作,学会较为复杂问题的求解。

2）实训内容

十进制数 N 和其他 d 进制数的转换是计算机实现计算的基本问题,其解决方法很多,其中一个简单算法基于下列原理：

$$N = (N \text{ div } d) \times d + N \text{ mod } d$$

其中,div 为整除运算；mod 为求余运算。

假设现要编制一个满足下列要求的程序：对于输入的任意一个非负十进制整数 N,选择一个要转换的进制 d,打印输出与其等值的 d 进制数。

3）实训思路

需要先了解一下这几种数据类型之间的转换规则,如图 2-15 所示。

在上述计算过程中,第一次求出的值为最低位,最后一次求出的值为最高位。而打印时应从高位到低位进行,恰好与计算过程相反。根据这个特点,可以通过入栈出栈来实现,即将计算过程中依次得到的 d 进制数码按顺序进栈；计算结束后,再顺序出栈,并按出栈顺序打印输出,即可得到给定的二进制数。这是利用栈后进先出特性的经典例子。

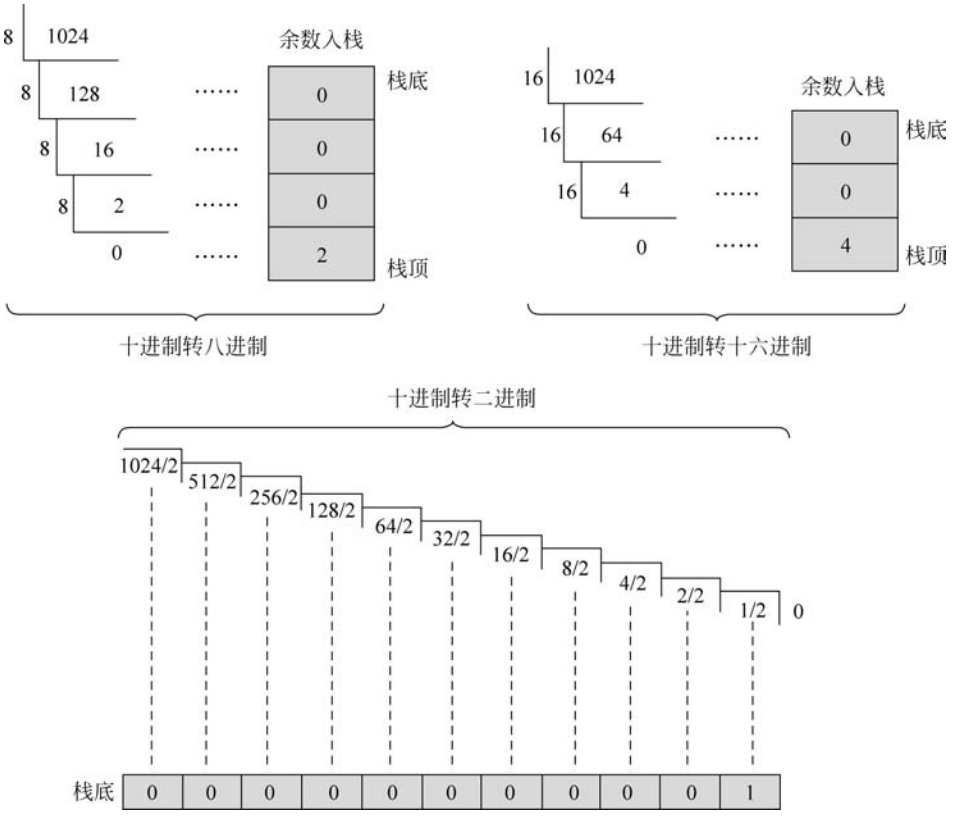

图 2-15 括号匹配过程

4) 关键代码

请读者理解以下代码并填空,运行得到相应结果。

【代码 2-23】

```
public static void dataConversion(int N,int d) throws Exception{
    LinkStack ls = new LinkStack();
    while(N!= 0){
        int x = N % d;
        ____①____                    //将 x 入栈
        N = N/d;
    }
    ____②____                        //打印 ls
}
```

参考答案:①ls. push(x);②ls. print();

测试端代码如下。

【代码 2-24】

```
public static void main(String[ ] args) throws Exception {
    Scanner input = new Scanner(System. in);
```

```
        System.out.println("请输入待转换的十进制正整数:");
        int number = input.nextInt();
        System.out.println("请输入要转换的进制:");
        int type = input.nextInt();
        System.out.println("转换结果");
        dataConversion(number,type);
    }
```

5）运行结果

运行结果如图 2-16 所示。

图 2-16　进制转换程序运行结果

2.3　项目实现

任务 1　限制曲数的歌曲播放器

这个任务要求限制曲数，考虑用顺序栈实现，曲数可由用户手动输入限制，如图 2-2 所示。用顺序栈实现点歌程序代码如下。

【代码 2-25】

```
public class PutPlateAction extends BaseAction {

//顺序栈
private int listCount = 1;
private SeqStack seqStack;
private List < String > listStack;

/ *
  * 设置长度
  * /
    public ActionForward toPutPlateList(ActionMapping mapping,
        ActionForm form, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
    listCount = 1;
    int length = Integer.valueOf(request.getParameter("length"));
    seqStack = new SeqStack(length);
```

```
        listStack = new ArrayList<String>();
        request.setAttribute("length", length);
        return new ActionForward("/pages/three/putPlateList.jsp");
    }
    /*
     * 更新长度
     */
    public ActionForward ajaxUpdateLength(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        listCount = 1;
        int length = Integer.valueOf(request.getParameter("length"));
        seqStack = new SeqStack(length);
        listStack = new ArrayList<String>();
        return null;
    }
    /*
     * 装载已播歌曲
     */
    public ActionForward ajaxLoadList(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        List<ShowBean> list = seqStack.getObjs();
        for(String str:listStack){
            list.add(new ShowBean(str, true));
        }
        renderText(response, getJSON(list));
        return null;
    }
    /*
     * 添加歌曲
     */
    public ActionForward ajaxPushList(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        String obj = "歌曲" + listCount;
        try {
            seqStack.push(obj);                    //进入待播栈
        } catch (Exception e) {
            renderText(response, "1");
            return null;
        }
        listCount++;                               //待播歌曲加 1
        renderText(response, "0");
        return null;
    }
    /*
     * 播放歌曲
     */
    public ActionForward ajaxPopList(ActionMapping mapping,
```

```
        ActionForm form, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
    if(!seqStack.notEmpty()){              //待播列表为空则返回
        renderText(response, "1");
        return null;
    }
    String obj = seqStack.pop() + "";      //获取栈顶的待播歌曲
    listStack.add(obj);                    //添加到已播歌曲列表
    renderText(response, "0");
    return null;
}
```

任务 2　不限制曲数的歌曲播放器

完成任务 1 后，客户提出播放列表不需要设定播放歌曲长度。请重新设计程序，模拟该播放器顺序播放功能。

该播放器程序包括两个功能按钮和两个展示待播歌曲和已播歌曲的文本框。当用户单击“添加歌曲”按钮后，将把第 n 首歌曲推入栈中，并放在第 $n-1$ 首歌曲的上面，此过程显示在待播歌曲一栏；单击“播放歌曲”按钮，则从待播歌曲栏的最上方移动一条歌曲进入已播歌曲栏，程序界面如图 2-17 所示。

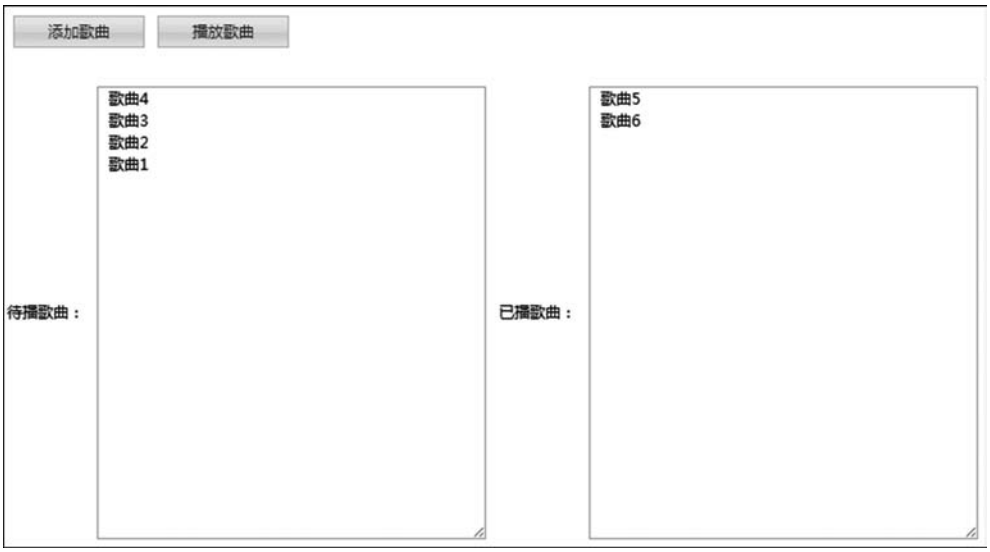

图 2-17　不限歌曲长度的播放器程序界面

观察图 2-17 发现，任务 2 和任务 1 的区别在于，没有限制待播歌曲的个数。而在之前的项目实现中，采用顺序栈的方式实现，这种方式需要用到数组，而数组是需要事先给定大小的，这也要求在使用顺序栈之前，先要预设一个最大的栈空间。接下来，我们将采用链栈的方式完成任务 2。链栈采用单链表的方式实现栈的特点，因此理论上无须考虑栈的最大空间，其大小会随着链表的元素自动变化。

用链栈实现不限制歌曲数的点歌程序代码如下。

【代码 2-26】

```
public class PutPlateAction extends BaseAction {
    //链栈
    private int linCount = 1;
    private LinStack linStack;
    private List < String > linList;

    public ActionForward toPutPlateLin(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        linCount = 1;
        linStack = new LinStack();
        linList = new ArrayList < String >();
        return new ActionForward("/pages/three/putPlateLin.jsp");
    }
    public ActionForward ajaxLoadLin(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        List < ShowBean > list = linStack.getObjs();
        for(String str:linList){
            list.add(new ShowBean(str, true));
        }
        renderText(response, getJSON(list));
        return null;
    }
    public ActionForward ajaxPushLin(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        String obj = "歌曲" + linCount;
        try {
            linStack.push(obj);
        } catch (Exception e) {
            renderText(response, "1");
            return null;
        }
        linCount++;
        renderText(response, "0");
        return null;
    }
    public ActionForward ajaxPopLin(ActionMapping mapping,
            ActionForm form, HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        if(!linStack.notEmpty()){
            renderText(response, "1");
            return null;
        }
        String obj = linStack.pop() + "";
        linList.add(obj);
        renderText(response, "0");
```

```
            return null;
        }
    }
```

2.4　小结

本模块讲解了栈的定义。栈的特点是先进后出，插入和删除的操作都是只能在栈的一端进行。接着分别介绍了顺序栈和链栈的存储结构和运算。通过播放器项目和动手实践等相关应用案例，加深了读者对栈概念的理解。

2.5　习题

1. 填空题

（1）设有一个空栈，现有输入序列为“1，2，3，4，5”，经过操作序列 push、pop、push、pop、push、push、pop 后，现在已出栈的序列为_____。

（2）设有栈 s，若线性表元素入栈顺序为“1，2，3，4”，得到的出栈序列为“1，3，4，2”，则用栈的基本运算 push、pop 描述的操作序列为_____。

（3）栈是限定仅在表尾进行插入或删除操作的线性表，其运算遵循_____的原则。

（4）在顺序栈中，当栈顶指示 top＝－1 时，表示_____；当 top＝MaxSize－1 时，表示_____。

（5）在顺序栈 s 中，出栈操作要执行的语句序列中有 s.top _____；进栈操作时要执行的语句序列中有 s.top _____。

（6）在链栈中，栈空的条件是 top _____。

2. 选择题

（1）栈结构通常采用的两种存储结构是（　　）。

A. 顺序存储结构和链式存储结构　　B. 散列方式和索引方式

C. 链表存储结构和数组　　D. 线性存储结构和非线性存储结构

（2）元素 a、b、c、d 依次进栈后，则栈顶元素为（　　）。

A. a　　B. b　　C. c　　D. d

（3）一个栈的进栈序列为 abcd，则栈的输出序列不可能为（　　）。

A. dcba　　B. abcd　　C. cabd　　D. cbad

（4）判断一个顺序栈 s 为空的条件是（　　）。

A. s.top＝－1　　B. s.top＝MaxSize－1

C. s.top！＝－1　　D. s.top！＝MaxSize

（5）判断一个顺序栈 s 为满的条件是（　　）。

A. s.top＝－1　　B. s.top＝MaxSize－1

C. s.top!=-1　　D. s.top!=MaxSize

(6) 向一个栈顶指针为 HS 的链栈中插入一个 s 所指结点时,则执行(　　)(不带头结点)。

A. HS.next=s;　　B. s.next= HS.next; HS.next=s;

C. s.next= HS; HS=s;　　D. s.next= HS; HS= HS.next;

(7) 从一个栈顶指针为 HS 的链栈中删除一个结点时,用 x 保存被删结点的值,则执行(　　)(不带头结点)。

A. x=HS; HS= HS.next;　　B. x=HS.data;

C. HS= HS.next; x=HS.data;　　D. x=HS.data; HS= HS.next;

3. 简答题

(1) 铁路进行列车调度时,常把站台设计成栈式结构的站台,如图 2-18 所示。试问:

① 设有编号为 1,2,3,4,5,6 的六辆列车,顺序开入栈式结构的站台,则可能的出栈序列有多少种?(2 分)

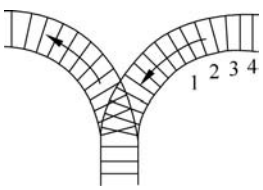

图 2-18　栈式结构的站台

② 若进站的六辆列车顺序如上所述,那么是否能够得到 435612、325641、154623 和 135426 的出站序列,如果不能,说明为什么不能;如果能,说明如何得到(即写出“进栈”或“出栈”的序列)(10 分,写对能和不能,得 8 分;理由 2 分)。

(2) 设计一种算法,能判断一个算术表达式中的圆括号配对是否正确(提示:对表达式进行扫描,凡遇到“(”就进栈,遇到“)”就退出栈顶的“(”。表达式扫描完毕时,栈若为空,则圆括号配对正确。顺序栈的定义 SeqStack 如下。

```
public class SeqStack {
    final int defaultSize = 10;
    int top;                    //栈顶指示
    Object[] stack;             //数组对象
    int maxStackSize;           //最大数据元素个数
    public SeqStack() {
        initiate(defaultSize);
    }
    //构造函数
    public SeqStack(int sz) {
        initiate(sz);
    }
    //初始化
    private void initiate(int sz) {
        maxStackSize = sz;
        top = 0;
        stack = new Object[sz];
    }
    //入栈
    public void push(Object obj) throws Exception {
```

```
            if (top == maxStackSize) {
                throw new Exception("堆栈已满!");
            }
            stack[top] = obj;          //保存栈顶元素
            top++;                     //产生新栈顶指示
        }
        //出栈
        public Object pop() throws Exception {
            if (top == 0) {
                throw new Exception("堆栈已空!");
            }
            top--;
            return stack[top];
        }
        //取栈顶元素
        public Object getTop() throws Exception {
            if (top == 0) {
                throw new Exception("堆栈已空!");
            }
            return stack[top - 1];
        }
        //判断堆栈是否为空
        public boolean notEmpty() {
            return (top > 0);
        }
    }
```