

# 云南大学数学与统计学院

## 上机实践报告

课程名称：数据结构与算法实验	年级：2013	上机实践成绩：
指导教师：陆正福	姓名：金洋	
上机实践名称：栈与队列实验	学号：20131910023	上机实践日期:2015.4.30
上机实践编号：No. 5	组号：4	上机实践时间:16:05

### 一、实验目的

- 1.熟悉与栈、队列等有关的数据结构与算法
- 2.熟悉主讲教材 Chapter 5 的代码片段
- 3.熟悉实验教材第 2 章的问题

### 二、实验内容

1. 线性表有关的数据结构设计 with 算法设计
2. 调试主讲教材 Chapter 5 的 Java 程序
3. 阅读实验教材第 2 章的问题，将 C 程序转化为 Java 程序（每组至少完成 1 个问题）

### 三、实验平台

个人计算机； Oracle/Sun Java 7 SE 或 EE

### 四、实验记录与实验结果分析

（注意记录实验中遇到的问题。实验报告的评分依据之一是实验记录的细致程度、实验过程的真实性、实验结果的解释和分析。如果涉及实验结果截屏，应选择白底黑字。）

#### 1. 数组栈（顺序栈）

##### FullStackException.java

```
public class FullStackException extends RuntimeException{
    public FullStackException(String err){
        super(err);
    }
}
```

##### EmptyStackException.java

```
public class EmptyStackException extends RuntimeException{
    public EmptyStackException(String err){
        super(err);
    }
}
```

Stack.java

```
public interface Stack<E> {  
    public int size();  
    public boolean isEmpty();  
    public E top();  
    public void push(E element) throws FullStackException;  
    public E pop() throws EmptyStackException;  
  
}
```

ArrayStack.java

```
public class ArrayStack<E> implements Stack<E>{  
    protected int capacity;//实际数组容量  
    public static final int CAPACITY=1000;  
    protected E S[];  
    protected int top=-1;  
    public ArrayStack(){  
        this(CAPACITY);  
    }  
    public ArrayStack(int capacity2) {  
        capacity=capacity2;  
        //泛型类中的泛型数据只能调用Object类中的方法  
        S=(E[]) new Object[capacity];  
    }  
    public int size(){  
        return(top+1);  
    }  
    public boolean isEmpty(){  
        return(top<0);  
    }  
    public void push(E ele) throws FullStackException{  
        if (size()==capacity)  
            throw new FullStackException("Stack is full!");  
        top++;  
        S[top]=ele;  
    }  
    public E top()throws EmptyStackException{  
        if (isEmpty())  
            throw new EmptyStackException("Stack is empty!");  
        return S[top];  
    }  
    public E pop()throws EmptyStackException{  
        if (isEmpty())  
            throw new EmptyStackException("Stack is empty!");  
        E topEle;  
        topEle=S[top];  
        S[top]=null;  
        top--;  
        return topEle;  
    }  
    public String toString(){
```

```

        String s;
        s="[";
        if (size()>0) s=s+S[0];
        if (size()>1)
            for (int i=1;i<size();i++){
                s=s+","+S[i];
            }
        s=s+"]";
        return s;
    }
    public void status(String op,Object element){
        System.out.print("-->" +op);
        System.out.println(",returns "+element);
        System.out.print("result;size="+size()+" ,isEmpty="+isEmpty());
        System.out.println(",stack: "+this);//?
    }
}

```

## 2. 测试数组栈

### TestArrayStack.java

```

public class TestArrayStack {

    public static void main(String[] args){
        Object o;//?
        ArrayStack<Integer> A=new ArrayStack<Integer>();
        A.status("new ArrayStack<Integer> A",null);
        A.push(7);
        A.status("A.push(7)", null);
        o=A.pop();
        A.status("A.pop()", o);
        A.push(9);
        A.status("A.push(9)", null);
        o=A.pop();
        A.status("A.pop()", o);

        ArrayStack<String> B=new ArrayStack<String>();
        B.status("new ArrayStack<String> B", null);
        B.push("Bob");
        B.status("B.push(\"Bob\")",null);
        B.push("Alice");
        B.status("B.push(\"Alice\")",null);
        o=B.pop();
        B.status("B.pop()", o);
        B.push("Eve");
        B.status("B.push(\"Eve\")",null);
    }
}

```



```

控制台
<已终止> ArrayStack [Java 应用程序] C:\Program Files\Java\jre1.5.0_12\bin\java
-->new ArrayStack<Integer> A,returns null
result;size=0,isEmpty=true,stack: []
-->A.push(7),returns null
result;size=1,isEmpty=false,stack: [7]
-->A.pop(),returns 7
result;size=0,isEmpty=true,stack: []
-->A.push(9),returns null
result;size=1,isEmpty=false,stack: [9]
-->A.pop(),returns 9
result;size=0,isEmpty=true,stack: []
-->new ArrayStack<String> B,returns null
result;size=0,isEmpty=true,stack: []
-->B.push("Bob"),returns null
result;size=1,isEmpty=false,stack: [Bob]
-->B.push("Alice"),returns null
result;size=2,isEmpty=false,stack: [Bob,Alice]
-->B.pop(),returns Alice
result;size=1,isEmpty=false,stack: [Bob]
-->B.push("Eve"),returns null
result;size=2,isEmpty=false,stack: [Bob,Eve]

```

### 3. 节点栈

#### Node.java

```

public class Node<E> {
    private E element;
    private Node<E> next;
    public Node(){
        this(null,null);
    }
    public Node(E e,Node<E> n){
        element=e;
        next=n;
    }
    public E getElement(){
        return element;
    }
    public Node<E> getNext() {
        return next;
    }
    public void setElement(E newEle){
        element=newEle;
    }
    public void setNext(Node<E> newNext){
        next=newNext;
    }
}

```

```
}
```

#### NodeStack.java

```
public class NodeStack<E> implements Stack<E>{
    protected Node<E> top;
    protected int size;
    public NodeStack(){
        top=null;
        size=0;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return top==null;
    }
    public E top() throws EmptyStackException{
        if (isEmpty()) throw new EmptyStackException("Stack is empty.");
        return top.getElement();
    }
    public void push(E element) throws FullStackException {
        Node<E> v=new Node<E>(element,top);
        top=v;
        size++;
    }
    public E pop() throws EmptyStackException {
        if (isEmpty()) throw new EmptyStackException("Stack is empty!");
        E topEle;
        topEle=top.getElement();
        top=top.getNext();
        size--;
        return topEle;
    }
}
```

#### 4. 用数组栈实现数组的逆转

```
import java.util.Arrays;
public class Stack_ReversingArray {

    public static <E> void reverse(E[] a){
        Stack<E> S=new ArrayStack<E>(a.length);
        for (int i=0;i<a.length;i++)
            S.push(a[i]);
        for (int i=0;i<a.length;i++)
```

```

        a[i]=S.pop();
    }
    public static void main(String[] args) {
        Integer[] a={2,4,6,8,10};
        String[] s={"a","b","e","f","g"};
        System.out.println("a="+Arrays.toString(a));
        System.out.println("s="+Arrays.toString(s));
        System.out.println("After Reversing:");
        reverse(a);
        reverse(s);
        System.out.println("a="+Arrays.toString(a));
        System.out.println("s="+Arrays.toString(s));
    }
}

```

```

<terminated> Stack_ReversingArray [Java Application] D:\Java\bin\javaw.exe (2015年5月
a=[2, 4, 6, 8, 10]
s=[a, b, e, f, g]
After Reversing:
a=[10, 8, 6, 4, 2]
s=[g, f, e, b, a]

```

## 5. 实验教材第2章问题1：实现顺序栈的各种基本运算

在 `ArrayStack` 类中增加方法：

```

public E getStack(int i){
    return S[i];
}

```

编写如下主函数

## TestArrayStackBaseOperation.java

```

public class TestArrayStackBaseOperation {

    public static void main(String[] args) {
        System.out.println("(1)初始化栈");
        ArrayStack<Character> A=new ArrayStack<Character>(5);
        System.out.print("(2)栈为: ");
        System.out.println(A.isEmpty()? "空": "非空");
        System.out.println("(3)依次进栈元素 a,b,c,d,e");
        A.push('a');
        A.push('b');
        A.push('c');
        A.push('d');
        A.push('e');
        System.out.print("(4)栈为: ");
        System.out.println(A.isEmpty()? "空": "非空");

        System.out.println("(5)栈长度为: "+A.size());
        System.out.println("(6)取栈顶元素: "+A.top());
        System.out.print("(7)从栈顶到栈顶输出元素: ");
        int i=A.top;
        while (i!=-1){
            System.out.print(A.getStack(i));
            i--;
        }

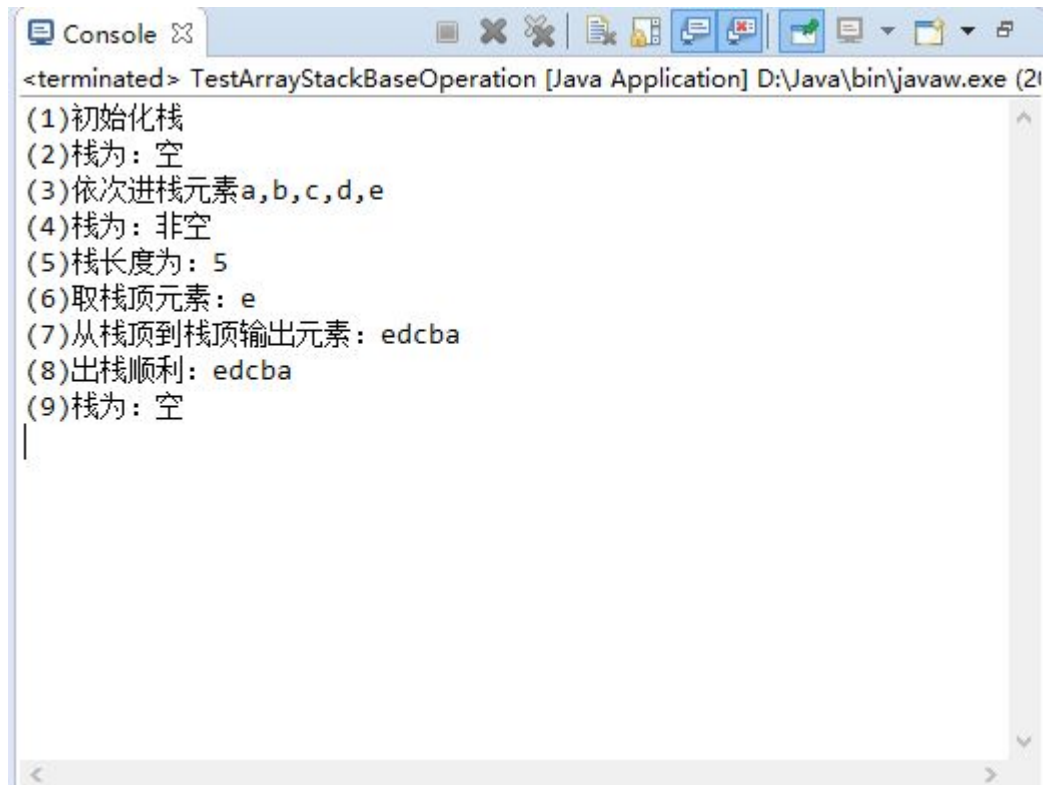
        System.out.println();
        System.out.print("(8)出栈顺利: ");
        while (!A.isEmpty()){
            System.out.print(A.pop());
        }
        System.out.println();
        System.out.print("(9)栈为: ");
        System.out.println(A.isEmpty()? "空": "非空");

    }

}

```

得到结果



对于 TestArrayStackBaseOperation.java 中的

```

System.out.print("(7)从栈顶到栈顶输出元素: ");
    int i=A.top;
    while (i!=-1){
        System.out.print(A.getStack(i));
        i--;
    }

```

如果划线部分改为 System.out.print(A.S[i]);

则会抛出异常 Exception in thread "main" java.lang.ClassCastException:  
[Ljava.lang.Object; cannot be cast to [Ljava.lang.Character;

## 6. 实验教材第 2 章问题 2: 检测表达式中出现的括号是否匹配

```

import java.io.*;
import java.util.Scanner;
public class ParenMatching {
    public static boolean isMatching(char left,char right){
        if (left=='(' && right==')') return true;
        if (left=='[' && right==']') return true;
        if (left=='{' && right=='}') return true;
        return false;
    }

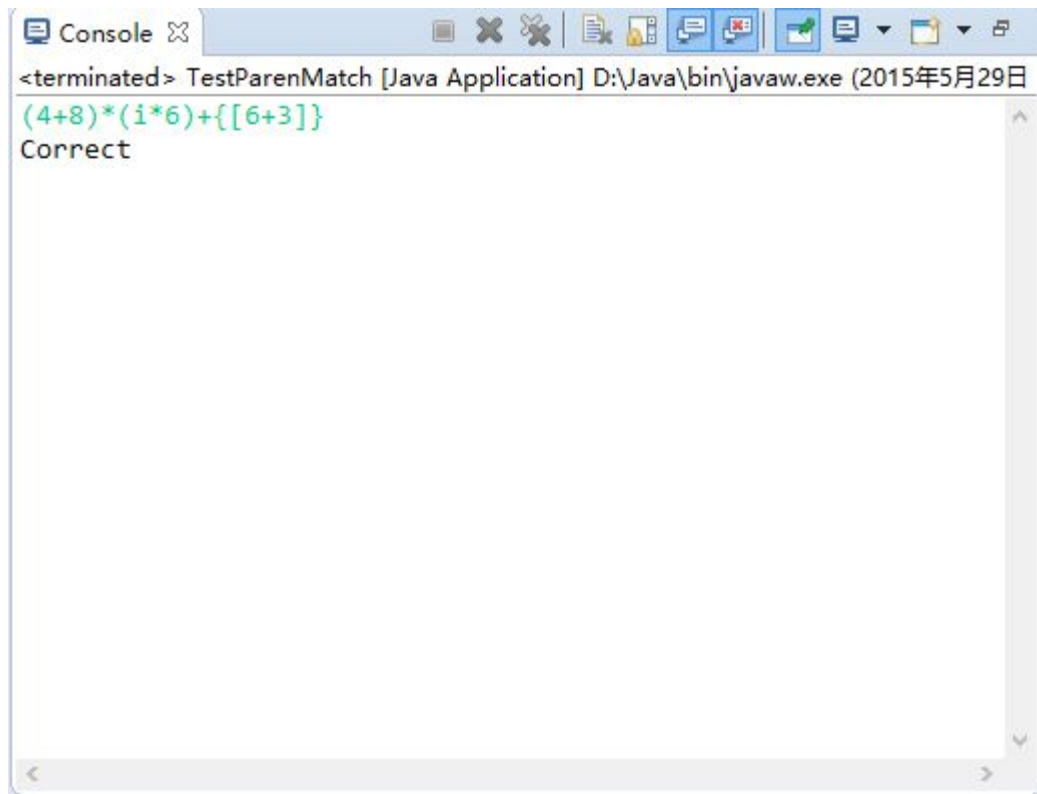
    public static boolean ParenMatch(char[] X,int n){
        ArrayStack<Character> S=new ArrayStack<Character>(n);

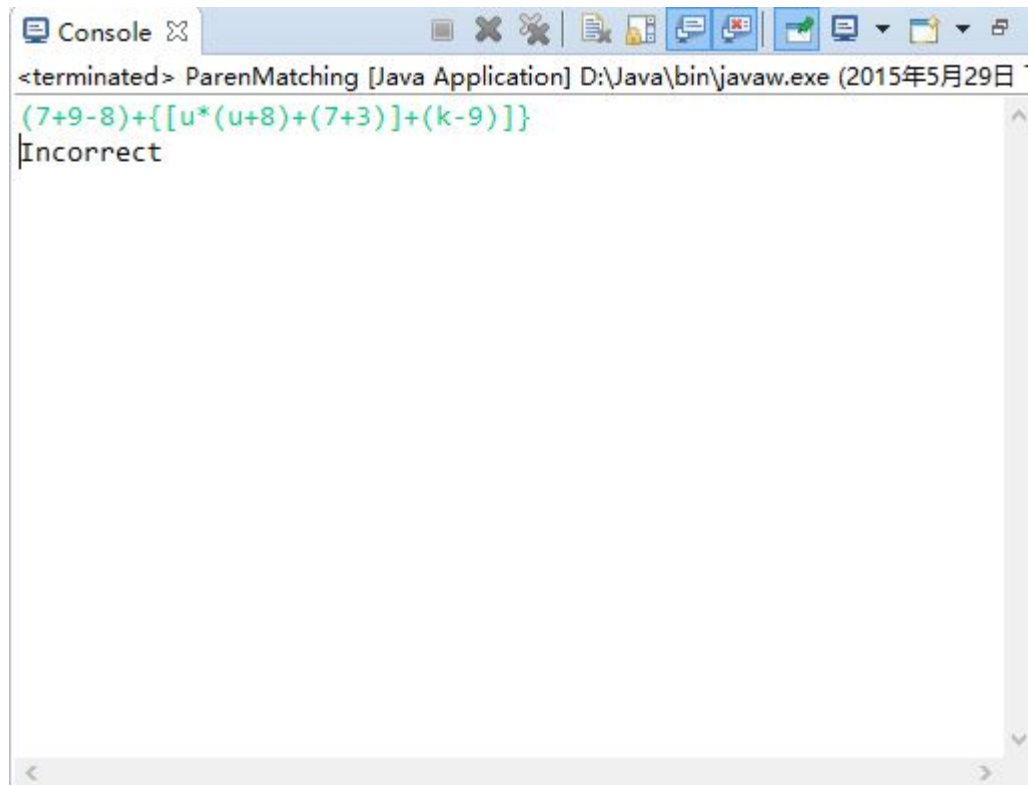
```



```
        for (int i=0;i<n;i++)
            if (X[i]=='(' || X[i]=='[' || X[i]=='{')
                S.push(X[i]);
            else if (X[i]==')' || X[i]==']' || X[i]=='}'){
                if (S.isEmpty())
                    return false;
                if (!isMatching(S.pop(),X[i]))
                    return false;
            }
        if (S.isEmpty()) return true;
        else return false;
    }
    public static void main(String[] args) {
        Scanner input=new Scanner(System.in);
        String s;
        s=input.nextLine();
        char[] X=s.toCharArray();
        System.out.println(ParenMatch(X,s.length())?"Correct":"Incorrect");
    }
}
```

测试结果:





## 7.实验教材第 2 章问题 3：实现表达式的求值

可分为两个子问题

①将中缀表达式转换为后缀表达式

### InfixToPostfix.java

```
public class InfixToPostfix {
    protected NodeStack<Character> operatorStack; //保存运算符的栈
    protected String postfixString;

    //构造方法
    public InfixToPostfix(){
        operatorStack=new NodeStack<Character>();
        postfixString=new String();
    }

    /**
     * 比较两个运算符的优先级
     * @param char pre:栈顶的运算符
     * @param char now:此时的运算符
     * @return int:pre 比 now 优先级高（同一级别，则表达式中左边的优先），返回 1；pre
     比 now 优先级低，返回-1；
     */
}
```

```

public int compareOperator(char pre, char now){
    if (pre=='(' || pre=='#') return -1;
    if (now=='+' || now=='-') return 1;
    if (pre=='*' || pre=='/') return 1;
    return -1;
}

```

```

/**
 * 将中缀表达式转化为后缀表达式
 * @param infixString
 */
public void toPostfix(String infixString){
    int i;
    char ch; //每次取出的字符

```

/\*当 ch 扫到表达式第一个运算符，需要将其入栈，但其面临与其前面的运算符比较，而其前面没有运算符，即它面临的栈是一个空栈或只存有 '(' 的栈，若不做处理，调用 compare 发放会出现意外

\* 我们考虑先将#作为栈底元素入栈，并在 compare 中增加一个判断条件：当前一个字符是 '(' 或 '#' 时...这样当第一个运算符做 compare(pre, now) 操作后，不会发生意外，令返回结果是 -1，做入栈操作

```

*/
operatorStack.push('#');
for (i=0; i<infixString.length(); i++){
    ch=infixString.charAt(i);

    //ch 是运算符
    if (ch=='+' || ch=='-' || ch=='*' || ch=='/'){
        //栈顶运算符的优先级比 ch 低，则将其直接入栈
        if (compareOperator(operatorStack.top(), ch)<0) {
            operatorStack.push(ch);
            postfixString+=' '; //在后缀表达式之尾加一个空格，以免
造成数字的混淆，如 3+75 和 37+5
        }

```

```

        else{
            //将比 ch 优先级高的运算符依次输出到表达式末尾
            while (compareOperator(operatorStack.top(), ch)>0){
                postfixString+=operatorStack.pop();
                postfixString+=' '; //在后缀表达式之尾加一个空
格，以免造成数字的混淆，如 3+75 和 37+5
            }
            operatorStack.push(ch);
        }
    }
}

```

```

//ch 是操作数，则直接将它输出到后缀表达式之尾
else if (ch>='0' && ch<='9') postfixString+=ch;

```

```

        //ch 是 '(', 则直接入栈
        else if (ch=='(') operatorStack.push(ch);

        //ch 是 ')', 则将 '(' 前的字符都放到后缀表达式之尾, 直至栈顶为 '(', 消去()
        else if (ch==')'){
            while (operatorStack.top()!='(')
                postfixString+=operatorStack.pop();
            operatorStack.pop();
        }
    }
    while(operatorStack.top()!='#')
        postfixString+=operatorStack.pop();//将存在栈中的元素弹出并加入到后缀表达式末尾
    operatorStack.pop();//将 '#' 弹出
}

/**
 * 返回后缀表达式
 * @return 后序表达式
 */
public String getpostfixString(){
    return postfixString;
}
}

```

## ②求后缀表达式的值

### CalculatePostfixExpression.java

```

public class CalculatePostfixExpression {

    NodeStack<Integer> numberStack;//保存操作数的栈
    public CalculatePostfixExpression(){
        numberStack=new NodeStack<Integer>();
    }

    /**
     * @param left: 左元
     * @param right: 右元
     * @param ch: 运算符
     * @return 两元运算的结果
     */
    public int operating(int left,int right,char ch){
        if (ch=='+') return left+right;
        if (ch=='-') return left-right;
        if (ch=='*') return left*right;
        if (ch=='/') return left/right;
        return 0;
    }
}

```

```

/**
 *
 * @param postfixString:后缀表达式的字符串
 * @return: 后缀表达式的计算结果
 */
public int calculate(String postfixString){
    int i=0;
    int right,left;
    char ch;
    while (i<postfixString.length()){
        ch=postfixString.charAt(i);
        //当前字符是运算符，连续出栈两元，先出栈的元素在右，后出栈的元素在
        //左，计算出结果后再入栈
        if (ch=='+' || ch=='-' || ch=='*' ||ch=='/') {
            right=numberStack.pop();
            left=numberStack.pop();
            numberStack.push(operating(left,right,ch));//连同操作符进
            //行计算
            i++;
        }
        //当前字符是数字，则将其后面出现的数字字符都转化为数值，再入栈
        else if (ch>='0' && ch<='9'){
            int temp=0;
            do{
                temp=temp*10+ch-48;
                i++;
            }while (i<postfixString.length()
            &&(ch=postfixString.charAt(i))>='0' &&(ch=postfixString.charAt(i))<='9');//若下
            //一个仍然是数字字符
            numberStack.push(temp);
        }

        //当前字符是空格，则下移一位
        else if (ch==' ') i++;
    }
    //最终留下的即为最终结果
    return numberStack.pop();
}
}

```

测试程序

TestCalculateInfixExpression.java

```

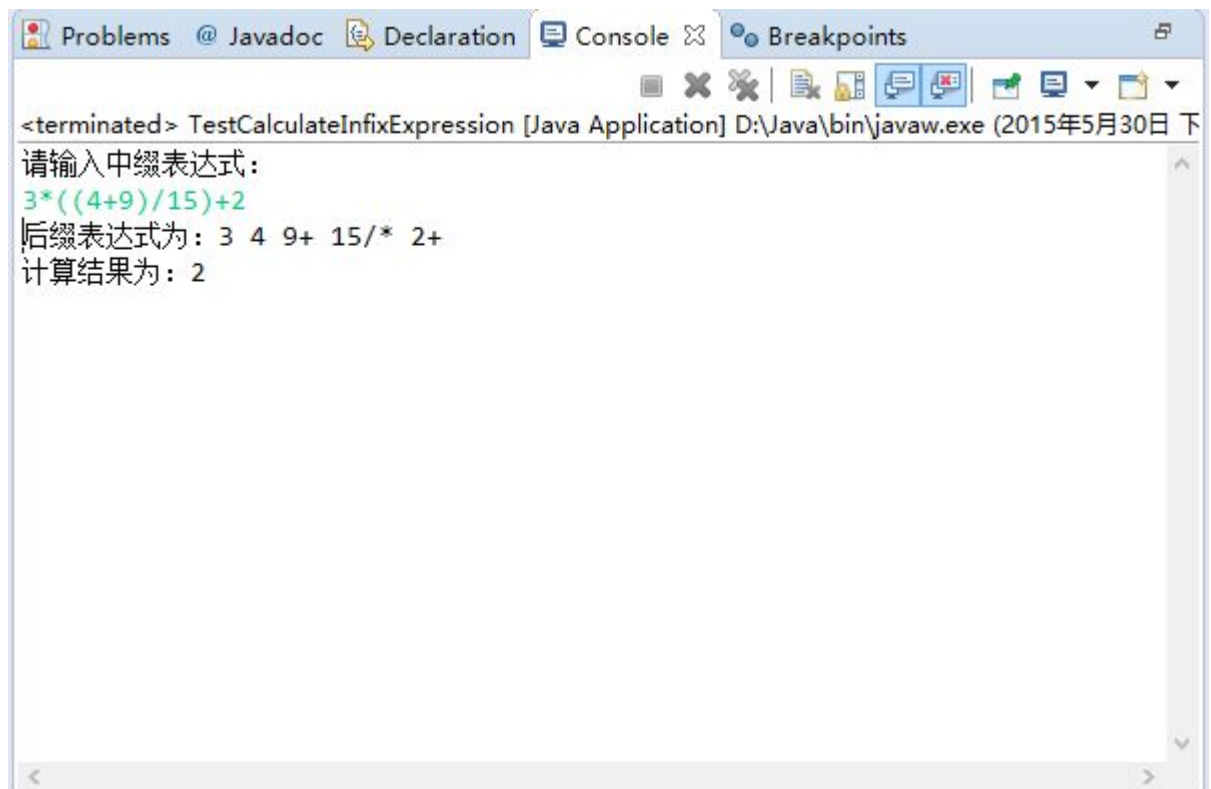
import java.util.Scanner;
public class TestCalculateInfixExpression {

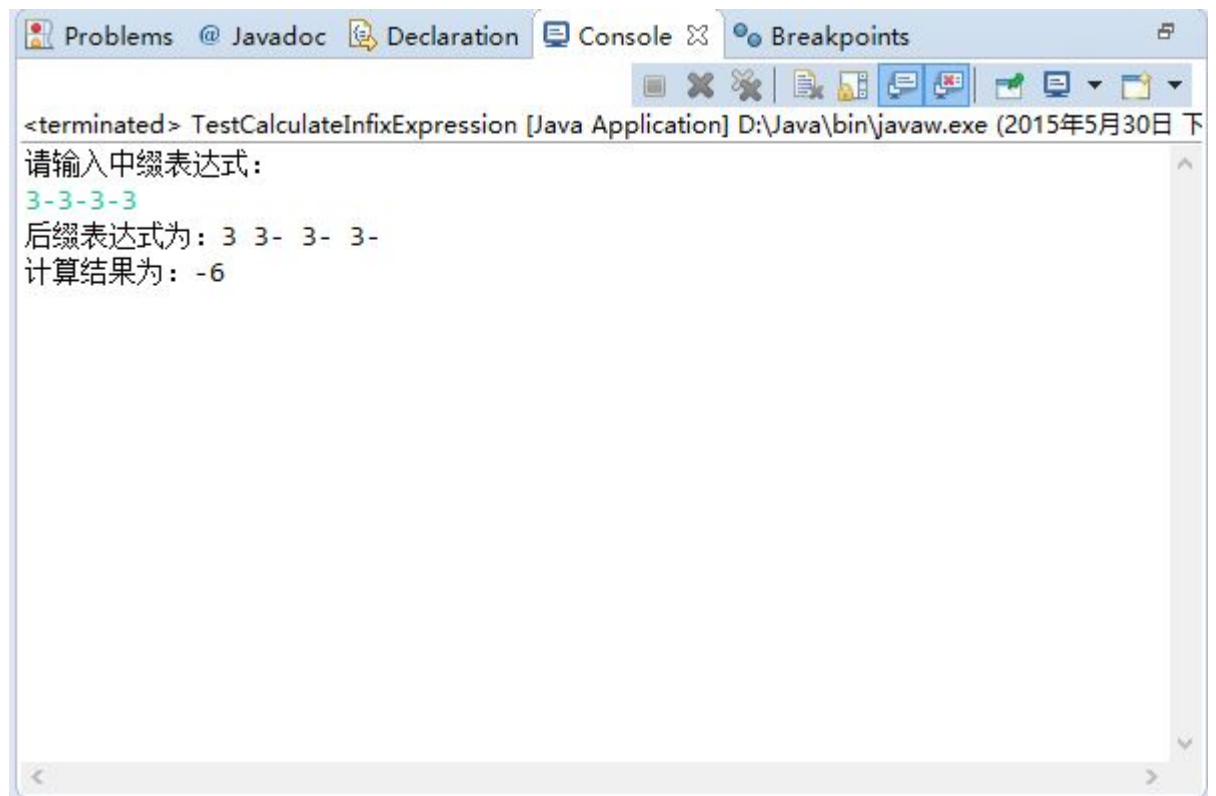
    public static void main(String[] args) {
        Scanner input=new Scanner(System.in);
    }
}

```

```
String s;  
System.out.println("请输入中缀表达式: ");  
s=input.nextLine();  
InfixToPostfix ITP=new InfixToPostfix();  
ITP.toPostfix(s);  
System.out.println("后缀表达式为: "+ITP.getpostfixString());  
CalculatePostfixExpression CPE=new CalculatePostfixExpression();  
System.out.println("计算结果为:  
"+CPE.calculate(ITP.getpostfixString()));  
}  
}
```

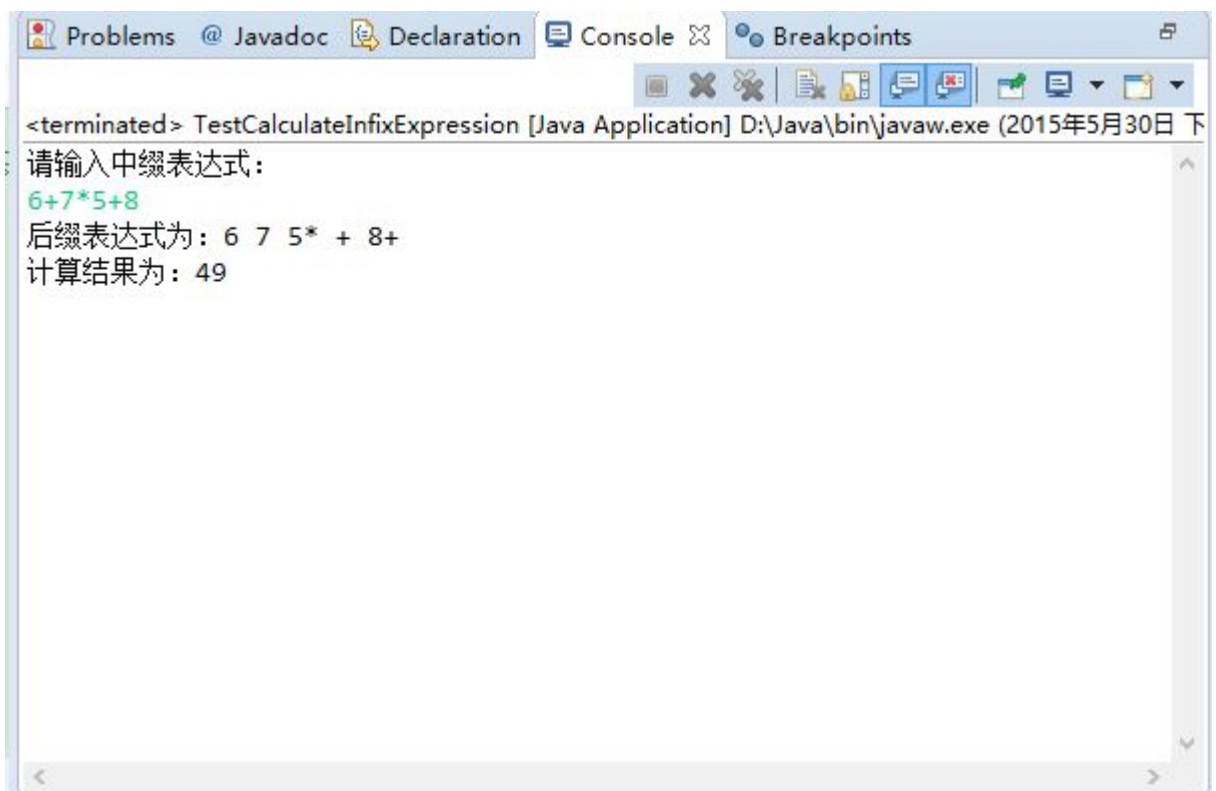
输出结果:





The screenshot shows a Java IDE console window titled "TestCalculateInfixExpression [Java Application] D:\Java\bin\javaw.exe (2015年5月30日 下)". The console output is as follows:

```
<terminated> TestCalculateInfixExpression [Java Application] D:\Java\bin\javaw.exe (2015年5月30日 下)
请输入中缀表达式:
3-3-3-3
后缀表达式为: 3 3- 3- 3-
计算结果为: -6
```



The screenshot shows a Java IDE console window titled "TestCalculateInfixExpression [Java Application] D:\Java\bin\javaw.exe (2015年5月30日 下)". The console output is as follows:

```
<terminated> TestCalculateInfixExpression [Java Application] D:\Java\bin\javaw.exe (2015年5月30日 下)
请输入中缀表达式:
6+7*5+8
后缀表达式为: 6 7 5* + 8+
计算结果为: 49
```

8.实验教材第 2 章问题 4: 以顺序结构实现双向栈共享同一个空间的入栈和出栈操作

只需在单向栈的基础上做修改, 对每个基本操作分为对哪个栈的操作即可

**BidirectionalStack.java**

```

public class BidirectionalStack<E> {
    protected int capacity; //实际数组容量
    public static final int CAPACITY=1000;
    protected E S[];
    protected int top0=-1;
    protected int top1=CAPACITY;
    public BidirectionalStack(){
        this(CAPACITY);
    }
    public BidirectionalStack(int capacity2) {
        capacity=capacity2;
        //泛型类中的泛型数据只能调用 Object 类中的方法
        S=(E[]) new Object[capacity];
        top1=capacity2;
    }

    public int getCapacity(){
        return capacity;
    }
    public int size(int i){
        switch (i){
            case 0:return top0+1;
            case 1:return (capacity-top1);
            default:return 0;
        }
    }
    public boolean isEmpty(int i){
        switch (i){
            case 0:return(top0<0);
            case 1:return (top1>=capacity);
            default:return true;
        }
    }
    public void push(E ele,int i) throws FullStackException{
        if (top0==top1)
            throw new FullStackException("Stack is full!");
        if (i==0){
            top0++;
            S[top0]=ele;
        }
        else if (i==1){
            top1--;
            S[top1]=ele;
        }
    }
    public E top(int i) throws EmptyStackException{
        if (isEmpty(i))
            throw new EmptyStackException("Stack is empty!");
        switch (i){
            case 0:return(S[top0]);
            case 1:return (S[top1]);
            default:return null;
        }
    }
}

```



```

    }

}

public E pop(int i) throws EmptyStackException{
    if (isEmpty(i))
        throw new EmptyStackException("Stack is empty!");
    E topEle = null;
    switch (i){
        case 0:
            topEle=S[top0];
            S[top0]=null;
            top0--;
            break;

        case 1:
            topEle=S[top1];
            S[top1]=null;
            top1++;
            break;

    }

    return topEle;
}

public String toString(int i){
    String s;
    s="[";
    if (i==0){
        if (size(i)>0) s=s+S[0];
        if (size(i)>1)
            for (int j=1;j<size(i);j++){
                s=s+","+S[j];
            }
    }

    else if (i==1){
        if (size(i)>0) s=s+S[capacity-1];
        if (size(i)>1)
            for (int j=1;j<size(i);j++){
                s=s+","+S[capacity-1-j];
            }
    }
    s=s+"]";
    return s;
}

public E getStack(int i){
    return S[i];
}

}

```

测试双向栈，按实验问题的要求编写主程序

**TestBidirectionalStack.java**

```
public class TestBidirectionalStack {

    public static void main(String[] args) {
        System.out.println(" (1) 初始化双向栈 B");
        BidirectionalStack<Character> B=new
BidirectionalStack<Character>(6); //初始化一个容量为 6 的双向栈

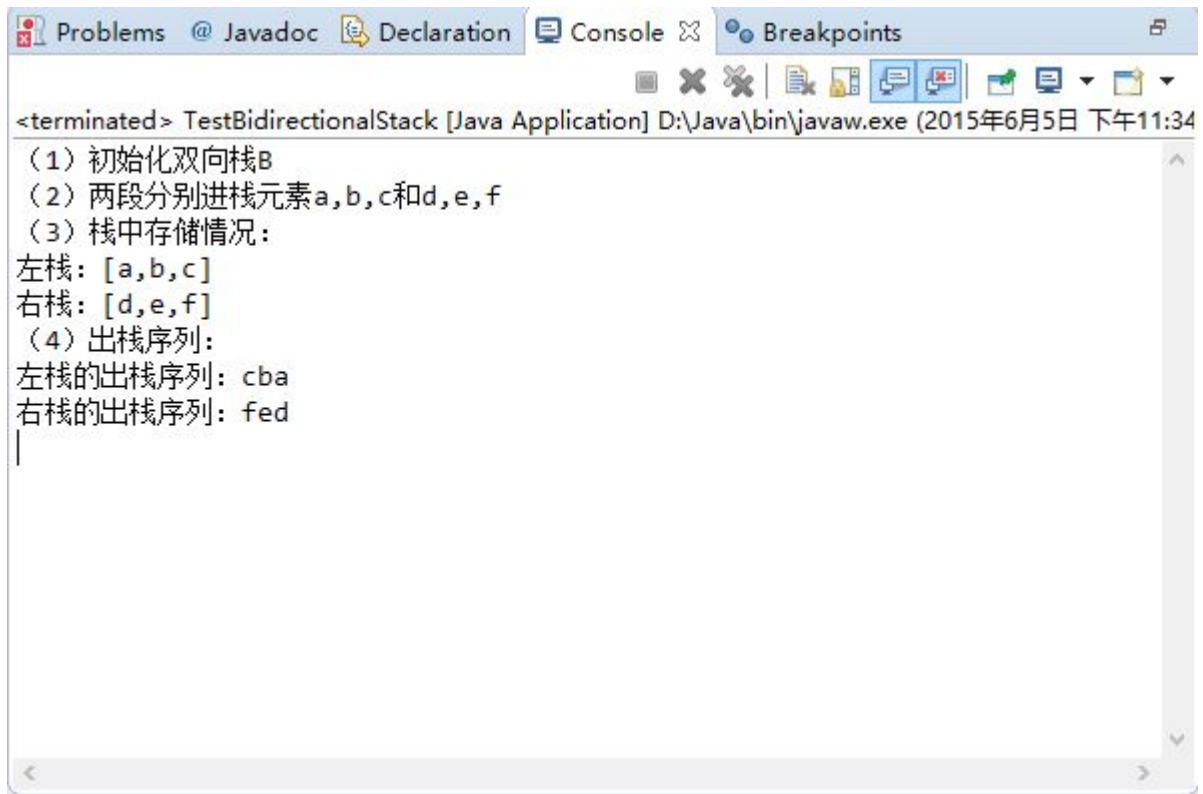
        System.out.println(" (2) 两段分别进栈元素 a,b,c 和 d,e,f");
        B.push('a',0);
        B.push('b',0);
        B.push('c',0);
        B.push('d',1);
        B.push('e',1);
        B.push('f',1);

        System.out.println(" (3) 栈中存储情况: ");
        System.out.println("左栈: "+B.toString(0));
        System.out.println("右栈: "+B.toString(1));

        System.out.println(" (4) 出栈序列: ");
        char e;
        System.out.print("左栈的出栈序列: ");
        for (int i=0;i<B.getCapacity()/2;i++){
            e=B.pop(0);
            System.out.print(e);
        }
        System.out.println();
        System.out.print("右栈的出栈序列: ");
        for (int i=0;i<B.getCapacity()/2;i++){
            e=B.pop(1);
            System.out.print(e);
        }
        System.out.println();

    }

}
```



## 9.实验教材第 2 章问题 5：实现顺序循环队列的各种基本运算

ADT 设计如下：

### IQueue.java

```
public interface IQueue<E> {
    public int size();

    public boolean isEmpty();

    public E front() throws EmptyQueueException;

    public void enqueue(E element) throws FullQueueException;;

    public E dequeue() throws EmptyQueueException;
}
```

具体实现：

### ArrayQueue.java

```
public class ArrayQueue<E> implements IQueue<E>{
    protected int f;//第一个元素的下边
    protected int r;//最后一个元素的下一个坐标
```

```

protected int N;//容量
protected E Q[];
public static final int CAPACITY=1000;//默认容量

public ArrayQueue(){
    this(CAPACITY);
}
public ArrayQueue(int capacity2) {
    N=capacity2;
    f=r=0;
    //泛型类中的泛型数据只能调用 Object 类中的方法
    Q=(E[]) new Object[N];
}
public int size() {
    //顺序循环队列，不能用 r-f 直接代表元素个数
    return (r+N-f)% N;
}

public boolean isEmpty() {
    return r==f;
}

public E front() throws EmptyQueueException {
    if (isEmpty()) throw new EmptyQueueException("Queue is empty!");
    return Q[f];
}

public void enqueue(E element) throws FullQueueException {
    if (size()==N-1) throw new FullQueueException("Queue is full!");//队
    //列满和空时都有 f=r,所以此处用 size 判断
    Q[r]=element;
    r=(r+1)%N;
}

public E dequeue() throws EmptyQueueException {
    if (isEmpty()) throw new EmptyQueueException("Queue is empty!");
    E temp;
    temp=Q[f];
    f=(f+1)%N;
    return temp;
}
}

```

测试循环队列，测试内容同实验教材：

## TestArrayQueue.java

```
public class TestArrayQueue {
    public static void main(String[] args) {
        System.out.println("(1)初始化队列");
        ArrayQueue<Character> Q=new ArrayQueue<Character>(6);//设置容量为 6;

        System.out.println("(2)队列是否为空: "+Q.isEmpty());

        System.out.println("(3)依次进队列元素 a,b,c");
        Q.enqueue('a');
        Q.enqueue('b');
        Q.enqueue('c');

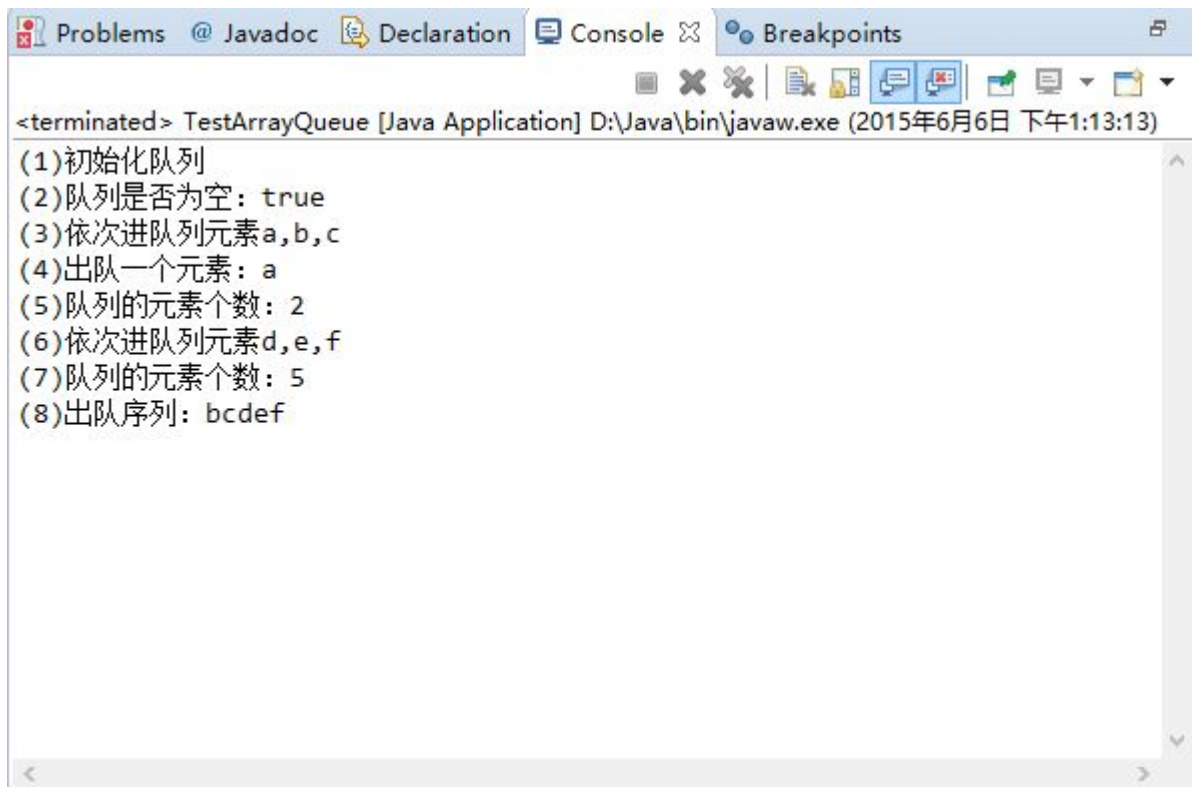
        System.out.println("(4)出队一个元素: "+Q.dequeue());

        System.out.println("(5)队列的元素个数: "+Q.size());

        System.out.println("(6)依次进队列元素 d,e,f");
        Q.enqueue('d');
        Q.enqueue('e');
        Q.enqueue('f');

        System.out.println("(7)队列的元素个数: "+Q.size());

        System.out.print("(8)出队序列: ");
        while(!Q.isEmpty()) System.out.print(Q.dequeue());
        System.out.println();
    }
}
```



The screenshot shows a Java IDE window with the 'Console' tab selected. The title bar indicates the application is 'TestArrayQueue [Java Application]' running at 'D:\Java\bin\javaw.exe' on '2015年6月6日 下午1:13:13'. The console output matches the program's logic:

```
<terminated> TestArrayQueue [Java Application] D:\Java\bin\javaw.exe (2015年6月6日 下午1:13:13)
(1)初始化队列
(2)队列是否为空: true
(3)依次进队列元素a,b,c
(4)出队一个元素: a
(5)队列的元素个数: 2
(6)依次进队列元素d,e,f
(7)队列的元素个数: 5
(8)出队序列: bcdef
```

## 五、实验体会

1. 堆栈是很灵活的数据结构，使用堆栈可以节省内存的开销。比如，递归是一种很消耗内存的算法，我们可以借助堆栈消除大部分递归，达到和递归算法同样的目的。
2. 队列中涉及入队和出队操作，若每次操作后都采用头尾指针向后移，将使得空间利用率大大降低。当我们知道了队列中同时存在元素的最大个数时，使用循环队列，指针指到数组末尾了，又重新指向数组头部，能大大节省空间；
3. 循环队列中，但队列满时就不能继续插入元素了，当队列满、空时均有  $f=r$ ；队列满时， $size=0$ ；故当需要插入元素时，判断是否有  $size()=N-1$ ，如果等式成立表示若再插入一元素就会造成队满，故应在此状态下停止插入元素；

4. 对于 TestArrayStackBaseOperation.java 中的  
System.out.print("(7)从栈顶到栈顶输出元素：");

```
int i=A.top;
while (i!=-1){
    System.out.print(A.getStack(i));
    i--;
}
```

如果划线部分改为 System.out.print(A.S[i]);

则会抛出异常 Exception in thread "main" java.lang.ClassCastException:

[Ljava.lang.Object; cannot be cast to [Ljava.lang.Character;

5. 用计算机求解四则运算，可以使用栈。因为栈的“先进后出”的特性正好满足了能通过后缀表达式去计算出四则运算式子的结果。而后缀表达式的转化也能使用栈对中缀表达式进行操作从而转化。明显地，由中缀表达式→后缀表达式， 后缀表达式→式子结果。 都需要使用到栈。

## 六、参考文献

1. 主讲课英文教材 **Goodrich, Tamassia: Data Structures and Algorithms in Java, 5th Edition International Student Version chapter 5**
2. 实验教材：汪萍，陆正福等编著 数据结构与算法的问题与实验 第2章
3. (如有其它参考文献，请列出)