Martin Götze, Bernhard Streit

7. Übung zu Informatik B

Tutorium: Marco Block, Fr 12–14 Uhr

1. **Zweite Potenzen – Shiftoperatoren**

   Schreiben Sie ein Java-Programm zur Bestimmung der Ausgabe 2^i für eine ganze positive Eingabe i. Ihr Algorithmus sollte *nicht* i mal die Zahl 2 mit sich multiplizieren, es geht viel schneller.

   Die Idee ist: verwende den Java Operator `<<`, er schiebt eine Zahl bitweise nach links, jede Verschiebung um ein Bit bedeutet aber eine Multiplikation mit 2, man kann also einfach `1L << i` zum Berechnen der Zweierpotenzen verwenden, also in Java:

```java
/** Bernhard Streit, Martin Götze
 *      7. Übung, Aufgabe 1
 *
 *  Dieses Programm berechnet für eine ganze positive Zahl i die
 *  2er-Potenz
 */
class Zweierpotenz {

    /** Zunächst die Main Methode, sie liest die Eingaben ein
     *  und gibt die entsprechenden Zweierpotenzen aus.
     */
    public static void main ( String[] argumente ) {
        if (argumente.length == 0) {
            System.out.println("Zu wenige Argumente !");
            System.out.println("Aufruf: java Zweierpotenz [Liste positiver
                 ganzer Zahlen]");
        }
        else {
            for (int i = 0; i < argumente.length; i++) {
                // jedes Argument einzeln
                try {
                    int zahl = Integer.parseInt(argumente[i]);
                    long potenz = zweitePotenz(zahl);
                    System.out.println("Es ist 2^" + zahl + " = " +
                         potenz);
                }
                catch (NumberFormatException e) { // keine Zahl:
                    System.out.println(argumente[i] + " ist keine Zahl.");
                }
                catch (IllegalArgumentException e) {
                    System.out.println(e.getMessage());
                }
            }
        }
        return;
    }

    /** Nun die Methode, die die 2er-Potenzen berechnet, einfach durch
         Shift
     */
    private static long zweitePotenz( int exponent ) throws
         IllegalArgumentException {
        if (exponent < 0)
            throw new IllegalArgumentException(exponent + " ist
                 negativ, nur nicht-negative Zahlen");
```

```
        return 1L << exponent;
    }
}
```

ein Testlauf ergab:

```
> java Zweierpotenz
Zu wenige Argumente !
Aufruf: java Zweierpotenz [Liste positiver ganzer Zahlen]
> java Zweierpotenz 0 4 10 -1 r
Es ist 2^0 = 1
Es ist 2^4 = 16
Es ist 2^10 = 1024
-1 ist negativ, nur nicht-negative Zahlen
r ist keine Zahl.
```

Scheint alles zu klappen.

2. **Klammern**

Ein Programmtext T kann neben anderen Zeichen Klammern der Form {, }, [,], (und) enthalten.

Wenn S ein leerer String oder ein String ohne Klammern ist, so heißt er korrekt geklammert. Wenn P, P', P'' korrekt geklammert sind, so sind auch

$$P'(P)P'', P'[P]P'', P'\{P\}P''$$

korrekt geklammert. Schreiben und implementieren Sie einen Algorithmus, der unter Benutzung eines Stacks in Zeit proportional zur Länge von L testet, ob T korrekt geklammert ist.

Also, zunächst der ADT für einen Stack mit den benötigten Exceptions:

```java
/** Martin Götze, Bernhard Streit
 *      7. Übung, Aufgabe 2
 *
 *   Ein Stack-Interface mit den dezugehörigen Fehlern, sonst nichts
 */

public interface Stack {
    // Die Zugriffs-Methoden
    public int size() ; // Anzahl der Elemente
    public boolean isEmpty(); // leer?
    public Object top() throws StackEmptyException; // oberstes Element
    // Die Methoden, die etwas ändern:
    public void push(Object obj) throws StackFullException; // obj ablegen
    public Object pop() throws StackEmptyException; // oberstes Element entfernen
}

// Ein allg. Stack-Fehler:
class StackException extends RuntimeException {
    public StackException() { super(); }
    public StackException(String msg) { super(msg); }
}

// und die beiden benötigten Exceptions
class StackFullException extends StackException {
    public StackFullException() { super(); }
    public StackFullException(String msg) { super(msg); }
}

class StackEmptyException extends StackException {
    public StackEmptyException() { super(); }
    public StackEmptyException(String msg) { super(msg); }
}
```

nun eine Implementierung (hier mit verketteter Liste):

```java
/** Martin Götze, Bernhard Streit
 *      7. Übung, Aufgabe 2
 *
```

```
 *  Eine Realisierung des Stack-Interfaces mittels einer verketteten Liste,
 *  für den Klammerparser
 */
public class LinkedListStack implements Stack {
    // Es wird der Kopf der Liste benötigt, dort wird eingefügt und entfernt:
    private Node head;
    /* um size() effizient zu implementieren, ist es empfehlenswert, die
       Anzahl der Einträge mitzuzählen */
    private int size;

    /** Der Konstruktor, es wird head mit null initialisiert und size
     * wird 0 gesetzt. */
    public LinkedListStack() {
        super();
        head = null;
        size = 0;
    }

    /* Nun die Implementierung der Methoden des Interfaces Stack: */
    // Die Zugriffs-Methoden
    public int size() {
        // Zum Glück mitgezählt:
        return size;
    }

    public boolean isEmpty() {
        // Wenn head auf nix zeigt, ist nix da
        return (head == null);
    }

    public Object top() throws StackEmptyException {
        if (isEmpty()) throw new StackEmptyException("Der Stack ist leer.");
        return head.value();
    }

    // Die Methoden, die etwas ändern:
    public void push(Object obj) throws StackFullException {
        // Also, einen neuen Knoten erzeugen, der auf head zeigt:
        Node newHead = new Node(obj, head);
        // Größe anpassen:
        size++;
        // und head "umbiegen"
        head = newHead;
    }

    public Object pop() throws StackEmptyException {
        // Was da?
        if (isEmpty()) throw new StackEmptyException("Der Stack ist leer.");
        // Ansonsten:
        Object obj = head.value();
```

```
        head = head.next();
        size--;
        return obj;
    }

    /** Nun noch die Klasse für die Knoten */
    private class Node {
        // Referenzen für Inhalt und Nachfolger:
        private Object value;
        private Node next;
        // Zugirffsmethoden für die Attribute:
        public Object value() { return value; }
        public Node next() { return next; }
        // und der Konstruktor:
        public Node(Object value, Node next) {
            super();
            this.value = value;
            this.next = next;
        }
    }

}
```

Nun aber zum eigentlichen Problem:

Im Prinzip ist die Idee recht einfach: Der String wird Zeichen für Zeichen durchgegangen, die öffnenenden Klammern werden auf dem Stack abgelegt, wird eine schließende gefunden, so wird das oberste Stackelement geholt und mit der schließenden vergleichen, wenns passt ok, ansonsten wird eine Fehlermeldung ausgegeben, genauso wie wenn der Stack leer ist,oder am Schluß offene Klammern übrigbleiben, also:

```
Verwende den Stack S
Für i ← 1 bis Länge des Strings
    Wenn das i−te Zeichen eine öffnende Klammer ist
        S.push(i−tes Zeichen)
    Wenn das i−te Zeichen eine schließende Klammer ist
        Wenn S.isEmpty()
            Fehlermeldung // zu ohne auf
        Wenn S.pop() und i−tes Zeichen kein Klammer−Paar sind
            Fehlermeldung
    // für dieses Zeichen: alles ok
Wenn nicht S.isEmpty() // zu wenig zu−Klammern:
    Fehlermeldung
```

Lineare Zeit in der Länge des Strings wird erreicht, da die StackMethoden konstante Zeit erfordern und somit für jedes einzelne Zeichen nur konastante Zeit benötigt wird.

Einen solchen Klammerungs-Parser realisiert zum Beispiel die Klasse

```
/** Martin Götze, Bernhard Streit
 *    7. Übung, Aufgabe 2
 *
 *  Dieses Klasse überprüft in der Methode parse für Strings, ob er
 *  richtig geklammert ist, d.h., ob die im Konstruktor übergebenen
    Klammerpaare
```

```java
 * paarweise auftreten.
 */

public class KlammerParser {

    /** die Attribute: Ein String mit denb öffnenden und einer mit den schließenden
     * Klammern, und eine Liste von Objekten, die die einzelnen Klammern repräsentieren,
     * i.e. wird eine öffnende Klammer (die in aufKlammern an Stelle i steht) gefunden,
     * so wird klammer[i] auf dem Stack abgelegt (wichtig ist die Referenz), wird dann
     * später eine zuKlammern.charAt(i) gefunden, so kann mit
     *   klammer[i] == S.pop()
     * auf Gleichheit überprüft werden.
     */
    private String aufKlammern, zuKlammern;
    private Klammer[] klammer;

    /** Die Konstruktoren: Der default-Konstruktor initialisiert die Instanz mit
     * den Standard-Klammer-paaren (), [], {}, der anderere aktzeptiert eine
     * beliebige Liste von Strings der Länge 2, deren Zeichen verschieden sein
     * müssen -> sonst Exception, das erste Zeichen ist das Klammer-Auf-Zeichen,
     * das zweite das Klammer-Zu-Zeichen
     */
    public KlammerParser() {
        this(new String[]{});
    }

    public KlammerParser(String[] zeichen) throws IllegalArgumentException {
        if (zeichen.length == 0)
            zeichen = new String[]{"()", "[]", "{}"}; // Standard
            // Initialisierung :
            klammer = new Klammer[zeichen.length];
            aufKlammern = zuKlammern = "";
        // Jetzt jedes Klammerpaar verarbeiten
            for (int i = 0; i < zeichen.length; i++) {
            if (zeichen[i].length() != 2)
                throw new IllegalArgumentException(zeichen[i] + " hat nicht Länge 2.");
            // ansonsten einfügen, falls nicht schon vorhanden:
            for (int j = 0; j < 2; j++)
                if (aufKlammern.indexOf(zeichen[i].charAt(j)) >= 0 || zuKlammern.indexOf(zeichen[i].charAt(j)) >= 0)
                    throw new IllegalArgumentException(zeichen[i].charAt(j) + " zweimal als Klammer übergeben => verboten.");
            aufKlammern += zeichen[i].charAt(0);
```

```
                zuKlammern += zeichen[i].charAt(1);
                klammer[i] = new Klammer(); // repräsentiert das i-te Klammersymbol
            }
            // Das war's.
        }

        /** Die parse-Methode überprüft, ob das übergebene Objekt richtig geklammert ist, falls nicht,
         *  liefert sie eine IllegalBracingException
         */
        public boolean parse(String text) throws IllegalBracingException {
            // zum Speichern:
            Stack S = new LinkedListStack();
            // jetzt einfach Zeichen für Zeichen durchgehen:
            for (int i = 0; i < text.length(); i++) {
                // das bearbeitete Zeichen:
                char aktuell = text.charAt(i);
                // 1. Fall: öffnende Klammer:
                if (aufKlammern.indexOf(aktuell) >= 0)
                    S.push(klammer[aufKlammern.indexOf(aktuell)]);
                // 2. Fall: schließende Klammer:
                if (zuKlammern.indexOf(aktuell) >= 0) {
                    // zu viele schließende Klammern ??
                    if (S.isEmpty())
                        throw new IllegalBracingException(aktuell + " an Stelle " + i + " ohne öffnende Klammer.");
                    // falsche schließende Klammer ??
                    if (S.pop() != klammer[zuKlammern.indexOf(aktuell)])
                        throw new IllegalBracingException("Falsche schließende " +
                             aktuell + " an Stelle " + i + ".");
                }
            }
            // Falls der Stack nicht leer ist, waren es mehr öffnende als schließende Klammern:
            if (!S.isEmpty()) throw new IllegalBracingException("Zu wenig schließende Klammern.");
            return true;
        }

        // Nur für die Klammer-Liste, die Klasse muss nichts können (außer: da sein)
        private class Klammer {};
}

/** Die Klasse IllegalBracingException für Klammerungsfehler */
class IllegalBracingException extends Exception {
        IllegalBracingException() { super(); }
        IllegalBracingException(String msg) { super(msg); }
}
```

zum Testen eine Test-Klasse:

```
/** Martin Götze, Bernhard Streit
```

```java
 *      7. Übung, Aufgabe 2
 *
 * Die Klasse KlammerParserTest testet den KlammerParser, indem jeder in der
 * Eingabe Zeile übergebene String auf korrekte Klammerung in der Standard−
 * Klammerung der Aufgabe (), [], {} untersucht wird.
 */
public class KlammerParserTest {
    /** Diese Klasse hat als Test−Klasse nur die main−Methode, es werden die
     * Eingaben nacheinander übergeben und geparst.
     */
    public static void main(String[] argumente) {
        if (argumente.length == 0) {
            System.out.println("Aufruf: java KlammerParserTest <Liste von Strings>");
        }
        else {
            KlammerParser parser = new KlammerParser();
            for (int i = 0; i < argumente.length; i++) {
                try {
                    parser.parse(argumente[i]);
                    System.out.println("Der String " + argumente[i] + " ist korrekt geklammert.");
                }
                catch (IllegalBracingException e) {
                    System.out.println("Fehler in " + argumente[i] + ": " + e.getMessage());
                }
            }
        }
    }
}
```

die Ausgabe war:

```
> java KlammerParserTest "()" "" "((((" "(4 + 5[8+7]){}" "}" "(]"
Der String () ist korrekt geklammert.
Der String  ist korrekt geklammert.
Fehler in ((((: Zu wenig schließende Klammern.
Der String (4 + 5[8+7]){} ist korrekt geklammert.
Fehler in }: } an Stelle 0 ohne öffnende Klammer.
Fehler in (]: Falsche schließende ] an Stelle 1.
> java KlammerParserTest
Aufruf: java KlammerParserTest <Liste von Strings>
```

Alles ok.

3. **bunte Stapel**

   Entwerfen Sie einen ADT für einen 2-Farben-Doppelstack, also einen roten und einen blauen Stack mit jeweils farbigen Operationen (rotes Push, blaues Push, ...). Geben Sie eine effiziente Implementierung mittels eines Arrays an. Dessen Größe soll so groß sein wie die der beiden Stacks zusammen.

   Also zunächst der ADT: Abstrakt muss er jede Stack-Methode doppelt enthalten, einmal in einer roten und einmal in einer blauen Version, man hat also das Java-Interface (es werden auch gleich die jeweiligen Fehlerklassen definiert):

```java
/** Martin Götze, Bernhard Streit
 *     7. Übung, Aufgabe 3
 *
 *  Diese Datei stellt ein Interface unseres ADTs eines Zweifarbigen
 *  Stacks zur Verfügung
 */

public interface BunterStack {
    /* Zunächst die Zugriffsmethoden : */
    /* die Methoden ..IsEmpty des jeweiligen Stacks */
    public boolean redIsEmpty();
    public boolean blueIsEmpty();
    /* nun die Tops */
    public Object redTop() throws StackEmptyException;
    public Object blueTop() throws StackEmptyException;
    /* und die Größen */
    public int redSize();
    public int blueSize();
    /* und nun die Methoden, die den Stack verändern */
    /* erst ..push() */
    public void redPush(Object o) throws StackFullException;
    public void bluePush(Object o) throws StackFullException;
    /* und ..pop() */
    public Object redPop() throws StackEmptyException;
    public Object bluePop() throws StackEmptyException;
}

/** Die Klassen für die Stack-Fehler werden von
 *  Exception abgeleitet :
 */

// Basis für voll & leer ist der allg . StackFehler:
class StackException extends RuntimeException {
    StackException() { super(); }
    StackException(String msg) { super(msg); }
}

class StackFullException extends StackException {
    StackFullException() { super(); }
    StackFullException(String msg) { super(msg); }
}

class StackEmptyException extends StackException {
    StackEmptyException() { super(); }
    StackEmptyException(String msg) { super(msg); }
```

```
}
```

Bei der Realisierung mittels eines Arrays wird wie folgt vorgegangen: Der rote Stack wird von vorn, der blaue von hinten aufgebaut und man merkt sich in je einer int-Variablen die Positionen der jeweiligen top-Elemente (wie schon beim Stack), dadurch können alle BunterStack-Operationen in konstanter Zeit (nämlich durch einfache Array-Zugriffe) realisiert werden (und das ist effizient), man hat z.B. in Java:

```
/** Martin Götze, Bernhard Streit
 *     7. Übung, Aufgabe 3
 *
 * Diese Datei stellt einen zweifarbigen Stack zur Verfügung, der
 * mittels eines festen Arrays implementiert wird.
 */
public class BunterArrayStack implements BunterStack {

    /* Die Länge des verwendeten Arrays verwalten */
    public final static int DEFAULTKAP = 1024;
    // Die Default-Länge
    public final int kapazitaet;

    // Das Feld, das die Stackelemente aufnehmen soll

    private Object[] feld;

    // zwei Int-Attribute, die sich merken, wo das Top-Element,
    // der beiden Stacks steht

    private int redTop, blueTop;

    /** Nun die Konstruktoren, einer der die Standard-Kapizität verwendet
     * und ein allgemeiner, in dem man die Kapizität übergeben kann.
     */

    public BunterArrayStack() {
        this(DEFAULTKAP);
    }

    public BunterArrayStack(int kap) {
        if (kap <= 0) kap = DEFAULTKAP;
        kapazitaet = kap;
        feld = new Object[kapazitaet];
        // Beide Stacks sind leer:
        redTop = -1;
        blueTop = kapazitaet;
        }

    /** Nun werden die Methoden des Interfaces BunterStack
     * mit Leben versehen
     */

    /* Zunächst die Zugriffsmethoden : */
```

```
    /* die Methoden ..IsEmpty des jeweiligen Stacks */
    public boolean redIsEmpty() {
        return redTop < 0;
    }

    public boolean blueIsEmpty()
    {
        return blueTop >= kapazitaet;
    }

    /* nun die Tops */
    public Object redTop() throws StackEmptyException {
        if (redIsEmpty()) throw new StackEmptyException("Der rote Stack ist leer");
        return feld[redTop];
    }

    public Object blueTop() throws StackEmptyException {
        if (blueIsEmpty()) throw new StackEmptyException("Der blaue Stack ist leer");
        return feld[blueTop];
    }

    /* und die Größen, sie ergeben sich leicht so: */
    public int redSize() {
        return redTop + 1;
    /* denn redTop ist die position des ersten roten Elements
       und das erste Feldelement hat die Nr 0. */
    }

    public int blueSize() {
        return kapazitaet - blueTop;
    /* denn blueTop ist position des ersten blauen Elements, das
       Feld hat kapazitaet Elemente. */
    }

    /* und nun die Methoden, die den Stack verändern */
    /* erst ..push() */
    public void redPush(Object o) throws StackFullException {
        if (redTop == blueTop - 1) // voll
            throw new StackFullException("Der Doppelstack ist voll.");
        // ansonsten einfügen:
        feld[++redTop] = o;
    }

    public void bluePush(Object o) throws StackFullException { // dito
        if (redTop == blueTop - 1) // voll
            throw new StackFullException("Der Doppelstack ist voll.");
        feld[--blueTop] = o;
    }

    /* und ..pop(), wie top() nur verändern */
    public Object redPop() throws StackEmptyException {
        if (redIsEmpty()) throw new StackEmptyException("Der rote
```

```
            Stack ist leer");
        return feld[redTop--];
    }

    public Object bluePop() throws StackEmptyException {
        if (blueIsEmpty()) throw new StackEmptyException("Der blaue Stack ist leer");
        return feld[blueTop++];
    }
    // Das war's.
}
```

Eine Test-Klasse:

```
/** Martin Götze, Bernhard Streit
 *    7. Übung, Aufgabe 3
 *
 * Eine Testklasse für unseren BuntenArrayStack, es werden (in main, sonst
 * gibt's eh' nix) einige Elemente in den Stack einfefügt und wieder herausgeholt,
 * der aktuelle Zustand wird ausgegeben, und es werden einige Fehler erzeugt.
 */

public class BunterArrayStackTest {

    /** wie gesagt, nur für Testzwecke, also nix außer der main-Methode und einer
     * Statusausgabe für den Stack */
    private static BunterStack bunt = new BunterArrayStack(10);

    /** Statusanzeige für den Stack */
    private static void status() {
        // die obersten Elemente lesen:
        String rotOben, blauOben;
        try { rotOben = bunt.redTop().toString(); }
        catch (StackEmptyException e) {
            rotOben = "leer";
        }
        try { blauOben = bunt.blueTop().toString(); }
        catch (StackEmptyException e) {
            blauOben = "leer";
        }
        System.out.println("rot: " + bunt.redSize() + " blau: " + bunt.blueSize() +
            " oberste Elemente: (" + rotOben + ", " + blauOben + ")");
   }

    public static void main(String[] argumente) {
        // So, los gehts, einfach ein paar Zahlen einfügen und wieder auslesen:
        int [] zahlen   = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
        // alles einfügen:
```

```java
        for (int i = 0; i < zahlen.length; i++) {
            System.out.print(" ... füge " + zahlen[i] + " in den " + ((i%2==0) ? "roten" : "blauen") +
                " Stack ein ... ");
            try { // es werden absichtlich zwei Exceptions ausgeloest
                if (i % 2 == 0)
                    bunt.redPush(new Integer(zahlen[i]));
                else
                    bunt.bluePush(new Integer(zahlen[i]));
            }
            catch (StackException e) {
                System.out.print(e.getMessage()); // diese sollte zweimal erscheinen.
            }
            System.out.println();
            status();
        }
        // und wieder auslesen
        for (int i = 0; i < zahlen.length; i++) {
            System.out.print("Lese zahl aus dem " + ((i%2 == 0) ? "roten" : "blauen") +
                " Stack ... ");
            try {
                if (i % 2 == 0)
                    System.out.print(bunt.redPop());
                else
                    System.out.print(bunt.bluePop());
            }
            catch (StackException e) {
                System.out.print(e.getMessage());
            }
            System.out.println();
            status();
        }
    } // Das war's
}
```

und die Ausgabe:

```
> java BunterArrayStackTest
 ... füge 1 in den roten Stack ein ...
rot : 1 blau : 0 oberste Elemente : (1, leer )
 ... füge 2 in den blauen Stack ein ...
rot : 1 blau : 1 oberste Elemente : (1, 2)
 ... füge 3 in den roten Stack ein ...
rot : 2 blau : 1 oberste Elemente : (3, 2)
 ... füge 4 in den blauen Stack ein ...
rot : 2 blau : 2 oberste Elemente : (3, 4)
 ... füge 5 in den roten Stack ein ...
rot : 3 blau : 2 oberste Elemente : (5, 4)
 ... füge 6 in den blauen Stack ein ...
rot : 3 blau : 3 oberste Elemente : (5, 6)
 ... füge 7 in den roten Stack ein ...
rot : 4 blau : 3 oberste Elemente : (7, 6)
 ... füge 8 in den blauen Stack ein ...
```

```
rot : 4  blau : 4  oberste  Elemente : (7, 8)
 ...  füge 9  in  den roten  Stack ein ...
rot : 5  blau : 4  oberste  Elemente : (9, 8)
 ...  füge 10  in  den blauen  Stack  ein ...
rot : 5  blau : 5  oberste  Elemente : (9, 10)
 ...  füge 11  in  den roten  Stack ein ... Der Doppelstack  ist   voll .
rot : 5  blau : 5  oberste  Elemente : (9, 10)
 ...  füge 12  in  den blauen  Stack ein ... Der Doppelstack  ist   voll .
rot : 5  blau : 5  oberste  Elemente : (9, 10)
Lese  zahl  aus dem roten  Stack ...9
rot : 4  blau : 5  oberste  Elemente : (7, 10)
Lese  zahl  aus dem blauen  Stack ...10
rot : 4  blau : 4  oberste  Elemente : (7, 8)
Lese  zahl  aus dem roten  Stack ...7
rot : 3  blau : 4  oberste  Elemente : (5, 8)
Lese  zahl  aus dem blauen  Stack ...8
rot : 3  blau : 3  oberste  Elemente : (5, 6)
Lese  zahl  aus dem roten  Stack ...5
rot : 2  blau : 3  oberste  Elemente : (3, 6)
Lese  zahl  aus dem blauen  Stack ...6
rot : 2  blau : 2  oberste  Elemente : (3, 4)
Lese  zahl  aus dem roten  Stack ...3
rot : 1  blau : 2  oberste  Elemente : (1, 4)
Lese  zahl  aus dem blauen  Stack ...4
rot : 1  blau : 1  oberste  Elemente : (1, 2)
Lese  zahl  aus dem roten  Stack ...1
rot : 0  blau : 1  oberste  Elemente: ( leer , 2)
Lese  zahl  aus dem blauen  Stack ...2
rot : 0  blau : 0  oberste  Elemente: ( leer ,  leer )
Lese  zahl  aus dem roten  Stack ... Der rote  Stack  ist   leer
rot : 0  blau : 0  oberste  Elemente: ( leer ,  leer )
Lese  zahl  aus dem blauen  Stack ... Der blaue  Stack  ist   leer
rot : 0  blau : 0  oberste  Elemente: ( leer ,  leer )
```

Alles ok.