Übungsblatt 11

Adrian Schollmeyer

Aufgabe 1

```
package me.adrian.aup11;

public class NatList
{
    private int[] data = new int[0];
    private int beginIterator = 0;
    private int endIterator = 0;

    public NatList()
    {
        this.resizeToFit(1);
    }

    private NatList(int[] initialData)
    {
        this.data = initialData;
        this.beginIterator = 0;
        this.endIterator = this.data.length;
    }

    public NatList pushFront(int prependedValue)
    {
        this.resizeToFit(this.length() + 1);
        this.beginIterator--;
        this.data[this.beginIterator] = prependedValue;
        return this;
    }

    private void resizeToFit(int minimumSize)
    {
        if (this.data.length < minimumSize) {
            // Some extra space to speed up further push operations
            int finalSize = (int) (minimumSize * 1.5);
            int[] newData = new int[finalSize];
            int oldLength = this.length();
            int j = newData.length - oldLength;
            newData = this.copyToArray(newData, j, newData.length, 0, oldLength);
            this.data = newData;
            this.beginIterator = newData.length - oldLength;
            this.endIterator = this.data.length;
        }
    }
```

```java

    private int[] copyToArray(int[] to, int toBeginIterator, int toEndIterator, int dataBeginIterator,
        int dataEndIterator)
    {
        if (dataEndIterator - dataBeginIterator != toEndIterator - toBeginIterator)
            throw new IllegalArgumentException("Iterator ranges don't match");

        for (int i = dataBeginIterator; i < dataEndIterator; i++) {
            to[toBeginIterator++] = this.at(i);
        }
        return to;
    }

    public int head()
    {
        assertNotEmpty();
        return this.at(0);
    }

    public int rhead()
    {
        assertNotEmpty();
        return this.at(this.length() - 1);
    }

    public NatList tail()
    {
        assertNotEmpty();
        return new NatList(this.at(1, this.length()));
    }

    public NatList rtail()
    {
        assertNotEmpty();
        return new NatList(this.at(0, this.length() - 1));
    }

    public int length()
    {
        return this.endIterator - this.beginIterator;
    }

    private int at(int offset)
    {
        if (this.beginIterator + offset >= this.endIterator)
            throw new IndexOutOfBoundsException("" + offset);

```

```
        return this.data[this.beginIterator + offset];
    }

    private int[] at(int beginOffset, int endOffset)
    {
        int[] res = new int[endOffset - beginOffset];
        for (int i = 0; i < endOffset - beginOffset; i++)
            res[i] = this.at(i + beginOffset);

        return res;
    }

    private void assertNotEmpty()
    {
        if (this.beginIterator >= this.endIterator)
            throw new IndexOutOfBoundsException("List too short");
    }

    @Override
    public boolean equals(Object other)
    {
        if (this == other)
            return true;

        if (other instanceof NatList) {
            NatList nl = (NatList) other;
            return arrayEquals(this.data, this.beginIterator, this.endIterator,
                nl.data, nl.beginIterator,
                    nl.endIterator);
        } else {
            return false;
        }
    }

    private static boolean arrayEquals(int[] lhs, int lhsBegin, int lhsEnd, int
        [] rhs, int rhsBegin, int rhsEnd)
    {
        if (lhsEnd - lhsBegin != rhsEnd - rhsBegin)
            return false;

        for (int i = lhsBegin; i < lhsEnd; i++) {
            if (lhs[i] != rhs[rhsBegin + i - lhsBegin])
                return false;
        }

        return true;
    }

    @Override
```

```
    public int hashCode()
    {
        int hash = 31;
        for (int i = this.beginIterator; i < this.endIterator; i++) {
            hash = 31 * hash + i;
        }
        return hash;
    }
}
```

Aufgabe 2

```
package me.adrian.aup11;

public class NatStack
{
    private NatList data = new NatList();

    public NatStack()
    {
    }

    public NatStack push(int value)
    {
        this.data.pushFront(value);
        return this;
    }

    public NatStack pop()
    {
        this.data = this.data.tail();
        return this;
    }

    public int top()
    {
        return this.data.head();
    }

    public boolean isEmpty()
    {
        return this.data.length() == 0;
    }

    @Override
    public boolean equals(Object other)
    {
        if (this == other)
            return true;

```

```
        if (other instanceof NatStack) {
            NatStack ns = (NatStack) other;
            return this.data.equals(ns.data);
        }

        return false;
    }

    @Override
    public int hashCode()
    {
        return this.data.hashCode();
    }
}
```

Aufgabe 3

(a)

```
  a(3)
→ X#a(3)
→ Y#a(3, 3)
→ Y#a(9, 3, 3)
→ 9 * 3 + 3
→ 27 + 3
→ 30
```

(b)

Das ausgegebene Ergebnis ist 42.

Aufgabe 4

(a)

Durch einen Überlauf bei der Eingabe einer zu großen Zahl kann es gemäß der IEEE-754-Norm dazu kommen, dass double den Wert „unendlich“annimmt.

(b)

Durch einen Überlauf bei der Eingabe einer zu großen Zahl kann das Vorzeichenbit umschalten, wodurch sich das Vorzeichen der Zahl ungewollt ändert.

(c)

(d)

```java
package me.adrian.aup11;

import java.util.ArrayList;
import java.util.List;

public class BigNum
{
    // Stored as LSB first
    private List<Byte> digits;
    private boolean negative = false;

    public BigNum(String num)
    {
        this(stringToDigits(num));
        if (num.charAt(0) == '-')
            negative = true;
    }

    private BigNum(List<Byte> digits)
    {
        this.digits = digits;
    }

    public static List<Byte> stringToDigits(String num)
    {
        List<Byte> digits = new ArrayList<>();
        if (num.charAt(0) == '-')
            num = num.substring(1);
        char[] chars = num.toCharArray();
        for (int i = num.length() - 1; i >= 0; i--) {
            digits.add(Byte.valueOf("" + chars[i]));
        }
        return digits;
    }

    @Override
    public boolean equals(Object other)
    {
        if (this == other)
            return true;

        if (other instanceof BigNum) {
            return this.toString().equals(other.toString());
        }

        return false;
```

```
    }

    @Override
    public int hashCode()
    {
        return this.toString().hashCode();
    }

    @Override
    public String toString()
    {
        StringBuffer buf = new StringBuffer();
        buf.append(digitsToString(this.digits));
        String numString = buf.toString();
        while (numString.startsWith("0"))
            numString = numString.substring(1);

        String sign = this.negative ? "-" : "";
        return sign + numString;
    }

    public static String digitsToString(List<Byte> digits)
    {
        StringBuffer buf = new StringBuffer();
        for (Byte b : digits) {
            buf.insert(0, b);
        }
        return buf.toString();
    }

    public BigNum mult(BigNum other)
    {
        BigNum res = new BigNum("0");
        BigNum thisAbs = this.abs();
        BigNum otherAbs = other.abs();
        for (BigNum i = new BigNum("0"); !i.equals(otherAbs); i = i.add(new
            BigNum("1"))) {
            res = res.add(thisAbs);
        }
        res.negative = this.negative ^ other.negative ? true : false;
        return res;
    }

    public BigNum abs()
    {
        if (!this.negative)
            return this;

        // Just trim the - sign
```

```
        BigNum abs = new BigNum(this.toString().substring(1));
        return abs;
    }

    public BigNum add(BigNum other)
    {
        ArrayList<Byte> newDigits = new ArrayList<>();
        // This way we can use List#set() everytime we access the list
        int maxListSize = Math.max(this.digits.size(), other.digits.size()) + 1;
        newDigits.ensureCapacity(maxListSize);
        for (int i = 0; i < maxListSize; i++)
            newDigits.add((byte) 0);

        for (int pos = 0; this.hasDigit(pos) || other.hasDigit(pos); pos++) {
            byte cur = newDigits.get(pos);
            byte curCarry = newDigits.get(pos + 1);
            byte[] res = addDecimalBits(cur, this.getDigitAt(pos));
            cur = res[0];
            curCarry = res[1];
            res = addDecimalBits(cur, other.getDigitAt(pos));
            cur = res[0];
            curCarry += res[1];
            newDigits.set(pos, cur);
            newDigits.set(pos + 1, curCarry);
        }

        newDigits.trimToSize();
        return new BigNum(newDigits);
    }

    private boolean hasDigit(int pos)
    {
        return this.digits.size() > pos;
    }

    public static byte[] addDecimalBits(byte lhs, byte rhs)
    {
        if (lhs > 9 || rhs > 9 || lhs < -9 || rhs < -9) {
            throw new IllegalArgumentException("Operands must be within the
                interval [-9, 9]");
        }

        byte add = (byte) (lhs + rhs);
        byte result = (byte) (add % 10);
        byte carry = (byte) ((add - result) / 10);
        return new byte[] { result, carry };
    }

    private byte getDigitAt(int pos)
```

```
    {
        byte unsignedValue = this.hasDigit(pos) ? this.digits.get(pos) : 0;
        return this.negative ? (byte) -unsignedValue : unsignedValue;
    }
}
```