Programmieren 2
Übung Semesterwoche 2

1. Stack (LIFO: Last-In--First-Out)
Implementieren und testen Sie eine Klasse Stack, so dass beliebig viele Objekte eines vorgegebenen Datentyps (z. B. String) auf dem Stack abgelegt werden können. Nehmen Sie als Vorlage das Beispiel von Folie 18 aus PRG2_SW2_DAT. Verwenden Sie aber eine ArrayList und Generics. Verwenden Sie die Unit Test Möglichkeiten von BlueJ.

```
public class Stack //ohne Generics
{
    private int maxSize;
    private int top;
    private Object[] stack;

    public Stack(int maxSize)
    {
        this.maxSize = maxSize;
        this.top = 0;
        stack = new Object[maxSize];
    }

    public boolean push(Object o)
    {
        if(top+1<=maxSize)
        {
            stack[top++] = o;
            return true;
        }
        else
        {
            return false;
        }
    }

    public Object pop()
    {
        if(top>0)
        {
            top--;
            Object o = stack[top];
            stack[top] = null;
            return o;
        }
        else
        {
            return null;
        }
    }

    public boolean isEmpty()
    {
        return top == 0;
    }

    public boolean isFull()
    {
        return stack.length == top;
    }
}
```

```
import java.util.*;
@SuppressWarnings("unchecked")

public class StringStack //mit String Objekten
{
    private int maxSize;
    private ArrayList<String> stack;

    public StringStack(int maxSize)
    {
        this.maxSize = maxSize;
        stack = new ArrayList();
    }

    public void push(String s)
    {
        if(stack.size()+1<=maxSize)
        {
            stack.add(stack.size(), s);
        }
    }

    public String pop()
    {
        int i = stack.size();
        if(stack.size()>0)
        {
            String s = stack.get(stack.size()-1);
            stack.remove(stack.size()-1);
            return s;
        }
        else
        {
            return "";
        }
    }

    public boolean isEmpty()
    {
        return stack.size() == 0;
    }

    public boolean isFull()
    {
        return stack.size() == maxSize;
    }
}
```

```
import java.util.*;

public class GenericStack<T> //mit generischen Objekten
{
    private int maxSize;
    private ArrayList<T> stack;

    public GenericStack(int maxSize)
    {
        this.maxSize = maxSize;
        stack = new ArrayList<T>();
    }

    public void push(T s)
    {
        if(stack.size()+1<=maxSize)
        {
            stack.add(stack.size(), s);
        }
    }

    public T pop()
    {
        int i = stack.size();
        if(stack.size()>0)
        {
            T s = stack.get(stack.size()-1);
            stack.remove(stack.size()-1);
            return s;
        }
        else
        {
            return null;
        }
    }

    public boolean isEmpty()
    {
        return stack.size() == 0;
    }

    public boolean isFull()
    {
        return stack.size() == maxSize;
    }
}
```

2. Queue (FIFO: First-In--First-Out)

Bei einer Warteschlange sind das Einfügen nur am Ende und das Auslesen nur am Anfang der Warteschlange erlaubt. Falls die maximale Anzahl von Einträgen, die sich zu einem Zeitpunkt in der Warteschlange befinden, nach oben begrenzt ist, implementiert man sie häufig mittels eines ausreichend dimensionierten Arrays, in dem ein Index head auf den Anfang der Warteschlange zeigt und ein Index tail auf die Position am Ende. Nach dem Löschen eines Elements wird head und nach dem Einfügen eines Elements tail erhöht. Das Erhöhen der Indizes wird jeweils modulo N ausgeführt, so dass man sich das Feld als Ring vorstellen kann, der zyklisch beschrieben wird. Vervollständigen Sie die folgende Klasse für einen Ringbuffer.

```
public class Ringbuffer
{
    private Object[] o;
    private int N = 0;
    private int head = 0;
    private int tail = 0;
    private int maxSize;

    public Ringbuffer(int maxSize)
    {
        this.maxSize = maxSize;
        o = new Object[maxSize];
    }

    public boolean enqueue(Object x)
    {
        if(N!=o.length) //Einfügen nur möglich, wenn noch Platz frei ist
        {
            o[tail] = x; //Einfügen des Objekt x an der letzten Position
            tail = (tail+1) % o.length;
            N++;
            return true;
        }
        else
        { return false; }
    }

    public Object dequeue()
    {
        if(!isEmpty())
        {
            Object x = o[head];
            o[head] = null;
            N--;
            head = (head+1) % o.length;
            return x;
        }
        else
        { return null; }
    }

    public int size()
    {
        return N;
    }

    public boolean isEmpty()
    {
        return N==0;
    }

    public boolean isFull()
    {
        return size()==maxSize;
    }
}
```

3. Liste

(a) Implementieren Sie eine doppelt verkettete Liste, die beliebige Objekte aufnehmen kann. Die Objekte sollen auf effiziente Art vor- und rückwärts ausgelesen werden können. Verwenden Sie die Unit Test Möglichkeiten von BlueJ.

```
public class LinkedList<T>
{
    private int count;
    private ListNode<T> head;
    private ListNode<T> tail;

    public LinkedList()
    {
        clear();
    }

    public void add(T data)
    {
        ListNode<T> n = new ListNode<T>(data);
        if(count==0)
        {
            // Erstes Element = Letztes Element
            this.head = n;
            this.tail = n;
        }
        else
        {
            // Dem gegenwärtigen letzten Element wird n als nächstes Element angefügt
            this.tail.setNext(n);

            //Danach wird n als letztes Element definiert
            this.tail = n;
        }
        count++;
    }

    public void clear()
    {
        this.count = 0;
        this.head = null;
        this.tail = null;
    }

    public ListNode<T> get(int index)
    {
        ListNode<T> n = head;

        // Durchlaufen der ganzen Liste bis zur gesuchten Position
        for(int i=0; i<index; i++)
        {
            n = n.next();
        }

        return n;
    }

    public boolean remove(int index)
    {
        //Fall 1: Keine Elemente in der Liste
        if(count==0)
        {
            return false;
        }

        //Fall 2: Element ist das erste Element der Liste
        else if(index==0)
        {
            this.head = head.next();
```

```
            count--;
            return true;
        }

        //Fall 3: Element ist das letzte Element der Liste
        else if(index==count-1)
        {
            this.tail = tail.prev();
            count--;
            return true;
        }

        //Fall 4: Beliebiges Element aus der Liste
        else
        {
            ListNode<T> n = get(index);
            n.prev().setNext(n.next());
            n.next().setPrev(n.prev());
            count--;
            return true;
        }
    }

    public ListNode<T> set(int index, T data)
    {
        ListNode<T> n = get(index);
        n.setData(data);

        return n;
    }

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

```
public class ListNode<T>
{
    private T data;
    private ListNode<T> next;
    private ListNode<T> prev;

    public ListNode(T data)
    {
        setData(data);
    }

    public ListNode<T> next()
    {
        return next;
    }

    public ListNode<T> prev()
    {
        return prev;
    }

    public void setNext(ListNode<T> next)
    {
        this.next = next;
    }

    public void setPrev(ListNode<T> prev)
    {
        this.prev = prev;
    }

    public T getData()
    {
        return data;
    }

    public void setData(T data)
    {
        this.data = data;
    }
}
```

Weitere Informationen zu verketteten Listen: http://en.wikipedia.org/wiki/Linked_list

(b) [optional] Studieren Sie die Klasse LinkedList im Java API.
Sehen Sie sich danach die Implementation1 in der Java Klassenbibliothek an (AbstractList.java, ArrayList.java) und veranschaulichen Sie sich das Funktionieren der Methode hasPrevious und hasNext mit einer Zeichnung.