ADP – Aufgabe 1 (Skizze)

Team: 14, Sonja Knuth & Sebastian Peters
Aufgabenaufteilung:

1. Sonja Knuth & Sebastian Peters:
   Gemeinsames Ausarbeiten der Skizze
2. Sebastian Peters:
   Digitalisieren & Nachbearbeitung

Quellenangaben: N/A
Bearbeitungszeitraum: 20. Oktober, ca. 5h + Nachbearbeitung 26. Oktober, ca. 1h
Aktueller Stand: Skizze ist fertig und nachbearbeitet
Änderungen in der Skizze: N/A
Skizze: s.u.

Spezifikationen des Programms:

Aufbau der Dateien:

Das Programm besteht aus vier ADTs, welche alle in separaten Dateien als Klasse erstellt werden soll (die jeweiligen Datei- bzw. Klassennamen sind angegeben).

Zusätzlich soll für jede Klasse eine eigene JUnit-Testdatei in einem anderen Package erstellt werden. Dabei ist die Konvention Klassenname + Test (z.B. „adtListTest.*jar*“) einzuhalten.

Syntax der Methoden:

<Name der Methode>: <Parameter 1> × *<Parameter 2>* × ... → *<Modifiziertes Objekt>*

Sofern die Methode nicht statisch ist (in so einem Falle angegeben), ist der erste Parameter immer das referenzierte Objekt. Alle Methoden sollen *public* sein.

Das modifizierte Objekt ist nicht zwangsläufig der Rückgabetyp (dieser ist bei allem Methoden gesondert angegeben, i.d.R. *void*).

Eine konkrete Umsetzung sehe da z.B. so aus:

```
public void addVertex(vertex inputV) {}
```

Die Parameternamen sind irrelevant (wie hier *inputV*). Wichtig ist nur, dass die Spezifikationen eingehalten werden, also Methodennamen und die Objektmengen.

Ganz wichtig: Keine *throws*! Bei Exceptions soll die Ausführung der Methode ignoriert werden (die Rückgaben sind in der jeweiligen Operation dabei die unmodifizierten Werte)!

JUnit-Tests Anmerkungen:

- Alle Tests müssen die Spezifikationen einhalten (sowohl vom ADT als auch von JUnit selbst), damit sie später einfach austauschbar sind.
- Es sollen sowohl positive als auch negative Tests geschrieben werden, die jeweils überprüfen, ob die Anforderungen der Methode eingehalten werden bzw. wann sie explizit ausgeschlossen sind.
  - Hinweis: Nicht alle Methoden können negative Tests haben!
- Der Aufbau ist aller Testmethoden besteht aus dem Methodennamen und zum Schluss eine selbsterklärende Beschreibung, was der Testfall tut also z.B. „*testRetrieveWhenOutOfBounds*“.
- *Alle* Methoden sollen getestet werden und jede Methode soll eine eigene Testmethode haben.
- Einige Tests haben Sonderfälle, die bei der Implementierung bedacht werden sollen.

ADT Liste (adtList.jar)

Vorgabe:

Funktional (nach außen)

1. Die Liste beginnt bei Position 1 **(Anmerkung: wenn im Nachfolgenden von „natürlichen Zahlen“ gesprochen wird, ist damit die Zahlenfolge 1 bis n ohne 0 gemeint!)**
2. Die Liste arbeitet nicht destruktiv, d.h. wird ein Element an einer vorhandenen Position eingefügt, wird das dort stehende Element um eine Position verschoben.
3. Die Elemente sind vom Typ „ganze Zahl“.

Technisch (nach innen) :

1. Die Liste ist intern mittels eines Arrays zu realisieren (…new int[?]…).

Objektmengen: pos (int), elem (int), list (adt)

Operationen:

`(static) create: ∅ → list`

- Erzeuge ein neues *list*-Objekt
- Dabei muss intern ein primitives „int[]“-Array benutzt werden (siehe oben)
- Es muss eine Möglichkeit geben, die Länge zu speichern. Diese ist zu diesem Zeitpunkt auf 0 zu setzen.
- *Rückgabetyp: list*

`isEmpty: list → bool`

- Überprüfe, ob die Länge gleich 0 ist und gebe in dem Fall *true* zurück, sonst *false*
- *Rückgabetyp: bool*

`laenge: list → int`

- Getter-Methode, die Länge der Liste berechnet und zurückgibt
- *Rückgabetyp: int*

`insert: list × pos × elem → list`

1. Durchsuche die Liste und ignoriere die Operation, wenn das Element bereits vorhanden ist
2. Fallunterscheidung:
    1. *pos* ist keine natürliche Zahl:
        1. ignorieren und Liste nicht verändern
    2. *pos* liegt in der Mitte der Liste
        1. Erhöhe die Länge der Liste um 1
        2. Die neue Liste besteht aus drei Teilen:
            1. Den unveränderten Teil der alten Liste bis zu *pos*
            2. Das einzufügende Element selbst
            3. Den Rest der alten Liste (also, dass diese Elemente alle nun an der Position n+1 stehen)
    3. *pos* ist um 1 größer als die Liste
        1. Erhöhe die Länge der Liste um 1
        2. Füge das Element an das Ende der Liste an
    4. *pos* ist größer als n+1
        1. Nicht zulässig, ignorieren und Liste nicht verändern

- *Rückgabetyp: void*

`delete: list × pos → list`

1. Durchsuche die Liste und ignoriere die Operation, wenn das Element nicht vorhanden ist
2. Fallunterscheidung:
    1. *pos* ist keine natürliche Zahl oder *pos* ist größer als die Liste:
        1. ignorieren und nicht verändern
    2. *pos* liegt in der Mitte der Liste
        1. Verringere die Länge der Liste um 1
        2. Die neue Liste besteht aus zwei Teilen:
            1. Den unveränderten Teil der alten Liste bis zu *pos*
            2. Ohne dem Element an *pos*, den Rest der alten Liste (also, dass diese Elemente alle nun an der Position n-1 stehen)

- *Rückgabetyp: void*

`find: list × elem → pos`

- Gehe durch die Liste von Position 1 an los
- Erhöhe bei jedem Schritt den Zähler um 1
- Durchsuche solange die Liste, bis das Element gefunden wurde
- Übersteigt der Zähler die Länge der Liste (elem nicht gefunden), gebe -1 zurück
- *Rückgabetyp: int*

`retrieve: list × pos → elem`

- Wenn *pos* weder eine natürliche Zahl ist noch im Bereich der Liste liegt (also größer als diese ist), so gebe -99999999 zurück
- Ansonsten, gebe das Element an der angegebenen Position zurück
- *Rückgabetyp: int (weil Elemente hier als int spezifiziert sind)*

concat: list × list → list

- Erstelle eine neue Liste
- Kopiere alle Elemente der ersten Liste hinein und füge im Anschluss alle Elemente der zweiten Liste hinzu
- Optimierung (optional): Wenn eine der beiden Listen leer ist, gebe sofort eine Kopie der nicht leeren Liste zurück. Sollten beide leer sein, gebe eine leere Liste zurück.
- *Rückgabetyp: list*

ADT Stack (adtStack.jar)

Vorgabe:

Technisch (nach innen)

1. Die ADT Stack ist mittels ADT Liste zu realisieren.

Objektmengen: elem (int), stack (adt)

Operationen:

createS: Ø → stack

- Erzeuge einen neues Stack-Objekt
- Dieses ist zu diesem Zeitpunkt von der Größe 0
- *Rückgabetyp: stack*

push: stack × elem → stack

- Lege das neue Element oben auf dem Stack ab
- *Rückgabetyp: void*

pop: stack → stack

- Wenn der Stack leer ist, verändere den Stack nicht
- Ansonsten, entferne das neuste Element des Stacks
- *Rückgabetyp: void*

top: stack → elem

- Wenn der Stack leer ist, ignorieren
- Ansonsten, finde das neuste Element des Stacks und gebe dieses zurück
- *Rückgabetyp: int*

isEmptyS: stack → bool

- Überprüfe ob der Stack leer ist und gebe *true* zurück falls ja
- *Rückgabetyp: bool*

ADT Queue (adtQueue.jar)

Vorgabe:

Technisch (nach innen)

1. Die ADT Queue ist mittels ADT Stack, wie in der Vorlesung vorgestellt, zu realisieren. Es sind z.B. zwei explizite Stacks zu verwenden und das „umstapeln“ ist nur bei Zugriff auf einen leeren „out-Stack“ durchzuführen.

Objektmengen: elem (int), queue (adt)

Operationen:

`createQ: Ø → queue`

- Erstelle ein neues Queue-Objekt
- (Hinweis, aus der Vorlesung: Die IN- & OUT-Stack Umsetzung bietet sich hier an.)
- *Rückgabetyp: queue*

`enqueue: queue × elem → queue`

- Füge an das Ende der Queue ein neues Element ein
- *Rückgabetyp: void*

`dequeue: queue → queue`

- Wenn die Queue leer ist, ignorieren
- Ansonsten, entferne das älteste Element von der Queue
- *Rückgabetyp: void*

`front: queue → elem`

- Wenn die Queue leer ist, ignorieren
- Suche das älteste Element aus der Queue und gebe dieses zurück (lösche es aber nicht)
- *Rückgabetyp: int*

`isEmptyQ: queue → bool`

- Überprüfe, ob es kein Elementg in der Queue gibt, und gib in diesem Fall *true* zurück
- *Rückgabetyp: bool*

ADT Array (adtArray.jar)

Vorgabe:

Funktional (nach außen)

1. Das Array beginnt bei Position 0.
2. **Das Array arbeitet destruktiv,** d.h. wird ein Element an einer vorhandenen Position eingefügt, wird das dort stehende Element überschrieben.
3. Die Länge des Arrays wird bestimmt durch die bis zur aktuellen Abfrage größten vorhandenen und explizit beschriebenen Position im array.
4. Das Array ist mit 0 initialisiert, d.h. greift man auf eine bisher noch nicht beschriebene Position im Array zu erhält man 0 als Wert.
5. Das Array hat keine Größenbeschränkung, d.h. bei der Initialisierung wird keine Größe vorgegeben.

Technisch (nach innen)

1. Die ADT Array ist mittels ADT Liste zu realisieren.

Objektmengen: pos (int), elem (int), array (adt)

Operationen:

`initA: Ø → array`

- Erstelle ein neues ADT-Array-Objekt
- Das Array zu diesem Zeitpunkt eine Länge von 0
- Es wird noch nichts bezüglich Inhalt festgelegt
- Gebe das neu erzeugte Array zurück
- *Rückgabetyp: array*

`setA: array x pos x elem → array`

- Wenn *pos* größer ist als die bisherige Länge des Arrays, setze Länge := pos
  - Außerdem, setze für alle Felder dazwischen, die nun neu entstanden sind, den Inhalt auf 0 (weil die Liste keine Lücken enthält, müssen in diesem Array diese simuliert erzeugt werden)
- Be- bzw. überschreibe den Inhalt an der Stelle *pos* mit *elem*
- *Rückgabetyp: void*

`getA: array x pos → elem`

- Wenn *pos* weder eine natürliche Zahl ist noch im Bereich des Arrays liegt (also größer als dieses ist), so ignoriere die Operation
- Ansonsten, finde das Element und gib es zurück
- *Rückgabetyp: int*

`lengthA: array → pos`

- Finde das an der höchsten Position stehende Element und gebe dessen Index wieder (dies ist die Länge, siehe Vorgabe)
- *Rückgabetyp: int*