

Team: 03, Sebastian Diedrich – Murat Korkmaz

Aufgabenaufteilung:

- Aufgaben, für die Teammitglied 1 verantwortlich ist:
 - (1) SkizzeDateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:
 - (1) folgt
- Aufgaben, für die Teammitglied 2 verantwortlich ist:
 - (1) SkizzeDateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:
 - (1) folgt

Quellenangaben: Vorlesung am 08.10.15

Bearbeitungszeitraum: 02.10. (1h), 09.10. (2h), 16.10. (2h)

Aktueller Stand: Skizze fertig

Änderungen in der Skizze:

- Angabe des Rückgabetyps bei der Fehlerbehandlung,
- Syntaktische Vorgabe bei der Benennung der Datei, der Klasse und Aufruf von Operationen
- Besprechung der Testfälle
(INFO: Da die späteren Testfälle zwischen den Gruppen ausgetauscht werden sollen, sind sie nur grob umschrieben worden – damit sich die Testfälle am Ende nicht zu sehr gleichen)

Skizze: (ab Seite 2)

Skizze Aufgabe 1:

Invariante für **alle** ADTs:

- Alle Operationen der ADTs sind objekt-orientiert:
 - a) Konstruktor (K)
 - b) Mutatoren (M)
 - c) Selektoren (S)

Testfälle für **alle** ADTs:

- Alle Operationen der ADTs sollen nach folgenden Kriterien getestet werden:
 - a) Erstellen neuer Objekte
 - b) Einfügen und Löschen von Elementen (Anzahl: 1 bis 10.000)
 - c) Abfrage von Länge und Positionen der Elemente (falls vorhanden)
 - d) Abfrage, ob sich Elemente in der Datenstruktur befinden
 - e) Zusammenführen von ADTs gleichen Typs (falls diese Funktion vorhanden ist)

Aufgabe: 1.1

Ziel: ADT-Liste implementieren

Folgendes soll immer gelten (Invarianten):

- I) funktional
 - Die Liste beginnt bei Position 1
 - Die Liste arbeitet nicht destruktiv (kein Element wird überschrieben, sondern alle nachfolgenden Elemente werden nach rechts verschoben)
 - Der Element-Typ sind „ganze Zahlen“
- II) technisch
 - Interne Realisierung soll mittels Java-Array umgesetzt werden (...new int[?])...

Die ADT-Liste enthält folgende Objektmengen:

- pos: Position eines Elementes innerhalb der Liste
- elem: Ein Element der ADT-Liste
- list: Die ADT-Liste, mit ggf. enthaltenen Elementen

Folgende Operationen sollen bereitgestellt werden (semantische Signatur):

- *create*: eine leere ADT-Liste erstellen (K)
(„nichts“ -> list)
Fehlerbehandlung: ignorieren (es wird kein Fehler geworfen)
- *isEmpty*: Abfrage, ob ADT-Liste kein Element enthält (S)
(list -> Wahrheitswert)
Fehlerbehandlung: ignorieren

- *laenge*: Abfrage, wie viele Elemente die ADT-Liste enthält (S)
(list -> ganze Zahl)
Fehlerbehandlung: 0 zurückgeben (Typ: ganze Zahl)
- *insert*: Ein Element an selbstgewählter Position in die ADT-Liste einfügen (M)
(list x Position x Element -> list)
Fehlerbehandlung: ignorieren
- *delete*: Ein Element an selbstgewählter Position aus der ADT-Liste entfernen (M)
(list x pos -> list)
Fehlerbehandlung: ignorieren
- *find*: Abfrage, an welcher Position sich ein Element befindet (Abfrage links nach rechts) (S)
(list x elem -> pos)
Fehlerbehandlung: 0 zurückgeben (Typ: pos)
- *retrieve*: Element aus ADT-Liste an selbstgewählter Position zurückgegeben (S)
(list x pos -> elem)
Fehlerbehandlung: 0 zurückgeben (Typ: elem)
- *concat*: zwei ADT-Listen zusammenfügen (linke Liste wird um Elemente der rechten erweitert) (M)
(list1 x list2 -> list1)
Fehlerbehandlung: ignorieren

Syntaktische Vorgaben:

Dateiname: ADTListe.jar

Klassenname: ADTListe

Anwendung der oben genannten Operationen:

create: ADTListe.create()

isEmpty: <Objektnamen>.isEmpty()

laenge: <Objektnamen>.laenge()

insert: <Objektnamen>.insert(pos,elem)

delete: <Objektnamen>.delete(pos)

find: <Objektnamen>.find(elem)

retrieve: <Objektnamen>.retrieve(pos)

concat: <Objektnamen>.concat(<Objektnamen2>)

Aufgabe: 1.2

Ziel: ADT-Stack implementieren

Folgendes soll immer gelten (Invarianten):

- I) funktional
 - LIFO (jüngstes hinzugefügtes Element wird als erstes zurückgegeben)
- II) technisch
 - Der ADT-Stack soll mittels ADT-Liste implementiert werden

Der ADT-Stack enthält folgende Objektmenngen:

- elem: Ein Element des ADT-Stacks
- stack: Ein ADT-Stack, mit ggf. Elementen

Folgende Operationen sollen bereitgestellt werden (semantische Signatur):

- *createS*: ein leeren ADT-Stack erstellen (K)
(„nichts“ -> stack)
Fehlerbehandlung: ignorieren (es wird kein Fehler geworfen)
- *push*: ein Element auf den Stack ablegen, dieses Element liegt ganz oben (M)
(stack x elem -> stack)
Fehlerbehandlung: ignorieren
- *pop*: oberstes Element aus dem Stack löschen (M)
(stack -> stack)
Fehlerbehandlung: ignorieren
- *top*: oberstes Element zurückgeben (S)
(stack -> elem)
Fehlerbehandlung: 0 zurückgeben (Typ: elem)
- *isEmptyS*: Abfrage, ob ADT-Stack kein Element enthält (S)
(stack -> Wahrheitswert)
Fehlerbehandlung: ignorieren

Syntaxtische Vorgaben:

Dateiname: ADTStack.jar

Klassenname: ADTStack

Anwendung der oben genannten Operationen:

createS: ADTStack.createS()

push: <Objektnamen>.push(elem)

pop: <Objektnamen>.pop()

top: <Objektnamen>.top()

isEmptyS: <Objektnamen>.isEmptyS()

Aufgabe: 1.3

Ziel: ADT-Queue implementieren

Folgendes soll immer gelten (Invarianten):

- I) funktional
 - FIFO (ältestes hinzugefügtes Element wird als erstes zurückgegeben)
- II) technisch
 - Die ADT-Queue soll mittels ADT-Stack implementiert werden

Die ADT-Queue enthält folgende Objektmenngen:

- elem: Ein Element der ADT-Queue
- queue: Ein ADT-Queue, mit ggf. Elementen

Folgende Operationen sollen bereitgestellt werden (semantische Signatur):

- *createQ*: eine leere ADT-Queue erstellen (K)
(„nichts“ -> queue)
Fehlerbehandlung: ignorieren (es wird kein Fehler geworfen)
- *front*: gibt das älteste Element zurück (S)
(queue -> elem)
Fehlerbehandlung: 0 zurückgeben (Typ: elem)
- *enqueue*: fügt ein Element an letzter Position der ADT-Queue ein (M)
(queue x elem -> queue)
Fehlerbehandlung: ignorieren
- *dequeue*: löscht das älteste Element aus der ADT-Queue (M)
(queue -> queue)
Fehlerbehandlung: ignorieren
- *isEmptyQ*: Abfrage, ob ADT-Queue kein Element enthält (S)
(queue -> Wahrheitswert)
Fehlerbehandlung: ignorieren

Syntaxtische Vorgaben:

Dateiname: ADTQueue.jar

Klassenname: ADTQueue

Anwendung der oben genannten Operationen:

createQ: ADTStack.createQ()

front: <Objektnamen>.front()

enqueue: <Objektnamen>.enqueue(elem)

dequeue: <Objektnamen>.dequeue()

isEmptyQ: <Objektnamen>.isEmptyS()

Aufgabe: 1.4

Ziel: ADT-Array implementieren

Folgendes soll immer gelten (Invarianten):

- I) funktional
 - Das ADT-Array beginnt bei Position 0
 - Das ADT-Array arbeitet destruktiv (wird ein Element an einer vorhandenen Position eingefügt, wird das dort stehende Element überschrieben)
 - Die Länge des ADT-Arrays wird durch die Position bestimmt, an der das letzte (vom Benutzer eingefügte) Element indiziert ist
 - Das ADT-Array ist mit 0 initialisiert (Fehlerbehandlung siehe unten)
 - Das ADT-Array hat keine Größenbeschränkung
- II) technisch
 - Das ADT-Array soll mittels ADT-Liste implementiert werden

Die ADT-Array enthält folgende Objektmengen:

- elem: Ein Element des ADT-Arrays
- array: Ein ADT-Array, mit ggf. Elementen
- pos: Die Position eines Elements im ADT-Array

Folgende Operationen sollen bereitgestellt werden (semantische Signatur):

- *initA*: einen leeren ADT-Array erstellen (K)
(„nichts“ -> array)
Fehlerbehandlung: ignorieren (es wird kein Fehler geworfen)
- *setA*: Einfügen eines Elementes an gewählter Position (M)
(array x pos x elem -> array)
Fehlerbehandlung: ignorieren
- *getA*: Das Element der gewählten Position zurückgeben (S)
(array x pos -> elem)
Fehlerbehandlung: 0 zurückgeben (Typ: elem)
- *lengthA*: gibt als Länge die Position des letzten Elements im ADT-Array zurück (leeres ADT-Array hat die Länge -1) (S)
(array -> pos)
Fehlerbehandlung: 0 zurückgeben (Typ: pos)

Syntaktische Vorgaben:

Dateiname: ADTArray.jar

Klassenname: ADTArray

Anwendung der oben genannten Operationen:

initA: ADTStack.initA()

setA: <Objektnamen>.setA(pos,elem)

getA: <Objektnamen>.getA(pos)

lengthA: <Objektnamen>.lengthA()