SKIZZE

SKIZZE ZUR AUFGABE 1 AUS DER VORLESUNGSREIHE “ALOGRITHMEN UND DATENSTRUKTUREN”

TEAM 10: ANTON, MESUT UND IGOR

Aufgabenaufteilung

Alle Aufgaben wurden gemeinsam entworfen und bearbeitet.

Quellenangaben

Grundkurs Programmieren in JAVA 6. Auflage, Wikipedia, Lehrveranstaltung

Bearbeitungszeitraum

Datum	**Dauer in Stunden**
08.10.15	4
10.10.15	2
11.10.15	5
15.10.15	2
29.19.15	10
Gesamt	**23**

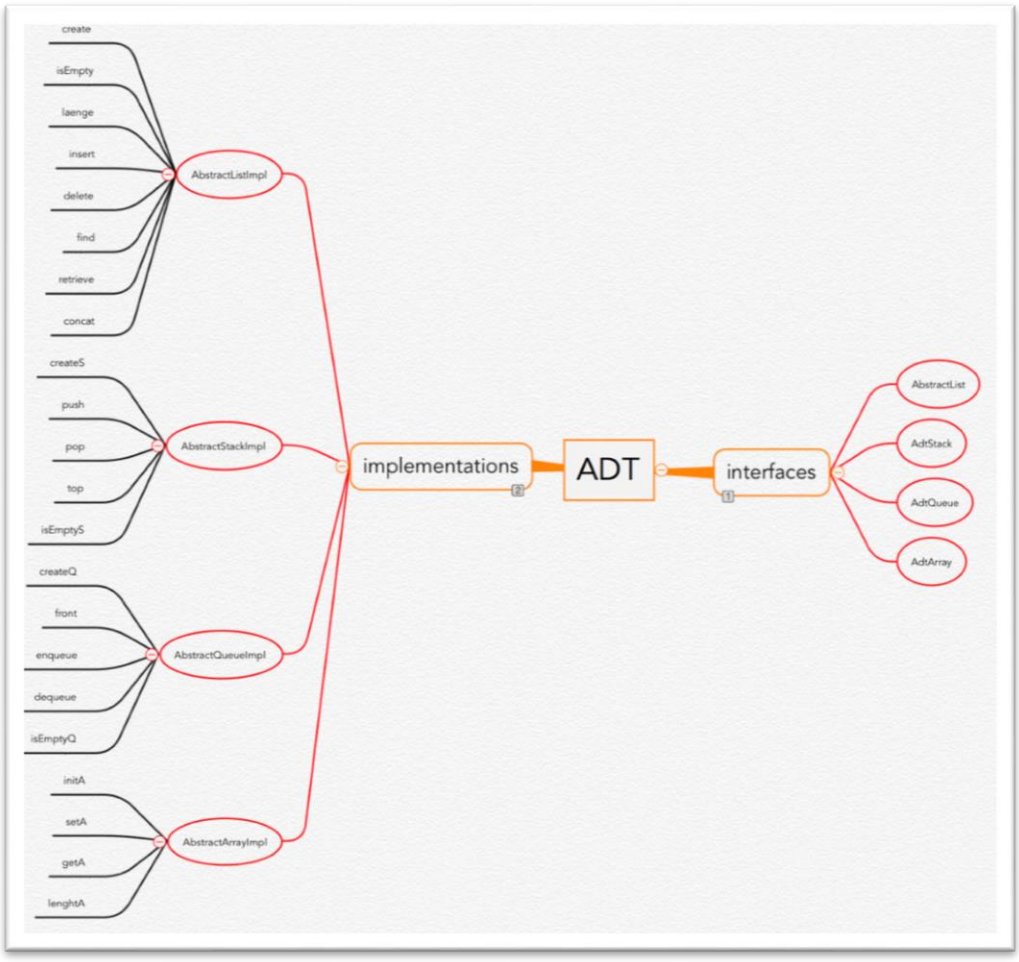
Aktueller Stand

Die Implementation wurde komplett neu überarbeitet, da wir die Aufgabe mit Interfaces gemacht haben und nicht funktional sondern imperativ. Die Tests von den anderen Gruppen funktionieren, unsere eigenen Tests haben wir überarbeitet und hinzugefügt.

Änderungen in der Skizze

Die Skizze wurde nicht weiter bearbeitet. Nur der aktuelle Stand wurde angepasst.

Skizze:



Es sollen folgende ADTs in Java implementiert werden: Liste („adtList.jar“), Stack („adtStack.jar“), Queue („adtQueue.jar“), Array („adtArray.jar“). Dazu müssen auch JUnit Tests erstellt werden: „adtListJUT.jar“, „adtStackJUT.jar“, „adtQueueJUT.jar“ und „adtArrayJUT.jar“. Die Dateien müssen genau diese Namen besetzen, damit sie mit den anderen Gruppen austauschbar bleiben. Es folgen nun die Beschreibungen und Signaturen der ADTs.

ADT Liste

Die Liste ist eine dynamische Datenstruktur, die eine Speicherung von miteinander in Beziehung stehenden Objekten erlaubt. Die Anzahl der Objekte ist im Vorhinein nicht bestimmt.

Funktionale Vorgabe

- Die Implementation ist durch die einfach verketteten Listen zu realisieren.
- Die Liste arbeitet nicht destruktiv (die Elemente sind im Notfall verschoben und nicht geändert).
- Die Elemente sind vom Typ „ganze Zahl“.

Technische Vorgabe

- Die ADT Liste ist intern mittels eines Java-Arrays zu realisieren.
- Die Indizierung der Liste fängt bei 1 an, anders als bei einem Array der den Startindexwert 0 hat.

Objektmengen

- pos (die Position des Elementes einer Liste), elem (das Element einer Liste), list (Das Objekt vom Typ ADT Liste).

Public Operationen (semantische Signatur/syntaktische Signatur):

create: ∅ → list

Syntax: { Input: void | Output: List list }

Mit Hilfe dieser Methode erzeugen wir eine neue Instanz der Klasse List.

isEmpty: list → bool

Syntax: { Input: List list | Output: boolean res}

Die Methode isEmpty prüft im Allgemeinen, ob die gegebene Liste mit Elementen belegt ist oder nicht. Im Fall einer leeren Liste liefert die Methode den Wert TRUE, sonst FALSE.

laenge: list → int

Syntax: { Input: List list | Output: int laenge }

Die Methode laenge gibt die Anzahl der Elemente in der vorgegebenen Liste aus. Leere Liste liefert den Integer-Wert 0 zurück.

insert: list × pos × elem → list

Syntax: { Input: List list1, int pos, int elem | Output: List list2 }

Bei dieser Funktion fügen wir Elemente in die Liste (die übrigens auch vorgegeben wird) ein. Wichtig ist, dass die angegebene Position eine gültige Position ist. Die Menge der gültigen Positionen fängt bei 1 an und läuft bis n+1, wo n die Länge dieser Liste ist. Sonst ist kein neues Element enthalten und die Methode liefert die unveränderte Liste zurück.

delete: list × pos → list

Syntax: { Input: List list1, int pos | Output: List list2 }

Bei dieser Methode löschen wir das auf der vorgegebenen Position stehende Element in der vorgegebenen Liste. Dabei sinkt die Anzahl der möglichen Positionen stets proportional in der Liste um 1. Der Löschvorgang ist ausschließlich auf die Position in der Liste zu begrenzen, da bei mehreren gleichen Elementen nicht alle Elemente, die gleich sind, gelöscht werden sollen. Falls die gegebene Liste leer oder die Position ungültig ist, liefern wir die Liste unverändert zurück.

find: list × elem → pos

Syntax: { Input: List list, int elem | Output: int pos }

Bei dieser Methode liefern wir die Position des vorgegebenen Elementes in der vorgegebenen Liste zurück. Beim Suchen einer Position kann es vorkommen, dass es gleiche Elemente gibt, sodass man als mögliche Ausgabe mehrere Positionen erhalten kann. Dies soll vermieden werden und es soll nur eine (erste von Anfang der Liste gefundene) Position geliefert werden. Falsche Abfragen sollen mit einer 0 bewertet werden, falls das gesuchte Element nicht existiert oder die Liste leer ist.

retrieve: list × pos → elem

Syntax: { Input: List list, int pos | Output: int elem }

Für die Methode retrieve gelten die gleichen Bedingungen wie bei find, jedoch auf das Element bezogen (die Position und das Element sind in dem Fall umgedreht im Vergleich zu find).

concat: list × list → list

Syntax: { Input: List list1, List list2 | Output: List list3 }

Bei concat vereinigen wir zwei bestehenden Listen. Die resultierende Liste enthält die Elemente aus der ersten Liste, danach folgen die Elemente der zweiten Liste. Die Reihenfolge (Nachfolger/Vorgänger) der Elemente bleibt dieselbe. Wenn eine der Listen oder beide Listen leer sind, besteht die resultierende Liste nur aus den Elementen der nichtleeren Liste oder je ist selbst leer.

ADT Stack

Der Stack bzw. Stapel ist ein Spezialfall der Liste, wo die Elemente nur von Oben (Top bzw. Kopf) abgelegt werden und auch nur von dort ausgelesen werden dürfen. Der Stack arbeitet nach dem LIFO-Prinzip (Last-In-First-Out), sodass das letzte gespeicherte Objekt den Stack auch als erstes verlässt.

Technische Vorgabe

- Der ADT Stack ist mittels ADT Liste zu realisieren.
- Die Elemente sind vom Typ „ganze Zahl“.

Objektmengen

- elem (das Element eines Stacks), stack (Das Objekt vom Typ ADT Stack).

Public Operationen (semantische Signatur/syntaktische Signatur):

createS: ∅ → stack

Syntax: { Input: void | Output: Stack stack }

Mit Hilfe dieser Methode erzeugen wir eine neue Instanz der Klasse Stack. Der Stack wird nicht auf eine bestimmte Anzahl der Elemente begrenzt.

push: stack × elem → stack

Syntax: { Input: Stack stack1, int elem | Output: Stack stack2 }

Diese Methode fügt ein neues Element als Top in Stack ein.

pop: stack → stack

Syntax: { Input: Stack stack1 | Output: Stack stack2 }

Bei dem Methodenaufruf wird zuerst überprüft, ob der vorhandene Stack leer ist. In solchem Fall ist wieder derselbe leere Stack zurückgegeben. Andernfalls wird das Top-Element aus dem Stack entfernt und der resultierende Stack zurückgeliefert.

top: stack → elem

Syntax: { Input: Stack stack | Output: int top }

Liefert das Top-Element des Stacks zurück. Das Element bleibt in dem Stack nach dem Aufruf enthalten. Bei dem Aufruf wird überprüft, ob der vorhandene Stack leer ist. In solchem Fall geben wir einen Integer Wert 0 raus.

isEmptyS: stack → bool

Syntax: { Input: Stack stack | Output: Boolean res }

Die Methode isEmpty prüft im Allgemeinen, ob der gegebene Stack überhaupt ein Top-Element besitzt. Im Fall eines leeren Stacks (kein Top-Element drin) liefert die Methode den Wert TRUE, sonst FALSE.

ADT Queue

Queue bzw. Schlange ist ein Spezialfall der Liste, wo die Elemente nur von hinterer Seite (Tail) abgelegt werden und nur von vorderer Seite (Kopf) ausgelesen werden dürfen nach dem FIFO-Prinzip (First-In-First-Out). Das bedeutet, dass die Objekte immer um 1 Position nach rechts geschoben werden, sobald ein neues Element hinzukommt.

Technische Vorgabe

- Die ADT Queue ist mittels ADT Stack zu realisieren.

- Es sind zwei explizite Stacks (in-Stack für Ablegung und out-Stack für Auslesen) zu verwenden und das „umstapeln“ ist nur bei Zugriff auf einen leeren „out-Stack“ durchzuführen.
- Bei der Umstapelung verschieben wir die Elemente aus dem in-Stack auf den out-Stack, dabei wird die Reihenfolge der verfügbaren Elemente des in-Stacks gespiegelt.
- Die Elemente sind vom Typ „ganze Zahl“.

Objektmengen

- elem (das Element einer Queue), queue (Das Objekt vom Typ ADT Queue).

Public Operationen (semantische Signatur/syntaktische Signatur):

createQ: ∅ → queue

Syntax: { Input: void | Output: Queue queue }

Mit Hilfe dieser Methode erzeugen wir eine neue Instanz der Klasse Queue. Die Queue wird nicht auf eine bestimmte Anzahl der Elemente begrenzt.

front: queue → elem (Selektor)

Syntax: { Input: Queue queue | Output: int front }

Gibt das erste Element der vorgegebenen Queue (das erste in Bezug auf dem Kopf, also das Top-Element des out-Stacks) zurück. Das Element bleibt in der Queue nach dem Aufruf bestehen. Bei dem Aufruf wird überprüft, ob die vorhandene Queue leer ist. In solchem Fall wird ein Integer Wert 0 zurückgeliefert.

enqueue: queue × elem → queue

Syntax: { Input: Queue queue1, int elem | Output: Queue queue2 }

Fügt das vorgegebene Element am Ende der vorgegebene Queue ein. Also, wird es auf dem in-Stack abgelegt.

dequeue: queue → queue (Mutator)

Syntax: { Input: Queue queue1 | Output: Queue queue2 }

Entfernt das erste Element der vorgegebenen Queue, also das Top-Element des out-Stacks und gibt die veränderte Queue zurück. Dabei kann der Out-Stack entweder leer oder nicht leer sein. Die Methode muss also auf diese Zustände reagieren können.

isEmptyQ: queue → bool

Syntax: { Input: Queue queue | Output: Boolean res }

Die Methode isEmptyQ prüft, ob die gegebene Queue mit Elementen belegt ist oder nicht (sowohl in- und out-Stack leer sind). Im Fall einer leeren Queue liefert die Methode den Wert TRUE, sonst FALSE.

ADT Array

Arrays (Felder) gestatten uns, mehrere Objekte über einen gemeinsamen Namen anzusprechen und nur durch einen Index zu unterscheiden. Dabei haben alle indizierten Elemente den gleichen Typ. Der Index selbst, der zum Unterscheiden und Ansprechen der Objekte dient, ist vom Typ Integer, wobei nur Werte größer oder gleich 0 (natürliche Zahlen) erlaubt sind.

Funktionale Vorgabe

- Das ADT Array beginnt bei der Position 0.
- Das ADT Array arbeitet destruktiv, d.h. wird ein Element an einer vorhandenen Position eingefügt, wird das dort stehende Element überschrieben.
- Die Länge des Arrays wird bestimmt durch die bis zur aktuellen Abfrage größter vorhandenen und explizit mittels beschriebener Position (Index) im Array.
- Das ADT Array ist mit 0 initialisiert, d.h. greift man auf eine bisher noch nicht beschriebene Position im Array zu, erhält man 0 als Wert.
- Das ADT Array hat keine Größenbeschränkung, d.h. bei der Initialisierung wird keine Größe vorgegeben.

Technische Vorgabe

- Das ADT Array ist mittels ADT Liste zu realisieren.
- Die Elemente sind vom Typ „ganze Zahl“.

Objektmengen:

- pos (die Position des Elementes eines Arrays), elem (das Element eines Arrays), array (Das Objekt vom Typ ADT Array).

Public Operationen (semantische Signatur/syntaktische Signatur):

initA: ∅ → array

Syntax: { Input: void | Output: Array array }

Mit Hilfe dieser Methode erzeugen wir eine neue Instanz der Klasse Array. Das Array wird nicht auf eine bestimmte Anzahl der Elemente begrenzt.

setA: array x pos x elem -> array

Syntax: { Input: Array array1, int pos, Int elem | Output: Array array2 }

Ersetzt das vorgegebene Element an bestimmter Position (Index) mit dem einzufügenden Element. Wenn die angegebene Position (Index) außerhalb jetziger Länge liegt, wird das Array verlängert, so dass das Element an die gewünschte Position eingefügt werden kann. Damit sind die nicht initialisierten Elemente inzwischen mit 0 angelegt.

getA: array x pos -> elem

Syntax: { Input: Array array, int pos | Output: int elem }

Gibt das Element an der angegebenen Position zurück. Falls die angegebene Position (Index) außerhalb jetziger Länge liegt, wird 0 zurückgegeben.

lengthA: array -> pos

Syntax: { Input: Array array | Output: int pos }

Die Methode lengthA gibt die Anzahl der Elemente in dem vorgegebenen Array aus. Das leere Array liefert den Integer-Wert 0 zurück.

Testen mit JUnit

- Alle obengenannten Methoden aufrufen.
- Jeder Aufruf mit Parameterübergabe soll 3 Zustände überprüfen: es gibt keine Elemente, es gibt genau ein Element, es gibt mehrere Elemente (auch große Datenmengen).
- Die fehlerhaften Situationen sollen berücksichtigt werden: z.B. die Einfügung des Elementes auf die Position -1, Löschen des Elementes auf der Position 1 nach dem Ende des vorgegebenen ADTs und ähnliche Fehlerhafte Situationen.