Team: 9, Tim Hagemann, Tim Hartig

Aufgabenaufteilung:

1. Implementation der Liste und Stack.
2. Implementation der Queue und Array.

Quellenangaben: Keine.

Begründung für Codeübernahme: Es wurde kein Code Übernommen.

Bearbeitungszeitraum: 15.10.14 12:30 – 22.10.14

Aktueller Stand: Skizze ist fertiggestellt und Verständnis ist vorhanden. Außerdem wurden Teile zum Testen des Verständnissen und der Realisierbarkeit implementiert.

ADT Liste

Die Liste ist ein Tupel, welches aus einem Typidentifier und einer Subliste besteht. Eine Subliste ist wiederum ein Tupel, welches entweder leer ist, um das Ende der Liste zu symbolisieren, oder aus einem beliebigen Element und einer weiteren Subliste besteht. Die Subliste ist also immer wieder in sich geschachtelt, bis das Ende der Liste erreicht ist.

create: Ø -> list

Erstellt eine Instanz von dem Listentupel mit einer leeren Subliste.

isEmpty : list -> bool

Es wird geprüft, ob die oberste, also erste, Subliste im Tupel die leere Subliste ist.

laenge: list -> int

Die Subliste wird durchiteriert, bis die leere Subliste erreicht ist. Die Länge der Liste der entspricht der Anzahl von Iterationsschritten.

insert: list x pos x elem -> list

Es wird über die Liste iteriert. Bei jedem Iterationsschritt wird das aktuelle Element der Ausgangsliste der Ergebnisliste angefügt. Falls man sich an der gewünschten Position in der Ausgangsliste befindet, werden zuerst das einzufügende Element und anschließend das aktuelle Element der Ausgangsliste an die Ergebnisliste gehängt. Ab diesem Punkt werden nur noch die restlichen Elemente der Ausgangsliste angefügt. Falls sich der Index der Position, an dem das Element eingefügt werden sollte, außerhalb der Länge der Liste befindet, wird das Element einfach an das Ende der Liste angefügt.

delete: list x pos -> list

Delete arbeitet ähnlich wie insert, der Unterschied liegt darin, dass anstatt des Einfügens eines neuen Elements, wird das Element am gewünschten Index ausgelassen und einfach übergangen. Danach läuft das anfügen der Elemente normal weiter.

find: list x elem -> pos

Es wird so lange über die Liste iteriert, bis das aktuelle Element der Liste dem gesuchten Element entspricht. Zurückgegeben wird die Anzahl notwendiger Iterationsschritte, um das Element zu finden, die der Position des Elements in der Liste entspricht.

retrieve: list x pos -> elem

Gibt das Element, das an der angegebenen Position in der Liste steht, zurück. Der Index **pos** entspricht der Anzahl Iterationsschritten, die notwendig sind, um zur gewünschten Stelle in der Liste zu gelangen, von der man das Element zurückgibt.

concat: list x list -> list

Es wird die erste Liste auf einen Iterator geschrieben. Danach wird die zweite Liste mit dem gleichen Verfahren darüber gesetzt.

ADT Stack

Der Stack besteht aus einem Typ und unserer Liste. Dabei repräsentiert das zuletzt eingefügte Element das oberste Element des Stacks.

createS: ∅ -> stack

Erzeugt eine neue Instanz des Tupels mit einer leeren Liste.

push: stack x elem -> stack

Fügt ein Element am Ende der Liste ein.

pop: stack -> stack

Entfernt das zuletzt hinzugefügte Element vom Stack und gibt diesen zurück. Beim leeren Stack passiert nichts.

top: stack -> elem

Gibt das zuletzt hinzugefügte Element des Stacks zurück (ohne den Stack zu verändern). Ist der Stack leer, passiert nichts.

isEmptyS: stack -> bool

Der Stack ist leer, wenn die interne Liste leer ist.

ADT Queue

Eine Queue besteht aus zwei Stacks - einem In-Stack und einem Out-Stack. Beim Hinzufügen eines Elements wird dieses im In-Stack abgelegt. Durch die Realisierung über zwei Stacks wird gewährleistet, dass das zuerst hinzugefügte Element auch als erstes wieder aus der Queue entnommen wird (FIFO-Prinzip). Es wird solange auf den In-Stack geschrieben, bis eine Leseoperation auf den Out-Stack ausgeführt wird. Dann wird der Inhalt des In-Stacks auf den Out-Stack umgeschichtet, und das erste Element des Out-Stacks gelesen. Ist der Out-Stack nicht leer, wird nur das erste Element des Out-Stacks gelesen, ohne den Inhalt aus dem In-Stack umzuschichten.

createQ: Ø -> queue
Erzeugt eine neue Instanz des Tupels mit zwei leeren Stacks (IN & OUT).

front: queue -> elem
Gibt das erste Element der Queue (--> oberstes Element des Out-Stacks) zurück, ohne es zu entfernen.

enqueue: queue x elem -> queue
Reiht das Element am Ende der Queue ein. D.h., es wird auf dem In-Stack abgelegt.

dequeue: queue -> queue
Entfernt das erste Element der Queue, also das oberste Element des Out-Stacks und gibt sie zurück.

isEmptyQ: queue -> bool
Prüft, ob sowohl In- und Out-Stack leer sind.

ADT Array

Ein Array besteht aus einem Typ und einer Liste, die standardmäßig leer ist. Das Array hat keine festgelegte Länge und kann somit immer wachsen, wenn nötig. Das Array beginnt bei der Position 0.

initA: Ø -> array
Erzeugt eine neue Instanz des Tupels mit einer leeren Liste.

setA: array x pos x elem -> array
Ersetzt das Element an angegebener Position mit dem einzufügenden Element. Wenn die angegebene Position über die Länge der aktuellen Liste hinaus geht, wird die Liste soweit wie nötig verlängert, so dass das Element an der gewünschten Position eingefügt werden kann.

getA: array x pos -> elem
Gibt das Element an der angegebenen Position zurück. Falls die angegebene Position über die aktuelle Länge hinaus geht, wird 0 zurückgegeben.

lengthA: array -> pos
Gibt die Länge der internen Liste zurück.

Betreff: Re: Skizze AD Gruppe 3 Team 9
Von: Christoph Klauck <christoph.klauck@haw-hamburg.de>
Datum: 23.10.2014 16:58
An: Tim Hartig <tim.hartig@haw-hamburg.de>, "'Hagemann, Tim'" <tim.hagemann@haw-hamburg.de>

Hallo zusammen!
ok angekommen.
Die Liste in der Gestalt 2-Tupel mit Element und Subliste ist zwar auf der einen Seite auch in Java realisierbar, jedoch ist unklar, warum Sie dieses Detail vorgeben. Wenn der Entwickler das nicht nachvollziehen kann, kann es sein, dass er es einfach ändert, da es keine Außenwirkung hat! Dann stimmen Skizze und Implementierung nicht mehr überein, was praktisch die Skizze im Nachhinein wertlos werden lässt.
Auf der anderen Seite sind Sie bei wichtigen Dingen sehr oberflächlich: muss man beim umschichten der Stapel etwas beachten? bzgl. Reihenfolge? Was ist im Fehlerfall, wenn zB retrieve auf eine ungültige Position zugreift?

Ihre Skizze ist daher in dem Sinne nicht "balanciert", da die Detailstufe unterschiedlich gehandhabt wurde.

Die Skizze ist angenommen.

MfG
C Klauck
Am 23.10.2014 um 09:42 schrieb Tim Hartig:

> Guten Morgen Herr Klauck,
>
> anbei erhalten Sie unsere angepasste Skizze.
>
> Einen schönen Tag noch und viele Grüße,
> Tim Hartig und Tim Hagemann
>
> ---
>
> **Von:** Christoph Klauck [mailto:christoph.klauck@haw-hamburg.de]
> **Gesendet:** Mittwoch, 22. Oktober 2014 10:49
> **An:** Hagemann, Tim
> **Cc:** Hartig, Tim
> **Betreff:** Re: Skizze AD Gruppe 3 Team 9
>
> Hallo zusammen!
> ok angekommen.
>
> Hier Anmerkungen zu Ihrer Skizze:
> - den Schnittstellen fehlen die Signaturen!
> - "Es wird über die Liste rekursiv iteriert mit zwei Akkumulatoren" ist ein Implementierungsdetail und hat so nichts in der Skizze zu suchen. Dazu aus Clean Code Developer: "Entwurf und Implementation müssen dem „Don't Repeat Yourself“-Prinzip gerecht werden. Deshalb sollten Entwurf und Implementation sich so wenig überlappen wie möglich."

> z.B. "wird die Liste bis zur angegebenen Position mit Nullen aufgefüllt" ist Implementierung, da Sie vorgeben, wie es zu implementieren ist. Wenn stattdessen gefodert wird, dass das array mit 0 iniztialisiert wurde, kann man das zB auch so implementieren, dass man bei nicht explizit gefüllten Positionen eine 0 zurück gibt, ohne diese als Elemente auch darstellen zu müssen.
>
> Bitte bis morgen 12:00 korrigieren: Signaturen aufnehmen und Implementierung raus, jedoch die funktionale erwatere Verhaltensweise der Schnittstellen drin lassen.
>
> MfG
> C Klauck
> Am 22.10.2014 um 10:01 schrieb Hagemann, Tim:
>
> > Guten Tag Herr Klauck,
> >
> > im Anhang befindet sich unser Skizze für den ersten Praktikumstermin.
> >
> > Mit freundlichen Grüßen,
> > Tim Hagemann und Tim Hartig