HSLU Hochschule Luzern

Algorithmen & Datenstrukturen (AD)

Übung SW02: Arrays, Listen, Stack und Queue

Themen:	Einfach verkettete Liste, doppelt verkettete Liste, Modellierung und Implementation, Generics, Iterator, Schnittstellen, Arrays, Rekursion.
Zeitbedarf:	ca. 270 Minuten
Wichtig:	Ziel ist, ausgewählte Datenstrukturen exemplarisch selbst zu implementieren, um deren Aufbau zu verstehen, und das algorithmische Denken zu üben. *Hinweis:* In der Praxis vermeidet man eigene Implementationen und verwendet stattdessen möglichst immer bereits vorhandene und erprobte Implementationen!

Ingmar Baetge, Version 1.8 (HS25)

1 Implementation einer einfach verketteten Liste (ca. 90')

1.1 Lernziel

- Implementation einer einfach verketteten Liste.
- Funktionsweise der typischen Operationen auf einer Liste verstehen.
- Systematische, schrittweise Implementation und Testen von Code.

1.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D12**.

1.3 Aufgaben

a.) Implementieren und testen Sie als erstes die Klasse, welche in Ihrem Modell ein einzelnes Element in der Liste repräsentieren soll (im Input finden Sie dazu ein UML-Diagramm, dort wird diese Klasse `Node` genannt). Als Daten sollen in der Liste zunächst Integer-Werte verwaltet werden.

b.) Implementieren Sie nun schrittweise die Klasse für die Liste. Vorerst soll nur **ein einziges** Element eingefügt werden können. Ob ein Element enthalten ist, können Sie z.B. über eine Methode `size()` abfragen. Testen Sie Ihre Implementation **immer** mit Unittests!

c.) Erweitern Sie die Methode zum Einfügen eines Elementes so, dass Sie auch mehr als ein Element (jeweils am Anfang der Liste) einfügen können. Nun müssen Sie aufpassen, dass Sie alle notwendigen Referenzen und Attribute in der richtigen Reihenfolge anpassen. Erweitern Sie Ihre Tests entsprechend.

d.) Ergänzen Sie als nächstes eine Methode, um zu prüfen, ob ein bestimmtes Element in der Liste enthalten ist. Dazu müssen Sie bereits intern über die Liste iterieren können.

e.) Ergänzen Sie eine Methode, welche das jeweils erste Element aus der Liste liefert und gleichzeitig aus der Liste entfernt! Welche Datenstruktur (bzw. welche Semantik) haben Sie nun aktuell implementiert?

f.) Ergänzen Sie eine weitere Methode, mit welcher das erste Vorkommen eines beliebigen Elements (welches Sie als aktuellen Parameter übergeben) aus der Liste entfernt, sofern dieses enthalten ist. Testen Sie hier besonders gewissenhaft, denn es gibt dabei mindestens vier (!) verschiedene Fälle, die auftreten können.
Tipp: Behelfen Sie sich mit einer Skizze einer konkreten Liste mit mindestens drei Elementen, um diese vier Fälle besser analysieren zu können.

g.) Passen Sie Ihre Klassen so an, dass sie generisch für beliebige Typen nutzbar ist! Wichtig: Zum Vergleich von Datenelementen müssen Sie jetzt die **`equals()`**-Methode verwenden. Wir setzen dafür voraus, dass die enthaltenen Datenelemente den equals-Contract einhalten! Auch hier ergänzen Sie wieder weitere Testfälle.

1.4 Optionale Aufgaben

h.) Programmieren Sie eine Klasse, welche das Iterator-Interface passend für Ihre Listen-Implementation implementiert, und es somit erlaubt, wie bei einer «normalen» Liste von Java über alle Elemente zu iterieren.

i.) Implementieren Sie mit Ihrer Listenklasse das **`Collection`**- oder gar das **`List`**-Interface von Java. Sie können dann in allen Aufgaben, welche eine entsprechende Datenstruktur verwendet, uneingeschränkt Ihre eigene Implementation verwenden!

2 Implementation eines Stacks mit Hilfe eines Arrays (ca. 60')

2.1 Lernziel

- Verständnis der Funktionsweise eines Stacks.
- Implementation als statische Datenstruktur.
- Schrittweise Entwicklung und kontinuierliches Testen.

2.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D12**.

2.3 Aufgaben

a.) Entwerfen Sie mit Hilfe eines UML-Klassendiagrammes eine Schnittstelle für einen Stack welcher Strings enthalten kann. Überlegen Sie: Welchen Einfluss auf die zur Verfügung gestellten Methoden hat die Tatsache, dass wir den Stack mit einem Array (→ statische Datenstruktur) implementieren werden?

b.) Programmieren Sie als erstes das Interface und dokumentieren Sie die Methoden mit JavaDoc. Erstellen Sie eine erste (noch weitgehend leere) Klasse, welche die Schnittstelle des Stacks nur gerade so weit implementiert, dass sie kompilierbar ist.

c.) Implementieren Sie die folgenden Abläufe mit Unit Tests (also keine `main()`-Methode!):

- Stack erstellen (leer): Explizit prüfen, ob dieser leer ist.
- Stack erstellen, ein Element einfügen: Prüfen, ob der Stack nicht leer ist.
- Stack mit Platz für ein Element anlegen, ein Element einfügen: Prüfen ob voll.

d.) Führen Sie die Testfälle aus. Es sollte Sie nicht überraschen: Eine Mehrheit der Testfälle wird aufgrund der noch fehlenden Implementationen failen. Ergänzen Sie nun schrittweise die Implementation, bis alle vorhandenen Testfälle «grün» sind!

Hinweis: Sie erleben gerade «test driven development» (tdd)[1]. Versuchen Sie ab sofort immer nach diesem Muster vorzugehen: Was will ich erreichen, Testfall/-fälle schreiben, und erst dann implementieren, bis alles grün ist! Das hat sehr viele Vorteile, und macht auch deutlich mehr Spass!

e.) Vervollständigen Sie nun die Funktionalität des Stacks und testen Sie diesen vollständig.

f.) Wie reagieren Ihre `push()`- und die `pop()`-Methoden, wenn der Stack voll bzw. leer ist? Es gibt im Wesentlichen zwei verschiedene Varianten, welche sind das? Überlegen Sie sich auch: Welche Variante wäre Ihnen aus Sicht der Nutzer*in eines Stacks lieber?

2.4 Optionale Aufgaben

g.) Schreiben Sie eine kleine Demo-Anwendung (`main()`-Methode in eigener Klasse), welche Ihren Stack verwendet, um die folgenden Wörter in umgekehrter Reihenfolge auf der Konsole auszugeben: «toll», «sind», «Datenstrukturen». 😊

h.) Setzen Sie Ihre Überlegungen zum Exception-Handling aus der Teilaufgabe f) konkret um!

i.) Passen Sie das Interface und ihre Implementation des Stacks an, dass er generisch für beliebige Typen nutzbar ist! Denken Sie daran: Arrays in Java sind **nicht** generisch.

[1] In OOP haben Sie das erstmals und vereinfacht als «Testfirst» kennengelernt.

3 Stackmaschine (ca. 60')

3.1 Lernziel

- Anwendung eines Stacks
- Achtung: Nächste Woche wird eine weitere Aufgabe auf diese Aufgabe aufbauen!

3.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D12**.

3.3 Aufgaben

Stellen wir uns vor, wir hätten eine sehr einfache Programmiersprache *J*, die aus den Befehlen LOAD x, ADD, MUL und PRINT besteht. Eine sogenannte Stackmaschine führt diese Programme aus und nutzt dafür einen Stack, auf dem natürliche Zahlen gespeichert werden. Dabei bedeuten die einzelnen Befehle folgendes:

Befehl	Beschreibung
LOAD x	Eine Zahl x auf den Stack legen (x ist eine positive natürliche Zahl)
ADD	Zwei Zahlen vom Stack nehmen, addieren und das Ergebnis auf den Stack legen
MUL	Zwei Zahlen vom Stack nehmen, multiplizieren und das Ergebnis auf den Stack legen
PRINT	Das oberste Stack-Element vom Stack nehmen und auf dem Bildschirm ausgeben

Um beispielsweise den Ausdruck $(2 + 3) * 4$ zu berechnen und das Ergebnis auf dem Bildschirm auszugeben, können wir nun folgendes Programm in *J* schreiben:

```
LOAD 2
LOAD 3
ADD
LOAD 4
MUL
PRINT
```

a.) Unser Beispielprogramm soll nun ausgeführt werden. Zeichnen Sie den Zustand des Stacks nach jeder Zeile.

b.) Geben Sie ein Stack-Programm für den Ausdruck $(4 + 5) * (2 + 3)$ an.

c.) Wie könnte man die Maschine erweitern, so dass auch Subtraktion und ganzzahlige Division (SUB / DIV) unterstützt werden?

Geben Sie ein Stack-Programm für den Ausdruck $5 * (6/(7 - 4))$ an.

d.) Schreiben Sie ein Java-Programm, welches ein solches Stack-Programm ausführt. Das Programm wird dabei in einer Liste von Strings abgelegt und dann zeilenweise ausführt. Für die Liste und den Stack können Sie entweder Ihre eigenen Klassen oder die Collections-Klassen von Java verwenden. Testen Sie Ihre Implementierung mit den Stack-Programmen aus a), b) und c).

4 Zweidimensionale Arrays, rekursive Suche (ca. 60')

4.1 Lernziel

- Arrays mit 2 Dimensionen verwenden
- Umsetzung einer rekursiven Suchstrategie

4.2 Grundlagen

Diese Aufgabe basiert auf dem Input **E12** und **D12**.

4.3 Aufgaben

In dieser Aufgabe ist ein Code-Fragment auf Ilias und ein Algorithmus im Pseudocode gegeben.

a) Laden Sie die Datei **`MazeSolver.java`** von der Ilias-Seite und binden Sie diese in Ihr Projekt ein. Sie müssen dazu noch den Package-Namen passend zu Ihrer Umgebung anpassen.

b) In dem Code finden Sie ein Labyrinth, implementiert als zweidimensionales Array, welches **`char`**-Elemente enthält. Dabei haben die Zeichen folgende Bedeutungen:

- Ein Leerzeichen steht für ein begehbares Feld
- # steht für eine Wand
- **`X`** steht für das Zielfeld oder den Ausgang des Labyrinths
- Ein Punkt markiert Felder, die bereits besucht wurden

Als Startfeld wird das Feld (0, 0) gewählt, dabei bedeutet die erste Zahl die Zeilennummer und die zweite Zahl die Spaltennummer. Gültige Richtungen im Labyrinth sind West, Ost, Nord und Süd, d.h. es sind keine diagonalen Bewegungen erlaubt.

Wie viele verschiedene Wege vom Startfeld zum Zielfeld finden Sie, wenn jedes Feld nur einmal betreten werden darf? Beobachten Sie sich selbst – wie gehen Sie dabei vor?

c) Implementieren Sie die Methode **`printMaze()`**, welche das Labyrinth auf der Konsole ausgibt.

d) Folgender Algorithmus sucht Wege durch das Labyrinth:

```
Funktion findPath(Zeile, Spalte):
    Wenn die Zeilen- oder Spaltennummer ungültig ist, dann
        beende die Funktion

    Wenn die Position (Zeile, Spalte) «X» enthält:
        Gebe das gesamte Labyrinth aus und beende die Funktion

    Wenn die Position (Zeile, Spalte) ein Leerzeichen enthält:
        Setze die Position (Zeile, Spalte) auf «.»
        Rufe «findPath» rekursiv auf allen Nachbarfeldern auf
        Setze die Position (Zeile, Spalte) wieder auf ein Leerzeichen.
```

Gestartet wird die Funktion mit dem Aufruf **`findPath(0, 0)`**. Setzen Sie den Algorithmus in Java um und überprüfen Sie, ob dieser Wege im Labyrinth findet.

e) Sie haben vielleicht bemerkt: Dieser Algorithmus findet alle Wege durch das Labyrinth. Wie kann man den Algorithmus verändern, so dass dieser bereits nach dem ersten gefundenen Weg endet?

f) **Optional:** In Ilias finden Sie eine Datei **`largeMaze.txt`** mit einem grösseren Labyrinth – nutzen Sie Ihr Programm, einen Weg durch das Labyrinth zu finden! Wie viele Wege durch dieses Labyrinth gibt es?

5 Optional: Implementation einer Queue mit Array als Ringbuffer (ca. 60')

5.1 Lernziel

- Array als Ringbuffer verwenden.
- Verstehen wie der `head`- und `tail`-Index berechnet und verwendet werden.
- Erkennen, wenn die Queue voll und leer ist.

5.2 Grundlagen

Diese Aufgabe basiert auf dem Input **D12**.

5.3 Aufgaben

Überlegen und konzipieren Sie zuerst in Ruhe das Design:

a.) Wir wollen eine Queue für acht einzelne Zeichen (`char`). Wir benötigen je einen Zeiger (Index) für den Kopf der Queue (wo wir ggf. ein Element entnehmen können) und für das Ende der Queue (wo wir ggf. weitere Elemente anhängen). Machen Sie eine Skizze der Datenstruktur als Array.

b.) Gehen Sie vorerst davon aus, dass die Queue leer ist (wir haben also Platz). Wie läuft das Einfügen genau ab? Auf welche Position soll der Index jeweils genau zeigen?
Wie läuft das Einfügen ab, wenn der Index zufällig auf den letzten Platz im Array zeigt `(length - 1)` und somit die Rotation passieren soll? Skizzieren Sie!

c.) Gehen Sie davon aus, dass die Queue bereits einige Elemente enthält. Wie soll das Entnehmen genau ablaufen? Auch hier stellt sich die Frage: Auf welche Position soll der Index für den Kopf der Queue zeigen? Skizzieren Sie auch hier!

d.) Wie lösen wir die Anforderung, dass der Index für das Ende der Queue den Index für den Kopf der Queue auf keinen Fall überholen darf? Was würde passieren, wenn doch?
Hinweis: Es wäre ohnehin nützlich zu wissen wie viele Elemente die Queue enthält!

Implementation, immer schrittweise getestet:

e.) Entwerfen Sie zuerst wieder eine Schnittstelle für die Queue. Welche Methoden wollen wir anbieten? Beginnen Sie erst danach mit der Implementation.

f.) Implementieren Sie als erstes die `toString()`-Methode der Queue-Implementation und geben Sie alle relevanten Attribute (auch den Inhalt des Arrays) zurück. Damit können Sie z.B. mit Hilfe von Log4J sehr einfach debuggen, in dem Sie am Ende jeder Operation den aktuellen Zustand ausgeben!

g.) Implementieren Sie die Methoden des Interfaces Stück für Stück und testen Sie fortlaufend bis alles korrekt funktioniert.

h.) Auch bei der Queue stellt sich die Frage was passieren soll, wenn man versucht ein Element aus einer leeren Queue zu entnehmen, oder ein Element in eine bereits volle Queue einzufügen. Wie entscheiden Sie sich?

i.) Auch diese Datenstruktur können Sie generisch implementieren!