Algorithmen & Datenstrukturen (AD)

Übung: Arrays, Listen, Stack und Queue (D1)

Themen: Einfach verkettet Liste, doppelt verkettete Liste, Modellierung und Implementation, Generics, Iterator, Schnittstellen.

Zeitbedarf: ca. 240min.

Wichtig: Ziel dieser Aufgaben ist es, ausgewählte Datenstrukturen mindestens teilweise selber zu implementieren, um deren Funktion und Aufbau besser zu verstehen sowie das algorithmische Denken und auch das Programmieren zu üben.
In der Praxis vermeidet man eigene Implementationen und verwendet stattdessen möglichst die bereits vorhandenen, erprobten und optimierten Implementationen.

Roland Gisler, Version 1.0 (FS 2017)

1 Modellierung von Listen (ca. 30')

1.1 Lernziele

- Objektorientierte Modellierung von Listen.
- Visualisierung mit UML

1.2 Grundlagen

Diese Aufgabe basiert auf dem Input D12. In dieser Aufgabe schreiben Sie (noch) keinen Code, sondern entwerfen ein objektorientiertes Modell für eine Liste.

1.3 Aufgaben

a.) Im Input finden Sie ein konzeptionelles Modell einer einfach verketteten Liste. Entwerfen Sie mit einem skizzierten UML-Klassendiagramm ein konkreteres Modell. Wir möchten in unsere Liste Objekte ablegen können, konkret vom Typ `Allocation` aus der Übung E0. Im Minimum müssen wir Elemente in die Liste einfügen und wieder entfernen können.
Hinweis: Ein konkretes Modell sollte die notwendigen Klassen, deren Attribute (mit Typ und Sichtbarkeit) und deren Methoden (mit Signatur, Rückgabewert und Sichtbarkeit) enthalten.

b.) Wir möchten über alle Elemente der Liste iterieren können. Ist das in ihrem aktuellen Modell möglich? Und entspricht das dem Konzept der Datenkapselung und des Information Hiding?

c.) Das Collection Framework von Java nutzt das Iterator-Konzept und Interface um eine Nutzerin einer Datenstruktur über alle Element iterieren zu lassen. Versuchen Sie dieses Konzept in ihrem Modell zu integrieren!
Hinweis: Überlegen Sie sich dazu genau wer (welche Klasse) muss was können (→Methoden ergänzen) und auf welche Attribute Zugriff haben?

d.) Erstellen Sie ein zweites Modell für eine doppelt verkettete Liste. Machen Sie sich die dafür notwendigen Erweiterungen oder Änderungen deutlich sichtbar.
Hinweis: Auf die Integration des `ListIterator` können Sie dabei verzichten.

e.) Optional: Modellieren Sie Ihre Listen als generische Liste. Welche Methoden und Attribute benötigen welche Typparameter?

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

2.1 Lernziel

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

2.2 Grundlagen

Diese Aufgabe basiert auf dem Input D12 und der Aufgabe 1.
Hinweis: Je gewissenhafter Sie die Aufgabe 1 gelöst haben, umso leichter wird es ihnen fallen, zumindest das Grundgerüst für diese Aufgabe zu implementieren. Sollten Sie Fehler in Ihrem Modell entdecken, korrigieren Sie diese auch in der Aufgabe 1, so dass das Modell und der Code möglichst übereinstimmen!

2.3 Aufgaben

a.) Implementieren und testen Sie als erstes die Klasse welche in Ihrem Modell ein Element in Ihrer Liste repräsentiert.

b.) Implementieren Sie nun die eigentliche 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 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.

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. Achtung: Wir setzen dafür voraus, dass die enthaltenen Datenelemente den Equals-Contract einhalten!

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

f.) Ergänzen Sie eine weitere Methode, mit welche ein beliebiges Element (welches Sie als aktuellen Parameter übergeben) aus der Liste entfernt, sofern dieses enthalten ist. Testen Sie gewissenhaft, denn es gibt dabei mindestens vier verschiedenen Fälle/Situationen die auftreten können. Tipp: Behelfen Sie sich mit einer Skizze der Listenstruktur um diese Fälle besser zu erkennen.

g.) Und nun die Herausforderung: 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.

h.) Optional für Fortgeschrittene: 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, Ihre eigene Implementation verwenden!

3 Implementation eines Stacks mit Hilfe eines Array (ca. 60')

3.1 Lernziel

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

3.2 Grundlagen

Diese Aufgabe basiert auf dem Input D12.

3.3 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 soweit implementiert, dass diese kompilierbar ist.

c.) Implementieren Sie nun die folgenden Abläufe als einzelne Unittests:
- einen Stack erstellen (leer); explizit prüfen ob dieser leer ist.
- einen Stack erstellen, ein Element einfügen; prüfen ob der Stack nicht leer ist.
- einen Stack mit Platz für ein Element anlegen, ein Element einfügen; prüfen ob voll.

Hinweis: Das Prüfen erfolgt natürlich immer über entsprechende Methoden des Stacks die aufgerufen werden, und deren Resultate mit `assert*`-Statements verglichen werden.

d.) Führen Sie die Testfälle aus. Es sollte Sie nicht überraschen: Eine Mehrheit der Testfälle wird aufgrund der noch fehlenden Implementation failen. Ergänzen Sie nun Schrittweise die Implementation, bis alle vorhandenen Testfälle «grün» sind!
Hinweis: Sie erleben gerade wieder «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 viele Vorteile und macht auch mehr Spass!

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

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 AnwenderIn des Stacks lieber? Passen Sie Ihre Implementation mindestens so an, dass ein konsistentes Verhalten ohne Fehler resultiert.

g.) Schreiben Sie eine kleine Demo-Anwendung (**`main()`**-Methode in eigener Klasse), welche Ihren Stack verwendet um die folgende Wortfolge umzudrehen und auf der Konsole auszugeben: «toll», «sind», «Datenstrukturen». ;-)

h.) Ergänzen Sie Ihre Implementation mit einer (eigenen) `StackFullException`.
Was ist besser geeignet: Eine Implementation als checked oder als unchecked Exception?

i.) Optional: Implementieren Sie Ihren Stack generisch.

---

[1] Mehr zu diesem Thema werden Sie im Modul VSK erfahren.

4 Implementation einer Queue mit Array als Ringbuffer (ca. 60')

4.1 Lernziel

- Array als Ringbuffer verwenden.
- Verstehen wie Head- und Tail-Index berechnet und verwendet werden.
- Erkennen, wenn die Queue voll und leer ist.

4.2 Grundlagen

Diese Aufgabe basiert auf dem Input D12.

4.3 Aufgaben

Überlegen und konzipieren Sie zuerst in Ruhe «auf Papier»:

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 können. 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 nun davon aus, dass die Queue 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.) Knacknuss: 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 dann passieren?). Kleiner Tipp: Es wäre grundsätzlich nützlich, immer zu wissen wie viele Elemente die Queue gerade enthält!

Implementation, immer schrittweise getestet:

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

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 mit Hilfe von Log4J sehr einfach debuggen, in dem Sie am Ende jeder Operation den aktuellen Zustand ausgeben!

g.) Implementieren Sie die Methoden schrittweise und testen Sie fortlaufend bis alles korrekt funktioniert. 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?
Überraschung: Es gibt tatsächlich Varianten, welche bei einer vollen Queue hemmungslos enthaltene Daten überschreiben. Das wollen wir in unserer Lösung aber definitiv nicht!

5 Optional: Speicherverwaltung mit eigener Liste

5.1 Lernziel

- Einsatz der selber implementierten Listen-Datenstruktur.

5.2 Grundlagen

Diese Aufgabe basiert auf dem Input D11. Grundlage bildet die Lösung der Aufgabe 2 und Vorarbeiten aus der Übung E0 von SW01.

5.3 Aufgaben

a.) Im Input D11 haben wir uns kurz über Funktionsweise von Speicherallokationen unterhalten. Ein erster Lösungsansatz wären zwei Listen: Je eine für den freien, und den belegten Speicher. Beginnen Sie mit der Implementation und verwenden Sie dazu Ihre eigene Liste von der Aufgabe 2!

b.) Mitunter werden Sie feststellen, dass daraus neue Anforderungen an die Liste gestellt werden. Erweitern Sie Ihre Liste!