Funktionale und objektorientierte Programmierkonzepte

Florian Kadner und Lukas Röhrig
(Gesamtleitung: Prof. Karsten Weihe)

Wintersemester 18/19
v1.0

Übungsblatt 12

Themen: Wrap-Up II

Relevante Folien: alle bis dahin

Abgabe der Hausübung: 08.02.2019 bis 23:55 Uhr

V Vorbereitende Übungen

V1 Schleifenerkennung in verzeigerten Strukturen

Für diese Aufgabe betrachten wir folgende Klasse für Listenelemente, die Sie auch schon in der Vorlesung und auf Übungsblatt 9 kennengelernt haben.

```
public class ListItem<T>{
  public T key;
  public ListItem<T> next;
}
```

In einer solchen Liste kann es theoretisch vorkommen, dass eine Schleife vorkommt. Damit ist gemeint, dass ein Element der Liste auf ein früheres zurückverweist. Somit kommt man beim linearen Durchlaufen der Liste in eine Schleife. Ein Beispiel dafür finden Sie in 1.

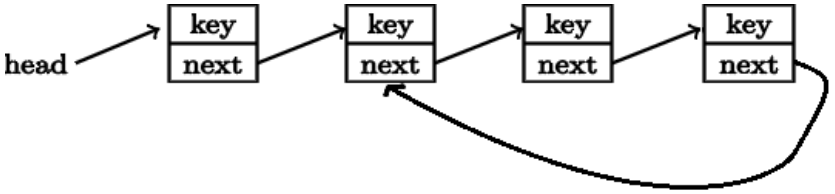

Abbildung 1: Eigene LinkedList-Klasse aus der Vorlesung mit Schleife.

In dieser Aufgabe soll es darum gehen verschiedene Ansätze kennenzulernen und zu vergleichen, mit denen solche Schleifen in linearen Listen entdeckt werden können.

Setzen Sie alle nachfolgend vorgestellten Ansätze in Java um! Diskutieren Sie anschließend die Vor- und Nachteile jedes Ansatzes.

Funktioniert überhaupt jeder der Ansätze? Wo kann es zu Problemen kommen?

V1.1 Jedes Element merken

Die erste Möglichkeit besteht darin jedes einzelne Element, welches bereits durchlaufen wurde, abzuspeichern. Bei jedem neuen Schritt wird dann geschaut, ob das Element bereits abgespeichert wurde. Zum Abspeichern kann in Java beispielsweise ein `HashSet` verwendet werden.

V1.2 Doppelte Verkettung nutzen

Sie können die Klasse für Listenelemente dahingehend erweitern, dass es nun zusätzlich einen Zeiger auf das vorherige Element gibt. Beim Durchlauf der Liste muss nun jedes Element das zuvor durchlaufende Element referenzieren, ansonsten liegt eine Schleife vor.

V1.3 Vergleich mit Startelement

Der Zeiger auf das nächste Element wird mit dem Startelement der Liste verglichen. Sind diese beiden Elemente gleich, so liegt eine Schleife vor.

V1.4 Jedes Element markieren

Die Klasse für Listenelemente soll um ein Attribut `public boolean visited` erweitert werden. Dieses ist initial auf `false` gesetzt und wird beim Durchlaufen auf die bereits besuchten Elemente auf `true` gesetzt. Kommen wir bei einem Element vorbei, welches bereits auf `true` gesetzt ist, muss eine Schleife vorliegen.

V1.5 Liste sortieren

Zunächst wird die Liste mittels Comparator aufsteigend sortiert. Anschließend wird beim Durchlauf jedes Element mit dem nächsten verglichen. Wenn der Zeiger auf ein kleineres Element verweist, muss eine Schleife vorliegen.

V1.6 Hase-Igel-Algorithmus

Der Algorithmus besteht aus dem gleichzeitigen Durchlauf der Liste mit unterschiedlichen Schrittweiten. Dabei werden zwei Zeiger auf Listenelemente benutzt, von denen der eine (Igel) bei jeder Iteration auf das nächste Element verschoben wird, während der andere (Hase) bei derselben Iteration auf das übernächste Element verschoben wird. Wenn die beiden Zeiger sich begegnen, also dasselbe Element referenzieren, hat die Liste eine Schleife. Wenn einer der beiden Zeiger das Ende der Liste erreicht, hat sie keine Schleife.

Implementieren Sie die Methode `boolean tortoiseAndHare(ListItem<T> head)`, welche den Kopf einer Liste übergeben bekommt und `true` genau dann zurückliefert, wenn die Liste einen Zyklus enthält.

V2 Objektorientierte Modellierung

In dieser Aufgabe geht es um das Athene-Bistro im Robert-Piloty-Gebäude. Das Bistro bietet verschiedene Speisen und Getränke an, welche Sie in der folgenden Aufgabe sinnvoll gemäß des objektorientierten Programmierkonzeptes modellieren sollen. Erstellen Sie eine Typhierarchie (vergleichen Sie dazu nochmal die Aufgaben V3, V5 und V6 von Blatt 8) welche mindestens die folgenden konkreten Elemente umfasst:

- die Getränketypen Softdrink (hier gibt es Cola und Fanta) und Bier
- die Snacks Schokoriegel (hier gibt es Snickers, Mars und Kinderschokolade), Chips und Eis
- die Speisen Bockwurst, Sandwich und Schnitzelbrötchen

Beachten Sie zur Modellierung folgende Hinweise:

- Auf Methoden und Attribute dürfen Sie in dieser Aufgabe verzichten.
- Fügen Sie sinnvolle und möglichst passend benannte Obertypen ein.
- Markieren Sie abstrakte Klassen und Interfaces eindeutig!
- Es gibt bei dieser Aufgabe keine „richtige“ Lösung. Begründen Sie, wieso Sie sich für genau diese Modellierung entschieden haben und was man eventuell anders hätte gestalten können.

V3 Absteigend sortiert?

Ergänzen Sie die folgende Methode und ihre Paramterliste:

```
1 /**
2  * Checks whether the given list is sorted in descending
      order.
3  *
4  * @param alox the list which has to be checked.
5  * @return true, if the list is sorted in descending order.
6  */
7 public boolean isSortedDescending(final List<T> alox) {
8
9 }
```

Ihre Methode muss mit generischen Listen umgehen können. Wenn die übergebene Liste leer ist, so ist `true` zurückzuliefern. Sie dürfen weiterhin davon ausgehen, dass die Liste nur Elemente enthält, welche nicht `null` sind.

V4 Map in Java

In Racket haben Sie zu Beginn des Semesters eine Funktion höherer Ordnung namens `map` kennengelernt. Außerdem haben Sie in der Vorlesung zunächst eine eigene Implementierung dieser Funktion gesehen. Zur Erinnerung:

```
;; Type: (X -> Y) (list of X) -> (list of Y)
(define (my-map fct list)
     (cond
           [(empty? list) empty]
           [else (cons (fct (first list))
                       (my-map fct (rest list)))]))
```

Auch in Java können wir mittels Streams ganz einfach auf die `map` Methode zugreifen. In dieser Aufgabe sollen Sie jedoch die Funktionalität selbst implementieren. Ergänzen Sie dazu den folgenden Code:

```
/**
 * Applies a method to every element of a list.
 * @param fct function to apply
 * @param list list to iterate over
 * @return [fct(list[0]), fct(list[1]), ..., fct(list[n-1])]
 */
public <T,R> List<R> map(Function<T,R> fct, List<T> list) {

}
```

Die Funktion `fct` realisieren Sie hierbei später über einen Lambda-Ausdruck.

Folgender Aufruf in Racket...

```
(define number-list (list 3 6 9))
(my-map (lambda (x) (* x 10)) number-list)
```

... wird dann später in Java folgendermaßen umgesetzt:

```
Function<Integer,Integer> multiply10 = new Function<>() {
    public String apply(Integer i) { return i*10; }
};

map(i -> i.multiply10(),
    Arrays.asList(new Integer[] {3, 6, 9}));
```

V5 Generics

Gegeben seien kurze Codeauszüge. Geben sie bei jedem der Auszüge (ohne Hilfe des Compilers!) an, ob der Code kompiliert. Wenn nicht, dann begründen Sie kurz wieso.

V5.1

```
public final class bar {
    public static <T> T oracle(T x, T y) {
        return x > y ? x : y;
    }
}
```

V5.2

```
public static void print(List<? extends X> alox) {
  for (X x : alox)
    System.out.print(x + " ");
}
```

V5.3

```
public class Y<T> {

  private static T y = null;

  public static T foo() {
    if (y == null)
      y = new Y<T>();

    return y;
  }
}
```

V5.4

```
class A {}
class B extends A {}
class C extends A {}
class D<T> {}

D<B> db = new D<>();
D<A>  da = dc;
```

H Zwölfte Hausübung Gesamt 12 Punkte

Graphen mit Adjazenzmatrizen

In dieser Hausübung beschäftigen wir uns erneut mit dem Thema Graphen, welche Ihnen bereits aus der Hausübung des ersten Wrap-Up Blatts bekannt sind. Als Graph bezeichnet man in der Informatik eine Datenstruktur, die eine Menge von Objekten, sowie eine Menge von Verbindungen zwischen diesen, speichern kann. Die Objekte eines Graphen werden Knoten genannt, die paarweisen Verbindungen zwischen den Knoten nennt man Kanten.

Um abzuspeichern, welche Knoten miteinander verbunden sind, wollen wir eine sogenannte Adjazenzmatrix (oder auch Nachbarschaftsmatrix) verwenden. Die Adjazenzmatrix eines Graphen ist eine Matrix, die speichert, welche Knoten des Graphen durch eine Kante verbunden sind. Sie besitzt für jeden Knoten eine Zeile und eine Spalte, woraus sich für n Knoten eine Matrix der Größe $n \times n$ ergibt. Ein Eintrag in der i-ten Zeile und j-ten Spalte gibt hierbei an, ob eine Kante vom i-ten zum j-ten Knoten führt. Steht an dieser Stelle NaN (Not a Number), ist keine Kante vorhanden – eine Zahl gibt an, dass eine Kante existiert und repräsentiert gleichzeitig das sogenannte Kantengewicht. In den folgenden Abbildungen sehen Sie einen Beispielgraphen, sowie seine zugehörige Adjazenzmatrix.

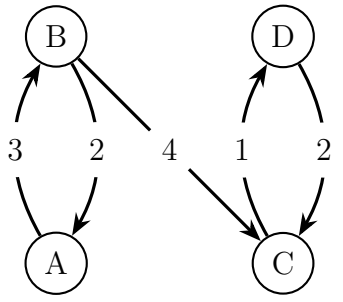

Abbildung 2: Ein Beispielgraph mit vier Knoten.

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} NaN & 3 & NaN & NaN \\ 2 & NaN & 4 & NaN \\ NaN & NaN & NaN & 1 \\ NaN & NaN & 2 & NaN \end{pmatrix} \end{array}$$

Abbildung 3: Adjazenzmatrix

Ihr Ziel in dieser Aufgabe ist es, eine Graphenklasse in Java von Grund auf zu implementieren. Um die Klasse möglichst vielfältig einsetzen zu können, werden Sie die Klasse sinnvoll generisch parametrisieren müssen. Wir wollen uns nicht auf einen konkreten Typ für die Knoten eines Graphen festlegen, die Klasse soll mit jedem beliebigen Typ umgehen können. Sie können dabei frei entscheiden, wie Sie die Knoten in der Klasse abspeichern. Die Kantengewichte eines Graphen wollen wir ebenfalls nicht auf einen konkreten Typ festlegen, jeder Typ, der das `Comparable`-Interface implementiert, ist zulässig. Zusätzlich soll es in unserer Graphen-Klasse möglich sein, dass ein Graph bei Bedarf mehrere Adjazenzmatrizen und somit auch mehrere Kantenmengen besitzt. Folgend ein kurzes Beispiel dafür: Wir betrachten einen Graph G, der zur Berechnung von Navigationsrouten verwendet werden soll. G besitzt zwei Adjazenzmatrizen A_{time} und A_{distance} und dadurch ist es uns möglich die kürzeste Strecke, gemessen in Zeit über die Matrix A_{time}, oder gemessen am Weg über die Matrix A_{distance} zu berechnen. Die Adjazenzmatrizen eines Graphen werden über einen eindeutigen Bezeichner von Typ `String` identifiziert, in unserem Beispiel `"time"` und `"distance"`. Ein Objekt der Graphenklasse darf nicht mehrere Adjazenzmatrizen mit dem gleichen Bezeichner besitzen. Überlegen sie sich, wie Sie mehrere Matrizen speichern können und wie Sie diese Matrizen über einen Bezeichner von Typ `String` identifizieren können.

H1 Modellierung 2 Punkte

Erstellen Sie nun zunächst eine Klasse `Graph` im Package `Main`. Modifizieren Sie die Klasse dahingehend, dass die aus dem obigen Einleitungstext genannten Anforderungen erfüllt werden können. Zusätzlich fügen Sie der Klasse einen Konstruktor hinzu. Dieser bekommt als einzigen Parameter das Kantengewicht übergeben, dass für eine nicht-existierende Kante steht. In dem Beispielgraph aus Abbildung 2 wäre dies NaN. Nach dem Aufruf des Konstruktors ist die Anzahl an Kanten und Knoten gleich null.

H2 Methode addNode 2 Punkte

Um unseren Graph befüllen zu können, schreiben Sie nun eine Methode `public void addNode`. Die Methode bekommt einen Knoten übergeben und fügt diesen dem Graphen hinzu. Passen Sie alle Adjazenzmatrizen diesbezüglich an. Alle Kantengewichte werden mit dem im Konstruktor übergebenen Parameter initialisiert. Dem ersten Knoten der dem Graphen hinzugefügt wurde, wird Reihe 0 und Spalte 0 in der Adjazenzmatrix zugewiesen, dem zweiten Knoten wird Reihe 1 und Spalte 1 zugewiesen usw.

H3 Methode setWeight 2 Punkte

Nachdem bereits das Einfügen neuer Knoten in den Graphen implementiert wurde, kümmern wir uns nun um die Kantensetzung zwischen diesen. Dazu schreiben Sie eine Methode `public void setWeight`. Die Methode bekommt als ersten Parameter den Bezeichner einer Adjazenzmatrix des Graphen als `String` übergeben. Als zweiten Parameter bekommt sie die Reihe der Matrix und als dritten Parameter die Spalte der Matrix jeweils als `int` übergeben. Der vierte Parameter ist das Kantengewicht, dass der Kante in der übergebenen Reihe und Spalte zugewiesen werden soll. Besitzt der Graph noch keine Adjazenzmatrix zum übergebenen Bezeichner, so ist eine neue Adjazenzmatrix für diesen Bezeichner einzurichten um dann das Kantengewicht zu setzen. Alle anderen Kantengewichte, der neuen Matrix, werden mit dem im Konstruktor übergebenen Parameter initialisiert. Sie dürfen davon ausgehen, dass die übergebene Reihe und Spalte kleiner ist als die aktuelle Anzahl an Knoten und müssen diesen Fall nicht gesondert behandeln.

H4 Methode getWeight und getNumberOfNodes 1 Punkt

Implementieren Sie eine `public`-Methode `getWeight`. Die Methode bekommt als ersten Parameter den Bezeichner einer Adjazenzmatrix des Graphen als `String` übergeben. Als zweiten Parameter bekommt sie die Reihe der Matrix als `int` übergeben. Als dritten Parameter die Spalte der Matrix als `int` übergeben. Zurückgegeben wird das Kantengewicht, dass sich an der übergebenen Reihe und Spalte in der Matrix mit dem übergebenen Bezeichner befindet. Sie dürfen davon ausgehen, dass eine Matrix mit dem übergebenen Bezeichner im Graphen existiert und dass die übergebene Reihe und Spalte kleiner ist als die aktuelle Anzahl an Knoten. Sie müssen diesen Fälle nicht gesondert behandeln.

Implementieren Sie eine Methode `public int getNumberOfNodes`. Die Methode hat keine Parameter und soll lediglich die aktuelle Anzahl an Knoten im Graph zurückgeben.

H5 Methode getAllPaths — 3 Punkte

Implementieren Sie nun eine Methode `public String[] getAllPaths`. Die Methode bekommt als ersten Parameter den Bezeichner einer Adjazenzmatrix des Graphen als `String` übergeben. Als zweiten Parameter bekommt sie den Index eines Knotens im Graphen als `int` übergeben. Der Index meint hier, welche Reihe und Spalte in der Adjazenzmatrix dem Knoten zugeordnet wird. Als dritten und letzten Parameter bekommt die Methode einen `boolean` übergeben, später dazu mehr. Die Methode soll ausgehend von dem übergebenen Startknoten alle möglichen Pfade finden, in denen kein Knoten mehrmals vorkommt. Dabei ist zu beachten, dass keiner der gefundenen Pfade ein Teilpfad eines anderen gefundenen Pfades ist. Zurückgegeben werden soll ein Array von `Strings`. Jeder String dieses Arrays steht dabei für einen Pfad. Die Reihenfolge in der die Pfade im Array vorkommen kann beliebig sein. Sie dürfen davon ausgehen, dass eine Matrix mit dem übergebenen Bezeichner im Graphen existiert und müssen diesen Fall nicht gesondert behandeln.

Folgend ein Beispiel: Wir gehen davon aus, dass wir ein Graphen-Objekt mit den Knoten und Kanten aus Abbildung 2 eingerichtet haben, die dazugehörige Adjazentmatrix wird mit dem `String "example"` identifiziert. Nun betrachten wir einen Aufruf der Methode `getAllPaths` dieses Objekts. Wir übergeben `"example"` als Bezeichner einer Adjazentmatrix und `1` als den Index eines Knotens (Knoten B aus Abbildung 2), den dritten Parameter betrachten wir zunächst nicht. Der Methodenaufruf liefert den Pfad von B nach A und den Pfad von B nach C nach D zurück.

Die Pfade als `String` sollen nach folgendem Schema formatiert werden: `"N1 -> N2 -> ... -> Nk"`. Die Kanten eines Pfades werden durch den String `" -> "` (Leerzeichen, Bindestrich, geschlossene spitze Klammer, Leerzeichen) angegeben. Die Knoten werden entweder durch ihren zugehörigen Index in der Adjazenzmatrix oder durch die `toString()`-Methode der Knoten angegeben. Ist der dritte Parameter der Methode `true`, so werden die Indizes der Knoten im Pfad-String angegeben, andernfalls wird die `toString()`-Methode der Knoten im Pfad-String angegeben.

Gehen wir nun davon aus, dass in unserem obigen Beispiel die Knoten des Graphens von Typ `String` sind, dann sieht unser zurückgegebenes Array folgendermaßen aus:

Wenn der dritter Parameter `== false` (Knoten dargestellt mittels `toString()`-Methode):

`{"B -> A", "B -> C -> D"}`.

Wenn der dritter Parameter `== true` (Knoten als Index):

`{"1 -> 0", "1 -> 2 -> 3"}`.

Der Pfad `"1 -> 2"` oder eben `"B -> C"`, darf im zurückgegebenen Array nicht vorhanden sein, da er ein Teilpfad von `"1 -> 2 -> 3"` oder eben `"B -> C -> D"` ist.

H6 Testen 2 Punkte

Abschließend wollen wir unsere implementierte Klasse testen. Dazu erstellen Sie eine neue Klasse `GraphTests` im Package `Main`. Implementieren Sie nun jeweils mindestens 3 Tests für die Methoden `addNode`, `setWeight` sowie `getAllPaths`.