Aufgabenstellungen und Übungsbeispiele
ALD-Übung – Sommersemester 2019

[A01] Generische Stack auf Basis von verketteten Listen

Implementieren Sie einen generischen Stack auf Basis von verketteten Listen. Eine Basisstruktur finden Sie im Projekt.

[A02] Generische Queue auf Basis von verketteten Listen

Implementieren Sie eine generische Queue auf Basis von verketteten Listen. Eine Basisstruktur finden Sie im Projekt.

[A03] Doppelt Verkettete Liste

Implementieren Sie eine verkettete Liste, bei welcher jeder Knoten, auf den nächsten Eintrag und auf seinen Vorgänger verweist.

Starten Sie ausgehend von der einfach verketteten Liste und erweitern Sie diese schrittweise.

```
public class Node
{
    private final Ausrede ausrede;

    private Node next;

    private Node previous;

    […]
}
```

Weitere Informationen:
http://de.wikipedia.org/wiki/Liste_(Datenstruktur)#Doppelt_.28mehrfach.29_verkettete_Liste

[A04] Baum traversieren

Implementieren Sie die zwei nachfolgenden Methoden für das Wörterbuch. Die dazu benötigten Klassen finden Sie im Projekt. Dort finden Sie auch JUnit-Tests, damit Sie Ihre Implementierung überprüfen können.

- public int countWordsInSubTree(Wort w)
- public Set<String> getWordsWithPrefix(String prefix)

[A05] Breitensuche

Implementieren Sie die zwei nachfolgenden Methoden, welche einen Baum in Breitenordnung traversieren. Die dazu benötigten Klassen finden Sie im Projekt. Dort finden Sie auch JUnit-Tests, damit Sie Ihre Implementierung überprüfen können.

- public List<Integer> getBreadthFirstOrder(Node<Integer> start)
- public List<Integer> getBreadthFirstOrderForLevel(Node<Integer> start, int level)

Tipp: bei der zweiten Funktion müssen Sie sich in geeigneter Weise für jeden Knoten die Ebene merken. Das können Sie auf zwei Arten tun: entweder die Klasse Node erweitern oder im Algorithmus eine eigene Wrapper-Klasse haben, die eine normale Node mit einem Level zusammen speichert.

[A06] Tiefensuche

Im Package A06_Tiefensuche finden Sie die notwendigen Basisklassen, um eine Tiefensuche in einem Baum zu implementieren. Auch passende JUnit-Tests, um Ihre Implementierung zu überprüfen, sind dort vorhanden.

Eingefügt in den Baum werden Film-Objekte, als Sortierkriterium soll deren Länge dienen. Sie müssen also die compare()-Methode entsprechend implementieren.

- public List<String> getNodesInOrder(Node<Film> node)
- public List<String> getMinMaxPreOrder(double min, double max)

[A07] BubbleSort

Im Package *A07_BubbleSort* finden Sie die Klasse *Person*. Erweitern Sie die Klasse um die Funktion

```
public int compareTo(Person p)
```

Die Funktion soll anhand des Nachnamens, bei gleichen Nachnamen anhand des Vornamens, eine alphabetische Ordnung der beiden Objekte ermöglichen.

Ähnlich der *compareTo()*-Funktion der *String*-Klasse soll eine negative Zahl retourniert werden, wenn die übergebene Person (*Nachname+Vorname*) später im Alphabet kommt, eine positive Zahl, wenn die übergebene Person früher im Alphabet kommt und Null, wenn Nachname und Vorname identisch sind.

Aufbauend auf dieser Klasse implementieren Sie im Package die Klasse *BubbleSort* die Methode

- public void sort(Person[] personen)

Im Package finden Sie auch passende JUnit-Tests. Diese leiten sich von der Klasse *PersonenSortTest* ab. Für die ausgeführten Tests müssen Sie also in dieser Klasse nachsehen.

[A08] Tiefensuche: Zusammenhängender Graph

Die Tiefensuche (depth-first search) ist auch ein für Bäume und Graphen gebräuchlicher Algorithmus. Während bei der Breitensuche ausgehend von einem Startknoten alle über die Kanten erreichbaren Knoten in eine Queue eingefügt und der Reihe nach abgearbeitet werden, kann die Tiefensuche entweder rekursiv oder mit Hilfe eines Stacks (anstatt der Queue) erfolgen.

Die Tiefensuche hat für Graphen gleich wie die Breitensuche eine Laufzeit von **O(V+E)**, da für einen vollständigen Durchlauf im schlechtesten Fall alle Knoten und Kanten je einmal besucht bzw. überprüft werden müssen.

Schreiben Sie einen Algorithmus, der mit Hilfe der Tiefensuche feststellt, ob ein Graph zusammenhängend ist bzw. aus wie vielen *zusammenhängenden Komponenten* ein Graph besteht. Dort finden Sie auch JUnit-Tests, mit denen Sie Ihre Lösung überprüfen können.

Die Methode `getNumberOfComponents(Graph g)` soll die Anzahl der zusammenhängenden Komponenten eines Graphs retournieren. Ist der Graph vollständig zusammenhängend, liefert die Funktion den Wert **1**.

Möglicher Algorithmus

1. Starten Sie die Tiefensuche an einem beliebigen Knoten
2. Markieren Sie im Zuge der Tiefensuche alle besuchten Knoten
3. Überprüfen Sie mit einer gesonderten Schleife, ob alle Knoten besucht worden sind
    - Wenn ja: fertig.
    - Wenn nein: Graph ist nicht zusammenhängend, neue Zusammenhangskomponente gefunden. Neue Tiefensuche bei einem unbesuchten Knoten starten. Weiter mit Schritt 2.
4. Retournieren der Anzahl der gefundenen Komponenten

Welche Laufzeit hat dieser Algorithmus?

[A09] Tiefensuche: Zyklen in Graphen

Ein Graph, gerichtet oder ungerichtet, enthält einen **Zyklus**, wenn ausgehend von einem Knoten ein Weg über vorhandene Kanten und Knoten zurück zum Ursprungsknoten existiert.

Schreiben Sie einen Algorithmus, der mit Hilfe der Tiefensuche feststellt, ob ein Graph zumindest einen Zyklus enthält und diesen Zyklus ausgibt (falls vorhanden). Der Algorithmus muss für gerichtete und ungerichtete Graphen funktionieren.

Die Methode `List<Integer> getCycle()` soll die Knoten eines (beliebigen) gefundenen Zyklus als Liste von Integer-Zahlen ausgeben. In der Liste müssen erster und letzter Knoten (Integer-Zahl) identisch sein (d.h. Start- bzw. Endpunkt des Zyklus kommt doppelt in der Liste vor). Enthält der Graph keinen Zyklus, soll `null` retourniert werden.

Möglicher Algorithmus

1. Starten Sie die Tiefensuche an einem beliebigen Knoten.
2. Markieren Sie im Zuge der Suche alle besuchten Knoten und speichern die Vorgänger(*) der Knoten in einer eigenen Datenstruktur ab.
3. Wenn Sie auf einen bereits besuchten Knoten stoßen: Zyklus gefunden, an Hand der Vorgänger die Liste der Knoten im Zyklus erstellen und retournieren.

4. Falls noch nicht alle Knoten besucht worden sind (unzusammenhängender Graph): Neue Tiefen- oder Breitensuche bei einem unbesuchten Knoten starten. Weiter mit Schritt 2.

(*) Für eine leichtere Konstruktion können sie auch alternativ mit den Nachfolgern anstatt den Vorgängern arbeiten.

Welche Laufzeit hat dieser Algorithmus?

[A10] DijkstraPQShortestPath

Im Package A10_DijkstraPQShortestPath finden Sie einige Klassen zu Graphen. Nehmen Sie sich die Zeit, diese Klassen durchzusehen, da sie als Grundgerüst auch bei der Klausur verwendet werden.

- **Hilfsklassen**
  - `Main`: Testklasse, die zum Aufrufen der anderen Klassen dient. Enthält die *main()*-Funktion zum Programmstart.
- **Klassen zum Speichern von Graphen**
  - `Graph`: Interface, das Funktionen zum Hinzufügen, Suchen und Entfernen von Kanten definiert. Die Parameter *u* und *v* der Funktionen sind die Nummern der Knoten (*u*: Start, *v*: Ziel). Achtung: In der Implementierung werden Knoten in aller Regel nur durch Integer-Zahlen repräsentiert, nicht durch eine eigene Klasse. Für Kanten gibt es die Klasse *WeightedEdge*.
  - `ListGraph`: Implementierung eines Graphen als ein Array von Listen (siehe Folien).
  - `ArrayGraph`: Implementierung eines Graphen als zweidimensionales Array (siehe Folien).
  - `WeightedEdge`: gewichtete Kante zwischen zwei Knoten.
- **Algorithmen**
  - `FindWay`: Abstrakte Basisklasse für alle Algorithmen, welche unter anderem bereits eine Hilfsfunktion enthält, um aus dem Vorgänger-Array eine Weg-Liste zu machen.
  - `DijkstraPQShortestPath`: Dijkstra-Algorithmus für lichte Graphen mit Priority-Queue (= Heap). Findet den kürzesten Weg zwischen zwei Punkten.
    - `Vertex`: Klasse für Knoten, die im Heap gespeichert werden. Neben der Knotennummer enthalten sie auch die aktuelle Priorität (bei Dijkstra: berechnete Entfernung des Knotens vom Startknoten; Variable *cost*)
    - `VertexHeap`: Heap für Knoten, der im Dijkstra-Algorithmus verwendet wird. Dijkstra verwendet vor allem die Funktion *setCost()*.

Implementieren Sie den Dijkstra-Algorithmus mit Heap (*DijkstraPQShortestPath*) anhand der Folien nach. Setzen Sie die beiden Arrays *pred[]* (Vorgänger) und *dist[]* (kumulierte Entfernung) entsprechend und verwenden Sie für den Heap die Klasse *VertexHeap*.

Notwendige Anpassungen sind mit TODO markiert.

[A11] DijkstraDGShortestPath

Analysieren Sie die fertige Lösung für den Dijkstra-Algorithmus ohne Heap (*DijkstraDGShortestPath*).

Erweitern Sie die Klasse *WeightedEdge*, welche eine Kante repräsentiert, um eine Eigenschaft *charge*, welche darüber Auskunft gibt, ob die jeweilige Kante eine Mautstraße repräsentiert (*true*: Mautstraße; *false*: keine Mautstraße). Der Graph hat auch eine zusätzliche Überladung von *addEdge*, welche den Wert für *charge* als letzten Parameter entgegennimmt.

Ändern Sie den Algorithmus von Dijkstra, und erweitern Sie die Methode *findWay* um einen Parameter (useChargeRoads). Übergibt der Aufrufer für den Parameter *useChargeRoads* den Wert *true*, so sollen sämtliche Straßen für Lösung in Betracht gezogen werden. Wird für *useChargeRoads* der Wert *false* übergeben, sollen nur Straßen in Betracht werden, die keine Mautstraßen sind.

[A12] DijkstraLand

Im Package A12_DijkstraLand finden Sie eine Klasse Dijkstra. Implementieren Sie innerhalb der vorgegebenen Methode dijkstra den Algorithmus von Dijkstra zum Finden des kürzesten Pfades zwischen den übergebenen Knoten von und nach.

Betrachten Sie den Graphen als **Bahnnetz**, bei dem jeder Knoten einen Bahnhof darstellt. Pro Knoten findet man in der verwendeten ListGraph-Klasse einen String, der Auskunft über das Land, indem sich der Bahnhof befindet, gibt (z. B. A, DE, CH, IT, H, SLO). Dieses Land kann beim Graph über die Methode ***getLand*** abgerufen sowie über ***setLand*** gesetzt werden. Erstere Methode nimmt die Nummer des gewünschten Knotens entgegen. Letztere neben dieser Nummer auch das zu hinterlegende Land.

Gehen Sie davon aus, die Verwendung einer **grenzüberschreitenden Kante** (= Kante, die Bahnhöfe zweier Länder verbindet) mit einem zusätzlichen Aufwand einhergeht und berücksichtigen Sie somit bei solchen Kanten ein **zusätzliches Gewicht von 1** bei der Berechnung der Entfernung.

Als Ergebnis soll eine Liste, welche den gefundenen Pfad beginnend beschreibt, retourniert werden. Die Reihenfolge der Einträge in dieser Liste soll jener Reihenfolge entsprechen, in der beginnend bei „von“ die einzelnen Knoten passiert werden. Für den Pfad 0 → 5 → 9 → 7 ist somit eine Liste mit dem folgenden Aufbau zurückzuliefern: [0, 5, 9, 7].

[A13] MergeSort

Im Package *A13_MergeSort* finden Sie die Klasse *Person*. Erweitern Sie die Klasse um die Funktion

```
public int compareTo(Person p)
```

Die Funktion soll anhand des Nachnamens, bei gleichen Nachnamen anhand des Vornamens, eine alphabetische Ordnung der beiden Objekte ermöglichen.

Ähnlich der *compareTo()*-Funktion der *String*-Klasse soll eine negative Zahl retourniert werden, wenn die übergebene Person (*Nachname+Vorname*) später im Alphabet kommt, eine positive Zahl, wenn die übergebene Person früher im Alphabet kommt und Null, wenn Nachname und Vorname identisch sind.

Aufbauend auf dieser Klasse implementieren Sie im Package die Klasse *MergeSort* die Methode

- public void sort(Person[] personen)

Im Package finden Sie auch passende JUnit-Tests. Diese leiten sich von der Klasse *PersonenSortTest* ab. Für die ausgeführten Tests müssen Sie also in dieser Klasse nachsehen.