Lehrstuhl für Informatik I
Algorithmen, Komplexität und
wissensbasierte Systeme
Universität Würzburg

Würzburg, den 9. Januar 2018

Prof. Dr. Alexander Wolff
Fabian Lipp, M. Sc.
Andre Löffler, M. Sc.
Johannes Blum, B. Sc.

7. Übungsblatt zur Vorlesung Algorithmen und Datenstrukturen (Winter 2017/18)

PABS **Aufgabe 1 – Gerichteten Graphen implementieren**

Implementieren Sie einen gerichteten Graphen in Java. Jeder Knoten des Graphen soll einen Verweis auf ein Objekt der Klasse Object speichern. Die gerichteten Kanten des Graphen sollen durch Adjazenzlisten repräsentiert werden. Speichern Sie die Knotenmenge des Graphen als Liste.

Verwenden Sie als Implementierung für die Listen die Klassen die vorgegebenen Klassen List und ListItem. Schreiben Sie die Klassen derart um, dass sie nun Objekte vom Typ DiGraphNode verwalten (anstatt wie bisher vom Typ Double). Verwenden Sie keine Datenstrukturen und Algorithmen der Java API oder zusätzlicher Bibliotheken, die Sie nicht selbst programmiert haben.

Hinweis: Bei dieser Aufgabe sind die PABS-Tests optional. Sie brauchen nicht alle Tests zu erfüllen, um Ihren Code als Lösung abgeben zu können, sofern ihr Code zumindest erfolgreich kompiliert. Entsprechend gibt es Teilpunkte für unvollständige oder fehlerhafte Lösungen.

a) Implementieren Sie im Paket diGraph die Klasse public class DiGraphNode . Diese soll einen Konstruktor

   - `public DiGraphNode(Object key)`

   enthalten, der einen neuen Knoten mit einem Verweis auf das Objekt key der Klasse Object erzeugt. Ferner soll die Klasse DiGraphNode über folgende Methoden verfügen:

   - public List getNeighbors() – gibt die Adjazenzliste des Knotens zurück
   - public Object getKey() – gibt das Objekt zurück, auf das der Knoten verweist

   Implementieren Sie außerdem ebenfalls im Paket diGraph die Klasse public class DiGraph. Diese Klasse soll einen Konstruktor

   - public DiGraph()

   enthalten, welcher einen Graphen mit einer leeren Knotenliste erzeugt. Mit zusätzlich soll die Klasse über folgende Methoden verfügen:

- public DiGraphNode find(Object key) – überprüft, ob der Graph einen Knoten v mit key.equals(v.getKey()) enthält und gibt entweder v oder Null zurück.
- public DiGraphNode addNode(Object key) – erzeugt einen neuen Knoten v mit Verweis auf key, fügt diesen dem Graphen hinzu und gibt dann v zurück
- public void addEdge(DiGraphNode v1, DiGraphNode v2) – erzeugt eine gerichtete Kante vom Knoten v1 zum Knoten v2 **6 Punkte**

b) Implementieren Sie in der Klasse DiGraph den Konstruktor

- public DiGraph(Object[ ] keys, boolean[ ][ ] adjacencyMatrix)

Dieser Konstruktor soll einen Graphen erzeugen, der für jedes Objekt im Feld keys einen Knoten enthält. Vom Knoten fürs Objekt keys[i] zum Knoten fürs Objekt keys[j] soll es genau dann eine gerichtete Kante im Graphen geben, wenn adjacencyMatrix[i][j] den Wert true hat. **4 Punkte**

Aufgabe 2 – Zweifärbbarkeit

Ein Graph $G = (V, E)$ heißt *zweifärbbar*, wenn eine Abbildung $c\colon V \to \{\text{rot}, \text{blau}\}$ existiert, so dass für jede Kante $\{u, v\} \in E$ gilt, dass $c(u) \neq c(v)$. Zwei zueinander benachbarte Knoten erhalten also stets unterschiedliche Farben.

a) Welches ist der kleinste Graph, der nicht zweifärbbar ist? **1 Punkt**

b) Entwerfen Sie einen Algorithmus in Pseudocode, der für einen gegebenen Graphen $G = (V, E)$ ermittelt, ob er zweifärbbar ist. Die Laufzeit des Algorithmus soll $O(|V| + |E|)$ sein. Achtung: G ist nicht notwendigerweise zusammenhängend. **3 Punkte**

Aufgabe 3 – Schlange aus zwei Stapeln analysieren

Zwei Stapel S_1, S_2 können wie folgt eine Schlange Q simulieren:

- Enqueue(x): führe S_1.Push(x) aus – lege also x auf den ersten Stapel.
- Dequeue(): Ist S_2 leer, führe solange S_2.Push(S_1.Pop()) aus, bis S_1 leer ist – verschiebe die Elemente der Reihe nach von S_1 auf S_2. Danach führe S_2.Pop() aus.
- Empty(): gibt true zurück, falls S_1 und S_2 leer sind, sonst false.

Betrachten Sie nun eine zufällige Folge der Länge n bestehend aus Enqueue-, Dequeue- und Empty-Operationen auf Q.

a) Argumentieren Sie, warum die *Worst-Case*-Laufzeit einer einzelnen Dequeue-Operation auf Q in $\Theta(n)$ liegt. **3 Punkte**

b) Zeigen Sie mit amortisierter Analyse: die Gesamtlaufzeit für jede Folge liegt ebenfalls in $\Theta(n)$. Eine einzelne Dequeue-Operation benötigt amortisiert also nur konstante Laufzeit. Wieso ist dies kein Widerspruch zu Teilaufgabe a)? **3 Punkte**

Bitte werfen Sie Ihre Lösungen bis **Dienstag, 16. Januar 2018, 10:10 Uhr** in den Vorlesungs-Briefkasten im Informatik-Gebäude. Geben Sie stets die Namen und Übungsgruppen aller BearbeiterInnen sowie die Übungsgruppe, in der das Blatt zurückgegeben werden soll, an.

Grundsätzlich sind stets alle Ihrer Aussagen zu begründen und Ihr Pseudocode ist stets zu kommentieren.

Die Lösungen zu den mit PABS gekennzeichneten Aufgaben, geben Sie bitte nur über das PABS-System ab. Vermerken Sie auf Ihrem Übungsblatt, in welchem Repository (*sXXXXXX*-Nummer) die Abgabe zu finden ist. Geben Sie Ihre Namen hier als Kommentare in den Quelltextdateien an.