Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

Informatik 8
RWTH Aachen

Datenstrukturen und Algorithmen (SS 2013)

Übungsblatt 3

Abgabe: Montag, **06.05.2013**, 14:00 Uhr

- Die Übungen sollen in Gruppen von zwei bis drei Personen bearbeitet werden.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auf die abgegebenen Lösungen.
- Schreiben Sie die Namen jedes Gruppenmitglieds sowie alle Matrikelnummern auch in die Quellcode-Dateien.
- Geben Sie Ihre Lösungen am **Anfang** der Globalübung, montags, 14:00 Uhr, ab.
- Schicken Sie den jeweiligen Quellcode bitte per **E-Mail** direkt an Ihre/n Tutor/in.
- Geben Sie außerdem den ausgedruckten Quellcode zusammen mit den schriftlichen Lösungen ab.
- Zu spät abgegebene Lösungen werden nicht bewertet.
- Sofern nicht anders gefordert, müssen alle Lösungen und Zwischenschritte kommentiert werden.

Prof. Dr. Leif Kobbelt
Thomas Ströder, Fabian Emmes
Sven Middelberg, Michael Kremer

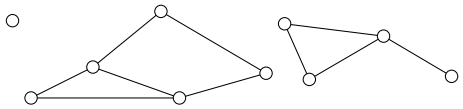
Informatik 8
RWTH Aachen

Aufgabe 1 (Graphen [**10 Punkte**])

1. Stellen Sie die folgende Adjazenzliste als Adjazenzmatrix A sowie als Diagramm dar. Berechnen Sie $(A+I)^2$ und $(A+I)^3$ wobei I die Einheitsmatrix ist. [4 Punkte]

   1: 3, 4
   2: 3
   3: 1, 2
   4: 1, 5
   5: 4

2. Gegeben sei ein ungerichteter, planarer Graph mit v Knoten, e Kanten, f Facetten und c Zusammenhangskomponenten. Die Zahl der Facetten eines planaren Graphen ist die Zahl der von Kanten begrenzten Gebiete in einer beliebigen planaren Darstellung des Graphen, die keine anderen Kanten enthalten. Das außerhalb des Graphen liegende Gebiet zählt hierbei nicht als Facette. Für den untenstehenden Graph ist z. B. $v = 10$, $e = 10$, $f = 3$, $c = 3$.

   Für solche Graphen gilt die Euler-Formel

   $$v - e + f = c.$$

   Diese Formel soll im Folgenden hergeleitet werden.

   (a) Zeigen Sie zunächst: Die Euler-Formel gilt für Bäume (hier: Baum = zusammenhängender, ungerichteter, zyklenfreier Graph). [2 Punkte]

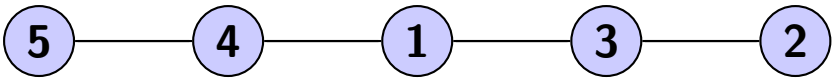
   (b) Beweisen Sie die allgemeine Euler-Formel für zusammenhängende Graphen und $e \geq 1$ durch Induktion über e. Verwenden sie im Induktionsschritt Teilaufgabe (a). [2 Punkte]

   (c) Beweisen Sie die Behauptung unter Verwendung von Teil (b) für nichtzusammenhängende Graphen. [2 Punkte]

Lösungsvorschlag

1.

$$A := \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$



$$A+I = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$(A+I)^2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$(A+I)^3 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

2. (a) Da Bäume zyklenfrei sind, haben sie auch keine Facetten. Somit gilt es, $v-e=1$ zu zeigen. Wir zeigen die Aussage induktiv über die Anzahl der Knoten v im Baum. Ein Baum mit nur einem Knoten erfüllt die Formel offensichtlich. Ein Baum mit mehr als einem Knoten hat mindestens ein Blatt. Entfernen wir dieses Blatt, entfällt auch genau eine Kante. Nach Induktionshypothese gilt $(v-1)-(e-1)=1$ und somit auch $v-e=1$.

   (b) Für die Hinzunahme einer Kante zu einem bestehenden Graphen G, für den $v-e+f=1$ gilt, existieren zwei Fälle, zwischen denen unterschieden werden muss:

   (1) Die neue Kante e' ist inzident zu einem in G bereits existierenden Knoten und einem neuen Knoten v'.

   (2) Die neue Kante wird zwischen zwei in G existierenden Knoten eingefügt.

   Sei G nun ein solcher Graph und sei G' der Graph nach Hinzunahme einer neuen Kante. v', e' und f' seien die in G' hinzugekommenen Knoten, Kanten und Facetten. Wir betrachten nun vier verschiedene Fälle.

   - Sei G ein Baum und erfülle e Fall (1). Es gilt, wie in Aufgabenteil (a) bereits gezeigt, $(v+v')-(e+e')=1$.
   - Sei G ein Baum und erfülle e' Fall (2). Offensichtlich wird damit ein Zyklus (Facette) erzeugt. Es gilt

   $$\begin{aligned} v-(e+e')+f' &= 1 \\ v-e+1-1 &= 1 \quad \text{da } f'=e'=1. \end{aligned}$$

   - Sei G ein zyklischer Graph und erfülle e' Fall (1). Es gilt

   $$\begin{aligned} (v+v')-(e+e')+f &= 1 \\ v-e+f+1-1 &= 1 \quad \text{da } v'=e'=1 \\ v-e+f &= 1. \end{aligned}$$

– Sei G ein zyklischer Graph und erfülle e' Fall (2). Da per Voraussetzung gilt, dass G zusammenhängend ist, wird mit Hinzunahme von e' ein neuer Zyklus (Facette) in G' erzeugt. Es gilt

$$\begin{aligned} v - (e + e') + (f + f') &= 1 \\ v - e + f - 1 + 1 &= 1 \quad \text{da } e' = f' = 1 \\ v - e + f &= 1. \end{aligned}$$

(c) Für $c = 1$ folgt die Aussage aus (b). Sei $c > 1$, dann läßt sich jeder Knoten, jede Kante und jede Facette genau einer Komponente zuordnen. Wähle eine beliebige Komponente in dem Graphen. Dann gilt nach (b) für diese einzelne Komponente $v' - e' + f' = 1$ (wobei v', e' und f' jeweils die Zahl der Knoten, Kanten und Facetten der Komponente sind). Für den Teil des Graphen ohne diese Komponente gilt nach Induktionshypothese $v - e + f = c$. Somit folgt für den gesamten Graphen $(v + v') - (e + e') + (f + f') = c + 1$, bei $v + v'$ Knoten, $e + e'$ Kanten, $f + f'$ Facetten und $c + 1$ Komponenten.

Aufgabe 2 (Programmierung: Heap [**10 Punkte**])

1. Implementieren Sie die Datenstruktur Heap (Warteschlange, die die Heap-Bedingung erfüllt) in Java. Verwenden Sie dazu die effiziente Implementierung auf einem Array (Vorlesung 1.4, Folien 11 ff.) Die Implementierung soll insbesondere die Funktionen `Enq` zum Hinzufügen eines Elements zum Heap, `Deq` zum Entfernen des obersten Elements und `Get` zum Zugriff auf das erste Element enthalten. In dem bereitgestellten Code ist die Warteschlange bereits als Integer-Array

   ```
   private int[] S;
   ```

   als Member-Variable der Klasse `Heap` deklariert. Diese wird im Konstruktor mit der festen Größe von 10 Elementen angelegt:

   ```
   this.S = new int[10];
   ```

   Vervollständigen Sie die im Code deklarierten Methoden `Enq`, `Deq`, sowie `Get`. Die zu bearbeitenden Stellen sind bereits im Code markiert. Sie dürfen alle existierenden Methoden in der Klasse `Heap` benutzen. [8 Punkte]

2. In der von uns bereitgestellten `main()`-Methode wird eine Instanz der Klasse `Heap` erzeugt und nacheinander einige Testwerte der Warteschlange hinzugefügt und wieder herausgenommen.

   Vervollständigen Sie die Funktion `printQueue` in der Klasse `Heap` so, dass sie die Prioritätsschlange mit Elementen x_0 bis x_{n-1} in der folgenden Form ausgibt:

   $$[x_0, x_1, x_2, \ldots, x_{n-1}]$$

   Beim Ausführen der `main()`-Methode erscheint nun jeweils das Zwischenergebnis jeder Operation. Verifizieren Sie, dass Ihre Implementierung stets das richtige Ergebnis produziert und die ausgegebene Prioritätsschlange zu jeder Zeit die Heap-Bedingung erfüllt. Führen Sie hierzu die von uns bereitgestellte `main()`-Methode aus. Editieren Sie die Klasse `MainClass` *nicht*! Sie brauchen zu diesem Aufgabenteil nichts aufschreiben. Drucken Sie lediglich die produzierte Ausgabe aus und geben Sie den Ausdruck zusammen mit dem Ausdruck des Quellcodes der Klasse `Heap` ab. [2 Punkte]

Lösungsvorschlag

1. Der in der Vorlesung vorgestellte Binärbaum, der die Heap-Eigenschaft erfüllt, kann in effizienter Art und Weise als Array implementiert werden. Dies geht, weil der Baum stets vollständig (linksseitig aufgefüllt ist). Da uns in einer Prioritätsschlange lediglich das *größte* Element interessiert, ist es belanglos, dass die jeweiligen Kindknoten im Baum keiner speziellen Ordnung untereinander unterliegen. Die Implementierung funktioniert so, dass der Wurzelknoten immer an Array-Position 1 (bzw. bei null-indizierten Arrays an Position 0) ist.

Die Kindknoten eines Knoten $i \geq 1$ sind dann an Array-Position $2i$ und $2i+1$. Dementsprechend befindet sich der Elternknoten eines Knoten i immer an Array-Position $\lfloor \frac{i}{2} \rfloor$. In unserem Fall beschränken wir die Länge des Arrays auf 10 Elemente.

Beim Einfügen und Herausnehmen eines Elements muss nun jeweils darauf geachtet werden, dass die Heap-Eigenschaft des Baumes erfüllt ist, d.h., dass

(a) der Baum immer linksseitig aufgefüllt ist und

(b) der Wert jedes Elternknoten stets größer als oder gleich der seiner Kinder ist.

In der gewählten Array-Implementierung ist der erste Punkt trivial, da wir per Konstruktion den Baum immer von links auffüllen. Es muss nur sichergestellt werden, dass der Back-Pointer immer auf die Array-Position *hinter* der letzten besetzten Position zeigt (in einer leeren Schlange auf Position 1). Somit gilt: Wenn der Back-Pointer auf Position 1 zeigt, ist die Schlange leer.

Die zweite Bedingung kann beim Einfügen oder Herausnehmen von Elementen verletzt werden und muss ggf. wiederhergestellt werden. Beide Funktionen werden im Folgenden beschrieben.

Einfügen Zeigt der Back-Pointer auf die elfte Stelle im Array (Länge + 1), ist die Schlange voll und ein Fehler wird zurückgegeben (hier eine Java-Exception). Ansonsten wird das neue Element an die letzte Position—also die Position, auf die der Back-Pointer zeigt—geschrieben und solange mit seinem Elternknoten vertauscht, bis man eine Position gefunden hat, an der der Elternknoten *größer als oder gleich* dem neu eingefügten Element ist. Der Back-Pointer wird um eins erhöht.

Herausnehmen Zeigt der Back-Pointer auf die erste Position im Array, wird ein Fehler ausgegeben. Nun wird der Wurzelknoten, also das Element an Array-Position 1, entfernt und durch das letzte Element im Array, also das Element an Position Back-Pointer - 1, ersetzt. Dies verletzt in der Regel die Heap-Eigenschaft, da das letzte Element mit sehr hoher Wahrscheinlichkeit kleiner ist als der Wurzelknoten (es sei denn, alle Elemente haben denselben Wert). Nun wird der neue Wurzelknoten solange mit dem *größeren* seiner beiden Kinder vertauscht, bis eine Position gefunden ist, an der beide Kindknoten einen kleineren oder maximal gleich großen Wert haben. Die Position des Back-Pointers wird um eins verringert.

Das Implementieren der Funktion `Get` ist trivial, da dort lediglich das erste Element im Array zurückgegeben werden muss. Bei einem leeren Array sollte dies einen Fehler produzieren—in dem Falle wieder eine Java-Exception.