Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
M. Bux, B. Grußien, L. Heimberg, P. Schäfer

Berlin, den 19.06.2017

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

# Übungsblatt 5

**Abgabe:** Montag den 03.07.2017 bis 11:10 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten neben Raum 3.321, RUD25.

Die Übungsblätter sind in Gruppen von zwei (in Ausnahmen drei) Personen zu bearbeiten. Jedes Übungsblatt muss bearbeitet werden. (Sie müssen mindestens ein Blatt für wenigstens eine Aufgabe jedes Übungsblattes abgeben.) Die Lösungen sind auf nach Aufgaben getrennten Blättern abzugeben. Heften Sie bitte die zu einer Aufgabe gehörenden Blätter vor der Abgabe zusammen. Vermerken Sie auf allen Abgaben die Namen und **CMS-Benutzernamen** aller Gruppenmitglieder, Ihre Abgabegruppe (z.B. AG123) aus Moodle, und den Übungstermin (z.B. Di 13 Uhr bei Marc Bux), zu dem Sie Ihre korrigierten Blätter zurückerhalten werden.

Beachten Sie die Informationen auf der Übungswebseite (https://hu.berlin/algodat17).

## Aufgabe 1 (Dijkstra mit negativen Kantenkosten) 4+3+4 = 11 Punkte

Eine Voraussetzung des Algorithmus von Dijkstra ist, dass alle Kantenkosten nichtnegativ (das heißt mindestens 0) sind.

1. Geben Sie ein (möglichst kleines) Beispiel eines kantengewichteten gerichteten Graphen mit beliebigen Kantenkosten an, der keinen negativen Kreis enthält und bei dem der Dijkstra-Algorithmus ein falsches Ergebnis produziert. Geben Sie auch den Startknoten an. Nennen Sie einen Knoten Ihres Beispiels, für den der Algorithmus ein falsches Ergebnis liefert, und beschreiben Sie kurz, inwiefern das Ergebnis falsch ist und wie das falsche Ergebnis zustande kommt.

2. Erläutern Sie kurz, wie man den Algorithmus von Dijkstra auf einfache Art erweitern kann, wenn man nicht nur an der Distanz zwischen einem Startknoten und einem beliebigen anderen Knoten interessiert ist sondern auch am kürzesten Pfad zwischen den Knoten.

3. Wir wollen die folgende natürliche Idee untersuchen: Falls ein gegebener gerichteter Graph $G$ auch negative Kantenkosten hat, jedoch keinen negativen Kreis enthält, dann addiere auf alle Kantenkosten eine Konstante $k$, so dass danach alle Kantenkosten nichtnegativ sind. Sei $G'$ der so modifizierte Graph. Beweisen oder widerlegen Sie, dass die kürzesten Pfade in $G$ und in $G'$ identisch sind.

**Aufgabe 2 (Binäre Max-Heaps)** **5+4+4+4 = 17 Punkte**

In der Vorlesung haben Sie bereits binäre Min-Heaps kennengelernt. In dieser Aufgabe betrachten wir binäre Max-Heaps sowie eine Methode, diese in Arrays zu implementieren. Beachten Sie, dass ein Array der Länge $n$ in dieser Aufgabe von 1 bis $n$ indiziert ist.
Im Gegensatz zu einem Min-Heap hat ein Max-Heap die Eigenschaft, dass der Wert jedes Knotens größer als der Wert seiner Kinder ist. Binäre Max-Heaps werden in dieser Aufgabe als Array repräsentiert. Für solche *heapgeordneten* Arrays gilt, dass für ein Element an Stelle $i$ des Arrays das linke Kind im Heap an Stelle $2i$ und das rechte Kind an Stelle $2i + 1$ im Array zu finden ist. Nehmen Sie die folgenden drei (korrekten) Prozeduren (fast_build_heap, extract_max und heapify) als gegeben an.

**heapify($A, i$)**

**Input:** dyn. Array $A$ mit $n$ Elementen, Index $i$

```
1:  left := 2i                    # linker Kindknoten
2:  right := 2i + 1               # rechter Kindknoten
3:  largest := i
4:  if left ≤ n and A[left] > A[largest] then
5:     largest := left
6:  end if
7:  if right ≤ n and A[right] > A[largest] then
8:     largest := right
9:  end if
10: if largest ≠ i then
11:    swap(A, i, largest)
12:    if largest ≤ ⌊n/2⌋ then
13:       heapify(A, largest)
14:    end if
15: end if
```

**fast_build_heap($A$)**

**Input:** dyn. Array $A$ mit $n$ Elementen

```
1: for i = ⌊n/2⌋ to 1 do
2:    heapify(A, i)
3: end for
```

**extract_max($A$)**

**Input:** dyn. Array $A$ mit $n$ Elementen
**Output:** maximales Element aus $A$

```
1: max := A[1]
2: swap(A, 1, n)
3: entferne letztes Element aus A
4: heapify(A, 1)
5: return max
```

Hierbei vertauscht swap($A$,$x$,$y$) die Einträge im Array $A$, die an Index $x$ und $y$ stehen. Bei $A$ handelt es sich um ein dynamisches Array mit veränderlicher Größe (vgl. ArrayList in Java). Wird also im Algorithmus extract_max($A$) in Zeile 3 das letzte Element aus $A$ entfernt, so reduziert sich auch die Größe $n$ von $A$ automatisch um eins.

1. Führen Sie einen Schreibtischtest für den Algorithmus fast_build_heap für das Eingabe-Array
$$A = [\,10,\ 5,\ 1,\ 13,\ 6,\ 14,\ 18,\ 20,\ 8,\ 3,\ 16,\ 9\,]$$
durch. Geben Sie für jede ausgeführte swap-Operation die Indizes der getauschten Elemente und das jeweils resultierende Array an. Stellen Sie zudem nach jeder ausgeführten swap-Operation den Binärbaum, den $A$ repräsentiert, graphisch dar.

2. Führen Sie einen Schreibtischtest für den Algorithmus extract_max für das heapgeordnete Array
$$A = [\,95,\ 40,\ 10,\ 21,\ 13,\ 3,\ 8,\ 12,\ 14,\ 5,\ 9,\ 1\,]$$
durch. Geben Sie jeweils nach den Schritten 2 und 3 in extract_max sowie nach jeder Ausführung einer swap-Operation in der Funktion heapify das veränderte Array $A$ an. Stellen Sie außerdem für jeden solchen Schritt den erzeugten Binärbaum auch graphisch dar. Geben Sie für jede swap-Operation zusätzlich die Positionen die getauscht werden an.

3. Sie haben einen binären Max-Heap gegeben, der durch ein heapgeordnetes Array repräsentiert ist. Zeigen Sie, dass ein *minimales* Element des Heaps mit höchstens $\lceil \frac{n}{2} \rceil - 1$ Vergleichen gefunden werden kann.

4. Im folgenden betrachten wir deterministische Algorithmen, die als Eingabe einen Wert $k$ und einen binären Max-Heap $H$ erhalten, und ausgeben, ob der Wert $k$ im Heap $H$ enthalten ist. Zeigen Sie, dass jeder solche Algorithmus im Worst Case $\Omega(n)$ viele Vergleiche benötigt.

   *Hinweis: Benutzen Sie die Tatsache, dass jeder deterministische Suchalgorithmus, der in einem ungeordneten Array mit $m$ Elementen nach einem speziellen Wert $k$ sucht, im Worst Case $\Omega(m)$ viele Vergleiche ausführen muss.*

**Aufgabe 3 (binäre Min-Heaps mit Update)** **4+4 = 8 Punkte**

In dieser Aufgabe sollen Sie die Operationen add(key, value) und update(key, value) für einen binären Min-Heap in Java implementieren. Die Größe eines solchen Heaps wird durch eine natürliche Zahl $n$ festgelegt. Ein binärer Min-Heap der Größe $n$ wird durch ein heapgeordnetes Array implementiert, welches maximal $n$ Tupel (*key*, *value*) speichert. In jedem solchen Tupel ist *key* eine beliebige natürliche Zahl und *value* eine natürliche Zahl aus der Menge $\{0, \ldots, n-1\}$.
Der *key* jedes Tupels soll dabei höchstens so groß sein wie der *key* seines linken und seines rechten Kindes (wenn vorhanden). Zudem soll der Heap zu jedem Zeitpunkt für jedes $i \in \{0, \ldots, n-1\}$ höchstens ein Tupel (*key*, *value*) mit $value = i$ enthalten.
Beachten Sie, dass ein Array der Länge $n$ in Java von 0 bis $n-1$ indiziert ist. Ergänzen Sie den fehlenden Code in der Klasse `MinHeap` in der Vorlage `MinHeap.java`. Implementieren Sie insbesondere die folgenden Methoden:

1. `add(int key, int value)`: Ist *value* keine natürliche Zahl aus $\{0, \ldots, n-1\}$ oder existiert im Heap bereits ein Tupel ($key'$, *value*), d.h., ein Tupel, welches ebenfalls den Wert *value* hat, dann wirft die Methode eine `BadValueException`. Ansonsten wird das Tupel (*key*, *value*) dem Heap hinzugefügt.

2. `update(int key, int value)`: Enthält der Heap ein Tupel ($key'$, *value*), so wird dieses durch das Tupel (*key*, *value*) ersetzt. Ansonsten wirft die Methode eine `BadValueException`.

   *Beispiel:* Sei der binäre Min-Heap durch das heapgeordnete Array $[(3,2),(4,3),(5,0),(8,1)]$ repräsentiert. Nach Änderung des Schlüssels des Tupels $(3,2)$ von 3 auf 9, hat das entstehende heapgeordnete Array die Form $[(4,3),(8,1),(5,0),(9,2)]$.

Implementieren Sie die Methoden `add` und `update` mit einer Worst Case Laufzeit von $\mathcal{O}(\log n)$. Achten Sie darauf, dass der Heap nach jedem Methodenaufruf die *Form-constraint* und die *Heap-constraint* (siehe Folie 35) erfüllt.
Sie können der Klasse `MinHeap` für die Implementierung der Methoden `add` und `update` auch neue Methoden und Attribute hinzufügen. Erweitern Sie gegebenenfalls auch den Rumpf des vorgegebenen Konstruktors `MinHeap(int n)`. Benutzen Sie jedoch keine zusätzlichen Bibliotheken.

*Hinweise zur Abgabe:* Stellen Sie sicher, dass alle Testfälle in der `main`-Methode der Datei `MinHeap.java` ohne Fehlermeldungen erfolgreich durchlaufen werden. Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, so dass die einzelnen Schritte nachvollziehbar sind. Die Abgabe der von Ihnen modifizierten Datei `MinHeap.java` erfolgt über Moodle.

**Aufgabe 4 (Hashing-Schreibtischtest)** **2+3+3+3+3 = 14 Punkte**

Beim offenen Hashing bestimmt die Sondierungsreihenfolge für jeden Schlüssel, in welcher Reihenfolge die Einträge der Hashtabelle auf einen freien Platz hin durchsucht werden. Da beim Hashing überwiegend mit modulo-Werten gerechnet wird, ist eine Hashtabelle der Größe $n$ von 0 bis $n-1$ indiziert.
Gegeben sei eine Hashtabelle mit 11 Feldern für Einträge, die durch $0, \ldots, 10$ indiziert sind, und die Hashfunktion $h(k) = k \bmod 11$.
Führen Sie für die folgenden Hashverfahren einen Schreibtischtest durch, indem Sie jeweils die Elemente 32, 43, 16, 26, 5, 60 und 54 in dieser Reihenfolge in eine anfangs leere Hashtabelle einfügen. Geben Sie die Hashtabelle nach jeder Einfügeoperation an.

1. **Hashing mit direkter Listenverkettung**
   Hierbei ist die Hashtabelle als Array von einfach verketteten Listen realisiert. Das einzufügende Element wird der jeweiligen Liste an ihrem Anfang hinzugefügt.

2. **Offenes Hashing mit linearem Sondieren**
   Hier wird das einzufügende Element bei Kollisionen an der nächsten freien Stelle links von der berechneten Position $h(k)$ eingefügt. Das heißt, die Sondierungsreihenfolge (die Reihenfolge, in der die Plätze im Array durchgegangen werden, bis erstmals ein freier Platz angetroffen wird) ist gegeben durch die Funktion $s(k,i) = (k-i) \bmod 11$ für $i = 0, \ldots, 10$.

3. **Doppeltes Hashing**
   Das doppelte Hashing ist ein offenes Hashing, bei dem die Sondierungsreihenfolge von einer zweiten Hashfunktion $h'(k) = 1 + (k \bmod 7)$ abhängt. Die Position für das $i$-te Sondieren ist bestimmt durch die Funktion $s(k,i) = (h(k) - i \cdot h'(k)) \bmod 11$ für $i = 0, \ldots, 10$.

4. **Geordnetes Hashing**
   Das geordnete Hashing ist ein offenes Hashing, bei dem die gemäß Sondierungsreihenfolge angetroffenen Elemente (hier absteigend) sortiert werden. Die Position eines neu einzufügenden Elements $k$ ist die erste Position gemäß der Sondierungereihenfolge, an der ein kleineres Element oder noch kein Element im Array steht. Wenn auf ein kleineres Element $k'$ getroffen wird, wird dieses durch das neue Element $k$ ersetzt. Danach wird das Element $k'$ neu nach diesem Verfahren eingefügt, und das Ganze rekursiv fortgesetzt. Beim geordneten Hashing werden stets Sondierungsververfahren benutzt, bei denen sich für das Element $k'$ anhand seiner Position direkt die Folgepositionen innerhalb der Sondierungsreihenfolge bestimmen lassen.

   Zum Sondieren nutzen Sie das doppelte Hashing mit der im 3. Aufgabenteil gegebenen Sondierungsfunktion.

5. **Uniformes offenes Hashing**
   Hier erhält jedes Element $k$ mit gleicher Wahrscheinlichkeit eine der 11! Permutationen von $\{0, 1, \ldots, 10\}$ als Sondierungsreihenfolge.

   Sei $L$ eine Funktion, die jedem Element $k$ uniform zufällig eine dieser 11! Permutationen zuweist. Weiterhin sei $L(k)[i]$, für $i = 0, \ldots, 10$, das $i$-te Element der Permutation $L(k)$. Die Sondierungsreihenfolge für ein Element $k$ ist durch die Funktion $s(k,i) = L(k)[i]$ gegeben.

   Als „zufällige“ Permutationen nutzen Sie:

$$L(32) = 6,8,4,0,9,1,5,2,10,3,7 \qquad L(5) = 1,9,5,3,10,4,7,8,6,0,2$$
$$L(43) = 6,0,4,9,7,10,1,2,5,8,3 \qquad L(60) = 2,5,0,9,3,1,8,4,7,10,6$$
$$L(16) = 2,10,9,3,0,8,7,6,1,4,5 \qquad L(54) = 1,9,6,0,4,5,10,2,8,7,3$$
$$L(26) = 6,9,4,5,2,8,10,1,3,7,0$$