Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Stefan Kratsch
Dr. K. Ahrens, F. Hegerfeld, F. Nelles, Dr. P. Schäfer

Berlin, den 25.06.2018

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 6

Abgabe: Montag den 09.07.2018 bis 11:10 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten (RUD 25, Raum 3.321). Die Übungsblätter sind in Gruppen von 2 Personen zu bearbeiten. 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 Ihre Namen, Ihre **CMS-Benutzernamen**, Ihre Abgabegruppe (z.B. AG123) aus Moodle, und Ihren Übungstermin (z.B. Do 13 Uhr bei Florian Nelles), zu dem Sie Ihre korrigierten Blätter zurückerhalten werden.
Beachten Sie die Informationen auf der Übungswebseite (https://hu.berlin/algodat18).

Konventionen:

- Die Indizierung aller Arrays, **mit Ausnahme von Aufgabe 1**, auf diesem Blatt beginnt bei 1.
- Mit der Aufforderung "Analysieren Sie die Laufzeit" ist gemeint, dass Sie eine möglichst gute obere Schranke der Zeitkomplexität angeben und diese begründen sollen.

Aufgabe 1 (Hashing-Schreibtischtest) — 2+3+4+4 = 13 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 m **ausnahmsweise** von 0 bis $m-1$ 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 14, 36, 9, 75, 25, 41 und 52 in dieser Reihenfolge in eine anfangs leere Hashtabelle einfügen. Geben Sie die Hashtabelle nach jeder Einfügeoperation an.
Beachten Sie, dass in dieser Aufgabe nach links sondiert wird und nicht nach rechts.

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

   Sei L eine Funktion, die jedem Element k uniform zufällig eine von 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:

$$L(14) = 4, 3, 5, 9, 2, 0, 10, 7, 1, 6, 8 \qquad L(25) = 0, 2, 10, 6, 1, 5, 8, 3, 4, 9, 7$$
$$L(36) = 4, 9, 5, 2, 8, 1, 0, 7, 10, 3, 6 \qquad L(41) = 7, 10, 9, 2, 6, 0, 3, 5, 8, 1, 4$$
$$L(9) = 7, 1, 2, 6, 9, 3, 8, 4, 0, 5, 10 \qquad L(52) = 0, 2, 4, 9, 5, 10, 1, 7, 3, 8, 6$$
$$L(75) = 4, 2, 5, 10, 7, 3, 1, 0, 6, 8, 9$$

Aufgabe 2 (Binäre Max-Heaps) **5+5+5+4 = 19 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.
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 (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
```

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: n := n − 1 #letztes Element löschen
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).

1. Führen Sie einen Schreibtischtest für den Algorithmus build_heap für das Eingabe-Array

$$A = [\,11,\ 4,\ 2,\ 6,\ 3,\ 9,\ 10,\ 1,\ 12,\ 5,\ 7,\ 8\,]$$

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. Beweisen Sie, dass der Algorithmus build_heap, welcher ein gegebenes Array A mit n Elementen in ein *max-heapgeordnetes* Array A' umwandelt, eine Laufzeit von $\mathcal{O}(n)$ hat.

Hinweis: Nehmen Sie an, dass A genau $2^j - 1$ viele Elemente enthält, für ein beliebiges $j \in \mathbb{N}$. Sie können außerdem ohne Beweis die folgende Identität benutzen:

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = 2.$$

4. 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.

Aufgabe 3 (Median – Algorithmenanalyse) **3 + 3 = 6 Punkte**

Betrachten Sie den folgenden Algorithmus:

Median_via_Heap(Array A)

Input: Array A mit n Elementen
Output: Medianelement aus A

```
1: B:= build_heap(A)              # d. h. B ist max-heapgeordnetes Array A
2: s = ⌊n/2⌋
3: C := [B[s+1], ..., B[n]]       # d.h. C ist Array aller Blattelemente des Max-Heaps B
4: D:= build_heap(C)              # d. h. D ist max-heapgeordnetes Array C
5: return extract_max(D)
```

Der Pseudocode von build_heap und extract_max ist bei Aufgabe 2 zu finden.

1. Analysieren Sie die Laufzeit des obigen Algorithmus.

2. Zeigen Sie, dass der obige Algorithmus zur Bestimmung des Medians **nicht** korrekt ist.

Aufgabe 4 (Verwendung von Heaps) **$3 \cdot 4 = 12$ Punkte**

1. Wie kann eine Queue mit Hilfe eines Min-Heaps realisiert werden? Beschreiben Sie dafür, wie die Methoden `Queue.enqueue(element)` und `Queue.dequeue()` nur unter Verwendung der auf Min-Heaps eingeführten Methoden realisiert werden können. Beide Methoden `Queue.enqueue(element)` und `Queue.dequeue()` sollen die Komplexität $\mathcal{O}(\log n)$ haben.

2. Wie kann ein Max-Heap für das Speichern von ganzen Zahlen nur mit Hilfe eines Min-Heaps realisiert werden? Beschreiben Sie dafür, wie die Methoden `MaxHeap.deleteMax()` und `MaxHeap.add(element)` nur unter Verwendung der auf Min-Heaps eingeführten Methoden realisiert werden können. Der Max-Heap soll das Maximum in $\mathcal{O}(\log n)$-Operationen löschen und extrahieren können und $\mathcal{O}(\log n)$-Operationen für das Einfügen eines neuen Elements brauchen.

3. Wie kann eine Datenstruktur unter Verwendung zweier Heaps (also zweier Min-Heaps, zweier Max-Heaps oder jeweils eines Min- und Max-Heaps) realisiert werden, so dass der Median der eingefügten Elemente stets in Laufzeit $\mathcal{O}(1)$ ermittelt werden kann? Beschreiben Sie dafür die beiden Methoden `printMedian()`, zum Ausgeben des Medians in $\mathcal{O}(1)$ Operationen, und `add(element)`, zum Einfügen eines neuen Elements in $\mathcal{O}(\log n)$ Operationen.

   Für eine Sequenz von n Zahlen $z_1, \ldots, z_n$ ist der Median allgemein definiert als der Wert, der an mittlerer Stelle steht, wenn man die Werte der Größe nach sortiert. Für eine aufsteigend sortierte Sequenz $z_1, \ldots, z_n$ mit $z_1 \leq \cdots \leq z_n$ ist der Median das Element $z_{(n+1)/2}$, falls n ungerade ist und das Element $z_{n/2+1}$, falls n gerade ist. Zum Beispiel: der Median der Sequenz $1, 10, 11$ ist 10 und der Median der Sequenz $0, 1, 2, 10$ ist 2.

Hinweis: Beschreiben Sie jeweils die geforderten Methoden und analysieren Sie deren Laufzeiten. Als Datenstrukturen dürfen ausschließlich die in der jeweiligen Aufgabenstellung genannten Heaps und zusätzlich primitive Datentypen verwendet werden.