Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Ulf Leser
E. Angriman, F. Nelles, M. Sänger, Dr. P. Schäfer

Berlin, den 05.06.2019

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 5

Abgabe: Mittwoch, den 19.06.2019, bis 11:10 Uhr vor der Vorlesung im Hörsaal. Die Übungsblätter sind in Gruppen von 2/3 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 den **Übungstermin** (z.B. Gruppe 2 oder Mo 13 Uhr bei M. Sänger), an dem Sie Ihre korrigierten Blätter zurückerhalten möchten.
Beachten Sie die Informationen auf der Übungswebseite (https://hu.berlin/algodat19).

Konventionen:

- Für ein Array A ist $|A|$ die Länge von A, also die Anzahl der Elemente in A. Die Indizierung aller Arrays auf diesem Blatt beginnt bei 1 (und endet also bei $|A|$).
- Mit der Aufforderung "Analysieren Sie die Laufzeit" ist gemeint, dass Sie eine möglichst gute obere Schranke der Zeitkomplexität angeben und diese begründen sollen.
- 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.

Aufgabe 1 (Binäre Suchbäume) — 4 + 2 = 6 Punkte

Sei B ein binärer Suchbaum mit Wurzel r, der die Schlüssel $b_1 < \ldots < b_n$ enthält. Für jeden Knoten k von B sei $k.\textbf{left}$ der linke Kindsknoten von k, falls dieser existiert, und **null**, falls k keinen linken Kindsknoten hat. Analog sei $k.\textbf{right}$ definiert. Ein Knoten k speichert zusätzlich $k.\textbf{size}$, die Größe des Teilbaums unter k, d.h., die Anzahl der Kindknoten des linken und rechten Teilbaums von k inklusive k selbst. Für jedes Blatt k gilt also $k.\textbf{size} = 1$. Für jeden inneren Knoten k, mit $k.\textbf{left} \neq \textbf{null}$ und $k.\textbf{right} \neq \textbf{null}$ gilt $k.\textbf{size} = k.\textbf{left}.\textbf{size} + k.\textbf{right}.\textbf{size} + 1$.

i) Entwerfen Sie einen möglichst effizienten Algorithmus **selectithLargestElement**(r, i), der als Eingabe die Wurzel r eines Baumes B und eine Zahl i, mit $1 \leq i \leq n$, erhält und den i't größten Schlüssel b_i zurückgibt. Die Punkte werden gestaffelt nach Effizienz der Lösung vergeben. Für einen korrekten Linearzeit-Algorithmus mit konstantem zusätzlichen Speicherbedarf gibt es die volle Punktzahl.

ii) Analysieren Sie die Best-Case und Worst-Case Laufzeit Ihres Algorithmus in Abhängigkeit von n.

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.
Betrachten Sie die drei Prozeduren buildHeap, deleteMax und siftDown (Pseudocodes stehen auf der folgenden Seite). 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 (analog zu ArrayList in Java). Wird also im Algorithmus deleteMax(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 buildHeap (siehe nächste Seite) für das Eingabe-Array

$$A = [\, 11,\ 6,\ 2,\ 14,\ 7,\ 15,\ 19,\ 21,\ 9,\ 4,\ 17,\ 10 \,]$$

   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 deleteMax für das heapgeordnete Array

$$A = [\, 96,\ 41,\ 11,\ 22,\ 14,\ 4,\ 9,\ 13,\ 15,\ 6,\ 10,\ 2 \,]$$

   durch. Geben Sie jeweils nach den Zeilen 2 und 3 in deleteMax sowie nach jeder Ausführung einer swap-Operation in der Funktion siftDown sowohl das veränderte Array A als auch jeweils den erzeugten Binärbaum graphisch dar. Geben Sie für jede swap-Operation die Positionen an, die getauscht werden.

siftDown(A, i)

Input: Heap als 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:         siftDown(A, largest)
14:     end if
15: end if
```

buildHeap(A)

Input: Dyn. Array A mit n unsortierten Elementen

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

deleteMax(A)

Input: Heap als 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: siftDown(A, 1)
5: return max
```

3. Sei H ein binärer Max-Heap mit n Elementen, der durch ein heapgeordnetes Array repräsentiert ist. Zeigen Sie, dass das *kleinste* Element von H mit höchstens $\lceil \frac{n}{2} \rceil - 1$ Vergleichen gefunden werden kann.

4. 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.

Aufgabe 3 (Implementierung d-närer Heaps) **3 + 3 + 3 + 4 = 13 Punkte**

In der Vorlesung haben Sie bereits *binäre Min-Heaps* unter dem Namen *Heaps* kennengelernt. In dieser Aufgabe betrachten wir mit *d-nären Min-Heaps* eine Verallgemeinerung binärer Min-Heaps, in denen Knoten anstatt maximal zwei Kindern, $d \geq 2$ Kinder haben können. Ferner lernen wir eine Methode kennen, solche d-nären Min-Heaps als Array zu repräsentieren und implementieren. Da es sich um eine Java-Implementierungsaufgabe handelt, beginnt im Folgenden die Indizierung von Arrays bei 0.
Für d-näre *heapgeordneten* Arrays gilt, dass für ein Element an Stelle i des Arrays das erste Kind im Heap an Stelle $d \cdot i + 1$, das zweite Kind an Stelle $d \cdot i + 2$, ..., und das d-te Kind an Stelle $d \cdot i + d$ im Array zu finden ist. Für den in Abbildung 1 dargestellten 3-nären Min-Heap ergibt sich somit das heapgeordnete Array $A = [7, 9, 12, 8, 11, 33, 25, 24, 17, 13, 14, 10]$.

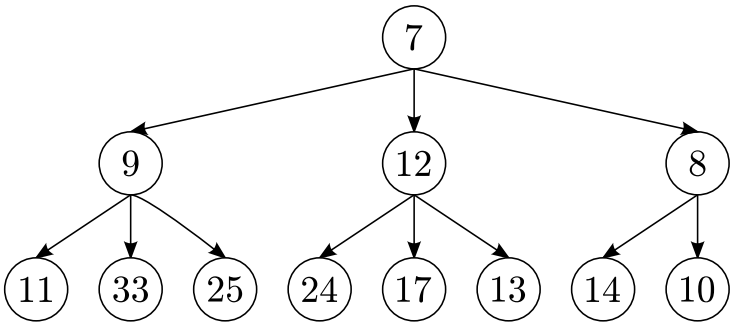

Abbildung 1: Ein 3-närer Heap.

Für die Implementierung d-närer heapgeordneter Arrays finden Sie auf der Übungswebsite[1] und Moodle die Vorlage `DaryHeap.java`. Darin sind, analog zu den Beschreibungen in der Vorlesung (Foliensatz „Priority Queues“, Folien 18 ff.) die nachfolgenden Funktionen vorgegeben:

- `siftDown(i)`: Lässt das Element an Stelle `i` im heapgeordneten Array nach unten sickern (Folie 23). Dabei wird es sukzessive mit dem kleinsten Kindelement vertauscht, solange mindestens ein Kindelement kleiner ist.
- `deleteMin()`: Liefert und entfernt das kleinste Element im heapgeordneten Array (Folie 23 ff.). Nach Entfernen des Minimums wird die Heap-Eigenschaft mittels `siftDown(i)` wiederhergestellt.

[1] https://hu.berlin/algodat19

Ergänzen Sie die Vorlage `DaryHeap.java` unter Ausnutzung der existierenden Funktionen-Stubs um die nachfolgenden Funktionen:

- `build(list)`: Erstellt ein neues heapgeordnetes Array, welches die Elemente der übergebenen Liste `list` enthält. Wie in der Vorlesung beschrieben (Folien 32 ff.) soll dabei nach der Bottom-Up-Sift-Down-Methode vorgegangen werden, um die lineare Laufzeit $\mathcal{O}(n)$ zu erreichen.
- `siftUp(i)`: Lässt das Element an Stelle `i` im heapgeordneten Array nach oben sickern. Analog zur Methode `siftDown(i)` und wie in der Vorlesung beschrieben (Folie 23) wird es dazu sukzessive mit Elternelementen verglichen und vertauscht, solange das Elternelement größer ist.
- `add(element)`: Fügt dem heapgeordneten Array ein neues Element `element` hinzu. Wie in der Vorlesung beschrieben (Folien 26 ff.) soll dafür die Methode `siftUp(i)` verwendet werden, um die logarithmische Laufzeit $\mathcal{O}(\log n)$ zu erreichen.
- `smallerAs(element)`: Gibt alle Elemente aus dem heapgeordneten Array, die kleiner als `element` sind, als Liste zurück, ohne dabei den Heap zu verändern. Die Laufzeit Ihres Algorithmus soll in $\mathcal{O}(k)$ sein, wobei k der Anzahl der Elemente im Array, die kleiner als `element` sind, entspricht.

Sie können davon ausgehen, dass jedes Element nur einmal in das heapgeordnete Array eingefügt wird. Sie brauchen sich also bei der Implementierung der Methoden `build(list)` und `add(element)` nicht um den Umgang mit Duplikaten zu kümmern. Zum Testen können Sie die `main`-Methode in der Vorlage `DaryHeap.java` verwenden. Achten Sie außerdem auf Randbedingungen und Spezialfälle, die von den Testfällen vielleicht nicht vollständig abgedeckt werden.

Hinweis: Ihr Java-Programm muss auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe des von Ihnen modifizierten Quellcodes `DaryHeap.java` erfolgt über Moodle.

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

Beim offenen Hashing bestimmt die Sondierungsreihenfolge, in welcher Reihenfolge die Einträge der Hashtabelle beim Einfügen eines neuen Schlüssels auf auf einen freien Platz hin durchsucht werden. Eine Hashtabelle der Größe n wird 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 Sondierungsreihenfolge, 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 Sondierungsverfahren 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:

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