Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Marius Kloft
Bux, Grußien, Schäfer, Tschorsch, Völlinger

Berlin, den 13.06.2016

Übungen zur Vorlesung
Algorithmen und Datenstrukturen

Übungsblatt 5

Abgabe: Montag den 27.06.2016 bis 10:55 Uhr vor der Vorlesung im Hörsaal oder bis 10:45 Uhr im Briefkasten (RUD 25, Raum 3.321). Die Übungsblätter sind in Gruppen von zwei (in Ausnahmefällen auch drei) 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 Matrikelnummern, den Namen Ihrer Goya-Gruppe und an welchem Übungstermin (Wochentag, Uhrzeit, Dozent) Sie die korrigierten Abgaben abholen möchten. Beachten Sie auch die aktuellen Hinweise auf der Übungswebsite unter https://hu.berlin/algodat16.

Konventionen:

- Für ein Array A ist $|A|$ die Länge von A, also die Anzahl der Elemente in A. Soweit nicht anders angegeben beginnt die Indizierung aller Arrays auf diesem Blatt bei 1 (und endet also bei $|A|$).
- Die durch gaußsche Klammern $\lfloor\ \rfloor$ gekennzeichnete Abrundungsfunktion weist einer reellen Zahl die nächstliegende nicht größere ganze Zahl zu.
- Mit der Aufforderung „Analysieren Sie die Laufzeit“ ist gemeint, dass Sie eine möglichst gute obere Schranke der Zeitkomplexität angeben und begründen sollen.
- Mit log wird der Logarithmus $\log_2$ zur Basis 2 bezeichnet.

Aufgabe 1 (Amortisierte Analyse, Aggregatmethode) — 2 + 4 + 2 = 8 Punkte

In der Vorlesung haben Sie mit der Guthaben- sowie der Potentialmethode zwei Verfahren für die amortisierte Analyse kennengelernt. In dieser Aufgabe beschäftigen wir uns mit der Aggregatmethode, bei der es sich um eine weitere Methode handelt (angedeutet auf Foliensatz „Amortisierte Analyse“, Folie 7). Bei der Aggregatmethode werden die durchschnittlichen Kosten einer Einzeloperation ermittelt, indem die Gesamtkosten aller Operationen bestimmt und anschließend durch die Anzahl der Operationen dividiert werden.

Gegeben sei eine Sequenz von Operationen $O = [o_1, ..., o_n]$. Nehmen Sie an, dass jede Operation o_i die Kosten i hat, wenn i eine Zweierpotenz ist (also für $i = 1, 2, 4, 8, ...$) und andernfalls kostet o_i genau 2. Lösen Sie die folgenden Teilaufgaben:

(a) Berechnen Sie die Kosten für die ersten 16 Operationen. Geben Sie dazu die Einzelkosten für die aufeinanderfolgenden Operationen sowie die aufsummierten Gesamtkosten für alle 16 Operationen an.

(b) Sei $n = 2^k$ für eine natürliche Zahl k. Bestimmen Sie einen Term $T(n)$ für die aufsummierten Gesamtkosten von n Operationen. $T(n)$ soll dabei nur von n und keinen weiteren Variablen abhängen.

(c) Erweitern Sie $T(n)$ für beliebige $n \in \mathbb{N}$. Zeigen Sie unter Verwendung der Aggregatmethode, dass die amortisierten Kosten pro Operation konstant sind. Die amortisierten Kosten ergeben sich als $\frac{T(n)}{n}$.

Aufgabe 2 (Suchverfahren) **(3 + 2 + 4) + 4 = 13 Punkte**

(a) Die in der Vorlesung vorgestellte Interpolationssuche eignet sich am besten dafür, in sortierten Daten mit annähernder Gleichverteilung der enthaltenen Werte zu suchen. Das Verfahren kann wie folgt implementiert werden:

Interpolationssuche

Input: sortiertes Array A der Länge n, zu suchende Zahl c
Output: **true** falls c in A, sonst **false**

```
1:  l := 1; r := n;
2:  while c ≥ A[l] and c ≤ A[r] do
3:    diff := c − A[l];
4:    range := A[r] − A[l];
5:    if range = 0 then
6:      rank := l;
7:    else
8:      rank := l + ⌊(r − l) * diff / range⌋;
9:    end if
10:   if c > A[rank] then
11:     l := rank + 1;
12:   else
13:     if c < A[rank] then
14:       r := rank − 1;
15:     else
16:       return true;
17:     end if
18:   end if
19: end while
20: return false;
```

1. Führen Sie einen Schreibtischtest für die Interpolationssuche durch, bei dem das folgende Array A nach dem Wert $c = 34$ durchsucht wird. Geben Sie dazu an, mit welchen Werten die Variablen l, r, diff, range und rank nach jedem Aufruf von Zeile 9 belegt sind.

$A =$	5	6	9	10	12	13	34	39	43	52	63	76

2. Geben Sie eine fünfelementige Worst-case-Instanz für die Interpolationssuche an, wobei das Element, nach dem gesucht wird, im Array enthalten sein soll. Geben Sie dazu das zu durchsuchende, sortierte Array mit fünf *verschiedenen* Elementen sowie das Element, nach dem gesucht werden soll, an.
3. Erläutern Sie, wie sich eine Worst-case-Instanz für beliebig große Arrays (ohne Duplikate) erstellen lässt, und begründen Sie, warum es sich um Worst-case-Instanzen handelt. Auch hier soll das gesuchte Element stets im Array enthalten sein.

(b) Die Fibonacci-Zahlen sind wie folgt definiert: $F_1 = 1$, $F_2 = 1$ und $F_k = F_{k-2} + F_{k-1}$ für alle $k > 2$. Wir betrachten die folgende von der Vorlesung abweichende Implementierung der Fibonacci-Suche:

Fibonacci-Suche

Input: sortiertes Array A der Länge n, zu suchende Zahl c
Output: **true** falls c in A, sonst **false**

```
1:  fib2 := 1; fib1 := 1; fib := 2;
2:  while fib < n do
3:    fib2 := fib1; fib1 := fib; fib := fib1 + fib2;
4:  end while
5:  offset := 0;
6:  while fib > 1 do
7:    m := min(offset + fib2, n);
8:    if c < A[m] then
9:      fib := fib2; fib1 := fib1 − fib2; fib2:= fib − fib1;
10:   else
11:     if c > A[m] then
12:       fib := fib1; fib1 := fib2; fib2 := fib − fib1; offset := m
13:     else
14:       return true;
15:     end if
16:   end if
17: end while
18: return false;
```

Führen Sie einen Schreibtischtest für die Fibonacci-Suche durch, bei dem das folgende Array B nach dem Wert $c = 17$ durchsucht wird. Geben Sie dazu an, mit welchen Werten die Variablen fib, fib1, fib2, offset und m nach jedem Aufruf von Zeile 7 belegt sind.

$B =$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Aufgabe 3 (Implementierung d-närer Heaps) **3 + 3 + 3 + 4 + 4 = 17 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 der Knoten anstatt zwei Kinder, $d \geq 2$ 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 $A = [7, 9, 12, 8, 11, 33, 25, 24, 17, 13, 14, 10]$.

Für die Implementierung d-närer heapgeordneter Arrays finden Sie auf der Übungswebsite die Vorlage `DaryHeap.java`. Darin sind, analog zu den Beschreibungen in der Vorlesung (Foliensatz „Priority Queues“, Folien 34 ff.) die nachfolgenden Funktionen vorgegeben:

- `siftDown(i)`: Lässt das Element an Stelle `i` im heapgeordneten Array nach unten sickern (Folie 40). 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 37 ff.). Nach Entfernen des Minimums wird die Heap-Eigenschaft mittels `siftDown(i)`

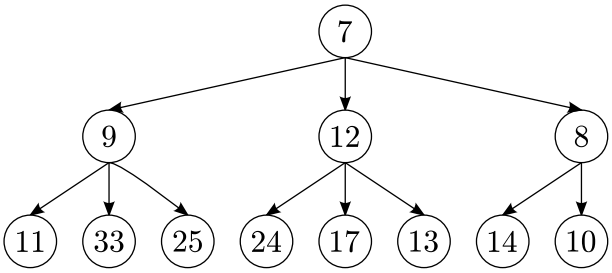

Abbildung 1: Ein 3-närer Heap.

wiederhergestellt.

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 45 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 40) 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 40 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.
- `heapSort(list)`: Sortiert die übergebene Liste `list` mit Hilfe des Sortierverfahrens *Heapsort*, beschrieben auf Folie 47.

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.

Hinweis zur Abgabe: Ihr Java-Programm muss unter *Java 1.7* auf dem Rechner *gruenau2* laufen. Kommentieren Sie Ihren Code, sodass die einzelnen Schritte nachvollziehbar sind. Die Abgabe der von Ihnen modifizierten Datei `DaryHeap.java` erfolgt über Goya.

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

In der Vorlesung haben Sie bereits *binäre Min-Heaps* kennengelernt. In dieser Aufgabe betrachten wir außerdem *binäre Max-Heaps*. Im Gegensatz zu einem Min-Heap hat ein Max-Heap die Eigenschaft, dass der Wert jedes Knotens größer als der Wert seiner Kinder ist.

(a) 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 Verwen-

dung der in Aufgabe 3 auf Min-Heaps eingeführten Methoden realisiert werden können. Beide Methoden `Queue.enqueue(element)` und `Queue.dequeue()` sollen die Komplexität $\mathcal{O}(\log n)$ haben.

(b) 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 in Aufgabe 3 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.

(c) 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.