

Übungsblatt 5

Abgabe: Montag, den 28.06.2021, bis 11:10 Uhr über Moodle. Die Übungsblätter sind in Gruppen von zwei (in Ausnahmefällen drei) Personen zu bearbeiten. Die Lösungen sind auf nach Aufgaben getrennten **PDFs** über Moodle abzugeben. Alle Teilaufgaben einer Aufgabe sind in einem PDF hochzuladen. Pro Gruppe genügt es, wenn eine Person die Lösung der Gruppe abgibt. Zur Bewertung wird der zuletzt hochgeladene Stand herangezogen. Vermerken Sie auf allen Abgaben Ihre **Namen**, Ihre **CMS-Benutzernamen** und Ihre **Abgabegruppe (z.B. AG123)** aus Moodle. Benennen Sie die hochgeladenen PDF-Dateien nach dem Schema $A\langle\text{Aufgabe}\rangle\text{-}\langle\text{Person1}\rangle\text{-}\langle\text{Person2}\rangle.pdf$, bspw. A03-Musterfrau-Beispiel.pdf für Aufgabe 3 von Lisa Musterfrau und Mark Beispiel. Die Auflistung der Namen kann in beliebiger Reihenfolge erfolgen.

Beachten Sie die Informationen im Moodle-Kurs (<https://hu.berlin/algodat21>).

Konventionen:

- Für ein Array A ist $|A|$ die Länge von A , also die Anzahl der Elemente in A . Die Indizierung aller Arrays beginnt bei 1 (und endet also bei $|A|$). Bitte beginnen Sie die Indizierung der Arrays in Ihren Lösungen auch bei 1.
- Mit der Aufforderung “Analysieren Sie die Laufzeit” ist gemeint, dass Sie eine möglichst gute obere Schranke der (Worst Case) Zeitkomplexität angeben und diese begründen sollen.
- Die Menge der natürlichen Zahlen \mathbb{N} enthält die Zahl 0.

Aufgabe 1 (Binäre Max-Heaps)

5+4+4 = 13 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 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 $2i + 1$ 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.

1. Führen Sie einen Schreibtischtest für den Algorithmus `buildHeap` (siehe nächste Seite) für das Eingabe-Array

$$A = [15, 7, 1, 21, 9, 22, 27, 30, 10, 5, 25, 13]$$

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 Schreibtischttest 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.

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.

`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

```

Aufgabe 2 (Implementierung d -närer Min-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 *d -näre Min-Heaps*, eine Verallgemeinerung binärer Min-Heaps, in denen Knoten, anstatt maximal zwei Kinder, $d \geq 2$ Kinder haben können. Ferner lernen wir eine Methode kennen, solche d -nären Min-Heaps als Array zu repräsentieren und zu implementieren. Da es sich um eine Java-Implementierungsaufgabe handelt, beginnt im Folgenden die Indizierung von Arrays bei 0.

Für d -näre *heapgeordnete* Arrays gilt, dass für ein Element e an Stelle i des Arrays das erste Kind von e an Stelle $d \cdot i + 1$, das zweite Kind an Stelle $d \cdot i + 2, \dots$, 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 in Moodle die Vorlage `DaryHeap.java`. Darin sind, analog zu den Beschreibungen in der Vorlesung (Foliensatz „Priority Queues“, Folien 18 ff.) die nachfolgenden Funktionen vorgegeben:

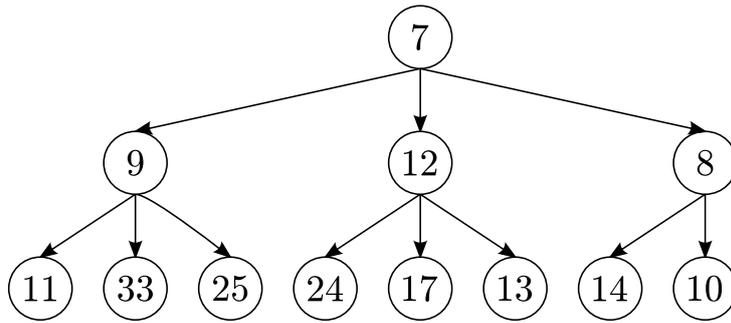


Abbildung 1: Ein 3-närer Heap.

- `siftDown(i)`: Lässt das Element an Stelle `i` im heapgeordneten Array nach unten sickern. 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. Nach Entfernen des Minimums wird die Heap-Eigenschaft mittels `siftDown(i)` 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 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 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 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. *Hinweis*: Diese Funktion benötigt keine der oben genannten Methoden.

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 3 (Hashing-Schreibtischtest)

2+3+3+3=11 Punkte

Eine Hashtabelle der Größe n wird von 0 bis $n-1$ indiziert. Gegeben sei eine Hashtabelle mit 11 Feldern, die durch $0, \dots, 10$ indiziert sind, und die Hashfunktion $h(k) = k \bmod 11$. Führen Sie für die folgenden Hashverfahren einen Schreibtischtest durch, indem Sie jeweils die Elemente 63, 12, 1, 30, 31, 74 und 45 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 ist gegeben durch die Funktion $s(k, i) = (k - i) \bmod 11$ für $i = 0, \dots, 10$.

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 $i = 0, \dots, 10$.

4. Geordnetes Hashing

Das geordnete Hashing ist ein offenes Hashing, bei dem die gemäß Sondierungsreihenfolge angetroffenen Elemente stets absteigend sortiert sind. 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.

Aufgabe 4 (Binäre Suchbäume)

5 + 3 + 5 = 13 Punkte

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

1. Entwerfen Sie einen möglichst effizienten Algorithmus **selectithLargestKey**(r, i), der als Eingabe die Wurzel r des 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 $\mathcal{O}(n)$ mit konstantem zusätzlichem Speicherbedarf gibt es die volle Punktzahl.
2. Analysieren Sie die Best-Case und Worst-Case Laufzeit Ihres Algorithmus aus Teilaufgabe 1. in Abhängigkeit von n .
3. Sei A ein duplikatfreies, sortiertes Array mit $n \geq 1$ ganzzahligen Elementen. Entwerfen Sie einen möglichst effizienten Algorithmus **buildMinSearchTree** in Pseudocode, der A als Eingabe nimmt und einen binären Suchbaum mit minimaler Höhe erzeugt. Der erzeugte Suchbaum soll alle Elemente von A enthalten. Als Ergebnis soll Ihr Algorithmus die Wurzel des binären Suchbaums zurückgeben. Die Punkte werden gestaffelt nach Effizienz der Lösung vergeben. Für einen korrekten Linearzeit-Algorithmus $\mathcal{O}(n)$ gibt es die volle Punktzahl.

Hinweis: Nutzen Sie die oben definierte Datenstruktur "Knoten". Sie können jedoch $k.\text{size}$ ignorieren.