Theoretische Aufgaben

Aufgabe 1

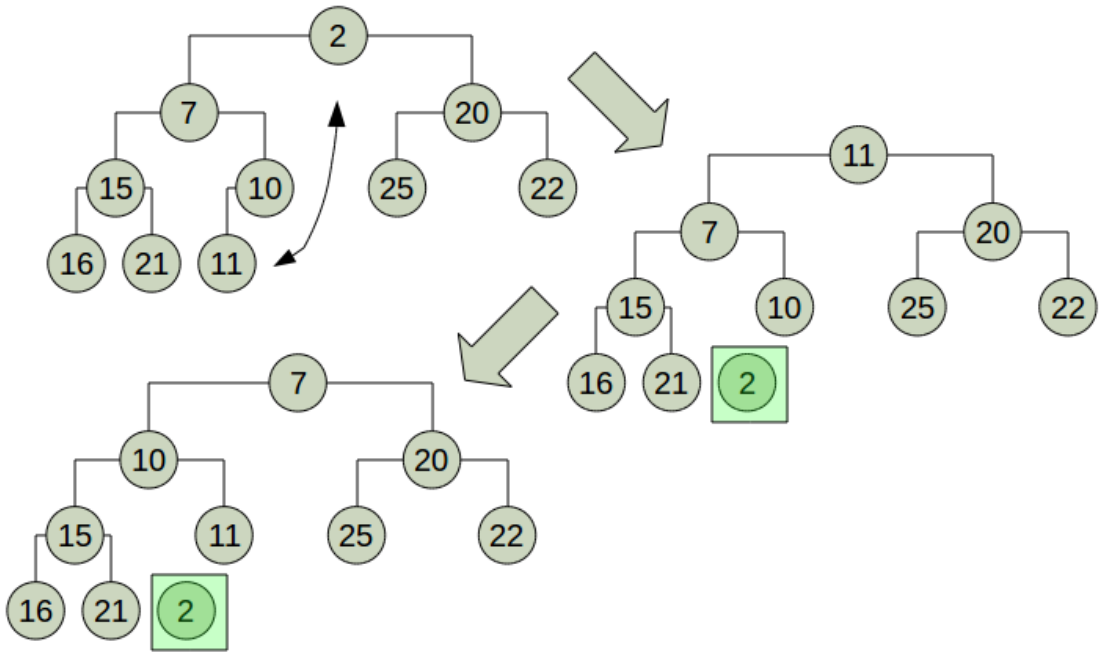

Abbildung 1: Durch das Umsortieren ergibt sich eine Reihenfolge von [7,10,20,15,11,25,22,16,21,2].

Aufgabe 2

Für jedes Blattelement gilt, dass es einen Vater hat, wobei $a_1 \geq a_2$, würde man einen Ast aufwärts nummerieren.
Schritt: Da auch der jeder Vater ein Unterelement eines Vaters a_3 ist, muss auch hier gelten $a_2 \geq a_3$. Da dies unabhängig für jeden Ast gilt, muss für das Wurzelelement a_n, worin alle Teilbäume zusammelaufen, gelten: $a_1 \geq a_2 \geq \cdots \geq a_n$. Somit muss die Wurzel den kleinsten vorkommenden Wert enthalten.

Aufgabe 3

Ein absteigend sortiertes Feld ist ein Max-Heap (sofern darin keine doppelten Elemente vorkommen), weil für jedes Element a_n gilt, dass $a_n > a_{n+1} > a_{n+2}$. Dies enstpricht der Max-Heap-Bedingung: Da auch auf tieferen Stufen des Baumes dieses Prinzip gelten muss, muss auch $a_n > a_{n+t1} > a_{n+t2}$ gelten, wobei $t1, t2 \geq 1$.

Aufgabe 4

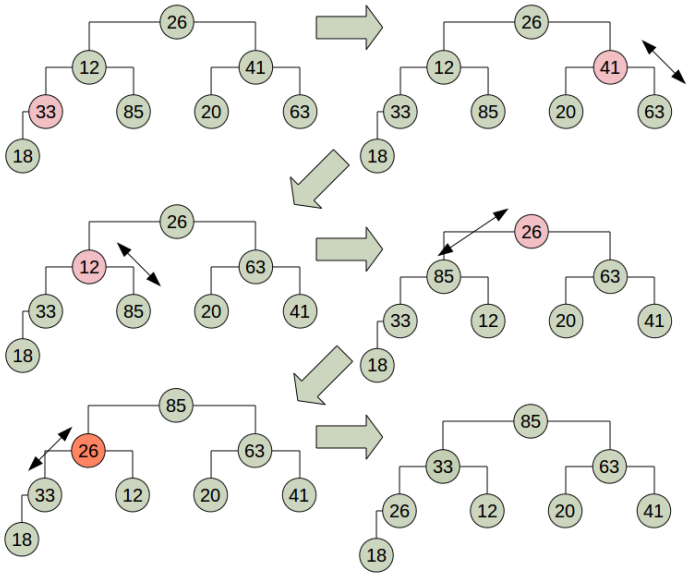

Abbildung 2: Die Blattelemente ausgeschlossen wird von unten her jedes Element durchgegangen und `MAX-HEAPIFY` ausgeführt. Erst bei der 5. Abbildung muss jedoch zwecks Verletzung der Heap-Eigenschaft auf die unteren Knoten zugegriffen werden.

Aufgabe 5

$T(n)$ von Heap sort in den beiden Fällen, wenn das Feld schon absteigend bzw. aufsteigend sortiert ist:

- Bei einem Max-Heap, welcher absteigend sortiert ist, ist die Laufzeit minimal. Dies führt zu folgender Annahme: $O(n)$ wird benötigt, um den Heap zu konstruieren. Für das Absenken der Elemente wird jedoch nur eine konstante Zeit verwendet, da jedes Element schon am richtigen Ort ist: $\Theta(1)$. Zusammengezählt ergibt $\Theta(1) * \Theta(n) = \Theta(n)$.
- Bei einem Max-Heap, der initial aufsteigend sortiert ist, also der *Worst Case*, muss jeder der Bäume beachtet werden. Wenn jedes Element abgesenkt werden muss, führt dies zu einer Laufzeit von $\Theta(n) * \Theta(log_2 n) = \Theta(n \cdot logn)$

Aufgabe 6

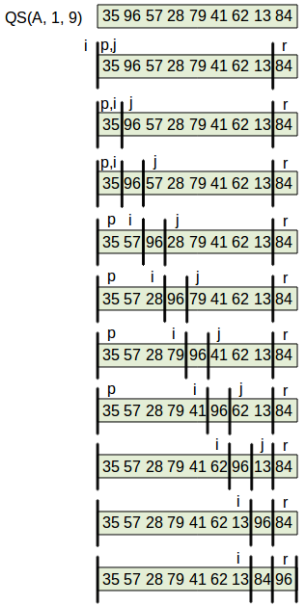

Abbildung 3: Beim ersten Aufruf wird die Zahl 96 heraufgesiebt und anscheinend mit dem pivot 84 getauscht.

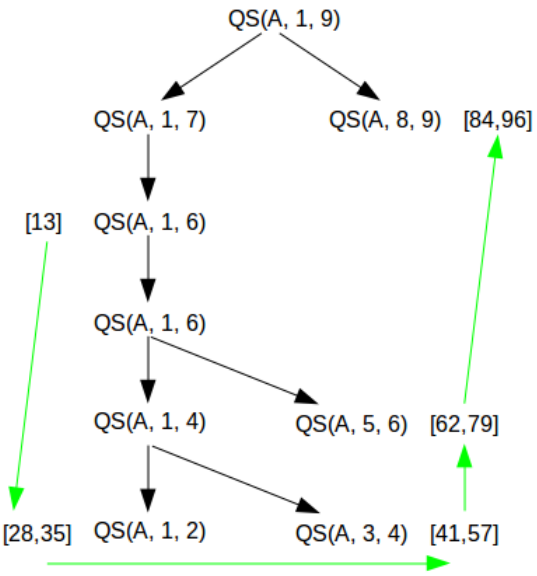

Abbildung 4: Die obenstehenden Aufrufe von Quicksort müssen gemacht werden, damit das Feld vollständig sortiert ist.

Praktische Aufgaben

Aufgabe 1

Siehe `NameVornameComparator.java` und den entsprechenden Output `MiniTestApp.out`.

Aufgabe 2

Wie auf Abbildung 5 angezeigt, wird ein *Stack Overflow Error* ausgelöst. Dies ist wahrscheinlich der Fall, weil Quicksort sich selbst rekursiv zu viele Male aufgerufen hat; Die daraus entstehende Kette von noch zu verarbeitenden Funktionsaufrufen wurde zu lang, als dass die Java VM diese noch verarbeiten

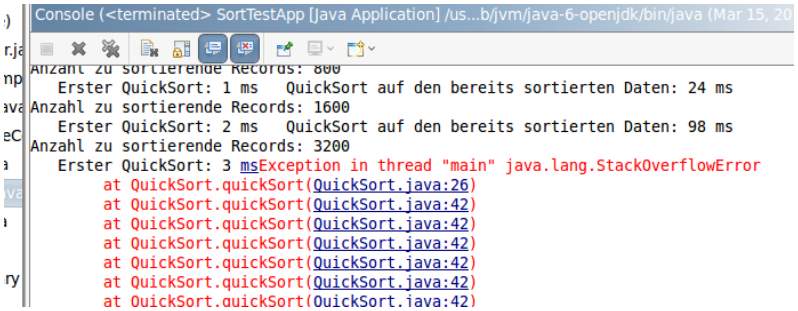

Abbildung 5: Ausgabe von `SortTestApp.java`.

könnte. Dies ist auch der Grund, weshalb der zweite Durchlauf mit sortiertem Input massiv länger braucht (Faktor 50 bei 1600 Elementen).
Eine so grosse Anzahl von Selbstaufrufen ist der Fall, weil eine bereits sortierte Menge nochmals sortiert werden sollte. Hierbei wird jedoch immer nur das letzte Element abgetrennt (der Pivot) und der Rest mit einem erneuten Aufruf von Quicksort zu sortieren versucht. Dies wiederholt sich beim *Stack Overflow Error* so lange, bis die Anzahl Aufrufe zuviel werden; Bei grösser werdenden n braucht es auch n Aufrufe von Quicksort.

Aufgabe 3

Siehe `Quicksort.java` und den entsprechenden Output `Quicksort.out`.