Algorithmen und Datenstrukturen: Übung 2

Alexander Waldenmaier

20. November 2020

Aufgabe 2.1

a) Im günstigsten Fall ist das Array bereits sortiert. Es wird einmal durchlaufen und bei jedem Paar festgestellt, dass dieses bereits größer als das vorige ist. Beim „Bogo-Sort"-Verfahren wäre dies etwa der erste Schritt, nach dem das Verfahren dann sofort beendet wäre.

$$\begin{aligned} V(n) &= n - 1 \\ \Rightarrow V(64) &= 63 \end{aligned}$$

b) Die in der Vorlesung vorgestellte gibt eine obere Schranke für die benötigte Anzahl an Vergleichen im average-case bei n Schritten wie folgt an:

$$\begin{aligned} V_{ac}(n) &\leq 2n \ln n \\ \Rightarrow V_{ac}(64) &\leq 2 \cdot 64 \ln 64 \\ &\leq 533 \end{aligned}$$

c) Das MergeSort-Verfahren benötigt laut Vorlesung höchstens die folgende Anzahl an Vergleichen, egal ob es sich um den best-, average- oder worst-case handelt:

$$\begin{aligned} V(n) &= 2 \cdot V(n/2) + n - 1 \leq n \log_2 n \\ \Rightarrow V(64) &\leq 384 \end{aligned}$$

d) Bei der Verwendung des MergeSort-Algorithmus ist bei Kenntnis der Länge n die Anzahl an benötigten Vergleichen eindeutig bestimmbar. Es handelt sich somit um einen Algorithmus mit fest kalkulierbarer Ausführungszeit, was bei der Entwicklung von zeitkritischen Anwendungen ein großer Vorteil ist. MergeSort ist im worst-case darüberhinaus doppelt so schnell wie HeapSort.

Augabe 2.2

Im worst-case liegt das Array in exakt verkehrter Reihenfolge vor:

```
arr = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Beim Sortieren mithilfe von QuickSort würden im obersten Funktionsaufruf 9 Vergleiche durchgeführt werden und dabei alle 9 Elemente links der 9 ins Array eingefügt werden. Anschließend würde im „linken" rekursiven Aufruf von QuickSort die linken 9 Elemente sortiert werden, mit dem Pivot-Element 8. Dies würde 8 Vergleiche benötigen und 8 Verschiebungen zur Folge haben. Der „rechte" QuickSort-Aufruf ist sofort beendet, da er nur das Element 9 beinhaltet. Auf die verschobenen 8 Elemente wird dann wieder das selbe Schema angewandt. Für die Gesamtanzahl an Vergleichen gilt somit:

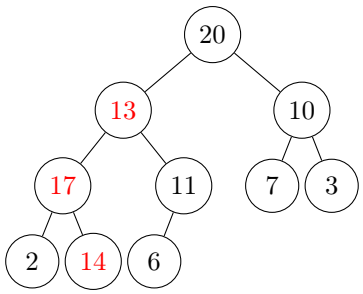
$$\begin{aligned} V_{wc}(n) &= (n-1) + (n-2) + ... + 1 + 0 = \frac{n(n-1)}{2} \\ \Rightarrow V_{wc}(10) &= \frac{10(10-1)}{2} = 45 \end{aligned}$$

Aufgabe 2.3

Da es sich um eine zufällige Permutation handelt, gehen wir in diesem Beispiel vom average-case aus. Ziel ist nun also, die Laufzeit in $\mathcal{O}$-Notation eines bestmöglichen Sortieralgorithmus für den average-case zu bestimmen. Von allen bislang bekannten Algorithmen wäre der „QuickSort"-Algorithmus eine beste Wahl, da er sich im average-case bei zufälligem Input nach $\Theta(n \log n)$ verhält. Demnach ist die Laufzeit (in Abhängigkeit von n) eines bestmöglichen Sortieralgorithmus im average-case $\in \mathcal{O}(n \log n)$.

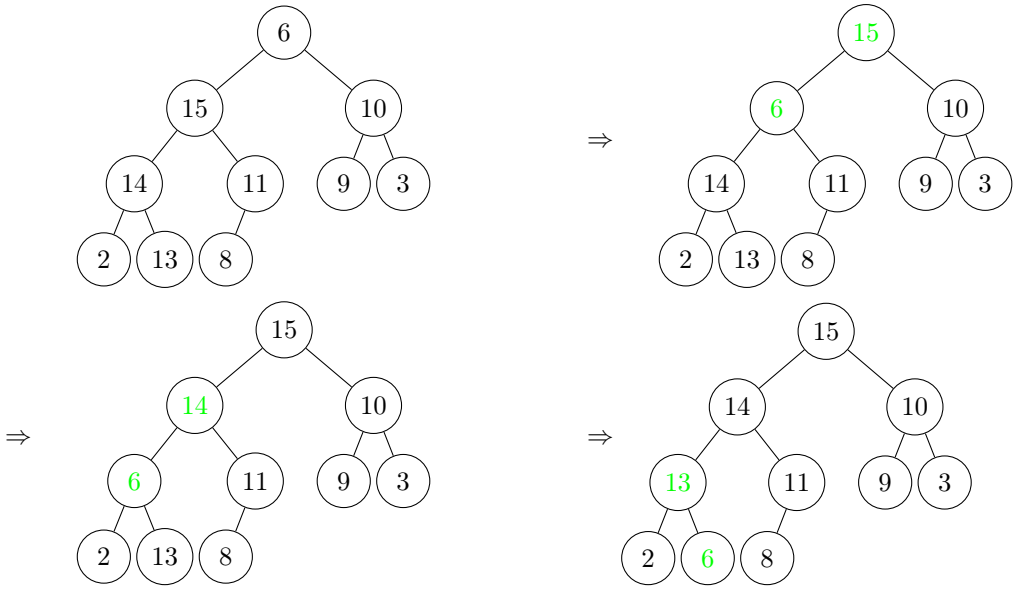
Aufgabe 2.4

a) Baumdarstellung des Heaps:



Es ist klar ersichtlich, dass die 17 den Platz mit der 13 tauschen sollte, da $17 > 13$. Daraus folgt dann auch, dass die 14 mit der 13 vertauscht werden muss, da $14 > 13$. Mit diesen zwei Tauschvorgängen wäre der Heap-Zustand wiederhergestellt.

b) Die Wiederherstellung des Heap-Zustands gelingt in drei Vertauschungen. In grün markiert sind die jeweils vertauschten Elemente:



c) Wir betrachten n Heap-Elemente a_i (mit $i \in 1, ..., n$), wobei i gleichzeitig den Rang des Elements wiederspiegelt (je höher, desto höher der Rang). Angenommen, das zweitgrößte Element a_{n-1} des Heaps ist nicht eines der Kinder der Wurzel a_n, so gilt für beide Kinder

$$a_{Kind} < a_{n-1} < a_n \ ,$$

da die obersten beiden Rangpositionen a_n und a_{n-1} bereits durch das Wurzelelement und das zweitgrößte Element besetzt sind. Wenn nun also das zweitgrößte Element a_{n-1} einem beliebigen der Kinder

untergeordnet wird, wird es damit automatisch einem Element geringeren Ranges untergeordnet. Das verletzt die Heap-Bedingung

$$a_{Mutter} > a_{Kind} \, .$$

Demnach muss eines der beiden Wurzel-Kinder gleichzeitig das zweitgrößte Element sein.

d) Bislang wurde festgestellt, dass eines der beiden Wurzel-Kinder das zweitgrößte Element ist. Versteht man die Wurzel als die „nullte“ Ebene, spricht nichts dagegen, dass das drittgrößte Element sich zusammen mit dem zweitgrößten auf der ersten Ebene befindet. Genauso könnte es sich auch in der zweiten Ebene als Kind des zweitgrößten Elements befinden, nicht jedoch als Kind des anderen Wurzelkinds (da dieses maximal das viertgrößte ist). Allerdings kann es sich keinesfalls in einer niedrigeren Ebene als der zweiten befinden: Das drittgrößte Element verhält sich dann nämlich zum zweitgrößten Element so wie das zweitgrößte Element sich zur Wurzel verhält. Zusammenfassend gilt: Das drittgrößte Element muss ein Kind des zweitgrößten Elements oder der Wurzel sein.

e) Wenn es sich um einen ausgeglichenen binären Baum handelt, bei dem also auch die letzte Ebene die volle Anzahl an möglichen Elementen besitzt, so kann sich das kleinste Element ausschließlich in der letzten Ebene befinden, da es kein Element gibt, wovon es selbst die Mutter sein kann. Wenn es sich allerdings um einen nicht ausgeglichenen Baum handelt, wenn also mindestens ein Element der vorletzten Ebene keine oder nicht alle Kinder besitzt, kann das kleinste Element auch in der vorletzten Ebene auftauchen. Dabei befindet es sich aber stets „rechts“ des letzten Elements der vorletzten Ebene, welches noch mindestens ein Kind hat.

Aufgabe 2.5

Abgabe in DOMjudge. Teamname: "test"