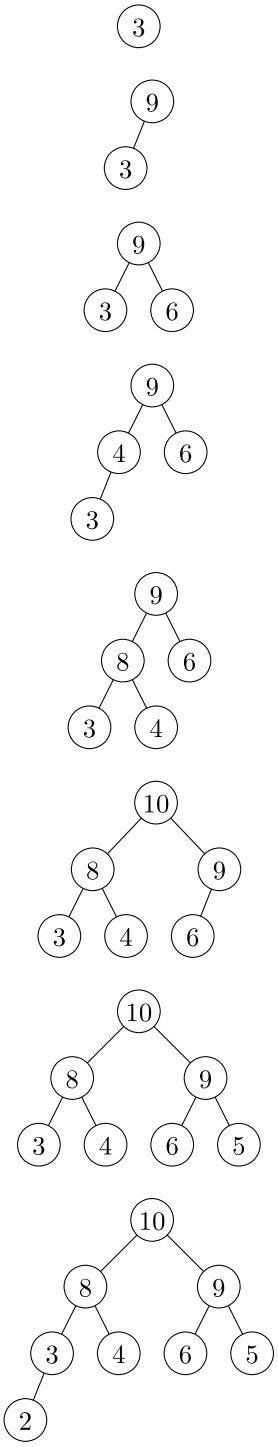
RWTH Aachen
Lehrgebiet Theoretische Informatik
Rossmanith–Hensel–Kuinke–Sánchez

SS 2016
Lösungsvorschlag 7
6. Juni 2016

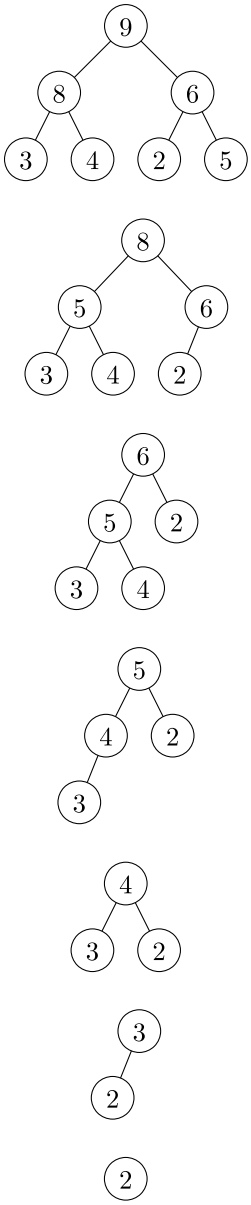
Übung zur Vorlesung Datenstrukturen und Algorithmen

Aufgabe T20

a) Als erstes fügen wir die Elemente der Reihe nach in den Heap ein.



Jetzt sind alle Elemente eingefügt worden. Wiederholtes Löschen der Wurzel liefert die Elemente in absteigender Reihenfolge.



b) Als erstes teilen wir das Array rekursiv in zwei Hälften, bis wir Arrays der Länge eins haben. Anschließend werden die beiden Hälften, die bereits sortiert sind, zusammengefügt um ein neues sortiertes Array zu erhalten.

Die Teilung und Mischung verläuft wie folgt, sortierte Unterarrays, die entstehen wenn die Rekursion terminiert, sind unterstrichen:

- $[3, 9, 6, 4, 8, 10, 5, 2]$
- split: $[3, 9, 6, 4]$ $[8, 10, 5, 2]$
- split: $[3, 9]$ $[6, 4]$ $[8, 10, 5, 2]$
- split: $\underline{[3]}$ $\underline{[9]}$ $[6, 4]$ $[8, 10, 5, 2]$
- merge: $\underline{[3, 9]}$ $[6, 4]$ $[8, 10, 5, 2]$
- split: $\underline{[3, 9]}$ $\underline{[6]}$ $\underline{[4]}$ $[8, 10, 5, 2]$

- merge: $\underline{[3,9]}$ $\underline{[4,6]}$ $[8,10,5,2]$
- merge: $\underline{[3,4,6,9]}$ $[8,10,5,2]$
- split: $\underline{[3,4,6,9]}$ $[8,10]$ $[5,2]$
- split: $\underline{[3,4,6,9]}$ $\underline{[8]}$ $\underline{[10]}$ $[5,2]$
- merge: $\underline{[3,4,6,9]}$ $\underline{[8,10]}$ $[5,2]$
- split: $\underline{[3,4,6,9]}$ $\underline{[8,10]}$ $\underline{[5]}$ $\underline{[2]}$
- merge: $\underline{[3,4,6,9]}$ $\underline{[8,10]}$ $\underline{[2,5]}$
- merge: $\underline{[3,4,6,9]}$ $\underline{[2,5,8,10]}$
- merge: $\underline{[2,3,4,5,6,8,9,10]}$

Aufgabe T21

	Quicksort	Heapsort	Mergesort	Insertion-Sort	Straight-Radix	Radix-Exchange
in-place	??	J	N	J	N	??
stabil	N	N	J	J	J	N
Laufzeit (worst-case)	n^2	$n \log n$	$n \log n$	n^2	nw	nw
Laufzeit (Durchschnitt)	$n \log n$	$n \log n$	$n \log n$	n^2	nw	nw
vergleichsbasiert	J	J	J	J	N	N

Bei den Radix-Sortierverfahren bezeichne w die Wortlänge. Quicksort und Radix-Exchange-Sort sind wegen des benötigten Stacks nicht in-place. Beide lassen sich jedoch so implementieren, daß der Stack in rekursiven Funktionsaufrufen „versteckt“ wird, während jeder einzelne Aufruf nur konstant viel Platz benötigt. Bei Sortierverfahren sagen manche, daß sie nicht in-place sind, wenn sie $\Omega(n)$ viel Platz zusätzlich brauchen, andere, schon bei mehr als konstant vielem Zusatzplatz.

Aufgabe T22

	Liste	Sortierte Liste / Array	Sortiertes Array	Binärer Suchbaum / AVL-Baum	Splay-Tree	Treap	Skip-List	Hashtabelle	Min-Heap
Random	N	N	N	N	N	N	J	J	J	N
Einfügen	1	n	1	n	n	$\log n$	$\log n$	$\log n$	1	n
Suchen	n	n	n	$\log n$	n	$\log n$	$\log n$	$\log n$	1	n
Löschen	n	n	n	n	n	$\log n$	$\log n$	$\log n$	1	n
Minimum	n	1	n	1	n	$\log n$	$\log n$	$\log n$	1	1
Maximum	n	1	n	1	n	$\log n$	$\log n$	$\log n$	n	n
Sort. Aus.	$n \log n$	n	$n \log n$	n	n	n	n	n	$n \log n$	$n \log n$

Aufgabe H18

a) Ein gutes Verfahren ist Bottom-up-Mergesort:

Wir numerieren die schreibbaren Festplatten von 1 bis 4. Wir lesen 16 GB von der *read-only* Festplatte, sortieren diese Unterliste im Speicher, und schreiben diesen abwechselnd auf Festplatte 1 und 2 bis wir das Ende der *read-only* Festplatte erreichen. Jetzt enthalten Festplatten 1 und 2 genau 512 GB an Strings in jeweils 16 GB großen sortierten Blöcken.

Wir nehmen dann einen 16 GB Block aus Festplatte 1 und einen 16 GB Block aus Festplatte zwei und durch *mischen* schreiben wir abwechselnd auf Festplatte 3 und 4 bis wir alle Blöcke gemischt haben. Jetzt enthalten Festplatten 3 und 4 je 512 GB an Strings in jeweils 32 GB großen sortierten Blöcken. Festplatten 1 und 2 können wir jetzt problemlos überschreiben.

Wir wiederholen jetzt diese Prozedur, wobei wir in jedem Schritt die Blockgröße verdoppeln, bis am Ende eine finale *merge*-Operation von zwei 512 GB großen Blöcken zu dem erwünschten Ergebnis führt.

b) Der erste Schritt um die zwei Listen aus geordneten 16 GB Blöcken zu schreiben braucht $1\text{TB}/16\text{GB} \cdot t_1 + t_2 + t_3 \approx 64t_1 + t_2 + t_3$. Beim Mischen können wir von zwei Festplatten parallel lesen, also braucht jede Mischoperation $t_2/2 + t_3$ Zeit. Die Blockgrößen wachsen exponentiell, also $16, 32, 64, 128, 256, 512, 1024$ und es gibt somit sechs *merges*. Somit ist die Gesamtlaufzeit ungefähr $64t_1 + 4t_2 + 7t_3$.

c) Durch Testen kann man herausfinden, daß 16 GB zu sortieren etwa 10min dauert.

Eine andere möglichkeit wäre es die Formel

$$C_n = 1.4n \cdot \log n$$

heranzuziehen (etwa vier Instruktionen pro Schleifendurchlauf bei Quicksort). Wenn wir schätzen, daß ein Vergleich $15\,\text{ns} = 1.5 \cdot 10^{-8}\,\text{s}$ dauert, bekommen wir mit $n = 16\,\text{GB}/16\,\text{byte} = 10^9$

$$C_n = 1.4 \cdot 10^9 \cdot \log 10^9 \cdot 1.5 \cdot 10^{-8}\,\text{s} \approx 7\,\text{m}.$$

Wählen wir $t_1 = 7\,\text{m} = 0.12\,\text{h}$

Mechanische Festplatten mit 7200 rpm, wie sie häufig verbaut werden, bieten Lesegeschwindigkeiten von ungefähr 128 MB/s und Schreibgeschwindigkeiten von ungefähr 120 MB/s. Damit ergibt sich für $t_2 = 1\,\text{TB}/128\,\text{MB/s} = 2.1\,\text{h}$ und für $t_3 = 2.3\,\text{h}$

Die Gesamtzeit nach der Formel aus b) ist somit ungefähr 32.5 Stunden um alle Strings zu sortieren.

Aufgabe H19

Folgende Javafunktion löst die Aufgabe *in-place* in linearer Zeit.

```
void orderArray(int[] a) {
  int numZeros = 0;
  for(int i = 0; i < a.length; i++) {
    if(a[i] == 0) {
      numZeros++;
    }
  }

  int c = 0;
  while(c < a.length) {
    if(a[c] == 0 || c == a[c] + numZeros - 1) {
      c++;
      continue;
    }
    if(a[c] < 0 || a[c] + numZeros - 1 ≥ a.length) {
      throw new RuntimeException("Value in Array outside"
                          + "of allowed range.");
    }
    if(a[c] == a[a[c] + numZeros - 1]) {
      throw new RuntimeException("Array contains " + a[c]
                          + " more than once.");
    }
    int temp = a[c];
    a[c] = a[temp + numZeros - 1];
    a[temp + numZeros - 1] = temp;
  }
}
```

Zuerst zeigen wir, daß das Ergebnis richtig ist, wenn das Array das gewünschte Format hat.

Das erste was die Funktion macht ist, die Anzahl an Nullen *numZeros* im Array zu zählen. Dieser Schritt braucht $O(m)$ Zeit. Wenn man die Anzahl der Nullen kennt, dann weiß man, daß ein Wert w an der Position $w + numZeros - 1$ vom Array stehen muß. Der Zähler c gibt uns die jetzige Position im Array. Wir vertauschen dann den Wert an der jetzigen Position mit dem Wert an der Position wo er sein sollte $(a[c] + numZeros - 1)$. Wenn in der jetzigen Position c eine Null oder der richtige Wert steht, wird der Zähler um Eins erhöht. Von 0 bis $numZeros - 1$ wird der Zähler nur erhöht, wenn eine Null an dieser Position steht. Danach wird der Zähler nur erhöht, wenn der richtige Wert in der Aufzählung nach den Nullen steht. Da wir bei jedem Vertausch einen Wert richtig setzen, können maximal m Vertauschungen stattfinden und der Algorithmus muß terminieren. Somit ist auch gezeigt, daß der Algorithmus in $O(m)$ Zeit läuft.

Es fehlt nur noch zu zeigen, daß ein Fehler ausgegeben wird, wenn die Eingabe nicht das richtige Format hat. Wenn ein Wert mehr als einmal vorkommt, wird solange vertauscht bis dieser Wert zum zweiten mal gefunden wird. Hier wird dann die zweite *RuntimeException* geworfen. Wenn einer der Werte nicht erlaubt ist, also nicht zwischen 0 und m minus die Anzahl an Nullen beträgt, dann wird die erste *RuntimeException* geworfen. Wenn in dem Array einer der Werte ab Eins fehlt, dann ist ein anderer Wert zu groß, oder ein anderer Wert wird wiederholt. Hiermit ist gezeigt, daß wenn das Array nicht das richtige Format hat, ein Fehler ausgegeben wird.