Übung: Höhere Sortieralgorithmen (A2)

1 Quicksort – theoretisch durchgespielt

a)

Trennelement: **17**

Erster Durchgang:

12 10 52_1 9 77 23 18 52_2 11 25 8 5 *17*

12 10 5 9 8 11 *17* 52_2 23 25 77 52_1 18

Zweiter Durchgang:

12 10 5 9 8 *11* | 17 | 52_2 23 25 77 52_1 *18*

8 10 5 9 *11* 12 | 17 | *18* 23 25 77 52_1 52_2

Dritter Durchgang:

9 10 5 *9* | 11 12 17 18 | 23 25 77 52_1 *52_2*

8 5 *9* 10 | 11 12 17 18 | 23 25 52_1 *52_2* 77

Vierter Durchgang:

8 *5* | 9 10 11 12 17 18 | 23 25 *52_1* | 52_2 77

5 8 | 9 10 11 12 17 18 | 23 25 *52_1* | 52_2 77

Fünfter Durchgang

5 8 9 10 11 12 17 18 | 23 *25* | 52_1 52_2 77

5 8 9 10 11 12 17 18 23 *25* 52_1 52_2 77

b) Die Zahl 52_1 war nach dem ersten Durchgang rechts von 52_2. Dass die Reihenfolge im zweiten Durchgang noch einmal (und zwar endgültig) änderte, ist reiner Zufall. Quicksort arbeitet *instabil*.

c) Beim ersten Durchgang kämen 12 (Index 0), 18 (Index 6) und 17 (Index 12) in Frage. Dadurch würde erneut 17 als Trennelement fungieren.

Beim zweiten Durchgang würde es links wiederum genau gleich ablaufen, rechts würde aber mit 25 ein anderes Element verwendet werden. Das könnte die Sortierung etwas beschleunigen und evtl. einen fünften Durchgang ersparen.

2 Quicksort - klassisch programmiert

d)

Elemente (`n`)	Messung (ms)
1000	2
5000	7
10'000	12
50'000	35
100'000	133
500'000	2993
1'000'000	11'888

Beispiel: Um welchen Faktor müsste eine Sortierung mit 1'000'000 Elementen länger dauern als eine Sortierung mit 500'000 bzw. 100'000 Elementen?

```
(1'000'000 * log 1'000'000) / (500'000 * log 500'000) = 2.1
(1'000'000 * log 1'000'000) / (100'000 * log 100'000) = 12
```

Realität:

```
11'888 / 2993 = 3.97
11'888 / 133 = 89.4
```

Das Laufzeitverhalten scheint eher $O(n^2)$ zu entsprechen (eine Verdoppelung der Elemente führt zu einer Verfierfachung der Laufzeit; eine Verzehnfachung der Elemente erhöht die Laufzeit ca. um Faktor 90).

3 Quick-Insertion-Sort

a) `m` ist der Schwellenwert, unter dem der Insertion-Sort verwendet werden soll

b) Ich sortiere 100'000 Zeichen mit verschiedenen `m`-Werten. Dabei erhalte ich folgende Laufzeiten:

m	Zeit (ms)
5	200
10	172
15	164
20	141
25	122
30	119
40	114
50	122
75	127
100	123
125	115
150	110
200	112

m	Zeit (ms)
250	110
500	115
1000	111

Bei einer Datenmenge von 100'000 Zeichen scheinen sinnvolle `m`-Werte von 25 bis 250 zu liegen.

c) Mit `m = 50` kann beim Quick-Insertion-Sort (QIS) gegenüber dem Quick-Sort (QS) einen kleineren Laufzeitvorteil festgestellt werden:

n	QS (ms)	QIS (ms)
1000	2	1
5000	4	9
10000	8	14
50000	34	30
100000	175	146
500000	2766	2196

4 Datenstruktur Heap

a)

5 Übersicht Sortieralgorithmen

a)

Algorithmus	O avg	O worst	O best	stabil	parallel	Merkmale
Direktes Einfügen (Insertion Sort)	$O(n^2)$	$O(n^2)$	$O(n)$	ja	nein	sortierter/u
Direktes Auswählen (Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	nein	nein	sortierter/u
Direktes Austauschen (Bubble Sort)	$O(n^2)$	$O(n^2)$	$O(n)$	ja	nein	Vergleich v
Shellsort	$O(n*\log^2 n)$	$O(n*\log^2 n)$	$O(n*\log n)$	nein	nein	Insertion S
Quicksort (`Arrays.sort()`)	$O(n*\log n)$	$O(n^2)$	$O(n*\log n)$	ja	ja	Divide & C
Mergesort (`Collections.sort()`)	$O(n*\log n)$	$O(n*\log n)$	$O(n*\log n)$	nein	ja	Divide & C
Heapsort	$O(n*\log n)$	$O(n*\log n)$	$O(n*\log n)$	ja	nein	basiert auf

b) `java.util.Arrays.sort(...)`: Quicksort

`java.util.Arrays.parallelSort(...)`: parallel sort-merge that breaks the array into sub-arrays that are themselves sorted and then merged

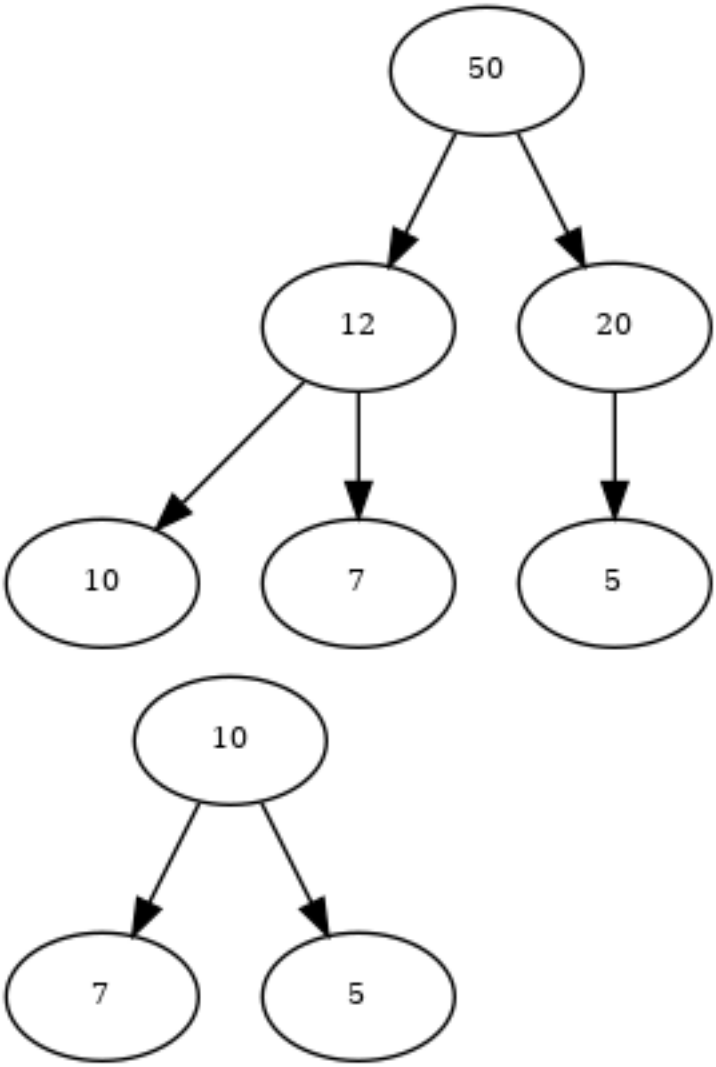

Figure 1: Heap nach dem Auffüllen und nach dem Entfernen

`java.util.Collections.sort(...)`: Mergesort