Gruppe $\boxed{14}$
Datenstrukturen und Algorithmen

Phil Pützstück, 377247
Benedikt Gerlach, 376944
Sebastian Hackenberg, 377550

Hausaufgabe 6

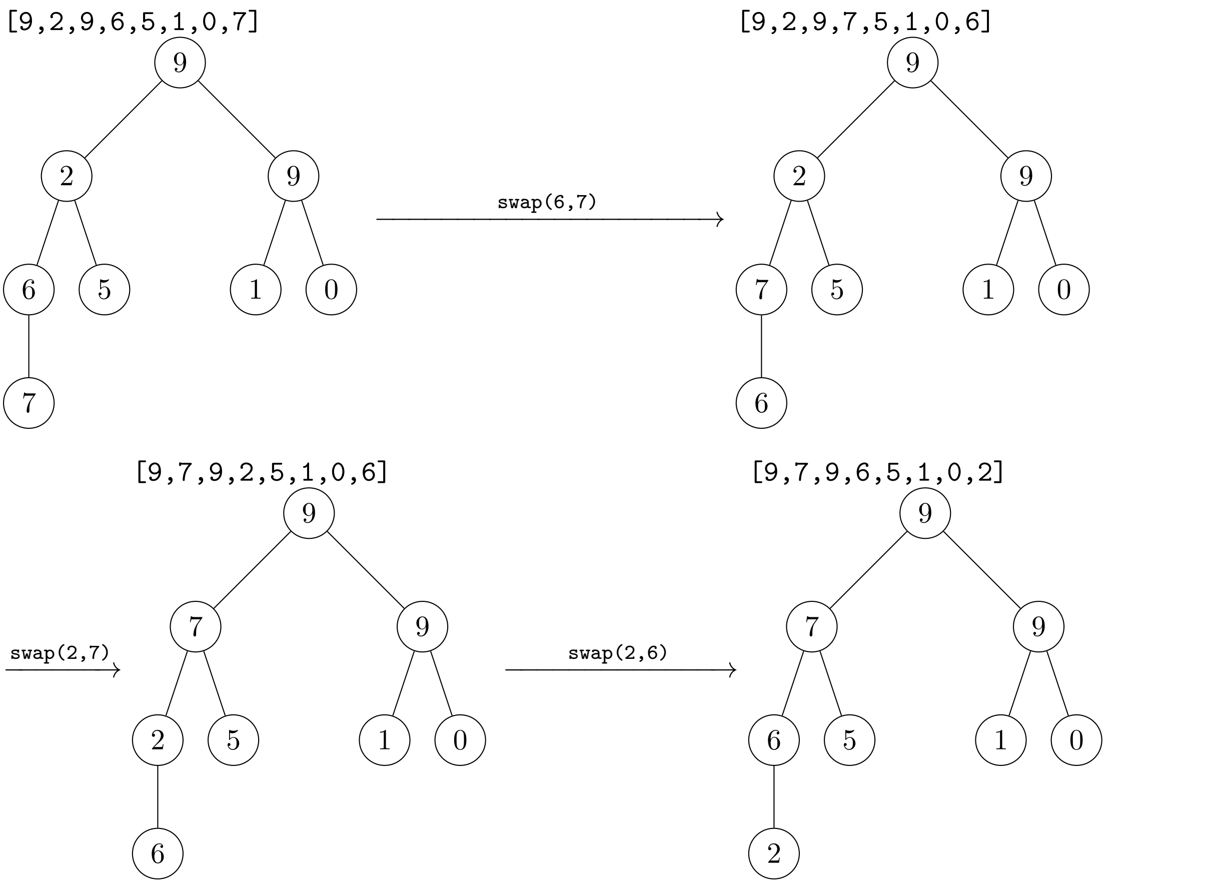
Aufgabe 1

```
[9,0,6,2,5,1,4,8]-> [0,9,6,2,5,1,4,8]-> [0,9,2,6,5,1,4,8]-> [0,2,6,9,5,1,4,8]
->[0,2,6,9,1,5,4,8]-> [0,2,6,9,1,5,4,8]-> [0,2,6,9,1,4,5,8]-> [0,1,2,4,5,6,8,9]
```

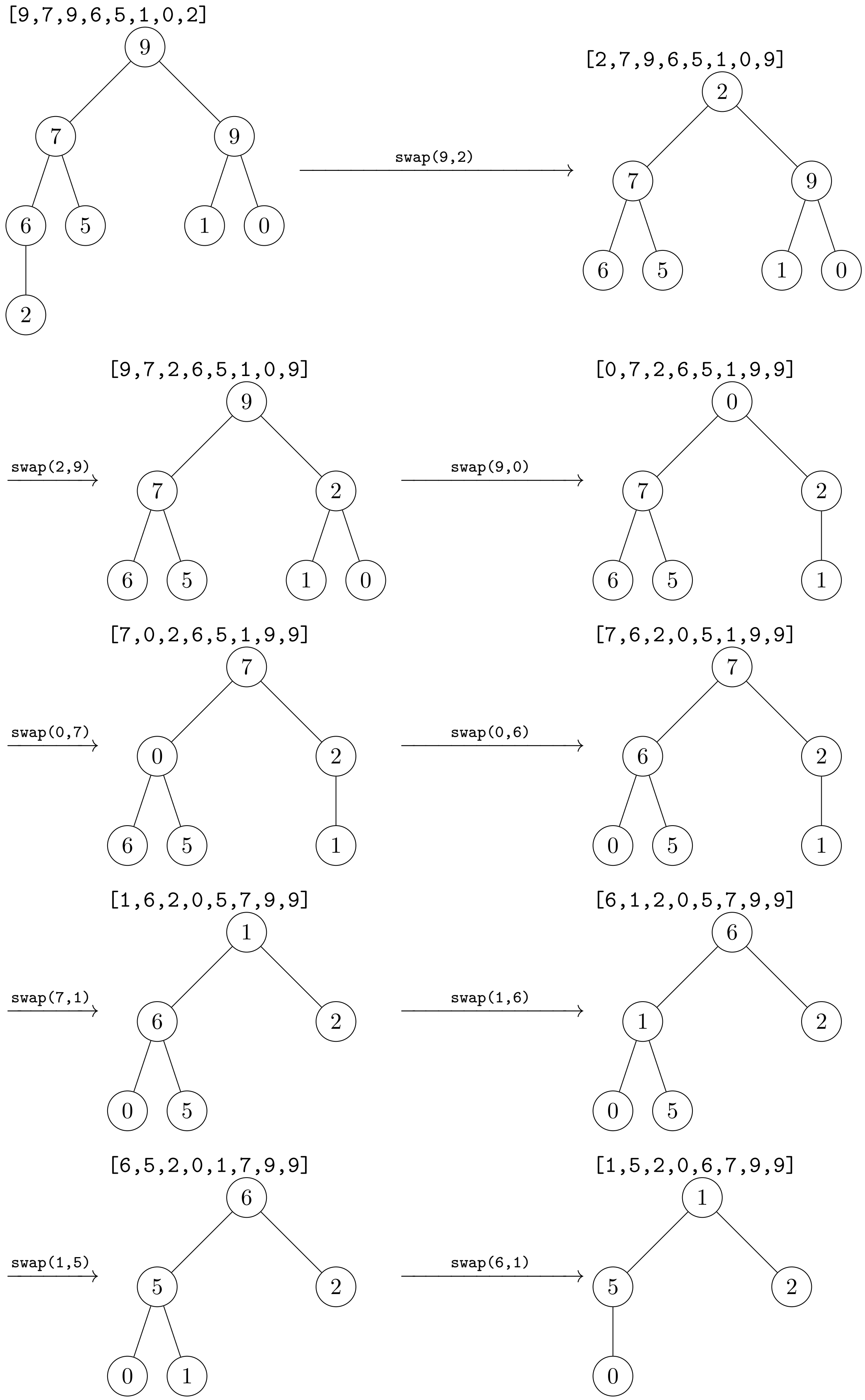
Aufgabe 2

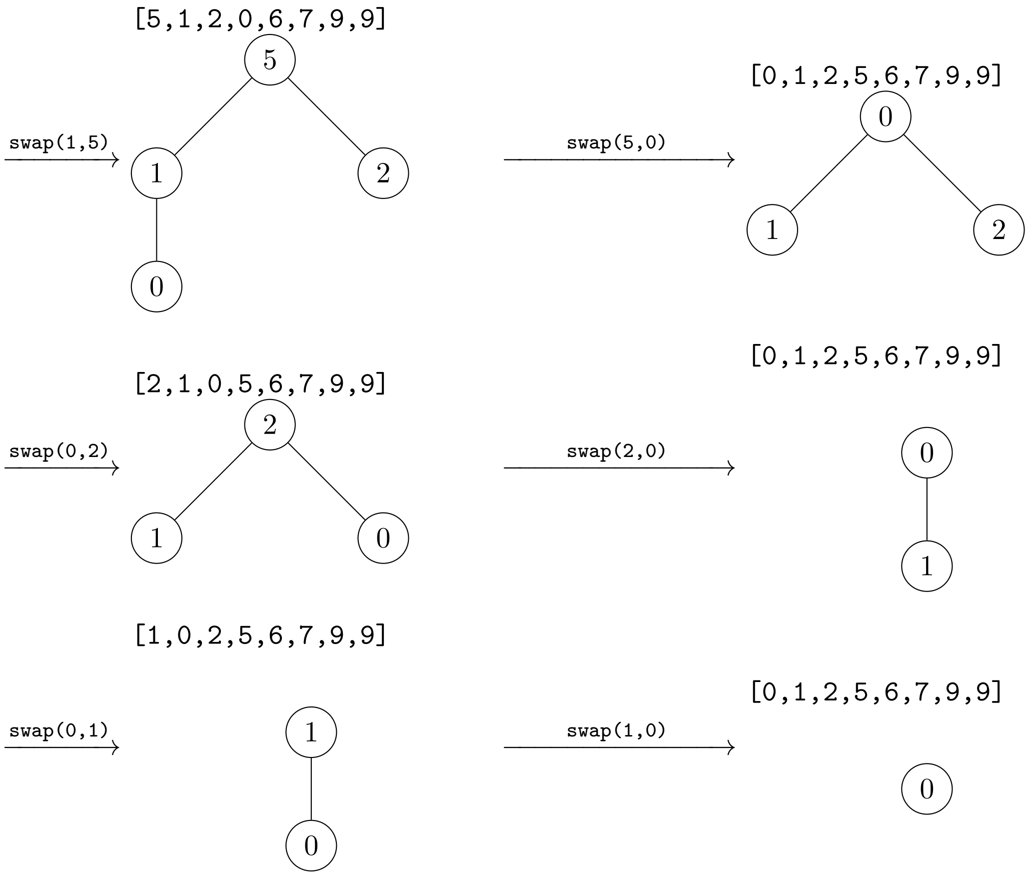
a)

Buildheap



Heapsort


[9,7,9,6,5,1,0,2]
swap(9,2)
[2,7,9,6,5,1,0,9]
swap(2,9)
[9,7,2,6,5,1,0,9]
swap(9,0)
[0,7,2,6,5,1,9,9]
swap(0,7)
[7,0,2,6,5,1,9,9]
swap(0,6)
[7,6,2,0,5,1,9,9]
swap(7,1)
[1,6,2,0,5,7,9,9]
swap(1,6)
[6,1,2,0,5,7,9,9]
swap(1,5)
[6,5,2,0,1,7,9,9]
swap(6,1)
[1,5,2,0,6,7,9,9]



b) Wir zählen analog zur Vorlesung die Anzahl der Vergleiche.

Der Best-case für `heapify` tritt ein wenn der Knoten schon die Wurzel eines Heaps ist. In diesem Fall muss der Algorithmus nur 1 Vergleich pro Kind durchführen, da die Annahme ist, dass alle Kinder ebenfalls schon Wurzeln von Heaps sind. Sei k die Anzahl der Kinder des Knoten, dann sind die Anzahl der benötigten Vergleiche $f_k(n) = k$, also unabhängig von der Eingabelänge n. Da k fest ist, haben wir $B(n) \in \Theta(f_k(n)) = \Theta(1)$. Eingaben hierfür wären `heapify([0], 1, 0)`, `heapify([1,0], 2, 0)` und `heapify([2,1,0], 3, 0)`.

Der Best-case für `buildHeap` tritt ein, wenn das ganze Array schon ein Heap ist. Hat der letzte innere Knoten nur ein Kind, können wir uns einen Vergleich sparen. In diesem Fall müssen wir also $f(n) = |2\lfloor \frac{n}{2} \rfloor - 1|$ Vergleiche durchführen. Denn ein Heap hat $\lfloor \frac{n}{2} \rfloor$ innere Knoten, welche alle bis auf den letzten mit ihren beiden Kindern verglichen werden müssen. Die Betragsstriche sind für den Fall $n = 1$, in dem 0 Vergleiche benötigt werden. Wir haben also $B(n) \in \Theta(f(n)) = \Theta(n)$. Eine Eingabe hierfür wäre z.B. `buildHeap([5,4,3,2,1,0])`.

Der Best-case für `heapSort` ist etwas spezieller. Wenn das Eingabearray nur die gleichen Werte enthält, so ist nach jeder `swap`-Operation mit der Wurzel und dem letzen Element der Heap immernoch ein Heap, da das letzte Element dann eben größer-gleich allen anderen ist. Dies spart viele Vergleichen mit `heapify`. Wir setzen den Best-case also auf ein Eingabearray mit nur den gleichen Werten, in dem der letzte innere Knoten wieder nur ein Kind hat. Im `for`-Loop wird das letzte Element mit dem ersten geswapt, was in unserem Best-case keinen Unterschied macht. Dann wird `heapify` aufgerufen und benötigt jedes mal k Vergleiche wobei k die Anzahl der Kinder der Wurzel ist. Insgesamt haben wir `buildHeap` $+ n$ mal (`for`-Loop) `heapify`, wobei `heapify` in den letzten beiden Durchläufen nur noch jeweil 1 und 0 Vergleiche benötigt. Also $f(n) = \left|2\left\lfloor \frac{n}{2} \right\rfloor - 1\right| + 2n - 3$ und damit $B(n) \in \Theta(f(n)) = \Theta(n)$. Eine Eingabe hierfür wäre z.B. `heapSort([0,0,0,0,0,0])`.

Aufgabe 3

[6,4,4,9,3,2,7,5] $\xrightarrow{\texttt{pivot=5}}$ [2,4,4,3,5,9,6,7] $\xrightarrow{\texttt{pivot=3}}$ [2,3,4,4,5,9,6,7]

$\xrightarrow{\texttt{pivot=2}}$ [2,3,4,4,5,9,6,7] $\xrightarrow{\texttt{pivot=4}}$ [2,3,4,4,5,9,6,7] $\xrightarrow{\texttt{pivot=7}}$ [2,3,4,4,5,6,7,9]

$\xrightarrow{\texttt{pivot=6}}$ [2,3,4,4,5,6,7,9] $\xrightarrow{\texttt{pivot=9}}$ [2,3,4,4,5,6,7,9]

Aufgabe 4

Offensichtlich tut der folgende Algorithmus das gewünschte:

```
stableSort(Element E[]) {
    int n = E.length();
    for (int i = 0; i < n; ++i)
        E[i].key = E[i].key*n + i

    sort(E);

    for (int i = 0; i < n; ++i)
        E[i].key = floor(E[i].key / n);
}
```

Aufgabe 5

a)
Insertionsort ist hier gut, da Insertionsort gut für kleine Arrays ist, da bei kleinen Eingabelängen ein Worst-case von n^2 nicht problematisch ist.

b)
Hier empfiehlt es sich, Counting-Sort zu verwenden. Der Algorithmus hat eben die gegebene Vorraussetzung, dass nur Schlüssel zwischen 0 und einem festen k existieren (hier $k = 4$). Dafür kann er unter dieser Vorraussetzung die Eingabe der Länge n in $\mathcal{O}(n + k)$ sortieren. Best-, Worst-, und Average-case machen hierbei keinen Unterschied. Dies ist deutlich schneller als alle anderen Sortierverfahren die zur Auswahl stehen.

c)
Für fast sortierte arrays ist Insertion-sort eine gute Wahl. Im Best-case (einem komplett sortierten Array) ist die Laufzeit linear. Dies ist ähnlich bei Bubblesort. Jedoch ist Insertionsort nochmal besser da in diesem Szenario alle Elemente sehr wahrscheinlich nur ein paar Plätze (oder auch keine) verschoben werden müssen. Insertionsort muss dann nur diese paar Plätze absuchen um die richtige Stelle zu finden, während Bubblesort das ganze Array durchläuft.

d)
Die Parallelisierbarkeit von Divide-and-Conquer Algorithmen spricht hier für Quick- sowie Merge-sort, jedoch ist die Arbeit vom Partitionieren deutlich besser zu teilen, da beim mergen selbst (was dem Großteil des Sortierverfahrens entspricht) nicht mehr parallel gearbeitet werden kann, während dass zusammenfügen der einzelnen Partitionen trivial ist. Also empfiehlt sich Quicksort.