

## Hausaufgabe 6

---

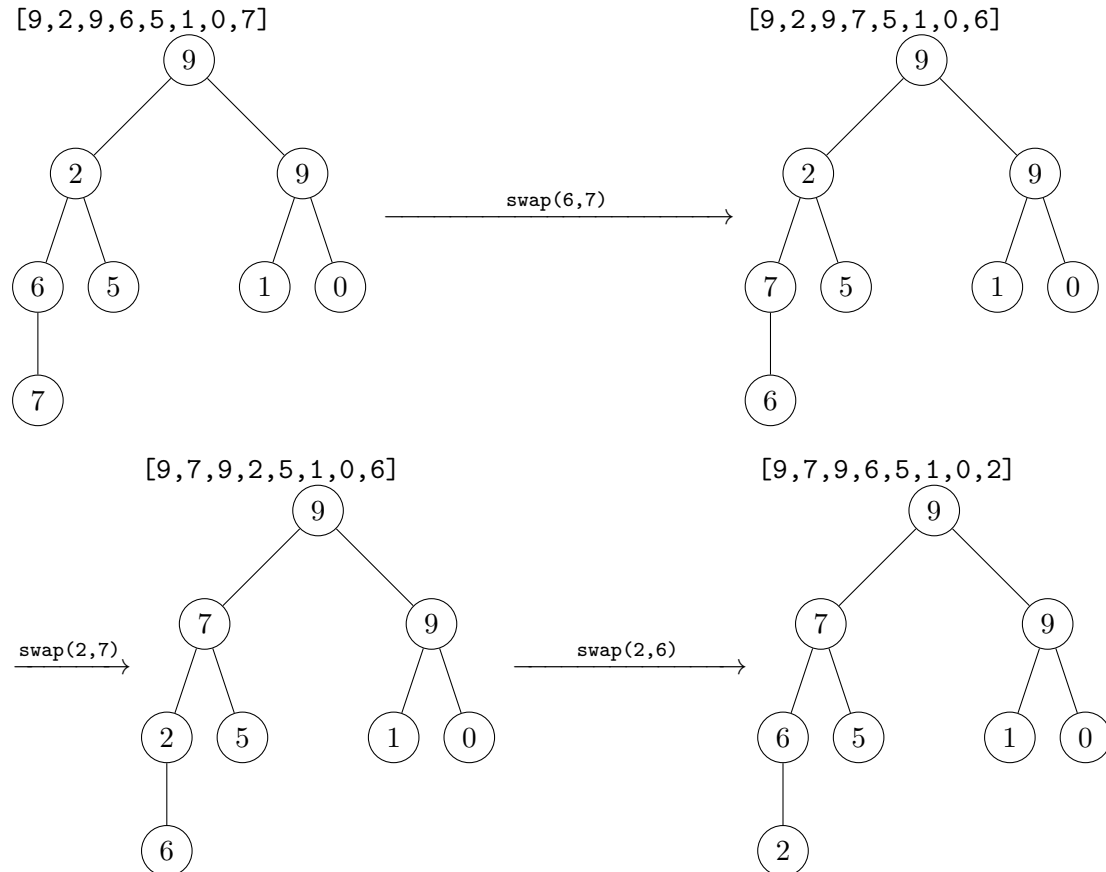
### Aufgabe 1

[9,0,6,2,5,1,4,8]  
[0,9,2,6,1,5,4,8]  
[0,2,6,9,1,4,5,8]  
[0,1,2,4,5,6,8,9]

### Aufgabe 2

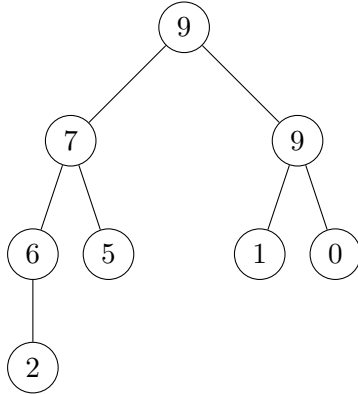
a)

#### Buildheap



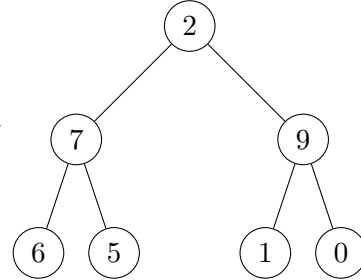
# Heapsort

[9,7,9,6,5,1,0,2]

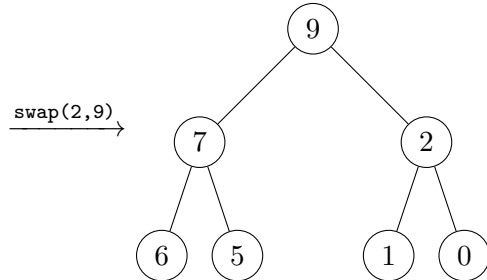


swap(9,2)

[2,7,9,6,5,1,0,9]



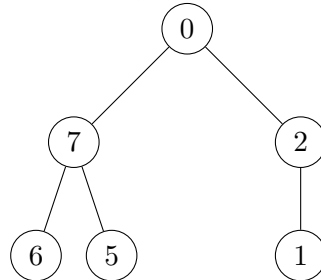
[9,7,2,6,5,1,0,9]



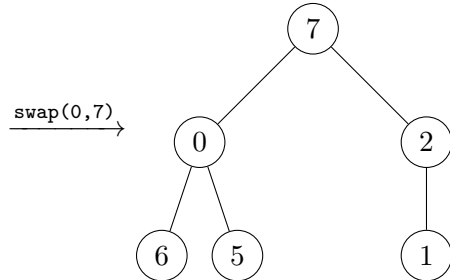
swap(2,9)

swap(9,0)

[0,7,2,6,5,1,9,9]



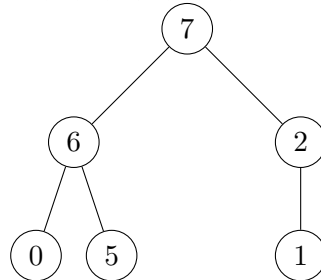
[7,0,2,6,5,1,9,9]



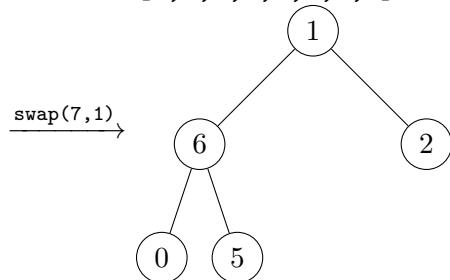
swap(0,7)

swap(0,6)

[7,6,2,0,5,1,9,9]



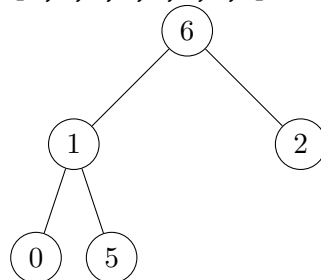
[1,6,2,0,5,7,9,9]



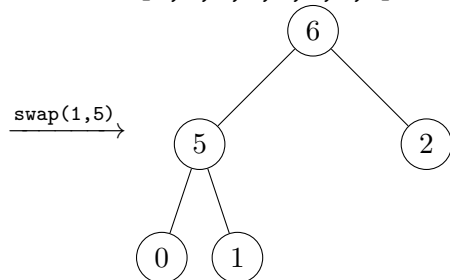
swap(7,1)

swap(1,6)

[6,1,2,0,5,7,9,9]



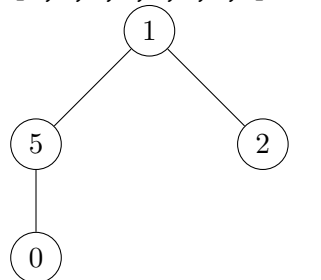
[6,5,2,0,1,7,9,9]

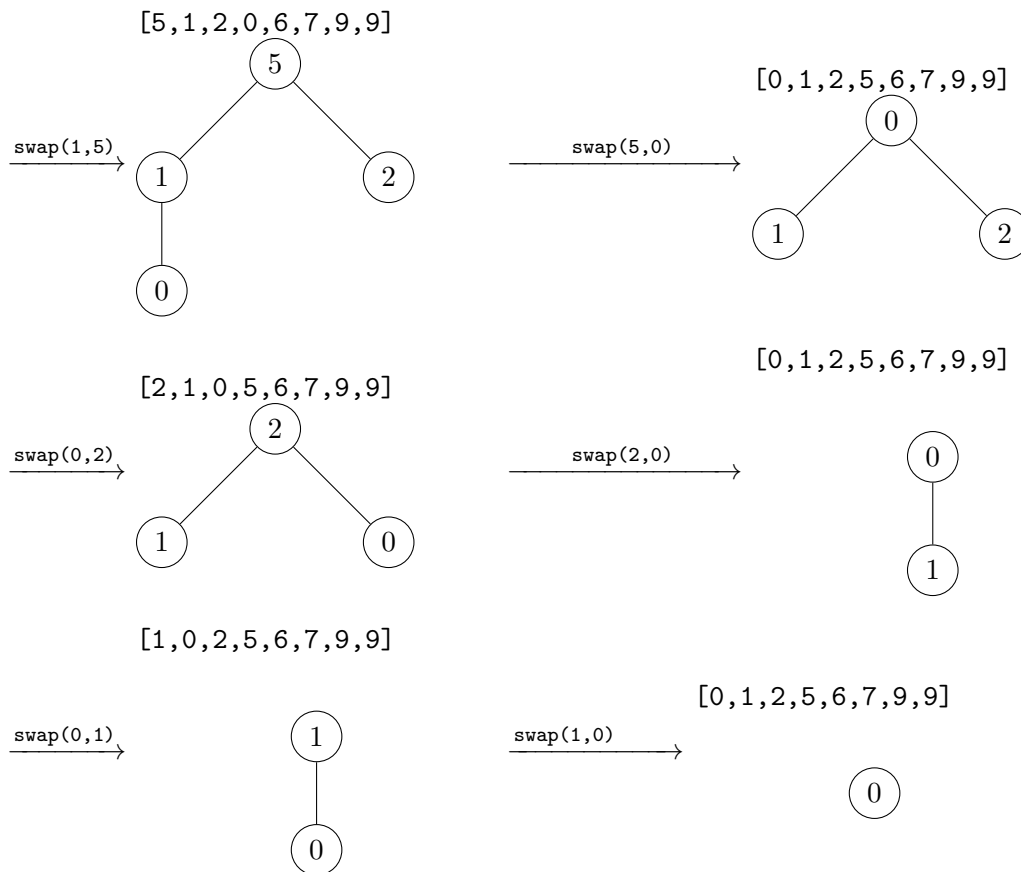


swap(1,5)

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  $f(n) = \lfloor 2 \lfloor \frac{n}{2} \rfloor - 1 \rfloor$  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 letzten 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 gewapt, 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 jeweils 1 und 0 Vergleiche benötigt. Also  $f(n) = \lfloor 2 \lfloor \frac{n}{2} \rfloor - 1 \rfloor + 2n - 3$  und damit  $B(n) \in \Theta(f(n)) = \Theta(n)$ . Eine Eingabe hierfür wäre z.B. **heapSort**([0,0,0,0,0,0,0]).

### Aufgabe 3

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

### Aufgabe 4

TODO

## Aufgabe 5

a) Insertionsort??

b)

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

c)

Für fast sortierte arrays ist Insertion-sort eine gute Wahl. Im best case (einem komplett sortierten Arrays) ist die Laufzeit linear. Dies ist ähnlich bei Bubblesort. Jedoch ist Insertion sort 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)

Es macht Sinn, diese Aufgabe mit Mergesort anzugehen, da die Assistenten nach den ersten 3 splits die 8 fast gleichgroßen Stapel unter sich aufteilen können und sofort Mergesort anwenden können. Dadurch können die 8 Assistenten parallel arbeiten, was bei anderen Sortiervverfahren nicht immer möglich ist. Sobald jeder der 8 fertig ist, mergen sie ihre sortierten Klausuren einfach.