HeapSort

HeapSort basiert, wie der Name bereits impliziert auf der Datenstruktur des Heaps. Man unterscheidet Min-Heaps und Max-Heaps.
In einem Max-Heap steht das größte Element in der Wurzel, bei einem Min-Heap steht dann logischerweise das kleinste Element in der Wurzel.

Definition eines Max-Heaps:
Eine Folge F= $k_1, k_2, ..., k_N$ von Schlüsseln nennen wir einen Heap, wenn $k_i \leq k_{\lfloor i/2 \rfloor}$ für $2 \leq i \leq N$ gilt. Anders ausgedrückt $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$, sofern $2i \leq N$ bzw. $2i + 1 \leq N$.

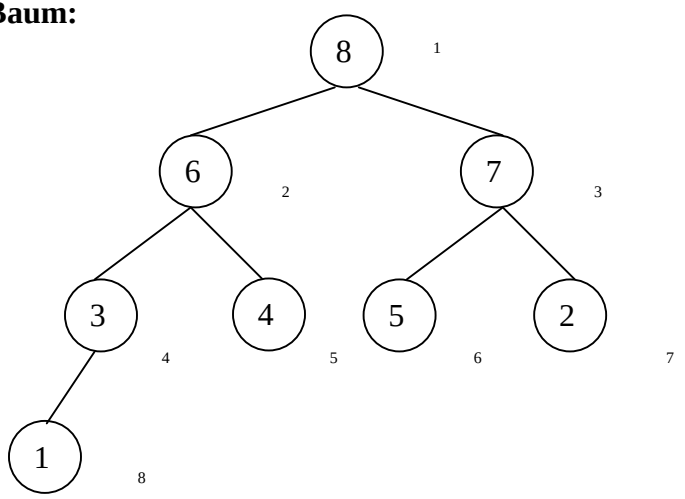
Aus der Definition geht hervor, dass die Element 2i und 2i+1 falls vorhanden, die Kinder des Knoten i sind. Dies kann man anhand einer Baumsdarstellung gut erkennen.

Bsp.: Betrachten wir als Beispiel die Folge F = 8,6,7,3,4,5,2,1.
Programm intern wird die Folge in einem Array gespeichert, um die Vorgehensweise von HeapSort aber auch die eigentliche Struktur zu erkennen, bietet sich eine Baumdarstellung des Heaps an.

Folge F als Array:

8	6	7	3	4	5	2	1
i	2i	2i+1					

Folge F als Baum:

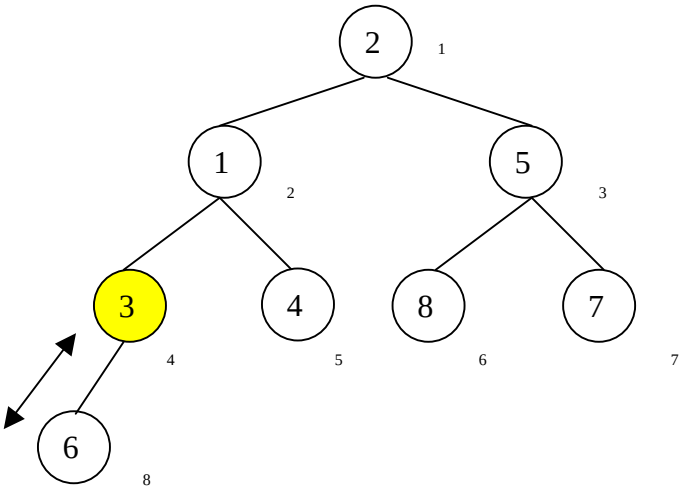


Wie man erkennt genügt die Folge F der Heap-Bedingung, da alle Väter größer als Ihre beiden Kinder sind.

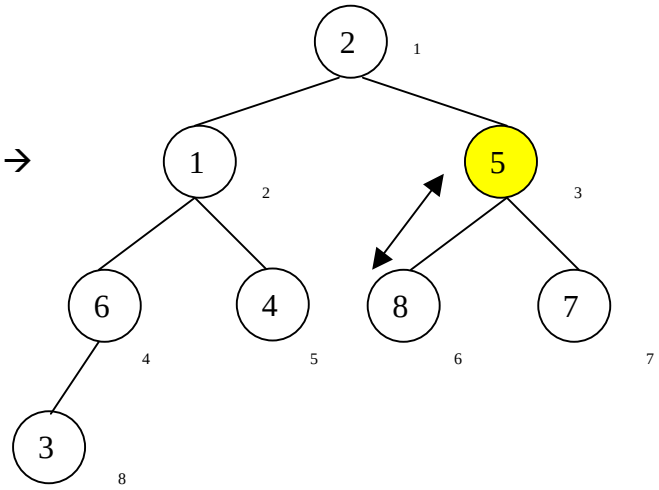
Das Aufbauen eines Heaps (Aufbauphase):

In der Regel entsprechen beliebige Folgen nicht der Heap-Bedingung. Daher muss man sie zunächst in einen Heap umwandeln um HeapSort auf sie anwenden zu können.
Dazu bedient man sich folgender Methode:
Eine Folge F= $k_1, k_2, ..., k_N$ von N Schlüsseln wird in einen Heap umgewandelt, indem die Schlüssel $k_{\lfloor n/2 \rfloor}, k_{\lfloor n/2 \rfloor - 1}, ..., k_1$ (in dieser Reihenfolge) in F versickern.

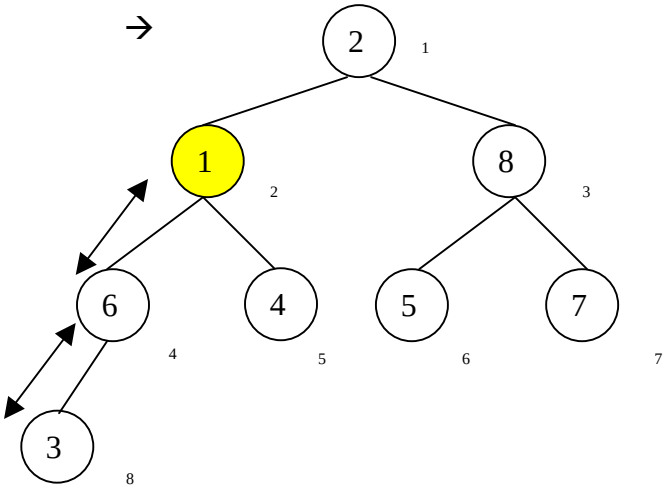
Betrachten wir dazu wieder ein Beispiel:
Sei F=2,1,5,3,4,8,7,6 die zu sortierende Folge. Die Umwandlung der Folge in einen Heap vollzieht sich dann wie in folgender Zeichnung dargestellt.
N=8 → $k_{\lfloor n/2 \rfloor} = 8/2=4$



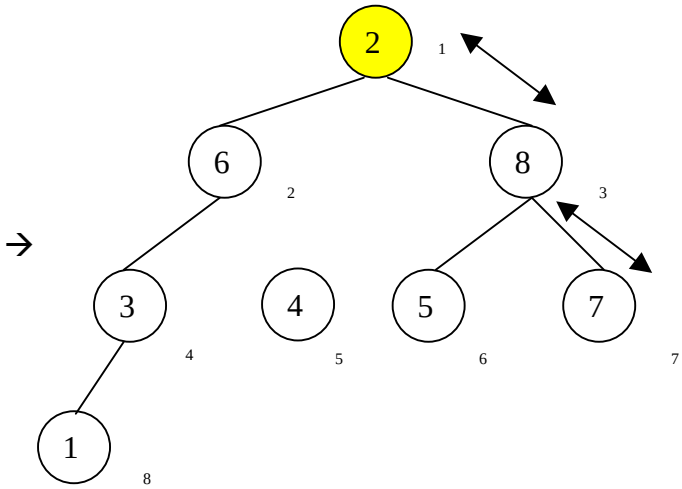
$k_{\lfloor n/2 \rfloor} = 4 \rightarrow k_4 = 3$
3<6 → Tausche 3, 6



$k_{\lfloor n/2 \rfloor - 1} = 3 \rightarrow k_3 = 5$
8>7 und 5<8 → Tausche 5,8

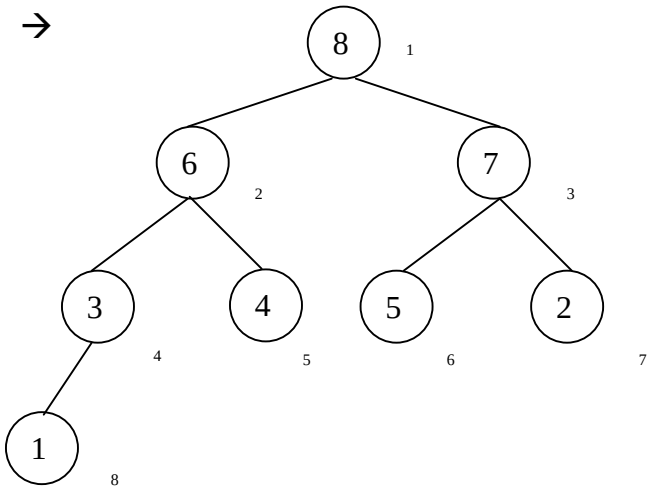


$k_{\lfloor n/2 \rfloor - 1} = 2 \rightarrow k_2 = 1$
6>4 und 1<6 → Tausche 1,6; 1<3 → Tausche 1,3



$k_{\lfloor n/2 \rfloor - 1} = 1 \rightarrow k_1 = 2$
8>6 und 2<8 → Tausche 2,8; 7>5 und 2<7 → Tausche 2,7

→



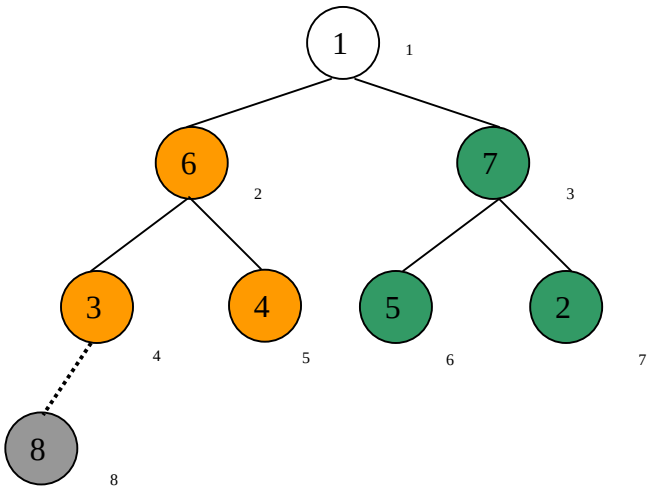
Damit ist die Heap-Eigenschaft hergestellt. Die Folge F sieht in Array-Darstellung nun folgendermaßen aus:

F= 8,6,7,3,4,5,2,1.

Das Sortieren im Heap (Selektionsphase)

Ist die Struktur des Heaps erst mal hergestellt, so ist das sortieren einfach. Da im Max-Heap das Maximum der Folge an der Wurzel steht, können wir das erste Element schon sofort ablesen. Um die weiteren Elemente zu bestimmen, entfernen wir das Maximum aus der Wurzel und tauschen es mit dem im Array am weitesten rechtsstehenden, nicht sortiertem Element aus (Im Baum also dem auf der jeweilig letzen Ebene am weitesten rechts, also das Element mit dem größten Index).

Durch das Vertauschen der Schlüssel ist die Heap-Eigenschaft möglicherweise zerstört. Hier liegt auch der schwierige Teil, nämlich das Wiederherstellen des Heaps. Um erneut einen Heap herzustellen nutzen wir die Tatsache aus, dass nach dem Tauschen/Entfernen der Wurzel noch zwei TeilHeaps vorliegen.

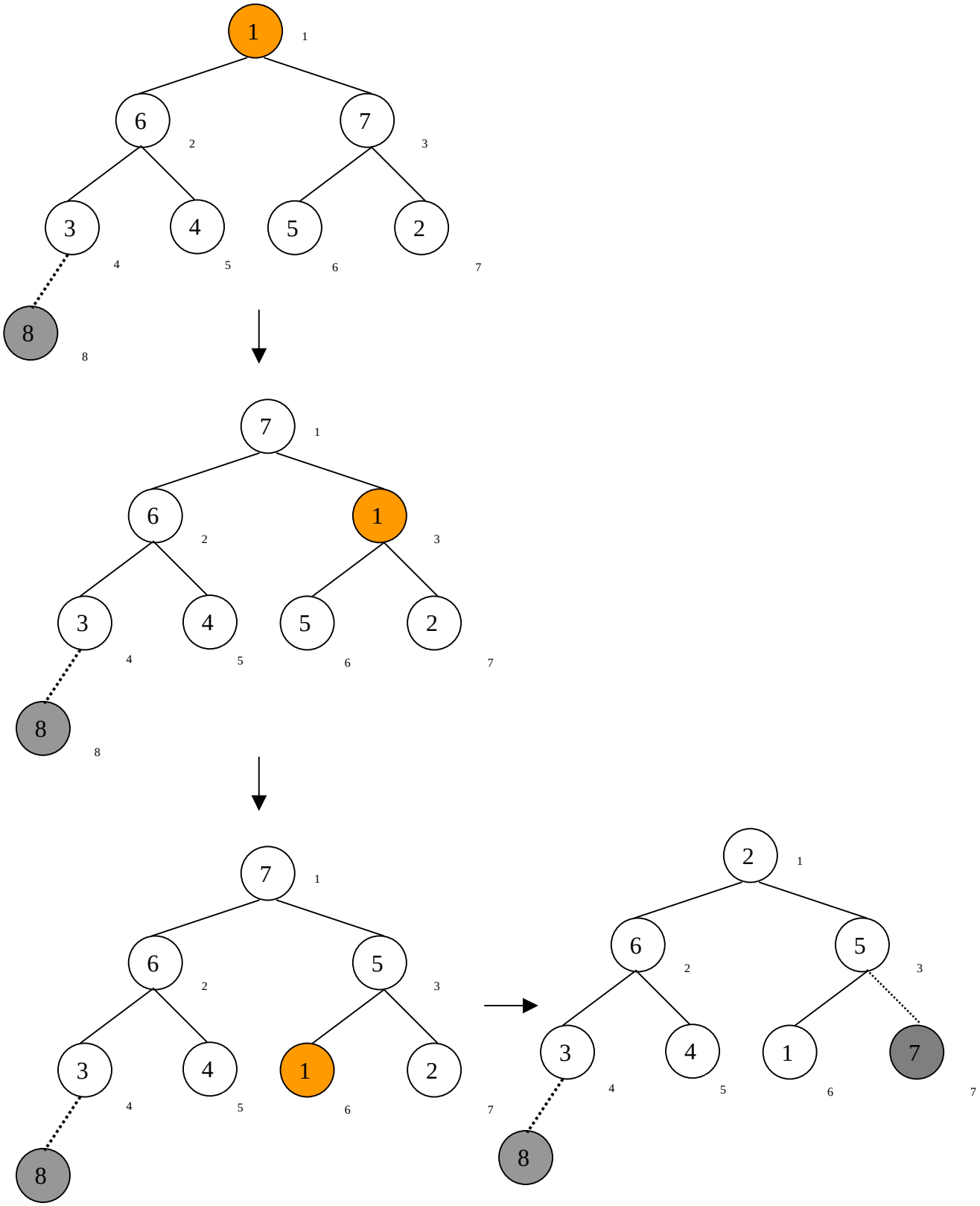


Teilheaps

Arraydarstellung:

1	6	7	3	4	5	2	8

Um die Heap-Eigenschaft wiederherzustellen, lassen wir die neue Wurzel, k_1, im Heap nach unten versickern, in dem wir ihn solange immer wieder mit dem größeren seiner beiden Söhne vertauschen, bis beide Söhne kleiner sind oder der Schlüssel unten angekommen ist.



Damit ist die Heap-Bedingung wieder hergestellt. Es wird wieder die Wurzel vertauscht, und man lässt die Wurzel wieder absinken.

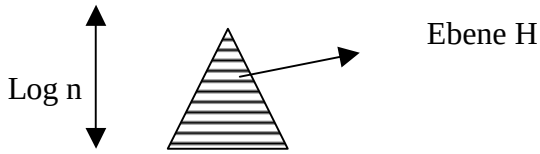
ArrayDarstellung:

2	6	5	3	4	1	7	8

Analyse von HeapSort

Die Aufbauphase des Heaps hat Laufzeit O(n), also lineare Laufzeit.

Warum:



$O(\Sigma_{h=1,\log n}\ h * \text{Anzahl der Knoten auf Höhe h})$; Anzahl der Knoten auf Höhe $h \leq n/2^h$
$= O(\Sigma_{h=1,\log n}\ h * n/2^h)$
$\leq O(n*\Sigma_{h=0,\infty}\ h/2^h) = O(n)$

Die Selektionsphase des Heaps hat die Laufzeit O(n log n).

Warum:

Für r= n,...,n-1,...,1:

Lasse Wurzel s[1] sinken. Dies hat jeweils die Kosten O(Höhe(1) im Heap S[1...r]) = O(log r)

→ Gesamtkosten= $O(\Sigma_{r=1,n} \log r) \leq O(\Sigma_{r=1,n} \log n) = O(n \log n)$

Ein vollständiger binärer Baum mit *n* Knoten hat die Tiefe $d \leq \log(n)$. Die Prozedur *downheap* benötigt daher höchstens log(*n*) Schritte. Die Prozedur *buildheap* ruft für jeden Knoten einmal *downheap* auf, benötigt also insgesamt höchstens $n \cdot \log(n)$ Schritte. Prozedur *heapsort* schließlich ruft einmal *buildheap* auf und ruft anschließend für jeden Knoten einmal *downheap* auf, benötigt also höchstens $2 \cdot n \cdot \log(n)$ Schritte.

FAZIT:

- HeapSort liegt sowohl im Worstcase als auch im AverageCase in n log n.
- Der Suchalgorithmus ist ein In-Place Algorithmus.
- Vorsortierung schadet nicht, kann sogar ausgenutzt werden.
- Iterativer Algorithmus