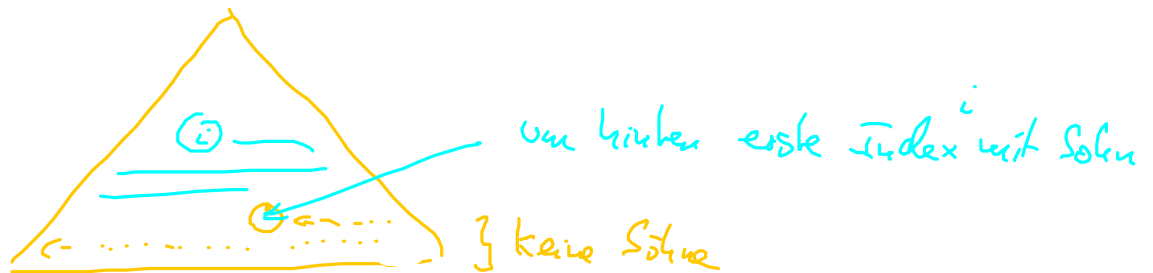


Heapsort

Wiederholung heap, heapify

Nutzen von heapify zur Herstellung der
Heap Eigenschaft

Array von links nach rechts durchlaufen
und $\text{heapify}(\text{vec}, i, n-1)$ aufrufen



Wenn $\text{heapify}(\text{vec}, i, n-1)$ aufgerufen wird, gilt
bereits die Heapeigenschaft in $i+1, i+2, \dots, n-1$
deswegen heapify anwendbar und stellt Heapeigenschaft
in $i, i+1, \dots, n-1$ her

```
for (int i = n/2 - 1; i >= 0; i--) {  
    heapify(vec, i, n-1);  
}
```

Nach dem Herstellen der Heap Eigenschaft kann jetzt einfach

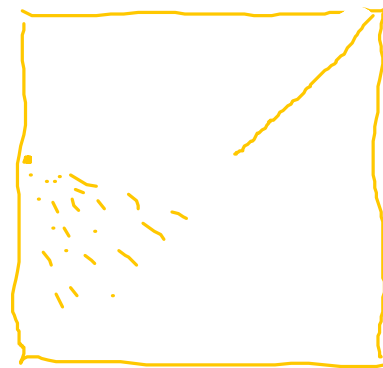
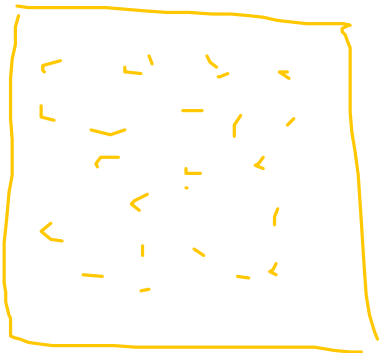
sortiert werden, indem man successive das größte Element aus dem Heap entfernt und die Heapeigenschaft wieder herstellt:

- letzte Komponente im Heap an erste Stelle setzen
- `heapify (vec, 0, bottom)` aufrufen
↑
momentanes letztes Element

Java Code siehe Prog. zur VL

Alle Tauschoperationen werden direkt im Array vorgenommen

Visualisierung von Heapsort:



Die Analyse von Heapsort

wir müssen zeigen: Heapsort (create heap) in $O(n)$

Zugriff auf Größe in $O(1)$ ✓

Entfernen (Wiederherstellen Heap Eig.) in $O(\log n)$ ✓

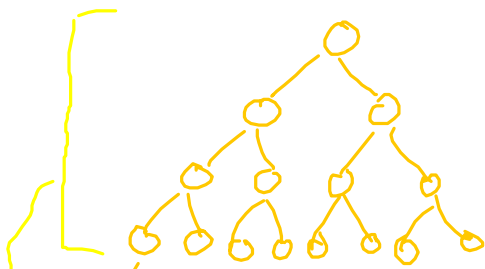
Wichtige Einsicht:

Heap ist ein voller binärer Baum

↑

alle Schichten voll bis auf letzte

ergibt Relation zwischen n und Höhe des Baums
↑
 h



← 1 Knoten in Schicht 0

2 1

2^i Knoten in Schicht i

← zwischen 1 und 2^h Knoten auf letzter Schicht, diese liegt in Höhe h

alle voll

$$2^1 + 2^2 + \dots + 2^{h-1}$$

$$= \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

$$\Rightarrow 2^h - 1 + 1 \leq n \leq 2^h - 1 + 2^h$$

\Rightarrow

$$2^h \leq n \leq 2^{h+1} - 1$$

\Rightarrow

$$h \leq \log n$$

Rekursive Aufrufe von heapify erfolgen jeweils auf einer um 1 höheren Schicht

=> maximal $h+1$ rekursive Aufruf

pro Aufruf eine Prüfaktion

größeren Sohn ermitteln

ggf mit Vater tauschen

Für Aufruf von `heapify` insgesamt maximal $h+1$ Prüfaktionen

=> Aufruf `heapify(vec, 0, last)` in Sortierphase

erfordert maximal $h+1 \leq (\log n + 1)$ Prüfaktionen

pro Prüfaktion ≤ 2 Vergleiche

3 Zuweisungen

=> Wiederherstellen des Heap Eigenschaft $5 \cdot (\log n + 1)$

$\in O(\log n)$

zum Heapaufbau



`heapify` für jeden Knoten

=> n -Aufwand `heapify`

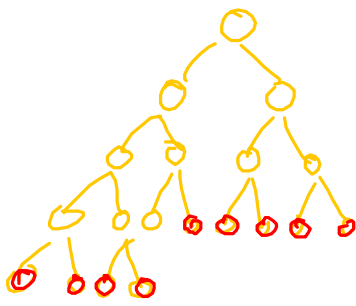
=> $O(n \log n)$

für viele Knoten (die auf Schichten $h-1, h-2, h-3$)

und wenige Prüfaktionen, dies ist aber die große Mehrheit

der Knoten

daher genauer analysieren



\circ max h Prüf
 $1 = h - (h-1)$ max $h-1$ Prüf
 \vdots
 $h-2$ 2
 $h-1$ max 1 Prüfak.
 \leftarrow Schicht h keine Prüf

Summe der Prüfkosten $\leq \sum_{\text{Schichten}} (\# \text{Knoten pro Schicht}) \cdot (\text{max Anzahl Prüfkosten für Knoten})$

$$= \sum_{i=1}^h 2^{h-i} i = 2^h \cdot \sum_{i=1}^h \frac{i}{2^i} \leq 2 \cdot n \in O(n)$$

$\underbrace{\sum_{i=1}^{\infty} \frac{i}{2^i} = 2}_{\leq n}$

$2n$ Prüfkosten $\hat{=}$ 3 Zuweisungen, 2 Vergleiche

\Rightarrow Herstellen Heapeigenschaft in $O(n)$

Kapitel 11 Untere Komplexitätsdranken
für das Sortieren

gelten für ALLE Sortieralgs (einer bestimmten Klasse)

also auch für solche, die noch gar nicht erfunden

↑ sind

großer Unterschied zu Listes: nur Analyse bestimmter Algorithmen

brauchen ein Modell für alle möglichen Folgen einer Klasse

deterministische Folgen, die auf paarweisen
Vergleichen von Schlüsselwerten

keine "Randomisierung" im Algo

d.h. bei derselben Eingabe entsteht
bei jedem Lauf dieselbe Folge
von Vergleichen

alle unsere Listen
Folgen

(nutzen kein Fifo
über Schlüsselwerte
aus)

Gegenbeispiel: Quicksort mit zufälliger
Wahl des Pivot elementes