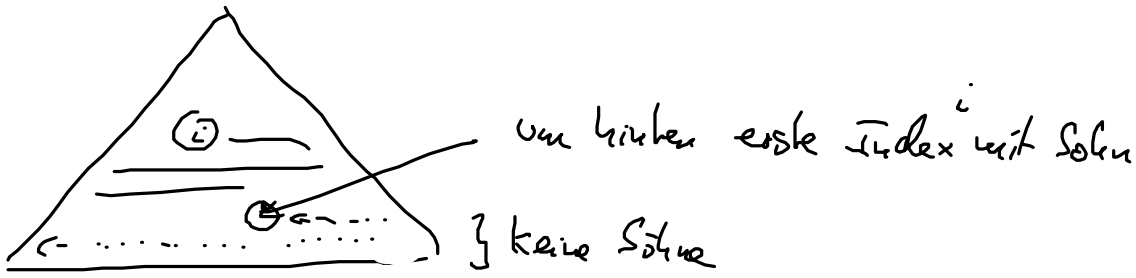
Heapsort

Wiederholung heap, heapify

Nutzen von heapify zur Herstellung der Heapeigenschaft

Array von hinten nach vorn durchlaufen und heapify(vec, i, n-1) aufrufen



Wenn heapify(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

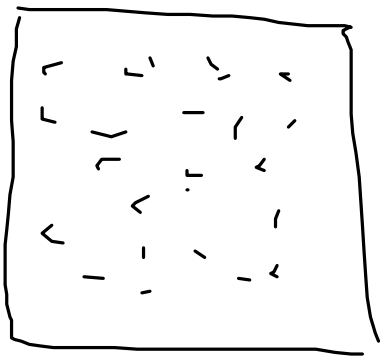
sortiert werden, indem man successive das größte Element aus dem Heap entfernt und die Heapeigenschaft wieder herstellt:

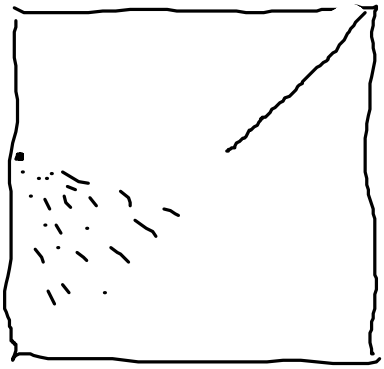
- letzte Komponente im Heap an erste Stelle setzen
- heapify (vec, 0, bottom) aufrufen
  ↑ momentanes letztes Element

Java Code siehe Prog. zur VL

Alle Tauschoperationen werden direkt im Array vorgenommen

Visualisierung von Heapsort:





Die Analyse von Heapsort

müssen zeigen: Heapaufbau (createHeap) in $O(n)$

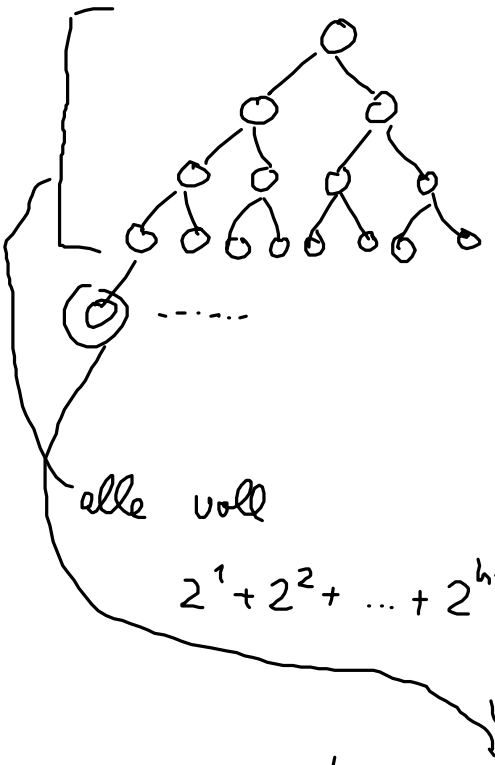
Zugriff auf Größte in $O(1)$ ✓

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 Baumes h



← 1 Knoten in Schicht 0

2 ---- 1

2^i Knoten in Schicht i

← zwischen $\underline{1}$ und $\underline{2^h}$ Knoten auf letzter Schicht, diese liegt in Höhe h

alle voll

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

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

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

$$\Rightarrow \quad \boxed{h \leq \log n}$$

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

$\Rightarrow$ maximal $h+1$ rekursive Aufrufe

pro Aufruf eine Präfaktion

größeren Sohn ermitteln

ggf mit Vater tauschen

Für Aufruf von Heapify insgesamt maximal $h+1$ Präfaktionen

$\Rightarrow$ Aufruf heapify (vec, 0, last) in Sortierphase

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

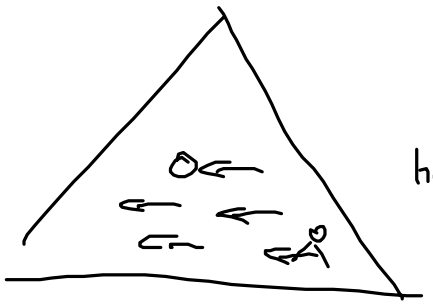
pro Präfaktion ≤ 2 Vergleiche

3 Zuweisungen

$\Rightarrow$ Wiederherstellen der Heap Eigenschaft $5 \cdot (\log n + 1)$

$\in \mathcal{O}(\log n)$

zum Heap aufbauen



heapify für jeden Knoten $\Rightarrow n \cdot$ Aufwand heapify

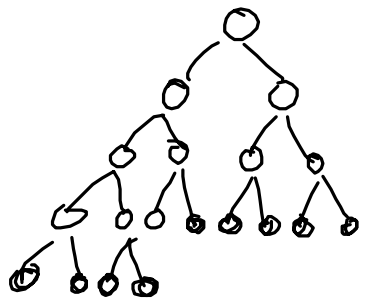
$\Rightarrow \mathcal{O}(n \log n)$

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

nur wenige Präfaktionen, dies ist aber die große Mehrheit

der Knoten

daher genauer analysieren



0 max h Perf
1 = h − (h−1) max h−1 Perf
⋮
h−2 2
keine Perfaktion ← h−1 max 1 Perfak.
← Schicht h keine Perf

Summe der Perfaktionen ≤ $\sum_{\text{Schichten}}$ (# Knoten pro Schicht) · (max Anzahl Perfaktionen für Knoten)

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

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

$2n$ Perfaktionen à 3 Zuweisungen, 2 Vergleiche

⇒ Herstellen Heapeigenschaft in $O(n)$

Kapitel 11 Untere Komplexitätsschranken

für das Sortieren

gelten für ALLE Sortieralgos (einer bestimmten Klasse)

also auch für solche, die noch gar nicht erfunden sind

großer Unterschied zu bisher: nur Analyse bestimmter Algorithmen

Brauchen ein Modell für alle möglichen Algos einer Klasse

deterministische Algos, 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 bisherigen Algos

(nutzen kein Info über Schlüsselwerte aus)

Gegenbeispiel: Quicksort weil zufälliges Wahl des Pivotelements