Informatik II

Übung 8

FS 2019

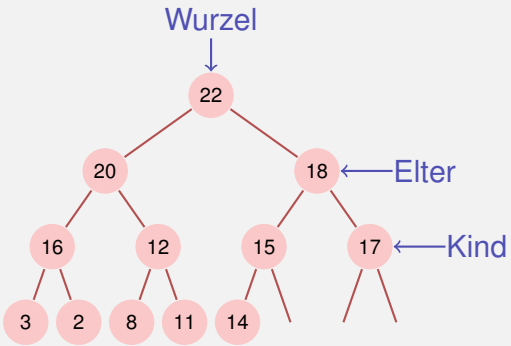
Heutiges Programm

1 Wiederholung Heaps

Max-]Heap[1]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. *Heap-Bedingung:* Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Elternknotens

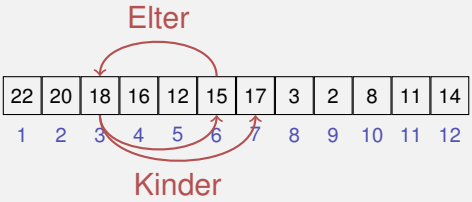


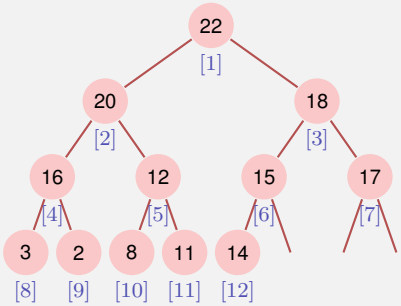
[1] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

Heap und Array

Baum $\rightarrow$ Array:

- Kinder$(i) = \{2i, 2i+1\}$
- Elter$(i) = \lfloor i/2 \rfloor$





Abhängig von Startindex![2]

[2] Für Arrays, die bei 0 beginnen: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

Höhe eines Heaps

Vollständiger binärer Baum der Höhe[3] h hat

$$1 + 2 + 4 + 8 + ... + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

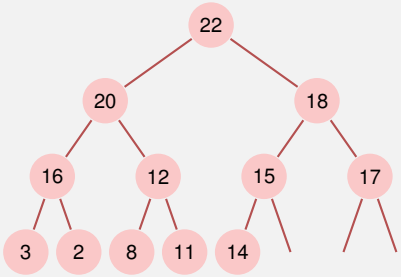
Knoten. Somit gilt für einen Heap der Höhe h:

$$\begin{aligned} & 2^{h-1} - 1 < n \leq 2^h - 1 \\ \Leftrightarrow \quad & 2^{h-1} < n + 1 \leq 2^h \end{aligned}$$

Also insbesondere $h(n) = \lceil \log_2(n+1) \rceil$ und $h(n) \in \Theta(\log n)$.

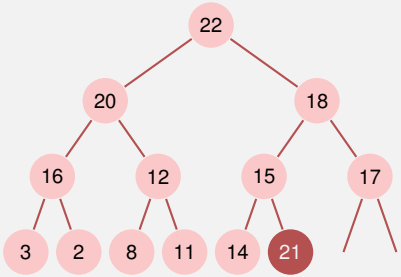
[3] Hier: Anzahl Kanten von der Wurzel zu einem Blatt

Einfügen



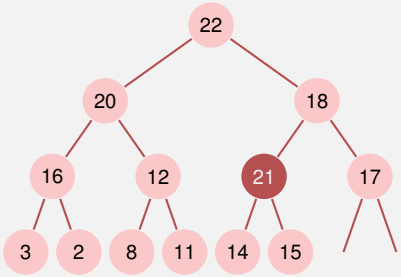
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.



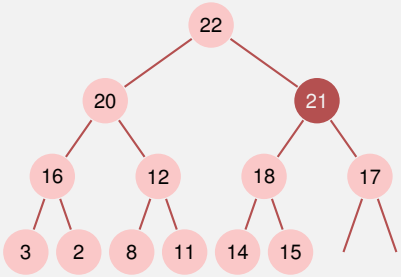
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



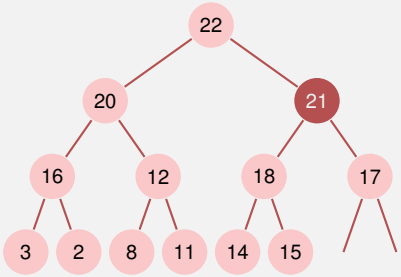
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.

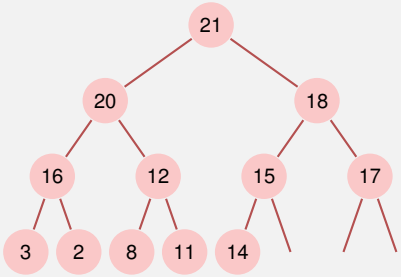


Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$

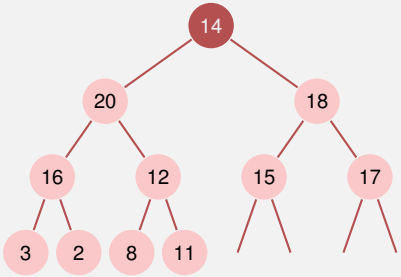


Maximum entfernen



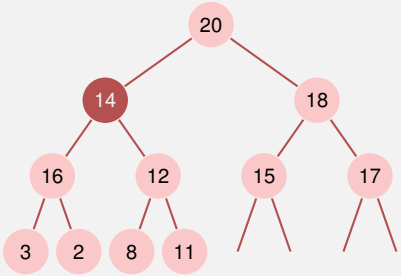
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.



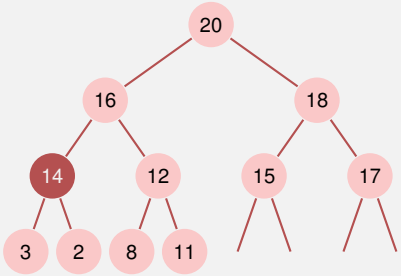
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



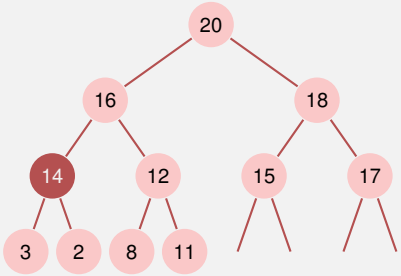
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Algorithmus Versickern(A, i, m)

Input: Array A mit Heapstruktur für die Kinder von i. Letztes Element m.

Output: Array A mit Heapstruktur für i mit letztem Element m.

while $2i \leq m$ **do**
- $j \leftarrow 2i$; // j linkes Kind
- **if** $j < m$ and $A[j] < A[j+1]$ **then**
  - $j \leftarrow j + 1$; // j rechtes Kind mit grösserem Schlüssel
- **if** $A[i] < A[j]$ **then**
  - swap($A[i], A[j]$)
  - $i \leftarrow j$; // weiter versickern
- **else**
  - $i \leftarrow m$; // versickern beendet

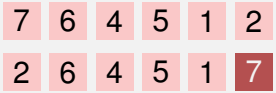
Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

Heap Sortieren

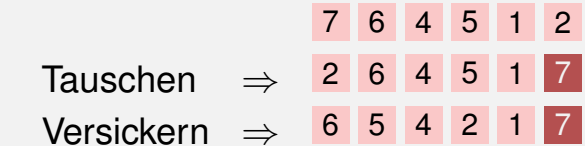
Tauschen $\Rightarrow$



$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

Heap Sortieren



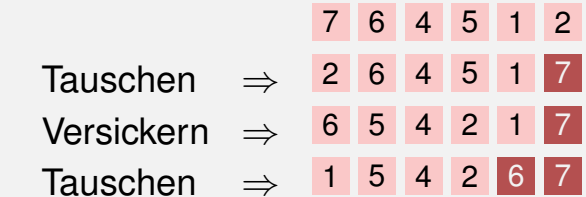
$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n - 1$);
- $n \leftarrow n - 1$

Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

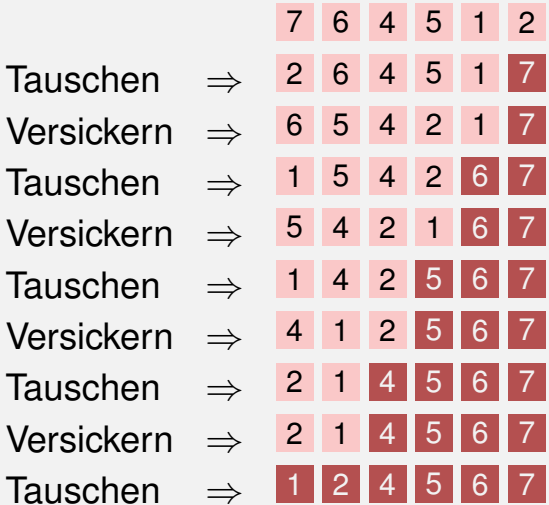
- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$



Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$



Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung:

Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung: Induktion von unten!

Algorithmus HeapSort(A, n)

Input: Array A der Länge n.
Output: A sortiert.

```
// Heap Bauen.
for i ← n/2 downto 1 do
    Versickere(A, i, n);
// Nun ist A ein Heap.
for i ← n downto 2 do
    swap(A[1], A[i])
    Versickere(A, 1, i − 1)
// Nun ist A sortiert.
```

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. $\Rightarrow$ Heap Sortieren kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch $\mathcal{O}(n \log n)$.

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Versickerpfade sind aber im Mittel viel kürzer und es gilt:

$$v(n) \in \mathcal{O}(\mathbf{n})$$

Fragen oder Anregungen?