

Prioritätenschlängen/Halden/Heaps

Prioritätenschlängen

- Einfügen, Löschen, Maximum (bzw. Minimum)
- Wir könnten Binäre Suchbäume benutzen
- Gibt es einfachere Datenstruktur?
- Gibt es effizientere Datenstruktur?
(Ja, aber AVL-Bäume oder andere, die nicht Teil dieser Vorlesung sind)

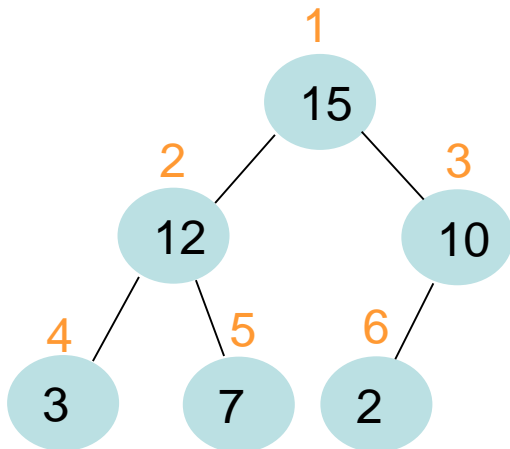
Anwendungen

- Ereignisgesteuerte Simulationen
- Sortieren mit Heapsort

Bäume als Arrays – Datenstruktur

Binäre Halden / Heaps

- Feld $A[1, \dots, \text{length}[A]]$
- Man kann Feld als vollständigen Binärbaum interpretieren
- D.h., alle Ebenen des Baums sind voll bis auf die letzte
- Zwei Attribute: $\text{length}[A]$ und $\text{heap-size}[A]$, $\text{heap-size} \leq \text{length}[A]$



1	2	3	4	5	6
15	12	10	3	7	2

Bäume als Arrays – Datenstruktur

Navigation

- Wurzel ist $A[1]$

Parent(i)

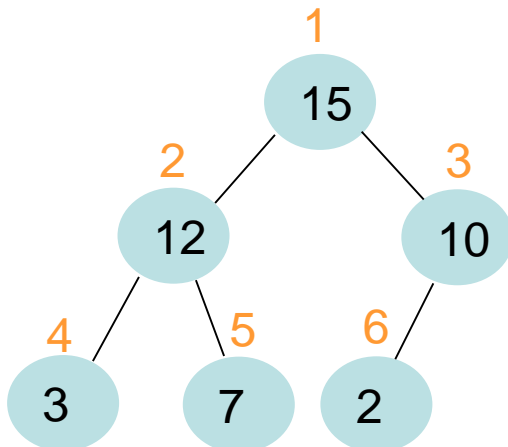
- **return** $\lfloor i/2 \rfloor$

Left(i)

- **return** $2i$

Right(i)

- **return** $2i+1$



1	2	3	4	5	6
15	12	10	3	7	2

Bäume als Arrays – Datenstruktur

Navigation

- Wurzel ist $A[1]$

Parent(i)

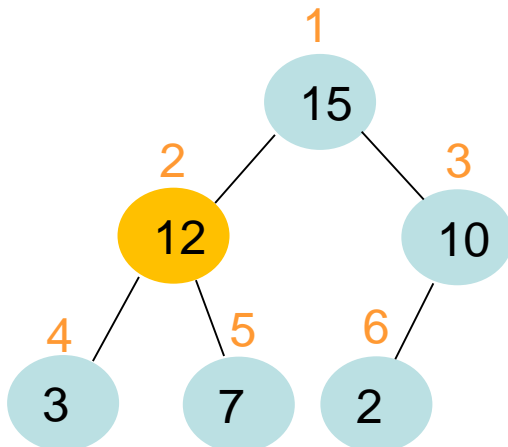
1. **return** $\lfloor i/2 \rfloor$

Left(i)

1. **return** $2i$

Right(i)

1. **return** $2i+1$



1	2	3	4	5	6
15	12	10	3	7	2

Bäume als Arrays – Datenstruktur

Navigation

- Wurzel ist $A[1]$

Parent(i)

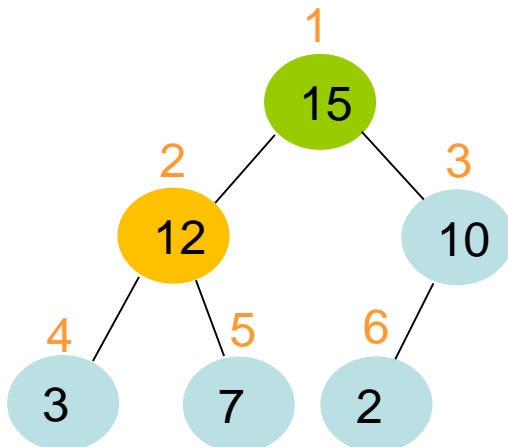
1. **return** $\lfloor i/2 \rfloor$

Left(i)

1. **return** $2i$

Right(i)

1. **return** $2i+1$



1	2	3	4	5	6
15	12	10	3	7	2

Bäume als Arrays – Datenstruktur

Navigation

- Wurzel ist $A[1]$

Parent(i)

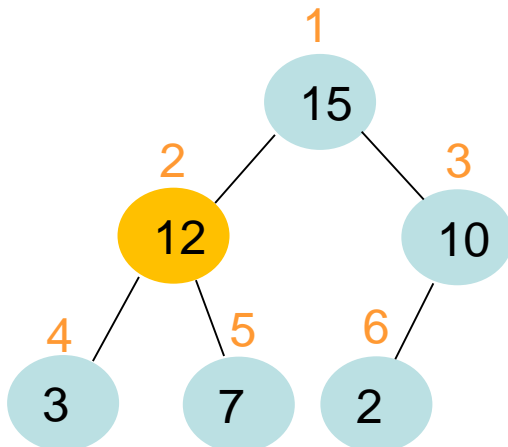
1. **return** $\lfloor i/2 \rfloor$

Left(i)

1. **return** $2i$

Right(i)

1. **return** $2i+1$



1	2	3	4	5	6
15	12	10	3	7	2

Bäume als Arrays – Datenstruktur

Navigation

- Wurzel ist $A[1]$

Parent(i)

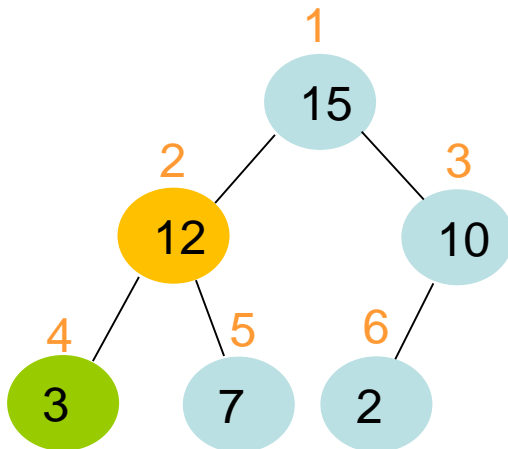
1. **return** $\lfloor i/2 \rfloor$

Left(i)

1. **return** $2i$

Right(i)

1. **return** $2i+1$



1	2	3	4	5	6
15	12	10	3	7	2

Bäume als Arrays – Datenstruktur

Navigation

- Wurzel ist $A[1]$

Parent(i)

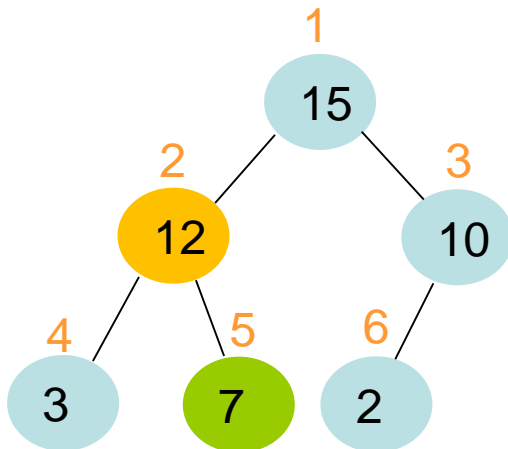
1. **return** $\lfloor i/2 \rfloor$

Left(i)

1. **return** $2i$

Right(i)

1. **return** $2i+1$

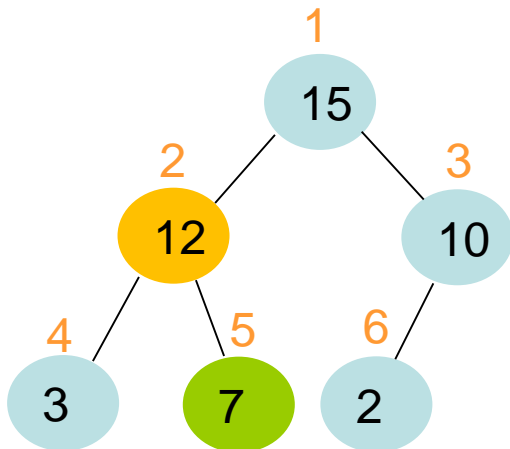


1	2	3	4	5	6
15	12	10	3	7	2

Heap – Datenstruktur

Heap Eigenschaft / Heap property / Heap Eigenschaft

- Für jeden Knoten i außer der Wurzel gilt $A[\text{Parent}(i)] \geq A[i]$

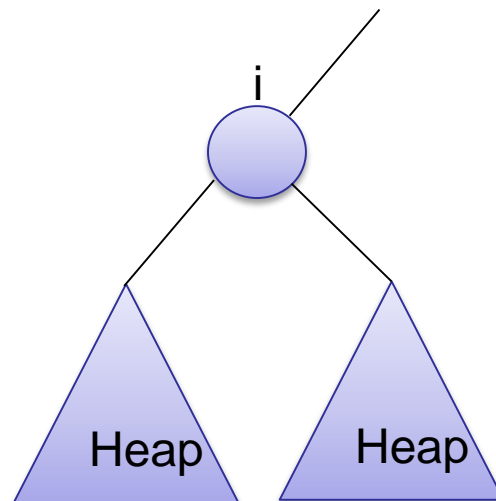


1	2	3	4	5	6
15	12	10	3	7	2

Heapify – Operation

Die Operation Heapify(A,i)

- Voraussetzung: Die Teilarrays mit Wurzel Left(i) und Right(i) sind Heaps
- $A[i]$ ist aber evtl. kleiner als seine Kinder
- Heapify(A,i) lässt i „absinken“, so dass die Heap Eigenschaft erfüllt wird

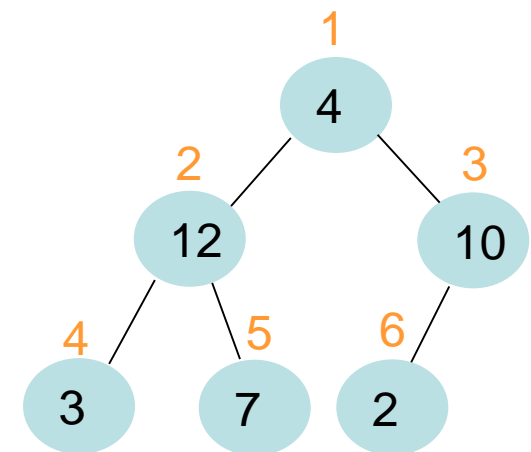


Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
4	12	10	3	7	2



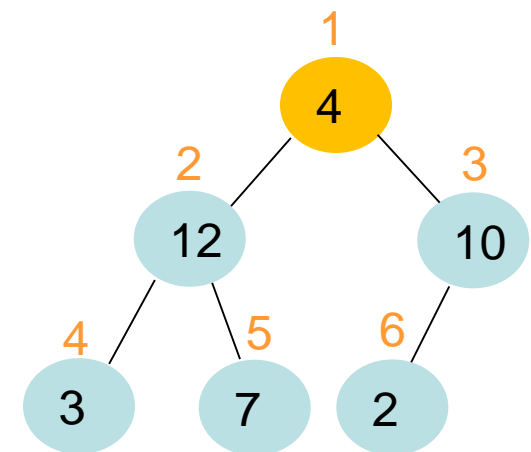
Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A,largest)

1	2	3	4	5	6
4	12	10	3	7	2

i



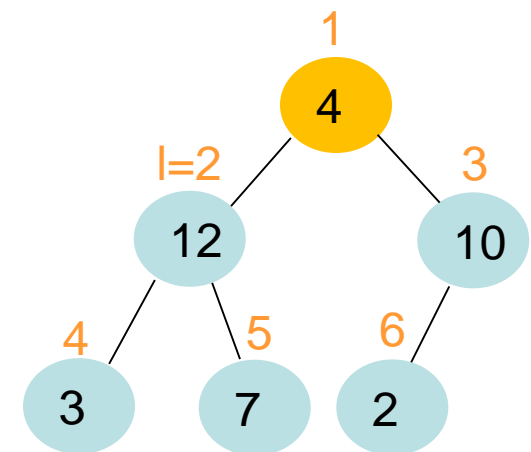
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
4	12	10	3	7	2
i	l				



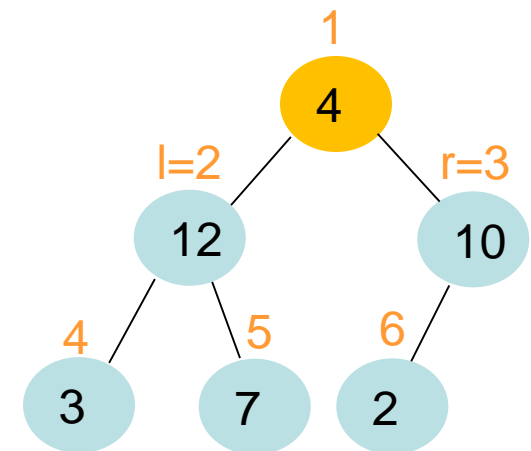
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
4	12	10	3	7	2
i	l	r			



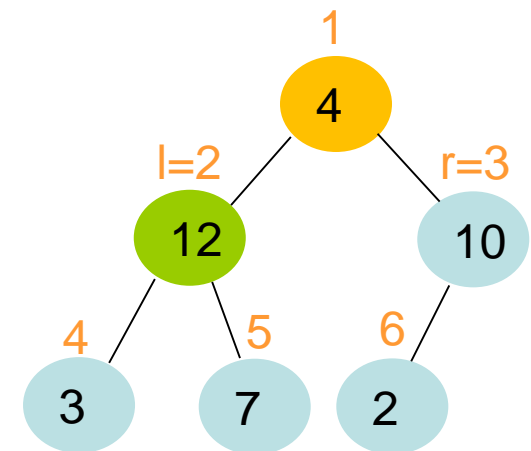
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A,largest)

1	2	3	4	5	6
4	12	10	3	7	2
i	l	r			



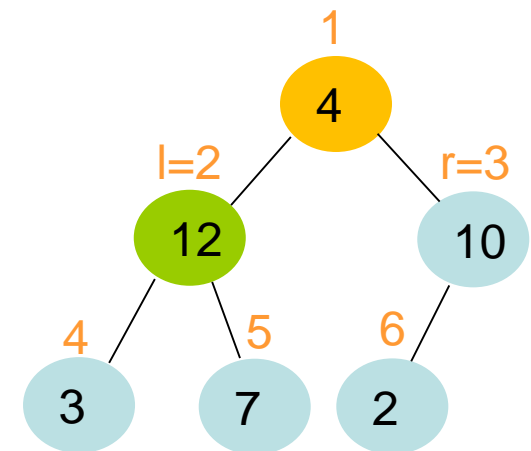
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A,largest)

1	2	3	4	5	6
4	12	10	3	7	2
i	l	r			



Heapify(A,1)

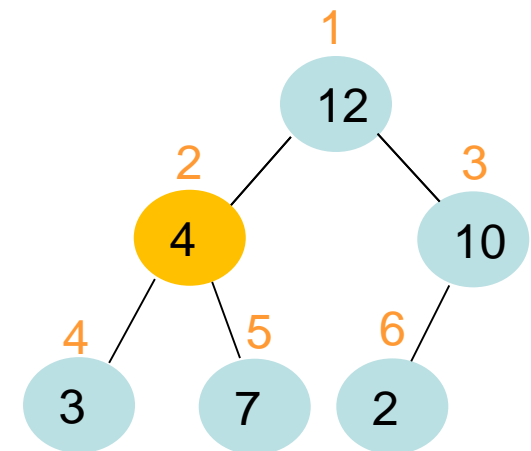
Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	4	10	3	7	2

i



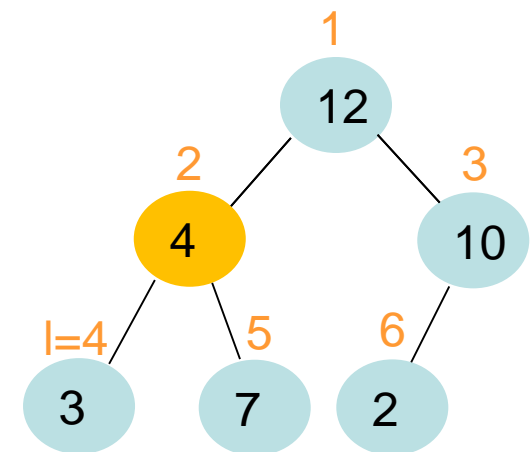
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	4	10	3	7	2
	i		l		



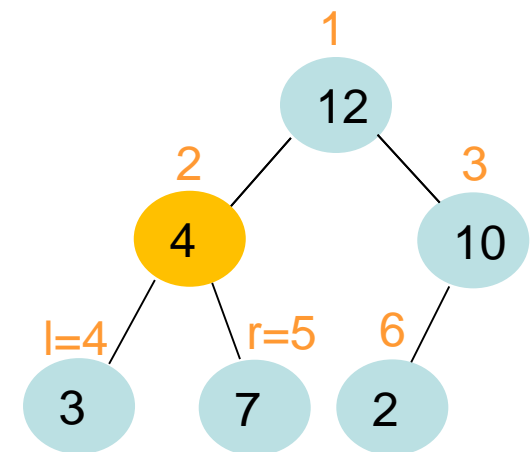
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	4	10	3	7	2
	i		l	r	



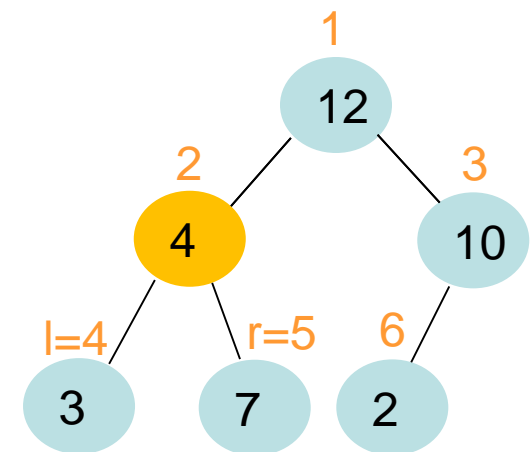
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	4	10	3	7	2
	i		l	r	



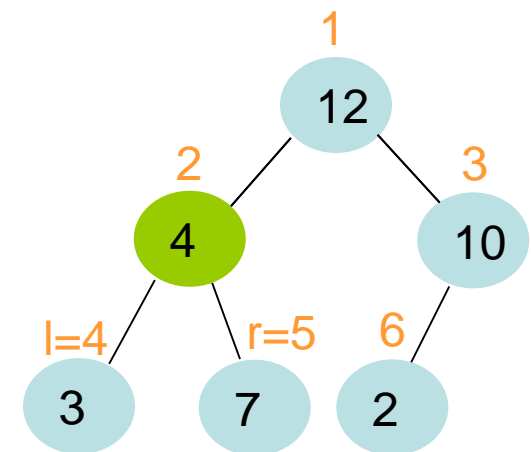
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A,largest)

1	2	3	4	5	6
12	4	10	3	7	2
	i		l	r	



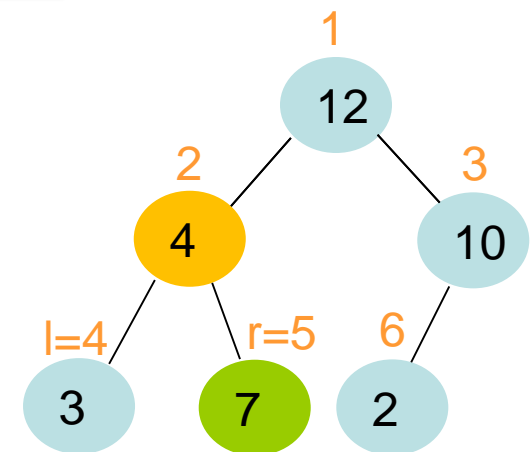
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	4	10	3	7	2
	i		l	r	



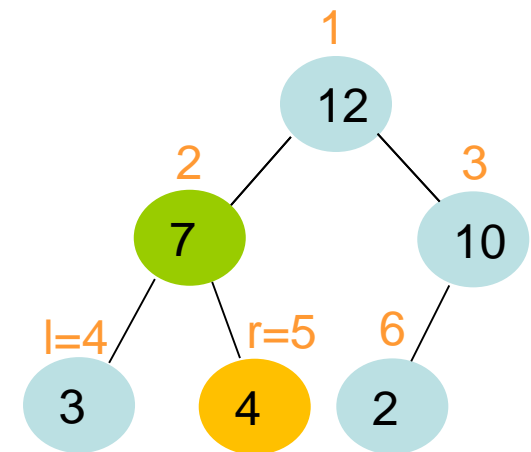
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2
	i		l	r	



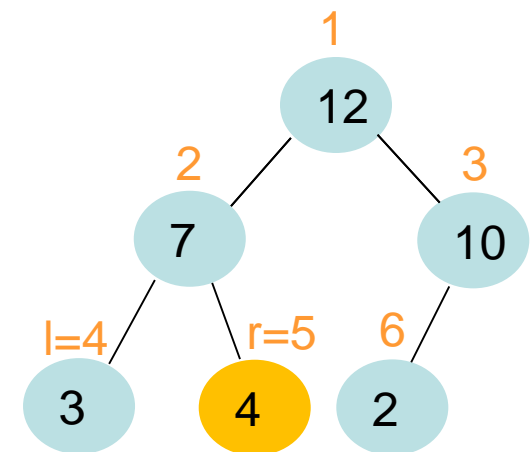
Heapify(A,1)

Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2
	i		l	r	



Heapify(A,1)

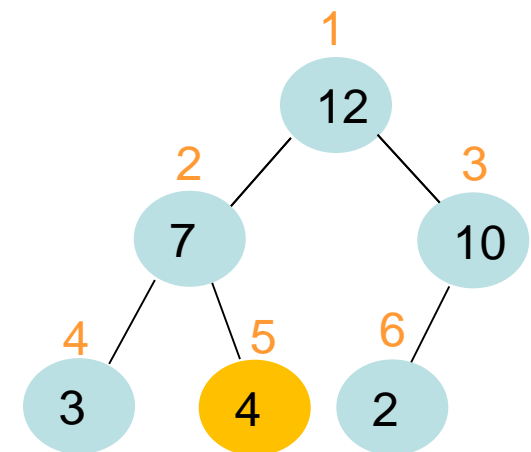
Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2

i



Heapify(A,1)

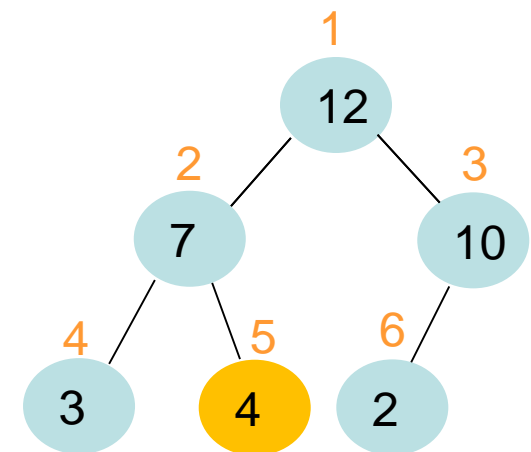
Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2

i
 $l=10$



Heapify(A,1)

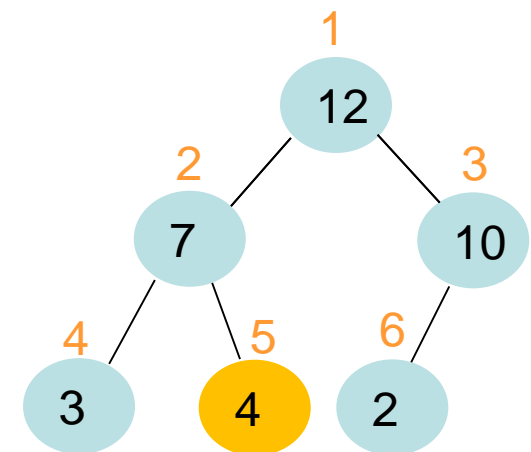
Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2

i
 $l=10$
 $r=11$



Heapify(A,1)

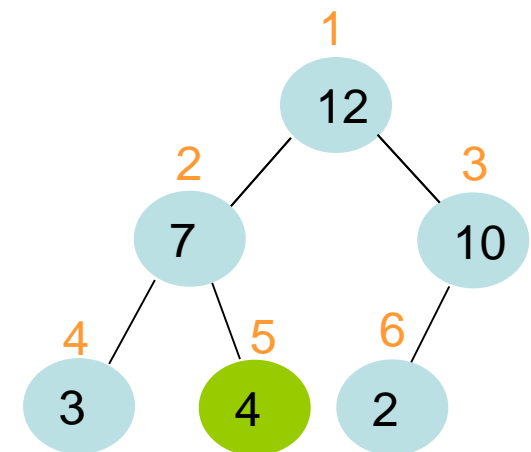
Heapify – Operation

Heapify(A,i)

1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2

i
 $l=10$
 $r=11$



Heapify(A,1)

Heapify – Operation

Heapify(A,i)

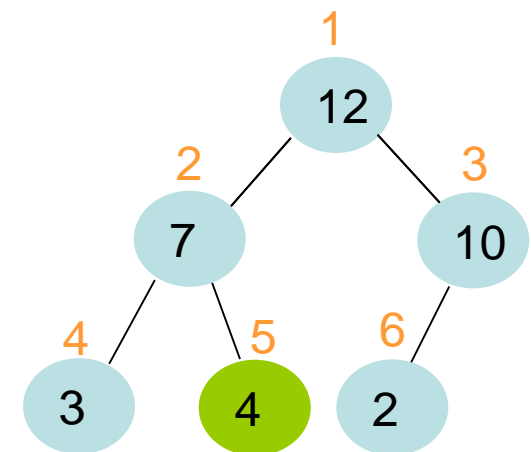
1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2

i

$l=10$

$r=11$



Heapify(A,1)

Heapify – Operation

Heapify(A,i)

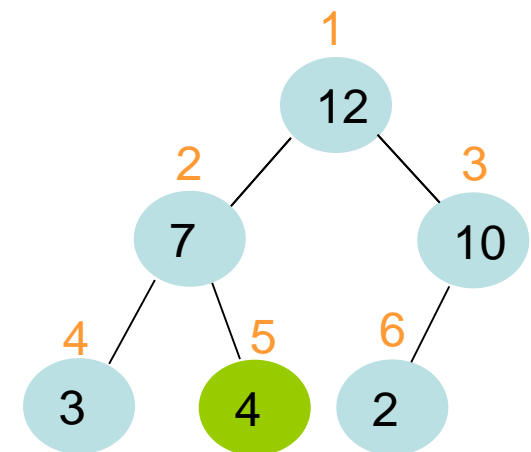
1. $l \leftarrow \text{left}(i)$
2. $r \leftarrow \text{right}(i)$
3. **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$ **then** $\text{largest} \leftarrow l$
4. **else** $\text{largest} \leftarrow i$
5. **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$ **then** $\text{largest} \leftarrow r$
6. **if** $\text{largest} \neq i$ **then** $A[i] \leftrightarrow A[\text{largest}]$
7. Heapify(A, largest)

1	2	3	4	5	6
12	7	10	3	4	2

i

$l=10$

$r=11$



Heapify(A,1)

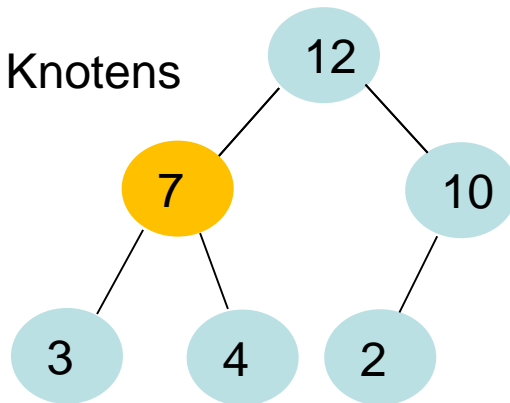
Wiederholung: Baumhöhe

Definition

- Die Höhe eines Knotens v in einem Baum ist die Höhe des Unterbaums von v

Beispiel

Höhe des Knotens
ist 1



Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- Zeige per Induktion über h , dass die Laufzeit $\leq c \cdot h$ ist

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- Zeige per Induktion über h , dass die Laufzeit $\leq c \cdot h$ ist
- (I.A.) $h=0$:
Z. 4: largest wird auf i gesetzt

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- Zeige per Induktion über h , dass die Laufzeit $\leq c \cdot h$ ist
- (I.A.) $h=0$:
 - Z. 4: largest wird auf i gesetzt
 - Z. 5: Keine Änderung.

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- Zeige per Induktion über h , dass die Laufzeit $\leq c \cdot h$ ist
- (I.A.) $h=0$:
 - Z. 4: largest wird auf i gesetzt
 - Z. 5: Keine Änderung.
 - Z. 6/7: Kein rekursiver Aufruf \Rightarrow Laufzeit ist c .

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- Zeige per Induktion über h , dass die Laufzeit $\leq c \cdot h$ ist
- (I.A.) $h=0$:
 - Z. 4: largest wird auf i gesetzt
 - Z. 5: Keine Änderung.
 - Z. 6/7: Kein rekursiver Aufruf \Rightarrow Laufzeit ist c .

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch .

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch .
- (I.S.) Betrachte Knoten der Höhe $h+1$.
 - Z. 3-5: largest wird auf i oder auf eines der Kinder von i gesetzt.
 - Z.6/7: Wenn rekursiver Aufruf durchgeführt, dann mit Kind von i

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch .
- (I.S.) Betrachte Knoten der Höhe $h+1$.
 - Z. 3-5: largest wird auf i oder auf eines der Kinder von i gesetzt.
 - Z.6/7: Wenn rekursiver Aufruf durchgeführt, dann mit Kind von i
- Kind von i hat Höhe h ; nach (I.V.) Laufzeit ch

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch .
- (I.S.) Betrachte Knoten der Höhe $h+1$.
 - Z. 3-5: largest wird auf i oder auf eines der Kinder von i gesetzt.
 - Z.6/7: Wenn rekursiver Aufruf durchgeführt, dann mit Kind von i
- Kind von i hat Höhe h ; nach (I.V.) Laufzeit ch
- **Restliche Laufzeit $\leq c$**

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch .
- (I.S.) Betrachte Knoten der Höhe $h+1$.
 - Z. 3-5: largest wird auf i oder auf eines der Kinder von i gesetzt.
 - Z.6/7: Wenn rekursiver Aufruf durchgeführt, dann mit Kind von i
- Kind von i hat Höhe h ; nach (I.V.) Laufzeit ch
- Restliche Laufzeit $\leq c$
- \Rightarrow Laufzeit maximal $ch+c = c(h+1)$.

Satz

- Die Laufzeit von $\text{Heapify}(A,i)$ ist $O(h)$, wobei h die Höhe des zu i zugehörigen Knotens in der Baumdarstellung des Heaps ist.

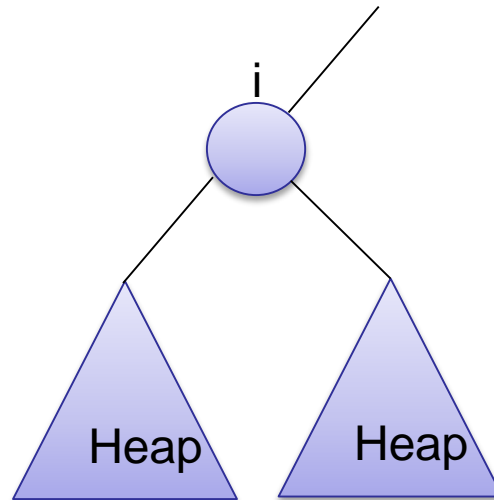
Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch .
- (I.S.) Betrachte Knoten der Höhe $h+1$.
 - Z. 3-5: largest wird auf i oder auf eines der Kinder von i gesetzt.
 - Z.6/7: Wenn rekursiver Aufruf durchgeführt, dann mit Kind von i
- Kind von i hat Höhe h ; nach (I.V.) Laufzeit ch
- Restliche Laufzeit $\leq c$
- \Rightarrow Laufzeit maximal $ch+c = c(h+1)$.

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.



Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.

■ *Beweis*

- Induktion über die Höhe von i .
- (I.A.) Höhe 0 oder 1: Einfaches nachprüfen
- (I.V.) Heapify erfüllt die Aussage des Satzes für Knoten i mit Höhe h .

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.

■ *Beweis*

- Induktion über die Höhe von i .
- (I.A.) Höhe 0 oder 1: Einfaches nachprüfen
- (I.V.) Heapify erfüllt die Aussage des Satzes für Knoten i mit Höhe h .

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- (I.S.) Betrachte Aufruf $\text{Heapify}(A,i)$ für Knoten i der Höhe $h+1 > 1$, wenn Unterbäume der Kindes von i bereits Heap Eigenschaft erfüllen

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- (I.S.) Betrachte Aufruf $\text{Heapify}(A,i)$ für Knoten i der Höhe $h+1 > 1$, wenn Unterbäume der Kindes von i bereits Heap Eigenschaft erfüllen
- l und r : linke bzw. rechte Kind von i

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- (I.S.) Betrachte Aufruf $\text{Heapify}(A,i)$ für Knoten i der Höhe $h+1 > 1$, wenn Unterbäume der Kindes von i bereits Heap Eigenschaft erfüllen
- l und r : linke bzw. rechte Kind von i
- Höhe von $i > 1 \Rightarrow l$ und r kleiner als $\text{heap-size}[A]$
 \Rightarrow die an l und r gespeicherten Werte sind im Heap

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- (I.S.) Betrachte Aufruf $\text{Heapify}(A,i)$ für Knoten i der Höhe $h+1 > 1$, wenn Unterbäume der Kindes von i bereits Heap Eigenschaft erfüllen
- l und r : linke bzw. rechte Kind von i
- Höhe von $i > 1 \Rightarrow l$ und r kleiner als $\text{heap-size}[A]$
 \Rightarrow die an l und r gespeicherten Werte sind im Heap
 $A[i]$, $A[l]$ und $A[r]$: Werte der entsprechenden Knoten

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- (I.S.) Betrachte Aufruf $\text{Heapify}(A,i)$ für Knoten i der Höhe $h+1 > 1$, wenn Unterbäume der Kindes von i bereits Heap Eigenschaft erfüllen
- l und r : linke bzw. rechte Kind von i
- Höhe von $i > 1 \Rightarrow l$ und r kleiner als $\text{heap-size}[A]$
 \Rightarrow die an l und r gespeicherten Werte sind im Heap
 $A[i]$, $A[l]$ und $A[r]$: Werte der entsprechenden Knoten
- **Z. 3-5: $\text{Heapify}(A,i)$ speichert Index von $\max\{A[i], A[l], A[r]\}$ in largest**

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.

■ Beweis

- (I.S.) Betrachte Aufruf $\text{Heapify}(A,i)$ für Knoten i der Höhe $h+1 > 1$, wenn Unterbäume der Kindes von i bereits Heap Eigenschaft erfüllen
- l und r : linke bzw. rechte Kind von i
- Höhe von $i > 1 \Rightarrow l$ und r kleiner als $\text{heap-size}[A]$
 \Rightarrow die an l und r gespeicherten Werte sind im Heap
 $A[i]$, $A[l]$ und $A[r]$: Werte der entsprechenden Knoten
- Z. 3-5: $\text{Heapify}(A,i)$ speichert Index von $\max\{A[i], A[l], A[r]\}$ in largest
- Maximum ist $A[i]$: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf₅₁

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.

■ Beweis

- (I.S.) Betrachte Aufruf $\text{Heapify}(A,i)$ für Knoten i der Höhe $h+1 > 1$, wenn Unterbäume der Kindes von i bereits Heap Eigenschaft erfüllen
- l und r : linke bzw. rechte Kind von i
- Höhe von $i > 1 \Rightarrow l$ und r kleiner als $\text{heap-size}[A]$
 \Rightarrow die an l und r gespeicherten Werte sind im Heap
 $A[i]$, $A[l]$ und $A[r]$: Werte der entsprechenden Knoten
- Z. 3-5: $\text{Heapify}(A,i)$ speichert Index von $\max\{A[i], A[l], A[r]\}$ in largest
- Maximum ist $A[i]$: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- $A[l]$ ist Maximum ($A[r]$ analog):
- Unterbäume von l und r sind Heaps $\Rightarrow A[l]$ und $A[r]$ sind größte Elemente in ihren Unterbäumen

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- $A[l]$ ist Maximum ($A[r]$ analog):
- Unterbäume von l und r sind Heaps $\Rightarrow A[l]$ und $A[r]$ sind größte Elemente in ihren Unterbäumen
 $A[l]$ ist $\max\{A[i], A[l], A[r]\} \Rightarrow A[l]$ ist größtes Element im Unterbaum von i

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- $A[l]$ ist Maximum ($A[r]$ analog):
- Unterbäume von l und r sind Heaps $\Rightarrow A[l]$ und $A[r]$ sind größte Elemente in ihren Unterbäumen
 $A[l]$ ist $\max\{A[i], A[l], A[r]\} \Rightarrow A[l]$ ist größtes Element im Unterbaum von i
- Z. 6: $A[i]$ wird mit $A[l]$ getauscht
Z. 7: Aufruf von Heapify für Unterbaum von l ; Höhe des Unterbaums ist h
Nach (I.V.): Nach Aufruf hat dieser Unterbaum Heap Eigenschaft

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- $A[l]$ ist Maximum ($A[r]$ analog):
- Unterbäume von l und r sind Heaps $\Rightarrow A[l]$ und $A[r]$ sind größte Elemente in ihren Unterbäumen
 $A[l]$ ist $\max\{A[i], A[l], A[r]\} \Rightarrow A[l]$ ist größtes Element im Unterbaum von i
- Z. 6: $A[i]$ wird mit $A[l]$ getauscht
Z. 7: Aufruf von Heapify für Unterbaum von l ; Höhe des Unterbaums ist h
Nach (I.V.): Nach Aufruf hat dieser Unterbaum Heap Eigenschaft
- Bei i ist Max. aller Elemente gespeichert und rechter Unterbaum erfüllt Heap Eigenschaft \Rightarrow Unterbaum von i ist ein Heap

Heap Eigenschaft und Heapify

Satz

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Heap Eigenschaft besitzen, dann ist diese nach der Operation $\text{Heapify}(A,i)$ für den Unterbaum von i erfüllt.
- *Beweis*
- $A[l]$ ist Maximum ($A[r]$ analog):
- Unterbäume von l und r sind Heaps $\Rightarrow A[l]$ und $A[r]$ sind größte Elemente in ihren Unterbäumen
 $A[l]$ ist $\max\{A[i], A[l], A[r]\} \Rightarrow A[l]$ ist größtes Element im Unterbaum von i
- Z. 6: $A[i]$ wird mit $A[l]$ getauscht
Z. 7: Aufruf von Heapify für Unterbaum von l ; Höhe des Unterbaums ist h
Nach (I.V.): Nach Aufruf hat dieser Unterbaum Heap Eigenschaft
- Bei i ist Max. aller Elemente gespeichert und rechter Unterbaum erfüllt Heap Eigenschaft \Rightarrow Unterbaum von i ist ein Heap

Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)

Heap – Aufbau

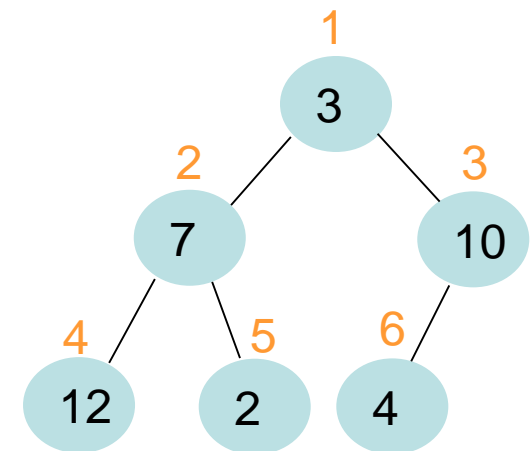
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



Heap – Aufbau

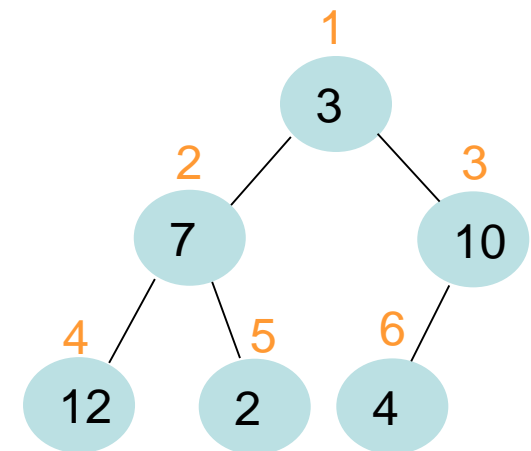
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



Heap – Aufbau

1	2	3	4	5	6
3	7	10	12	2	4

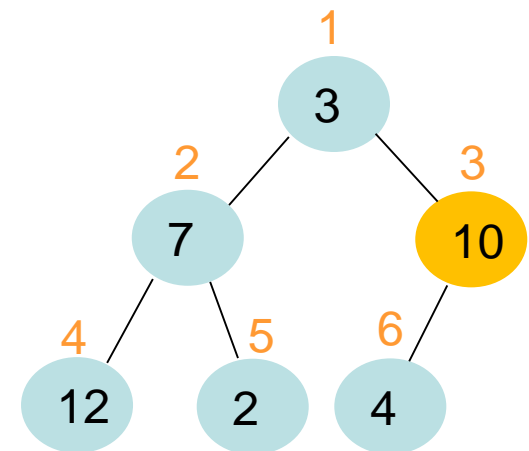
i

Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



Heap – Aufbau

1	2	3	4	5	6
3	7	10	12	2	4

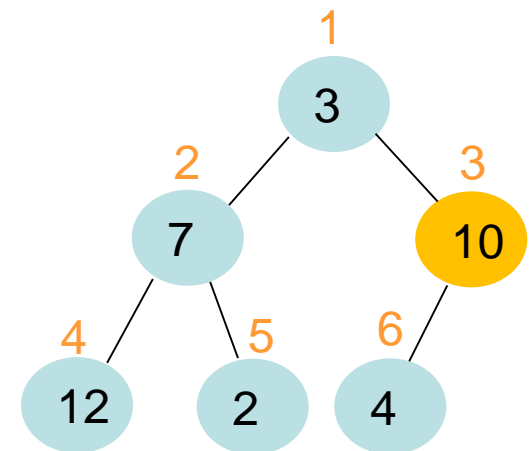
i

Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



Heap – Aufbau

1	2	3	4	5	6
3	7	10	12	2	4

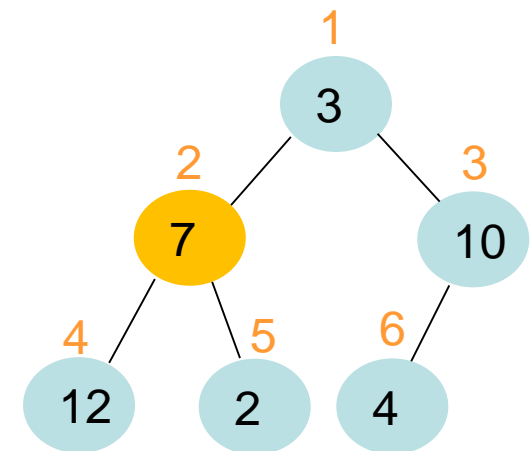
i

Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



Heap – Aufbau

1	2	3	4	5	6
3	12	10	7	2	4

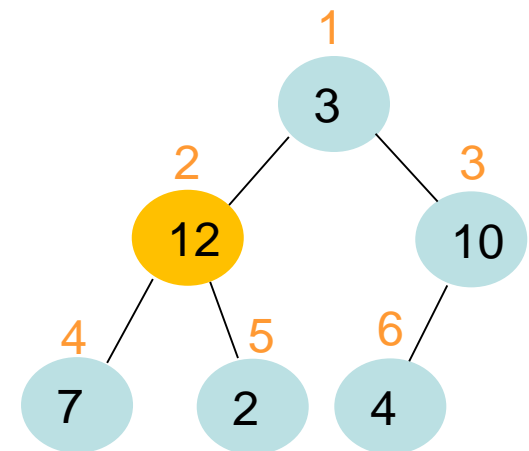
i

Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



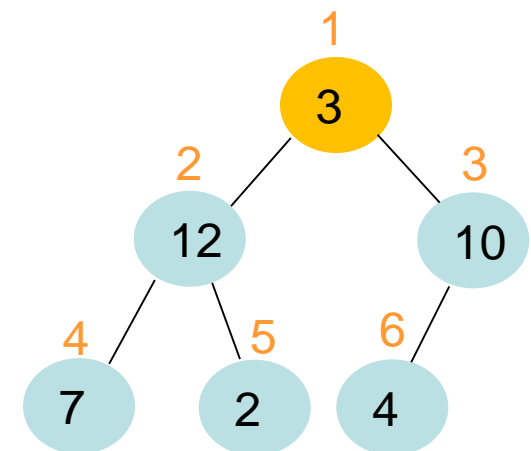
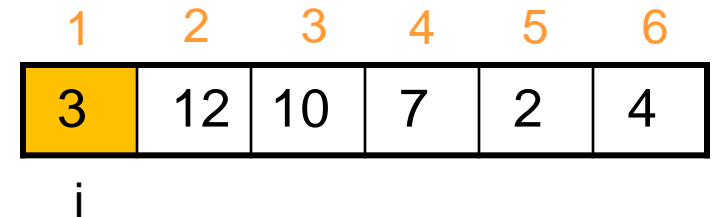
Heap – Aufbau

Aufbau eines Heaps (Halde)

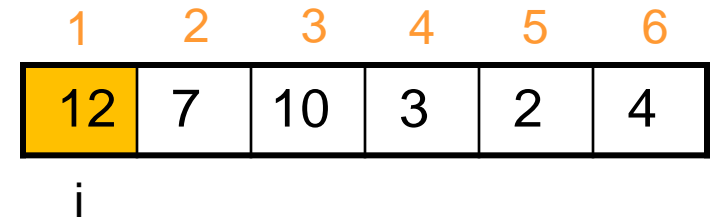
- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



Heap – Aufbau

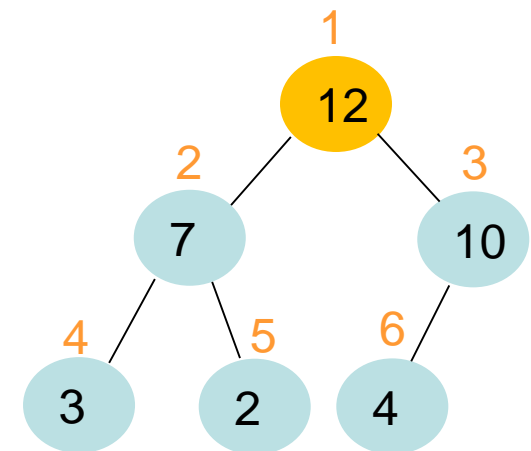


Aufbau eines Heaps (Halde)

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)



Heap – Aufbau

Aufbau eines Heaps (Halde)

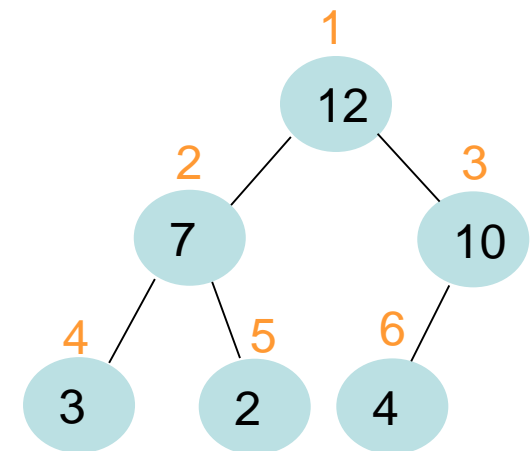
- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size \leftarrow length[A]
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1 **do**
3. Heapify(A,i)

1	2	3	4	5	6
12	7	10	3	2	4

i



Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.
- *Beweis*
- **Korrektheit:**
(Inv.) Unterbäume der Knoten größer als i besitzen Heap Eigenschaft

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.
- *Beweis*
- Korrektheit:
(Inv.) Unterbäume der Knoten größer als i besitzen Heap Eigenschaft
- Gilt insbesondere für die Unterbäume der Kinder von i

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.
- *Beweis*
- Korrektheit:
(Inv.) Unterbäume der Knoten größer als i besitzen Heap Eigenschaft
- Gilt insbesondere für die Unterbäume der Kinder von i
- Aus vorherigen Satz folgt, dass Invariante durch Heapify aufrechterhalten wird

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.
- *Beweis*
- Korrektheit:
(Inv.) Unterbäume der Knoten größer als i besitzen Heap Eigenschaft
- Gilt insbesondere für die Unterbäume der Kinder von i
- Aus vorherigen Satz folgt, dass Invariante durch Heapify aufrechterhalten wird
- Damit gilt am Ende der Schleife die Heap Eigenschaft für die Wurzel
- \Rightarrow Build-Heap erzeugt Heap.

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.
- *Beweis*
- Korrektheit:
(Inv.) Unterbäume der Knoten größer als i besitzen Heap Eigenschaft
- Gilt insbesondere für die Unterbäume der Kinder von i
- Aus vorherigen Satz folgt, dass Invariante durch Heapify aufrechterhalten wird
- Damit gilt am Ende der Schleife die Heap Eigenschaft für die Wurzel
- \Rightarrow Build-Heap erzeugt Heap.

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.
- *Beweis*
- Einfache Laufzeitanalyse: Jedes Heapify benötigt $O(h) = O(\log n)$ Laufzeit. Da es insgesamt $O(n)$ Heapify Operationen gibt, ist die Laufzeit $O(n \log n)$.

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.
- *Beweis*
- Schärfere Laufzeitanalyse:
- Beobachtung: In einem Heap mit Höhe H gibt es maximal 2^0 Knoten mit Höhe H , 2^1 Knoten mit Höhe $H-1$, 2^2 Knoten mit Höhe $H-2$, usw.

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.

- *Beweis*

- Schärfere Laufzeitanalyse:
- Beobachtung: In einem Heap mit Höhe H gibt es maximal 2^0 Knoten mit Höhe H , 2^1 Knoten mit Höhe $H-1$, 2^2 Knoten mit Höhe $H-2$, usw.
- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H = \lfloor \log n \rfloor$:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \dots) = O\left(\sum_{h=0}^H (h+1) \cdot 2^{H-h}\right) = O\left(2^H \cdot \sum_{h=0}^H \frac{h+1}{2^h}\right) = O\left(n \cdot \sum_{h=0}^H \frac{h+1}{2^h}\right)$$

Satz

- Mit Hilfe des Algorithmus Build-Heap kann man einen Heap in $O(n)$ Zeit aufbauen.

- *Beweis*

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \dots) = O\left(\sum_{h=0}^H (h+1) \cdot 2^{H-h}\right) = O\left(2^H \cdot \sum_{h=0}^H \frac{h+1}{2^h}\right) = O\left(n \cdot \sum_{h=0}^H \frac{h+1}{2^h}\right)$$

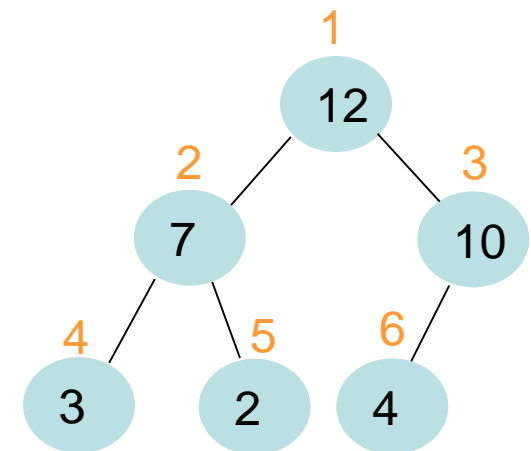
- Es gilt $\sum_{h=0}^{\infty} \frac{h+1}{2^h} = O(1)$
- Somit folgt eine Laufzeit von $O(n)$.

Max Heap – Extract Max

Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. max \leftarrow A[1]
3. A[1] \leftarrow A[heap-size[A]]
4. heap-size[A] \leftarrow heap-size[A]-1
5. Heapify(A,1)
6. **return** max

1	2	3	4	5	6
12	7	10	3	2	4

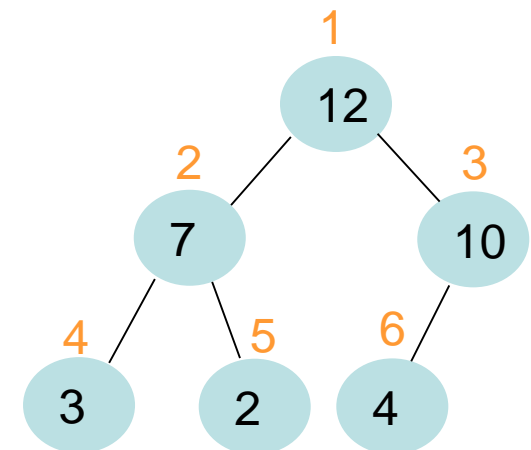


Max Heap – Extract Max

Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. max \leftarrow A[1]
3. A[1] \leftarrow A[heap-size[A]]
4. heap-size[A] \leftarrow heap-size[A]-1
5. Heapify(A,1)
6. **return** max

1	2	3	4	5	6
12	7	10	3	2	4



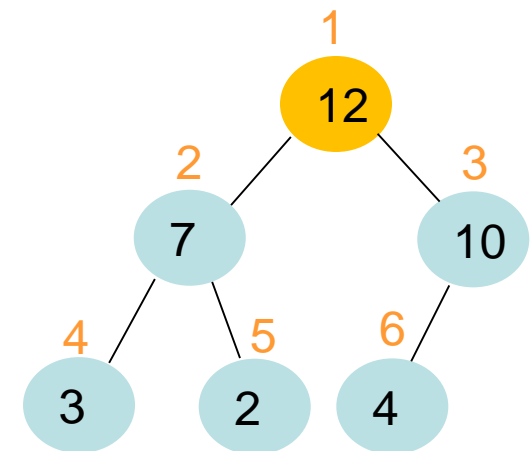
Max Heap – Extract Max

Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. **max** \leftarrow A[1]
3. A[1] \leftarrow A[heap-size[A]]
4. heap-size[A] \leftarrow heap-size[A]-1
5. Heapify(A,1)
6. **return** max

1	2	3	4	5	6
12	7	10	3	2	4

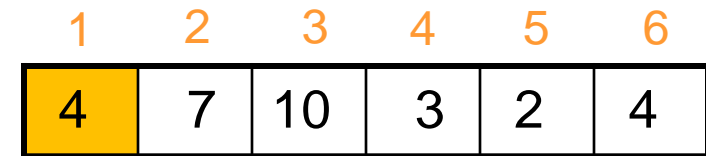
max = 12



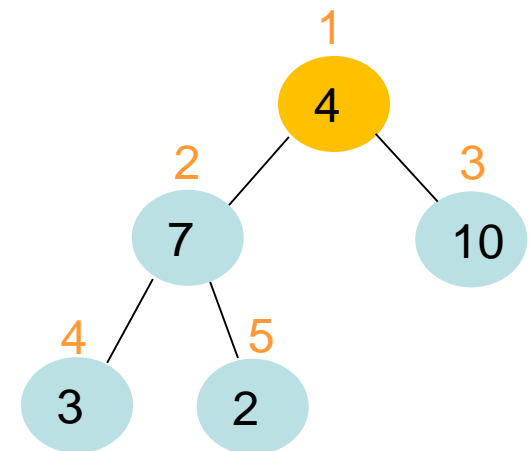
Max Heap – Extract Max

Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. max \leftarrow A[1]
3. A[1] \leftarrow A[heap-size[A]]
4. heap-size[A] \leftarrow heap-size[A]-1
5. Heapify(A,1)
6. **return** max



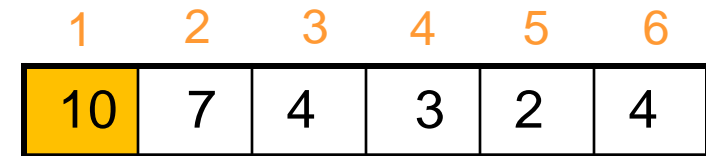
max = 12



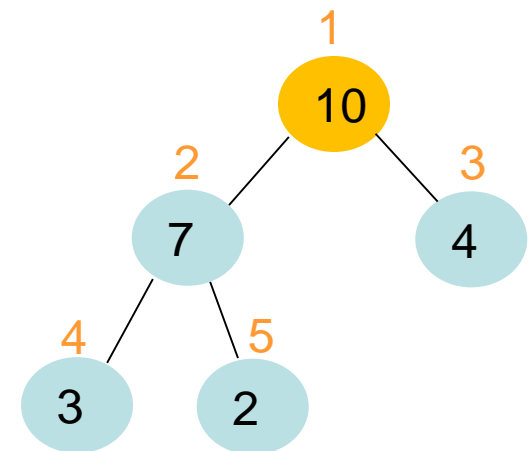
Max Heap – Extract Max

Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. max \leftarrow A[1]
3. A[1] \leftarrow A[heap-size[A]]
4. heap-size[A] \leftarrow heap-size[A]-1
5. Heapify(A,1)
6. **return** max



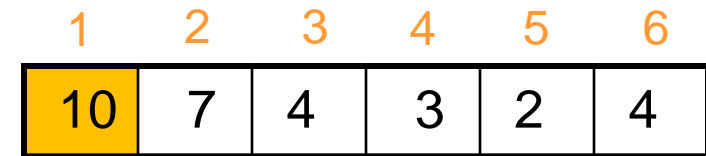
max = 10



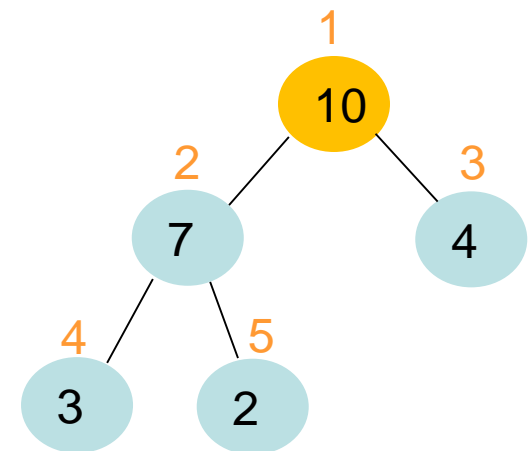
Max Heap – Extract Max

Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. max \leftarrow A[1]
3. A[1] \leftarrow A[heap-size[A]]
4. heap-size[A] \leftarrow heap-size[A]-1
5. Heapify(A,1)
6. **return** max



max = 10



Max Heap – Extract Max

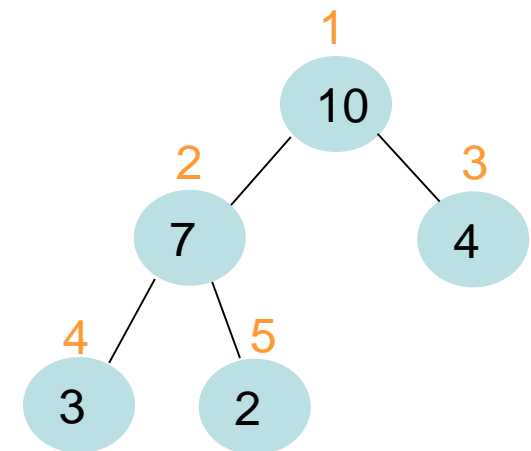
Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. max \leftarrow A[1]
3. A[1] \leftarrow A[heap-size[A]]
4. heap-size[A] \leftarrow heap-size[A]-1
5. Heapify(A,1)
6. **return** max

Laufzeit

- $O(\log n)$

1	2	3	4	5	6
10	7	4	3	2	4

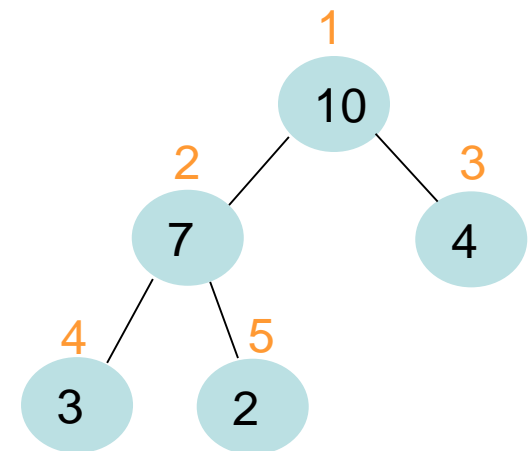


Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

1	2	3	4	5	6
10	7	4	3	2	4

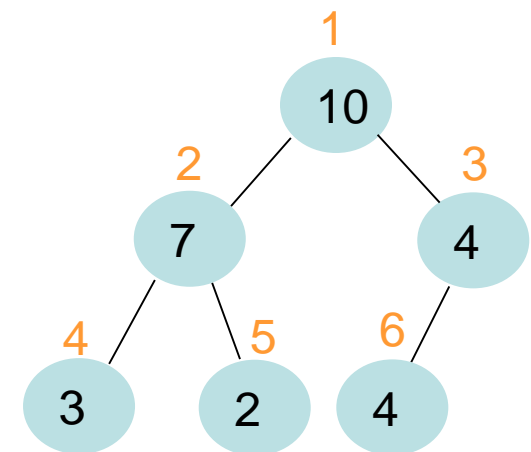


Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

1	2	3	4	5	6
10	7	4	3	2	4

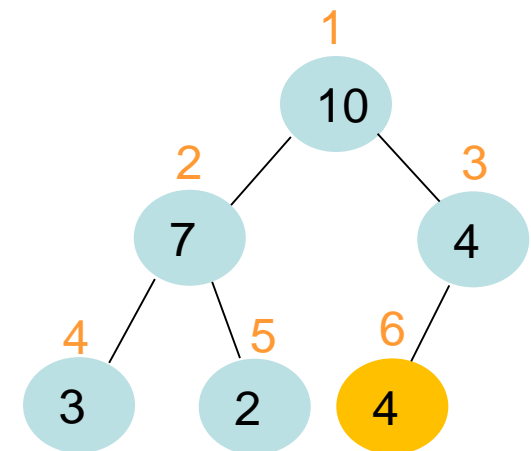
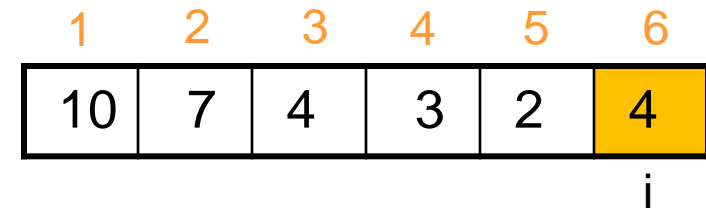


Heap-Insert(A, 11)

Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

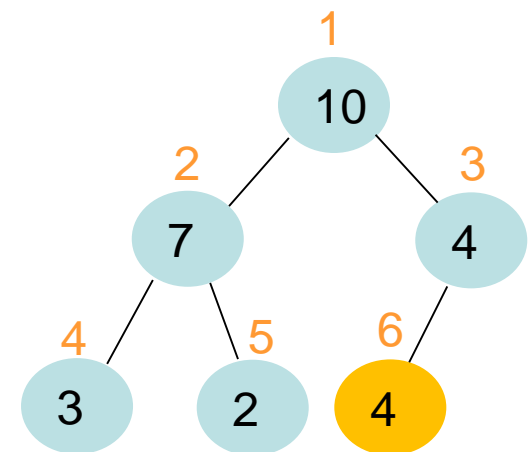
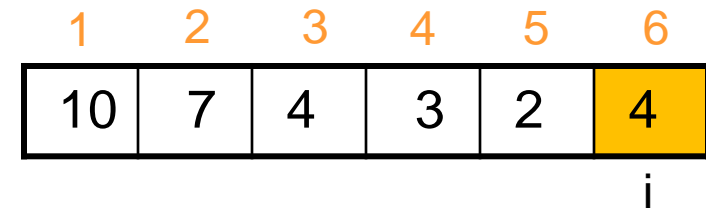


Heap-Insert(A, 11)

Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

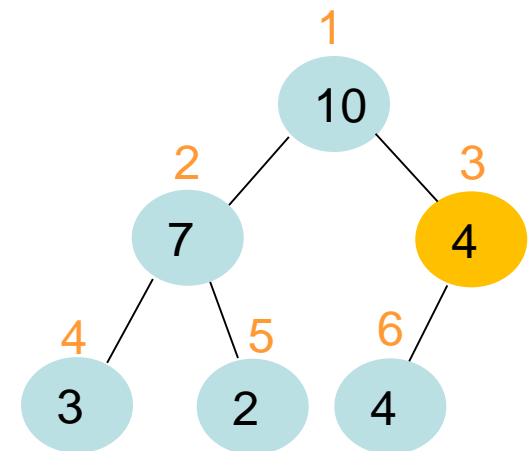
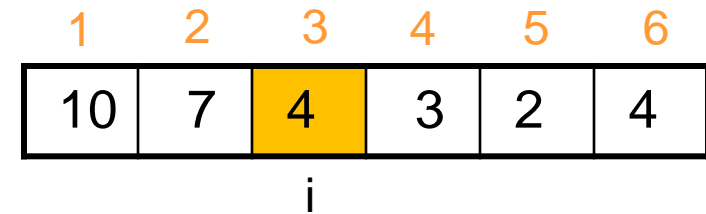


Heap-Insert(A, 11)

Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

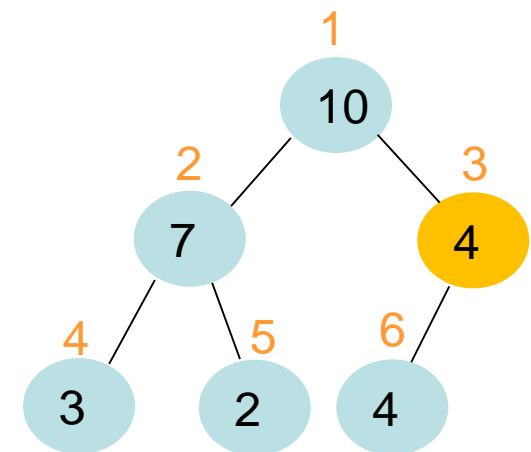
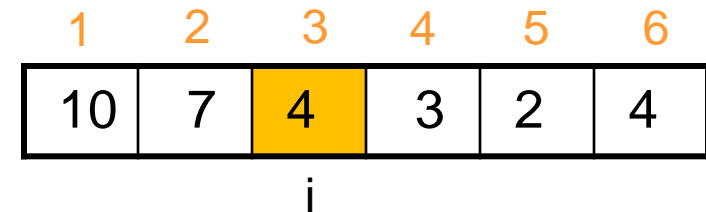


Heap-Insert(A, 11)

Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

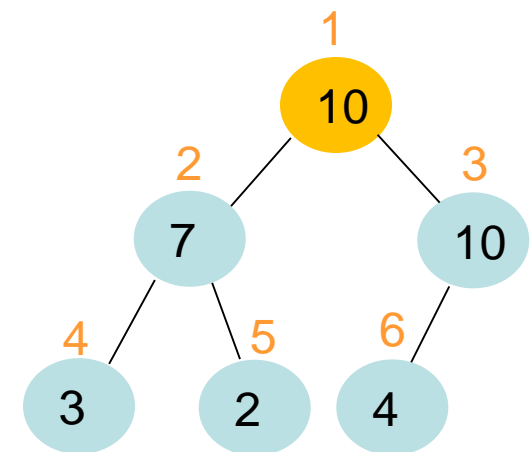
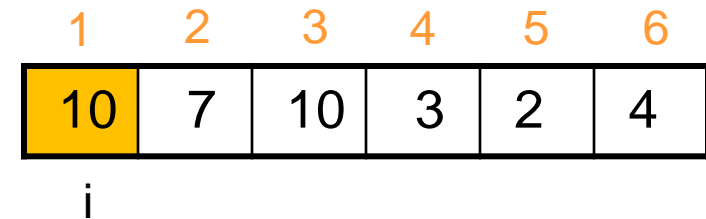


Heap-Insert($A, 11$)

Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

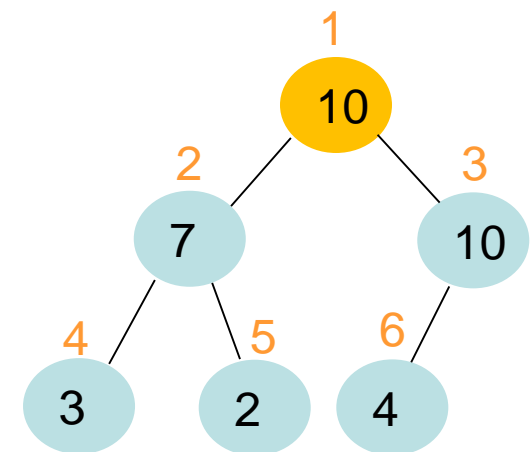
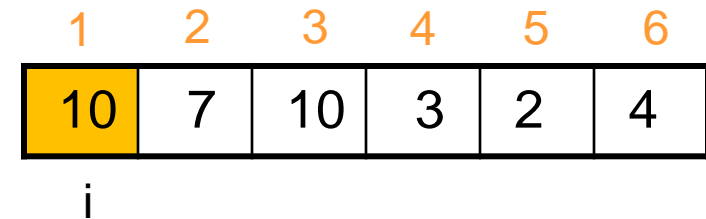


Heap-Insert(A, 11)

Max Heap – Einfügen

Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$



Heap-Insert(A, 11)

Max Heap – Einfügen

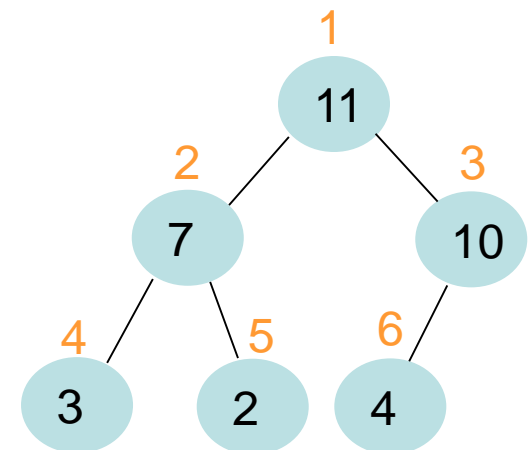
Heap-Insert(A, key)

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $i \leftarrow \text{heap-size}[A]$
3. **while** $i > 1$ and $A[\text{Parent}(i)] < \text{key}$ **do**
4. $A[i] \leftarrow A[\text{Parent}(i)]$
5. $i \leftarrow \text{Parent}(i)$
6. $A[i] \leftarrow \text{key}$

Laufzeit

- $O(\log n)$

1	2	3	4	5	6
11	7	10	3	2	4

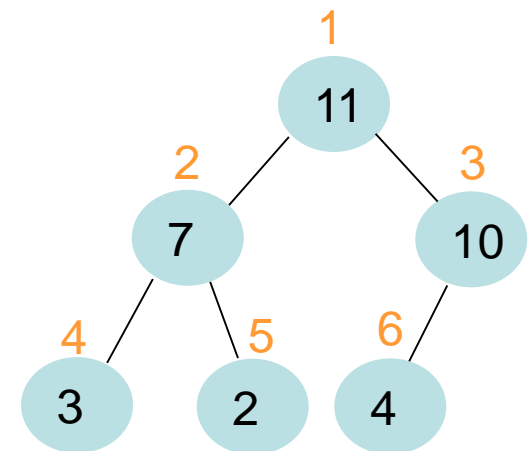


Heapsort

Heapsort(A)

1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)

1	2	3	4	5	6
11	7	10	3	2	4

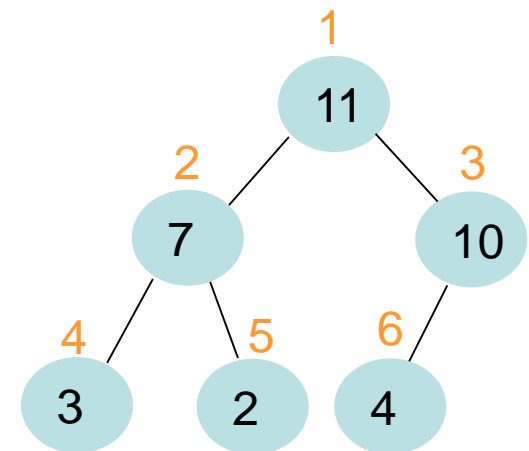


Heapsort

Heapsort(A)

1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)

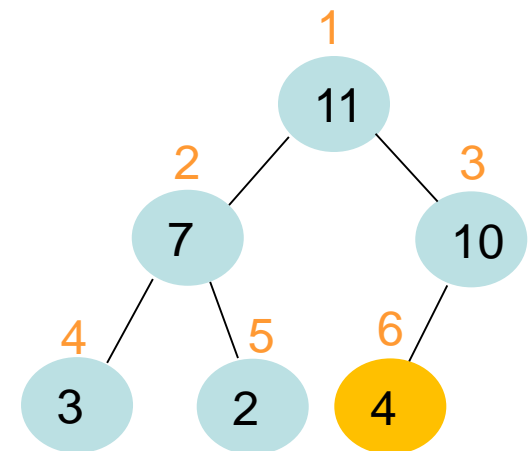
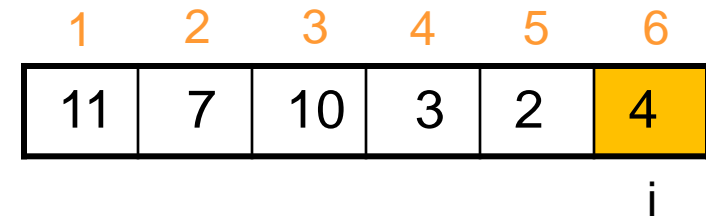
1	2	3	4	5	6
11	7	10	3	2	4



Heapsort

Heapsort(A)

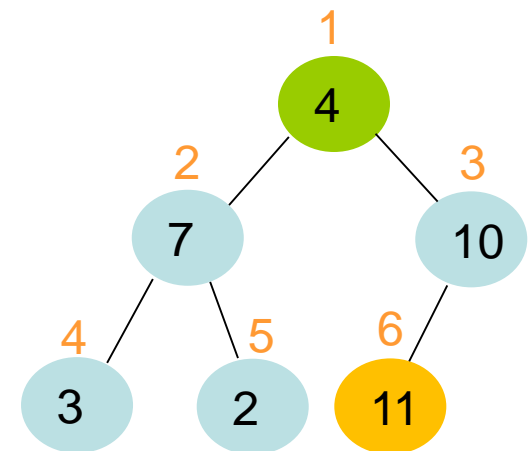
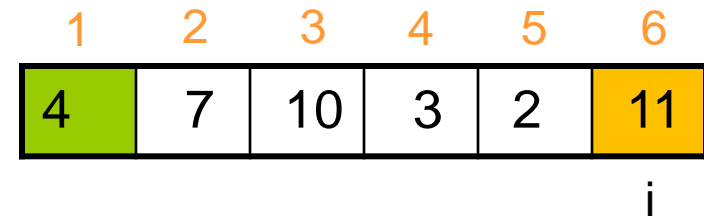
1. Build-Heap(A)
2. **for $i \leftarrow \text{length}[A]$ downto 2 do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

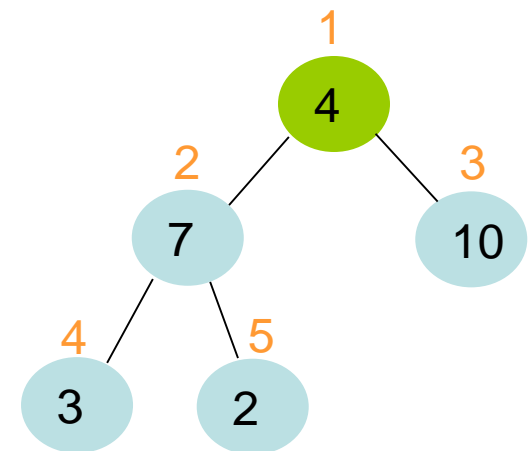
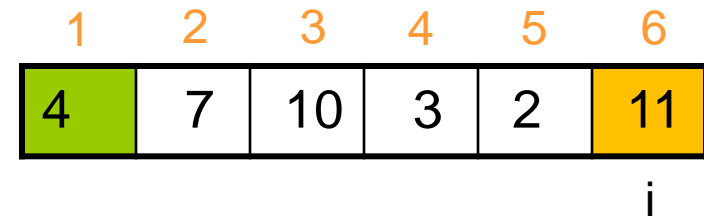
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

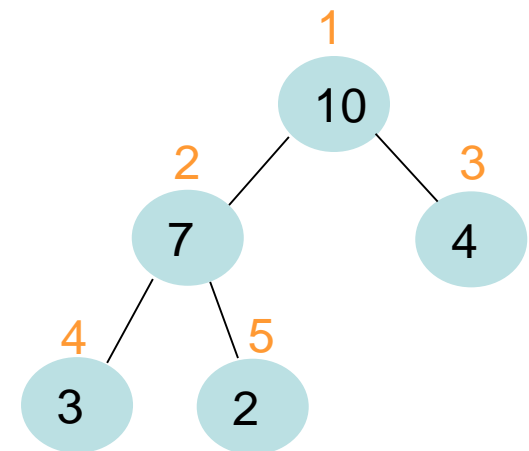
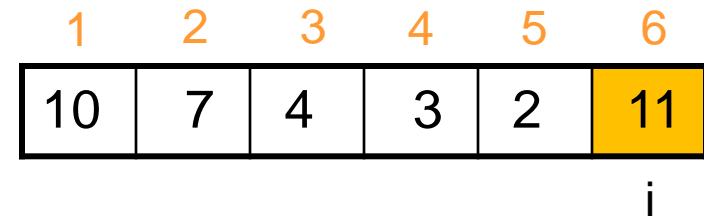
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

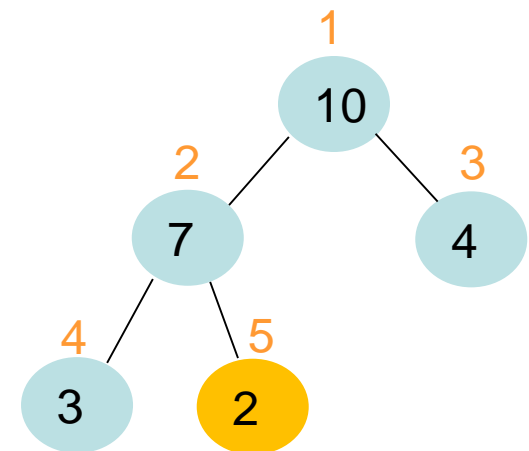
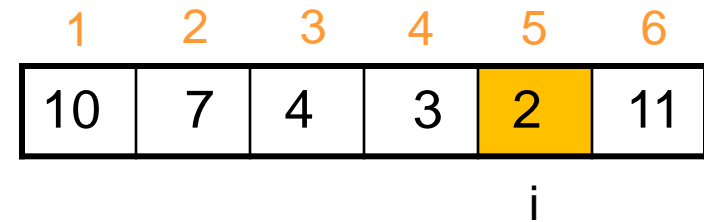
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

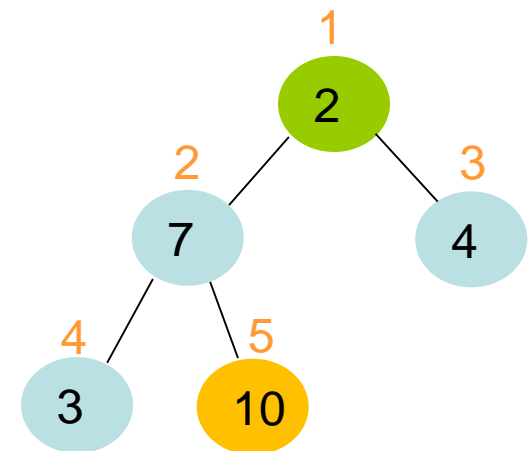
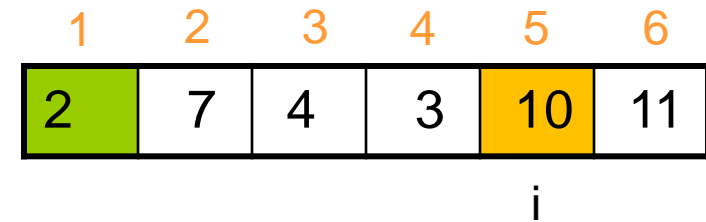
1. Build-Heap(A)
2. **for $i \leftarrow \text{length}[A]$ downto 2 do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

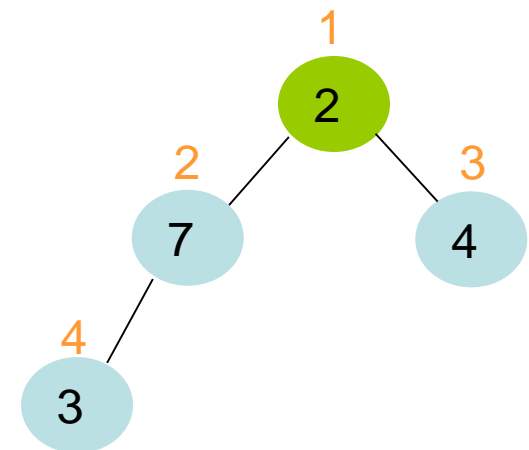
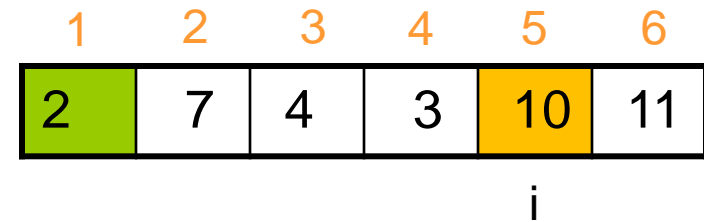
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

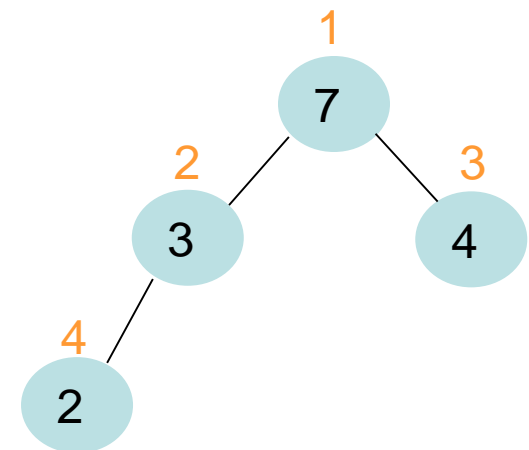
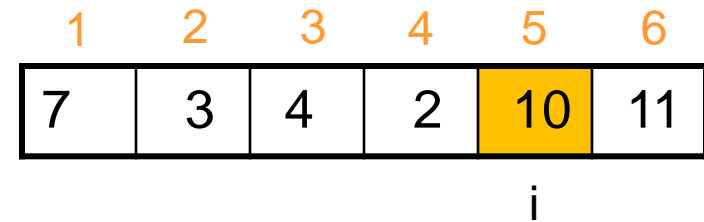
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)

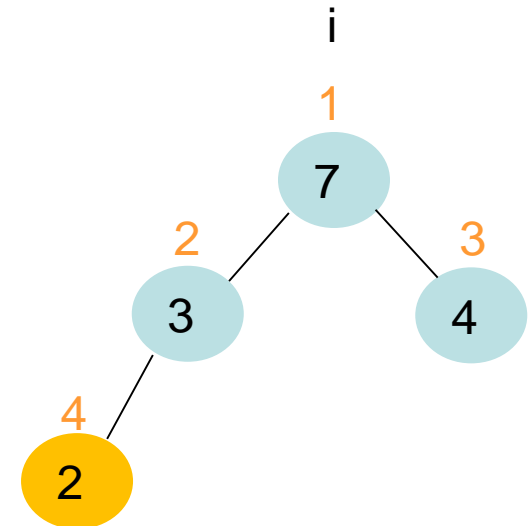


Heapsort

Heapsort(A)

1. Build-Heap(A)
2. **for $i \leftarrow \text{length}[A]$ downto 2 do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)

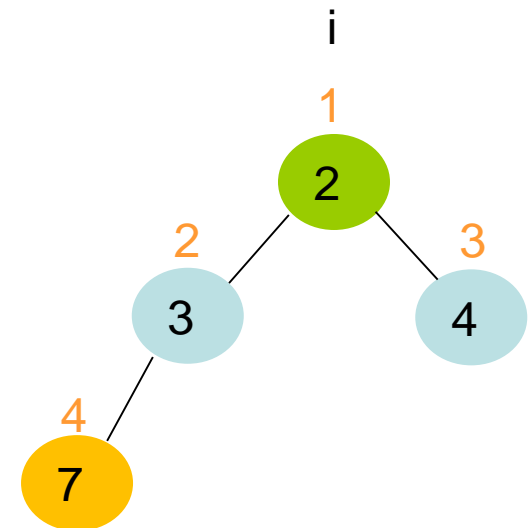
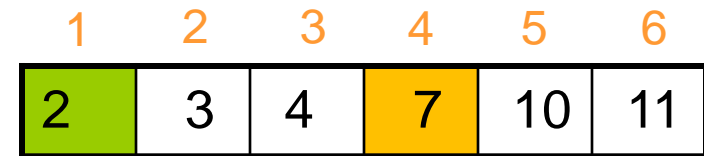
1	2	3	4	5	6
7	3	4	2	10	11



Heapsort

Heapsort(A)

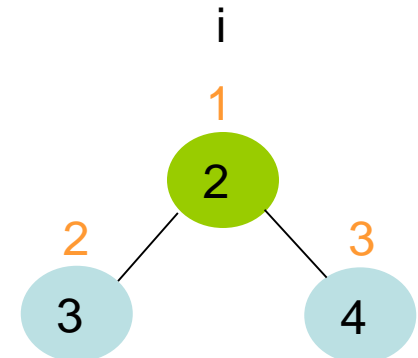
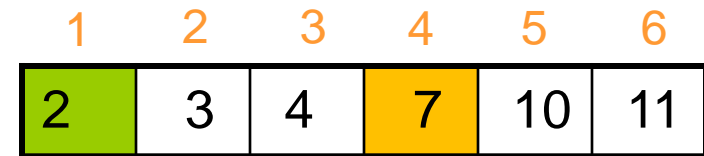
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

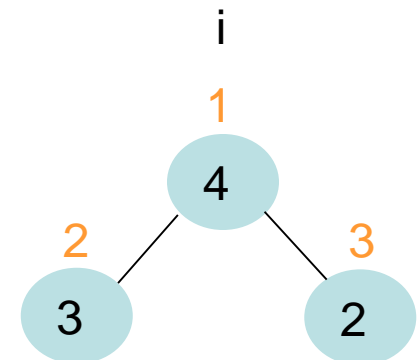
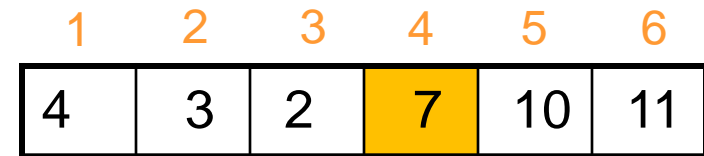
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

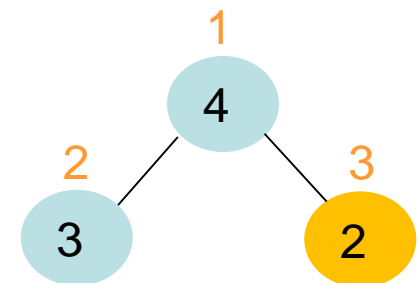
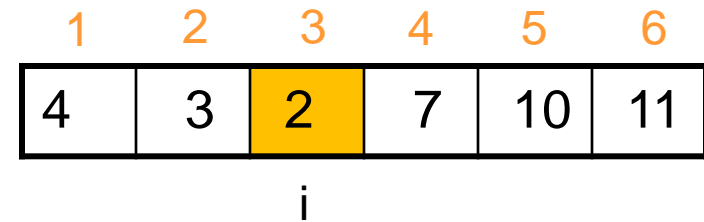
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

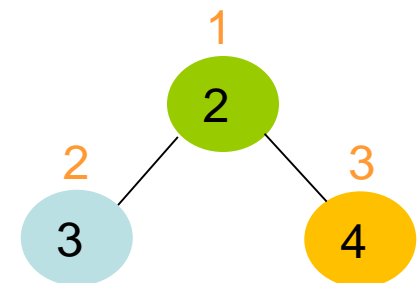
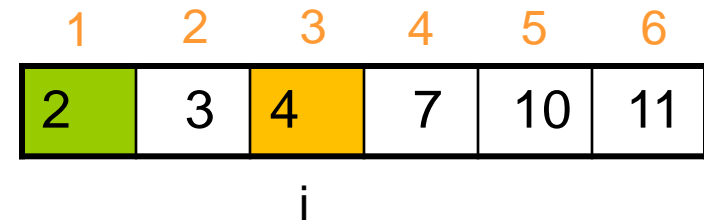
1. Build-Heap(A)
2. **for $i \leftarrow \text{length}[A]$ downto 2 do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

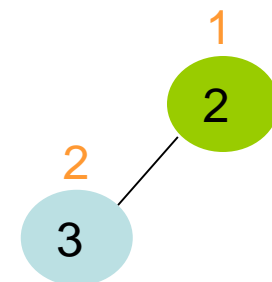
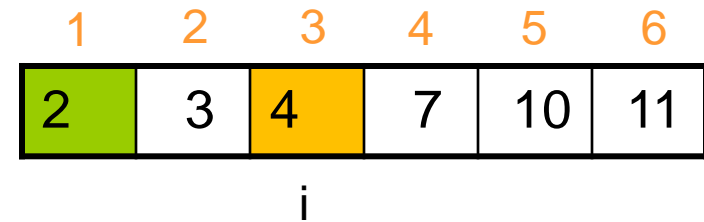
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

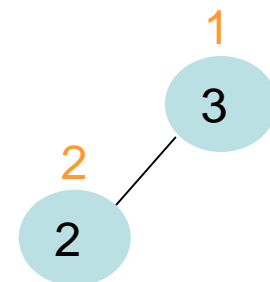
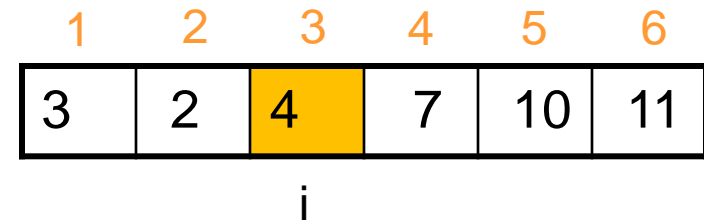
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

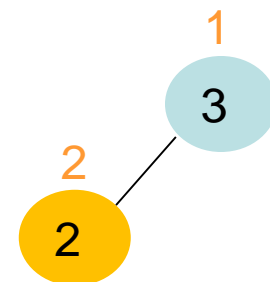
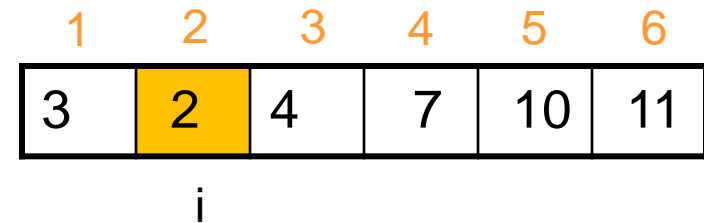
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

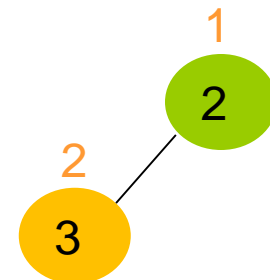
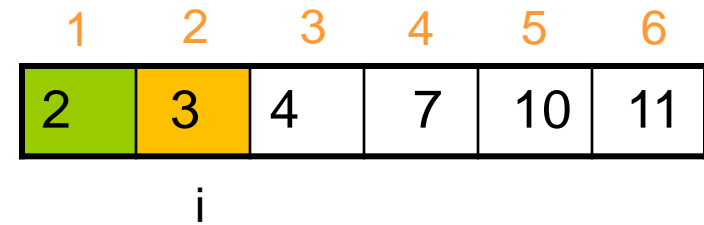
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

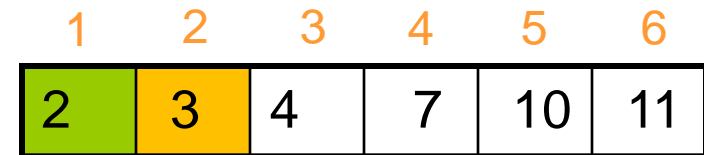
1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



Heapsort

Heapsort(A)

1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)



i



Heapsort

Heapsort(A)

1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)

1	2	3	4	5	6
2	3	4	7	10	11

i

1
2

Heapsort

Heapsort(A)

1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)

1	2	3	4	5	6
2	3	4	7	10	11

Heapsort - Laufzeit

Heapsort(A)

1. Build-Heap(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2 **do**
3. $A[1] \leftrightarrow A[i]$
4. $\text{heap-size}[A] \leftarrow \text{heap-size}[A]-1$
5. Heapify(A,1)

Laufzeit

- $O(n \log n)$

1	2	3	4	5	6
2	3	4	7	10	11

Zusammenfassung Halden (Heaps)

- Einfügen, Löschen, Maximum extrahieren in $O(\log n)$ Zeit
- Sortieren mit Hilfe von Heaps in $O(n \log n)$
- Heapsort braucht keinen zusätzlichen Speicherplatz
- Einfache Implementierung
- Beispiel für die gelungene Kombination von einer Datenstruktur und eines Algorithmus