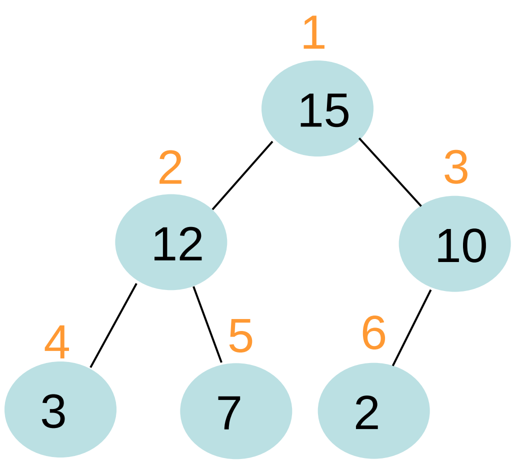
Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Datenstrukturen

Binäre Halden

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



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

Datenstrukturen

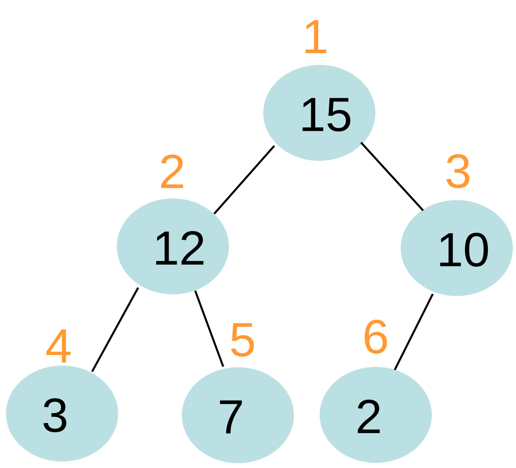
Navigation

- 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

Datenstrukturen

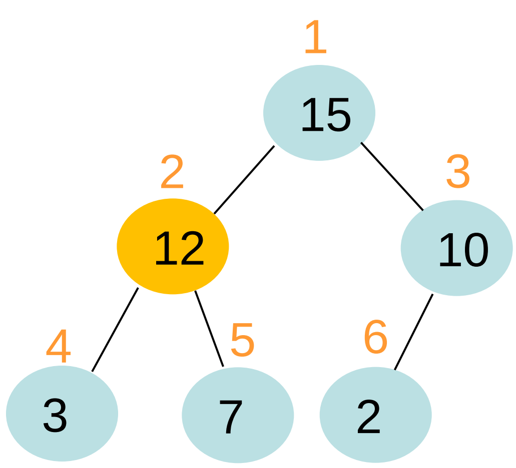
Navigation

- 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

Datenstrukturen

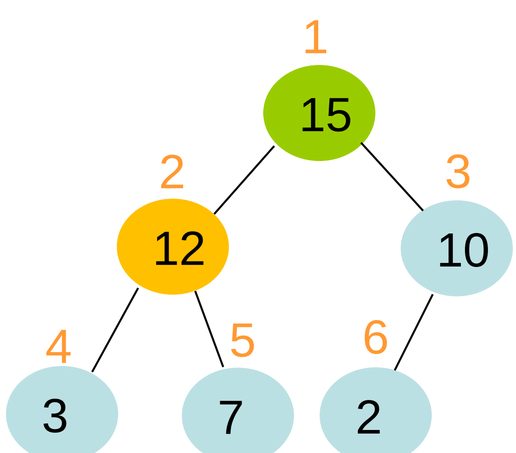
Navigation

- 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

Datenstrukturen

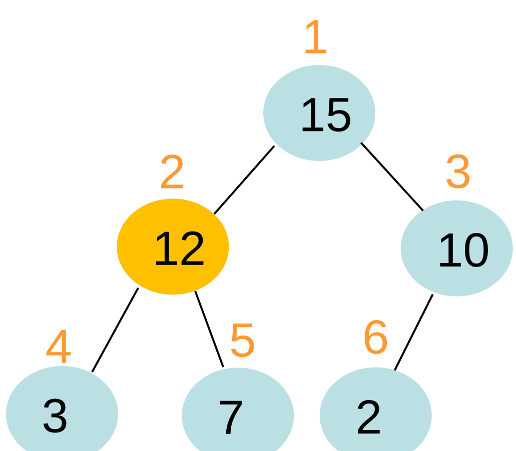
Navigation

- Wurzel ist A[1]

Left(i)
1. **return** 2i

Parent(i)
1. **return** $\lfloor i/2 \rfloor$

Right(i)
1. **return** 2i+1



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

Datenstrukturen

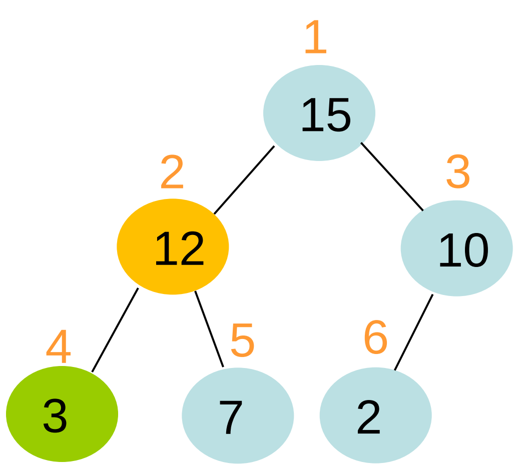
Navigation

- Wurzel ist A[1]

Left(i)
1. **return** 2i

Parent(i)
1. **return** $\lfloor i/2 \rfloor$

Right(i)
1. **return** 2i+1



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

Datenstrukturen

Navigation

- Wurzel ist A[1]

Parent(i)

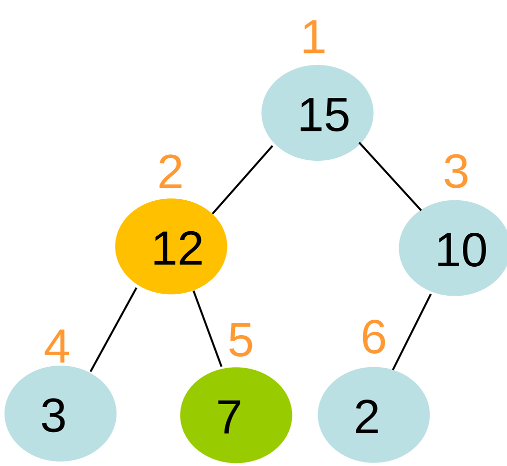
1. **return** $\lfloor i/2 \rfloor$

Left(i)

1. **return** 2i

Right(i)

1. **return** 2i+1

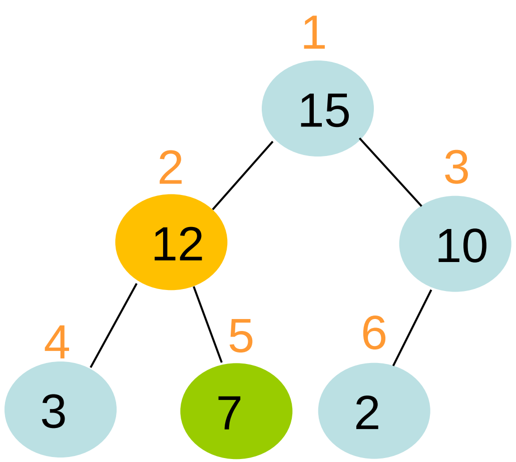


1	2	3	4	5	6
15	12	10	3	7	2

Datenstrukturen

Haldeneigenschaft

- Für jeden Knoten i außer der Wurzel gilt A[Parent(i)]≥A[i]

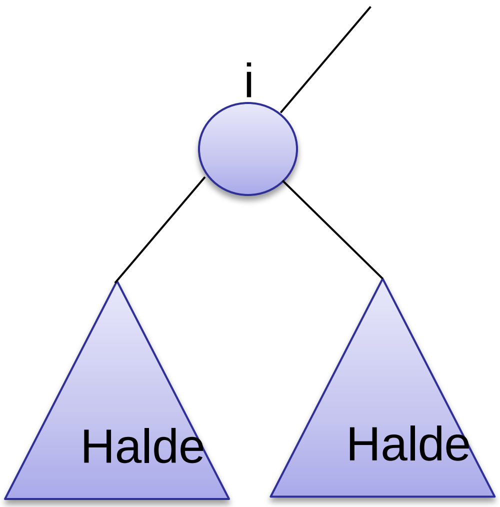


1	2	3	4	5	6
15	12	10	3	7	2

Datenstrukturen

Die Operation Heapify(A,i)

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

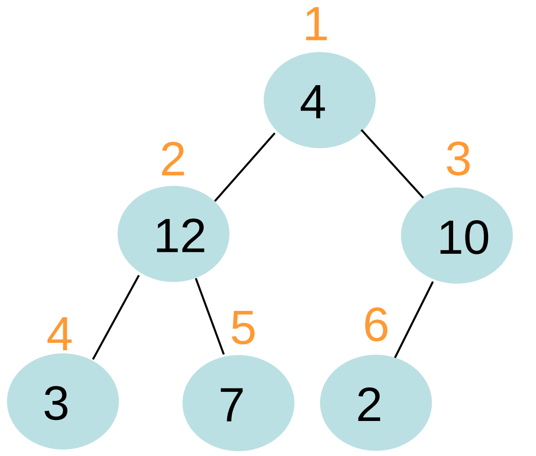


Datenstrukturen

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

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7.   Heapify(A,largest)

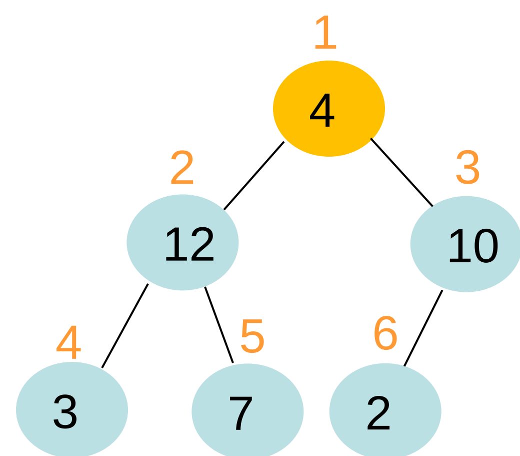


Datenstrukturen

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

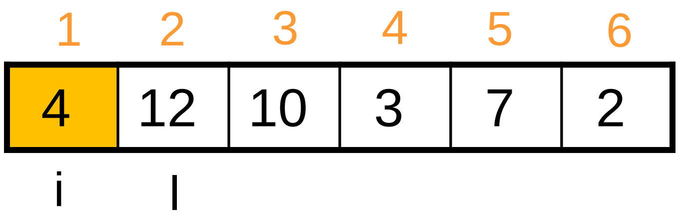
Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



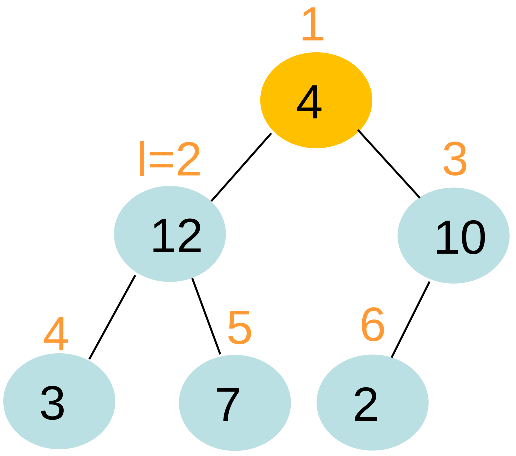
Heapify(A,1)

Datenstrukturen



Heapify(A,i)

$l \leftarrow$ left(i)
$r \leftarrow$ right(i)
if $l \leq$heap-size[A] and A[l]>A[i] **then** largest $\leftarrow l$
else largest $\leftarrow$ i
if $r \leq$heap-size[A] and A[r]>A[largest] **then** largest $\leftarrow$ r
if largest$\neq$i **then** A[i] $\leftrightarrow$ A[largest]
    Heapify(A,largest)

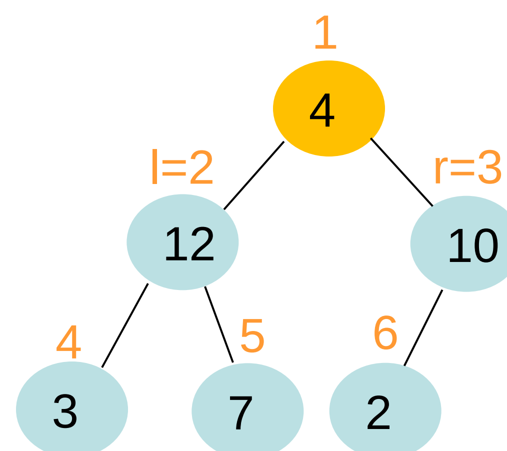


Heapify(A,1)

Datenstrukturen

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

```
Heapify(A,i)
l ← left(i)
r ← right(i)
if l≤heap-size[A] and A[l]>A[i] then largest ← l
else largest ← i
if r≤heap-size[A] and A[r]>A[largest] then largest ← r
if largest≠i then A[i] ↔ A[largest]
    Heapify(A,largest)
```



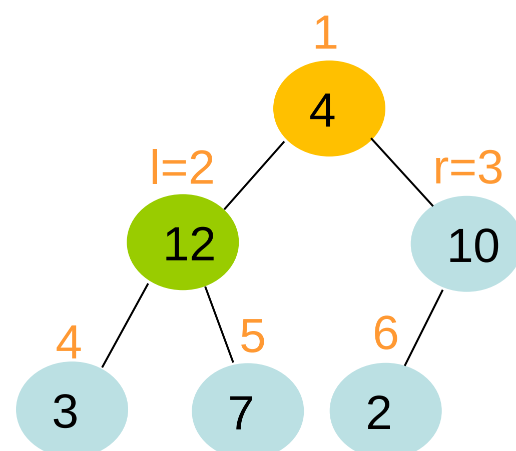
Heapify(A,1)

Datenstrukturen

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

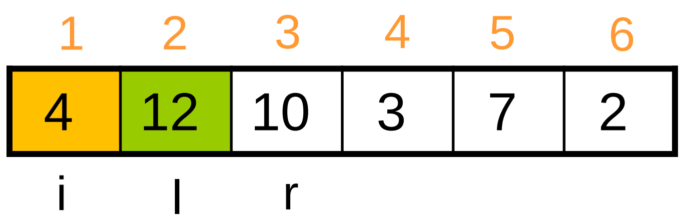
Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



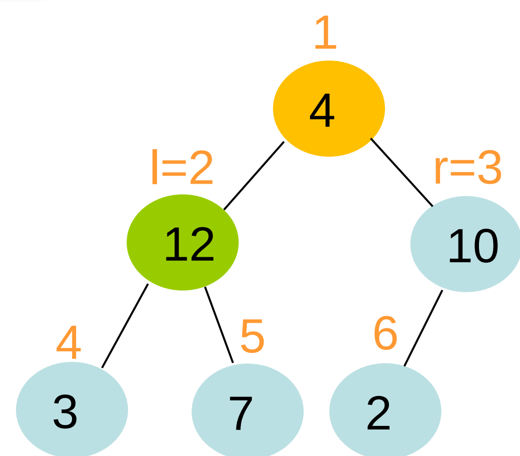
Heapify(A,1)

Datenstrukturen



Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



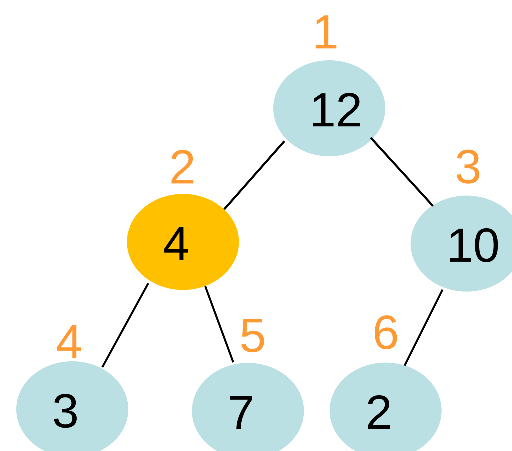
Heapify(A,1)

Datenstrukturen

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

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



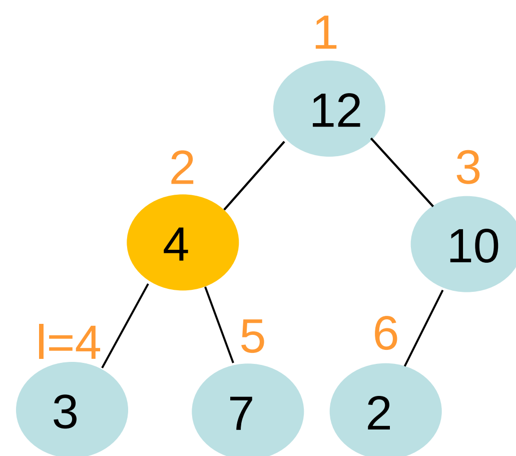
Heapify(A,1)

Datenstrukturen

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

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



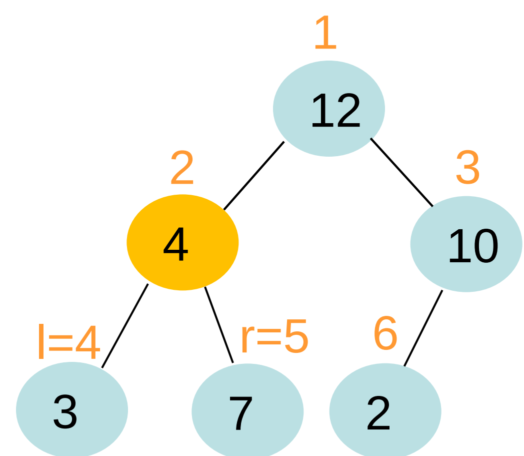
Heapify(A,1)

Datenstrukturen

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

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)

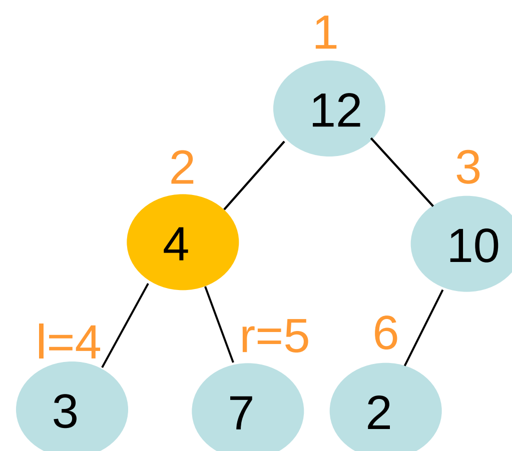


Heapify(A,1)

Datenstrukturen

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

```
Heapify(A,i)
l ← left(i)
r ← right(i)
if l≤heap-size[A] and A[l]>A[i] then largest ← l
else largest ← i
if r≤heap-size[A] and A[r]>A[largest] then largest ← r
if largest≠i then A[i] ↔ A[largest]
    Heapify(A,largest)
```



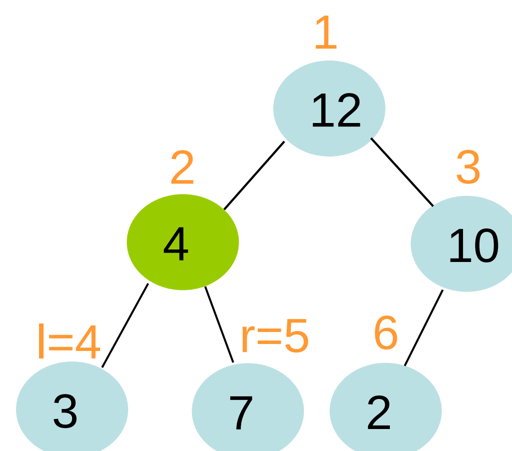
Heapify(A,1)

Datenstrukturen

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

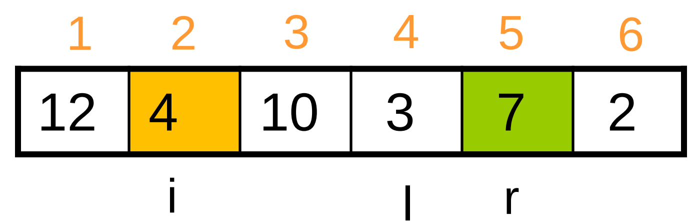
Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



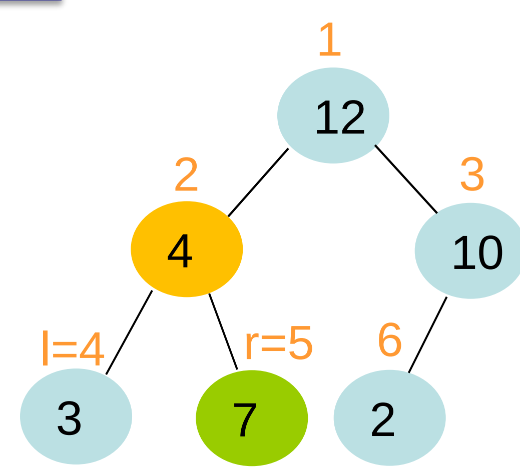
Heapify(A,1)

Datenstrukturen



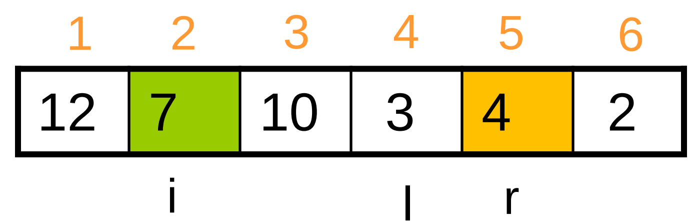
Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



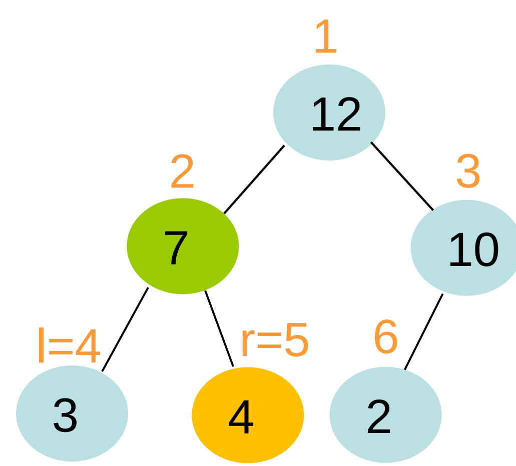
Heapify(A,1)

Datenstrukturen



Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



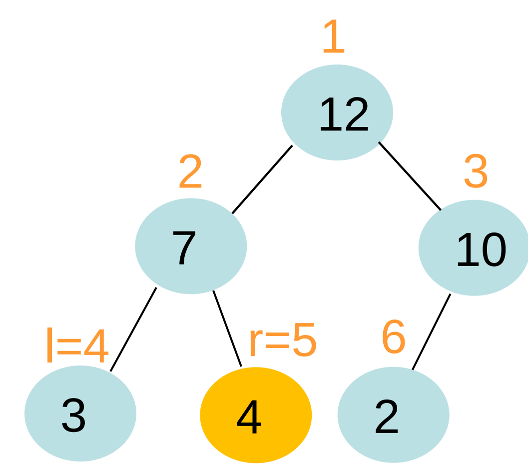
Heapify(A,1)

Datenstrukturen

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

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. $\boxed{\text{Heapify(A,largest)}}$



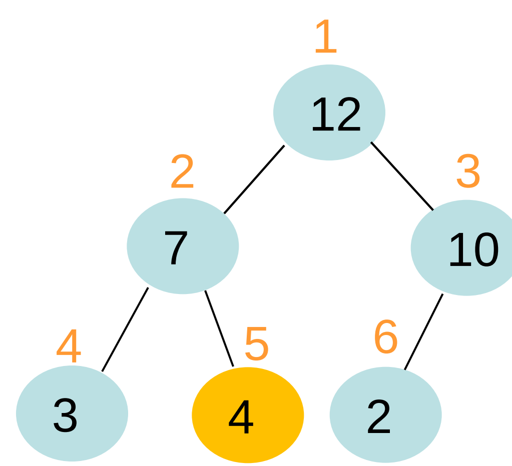
Heapify(A,1)

Datenstrukturen

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

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



Heapify(A,1)

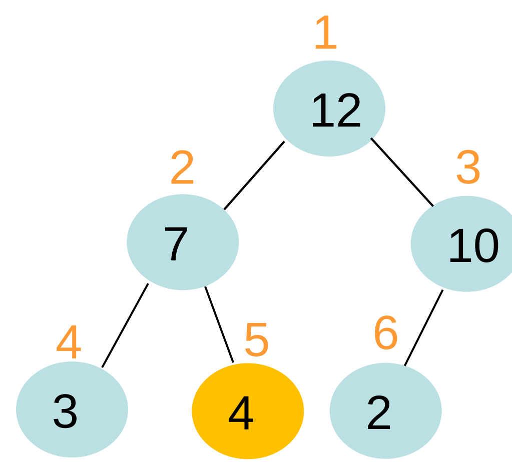
Datenstrukturen

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

l=10

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



Heapify(A,1)

Datenstrukturen

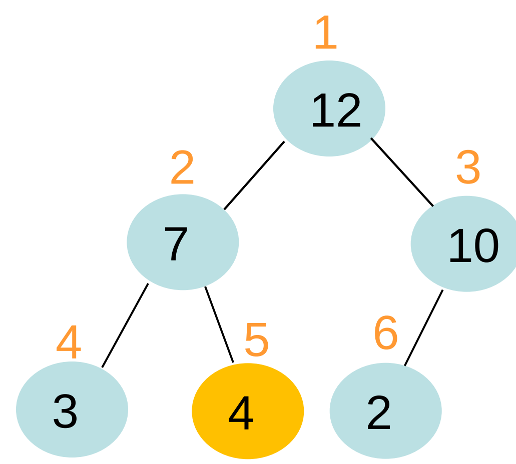
1	2	3	4	5	6
12	7	10	3	4	2
				i	

l=10

r=11

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



Heapify(A,1)

Datenstrukturen

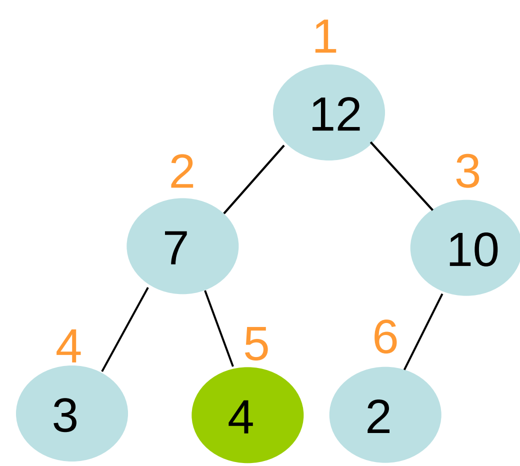
Heapify(A,i)

l ← left(i)
r ← right(i)
if l≤heap-size[A] and A[l]>A[i] **then** largest ← l
else largest ← i
if r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
if largest≠i **then** A[i] ↔ A[largest]
    Heapify(A,largest)

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

l=10

r=11



Heapify(A,1)

Datenstrukturen

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

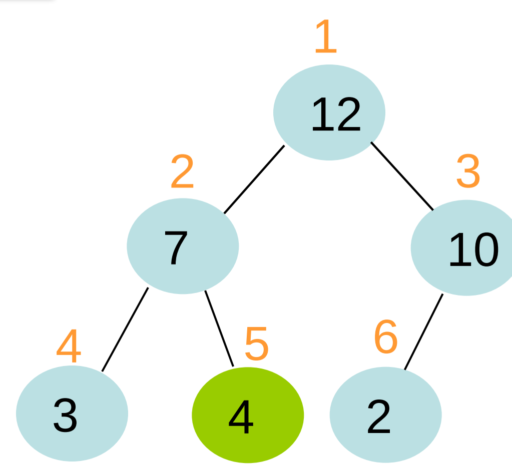
i

l=10

r=11

Heapify(A,i)

1. l ← left(i)
2. r ← right(i)
3. **if** l≤heap-size[A] and A[l]>A[i] **then** largest ← l
4. **else** largest ← i
5. **if** r≤heap-size[A] and A[r]>A[largest] **then** largest ← r
6. **if** largest≠i **then** A[i] ↔ A[largest]
7. Heapify(A,largest)



Heapify(A,1)

Datenstrukturen

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

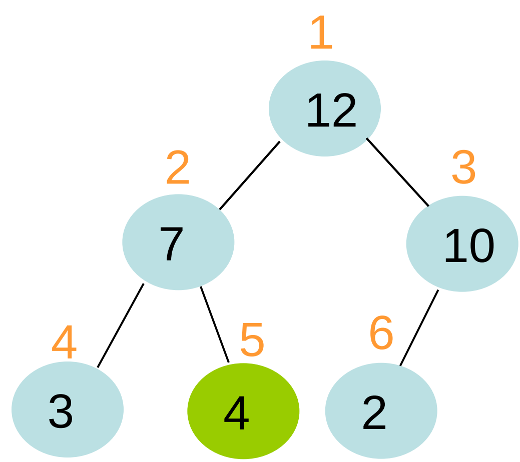
i

l=10

r=11

Heapify(A,i)

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



Heapify(A,1)

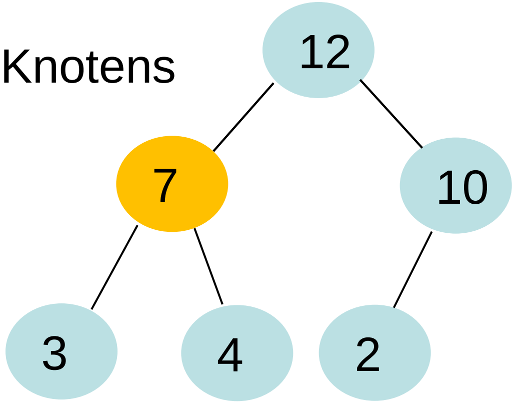
Datenstrukturen

Definition

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

Beispiel

Höhe des Knotens
ist 1



Datenstrukturen

Satz 44

- Die Laufzeit von 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 $\le c\cdot(h+1)$ 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.

Datenstrukturen

Satz 44

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

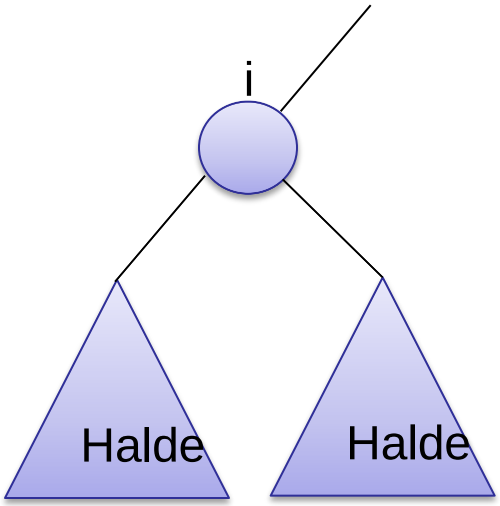
Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit c(h+1).
- (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 c(h+1)
- Restliche Laufzeit ≤c
- $\Rightarrow$ Laufzeit maximal ch+c = c(h+2).

Datenstrukturen

Satz 45

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Haldeneigenschaft besitzen, dann ist diese nach der Operation Heapify(A,i) für den Unterbaum von i erfüllt.



Datenstrukturen

Satz 45

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Haldeneigenschaft besitzen, dann ist diese nach der Operation 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.

Datenstrukturen

Satz 46

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Haldeneigenschaft besitzen, dann ist diese nach der Operation Heapify(A,i) für den Unterbaum von i erfüllt.

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

Datenstrukturen

Satz 46

- Wenn die Unterbäume des rechten bzw. linken Kindes von i die Haldeneigenschaft besitzen, dann ist diese nach der Operation Heapify(A,i) für den Unterbaum von i erfüllt.

- *Beweis*
- A[l] ist Maximum (A[r] analog):
- Unterbäume von l und r sind Halden $\Rightarrow$ 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 Haldeneigenschaft
- Bei i ist Max. aller Elemente gespeichert und rechter Unterbaum erfüllt Haldeneigenschaft $\Rightarrow$ Unterbaum von i ist Halde

Datenstrukturen

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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

Datenstrukturen

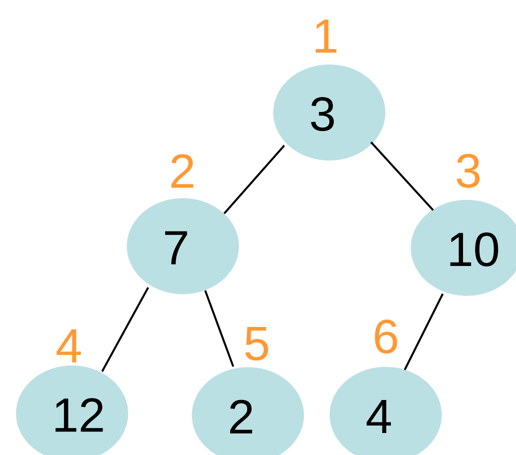
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Datenstrukturen

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

Aufbau einer Halde

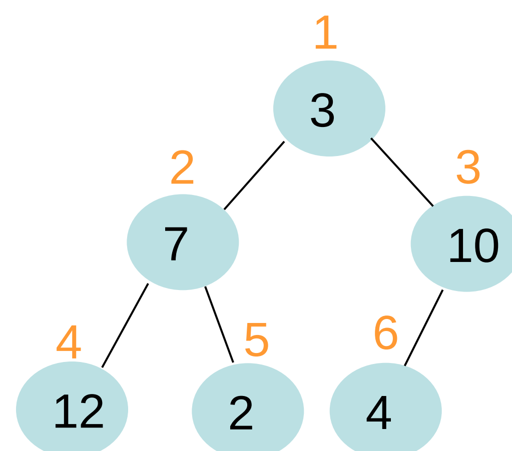
- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

$\boxed{\text{heap-size} \leftarrow \text{length[A]}}$

for i $\leftarrow \lfloor$length[A]/2$\rfloor$ **downto** 1 **do**

    Heapify(A,i)



Datenstrukturen

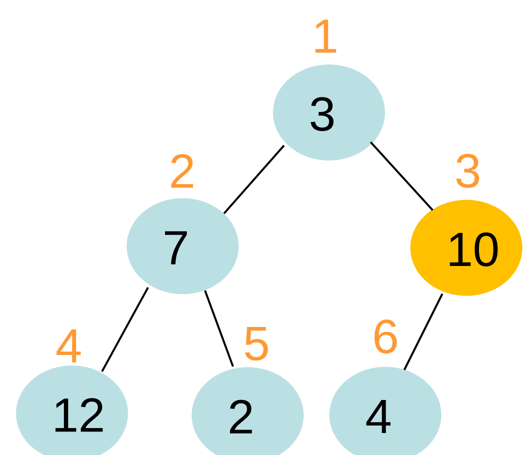
1	2	3	4	5	6
3	7	10	12	2	4
		i			

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Datenstrukturen

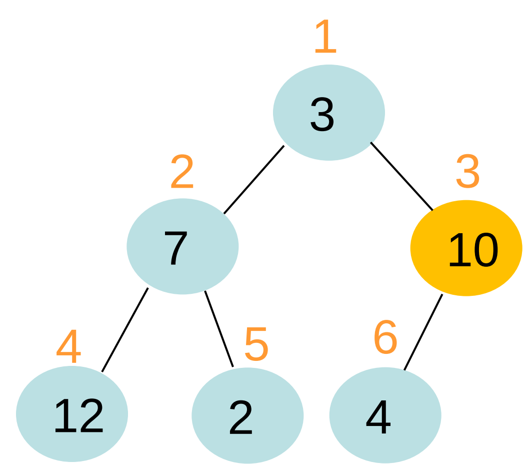
1	2	3	4	5	6
3	7	10	12	2	4
		i			

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Datenstrukturen

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

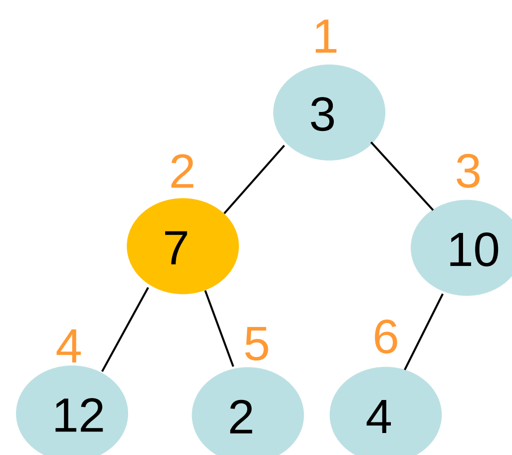
i

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size ← length[A]
2. **for** i ← ⌊length[A]/2⌋ **downto** 1 **do**
3. Heapify(A,i)



Datenstrukturen

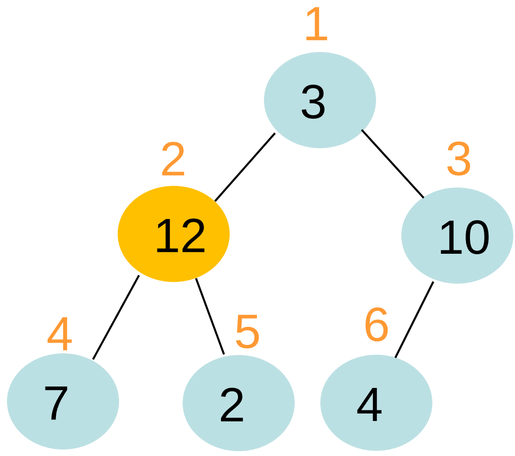
1	2	3	4	5	6
3	12	10	7	2	4
	i				

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Datenstrukturen

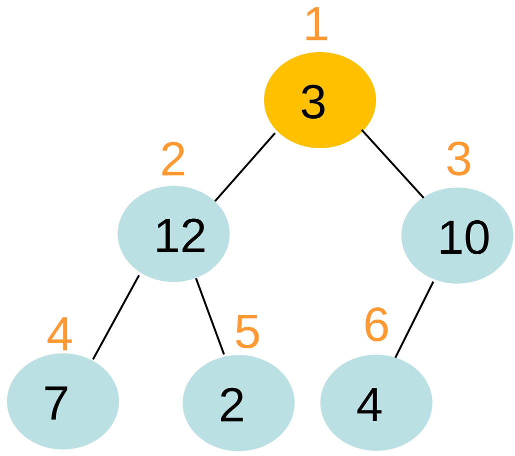
1	2	3	4	5	6
3	12	10	7	2	4
i					

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

1. heap-size ← length[A]
2. **for** i ← ⌊length[A]/2⌋ **downto** 1 **do**
3. Heapify(A,i)



Datenstrukturen

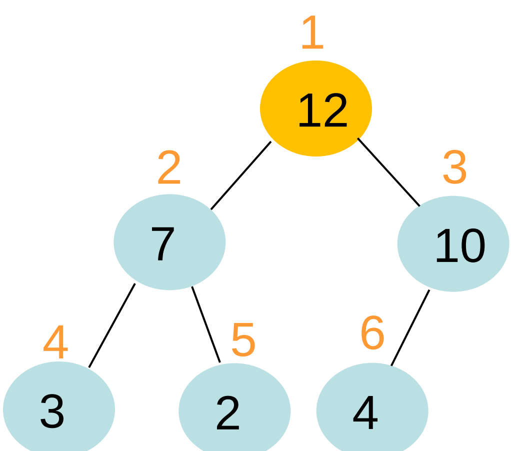
1	2	3	4	5	6
12	7	10	3	2	4
i					

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Datenstrukturen

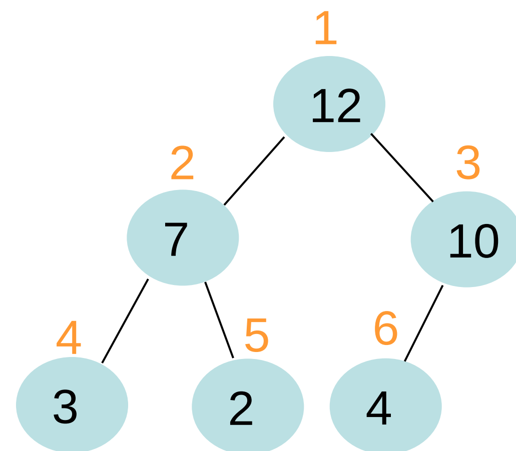
1	2	3	4	5	6
12	7	10	3	2	4
i					

Aufbau einer Halde

- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Datenstrukturen

Satz 47

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

- *Beweis*
- Korrektheit:
  (Inv.) Unterbäume der Knoten größer als i besitzen Haldeneigenschaft
- 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 Haldeneigenschaft für die Wurzel
- $\Rightarrow$ Build-Heap erzeugt Halde.

Datenstrukturen

Satz 47

- Mit Hilfe des Algorithmus Build-Heap kann man eine Halde 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).

Datenstrukturen

Satz 47

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

- *Beweis*
- Schärfere Laufzeitanalyse:
- Beobachtung: In einer Halde 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 $H=\lfloor \log n \rfloor$:

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

Datenstrukturen

Satz 47

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

- *Beweis*

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

- Es gilt $\sum_{h=0}^{\infty}\frac{h+1}{2^{h}} = O(1)$

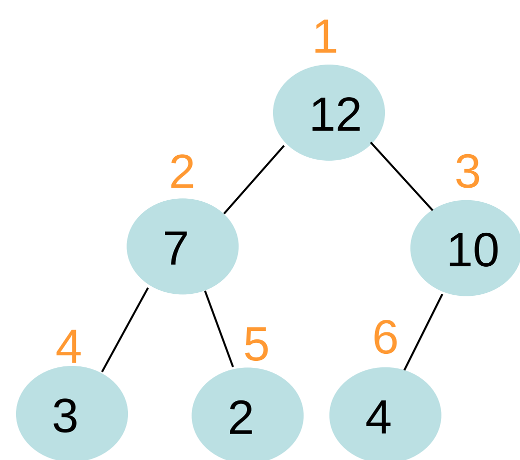
- Somit folgt eine Laufzeit von O(n).

Datenstrukturen

Heap-Extract-Max(A)

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

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

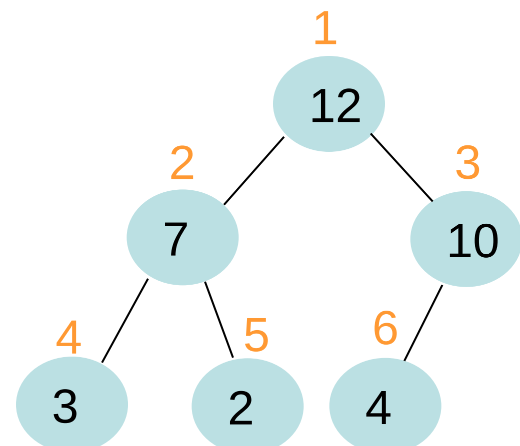


Datenstrukturen

Heap-Extract-Max(A)

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

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



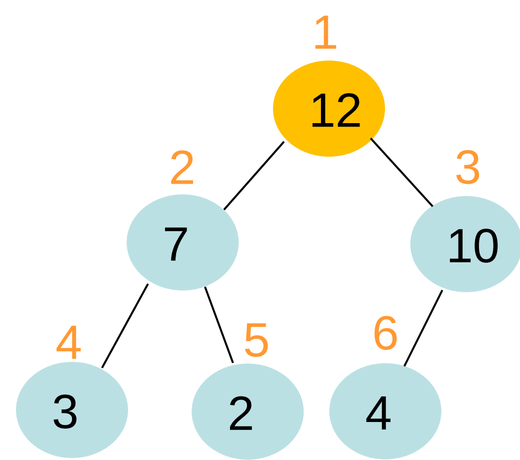
Datenstrukturen

Heap-Extract-Max(A)

1. **if** heap-size[A] < 1 **then** error „Kein Element vorhanden!“
2. $\boxed{\text{max} \leftarrow \text{A[1]}}$
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



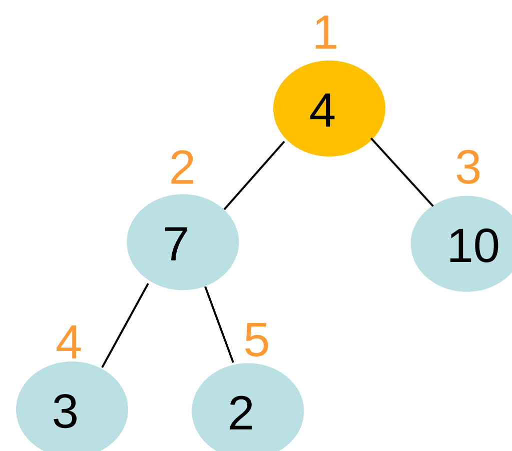
Datenstrukturen

Heap-Extract-Max(A)

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

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

max =12



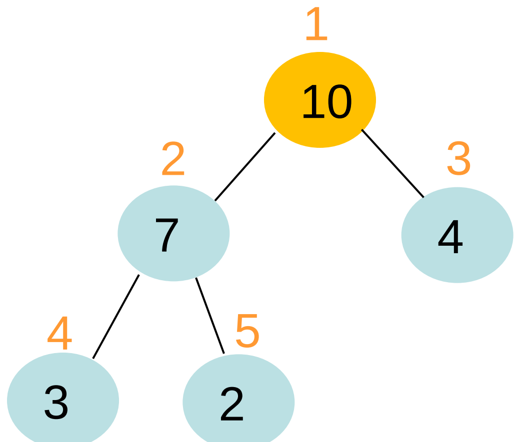
Datenstrukturen

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. $\boxed{\text{Heapify(A,1)}}$
6. **return** max

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

max =12



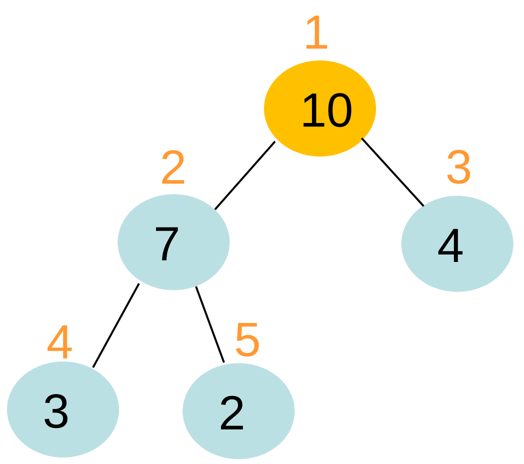
Datenstrukturen

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
10	7	4	3	2	4

max =12

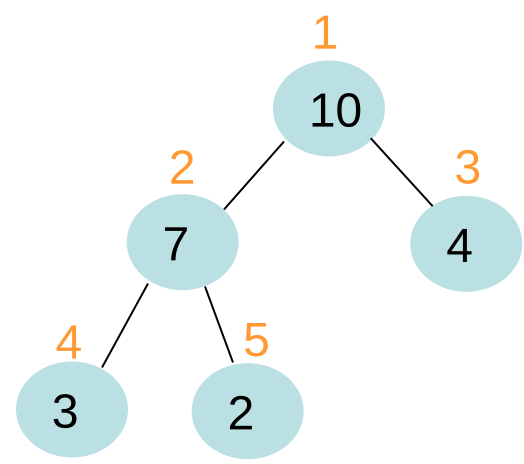


Datenstrukturen

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
10	7	4	3	2	4



Laufzeit

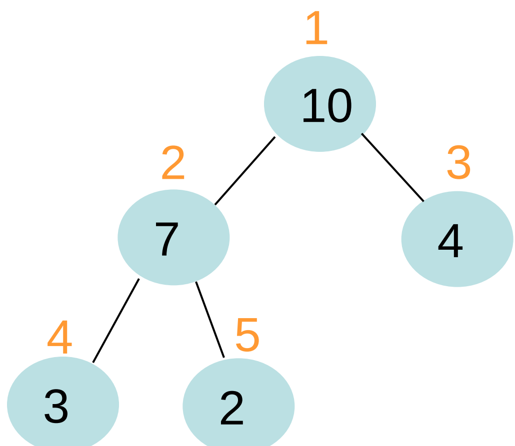
- O(log n)

Datenstrukturen

Heap-Insert(A,key)

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

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

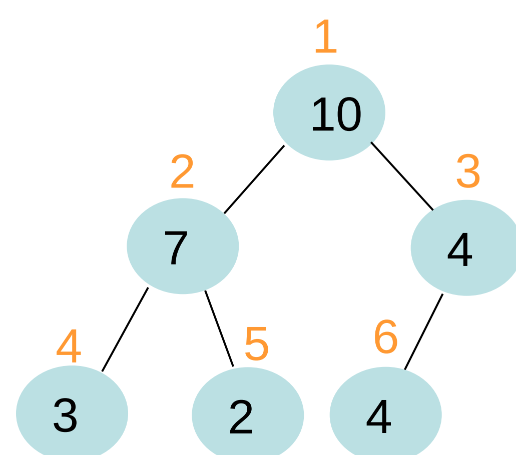


Datenstrukturen

Heap-Insert(A,key)

1. heap-size[A] ← heap-size[A]+1
2. i ← heap-size[A]
3. **while** i>1 and A[Parent(i)] < key **do**
4. A[i] ← A[Parent(i)]
5. i ← Parent(i)
6. A[i] ← key

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

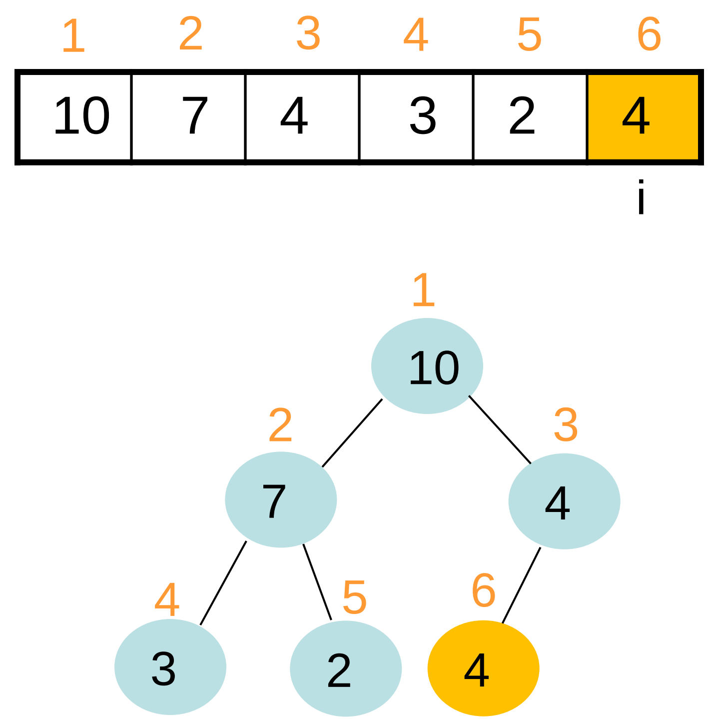


Heap-Insert(A,11)

Datenstrukturen

Heap-Insert(A,key)

1. heap-size[A] ← heap-size[A]+1
2. i ← heap-size[A]
3. **while** i>1 and A[Parent(i)] < key **do**
4.   A[i] ← A[Parent(i)]
5.   i ← Parent(i)
6. A[i] ← key



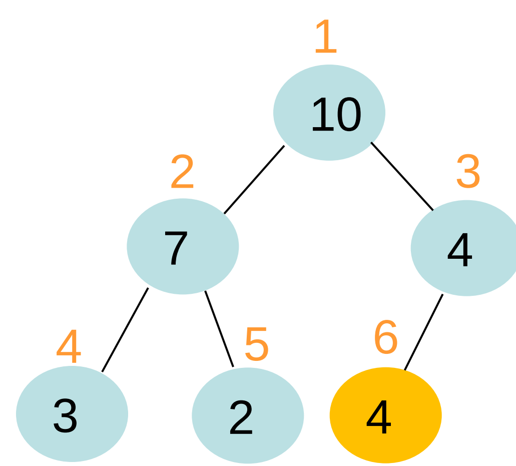
Heap-Insert(A,11)

Datenstrukturen

Heap-Insert(A,key)

1. heap-size[A] ← heap-size[A]+1
2. i ← heap-size[A]
3. **while** i>1 and A[Parent(i)] < key **do**
4. A[i] ← A[Parent(i)]
5. i ← Parent(i)
6. A[i] ← key

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

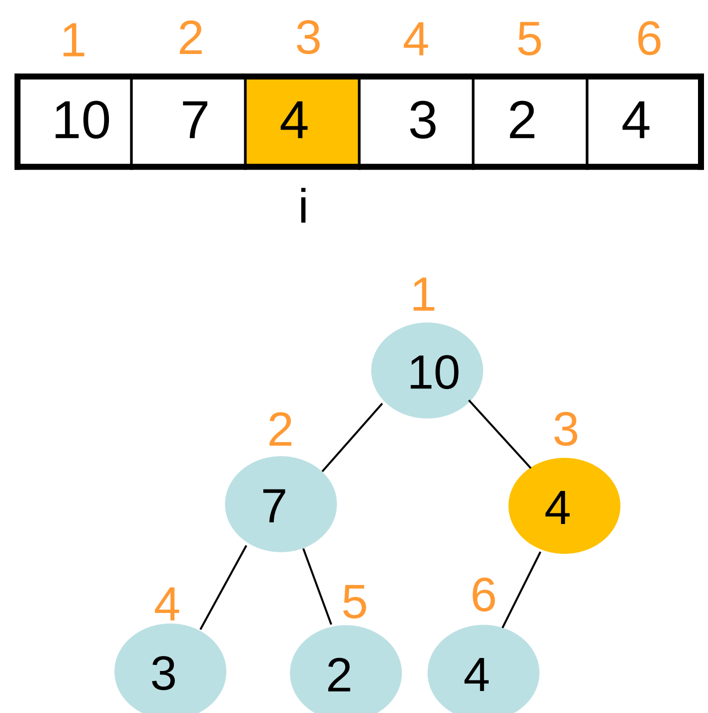


Heap-Insert(A,11)

Datenstrukturen

Heap-Insert(A,key)

1. heap-size[A] ← heap-size[A]+1
2. i ← heap-size[A]
3. **while** i>1 and A[Parent(i)] < key **do**
4. A[i] ← A[Parent(i)]
5. i ← Parent(i)
6. A[i] ← key

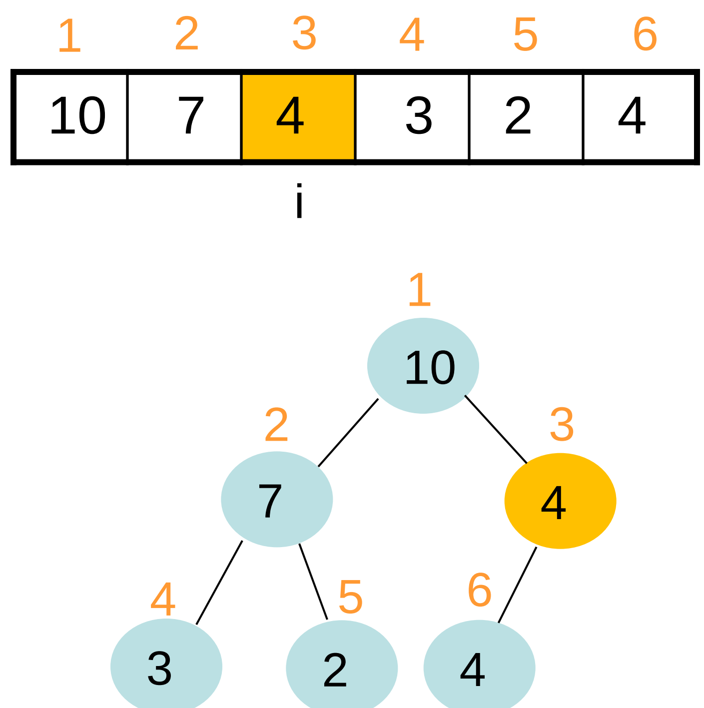


Heap-Insert(A,11)

Datenstrukturen

Heap-Insert(A,key)

1. heap-size[A] ← heap-size[A]+1
2. i ← heap-size[A]
3. **while** i>1 and A[Parent(i)] < key **do**
4. A[i] ← A[Parent(i)]
5. i ← Parent(i)
6. A[i] ← key

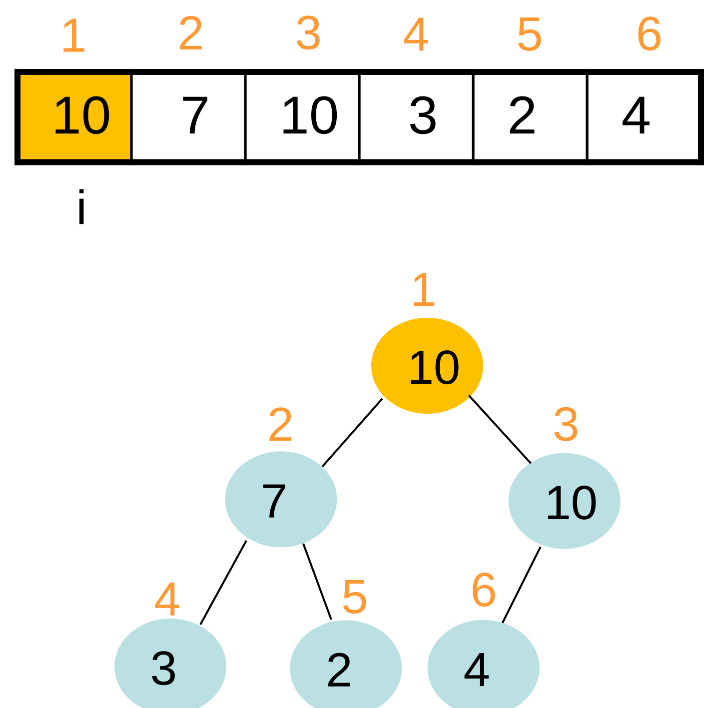


Heap-Insert(A,11)

Datenstrukturen

Heap-Insert(A,key)

1. heap-size[A] ← heap-size[A]+1
2. i ← heap-size[A]
3. **while** i>1 and A[Parent(i)] < key **do**
4. A[i] ← A[Parent(i)]
5. i ← Parent(i)
6. A[i] ← key

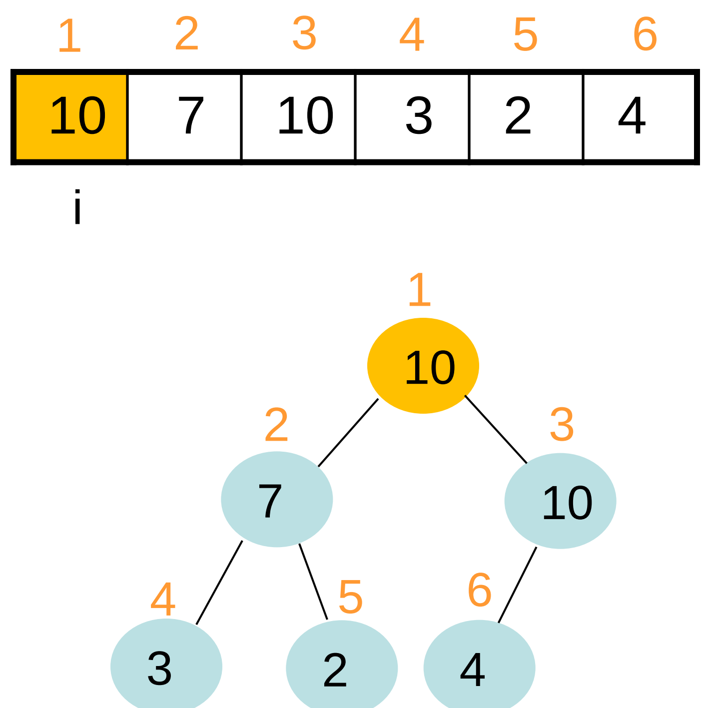


Heap-Insert(A,11)

Datenstrukturen

Heap-Insert(A,key)

1. heap-size[A] ← heap-size[A]+1
2. i ← heap-size[A]
3. **while** i>1 and A[Parent(i)] < key **do**
4. A[i] ← A[Parent(i)]
5. i ← Parent(i)
6. A[i] ← key



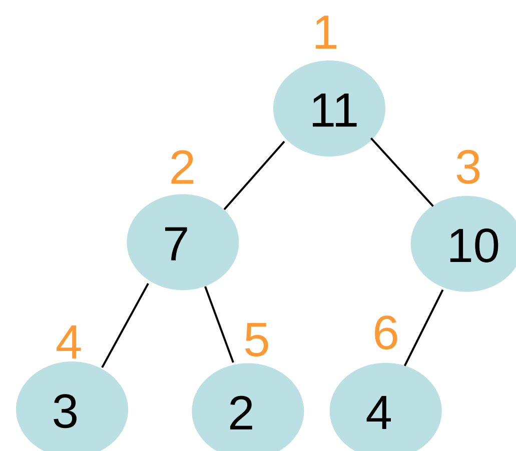
Heap-Insert(A,11)

Datenstrukturen

Heap-Insert(A,key)

```
heap-size[A] ← heap-size[A]+1
i ← heap-size[A]
while i>1 and A[Parent(i)] < key do
    A[i] ← A[Parent(i)]
    i ← Parent(i)
A[i] ← key
```

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



Laufzeit

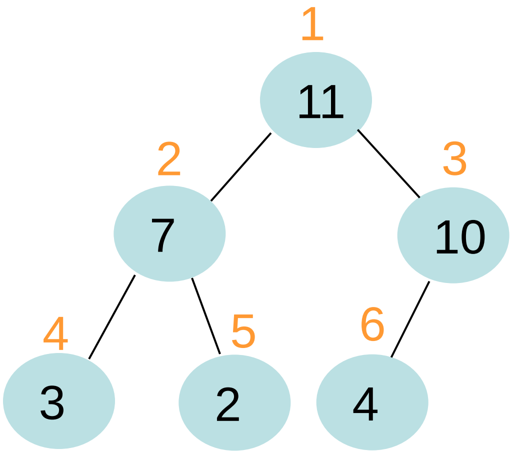
- O(log n)

Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

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

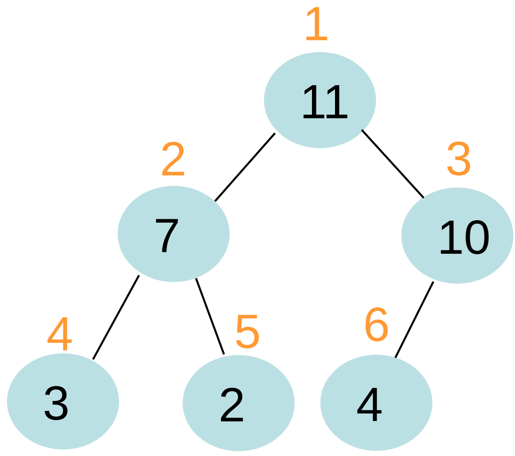


Datenstrukturen

Heapsort(A)

```
Build-Heap(A)
for i ← length[A] downto 2 do
    A[1] ↔ A[i]
    heap-size[A] ← heap-size[A]-1
    Heapify(A,1)
```

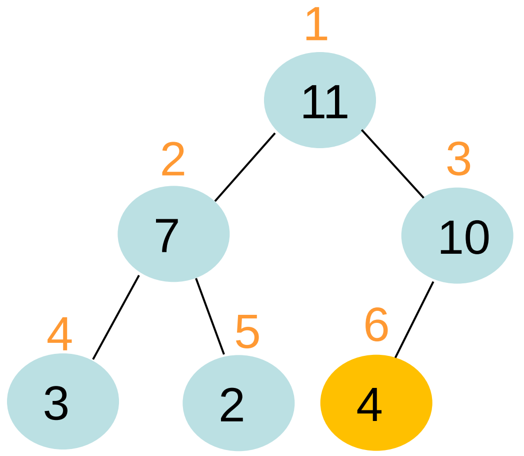
1	2	3	4	5	6
11	7	10	3	2	4



Datenstrukturen

Heapsort(A)
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

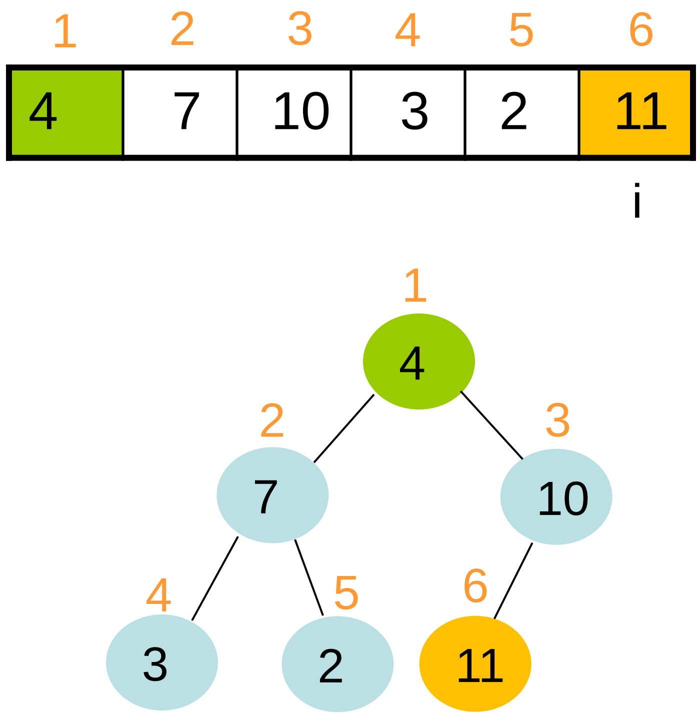
1	2	3	4	5	6
11	7	10	3	2	4
					i



Datenstrukturen

Heapsort(A)

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

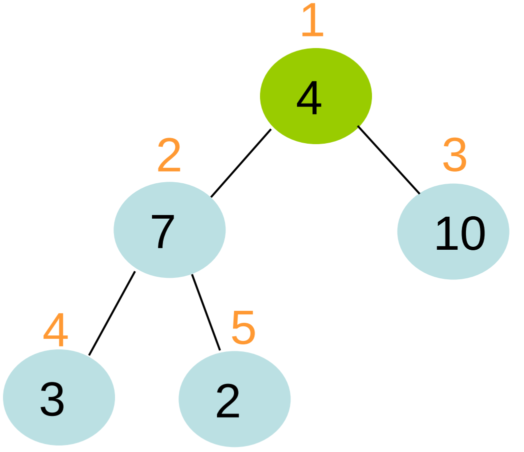


Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

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

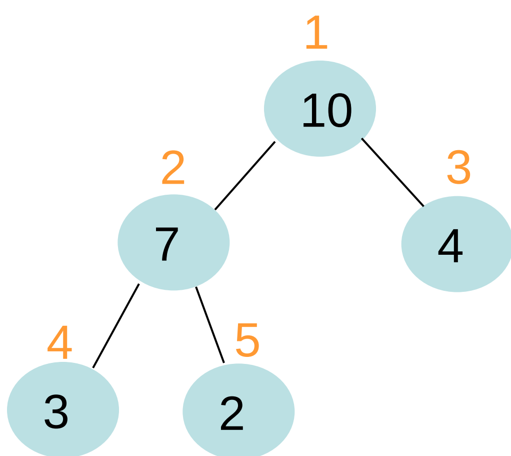


Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

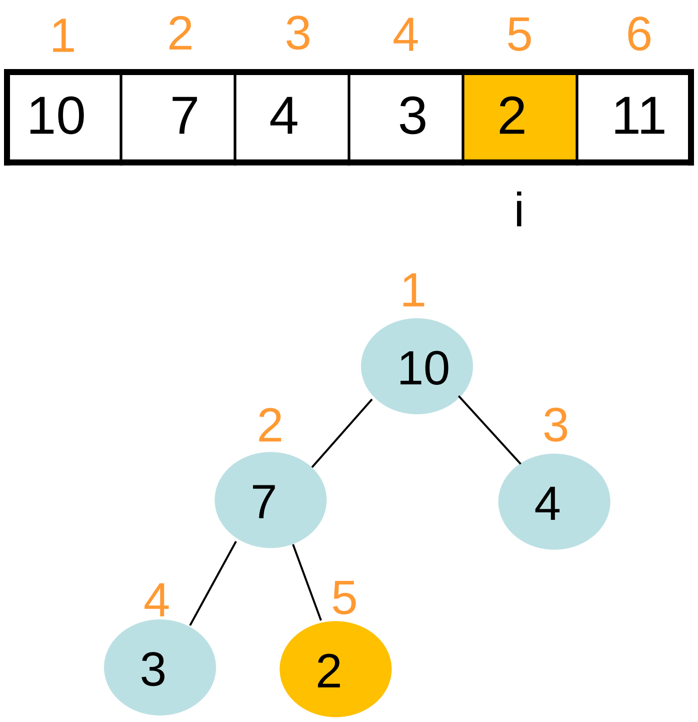
1	2	3	4	5	6
10	7	4	3	2	11
					i



Datenstrukturen

Heapsort(A)

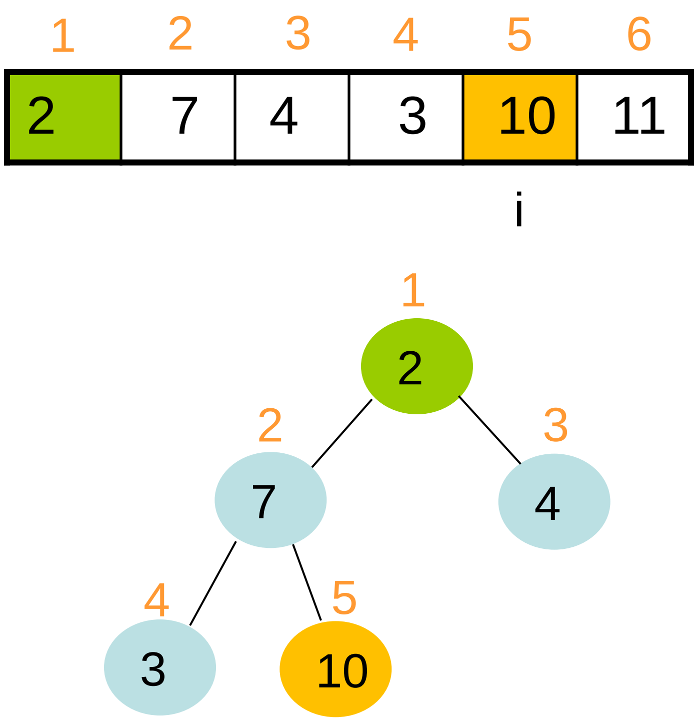
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

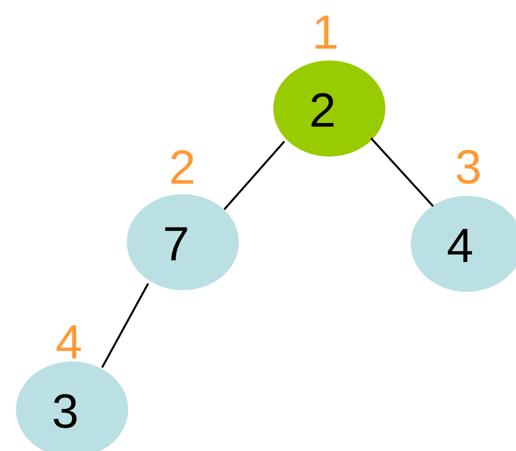


Datenstrukturen

Heapsort(A)

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

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

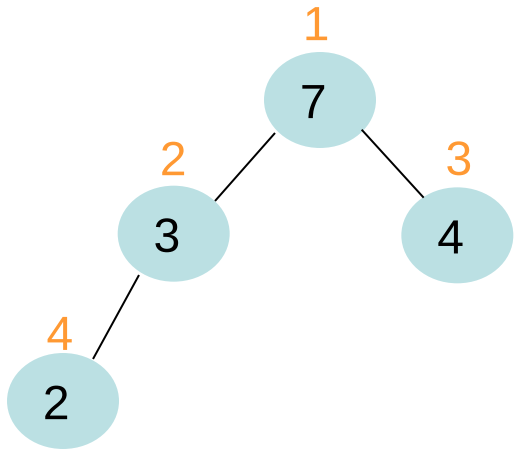


Datenstrukturen

Heapsort(A)

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

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

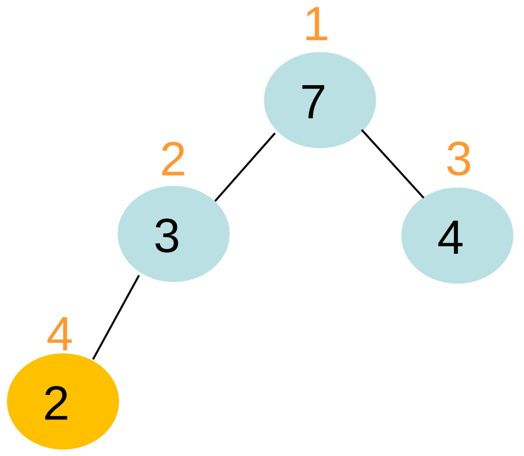


Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

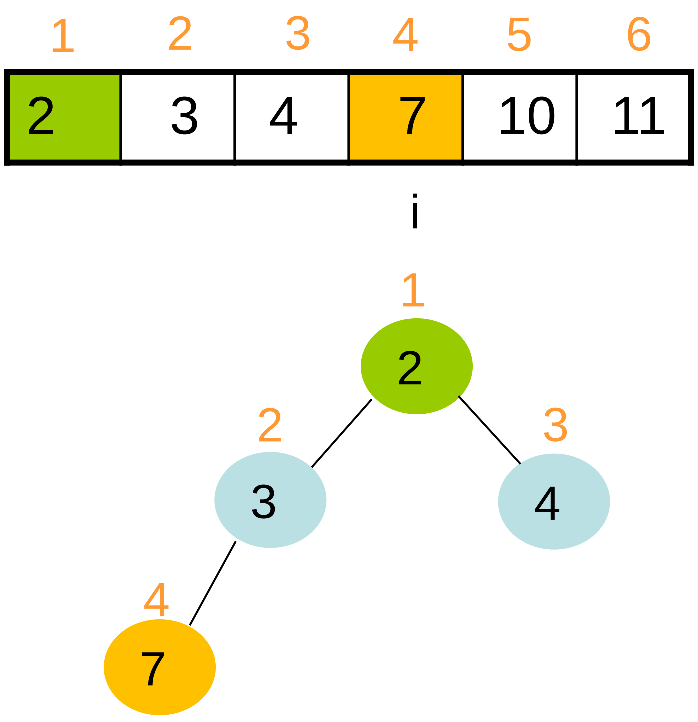
1	2	3	4	5	6
7	3	4	2	10	11
			i		



Datenstrukturen

Heapsort(A)

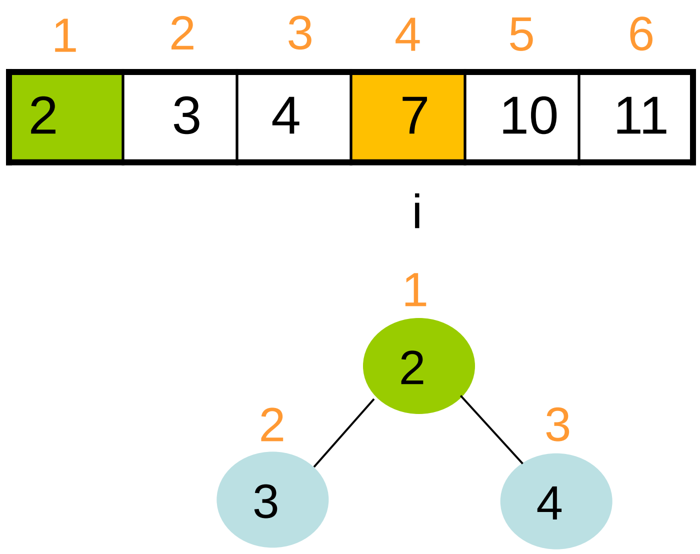
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. $\boxed{\text{A[1]} \leftrightarrow \text{A[i]}}$
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

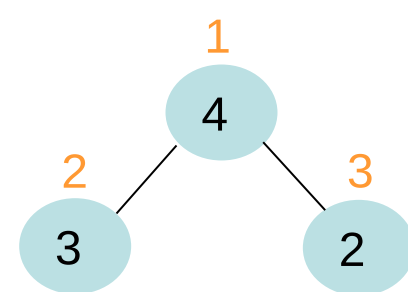


Datenstrukturen

Heapsort(A)

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

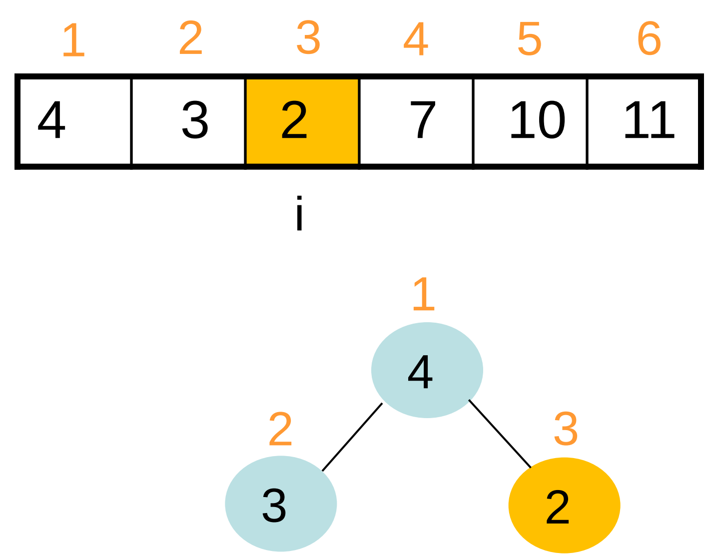
1	2	3	4	5	6
4	3	2	7	10	11
			i		



Datenstrukturen

Heapsort(A)

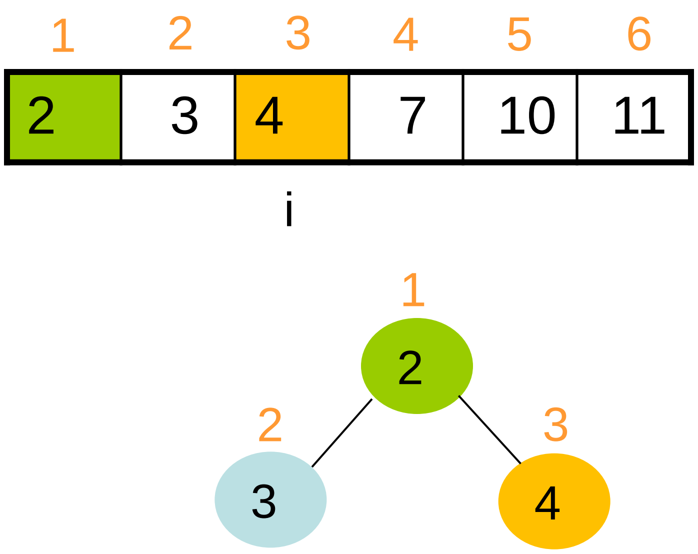
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Datenstrukturen

Heapsort(A)

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

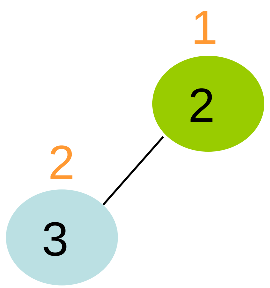


Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

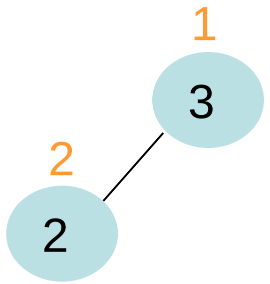
1	2	3	4	5	6
2	3	4	7	10	11
		i			



Datenstrukturen

Heapsort(A)
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

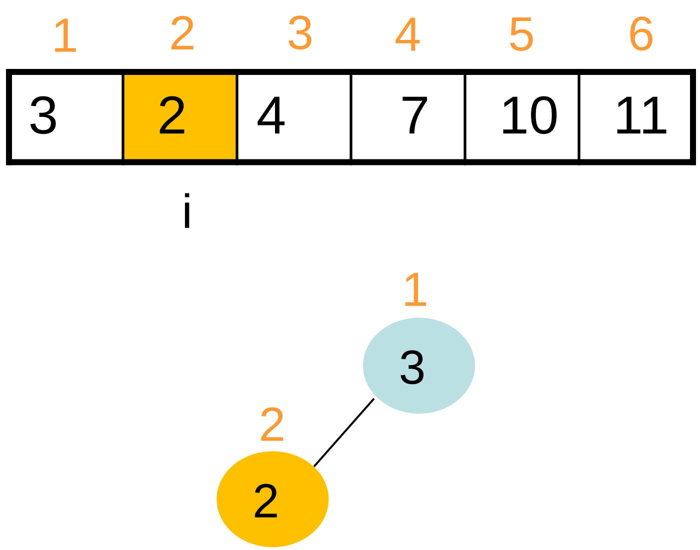
1	2	3	4	5	6
3	2	4	7	10	11
		i			



Datenstrukturen

Heapsort(A)

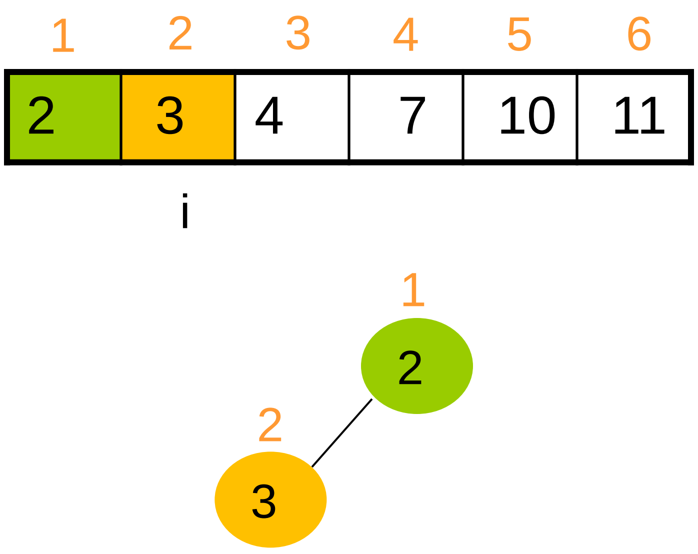
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Datenstrukturen

Heapsort(A)

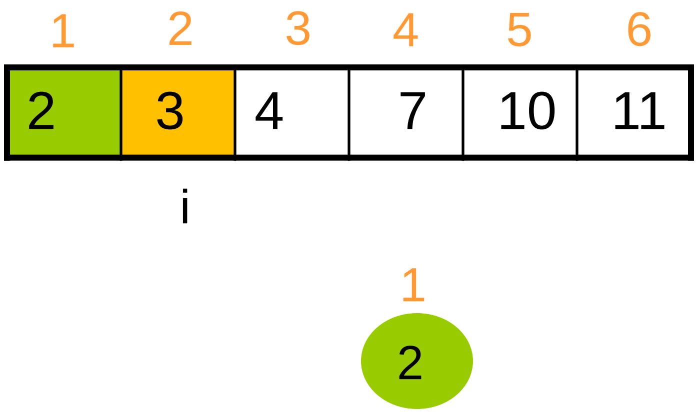
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. $\boxed{\text{A[1]} \leftrightarrow \text{A[i]}}$
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

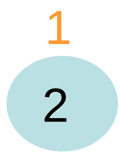


Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

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



Datenstrukturen

Heapsort(A)

1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

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

Laufzeit

- O(n log n)

Datenstrukturen

Zusammenfassung (Halden)

- Einfügen, Löschen, Maximum extrahieren in O(log n) Zeit
- Sortieren mit Hilfe von Halden in O(n log n)
- Heapsort braucht keinen zusätzlichen Speicherplatz
- Einfache Implementierung
- Bespiel für Kombination von Datenstruktur und Algorithmus