Prioritätenschlangen/Halden/Heaps

Prioritätenschlangen

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

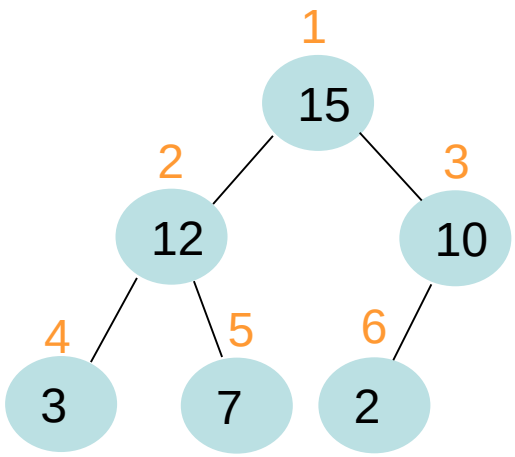
Anwendungen

- Ereignisgesteuerte Simulationen
- Sortieren mit Heapsort

Bäume als Arrays – Datenstruktur

Binäre Halden / Heaps

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

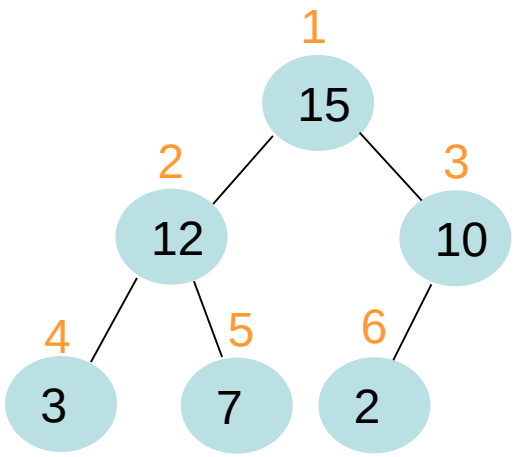
- **return** ⌊i/2⌋

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

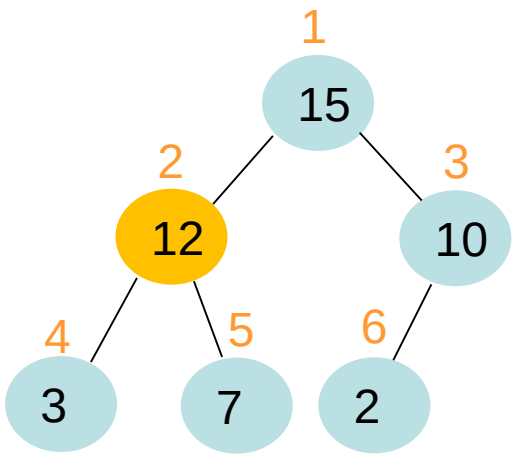
Navigation

- Wurzel ist A[1]

Parent(i)
1. **return** ⌊i/2⌋

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

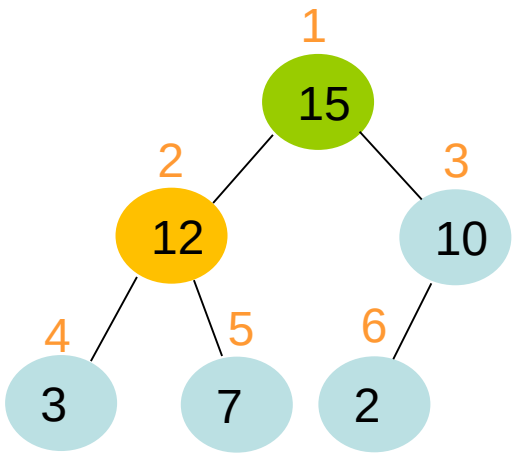
Navigation

- Wurzel ist A[1]

Parent(i)
1. **return** ⌊i/2⌋

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

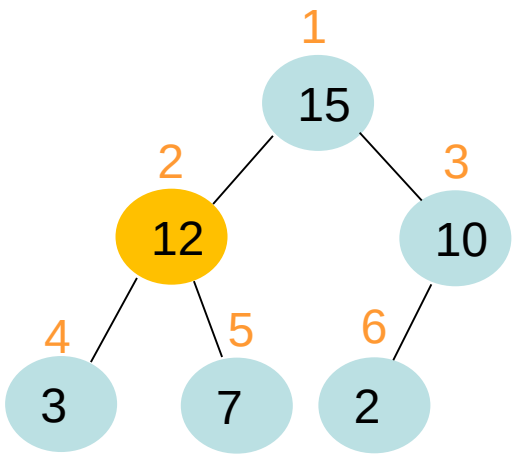
Navigation

- Wurzel ist A[1]

Parent(i)
1. **return** ⌊i/2⌋

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

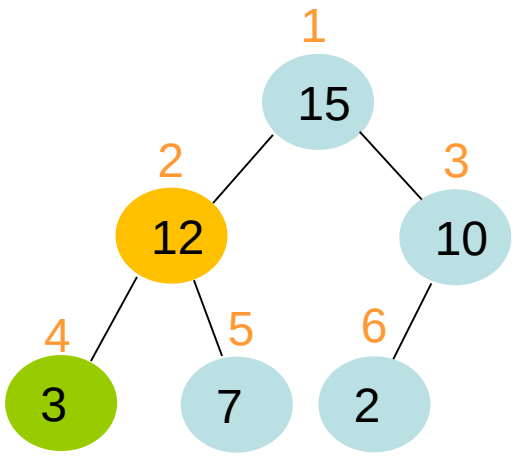
Navigation

- Wurzel ist A[1]

Parent(i)
1. **return** ⌊i/2⌋

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

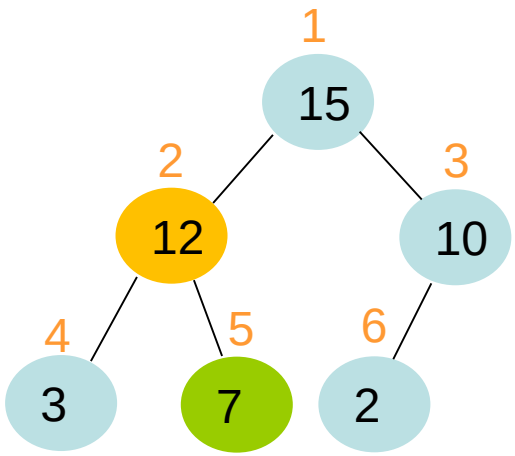
Navigation

- Wurzel ist A[1]

Parent(i)
1. **return** ⌊i/2⌋

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

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



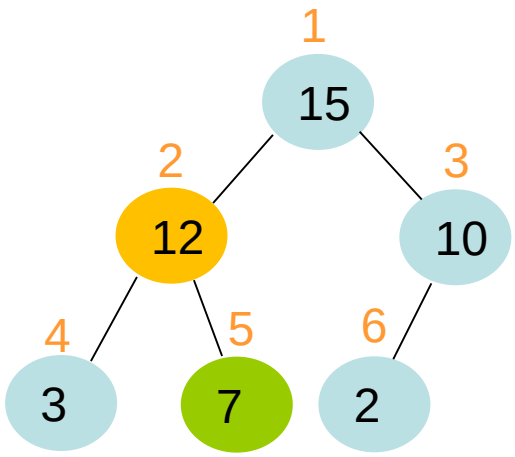
1	2	3	4	5	6
15	12	10	3	7	2

Max Heap – Datenstruktur

Max Heap Eigenschaft / Max Heap property

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

(Min Heap Eigenschaft / Min Heap property analog mit ≤)

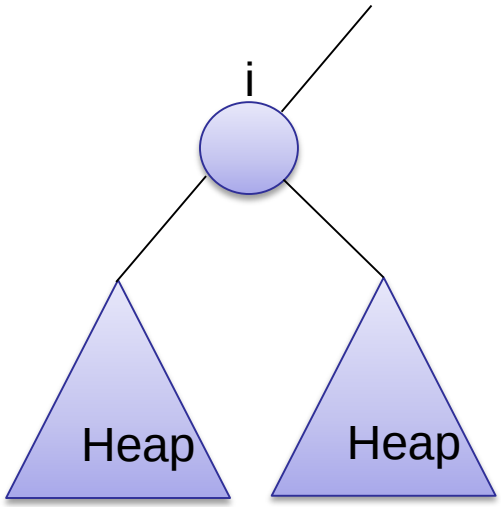


1	2	3	4	5	6
15	12	10	3	7	2

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

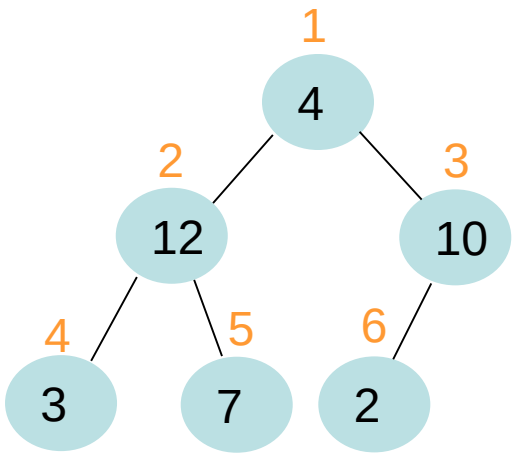


Heapify – Operation

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)



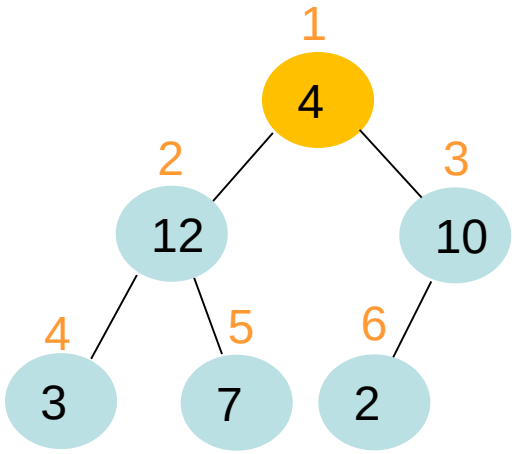
Heapify – Operation

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)



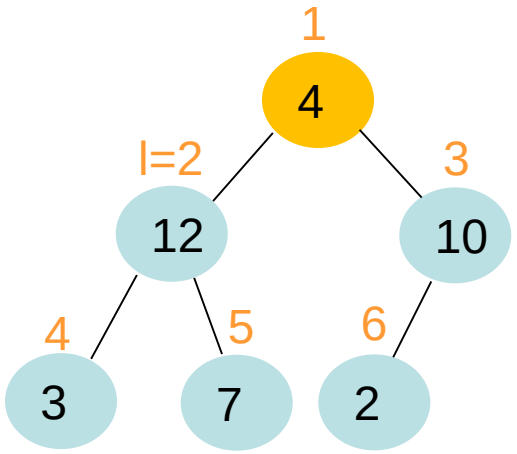
Heapify(A,1)

Heapify – Operation

1	2	3	4	5	6
4	12	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)



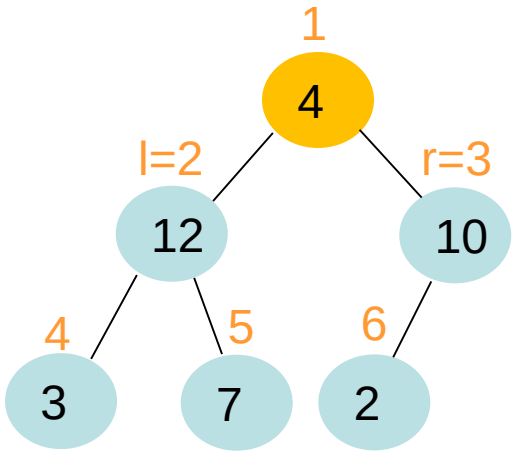
Heapify(A,1)

Heapify – Operation

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)



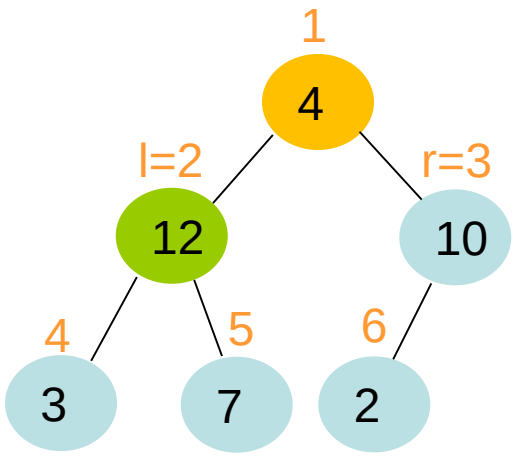
Heapify(A,1)

Heapify – Operation

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)



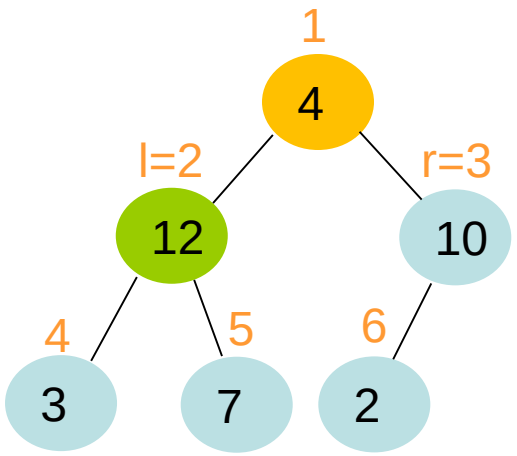
Heapify(A,1)

Heapify – Operation

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

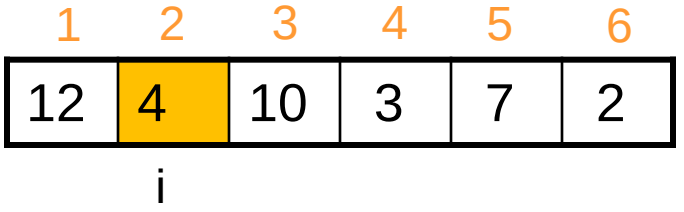
Heapify(A,i)

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)



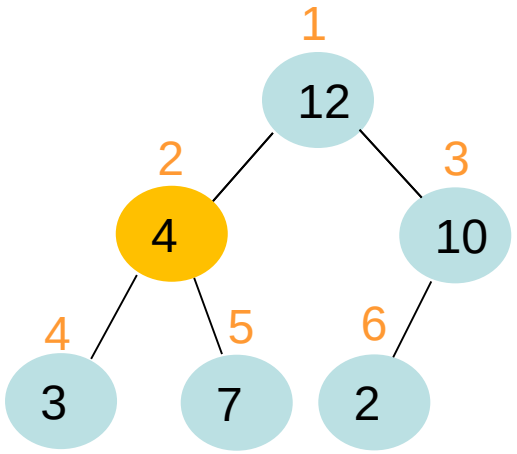
Heapify(A,1)

Heapify – Operation



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)



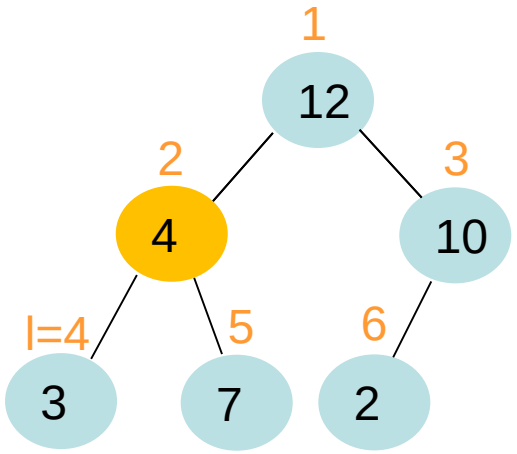
Heapify(A,1)

Heapify – Operation

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

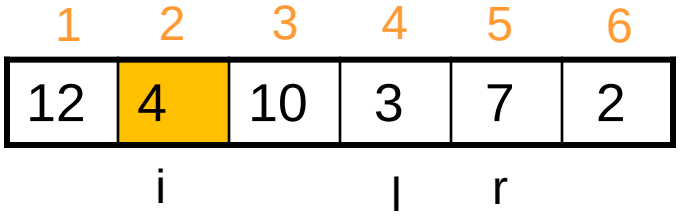
Heapify(A,i)

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)



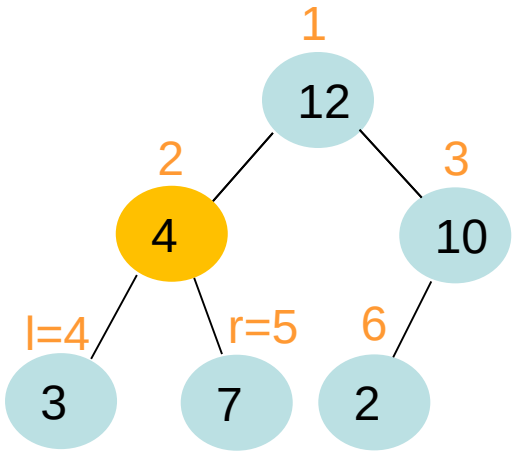
Heapify(A,1)

Heapify – Operation



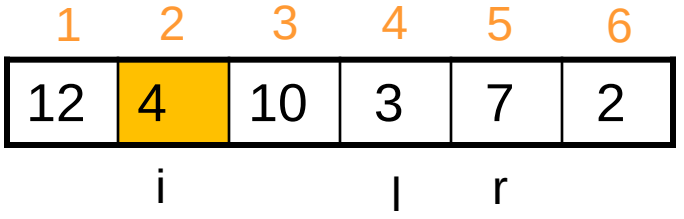
Heapify(A,i)

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)



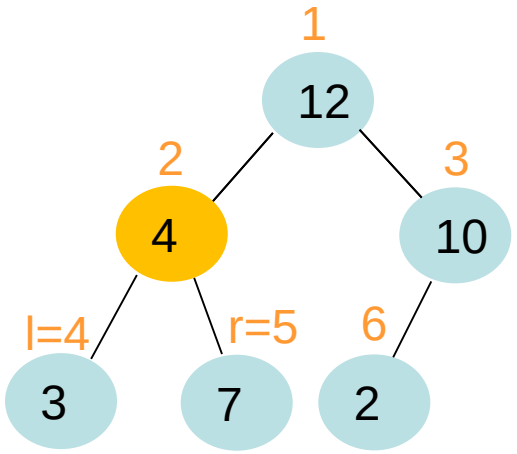
Heapify(A,1)

Heapify – Operation



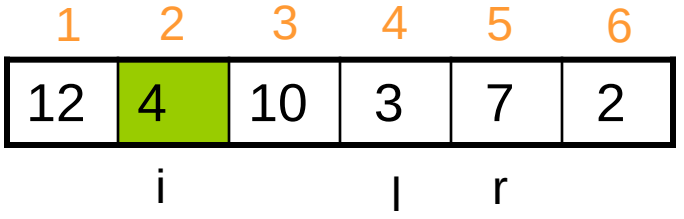
Heapify(A,i)

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)



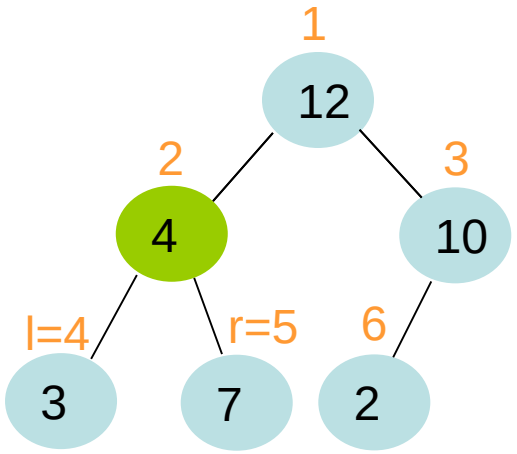
Heapify(A,1)

Heapify – Operation



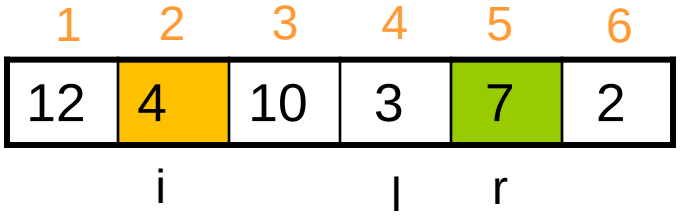
Heapify(A,i)

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)



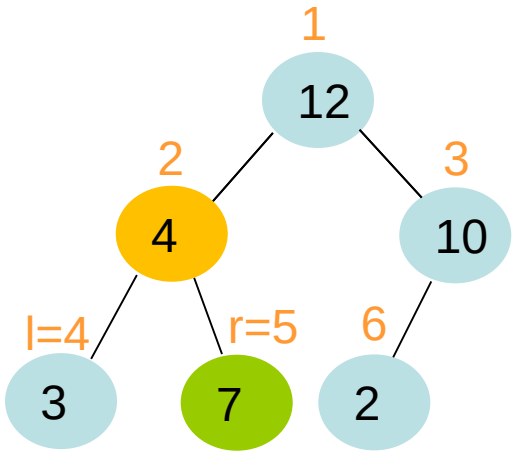
Heapify(A,1)

Heapify – Operation



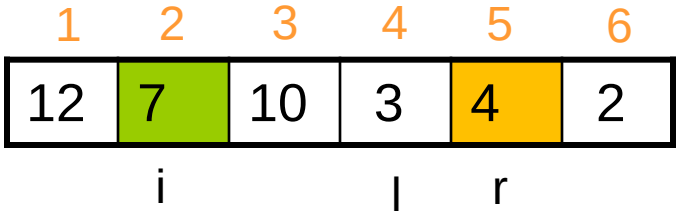
Heapify(A,i)

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)



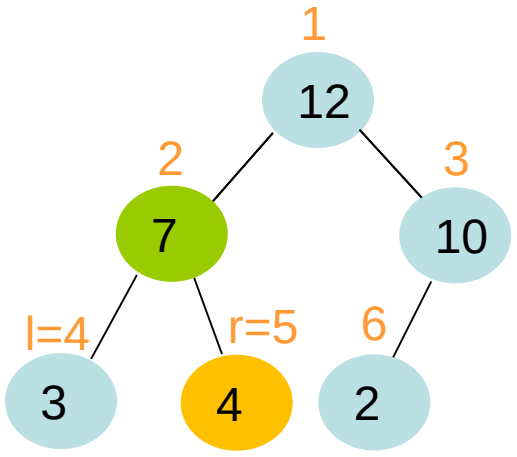
Heapify(A,1)

Heapify – Operation



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)



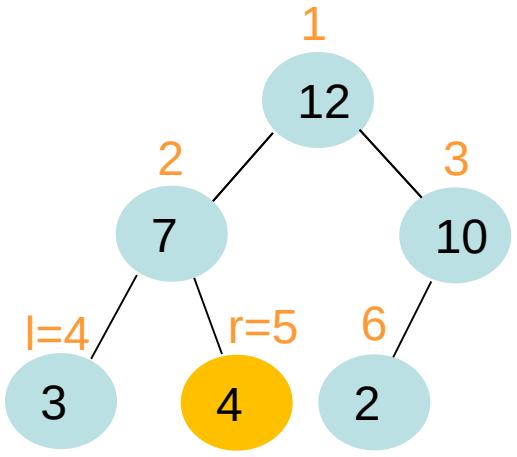
Heapify(A,1)

Heapify – Operation

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. Heapify(A,largest)



Heapify(A,1)

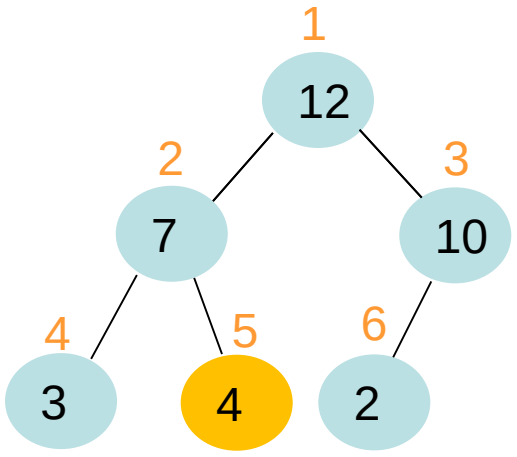
Heapify – Operation

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)

Heapify – Operation

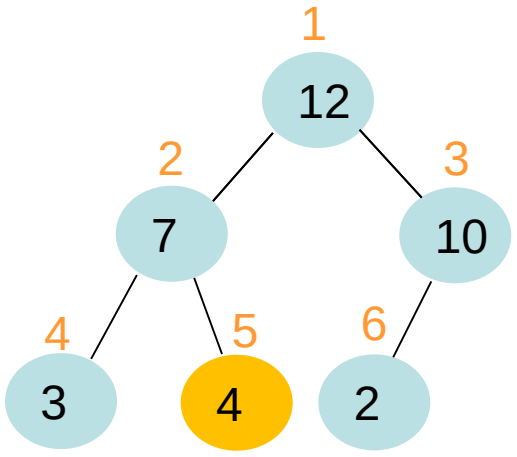
1	2	3	4	5	6
12	7	10	3	4	2

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)

Heapify – Operation

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

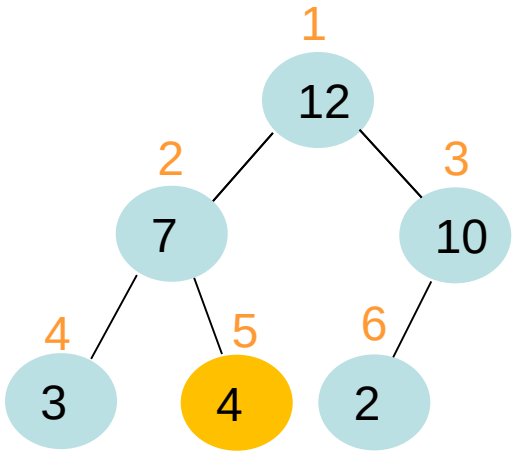
i

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)

Heapify – Operation

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

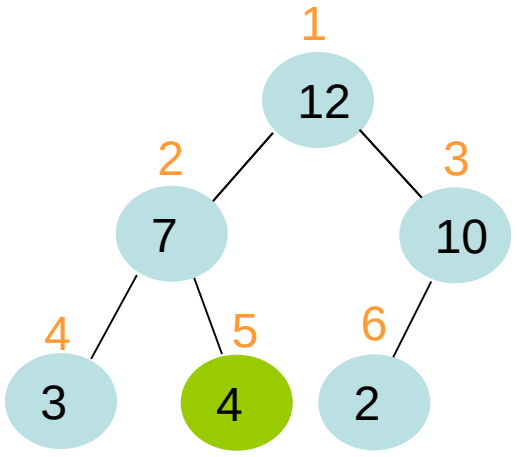
i

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)

Heapify – Operation

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

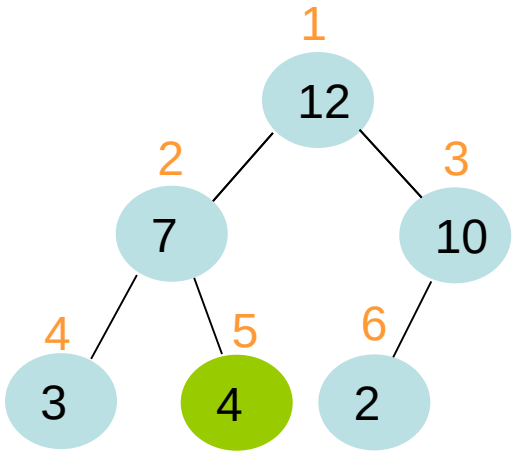
i

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)

Heapify – Operation

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

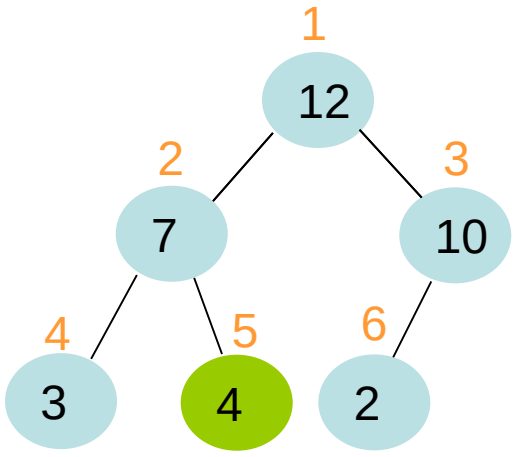
i

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)

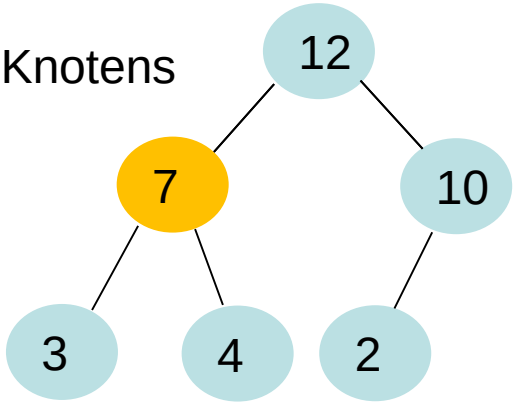
Wiederholung: Baumhöhe

Definition

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

Beispiel

Höhe des Knotens ist 1



Heapify – Laufzeit

Satz

- 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 ≤c·h ist

Heapify – Laufzeit

Satz

- 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 ≤c·h ist
- (I.A.) h=0:
  Z. 4: largest wird auf i gesetzt

Heapify – Laufzeit

Satz

- 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 ≤c·h ist
- (I.A.) h=0:
  Z. 4: largest wird auf i gesetzt
  Z. 5: Keine Änderung.

Heapify – Laufzeit

Satz

- 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 ≤c·h ist
- (I.A.) h=0:
  Z. 4: largest wird auf i gesetzt
  Z. 5: Keine Änderung.
  Z. 6/7: Kein rekursiver Aufruf ⇒ Laufzeit ist c.

Heapify – Laufzeit

Satz

- 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 $\leq c \cdot h$ 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.

Heapify – Laufzeit

Satz

- 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 ch.

Heapify – Laufzeit

Satz

- 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 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

Heapify – Laufzeit

Satz

- 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 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

Heapify – Laufzeit

Satz

- 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 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 ≤c

Heapify – Laufzeit

Satz

- 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 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 ≤c
- ⇒ Laufzeit maximal ch+c = c(h+1).

Heapify – Laufzeit

Satz

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

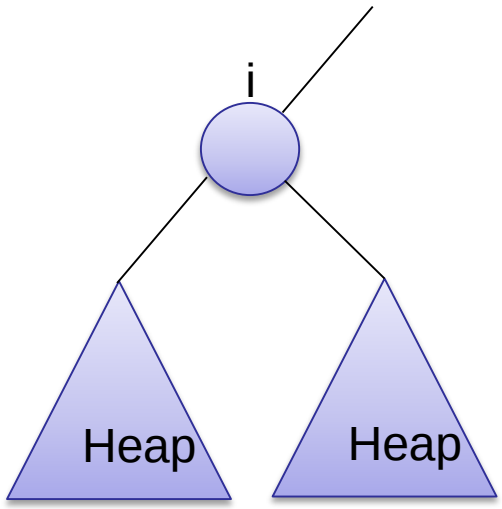
Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit 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 ≤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 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 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 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 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 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 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 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 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 Heap Eigenschaft 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 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 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 Heap Eigenschaft 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 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 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 Heap Eigenschaft 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 im Heap
  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

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 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 Heap Eigenschaft 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 im Heap
  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]: 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 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 Heap Eigenschaft 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 im Heap
  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]: 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 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 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 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 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 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 – 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 ← length[A]
2. **for** i ←⌊length[A]/2⌋ **downto** 1 **do**
3. Heapify(A,i)

Heap – Aufbau

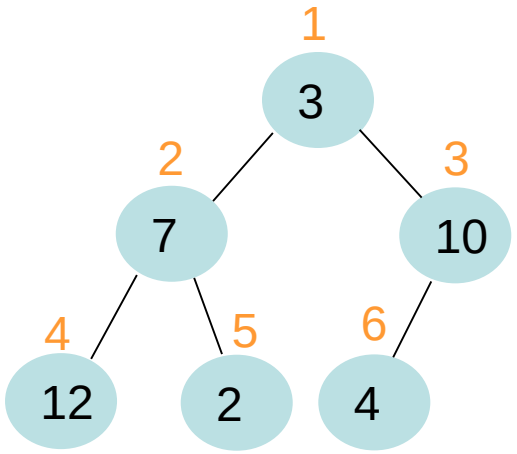
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau eines Heaps (Halde)

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

Build-Heap(A)

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



Heap – Aufbau

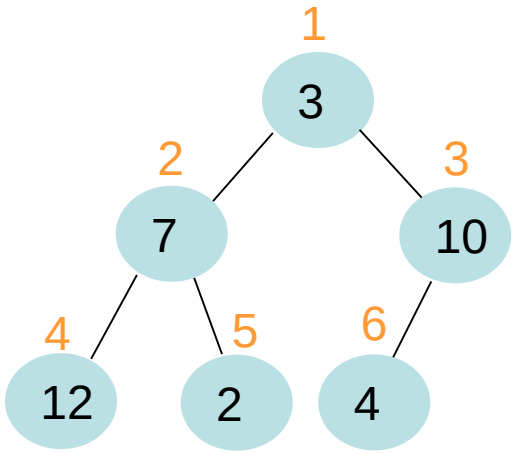
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau eines Heaps (Halde)

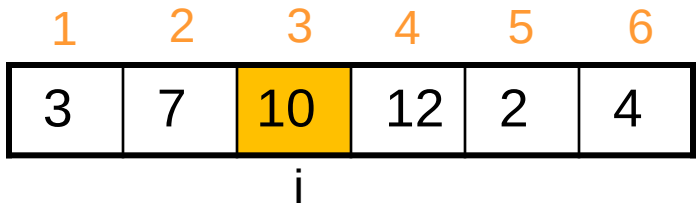
- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Heap – Aufbau

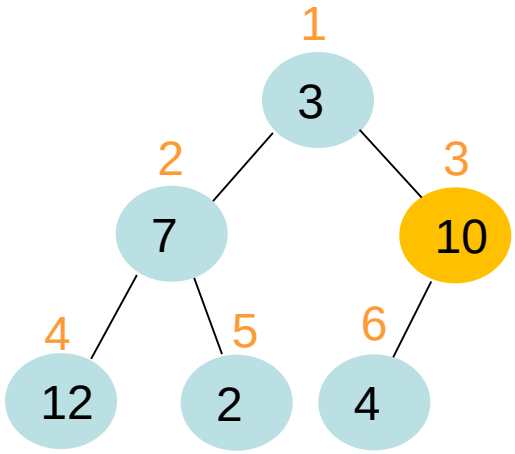


Aufbau eines Heaps (Halde)

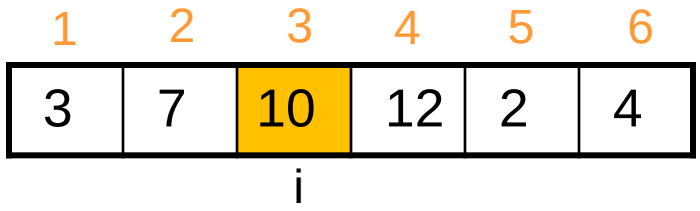
- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Heap – Aufbau

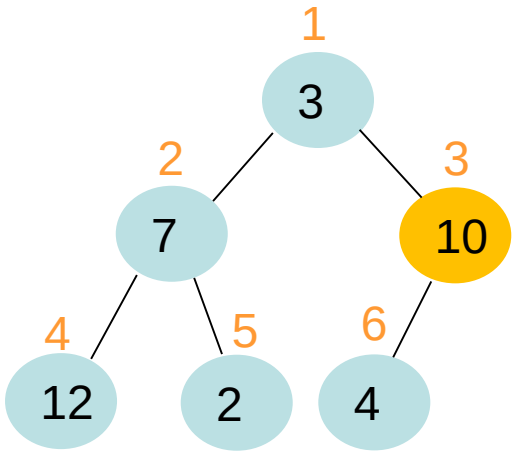


Aufbau eines Heaps (Halde)

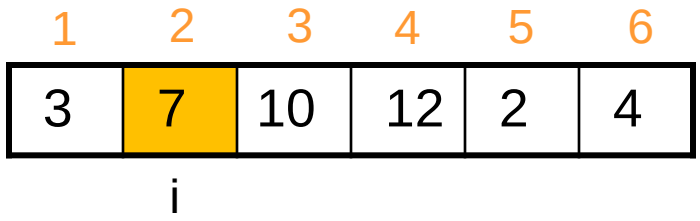
- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Heap – Aufbau

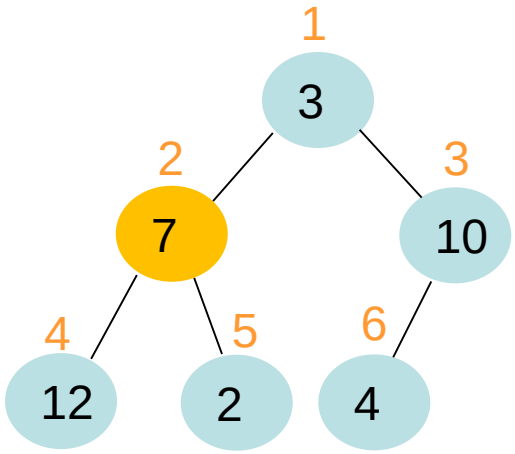


Aufbau eines Heaps (Halde)

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

Build-Heap(A)

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



Heap – Aufbau

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

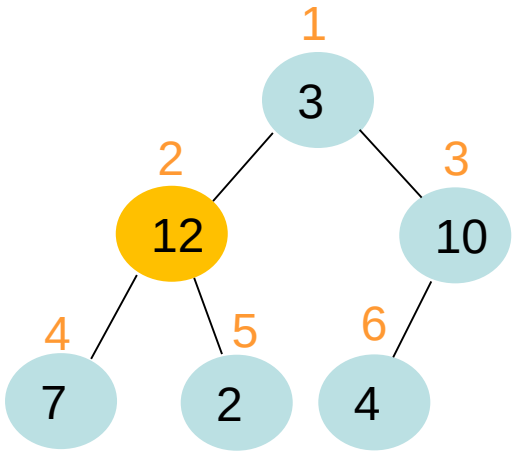
i

Aufbau eines Heaps (Halde)

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

Build-Heap(A)

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



Heap – Aufbau

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

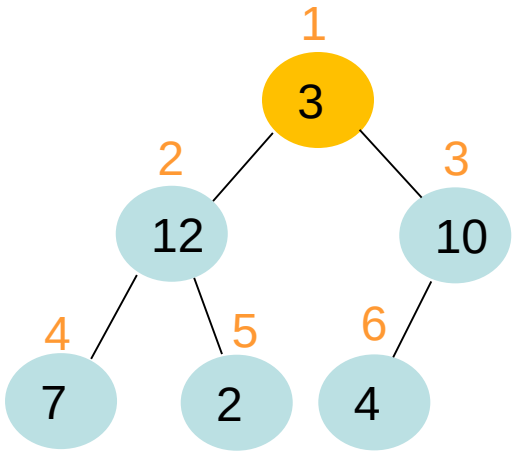
i

Aufbau eines Heaps (Halde)

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

Build-Heap(A)

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



Heap – Aufbau

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

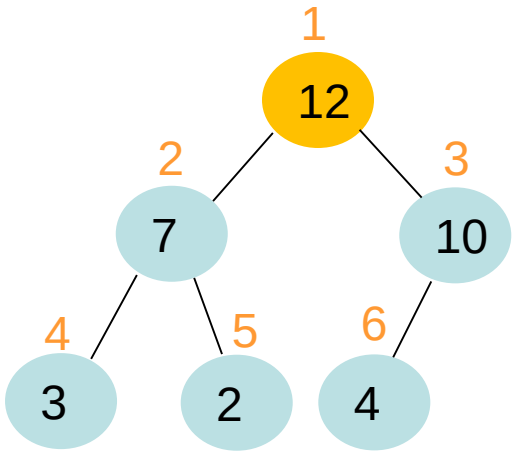
i

Aufbau eines Heaps (Halde)

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

Build-Heap(A)

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



Heap – Aufbau

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

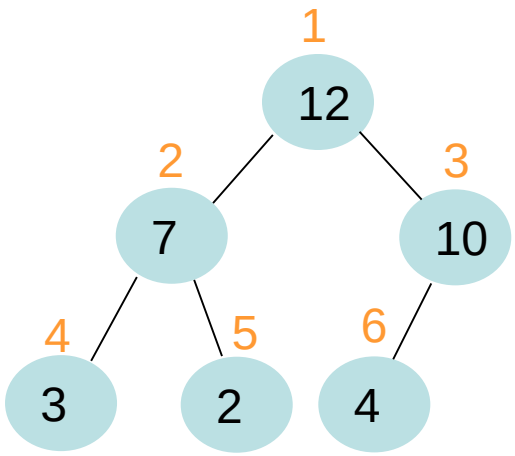
i

Aufbau eines Heaps (Halde)

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

Build-Heap(A)

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



Heap – Aufbau

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

Heap – Aufbau

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

Heap – Aufbau

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

Heap – Aufbau

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.

Heap – Aufbau

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.

Heap – Aufbau

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).

Heap – Aufbau

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.

Heap – Aufbau

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 $H=\lfloor \log n \rfloor$:

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

Heap – Aufbau

Satz

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

- *Beweis*

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

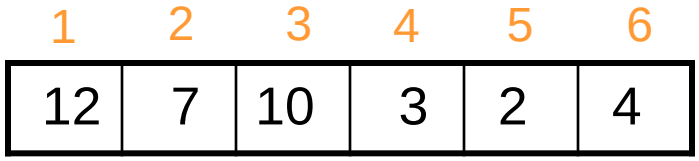
- Es gilt $\sum_{h=0}^{\infty} \frac{h+1}{2^h} = O(1)$

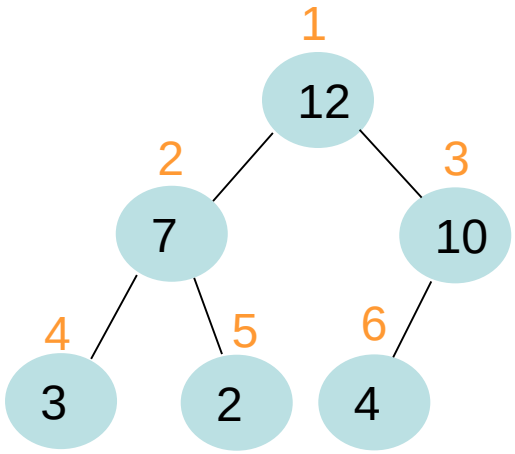
- Somit folgt eine Laufzeit von O(n).

Max Heap – Extract Max

Heap-Extract-Max(A)

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

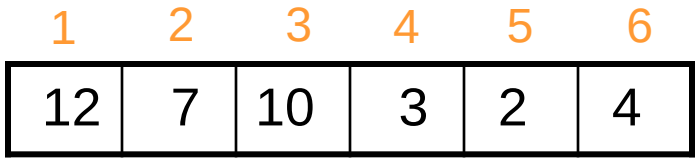


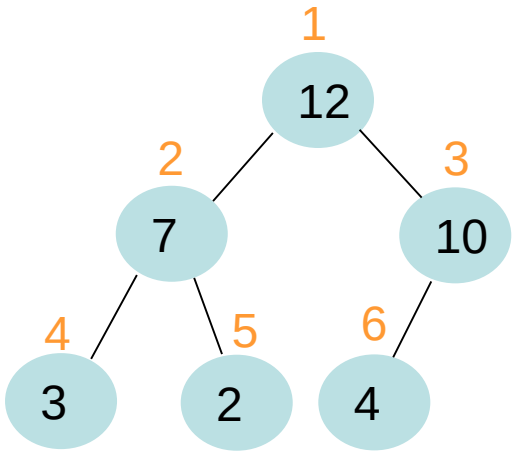


Max Heap – Extract Max

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

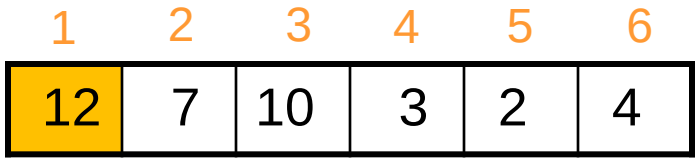




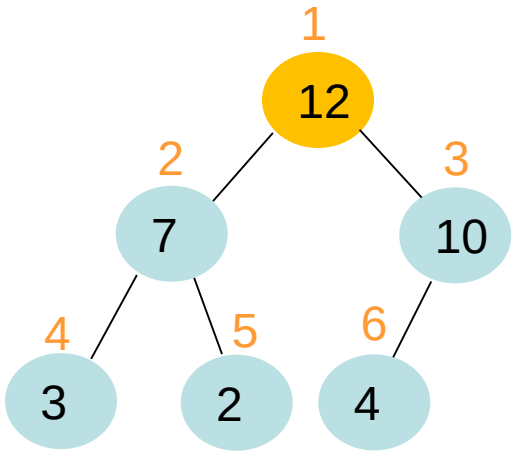
Max Heap – Extract Max

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



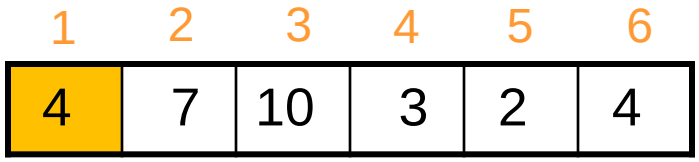
max =12



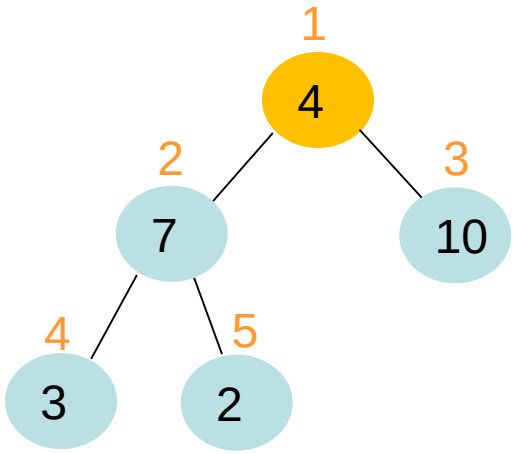
Max Heap – Extract Max

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



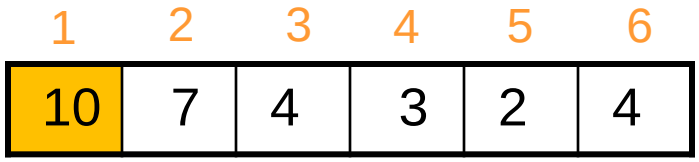
max =12



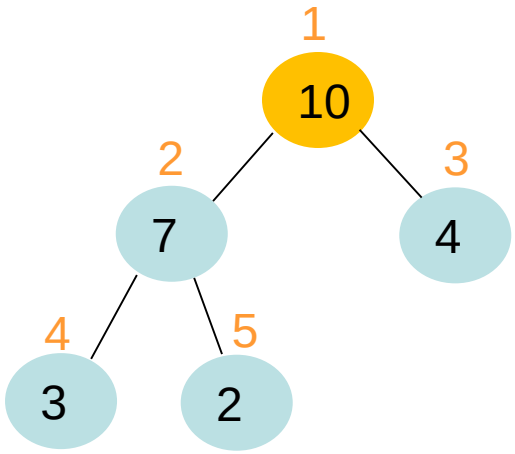
Max Heap – Extract Max

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



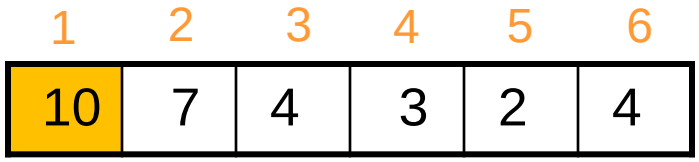
max =12



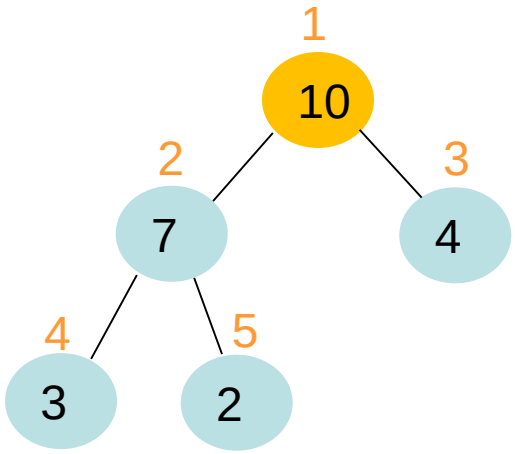
Max Heap – Extract Max

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



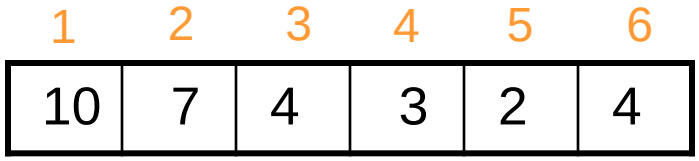
max =12



Max Heap – Extract Max

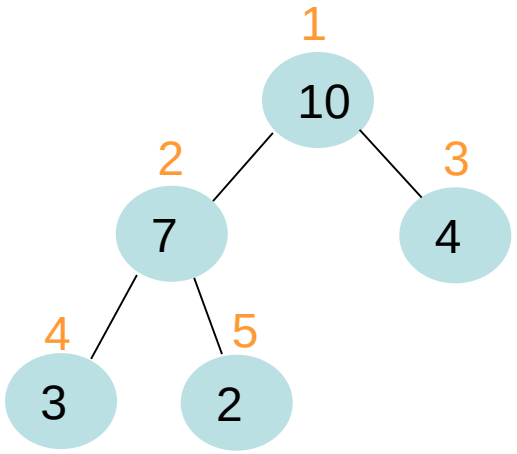
Heap-Extract-Max(A)

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



Laufzeit

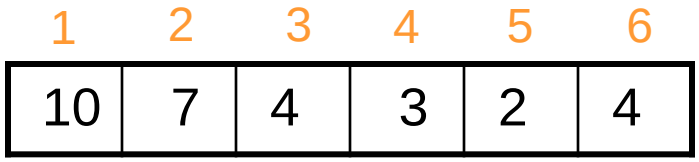
- O(log n)

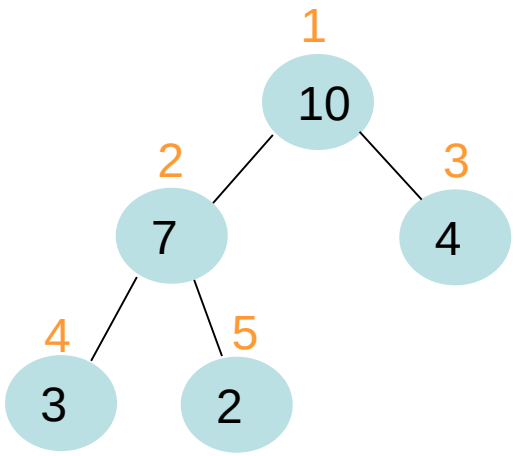


Max Heap – Einfügen

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



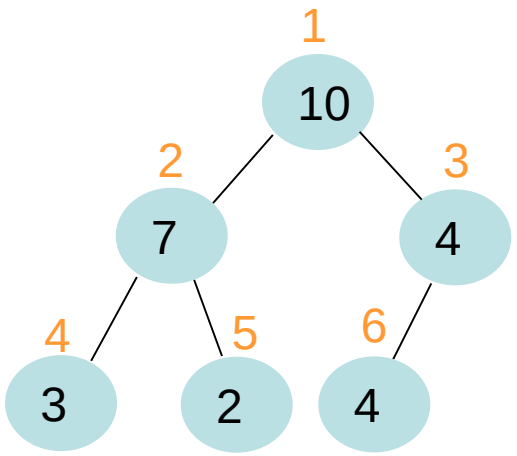


Max Heap – Einfügen

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

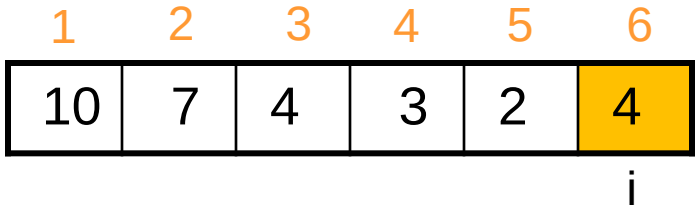


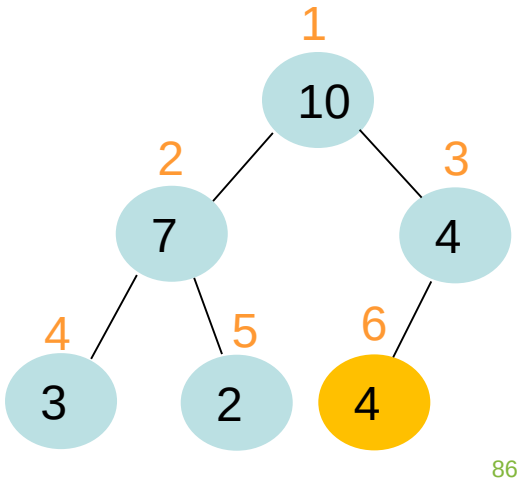
Heap-Insert(A,11)

Max Heap – Einfügen

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



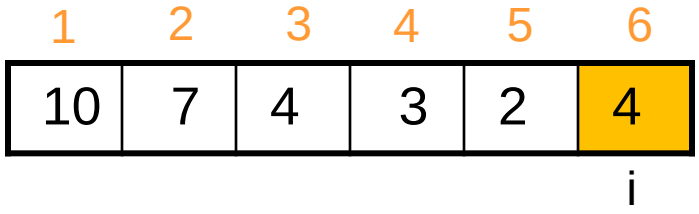


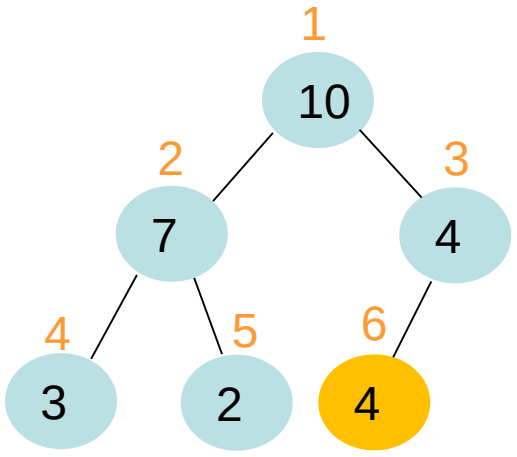
Heap-Insert(A,11)

Max Heap – Einfügen

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



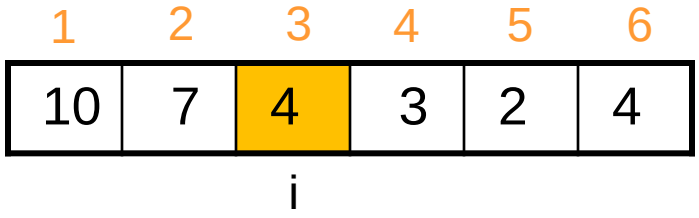


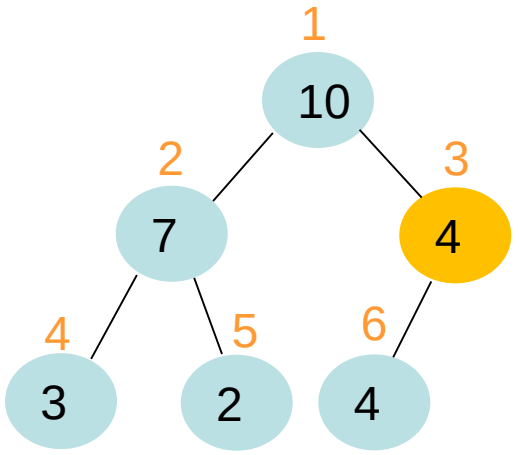
Heap-Insert(A,11)

Max Heap – Einfügen

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



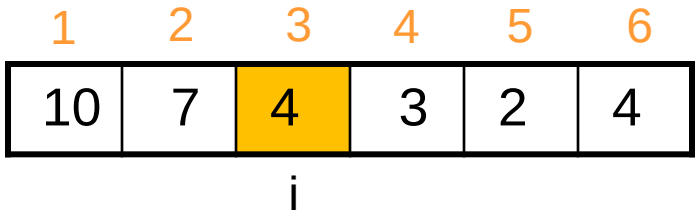


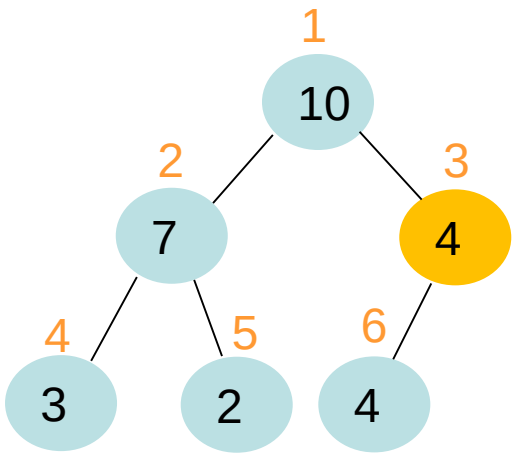
Heap-Insert(A,11)

Max Heap – Einfügen

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



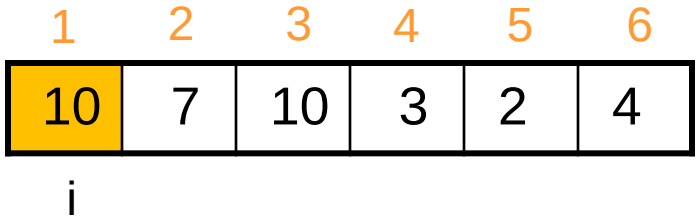


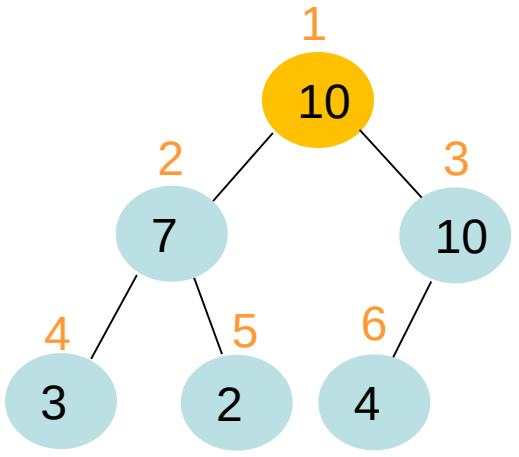
Heap-Insert(A,11)

Max Heap – Einfügen

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



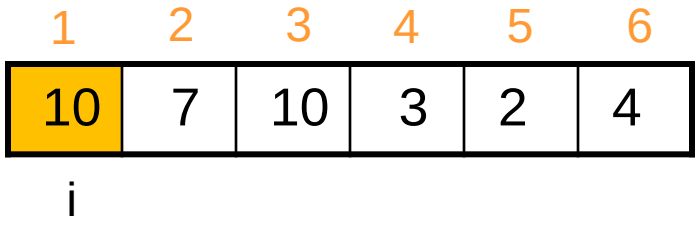


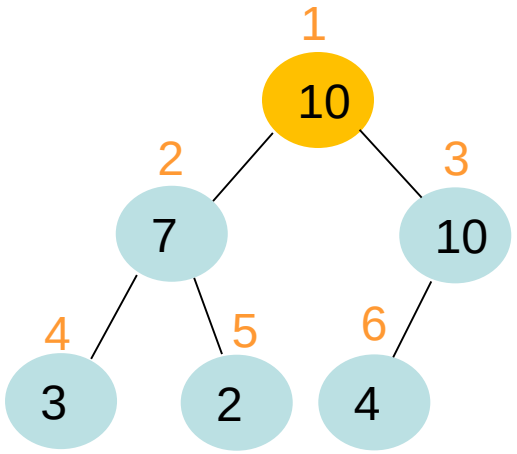
Heap-Insert(A,11)

Max Heap – Einfügen

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)

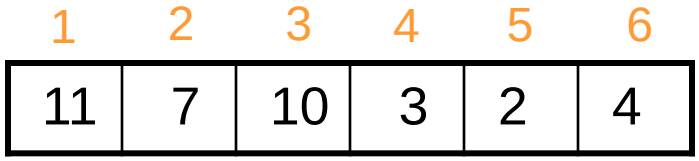
Max Heap – Einfügen

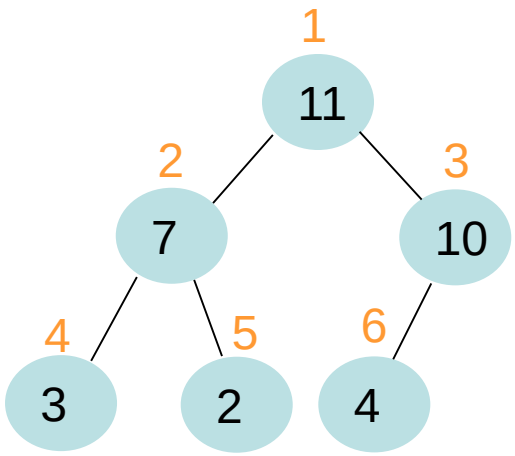
Heap-Insert(A,key)

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

Laufzeit

- O(log n)

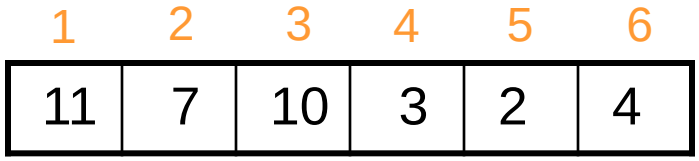


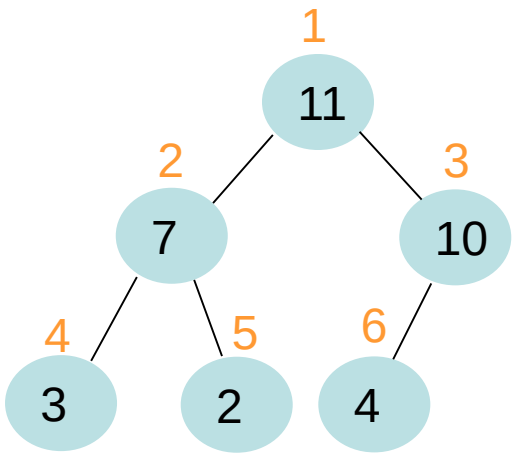


Heapsort

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)

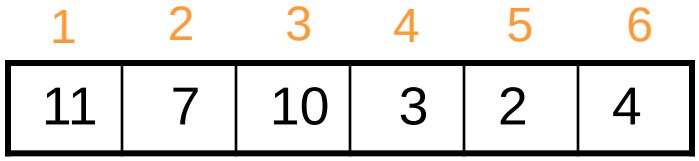


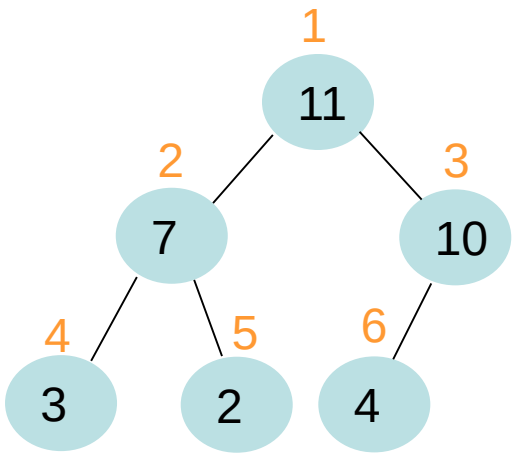


Heapsort

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)

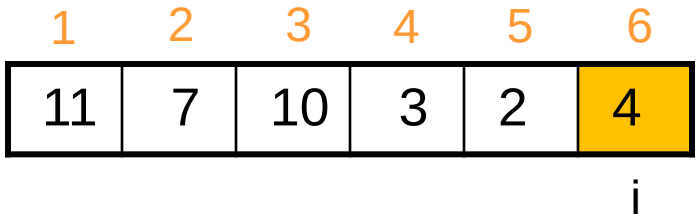


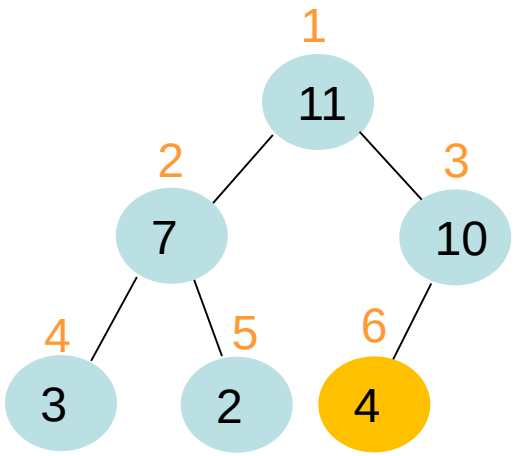


Heapsort

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)

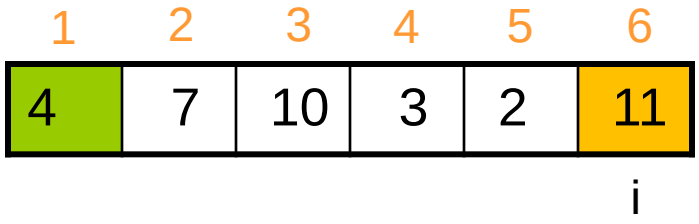


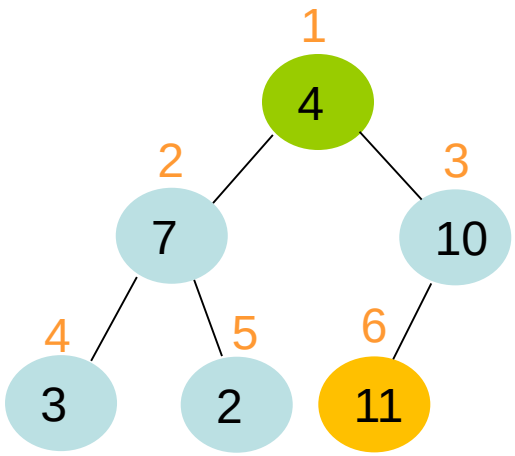


Heapsort

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)

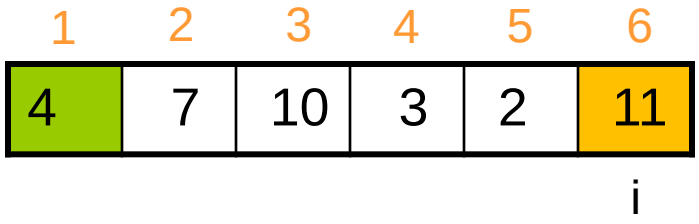


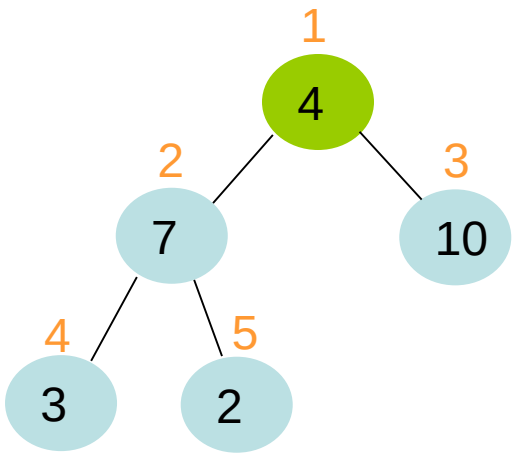


Heapsort

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)

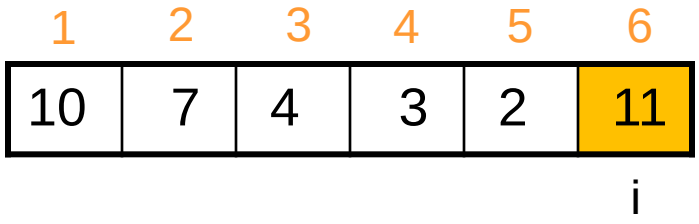


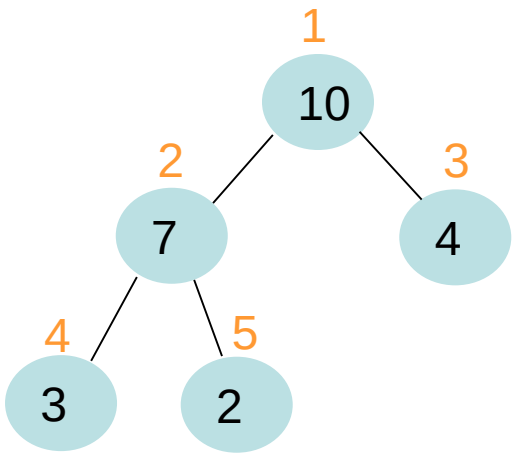


Heapsort

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)

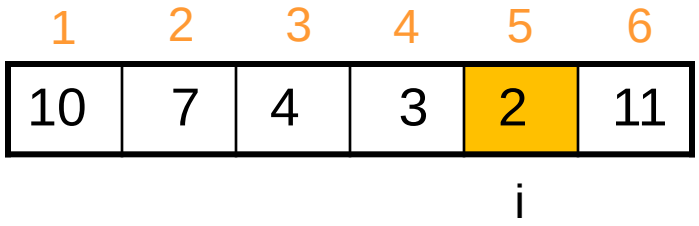


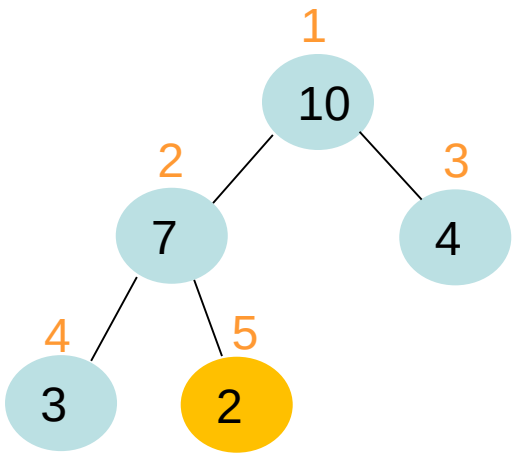


Heapsort

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)

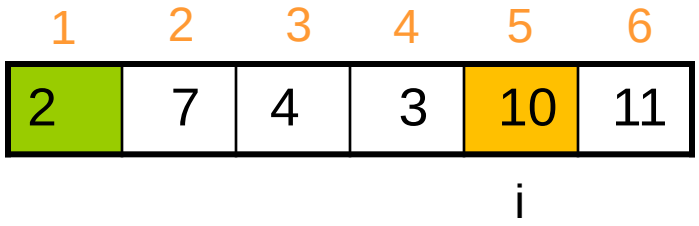


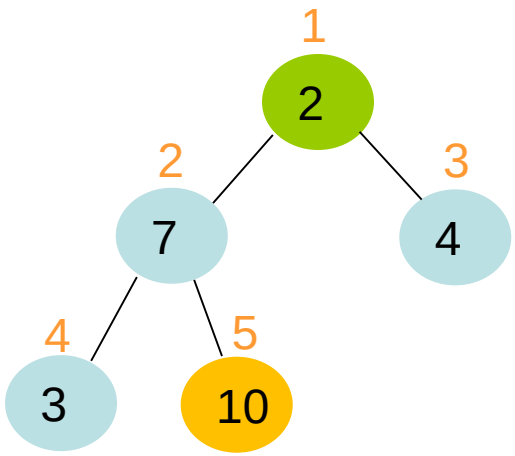


Heapsort

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)

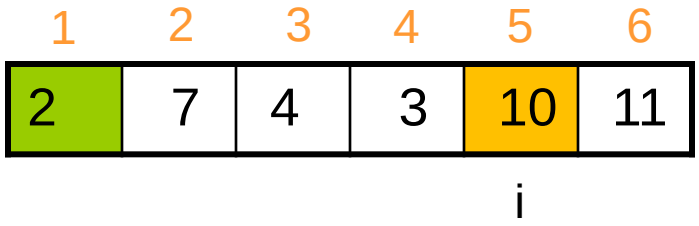


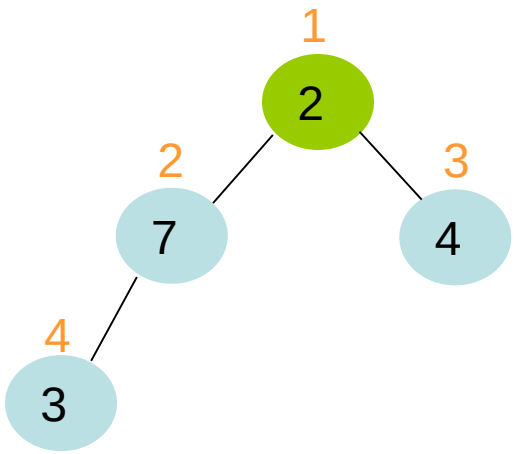


Heapsort

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)

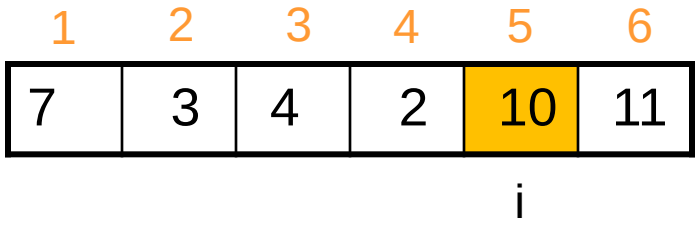


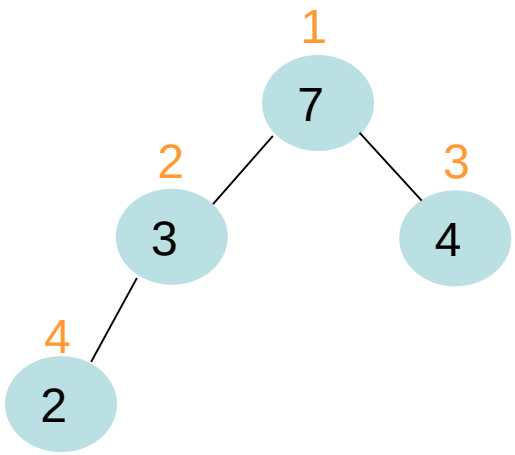


Heapsort

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)

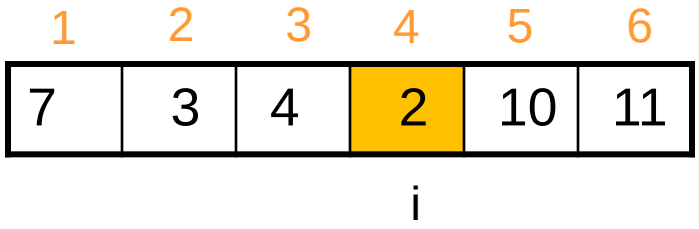


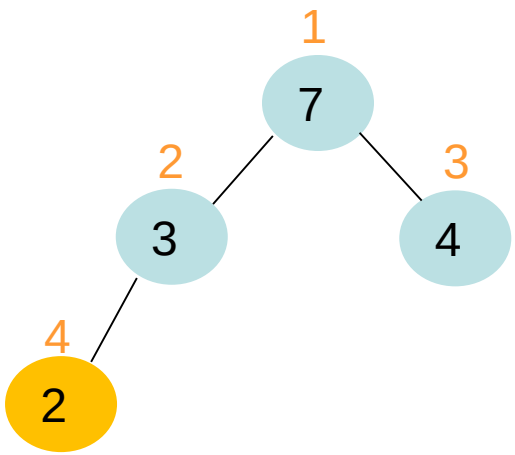


Heapsort

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)

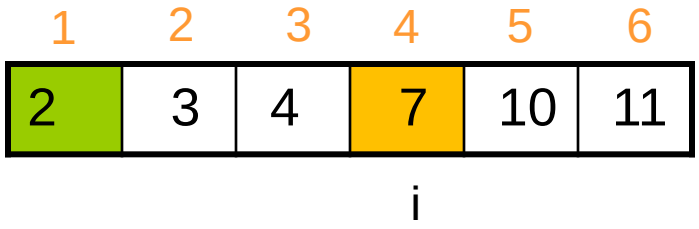


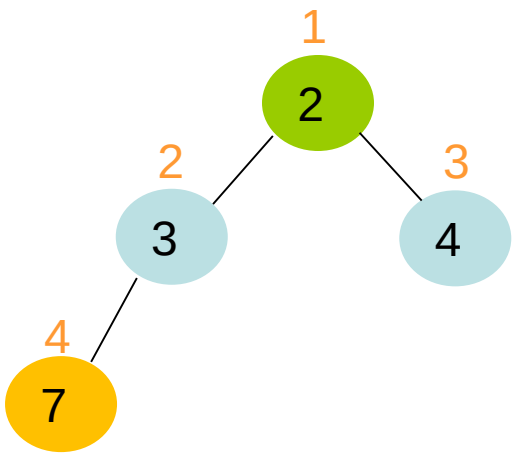


Heapsort

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)

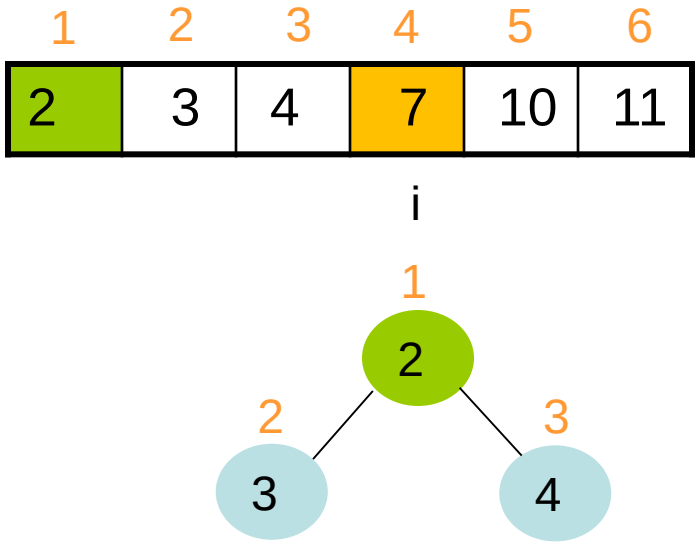




Heapsort

Heapsort(A)

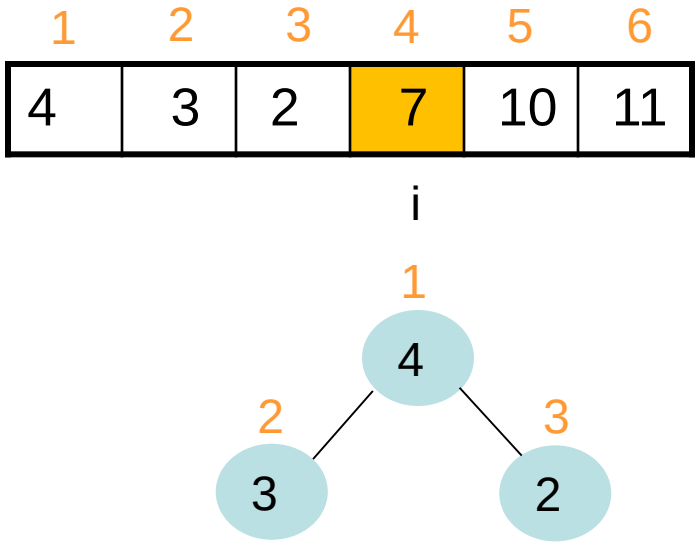
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Heapsort

Heapsort(A)

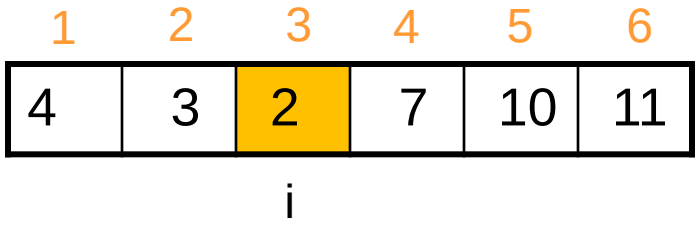
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)

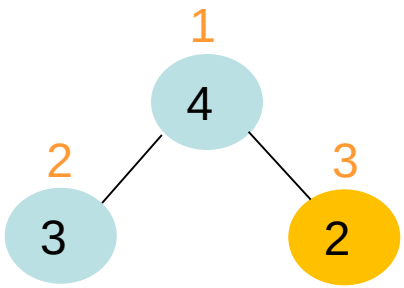


Heapsort

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)

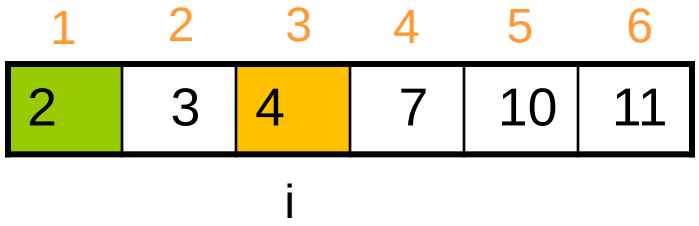


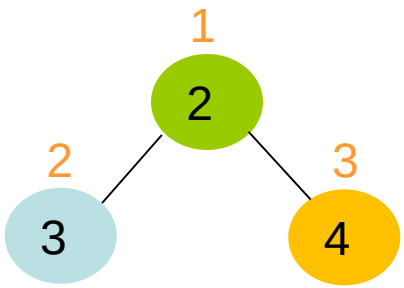


Heapsort

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)

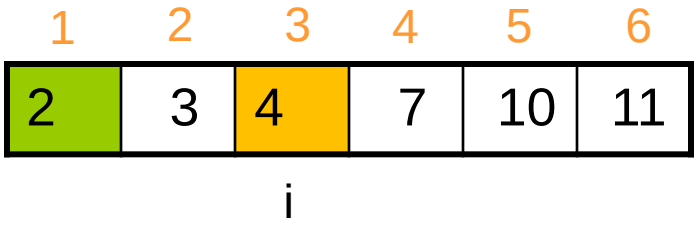


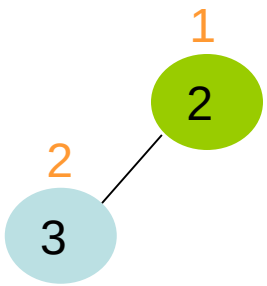


Heapsort

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)

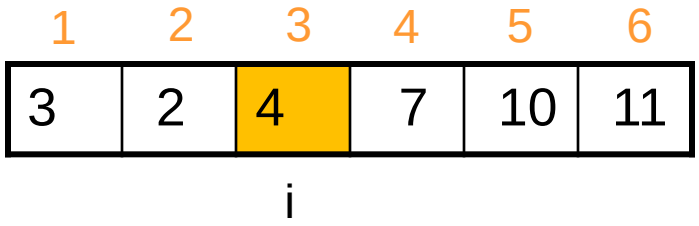


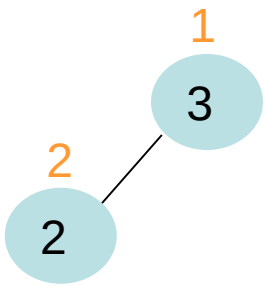


Heapsort

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)

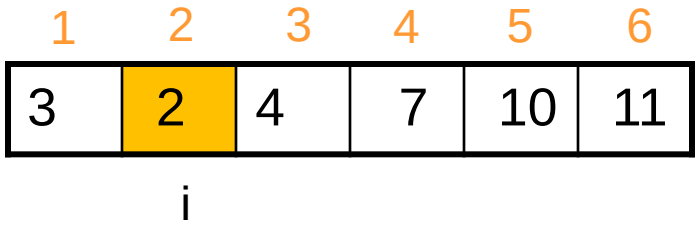


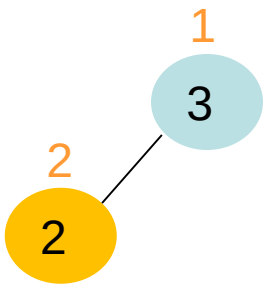


Heapsort

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)

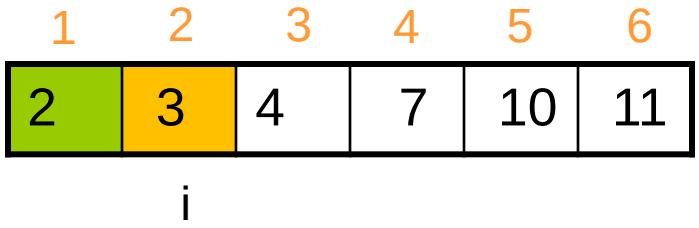


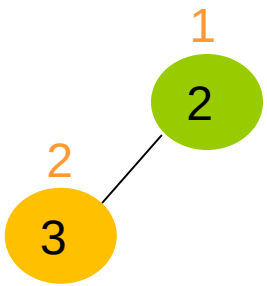


Heapsort

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)

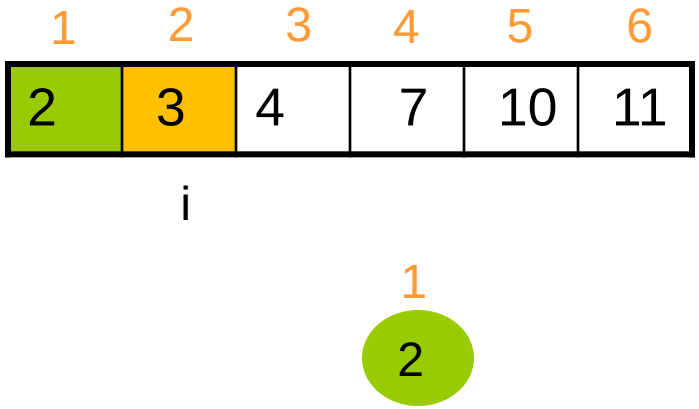




Heapsort

Heapsort(A)

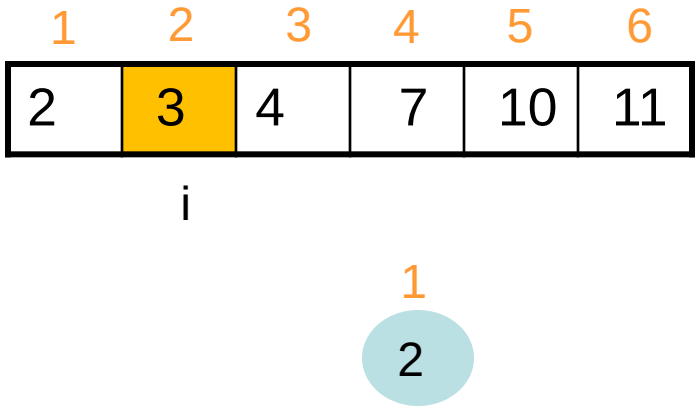
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Heapsort

Heapsort(A)

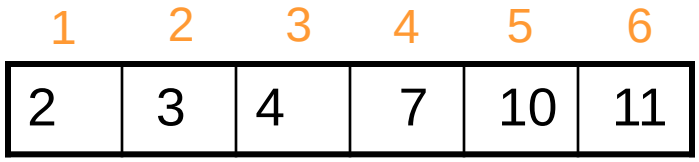
1. Build-Heap(A)
2. **for** i ← length[A] **downto** 2 **do**
3. A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A]-1
5. Heapify(A,1)



Heapsort

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)



Heapsort - Laufzeit

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)

Heaps / Halden

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