Prioritätenschlangen/Halden/Heaps

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 25/26

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Algorithmen, Pseudocode, Sortieren I“: Insertion Sort
VL 2 „Algorithmen, Pseudocode, Sortieren II“: Selection Sort, Bubble Sort, Count Sort
VL 3 „Laufzeit und Speicherplatz“: Laufzeitanalyse der vorgestellten Sortierverfahren
VL 4 „Einfache Datenstrukturen“: Arrays, verkettete Listen, Structs in C, Stack, Queue
VL 5 „Bäume“: Binärbäume, Baumtraversierung, Laufzeitanalyse Baumoperationen
VL 6 „Teile und Herrsche I“: Einführung der algorithmischen Methode, Merge Sort
VL 7 „Korrektheitsbeweise“: Rechnermodel, Beispielbeweise
VL 8 „Dateien in C“: Dateien, Dateisysteme, Verzeichnisse, Dateiverwaltung mit C
VL 9 „Prioritätenschlangen/Halden/Heaps “: Heap Sort, Binärer Heap, Heap Operationen
VL 10 „Fortgeschrittene Sortierverfahren“: Quick Sort, Radix Sort
VL 11 „AVL Bäume“: Definition, Baumoperationen, Traversierung
VL 12 „Teile und Herrsche II“: Generalisierung des algorithmischen Prinzips, Mastertheorem
VL 13 „Q & A“: Offene Vorlesung/Wiederholung

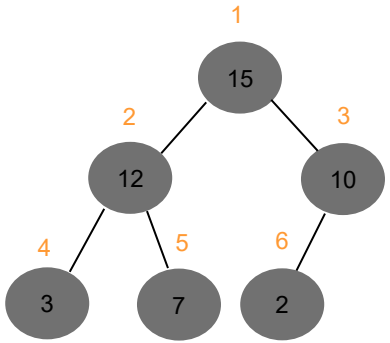
Prioritätenschlangen / Heaps

- Abstrakte Datenstruktur
  - Basiert meist auf einem Baum
  - Jedes Element hat einen Schlüssel (häufig die Elemente selbst), das die Priorität des Elements festlegt
  - Partiell geordneter Baum

- Anwendungen
  - Sortieren (HeapSort)
  - Zugriff auf Min/Max in O(1)
  - Scheduling
  - Etc.

Heap-Bedingung

- Max-Heap-Bedingung:
  - Die Schlüssel der Kinder eines Knotens sind stets kleiner als die ihres Parent $\Rightarrow$ an der Wurzel des Baumes ist immer ein Element mit maximalem Schlüssel
  - Min-Heap-Bedingung entsprechend



Heap-Implementierung

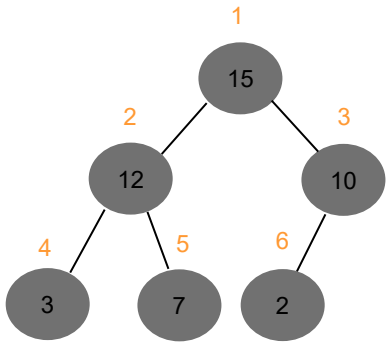
Technische Universität Berlin

Operationen: Einfügen, Löschen, Maximum (bzw. Minimum)

- Wir könnten binäre Suchbäume benutzen
- Gibt es einfachere Datenstrukturen?
- Gibt es effizientere Datenstrukturen?
  (Ja, AVL-Bäume oder andere, die nicht Teil dieser Vorlesung sind)
- Häufig als Array implementiert

Binäre Halden: Bäume als Arrays

- Array A[1,…,length(A)]
- Man kann ein Array 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(A) ≤ length(A)

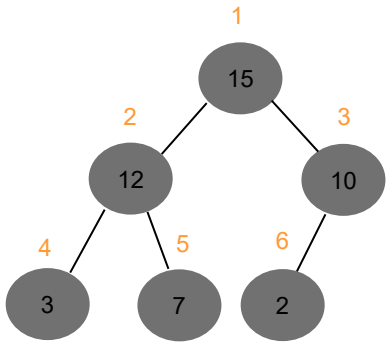


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

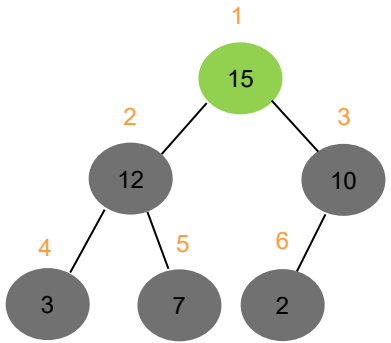


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

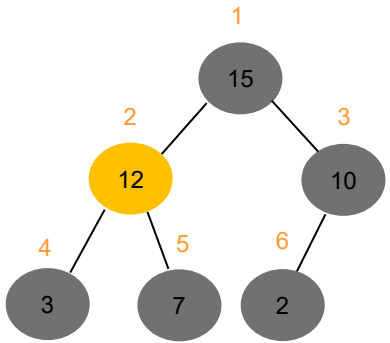


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

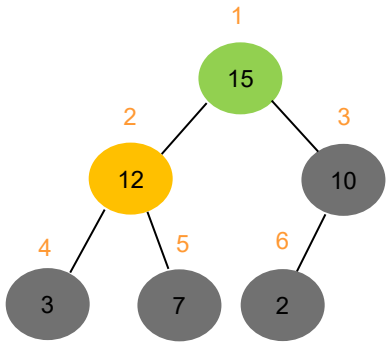


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

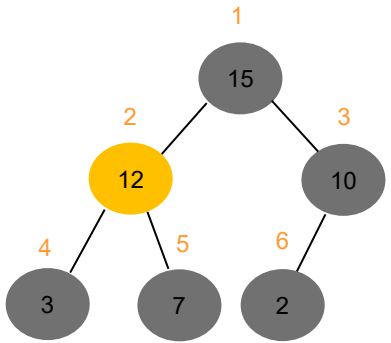


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

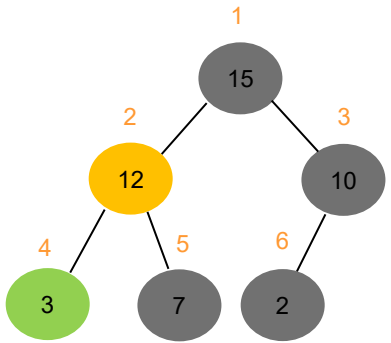


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

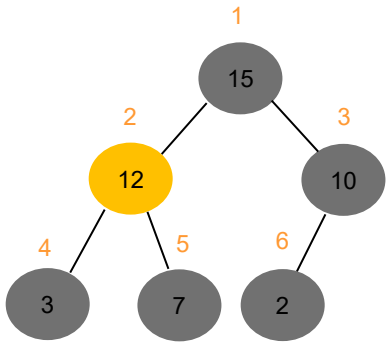


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

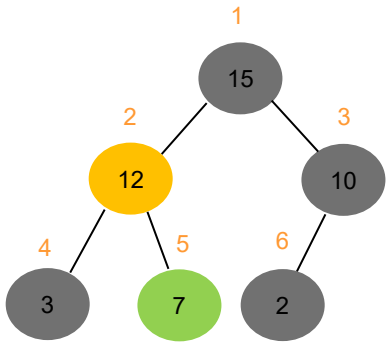


1	2	3	4	5	6
15	12	10	3	7	2

Binäre Halden: Bäume als Arrays

Navigation

- Wurzel: 1
- Parent(i): $\lfloor i/2 \rfloor$
- Left(i): 2i
- Right(i): 2i+1

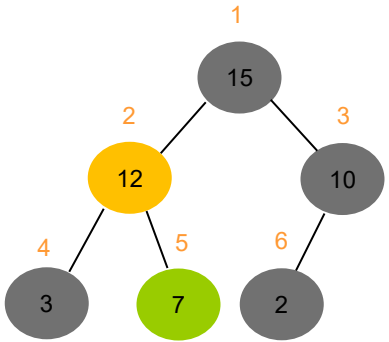


1	2	3	4	5	6
15	12	10	3	7	2

Max Heap Eigenschaft (Max Heap property)

Technische Universität Berlin

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



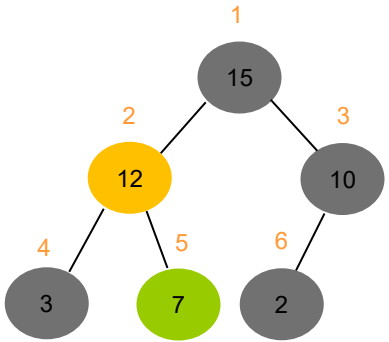
1	2	3	4	5	6
15	12	10	3	7	2

Open Distributed Sytems ODS

Max Heap Eigenschaft (Max Heap property)

Technische Universität Berlin

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

Min Heap Eigenschaft / min Heap property: analog mit ≤

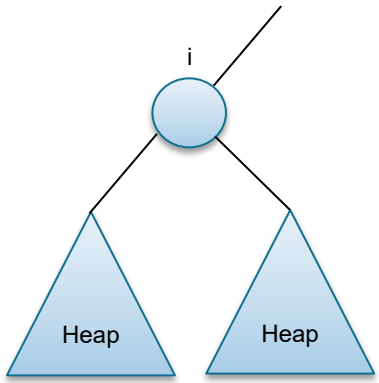
Open Distributed Sytems
ODS

Heapify – Operation

Voraussetzung:

- Die Teilarrays mit Wurzel Left(i) und Right(i) sind Heaps
- A[i] ist aber evtl. kleiner als seine Kinder

➢ Heapify(A,i) lässt i „absinken“, so dass die Heap Eigenschaft erfüllt wird

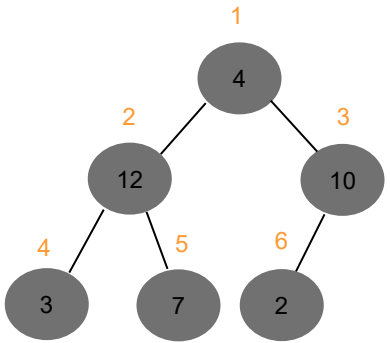


Heapify – Operation

Heapify(A,i)

1. l ← 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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)

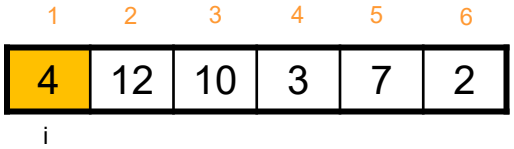
1	2	3	4	5	6
4	12	10	3	7	2

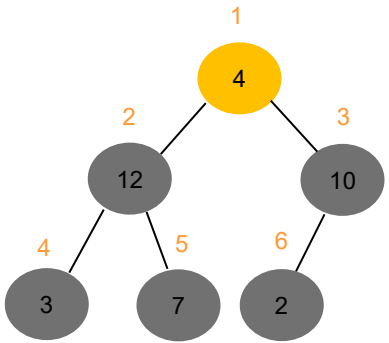


Heapify – Operation

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



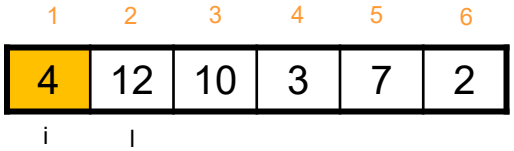


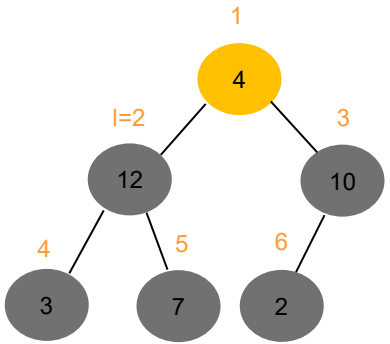
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



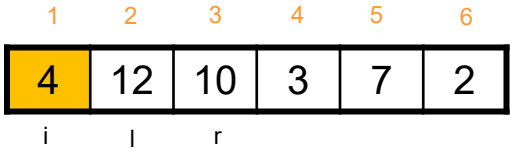


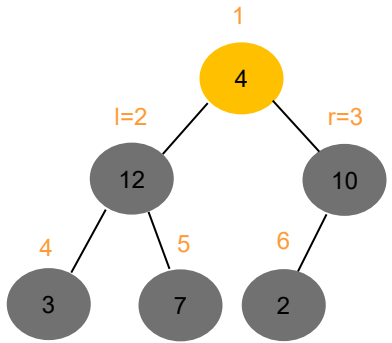
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



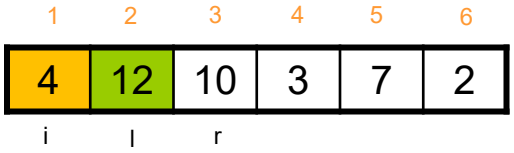


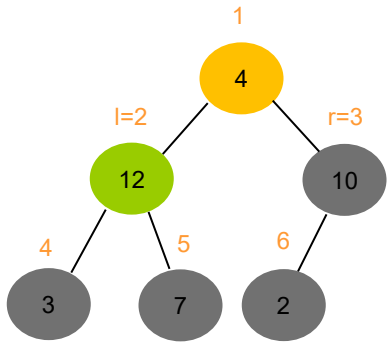
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



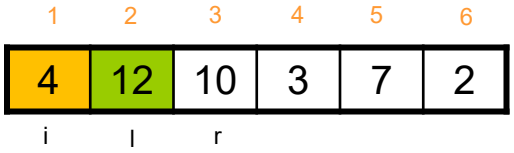


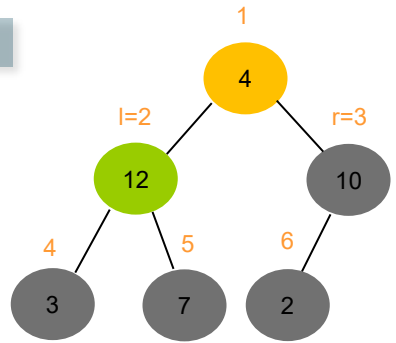
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



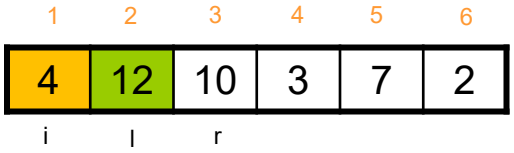


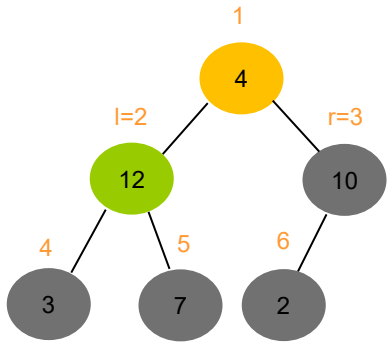
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



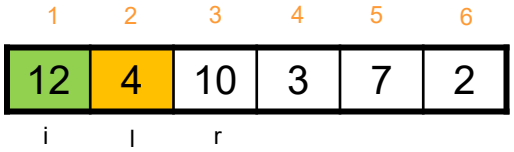


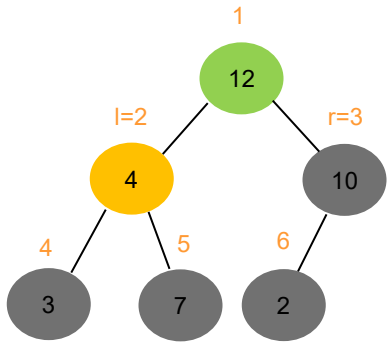
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



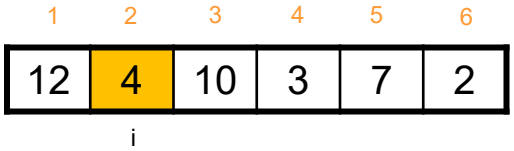


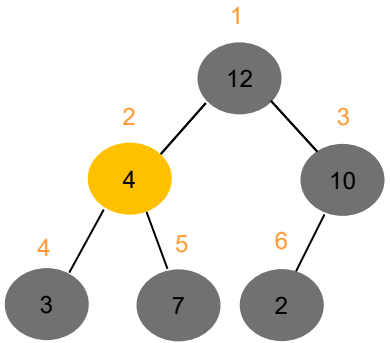
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



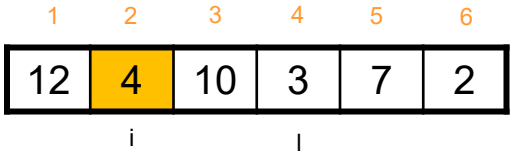


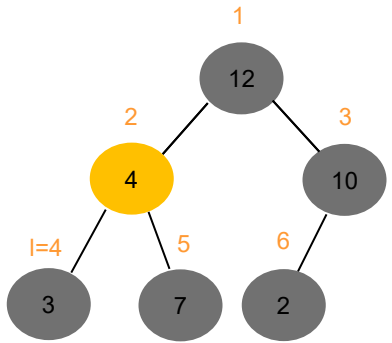
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



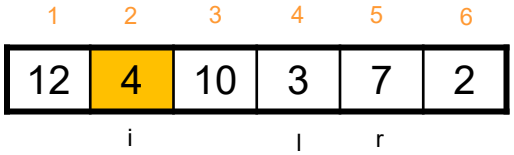


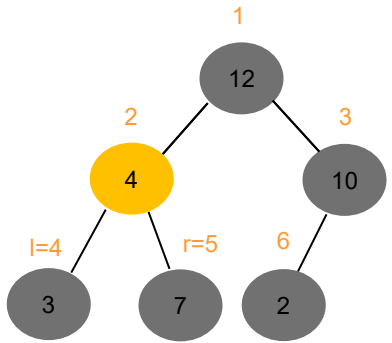
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



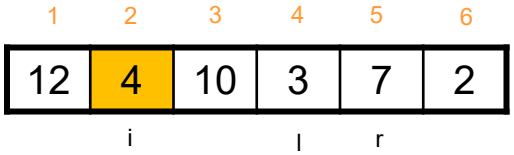


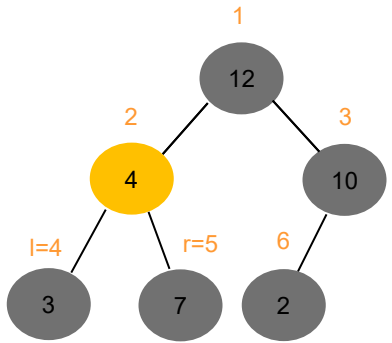
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



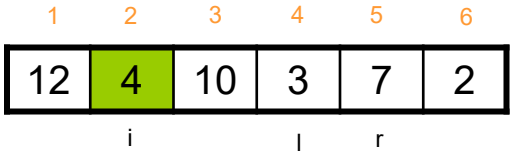


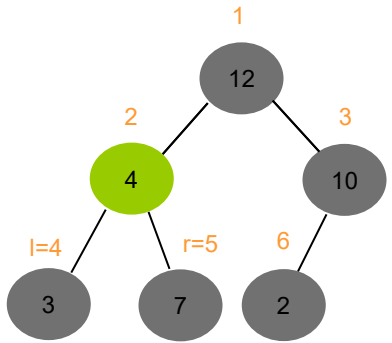
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



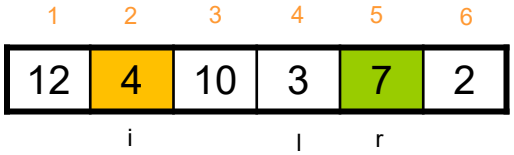


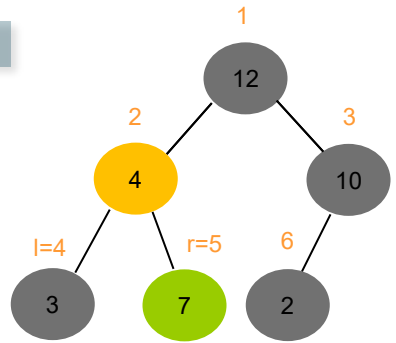
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



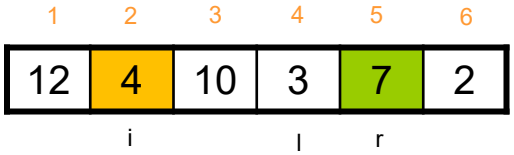


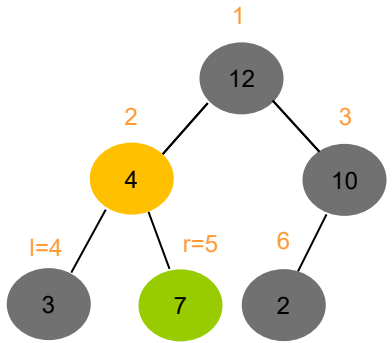
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



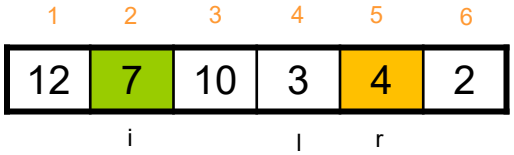


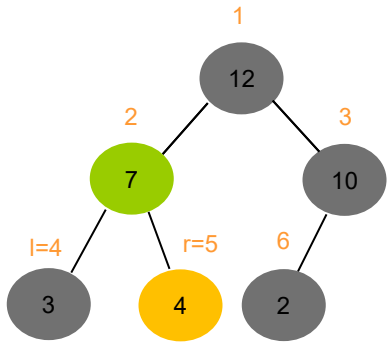
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



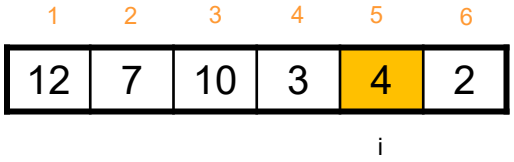


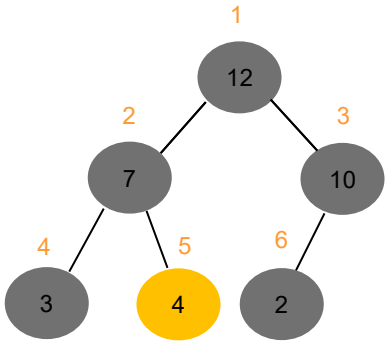
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



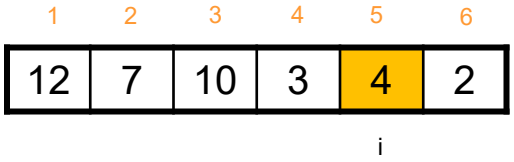


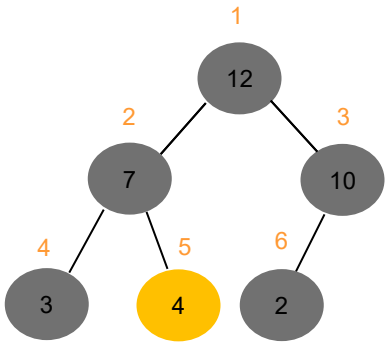
Heapify(A,1)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



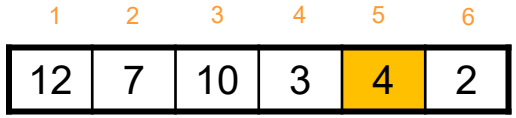


Heapify(A,1)

Heapify – Operation

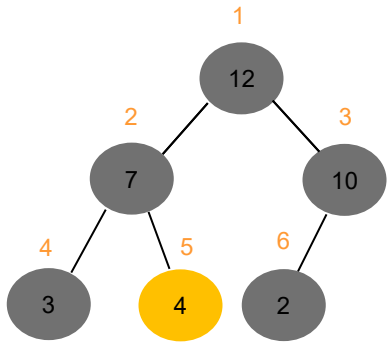
Heapify(A,i)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



l=10

r=11

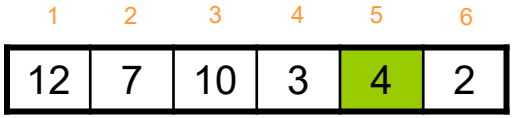


Heapify(A,1)

Heapify – Operation

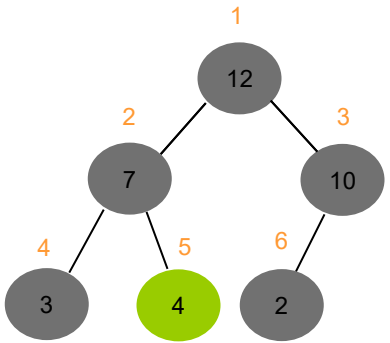
Heapify(A,i)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



l=10

r=11

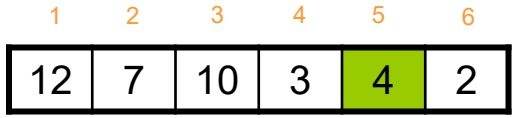


Heapify(A,1)

Heapify – Operation

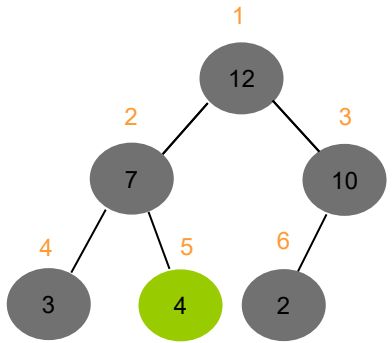
Heapify(A,i)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



l=10

r=11

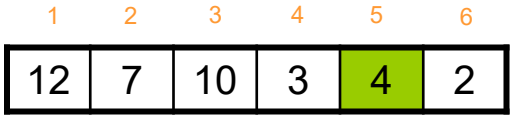


Heapify(A,1)

Heapify – Operation

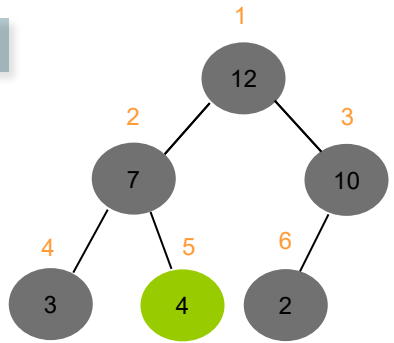
Heapify(A,i)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



l=10

r=11

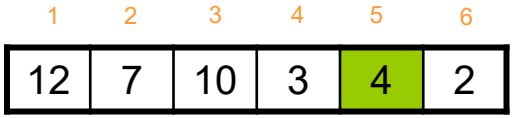


Heapify(A,1)

Heapify – Operation

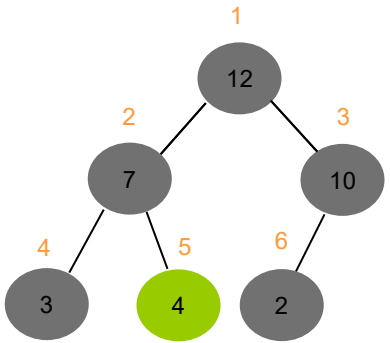
Heapify(A,i)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



l=10

r=11

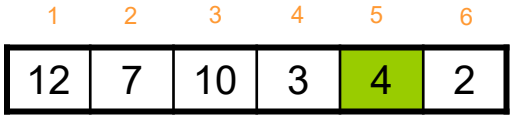


Heapify(A,1)

Heapify – Operation

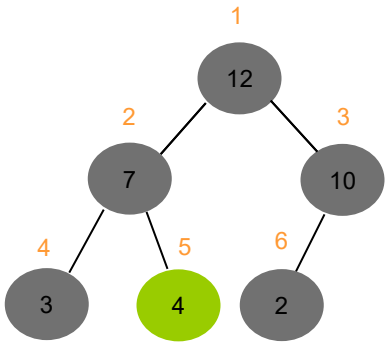
Heapify(A,i)

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**
7. A[i] ↔ A[largest]
8. Heapify(A,largest)



l=10

r=11



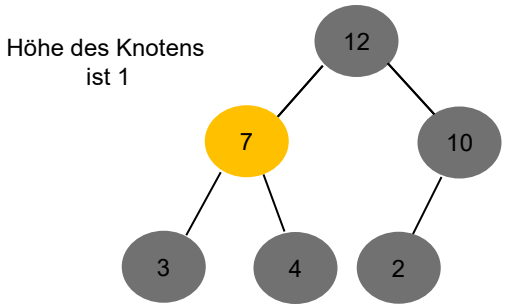
Heapify(A,1)

Wiederholung: Baumhöhe

Technische Universität Berlin

Definition:

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



Open Distributed Sytems ODS

Heapify – Laufzeit

Technische Universität Berlin

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 ≤ $c \cdot h$ 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 ≤ $c \cdot h$ ist

(I.A.) $h=0$:

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

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 ≤ c·h ist

(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: Zeige per Induktion über h, dass die Laufzeit ≤ c·h ist

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

(I.S.) Betrachte Knoten der Höhe 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: Zeige per Induktion über h, dass die Laufzeit ≤ c·h ist

(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.

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.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: Zeige per Induktion über h, dass die Laufzeit ≤ c·h ist

(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: Zeige per Induktion über h, dass die Laufzeit ≤ c·h ist

(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: Zeige per Induktion über h, dass die Laufzeit ≤ c·h ist

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

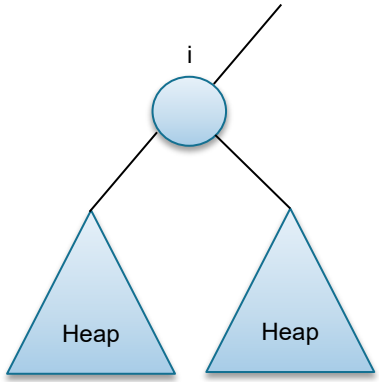
(I.S.) Betrachte Knoten der Höhe h+1.

- 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).

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und 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 und 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.

Heap Eigenschaft und Heapify

Technische Universität Berlin

Satz: Wenn die Unterbäume des rechten und 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

Heap Eigenschaft und Heapify

Technische Universität Berlin

Satz: Wenn die Unterbäume des rechten und 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

Technische Universität Berlin

Satz: Wenn die Unterbäume des rechten und 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.

(I.S.) Betrachte Aufruf *Heapify(A,i)* für Knoten i der Höhe h+1>1, wenn Unterbäume der Kinder von i bereits Heap Eigenschaft erfüllen

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und 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.S.) Betrachte Aufruf *Heapify(A,i)* für Knoten i der Höhe h+1>1, wenn Unterbäume der Kinder von i bereits Heap Eigenschaft erfüllen

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und 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.S.) Betrachte Aufruf *Heapify(A,i)* für Knoten i der Höhe h+1>1, wenn Unterbäume der Kinder von i bereits Heap Eigenschaft erfüllen

- l und r: linkes bzw. rechtes Kind von i

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und 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.S.) Betrachte Aufruf *Heapify(A,i)* für Knoten i der Höhe h+1>1, wenn Unterbäume der Kinder von i bereits Heap Eigenschaft erfüllen

- l und r: linkes bzw. rechtes Kind von i
- Höhe von i>1

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und 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.S.) Betrachte Aufruf *Heapify(A,i)* für Knoten i der Höhe h+1>1, wenn Unterbäume der Kinder von i bereits Heap Eigenschaft erfüllen

- l und r: linkes bzw. rechtes 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 und 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.S.) Betrachte Aufruf *Heapify(A,i)* für Knoten i der Höhe h+1>1, wenn Unterbäume der Kinder von i bereits Heap Eigenschaft erfüllen

- l und r: linkes bzw. rechtes 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

- 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 und 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.S.) Betrachte Aufruf *Heapify(A,i)* für Knoten i der Höhe h+1>1, wenn Unterbäume der Kinder von i bereits Heap Eigenschaft erfüllen

- l und r: linkes bzw. rechtes 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

- Z. 3-5: *Heapify(A,i)* speichert Index von max{A[i], A[l], A[r]} in largest
- Fallunterscheidung: A[i] ist Maximum / A[l] ist Maximum / A[r] ist Maximum

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum analog):

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum 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 und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum 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 und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum 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

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum 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

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum 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

  $\Rightarrow$ Nach (I.V.): Nach Aufruf hat dieser Unterbaum Heap Eigenschaft

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum 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

  $\Rightarrow$ Nach (I.V.): Nach Aufruf hat dieser Unterbaum Heap Eigenschaft
  - Bei i ist Max. aller Elemente gespeichert und rechter Unterbaum erfüllt Heap Eigenschaft

Heap Eigenschaft und Heapify

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

Beweis: Fortsetzung Induktion über die Höhe von i. / (I.S.) – Fallunterscheidung:

- A[i] ist Maximum: Heap Eigenschaft ist bereits erfüllt; kein rekursiver Aufruf
- A[l] ist Maximum (A[r] ist Maximum 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

  $\Rightarrow$ 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

Idee:

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

Build-Heap(A)

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

Heap – Aufbau

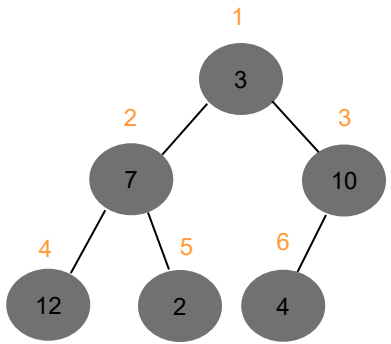
Idee:

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

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

Build-Heap(A)

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



Heap – Aufbau

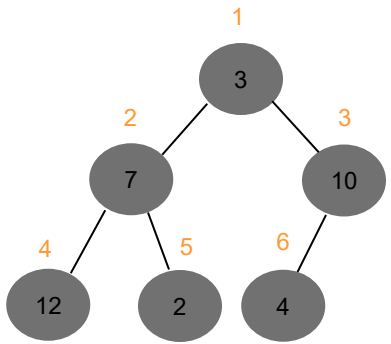
Idee:

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

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

Build-Heap(A)

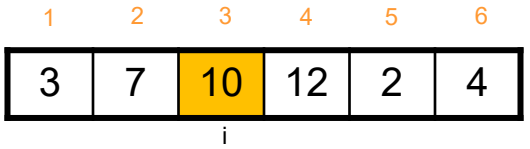
1. heap-size(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1 do
3. Heapify(A,i)



Heap – Aufbau

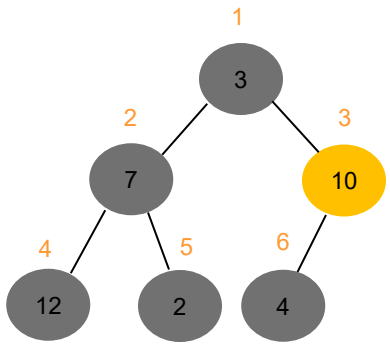
Idee:

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



Build-Heap(A)

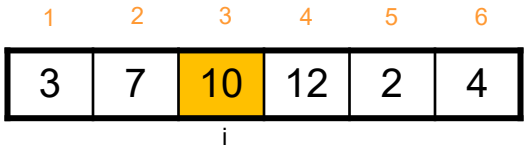
1. heap-size(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1 do
3. Heapify(A,i)



Heap – Aufbau

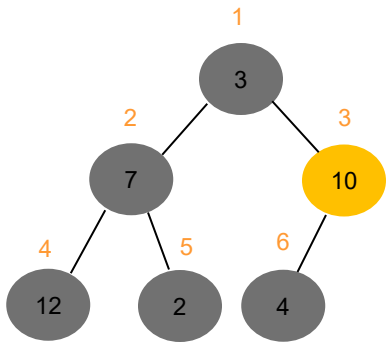
Idee:

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



Build-Heap(A)

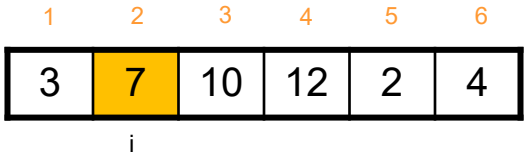
1. heap-size(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1 do
3. Heapify(A,i)



Heap – Aufbau

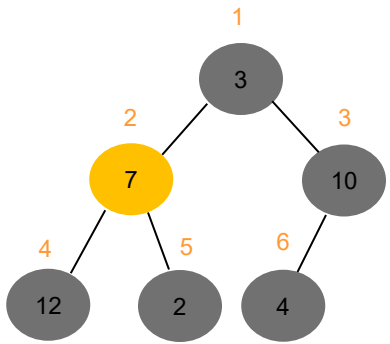
Idee:

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



Build-Heap(A)

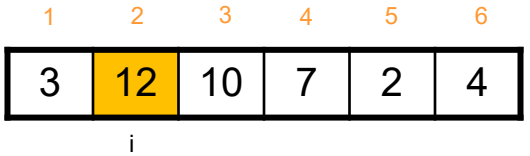
1. heap-size(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1 do
3. Heapify(A,i)



Heap – Aufbau

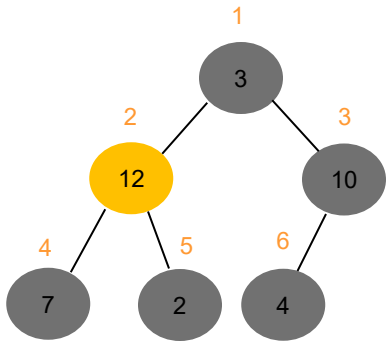
Idee:

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



Build-Heap(A)

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



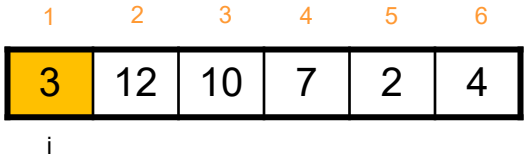
Heap – Aufbau

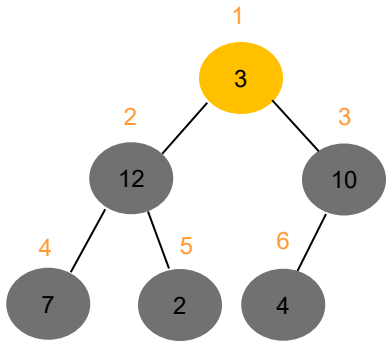
Idee:

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

Build-Heap(A)

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

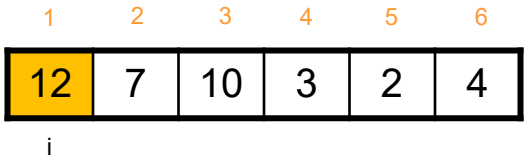




Heap – Aufbau

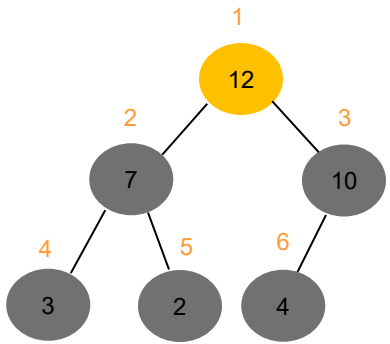
Idee:

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



Build-Heap(A)

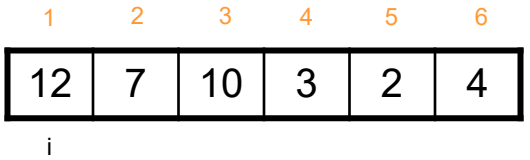
1. heap-size(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1 do
3. Heapify(A,i)



Heap – Aufbau

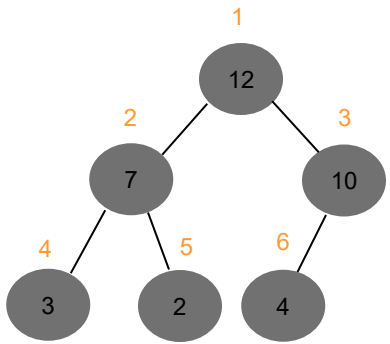
Idee:

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



Build-Heap(A)

1. heap-size(A) ← 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.

Heap – Aufbau

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

Beweis (Korrektheit):

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 die 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 die 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 die 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 die Heap Eigenschaft

- Gilt insbesondere für die Unterbäume der Kinder von i
- Aus vorherigen Satz folgt, dass Invariante durch Heapify aufrechterhalten wird

Satz: Wenn die Unterbäume des rechten und linken Kindes von *i* die Heap-Eigenschaft besitzen, dann ist diese nach der Operation *Heapify(A,i)* für den Unterbaum von *i* erfüllt.

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 die 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

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 die 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.

Heap – Aufbau

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

Beweis (Einfache Laufzeitanalyse):

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.

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):

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse):

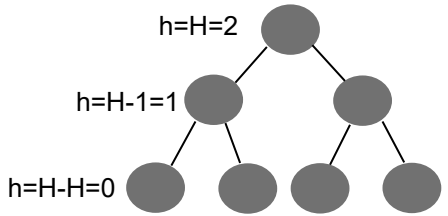
- In einem Heap mit Höhe H gibt es maximal
  2^0 Knoten mit Höhe H,
  2^1 Knoten mit Höhe H-1,
  2^2 Knoten mit Höhe H-2,
  …
  2^H Knoten mit Höhe H-H = 0 (Blätter)

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse):

- 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,
  …
  2^H Knoten mit Höhe H-H = 0 (Blätter)



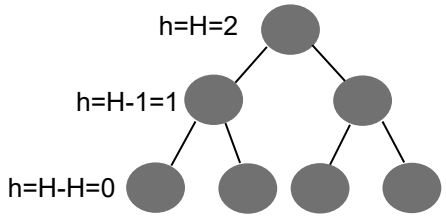
Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse):

- 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,
  …
  2^H Knoten mit Höhe H-H = 0 (Blätter)

  } H+1 Ebenen

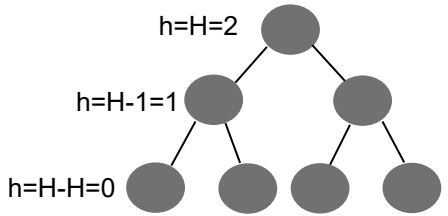


Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse):

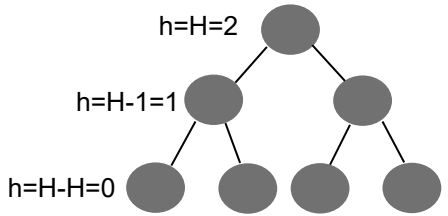
- In einem Heap mit Höhe H gibt es maximal
  2^0 Knoten mit Höhe H,
  …
  2^H Knoten mit Höhe H-H = 0 (Blätter)
  } H+1 Ebenen
- D.h. wir bauen 2^H Heaps der Höhe 0, 2^{H-1} Heaps der Höhe 1, …, 2^0 Heap der Höhe H auf



Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse):



- In einem Heap mit Höhe H gibt es maximal
  2^0 Knoten mit Höhe H,
  …
  2^H Knoten mit Höhe H-H = 0 (Blätter)
  } H+1 Ebenen
- D.h. wir bauen 2^H Heaps der Höhe 0, 2^{H-1} Heaps der Höhe 1, …, 2^0 Heap der Höhe H auf
- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H=\lfloor \log n \rfloor$:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \cdots + (H-H) \cdot 2^H)$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe H=⌊log n⌋:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \cdots + \big(H-(H-1)\big) \cdot 2^{H-1} + (H-H) \cdot 2^H) =$$

$$O(0 \cdot 2^H + 1 \cdot 2^{H-1} + \cdots + (H-1) \cdot 2^1 + H \cdot 2^0) =$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe H=⌊log n⌋:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \cdots + \big(H-(H-1)\big) \cdot 2^{H-1} + (H-H) \cdot 2^H) =$$

$$O(0 \cdot 2^H + 1 \cdot 2^{H-1} + \cdots + (H-1) \cdot 2^1 + H \cdot 2^0) =$$

$$O(\textstyle\sum_{h=0}^{H} h \cdot 2^{H-h}) =$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H=\lfloor \log n \rfloor$:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \cdots + \big(H-(H-1)\big) \cdot 2^{H-1} + (H-H) \cdot 2^H) =$$

$$O(0 \cdot 2^H + 1 \cdot 2^{H-1} + \cdots + (H-1) \cdot 2^1 + H \cdot 2^0) =$$

$$O(\sum_{h=0}^{H} h \cdot 2^{H-h}) = O(\sum_{h=0}^{H} \frac{h \cdot 2^H}{2^h}) =$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe H=⌊log n⌋:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \cdots + \big(H-(H-1)\big) \cdot 2^{H-1} + (H-H) \cdot 2^H) =$$

$$O(0 \cdot 2^H + 1 \cdot 2^{H-1} + \cdots + (H-1) \cdot 2^1 + H \cdot 2^0) =$$

$$O(\sum_{h=0}^{H} h \cdot 2^{H-h}) = O(\sum_{h=0}^{H} \frac{h \cdot 2^H}{2^h}) = O(2^H \cdot \sum_{h=0}^{H} \frac{h}{2^h}) =$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe H=⌊log n⌋:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \cdots + \big(H-(H-1)\big) \cdot 2^{H-1} + (H-H) \cdot 2^H) =$$

$$O(0 \cdot 2^H + 1 \cdot 2^{H-1} + \cdots + (H-1) \cdot 2^1 + H \cdot 2^0) =$$

$$O(\sum_{h=0}^{H} h \cdot 2^{H-h}) = O(\sum_{h=0}^{H} \frac{h \cdot 2^H}{2^h}) = O(2^H \cdot \sum_{h=0}^{H} \frac{h}{2^h}) =$$

Max. Anzahl der Knoten in einem bin. Baum der Höhe H: $n = 2^{H+1} - 1$

$$2^H < 2^{H+1} - 1 \Longrightarrow 2^H < n \Rightarrow O\big(2^h\big) = O(n)$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H=\lfloor \log n \rfloor$:

$$O(H \cdot 2^0 + (H-1) \cdot 2^1 + \cdots + \big(H-(H-1)\big) \cdot 2^{H-1} + (H-H) \cdot 2^H) =$$

$$O(0 \cdot 2^H + 1 \cdot 2^{H-1} + \cdots + (H-1) \cdot 2^1 + H \cdot 2^0) =$$

$$O(\sum_{h=0}^{H} h \cdot 2^{H-h}) = O(\sum_{h=0}^{H} \frac{h \cdot 2^H}{2^h}) = O(2^H \cdot \sum_{h=0}^{H} \frac{h}{2^h}) =$$

$$O(n \cdot \sum_{h=0}^{H} \frac{h}{2^h})$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H=\lfloor \log n \rfloor$:

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

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H=\lfloor \log n \rfloor$:

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

Ableitung einer geometrischen partiellen Summe

https://de.wikipedia.org/wiki/Geometrische_Reihe#Herleitung_der_Varianten

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H=\lfloor \log n \rfloor$:

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

Ableitung einer geometrischen Reihe

$$\sum_{k=0}^{\infty} k \cdot q^k = \frac{q}{(1-q)^2} \qquad |q| < 1$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe $H=\lfloor \log n \rfloor$:

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

Ableitung einer geometrischen Reihe

$$\sum_{k=0}^{\infty} k \cdot q^k = \frac{q}{(1-q)^2} \qquad |q| < 1$$

$$\sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = \frac{\frac{1}{2}}{\frac{1}{4}} = 2$$

Heap – Aufbau

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

Beweis (Schärfere Laufzeitanalyse): (Fortsetzung)

- Damit ergibt sich als Gesamtlaufzeit bei n Knoten und Höhe H=⌊log n⌋:

$$O(n \cdot \underbrace{\sum_{h=0}^{H} \frac{h}{2^h}}_{C \Rightarrow O(1)})$$

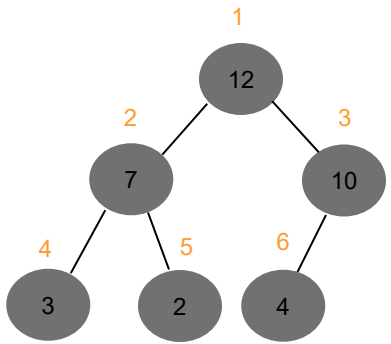
⇒ Laufzeit ist in $O(n)$

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

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

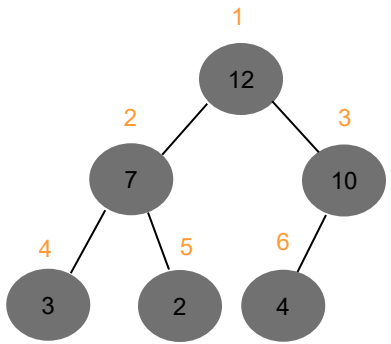


Max Heap – Extract Max

Heap-Extract-Max(A)

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

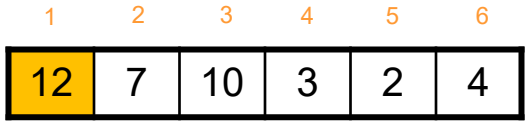
1	2	3	4	5	6
12	7	10	3	2	4



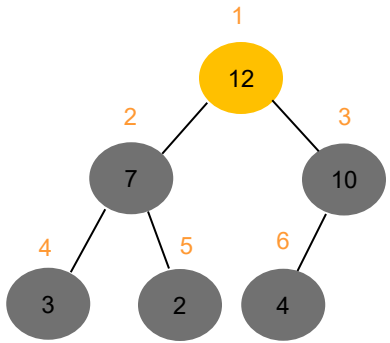
Max Heap – Extract Max

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



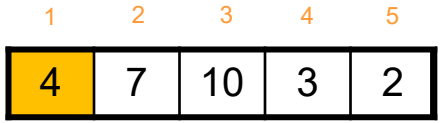
max =12



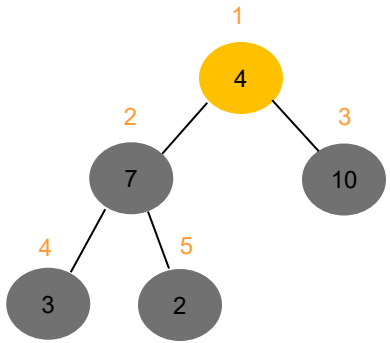
Max Heap – Extract Max

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



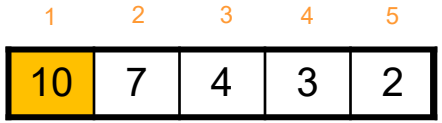
max =12



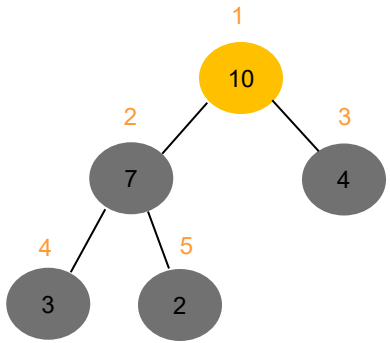
Max Heap – Extract Max

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



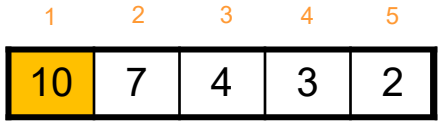
max =12



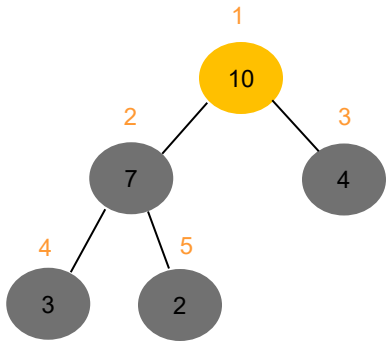
Max Heap – Extract Max

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

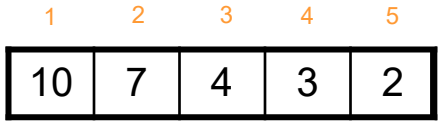


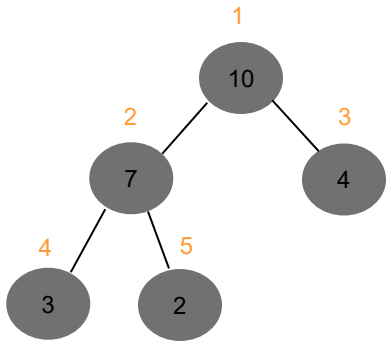
Max Heap – Extract Max

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))

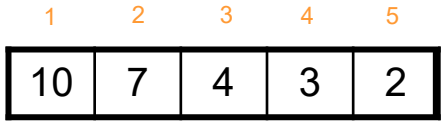


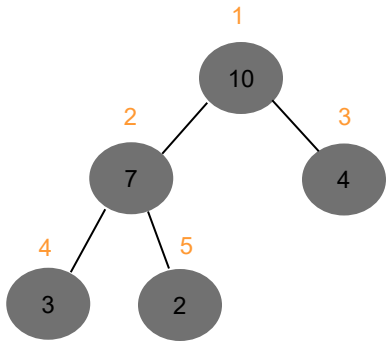


Max Heap – Einfügen

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

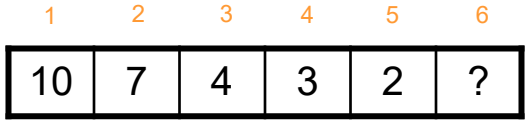




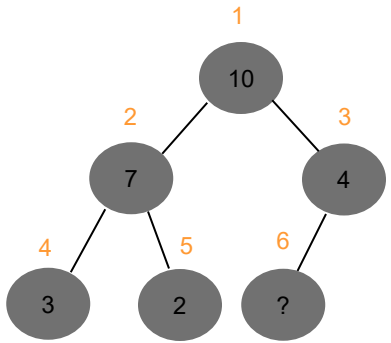
Max Heap – Einfügen

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



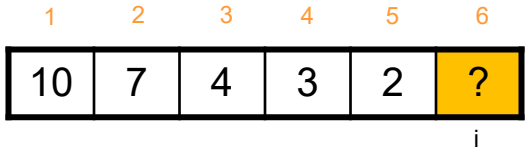
Heap-Insert(A,11)



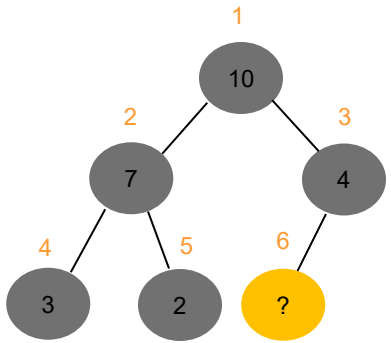
Max Heap – Einfügen

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



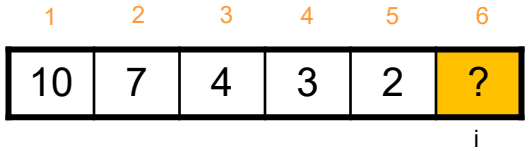
Heap-Insert(A,11)



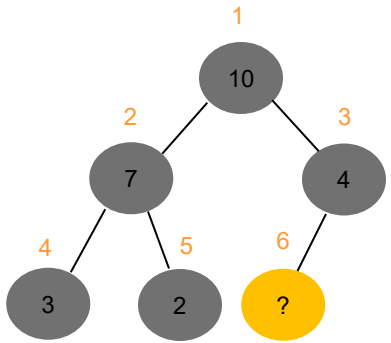
Max Heap – Einfügen

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



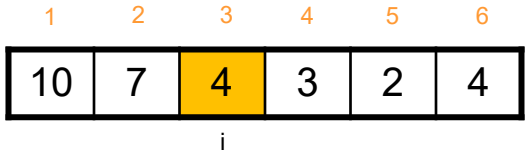
Heap-Insert(A,11)



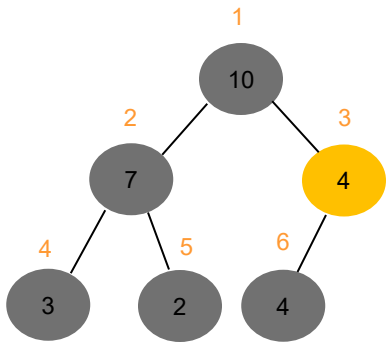
Max Heap – Einfügen

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



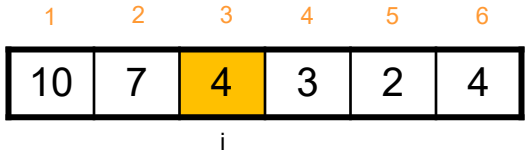
Heap-Insert(A,11)



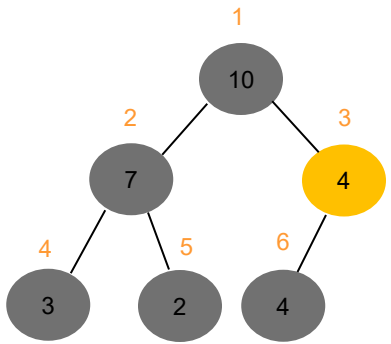
Max Heap – Einfügen

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



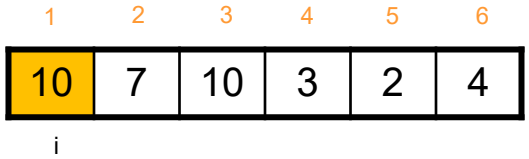
Heap-Insert(A,11)



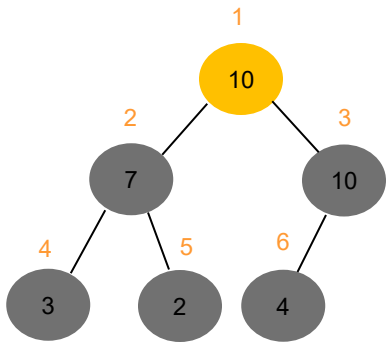
Max Heap – Einfügen

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



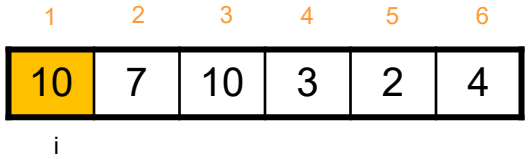
Heap-Insert(A,11)



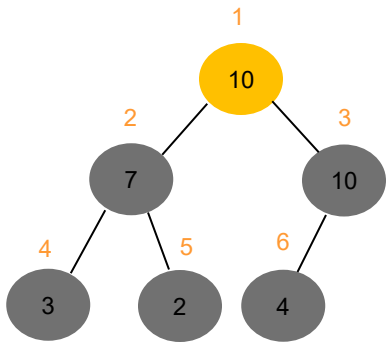
Max Heap – Einfügen

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



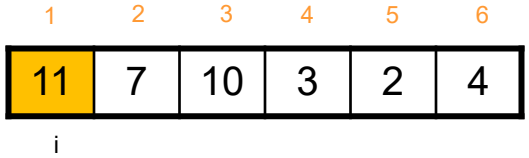
Heap-Insert(A,11)



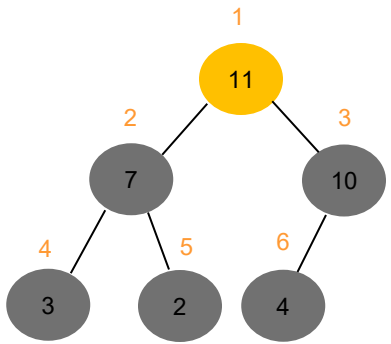
Max Heap – Einfügen

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



Heap-Insert(A,11)

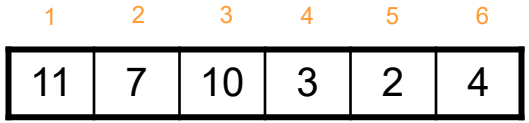


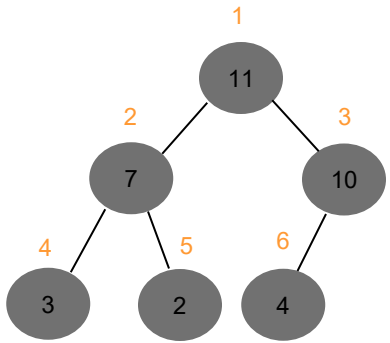
Max Heap – Einfügen

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))

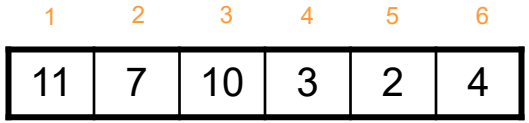


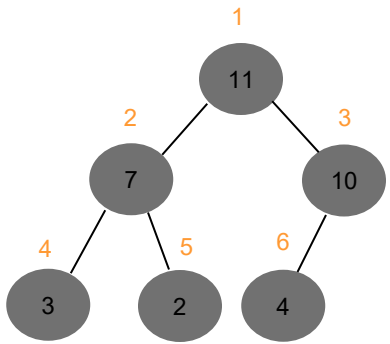


Heapsort

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)



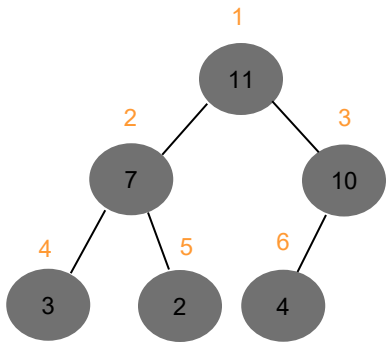


Heapsort

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)

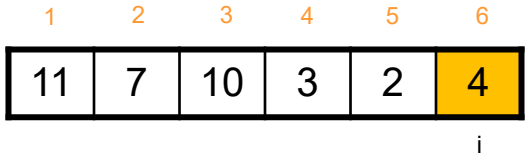
1	2	3	4	5	6
11	7	10	3	2	4

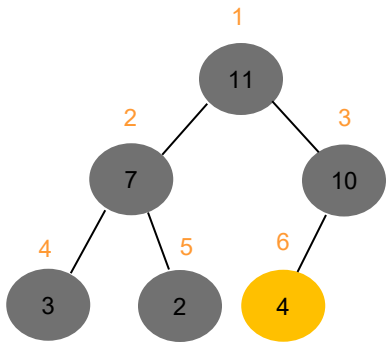


Heapsort

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)

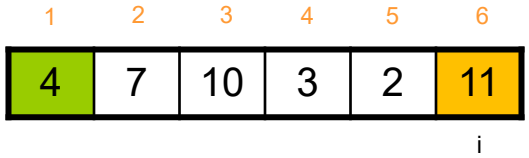


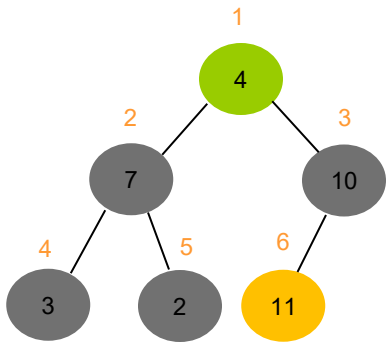


Heapsort

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)

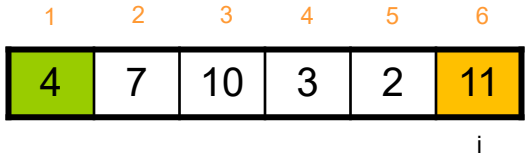


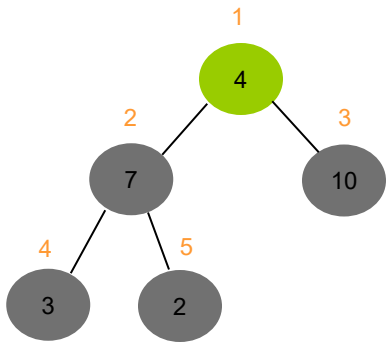


Heapsort

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)

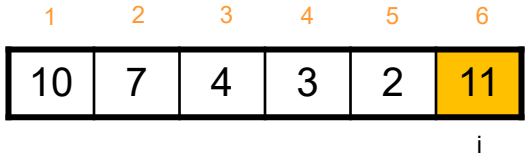


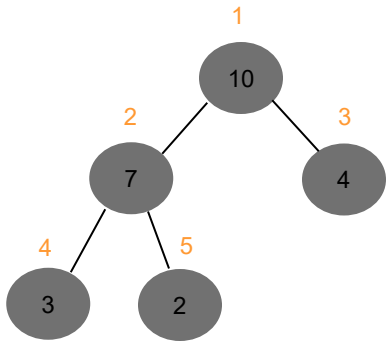


Heapsort

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)

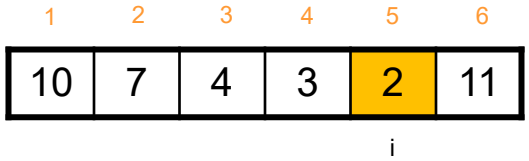


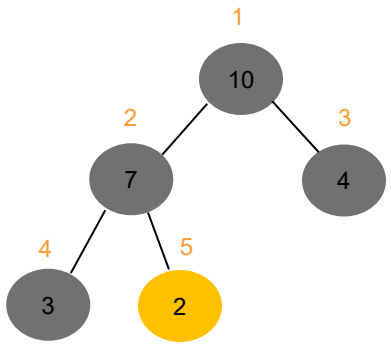


Heapsort

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)

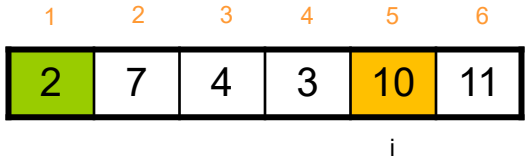


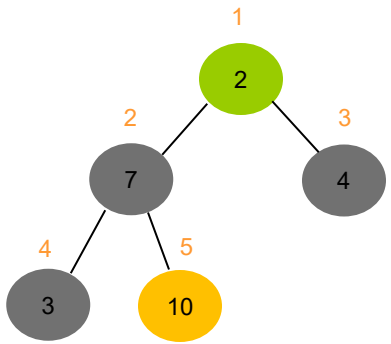


Heapsort

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)

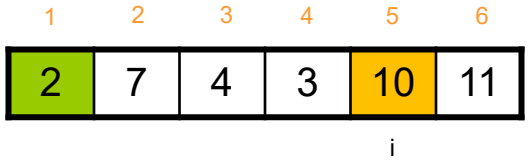


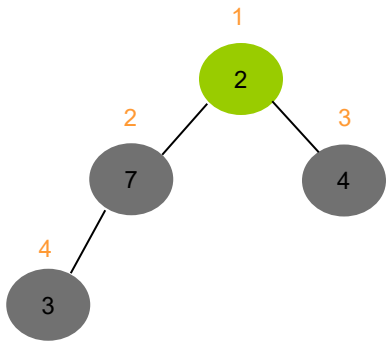


Heapsort

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)

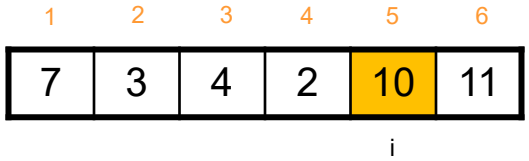


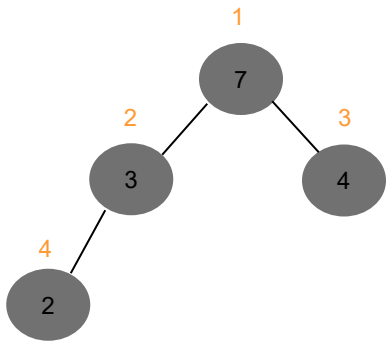


Heapsort

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)

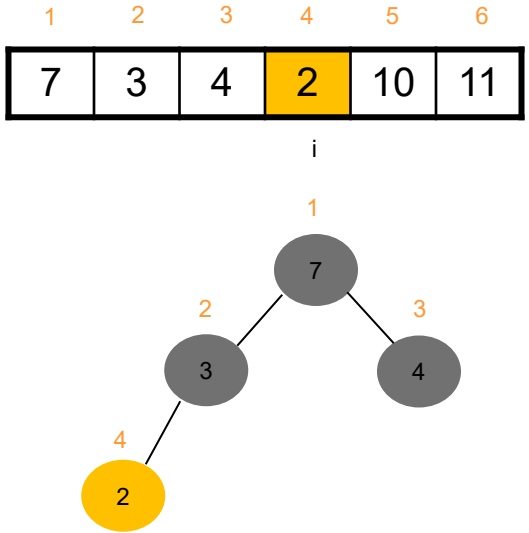




Heapsort

Heapsort(A)

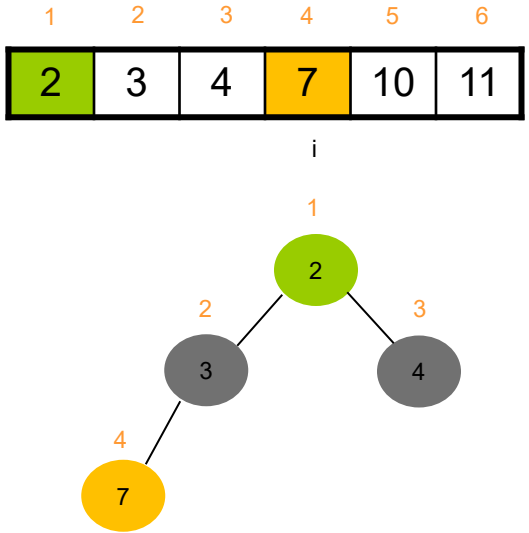
1. Build-Heap(A)
2. for i ← length(A) downto 2 do
3. A[1] ↔ A[i]
4. heap-size(A) ← heap-size(A)-1
5. Heapify(A,1)



Heapsort

Heapsort(A)

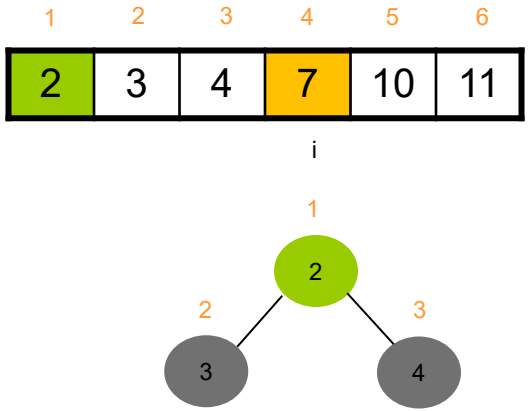
1. Build-Heap(A)
2. for i ← length(A) downto 2 do
3. A[1] ↔ A[i]
4. heap-size(A) ← heap-size(A)-1
5. Heapify(A,1)



Heapsort

Heapsort(A)

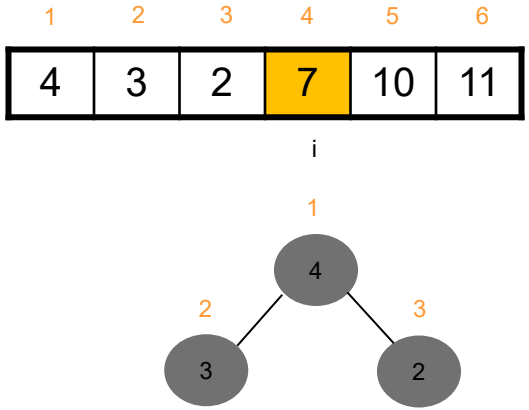
1. Build-Heap(A)
2. for i ← length(A) downto 2 do
3. A[1] ↔ A[i]
4. heap-size(A) ← heap-size(A)-1
5. Heapify(A,1)



Heapsort

Heapsort(A)

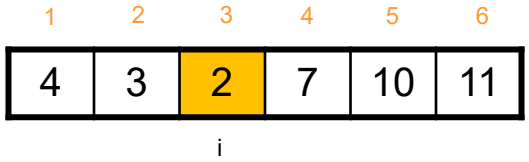
1. Build-Heap(A)
2. for i ← length(A) downto 2 do
3. A[1] ↔ A[i]
4. heap-size(A) ← heap-size(A)-1
5. Heapify(A,1)

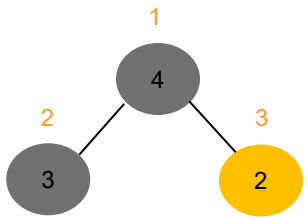


Heapsort

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)

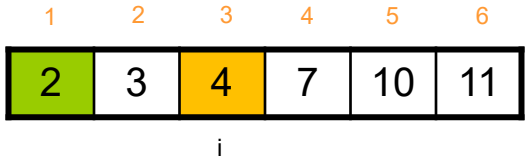


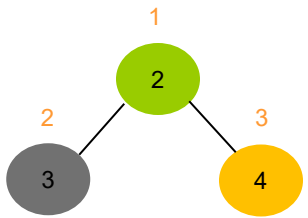


Heapsort

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)

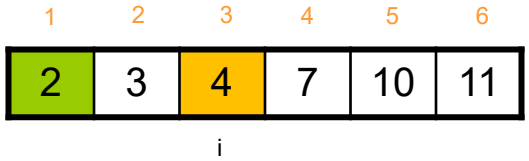


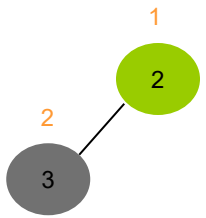


Heapsort

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)

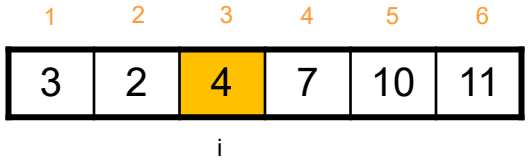


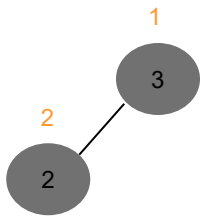


Heapsort

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)

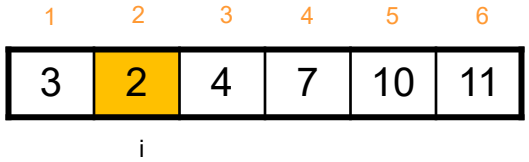


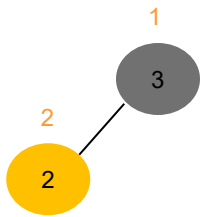


Heapsort

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)

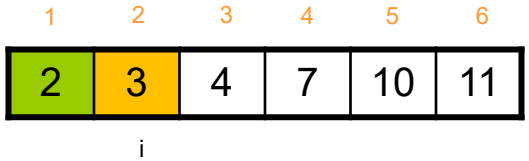


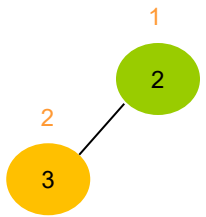


Heapsort

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)

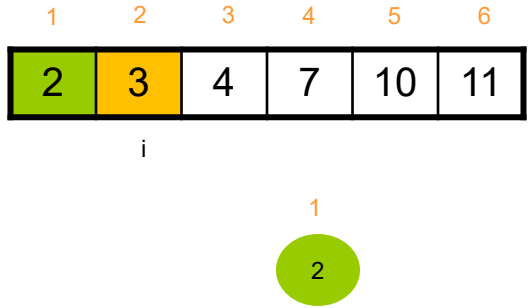




Heapsort

Heapsort(A)

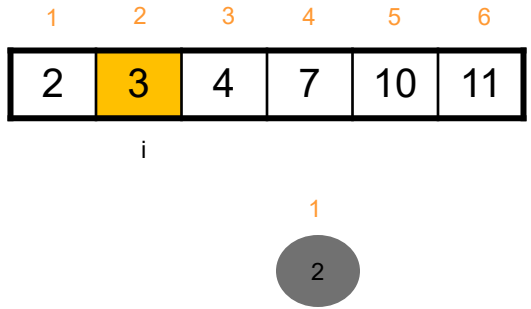
1. Build-Heap(A)
2. for i ← length(A) downto 2 do
3. A[1] ↔ A[i]
4. heap-size(A) ← heap-size(A)-1
5. Heapify(A,1)



Heapsort

Heapsort(A)

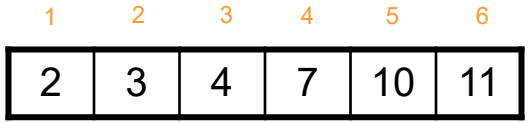
1. Build-Heap(A)
2. for i ← length(A) downto 2 do
3. A[1] ↔ A[i]
4. heap-size(A) ← heap-size(A)-1
5. Heapify(A,1)



Heapsort

Heapsort(A)

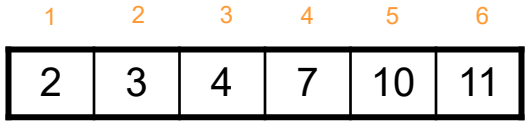
1. Build-Heap(A)
2. for i ← length(A) downto 2 do
3. A[1] ↔ A[i]
4. heap-size(A) ← heap-size(A)-1
5. Heapify(A,1)



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)



Laufzeit: O(n log(n))

Zusammenfassung Heaps / Halden

- 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 einer Datenstruktur und eines Algorithmus

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Algorithmen, Pseudocode, Sortieren I“: Insertion Sort
VL 2 „Algorithmen, Pseudocode, Sortieren II“: Selection Sort, Bubble Sort, Count Sort
VL 3 „Laufzeit und Speicherplatz“: Laufzeitanalyse der vorgestellten Sortierverfahren
VL 4 „Einfache Datenstrukturen“: Arrays, verkettete Listen, Structs in C, Stack, Queue
VL 5 „Bäume“: Binärbäume, Baumtraversierung, Laufzeitanalyse Baumoperationen
VL 6 „Teile und Herrsche I“: Einführung der algorithmischen Methode, Merge Sort
VL 7 „Korrektheitsbeweise“: Rechnermodel, Beispielbeweise
VL 8 „Dateien in C“: Dateien, Dateisysteme, Verzeichnisse, Dateiverwaltung mit C
VL 9 „Prioritätenschlangen/Halden/Heaps “: Heap Sort, Binärer Heap, Heap Operationen
VL 10 „Fortgeschrittene Sortierverfahren“: Quick Sort, Radix Sort
VL 11 „AVL Bäume“: Definition, Baumoperationen, Traversierung
VL 12 „Teile und Herrsche II“: Generalisierung des algorithmischen Prinzips, Mastertheorem
VL 13 „Q & A“: Offene Vorlesung/Wiederholung