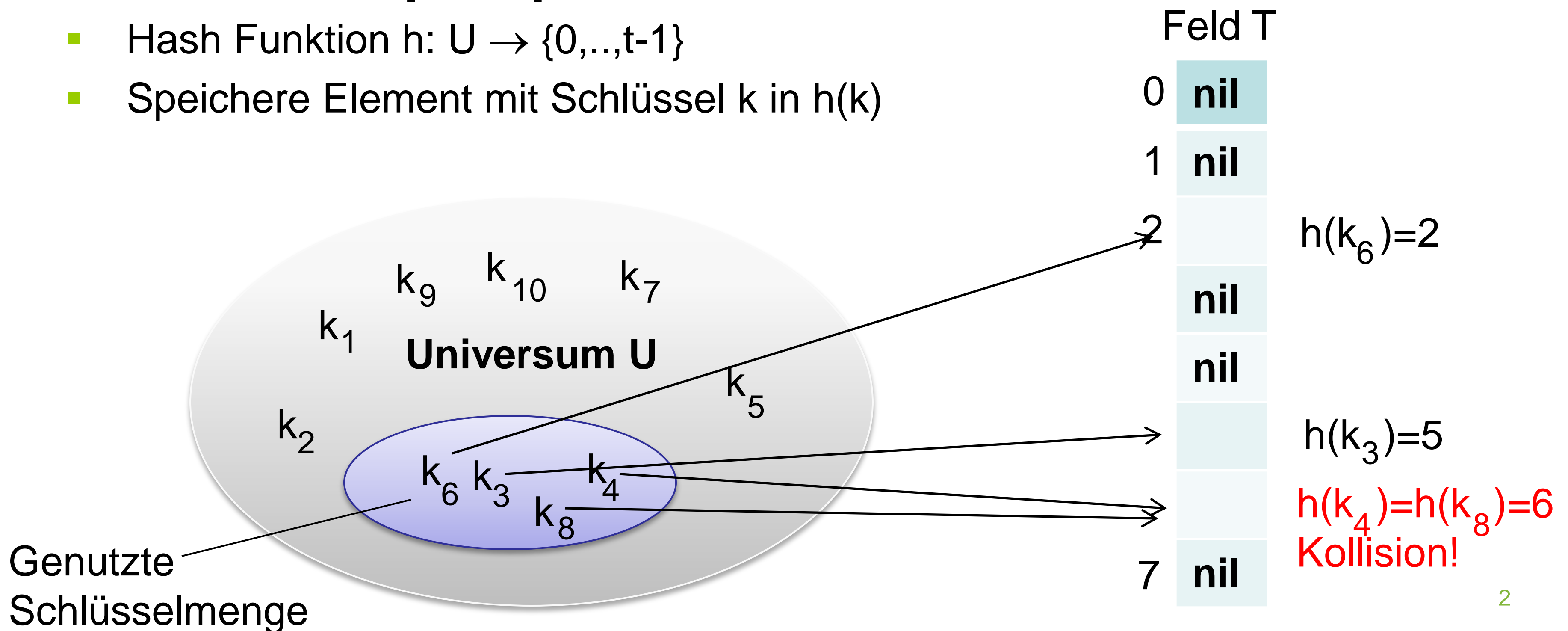
Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Datenstrukturen

Hashing

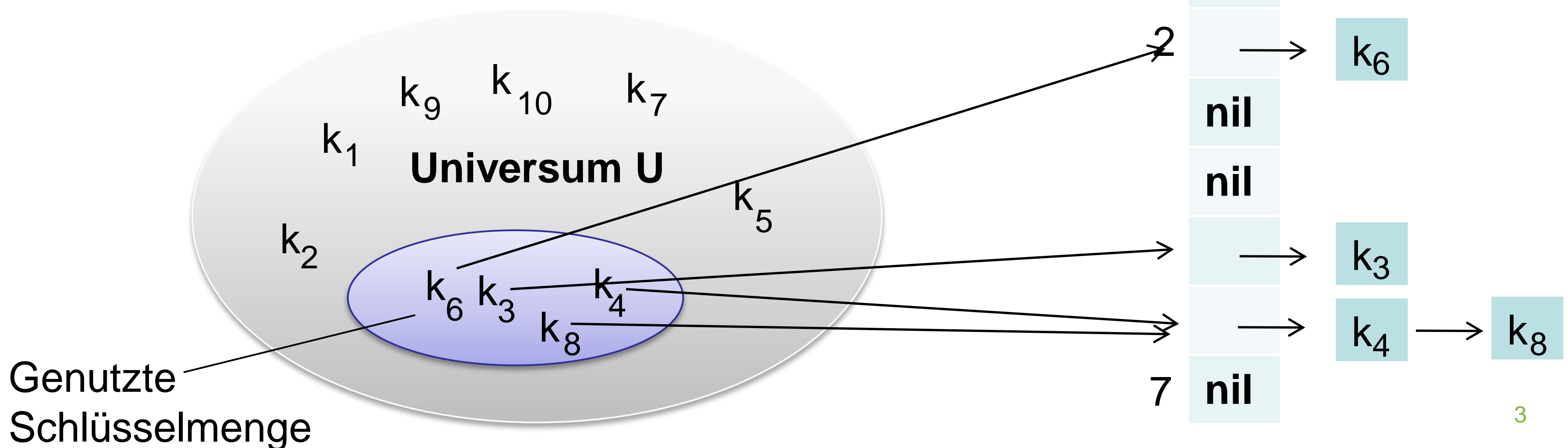
- Universum U={0,..,m-1}
- Hash Tabelle T[0,..,t-1]
- Hash Funktion h: U $\rightarrow$ {0,..,t-1}
- Speichere Element mit Schlüssel k in h(k)



Datenstrukturen

Hashing mit Verkettung

- Universum U={0,..,m-1}
- Hash Tabelle T[0,..,t-1]
- Hash Funktion h: U → {0,..,t-1}
- Speichere Element mit Schlüssel k in h(k)
- Löse Kollisionen durch Listen auf (wie vorhin)



Datenstrukturen

Operationen

Einfügen(k)

- Füge neuen Schlüssel k am Ende der Liste T[h(k)] ein

Löschen(x)

- Lösche Element x aus Liste T[h(key[x])]

Suche(k)

- Suche nach k in Liste T[h(k)]

Datenstrukturen

Idee

- Wähle h zufällig (aus einer Menge von geeigneten Kandidaten H)

Datenstrukturen

Universelles Hashing

- Sei H eine Menge von Hashfunktionen von U nach {0,..,t-1}. Die Menge H heißt **universell**, wenn für jedes Paar von unterschiedlichen Schlüsseln $x,y \in U$ gilt, dass die Anzahl der Funktion $h \in H$ mit $h(x)=h(y)$ genau $|H|/t$ ist.

Anders gesagt:

- Wenn man x und y vorgibt und dann ein zufälliges $h \in H$ wählt, so ist die Kollisionswahrscheinlichkeit von x und y genau 1/t
- Oder: Die durchschnittliche Anzahl Kollisionen (über H) von x und y ist 1/t

Datenstrukturen

Satz 43

- Sei $M \subseteq U$ eine Menge von n Schlüsseln, T eine Tabelle mit $t \geq n$ Einträgen und sei H eine universelle Klasse von Hashfunktionen von U nach {0,..,t-1}. Wenn h aus H zufällig ausgewählt wird und danach zum Speichern von M in T benutzt wird, so ist die durchschnittliche Anzahl Kollisionen eines vorher festgelegten Schlüssels x höchstens 1.

Beweis

- Für jedes Paar y,z sei c(y,z)=1, wenn h(y)=h(z) gilt
- Der durchschn. Wert **E**[c(y,z)] von c(y,z) (über H) ist 1/t
- Sei nun C(x) die Anzahl Kollisionen mit Schlüssel x, d.h.
- $C(x) = \sum_{y \in M, y \neq x} c(x,y)$

Datenstrukturen

Beweis

- Für jedes Paar y,z sei c(y,z)=1, wenn h(y)=h(z) gilt
- Der durchschn. Wert **E**[c(y,z)] von c(y,z) (über H) ist 1/t
- Sei nun C(x) die Anzahl Kollisionen mit Schlüssel x, d.h.
- $C(x) = \sum_{y \in M, y \neq x} c(x,y)$
- Damit ist der durchschn. Wert **E**[C(x)] von C(x)
- $\mathbf{E}[C(x)] = \sum_{y \in M, y \neq x} \mathbf{E}[c(x,y)] \leq n / t$
- Da n≤t folgt **E**[C(x)] ≤ 1.

Datenstrukturen

Konstruktion von H (ohne Beweis)

- Setze Tabellengröße t auf Primzahl
- Teile Schlüssel x in r+1 bytes $x_0, x_1, \ldots, x_r$ auf, so dass
  (a) $x = \langle x_0, x_1, \ldots, x_r \rangle$ und
  (b) jedes Byte Maximalwert höchstens t hat
- Sei $a = \langle a_0, a_1, \ldots, a_r \rangle$ Sequenz von r+1 zufälligen Werten aus $\{0, \ldots, t-1\}$
- Definiere Hashfunktion $h_a \in H$:
- $h_a(x) = \sum a_i \cdot x_i \bmod t$
- Damit ist H die Vereinigung der h_a und $|H| = t^{r+1}$
- Funktioniert bis Universumsgröße t^{r+1}
- Also $|H| \approx |U|$, d.h. wir benötigen bei guter Speicherung soviel Bits um |H| zu speichern wie ein Schlüssel aus U groß ist

Datenstrukturen

Zusammenfassung

- Hashing nutzt Vorteil der direkten Addressierung (O(1) Suchzeit), reduziert aber gleichzeitig den Speicherbedarf auf O(n)
- Hashfunktion kann zufällig aus universeller Klasse von Hashfunktionen gewählt werden
- Dies garantiert durchschnittliche Suchzeit O(1)

Datenstrukturen

Prioritätenschlangen

- Einfügen, Löschen, Maximum (bzw. Minimum)
- Wir könnten AVL-Bäume benutzen
- Gibt es einfachere Datenstruktur?
- Gibt es effizientere Datenstruktur?
  (Ja, aber nicht in dieser Vorlesung)

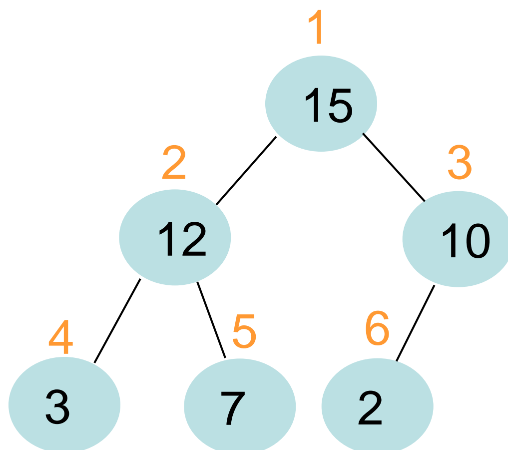
Anwendungen

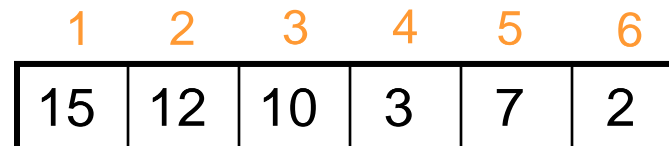
- Ereignisgesteuerte Simulationen
- Sortieren mit Heapsort

Datenstrukturen

Binäre Halden

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





Datenstrukturen (siehe Vollversion)

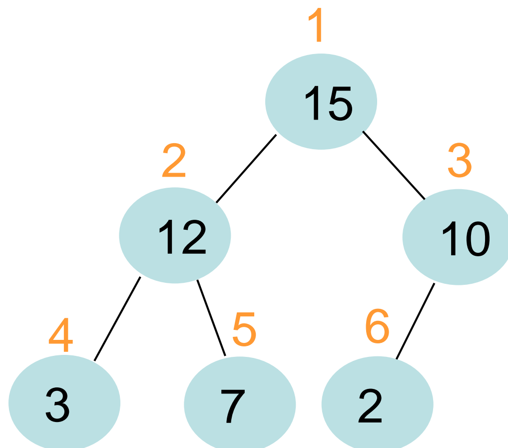
Navigation

- Wurzel ist A[1]

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

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

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

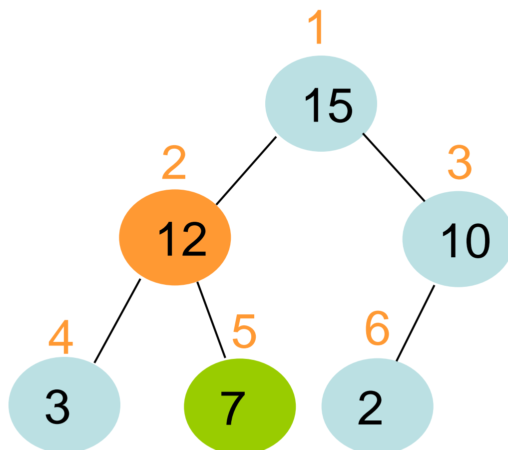


1	2	3	4	5	6
15	12	10	3	7	2

Datenstrukturen

Haldeneigenschaft

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

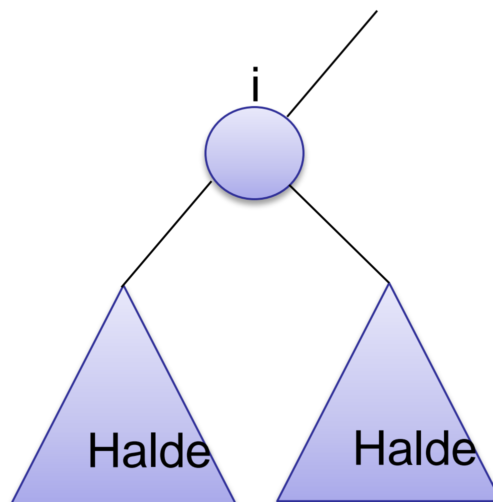


1	2	3	4	5	6
15	12	10	3	7	2

Datenstrukturen

Die Operation Heapify(A,i)

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

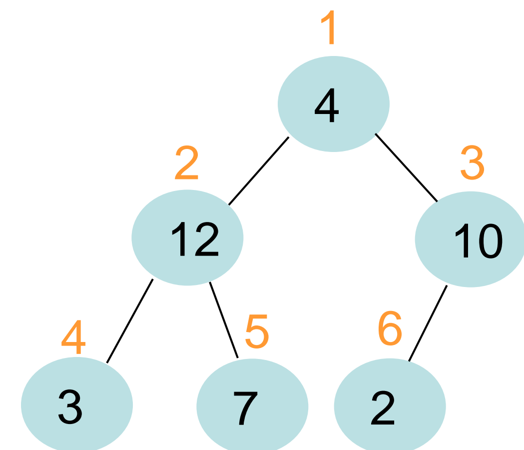


Datenstrukturen (siehe Vollversion)

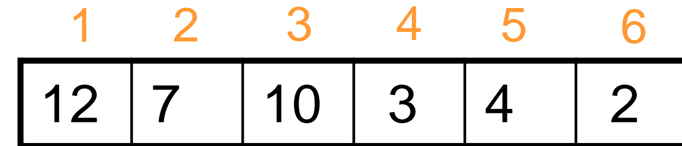
1	2	3	4	5	6
4	12	10	3	7	2

Heapify(A,i)

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

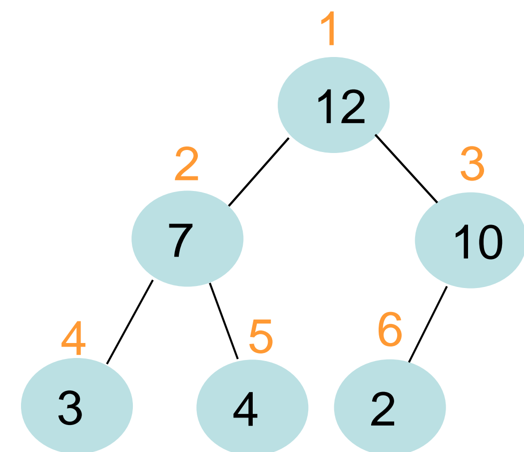


Datenstrukturen (siehe Vollversion)



Heapify(A,i)

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



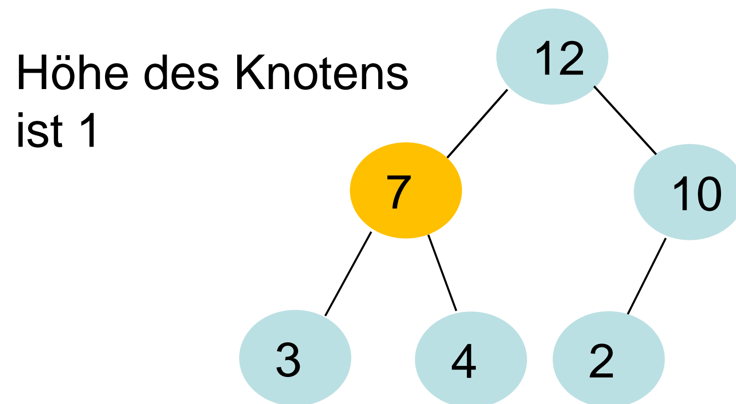
Heapify(A,1)

Datenstrukturen

Definition

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

Beispiel



Datenstrukturen

Satz 44

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

Beweis

- Zeige per Induktion über h, das die Laufzeit ≤c·h ist
(für c hinreichend große Konstante)
- (I.A.) Ist h=0, so wird in Zeile 4 largest auf i gesetzt. In Zeile 5 wird dieser Wert nicht verändert. Daher wird in Zeile 6 kein rekursiver Aufruf durchgeführt und die Laufzeit ist c.
- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch.

Datenstrukturen

Satz 44

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

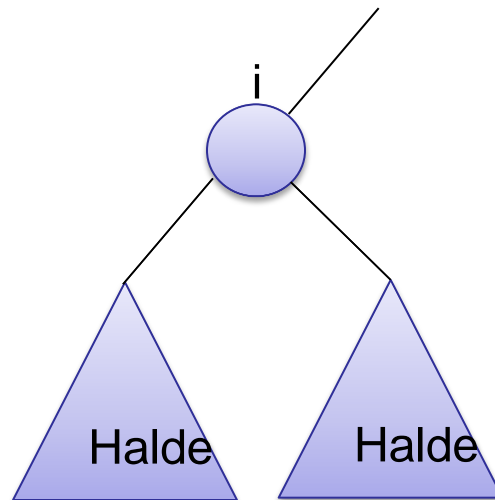
Beweis

- (I.V.) Für Knoten der Höhe h ist die Laufzeit ch.
- (I.S.) Betrachte einen Knoten der Höhe h+1. In Zeile 3-5 wird largest entweder auf i oder auf den Wert eines der Kinder von i gesetzt. Wird nun in Zeile 6 ein rekursiver Aufruf durchgeführt, so geschieht dies mit einem der Kinderknoten. Diese haben Höhe h und daher hat der rekursive Aufruf nach (I.V.) eine Laufzeit von ch. Die übrige Laufzeit ist durch c beschränkt. Damit ergibt sich eine Laufzeit von maximal ch+c = c(h+1).

Datenstrukturen

Satz 45

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



Datenstrukturen

Satz 45

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

- *Beweis*
- Induktion über die Höhe von i.
- (I.A.) Hat i Höhe 0 oder 1, so kann man einfach nachprüfen, dass Heapify die Aussage des Satzes erfüllt.
- (I.V.) Heapify erfüllt die Aussage des Satzes für Knoten i mit Höhe h.

Datenstrukturen

Satz 45

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

- *Beweis*
- (I.S.) Betrachte Aufruf Heapify(A,i) für einen Knoten i der Höhe h+1>1 und wenn die Unterbäume des rechten bzw. linken Kindes von i bereits die Haldeneigenschaft erfüllen. Es seien l und r das linke bzw. rechte Kind von i. Da die Höhe von i größer 1 ist, sind l und r kleiner als heap-size[A] sind die an l und r gespeicherten Werte in der Halde. Es seien A[i], A[l] und A[r] die Werte, die an den entsprechenden Knoten gespeichert wurden. Heapify(A,i) speichert in Zeilen 3-5 den Index des Maximums von A[i], A[l] und A[r] in der Variable largest. Ist das Maximum A[i], so ist die Haldeneigenschaft bereits erfüllt, da die Unterbäume von l und r bereits Halden sind.

Datenstrukturen

Satz 45

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

- *Beweis*
- Es bleibt der Fall, dass A[i] nicht das Maximum ist. Wir betrachten den Fall, dass das Maximum in A[l] ist. Der andere Fall ist analog. Da die Unterbäume von l und r Halden sind, sind A[l] und A[r] die größten Elemente, die in den jeweiligen Unterbäumen gespeichert sind. Da A[l] das Maximum von A[i], A[l] und A[r] ist, ist es das größte Element im Unterbaum von i. In Zeile 6 wird nun A[i] mit A[l] getauscht und in Zeile 7 wird rekursiv Heapify für den Unterbaum des linken Kindes von l aufgerufen.

Datenstrukturen

Satz 45

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

- *Beweis*
- In Zeile 6 wird nun A[i] mit A[l] getauscht und in Zeile 7 wird rekursiv Heapify für den Unterbaum des linken Kindes von i aufgerufen. Dieses hat Höhe h und die Unterbäume seiner Kinder besitzen die Haldeneigenschaft. Somit besitzt der Unterbaum von l nach (I.V.) nach dem Aufruf von Heapify ebenfalls die Halden-eigenschaft. Da an Knoten i nun das Maximum aller Elemente im Unterbaum von i gespeichert ist und der rechte Unterbaum ebenfalls die Haldeneigenschaft erfüllt, ist der Unterbaum von i eine Halde.

Datenstrukturen (siehe Vollversion)

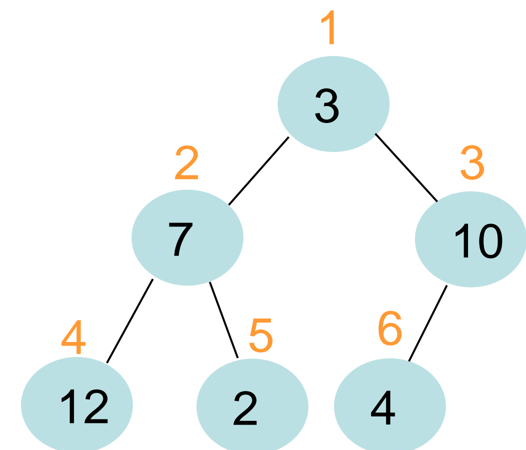
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau einer Halde

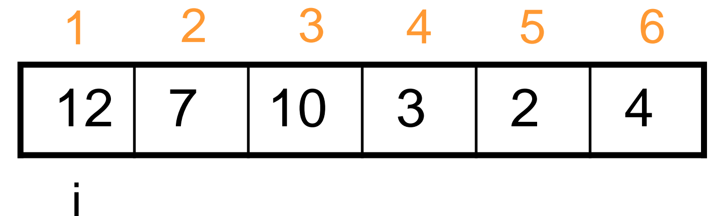
- Jedes Blatt ist eine Halde
- Baue Halde „von unten nach oben“ mit Heapify auf

Build-Heap(A)

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



Datenstrukturen (siehe Vollversion)

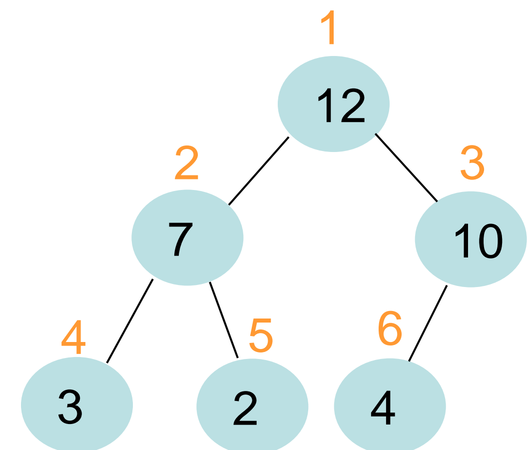


Aufbau einer Halde

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

Build-Heap(A)

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



Datenstrukturen

Satz 46

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

- *Beweis*
- Wir zeigen zunächst die Korrektheit. In der for-Schleife gilt die Invariante, dass die Unterbäume der Knoten größer als i bereits die Haldeneigenschaft besitzen. Dies gilt insbesondere für die Unterbäume der Kinder von i. Aus unserem vorherigen Satz folgt, dass die Invariante durch Heapify aufrechterhalten wird. Damit gilt am Ende der Schleife die Haldeneigenschaft für die Wurzel und somit erzeugt Build-Heap eine Halde.
- Einfache Laufzeitanalyse: Jedes Heapify benötigt O(h) = O(log n) Laufzeit. Da es insgesamt O(n) Heapify Operationen gibt, ist die Laufzeit O(n log n).

Datenstrukturen

Satz 46

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

- *Beweis*
- Schärfere Laufzeitanalyse:
- Beobachtung: In einer Halde mit Höhe H gibt es maximal 2^0 Knoten mit Höhe H, 2^1 Knoten mit Höhe H-1, 2^2 Knoten mit Höhe H-2, usw.

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

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

Datenstrukturen

Satz 46

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

- *Beweis*

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

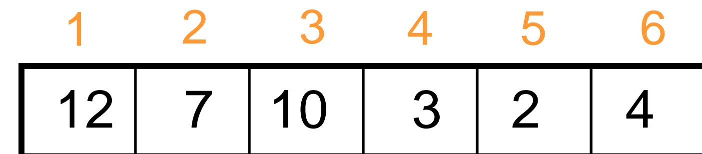
- Es gilt $\sum_{h=0}^{\infty} \frac{h}{2^h} = O(1)$

- Somit folgt eine Laufzeit von O(n).

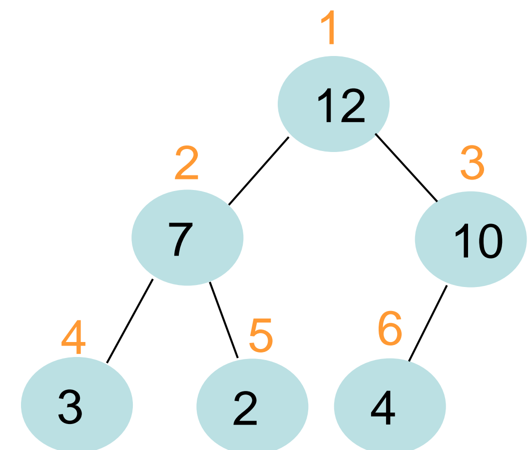
Datenstrukturen (siehe Vollversion)

Heap-Extract-Max(A)

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



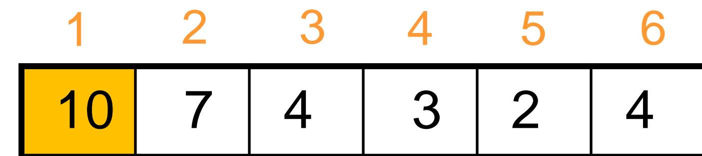
1	2	3	4	5	6
12	7	10	3	2	4



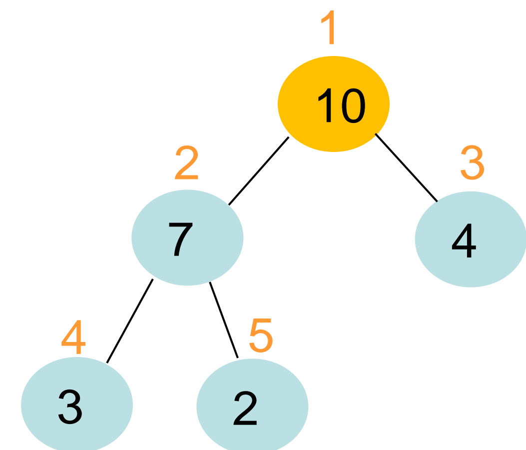
Datenstrukturen (siehe Vollversion)

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



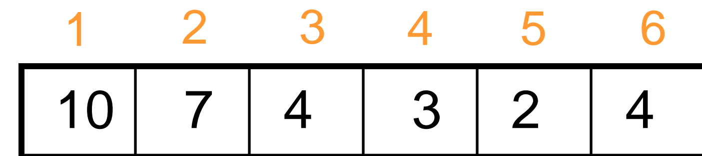
Laufzeit

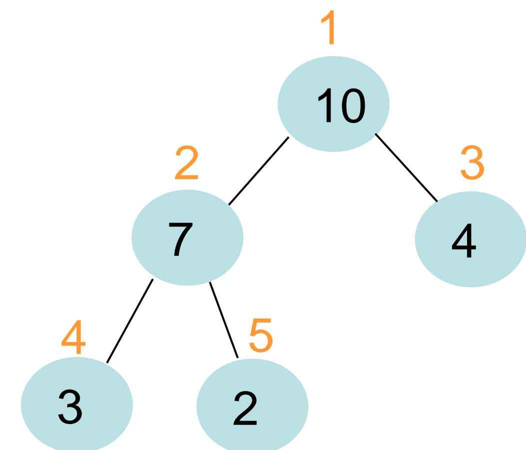
- O(log n)

Datenstrukturen (siehe Vollversion)

Heap-Insert(A,key)

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





Heap-Insert(A,11)

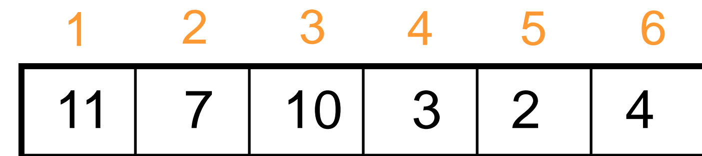
Datenstrukturen (siehe Vollversion)

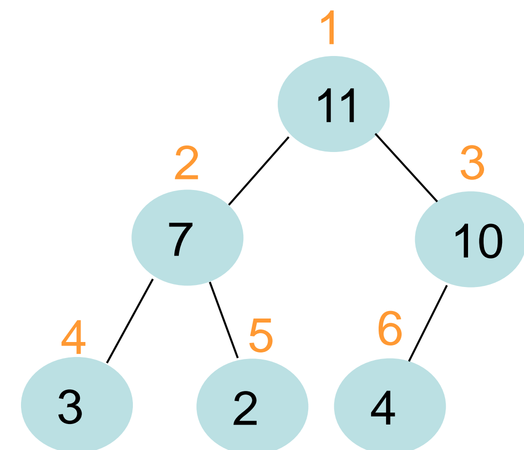
Heap-Insert(A,key)

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)





Heap-Insert(A,11)