Algorithmen & Komplexität

Johannes Lengler
Institut für Theoretische Informatik

Kapitel 4:

Datenstrukturen

Datenstrukturen

Suchbäume:

- Operationen: *Find, Insert, Delete, FindMin*

Mengen:

- Operationen: *Find, Insert, Union*

Wörterbücher:

- Operationen: *Find, Insert, Delete*

Vorrangwarteschlangen:

- Operationen: *Insert, ExtractMin, DecreaseKey*

Kapitel 4.4:

Vorrangwarteschlangen (Priority Queues)

Algorithmus von Dijkstra

Algorithmus 2.5 Algorithmus von Dijkstra

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$ mit $\ell \geq 0$ und Knoten $s, t \in V$
Ausgabe: Ein kürzester s–t-Pfad in N

{ *Initialisierung* }
$W := \emptyset$; $\rho[s] := 0$; pred$[s] := nil$;
for all $v \in V \setminus \{s\}$ **do** $\rho[v] := \infty$; Insert(v,∞)

{ *Traversieren und Markieren* }
while $t \notin W$ **do**
{ *Traversieren* }
Finde ein $x_0 \in V \setminus W$ mit $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$; x_0 = ExtractMin
$W := W \cup \{x_0\}$;
{ *Markieren* }
for all $v \in \Gamma(x_0) \cap (V \setminus W)$ mit $\rho[v] > \rho[x_0] + \ell(x_0, v)$ **do**
$\rho[v] := \rho[x_0] + \ell(x_0, v)$; pred$[v] := x_0$; DecreaseKey(v, $\rho[x_0] + l(x_0,v)$)

Laufzeit:

O(n · Laufzeit von Insert + n · Laufzeit von ExtractMin +
m · Laufzeit von DecreaseKey)

Vorrangwarteschlangen - Realisierungen

	sortierte Liste	Feld/Liste (unsortiert)	Suchbaum
Insert			
Extract-Min			
Decrease-Key			

Vorrangwarteschlangen - Realisierungen

	sortierte Liste	Feld/Liste (unsortiert)	Suchbaum
Insert	O(n)		
Extract-Min	O(1)		
Decrease-Key	O(n)		

Vorrangwarteschlangen - Realisierungen

	sortierte Liste	Feld/Liste (unsortiert)	Suchbaum
Insert	O(n)	O(1)	
Extract-Min	O(1)	O(n)	
Decrease-Key	O(n)	O(1)	

Vorrangwarteschlangen - Realisierungen

	sortierte Liste	Feld/Liste (unsortiert)	Suchbaum
Insert	O(n)	O(1)	O(log n)
Extract-Min	O(1)	O(n)	O(log n)
Decrease-Key	O(n)	O(1)	O(log n)

Fibonacci-Heaps:

Insert:	O(1) *
ExtractMin:	O(log n) *
DecreaseKey:	O(1) *

* amortisierte Kosten

Fibonacci-Heaps: Eigenschaften

Satz:
Beginnend mit einem *leeren* Fibonacci-Heap benötigen
r Insert,
s Extract-Min,
t Decrease-Key
zusammen (amortisiert) Laufzeit $O(r + s + t \log n)$.

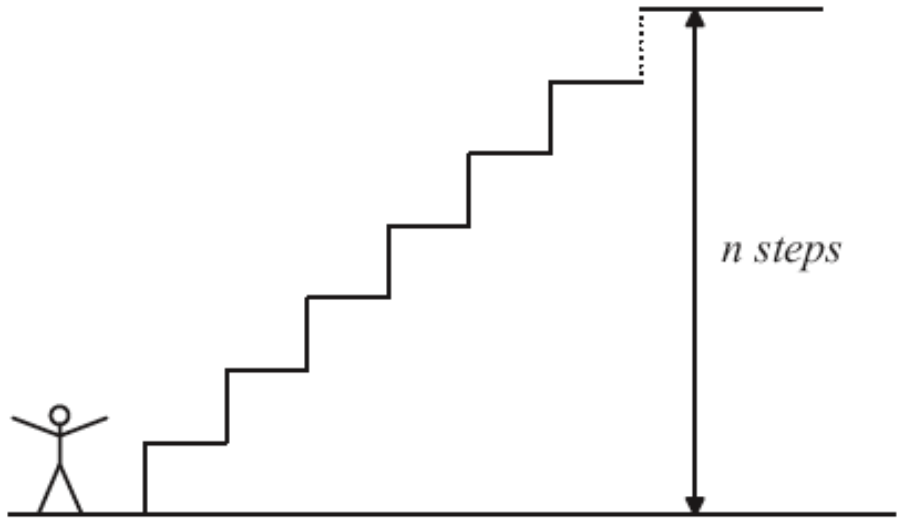
n = maximale Anzahl der Elemente im Fibonacci-Heap.

Für Algorithmus von Dijkstra/Prim heisst das:

n Insert,
n Extract-Min und
m Decrease-Key

können in Zeit $O(m+ n \log n)$ ausgeführt werden

Amortisierte Analyse - Beispiel



Annahmen:
- pro Stufe eine Sekunde, unabh. von Richtung
- wir beginnen unten

Zwei Arten von Operationen:

Rauf: Steige eine Stufe hinauf, falls noch nicht ganz oben.

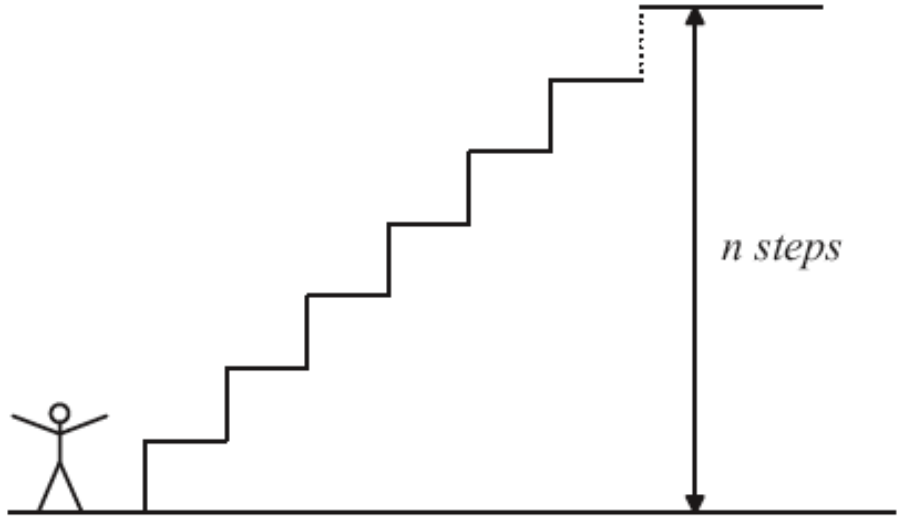
Runter: Gehe ganz nach unten

Frage:

Wie lange dauert eine Operation höchstens?

Wie lange dauert eine Folge von k Operationen höchstens?

Amortisierte Analyse - Beispiel



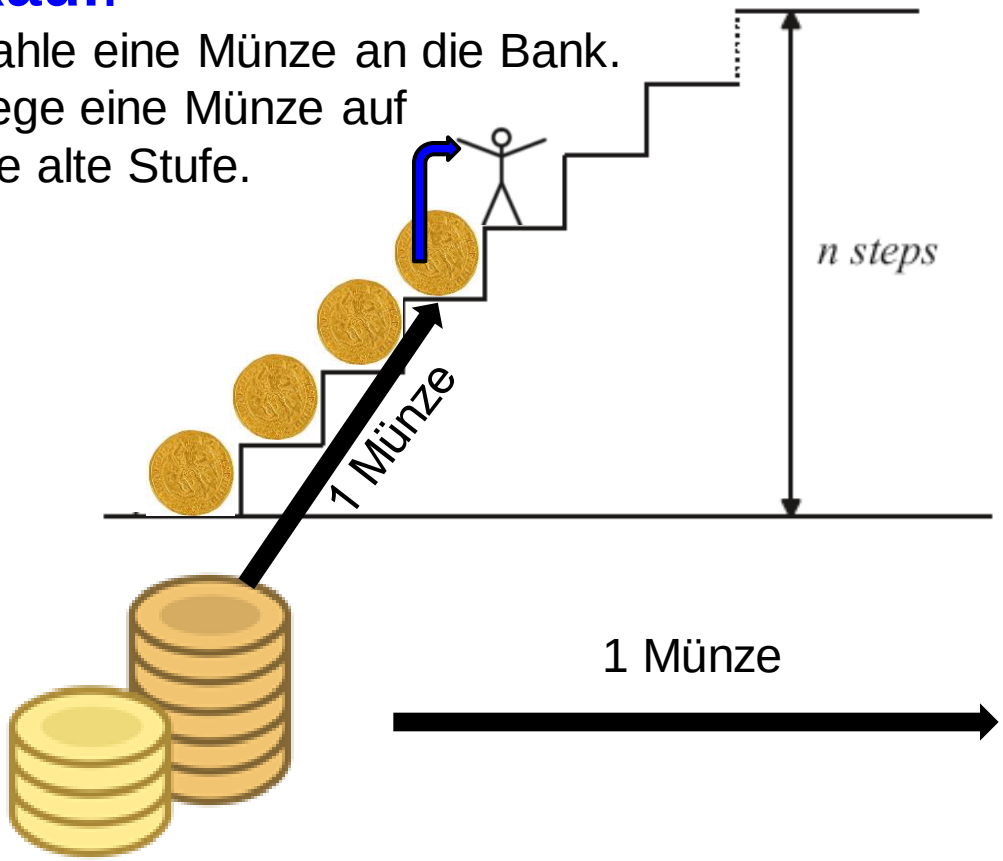
1 Münze pro Stufe

Unübersichtlich!
Für manche Operationen bezahlen wir 1 Münze, für manche n Münzen

Amortisierte Analyse - Beispiel

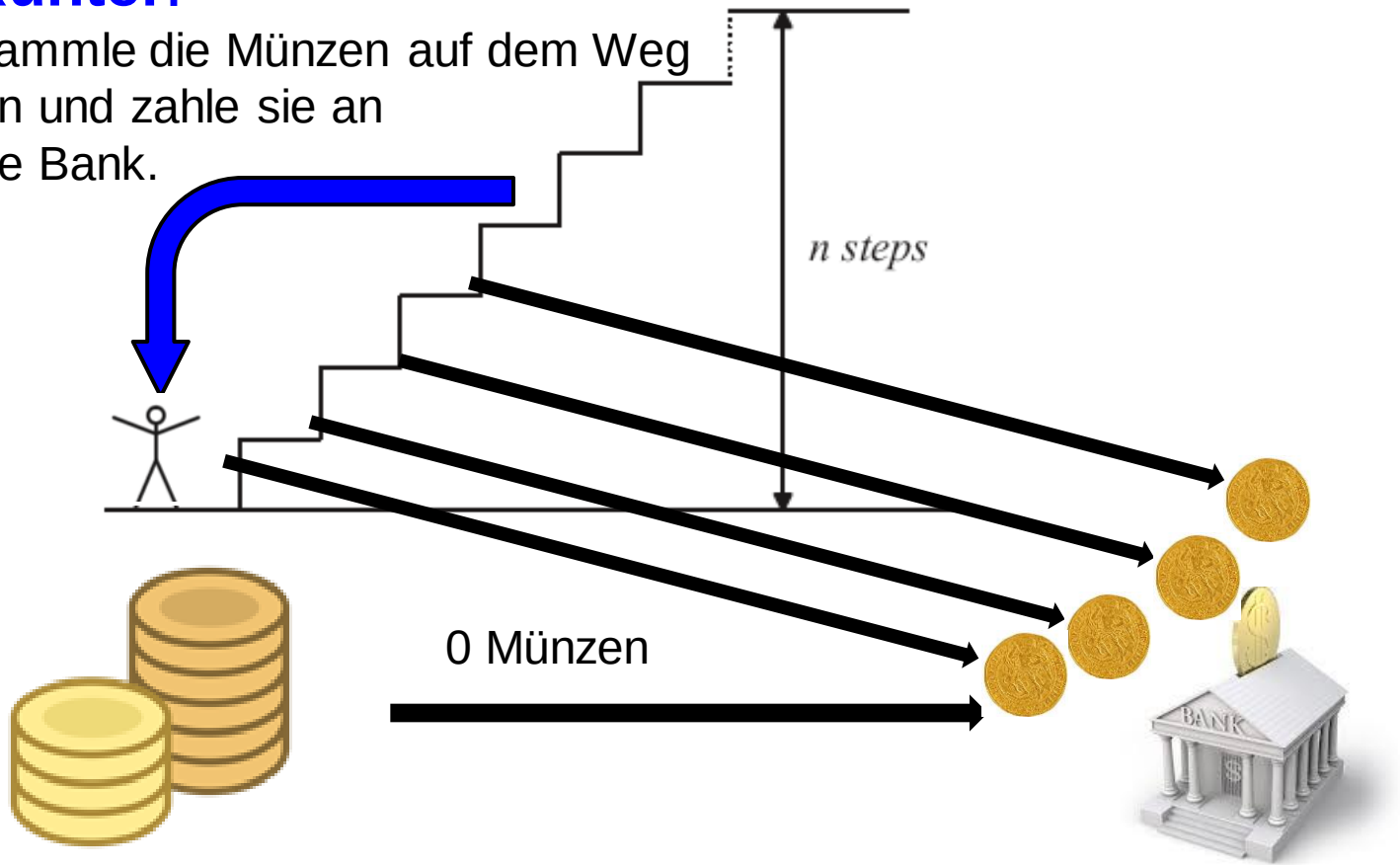
Rauf:

Zahle eine Münze an die Bank.
Lege eine Münze auf die alte Stufe.

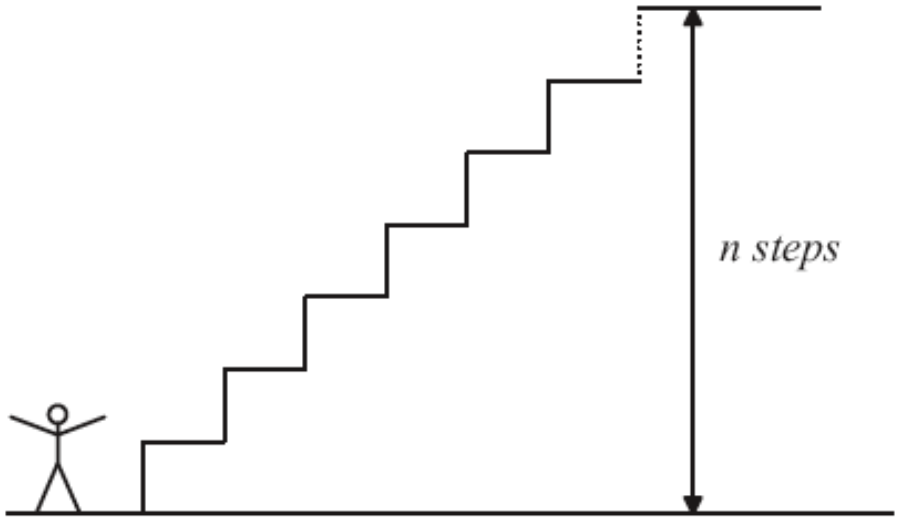


Amortisierte Analyse - Beispiel

Runter:
Sammle die Münzen auf dem Weg ein und zahle sie an die Bank.



Amortisierte Analyse - Beispiel



Jede Runde:
Wir bezahlen für jeden Schritt eine Münze an die Bank.
Wir nehmen höchstens zwei Münzen aus dem Vorrat.

Wir zahlen nur an die Bank, was wir zuvor aus dem Vorrat genommen haben!
Also: Wir machen in k Runden höchstens 2k Schritt!

Amortisierte Analyse - Beispiel

Mathematischer Beweis:

$\phi(i) :=$ Höhe nach i Aktionen

Zahl der Münzen im System

$t(i) :=$ Gesamtlaufzeit der ersten i Aktionen

$a(i) := \big(\phi(i) + t(i)\big) - \big(\phi(i-1) + t(i-1)\big)$

amortisierte Kosten

Es gilt:

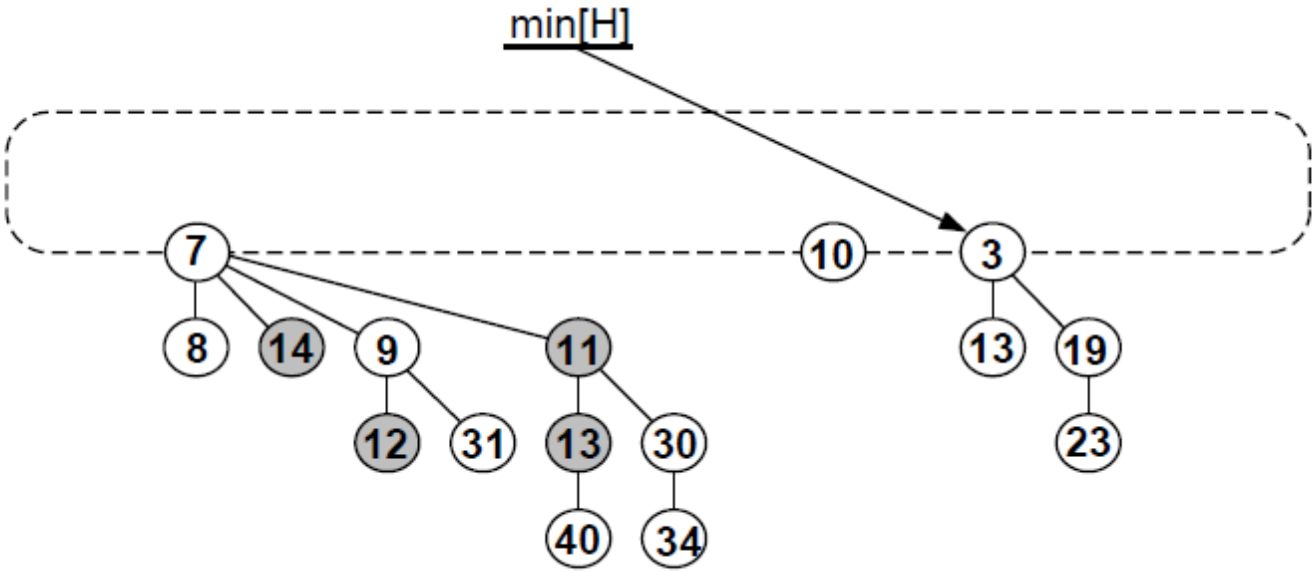
$$a(i) \leq 2 \quad \text{(Fallunterscheidung)}$$

$$\Longrightarrow \quad \phi(k) + t(k) = \sum_{i=1}^{k} a(i) \leq 2k \quad \text{(Induktion)}$$

$$\Longrightarrow \quad t(k) \leq 2k$$

Fibonacci Heaps

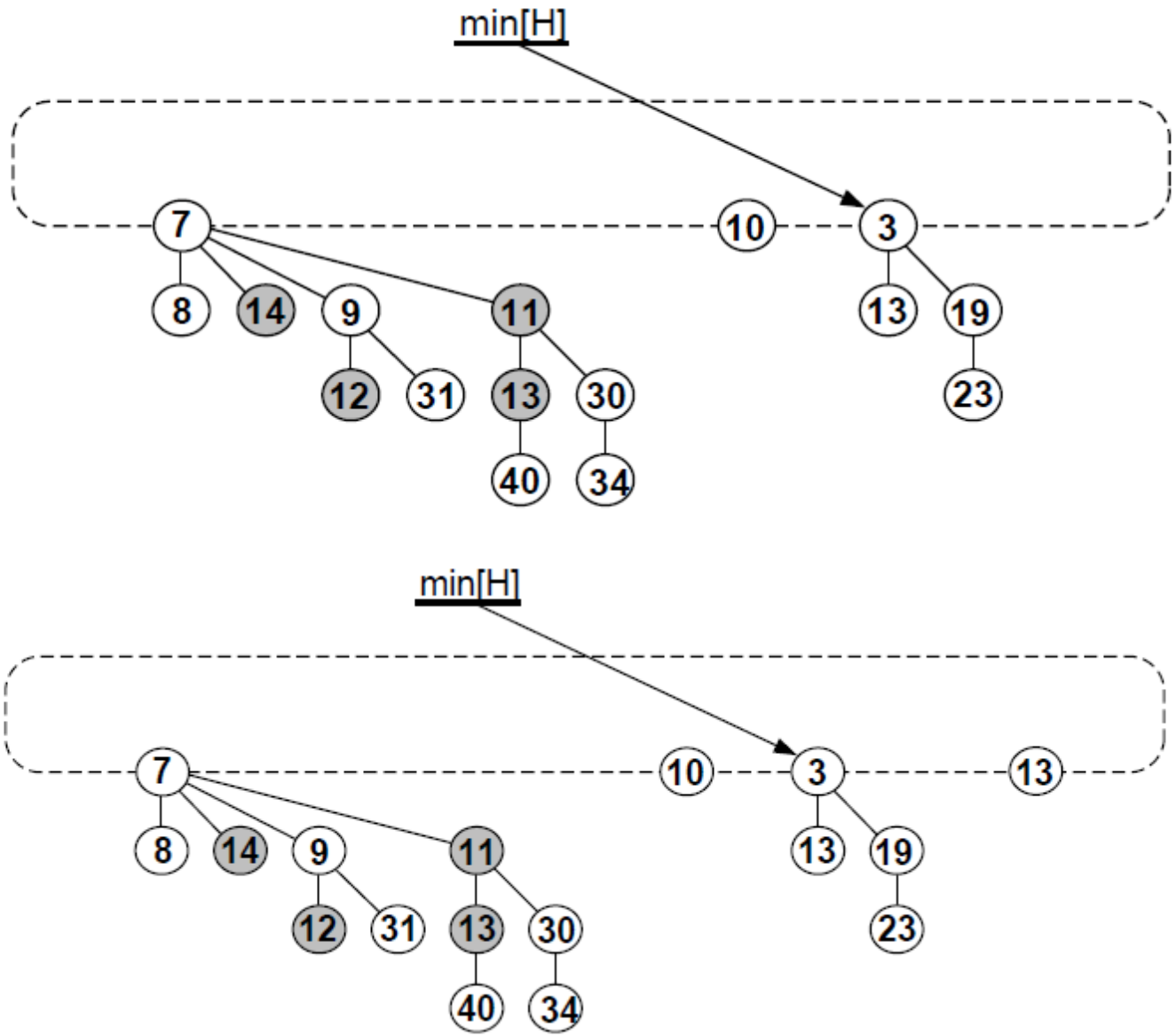
Fibonacci-Heap



Fibonacci-Heap:

- Kollektion von gewurzelten Bäumen
- Wurzeln in einer verketteten Liste (*Wurzelliste*)
- *Heap-Bedingung*: key(parent) <= key(child)
- Zeiger auf die minimale Wurzel (globales Minimum)
- Rang(v) := Anzahl Kinder von v
- manche Knoten sind *markiert* (dazu später)

Insert

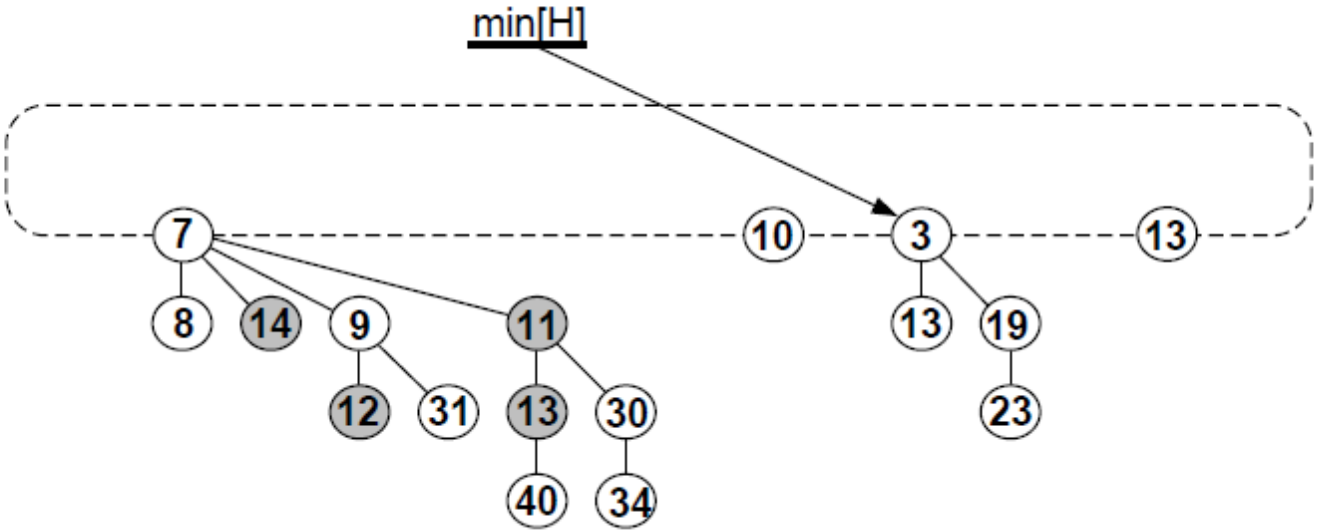

min[H]
7
10
3
8
14
9
11
13
19
12
31
13
30
23
40
34
min[H]
7
10
3
13
8
14
9
11
13
19
12
31
13
30
23
40
34

Insert

Insert:

- Knoten wird als einelementiger Baum in die Wurzelliste eingefügt.
- Zeiger min[H] wird falls notwendig auf diesen Knoten umgesetzt.

ExtractMin



ExtractMin

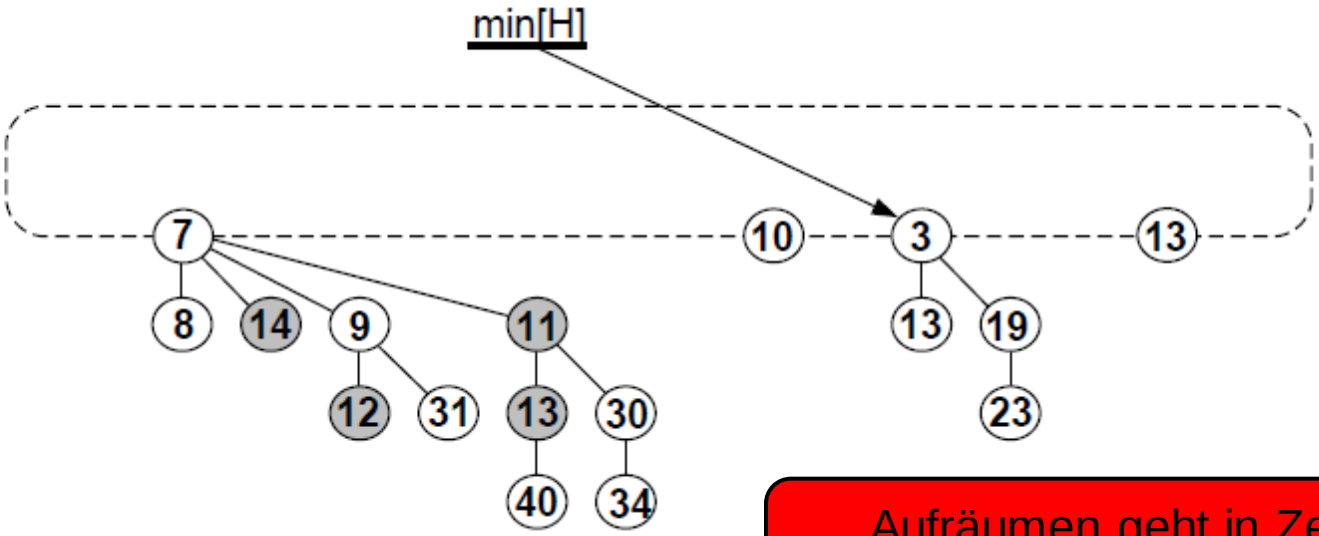
ExtractMin:

- Wurzel desjenigen Baumes, auf den min[H] zeigt, wird entfernt.
- Alle Kinder dieses Knotens werden in die Wurzelliste eingefügt.
- Datenstruktur wird aufgeräumt:

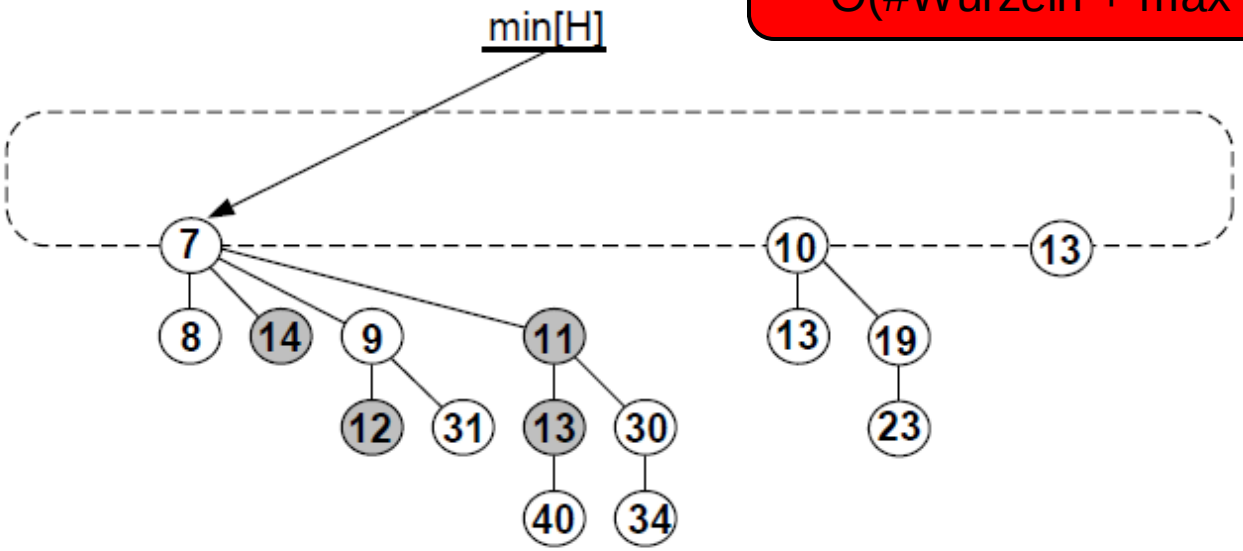
  → Wurzelliste soll keine zwei Knoten mit dem gleichen Rang (= Anzahl Kinder) enthalten

  (Wenn doch, dann mache die Wurzel mit grösserem Schlüssel zum Kind der anderen Wurzel)

ExtractMin



Aufräumen geht in Zeit
O(#Wurzeln + max-Rang)



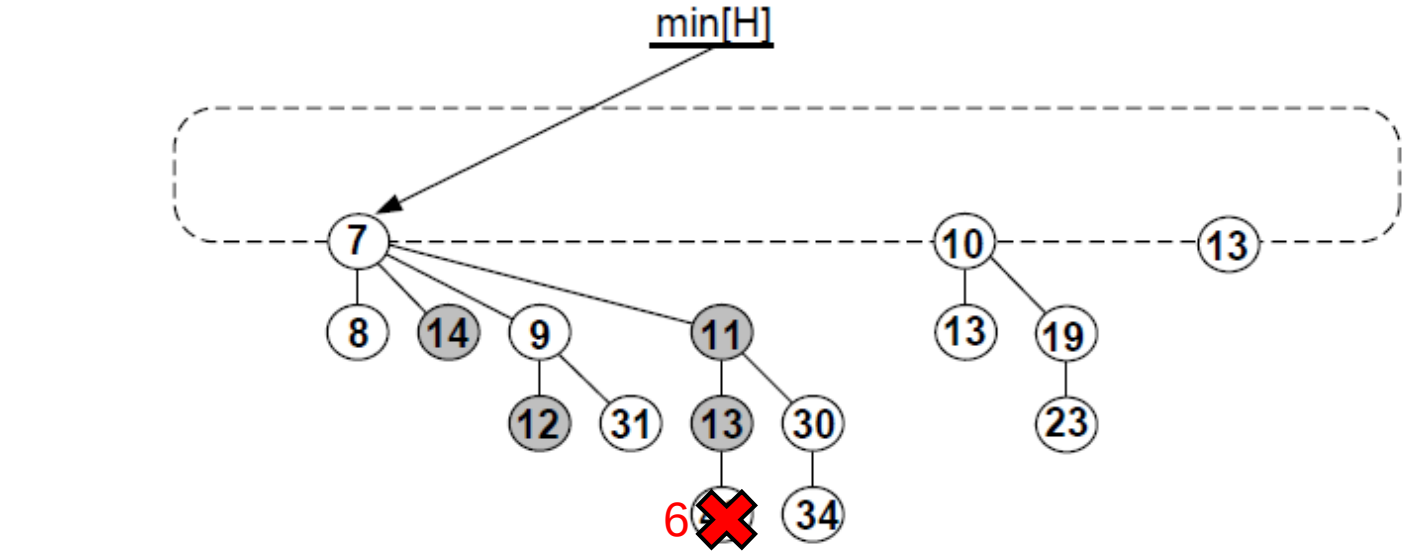
DecreaseKey

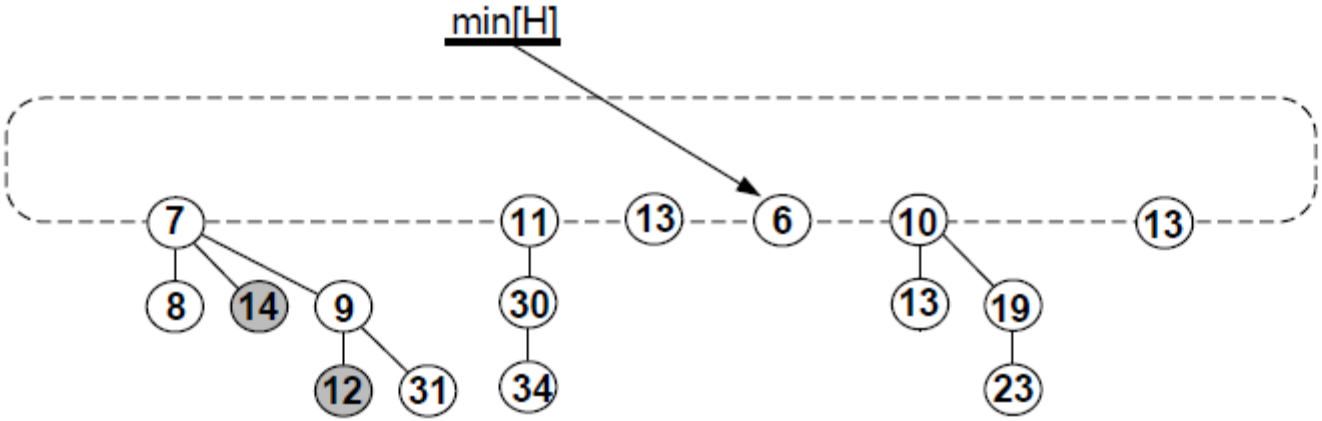
DecreaseKey:

- Hänge den Knoten von seinem Eltern-Knoten ab und füge ihn mit dem neuen Wert in die Wurzelliste ein.
- Betrachte Elternknoten:
  - Falls dieser eine Wurzel ist, tue nichts. Sonst:
  - Falls dieser noch kein Kind verloren hat (marked=false), setze marked auf true.
  - Falls marked=true ist, hänge auch diesen Knoten von seinem Elternknoten ab und füge ihn in die Wurzelliste ein (und setze marked = false), usw.

  → Cascading-Cuts

DecreaseKey





amortisierte Analyse

(1) Für jeden F-Heap H mit n Elementen gilt:

$$rank[x] \leq \log_{1.618}(n) \quad \text{für alle } x \in H$$

(2) Verwende amortisierte Analyse um zu zeigen:

Bezeichnet a(i) die amortisierte Laufzeit der i-ten Operation, so gilt:

$$a(i) = \begin{cases} O(1) & \text{i-te Operation Insert oder DecreaseKey} \\ O(\log n) & \text{i-te Operation ExtractMin} \end{cases}$$