Karlsruher Institut für Technologie

Tutorium Algorithmen 1

07 · Prioritätswarteschlangen · 10.6.2024

Peter Bohner Tutorium 3

Prioritätswarteschlangen

- bei uns: wichtigstes Element $\widehat{=}$ niedrigste Priorität (sog. Min-Heaps)
- richtige Reihenfolge beim Entfernen ist wichtig

deleteMin(): Entfernt wichtigstes Element
- wird i.d.R. aufgerufen, bis Warteschlange leer ist (n mal)

insert(e, p): Fügt neues Element mit Priorität p hinzu
- einzige Methode zu hinzufügen $\Rightarrow$ wird immer n mal aufgerufen

decPrio(e, p): Reduziere Priorität von Element
- p ist neue Priorität
- Vorsicht: Element wird hier **wichtiger**

- Anwendungen
  - Dijkstra-Algorithmus
  - Jarnik-Prim-Algorithmus (machen wir noch)

Peter Bohner - peter.bohner@student.kit.edu

Anwendung

Aufgabenverwaltung: Welches Übungsblatt mache ich wann/überhaupt?

	HM II	LA II	Algo I	SWT I	
Abgabe in	7d	3d	5d	10d	...

Idee: Greedy-Algorithmus

- Wähle immer lokal optimale Lösung
- Gesamtlösung ist meistens gut genug
- Abwägung: schlechter als optimale Lösung, aber billiger zu berechnen

Für uns:

- Wähle eine Aufgabe (nach Kriterium)
- Mache sie, wenn die Zeit ausreicht
- Wiederhole bis keine Aufgabe übrig ist

Priorität $\widehat{=}$ Priorität des Moduls
$\rightsquigarrow$ Fokus auf wichtige Aufgaben

Priorität $\widehat{=}$ Zeit bis zur Abgabe
$\rightsquigarrow$ viele Aufgaben werden erledigt

Naive Implementierung

Karlsruher Institut für Technologie

Liste mit Tupeln von Daten mit Prioritäten

- **min**(): Naive Suche nach kleinstem Element ($O(n)$)
- **deleteMin**(): Führt **min**() aus und entferne den entsprechenden Tupel ($O(n)$)
- **insert**(e, p): Tupel in Liste einfügen ($\Theta(1)$)
- **decPrio**(e, p): ggf. Tupel finden und ändern ($O(n)/O(1)$)

Frage: Laufzeit von n mal insert & delete?

$O(n^2) \Rightarrow$ langsam

Varianten

- Liste ist sortiert. Frage: Welche Laufzeit haben die Operationen dann?

$\Rightarrow$ min,deleteMin $\in \Theta(1)$, insert, decPrio $\in \Theta(n)$

- Einfügen immer in die kleinere Queue
- deleteMin() immer auf die Queue mit dem kleinsten Element

min

PriorityQueue 1 PriorityQueue 2

$\Rightarrow$ Nur um konstante Faktoren schneller

Binary Heaps

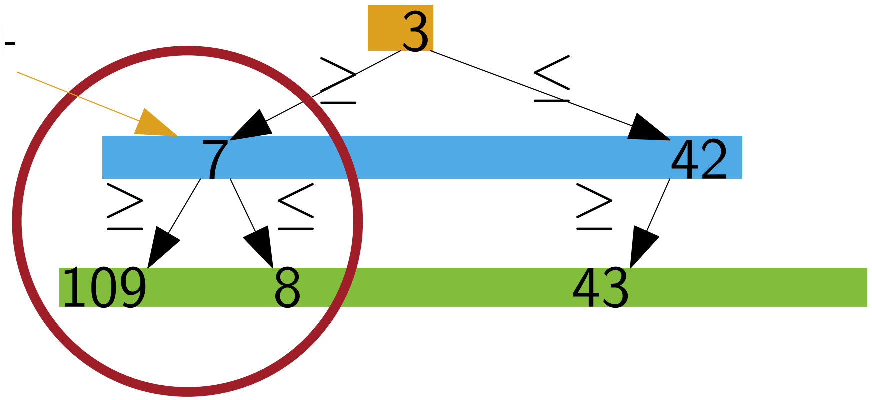
- speichere Elemente in Baum
- Kinder haben immer höhere Priorität (Heap-Eigenschaft)

min(): Wurzel von Baum

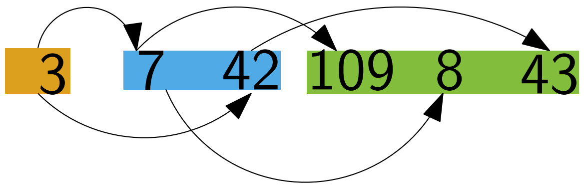
- Jeder Teilbaum ist selber ein binary heap
- Jeder Knoten ist Minimum seines Teilbaums

Zahlen stehen für Element + Priorität (Zahlenwert ist Priorität)

Dieser Knoten ist immer Minimum dieses Teilbaums



Array-Repräsentation:



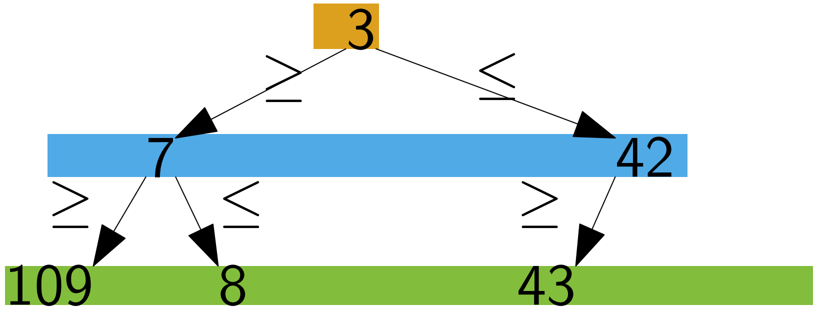
linkes Kind: $2i + 1$
rechtes Kind: $2i + 2$

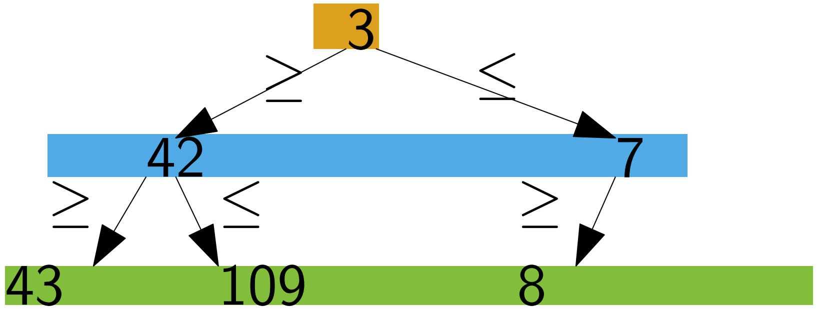
$\Rightarrow$ Baum muss immer von links nach rechts aufgefüllt werden

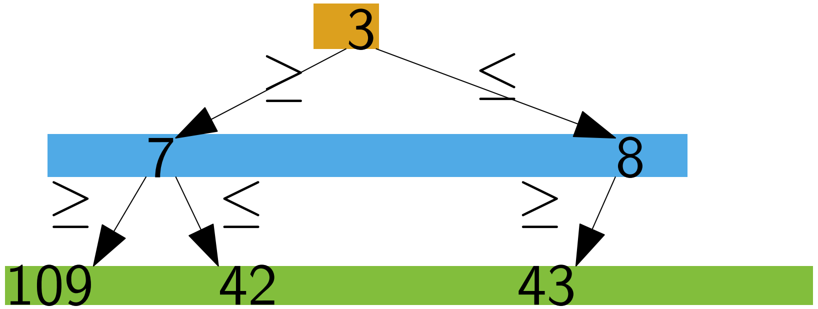
Peter Bohner - peter.bohner@student.kit.edu

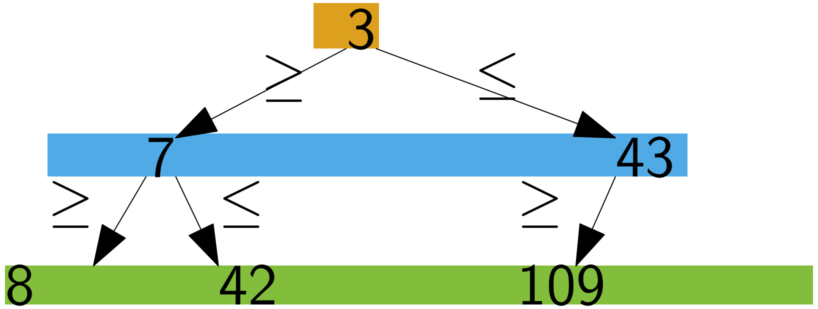
Beispiel

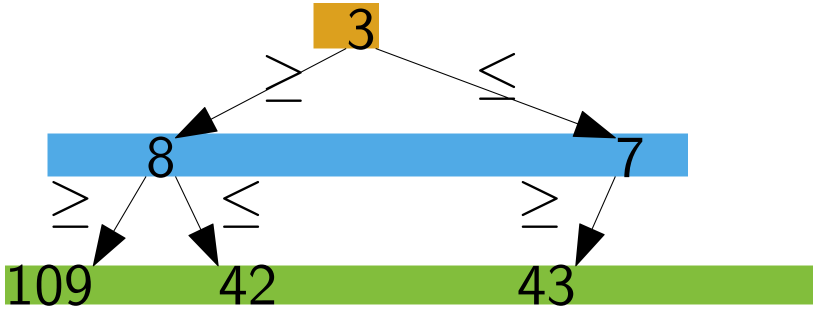
- Alle diese Heaps sind äquivalent
  - gültiger min-heap (erfüllt Heap-Eigenschaft)
  - gleiche Elemente



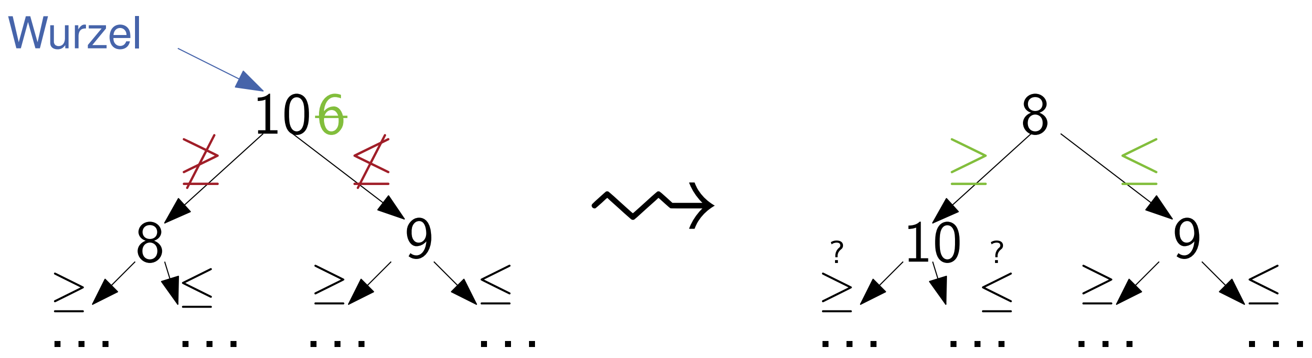








Änderungen der Wurzel



- Verkleinern der Wurzel erhält Heap-Eigenschaft
- Vergrößern kann Heap-Eigenschaft beschädigen
- Reparatur: Wir tauschen Minimum mit Wurzel
  - Minimum ist Wurzel $\Rightarrow$ nichts ändert sich
  - Minimum ist rechts $\Rightarrow$links ändert sich nichts
  - Minimum ist links $\Rightarrow$rechts ändert sich nichts

$\Rightarrow$ geänderter Teilbaum (hier links) ist ggf. noch ungültig

Änderungen der Wurzel II

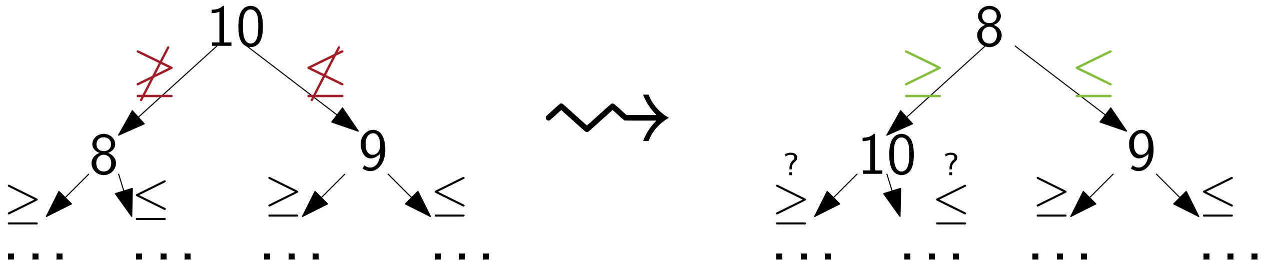
- Knoten $v \in V$ hat

  $v.prio$ Priorität

  $v.value$ Wert

  $v.left$ Linkes Kind (oder $\perp$)

  $v.right$ Rechts Kind (oder $\perp$)



sinkDown(v)

if $v.value > v.left.value \wedge v.left.value < v.right.value$ **then**

  swap$(v, v.left)$ // Tauscht Werte & Prio

  sinkDown$(v.left)$

else if $v.value > v.right.value$ **then**

  swap$(v, v.right)$ // Tauscht Werte & Prio

  sinkDown$(v.right)$

fi

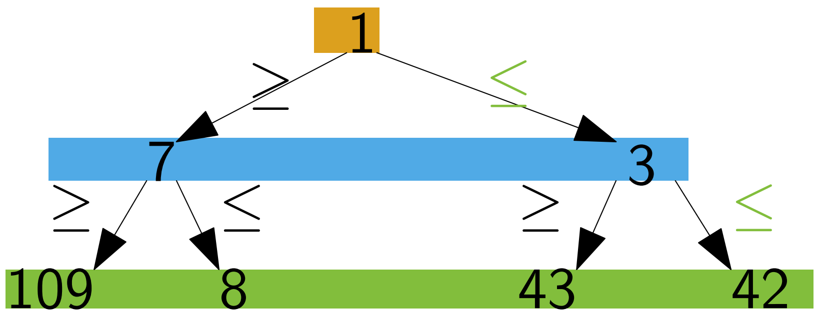
Frage: Welche Laufzeit hast sinkDown?

- worst-case $O(\log n)$ für n Einträge

$\rightsquigarrow$ In der Realität eher Array-Indices statt Knoten und Array-swaps statt **swap**

insert für binary heaps

- Einfügen immer von links nach rechts
- Beispiel: **insert**(1)
- Wir reparieren wieder lokal die Heapeigenschaft
- Unterschied: Wir gehen hier nach oben
- linker Teilbaum erfüllt Heap-Eigenschaft, da sich Wurzel nur reduziert
- rechter Teilbaum erfüllt Heap-Eigenschaft, da neue Wurzel untere Schranke war



bubbleUp(v)

```
if v.parent = ⊥ then return
if v.prio < v.parent.priothen
  swap(v, v.parent) // Tauscht Werte & Prio
  bubbleUp(v.parent)
fi
```

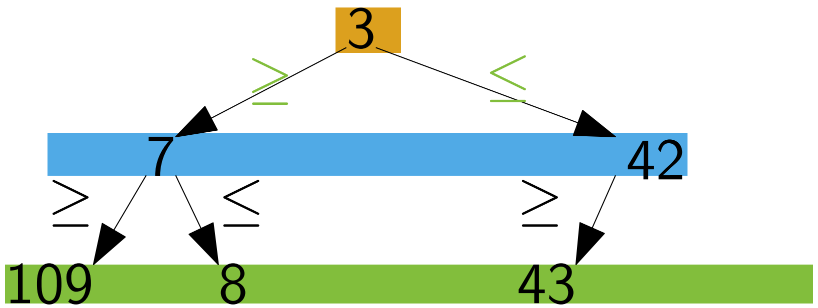
Frage: Was ist die Laufzeit von bubbleUp?

$\Rightarrow$ worst-case $O(\log n)$

Peter Bohner - peter.bohner@student.kit.edu

deleteMin

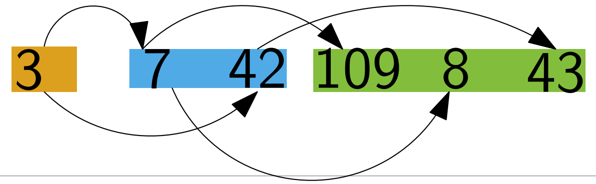
- Idee: Entferne Element rechts unten
- Ersetze Wurzel durch dieses Element
- Lass es runter sinken (sinkDown)



Geben Sie den Zustand folgender Heaps an, nachdem jeweils einmal deleteMin ausgeführt wurde (3min)

$\langle 1, 2, 3, 30, 31, 32 \rangle \quad \rightsquigarrow \langle 2, 30, 3, 32, 31 \rangle$

$\langle 4, 17, 13, 40, 43, 14, 40 \rangle \quad \rightsquigarrow \langle 13, 17, 14, 40, 43, 40 \rangle$



Heaps verschmelzen

Karlsruher Institut für Technologie

Gegeben sind zwei binäre Min-Heaps als Graphen mit den Größen n und m und ein weiteres Element v. Geben Sie an, wie man in $O(\log n + \log m)$ alle drei zu einem Heap zusammenfügen kann. (3min)

Wir erstellen einen Knoten mit Priorität $-\infty$

Die beiden Heaps sind die Kinder dieses Knotens

Dieser Baum ist ein Heap, weil die Heap-Eigenschaft in den Ausgangsheaps und in dem Kindern der Wurzel erfüllt ist.

Wir ersetzen nun die Wurzel durch v

Dies entspricht einem erhöhen der Priorität der Wurzel und dies lässt sich mit einem **sinkDown** reparieren

Das sinkDown geht nur in einen der beiden Heaps, daher ist die Laufzeit in $O(\log n + \log m)$

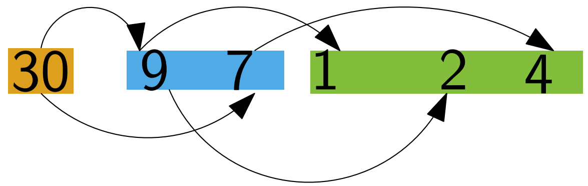
Beachte: Die Lösung muss nicht unbedingt voll besetzt sein

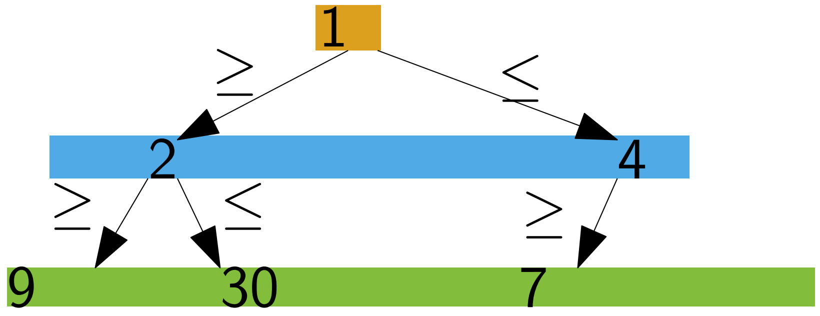
Peter Bohner - peter.bohner@student.kit.edu

buildHeap

Umwandlung: Array $\mapsto$ binary heap

Erinnerung: Array-Repräsentation





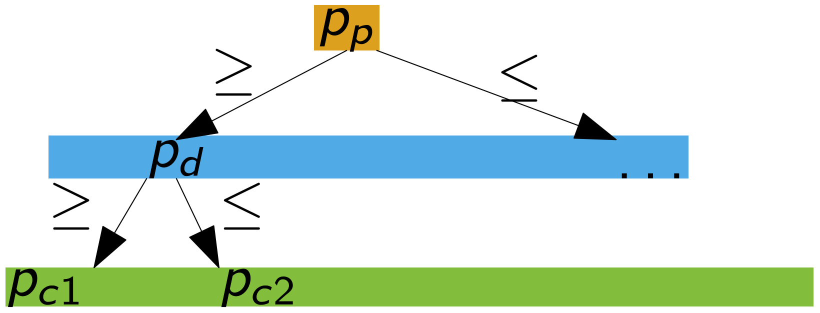
1. „Schritt“: Wir interpretieren das Array als Baum
Wir benutzen Operation aus Aufgabe sytematisch:

- Alle Blätter sind trivialerweise Heaps
- Wir wenden die Operation auf die vorletzte Ebene an
  - Die Knoten sind v und die Kinder sind die Heaps
- Dies wiederholen wir für alle Ebenen von unten nach oben

- **buildHeap** hat eine Laufzeit von $O(n)$ (siehe VL)

decreaseKey

- p_d wird geändert (d wie decrease)
- p_p ist Wurzel (p wie parent)
- c_1, c_2 Kinder 1 u. 2
- p_n ist neuer Wert an Stelle p_d



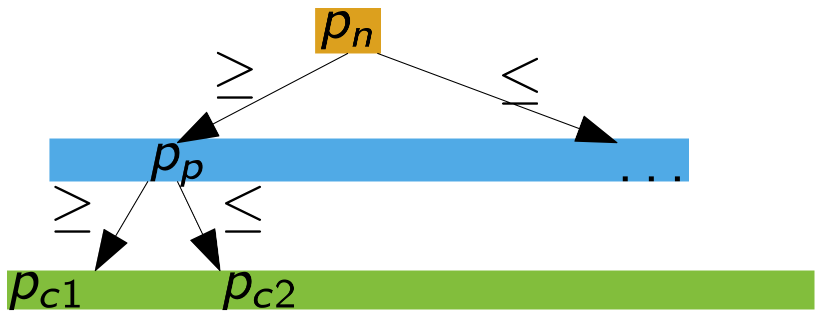
trivial für $p_d > p_n \geq p_p$ (Heap-Eigenschaft bleibt erhalten)

Für $p_p > p_n$ tausche p_r und p_d
$\leq$ ist transitiv $\Rightarrow$ Heapeigenschaft im p_d Teilbaum erhalten

- Wert von Elternknoten hat sich reduziert

$\Rightarrow$ decreaseKey auf Elternknoten ausführen

- Wert von Wurzel kann bedenkenlos reduziert werden(Abbruchbedingung)



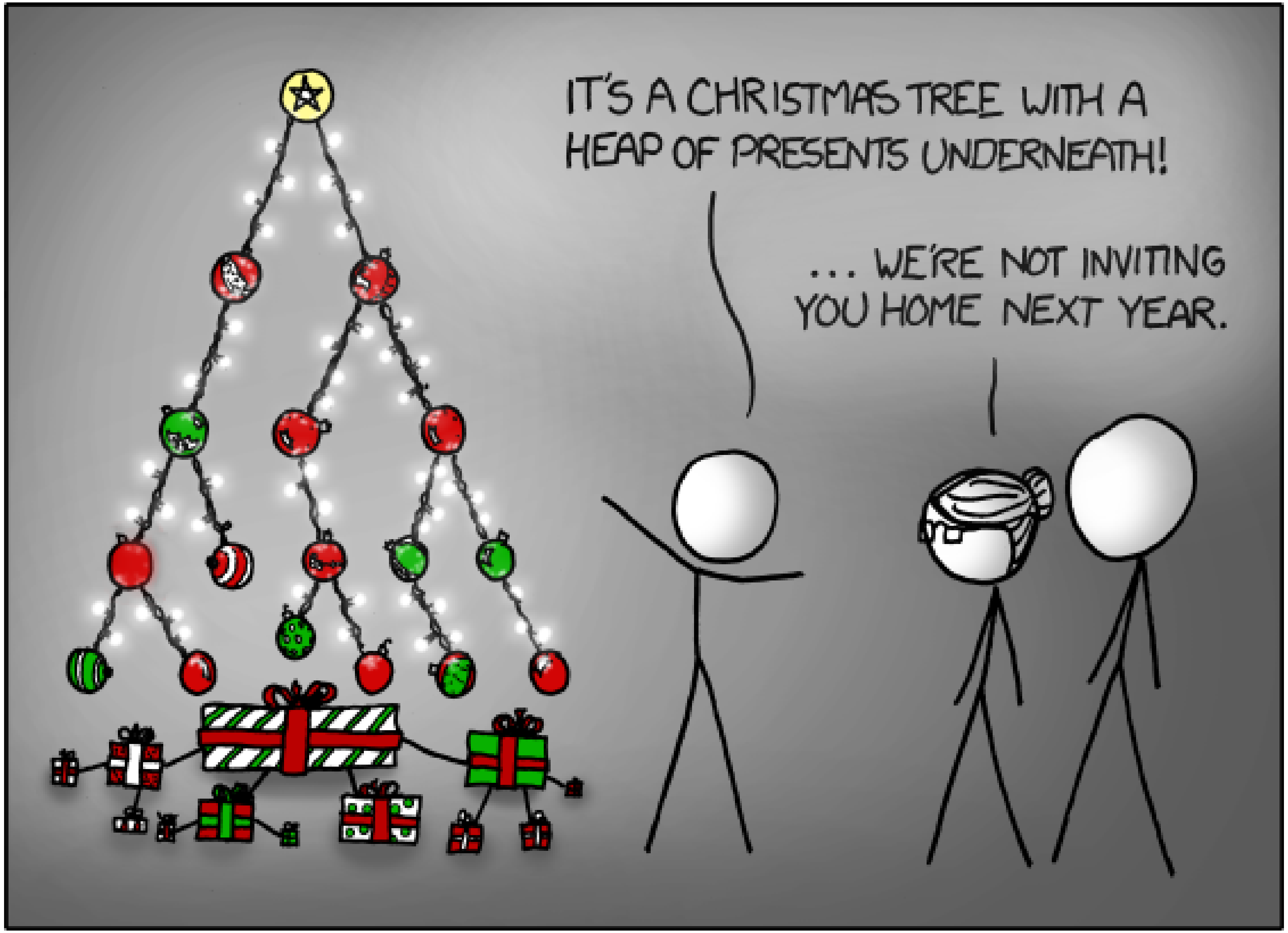
$\Rightarrow$ Prinzip gleich wie Wert ändern und einmal **bubbleUp**

Fragen?

Fragen!

Peter Bohner - peter.bohner@student.kit.edu

Ende



https://xkcd.com/835/

Peter Bohner - peter.bohner@student.kit.edu