

# Algorithmen 1

## Prioritätswarteschlange: binärer Heap



# Erinnerung: Priority-Queue für Dijkstra

## Dijkstras Algorithmus (Berechnung kürzester Wege)

- pro Schritt: finde unexplorierten Knoten mit kleinster aktuell bekannter Distanz
- gewünschte Datenstruktur: Priority-Queue
  - **push**( $v, 7$ ): füge  $v$  mit Priorität 7 ein
  - **popMin**(): extrahiere Element mit kleinster Priorität
  - **decPrio**( $v, 4$ ): verkleinere Priorität von  $v$  auf 4

## Anmerkung

- Priorität wird oft auch **Schlüssel (key)** genannt
- weitere gängige Namen für die Operationen: **insert**, **extractMin**, **decreaseKey**

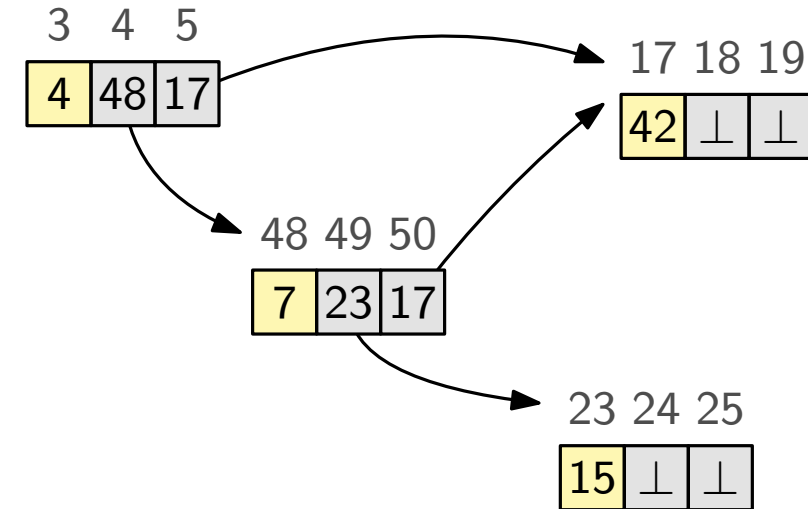
## Ziel für heute

- Datenstruktur entwickeln, die diese Operationen unterstützt
- jede Operation sollte schnell sein ( $O(\log n)$ )
- zunächst: nur **push** und **popMin**

# Erinnerung und Anmerkung: Verzeigerte Strukturen

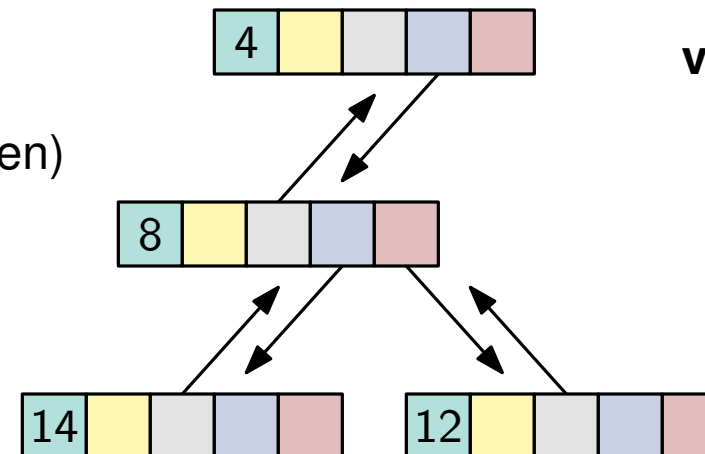
## Verzeigerte Strukturen

- viele kleine Stückchen Speicher (Knoten)
- ein Knoten speichert:
  - Daten, die uns tatsächlich interessieren
  - Speicheradressen anderer Knoten (Zeiger)
- Zugriff durch Navigation entlang Zeiger

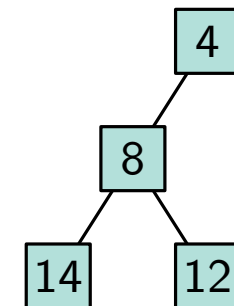


## Heute: Binärer Baum

- jeder Knoten speichert:
  - **Priorität**
  - **andere Daten** (z.B. Knoten eines Graphen)
  - **linkes und rechtes Kind**
  - **Elternknoten**
- vereinfachte Darstellung: zeige nur Prioritäten; alles andere ist implizit



### vereinfachte Darstellung



# Wo liegt die Schwierigkeit?

## Lösung 1: extreme Ordnung

- speichere Elemente als sortierte Folge (nach Priorität)
- super: **popMin** geht in  $O(1)$
- Problem: **push** ist schwierig  $\rightarrow \Theta(n)$   
 (das leidige Problem: in Listen ist Suche langsam, in Arrays ist Einfügen langsam)

## Lösung 2: extreme Unordnung

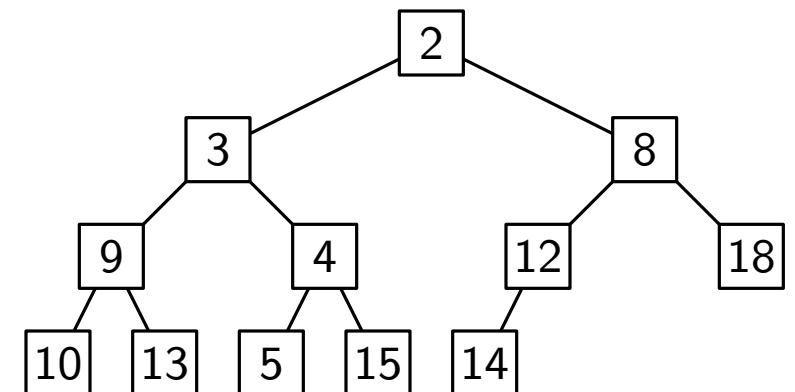
- speichere Elemente als (unsortierte) Folge
- super: **push** geht in  $O(1)$
- Problem: **popMin** ist schwierig  $\rightarrow \Theta(n)$

## Gute Lösung: Binärbaum mit ein bisschen Ordnung

- sortiere jeden Pfad von der Wurzel zu einem Blatt
- genug Ordnung um das Minimum schnell zu finden
- kurze sortierte Pfade  $\rightarrow$  Sortierung aufrechterhalten billig

### Erinnerung:

- **push**( $x$ ): Element mit Priorität  $x$  einfügen
- **popMin**(): Element mit kleinster Priorität extrahieren



# Binärer Min-Heap

## Min-Heap Eigenschaft

- Priorität eines Knotens  $\leq$  Prioritäten der Kinder
- damit gilt: alle Wurzel–Blatt Pfade sind sortiert
- und: Minimum in der Wurzel

## Form des Baumes: vollständiger Binärbaum

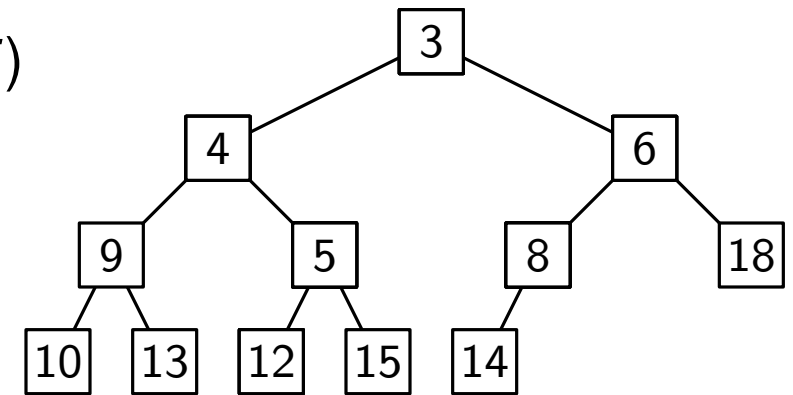
- außer letztes Layer: alle voll besetzt ( $2^i$  Knoten auf Layer  $i$ )
- letztes Layer: von links nach rechts gefüllt
- beachte:  $\Theta(\log n)$  viele Layer

## Todo: Aufrechterhalten dieser beiden Eigenschaften

- **push**( $x$ ):  $x$  erstmal als neues Blatt unten anhängen
- dann:  $x$  auf dem Pfad zur Wurzel nach oben tauschen (bubble-up)
- **popMin**(): Wurzel entfernen und durch letztes Blatt ersetzen
- dann: Wurzel zum kleineren Kind nach unten tauschen (sink-down)

### Erinnerung:

- **push**( $x$ ): Element mit Priorität  $x$  einfügen
- **popMin**(): Element mit kleinster Priorität extrahieren



**Laufzeit:**  $\Theta(\log n)$

# Binärer Min-Heap: Überblick

## Invariante

- Heap-Eigenschaft: Elter  $\leq$  Kind
- Form: vollständiger Binärbaum

## Operation `push(x)`

- füge  $x$  als neues Blatt ein
- tausche  $x$  hoch solange nötig

## Operation `popMin()`

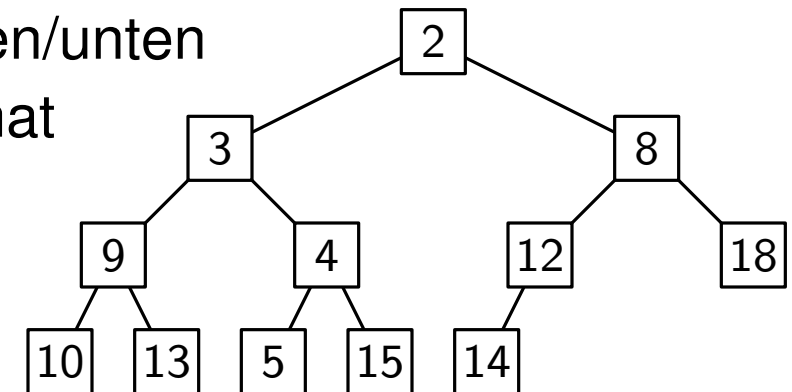
- ersetze Wurzel durch letztes Blatt  $x$
- tausche  $x$  runter solange nötig

## Beachte

- erster Schritt der Operationen verletzt Heap-Eigenschaft nur an einer Stelle
- Fix: tausche den entsprechenden Knoten iterativ nach oben/unten
- nur  $\Theta(\log n)$  Tauschoperationen, da Baum Höhe  $\Theta(\log n)$  hat

## Weitere Operationen: Priorität ändern, Eintrag löschen

- können ähnlich implementiert werden
- in vielen Implementierungen nicht verfügbar
- Trick: Lazy Evaluation (später)



# Umsetzung als Array

## In der Praxis effiziente Repräsentation

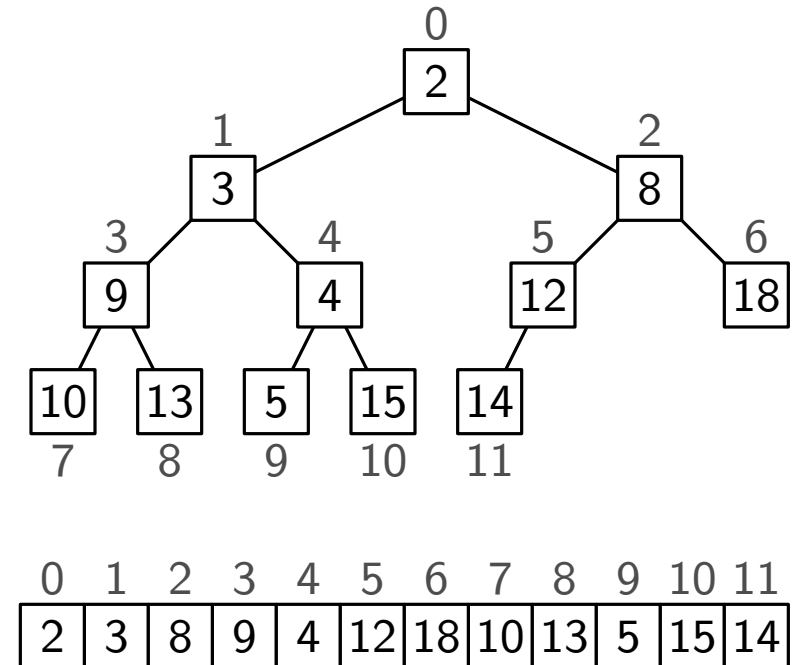
- Arrays oft effizienter als verzeigerte Struktur
- Struktur eigentlich klar: vollständiger Binärbaum
- speichere Knoten hintereinander in einem Array

## Repräsentation eines Knotens

- sei  $v \in [0, n)$  eine ganze Zahl
- $v$  repräsentiert den Knoten an Stelle  $v$  im Array

## Kind- und Elternknoten von $v$

- linkes Kind:  $2v + 1$
- rechtes Kind:  $2v + 2$
- Elternknoten:  $\lfloor \frac{v-1}{2} \rfloor$



## Beweisskizze

- Knoten auf Lage  $i - 1$ :  $[2^{i-1} - 1, 2^i - 1)$
- Knoten auf Lage  $i$ :  $[2^i - 1, 2^{i+1} - 1)$

# Algorithmus → Pseudocode (Grundoperationen)

**class** MINHEAP

Array  $A$  // tree as array

$n()$

| **return**  $A.size()$

**leftChild**(Node  $v$ )

| **return**  $2v + 1$

**rightChild**(Node  $v$ )

| **return**  $2v + 2$

**parent**(Node  $v$ )

| **return**  $\lfloor \frac{v-1}{2} \rfloor$

**swap**(Node  $u$ , Node  $v$ )

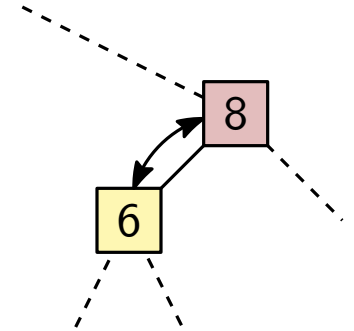
|  $A[u], A[v] := A[v], A[u]$

**bubbleUp**(Node  $v$ )

| **if**  $v \neq 0$  **and**  $A[v] < A[\text{parent}(v)]$  **then**

| | **swap**( $v$ ,  $\text{parent}(v)$ )

| | **bubbleUp**( $\text{parent}(v)$ )



**sinkDown**(Node  $v$ )

|  $u_\ell, u_r := \text{leftChild}(v), \text{rightChild}(v)$

|  $u := v$  // minimum of the three nodes

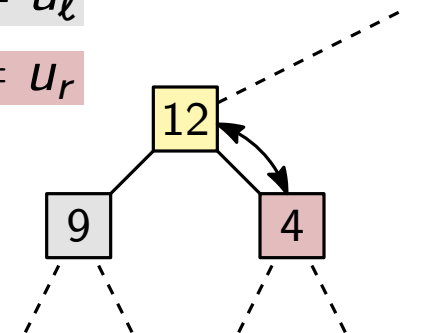
| **if**  $u_\ell < n()$  **and**  $A[u_\ell] < A[u]$  **then**  $u := u_\ell$

| **if**  $u_r < n()$  **and**  $A[u_r] < A[u]$  **then**  $u := u_r$

| **if**  $u \neq v$  **then**

| | **swap**( $u$ ,  $v$ )

| | **sinkDown**( $u$ )





# Algorithmus → Pseudocode

**class** MINHEAP

Array  $A$  // tree as array

$n()$

**leftChild**(Node  $v$ )

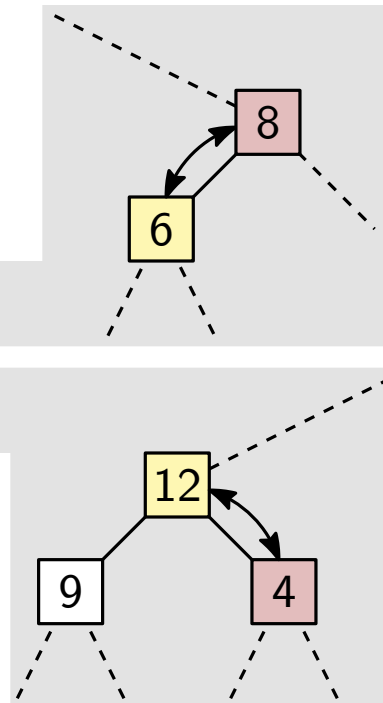
**rightChild**(Node  $v$ )

**parent**(Node  $v$ )

**swap**(Node  $u$ , Node  $v$ )

**bubbleUp**(Node  $v$ )

**sinkDown**(Node  $v$ )



**push**(Priority  $x$ )

$A$ .**pushBack**( $x$ )

**bubbleUp**( $n() - 1$ )

**popMin**()

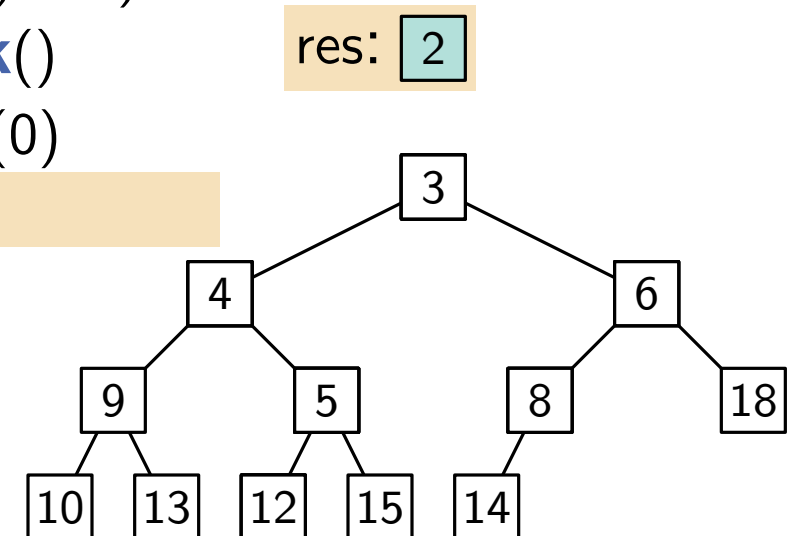
res :=  $A[0]$

**swap**(0,  $n() - 1$ )

$A$ .**popBack**()

**sinkDown**(0)

**return** res



# Anmerkungen

## Prioritäten müssen keine Zahlen sein

- das einzige, was wir mit den Prioritäten machen: vergleichen
- Prioritäten können Elemente einer beliebigen geordneten Menge sein; z.B. Strings
- Laufzeit:  $\Theta(\log n)$  Vergleiche pro Operation

## Heapsort

- Schritt 1: alle  $n$  Elemente mit **push** einfügen  $\Theta(n \log n)$
- Schritt 2:  $n$  mal das minimale Element mit **popMin** entfernen  $\Theta(n \log n)$
- damit: Elemente werden aufsteigend sortiert entnommen

## Untere Schranke

- vergleichsbasiertes Sortieren braucht  $\Omega(n \log n)$  → **push** oder **popMin** braucht  $\Omega(\log n)$
- schlagbar, wenn Prioritäten kleine ganze Zahlen sind:  
benutze Buckets, wie bei Bucketsort mit den Prioritäten als Indizes eines Arrays

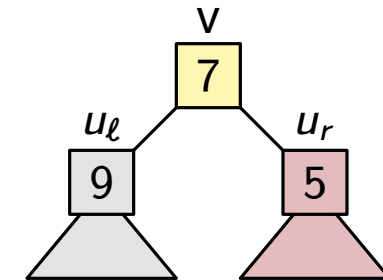
# Initialer Aufbau

## Situation

- gegeben: Folge von Prioritäten (unsortiert)
- Ziel: mache daraus einen Heap
- einfache Lösung: rufe  $n$  mal **push** auf  $\rightarrow$  Laufzeit  $\Theta(n \log n)$
- Geht es besser?

## Beobachtung

- Annahme: Teilbaum unter  $u_\ell$  und  $u_r$  ist bereits Heap
- dann: **sinkDown**( $v$ ) macht Teilbaum unter  $v$  zum Heap
- Algo: rufe bottom-up für jeden Knoten einmal **sinkDown** auf



## Vorüberlegung zur Laufzeit

- Kosten für die Wurzel:  $\Theta(\log n)$
- aber: Knoten weiter unten im Baum können auch nicht so weit sinken
- und: die meisten Knoten liegen weiter unten im Baum

# Initialer Aufbau: Umsetzung im Detail

```
class MINHEAP
```

```
  Array A      // tree as array
```

```
  n()
```

```
  sinkDown(Node v)
```

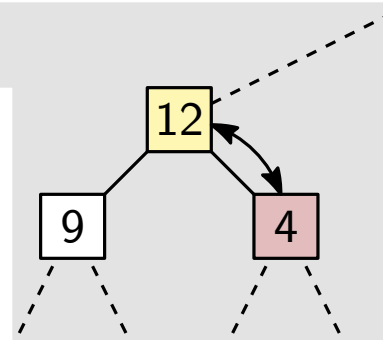
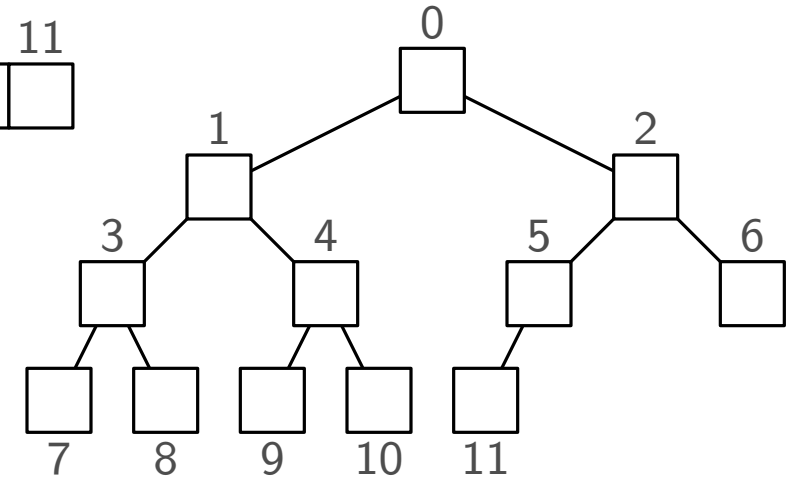
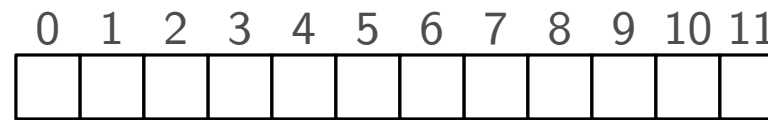
```
  buildHeap(Array priorities)
```

```
    A := priorities
```

```
    for v = n() - 1, ..., 0 do
```

```
      sinkDown(v)
```

Erinnerung: Array-Repräsentation des Baums



## Anmerkung: Korrektheit

- **sinkDown** wird bottom-up angewendet: tieferes Layer fertig bevor höheres beginnt
- induktiv: beim Aufruf **sinkDown**(v) sind Teilbäume unter den Kindern von v schon Heaps
- nach **sinkDown**(v) ist Teilbaum unter v ein Heap

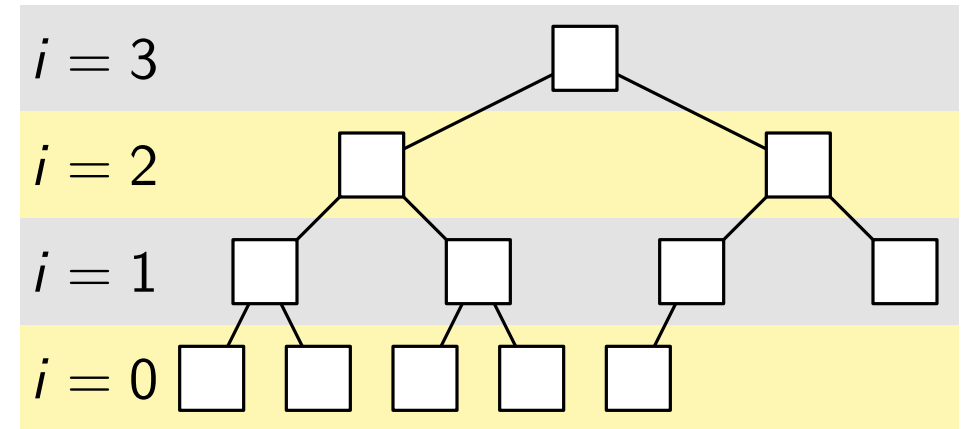
# Initialer Aufbau: Laufzeit

## Kosten für den Aufruf `sinkDown(v)`

- sei  $i$  die Distanz von  $v$  zu einem Blatt
- $v$  wandert höchstens  $i$  Schritte nach unten
- Laufzeit:  $\Theta(i + 1)$

Anzahl Knoten für festes  $i$ :  $\Theta\left(\frac{n}{2^i}\right)$

Gesamtkosten:  $\Theta\left(\sum \frac{n \cdot i}{2^i}\right) = \Theta(n)$



**Nebenrechnung:**

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \leq 1$$

$$\left. \begin{array}{l} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \leq \frac{1}{2} \\ + \frac{1}{8} + \frac{1}{16} + \dots \leq \frac{1}{4} \\ \vdots \end{array} \right\} = \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2$$

# Priority-Queues in der Wildnis

**Java** <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/PriorityQueue.html>  
**Class PriorityQueue<E>**

The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily.

Implementation note: this implementation provides  $O(\log(n))$  time for the enqueueing and dequeuing methods (`offer`, `poll` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek` and `size`).

## Method Summary

Modifier and Type	Method	Description
boolean	<code>add(E e)</code>	Inserts the specified element into this priority queue.
boolean	<code>offer(E e)</code>	Inserts the specified element into this priority queue.
E	<code>peek()</code>	Returns, but does not remove, the head of this queue.

## Beachte

- unterstützt keine weiteren Operationen wie **decPrio** (für Dijkstra nötig)
- nutze dafür Lazy Evaluation Trick (gleich)

# decPrio mittels Lazy Evaluation

## Grundsätzliche Idee

- markiere zu ändernden Knoten als veraltet
- füge Änderung als neuen Knoten ein
- überspringe veraltete Knoten bei **popMin**

## Tatsächliche Laufzeit

- **decPrio** hat dann die selbe Laufzeit wie **push**
- **popMin** ist aber manchmal teuer: viele veraltete Knoten müssen übersprungen werden
- **decPrio** ist also faul und hinterlässt zusätzliche Arbeit für **popMin**

## Amortisierte Laufzeit

- charge zusätzliche Kosten bei **popMin** auf verantwortlichen **decPrio**-Aufruf
- **popMin** wird so alle Kosten los außer für ein nicht-veraltetes Element  $\rightarrow \Theta(\log n)$
- jedem **decPrio**-Aufruf wird nur einmal Kosten zugewiesen  $\rightarrow \Theta(\log n)$

# Lazy Evaluation für Dijkstra

**Dijkstra**(*Graph G, Node s*)

$d :=$  Array of size  $n$  initialized with  $\infty$   
 $d[s] := 0$

PriorityQueue  $Q :=$  empty priority queue

**for** Node  $v$  in  $V$  **do**

$Q.\text{push}(v, d[v])$

**while**  $Q \neq \emptyset$  **do**

$u := Q.\text{popMin}()$

**for** Node  $v$  in  $N(u)$  **do**

**if**  $d[v] > d[u] + \text{len}(u, v)$  **then**

$d[v] := d[u] + \text{len}(u, v)$

$Q.\text{decPrio}(v, d[v])$

**Dijkstra**(*Graph G, Node s*)

$d :=$  Array of size  $n$  initialized with  $\infty$   
 $d[s] := 0$

PriorityQueue  $Q :=$  PQ containing only  $s$

$\text{done} :=$  Array of size  $n$  init with FALSE

**while**  $Q \neq \emptyset$  **do**

$u := Q.\text{popMin}()$

**if**  $\text{done}[u]$  **then continue**

**for** Node  $v$  in  $N(u)$  **do**

**if**  $d[v] > d[u] + \text{len}(u, v)$  **then**

$d[v] := d[u] + \text{len}(u, v)$

$Q.\text{push}(v, d[v])$

$\text{done}[u] = \text{TRUE}$



# Heaps

## Priority-Queue

- **push**( $x$ ) Element mit Priorität  $x$  einfügen
- **popMin**() Element mit minimaler Priorität extrahieren
- Trick für weitere Operationen (z.B. **decPrio**): Lazy Evaluation
  - markiere zu ändernden Knoten als veraltet → merken wir bei **popMin**

## Binärer Heap

- ausreichend viel Ordnung, um Minimum schnell zu finden
- ausreichend wenig Ordnung, um diese nach einer Änderung wieder herzustellen
- Operationen in  $\Theta(\log n)$  und sehr schnell in der Praxis

## Geht es besser?

- etwas mehr Unordnung → aufräumen wenn nötig
- Binomial-Heap: verbessert **push** auf amortisiert  $\Theta(1)$
- Fibonacci-Heap: verbessert zusätzlich **decPrio** auf amortisiert  $\Theta(1)$