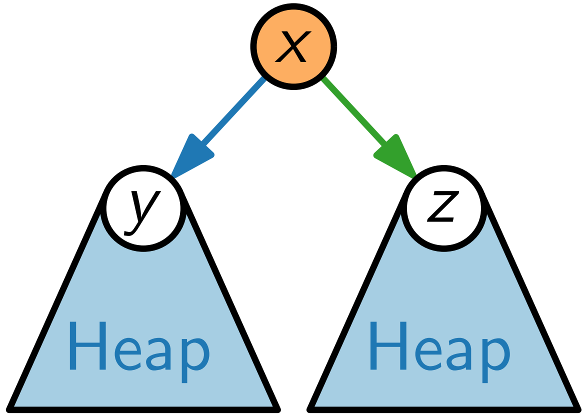
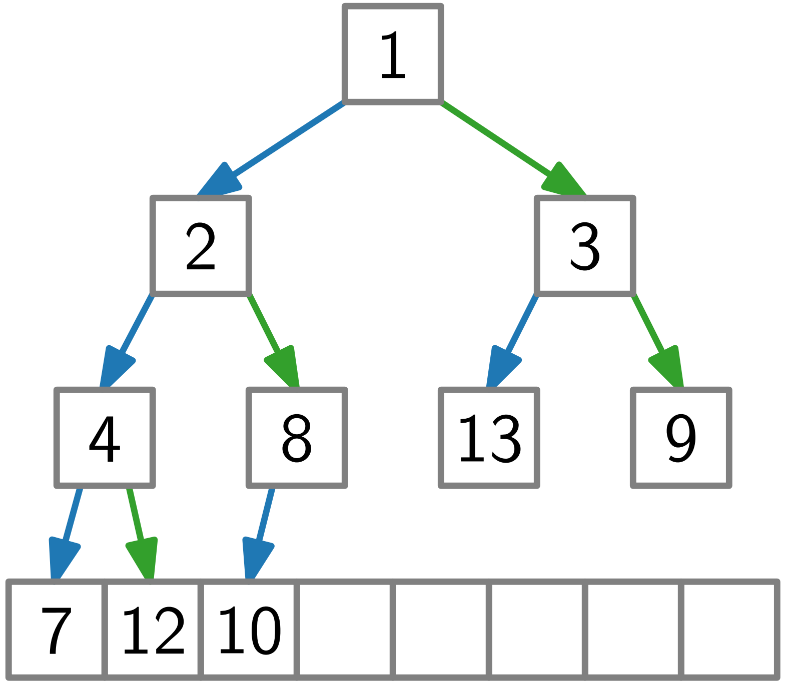
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen



Vorlesung 6: HeapSort



Alexander Wolff

Wintersemester 2025

Bildquelle: `https://sqlrambling.net/category/sql/heaps`

Wir bauen eine Datenstruktur

Datenstruktur.

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp.

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

Implementierung.

wie wird die gewünschte Funktionalität realisiert:

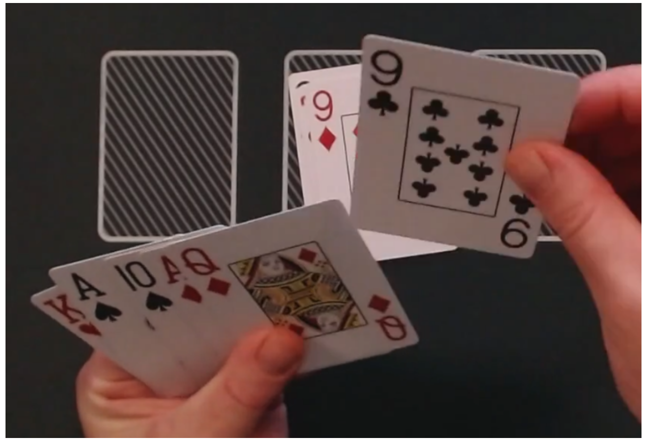
- wie sind die Daten gespeichert (Feld, Liste, ...)?
- welche Algorithmen implementieren die Operationen?

Ein simples Kartenspiel

2 mögliche Züge:

(1) Karte ziehen

(2) **kleinste** Karte spielen



Frage: Welche Laufzeit haben Schritt (1) und (2)?

Antwort: Kommt drauf an! Wie halten wir die Karten in der Hand?

Datenstruktur

Abstrakter Datentyp: Prioritätsschlange

Karten

verwaltet Elemente einer Menge M,

Kartenwert

wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

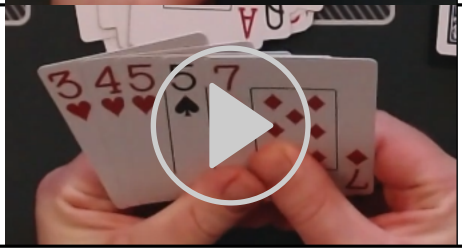
verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	**Funktionalität**
$\textsc{Insert}$(element x)	$M = M \cup \{x\}$
element $\textsc{FindMin}()$	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element $\textsc{ExtractMin}()$	$x = \textsc{FindMin}()$; $M = M \setminus \{x\}$; liefere x

Implementierung

Aufgabe:

- Wie würden Sie die Methoden einer Prioritätsschlange implementieren?
- Welche Laufzeiten liefert Ihre Implementierung im schlechtesten Fall?

Implementierung		Insert	FindMin	ExtractMin
Karten in beliebiger Reihenfolge		$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Karten in beliebiger Reihenfolge, kleinste markiert		$\Theta(1)$	$\Theta(1)$	müssen wieder kleinste Karte finden $\Theta(n)$
Karten in **sortierter** Reihenfolge		$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Prioritätsschlange

Abstrakter Datentyp: **Prioritätsschlange**

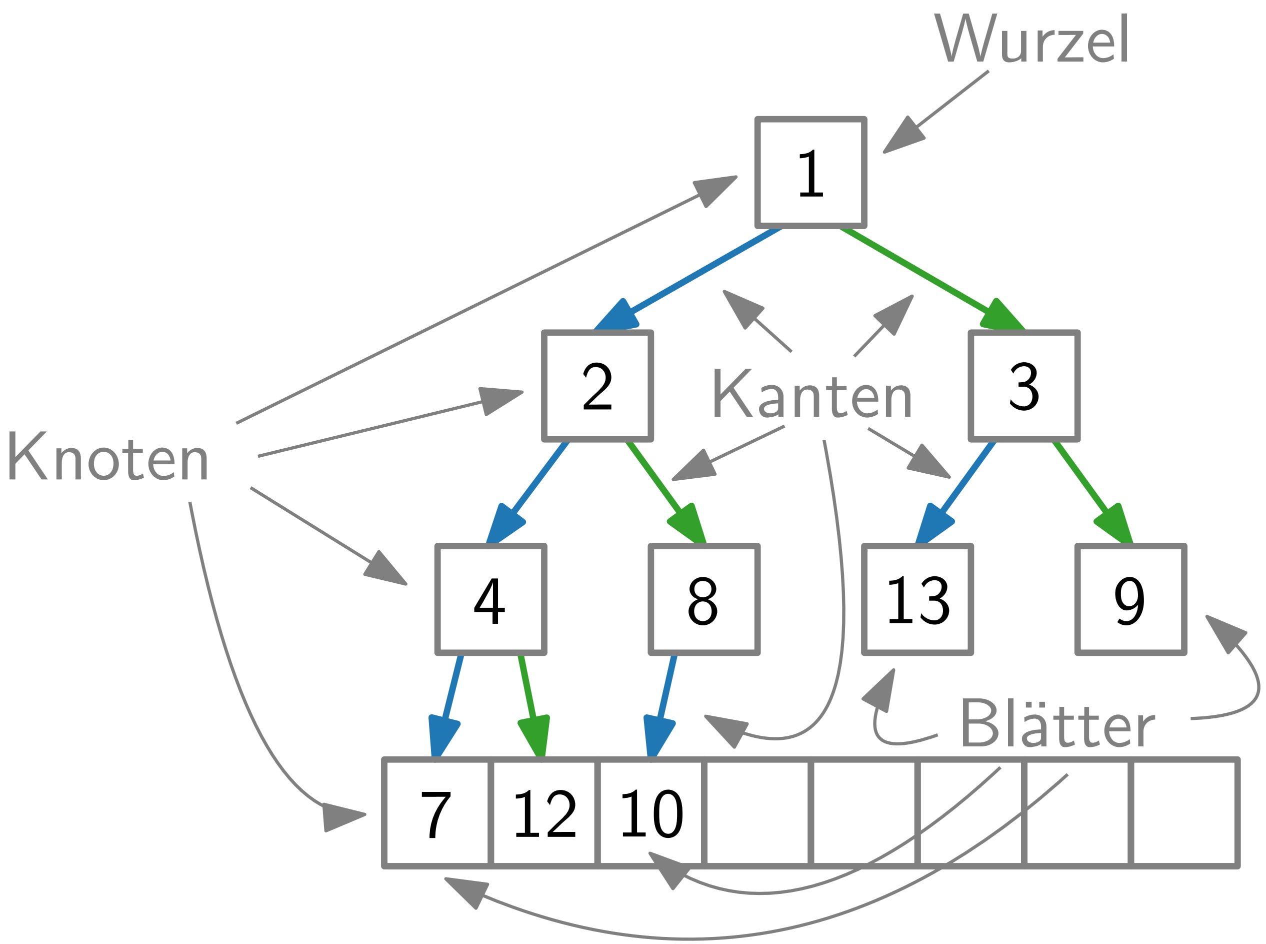
verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Operation	**Funktionalität**
INSERT(element x)	$M = M \cup \{x\}$
element FINDMIN()	liefere $x \in M$ mit: $x.key = \min\{y.key \mid y \in M\}$
element EXTRACTMIN()	$x = \text{FindMin}();\ M = M \setminus \{x\};$ liefere x
DECREASEKEY (element x, priorität p)	$x.key = p$

Implementierung

Implementierung	Insert	FindMin	ExtractMin	DecreaseKey
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in **sortierter** Reihenfolge	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten als MinHeap	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Min-Heaps

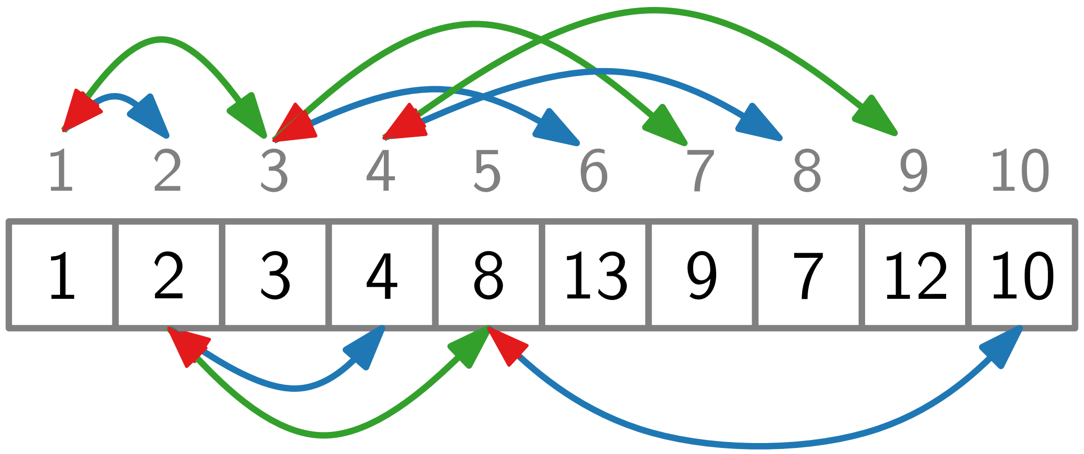


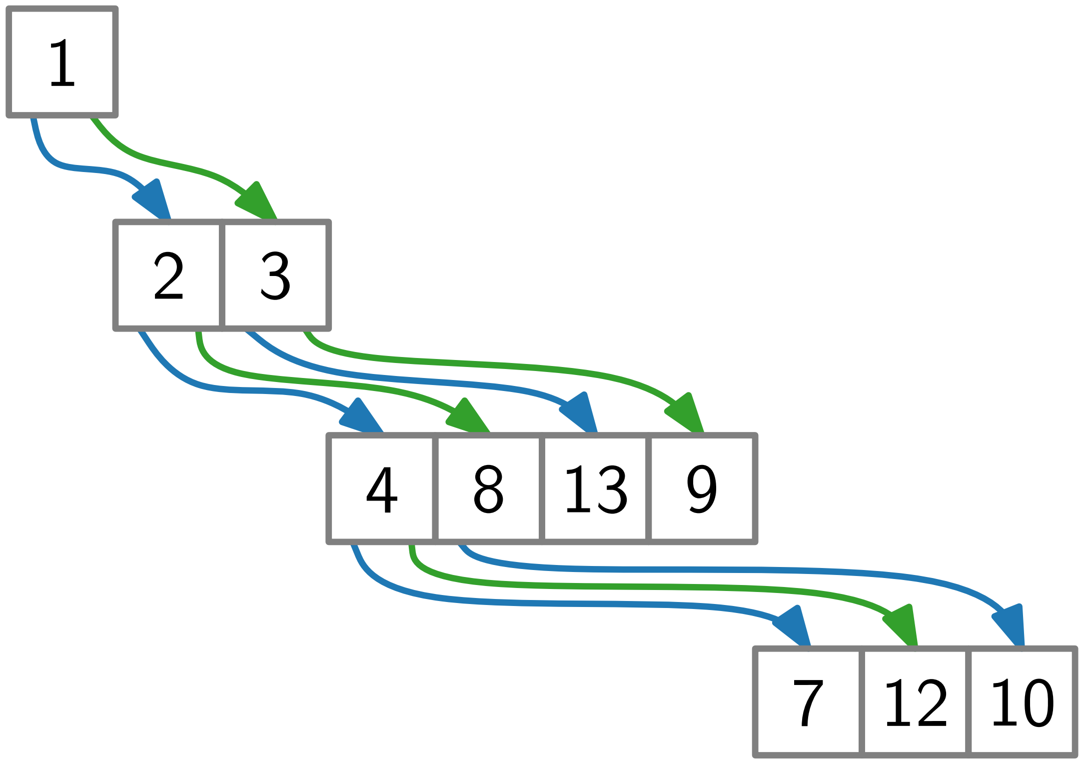
Min-Heaps

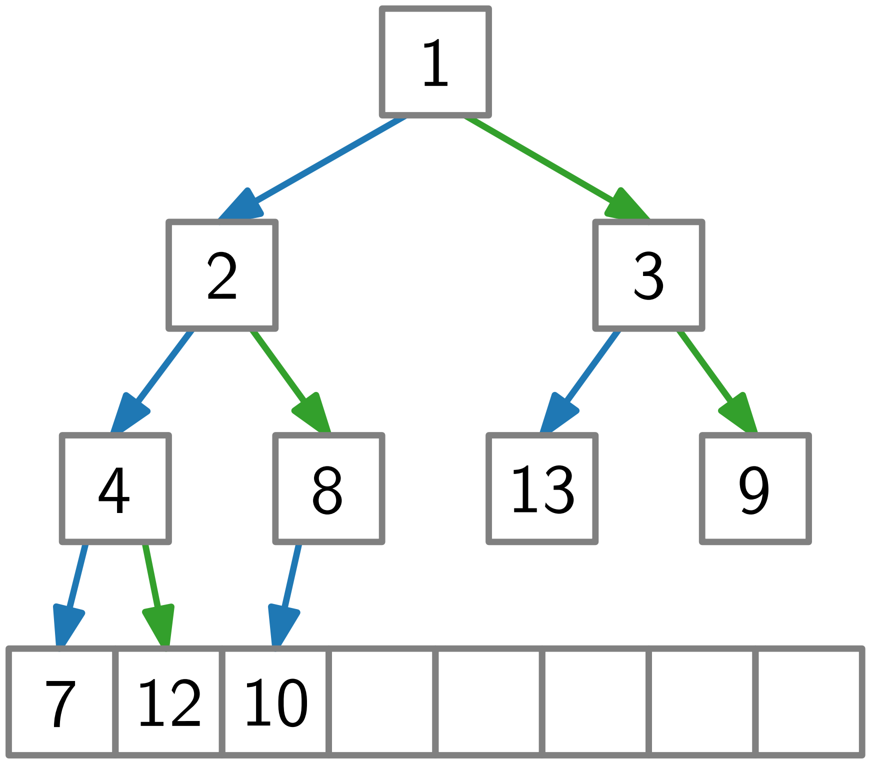
sehr schnelle Rechenoperationen!

Pfeile implementieren:

LEFT(index i)	**return**	$2i$
RIGHT(index i)	**return**	$2i+1$
PARENT(index i)	**return**	$\lfloor i/2 \rfloor$

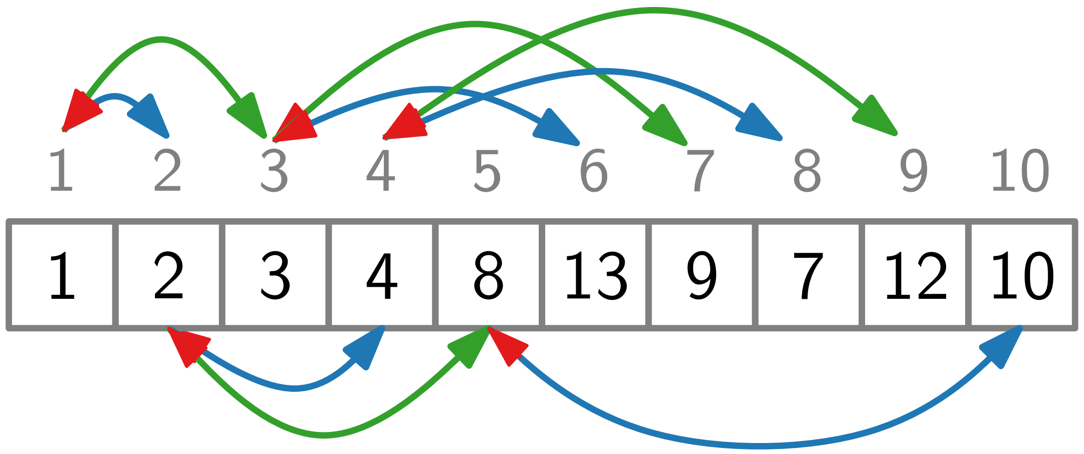






Min-Heaps

sehr schnelle Rechenoperationen!

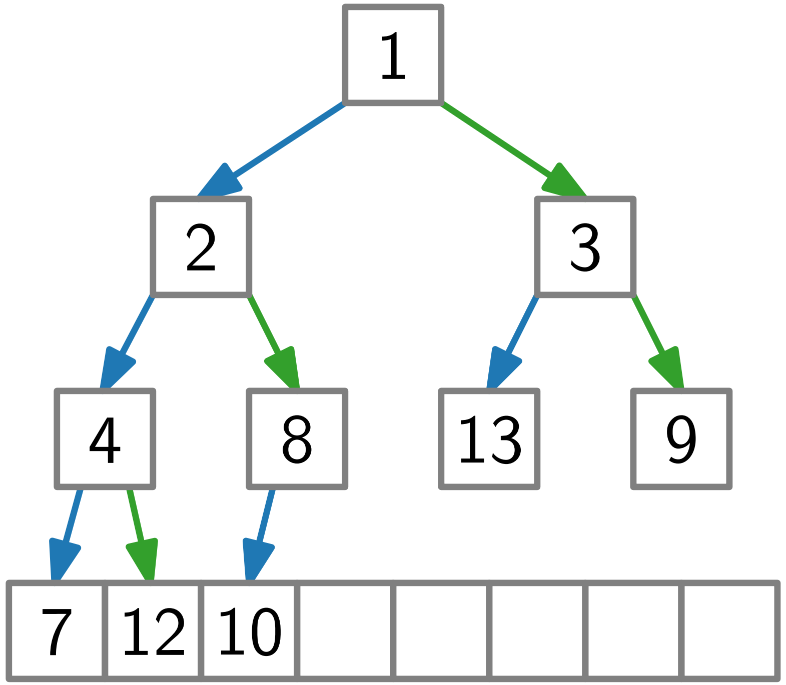


Pfeile implementieren:

LEFT(index i)	**return**	$2i$
RIGHT(index i)	**return**	$2i + 1$
PARENT(index i)	**return**	$\lfloor i/2 \rfloor$

Definition.
Ein **Heap** ist ein Feld, das einem binären Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,
- die letzte Ebene v.l.n.r. gefüllt ist und
- die **Heap-Eigenschaft** gilt.



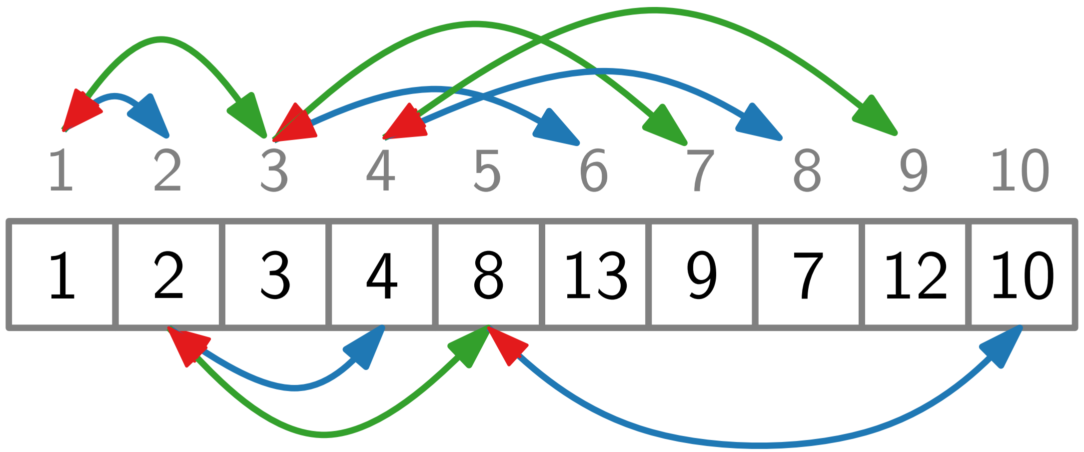
Definition.
Ein Heap hat die **Min-Heap-Eigenschaft**,

wenn für jeden Knoten $i > 1$ gilt: $A[\text{PARENT}(i)] \leq A[i]$.

So ein Heap heißt **Min-Heap**.

Min-Heaps

sehr schnelle Rechenoperationen!

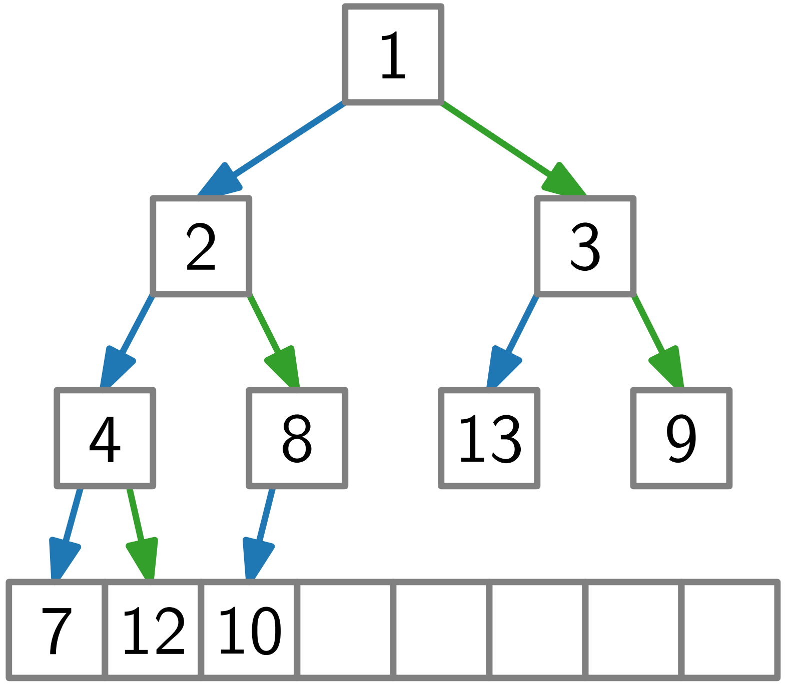


Pfeile implementieren:

LEFT(index i)	**return**	$2i$
RIGHT(index i)	**return**	$2i+1$
PARENT(index i)	**return**	$\lfloor i/2 \rfloor$

Definition.
Ein **Heap** ist ein Feld, das einem $\boxed{\text{binären}}$ Baum entspricht, bei dem

- alle Ebenen außer der letzten voll sind,
- die letzte Ebene v.l.n.r. gefüllt ist und
- die **Heap-Eigenschaft** gilt.



Definition.
Ein Heap hat die **~~Min~~-Heap-Eigenschaft**,
Max
wenn für jeden Knoten $i > 1$ gilt: $A[\text{PARENT}(i)] \cancel{\leq} A[i]$.
$\geq$

So ein Heap heißt **~~Min~~-Heap**.
Max

Baustelle

7	12	10	4	8	13	3	1	9	2

„totales Chaos“

1	2	3	4	7	13	10	12	9	8

Min-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

Nimm MergeSort!

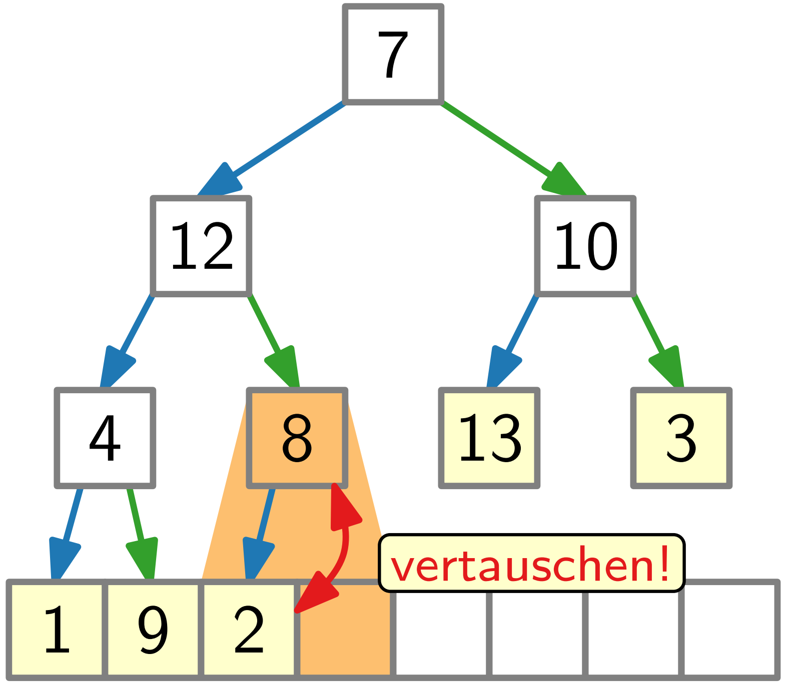
1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite **bottom-up**:
Erst die Blätter...



Baustelle

7	12	10	4	8	13	3	1	9	2

„totales Chaos“

1	2	3	4	7	13	10	12	9	8

Min-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $\mathcal{O}(n \log n)$ Zeit einen Min-Heap!

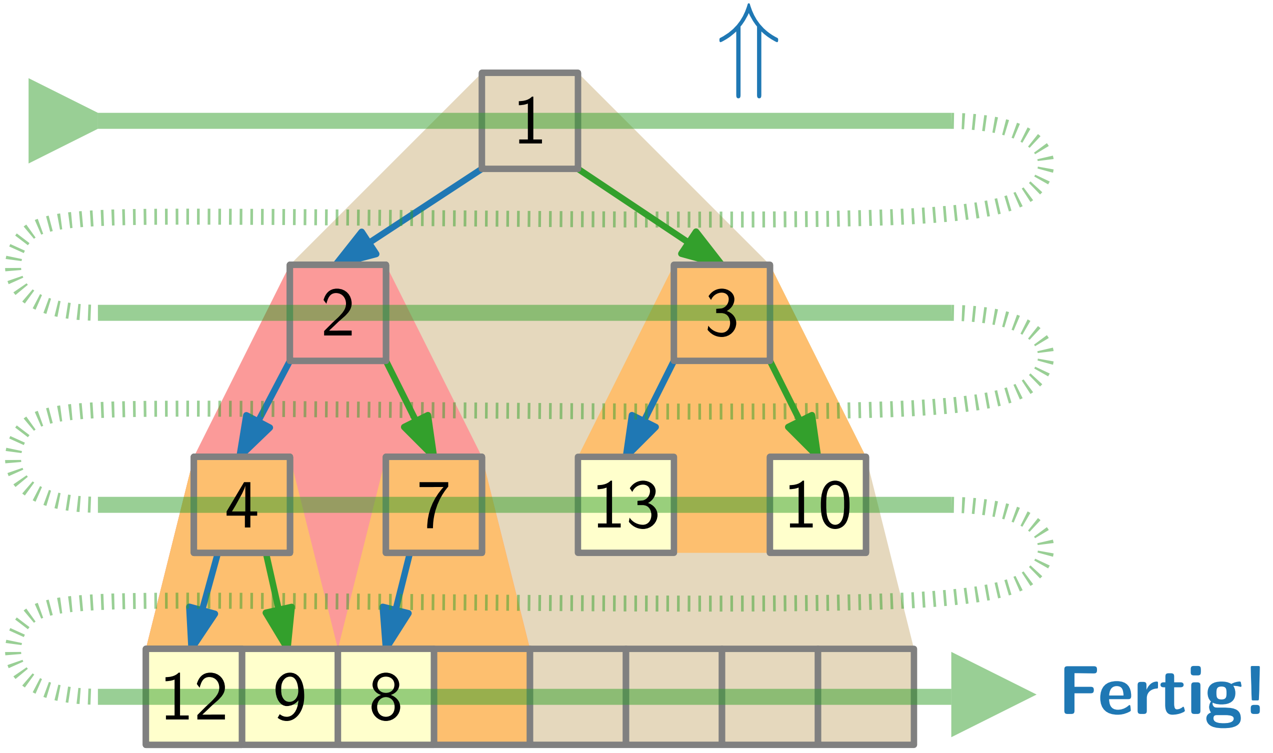
Nimm MergeSort!

1	2	3	4	7	8	9	10	12	13

aufsteigende Sortierung

1	2	3	4	7	13	10	12	9	8

– **Ergebnis** –



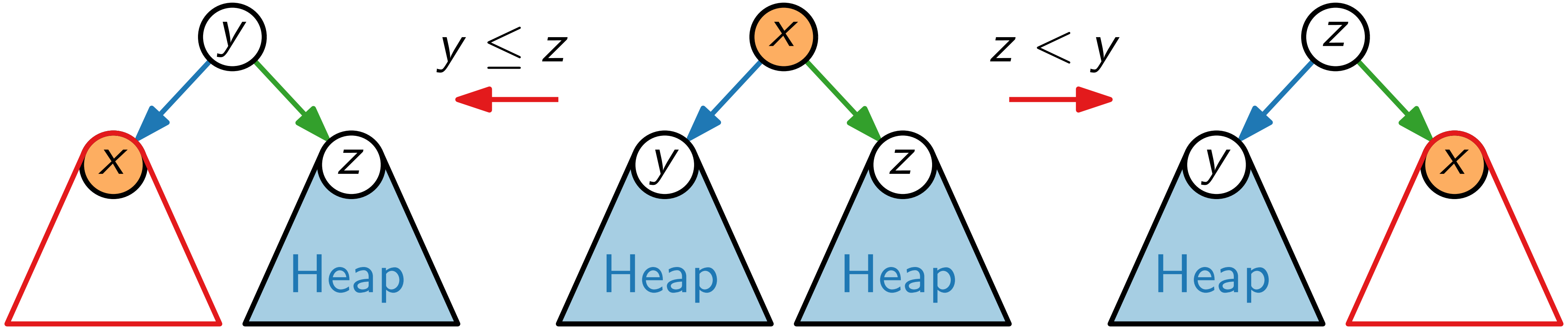
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

Idee: Nutze Baumstruktur!
Arbeite **bottom-up**:
Erst die Blätter...

Elementaroperation

„Versickere“ x, falls x zu groß, d.h. falls $x > \min(y, z)$



MinHeapify(int[] A, index i)
  ℓ = left(i); r = right(i)
  $min = i$
  if $\ell \leq A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
    $min = \ell$
  if $r \leq A.heap\text{-}size$ **and** $A[r] < A[min]$ **then**
    $min = r$
  if $min \neq i$ **then**
    $A[i] \leftrightarrow A[min]$
    MinHeapify(A, min)

Lokale Strategie: top-down

Laufzeit? $T_{\text{MH}}(n, i)$

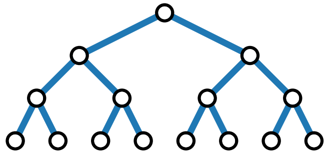
$:=$ Anzahl der Tauschoperationen

$\leq$ Länge des Weges von Knoten i zu einem Blatt

$\leq$ Höhe von i im Heap

$\leq$ Höhe des Heaps

$\leq \lfloor \log_2 n \rfloor$



Das große Ganze

Lokale Strategie: **top-down**

Laufzeit: $T_{\text{MH}}(n, i) \leq$ Höhe von Knoten i im Heap der Größe n

Globale Strategie: **bottom-up**

$\textsc{BuildMinHeap}(\text{int}[]\ A)$
  $A.\mathit{heap\text{-}size} = A.\mathit{length}$
  for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1 **do**
    $\textsc{MinHeapify}(A, i)$

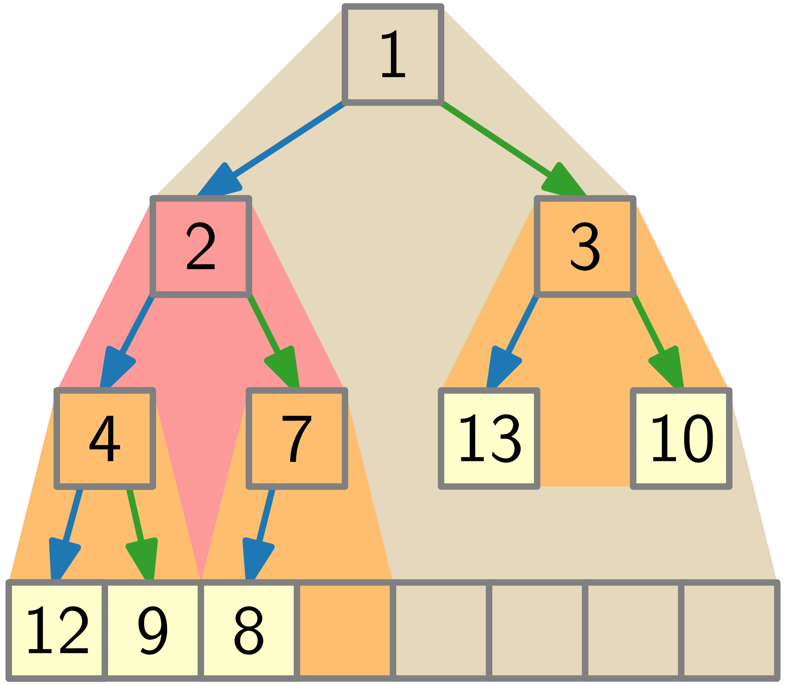
Laufzeit. grob: $\mathcal{O}(n \log n)$

genauer: $T_{\text{BMH}}(n) =$

$$= \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\text{MH}}(n, i)$$

$$\approx \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \frac{n}{16} \cdot 3 + \dots$$

$$= n \cdot \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\frac{1}{2}\right)^{i+1} = ?$$



Fortsetzung Laufzeitanalyse

$$T_{\text{BMH}}(n) \approx n \cdot \sum_{i=1}^{\lfloor \log n \rfloor} i \cdot \left(\frac{1}{2}\right)^{i+1} \leq \frac{n}{4} \sum_{i=1}^{\cancel{\lfloor \log n \rfloor}\ \infty} i \cdot \left(\frac{1}{2}\right)^{i-1}$$

2) $\sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q}$ **geometrische Reihe**

2') $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$

Quotientenregel:

$$\left(\frac{f}{g}\right)' = \frac{gf' - g'f}{g^2}$$

ableiten!

ableiten!

Wir hätten gerne: $\sum_{i=1}^{\infty} i q^{i-1} = \left(\sum_{i=1}^{\infty} q^i\right)' = \left(\frac{1}{1-q}\right)' = \frac{(1-q)\cdot 0 - (-1)\cdot 1}{(1-q)^2} = \frac{1}{(1-q)^2}$

$$\Rightarrow \quad T_{\text{BMH}}(n) \leq \frac{n}{4} \sum_{i=1}^{\infty} i \cdot \left(\frac{1}{2}\right)^{i-1} = \frac{n}{4} \cdot \frac{1}{\left(\frac{1}{2}\right)^2} = n$$

Satz. Ein Heap von n Elementen kann in $\Theta(n)$ Zeit berechnet werden.

Übung Heap-Aufbau

Aufgabe. Bauen Sie einen Heap mit BuildMinHeap!

10	1	9	4	3	7	8	2	6	5

$\longrightarrow$

MinHeapify(int[] A, index i)
  $\ell = \text{LEFT}(i)$; $r = \text{RIGHT}(i)$
  $min = i$
  if $\ell \leq A.heap\text{-}size$ **and** $A[\ell] < A[i]$ **then**
    $min = \ell$
  if $r \leq A.heap\text{-}size$ **and** $A[r] < A[min]$ **then**
    $min = r$
  if $min \neq i$ **then**
    $A[i] \leftrightarrow A[min]$
    MinHeapify(A, min)

BuildMinHeap(int[] A)
  $A.heap\text{-}size = A.length$
  for $i = \lfloor A.length/2 \rfloor$ **downto** 1 **do**
    MinHeapify(A, i)

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M,
wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

$\textsc{FindMin}()$ — $\mathcal{O}(1)$

return $A[1]$

$\textsc{ExtractMin}()$ — $\mathcal{O}(\log n)$

if $A.heap\text{-}size < 1$ **then**
  error „Heap underflow"
$min = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size{-}{-}$
$\textsc{MinHeapify}(A, 1)$
return min

$\textsc{DecreaseKey}(\text{index } i, \text{ prio. } p)$ — $\mathcal{O}(\log n)$

if $p > A[i]$ **then error** „prio. too large"
$A[i] = p$
while $i > 1$ **and** $A[\textsc{parent}(i)] > A[i]$
  $A[i] \leftrightarrow A[\textsc{parent}(i)]$
  $i = \textsc{parent}(i)$

$\textsc{Insert}(\text{priorität } p)$ — $\mathcal{O}(\log n)$

$A.heap\text{-}size{+}{+}$
if $A.heap\text{-}size > A.length$ **then error...**
$A[A.heap\text{-}size] = \infty$
$\textsc{DecreaseKey}(A.heap\text{-}size, p)$

Laufzeiten?

HeapSort

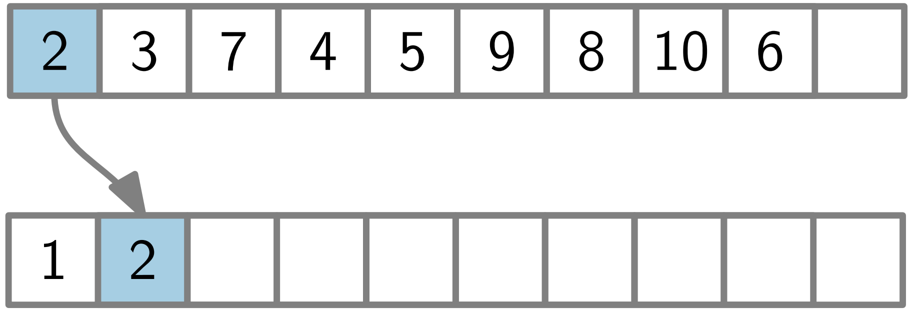
Idee:

- ExtractMin() gibt kleinstes Heap-Element aus.

int[] HeapSort(int[] A)

Schreiben *Sie* den Pseudocode. Verwenden Sie BuildMinHeap und ExtractMin.

Min-Heap



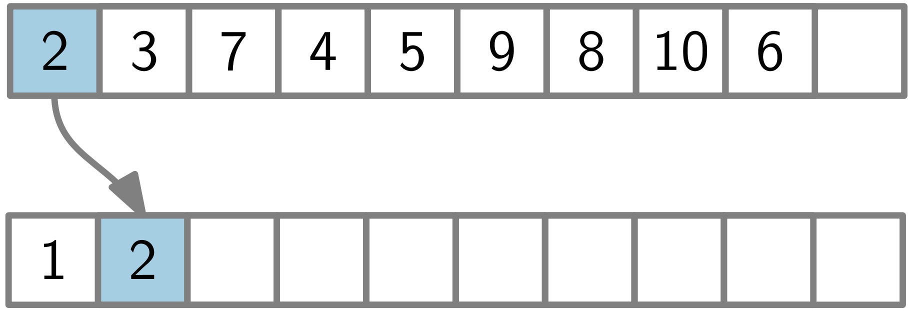
HeapSort

Idee:

- ExtractMin() gibt kleinstes Heap-Element aus.

int[] HeapSort(int[] A)
  BuildMinHeap(A)
  B = **new int**[$A.length$]
  for $i = 1$ **to** $A.length$ **do**
    $B[i]$ = ExtractMin()
  return B

Min-Heap



Laufzeit:

Obere Schranke: $T_{\mathrm{HS}}(n) \in \mathcal{O}(n) + n \cdot \mathcal{O}(\log n) = \mathcal{O}(n \log n)$

Untere Schranke: $c \cdot n + \sum_{i=1}^{n} c' \cdot \log_2 i \geq c' \sum_{i=\frac{n}{2}}^{n} \log_2 \frac{n}{2} \in \Omega(n \log n)$

Satz. HeapSort sortiert n Schlüssel in $\Theta(n \log n)$ Zeit.

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortieralgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✗

Vom Heap zur Sortierung (in-situ)

Idee:
- ExtractMin() gibt kleinstes Heap-Element aus.
- Letztes Feldelement wird dadurch frei

```
HeapSort(int[] A)
  BuildMinHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMin()
  for i = 1 to ⌊A.length/2⌋ do
    A[i] ↔ A[n − i]
```

2	3	7	4	5	9	8	10	6	1

Min-Heap

10	9	8	7	6	5	4	3	2	1

```
HeapSort(int[] A)
  BuildMaxHeap(A)
  for i = A.length downto 2 do
    A[i] = ExtractMax()
```

1	2	3	4	5	6	7	8	9	10

Vergleich Laufzeiten

Ein **in-situ**-Algorithmus benötigt nur $\mathcal{O}(1)$ extra Speicher.

Ein Sortieralgorithmus ist **stabil**, wenn er gleiche Schlüssel in der Ursprungsreihenfolge belässt.

	Bester Fall	Schlechtester Fall	in-situ	stabil
InsertionSort	$\Theta(n)$	$\Theta(n^2)$	✓	✓
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	✓	✓
BubbleSort	$\Theta(n)$	$\Theta(n^2)$	✓	✓
MergeSort	$\Theta(n \log n)$	$\Theta(n \log n)$	✗	✓
HeapSort	$\Theta(n \log n)$	$\Theta(n \log n)$	✓	✗