

Algorithmen und Datenstrukturen

Wintersemester 2020/21

6. Vorlesung

Prioritäten setzen

Heute: 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?

Anwendung: Prozesssteuerung

Anwendung: steuere System durch Verwaltung von unterschiedlich wichtigen Prozessen

Anforderung:

- Prozesse (mit ihrer Priorität) einfügen
- Prozess mit höchster Priorität finden/löschen
- Priorität von Prozessen erhöhen

modelliere



Abstrakter Datentyp: **Prioritätsschlange**

verwaltet Elemente einer Menge M ,
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.

<i>Operation</i>	<i>Funktionalität</i>
<code>Insert</code> (element x)	$M = M \cup \{x\}$
element <code>FindMax</code> ()	liefere $x \in M$ mit $x.key = \max\{y.key \mid y \in M\}$
element <code>ExtractMax</code> ()	$x = \text{FindMax}(); M = M \setminus \{x\};$ liefere x
<code>IncreaseKey</code> (element x , priorität p)	$x.key = p$

Implementation

Aufgabe: Diskutieren Sie mit Ihrer NachbarIn:

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

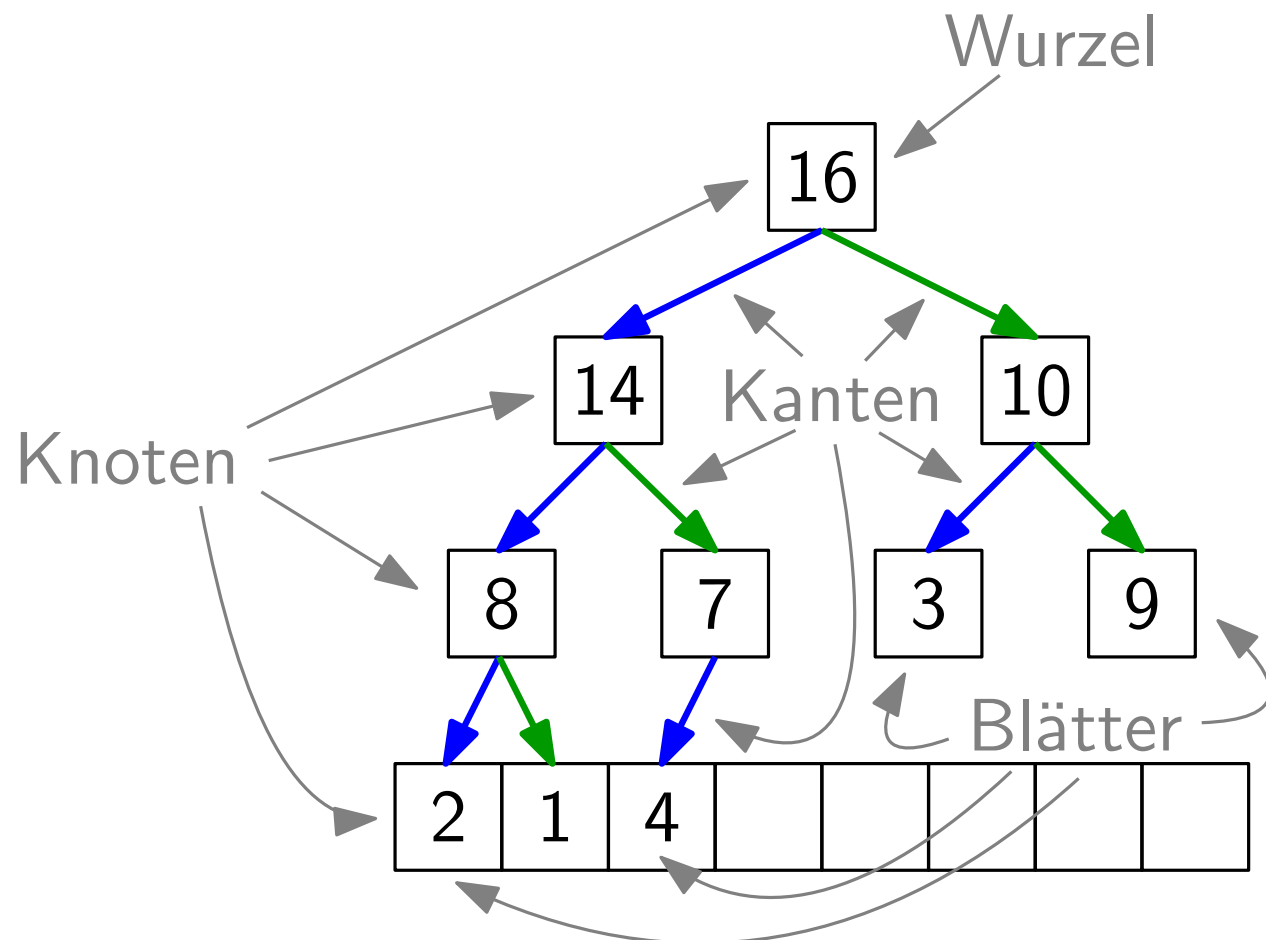
	Insert	FindMax	ExtractMax	IncreaseKey
W-C-Laufzeiten meiner Implement.*	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
heute: Implementierung als Heap (Haufen)	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Das ist exponentiell besser!

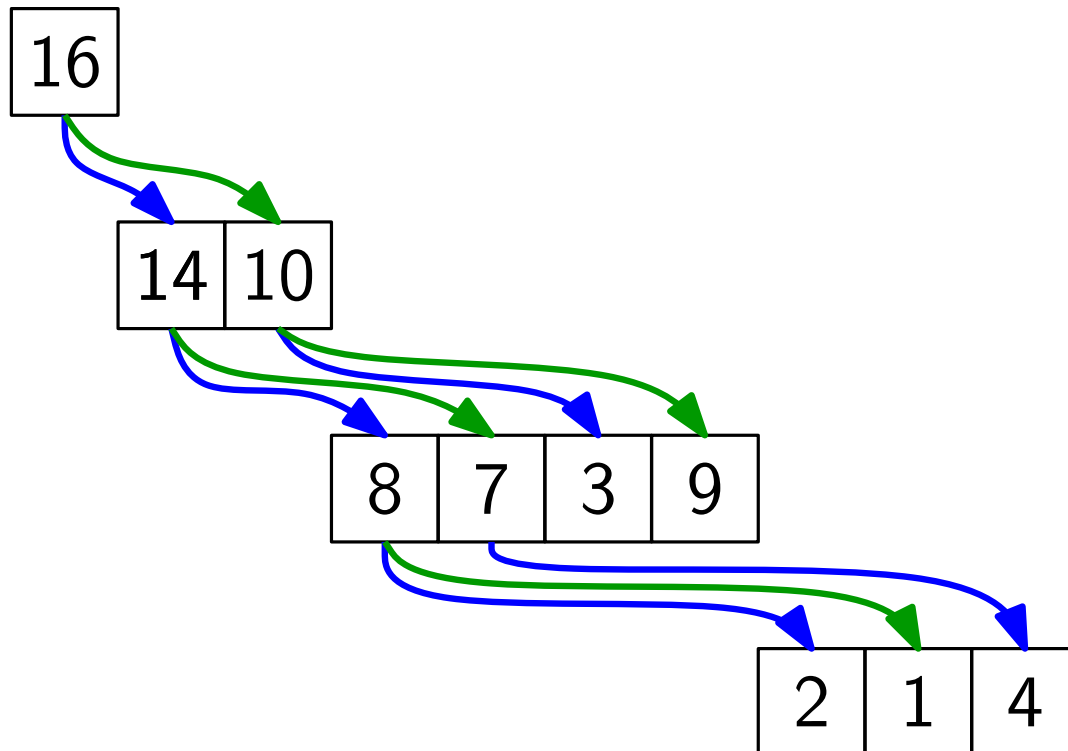
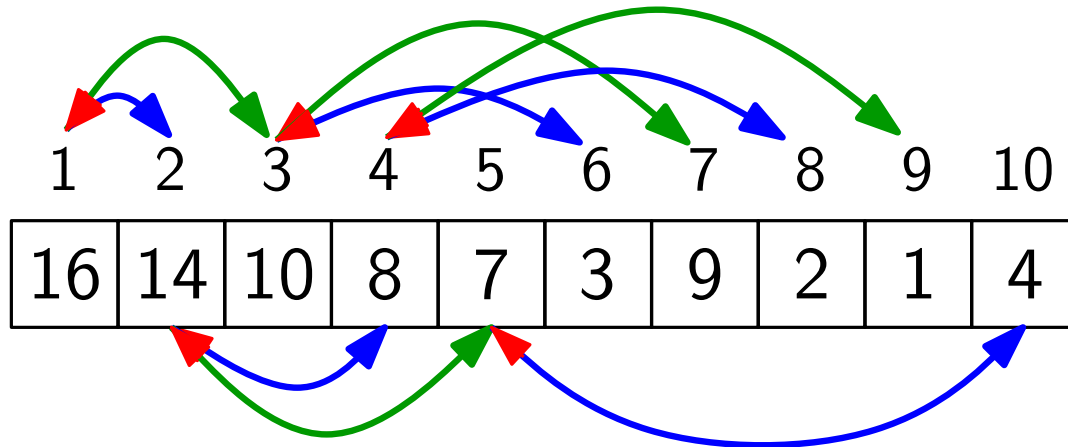
- * {
- Daten werden in einem Feld gespeichert.
 - Neue Elemente werden hinten angehängt (unsortiert).
 - Maximum wird immer aufrechterhalten.
 - Bei IncreaseKey gehe ich von Direktzugriff (via Index) aus.

Achtung: Das Feld bekommt bei einer naiven Implementierung durch ExtractMax im Laufe der Zeit Lücken. Wie kann man das verhindern, ohne Elemente zu verschieben?

Bäume, gut gepackt



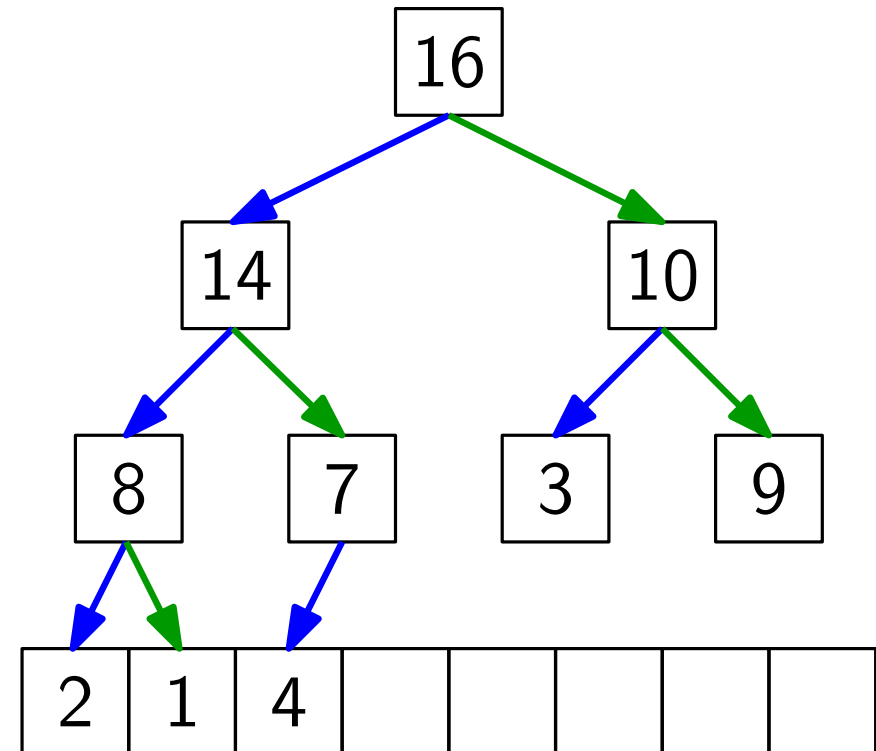
Bäume, gut gepackt



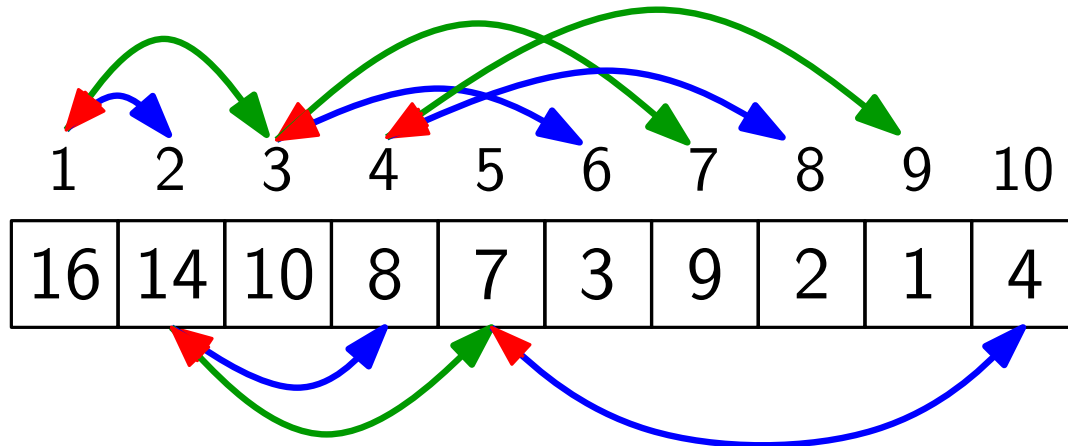
sehr schnelle Rechenoperationen!

Pfeile implementieren:

$\text{left}(\text{index } i)$ **return** $2i$
 $\text{right}(\text{index } i)$ **return** $2i + 1$
 $\text{parent}(\text{index } i)$ **return** $\lfloor i/2 \rfloor$



Bäume, gut gepackt



Definition:

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

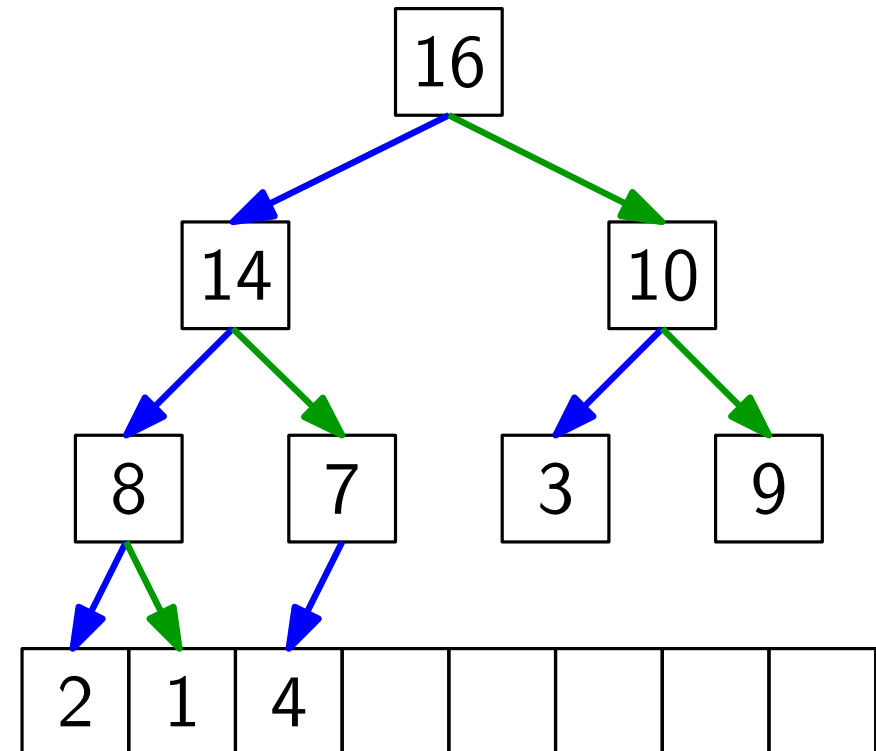
- alle Ebenen außer der letzten voll sind,
- die letzte Ebene v.l.n.r. gefüllt ist und
- die *Heap-Eigenschaft* gilt.

sehr schnelle Rechenoperationen!

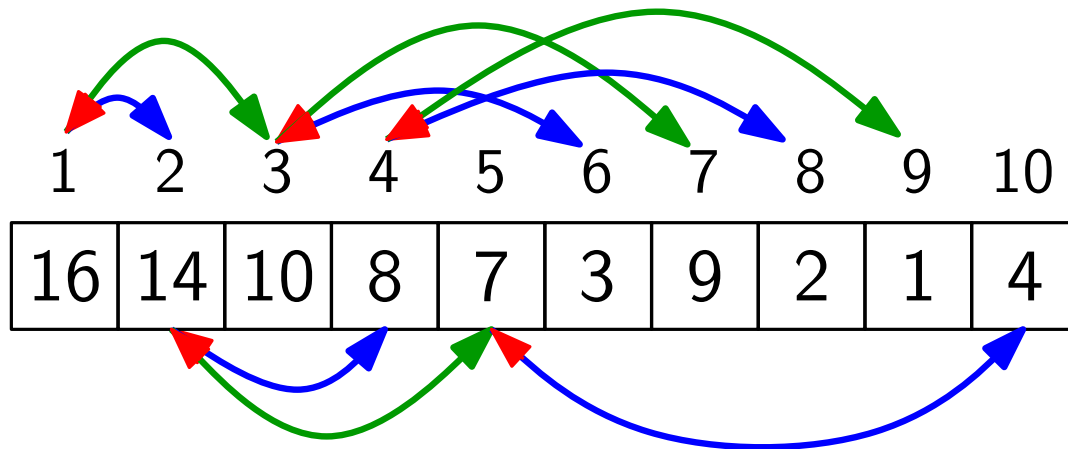
Pfeile implementieren:

```

left(index  $i$ )    return  $2i$ 
right(index  $i$ )   return  $2i + 1$ 
parent(index  $i$ ) return  $\lfloor i/2 \rfloor$ 
  
```



Bäume, gut gepackt



Definition:

Ein Heap hat die

Max-Heap-Eigenschaft,

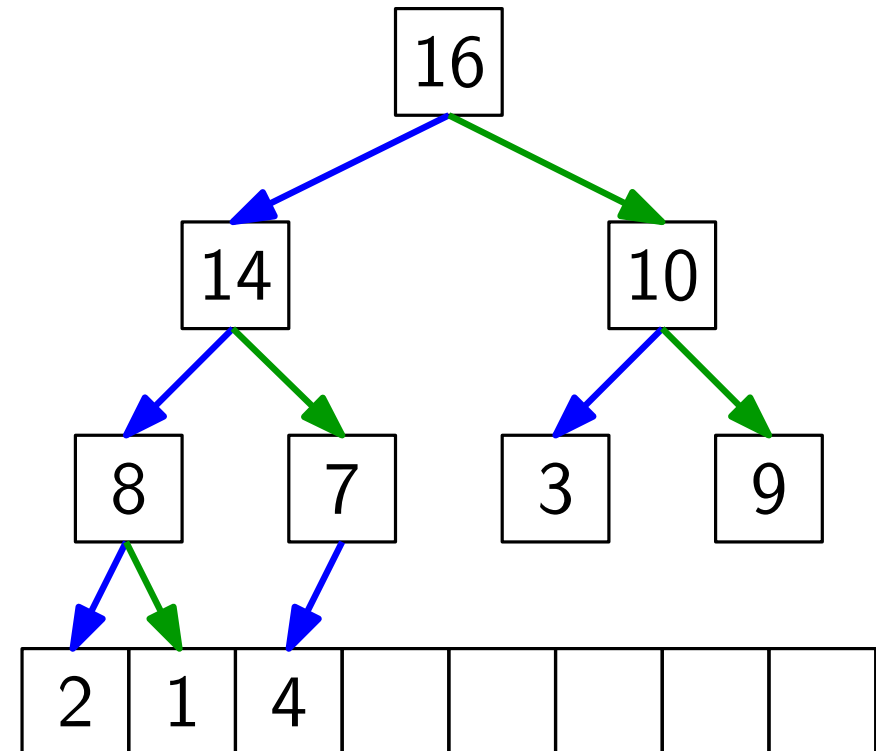
wenn für jeden Knoten $i > 1$ gilt:
 $A[\text{parent}(i)] \geq A[i]$.

So ein Heap heißt *Max-Heap*.

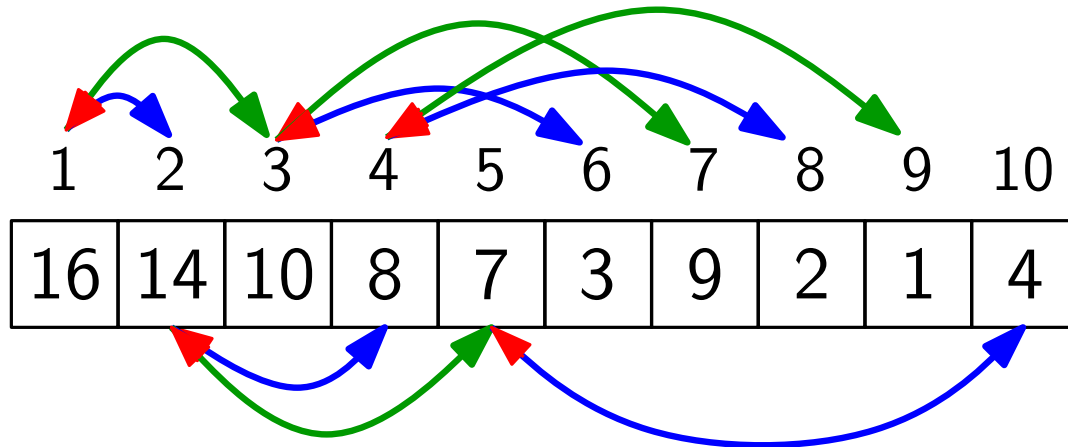
sehr schnelle Rechenoperationen!

Pfeile implementieren:

$\text{left}(\text{index } i)$ **return** $2i$
 $\text{right}(\text{index } i)$ **return** $2i + 1$
 $\text{parent}(\text{index } i)$ **return** $\lfloor i/2 \rfloor$



Bäume, gut gepackt



Definition:

Ein Heap hat die

~~Min~~ ~~Max~~-Heap-Eigenschaft,
wenn für jeden Knoten $i > 1$ gilt:
 $A[\text{parent}(i)] \not\leq A[i]$.

So ein Heap heißt ~~Max~~-Heap.

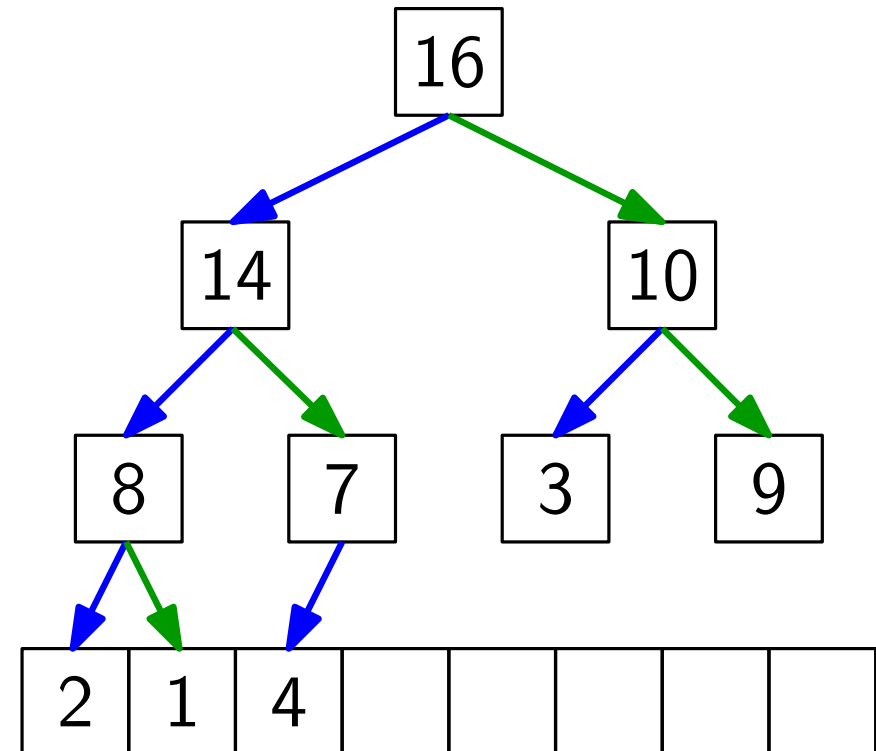
sehr schnelle Rechenoperationen!

Pfeile implementieren:

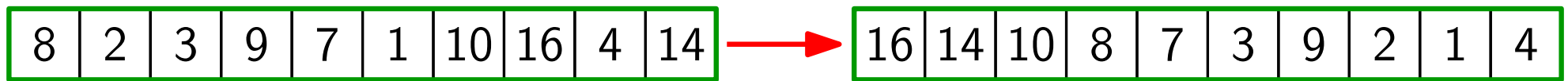
```

left(index  $i$ )    return  $2i$ 
right(index  $i$ )   return  $2i + 1$ 
parent(index  $i$ ) return  $\lfloor i/2 \rfloor$ 

```



Baustelle

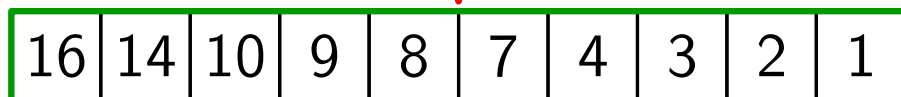


„totales Chaos“

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

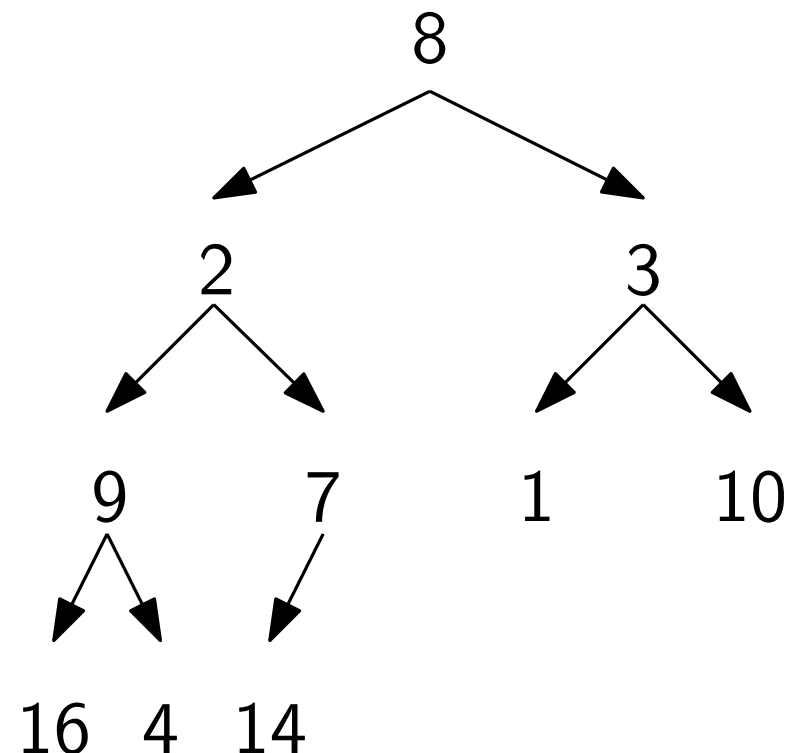


Absteigende Sortierung

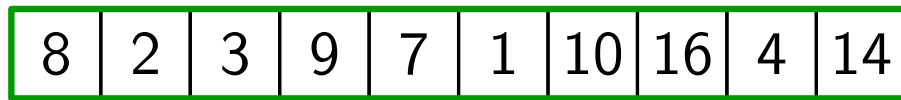
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

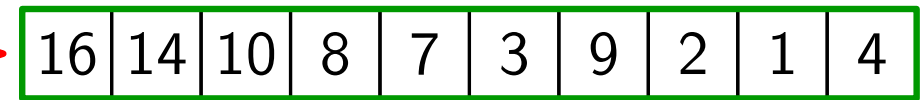
Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle



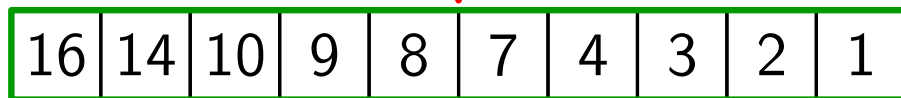
„totales Chaos“



Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

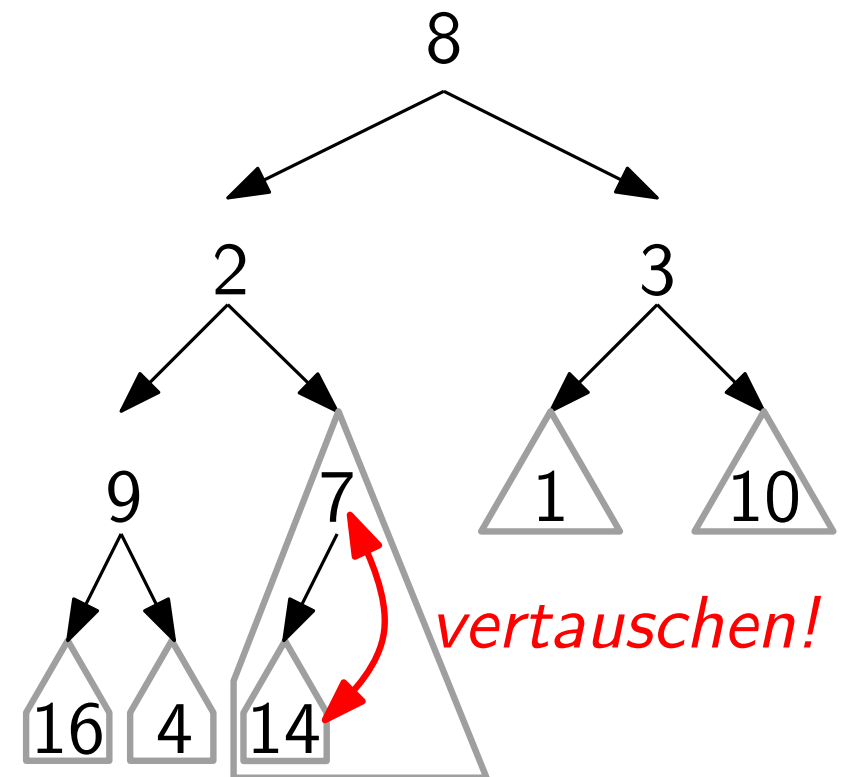


Absteigende Sortierung

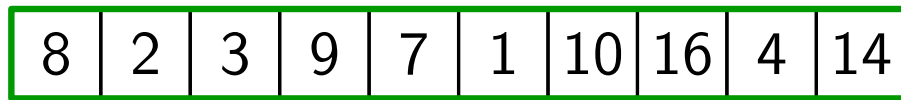
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

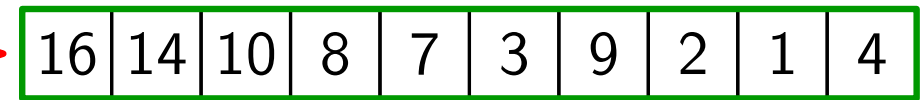
Idee: Nutze Baumstruktur!
Arbeite *bottom-up*:
Erst die Blätter...



Baustelle



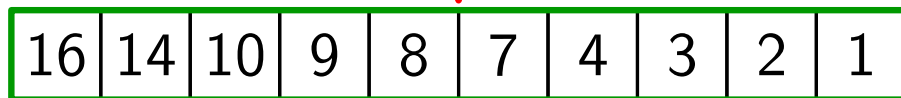
„**totales Chaos**“



Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

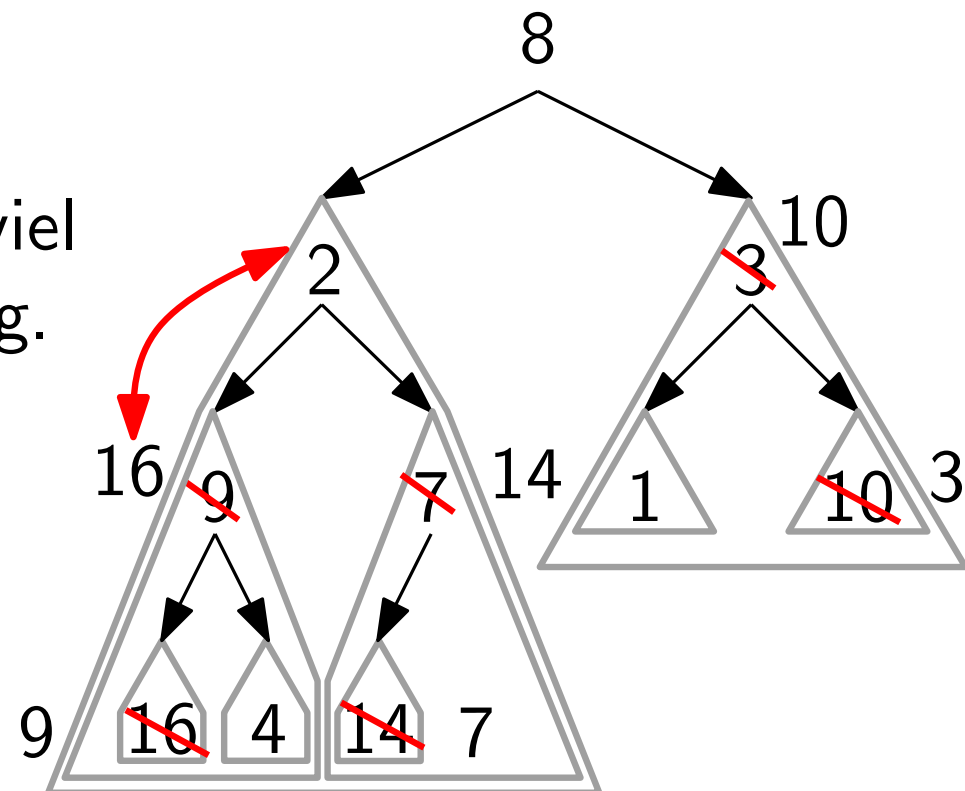


Absteigende Sortierung

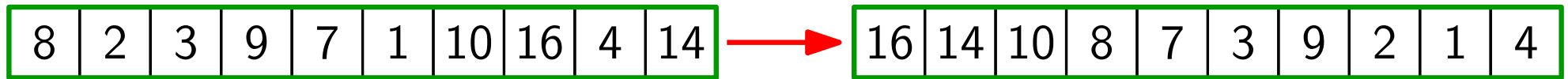
Fertig? Nicht ganz: Heap-Eig. viel schwächer als Sortierung.

Hoffen: Schnellere Berechnung!

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



Baustelle

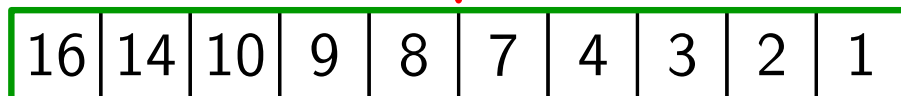


„**totales Chaos**“

Max-Heap-Eigenschaft

Aufgabe: Berechnen Sie in $O(n \log n)$ Zeit einen Max-Heap!

Nimm MergeSort!

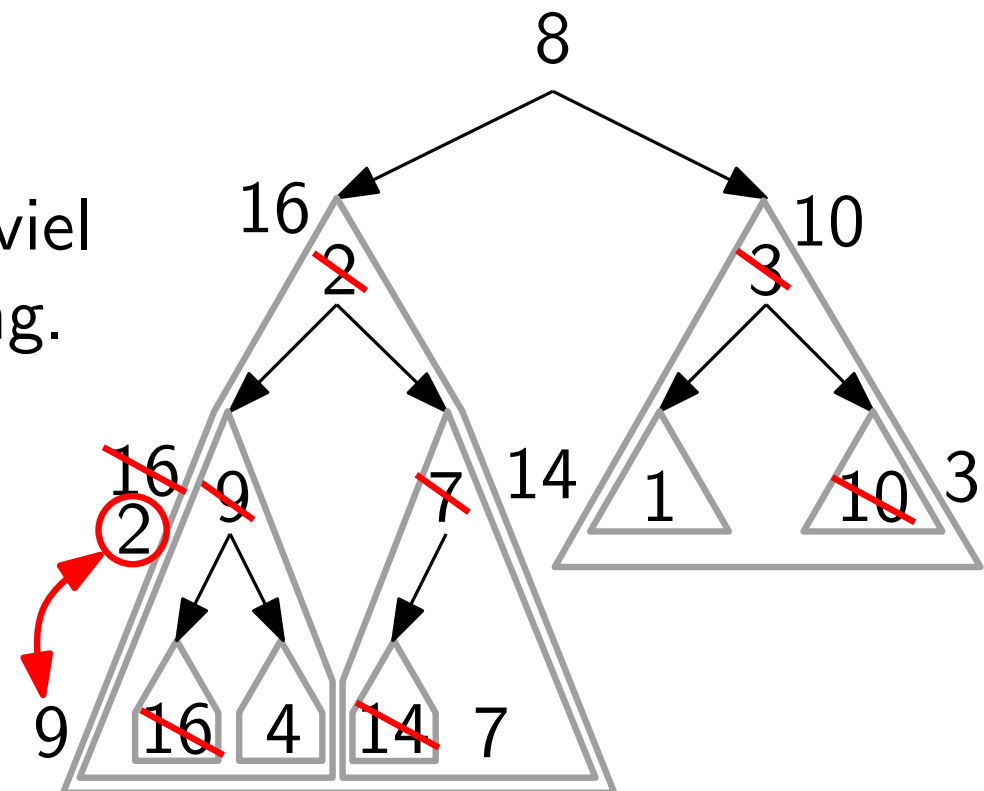


Absteigende Sortierung

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

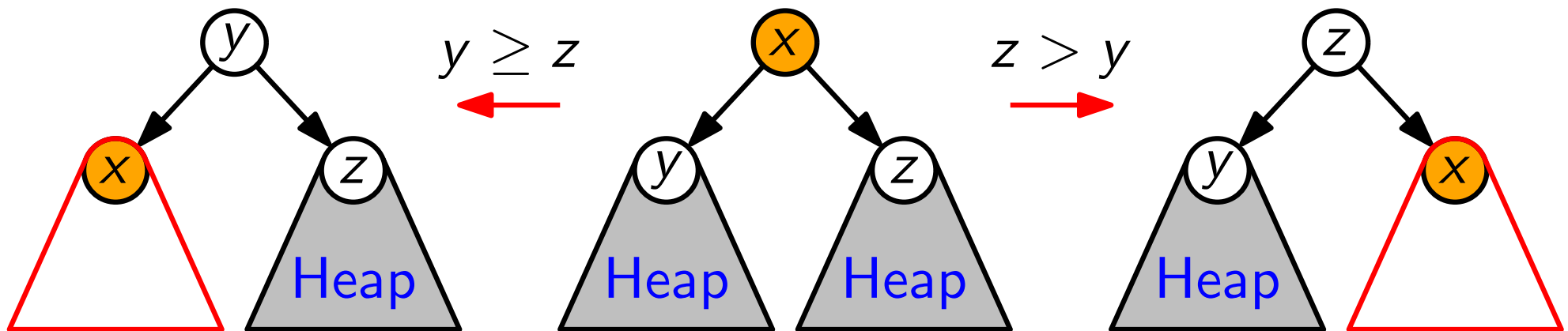
Hoffen: Schnellere Berechnung!

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



Elementaroperation

„Versickere“ x , falls x zu klein, d.h. falls $x < \max(y, z)$



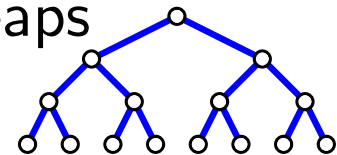
```

MaxHeapify(int A[], index i)
  l = left(i); r = right(i)
  if l ≤ A.heap-size and A[l] > A[i] then
    largest = l
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
  
```

Lokale Strategie: *top-down*

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

- := Anzahl der Swaps
- ≤ Länge des Weges von Knoten i zu einem Blatt
- ≤ Höhe von i im Heap
- ≤ Höhe des Heaps
- ≤ $\lfloor \log_2 n \rfloor$



Das große Ganze

Lokale Strategie: *top-down*

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

Globale Strategie: *bottom-up*

```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1
    do MaxHeapify(A, i)
```

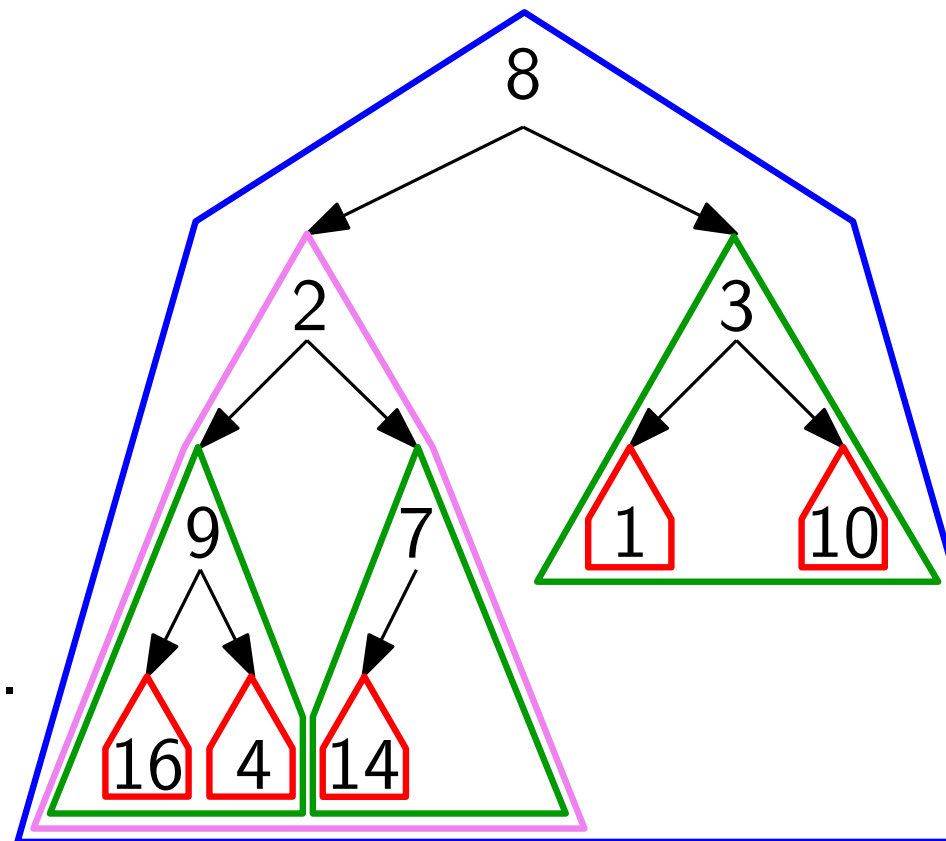
Laufzeit. grob: $O(n \log n)$

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

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

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

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



Forts. Laufzeitanalyse

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

Vgl. geometrische Reihe: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ (falls $|x| < 1$)

ableiten!

ableiten!

Wir hätten gerne: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$

Quotientenregel:

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

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

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

Übung Heap-Aufbau

Aufgabe: Bauen Sie einen Heap mit BuildMaxHeap!



```
BuildMaxHeap(int A[])
  A.heap-size = A.length
  for i = ⌊A.length/2⌋ downto 1 do
    MaxHeapify(A, i)
```

```
MaxHeapify(int A[], index i)
  ℓ = left(i); r = right(i)
  if ℓ ≤ A.heap-size and A[ℓ] > A[i] then
    largest = ℓ
  else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
  if largest ≠ i then
    swap(A, i, largest)
    MaxHeapify(A, largest)
```

Zurück zu Prioritätsschlangen

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge,
wobei jedes Element der Menge eine Priorität hat.

FindMax() $O(\quad)$
return A[1]

Laufzeiten?

ExtractMax() $O(\quad)$
if $A.heap-size < 1$ then
 error "Heap underflow"
 $max = A[1]$
 $A[1] = A[A.heap-size]$
 $A.heap-size --$
MaxHeapify(A, 1)
return max

IncreaseKey(index i , prio. p) $O(\quad)$
if $p < A[i]$ then error "prio. too small"
 $A[i] = p$
while $i > 1$ and $A[parent(i)] < A[i]$
 swap(A, i , parent(i))
 $i = parent(i)$

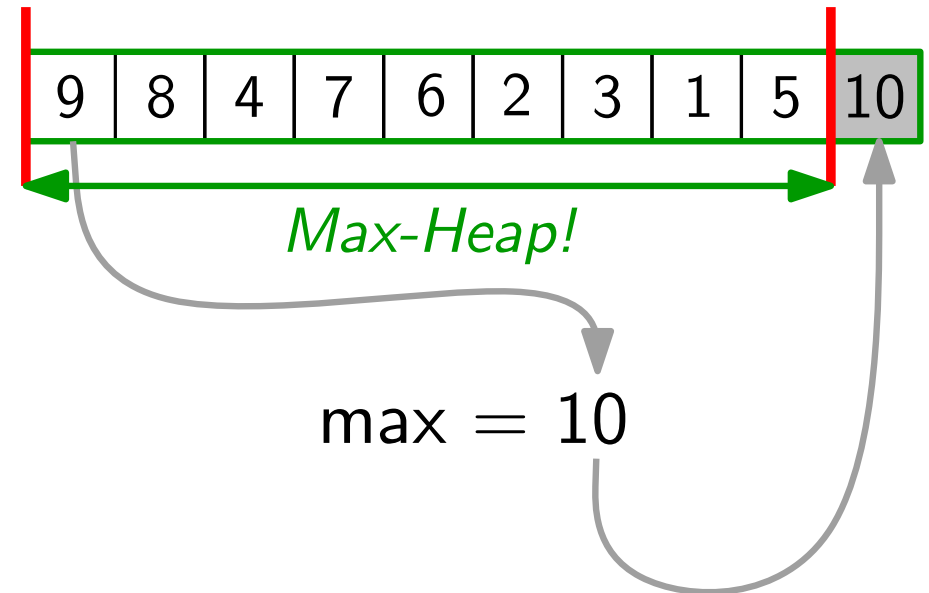
Insert(priorität p) $O(\quad)$
 $A.heap-size ++$
if $A.heap-size > A.length$ then error...
 $A[A.heap-size] = -\infty$
IncreaseKey($A.heap-size$, p)

Vom Heap zur Sortierung

- Idee:**
- ExtractMax() gibt rechtestes Heap-Element frei.
 - Speichere dort das extrahierte Maximum.

HeapSort(int[] A)

Schreiben Sie den Pseudocode.
Verwenden Sie BuildMaxHeap
und ExtractMax.

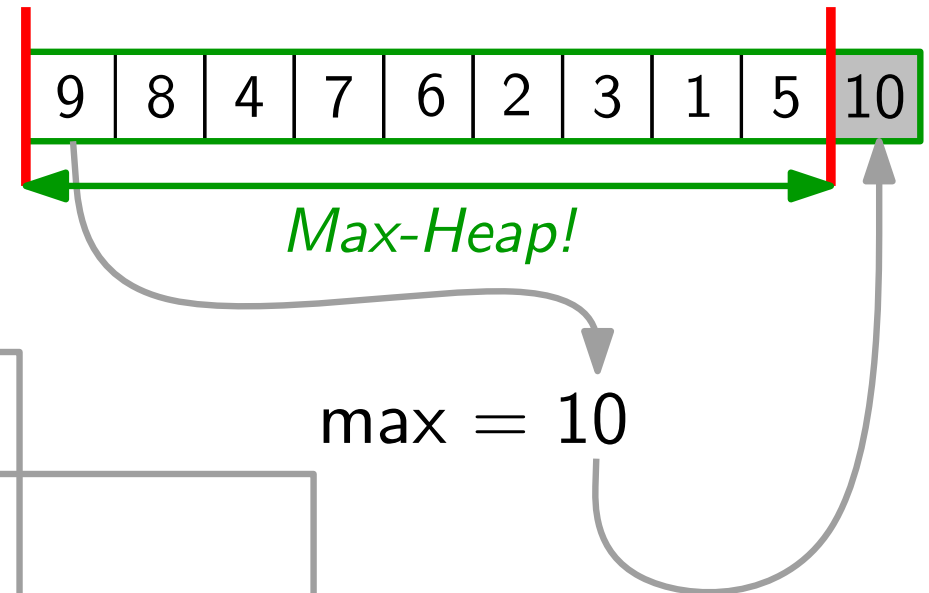


Vom Heap zur Sortierung

- Idee:**
- ExtractMax() gibt rechtestes Heap-Element frei.
 - Speichere dort das extrahierte Maximum.

```

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



Laufzeit: $T_{HS}(n) \in O(n) + (n - 1) \cdot O(\log n) = O(n \log n)$

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

Zusammenfassung Sortierverfahren

	InsertionSort	MergeSort	HeapSort
Worst-Case-Laufzeit	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Avg.-Case-Laufzeit	$\Theta(n^2)$ <i>Warum?</i>	$\Theta(n \log n)$	$\Theta(n \log n)$
Best-Case-Laufzeit	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
in situ ¹ (<i>in place</i>)	✓	✗	✓
stabil ²	✓	✓	✗

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

²) Sortieralg. *stabil*, wenn er gleiche Schlüssel in Ursprungsreihenfolge belässt.