7. Heapsort

- Werden sehen, wie wir durch geschicktes Organsieren von Daten effiziente Algorithmen entwerfen können.

- Genauer werden wir immer wieder benötigte Operationen durch Datenstrukturen unterstützen.

- Heapsort: Heap (Familie der Priority Queues)

- Werden im Laufe des Semesters viel mehr über Datenstrukturen und ihren Zusammenhang mit effizienten Algorithmen lernen.

Heapsort

Motivation: Betrachte folgendes Sortierverfahren.

Eingabe: Array *A*

Ausgabe: Zahlen in *A* in aufsteigender Reihenfolge sortiert.

Max-Sort(A):
  for $i \leftarrow$ length(A) downto 2 do
    $m \leftarrow$ Max-Search(A[1..i]) // gibt Index des Max. zurück
    $A[m] \leftrightarrow A[i]$

Frage: können wir mithilfe einer geeigneten Datenstruktur schneller das Maximum bestimmen als mit Max-Search?

Priority Queue

M: Menge von Elementen

Jedes Element e identifiziert über key(e).

Operationen:

- insert(M,e): $M := M \cup \{e\}$
- min(M): gib $e \in M$ mit minimalem key(e) aus
- deleteMin(M): wie M.min, aber zusätzlich $M := M \setminus \{e\}$, für e mit minimalem key(e)

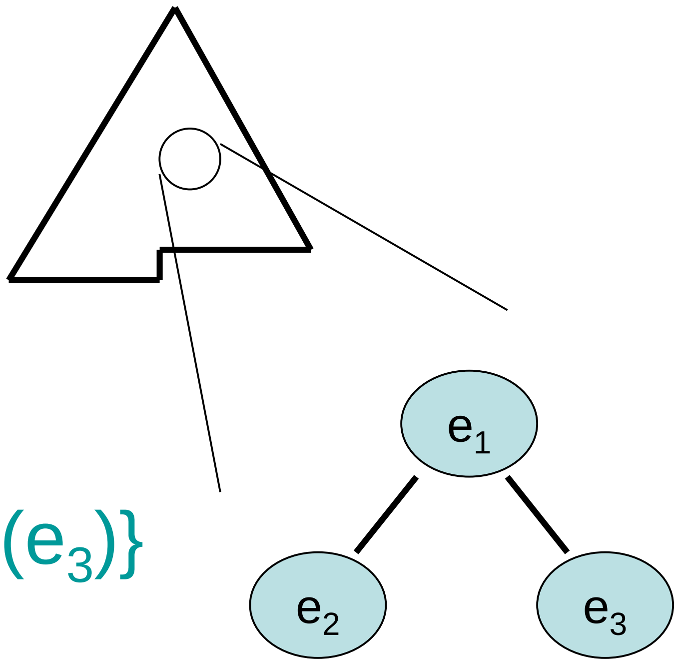
Priority Queue als Heap

Idee: organisiere Daten im binären Baum

Bewahre zwei Invarianten:

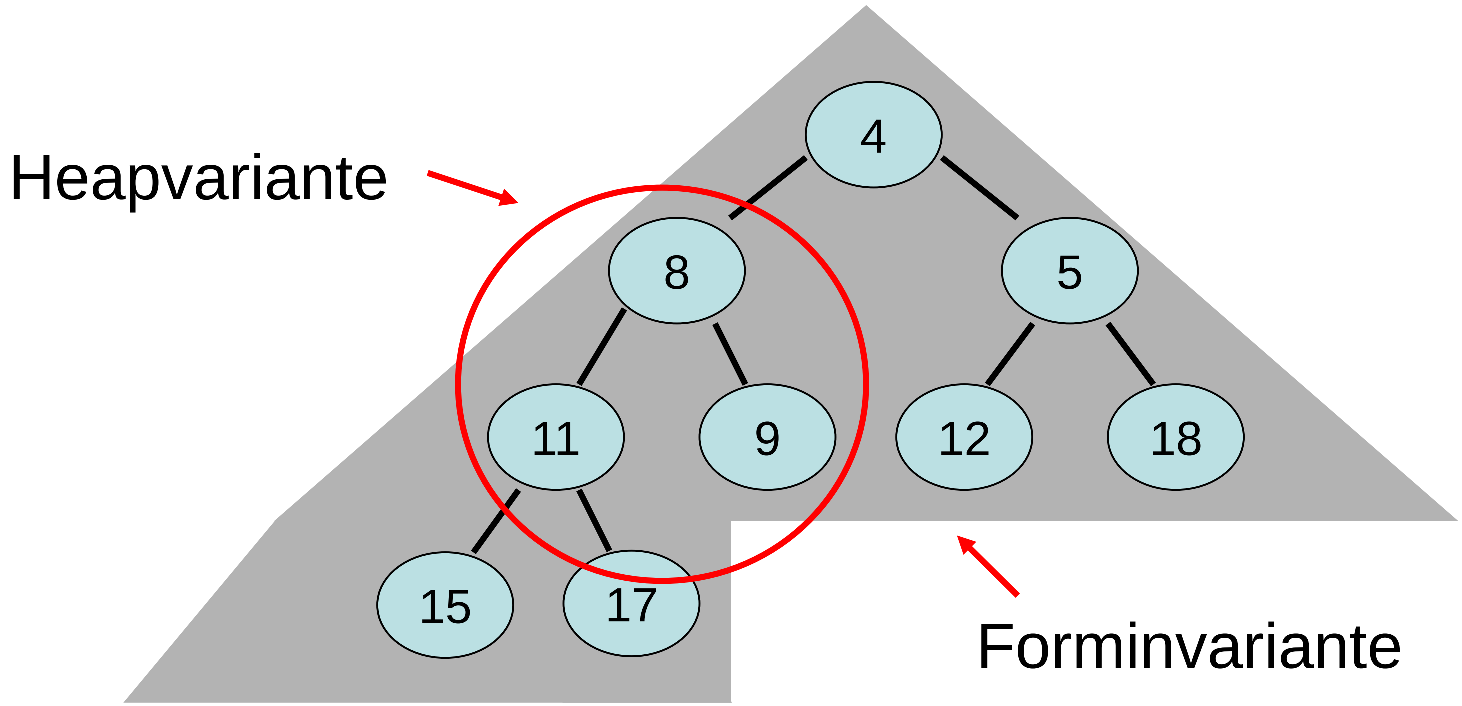
- Form-Invariante: vollst. Binärbaum bis auf unterste Ebene
- Heap-Invariante:

$$key(e_1) \leq \min\{key(e_2), key(e_3)\}$$



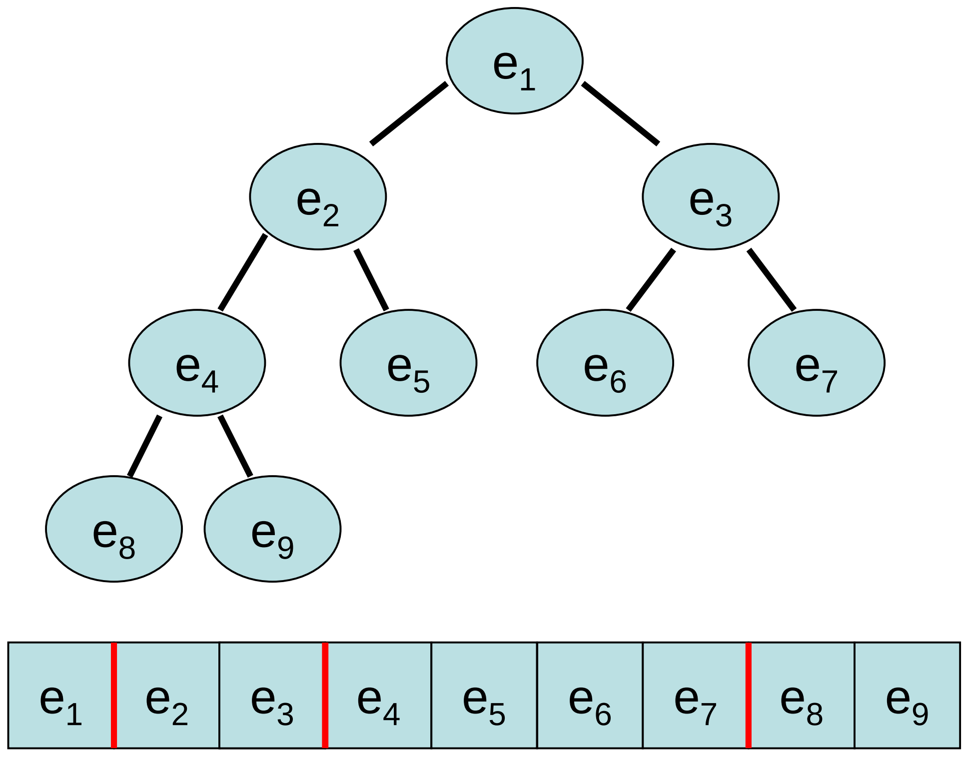
Heap

Beispiel:



Heap

Realisierung eines Binärbaums als Feld:



Heap

Realisierung eines Binärbaums als Feld:

e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9

- A: Array [1..N] of Element ($N \geq n$, n=#Elemente)
- Kinder von e in A[i]: in A[2i], A[2i+1]
- Form-Invariante: A[1],…,A[n] besetzt
- Heap-Invariante:

$$key(A[i]) \leq \min\{key(A[2i]), key(A[2i+1])\}$$

Heap

Definition 7.1: Ein Heap über einem Array A ist das Array A der Größe N zusammen mit einem Parameter n:=heap-size[A]$\leq N$ und drei Funktionen

$$\text{Parent, Left, Right: } \{1,\dots,n\} \to \{1,..,n\}$$

Dabei gilt:

1. $1 \leq n \leq N$
2. $\text{Parent}(i) = \lfloor i/2 \rfloor$ für alle $i \in \{1,..,n\}$
3. $\text{Left}(i) = 2i$ für alle $i \in \{1,\dots,n\}$
4. $\text{Right}(i) = 2i+1$ für alle $i \in \{1,\dots,n\}$

Die Elemente $A[1],\dots,A[n]$ heißen Heapelemente.

Heap

Definition 7.2: Ein Heap heißt

1. max-Heap, wenn für alle $i \in \{2,\ldots,n\}$ gilt

$$\mathrm{key}(A[\mathrm{Parent}(i)]) \geq \mathrm{key}(A[i])$$

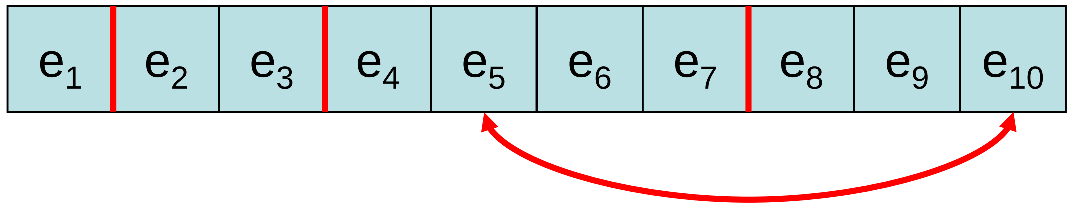
2. min-Heap, wenn für alle $i \in \{2,\ldots,n\}$ gilt

$$\mathrm{key}(A[\mathrm{Parent}(i)]) \leq \mathrm{key}(A[i])$$

Bemerkungen:

- $\mathrm{key}(A[\mathrm{Parent}(i)]) \leq \mathrm{key}(A[i])$ ist äquivalent zu $\mathrm{key}(A[i]) \leq \min\{\mathrm{key}(A[\mathrm{Left}(i)]),\mathrm{key}(A[\mathrm{Right}(i)])\}$
- Wir werden uns zunächst mit dem min-Heap beschäftigen und diesen einfach Heap nennen.

Realisierung eines Binärbaums als Feld:



insert(A,e):

- Form-Invariante: $n \leftarrow n+1$; $A[n] \leftarrow e$
- Heap-Invariante: vertausche e mit Vater bis $key(A[Parent(k)]) \leq key(e)$ für e in $A[k]$ oder e in $A[1]$

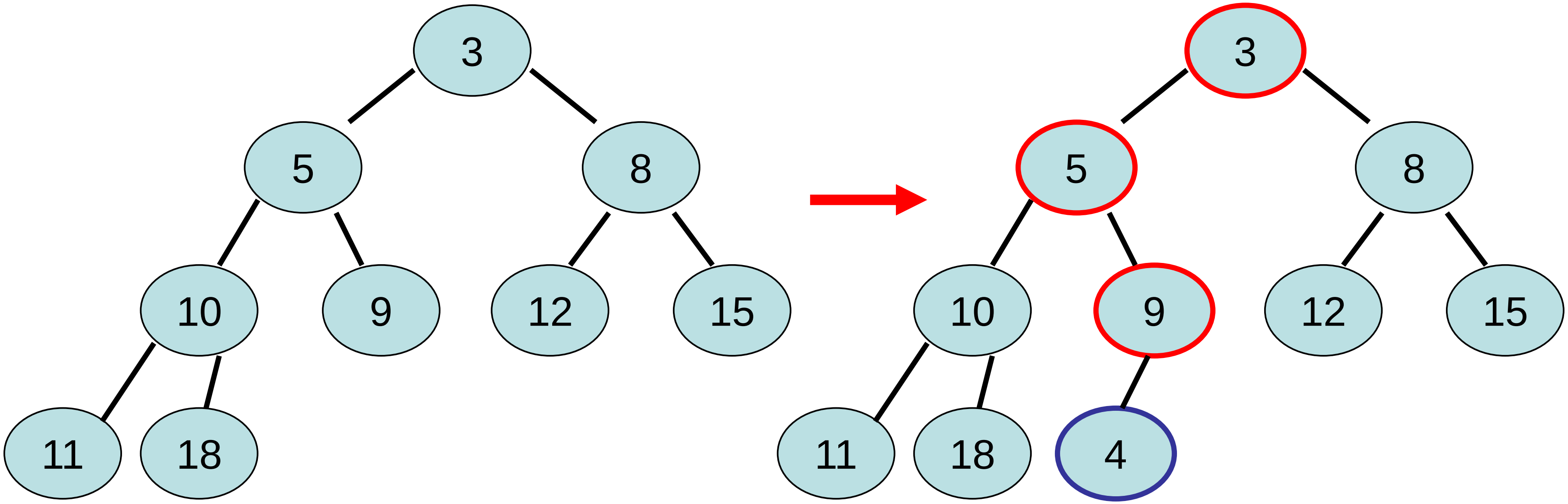
Insert Operation

Insert(A,e):
  $n \leftarrow n+1$; $A[n] \leftarrow e$
  HeapifyUp(A,n)

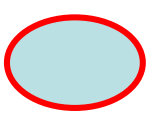
HeapifyUp(A,i):
  while $i>1$ and key(A[Parent(i)])>key(A[i]) do
    $A[i] \leftrightarrow A[\mathrm{Parent}(i)]$
    $i \leftarrow \mathrm{Parent}(i)$

Für Analyse: wir nehmen vereinfachend an,
  dass key(A[i]) = A[i].

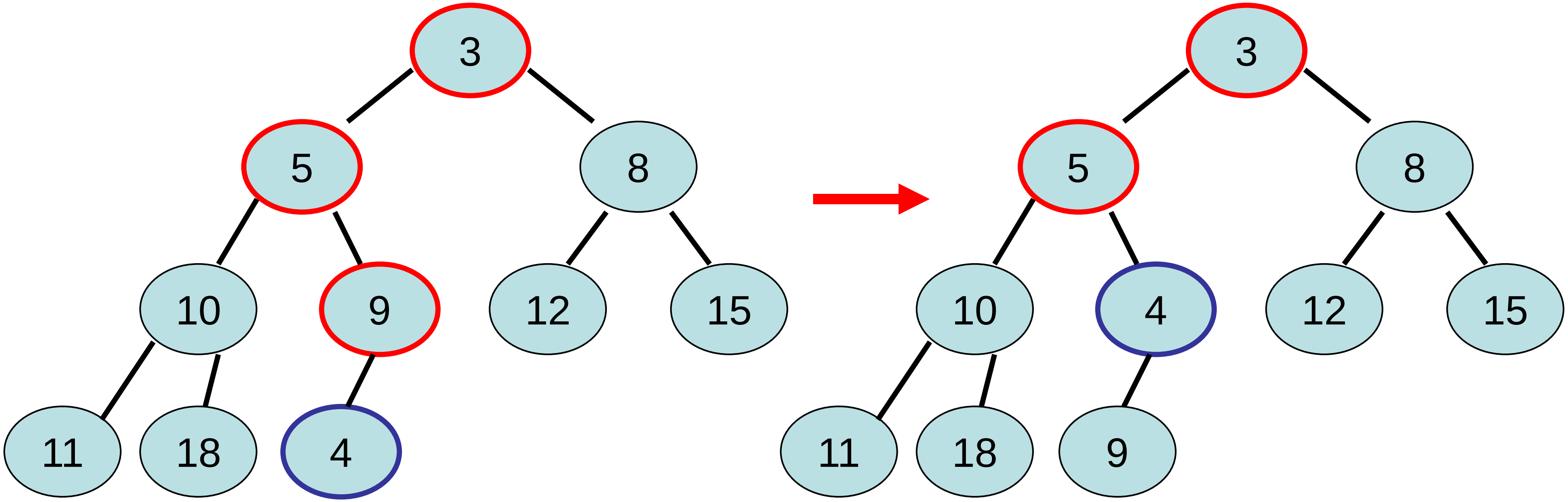
Insert Operation



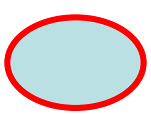
Invariante: A[k] ist minimal für Teilbaum von A[k]

 : Knoten, die Invariante eventuell verletzen

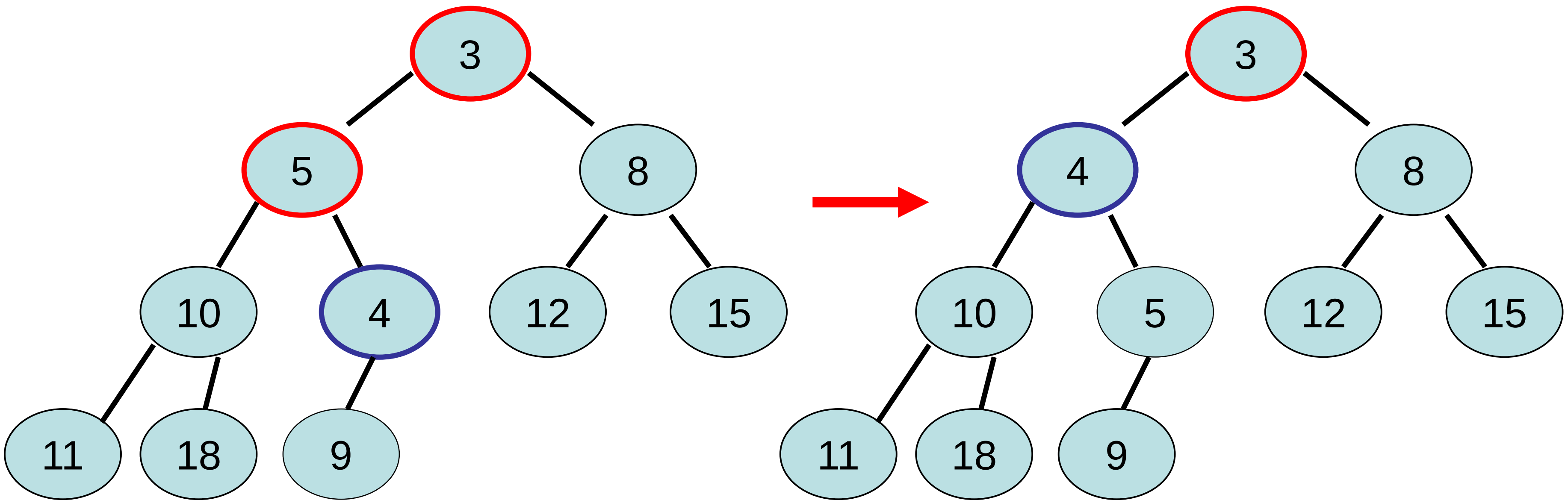
Insert Operation



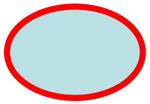
Invariante: A[k] ist minimal für Teilbaum von A[k]

 : Knoten, die Invariante eventuell verletzen

Insert Operation

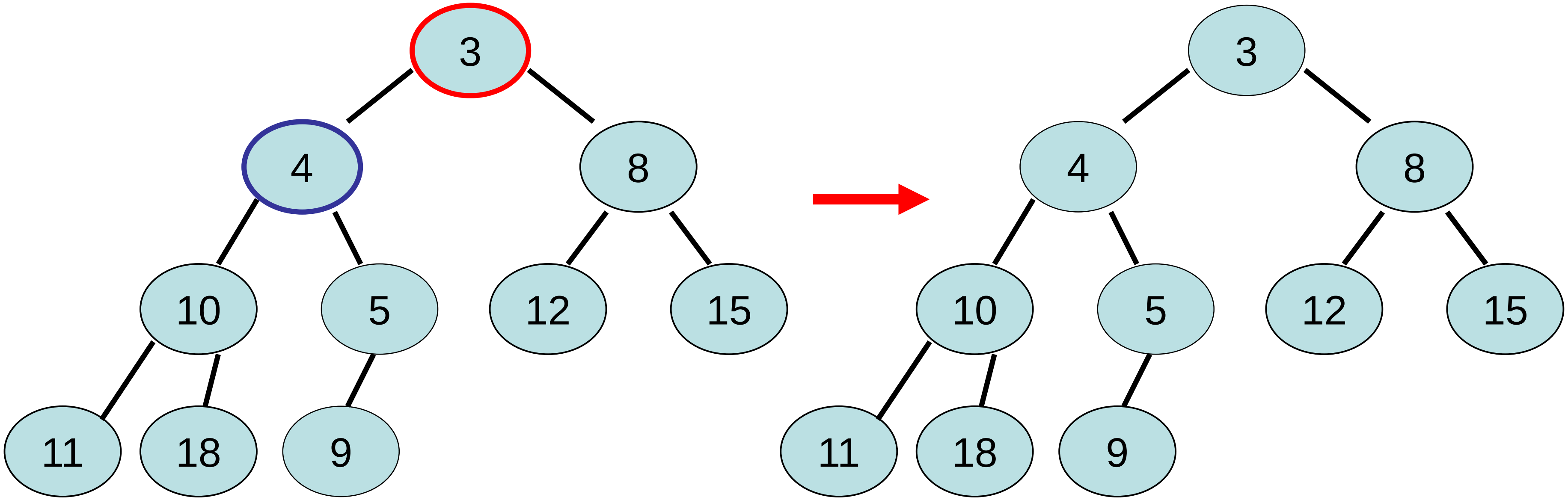


Invariante: A[k] ist minimal für Teilbaum von A[k]

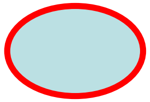


: Knoten, die Invariante eventuell verletzen

Insert Operation



Invariante: A[k] ist minimal für Teilbaum von A[k]


: Knoten, die Invariante eventuell verletzen

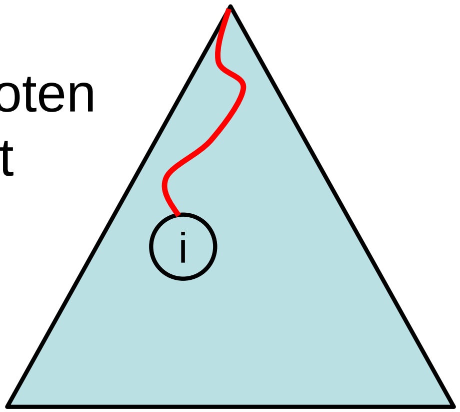
Insert Operation

- $P(i)$: Menge der Pos. aller Vorgänger von $A[i]$ (d.h. $\lfloor i/2 \rfloor$, $\lfloor i/4 \rfloor$ usw. bis 1)
- $T(i)$: Menge der Pos. aller Elemente im Teilbaum mit Wurzel $A[i]$ (d.h. $2i$, $2i+1$, $2(2i)$, $2(2i+1)$,…)

Schleifeninvariante $I(i)$: (i: Pos. des eingefügten Elements)

$\forall\ j\in\{1,\ldots,n\}$: $A[j]=\min\{\ A[k]\ |\ k\in T(j)\setminus\{i\}\ \}$

Heap OK außer für Knoten in $P(i)$, für die i ignoriert werden muss



Insert Operation

Schleifeninvariante I(i):
$\forall\ j\in\{1,\dots,n\}$: $A[j]=\min\{\, A[k] \mid k\in T(j)\setminus\{i\}\,\}$

Initialisierung: zu Beginn von HeapifyUp ist I(n) trivialerweise wahr und damit auch I(i) am Anfang des ersten Durchlaufs der while-Schleife.

Erhaltung:

- Anfangs gelte I(i). Da $A[\lfloor i/2\rfloor]>A[i]$ und nach I(i) auch $A[\lfloor i/2\rfloor]=\min\{\, A[k] \mid k \in T(\lfloor i/2\rfloor)\setminus\{i\}\,\}$, ist nach Vertau-schung von $A[i]$ und $A[\lfloor i/2\rfloor]$ sowohl $A[i]=\min\{\, A[k] \mid k\in T(j)\setminus\{\lfloor i/2\rfloor\}\,\}$ als auch $A[\lfloor i/2\rfloor]=\min\{\, A[k] \mid k\in T(j)\setminus\{\lfloor i/2\rfloor\}\,\}$ und damit I(i) am Ende der while-Schleife wieder wahr.

Insert Operation

Terminierung: am Ende gilt $I(i)$ mit $i=1$ oder $A[\lfloor i/2 \rfloor] \leq A[i]$.

Fall 1: $i=1$: Wegen $I(i)$ gilt dann $A[j]=\min\{ A[k] \mid k \in T(j) \}$ für alle j, d.h. Heapeigenschaft überall wieder hergestellt

Fall 2: $A[\lfloor i/2 \rfloor] \leq A[i]$: Wegen $I(i)$ gilt für alle $j \notin P(i)$ bereits, dass $A[j]=\min\{ A[k] \mid k \in T(j) \}$. Für alle $j \in P(i)$ wissen wir aber durch $I(i)$ nur, dass $A[j]=\min\{ A[k] \mid k \in T(j) \setminus \{i\} \}$. Daraus folgt, dass für alle $j \in P(i) \setminus \{\lfloor i/2 \rfloor\}$, $A[j] \leq A[\lfloor i/2 \rfloor]$. Da weiterhin $A[\lfloor i/2 \rfloor] \leq A[i]$, ist damit auch $A[j] \leq A[i]$ für alle $j \in P(i)$ und daher $A[j]=\min\{ A[k] \mid k \in T(j) \}$ für alle $j \in P(i)$. Auch in diesem Fall ist die Heapeigenschaft also wieder hergestellt.

Insert Operation

Laufzeit:

Insert(A,e):
  $n \leftarrow n+1$; $A[n] \leftarrow e$ $\quad O(1)$
  HeapifyUp(A,n)

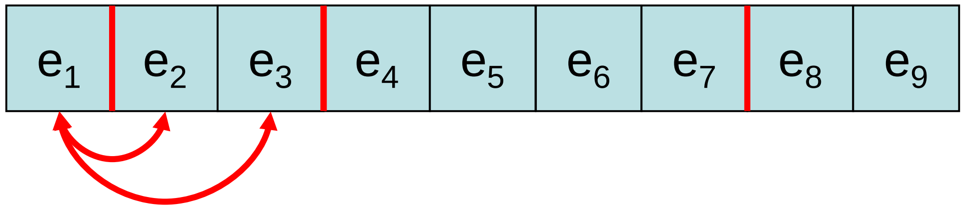
HeapifyUp(A,i):
  while $i>1$ and $key(A[Parent(i)])>key(A[i])$ do $\quad \Sigma_{j=1}^{k}(T(B)+T(I))$
    $A[i] \leftrightarrow A[Parent(i)]$ $\quad O(1)$
    $i \leftarrow Parent(i)$ $\quad O(1)$

$O(k)$

Problem: was ist k?
Verwende Potenzialfunktion $\phi(j) = \log i(j)$
Anfang: $\phi(1) = \log n$
Für alle j: $\phi(j+1) \leq \phi(j) -1$
Ende spätestens wenn $\phi(j) \leq 0$

Also ist $k \leq \log n +1$

deleteMin Operation



deleteMin(A):

- Form-Invariante: $A[1] \leftarrow A[n]$; $n \leftarrow n-1$
- Heap-Invariante: starte mit e in A[1]. Vertausche e mit Kind mit min Schlüssel bis $A[k] \leq \min\{A[Left(k)], A[Right(k)]\}$ für Position k von e oder e in Blatt

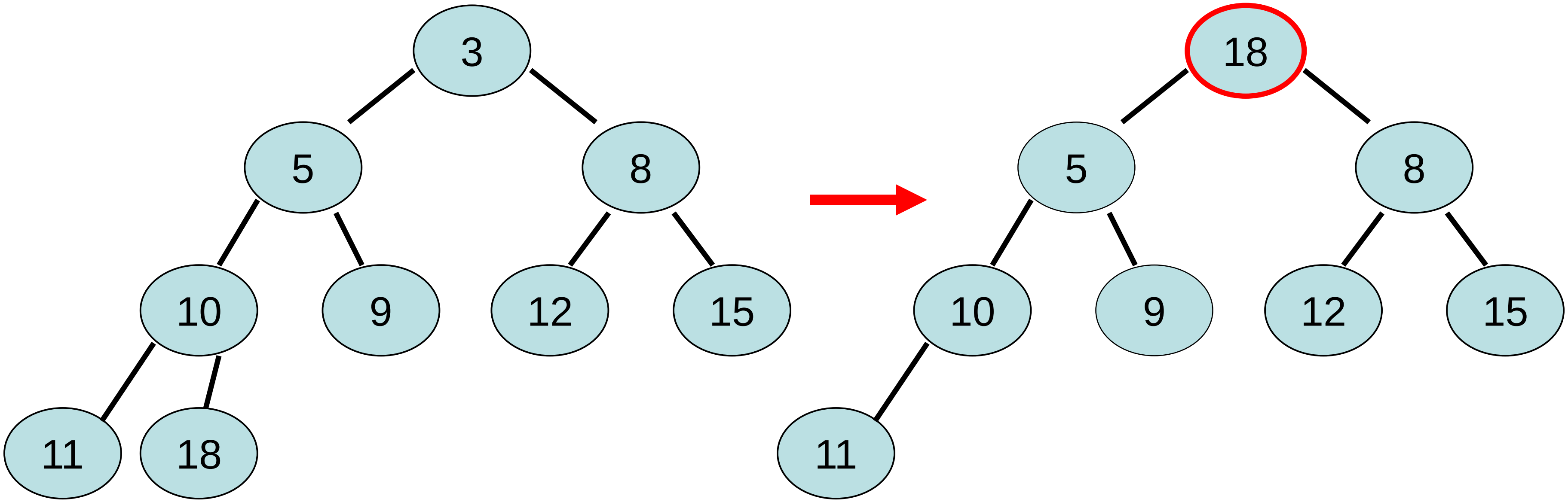
deleteMin Operation

DeleteMin(A):
    e←A[1]; A[1]←A[n]; n←n-1
    HeapifyDown(A,1)
    return e

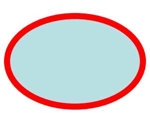
Laufzeit: O(log n)
(über Potenzialmethode)

HeapifyDown(A,i):
    while Left(i)≤n do
        if Right(i)>n then m←Left(i) ▹ m: Pos. des min. Kindes
        else
            if key(A[Left(i)])<key(A[Right(i)]) then m←Left(i)
                                                                else m←Right(i)
        if key(A[i])≤key(A[m]) then return   ▹ Heap-Inv gilt
        A[i] ↔ A[m]; i←m

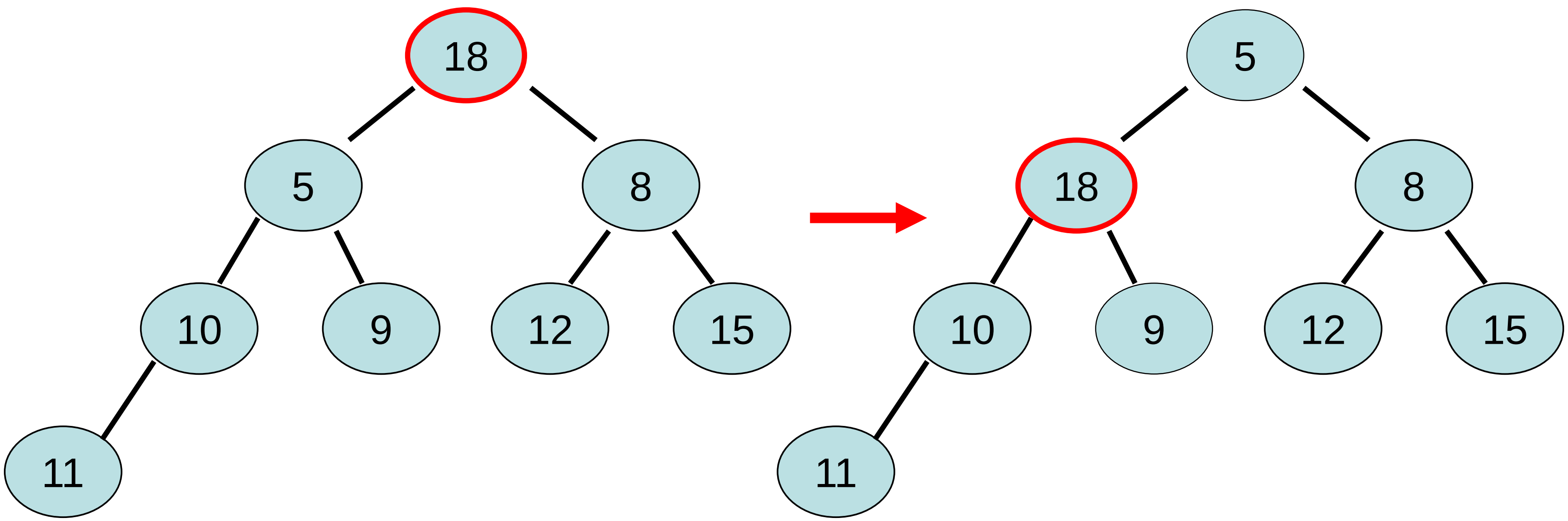
deleteMin Operation



Invariante: A[k] ist minimal für Teilbaum von A[k]

 : Knoten, die Invariante eventuell verletzen

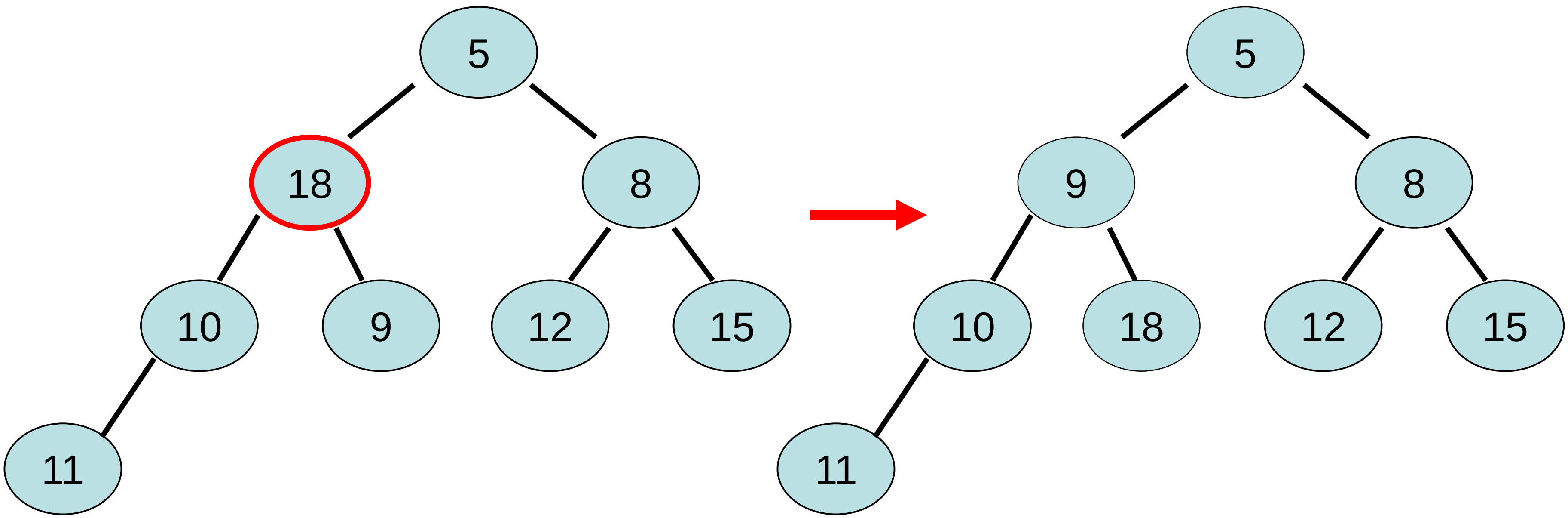
deleteMin Operation



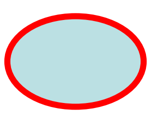
Invariante: A[k] ist minimal für Teilbaum von A[k]

: Knoten, die Invariante eventuell verletzen

deleteMin Operation



Invariante: A[k] ist minimal für Teilbaum von A[k]


: Knoten, die Invariante eventuell verletzen

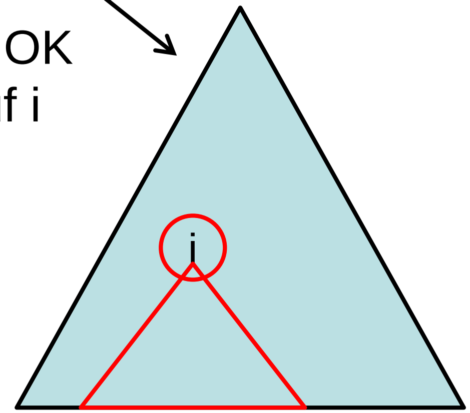
deleteMin Operation

- T(i): Menge der Positionen aller Elemente im Teilbaum mit Wurzel A[i] (d.h. 2i, 2i+1, 2(2i), 2(2i+1),…)

Schleifeninvariante I(i):

$\forall\ j \in \{1,\ldots,n\} \setminus \{i\}$: $A[j]=\min\{\ A[k] \mid k \in T(j)\ \}$

Heap OK
bis auf i



deleteMin Operation

Schleifeninvariante I(i):

$\forall\ j\in\{1,\ldots,n\}\backslash\{i\}: A[j]=\min\{\ A[k]\ |\ k \in T(j)\ \}$

Initialisierung: zu Beginn von HeapifyDown ist I(1) trivialerweise wahr und damit auch I(i) am Anfang des ersten Durchlaufs der while-Schleife.

Erhaltung:

- o.B.d.A. sei $A[2i]=\min\{A[2i],A[2i+1]\}$
- $A[i]>A[2i]$: dann ist $A[2i]\leq\min\{\ A[k]\ |\ k \in T(i)\ \}$

  d.h. nach Vertauschung von A[i] und A[2i] und der Aktualisierung von i auf 2i ist Invariante I(i) wieder wahr

Terminierung: am Ende ist $i>n/2$ oder $A[i]\leq\min\{A[2i],A[2i+1]\}$.

- $i>n/2$: Heapeigenschaft folgt, da A[i] Blatt ist
- $A[i]\leq\min\{A[2i],A[2i+1]\}$: Heapeigenschaft folgt aus I(i)

Aufbau eines Heaps

BuildHeap(A): baue mit Elementen in A einen Heap auf.

- Naive Implementierung über n insert(A,e)-Operationen: Laufzeit $O(n \log n)$
- Bessere Implementierung:

```
BuildHeap(A):
n←heap-size[A]
for i←⌊n/2⌋ downto 1 do
    HeapifyDown(A,i)
```

Aufwand (mit $k=\lceil \log n \rceil$):

$$O(\sum_{0\leq l<k} 2^l (k-l)) = O(2^k \sum_{j\geq 0} j/2^j) = O(n)$$

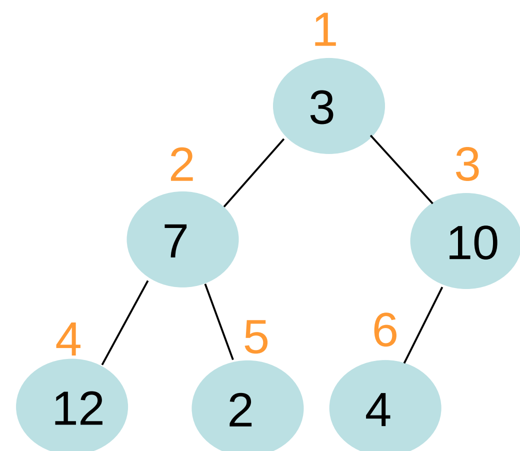
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau eines Heaps:

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

BuildHeap(A)

1. n $\leftarrow$ heap-size[A]
2. **for** i $\leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
3. HeapifyDown(A,i)



Aufbau eines Heaps

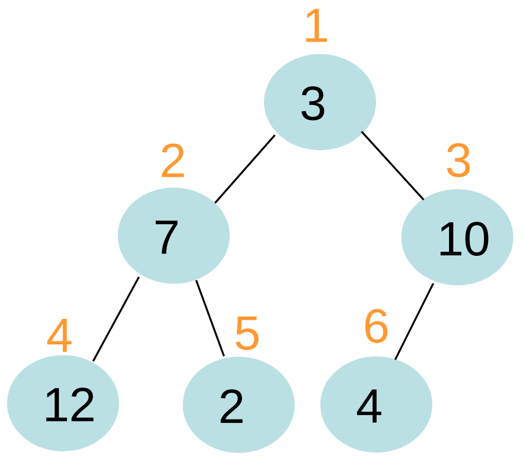
1	2	3	4	5	6
3	7	10	12	2	4

Aufbau eines Heaps:

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

BuildHeap(A)

1. n $\leftarrow$ heap-size[A]
2. **for** i $\leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
3. HeapifyDown(A,i)



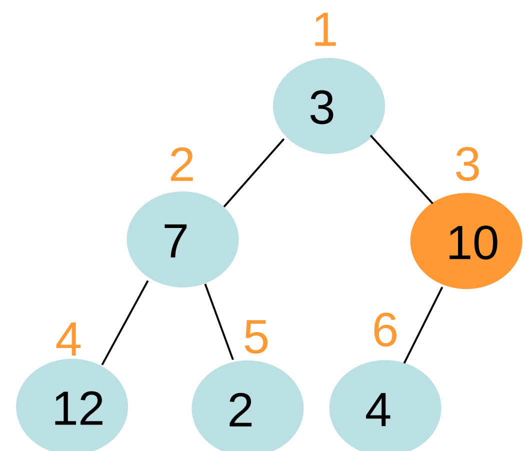
Aufbau eines Heaps

1	2	3	4	5	6
3	7	10	12	2	4
		i			

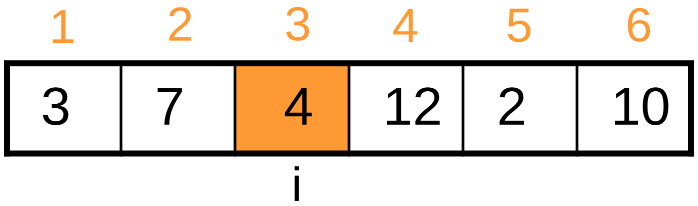
Aufbau eines Heaps:

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

```
BuildHeap(A)
n ← heap-size[A]
for i ←⌊n/2⌋ downto 1 do
      HeapifyDown(A,i)
```



Aufbau eines Heaps

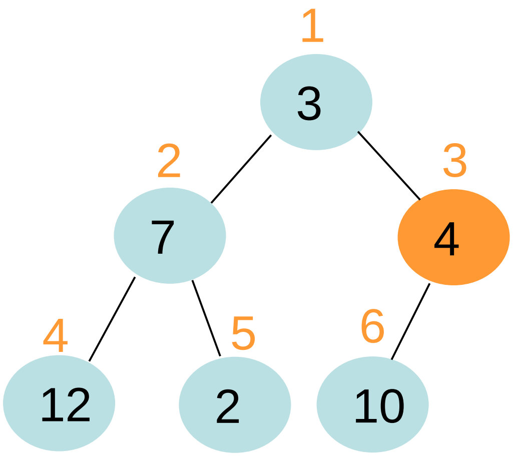


Aufbau eines Heaps:

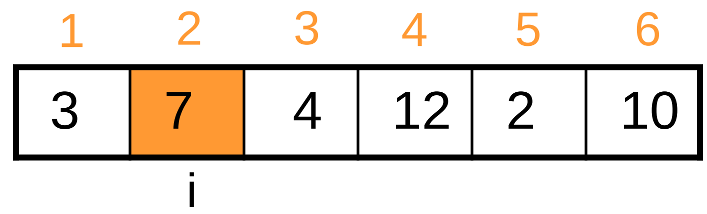
- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

BuildHeap(A)

1. n $\leftarrow$ heap-size[A]
2. **for** i $\leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
3. HeapifyDown(A,i)



Aufbau eines Heaps

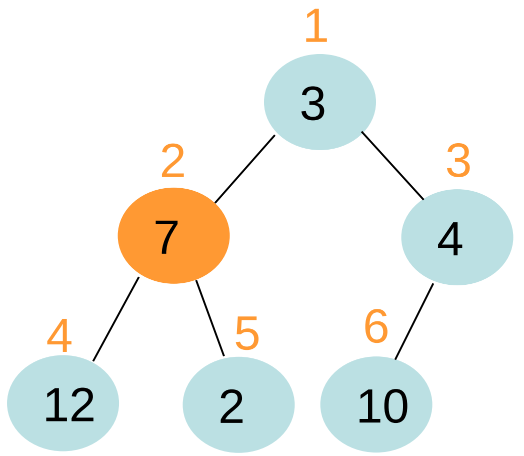


Aufbau eines Heaps:

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

BuildHeap(A)

1. n $\leftarrow$ heap-size[A]
2. **for** i $\leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
3. HeapifyDown(A,i)



Aufbau eines Heaps

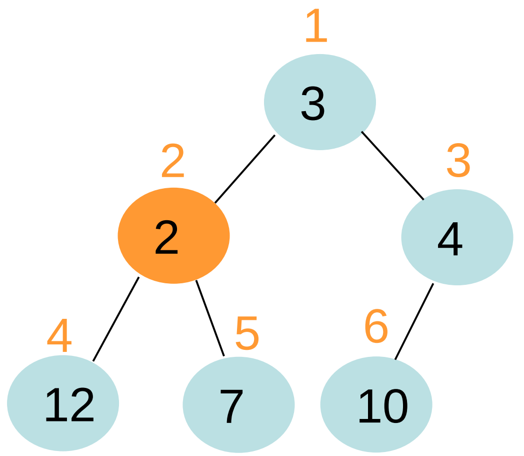
1	2	3	4	5	6
3	2	4	12	7	10
	i				

Aufbau eines Heaps:

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

BuildHeap(A)

1. n $\leftarrow$ heap-size[A]
2. **for** i $\leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
3. HeapifyDown(A,i)



Aufbau eines Heaps

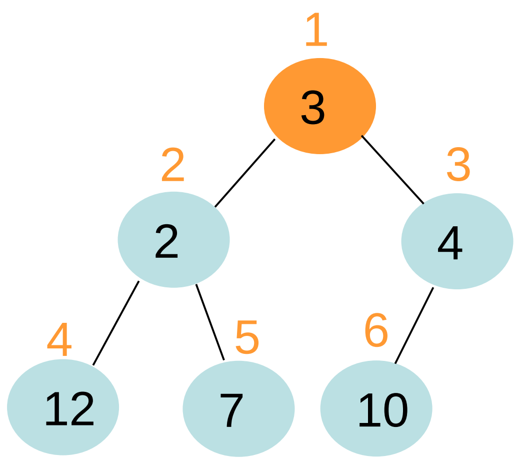
1	2	3	4	5	6
3	2	4	12	7	10
i					

Aufbau eines Heaps:

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

BuildHeap(A)

1. n $\leftarrow$ heap-size[A]
2. **for** i $\leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
3. HeapifyDown(A,i)



Aufbau eines Heaps

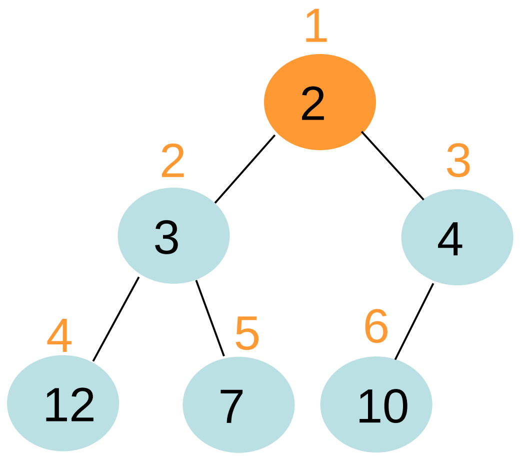
1	2	3	4	5	6
2	3	4	12	7	10
i					

Aufbau eines Heaps:

- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

BuildHeap(A)

1. n ← heap-size[A]
2. **for** i ←⌊n/2⌋ **downto** 1 **do**
3. HeapifyDown(A,i)



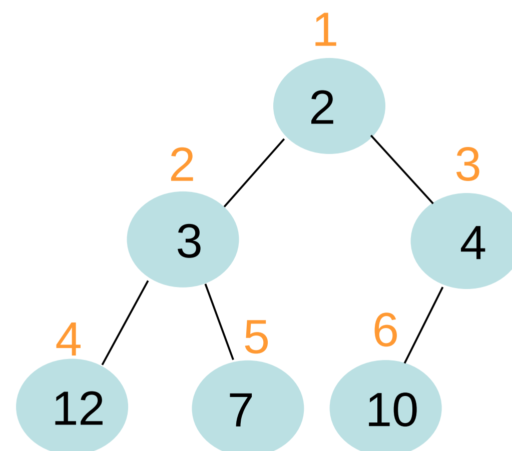
Aufbau eines Heaps

1	2	3	4	5	6
2	3	4	12	7	10

Aufbau eines Heaps:

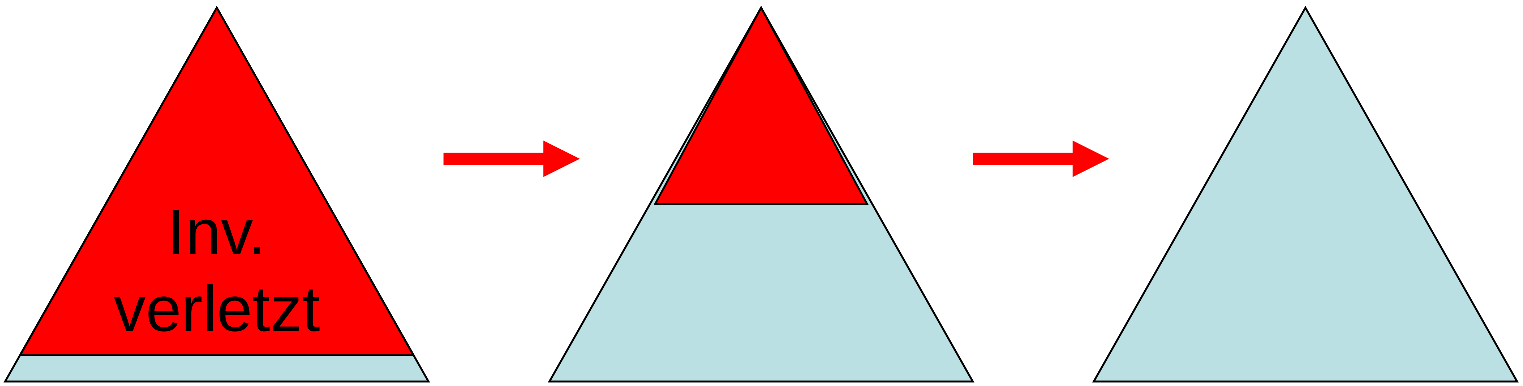
- Jedes Blatt ist ein Heap
- Baue Heap „von unten nach oben“ mit HeapifyDown auf

```
BuildHeap(A)
n ← heap-size[A]
for i ← ⌊n/2⌋ downto 1 do
      HeapifyDown(A,i)
```



Aufbau eines Heaps

HeapifyDown(A,i) für $i=\lfloor n/2 \rfloor$ runter bis 1:



Invariante I(i): $\forall$ j>i: A[j] min. für Teilbaum von A[j]

Heap

Laufzeiten:

- BuildHeap(A): O(n)
- insert(A,e): O(log n)
- min(A): O(1)
- deleteMin(A): O(log n)

Verwendung von max-Heap in Max-Sort ergibt das Sortierverfahren Heapsort

Heapsort

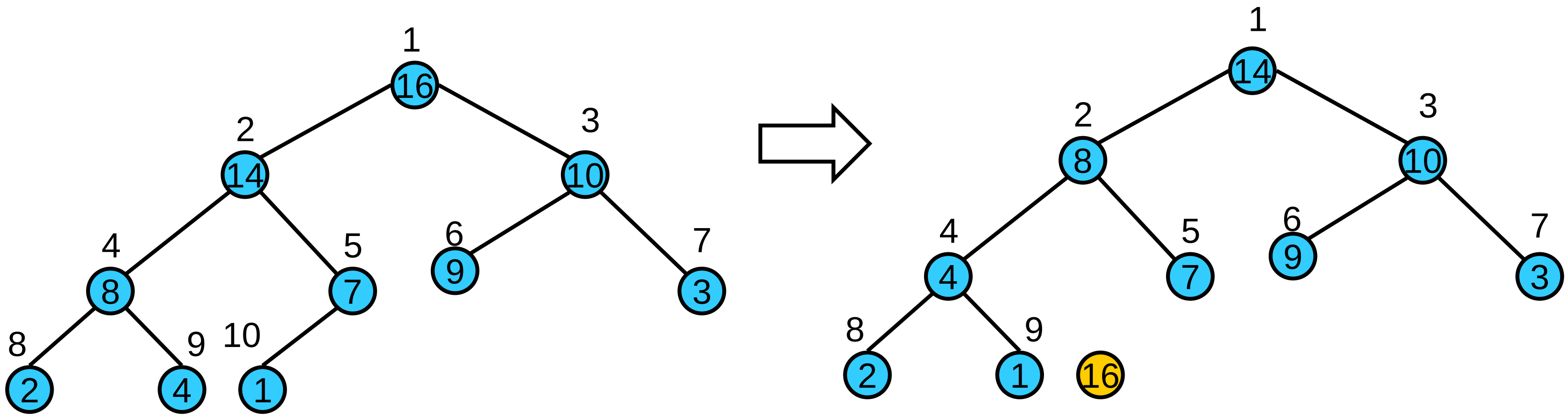
Eingabe: Array *A*

Ausgabe: Zahlen in *A* in aufsteigender Reihenfolge sortiert.

Heapsort(A):
  Build-Max-Heap(A) // wie BuildHeap, aber für max-Heap
  for i←length(A) downto 2 do
    A[i]←DeleteMax(A)   // A[i] ← Maximum in A[1..i]

Korrektheit: folgt aus Korrektheit von Build-Max-Heap(A) und DeleteMax(A) und der Schleifeninvariante
I(i): A[i+1..length(A)] enthält die maximalen Eingabezahlen von A in aufsteigend sortierter Reihenfolge

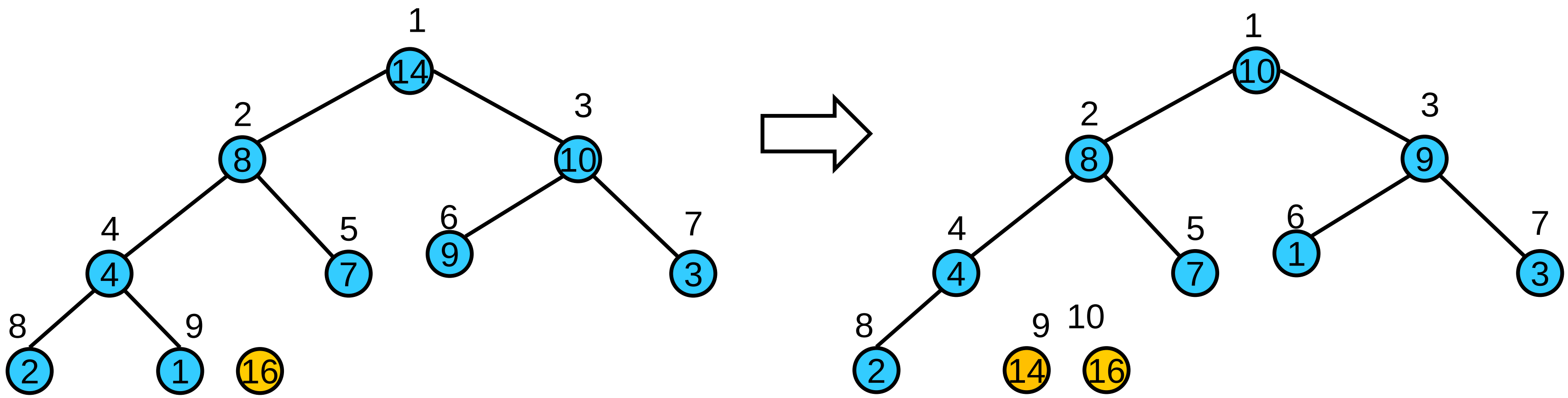
Illustration von Heapsort



16	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**

14	**8**	**10**	**4**	**7**	**9**	**3**	**2**	**1**	**16**

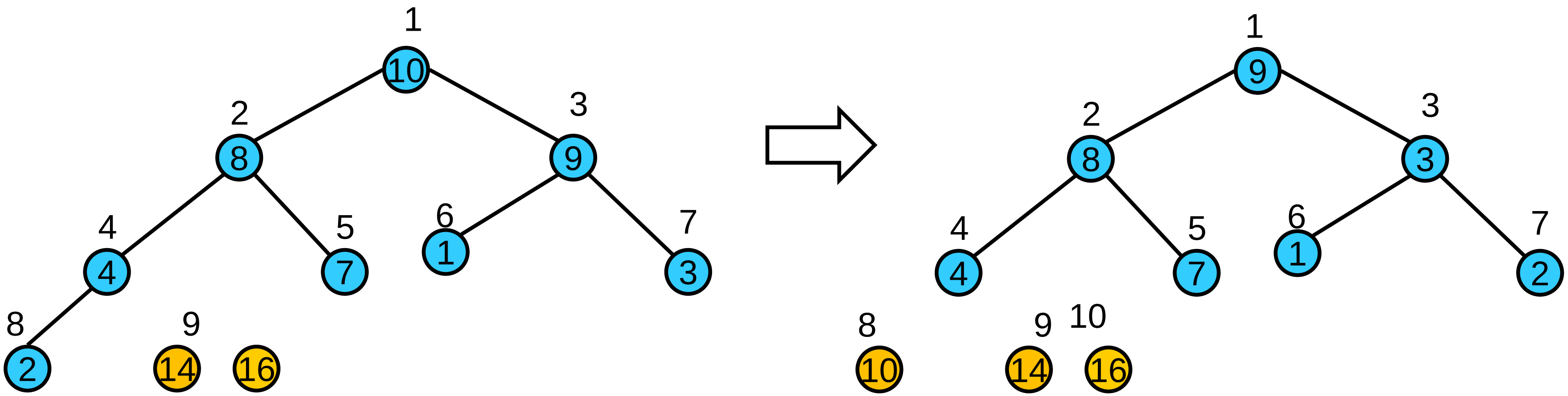
Illustration von Heapsort



14	8	10	4	7	9	3	2	1	16

10	8	9	4	7	9	3	2	14	16

Illustration von Heapsort



10	8	9	4	7	9	3	2	14	16

9	8	3	4	7	1	2	10	14	16

Laufzeit von Heapsort

Satz 7.3: Heapsort besitzt Laufzeit $\mathrm{O}\left(n\log(n)\right)$.

Beweisskizze:

1. Aufruf von Build-Max-Heap: $\mathrm{O}\left(n\right)$.
2. for-Schleife: (n-1)-mal durchlaufen.
3. Pro Durchlauf Laufzeit $\mathrm{O}\left(\log(n)\right)$ (DeleteMax).

Changelog

02.05.16: Folien 2 (neu), 16, 17, 18 (neu), 39

06.05.16: Folien 27, 39, 40-42, 43