Übung Algorithmen I
27.5.15

Christoph Striecks
Christoph.Striecks@kit.edu

(Mit Folien von Julian Arz, Timo Bingmann und Sebastian Schlag.)

Roadmap

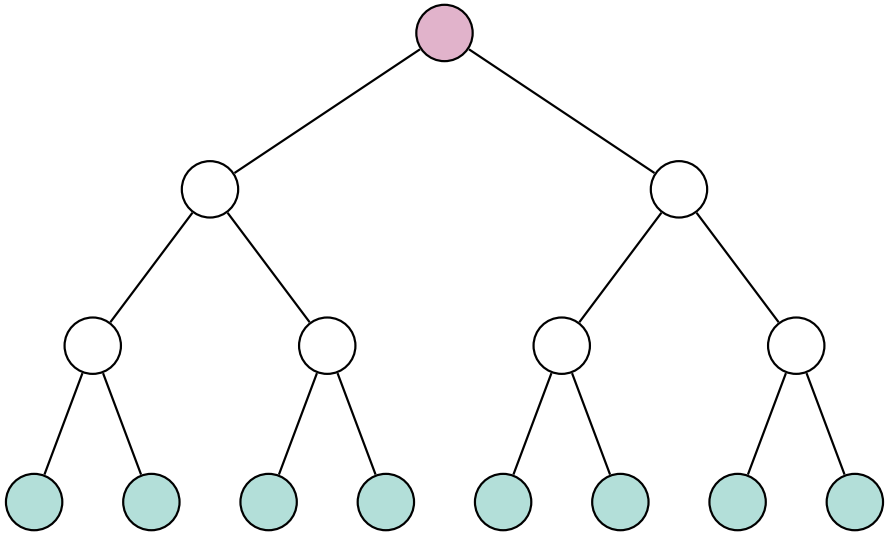
- Organisation
- Perfekte Binärbäume
- Heapsort mit Max-Heap
- Adressierbare binäre Heaps

Organisation

- Heute kein Übungsblatt, nächstes Blatt am 3.6.15
- Übungsklausur am Montag, dem 8.6.15, um 15.45, im Audimax

Perfekte Binärbäume

Perfekter Binärbaum



Kennzahlen:

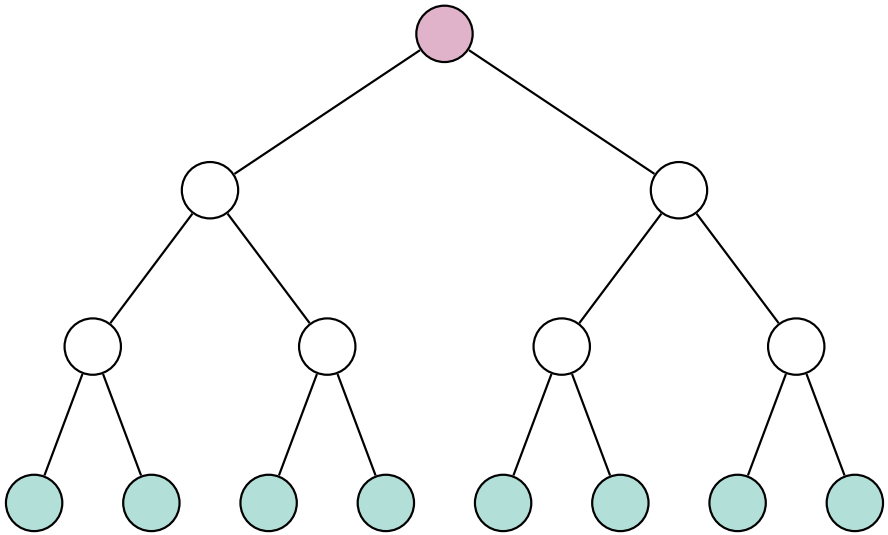
Knoten n

Blätter b

Innere Knoten c

Höhe h

Perfekter Binärbaum

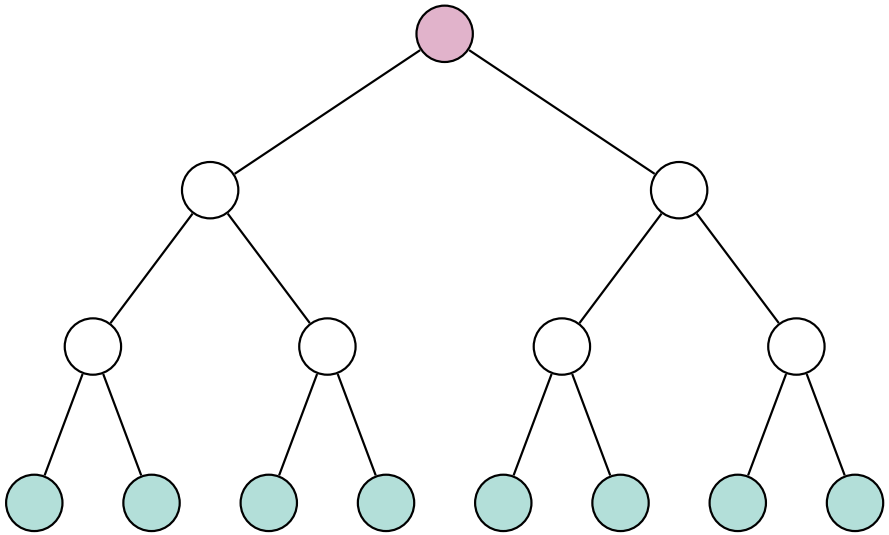


Beispiel:

$n = 15$ Knoten
$b = 8$ Blätter
$c = 7$ innere Knoten
$h = 3$ Höhe

Definition: Die Höhe $h(v)$ eines Knotens v ist die Anzahl der Kanten auf dem eindeutig bestimmten Pfad von der Wurzel zu v.

Perfekter Binärbaum



Beispiel:

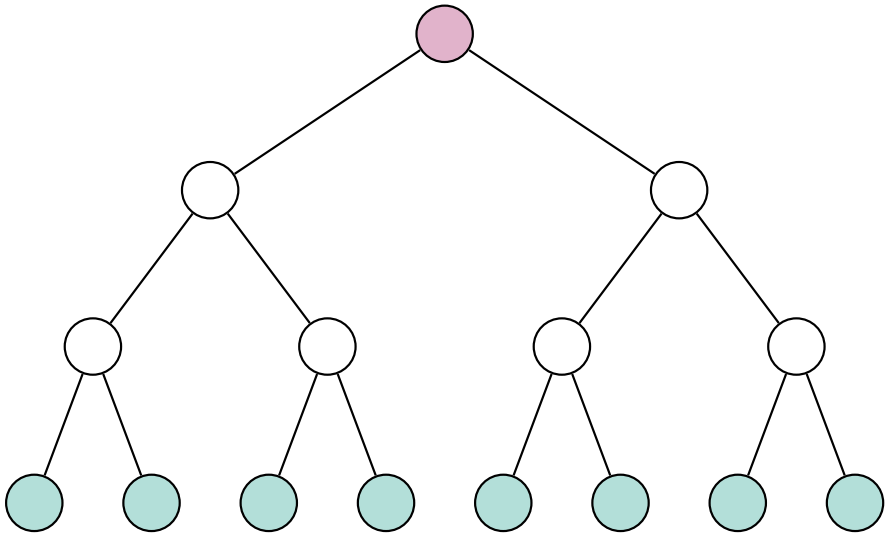
$n = 15$ Knoten
$b = 8$ Blätter
$c = 7$ innere Knoten
$h = 3$ Höhe

Definition: Die Höhe h eines Baums ist die maximale Pfadlänge von der Wurzel zu einem Blatt.

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$h = \lfloor \log_2 n \rfloor$$

Perfekter Binärbaum



Beispiel:

$n = 15$ Knoten
$b = 8$ Blätter
$c = 7$ innere Knoten
$h = 3$ Höhe

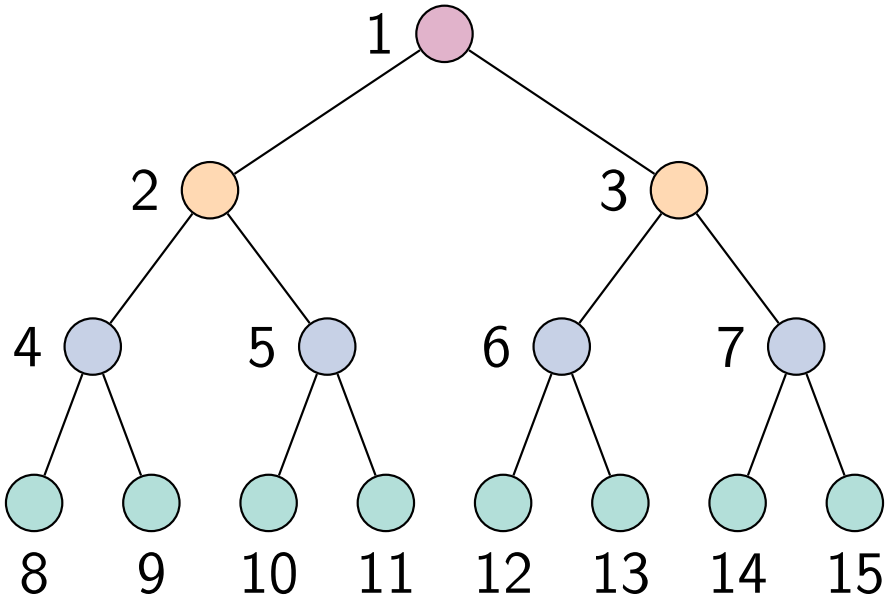
Satz: Die Anzahl aller Knoten n ist nur linear abhängig von der Anzahl der Blätter b.

$$b = 2^h$$

$$n = b + c = 2^h + (2^h - 1)$$

$$n = \boxed{2b - 1} = \boxed{2c + 1}$$

Perfekter Binärbaum



Beispiel:

$n = 15$ Knoten

$b = 8$ Blätter

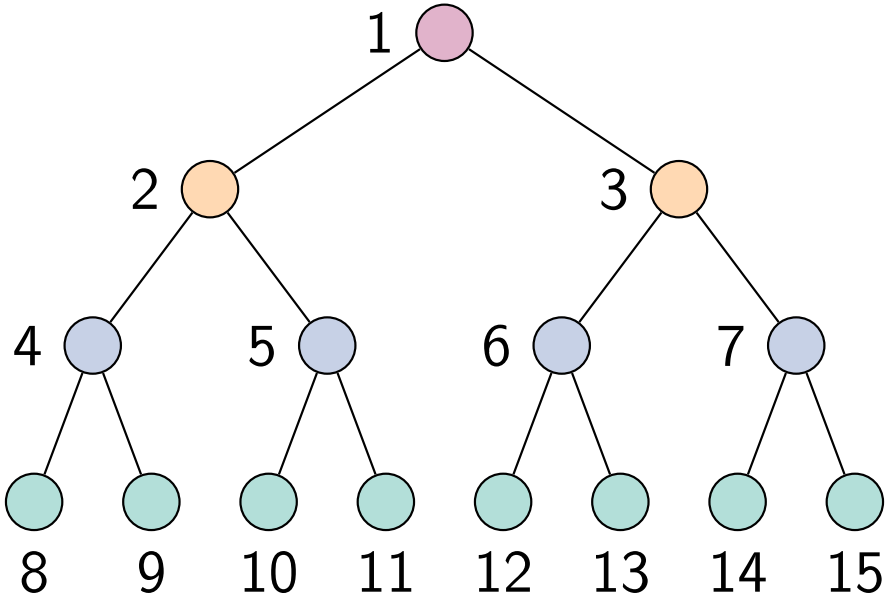
$c = 7$ innere Knoten

$h = 3$ Höhe

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

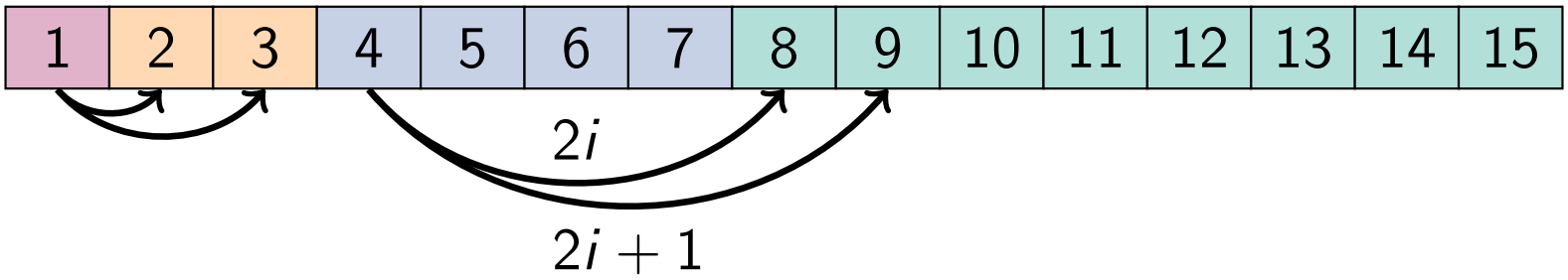
(Level-order-Traversierung.)

Perfekter Binärbaum



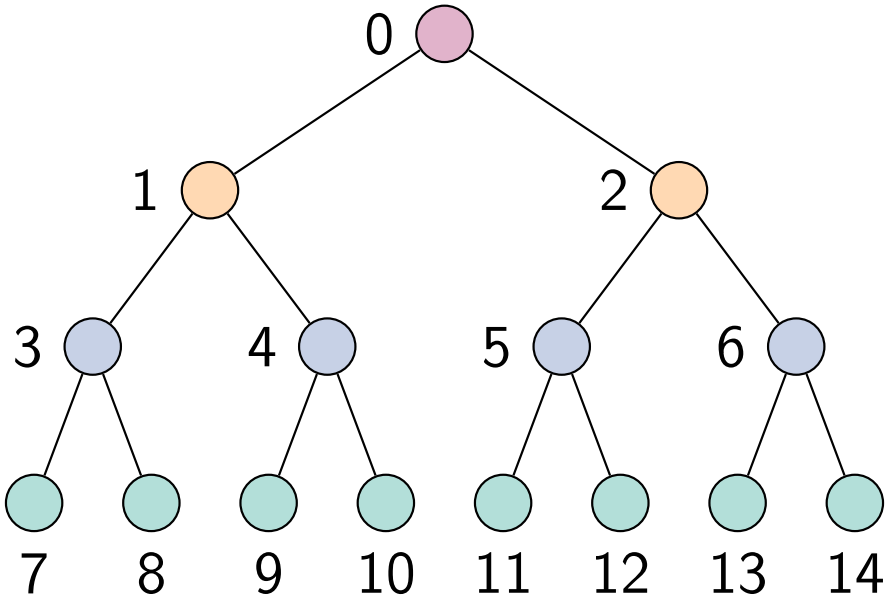
Beispiel:

$n = 15$ Knoten
$b = 8$ Blätter
$c = 7$ innere Knoten
$h = 3$ Höhe



(Level-order-Traversierung.)

Perfekter Binärbaum



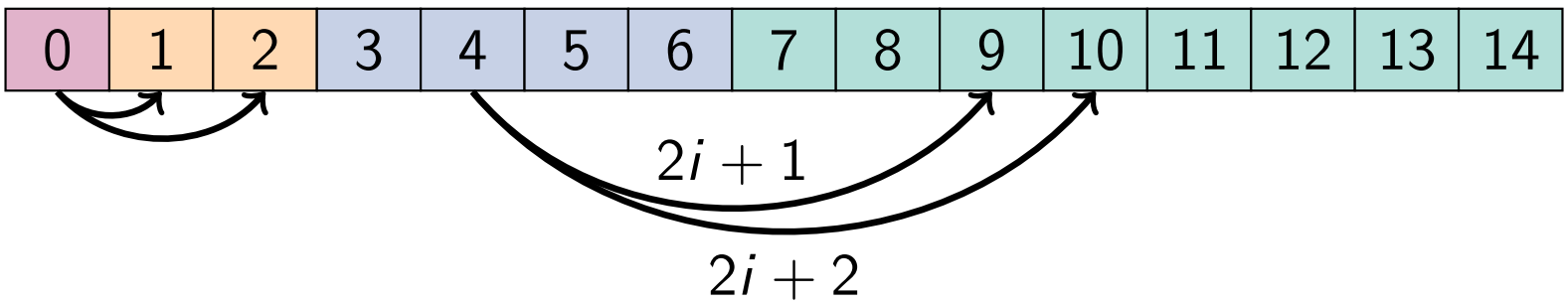
Beispiel:

$n = 15$ Knoten

$b = 8$ Blätter

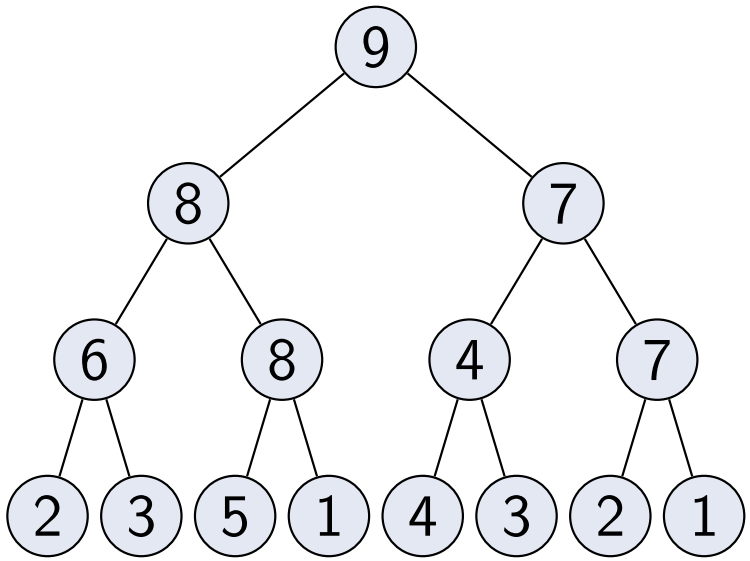
$c = 7$ innere Knoten

$h = 3$ Höhe



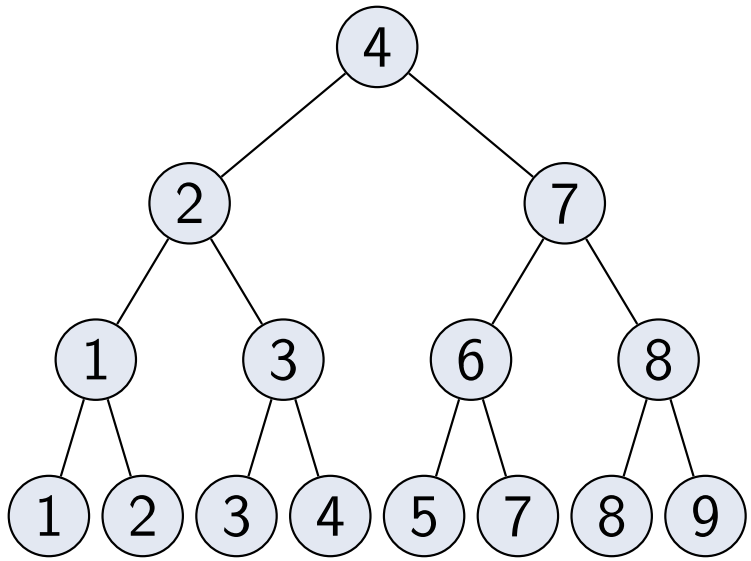
(Max-)Heap und Suchbaum

Max-Heap



Invariante: Für alle Knoten v gilt $\mathit{parent}(v) \geq v$.

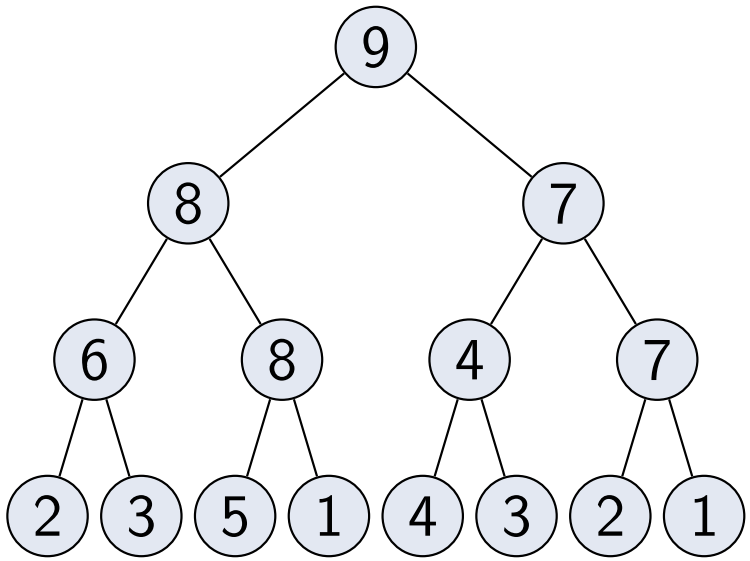
Suchbaum



Invariante: Für alle Knoten v gilt $\forall w \in \mathit{left}(v) : w \leq v$ und $\forall w \in \mathit{right}(v) : w > v$.

(Max-)Heap und Suchbaum

Max-Heap



Invariante: Für alle Knoten v gilt $parent(v) \geq v$.

Priority-Queue-Operationen:

insert(v)

deleteMax(v)

Heap-Operationen:

siftUp(v)

siftDown(v)

buildHeap$(h[1..n])$

Heapsort mit Max-Heap

Heapsort

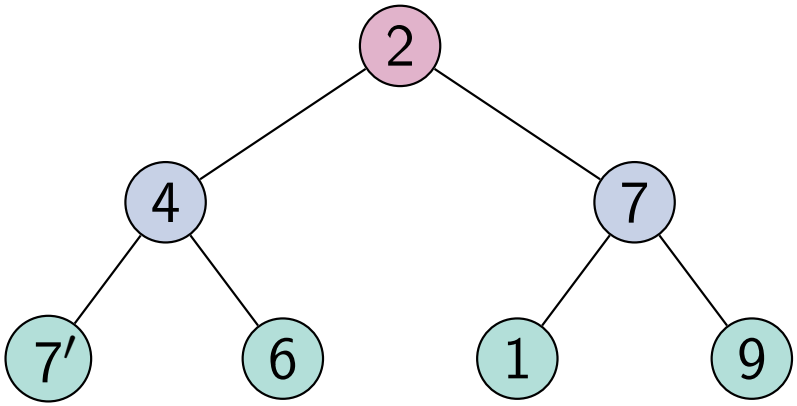
Procedure selectionSort(a : **Array of** $\mathbb{R}$)
  for $i := 0$ **to** $n-1$ **do**
    $min := i$
    for $j := i+1$ **to** $n-1$ **do**
      if $A[j] < A[min]$ **then** $min := j$
    swap($A[i], A[min]$)

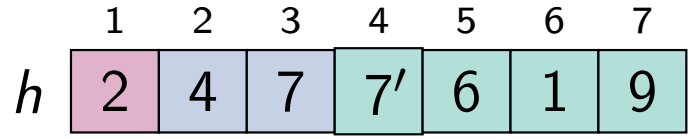
- Laufzeit: n Iterationen, pro Iteration $O(n) \Rightarrow O(n^2)$
- Operationen: sucheMinimum in $O(n)$, entferneMinimum in $O(1)$

Wem kommen diese Operationen bekannt vor?
$\Rightarrow$ deleteMin der Priority Queue, Zeit: $O(\log n)$

- Heap als bessere Datenstruktur für Selectionsort
- $O(n^2)$ vs. $O(n \log n)$

Heapsort mit Max-Heap

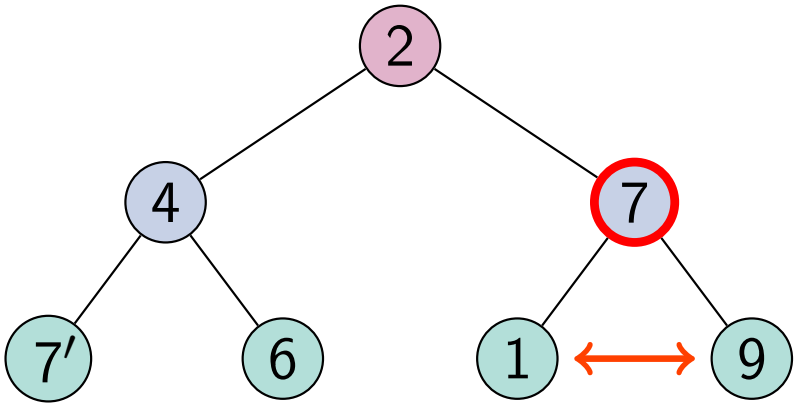


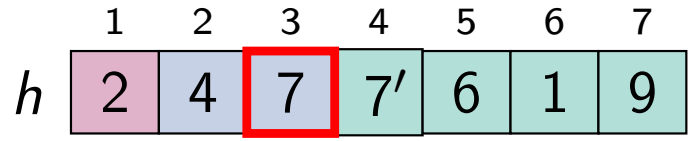


Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Heapsort mit Max-Heap



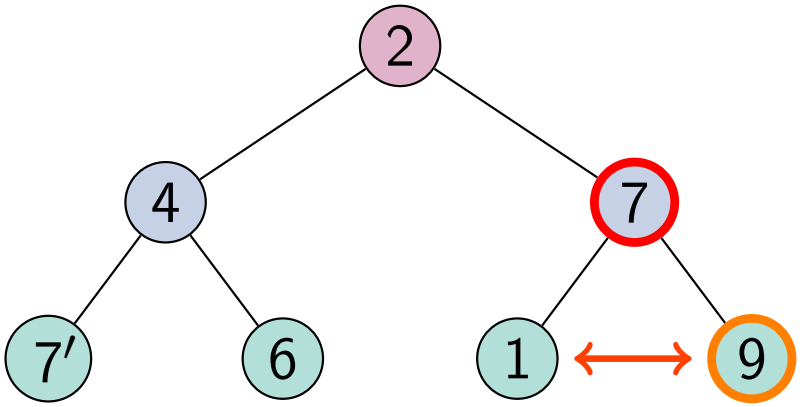


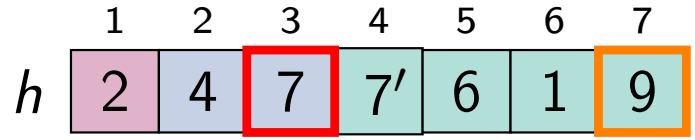
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(3)

Heapsort mit Max-Heap



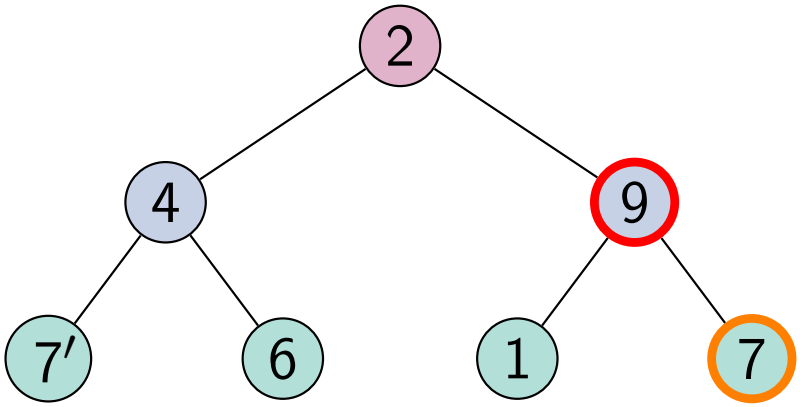


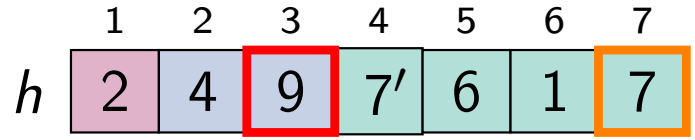
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(3)

Heapsort mit Max-Heap



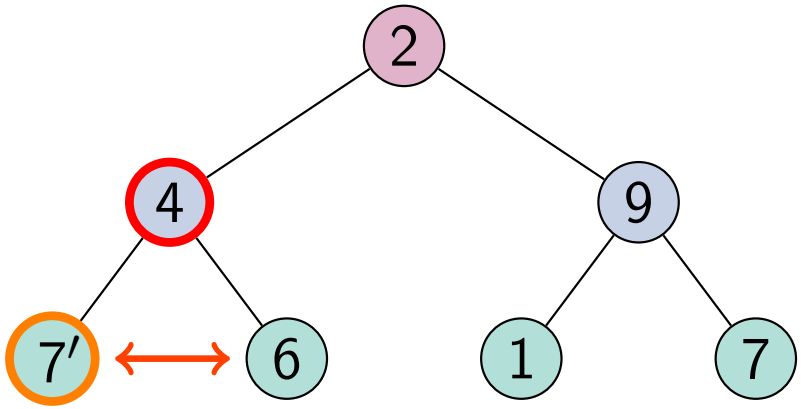


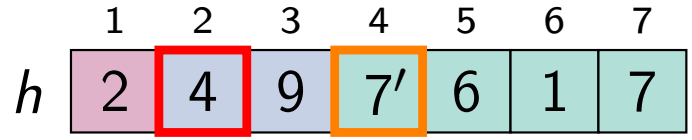
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(3) →
swap(3,7)

Heapsort mit Max-Heap



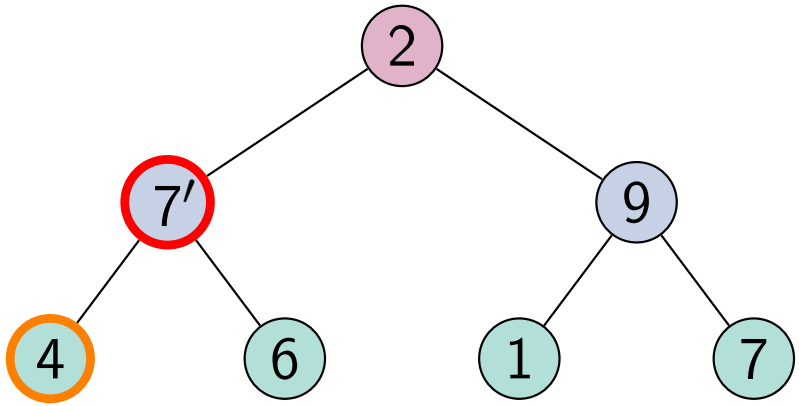


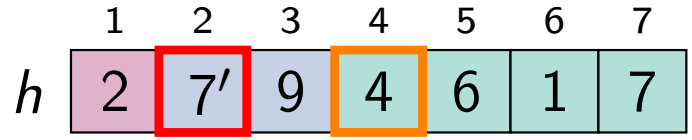
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(2)

Heapsort mit Max-Heap



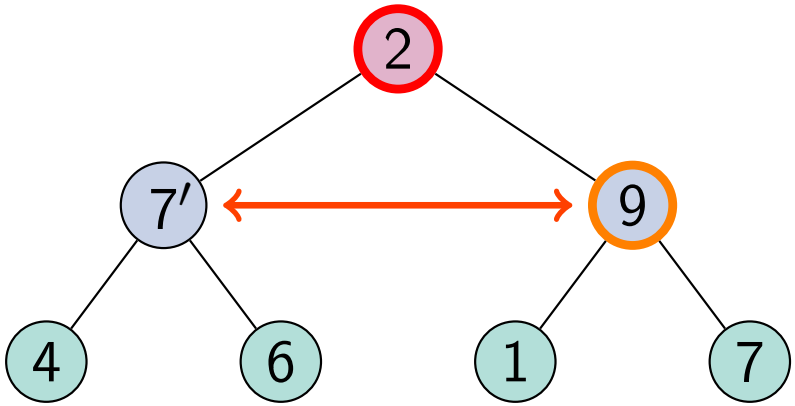


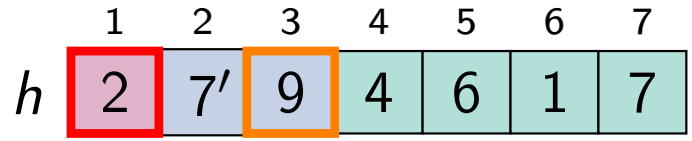
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(2) →
swap(2,4)

Heapsort mit Max-Heap



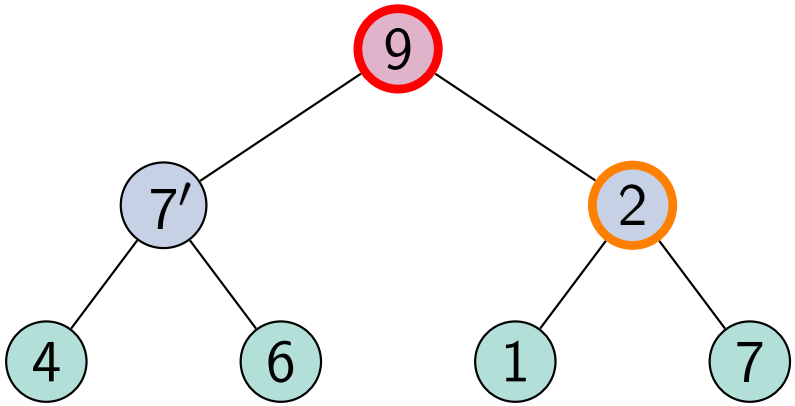


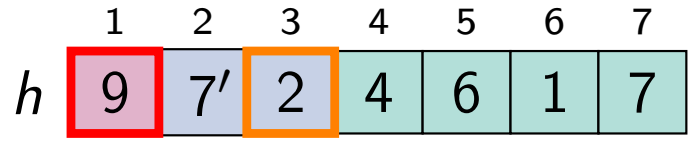
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(1)

Heapsort mit Max-Heap



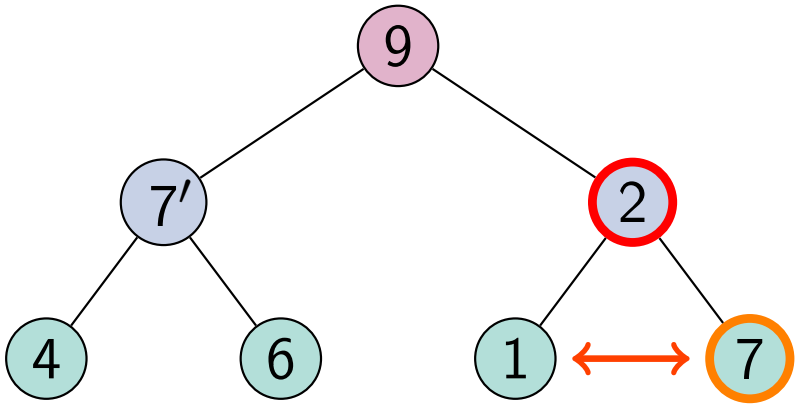


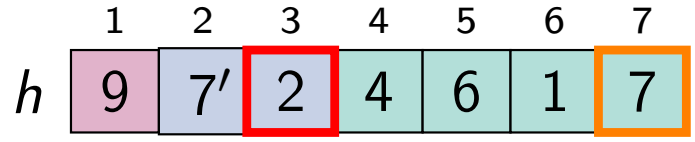
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(1) →
swap(1,3)

Heapsort mit Max-Heap



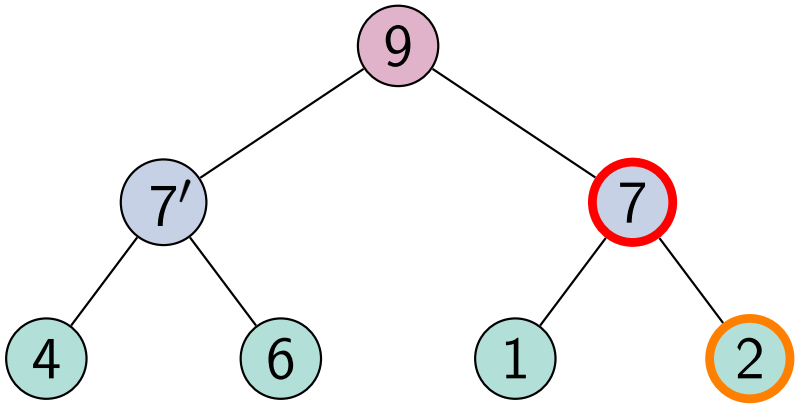


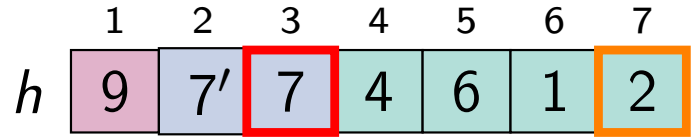
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(1) →
swap(1,3)
siftDown(3)

Heapsort mit Max-Heap



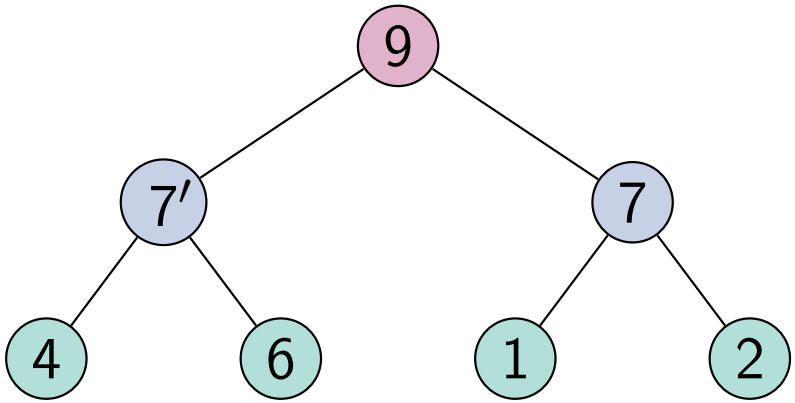


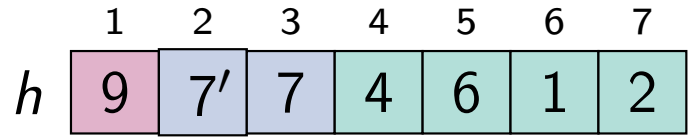
Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

siftDown(1) →
swap(1,3)
siftDown(3) →
swap(3,7)

Heapsort mit Max-Heap

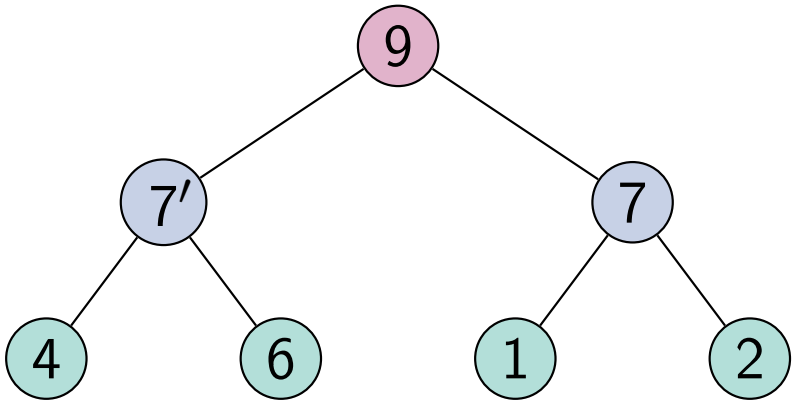




Procedure buildHeap($h[1..n]$)
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do**
    siftDown(i)

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Heapsort mit Max-Heap

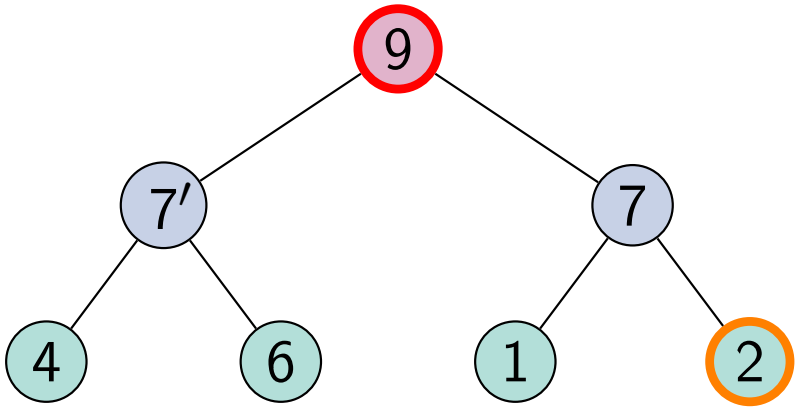


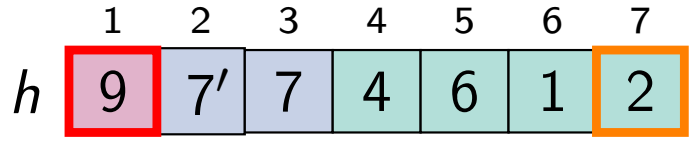
	1	2	3	4	5	6	7
h	9	$7'$	7	4	6	1	2

Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

Heapsort mit Max-Heap



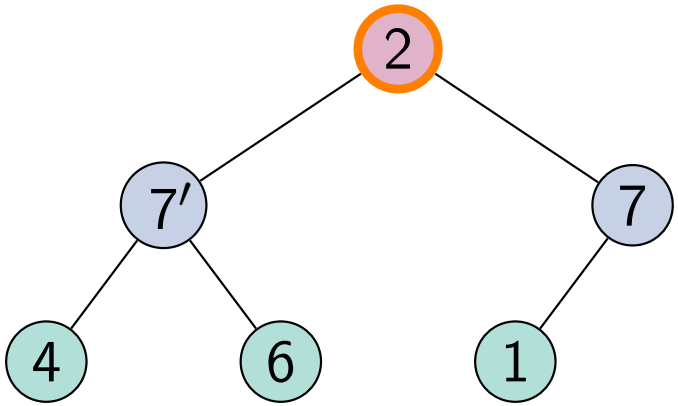


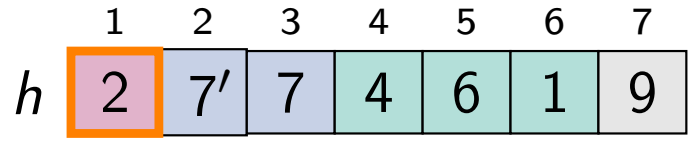
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,7)

Heapsort mit Max-Heap



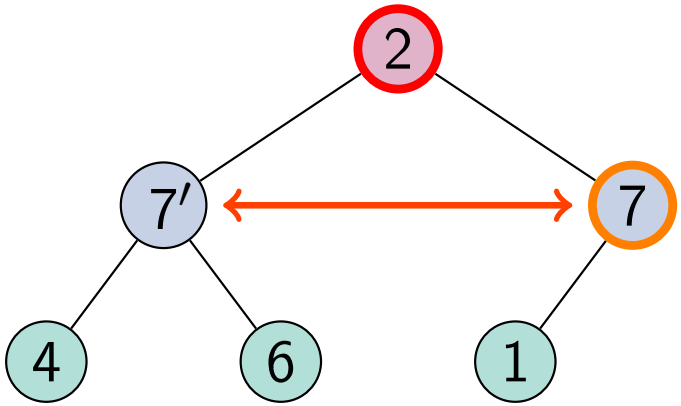


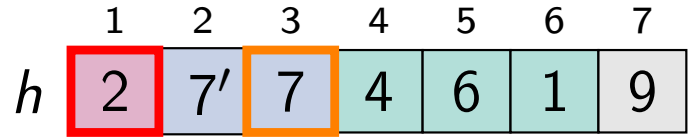
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,7)

Heapsort mit Max-Heap



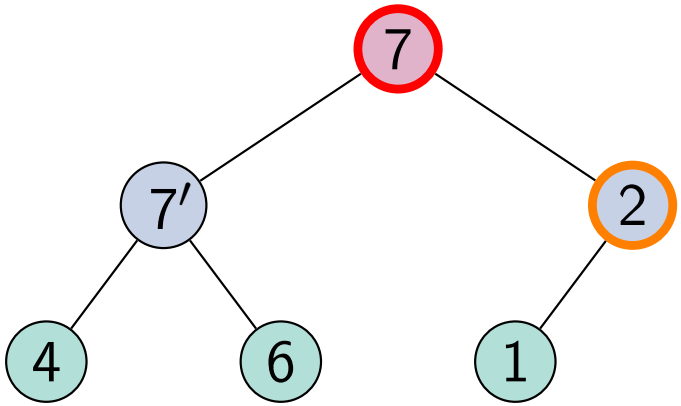


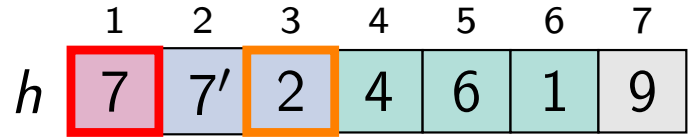
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,7)
siftDown(1) $\rightarrow$
swap(1,2)

Heapsort mit Max-Heap



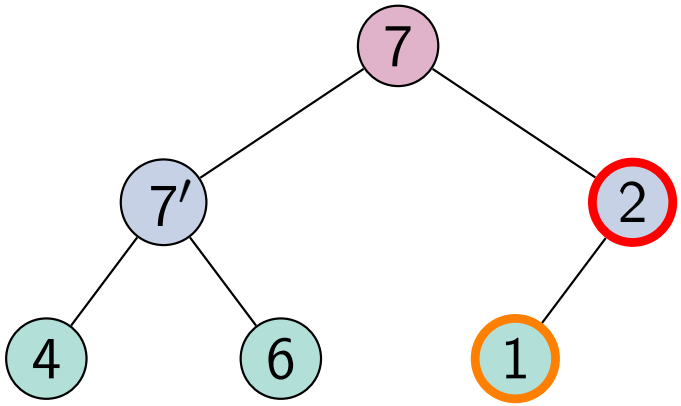


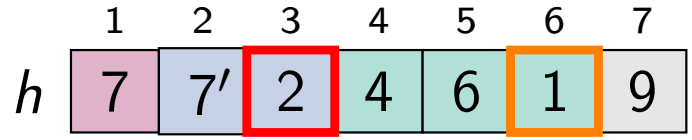
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,7)
siftDown(1) $\rightarrow$
swap(1,3)
siftDown(3)

Heapsort mit Max-Heap



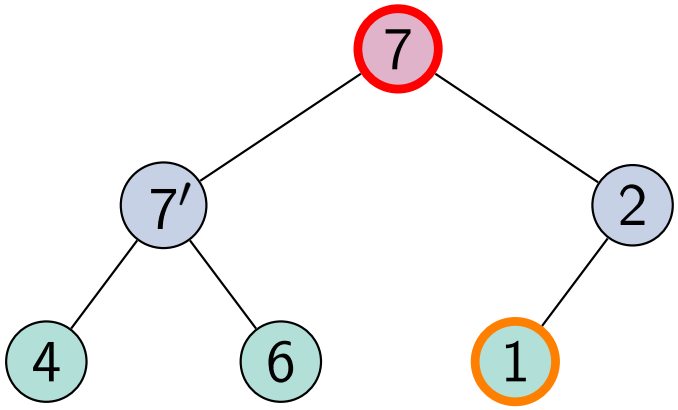


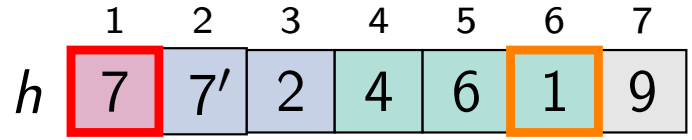
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,7)
siftDown(1) $\rightarrow$
swap(1,3)
siftDown(3)

Heapsort mit Max-Heap



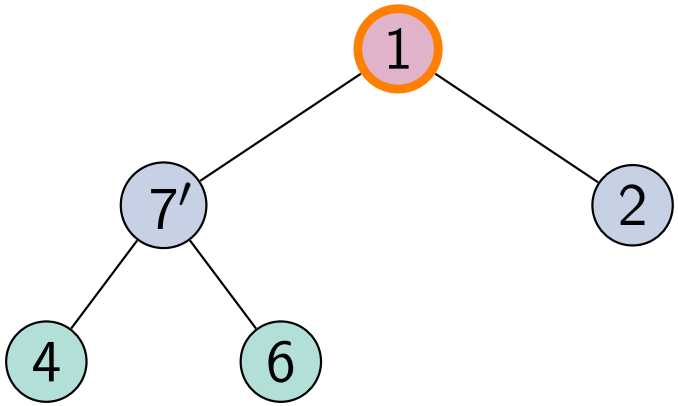


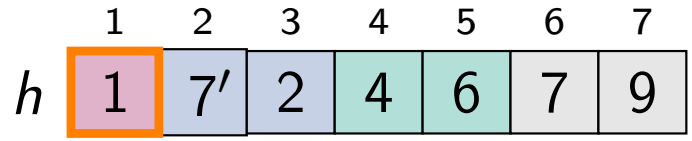
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,6)

Heapsort mit Max-Heap



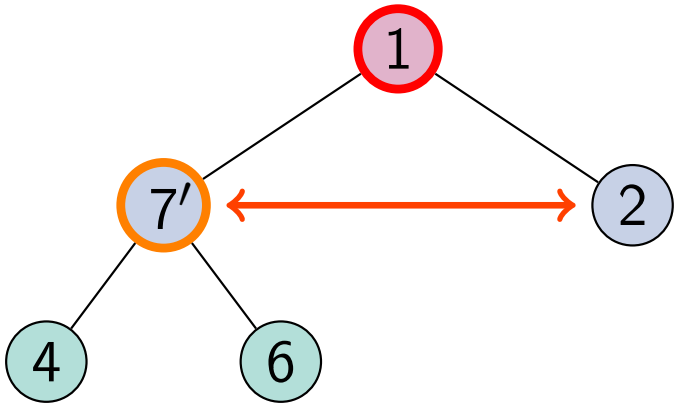


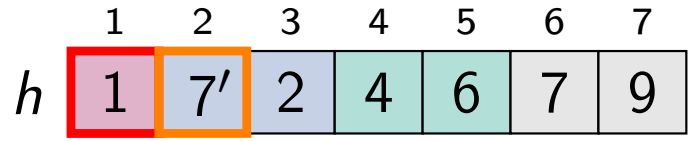
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,6)

Heapsort mit Max-Heap



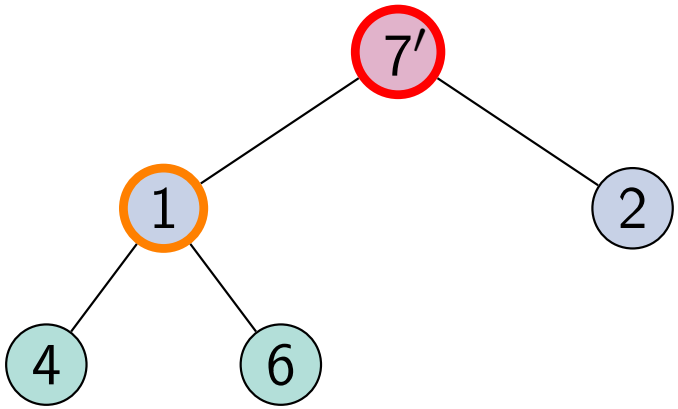


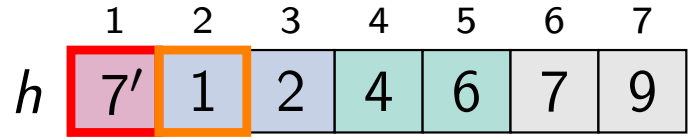
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,6)
siftDown(1) $\rightarrow$
swap(1,2)

Heapsort mit Max-Heap



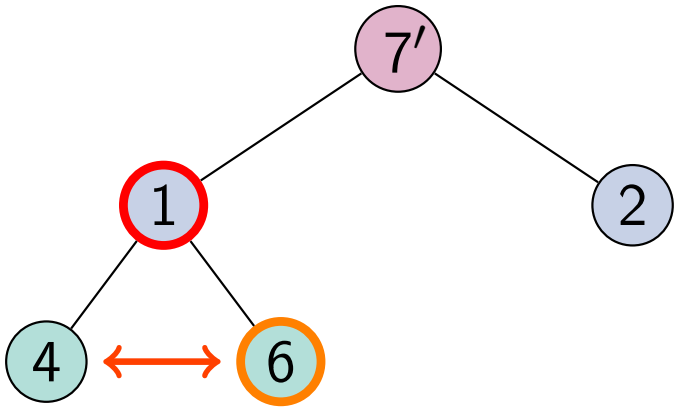


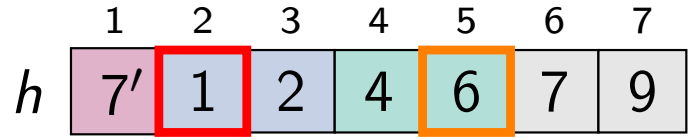
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,6)
siftDown(1) $\rightarrow$
swap(1,2)
siftDown(2)

Heapsort mit Max-Heap



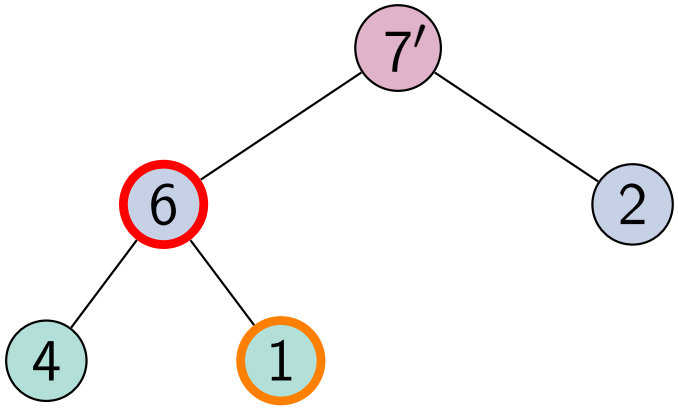


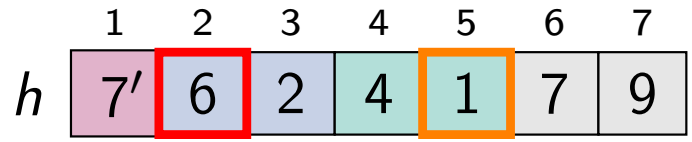
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,6)
siftDown(1) $\rightarrow$
swap(1,2)
siftDown(2)

Heapsort mit Max-Heap



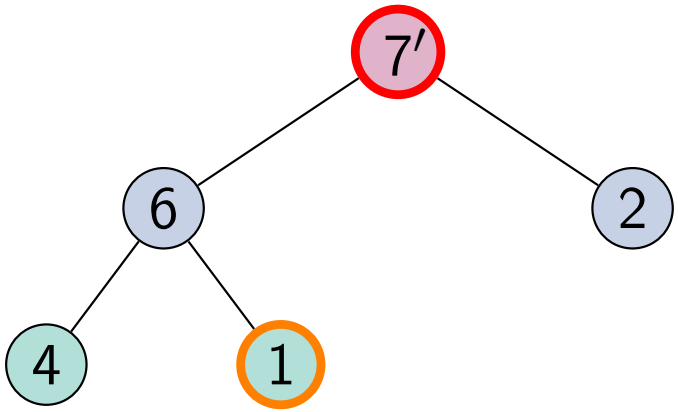


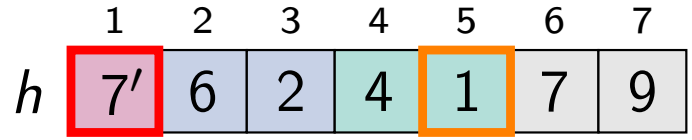
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,6)
siftDown(1) $\rightarrow$
swap(1,2)
siftDown(2) $\rightarrow$
swap(2,5)

Heapsort mit Max-Heap



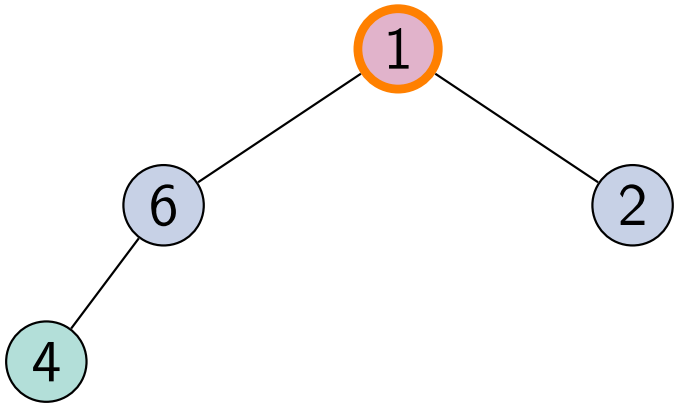


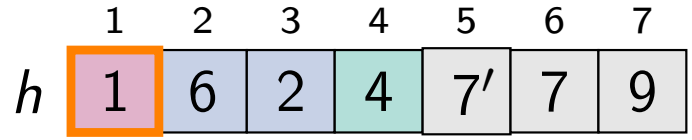
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,5)

Heapsort mit Max-Heap



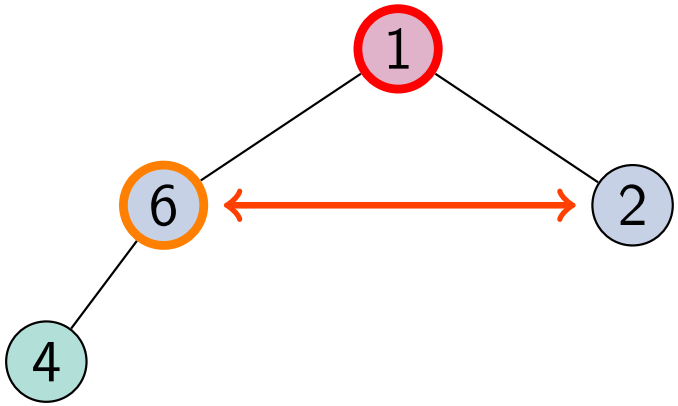


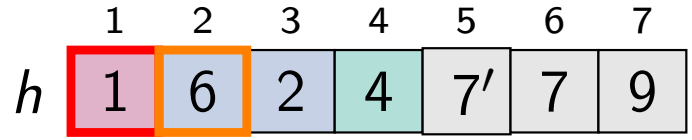
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,5)

Heapsort mit Max-Heap



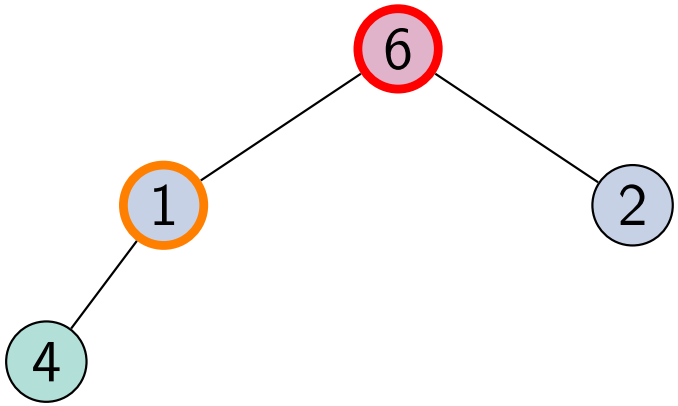


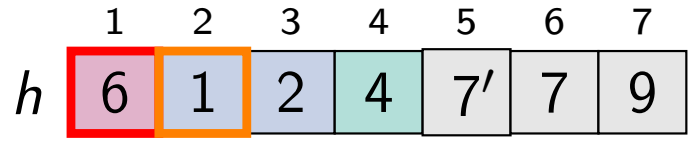
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,5)
siftDown(1) $\rightarrow$
swap(1,2)

Heapsort mit Max-Heap



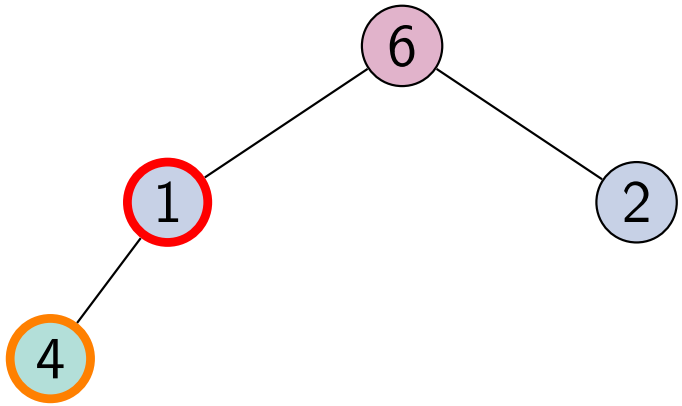


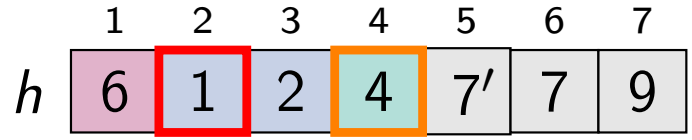
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,5)
siftDown(1) $\rightarrow$
swap(1,2)
siftDown(2)

Heapsort mit Max-Heap



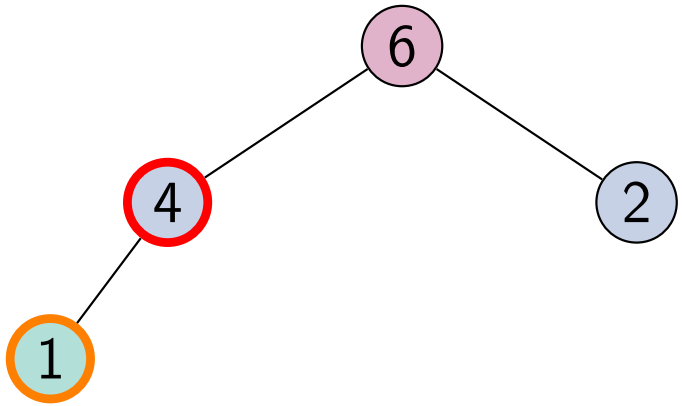


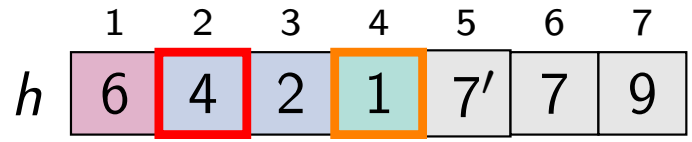
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,6)
siftDown(1) $\rightarrow$
swap(1,2)
siftDown(2)

Heapsort mit Max-Heap



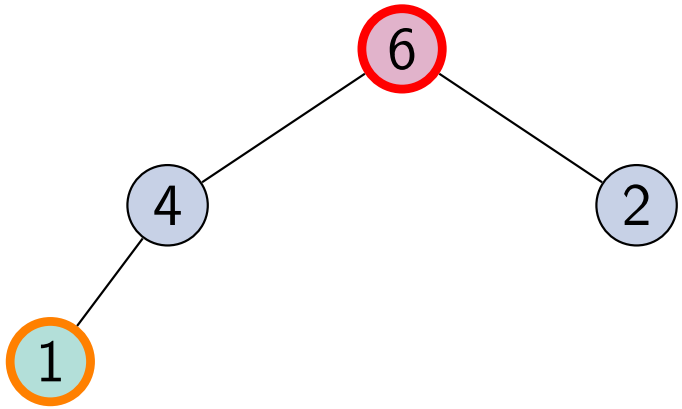


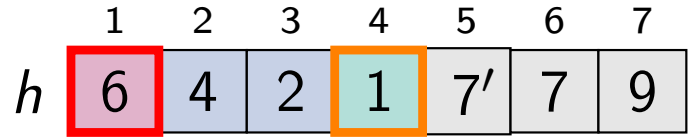
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n $>$ 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,5)
siftDown(1) $\rightarrow$
swap(1,2)
siftDown(2) $\rightarrow$
swap(2,4)

Heapsort mit Max-Heap



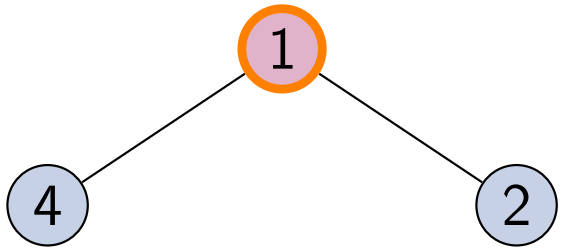


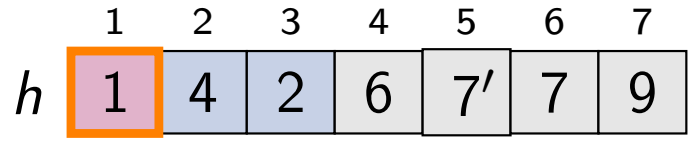
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,4)

Heapsort mit Max-Heap



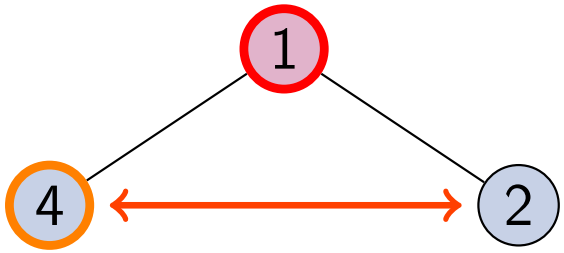


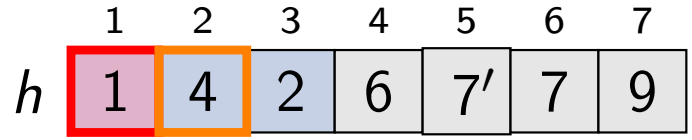
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,4)

Heapsort mit Max-Heap



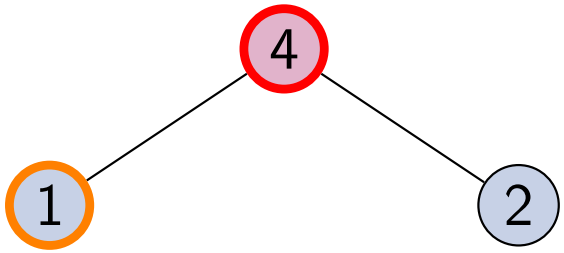


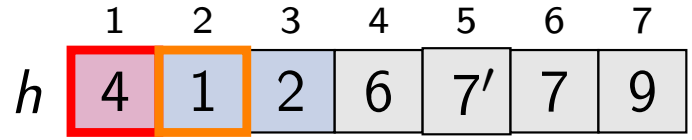
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,4)
siftDown(1) $\rightarrow$
swap(1,2)

Heapsort mit Max-Heap



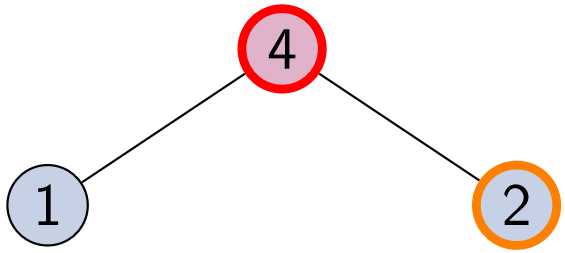


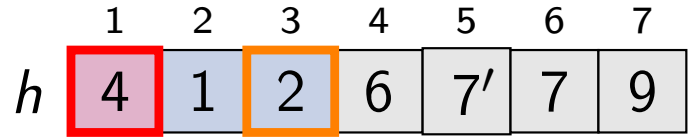
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,4)
siftDown(1) $\rightarrow$
swap(1,2)
siftDown(2)

Heapsort mit Max-Heap



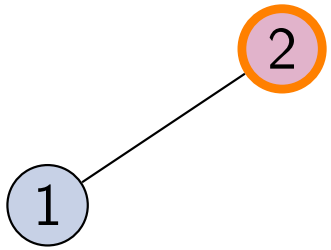


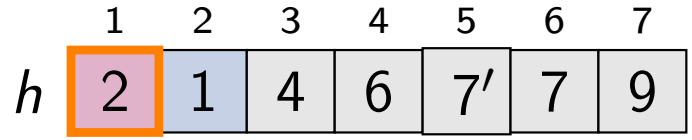
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,3)

Heapsort mit Max-Heap



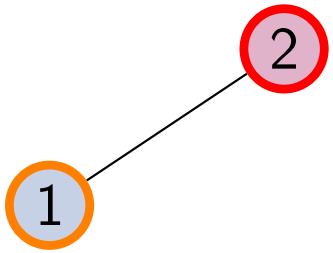


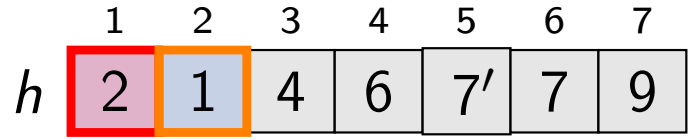
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,3)

Heapsort mit Max-Heap



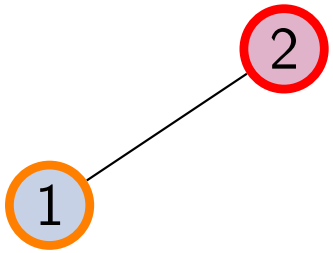


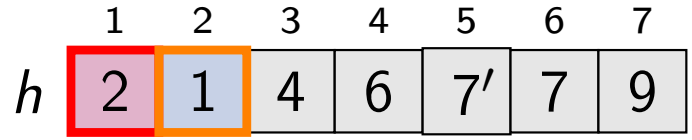
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,3)
siftDown(1)

Heapsort mit Max-Heap





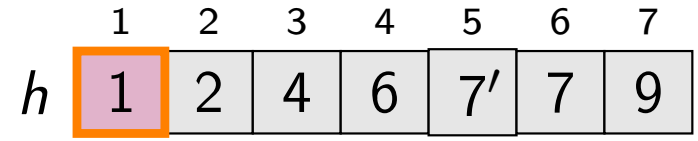
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,2)

Heapsort mit Max-Heap

1



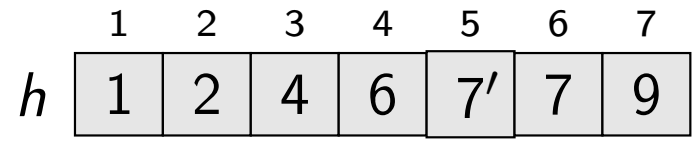
Procedure siftDown($i : \mathbb{N}$)
  if $2i > n$ **then return**
  $k := 2i$
  if $2i + 1 \leq n \wedge h[2i] \leq h[2i + 1]$ **then**
    $k := k + 1$
  if $h[i] < h[k]$ **then**
    swap($h[i], h[k]$)
    siftDown(k)

Procedure heapSort($h[1..n]$)
  buildHeap(h)
  while n > 1
    swap($h[1], h[n]$)
    $n := n - 1$
    siftDown(1)

swap(1,2)

Heapsort mit Max-Heap

1




```
Procedure siftDown(i : ℕ)
    if 2i > n then return
    k := 2i
    if 2i + 1 ≤ n ∧ h[2i] ≤ h[2i + 1] then
        k := k + 1
    if h[i] < h[k] then
        swap(h[i], h[k])
        siftDown(k)
```

```
Procedure heapSort(h[1..n])
    buildHeap(h)
    while n > 1
        swap(h[1], h[n])
        n := n − 1
        siftDown(1)
```

Adressierbare binäre Heaps

Anwendungsbeispiel

Anwendungsbeispiel

Problem:

- Studierende während Prüfungsvorbereitung
- Pool von 50 Prüfungsaufgaben
- Idee: wähle die einfachste Aufgabe (die noch übrig ist)
- Zusatz: neue Aufgaben kommen hinzu

Verfahren:

- **Initialisierung:**
  - Schätze für alle Aufgaben den Schwierigkeitsgrad
  - Füge in Heap ein (Schwierigkeitsgrad als Schlüssel)
- **While** "noch Aufgaben übrig"
  - Bearbeite nächste Aufgabe (*deleteMin*)
  - Füge neue Aufgaben in Heap ein (*insert*)

Anwendungsbeispiel

Anwendungsbeispiel

Jetzt:

- Studierende bekommen Übung in manchen Aufgabenarten
- Dadurch werden diese leichter
- Außerdem: Aufgabentyp X kommt nicht mehr vor
- Diese müssen nachträglich entfernt werden

Verfahren:

- Wie vorher: Schätze Schwierigkeitsgrad und füge in Heap ein
- **While** noch Aufgaben übrig
  - Bearbeite nächste Aufgabe (*deleteMin*)
  - Füge neue Aufgaben in Heap ein (*insert*)
  - Veringere Schwierigkeitsgrad mancher Aufgaben (decreaseKey)
  - Lösche manche Aufgaben (remove)

Adressierbare binäre Heaps

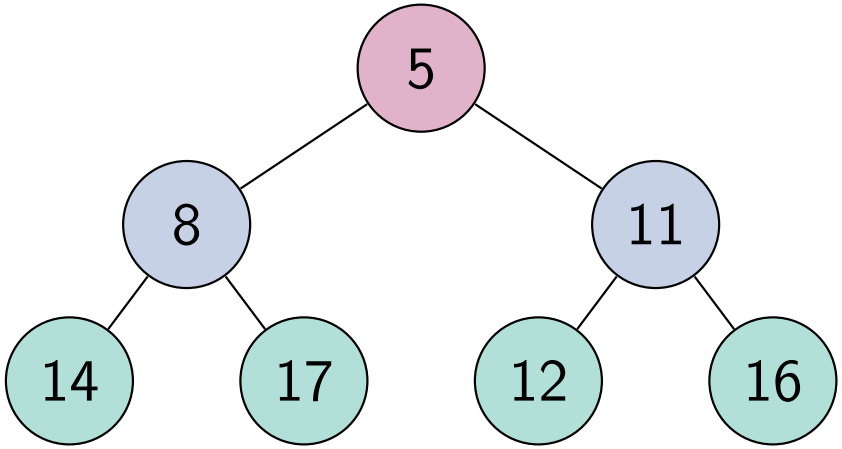
Idee:

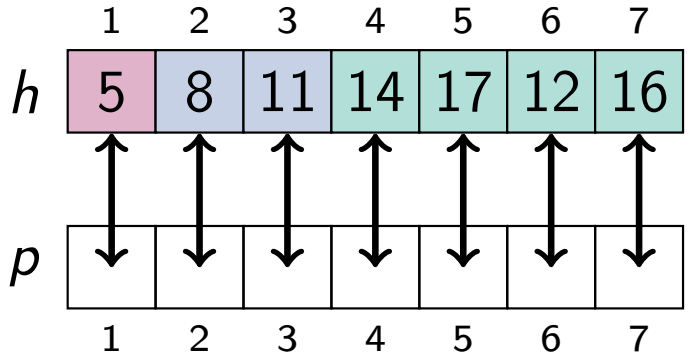
- Proxy-Array p speichert Handles auf Elemente im Heap H
- Ermöglicht Operationen
  - $H.decreaseKey(p[i], k)$ (Schlüssel veringern)
  - $H.remove(p[i], k)$ (Element enfernen)

Problem:

- Heap ist in Bewegung
- $\rightarrow$ Handles müssen beim Tauschen auch vertauscht werden
- $\rightarrow$ Elemente im Heap brauchen „Rückpointer“ auf Einträge im Proxy-Array

Adressierbare binäre Heaps

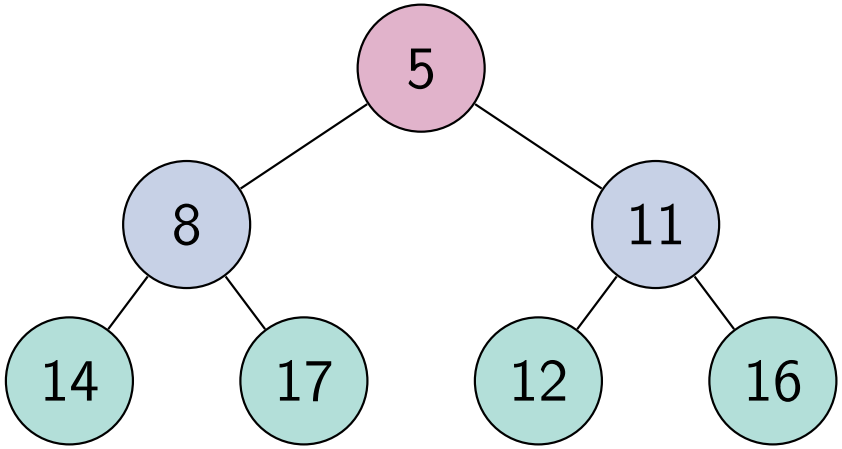


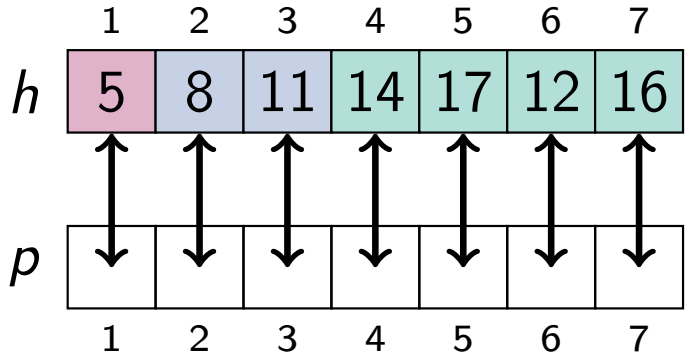


Procedure decreaseKey(h : Handle,
k : Key)
assert $k \leq \text{key}(h)$
key(h):= k
siftUp(h)

Achtung: **invariant** $\text{proxy}(p[i]) = i$

Adressierbare binäre Heaps



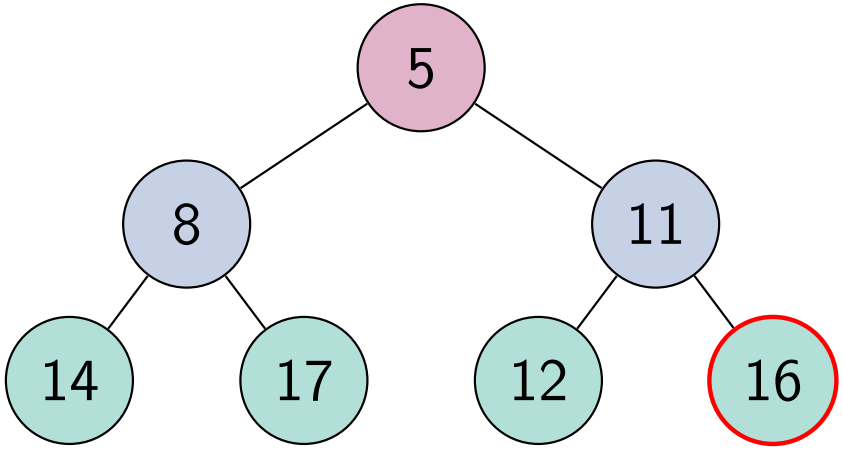


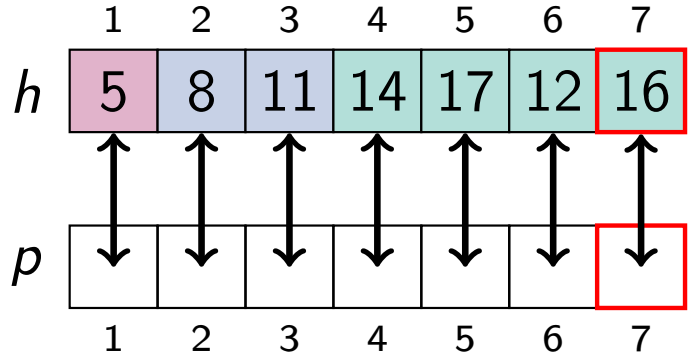
Procedure decreaseKey(h : Handle, k : Key)
assert $k \leq \text{key}(h)$
$\text{key}(h) := k$
siftUp(h)

Operation:
decreaseKey$(p[7], 6)$

Achtung: **invariant** $\text{proxy}(p[i]) = i$

Adressierbare binäre Heaps



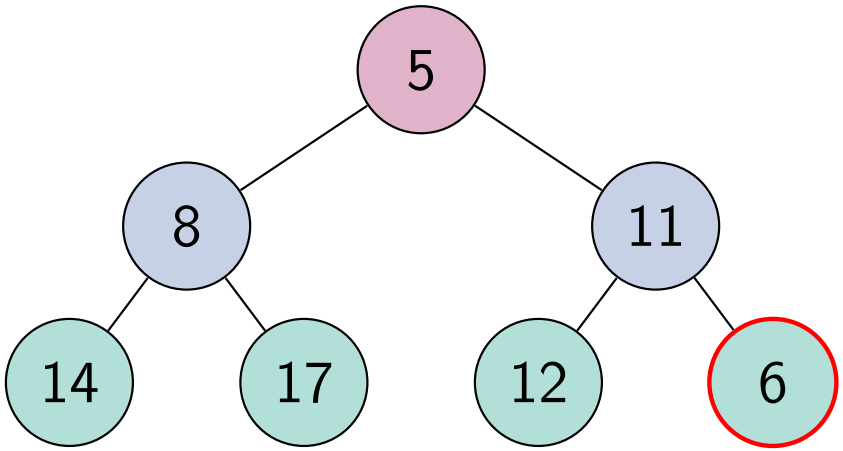


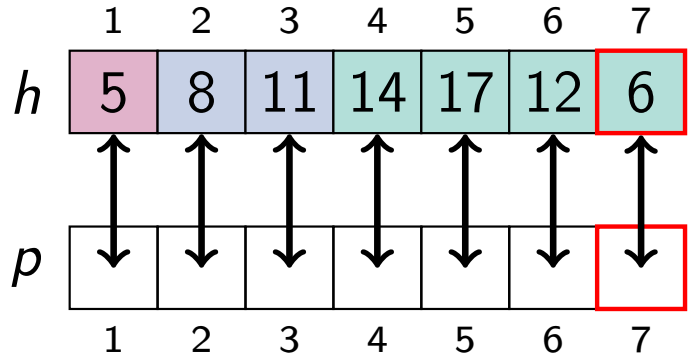
Procedure decreaseKey(h : Handle, k : Key)
assert $k \leq \text{key}(h)$
$\text{key}(h) := k$
siftUp(h)

Operation:
decreaseKey$(p[7], 6)$

Achtung: **invariant** $\text{proxy}(p[i]) = i$

Adressierbare binäre Heaps



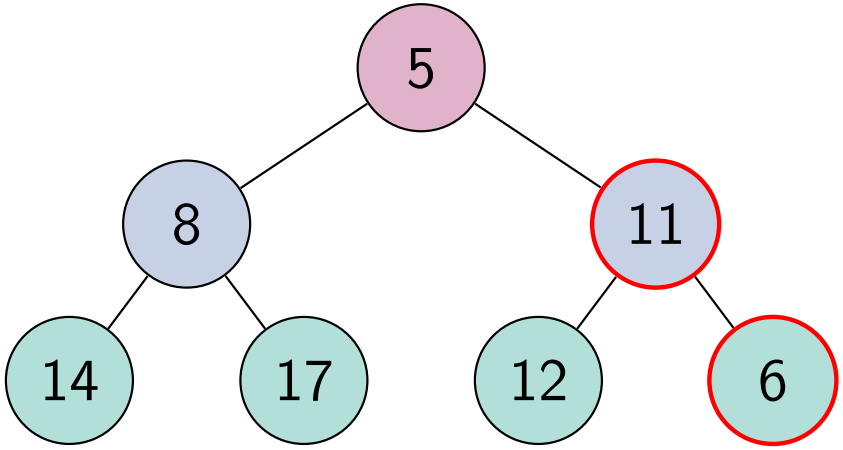


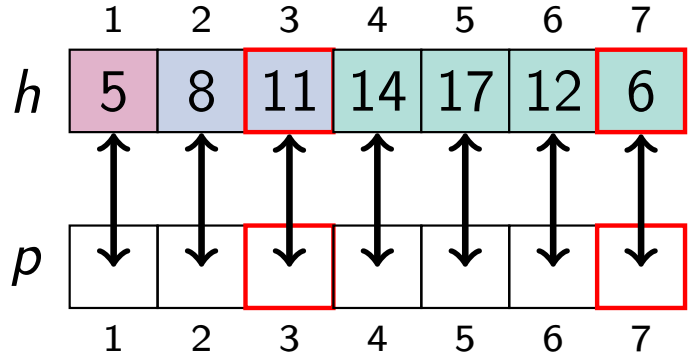
Procedure decreaseKey(h : Handle, k : Key)
assert $k \leq \text{key}(h)$
$\text{key}(h) := k$
siftUp(h)

Operation:
decreaseKey$(p[7], 6)$

Achtung: **invariant** $\text{proxy}(p[i]) = i$

Adressierbare binäre Heaps



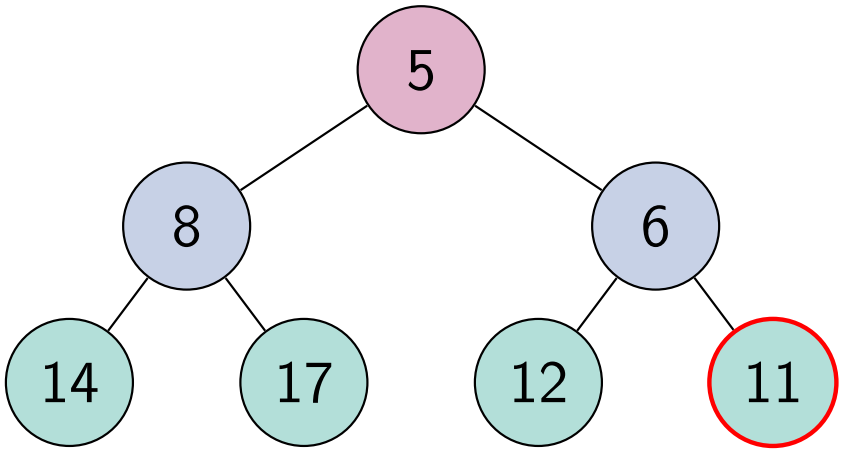


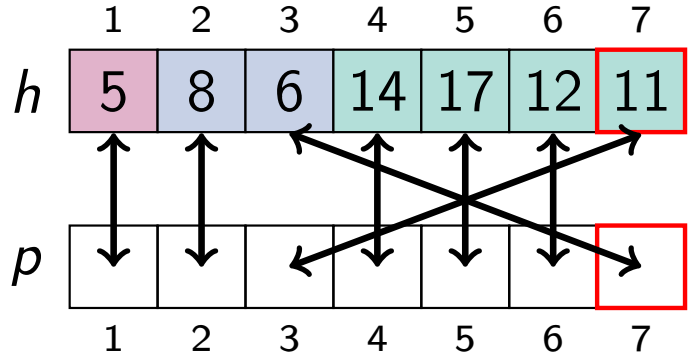
Procedure decreaseKey(h : Handle, k : Key)
assert $k \leq \text{key}(h)$
$\text{key}(h) := k$
siftUp(h)

Operation:
decreaseKey$(p[7], 6)$

Achtung: **invariant** $\text{proxy}(p[i]) = i$

Adressierbare binäre Heaps





Procedure decreaseKey(h : Handle, k : Key)
assert $k \leq \text{key}(h)$
$\text{key}(h) := k$
siftUp(h)

Operation:
decreaseKey$(p[7], 6)$

Achtung: **invariant** $\text{proxy}(p[i]) = i$