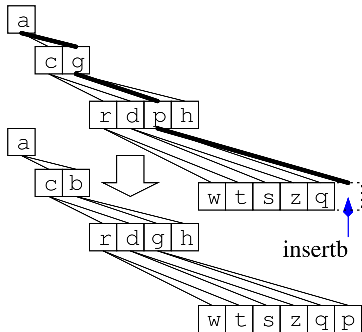
Einfügen

Procedure insert(e : $Element$)
  assert $n < w$
  n++ ; $h[n]$:= e
  siftUp(n)

Procedure siftUp(i : $\mathbb{N}$)
  assert the heap property holds
    except maybe at position i
  if $i = 1 \vee h[\lfloor i/2 \rfloor] \leq h[i]$ **then return**
  swap($h[i], h[\lfloor i/2 \rfloor]$)
  siftUp($\lfloor i/2 \rfloor$)



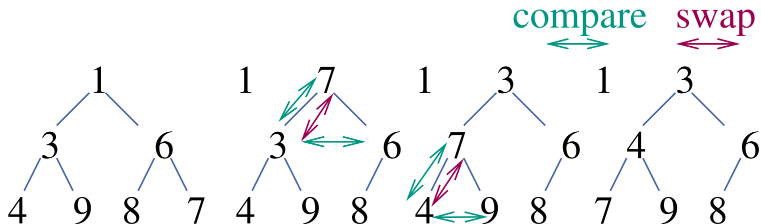
Function deleteMin : Element

  result= $h[1]$: Element

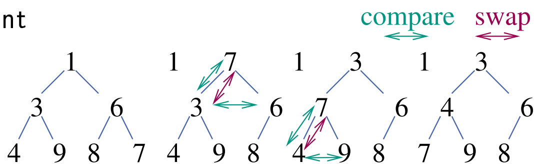
  $h[1] := h[n]$; n−−

  siftDown(1)

  return result

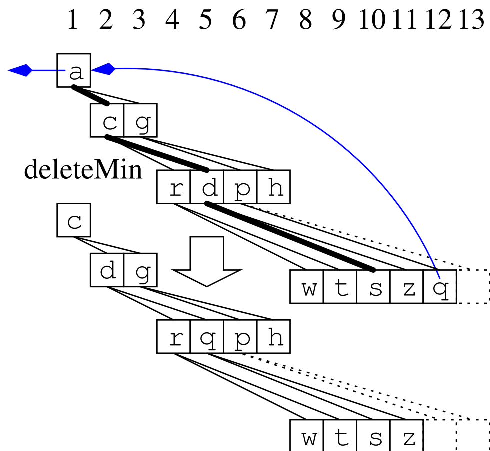


Function deleteMin : Element
  result=$h[1]$: Element
  $h[1]:= h[n]$; n−−
  siftDown(1)
  return result



Procedure siftDown(i : $\mathbb{N}$)
  assert heap property except, possibly at $j = 2i$ and $j = 2i + 1$
  if $2i \leq n$ **then** // i is not a leaf
    if $2i + 1 > n \vee h[2i] \leq h[2i + 1]$ **then** $m:= 2i$ **else** $m:= 2i + 1$
    assert $\nexists$sibling(m) $\vee$ h[sibling(m)] $\geq h[m]$
    if $h[i] > h[m]$ **then** // heap property violated
      swap($h[i], h[m]$)
      siftDown(m)
  assert the heap property holds for the subtree rooted at i

deleteMin: Beispiel



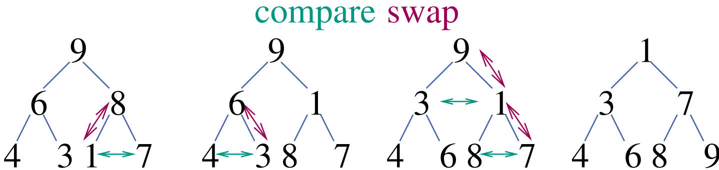
Binärer Heap – Analyse

- **Satz:** min dauert $O(1)$. □
- **Lemma:** Höhe ist $\lfloor \log n \rfloor$ □
- **Satz:** insert dauert $O(\log n)$.
- **Satz:** deleteMin dauert $O(\log n)$.
- **Beweis:** Zeit $O(1)$ pro Schicht. □

Binärer Heap – Konstruktion

Procedure buildHeapBackwards
  for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** siftDown(i)

Beispiel: Binärer Heap – Konstruktion



Binärer Heap – Konstruktion

- **Satz:** buildHeap läuft in Zeit $\mathrm{O}(n)$
- **Beweis:** Sei $k = \lfloor \log n \rfloor$.
  In Tiefe $\ell \in 0..\lfloor \log n \rfloor$:
  - 2^ℓ Aufrufe von siftDown
  - Kosten je $\mathrm{O}(k-\ell)$. Insgesamt:

$$\mathrm{O}\left(\sum_{0\le\ell<k} 2^\ell(k-\ell)\right) = \mathrm{O}\left(2^k \sum_{0\le\ell<k} \frac{k-\ell}{2^{k-\ell}}\right) = \mathrm{O}\left(2^k \underbrace{\sum_{j\ge1} \frac{j}{2^j}}_{\mathrm{O}(1)!}\right)$$

$$= \mathrm{O}\left(2^k\right) = \mathrm{O}(n)$$

Ein nützlicher Rechentrick

$$\sum_{j\geq 1} j\cdot 2^{-j} = \sum_{j\geq 1} 2^{-j} + \sum_{j\geq 2} 2^{-j} + \sum_{j\geq 3} 2^{-j} + \dots$$
$$= (1 + 1/2 + 1/4 + 1/8 + \dots)\cdot \sum_{j\geq 1} 2^{-j}$$
$$= 2\cdot 1 = 2$$

```
   1/2   +   1/4   +   1/8   +   1/16   +    ... = 1
             1/4   +   1/8   +   1/16   +    ... = 1/2
                       1/8   +   1/16   +    ... = 1/4
                                 1/16   +    ... = 1/8
                                  ...            = ...
--------------------------------------------------------
1*1/2 + 2*1/4 + 3*1/8 + 4*1/16   +    ... =  2
```

Heapsort

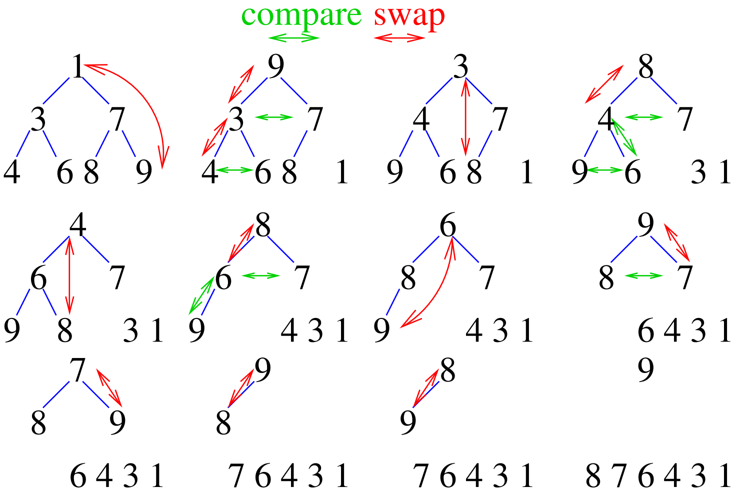
```
Procedure heapSortDecreasing(a[1..n])
    buildHeap(a)
    for i := n downto 2 do
        h[i]:= deleteMin
```

Laufzeit: $O(n \log n)$

Andere Sichtweise: effiziente Implementierung von Sortieren durch Auswahl

Frage: Wie sortiert man aufsteigend?

Heapsort: Beispiel



Heapsort $\leftrightarrow$ Quicksort $\leftrightarrow$ Mergesort

	Heapsort	Quicksort	Mergesort
Vergleiche	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$
E[Vergleiche]	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
zusätzl. Platz	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Cachezugriffe (B =Blockgröße)	$\mathcal{O}(n \log n)$	$\mathcal{O}(\frac{n}{B} \log n)$	$\mathcal{O}(\frac{n}{B} \log n)$

Kompromiss: z. B.
introspektives Quicksort der C++ Standardbibliothek:
Quicksort starten. Zu wenig Fortschritt? Umschalten auf Heapsort.

Adressierbare Prioritätslisten

Procedure build($\{e_1, \ldots, e_n\}$) $M := \{e_1, \ldots, e_n\}$
Function size **return** $|M|$
Procedure insert(e) $M := M \cup \{e\}$
Function min **return** $\min M$
Function deleteMin $e := \min M$; $M := M \setminus \{e\}$; **return** e
Function remove(h : Handle) $e := h$; $M := M \setminus \{e\}$; **return** e
Procedure decreaseKey(h : Handle, k : Key)
    assert key(h) $\geq k$; key(h) $:= k$
Procedure merge(M') $M := M \cup M'$

Adressierbare Prioritätslisten: Anwendungen

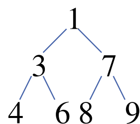
Greedy-Algorithmus:

```
while solution not complete do
    add the best available “piece” to the solution
    update piece priorities       // e.g., using addressable priority queue
```

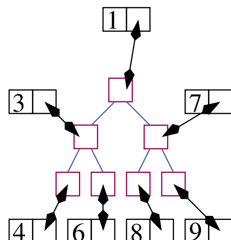
Beispiele:

- Dijkstras Algorithmus für kürzeste Wege
- Jarník-Prim Algorithmus für minimale Spannbäume
- Scheduling: Jobs $\rightarrow$ am wenigsten belastete Maschine
- Hierarchiekonstruktion für Routenplanung
- Suche nach erfüllenden Belegungen aussagenlog. Formeln?

Adressierbare Binäre Heaps



Problem: Elemente bewegen sich.
Dadurch werden Elementverweise ungültig.
(Ein) **Ausweg:** Unbewegliche Vermittler-Objekte.
Invariante: proxy(e) verweist auf Position von e.
$\rightsquigarrow$ Vermittler bei jeder Vertauschung aktualisieren.
$\rightsquigarrow$ Rückverweis Element $\rightarrow$ Vermittler



Laufzeit:
$O(\log n)$ für alle Operationen ausser merge und buildHeap, die $O(n)$ brauchen.

Adressierbare Prioritätslisten – Laufzeiten

Operation	Binary Heap	Fibonacci Heap (Buch)
build	$O(n)$	$O(n)$
size	$O(1)$	$O(1)$
min	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$
remove	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(1)$ am.
merge	$O(n)$	$O(1)$

Prioritätslisten: Mehr

- Untere Schranke $\Omega(\log n)$ für deleteMin, vergleichsbasiert. Übung
- ganzzahlige Schlüssel (stay tuned)
- extern: Geht gut (nichtaddressierbar)
- parallel: Semantik?

Prioritätslisten: Zusammenfassung

- Häufig benötigte Datenstruktur
- Addressierbarkeit ist nicht selbstverständlich
- Binäre Heaps sind einfache, relativ effiziente Implementierung

Was haben wir jenseits von Prioritätslisten gelernt?

- implizites Layout von Binärbäumen
- $\sum_j j 2^j$
- Heapsort (inplace!)