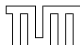
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

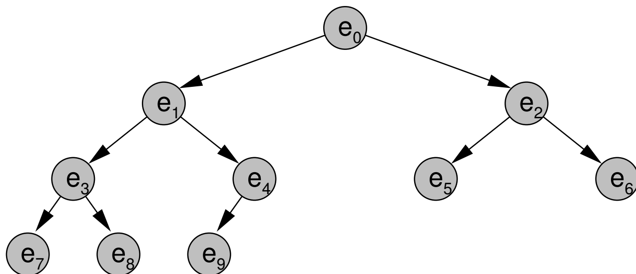
Sommersemester 2010

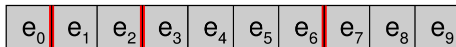


Übersicht

1 Priority Queues
- Heap

Binärer Heap als Feld





- Kinder von Knoten $H[i]$ in $H[2i+1]$ und $H[2i+2]$
- Form-Invariante: $H[0] \dots H[n-1]$ besetzt
- Heap-Invariante: $H[i] \leq \min\{H[2i+1], H[2i+2]\}$

Binärer Heap als Feld

insert(e)

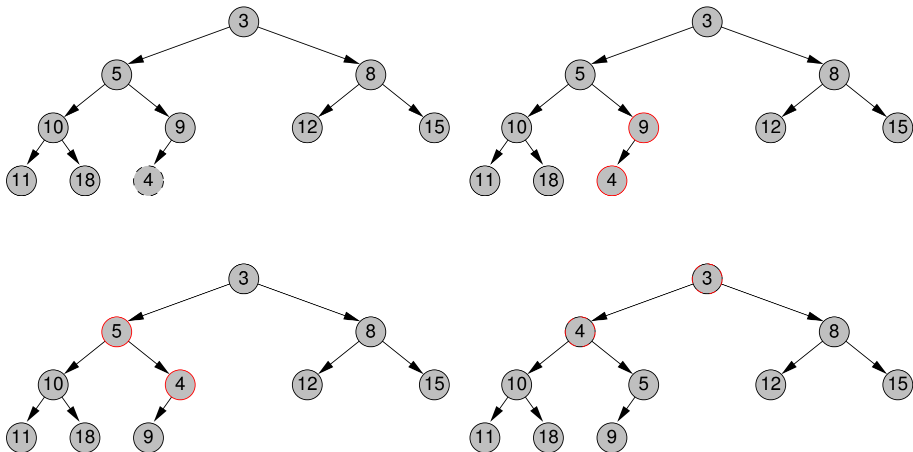
- Form-Invariante: $H[n] = e$; siftUp(n); n++;
- Heap-Invariante:

  vertausche e mit seinem Vater bis
  $\text{prio}(H[\lfloor (k-1)/2 \rfloor]) \leq \text{prio}(e)$ für e in $H[k]$ (oder e in $H[0]$)

  ```
  void siftUp(i) {
     while (i > 0 ∧ prio(H[(i − 1)/2]) > prio(H[i])) {
        swap(H[i], H[(i − 1)/2]);
        i = (i − 1)/2;
     }
  }
  ```

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

Heap - siftUp()



Binärer Heap als Feld

deleteMin()

- Form-Invariante:
  $e = H[0]$;
  $n--$;
  $H[0] = H[n]$;
  siftDown(0);
  return e;

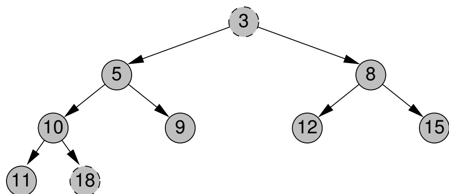
- Heap-Invariante: (siftDown)
  vertausche e (anfangs Element in $H[0]$) mit dem Kind, dass die kleinere Priorität hat, bis e ein Blatt ist oder
  $\text{prio}(e) \leq \min\{\text{prio}(c_1(e)), \text{prio}(c_2(e))\}$.

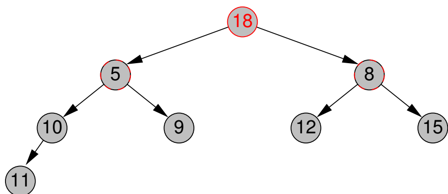
- Laufzeit: $\mathcal{O}(\log n)$

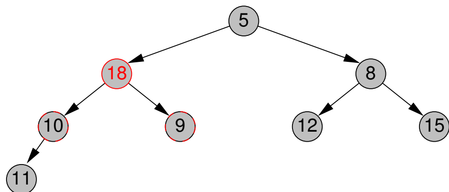
Binärer Heap als Feld

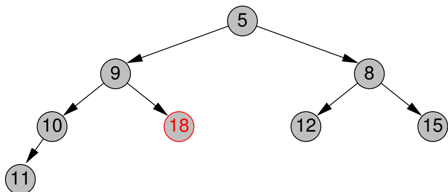
```
void siftDown(i) {
   int m;
   while (2i + 1 < n) {
      if (2i + 2 ≥ n)
         m = 2i + 1;
      else
         if (prio(H[2i + 1]) < prio(H[2i + 2]))
            m = 2i + 1;
         else m = 2i + 2;
      if (prio(H[i]) ≤ prio(H[m]))
         return;
      swap(H[i], H[m]);
      i = m;
   }
}
```

Heap - siftDown()









Binärer Heap / Aufbau

build($\{e_0, \ldots, e_{n-1}\}$)

- naiv:

  Für alle $i \in \{0, \ldots, n-1\}$:
  insert(e_i)

$\Rightarrow$ Laufzeit: $\Theta(n \log n)$

Binärer Heap / Aufbau

build($\{e_0, \ldots, e_{n-1}\}$)

effizient:

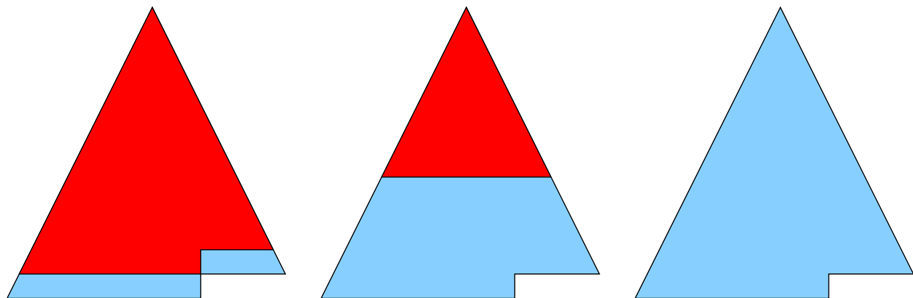
- Für alle $i \in \{0, \ldots, n-1\}$:
  $H[i] := e_i$.
- Für alle $i \in \left\{\left\lfloor \frac{n-1}{2} \right\rfloor, \ldots, 0\right\}$:
  siftDown(i)

Laufzeit:

- $k = \lfloor \log n \rfloor$: Baumtiefe (gemessen in Kanten)
- Kosten für siftDown von Level ℓ aus sind proportional zur Resttiefe $(k - \ell)$
- Es gibt $\leq 2^\ell$ Knoten in Tiefe ℓ.

$$\mathcal{O}\left(\sum_{1 \leq \ell < k} 2^\ell (k - \ell)\right) \subseteq \mathcal{O}\left(2^k \sum_{0 \leq \ell < k} \frac{k-\ell}{2^{k-\ell}}\right) \subseteq \mathcal{O}\left(2^k \sum_{j \geq 1} \frac{j}{2^j}\right) \subseteq \mathcal{O}(n)$$

Binärer Heap / Aufbau



Laufzeiten des Binären Heaps

- min(): $\mathcal{O}(1)$
- insert(e): $\mathcal{O}(\log n)$
- deleteMin(): $\mathcal{O}(\log n)$
- build($e_0, \ldots, e_{n-1}$): $\mathcal{O}(n)$

Zusätzliche Operationen für erweiterte (adressierbare) Priority Queue:

- Handle h insert(Element e): $\mathcal{O}(\log n)$
- remove(Handle h): $\mathcal{O}(\log n)$
- decreaseKey(Handle h, int k): $\mathcal{O}(\log n)$
- M.merge(Q): $\Theta(n)$

HeapSort

Verbesserung von SelectionSort:

- erst build$(e_0, \ldots, e_{n-1})$: $\mathcal{O}(n)$
- dann $n \times$ deleteMin(): $\mathcal{O}(n \log n)$
- in-place, aber nicht stabil
- Gesamtlaufzeit: $\mathcal{O}(n \log n)$