Script generated by TTT

Title: Täubig: GAD (19.06.2012)

Date: Tue Jun 19 14:32:49 CEST 2012

Duration: 61:56 min

Pages: 24

Übersicht

7 Priority Queues

- Allgemeines
- Heaps
- Binomial Heaps

Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich mehr als zwei Runs (k) mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow$ $(k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k(n/M) \rceil$

$\Rightarrow$ $(2n/B)\left(1 + \left\lceil \log_{M/B}(n/M) \right\rceil\right)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine einzige Merge-Phase (erst M/B Runs der Größe M, dann einmal Merge)

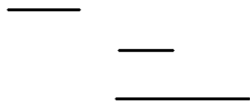
Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren
- In jeder Merge-Phase wird das ganze Feld einmal gelesen und geschrieben

$\Rightarrow$ $(2n/B)(1 + \lceil \log_2(n/M) \rceil)$ Block-Transfers

Übersicht

7 Priority Queues
- Allgemeines
- Heaps
- Binomial Heaps



Adressierbare Prioritätswarteschlangen

Zusätzliche Operationen für adressierbare PQs:

- Handle insert(Element e): wie zuvor, gibt aber ein Handle (Referenz / Zeiger) auf das eingefügte Element zurück
- remove(Handle h): lösche Element spezifiziert durch Handle h
- decreaseKey(Handle h, int k): reduziere Schlüssel / Priorität des Elements auf Wert k (je nach Implementation evt. auch um Differenz k)
- M.merge(Q): $M = M \cup Q$; $Q = \emptyset$;

Prioritätswarteschlangen mit Listen

Priority Queue mittels unsortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $O(n)$
- insert(Element e): Zeit $O(1)$
- min(), deleteMin(): Zeit $O(n)$ $\Theta(n)$

Priority Queue mittels sortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $O(n \log n)$
- insert(Element e): Zeit $O(n)$
- min(), deleteMin(): Zeit $O(1)$

$\Rightarrow$ Bessere Struktur als eine Liste notwendig!

Übersicht

7 Priority Queues
- Allgemeines
- Heaps
- Binomial Heaps

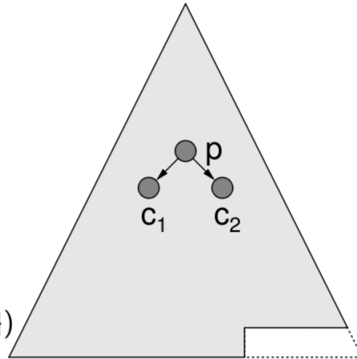
Binärer Heap

Idee: verwende Binärbaum

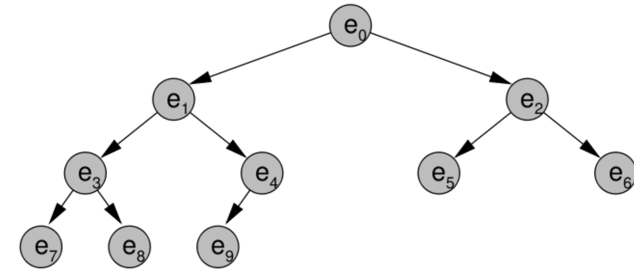
Bewahre zwei Invarianten:

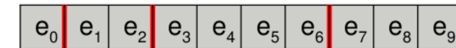
- Form-Invariante: fast vollständiger Binärbaum
- Heap-Invariante:

  prio(p) ≤ min {prio(c1), prio(c2)}



Binärer Heap als Feld





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

Binärer Heap als Feld

insert(e)

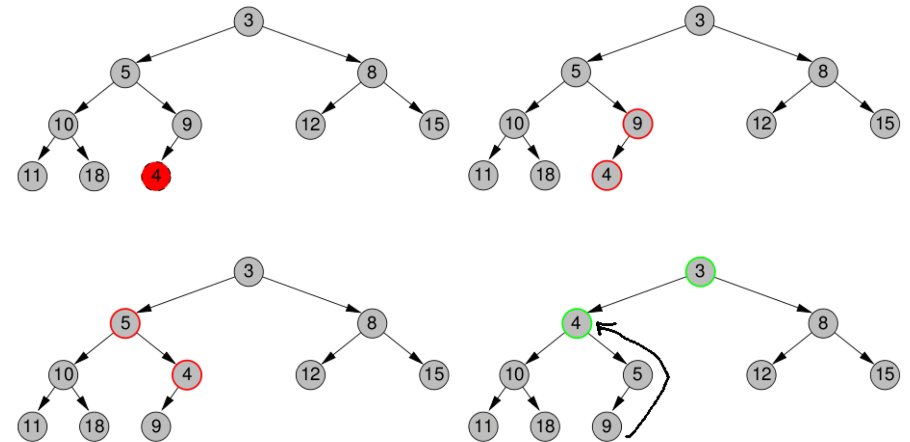
- Form-Invariante: $H[n] = e$; siftUp(n); n++;
- Heap-Invariante:

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

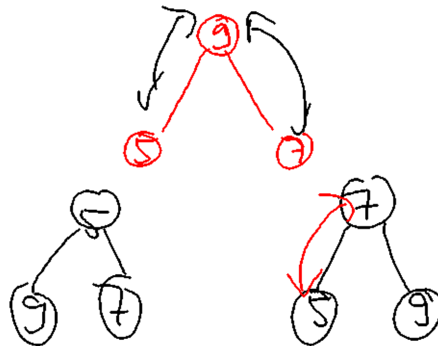
```
siftUp(i) {
   while (i > 0 ∧ prio(H[⌊(i − 1)/2⌋]) > prio(H[i])) {
      swap(H[i], H[⌊(i − 1)/2⌋]);
      i = ⌊(i − 1)/2⌋;
   }
}
```

- Laufzeit: $O(\log n)$

Heap - siftUp()



Binärer Heap als Feld



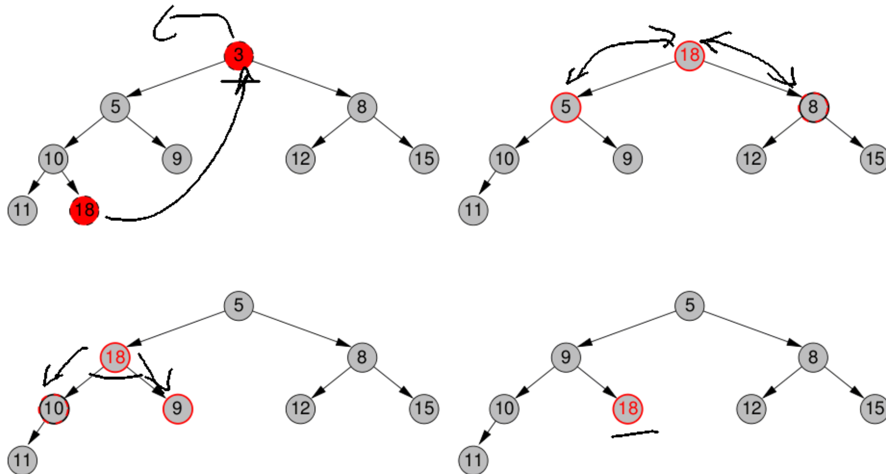
deleteMin()

- 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: $O(\log n)$

Binärer Heap als Feld

```
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)

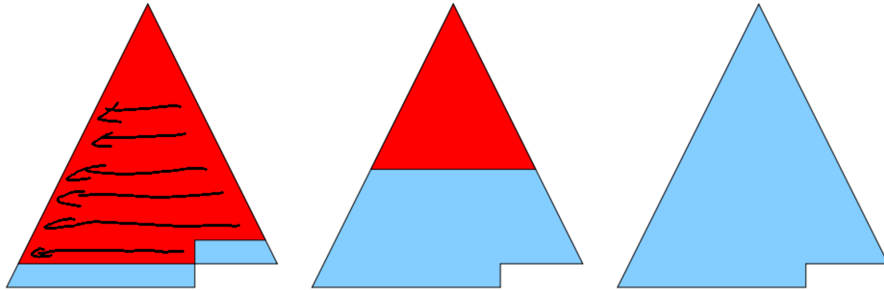
$\Rightarrow$ Laufzeit: $\Theta(n \log n)$

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 $i \in \left\{\left\lfloor \frac{n}{2} \right\rfloor - 1, \ldots, 0\right\}$:
  siftDown(i)



Binärer Heap / Aufbau

Laufzeit:

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

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

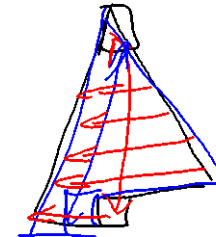
$$\begin{aligned}\sum_{j \geq 1} j \cdot 2^{-j} &= \sum_{j \geq 1} 2^{-j} + \sum_{j \geq 2} 2^{-j} + \sum_{j \geq 3} 2^{-j} + \ldots \\ &= 1 \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{2} \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{4} \cdot \sum_{j \geq 1} 2^{-j} + \ldots \\ &= (1 + 1/2 + 1/4 + \ldots) \sum_{j \geq 1} 2^{-j} = 2 \cdot 1 = 2\end{aligned}$$

Laufzeiten des Binären Heaps

- min(): $O(1)$
- insert(e): $O(\log n)$
- deleteMin(): $O(\log n)$
- build($e_0, \ldots, e_{n-1}$): $O(n)$
- M.merge(Q): $\Theta(n)$

Adressen bzw. Feldindizes in array-basierten Binärheaps können nicht als Handles verwendet werden, da die Elemente bei den Operationen verschoben werden
$\Rightarrow$ ungeeignet als adressierbare PQs (kein remove bzw. decreaseKey)

HeapSort



Verbesserung von SelectionSort:

- erst build($e_0, \ldots, e_{n-1}$): $O(n)$
- dann $n \times$ deleteMin():
  vertausche in jeder Runde erstes und letztes Heap-Element, dekrementiere Heap-Größe und führe siftDown(0) durch:
  $O(n \log n)$

$\Rightarrow$ sortiertes Array entsteht von hinten,
ansteigende Sortierung kann mit Max-Heap erzeugt werden

- in-place, aber nicht stabil
- Gesamtlaufzeit: $O(n \log n)$

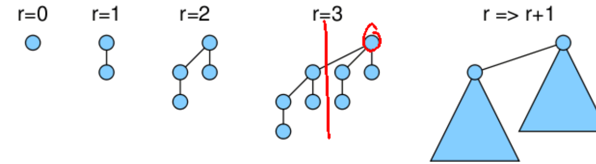
Übersicht

7 Priority Queues
- Allgemeines
- Heaps
- Binomial Heaps

Binomial Heaps

basieren auf Binomial-Bäumen

- Form-Invariante:



- Heap-Invariante:

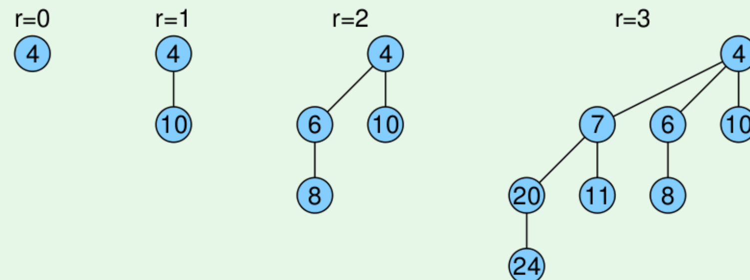
$$\text{prio(Vater)} \leq \text{prio(Kind)}$$

wichtig: Elemente der Priority Queue werden in Heap Items gespeichert, die eine feste Adresse im Speicher haben und damit als Handles dienen können (im Gegensatz zu array-basierten Binärheaps)

Binomial-Bäume

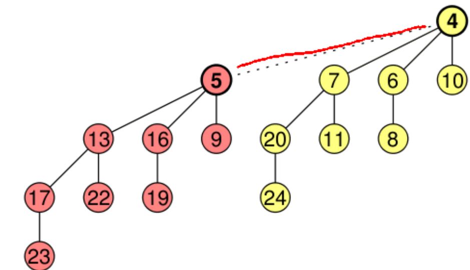
Beispiel

Korrekte Binomial-Bäume:

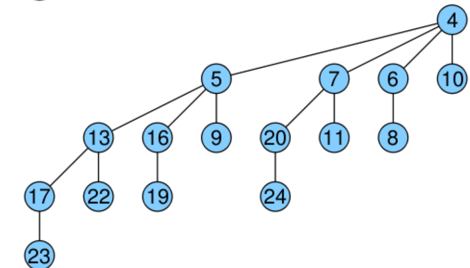


Binomial-Baum: Merge

Wurzel mit größerem Wert wird neues Kind der Wurzel mit kleinerem Wert! (Heap-Bedingung)

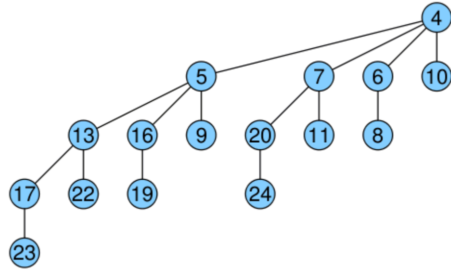


aus zwei B_{r-1} wird ein B_r



Binomial-Baum: Löschen der Wurzel (deleteMin)

aus einem B_r



werden $B_{r-1}, \ldots, B_0$