Script generated by TTT

Title: Seidl: GAD (09.06.2015)

Date: Tue Jun 09 13:45:18 CEST 2015

Duration: 147:18 min

Pages: 61

QuickSelect

teilt das Feld jeweils in 3 Teile:

- a Elemente kleiner als das Pivot
- b Elemente gleich dem Pivot
- c Elemente größer als das Pivot

$T(n)$: erwartete Laufzeit bei n Elementen

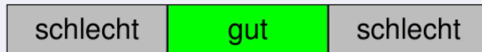
Satz

Die erwartete Laufzeit von QuickSelect ist linear: $T(n) \in O(n)$.

QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als $2/3$ der aktuellen Feldgröße sind:



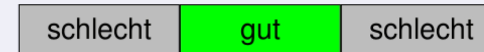
$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] \geq 1/3$$

QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als $2/3$ der aktuellen Feldgröße sind:



$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] = 1/3$$

Erwartete Zeit bei n Elementen

- linearer Aufwand außerhalb der rekursiven Aufrufe: cn
- Pivot gut (Wsk. $1/3$): Restaufwand $\leq T(2n/3)$
- Pivot schlecht (Wsk. $2/3$): Restaufwand $\leq T(n-1) < T(n)$

QuickSelect

Beweis.

$$
\begin{aligned}
T(n) &\le cn + p \cdot T(n \cdot 2/3) + (1-p) \cdot T(n) \\
p \cdot T(n) &\le cn + p \cdot T(n \cdot 2/3) \\
T(n) &\le cn/p + T(n \cdot 2/3) \\
&\le cn/p + c \cdot (n \cdot 2/3)/p + T(n \cdot (2/3)^2) \\
&\dots \text{ wiederholtes Einsetzen} \\
&\le (cn/p)(1 + 2/3 + 4/9 + 8/27 + \dots) \\
&\le \frac{cn}{p} \cdot \sum_{i \ge 0} (2/3)^i \\
&\le \frac{cn}{1/3} \cdot \frac{1}{1 - 2/3} = 9cn \in O(n)
\end{aligned}
$$

□

Übersicht

6 Sortieren
- Einfache Verfahren
- MergeSort
- Untere Schranke
- QuickSort
- Selektieren
- Schnelleres Sortieren
- Externes Sortieren

Sortieren schneller als $O(n \log n)$

Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
  z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer / das erste Zeichen.
  Nur bei gleichem Anfang kommt es auf mehr Ziffern / Zeichen an.

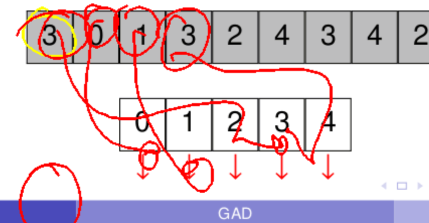
Sortieren | Schnelleres Sortieren

Sortieren schneller als $O(n \log n)$

Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
  z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer / das erste Zeichen.
  Nur bei gleichem Anfang kommt es auf mehr Ziffern / Zeichen an.

- Annahme: Elemente sind Zahlen im Bereich $\{0, \dots, K-1\}$
- Strategie: verwende Feld von K Buckets (z.B. Listen)



Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```


Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n+K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n+K)$

Sortieren schneller als $\mathcal{O}(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n+K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n+K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

⇒ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

RadixSort

- verwende K-adische Darstellung der Schlüssel
- Annahme:
  Schlüssel sind Zahlen aus $\{0,\ldots,K^d-1\}$ repräsentiert durch d Stellen von Ziffern aus $\{0,\ldots,K-1\}$
- sortiere zunächst entsprechend der niedrigstwertigen Ziffer mit kSort und dann nacheinander für immer höherwertigere Stellen
- behalte Ordnung der Teillisten bei

RadixSort

```
radixSort(Sequence<Elem> s) {
    for (int i = 0; i < d; i++)
        kSort(s,i);      // sortiere gemäß key_i(x)
                         // mit key_i(x) = (key(x)/K^i) mod K
}
```


RadixSort

radixSort(Sequence<Elem> s) {
for (int $i = 0$; $i < d$; i++)
kSort(s,i); // sortiere gemäß $key_i(x)$
// mit $key_i(x) = (key(x)/K^i) \bmod K$
}

RadixSort

radixSort(Sequence<Elem> s) {
for (int $i = 0$; $i < d$; i++)
kSort(s,i); // sortiere gemäß $key_i(x)$
// mit $key_i(x) = (key(x)/K^i) \bmod K$
}

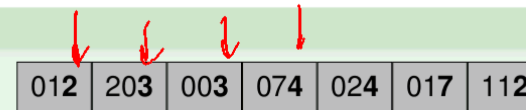
Verfahren funktioniert, weil kSort stabil ist:

Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i-1 \ldots 0$ während der Sortierung nach Ziffer i

Laufzeit: $O(d(n+K))$ für n Schlüssel aus $\{0, \ldots, K^d - 1\}$

RadixSort

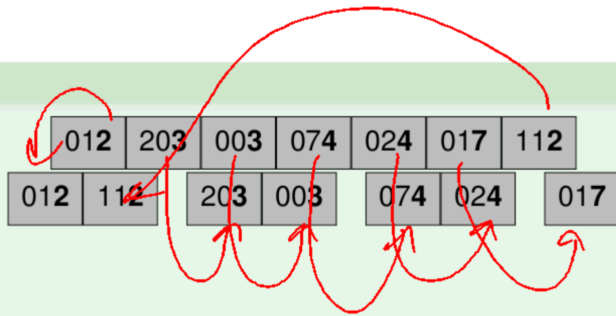
Beispiel



012	203	003	074	024	017	112

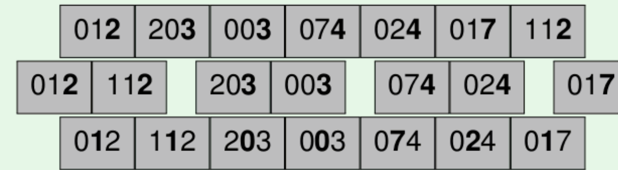
RadixSort

Beispiel



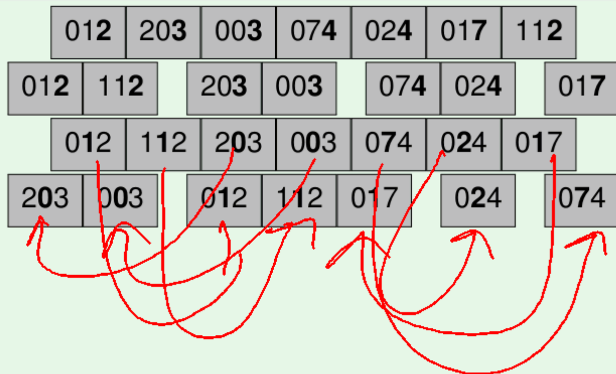
RadixSort

Beispiel



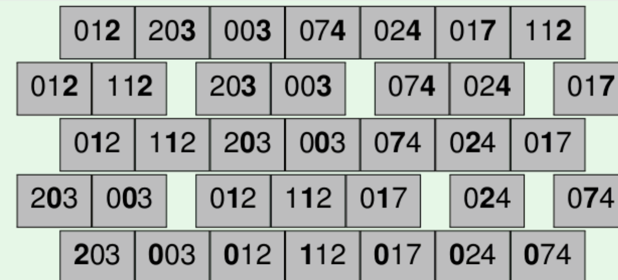
RadixSort

Beispiel



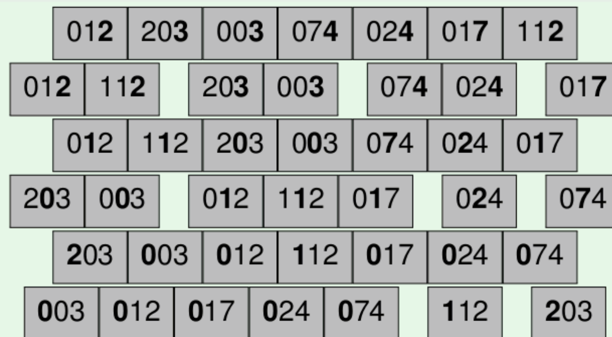
RadixSort

Beispiel



RadixSort

Beispiel

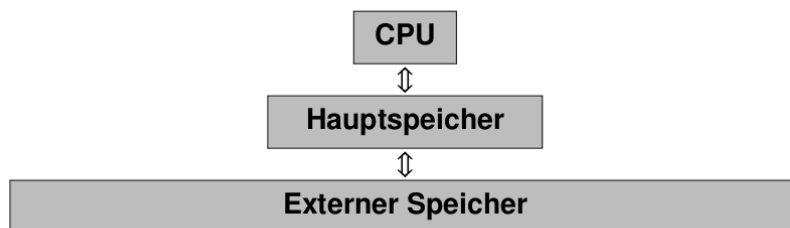


Übersicht

6 Sortieren
- Einfache Verfahren
- MergeSort
- Untere Schranke
- QuickSort
- Selektieren
- Schnelleres Sortieren
- Externes Sortieren

Externes Sortieren

Heutige Computer:



- Hauptspeicher hat Größe M
- Transfer zwischen Hauptspeicher und externem Speicher mit Blockgröße B

Externes Sortieren

Problem:

Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:

Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Externes Sortieren

Problem:

Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:
Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Lösung: Verwende MergeSort

Vorteil:

MergeSort verwendet oft konsekutive Elemente (Scanning)
(geht auf Festplatte schneller als Random Access-Zugriffe)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

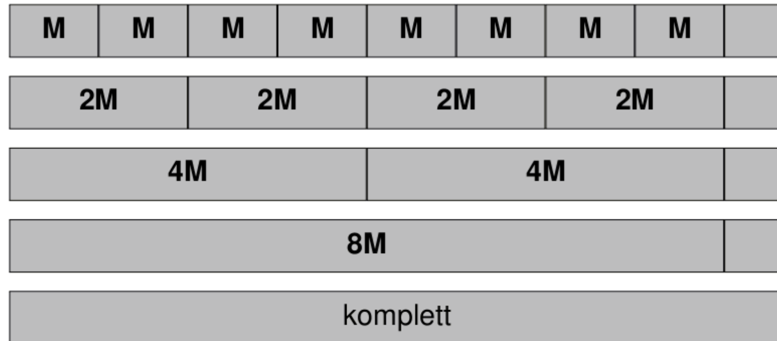
$\Rightarrow$ benötigt n/B Blocklese- und n/B Blockschreiboperationen
Laufzeit: $2n/B$ Transfers

- ergibt sortierte Bereiche (Runs) der Größe M

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $\lceil \log_2(n/M) \rceil$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



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 ⇒ neuen Block laden
- Wenn Ausgabepuffer voll ⇒ Block auf Festplatte schreiben und Ausgabepuffer leeren

Sortieren | Externes Sortieren

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 ⇒ neuen Block laden
- Wenn Ausgabepuffer voll ⇒ Block auf Festplatte schreiben und Ausgabepuffer leeren

- In jeder Merge-Phase wird das ganze Feld einmal gelesen und geschrieben

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

Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich $k \geq 2$ Runs 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

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

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

⇒ $(2n/B)\left(1 + \lceil \log_{M/B}(n/M) \rceil\right)$ 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)

Übersicht

7 Priority Queues
- Allgemeines
- Heaps
- Binomial Heaps

Prioritätswarteschlangen

M: Menge von Elementen

prio(e): Priorität von Element e

Operationen:

- M.build($\{e_1, \ldots, e_n\}$): $M = \{e_1, \ldots, e_n\}$
- M.insert(Element e): $M = M \cup e$
- Element M.min(): gib ein e mit minimaler Priorität prio(e) zurück
- Element M.deleteMin():
  entferne Element e mit minimalem Wert prio(e)
  und gib es zurück

Adressierbare Prioritätswarteschlangen

Zusätzliche Operationen für adressierbare Priority Queues:

- 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 $O(n)$

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!

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 $O(n)$

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!

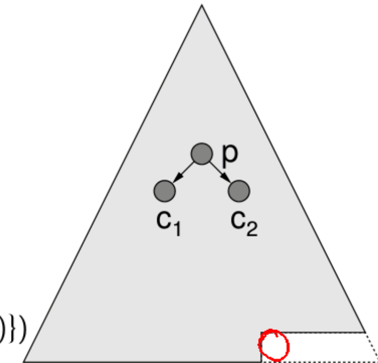
Binärer Heap

Idee: verwende Binärbaum

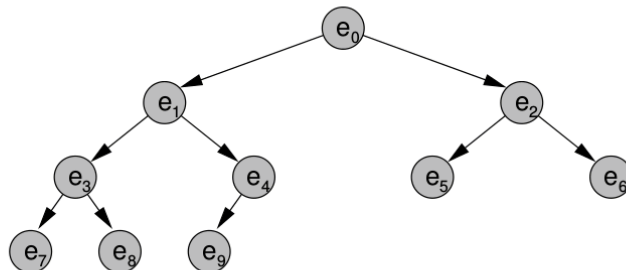
Bewahre zwei Invarianten:

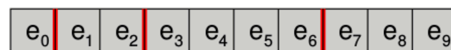
- Form-Invariante: fast vollständiger Binärbaum
- Heap-Invariante:

  prio(p) $\leq$ 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)

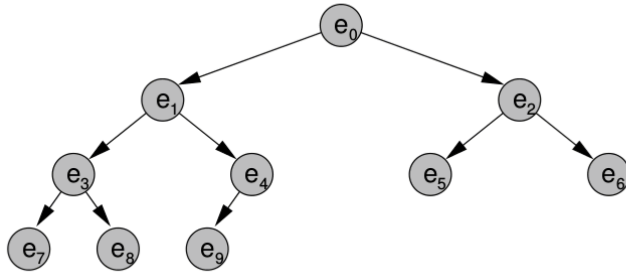
- Form-Invariante: $H[n] = e$; siftUp(n); n++;
- Heap-Invariante:

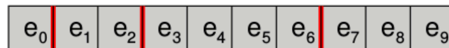
  vertausche e mit seinem Vater bis
  prio($H[\lfloor (k-1)/2 \rfloor]$) $\leq$ prio(e) für e in $H[k]$ (oder e in $H[0]$)

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

- Laufzeit: $O(\log n)$

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:

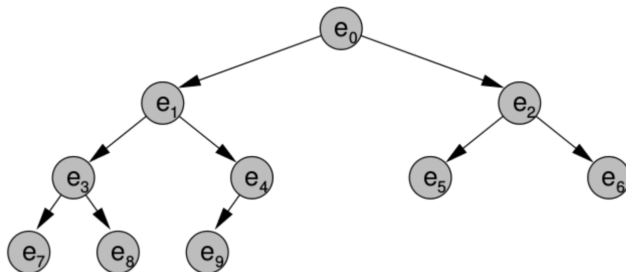
  vertausche e mit seinem Vater bis
  $\text{prio}(H[\lfloor (k-1)/2 \rfloor]) \leq \text{prio}(e)$ für e in $H[k]$ (oder e in $H[0]$)

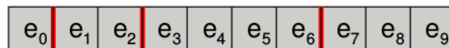
  siftUp(i) {
  while ($i > 0 \wedge \text{prio}(H[\lfloor (i-1)/2 \rfloor]) > \text{prio}(H[i])$) {
  swap($H, i, \lfloor (i-1)/2 \rfloor$);
  $i = (i-1)/2;$
  }
  }

- Laufzeit: $O(\log n)$

Priority Queues Heaps

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: $H[i] \leq \min\{H[2i+1], H[2i+2]\}$

H. Seidl (TUM) GAD SS'15 285

Priority Queues Heaps

Binärer Heap als Feld

insert(e)

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

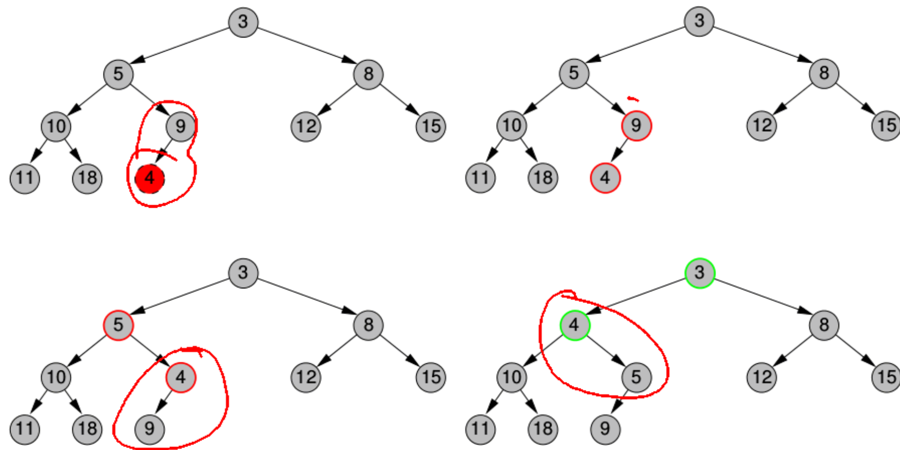
  vertausche e mit seinem Vater bis
  $\text{prio}(H[\lfloor (k-1)/2 \rfloor]) \leq \text{prio}(e)$ für e in $H[k]$ (oder e in $H[0]$)

  siftUp(i) {
  while ($i > 0 \wedge \text{prio}(H[\lfloor (i-1)/2 \rfloor]) > \text{prio}(H[i])$) {
  swap($H, i, \lfloor (i-1)/2 \rfloor$);
  $i = (i-1)/2;$
  }
  }

- Laufzeit: $O(\log n)$

H. Seidl (TUM) GAD SS'15 286

Heap - siftUp()



Binärer Heap als Feld

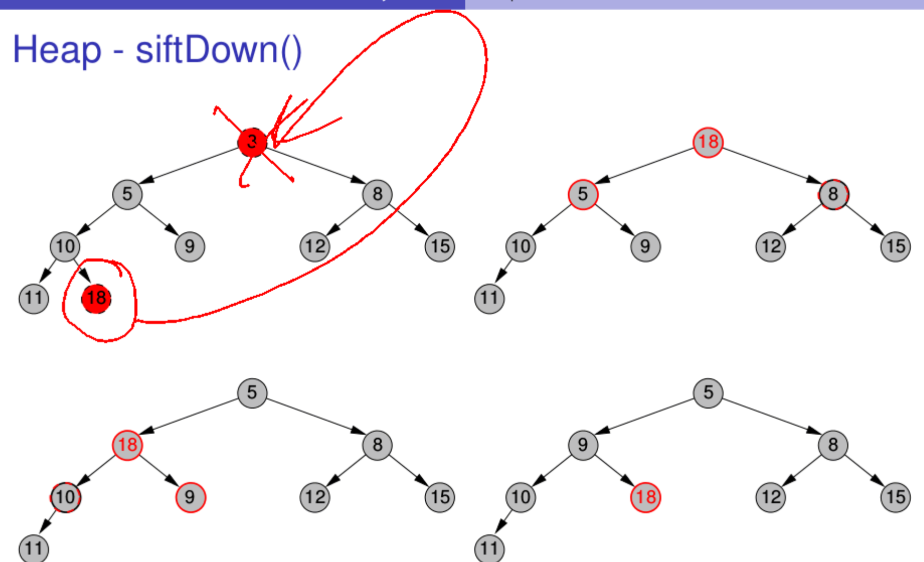
deleteMin()

- 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, das 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, m);
    i = m;
  }
}
```


Heap - siftDown()



Binärer Heap / Aufbau

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

- naiv:

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

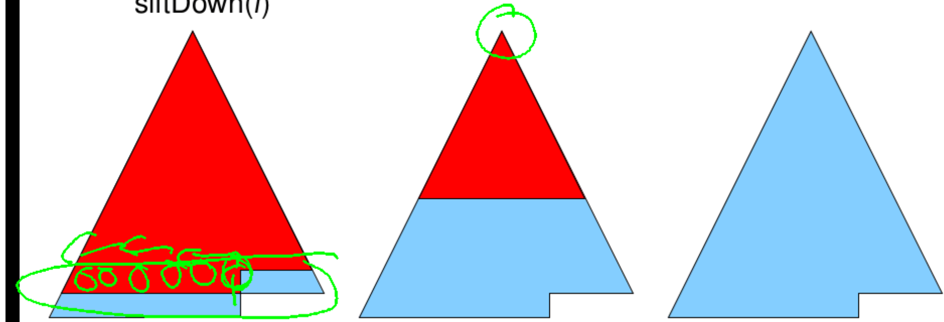
$\Rightarrow$ Laufzeit: $\Theta(n \log n)$

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:
  $O(n \log n)$

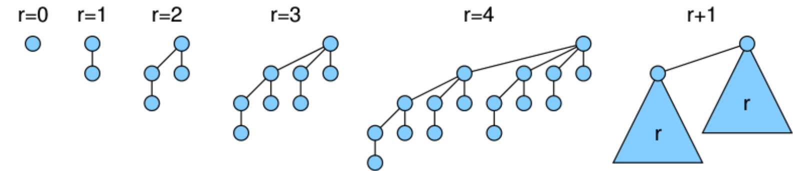
$\Rightarrow$ sortiertes Array entsteht von hinten, ansteigende Sortierung kann mit Max-Heap erzeugt werden

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

Binomial-Bäume

Binomial Heaps bestehen aus Binomial-Bäumen

- Form-Invariante:



- Heap-Invariante:

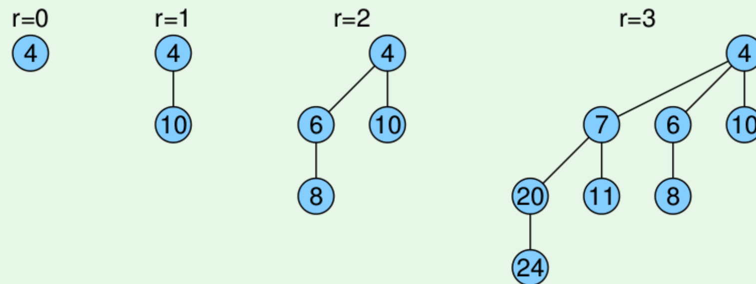
$$\text{prio(Vater)} \leq \text{prio(Kind)}$$

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

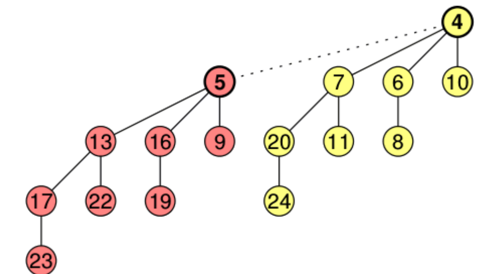
Beispiel

Korrekte Binomial-Bäume:

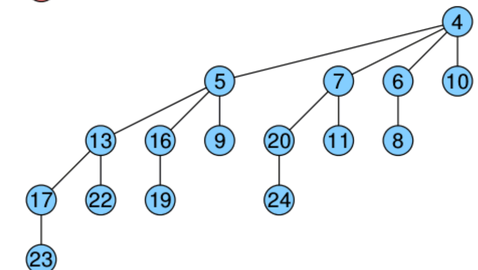


Binomial-Baum: Merge

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

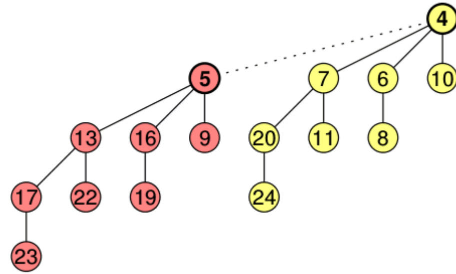


aus zwei B_{r-1} wird ein B_r

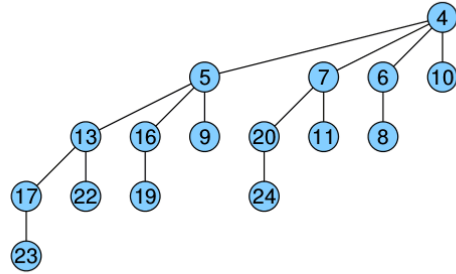


Binomial-Baum: Merge

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

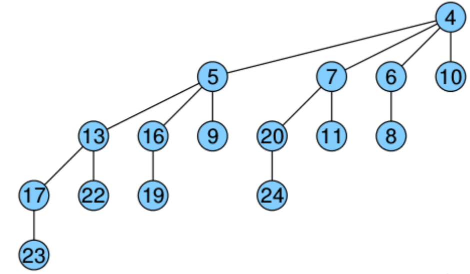


aus zwei B_{r-1} wird ein B_r

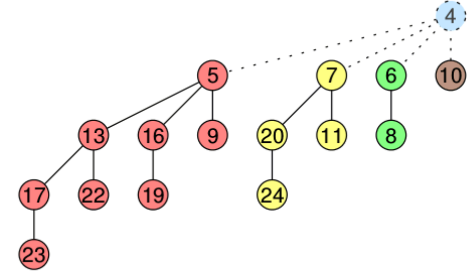


Binomial-Baum: Löschen der Wurzel (deleteMin)

aus einem B_r



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



Binomial-Baum: Knotenanzahl

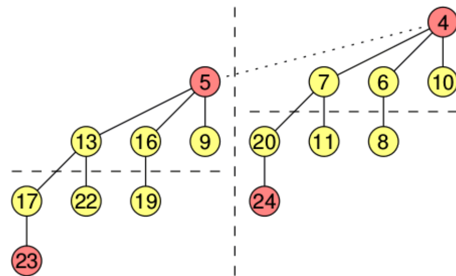
B_r hat auf Level $k \in \{0, \dots, r\}$ genau $\binom{r}{k}$ Knoten

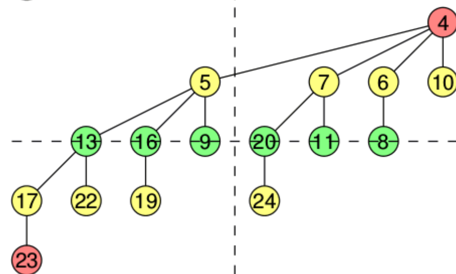
Warum?

Bei Bau des B_r aus 2 B_{r-1} gilt:

$$\binom{r}{k} = \binom{r-1}{k-1} + \binom{r-1}{k}$$

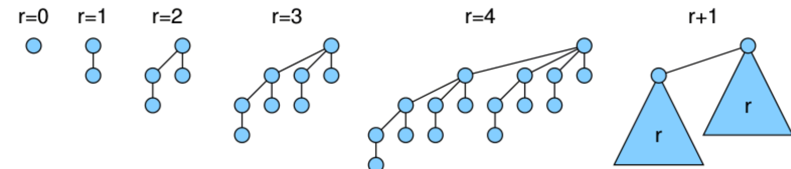
Insgesamt: B_r hat 2^r Knoten





Binomial-Bäume

Eigenschaften von Binomial-Bäumen:



Binomial-Baum vom Rang r

- hat Höhe r (gemessen in Kanten)
- hat maximalen Grad r (Wurzel)
- hat auf Level $\ell \in \{0, \dots, r\}$ genau $\binom{r}{\ell}$ Knoten
- hat $\sum_{\ell=0}^{r} \binom{r}{\ell} = 2^r$ Knoten
- zerfällt bei Entfernen der Wurzel in r Binomial-Bäume von Rang 0 bis $r-1$

Binomial-Baum: Knotenanzahl

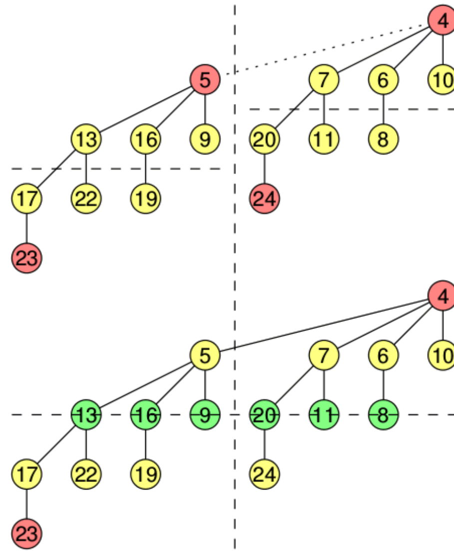
B_r hat auf Level $k \in \{0, \ldots, r\}$ genau $\binom{r}{k}$ Knoten

Warum?

Bei Bau des B_r aus 2 B_{r-1} gilt:

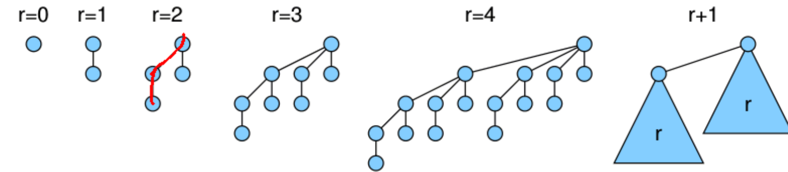
$$\binom{r}{k} = \binom{r-1}{k-1} + \binom{r-1}{k}$$

Insgesamt: B_r hat 2^r Knoten



Binomial-Bäume

Eigenschaften von Binomial-Bäumen:



Binomial-Baum vom Rang r

- hat Höhe r (gemessen in Kanten)
- hat maximalen Grad r (Wurzel)
- hat auf Level $\ell \in \{0, \ldots, r\}$ genau $\binom{r}{\ell}$ Knoten
- hat $\sum_{\ell=0}^{r} \binom{r}{\ell} = 2^r$ Knoten
- zerfällt bei Entfernen der Wurzel in r Binomial-Bäume von Rang 0 bis $r-1$