Script generated by TTT

Title: Seidl: GAD (25.05.2016)

Date: Wed May 25 13:22:42 CEST 2016

Duration: 47:14 min

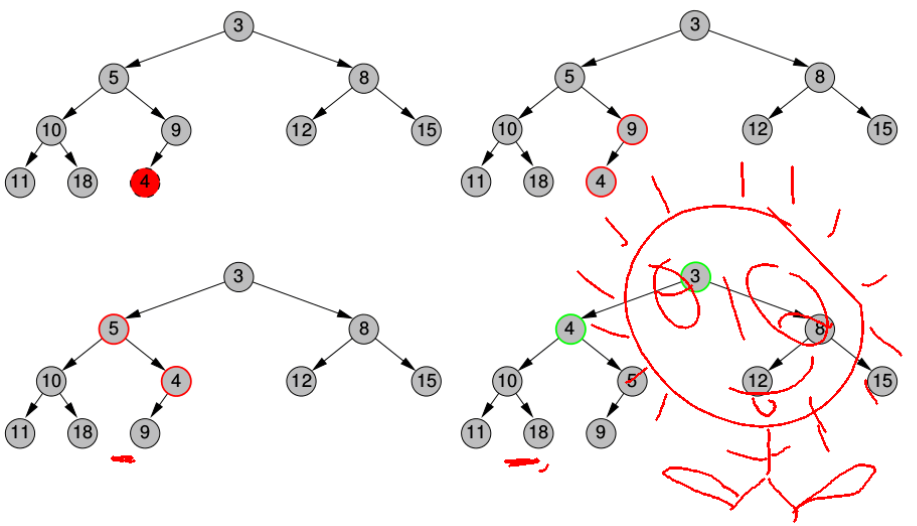
Pages: 35

QuickSelect

Methode analog zu QuickSort

```
Element quickSelect(Element[] a, int l, int r, int k) {
  // a[l...r]: Restfeld,   k: Rang des gesuchten Elements
  if (r == l) return a[l];
  int z = zufällige Position in {l,...,r};   swap(a,z,r);
  Element p = a[r];   int i = l − 1;   int j = r;
  do {   // spalte Elemente in a[l,...,r − 1] nach Pivot p
    do i++ while (a[i] < p);
    do j−− while (a[j] > p && j ≠ l);
    if (i < j) swap(a,i,j);
  } while (i < j);
  swap (a,i,r);   // Pivot an richtige Stelle
  if (k < i) return quickSelect(a, l, i − 1, k);
  if (k > i) return quickSelect(a, i + 1, r, k);
  else return a[k];   // k == i
}
```

Heap - siftUp()



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 $H[0]$)

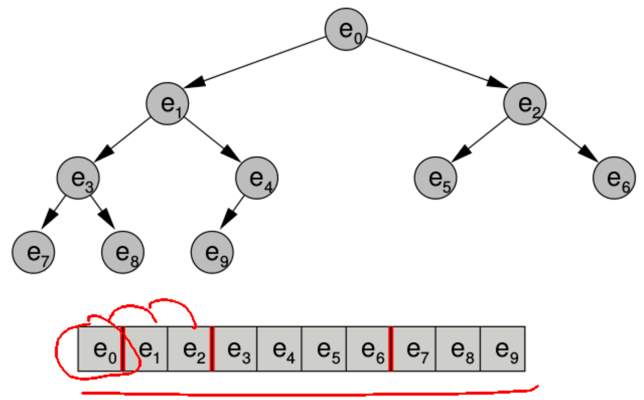
  ```
  siftUp(i) {
    while (i > 0 ∧ prio(H[⌊(i − 1)/2⌋]) > prio(H[i])) {
      swap(H, i, ⌊(i − 1)/2⌋);
      i = (i − 1)/2;
    }
  }
  ```
- Laufzeit: $O(\log n)$

Binärer Heap als Feld

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



- 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

deleteMin()

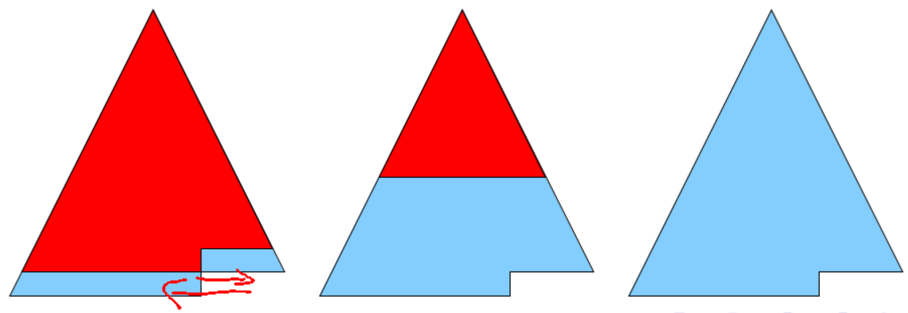
- Form-Invariante:
  $e = H[0]$;
  $n--$;
  $H[0] = H[n]$;
  siftDown(0);
  return e;
- Heap-Invariante: (siftDown)
  vertausche e (anfangs Element in $H[0]$) mit dem Kind, das die kleinere Priorität hat, bis e ein Blatt ist oder
  $\text{prio}(e) \leq \min\{\text{prio}(c_1(e)), \text{prio}(c_2(e))\}$.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

Binärer Heap / 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 als Feld

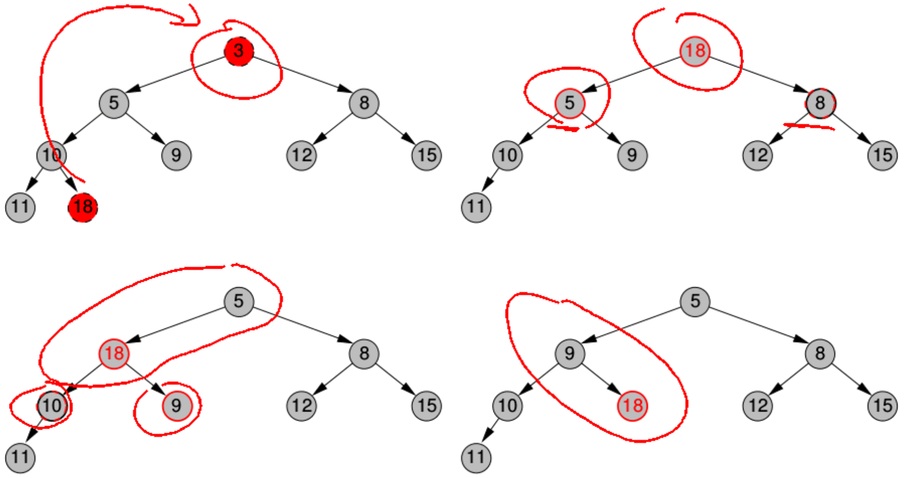
deleteMin()

- 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: $O(\log n)$

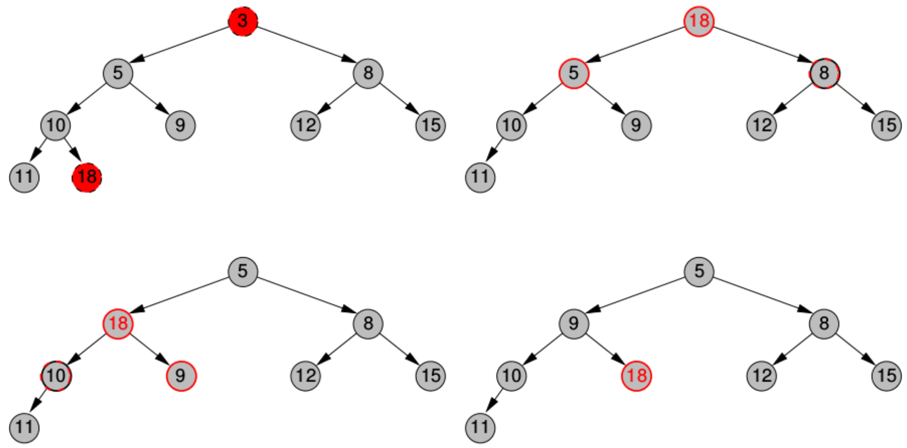
Binärer Heap als Feld

```
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()



Heap - siftDown()



Binärer Heap / Aufbau

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

- naiv:

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

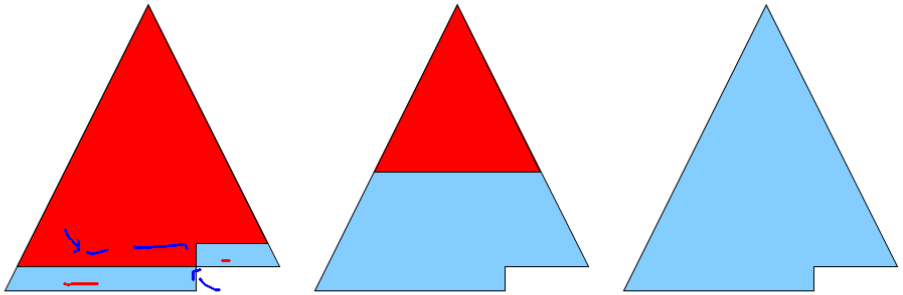
$\Rightarrow$ Laufzeit: $\Theta(n \log n)$

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.

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

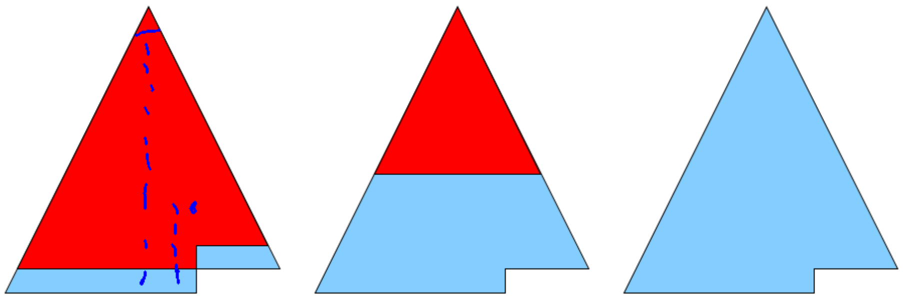
$$\begin{aligned}
\sum_{j \geq 1} j \cdot 2^{-j} &= \sum_{j \geq 1} 2^{-j} + \sum_{j \geq 2} 2^{-j} + \sum_{j \geq 3} 2^{-j} + \ldots \\
&= 1 \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{2} \cdot \sum_{j \geq 1} 2^{-j} + \frac{1}{4} \cdot \sum_{j \geq 1} 2^{-j} + \ldots \\
&= (1 + 1/2 + 1/4 + \ldots) \sum_{j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

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.

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

$$\sum_{j\geq1} j\cdot 2^{-j} = \sum_{j\geq1} 2^{-j} + \sum_{j\geq2} 2^{-j} + \sum_{j\geq3} 2^{-j} + \ldots$$
$$= 1\cdot\sum_{j\geq1} 2^{-j} + \frac{1}{2}\cdot\sum_{j\geq1} 2^{-j} + \frac{1}{4}\cdot\sum_{j\geq1} 2^{-j} + \ldots$$
$$= (1+1/2+1/4+\ldots)\sum_{j\geq1} 2^{-j} = 2\cdot 1 = 2$$

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.

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

$$\sum_{j\geq1} j\cdot 2^{-j} = \sum_{j\geq1} 2^{-j} + \sum_{j\geq2} 2^{-j} + \sum_{j\geq3} 2^{-j} + \ldots$$

$$\sum_{j=0} q^j = \frac{1}{1-q}$$

$$\sum_{j=0} q^j = q^0 + \sum_{j\geq0} q^j$$

$$= 1\cdot\sum_{j\geq1} 2^{-j} + \frac{1}{2}\cdot\sum_{j\geq1} 2^{-j} + \frac{1}{4}\cdot\sum_{j\geq1} 2^{-j} + \ldots$$
$$= (1+1/2+1/4+\ldots)\sum_{j\geq1} 2^{-j} = 2\cdot 1 = 2$$

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.

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

$$\sum_{j\geq1} j\cdot 2^{-j} = \sum_{j\geq1} 2^{-j} + \sum_{j\geq2} 2^{-j} + \sum_{j\geq3} 2^{-j} + \ldots$$
$$= 1\cdot\sum_{j\geq1} 2^{-j} + \frac{1}{2}\cdot\sum_{j\geq1} 2^{-j} + \frac{1}{4}\cdot\sum_{j\geq1} 2^{-j} + \ldots$$
$$= (1+1/2+1/4+\ldots)\sum_{j\geq1} 2^{-j} = 2\cdot 1 = 2$$

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

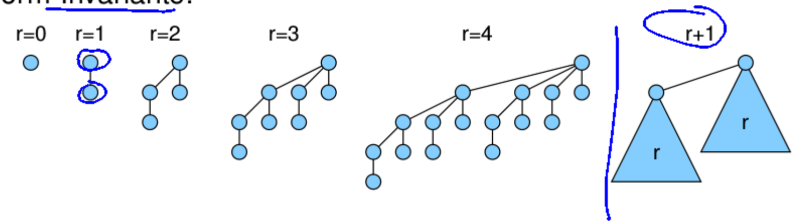
$\Rightarrow$ sortiertes Array entsteht von hinten,
ansteigende Sortierung kann mit Max-Heap erzeugt werden

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

Binomial-Bäume

Binomial Heaps bestehen aus Binomial-Bäumen

- Form-Invariante:



- Heap-Invariante:

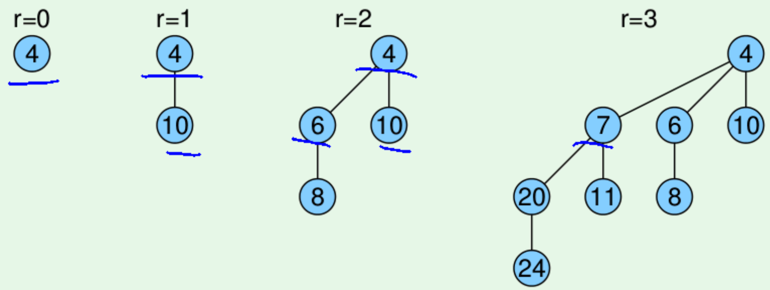
$$\text{prio(Vater)} \leq \text{prio(Kind)}$$

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

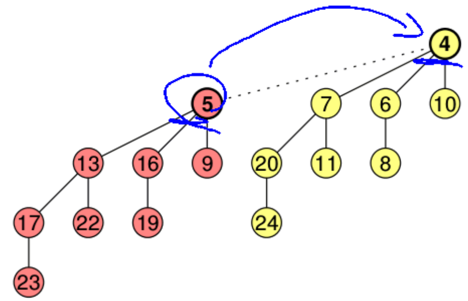
Beispiel

Korrekte Binomial-Bäume:

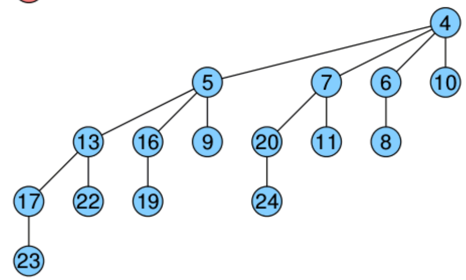


Binomial-Baum: Merge

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

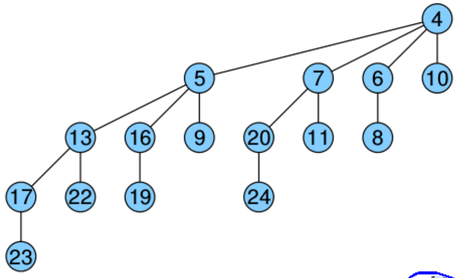


aus zwei B_{r-1} wird ein B_r

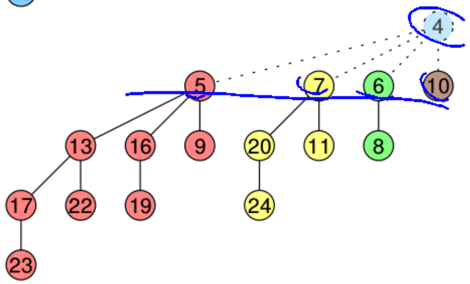


Binomial-Baum: Löschen der Wurzel (deleteMin)

aus einem B_r



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



Binomial-Baum: Knotenanzahl

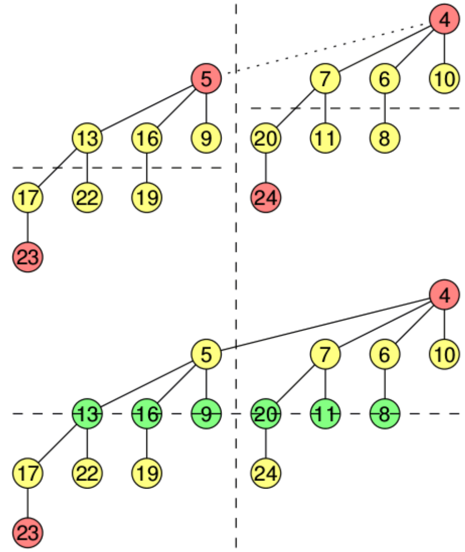
B_r hat auf Level $k \in \{0, \ldots, r\}$ genau $\binom{r}{k}$ Knoten

Warum?

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

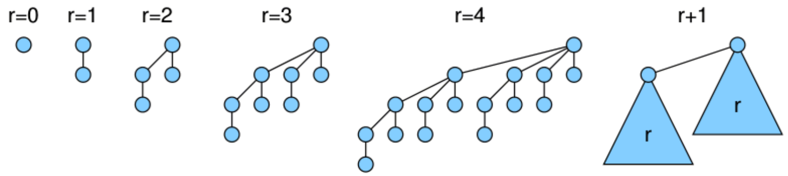
$$\binom{r}{k} = \binom{r-1}{k-1} + \binom{r-1}{k}$$

Insgesamt: B_r hat 2^r Knoten



Binomial-Bäume

Eigenschaften von Binomial-Bäumen:

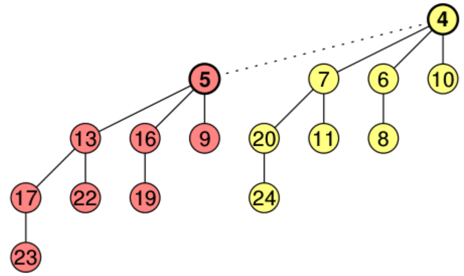


Binomial-Baum vom Rang r

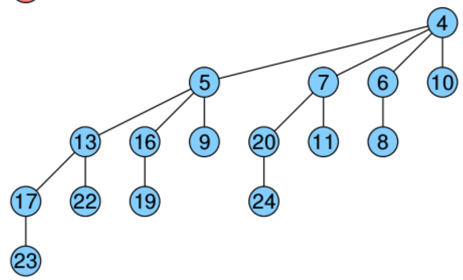
- hat Höhe r (gemessen in Kanten)
- hat maximalen Grad r (Wurzel)
- hat auf Level $\ell \in \{0, \ldots, r\}$ genau $\binom{r}{\ell}$ Knoten
- hat $\sum_{\ell=0}^{r} \binom{r}{\ell} = 2^r$ Knoten
- zerfällt bei Entfernen der Wurzel in r Binomial-Bäume von Rang 0 bis $r-1$

Binomial-Baum: Merge

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



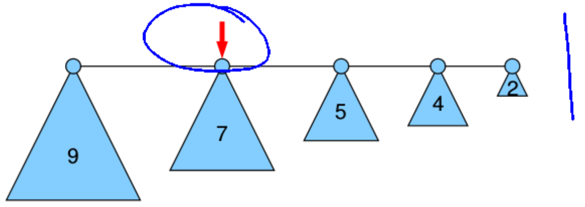
aus zwei B_{r-1} wird ein B_r



Binomial Heap

Binomial Heap:

- verkettete Liste von Binomial-Bäumen
- pro Rang maximal 1 Binomial-Baum
- Zeiger auf Wurzel mit minimalem Prioritätswert

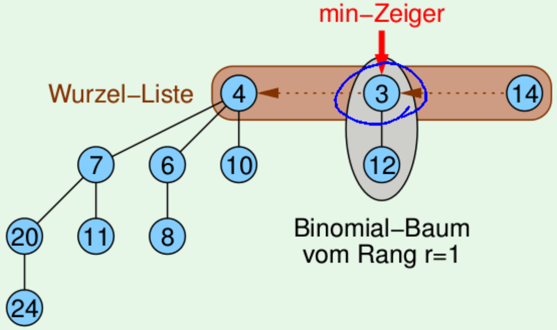


(Zahlen: Ränge)

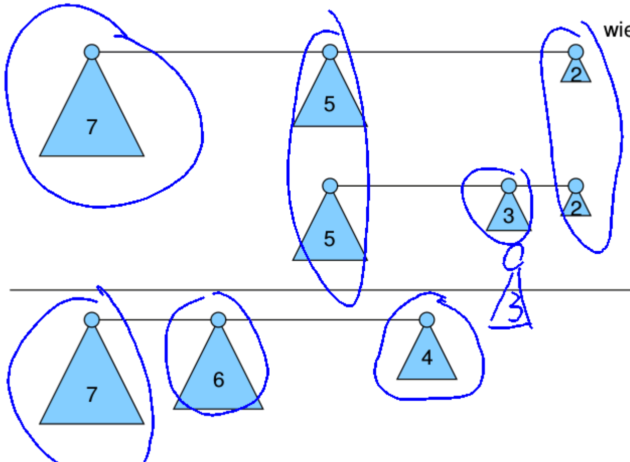
Binomial Heap

Beispiel

Korrekter Binomial Heap:



Merge von zwei Binomial Heaps



wie Binäraddition:

10100100

+ 101100

= 11010000

Aufwand für Merge: $O(\log n)$

Binomial Heaps

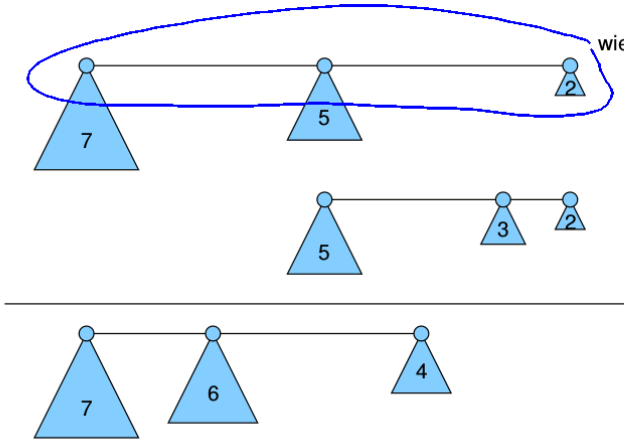
B_i: Binomial-Baum mit Rang i

Operationen:

- merge: $O(\log n)$
- insert(e): Merge mit B_0, Zeit $O(\log n)$
- min(): spezieller Zeiger, Zeit $O(1)$
- deleteMin():
  sei das Minimum in B_i,
  durch Löschen der Wurzel zerfällt der Binomialbaum in
  $B_0, \ldots, B_{i-1}$
  Merge mit dem restlichen Binomial Heap kostet $O(\log n)$

Merge von zwei Binomial Heaps

wie Binäraddition:



10100100

+ 101100

= 11010000

Aufwand für Merge: $\mathcal{O}(\log n)$

Binomial Heaps

Weitere Operationen:

- decreaseKey(h, k): siftUp-Operation in Binomial-Baum für das Element, auf das h zeigt, dann ggf. noch min-Zeiger aktualisieren

  Zeit: $\mathcal{O}(\log n)$

- remove(h): Sei e das Element, auf das h zeigt. Setze prio(e) $= -\infty$ und wende siftUp-Operation auf e an bis e in der Wurzel, dann weiter wie bei deleteMin

  Zeit: $\mathcal{O}(\log n)$

Bessere Laufzeit mit Fibonacci-Heaps

Fibonacci-Heaps

Verbesserung von Binomial Heaps mit folgenden Kosten:

- min, insert, merge: $\mathcal{O}(1)$ (worst case)
- decreaseKey: $\mathcal{O}(1)$ (amortisiert)
- deleteMin, remove: $\mathcal{O}(\log n)$ (amortisiert)

Wir werden darauf bei den Graph-Algorithmen zurückgreifen.

Vergleich Wörterbuch / Suchstruktur

- S: Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

- S.insert(Elem e): $S = S \cup \{e\}$
- S.remove(Key k): $S = S \setminus \{e\}$, wobei e das Element mit key(e) $== k$ ist
- S.find(Key k): (Wörterbuch) gibt das Element $e \in S$ mit key(e) $== k$ zurück, falls es existiert, sonst null
- S.locate(Key k): (Suchstruktur) gibt das Element $e \in S$ mit minimalem Schlüssel key(e) zurück, für das key(e) $\geq k$