5. Vorrangwarteschlangen — Priority Queues

Priority Queues unterstützen die Operationen
Insert(), *Delete()*, *ExtractMin()*, *FindMin()*, *DecreaseKey()*, *Merge()*.

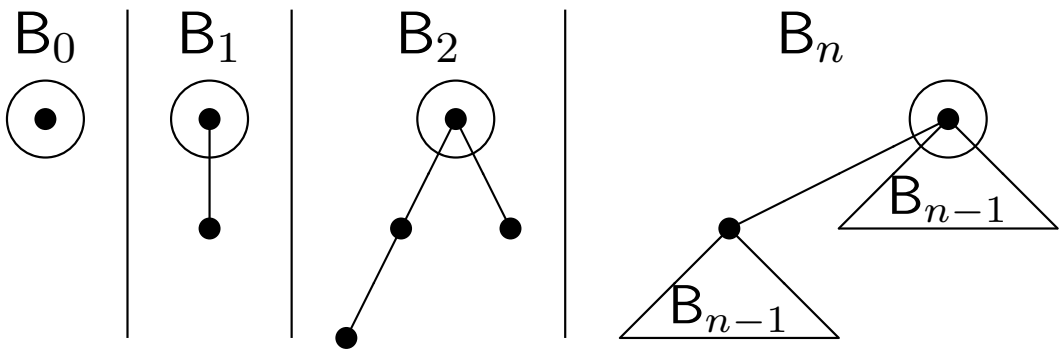
Priority Queues per se sind nicht für *IsElement()*-Anfragen, also zum Suchen geeignet. Falls benötigt, muss dafür eine passende Wörterbuch-Datenstruktur parallel mitgeführt werden.

5.1 Binomial Queues (binomial heaps)

Binomialwarteschlangen (Binomial Queues/Binomial Heaps) werden mit Hilfe von Binomialbäumen konstruiert.

Definition 39

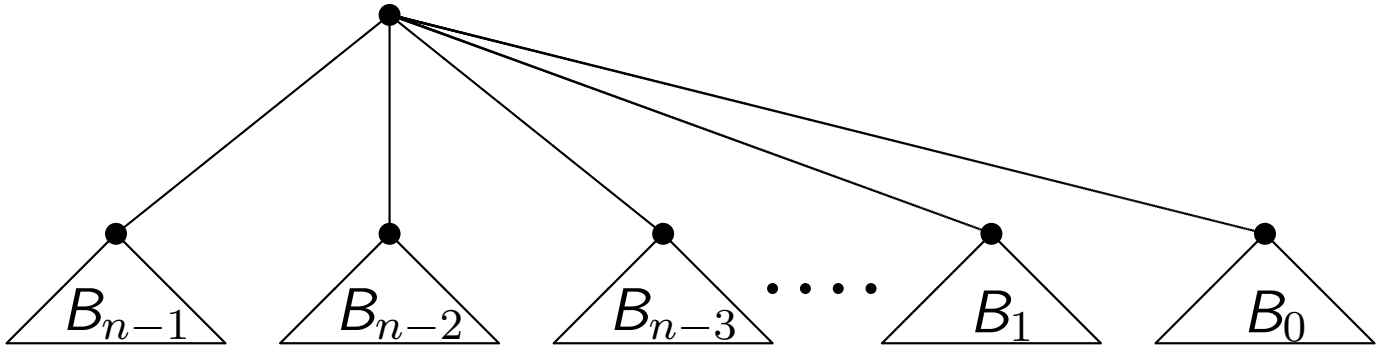
Die Binomialbäume B_n, $n \geq 0$, sind rekursiv wie folgt definiert:



Achtung: Binomialbäume sind offensichtlich keine Binärbäume!

Lemma 40

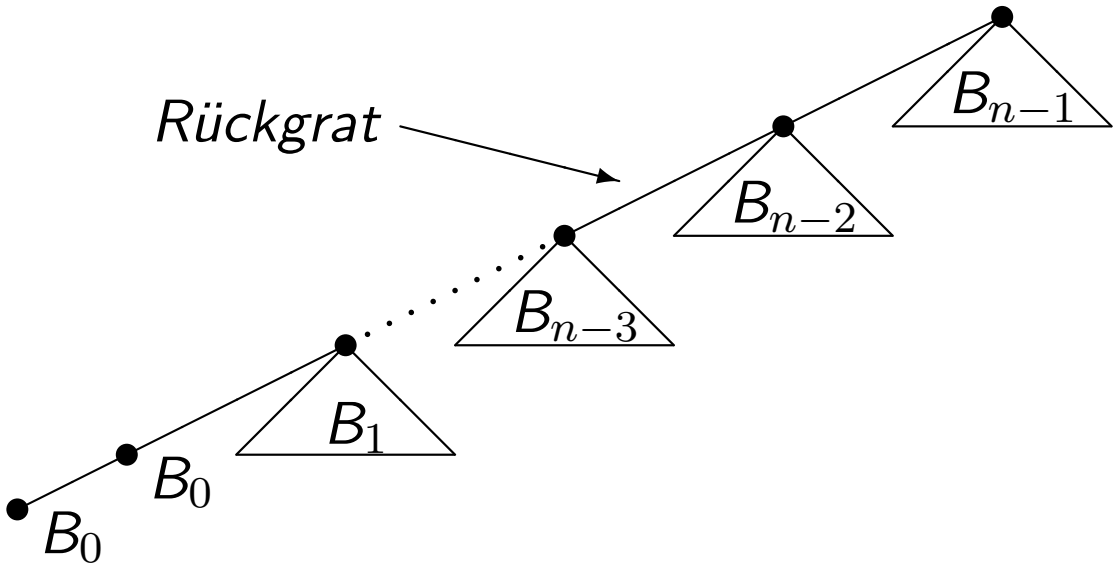
Ein B_n lässt sich wie folgt zerlegen:



Erste Zerlegung von B_n: der Breite nach

Lemma 40

Ein B_n lässt sich wie folgt zerlegen:



Zweite Zerlegung von B_n: der Tiefe nach

Satz 41

Für den Binomialbaum B_n gilt:

1. *B_n hat 2^n Knoten.*
2. *Die Wurzel von B_n hat Grad n.*
3. *B_n hat Höhe/Tiefe n.*
4. *B_n hat $\binom{n}{i}$ Knoten in Tiefe i.*

Beweis:

- zu 1: Induktion unter Verwendung der Definition
- zu 2: Siehe erste Zerlegung von Binomialbäumen
- zu 3: Siehe zweite Zerlegung von Binomialbäumen
- zu 4: Induktion über n:

  (I) $n = 0 : B_0$ hat 1 Knoten in Tiefe $i = 0$ und 0 Knoten in Tiefe $i > 0$, also $\binom{0}{i}$ Knoten in Tiefe i.

  (II) Ist N_i^n die Anzahl der Knoten in Tiefe i im B_n, so gilt unter der entsprechenden Induktionsannahme für B_n, dass

$$N_i^{n+1} = N_i^n + N_{i-1}^n = \binom{n}{i} + \binom{n}{i-1} = \binom{n+1}{i},$$

  woraus wiederum per Induktion die Behauptung folgt.

□

Bemerkung:

Eine mögliche Implementierung von Binomialbäumen ist das Schema „Pointer zum ersten Kind und Pointer zum nächsten Geschwister“.

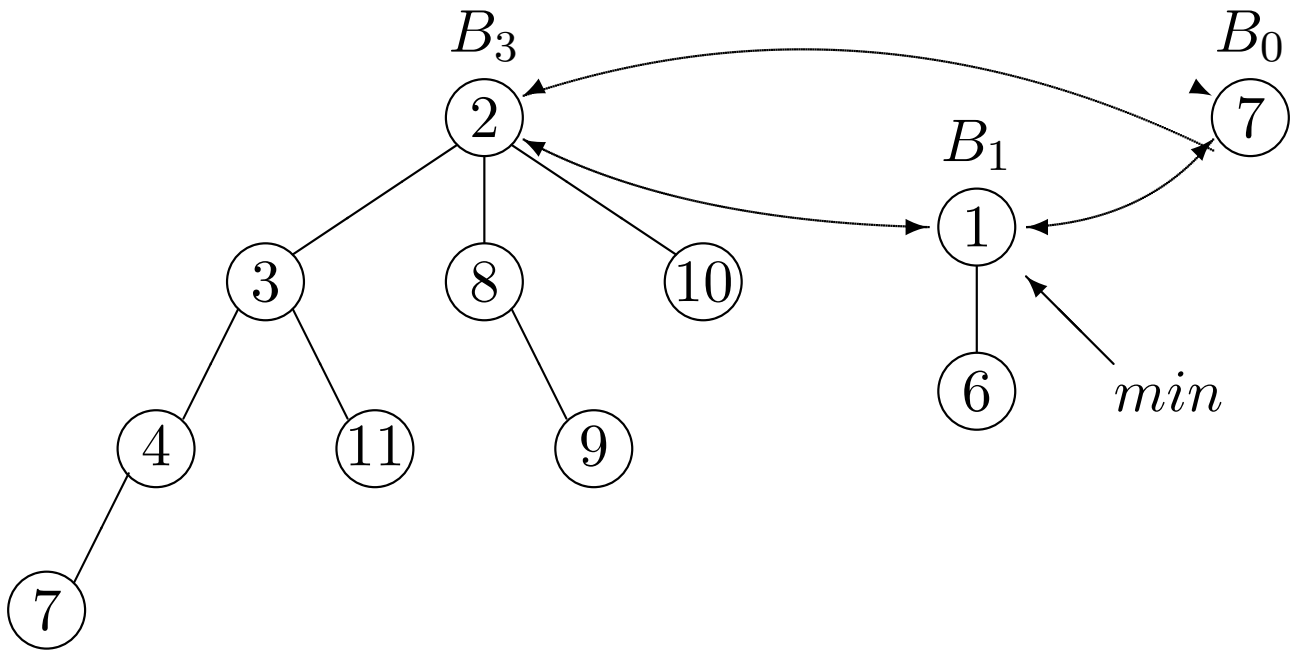
Definition 42

Eine Binomial Queue mit n Elementen wird wie folgt aufgebaut:

1. Betrachte die Binärdarstellung von n.
2. Für jede Position i mit einem 1-Bit wird ein Binomialbaum B_i benötigt (der 2^i Knoten hat).
3. Verbinde die (Wurzeln der) Binomialbäume in einer doppelt verketteten zirkulären Liste.
4. Beachte, dass innerhalb jedes Binomialbaums die Heap-Bedingung erfüllt sein muss. Dadurch enthält die Wurzel eines jeden Binomialbaums gleichzeitig sein minimales Element.
5. Richte einen Min-Pointer auf das Element in der Wurzel-Liste mit minimalem Schlüssel ein.

Beispiel 43

Beispiel: $n = 11 = (1011)_2$**:**



Operationen für Binomial Queues:

- *IsElement*: Die Heap-Bedingung wirkt sich nur auf die Anordnung der Datenelemente innerhalb jedes einzelnen Binomialbaums aus, regelt aber nicht, in welchem Binomialbaum ein gegebenes Element gespeichert ist. Tatsächlich kann ein Element in jedem der vorhandenen Binomialbäume stehen. Das Suchen ist hier nicht effizient implementierbar, denn im worst-case müsste jedes Element der Binomialbäume angeschaut werden.

  Also wären zum Suchen eines Elements schlimmstenfalls $2^0 + 2^1 + \cdots + 2^{\lfloor \log n \rfloor - 1} = \Theta(n)$ Elemente zu betrachten.

  Daher wird eine gesonderte Datenstruktur, etwa ein Suchbaum, für die *IsElement*-Operation verwendet. Damit ist Suchen mit Zeitaufwand $\mathcal{O}(\log n)$ möglich.

Operationen für Binomial Queues:

- *Merge*: Das Vorgehen für das Merge (disjunkte Vereinigung) zweier Binomial Queues entspricht genau der Addition zweier Binärzahlen: Ein einzelnes B_i wird übernommen, aus zwei B_i's wird ein B_{i+1} konstruiert. Damit die Heap-Bedingung erhalten bleibt, wird als Wurzel des entstehenden B_{i+1} die Wurzel der beiden B_i mit dem kleineren Schlüssel genommen.

  Allerdings kann ein solcher „Übertrag“ dazu führen, dass im nächsten Verschmelzungsschritt drei B_{i+1} zu verschmelzen sind. Dann wird unter Beachtung obiger Regel ein B_{i+2} gebildet und einer der B_{i+1} unverändert übernommen.

Algorithmus:

for $i := 0, 1, 2, 3, \ldots$ **do**
  if ($\exists$ genau 3 B_is) **then**
    verbinde zwei der B_i's zu einem B_{i+1} und behalte das dritte B_i
  elif ($\exists$ genau 2 B_is) **then**
    verbinde sie zu einem B_{i+1}
  elif ($\exists$ genau ein B_i) **then**
    übernimm es
  fi
od

Zeitkomplexität:

$$\boxed{\mathcal{O}(\log n) = \mathcal{O}(\log(n_1 + n_2))}$$

Operationen für Binomial Queues:

- *Insert*: Die *Insert*-Operation wird einfach durch eine *Merge*-Operation mit einem B_0 implementiert.

Beispiel 44

1	0	0	1	0	1	1	1
(B_7)			(B_4)		(B_2)	(B_1)	(B_0)
							1
1	0	0	1	1	0	0	0

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

Operationen für Binomial Queues:

- Initialisierung einer BQ durch n sukzessive *Insert*-Operationen: Hier ergäbe die obige Abschätzung einen Gesamtaufwand von $\mathcal{O}(n \log n)$, was allerdings schlecht abgeschätzt ist.

Wir sind an den Kosten zum sukzessiven Aufbau einer Binomial Queue mit n Elementen interessiert, die übrigens identisch sind zum Aufwand für das binäre zählen von 0 bis n, wenn jeder Zählschritt und jeder Übertrag jeweils eine Zeiteinheit kosten:

$$\underbrace{\underbrace{0+1}_{1}+1}_{2\text{ Schritte}}+1+1+1$$

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{0+1}_{1}+1}_{2\text{ Schritte}}+1}_{3}+1}_{4}+1}_{5}$$

Sei a_n die Anzahl der Schritte (Einfügen des neuen Elements und anschließende Verschmelzungen) beim Mergen eines Knotens zu einer Queue mit $n-1$ Elementen. Dann gilt für a_n:

n	a_n
1	1
2	2
... 4	1 3
... 8	1 2 1 4
... 16	1 2 1 3 1 2 1 5
... 32	1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 6

Man erkennt sofort, dass jede Zeile (außer den beiden ersten) doppelt so lange ist wie die vorhergehende und dabei die Folge aller vorhergehenden Zeilen enthält, wobei das letzte Element noch um eins erhöht ist.

Damit ergibt sich für den Gesamtaufwand der sukzessiven Erzeugung einer Binomial Queue mit n Elementen

$$\begin{aligned} T_n &= \sum_{i=1}^{n} a_i \\ &= n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \cdots \\ &\leq 2n\,. \end{aligned}$$

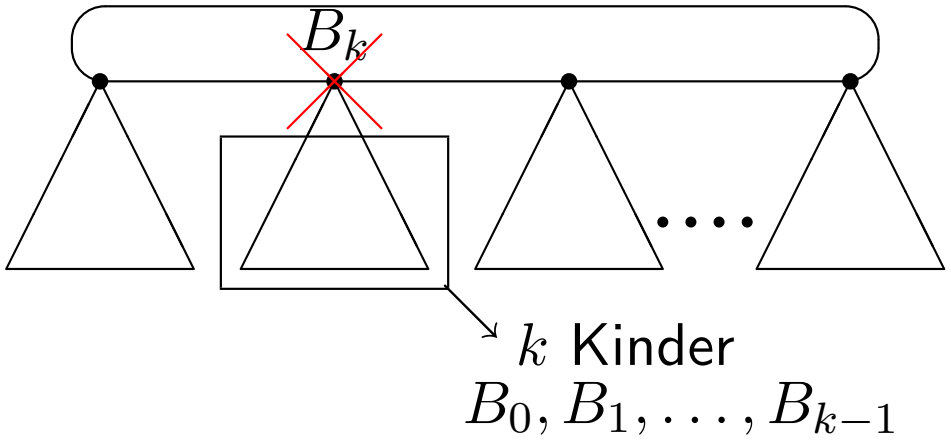
Operationen für Binomial Queues:

- *FindMin*: Diese Operation ist trivial ausführbar, denn es ist ein Pointer auf das minimale Element gegeben.

Zeitkomplexität: $\mathcal{O}(1)$

Operationen für Binomial Queues:

- *ExtractMin*:Das Minimum ist auf Grund der Heapbedingung Wurzel eines Binomialbaums B_k in der Liste. Wird es gelöscht, so zerfällt der Rest in k Teilbäume $B_0, B_1, \ldots, B_{k-1}$:



Die Teilbäume $B_0, B_1, \ldots, B_{k-1}$ sind alle zur verbleibenden Queue zu mergen. Außerdem muss der Min-Pointer aktualisiert werden.
Der Zeitaufwand für die *ExtractMin*-Operation ist daher:

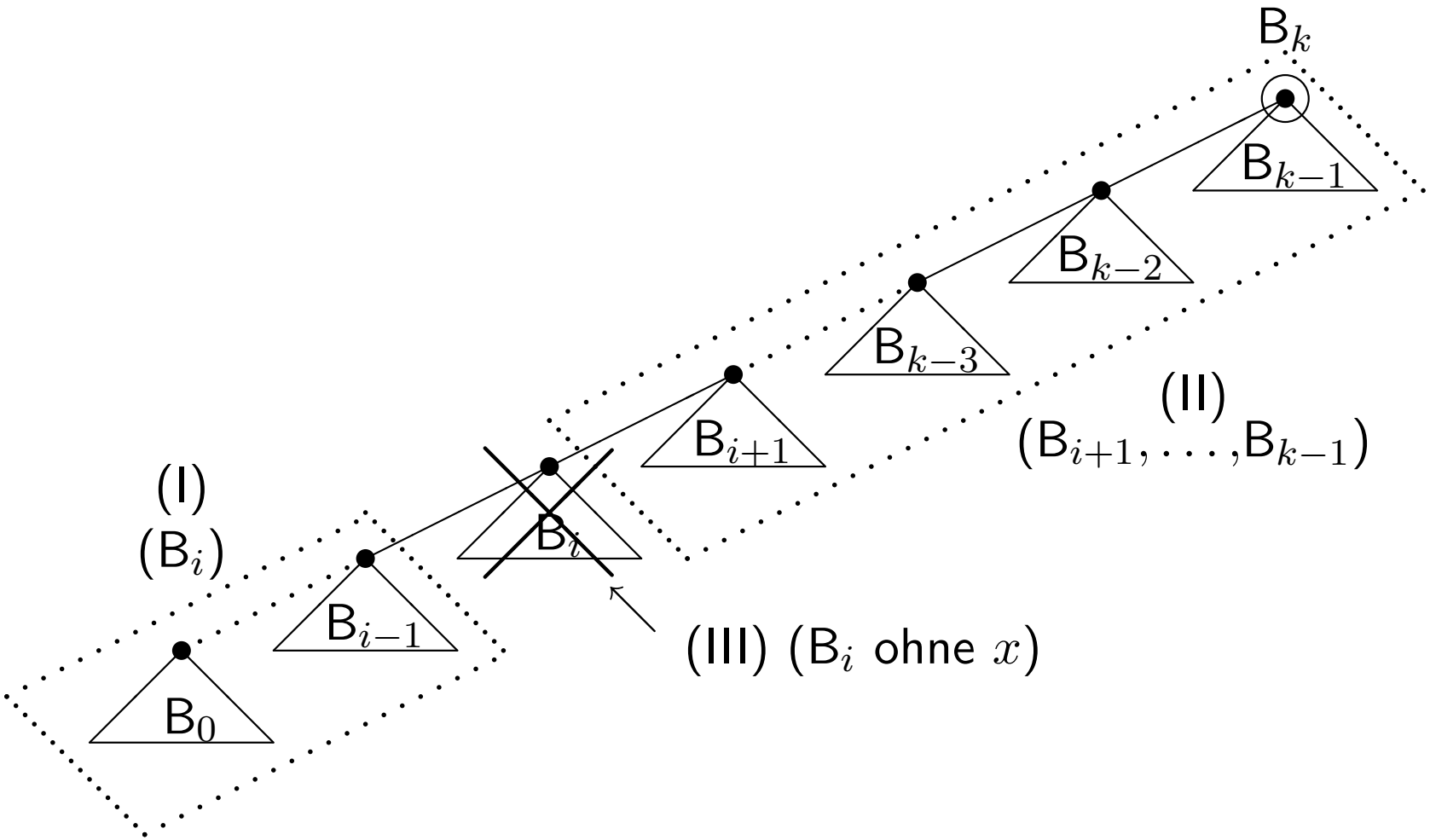
$$\boxed{\mathcal{O}(\log n)}$$

Operationen für Binomial Queues:

- *Delete*: Lösche Knoten x:

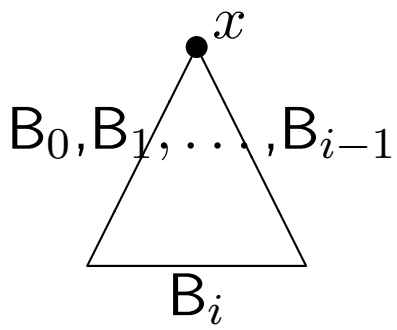
1. Fall: x ist Min-Wurzel: s.o.
2. Fall: x ist eine andere Wurzel in der Wurzelliste. Analog zu oben, ohne den Min-Pointer zu aktualisieren.
3. Fall: x ist nicht Wurzel eines Binomialbaumes.

Angenommen, x ist in einem B_k enthalten:



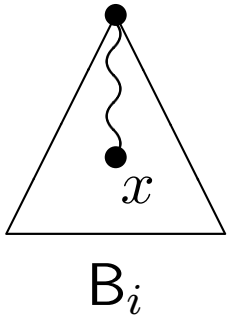
Es ergeben sich im 3. Fall folgende Unterfälle:

(3a) x ist Wurzel von B_i:



Verfahre wie im 2. Fall.

(3b) x ist nicht Wurzel des B_i:



Wiederhole rekursiv Fall 3, bis Fall (3a) eintritt.

Insgesamt muss die Binomial Queue ohne B_i mit einer aus einer Teilmenge von $\{B_0, \ldots, B_{k-1}\}$ bestehenden Binomial Queue vereinigt werden.

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

Operationen für Binomial Queues:

- *DecreaseKey*: Verkleinere $k(x)$

1. Fall: x ist die Min-Wurzel: keine Strukturänderung nötig
2. Fall: x ist eine andere Wurzel: keine Strukturänderung nötig, ggf. Aktualisierung des Min-Pointers
3. Fall: Sonst wie *Delete*(x), aber Einfügen/Merge des Baumes mit Wurzel x und dem neuen reduzierten Schlüssel $k(x)$ in die Wurzelliste.

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

Satz 45

Binomial Queues haben für die Operationen Insert, Delete, ExtractMin, FindMin, Merge und Initialize jeweils worst-case-Kosten von

$$\mathcal{O}(\log n)\,.$$

Die ursprüngliche Literatur zu Binomial Queues:

Vuillemin, Jean:
A data structure for manipulating priority queues
Commun. ACM **21**(4), pp. 309–315 (1978)
http://doi.acm.org/10.1145/359460.359478

Brown, M.R.:
Implementation and analysis of binomial queue algorithms
SIAM J. Comput. **7**(3), pp. 298–319 (1978)
http://siamdl.aip.org/dbt/dbt.jsp?KEY=SMJCAT&Volume=7&Issue=3

5.2 Fibonacci-Heaps

Vorbemerkungen:

1. Fibonacci-Heaps stellen eine Erweiterung der Binomial Queues und eine weitere Möglichkeit zur Implementierung von Priority Queues dar. Die amortisierten Kosten für die Operationen *Delete()* und *ExtractMin()* betragen hierbei $\mathcal{O}(\log n)$, die für alle anderen Heap-Operationen lediglich $\mathcal{O}(1)$.
   Natürlich können die worst-case-Gesamtkosten für n *Insert* und n *ExtractMin* nicht unter $\Omega(n \log n)$ liegen, denn diese Operationen zusammengenommen stellen einen Sortieralgorithmus mit unterer Schranke $\Omega(n \log n)$ dar.

5.2 Fibonacci-Heaps

Vorbemerkungen:

2. Die Verwendung von Fibonacci-Heaps erlaubt eine Verbesserung der Komplexität der Algorithmen für minimale Spannbäume sowie für Dijkstra's kürzeste Wege-Algorithmus, denn diese verwenden relativ häufig die *DecreaseKey*-Operation, welche durch Fibonacci-Heaps billig zu implementieren ist. Bei einem Algorithmus, bei dem die *Delete*- und *ExtractMin*-Operationen nur einen geringen Anteil der Gesamtoperationen darstellen, können Fibonacci-Heaps asymptotisch schneller als Binomial Queues sein.

5.2.1 Die Datenstruktur

- Die Schlüssel sind an den Knoten von Bäumen gespeichert.
- Jeder Knoten hat folgende Größen gespeichert:
  - Schlüssel und Wert
  - Rang (= Anzahl der Kinder)
  - Zeiger zum ersten Kind, zum Vater (NIL im Falle der Wurzel), zu doppelt verketteter Liste der Kinder
  - Markierung $\in \{0, 1\}$ (außer Wurzel)

Bemerkung: Die doppelte Verkettung der Kinder- bzw. Wurzellisten in Heaps erlaubt das Löschen eines Listeneintrages in Zeit $\mathcal{O}(1)$.

Binomial-Queue vs. Fibonacci-Heap:

- Beide sind Wälder von Bäumen, innerhalb derer die Heap-Bedingung gilt.
- *a priori* keine Einschränkung für die Topologie der Bäume, aber ohne *Delete*- oder *DekreaseKey*-Operationen (u.ä.) bleibt ein Binomialwald ein Binomialwald und ein Fibonacci-Heap ein Wald aus Binomialbäumen.
- Fibonacci-Heaps kennen keine Invariante der Form „Nur Bäume verschiedenen Wurzel-Rangs“.
- Fibonacci-Heaps: lazy merge.
- Fibonacci-Heaps: lazy delete.

Überblick:

Operationen	worst case	amortisiert
Insert	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Merge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
FindMin	$\mathcal{O}(1)$	$\mathcal{O}(1)$
DecreaseKey	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Delete	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
ExtractMin	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$