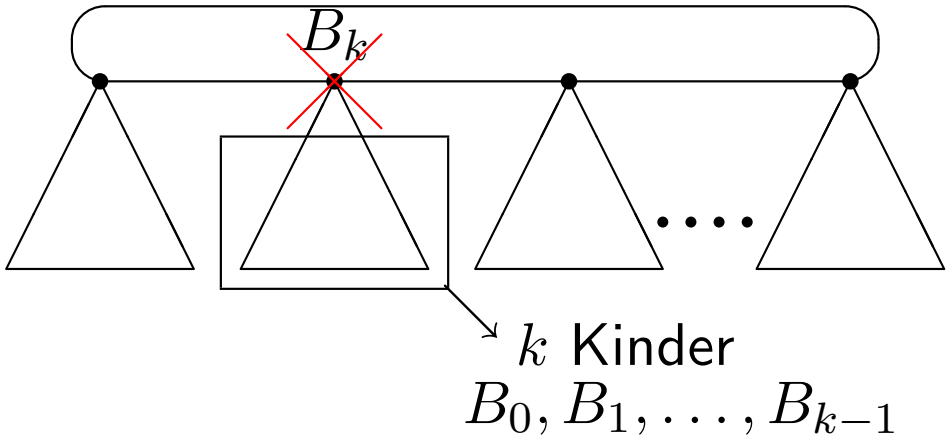
Operationen für Binomial Queues:

- *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:

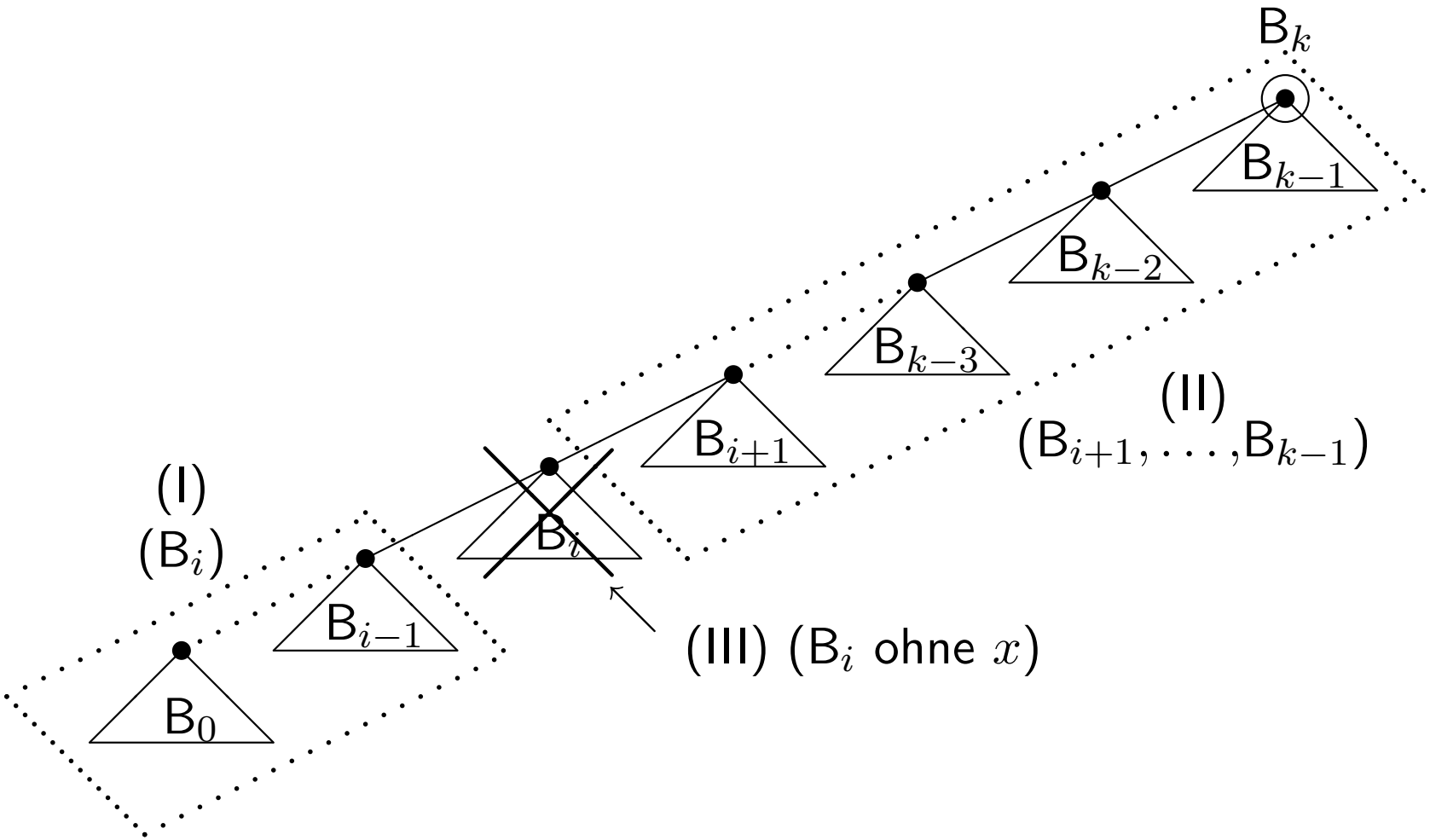
$$\boxed{\mathcal{O}(\log n)}$$

Operationen für Binomial Queues:

- *Delete*: Lösche Knoten x:

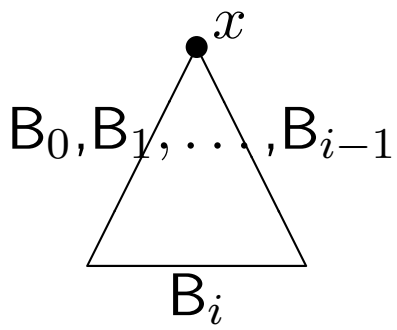
1. Fall: x ist Min-Wurzel: s.o.
2. Fall: x ist eine andere Wurzel in der Wurzelliste. Analog zu oben, ohne den Min-Pointer zu aktualisieren.
3. Fall: x ist nicht Wurzel eines Binomialbaumes.

Angenommen, x ist in einem B_k enthalten:



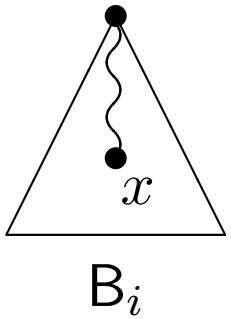
Es ergeben sich im 3. Fall folgende Unterfälle:

(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_k 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 46

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:

Jean Vuillemin:

A data structure for manipulating priority queues,
Commun. ACM **21**(4), pp. 309–315 (1978)

Mark R. Brown:

Implementation and analysis of binomial queue algorithms,
SIAM J. Comput. **7**(3), pp. 298–319 (1978)

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.

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:

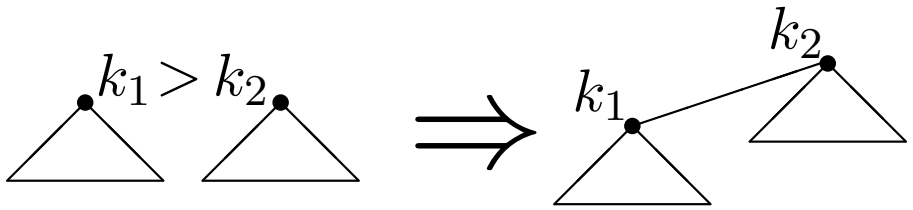
- Beide sind Wälder von Bäumen, innerhalb derer die Heap-Bedingung gilt.
- *a priori* keine Einschränkung für die Topologie der Bäume, aber ohne *Delete*- oder *DekreaseKey*-Operationen (u.ä.) bleibt ein Binomialwald ein Binomialwald und ein Fibonacci-Heap ein Wald aus Binomialbäumen.
- Fibonacci-Heaps 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	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Delete	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
ExtractMin	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Linken von Bäumen:

Zwei Bäume desselben Wurzel-Rangs werden unter Einhaltung der Heap-Bedingung verbunden. Sind k_1 und k_2 die Wurzeln der zwei zu linkenden Bäume, so wird ein neuer Baum aufgebaut, dessen Wurzel bzgl. der Schlüssel das Minimum von k_1 und k_2 ist. Sei dies k_2, dann erhält der Baum mit Wurzel k_2 als zusätzlichen Unterbaum den Baum mit Wurzel k_1, d.h. der Rang von k_2 erhöht sich um 1.



Dies ist gerade der konstruktive Schritt beim Aufbau von Binomialbäumen.

Zur Verbesserung der amortisierten Zeitschranken:

Kaskadierendes Abschneiden:

- Ein Knoten ist im „Anfangszustand“, wenn
  - i) er durch einen Linkschritt Kind eines anderen Knotens wird oder
  - ii) er in die Wurzelliste eingefügt wird.

  Sein Markierungsbit wird in jedem dieser Fälle zurückgesetzt.
- Wenn ein Knoten ab seinem Anfangszustand zum zweitenmal ein Kind verliert, wird er samt seines Unterbaums aus dem aktuellen Baum entfernt und in die Wurzelliste eingefügt.
- Der kritische Zustand (ein Kind verloren) wird durch Setzen des Markierungsbits angezeigt.

Algorithmus:

co x verliert Kind **oc**
while x markiert **do**
    entferne x samt Unterbaum
    entferne Markierung von x
    füge x samt Unterbaum in Wurzelliste ein
    if $P(x)$=NIL **then return fi**
    $x := P(x)$ **co** (Vater von x) **oc**
od
if x nicht Wurzel **then** markiere x **fi**

Operationen in Fibonacci-Heaps:

1. *Insert*(x): Füge B_0 (mit dem Element x) in die Wurzelliste ein. Update Min-Pointer.

   Kosten $\mathcal{O}(1)$

2. *Merge()*: Verbinde beide Listen und aktualisiere den Min-Pointer.

   Kosten $\mathcal{O}(1)$

3. *FindMin()*: Es wird das Element ausgegeben, auf das der Min-Pointer zeigt. Dabei handelt es sich sicher um eine Wurzel.

   Kosten $\mathcal{O}(1)$

Operationen in Fibonacci-Heaps:

4. *Delete*(x)
   - i) Falls x Min-Wurzel, *ExtractMin*-Operator (s.u.) benutzen
   - ii) Sonst:

füge Liste der Kinder von x in die Wurzelliste ein; lösche x
if $P(x)$=NIL **then return fi** **co** x ist Wurzel **oc**
while true do
    $x := P(x)$
    if $P(x)$=NIL **then return fi** **co** x ist Wurzel **oc**
    if Markierung(x)=0 **then** Markierung(x):=1; **return**
    else
        hänge x samt Unterbaum in Wurzelliste
        entferne Markierung von x (da x nun Wurzel)
    fi
od

Kosten: $\mathcal{O}(1 + \#\text{kask. Schnitte})$

Operationen in Fibonacci-Heaps:

5. *ExtractMin()*: Diese Operation hat auch Aufräumfunktion und ist daher recht kostspielig. Sei x der Knoten, auf den der Min-Pointer zeigt.

   entferne x aus der Liste
   konkateniere Liste der Kinder von x mit der Wurzelliste
   while $\exists \geq 2$ Bäume mit gleichem Wurzel-Rang i **do**
       erzeuge Baum mit Wurzel-Rang $i+1$
   od
   update Min-Pointer

Man beachte, dass an jedem Knoten, insbesondere jeder Wurzel, der Rang gespeichert ist. Zwar vereinfacht dies die Implementierung, doch müssen noch immer Paare von Wurzeln gleichen Rangs effizient gefunden werden.

Wir verwenden dazu ein Feld (Array), dessen Positionen je für einen Rang stehen. Die Elemente sind Zeiger auf eine Wurzel dieses Rangs. Es ist garantiert, dass ein Element nur dann unbesetzt ist, wenn tatsächlich keine Wurzel entsprechenden Rangs existiert. Nach dem Entfernen des Knoten x aus der Wurzelliste fügen wir die Kinder eines nach dem anderen in die Wurzelliste ein und aktualisieren in jedem Schritt die entsprechende Feldposition. Soll eine bereits besetzte Position des Arrays beschrieben werden, so wird ein Link-Schritt ausgeführt und versucht, einen Pointer auf die neue Wurzel in die nächsthöhere Position im Array zu schreiben. Dies zieht evtl. weitere Link-Schritte nach sich. Nach Abschluss dieser Operation enthält der Fibonacci-Heap nur Bäume mit unterschiedlichem Wurzel-Rang.

Kosten: $\mathcal{O}(\text{max. Rang } + \#\text{Link-Schritte})$