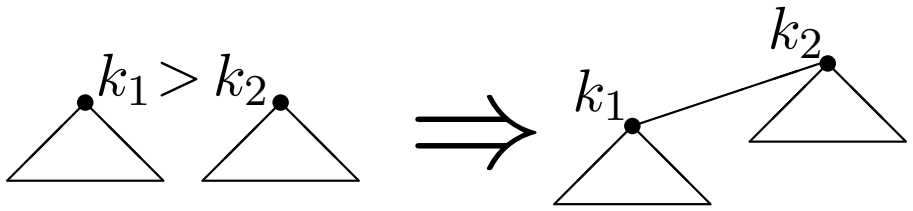
Linken von Bäumen:

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
markiere x

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

Operationen in Fibonacci-Heaps:

6. *DecreaseKey*(x, Δ):

   entferne x samt Unterbaum
   füge x mit Unterbaum in die Wurzelliste ein
   $k(x) := k(x) - \Delta$; aktualisiere Min-Pointer
   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: $\mathcal{O}(1 + \#\text{kask. Schnitte})$

Bemerkung:
Startet man mit einem leeren Fibonacci-Heap und werden ausschließlich die aufbauenden Operationen *Insert*, *Merge* und *FindMin* angewendet, so können nur Binomialbäume enstehen. In diesem natürlichen Fall liegt also stets ein Binomialwald vor, der jedoch i.a. nicht aufgeräumt ist. Das heißt, es existieren ev. mehrere Binomialbäume B_i desselben Wurzelgrads, die nicht paarweise zu B_{i+1}-Bäumen verschmolzen sind.

Dies geschieht erst bei der Operation *ExtractMin*. Man beachte, dass nur nach einer solchen Operation momentan ein Binomial Heap vorliegt, ansonsten nur ein Binomialwald.

Treten auch *DecreaseKey*- und/oder *Delete*-Operationen auf, so sind die Bäume i.a. keine Binomialbäume mehr.

5.2.2 Amortisierte Kostenanalyse für Fibonacci-Heaps

Kostenanalyse für Folgen von Operationen:

i) Summieren der worst-case-Kosten wäre zu pessimistisch. Der resultierende Wert ist i.a. zu groß.

ii) average-case:
- Aufwand für Analyse sehr hoch
- welcher Verteilung folgen die Eingaben?
- die ermittelten Kosten stellen keine obere Schranke für die tatsächlichen Kosten dar!

iii) amortisierte Kostenanalyse:

average-case-Analyse über worst-case-Operationenfolgen

Definition 46

Wir führen für jede Datenstruktur ein Bankkonto ein und ordnen ihr eine nichtnegative reelle Zahl bal, ihr Potenzial (bzw. Kontostand) zu. Die amortisierten Kosten für eine Operation ergeben sich als Summe der tatsächlichen Kosten und der Veränderung des Potenzials (Δbal), welche durch die Operation verursacht wird:

$t_i :=$ tatsächliche Kosten der i-ten Operation

$\Delta bal_i = bal_i - bal_{i-1}$: Potenzialveränderung durch i-te Operation

$a_i = t_i + \Delta bal_i$: amortisierte Kosten der i-ten Operation

$m =$ Anzahl der Operationen

Falls $bal_m \geq bal_0$ (was bei $bal_0 = 0$ stets gilt):

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Delta bal_i) = \sum_{i=1}^{m} t_i + bal_m - bal_0 \geq \sum_{i=1}^{m} t_i$$

In diesem Fall sind die amortisierten Kosten einer Sequenz eine obere Schranke für die tatsächlichen Kosten.

Anwendung auf Fibonacci-Heaps:
Wir setzen

$$bal := \#\text{ Bäume} + 2\#(\text{markierte Knoten} \neq \text{Wurzel})$$

Lemma 47

Sei x ein Knoten im Fibonacci-Heap mit Rang$(x) = k$. Seien die Kinder von x sortiert in der Reihenfolge ihres Anfügens an x. Dann ist der Rang des i-ten Kindes $\geq i - 2$.

Beweis:

Zum Zeitpunkt des Einfügens des i-ten Kindes ist Rang$(x) = i - 1$.

Das einzufügende i-te Kind hat zu dieser Zeit ebenfalls Rang $i - 1$.

Danach kann das i-te Kind höchstens eines seiner Kinder verloren haben

$$\Rightarrow \text{Rang des } i\text{-ten Kindes} \geq i - 2\,.$$

□

Satz 48

Sei x Knoten in einem Fibonacci-Heap, Rang$(x) = k$. Dann enthält der (Unter-)Baum mit Wurzel x mindestens F_{k+2} Elemente, wobei F_k die k-te Fibonacci-Zahl bezeichnet.

Da

$$F_{k+2} \geq \Phi^k$$

für $\Phi = (1+\sqrt{5})/2$ (Goldener Schnitt), ist der Wurzelrang also logarithmisch in der Baumgröße beschränkt.

Wir setzen hier folgende Eigenschaften der Fibonacci-Zahlen F_k voraus:

$$F_{k+2} \geq \Phi^k \text{ für } \Phi = \frac{1+\sqrt{5}}{2} \approx 1,618034\,;$$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i\,.$$

Beweis:

Sei f_k die minimale Anzahl von Elementen in einem „Fibonacci-Baum“ mit Wurzel-Rang k.
Aus dem vorangehenden Lemma folgt:

$$f_k \geq f_{k-2} + f_{k-3} + \ldots + f_0 + \underbrace{1}_{\text{1. Kind}} + \underbrace{1}_{\text{Wurzel}},$$

also (zusammen mit den offensichtlichen Anfangsbedingungen $f_0 = 1$ bzw. $f_1 = 2$ und den obigen Eigenschaften der Fibonacci-Zahlen):

$$f_k \geq F_{k+2}$$

□

Satz 49

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) *sind:*

1. *Insert:* $t = \mathcal{O}(1), \Delta bal = +1 \Rightarrow a = \mathcal{O}(1)$.
2. *Merge:* $t = \mathcal{O}(1), \Delta bal = 0 \Rightarrow a = \mathcal{O}(1)$.
3. *FindMin:* $t = \mathcal{O}(1), \Delta bal = 0 \Rightarrow a = \mathcal{O}(1)$.

Satz 49

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) *sind:*

4. *Delete (nicht Min-Knoten):*

   Einfügen der Kinder von x *in Wurzelliste:*
   $\Delta bal =$ *Rang*(x)
   Jeder kask. Schnitt erhöht $\#$*Bäume um* 1
   $\Delta bal = \#$*kask. Schnitte*
   Jeder Schnitt vernichtet eine Markierung
   $\Delta bal = -2 \cdot \#$*kask. Schnitte*
   Letzter Schnitt erzeugt ev. eine Markierung
   $\Delta bal = 2$
   $\Rightarrow$ *Jeder kask. Schnitt wird vom Bankkonto bezahlt und verschwindet amortisiert*

   $\Rightarrow a = \mathcal{O}(\log n)$

Satz 49

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) *sind:*

5. *ExtractMin:*

   Einfügen der Kinder von x *in Wurzelliste:*
   $\Delta bal =$ *Rang*(x)
   Jeder Link-Schritt verkleinert $\#$*Bäume um* 1
   $\Delta bal = -\#$*Link-Schritte*
   $\Rightarrow$ *Jeder Link-Schritt wird vom Bankkonto bezahlt und verschwindet amortisiert*

   $\Rightarrow a = \mathcal{O}(\log n)$

Satz 49

Die amortisierten Kosten der Fibonacci-Heap-Operationen ($a_i = t_i + \Delta bal_i$) *sind:*

6. *DecreaseKey: Es ist* $\Delta bal \leq 4 - (\#$*kaskadierende Schnitte*$)$.

$$\left.\begin{array}{l} \text{Jeder kask. Schnitt erhöht } \#\text{Bäume um } 1 \\ \Delta bal = \#\text{kask. Schnitte} \\ \text{Jeder Schnitt vernichtet eine Markierung} \\ \Delta bal = -2 \cdot \#\text{kask. Schnitte} \\ \text{Letzter Schnitt erzeugt ev. eine Markierung} \\ \Delta bal = 2 \end{array}\right\} \Rightarrow a = \mathcal{O}(1)$$

Beweis:

s.o. □

Literatur zu Fibonacci-Heaps:

Fredman, Michael L. and Robert Endre Tarjan:
Fibonacci heaps and their uses in improved network optimization algorithms
J. ACM **34**(3), pp. 596–615 (1987)
http://doi.acm.org/10.1145/28869.28874

Driscoll, James R. and Harold N. Gabow and Ruth Shrairman and Robert E. Tarjan:
Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation
Commun. ACM **31**(11), pp. 1343–1354 (1988)
http://doi.acm.org/10.1145/50087.50096