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 47

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

$\Delta bal_i = bal_i - bal_{i-1}$: Veränderung des Potenzials durch die 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 48

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 49

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 50

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 50

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 50

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 50

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:

Michael L. Fredman, Robert Endre Tarjan:
Fibonacci heaps and their uses in improved network optimization algorithms
Journal of the ACM **34**(3), pp. 596–615 (1987)

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

Mikkel Thorup:
Equivalence between priority queues and sorting
Journal of the ACM **54**(6), Article 28 (2007)

6. Sich selbst organisierende Datenstrukturen

6.1 Motivation

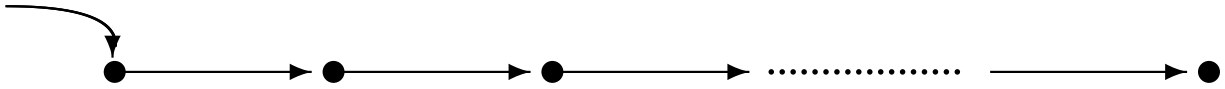
- einfach, wenig Verwaltungsoverhead
- effizient im amortisierten Sinn

6.2 Sich selbst organisierende lineare Listen

Bei der Implementierung von Wörterbüchern ist auf möglichst effiziente Weise eine Menge von bis zu n Elementen zu organisieren, dass die folgenden Operationen optimiert werden:

- Einfügen
- Löschen
- Vorhandensein überprüfen

Dabei soll die Effizienz der Implementierung bei beliebigen Sequenzen von m Operationen untersucht werden. Die Menge wird als unsortierte Liste dargestellt.



Operationen:

1. *Access*(i): Suchen des Elements i:
   Die Liste wird von Anfang an durchsucht, bis das gesuchte Element gefunden ist. Für das i-te Element betragen die tatsächlichen Kosten i Einheiten.
2. *Insert*(i): Einfügen des Elements i:
   Die Liste wird von Anfang an durchsucht, und wenn sie vollständig durchsucht wurde und das Element nicht enthalten ist, so wird es hinten angefügt. Dies erfordet $(n+1)$ Schritte, wenn die Länge der Liste n ist.
3. *Delete*(i): Löschen des Elements i:
   Wie bei *Access* wird die Liste zuerst durchsucht und dann das betreffende Element gelöscht. Dies kostet im Falle des i-ten Elements i Schritte.

Nach Abschluss einer jeden Operation können Umordnungsschritte stattfinden, die spätere Operationen ev. beschleunigen.