

Beweis:

Sei f_k die minimale Anzahl von Elementen in einem „Fibonacci-Baum“ mit Wurzel-Rang k .

Aus dem vorangehenden **Lemma** folgt:

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

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:

$$\Delta bal = \text{Rang}(x)$$

Jeder kask. Schnitt erhöht #Bäume um 1

$$\Delta bal = \#kask. \text{ Schnitte}$$

Jeder Schnitt vernichtet eine Markierung

$$\Delta bal = -2 \cdot \#kask. \text{ Schnitte}$$

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:

$$\Delta bal = \text{Rang}(x)$$

Jeder Link-Schritt verkleinert #Bäume um 1

$$\Delta bal = -\#\text{Link-Schritte}$$

\Rightarrow Jeder Link-Schritt wird vom Bankkonto bezahlt und verschwindet amortisiert

$$\left. \begin{array}{l} \text{Einfügen der Kinder von } x \text{ in Wurzelliste:} \\ \Delta bal = \text{Rang}(x) \\ \text{Jeder Link-Schritt verkleinert \#Bäume um 1} \\ \Delta bal = -\#\text{Link-Schritte} \\ \Rightarrow \text{Jeder Link-Schritt wird vom Bankkonto} \\ \text{bezahlt und verschwindet amortisiert} \end{array} \right\} \Rightarrow a = \mathcal{O}(\log n)$$

Satz 50

Die *amortisierten* Kosten der Fibonacci-Heap-Operationen

($a_i = t_i + \Delta bal_i$) sind:

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

Jeder kask. Schnitt erhöht #Bäume um 1
 $\Delta bal = \#kask. Schritte$
Jeder Schnitt vernichtet eine Markierung
 $\Delta bal = -2 \cdot \#kask. Schritte$
Letzter Schnitt erzeugt ev. eine Markierung
 $\Delta bal = 2$




} $\Rightarrow a = \mathcal{O}(1)$

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

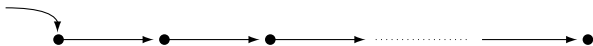
- einfach, wenig Verwaltungsoverhead
- effizient im amortisierten Sinn

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(x)*: Suchen des Elements x :
Die Liste wird von Anfang an durchsucht, bis das gesuchte Element x gefunden ist. Ist x das i -te Element in der Liste, so betragen die tatsächlichen Kosten i Einheiten.
- 2 *Insert(x)*: Einfügen des Elements x :
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 erfordert $(n' + 1)$ Schritte, wobei n' die aktuelle Länge der Liste ist.
- 3 *Delete(x)*: Löschen des Elements x :
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.

Annahme: Wir nehmen an, dass nach Ausführung jeder *Access*- oder *Insert*-Operation, die auf Element i angewandt wird, dieses Element **kostenlos** um beliebig viele Positionen in Richtung Listenanfang geschoben werden kann. Wir nennen jede daran beteiligte Elementvertauschung einen **kostenlosen** Tausch. Jeder andere Tausch zweier Elemente ist ein **bezahlter** Tausch und kostet eine Zeiteinheit.

Ziel:

Es soll eine einfache Regel gefunden werden, durch die mittels obiger Vertauschungen die Liste so organisiert wird, dass die Gesamtkosten einer Folge von Operationen minimiert werden. Wir sprechen von **selbstorganisierenden linearen Listen** und untersuchen hierzu folgende Regeln:

- MFR (Move-to-Front-Rule): *Access* und *Insert* stellen das betreffende Element vorne in die Liste ein und verändern ansonsten nichts.
- TR (Transposition-Rule): *Access* bringt das Element durch eine Vertauschung um eine Position nach vorne, *Insert* hängt es ans Ende der Liste an.

Bemerkung:

Wir werden im Folgenden sehen, dass MFR die Ausführung einer Sequenz von Operationen *Access*, *Insert* und *Delete* mit einer amortisierten Laufzeit erlaubt, die um höchstens den **konstanten Faktor 2** schlechter ist als ein optimaler Listenalgorithmus.

Für TR dagegen gibt es keinen solchen konstanten Faktor. Sie kann verglichen mit MTR bzw. einem optimalen Algorithmus beliebig schlecht sein.

Für die Algorithmen nehmen wir an:

- 1 die Kosten für $Access(x)$ und $Delete(x)$ sind gleich der Position von x in der Liste (vom Anfang an gezählt);
- 2 die Kosten für $Insert(x)$ (x nicht in Liste) sind die Länge der Liste nach der $Insert$ -Operation (d.h., neue Elemente werden zunächst am Listenende angehängt);
- 3 Ein Algorithmus kann die Liste jederzeit umorganisieren (die Reihenfolge ändern):
 - (a) **kostenlose** Vertauschung: nach einer $Access(x)$ - oder $Insert(x)$ -Operation kann x beliebig näher an den Anfang der Liste verschoben werden;
 - (b) **bezahlte** Vertauschung: jede andere Vertauschung benachbarter Listenelemente kostet eine Einheit.

Bemerkung: MFR (und auch TR) benutzen nur kostenlose Vertauschungen. 3(a) ist pessimistisch für MFR.

Satz 51

Sei s eine beliebige Folge von Access-, Insert- und Delete-Operationen, beginnend mit einer leeren Liste. Sei A ein beliebiger (optimaler) Wörterbuch-Algorithmus mit obigen Einschränkungen. Sei $C_A(s)$ der Zeitaufwand von A auf s , ohne bezahlte Vertauschungen. Sei $X_A(s)$ die Anzahl der bezahlten Vertauschungen. Sei $C_{MFR}(s)$ der Gesamtaufwand der Move_to_Front-Heuristik auf s . Sei $F_A(s)$ die Anzahl der kostenlosen Vertauschungen, die A auf s ausführt. Sei $m := |s|$ die Länge von s . Dann

$$C_{MFR}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m \\ (\leq 2 \cdot (C_A(s) + X_A(s)) .)$$

Beweis:

L_A bzw. L_{MFR} seien die Listen, A bzw. MFR seien die Algorithmen.

Potenzial := Anzahl der Inversionen in L_{MFR} im Vergleich zu L_A ,
d.h., falls o.B.d.A. $L_A = (x_1, x_2, \dots, x_n)$, dann ist das Potenzial

$$bal(L_{MFR}, L_A) = |\{(i, j); i < j, x_j \text{ ist in } L_{MFR} \text{ vor } x_i\}|$$

Beweis (Forts.):

Wir betrachten im Folgenden $Insert(x)$ als ein *Access* auf ein (fiktives) $n + 1$ -tes Element. Eine Operation ($Access(x_i)$ bzw. $Delete(x_i)$) kostet für A dann i Einheiten und für MFR t Einheiten, wobei t die Position von x_i in L_{MFR} ist.

- i) A ändert (bei *Access* bzw. *Delete*) L_A in kanonischer Weise. Für Element x_i sind die Kosten dann i .

Beweis (Forts.):

- iii) Die amortisierten Kosten einer **kostenlosen** Vertauschung durch Algorithmus A sind ≤ -1 :

$$\begin{aligned} L_A : & \quad x_1 \ x_2 \ \dots \ x_{i-1} \ x_i \ \dots \ x_n && \text{eine Inversion weniger} \\ L_{\text{MFR}} : & \quad x_i \ \dots \dots \ x_{i-1} \ \dots \dots \end{aligned}$$

- iv) Die amortisierten Kosten, die Algorithmus A durch eine **bezahlte** Vertauschung verursacht, sind ≤ 1 :

$$\begin{aligned} L_A : & \quad x_1 \ x_2 \ \dots \ x_{i-1} \ x_i \ \dots \ x_n && \text{plus eine Inversion} \\ L_{\text{MFR}} : & \quad \dots \ x_{i-1} \ \dots \ x_i \ \dots \dots \end{aligned}$$

Beweis (Forts.):

Ist das Anfangspotenzial = 0, so ergibt sich

$$C_{\text{MFR}}(s) \leq \text{amort. Kosten MFR} \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$



Anmerkung: Wenn wir nicht mit einer leeren Liste (oder zwei identischen Listen, d.h. $L_A = L_{\text{MFR}}$) starten, ergibt sich:

$$C_{\text{MFR}} \leq 2C_A(s) + X_A(s) - F_A(s) - m + n^2/2.$$



Daniel D. Sleator, Robert E. Tarjan:

Amortized efficiency of list update and paging rules

Commun. ACM **28**(2), pp. 202–208 (1985)

6.3 Sich selbst organisierende Binärbäume

Wir betrachten hier Binärbäume für Priority Queues.

Definition 52

Ein **Leftist-Baum** (Linksbaum) ist ein (interner) binärer Suchbaum, so dass für jeden Knoten gilt, dass ein kürzester Weg zu einem Blatt (externer Knoten!) über das rechte Kind führt [s. Knuth].

Zur Analyse von Linksbäumen verweisen wir auf



Knuth, Donald E.:

The Art of Computer Programming. Volume 3 / Sorting and Searching

Addison-Wesley Publishing Company: Reading, MA (1973)
pp. 150ff [[pdf \(54MB\)](#), [djvu \(8MB\)](#)]

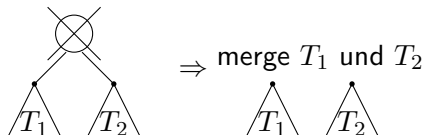
Wir betrachten nun eine durch Linksbäume motivierte, sich selbst organisierende Variante als Implementierung von Priority Queues. Die dabei auftretenden Bäume brauchen **keine** Linksbäume zu sein!

Operationen:

- *Insert*
 - *FindMin*
 - *ExtractMin*
 - *Delete*
 - *Merge*
- } alle Operationen lassen sich auf *Merge* zurückführen

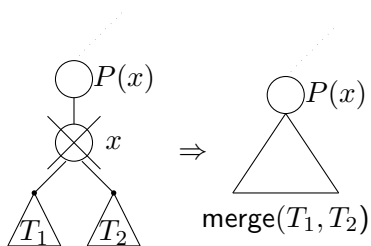
Operationen:

- 1 $Insert(T, x)$: neuer Baum mit x als einzigem Element, merge diesen mit T
- 2 $FindMin$: \surd (Minimum an der Wurzel, wegen Heap-Bedingung)
- 3 $ExtractMin$:



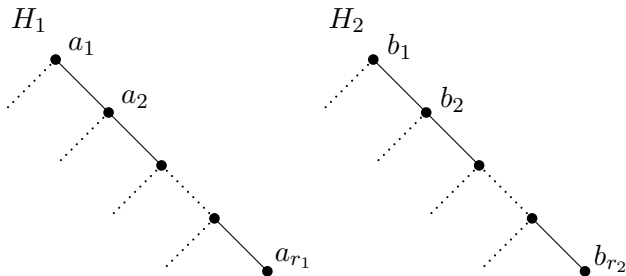
Operationen:

4 Delete:



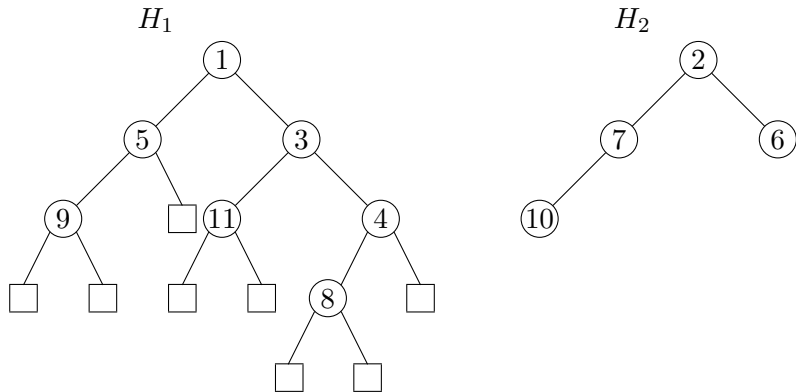
Operationen:

- ④ **Merge:** Seien zwei Heaps H_1 und H_2 gegeben, sei a_1, \dots, a_{r_1} die Folge der Knoten in H_1 von der Wurzel zum rechtesten Blatt, sei b_1, \dots, b_{r_2} die entsprechende Folge in H_2 .



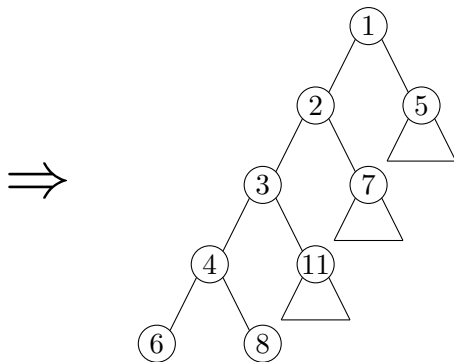
Sei $(c_1, c_2, \dots, c_{r_1+r_2})$ die durch Merging aus (a_1, \dots, a_{r_1}) und (b_1, \dots, b_{r_2}) entstehende (sortierte) Folge. Mache $(c_1, \dots, c_{r_1+r_2})$ zum **linksten** Pfad im neuen Baum und hänge den jeweiligen anderen (d.h. linken) UB a_i bzw. b_j als **rechten** UB des entsprechenden c_k an.

Beispiel 53



(Bemerkung: H_1 ist kein Linksbaum!)

Beispiel 53

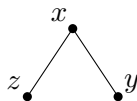


(Bemerkung: H_1 ist kein Linksbaum!)

Amortisierte Kostenanalyse der Merge-Operation:

Sei

$w(x) :=$ Gewicht von $x = \#$ Knoten im UB von x einschließlich x



(x, y) schwer $\Rightarrow (x, z)$ leicht

Kante (x, y) heißt **leicht**, falls $2 \cdot w(y) \leq w(x)$

Kante (x, y) heißt **schwer**, falls $2 \cdot w(y) \geq w(x)$

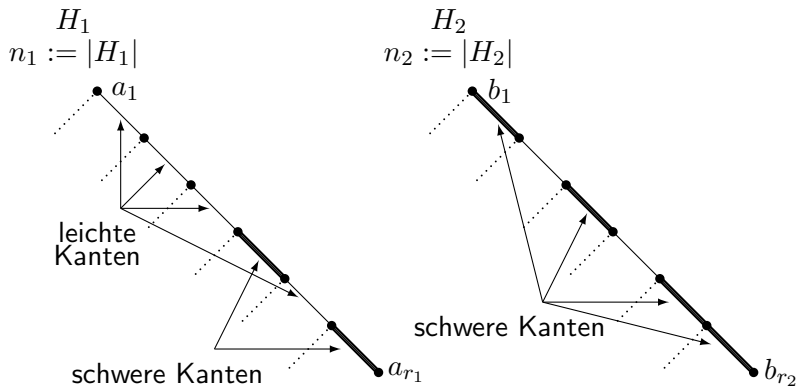
Es gilt: Es ist immer ein Kind über eine **leichte**, höchstens eins über eine **schwere** Kante erreichbar (d.h. die beiden Fälle oben schließen sich wechselseitig aus!). .

Beobachtungen:

- 1 (x, y) schwer $\Rightarrow (x, z)$ leicht
- 2 Da sich über eine leichte Kante das Gewicht mindestens halbiert, kann ein Pfad von der Wurzel zu einem Blatt höchstens $\lg n$ leichte Kanten enthalten.

Setze

Potenzial := # schwere Kanten zu rechten Kindern



Sei $s_1 :=$ die Zahl der schweren Kanten auf dem rechten Pfad in H_1 , s_2 analog.

$$r_1 \leq s_1 + \text{ld } n_1$$

$$r_2 \leq s_2 + \text{ld } n_2$$

Amortisierte Kosten für *Merge*:

$$\leq r_1 + r_2 + \text{ld}(n_1 + n_2) - s_1 - s_2$$

$$\leq \text{ld } n_1 + \text{ld } n_2 + \text{ld}(n_1 + n_2)$$

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

da die s_1 schweren Kanten auf dem rechten Pfad in H_1 nun zu linken Kindern verlaufen (ebenso für die in H_2) und höchstens $\text{ld}(n_1 + n_2)$ Knoten auf dem linken Pfad im neuen Baum eine schwere Kante zum rechten (und damit eine leichte Kante zum linken) Kind haben können.

Satz 54

Eine Folge von m Merge-Operationen benötigt Zeit

$$\mathcal{O}\left(\sum_{i=1}^m \log(n_i)\right),$$

wobei n_i die Größe des Baumes ist, der in der i -ten Merge-Operation geschaffen wird.

Beweis:

s.o.

