

**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 Elementenvertauschung 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

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

## 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  $\leq -1$ :

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

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

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

## 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 \(8MB\)](#) ]

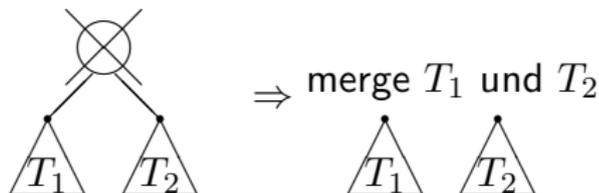
Im Folgenden betrachten wir eine durch Linksbäume motivierte, sich selbst organisierende Variante als Implementierung von Priority Queues.

# Operationen:

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

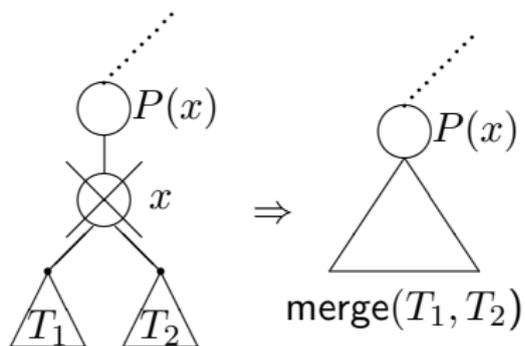
# Operationen:

- 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  $ExtractMin$ :



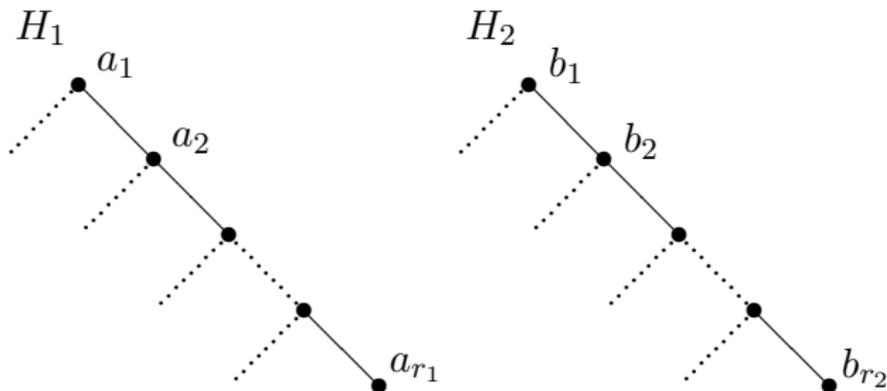
# Operationen:

## 4 Delete:



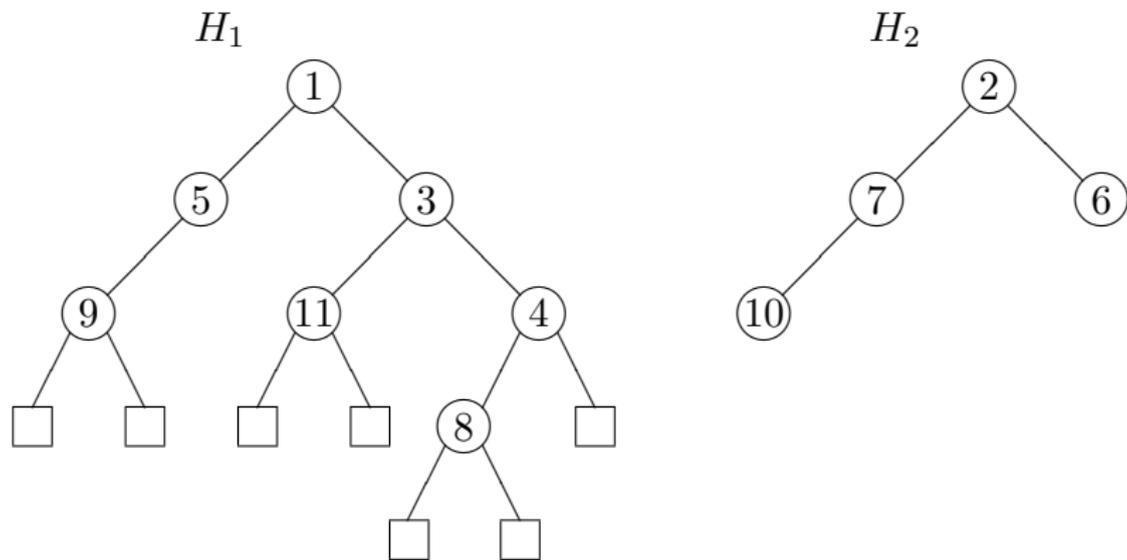
## Operationen:

- 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  $H_2$ .



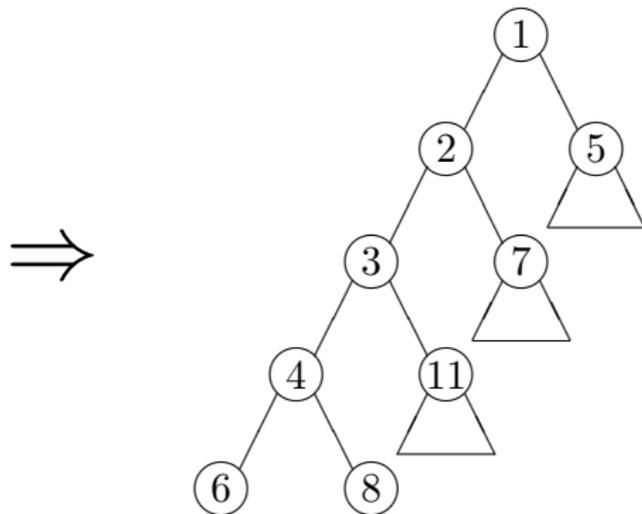
Sei  $(c_1, c_2, \dots, c_{r_1+r_2})$  die durch Merging aus  $(a_1, \dots, a_{r_1})$  und  $(b_1, \dots, b_{r_2})$  entstehende (sortierte) Folge. Mache  $(c_1, \dots, c_{r_1+r_2})$  zum **linksten** Pfad im neuen Baum und hänge den jeweiligen anderen (d.h. linken) UB  $a_i$  bzw.  $b_j$  als **rechten**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

## Beispiel 53



(Bemerkung:  $H_1$  ist kein Linksbaum!)

## Beispiel 53

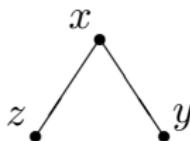


(Bemerkung:  $H_1$  ist kein Linksbaum!)

# 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  $2 \cdot w(y) \geq w(x)$

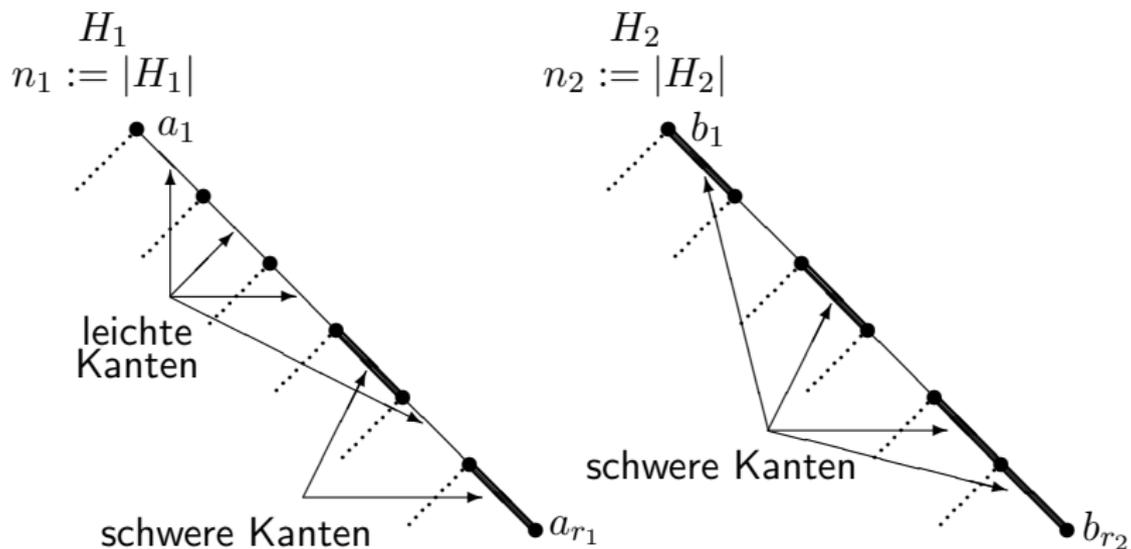
**Es gilt:** Es ist immer ein Kind über eine **leichte**, eins über eine **schwere** Kante erreichbar (d.h., die beiden Fälle schließen sich wechselseitig aus!).

## 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*:

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

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

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

## 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.

