6. Sich selbst organisierende Datenstrukturen

6.1 Motivation

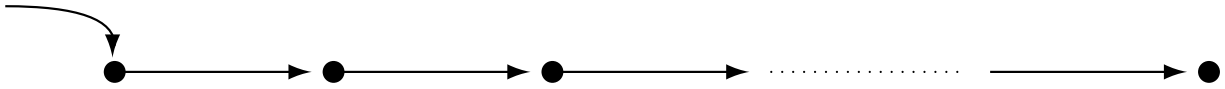
- einfach, wenig Verwaltungsoverhead
- effizient im amortisierten Sinn

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 erfordet $(n' + 1)$ 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 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

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

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. $L_{\mathsf{A}} = (x_1, x_2, \ldots, x_n)$, dann ist das Potenzial

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

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.):

ii) Die amortisierten Kosten einer Operation (*Access*(x_i) oder *Delete*(x_i)) für MFR sind $\leq 2i-1$:

L_A : $x_1\ x_2\ \ldots\ x_i\ \ldots\ldots\ldots x_n$

L_{MFR} : $\ldots\ldots\ldots\ x_j\ \ldots\ x_i\ \ldots$

$\uparrow$

t

Sei $k = \#$Elemente x_j, $j > i$, die in L_{MFR} vor x_i stehen.
Sei t die Position von x_i im L_{MFR}. Dann

$$t \leq i + k; \qquad \Rightarrow \qquad k \geq t - i$$

$$\Delta bal \leq -k + (i-1) \leq 2i - 1 - t \qquad \text{(bei } \textit{Access}\text{)}$$

$$\Delta bal = -k \leq i - t \qquad \text{(bei } \textit{Delete}\text{)}$$

Also: Amortisierte Kosten $\leq t + \Delta bal \leq 2i - 1$.

Beweis (Forts.):

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

L_A : $x_1\ x_2 \ldots \overset{\frown}{x_{i-1}\ x_i} \ldots x_n$ eine Inversion weniger

L_{MFR} : $x_i \ldots\ldots x_{i-1} \ldots\ldots$

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

L_A : $x_1\ x_2 \ldots \overset{\frown}{x_{i-1}\ x_i} \ldots x_n$ plus eine Inversion

L_{MFR} : $\ldots x_{i-1} \ldots x_i \ldots\ldots$

Beweis (Forts.):

Ist das Anfangspotenzial $= 0$, so ergibt sich

$$\mathsf{C}_{\mathsf{MFR}}(s) \leq \text{amort. Kosten MFR } \leq 2 \cdot \mathsf{C}_{\mathsf{A}}(s) + \mathsf{X}_{\mathsf{A}}(s) - \mathsf{F}_{\mathsf{A}}(s) - m$$

□

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

$$\mathsf{C}_{\mathsf{MFR}} \leq 2\mathsf{C}_{\mathsf{A}}(s) + \mathsf{X}_{\mathsf{A}}(s) - \mathsf{F}_{\mathsf{A}}(s) - m + n^2/2\,.$$

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*

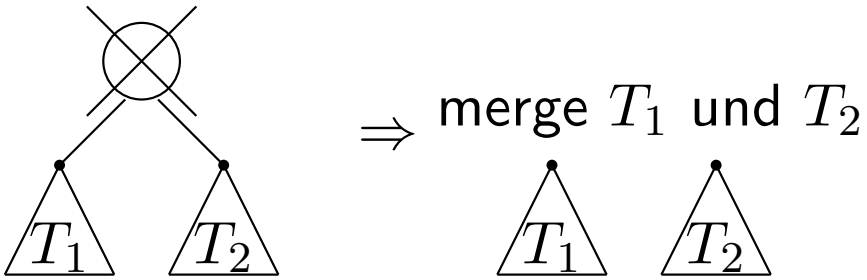
} alle Operationen lassen sich auf *Merge* zurückführen

Operationen:

1. *Insert*(T, x): neuer Baum mit x als einzigem Element, merge diesen mit T

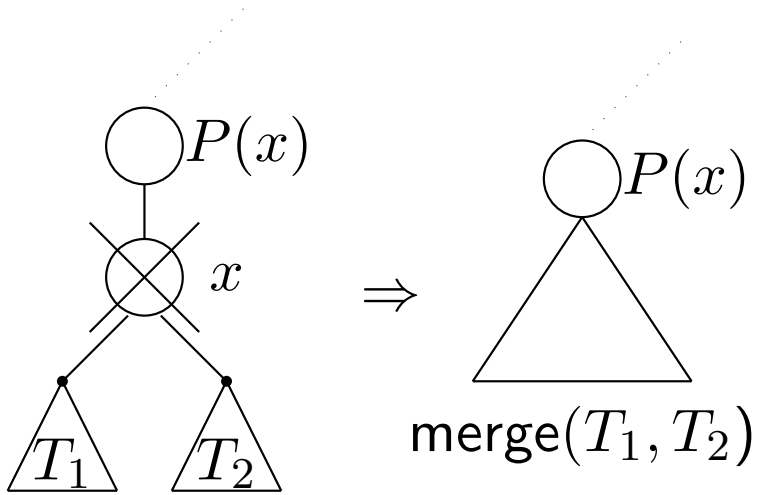
2. *FindMin*: $\surd$ (Minimum an der Wurzel, wegen Heap-Bedingung)

3. *ExtractMin*:



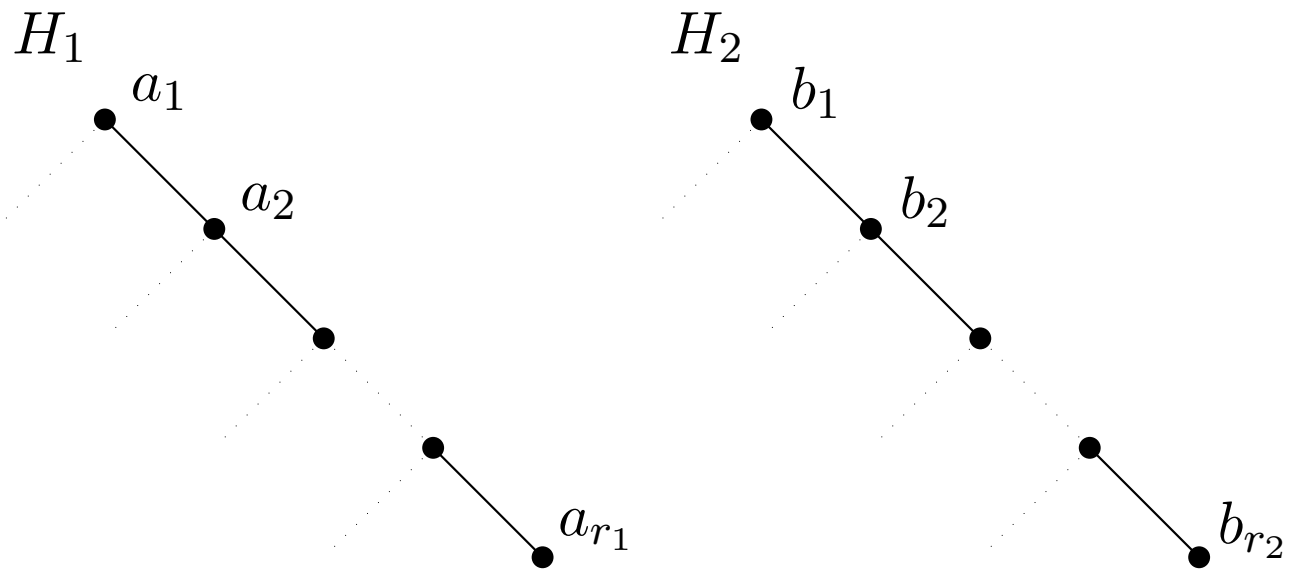
Operationen:

4. *Delete*:



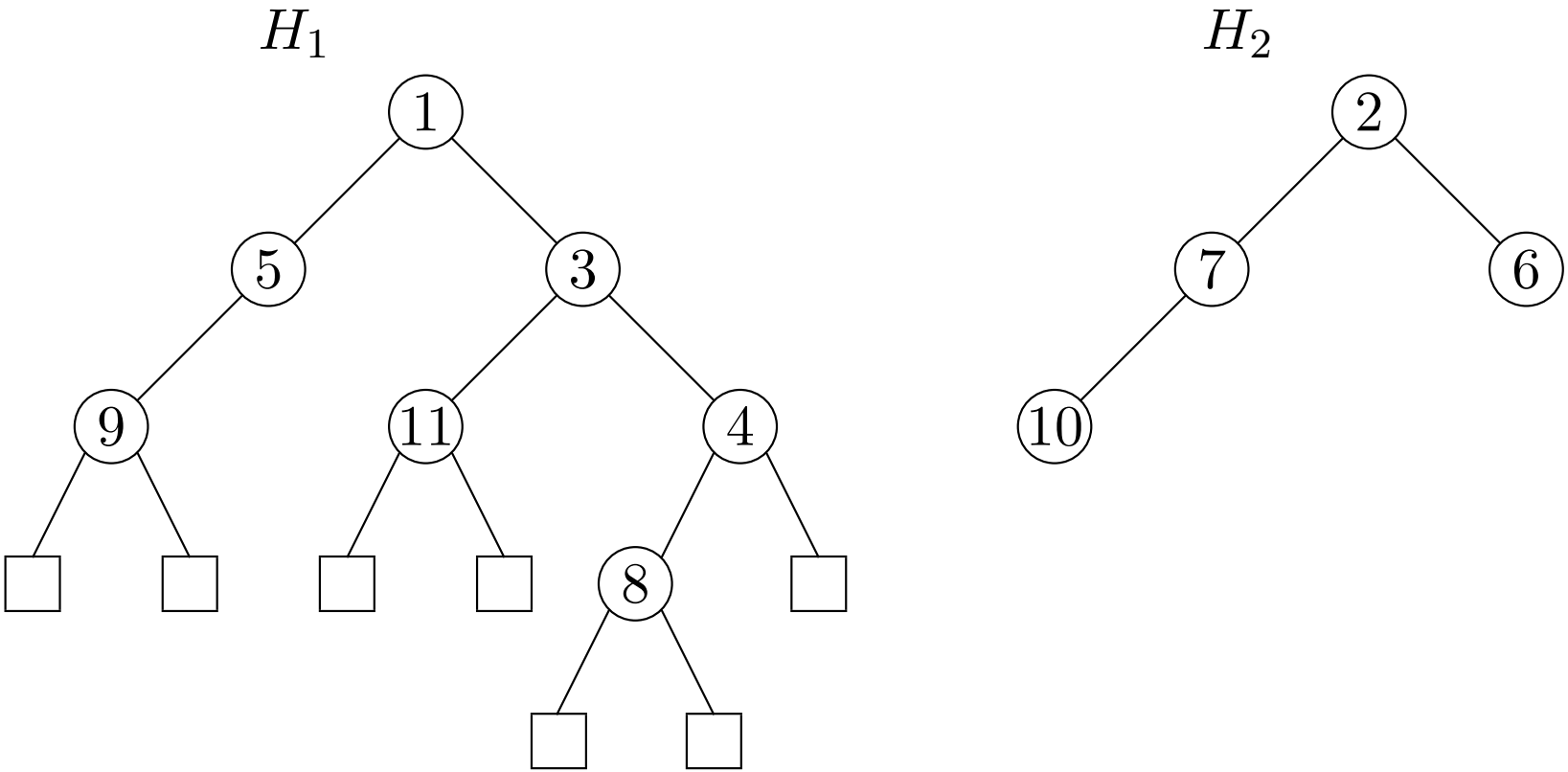
Operationen:

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



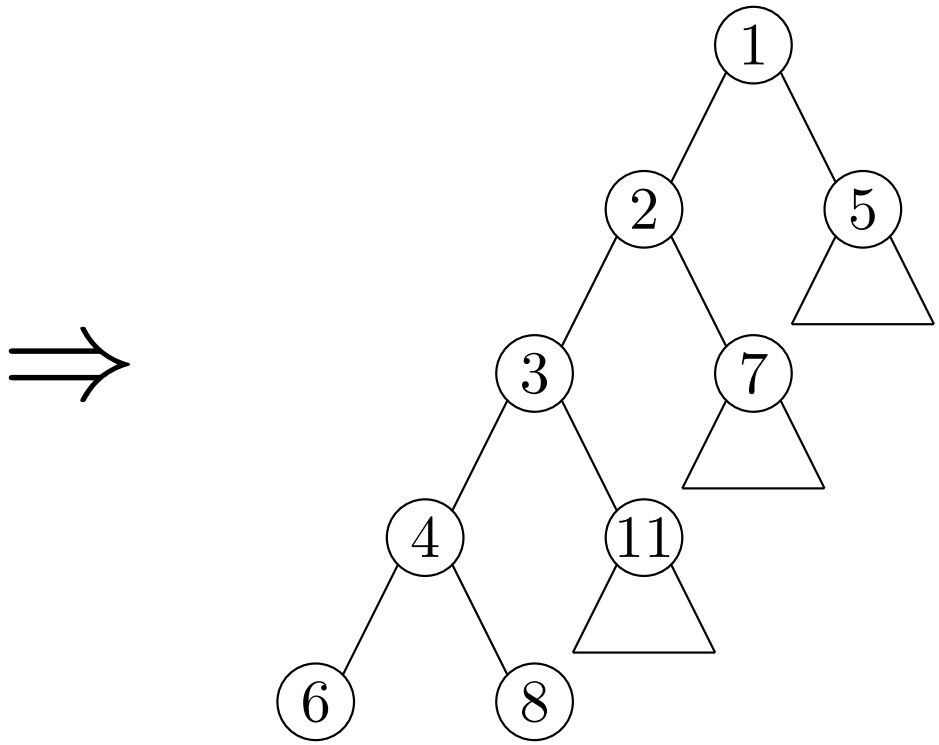
Sei $(c_1, c_2, \ldots, c_{r_1+r_2})$ die durch Merging aus $(a_1, \ldots, a_{r_1})$ und $(b_1, \ldots, b_{r_2})$ entstehende (sortierte) Folge. Mache $(c_1, \ldots, c_{r_1+r_2})$ zum linkesten Pfad im neuen Baum und hänge den jeweiligen anderen (d.h. linken) UB a_i bzw. b_j als rechten UB des entsprechenden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

Beispiel 53



(Bemerkung: H_1 ist kein Linksbaum!)

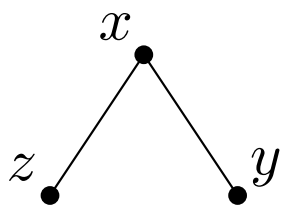
Beispiel 53



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

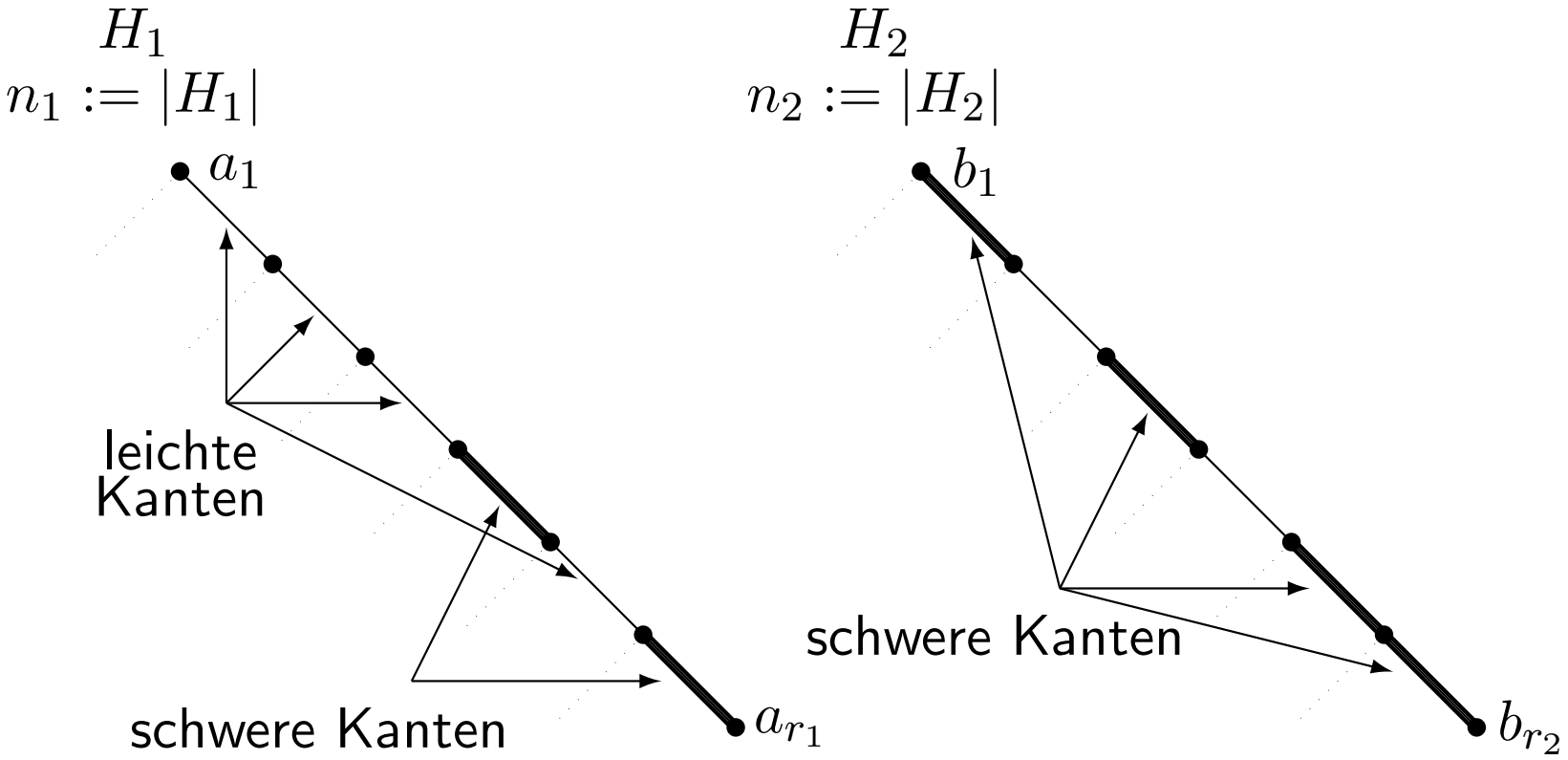
Es gilt: Es ist immer ein Kind über eine leichte, höchstens eins über eine schwere Kante erreichbar (d.h., die beiden Fälle schließen sich wechselseitig aus!).

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 $\mathrm{ld}\, n$ leichte Kanten enthalten.

Setze

$$\text{Potenzial} := \#\text{ schwere Kanten zu rechten Kindern}$$


H_1
$n_1 := |H_1|$
a_1
leichte
Kanten
schwere Kanten
a_{r_1}
H_2
$n_2 := |H_2|$
b_1
schwere Kanten
b_{r_2}

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

$$r_1 \leq s_1 + \mathrm{ld}\, n_1$$
$$r_2 \leq s_2 + \mathrm{ld}\, n_2$$

Amortisierte Kosten für *Merge:*

$$\begin{aligned} &\leq r_1 + r_2 + \mathrm{ld}(n_1 + n_2) - s_1 - s_2 \\ &\leq \mathrm{ld}\, n_1 + \mathrm{ld}\, n_2 + \mathrm{ld}(n_1 + n_2) \\ &= \mathcal{O}(\log(n_1 + n_2)) \end{aligned}$$

Satz 54

Eine Folge von m Merge-Operationen benötigt Zeit

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

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

Beweis:

s.o. □