Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Algorithmen und Datenstrukturen

Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM, $\mathcal{O}$-Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

In dieser (und auch der nächsten) Vorlesung werden wir erfahren,

1. wie man andere Mengenoperationen effizient mit (fast) ausgeglichenen Suchbäumen implementiert,

genauer werden wir hierzu wieder AVL-Bäume heranziehen

2. wie man Mengen mit anderen “Bedürfnissen” verwaltet, z.B.:

2a. Prioritätswarteschlangen oder

2b. Äquivalenzklassen

Mengenoperationen mit Suchbäumen

Mengen S werden häufig entweder als Bitvektoren (wenn das Universum U klein ist) oder als Suchbäume gespeichert (wenn U groß ist).
Wir betrachten im Folgenden die Speicherung von Mengen als Suchbäume.
1. Ist S als Suchbaum T gespeichert, so lässt sich die Operation Seq(T) (S der Größe nach sortiert ausgeben) in Zeit $\mathcal{O}(|S|)$ ausführen.
Beweis: T in inorder-Reihenfolge durchlaufen.
2. Ist S als Folge $x_1 < x_2 < ... < x_n$ gegeben, so lässt sich in Zeit $\mathcal{O}(|S|)$ ein Suchbaum erstellen, der gleichzeitig AVL- und $BB[1/3]$-Baum ist.
Beweis: Bekannte Baum-Interpretation eines Feldes.

Gewünschte Mengenoperationen:

- IstTeilmenge (S_1, S_2) — if $S_1 \subseteq S_2$ then true else false
- Vereinigung (S_1, S_2, S_3) — $S_3 \leftarrow S_1 \cup S_2$, S_3 Suchbaum
- Durchschnitt (S_1, S_2, S_3) — $S_3 \leftarrow S_1 \cap S_2$,
- Differenz (S_1, S_2, S_3) — $S_3 \leftarrow S_1 - S_2$,
- Kopie (S_1, S_2) — $S_2 \leftarrow S_1$,

- Anhängen (S_1, S_2, S_3) — if $\max(S_1) < \min(S_2)$ then $S_3 \leftarrow S_1 \cup S_2$,
- Spalten (S_1, a, S_2, S_3) — $S_2 \leftarrow \{x \in S_1 \mid x \le a\}$, $S_3 \leftarrow \{x \in S_1 \mid x > a\}$

Lemma: Bei Speicherung von Mengen als Suchbäume können die Operationen IstTeilmenge, Vereinigung, Durchschnitt und Differenz in Zeit $\mathcal{O}(|S_1| + |S_2|)$, die Operation Kopie in Zeit $\mathcal{O}(|S_1|)$ ausgeführt werden.

Beweis: a) Kopie ist klar

b) Seien T_1und T_2 Suchbäume für S_1 und S_2.

i) Bilde Seq(T_1) und Seq(T_2),

ii) Führe die Operationen auf den geordneten Folgen aus i) aus (dies kann gleichzeitig mit i) geschehen)

iii) bilde aus der Folge aus ii) einen Suchbaum

Zeitbedarf insgesamt: $\mathcal{O}(|S_1| + |S_2|)$.

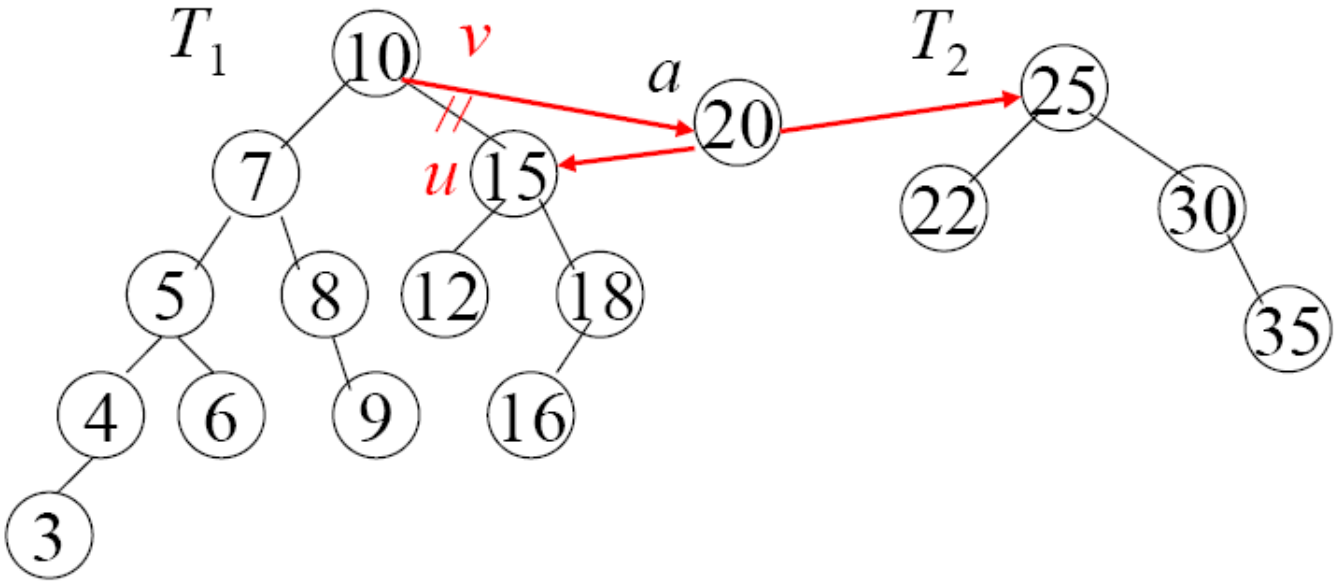
Im Folgenden nehmen wir an, dass die Mengen als AVL-Bäume gespeichert sind.

Hilfsalgorithmus Anhängen*(T_1, a, T_2, T_3)

Gegeben: AVL-Bäume T_1, T_2 für S_1, S_2, sowie Element a. Es gelte $\max(S_1) < a < \min(S_2)$

Gesucht: AVL-Baum T_3 für $S_3 = S_1 \cup \{a\} \cup S_2$

Beispiel:



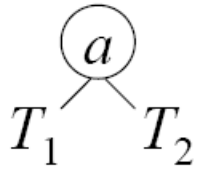
1. Versuch: Sei o.B.d.A. $H\ddot{o}he(T_1) \geq H\ddot{o}he(T_2)$

a) Steige in T_1 ganz rechts hinab bis zum Knoten u mit Höhe(T_u) = Höhe(T_2).

b) i) u ist Wurzel von T_1:

Mache a zum Vater von T_1 und T_2

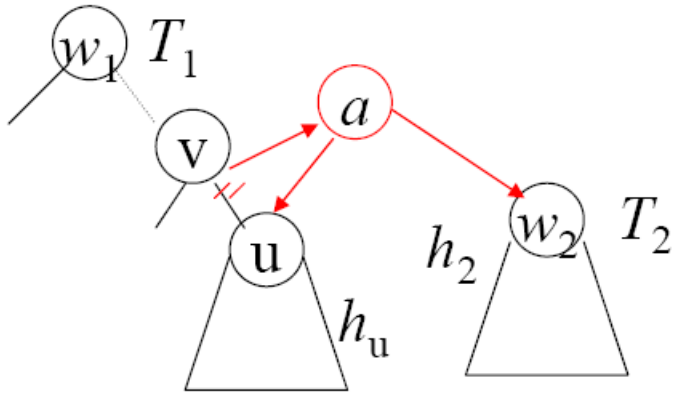
Höhe (T_1) = Höhe (T_2)



ii): u hat Vater v:

Mache a zum Vater von T_2
und zum Vater von u,

mache v zum Vater von a

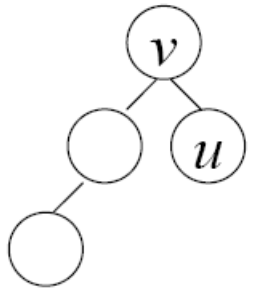


Fehlversuch!

Gründe für den Fehlversuch:

1. Knoten u existiert nicht immer, denn die Höhe der Bäume kann beim Absteigen um mehr als je 1 vermindert werden (im Beisp. : Höhe(10) = 4, Höhe(15) = 2)
2. Der neue Baum ist i. allg. kein AVL-Baum und lässt sich auch nicht durch einfaches Rebalancieren dazu machen.

Analyse: Sei T AVL-Baum und sei v Vater von u.
Seien h_u, h_v die Höhen von u bzw. v.



a) Es gilt $h_v > h_u \geq h_v - 2$, da T AVL-Baum

b) Wenn u einziger Sohn von v ist, folgt $h_u = 0$

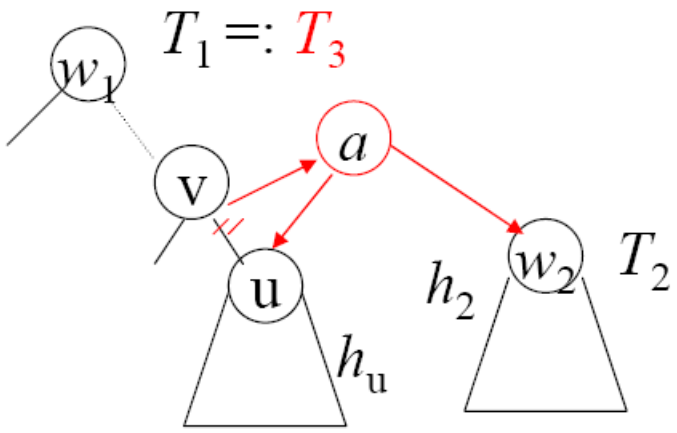
c) $h_u = h_v - 2$ gdw. der Teilbaum von v, in dem u liegt, ist kleiner als der andere Teilbaum von v.

Folgerung: Beim Abstieg ganz rechts im Baum sinkt die Höhe der Unterbäume um 1 oder 2 bis zur Höhe 1 oder 0.

2. Versuch. Sei o.B.d.A. $H\ddot{o}he(T_1) \geq H\ddot{o}he(T_2)$

Algorithmus Anhängen*(T_1, T_2, a, T_3)

1. Steige in T_1 ganz rechts ab bis zum ersten Knoten u mit
   $h_u \leq H\ddot{o}he(T_2)+1$
2. Mache a zum Vater von u und T_2.
   Falls u in T_1 einen Vater v hatte,
   mache a zum rechten Sohn von v
3. Rebalanciere ab v aufwärts.



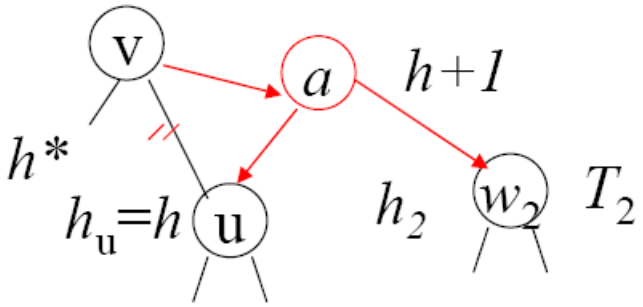
Korrektheit des Algorithmus:

a) der neue Baum T ist Suchbaum für $S_3 = S_1 \cup \{a\} \cup S_2$

b) T ist AVL-Baum, denn

1. u existiert nach Folgerung, d.h. T existiert

2. Sei $h_2 = \text{Höhe}(T_2)$, $h := h_u$
   also $h_2 \le h_u = h \le h_2 + 1$, daher a in Balance

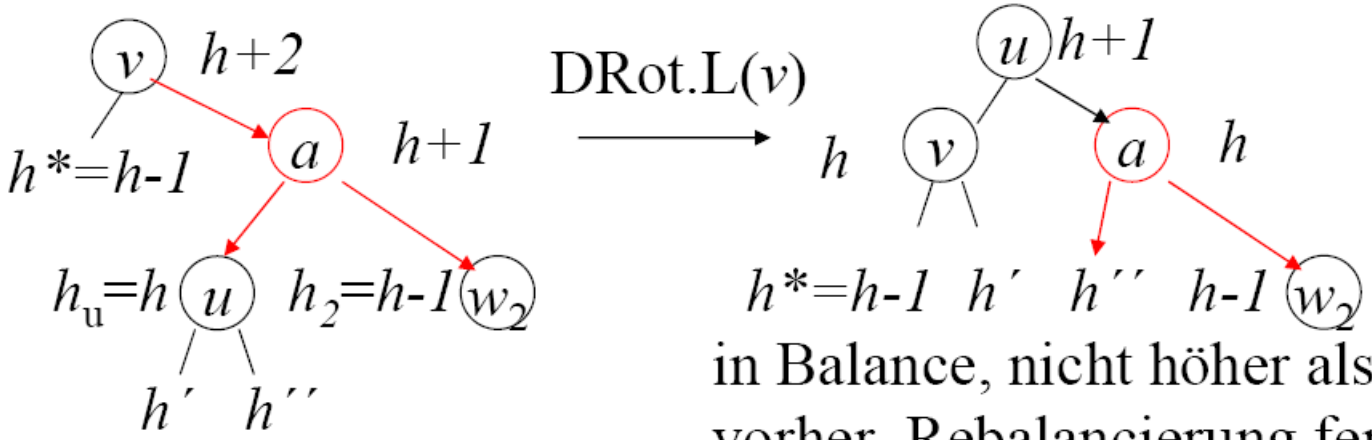


3 Fälle: $h^* = h\text{-}1$, zu zeigen jeweils: lokale

$h^* = h$, Rebalancierungen auf dem

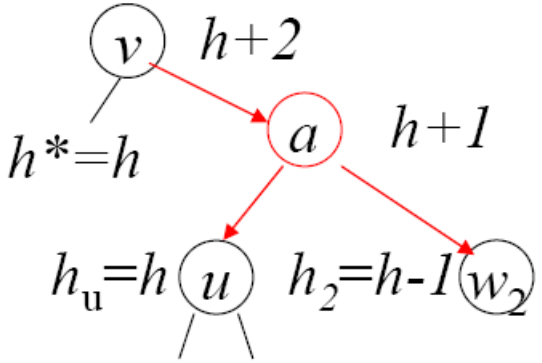
$h^* = h+1$. Suchpfad liefern AVL-Baum

Fall 2.1: $h^* = h\text{-}1$, daraus folgt $h = h_2+1$, denn dann wurde der rechte Teilbaum von T_1 in Einerschritten abwärts bis $h_u = h = h_2+1$ traversiert.



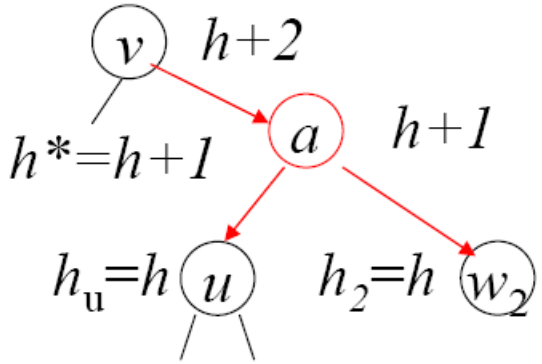
in Balance, nicht höher als vorher, Rebalancierung fertig

Fall 2.2: $h^* = h$



in Balance, höher geworden, Aufstieg und Rebalancierung fortsetzen

Fall 2.3: $h^* = h+1$



in Balance, nicht höher als vorher, Rebalancierung fertig

Zeitbedarf: O(Höhe(T_1) – Höhe(T_2)),
denn man muss ganz rechts in T_1 einen Weg der Länge Höhe(T_1) – Höhe(T_2)+c ab- und evtl. wieder aufsteigen.

Bem. Der Fall $Höhe(T_1) < Höhe(T_2)$ geht analog

Algorithmus Anhängen (S_1, S_2 , S_3)

- Sei T_i AVL-Suchbaum für S_i mit $i = 1, 2$;
- sei $\max(S_1) < \min(S_2)$

1. Lösche größtes a aus S_1 in T_1 (a ist rechtester Knoten), es entstehe T_1', a, T_2
2. Anhängen* (T_1', a, T_2 , T_3).
3. Dann ist T_3 AVL-Suchbaum für $S_3 = S_1 \cup S_2$

Zeitbedarf: 1. O(Höhe(T_1))

2. $O(Höhe(S_1) - Höhe(S_2))$

$\leq O(\max(Höhe(T_1), Höhe(T_2)))$

$= O(\max(\log\|S_1\|, \log\|S_2\|))$

Algorithmus Spalten (S_1, a, S_2 , S_3)

Sei T_1 AVL-Suchbaum für S_1;

Verwalte linke und rechte Liste von Knoten u. Teilbäumen

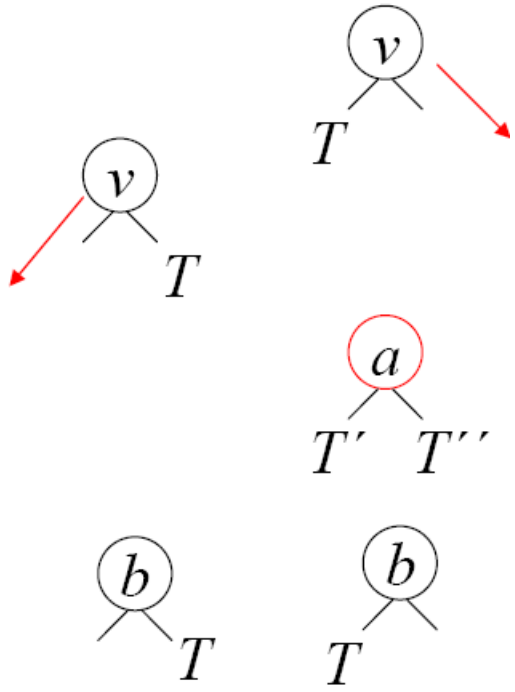
1. Steige in den Suchpfad durch Suche(a) hinab.
   Sei v aktueller Knoten:

   a) Suchpfad führt zu rechtem Sohn von v:
   Nimm T, v in linke Liste auf.

   b) Suchpfad führt zu linkem Sohn von v:
   Nimm v, T in rechte Liste auf.

   c) Suchpfad endet bei $v = a$:
   Nimm T', v in linke Liste auf,
   Nimm T'' in rechte Liste auf.

   d) Suchpfad endet bei $v = b$ mit $b \neq a$
   Falls $a < b$, nimm v, T in rechte Liste auf,
   falls $a > b$, nimm T, v in linke Liste auf.



- Es entstehen Listen, jeweils aufsteigend sortiert sind.
  links: $T_1', a_1', T_2', a_2', \ldots T_l', a_l'$. Abarbeitung ←
  rechts: (a_m''), $T_m'', \ldots a_2'', T_2'', a_1'', T_1''$. Abarbeitung →

2. Wende sukzessiv Anhängen* auf die linke und rechte Liste an, kleine Bäume zuerst:

   links: Einfügen $(T_l', a_l') = T_l^*$

   For $i := l$ downto 2 do Anhängen*$(T'_{i-1}, a'_{i-1}, T_i^*, T_{i-1}^*)$

   es entstehe T_2

   rechts: Einfügen $(T_m'', a_m') = T_m^*$

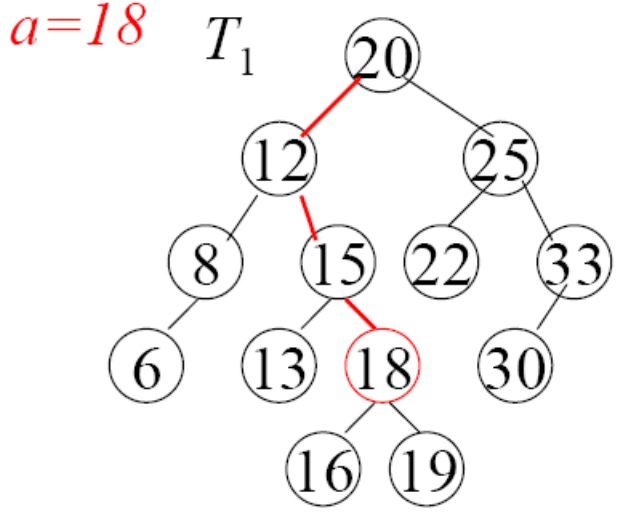
   For $i := m$ downto 2 do Anhängen*$(T_i^*, a''_{i-1}, T''_{i-1}, T_{i-1}^*)$

   es entstehe T_3

Es gilt: T_2 und T_3 sind AVL-Suchbäume für

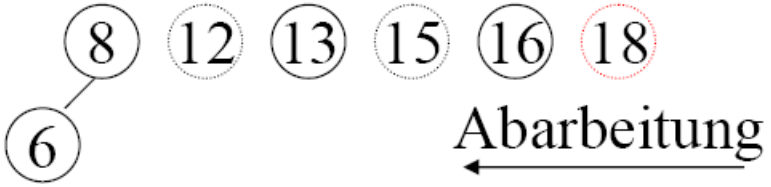
$$S_2 = \{x \in S_1 \mid x \le a\}, \quad S_3 = \{x \in S_1 \mid x > a\}$$

Beispiel für Spalten (S_1, a, S_2 , S_3)



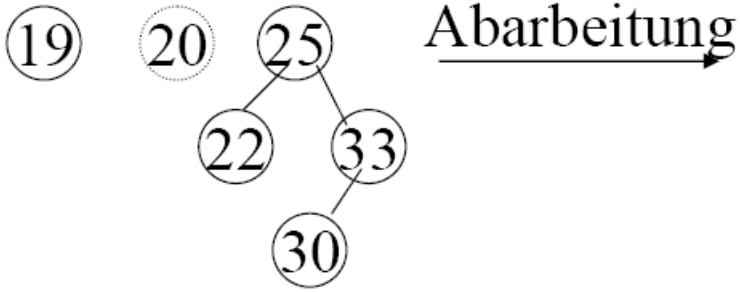
links von 18:

T_1', a_1', T_2', a_2', T_3', a_3'.



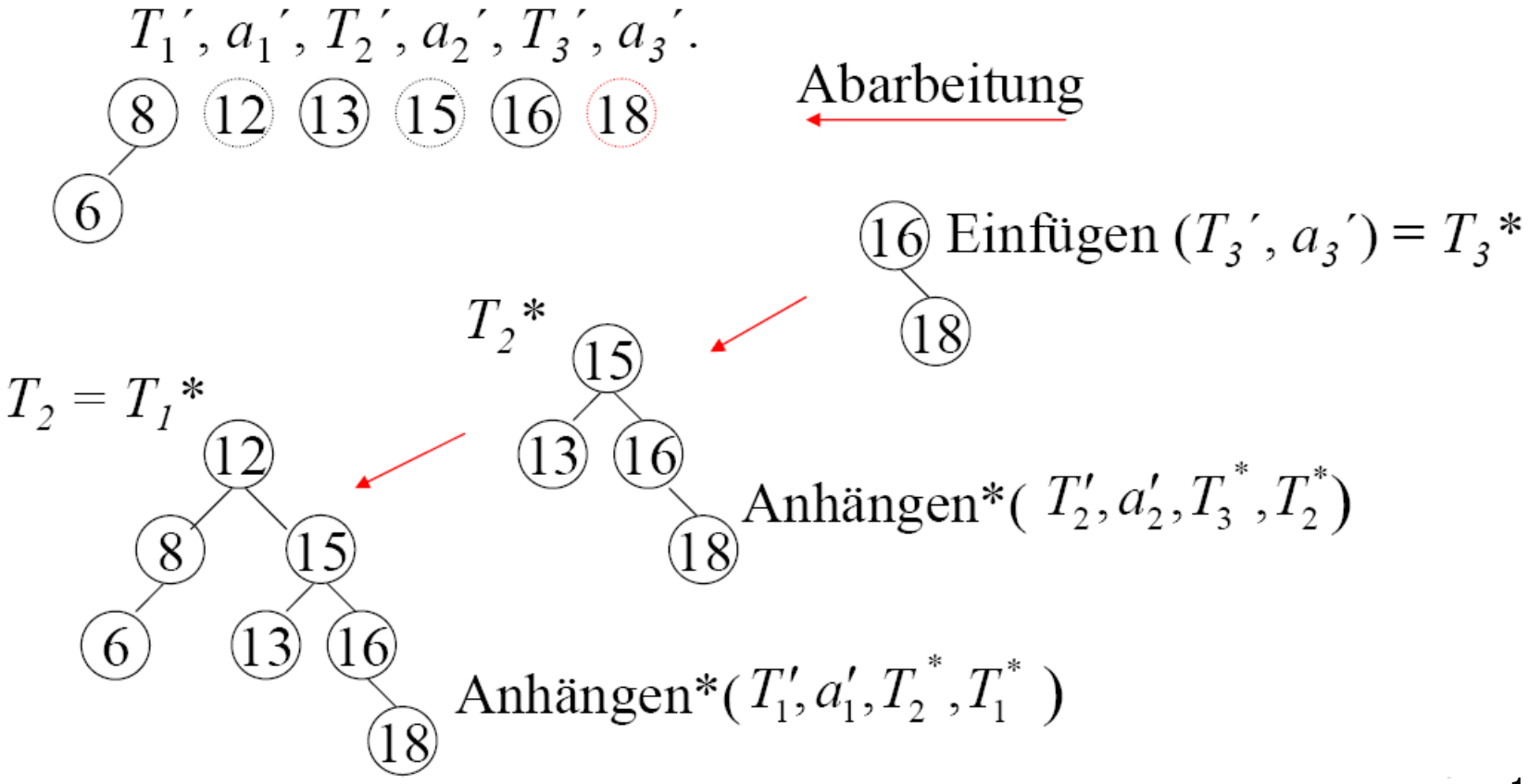
rechts von 18:

T_2'', a_1'', T_1''.



Aufbau der Teilbäume durch Einfügen und Anhängen*

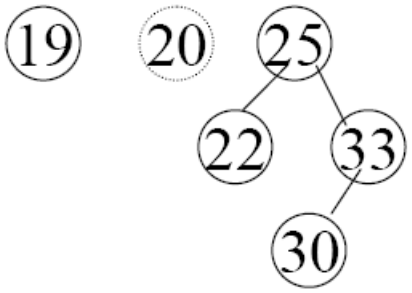
links von 18:



Aufbau der Teilbäume durch Einfügen und Anhängen*

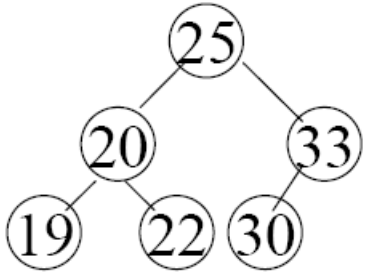
rechts von 18:

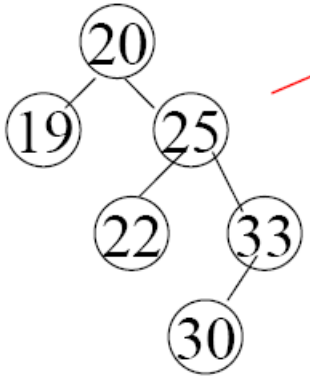
T_2'', a_1'', T_1''.



Abarbeitung

$T_2 = T_1$*





Rot.Links(20) ist Teil v.
Anhängen*($T_2^*, a_1'', T_1'', T_1^*$)

T_2*

Zwischenzustand Anhängen*($T_2^*, a_1'', T_1'', T_1^*$)

Einfügen (T_2'', a_2') = T_2*

Zeitbedarf von Spalten (S_1, a, S_2 , S_3):

Sei $h'_i = Höhe(T'_i)$, $h_i^* = Höhe(T_i^*)$,

Es gilt a) $h'_{i+1} \le h'_i$,

b) $\left|h'_{i-1} - h_i^*\right| \le h'_{i-1} - h'_i + 2$

umständlich zu zeigen, hier ohne Beweis

Also Aufbau von T_{i-1}^* aus T'_{i-1}, T_i^* kostet $c \cdot \left(\left|h'_{i-1} - h_i^*\right| + 1\right)$

Zeit für den Aufbau von T_1^* aus $T'_1, \ldots T'_l$:

$$c \cdot \sum_{i=l}^{2} \left(\left|h'_{i-1} - h_i^*\right| + 1\right) \le c \cdot \sum_{i=l}^{2} \left(h'_{i-1} - h'_i + 3\right) \quad \text{wegen b)}$$

$$\le c \cdot Höhe(T)$$

Also Zeit für linke und rechte Liste: jeweils O(Höhe(T))

Daher Gesamtzeit des Algorithmus O(Höhe(T)) = O(log $\|S\|$)

Vorrangwarteschlangen: eine JAVA-Sicht, siehe `http://java.sun.com/j2se/1.5.0/docs/api/java/util/PriorityQueue.html`.

Method Summary	
boolean	**add**(E o) Adds the specified element to this queue.
void	**clear**() Removes all elements from the priority queue.
Comparator<? super E>	**comparator**() Returns the comparator used to order this collection, or `null` if this collection is sorted according to its elements natural ordering (using `Comparable`).
Iterator<E>	**iterator**() Returns an iterator over the elements in this queue.
boolean	**offer**(E o) Inserts the specified element into this priority queue.
E	**peek**() Retrieves, but does not remove, the head of this queue, returning `null` if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or `null` if this queue is empty.
boolean	**remove**(Object o) Removes a single instance of the specified element from this queue, if it is present.
int	**size**() Returns the number of elements in this collection.

Beim NIST findet man... (National Institute of Standards),
`http://www.nist.gov/dads/HTML/priorityque.html`
A *priority queue* is an abstract data type to efficiently support finding the item with the highest priority across a series of operations.
The basic operations are: insert, find-minimum (or maximum), and delete-minimum (or maximum).
Some implementations also efficiently support join two priority queues (meld), delete an arbitrary item, and increase the priority of a item (decrease-key).
Formal Definition: The operations new(), insert(v, PQ), find-minimum or min(PQ), and delete-minimum or dm(PQ) may be defined with axiomatic semantics as follows.
1. new() returns a priority queue
2. min(insert(v, new())) = v
3. dm(insert(v, new())) = new()
4. min(insert(v, insert(w, PQ))) = if priority(v) < priority(min(insert(w, PQ))) then v else min(insert(w, PQ))
5. dm(insert(v, insert(w, PQ))) = if priority(v) < priority(min(insert(w, PQ))) then insert(w, PQ) else insert(v, dm(insert(w, PQ)))

where PQ is a priority queue, v and w are items, and priority(v) is the priority of item v.
Hinweis: Beim NIST findet man ein eigenes Wörterbuch über Datenstrukturen, siehe `http://www.nist.gov/dads/`

Zur Implementierung von Vorrangwarteschlangen

1. Mit AVL-Bäumen kann man die Operationen insert, min und dm in logarithmischer Zeit erhalten.

2. Alternativ bieten sich Min-Heaps (manchmal auch *partiell geordnete Bäume* genannt) an:
wie aus VL9 bekannt, kostet Einfügen sowie dm logarithmische Zeit, die Minimumssuche selbst geht sogar in konstanter Zeit.

Anwendungen

Prioritätswarteschlangen können für die Implementierung diskreter Ereignissimulationen (siehe VL über Simulation von N. Müller) genutzt werden.
Dabei werden zu den jeweiligen Ereignissen als Schlüssel die Zeiten berechnet, das Ereignis-Zeit-Paar in die Vorrangwarteschlange eingefügt und die Vorrangwarteschlange gibt dann das jeweils aktuelle (d.h. als nächstes zu verarbeitende) Ereignis aus.
Greedy-Algorithmen machen ebenfalls von Prioritätswarteschlangen Gebrauch, da dort häufig das Minimum, bzw. Maximum, einer Menge bestimmt werden muss.
Wir werden noch in der letzten VL sehen, wie Vorrangwarteschlangen beim Algorithmus von Kruskal zum Einsatz kommen, ebenso wie die in der kommenden VL vorgestellte Datenstruktur für Äquivalenzklassen.