Inhalt

11 Suchbäume
- Binäre Bäume, DS Binbaum
- Binäre Suchbäume
- AVL-Bäume

Bäume

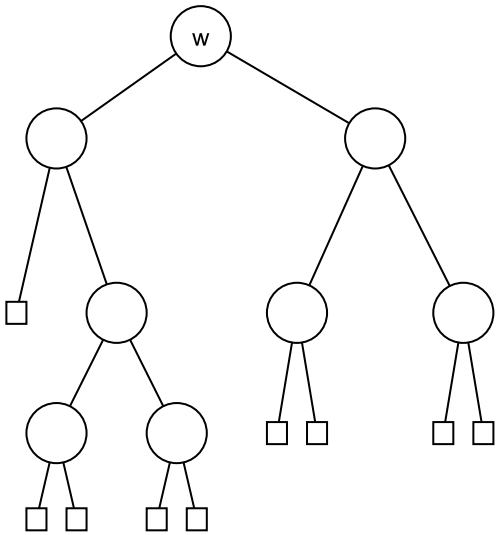
- Ein Baum besteht aus einer Menge von Knoten V („vertex“) und einer Menge von Kanten E („edge“). Die Kanten definieren eine „Eltern-Kind“-Relation auf der Knotenmenge: $E \subseteq V \times V$
- Ein Baum kann leer sein, dh $V = \emptyset$ (und damit auch $E = \emptyset$)
  Der leere Baum wird mit $\square$ bezeichnet.
- Ist $e = (a, b)$ eine Kante des Baums, dann nennt man a den „Vater“ (oder Elternknoten) von b und b nennt man „Kind“ von a.
- Ein nicht-leerer Baum hat genau einen speziellen Wurzel-Knoten, der keinen Elternknoten hat.
- Jeder andere Knoten des Baums hat genau einen Elternknoten.
  Ein Elternknoten kann mehrere Kinder haben.
- Jeder Knoten w definiert einen Teilbaum, dessen Wurzel $= w$ ist.

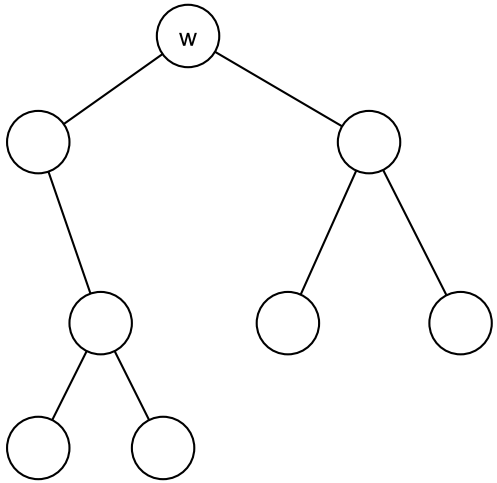
Spezialfall: Binärbaum

Wir bertachten speziell Binärbäume, für die gilt:

- Jeder Knoten hat höchstens zwei Kinder.
  Oder anders ausgedrückt:
  Jeder Knoten hat genau zwei Kinder, die aber (einzeln oder beide) leer sein können.
- Die (beiden) Kinder werden auch als „linker Sohn“ bzw „rechter Sohn“ bezeichnet.

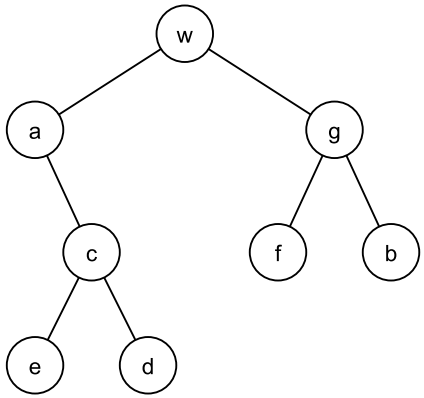
Darstellung mit expliziter Angabe der leeren Teilbäume oder ohne:





Knoten, Blätter, Pfade

- Die Knoten können auch einen „Inhalt“ (Werte oder Bezeichnungen) haben.
- Knoten mit (einem oder mehreren) Kindern heißen auch innere Knoten.
- Knoten ohne weitere Kinder (oder mit nur leeren Bäumen als Kindern) heißen auch Blätter.
- Eine Folge von Knoten von der Wurzel zu einem Blatt heißt Pfad.



Höhe und Anzahl der Knoten

- Die Höhe eines nicht-leeren Baumes ist rekursiv definiert:

$$h(b) = \begin{cases} 0 & \text{falls } b = \text{ leer} \\ 1 + max(h(t_l), h(t_r)) & \text{sonst} \end{cases}$$

  Die Höhe ist also gleich der Anzahl der Kanten eines längsten Pfad im Baum.
- Ein Binärbaum der Höhe h enthält höchstens $2^{h+1} - 1$ -viele Knoten.

$$n \leq 2^{h+1} - 1$$

- Ein Baum heißt „balanciert“, wenn sich in jedem Knoten die Höhe der Teilbäume um höchstens 1 unterscheidet.
- Ein balancierter Baum mit n Knoten hat eine Höhe von $\mathcal{O}(\log_2(n))$

$$h \in \mathcal{O}(\log_2(n))$$

Implementierung: DS BinBaum

- Ähnlich wie EVL bzw DVL, aber
- zwei Referenzen auf zwei Nachfolger (linker Sohn bzw rechter Sohn)

```
public class BinBaum<T> {

    protected Knoten root;

    // innere Klasse ///////////////
    protected class Knoten{
        protected T val;
        protected Knoten ltb;
        protected Knoten rtb;

        public Knoten(T v){
            val = v;
            ltb = null;
            rtb = null;
        }
    }
```

Aufbau eines BinBaums

Zum Aufbau (Einfügen von Elementen) nutzen wir drei Konstruktoren:

```
public BinBaum(Knoten l, T v, Knoten r) {
    root = new Knoten(v);
    root.ltb = l;
    root.rtb = r;
}

public BinBaum(T v) {
    this(null, v, null);
}

public BinBaum() {
    root = null;
}
```

Bemerkungen

1. Knoten $\neq$ Baum, aber ...
   - wir „identifizieren“ in unserer Sprechweise häufig einen Knoten w mit dem Teilbaum, dessen Wurzel der Knoten w ist.
   - In den Implementierungen arbeiten wir nur mit Knoten!

2. Da wir i.f. spezielle Suchbäume als Unterklasse von BinBaum implementieren wollen, deklarieren wir das Attribut root und die innere Knotenklasse und ihre Attribute als protected.

Rekursion

Viele Methoden lassen sich leicht rekursiv implementieren, zB

- **int** size ()
- **int** hoehe()
- String toString ()

Bemerkung:

- Das ist zwar „elegant“, aber meist nicht sehr effizient.
- Häufig genutzt, um (in Java-Syntax oder Pseudocode) die Semantik der Operationen zu definieren.
- Für reale Implementierungen nach Möglichkeit zu vermeiden!

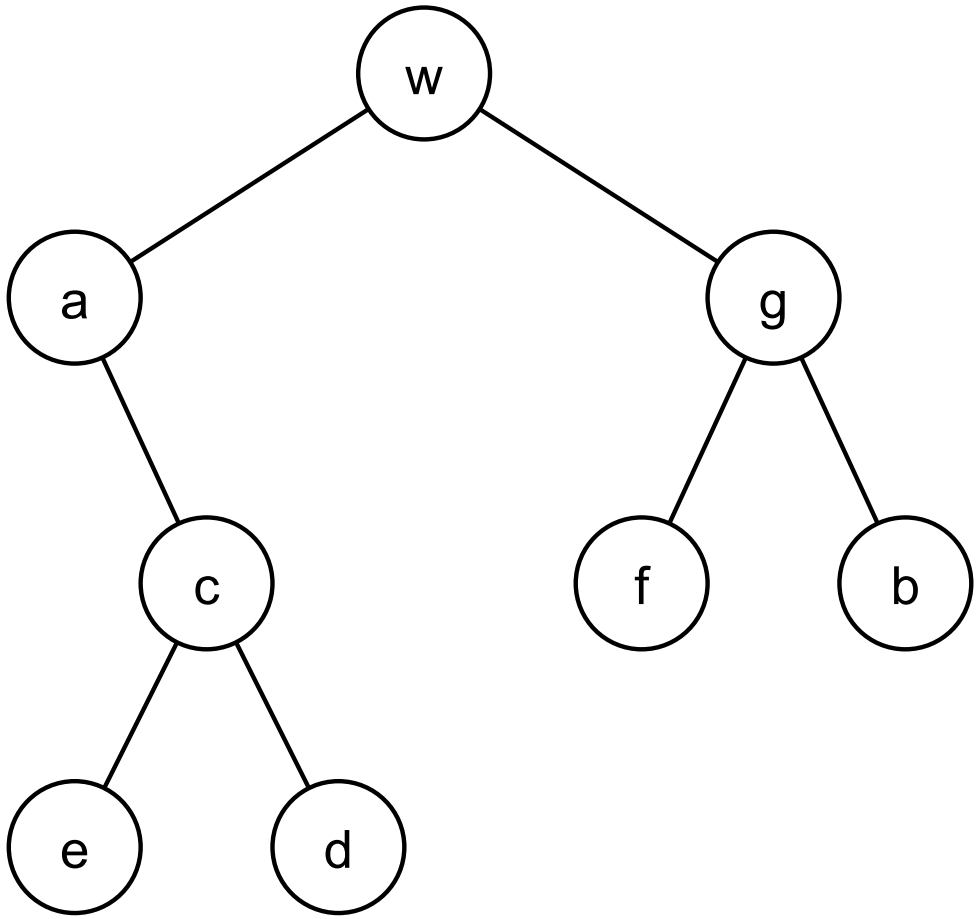
Baum-Traversierung

Neben dem Breitendurchlauf („BFS - Breadth- First-Search“ $\rightsquigarrow$ Übung), in dem die Knoten eines Baums „ebenenweise“ besucht werden, gibt es drei Varianten von Tiefendurchläufen, in denen zunächst jeweils ein Pfad von der Wurzel zu einem Blatt verfolgt wird:

- preorder: zuerst der Wurzelknoten, dann (rekursiv) der linke Teilbaum, dann (rekursiv) der rechte Teilbaum
- inorder: zuerst (rekursiv) der linke Teilbaum, dann die Wurzel, dann (rekursiv) der rechte Teilbaum
- postorder: zuerst (rekursiv) der linke Teilbaum, dann (rekursiv) der rechte Teilbaum, zuletzt die Wurzel

Beispiel

Traversierungen des Baums



inorder: Beispiel-Implementierung

```
public DynArray<T> inorder() {
    return inorder(root);
}

private DynArray<T> inorder(Knoten k) {
    DynArray<T> arr = new DynArray<>();
    if (k == null) {
        return arr;
    }

    DynArray<T> links = inorder(k.ltb);
    DynArray<T> rechts = inorder(k.rtb);

    for(T v: links)
        arr.add(v);
    arr.add(k.val);
    for(T v: rechts)
        arr.add(v);
    return arr;
}
```

Inhalt

11 Suchbäume
- Binäre Bäume, DS Binbaum
- Binäre Suchbäume
- AVL-Bäume

Binärer Suchbaum

Ein Binärbaum b mit Knoteneinträgen aus einer Menge T heißt (binärer) Suchbaum, wenn

- auf T eine (totale) Ordnungsrelation $\leq$ definiert ist und
- für jeden Teilbaum (ltb, w, rtb) von b gilt:
  für alle Knoten $x \in ltb$ und alle Knoten $y \in rtb$ ist

$$x.value < w.value < y.value$$

Bemerkungen: Wir setzen also zweierlei voraus:

- Die Knoteneinträge sind von einem Datentyp, der Comparable implementiert.
- Der Baum enthält keine Duplikate.
- Mit den Operationen get(), contains(), insert(), delete() können Binäre Suchbäume also zu Darstellung von Mengen genutzt werden.

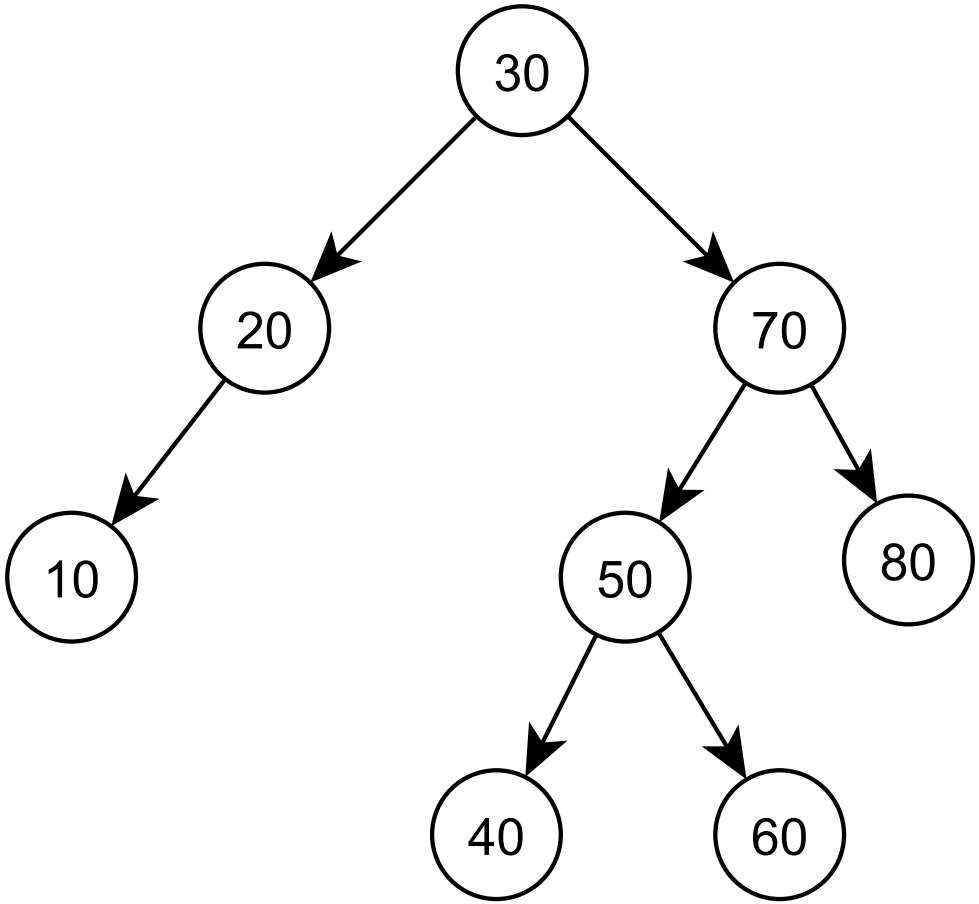
Suche in einem Suchbaum

- Die Methode **boolean** contains(T e) muss dann nicht mehr alle Knoten des Baums durchsuchen, sondern nur noch die Knoten entlang eines Pfades von der Wurzel zu einem Blatt.
- Für einen balancierten Suchbaum bedeutet dies, dass contains mit einem Aufwand von $\mathcal{O}(\log n)$ arbeitet.
- Algorithmus in Pseudocode (rekursiv formuliert):

```
boolean contains(Knoten k, T e):
    falls  k = leer:  return  false
    falls  e = k.value:  return  true
    falls  e < k.value:  return  contains(k.ltb, e)
    falls  e > k.value:  return  contains(k.rtb, e)
```

Beispiel

Suchen des Wertes 50 bzw des Wertes 12 im Baum



Einfügen in einen Suchbaum

Nach demselben Prinzip können Elemente in einen Suchbaum b eingefügt werden, so dass die Suchbaum-Eigenschaft erhalten bleibt:

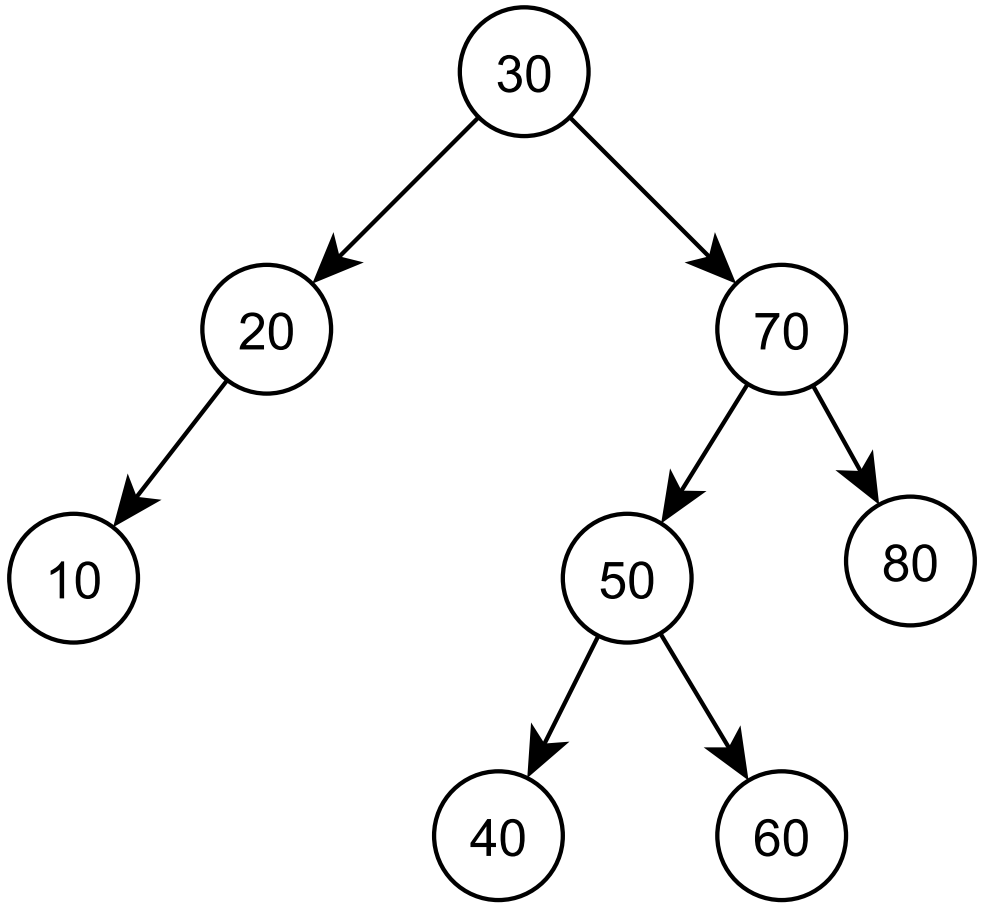
- Suche den einzufügenden Wert e im Baum b, bis
- entweder der Wert gefunden wurde:
  dann ist der Wert bereits enthalten, also nichts einzufügen
- oder die Suche bei einem leeren Knoten (also erfolglos) endet:
  dann ist genau dies die (eindeutig bestimmte) Position, an der das Element einzufügen ist.

„Problem“:

- muss beim Einfügen (k == **null**) auf den Vaterknoten von k zugreifen
- daher in vielen Implementierungen: verwalte in der Knotenklasse ein weiteres Attribut Knoten father

Beispiel

Einfügen der Werte 35 und 25 in den Baum



Löschen aus einem Suchbaum

Beim Löschen von Elementen aus einem binären Suchbaum sind drei Fälle zu unterscheiden:

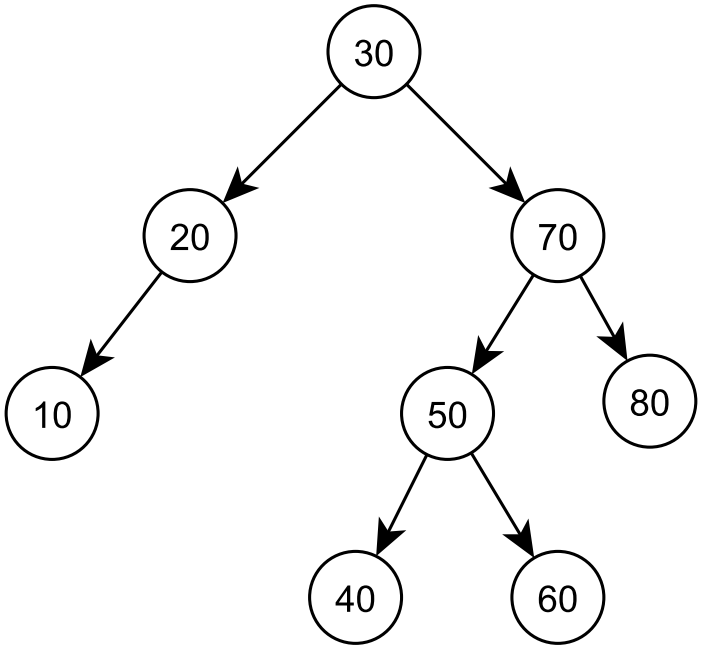
1. Das zu löschende Element ist Blatt des Baums.
2. Das zu löschende Element hat (nur) genau einen Kindknoten.
3. Das zu löschende Element ist innerer Knoten mit zwei nicht-leeren Kindknoten.

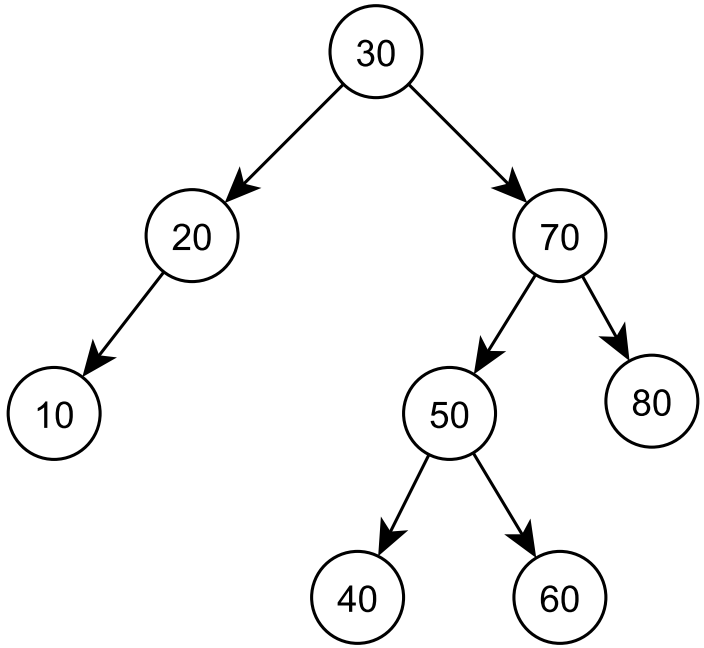
Die Fälle 1 und 2 sind leicht zu lösen:

1. Entferne das Blatt.
2. Der (einzige) Kindknoten „rutscht“ eine Ebene nach oben.

Beispiel

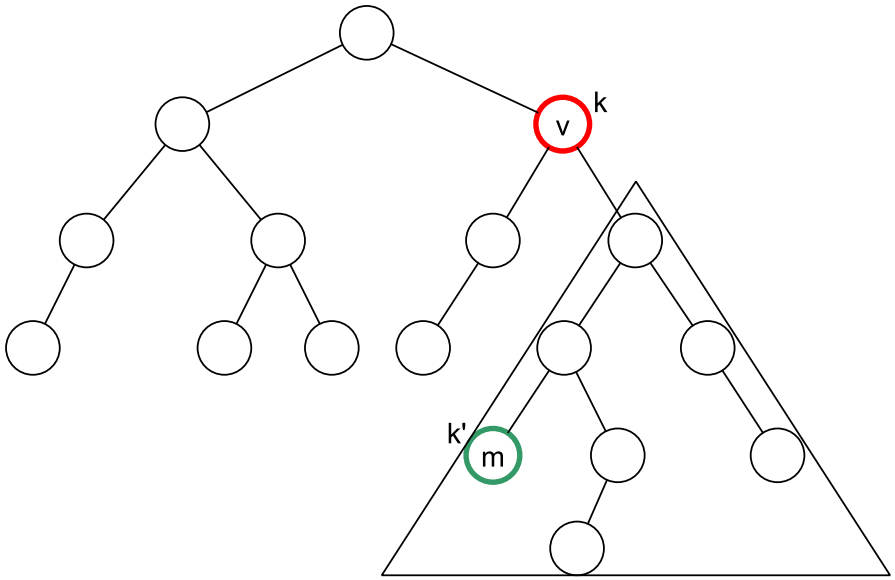
Löschen der Werte 40 (Fall 1) bzw 20 (Fall 2) aus dem Baum





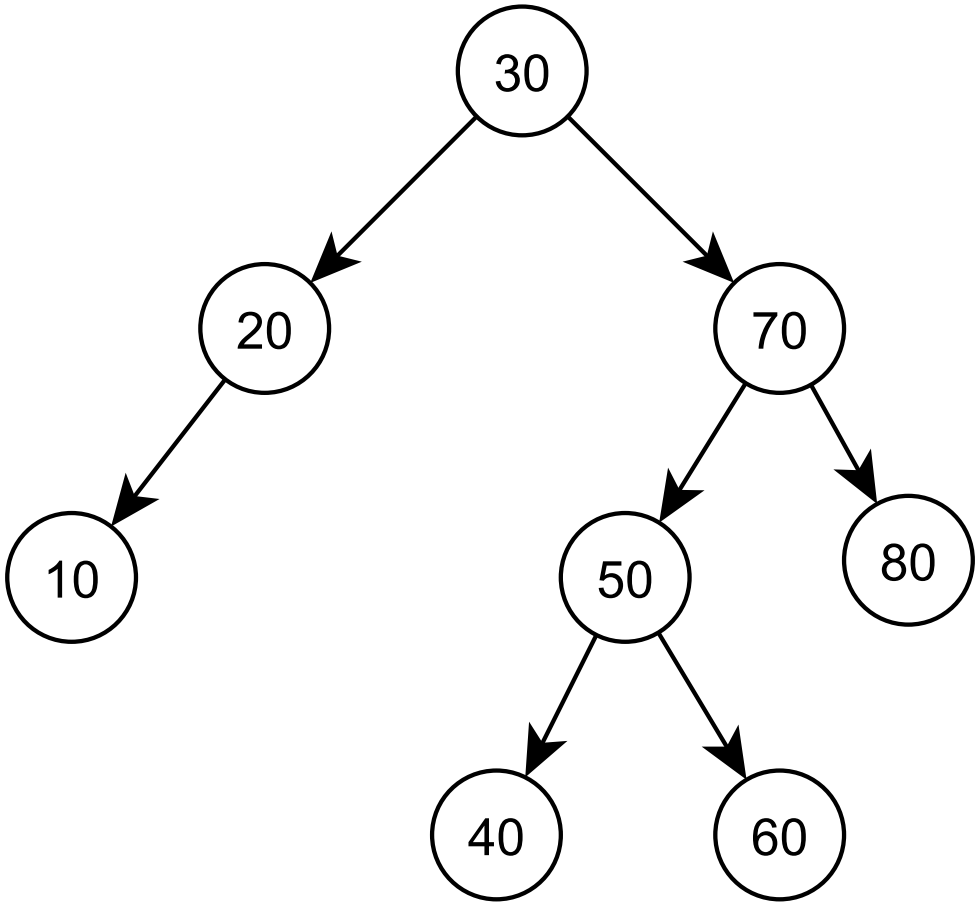
3. Fall: Löschen eines inneren Knotens

- Ersetze im zu löschenden Knoten k seinen Wert v durch den kleinsten Wert m im rechten Teilbaum von k: das ist der Wert des am weitesten links liegenden Knoten k' im rechten Teilbaum von k.
- Lösche den Knoten k'.
- Der Knoten k' hat garantiert keinen linken Sohn.
  Das Löschen von Knoten k' erfolgt also nach Fall 1 oder Fall 2.



Beispiel

Löschen der Wertes 70 aus dem Baum



Fazit: Komplexität der Operationen

In einem binären Suchbaum ist die Operation

- get() in $\mathcal{O}(1)$ implementierbar.

Die die drei Operationen

- contains(T e)
- insert (T e)
- delete (T e)

verfolgen jeweils nur genau einen Pfad des Baums.

- Die Komplexität der drei Operationen liegt also in $\mathcal{O}(h)$, wobei h die Höhe des Baums ist.
- Für einen balancierten Baum ist das also $\mathcal{O}(\log n)$.

Implementierung von Mengen als Suchbaum

- Suchbäume können verwendet werden, um Mengen über einem „vergleichbaren“ Typ zu implementieren.
- Alle drei Operationen lassen sich in $\mathcal{O}(\log n)$ Zeit implementieren, falls die Einfüge- und Lösch-Operationen einen balancierten Baum erzeugen.
- Damit wäre die Implementierung im best case besser als alle bisher vorgestellten Implementierungen.
- Nachteil: im worst case benötigen alle drei Operationen $\mathcal{O}(n)$ Zeit.
- Der worst case tritt ein, wenn die Einfüge- bzw Lösch-Reihenfolge der Elemente den Baum zu einer Liste „entarten“ lässt.

Sortieren mittels Suchbäumen

- Die inorder-Traversierung eines binären Suchbaums liefert die Elemente in aufsteigend sortierter Reihenfolge.
- Suchbäume können also zur Sortierung von Datenmengen genutzt werden.
- Eine Menge kann in $\mathcal{O}(n \cdot \log n)$- Zeit sortiert werden, wenn der entsprechende Suchbaum balanciert ist.
- Wie kann der schlechteste Fall (Höhe $= \mathcal{O}(n)$) vermieden werden und ein guter Fall (Höhe $= \mathcal{O}(\log n)$) garantiert werden?

Inhalt

11 Suchbäume
- Binäre Bäume, DS Binbaum
- Binäre Suchbäume
- AVL-Bäume

AVL-Bäume - Balancierte Suchbäume

Ein binärer Suchbaum heißt AVL-Baum[1], wenn für jeden Knoten k gilt:

$$|h(k.ltb) - h(k.rtb)| \leq 1$$

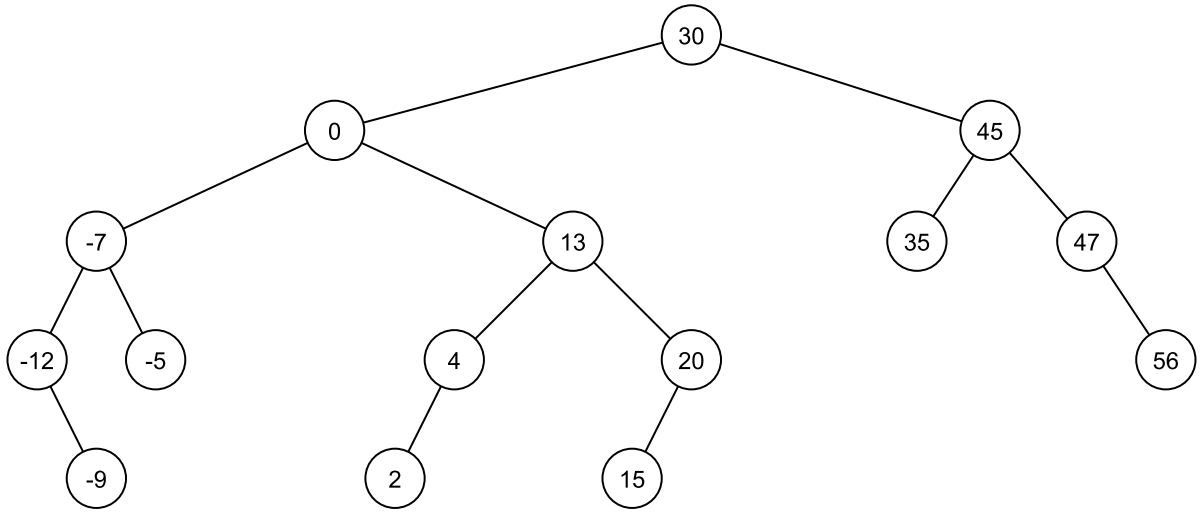
dh wenn sich in jedem Knoten k die Höhe der beden Teilbäume um höchstens 1 unterscheidet.

- Der Wert $h(k.ltb) - h(k.rtb)$ heißt auch „Balance“ im Knoten k.
- Für einen AVL-Baum T mit n Knoten gilt:

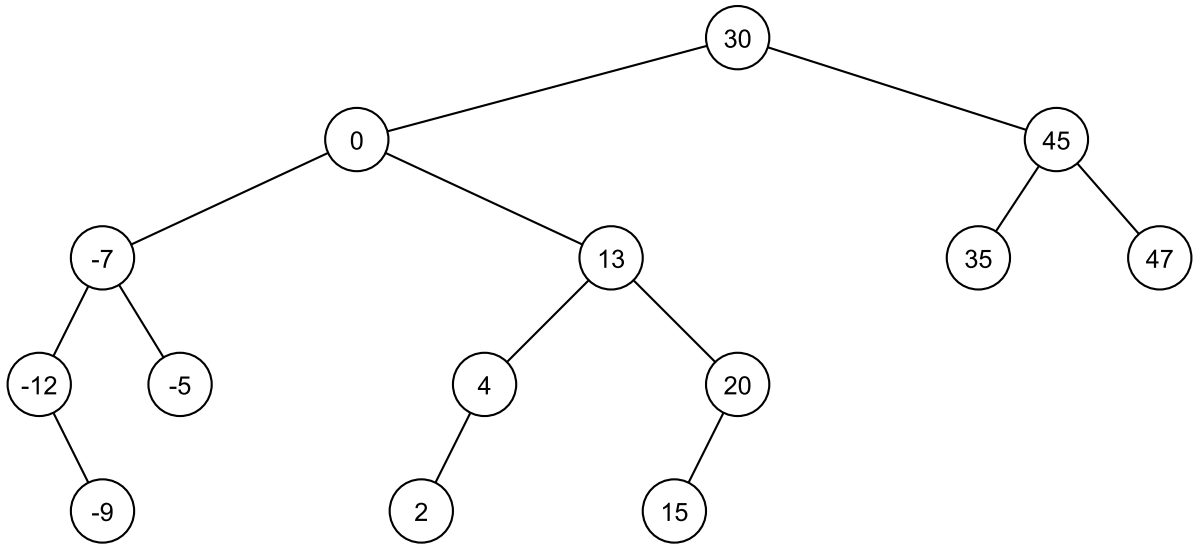
$$h(T) \leq 1.5 \cdot \log_2(n + 1.5) \in \mathcal{O}(\log(n))$$

[1] AVL steht für die Namen der beiden Entwickler G.M.Adelson-Velski und J.M.Landis.

Beispiele



AVL-Baum



Kein AVL-Baum

Rebalancierung

- Bäume der Höhe 0 oder 1 sind immer balanciert.
- Die Balance kann durch Einfügen oder Löschen eines Knotens gestört werden.
- Daher: Kontrolliere nach jedem Einfügen/Löschen, ob die AVL-Eigenschaft in einem Knoten verletzt ist.
- Führe ggf. ein rebalance aus, das die AVL-Eigenschaft wieder herstellt.
- Das Rebalancieren geschieht durch Rotation von Knoten.

Beobachtung:

- Wenn in einem Knoten w die Balance gestört ist, dann hat der Teilbaum $T(w)$ mindestens die Höhe 2 und dann sind an der Verletzung immer ein Kind v und ein Enkel u von w beteiligt.
- Die drei beteiligten Knoten u, v, w haben immer insgesamt vier Teilbäume T_1, T_2, T_3, T_4 (die ggf. leer sein können).

Vier mögliche Lagen von w-v-u

1. $u \leq v \leq w$
2. $v \leq u \leq w$
3. $w \leq u \leq v$
4. $w \leq v \leq u$

In allen 4 Fällen seien die Teilbäume so nummeriert, dass für alle Knoten $p \in T_1,\ q \in T_2,\ r \in T_3,\ s \in T_4$ gilt:

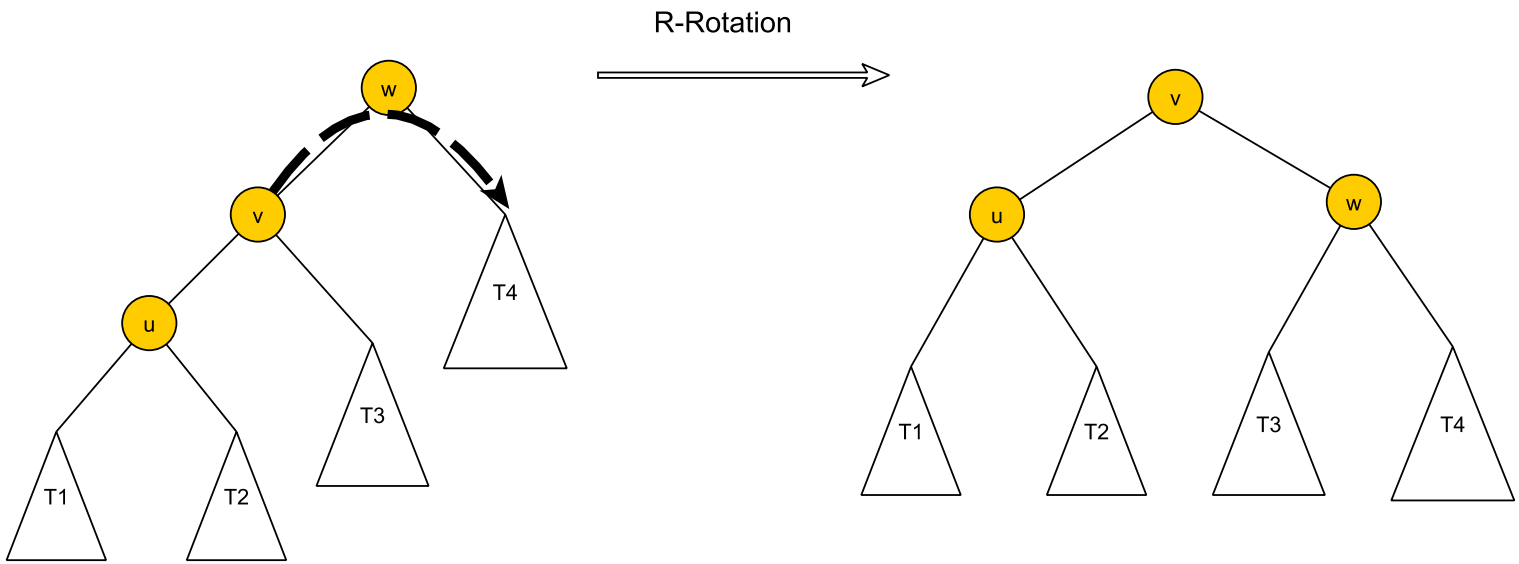
$$p.val \leq q.val \leq r.val \leq s.val$$

Oder kurz:

$$T_1 \leq T_2 \leq T_3 \leq T_4$$

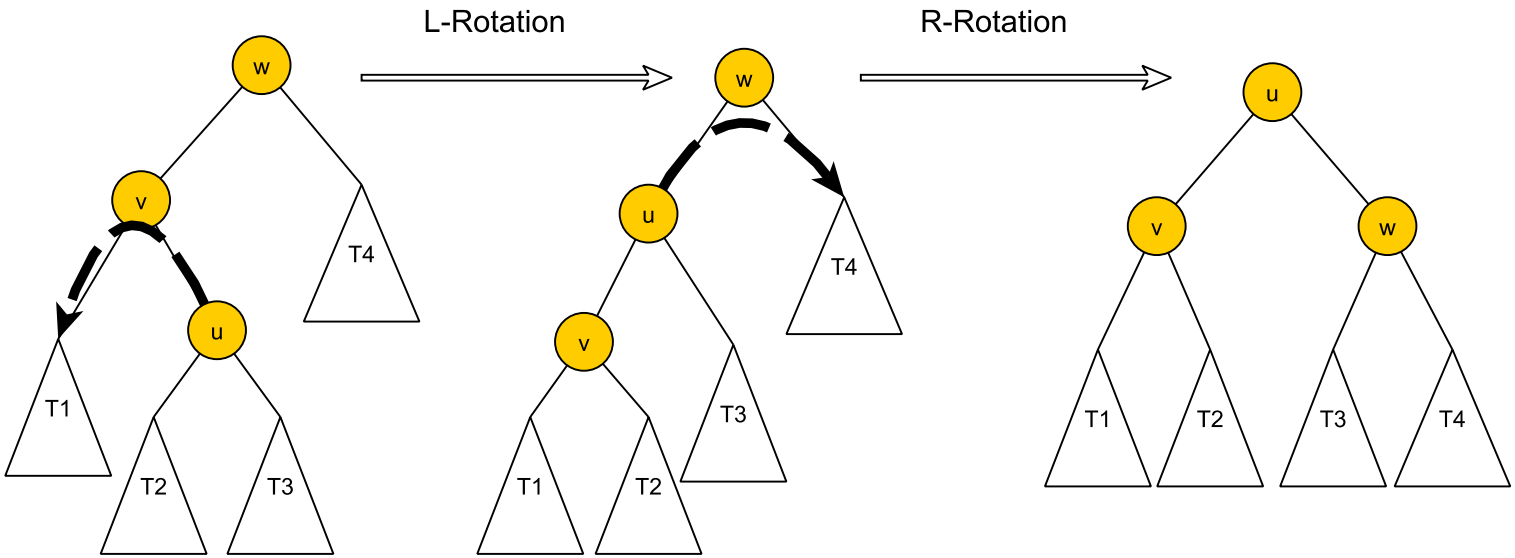
1. Fall: $u \leq v \leq w$

- Führe eine „Rechts-Rotation“ aus, um Balance wieder herzustellen.
- Anschließend hat der Baum eine um 1 geringere Höhe als vor der Rotation.



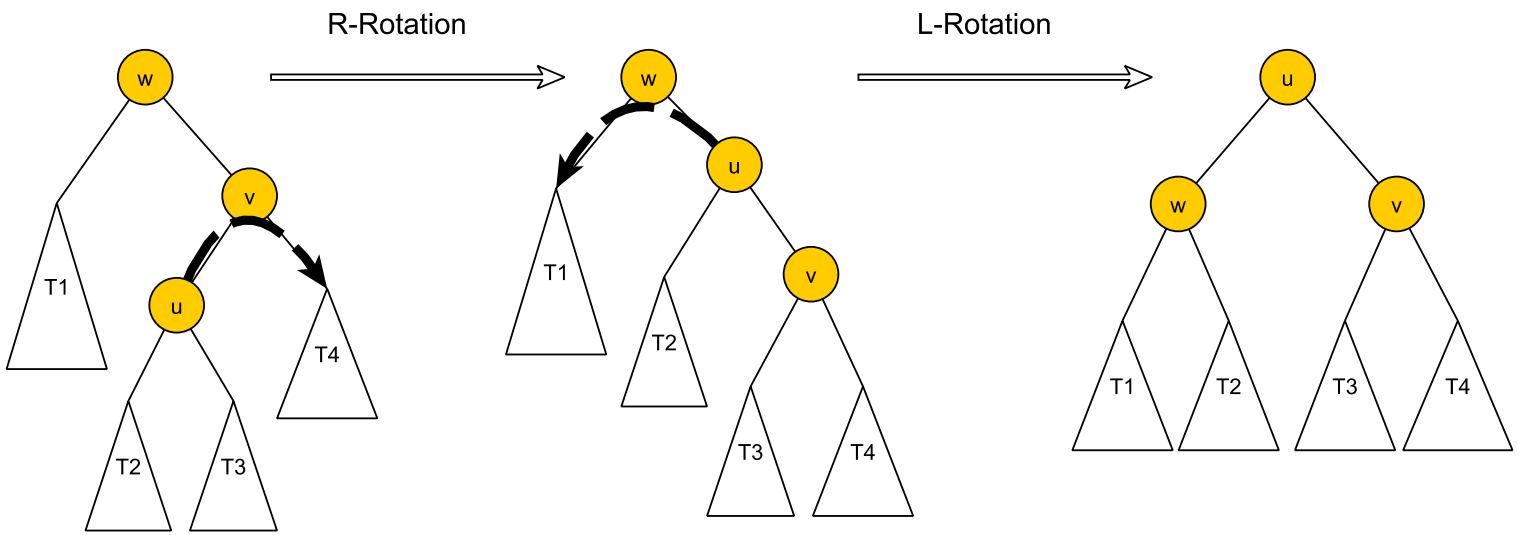
2. Fall: $v \leq u \leq w$

- Führe eine „Doppel-Rotation Links-Rechts“ aus, um Balance wieder herzustellen.



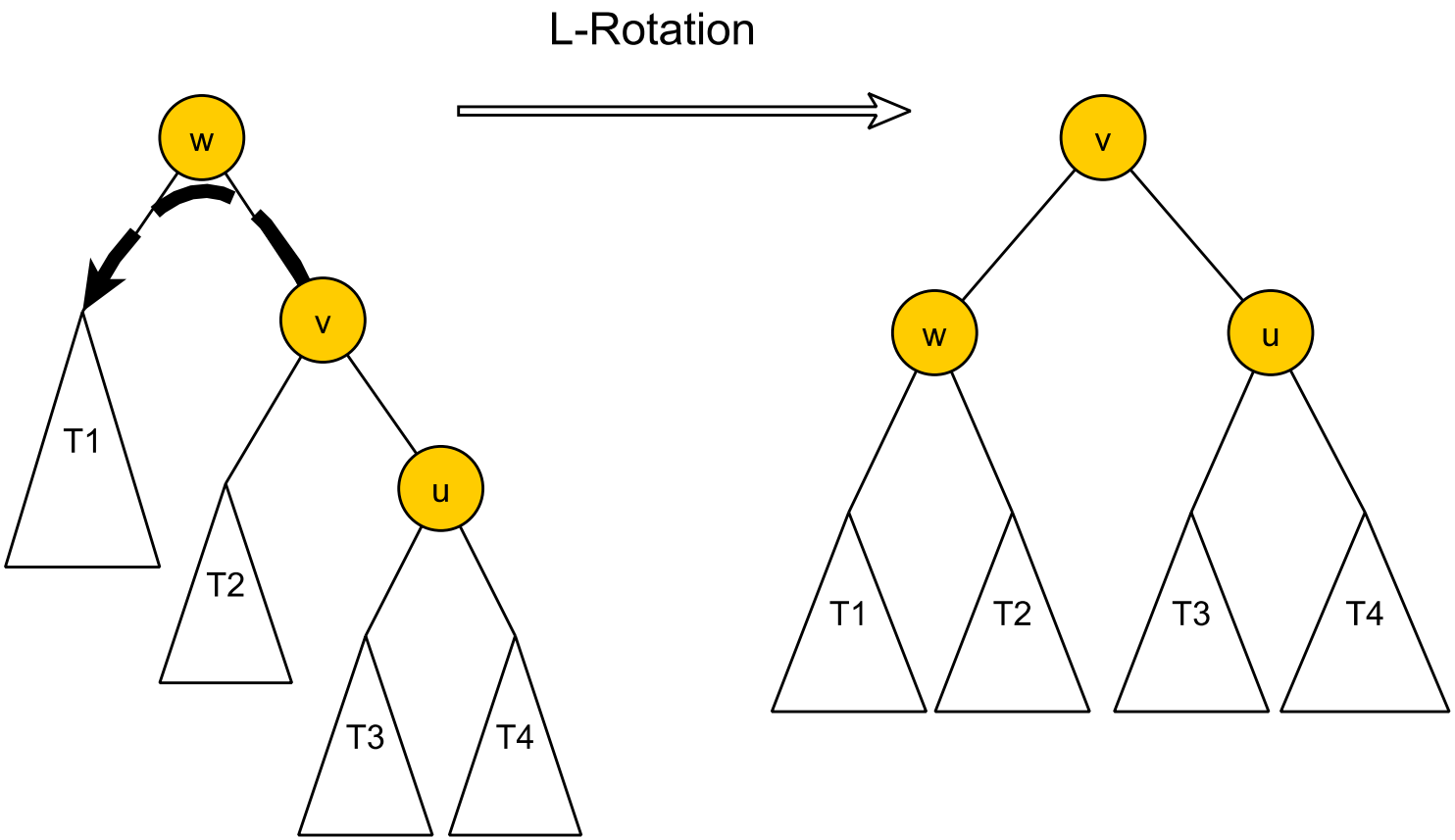
3. Fall: $w \leq u \leq v$

- Führe eine „Doppel-Rotation Rechts-Links“ aus, um Balance wieder herzustellen.



4. Fall: $w \leq v \leq u$

- Führe eine „Links-Rotation“ aus, um Balance wieder herzustellen.



Rebalance

In allen 4 Fällen hat der Teilbaum, der im Ungleichgewicht war, nach der Rotation eine um 1 geringere Höhe als vor der Rotation.

- Rebalance nach insert:
  - Falls der Baum vor dem Einfügen balanciert war und
  - ... das Ungleichgewicht durch eine Einfüge-Operation verursacht wurde,
  - hat der Teilbaum nach der Rotation wieder die ursprüngliche Höhe
  - ... und der Gesamtbaum ist wieder balanciert.
- Rebalance nach delete:
  - Falls der Baum vor dem Einfügen balanciert war und
  - ... das Ungleichgewicht durch eine Lösch-Operation verursacht wurde,
  - hat der Teilbaum nach der Rotation eine geringere Höhe.
  - Das Ungleichgewicht kann sich auf den Vaterknoten des betrachteten Teilbaums fortpflanzen.
  - Die Notwendigkeit zu Rotationen kann sich nach oben bis zur Wurzel fortpflanzen.

Analyse

- Höhe und Balance eines Baumes können als zusätzliche Attribute in jedem Knoten gespeichert und bei jedem Einfügen und Löschen aktualisert werden.
- Jede Rotation kann in konstanter Zeit ausgeführt werden.
- Einfügen in einen AVL-Baum kostet $\mathcal{O}(\log n)$-Zeit für das Finden der korrekten Einfüge-Position, plus konstante Zeit für das Einfügen und ggf notwendige Rotation.
- Löschen aus einem AVL-Baum kostet $\mathcal{O}(\log n)$-Zeit für das Finden des Wertes plus konstante Zeit für das Löschen plus maximal $(\log n)$-viele Rotationen.

Fazit:

- AVL-Bäume garantieren `contains()`, `insert()` und `delete()` in $\mathcal{O}(\log n)$-Zeit.
- Die Balance wird in jedem Schritt gewahrt.