Einführung in die Informatik II
Entwurf durch Verträge:
2. Prädikatenlogik,
Verträge für Lösungsklassen

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2004

Mai 2004

Aussagenlogik (Wiederholung)

- Die **Aussagenlogik** behandelt Formeln, in denen nur Terme vom Typ Boolean aufreten.
  - p = *"Rostig ist Partner im Treueprogrammm Gold & Silber."*
  - q = *"Lustig ist Partner im Treueprogrammm Gold & Silber."*
  - p ∨ q = true
- "Rostig ist Partner im Treueprogramm Gold & Silber" ist ein Beispiel einer ***elementaren Aussage,***
- p und q heißen ***Identifikatoren***,
- p ∨ q ist ein Beispiel eines ***booleschen Term***s,
- p ∨ q = true ist eine ***Formel***, die abhängig von der ***Belegung*** von p und q entweder wahr oder falsch ist.

Prädikatenlogik

- Die **Prädikatenlogik** ist allgemeiner als die Aussagenlogik. Sie untersucht Aussagen, die auch Variablen enthalten können, wobei die Variablen Werte aus einer beliebigen Trägermenge (nicht nur Boolesche Werte) annehmen können.
  - Eine Aussage, die auch Variable enthalten kann, heißt **prädikatenlogischer Ausdruck** oder oft kurz **Prädikat**.
- **Beispiel eines Prädikats:**
  - *"x ist Programmpartner im 'Treueprogramm Gold & Silber'."*, wobei `x ∈ M = {Rostig, Luftig, Oktoni}`
  - *"x ist Programmpartner im 'Treueprogramm Gold & Silber'."* ist wahr, wenn `x` den Wert `Rostig` hat, aber falsch, wenn `x` den Wert `Lustig` hat.
- Die Prädikatenlogik ist ein wichtiges Werkzeug für die Spezifikation von Verträgen und für die Verifikation von Programmen.
- OCL ist eine Prädikatenlogik.

Definition: Prädikat

- **Definition Prädikat:** Eine Abbildung
  - P: $M_1 \times M_2 \times \ldots \times M_n \rightarrow \mathbf{B}$

  von Tupeln über gegebenen Trägermengen $M_1 \times M_2 \times \ldots \times M_n$ in die Menge der Wahrheitswerte **B** heißt ein ***n-stelliges Prädikat*** oder eine ***n-stellige Boolesche Funktion*** über der Menge $M_1 \times \ldots \times M_n$.
- **Beispiele von n-stelligen Prädikaten:**
  - Die Abbildung *"x studiert Informatik"*, wobei x ein Element der Menge M_1 aller Studenten ist, ist ein einstelliges Prädikat über M_1.
  - Die Abbildung *"$x \geq y$"*, wobei x und y natürliche Zahlen sind, definiert ein zweistelliges Prädikat.
  - Allgemein: Die Abbildungen $\leq$, $\neq$, $=$, $>$, $<$ sind zweistellige Prädikate über der Menge der natürlichen oder reellen Zahlen

Allgemeingültigkeit und Erfüllbarkeit von Prädikaten

- Für uns ist interessant, dass wir aus Prädikaten über der gesamten Trägermenge elementare Aussagen formen können:
  Sei M eine Trägermenge.
- Für ***jedes Prädikat p(x)*** können wir immer die folgenden zwei elementaren ***Aussagen*** formulieren:
  - 1. *"Für alle $x \in M$ gilt p(x)"*
    - Ist diese Aussage wahr, dann heißt das Prädikat p(x) **allgemeingültig.**
  - 2. *"Es gibt mindestens ein $x \in M$, so dass p(x) gilt"*
    - Ist diese Aussage wahr, dann heißt das Prädikat p(x) **erfüllbar.**
- **Beispiel**: Sei M = {Schaldi, Oktoni, Rostig, Luftig, Lustig} und p(x) = *"x ist Programmpartner im 'Treueprogramm Gold & Silber'."*
  - p(x) ist nicht allgemeingültig (Lustig ist nicht im Treueprogramm), aber erfüllbar (Schaldi ist im Treueprogramm).

Quantoren und Quantifizierung

- Die Aussage "Für alle x ∈ M gilt p(x)" schreibt man auch als
  - ∀x ∈ M: p(x) oder
  - ∀m x:p(x), wobei m der Typ (Klasse) von M ist.
    - Sprechweise: "Für alle x vom Typ m gilt p(x)“
- Die Aussage "Es gibt mindestens ein x ∈ M, für das p(x) gilt." schreibt man auch kurz als
  - ∃ x ∈ M: p(x) oder
  - ∃ m x: p(x), wobei m der Typ (Klasse) von M ist.
    - Sprechweise: "Es gibt mindestens ein x vom Typ m, für das p(x) wahr ist.
- Den Operator ∀ nennt man den Allquantor oder Universalquantor, den Operator ∃ nennt man den Existenzquantor
- Definition: Die Umwandlung eines Prädikats p(x) in eine Aussage, indem man es mit einem Quantor versieht, heißt Quantifizierung.

Interpretation von quantifizierten Prädikaten

- Dem Term $\forall x \in M: p(x)$ ordnen wir den Wahrheitswert wahr zu, falls für alle $x \in M$ der Wert von $p(x)$ wahr ist, sonst falsch.

- Gegeben sei eine Belegung β. $\beta[a/x]$ ist dann die Belegung, die x mit a belegt und sonst mit β übereinstimmt:

$$\beta[a/x](z) = \begin{cases} a & \text{falls } z = x \\ \beta(z) & \text{sonst} \end{cases}$$

- Die Interpretation I_β von $\forall x \in M: p(x)$ ist:

$$I_\beta [\forall x \in M: p(x)] = \begin{cases} L & \text{falls für alle } a \in M: \; I_{\beta[a/x]} [p(x)] = L \\ O & \text{sonst} \end{cases}$$

- Die Interpretation I_β von $\exists x \in M: p(x)$ ist:

$$I_\beta [\exists x \in M: p(x)] = \begin{cases} L & \text{falls für ein } a \in M: \; I_{\beta[a/x]} [p(x)] = L \\ O & \text{sonst} \end{cases}$$

Termersetzungsregeln gibt es auch in der Prädikatenlogik

- Ein Identifikator x in der Aussage ∀m x:p(x) oder (¬∃ m x: p(x)) heißt **gebundener Identifikator**, denn er wird durch den Quantor gebunden.
- **Wiederholung** (aus Info I - Termersetzungssysteme)**:** Sei t ein Term.
  - wenn wir ***x*** durch ***y*** ersetzen, dann schreiben wir: t[y/x]
  - und lesen: "Substitution von x durch y".
- **Gesetz der Umbenennung für gebundene Identifikatoren:** Gebundene Identifikatoren können umbenannt werden, ohne dass sich die Bedeutung der Prädikate ändert.
  Sei t der Name eines Prädikates, also t = p(x). Es gelten:
  - (∀m x: t) = ∀m y: (t[y/x]), falls y nicht frei in t vorkommt
  - (∃ m x: t) = ∃ m y: (t[y/x]), falls y nicht frei in t vorkommt
- Aussagen, die durch Umbenennung gebundener Identifikatoren entstehen, sind semantisch äquivalent.
  Komplizierter wird es bei freien Identifikatoren.

Termersetzungsregeln bei freien Identifikatoren

- Sei x ein ein gebundener Identifikator und y ein freier Identifikator. Dann gilt folgende Termersetzungsregel:
  - (∀m x: t) [t'/y] = ∀m x:(t[t'/y])
- Eine kleine Komplikation haben wir, wenn x in t zwar gebunden, in t' aber frei ist.
  - Wenn wir in einem solchen Fall die Substitution [t'/y] ohne weiteres ausführen würden, wäre x plötzlich auch in t' gebunden!
- Lösung
  - Wir benennen x in der Aussage (∀m x:t) um, und zwar mit Hilfe des Gesetzes der Umbenennung für gebundene Identifikatoren: (∀m z:t)
    Dann erst nehmen wir die Substitution [t'/y] in (∀m z:t) [t'/y] vor.

Prädikate in OCL

- Die meisten Einschränkungen, die wir bisher in OCL formuliert haben, waren Prädikate.
- Beispiel: Der OCL-Ausdruck

  KundenKarte

  gedruckterName = kunde.titel.concat(kunde.name)

  beschreibt ein 4-stelliges Prädikat p: M × K × T × N → **B**

  – p(m,k,t,n): *"Auf der Karte k des Kunden m ist der Titel t und der Namen n des Kunden m gedruckt."*

  – Die Trägermengen sind:

    – M ={m: m ist Kunde im "Treueprogramm Gold & Silber"}

    – K = {k: k ist Kundenkarte im "Treueprogramm Gold & Silber"}

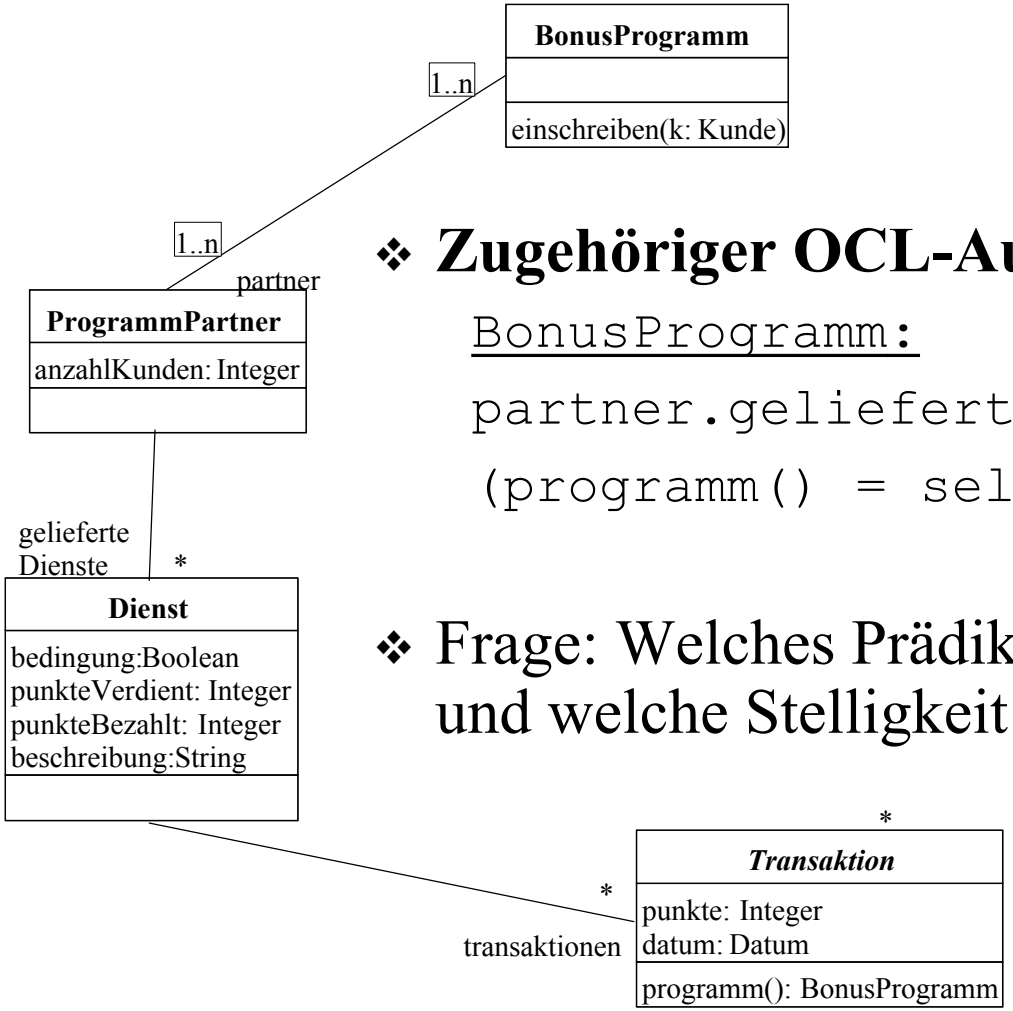
    – T = N = {n: n ist Zeichenkette über dem Alphabet A = {'a', ..., 'z'}
- OCL erlaubt auch die Quantifizierung von Prädikaten.

Quantoren in OCL

- OCL unterstützt beide Arten von Quantoren:
- Der Allquantor ∀ heißt in OCL **`forAll:`**
  - **`forAll`** nimmt einen OCL-Ausdruck **`o`** als Parameter und ist wahr, wenn dieser Ausdruck für alle Elemente einer Sammlung wahr ist, sonst ergibt **`forAll`** falsch.
  - Signatur:
    `collection->forAll(o:OClAusdruck):Boolean`
- Der Existenzquantor ∃ heißt in OCL **`exists`**
  - **`exists`** nimmt einen OCL-Ausdruck **`o`** als Parameter und ist wahr, wenn dieser Ausdruck für mindestens ein Element einer Sammlung wahr ist, sonst ergibt **`exists`** falsch.
  - Signatur:
    `collection->exists(o:OClAusdruck):Boolean`

Einsatz des Allquantors in OCL

- **Einschränkung:** *"Die Dienste, die ein Programmpartner in einem Bonusprogramm anbietet, sind nur in diesem Programm nutzbar, d.h. alle Transaktionen müssen innerhalb des Programms ablaufen."*



- **Zugehöriger OCL-Ausdruck:**

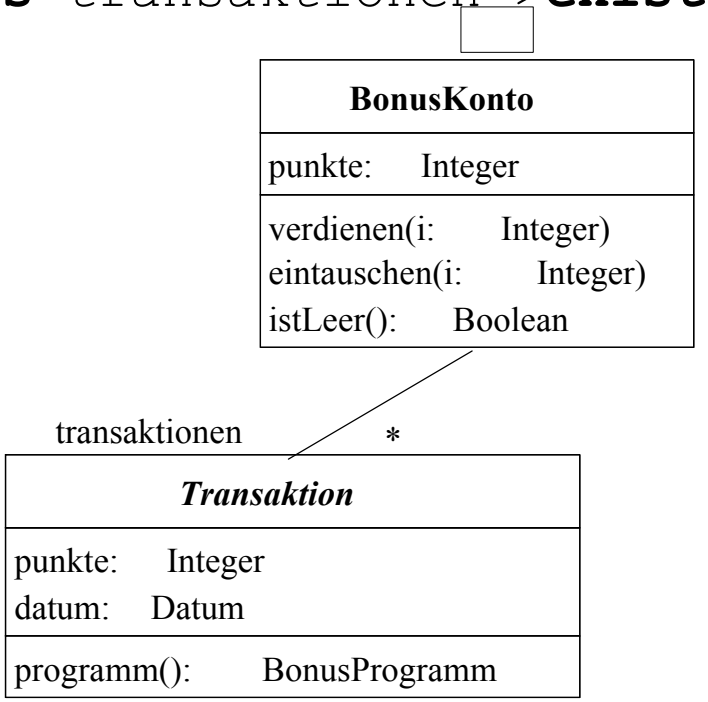
```
BonusProgramm:
partner.gelieferteDienste.transaktionen->forAll
(programm() = self)
```

- Frage: Welches Prädikat beschreibt dieser OCL-Ausdruck und welche Stelligkeit hat es?

Einsatz des Existenzquantors in OCL

- **Einschränkung:** "Wenn in einem Bonuskonto die Punkte größer als Null sind, dann gibt es mindestens eine Transaktion mit mehr als Null Punkten."
- **Zugehöriger OCL-Ausdruck:**

```
BonusKonto:
punkte > 0 implies transaktionen->exists(punkte > 0)
```



Beziehung zwischen Existenzquantor und Allquantor

- In der Prädikatenlogik können wir den Allquantor durch Negation über den Existenzquantor ausdrücken und umgekehrt:
  - (∀m x: p(x)) = (¬∃ m x: ¬p(x))
  - (∃ m x: p(x)) = (¬∀m x: ¬p(x))
- Für eine OCL-Sammlung **s** und einen OCL-Ausdruck **o** gilt also:
  - `s->forAll(o) = not (s->exists(not o))`
  - `s->exists(o) = not (s->forAll(not o))`
- Beispiel: Aus dem Prädikat

```
BonusKonto:
points > 0 implies transaktionen->exists(punkte > 0)
```

erhalten wir durch Anwendung der Termersetzungsregeln für Prädikate:

```
BonusKonto:
points > 0 implies
  not(transaktionen->forAll(not(punkte>0)))
```

Einsatz von Verträgen: Anwendungsklassen vs. Lösungsklassen

- Die TUMBoS-Klassen stammen aus der Anwendungsdomäne:
  - Die Verträge spezifizieren anwendungsbezogene Einschränkungen
- Verträge können auch für Klassen in der Lösungsdomäne (***Lösungs-klassen)*** vereinbart werden

  ⇒ Spezifikation anwendungsunabhängiger Einschränkungen
- Besonders interessant: Verträge für wiederverwendbare Lösungsklassen (als Bestandteil von Klassenbibliotheken):
  - Vertrag liefert dem Anwendungsentwickler Informationen über
    - Invarianten der Lösungsklassen
    - Eigenschaften von Algorithmen und Datenstrukturen

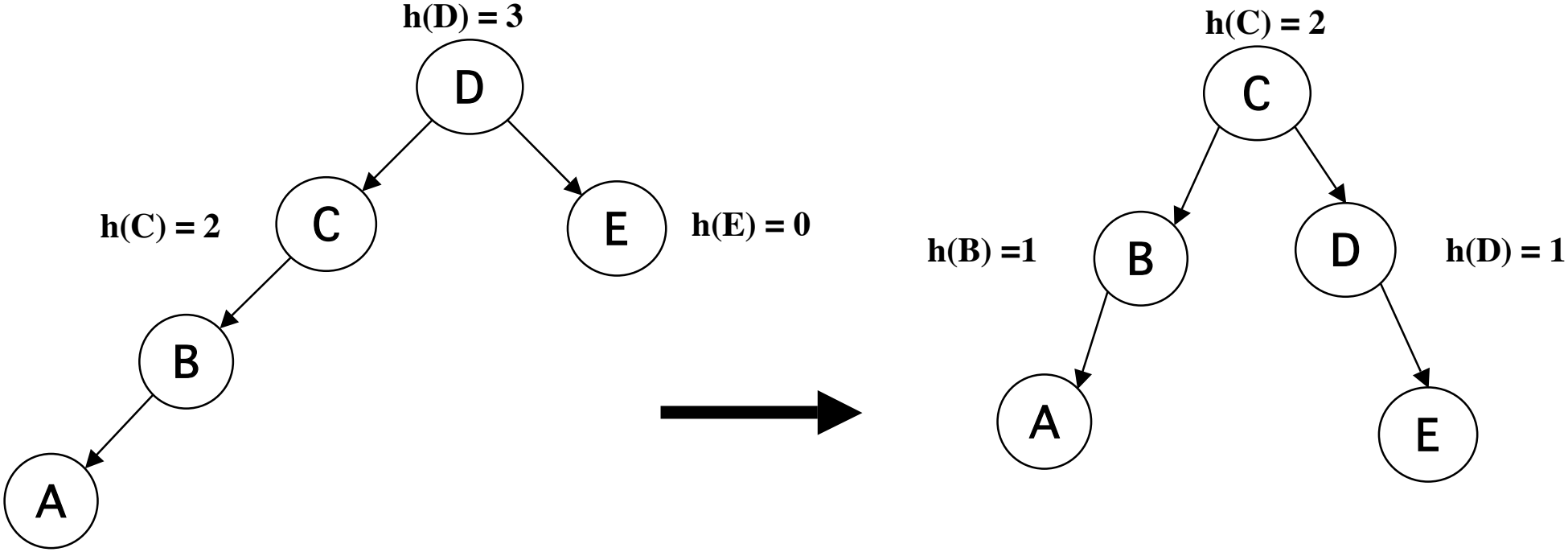
  ⇒ Unterstützung des Anwendungsentwicklers bei der Auswahl von Algorithmen und Datenstrukturen, die zur Lösung des jeweiligen Anwendungsproblems geeignet sind.
- Beispiel: Spezifikation von AVL-Bäumen und ihre Eigenschaften

AVL Bäume: Motivation

- In Info I hatten wir kennengelernt, dass ein sortierter Binärbaum *b* mit *n* Knoten im ungünstigsten Fall zu einem Baum der Höhe *n* (entspricht einer verketteten Liste) entarten kann.
- **(Wiederholung) Definition Höhe eines Baums:** Die Höhe eines Baums ist das Maximum der "Höhen" aller seiner Blätter.
  - Die "Höhe" eines Blattes in einem Baum ist die Länge des Pfads von der Baumwurzel bis zu diesem Blatt.
  - Die Höhe eines leeren Baums sei -1.
  - Die Höhe eines Baums *b* bezeichnen wir mit *h(b)*.
- Komplexität von Bäumen: Der Aufwand für Einfügen, Suchen, Löschen ist beim sortierten Binärbaum abhängig von der Höhe des Baums:
  - nicht entarteter Baum: Aufwand *O(log n)*
  - entarteter Baum: Aufwand *O(n)*

Kennzeichen für entartete Bäume

- Das Kennzeichen für einen entarteten Baum: linker und rechter Unterbaum unterscheiden sich stark in ihrer Höhe.



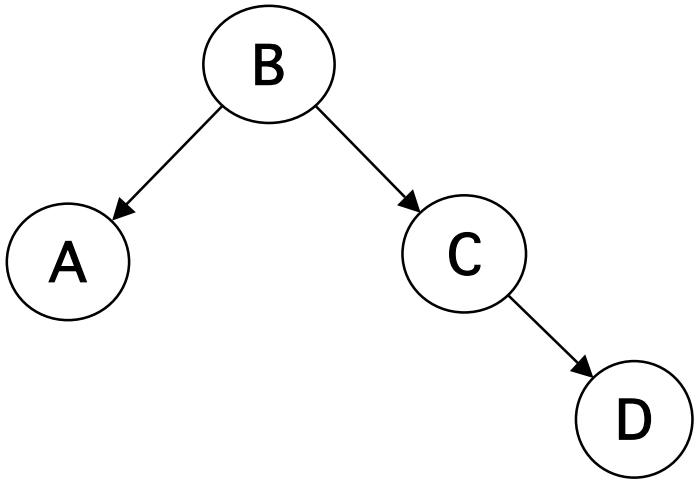
- Um die logarithmische Komplexität zu behalten, müssen wir also verhindern, dass ein Baum entartet.
- Idee: Ausgleichen der Höhen der Unterbäume ("Höhenbalancierung")

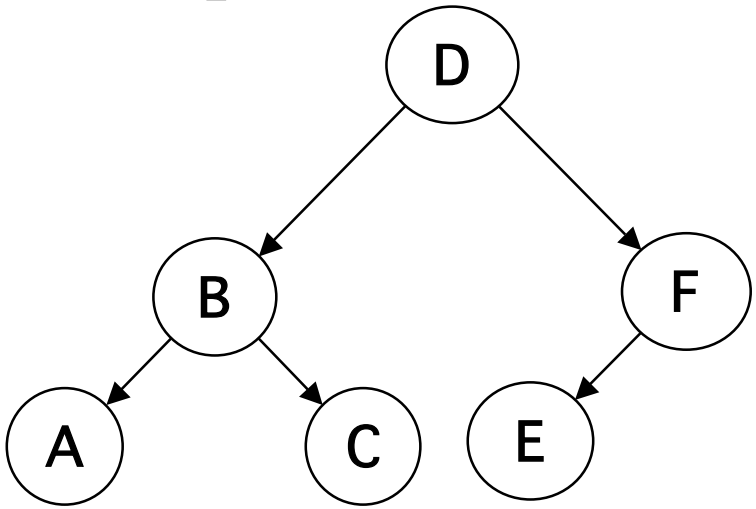
AVL-Bäume: Definition

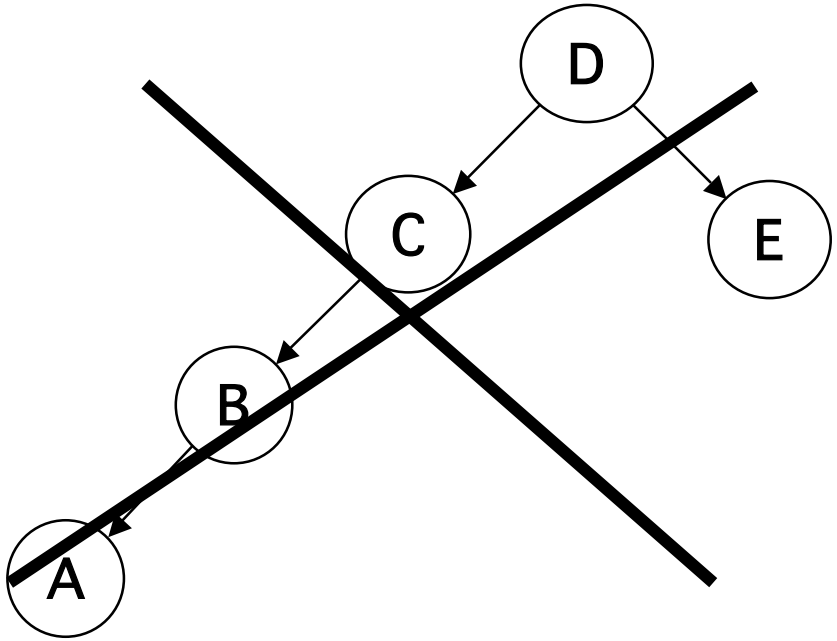
- **Definition AVL-Bedingung:** Sei *k* ein beliebiger Knoten in einem Binärbaum. Der Knoten *k* erfüllt die ***AVL-Bedingung***, wenn für seine beiden Unterbäume *k.l* und *k.r* gilt:
  - $| h(k.l) - h(k.r) | \leq 1$

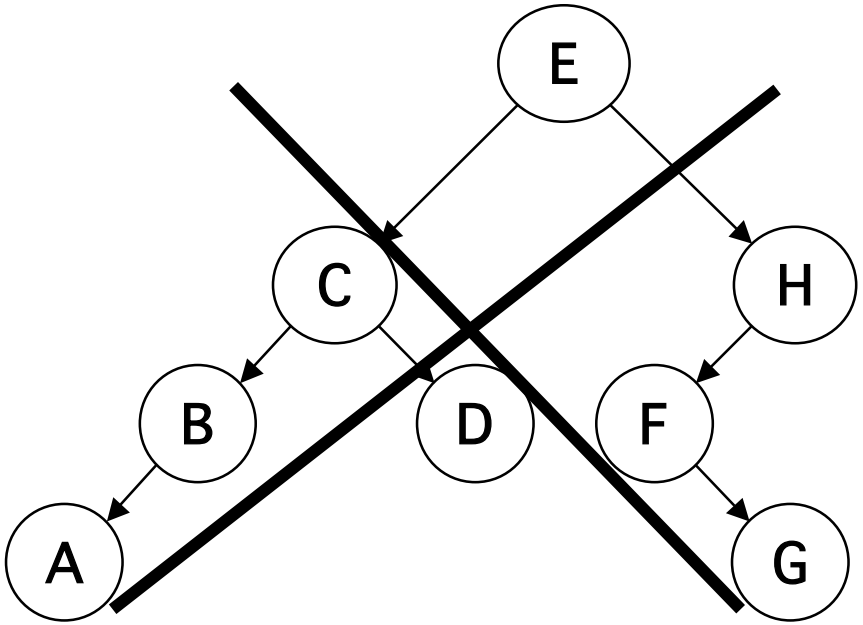
  Ein Knoten erfüllt die AVL-Bedingung also genau dann, wenn sich die Höhen seiner beiden Unterbäume höchstens um 1 unterscheiden.
- **Definition AVL-Baum:** Ein AVL-Baum *b* ist ein Binärbaum, für den gilt: ***Alle*** Knoten des Baums *b* erfüllen die AVL-Bedingung.
- Bemerkung: AVL-Bäume wurden 1962 von Adelson-Velski und Landis entdeckt.

Erkennung von AVL-Bäumen (Beispiele)







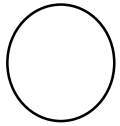


Eigenschaften von AVL-Bäumen

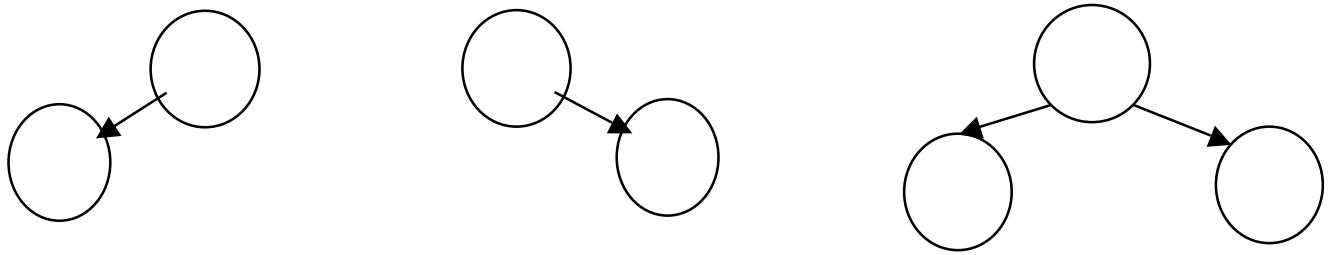
- Ein AVL-Baum der Höhe *h* hat mindestens F_{h+1} *-1* Knoten, wobei F_h die *h*-te Fibonacci Zahl ist.
  - Definition der Fibonacci-Zahlen: $F_0 = 0$, $F_1 = 1$, $F_{n+2} = F_{n+1} + F_n$
- Den Beweis führen wir durch vollständige Induktion

Ein AVL-Baum der Höhe h *hat mindestens* F_{h+1} -1 *Knoten (1)*

- Sei $n_{min}(h)$ die minimale Anzahl der Knoten im Baum mit der Höhe h
- **Induktionsanfang:**
  - Für $h = 0$ gibt es genau einen nicht-leeren AVL-Baum mit 1 Knoten, d.h. $n_{min}(0) = 1$:
    - $n_{min}(0) = 1 \geq F_1 - 1 = 0$, denn $F_1 = 1$



  - Für $h = 1$ gibt es genau drei verschiedene nicht-leere AVL-Bäume (mit zwei oder drei Knoten, d.h. $n_{min}(1) = 2$):
    - $n_{min}(1) = 2 \geq F_2 - 1 = 0$, denn $F_2 = 1$



  - Für $h = 0$ und $h = 1$ ist die Annahme somit korrekt.

Ein AVL-Baum der Höhe h *hat mindestens* F_{h+1} -1 *Knoten (2)*

- **Induktionsschritt h-2,h-1→ h:**
- Seien nun *k.l* und *k.r* die Kinder des Knotens *k*.
  Wir nehmen an, dass die Behauptung für *k.l* und *k.r* richtig ist, d.h. ein AVL-Baum der Höhe *h* - 2 hat mindestens F_{h-1} - 1 Knoten und ein AVL-Baum der Höhe *h* - 1 hat mindestens F_h - 1 Knoten .
- Es gilt: $h(k) = 1 + max(h(k.l), h(k.r))$ (Definition der Höhe eines Baums)
- *k* erfüllt die AVL-Bedingung ⇒ 3 Fälle:
  $h(k.l) = h(k.r)$, $h(k.l) = h(k.r) + 1$ und $h(k.l) = h(k.r) - 1$
- Im schlechtesten Fall hat ein Unterbaum die Höhe *h-2* und einer die Höhe *h-1:*

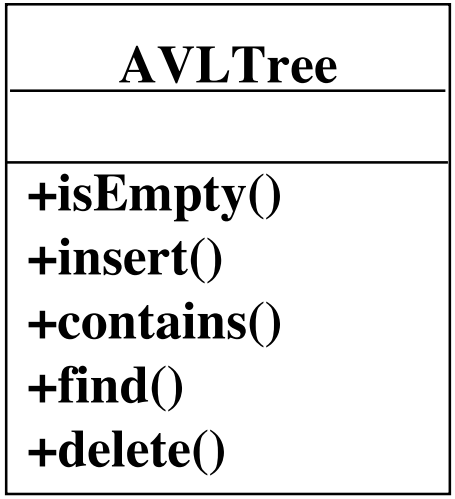
$$
\begin{aligned}
n_{min}(h) &= 1 + n_{min}(h-2) + n_{min}(h-1) && \text{-- Minimale Anzahl von Baum der Höhe } h \\
&\geq 1 + F_{h-1} - 1 + F_h - 1 && \text{-- Induktionsannahme} \\
&= F_{h-1} + F_h - 1 && \text{-- Umformung} \\
&= F_{h+1} - 1 && \text{-- Def inition Fibonacci: } F_{n+2} = F_{n+1} + F_n
\end{aligned}
$$

Modellierung von AVL-Bäumen

- Wir modellieren die Klasse **`AVLTree`** mit derselben Schnittstelle wie die sortierten Binärbäume, d.h. wir können die Modellierung der Klasse **`Tree`** aus Info 1 - Vorlesung 9 wieder verwenden.
- Wir ergänzen noch eine Operation

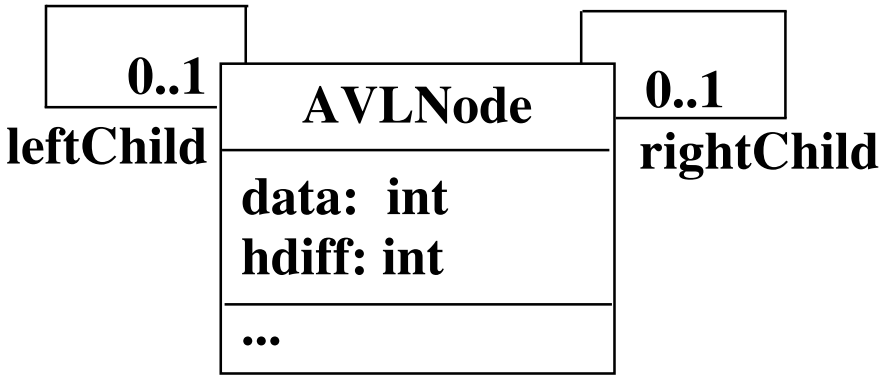
  `contains (key: int): boolean`

  Die Operation **`contains()`** liefert **`true`**, wenn mindestens ein Knoten im Baum den Wert **`key`** hat.



Modellierung von AVL-Baumknoten

- Um jederzeit über die Höhendifferenzen der Knoten Bescheid zu wissen, definieren wir außerdem ein zusätzliches Attribut **`hdiff`** in der Knoten-Klasse, die wir **`AVLNode`** nennen:



- Damit die AVL-Bedingung eingehalten ist, darf **`hdiff`** nur die Werte -1, 0 oder 1 annehmen.

Spezifikations-Operationen (1)

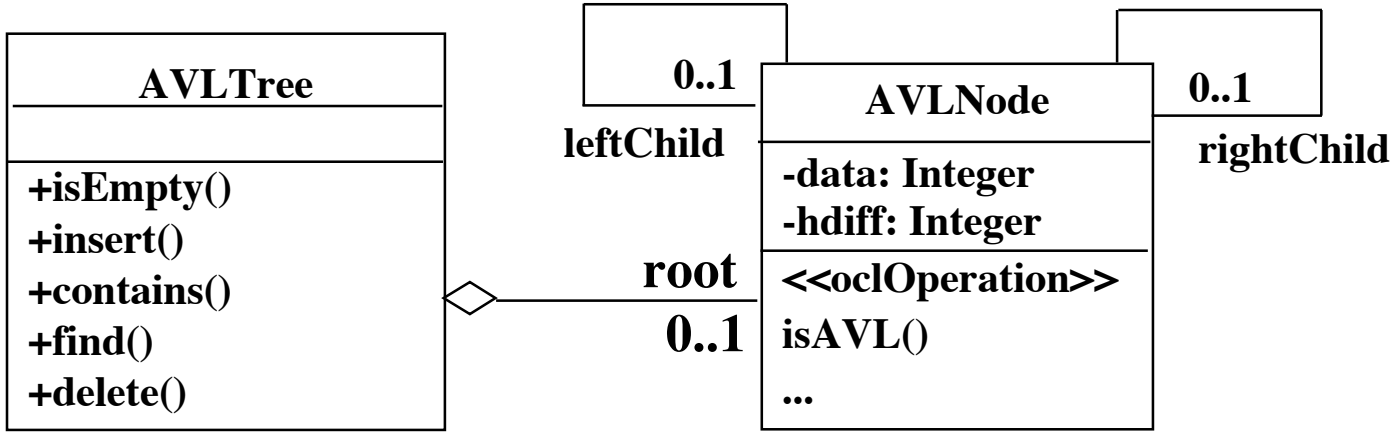
- Manchmal ist es sinnvoll, für eine Klasse Attribute oder Operationen zu definieren, die wir nur zur Spezifikation der Verträge brauchen. UML definiert dafür einen Stereotyp **`<<oclOperation>>`**.
- Wir können beispielsweise die AVL-Bedingung so spezifizieren:
- **Spezifikation AVL-Bedingung:** Sei *k* ein beliebiger Knoten im AVL-Baum.
  - Für *k* gilt: **`abs(`***k*`**`**`.hdiff) <= 1`**
  - Wenn *k* einen linken Kindknoten *l* hat, dann gilt die AVL-Bedingung auch für *l*.
  - Wenn *k* einen rechten Kindknoten *r* hat, dann gilt die AVL-Bedingung auch für *r*.
- Diese rekursive Definition brauchen wir bei der Implementation von AVL-Bäumen nicht, bei der Spezifikation von Vor- und Nachbedingungen von AVL-Baumoperationen ist sie aber nützlich.
- Wir deklarieren sie deshalb in OCL als Spezifikations-Operation.

Spezifikations-Operationen (2)

- **Definition Spezifikations-Operation:** Eine Operation, die mit dem Stereotyp **`<<oclOperation>>`** als Präfix versehen ist.
- Für Spezifikations-Operationen gilt:
  - Die Operation kann im UML-Modell sichtbar gemacht werden.
  - Die Operation muss in keiner Implementation des Modells erscheinen. Sie wird nur für Spezifikationszwecke benutzt.
  - Spezifikations-Operationen kann man z.B. sehr gut beim Überprüfen von Invarianten, Vor- und Nachbedingungen einsetzen.
- Bemerkung:
  - Bei der Konstruktion von Programmen können Spezifikations-Operationen oft so wichtig werden, dass man sie auch bei der Implementation sinnvoll einsetzen kann.

    $\Rightarrow$ Laufzeitüberprüfung von Vor-und Nachbedingungen

Modellierung der AVL-Bedingung durch Spezifikations-Operation `isAVL`



OCL-Modell:

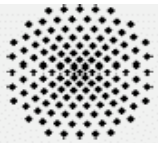
```
AVLNode::isAVL(): Boolean
post: result =
        (abs(hdiff) <= 1) and
        (leftChild->notEmpty implies leftChild.isAVL()) and
        (rightChild->notEmpty implies rightChild.isAVL())
```

Optionale Attribute (mit Multiplizität `0..1`) werden in OCL als Menge behandelt. Damit kann einfach überprüft werden, ob das Attribut vorhanden ist oder nicht.

Ferienakademie

- Letzter Bewerbungstermin: 28. Mai 2004, also morgen.





Universität Erlangen-Nürnberg - Technische Universität München - Universität Stuttgart

Ferien-Akademie

Sarntal (Südtirol) - Sonntag, 19. September bis Freitag, 1. Oktober 2004

Kurs 1:
Polynome – Effiziente Algorithmen und Anwendungen

Zielgruppe

Studierende im Grundstudium der Fachrichtung Informatik mit Interesse an mathematischen und algorithmischen Fragestellungen

Dozenten

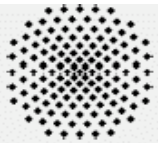
- Prof. Dr. Hans-Joachim Bungartz (Stuttgart)
- Prof. Dr. Ernst W. Mayr (TU München)

Jeder Teilnehmer hält einen Vortrag zum oben genannten Thema. Von den Teilnehmern wird dabei bei der Vorbereitung und Durchführung der Kurse eine aktive Beteiligung erwartet. Die **Auswahl** unter den Bewerbern erfolgt **aufgrund** ihrer **Qualifikation**. Sie wird vom Direktor der Ferienakademie und den Beauftragten der beteiligten Universitäten in Abstimmung mit den Dozenten der jeweiligen Kurse vorgenommen. **Fahrt- und Aufenthaltskosten für die Teilnehmer werden aus Spendenmitteln getragen.**

Abgabe der Bewerbungsunterlagen bis 28.Mai 2004

Bewerbungsformulare gibt es im Internet unter **www5.in.tum.de/FA** oder bei allen **Dozenten** der Ferienakademie. Weitere Fragen zu Kurs und Sonstigem an **kosub@in.tum.de.**





Universität Erlangen-Nürnberg - Technische Universität München - Universität Stuttgart

Ferien-Akademie

Sarntal (Südtirol) - Sonntag, 19. September bis Freitag, 1. Oktober 2004

Kurs 2:
Context-Awareness

Zielgruppe

Studierende im Hauptstudium. Studierende im Grundstudium mit ausgezeichneten Qualifikationen können sich ebenfalls bewerben.

Dozenten

Prof. B. Brügge, Ph.D. (Technische Universität München) • **Prof. Dr. H. Niemann** (Universität Erlangen-Nürnberg)
Prof. Dr. Dr. h.c. K. Rothermel (Universität Stuttgart)• **Prof. A. Smailagic, Ph.D.** (Carnegie Mellon University)
Stefan Holtel, Vodafone-Pilotentwicklung

In diesem Kurs wird ein Prototyp für ein visionäres System aus dem Bereich eHealth konstruiert, wobei die eingesetzten Techniken von Brainstorming bis zum Einsatz von digitalen Filmtechniken gehen. Von den Teilnehmern wird dabei bei der Vorbereitung und Durchführung der Projektes eine aktive Beteiligung erwartet.

siehe auch: http://www5.in.tum.de/FA/_2004/K2/kursinfo_K2.html

Abgabe der Bewerbungsunterlagen bis 28.Mai 2004
Bewerbungsformulare gibt es im Internet unter **www5.in.tum.de/FA** oder bei allen **Dozenten** der Ferienakademie. Weitere Fragen zum Kurs an **bruegge@in.tum.de.**

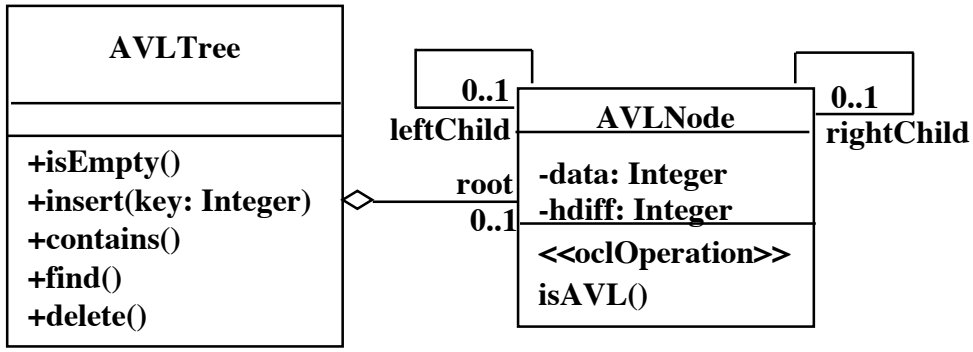
Spezifikation der Operationen für AVL-Bäume

- **Idee:** Wiederverwendung der Methoden **`insert()`**, **`find()`** und **`delete()`** für sortierte Binärbäume aus Info I - Vorlesung 9.
- **Frage:** Welche dieser Methoden können die AVL-Bedingung verletzen?
- Durchsuchen ändert den Zustand des durchsuchten Baums nicht

  ⇒ **`find()`** kann ohne Änderungen wieder verwendet werden
- Einfügen und Löschen ändern den Zustand des Baums

  ⇒ **`insert()`** und **`delete()`** müssen so angepasst werden, dass sie die AVL-Bedingung einhalten
- Im folgenden betrachten wir die Spezifikation und die sich daraus ergebenden Konsequenzen für die Implementierung für die **`insert()`**-Operation.
- Für **`delete()`** verläuft die Argumentation im wesentlichen analog.

OCL Spezifikation der `insert()`-Operation



- Zum Einfügen von Elementen in AVL-Bäumen verwenden wir die Methode **`insert()`** aus Info I - Vorlesung 9 zum Einfügen von Elementen in sortierten Binärbäumen.
- Nachbedingung zur Spezifikation von **`insert()`**:

  `AVLTree::insert(key: Integer)`

  `post: contains(key) = true`
- Vorbedingung: mehrfaches Einfügen von Elementen verhindern.

  `AVLTree::insert(key: Integer)`

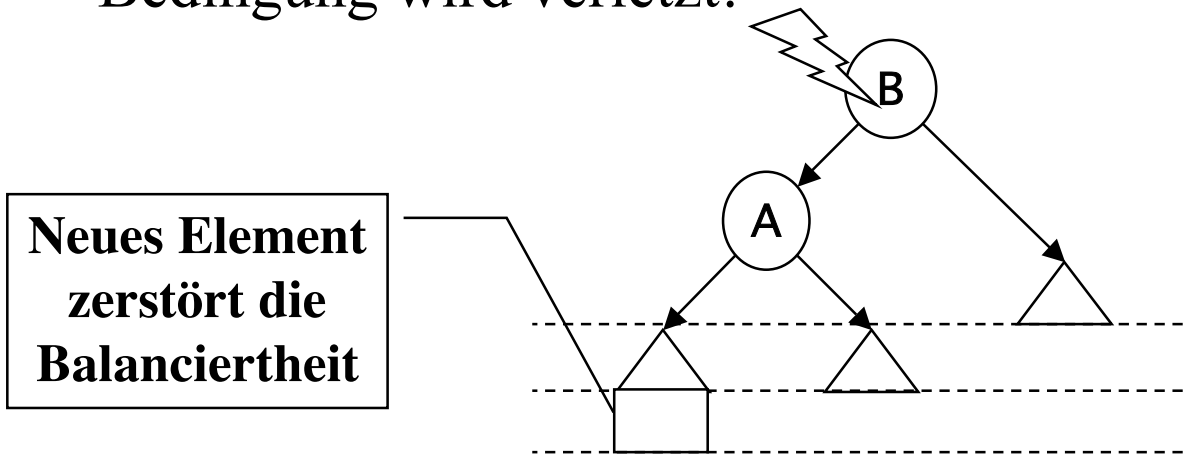
  `pre: contains(key) = false`
- Weitere Nachbedingung: Alle Knoten des Baums erfüllen auch nach dem Einfügen eines Elements die AVL-Bedingung:

  `AVLTree::insert(key: Integer)`

  `post: root.isAVL() = true`

Ein AVL-Baum kann durch `insert()` unbalanciert werden

- Während wir **`insert()`** ausführen, kann direkt nach dem Einfügen des Elements der Baum unbalanciert werden, d.h. die AVL-Bedingung wird verletzt!



- In diesem Fall müssen wir den Baum vor Ende der Ausführung von **`insert()`** wieder balancieren.
- Zur Balancierung des AVL-Baums führen wir eine oder mehrere sogenannte **Rotationen** durch.
  - **Ziel:** Ausgleich der Höhendifferenz durch Änderung der Baumstruktur bei Erhaltung der Sortiertheit

Arten von Rotationen

- **Einfache Rotation**
  - Einfache Rotationen verwendet man, nachdem ein neues Element *außen* an einen AVL-Baum angehängt wurde, und der Baum dadurch unbalanciert wird. Wir unterscheiden 2 Fälle:

    1. Der Knoten wird links außen angehängt: Rechtsrotation

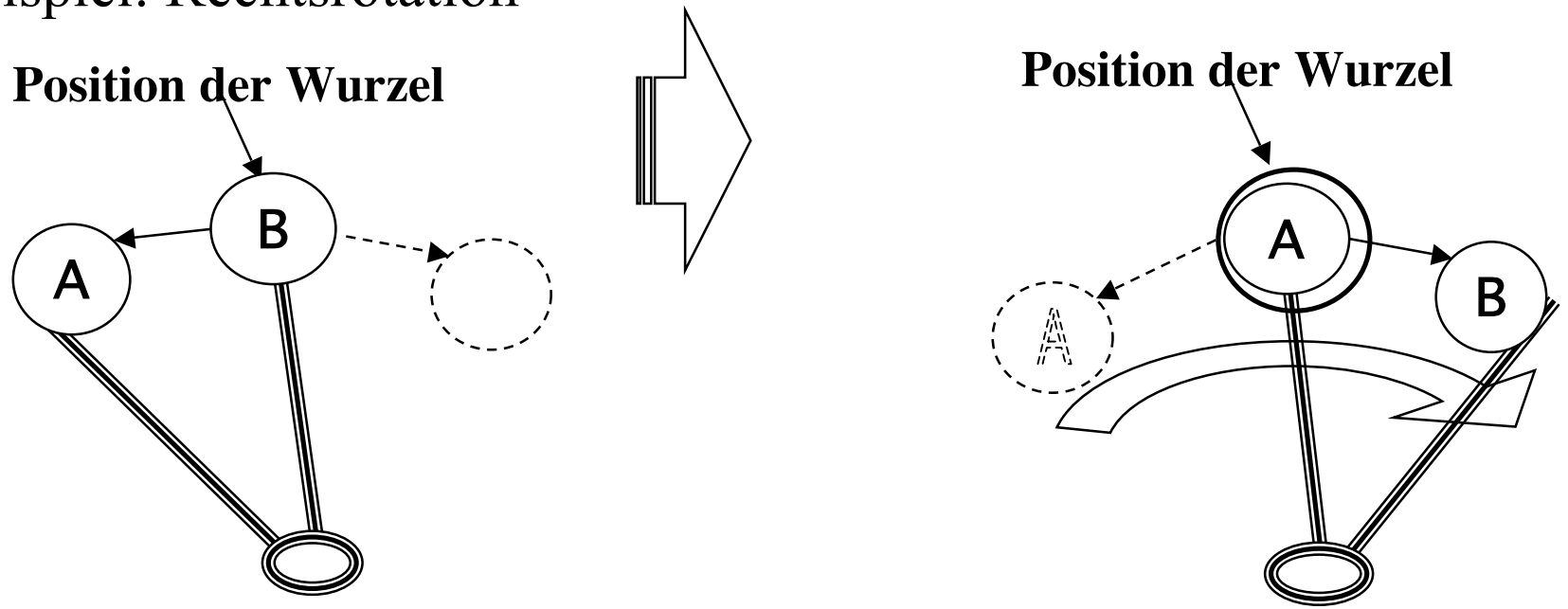
    2. Der Knoten wird rechts außen angehängt: Linksrotation
- **Doppelrotation**:
  - Doppelrotationen benötigt man, nachdem ein neues Element *in der Mitte* an einen AVL-Baum angehängt wurde, und der Baum dadurch unbalanciert wird. Wir unterscheiden 2 Fälle:

    1. Der Knoten wird im rechten Unterbaum eingehängt:
    Wir machen erst eine Linksrotation, dann eine Rechtsrotation

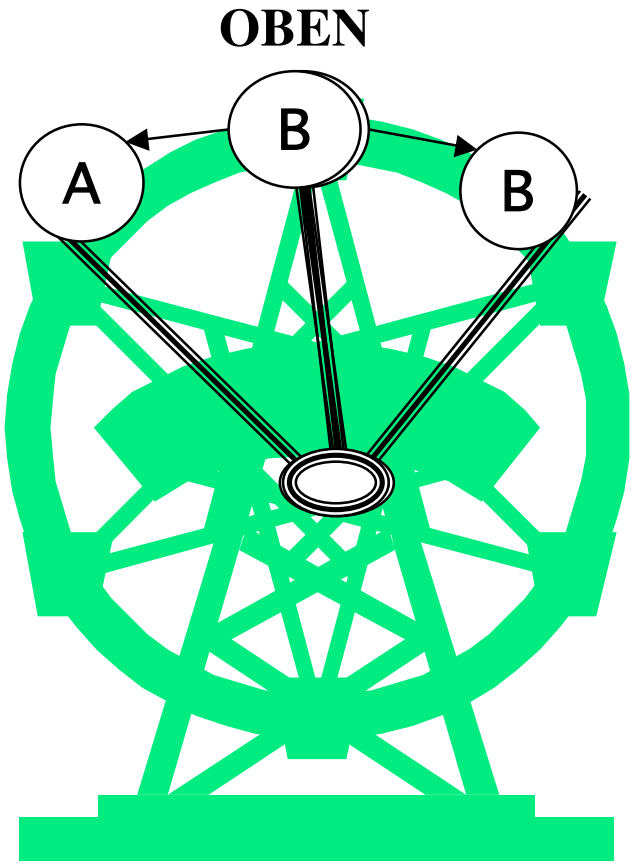
    2. Der Knoten wird im linken Unterbaum eingehängt:
    Wir machen erst eine Rechtsrotation, dann eine Linksrotation

Was bedeutet Rotation?

- **Idee hinter dem Begriff:** Stellen Sie sich vor, die Knoten eines 3-elementigen Baums sind auf den Speichen eines Rades angeordnet. Das Rad wird dann um eine Position nach links oder rechts "rotiert". Die Position der Wurzel ist dabei fest.
- Beispiel: Rechtsrotation

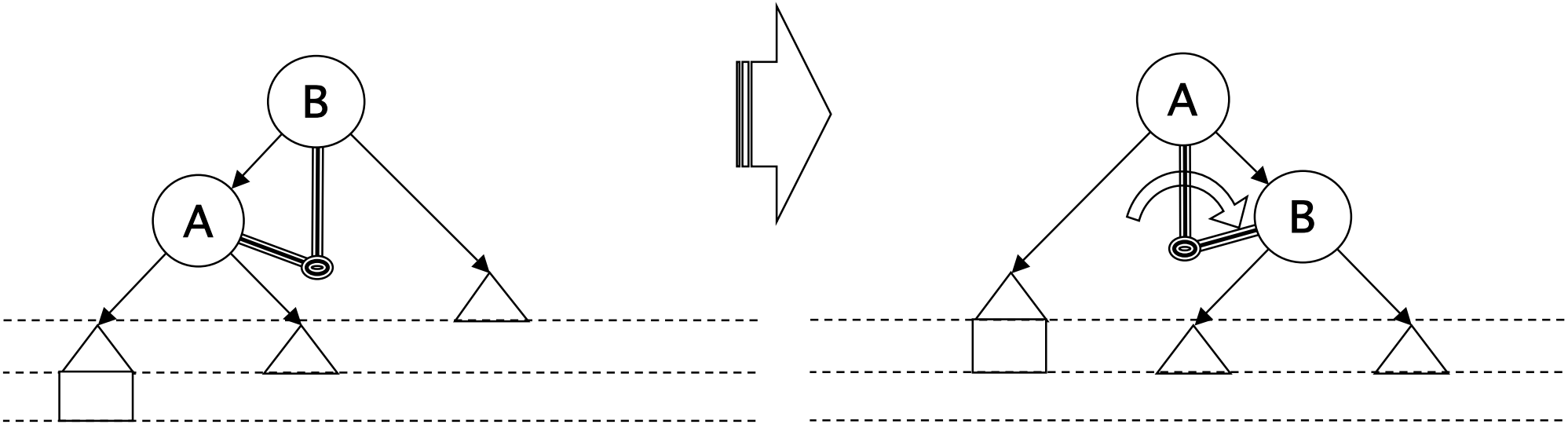


Rechtsrotation



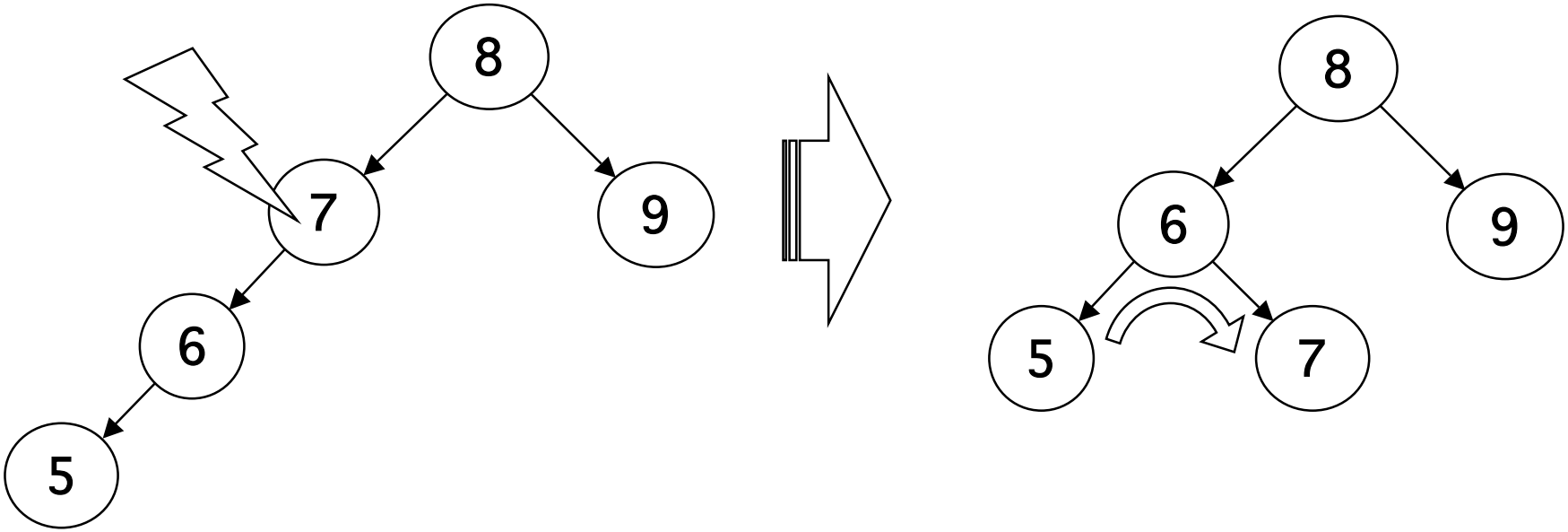
Balancierung von AVL-Bäumen: Einfache Rotationen

- Einfache Rotationen verwendet man, nachdem ein neues Element *außen* an einen AVL-Baum abgehängt wurde, und der Baum dadurch unbalanciert wird.
- Beispiel: Rechtsrotation



Beispiel: Einfache Rotation in AVL-Bäumen

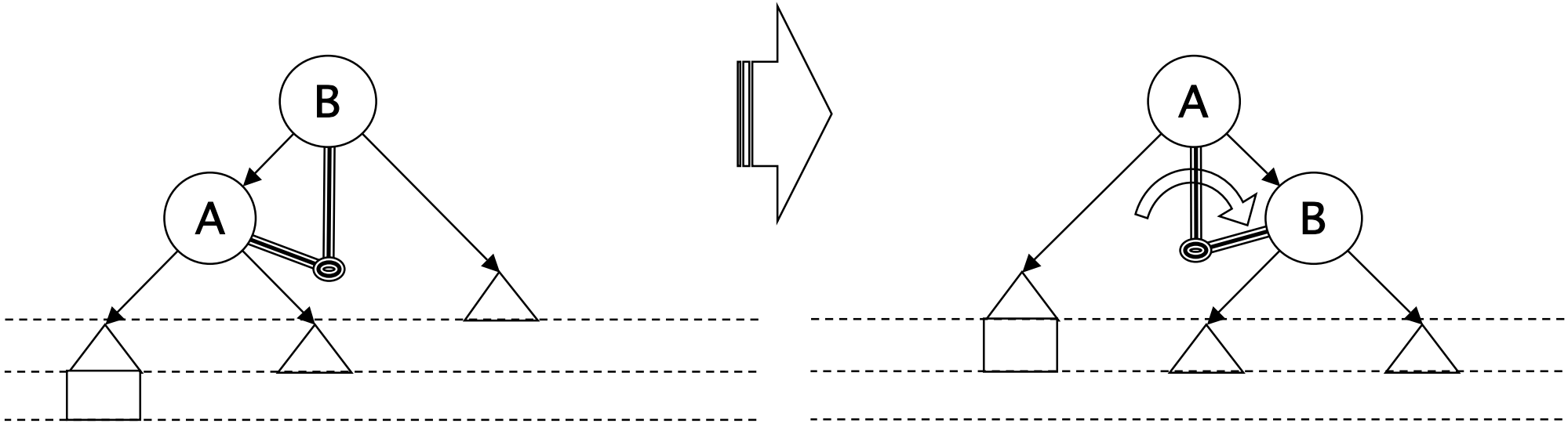
- Einfügen von Knoten **5**
  - AVL-Bedingung ist verletzt für Knoten **7.** Warum?
  - h(**6**) - h(**7**.r) = 2 (siehe AVL-Bedingung auf Folie 19)



- Zur Wiederherstellung der Einhaltung der AVL-Bedingung genügt eine Rechtsrotation.
- Leichte Komplikation, wenn Knoten **6** vor der Rotation einen rechten Unterknoten hat

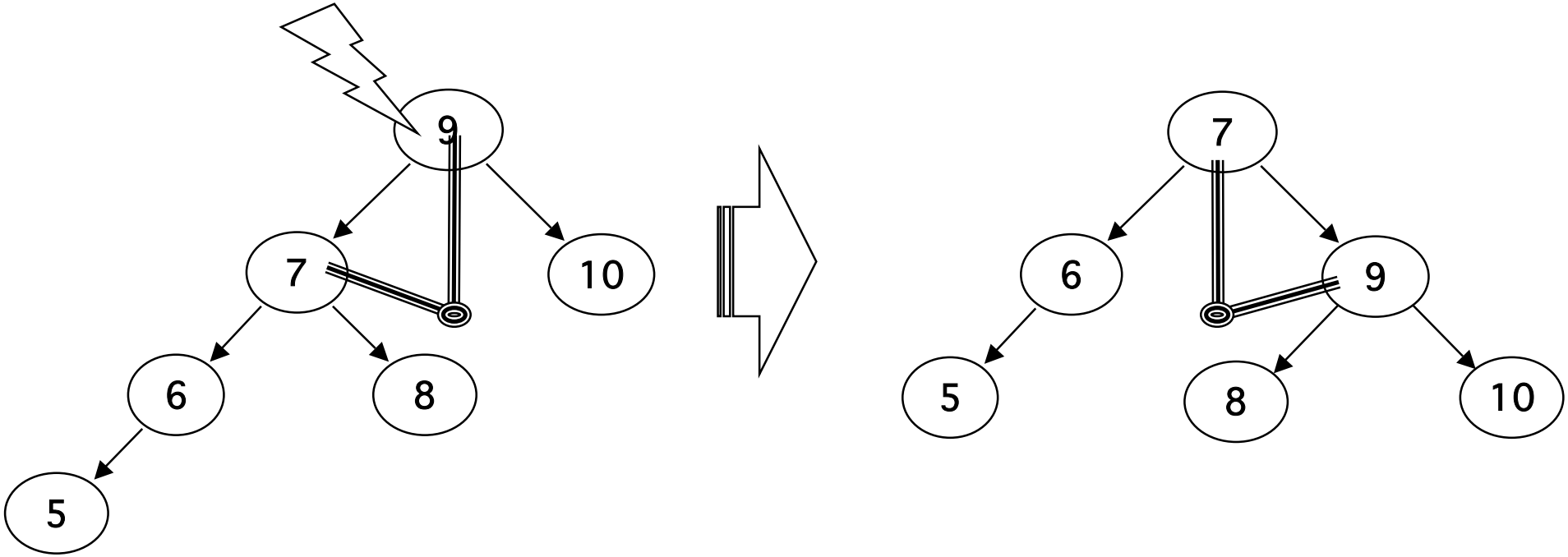
Überkreuz-Zug (cross-over move)

- Wenn **A** auch einen rechten Unterknoten hat, ist ein sogenannter Überkreuz-Zug notwendig:
  - Bei der einfachen Rotation wird der rechte Unterknoten von **A** (der ja größer ist als **A**) als linker Unterknoten von **B** eingehängt (denn er ist kleiner als **B**):



Beispiel: Überkreuz-Zug in AVL-Bäumen

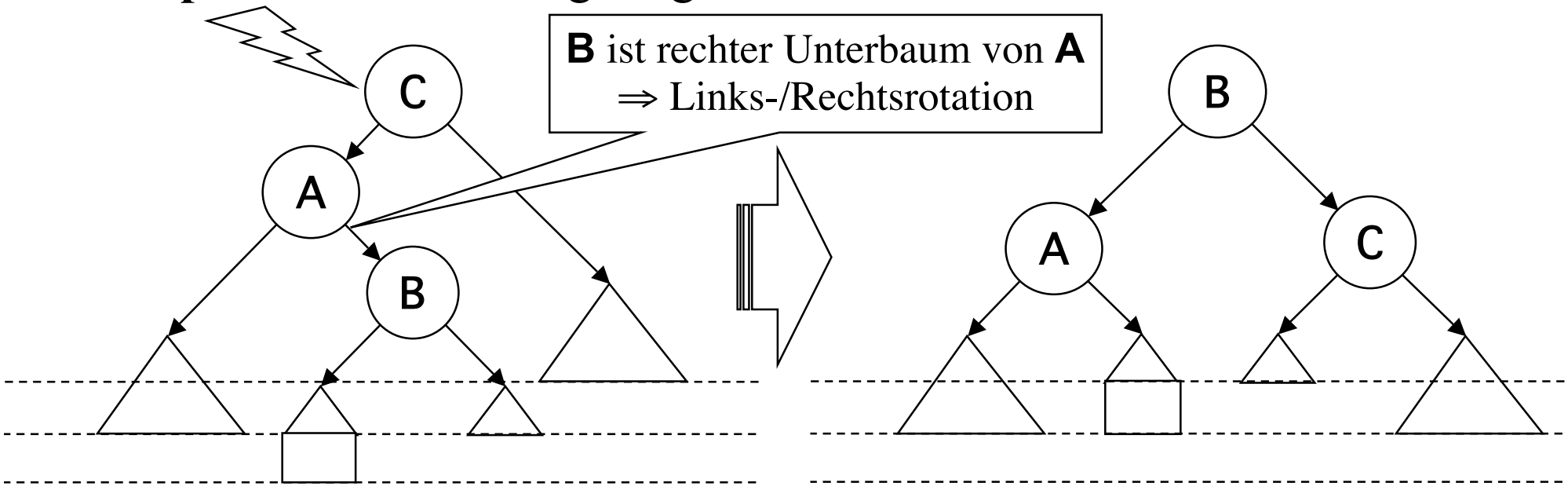
- Einfügen von Knoten **5**
  - AVL-Bedingung verletzt für Knoten **9.** Warum?
  - h(**7**) - h(**10**) = 2



- Nach dem Überkreuz-Zug über Knoten **9** ist die AVL-Bedingung wieder erfüllt.

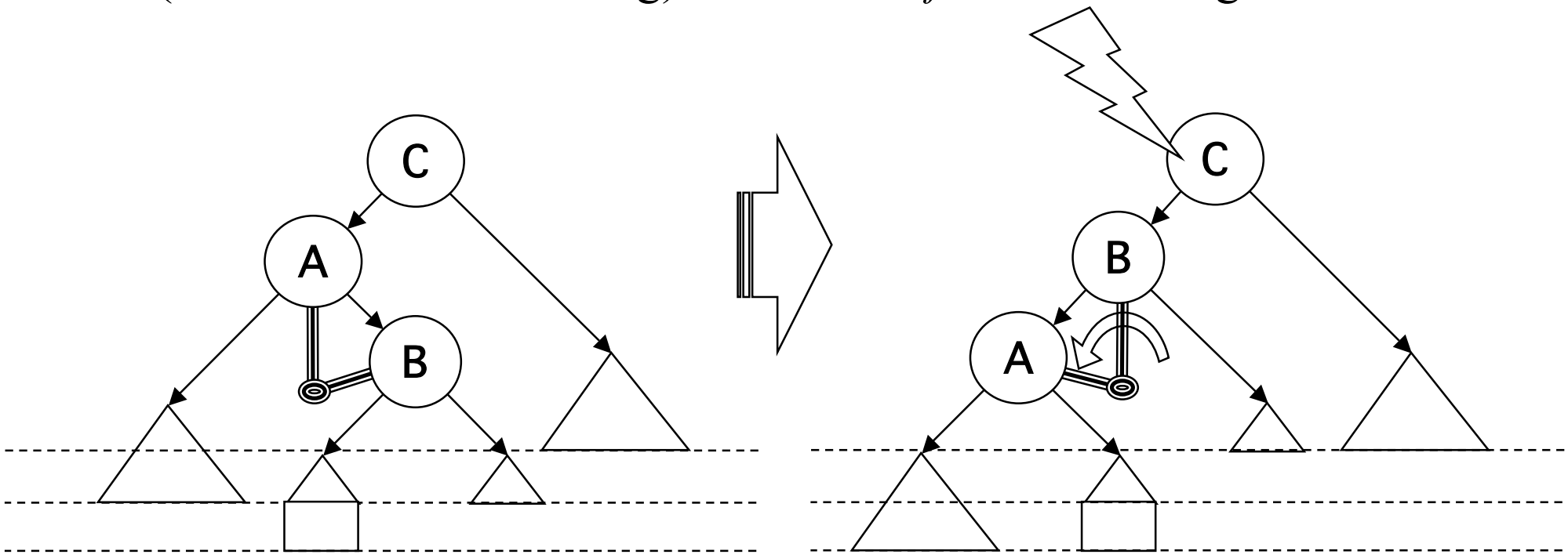
Balancierung von AVL-Bäumen: Doppelrotation

- Doppelrotationen benötigt man, wenn ein neues Element *in die Mitte* eines AVL-Baum gehängt wird und der Baum die Balance verliert.
- Eine Doppelrotation besteht aus 2 Einzelrotationen:
  - Der Knoten wird in einem rechten Unterbaum eingehängt
    ⇒ Linksrotation, gefolgt von Rechtsrotation
  - Der Knoten wird in einem linken Unterbaum eingehängt
    ⇒ Rechtsrotation, gefolgt von Linksrotation
- **Beispiel**: Linksrotation gefolgt von Rechtsrotation



Die Doppelrotation im Detail (1)

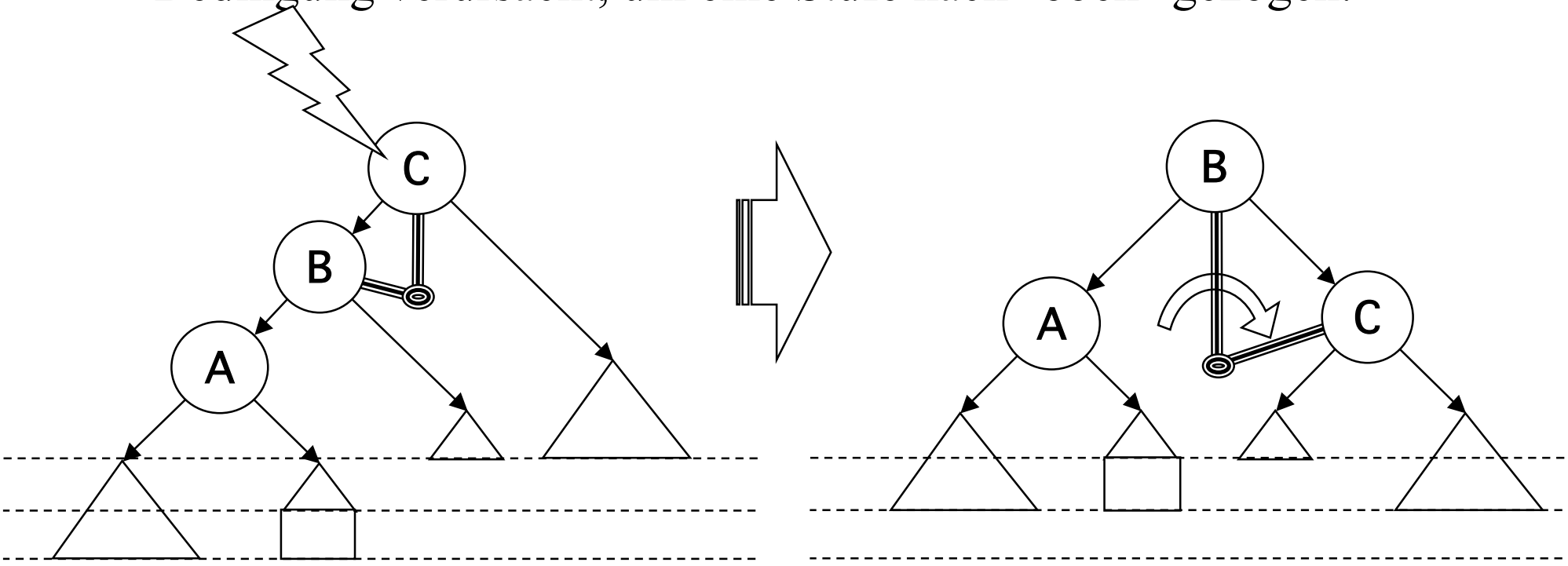
- Während der doppelten Rotation wird zunächst die Wurzel des Unterbaums, der die AVL-Bedingung verletzt, durch **Linksrotation** (evtl. mit Überkreuz-Zug) *zu einem äußeren Knoten* gemacht:



- Die Verletzung der AVL-Bedingung ist damit noch nicht beseitigt! Deshalb benötigen wir den zweiten Rotationsschritt.

Die Doppelrotation im Detail (2)

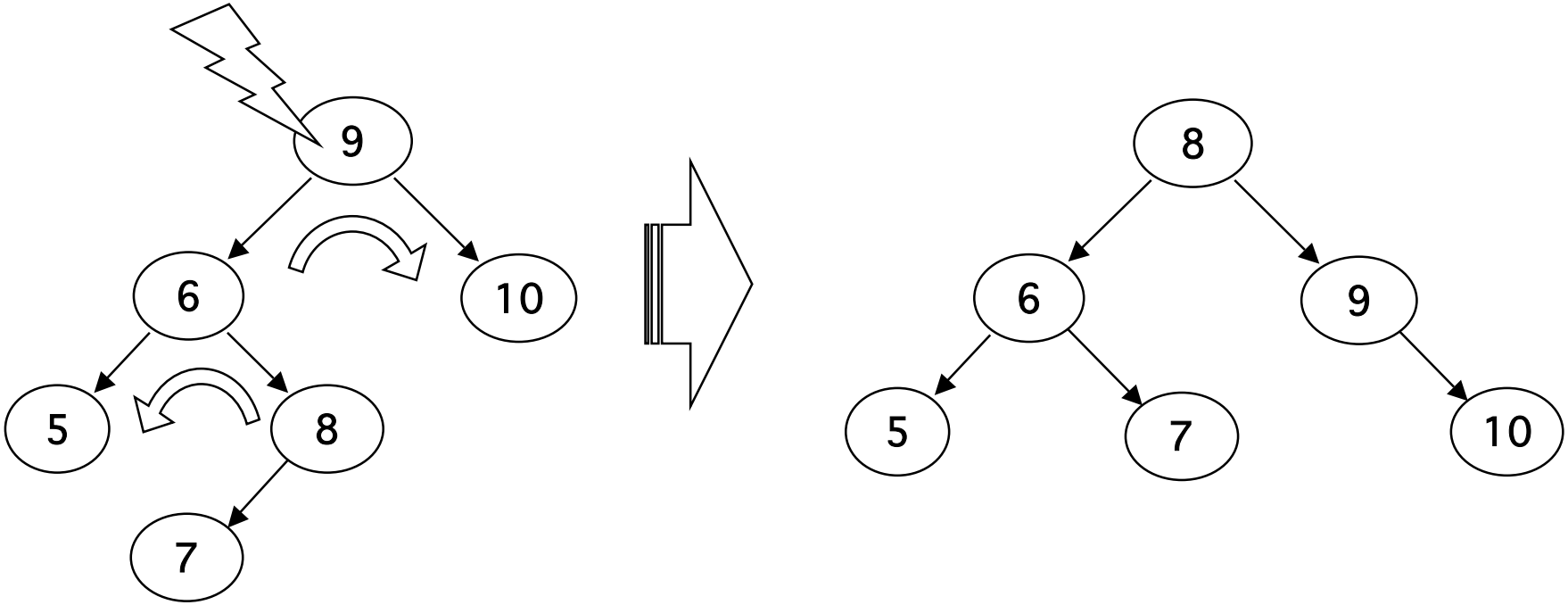
- Durch eine unmittelbar **anschließende Rechtsrotation** (evtl. mit Überkreuz-Zug) wird der Teilbaum, der die Verletzung der AVL-Bedingung verursacht, um eine Stufe nach "oben" gezogen:



- Danach ist die AVL-Bedingung wieder erfüllt.

Beispiel: Doppelrotation in AVL-Bäumen

- Einfügen von Knoten **7**
  - AVL-Bedingung verletzt für Knoten **9.** Warum?
  - h(**6**) - h(**10**) = 2



- Nach der Doppelrotation (Links-/Rechtsrotation) über die Knoten **6** und **9** ist die AVL-Bedingung wieder erfüllt.

Algorithmus für die Implementation von `insert()`

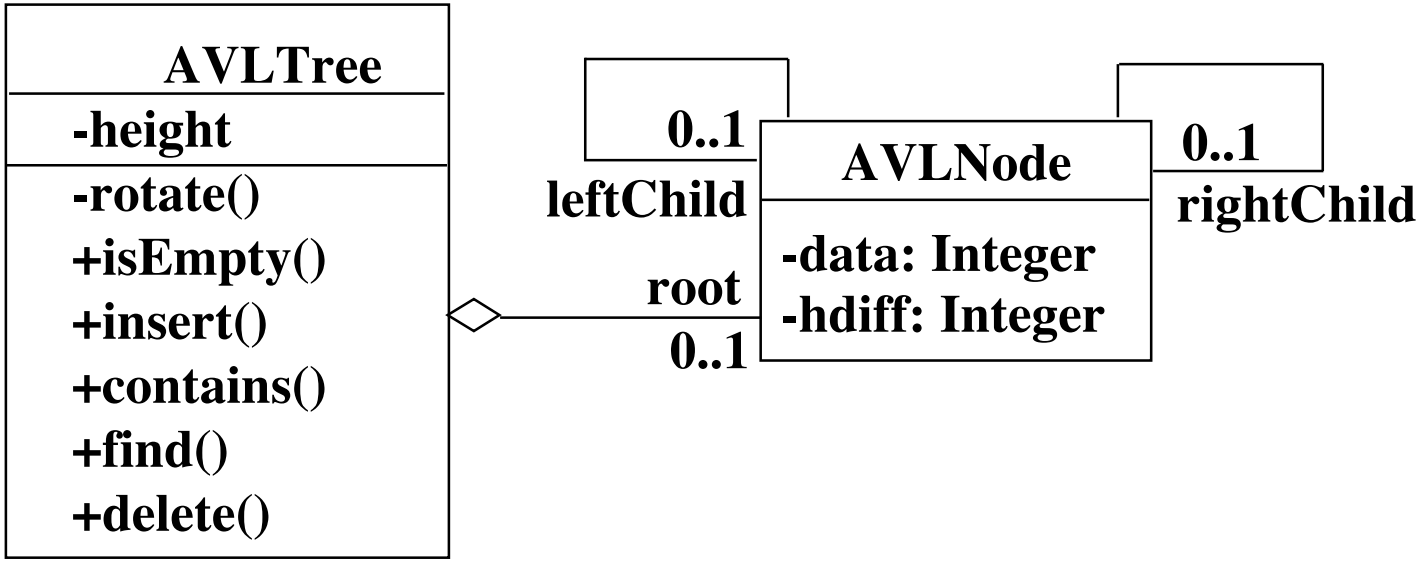
1. Zunächst fügen wir den neuen Knoten *x* ein (wie beim binären Suchbaum)
2. Von der Einfügeposition gehen wir den Pfad zurück *in Richtung Wurzel*
3. Für jeden Knoten *k,* den wir auf diesem Pfad erreichen, vergleichen wir den aktualisierten Wert von **`hdiff`** mit dem alten Wert (vor dem Einfügen des Knotens *x*). Dabei unterscheiden wir 3 Fälle:
   - **`(abs(hdiff@pre) = 1) and (hdiff = 0)`**:
     k.l und *k.r* wurden durch das Einfügen auf gleiche Höhe gebracht, d.h. *h(k)* hat sich nicht geändert:
     - fertig mit **`insert()`**
   - **`(hdiff@pre = 0) and (abs(hdiff) = 1)`**:
     h(k.l) und *h(k.r)* unterscheiden sich nach dem Einfügen um 1, d.h. auch *h(k)* hat sich um 1 erhöht; **`hdiff`** muss also auch für den Vater von *k* aktualisiert werden:
     - Überprüfung mit dem Vaterknoten fortsetzen
   - **`abs(hdiff@pre) = 1) and (abs(hdiff) = 2)`**:
     Knoten *k* wurde durch das Einfügen aus der Balance gebracht:
     - Rotation zur Ausbalancierung durchführen
     - fertig mit **`insert()`**

Algorithmus für die Implementation von `delete()`

- Analog zu **`insert()`**
  1. Löschen des Knotens (wie beim sortierten Binärbaum)
  2. Ausbalancieren des Baums
- **Wichtig:** Anders als beim Einfügen können beim Löschen mehrere Rotationen zum Ausbalancieren des Baums erforderlich sein (maximal eine für jeden Knoten auf dem Pfad vom gelöschten Knoten zur Wurzel des Baums).

Modell eines AVL-Baumes (Revidierte Version)

UML-Modell:



Zusätzliche Vor-/Nachbedingungen:

```
AVLTree::rotate(node: AVLNode)
pre:  abs(node.hdiff) > 1
post: abs(node.hdiff) <= 1
```

Komplexität von Operationen auf AVL-Bäumen (1)

- Für die maximale Anzahl von Knoten n gilt folgende Ungleichung:

  $2^{h+1} - 1 \geq n \geq F_{h+1} - 1$

  wobei $F_{h+1} = 1/\sqrt{5} \cdot [(1/2 \cdot (1 + \sqrt{5}))^{h+1} - (1/2 \cdot (1 - \sqrt{5}))^{h+1}]$
  (siehe Goos II, S. 248)
- Die obere Schranke gilt nach der Definition von Binärbäumen der Höhe h.
- Die untere Schranke haben wir in dieser Vorlesung bewiesen.
- Addieren wir 1 und logarithmieren die Ungleichung $n + 1 \geq F_{h+1}$, dann erhalten wir:

  $c \cdot \log_2 n + 1 \geq h \qquad (c < 2)$
- Das bedeutet, dass die Höhe eines AVL-Baums von der Wurzel bis zu einem beliebigen Knoten höchstens $2 \cdot \log_2 n + 1$ ist.
  - Die Suchoperationen **`find()`** und **`contains()`** haben deshalb im schlechtesten Fall die Komplexität *O(log n)*.

Komplexität von Operationen auf AVL-Bäumen (2)

- Komplexität für Einfügen und Löschen in sortierten Binärbäumen: *O(h)*
  - im AVL-Baum gilt: *h = O(log n)*
  - Notwendige Rotationen sind in konstanter Zeit *O(1)* durchführbar.
- Ausbalancieren nach Einfügen erfordert höchstens eine (einfache oder doppelte) Rotation

  ⇒ Komplexität von **`insert()`**: *O(log n) + O(1) = O(log n)*
- Ausbalancieren nach Löschen erfordert höchstens *h* (einfache oder doppelte) Rotationen

  ⇒ Komplexität von **`delete()`**: *O(log n) + O(log n) = O(log n)*
- Einfügen und Löschen in AVL-Bäumen haben also auch im schlechtesten Fall eine Komplexität von *O(log n)*.

Zusammenfassung

- Prädikatenlogik ist eine Verallgemeinerung der Aussagenlogik
  - Variablen
  - Quantoren
- Allgemeingültigkeit und Erfüllbarkeit von Prädikaten
- Termersetzungsregeln für Prädikate
- Quantoren in OCL
- Anwendungsklassen vs. Lösungsklassen
- AVL-Bäume
- Rotation in AVL-Bäumen
  - Einfache Rotation
  - Doppelrotation
- Komplexität der AVL-Baum-Operationen: *O(log n)*