Universität Bremen

Praktische Informatik 2
Bäume (Forts.)

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

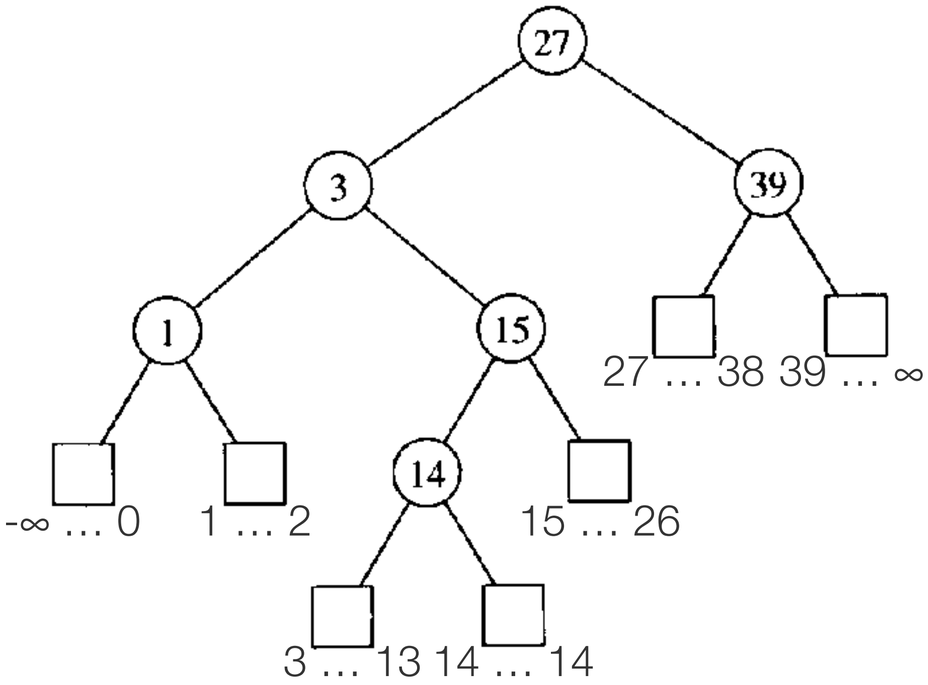
Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Universität Bremen

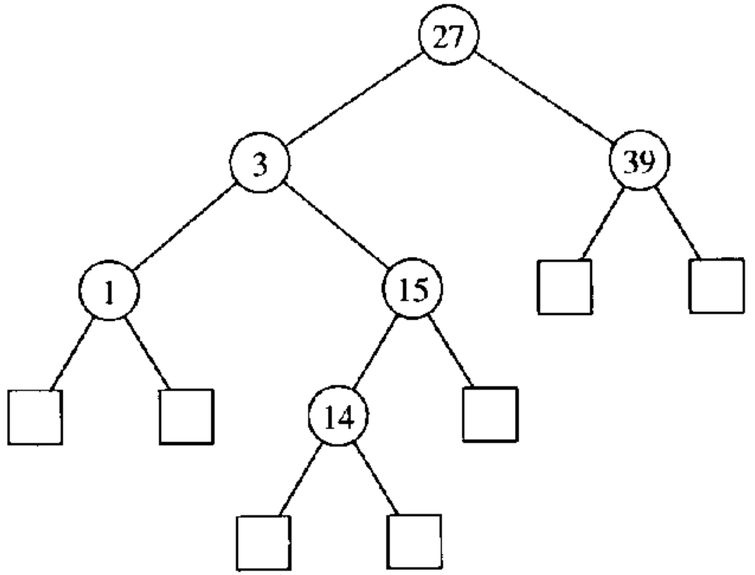
Suchbäume

- Dienen zum schnellen Auffinden von Werten, möglichst in **O(log n)**
- Die Schlüssel aller linken Nachfolger sind kleiner als der Schlüssel eines Knotens, die Schlüssel aller rechten sind größer (oder gleich)
- Ein **natürlicher Baum** wird durch zufälliges Einfügen erzeugt (nicht unbedingt **balanciert**)
- Alternative Schreibweise:
  (((□ 1 □) 3 ((□ 14 □) 15 □)) 27 (□ 39 □))
- Sichtweise: Blätter repräsentieren Intervalle im Wertebereich der Schlüssel



Universität Bremen

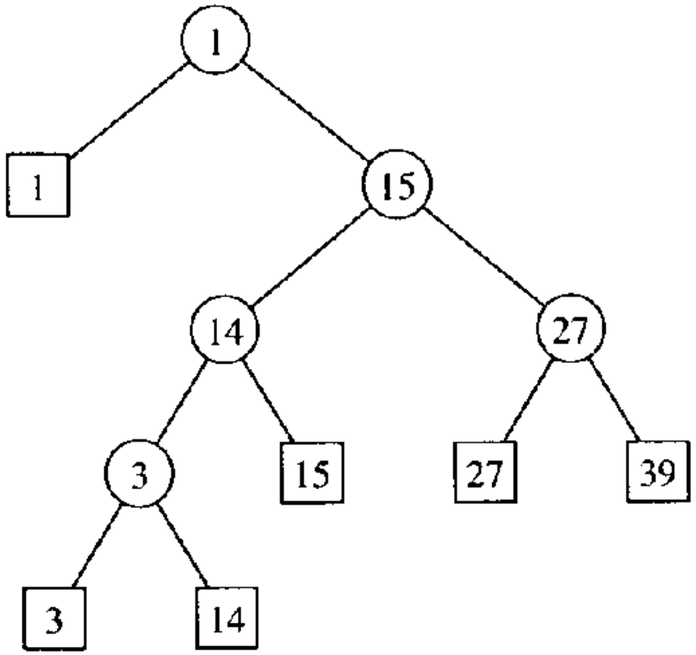
Normaler Suchbaum



- Werte stehen in inneren Knoten und die Blätter sind leer
- **Suchen, Fall 1**: Knoten ist innerer Knoten
  - Falls Schlüssel gleich gesuchtem Wert, dann gefunden
  - Ansonsten abhängig von Vergleich zwischen Wert und Schlüssel links oder rechts weitersuchen
- **Suchen, Fall 2**: Knoten ist Blatt → Wert nicht gefunden
- Aufwand: **O(**Höhe des Baumes**)**

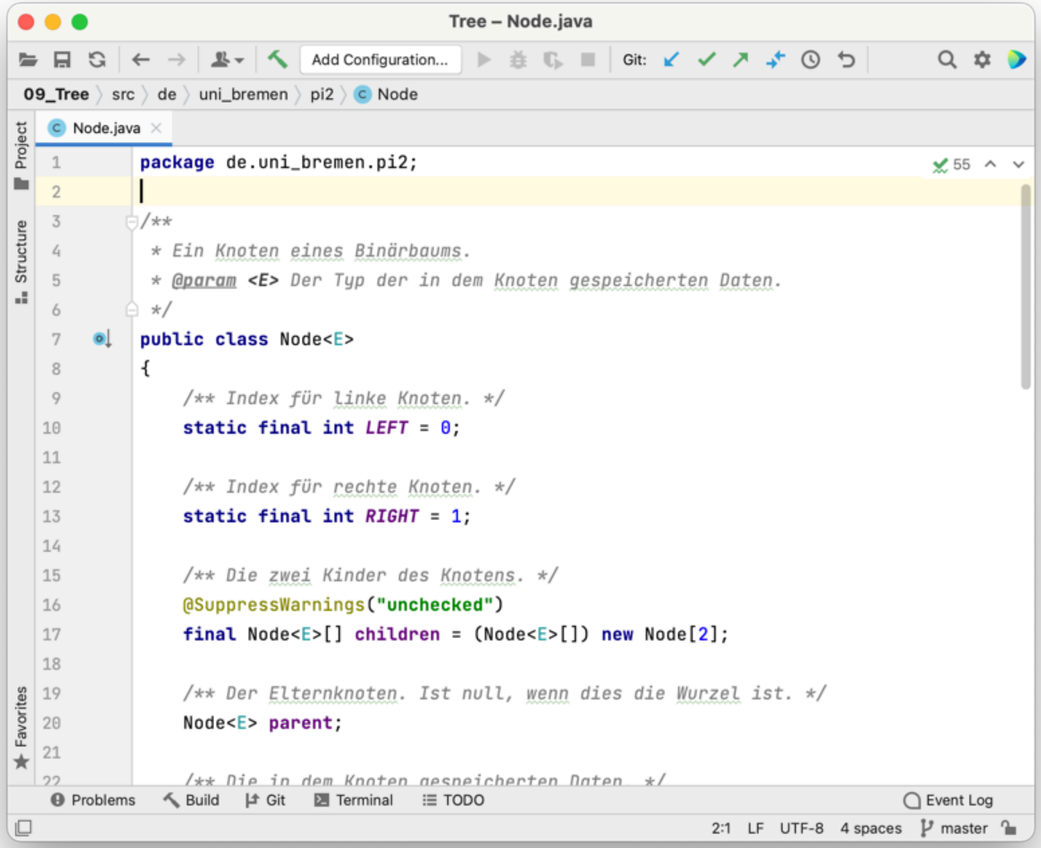
Blattsuchbäume

- Werte stehen in Blättern, innere Knoten enthalten nur „Wegweiser“, z.B. größten Wert des linken Teilbaums
- **Suchen, Fall 1**: Knoten ist innerer Knoten
  - Abhängig von Vergleich zwischen Wert und Wegweiser links oder rechts weitersuchen
- **Suchen, Fall 2**: Knoten ist Blatt
  - Wenn Wert gleich Schlüssel, dann gefunden
  - Ansonsten nicht gefunden

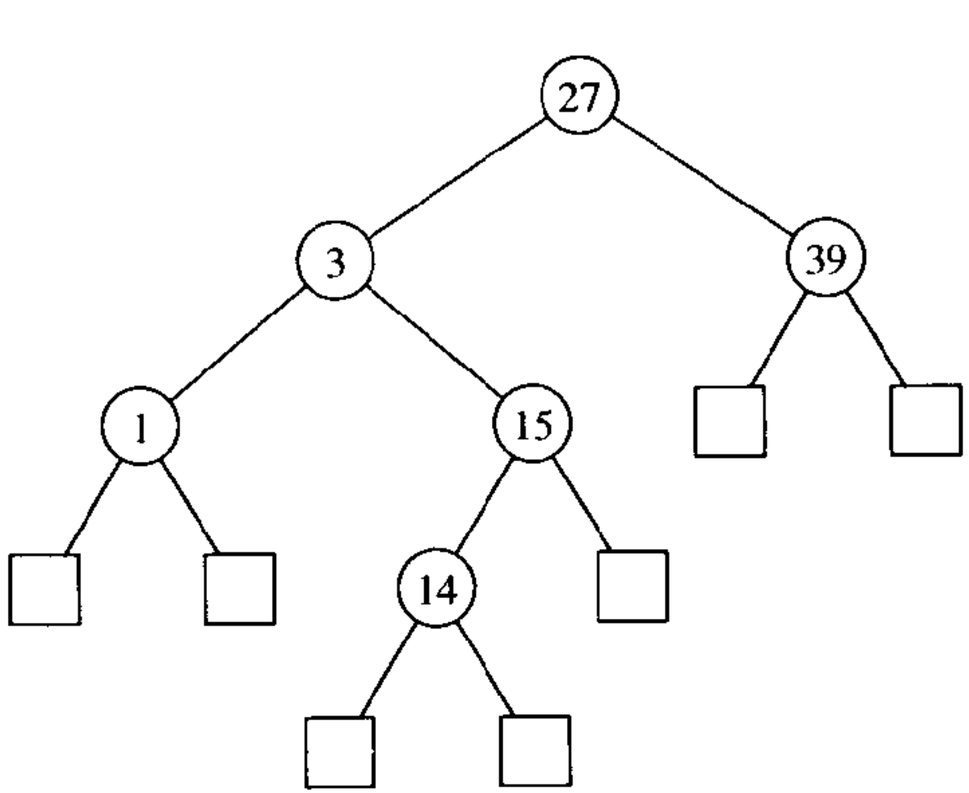


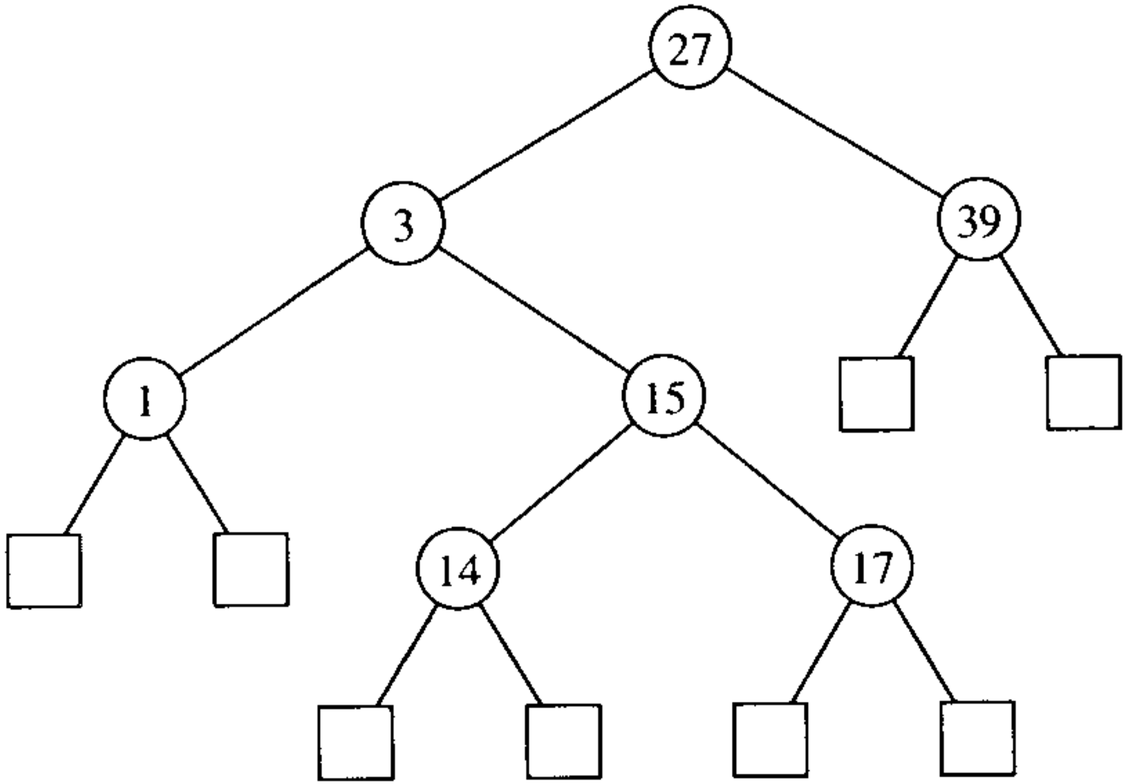
Universität Bremen

Suchen in Suchbaum: Demo



Einfügen in einen Suchbaum: Beispiel





insert(17)

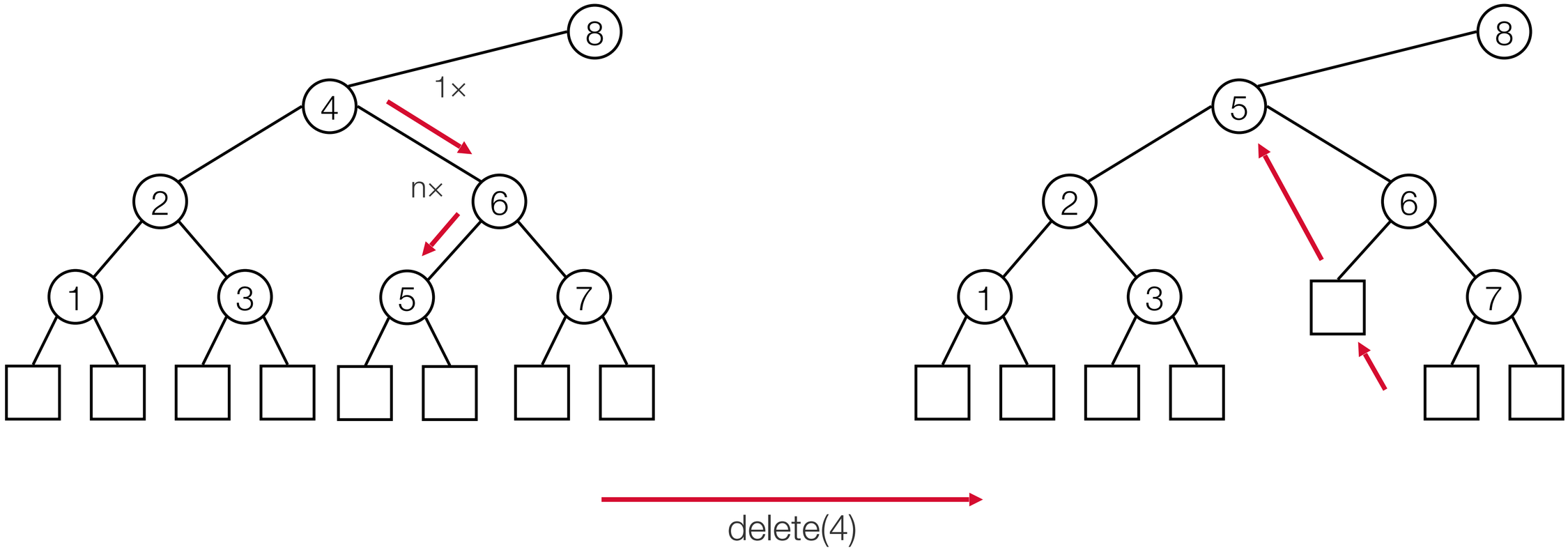
Universität Bremen

Suchbaum: Einfügen

- Suche nach einzufügenden Wert, wobei selbst dann weitergesucht wird, wenn er gefunden wurde
  - Außer, wenn Suchbaum eine Menge repräsentiert
- Dadurch wird auf jeden Fall ein Blatt gefunden
- Dieses wird durch einen neuen Knoten mit dem einzufügenden Wert ersetzt

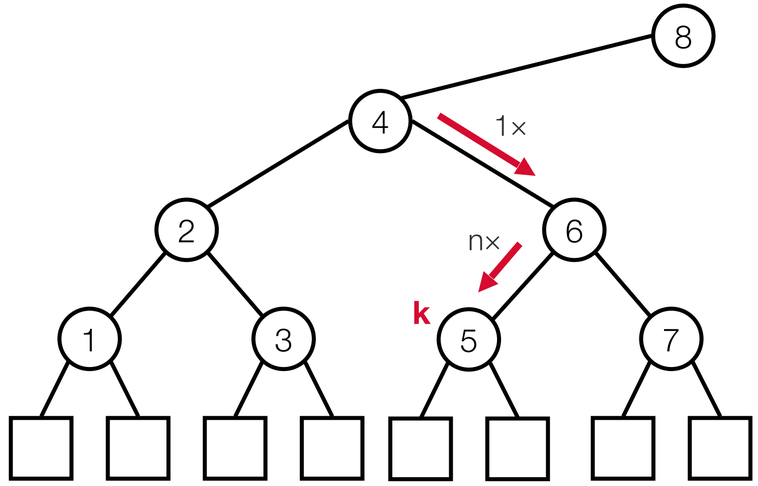
```
public void insert(final E data)
{
    Node<E> parent = null;
    Node<E> current = root;
    int direction = LEFT;
    while (current != null) {
        parent = current;
        direction = data.compareTo(current.data) < 0
                ? LEFT : RIGHT;
        current = current.children[direction];
    }
    final Node<E> child = newNode(data);
    setChild(parent, child, direction);
}
```

Löschen aus einem Suchbaum: Beispiel



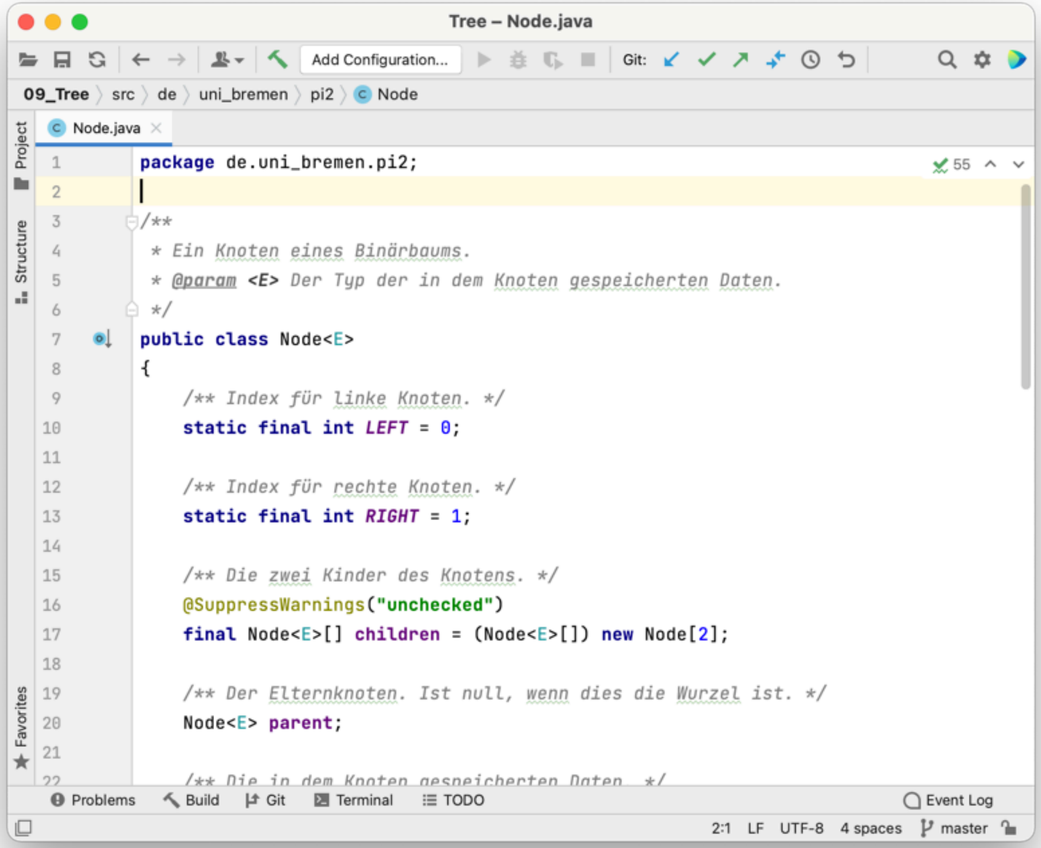
Suchbaum: Löschen

- Suche den zu löschenden Knoten
- Wenn er gefunden wurde
  - Falls einer seiner Nachfolger leer ist, ersetze ihn durch den anderen Nachfolger
  - Ansonsten suche im rechten Teilbaum den Knoten **k** mit dem kleinsten Wert, schreibe dessen Wert in den zu löschenden Knoten und lösche stattdessen den Knoten **k**

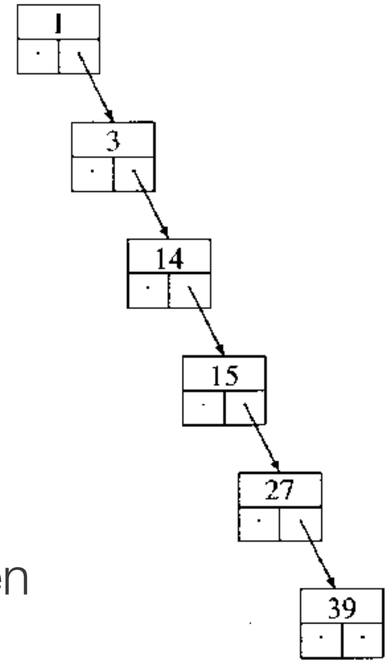


Universität Bremen

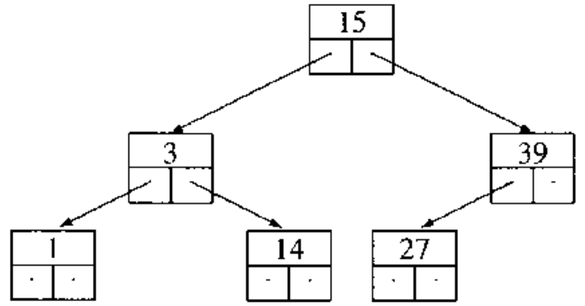
Löschen aus Suchbaum: Demo



Balancierte Bäume

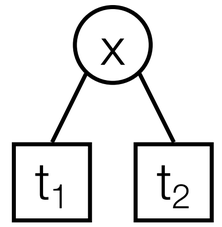


- Suchdauer kann durch degenerierte Bäume stark ansteigen
- Suchbäume sollten balanciert sein, damit dies nicht passiert (Höhe $\leq c \cdot \log_2 n$)
- Idee: Baum ist grundsätzlich balanciert, aber Einfügen und Löschen kann Balance stören
  - Muss nach diesen Operationen wieder hergestellt werden
  - Entlang des Pfades zwischen Änderung und Wurzel
- Zwei zentrale Fragen
  - Wie ist das Kriterium für Balanciertheit?
  - Wie kann es wieder wieder hergestellt werden, wenn es verletzt wurde?

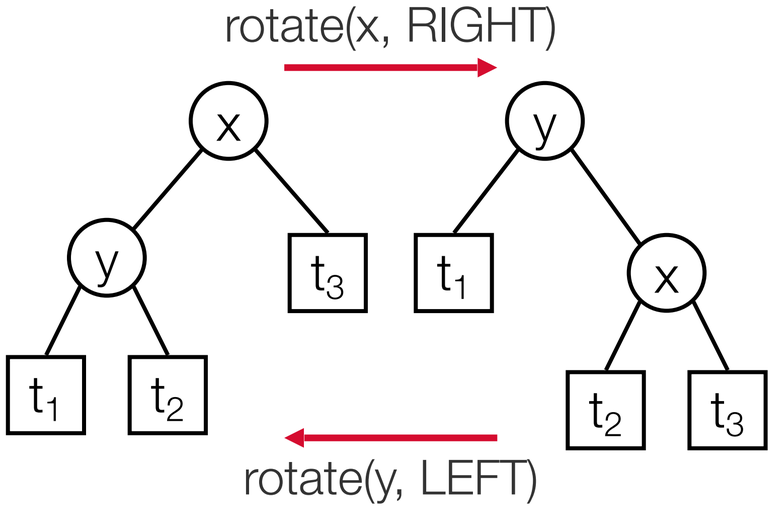


Balancierte Bäume: Rotieren

- Fehlenden Balance eines Knotens bedeutet üblicherweise, dass ein Kindbaum höher ist als der andere



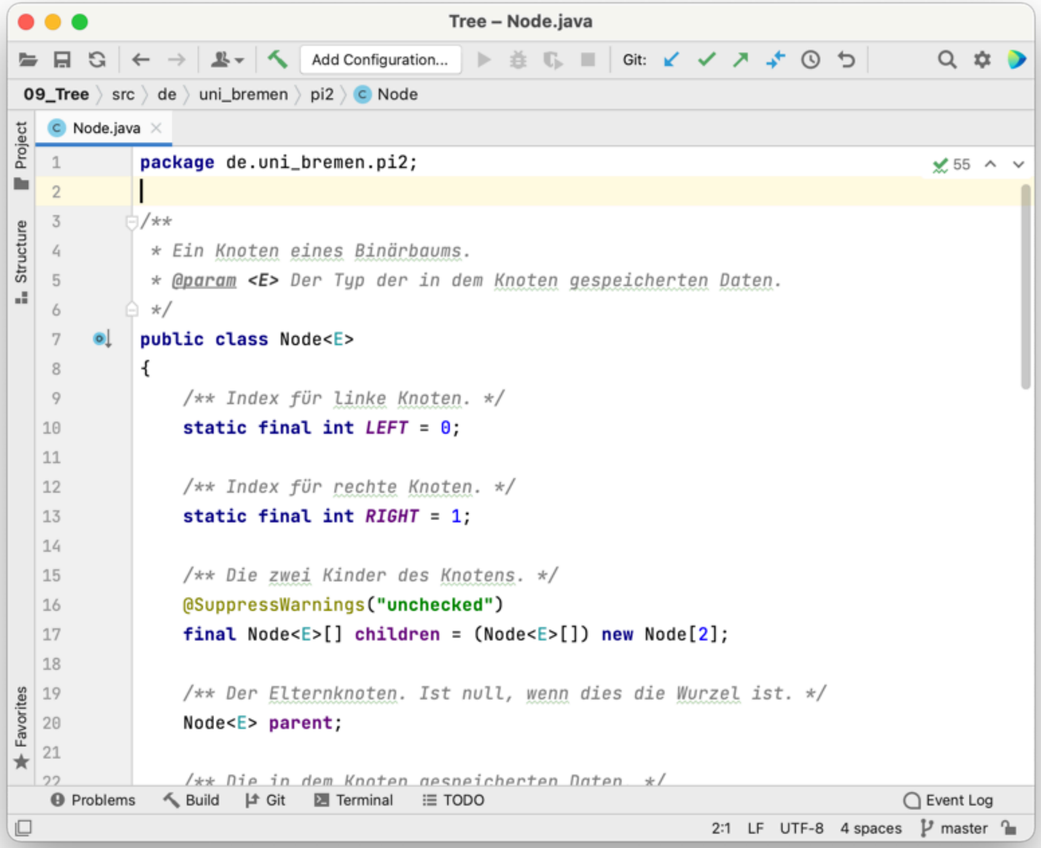
- Rotieren eines Knotens: Verringere Höhe eines Kindbaums, erhöhe die des anderen
  - Der Knoten selbst wandert dabei nach unten
  - **Wichtig**: Sortierung der Schlüssel bleibt erhalten



- Der innere Enkelbaum (**t_2**) wechselt dabei die Seite
  - **Achtung**: Ist er höher als sein Geschwisterbaum, bringt Rotieren nichts

Universität Bremen

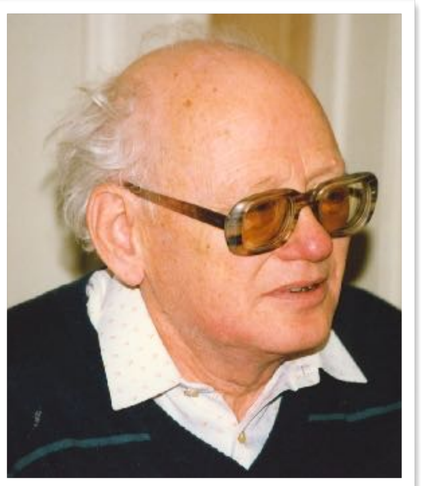
Rotieren: Demo



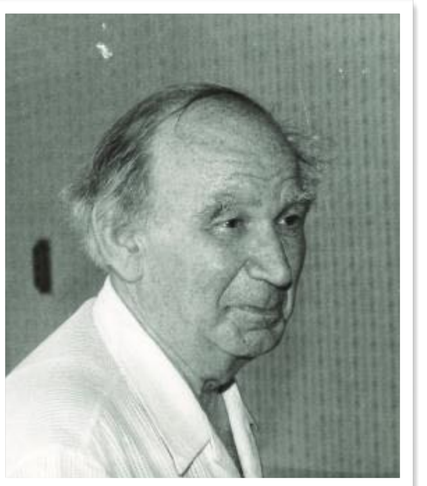
AVL-Baum

- Ein Baum ist **AVL-ausgeglichen** (**höhenbalanciert**), wenn für jeden Knoten des Baumes gilt, dass sich die Höhe seines linken Teilbaums von der des rechten Teilbaums um maximal 1 unterscheidet

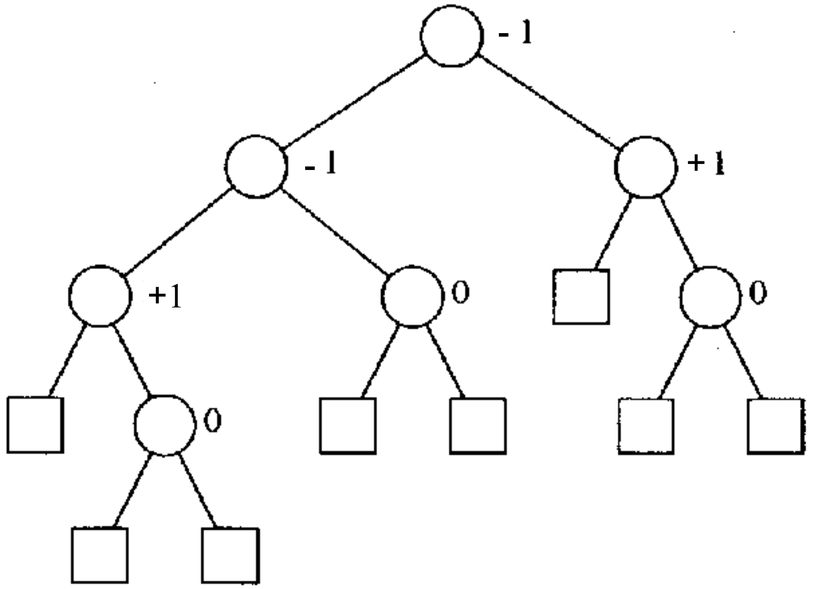
- Dieser Unterschied wird **Neigung** (**slope**) genannt
  - **slope = height(RIGHT) - height(LEFT)**
  - Wird entweder direkt oder indirekt (als Höhe) in den Knoten gespeichert
  - Muss bei Veränderungen am Baum aktualisiert werden



Georgi Maximowitsch
Adelson-Velski

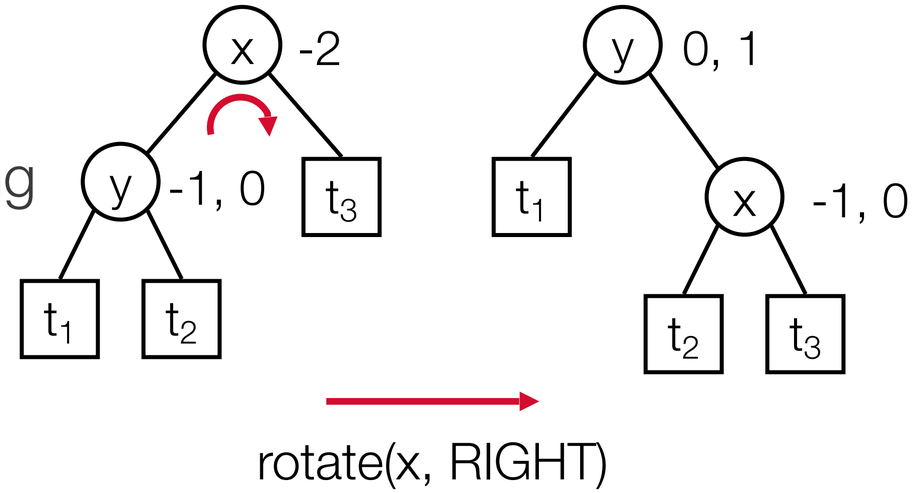


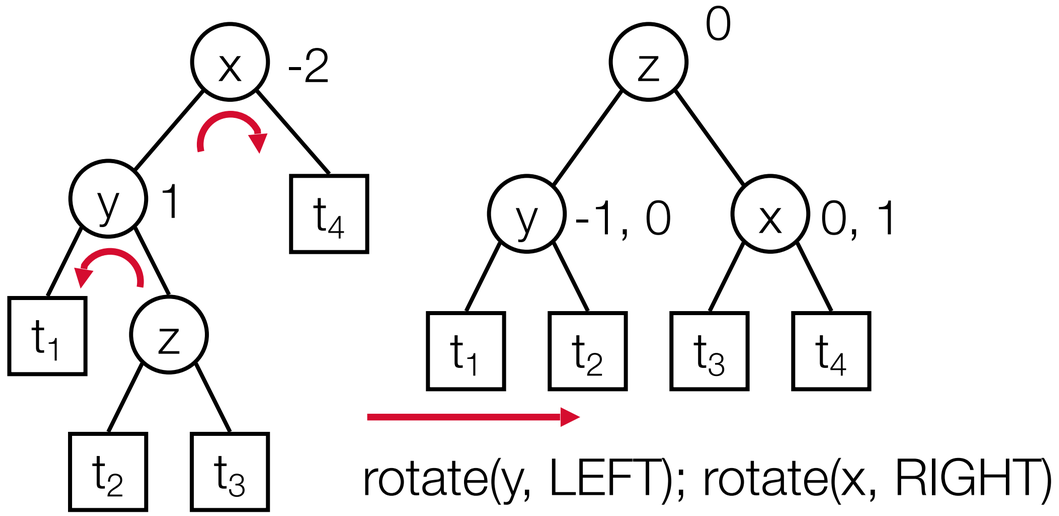
Jewgeni Michailowitsch
Landis



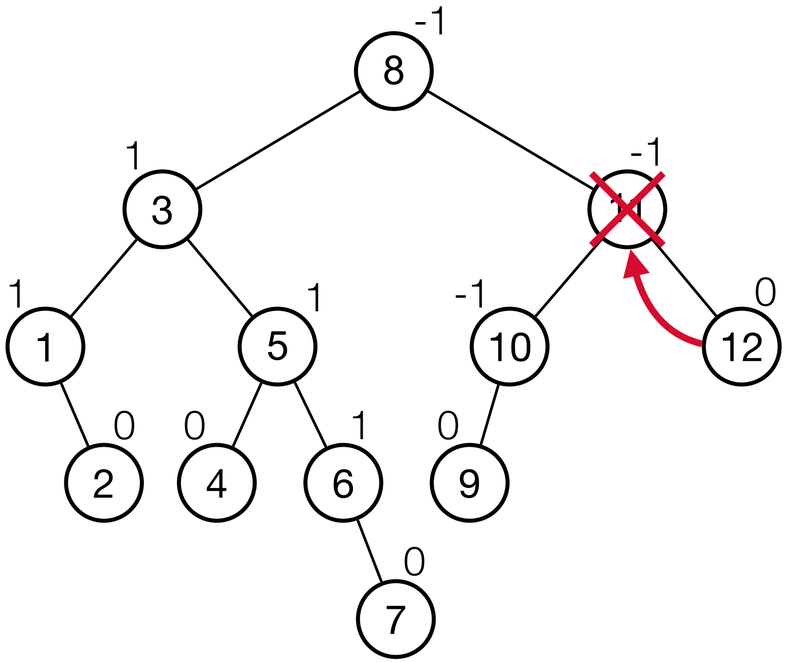
AVL-Baum: Rotieren

- Wenn **|slope| = 2**, rotiere in entgegen gesetzte Richtung
- **Sonderfall**: Wenn zukünftige Wurzel (**y**) nach **innen** geneigt ist, dann diese erst nach **außen** rotieren
- Unbalancierte Knoten können auf dem gesamten Pfad bis zur Wurzel existieren
  - Nach dem Einfügen muss maximal **einmal** (doppel-)rotiert werden
  - Nach dem Löschen muss möglicherweise **öfter** rotiert werden

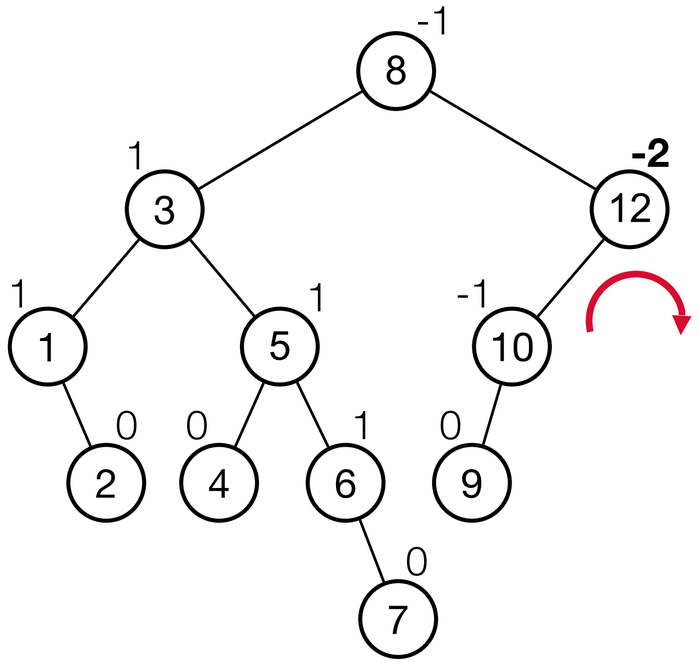




AVL-Baum: Beispiel

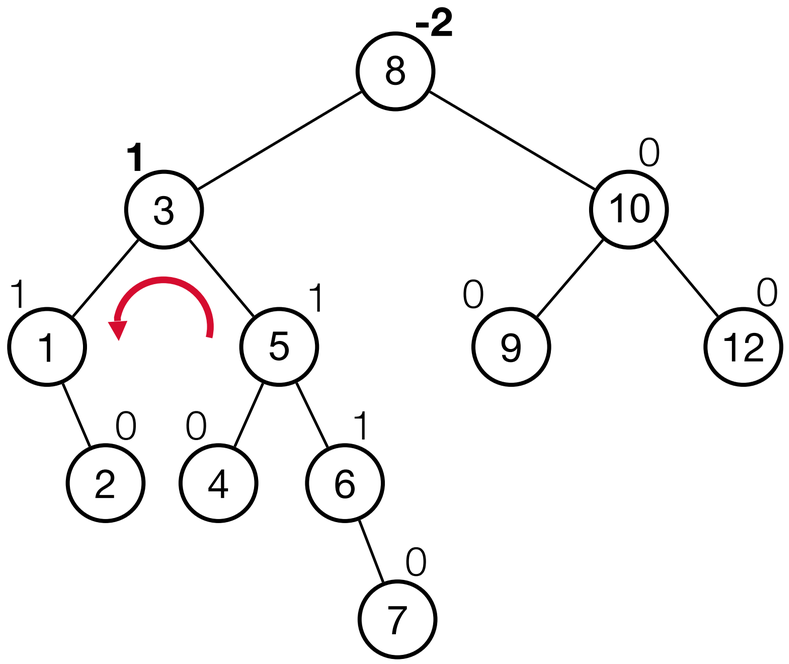


Löschen und Ersetzen

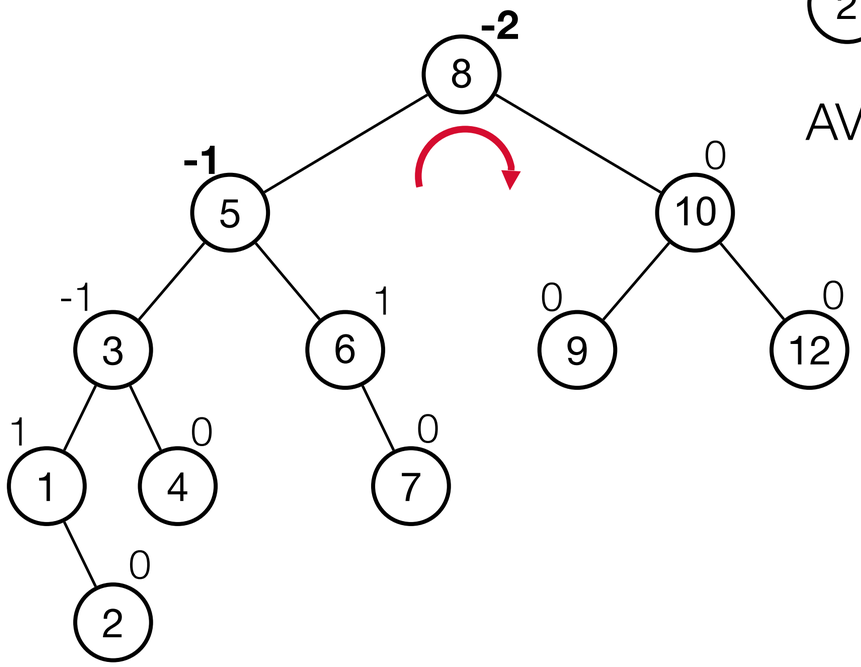


Rechtsrotation

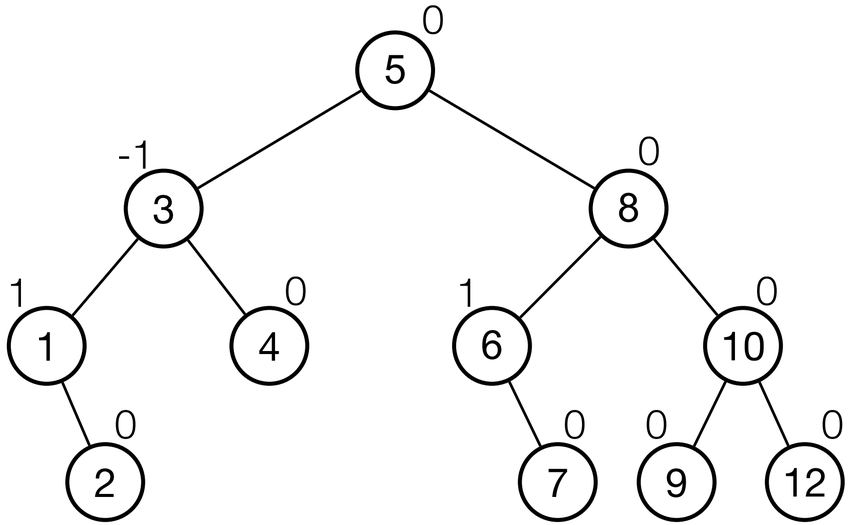
AVL-Baum: Beispiel



Doppelrotation rechts, Teil 1



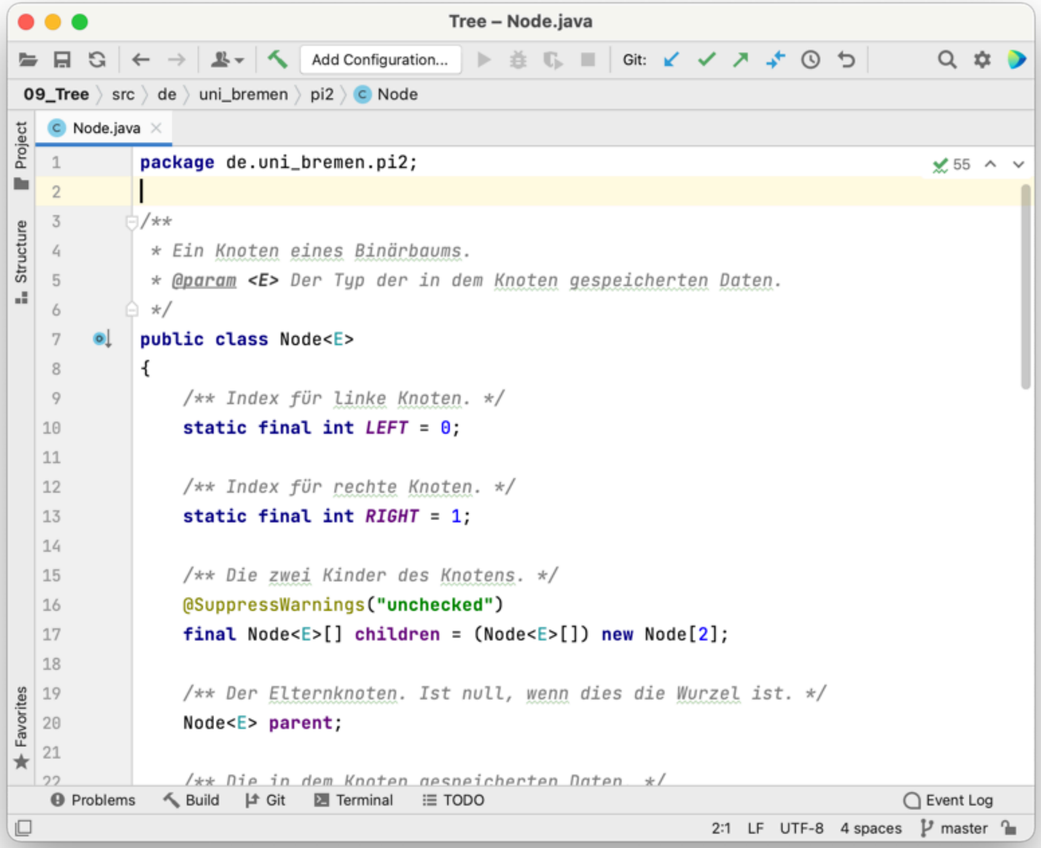
Doppelrotation rechts, Teil 2



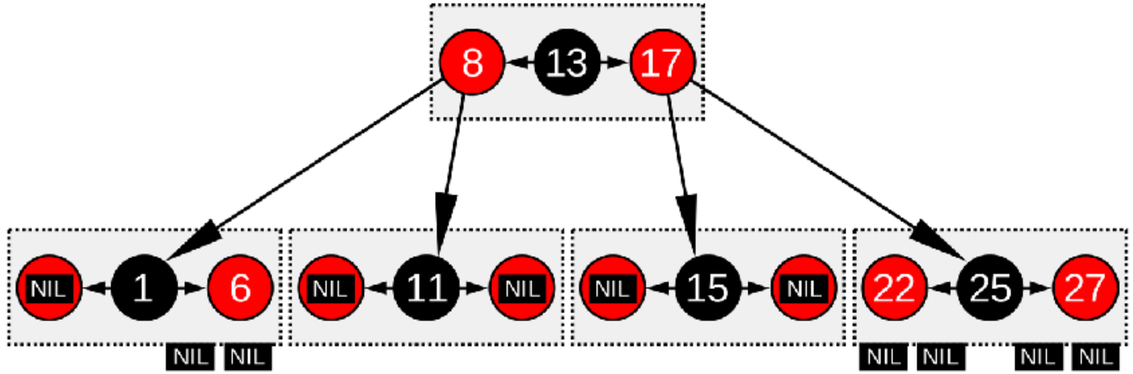
AVL-Bedingung wieder hergestellt

Universität Bremen

AVL-Baum: Demo

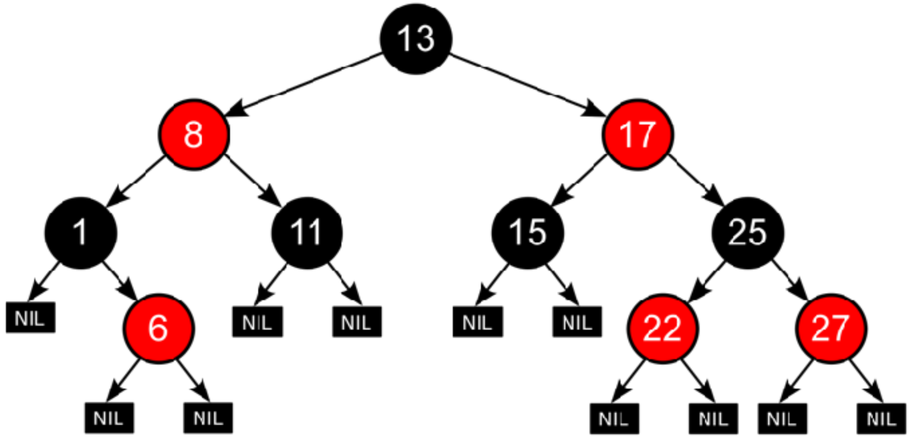


Rot-Schwarz-Baum

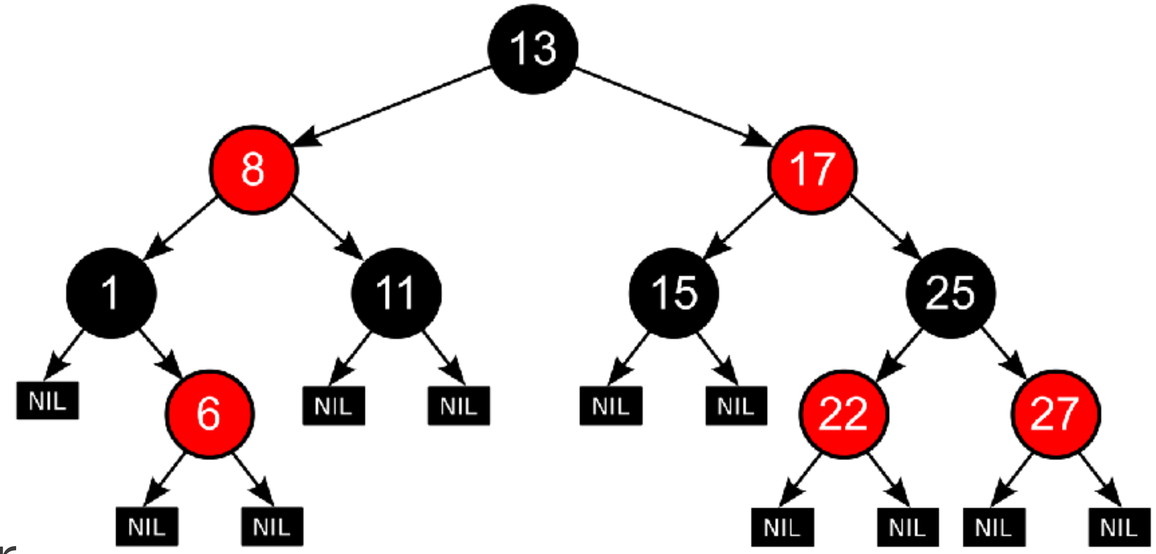


- Von Rudolf Bayer 1972 als Spezialfall der von ihm entwickelten B-Bäume beschrieben
- Idee: Tiefe aller Blätter ist gleich in einem Baum der Ordnung 4
  - Innere Knoten können bis zu drei Werte speichern und bis zu vier Kinder haben
- Umsetzung: Jeder Viererknoten wird durch bis zu drei binäre Knoten repräsentiert
  - Ein schwarzer entspricht dem ursprünglichen Knoten
  - Bis zu zwei rote Kinder speichern weitere Werte
- → **Schwarz**tiefe aller Blätter ist gleich

Rudolf Bayer



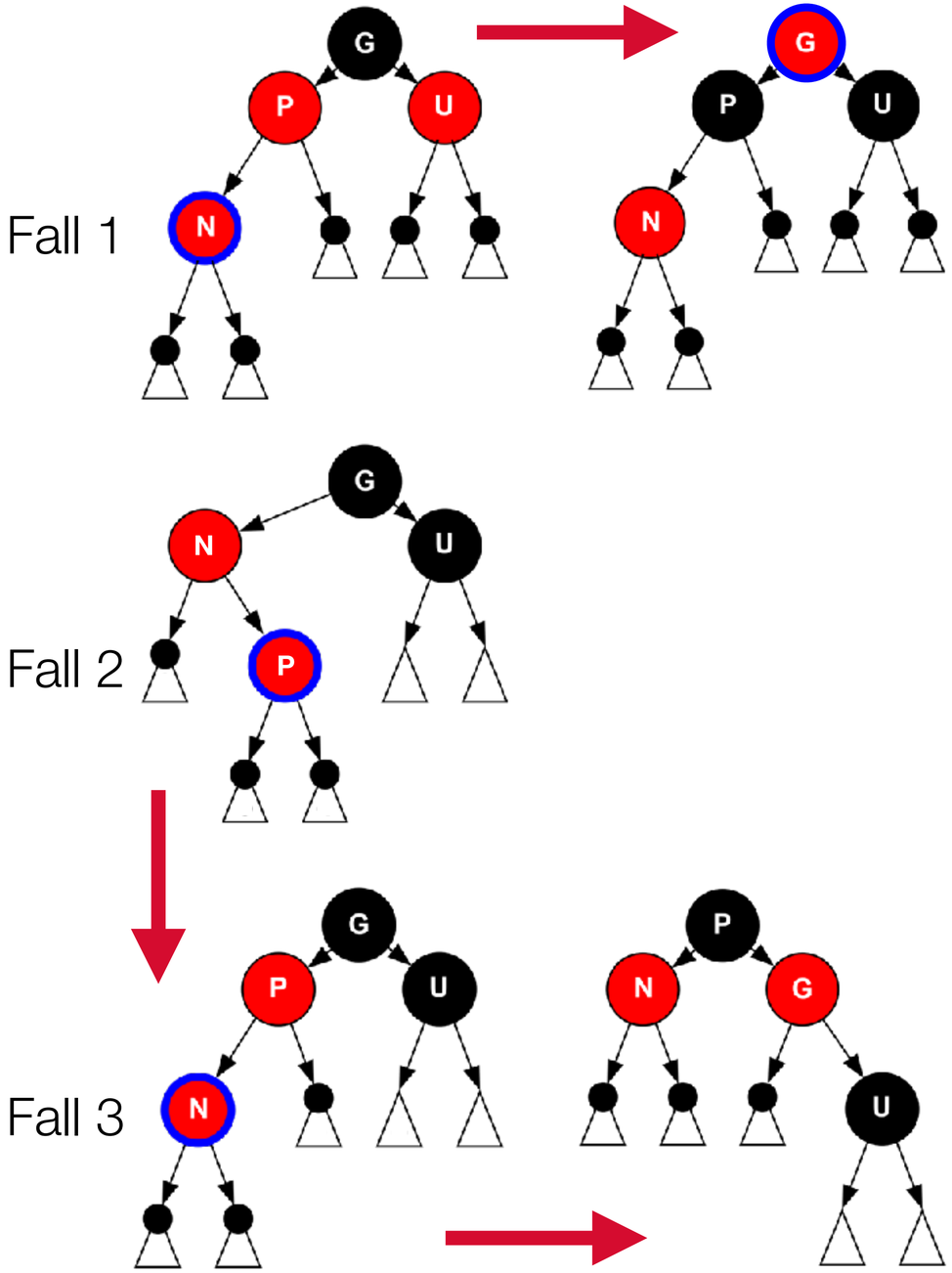
Rot-Schwarz-Baum: Kriterien



1. Knoten sind entweder schwarz oder rot
2. Alle Blätter (null-Knoten) sind schwarz
3. Rote Knoten haben nur schwarze Kinder
4. Für jeden Knoten gilt, dass alle Pfade zu darunter liegenden Blättern gleich viele schwarze Knoten enthalten
5. Die Wurzel ist schwarz (wäre nicht zwingend notwendig)

Rot-Schwarz-Baum: Einfügen

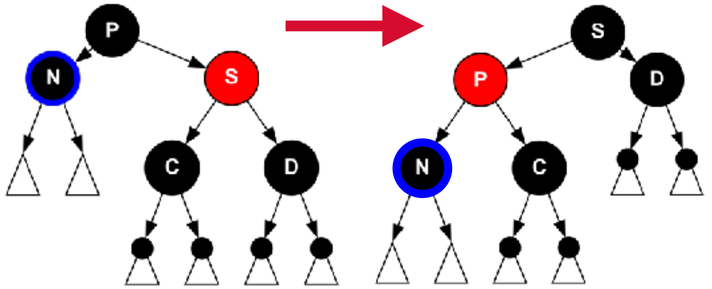
- Neuer Knoten standardmäßig rot (und ein „Problemknoten“)
- Ansatz: Wenn zwei rote Knoten hintereinander, suche zwei schwarze Knoten im Nachbarpfad, zwischen die noch ein roter Knoten passt
- Solange Elternknoten des Problemknotens rot
    1. Tante ist rot → umfärben, mit neuem Problemknoten zwei höher weitermachen
    2. Tante ist schwarz, Problemknoten ist innen → nach außen rotieren, mit Fall 3 weitermachen
    3. Tante ist schwarz, Problemknoten ist außen → Problemknoten hochrotieren, umfärben, fertig
- Die Wurzel auf jeden Fall schwarz färben



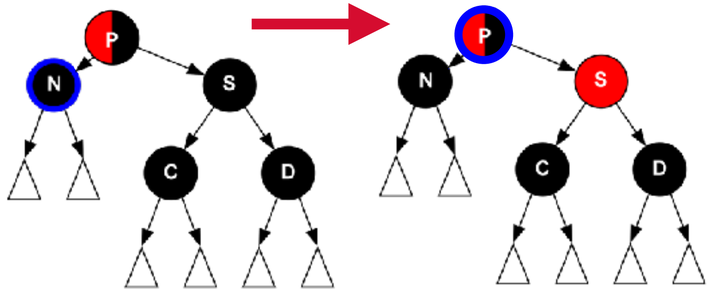
Rot-Schwarz-Baum: Löschen

- Unbalanciert nur, wenn schwarzer Knoten gelöscht wurde
- Ansatz: Auf Pfad zur Wurzel roten Knoten suchen, der schwarz gefärbt werden kann (bis gefunden, Schwarzhöhe von Nachbarn verringern)
- Solange Problemknoten schwarz und nicht Wurzel
    1. Schwester rot → Hochrotieren, umfärben, weitere Fälle betrachten
    2. Beide Nichten schwarz → Schwester rot färben, weitermachen mit Elternknoten als neuem Problemknoten
    3. Innere Nichte rot → rotiere sie nach außen, weiter mit Fall 4
    4. Äußere Nichte rot → rotiere sie hoch, umfärben, Wurzel neuer Problemknoten
- Färbe Problemknoten schwarz

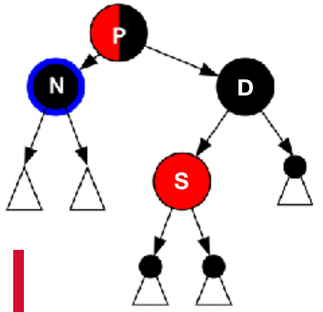
Fall 1



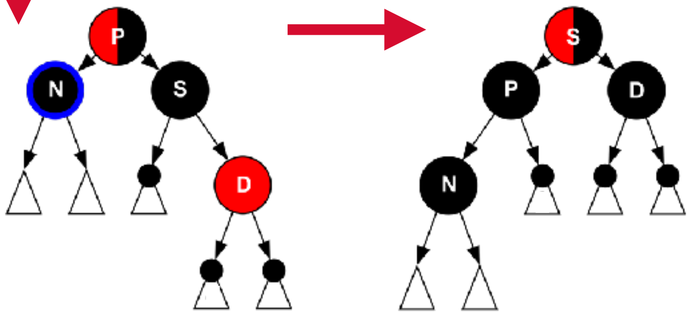
Fall 2



Fall 3

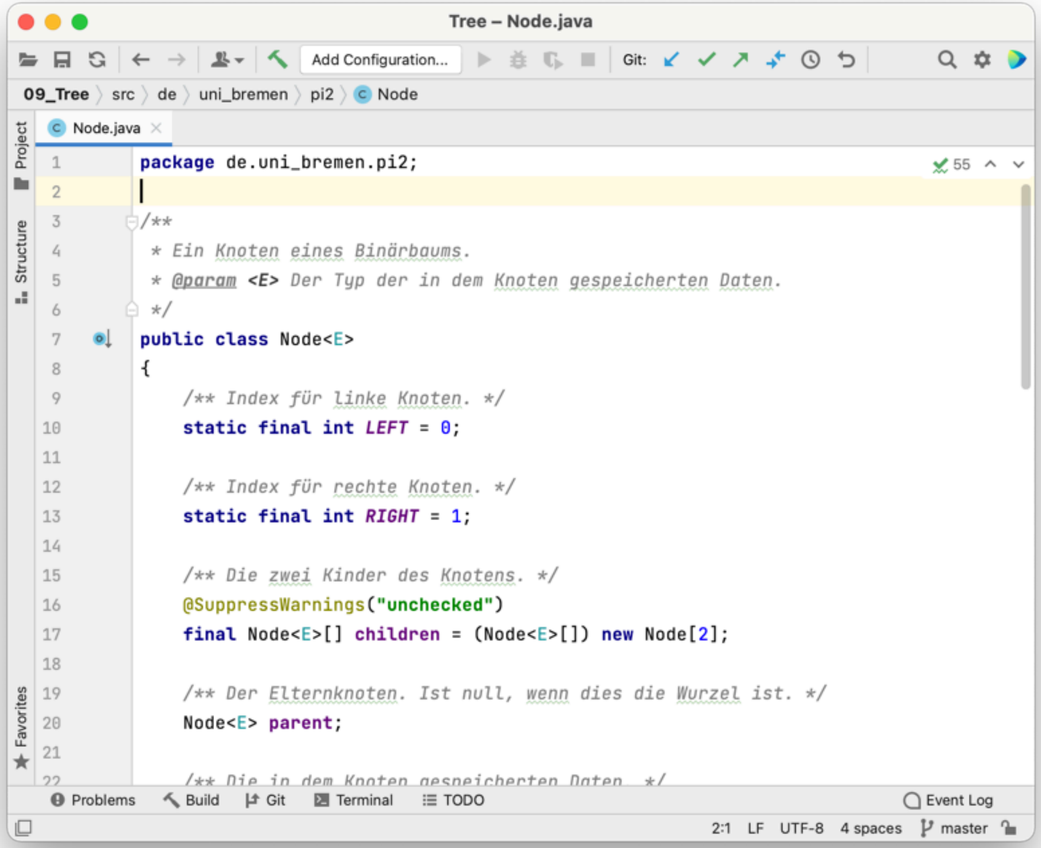


Fall 4



Universität Bremen

Rot-Schwarz-Baum: Demo



Universität Bremen

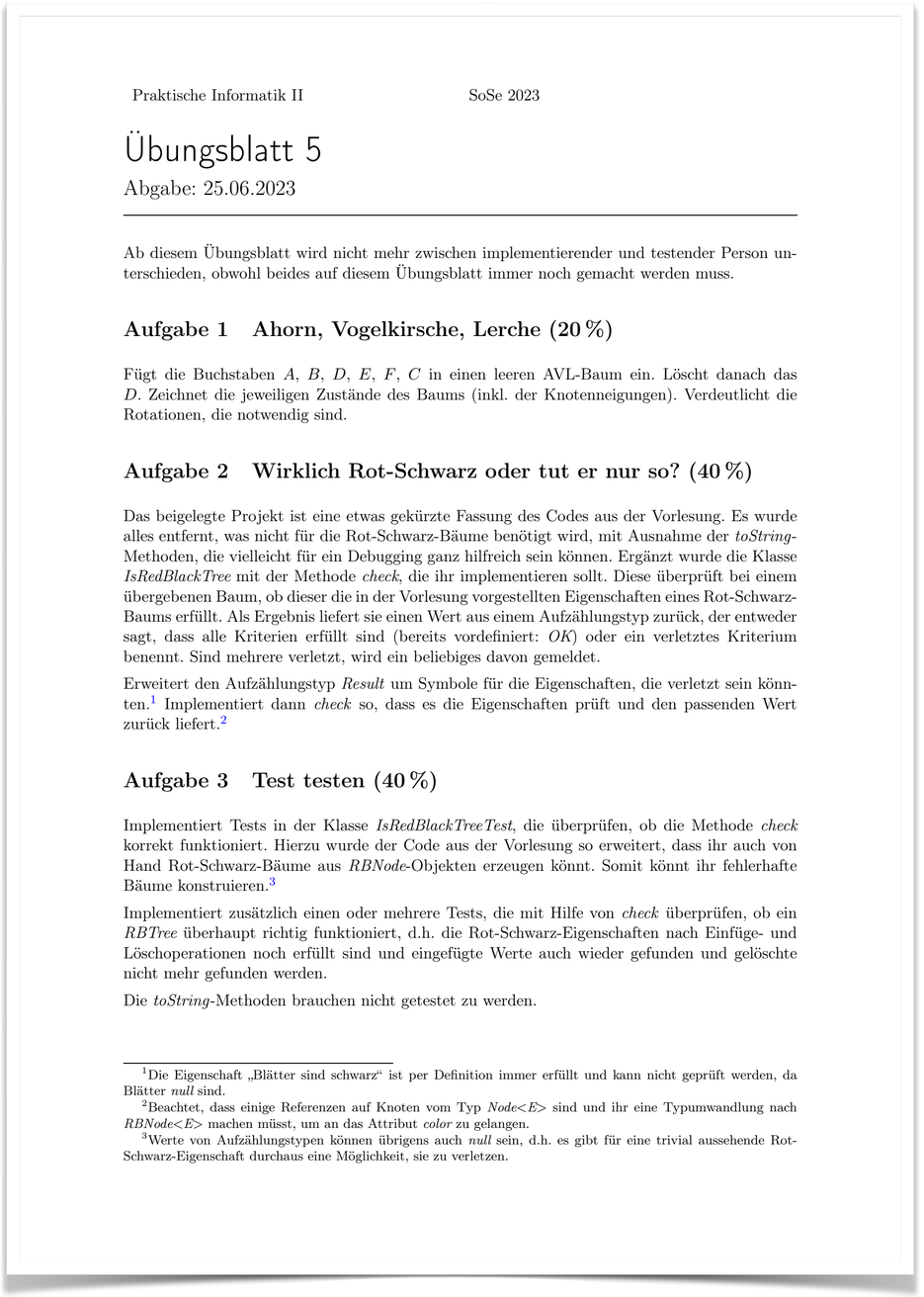
Zusammenfassung der Konzepte

- (**Normaler**) **Suchbaum** und **Blattsuchbaum**
- **Suchen**, **Einfügen** und **Löschen**
- **Balancierter Baum**
  - **AVL-Baum**
  - **Rot-Schwarz-Baum**

Universität Bremen

Übungsblatt 5

- Aufgabe 1: AVL-Bäume zeichnen
- Aufgabe 2: Rot-Schwarz-Baum-Eigenschaften prüfen
- Aufgabe 3: Prüffunktion und Vorlesungsimplementierung testen
- Personen-gebundene Zuordnung von Implementierung und Testen entfällt



Praktische Informatik II SoSe 2023

Übungsblatt 5

Abgabe: 25.06.2023

Ab diesem Übungsblatt wird nicht mehr zwischen implementierender und testender Person unterschieden, obwohl beides auf diesem Übungsblatt immer noch gemacht werden muss.

Aufgabe 1 Ahorn, Vogelkirsche, Lerche (20 %)

Fügt die Buchstaben *A*, *B*, *D*, *E*, *F*, *C* in einen leeren AVL-Baum ein. Löscht danach das *D*. Zeichnet die jeweiligen Zustände des Baums (inkl. der Knotenneigungen). Verdeutlicht die Rotationen, die notwendig sind.

Aufgabe 2 Wirklich Rot-Schwarz oder tut er nur so? (40 %)

Das beigelegte Projekt ist eine etwas gekürzte Fassung des Codes aus der Vorlesung. Es wurde alles entfernt, was nicht für die Rot-Schwarz-Bäume benötigt wird, mit Ausnahme der *toString*-Methoden, die vielleicht für ein Debugging ganz hilfreich sein können. Ergänzt wurde die Klasse *IsRedBlackTree* mit der Methode *check*, die ihr implementieren sollt. Diese überprüft bei einem übergebenen Baum, ob dieser die in der Vorlesung vorgestellten Eigenschaften eines Rot-Schwarz-Baums erfüllt. Als Ergebnis liefert sie einen Wert aus einem Aufzählungstyp zurück, der entweder sagt, dass alle Kriterien erfüllt sind (bereits vordefiniert: *OK*) oder ein verletztes Kriterium benennt. Sind mehrere verletzt, wird ein beliebiges davon gemeldet.

Erweitert den Aufzählungstyp *Result* um Symbole für die Eigenschaften, die verletzt sein könnten.[1] Implementiert dann *check* so, dass es die Eigenschaften prüft und den passenden Wert zurück liefert.[2]

Aufgabe 3 Test testen (40 %)

Implementiert Tests in der Klasse *IsRedBlackTreeTest*, die überprüfen, ob die Methode *check* korrekt funktioniert. Hierzu wurde der Code aus der Vorlesung so erweitert, dass ihr auch von Hand Rot-Schwarz-Bäume aus *RBNode*-Objekten erzeugen könnt. Somit könnt ihr fehlerhafte Bäume konstruieren.[3]

Implementiert zusätzlich einen oder mehrere Tests, die mit Hilfe von *check* überprüfen, ob ein *RBTree* überhaupt richtig funktioniert, d.h. die Rot-Schwarz-Eigenschaften nach Einfüge- und Löschoperationen noch erfüllt sind und eingefügte Werte auch wieder gefunden und gelöschte nicht mehr gefunden werden.

Die *toString*-Methoden brauchen nicht getestet zu werden.

[1] Die Eigenschaft „Blätter sind schwarz“ ist per Definition immer erfüllt und kann nicht geprüft werden, da Blätter *null* sind.

[2] Beachtet, dass einige Referenzen auf Knoten vom Typ *Node<E>* sind und ihr eine Typumwandlung nach *RBNode<E>* machen müsst, um an das Attribut *color* zu gelangen.

[3] Werte von Aufzählungstypen können übrigens auch *null* sein, d.h. es gibt für eine trivial aussehende Rot-Schwarz-Eigenschaft durchaus eine Möglichkeit, sie zu verletzen.