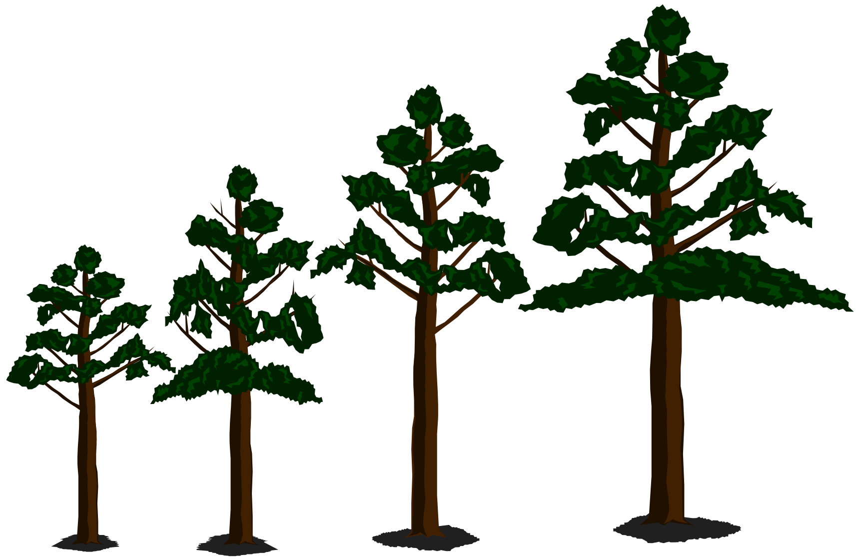
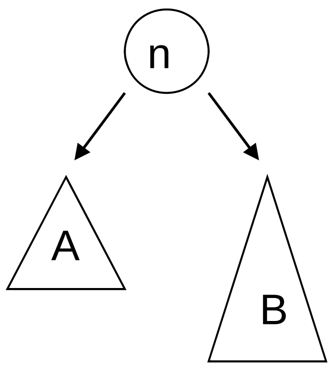
Ausgeglichene Bäume, B-Bäume



- Sie kennen die Bedingungen für die Ausgeglichenheit von Bäumen.
- Sie kennen AVL und B-Bäume.

Balanciertheit von Bäumen

- bei einem vollen Baum sind alle bis auf die letzte Stufe voll gefüllt
- ein Binärbaum hat im optimalen Fall bei n Elementen eine Tiefe von $trunc(log_2 n)$
- i.M. $log_2 n = \ln n / \ln 2$
- wenn man Daten in beliebiger Reihenfolge in einen Binärbaum einfügt, werden die beiden Teilbäume unterschiedlich schwer (Anzahl Knoten) und unterschiedlich tief sein.



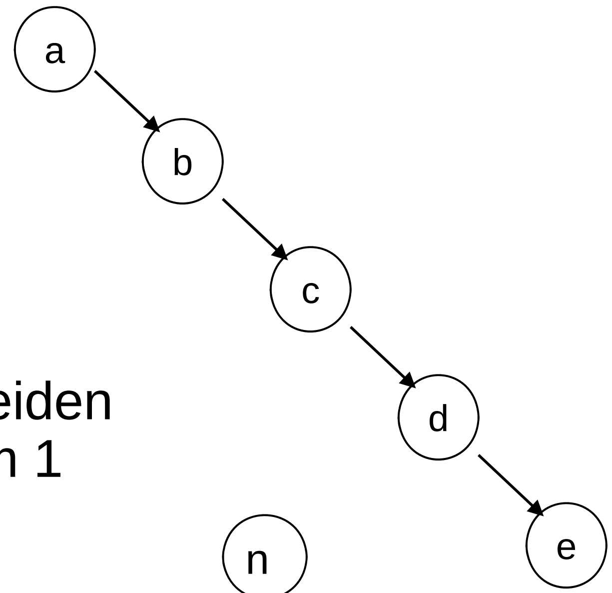
Tiefe im Mittel $2*log_2 n$ (bei gleichverteilten Daten)

Übung

- Zeichnen sie alle möglichen sortierten Binärbäume der Knoten mit den Werten A,B,C auf
- Zeichnen Sie den sortieren Binärbaum auf, der beim Einfügen der Zeichenkette entsteht THEQUICKBROWN
- Erstellen Sie einen optimal balancierten Baum mit den Buchstaben der Zeichenkette: THEQUICKBROWN als Knotenwerten (Anfang des Satzes: "the quick brown fox jumps over the lazy dog")
- Können Sie einen Algorithmus herleiten?

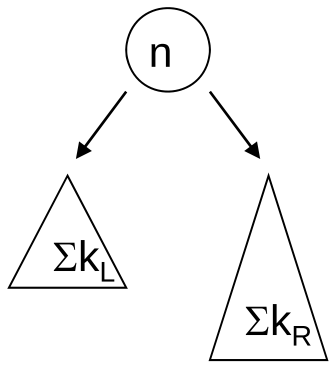
Balanciertheit von Bäumen

- Schlimmster Fall: Daten werden in sortierter Reihenfolge eingefügt
- Der Baum degeneriert zur Liste:



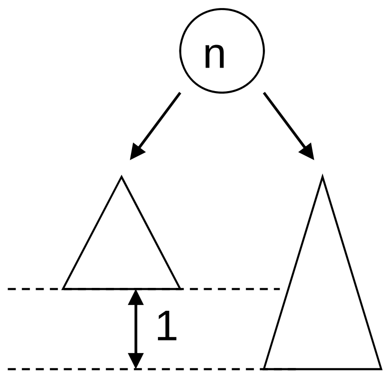
- Vollständig ausgeglichen:
  Für jeden Knoten gilt: Das Gewicht der beiden Teilbäume unterscheidet sich maximal um 1

  Modell: "Ein Mobile, das (fast) optimal ausbalanciert ist"



Ausgeglichenheit von Bäumen: AVL-Bedingung

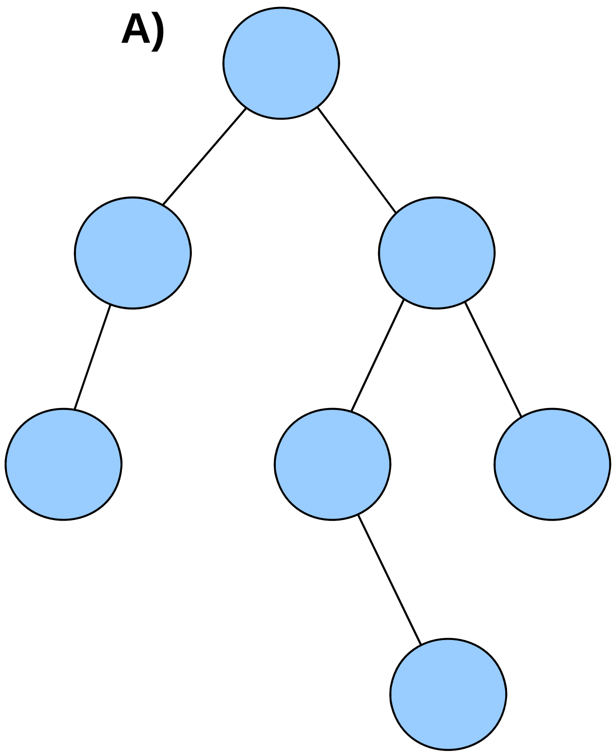
- AVL-Ausgeglichenheit:
  Für jeden Knoten gilt: Die Tiefen der beiden Teilbäume unterscheiden sich maximal um 1
  (AVL: **A**delson-**V**elskij; **L**andis)
- Ein vollständig ausgeglichener Baum ist auch ein AVL-Baum
- Vorteile
  - Einfacher zu realisieren als Gewichtsbedingung
  - Degenerierung zu einer Liste ist nicht möglich
  - Suchoperationen sind schnell
- Nachteile
  - Zusätzlicher Aufwand bei der Programmierung
  - Einfügen und Löschen sind aufwändiger



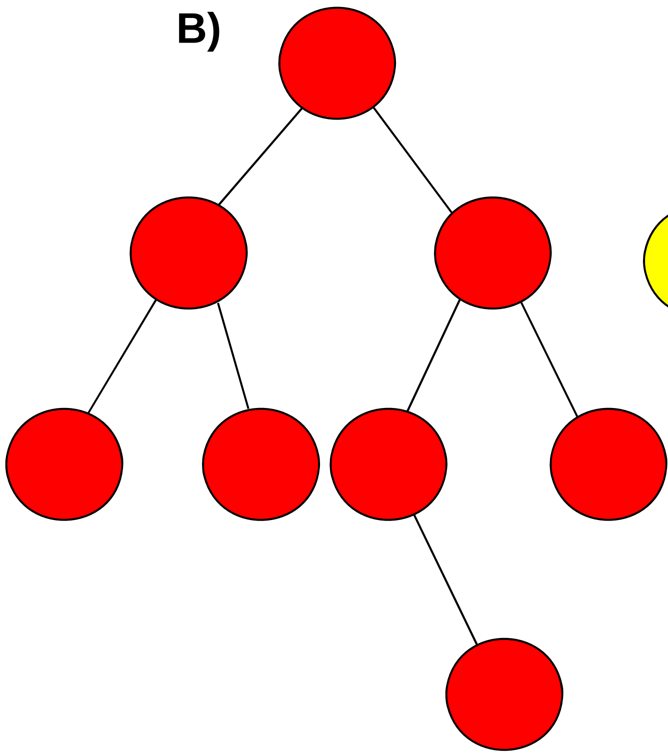
Aufgabe

- Welche der folgende Bäume erfüllen das Kriterium der AVL-Ausgeglichenheit, welche das der vollständigen Ausgeglichenheit

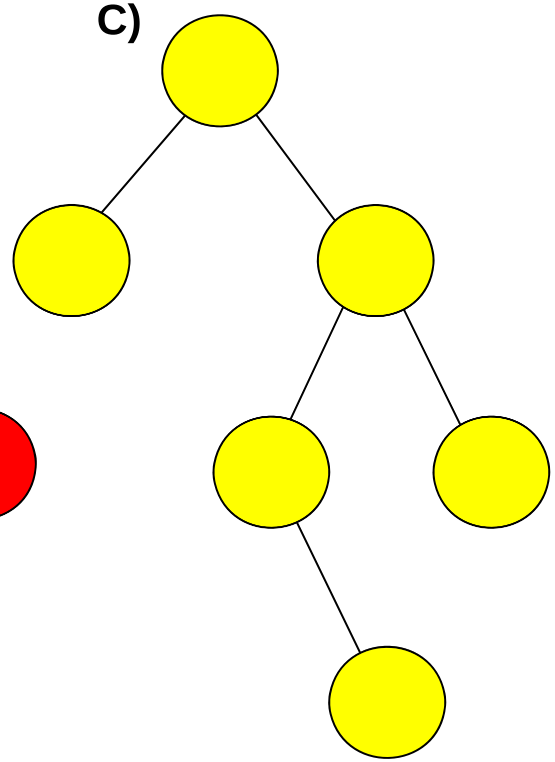
A)



B)



C)

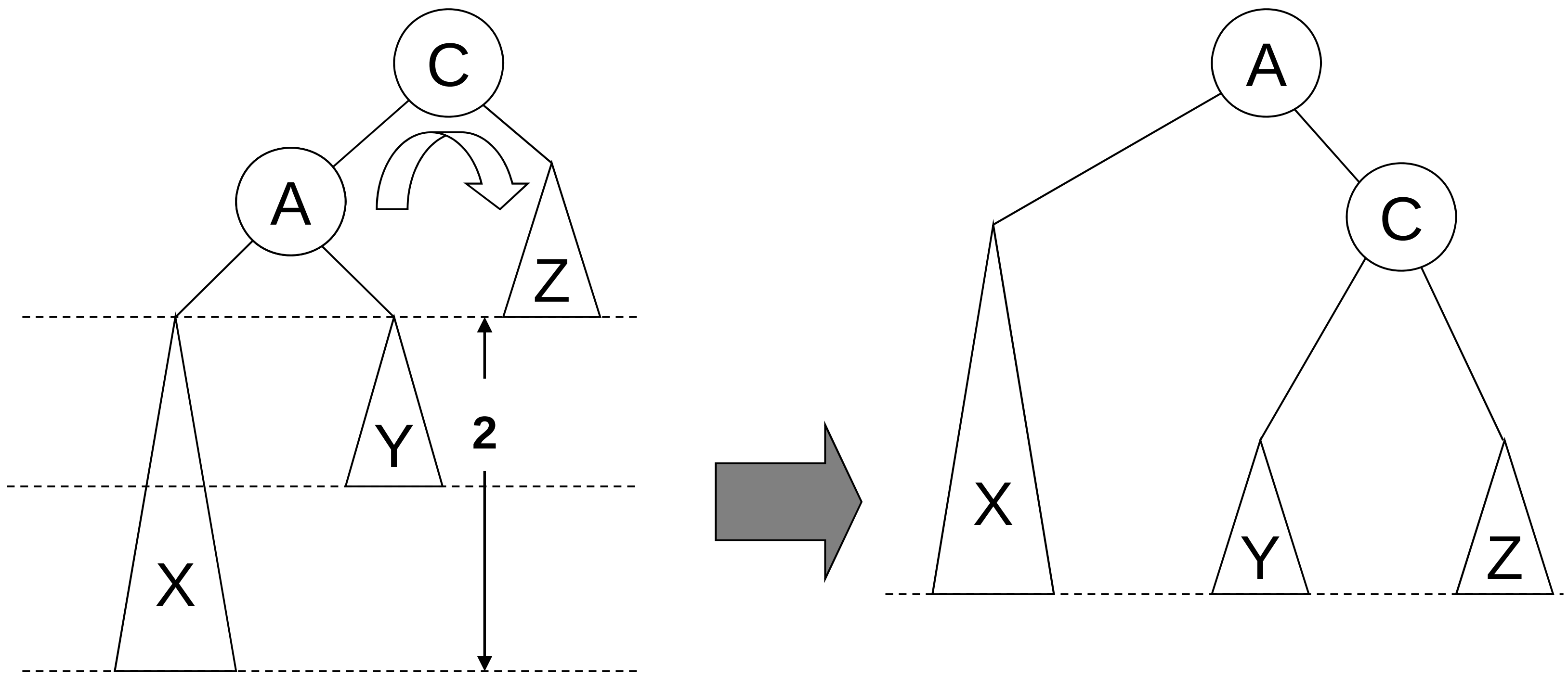


Operationen

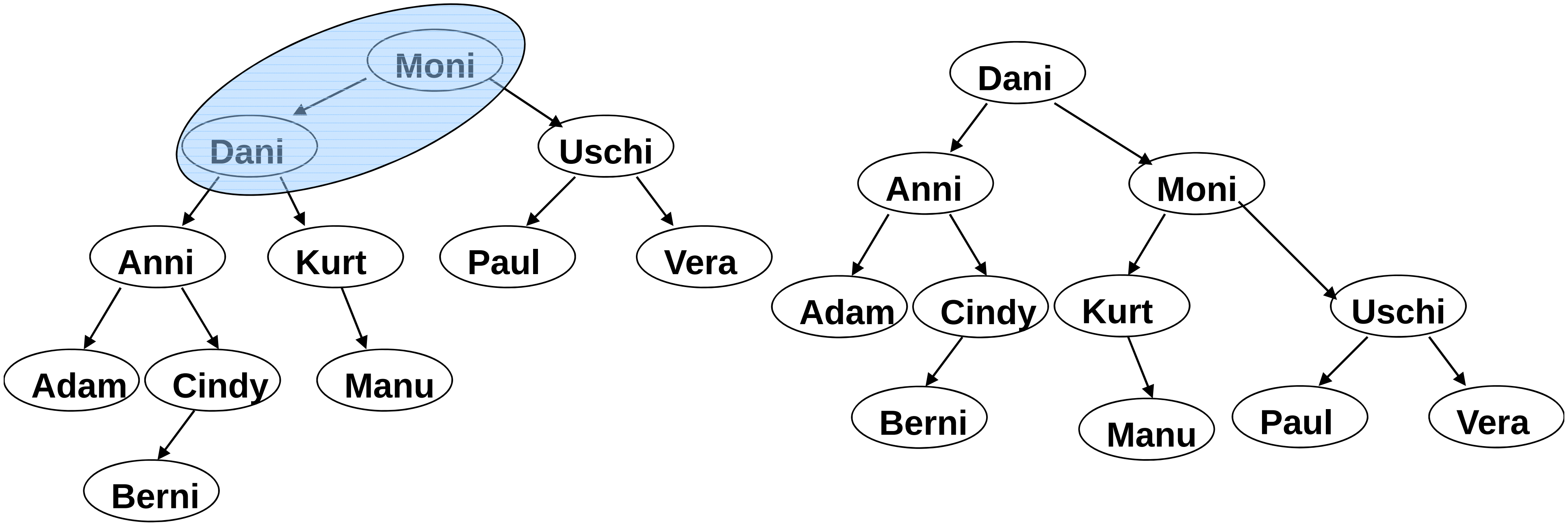
- **Suchoperationen** und **Iterationen** unverändert.
- Bei allen **Einfüge- und Löschoperationen** wird sichergestellt, dass die AVL-Ausgleichsbedingung erhalten bleibt.
- Zum Wiederherstellen der Ausgleichsbedingung werden **Rotationen** eingesetzt.

Einzelrotation

Der linke Teilbaum von C ist um 2 höher als der rechte
➔ Hebe den linken Teilbaum und senke den rechten



Einzelrotation, Beispiel



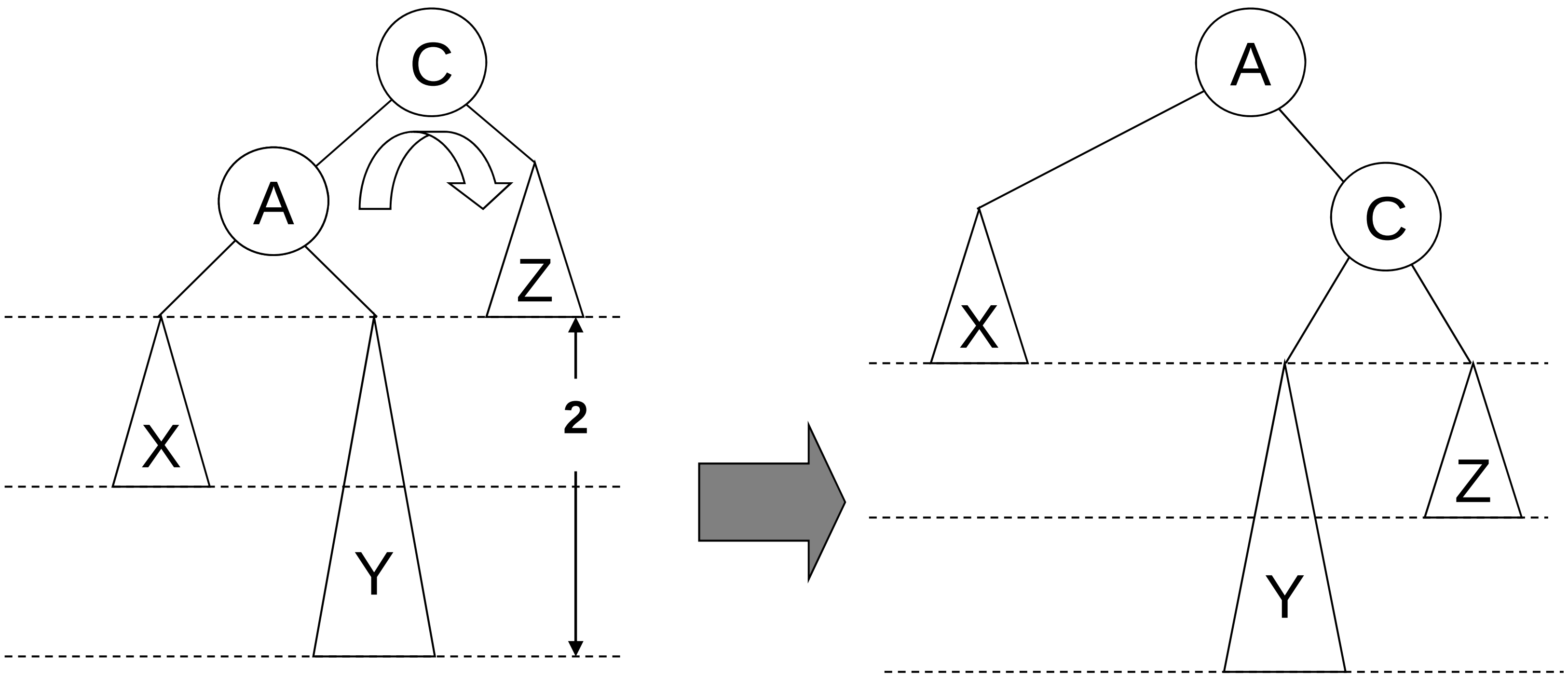

```
Tree<T> rotRight() {
      Tree x = left;
      left = x.right;
      x.right = this;
      return x;
}
```

Beispiel:
Moni = Moni.rotRight()

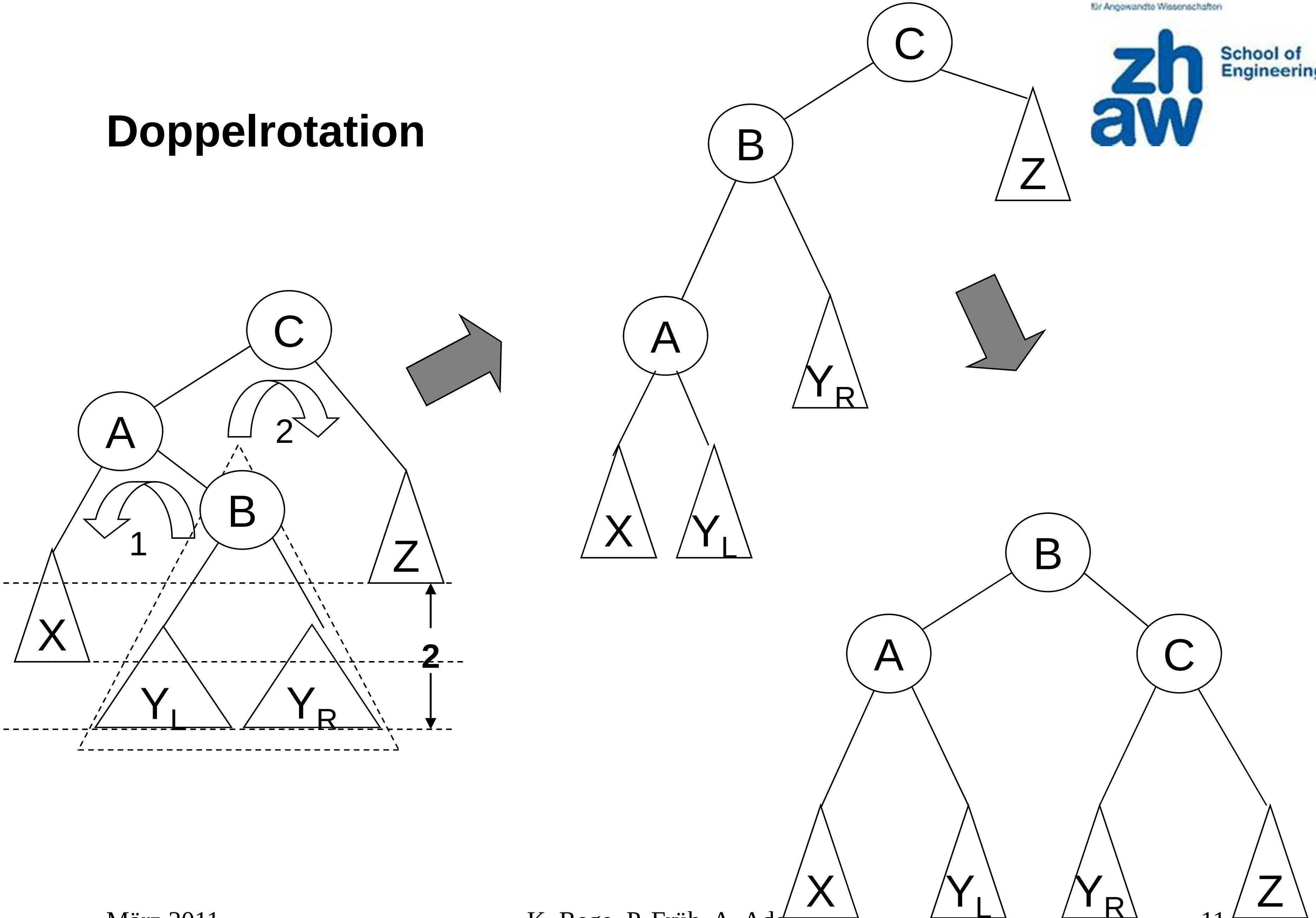
```
Tree<T> rotLeft() {
      Tree x = right;
      right = x.left;
      x.left = this;
      return x;
}
```

Problemfälle bei der Einzelrotation

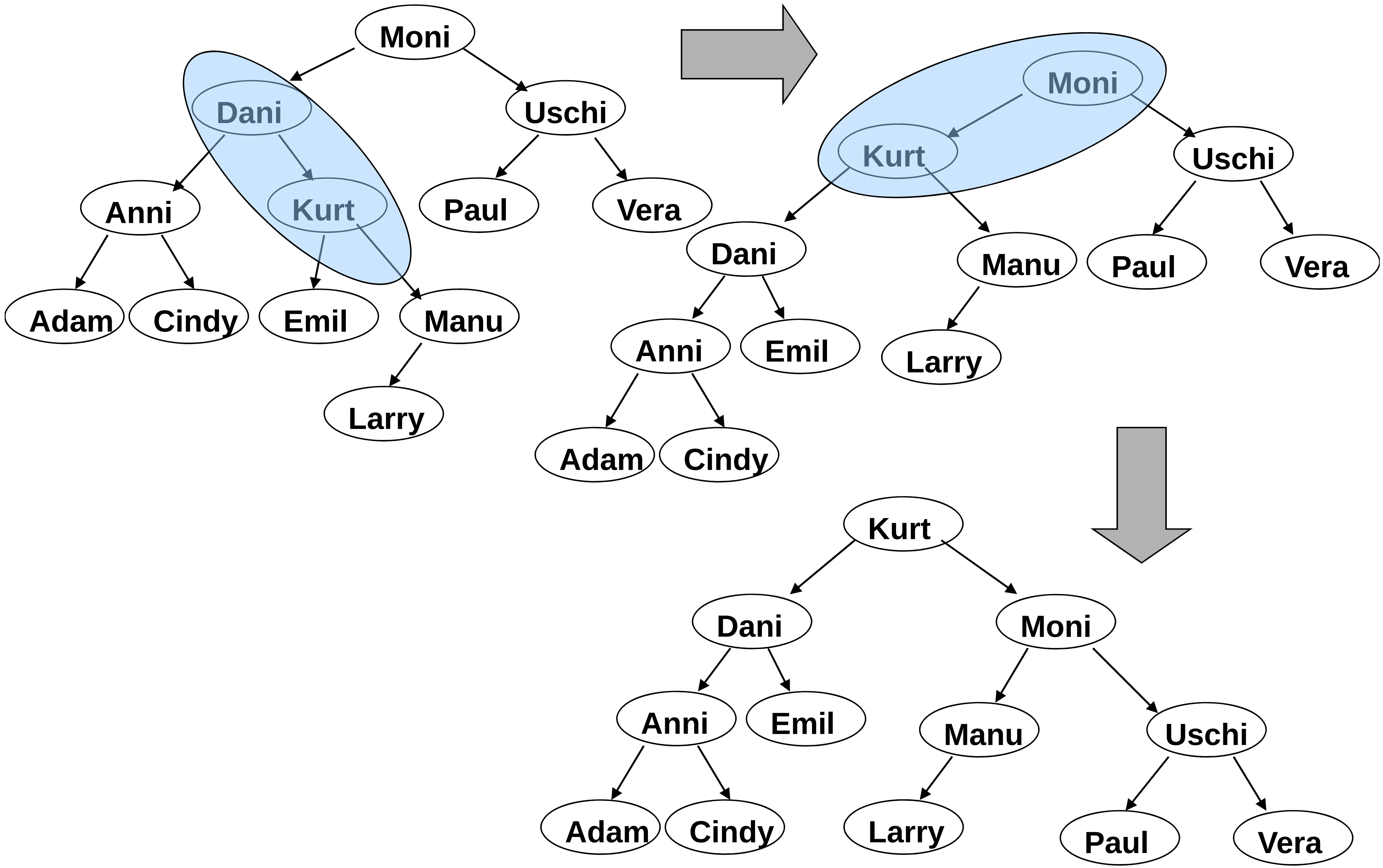
- Der linke Teilbaum von C ist um 2 höher als der rechte, aber der problematische Teilbaum Y liegt in der Mitte.
- Kann nicht mit Einzelrotation balanciert werden.



Doppelrotation



Doppelrotation, Beispiel



AVL-Baum, Java Code

```
class AVLBaum<T extends Comparable<T>> {
    . . .
    T element;
    AVLBaum<T> right;
    AVLBaum<T> left;
    int height;
    . . .
    private static int height(AVLBaum<T> b)
        { return b == null ? -1 : b.height; }
    . . .
}
```

```
private AVLBaum<T> insert(T x, AVLBaum<T> b) {
    if (b == null) b = new AVLBaum<T>(x, null, null);
    // man erkennt hier die Signatur des oben zu definierenden
    // Konstruktors für den AVLBaum
    else if (x.compareTo(b.element) < 0 )
       b.left = insert(x, b.left);
    if(height(b.left) - height(b.right) == 2)
       if (x.compareTo(b.left.element) < 0 ) b = rotateWithLeftChild(b);
       else b = doubleWithLeftChild(b);
    }
    else if (x.compareTo(b.element) > 0 ) { ...analog mit rechts... };
    b.height = max(height(b.left), height(b.right)) + 1;
    return b;
}
```

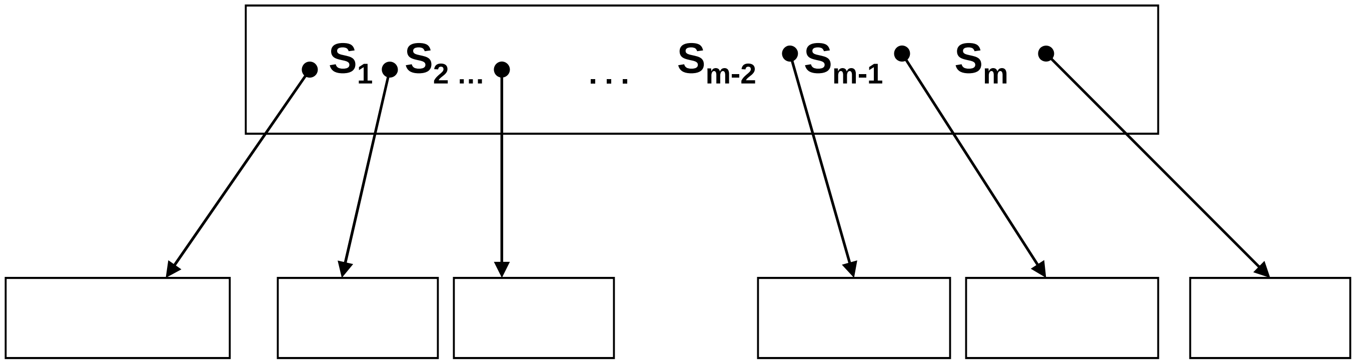
AVL-Baum, Java Code

```
private static AVLBaum<T> rotateWithLeftChild(AVLBaum<T> k2) {
    AVLBaum<T> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max(height(k2.left), height(k2.right)) + 1;
    k1.height = max(height(k1.left), k2.height)) + 1;
    return k1;
}

private static AVLBaum<T> doubleWithLeftChild(AVLBaum<T> k3) {
    k3.left = rotateWithLeftChild(k3.left);
    return rotateWithLeftChild(k3);
}
```

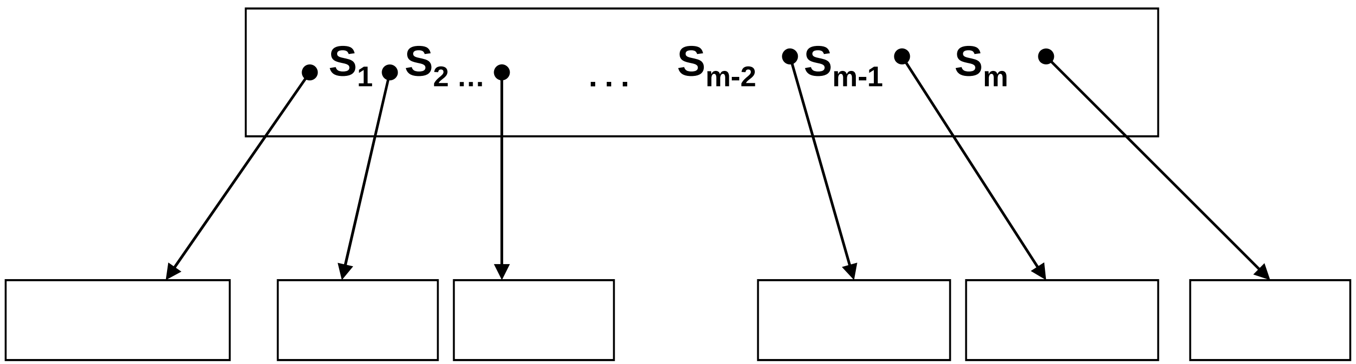
B-Bäume: Bäume mit maximal *n* Kindern

- "B" für balanciert (Rudolf Bayer, 1972):
  B-Bäume werden bei der Konstruktion (Einfügen und Löschen) automatisch balanciert.
- In einem B-Baum der Ordnung *n* enthalten alle Knoten (ausser die Wurzel) mindestens *n*/2 und höchstens *n*-1 Schlüssel.
- Jeder Knoten ist entweder ein Blatt oder enthält *m*+1 Kinder, wobei *m* gleich der Anzahl Schlüssel des Knotens ist.

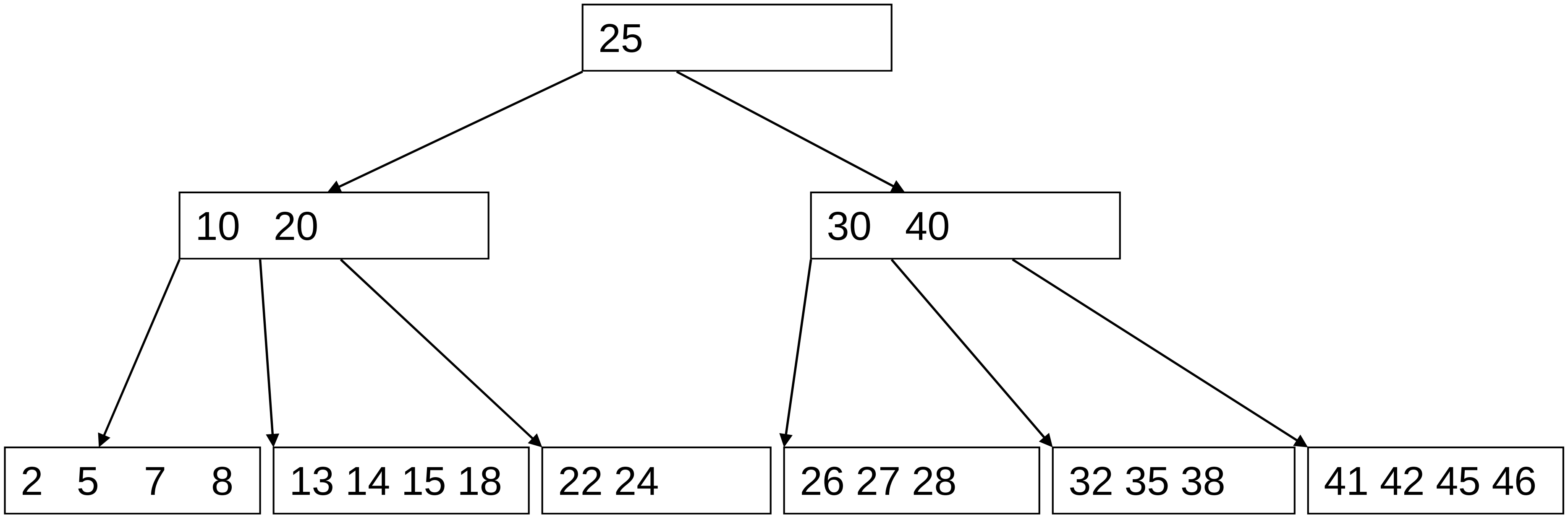


B-Bäume: Bäume mit maximal *n* Kindern

- Für die Schlüssel und Verweise gilt:
  - innerhalb eines Knotens sind alle Schlüssel sortiert
  - in jedem Kindknoten liegen alle Schlüssel zwischen den beiden entsprechenden Schlüsseln im Vater-Knoten
  - alle Schlüssel im 0-ten Kindknoten sind kleiner als der erste Schlüssel S_1, alle Schlüssel im *m*-ten Kindknoten sind grösser als der letzte Schlüssel S_m
- Alle Blätter liegen auf derselben Stufe

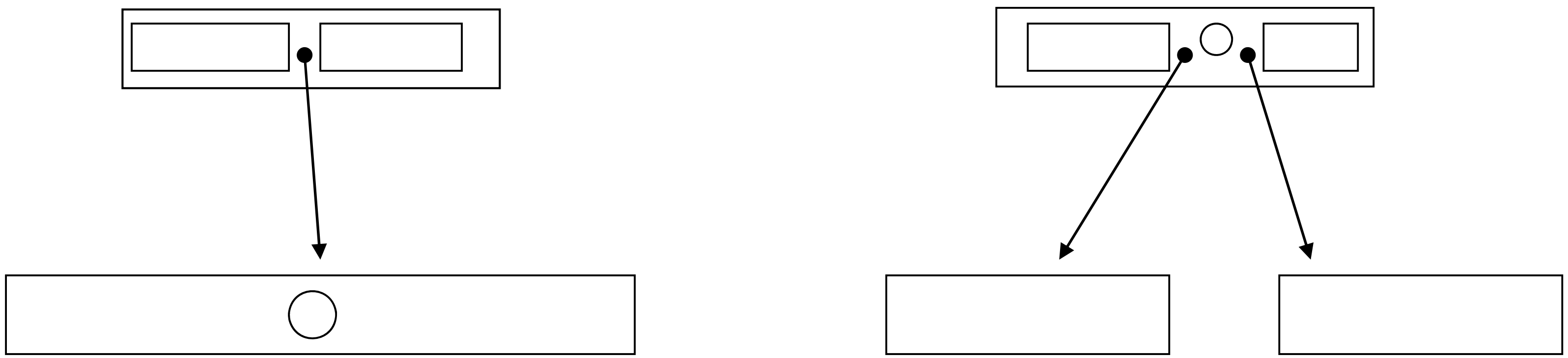


Beispiel eines B-Baums 3. Ordnung

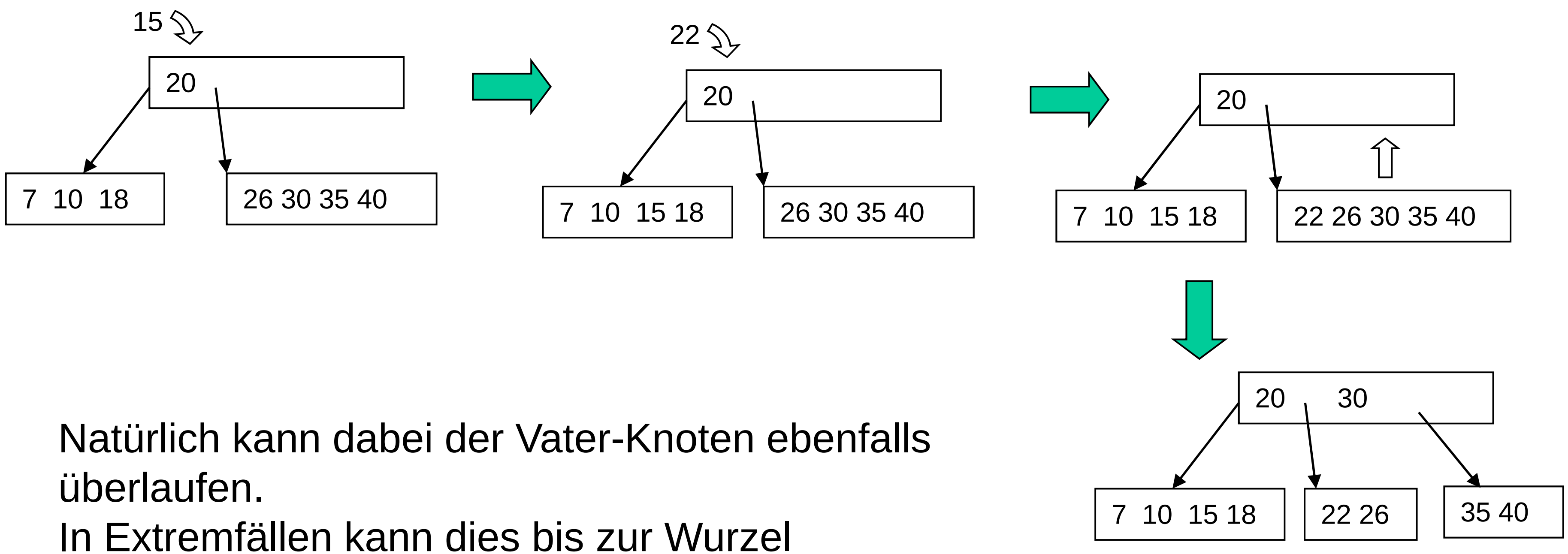


Einfügen

- Eingefügt wird immer in den Blättern.
- Einfügen innerhalb Knoten bis dieser voll: Überlauf
- evtl. Ausgleichen zwischen Nachbarknoten
- sonst: Aufteilen in zwei Knoten und "heraufziehen" des mittleren Elements in den Vaterknoten
- falls dieser überläuft: gleich verfahren



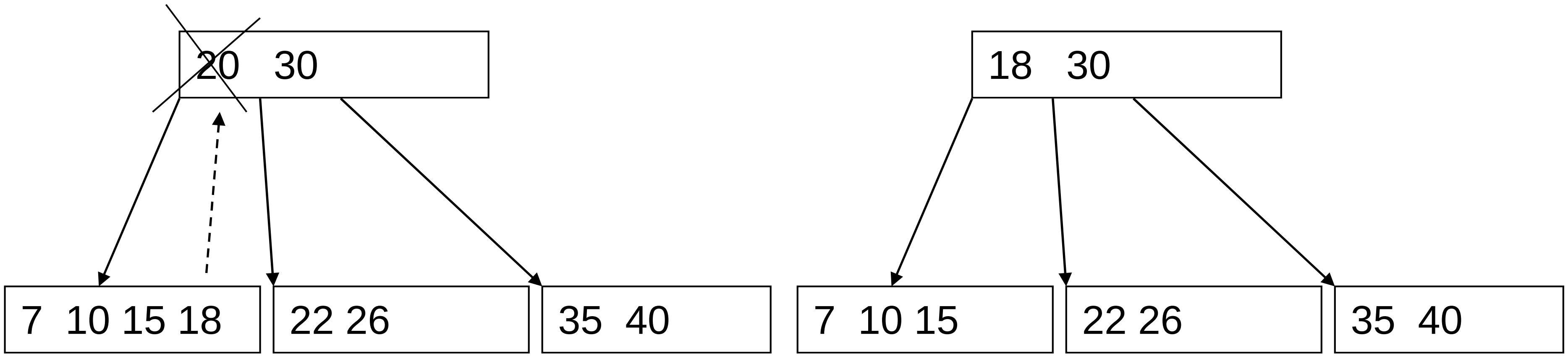
Beispiel einer Einfügeoperation



Natürlich kann dabei der Vater-Knoten ebenfalls überlaufen.
In Extremfällen kann dies bis zur Wurzel propagieren. Dann ändert sich die Höhe des Baumes
➔ B-Bäume wachsen von den Blättern zur Wurzel.

Löschen

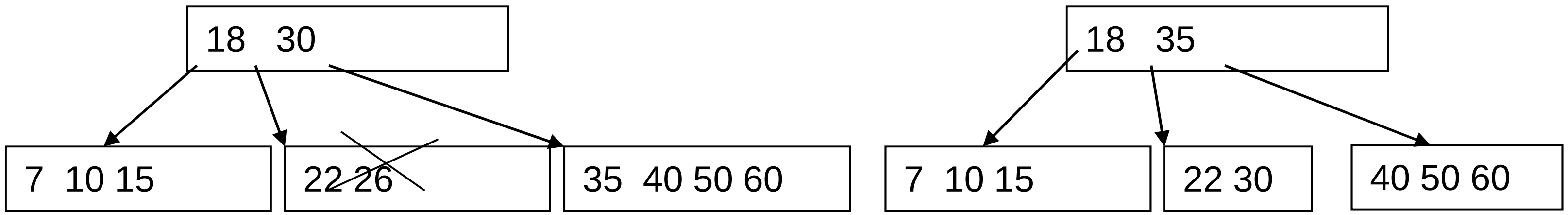
- zu löschendes Element ist in einem Blatt ✓
- zu löschendes Element ist in einem inneren Knoten
  – gleich verfahren wie bei Binärbaum: Ersatzwert suchen



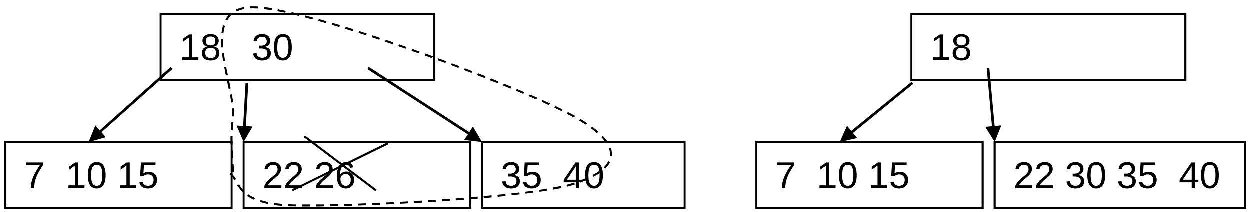
Löschen: Verschmelzen von Knoten

Unterlauf: Ein Knoten enthält weniger als *n*/2 Schlüssel.

- "Ausleihen" bei einem Nachbarknoten



- Wenn nicht beim Nachbarknoten ausgeliehen werden kann ➔ Zwei Knoten werden zu einem zusammengefasst:



Link: Applet, das das Einfügen und Löschen von Elementen in einem B-Baum demonstriert.

http://slady.net/java/bt/view.php

Anwendung

- Organisation der Daten auf Disk mit fester Blockgrösse, z.B. 1024. Z.B. Windows NTFS-Filesystem.
- Ein Diskblock kann entweder Daten enthalten oder n-1 Schlüssel und n Verweise (auf Nachfolgerknoten).
- mit wenigen Diskzugriffen kann der Datensatz gefunden werden.
- Indizierter Datenbankzugriff

Suchen

1) den Wurzelblock lesen
2) gegebenen Schlüssel S auf dem gelesenen Block suchen
3) wenn gefunden, Datenblock lesen fertig

4) ansonsten i finden, sodass $S_i < S < S_{i+1}$
5) Block Nr i einlesen, Schritte 2 bis 5 wiederholen

Tiefe des Baumes $\lceil \log_{\text{Anzahl Verweise}} \text{Anzahl Elemente} \rceil$

Anzahl Zugriffe: proportional zu Tiefe des Baumes

Annahme: mehrere hundert Schlüssel und Verweise pro Block ->
Tiefe des Baumes selten grösser als 5 bis 6

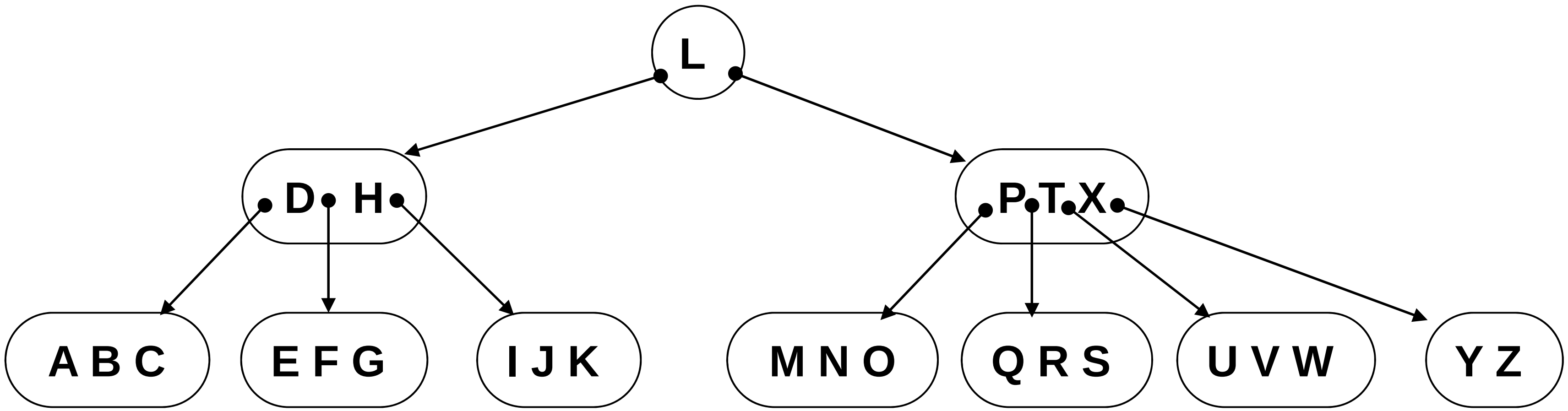
Übung

- Ein Diskblock mit 1024 Bytes pro Knoten
- Ein Schlüssel belege 4 Bytes, ein Zeiger auf einen Kindknoten belege ebenfalls 4 Bytes.
  - Was ist die maximale Ordnung eines B-Baums?
  - Wie tief ist dieser B-Baum bei 1'000'000 Werten falls er
    - maximal
    - minimal

    gefüllt ist?
  - Bei 10^9 Werten?

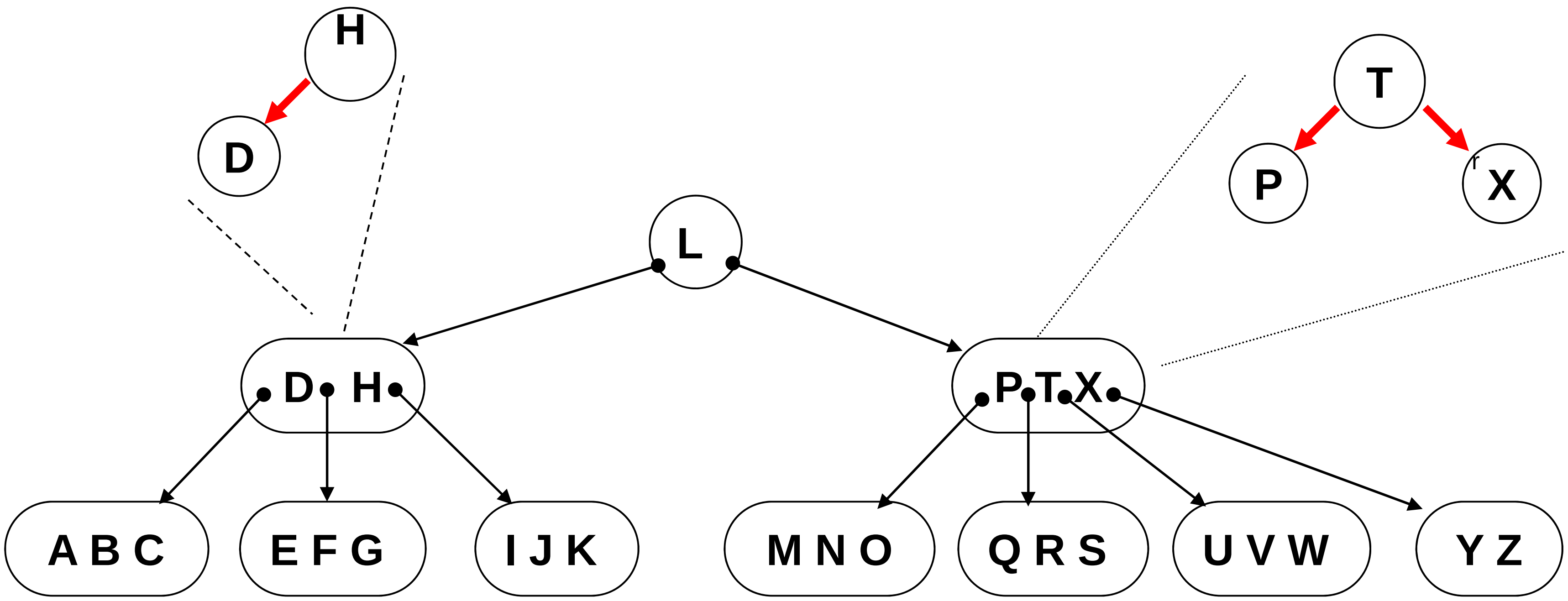
2-3-4 Bäume: B-Bäume mit maximal 4 Kindern

- Ein 2-3-4 Baum hat 2 oder 3 oder 4 Kinder (Ausnahme Wurzel)
- Binärbaum: ein Knoten hat maximal 2 Kinder
- 2-3-4 Bäume: ein Knoten hat maximal 4 Kinder
  - 3 Elemente (sortiert innerhalb Knoten)
  - 4 Zeiger auf Nachfolger



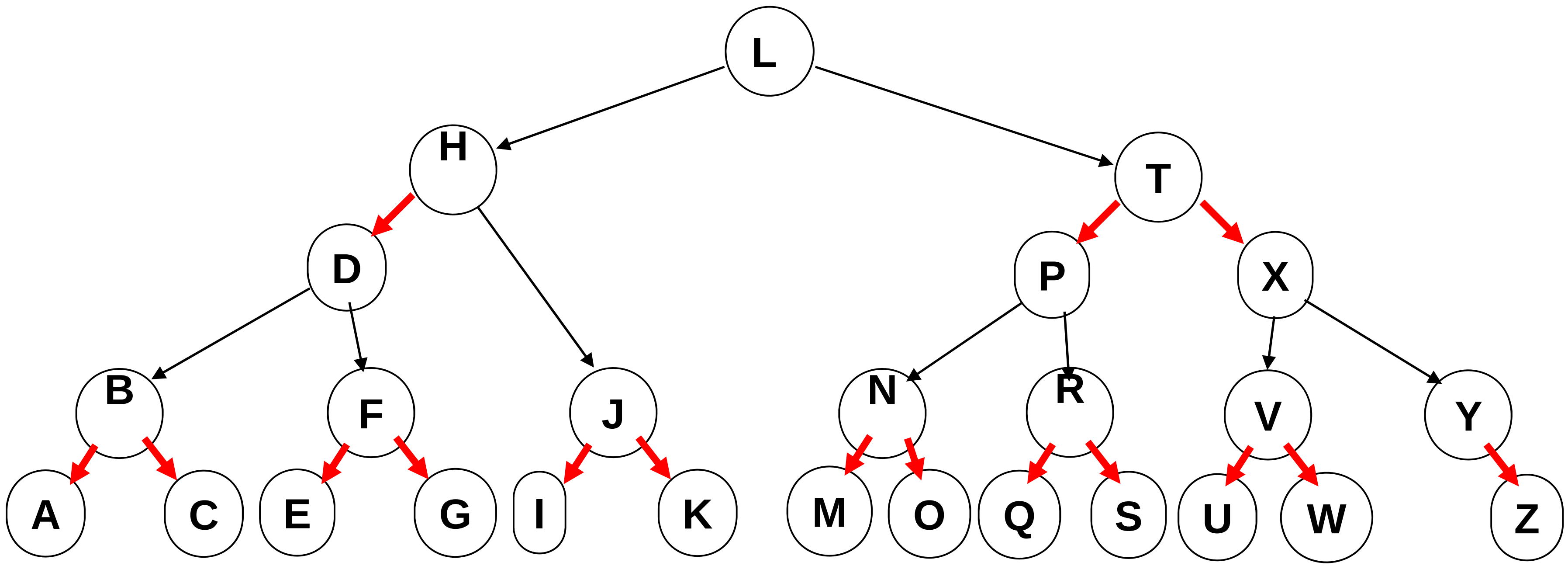
Rot-Schwarz Bäume

- Implementation von 2-3-4 Bäumen mit Binärbäumen
- 3er und 4er Knoten werden durch Binärbäume implementiert.



Rot-Schwarz Bäume

- Auf eine rote Kante muss immer eine schwarze Kante folgen
- Vorteile: Binärbäume (Einfachheit) + B-Bäume (Ausgeglichenheit)
- Weniger gut balanciert als AVL-Baum, aber Einfüge- und Lösch-operationen schneller



Zusammenfassung

- Sortierte Binärbäume
  - Einfügen
  - Suchen
  - Löschen
- Balancieren von Bäumen
  - Super-Balanciert: Gewicht von linkem und rechtem Teilbaum +/-1
  - AVL-Balanciert: Tiefe unterscheidet sich nur um +/-1
  - einfach und doppel-Rotationen
- B-Bäume
  - bis n Nachfolgeknoten
- Rot-Schwarz Bäume