Balancierte Binär- und B-Bäume

- Sie kennen die Kriterien um die Ausgeglichenheit von Binär-Bäumen zu bestimmen
- Sie können AVL Bäume implementieren
- Sie wissen, was B-Bäume und rot-schwarz Bäume sind und wie man sie einsetzt

Suchen und Tiefe

Suchen im Sortierten Binär Baum

Suche x im Baum B:

- **Wenn x == Wurzelelement gilt, haben wir x gefunden.**
- **Wenn x > Wurzelelement gilt, wird die Suche im rechten Teilbaum von B fortgesetzt, sonst im linken Teilbaum.**

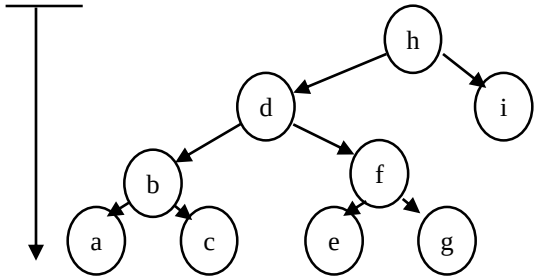
```
public Object search(TreeNode<T> node, T  x) {
  if (node == null) return node;
  else if (x.compareTo(node.element) == 0)
    return node;
  else if (x.compareTo(node.element) <= 0)
    return search(node.left,x);
  else
    return search(node.right,x);
}
```

- Bei einem vollen Binärbaum müssen lediglich ~ Log_2 #Elemente Schritte durchgeführt werden bis Element gefunden wird.
- Entspricht Aufwand für binäres Suchen
- sehr effizient Bsp: 1000 Elemente -> 10 Schritte; 1'000'000 -> 20 Schritte

Zugriffszeiten und Tiefe

- Die Zugriffszeit (Such- und Einfügezeit) von Elementen ist proportional zur Tiefe des Baumes

- Ziel: bei gegebener Anzahl Elemente einen Baum mit möglichst geringer Tiefe

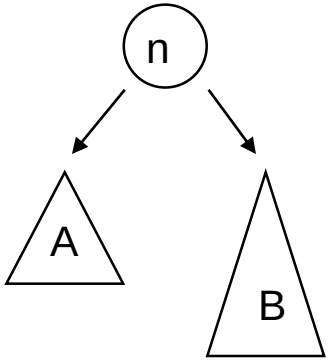


- Probleme:
  - neue Knoten können nur unten angehängt werden
  - einzufügende Elemente sind meist nicht à priori bekannt
  - bei "unglücklicher" Reihenfolge entstehen sehr ungleichmässige, d.h. "unbalancierte" Bäume

Balancieren

Balanciertheit von Bäumen

- bei einem vollen Baum sind alle bis auf die letzte Stufe voll gefüllt
- ein Binärbaum mit n Elementen hat im Minimum eine Tiefe von $\lceil \log_2(n+1) \rceil$
- wenn man Daten in beliebiger Reihenfolge in einen Binärbaum „naiv“ (wie bisher „einfach so“) einfügt, werden die beiden Teilbäume unterschiedlich schwer 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

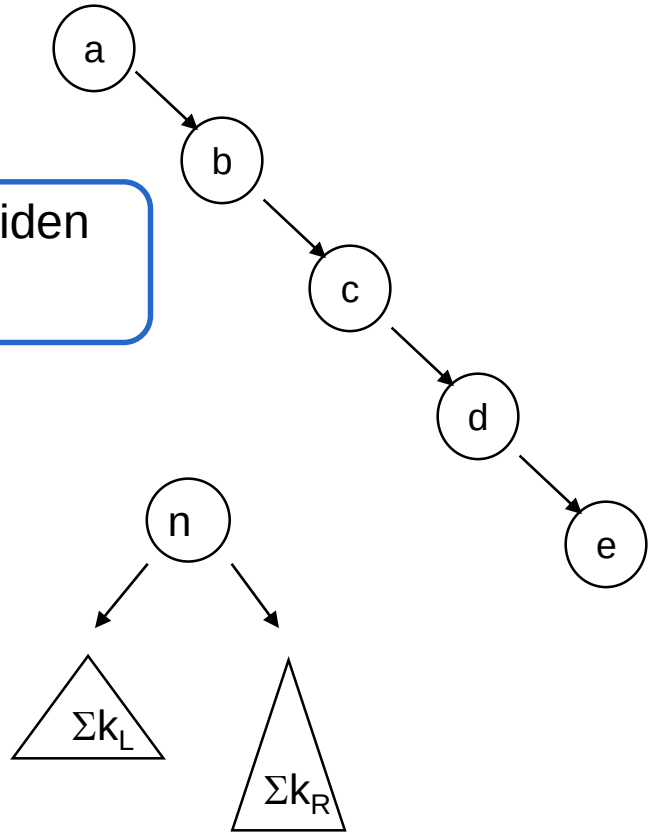
- Zeichnen Sie den sortieren Binärbaum auf, der beim Einfügen der Zeichenkette entsteht THEQUICKBROWN

- 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 möglich: z.B: Daten werden in sortierter Reihenfolge eingefügt
- Der Baum degeneriert zur Liste:

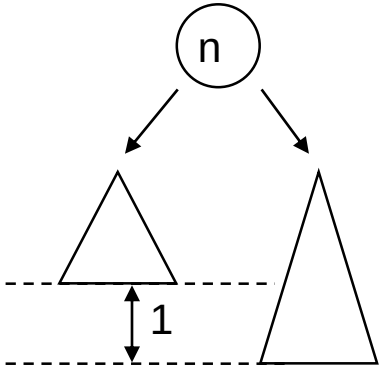


Vollständig ausgeglichen: die *Gewichte* der beiden Teilbäume unterscheiden sich maximal um 1

Analogie: "ein Mobile, das ausbalanciert ist"

Ausgeglichenheit von Bäumen: AVL-Bedingung

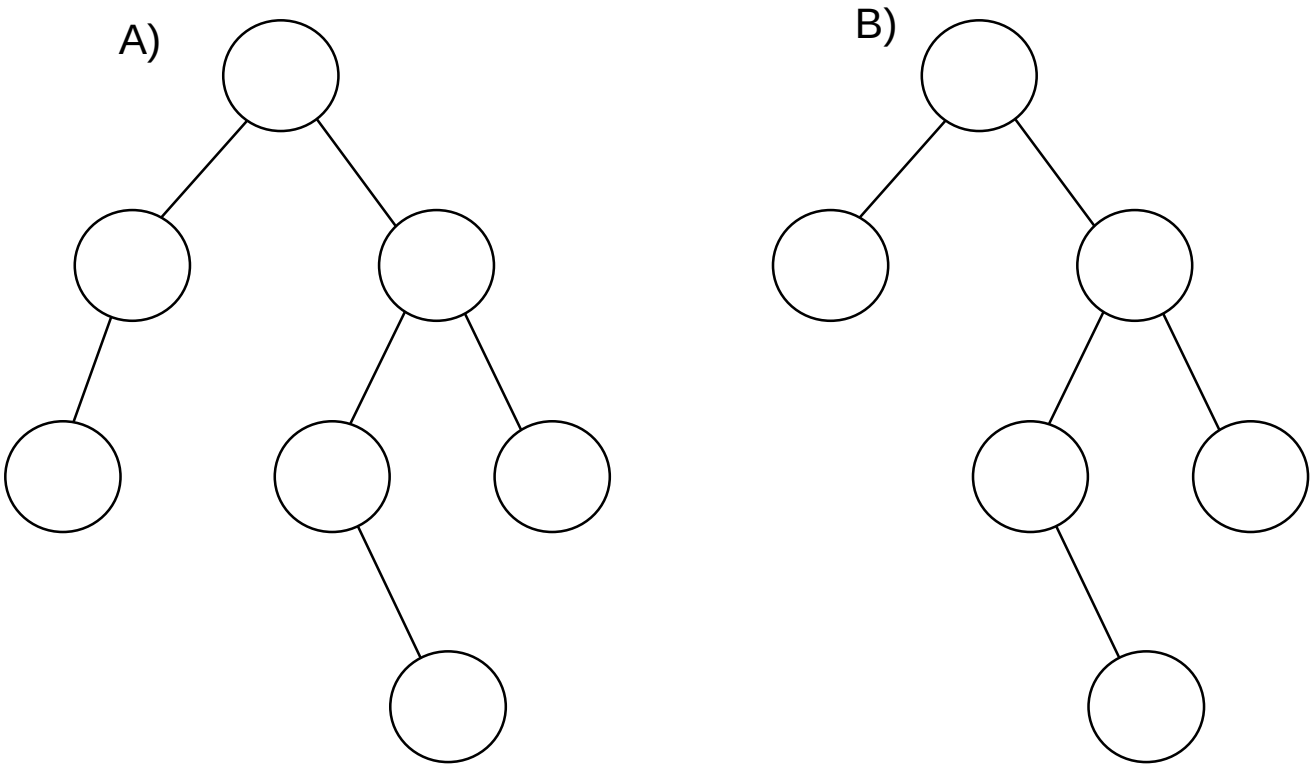
- AVL-Ausgeglichenheit: die *Tiefen* der beiden Teilbäume unterscheiden sich maximal um 1
- Beim Einfügen und Löschen sorgt man dafür, dass die AVL-Ausgleichbedingung erhalten bleibt.



- Vorteile
  - einfacher zu realisieren als Gewichtsbedingung
  - Degenierung zu einer Liste ist nicht möglich
  - Suchoperationen sind schnell: **O(log n)**
- Nachteile
  - zusätzlicher Aufwand bei der Programmierung, Einfügen und Löschen sind komplizierter

Aufgabe

- Erfüllt einer der beiden Bäume das Kriterium der AVL-Ausgeglichenheit? Wenn ja, welcher?

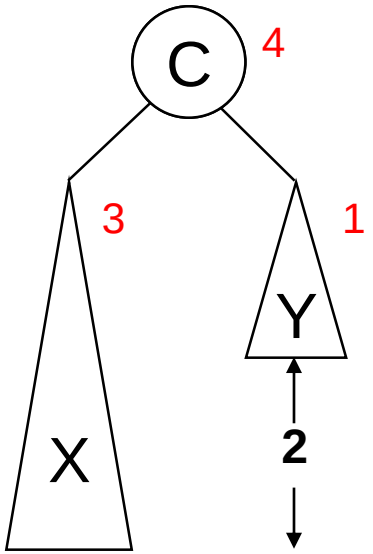


Operationen

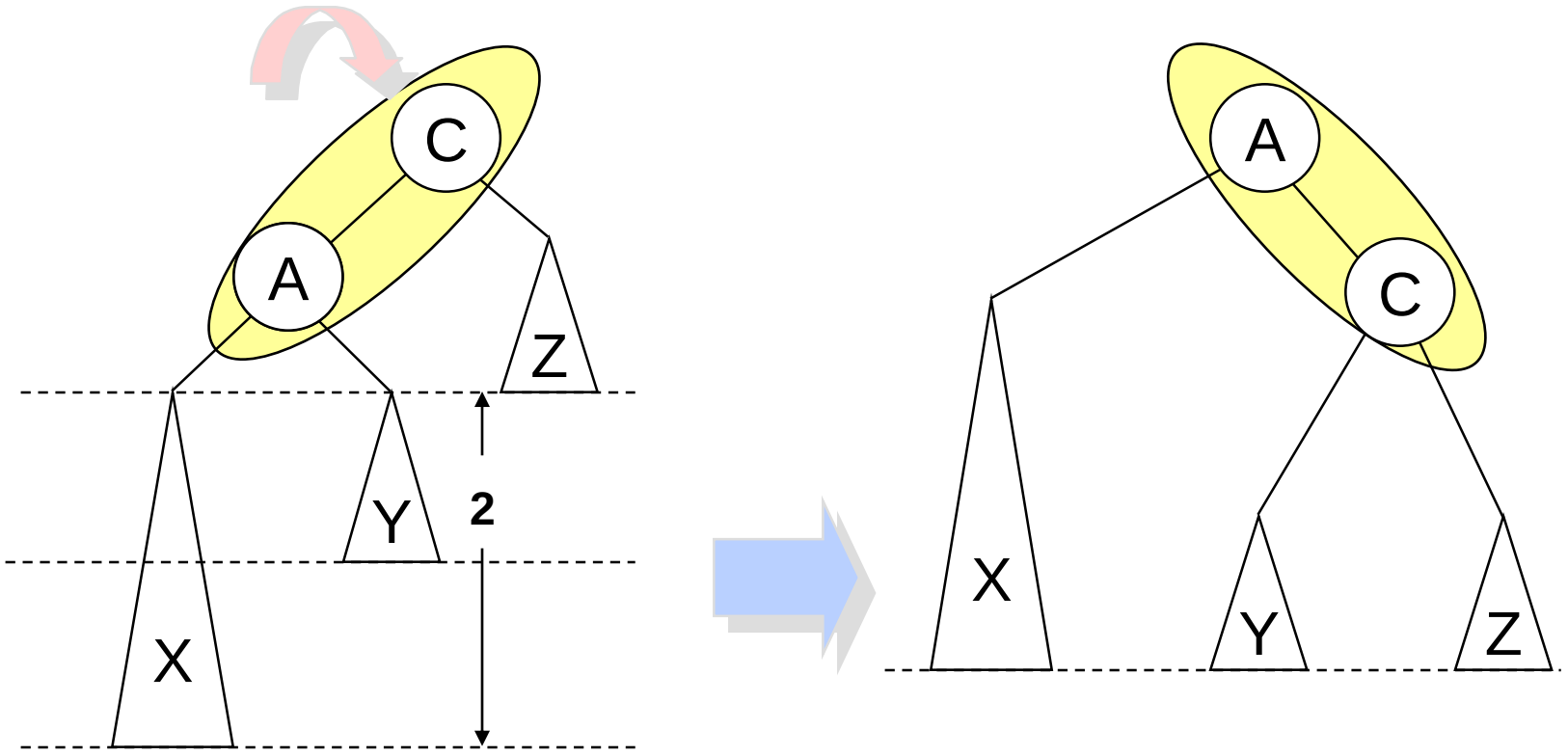
- **Suchen** und **Traversieren** unverändert (ist und bleibt ein Binärbaum)

- Bei allen **Einfüge- und Löschoperationen** wird sichergestellt, dass die AVL-Ausgleichsbedingung erhalten bleibt
  - es muss Buch geführt werden, wie tief die darunter gelegenen Teilbäume sind (pro Knoten eine Zahl)
  - wird die Differenz zwischen linkem und rechten Teilbaum > 1 muss etwas unternommen werden

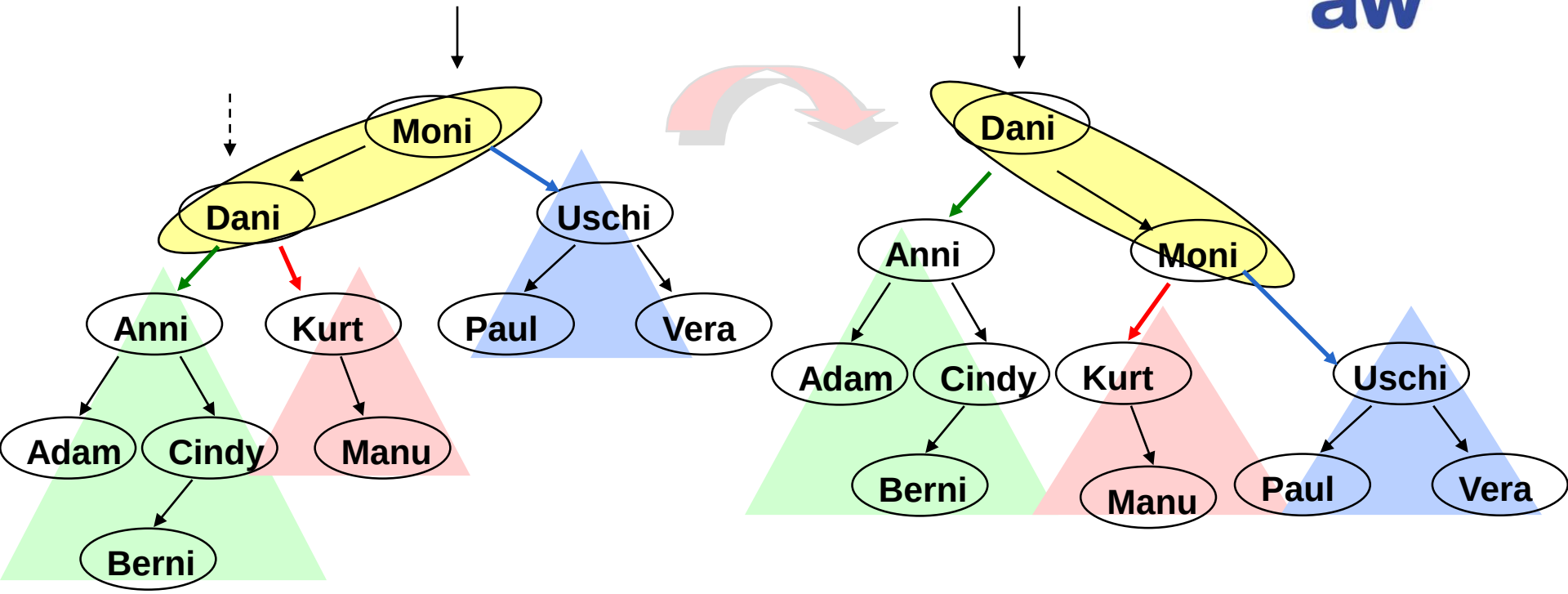
- Zum Wiederherstellen der Ausgleichsbedingung werden sog. **Rotationen** eingesetzt



Einzelrotation



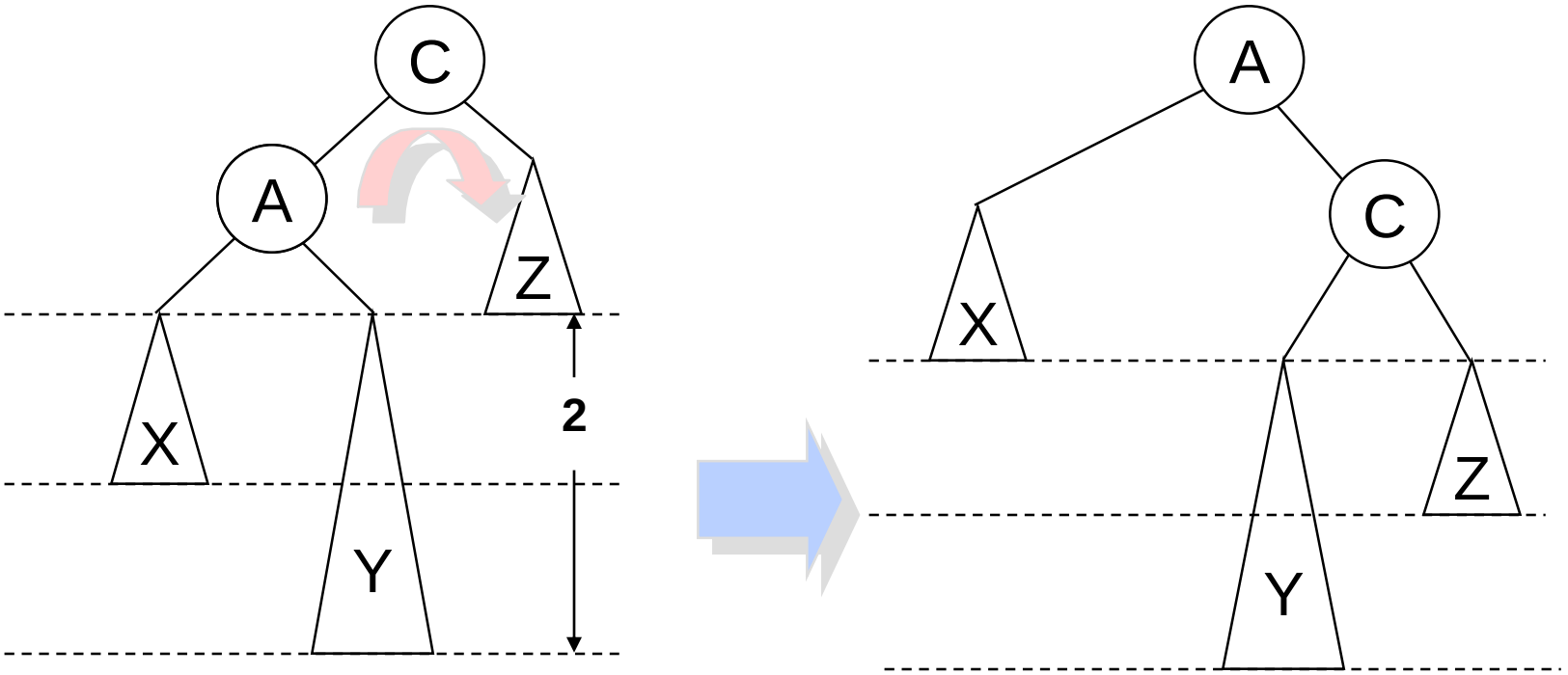
Einzelrotation Rechts, Beispiel



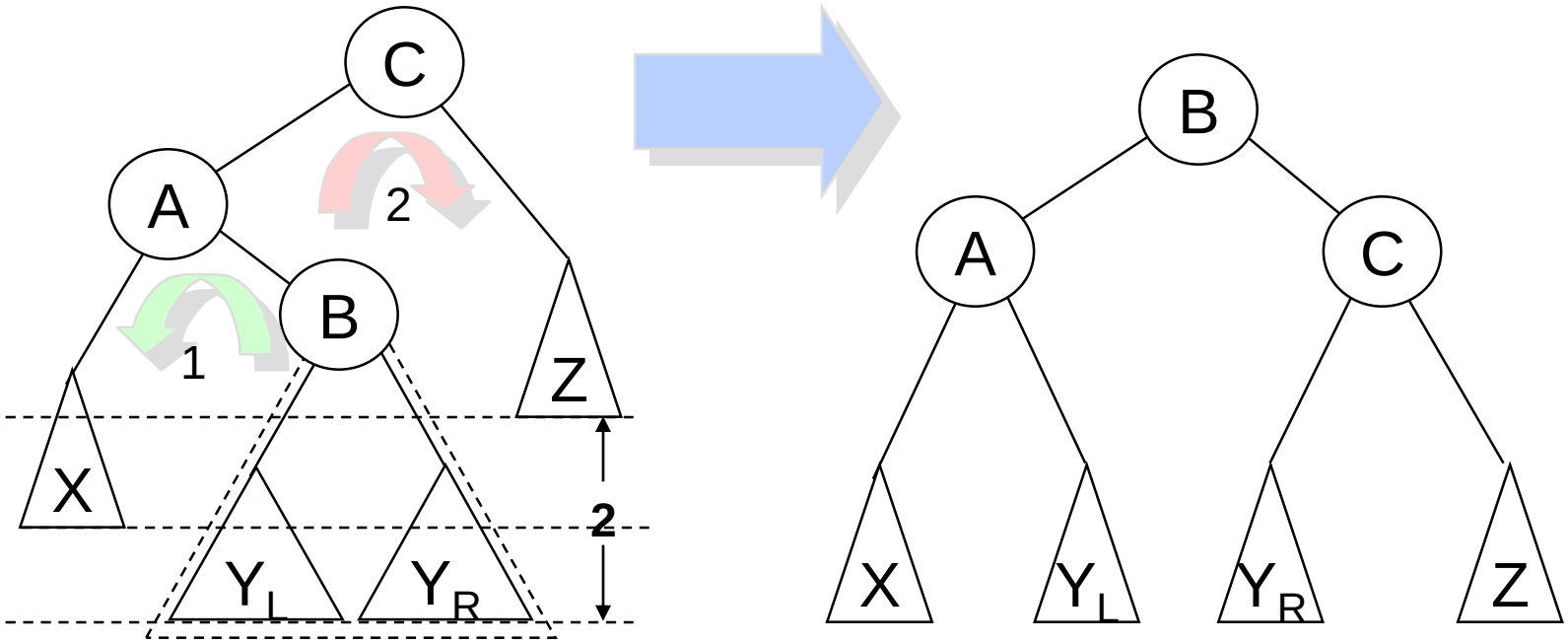

```
Node rotateR(Node p) {
    Node k = p.l;
    p.l = k.r;
    k.r = p;
    return k;
}
```

Problemfälle bei der Einzelrotation

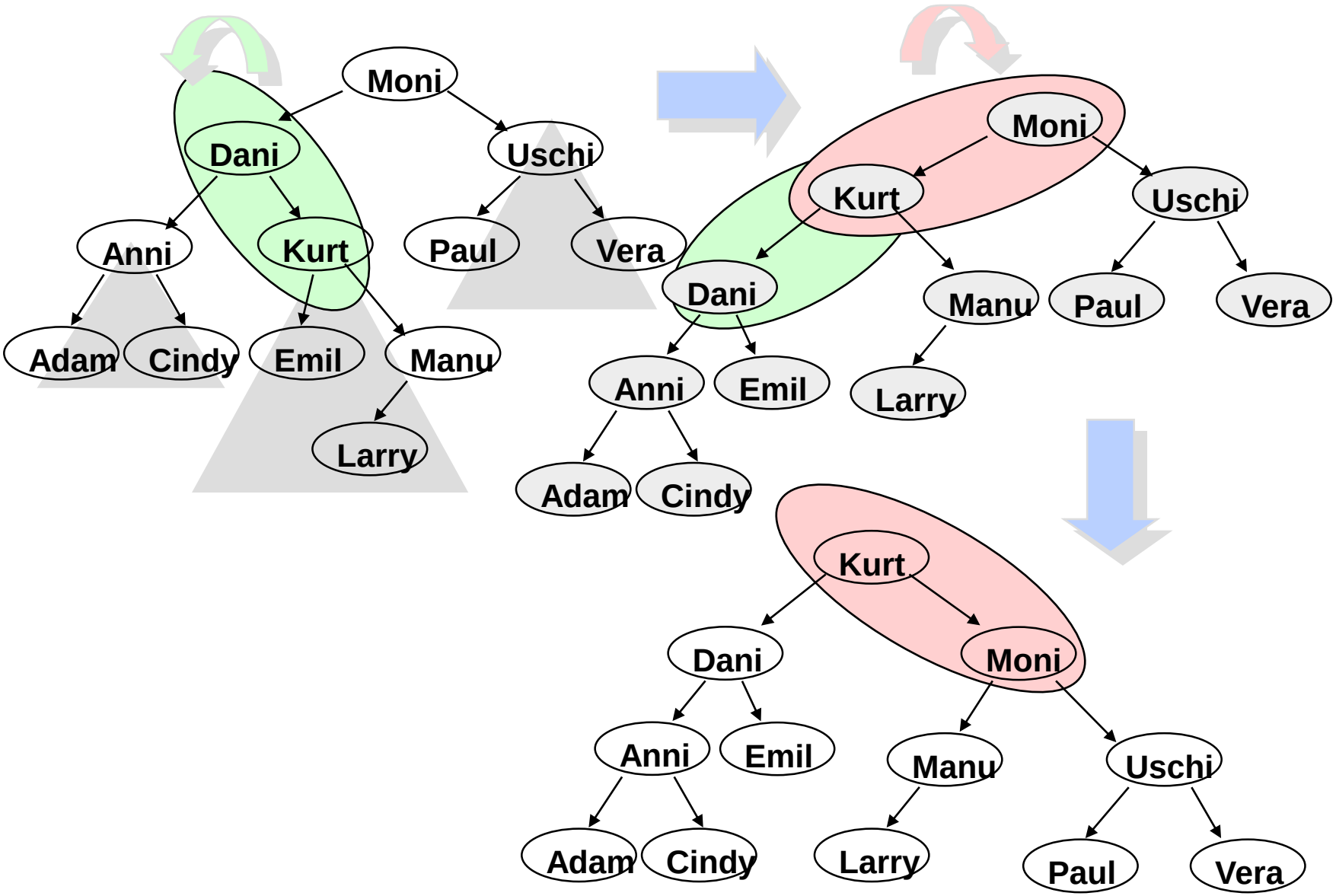
- kann nicht mit Einzelrotation balanciert werden:



Doppelrotation



Doppelrotation, Beispiel RotateLR



AVL Baum Implementation

- Knoten

```
class TreeNode<T extends Comparable<T>> {
T element;
TreeNode left, right;
int height;
int count;

TreeNode(T element){
     this.element = element;
     height = 1;
     count = 1;

}
TreeNode(T element, TreeNode left, TreeNode right){
     this(element); this.left = left; this.right = right;
}

T getValue(){return element;}
}
```

Höhe des Teilbaums

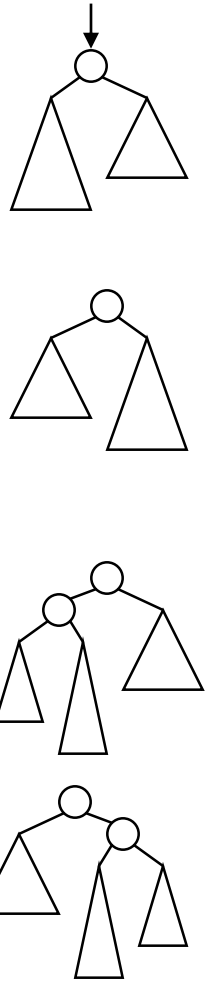
Duplikate

- AVL Baum

```
public class AVLSearchTree<T extends Comparable<T>> implements Tree<T> {
/** The tree root. */
private TreeNode root;

/**
* Return the height of node t, or 0, if null.
*/
private int height(TreeNode t) {
     return t == null ? 0 : t.height;
}
```

AVL Baum Implementation

- Rotation Methoden

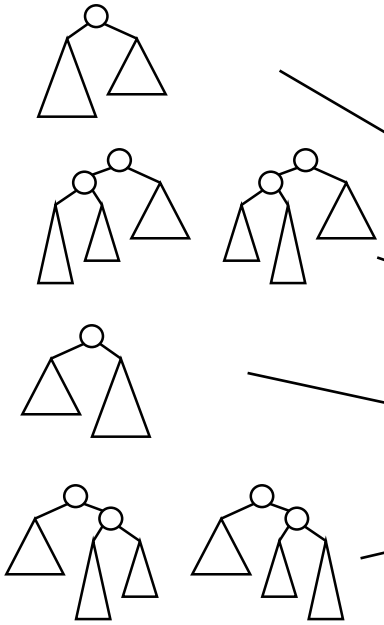


```
private static Node rotateR(Node k2) {
 Node k1 = k2.left;
 k2.left = k1.right;
 k1.right = k2;
 k2.height = Math.max(height(k2.left), height(k2.right)) + 1;
 k1.height = Math.max(height(k1.left), k2.height) + 1;
 return k1;
}

private static Node rotateL(Node k1) {
 Node k2 = k1.right;
 k1.right = k2.left;
 k2.left = k1;
 k1.height = Math.max(height(k1.left), height(k1.right)) + 1;
 k2.height = Math.max(height(k2.right), k1.height) + 1;
 return k2;
 }

private static Node rotateLR(Node k3) {
 k3.left = rotateL(k3.left);
 return rotateR(k3);
}

private static Node rotateRL(Node k1) {
 k1.right = rotateR(k1.right);
 return rotateL(k1);
}
```

AVL Baum Implementation

- Einfügen

- Balancieren



```
private TreeNode insertAt(TreeNode p, T element) {
  if (p == null) {
     return  new TreeNode<T>(element);
  } else {
     int c = element.compareTo((T) p.element);
     if (c == 0) {
        p.count++;
     } else if (c < 0) {
        p.left = insertAt(p.left, element);
     } else if (c > 0) {
        p.right = insertAt(p.right, element);
     }
  }
  return balance(p);
}

private TreeNode<T> balance(TreeNode<T> p) {
  if (p == null) return null;
  if (height(p.left) - height(p.right) == 2) {
     if (height(p.left.left) > height(p.left.right)) {
        p = rotateR(p);
     } else {

     }
  } else if (height(p.right) - height(p.left) == 2) {
     if (height(p.right.right) > height(p.right.left)) {

     } else {

     }
  }
  p.height = Math.max(height(p.left), height(p.right)) + 1;
  return p;
}
```

gleiche Werte dürfen nicht mehr einfach links eingefügt werden

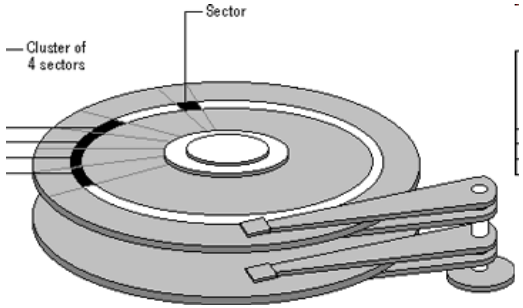
Zusammenfassung Binär-Bäume

- Binär-Bäume sind Bäume mit 2 Nachfolgern
- Sortierte Binär-Bäume (auch Suchbäume genannt) erfüllen zusätzlich das Kriterium K_{links} <= aktueller Knoten und K_{rechts} > aktueller Knoten
- Bei balancierten Bäumen werden Duplikate im Knoten "gezählt"
- Einfüge-/Lösch und Suchoperationen sind einfach und effizient: wachsen mit dem Log der Anzahl Knoten im Baum
- Die meisten Operationen können einfach rekursiv programmiert werden
- Zur Verhinderung, dass degenerierte Fälle entstehen, können Ausgleichsoperationen (Rotationen) angewandt werden
- Die Bedingung, dass der Höhenunterschied zwischen linkem und rechtem Teilbaum maximal 1 ist, wird als AVL Ausgeglichenheitsbedingung bezeichnet
- Diese führen aber dazu dass Mutationen (etwas) aufwendiger werden

B-Bäume

B-Bäume

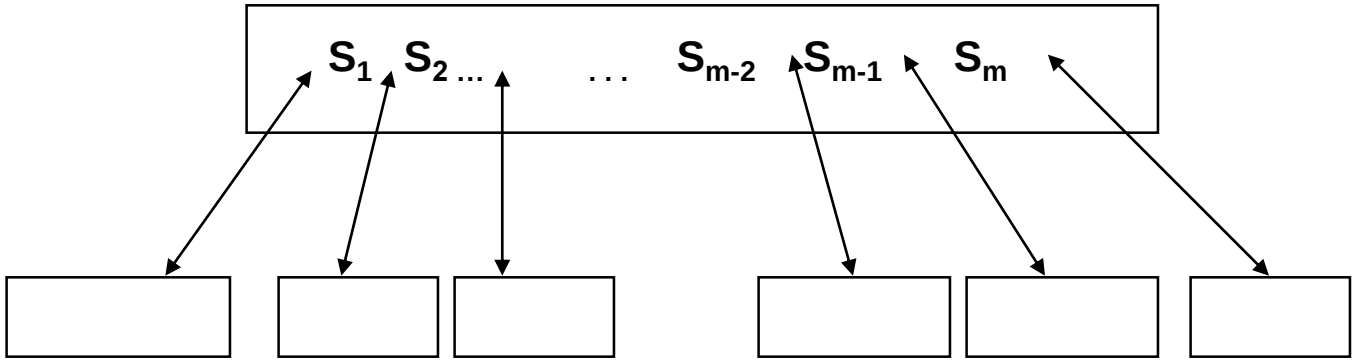
- Binär-Bäume eignen sich gut für Strukturen im Hauptspeicher
- Schlecht wenn Daten auf Disk liegen:
  - Viele wahlfreie (random) Zugriffe auf Daten
  - Random Access für Disk ist sehr teuer (Kopfpositionierzeit + Lesezeit ca. ~10ms) Vergleich: Hauptspeicher Zugriff ~10 ns



- Idee: Baum so aufbauen, dass die wahlfreien Zugriffe auf Disk Blöcke minimiert wird.
  - möglichst viel Information pro Diskblock
  - möglichst breiter Baum, damit geringe Tiefe ~ Zugriffe
  - alle Knoten gleich gross (z.B. 1024 Bytes)
  - Baum ist immer ausgeglichen

B-Bäume: Bäume mit n Nachfolgern

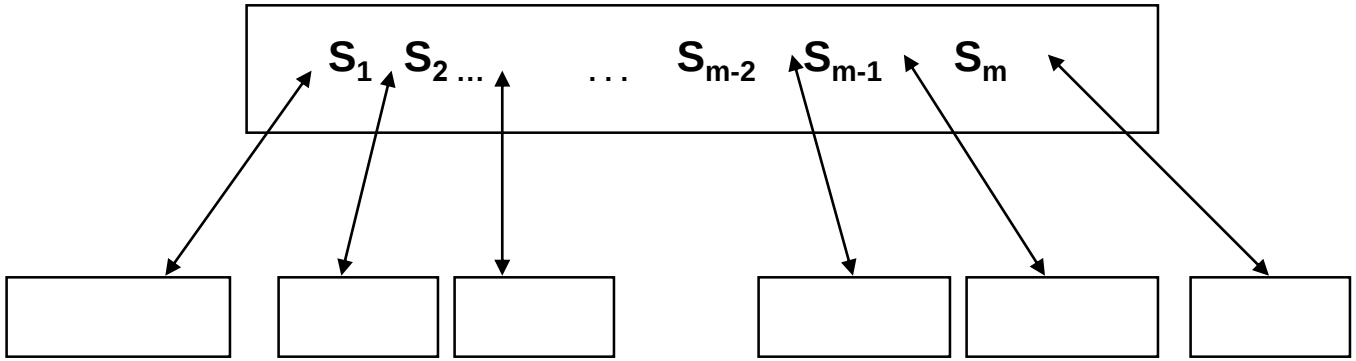
- "B" steht für den Erfinder Rudolf Bayer (*1939)
- B-Bäume werden bei der Konstruktion (Einfügen und Löschen) automatisch balanciert.
- In einem B-Baum der Ordnung n enthält jeder Knoten ausser der Wurzel mindestens n/2 und höchstens n Schlüssel.
- Jeder Knoten ist entweder ein Blatt oder hat m+1 Nachfolger, wobei m die Anzahl Schlüssel des Knotens ist.



…B-Bäume: Bäume mit maximal n Nachfolgern

- Für die Schlüssel und Verweise gilt:
  - innerhalb eines Knotens sind alle Schlüssel sortiert
  - alle Schlüssel im j–1-ten Nachfolgerknoten sind kleiner,
  - alle Schlüssel im j-ten Nachfolgerknoten sind grösser oder gleich

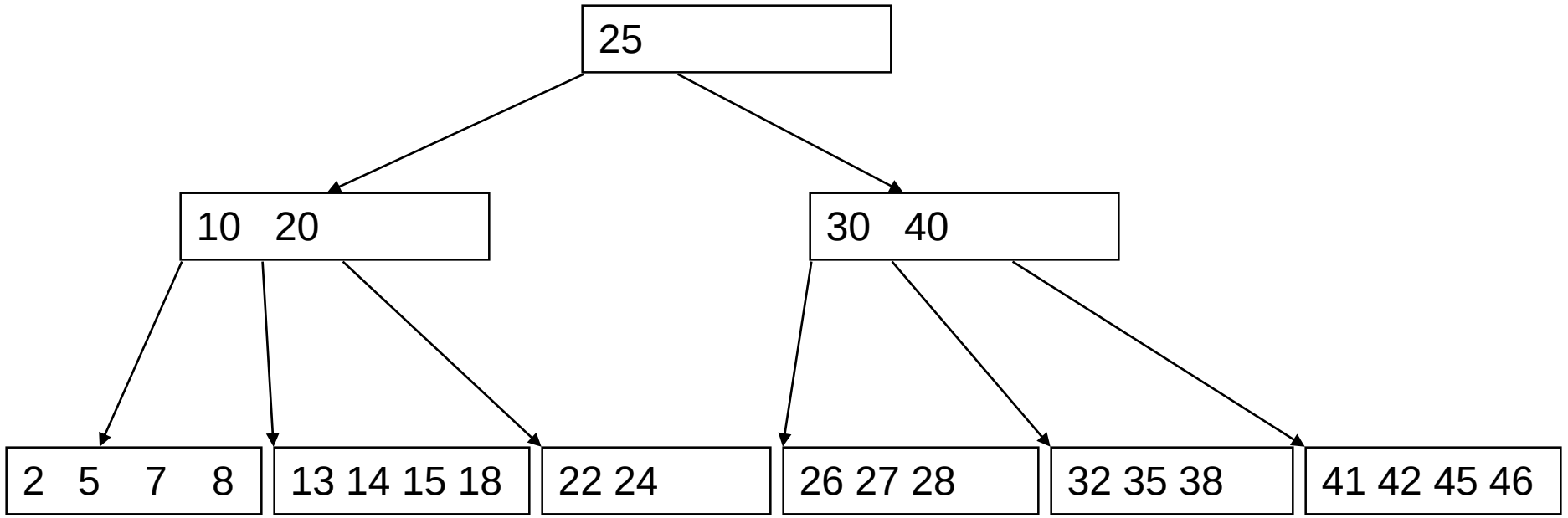
  dem Schlüssel s_j
- Alle Blätter liegen auf derselben Stufe
- Die inneren Knoten enthalten Schlüssel und Verweise. Die Blätter enthalten die Information.



Anwendungen von B-Bäumen:

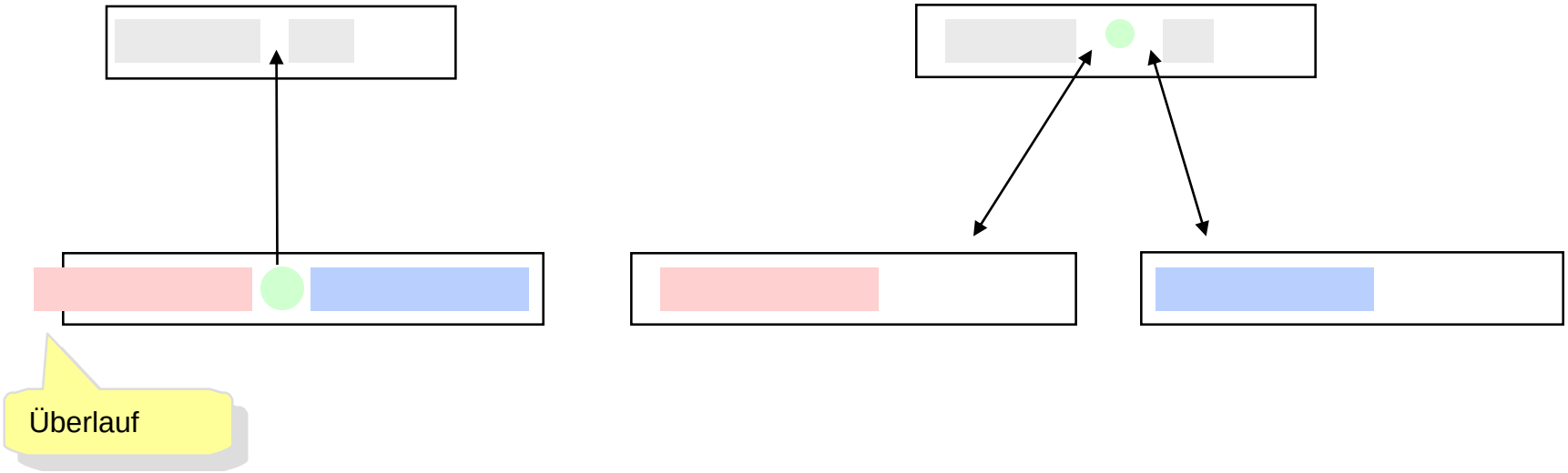
- Organisation der Daten auf Disk mit fester Blockgrösse, z.B. Windows NTFS-Filesystem.
- Ein Diskblock (Cluster) kann entweder Daten enthalten oder n-1 Schlüssel und n Verweise (auf Nachfolgerknoten). n wird so gewählt, dass die Diskblöcke optimal ausgenützt werden.
- mit wenigen Diskzugriffen kann ein bestimmter Datensatz gefunden werden. O(log N)
- Indexe in Datenbanken

Beispiel eines B-Baums mit max 4 Schlüsseln

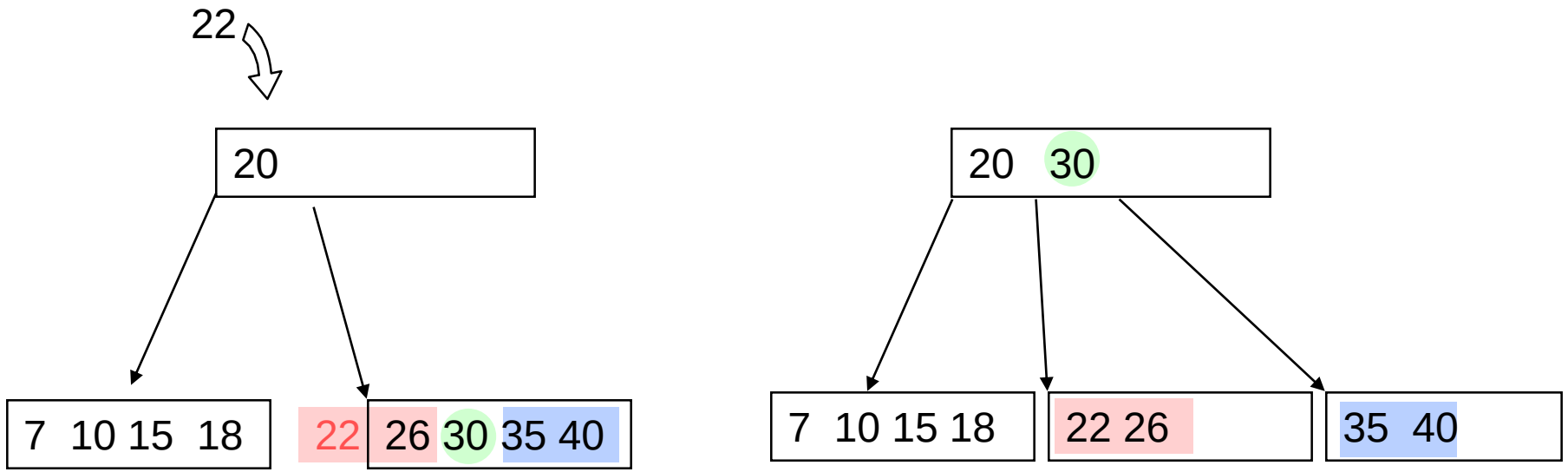


Einfügen

- Eingefügt wird immer in den Blättern.
- Einfügen innerhalb Knoten (solange Platz),
- sobald dieser voll : Überlauf
  - Aufteilen in zwei Knoten und "heraufziehen" des mittleren Elements in den Vaterknoten
  - falls dieser überläuft: gleich verfahren



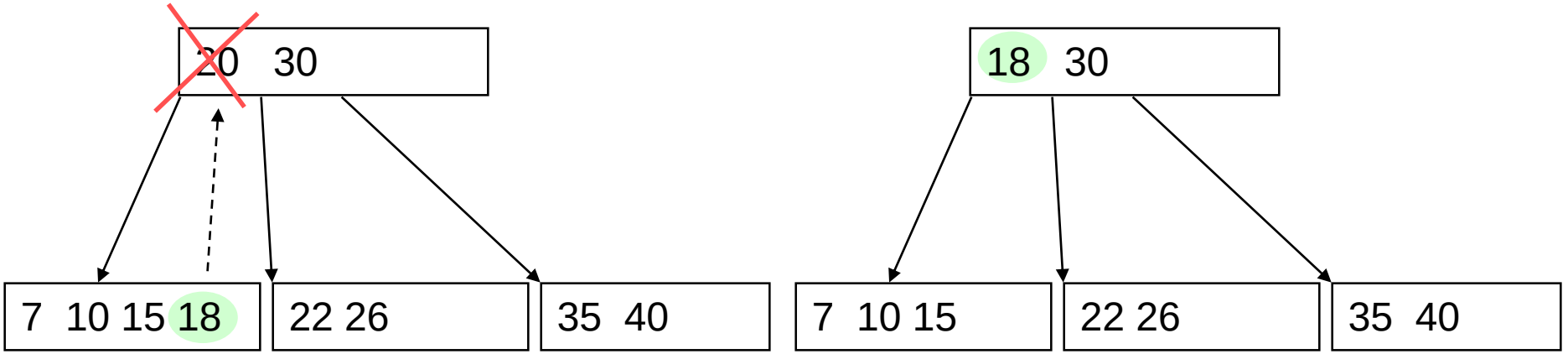
Beispiel einer Einfügeoperation: Aufteilen



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

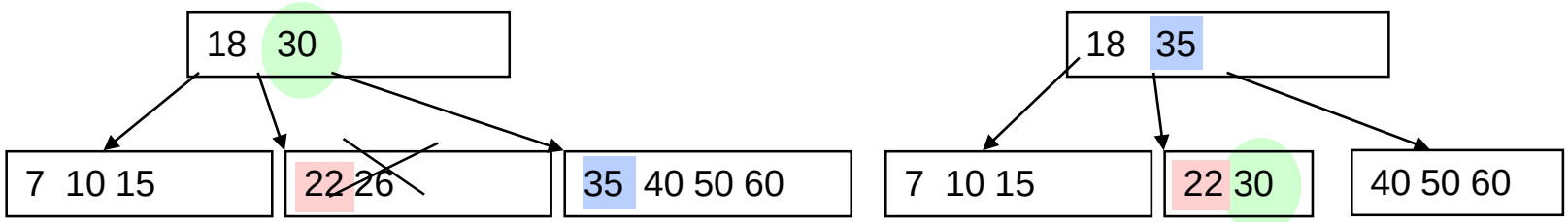
- Zu löschendes Element ist in einem Blatt
- Zu löschendes Element ist einem inneren Knoten ✓
  - gleich verfahren wie bei Binärbaum: Ersatzwert suchen



Löschen: Verschmelzen von Knoten

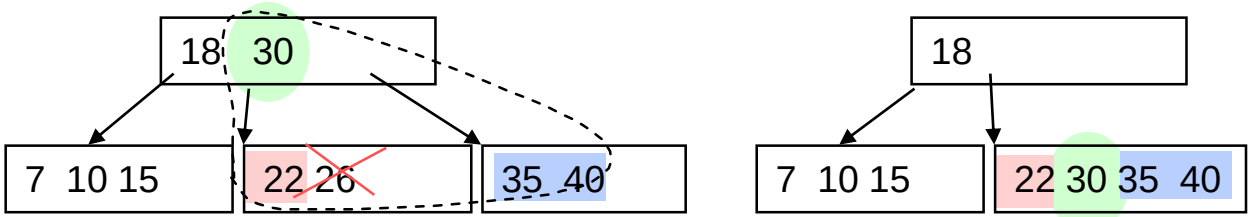
Unterlauf: Ein Knoten enthält weniger als n/2 Schlüssel.

- "Ausleihen" bei einem Nachbarknoten



oder

- Zwei benachbarte Knoten können zu einem zusammengefasst



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 $\lceil \log_{\text{Anzahl Verweise}} \text{Anzahl Elemente} \rceil$

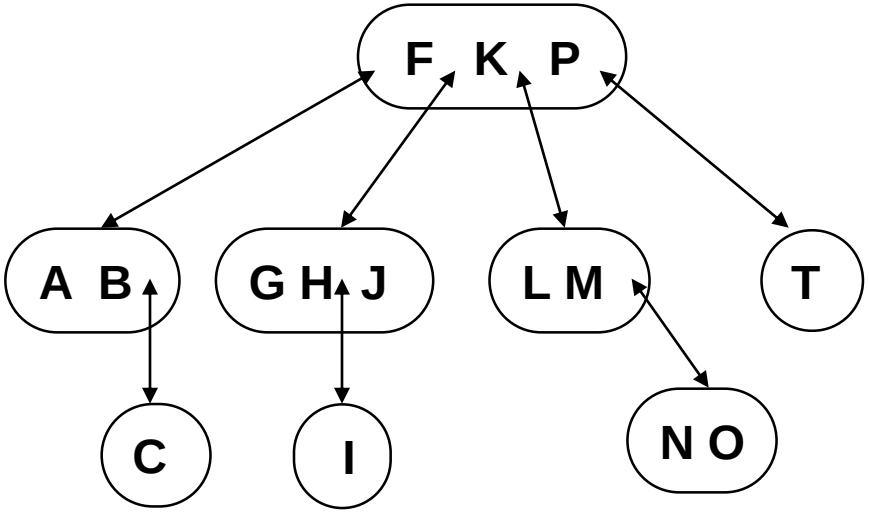
Anzahl Zugriffe: proportional zu Tiefe des Baumes

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

Rot-Schwarz Bäume

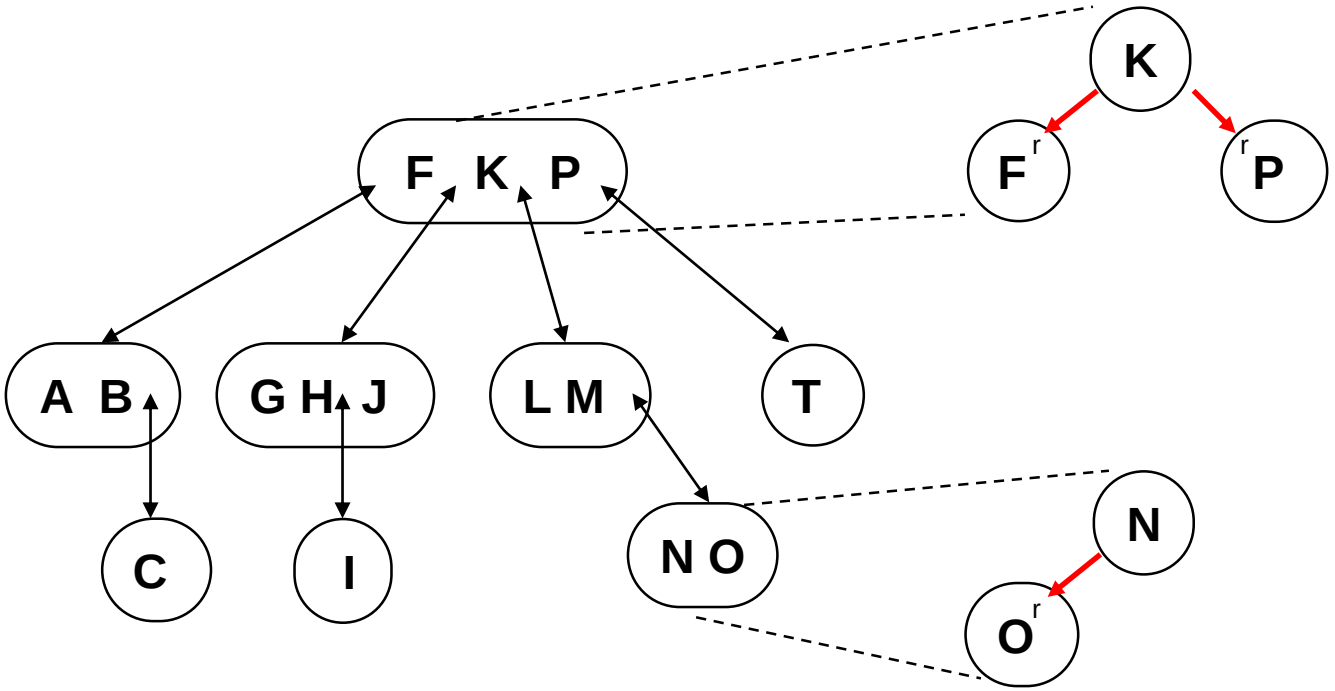
B-Bäume im Hauptspeicher

- B-Bäume mit maximal 4 Nachfolgern; 2-3-4 Bäume (= Ordnung 4)



Rot-Schwarz Bäume

- Grosse Knoten werden durch Binärbäume mit "roten" Kanten/Knoten implementiert
  - man spricht deshalb auch von roten Knoten/Kanten
- Ausgleichverfahren so dass Rot-Schwarz Bedingungen erhalten bleiben



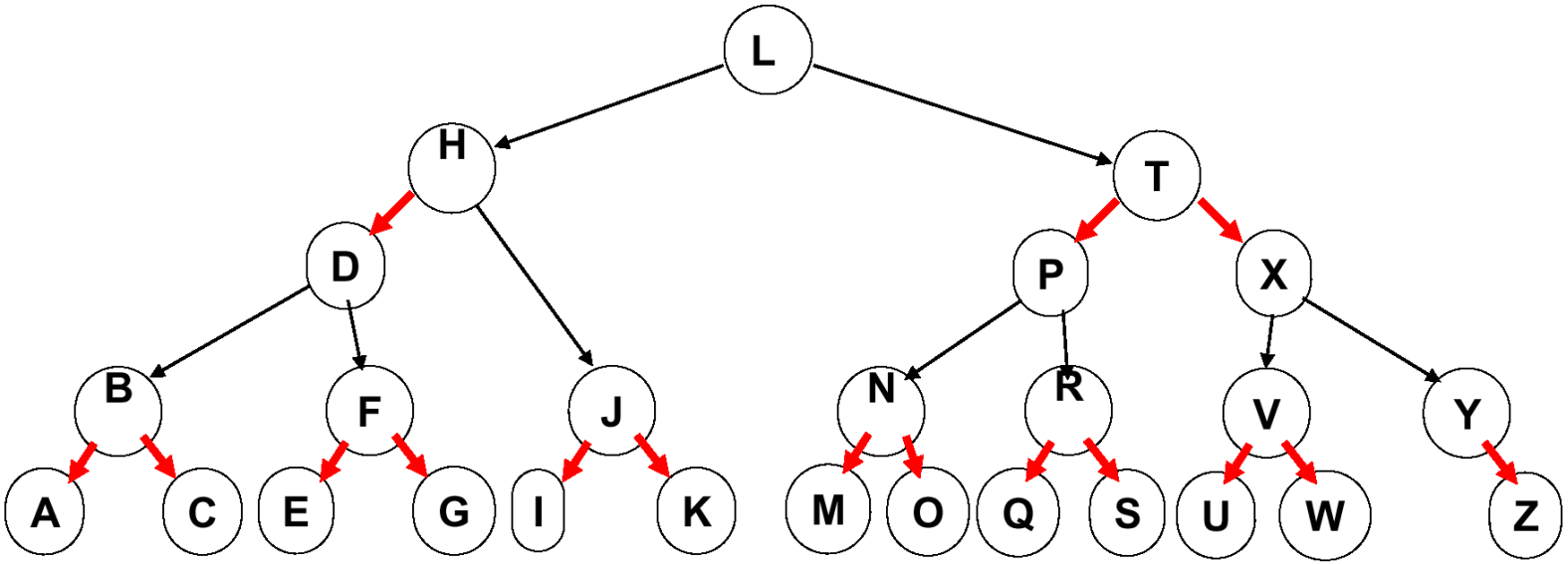
Rot-Schwarz Bedingung

1. Jeder Knoten im Baum ist entweder rot oder schwarz.
2. Die Wurzel des Baums ist schwarz.
3. Jedes Blatt (NIL-Knoten) ist schwarz.
4. Kein roter Knoten hat ein rotes Kind.
5. Jeder Pfad, von einem gegebenen Knoten zu seinen Blattknoten, enthält die gleiche Anzahl schwarzer Knoten (Schwarzhöhe/Schwarztiefe).

- B-Baum Ausgeglichenheit
  - Für die Tiefe zählt man nur die Schwarzen Knoten -> Max 2* Knoten

Rot-Schwarz Eigenschaften

- Auf eine rote Kante muss immer eine schwarze Kante folgen
- Vorteil: Einfachheit von Binärbäumen und Ausgeglichenheit von B-Bäumen
- Weniger gut balanciert als AVL Baum, aber Einfüge und Löschoperationen sind schneller.



Eigenschaften von Baumtypen

- Binärbaum
  - allgemeine Grundform, einfach in der Implementierung
- B-Baum:
  - Optimiert für Massenspeicherzugriff. Es gibt mindestens n/2 Unterbäume. Dadurch wird der Baum weniger hoch
    ⇒ weniger Plattenzugriffe notwendig
- 2-3-4 Baum
  - B-Baum mit max. 4 Nachfolgern
- Rot-Schwarz Baum:
  - Durch "Färben" der Kanten/Kanten kann 2-3-4 Baum als Binärbaum implementiert werden: java.util.TreeMap
- Splay-Tree (nicht behandelt):
  - Knoten, auf welche oft ein Zugriff erfolgt, werden zum root-Knoten rotiert.
  - Zugriff wird schneller (self-adjusting strategy)

Zusammenfassung

- 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
- 2-3-4 Bäume (Spezialfall eines B-Baums mit n=4)
  - bis zu 4 Nachfolgeknoten
  - rot-schwarz Bäume 2-3-4 Bäume als Binärbäume