Teil 1:
Suchen

- Problemstellung
- Elementare Suchverfahren
- Hashverfahren
- Binäre Suchbäume
- Ausgeglichene Bäume
  - AVL-Bäume
  - Splay-Bäume
- B-Bäume
- Digitale Suchbäume
- Heaps

Definition von AVL-Bäumen

Ziel

Ausgeglichene Bäume: Suchbäume mit einer maximalen Höhe von O(log n).

Damit könnten alle Dictionary-Operationen in O(log n) ausgeführt werden.

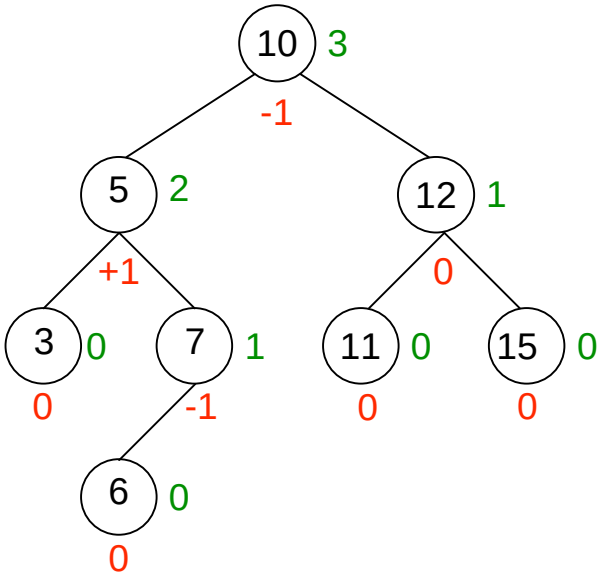
Definition AVL-Baum

Ein binärer Suchbaum heißt AVL-Baum oder (höhen-)balancierter Baum, falls sich für jeden Knoten k die Höhen der beiden Teilbäume um höchstens 1 unterscheiden.

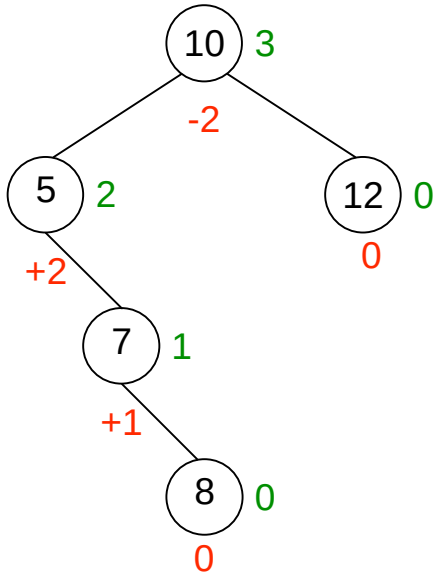
Die Abkürzung AVL geht zurück auf Adelson-Velskij und Landis.

Ein AVL-Baum hat eine maximale Höhe von etwa 1.5 $\log_2 n$. [Ottmann u. Widmayer].

Beispiel für AVL-Baum:



Beispiel für Nicht-AVL-Baum



Höhe des Teilbaums

Höhenunterschied
= Höhe rechter Teilbaum
– Höhe linker Teilbaum

(Beachte: Höhe eines leeren Teilbaums ist -1.)

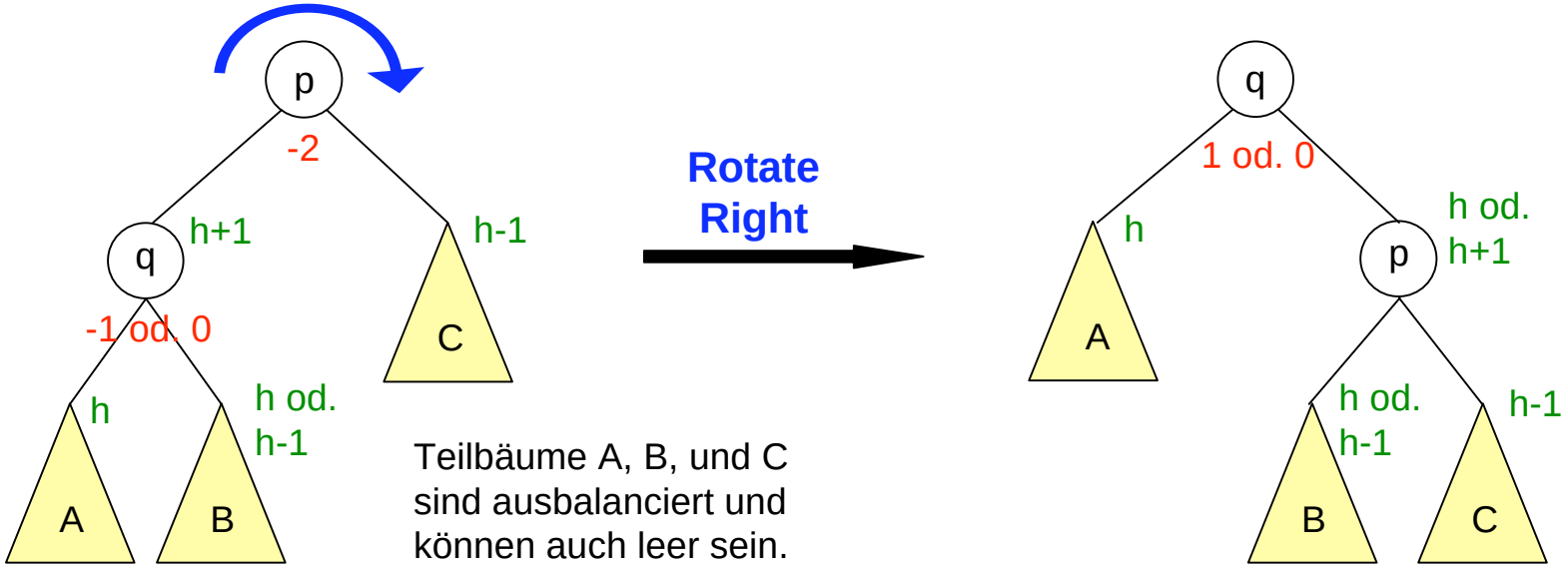
Rotationen (1)

Zentrale Eigenschaft von AVL-Bäumen:

Ein unbalancierter Baum, der einen Höhenunterschied von -2 (linkslastiger Baum) oder +2 (rechtslastiger Baum) hat und dessen Teilbäume ausbalanciert sind, lassen sich durch eine einfache Rotationsoperation ausbalancieren.

Fall A: Baum ist linkslastig, d.h. Höhenunterschied = -2

Unterfall A1: linker Teilbaum hat Höhenunterschied -1 oder 0:

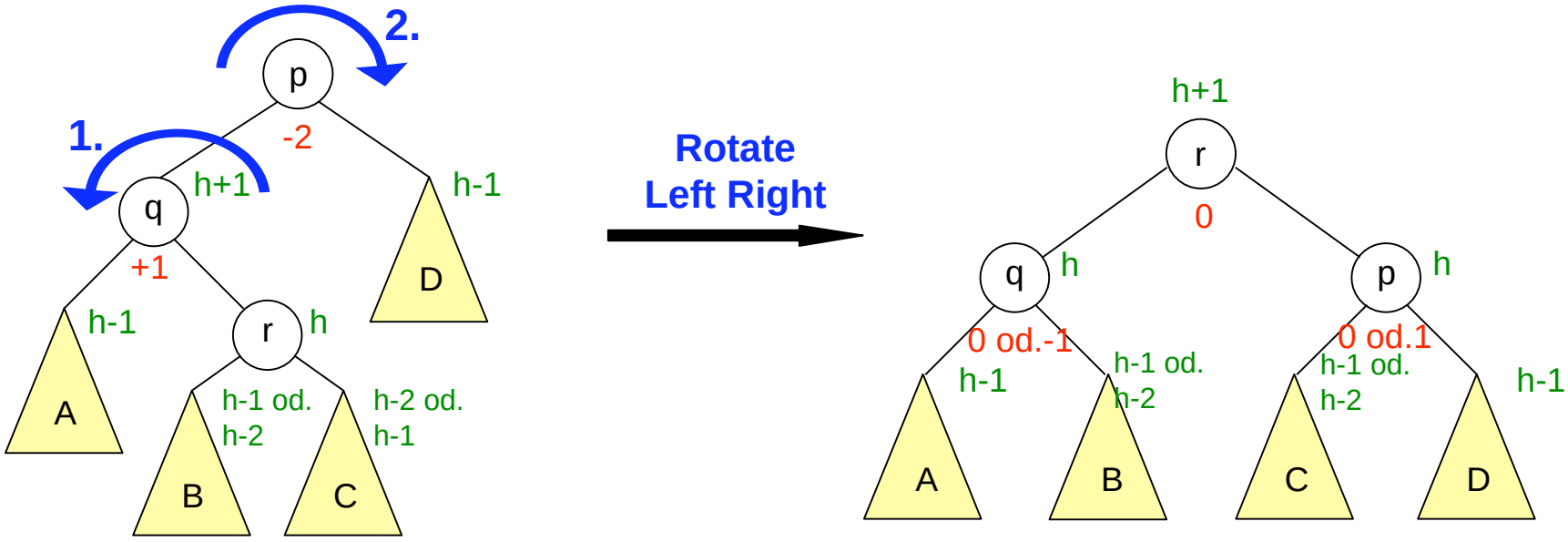


Teilbäume A, B, und C sind ausbalanciert und können auch leer sein.

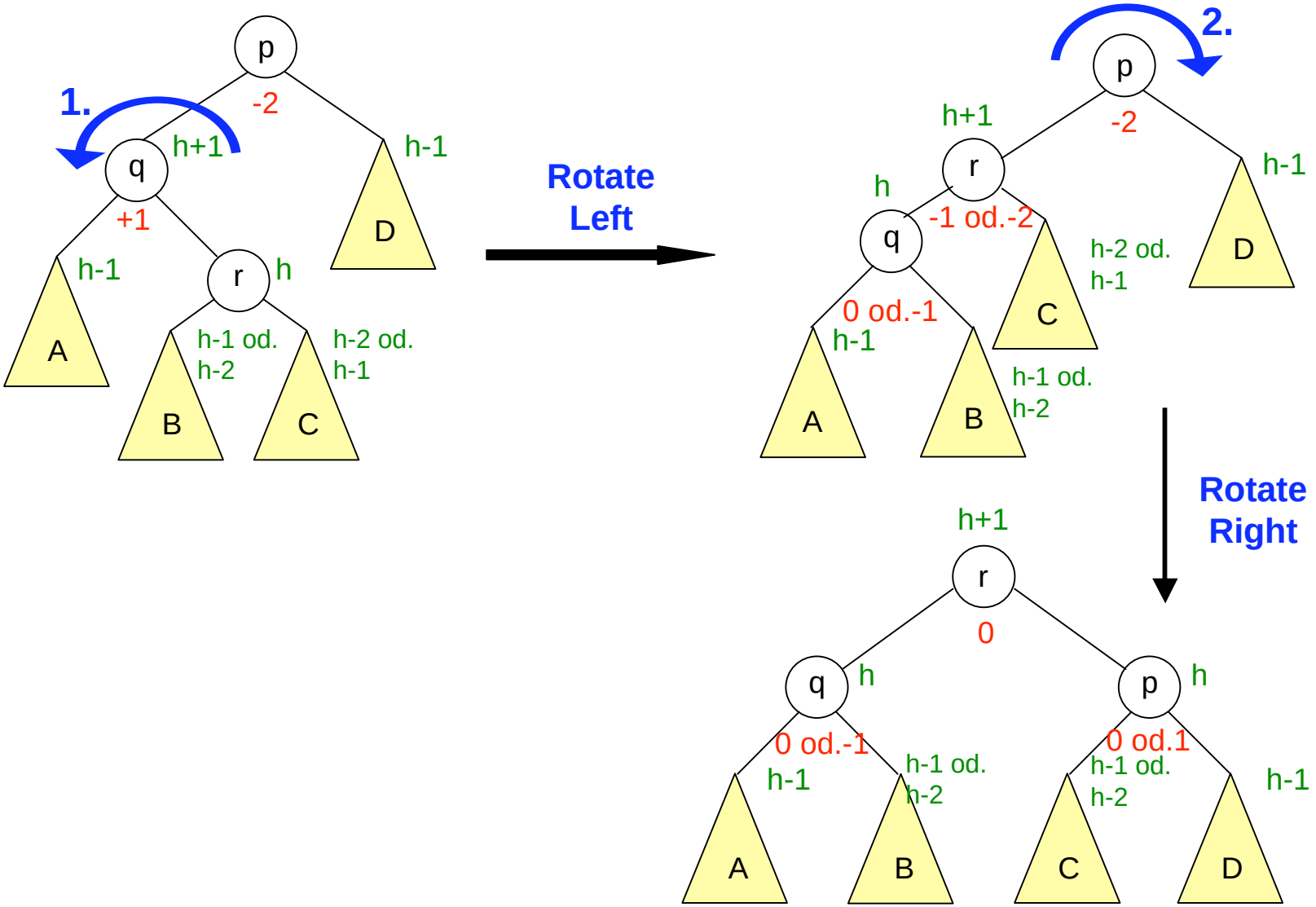
Rotationen (2)

Fall A: Baum ist linkslastig, d.h. Höhenunterschied = -2

Unterfall A2: linker Teilbaum hat Höhenunterschied +1:



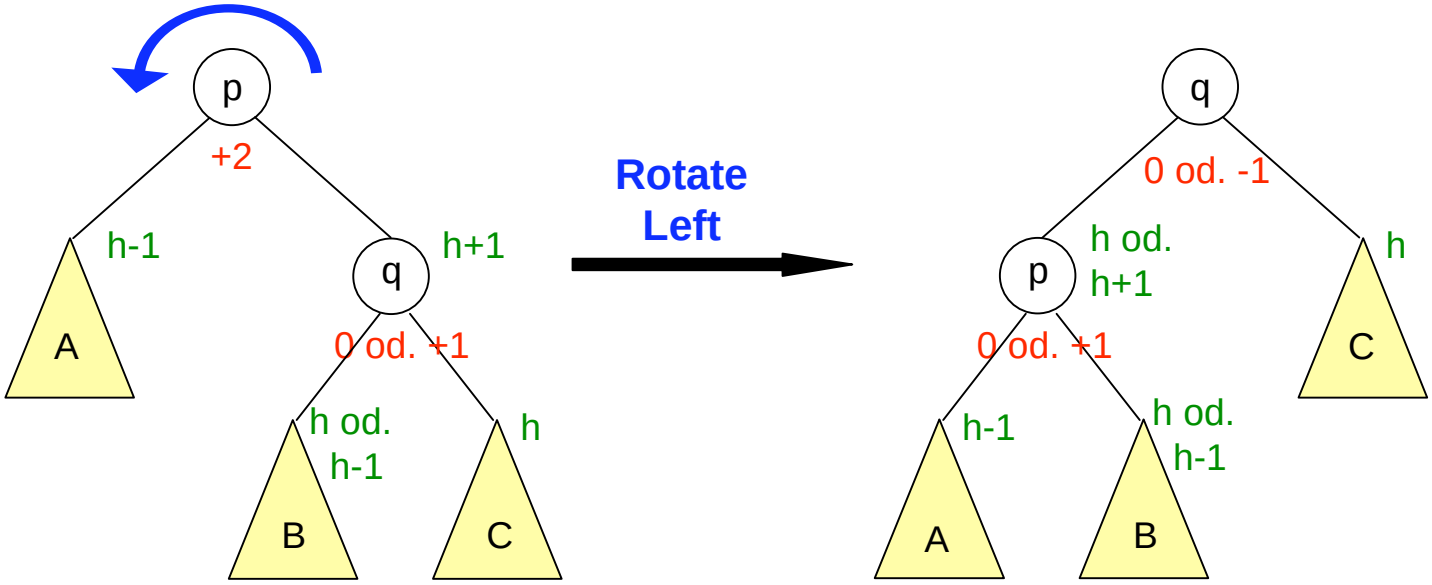
Rotationen (2b)



Rotationen (3)

Fall B: Baum ist rechtslastig, d.h. Höhenunterschied = +2

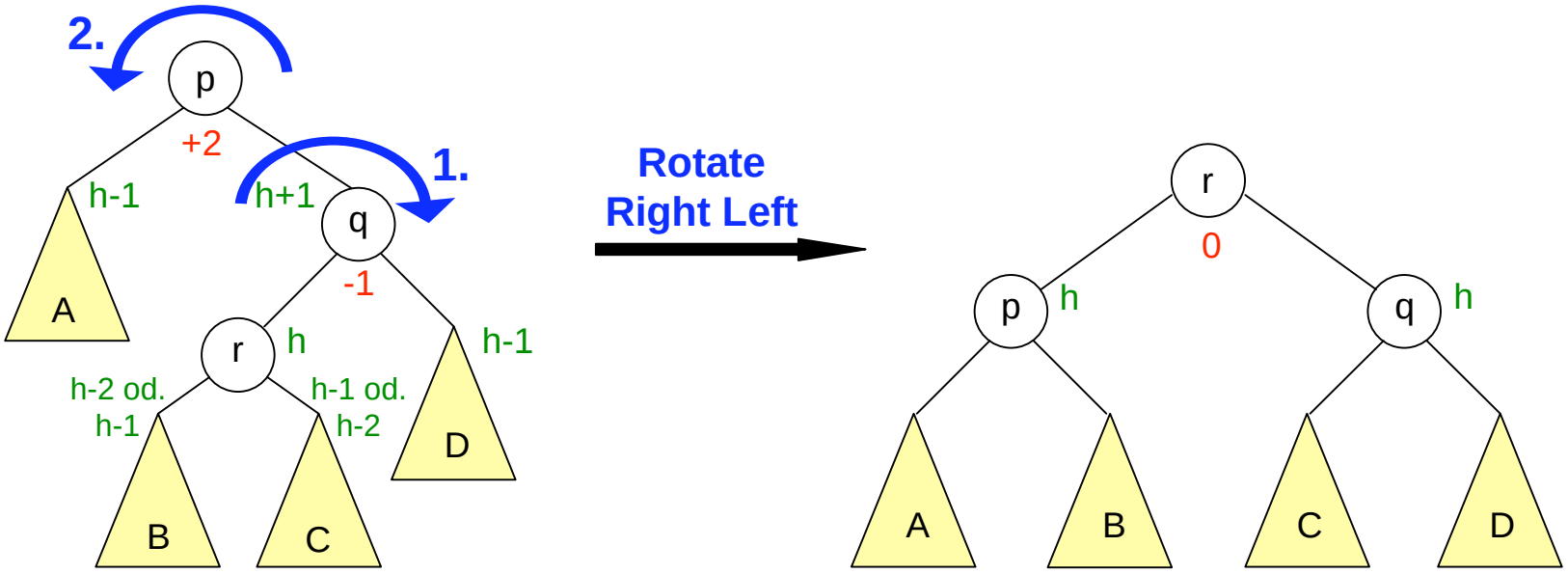
Unterfall B1: rechter Teilbaum hat Höhenunterschied 0 oder +1:



Rotationen (4)

Fall B: Baum ist rechtslastig, d.h. Höhenunterschied = +2

Unterfall B2: rechter Teilbaum hat Höhenunterschied -1:



Einfügen und Löschen in AVL-Bäumen

Idee für Algorithmus:

1) Füge ein bzw. lösche wie bei binären Suchbäumen.

2) Gehe dann von der Einfügestelle bzw. Löschstelle bis zur Wurzel
und balanciere falls notwendig mit einer Rotationsoperation lokal aus.

Bemerkungen:

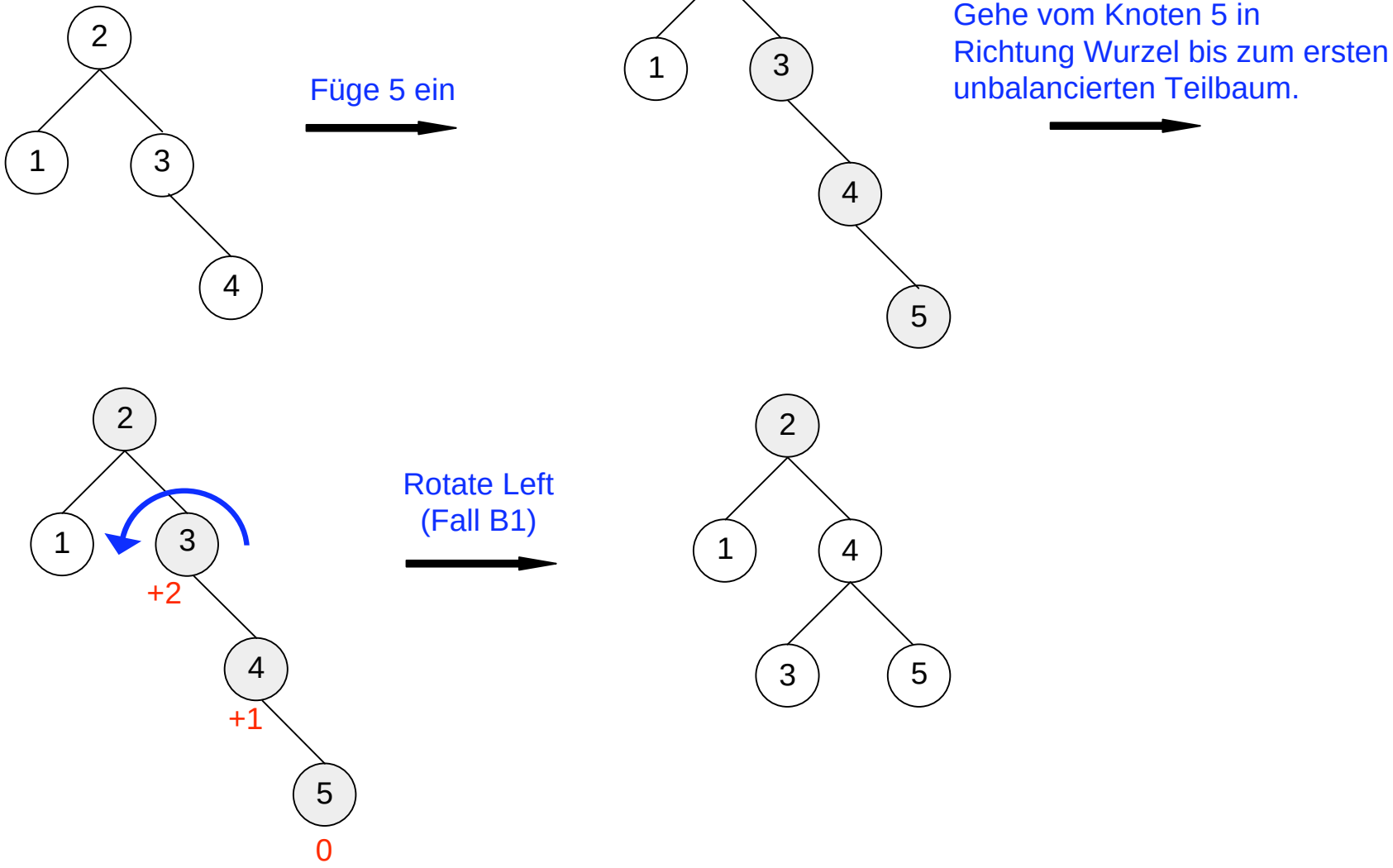
- Da der Baum vor dem Einfügen bzw. Löschen ausbalanciert war, haben alle Teilbäume einen Höhenunterschied von -1, 0 oder +1.
Damit können nach dem Schritt 1) nur die Fälle A1, A2, B1 oder B2 auftreten.

- Es läßt sich zeigen, dass beim Schritt 2) maximal eine Rotationsoperation notwendig ist.

  Beweisskizze:
  für jeden der Fälle A1, A2, B1 und B2 zeigt man, dass der Teilbaum nach dem Anwenden der entsprechenden Rotationsoperation die gleiche Höhe besitzt wie vor dem Einfügen bzw. Löschen (Schritt 1). Damit können weiter oben im Baum keine weiteren unbalancierten Stellen entstehen.

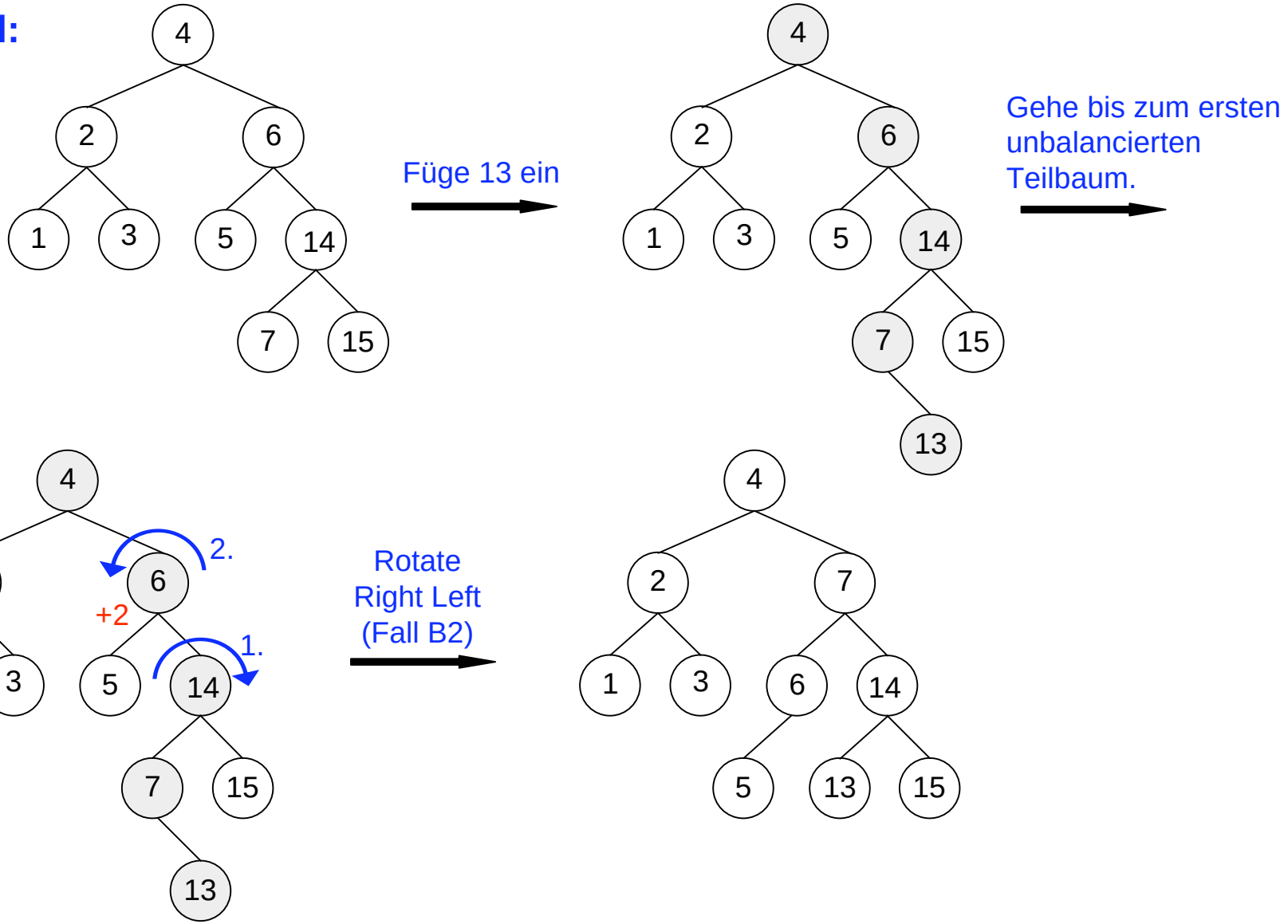
Einfügen in AVL-Bäumen (1)

Beispiel:



Einfügen in AVL-Bäumen (2)

Beispiel:



Algorithmen (1)

Datenstruktur

Wie bei binären Suchbäumen.
Zusätzlich wird für jeden Knoten im Baum noch die Höhe des entsprechenden Teilbaums abgespeichert.

```
struct Node {
        int height;
        KeyType key; ValueType value;
        Node* left;     // linkes Kind
        Node* right;    // rechtes Kind
};
```

Algorithmen

Erweitere insert-Operation für binäre Suchbäume um Balancieroperation.
(Die remove-Operation wird analog erweitert. Die search-Operation bleibt unverändert.)

```
bool insertR(KeyType k, ValueType v, Node*& p) {
        bool ret;
        if (p == 0) {
                p = new Node;
                p->key = k; p->value = v;  p->height = 0;
                ret = true; }
        else if (k < p->key)
                ret = insertR(k,v,p->left);
        else if (k > p->key)
                ret = insertR(k,v,p->right);
        else    // k bereits vorhanden
                ret = false;
        if (ret)
                balance(p);
        return ret;
}
```

Ein Blatt hat die Höhe 0

Knoten wurde eingefügt: daher Höhe aktualisieren und ausbalancieren, falls notwendig.

Die blauen Anweisungen wurden eingefügt, damit die Balancieroperation nach rekursivem Aufruf und vor Verlassen der Funktion durchgeführt werden kann.

Algorithmen (2)

```
void balance(Node*& p) {
      if (p == 0) return;
      // Höhe aktualisieren:
      p->height = max(getHeight(p->left), getHeight(p->right)) + 1;
      // Balanciere aus, da Baum linkslastig:
      if (getBalance(p) == -2) {
            if (getBalance(p->left) <= 0) // d.h. == -1 od. 0
                  rotateRight(p);
            else // getBalance(p->left) == +1
                  rotateLeftRight(p);
      }
      // Balanciere aus, da Baum rechtslastig:
      else if (getBalance(p) == +2) {
            if (getBalance(p->right) >= 0) // d.h. == 0 od. +1
                  rotateLeft(p);
            else // getBalance(p->left) == -1
                  rotateRightLeft(p);
}

int getHeight(const Node* p) {
      if (p == 0) return -1;
      else return p->height; }

int getBalance(const Node* p) {
      if (p==0) return 0;
      else return getHeight(p->right) - getHeight(p->left); }
```

Fall A1

Fall A2

Fall B1

Fall B2

Höhe eines Baums.

Ein leerer Baum hat die Höhe -1.

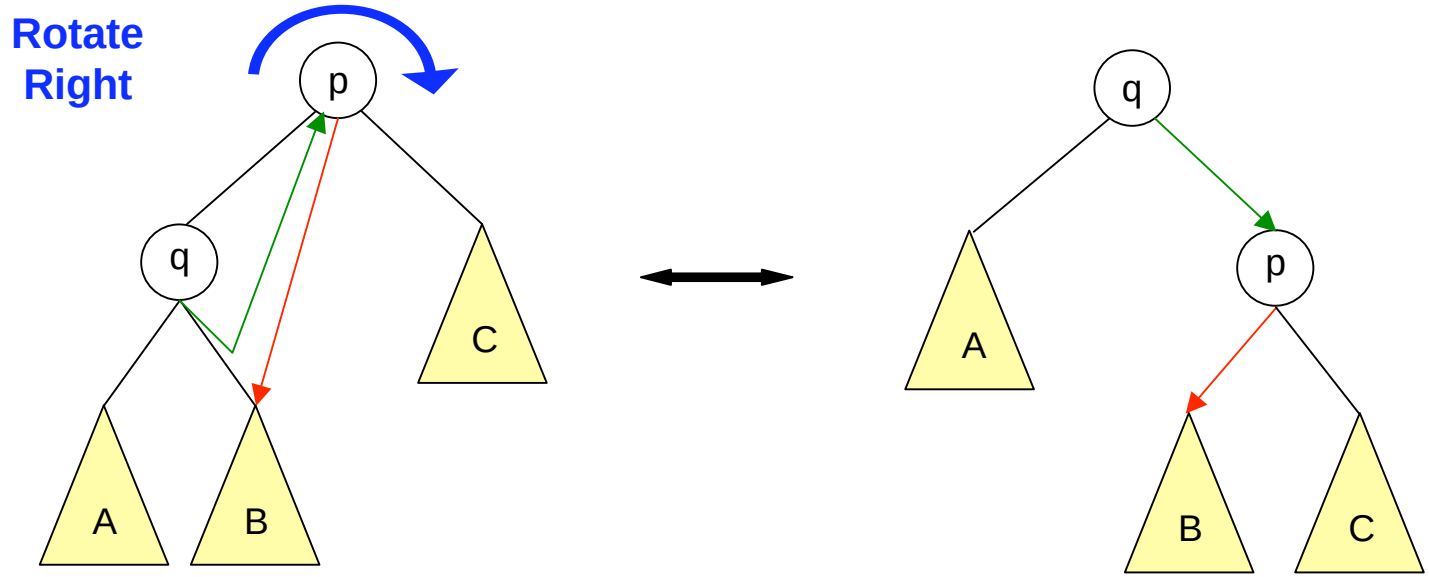
Höhenunterschied zwischen linken und rechten Unterbaum.

Bei einem leeren Baum, wird der Höhenunterschied als 0 definiert.

Algorithmen (3)

```
void rotateRight (Node*& p)
// Es muss p->left != 0 sein! Sonst könnte der Baum gar nicht
// unbalanciert sein...
{
	Node* q = p->left;
	p->left = q->right;
	q->right = p;

	p->height = max(getHeight(p->left), getHeight(p->right)) + 1;
	q->height = max(getHeight(q->left), getHeight(q->right)) + 1;
	p = q;
}
```

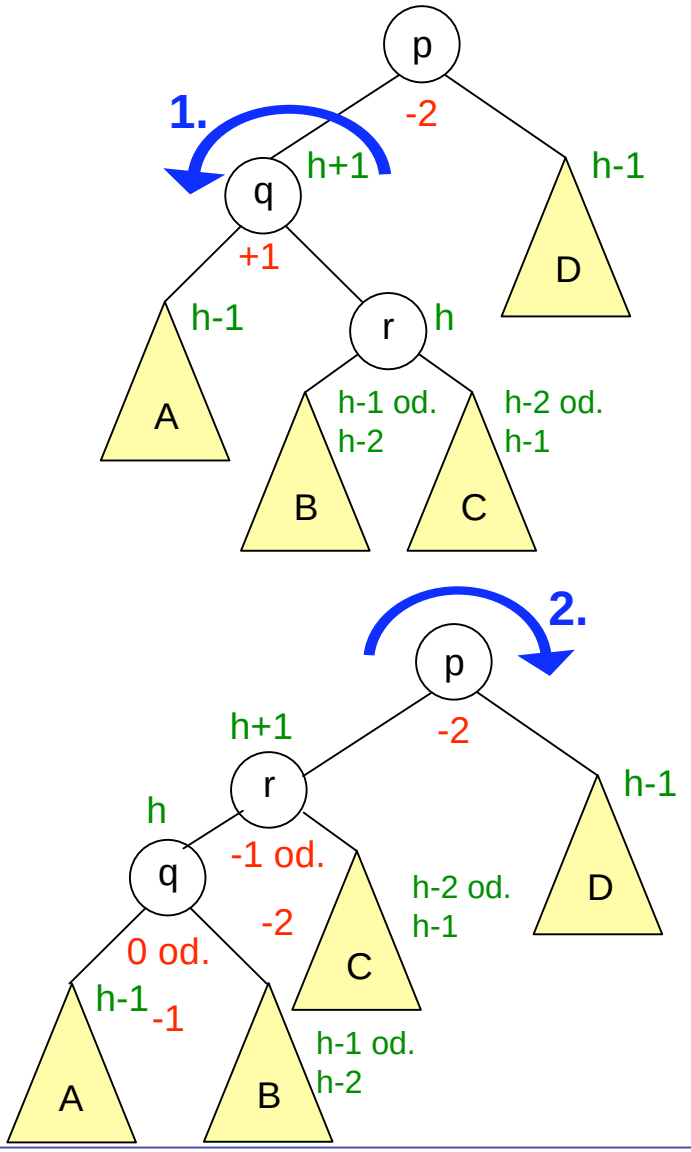


Algorithmen (4)

```
void rotateLeft (Node*& p)
{
      // analog zu rotateRight;
}

void rotateLeftRight (Node*& p)
// Es muss p->left != 0 sein!
{
      rotateLeft(p->left);
      rotateRight(p);
}

void rotateRightLeft (Node*& p)
// Es muss p->right != 0 sein!
{
      rotateRight(p->right);
      rotateLeft(p);
}
```



Teil 1:
Suchen

- Problemstellung
- Elementare Suchverfahren
- Hashverfahren
- Binäre Suchbäume
- Ausgeglichene Bäume
  - AVL-Bäume
  - Splay-Bäume
- B-Bäume
- Digitale Suchbäume
- Heaps

Splay-Bäume

- Splay-Bäume sind selbstanordnende Bäume.

- Ziel: automatische Anpassung an Zugriffshäufigkeiten ohne explizite Speicherung von Balance-Information oder Häufigkeiten

- Sind die Zugriffshäufigkeiten fest und vorher bekannt, so lassen sich optimale Suchbäume konstruieren, die die Suchkosten minimieren (s. Ottmann & Widmayer, 5.7)

- Hier: Zugriffshäufigkeiten unbekannt oder variabel.

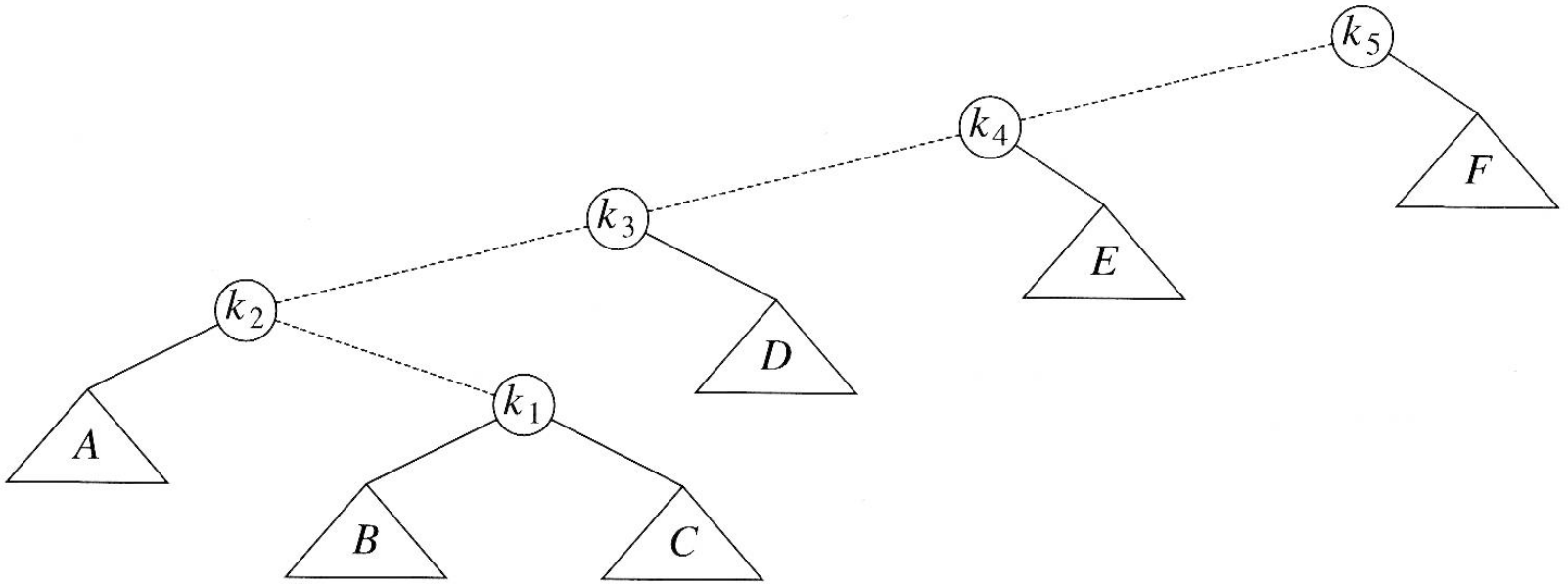
- Grundidee: Move-to-root-Strategie - nach jedem Zugriff wird ein Knoten durch Rotationen in Richtung der Wurzel bewegt.

- Splay: engl. verbreitern - Baum wird in die Breite angeordnet.

Eine einfache Idee (die nicht funktioniert) - 1

Jeder Knoten auf dem Zugangspfad vertauscht in einer Rotation die Position mit seinem Vorfahren.

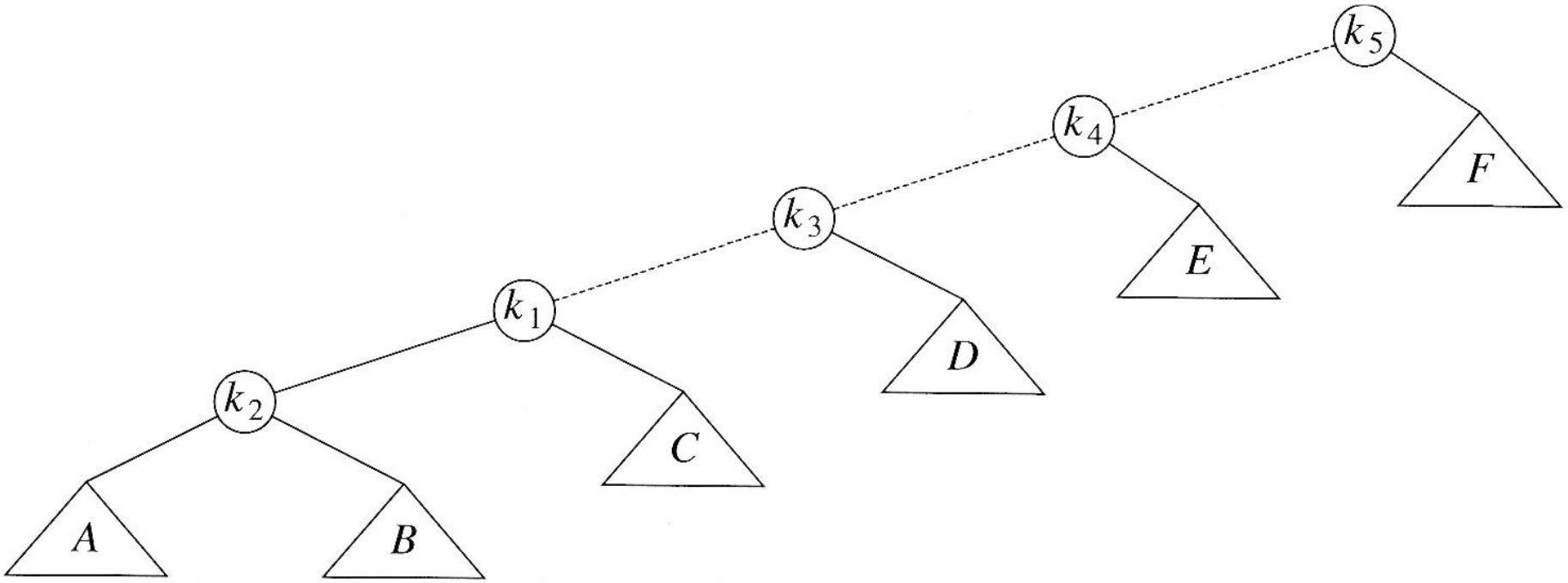
Beispiel: Zugriff auf k_1



[Weiss, 1999]

Eine einfache Idee (die nicht funktioniert) - 2

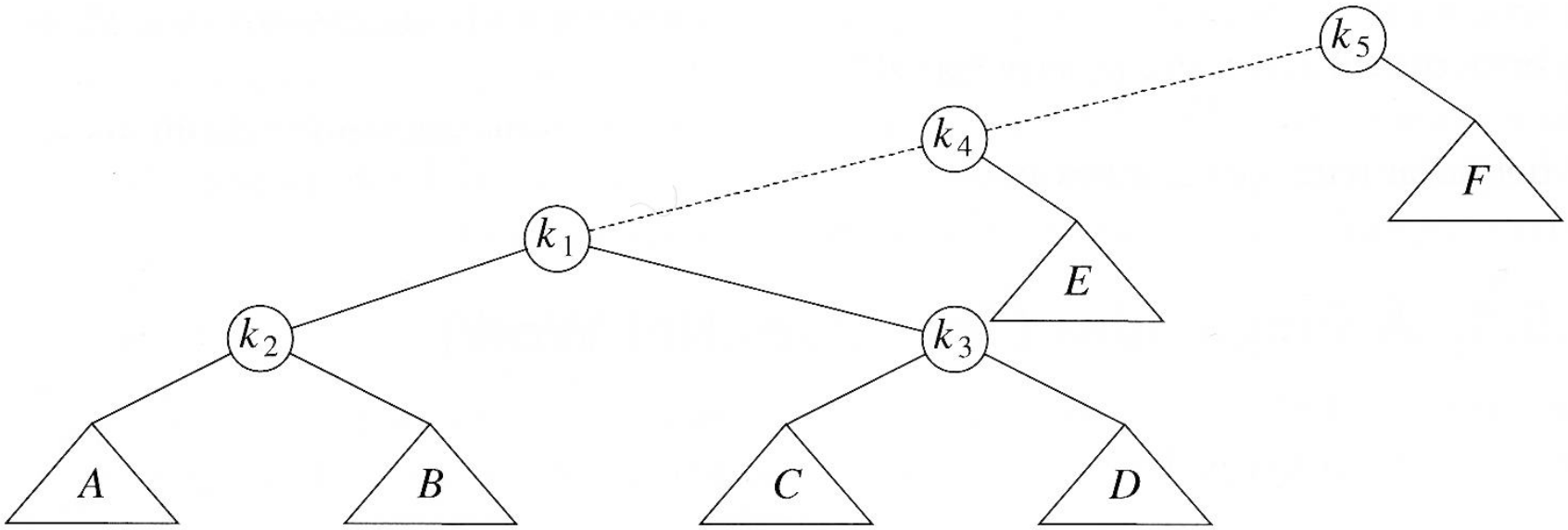
Rotation k_1 <-> k_2



[Weiss, 1999]

Eine einfache Idee (die nicht funktioniert) - 3

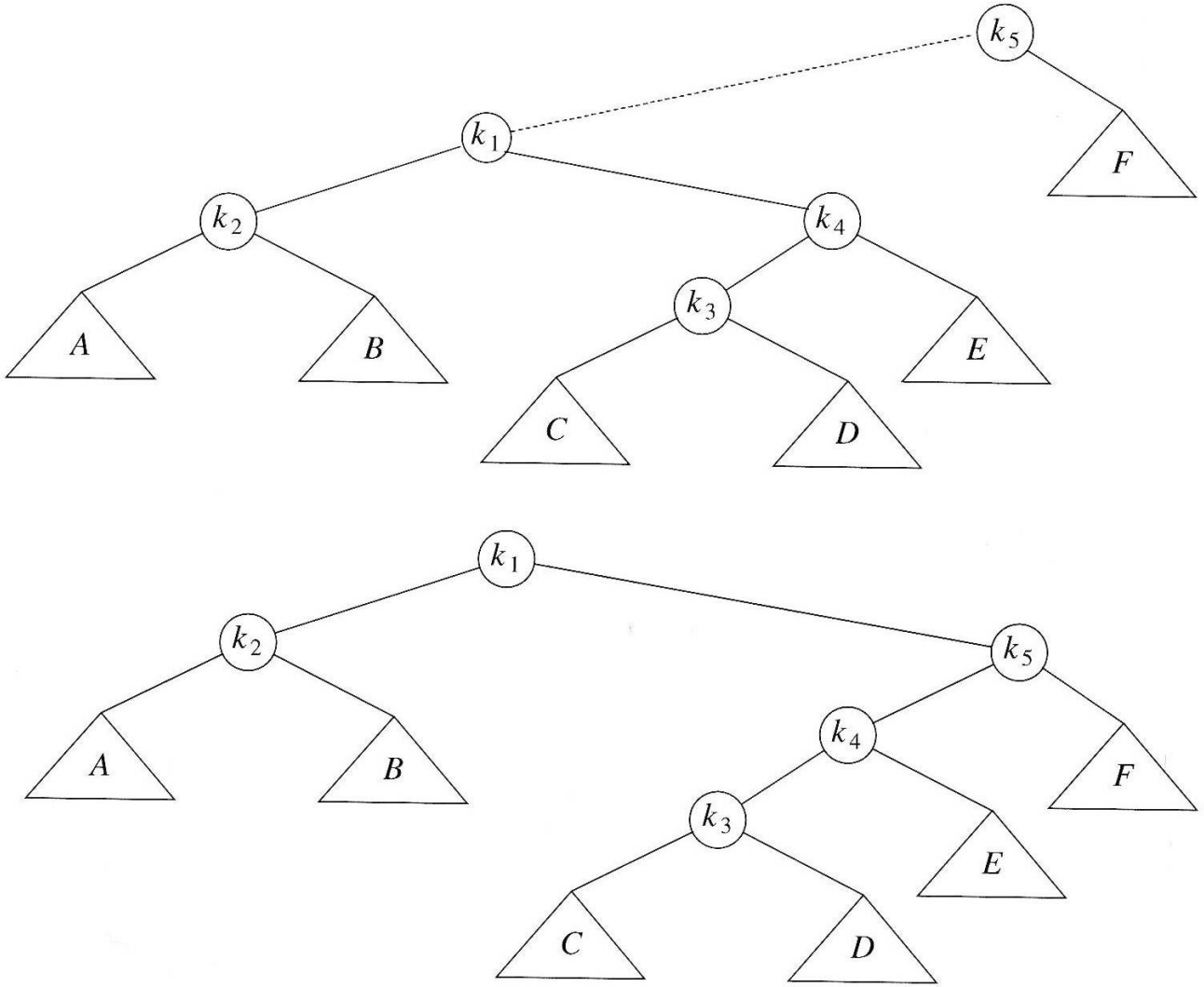
Rotation k_1 <-> k_3



[Weiss, 1999]

Eine einfache Idee (die nicht funktioniert) - 4

Rotation k_1 <-> k_4 und k_1 <-> k_5



[Weiss, 1999]

Eine einfache Idee (die nicht funktioniert) - 5

Durch die Rotationen wird der Knoten nach dem Zugriff bis an die Wurzel transportiert.

⇒ nächster Zugriff ist extrem schnell.

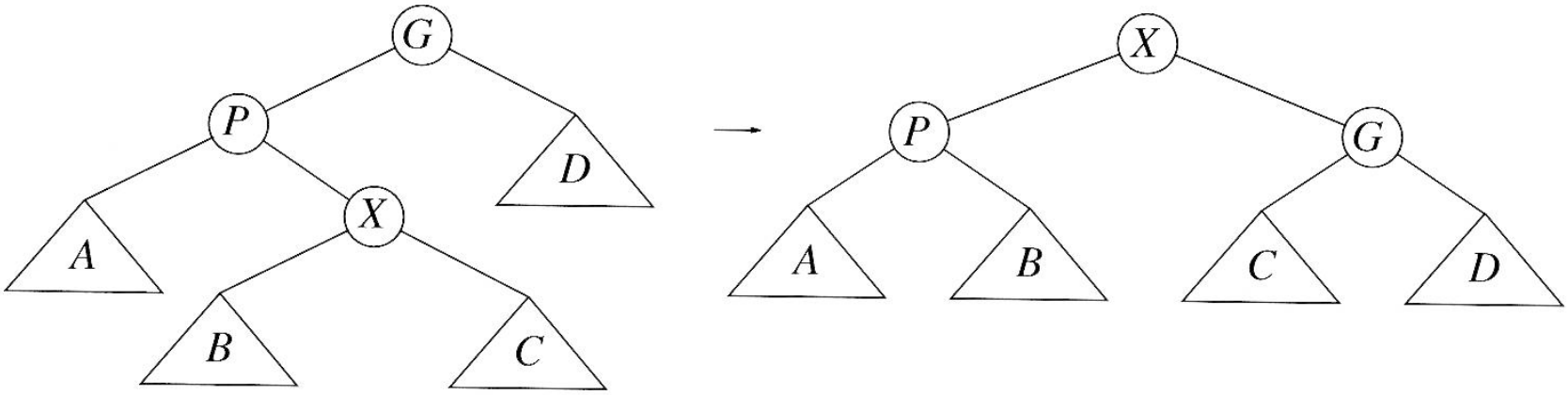
Aber: Ein anderer Knoten (k3) wurde dafür fast bis unten gedrückt.

Im Extremfall (z.B. bei einem Baum aus linken Kindern) wird die Struktur nicht wesentlich verbessert bzw. ergeben sich repetitive Folgen von Baumzuständen => O(MN) bei M Zugriffen.

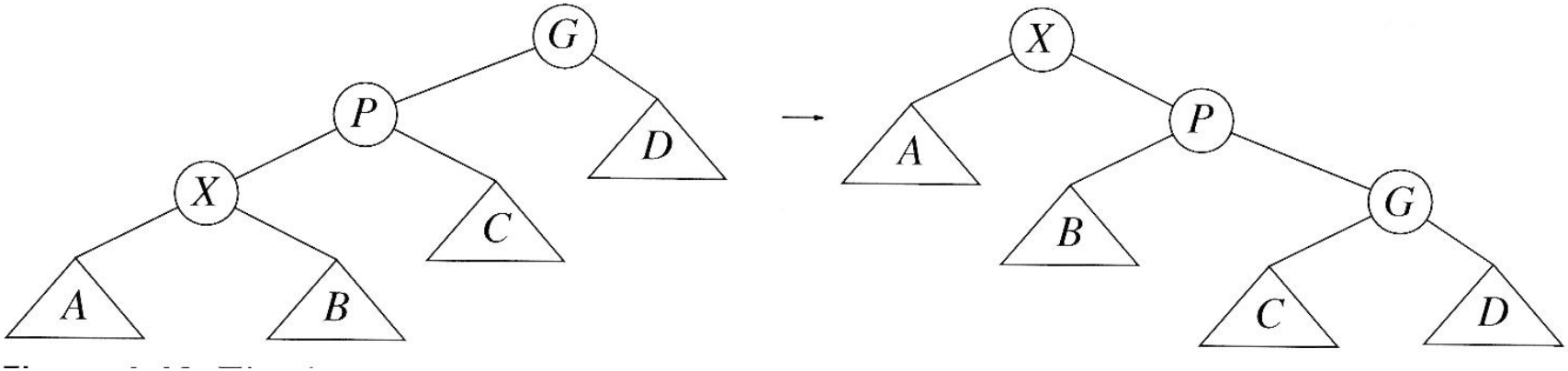
=> Baum muß so umstrukturiert werden, das er möglichst „breit“ wird (splaying).

Splaying-Strategie

2 Regeln (+ 2 Spiegelbilder):



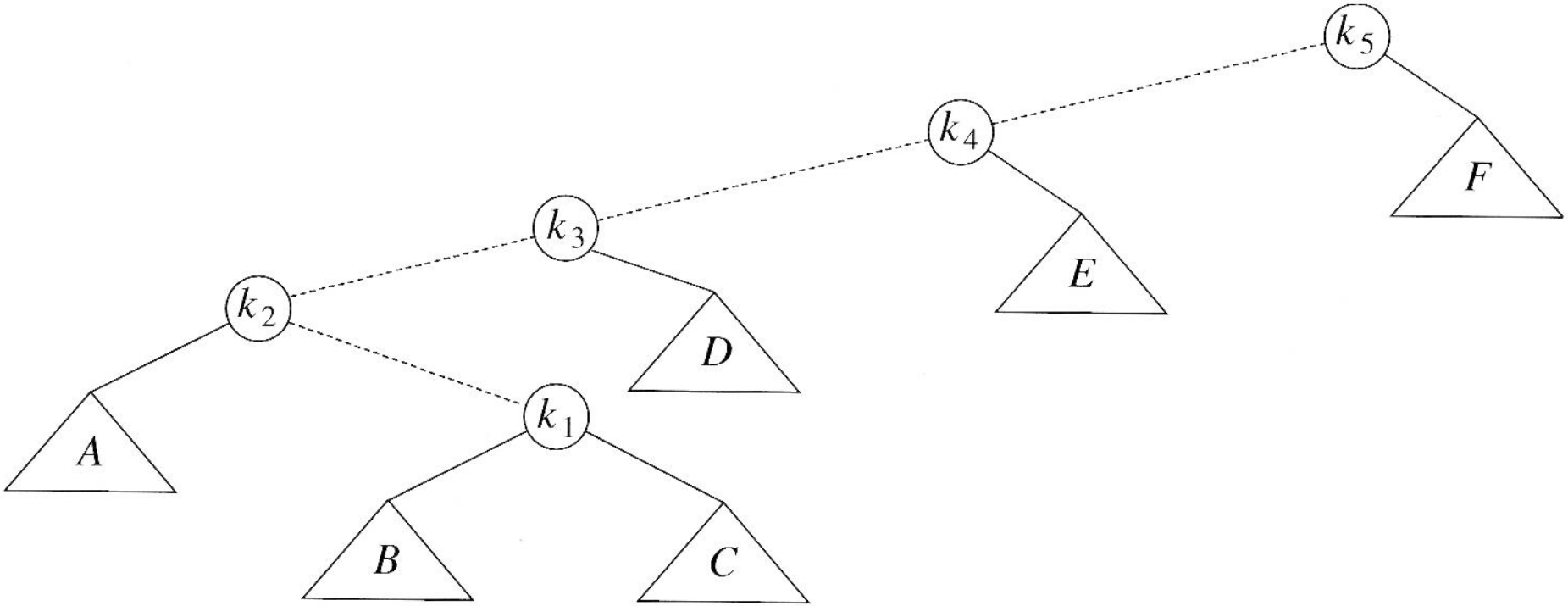
„Zick-Zack“ => Doppelte Rotation wie im AVL-Baum



„Zick-Zick“

[Weiss, 1999]

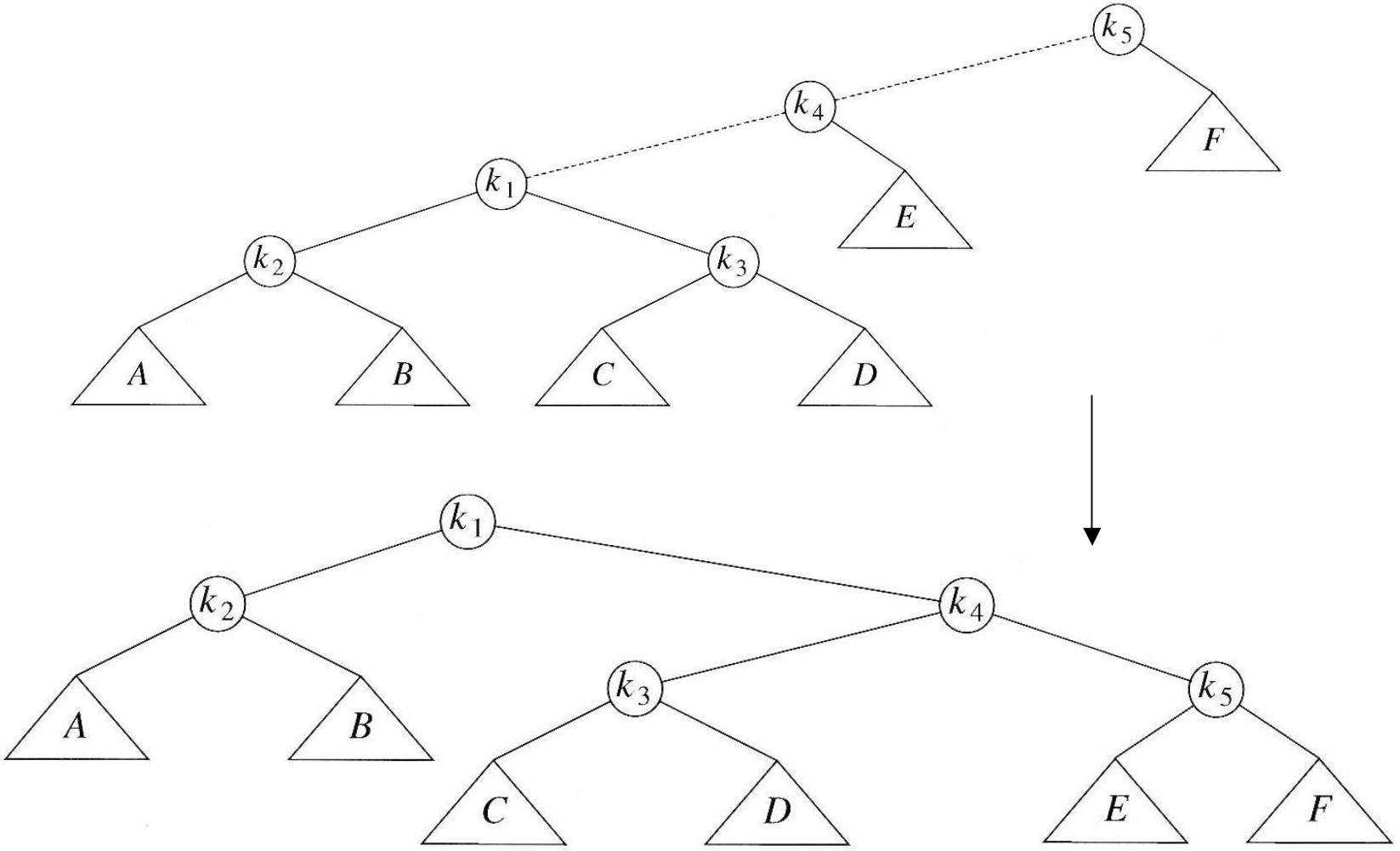
Splaying: Beispiel (1)



„Zick-Zack“

[Weiss, 1999]

Splaying: Beispiel (2)

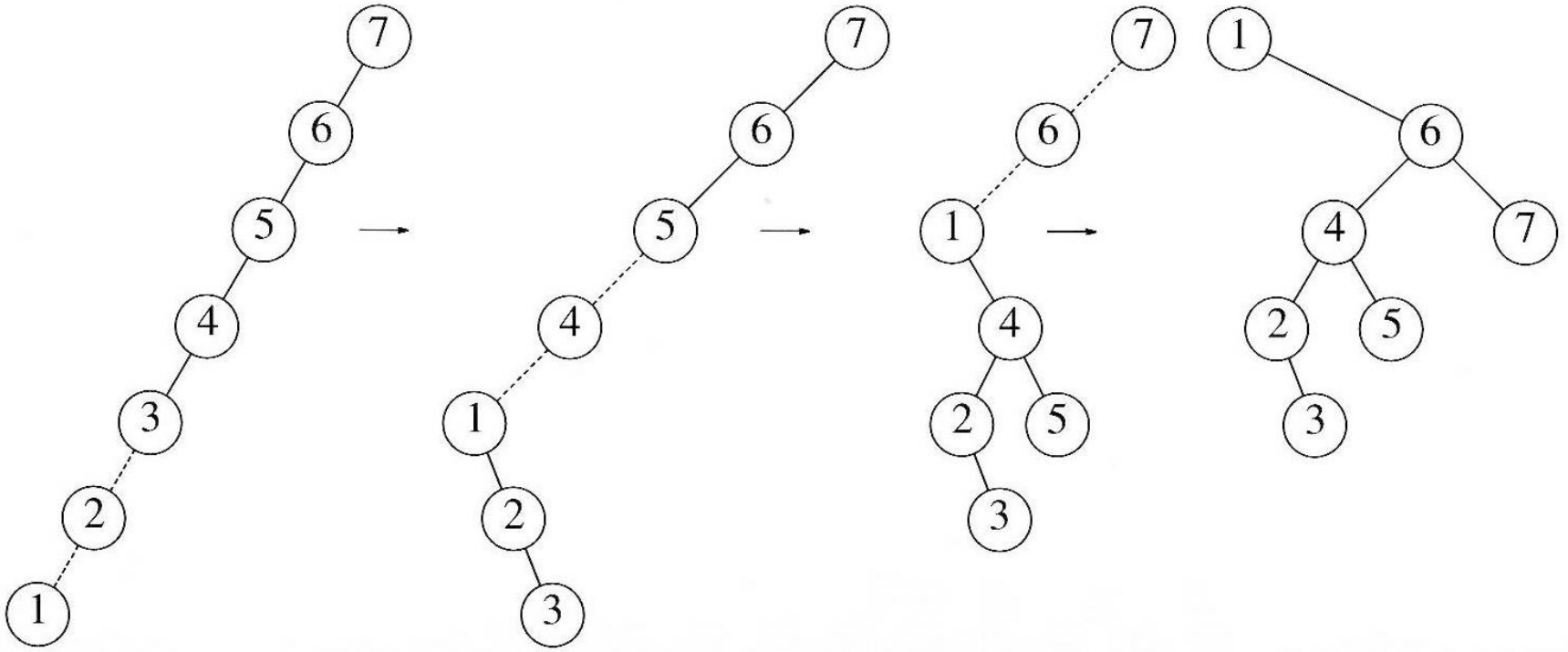


„Zick-Zick“

[Weiss, 1999]

Verbreiterte Baumstruktur durch Splaying

Splaying bewegt nicht nur die Knoten nach einem Zugriff an die Wurzel, sondern halbiert bei unbalancierten Bäumen oft die Tiefe der meisten Knoten auf dem Zugangspfad.



[Weiss, 1999]

Amortisierte Worst-Case-Analyse

Amortisierte Analyse: statt der Laufzeit einer *einzigen* Operation wird stattdessen die Laufzeit per Operation in einer *Folge von M Operationen* abgeschätzt.

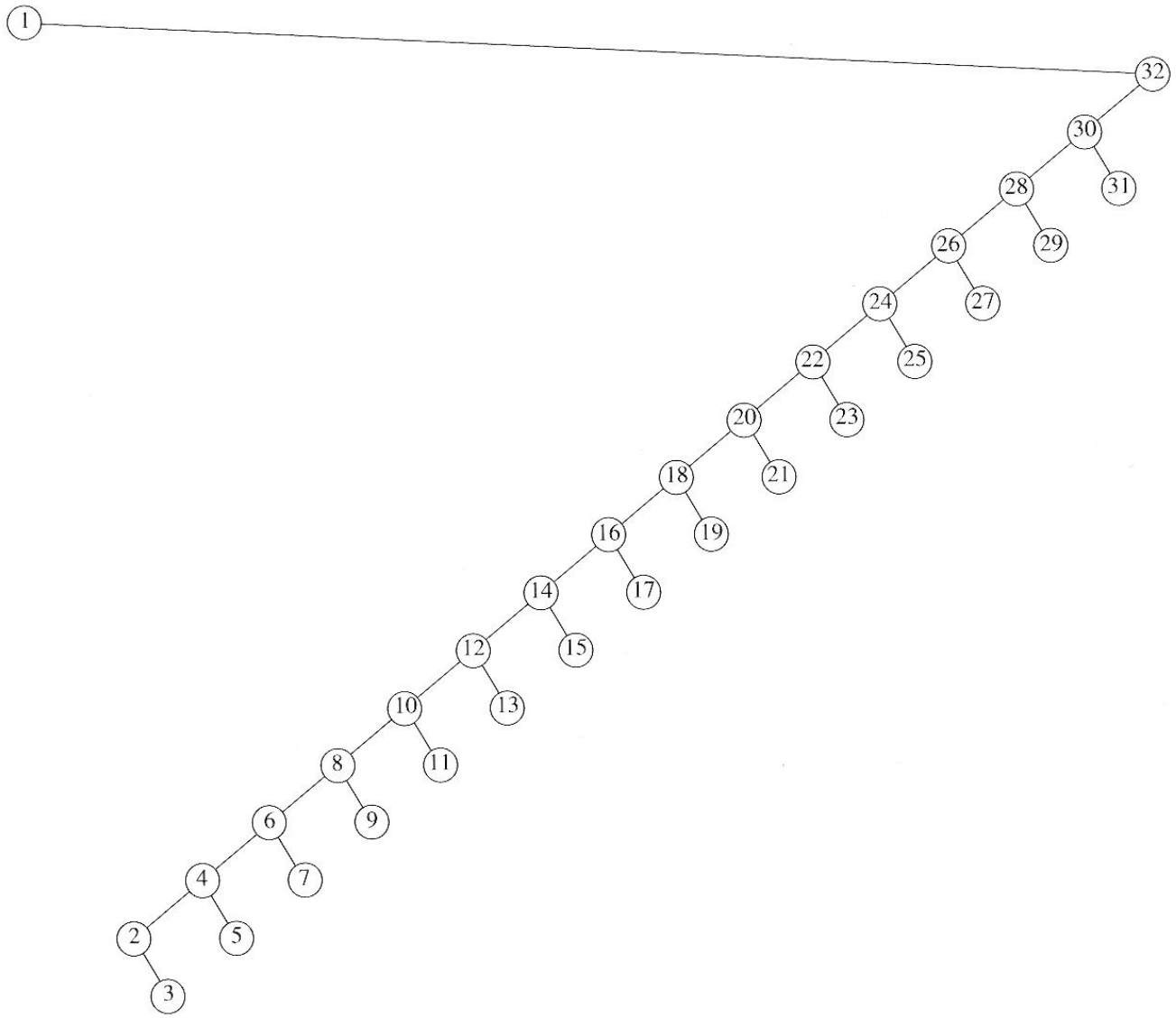
Bei einer Worst-Case-Laufzeit von O(M f(N)) ist die amortisierte Laufzeit O(f(N)) pro Operation.

Man kann zeigen (s. Ottmann & Widmayer, 5.4.2):

Splay-Bäume haben eine amortisierte Laufzeit von O(log N), egal welche Sequenz von Operationen gewählt wird.

D.h. obwohl einzelne Operationen O(N) brauchen können, ist *garantiert*, daß die nachfolgenden Operationen um so kürzer sind, so daß sich bei hinreichend großen M insgesamt O(log N) ergeben.

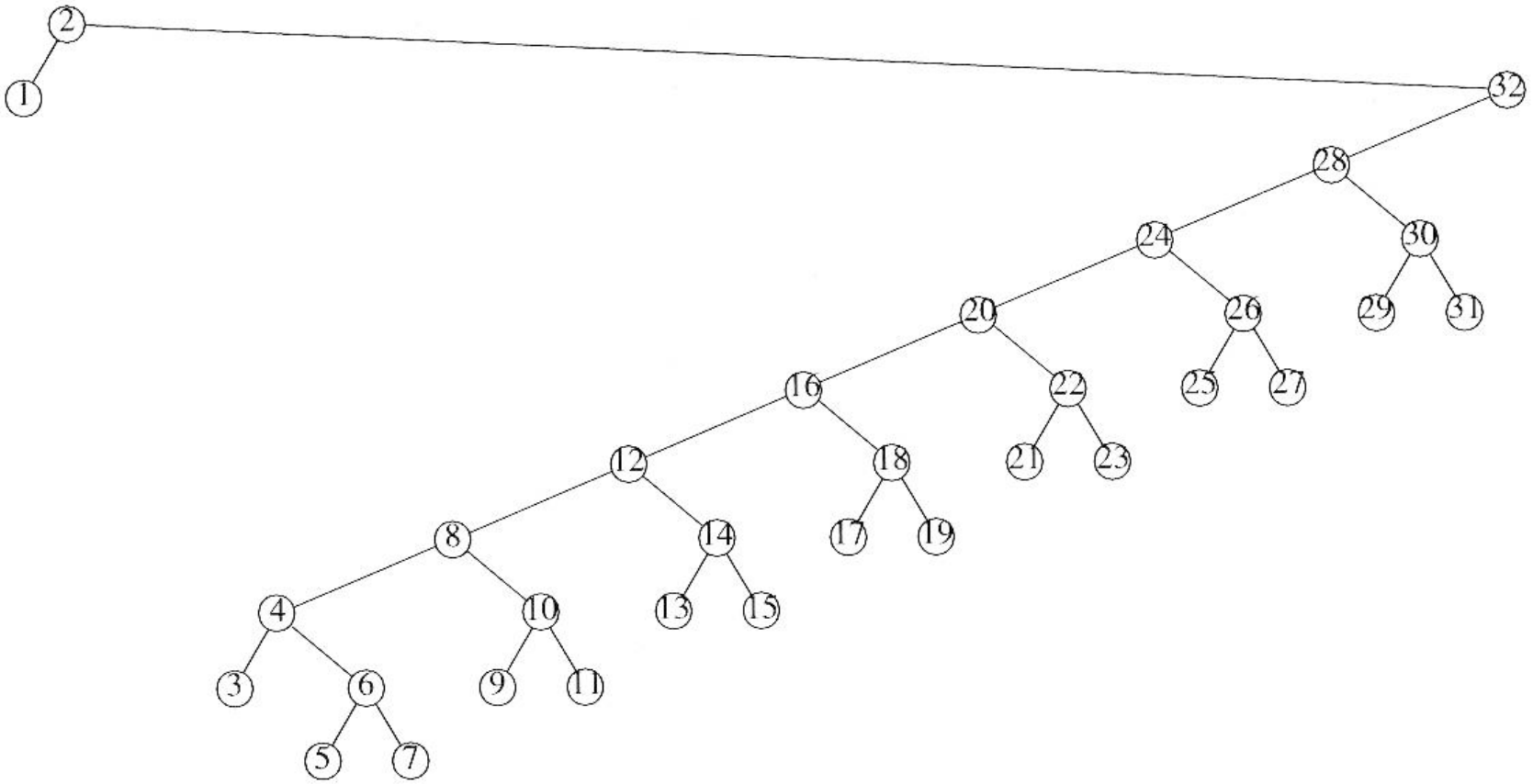
Beispiel für Worst-Case (1)



Splaying an Knoten 1 an einem Baum aus linken Kindern

[Weiss, 1999]

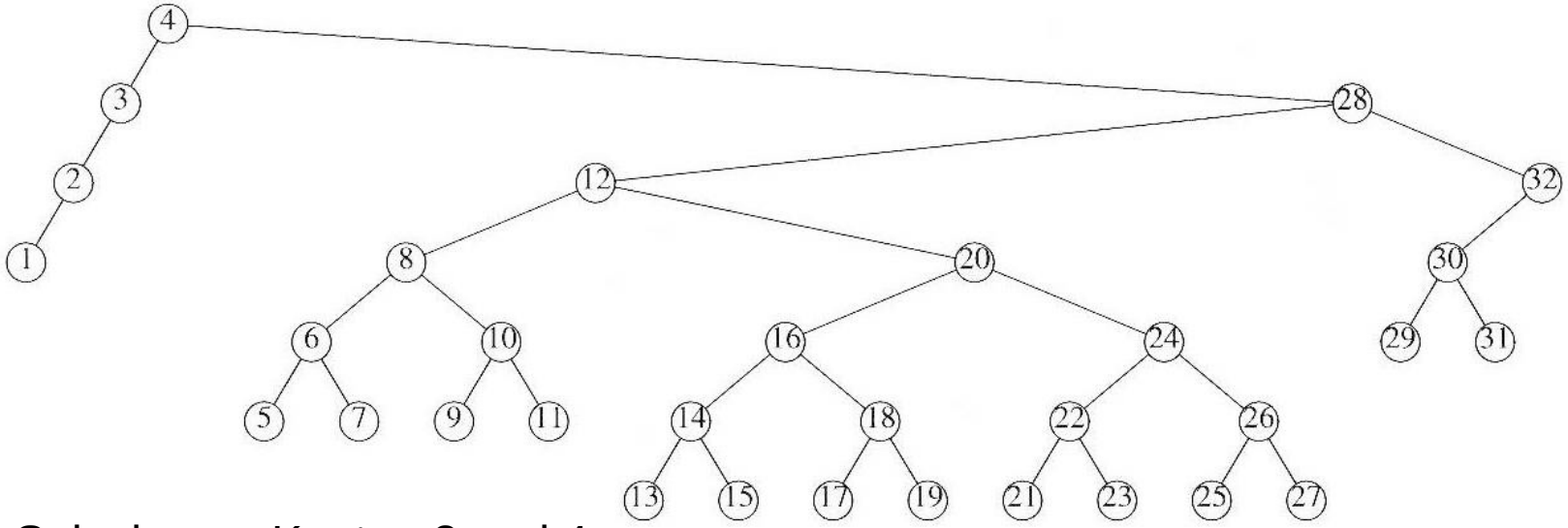
Beispiel für Worst-Case (2)



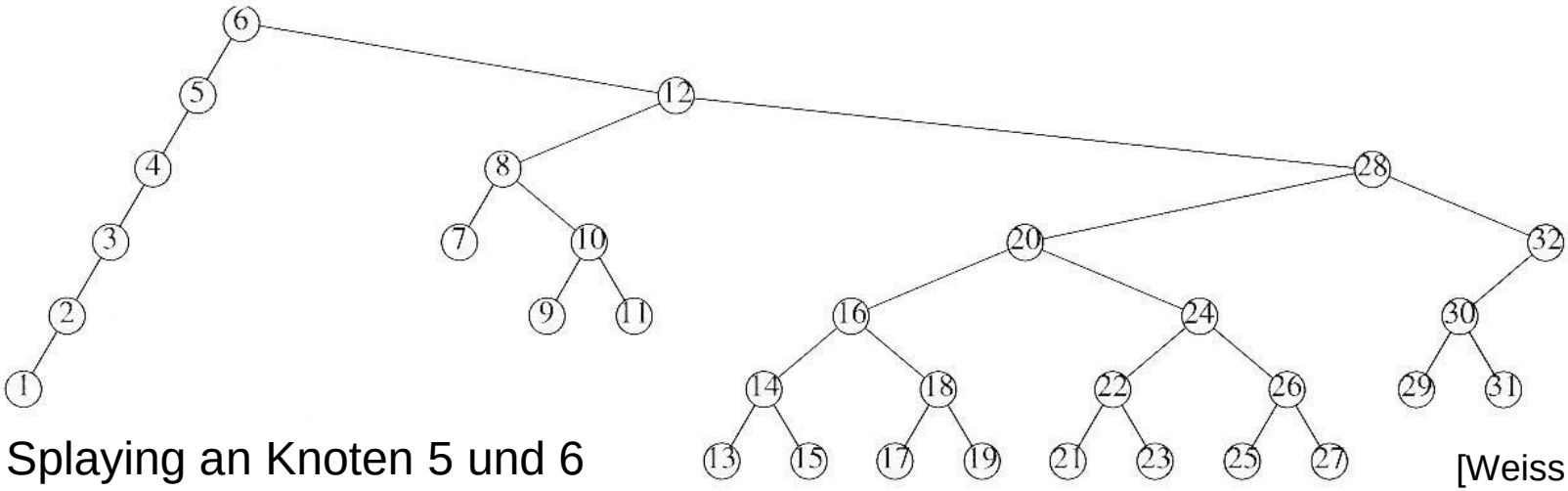
nach Splaying an Knoten 2

[Weiss, 1999]

Beispiel für Worst-Case (3)



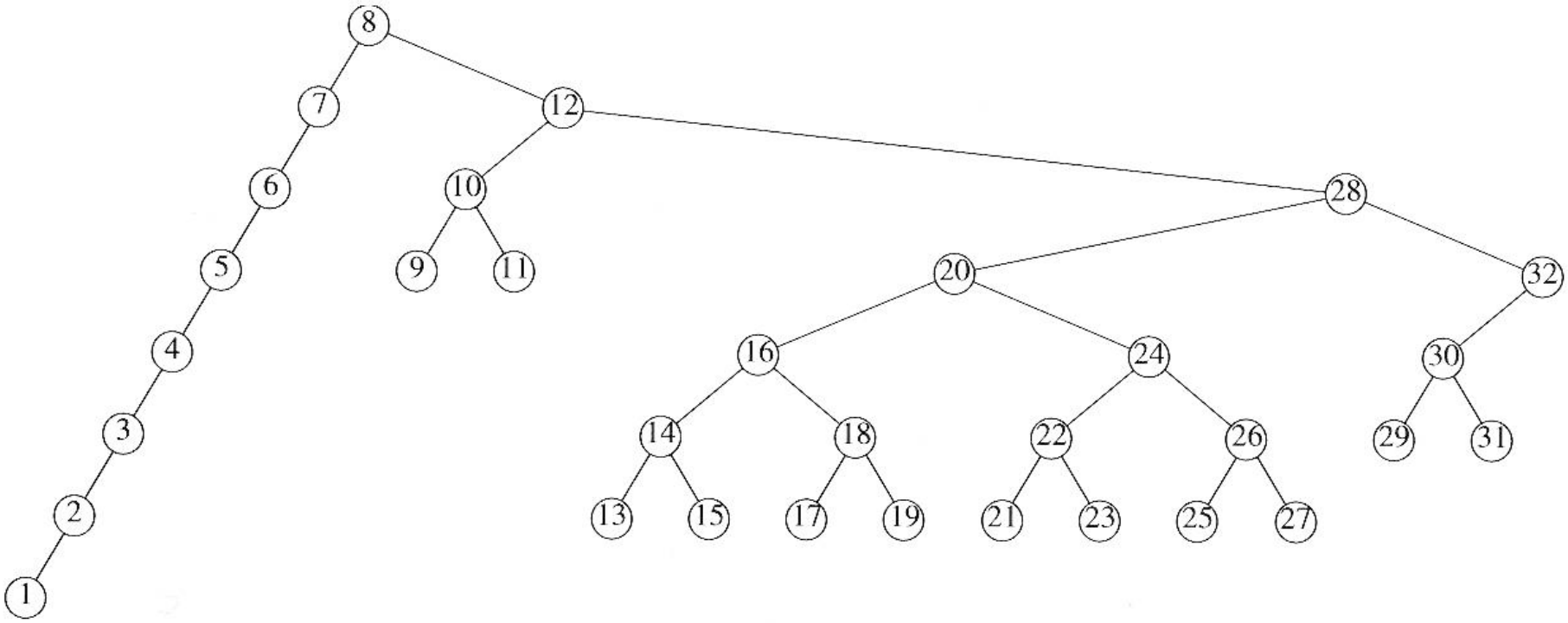
nach Splaying an Knoten 3 und 4



nach Splaying an Knoten 5 und 6

[Weiss, 1999]

Beispiel für Worst-Case (4)



nach Splaying an Knoten 7 und 8

[Weiss, 1999]