Teil 1:
Suchen

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

Randomisierte Suchbäume

- Binärbäume haben durchschnittlich über alle möglichen Inputfolgen O(log n) Zugriffszeit, aber es gibt bestimmte Inputfolgen (z.B. aufsteigend geordnet), für die O(n) gilt.

- Grundidee: Randomisierung, d.h. jede Inputfolge wird zufällig permutiert, bevor sie in einem Suchbaum angeordnet wird.

- Dadurch wird durchschnittlich über all Permutationen O(log n) Zugriffszeit erzielt, selbst bei ständiger Wiederholung einer besonders ungünstigen Inputfolge.

- Vorgehensweise ist analog zum randomisierten Quicksort.

Treaps

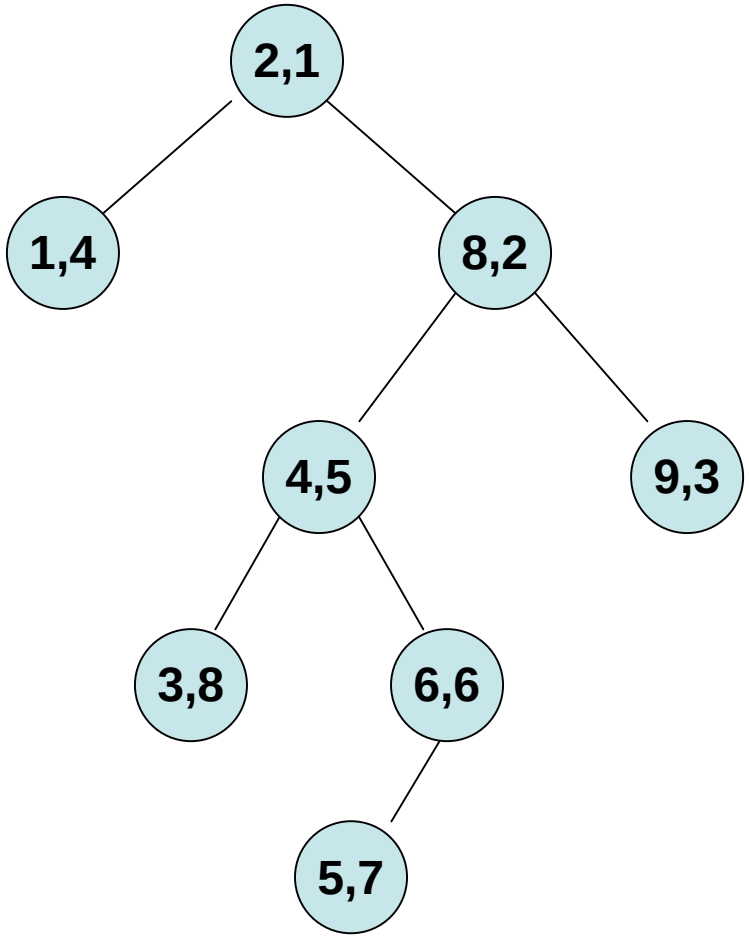
Treaps: Suchbäume, in denen in jedem Knoten zusätzlich zu jedem Suchschlüssel eine Priorität gespeichert werden, für die gelten muß:

1. Suchbaumbedingung: Für jedes y im linken Teilbaum eines Knotens p gilt y.key < p.key für jedes x im rechten Teilbaum p.key < x.key.

2. Heapbedingung: Für jedes Kind z von p gilt p.priority < z.priority.

Treap: Hybrid aus Baum (Tree) und Vorrangswarteschlange (Heap).

Beispiel: Treap mit ganzzahliger Priorität



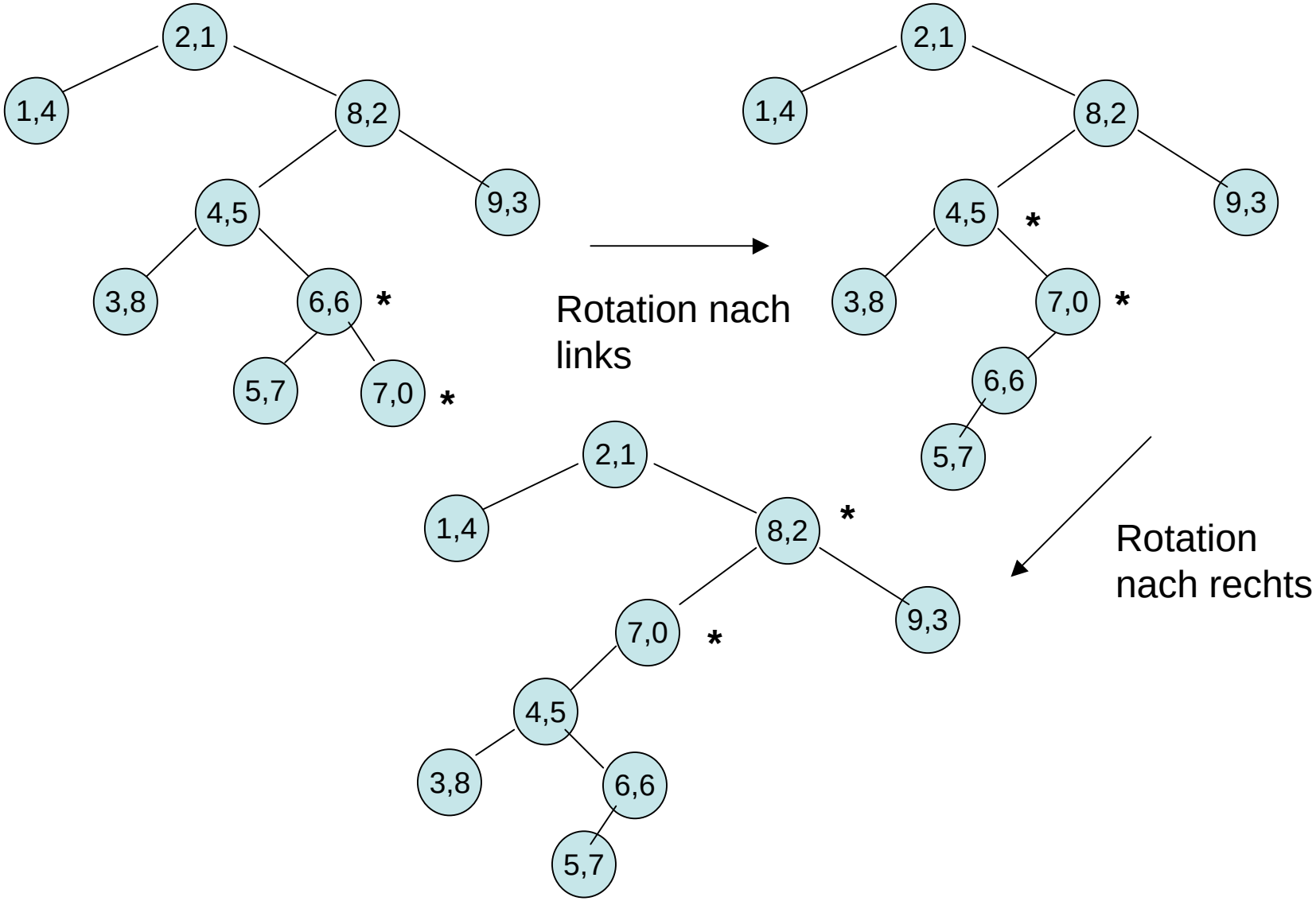
Suchen, Einfügen und Löschen in Treaps

Suchen: genau wie in Binärbäumen

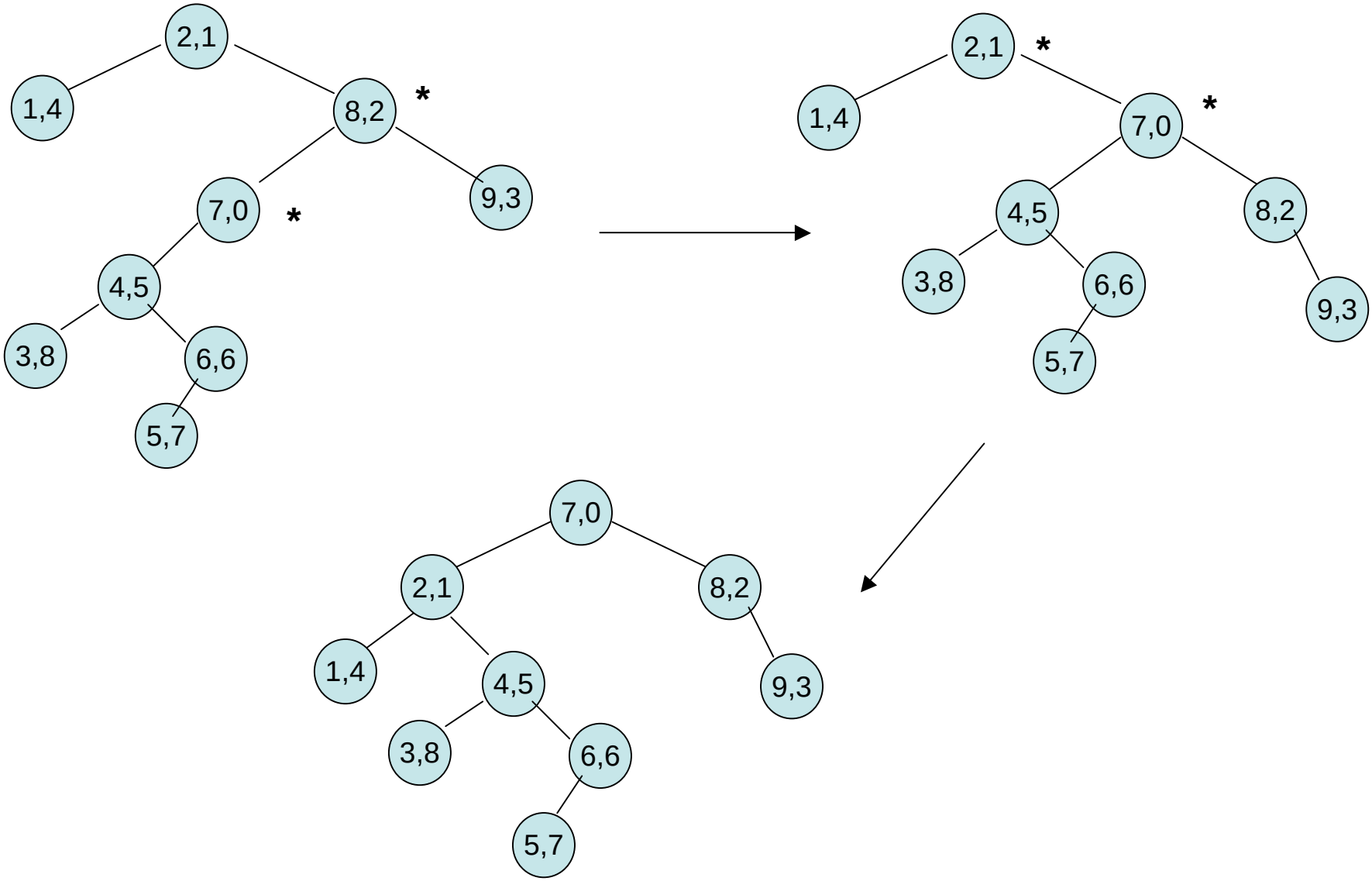
Einfügen:
• wie bisher wird zunächst gesucht und dann an das Blatt, an dem die erfolglose Suche endet, angehängt.
• I.A. genügt der Baum nach Einfügen des Blattes nicht mehr der Heapbedingung, d.h. das Blatt kann eine niedrigere Priorität als sein Vorfahr haben.
• Durch Rotationen (links oder rechts) wird der neue Knoten so weit nach oben bewegt, bis die Heapbedingung wieder erfüllt ist. Gleichzeitig erhalten Rotationen die Suchbaumbedingung.

Löschen:
genau umgekehrt: zu löschender Knoten wird so lange nach unten (in Richtung des Knotens mit kleinerer Priorität) bewegt, bis er ein Blatt wird und dann gelöscht.

Beispiel: Einfügen und Löschen in Treaps (1)



Beispiel: Einfügen und Löschen in Treaps (2)



Treaps mit zufälligen Prioritäten

Ein randomisierter Suchbaum wird durch Prioritäten erreicht, die zufällig und unabhängig voneinander nach einer Gleichverteilung gewählt werden.

Voraussetzungen:
- eindeutige Prioritäten
- jede Permutation der Schlüssel muß gleich wahrscheinlich sein.
- Zufallsreihenfolge muß benutzerseitig unbekannt sein.

Bei jeder Einfügeoperation heißt das: die neu erzeugt Priorität muß mit gleicher Wahrscheinlichkeit in jedes Intervall fallen, das sich zwischen den bereits existierenden Prioritäten befindet.

Es läßt sich zeigen (Ottmann & Widmayer, 2002):
- durchschnittliche Suchpfadlänge ist O(log n)
- Löschen und Einfügen sind O(log n)
- im Durchschnitt werden weniger als 2 Rotationen benötigt, um Heapbedingung wieder herzustellen.

Praktische Realisierung

Ziel: die neu erzeugt Priorität muß mit gleicher Wahrscheinlichkeit in jedes Intervall fallen, das sich zwischen den bereits existierenden Prioritäten befindet.

Vorgehensweise:

1. Prioritäten werden als binäre Gleitkommazahlen in [0,1] erzeugt (z.B. 0.1101111001).

2. Nach Einfügen als Blatt wird Priorität als zufälliger Binärstring der Form $0.b_1b_2b_3...$ erzeugt. Dabei wird solange ein weiteres Zufallsbit angehängt bis entweder Priorität(Kind) > Priorität(Vorfahr) oder Priorität(Kind) < Priorität(Vorfahr) gilt.

3. Nach jeder Rotation wird überprüft, ob evtl. noch weitere Bits angefügt werden müssen.

Teil 1:
Suchen

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

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.

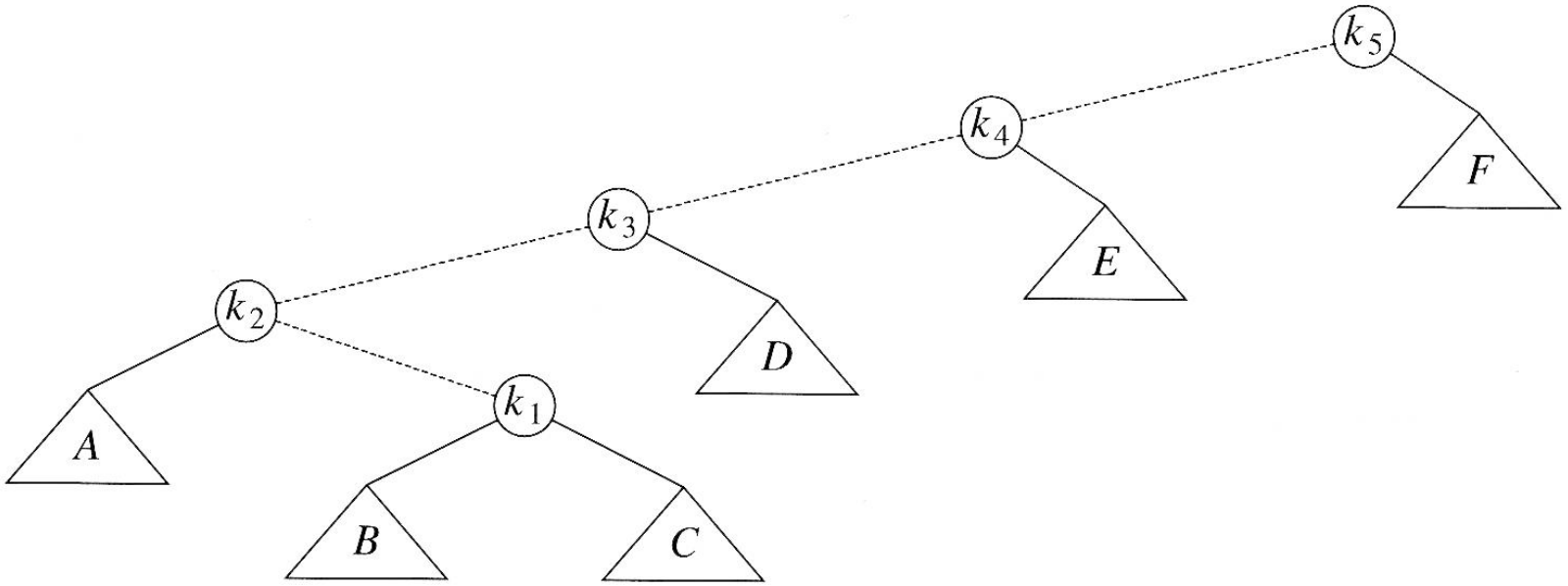
- Grundidee: Move-to-root-Strategie - nach jedem Zugriff wird ein Knoten durch Rotationen in Richtung der Wurzel bewegt.

- 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.

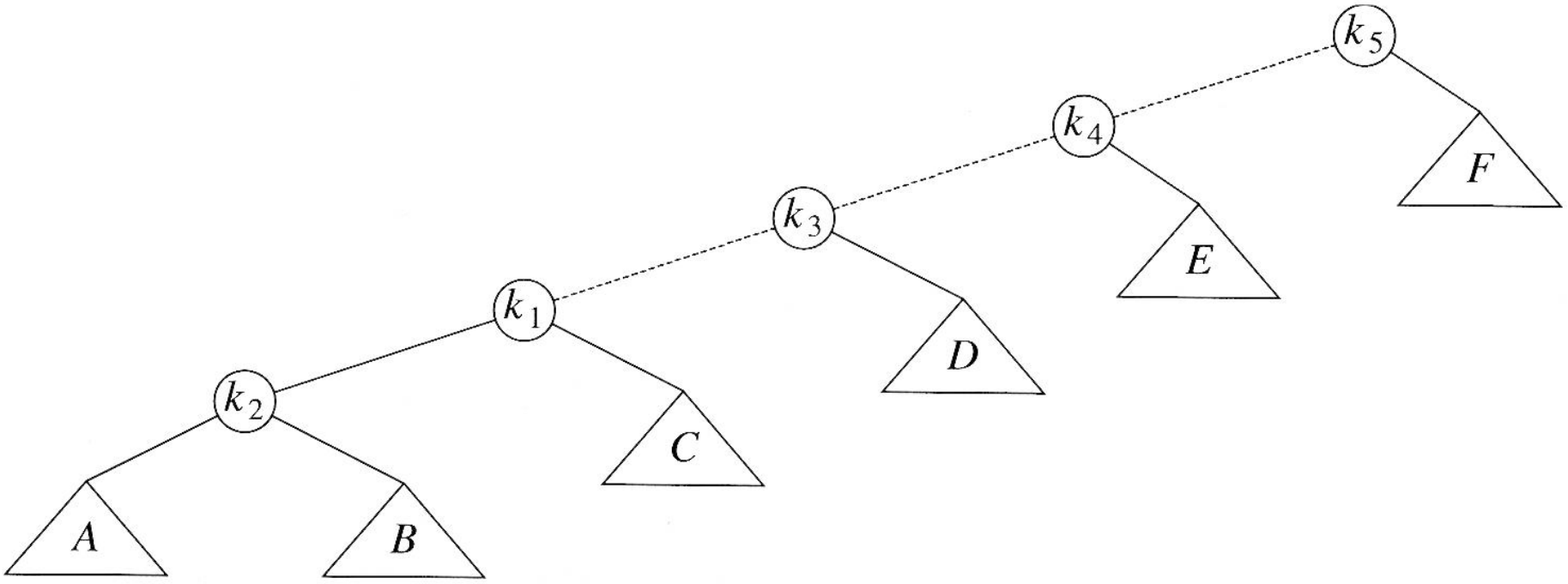
Beispiel: Zugriff auf k_1



[Weiss, 1999]

Eine einfache Idee (die nicht funktioniert) - 2

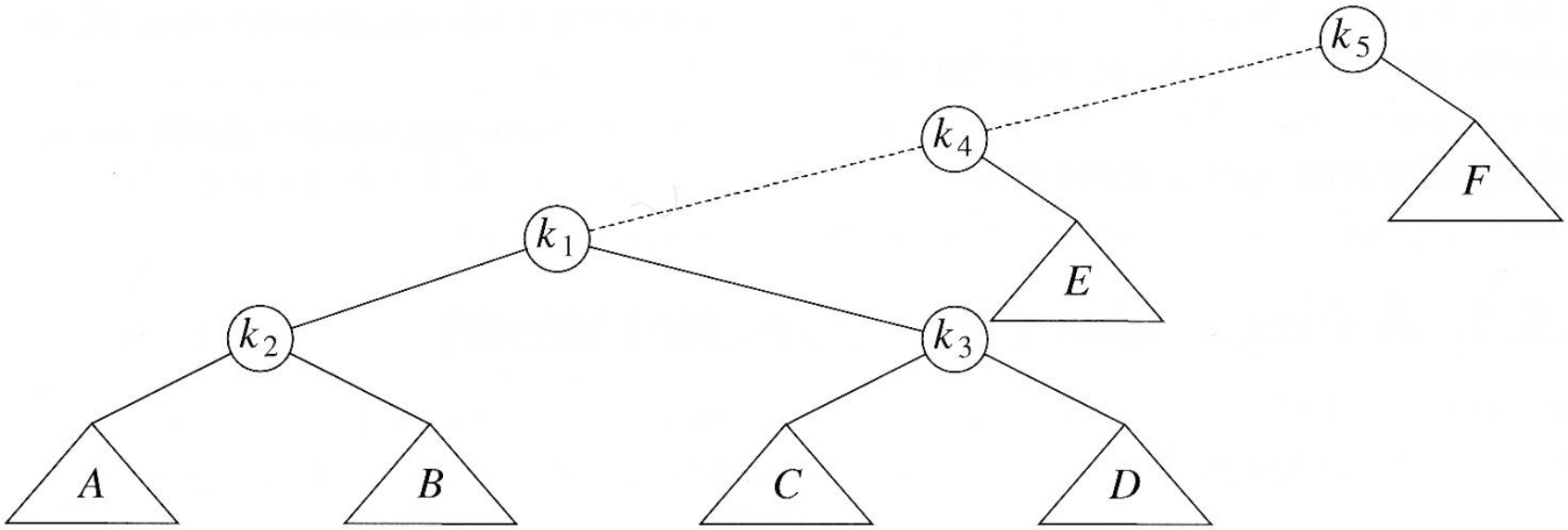
Rotation k_1 <-> k_2



[Weiss, 1999]

Eine einfache Idee (die nicht funktioniert) - 3

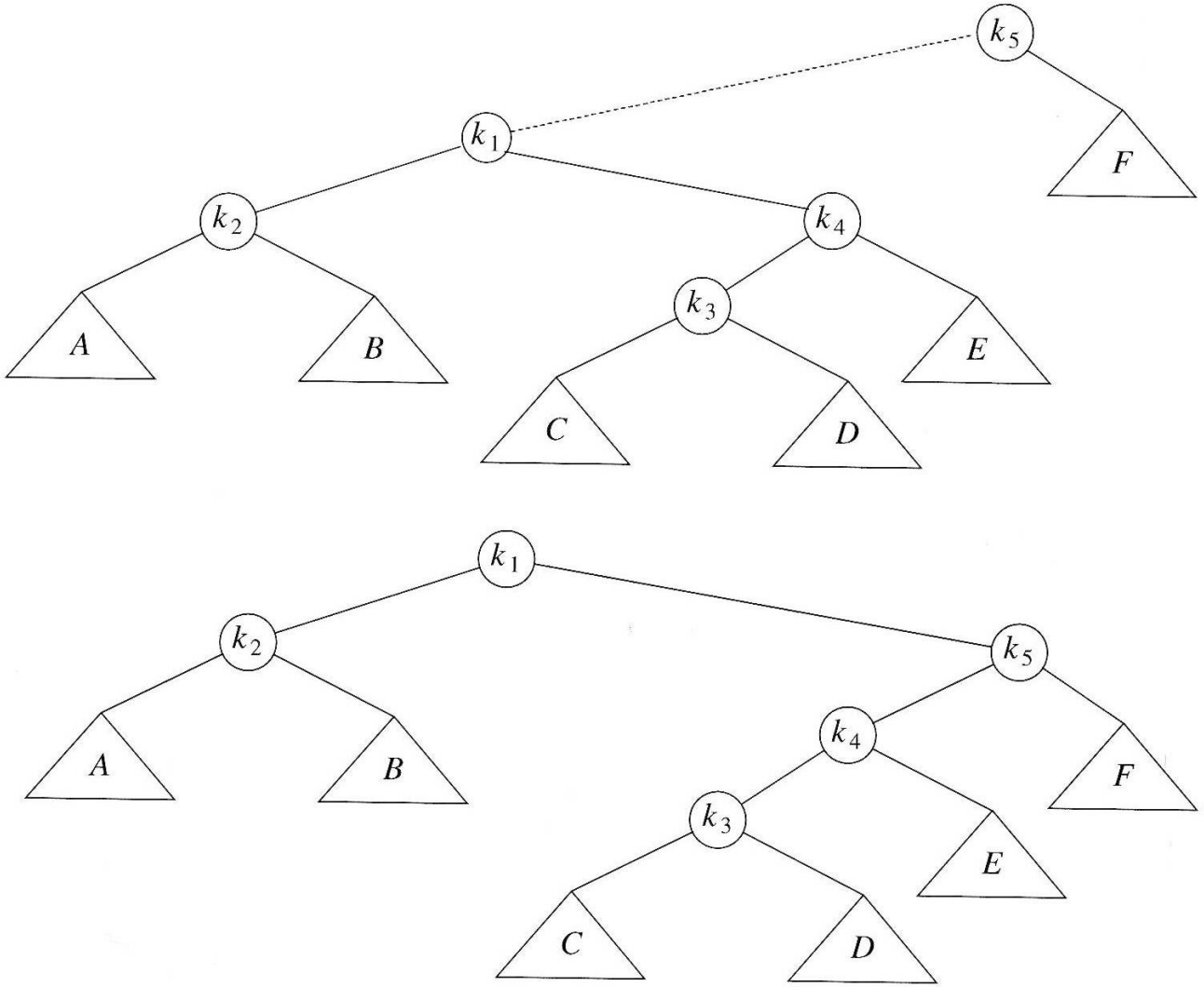
Rotation k_1 <-> k_3



[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.

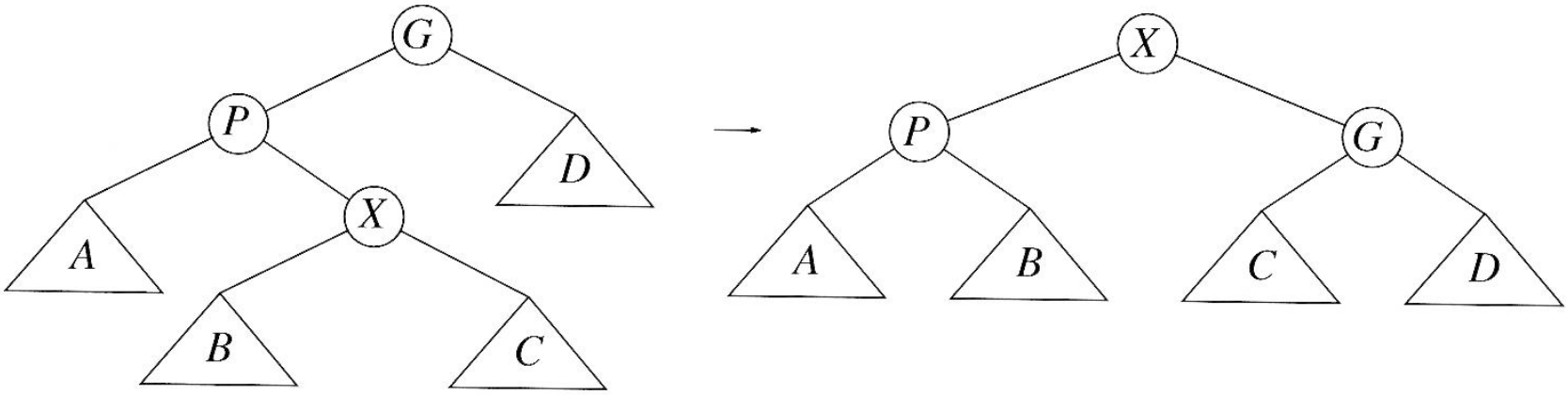
Aber: Ein anderer Knoten (k3) wurde dafür fast bis unten gedrückt.

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.

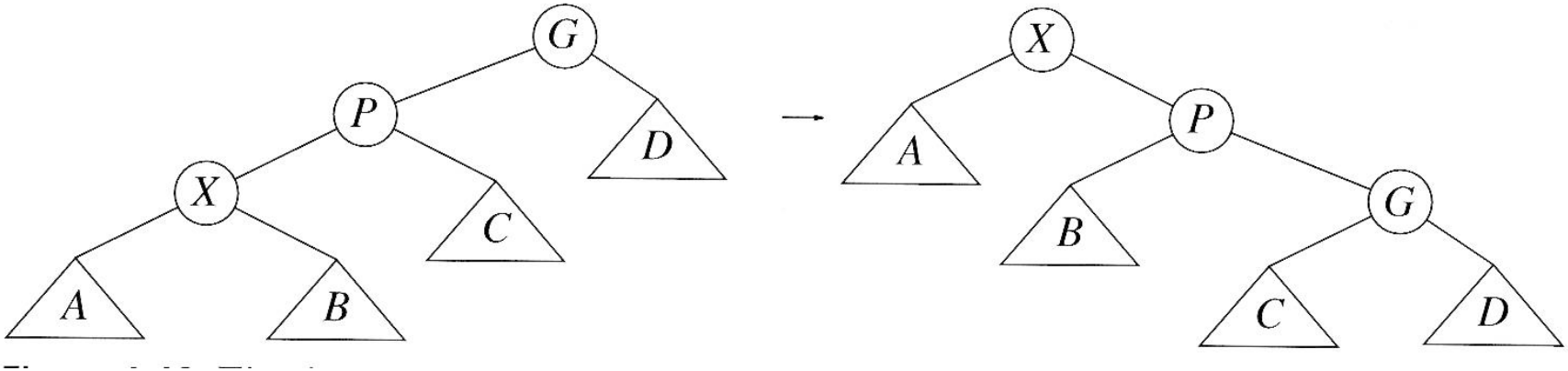
=> Baum muß so umstrukturiert werden, das er möglichst „breit“ wird (splaying).

Splaying-Strategie

2 Regeln (+ 2 Spiegelbilder):



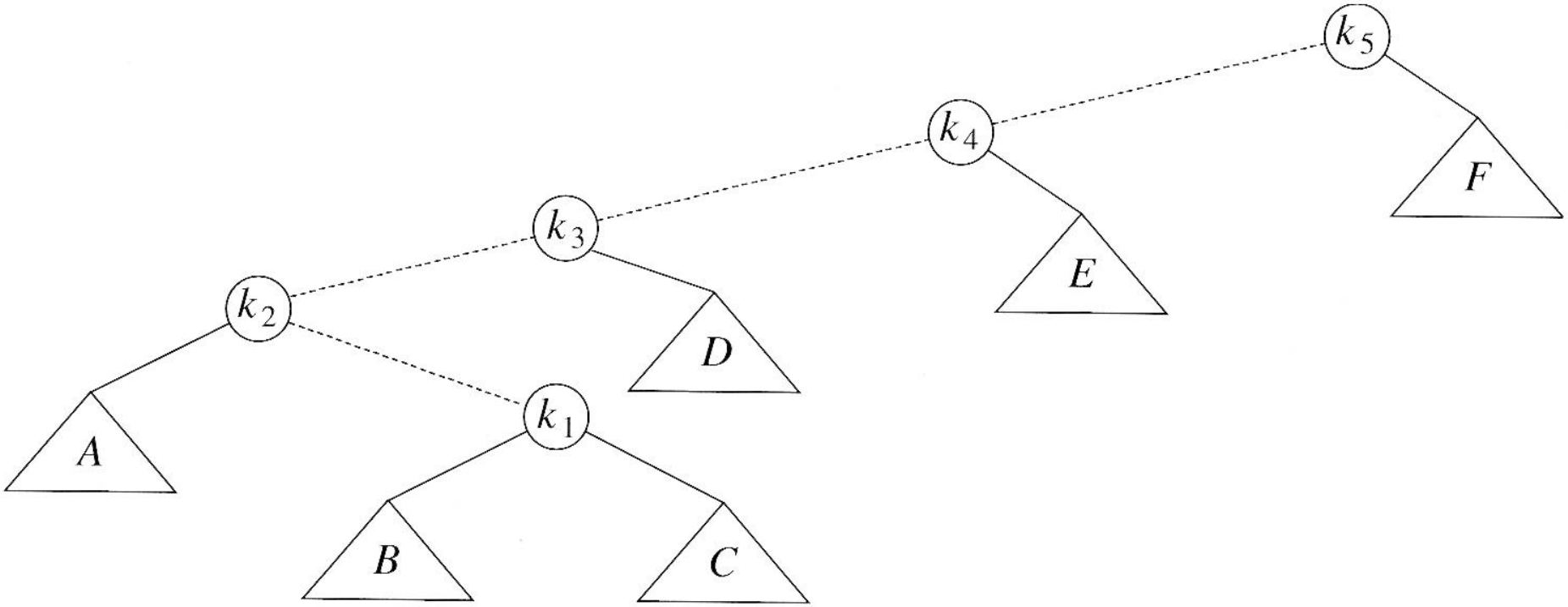
„Zick-Zack“ => Doppelte Rotation wie im AVL-Baum



„Zick-Zick“

[Weiss, 1999]

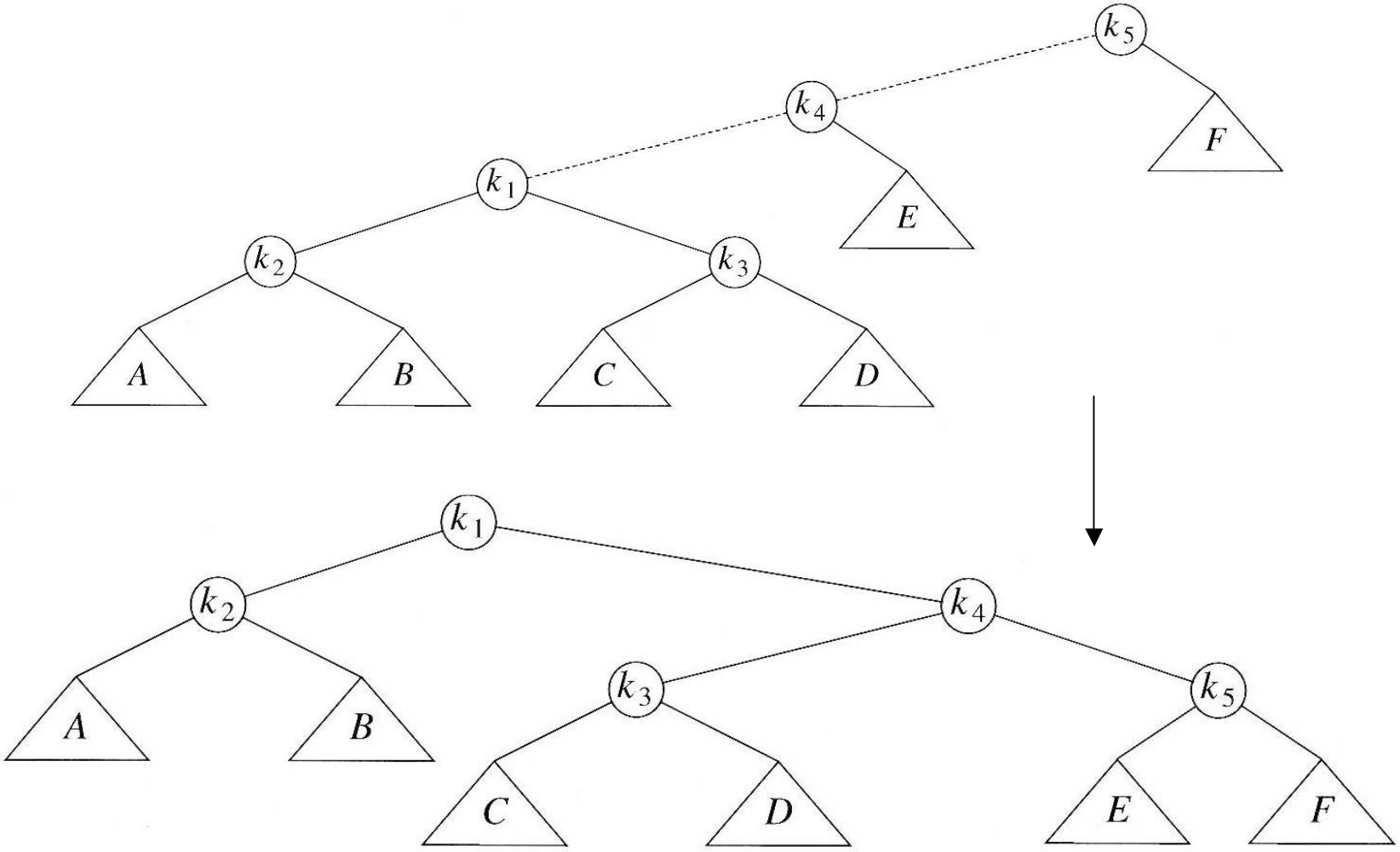
Splaying: Beispiel (1)



„Zick-Zack“

[Weiss, 1999]

Splaying: Beispiel (2)

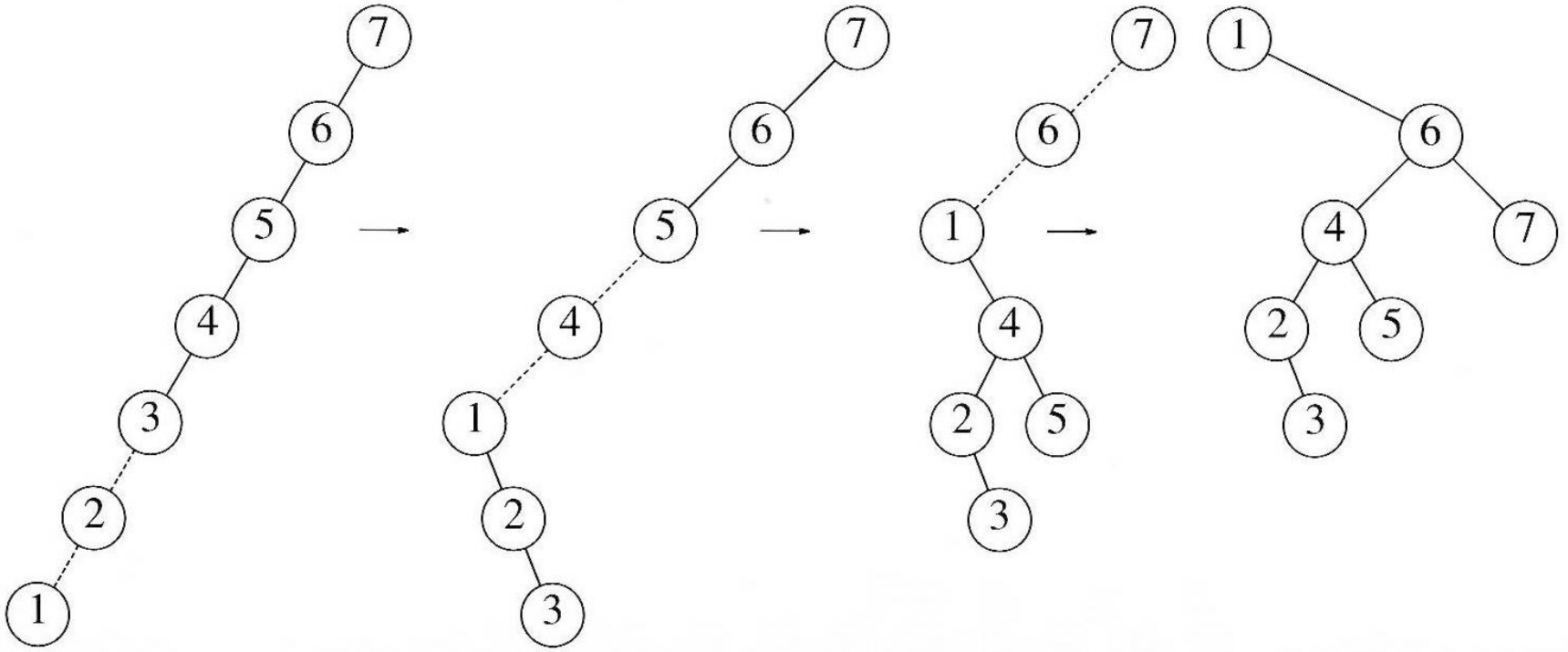


„Zick-Zick“

[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.

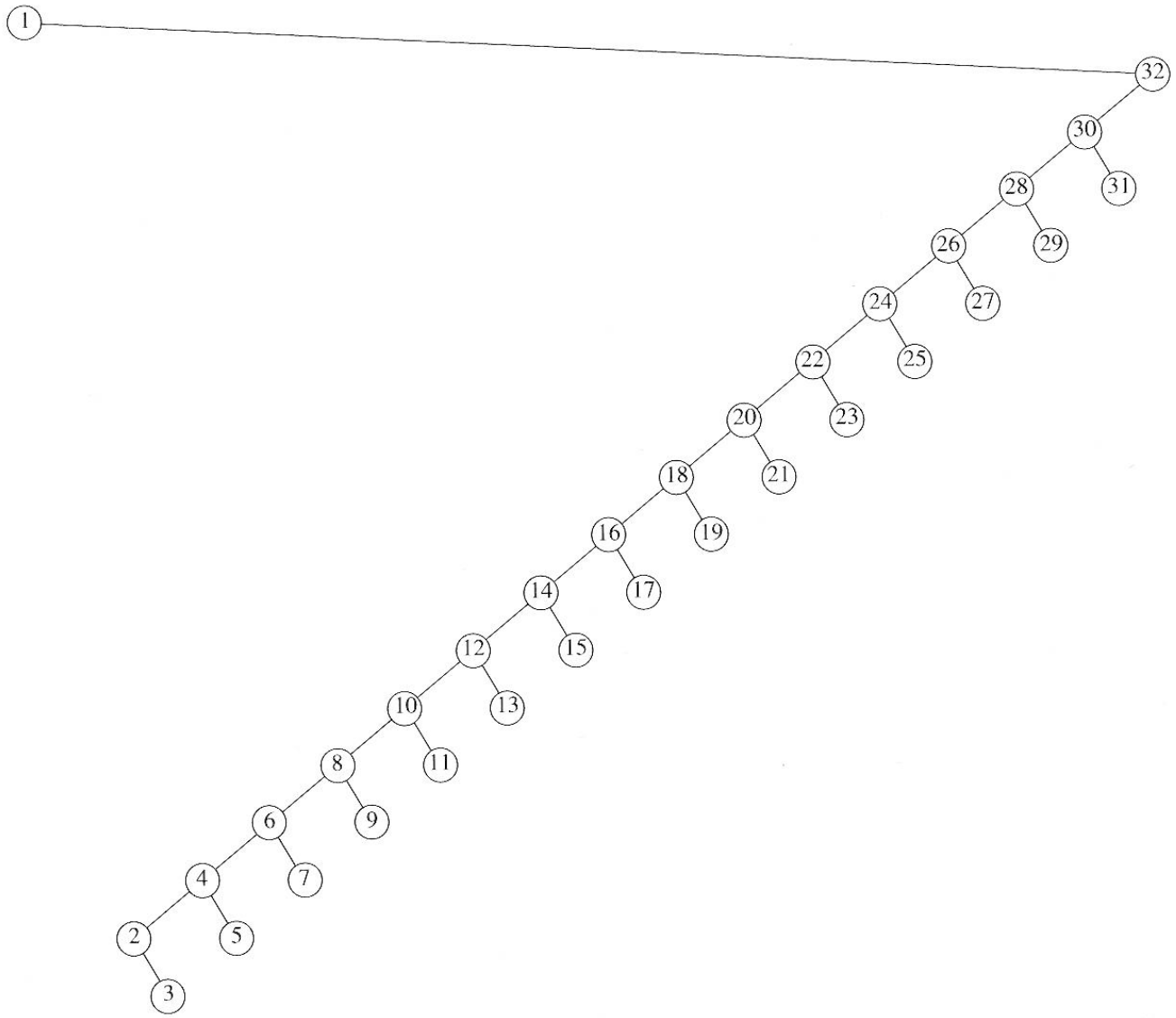
Bei einer Worst-Case-Laufzeit von O(M f(N)) ist die amortisierte Laufzeit O(f(N)) pro Operation.

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.

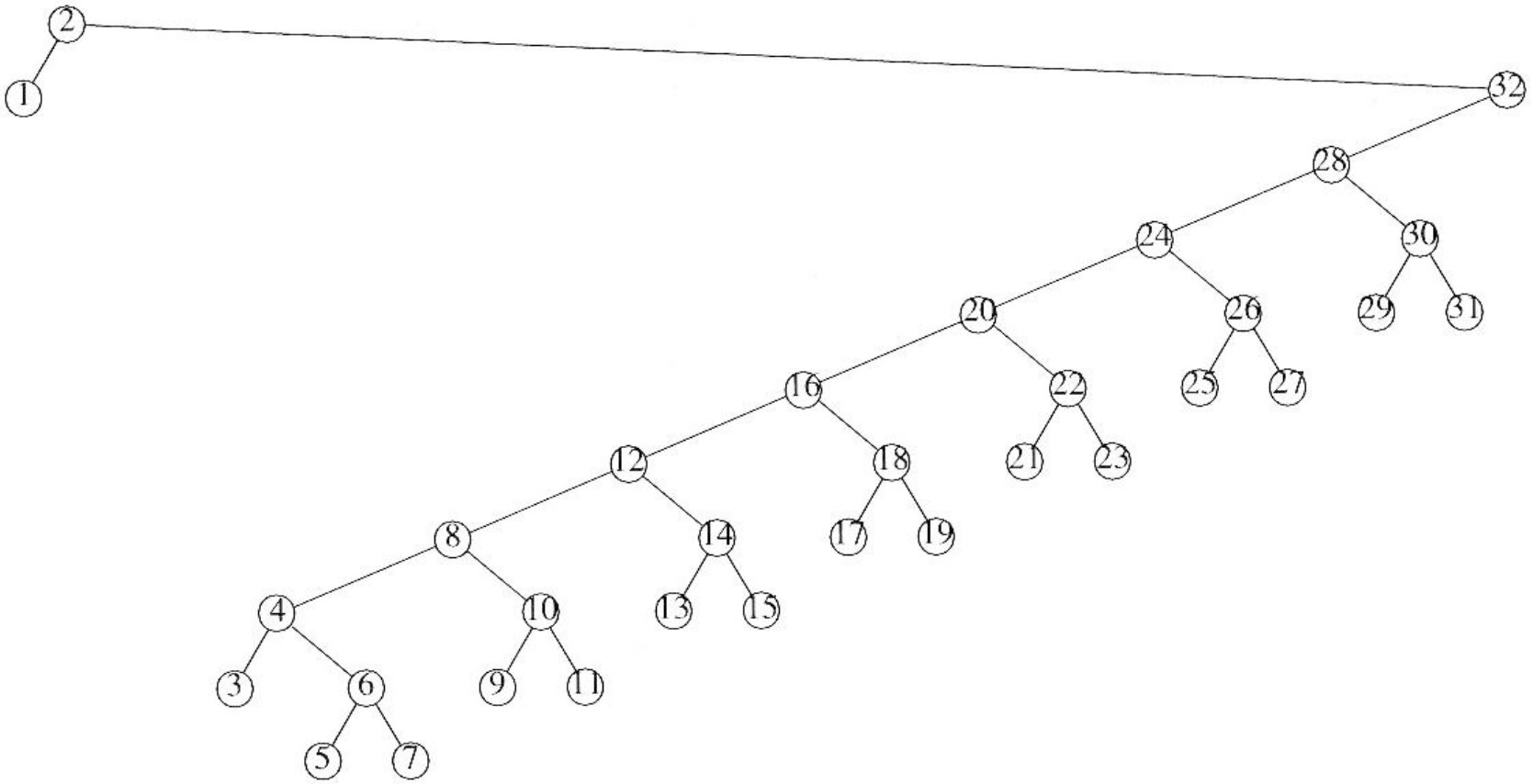
Beispiel für Worst-Case (1)



Splaying an Knoten 1 an einem Baum aus linken Kindern

[Weiss, 1999]

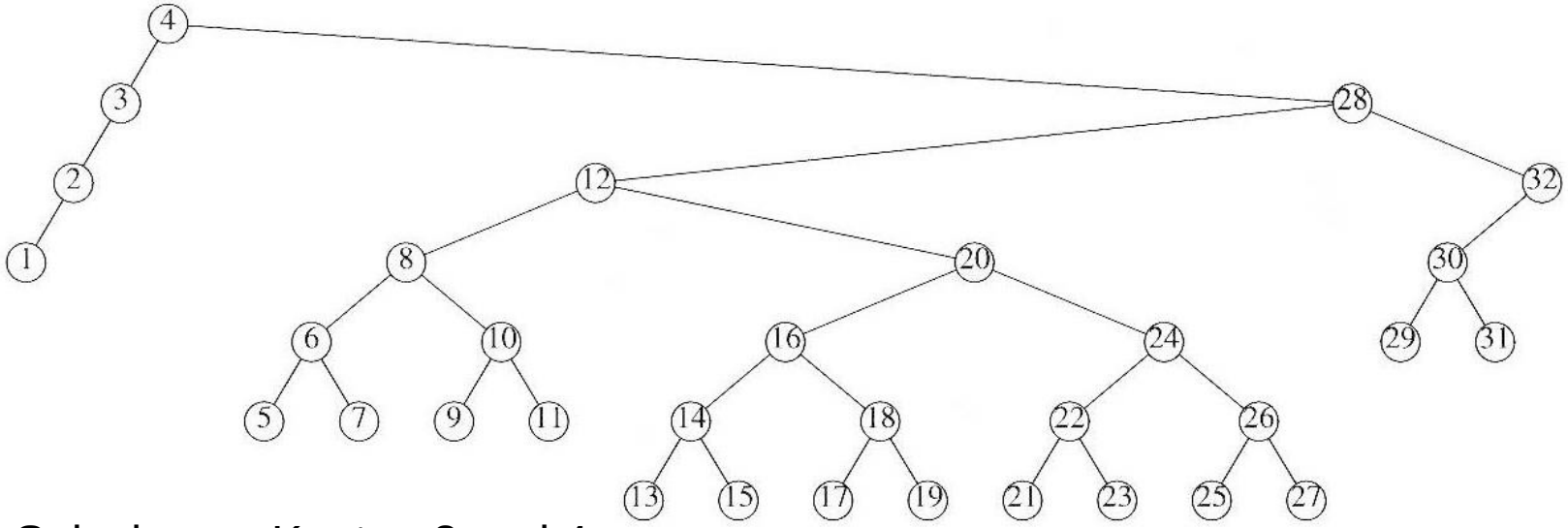
Beispiel für Worst-Case (2)



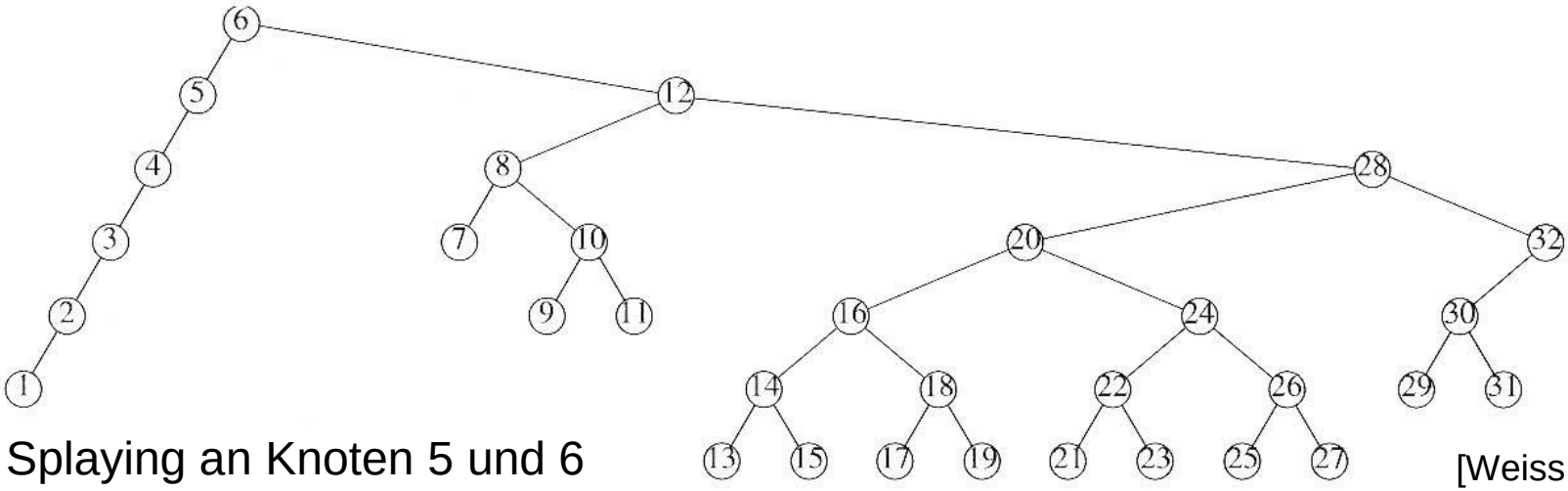
nach Splaying an Knoten 2

[Weiss, 1999]

Beispiel für Worst-Case (3)



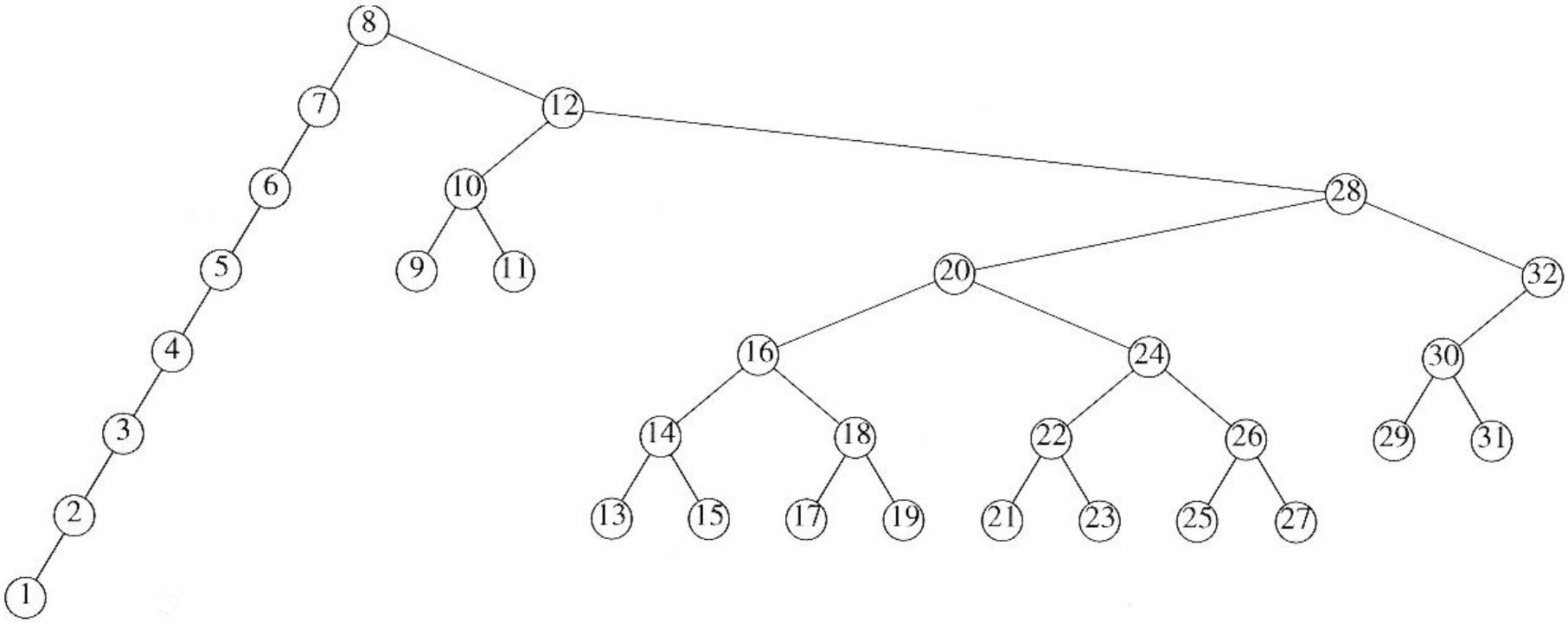
nach Splaying an Knoten 3 und 4



nach Splaying an Knoten 5 und 6

[Weiss, 1999]

Beispiel für Worst-Case (4)



nach Splaying an Knoten 7 und 8

[Weiss, 1999]