Algorithmen und Datenstrukturen

Teil 10:
Suchen (I) – Felder und Bäume

DHBW Stuttgart Campus Horb
Fakultät Technik
Studiengang Informatik
Dozent: Olaf Herden
Stand: 06/2020

Gliederung

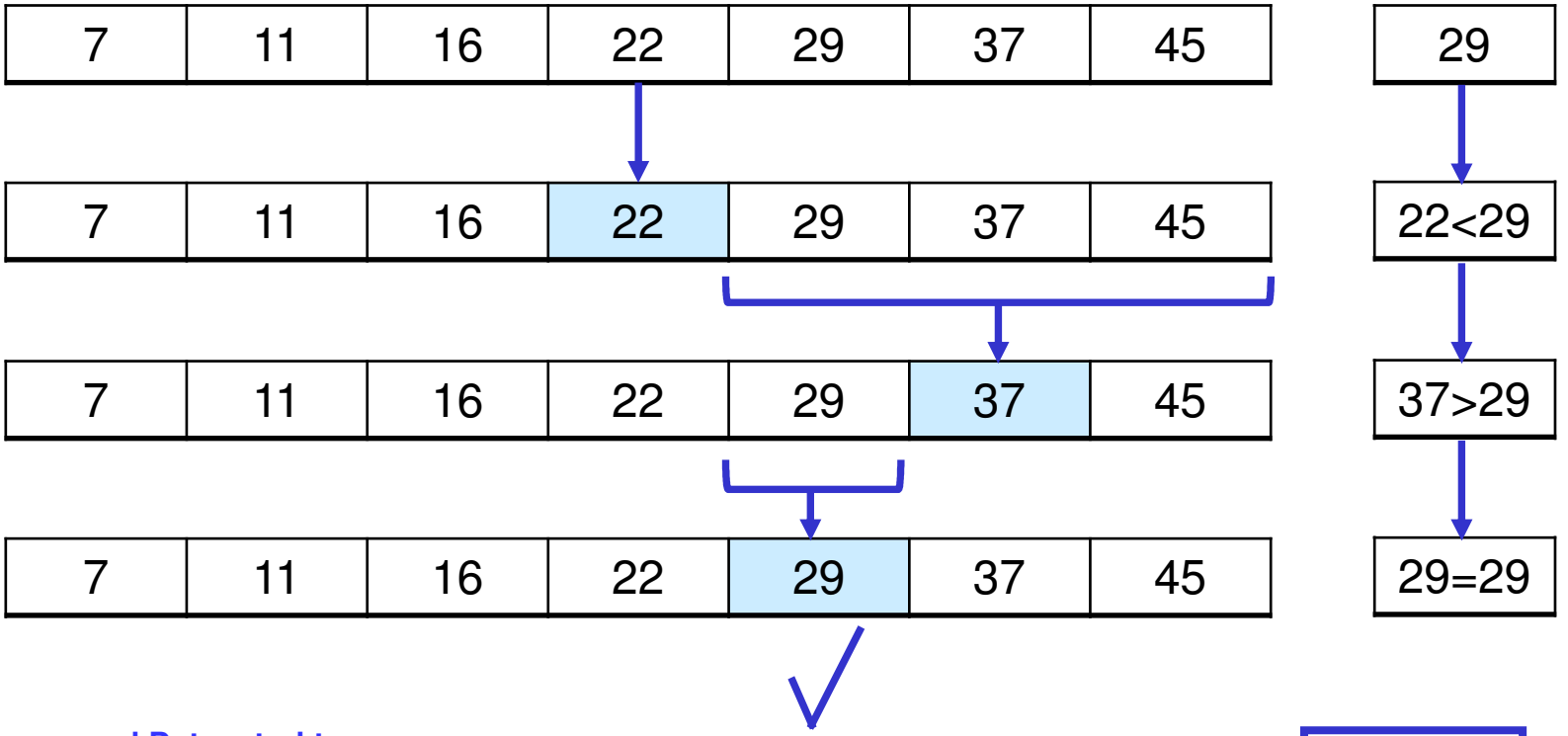
- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

Suchen in Feldern

- Suchen in Feld ohne weitere Annahmen
- Vorgehen:
  - Beginnen beim ersten Element
  - Bis zum Ende durchgehen
- Komplexität:
  - Maß: Anzahl von Vergleichen
  - Best case (Gesuchtes Element an erster Position): O(1)
  - Worst case (Gesuchtes Element an letzter Position oder nicht im Feld): O(n)
  - Average case: Im Mittel sind n/2 Vergleiche notwendig, falls gesuchtes Element im Feld $\Rightarrow$ O(n)

Binäres Suchen (I)

- Ist Feld sortiert, kann binäres Suchen angewendet werden:
  - Zunächst gesuchtes Element e mit mittlerstem vergleichen
  - Ist e größer, Vorgehen im rechten Teilfeld fortsetzen, ansonsten im linken
  - Verfahren fortsetzen, bis Element gefunden oder einelementiges Feld übrigbleibt
- Beispiel:



Binäres Suchen (II)

- Komplexität:
  - Maß: Anzahl Vergleiche
  - Best case (Gesuchtes Element an mittlerster Position): O(1)
  - Worst case (Gesuchtes Element nicht im Feld):
    - Frage: Wann ist einelementiges Feld erreicht?
    - Mit jedem Element halbiert sich die Anzahl der Element im relevanten Feld: n, n/2, n/4, n/8, ... $\Rightarrow$ Nach $\log_2 n$ Schritten ist einelementiges Feld erreicht $\Rightarrow O(\log_2 n)$
  - Average case: Im Mittel halb so viele Vergleiche notwendig wie im worst case, falls gesuchtes Element im Feld enthalten ist $\Rightarrow O(\log_2 n)$

Vergleich lineare vs. binäre Suche (I)

- Effizienzsteigerung:

Anzahl Elemente	Maximale Anzahl Vergleiche Lineare Suche	Maximale Anzahl Vergleiche Binäre Suche
1.000	1.000	10
2.000	2.000	11
5.000	5.000	13
1.000.000	1.000.000	20
2.000.000	2.000.000	21
5.000.000	5.000.000	23
1.000.000.000	1.000.000.000	30
2.000.000.000	2.000.000.000	31
5.000.000.000	5.000.000.000	33

Vergleich lineare vs. binäre Suche (II)

- Zusammenfassung Komplexitäten:

	Best Case	Average Case	Worst Case
Lineare Suche	O(1)	O(n)	O(n)
Binäre Suche	O(1)	$O(\log_2 n)$	$O(\log_2 n)$

- Fazit: Binäre Suche in großen Datenbeständen deutlich schneller
- Allerdings: Vernachlässigung von Kosten für:
  - Sortierung
  - Einfügen an richtiger Stelle
  - Schließen von Lücken beim Löschen (Umspeichern des Feldes)

Interpolationssuche

- Weitere Verbesserung binärer Suche
- Idee:
  - Verbesserung binärer Zugriff durch Ersetzen des Teilungsfaktors ½ durch angepassten Interpolationswert m
  - Annahme: Datensätze <d_1, ..., d_n> sind bezgl. ihrer Schlüsselwerte $s(d_1)$, ..., $s(d_n)$ etwa gleich verteilt
  - Ermittlung erwartete Position m mit gesuchtem Schlüsselwert s:

$$m = 1 + \left(\frac{s - s(d_1)}{s(d_n) - s(d_1)} \right) \cdot (n-1)$$

  - Analogie: Bei Suchen von „Zausel“ im Telefonbuch weiter hinten aufschlagen, bei „Abel“ recht weit vorne und bei „Meier“ etwa in der Mitte
- Eigenschaften:
  - Im Durchschnitt liegt der Aufwand bei $O(\log_2(\log_2(n)))$
  - Im worst case liegt der Aufwand bei O(n)

Suchen in dynamischen Strukturen

- Listen müssen (unabhängig von Sortierung) immer von vorne nach hinten durchsucht werden
- Komplexität:
  - Maß: Anzahl von Vergleichen
  - Best case (Gesuchtes Element steht an erster Position): O(1)
  - Worst case und Average case: O(n)
  - Zusätzlich: Konstanten für die Navigation (Umhängen der Zeiger)

Übersicht

- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

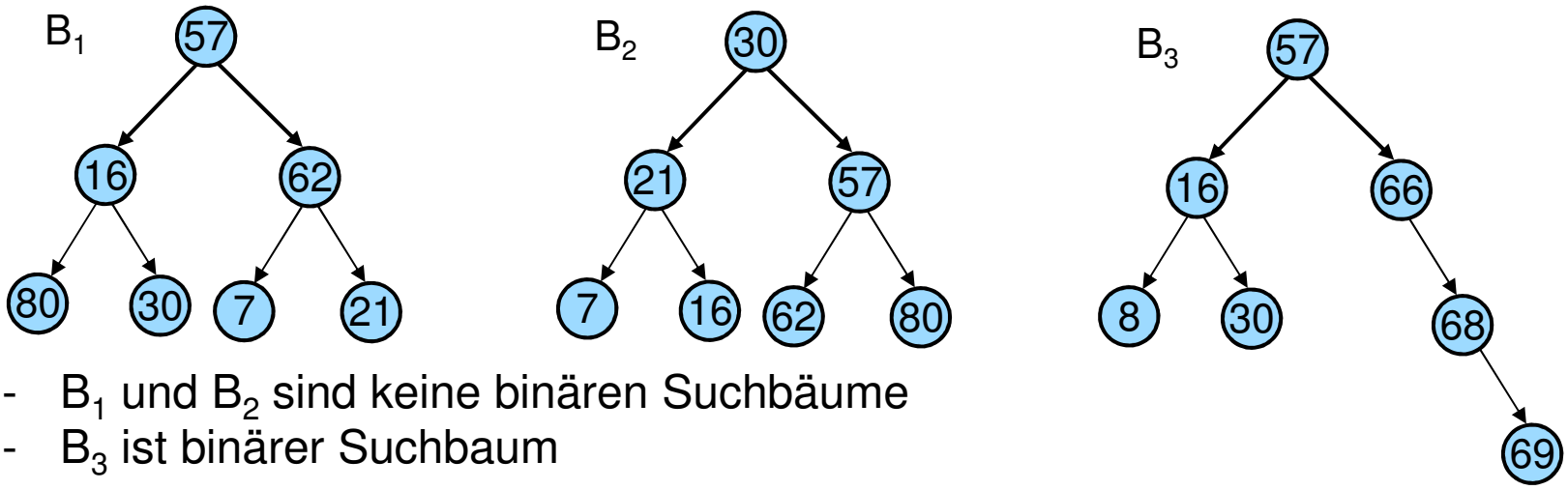
Motivation

- Ziele:
  - Effiziente Verarbeitung großer geordneter Datenbestände
  - Insbesondere effiziente Aufsuchoperationen
  - Sortierte Verarbeitung des gesamten Datenbestandes
- Annahmen:
  - Zugriff auf Datensätze erfolgt über Schlüssel
  - Auf den Schlüsselwerten ist eine Ordnung definiert
  - Schlüsselwerte sind eindeutig im gesamten Datenbestand
  - Letzte Annahme vereinfacht die Darstellung etwas, durch leichte Modifikation der Algorithmen kann diese Forderung aufgehoben werden

Definition

- Binärbaum B heißt binärer Suchbaum, wenn
  - B ist leer

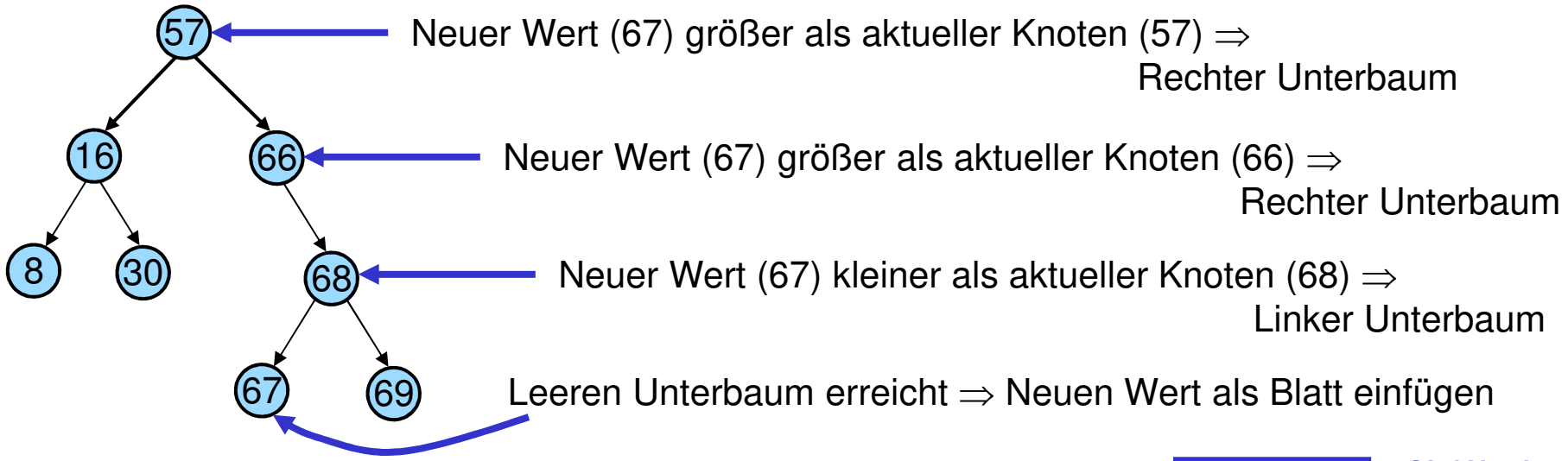
  oder jeder Knoten in B enthält einen Schlüssel mit:
  - Alle Schlüssel im linken Unterbaum von B sind kleiner als der Schlüssel in der Wurzel von B
  - Alle Schlüssel im rechten Unterbaum von B sind größer als der Schlüssel in der Wurzel von B
  - Linker und rechter Unterbaum von B sind jeweils auch binäre Suchbäume.
- Beispiele:



  - B_1 und B_2 sind keine binären Suchbäume
  - B_3 ist binärer Suchbaum

Einfügen eines Knotens

- Neue Knoten werden immer als Blätter eingefügt
- Position des Blattes wird durch den Schlüssel des neuen Knotens festgelegt
- Ist der Baum leer, so bildet der einzufügende Knoten die Wurzel
- Ist der Baum nicht leer, so wird an der Wurzel beginnend in den rechten (wenn neuer Schlüssel größer als Knotenwert ist) oder linken Unterbaum (wenn neuer Schlüssel kleiner als Knotenwert ist) gegangen, bis man bei einem leeren Unterbaum angekommen ist
- Beispiel: Füge 67 ein

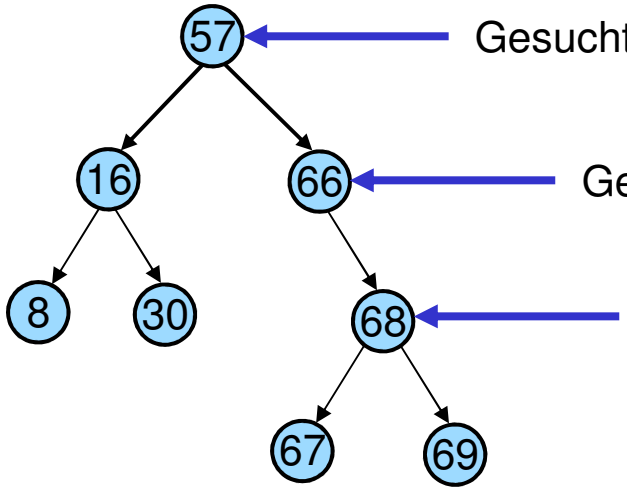


Anmerkungen

- Zu gegebener Schlüsselmenge gibt es eine Vielzahl von binären Suchbäumen
- Reihenfolge des Einfügens bestimmt hierbei Aussehen des binären Suchbaums
- Es gilt:
  - n Schlüssel $\Rightarrow$ n! verschiedene Schlüsselfolgen
  - Einige führen aber auf den gleichen binären Suchbaum
  - Bei n Schlüsseln gibt es $\frac{1}{n+1} \cdot \binom{2n}{n}$ verschiedene binäre Suchbäume
- Einfügereihenfolge beeinflusst maßgeblich die Höhe des Baums
- Schlimmster Fall bei sortierter Einfügereihenfolge: Binärer Suchbaum degeneriert zu linearer Liste
- Einfügealgorithmus sehr einfach, da keine Ausgleichs- oder Reorganisationsoperationen notwendig sind

Suchen eines Knoten (I)

- Position eines Knotens wird nach gleichem Verfahren wie beim Einfügen eines Knotens gesucht
- Methoden liefern eine Referenz auf den Knoten zurück, oder den Wert null bei nicht erfolgreicher Suche
- Beispiele: Suche den Wert 68



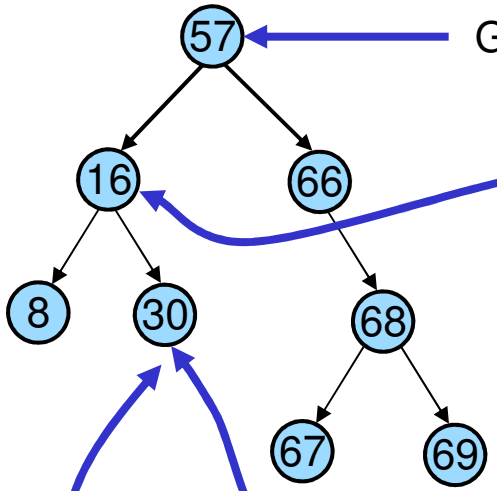
Gesuchter Wert (68) größer als aktueller Knoten (57) ⇒ Rechter Unterbaum

Gesuchter Wert (68) größer als aktueller Knoten (66) ⇒ Rechter Unterbaum

Gesuchter Wert (68) entspricht aktuellem Knoten
⇒ Liefere Verweis auf aktuellen Knoten als Ergebnis

Suchen eines Knoten (II)

- Beispiel: Suche den Wert 44



Gesuchter Wert (44) kleiner als aktueller Knoten (57) ⇒
Linker Unterbaum

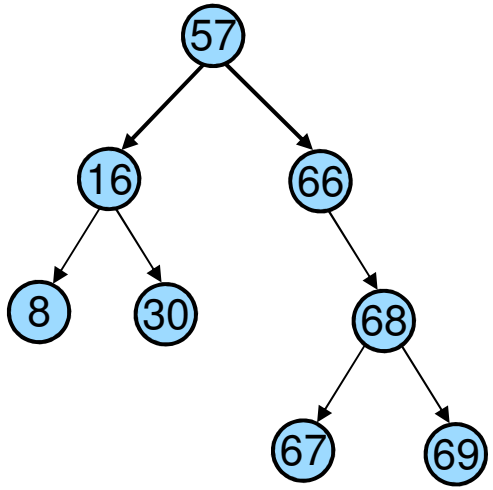
Gesuchter Wert (44) größer als aktueller Knoten (16) ⇒
Rechter Unterbaum

Gesuchter Wert (44) größer als aktueller Knoten (30)
⇒ Rechter Unterbaum

Aktueller Unterbaum ist leer
⇒ Gesuchtes Element nicht im Baum
⇒ Liefere null als Resultat zurück

Sortierte Ausgabe aller Knoten

- Sortierte Ausgabe aller Knoten wird durch Inorder-Durchlauf des binären Suchbaums erreicht
- Zur Erinnerung: Inorder-Durchlauf:
  - Besuche linken Unterbaum
  - Besuche Wurzel
  - Besuche rechten Unterbaum
- Beispiel:

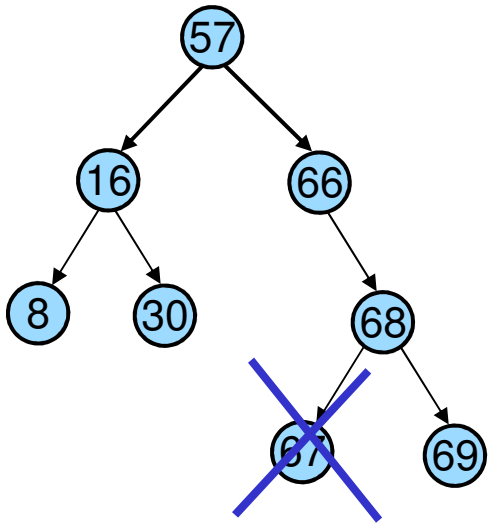


  - 8 16 30 57 66 67 68 69

Löschen eines Knoten (I)

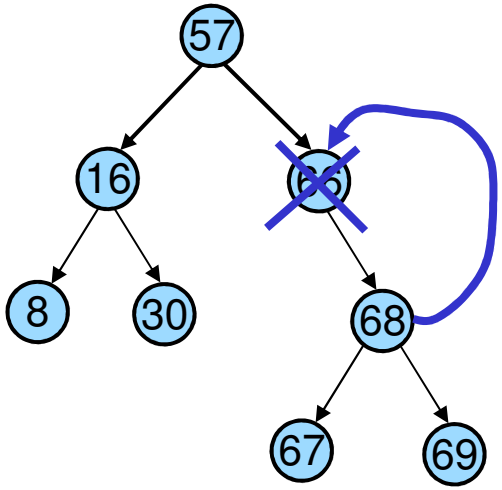
- Vorgehensweise in zwei Teilschritten:
  - Position des Knoten bestimmen (analog zum Vorgehen beim Einfügen)
  - Eigentliches Löschen des Knoten
  - Hierbei müssen drei Fälle unterschieden werden:
    (1) Zu löschender Knoten ist Blatt
    (2) Zu löschender Knoten besitzt linken oder rechten leeren Unterbaum
    (3) Zu löschender Knoten besitzt zwei nicht-leere Unterbäume

- Fall 1: Zu löschender Knoten ist Blatt
  - Blatt kann ohne weitere Maßnahmen entfernt werden
  - Beispiel: Lösche 67

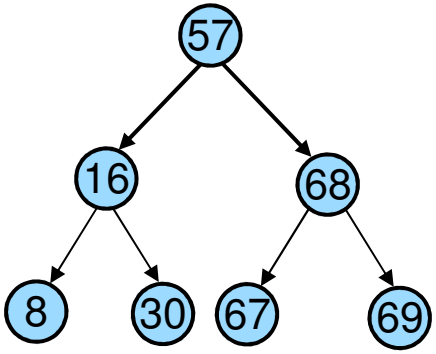


Löschen eines Knoten (II)

- Fall 2: Zu löschender Knoten besitzt linken oder rechten leeren Unterbaum
  - Zu löschender Knoten wird entfernt und durch die Wurzel des nicht-leeren Unterbaums ersetzt
  - Beispiel: Lösche 66

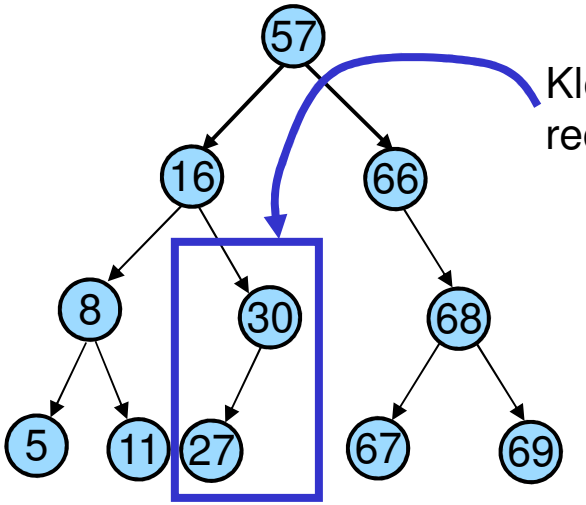


führt zu



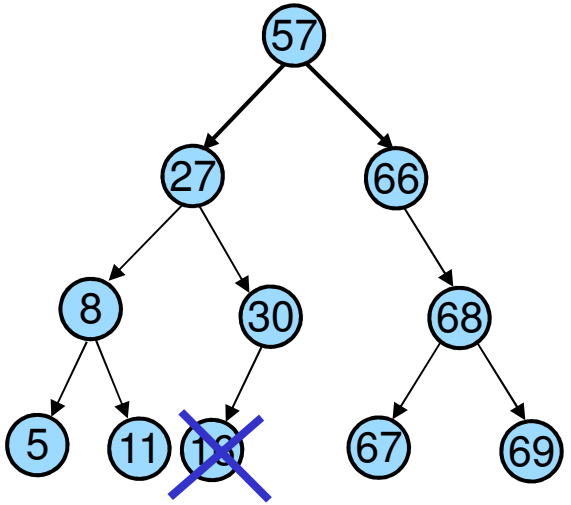
Löschen eines Knoten (III)

- Fall 3: Zu löschender Knoten x besitzt zwei nicht-leere Unterbäume
  - Schwierigste der drei Fälle: Wo sollen Unterbäume eingehängt werden?
  - Lösungsmöglichkeit 1:
    - Sei x‘ der Knoten mit dem kleinsten Schlüssel im rechten Unterbaum von x (Inorder-Nachfolger)
    - x‘ kann rechten Nachfolger haben, aber keinen linken
    - Vertausche die Werte von x und x‘
    - Lösche aktuellen Knoten x $\Rightarrow$ Fall 1 oder Fall 2
  - Beispiel: Lösche 16



Kleinstes Element im rechten Unterbaum ist 27.

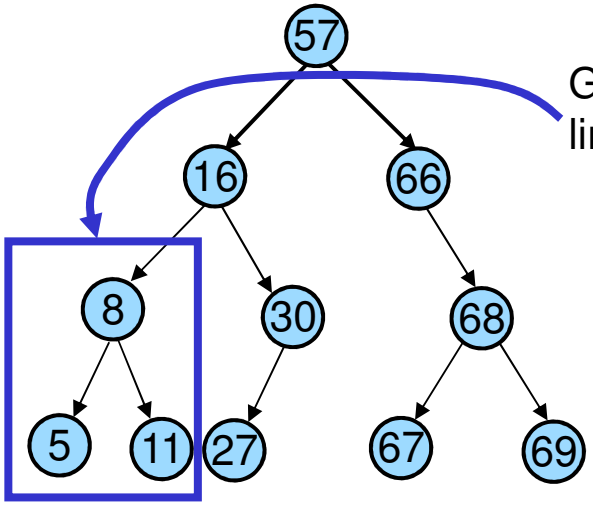
Vertauschen von 16 und 27 führt zu folgendem Baum



16 ist jetzt Blatt und kann gemäß Fall 1 gelöscht werden

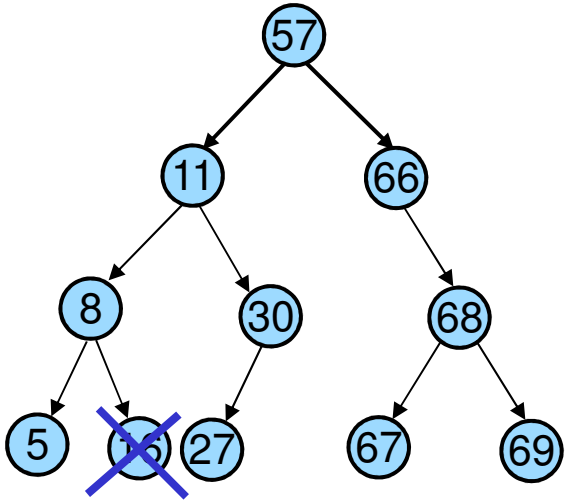
Löschen eines Knoten (IV)

- Fall 3 (Fortsetzung):
  - Lösungsmöglichkeit 2:
    - Sei x‘ der Knoten mit dem größten Schlüssel im linken Unterbaum von x (Inorder-Vorgänger)
    - x‘ kann linken Nachfolger haben, aber keinen rechten
    - Vertausche die Werte von x und x‘
    - Lösche aktuellen Knoten x $\Rightarrow$ Fall 1 oder Fall 2
  - Beispiel: Lösche 16



Größtes Element im linken Unterbaum ist 11

Vertauschen von 16 und 11 führt zu folgendem Baum



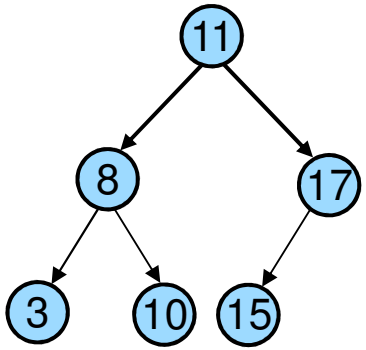
16 ist jetzt Blatt und kann gemäß Fall 1 gelöscht werden

Aufwand der Operationen

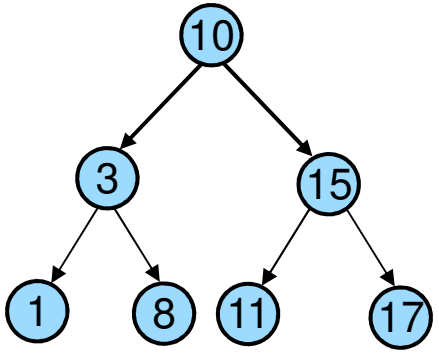
- Ausgabe aller Knoten:
  - Jeder Knoten muss einmal besucht werden $\Rightarrow$ O(n)
- Für Einfügen, Suchen, Löschen:
  - Maß für die Komplexität: Anzahl der Vergleiche
  - Aufwand ist maximal jeweils der Weg von Wurzel zu Blatt
  - Damit ist Aufwand entscheidend von der Höhe des Baums abhängig
  - Worst case: Baum ist lineare Liste $\Rightarrow$ O(n)
  - Best case: Baum ist ausgeglichen $\Rightarrow$ $O(\log_2 n)$
  - Average case: Hierfür kann man beweisen, dass nur ein mittlerer Mehraufwand von ca. 39% gegenüber dem best case notwendig ist
  - Aber: Keine Garantie hierfür, was aber in vielen Anwendungen gewünscht ist
- Idee: Bäume bei Einfügen und Löschen durch Reorganisationsoperationen so gestalten, dass sie immer (möglichst) ausgeglichen sind

Ausgeglichene binäre Suchbäume

- Reorganisationsoperationen kosten natürlich auch Aufwand
- Daher muss abgeklärt werden, ob dieser gerechtfertigt ist
- Beispiel: Einfügen von 1



führt unter Bewahren der Ausgeglichenheit zu



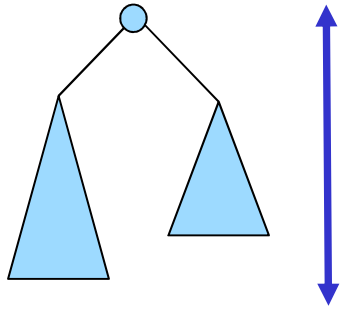
- Dies ist ein worst case-Szenario: Alle Knoten müssen verschoben werden, was Aufwand O(n) bedeutet
- Damit übersteigen die Reorganisationskosten die eigentlichen Suchkosten (O(n) gegenüber $O(\log_2 n)$)
- Fazit: Ausgeglichene Bäume sind in der Praxis nicht geeignet!

Balancierte binäre Suchbäume (I)

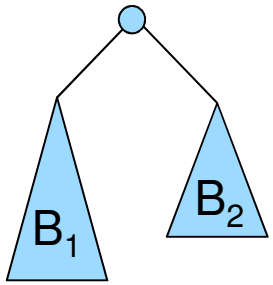
- Balancierte binäre Suchbäume als „fast ausgeglichene“ binäre Suchbäume
- Stellen Kompromiss zwischen ausgeglichenen binären Suchbäumen und natürlichen binären Suchbäumen dar
- Gewisse Abweichungen von der Ausgeglichenheit werden erlaubt, ohne das Zugriffsverhalten total zu zerstören
- Reorganisation darf nicht mehr global den ganzen Baum betreffen
- Transformationen dürfen nur noch lokal auf dem Pfad von der Wurzel bis zur Position der Änderung wirken
- Durch diese Einschränkung erreicht man, dass auch der Reorganisationsaufwand maximal $O(\log_2 n)$ beträgt

Balancierte binäre Suchbäume (II)

- „Fast ausgeglichen“ etwas formaler: Für alle Knoten im Baum soll gelten, dass Anzahl der Knoten im linken Unterbaum „ungefähr gleich“ Anzahl der Knoten im rechten Unterbaum entspricht
- Zwei Ansätze, um dies einzuhalten:
  - Höhenbalancierte Bäume
    - Differenz der Höhen beider Unterbäume ist beschränkt
    - Beispiele:
      - AVL-Baum (Adelson-Velsky und Landis)
      - HB-Baum



  - Gewichtsbalancierte Bäume
    - Differenz zwischen Knotenmengen beider Unterbäume muss beschränkt sein
    - Beispiel:
      - BB(α)-Baum (Bounded Balance, α Faktor für Gleichgewicht)



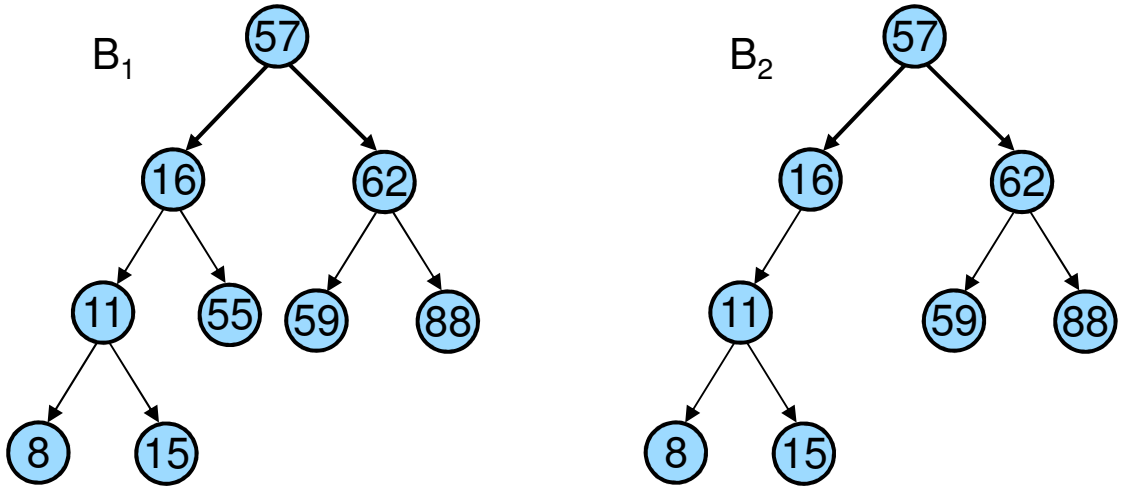
$|grad(B_1) - grad(B_2)| \leq Grenze$

Übersicht

- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

AVL-Bäume

- Definition: Ein binärer Suchbaum B heißt AVL-Baum, genau dann wenn für alle Knoten k von B gilt: Höhendifferenz der beiden Unterbäume von k ist höchstens 1.
- Beispiele:



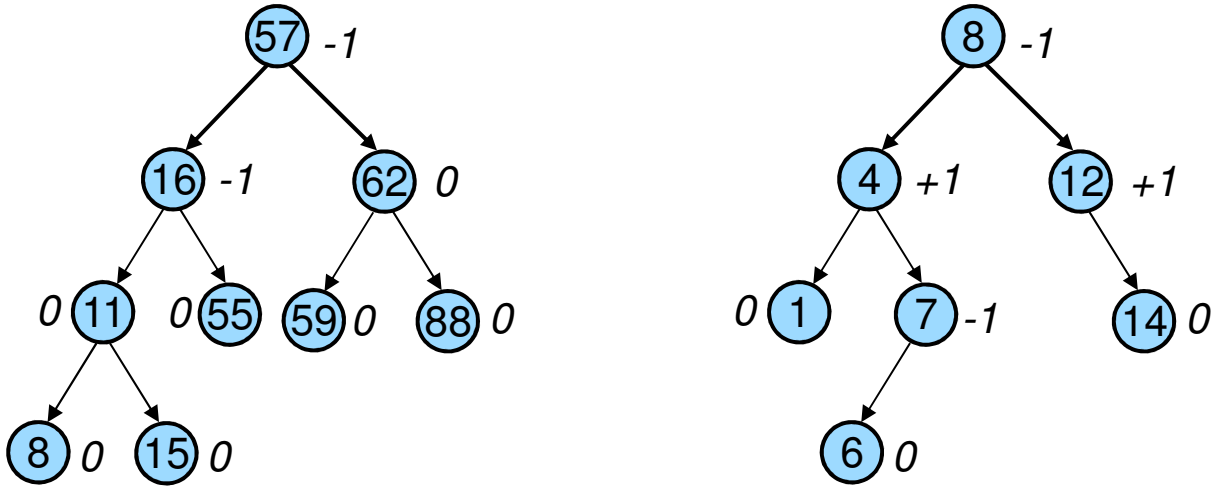
  - B_1 ist AVL-Baum
  - B_2 ist kein AVL-Baum, da Knoten 16 einen linken Unterbaum der Höhe 2 und einen rechten Unterbaum der Höhe 0 hat

Balancefaktor in AVL-Bäumen

- Definition: Sei B AVL-Baum.

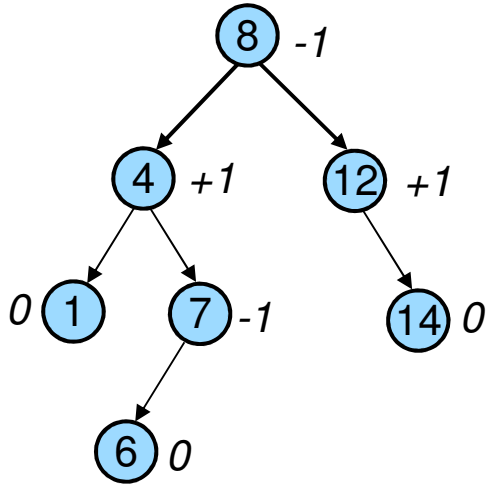
  Jedem Knoten k von B wird als Balancefaktor die Höhendifferenz zwischen linkem und rechtem Unterbaum von k zugeordnet. Mögliche Werte sind –1, 0 und 1.

- Beispiel:

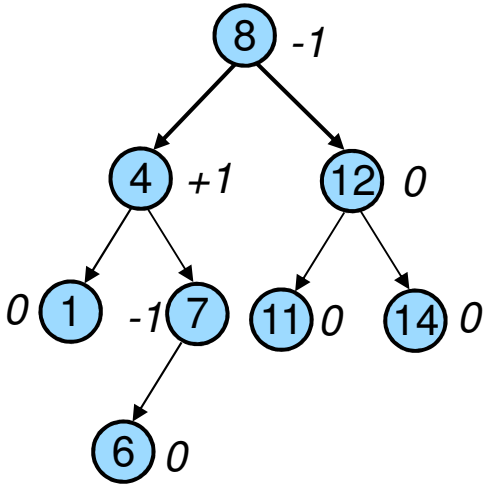


Einfügen in AVL-Bäumen: Problem

- Einfügen ist nicht so einfach wie in binären Suchbäumen
- Beispiel:

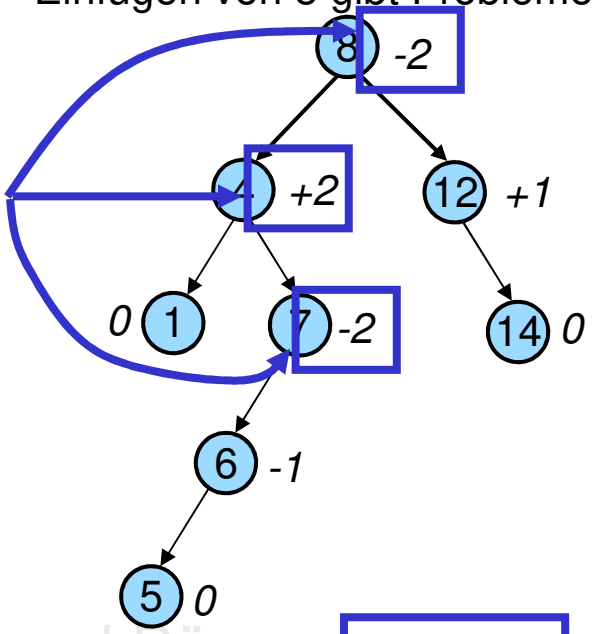


Einfügen von 11 problemlos:



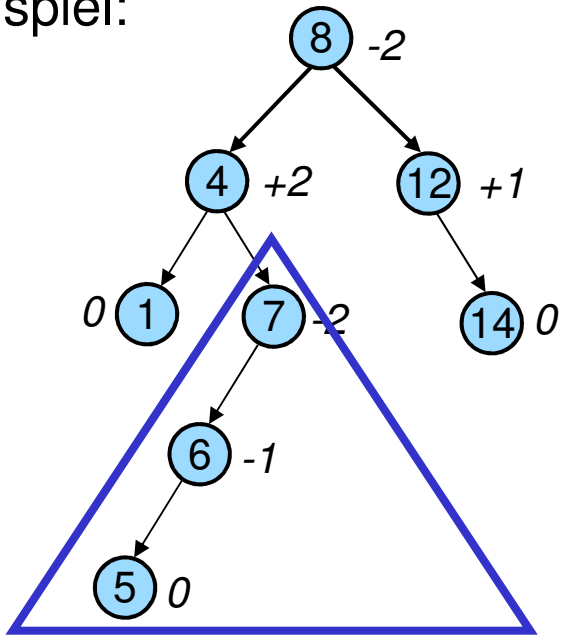
Einfügen von 5 gibt Probleme:

AVL-Eigenschaft verletzt

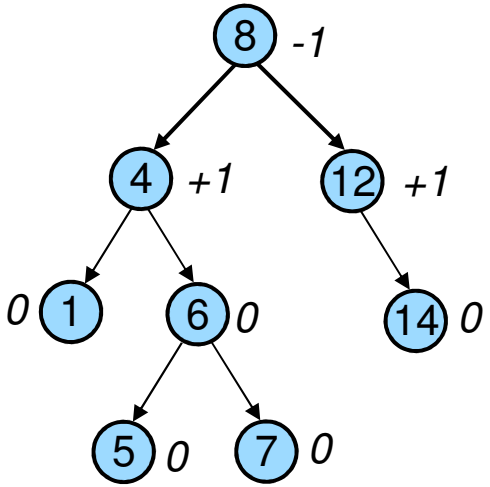


Einfügen in AVL-Bäumen: Problem

- Wie kann AVL-Eigenschaft wieder hergestellt werden?
- Beispiel:



führt zu



- Reorganisiere diesen Unterbaum
- Geschieht durch Rechtsverschiebung der Knoten
- Wird daher als Rechtsrotation (R-Rotation) bezeichnet

Einfügen in AVL-Bäumen

- Generelles Vorgehen zum Einfügen in AVL-Bäume:
  - Wie bei binären Suchbäumen wird neues Element x als Blatt eingefügt
  - Balancefaktor kann nur entlang des Weges von der Wurzel zur Einfügestelle verändert worden sein
  - Aufzusuchen ist der unterste Knoten k mit Balancefaktor 2 bzw. –2
  - Gibt es keinen solchen Knoten, ist AVL-Eigenschaft durch Einfügen nicht verloren gegangen
  - Gibt es einen solchen Knoten, dann unterscheide folgende Fälle:

    (1) Überhang im linken Unterbaum links von k

    (2) Überhang im mittleren Unterbaum links von k

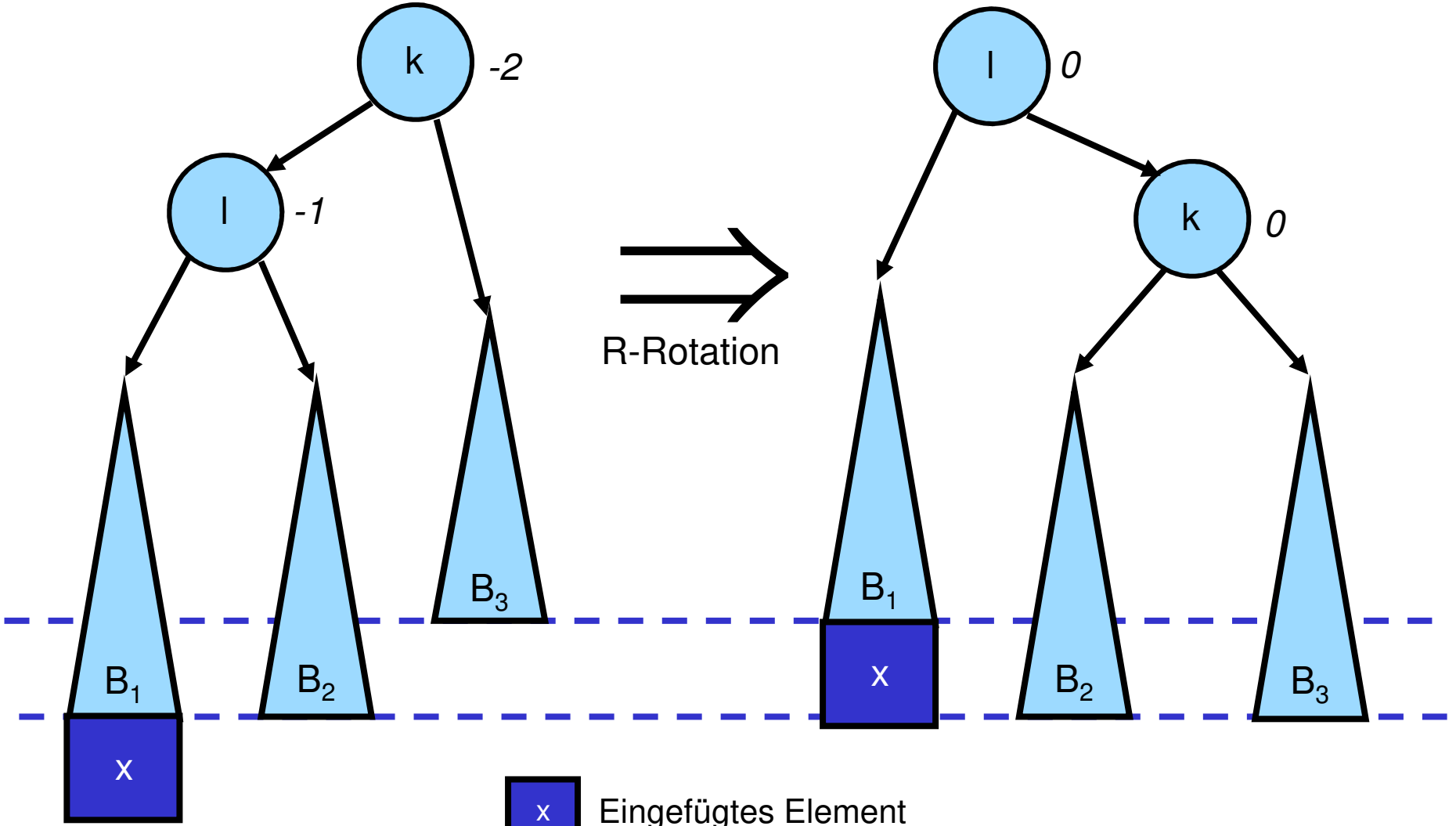
    Symmetrisch dazu:

    (3) Überhang im rechten Unterbaum rechts von k

    (4) Überhang im mittleren Unterbaum rechts von k
  - Anm.: In den Fällen (1) und (2) besitzt k jeweils den Wert –2, in den Fällen (3) und (4) den Wert 2

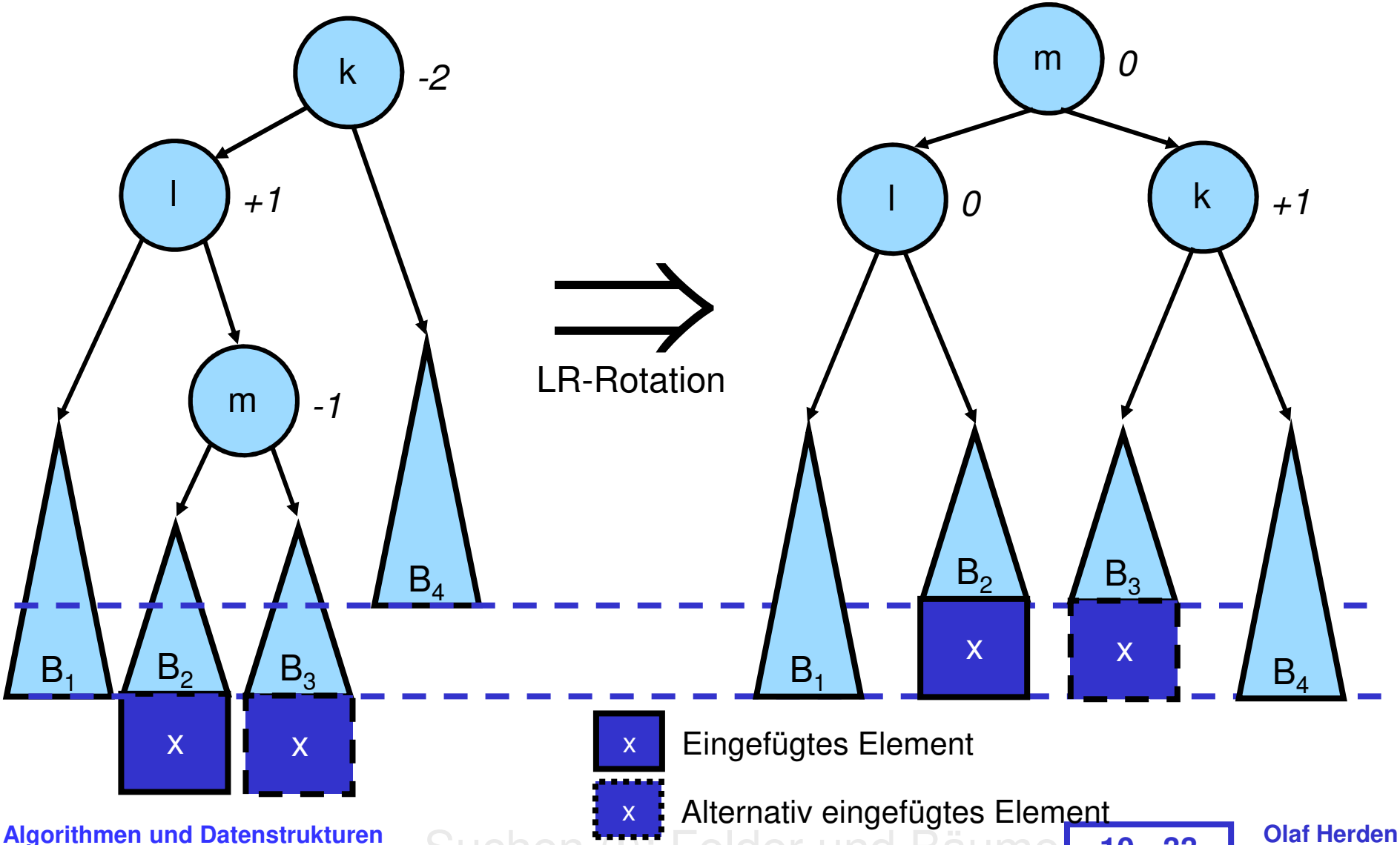
Fall 1: Überhang im linken Unterbaum links von k

- Situation ist links skizziert, Abhilfe schafft eine Rechts-Rotation



Fall 2: Überhang im mittleren Unterbaum links von k

- Links-Rechts-Rotation um Knoten l und dann um Knoten m

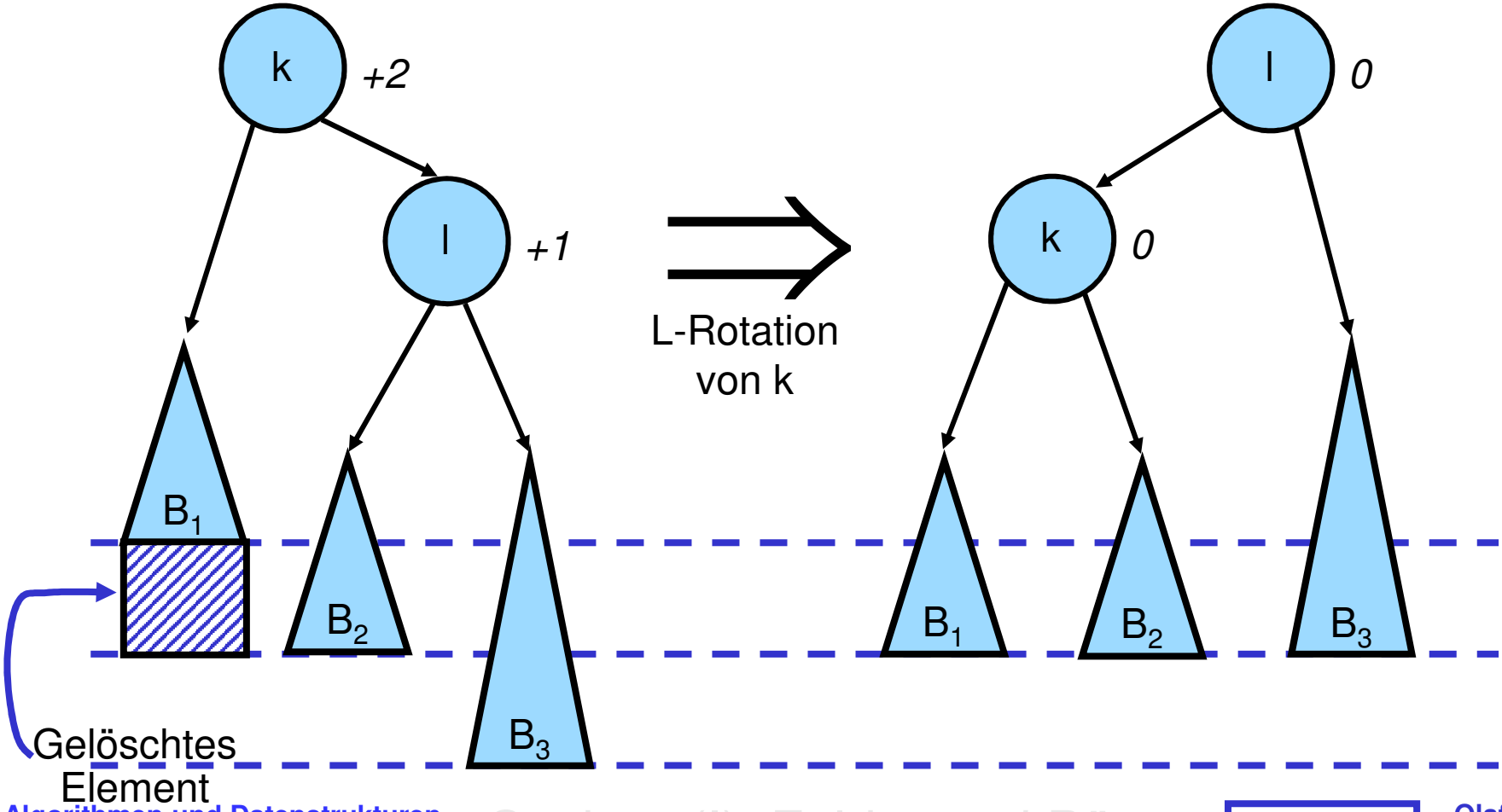


Löschen in AVL-Bäumen

- Geschieht zunächst analog zum Löschen in binären Suchbäumen (zu löschendes Element wird durch seinen Inorder-Nachfolger ersetzt)
- Beginnend an der untersten veränderten Stelle muss der Baum nach der Löschoperation längs des Pfades zur Wurzel ausgeglichen werden
- Hierbei sind im Gegensatz zum Einfügen u.U. mehrere Rotationen notwendig
- Folgende Fälle sind für einen Knoten k zu unterscheiden, dessen linker Unterbaum durch Löschen in der Höhe um 1 verringert wurde:
  - (1) Balancefaktor k = -1
    Setze Balancefaktor k = 0 und mache mit dem Vorgänger von k weiter
  - (2) Balancefaktor k = 0
    Setze Balancefaktor k = +1 und beende das Ausgleichen
  - (3) Balancefaktor k = +1 und l sei der rechte Nachfolger von k, unterscheide
    - a) Balancefaktor von l = +1
    - b) Balancefaktor von l = 0
    - c) Balancefaktor von l = -1
- Anm.: Symmetrisch für rechten Unterbaum

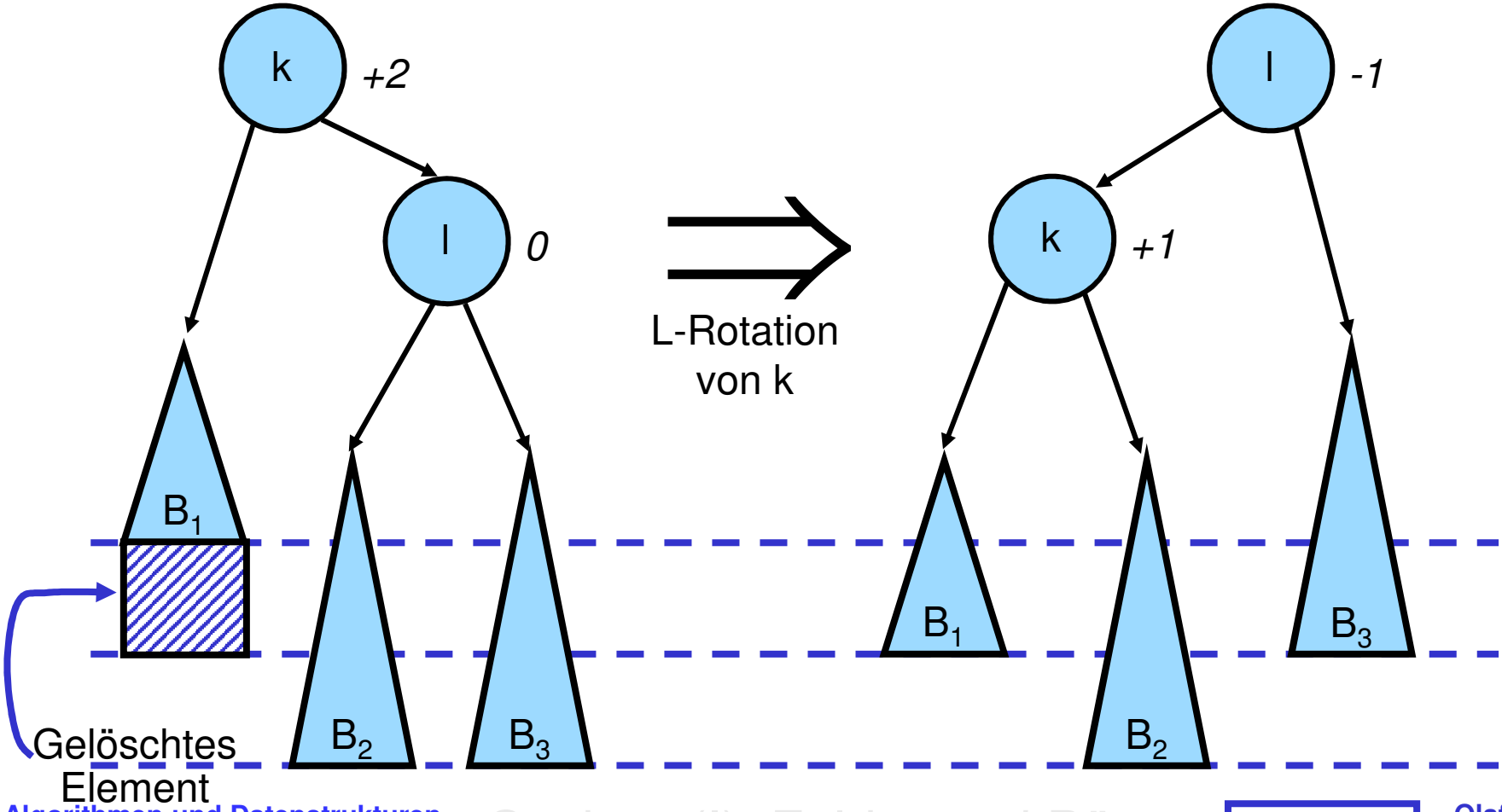
Löschen in AVL-Baum: Fall 3a

- Balancefaktor von l ist +1 $\Rightarrow$ Links-Rotation von k
- Gesamthöhe des Unterbaums unter k bzw. l um 1 erniedrigt
- Daher anschließend mit dem Vorgänger von l weitermachen



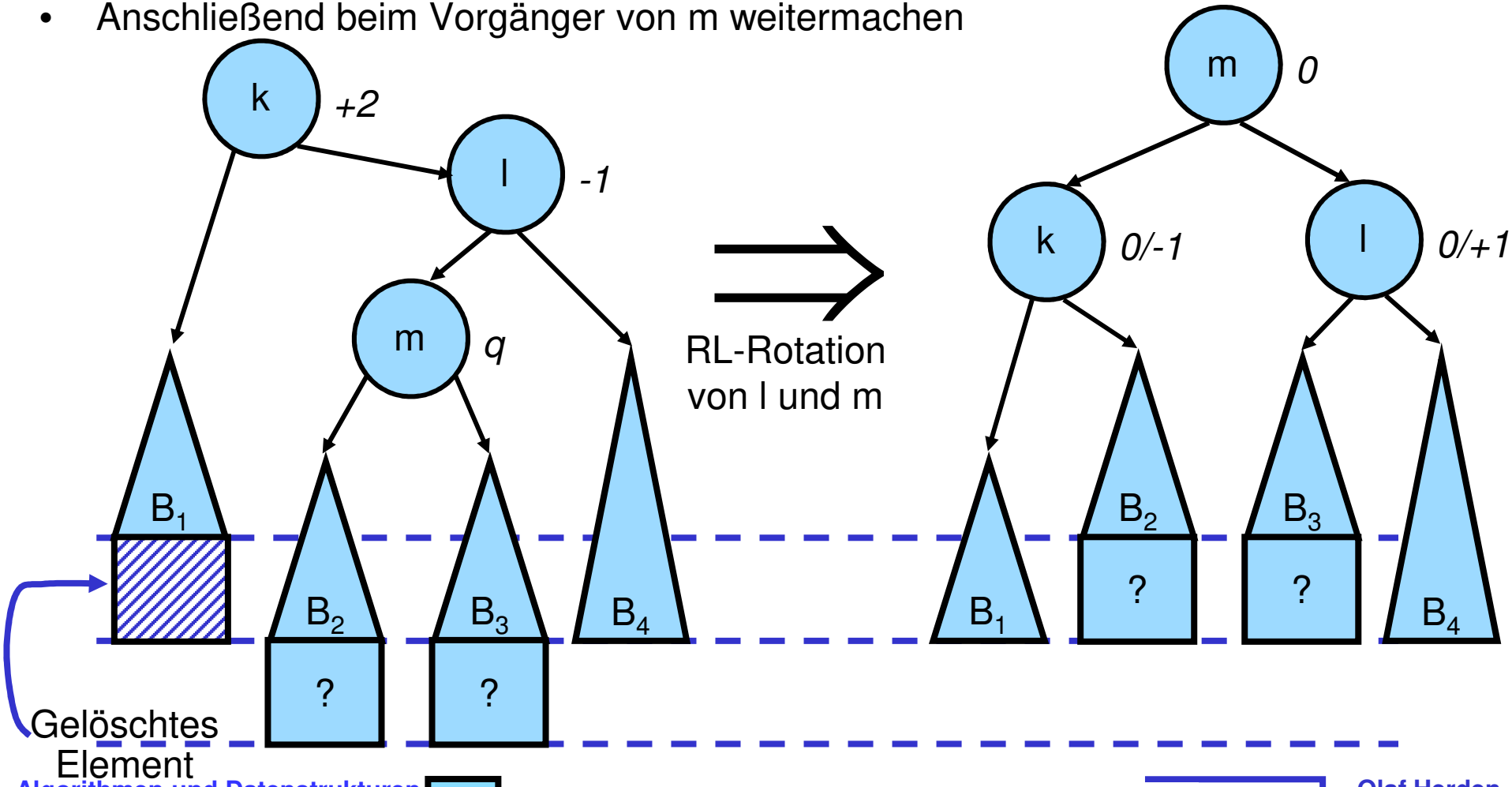
Löschen in AVL-Baum: Fall 3b

- Balancefaktor von l ist 0 $\Rightarrow$ Links-Rotation von k
- Gesamthöhe des Unterbaums unter k bzw. l bleibt unverändert
- Daher kann das Ausgleichen beendet werden



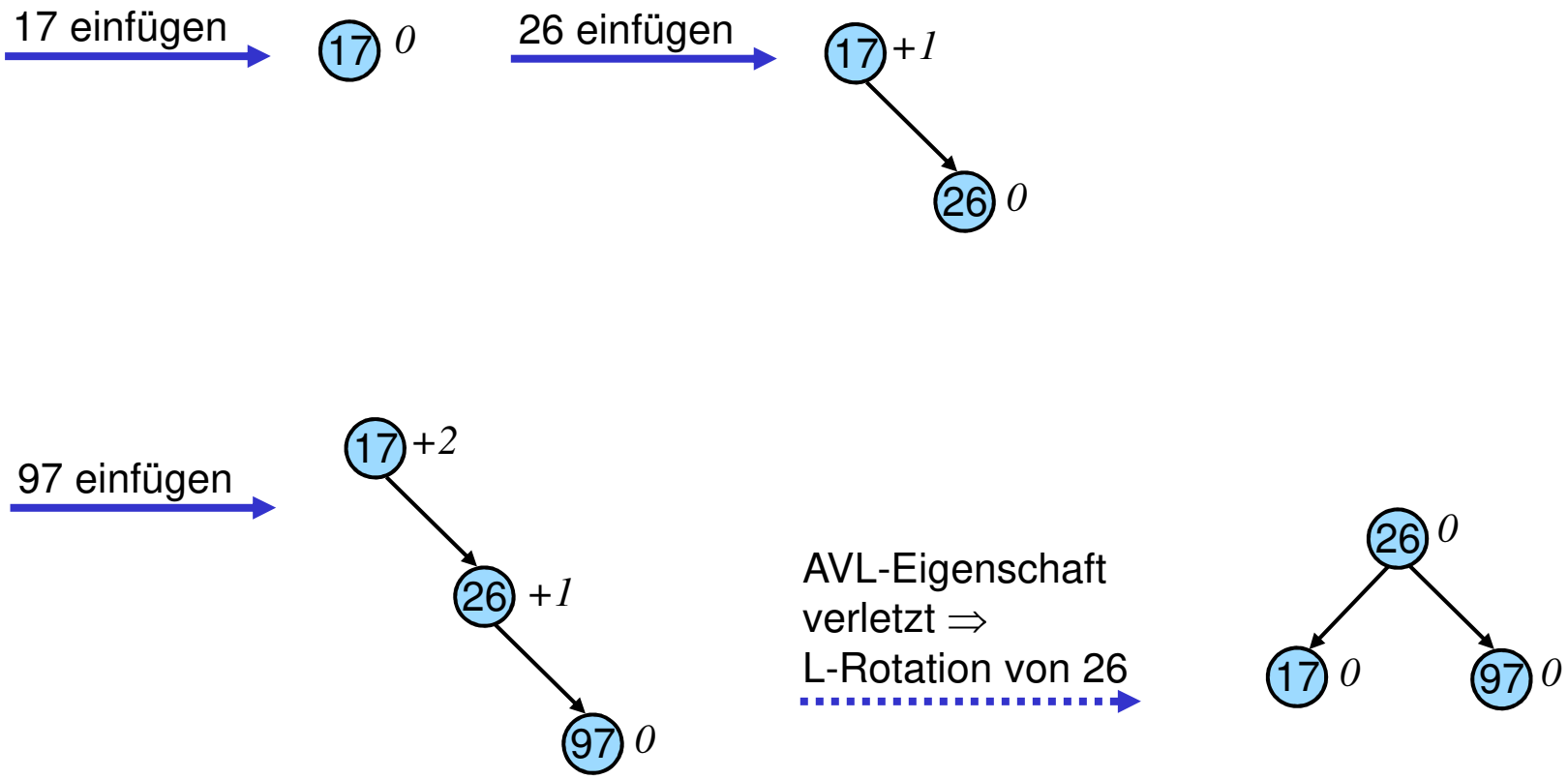
Löschen in AVL-Baum: Fall 3c

- Balancefaktor von l ist –1, m sei linker Nachfolger von l mit Balancefaktor q ∈ {-1, 0, +1}, d.h. in B_2 (q = -1) oder B_3 (q = 1) fehlt eine Stufe oder nicht (q = 0)
- RL-Rotation, zunächst R-Rotation von l, dann L-Rotation von m
- Anschließend beim Vorgänger von m weitermachen



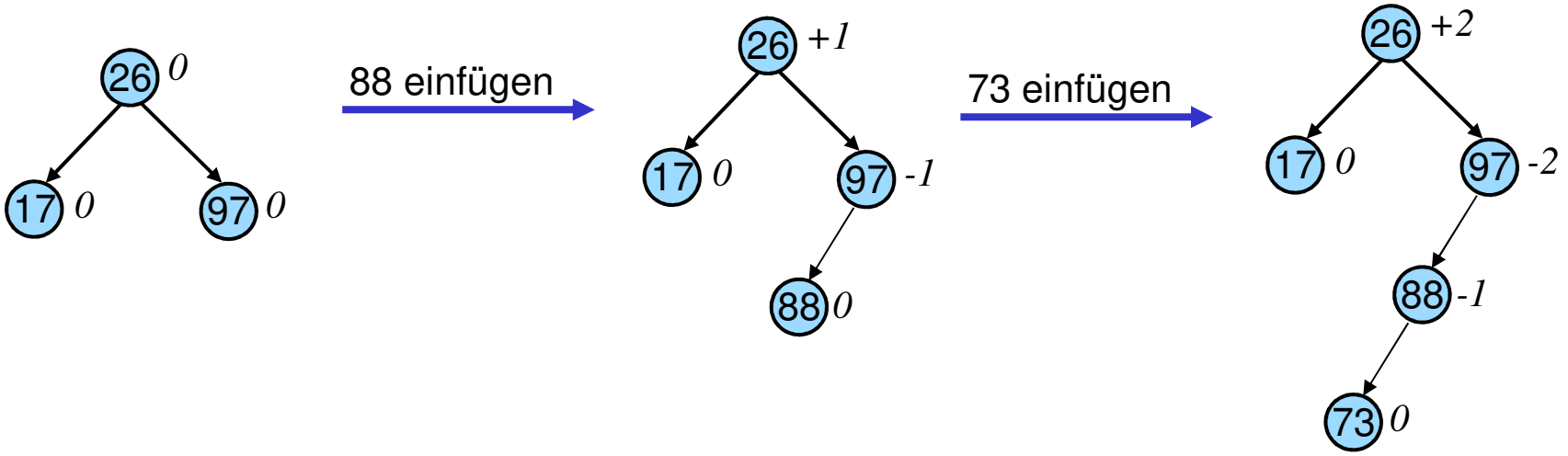
Beispiel (I)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

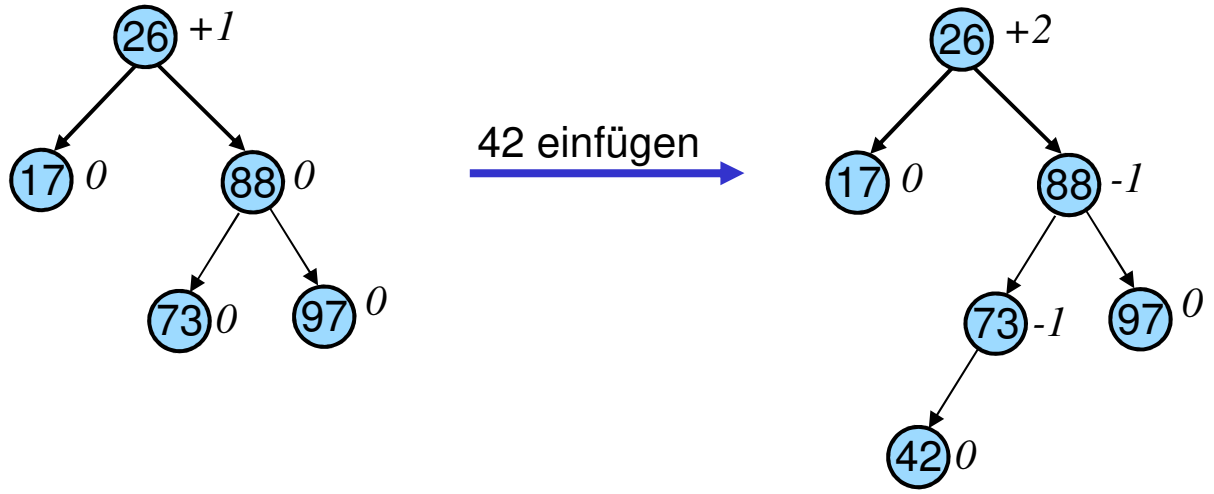


Beispiel (II)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

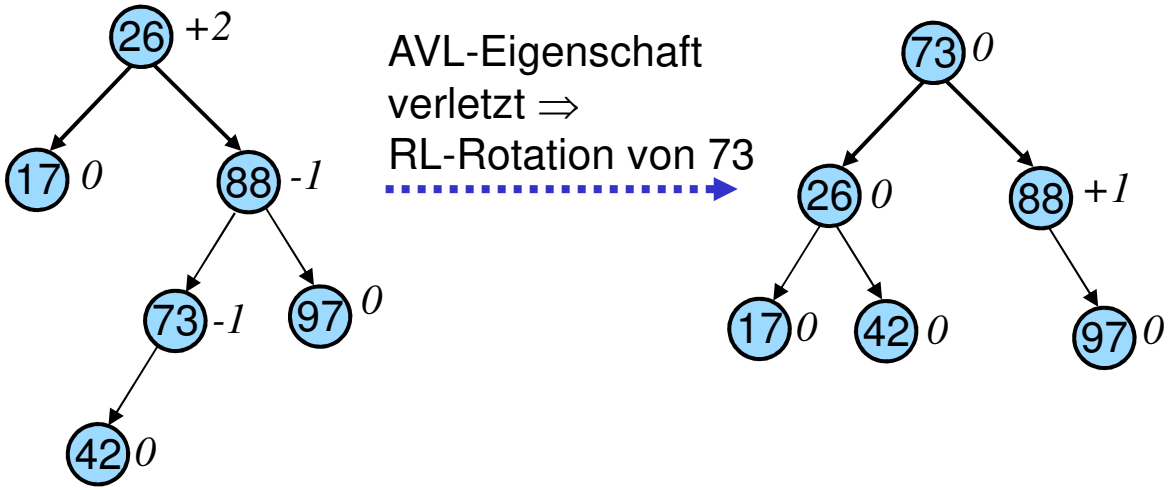


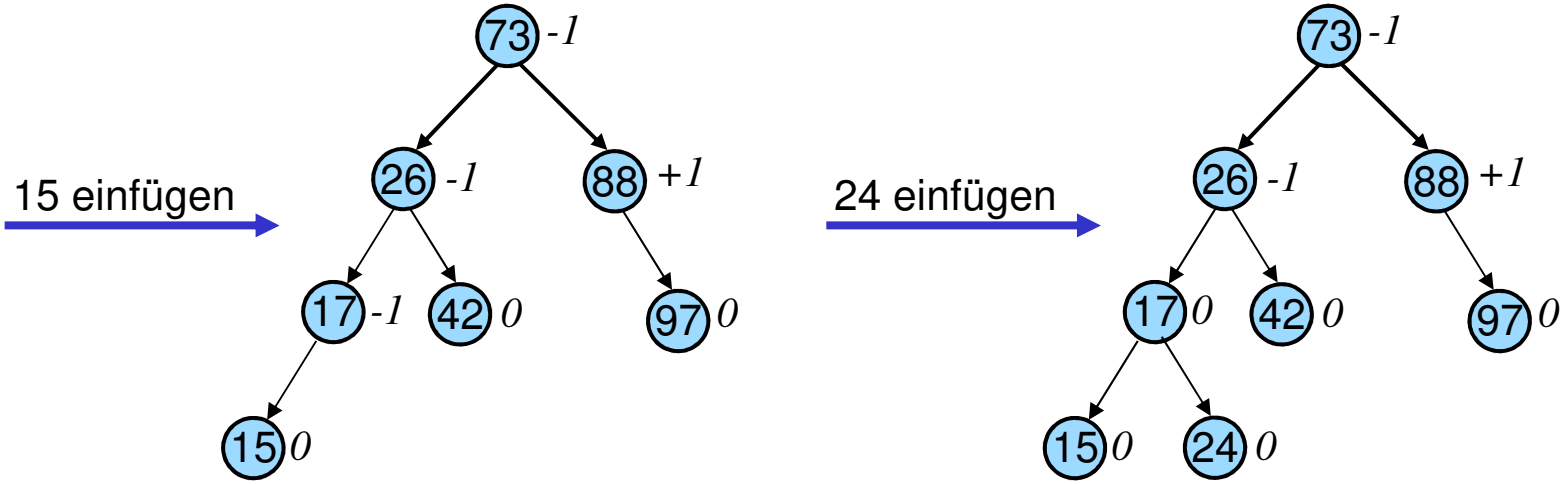
AVL-Eigenschaft verletzt ⇒ R-Rotation von 88



Beispiel (III)

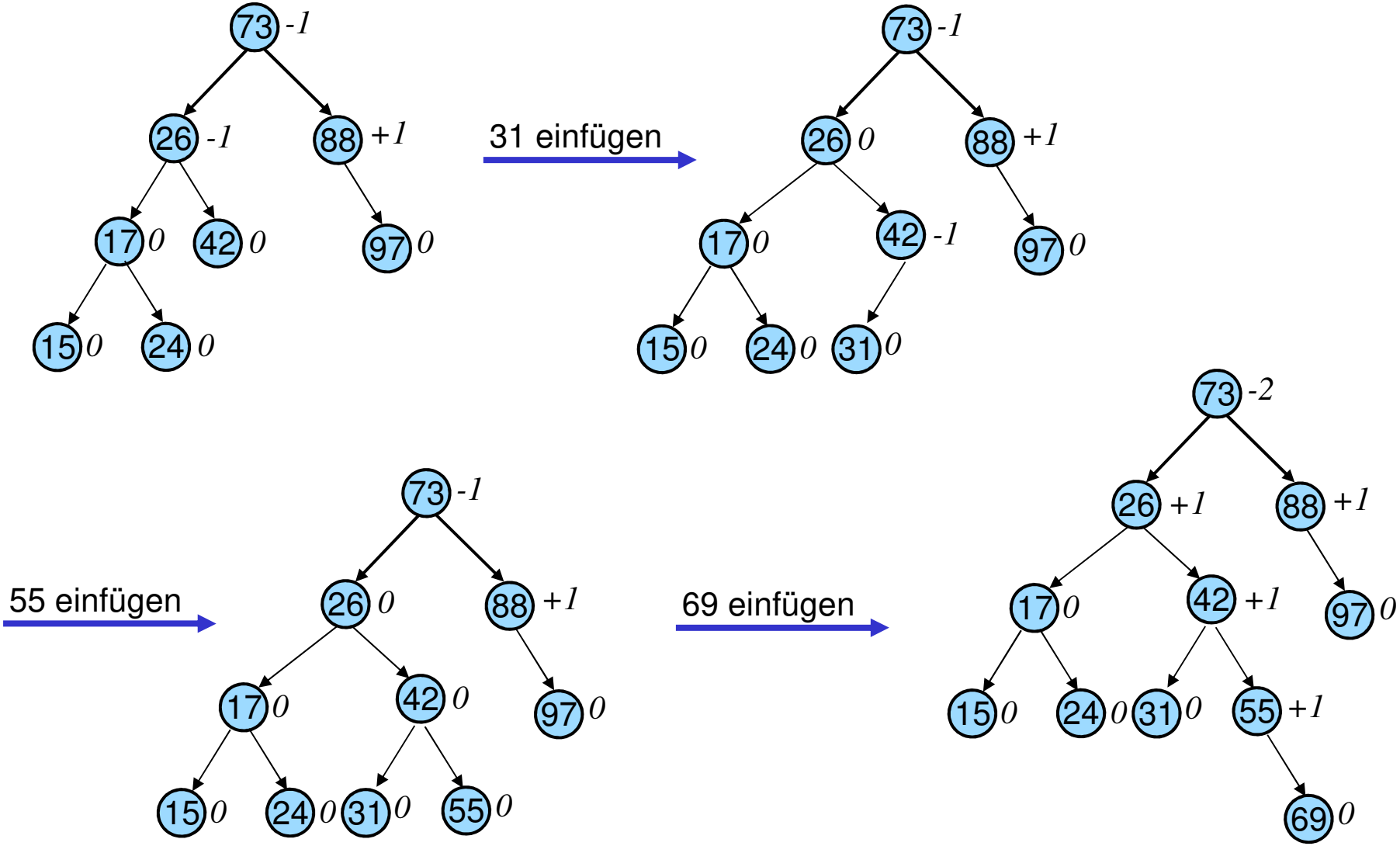
- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69





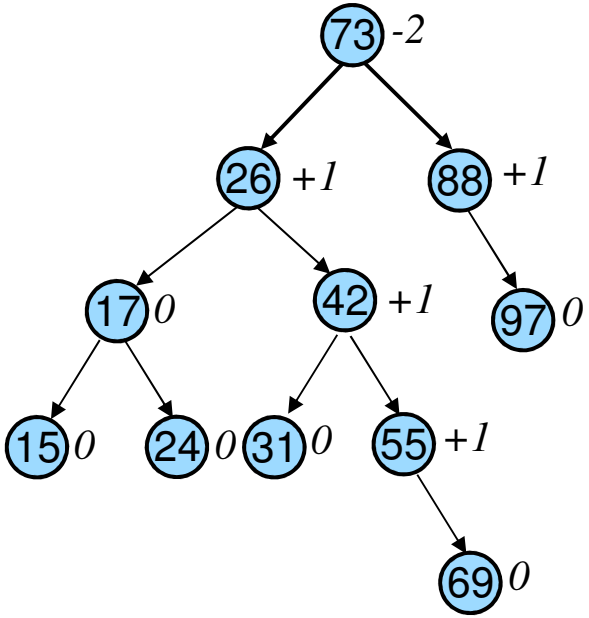
Beispiel (IV)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

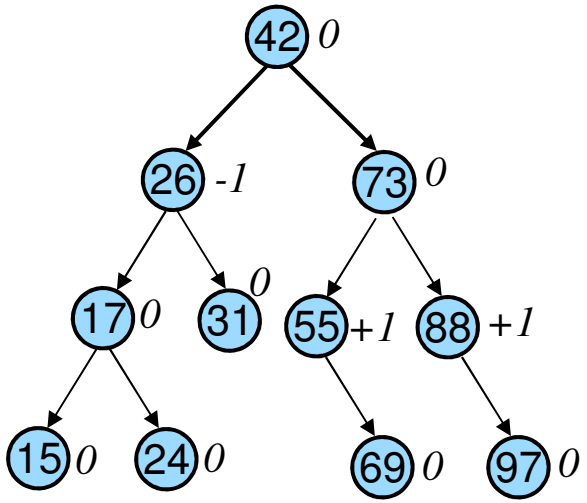


Beispiel (V)

- Baue AVL-Baum auf: 17, 26, 97, 88, 73, 42, 15, 24, 31, 55, 69

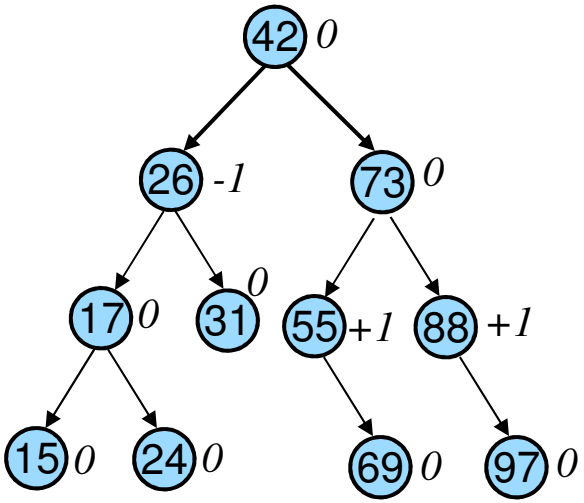


AVL-Eigenschaft verletzt ⇒
LR-Rotation von 42

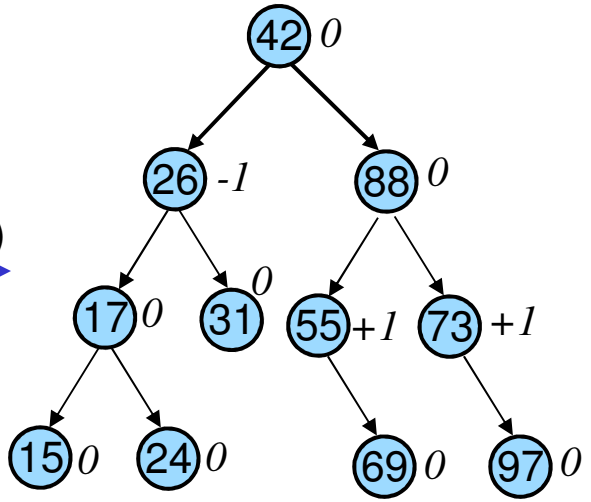


Beispiel (VI)

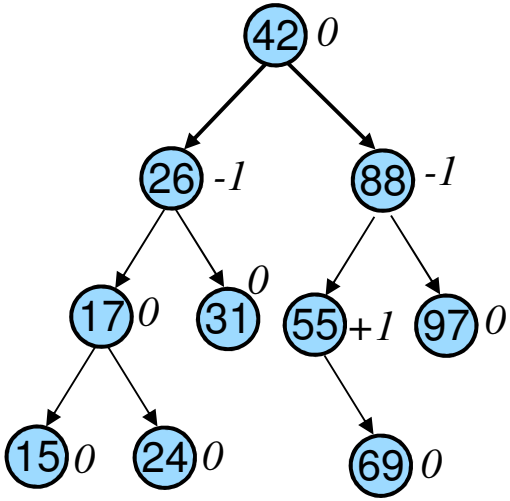
- Lösche Knoten 73



Vertausche zu lö-
schenden Knoten (73)
mit Inorder-Nachfolger (88)



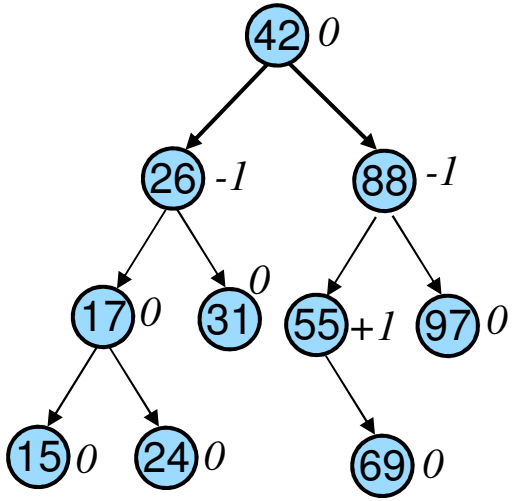
Knoten (73) kann an neuer
Position durch Umhängen
gelöscht werden



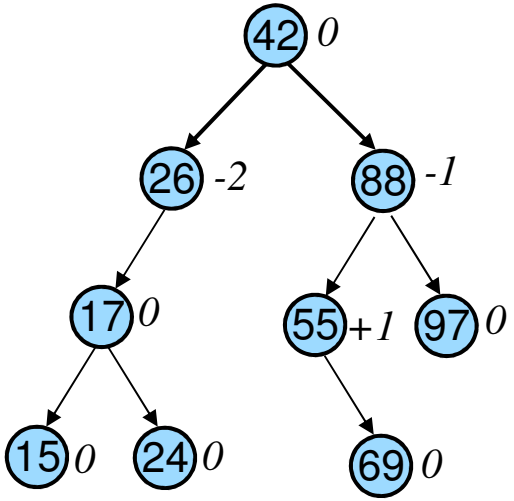
AVL-Eigenschaft
nicht verletzt
⇒ Operation beendet

Beispiel (VII)

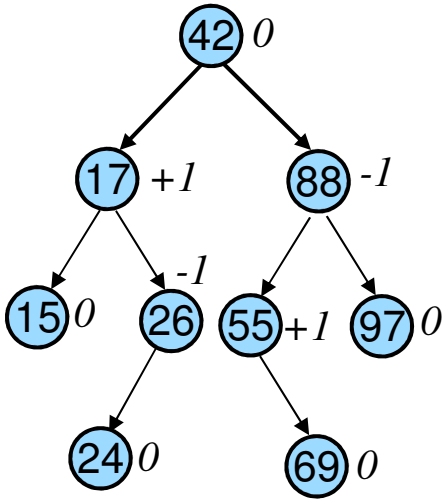
- Lösche Knoten 31



Zu löschender Knoten (31) kann als Blatt direkt gelöscht werden



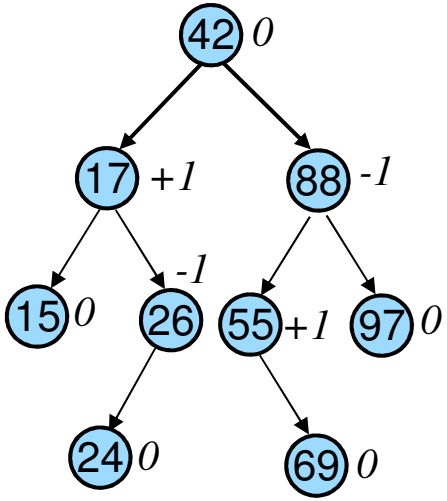
Knoten 26 verletzt AVL-Eigenschaft (symm. zu Fall 3b)
⇒ R-Rotation



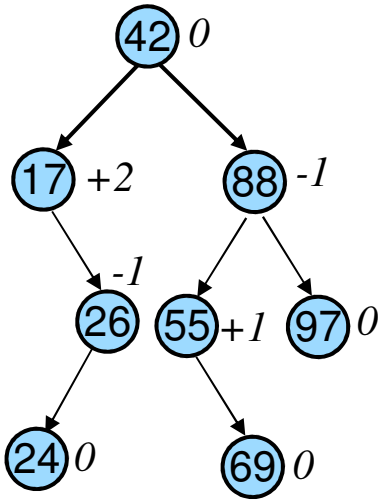
AVL-Eigenschaft wieder hergestellt
⇒ Operation beendet

Beispiel (VII)

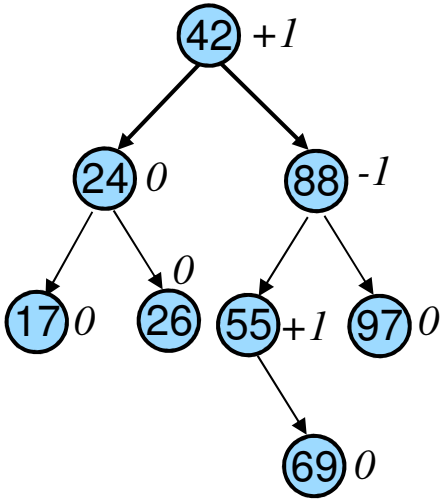
- Lösche Knoten 15



Zu löschender Knoten (15) kann als Blatt direkt gelöscht werden



Knoten 17 verletzt AVL-Eigenschaft (Fall 3c)
⇒ RL-Rotation



AVL-Eigenschaft wieder hergestellt
⇒ Operation beendet

Bewertung AVL-Baum

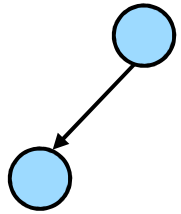
- Einfügen und Löschen kosten im worst case bzw. average case soviel wie die maximale bzw. mittlere Höhe eines AVL-Baums mit n Knoten beträgt
- Folgende Resultate wurden empirisch ermittelt:
  - Mittlere Höhe eines AVL-Baums mit n Knoten ist ca. $\log_2 n + 0.25$ (also fast genauso gut wie vollständig ausgeglichene Bäume)
  - Wahrscheinlichkeit einer Rotation beim Einfügen: ca. 50%
  - Wahrscheinlichkeit einer Rotation beim Löschen: ca. 20%
- Weiterhin kann man zeigen:
  - Für die Höhe h_b eines AVL-Baums mit n Knoten gilt:

$$\lfloor \log_2 n \rfloor + 1 < h_b < 1.44 \cdot \log_2 (n+1)$$

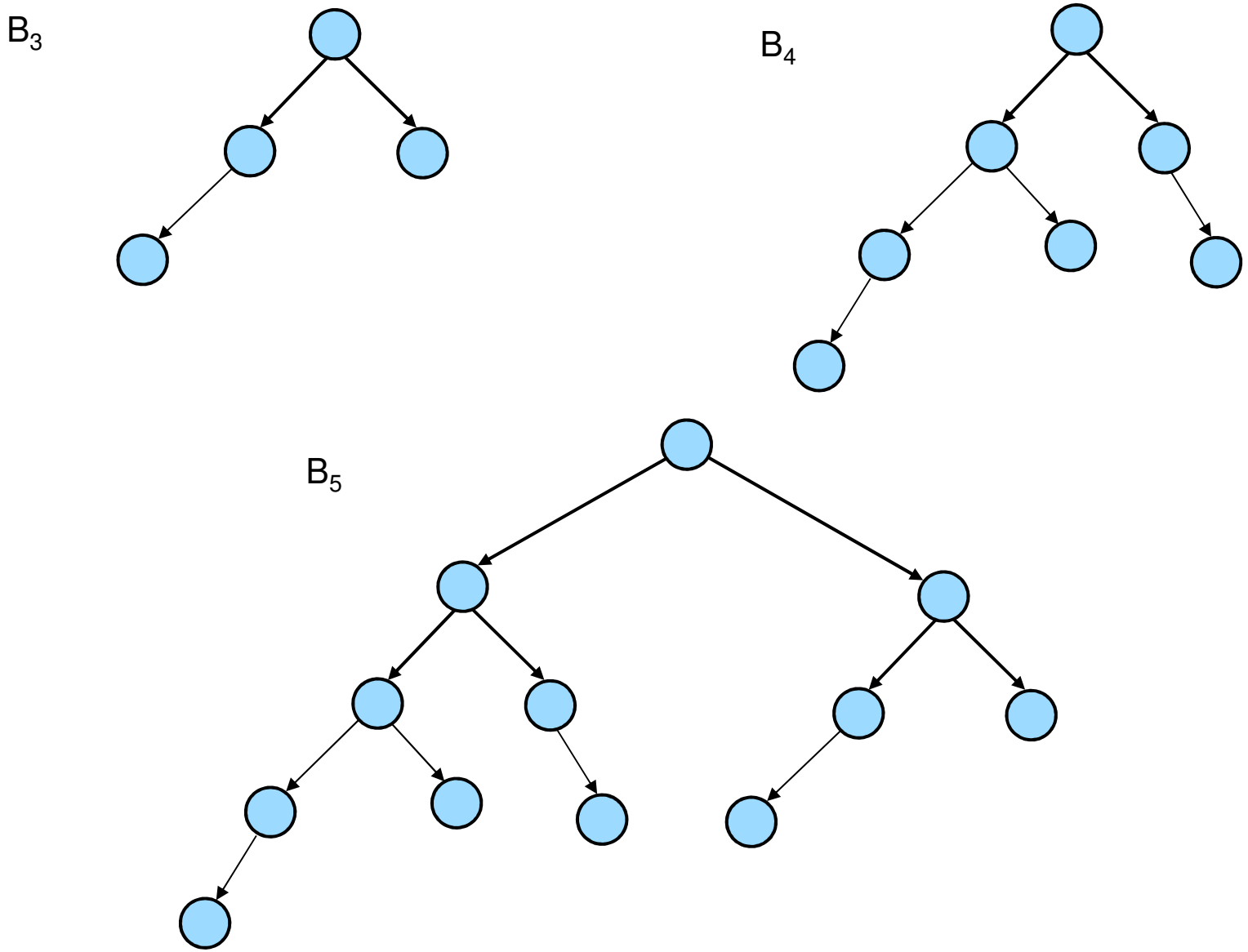
  - Maximale Suchzeit eines AVL-Baum damit höchstens ca. 44% höher als bei einem vollständig ausgeglichenen Baum
  - Mittlere Suchzeit aber erheblich günstiger, nie schlechter als O(log n)

Fibonacci-Bäume (I)

- Obere Schranke lässt sich durch eine als Fibonacci-Bäume bezeichnete Unterklasse herleiten
- Aufbau von Fibonacci-Bäumen:
  - Leerer Baum ist Fibonacci-Baum der Höhe 0
  - Einzelner Knoten ist Fibonacci-Baum der Höhe 1
  - Sind B_{h-1} und B_{h-2} Fibonacci-Bäume der Höhe h-1 bzw. h-2, dann ist der Baum mit einer neuen Wurzel und B_{h-1} und B_{h-2} als linkem und rechtem Unterbaum Fibonacci-Baum der Höhe h
  - Kein anderer Baum ist Fibonacci-Baum
- Fibonacci-Bäume:

B_0 B_1  B_2 

Fibonacci-Bäume (II)



Übersicht

- Suchen in Feldern und Listen
- Binäre Suchbäume
- AVL-Bäume
- Gewichtsbalancierte Bäume

Gewichtsbalancierte Bäume (I)

- Gewichtsbalancierte Bäume (Synonym: BB-Bäume (Bounded Balance)): Zulässige Abweichung der Struktur vom ausgeglichenen Binärbaum wird als Differenz zwischen Anzahl der Knoten im rechten und linken Unterbaum festgelegt
- Definition:

  Sei B ein binärer Suchbaum mit n Knoten, sei B_l linker Unterbaum mit n_l Knoten.

  Dann heißt $p(B) = \frac{(n_l + 1)}{(n + 1)}$ Wurzelbalance von B.

- Definition:

  Binärbaum B heißt gewichtsbalanciert (Schreibweise: BB(α)), wenn für jeden Unterbaum B‘ von B $\alpha \leq p(B') \leq 1 - \alpha$ gilt.

Gewichtsbalancierte Bäume (II)

- Wahl des Parameters α bestimmt wie streng die Gewichtsbalancierung ausgelegt wird:
  - $\alpha = ½$: Es werden nur vollständig ausgeglichene Binärbäume akzeptiert
  - $\alpha < ½$: Kriterium wird zunehmend gelockert
- Je kleiner α gewählt wird, desto weniger Reorganisationsoperationen sind notwendig, umso degenerierter kann der Baum aber auch werden
- Im Extremfall $\alpha = 0$ kann der Baum zur Liste werden
- Praktisch realistischer Wert: $\alpha = 0{,}3$
- Für Reorganisation zum Einhalten der Balance-Eigenschaften kommen die selben Rotationstypen wie beim AVL-Baum zum Einsatz
- Kosten für Suche und Aktualisierung bei $\alpha = 0{,}3$: $O(\log_2 n)$
- Damit in Effizienz mit AVL-Bäumen vergleichbar

Vergleich

	Feld	Liste	Binärer Suchbaum	AVL-Baum
Knoten Suchen	$O(\log_2 n)$	$O(n)$	$O(n)$	$O(\log_2 n)$
Alle Knoten sortiert ausgeben	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Knoten einfügen	$O(\log_2 n) + O(n)$	$O(n)$	$O(n)$	$O(\log_2 n)$
Knoten löschen	$O(\log_2 n) + O(n)$	$O(n)$	$O(n)$	$O(\log_2 n)$

Zusammenfassung

- Suchen in Feldern und Listen:
  - Lineare Suche
  - Binäre Suche
  - Interpolationssuche
- Binäre Suchbäume:
  - Definition
  - Operationen: Einfügen, Suchen, Löschen, Ausgeben
  - Problem: Degenerierung zur Liste
- AVL-Bäume:
  - Höhenbalancierter Baum, Balancefaktoren
  - Einfügen, Löschen
  - Rotationen
- Gewichtsbalancierte Bäume:
  - Anzahl der Knoten in linken und rechtem Unterbaum darf sich nicht „allzu stark“ unterscheiden

Übungen (I)

- **Aufgabe 1**

  Gegeben sei die folgenden Zahlenfolge:

  23, 66, 42, 11, 5, 69, 77, 55, 16.

  a) Bauen Sie schrittweise einen binären Suchbaum auf!
  b) Löschen Sie aus dem binären Suchbaum nacheinander die Knoten 77, 42 und 23.
  c) Bauen Sie schrittweise einen AVL-Baum auf!
  d) Löschen Sie aus dem AVL-Baum nacheinander die Knoten 5, 42 und 77.

- **Aufgabe 2**

  Zeichnen Sie die zu den in der Vorlesung vorgestellten Rotationen beim Einfügen und Löschen in einem AVL-Baum die symmetrischen Fälle.

- **Aufgabe 3**

  a) Wie viele Knoten hat ein Fibonacci-Baum der Höhe n?
  b) Zeichnen Sie alle verschiedenen Fibonacci-Bäume der Höhe 4!
  c) Wie viele verschiedene Fibonacci-Bäume der Höhe n gibt es?

Übungen (II)

- **Aufgabe 4**

	wahr	falsch
Einfügen im binären Suchbaum geschieht immer im linken Teilbaum.		
Jeder Heap ist ein binärer Suchbaum.		
Suchen in einem Feld hat konstanten Aufwand.		
Bei gewichtsbalancierten Bäumen ist der Faktor α möglichst klein zu wählen.		
AVL-Bäume sind höhenbalancierte Suchbäume.		

Übungen (III)

- **Aufgabe 5**

  In einem Binärbaum seien die Zahlen zwischen 1 und 1000 gespeichert.
  Nun soll die Zahl 363 gesucht werden.
  Welche der folgenden Sequenzen kann nicht die überprüfte Knotenfolge sein?

  a) 2, 252, 401, 398, 330, 344, 397, 363

  b) 924, 220, 911, 244, 898, 258, 362, 363

  c) 925, 202, 911, 240, 912, 245, 363

  d) 2, 399, 387, 219, 266, 382, 381, 278, 363

  e) 935, 278, 347, 621, 299, 392, 358, 363

Übungen (IV)

- **Aufgabe 6**

  Karl-Heinz glaubt, er habe eine bemerkenswerte Eigenschaft binärer Suchbäume entdeckt. Nehmen Sie an, die Suche in einem binären Suchbaum nach dem Schlüssel k würde in einem Blatt enden. Betrachten Sie drei Mengen: A, die Menge der Schlüssel, die links vom Suchpfad liegen, B, die Menge der Schlüssel auf dem Suchpfad, und C, die Menge der Schlüssel rechts vom Suchpfad. Karl-Heinz behauptet, dass jedes Tripel von Schlüsseln $a \in A$, $b \in B$ und $c \in C$ die Ungleichung $a <= b <= c$ erfüllen muss.

  Geben Sie ein möglichst kleines Gegenbeispiel zu der Behauptung an!