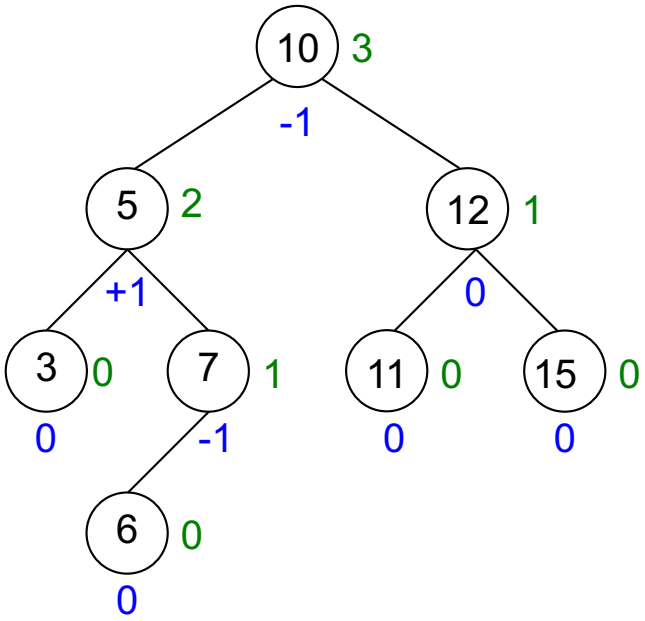
4. Balancierte Suchbäume

- AVL-Bäume
- B-Bäume
- 2-3-4-Bäume und Rot-Schwarzbäume

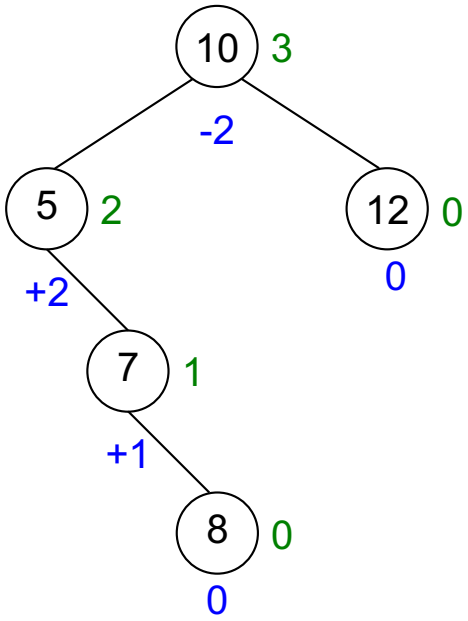
Definition von AVL-Bäumen

- AVL-Bäume sind (höhen)balancierte binäre Suchbäume.
- Für jeden Knoten unterscheiden sich die Höhen der beiden Teilbäume um höchstens 1.
- Die Abkürzung AVL geht zurück auf Adelson-Velskij und Landis, 1962.

AVL-Baum:



Nicht-AVL-Baum



Höhe des Teilbaums

Höhenunterschied
= Höhe rechter Teilbaum
− Höhe linker Teilbaum

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

Höhe von AVL-Bäumen

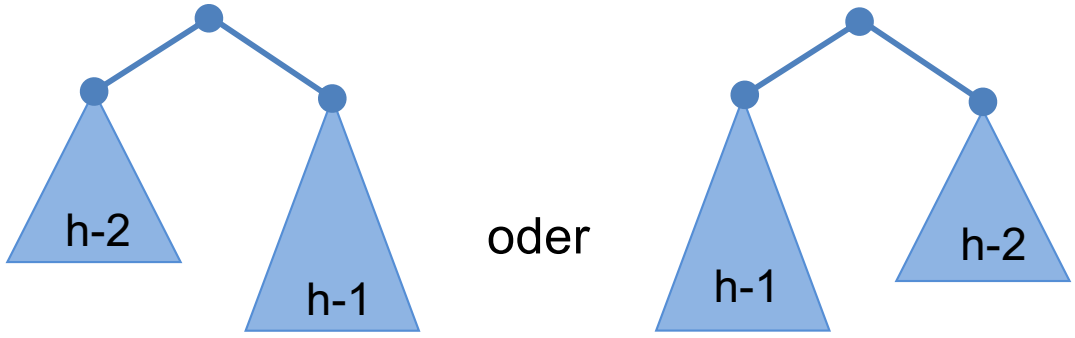
- Entscheidend für die Komplexität der Algorithmen ist die Höhe der AVL-Bäume.
- Also: Wie groß ist die maximale Höhe eines AVL-Baums mit n Knoten?
- Wir betrachten dazu minimale AVL-Bäume.
  Das sind AVL-Bäume die bei gegebener Höhe h eine minimale Knotenanzahl haben.
- Minimale AVL-Bäume mit n Knoten sind besonders dünn besetzt und erreichen daher eine maximale Höhe für n Knoten.

Minimale AVL-Bäume

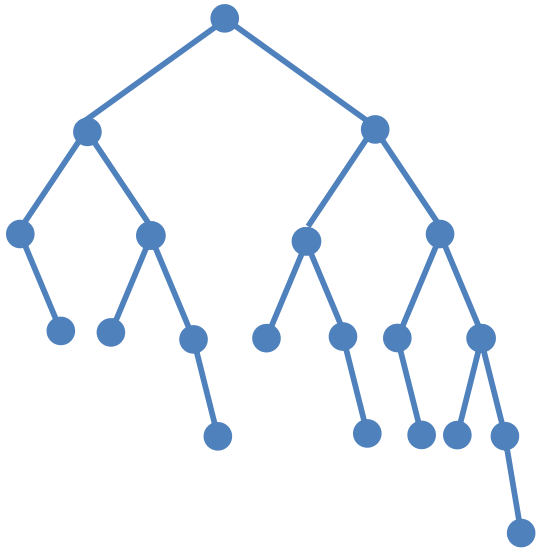
- Minimale AVL-Bäume lassen sich durch folgende Rekursionsvorschrift bauen:
  - Minimaler AVL-Baum der Höhe h = 0:

  - Minimaler AVL-Baum der Höhe h = 1:

  - Minimaler AVL-Baum der Höhe h ergibt sich aus minimalen AVL-Bäumen der Höhe h-1 und h-2:



Minimaler AVL-Baum der Höhe h = 5



- AVL-Baum ist für die Höhe h = 5 minimal:
  - Das rechte Blatt darf nicht entfernt werden, weil die Höhe dann kleiner werden würde.
  - Die anderen Blätter dürfen nicht gelöscht werden, weil dann die AVL-Balancierung verletzt werden würde.

Anzahl Knoten in einem minimalen AVL-Baum (1)

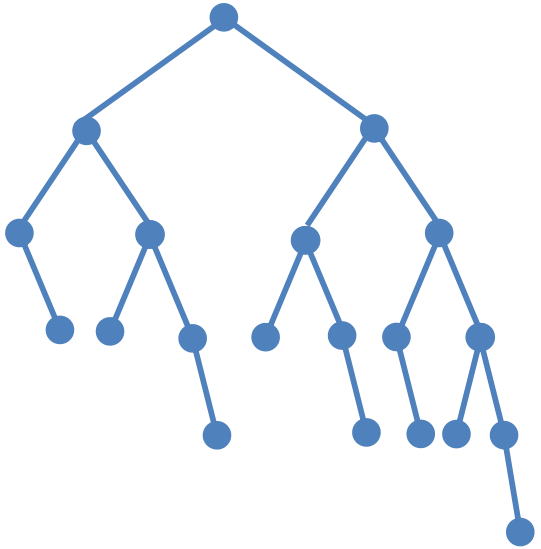
- Für N(h) als Anzahl der Knoten in einem minimalen AVL-Baum der Höhe h gilt:

$$N(0) = 1$$
$$N(1) = 2$$
$$N(h) = 1 + N(h-2) + N(h-1) \quad \text{falls } h \geq 2$$

Minimaler AVL-Baum
der Höhe h=0 mit N(0) = 1

Minimaler AVL-Baum
der Höhe h=2 mit N(2) = 4

Minimaler AVL-Baum
der Höhe h = 1 mit N(1) = 2



Minimaler AVL-Baum
der Höhe h=5 mit N(5) = 20

Anzahl Knoten in einem minimalen AVL-Baum (2)

- N(h) erinnert stark an die Fibonacci-Funktion fib(h) = fib(h-1) + fib(h-2).
  Durch vollständige Induktion lässt sich zeigen:

$$N(h) = fib(h+3) - 1$$

- Mit expliziter Form der Fibonacci-Funktion:

$$\begin{aligned} N(h) &= fib(h+3) - 1 \\ &= \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right) - 1 \\ &\geq \left(1 + \frac{2}{\sqrt{5}} \right) \cdot \left(\frac{1+\sqrt{5}}{2} \right)^{h} - 2 \end{aligned}$$

- Nach h aufgelöst:

$$h \leq 1.44 \cdot \log_2(N(h)+2) - 1.33$$

Höhe eines AVL-Baums

- Für die Höhe h eines AVL-Baums mit n Knoten gilt:

$$\lfloor \log_2(n) \rfloor \leq h \leq 1.44 * \log_2(n+2) - 1.33$$

Höhe eines vollständigen Binärbaums mit n Knoten

Höhe eines minimalen AVL-Baums mit n Knoten

- Alle Dictionary-Operationen lassen sich daher in O(log n) realisieren.

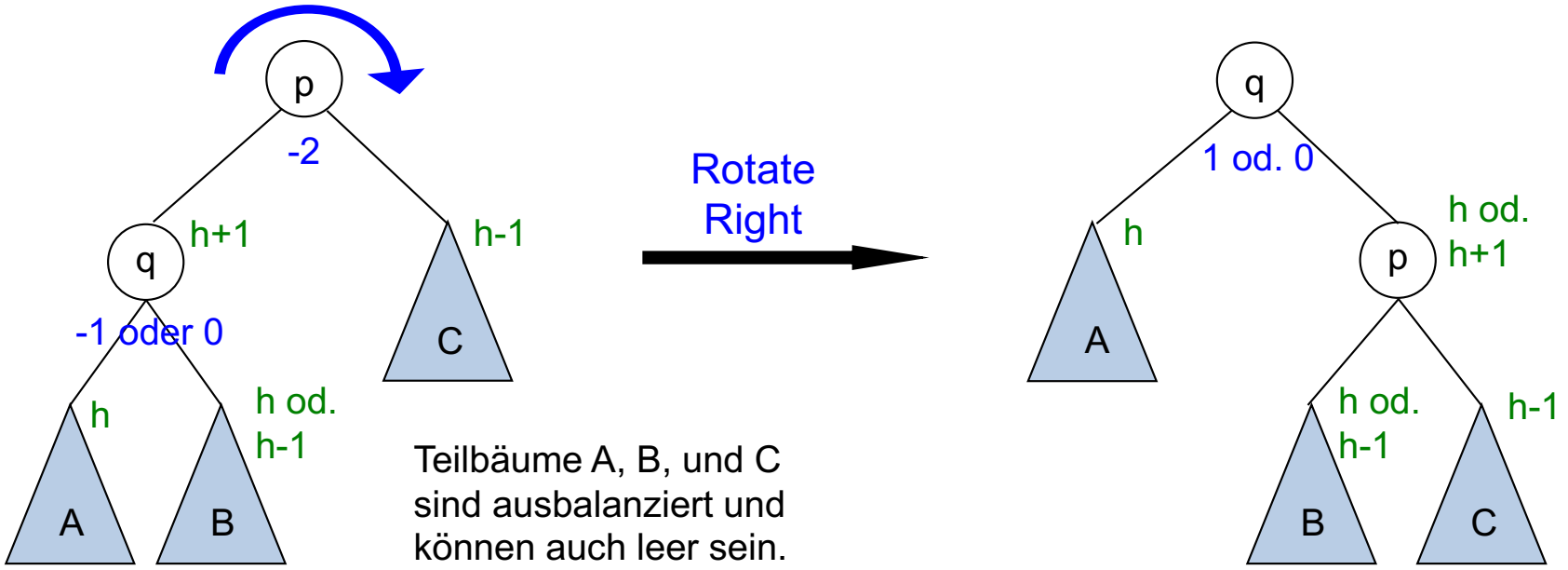
Dictionary-Operationen

- **search**
  - wie bei binären Suchbäumen
- **insert und remove**
  - füge bzw. lösche wie bei binären Suchbäumen
  - da der Baum zuvor ausbalanciert war (Höhenunterschied von maximal 1 an allen Knoten), kann nach Durchführung der Operation höchstens ein Höhenunterschied von 2 (an evtl. mehreren Knoten) entstehen.
  - Gehe dann von der Einfügestelle bzw. Löschstelle bis zur Wurzel und balanciere - falls notwendig - mit einer der 4 folgenden Rotationsoperationen lokal aus.
  - Balancierung lässt sich besonders geschickt beim rekursiven Aufstieg durchführen.

Rechts-Rotation

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

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

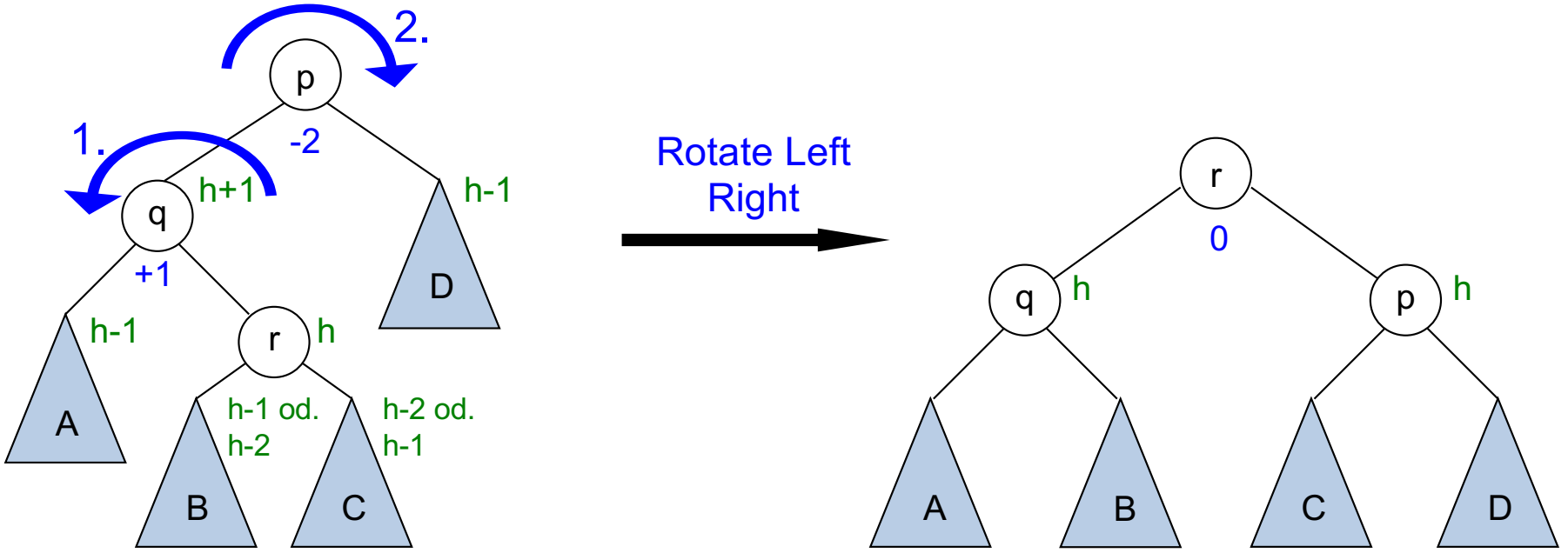


Teilbäume A, B, und C sind ausbalanziert und können auch leer sein.

Links-Rechts-Rotation

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

Unterfall A2: linker Teilbaum hat Höhenunterschied +1:

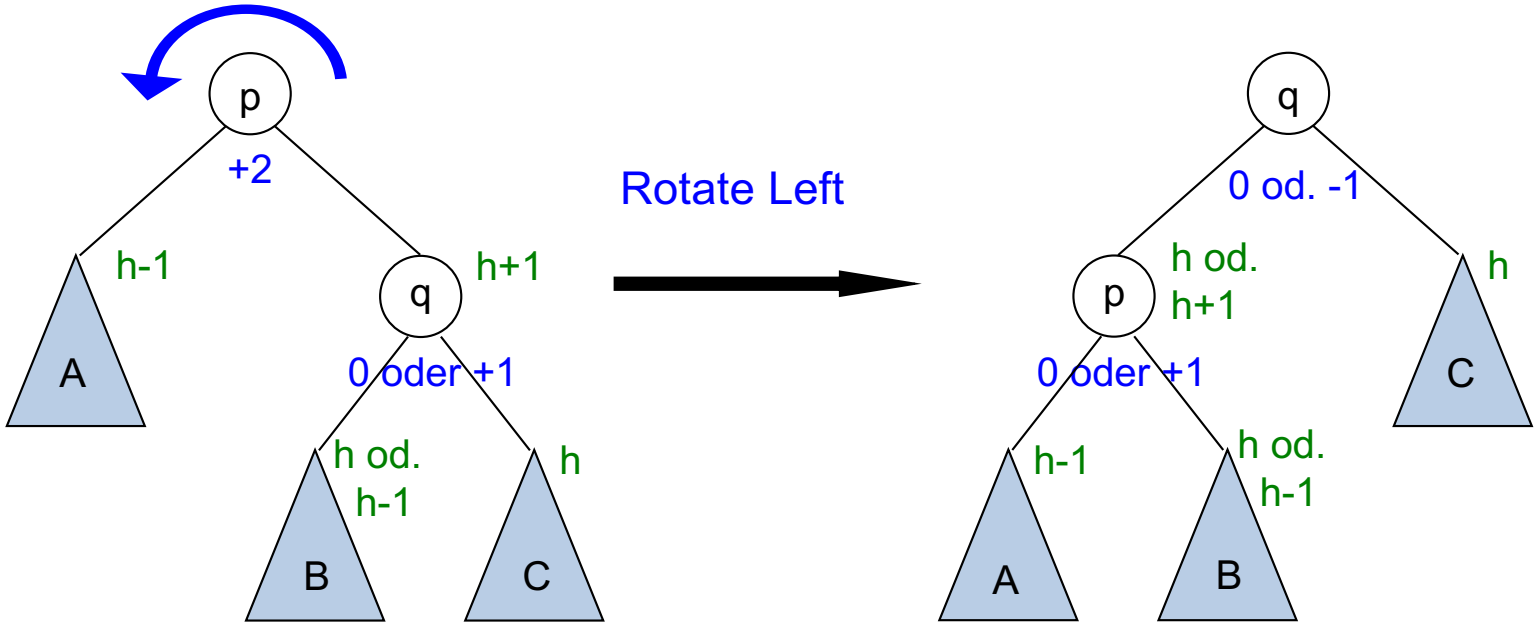


Teilbäume B und C haben aber nicht beide die Höhe h-2.

Links-Rotation

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

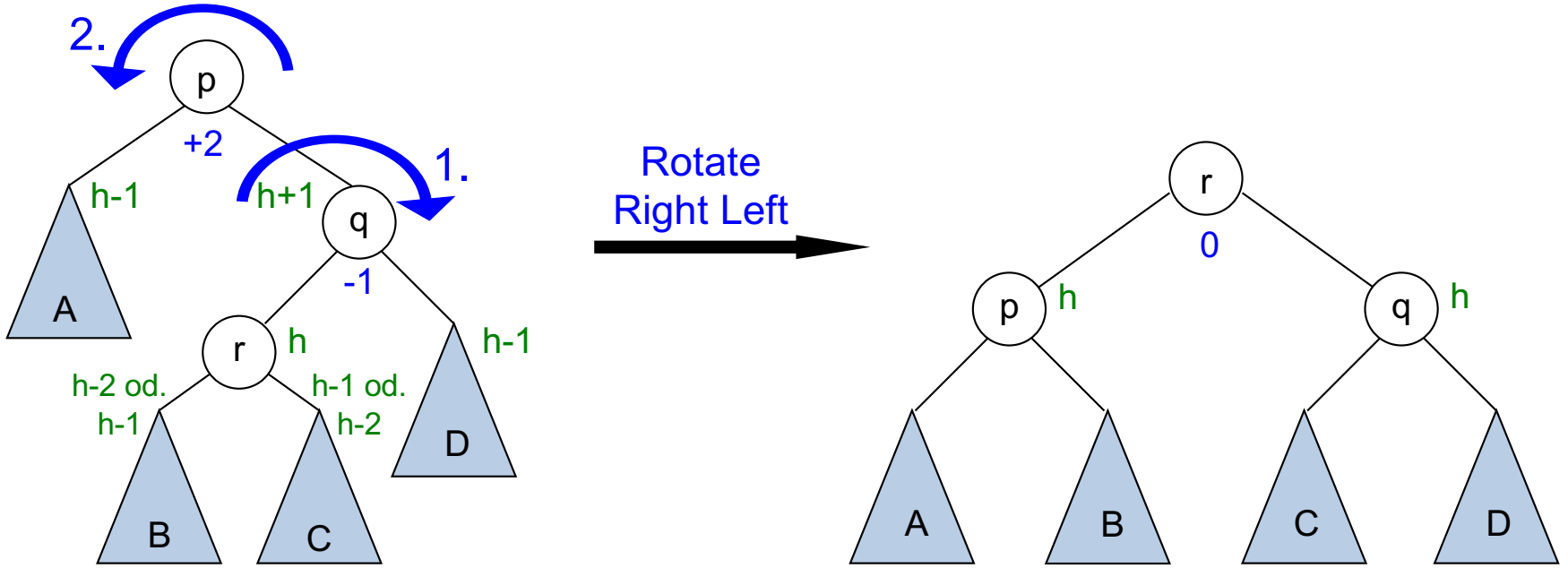
Unterfall B1: rechter Teilbaum hat Höhenunterschied 0 oder +1:



Rechts-Links-Rotation

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

Unterfall B2: rechter Teilbaum hat Höhenunterschied -1:

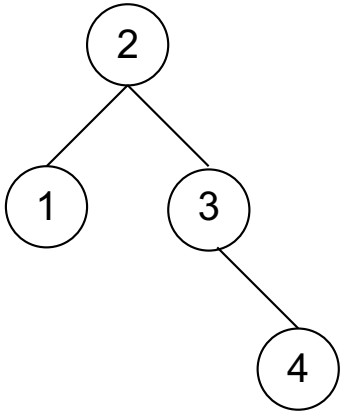


Teilbäume B und C haben aber nicht beide die Höhe h-2.

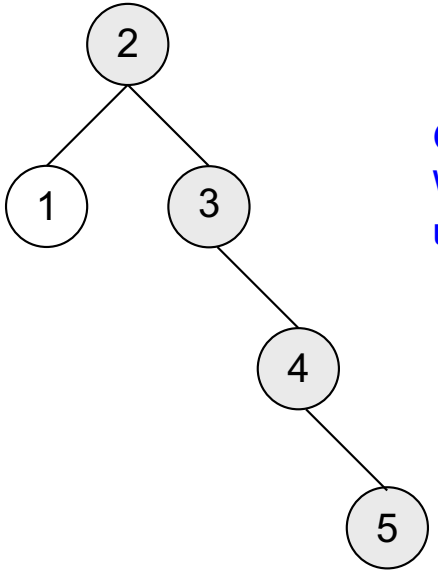
Bemerkungen

- Da der Baum vor dem Einfügen bzw. Löschen balanciert war, haben alle Teilbäume einen Höhenunterschied von -1, 0 oder +1. Damit können nach einem Einfüge- oder Löschschritt nur die Fälle A1, A2, B1 oder B2 auftreten.
- Es lässt sich zeigen, dass beim Einfügen insgesamt maximal eine der 4 Rotationsoperationen notwendig ist.
- Beim Löschen müssen i.a. mehrere Rotationsoperationen durchgeführt werden (siehe Beispiel später)

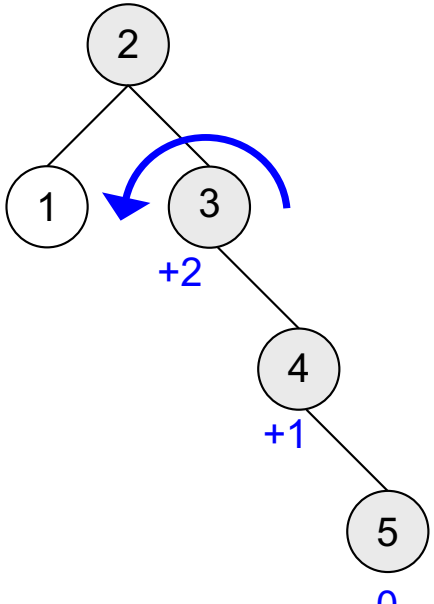
Beispiel zu Einfügen in AVL-Bäumen (1)



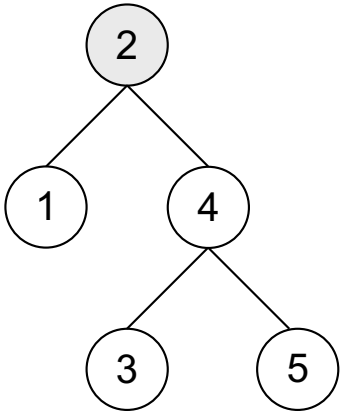
Füge 5 ein



Gehe vom Knoten 5 in Richtung Wurzel bis zum ersten unbalanzierten Teilbaum.

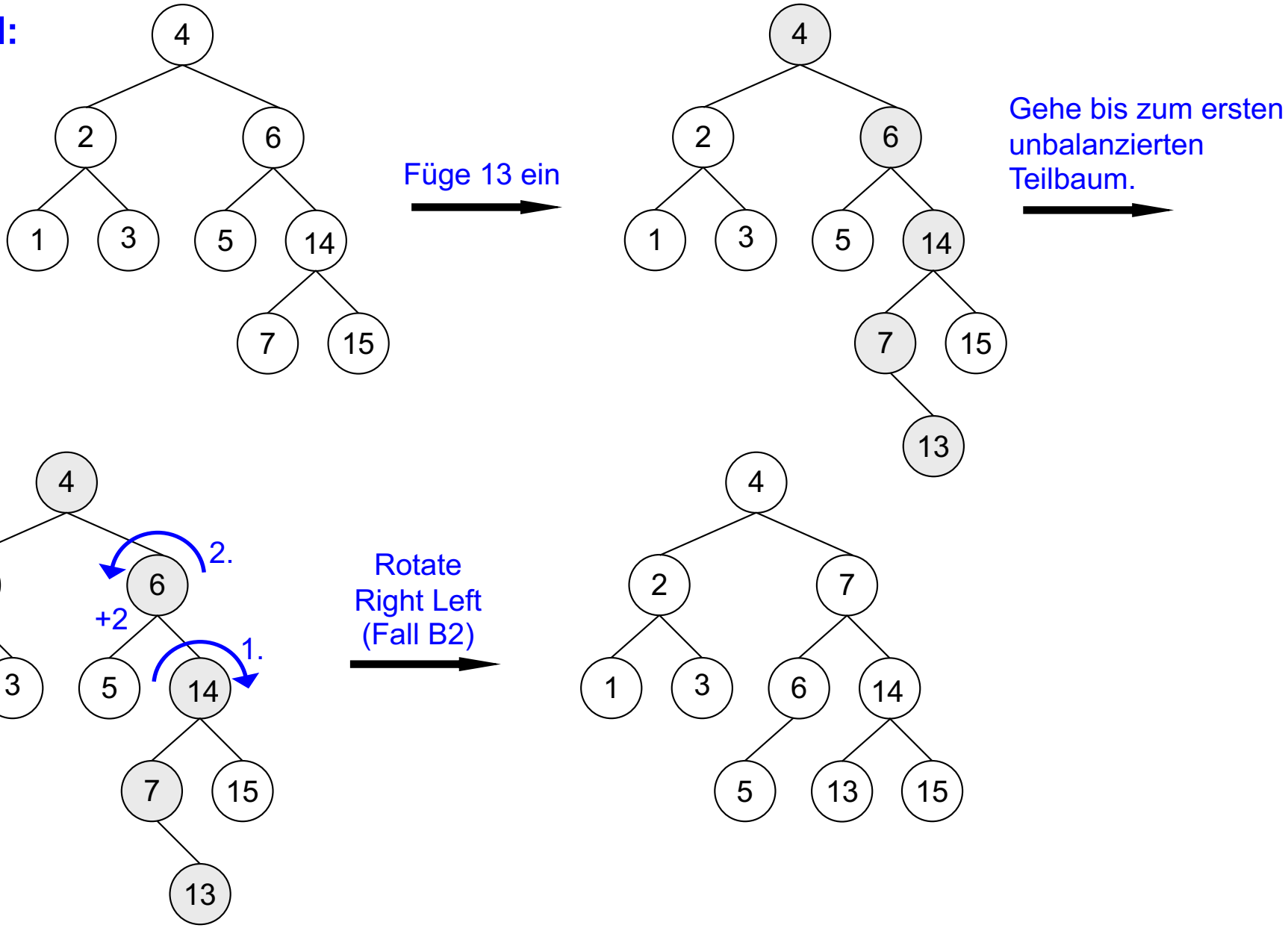


Rotate Left
(Fall B1)



Beispiel zu Einfügen in AVL-Bäumen (2)

Beispiel:



Algorithmen (1)

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

```
private static class Node<K, V> {
        int height;
        K key;
        V value;
        Node<K, V> left;
        Node<K, V> right;
        private Node(K k, V v) {
                height = 0;
                key = k;
                value = v;
                left = null;
                right = null;
        }
}
```

```
private int getHeight(Node<K,V> p) {
        if (p == null)
                return -1;
        else
                return p.height;
    }

private int getBalance(Node<K,V> p) {
        if (p == null)
                return 0;
        else
                return getHeight(p.right) - getHeight(p.left);
}
```

Algorithmen (2)

- Erweitere die rekursive insertR-Operation für binäre Suchbäume um eine Balanzieroperation, die beim rekursiven Aufstieg durchgeführt wird.

```
private Node<K,V> insertR(K key, V value, Node<K,V> p) {
        if (p == null) {
                p = new Node(key, value);
                oldValue = null;
        }
        else if (key.compareTo(p.key) < 0)
                p.left = insertR(key, value, p.left);
        else if (key.compareTo(p.key) > 0)
                p.right = insertR(key, value, p.right);
        else {  // Schlüssel bereits vorhanden:
                oldValue = p.value;
                p.value = value;
        }
        p = balance(p);
        return p;
}
```

- Die removeR- und getRemMinR-Operation werden analog erweitert.
- Die searchR-Operation bleibt unverändert.

Algorithmen (3)

```
private Node<K,V> balance(Node<K,V> p) {
    if (p == null)
        return null;
    p.height = Math.max(getHeight(p.left), getHeight(p.right)) + 1;
    if (getBalance(p) == -2) {
        if (getBalance(p.left) <= 0)
            p = rotateRight(p);
        else
            p = rotateLeftRight(p);
    }
    else if (getBalance(p) == +2) {
        if (getBalance(p.right) >= 0)
            p = rotateLeft(p);
        else
            p = rotateRightLeft(p);
    }
    return p;
  }
```

Höhe aktualisieren.

Fall A1

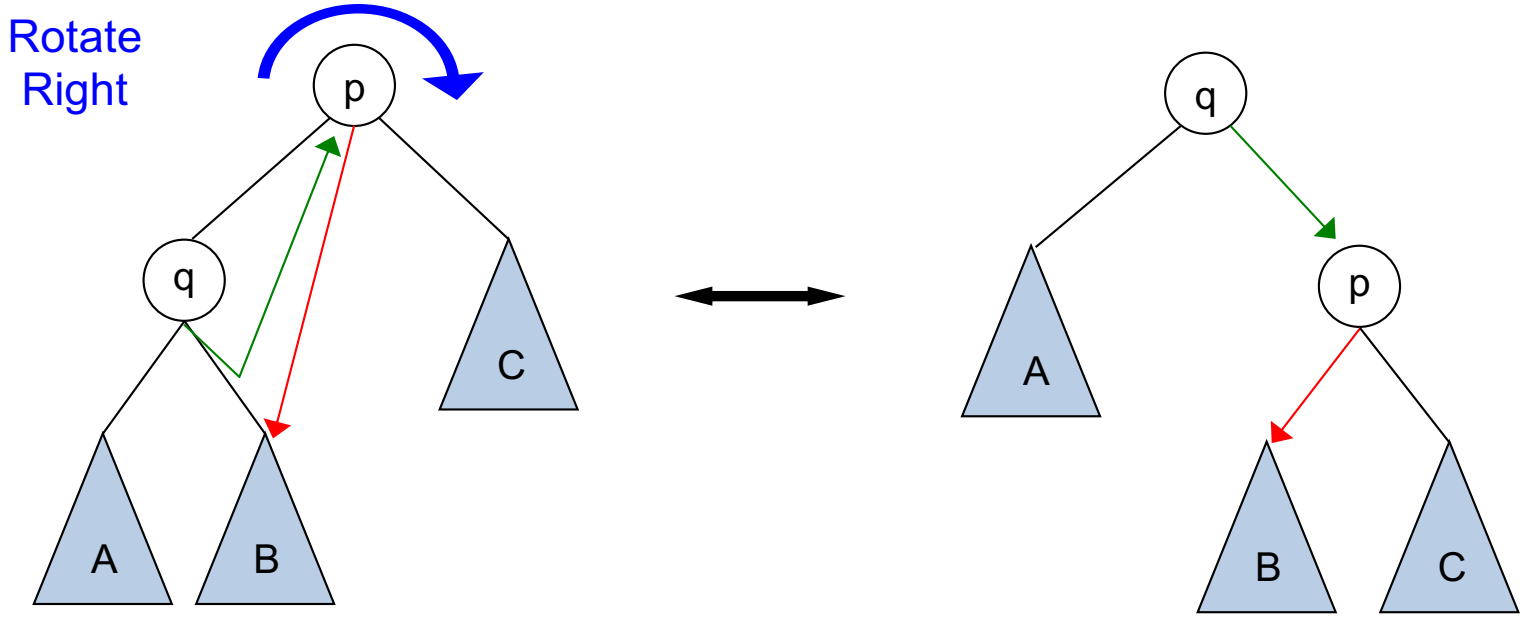
Fall A2

Fall B1

Fall B2

Algorithmen (3)

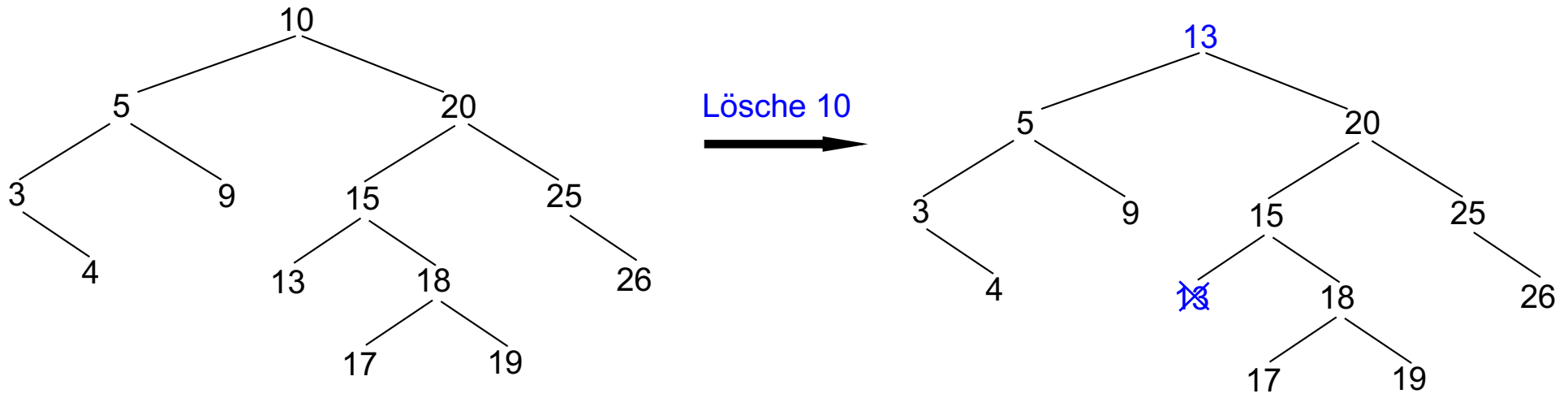
```
private Node<K,V> rotateRight(Node<K,V> p) {
        assert p.left != null;
        Node<K, V> q = p.left;
        p.left = q.right;
        q.right = p;
        p.height = Math.max(getHeight(p.left), getHeight(p.right)) + 1;
        q.height = Math.max(getHeight(q.left), getHeight(q.right)) + 1;
        return q;
}
```

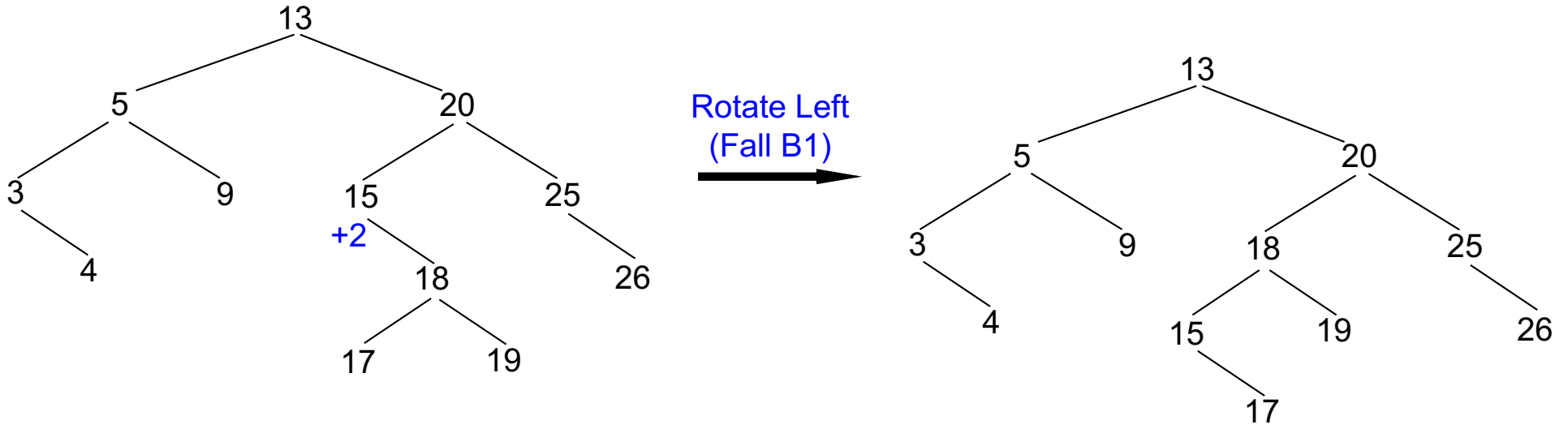


Algorithmen (4)

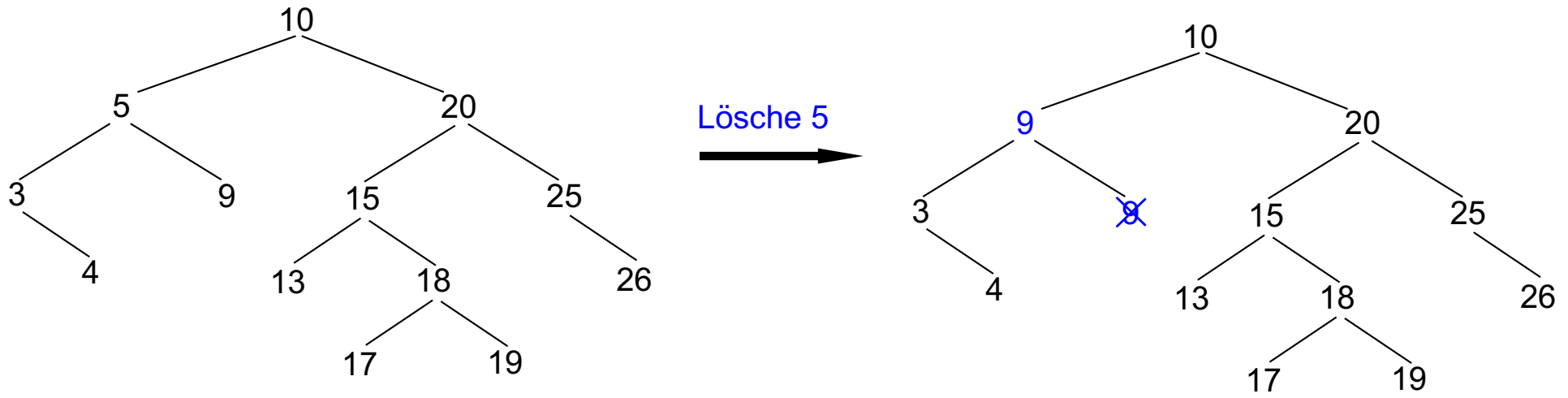
```
private Node<K,V> rotateLeft(Node<K,V> p) {
        // analog zu rotateRight
}

private Node<K,V> rotateLeftRight(Node<K,V> p) {
        assert p.left != null;
        p.left = rotateLeft(p.left);
        return rotateRight(p);
}

private Node<K,V> rotateRightLeft(Node<K,V> p) {
        assert p.right != null;
        p.right = rotateRight(p.right);
        return rotateLeft(p);
}
```

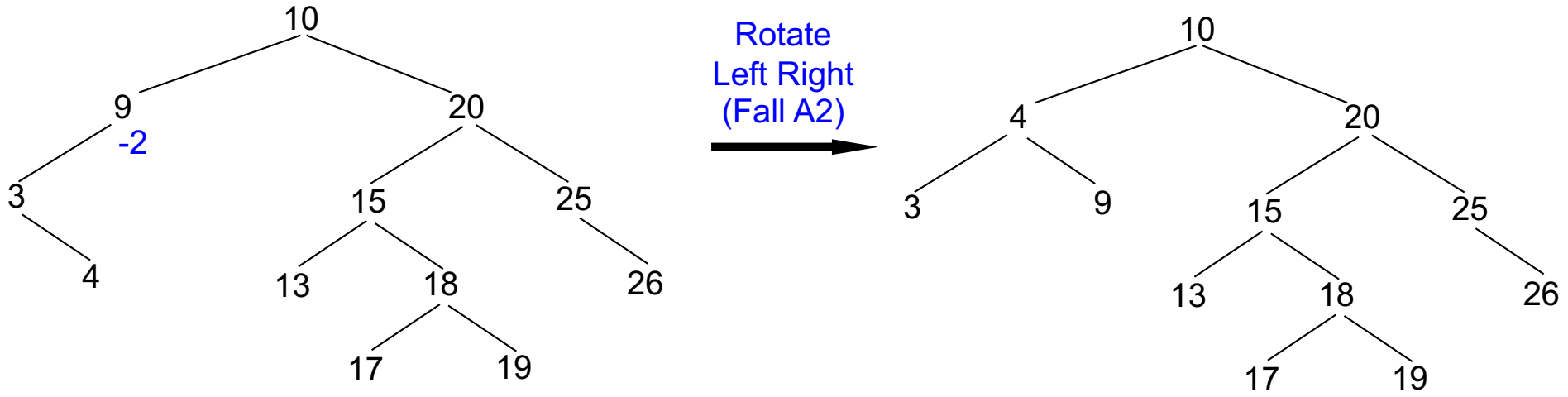
Beispiel 1 zu Löschen in AVL-Bäumen



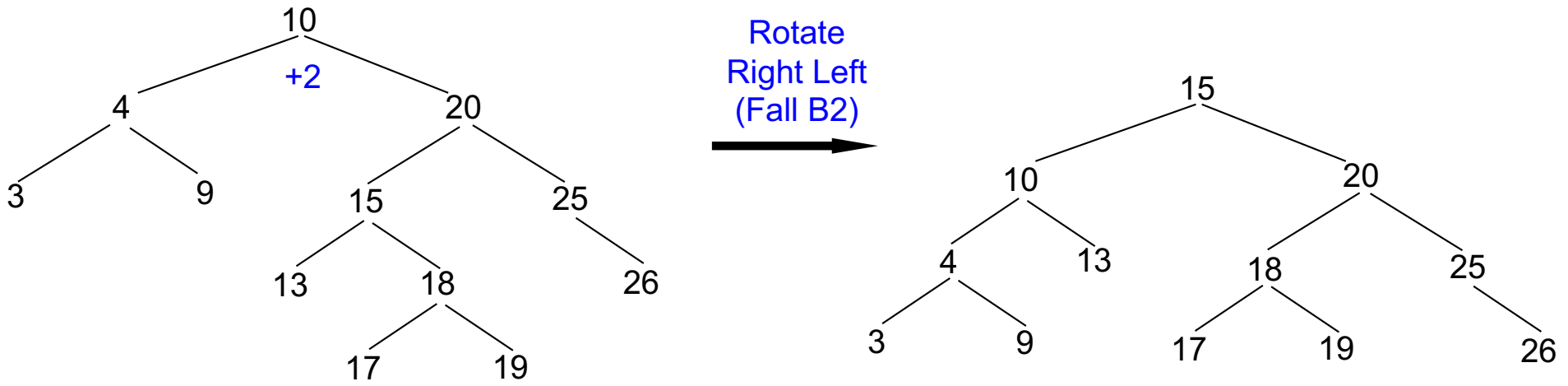


Beispiel 2 zu Löschen in AVL-Bäumen (1)





Beispiel 2 zu Löschen in AVL-Bäumen (2)



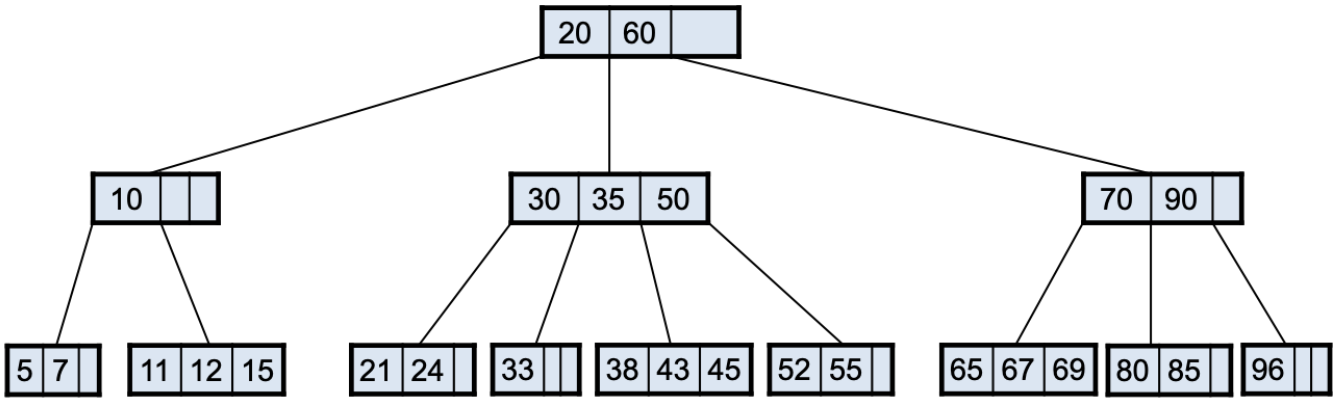
4. Balancierte Suchbäume

- AVL-Bäume
- B-Bäume
- 2-3-4-Bäume und Rot-Schwarzbäume

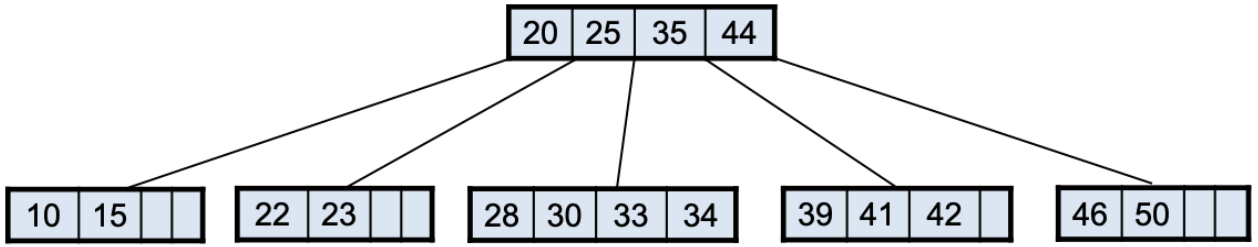
Definition von B-Bäumen der Ordnung m

1. Balancierung: Jedes Blatt hat die gleiche Tiefe.
2. Knoten enthalten mehrere Schlüssel: zwischen $\lceil m/2 \rceil - 1$ und m-1 (jeweils einschl.).
   Ausnahme Wurzel: minimal 1 Schlüssel.
3. Anzahl Kinder:
   Jeder Knoten (außer den Blättern) mit i vielen Schlüsseln hat genau i+1 Kinder.
4. Schlüsselordnung (Zick-Zack-Ordnung): nächste Folie.

B-Baum der Ordnung m = 4 und der Höhe 2



B-Baum der Ordnung m = 5 und der Höhe 1

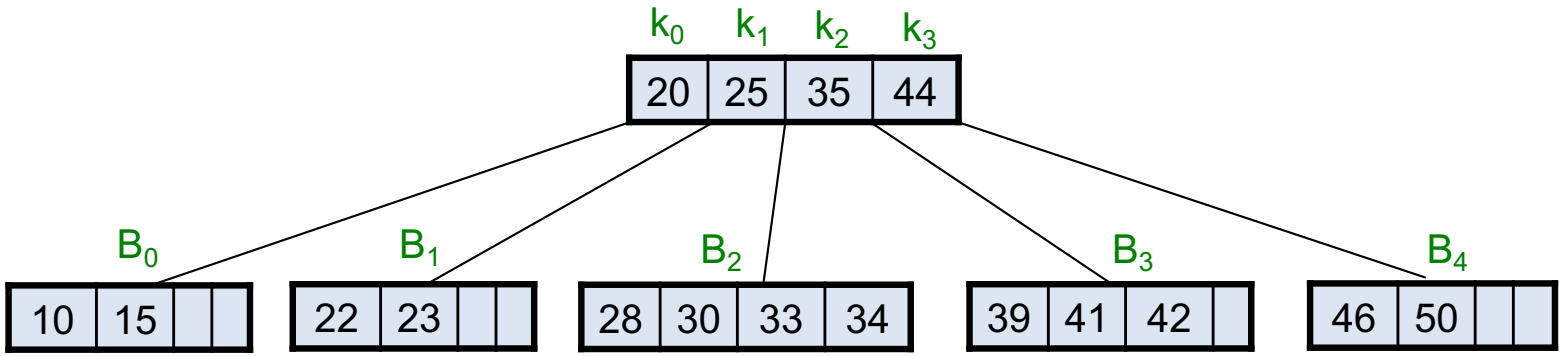


Schlüsselordnung (Zick-Zack-Ordnung)

- Für jeden Knoten K mit Schlüsseln $k_0, k_1, \ldots, k_{i-2}$ und Teilbäumen $B_0, B_1, \ldots, B_{i-1}$ gilt:

  Schlüssel in B_0 < k_0 <
  Schlüssel in B_1 < k_1 <
  Schlüssel in B_2 < k_2 <
  …
  Schlüssel in B_{i-2} < k_{i-2} <
  Schlüssel in B_{i-1} .

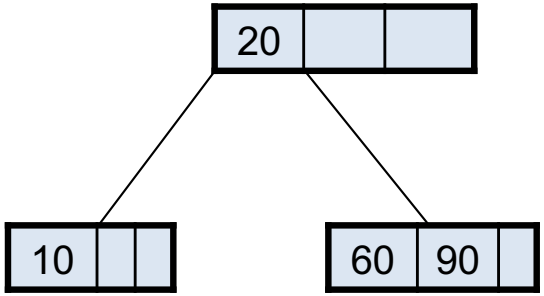
- Folgerung: In allen Knoten (inkl. Blättern) müssen die Schlüssel geordnet sein.



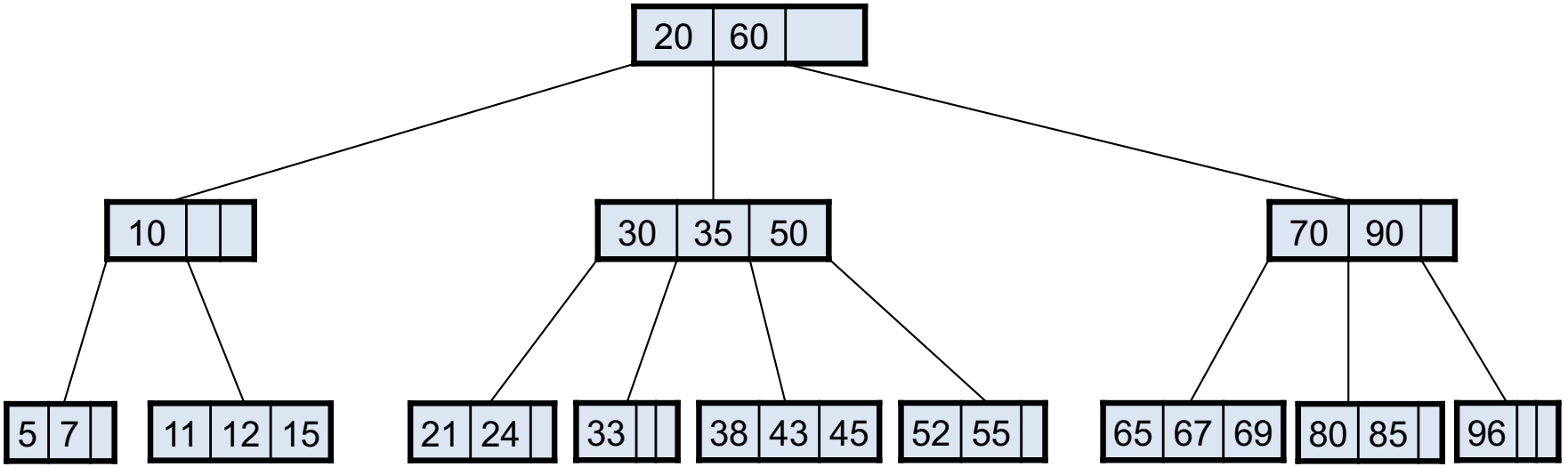
Beispiele für B-Bäume der Ordnung m = 4

B-Baum der Höhe 0

B-Baum der Höhe 1

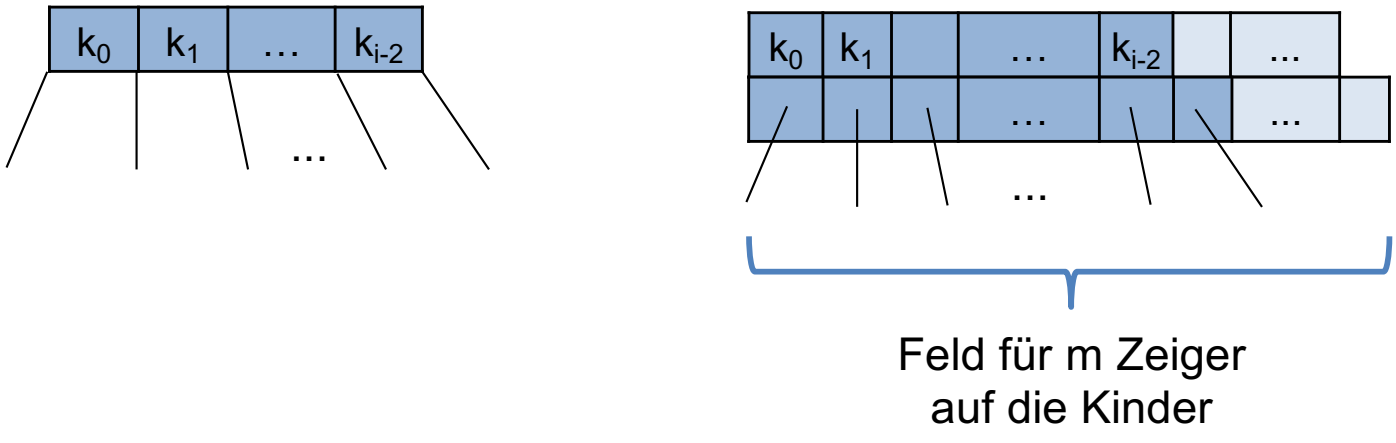


B-Baum der Höhe 2



Implementierung von B-Bäumen der Ordnung m

- Jeder Knoten besteht aus
  - einem statischen Feld der Größe m-1 für die Schlüssel und
  - einem statischem Feld der Größe m für die Zeiger auf die Kinder.

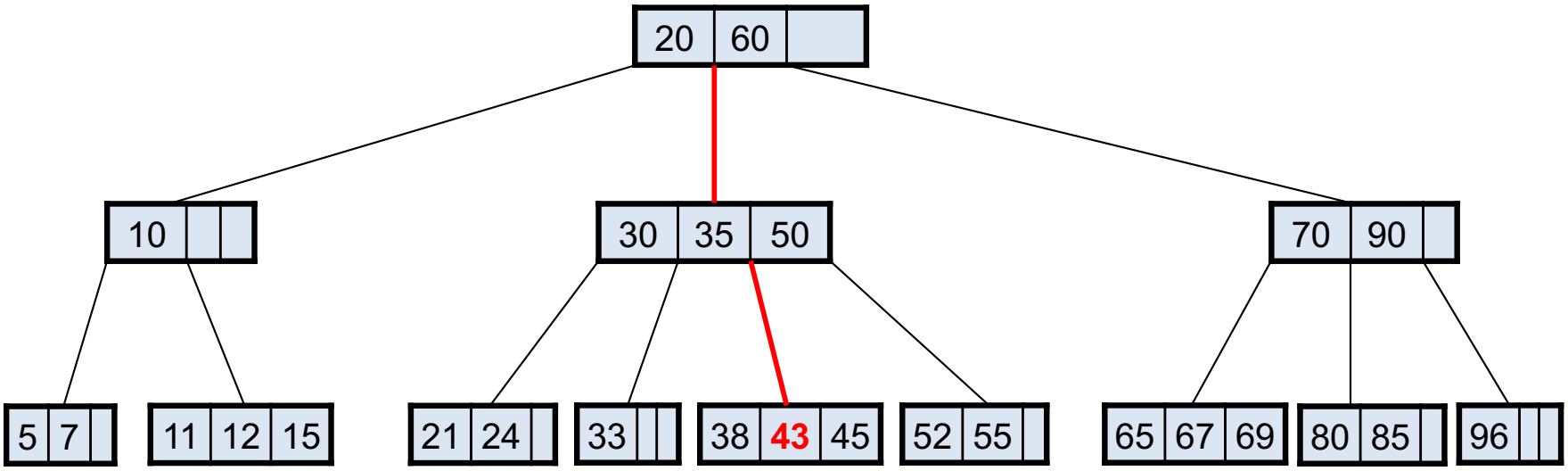


Suchen in B-Bäumen

```
V search(K key, Node<K,V> p) {

    if (p == 0)
        return null;      // nicht gefunden;

    suche key in Schlüsselmenge von Knoten p (beispielsweise mit binärer Suche);
    if (key gefunden)
        return Daten;

    p = Teilbaum, in dem key liegen könnte;
    return search(key, p);
}
```

Rekursiver Aufruf

Beispiel: Suchen nach k = 43

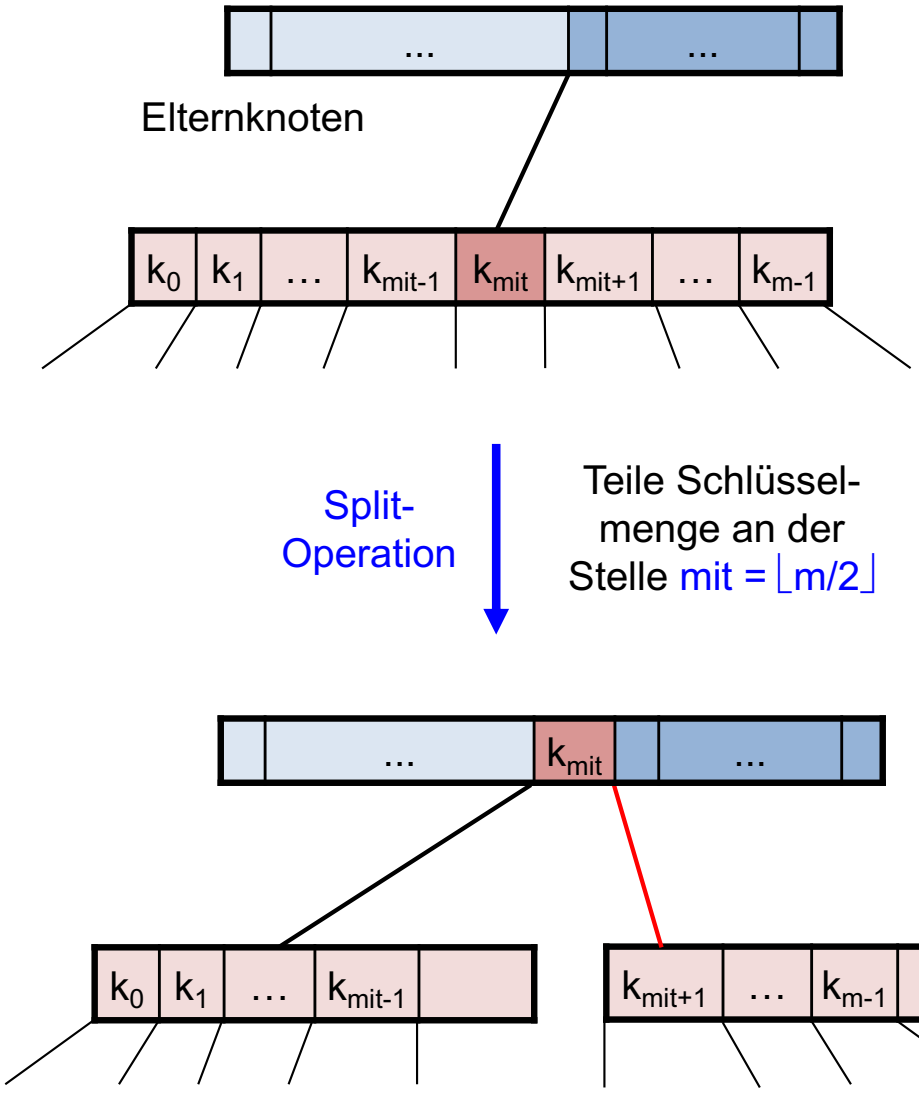


Einfügen in B-Bäumen

```
V insert(K key, V value, Node<K,V> p) {
    suche Schlüssel key im Baum p;
    if (key gefunden) {
        speichere Daten in oldValue;
        ersetze Daten durch value;
        return oldValue;
    }
    füge key, value an der Blattstelle ein,
    wo die Suche beendet wurde;
    if (kein Schlüsselüberlauf)
        return null;
    for (alle Knoten k vom aktuellen Knoten bis zur Wurzel)  {
        if (Schlüsselüberlauf bei k)
            führe Split-Operation für k durch;
        else
            return null;
    }
}
```

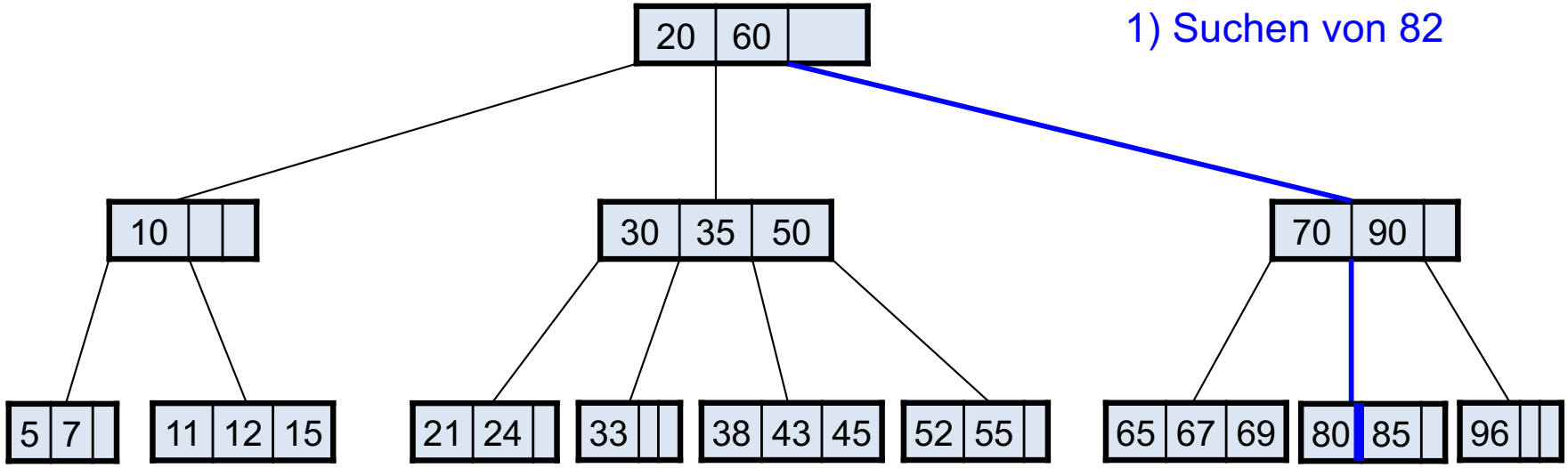
- Einfügen immer in einem Blatt.
- Schlüsselüberlauf, falls Anzahl der Schlüssel gleich m.
- insert kann wie bei AVL-Bäumen rekursiv realisiert werden.
  Die Behandlung der Schlüssel-überläufe geschieht dann beim rekursiven Aufstieg.
- Muss die Wurzel gesplittet werden, dann entsteht eine neue Wurzel (siehe Beispiel)

Split-Operation bei Schlüsselüberlauf

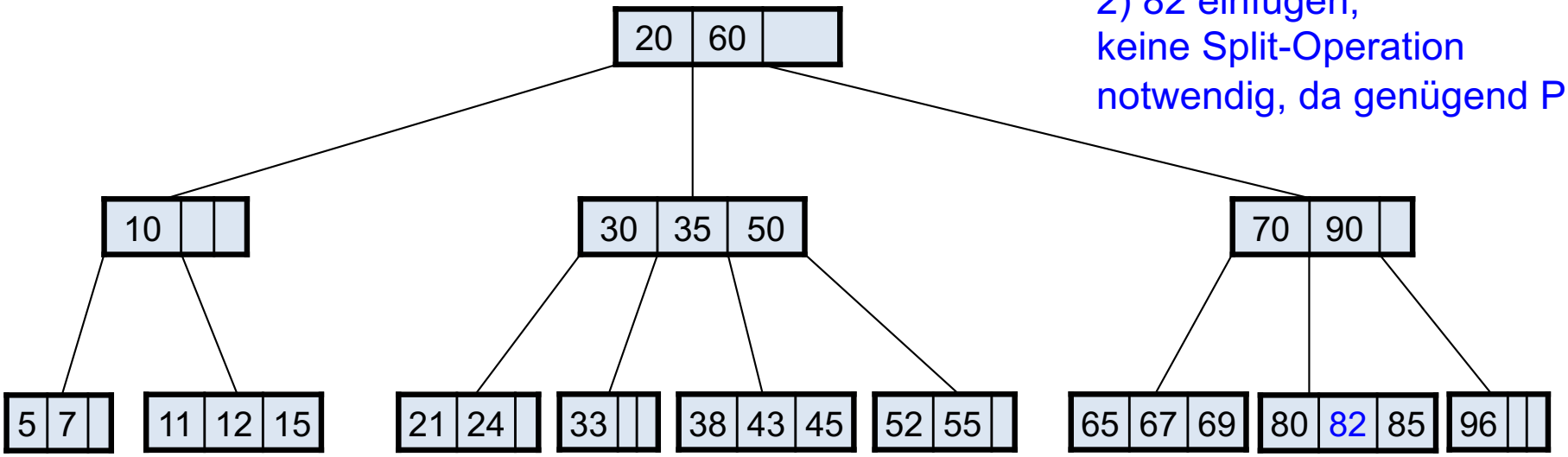


Aktueller Knoten mit Schlüsselüberlauf:
Knoten enthält genau m Schlüssel

Beispiel 1: Einfügen von 82

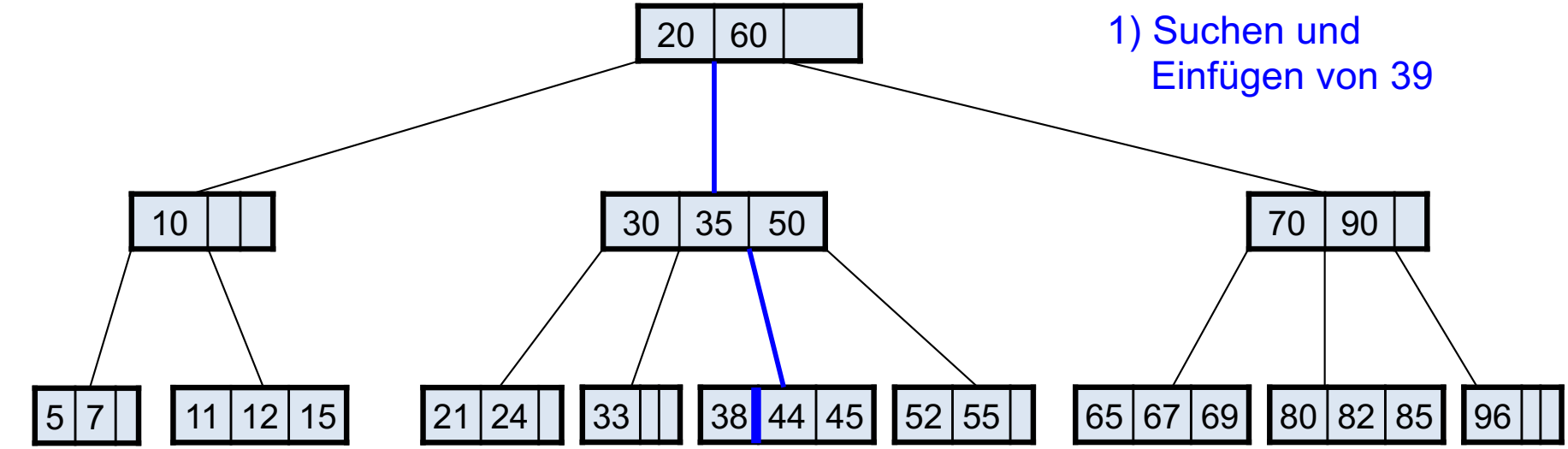


1) Suchen von 82

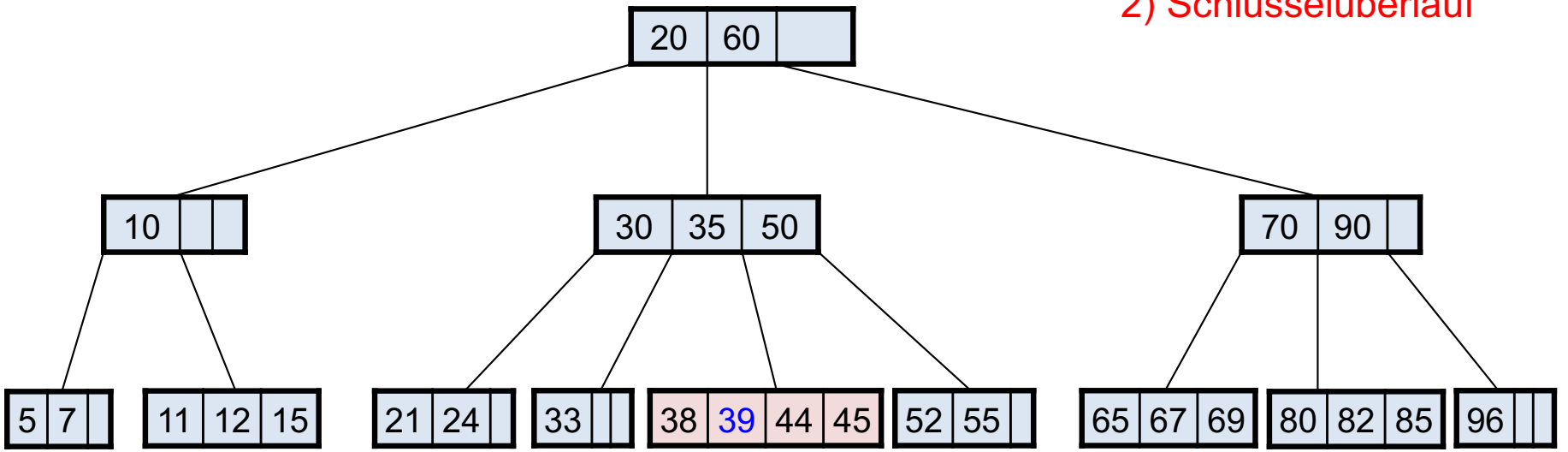


2) 82 einfügen;
keine Split-Operation
notwendig, da genügend Platz

Beispiel 2: Einfügen von 39

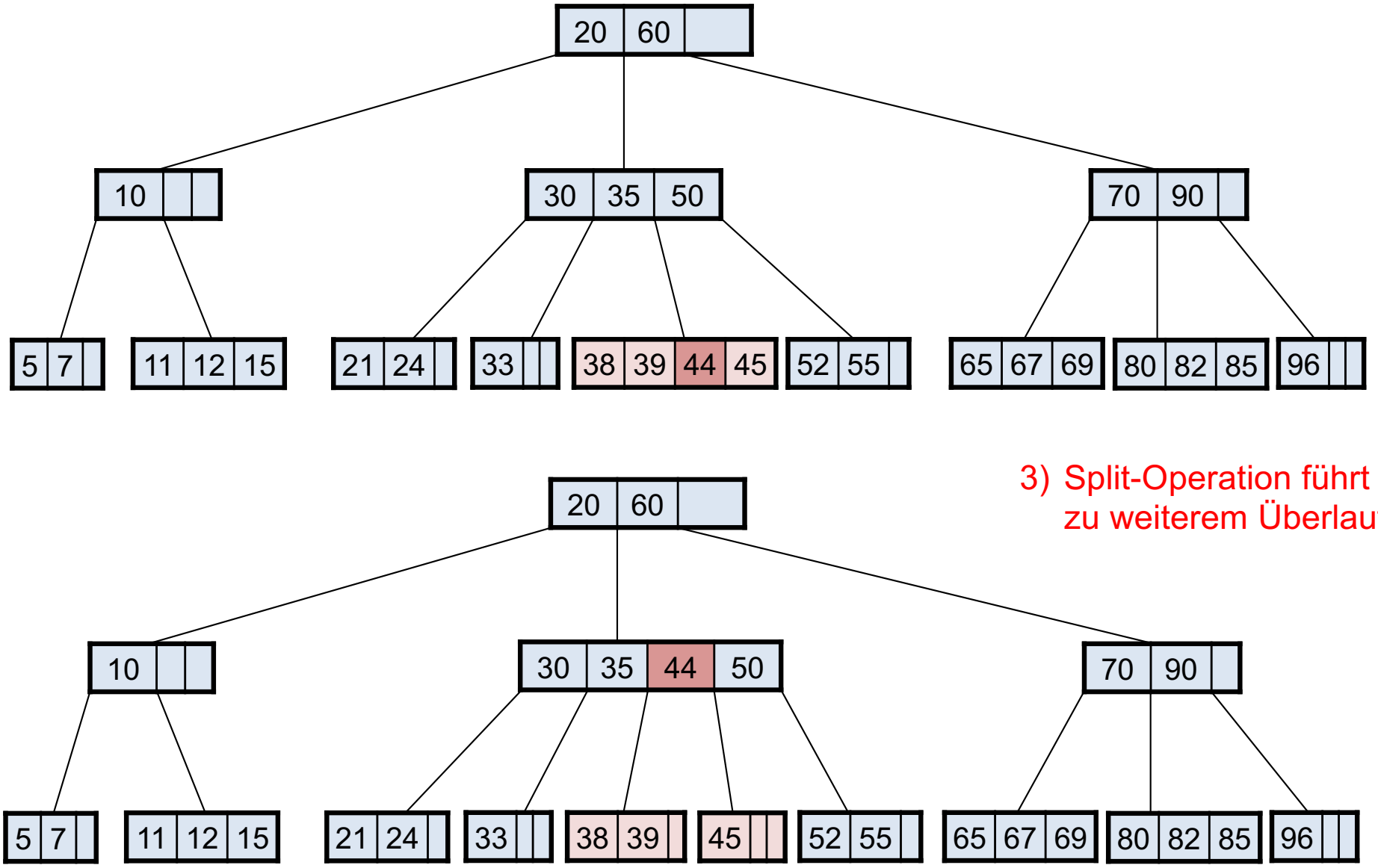


1) Suchen und Einfügen von 39

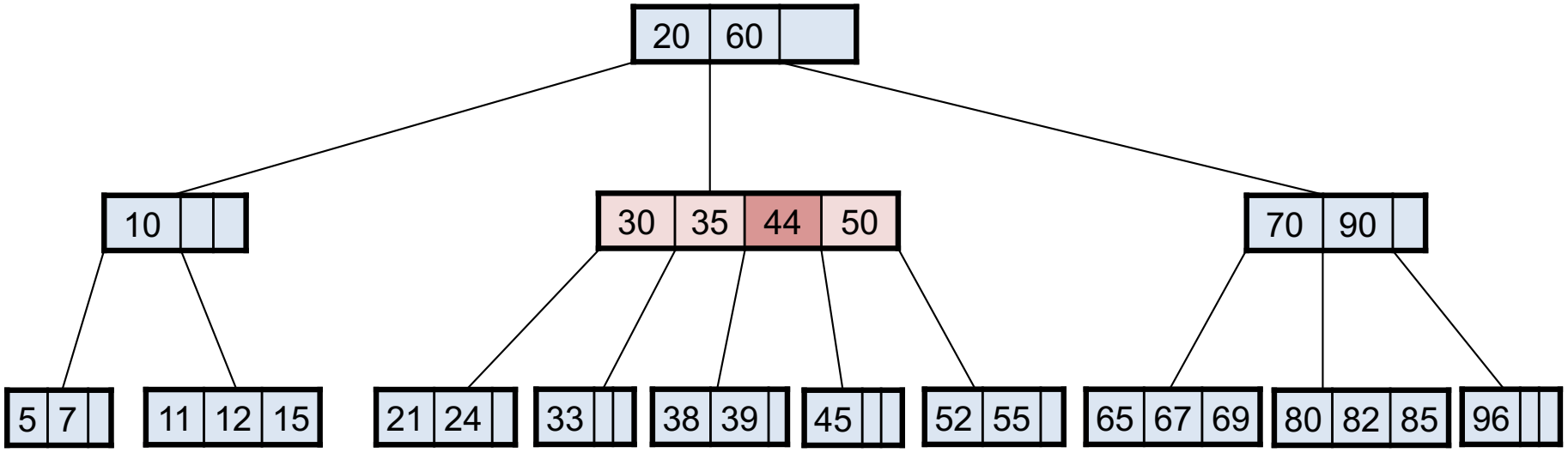


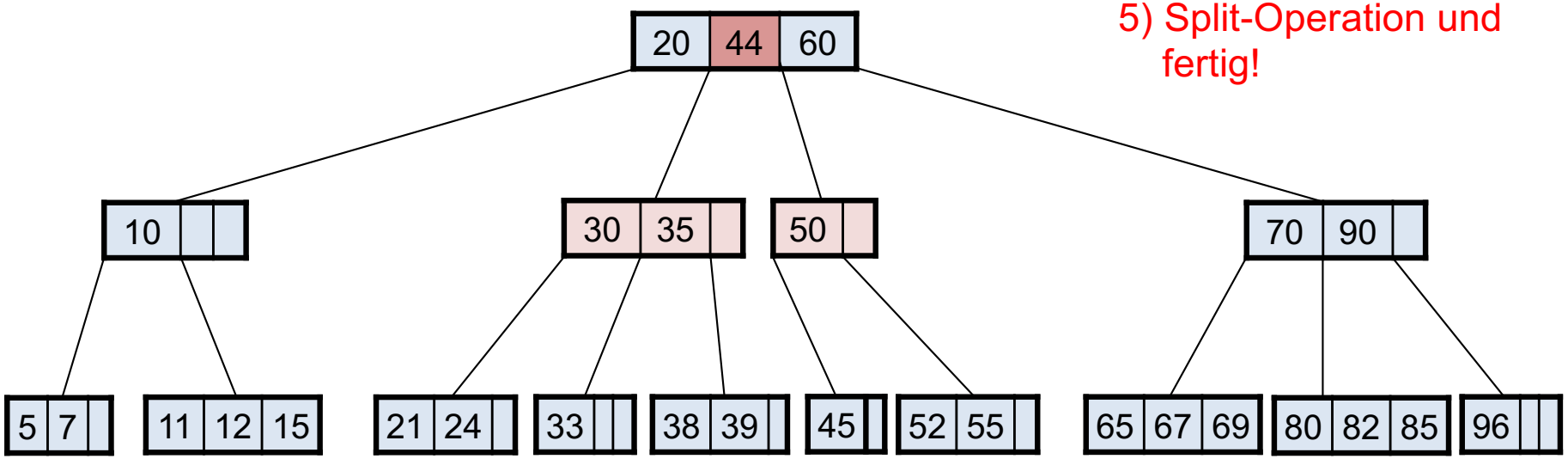
2) Schlüsselüberlauf

Beispiel 2: Einfügen von 39 (Fortsetzung)



Beispiel 2: Einfügen von 39 (Fortsetzung)





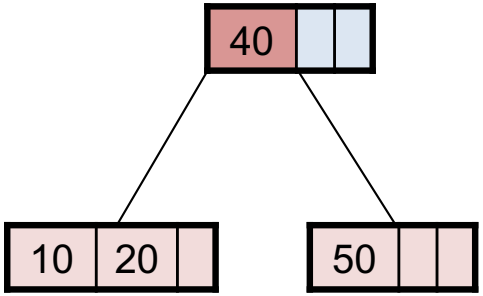
5) Split-Operation und fertig!

Beispiel 3: Erzeugen eines neuen Knotens bei einer Einfüge-Operation

B-Baum mit genau einem Knoten mit 3 Schlüsseln

10 wird gesucht und eingefügt. Überlauf!

Split-Operation anwenden, da nicht genügend Platz. Dabei wird ein neuer Knoten erzeugt.

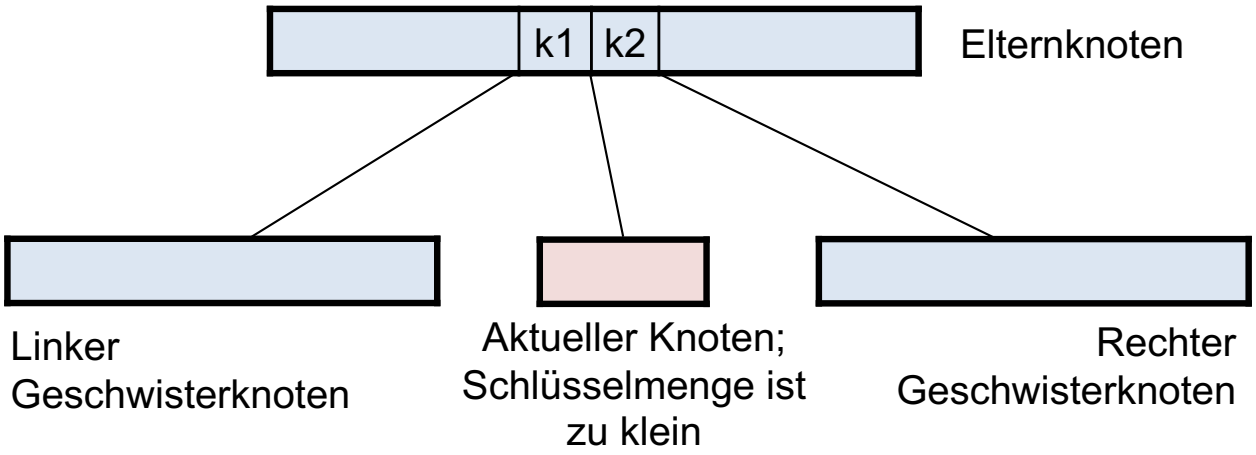


Löschen in B-Bäumen

```
V remove(K key, Node<K,V> p) {

    suche Schlüssel key im Baum;

    if (key nicht gefunden)
        return null;

    if (key befindet sich im Blatt)
        speichere Daten in oldValue und lösche key;
    else
        speichere Daten in oldValue und ersetze key (mit value)
        durch nächst größeren Schlüssel key' (mit value') und lösche key';
        // key' befindet sich im rechten Teilbaum von key ganz links in einem Blatt

    for (alle Knoten k vom aktuellen Knoten bis zur Wurzel)  {
        if ( Schlüsselunterlauf bei k) {
            if (Geschwisterknoten kann Schlüssel abgeben)
                Übernahme von Schlüsseln vom Geschwisterknoten;
            else
                Verschmelze Knoten k mit einem Geschwisterknoten;
        }
        else
            return oldValue;
    }
}
```

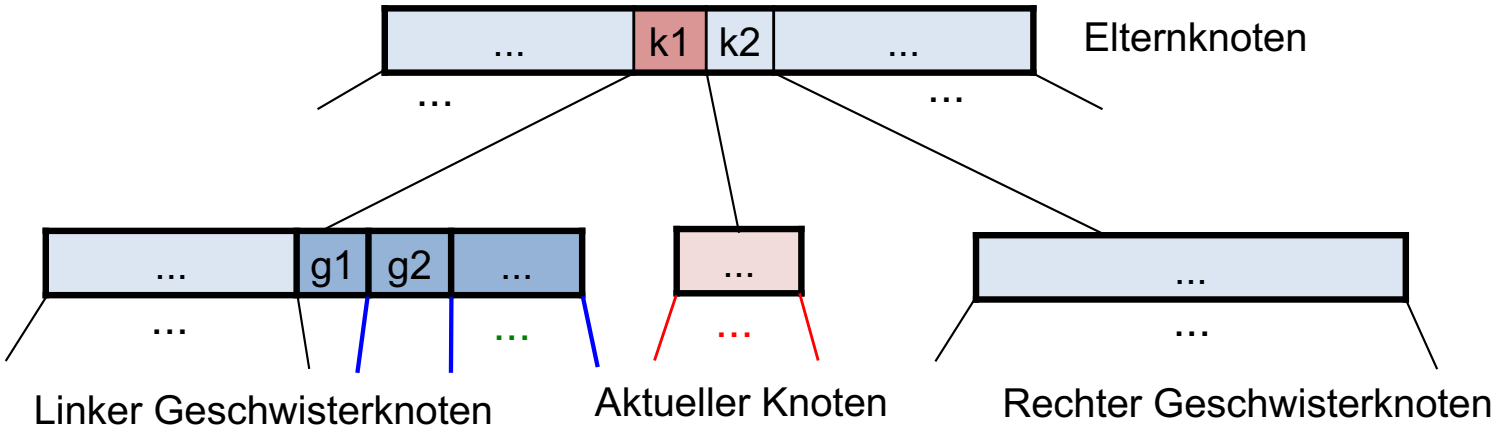
- Löschen immer in einem Blatt.
- Schlüsselunterlauf, falls Anzahl der Schlüssel kleiner als $\lceil m/2 \rceil - 1$.
- remove kann wie bei AVL-Bäumen rekursiv realisiert werden.
  Die Behandlung der Schlüsselunterläufe geschieht dann beim rekursiven Aufstieg.
- Hat die Wurzel genau 2 Kinder, die verschmolzen werden müssen, dann wird die Wurzel leer und muss entfernt werden (siehe Beispiel).

Behandlung eines Schlüsselunterlaufs: Übernahme von Schlüssel

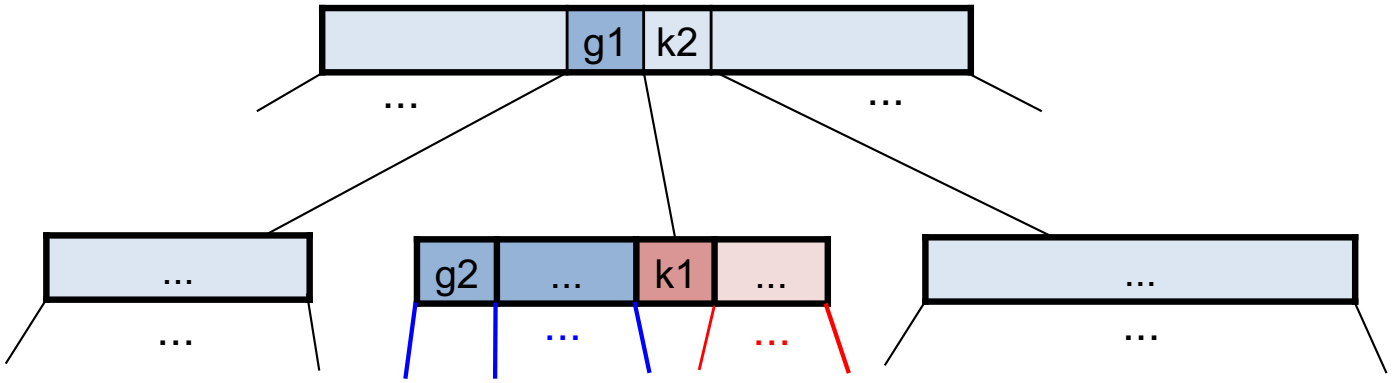


- Prüfe zuerst, ob linker Geschwisterknoten Schlüssel abgeben kann. Falls nicht, dann prüfe rechten Knoten.
- Abgabe von Schlüsseln, so dass Knoten in etwa gleich groß werden.
- Verfahren kann dann abgebrochen werden, da im Elternknoten kein Schlüsselunterlauf entstehen kann.

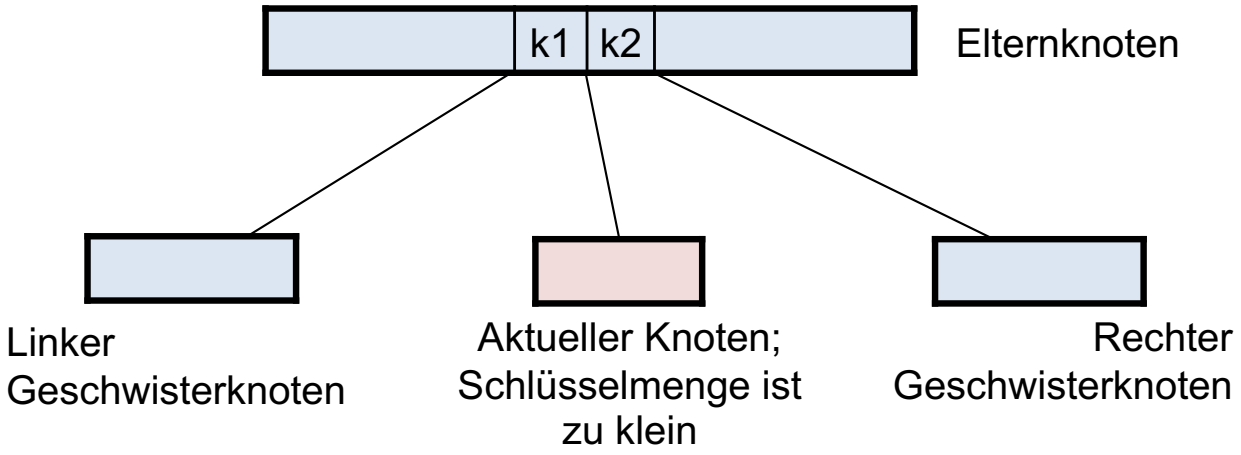
Übernahme von Schlüssel (hier vom linken Geschwisterknoten)



Linker Geschwisterknoten gibt dem aktuellen Knoten gerade soviel Schlüssel (und Kindzeiger) ab, so dass sich die Knotenanzahl um maximal 1 unterscheidet.

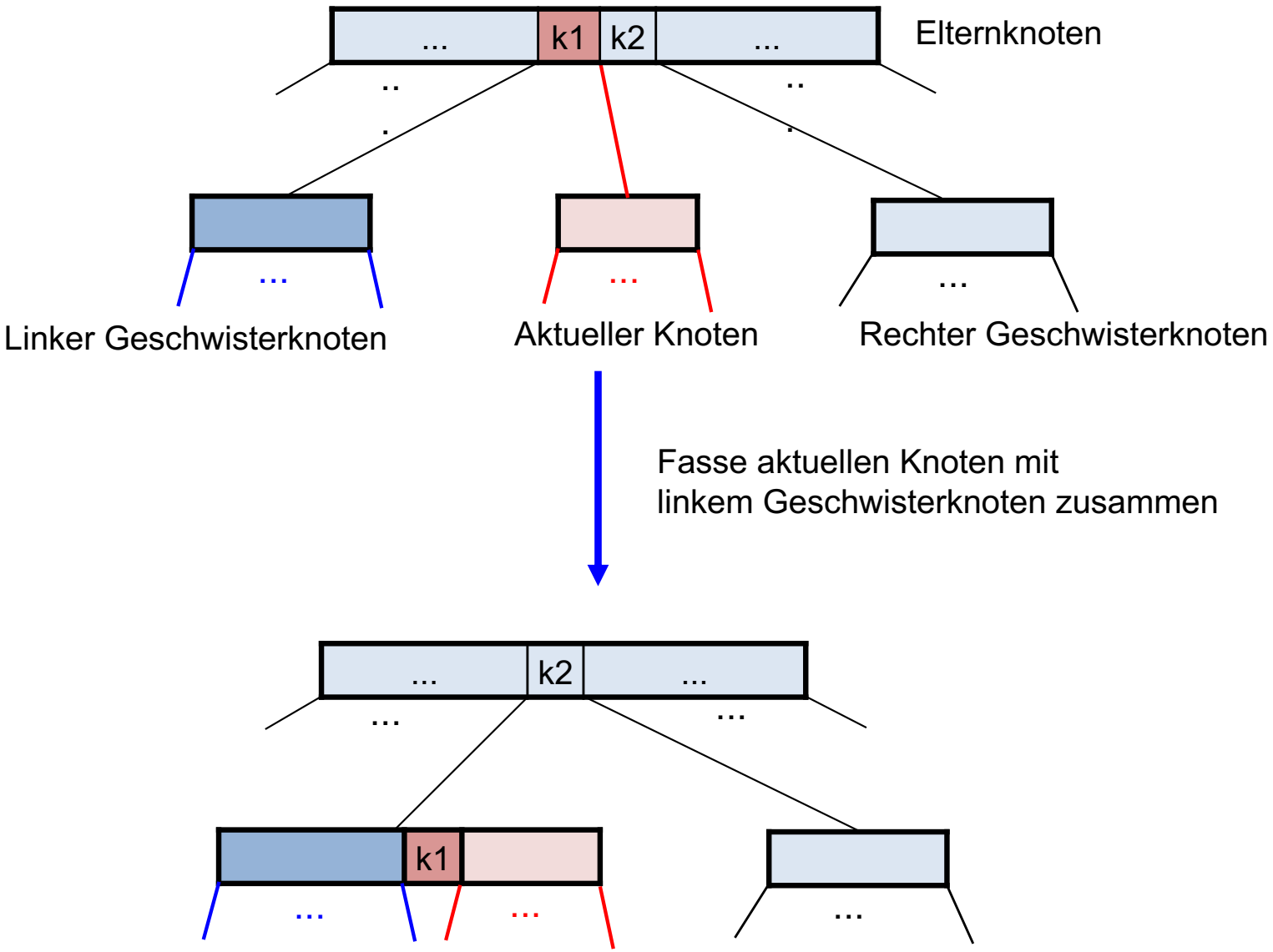


Behandlung eines Schlüsselunterlaufs: Verschmelzung



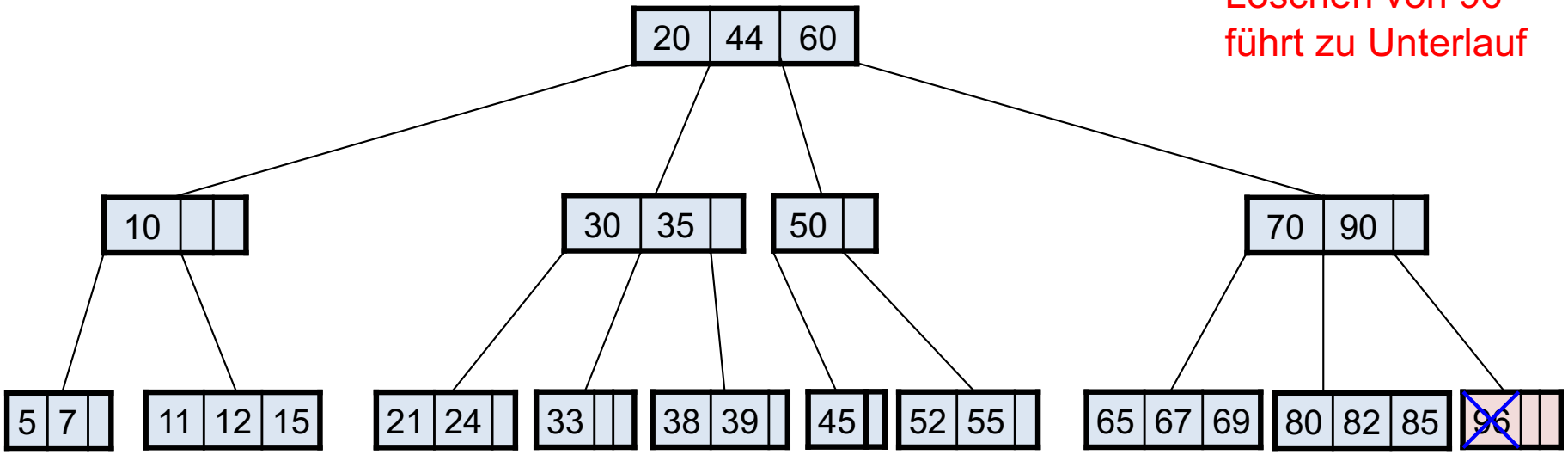
- Beide Geschwisterknoten haben zu wenig Schlüssel, um Schlüssel abgeben zu können.
- Führe daher Verschmelzung mit einem Geschwisterknoten durch.
- Fasse aktuellen Knoten mit linkem Geschwisterknoten (falls vorhanden) zusammen. Sonst fasse mit rechtem Geschwisterknoten zusammen.
- Bei Elternknoten wird ein Schlüssel nach unten verschoben. Daher kann nun beim Elternknoten ein Schlüsselunterlauf vorkommen. Führe daher Verfahren gegebenenfalls rekursiv beim Elternknoten fort.

Verschmelzung (hier mit linkem Geschwisterknoten)

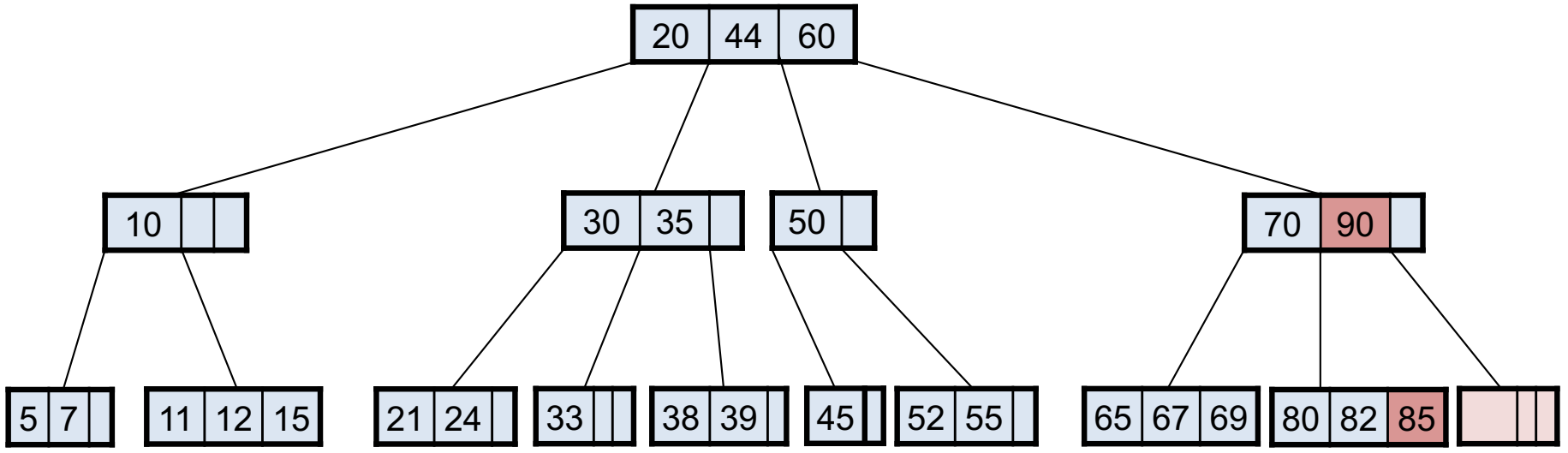


Beispiel 1: Löschen von 96

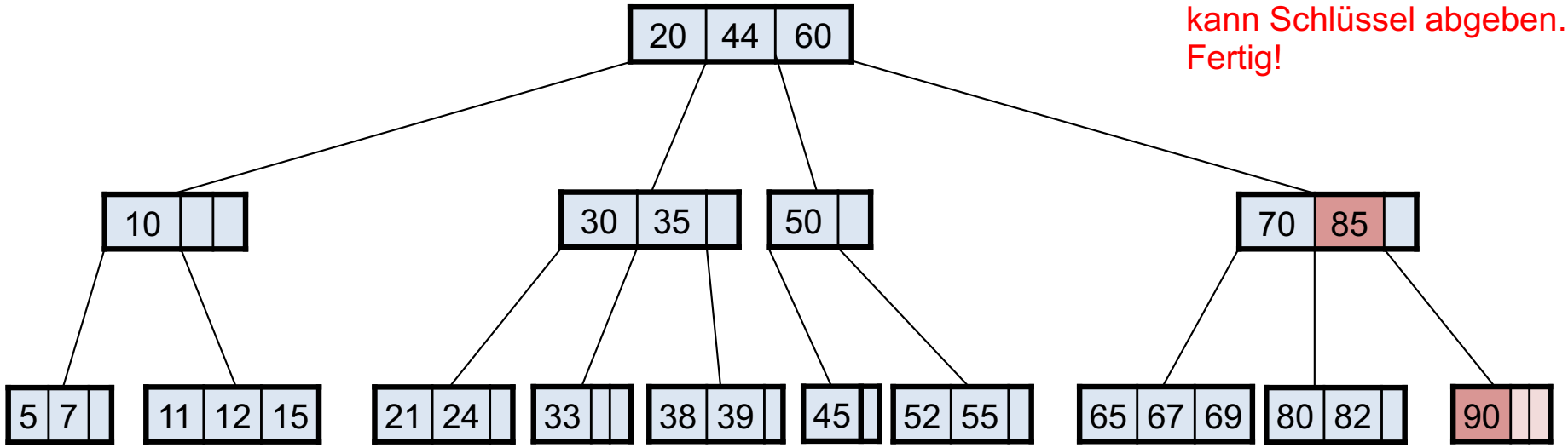
1) Suchen und Löschen von 96 führt zu Unterlauf



Beispiel 1: Löschen von 96 (Fortsetzung)

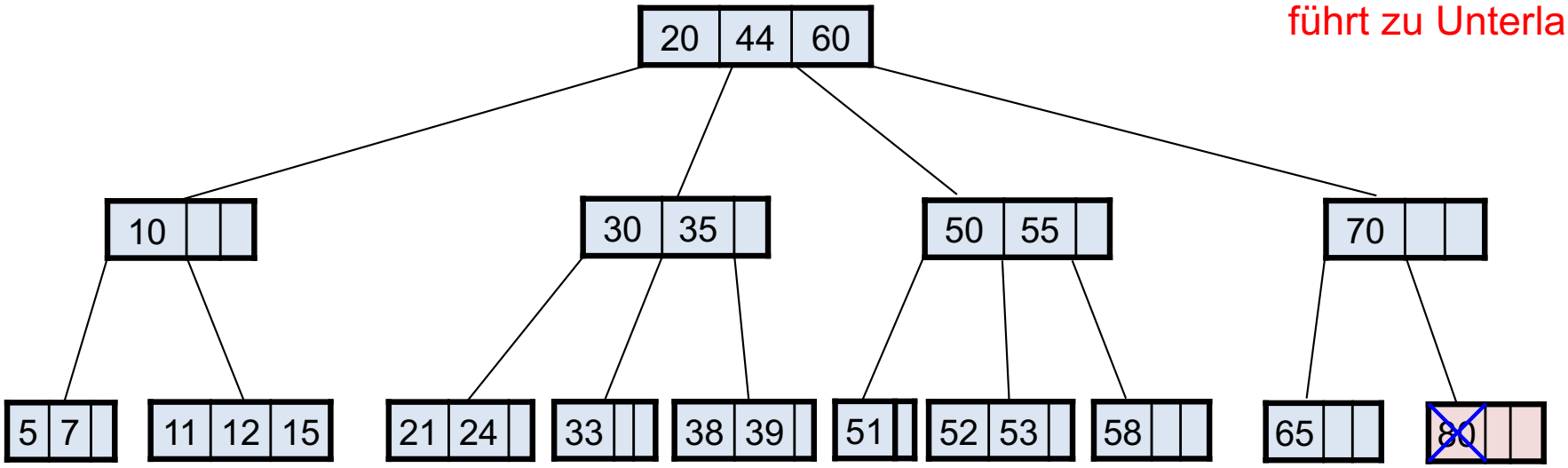


2) Linker Geschwisterknoten kann Schlüssel abgeben. Fertig!

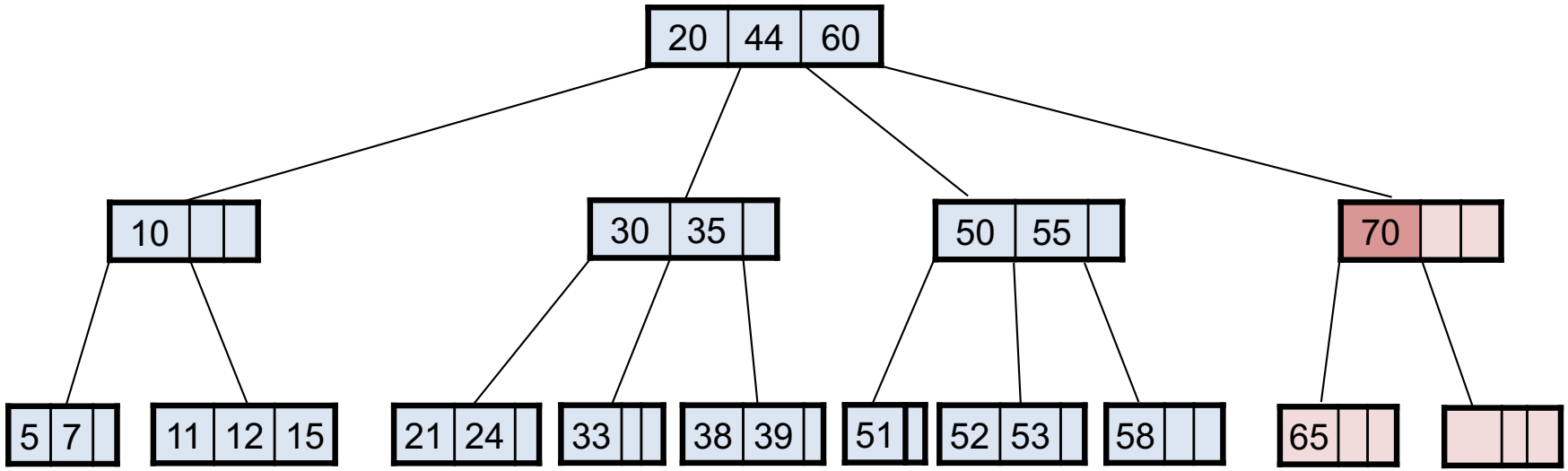


Beispiel 2: Löschen von 80

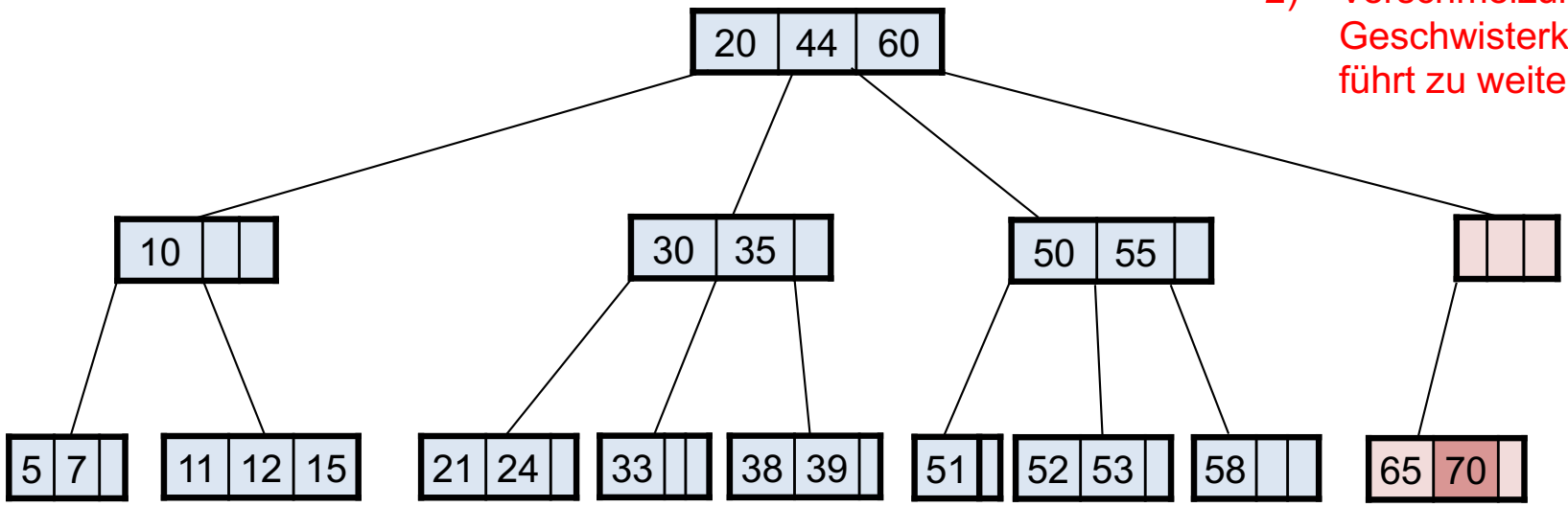
1) Suchen und Löschen von 80 führt zu Unterlauf



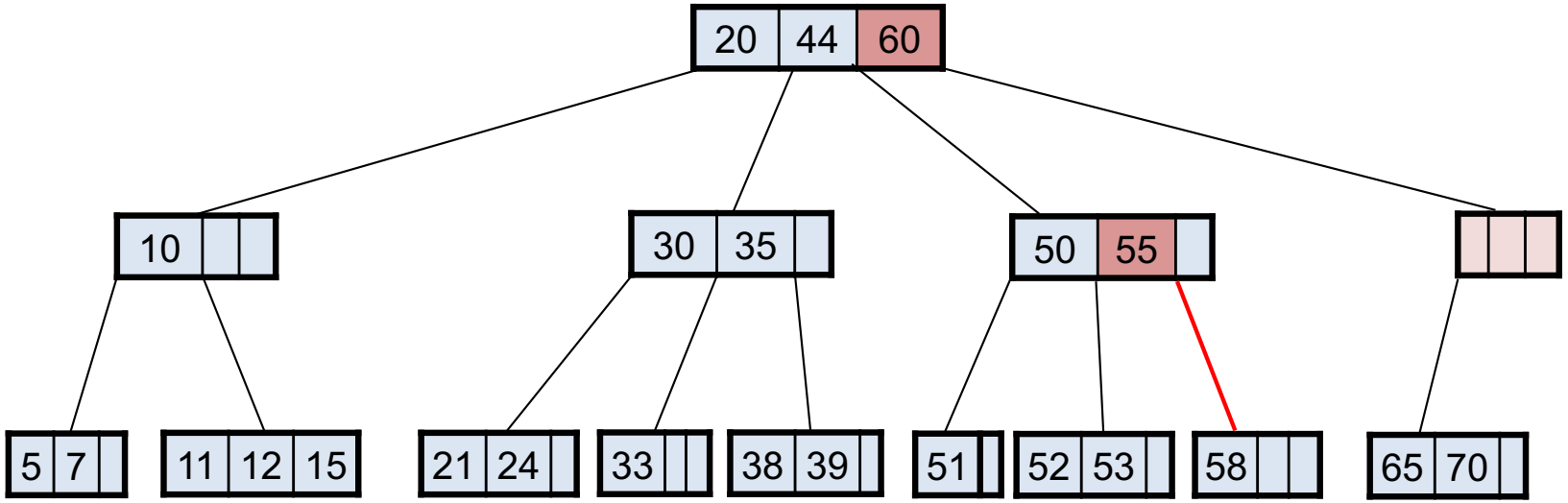
Beispiel 2: Löschen von 80 (Fortsetzung)



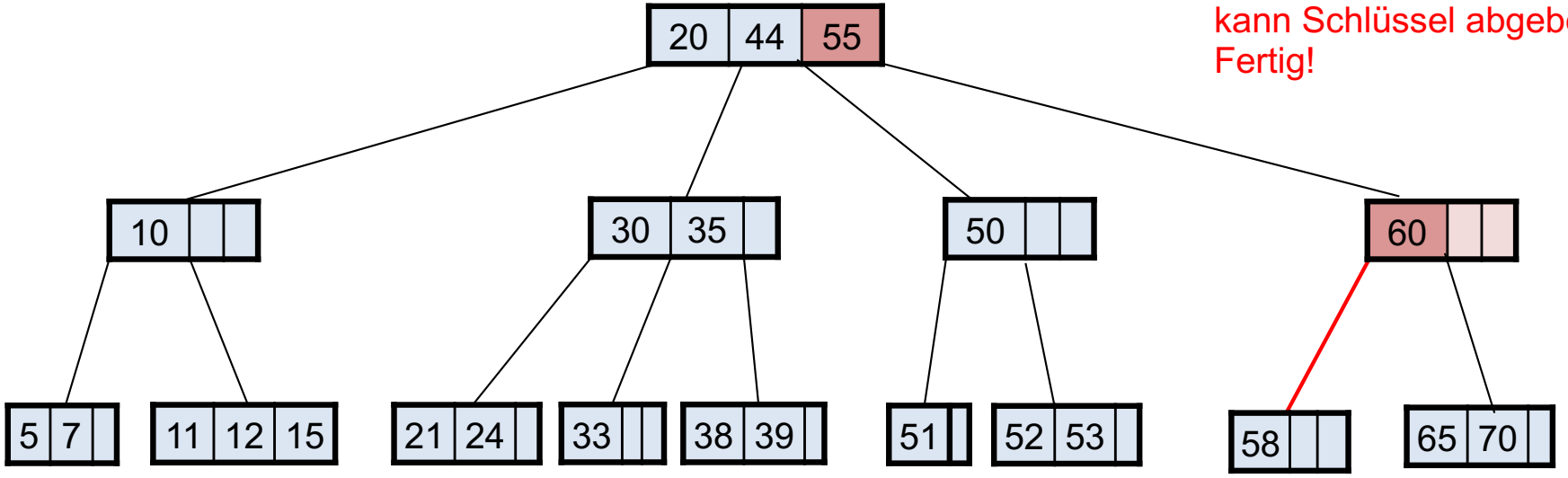
2) Verschmelzung mit linkem Geschwisterknoten führt zu weiterem Unterlauf.



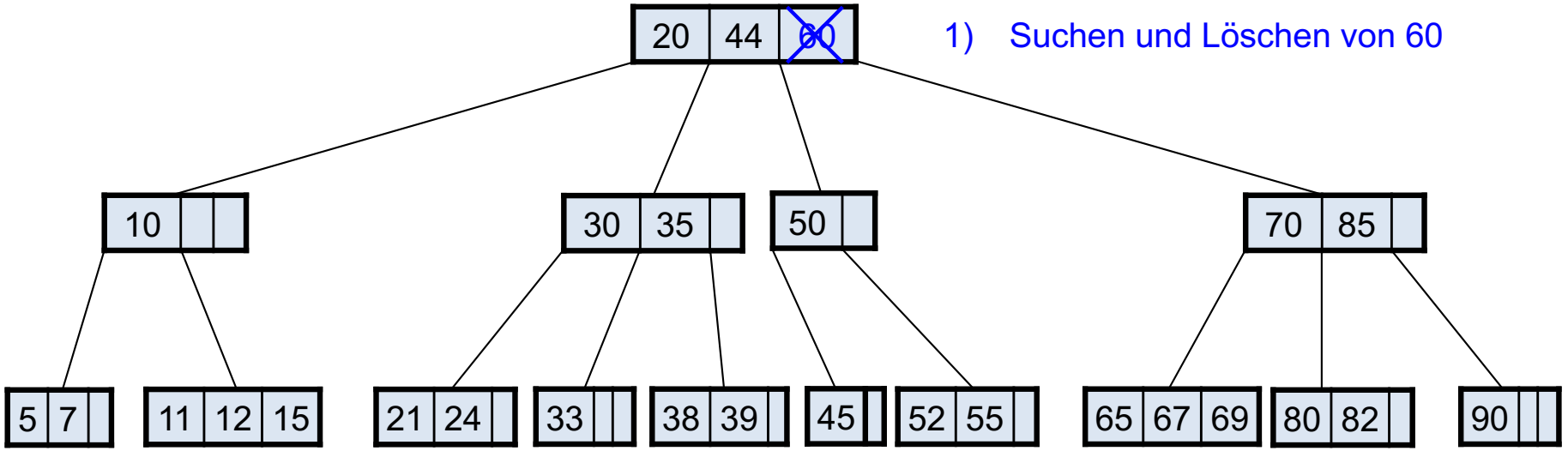
Beispiel 2: Löschen von 80 (Fortsetzung)



3) Linker Geschwisterknoten kann Schlüssel abgeben. Fertig!



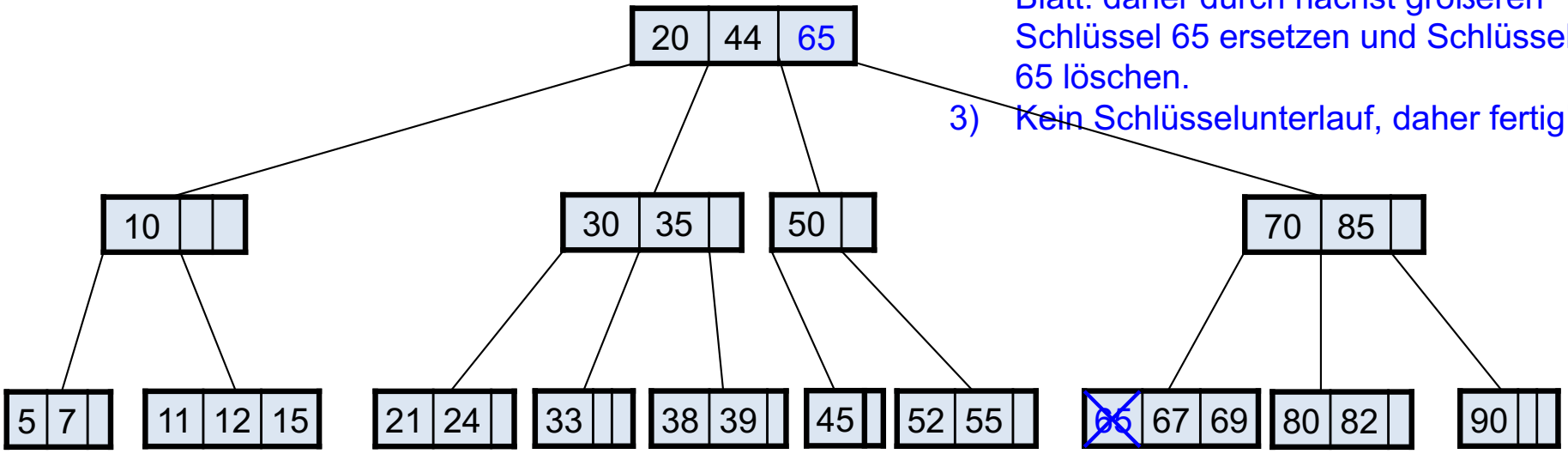
Beispiel 3: Löschen von 60



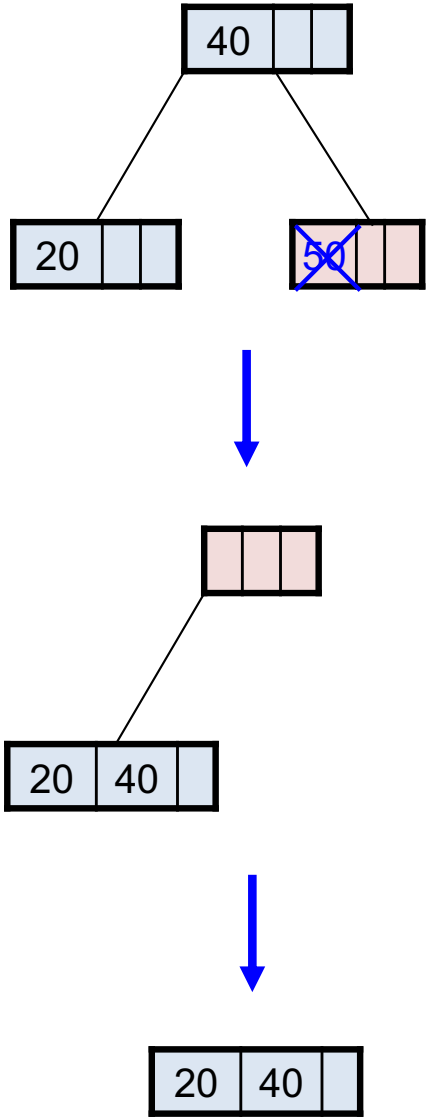
1) Suchen und Löschen von 60

2) Schüssel 60 befindet sich nicht in Blatt: daher durch nächst größeren Schlüssel 65 ersetzen und Schlüssel 65 löschen.

3) Kein Schlüsselunterlauf, daher fertig!



Beispiel 4: Wurzel muss entfernt werden



1) Suchen und Löschen von 50

2) Schlüsselunterlauf;
Verschmelzung mit
Geschwisterknoten

3) Schlüsselunterlauf in der Wurzel;
Wurzel entfernen

Laufzeit

Höhe eines B-Baumes mit n Schlüsseln:

- Im schlechtesten Fall hat die Wurzel 2 Kinder und jeder andere Nicht-Blatt-Knoten $\lceil m/2 \rceil$ viele Kinder.
- Damit ergibt sich eine maximale Höhe von (siehe [Ottmann u. Widmayer]):

$$h \leq \log_{\lceil m/2 \rceil} (n+1)/2$$

Beispiel:

- Für m = 1024 und n = 10^8 ergibt sich eine maximale Höhe von $h \leq 2.84$

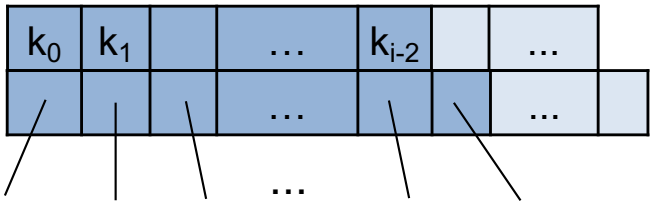
Aufwand für Suchen, Einfügen und Löschen:

- Da die Höhe eines B-Baums durch $\log_{\lceil m/2 \rceil} (n+1)/2$ beschränkt ist, ergibt sich eine maximale Laufzeit von:

$$T(n) = O(\log_{\lceil m/2 \rceil} (n))$$

Speicherplatzausnutzung

- Bei einer Implementierung mit statischen Feldern der Größe m stellt sich die Frage der Speicherplatzausnutzung.



- Wenn eine zufällig gewählte Folge von n Schlüsseln in einen anfangs leeren B-Baum der Ordnung m eingefügt werden, dann ist eine Speicherplatzausnutzung zu erwarten von [Ottmann u. Widmayer]):

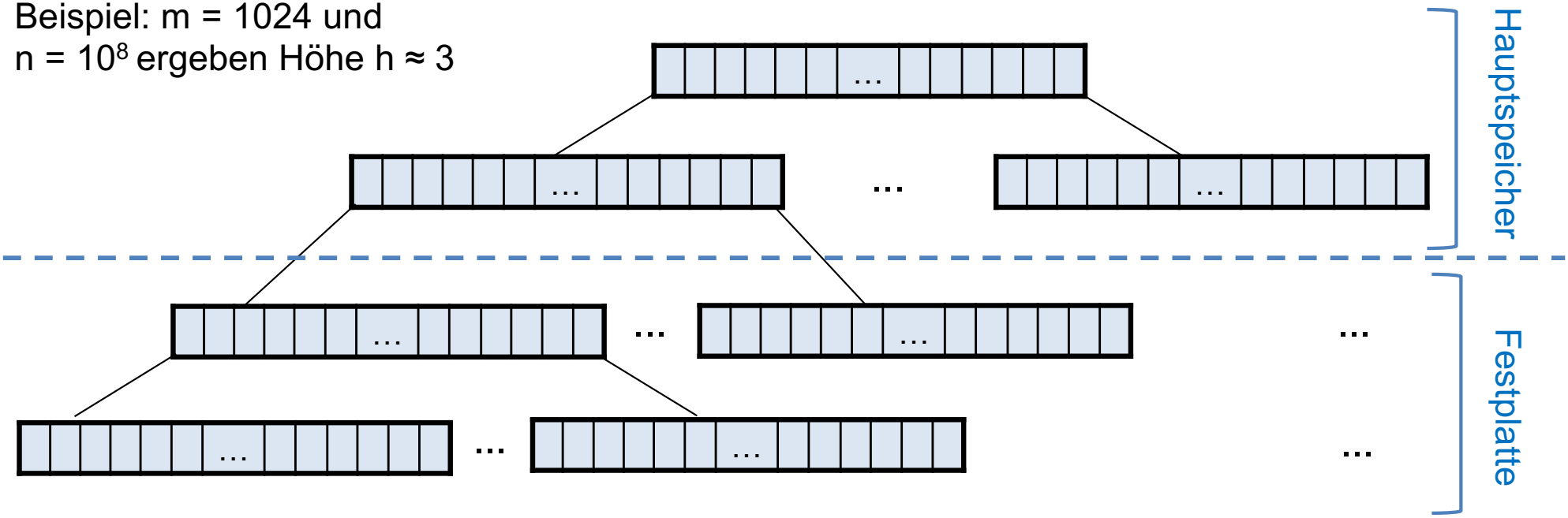
  $\ln(2) \approx 69\%$.

- Falls eine absteigend oder aufsteigend sortierte Folge in einen leeren B-Baum einfügt wird, ergibt sich eine besonders schlechte Speicherplatzausnutzung, die aber immer noch bei ca. 50% liegt.

Anwendung: Externe Suchverfahren

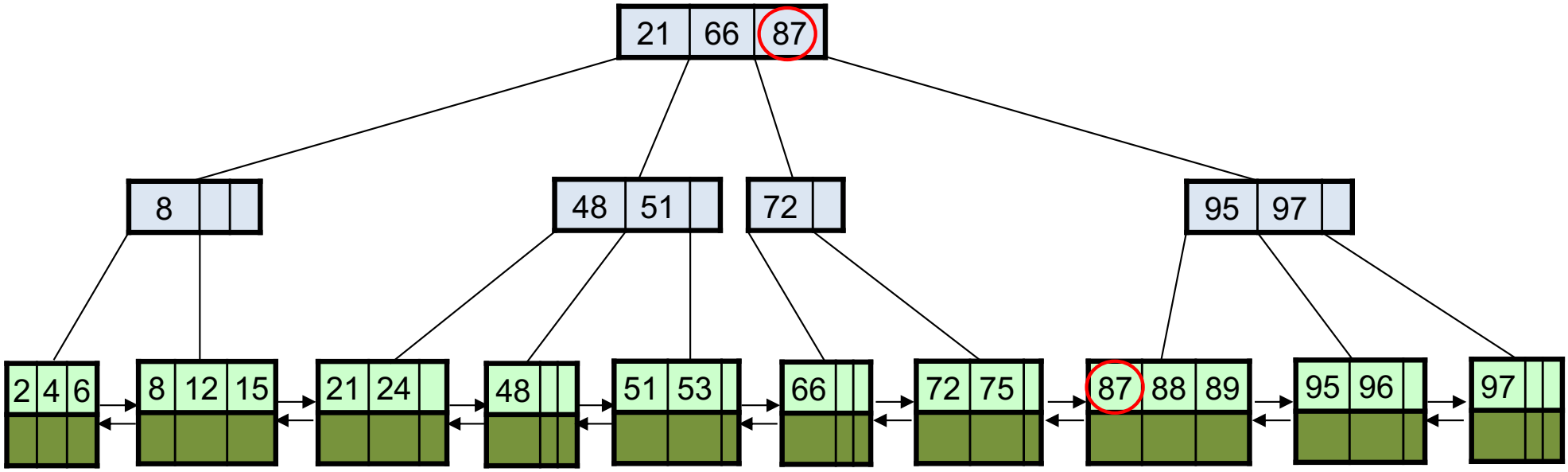
- Die Menge der Datensätze ist so groß, dass sie nur auf Festplatte abgespeichert werden können.
- Knotengröße m wird so gewählt, dass mit einem Festplattenzugriff die gesamten Daten eines Knotens (d.h. Schlüssel und Zeiger; Indexseite) gelesen werden können.
- Obere Indexseiten können im Hauptspeicher gehalten werden.

Beispiel: m = 1024 und
$n = 10^8$ ergeben Höhe $h \approx 3$



Variante: B^+-Baum

- Die Datensätze werden nur in den Blättern abgespeichert.
- Um die Bereichssuche zu unterstützen (finde alle Datensätze mit Schlüssel $k \in [k1, k2]$), werden die Blätter miteinander verkettet.
- In den Nicht-Blatt-Knoten stehen nur Schlüssel und Zeiger zu Navigationszwecken. Der i-te Schlüssel k_i befindet sich noch zusätzlich als kleinster Schlüssel im Teilbaum B_{i+1} (muss in einem Blatt sein!)
- Für die Anzahl der Schlüssel in den Blättern kann eine andere Minimal- und Maximalzahl gelten.



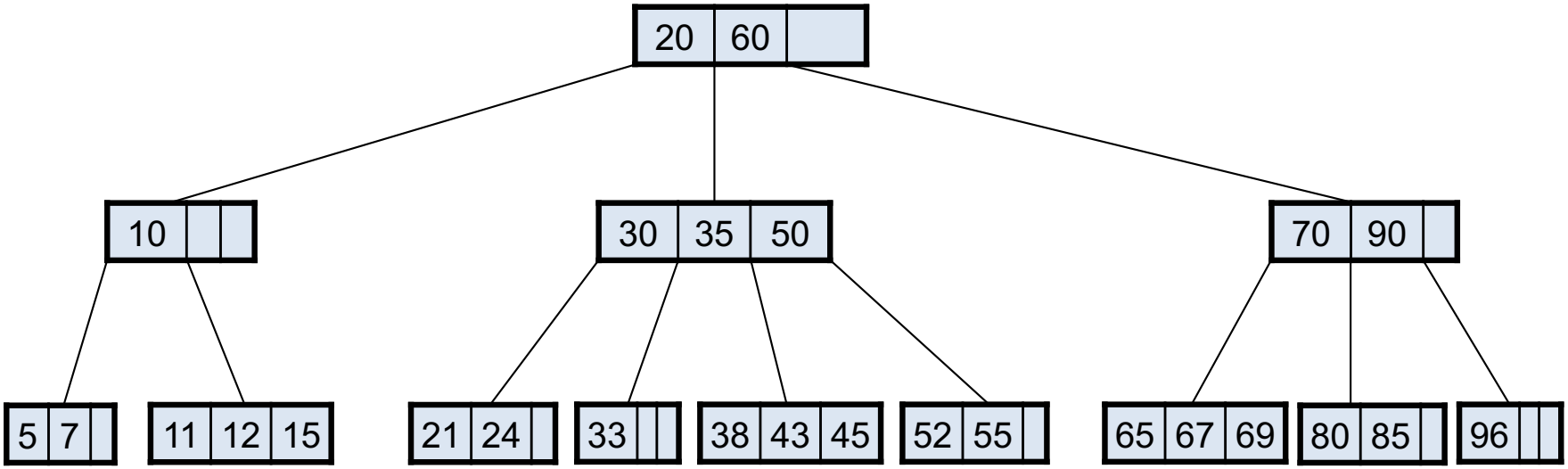
Datensätze mit Schlüssel und Nutzdaten

4. Balancierte Suchbäume

- AVL-Bäume
- B-Bäume
- 2-3-4-Bäume und Rot-Schwarzbäume

2-3-4-Bäume

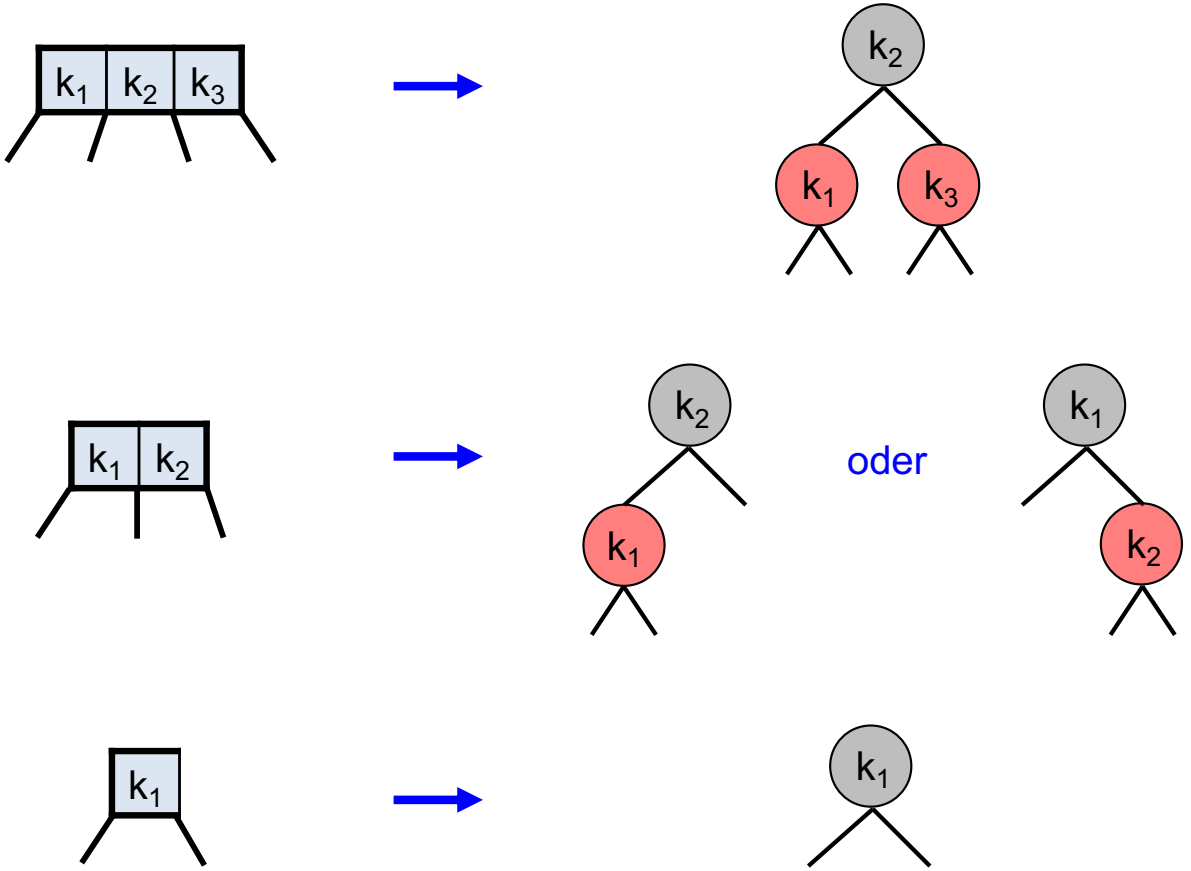
- B-Bäume der Ordnung 4 nennt man auch 2-3-4-Bäume.
  Ein Nicht-Blatt-Knoten hat 2, 3 oder 4 Kinder.



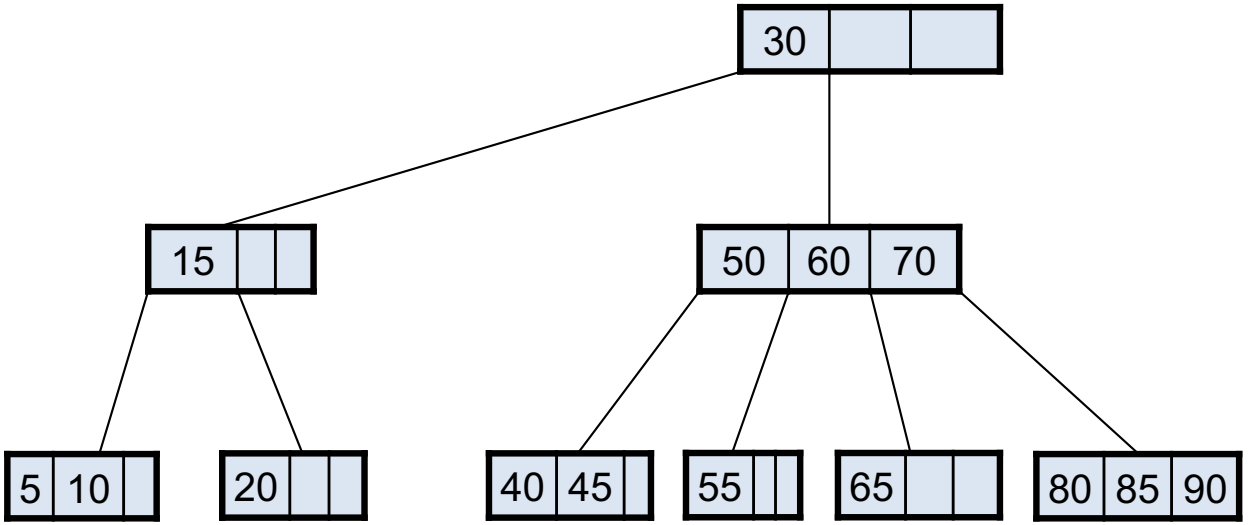
- 2-3-4-Bäume sind eine sehr gut geeignete Alternative zu AVL-Bäumen.
- Jedoch sind die B-Baum-Operationen wie split, Übernahme von Schlüssel und Verschmelzung bei m = 4 verhältnismäßig kompliziert.
- Wir werden 2-3-4-Bäumen als binäre Suchbäume mit rot bzw. schwarz gefärbten Knoten (Rot-Schwarz-Bäume) implementieren.
  Einsatz: Java-Collections und STL-Bibliothek von C++.

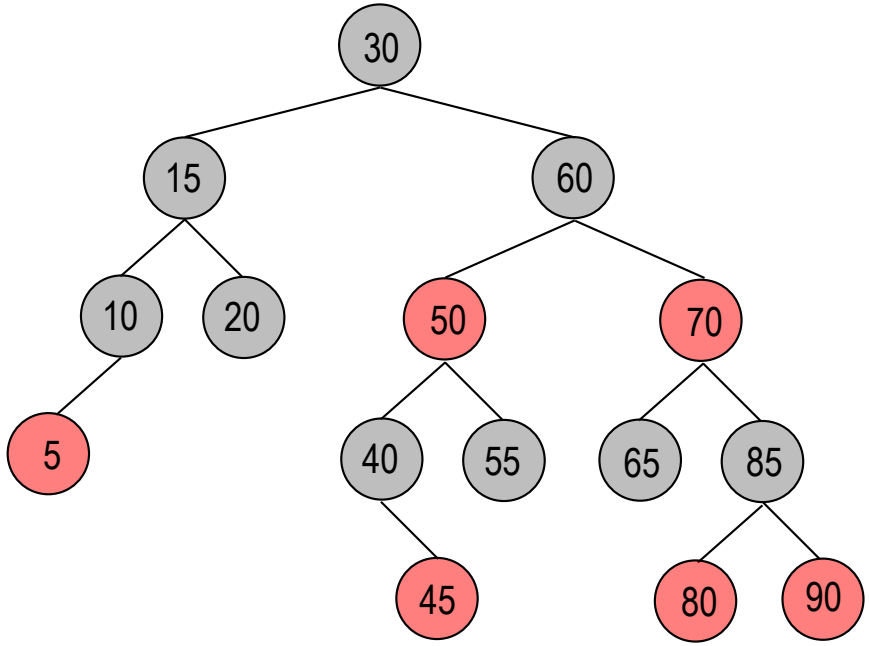
Rot-Schwarz-Bäume

- Rot-Schwarz-Bäume sind binäre Suchbäume, dessen Knoten entweder rot oder schwarz sind und dabei bestimmte Färbungsregeln (später) erfüllen.
- Ein 2-3-4-Baum kann in ein Rot-Schwarz-Baum transformiert werden, indem auf jeden Knoten des 2-3-4-Baums eine der folgenden Regeln angewendet wird:



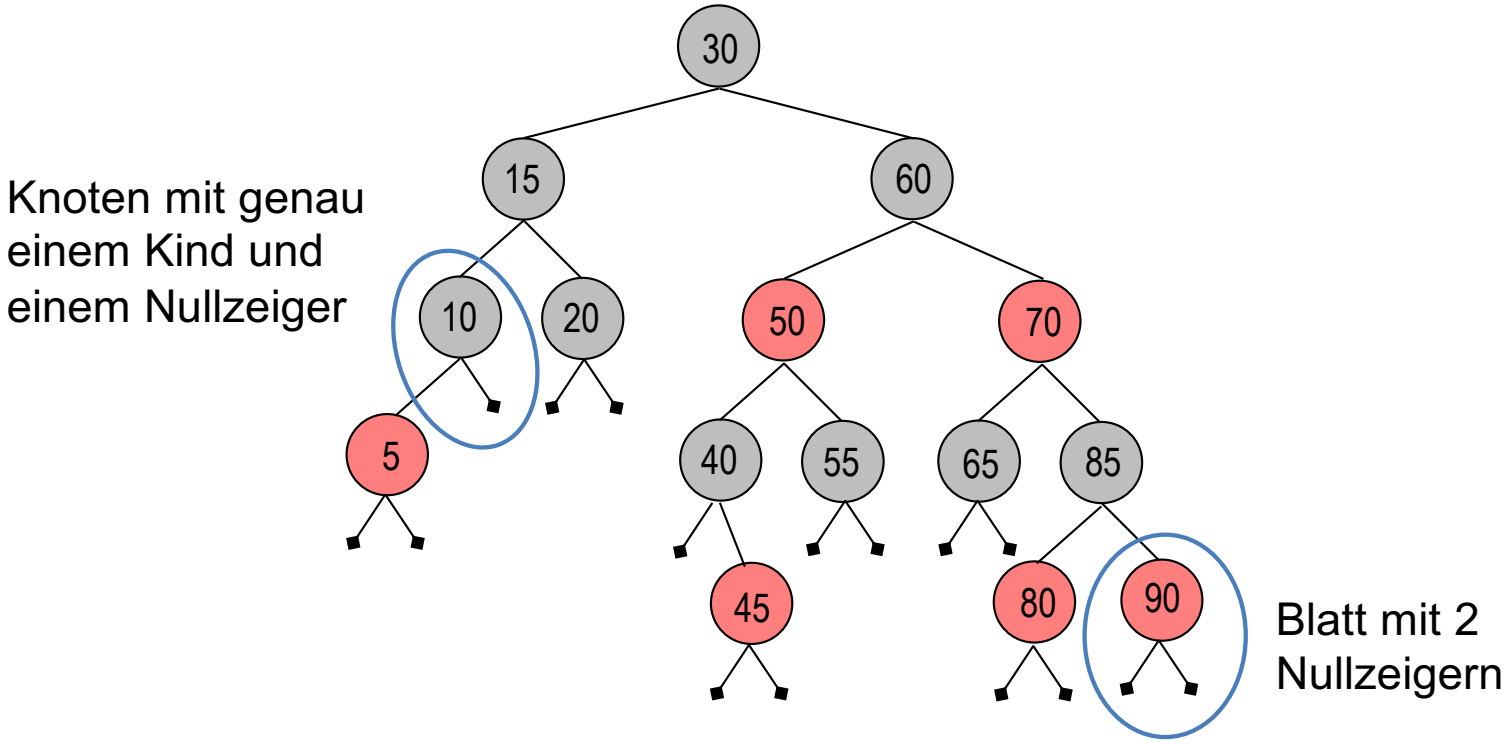
Beispiel


30
15
50 60 70
5 10
20
40 45
55
65
80 85 90


30
15
60
10
20
50
70
5
40
55
65
85
45
80
90

Rot-Schwarz-Baum mit Nullzeigern

- Für die Beschreibung der Färbungsregeln für Rot-Schwarz-Bäume werden die Null-Zeiger (Zeiger auf Kindknoten hat den Wert null) explizit dargestellt:



Färbungsregeln von Rot-Schwarz-Bäumen

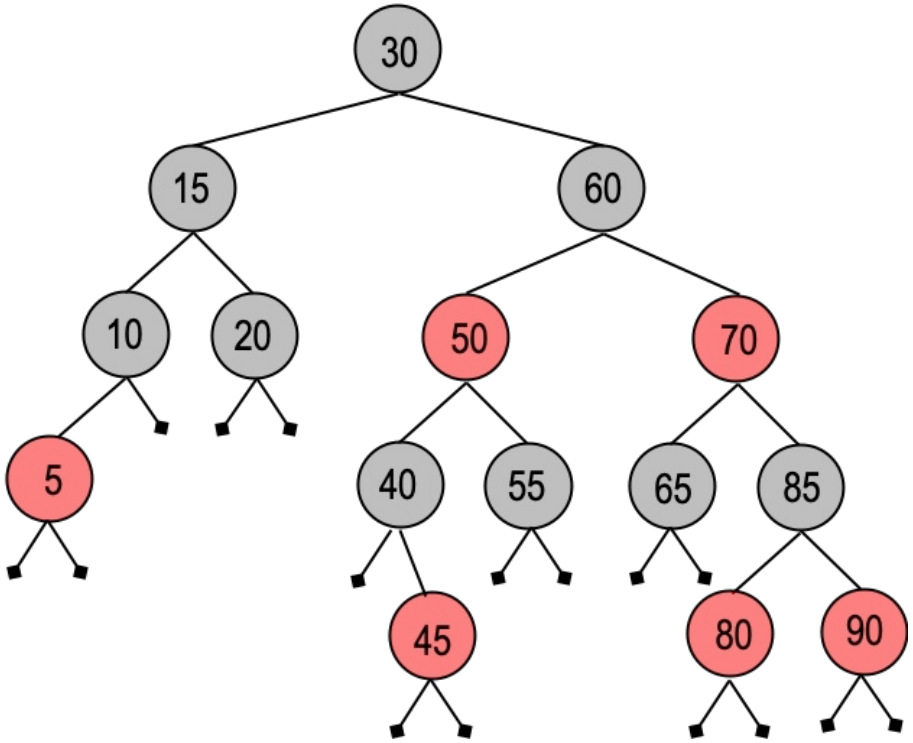
Ein Rot-Schwarz-Baum ist ein binärer Suchbaum mit folgenden Färbungsregeln:

(1) Jeder Knoten ist entweder rot oder schwarz.

(2) Die Wurzel ist immer schwarz.

(3) Ein roter Knoten darf kein rotes Kind haben.

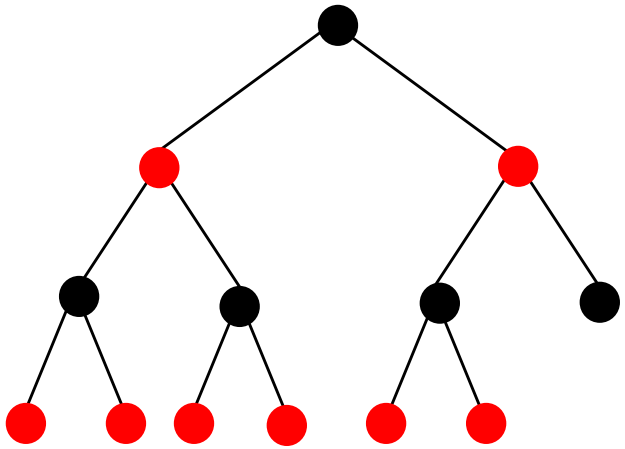
(4) Jeder Pfad von der Wurzel zu einem Null-Zeiger hat die gleiche Anzahl an schwarzen Knoten (wird auch Schwarzhöhe genannt).



Rot-Schwarz-Baum mit Schwarzhöhe 3

Maximale Rot-Schwarz-Bäume

- Vollständige Binärbaume mit passender Färbung sind maximale Rot-Schwarz-Bäume. D.h. maximale Anzahl von Knoten bei gegebener Höhe.



(Null-Zeiger sind weggelassen)

- Bei der Färbung ist zu beachten, dass die letzte Ebene im Allgemeinen nicht vollständig gefüllt ist, und daher rot gefärbt werden muss.
- Es gilt: $h = \lfloor \log_2(n) \rfloor$

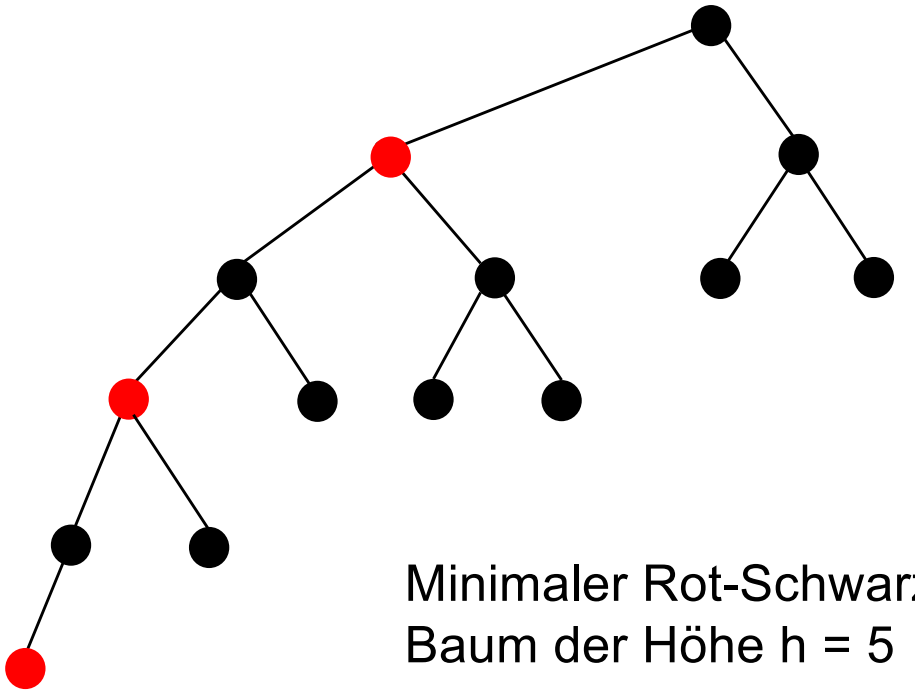
Minimale Rot-Schwarz-Bäume

- Rot-Schwarz-Bäume mit minimaler Anzahl von Knoten bei gegebener Höhe h müssen wie folgt aufgebaut sein:
  - ein Pfad der Länge h mit wechselnden Knotenfarben
  - der restliche Teil des Baums enthält nur schwarze Knoten

- Beachte: Falls eines der Blätter (außer dem roten Blatt ganz links) gelöscht wird, dann wird die Färbungsregel (4) verletzt.

- Durch Induktion lässt sich zeigen:

  $h \approx 2*\log_2(n+2) - 3$



Minimaler Rot-Schwarz-Baum der Höhe h = 5

(Null-Zeiger sind weggelassen)

Höhe eines Rot-Schwarz-Baums

- Für die Höhe h eines Rot-Schwarz-Baums mit n Knoten gilt also:

$$\lfloor \log_2(n) \rfloor \leq h \leq 2*\log_2(n+2) - 3$$

Höhe eines vollständigen Binärbaums

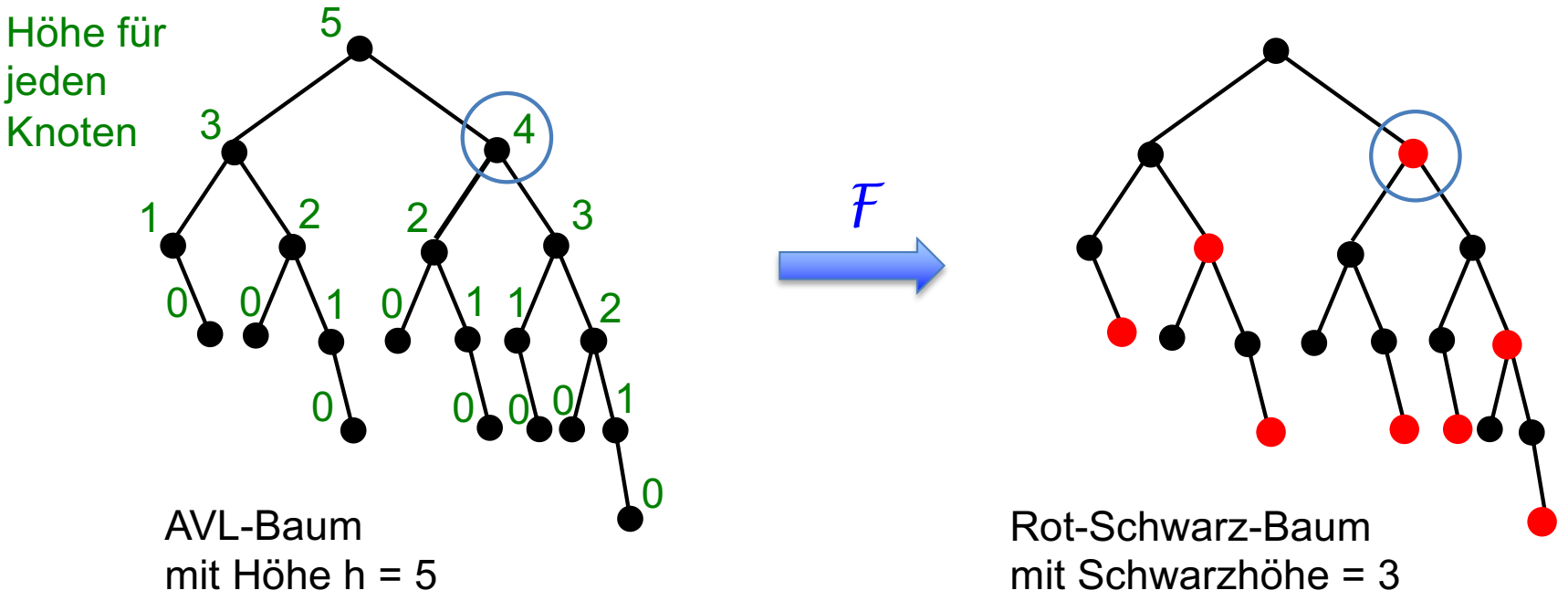
Höhe eines minimalen Rot-Schwarz-Baums

- Alle Dictionary-Operationen lassen sich daher in O(log n) realisieren.

AVL-Bäume als Rot-Schwarz-Bäume

- Färbungsregel $\mathcal{F}$:

  jeder Knoten eines AVL-Baums mit einer geraden Höhe und einem Elternknoten mit einer ungeraden Höhe wird rot gefärbt.



AVL-Baum
mit Höhe h = 5

Rot-Schwarz-Baum
mit Schwarzhöhe = 3

- Durch Induktion über die Höhe h lässt sich zeigen:

  Falls t ein AVL-Baum der Höhe h ist, dann ist $\mathcal{F}$(t) ein Rot-Schwarz-Baum mit Schwarzhöhe $\lfloor h/2 \rfloor + 1$.

Einfügen eines Schlüssels

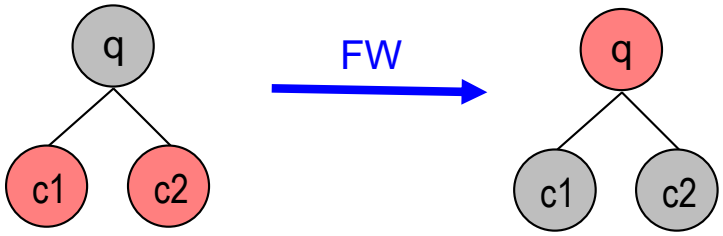
```
if (root == null)
        root = new Node(key, black);
        return; // fertig!

q = root;    // aktueller Knoten q ist die Wurzel. root != null
p = null;    // parent von q

while (q != null) {
        evtl. Farbwechsel (FW);
        evtl. eine der Rotationsoperationen R, LR, L, RL;
        p = q;
        if (key < q.key)
                q = q.left;
        else if (key > q.key)
                q = q.right;
        else // Schlüssel bereits vorhanden;
                return;
}

if (key < p.key)
        p.left = new Node(key, red);
else
        p.right = new Node(key, red);
evtl. eine der Rotationsoperationen R, LR, L, RL;
```

- durchlaufe Rot-Schwarz-Baum von seiner Wurzel bis zu dem Blatt, unter dem der neue Knoten mit Schlüssel key eingefügt werden soll;
- dabei wird mit Farbwechsel rote Farbe nach oben verschoben.
- ein Verstoß gegen Regel (3) (keine zwei rote Knoten übereinander) wird durch Rotationsoperationen vermieden.

- da der neu eingefügte Knoten rot ist, ist die Einhaltung der Regel (4) gewährleistet (gleiche Anzahl schwarzer Knoten auf jedem Pfad).

Farbwechsel

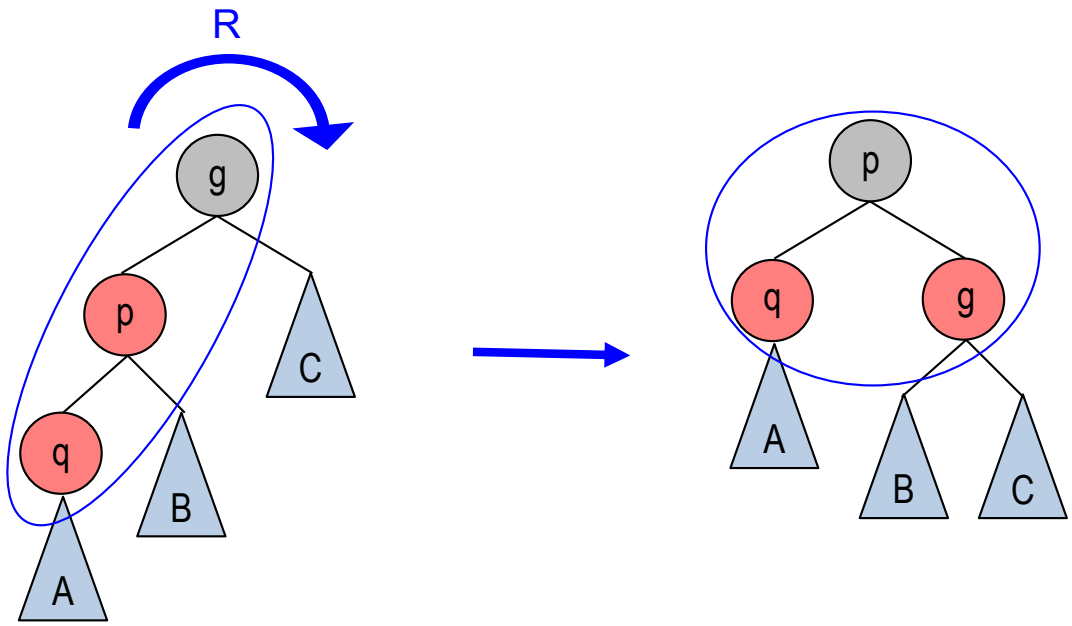
- Führe bei Knoten q, der zwei rote Kinder hat, einen Farbwechsel FW durch.



- Falls q die Wurzel ist, dann bleibt q schwarz.

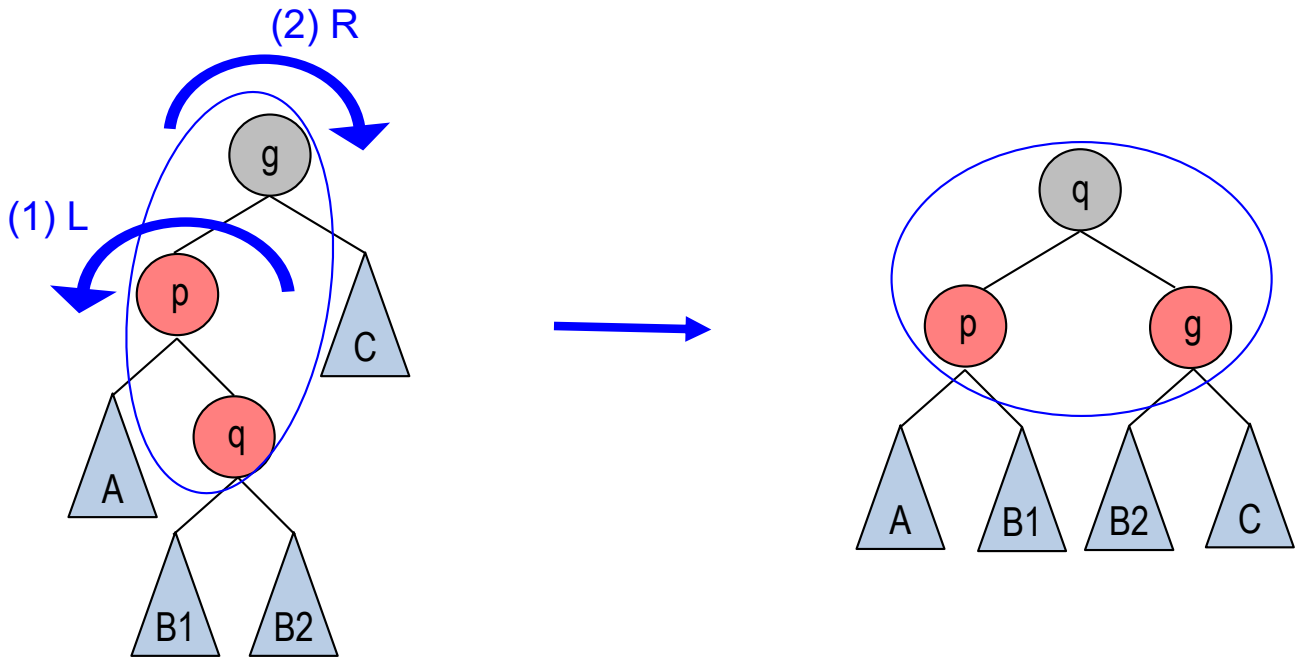
- Falls durch Farbwechsel (FW) zwei aufeinanderfolgende rote Knoten entstehen (d.h. Elternknoten von q ist rot), dann muss danach eine der Rotationsoperationen R, LR, L oder RL durchgeführt.

Rotationsoperation R



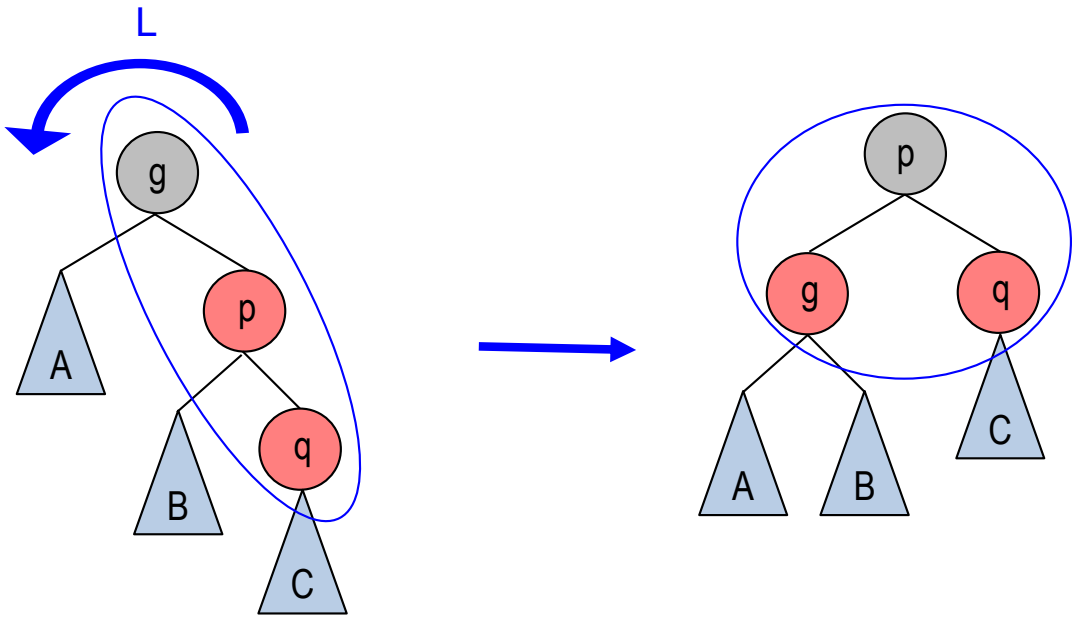
- Rotation wird um Knoten g nach rechts durchgeführt.
  p (parent) und g (grandparent) werden dabei umgefärbt.
- Wurzel von C (falls nicht leer) muss schwarz sein.
  (Begründung in [Weiss 2011])
  Daher keine weitere Verletzung der Farbregel.

Rotationsoperation LR



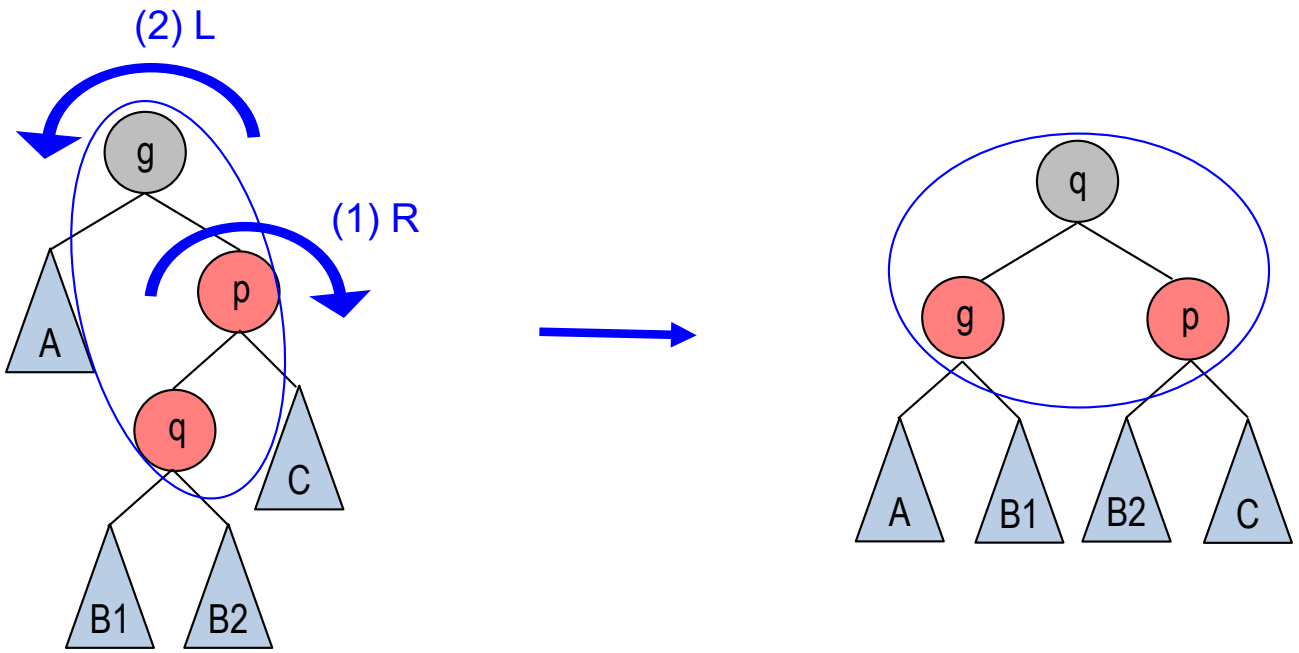
- Rotation wird um Knoten p nach links und dann um g nach rechts durchgeführt. q und g werden dabei umgefärbt.
- Wurzel von C (falls nicht leer) muss schwarz sein.
  (Begründung in [Weiss 2011])
  Daher keine weitere Verletzung der Farbregel.

Rotationsoperation L



- Rotation wird um Knoten g nach links durchgeführt.
  p und g werden dabei umgefärbt.
- Wurzel von A (falls nicht leer) muss schwarz sein.
  (Begründung in [Weiss 2011])
  Daher keine weitere Verletzung der Farbregel.

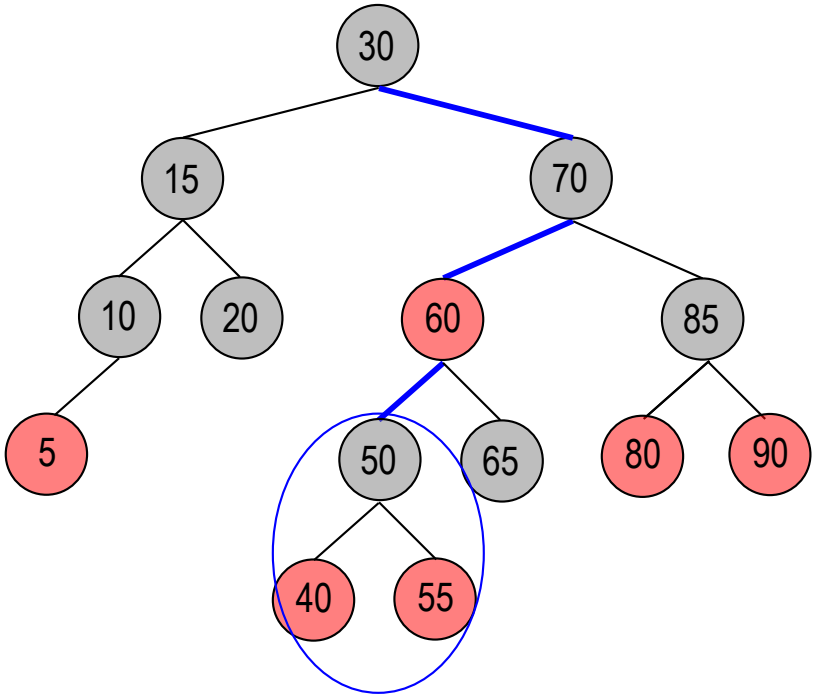
Rotationsoperation RL

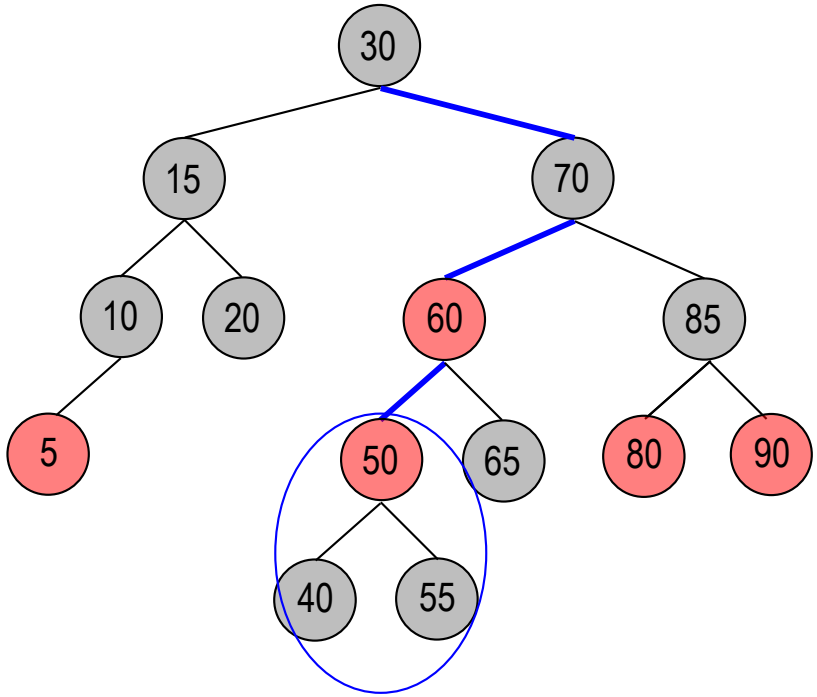


- Rotation wird um Knoten p nach rechts und dann um g nach links durchgeführt. q und g werden dabei umgefärbt.
- Wurzel von A (falls nicht leer) muss schwarz sein.
  (Begründung in [Weiss 2011])
  Daher keine weitere Verletzung der Farbregel.

Beispiel (1)

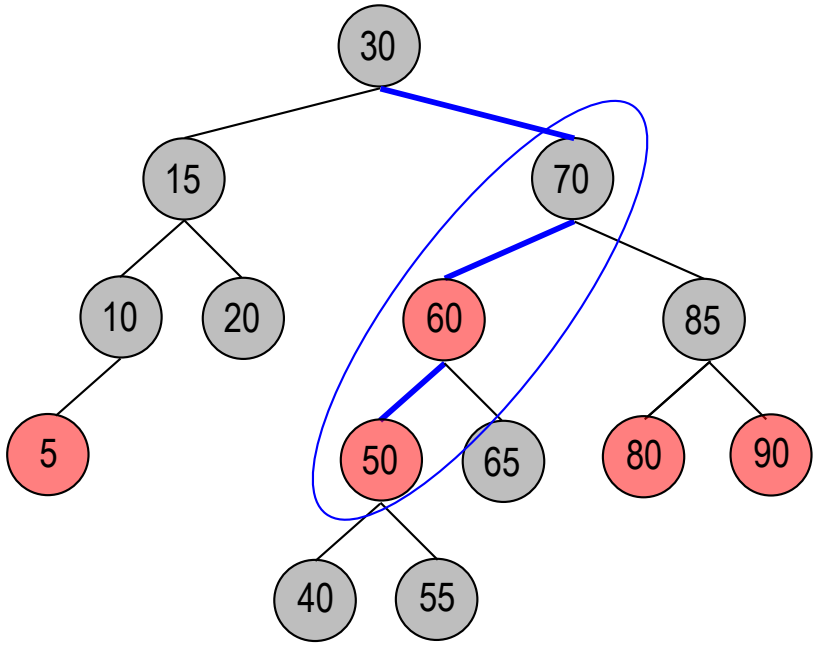
- Einfügen von 45.
- Der Baum wird bei der Wurzel 30 beginnend top down durchlaufen:
  30, 70, 60, 50.
- Bei 50 wird ein Farbwechsel FW durchgeführt.

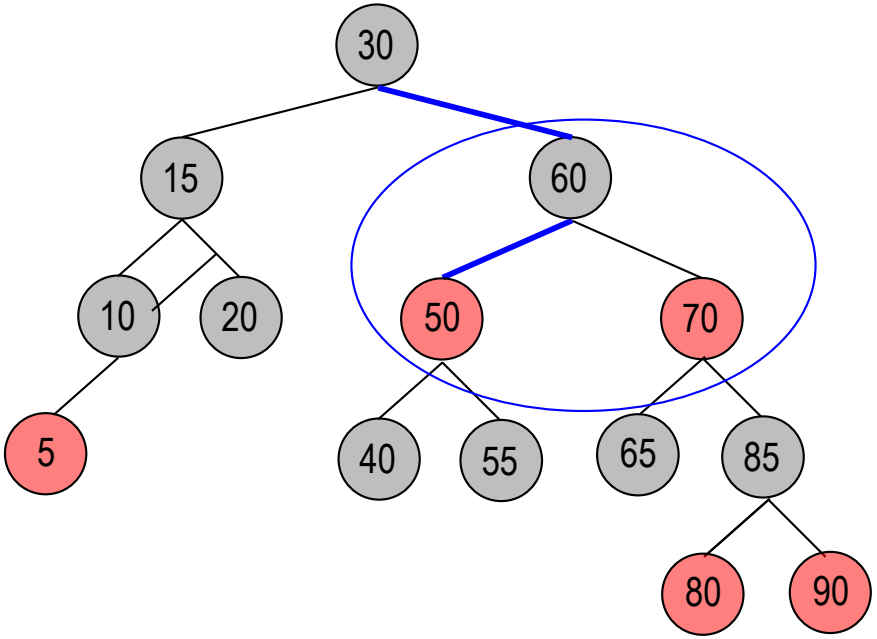




Beispiel (2)

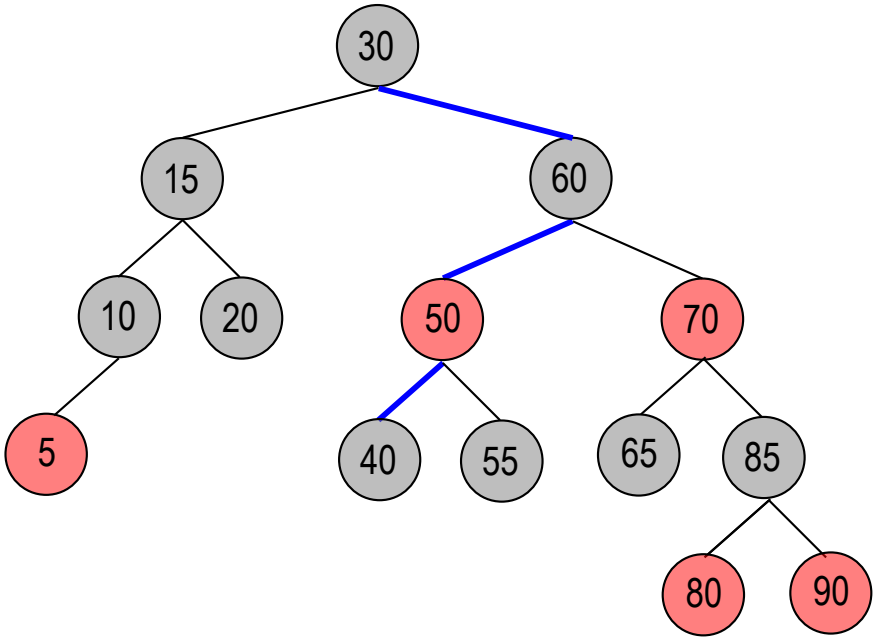
- Durch den Farbwechsel wird eine Rechtsrotation R notwendig

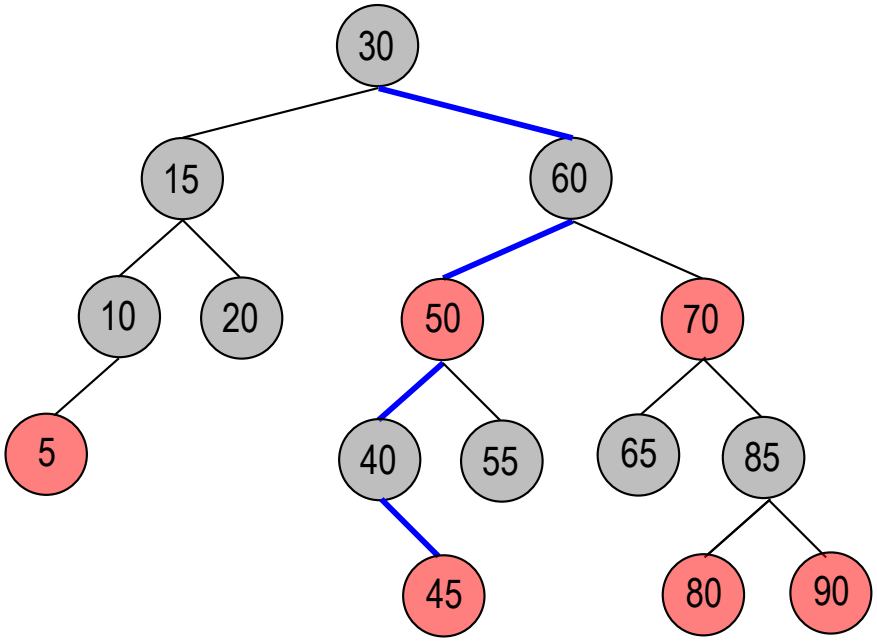




Beispiel (3)

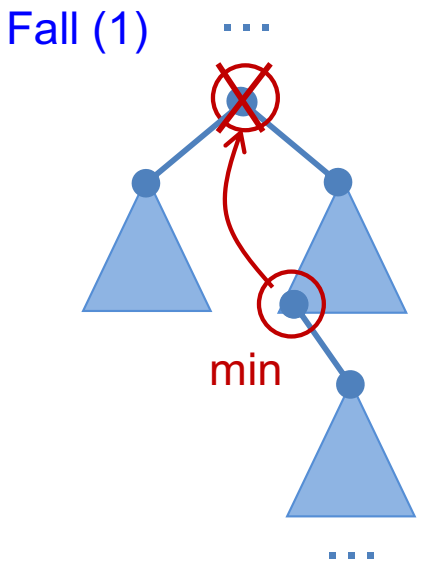
- Der Durchlauf kann bei Knoten 50 fortgesetzt werden. Dadurch wird als nächstes Knoten 40 besucht.
- Der neue Knoten 45 kann rechts von 40 als roter Knoten eingefügt werden.
- Fertig!

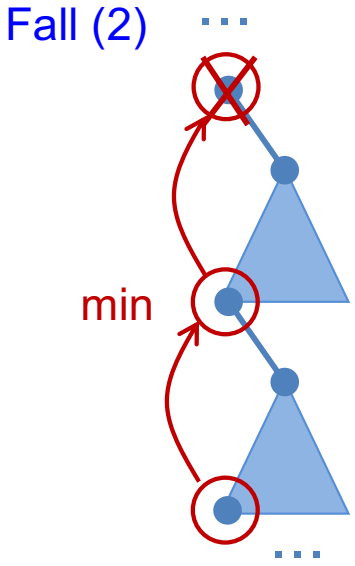


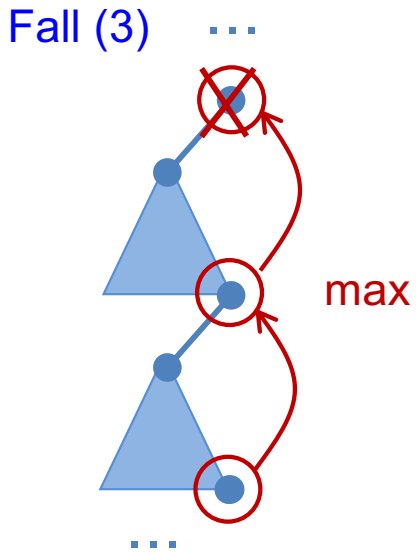


Löschen eines Schlüssels (1)

- Idee: Gelöscht wird immer ein rotes Blatt.
- Suche den zu löschenden Knoten durch iterativen top-down-Durchlauf.
  Behandele 3 unterschiedliche Fälle:
  - (1) Hat der zu löschende Knoten zwei Kinder, dann wird er durch den kleinsten Knoten min im rechten Teilbaum ersetzt. min wird als nächstes gelöscht (Fall (2))
  - (2) Hat der zu löschende Knoten genau ein rechtes Kind, dann wird er durch den kleinsten Knoten min im rechten Teilbaum ersetzt.
    min wird als nächstes gelöscht.
  - (3) Hat der zu löschende Knoten genau ein linkes Kind, dann wird er durch den größten Knoten max im linken Teilbaum ersetzt.
    max wird als nächstes gelöscht.

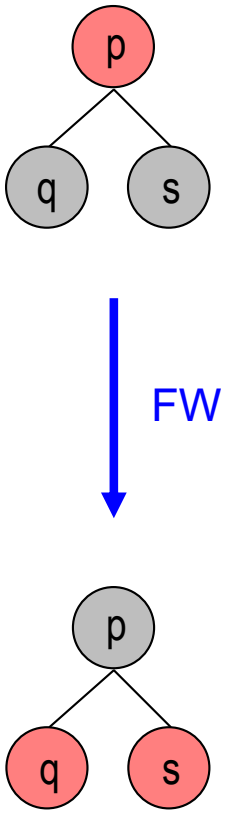






Löschen eines Schlüssels (2)

- Das zu löschende Blatt muss rot sein!
  Dann kann das Blatt einfach ausgehängt werden.
- Dazu wird beim iterativen top-Down-Durchlauf gewährleistet, dass der aktuelle Knoten q (außer der Wurzel) immer rot gefärbt wird.
- Dazu werden Farbwechsel (FW) und Rotationsoperationen durchgeführt.
- Rotationsoperationen:
  2 * 5 Fälle.
  Zusätzlich 6 separate Fälle bei der Wurzel.
- Näheres siehe [Weiss 2011].



Laufzeitmessungen

Daten-struktur	$N = 10^6$					$N = 10^7$				
	N*insert	N*search	min	d_{av}	h	N*insert	N*search	min	d_{av}	h
Rot-Schwarz-Baum	0.68 sec	0.71 sec	15.0	18.4	24.4	10.66 sec	9.05 sec	18.0	21.8	28.8
AVL-Baum	0.81 sec	0.68 sec	14.5	18.3	23.0	12.93 sec	7.97 sec	17.0	21.7	27.0
Java TreeMap	0.66 sec	0.74 sec	n.a.	n.a.	n.a.	10.34 sec	8.98 sec	n.a.	n.a.	n.a.

- Zeiten sind über 10 Versuche gemittelt.
- N*insert: es wurden N Zufallszahlen in einen leeren Baum eingefügt.
- N*search: es wurden N Zufallszahlen in einem bereits mit N Zahlen gefüllten Baum gesucht (mit großer Wahrscheinlichkeit nicht erfolgreiche Suche!).
- min: Länge eines kürzesten Pfads von der Wurzel zu einem Blatt
- d_{av}: durchschnittliche Tiefe aller Knoten im Baum
- h: Höhe des Baums.