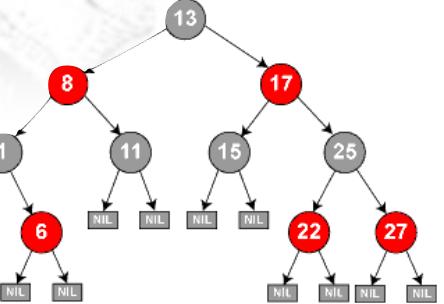
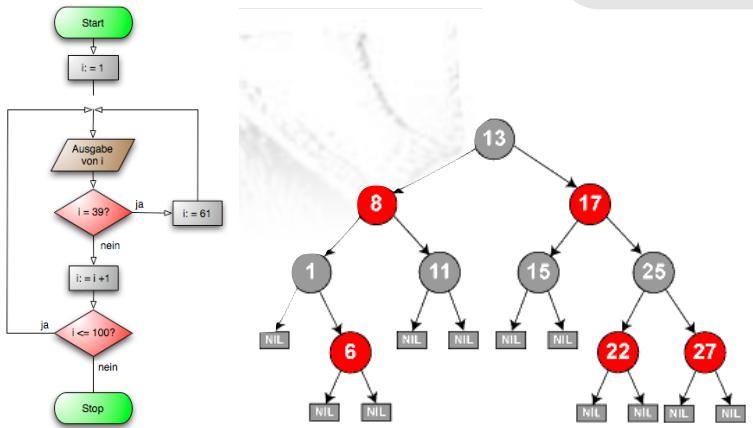




# Algorithmen und Datenstrukturen



Version 1.27 vom 09. Mai 2023

# V. Suchen in Mengen

---

## 5. Suchen in Mengen

- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume
  - 5.5.1. Gewichtsbalanceierte Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume
- 5.6. *Priority Queue* und *Heap*

# V. Suchen in Mengen

## 5. Suchen in Mengen

### 5.1. Einführung

- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume
  - 5.5.1. Gewichtsbilanzierte Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume
- 5.6. *Priority Queue* und *Heap*

# Einführung

- **Gegeben:** eine Menge von Records, die eindeutig durch einen Schlüssel identifiziert werden können

# Einführung

- **Gegeben:** eine Menge von Records, die eindeutig durch einen Schlüssel identifiziert werden können
- **Aufgabe:** finde zu gegebenem Schlüssel den passenden Record

# Einführung

- **Gegeben:** eine Menge von Records, die eindeutig durch einen Schlüssel identifiziert werden können
- **Aufgabe:** finde zu gegebenem Schlüssel den passenden Record
- **Notation:** Universum  $U :=$  Menge aller möglichen Schlüssel
  - **Compilerbau:** Bezeichner in Symboltabelle nur 6 Zeichen (Buchstaben und Zahlen); dann  $|U| = (26+10)^6$
  - **Konten einer Bank:** 6-stellige Kontonummer  $|U| = 10^6$

# Einführung

- **Gegeben:** eine Menge von Records, die eindeutig durch einen Schlüssel identifiziert werden können
- **Aufgabe:** finde zu gegebenem Schlüssel den passenden Record
- **Notation:** Universum  $U :=$  Menge aller möglichen Schlüssel
  - **Compilerbau:** Bezeichner in Symboltabelle nur 6 Zeichen (Buchstaben und Zahlen); dann  $|U| = (26+10)^6$
  - **Konten einer Bank:** 6-stellige Kontonummer  $|U| = 10^6$
- **Spezifikation des Suchproblems**
  - Gegeben: endliche Schlüsselmenge  $S \subseteq U$  mit  $|S|=n$  und Schlüssel  $A \in U$  ( $S$  meist in Form eines Feldes)
  - **Aufgabe:** Gibt es ein  $i$  mit  $S[i]=A$ ?

# Einführung

- **Gegeben:** eine Menge von Records, die eindeutig durch einen Schlüssel identifiziert werden können
- **Aufgabe:** finde zu gegebenem Schlüssel den passenden Record
- **Notation:** Universum  $U :=$  Menge aller möglichen Schlüssel
  - **Compilerbau:** Bezeichner in Symboltabelle nur 6 Zeichen (Buchstaben und Zahlen); dann  $|U| = (26+10)^6$
  - **Konten einer Bank:** 6-stellige Kontonummer  $|U| = 10^6$
- **Spezifikation des Suchproblems**
  - Gegeben: endliche Schlüsselmenge  $S \subseteq U$  mit  $|S| = n$  und Schlüssel  $A \in U$  ( $S$  meist in Form eines Feldes)
  - **Aufgabe:** Gibt es ein  $i$  mit  $S[i] = A$ ?
- **Allgemeine Beobachtung**
  - Erfolgsuche braucht maximal  $n$  Vergleiche

# V. Suchen in Mengen

## 5. Suchen in Mengen

5.1. Einführung

### 5.2. Einfache Implementierungen

5.3. *Hashing*

5.4. Binäre Suchbäume

5.5. Balancierte Bäume

    5.5.1. Gewichtsbilanzierte Bäume

    5.5.2. AVL-Bäume

    5.5.3. (a,b)-Bäume

    5.5.4. rot/schwarz-Bäume

5.6. *Priority Queue* und *Heap*

# Sequentielle (lineare) Suche

- Durchsucht das Feld sequentiell
- Das Feld darf unsortiert sein
- Implementierung (JAVA):

```
// Wenn nicht gefunden, zeigt der Index hinter  
// das letzte Element (=field.length)  
public static int search(int a,int field[]) {  
    for (int i=0 ; i<field.length ; ++i)  
        if (field[i]==a) return i;  
    return field.length;  
}
```

- Komplexität (Zahl der Vergleiche)
  - **worst case:**  $n$  Vergleiche
  - **average case:**  $(n/2)$  Vergleiche
  - **best case:** 1 Vergleich

# Binäre Suche

## B Binäre Suche

- Sortierte Folge mit 9 Elementen
- gesucht: 6

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

# Binäre Suche

## B Binäre Suche

- Sortierte Folge mit 9 Elementen
- gesucht: 6
- Betrachte Element an Position  $0 + \lceil (8-0)/2 \rceil = 4$



# Binäre Suche

## B Binäre Suche

- Sortierte Folge mit 9 Elementen
- gesucht: 6
- Betrachte Element an Position  $0 + \lceil (8-0)/2 \rceil = 4$
- Da  $6 > 5$  kann gesuchter Index nur im Intervall [5,8] sein
- Betrachte Element an Position  $5 + \lceil (8-5)/2 \rceil = 7$



# Binäre Suche

## B Binäre Suche

- Sortierte Folge mit 9 Elementen
- gesucht: 6
- Betrachte Element an Position  $0 + \lceil (8-0)/2 \rceil = 4$
- Da  $6 > 5$  kann gesuchter Index nur im Intervall [5,8] sein
- Betrachte Element an Position  $5 + \lceil (8-5)/2 \rceil = 7$
- Da  $6 < 8$  kann gesuchter Index nur im Intervall [5,6] sein
- Betrachte Element an Position  $5 + \lceil (6-5)/2 \rceil = 6$



# Binäre Suche

## B Binäre Suche

- Sortierte Folge mit 9 Elementen
- gesucht: 6
- Betrachte Element an Position  $0 + \lceil (8-0)/2 \rceil = 4$
- Da  $6 > 5$  kann gesuchter Index nur im Intervall [5,8] sein
- Betrachte Element an Position  $5 + \lceil (8-5)/2 \rceil = 7$
- Da  $6 < 8$  kann gesuchter Index nur im Intervall [5,6] sein
- Betrachte Element an Position  $5 + \lceil (6-5)/2 \rceil = 6$
- Da  $6 < 7$  kann gesuchter Index nur im Intervall [5,5] sein - **gefunden!**



# Binäre Suche

- **Voraussetzung:** Feld ist sortiert!

# Binäre Suche

- **Voraussetzung:** Feld ist sortiert!

# Binäre Suche

- **Voraussetzung:** Feld ist sortiert!
- **Idee:** Suchbereich sukzessive halbieren - Suche in richtiger Hälfte fortsetzen

# Binäre Suche

- **Voraussetzung:** Feld ist sortiert!
- **Idee:** Suchbereich sukzessive halbieren - Suche in richtiger Hälfte fortsetzen
- **Implementierung (JAVA):**

```
public static int binSearch(int a, int field []) {  
    int left=0, right=field.length-1;  
    while (left<=right) {  
        int middle = (left+right)/2;  
        if (a==field[middle]) return middle;  
        else if (a<field[middle]) right=middle-1; // im linken Teilfeld  
        else /* a>field[middle] */ left=middle+1; // im rechten Teilfeld  
    }  
    return field.length;  
}
```

# Binäre Suche: Komplexitätsanalyse

# Binäre Suche: Komplexitätsanalyse

- **Komplexität (Zahl der Vergleiche)**

**Annahme:** Feld enthält  $n=2^k-1$  Elemente

# Binäre Suche: Komplexitätsanalyse

- **Komplexität (Zahl der Vergleiche)**

**Annahme:** Feld enthält  $n=2^k-1$  Elemente

- **best case:** 1 Vergleich - Element wird sofort gefunden

# Binäre Suche: Komplexitätsanalyse

- **Komplexität (Zahl der Vergleiche)**

**Annahme:** Feld enthält  $n=2^k-1$  Elemente

- **best case:** 1 Vergleich - Element wird sofort gefunden
- **worst case:** Element wird nach dem letzten Teiltvorgang gefunden  
(es ist auf dem untersten Level des Suchbaumes)

$$T(n) = T\left(\frac{n-1}{2}\right) + 1 = \text{ld } (n+1)$$

# Binäre Suche: Komplexitätsanalyse

## • Komplexität (Zahl der Vergleiche)

Annahme: Feld enthält  $n=2^k-1$  Elemente

- **best case:** 1 Vergleich - Element wird sofort gefunden
- **worst case:** Element wird nach dem letzten Teiltvorgang gefunden  
(es ist auf dem untersten Level des Suchbaumes)

$$T(n) = T\left(\frac{n-1}{2}\right) + 1 = \lg(n+1)$$

- **average case:** Wir berechnen den Mittelwert der Suchkosten, die entstehen, wenn wir nach jedem Element suchen:

Auf Level  $i$  sind  $2^i$  Knoten. Um Element auf Level  $i$  zu finden, benötigen wir  $(i+1)$  Vergleiche; im Mittel also:

$$\begin{aligned} T(n) &= \frac{1}{2^k - 1} \cdot \sum_{i=0}^{k-1} (i+1) \cdot 2^i \\ &= \frac{1}{2^k - 1} \cdot ((k-1) \cdot 2^k + 1) \\ &= (k-1) + \frac{k-1}{2^k - 1} + \frac{1}{2^k - 1} \\ &= \lg(n+1) - 1 + \frac{\lg(n+1) - 1}{n} + \frac{1}{n} \\ &\leq \lg(n+1) \end{aligned}$$

# Binäre Suche: Komplexitätsanalyse - ausführlich 1

## • **worst case** Ausführliche Herleitung:

Wir setzen zunächst sukzessive ein.  $T_i(n)$  bezeichne die Gleichung, die wir nach em  $i$ -ten Einsetzen erhalten haben:

$$T_0(n) = T\left(\frac{n-1}{2}\right) + 1$$

$$T_1(n) = T\left(\frac{\frac{n-1}{2}-1}{2}\right) + 1 + 1 = T\left(\frac{n-1}{4} - \frac{1}{2}\right) + 2$$

$$T_2(n) = T\left(\frac{\frac{n-1}{4} - \frac{1}{2} - 1}{2}\right) + 1 + 1 + 1 = T\left(\frac{n-1}{8} - \frac{1}{4} - \frac{1}{2}\right) + 3$$

⋮

$$T_i(n) = T\left(\frac{n-1}{2^{i+1}} - \sum_{j=0}^{i-1} \frac{1}{2^{j+1}}\right) + (i+1)$$

Wir wissen, dass  $T(0) = 0$  gilt. Die Frage ist also, nach der wievielten Iteration das Argument an  $T$  kleiner oder gleich 0 wird.

$$\frac{n-1}{2^{i+1}} - \sum_{j=0}^{i-1} \frac{1}{2^{j+1}} = 0$$

Wir müssen diese Gleichung nach  $i$  auflösen.

# Binäre Suche: Komplexitätsanalyse - ausführlich 2

- **worst case** Ausführliche Herleitung:

$$\frac{n-1}{2^{i+1}} - \sum_{j=0}^{i-1} \frac{1}{2^{j+1}} = \frac{n-1}{2^{i+1}} - \left( \frac{1}{2^1} + \cdots + \frac{1}{2^i} \right) = 0$$

$$\frac{n-1}{2^{i+1}} - \sum_{j=0}^{i-1} \frac{1}{2^{i-j}} = 0$$

$$\frac{n-1}{2^{i+1}} - \sum_{j=0}^{i-1} 2^{-i+j} = 0$$

$$\frac{n-1}{2^{i+1}} - \frac{1}{2^i} \cdot \sum_{j=0}^{i-1} 2^j = 0$$

$$\frac{n-1}{2^{i+1}} - \frac{1}{2^i} \cdot (2^i - 1) = 0$$

$$\frac{n-1}{2^{i+1}} - 1 + \frac{1}{2^i} = 0$$

$$\frac{n-1}{2^{i+1}} + \frac{2}{2^{i+1}} = 1$$

$$n+1 = 2^{i+1}$$

$$\text{ld}(n+1) - 1 = i$$

Nach dem  $i$ -ten Einsetzen haben wir:

$$\begin{aligned} T_i(n) &= T \left( \frac{n-1}{2^{i+1}} - \sum_{j=0}^{i-2} \frac{1}{2^{j+1}} \right) + (i+1) \\ &= T(0) + (i+1) \end{aligned}$$

und mit  $T(0) = 0$  und  $i = \text{ld}(n+1) - 1$  sofort

$$T(n) = \text{ld}(n+1)$$

# Binäre Suche: Komplexitätsanalyse - ausführlich 3

## • average case

Ausführliche Herleitung:

$$\begin{aligned} T(n) &= \frac{1}{2^k - 1} \cdot \sum_{i=0}^{k-1} (i+1) \cdot 2^i \\ &= \frac{1}{2^k - 1} \cdot \left( \sum_{i=0}^{k-1} i \cdot 2^i + \sum_{i=0}^{k-1} 2^i \right) \\ &= \frac{1}{2^k - 1} \cdot ((k-1) \cdot 2^{k+1} - k \cdot 2^k + 2 + (2^k - 1)) \\ &= \frac{1}{2^k - 1} \cdot ((k-1) \cdot 2^{k+1} - (k-1) \cdot 2^k + 1) \\ &= \frac{1}{2^k - 1} \cdot ((k-1) \cdot (2^{k+1} - 2^k) + 1) \\ &= \frac{1}{2^k - 1} \cdot ((k-1) \cdot 2^k + 1) \\ &= \frac{1}{2^k - 1} \cdot ((k-1) \cdot (2^k - 1) + (k-1) + 1) \\ &= (k-1) + \frac{k-1}{2^k - 1} + \frac{1}{2^k - 1} \\ &= \text{ld}(n+1) - 1 + \frac{\text{ld}(n+1) - 1}{n} + \frac{1}{n} \\ &\leq \text{ld}(n+1) \end{aligned}$$

- Offenbar ist das Suchen in einem sortierten Feld effizienter
- Vorbild: Suche in einem sortierten Wörterbuch
- Frage: Aufbau und Pflege des sortierten Wörterbuchs (Menge  $S \subseteq U$ )
  - Welche Datenstruktur?
  - Wie effizient lassen sich typische Operationen realisieren?
  - Typische Operationen sind
    - Einfügen  $insert(a, S)$  und Löschen  $delete(a, S)$
    - Suchen  $search(a, S)$
    - Vereinigung, Durchschnitt und Differenz zweier Wörterbücher

# Bitvektor-Darstellung für Mengen

- Geeignet für kleine Universen  $U$

# Bitvektor-Darstellung für Mengen

- **Geeignet für kleine Universen  $U$**
- **Suche hier nur als Test auf Enthaltensein**  
→ keine Index-Ermittlung

# Bitvektor-Darstellung für Mengen

- **Geeignet für kleine Universen  $U$**
- **Suche hier nur als Test auf Enthaltensein**
  - keine Index-Ermittlung
- **Voraussetzung:** Abbildung der Schlüssel in die natürlichen Zahlen
  - Sei  $|U|=N$ , dann ist  $\varphi: U \rightarrow \{0, \dots, N-1\}$  eine Kodierungsfunktion (wie bei Bucket Sort)

# Bitvektor-Darstellung für Mengen

- **Geeignet für kleine Universen  $U$**
- **Suche hier nur als Test auf Enthaltensein**
  - keine Index-Ermittlung
- **Voraussetzung:** Abbildung der Schlüssel in die natürlichen Zahlen
  - Sei  $|U|=N$ , dann ist  $\varphi: U \rightarrow \{0, \dots, N-1\}$  eine Kodierungsfunktion (wie bei Bucket Sort)
- **Idee:** Verwende kodierten Schlüssel  $\varphi(a)$  als Index in **Bitvektor**:
  - $\text{Bit}[\varphi(a)]=0 \Rightarrow a \notin S$
  - $\text{Bit}[\varphi(a)]=1 \Rightarrow a \in S$

# Bitvektor-Darstellung für Mengen

- Geeignet für kleine Universen  $U$
- Suche hier nur als Test auf Enthaltensein  
→ keine Index-Ermittlung
- Voraussetzung: Abbildung der Schlüssel in die natürlichen Zahlen
  - Sei  $|U|=N$ , dann ist  $\varphi: U \rightarrow \{0, \dots, N-1\}$  eine Kodierungsfunktion (wie bei Bucket Sort)
- Idee: Verwende kodierten Schlüssel  $\varphi(a)$  als Index in Bitvektor:
  - $\text{Bit}[\varphi(a)]=0 \Rightarrow a \notin S$
  - $\text{Bit}[\varphi(a)]=1 \Rightarrow a \in S$
- Gut: Einfügen, Löschen und Suchen mit  $O(1)$ !

# Bitvektor-Darstellung für Mengen

- Geeignet für kleine Universen  $U$
- Suche hier nur als Test auf Enthaltensein  
→ keine Index-Ermittlung
- Voraussetzung: Abbildung der Schlüssel in die natürlichen Zahlen
  - Sei  $|U|=N$ , dann ist  $\varphi: U \rightarrow \{0, \dots, N-1\}$  eine Kodierungsfunktion (wie bei Bucket Sort)
- Idee: Verwende kodierten Schlüssel  $\varphi(a)$  als Index in Bitvektor:
  - $\text{Bit}[\varphi(a)]=0 \Rightarrow a \notin S$
  - $\text{Bit}[\varphi(a)]=1 \Rightarrow a \in S$
- Gut: Einfügen, Löschen und Suchen mit  $O(1)$ !

# Bitvektor: Vermeidung der Initialisierung

- **Initialisierung des Bitvektors kann ganz vermieden werden, wenn mehr Speicherplatz zur Verfügung steht**

# Bitvektor: Vermeidung der Initialisierung

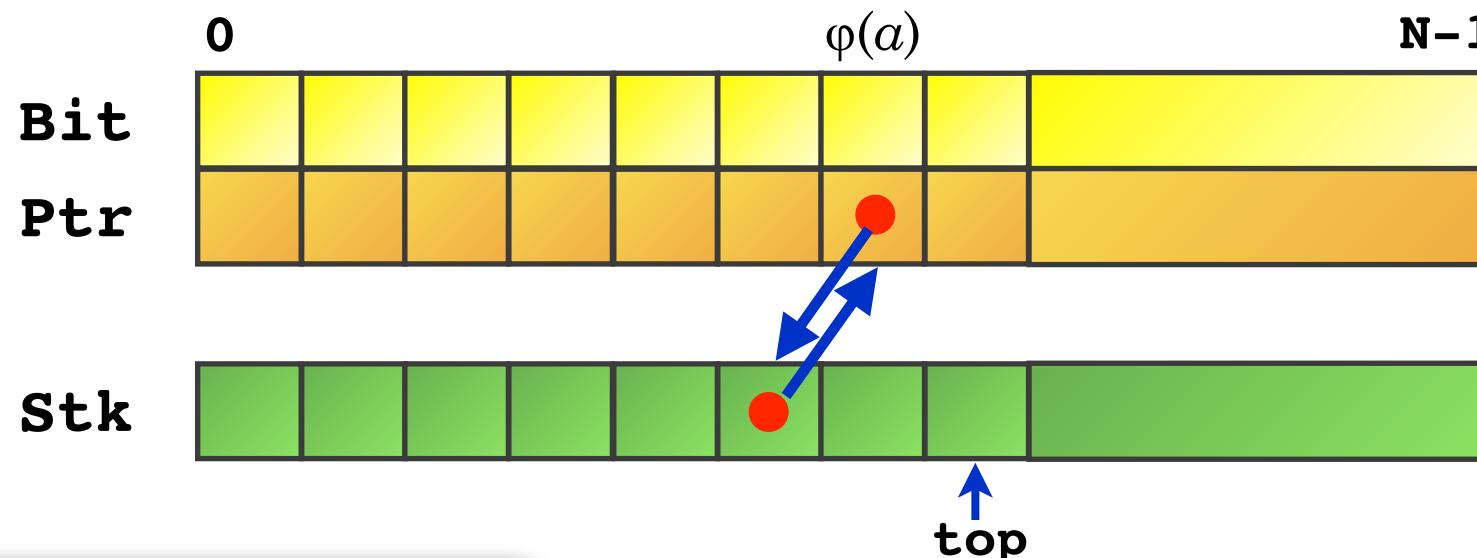
- **Initialisierung des Bitvektors kann ganz vermieden werden, wenn mehr Speicherplatz zur Verfügung steht**
- **Man benötigt einen Stack `Stk` (realisiert über ein Feld und einen top-Index) und ein Zeigerfeld `Ptr` (zur Aufnahme von Indizes)**

# Bitvektor: Vermeidung der Initialisierung

- Initialisierung des Bitvektors kann ganz vermieden werden, wenn mehr Speicherplatz zur Verfügung steht
- Man benötigt einen Stack **Stk** (realisiert über ein Feld und einen top-Index) und ein Zeigerfeld **Ptr** (zur Aufnahme von Indizes)
- Idee: Das Konzept beruht auf der Invarianten:  $a \in S \Leftrightarrow$ 
  - $\text{Bit}[\varphi(a)] = 1$  und
  - $0 \leq \text{Ptr}[\varphi(a)] \leq \text{top}$  und
  - $\text{Stk}[\text{Ptr}[\varphi(a)]] = \varphi(a)$

# Bitvektor: Vermeidung der Initialisierung

- Initialisierung des Bitvektors kann ganz vermieden werden, wenn mehr Speicherplatz zur Verfügung steht
- Man benötigt einen Stack **Stk** (realisiert über ein Feld und einen top-Index) und ein Zeigerfeld **Ptr** (zur Aufnahme von Indizes)
- Idee: Das Konzept beruht auf der Invarianten:  $a \in S \Leftrightarrow$ 
  - $\text{Bit}[\varphi(a)] = 1$  und
  - $0 \leq \text{Ptr}[\varphi(a)] \leq \text{top}$  und
  - $\text{Stk}[\text{Ptr}[\varphi(a)]] = \varphi(a)$



# Implementierung: Bitvektor-Darstellung (JAVA) I

```
public class BitVek {  
    boolean Bit[];  
    int Ptr[];  
    int Stk[];  
    int top;  
  
    public BitVek(int size) {  
        Bit = new boolean[size];  
        Ptr = new int[size];  
        Stk = new int[size];  
        top=-1;  
    }  
  
    public boolean search(int i) {  
        if ((0 <= Ptr[i]) && (Ptr[i] <= top) && (Stk[Ptr[i]] == i))  
            return Bit[i];  
        else return false;  
    }  
    ...
```

# Implementierung: Bitvektor-Darstellung (JAVA) II

```
...
public void insert(int i) {
    if ((0 <= Ptr[i]) && (Ptr[i] <= top) && (Stk[Ptr[i]] == i))
        Bit[i] = true;
    else {
        top++;
        Ptr[i] = top;
        Stk[top] = i;
        Bit[i] = true;
    }
}

public void delete(int i) {
    Bit[i]=false;
}

}
```

# Anmerkungen

- Die Variable  $\text{top}$  gibt die maximale Anzahl der jemals angesprochenen Elemente von  $S \subseteq U$  an.

# Anmerkungen

- Die Variable `top` gibt die maximale Anzahl der jemals angesprochenen Elemente von  $S \subseteq U$  an.
- bei der `delete`-Operation wird nichts zurückgesetzt außer `Bit[i]`

# Anmerkungen

- Die Variable **top** gibt die maximale Anzahl der jemals angesprochenen Elemente von  $S \subseteq U$  an.
- bei der **delete**-Operation wird nichts zurückgesetzt außer **Bit[i]**
- Die Information des Bitvektors **Bit[0..N-1]** könnte als zusätzliche Information (ein Bit) im Zeigerfeld **Ptr[0..N-1]** hinterlegt sein

# Anmerkungen

- Die Variable `top` gibt die maximale Anzahl der jemals angesprochenen Elemente von  $S \subseteq U$  an.
- bei der `delete`-Operation wird nichts zurückgesetzt außer `Bit[i]`
- Die Information des Bitvektors `Bit[0..N-1]` könnte als zusätzliche Information (ein Bit) im Zeigerfeld `Ptr[0..N-1]` hinterlegt sein
- Die Methode funktioniert auch für die Initialisierung großer Real-/Integer-Felder in numerischen Anwendungen

# Vergleich einfacher Suchverfahren

Methode	#Vergleiche	Platz	Vorteil	Nachteil
lineare Suche	$O(N)$	$N$	keine Initialisierung	hohe Suchkosten
binäre Suche	$O(\log N)$	$N$	worst case in $O(\log N)$	sortiertes Feld
Bit-Vektor	$O(1)$	$N$	schnellstmögliche Suche	nur für kleine Universen / Initialisierungsaufwand $O(N)$
Bit-Vektor ohne Initialisierung	$O(1)$	$3 \cdot N$	schnellstmögliche Suche mit Initialisierung $O(1)$	nur für kleine Universen

# V. Suchen in Mengen

---

## 5. Suchen in Mengen

- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. Hashing**
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume
  - 5.5.1. Gewichtsbalanceierte Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume
- 5.6. *Priority Queue* und *Heap*

# Motivation

- **Counting Sort / Bit-Vektor-Darstellung von Mengen**
  - die **Speicherstelle** eines **Record** wird allein aus dem **Schlüssel** berechnet
  - Daher: Einfügen / Suche mit  $O(1)$

- **Counting Sort / Bit-Vektor-Darstellung von Mengen**
  - die **Speicherstelle** eines **Record** wird allein aus dem **Schlüssel** berechnet
  - Daher: Einfügen / Suche mit  $O(1)$
- **Hashing ist eine Erweiterung dieses Konzeptes**
  - aber: Zuordnung Schlüssel  $\leftrightarrow$  Speicherstelle **nicht mehr ein-eindeutig**
  - es kann zu s.g. Kollisionen kommen; d.h. mehrere Schlüssel werden einer Speicherstelle zugeordnet

# Motivation

- **Counting Sort / Bit-Vektor-Darstellung von Mengen**
  - die **Speicherstelle** eines **Record** wird allein aus dem **Schlüssel** berechnet
  - Daher: Einfügen / Suche mit  $O(1)$
- **Hashing ist eine Erweiterung dieses Konzeptes**
  - aber: Zuordnung Schlüssel  $\leftrightarrow$  Speicherstelle **nicht mehr ein-eindeutig**
  - es kann zu s.g. Kollisionen kommen; d.h. mehrere Schlüssel werden einer Speicherstelle zugeordnet
- **Auch bekannt als „gestreute Speicherung“**

# Hashing: Das Prinzip

- **Prinzip**
  - in **Hashtabelle**  $T[0..m-1]$  können  $m$  Records gespeichert werden

# Hashing: Das Prinzip

- **Prinzip**
  - in **Hashtabelle**  $T[0..m-1]$  können  $m$  Records gespeichert werden
  - jedem Element  $a \in U$  ( $U$ =Universum;  $|U|=N$ ) wird durch **Hashfunktion**  $h$  ein Index in die Hashtabelle  $T$  zugeordnet:

$$h: U \rightarrow \{0..m-1\}$$

# Hashing: Das Prinzip

- **Prinzip**
  - in **Hashtabelle**  $T[0..m-1]$  können  $m$  Records gespeichert werden
  - jedem Element  $a \in U$  ( $U$ =Universum;  $|U|=N$ ) wird durch **Hashfunktion**  $h$  ein Index in die Hashtabelle  $T$  zugeordnet:
$$h: U \rightarrow \{0..m-1\}$$
  - $a$  kann demnach in  $T[h(a)]$  gespeichert werden wenn an dieser Position noch Platz ist ...

# Hashing: Das Prinzip

- **Prinzip**

- in **Hashtabelle**  $T[0..m-1]$  können  $m$  Records gespeichert werden
- jedem Element  $a \in U$  ( $U$ =Universum;  $|U|=N$ ) wird durch **Hashfunktion**  $h$  ein Index in die Hashtabelle  $T$  zugeordnet:

$$h: U \rightarrow \{0..m-1\}$$

- $a$  kann demnach in  $T[h(a)]$  gespeichert werden wenn an dieser Position noch Platz ist ...
- in der Regel  $m < N$ ; daher: **Kollisionen** ( $h$  ist nicht injektiv!); d.h. möglich ist

$$h(x) = h(y) \text{ für } x \neq y$$

# Hashing: Das Prinzip

- **Prinzip**

- in **Hashtabelle**  $T[0..m-1]$  können  $m$  Records gespeichert werden
- jedem Element  $a \in U$  ( $U=\text{Universum}; |U|=N$ ) wird durch **Hashfunktion**  $h$  ein Index in die Hashtabelle  $T$  zugeordnet:

$$h: U \rightarrow \{0..m-1\}$$

- $a$  kann demnach in  $T[h(a)]$  gespeichert werden wenn an dieser Position noch Platz ist ...
- in der Regel  $m < N$ ; daher: **Kollisionen** ( $h$  ist nicht injektiv!); d.h. möglich ist

$$h(x) = h(y) \text{ für } x \neq y$$

- $a$  wird also nicht notwendigerweise in  $T[h(a)]$  gespeichert; zwei Möglichkeiten:

# Hashing: Das Prinzip

- **Prinzip**

- in **Hashtabelle**  $T[0..m-1]$  können  $m$  Records gespeichert werden
- jedem Element  $a \in U$  ( $U=\text{Universum}; |U|=N$ ) wird durch **Hashfunktion**  $h$  ein Index in die Hashtabelle  $T$  zugeordnet:

$$h: U \rightarrow \{0..m-1\}$$

- $a$  kann demnach in  $T[h(a)]$  gespeichert werden wenn an dieser Position noch Platz ist ...
- in der Regel  $m < N$ ; daher: **Kollisionen** ( $h$  ist nicht injektiv!); d.h. möglich ist
$$h(x) = h(y) \quad \text{für} \quad x \neq y$$
- $a$  wird also nicht notwendigerweise in  $T[h(a)]$  gespeichert; zwei Möglichkeiten:
  - $T[h(a)]$  enthält einen **Verweis auf einen Bucket (offenes Hashing)**

# Hashing: Das Prinzip

- **Prinzip**

- in **Hashtabelle**  $T[0..m-1]$  können  $m$  Records gespeichert werden
- jedem Element  $a \in U$  ( $U=\text{Universum}; |U|=N$ ) wird durch **Hashfunktion**  $h$  ein Index in die Hashtabelle  $T$  zugeordnet:

$$h: U \rightarrow \{0..m-1\}$$

- $a$  kann demnach in  $T[h(a)]$  gespeichert werden wenn an dieser Position noch Platz ist ...
- in der Regel  $m < N$ ; daher: **Kollisionen** ( $h$  ist nicht injektiv!); d.h. möglich ist

$$h(x) = h(y) \quad \text{für} \quad x \neq y$$

- $a$  wird also nicht notwendigerweise in  $T[h(a)]$  gespeichert; zwei Möglichkeiten:
  - $T[h(a)]$  enthält einen **Verweis auf einen Bucket (offenes Hashing)**
  - mittels einer **Sondierungsfunktion** wird eine alternative Speicheradresse berechnet (**geschlossenes Hashing**)

# Hashing: Beispiel

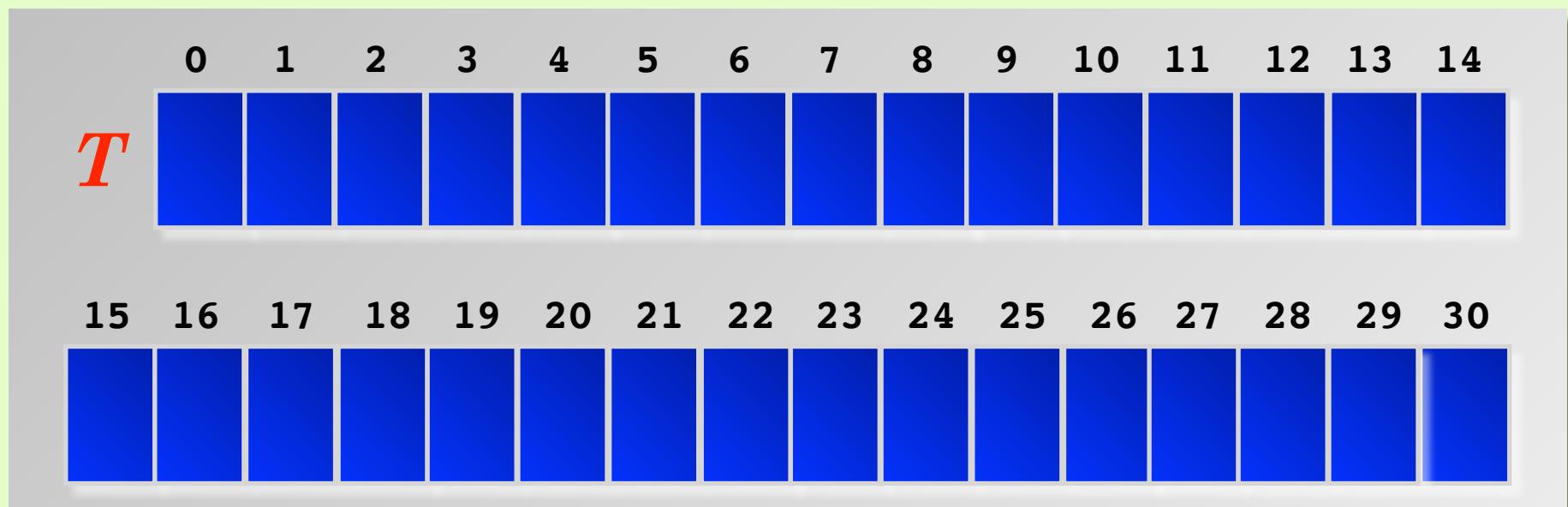
## B Hashing

Gegeben:

- Hashtabelle  $T$  mit  $m=31$  Einträgen
- Hashfunktion  $h(x)=x \bmod 31$

Wir tragen nacheinander die folgenden Werte in die Hashtabelle ein:

**71, 107, 134, 162, 190, 344, 364, 495, 496, 640, 699, 769, 801, 833, 963**



# Hashing: Beispiel

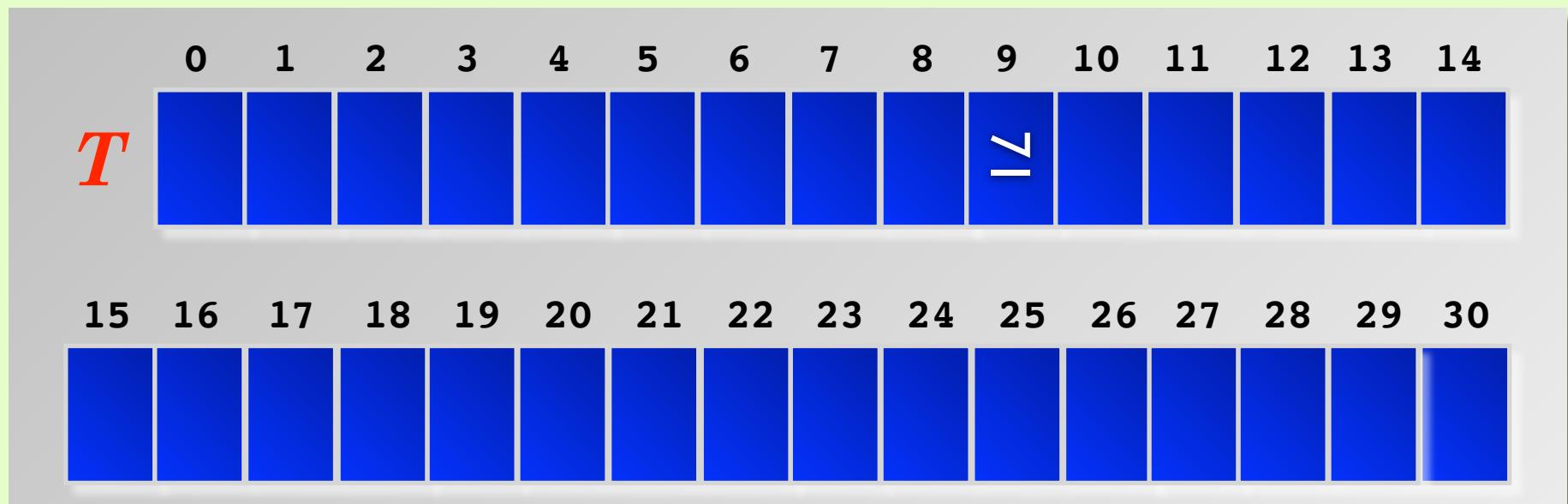
## B Hashing - Einfügen in Hashtabelle I

Gegeben:

- Hashtabelle  $T$  mit  $m=31$  Einträgen
- Hashfunktion  $h(x)=x \bmod 31$

Wir tragen nacheinander die folgenden Werte in die Hashtabelle ein:

**71, 107, 134, 162, 190, 344, 364, 495, 496, 640, 699, 769, 801, 833, 963**



z.B.:  $h(71)=71 \bmod 31 = 9$

# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle II

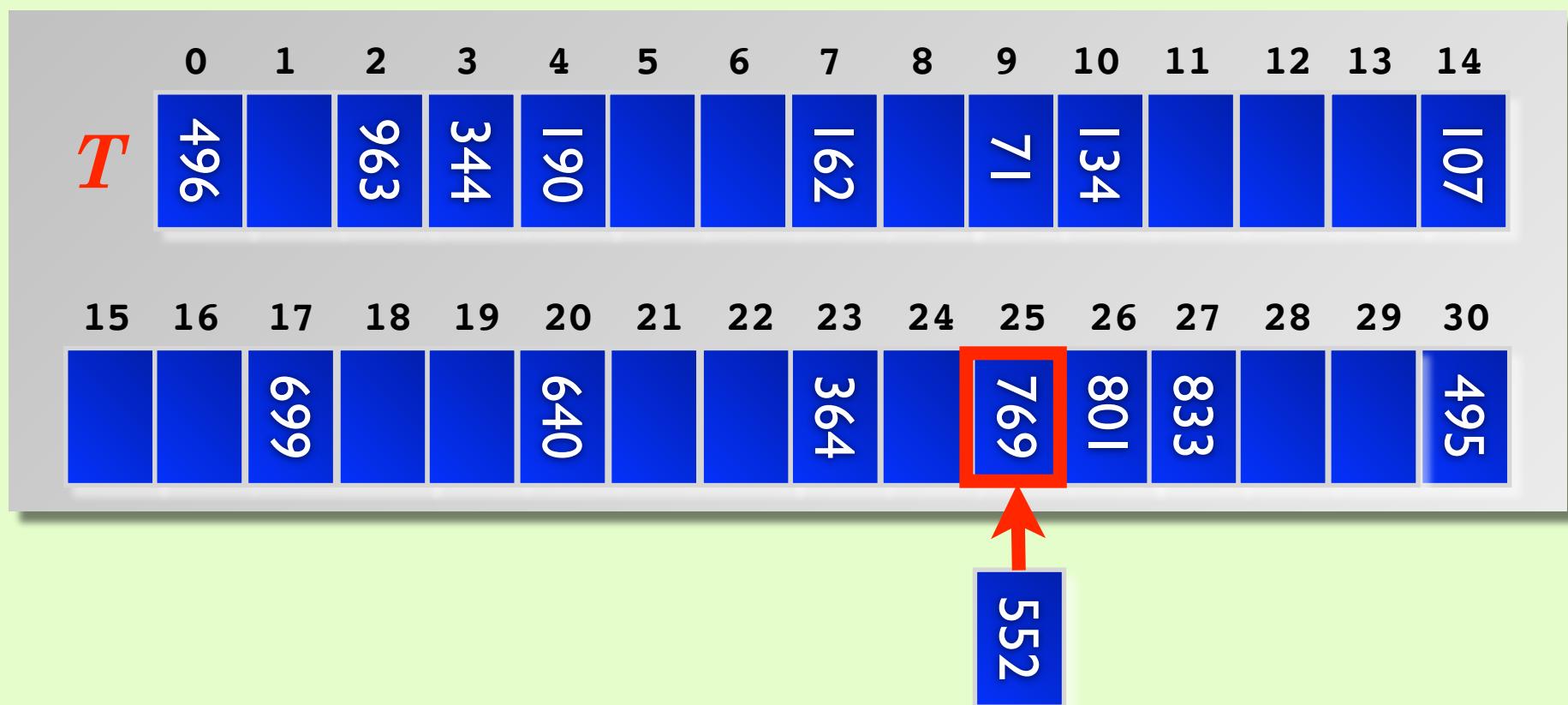
- Beim Hinzufügen des Schlüssels 552 kommt es zu einer **Kollision** bei Index 25

$T$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
	496		963	344	190			162		71	134				107	
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
			699		640			364		769	801	833			495	

# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle II

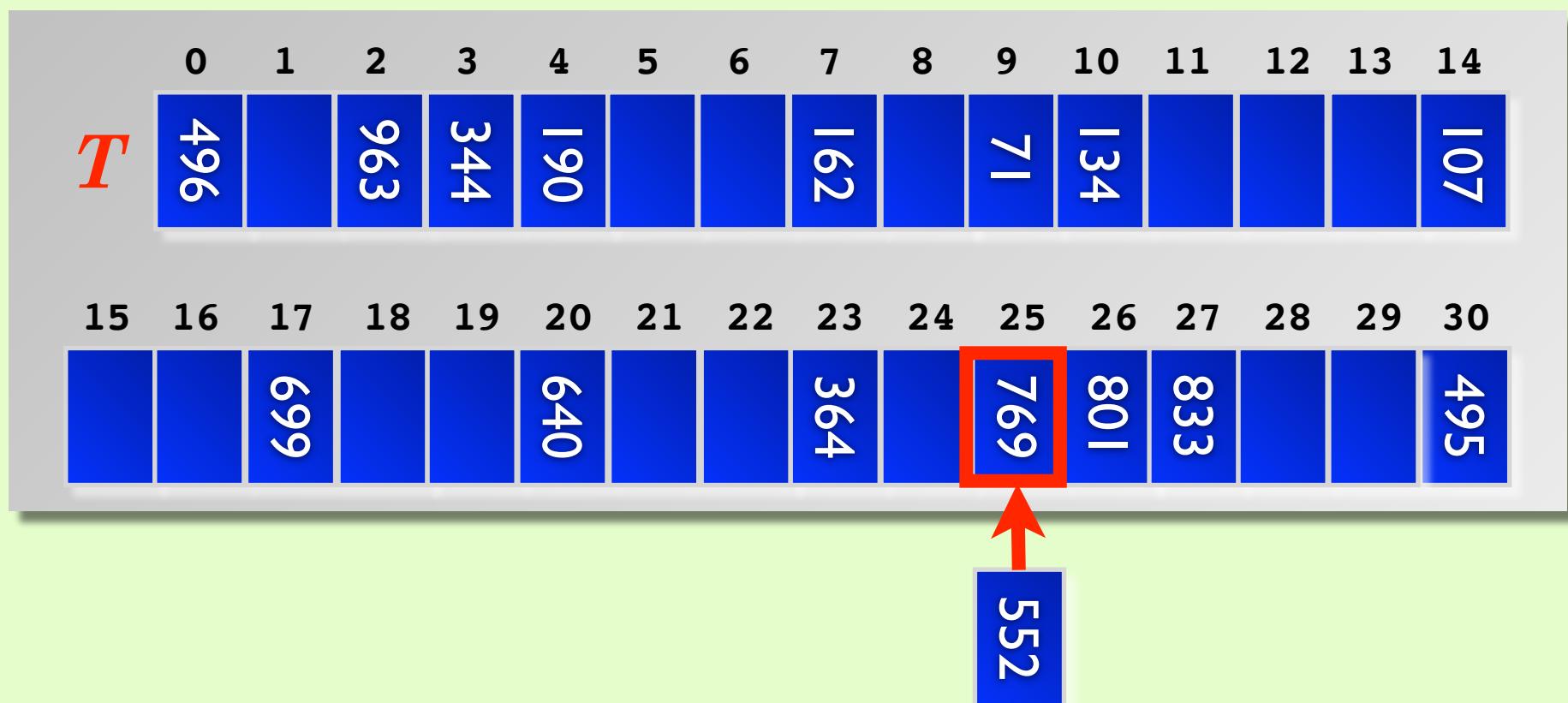
- Beim Hinzufügen des Schlüssels 552 kommt es zu einer **Kollision** bei Index 25



# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle II

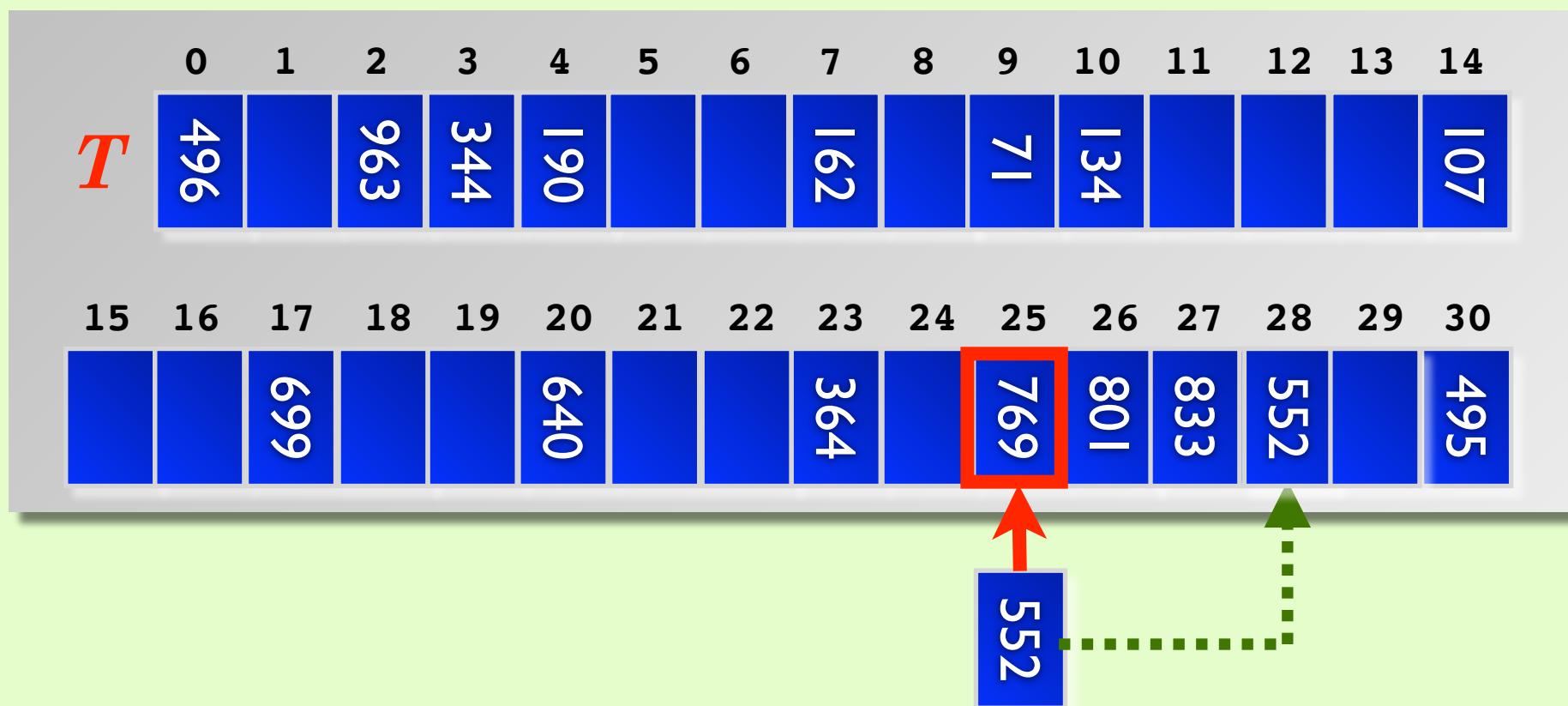
- Beim Hinzufügen des Schlüssels 552 kommt es zu einer **Kollision** bei Index 25
- **Kollisionsstrategie** (Sondierungsfunktion): Schlüssel auf den nächsten freien Speicherplatz rechts eintragen



# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle II

- Beim Hinzufügen des Schlüssels 552 kommt es zu einer **Kollision** bei Index 25
  - **Kollisionsstrategie** (Sondierungsfunktion): Schlüssel auf den nächsten freien Speicherplatz rechts eintragen



# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle III

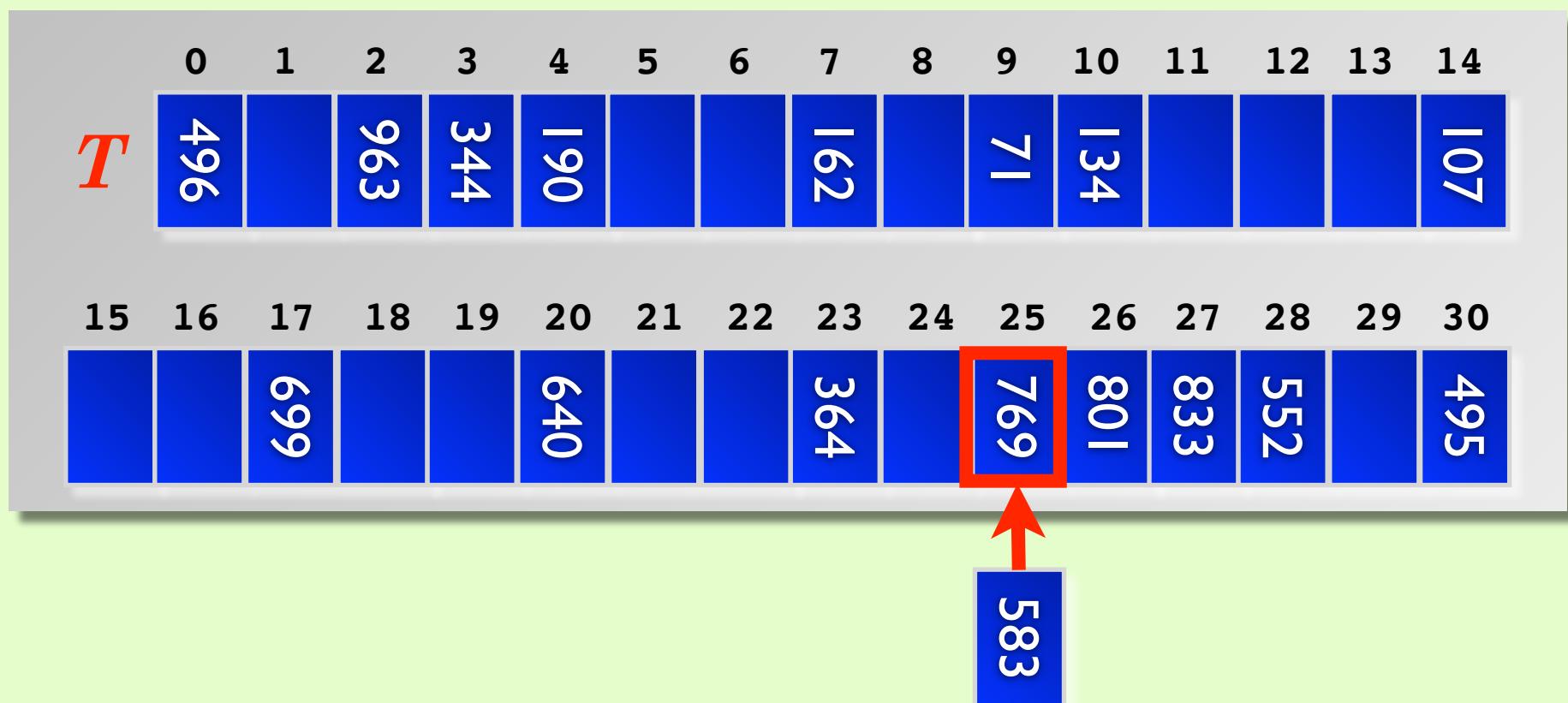
- Beim Hinzufügen des Schlüssels 583 kommt es erneut zu einer **Kollision** beim Index 25

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
<b>T</b>	496		963	344	190			162		71	134				107	
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
			699		640			364		769	801	833	552		495	

# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle III

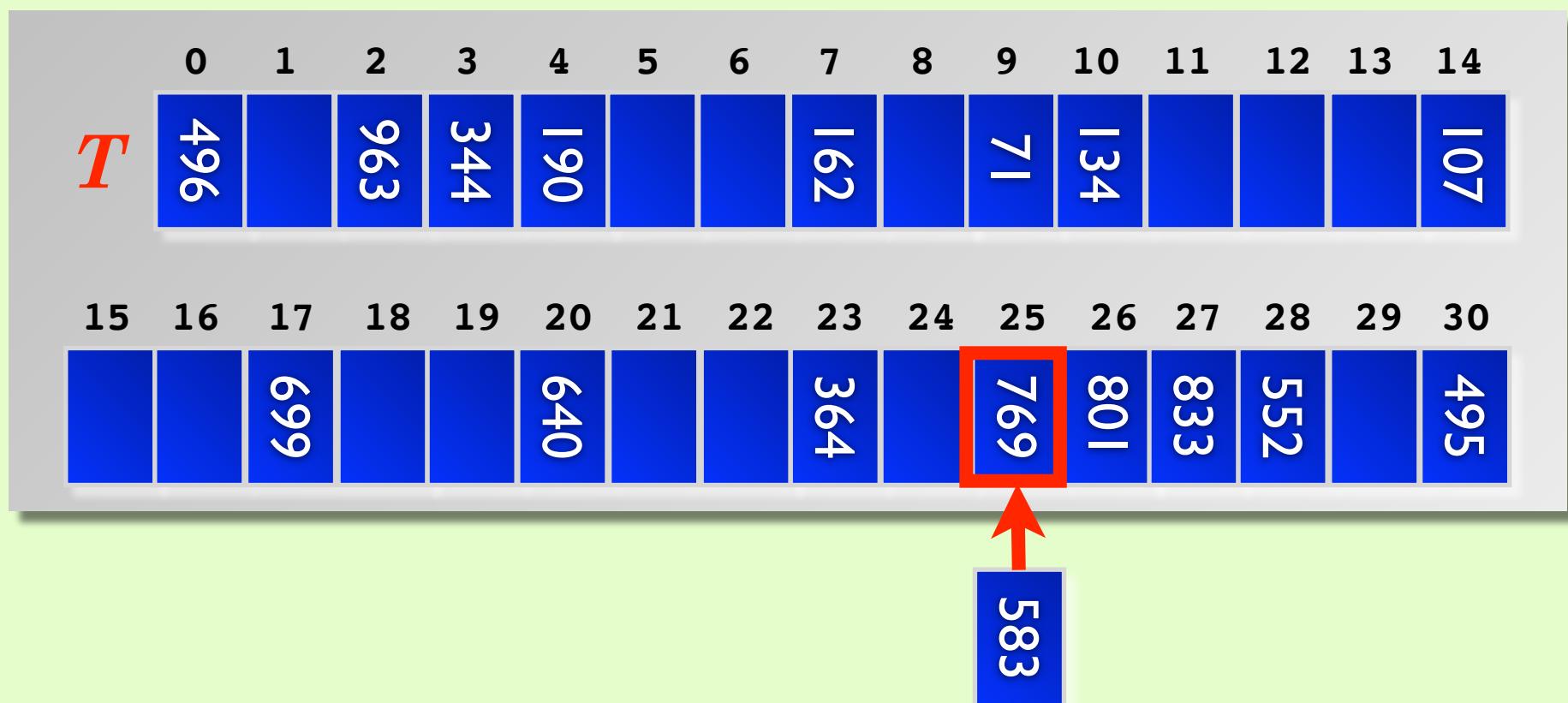
- Beim Hinzufügen des Schlüssels 583 kommt es erneut zu einer **Kollision** beim Index 25



# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle III

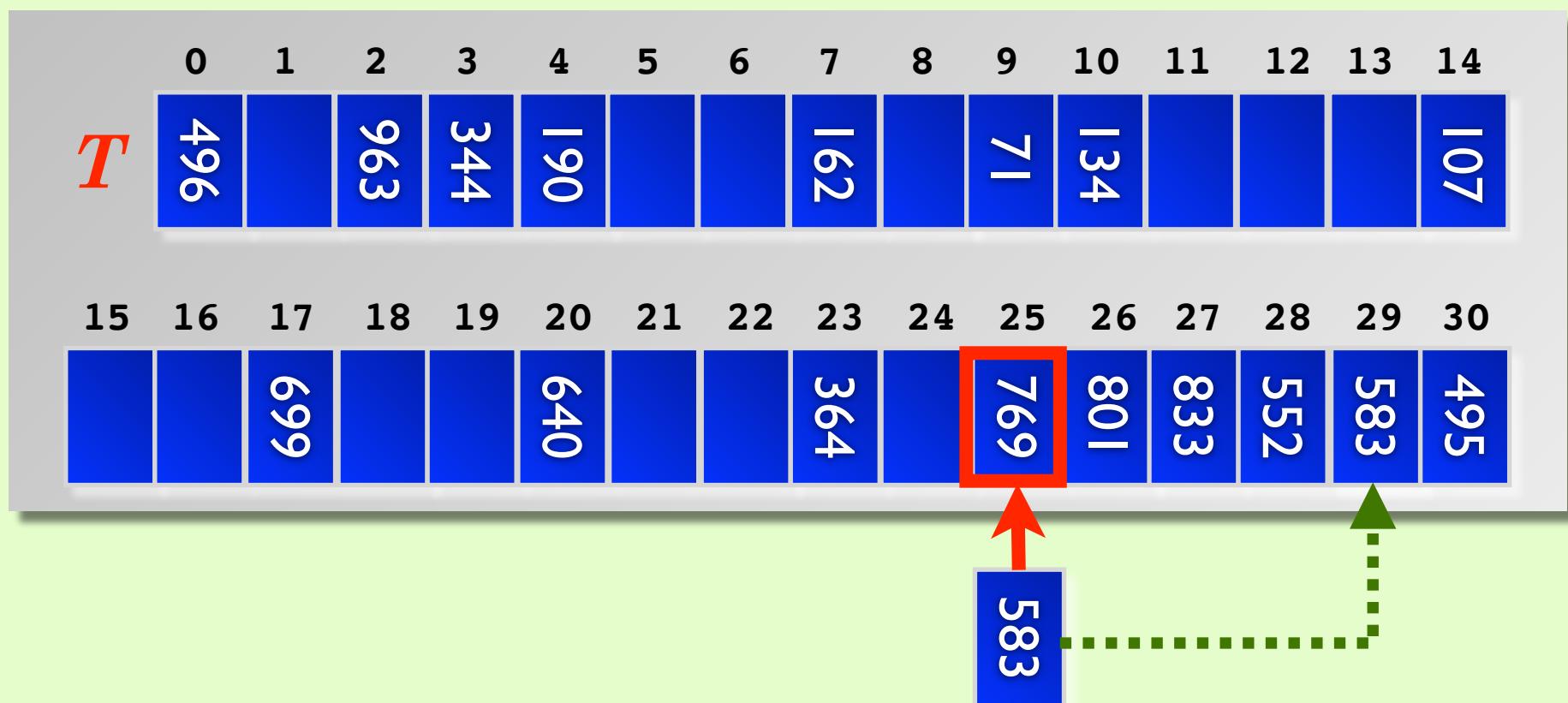
- Beim Hinzufügen des Schlüssels 583 kommt es erneut zu einer **Kollision** beim Index 25
- **Kollisionsstrategie** (Sondierungsfunktion): Schlüssel auf den nächsten freien Speicherplatz rechts eintragen



# Hashing: Beispiel

## B Hashing - Einfügen in Hashtabelle III

- Beim Hinzufügen des Schlüssels 583 kommt es erneut zu einer **Kollision** beim Index 25
- **Kollisionsstrategie** (Sondierungsfunktion): Schlüssel auf den nächsten freien Speicherplatz rechts eintragen



# Hashing: Beispiel

## B Hashing - Probleme beim Sondieren

- Bei der gewählten Kollisionsstrategie (lineares Sondieren) kommt es zu **Häufungen (Clustern)**, was den Einfügeaufwand erhöht

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	496		963	344	190			162		71	134				107
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	30
			699		640			364		769	801	833	552	583	495

# Hashing: Beispiel

## B Hashing - Probleme beim Sondieren

- Bei der gewählten Kollisionsstrategie (lineares Sondieren) kommt es zu **Häufungen (Clustern)**, was den Einfügeaufwand erhöht
- Durch die **Clusterbildung** wird auch das Suchen aufwändiger  
→ Die Vorteile des Hashing schwinden ...

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
<b>T</b>	496		963	344	190		162		71	134				107	
15	16	17	18	19	20	21	22	23	24	25	26	27	28	30	
		699			640			364		769	801	833	552	583	495

# Hashing: Beispiel

## B Hashing - Suchen

Suche nach  $a$ :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<b>T</b>	496		963	344	190			162		71	134				107
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	30
			699		640			364		769	801	833	552	583	495

# Hashing: Beispiel

## B Hashing - Suchen

Suche nach  $a$ :

- prüfe, ob  $T[h(a)] = a$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<b>T</b>	496		963	344	190			162		71	134				107
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	30
			699		640			364		769	801	833	552	583	495

# Hashing: Beispiel

## B Hashing - Suchen

Suche nach  $a$ :

- prüfe, ob  $T[h(a)] = a$
- Wenn nein: die rechts von  $h(a)$  stehenden Elemente durchsuchen, bis

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<b>T</b>	496		963	344	190			162		71	134				107
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	30
			699		640			364		769	801	833	552	583	495

# Hashing: Beispiel

## B Hashing - Suchen

Suche nach  $a$ :

- prüfe, ob  $T[h(a)] = a$
- Wenn nein: die rechts von  $h(a)$  stehenden Elemente durchsuchen, bis
  - das Element gefunden wurde,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<b>T</b>	496		963	344	190			162		71	134				107
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	30
			699		640			364		769	801	833	552	583	495

# Hashing: Beispiel

## B Hashing - Suchen

Suche nach  $a$ :

- prüfe, ob  $T[h(a)] = a$
- Wenn nein: die rechts von  $h(a)$  stehenden Elemente durchsuchen, bis
  - das Element gefunden wurde,
  - ein leerer Tabellenplatz erreicht wurde oder wir am Ausgangspunkt angelangt (nicht gefunden)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
<b>T</b>	496		963	344	190		162		71	134				107	
15	16	17	18	19	20	21	22	23	24	25	26	27	28	30	
		699			640			364		769	801	833	552	583	495

# Anforderungen an die Hashfunktion

An die Hashfunktion  $h$  werden folgende Anforderungen gestellt:

- Die gesamte Hashtabelle wird abgedeckt -  $h$  ist **surjektiv**
- $h(x)$  soll die **Schlüssel**  $x$  möglichst **gleichmäßig** auf die Hashtabelle **verteilen** (Vermeidung von Kollisionen)
- Die Berechnung soll **effizient** (nicht zeitaufwendig) sein

Nachfolgend werden einige typische Hashfunktionen untersucht

- Dabei gehen wir davon aus, dass für alle Schlüssel  $a \in U$  gilt
$$a \in \mathbb{N}_0$$
- Fast alle in der Praxis vorkommenden Schlüssel lassen sich entsprechend umwandeln; z.B. durch Umsetzung von Buchstaben in Zahlen

# Hashfunktion: Divisions-Rest-Methode

## D **Divisions-Rest-Methode**

Sei  $m$  die Größe der Hashtabelle. Dann definiert man  $h(x) : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  wie folgt:

$$h(x) = x \bmod m$$

# Hashfunktion: Divisions-Rest-Methode

## D **Divisions-Rest-Methode**

Sei  $m$  die Größe der Hashtabelle. Dann definiert man  $h(x) : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  wie folgt:

$$h(x) = x \bmod m$$

Eine surjektive und gleichmäßige Verteilung erreicht man z.B. durch folgende Wahl für  $m$ :

- $m$  ist Primzahl
- $m$  teilt nicht  $2^i \cdot j$ , wobei  $i, j$  kleine Zahlen aus  $\mathbb{N}_0$  sind

# Hashfunktion: Divisions-Rest-Methode

## D Divisions-Rest-Methode

Sei  $m$  die Größe der Hashtabelle. Dann definiert man  $h(x) : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  wie folgt:

$$h(x) = x \bmod m$$

Eine surjektive und gleichmäßige Verteilung erreicht man z.B. durch folgende Wahl für  $m$ :

- $m$  ist Primzahl
- $m$  teilt nicht  $2^i \cdot j$ , wobei  $i, j$  kleine Zahlen aus  $\mathbb{N}_0$  sind

✓ einfache Berechenbarkeit

✗ Tendenz zum Clustering

Aufeinanderfolgende Schlüssel werden tendenziell an aufeinanderfolgenden Speicherstellen abgelegt

# Beispiel: Divisions-Rest-Methode

## B Beispiel: Divisions-Rest-Methode

Die Zeichenketten  $c_1, \dots, c_k$  werden auf  $\mathbb{N}_0$  abgebildet und auf die Hashtabelle verteilt:

$$h(c_1, \dots, c_k) = \left( \sum_{i=1}^k N(c_i) \right) \bmod m$$

mit  $N(A) = 1, N(B) = 2, \dots, N(Z) = 26$ .

Zur Vereinfachung werden nur die ersten drei Buchstaben betrachtet und es wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Wir wählen  $m = 17$  und  $S$ =deutsche Monatsnamen (ohne Umlaute) - es kommt zu drei Kollisionen.

# Beispiel: Divisions-Rest-Methode

## B Beispiel: Divisions-Rest-Methode

Die Zeichenketten  $c_1, \dots, c_k$  werden auf  $\mathbb{N}_0$  abgebildet und auf die Hashtabelle verteilt:

$$h(c_1, \dots, c_k) = \left( \sum_{i=1}^k N(c_i) \right) \bmod m$$

mit  $N(A) = 1, N(B) = 2, \dots, N(Z) = 26$ .

Zur Vereinfachung werden nur die ersten drei Buchstaben betrachtet und es wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Wir wählen  $m = 17$  und  $S$ =deutsche Monatsnamen (ohne Umlaute) - es kommt zu drei Kollisionen.

0	November
1	April, Dezember
2	Maerz
3	
4	
5	
6	Mai, September
7	
8	Januar
9	Juli
10	
11	Juni
12	August, Oktober
13	Februar
14	
15	
16	

# Hashfunktion: Mittel-Quadrat-Methode

- **Ziel:** benachbarte Schlüssel auf die ganze Hashtabelle verteilen  
(Vermeidung der Clusterbildung)

# Hashfunktion: Mittel-Quadrat-Methode

- **Ziel:** benachbarte Schlüssel auf die ganze Hashtabelle verteilen  
(Vermeidung der Clusterbildung)

## D Mittel-Quadrat-Methode

$h(x) = \text{mittlerer Block der Ziffern von } x^2$

# Hashfunktion: Mittel-Quadrat-Methode

- **Ziel:** benachbarte Schlüssel auf die ganze Hashtabelle verteilen  
(Vermeidung der Clusterbildung)

## D Mittel-Quadrat-Methode

$$h(x) = \text{mittlerer Block der Ziffern von } x^2$$

- Da der mittlere Block von **allen Ziffern** von  $x$  abhängt, wird eine größere Streuung erreicht

# Hashfunktion: Mittel-Quadrat-Methode

- **Ziel:** benachbarte Schlüssel auf die ganze Hashtabelle verteilen  
(Vermeidung der Clusterbildung)

## D Mittel-Quadrat-Methode

$$h(x) = \text{mittlerer Block der Ziffern von } x^2$$

- Da der mittlere Block von **allen Ziffern** von  $x$  abhängt, wird eine größere Streuung erreicht
- Für  $m=100$  ergibt sich im Vergleich zur Divisions-Rest-Methode:

$x$	$x \bmod 100$	$x^2$	$h(x)$
127	27	16129	12
128	28	16384	38
129	29	16641	64

# Wahrscheinlichkeit von Kollisionen I

- **Hauptproblem beim Hashing: Kollisionen**

# Wahrscheinlichkeit von Kollisionen I

- Hauptproblem beim Hashing: Kollisionen

**Wie häufig kommt es  
zu Kollisionen?**

# Wahrscheinlichkeit von Kollisionen I

- Hauptproblem beim Hashing: **Kollisionen**

**Wie häufig kommt es  
zu Kollisionen?**

- Analoges Problem der Mathematik: **Geburtstags-Paradoxon:**

- Wie groß ist die Wahrscheinlichkeit, dass zwei aus  $n$  Personen am gleichen Tag Geburtstag haben?
- Analogie zum Hashing:
  - $m=365$  Tage  $\hat{=}$  Größe der Hashtabelle (Zelle  $\hat{=}$  Geburtstag)
  - $n$  Personen  $\hat{=}$  Anzahl der Elemente von  $S \subseteq U$

# Wahrscheinlichkeit von Kollisionen I

- **Hauptproblem beim Hashing: Kollisionen**

**Wie häufig kommt es  
zu Kollisionen?**

- **Analoges Problem der Mathematik: Geburtstags-Paradoxon:**
  - Wie groß ist die Wahrscheinlichkeit, dass zwei aus  $n$  Personen am gleichen Tag Geburtstag haben?
  - Analogie zum Hashing:
    - $m=365$  Tage  $\hat{=}$  Größe der Hashtabelle (Zelle  $\hat{=}$  Geburtstag)
    - $n$  Personen  $\hat{=}$  Anzahl der Elemente von  $S \subseteq U$
- **Annahme:** Die Hashfunktion sei ideal; d.h. die Verteilung über die Hashtabelle sei gleichförmig

# Wahrscheinlichkeit von Kollisionen II

- Für die Wahrscheinlichkeit **mindestens einer Kollision** bei  $n$  Schlüsseln und einer  $m$ -elementigen Hashtabelle  $P_{\text{Koll}}(n,m)$  gilt:

# Wahrscheinlichkeit von Kollisionen II

- Für die Wahrscheinlichkeit **mindestens einer Kollision** bei  $n$  Schlüsseln und einer  $m$ -elementigen Hashtabelle  $P_{\text{Koll}}(n,m)$  gilt:

$$P_{\text{Koll}}(n,m) = 1 - P_{\text{NoKoll}}(n,m)$$

# Wahrscheinlichkeit von Kollisionen II

- Für die Wahrscheinlichkeit **mindestens einer Kollision** bei  $n$  Schlüsseln und einer  $m$ -elementigen Hashtabelle  $P_{\text{Koll}}(n,m)$  gilt:

$$P_{\text{Koll}}(n,m) = 1 - P_{\text{NoKoll}}(n,m)$$

- Sei  $p(i;m)$  die Wahrscheinlichkeit, dass der  $i$ -te Schlüssel ( $i=1,\dots,n$ ) auf einen freien Platz abgebildet wird - wie alle Schlüssel zuvor auch.

# Wahrscheinlichkeit von Kollisionen II

- Für die Wahrscheinlichkeit **mindestens einer Kollision** bei  $n$  Schlüsseln und einer  $m$ -elementigen Hashtabelle  $P_{\text{Koll}}(n,m)$  gilt:

$$P_{\text{Koll}}(n,m) = 1 - P_{\text{NoKoll}}(n,m)$$

- Sei  $p(i;m)$  die Wahrscheinlichkeit, dass der  $i$ -te Schlüssel ( $i=1,\dots,n$ ) auf einen freien Platz abgebildet wird - wie alle Schlüssel zuvor auch.
- Es gilt:

$$\begin{aligned} p(1; m) &= \frac{m-0}{m} = 1 - \frac{0}{m} && , \text{weil } 0 \text{ Plätze belegt und } m \text{ Plätze frei sind} \\ p(2; m) &= \frac{m-1}{m} = 1 - \frac{1}{m} && , \text{weil } 1 \text{ Platz belegt und } m - 1 \text{ Plätze frei sind} \\ &\vdots && \\ p(i; m) &= \frac{m-i+1}{m} = 1 - \frac{i-1}{m} && , \text{weil } i - 1 \text{ Plätze belegt und } m - i + 1 \text{ Plätze frei sind} \end{aligned}$$

# Wahrscheinlichkeit von Kollisionen II

- Für die Wahrscheinlichkeit **mindestens einer Kollision** bei  $n$  Schlüsseln und einer  $m$ -elementigen Hashtabelle  $P_{\text{Koll}}(n,m)$  gilt:

$$P_{\text{Koll}}(n,m) = 1 - P_{\text{NoKoll}}(n,m)$$

- Sei  $p(i;m)$  die Wahrscheinlichkeit, dass der  $i$ -te Schlüssel ( $i=1,\dots,n$ ) auf einen freien Platz abgebildet wird - wie alle Schlüssel zuvor auch.
- Es gilt:

$$\begin{aligned} p(1; m) &= \frac{m-0}{m} = 1 - \frac{0}{m} && , \text{weil } 0 \text{ Plätze belegt und } m \text{ Plätze frei sind} \\ p(2; m) &= \frac{m-1}{m} = 1 - \frac{1}{m} && , \text{weil } 1 \text{ Platz belegt und } m-1 \text{ Plätze frei sind} \\ &\vdots && \\ p(i; m) &= \frac{m-i+1}{m} = 1 - \frac{i-1}{m} && , \text{weil } i-1 \text{ Plätze belegt und } m-i+1 \text{ Plätze frei sind} \end{aligned}$$

- $P_{\text{NoKoll}}(n,m)$  ist dann das Produkt der Wahrscheinlichkeiten  $p(1;m), \dots, p(n;m)$ :

$$\begin{aligned} P_{\text{NoKoll}} &= \prod_{i=1}^n p(i; m) \\ &= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \end{aligned}$$

# Das Geburtstagsproblem (-paradoxon)

- Beim Geburtstagsproblem ist  $m=365$ :

$$P_{\text{Koll}} = 1 - \prod_{i=0}^{n-1} \left(1 - \frac{i}{365}\right)$$

# Das Geburtstagsproblem (-paradoxon)

- Beim Geburtstagsproblem ist  $m=365$ :

$$P_{\text{Koll}} = 1 - \prod_{i=0}^{n-1} \left(1 - \frac{i}{365}\right)$$

$n$	$P_{\text{Koll}}(n,m)$
10	0,11695
20	0,41144
22	0,4757
<b>23</b>	<b>0,5073</b>
24	0,53835
30	0,70632
40	0,89123
50	0,97037

# Das Geburtstagsproblem (-paradoxon)

- Beim Geburtstagsproblem ist  $m=365$ :

$$P_{\text{Koll}} = 1 - \prod_{i=0}^{n-1} \left(1 - \frac{i}{365}\right)$$

$n$	$P_{\text{Koll}}(n,m)$
10	0,11695
20	0,41144
22	0,4757
23	0,5073
24	0,53835
30	0,70632
40	0,89123
50	0,97037

Die Wahrscheinlichkeit,  
dass in einer Gruppe von  
**23** Personen mindestens  
zwei am gleichen Tag  
Geburtstag haben beträgt

**50,7%**

# Größe der Hashtabelle I

**Wie groß sollte eine Hashtabelle sein?**

## Wie groß sollte eine Hashtabelle sein?

- Oder, genauer: Wie muss die Größe der Hashtabelle  $m$  mit der Zahl der Elemente  $n$  **wachsen**, damit  $P_{\text{Koll}}(n, m)$  und somit auch  $P_{\text{NoKoll}}(n, m)$  konstant bleibt?

## Wie groß sollte eine Hashtabelle sein?

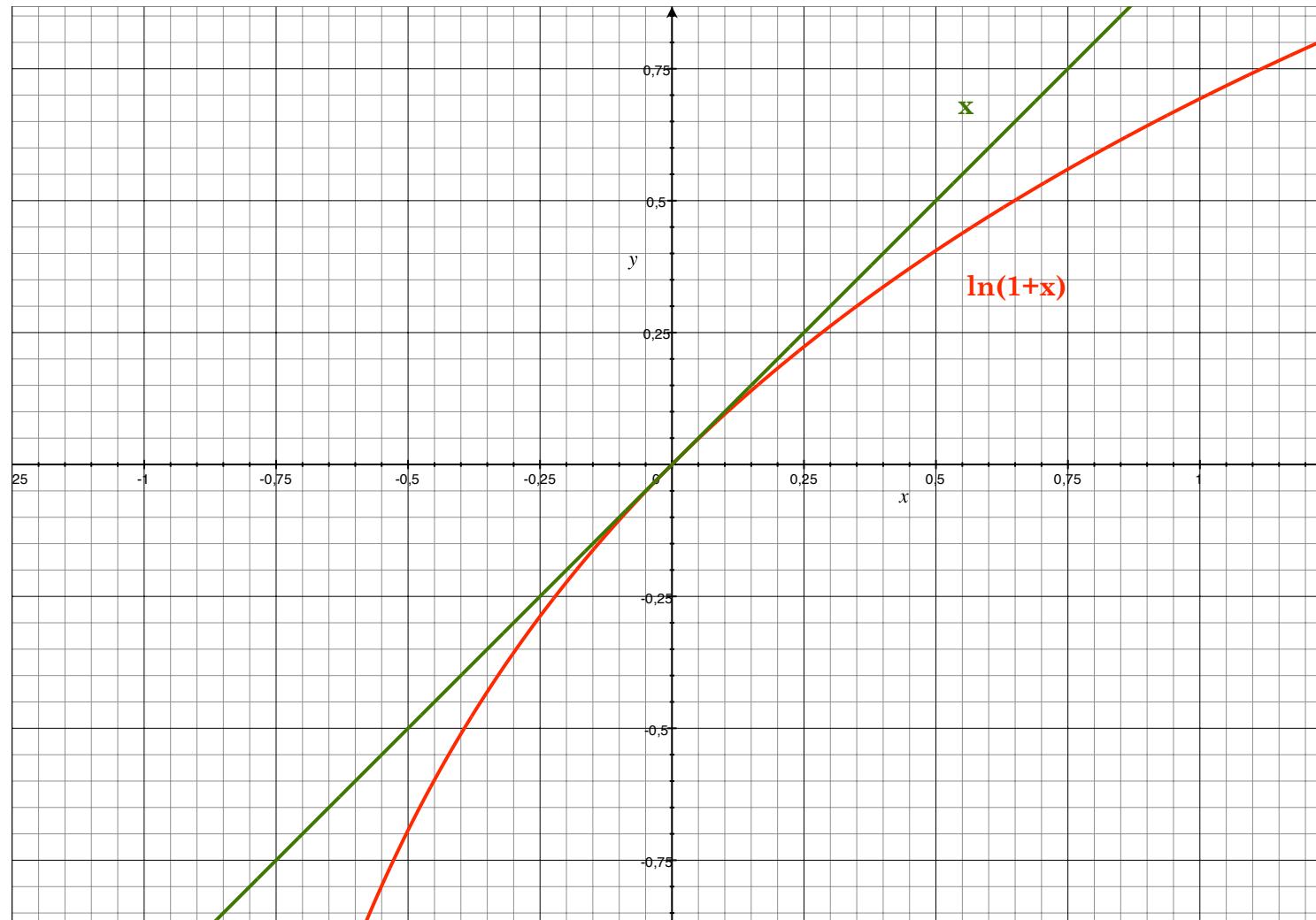
- Oder, genauer: Wie muss die Größe der Hashtabelle  $m$  mit der Zahl der Elemente  $n$  **wachsen**, damit  $P_{\text{Koll}}(n, m)$  und somit auch  $P_{\text{NoKoll}}(n, m)$  konstant bleibt?

$$\begin{aligned} P_{\text{NoKoll}} &= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \\ &\quad (* e^{\ln(x)} = \exp(\ln(x)) = x *) \\ &= \exp \left( \ln \left( \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \right) \right) \\ &\quad (* \ln(a \cdot b) = \ln(a) + \ln(b) *) \\ &= \exp \left( \sum_{i=0}^{n-1} \ln \left(1 - \frac{i}{m}\right) \right) \end{aligned}$$

# Größe der Hashtabelle II

- Betrachtet man den Verlauf des natürlichen Logarithmus so kann man für sehr kleine  $\varepsilon$  wie folgt abschätzen:

$$\ln(1+\varepsilon) \approx \varepsilon$$



# Größe der Hashtabelle III

- Mit der Abschätzung  $\ln \left(1 - \frac{i}{m}\right) \approx -\frac{i}{m}$  erhalten wir

$$\begin{aligned} P_{\text{NoKoll}} &= \exp \left[ - \sum_{i=0}^{n-1} \frac{i}{m} \right] \\ &= \exp \left[ - \frac{1}{m} \cdot \sum_{i=0}^{n-1} i \right] \\ &= \exp \left( - \frac{n \cdot (n-1)}{2 \cdot m} \right) \\ &\approx \exp \left( - \frac{n^2}{2 \cdot m} \right) \end{aligned}$$

# Größe der Hashtabelle III

- Mit der Abschätzung  $\ln \left(1 - \frac{i}{m}\right) \approx -\frac{i}{m}$  erhalten wir

$$\begin{aligned} P_{\text{NoKoll}} &= \exp \left[ - \sum_{i=0}^{n-1} \frac{i}{m} \right] \\ &= \exp \left[ - \frac{1}{m} \cdot \sum_{i=0}^{n-1} i \right] \\ &= \exp \left( - \frac{n \cdot (n-1)}{2 \cdot m} \right) \\ &\approx \exp \left( - \frac{n^2}{2 \cdot m} \right) \end{aligned}$$

Um Kollisionswahrscheinlichkeit konstant zu halten muss Größe der Hashtabelle  $m$  **quadratisch** mit Zahl zu speichernden Elementen  $n$  **wachsen!**

- **Offenes Hashverfahren (*open hashing*)**

- arbeiten mit **dynamischem Speicher** (z.B. verketteten Listen)
- können daher **beliebig viele Schlüssel** unter einem Hashtabellen-Eintrag unterbringen
- werden auch „Hashing mit Verkettung“ genannt

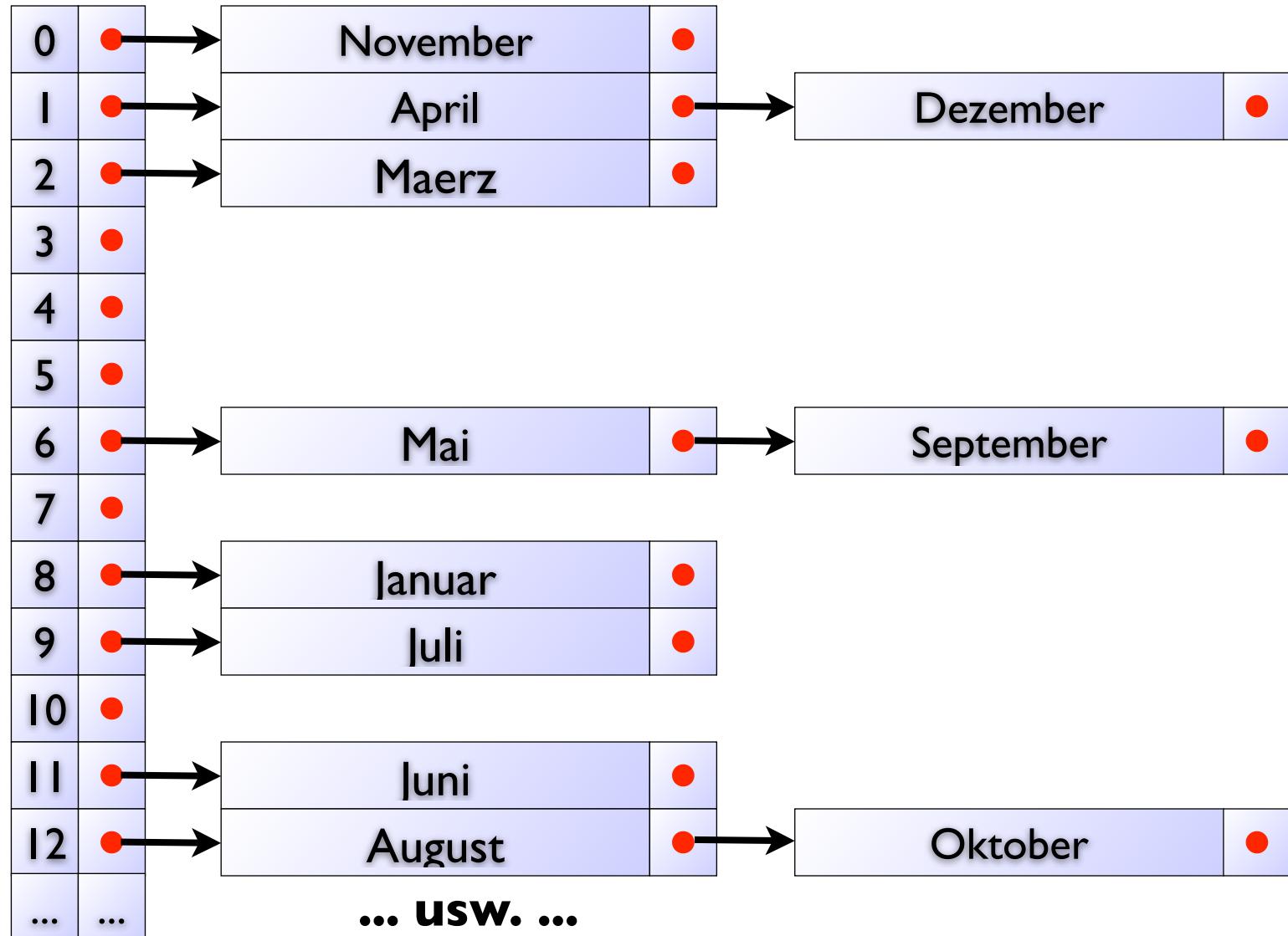
- **Offenes Hashverfahren (*open hashing*)**
  - arbeiten mit **dynamischem Speicher** (z.B. verketteten Listen)
  - können daher **beliebig viele Schlüssel** unter einem Hashtabellen-Eintrag unterbringen
  - werden auch „Hashing mit Verkettung“ genannt
- **Geschlossenes Hashverfahren (*closed hashing*)**
  - können nur eine **begrenzte Zahl von Schlüsseln** (meistens nur einen) unter einem Hashtabellen-Eintrag unterbringen.
  - arbeiten mit einem Array und werden wegen des abgeschlossenen Speicherangebots als geschlossen bezeichnet.
  - müssen bei Kollisionen mittels Sondierungsfunktionen neue Adressen suchen
  - Man spricht auch vom „Hashing mit offener Adressierung“

# *Open Hashing*

- **Die Hashtabelle besteht aus  $m$  linearen Listen**
- **Für das Beispiel mit den Monatsnamen ergibt sich z.B.:**

# Open Hashing

- Die Hashtabelle besteht aus  $m$  linearen Listen
- Für das Beispiel mit den Monatsnamen ergibt sich z.B.:



# *Open Hashing: Kostenabschätzung*

- **Jede der Operationen  $insert(x, S)$ ,  $delete(x, S)$  und  $search(x, S)$  setzt sich aus zwei Operationen zusammen:**
  - Berechnung der Adresse und Aufsuchen des Behälters
  - Durchlaufen der unter der Adresse gespeicherten Einträge

# *Open Hashing: Kostenabschätzung*

- **Jede der Operationen  $insert(x, S)$ ,  $delete(x, S)$  und  $search(x, S)$  setzt sich aus zwei Operationen zusammen:**
  - Berechnung der Adresse und Aufsuchen des Behälters
  - Durchlaufen der unter der Adresse gespeicherten Einträge
- Sei  $\alpha = \frac{n}{m}$  der **Belegungsfaktor**

# Open Hashing: Kostenabschätzung

- Jede der Operationen  $insert(x, S)$ ,  $delete(x, S)$  und  $search(x, S)$  setzt sich aus zwei Operationen zusammen:
  - Berechnung der Adresse und Aufsuchen des Behälters
  - Durchlaufen der unter der Adresse gespeicherten Einträge
- Sei  $\alpha = \frac{n}{m}$  der **Belegungsfaktor**
- **Zeitkomplexität**
  - Adresse berechnen:  $O(1)$
  - Behälter aufsuchen:  $O(1)$
  - Liste durchsuchen
    - im **average case** (erfolgreiche Suche)  $O(1 + \alpha/2)$
    - im **worst case** (erfolglose Suche)  $O(\alpha) = O(n)$

# Open Hashing: Kostenabschätzung

- Jede der Operationen  $insert(x, S)$ ,  $delete(x, S)$  und  $search(x, S)$  setzt sich aus zwei Operationen zusammen:
  - Berechnung der Adresse und Aufsuchen des Behälters
  - Durchlaufen der unter der Adresse gespeicherten Einträge
- Sei  $\alpha = \frac{n}{m}$  der **Belegungsfaktor**
- **Zeitkomplexität**
  - Adresse berechnen:  $O(1)$
  - Behälter aufsuchen:  $O(1)$
  - Liste durchsuchen
    - im **average case** (erfolgreiche Suche)  $O(1 + \alpha/2)$
    - im **worst case** (erfolglose Suche)  $O(\alpha) = O(n)$
- **Platzkomplexität**
  - beträgt  $O(m + n)$

# Open Hashing: Kostenabschätzung

- Jede der Operationen  $insert(x, S)$ ,  $delete(x, S)$  und  $search(x, S)$  setzt sich aus zwei Operationen zusammen:

- Berechnung der Adresse und Aufsuchen des Behälters
- Durchlaufen der unter der Adresse gespeicherten Einträge

- Sei  $\alpha = \frac{n}{m}$  der **Belegungsfaktor**

- **Zeitkomplexität**

- Adresse berechnen:  $O(1)$
- Behälter aufsuchen:  $O(1)$
- Liste durchsuchen
  - im **average case** (erfolgreiche Suche)  $O(1 + \alpha/2)$
  - im **worst case** (erfolglose Suche)  $O(\alpha) = O(n)$

- **Platzkomplexität**

- beträgt  $O(m + n)$

## Verhalten

- $a << l$ : wie Array-Adressierung
- $a \approx l$ : Übergangsbereich
- $a >> l$ : wie verkettete Liste

# *Closed Hashing*

- **Voraussetzung:**  $n < m$  (Hashtabelle groß genug für alle Elemente)

# *Closed Hashing*

- **Voraussetzung:**  $n < m$  (Hashtabelle groß genug für alle Elemente)
- **Arbeitet mit einem statischen Array**

# *Closed Hashing*

- **Voraussetzung:**  $n < m$  (Hashtabelle groß genug für alle Elemente)
- **Arbeitet mit einem statischen Array**
- **Kollisionen:** neuerlicher Adressberechnung („Offene Adressierung“)

- **Voraussetzung:**  $n < m$  (Hashtabelle groß genug für alle Elemente)
- **Arbeitet mit einem statischen Array**
- **Kollisionen:** neuerlicher Adressberechnung („Offene Adressierung“)
- **Dazu:** **Sondierungsfunktion** (auch **Rehashing-Funktion** genannt).
  - listet Schritt für Schritt alle abzusuchenden Positionen auf (liefert also eine Permutationen aller Hashtabellenplätze)
  - Sondierungsschritt als Parameter der Hashfunktion:  
 $h(a,j)$  ist die Position von  $a$  im  $j$ -ten Sondierungsschritt

# Closed Hashing

- **Voraussetzung:**  $n < m$  (Hashtabelle groß genug für alle Elemente)
- **Arbeitet mit einem statischen Array**
- **Kollisionen:** neuerlicher Adressberechnung („Offene Adressierung“)
- **Dazu: Sondierungsfunktion** (auch *Rehashing-Funktion* genannt).
  - listet Schritt für Schritt alle abzusuchenden Positionen auf (liefert also eine Permutationen aller Hashtabellenplätze)
  - Sondierungsschritt als Parameter der Hashfunktion:  
 $h(a,j)$  ist die Position von  $a$  im  $j$ -ten Sondierungsschritt
- Zwei Arten von Kollisionen werden Unterschieden:
  - **Primärkollision:**  $h(x,0) = h(y,0)$
  - **Sekundärkollision:**  $h(x,j) = h(y,0)$  für  $j=1,2,\dots$

# Vorsicht beim Löschen!

- **Achtung**  $\text{delete}(x, S)$  erfordert eine **Sonderbehandlung!**

Elemente, die an  $x$  mittels **Rehashing** vorbeigeleitet wurden, müssen auch nach Löschen von  $x$  noch gefunden werden

- **Lösung**

- Ergänze  $U$  um zwei Werte **empty** und **deleted**
- **deleted** kennzeichnet, dass nach diesem Feld noch weitergesucht werden muss

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
<b>T</b>	496			963	344	190			162		71	134			107	
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	
						640			364		769	801	833	552	582	495

—  $a \bmod 31 = 25$

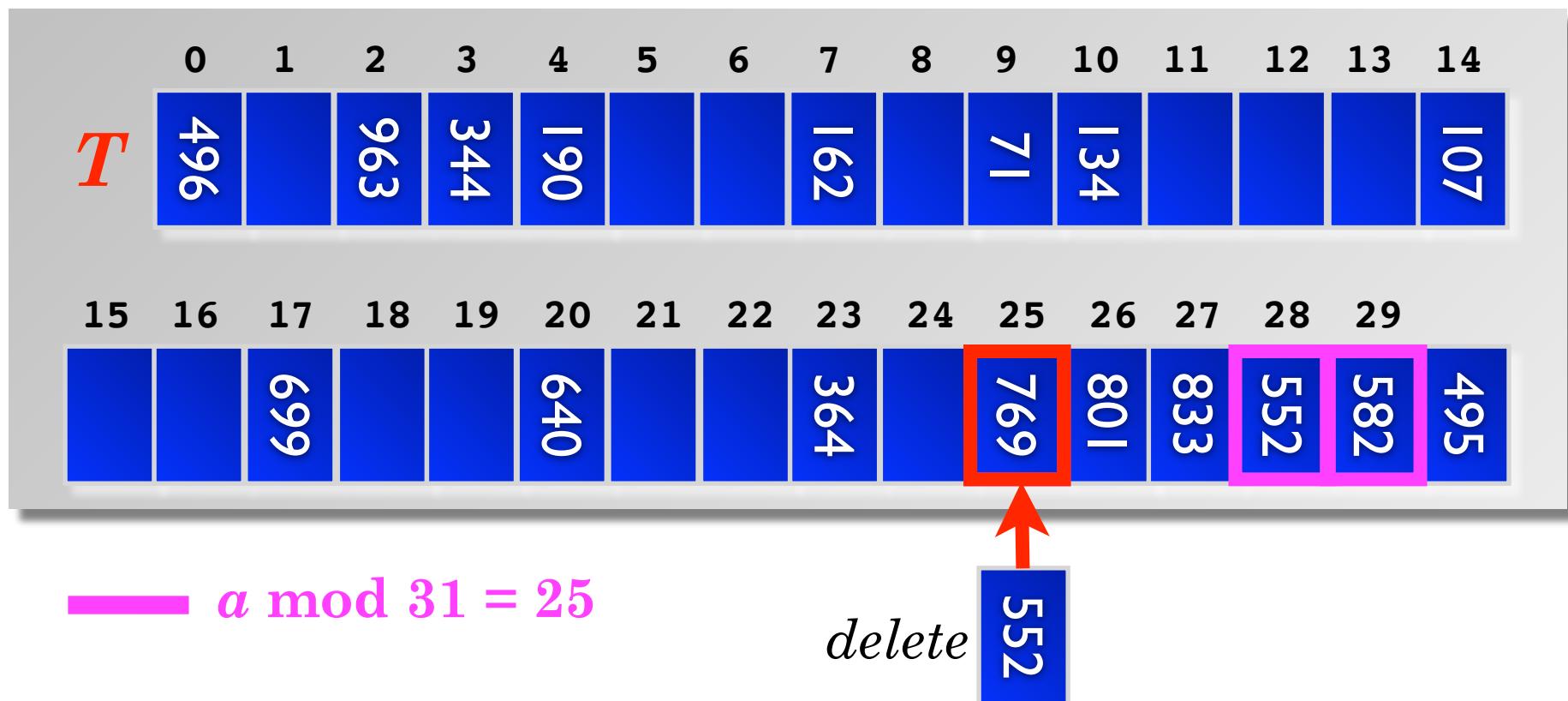
# Vorsicht beim Löschen!

- **Achtung**  $\text{delete}(x, S)$  erfordert eine **Sonderbehandlung!**

Elemente, die an  $x$  mittels **Rehashing** vorbeigeleitet wurden, müssen auch nach Löschen von  $x$  noch gefunden werden

- **Lösung**

- Ergänze  $U$  um zwei Werte **empty** und **deleted**
- **deleted** kennzeichnet, dass nach diesem Feld noch weitergesucht werden muss



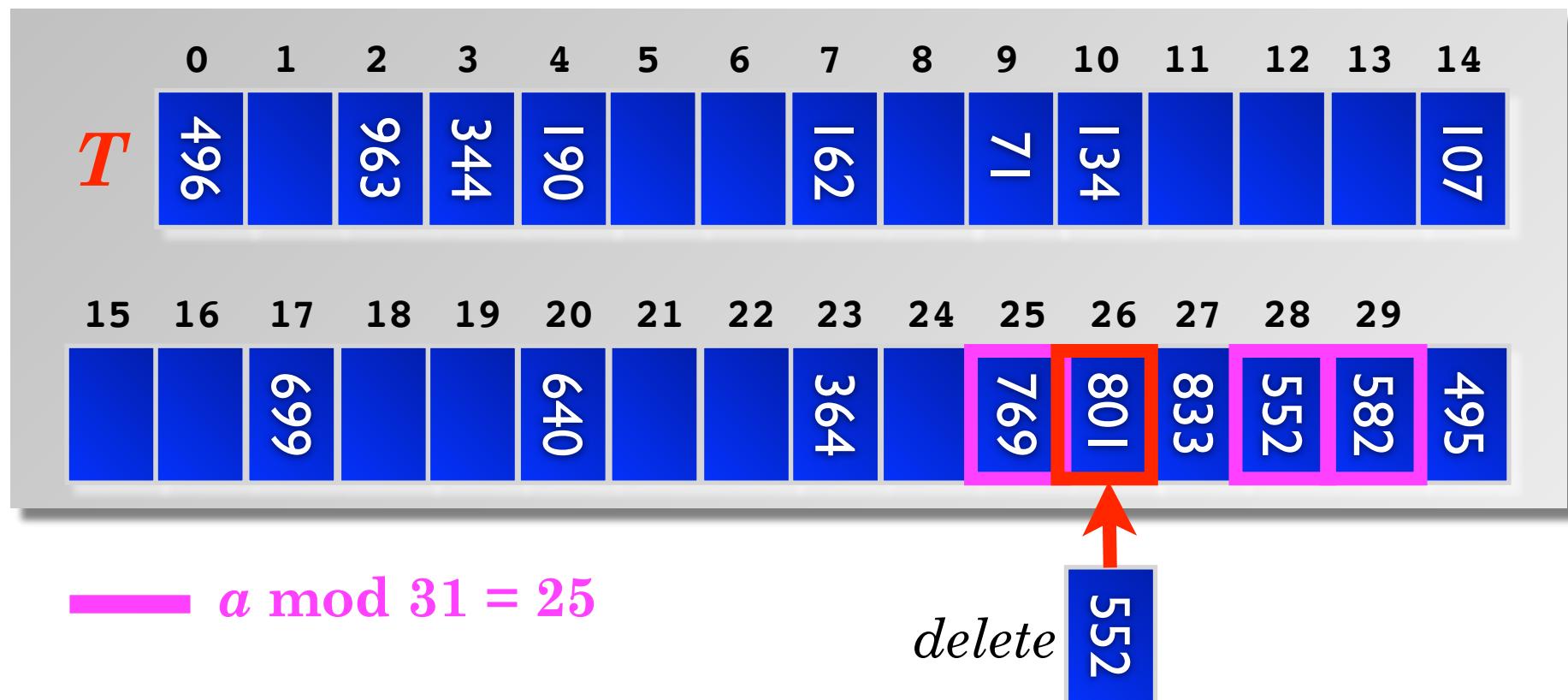
# Vorsicht beim Löschen!

- **Achtung**  $\text{delete}(x, S)$  erfordert eine **Sonderbehandlung!**

Elemente, die an  $x$  mittels **Rehashing** vorbeigeleitet wurden, müssen auch nach Löschen von  $x$  noch gefunden werden

- **Lösung**

- Ergänze  $U$  um zwei Werte **empty** und **deleted**
- **deleted** kennzeichnet, dass nach diesem Feld noch weitergesucht werden muss



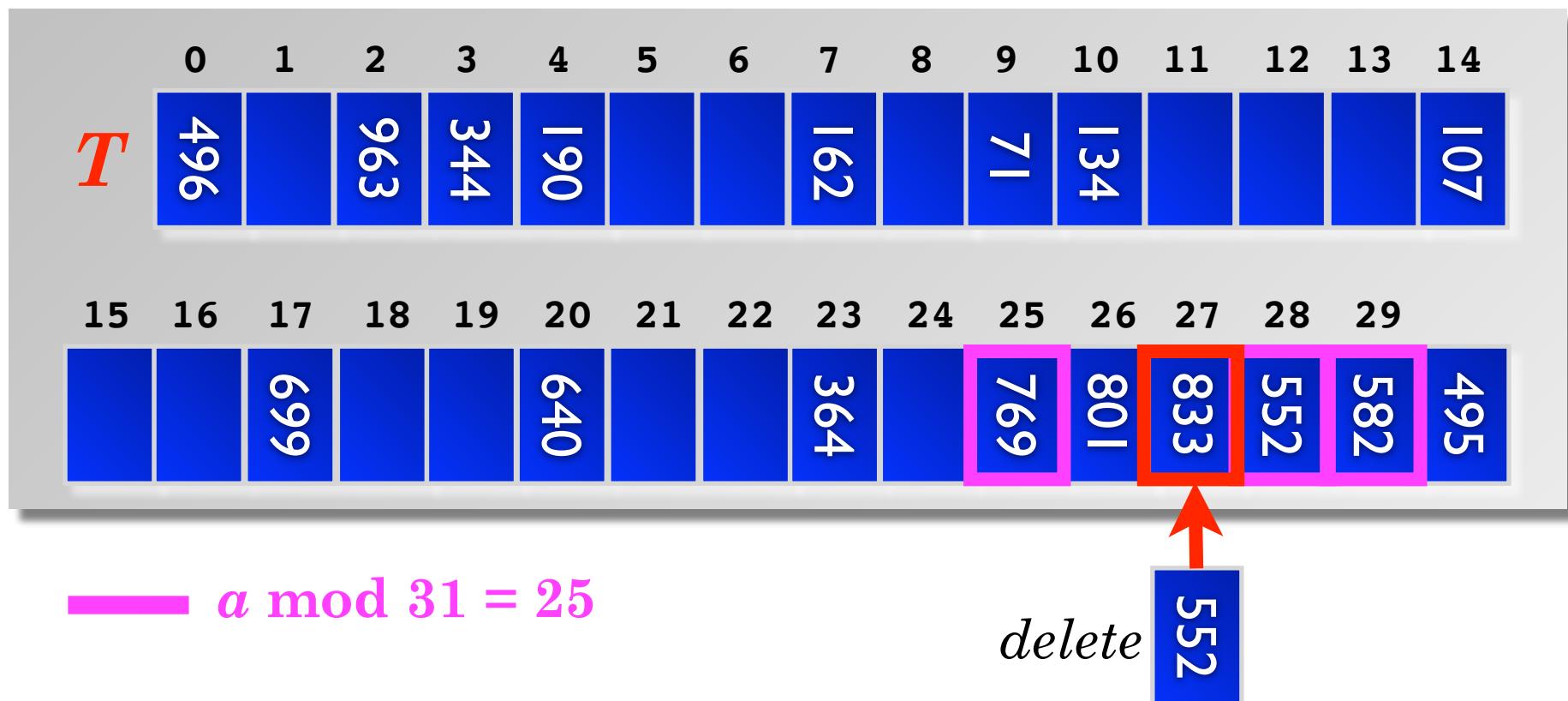
# Vorsicht beim Löschen!

- **Achtung**  $\text{delete}(x, S)$  erfordert eine **Sonderbehandlung!**

Elemente, die an  $x$  mittels **Rehashing** vorbeigeleitet wurden, müssen auch nach Löschen von  $x$  noch gefunden werden

- **Lösung**

- Ergänze  $U$  um zwei Werte **empty** und **deleted**
- **deleted** kennzeichnet, dass nach diesem Feld noch weitergesucht werden muss



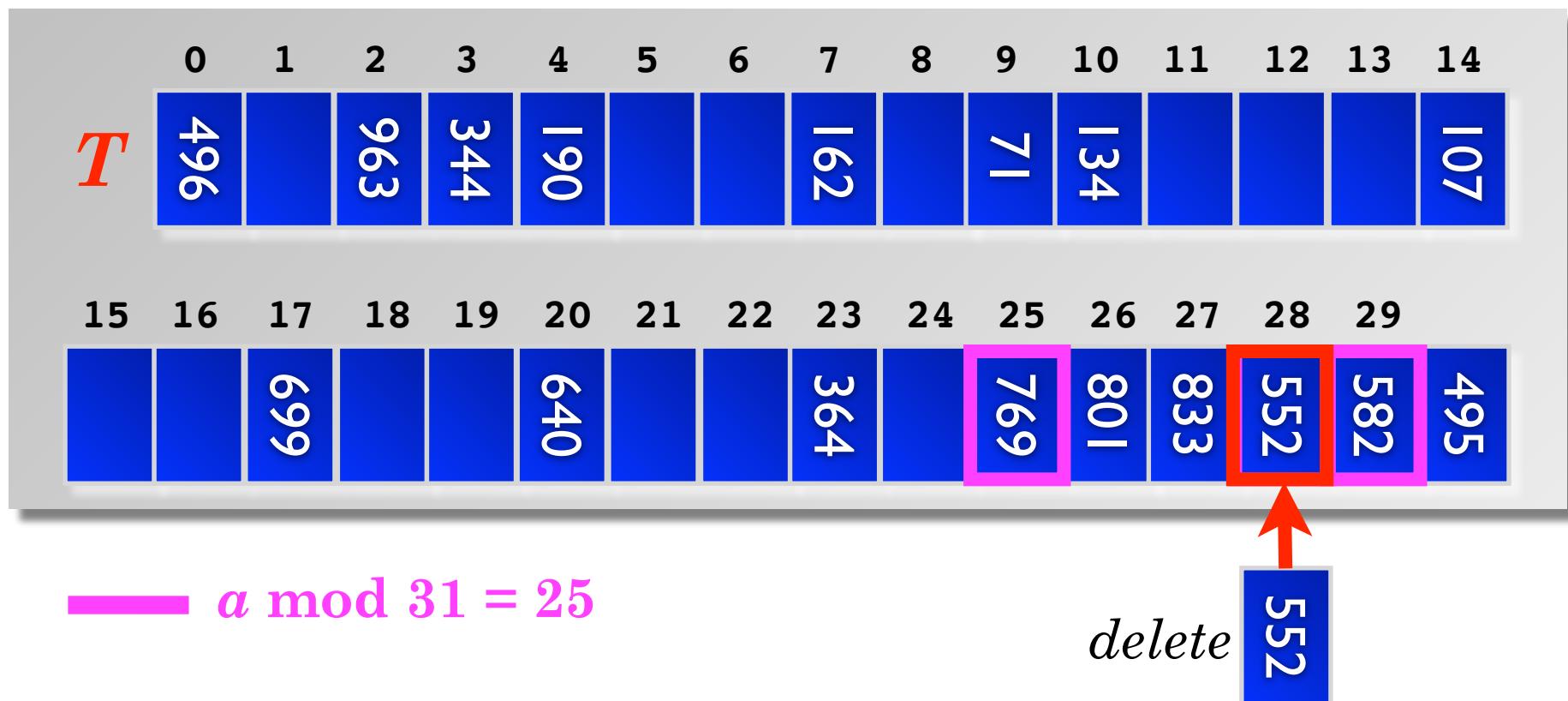
# Vorsicht beim Löschen!

- **Achtung**  $\text{delete}(x, S)$  erfordert eine **Sonderbehandlung!**

Elemente, die an  $x$  mittels **Rehashing** vorbeigeleitet wurden, müssen auch nach Löschen von  $x$  noch gefunden werden

- **Lösung**

- Ergänze  $U$  um zwei Werte **empty** und **deleted**
- **deleted** kennzeichnet, dass nach diesem Feld noch weitergesucht werden muss



# Vorsicht beim Löschen!

- **Achtung**  $\text{delete}(x, S)$  erfordert eine **Sonderbehandlung!**

Elemente, die an  $x$  mittels **Rehashing** vorbeigeleitet wurden, müssen auch nach Löschen von  $x$  noch gefunden werden

- **Lösung**

- Ergänze  $U$  um zwei Werte **empty** und **deleted**
- **deleted** kennzeichnet, dass nach diesem Feld noch weitergesucht werden muss

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
<b>T</b>	496			963	344	190			162		71	134			107	
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	
						640			364		769	801	833	deleted	582	495

—  $a \bmod 31 = 25$

# *Rehashing* durch lineares Sondieren

# Rehashing durch lineares Sondieren

- **Sondierungsfunktion:**

$$h(x,j) = (h(x) + j) \bmod m \quad 0 \leq j \leq m-1$$

# Rehashing durch lineares Sondieren

- **Sondierungsfunktion:**

$$h(x, j) = (h(x) + j) \bmod m \quad 0 \leq j \leq m-1$$

- **Problematisch ist die Clusterbildung:**

- Tendenziell entstehen immer längere zusammenhängende Abschnitte in der Hashtabelle
- Such- und Einfügezeit verschlechtern sich deutlich

0	November
1	April (Dezember)
2	Maerz
3	Dezember
4	
5	
6	Mai, (September)
7	September
8	Januar
9	Juli
10	
11	Juni
12	August, (Oktober)
13	Februar
14	Oktober
15	
16	

# Rehashing durch lineares Sondieren

- **Sondierungsfunktion:**

$$h(x,j) = (h(x) + j) \bmod m \quad 0 \leq j \leq m-1$$

- **Problematisch ist die Clusterbildung:**

- Tendenziell entstehen immer längere zusammenhängende Abschnitte in der Hashtabelle
- Such- und Einfügezeit verschlechtern sich deutlich

- **Verallgemeinerung:**

$$h(x,j) = (h(x) + c \cdot j) \bmod m \quad 0 \leq j \leq m-1$$

Dabei sollten  $c$  und  $m$  teilerfremd sein

0	November
1	April (Dezember)
2	Maerz
3	Dezember
4	
5	
6	Mai, (September)
7	September
8	Januar
9	Juli
10	
11	Juni
12	August, (Oktober)
13	Februar
14	Oktober
15	
16	

# Rehashing durch quadratisches Sondieren

- **Sondierungsfunktion:**

$$h(x, j) = (h(x) + j^2) \bmod m \quad 0 \leq j \leq m-1$$

- **Warum  $j^2$ ?**

- Ist  $m$  eine Primzahl dann sind die Zahlen  $j^2 \bmod m$  alle verschieden für  $j=0, \dots, \lfloor(m/2)\rfloor$

- **Verfeinerung:**  $1 \leq j \leq \frac{m-1}{2}$

$$\begin{aligned} h(x, 0) &= h(x) \bmod m \\ h(x, 2j - 1) &= (h(x) + j^2) \bmod m \\ h(x, 2j) &= (h(x) - j^2) \bmod m \end{aligned}$$

wähle  $m$  als Primzahl mit  $m \bmod 4 = 3$  (Basis: Zahlentheorie)

- **Ergebnis:**

- ✖ Keine Verbesserung für Primärkollisionen
- ✓ vermeidet Clusterbildung bei Sekundärkollisionen
- ✓ Insgesamt: Wahrscheinlichkeit der Bildung langer Ketten sinkt!

# Doppelhashing 1

- Wähle zwei Hashfunktionen  $h(x)$  und  $h'(x)$  für die gilt:

# Doppelhashing 1

- Wähle zwei Hashfunktionen  $h(x)$  und  $h'(x)$  für die gilt:

$$P(h(x) = h(y)) = \frac{1}{m} \quad x \neq y$$

$$P(h'(x) = h'(y)) = \frac{1}{m}$$

# Doppelhashing 1

- Wähle zwei Hashfunktionen  $h(x)$  und  $h'(x)$  für die gilt:

$$P(h(x) = h(y)) = \frac{1}{m} \quad x \neq y$$

$$P(h'(x) = h'(y)) = \frac{1}{m}$$

- Kollisionswahrscheinlichkeiten sollte unabhängig sein; d.h.

$$P(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2}$$

# Doppelhashing 1

- Wähle zwei Hashfunktionen  $h(x)$  und  $h'(x)$  für die gilt:

$$P(h(x) = h(y)) = \frac{1}{m} \quad x \neq y$$

$$P(h'(x) = h'(y)) = \frac{1}{m}$$

- Kollisionswahrscheinlichkeiten sollte unabhängig sein; d.h.

$$P(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2}$$

- **Sondierungsfunktion:**

$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

- **Wichtig: Sondieren Abhängig vom Schlüssel!**

Erste Sondierung bestimmt **nicht** die gesamte Sequenz

- **Jedes Paar  $h(x)$  und  $h'(x)$  erzeugt andere Sequenz**

Daher können  $m^2$  verschiedene Permutationen erzeugt werden

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B  $m' = m-1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$

$$m = 11$$

$$m' = 10$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$

0	22
1	
2	
3	14
4	4
5	
6	28
7	
8	
9	31
10	10

$$m = 11$$

$$m' = 10$$

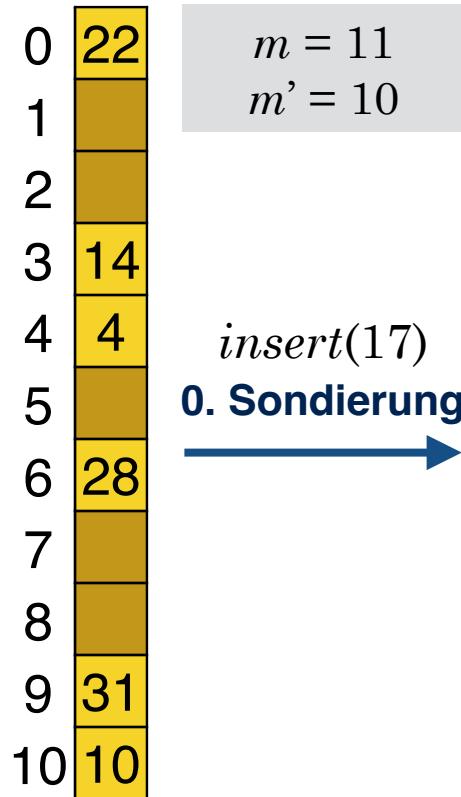
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m$ , „etwas“ kleiner (z.B.  $m = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



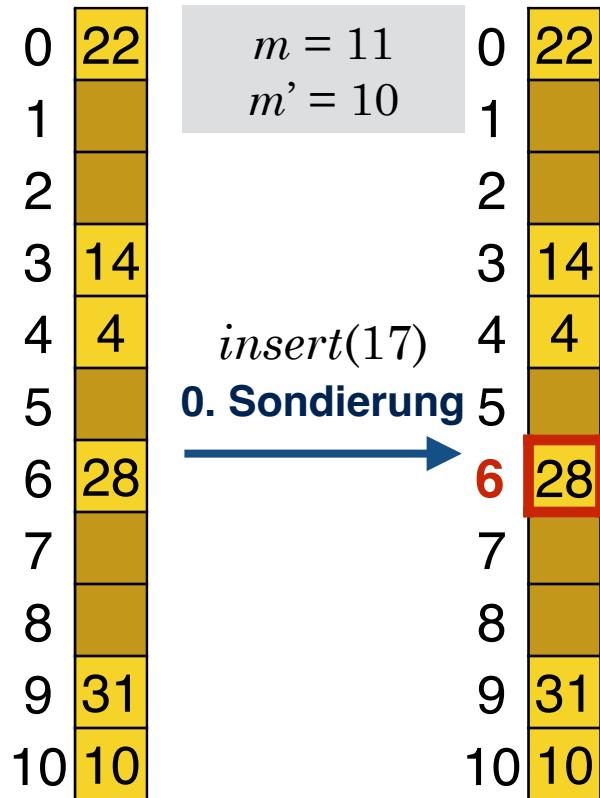
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m$ , „etwas“ kleiner (z.B.  $m = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



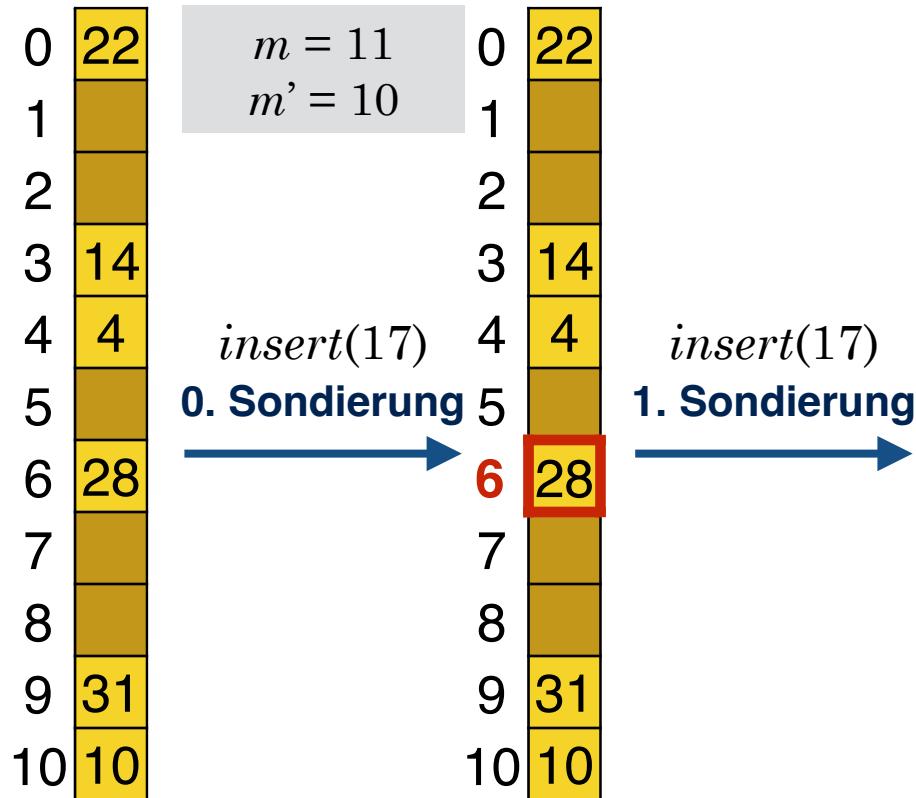
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



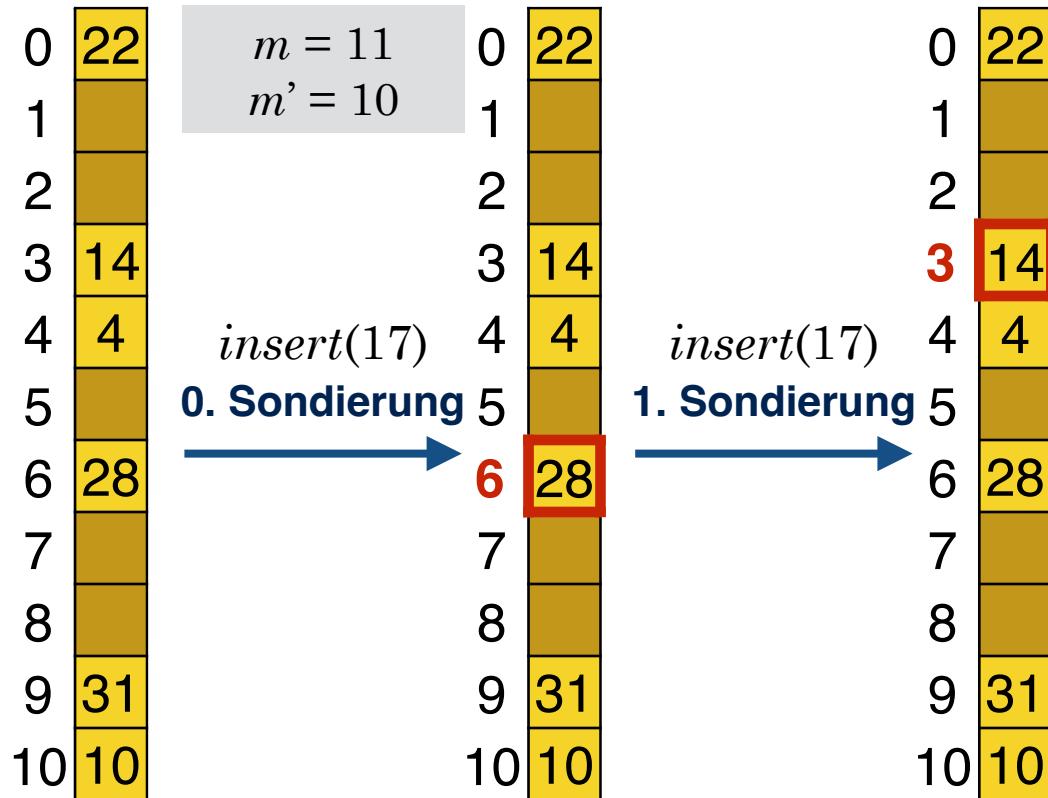
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



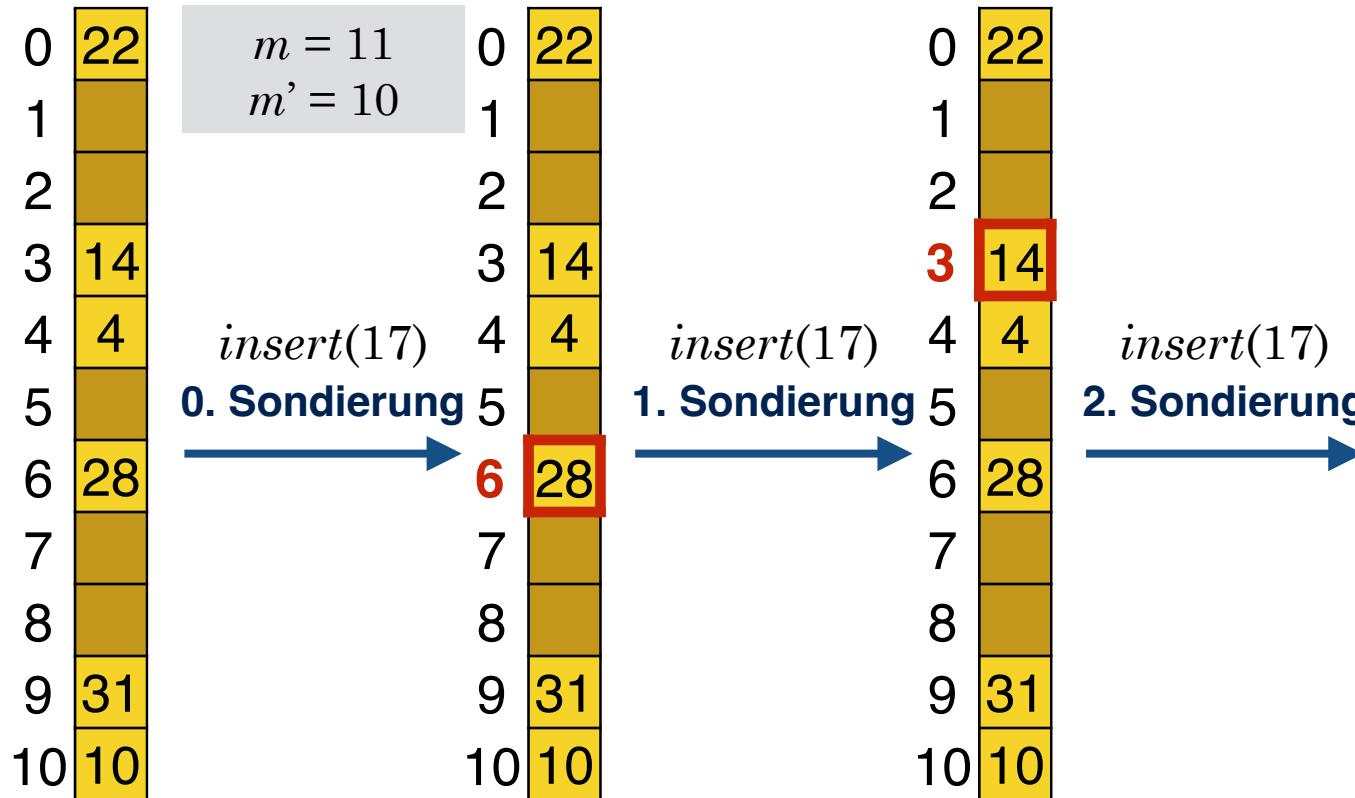
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



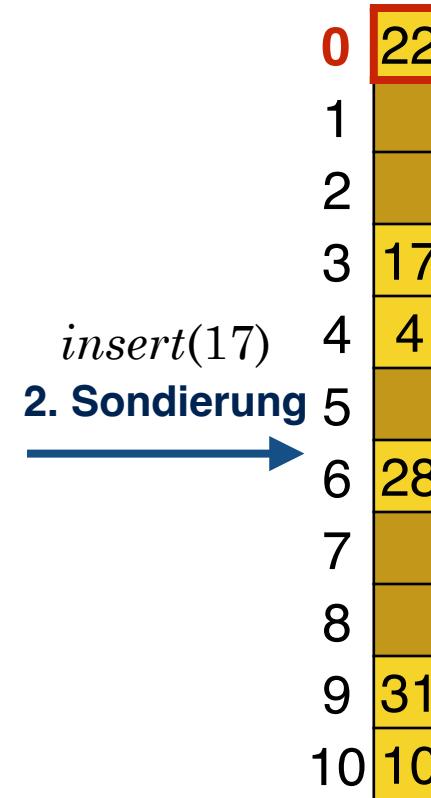
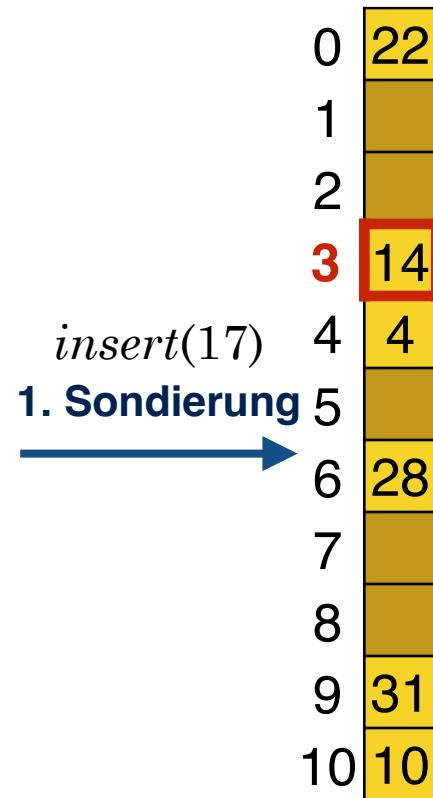
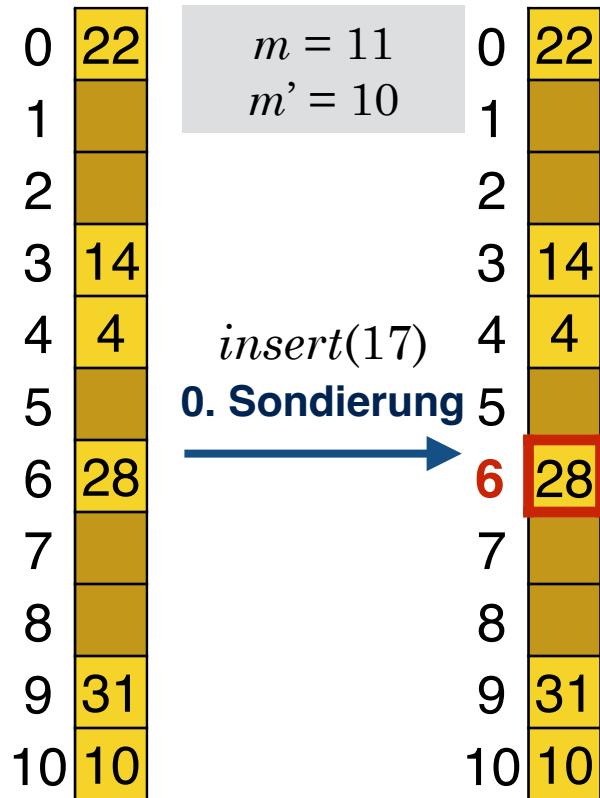
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



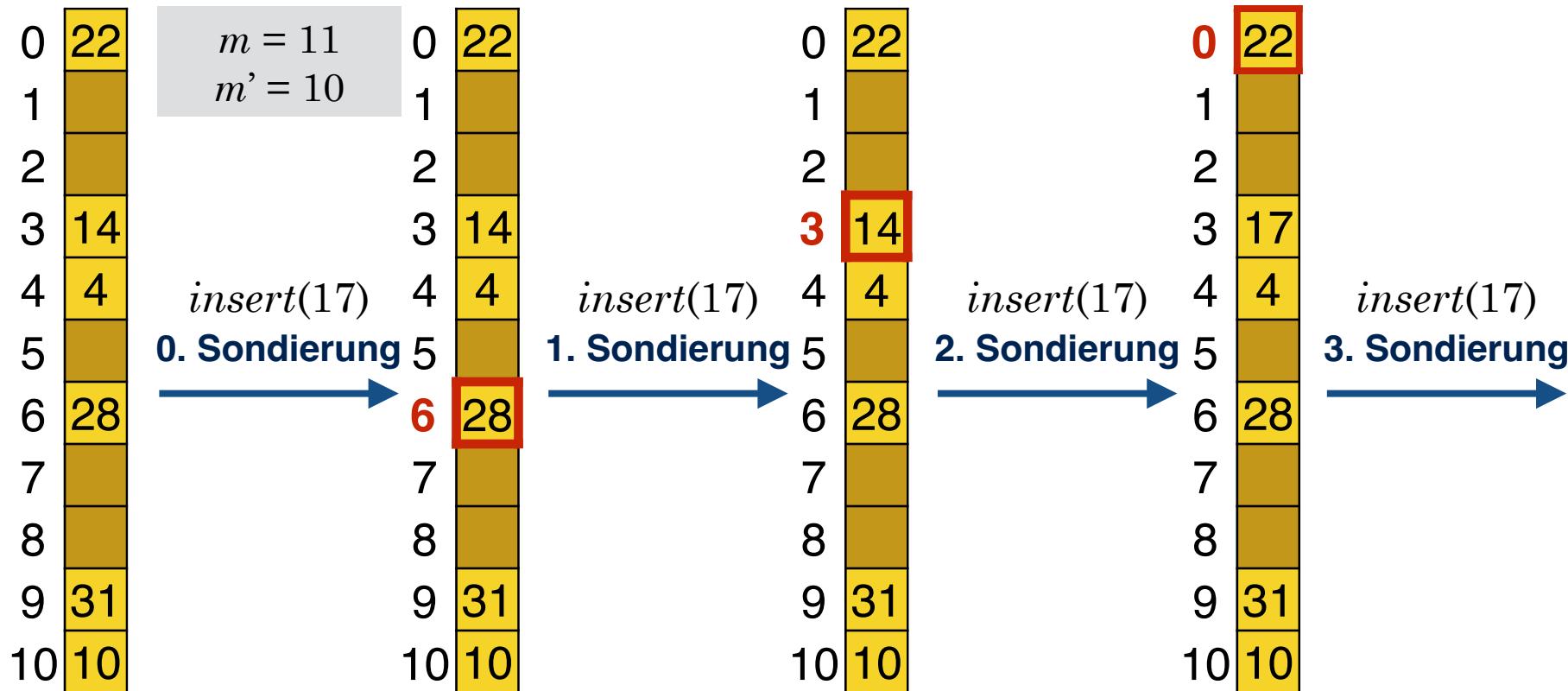
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



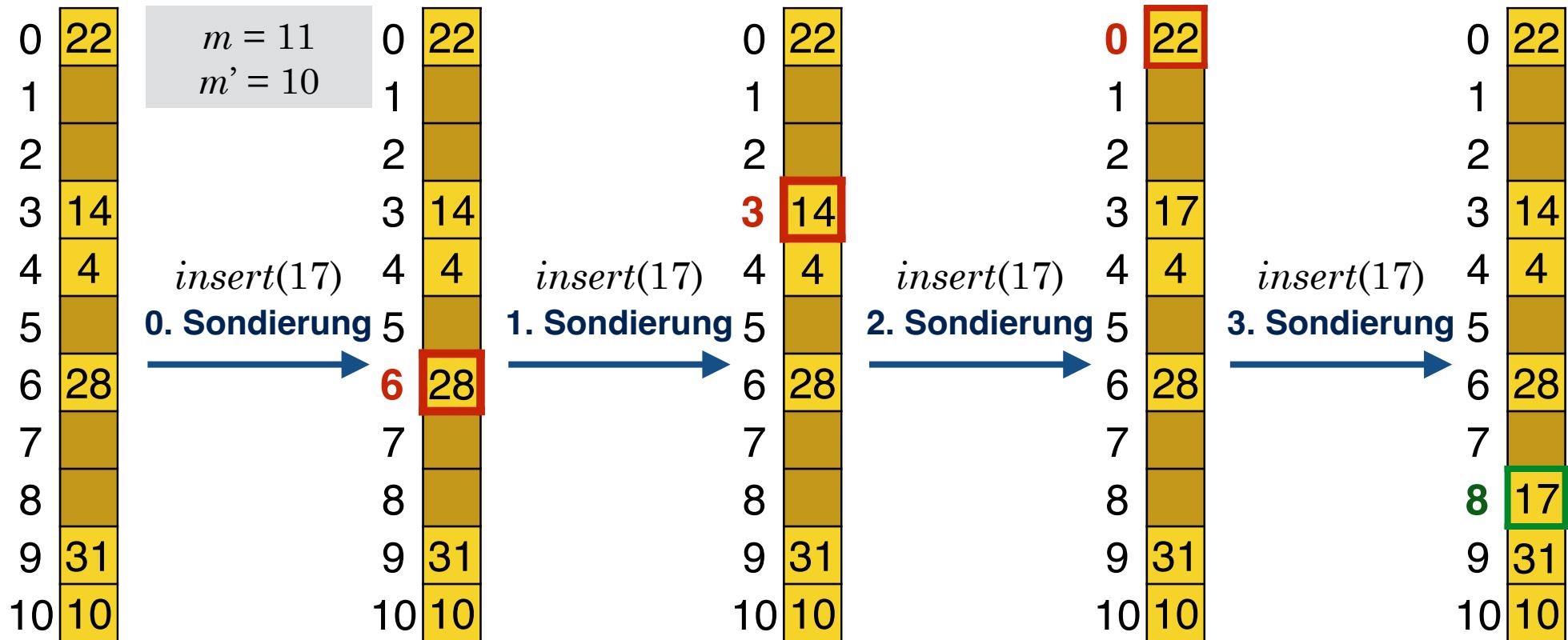
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



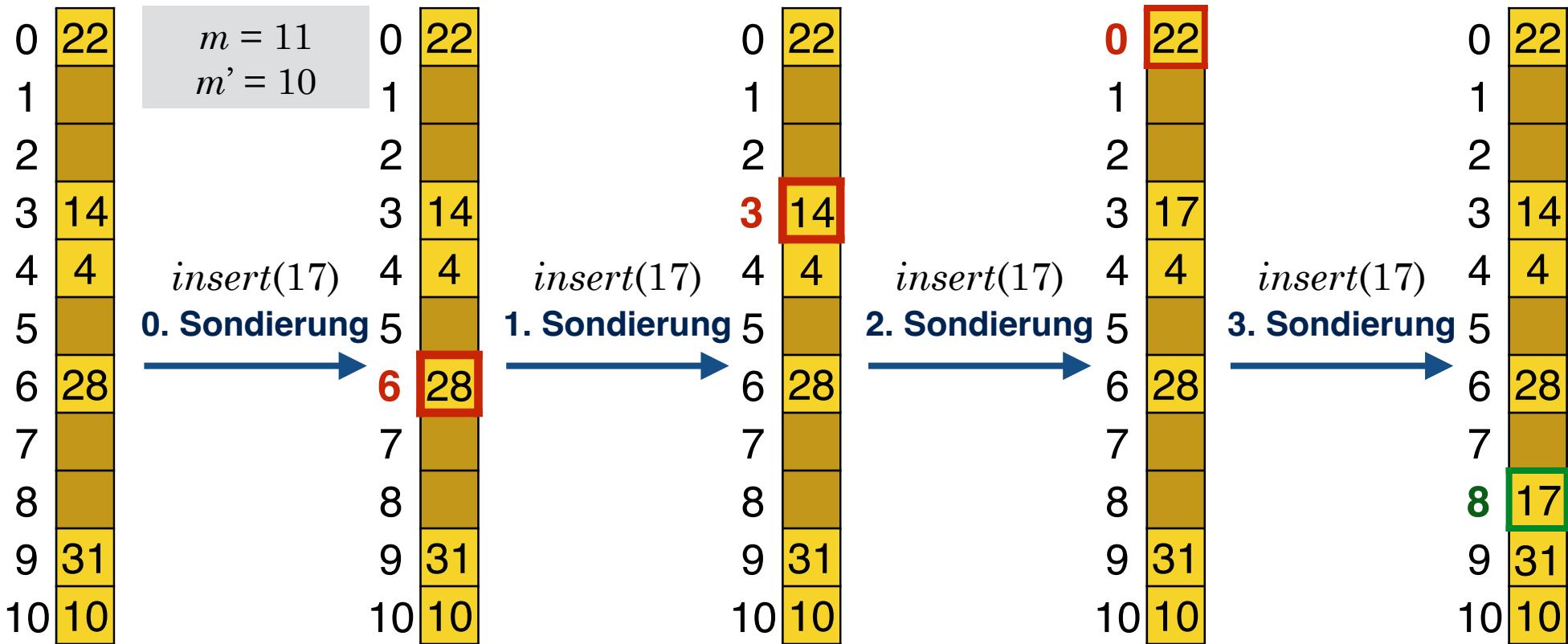
$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

# Doppelhashing

- **Beispiel:** wähle  $m$  als Primzahl  $m'$ , „etwas“ kleiner (z.B.  $m' = m - 1$ )

$$h(x) = x \bmod m$$

$$h'(x) = 1 + (x \bmod m')$$



- **Eigenschaften:**

$$h(x, j) = (h(x) + j \cdot h'(x)) \bmod m \quad \text{mit } 0 \leq j \leq m - 1$$

- ✓ Beste hier vorgestellte Strategie zur Kollisionsvermeidung
- ✓ Kaum unterscheidbar vom idealen Hashing

# Zusammenfassung

- **Anwendung**

- $|U| \gg |S|$
- statische Dictionaries (z.B. Telefonbuch, PLZ-Verzeichnis)

- **Vorteil**

- ✓ im *average case* sehr effizient (oft  $O(1)$ , aber abhängig vom Belegungsfaktor)

- **Nachteile**

- ✗ **Skalierung**: Größe der Hashtabelle muss vorher bekannt sein.  
(Abhilfe: Spiralhashing, lineares Hashing)
- ✗ keine Bereichanfragen, keine Ähnlichkeitsanfragen

- -> Lösung: Suchbäume

# Zusammenfassung

- **Anwendung**

- $|U| \gg |S|$
- statische Dictionaries (z.B. Telefonbuch, PLZ-Verzeichnis)

- **Vorteil**

- ✓ im *average case* sehr effizient (oft  $O(1)$ , aber abhängig vom Belegungsfaktor)

- **Nachteile**

- ✗ **Skalierung**: Größe der Hashtabelle muss vorher bekannt sein.  
(Abhilfe: Spiralhashing, lineares Hashing)
- ✗ keine Bereichanfragen, keine Ähnlichkeitsanfragen

○ -> Lösung: Suchbäume

# V. Suchen in Mengen

## 5. Suchen in Mengen

- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume**
- 5.5. Balancierte Bäume
  - 5.5.1. Gewichtsbilanzierte Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume
- 5.6. *Priority Queue* und *Heap*

# Suchen: Bisher betrachtete Verfahren

- **Felder**

- Suche langsam:  $O(n)$
- Einfügen schnell / Entfernen mit  $O(n)$
- keine Bereichsanfragen

# Suchen: Bisher betrachtete Verfahren

- **Felder**

- Suche langsam:  $O(n)$
- Einfügen schnell / Entfernen mit  $O(n)$
- keine Bereichsanfragen

- **Sortierte Felder**

- Suche schnell:  $O(\lg n)$
- Einfügen / Entfernen mit  $O(n)$
- Bereichsanfragen möglich (z.B. alle Einträge zwischen 120 und 177)

# Suchen: Bisher betrachtete Verfahren

- **Felder**

- Suche langsam:  $O(n)$
- Einfügen schnell / Entfernen mit  $O(n)$
- keine Bereichsanfragen

- **Sortierte Felder**

- Suche schnell:  $O(\lg n)$
- Einfügen / Entfernen mit  $O(n)$
- Bereichsanfragen möglich (z.B. alle Einträge zwischen 120 und 177)

- **Hashing**

- Suche u.U.  $O(1)$ , aber:
  - abhängig von Belegung und
  - eingesetzter Hashfunktion
- Aufwand für Einfügen / Entfernen abhängig von Kollisionsstrategie
- Zahl der Elemente muss vorher bekannt sein
- keine Bereichsanfragen möglich

# Suchbäume:

- **Such-Bäume**

- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\lg n)$  realisierbar
- Bereichssuchen werden unterstützt

# Suchbäume:

- **Such-Bäume**

- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\lg n)$  realisierbar
- Bereichssuchen werden unterstützt

## B Binärer Suchbaum

Ausgangspunkt: **Binäre Suche**

# Suchbäume:

- **Such-Bäume**

- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\log n)$  realisierbar
- Bereichssuchen werden unterstützt

## B Binärer Suchbaum

Ausgangspunkt: **Binäre Suche**

- Start in der **Mitte** –> Wurzel



# Suchbäume:

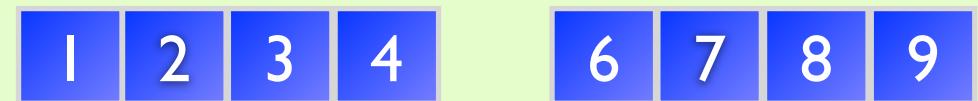
- **Such-Bäume**

- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\log n)$  realisierbar
- Bereichssuchen werden unterstützt

## B Binärer Suchbaum

Ausgangspunkt: **Binäre Suche**

- Start in der **Mitte** –> Wurzel
- Aufteilung in
  - linken Teil
  - rechten Teil



# Suchbäume:

- **Such-Bäume**

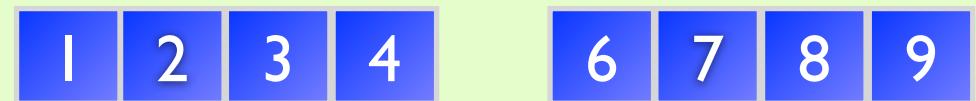
- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\log n)$  realisierbar
- Bereichssuchen werden unterstützt

## B Binärer Suchbaum

Ausgangspunkt: **Binäre Suche**

- Start in der **Mitte** –> Wurzel
- Aufteilung in
  - linken Teil
  - rechten Teil

} jeweils ohne Mitte



# Suchbäume:

- **Such-Bäume**

- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\log n)$  realisierbar
- Bereichssuchen werden unterstützt

## B Binärer Suchbaum

Ausgangspunkt: **Binäre Suche**

- Start in der **Mitte** –> Wurzel
- Aufteilung in
  - linken Teil
  - rechten Teil

} jeweils ohne Mitte
- Rekursiv weiter:
  - linker Teilbaum aus linker Hälfte
  - rechter Teilbaum aus rechter Hälfte



# Suchbäume:

## ● Such-Bäume

- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\log n)$  realisierbar
- Bereichssuchen werden unterstützt

## B Binärer Suchbaum

Ausgangspunkt: **Binäre Suche**

- Start in der **Mitte** –> Wurzel
- Aufteilung in
  - linken Teil
  - rechten Teil

} jeweils ohne Mitte
- Rekursiv weiter:
  - linker Teilbaum aus linker Hälfte
  - rechter Teilbaum aus rechter Hälfte



# Suchbäume:

- **Such-Bäume**

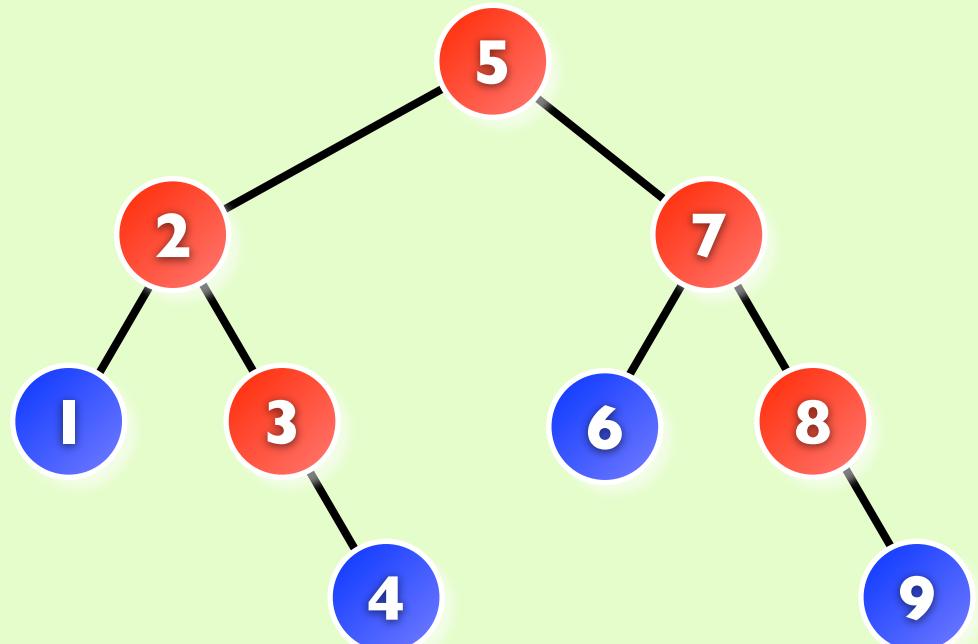
- beliebig dynamisch erweiterbar
- Operationen Einfügen, Löschen und Suchen mit  $O(\lg n)$  realisierbar
- Bereichssuchen werden unterstützt

## B Binärer Suchbaum

Ausgangspunkt: **Binäre Suche**

- Start in der **Mitte** –> Wurzel
- Aufteilung in
  - linken Teil
  - rechten Teil

} jeweils ohne Mitte
- Rekursiv weiter:
  - linker Teilbaum aus linker Hälfte
  - rechter Teilbaum aus rechter Hälfte



# Definition: Binärer Suchbaum

## D Binärer Suchbaum

Ein **binärer Suchbaum** für eine  $n$ -elementige Menge  $S = \{x_1 < x_2 < \dots < x_n\}$  ist ein binärer Baum mit  $n$  Knoten  $\{v_1, \dots, v_n\}$ . Die Knoten sind mit den Elementen von  $S$  beschriftet; d.h. es gibt eine surjektive Abbildung  $Cont : \{v_1, \dots, v_n\} \rightarrow S$ . Die Beschriftung bewahrt die Ordnung; d.h. wenn  $v_l$  im linken Unterbaum ist,  $v_r$  im rechten Unterbaum und  $v_w$  die Wurzel, so gilt:

$$Cont(v_l) < Cont(v_w) < Cont(v_r)$$

# Definition: Binärer Suchbaum

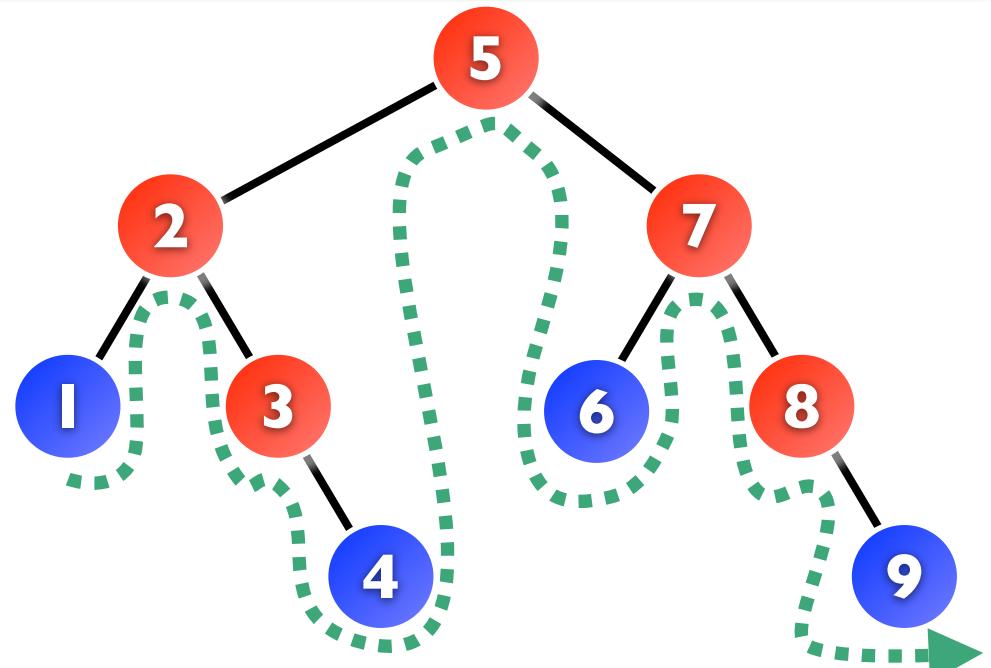
## D Binärer Suchbaum

Ein **binärer Suchbaum** für eine  $n$ -elementige Menge  $S = \{x_1 < x_2 < \dots < x_n\}$  ist ein binärer Baum mit  $n$  Knoten  $\{v_1, \dots, v_n\}$ . Die Knoten sind mit den Elementen von  $S$  beschriftet; d.h. es gibt eine surjektive Abbildung  $Cont : \{v_1, \dots, v_n\} \rightarrow S$ . Die Beschriftung bewahrt die Ordnung; d.h. wenn  $v_l$  im linken Unterbaum ist,  $v_r$  im rechten Unterbaum und  $v_w$  die Wurzel, so gilt:

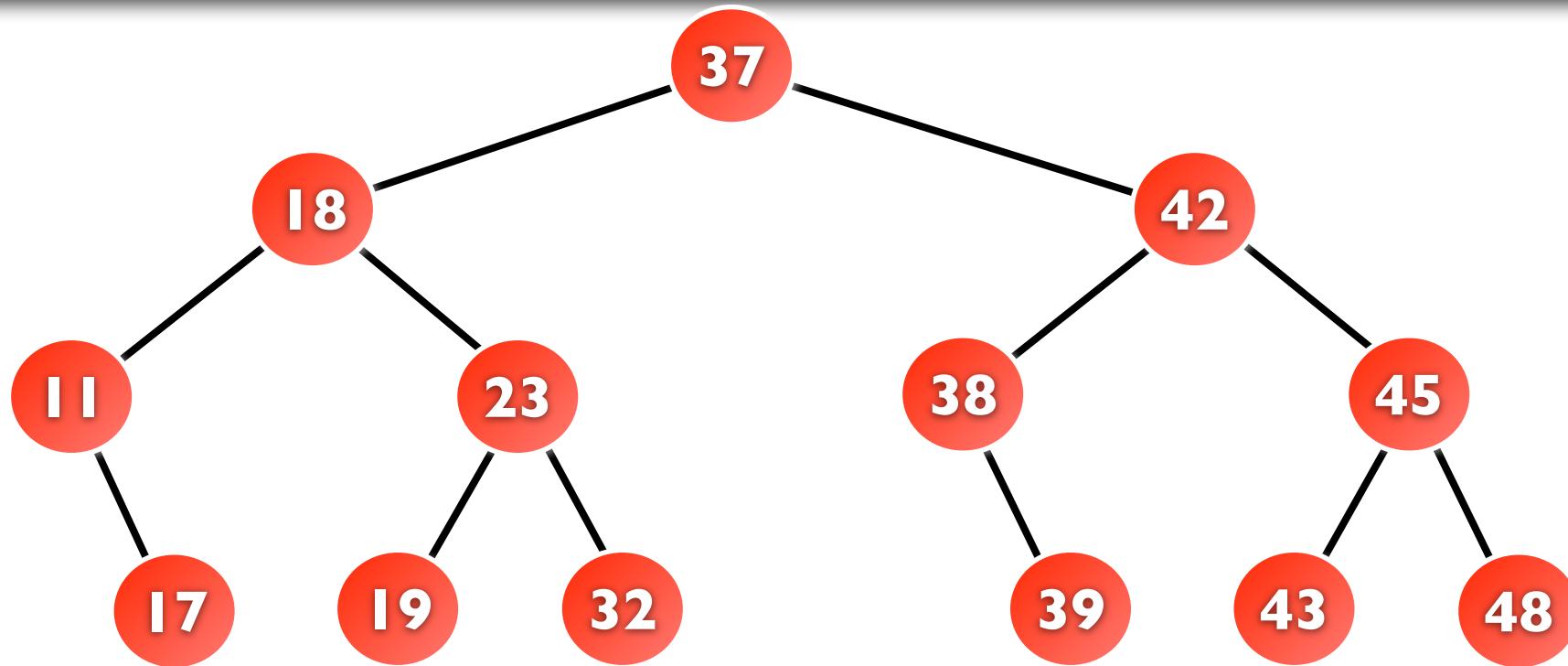
$$Cont(v_l) < Cont(v_w) < Cont(v_r)$$

- **Äquivalente Definition:**

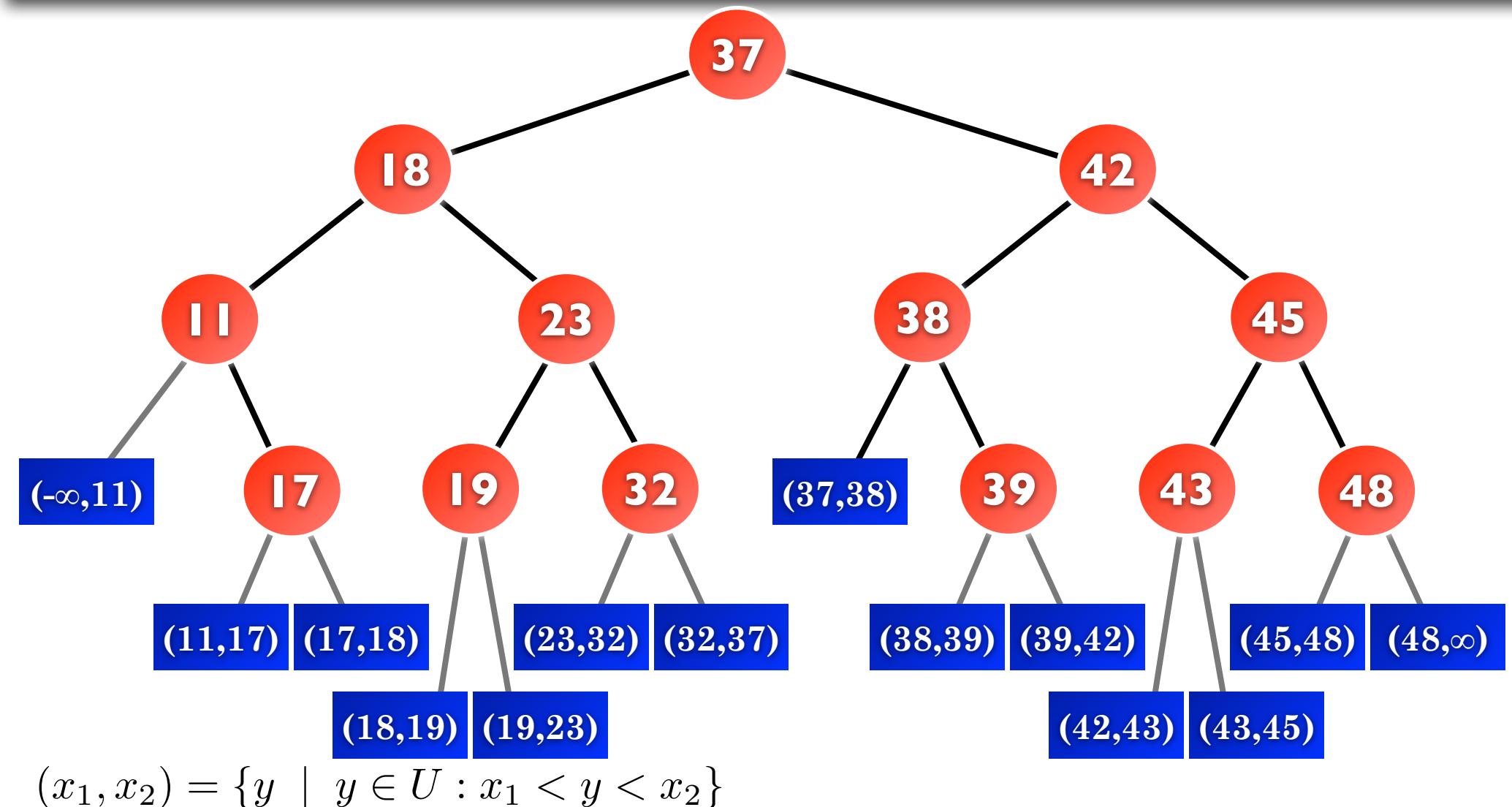
- Ein binärer Suchbaum ist ein Binärbaum, dessen Inorder-Durchlauf die Ordnung auf  $S$  ergibt.



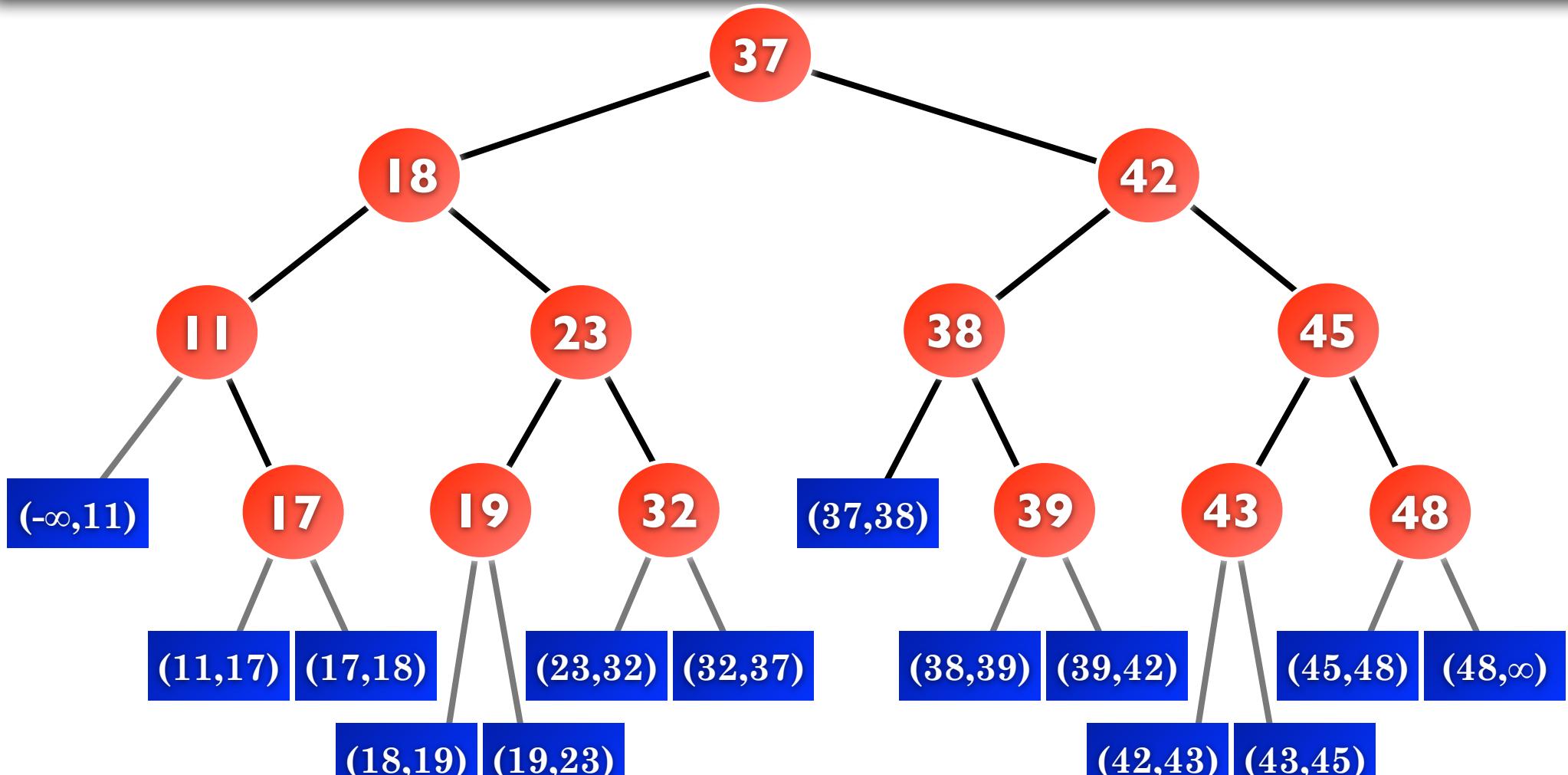
# Binärer Suchbaum mit Bereichsblättern



# Binärer Suchbaum mit Bereichsblättern



# Binärer Suchbaum mit Bereichsblättern



$$(x_1, x_2) = \{y \mid y \in U : x_1 < y < x_2\}$$

Die Blätter des binären Suchbaums stellen die Intervalle von  $U$  dar, die **kein Element von  $S$**  enthalten - erfolglose Suchoperationen enden immer in Blättern.

**Bereichsblätter werden nicht explizit gespeichert!**

# Suchbaum: Implementierung (JAVA)

- **Generische Klasse TreeNode**

```
// Knoten eines Binärbaumes: Schlüssel müssen vergleichbar sein
public class TreeNode<T extends Comparable<T>> {
    public T key;                      // Schlüssel
    public TreeNode<T> left=null;      // linker Teilbaum
    public TreeNode<T> right=null;     // rechter Teilbaum

    public TreeNode(T k) { key=k; }
}
```

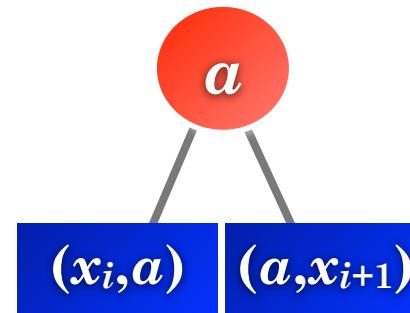
# Suchbaum: Implementierung (JAVA)

- **Generische Klasse SearchTree**

```
public class SearchTree<T extends Comparable<T>> {  
    private TreeNode<T> root=null; // Wurzelknoten  
    public SearchTree() {}  
    // Test, ob v als Knoten im Baum  
    public boolean search(T v) {  
        return search(v,root) != null;  
    }  
    // Suche nach dem Schlüssel im Teilbaum node.  
    // Im Erfolgsfall wird der gefundene Knoten, ansonsten  
    // "null" zurückgegeben.  
    protected TreeNode<T> search(T v,TreeNode<T> node) {  
        if (node==null) return null;  
        int cmp = v.compareTo(node.key);  
        if (cmp==0) return node;  
        else if (cmp<0) return search(v,node.left);  
        else if (cmp>0) return search(v,node.right);  
        else return null;  
    }  
}
```

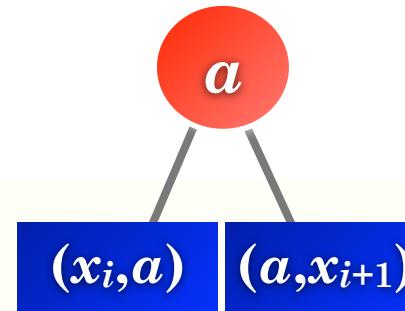
# *insert(a,S)* Einfügen im Binärbaum

- **Idee:** Ersetzte das Blatt  $(x_i, x_{i+1})$  mit dem Intervall in dem  $a$  liegt durch den folgenden Teilbaum



# *insert(a,S)* Einfügen im Binärbaum

- **Idee:** Ersetzte das Blatt  $(x_i, x_{i+1})$  mit dem Intervall in dem  $a$  liegt durch den folgenden Teilbaum



```
public boolean insert(T v) {  
    TreeNode<T> parent=null;  
    TreeNode<T> child=root;  
    while (child!=null) { // suche Wurzel zu Blatt, in dem Suche endet  
        parent=child;  
        int cmp=v.compareTo(child.key);  
        if (cmp== 0) return false; // keine doppelten Einträge!  
        else if (cmp< 0) child=child.left;  
            else child=child.right;  
    }  
    // Ist dies der erste Eintrag?  
    if (parent==null) root=new TreeNode<T>(v);  
    else if (v.compareTo(parent.key)<0) parent.left=new TreeNode<T>(v);  
        else parent.right=new TreeNode<T>(v);  
    return true;  
}
```

# *delete(a,S)*: Löschen im Binärbaum

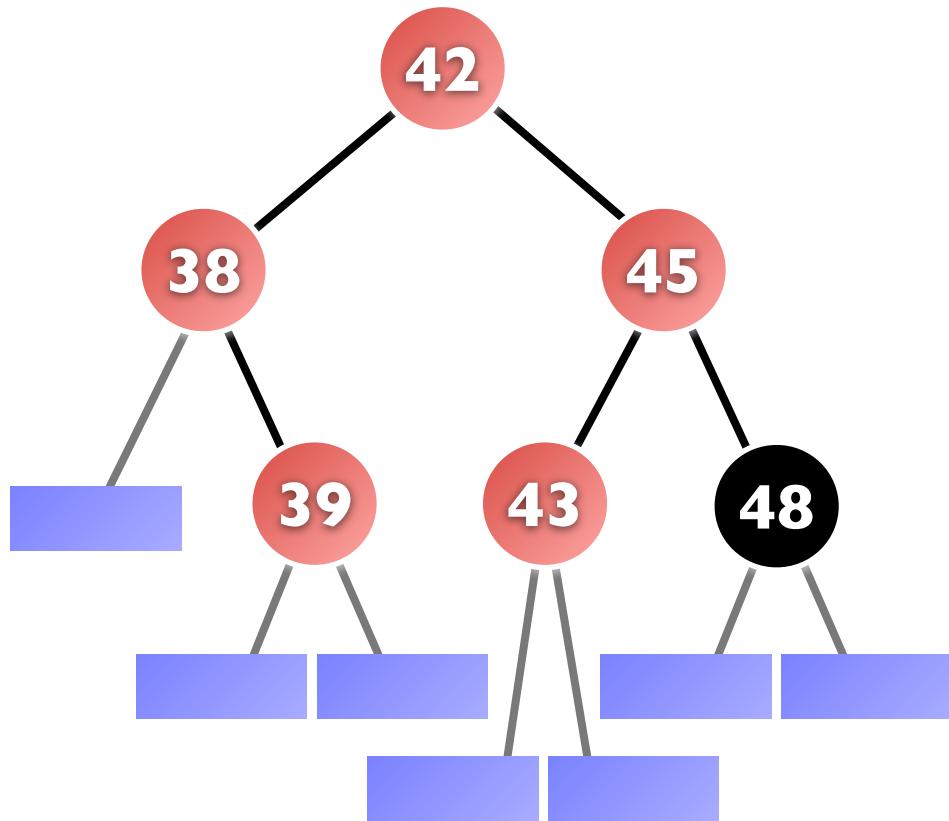
- Drei Fälle sind zu unterscheiden

**1. Der zu löschende Knoten ist ein Blatt**

# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

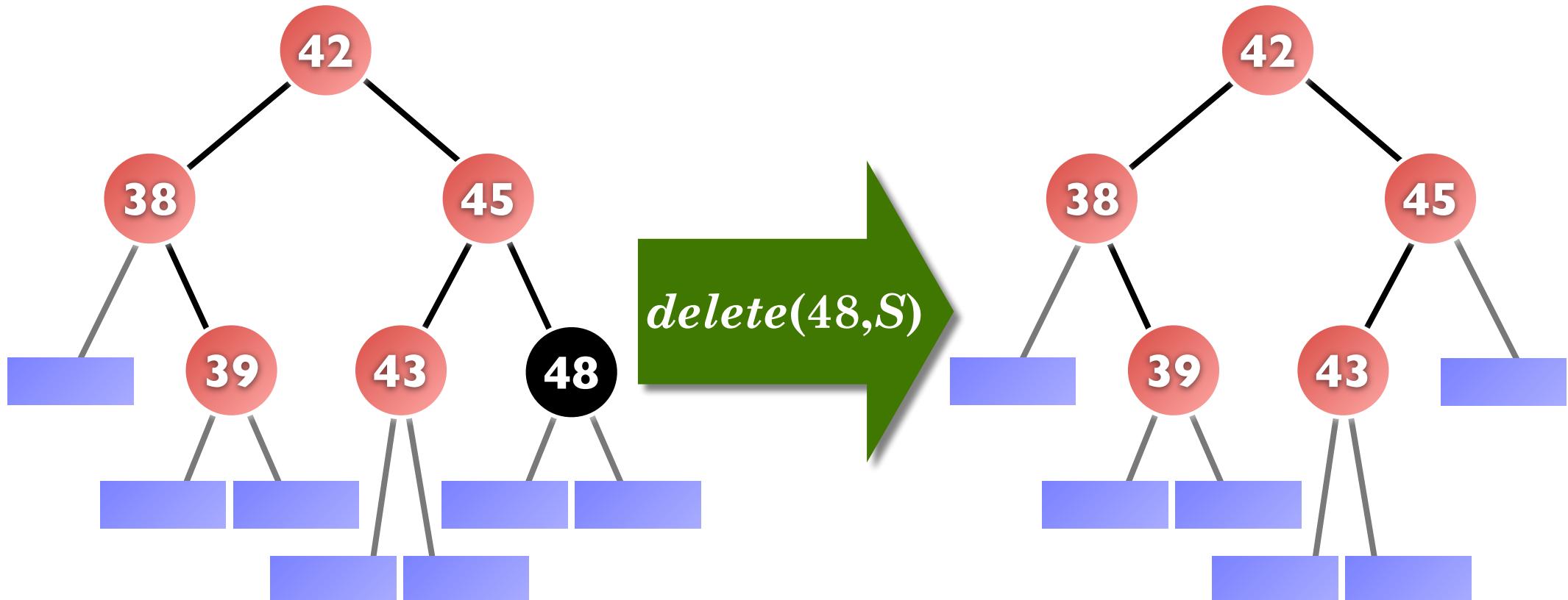
## 1. Der zu löschende Knoten ist ein Blatt



# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

## 1. Der zu löschende Knoten ist ein Blatt

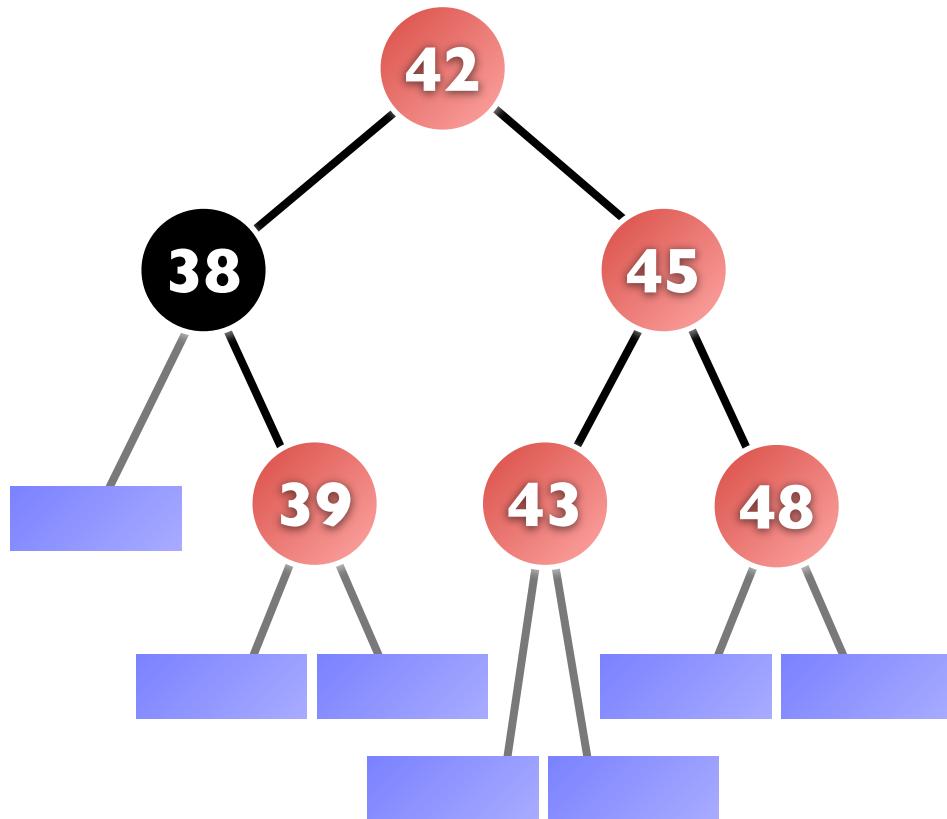


Der Knoten kann problemlos gelöscht werden!

# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

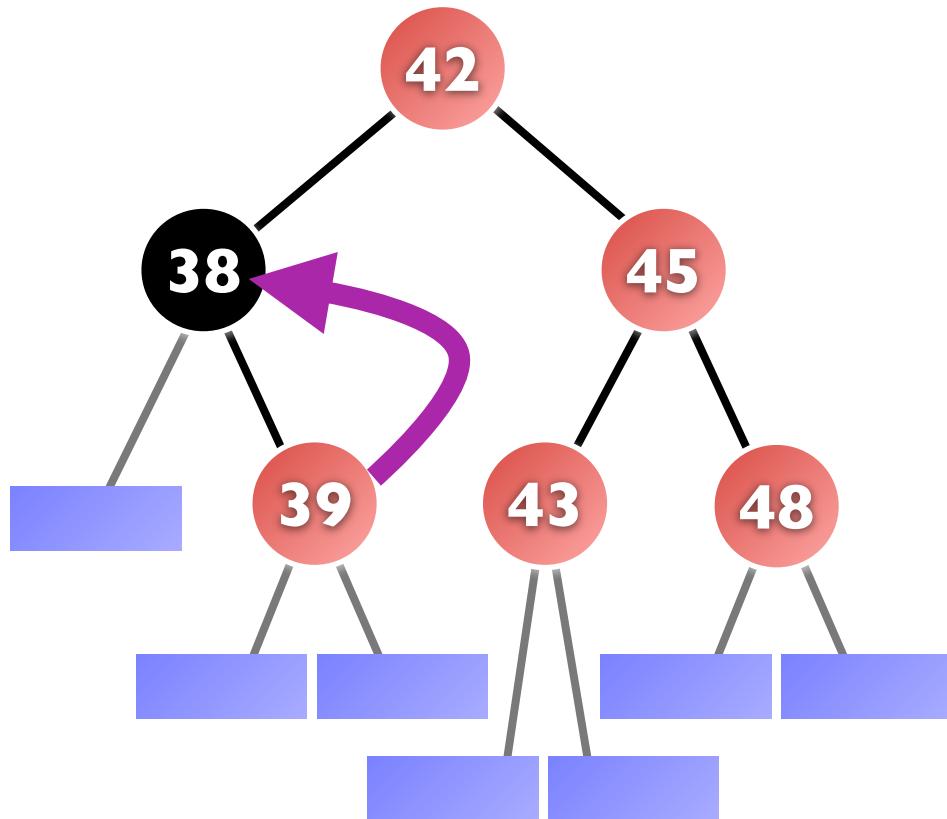
## 2. Der zu löschende Knoten hat *einen* Kindknoten



# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

## 2. Der zu löschende Knoten hat *einen* Kindknoten

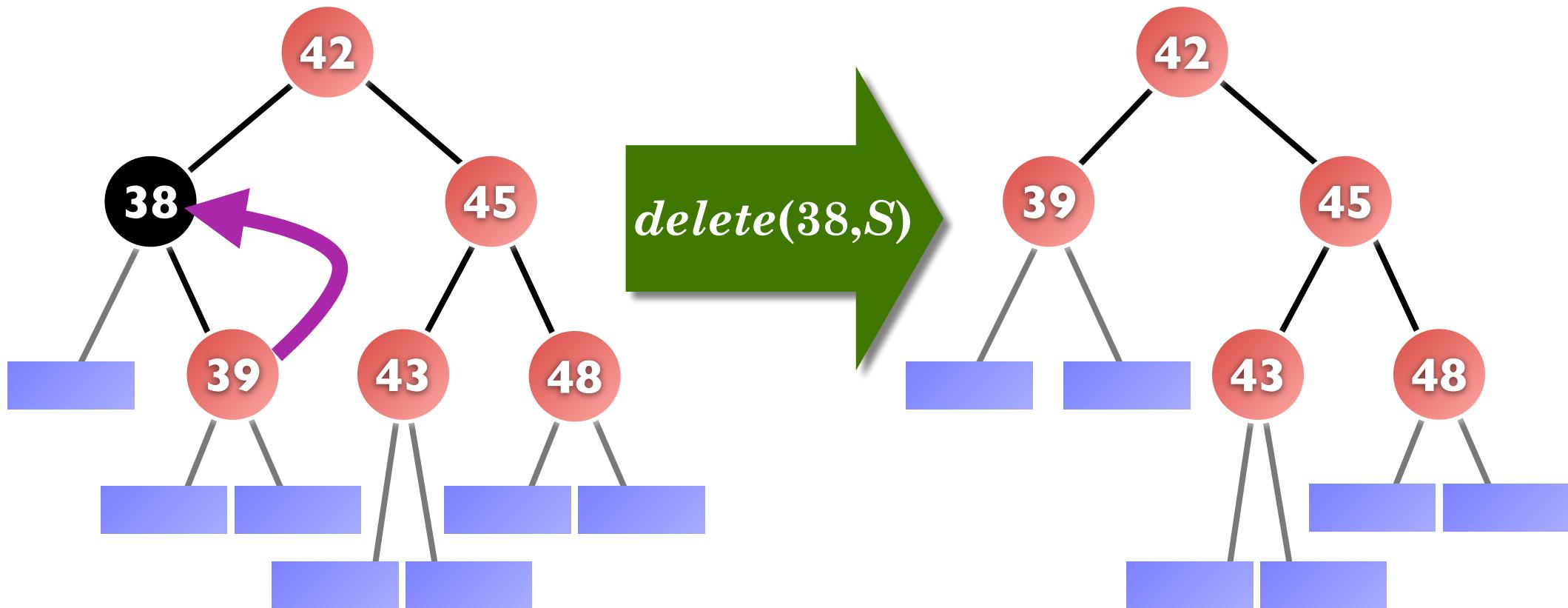


Der Kindknoten rückt an die Stelle des zu löschenen Knoten

# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

## 2. Der zu löschende Knoten hat *einen* Kindknoten

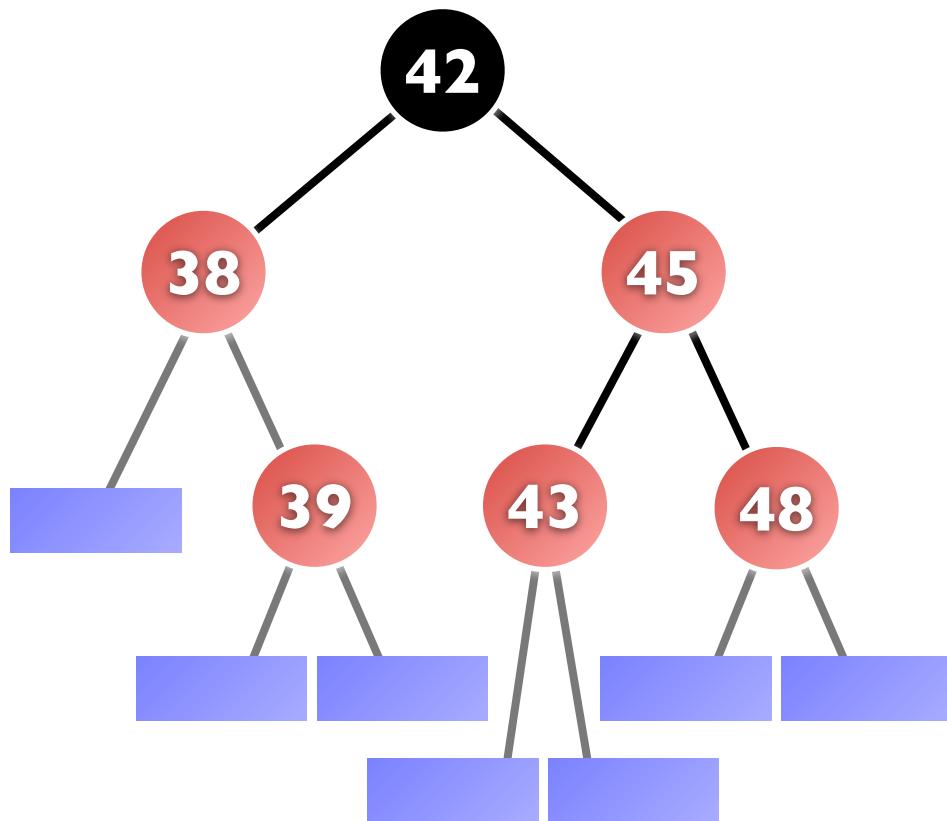


Der Kindknoten rückt an die Stelle des zu löschenen Knoten

# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

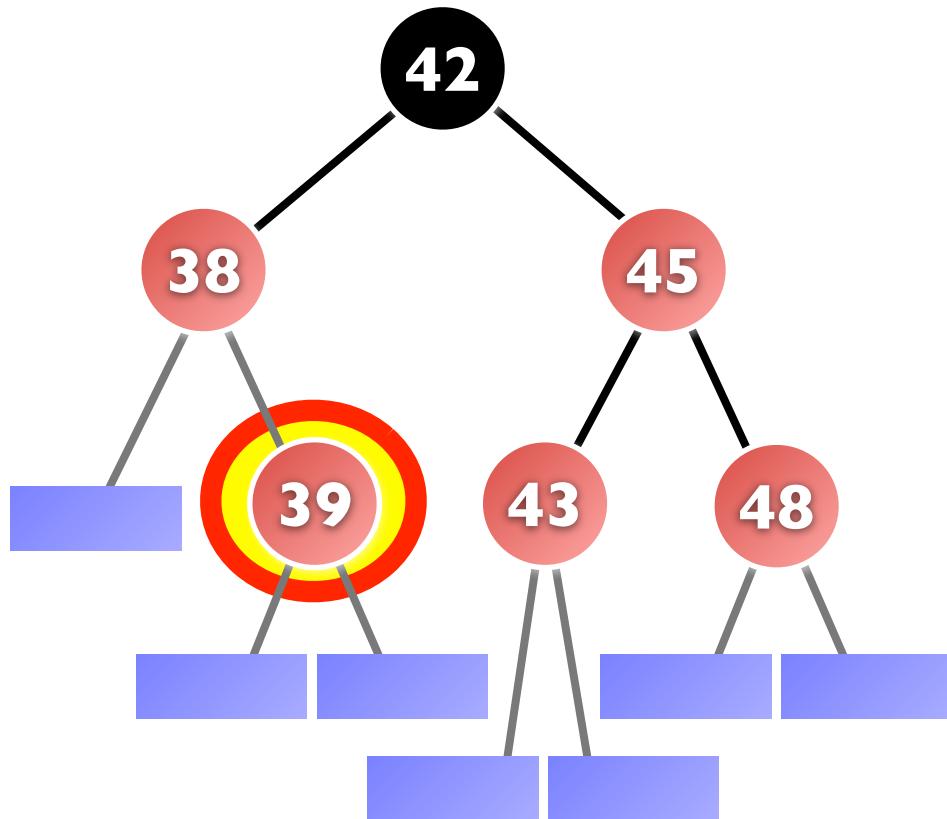
## 3. Der zu löschende Knoten hat **zwei Kindknoten**



# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

## 3. Der zu löschende Knoten hat **zwei Kindknoten**

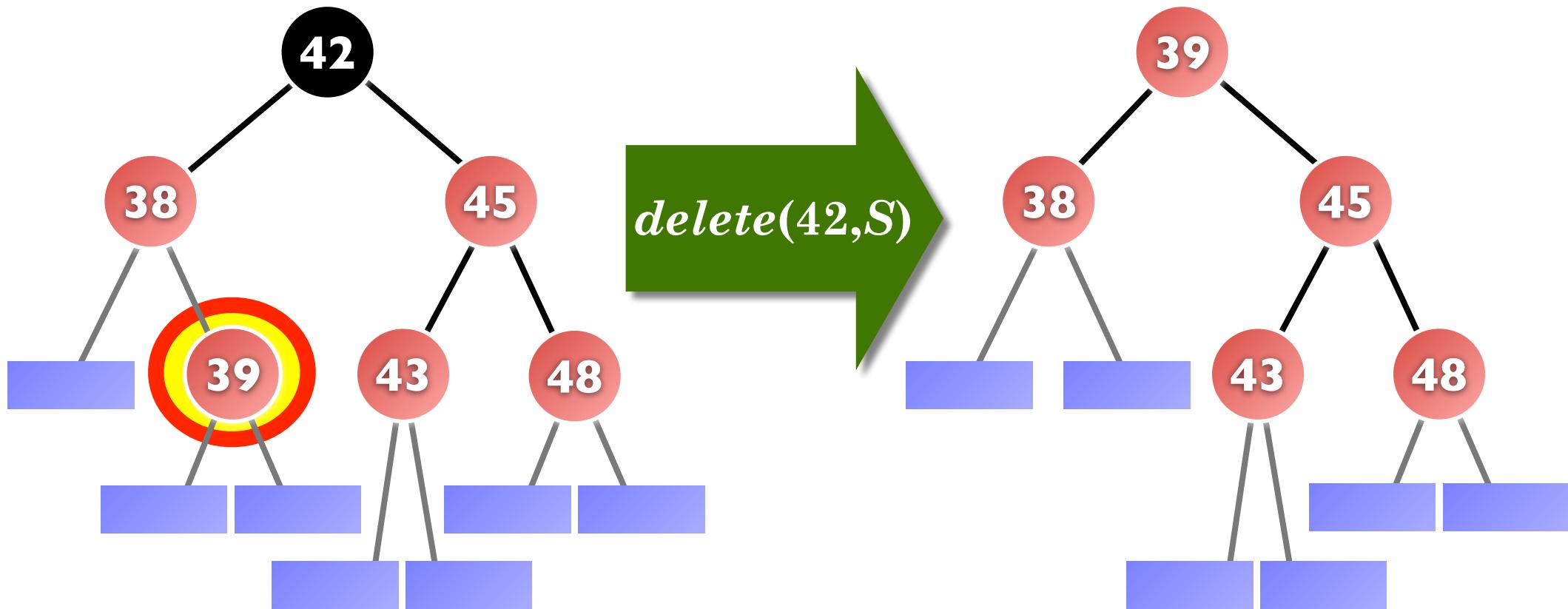


Der Knoten wird durch den größten Knoten seines linken Teilbaumes ersetzt - dieser steht im linken Teilbaum ganz rechts!

# *delete(a,S)*: Löschen im Binärbaum

- Drei Fälle sind zu unterscheiden

## 3. Der zu löschende Knoten hat **zwei Kindknoten**



Der Knoten wird durch den größten Knoten seines linken Teilbaumes ersetzt - dieser steht im linken Teilbaum ganz rechts!

# Komplexitätsanalyse - Aufbau des Suchbaumes I

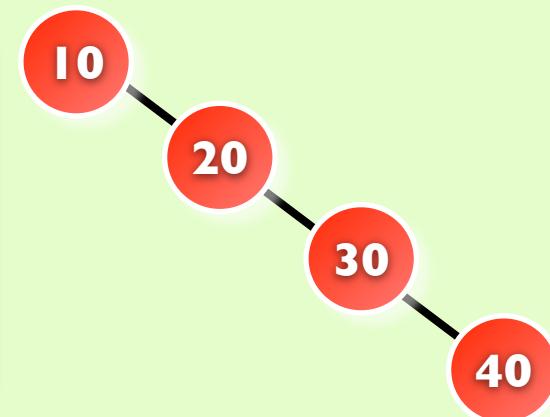
- **Annahme:** Baum wurde durch  $n$  *insert*-Operationen aufgebaut
- $C(n)$ : Zahl der Vergleiche, um Baum mit  $n$  Knoten aufzubauen.
- **worst case:** Der Baum entartet zur linearen Liste;  $n-1$  Vergleiche erforderlich, um  $n$ -tes Element „hinten“ anzufügen:

# Komplexitätsanalyse - Aufbau des Suchbaumes I

- **Annahme:** Baum wurde durch  $n$  *insert*-Operationen aufgebaut
- $C(n)$ : Zahl der Vergleiche, um Baum mit  $n$  Knoten aufzubauen.
- **worst case:** Der Baum entartet zur linearen Liste;  $n-1$  Vergleiche sind erforderlich, um  $n$ -tes Element „hinten“ anzufügen:

## B Zur linearen Liste entarteter Suchbaum

```
SearchTree t=new SearchTree();
t.insert(10); // Elemente werden
t.insert(20); // in aufsteigender
t.insert(30); // Reihenfolge
t.insert(40); // eingefügt
```

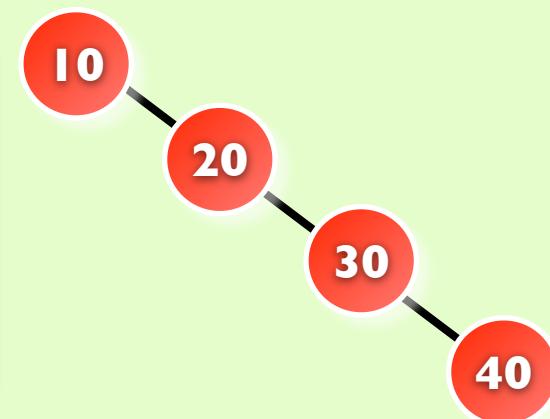


# Komplexitätsanalyse - Aufbau des Suchbaumes I

- **Annahme:** Baum wurde durch  $n$  *insert*-Operationen aufgebaut
- $C(n)$ : Zahl der Vergleiche, um Baum mit  $n$  Knoten aufzubauen.
- **worst case:** Der Baum entartet zur linearen Liste;  $n-1$  Vergleiche sind erforderlich, um  $n$ -tes Element „hinten“ anzufügen:

## B Zur linearen Liste entarteter Suchbaum

```
SearchTree t=new SearchTree();
t.insert(10); // Elemente werden
t.insert(20); // in aufsteigender
t.insert(30); // Reihenfolge
t.insert(40); // eingefügt
```



Die Gesamtzahl Vergleiche:  $C(n) = \sum_{i=1}^n (i - 1) = \sum_{i=0}^{n-1} i = \frac{n(n - 1)}{2}$

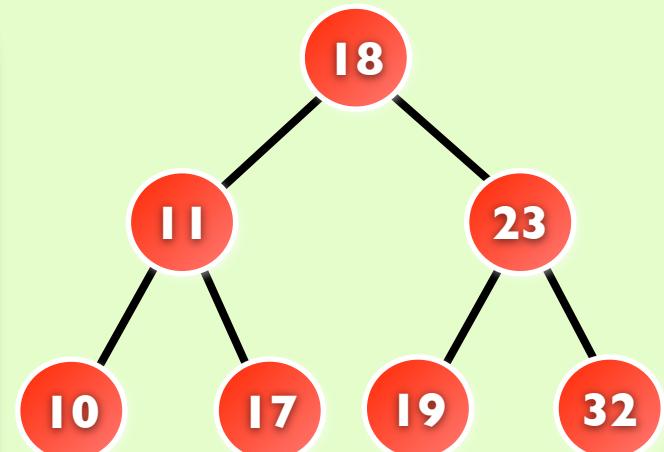
Aufwand:  $\frac{C(n)}{n} = \frac{(n - 1)}{2}$

# Komplexitätsanalyse - Aufbau des Suchbaumes II

- **best case:** Es wird ein Binärbaum minimaler Höhe  $h = \lfloor \lg n \rfloor$  konstruiert, der maximal  $n=2^{h+1}-1$  Knoten enthält.

## B Kostengünstigste Konstruktion eines Binärbaums

```
t.insert(18); // Es wird schritt-  
t.insert(11); // weise ein links-  
t.insert(23); // vollständiger Binär-  
t.insert(10); // baum erzeugt  
t.insert(17);  
t.insert(19);  
t.insert(32);
```



**Erinnerung:** Ein links-vollständiger Binärbaum hat minimale Höhe.

- Uns interessiert hier weniger die Höhe des Baumes (*die abhängig von der Zahl der Kanten ist*), als vielmehr die Zahl der durchlaufenen Knoten (*mit denen wir vergleichen müssen - also die Zahl der Level des Baumes*):
- Ein Baum der Höhe  $h$  hat  $l=h+1$  Level ( $l$  ist die Levelanzahl)  
(*Ein Pfad der Länge  $h$  hat  $h+1$  Knoten*)

# Komplexitätsanalyse - Aufbau des Suchbaumes III

- **best case:** Es wird ein Binärbaum minimaler **Levelzahl**  $l = \lfloor \log_2(n) \rfloor + 1$  konstruiert, der maximal  $n=2^l-1$  Knoten enthält.

# Komplexitätsanalyse - Aufbau des Suchbaumes III

- **best case:** Es wird ein Binärbaum minimaler **Levelzahl**  $l = \lfloor \log_2(n) \rfloor + 1$  konstruiert, der maximal  $n=2^l-1$  Knoten enthält.

## Level i

0

#Knoten je Teilbaum



1

$$\frac{n-1}{2}$$



$$\frac{n-1}{2}$$



2

$$\frac{\frac{n-1}{2}-1}{2}$$



$$\frac{\frac{n-1}{2}-1}{2}$$



$$\frac{\frac{n-1}{2}-1}{2}$$



$$\frac{\frac{n-1}{2}-1}{2}$$



# Komplexitätsanalyse - Aufbau des Suchbaumes III

- **best case:** Es wird ein Binärbaum minimaler **Levelzahl**  $l = \lfloor \log_2(n) \rfloor + 1$  konstruiert, der maximal  $n=2^l-1$  Knoten enthält.

## Level i

0

#Knoten je Teilbaum



1

$$\frac{n-1}{2}$$



$$\frac{n-1}{2}$$

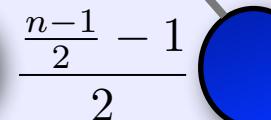


2

$$\frac{\frac{n-1}{2}-1}{2}$$



$$\frac{\frac{n-1}{2}-1}{2}$$



$$\frac{\frac{n-1}{2}-1}{2}$$



$$\frac{\frac{n-1}{2}-1}{2}$$



$n-1$  Vergleiche mit der Wurzel; je  $\frac{n-1}{2}$  Knoten wandern in die Teilbäume.  
Also ergibt sich für die Gesamtzahl der Vergleiche:

$$C(n) = \underbrace{n-1}_{\text{Wurzel}} +$$

$$\underbrace{2 \cdot C\left(\frac{n-1}{2}\right)}_{\text{Vergleiche zum Aufbau rechter und linker Teilbaum}}$$

# Komplexitätsanalyse - Aufbau des Suchbaumes IV

**Alternativer Ansatz** (ohne Rekursionsgleichung):

# Komplexitätsanalyse - Aufbau des Suchbaumes IV

**Alternativer Ansatz** (ohne Rekursionsgleichung):

**Level i**

**#Knoten unter Wurzel**

**0**

#Knoten je Teilbaum

$$2^0 \cdot (n - 1)$$

**1**

$$\frac{n - 1}{2}$$



$$\frac{n - 1}{2}$$



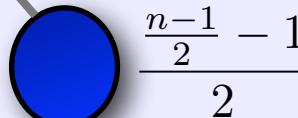
$$2^1 \cdot \left( \frac{n - 1}{2} - 1 \right)$$

**2**

$$\frac{\frac{n-1}{2} - 1}{2}$$



$$\frac{\frac{n-1}{2} - 1}{2}$$



$$\frac{\frac{n-1}{2} - 1}{2}$$



$$2^2 \cdot \left( \frac{\frac{n-1}{2} - 1}{2} - 1 \right)$$

# Komplexitätsanalyse - Aufbau des Suchbaumes IV

**Alternativer Ansatz** (ohne Rekursionsgleichung):

**Level i**

**#Knoten unter Wurzel**

**0**

#Knoten je Teilbaum

$$2^0 \cdot (n - 1)$$

**1**

$$\frac{n - 1}{2}$$

$$\frac{n - 1}{2}$$

$$2^1 \cdot \left( \frac{n - 1}{2} - 1 \right)$$

**2**

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$2^2 \cdot \left( \frac{\frac{n-1}{2} - 1}{2} - 1 \right)$$

**Auf Level Nr.  $i$  ( $0 \leq i \leq l-1$ ):**

# Komplexitätsanalyse - Aufbau des Suchbaumes IV

## Alternativer Ansatz (ohne Rekursionsgleichung):

### Level i

### #Knoten unter Wurzel

0

#Knoten je Teilbaum

$$2^0 \cdot (n - 1)$$

1

$$\frac{n - 1}{2}$$

$$\frac{n - 1}{2}$$

$$2^1 \cdot \left( \frac{n - 1}{2} - 1 \right)$$

2

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$2^2 \cdot \left( \frac{\frac{n-1}{2} - 1}{2} - 1 \right)$$

Auf Level Nr.  $i$  ( $0 \leq i \leq l-1$ ):

- befinden sich  $2^i$  Knoten

# Komplexitätsanalyse - Aufbau des Suchbaumes IV

## Alternativer Ansatz (ohne Rekursionsgleichung):

### Level i

### #Knoten unter Wurzel

0

#Knoten je Teilbaum

$$2^0 \cdot (n - 1)$$

1

$$\frac{n - 1}{2}$$

$$\frac{n - 1}{2}$$

$$2^1 \cdot \left( \frac{n - 1}{2} - 1 \right)$$

2

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$2^2 \cdot \left( \frac{\frac{n-1}{2} - 1}{2} - 1 \right)$$

Auf Level Nr.  $i$  ( $0 \leq i \leq l-1$ ):

- befinden sich  $2^i$  Knoten
- jeder dieser Knoten ist Wurzel eines Teilbaumes mit  $2^{l-i-1}$  Knoten

# Komplexitätsanalyse - Aufbau des Suchbaumes IV

## Alternativer Ansatz (ohne Rekursionsgleichung):

### Level i

### #Knoten unter Wurzel

0

#Knoten je Teilbaum

$$2^0 \cdot (n - 1)$$

1

$$\frac{n - 1}{2}$$

$$\frac{n - 1}{2}$$

$$2^1 \cdot \left( \frac{n - 1}{2} - 1 \right)$$

2

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$2^2 \cdot \left( \frac{\frac{n-1}{2} - 1}{2} - 1 \right)$$

Auf Level Nr.  $i$  ( $0 \leq i \leq l-1$ ):

- befinden sich  $2^i$  Knoten
- jeder dieser Knoten ist Wurzel eines **Teilbaumes mit  $2^{l-i-1}$  Knoten**
- Jeder Knoten des Teilbaums muss mit dem Wurzelknoten verglichen werden (ausser der Wurzel selbst) - also  **$2^{l-i-2}$  Vergleiche pro Wurzel** für Level  $i$

# Komplexitätsanalyse - Aufbau des Suchbaumes IV

## Alternativer Ansatz (ohne Rekursionsgleichung):

### Level i

### #Knoten unter Wurzel

0

#Knoten je Teilbaum

$$2^0 \cdot (n - 1)$$

1

$$\frac{n - 1}{2}$$

$$\frac{n - 1}{2}$$

$$2^1 \cdot \left( \frac{n - 1}{2} - 1 \right)$$

2

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$\frac{\frac{n-1}{2} - 1}{2}$$

$$2^2 \cdot \left( \frac{\frac{n-1}{2} - 1}{2} - 1 \right)$$

Auf Level Nr.  $i$  ( $0 \leq i \leq l-1$ ):

- befinden sich  $2^i$  Knoten
- jeder dieser Knoten ist Wurzel eines **Teilbaumes mit  $2^{l-i}-1$  Knoten**
- Jeder Knoten des Teilbaums muss mit dem Wurzelknoten verglichen werden (ausser der Wurzel selbst) - also  **$2^{l-i}-2$  Vergleiche pro Wurzel** für Level  $i$

Gesamtzahl  
der Vergleiche:

$$C(n) = \sum_{i=0}^{l-2} 2^i \cdot (2^{l-i} - 2)$$

# Komplexitätsanalyse - Aufbau des Suchbaumes V

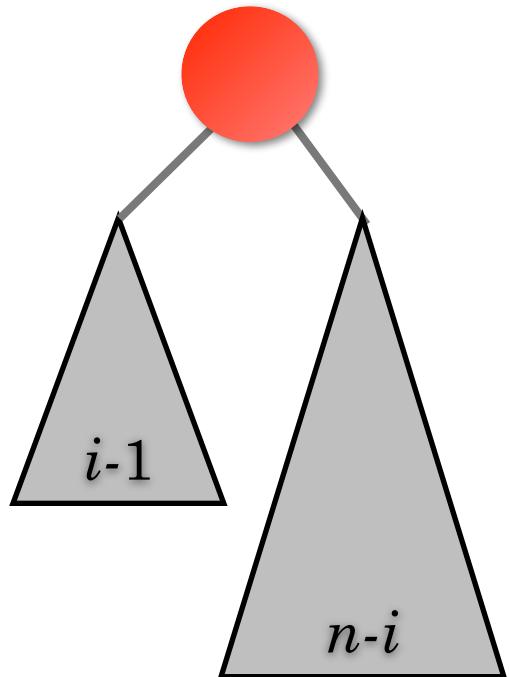
$$\begin{aligned} C(n) &= \sum_{i=0}^{l-2} 2^i \cdot (2^{l-i} - 2) \quad (* \text{ ausmultiplizieren } *) \\ &= \sum_{i=0}^{l-2} 2^l - 2^{i+1} \quad (* \text{ Summe aufspalten } *) \\ &= \sum_{i=0}^{l-2} 2^l - \sum_{i=0}^{l-2} 2^{i+1} \quad (* \text{ Erste Summe ausrechnen / Zweite verkürzen } *) \\ &= (l-1) \cdot 2^l - \left[ \sum_{i=0}^{l-1} 2^i - 1 \right] \quad (* \text{ Geometrische Reihe ausrechnen } *) \\ &= (l-1) \cdot 2^l - \left[ \frac{2^{l-1} - 1}{2 - 1} - 1 \right] \\ &= (l-1) \cdot 2^l - [2^{l-1} - 2] \\ &= (l-1) \cdot 2^l - \frac{1}{2} \cdot 2^l + 2 \\ &= (l-1,5) \cdot 2^l + 2l \quad (* n = 2^l - 1 \Leftrightarrow \text{ld}(n+1) = l *) \\ &= [\text{ld } (n+1) - 1,5] \cdot 2^l + 2 \quad (* n = 2^l - 1 \Leftrightarrow 2^l = n+1 *) \\ &= [\text{ld } (n+1) - 1,5] \cdot (n+1) + 2 \\ &= (n+1) \cdot \text{ld } (n+1) - 1.5n + 0.5 \end{aligned}$$

$$\sum_{i=a}^b c^i = \frac{c^b - c^a}{c - 1}$$

Aufwand:  $\frac{C(n)}{n} \approx \text{ld } n$

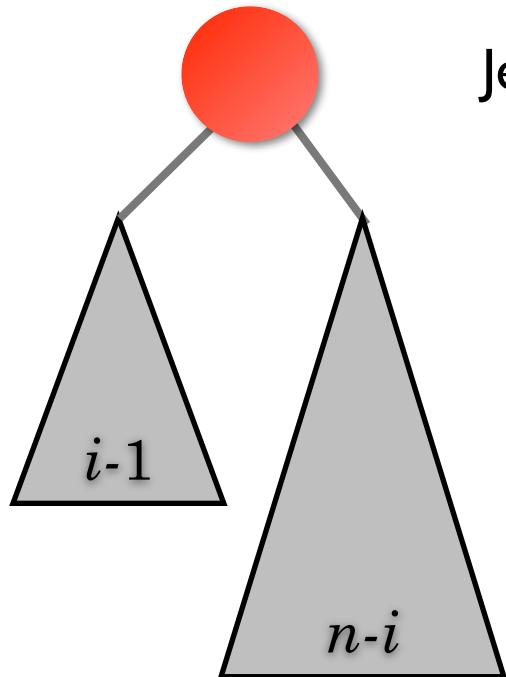
# Komplexitätsanalyse - Aufbau des Suchbaumes VI

- **average case:** von  $n$  Elementen werden  $i-1$  im linken Teilbaum eingetragen und der Rest von der Wurzel im rechten



# Komplexitätsanalyse - Aufbau des Suchbaumes VI

- **average case:** von  $n$  Elementen werden  $i-1$  im linken Teilbaum eingetragen und der Rest von der Wurzel im rechten

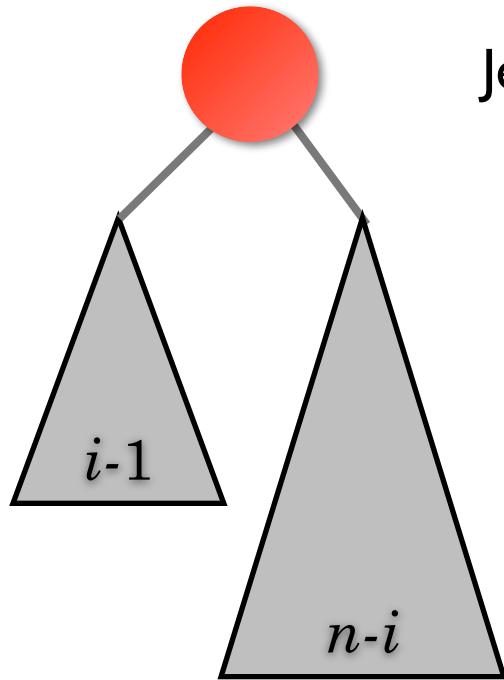


Jede Aufteilung  $i=1,\dots,n$  ist gleich wahrscheinlich

$$\begin{aligned} C(n) &= n - 1 && \text{alle müssen an der Wurzel vorbei} \\ &+ C(i - 1) && \text{Vergleiche im linken Teilbaum} \\ &+ C(n - i) && \text{Vergleiche im rechten Teilbaum} \end{aligned}$$

# Komplexitätsanalyse - Aufbau des Suchbaumes VI

- **average case:** von  $n$  Elementen werden  $i-1$  im linken Teilbaum eingetragen und der Rest von der Wurzel im rechten



Jede Aufteilung  $i=1,\dots,n$  ist gleich wahrscheinlich

$$\begin{aligned} C(n) &= n - 1 && \text{alle müssen an der Wurzel vorbei} \\ &+ C(i - 1) && \text{Vergleiche im linken Teilbaum} \\ &+ C(n - i) && \text{Vergleiche im rechten Teilbaum} \end{aligned}$$

$$\begin{aligned} C(n) &= \frac{1}{n} \sum_{i=1}^n [n - 1 + C(i - 1) + C(n - i)] \\ &= n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C(i) && \text{(wie QuickSort!)} \\ &= 2n \cdot \ln n + \dots \\ &= \dots \\ &= 1.386 \cdot n \operatorname{ld} n + \dots \end{aligned}$$

Aufwand:

$$\frac{C(n)}{n} \approx 1.386 \operatorname{ld} n$$

# Komplexität der Operationen

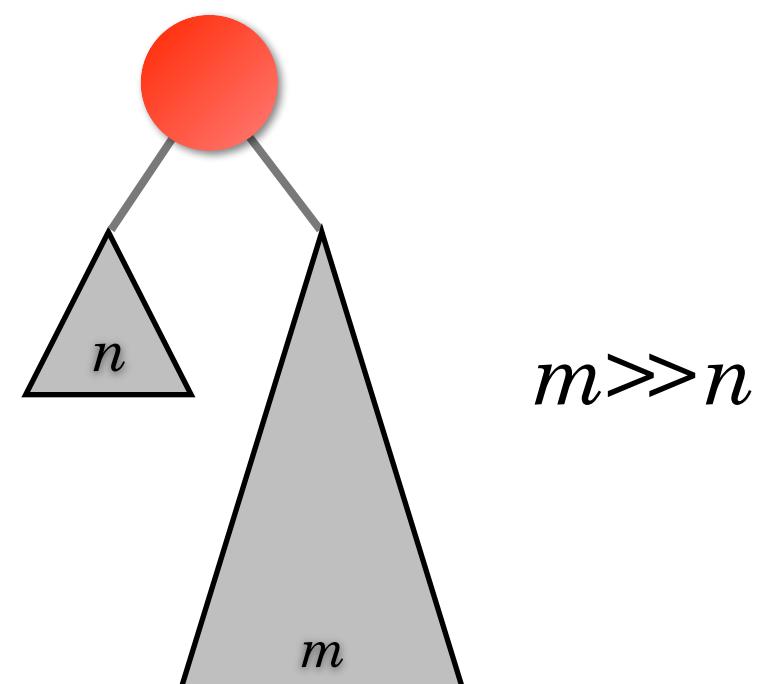
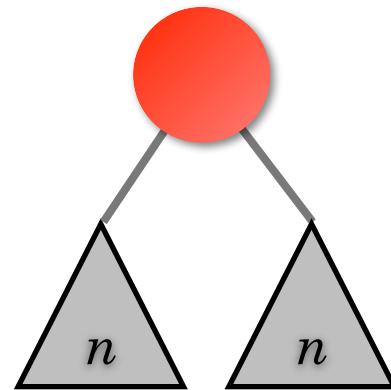
- **Komplexität für *search*, *insert* und *delete*:**
  - im wesentlichen: Baum traversieren und einige lokale Änderungen mit  $O(1)$
  - daher insgesamt:  $O(h(t))$ , wobei  $h(t)$  die Höhe des Baumes ist

# Komplexität der Operationen

- **Komplexität für *search*, *insert* und *delete*:**
  - im wesentlichen: Baum traversieren und einige lokale Änderungen mit  $O(1)$
  - daher insgesamt:  $O(h(t))$ , wobei  $h(t)$  die Höhe des Baumes ist
- **Also: Komplexität hängt von Höhe des Baumes ab!**

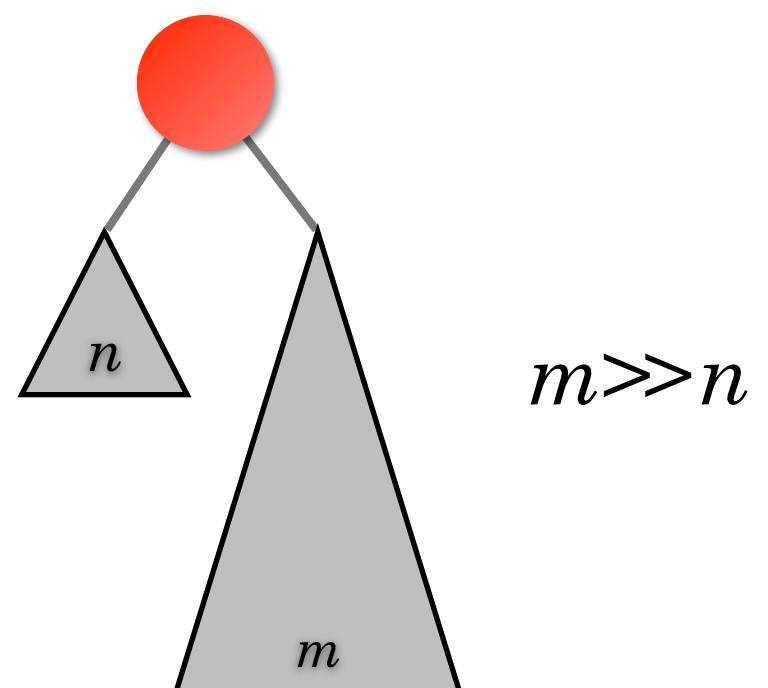
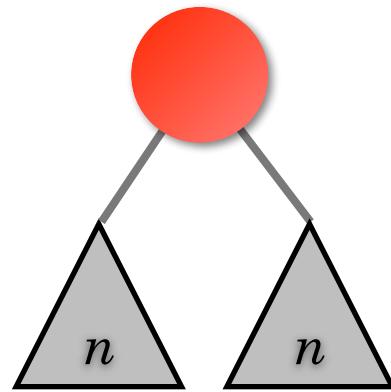
# Komplexität der Operationen

- **Komplexität für *search*, *insert* und *delete*:**
  - im wesentlichen: Baum traversieren und einige lokale Änderungen mit  $O(1)$
  - daher insgesamt:  $O(h(t))$ , wobei  $h(t)$  die Höhe des Baumes ist
- **Also: Komplexität hängt von Höhe des Baumes ab!**
- **Problem:** Baum kann nach vielen Einfüge- und Löschoperationen entarten:



# Komplexität der Operationen

- **Komplexität für *search*, *insert* und *delete*:**
  - im wesentlichen: Baum traversieren und einige lokale Änderungen mit  $O(1)$
  - daher insgesamt:  $O(h(t))$ , wobei  $h(t)$  die Höhe des Baumes ist
- **Also: Komplexität hängt von Höhe des Baumes ab!**
- **Problem:** Baum kann nach vielen Einfüge- und Löschoperationen entarten:



**Ziel:** gestalte Einfüge- und Löschoperationen so, dass Höhe der Teilbäume in etwa gleich bleibt.

# V. Suchen in Mengen

## 5. Suchen in Mengen

- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume**
  - 5.5.1. Gewichtsbalance Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume
- 5.6. *Priority Queue* und *Heap*

# Motivation

- **Problem:** Baum kann nach vielen Einfüge- und Löschoperationen entarten - im Extremfall zur linearen Liste.  
Dann: Komplexität der Suche immer näher an  $O(n)$

# Motivation

- **Problem:** Baum kann nach vielen Einfüge- und Löschoperationen entarten - im Extremfall zur linearen Liste.  
Dann: Komplexität der Suche immer näher an  $O(n)$
- **Abhilfe:** **Balancierte Bäume**, die garantierte Komplexität  $O(\lg n)$  haben.

# Motivation

- **Problem:** Baum kann nach vielen Einfüge- und Löschoperationen entarten - im Extremfall zur linearen Liste.  
Dann: Komplexität der Suche immer näher an  $O(n)$
- **Abhilfe:** **Balancierte Bäume**, die garantierte Komplexität  $O(\lg n)$  haben.
- Zwei **Balance-Kriterien** werden unterschieden:
  - **Gewichtsbalanceierte Bäume:**
    - Halte **Anzahl der Blätter** in Teilbäumen möglichst gleich  
(werden hier nur am Rande betrachtet)
  - **Höhenbalanceierte Bäume:**
    - Halte den **Höhenunterschied** der Teilbäume möglichst gering

# V. Suchen in Mengen

---

## 5. Suchen in Mengen

5.1. Einführung

5.2. Einfache Implementierungen

5.3. *Hashing*

5.4. Binäre Suchbäume

### 5.5. Balancierte Bäume

#### 5.5.1. Gewichtsbalanceierte Bäume

5.5.2. AVL-Bäume

5.5.3. (a,b)-Bäume

5.5.4. rot/schwarz-Bäume

5.6. *Priority Queue* und *Heap*

# Gewichtsbalancierte Bäume

## D Wurzelbalance / Gewichtsbalancierter Baum

Sei  $T$  ein Binärbaum mit rechtem Teilbaum  $T_r$  und linkem Teilbaum  $T_l$ . Dann ist

$$\rho(T) = \frac{\text{weight}(T_l)}{\text{weight}(T)} = 1 - \frac{\text{weight}(T_r)}{\text{weight}(T)}$$

die **Wurzelbalance** (oder kurz die Balance) von  $T$ . Dabei steht  $\text{weight}(T)$  für die Anzahl der Bereichsblätter von  $T$ . Ein **Baum**  $T$  ist **von beschränkter Balance  $\alpha$** , wenn für jeden Unterbaum  $T'$  von  $T$  gilt:

$$\alpha \leq \rho(T') \leq 1 - \alpha$$

Die Menge aller Bäume von beschränkter Balance  $\alpha$  heißt **BB( $\alpha$ )** ( $BB \equiv Bounded\ Balance$ ). Bäume von beschränkter Balance nennt man auch **Gewichtsbalancierte Bäume**.

# Gewichtsbalancierte Bäume

D

## Wurzelbalance / Gewichtsbalancierter Baum

Sei  $T$  ein Binärbaum mit rechtem Teilbaum  $T_r$  und linkem Teilbaum  $T_l$ . Dann ist

$$\rho(T) = \frac{\text{weight}(T_l)}{\text{weight}(T)} = 1 - \frac{\text{weight}(T_r)}{\text{weight}(T)}$$

die **Wurzelbalance** (oder kurz die Balance) von  $T$ . Dabei steht  $\text{weight}(T)$  für die Anzahl der Bereichsblätter von  $T$ . Ein **Baum**  $T$  ist **von beschränkter Balance  $\alpha$** , wenn für jeden Unterbaum  $T'$  von  $T$  gilt:

$$\alpha \leq \rho(T') \leq 1 - \alpha$$

Die Menge aller Bäume von beschränkter Balance  $\alpha$  heißt **BB( $\alpha$ )** ( $BB \equiv Bounded\ Balance$ ). Bäume von beschränkter Balance nennt man auch **Gewichtsbalancierte Bäume**.

- **Achtung:**
  - **Bereichsblätter!** Blätter werden daher nicht als Teilbäume betrachtet!
    - Zur Erinnerung: Bereichsblätter werden nicht explizit gespeichert
    - Nur inneren Knoten und Wurzel wird eine Balance zugeordnet
    - bei Höhenberechnung werden Blätter **nicht** berücksichtigt, wohl aber bei der Gewichtsberechnung!

# Beispiel: Gewichtsbalancierter Baum

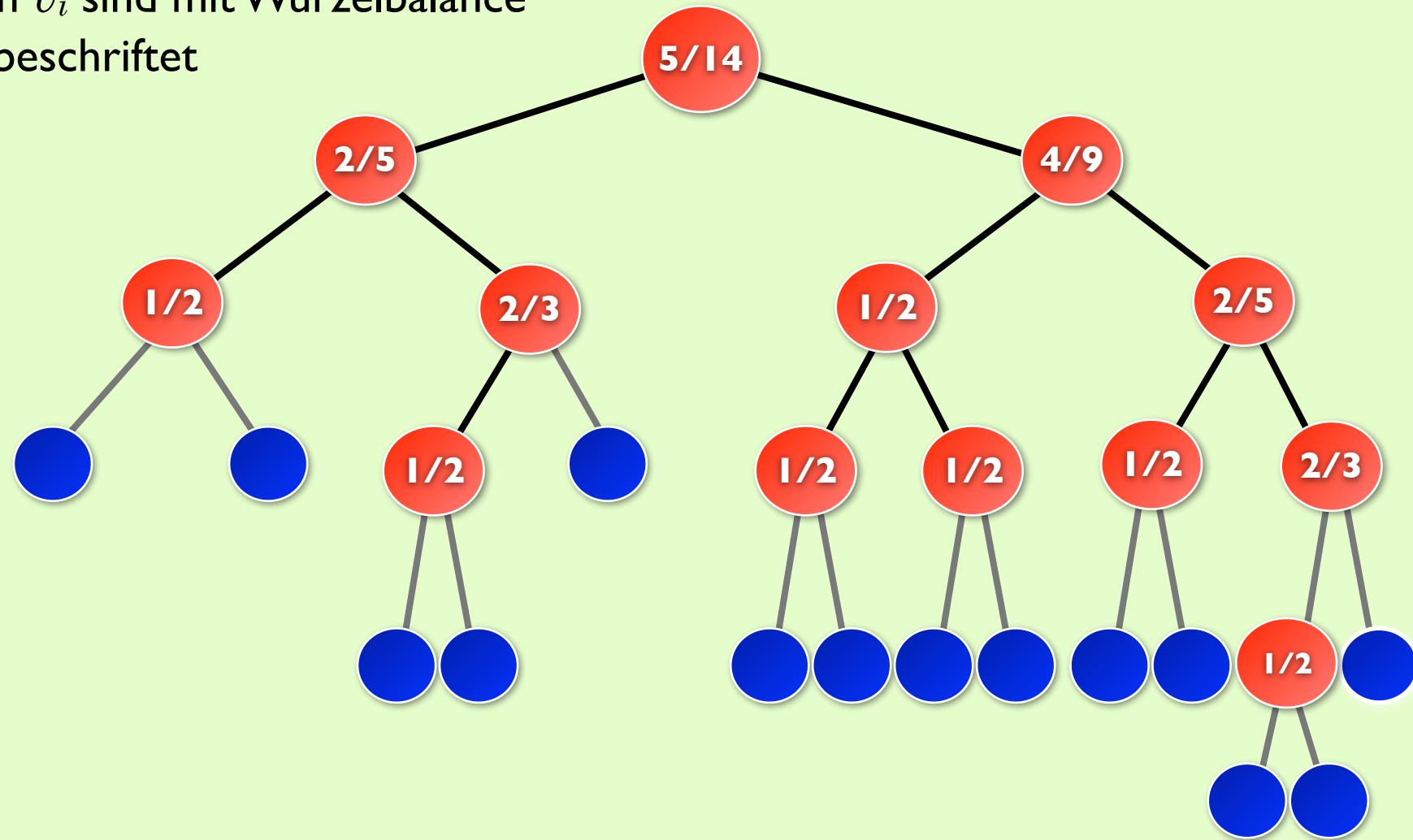
## B Gewichtsbalancierter Baum

Knoten  $v_i$  sind mit Wurzelbalance  
 $\rho(T_i)$  beschriftet

# Beispiel: Gewichtsbalancierter Baum

## B Gewichtsbalancierter Baum

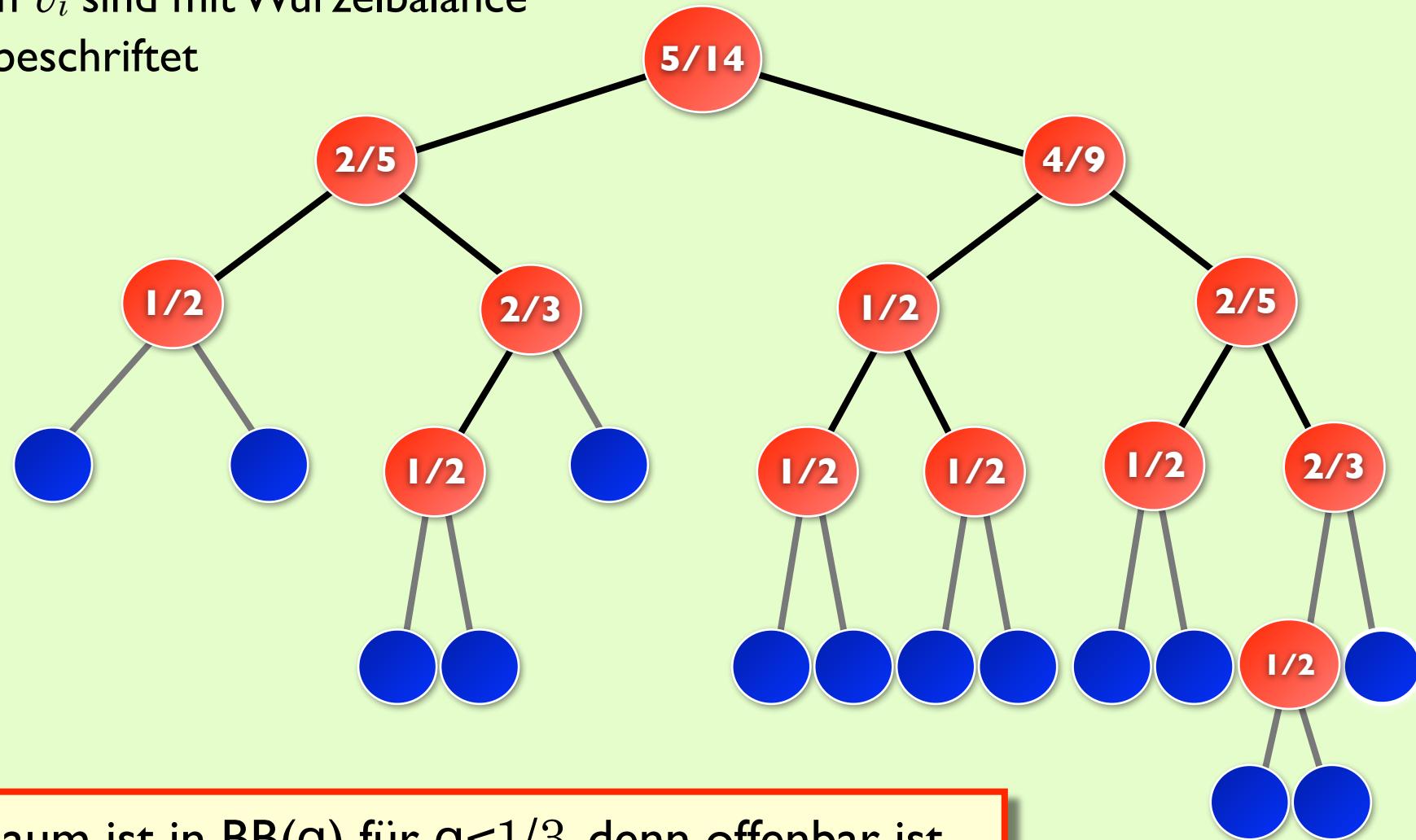
Knoten  $v_i$  sind mit Wurzelbalance  $\rho(T_i)$  beschriftet



# Beispiel: Gewichtsbalancierter Baum

## B Gewichtsbalancierter Baum

Knoten  $v_i$  sind mit Wurzelbalance  $\rho(T_i)$  beschriftet



Der Baum ist in BB( $\alpha$ ) für  $\alpha \leq 1/3$ , denn offenbar ist

$$1/3 \leq \rho(T) \leq 2/3$$

# Gewichtsbalancierte Bäume: Bedeutung von $\alpha$

- Für  $\alpha=1/2$  ist genau die Hälfte aller Blätter im linken Teilbaum

# Gewichtsbalancierte Bäume: Bedeutung von $\alpha$

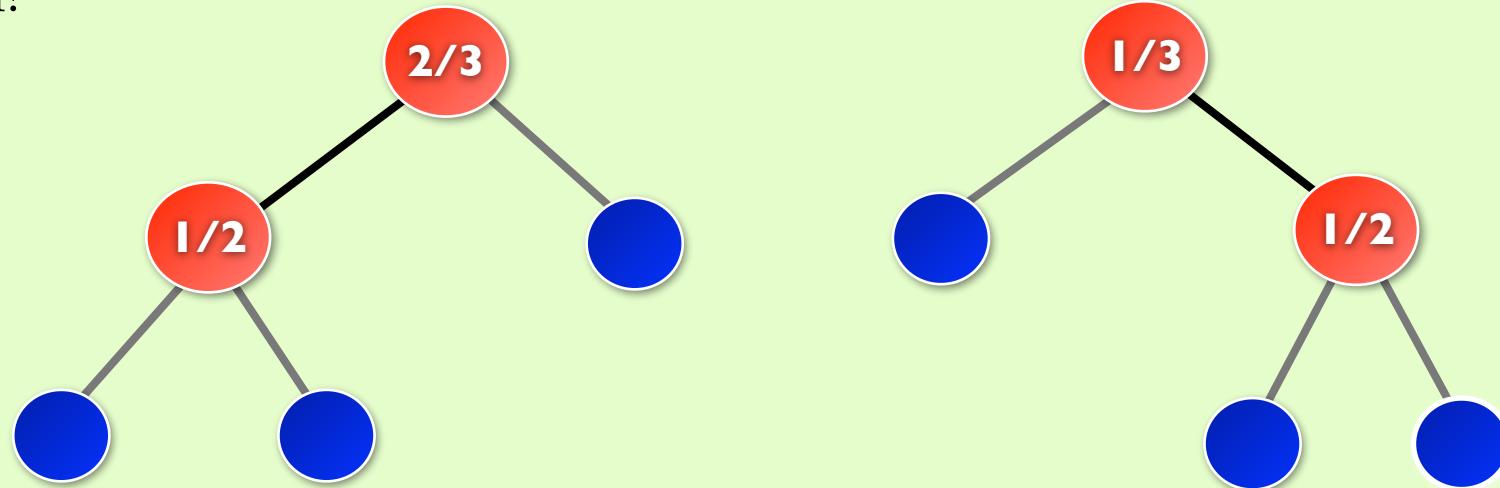
- Für  $\alpha=1/2$  ist genau die Hälfte aller Blätter im linken Teilbaum
- Je näher  $\alpha$  an  $1/2$  ist, desto ausgeglichener ist der Baum!

# Gewichtsbalancierte Bäume: Bedeutung von $\alpha$

- Für  $\alpha=1/2$  ist genau die Hälfte aller Blätter im linken Teilbaum
- Je näher  $\alpha$  an  $1/2$  ist, desto ausgeglichener ist der Baum!
- **Problem:**  $1/2$  nicht sinnvoll, da sich für viele Knotenzahlen kein gewichtsbalancierter Baum mehr finden lässt  
Knotengrad muss überall 2 sein - Bereichsblätter

## B Gewichtsbalancierter Baum - Probleme mit $\alpha=1/2$

Es gibt nur zwei Bäume mit zwei inneren Knoten. Diese haben als Wurzelbalance  $\frac{2}{3}$  und  $\frac{1}{3}$  und liegen somit in  $BB(\frac{2}{3})$  - in  $BB(\frac{1}{2})$  enthält keinen Baum mit 2 inneren Knoten!



# Gewinn der Konstruktion ....

- **Einfüge- und Löschoperationen können Baum aus dem Gleichgewicht bringen**
- **Dann: Rebalancierung erforderlich**  
Mit  $O(\lg n)$  kostengünstig möglich - hier nicht gezeigt

# Gewinn der Konstruktion ....

- **Einfüge- und Löschoperationen können Baum aus dem Gleichgewicht bringen**
- **Dann: Rebalancierung erforderlich**  
Mit  $O(\lg n)$  kostengünstig möglich - hier nicht gezeigt

## S Bäume mit beschränkter Balance: Maximale Höhe

Sei  $T \in \text{BB}[\alpha]$  ein Baum mit  $n$  Knoten, dann gilt

$$\text{Höhe}(T) \leq 1 + \frac{\lg (n+1) - 1}{\lg \left( \frac{1}{1-\alpha} \right)}$$

# Gewinn der Konstruktion ....

- **Einfüge- und Löschoperationen können Baum aus dem Gleichgewicht bringen**
- **Dann: Rebalancierung erforderlich**  
Mit  $O(\text{Id } n)$  kostengünstig möglich - hier nicht gezeigt

## S Bäume mit beschränkter Balance: Maximale Höhe

Sei  $T \in \text{BB}[\alpha]$  ein Baum mit  $n$  Knoten, dann gilt

$$\text{Höhe}(T) \leq 1 + \frac{\text{Id } (n+1) - 1}{\text{Id } \left( \frac{1}{1-\alpha} \right)}$$

Für  $\alpha = 1 - \frac{\sqrt{2}}{2}$  besagt der Satz, dass gilt:

$$\text{Höhe}(T) \leq 2 \cdot \text{Id } (n+1) - 1$$

Die maximale Suchzeit in Bäumen der Klasse  $\text{BB}[1 - \frac{\sqrt{2}}{2}]$  liegt damit höchstens um einen Faktor 2 über dem Optimum.

# Beweis: Höhe Gewichtsbalancierter Baum I

Sei  $T \in \text{BB}[\alpha]$  ein Baum mit  $n$  Knoten,  $h = \text{Höhe}(T)$  und  $v_0, v_1, \dots, v_{h-1}$  ein Pfad von der Wurzel zum Knoten  $v_{h-1}$  der Höhe  $h - 1$ . Sei ferner  $l_i = \text{leafCount}(T_i)$  die Anzahl der Blätter im Unterbaum  $T_i$  für  $0 \leq i \leq h - 1$ . Dann gilt:

$$(* \text{ Bereichsblätter !*}) \quad 2 \leq l_{h-1} \quad \text{und}$$

$$\rho(v_{i+1}) = \frac{l_{i+1}}{l_i} \leq (1 - \alpha) \quad \text{für } 0 \leq i \leq h - 1 \text{ bzw.}$$
$$l_{i+1} \leq (1 - \alpha) \cdot l_i \quad \text{für } 0 \leq i \leq h - 1$$

weil  $T$  von beschränkter Balance  $\alpha$  ist.

Aus

$$l_{i+1} \leq (1 - \alpha) \cdot l_i \quad \text{und} \quad l_i \leq (1 - \alpha) \cdot l_{i-1}$$

folgt

$$l_{i+1} \leq (1 - \alpha) \underbrace{[(1 - \alpha) \cdot l_{i-1}]}_{\geq l_i}$$

Wiederholtes Anwenden dieser Überlegung führt schließlich zu:

$$2 \leq l_{h-1} \leq (1 - \alpha)^{h-1} \cdot l_0 = (1 - \alpha)^{h-1} \cdot (n + 1)$$

# Beweis: Höhe Gewichtsbalancierter Baum II

$$2 \leq l_{h-1} \leq (1 - \alpha)^{h-1} \cdot h_0 = (1 - \alpha)^{h-1} \cdot (n + 1)$$

Logarithmieren und Auflösen nach  $h$ :

$$\text{Id } 2 \leq \text{Id} [(1 - \alpha)^{h-1} \cdot (n + 1)]$$

(\*  $\log(a \cdot b) = \log(a) + \log(b)$  \*)

$$1 \leq \text{Id} (1 - \alpha)^{h-1} + \text{Id} (n + 1)$$

(\*  $\log(a^b) = b * \log(a)$  \*)

$$\text{Id} (n + 1) - 1 \geq -(h - 1) \cdot \text{Id} (1 - \alpha)$$

(\*  $-\log(a) = \log(\frac{1}{a})$  \*)

$$\text{Id} (n + 1) - 1 \geq (h - 1) \cdot \text{Id} \left( \frac{1}{1 - \alpha} \right)$$

$$\frac{\text{Id} (n + 1) - 1}{\text{Id} \left( \frac{1}{1 - \alpha} \right)} \geq h - 1$$

$$\frac{\text{Id} (n + 1) - 1}{\text{Id} \left( \frac{1}{1 - \alpha} \right)} + 1 \geq h$$

# V. Suchen in Mengen

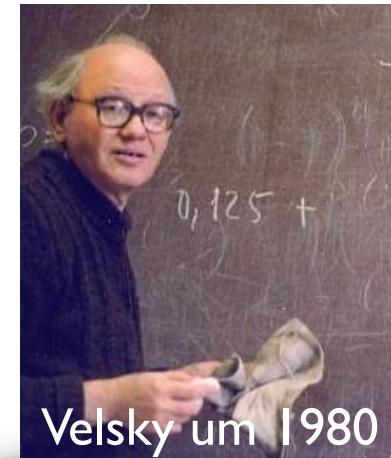
---

## 5. Suchen in Mengen

- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume**
  - 5.5.1. Gewichtsbalanceierte Bäume
  - 5.5.2. AVL-Bäume**
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume
- 5.6. *Priority Queue* und *Heap*

# AVL Bäume

- 1962 von Georgi Maximowitsch Adelson-Welski (engl: Velsky) und Jewgeni Michailowitsch Landis entwickelt



- Älteste Datenstruktur für balancierte Bäume
- Vielleicht **nicht die effizienteste Methode**, aber gut zur Darstellung des Prinzips „Balancierter Baum“

## D AVL-Baum

Ein **AVL-Baum** ist ein binärer Suchbaum mit einer Struktur-Invarianten: Für jeden Knoten gilt, dass sich die **Zahl der Level** seiner beiden Teilbäume um maximal Eins unterscheiden.

**Hinweis:** In der Literatur wird die Invariante häufig der an Höhe der Teilbäume festgemacht. Die Höhe eines Baumes ist dann, anders als hier, als Zahl der durchlaufenen Knoten von der Wurzel zum tiefsten Blatt definiert.

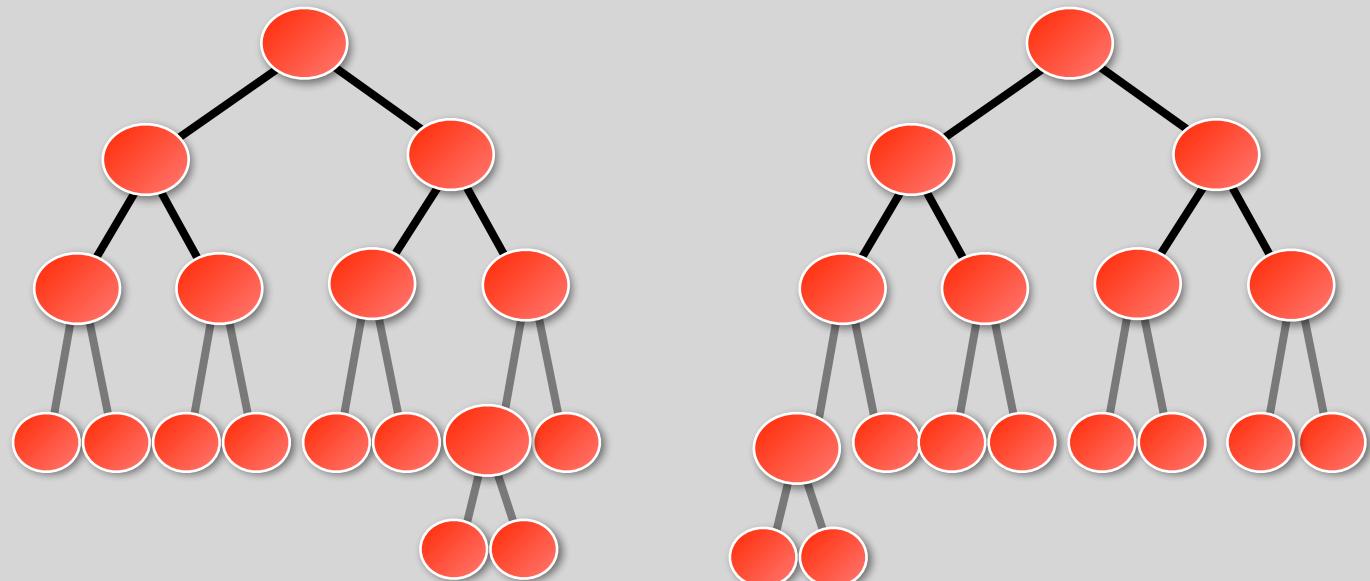
# AVL-Baum: Beispiele

## B AVL-Bäume

# AVL-Baum: Beispiele

## B AVL-Bäume

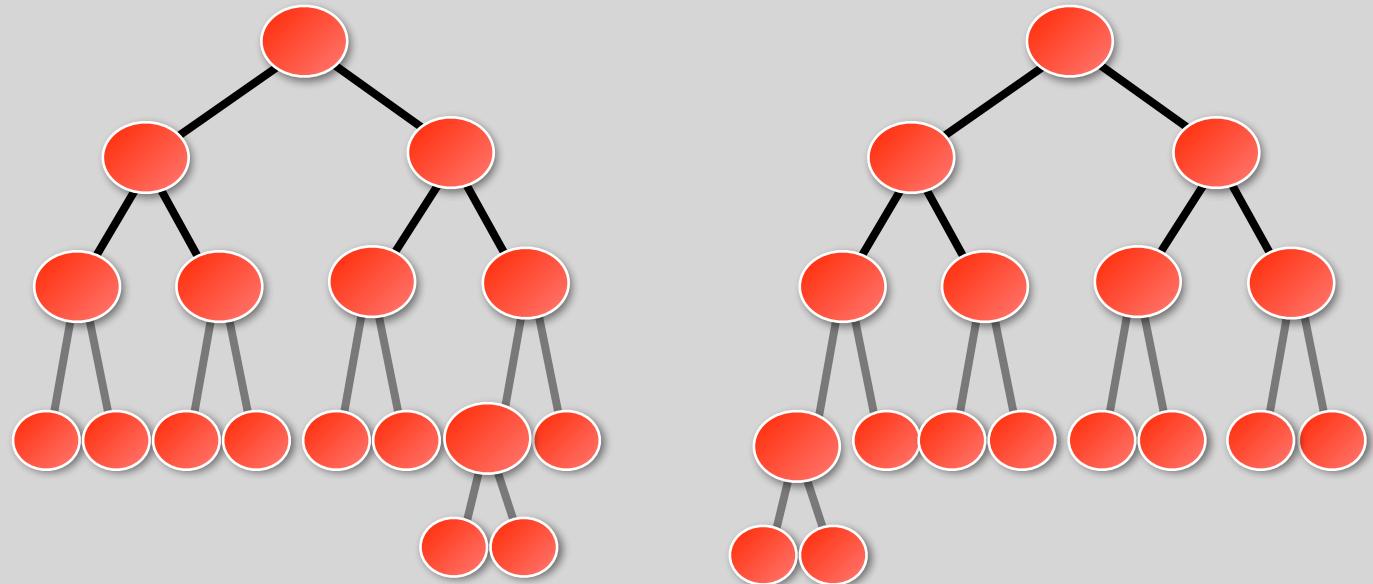
### AVL-Bäume:



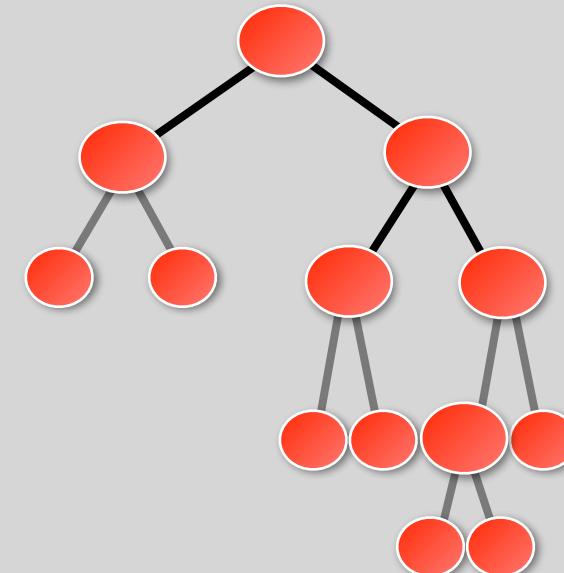
# AVL-Baum: Beispiele

## B AVL-Bäume

### AVL-Bäume:



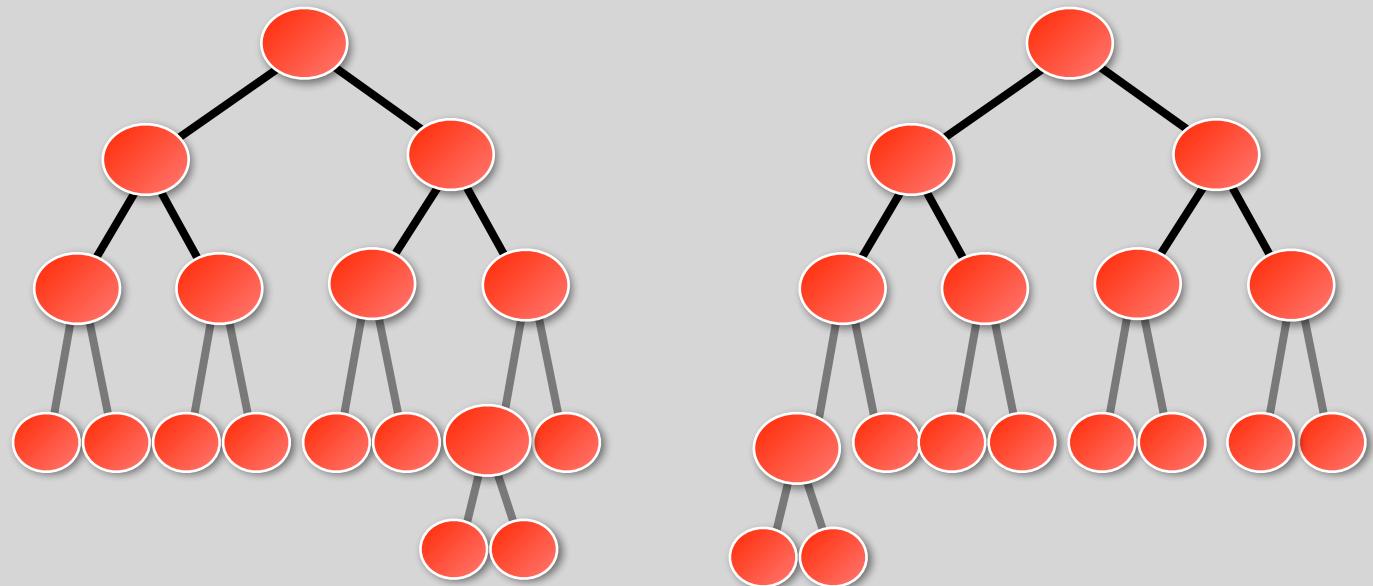
### Kein AVL-Baum:



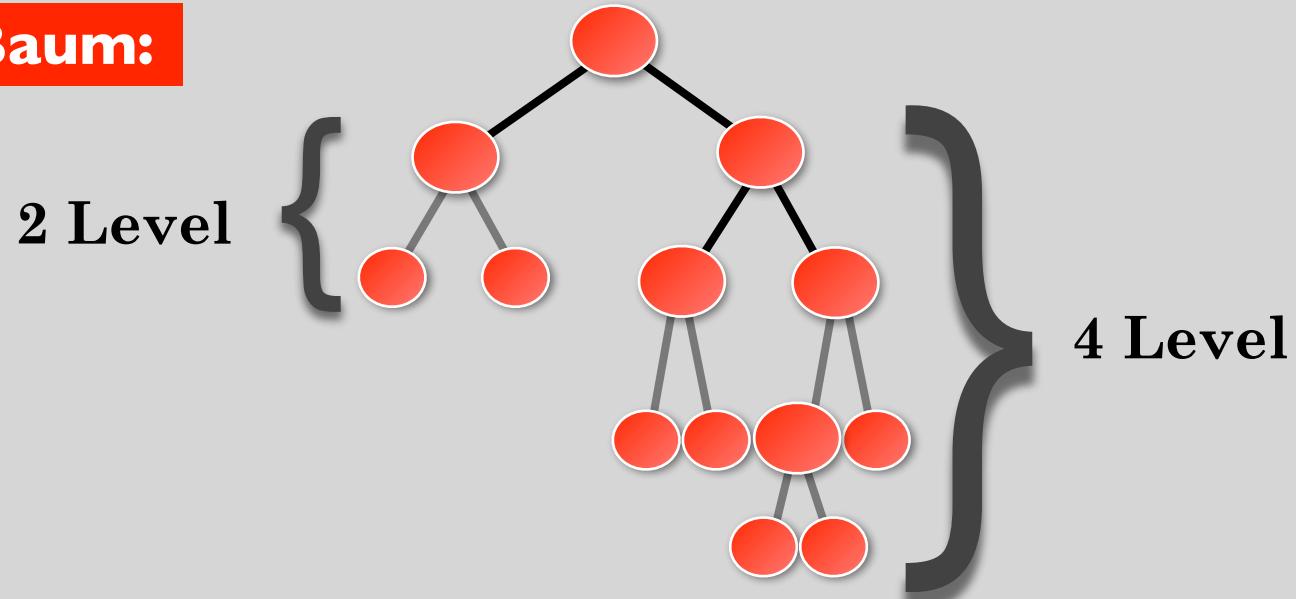
# AVL-Baum: Beispiele

## B AVL-Bäume

### AVL-Bäume:



### Kein AVL-Baum:



# Höhe eines AVL-Baums I

- Wie hoch kann ein AVL-Baum im *worst case* werden?

# Höhe eines AVL-Baums I

- Wie hoch kann ein AVL-Baum im *worst case* werden?
- $N(h)$ : *minimale Anzahl der Knoten* in AVL-Baum der Höhe  $h$

# Höhe eines AVL-Baums I

- Wie hoch kann ein AVL-Baum im *worst case* werden?
- $N(h)$ : *minimale Anzahl der Knoten* in AVL-Baum der Höhe  $h$ 
  - $h=0$ : Der Baum besteht nur aus der Wurzel

$$N(0)=1$$

# Höhe eines AVL-Baums I

- Wie hoch kann ein AVL-Baum im *worst case* werden?
  - $N(h)$ : *minimale Anzahl der Knoten* in AVL-Baum der Höhe  $h$
- $h=0$ : Der Baum besteht nur aus der Wurzel  
 $N(0)=1$
  - $h=1$ : Da minimal gefüllte Bäume betrachtet werden, enthält die nächste Ebene nur einen Knoten:

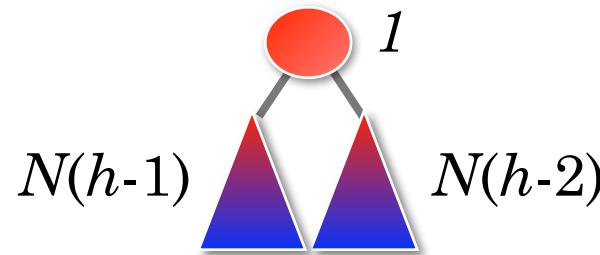


und es gilt

$$N(1)=2$$

# Höhe eines AVL-Baums I

- Wie hoch kann ein AVL-Baum im *worst case* werden?
  - $N(h)$ : *minimale Anzahl der Knoten* in AVL-Baum der Höhe  $h$
- $h=0$ : Der Baum besteht nur aus der Wurzel  
 $N(0)=1$
  - $h=1$ : Da minimal gefüllte Bäume betrachtet werden, enthält die nächste Ebene nur einen Knoten:
- oder
- 
- und es gilt
- $$N(1)=2$$
- $h \geq 2$ : Methode: Kombiniere 2 Bäume der Höhe  $h-1$  und  $h-2$  zu einem neuen minimal gefülltem Baum



# Höhe eines AVL-Baums I

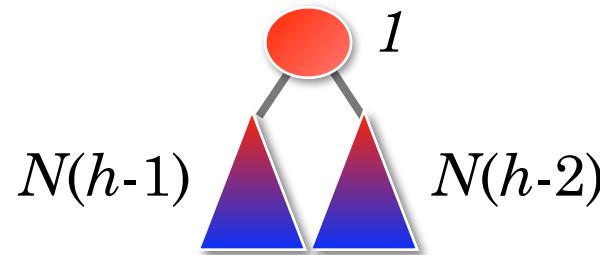
- Wie hoch kann ein AVL-Baum im *worst case* werden?
- $N(h)$ : *minimale Anzahl der Knoten* in AVL-Baum der Höhe  $h$
- $h=0$ : Der Baum besteht nur aus der Wurzel  
 $N(0)=1$
- $h=1$ : Da minimal gefüllte Bäume betrachtet werden, enthält die nächste Ebene nur einen Knoten:



und es gilt

$$N(1)=2$$

- $h \geq 2$ : Methode: Kombiniere 2 Bäume der Höhe  $h-1$  und  $h-2$  zu einem neuen minimal gefülltem Baum



- Rekursionsgleichung:

$$N(h)=1 + N(h-1) + N(h-2)$$

# Höhe eines AVL-Baums II

- **Kommt Ihnen das irgendwie bekannt vor?**

$$N(h) = 1 + N(h-1) + N(h-2)$$

# Höhe eines AVL-Baums II

- **Kommt Ihnen das irgendwie bekannt vor?**

$$N(h)=1 + N(h-1) + N(h-2)$$

- **Ähnlichkeit mit der Gleichung für die Fibonacci-Zahlen:**

$$\text{Fib}(n)=\text{Fib}(n-1)+\text{Fib}(n-2)$$

# Höhe eines AVL-Baums II

- Kommt Ihnen das irgendwie bekannt vor?

$$N(h) = 1 + N(h-1) + N(h-2)$$

- Ähnlichkeit mit der Gleichung für die Fibonacci-Zahlen:

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

$h$	0	1	2	3	4	5	6	7	8
$\text{Fib}(h)$	1	1	2	3	5	8	13	21	34
$N(h)$	1	2	4	7	12	20	33	54	88

# Höhe eines AVL-Baums II

- Kommt Ihnen das irgendwie bekannt vor?

$$N(h) = 1 + N(h-1) + N(h-2)$$

- Ähnlichkeit mit der Gleichung für die Fibonacci-Zahlen:

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

$h$	0	1	2	3	4	5	6	7	8
$\text{Fib}(h)$	1	1	2	3	5	8	13	21	34
$N(h)$	1	2	4	7	12	20	33	54	88

$$N(h) = \text{Fib}(h+2)-1$$

# Höhe eines AVL-Baums II

- Kommt Ihnen das irgendwie bekannt vor?

$$N(h) = 1 + N(h-1) + N(h-2)$$

- Ähnlichkeit mit der Gleichung für die Fibonacci-Zahlen:

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

$h$	0	1	2	3	4	5	6	7	8
$\text{Fib}(h)$	1	1	2	3	5	8	13	21	34
$N(h)$	1	2	4	7	12	20	33	54	88

$$N(h) = \text{Fib}(h+2)-1$$

Beweis: vollständige Induktion ...

# Höhe eines AVL-Baums III

- Wir hatten

$$N(h) = \text{Fib}(h+2) - 1$$

- Wie wir später sehen werden gilt

$$\text{Fib}(n) = \frac{1}{\sqrt{5}} \cdot (\phi^n - \hat{\phi}^n) \quad \text{mit } \phi = \frac{1 + \sqrt{5}}{2} \text{ und } \hat{\phi} = \frac{1 - \sqrt{5}}{2}$$

- mit  $n \geq N(h)$  folgt jetzt

$$\begin{aligned} n &\geq \left[ \frac{1}{\sqrt{5}} \cdot (\phi^{h+2} - \hat{\phi}^{h+2}) \right] - 1 \\ &\geq \phi^h - 1 \\ n + 1 &\geq \phi^h \\ \log_\phi(n + 1) &\geq h \\ h &\leq \log_\phi(n + 1) \\ h &\leq \frac{\ln(2)}{\ln \phi} \cdot \text{ld}(n + 1) \\ h &\leq 1.4404 \cdot \text{ld}(n + 1) \end{aligned}$$

**Ergebnis:** Ein AVL Baum mit  $n$  Knoten ist maximal 44% höher als ein vollständig ausgeglichener Binärbaum!

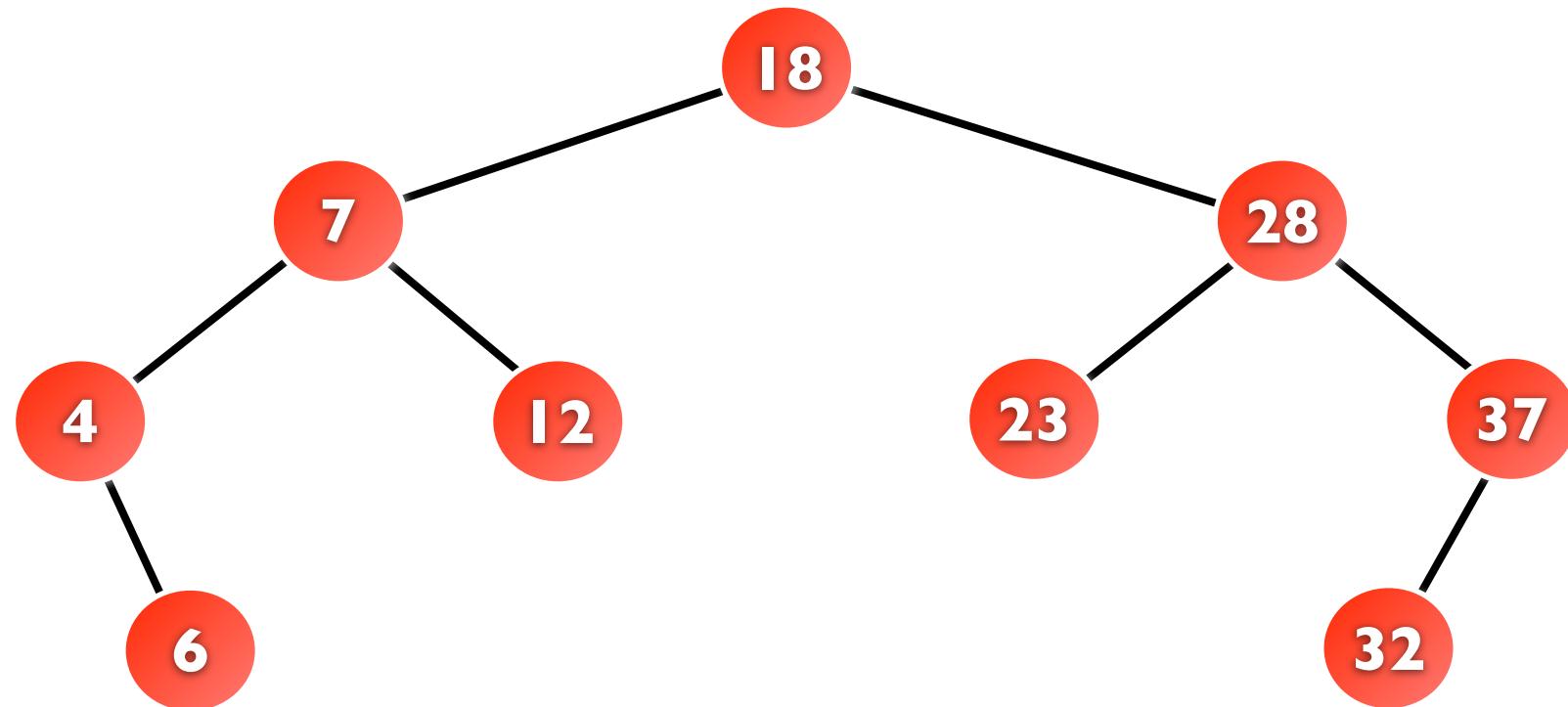
# Operationen auf AVL-Bäumen

- **Suchen**
  - analog zu binärem Suchbaum
- **Einfügen / Löschen**
  1. wie bei binärem Suchbaum: suche Einfügestelle bzw. zu entfernenden Knoten und führe Einfügen bzw. Entfernen aus
  2. Stelle fest, ob AVL-Eigenschaft an der Einfüge- bzw. Entfernungsstelle noch gilt.
  3. Rebalancierung durch Rotation
- **Untersuchung:**
  - Wie kann man feststellen, ob die AVL-Eigenschaft noch gilt?
  - Wie kann ein Baum rebalanciert werden?

# Vorgehensweise

- Bei jedem Knoten  $T$  wird die **Leveldifferenz  $b$**  (die Balance) der beiden Teilbäume  $T_l$  und  $T_r$  abgespeichert:

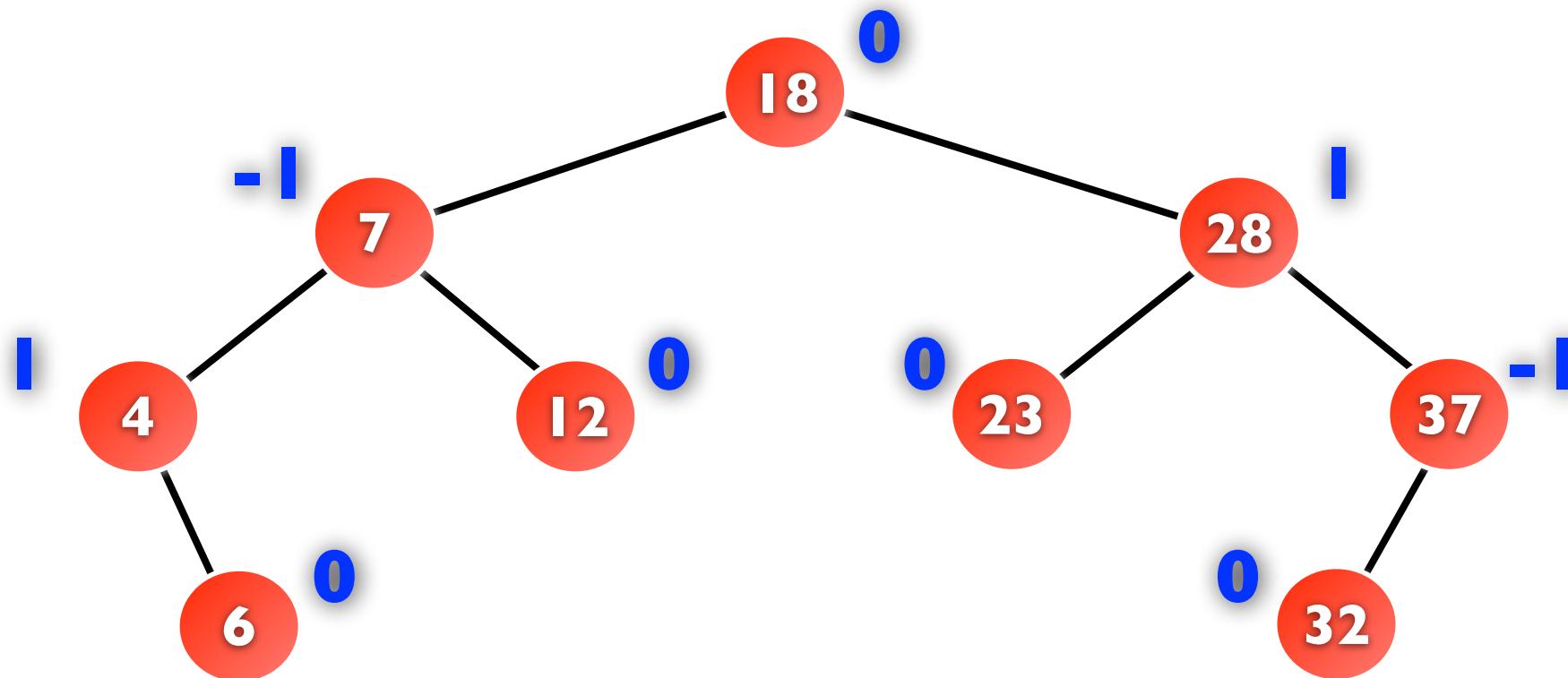
$$b = \text{LevelZahl}(T_r) - \text{LevelZahl}(T_l)$$



# Vorgehensweise

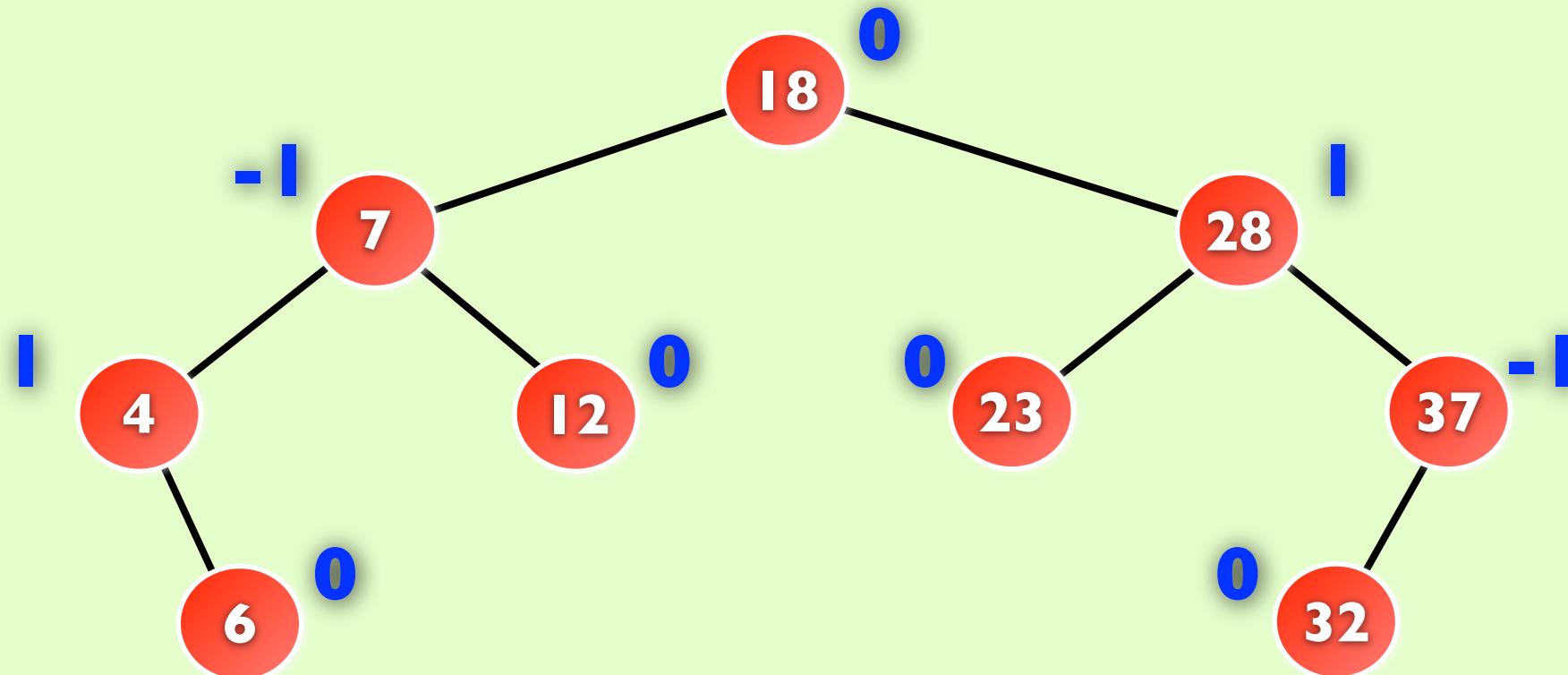
- Bei jedem Knoten  $T$  wird die **Leveldifferenz  $b$**  (die Balance) der beiden Teilbäume  $T_l$  und  $T_r$  abgespeichert:

$$b = \text{LevelZahl}(T_r) - \text{LevelZahl}(T_l)$$



# Vorgehensweise

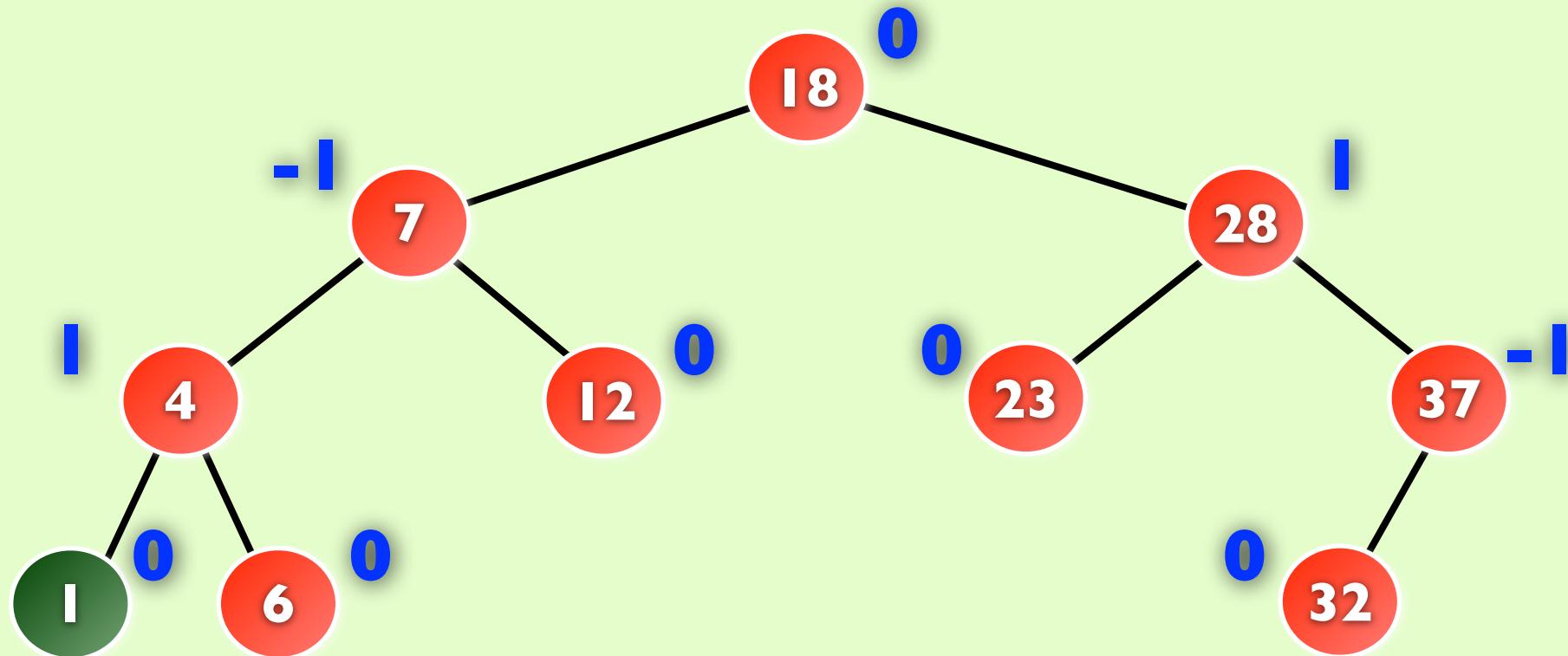
## B Einfügen in einen AVL-Baum



- Einfügen der **1** ist unproblematisch - Balance bleibt im Rahmen
- Einfügen der **33** bringt den Baum aus der Balance - mehrere Knoten haben die Balance -2 oder 2.

# Vorgehensweise

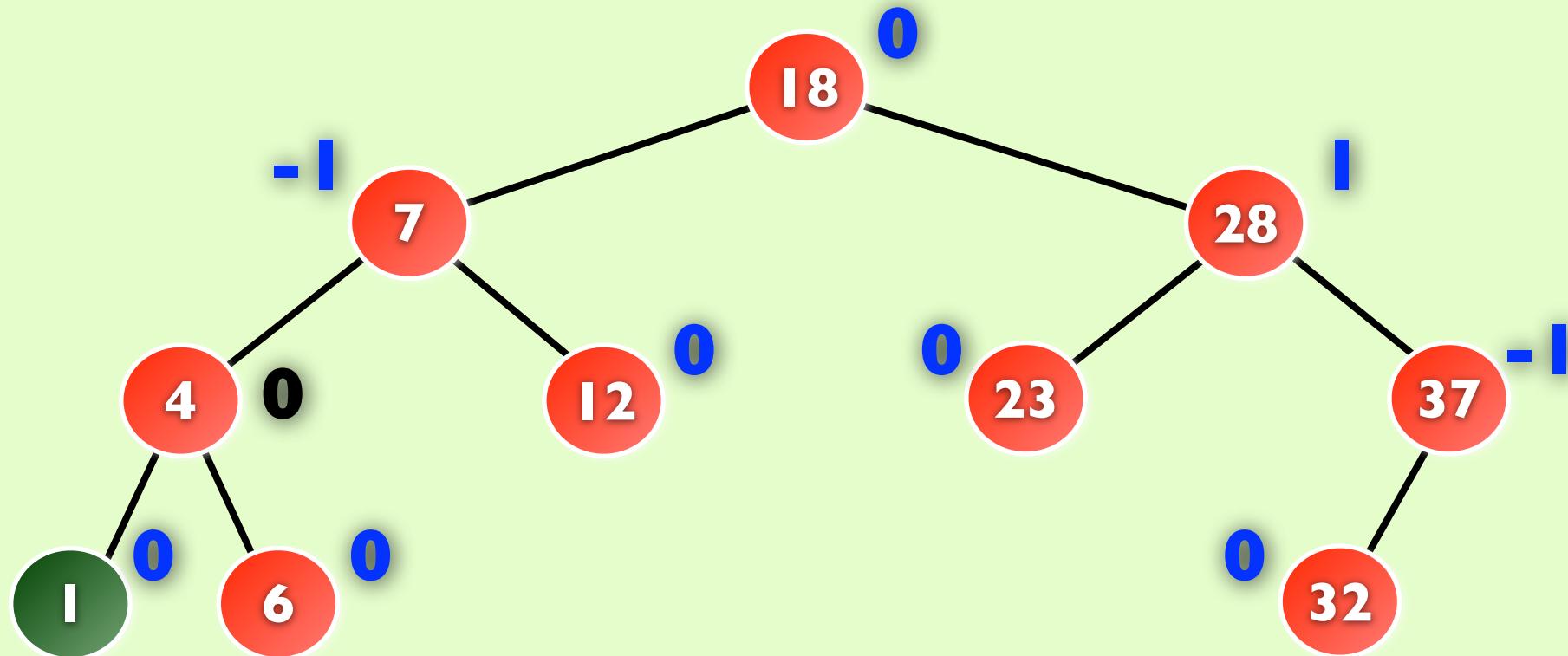
## B Einfügen in einen AVL-Baum



- Einfügen der **I** ist unproblematisch - Balance bleibt im Rahmen
- Einfügen der **33** bringt den Baum aus der Balance - mehrere Knoten haben die Balance -2 oder 2.

# Vorgehensweise

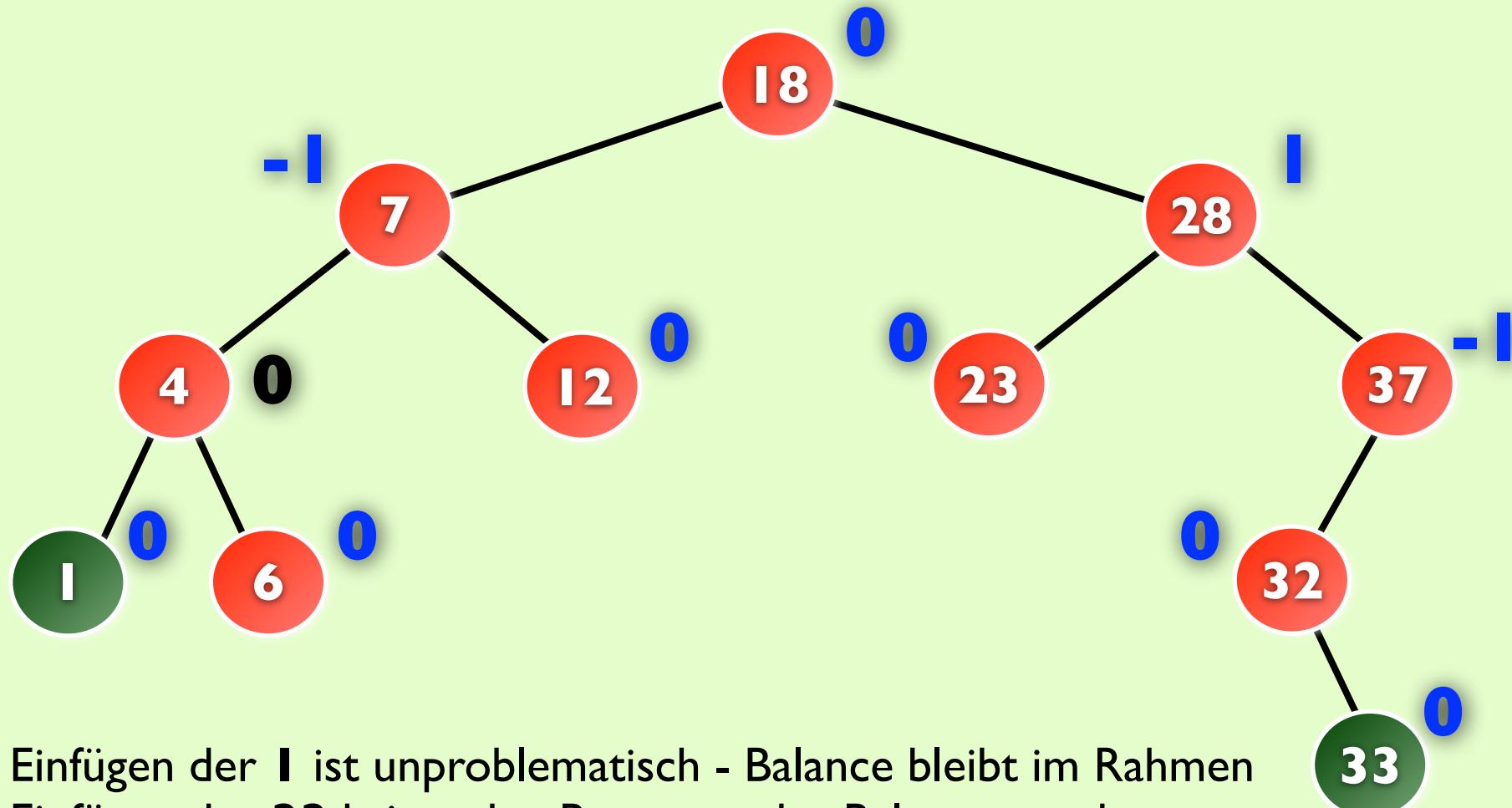
## B Einfügen in einen AVL-Baum



- Einfügen der **I** ist unproblematisch - Balance bleibt im Rahmen
- Einfügen der **33** bringt den Baum aus der Balance - mehrere Knoten haben die Balance -2 oder 2.

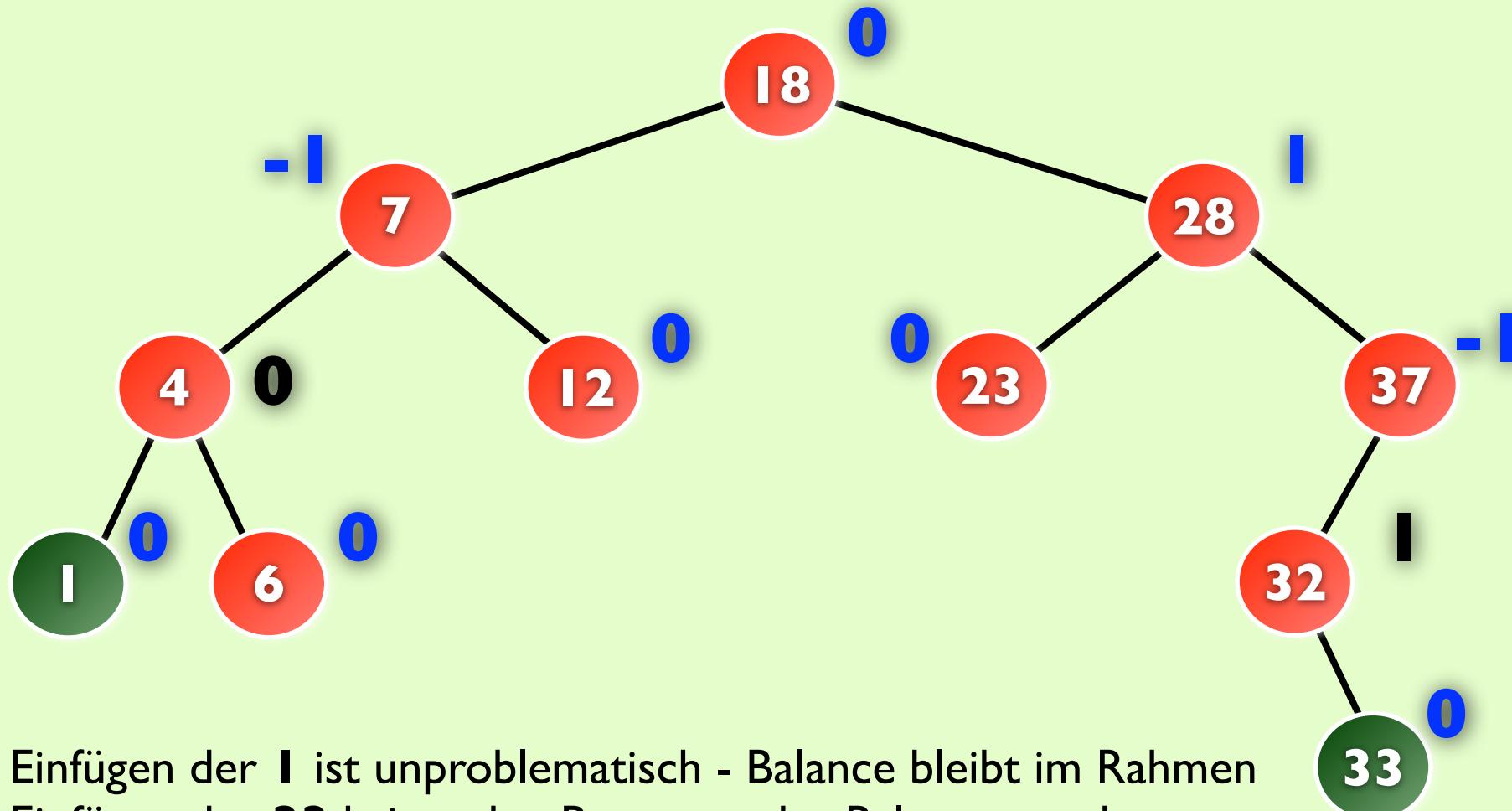
# Vorgehensweise

## B Einfügen in einen AVL-Baum



# Vorgehensweise

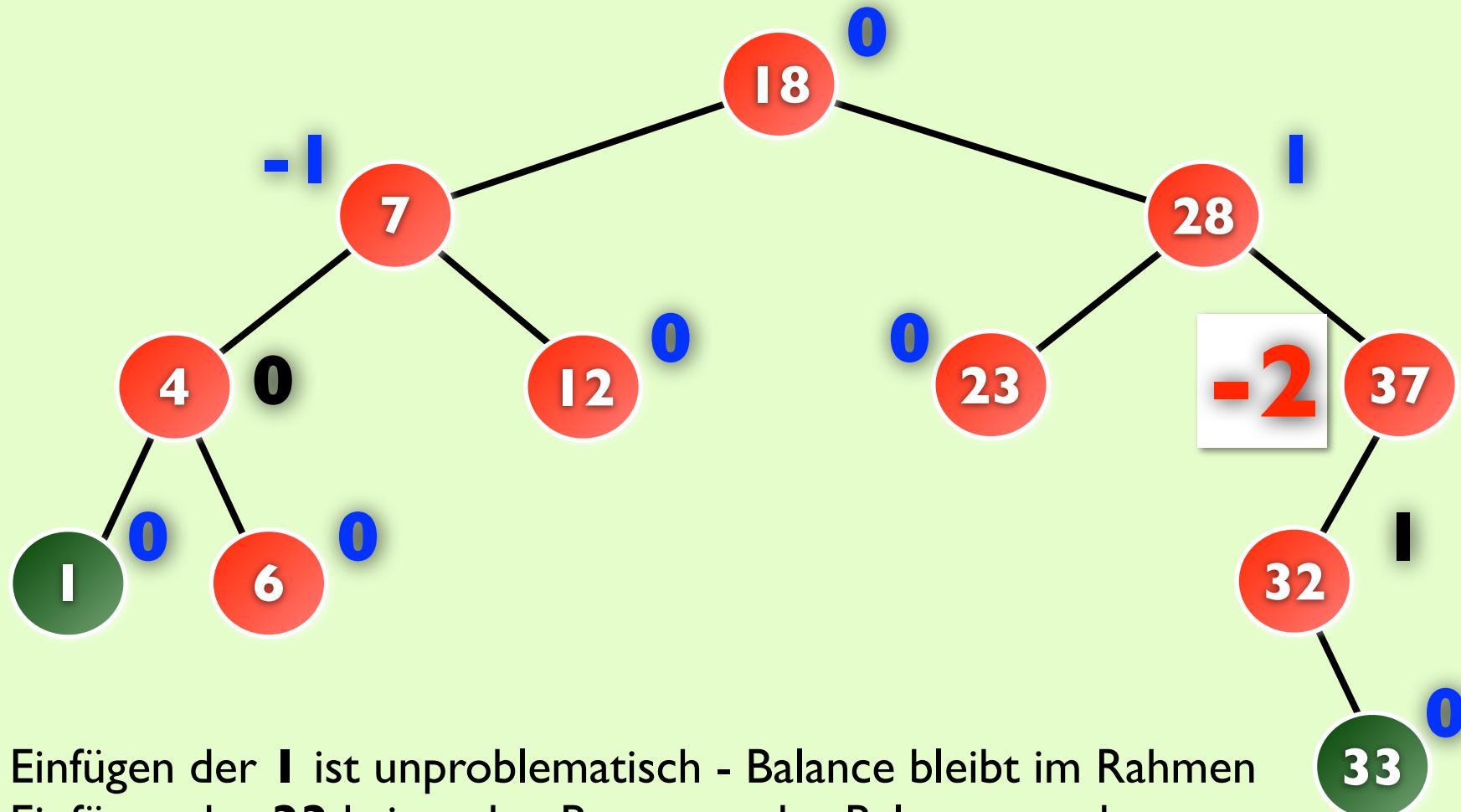
## B Einfügen in einen AVL-Baum



- Einfügen der **1** ist unproblematisch - Balance bleibt im Rahmen
- Einfügen der **33** bringt den Baum aus der Balance - mehrere Knoten haben die Balance -2 oder 2.

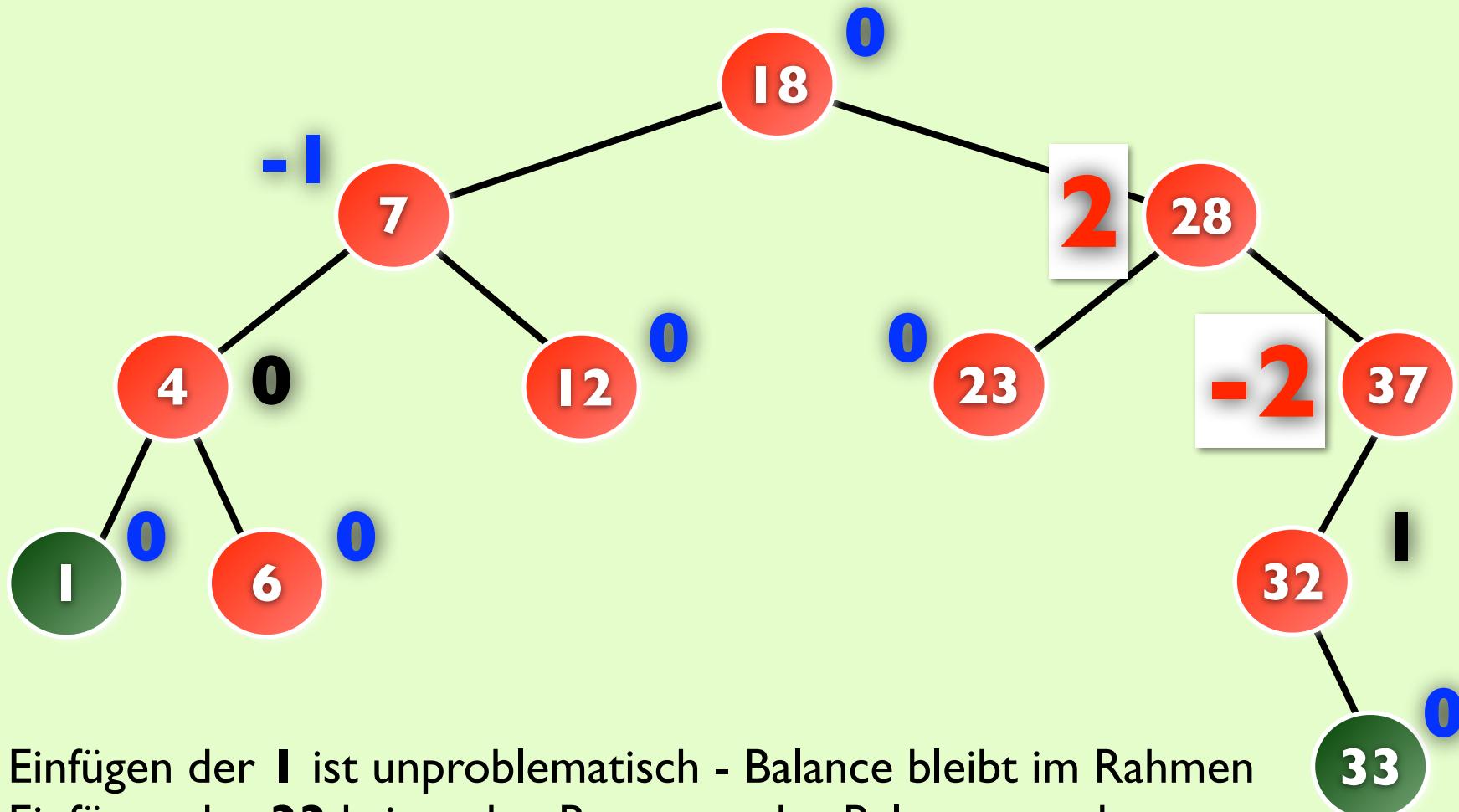
# Vorgehensweise

## B Einfügen in einen AVL-Baum



# Vorgehensweise

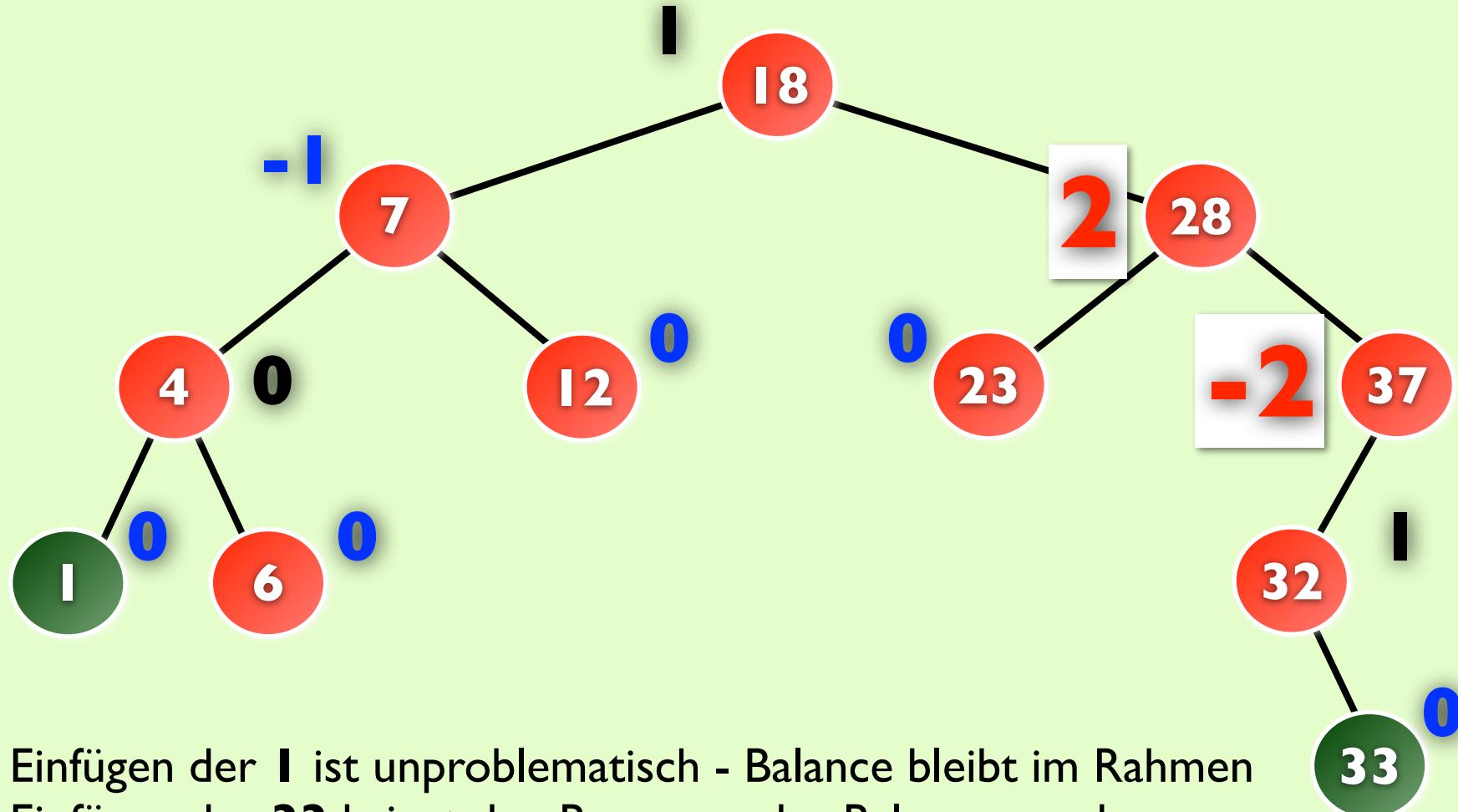
## B Einfügen in einen AVL-Baum



- Einfügen der **1** ist unproblematisch - Balance bleibt im Rahmen
- Einfügen der **33** bringt den Baum aus der Balance - mehrere Knoten haben die Balance -2 oder 2.

# Vorgehensweise

## B Einfügen in einen AVL-Baum

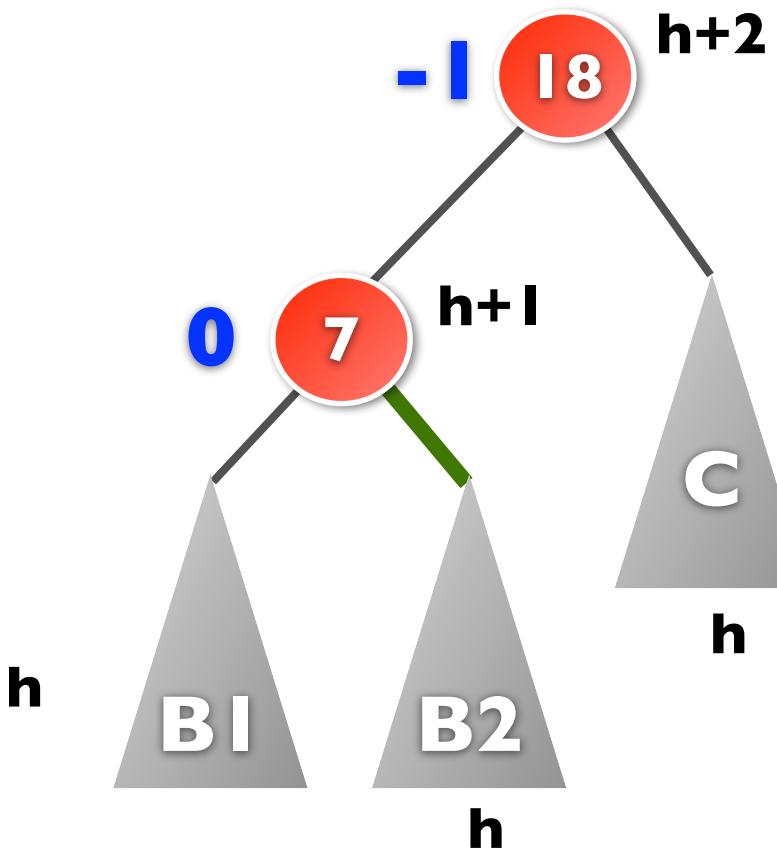


# Einfügen - Rebalancierung durch Rotation

- **Zunächst:** Einfügen wie in Binärbaum (neues Blatt)
- **Beim Einfügen kann die Balance gestört werden**  
(Ein oder mehrere Knoten haben Balance -2 oder 2)
- **Nach dem Einfügen:**
  - Rückwärts in Richtung Wurzel gehen und den **ersten** Knoten suchen, dessen Balance gestört ist (diesen nennt man auch den **kritischen Knoten**)
  - Suche nach kritischem Knoten hat im *worst case* die Komplexität  $O(\lg n)$  (die maximale Tiefe eines AVL-Baums ist ja  $1.44 \lg(n)$ )
  - Ist die Balance an einer Stelle gestört, kann sie in konstanter Zeit ( $O(1)$ ) mittels einer **Rotation** oder einer **Doppelrotation** wiederhergestellt werden.

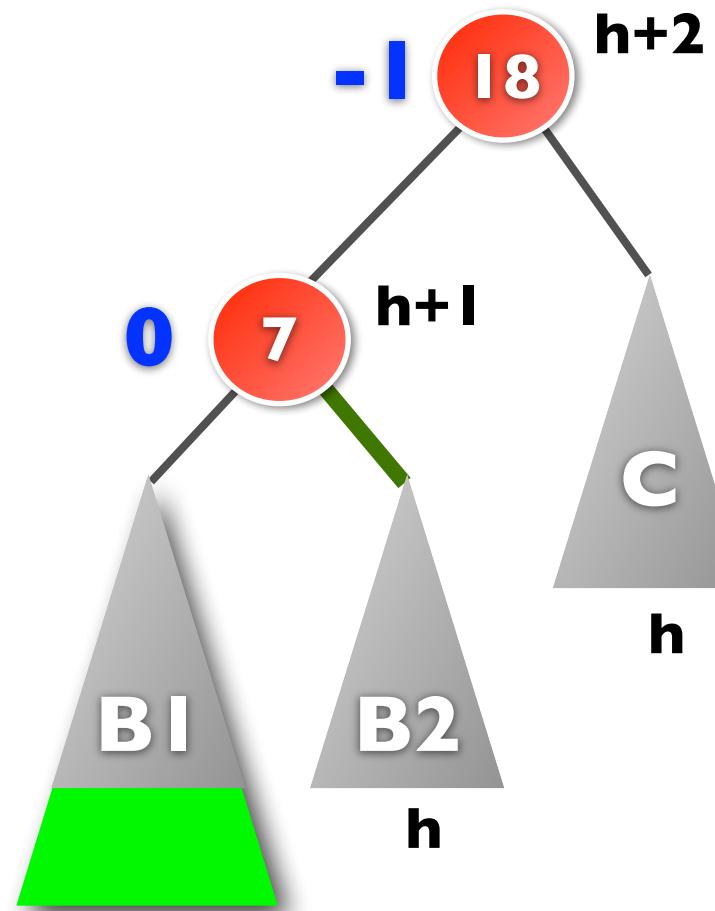
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



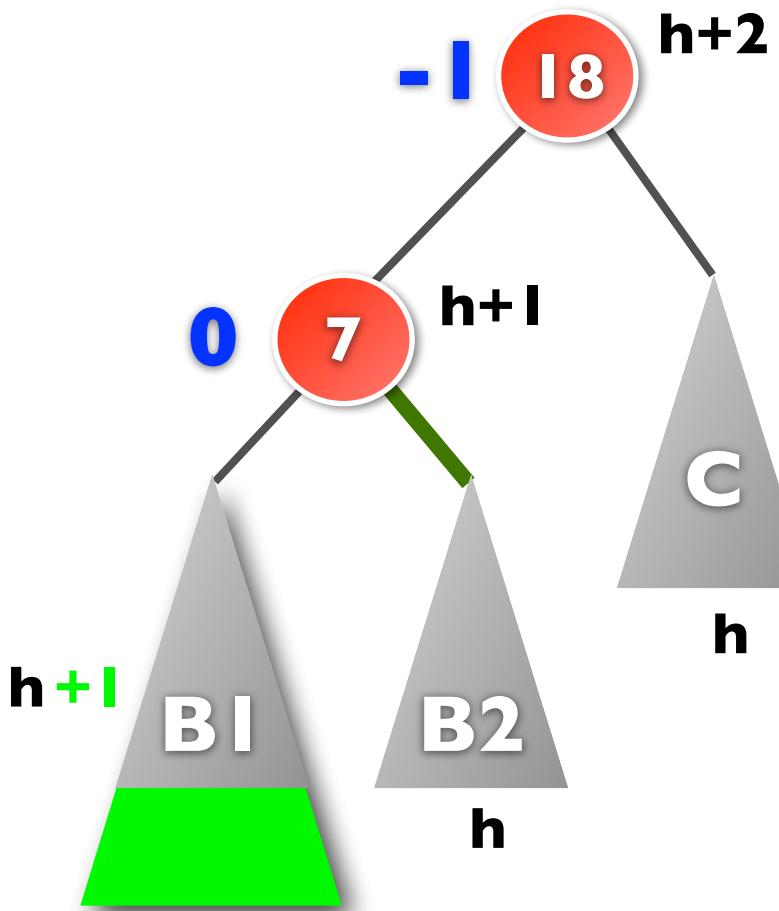
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



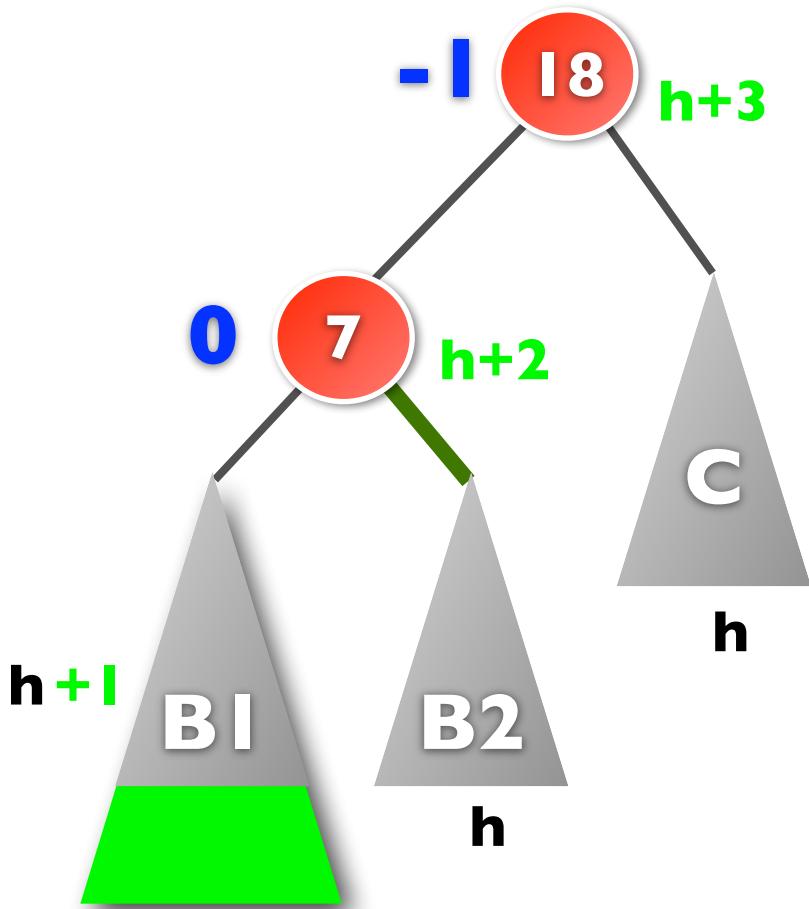
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



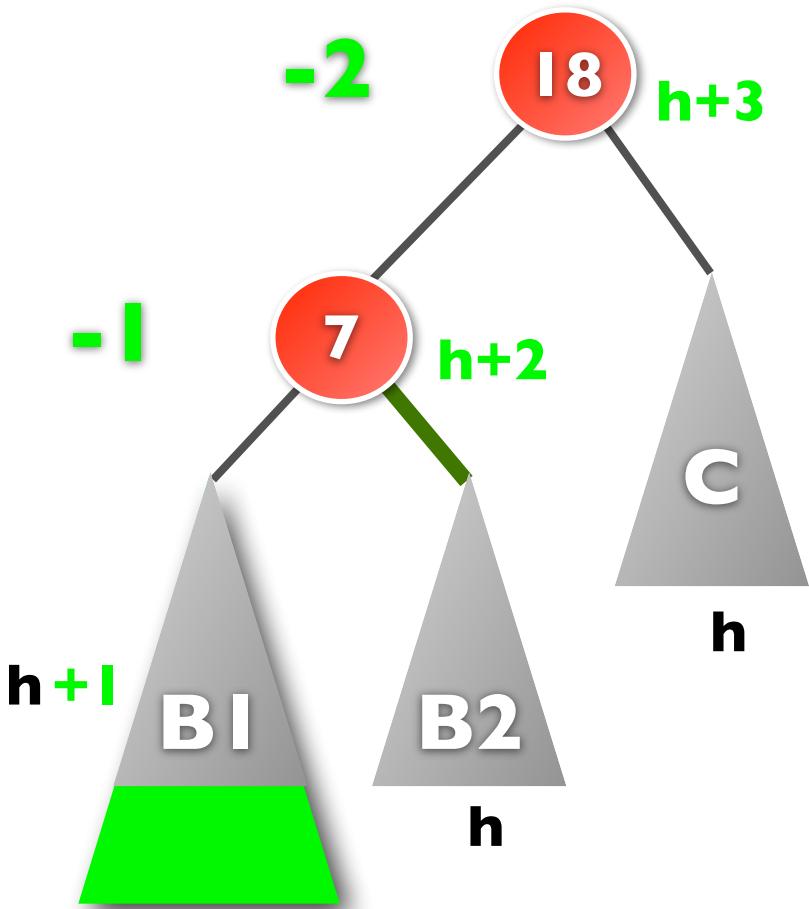
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



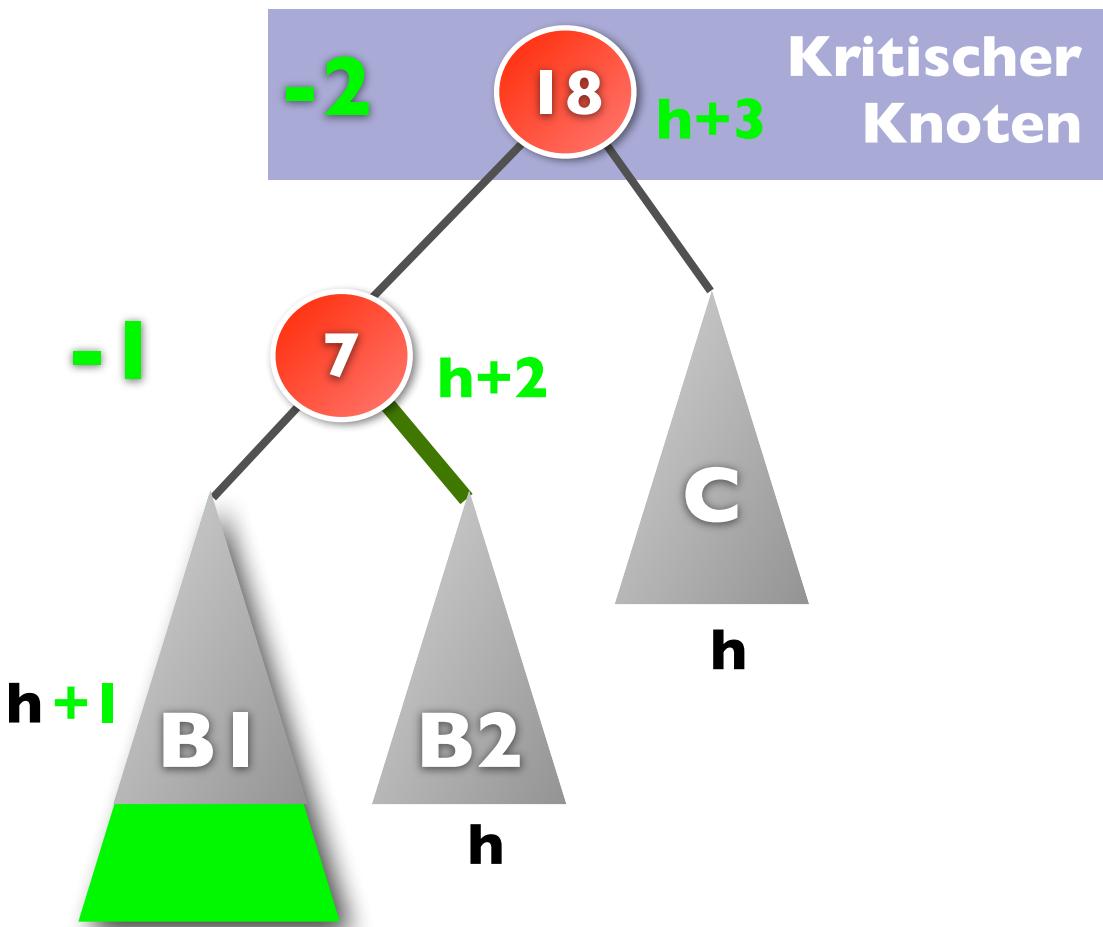
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



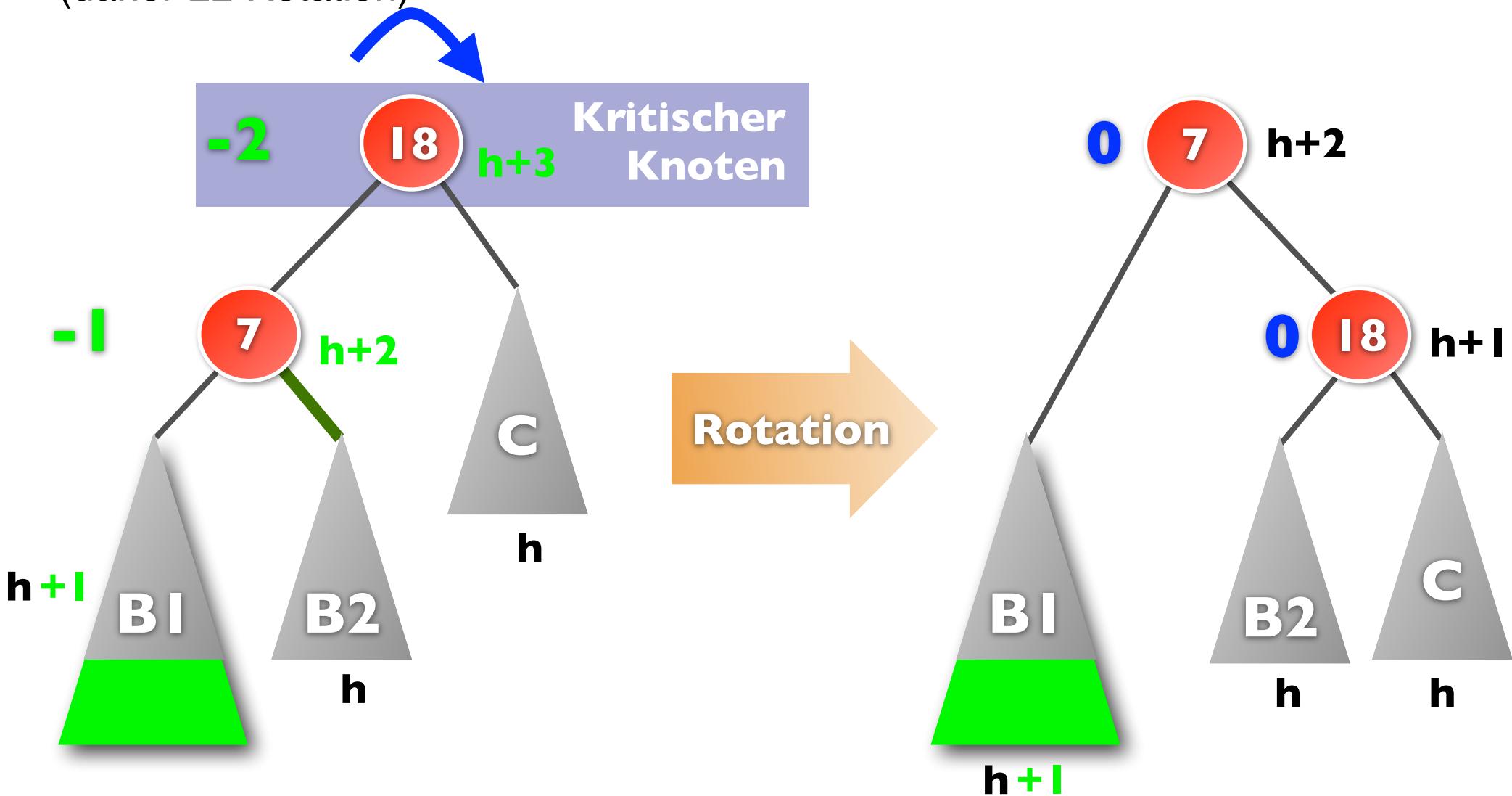
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



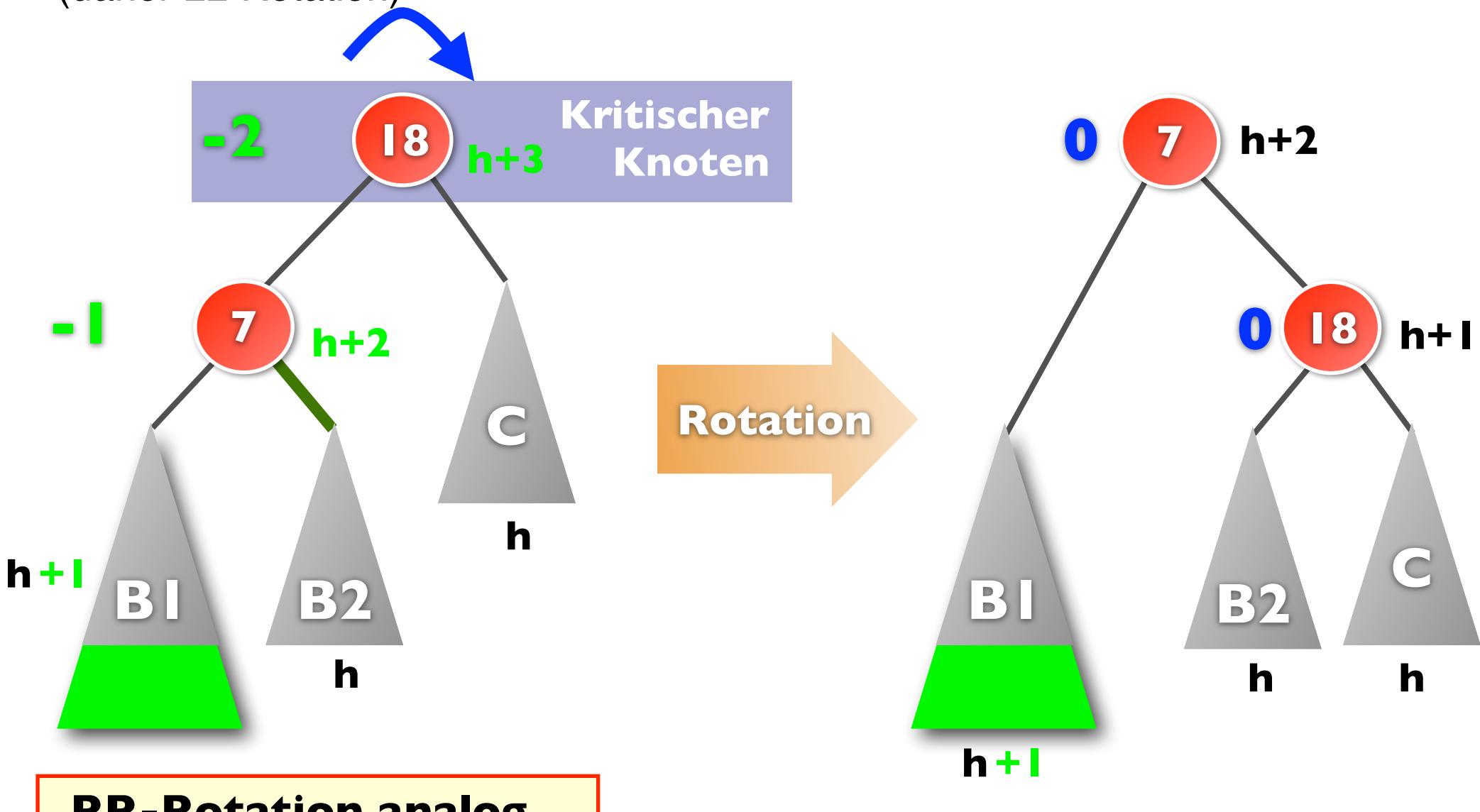
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



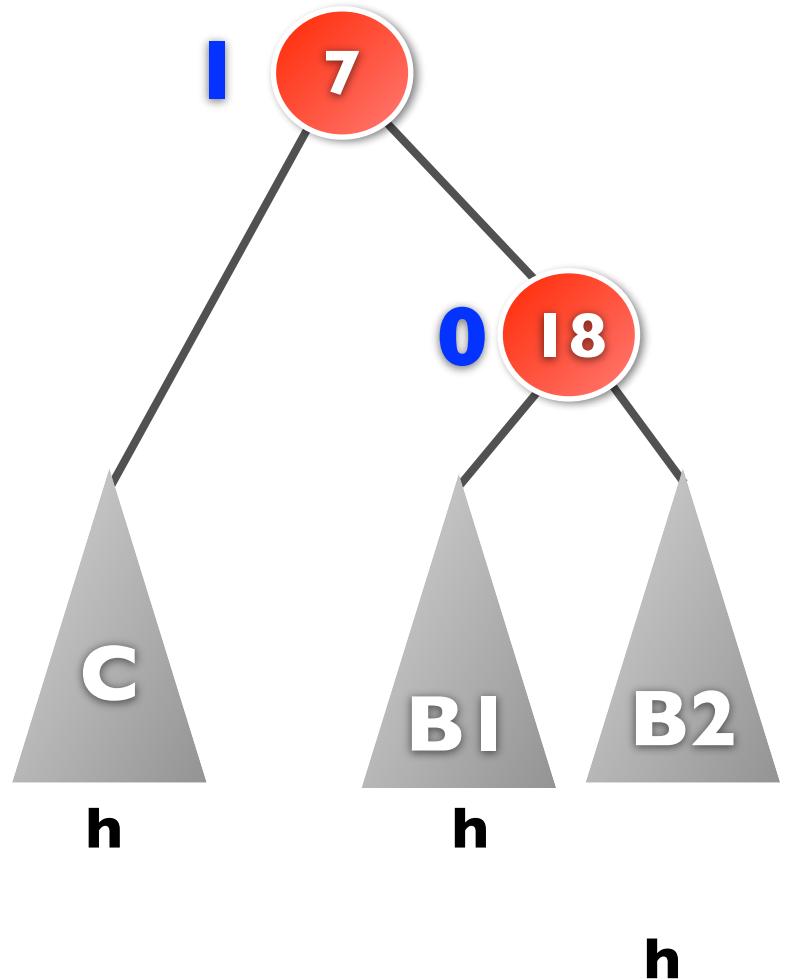
# LL-Rotation

- Einfügen fand im linken Teilbaum des linken Teilbaums statt  
(daher LL-Rotation)



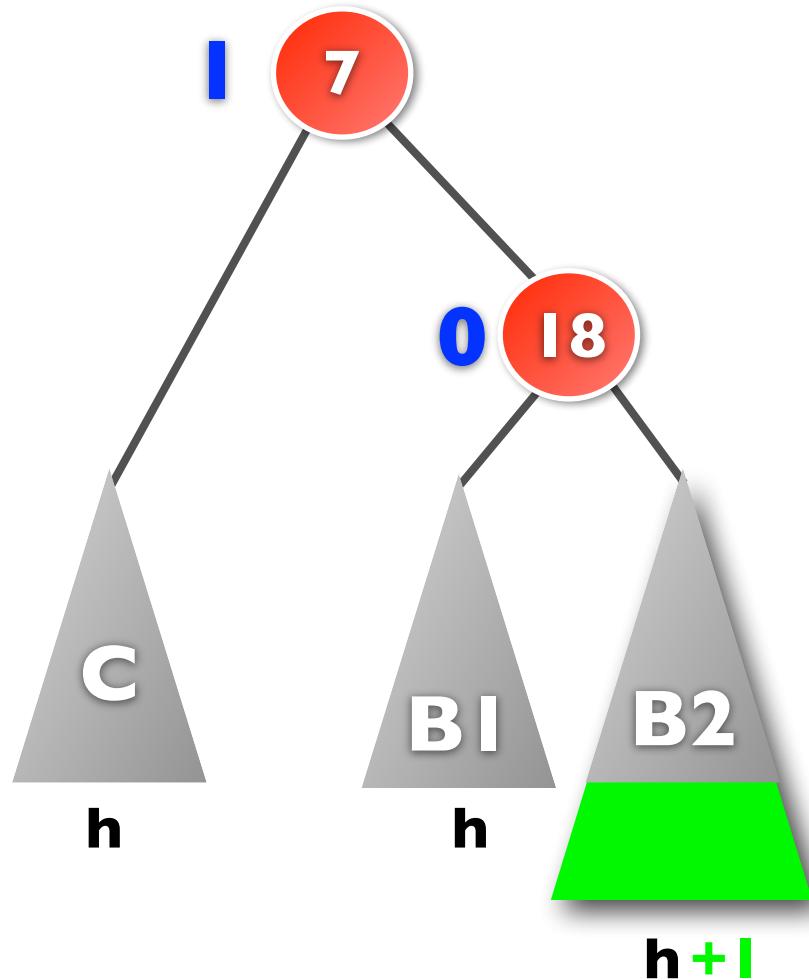
# RR-Rotation

- **Einfügen fand im rechten Teilbaum des rechten Teilbaumes statt**  
(daher RR-Rotation)



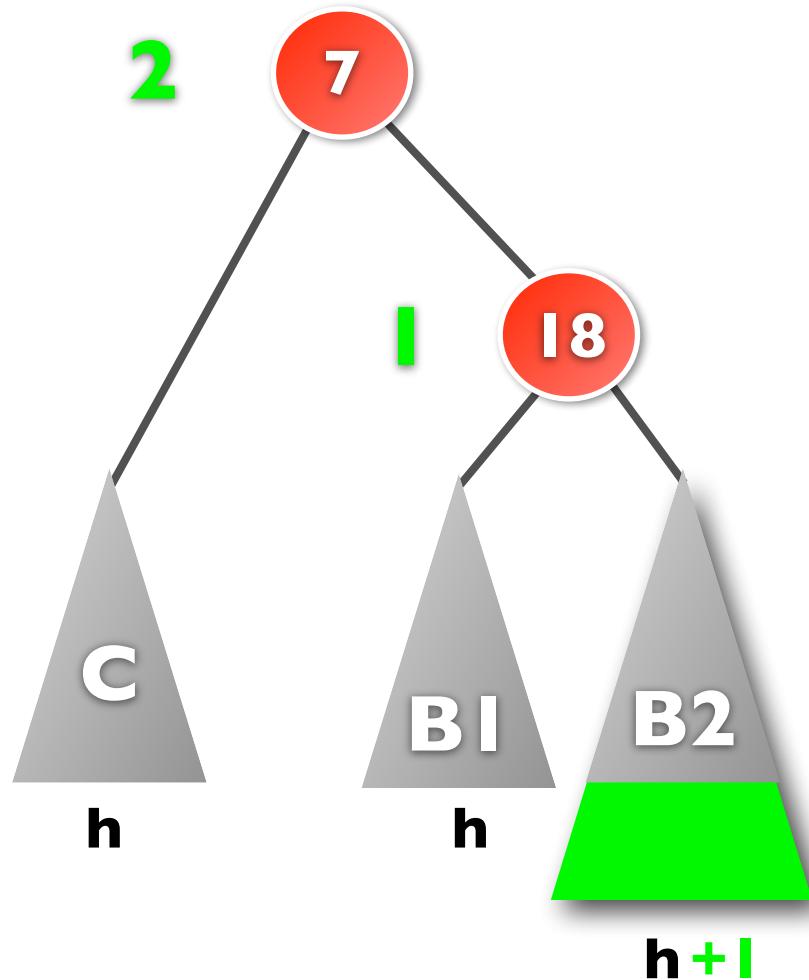
# RR-Rotation

- **Einfügen fand im rechten Teilbaum des rechten Teilbaumes statt**  
(daher RR-Rotation)



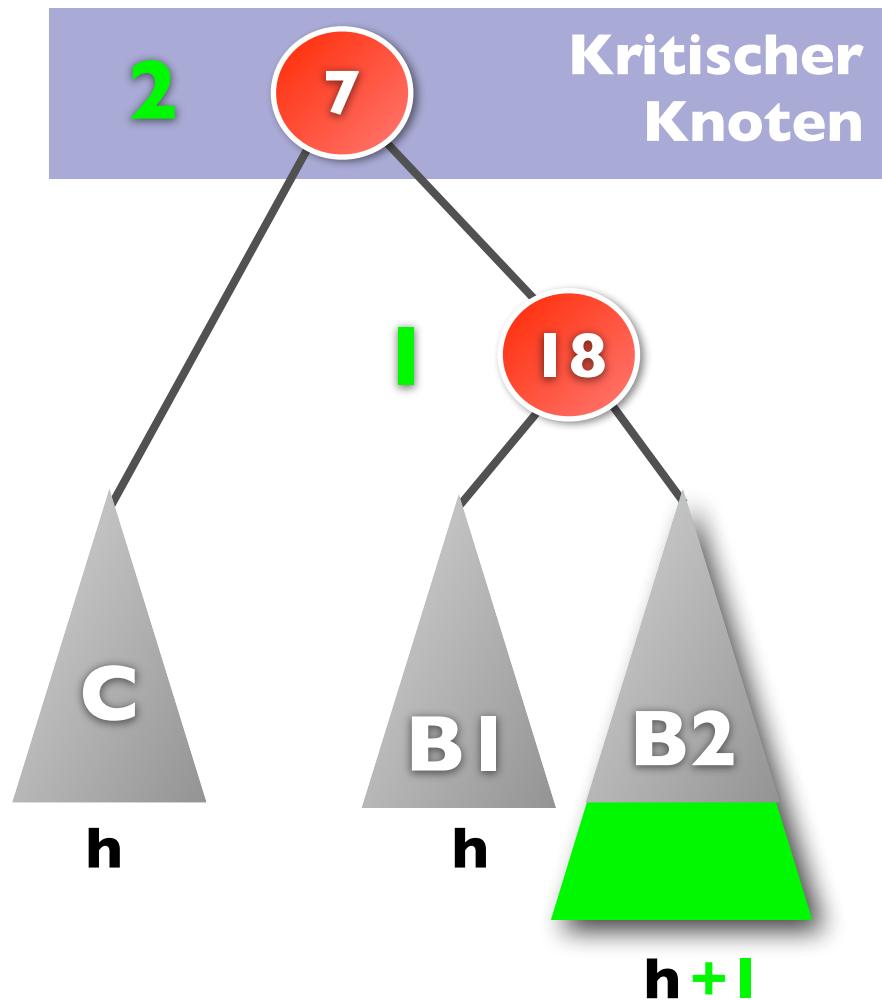
# RR-Rotation

- **Einfügen fand im rechten Teilbaum des rechten Teilbaumes statt**  
(daher RR-Rotation)



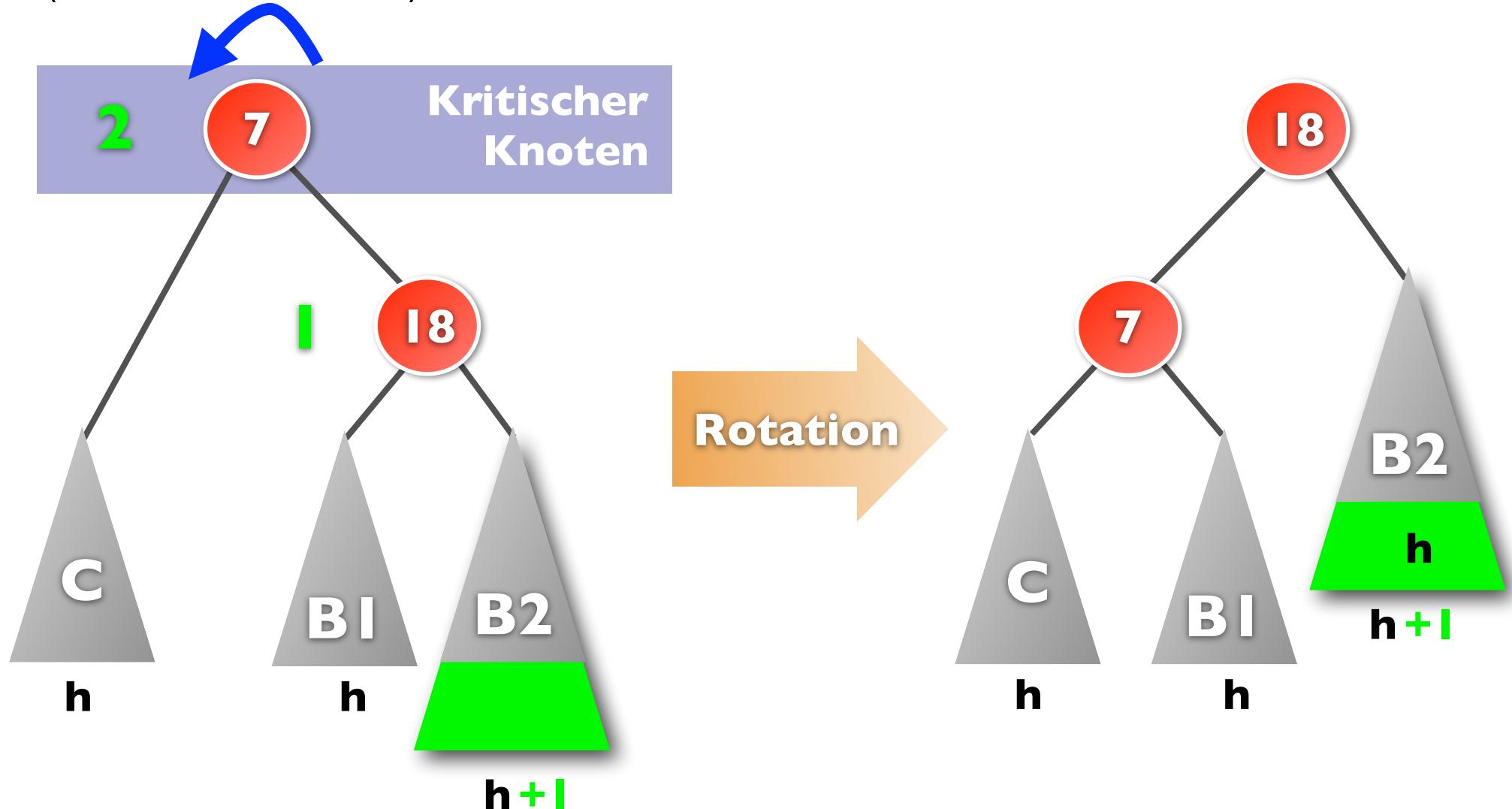
# RR-Rotation

- **Einfügen fand im rechten Teilbaum des rechten Teilbaumes statt**  
(daher RR-Rotation)



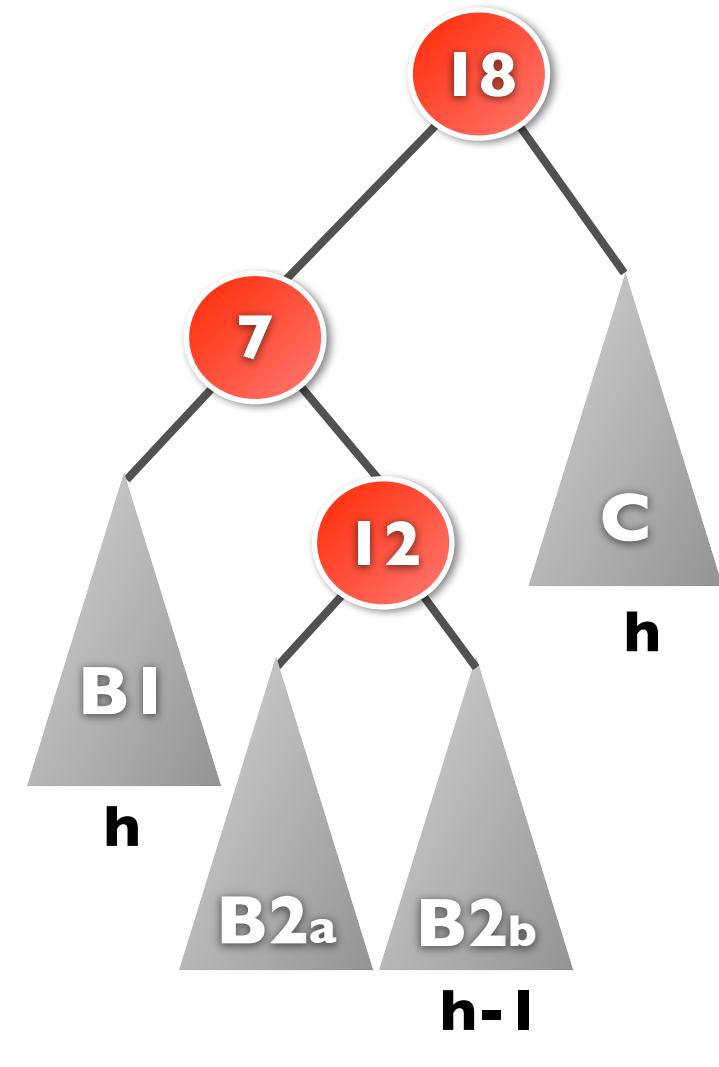
# RR-Rotation

- Einfügen fand im **rechten Teilbaum des rechten Teilbaumes** statt  
(daher RR-Rotation)



# RL-Rotation (Doppelrotation)

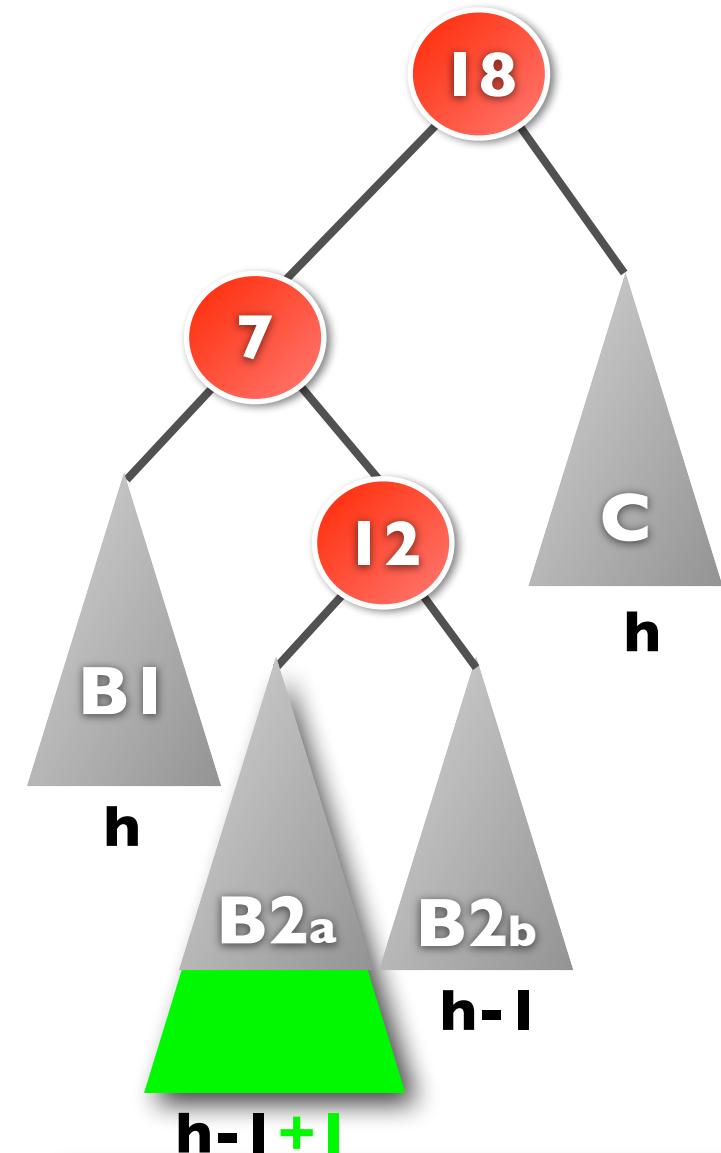
- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



**h - l**

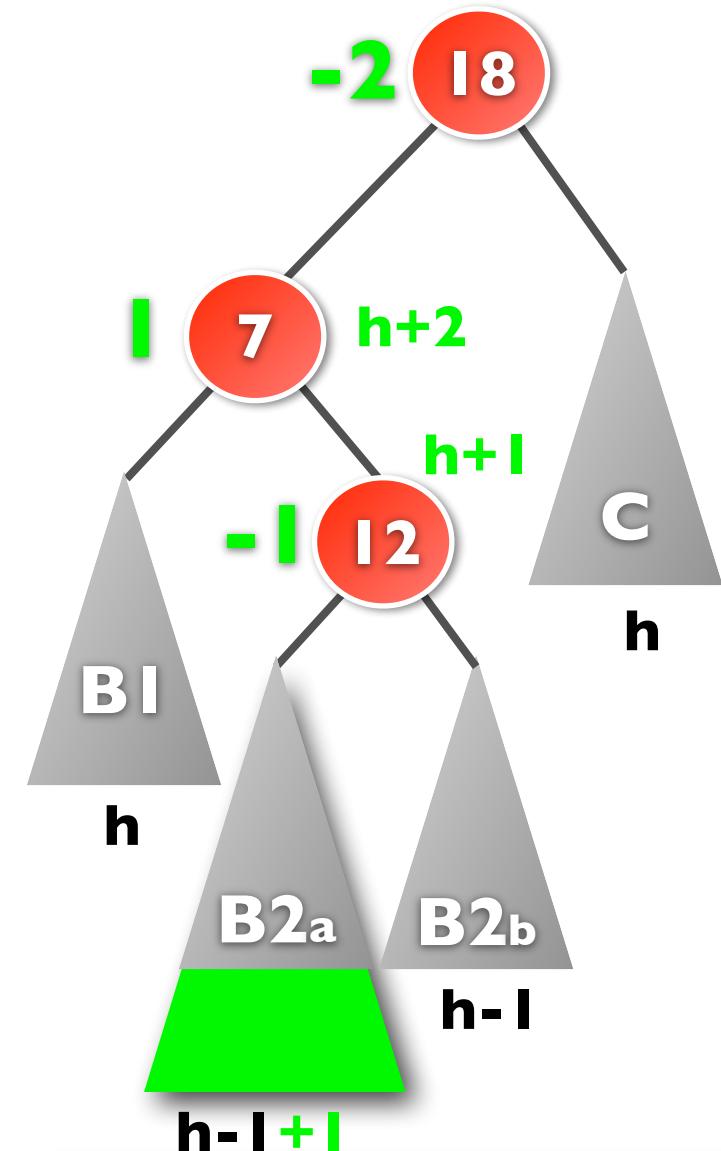
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



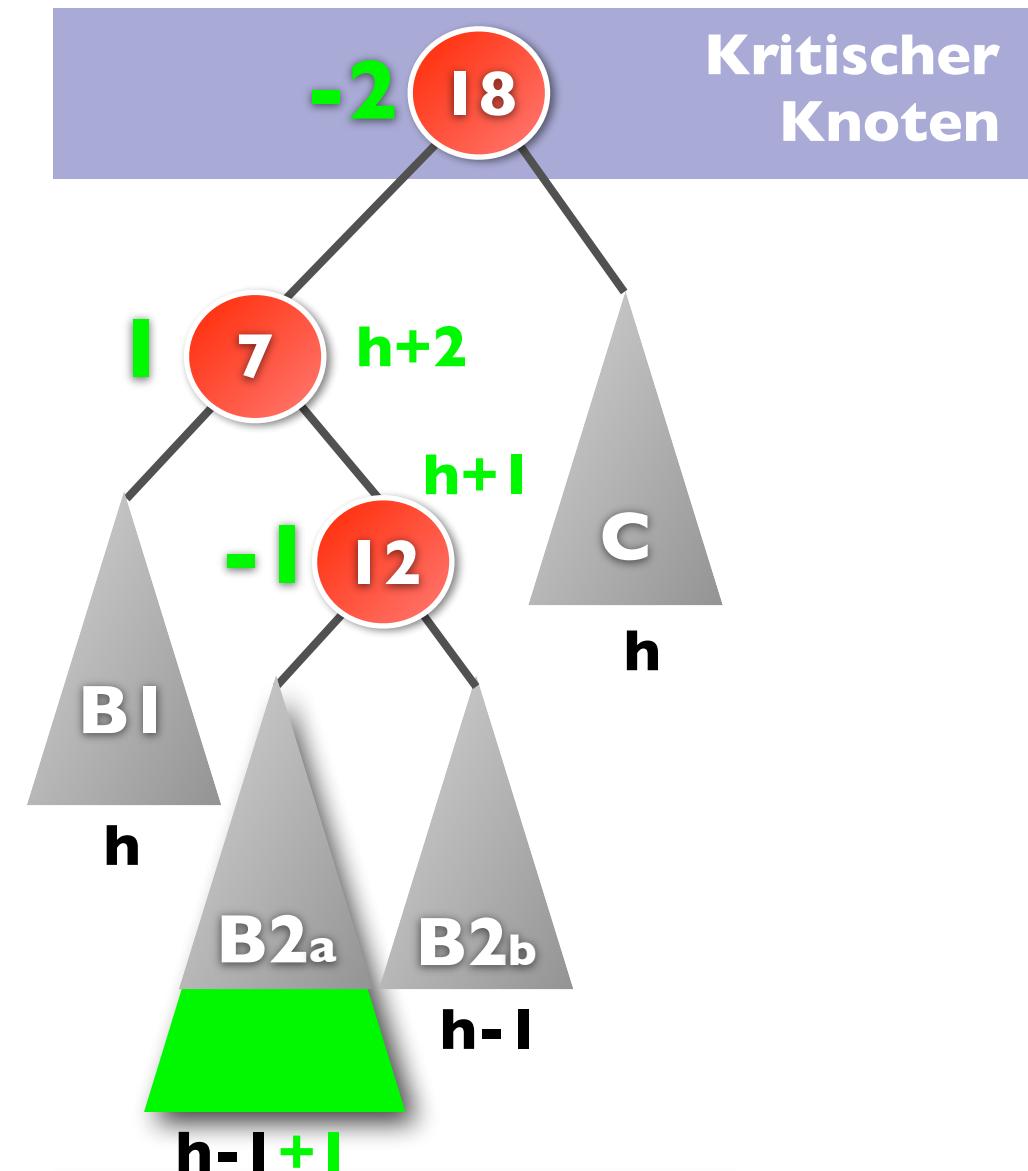
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



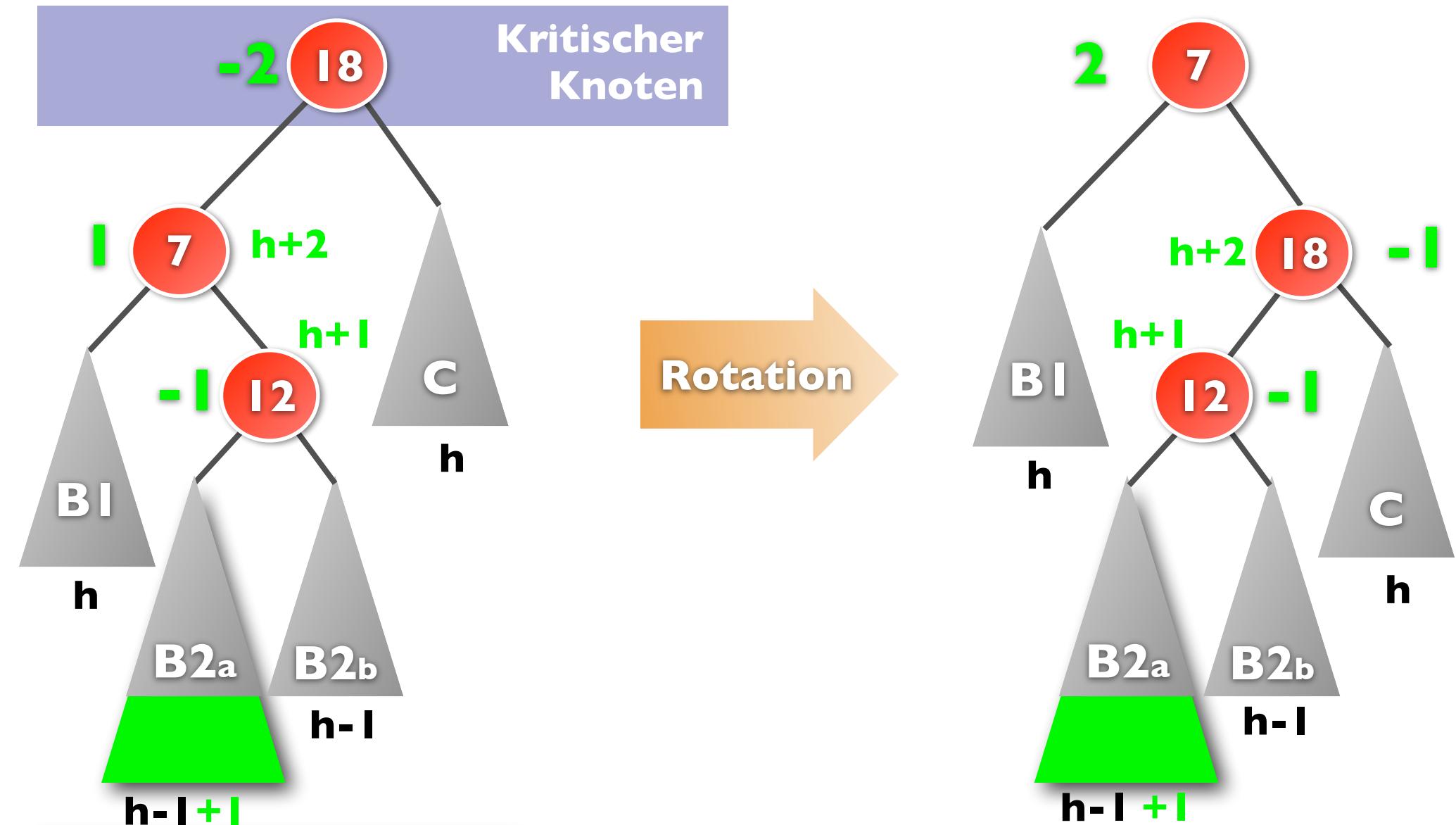
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



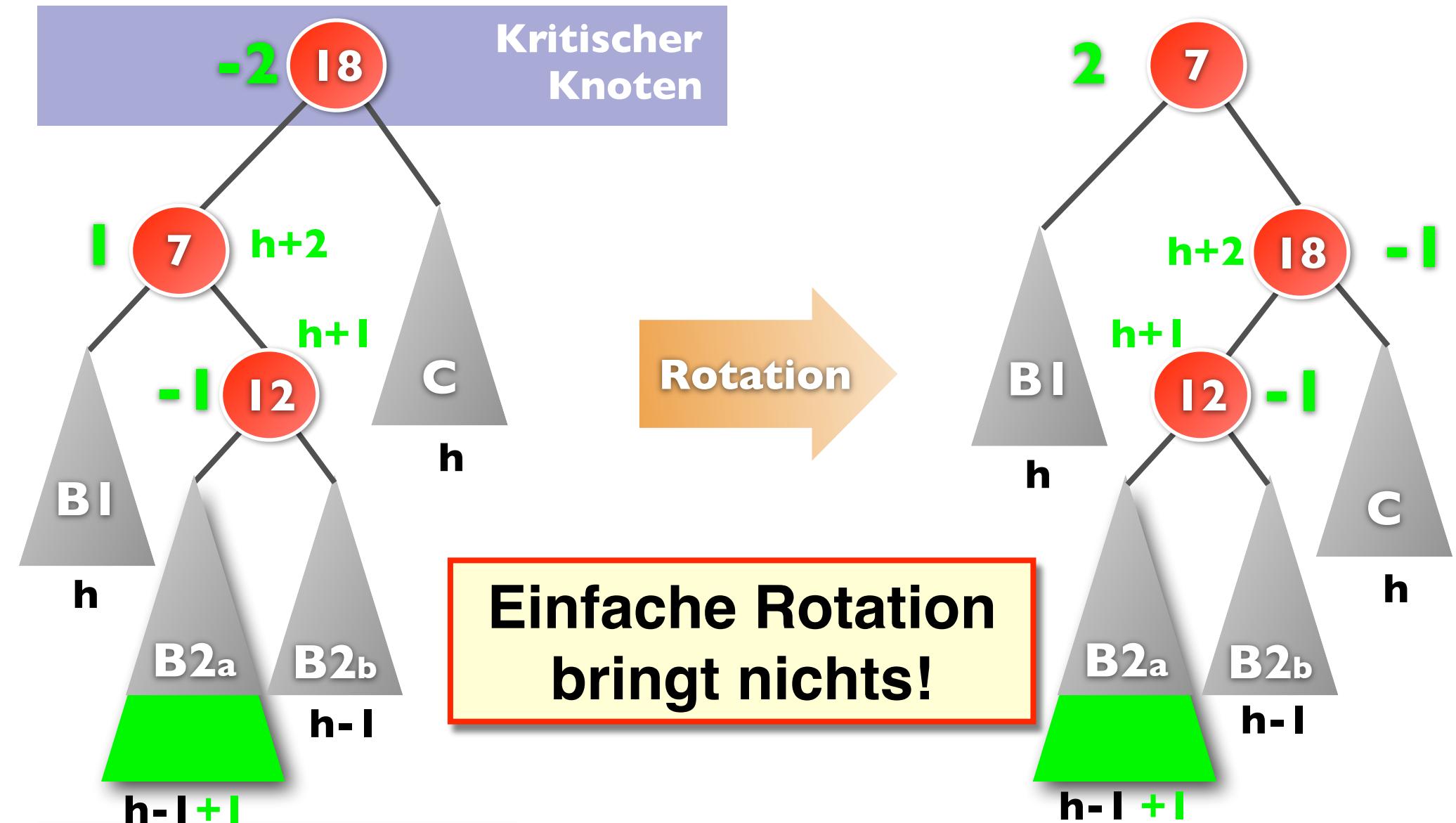
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



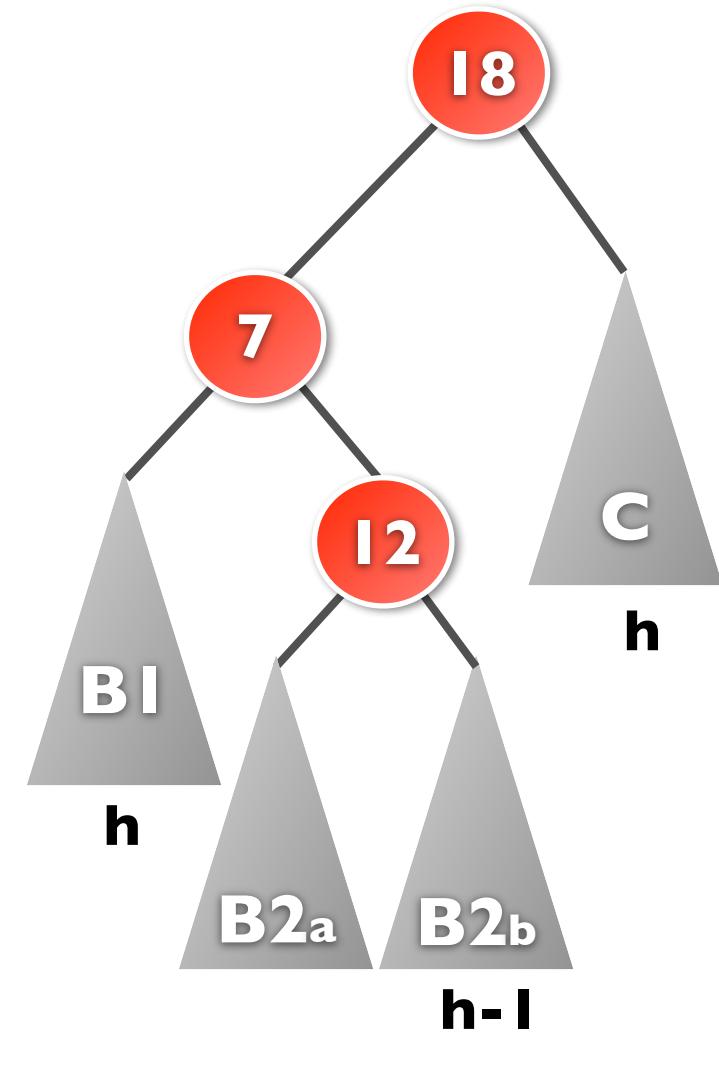
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



# RL-Rotation (Doppelrotation)

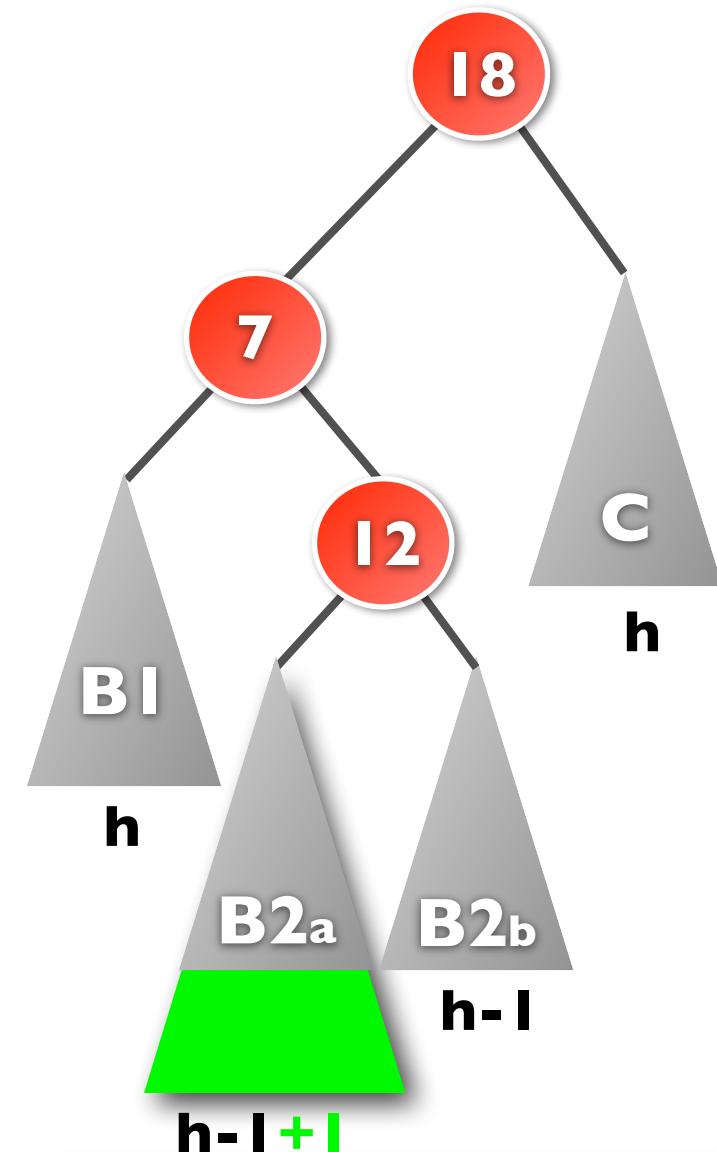
- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



**h-l**

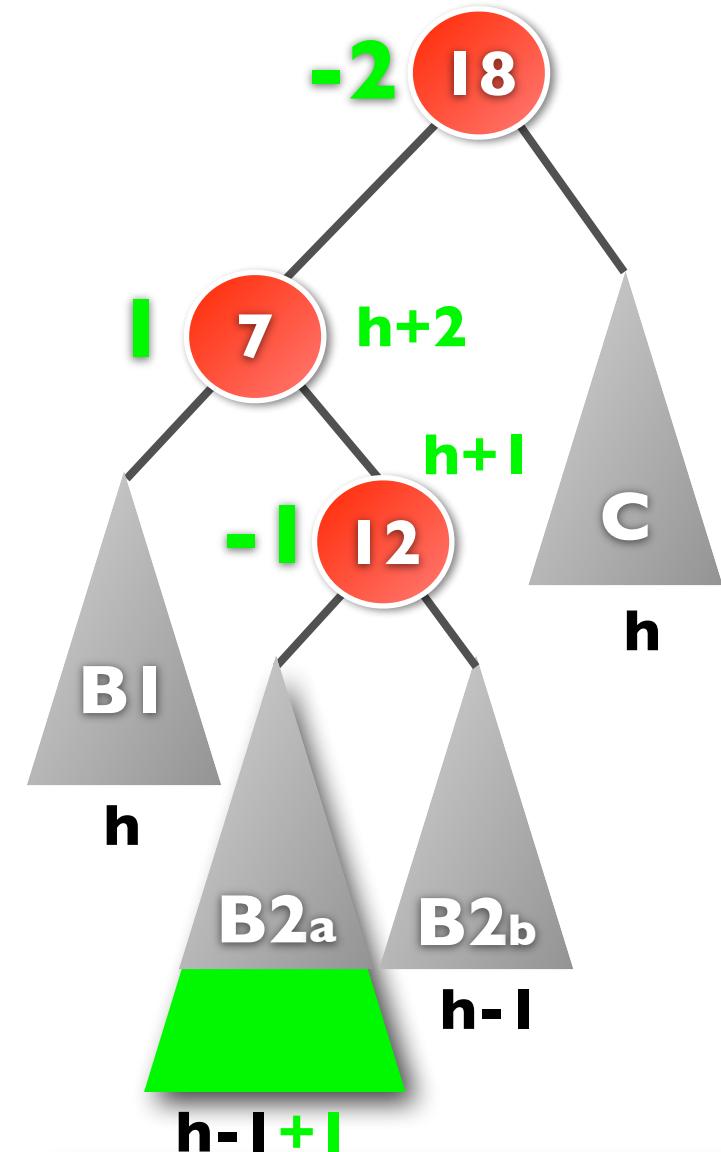
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



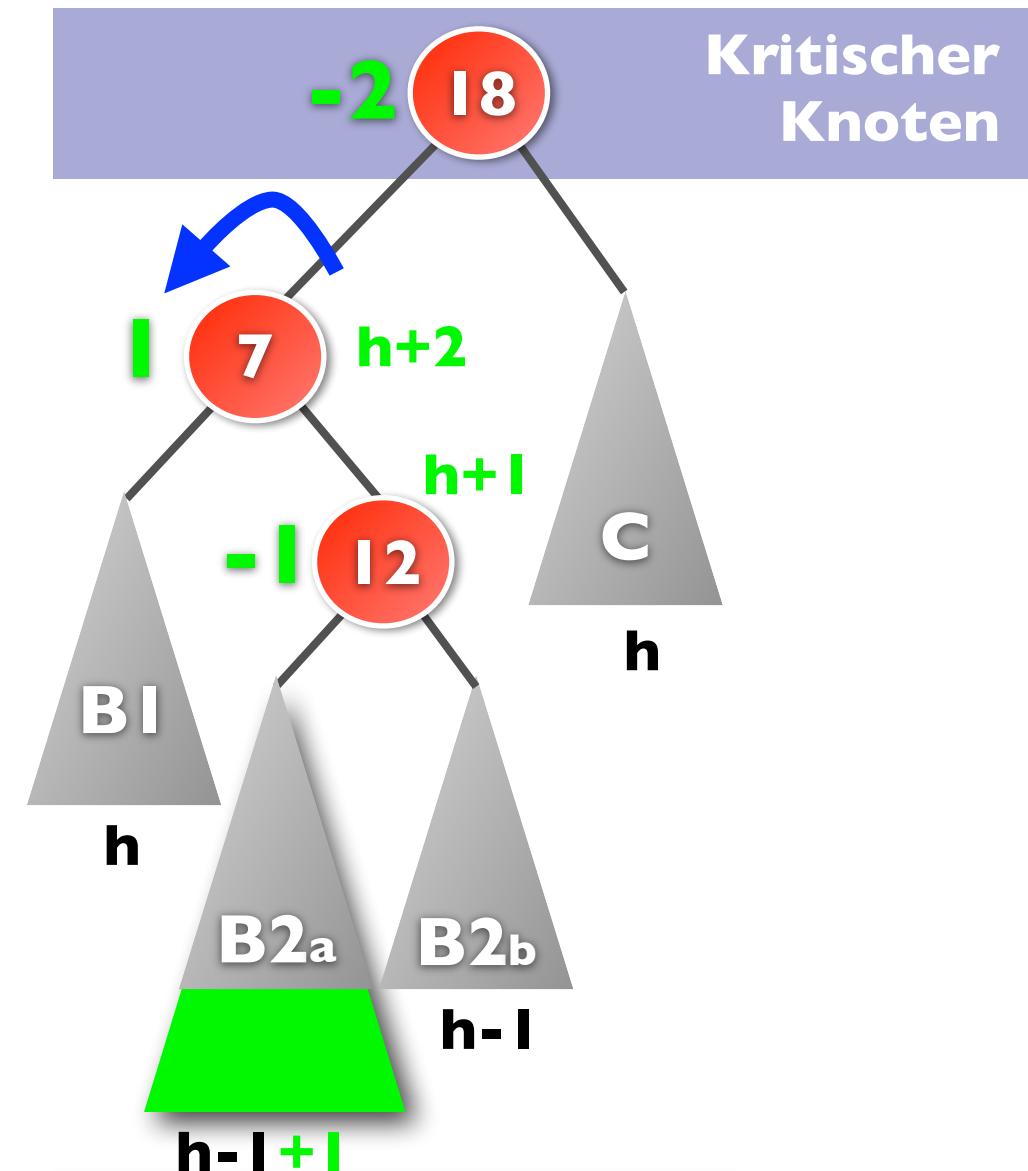
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



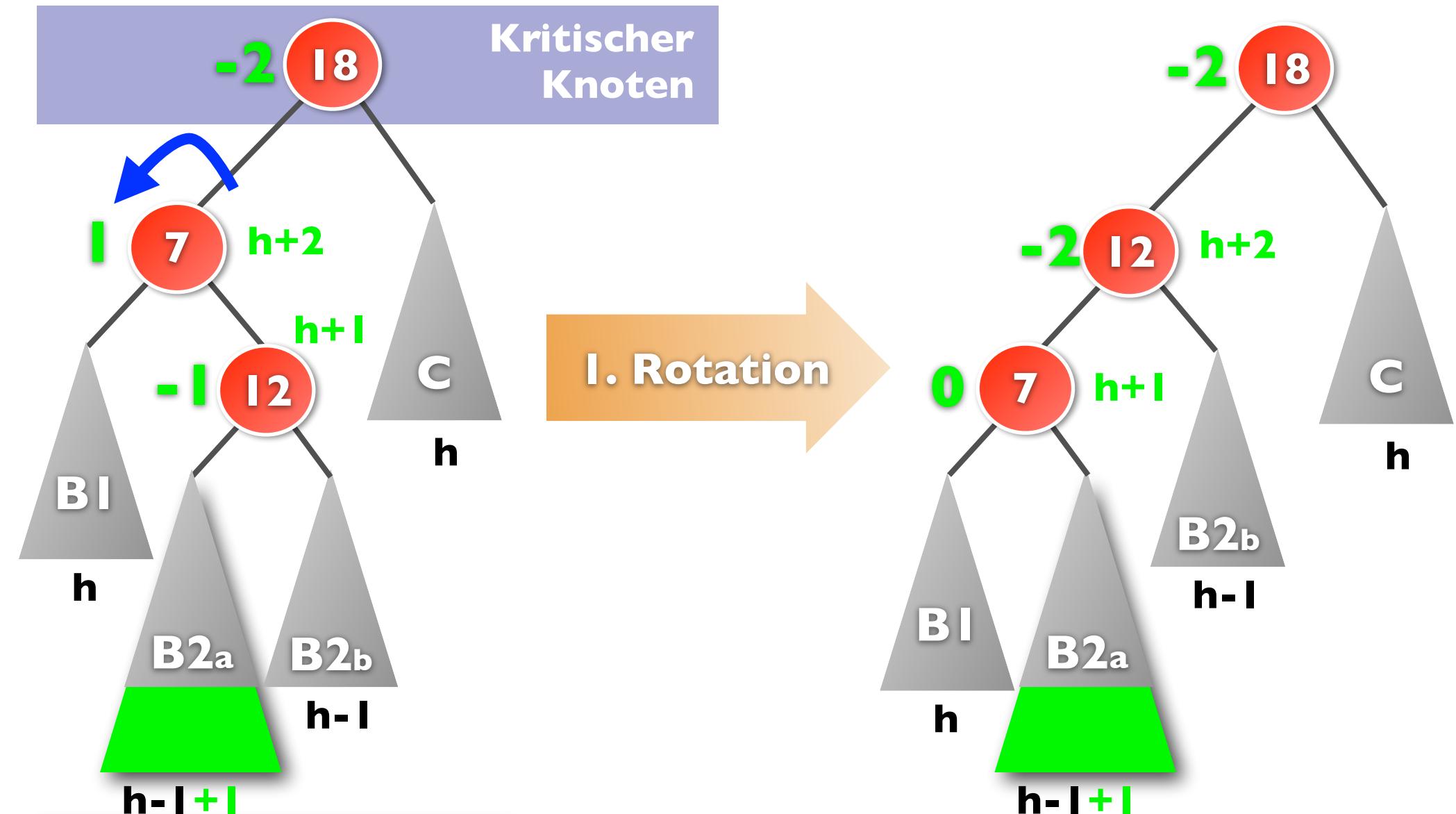
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



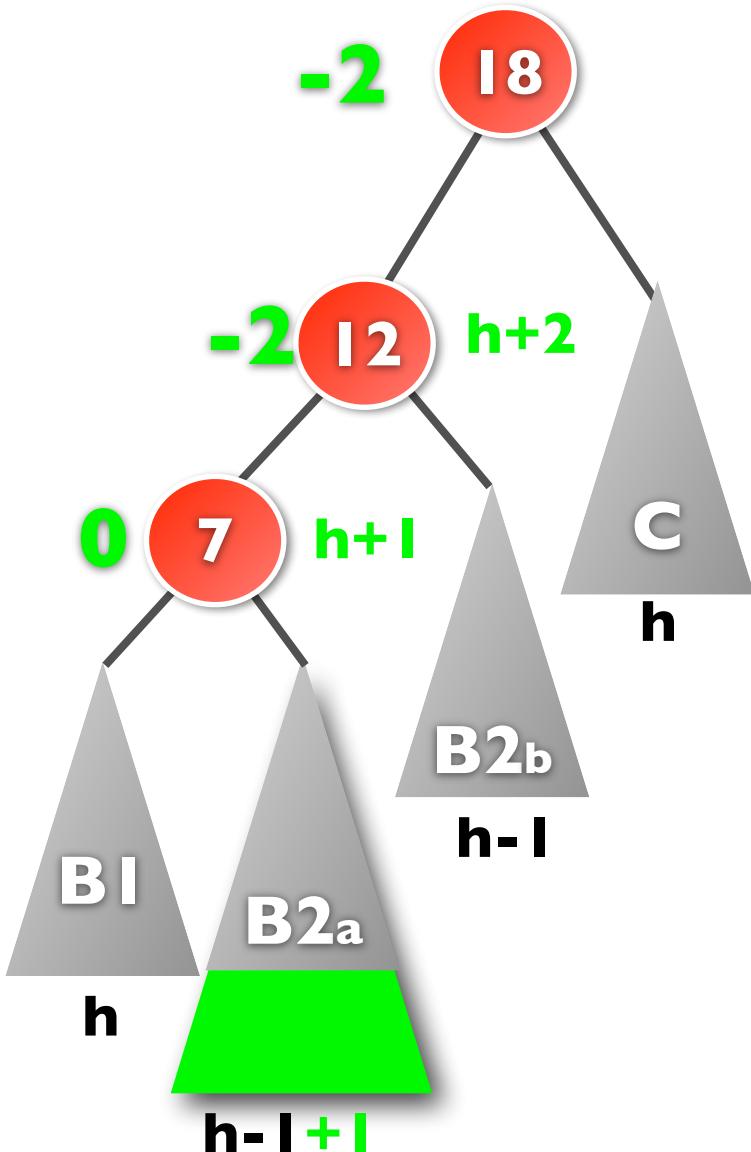
# RL-Rotation (Doppelrotation)

- Eine Rotation reicht **nicht** aus, da der problematische Teilbaum innen liegt.



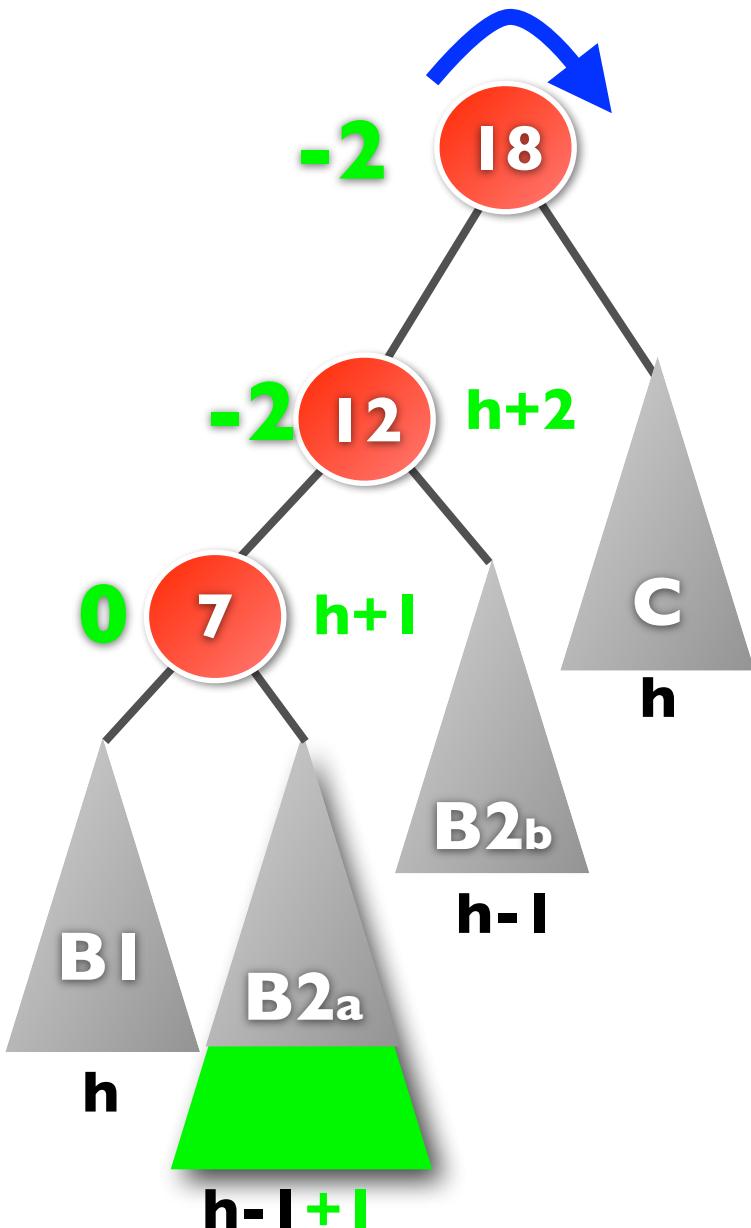
# RL-Rotation (Doppelrotation)

- Die zweite Rotation gleicht den Baum aus:



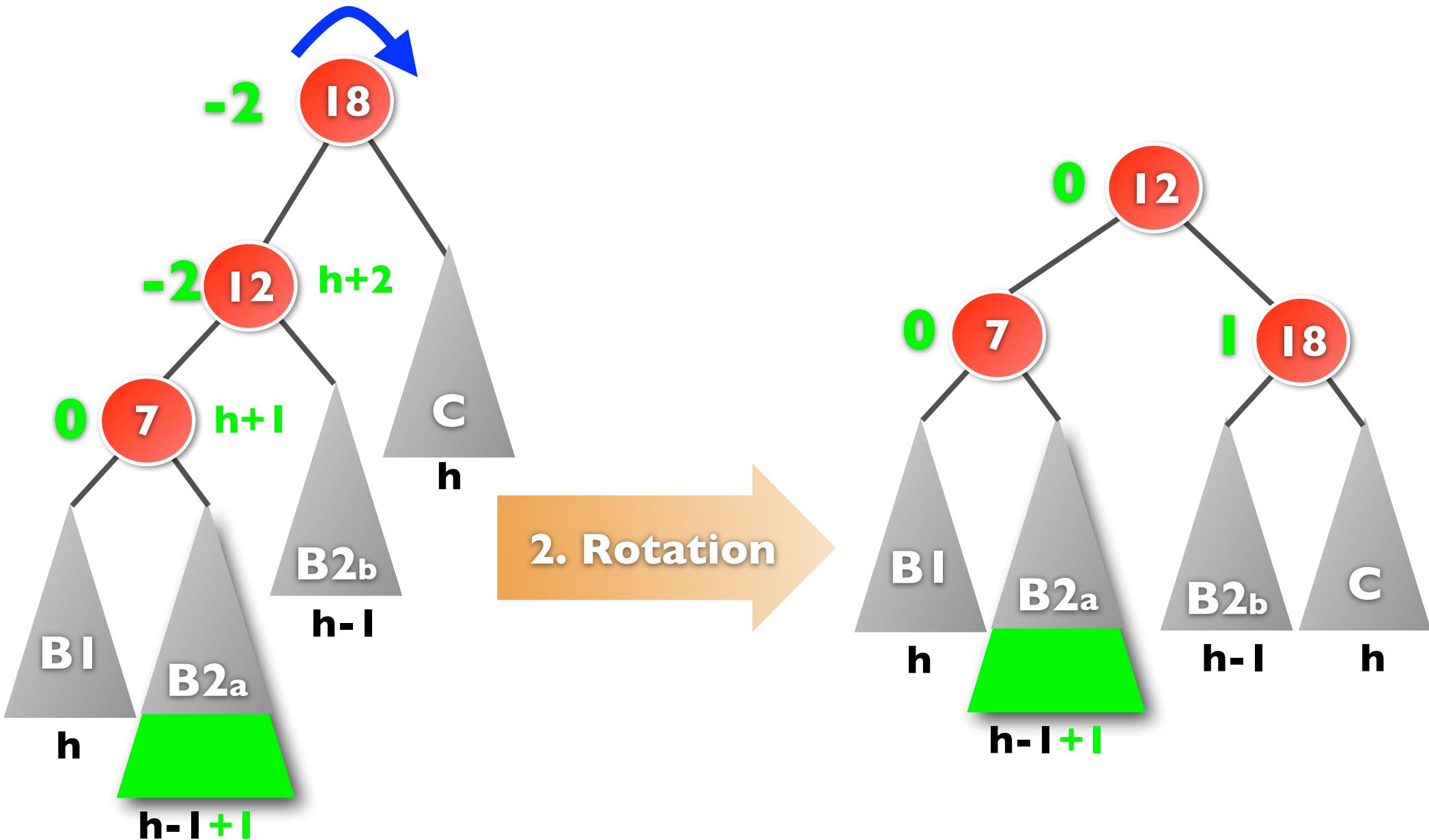
# RL-Rotation (Doppelrotation)

- Die zweite Rotation gleicht den Baum aus:



# RL-Rotation (Doppelrotation)

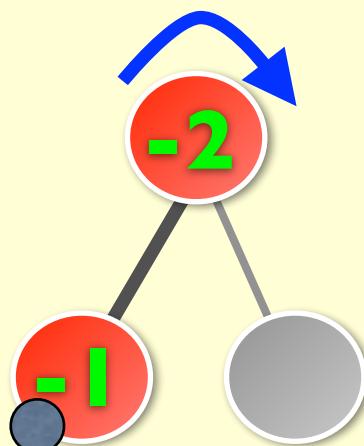
- Die zweite Rotation gleicht den Baum aus:



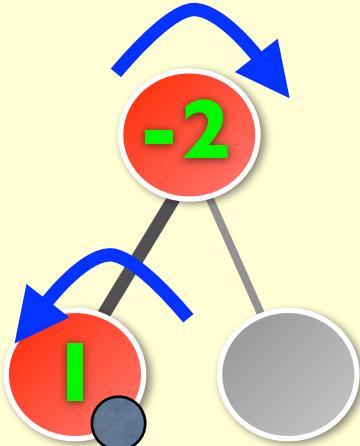
- **Das Löschen lässt sich analog zum Einfügen realisieren:**
  - Eigentliches Löschen: wie Binärbaum; dann:
  - kritischen Knoten bestimmen (nächster Vorgänger zum tatsächlich entfernten Knoten mit  $b = \pm 2$ )
  - kritischer Knoten ist Ausgangspunkt der Reorganisation (=Rotation)
  - Rotationstyp wird bestimmt, als ob im **gegenüberliegenden Unterbaum** ein Knoten eingefügt worden wäre  
*(siehe nächste Folie)*

# Übersicht: Wann wie rotieren?

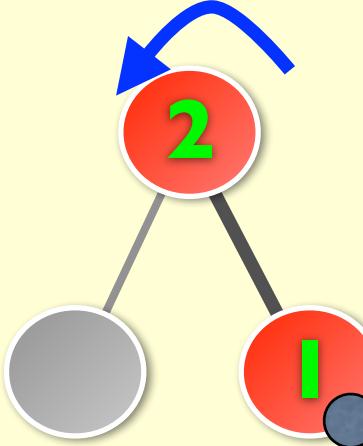
## Einfügen



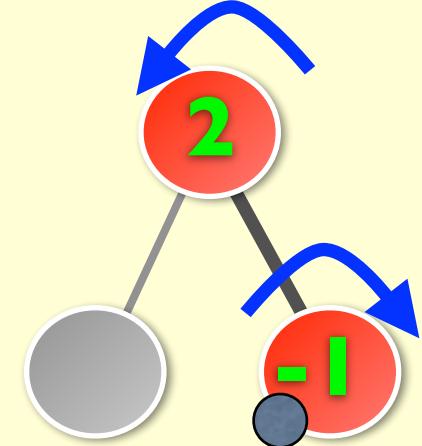
**LL-einfach**



**RL-doppelt**



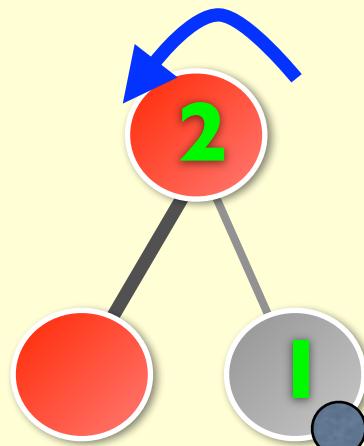
**RR-einfach**



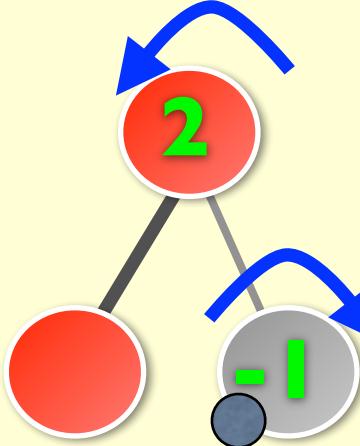
**LR-doppelt**

**Rot** markierte Knoten wurden beim Einfügen / Löschen besucht. **Einfügen**: Bei „Rückweg“ entscheiden Balancen vom kritischen Knoten und dessen Vorgänger (auf Rückweg) über Rotationstyp. **Löschen**: Hier entscheiden Balancen des kritischen Knotens und des Bruderknotens des Vorgängers über Rotationstyp - so als ob man im benachbarten Teilbaum eingefügt hätte.

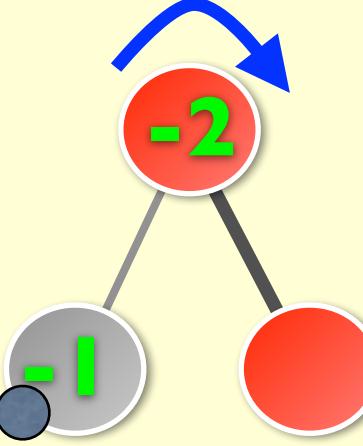
## Löschen



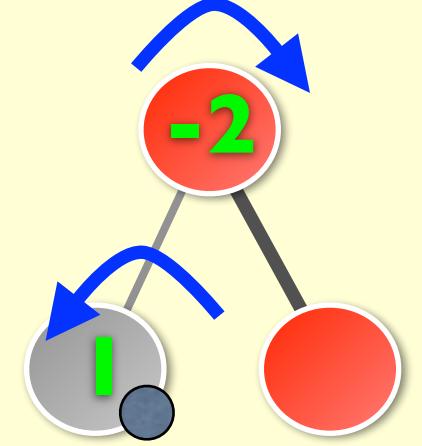
**RR-einfach**



**LR-doppelt**



**LL-einfach**



**RL-doppelt**

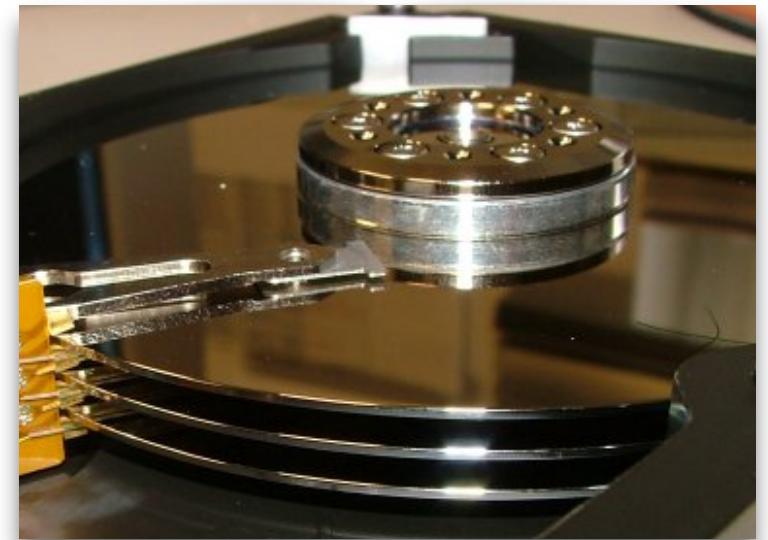
# V. Suchen in Mengen

## 5. Suchen in Mengen

- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume**
  - 5.5.1. Gewichtsbalanceierte Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume**
  - 5.5.4. rot/schwarz-Bäume
- 5.6. *Priority Queue* und *Heap*

# $(a,b)$ -Bäume - Motivation

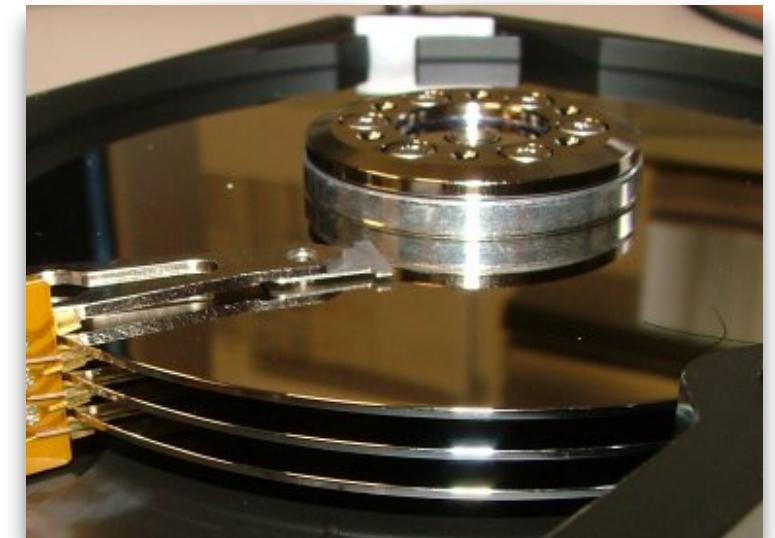
- **Problem: Externe Suche**
  - Externe Speichermedien (z.B. Festplatte / Band) bieten deutlich mehr Platz als der Hauptspeicher (das RAM) eines Computers



# $(a,b)$ -Bäume - Motivation

- **Problem: Externe Suche**

- Externe Speichermedien (z.B. Festplatte / Band) bieten deutlich mehr Platz als der Hauptspeicher (das RAM) eines Computers
- Die zu durchsuchenden Daten sind daher meist extern gespeichert - man spricht dann von **externer Suche**



# $(a,b)$ -Bäume - Motivation

- **Problem: Externe Suche**

- Externe Speichermedien (z.B. Festplatte / Band) bieten deutlich mehr Platz als der Hauptspeicher (das RAM) eines Computers
- Die zu durchsuchenden Daten sind daher meist extern gespeichert - man spricht dann von **externer Suche**
- (wahlfreier) Zugriff auf externe Speichermedien ist **langsam**
  - **Grund:** bewegliche Teile: Platten und Schreib-/Lesekopf
  - **Problem:** Zugriffszeit dominiert u.U. die Rechenzeit eines Verfahrens
  - **Daher:** Zahl der Speicherzugriffe bei Analyse berücksichtigen



# $(a,b)$ -Bäume - Motivation

- **Problem: Externe Suche**

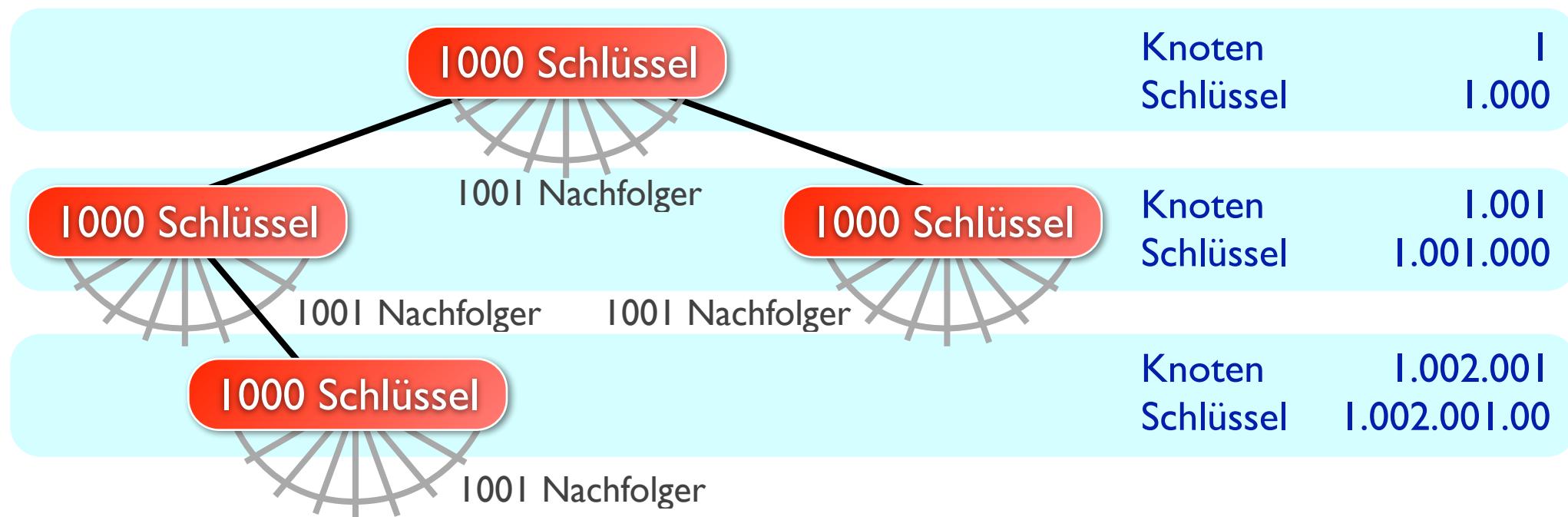
- Externe Speichermedien (z.B. Festplatte / Band) bieten deutlich mehr Platz als der Hauptspeicher (das RAM) eines Computers
- Die zu durchsuchenden Daten sind daher meist extern gespeichert - man spricht dann von **externer Suche**
- (wahlfreier) Zugriff auf externe Speichermedien ist **langsam**
  - **Grund:** bewegliche Teile: Platten und Schreib-/Lesekopf
  - **Problem:** Zugriffszeit dominiert u.U. die Rechenzeit eines Verfahrens
  - **Daher:** Zahl der Speicherzugriffe bei Analyse berücksichtigen
- Lese-/Schreibzugriff erfolgen blockweise
  - → finde Datenstruktur die **blockorientierten Zugriff** unterstützt



# $(a,b)$ -Bäume - Motivation

- **Idee: Mehrwegbäume**

- Speichere nicht nur einen Schlüssel in einem Knoten, sondern mindestens so viele, wie in einem Block Platz finden:



- Ein Knoten kann mit **einem Platten-/Bandzugriff** geladen werden
- Schlüssel sind innerhalb des Knotens **sortiert**  
**somit:** binäres Suchen möglich!

# Definition Mehrwegsuchbaum

## D Mehrwegsuchbaum

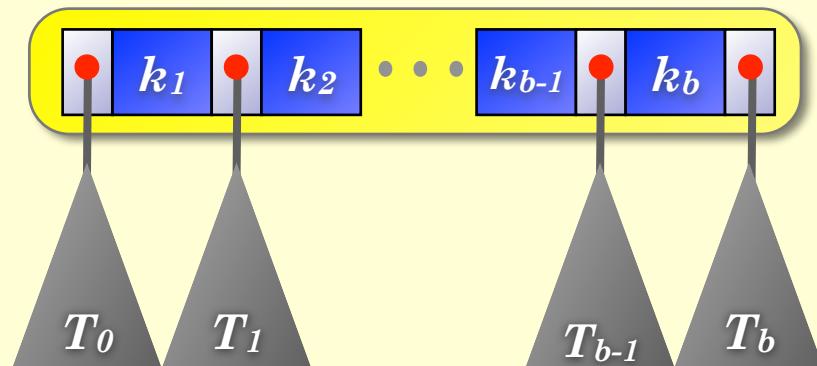
Der leere Baum ist ein **Mehrwegsuchbaum** mit der leeren Schlüsselmenge. Seien  $T_0, \dots, T_b$  Mehrwegsuchbäume mit Schlüsselmengen  $\overline{T_0}, \dots, \overline{T_b}$  und sei  $k_1, \dots, k_b$  eine Menge von Schlüsseln mit

$$k_1 < k_2 < \dots < k_b$$

Dann ist die Folge

$$T_0 \ k_1 \ T_1 \ k_2 \ T_2 \ k_3 \ \dots \ k_b \ T_b$$

bzw. grafisch



ein Mehrwegsuchbaum, wenn gilt:

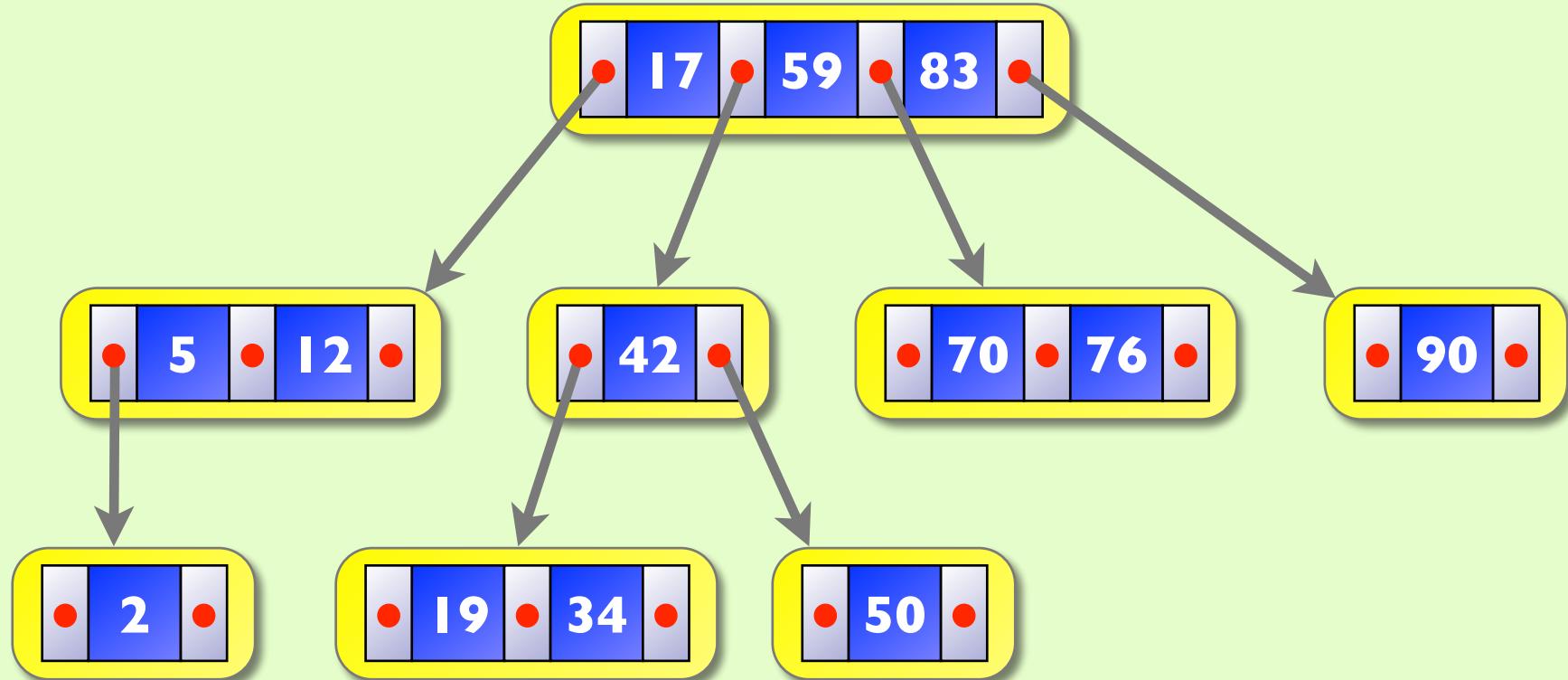
$$\forall x \in \overline{T_0} : x < k_1$$

$$\forall x \in \overline{T_b} : x > k_b$$

$$\forall x \in \overline{T_i} : k_i < x < k_{i+1} \quad \text{für } 1 < i < b$$

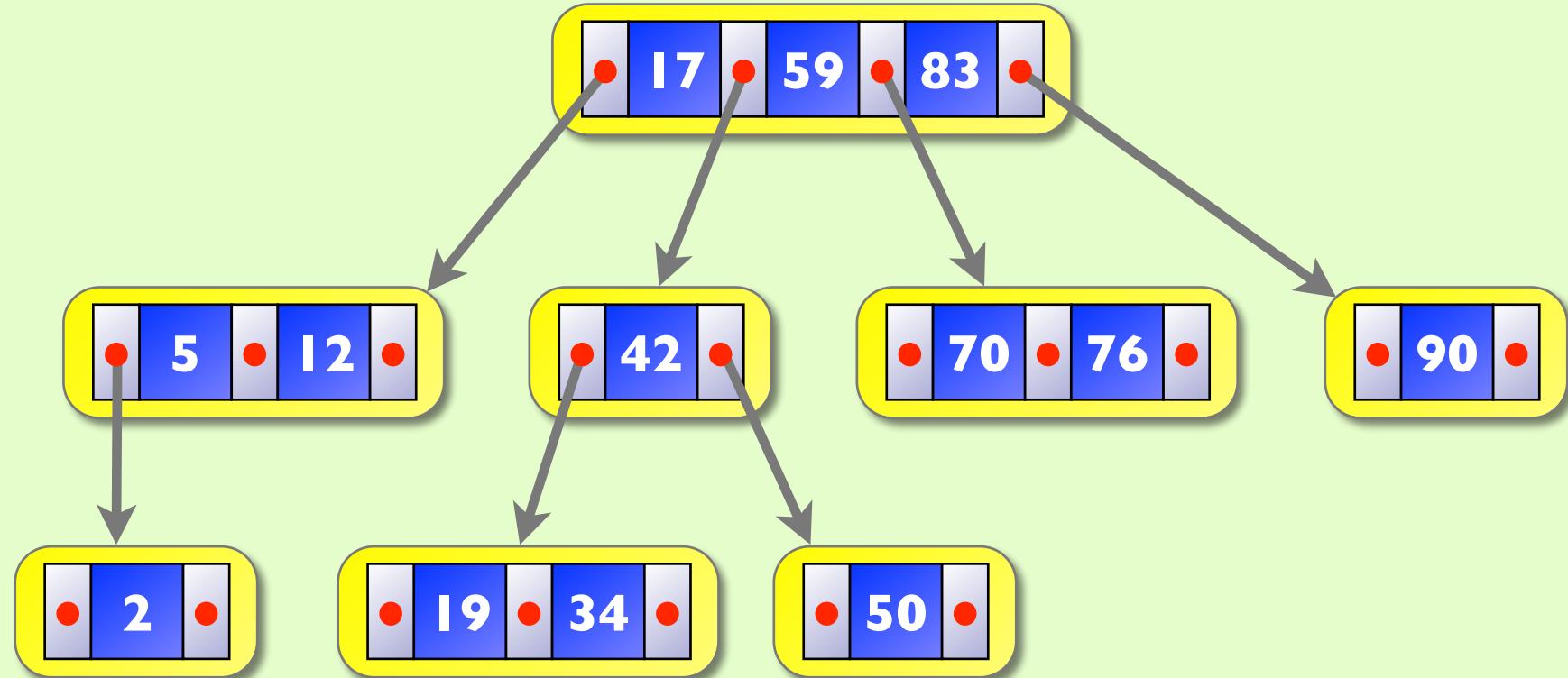
# Beispiel: Mehrwegsuchbaum

## B Ein Mehrwegsuchbaum



# Beispiel: Mehrwegsuchbaum

## B Ein Mehrwegsuchbaum



- **Beachte:**
  - Knoten enthalten **mehrere** Schlüssel
  - Schlüssel sind innerhalb eines Knotens **sortiert**
  - Schlüssel definieren **Intervallgrenzen**

# Definition $(a,b)$ Bäume

## D **$(a,b)$ -Baum und B-Baum**

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2$  und  $b \geq 2a - 1$ .  $\rho(v)$  sei die Anzahl der Söhne des Knotens  $v$ . Ein Mehrwegsuchbaum heißt  **$(a,b)$ -Baum**, wenn

1. alle Blätter die gleiche Tiefe haben,
2. für jeden Knoten  $v$  gilt:  $\rho(v) \leq b$ ,
3. für alle Knoten  $v$  (außer der Wurzel)  $\rho(v) \geq a$  gilt und
4. die Wurzel mindestens zwei Söhne hat.

Gilt  $b = 2a - 1$ , dann spricht man von einem **B-Baum der Ordnung  $a - 1$** .

# Definition $(a,b)$ Bäume

Begründung folgt!

## D **$(a,b)$ -Baum und B-Baum**

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2$  und  $b \geq 2a - 1$ .  $\rho(v)$  sei die Anzahl der Söhne des Knotens  $v$ . Ein Mehrwegsuchbaum heißt  **$(a,b)$ -Baum**, wenn

1. alle Blätter die gleiche Tiefe haben,
2. für jeden Knoten  $v$  gilt:  $\rho(v) \leq b$ ,
3. für alle Knoten  $v$  (außer der Wurzel)  $\rho(v) \geq a$  gilt und
4. die Wurzel mindestens zwei Söhne hat.

Gilt  $b = 2a - 1$ , dann spricht man von einem **B-Baum der Ordnung  $a - 1$** .

# Definition $(a,b)$ Bäume

Begründung folgt!

## D $(a,b)$ -Baum und B-Baum

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2$  und  $b \geq 2a - 1$ .  $\rho(v)$  sei die Anzahl der Söhne des Knotens  $v$ . Ein Mehrwegsuchbaum heißt  **$(a,b)$ -Baum**, wenn

1. alle Blätter die gleiche Tiefe haben,
2. für jeden Knoten  $v$  gilt:  $\rho(v) \leq b$ ,
3. für alle Knoten  $v$  (außer der Wurzel)  $\rho(v) \geq a$  gilt und
4. die Wurzel mindestens zwei Söhne hat.

Gilt  $b = 2a - 1$ , dann spricht man von einem **B-Baum der Ordnung  $a - 1$** .

- **$a$** : minimaler **Knotengrad** (minimale Zahl der Kindknoten)
- **$b$** : maximaler **Knotengrad** (maximale Zahl der Kindknoten)
- **Ordnung**: minimale **Schlüsselzahl**

# Definition $(a,b)$ Bäume

Begründung folgt!

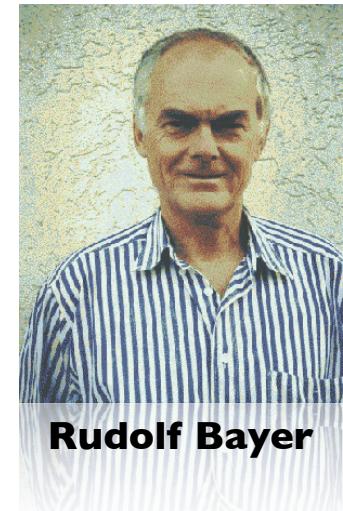
## D $(a,b)$ -Baum und B-Baum

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2$  und  $b \geq 2a - 1$ .  $\rho(v)$  sei die Anzahl der Söhne des Knotens  $v$ . Ein Mehrwegsuchbaum heißt  **$(a,b)$ -Baum**, wenn

1. alle Blätter die gleiche Tiefe haben,
2. für jeden Knoten  $v$  gilt:  $\rho(v) \leq b$ ,
3. für alle Knoten  $v$  (außer der Wurzel)  $\rho(v) \geq a$  gilt und
4. die Wurzel mindestens zwei Söhne hat.

Gilt  $b = 2a - 1$ , dann spricht man von einem **B-Baum der Ordnung  $a - 1$** .

- **$a$** : minimaler **Knotengrad** (minimale Zahl der Kindknoten)
- **$b$** : maximaler **Knotengrad** (maximale Zahl der Kindknoten)
- **Ordnung**: minimale **Schlüsselzahl**
- **B-Bäume** gehen zurück auf Bayer und McCraight (1972)



## D B-Baum der Ordnung $m$

Sei  $m \geq 2$  eine natürliche Zahl. Ein Mehrwegsuchbaum heißt **B-Baum der Ordnung  $m$** , wenn gilt:

1. alle Blätter haben die gleiche Tiefe,
2. alle Knoten mit Ausnahme der Wurzel enthalten mindestens  $m$  und höchstens  $2 \cdot m$  Schlüssel. Die Wurzel enthält mindestens einen und maximal  $2 \cdot m$  Schlüssel.
3. Alle inneren Knoten mit  $s$  Schlüsseln haben genau  $s + 1$  Söhne.

# Definition B-Baum der Ordnung $m$

- Es gibt alternative Definitionen für B-Bäume

## D B-Baum der Ordnung $m$

Sei  $m \geq 2$  eine natürliche Zahl. Ein Mehrwegsuchbaum heißt **B-Baum der Ordnung  $m$** , wenn gilt:

1. alle Blätter haben die gleiche Tiefe,
2. alle Knoten mit Ausnahme der Wurzel enthalten mindestens  $m$  und höchstens  $2 \cdot m$  Schlüssel. Die Wurzel enthält mindestens einen und maximal  $2 \cdot m$  Schlüssel.
3. Alle inneren Knoten mit  $s$  Schlüsseln haben genau  $s + 1$  Söhne.

# Definition B-Baum der Ordnung $m$

- Es gibt alternative Definitionen für B-Bäume

## D B-Baum der Ordnung $m$

Sei  $m \geq 2$  eine natürliche Zahl. Ein Mehrwegsuchbaum heißt **B-Baum der Ordnung  $m$** , wenn gilt:

1. alle Blätter haben die gleiche Tiefe,
2. alle Knoten mit Ausnahme der Wurzel enthalten mindestens  $m$  und höchstens  $2 \cdot m$  Schlüssel. Die Wurzel enthält mindestens einen und maximal  $2 \cdot m$  Schlüssel.
3. Alle inneren Knoten mit  $s$  Schlüsseln haben genau  $s + 1$  Söhne.

- Offenbar gilt: ein B-Baum mit der Ordnung  $m$  ist ein  $(m+1, 2m+1)$ -Baum.

# Übersicht: Definitionen für Mehrwegsuchbäume

- **Vorsicht! Die Literatur ist sehr uneinheitlich!**
  - Manchmal liegt der Definition die Zahl der Schlüssel zu Grunde (siehe z.B. Güting2004);
  - man findet aber auch Definitionen für B-Bäume, die auf der Zahl der Söhne aufbauen (sie z.B. Cormen et. al. 2003)
- Hier eine Übersicht über die gebräuchlichen Definitionen:

	<b>(a,b)-Baum</b> $(b \geq 2 \cdot a - 1)$	<b>B-Baum</b>	
		der Ordnung $m$	von minimalem Grad $k$
minimale Schlüsselzahl	$a-1$	$m$	$k-1$
maximale Schlüsselzahl	$b-1$	$2 \cdot m$	$2 \cdot k-1$
minimaler Knotengrad	$a$	$m+1$	$k$
maximaler Knotengrad	$b$	$2 \cdot m+1$	$2 \cdot k$

# Übersicht: Definitionen für Mehrwegsuchbäume

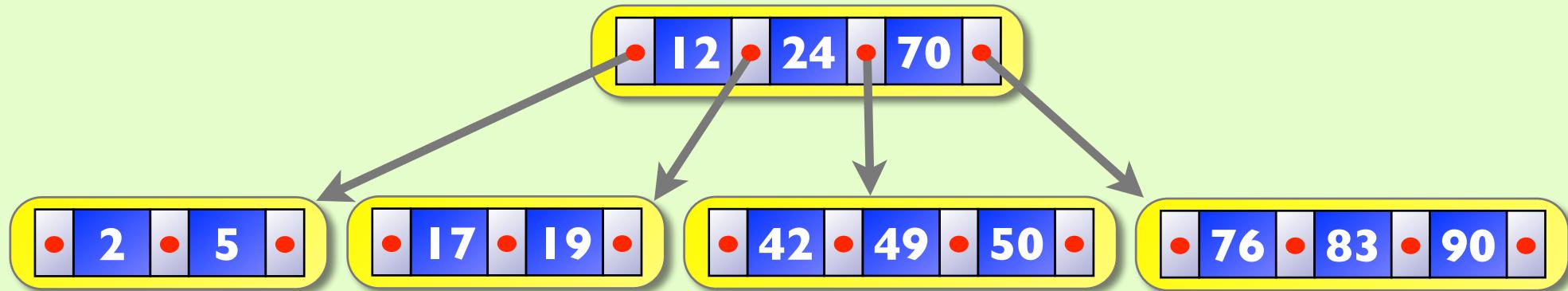
- **Vorsicht! Die Literatur ist sehr uneinheitlich!**
  - Manchmal liegt der Definition die Zahl der Schlüssel zu Grunde (siehe z.B. Güting2004);
  - man findet aber auch Definitionen für B-Bäume, die auf der Zahl der Söhne aufbauen (sie z.B. Cormen et. al. 2003)
- Hier eine Übersicht über die gebräuchlichen Definitionen:

	<b>(a,b)-Baum</b> $(b \geq 2 \cdot a - 1)$	<b>B-Baum</b> der Ordnung $m$	von minimalem Grad $k$
minimale Schlüsselzahl	$a - 1$	$m$	$k - 1$
maximale Schlüsselzahl	$b - 1$	$2 \cdot m$	$2 \cdot k - 1$
minimaler Knotengrad	$a$	$m + 1$	$k$
maximaler Knotengrad	$b$	$2 \cdot m + 1$	$2 \cdot k$

Wir betrachten nur noch B-Bäume der Ordnung  $m$

# Beispiel: B-Baum

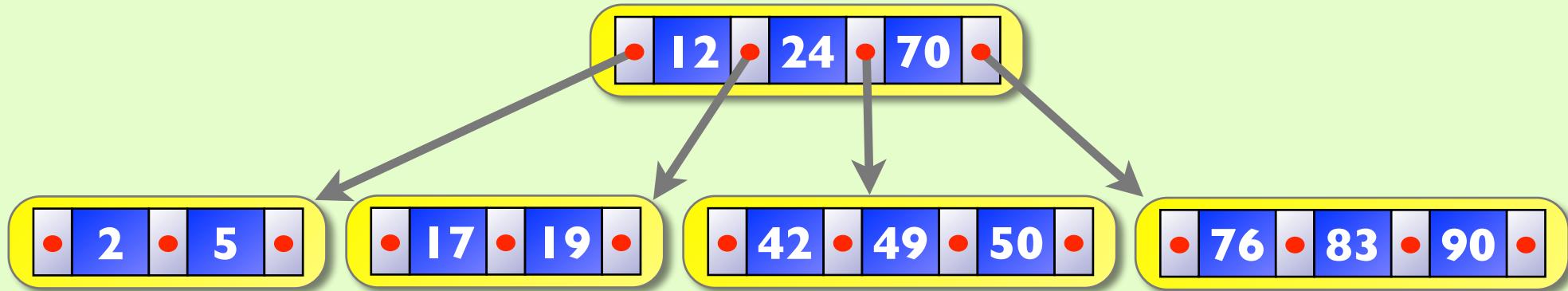
## B B-Baum der Ordnung 2



**Anmerkung:** nicht benutzte Schlüsselplätze wurden nicht dargestellt

# Beispiel: B-Baum

## B B-Baum der Ordnung 2



**Anmerkung:** nicht benutzte Schlüsselplätze wurden nicht dargestellt

- Manchmal speichert man in den Knoten nur Schlüsselwerte. Die eigentlichen Records stehen dann an den Blättern (**blattorientierte Speicherung**)
- Formal dürfen Schlüssel dann doppelt vorkommen; d.h. die Definition des Mehrwegsuchbaumes ändert sich wie folgt:

$$\forall x \in T_0 : x \leq k_1$$

$$\forall x \in T_b : x > k_1$$

$$\forall x \in T_i : k_i < x \leq k_{i+1} \quad \text{für } 1 < i < b$$

# Höhe von B-Bäumen

- Die Anzahl der Zugriffe auf den externen Speicher ist proportional zur Höhe eines B-Baumes
- Wir analysieren hier nur den *worst case*

# Höhe von B-Bäumen

- Die Anzahl der Zugriffe auf den externen Speicher ist proportional zur Höhe eines B-Baumes
- Wir analysieren hier nur den **worst case**

## S Maximale Höhe eines B-Baums der Ordnung m

Sei  $T$  ein B-Baum der Ordnung  $m$  mit insgesamt  $n$  Schlüsseln. Dann gilt:

$$\text{Höhe}(T) \leq \log_{m+1} \frac{n+1}{2}$$

# Höhe von B-Bäumen

- Die Anzahl der Zugriffe auf den externen Speicher ist proportional zur Höhe eines B-Baumes
- Wir analysieren hier nur den **worst case**

## S Maximale Höhe eines B-Baums der Ordnung m

Sei  $T$  ein B-Baum der Ordnung  $m$  mit insgesamt  $n$  Schlüsseln. Dann gilt:

$$\text{Höhe}(T) \leq \log_{m+1} \frac{n+1}{2}$$

- **Beweis:** siehe nächste Folie

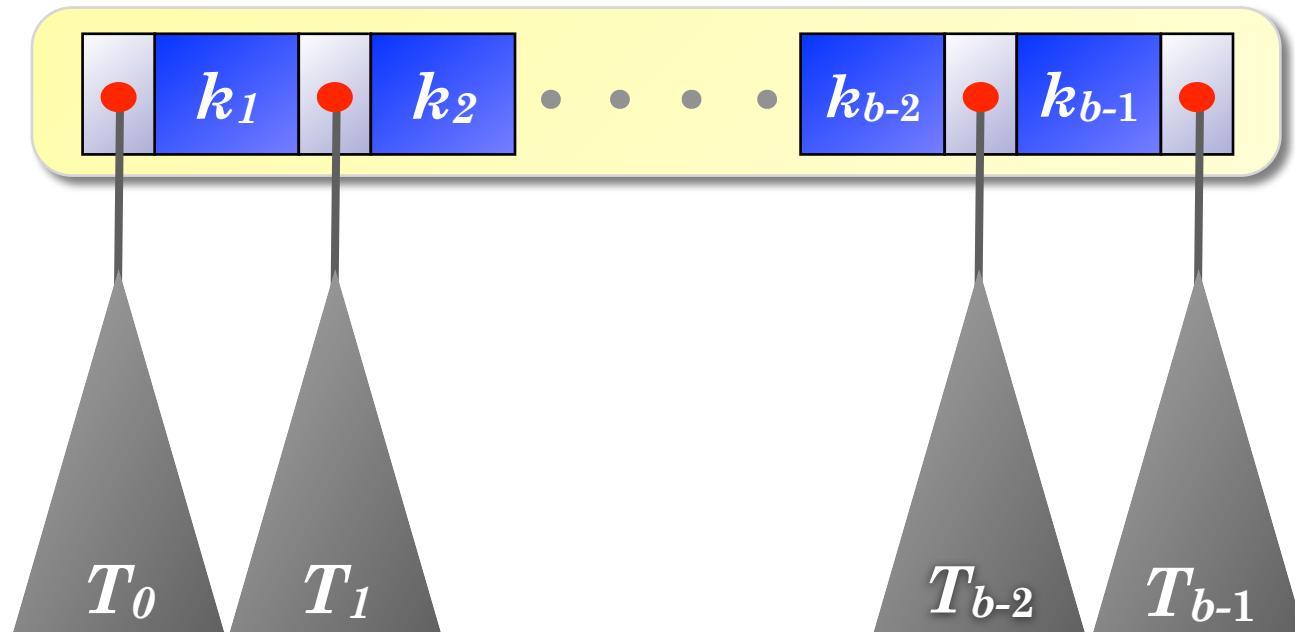
# Beweis: Maximale Höhe B-Baum

Wenn der B-Baum die  $h = \text{Höhe}(T)$  hat, so enthält die Wurzel mindestens einen Schlüssel - alle anderen Knoten haben mindestens  $m$  Schlüssel und somit  $m + 1$  Kindknoten. Folglich haben wir mindestens 2 Knoten auf Level 1;  $2 \cdot (m + 1)$  Knoten auf Level 2;  $2 \cdot (m + 1) \cdot (m + 1)$  Knoten auf Level 3 usw. bis zu Level  $h$ , wo wir  $2 \cdot (m + 1)^{h-1}$  Knoten haben. Jeder Knoten enthält minimal  $m$  Schlüssel. Daher muss die Schlüsselzahl  $n$  folgender Ungleichung genügen:

$$\begin{aligned} n &\geq 1 + m \cdot \sum_{i=1}^h 2 \cdot (m + 1)^{i-1} \\ &\geq 1 + m \cdot \sum_{i=0}^{h-1} 2 \cdot (m + 1)^i \\ &\geq 1 + 2 \cdot m \cdot \sum_{i=1}^{h-1} (m + 1)^i \\ &\geq 1 + 2 \cdot m \cdot \left( \frac{(m + 1)^h - 1}{m} \right) \\ &\geq 2 \cdot (m + 1)^h - 1 \\ \frac{n+1}{2} &\geq (m + 1)t^h \\ \log_{m+1} \frac{n+1}{2} &\geq h \end{aligned}$$

# Suchen in einem B-Baum

- **Sehr ähnlich zur binären Suche**
- **aber: innerhalb eines Knotens z.B. lineare Suche nach Intervallgrenze**
- Gesucht: Schlüssel  $k$ . Suche kleinstes  $i$  mit  $1 \leq i \leq b$  so dass  $k \leq k_i$ 
  - Entweder ist  $k_i$  der gesuchte Schlüssel,
  - oder man findet ihn, indem man den Teilbaum  $T_{i-1}$  durchsucht
- Kann kein solches  $i$  gefunden werden, so durchsucht man Teilbaum  $T_b$



# Einfügen in einen B-Baum der Ordnung $m$

- Zunächst wird der einzufügende Schlüssel  $k$  gesucht - ist er noch nicht im Baum, so endet die Suche in einem (imaginären) Bereichsblatt

# Einfügen in einen B-Baum der Ordnung $m$

- Zunächst wird der einzufügende Schlüssel  $k$  gesucht - ist er noch nicht im Baum, so endet die Suche in einem (imaginären) Bereichsblatt
- **Zwei Situationen sind zu unterscheiden**

# Einfügen in einen B-Baum der Ordnung $m$

- Zunächst wird der einzufügende Schlüssel  $k$  gesucht - ist er noch nicht im Baum, so endet die Suche in einem (imaginären) Bereichsblatt
- **Zwei Situationen sind zu unterscheiden**
  - ① **Der Vaterknoten kann noch einen Schlüssel aufnehmen:**  
Schlüssel unter Beachtung der Sortierung in Vaterknoten einfügen.

# Einfügen in einen B-Baum der Ordnung $m$

- Zunächst wird der einzufügende Schlüssel  $k$  gesucht - ist er noch nicht im Baum, so endet die Suche in einem (imaginären) Bereichsblatt
- **Zwei Situationen sind zu unterscheiden**

## ① Der Vaterknoten kann noch einen Schlüssel aufnehmen:

Schlüssel unter Beachtung der Sortierung in Vaterknoten einfügen.

## ② Der Vaterknoten $v$ ist voll (Überlauf):

Der Vaterknoten enthält  $2m$  Schlüssel. Wir bilden die sortierte Folge  $k_1, k_2, \dots, k_m, k_{m+1}, k_{m+2}, \dots, k_{2m}, k_{2m+1}$

und **teilen** diese an der Stelle  $m+1 = \lceil (2m+1)/2 \rceil$  (**Median**), um einen neuen Teilbaum zu bilden:

$T_l \ k_{m+1} T_r$  mit dem linken Teilbaum  $T_l$  und dem rechten Teilbaum  $T_r$ :

# Einfügen in einen B-Baum der Ordnung $m$

- Zunächst wird der einzufügende Schlüssel  $\textcolor{brown}{k}$  gesucht - ist er noch nicht im Baum, so endet die Suche in einem (imaginären) Bereichsblatt
- **Zwei Situationen sind zu unterscheiden**

## ① Der Vaterknoten kann noch einen Schlüssel aufnehmen:

Schlüssel unter Beachtung der Sortierung in Vaterknoten einfügen.

## ② Der Vaterknoten $v$ ist voll (Überlauf):

Der Vaterknoten enthält  $2m$  Schlüssel. Wir bilden die sortierte Folge  $k_1, k_2, \dots, k_m, \textcolor{brown}{k}_{m+1}, k_{m+2}, \dots, k_{2m}, k_{2m+1}$

und **teilen** diese an der Stelle  $\textcolor{blue}{m+1} = \lceil (2m+1)/2 \rceil$  (**Median**), um einen neuen Teilbaum zu bilden:

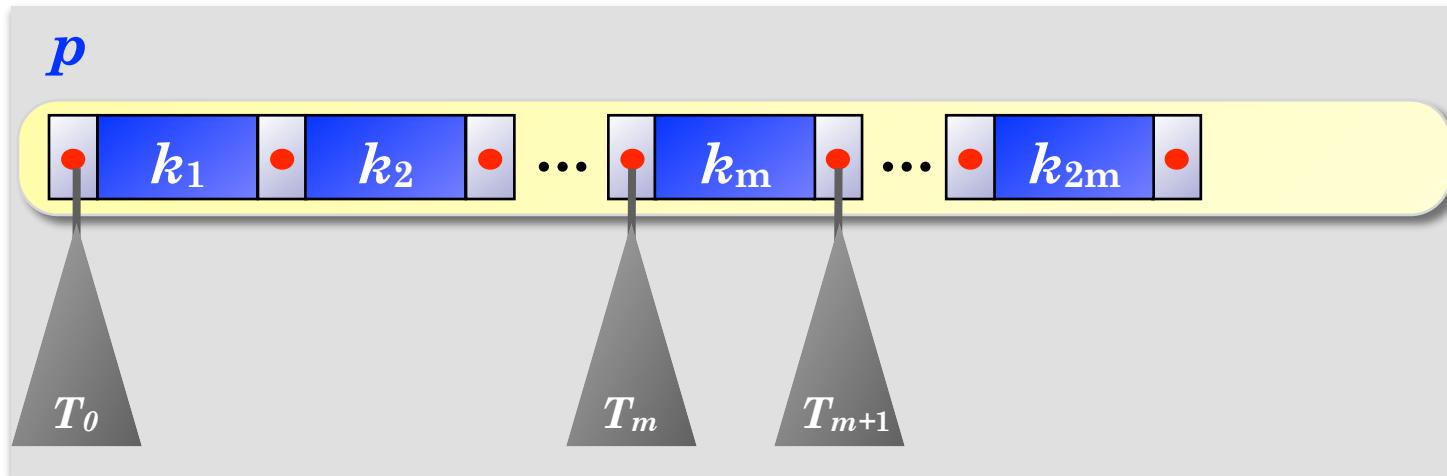
$T_l \textcolor{brown}{k}_{m+1} T_r$  mit dem linken Teilbaum  $T_l$  und dem rechten Teilbaum  $T_r$ :

- $T_l = T_0 \ k_1 \ T_1 \ \dots \ T_{m-1} \ k_m \ T_m$
- $T_r = T_{m+1} \ k_{m+2} \ T_{m+2} \ \dots \ T_{2m} \ k_{2m+1} \ T_{2m+1}$

Den Knoten  $k_{m+1}$  fügen wir nun in den Vaterknoten von  $v$  ein, der wieder überlaufen kann, usf. bis wir evtl. die Wurzel aufteilen müssen und ein  $k_{m+1}$  neue Wurzel wird

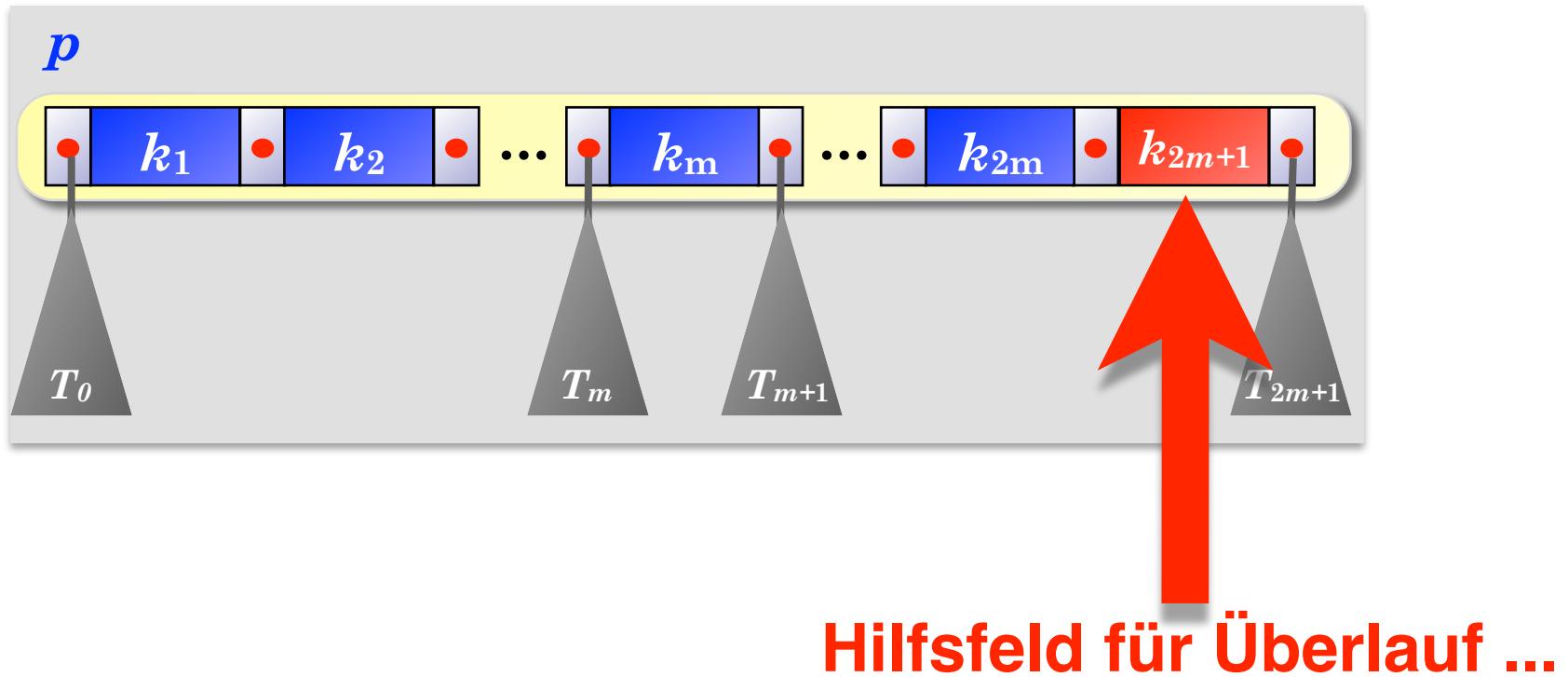
# Implementierungsdetail:

- Im Sinne einer einfachen Implementierung / Darstellung gehen wir davon aus, dass ein Knoten vorübergehend  $2m+1$  Schlüssel aufnehmen kann!
  - interne Speicherung: maximal  $2m+1$  Schlüssel (vorübergehend)
  - externe Speicherung: maximal  $2m$  Schlüssel



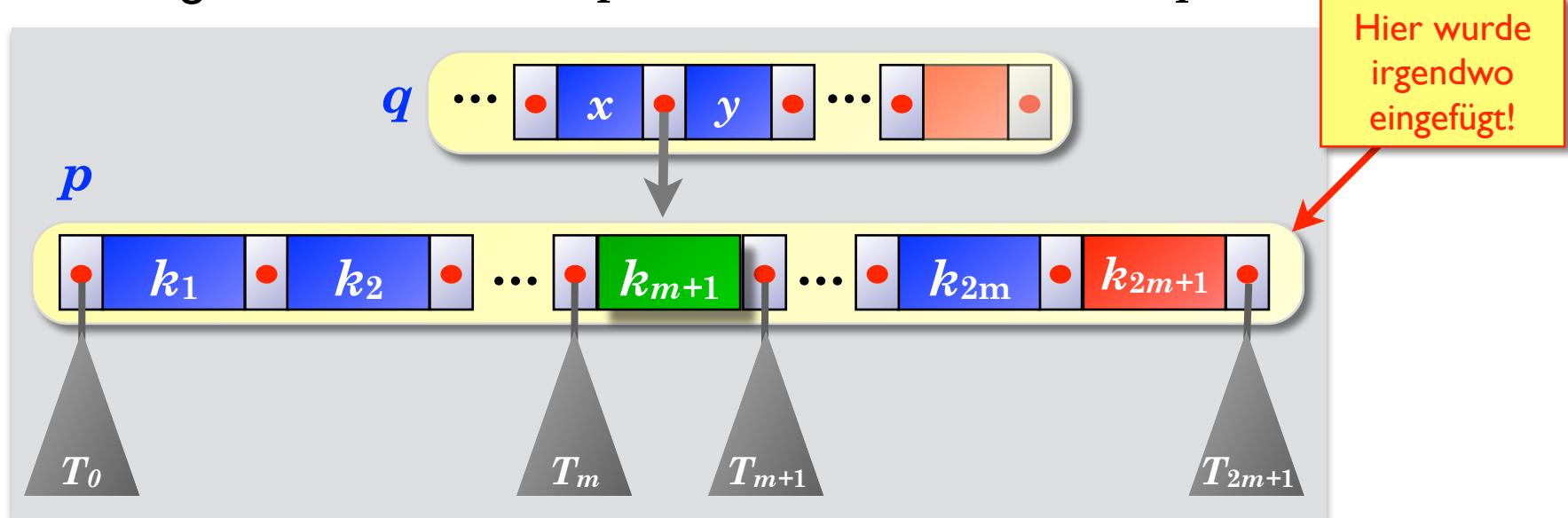
# Implementierungsdetail:

- Im Sinne einer einfachen Implementierung / Darstellung gehen wir davon aus, dass ein Knoten vorübergehend  $2m+1$  Schlüssel aufnehmen kann!
  - interne Speicherung: maximal  $2m+1$  Schlüssel (vorübergehend)
  - externe Speicherung: maximal  $2m$  Schlüssel



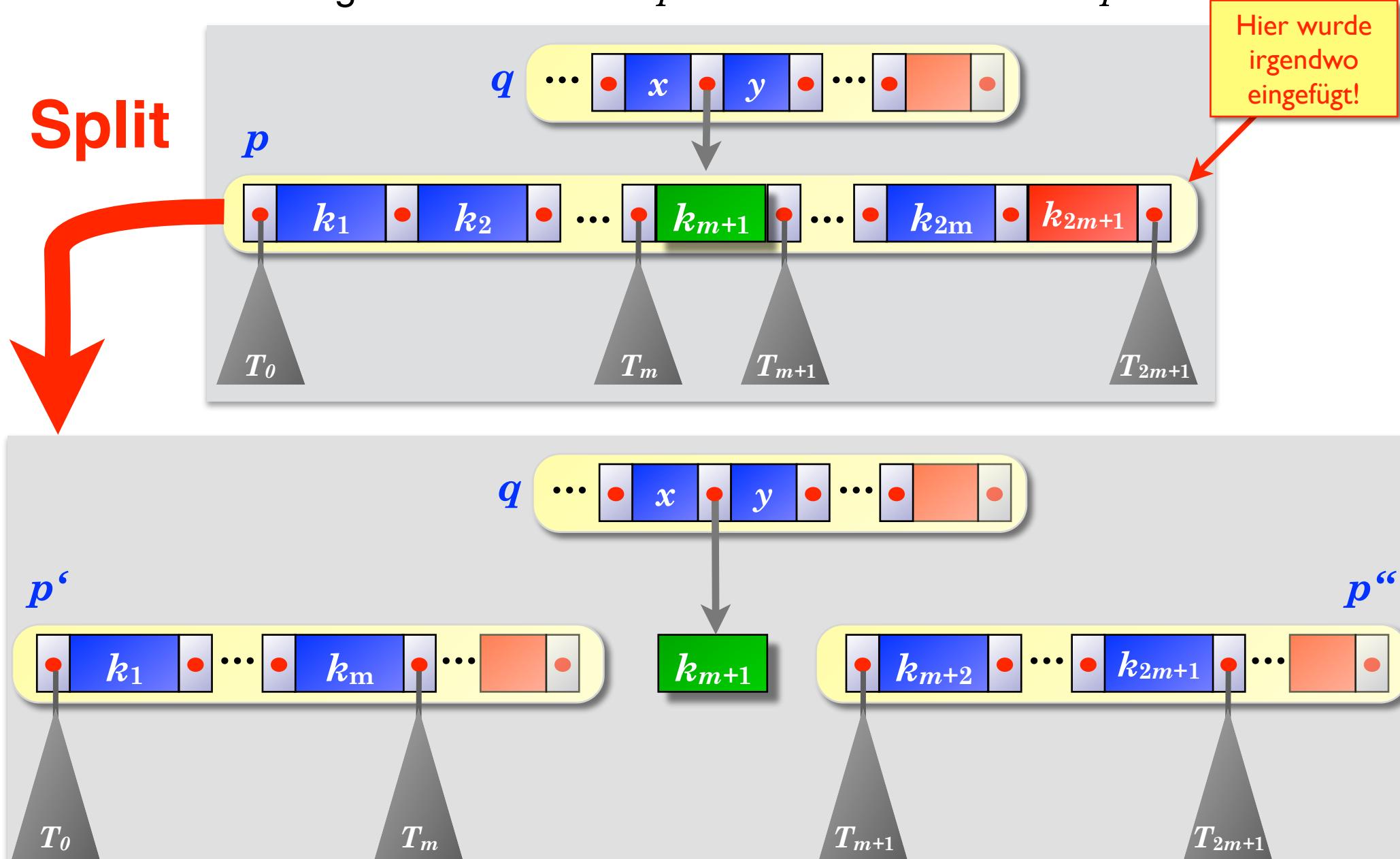
# B-Baum: Teilung bei Überlauf - 1. Fall 1

- **1. Fall:** Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



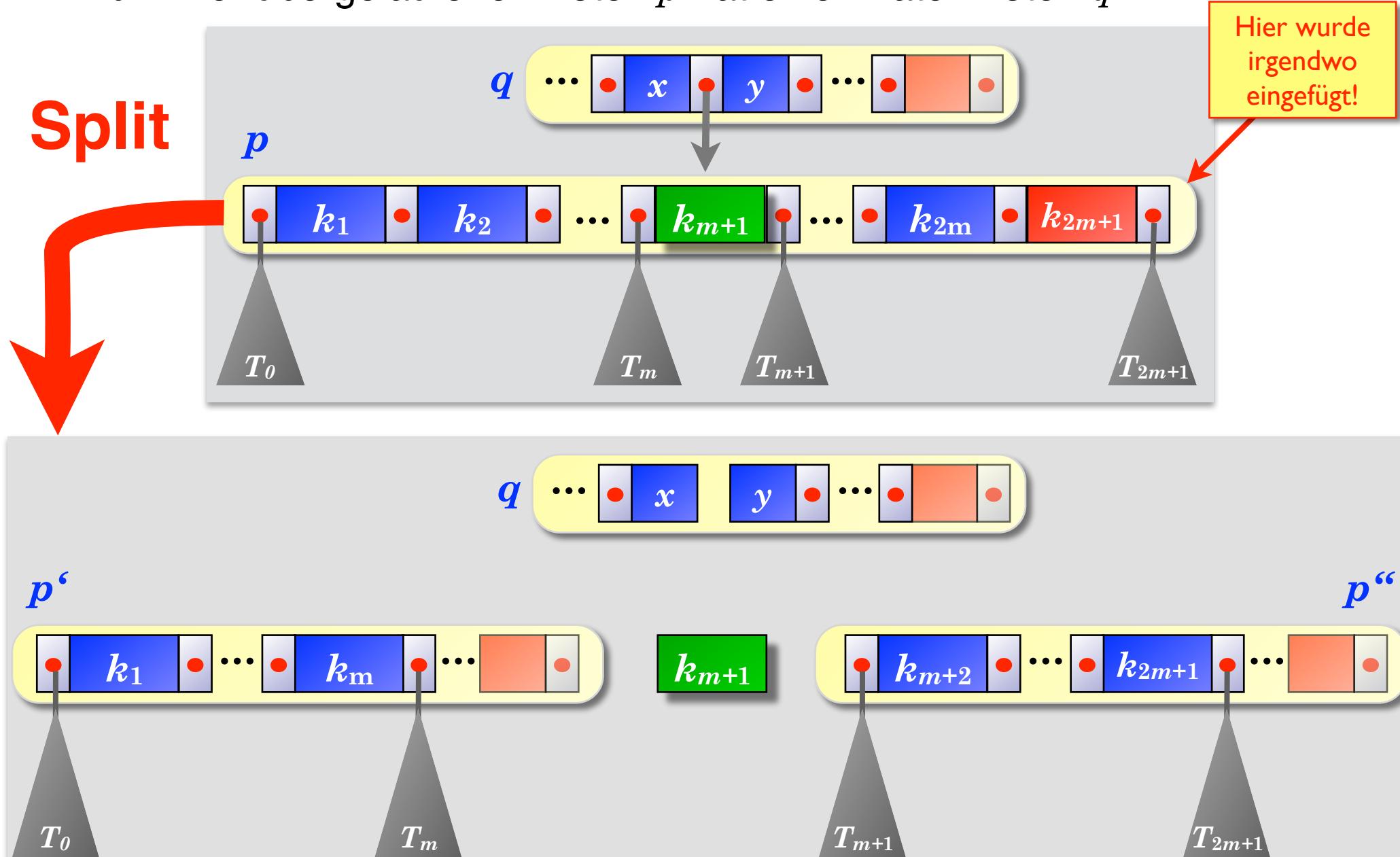
# B-Baum: Teilung bei Überlauf - 1. Fall 1

- **1. Fall:** Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



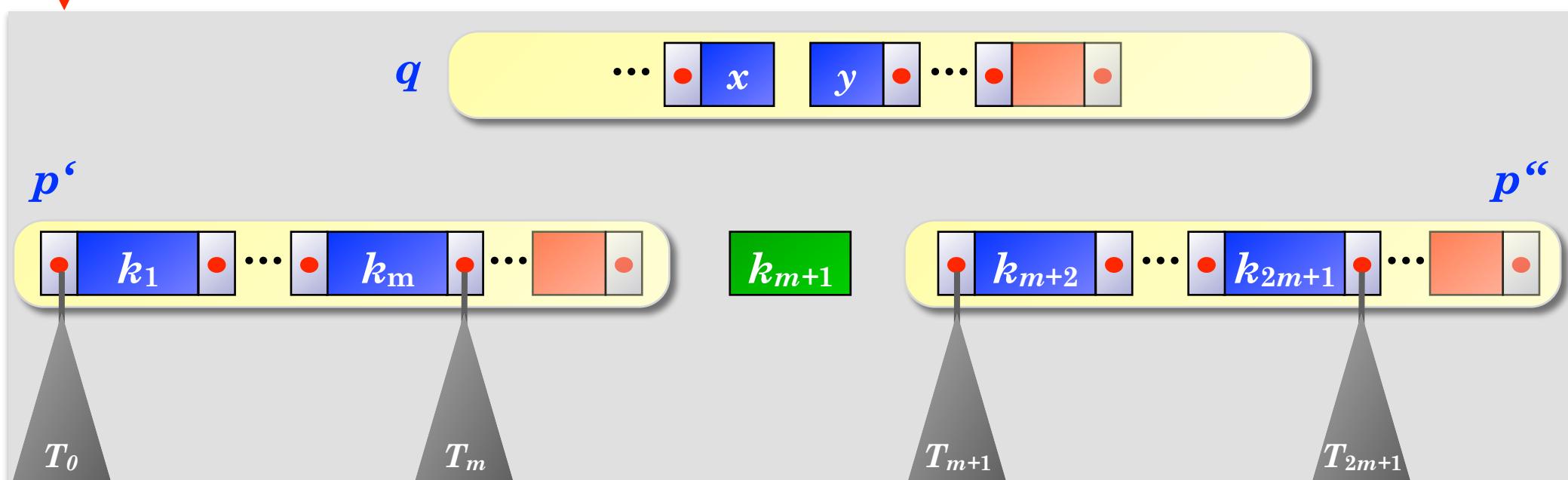
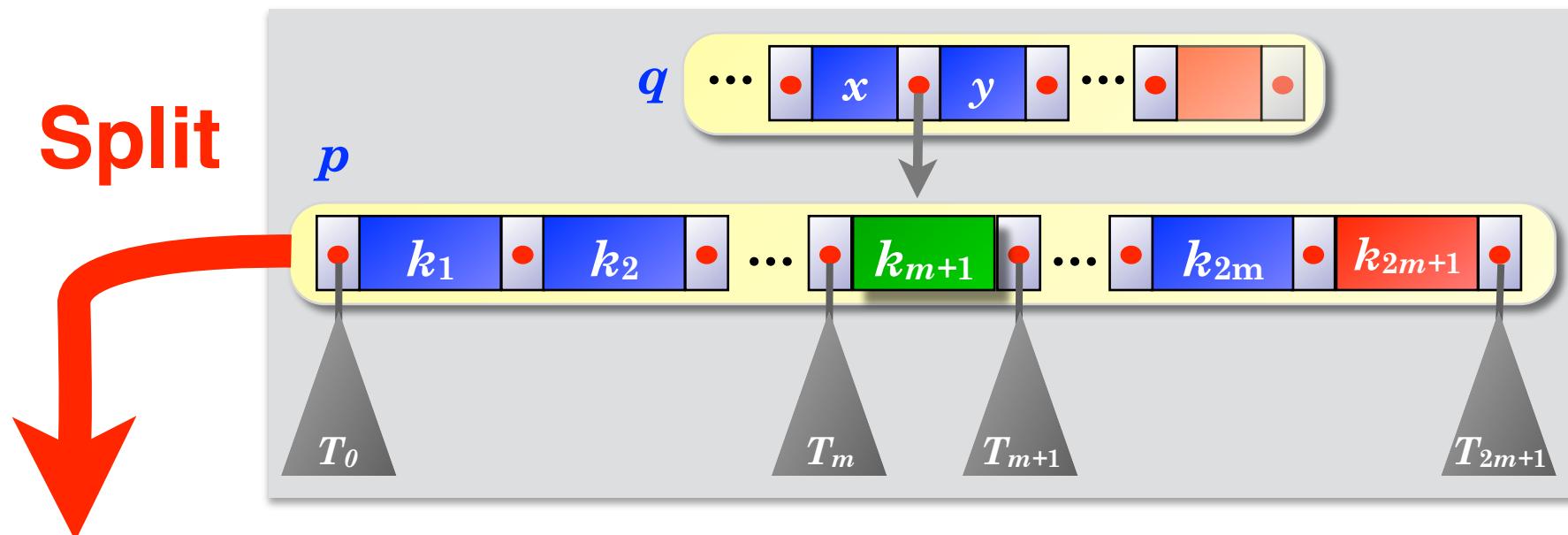
# B-Baum: Teilung bei Überlauf - 1. Fall 1

- **1. Fall:** Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



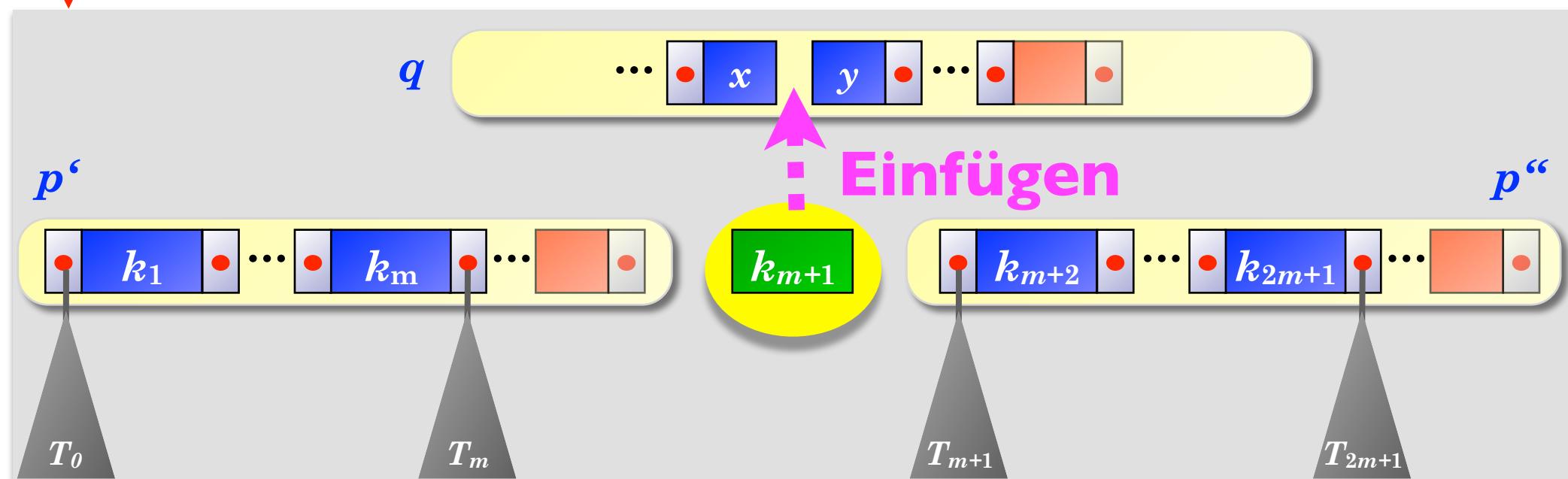
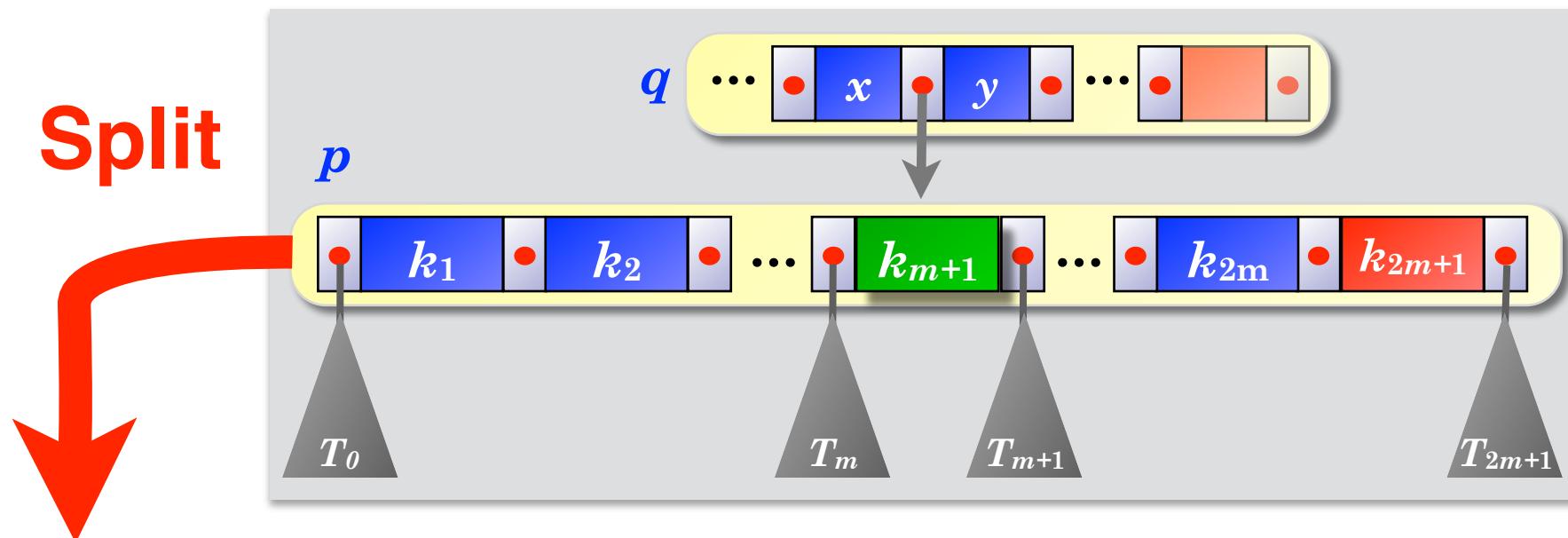
# B-Baum: Teilung bei Überlauf - 1. Fall 2

- 1. Fall: Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



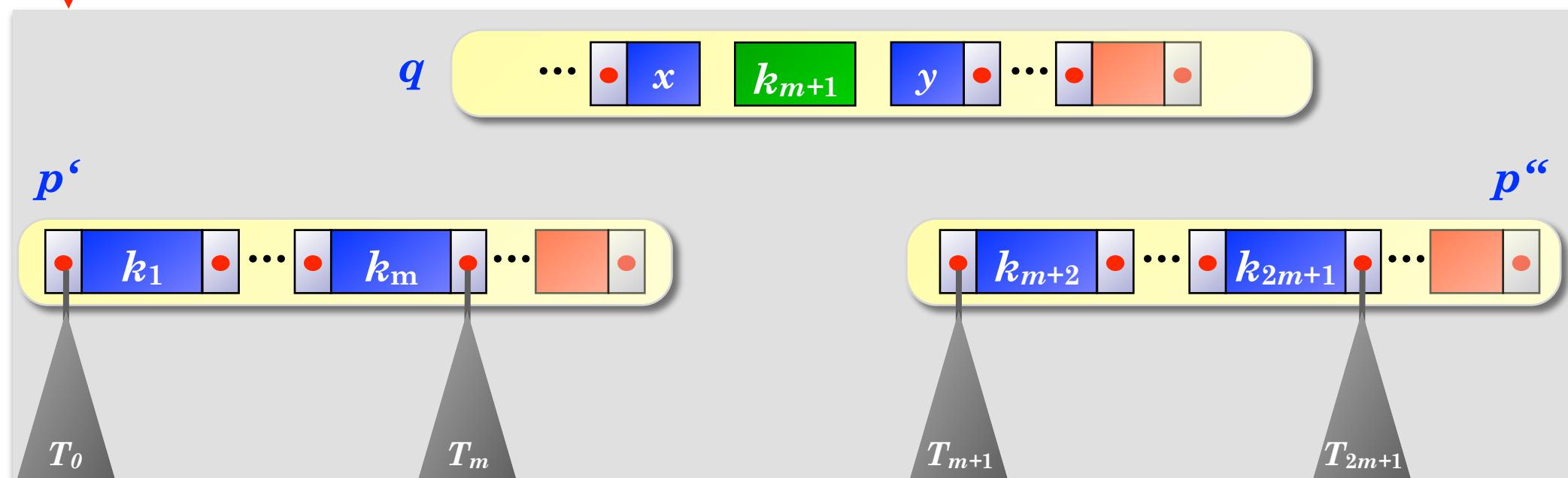
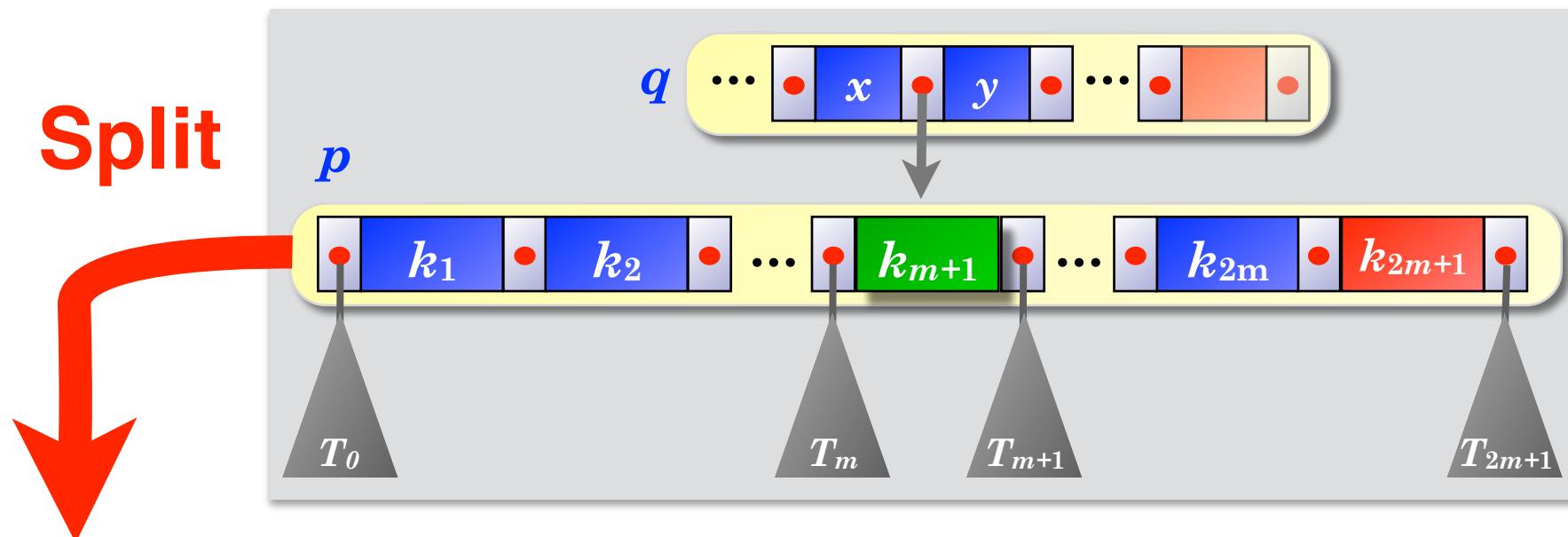
# B-Baum: Teilung bei Überlauf - 1. Fall 2

- 1. Fall: Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



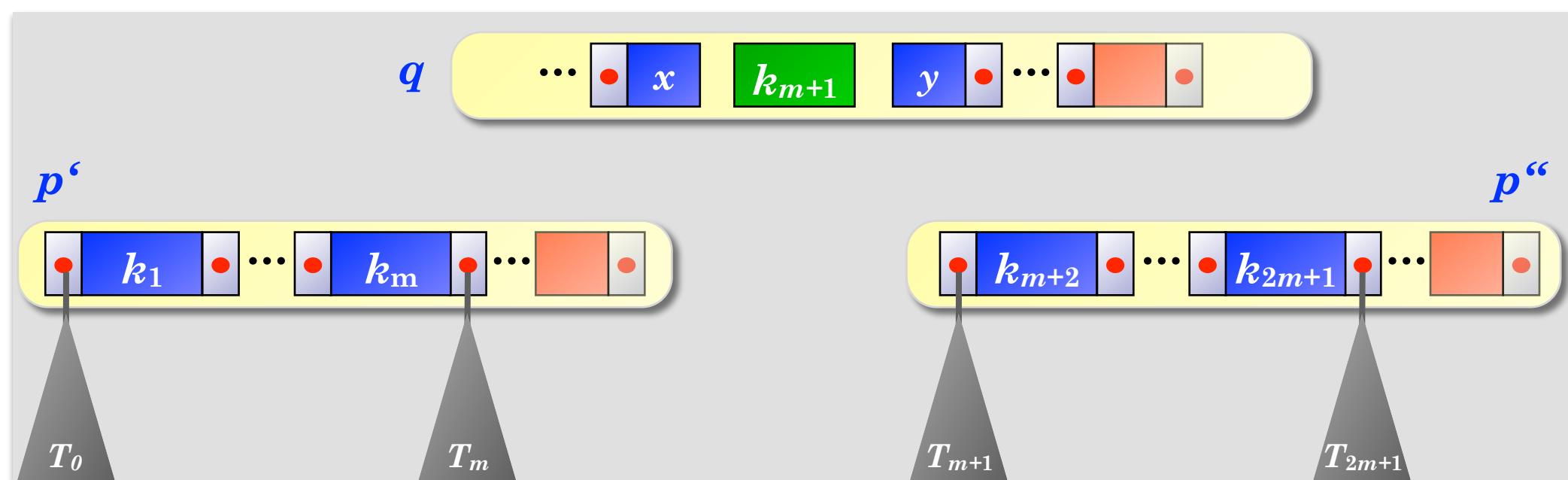
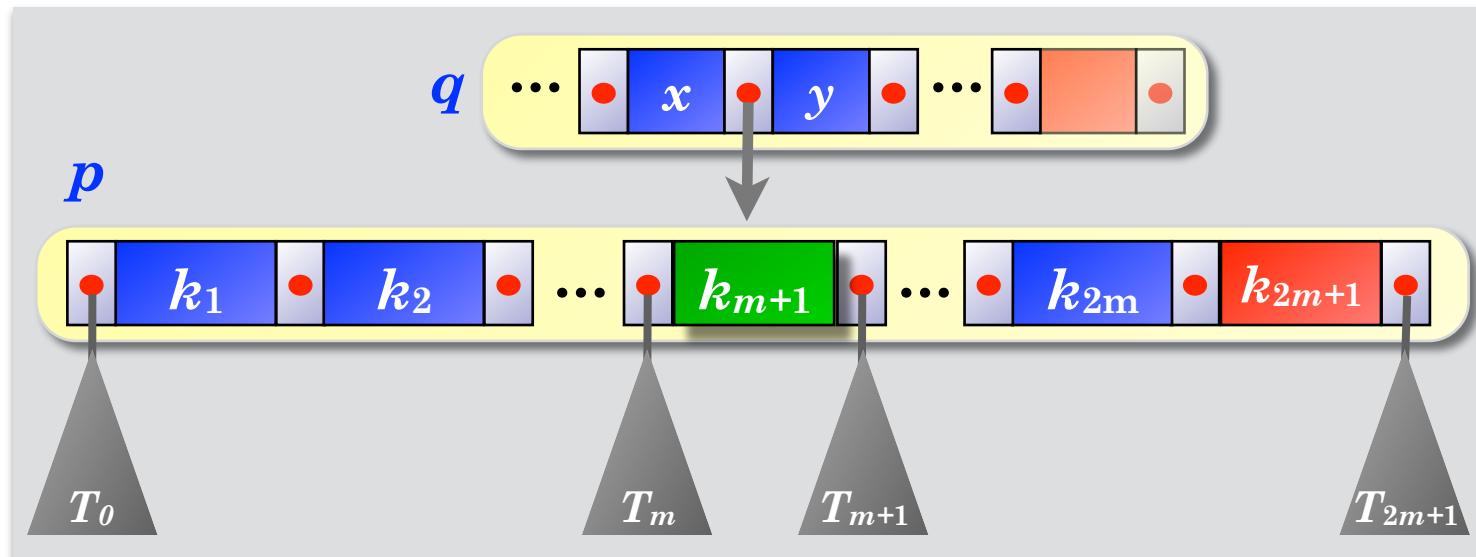
# B-Baum: Teilung bei Überlauf - 1. Fall 2

- 1. Fall: Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



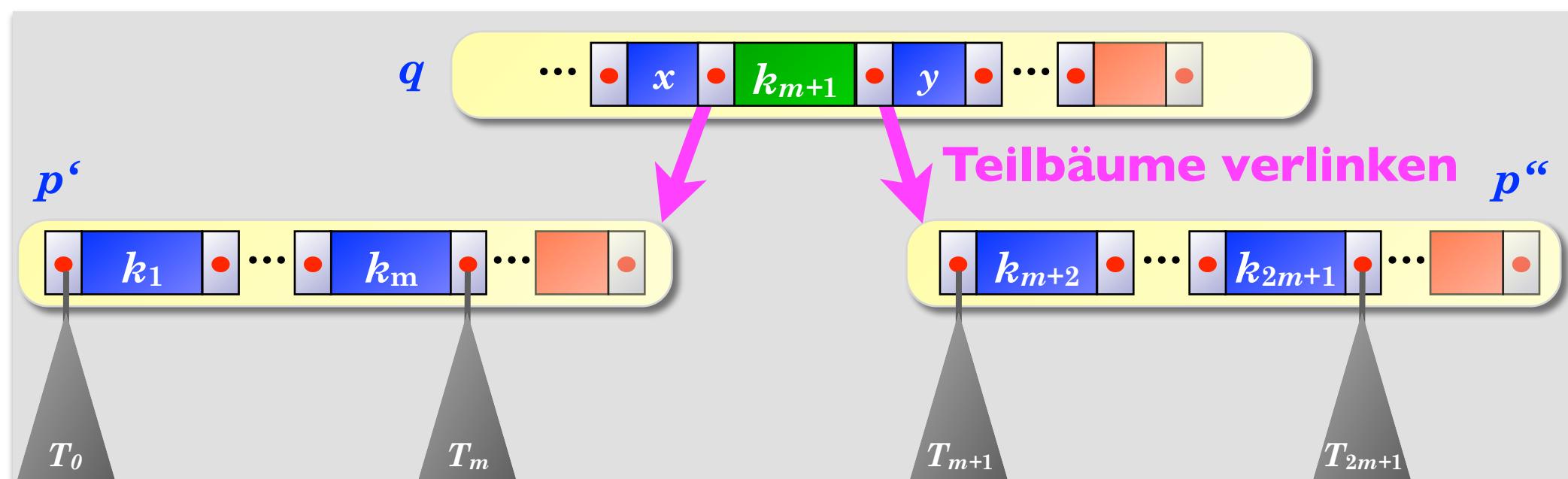
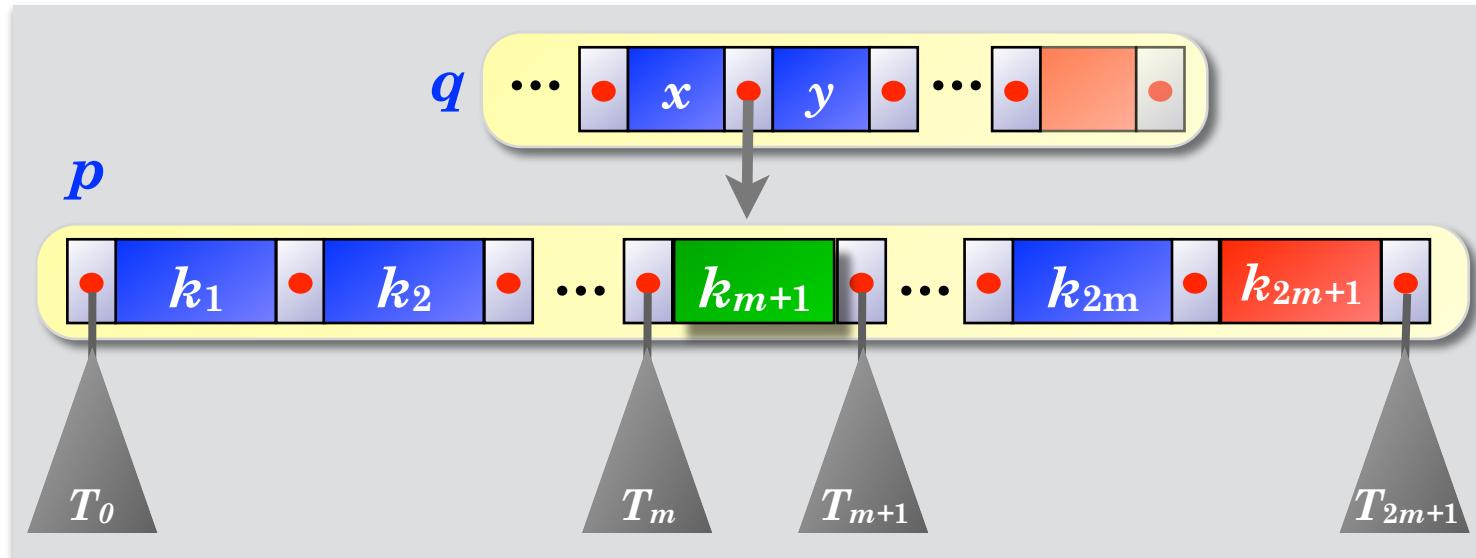
# B-Baum: Teilung bei Überlauf - 1. Fall 3

- 1. Fall: Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



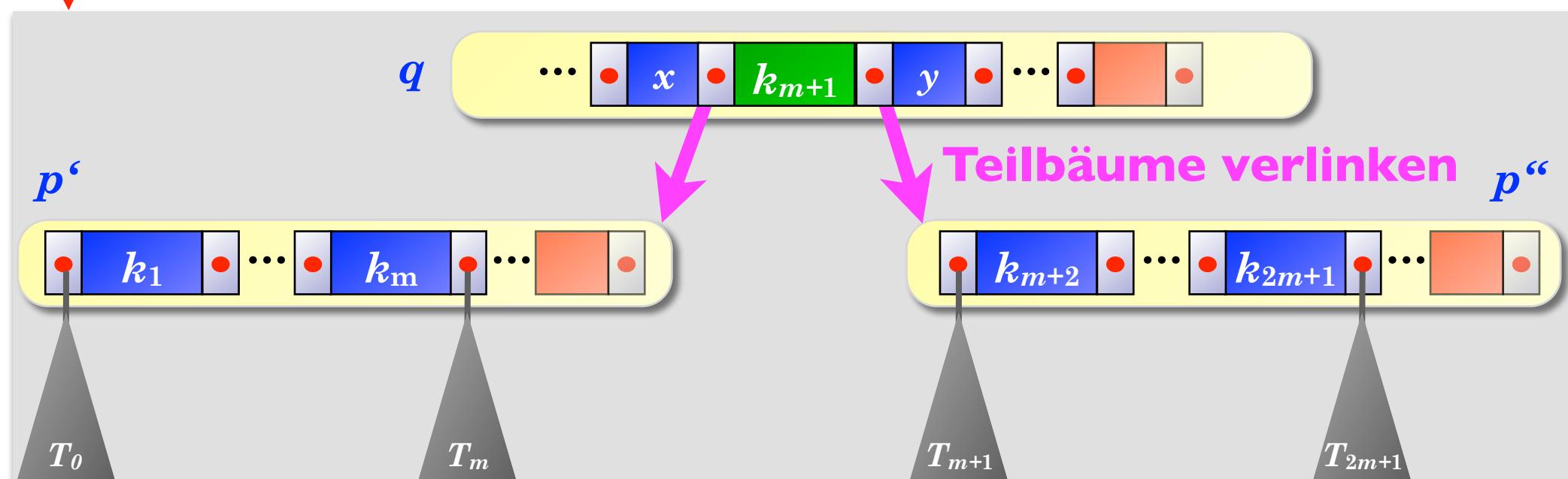
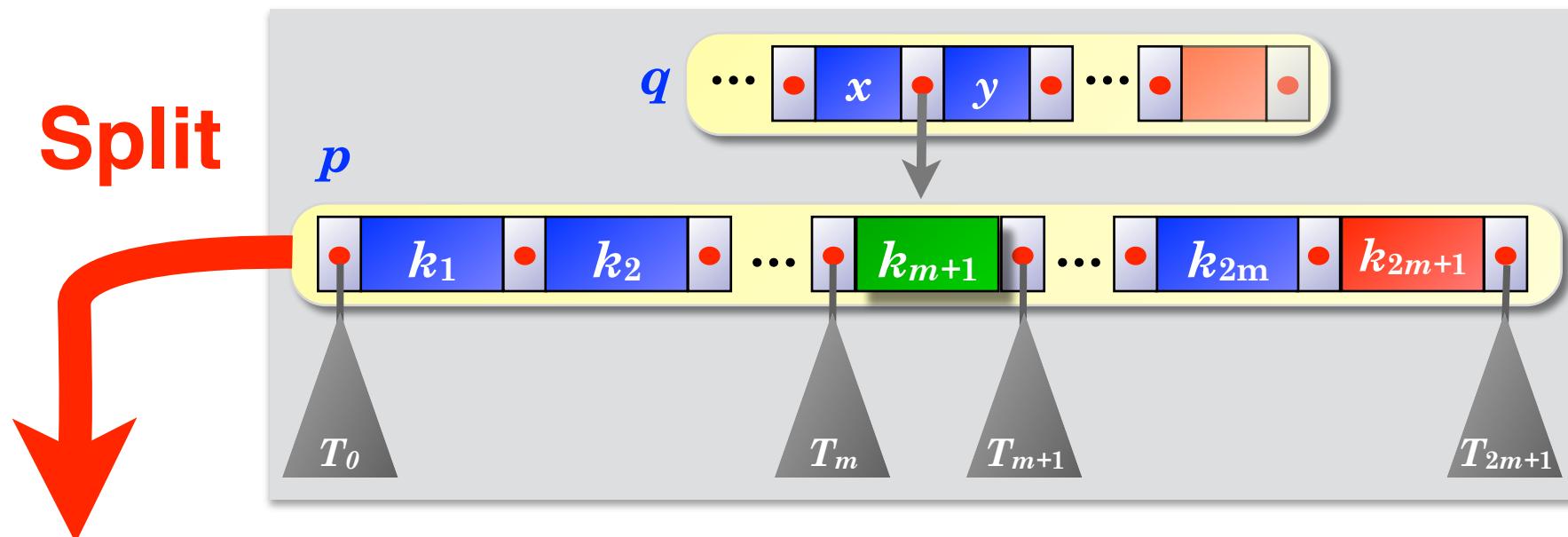
# B-Baum: Teilung bei Überlauf - 1. Fall 3

- 1. Fall: Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$

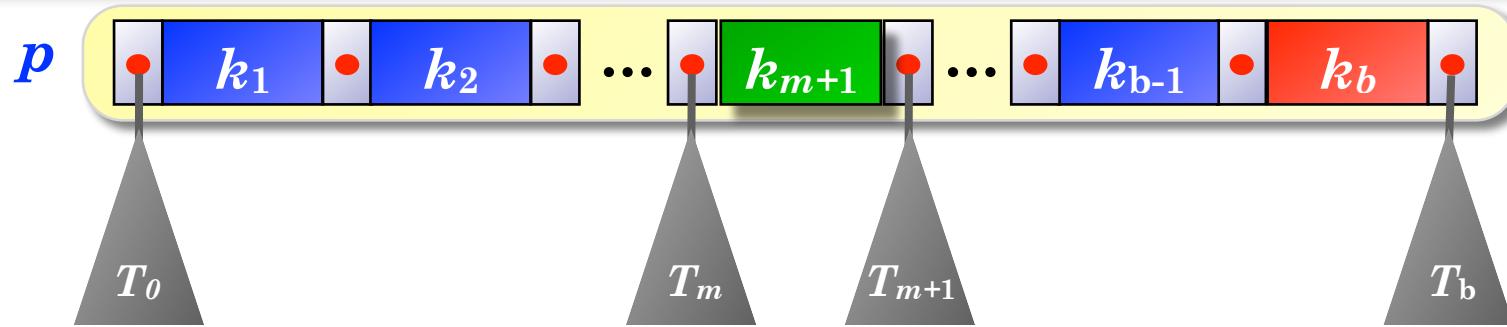


# B-Baum: Teilung bei Überlauf - 1. Fall 3

- 1. Fall: Der übergelaufene Knoten  $p$  hat einen Vaterknoten  $q$



# Rückblick auf Definition ( $a,b$ )-Baum ...

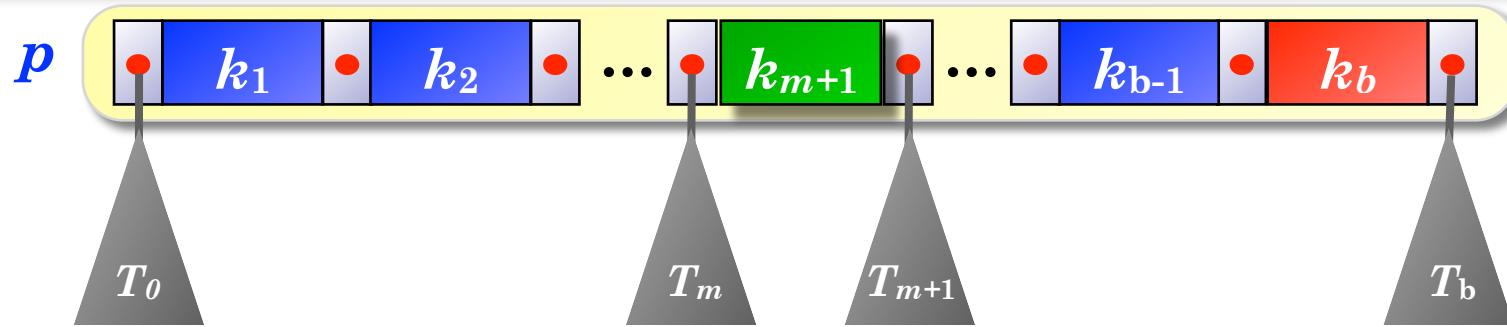


## D ( $a,b$ )-Baum

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2$  und  $b \geq 2a - 1$ .  $\rho(v)$  sei die Anzahl der Söhne des Knotens  $v$ . Ein Mehrwegsuchbaum heißt **( $a,b$ )-Baum**, wenn

...

# Rückblick auf Definition ( $a,b$ )-Baum ...



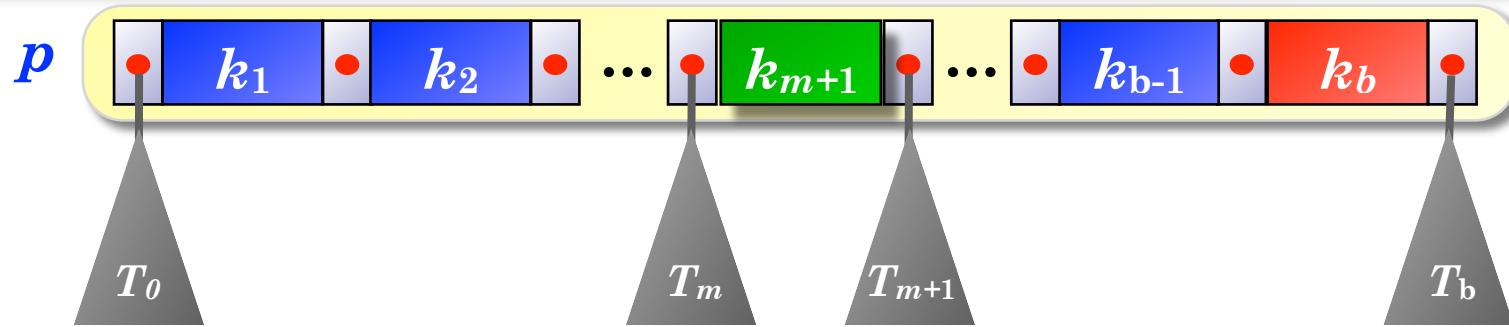
Klar, mindestens ein Schlüssel!

## D **$(a,b)$ -Baum**

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2$  und  $b \geq 2a - 1$ .  $\rho(v)$  sei die Anzahl der Söhne des Knotens  $v$ . Ein Mehrwegsuchbaum heißt  **$(a,b)$ -Baum**, wenn

...

# Rückblick auf Definition $(a,b)$ -Baum ...



**Beim Teilen:**  $b-1$  Schlüssel ( $k_{m+1}$  wandert in Vaterknoten) werden auf Knoten verteilt. Je Knoten ergeben sich also minimal  $\lfloor (b-1)/2 \rfloor + 1$  Nachfolger; es folgt:

$$a \leq \left\lfloor \frac{b-1}{2} \right\rfloor + 1$$

$$a-1 \leq \frac{b-1}{2}$$

$$2a-2 \leq b-1$$

$$2a-1 \leq b$$

Klar, mindestens ein Schlüssel!

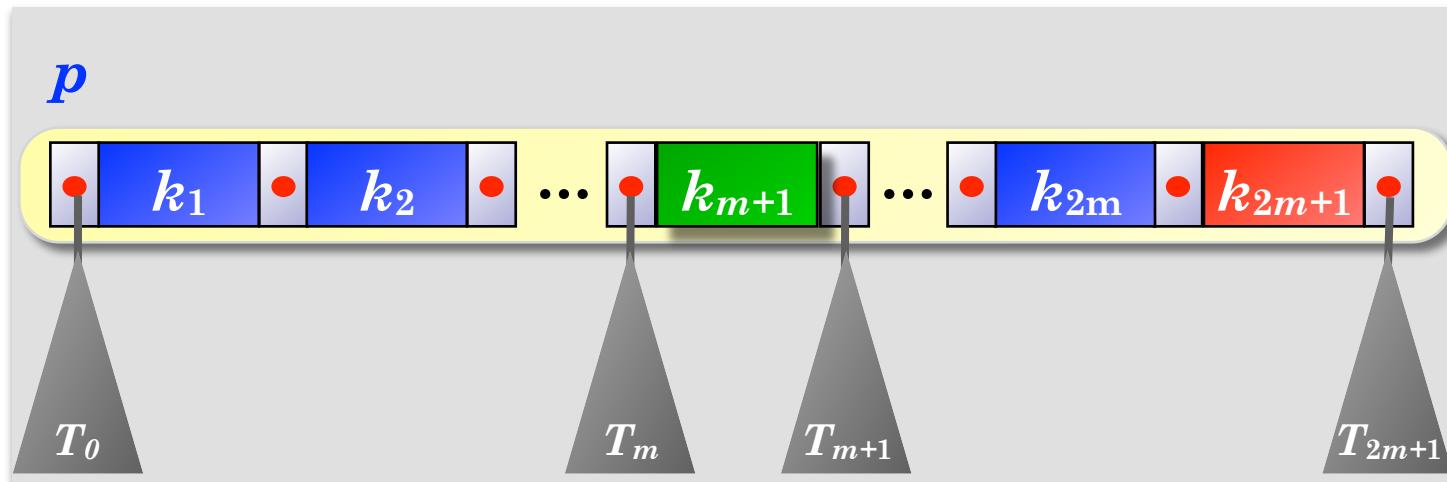
## D $(a,b)$ -Baum

Seien  $a, b \in \mathbb{N}$  mit  $a \geq 2$  und  $b \geq 2a - 1$ .  $\rho(v)$  sei die Anzahl der Söhne des Knotens  $v$ . Ein Mehrwegsuchbaum heißt  $(a,b)$ -Baum, wenn

...

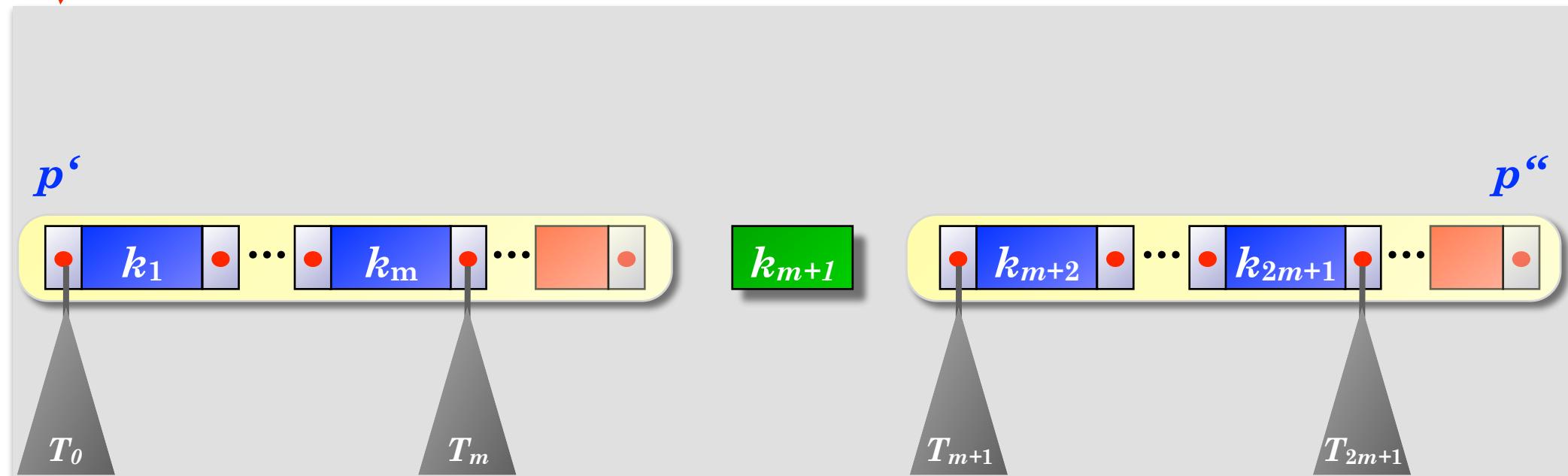
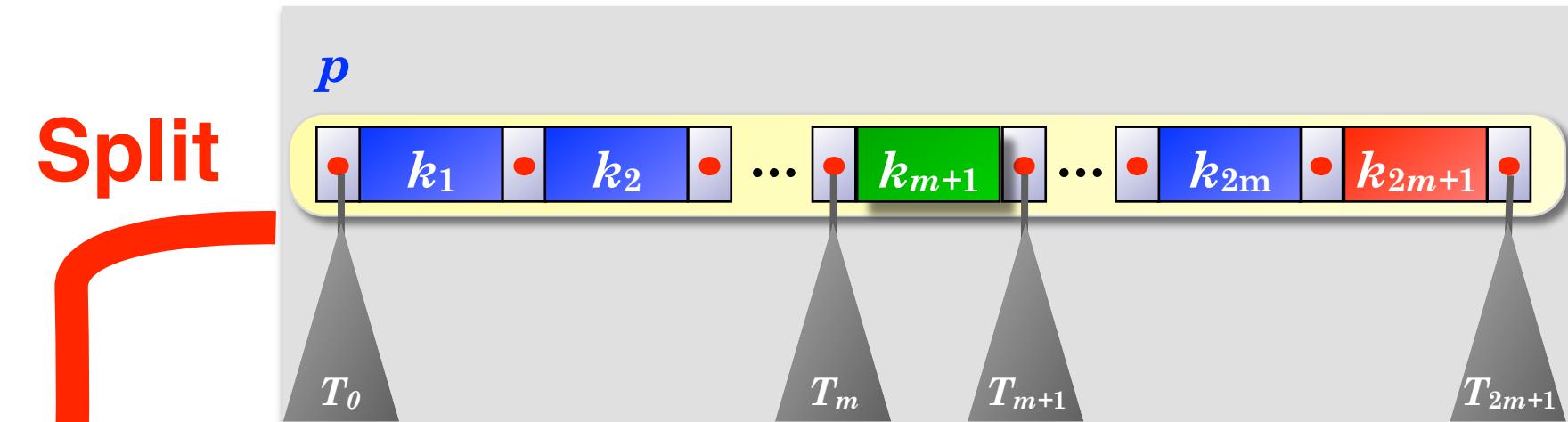
# B-Baum: Teilung bei Überlauf - 2. Fall 1

- **2. Fall:** Der übergelaufene Knoten  $p$  ist die Wurzel



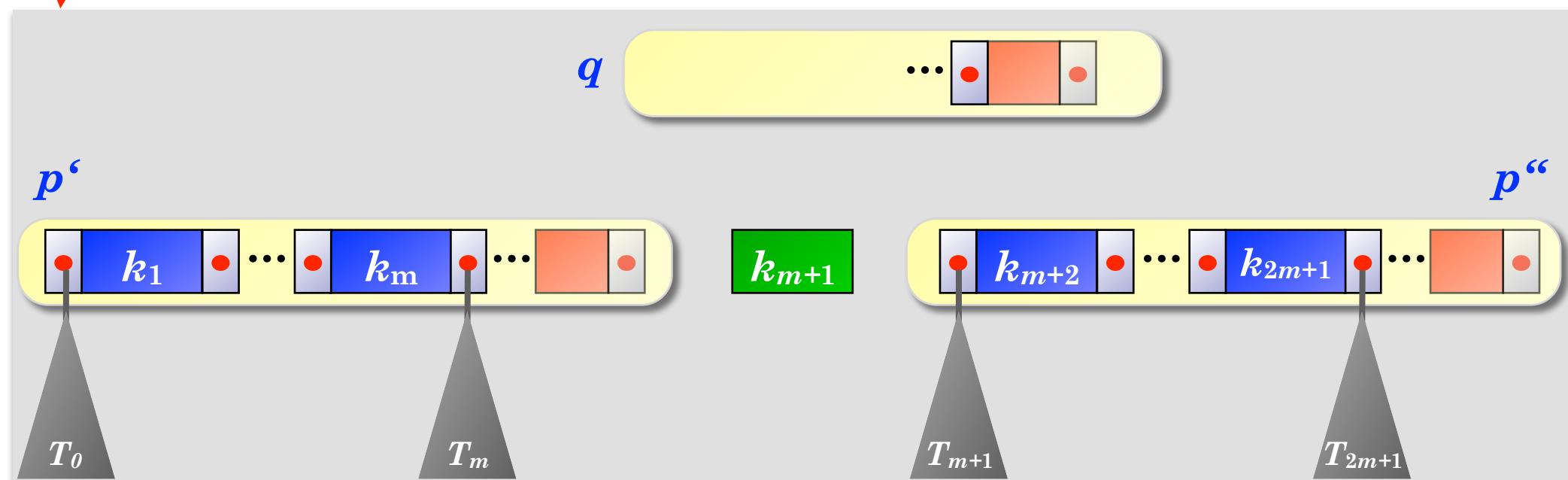
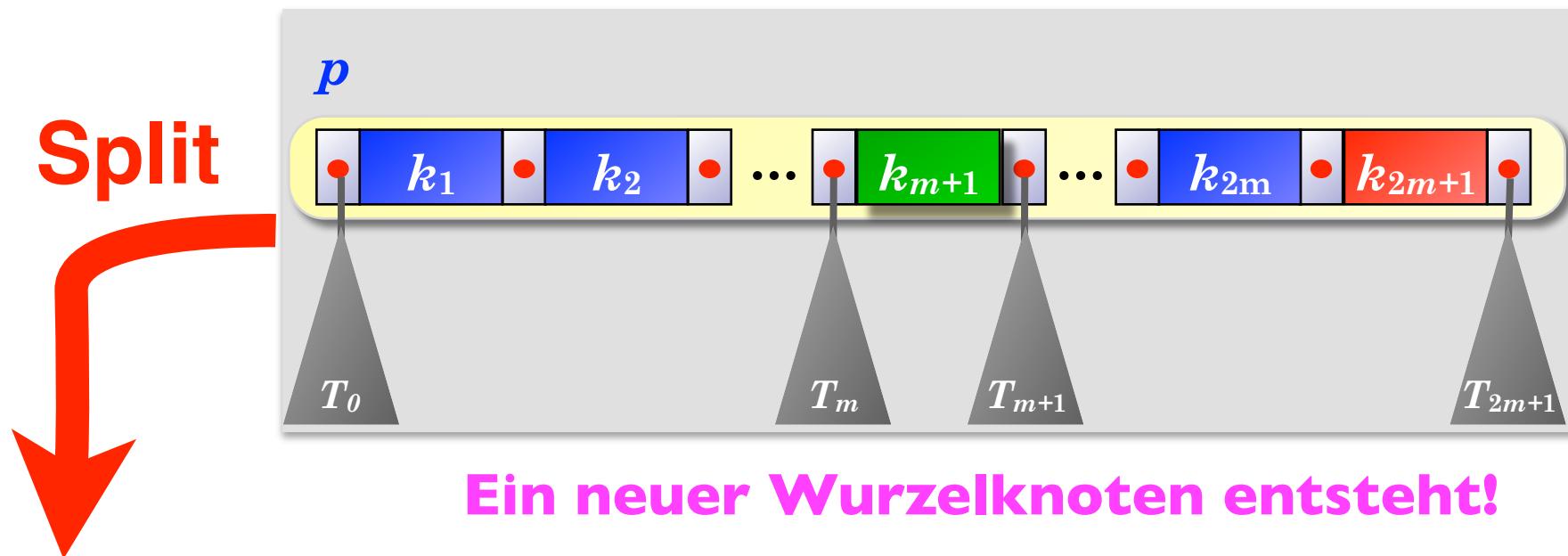
# B-Baum: Teilung bei Überlauf - 2. Fall 1

- 2. Fall: Der übergelaufene Knoten  $p$  ist die Wurzel



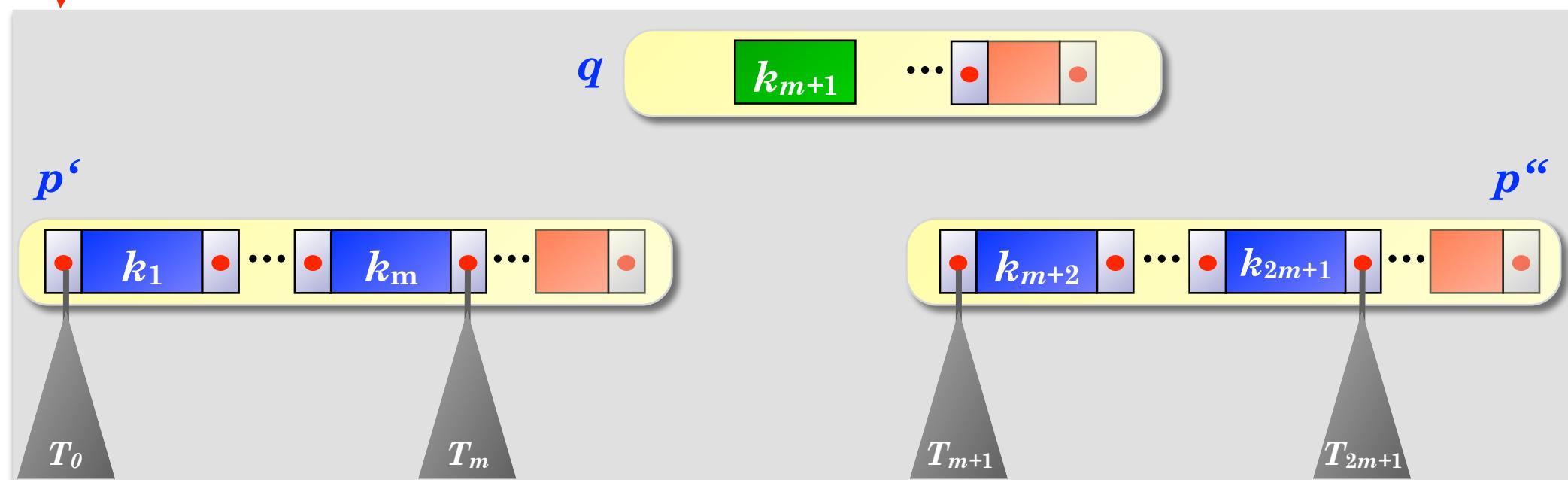
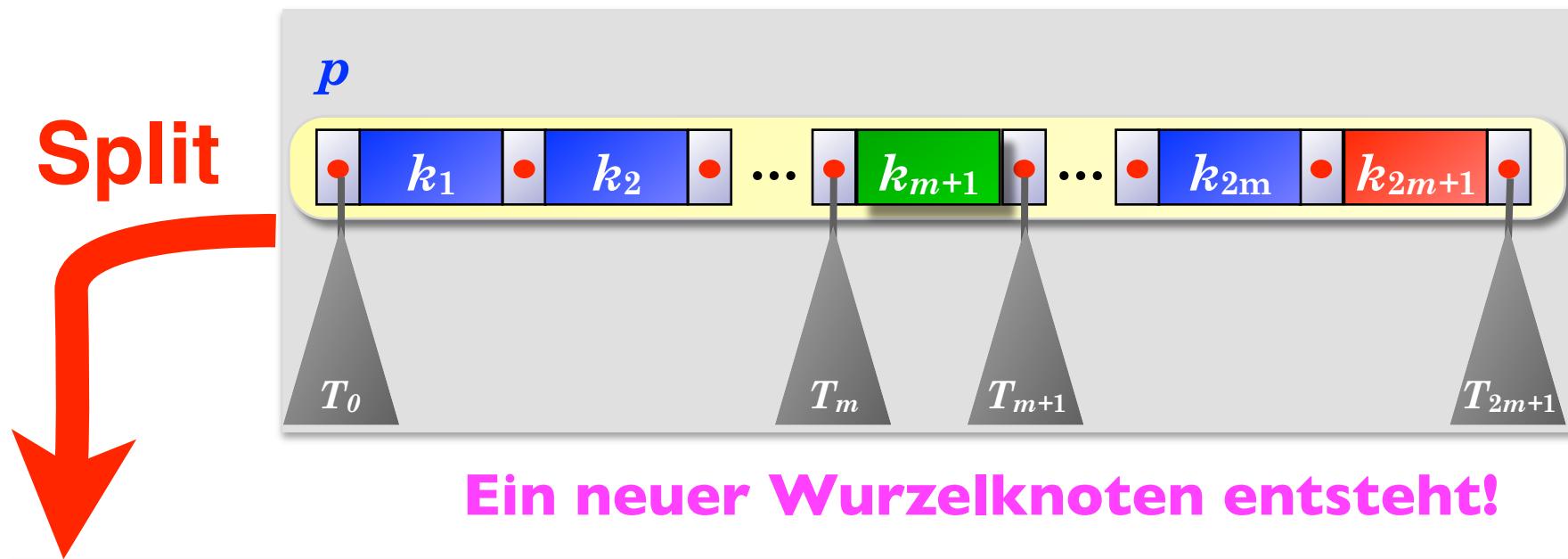
# B-Baum: Teilung bei Überlauf - 2. Fall 2

- 2. Fall: Der übergelaufene Knoten  $p$  ist die Wurzel



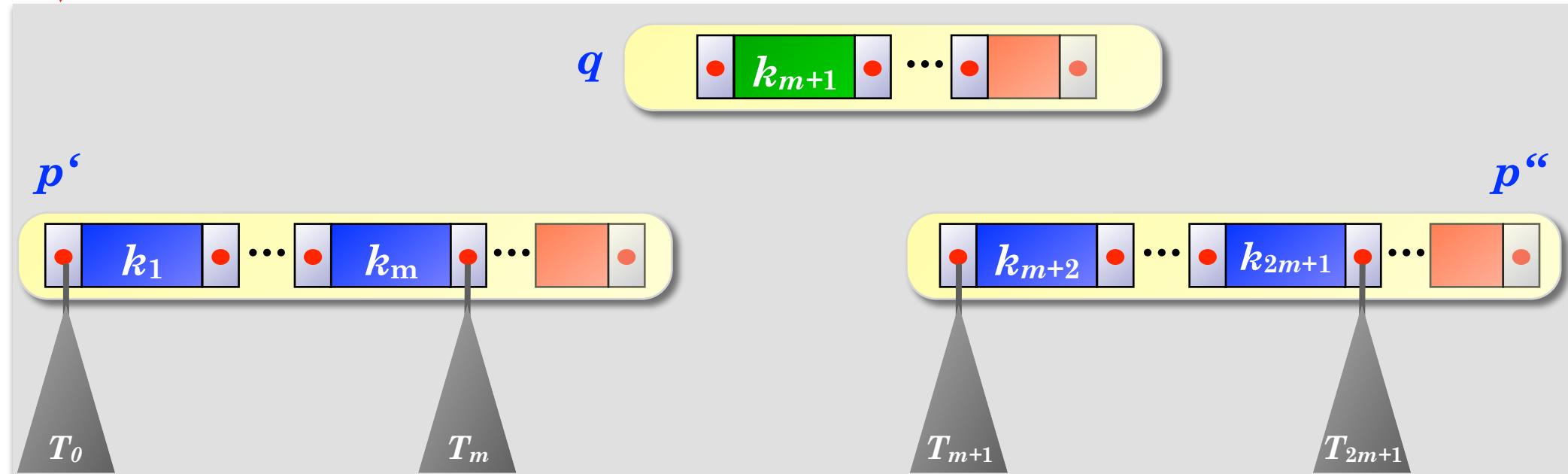
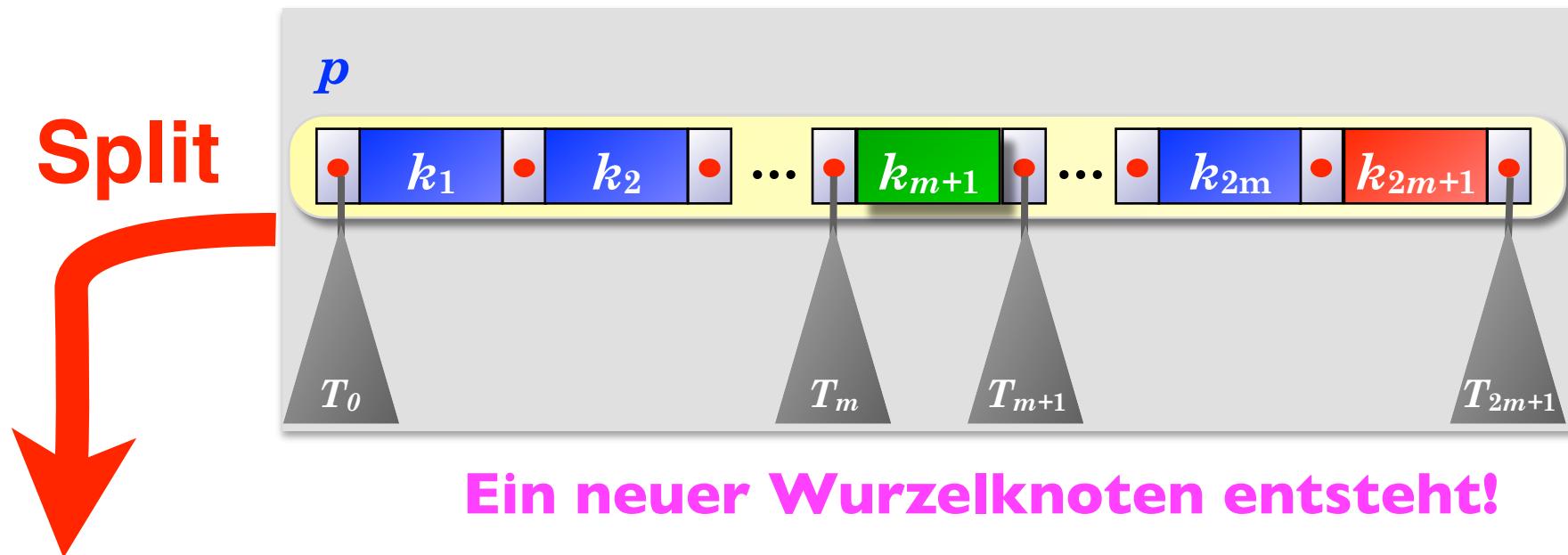
# B-Baum: Teilung bei Überlauf - 2. Fall 2

- 2. Fall: Der übergelaufene Knoten  $p$  ist die Wurzel



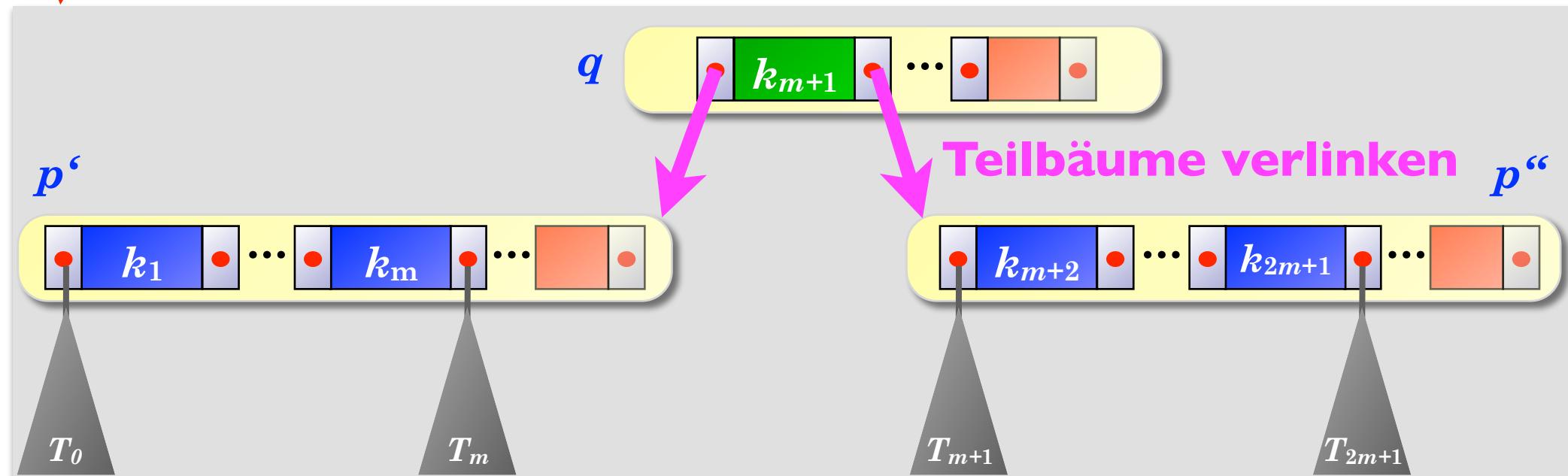
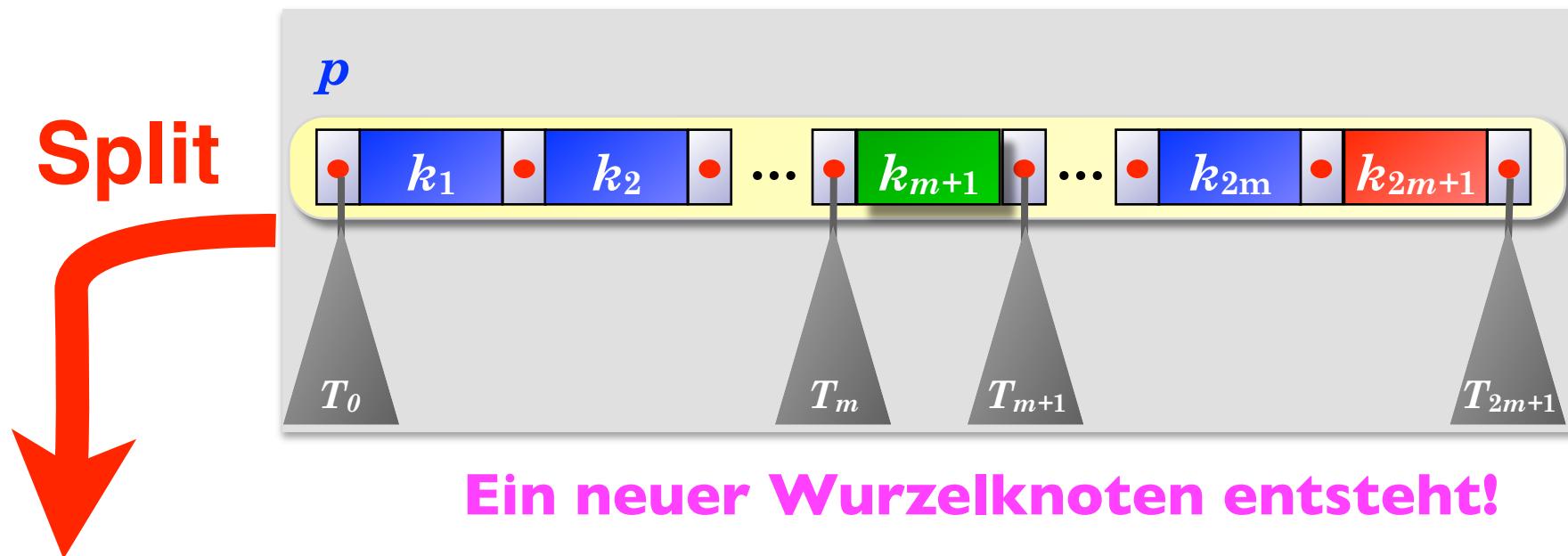
# B-Baum: Teilung bei Überlauf - 2. Fall 3

- 2. Fall: Der übergelaufene Knoten  $p$  ist die Wurzel



# B-Baum: Teilung bei Überlauf - 2. Fall 3

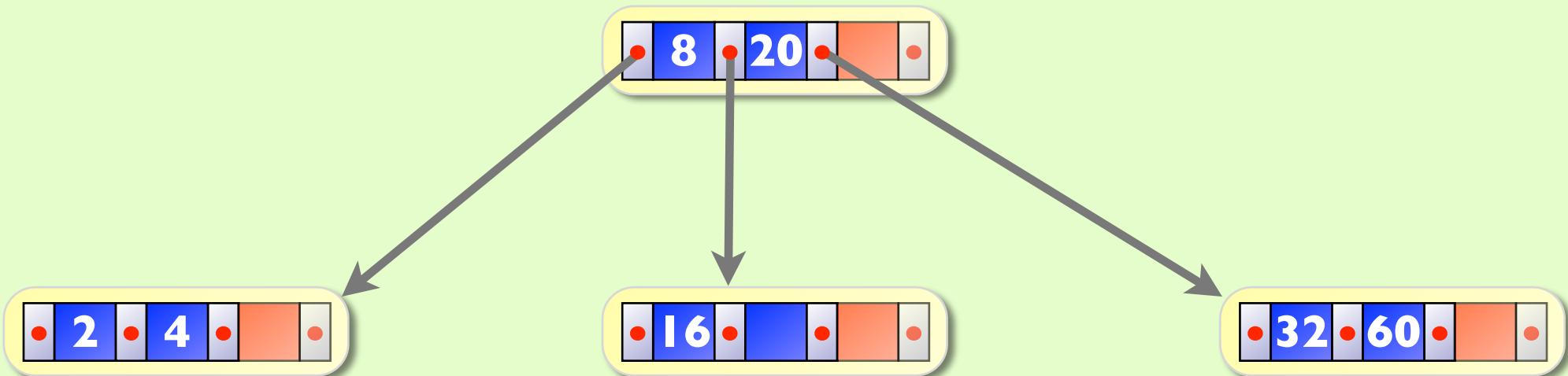
- 2. Fall: Der übergelaufene Knoten  $p$  ist die Wurzel



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

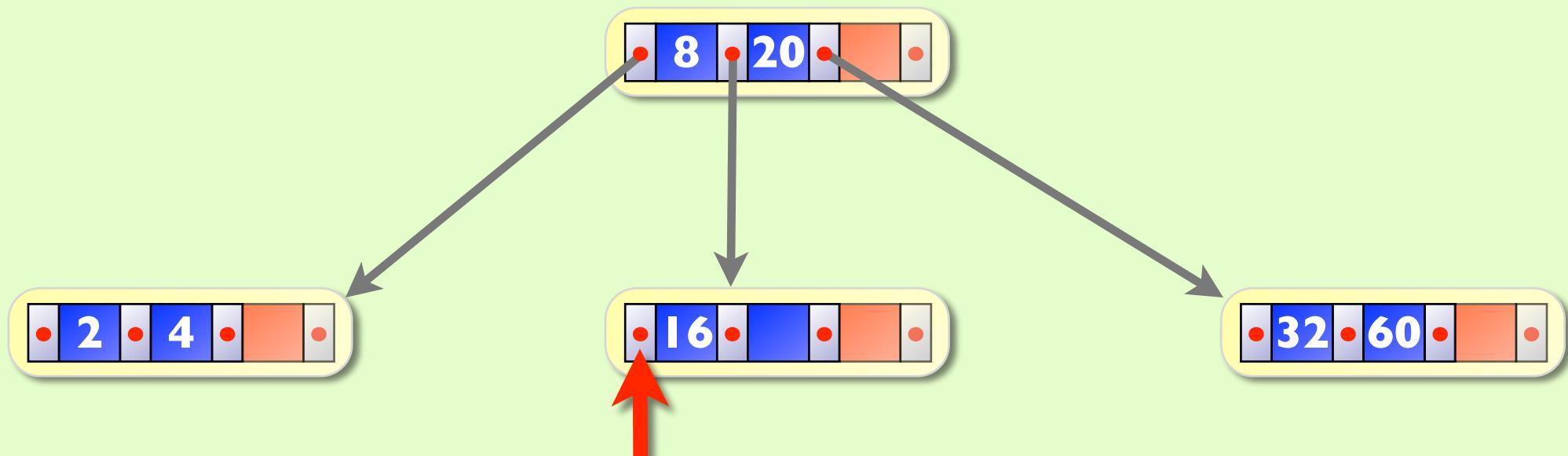
Einzufügen: 12



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

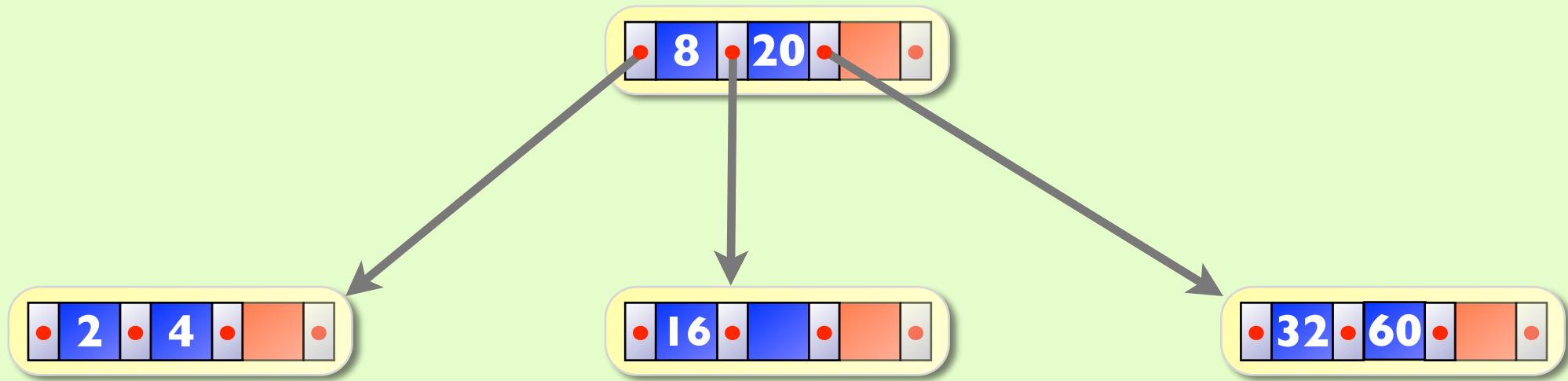
Einzufügen: 12



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

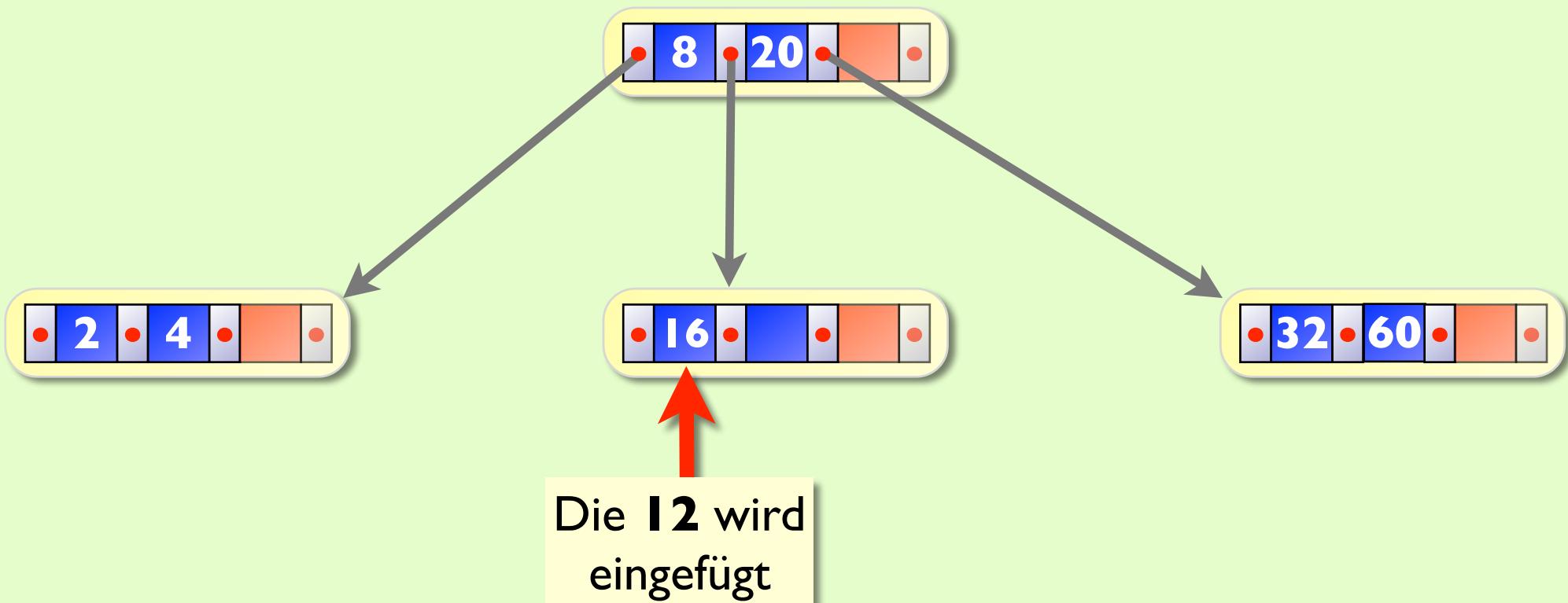
Einzufügen: 12



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

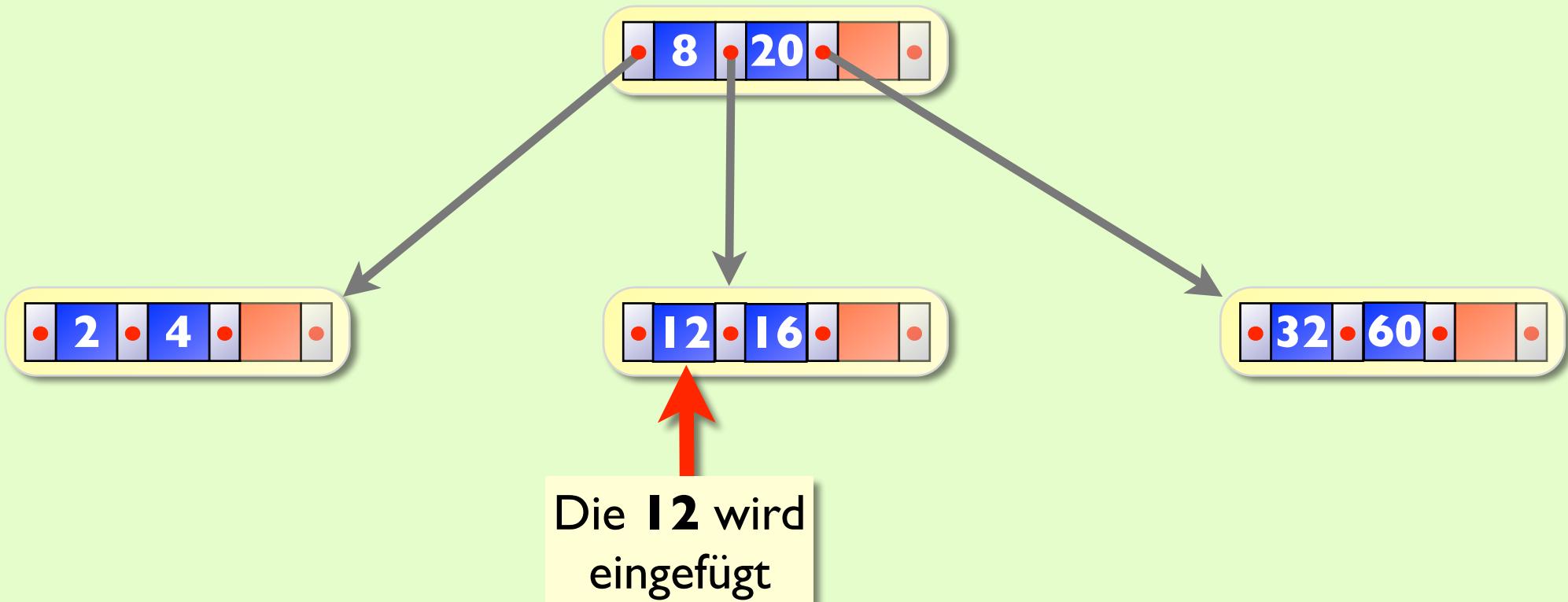
Einzufügen: 12



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 12



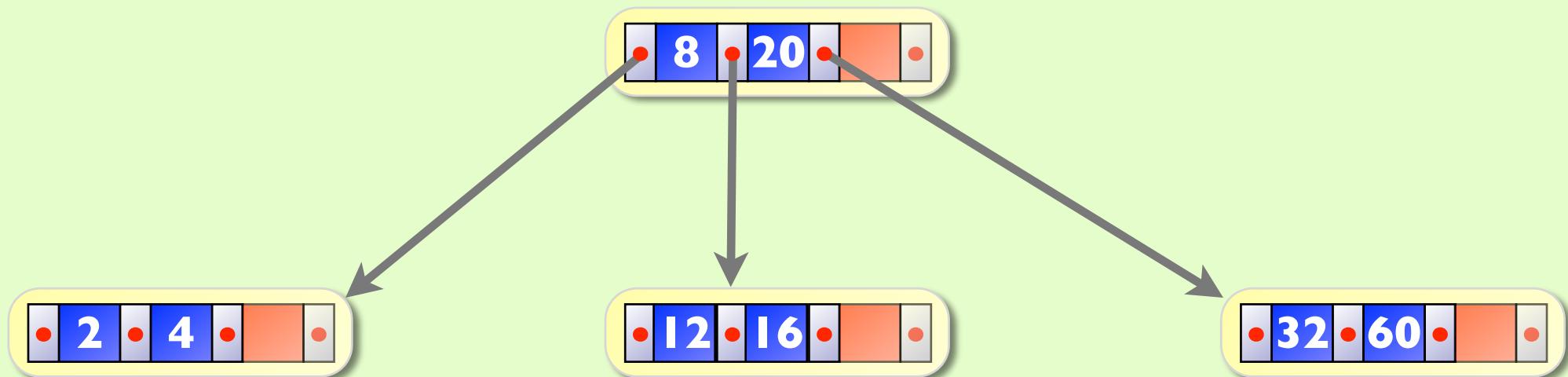
# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

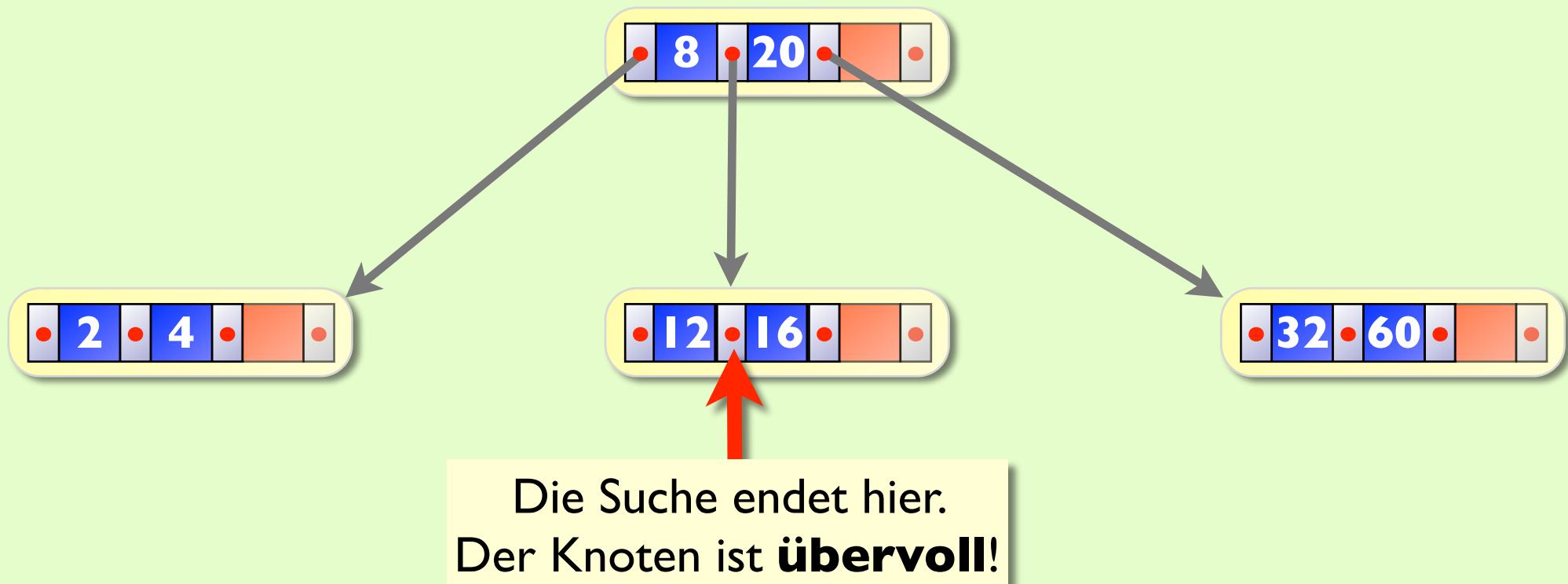
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

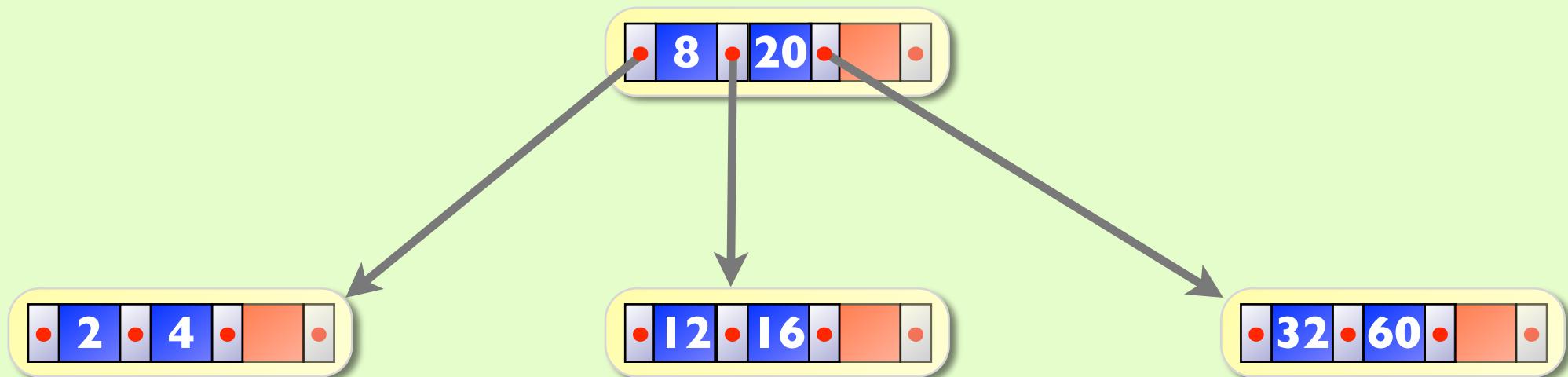
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

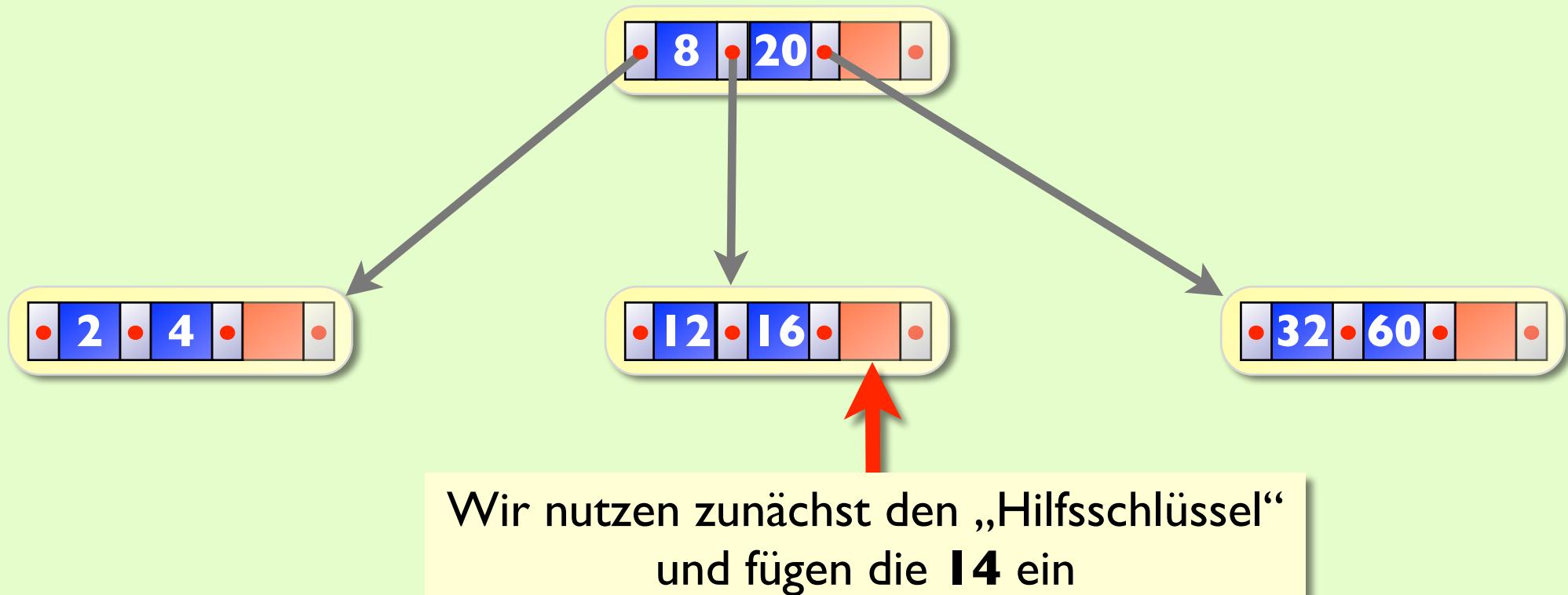
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

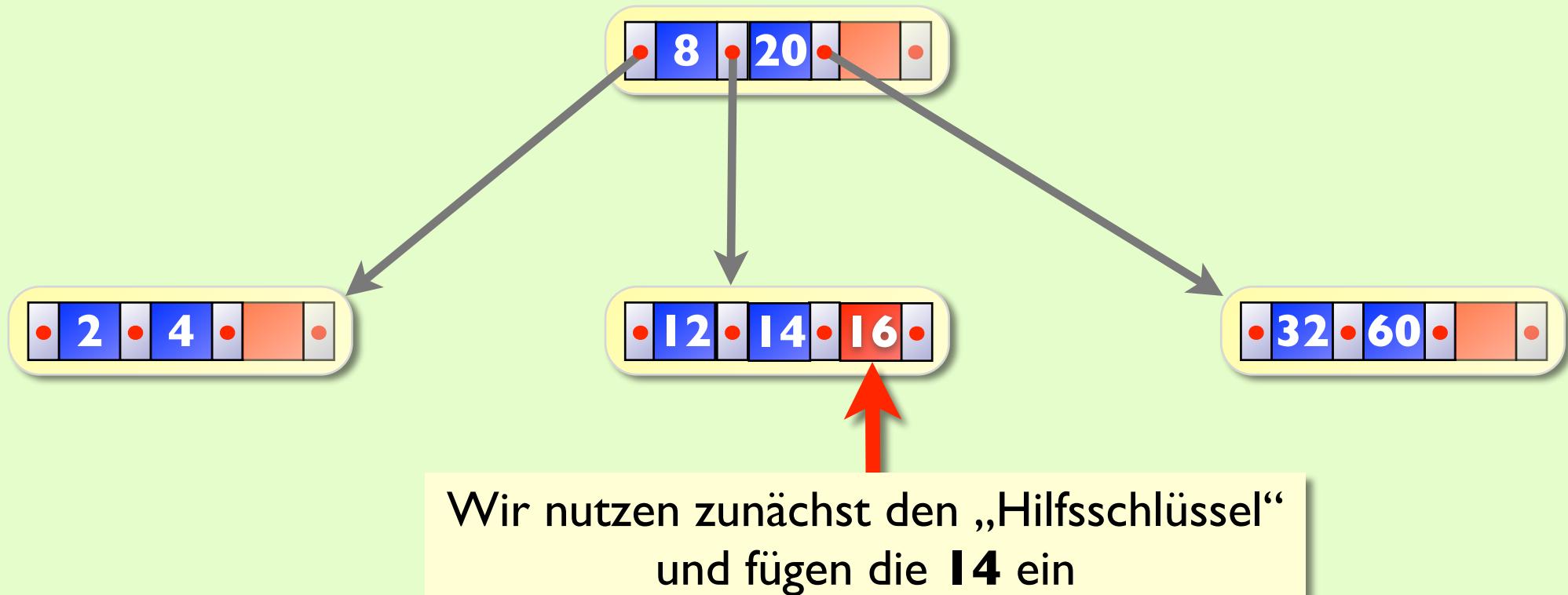
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

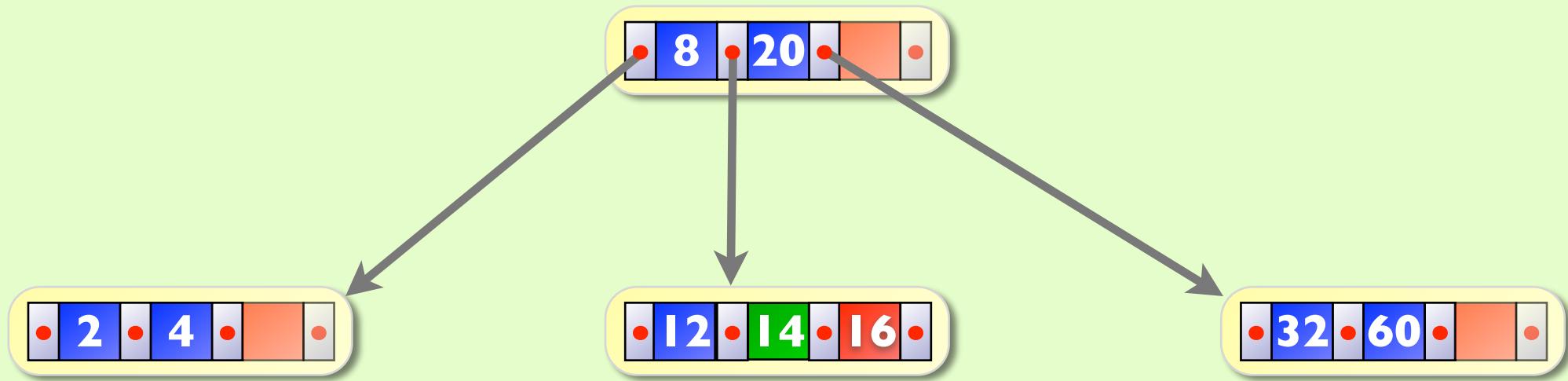
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

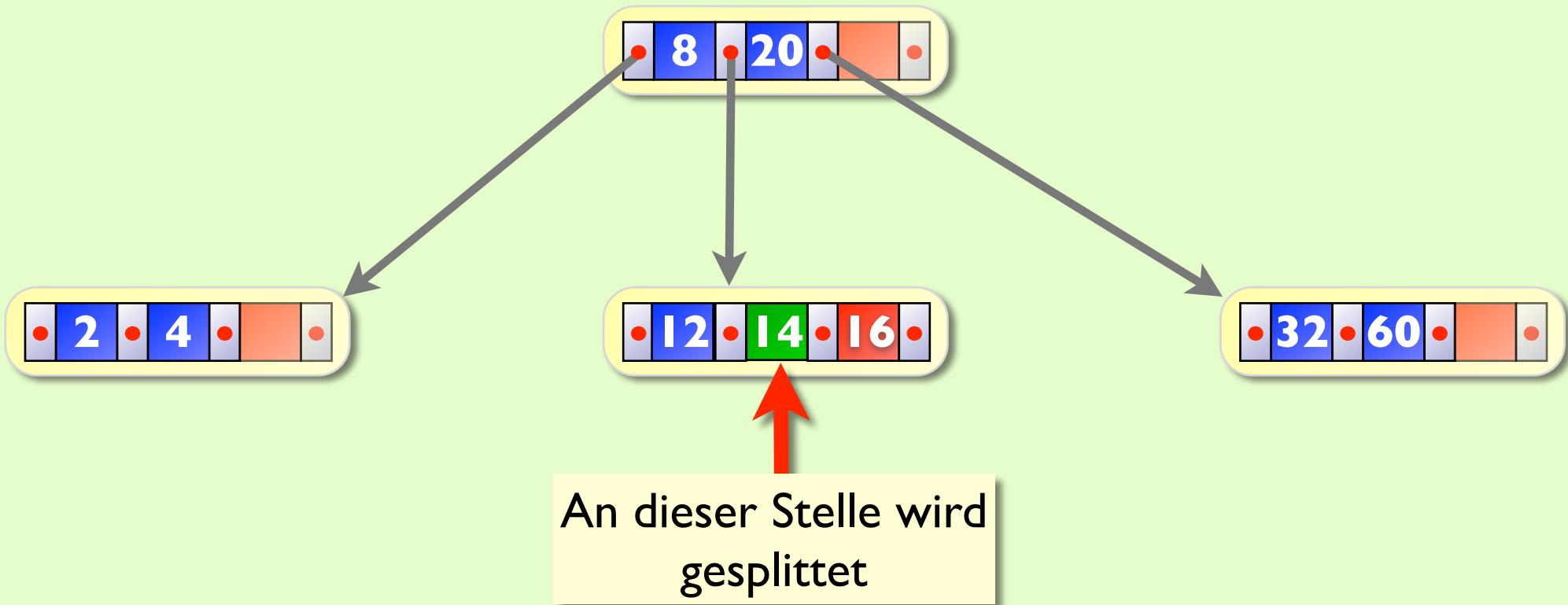
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

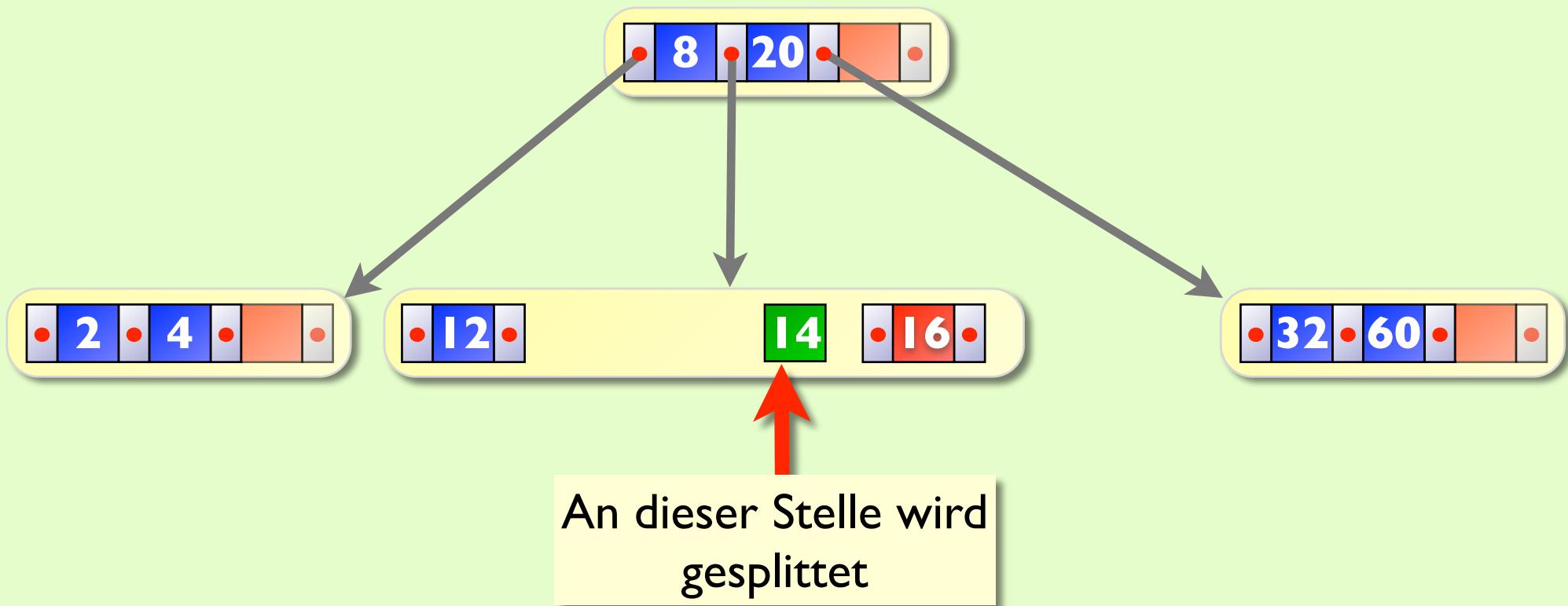
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

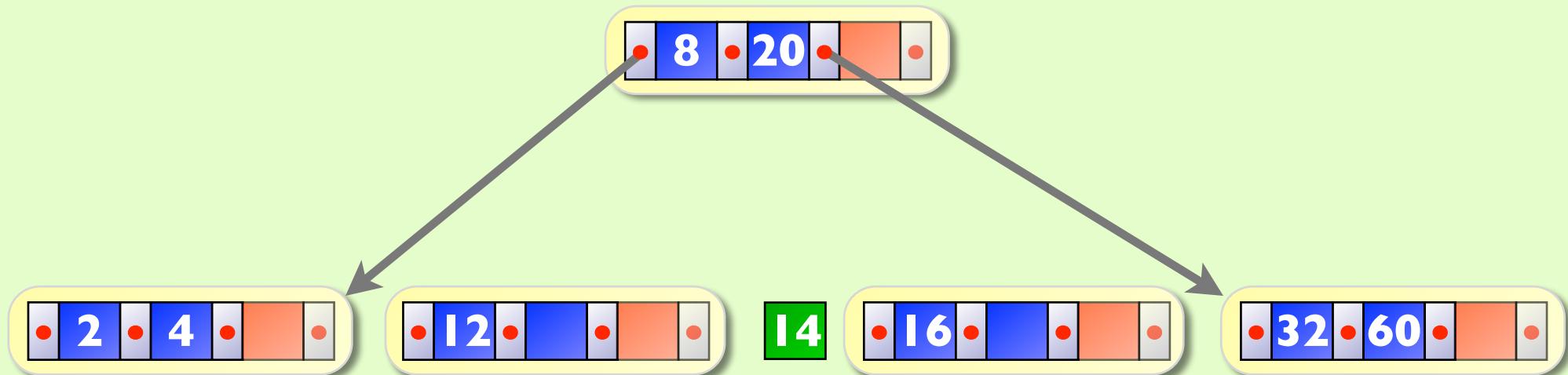
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

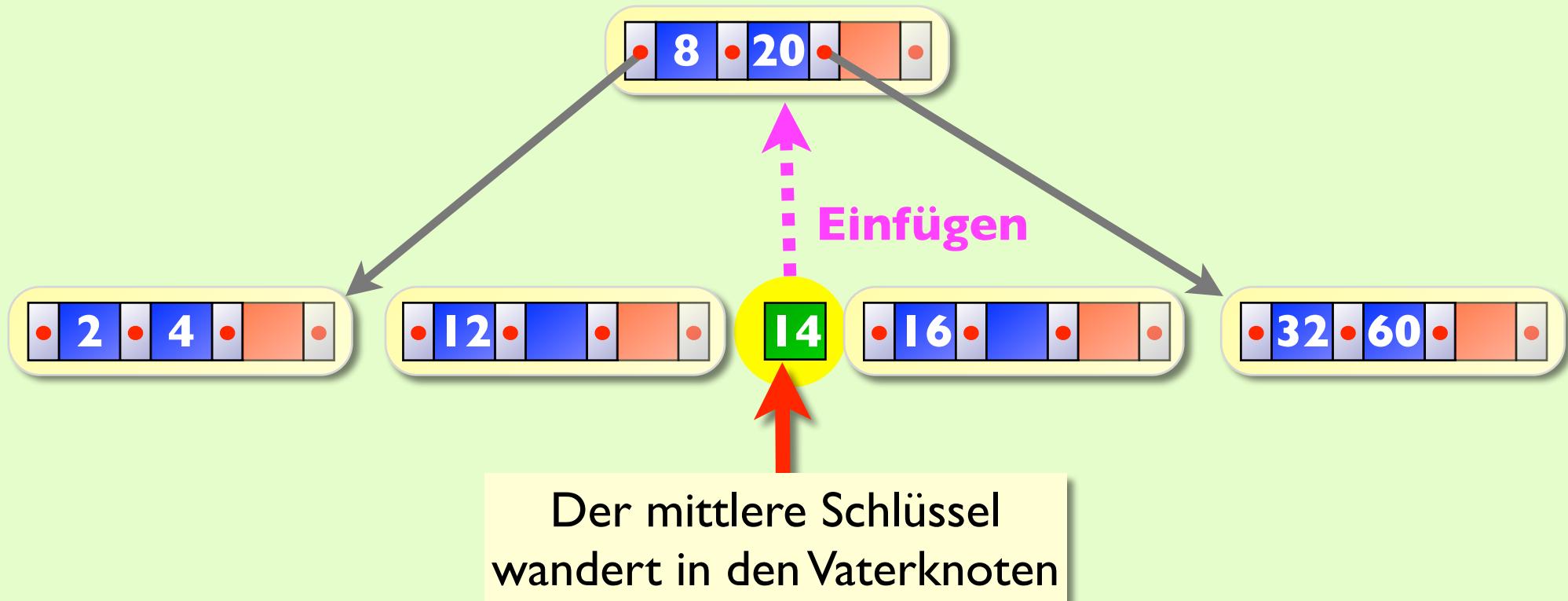
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

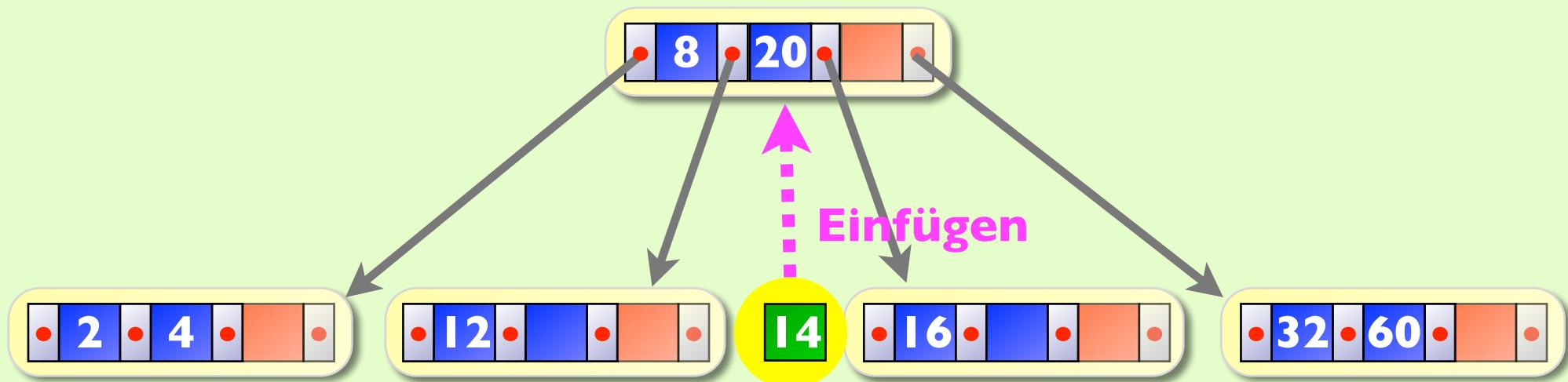
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 14

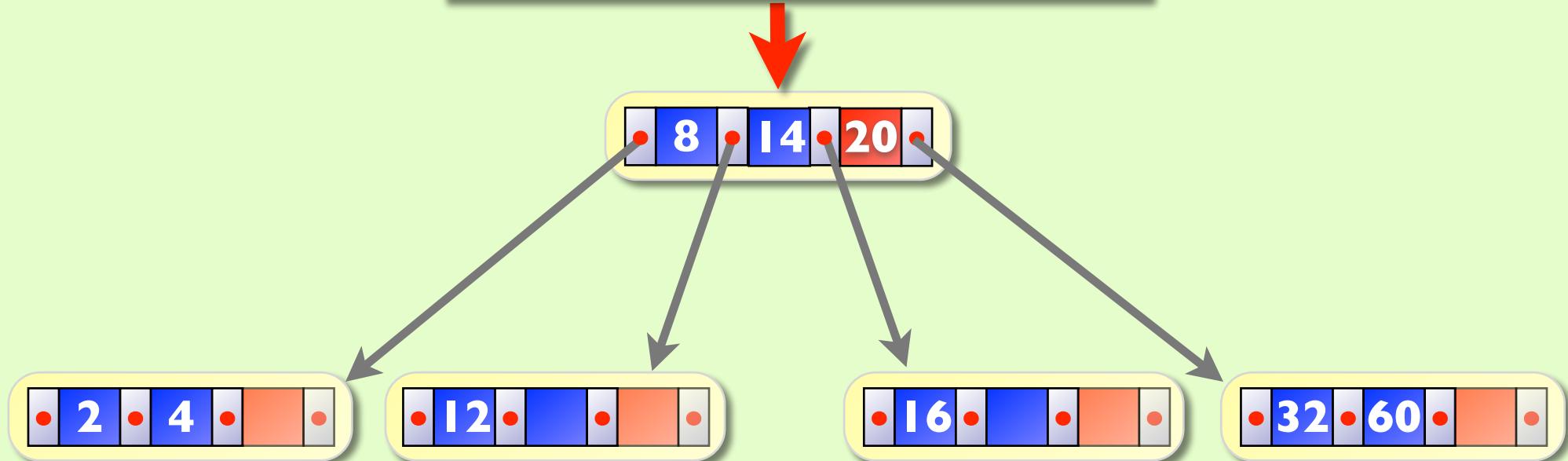


# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 14

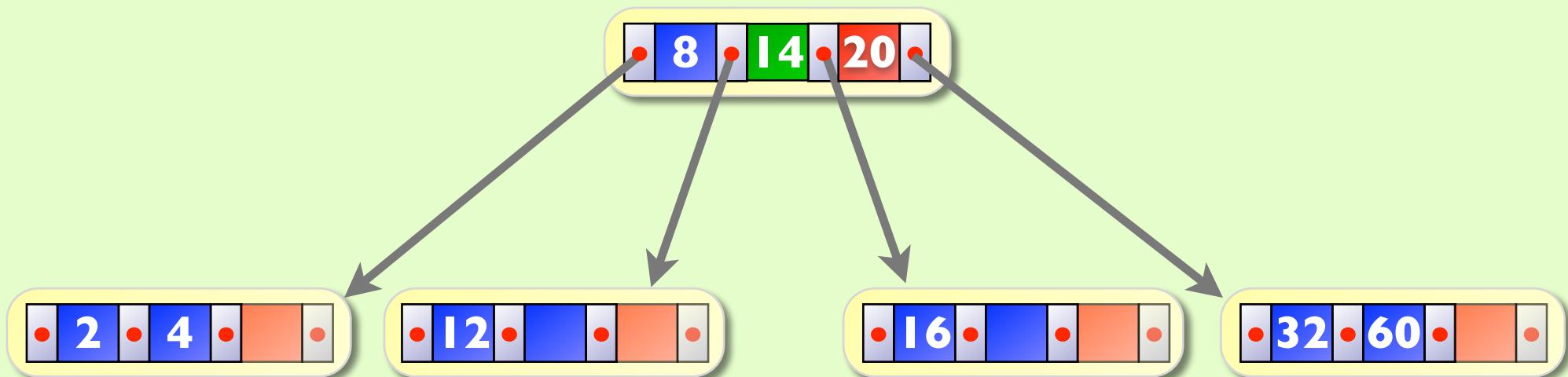
Der Wurzelknoten ist **übergeladen**!



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 14

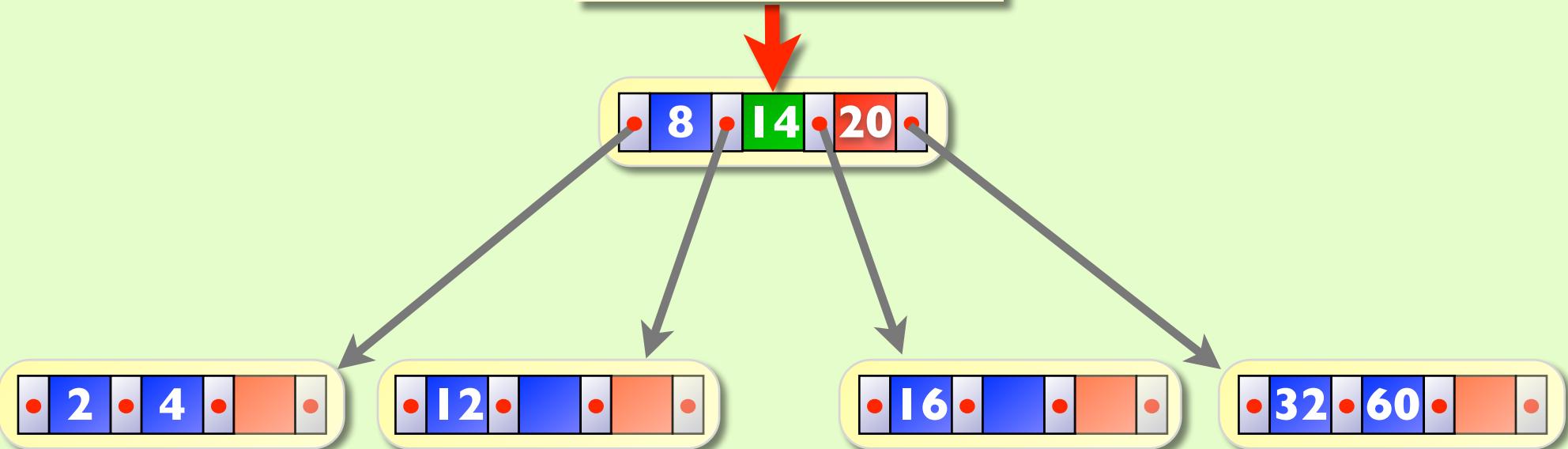


# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 14

Hier wird gesplittet!

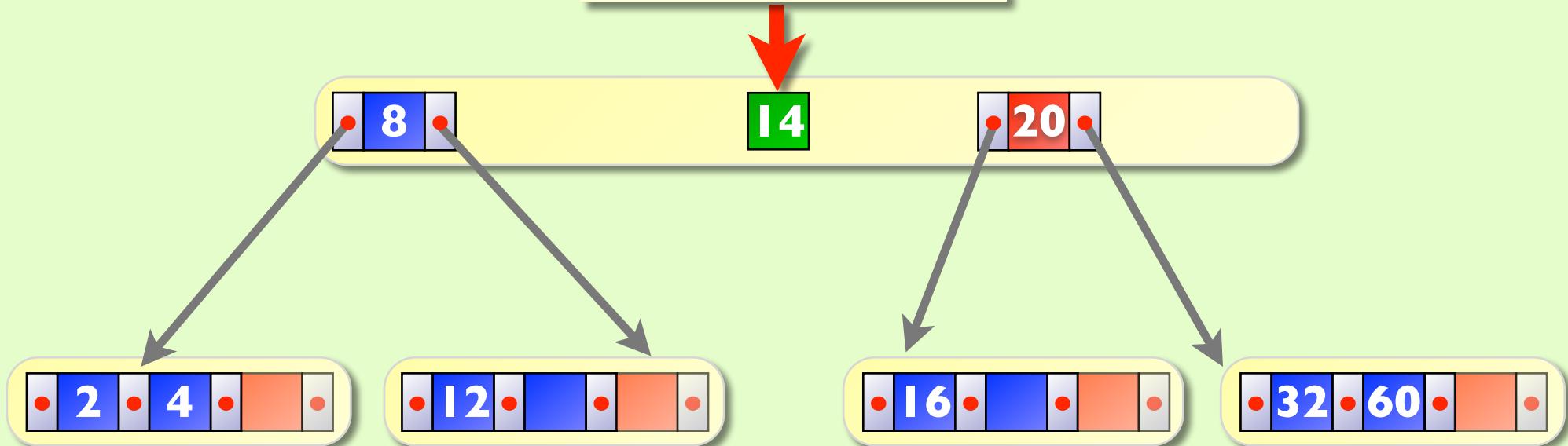


# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 14

Hier wird gesplittet!

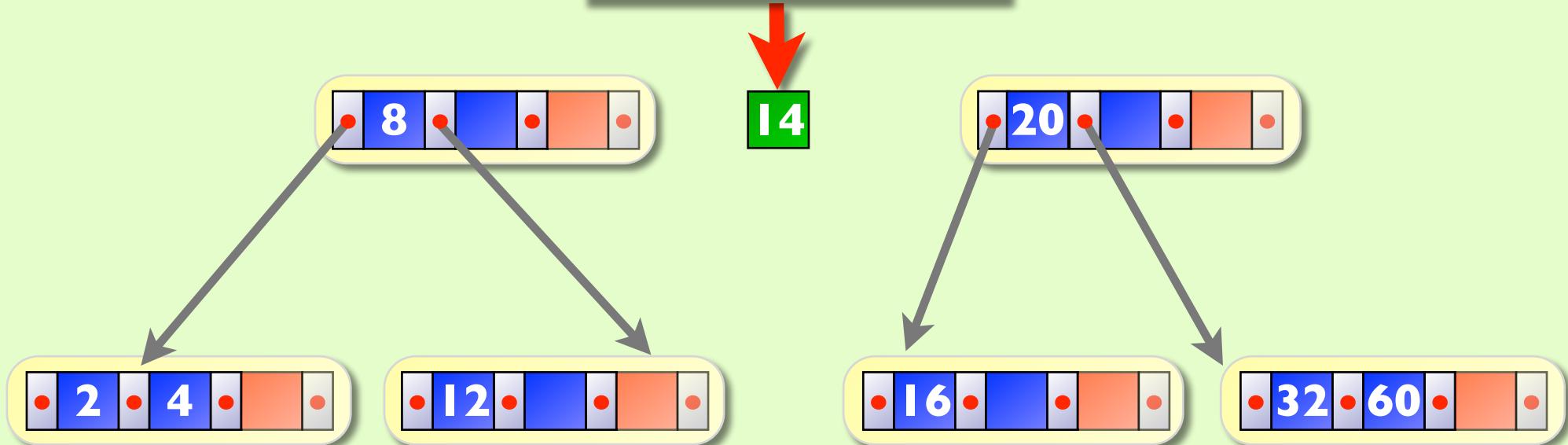


# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 14

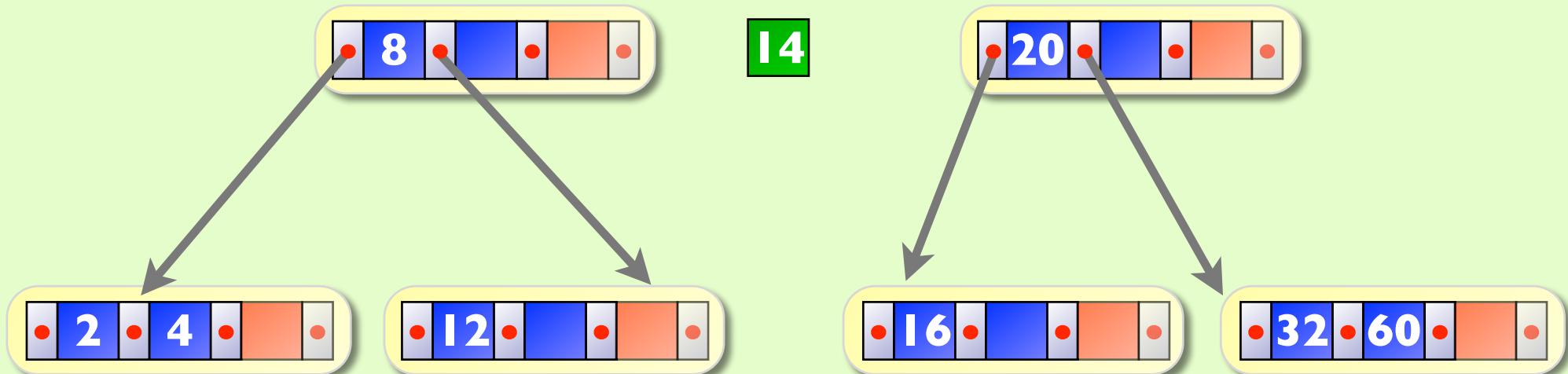
Hier wird gesplittet!



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

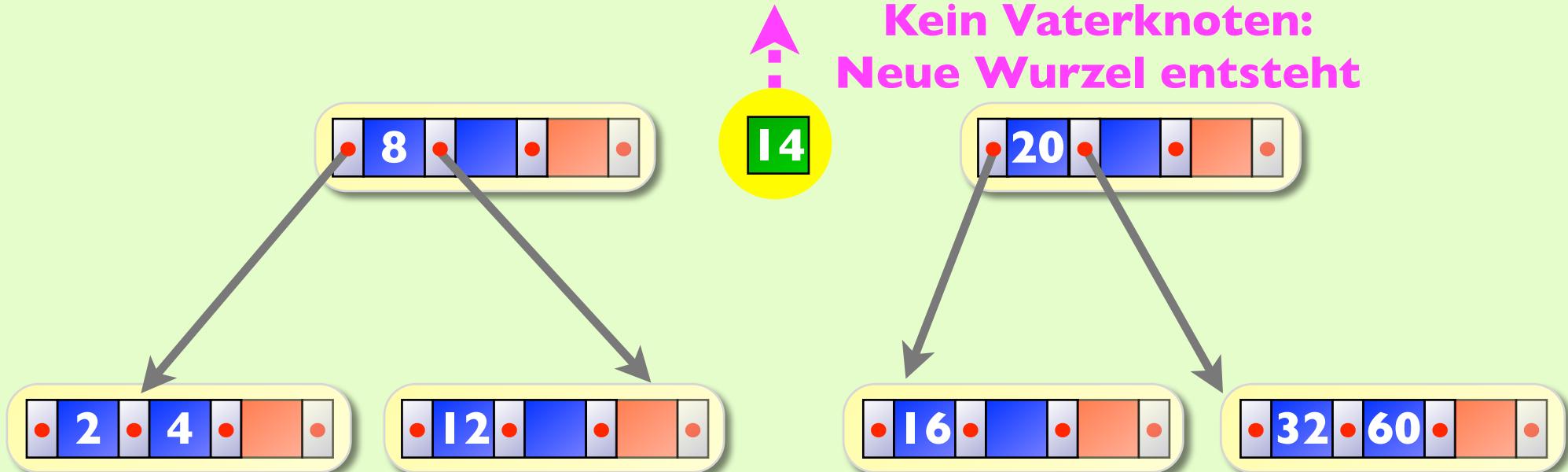
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

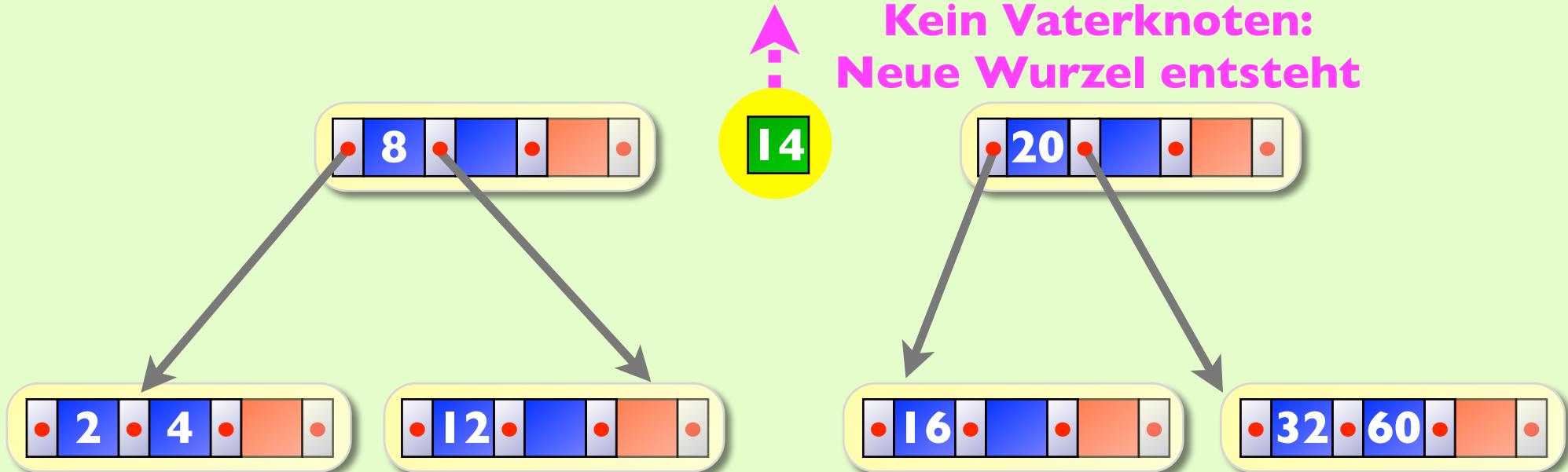
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

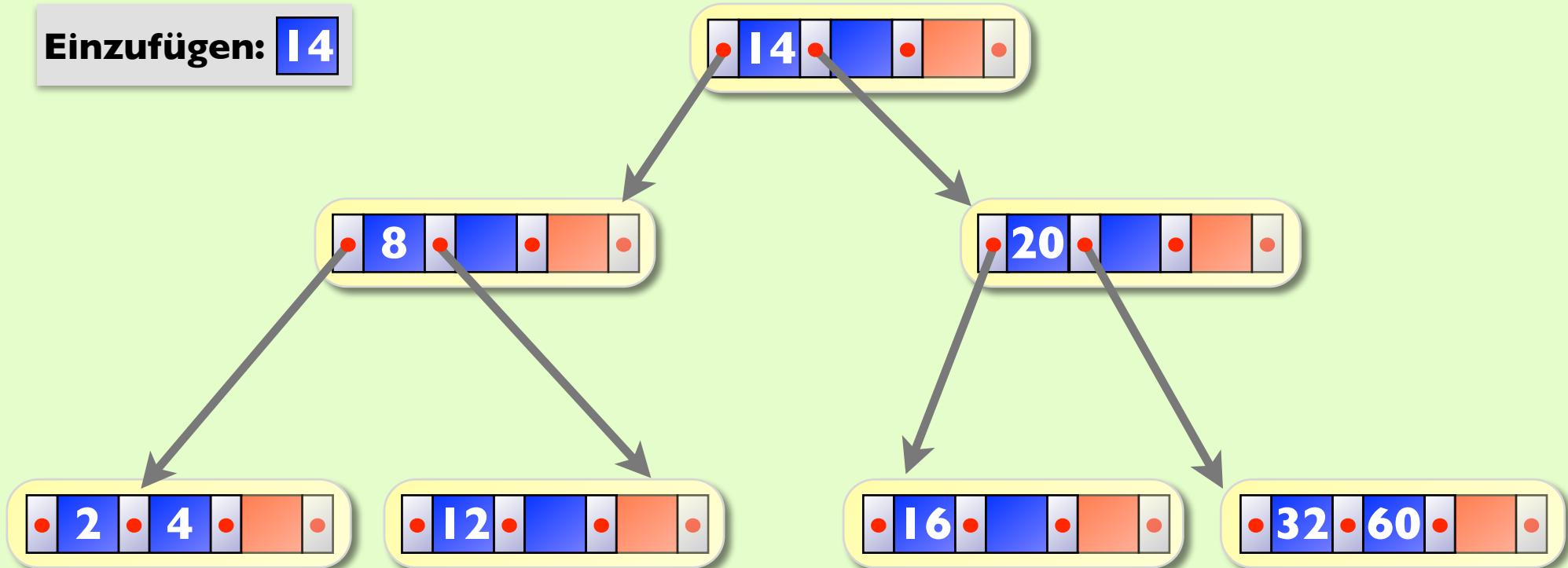
Einzufügen: 14



# Beispiel: Einfügen in einen B-Baum

## B B-Baum der Ordnung I - Einfügen

Einzufügen: 14



# Löschen im B-Baum der Ordnung $m$

# Löschen im B-Baum der Ordnung $m$

- Der zu löschende Schlüssel  $k$  wird zunächst gesucht - die Suche endet im Knoten  $v$

# Löschen im B-Baum der Ordnung $m$

- Der zu löschende Schlüssel  $k$  wird zunächst gesucht - die Suche endet im Knoten  $v$
- **Zwei Situationen sind zu unterscheiden:**

# Löschen im B-Baum der Ordnung $m$

- Der zu löschende Schlüssel  $k$  wird zunächst gesucht - die Suche ende im Knoten  $v$
- **Zwei Situationen sind zu unterscheiden:**
  - ① Der Knoten  $v$  hat nach Löschen von  $k$  **mindestens  $m$  Schlüssel**  
Dann sind wir fertig (*abgesehen von “trivialen” Umstrukturierungen*)

# Löschen im B-Baum der Ordnung $m$

- Der zu löschende Schlüssel  $k$  wird zunächst gesucht - die Suche ende im Knoten  $v$
- **Zwei Situationen sind zu unterscheiden:**
  - ① Der Knoten  $v$  hat nach Löschen von  $k$  **mindestens  $m$  Schlüssel**  
Dann sind wir fertig (*abgesehen von “trivialen” Umstrukturierungen*)
  - ② Der Knoten  $v$  hat nach Löschen von  $k$  **weniger als  $m$  Schlüssel**  
**(Unterlauf)**

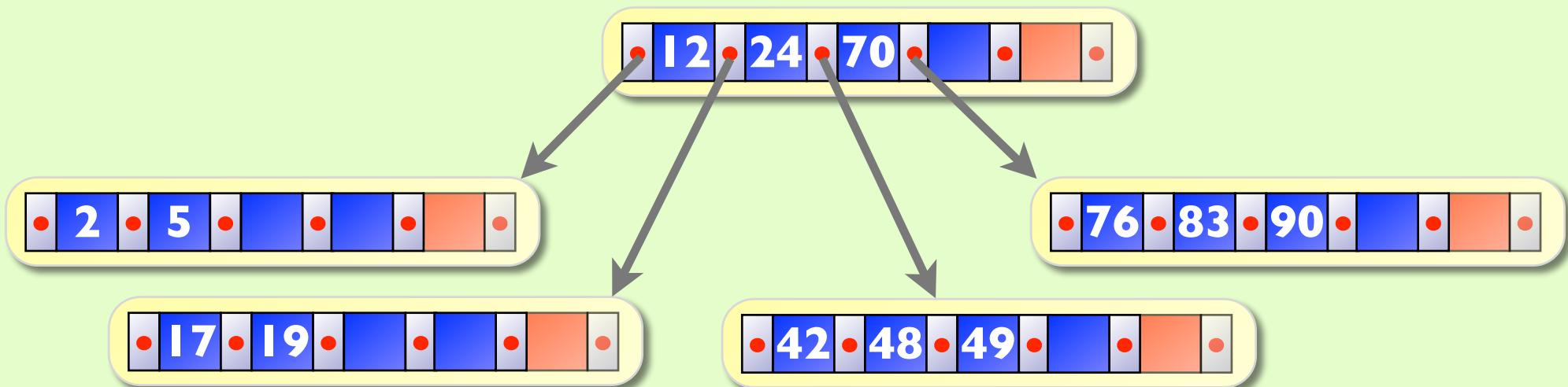
Der Unterlauf von  $v$  wird ausgeglichen durch

- **Balancieren** mit einem Bruderknoten; oder
- **Mischen** von  $v$  mit Vater- und Bruderknoten  
(falls Verschmelzen zu einem Überlauf führen würde)

# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

Zu löschen: 70

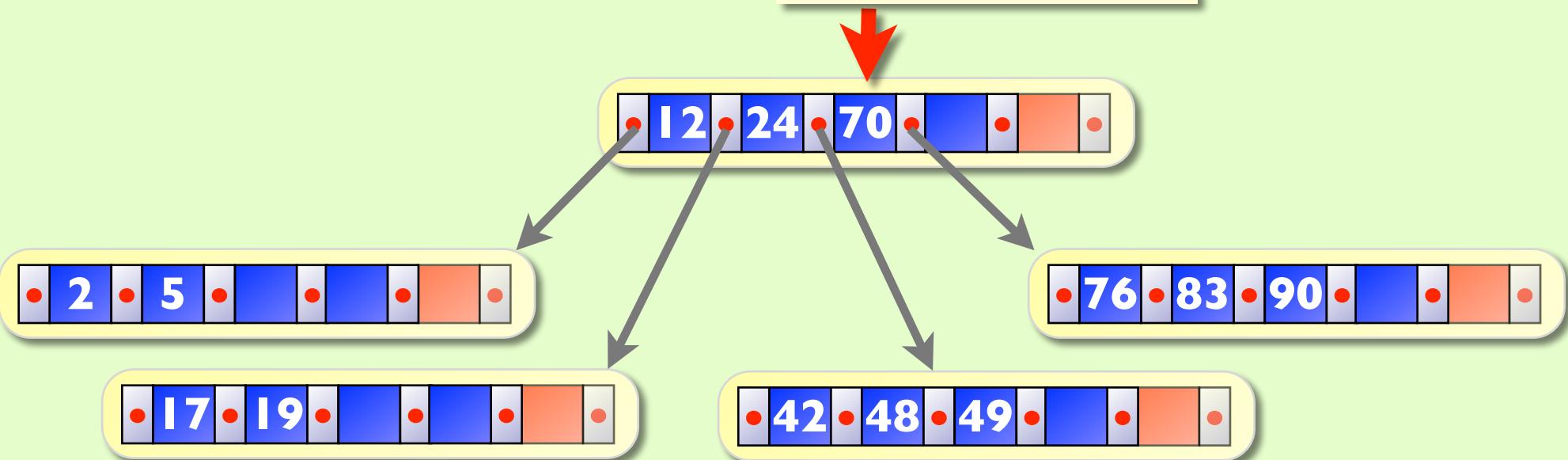


# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

Zu löschen: 70

Kein Unterlauf nach  
Löschen der 70

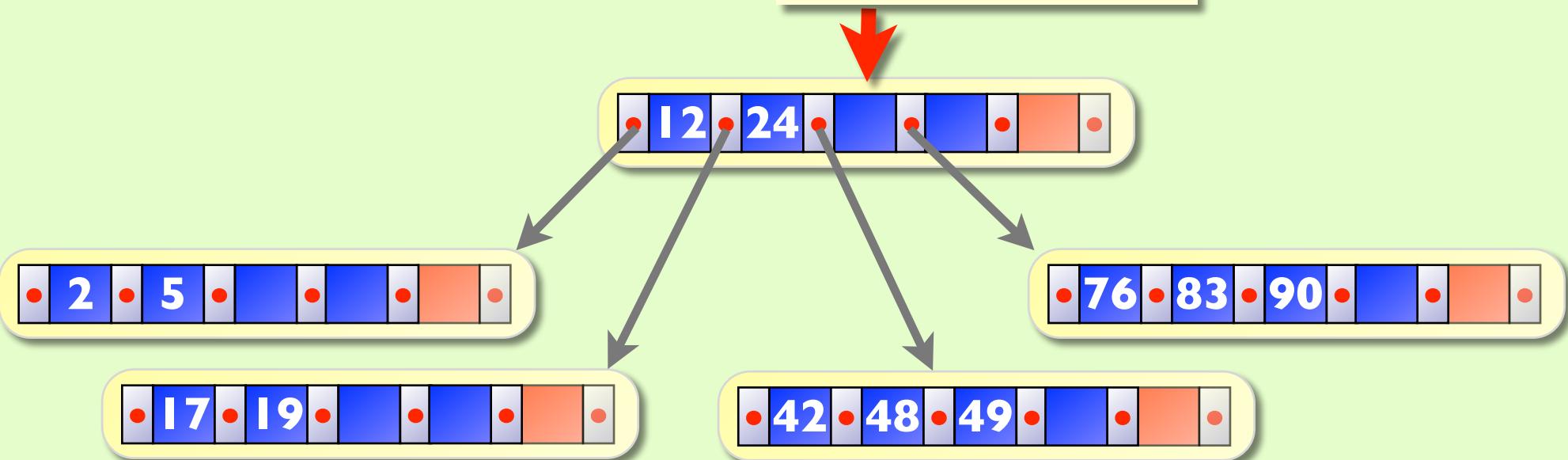


# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

Zu löschen: 70

Kein Unterlauf nach  
Löschen der 70

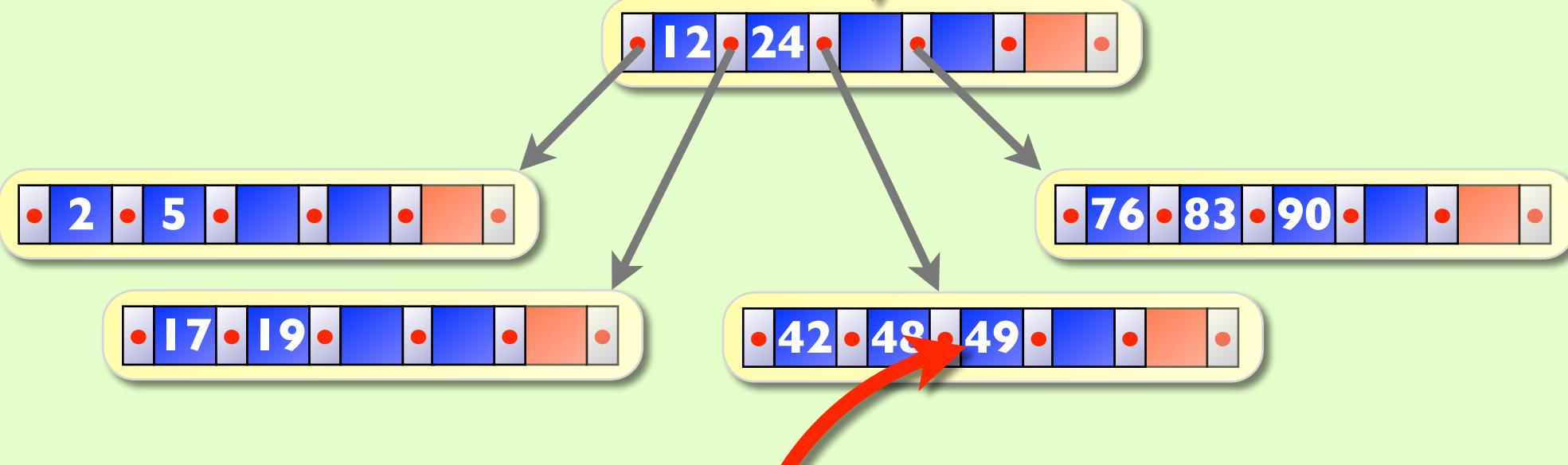


# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

Zu löschen: 70

Kein Unterlauf nach  
Löschen der 70



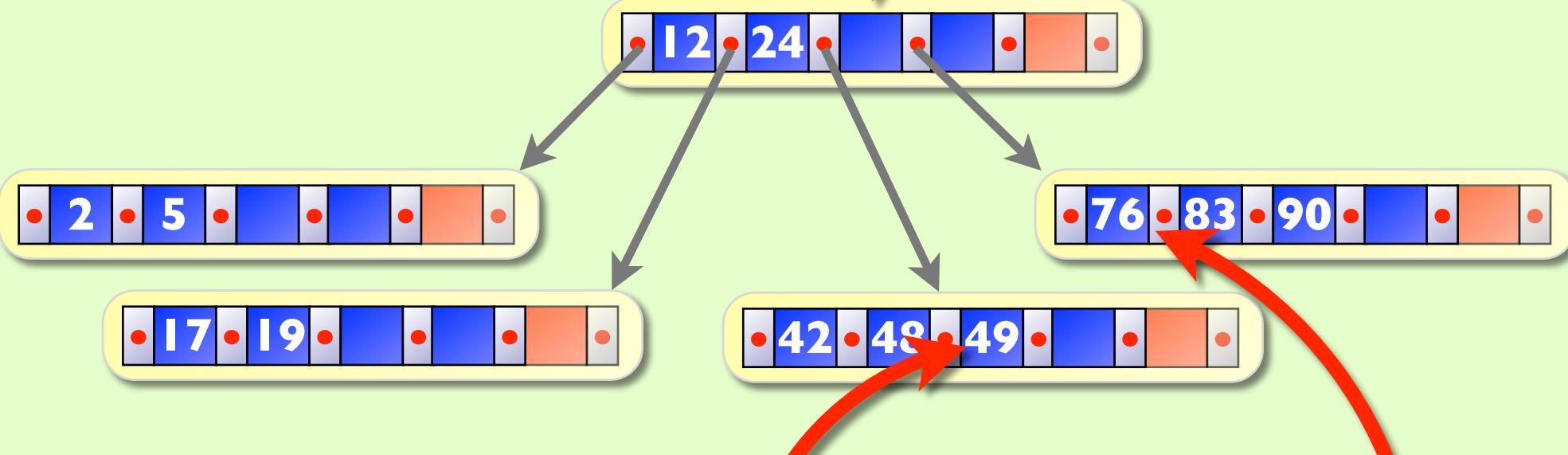
Bewege den größten  
Schlüssel (49) nach oben

# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

Zu löschen: 70

Kein Unterlauf nach  
Löschen der 70



Bewege den größten  
Schlüssel (49) nach oben

**oder**

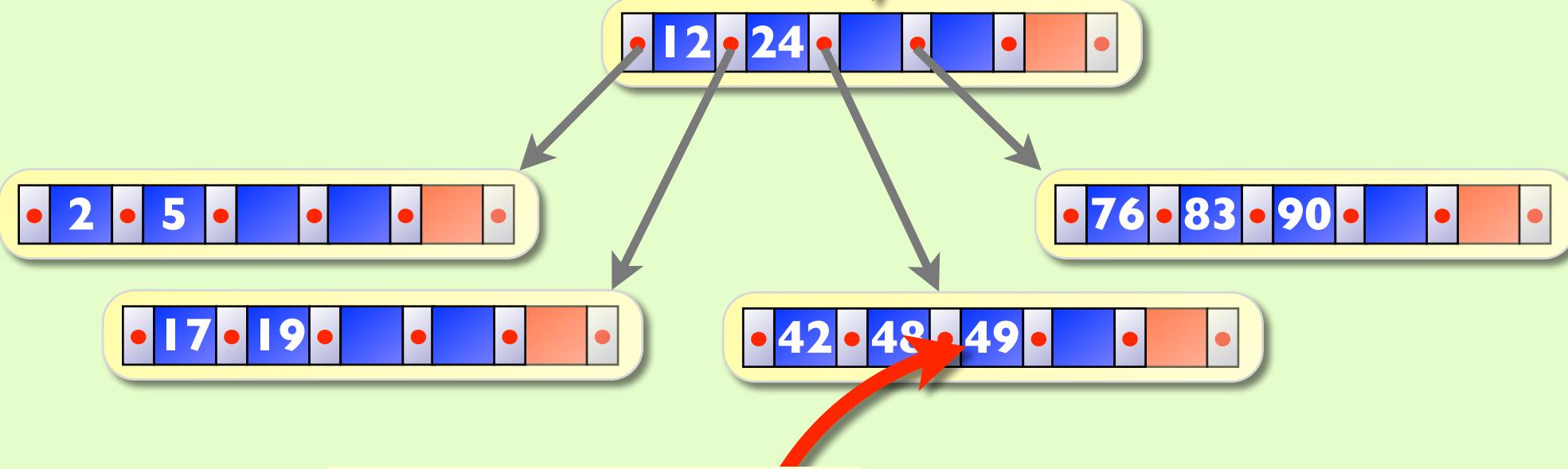
Bewege den kleinsten  
Schlüssel (76) nach oben

# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

Zu löschen: 70

Kein Unterlauf nach  
Löschen der 70



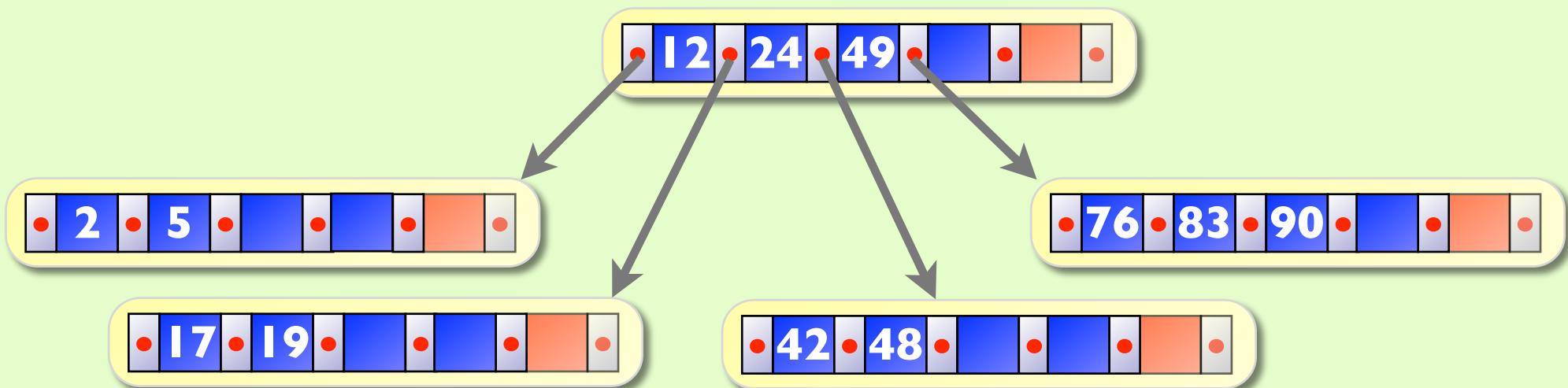
Bewege den größten  
Schlüssel (49) nach oben

**Geschickter. Warum?**

# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

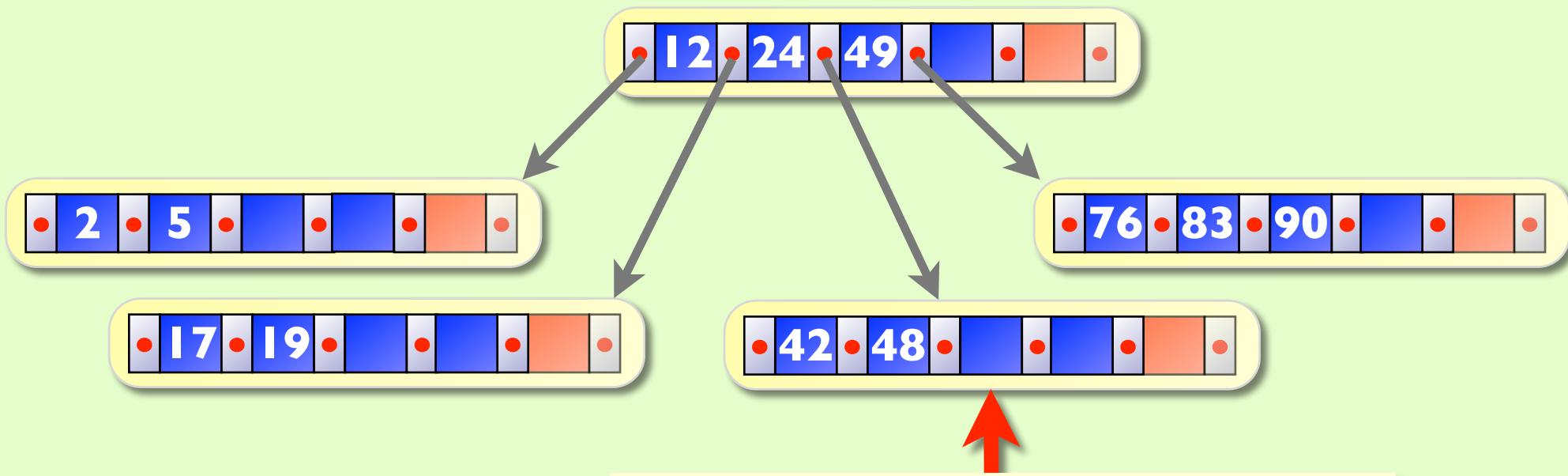
Zu löschen: 70



# Löschen: „Triviale“ Umstrukturierungen

## B B-Baum der Ordnung 2 - Triviale Umstrukturierungen ...

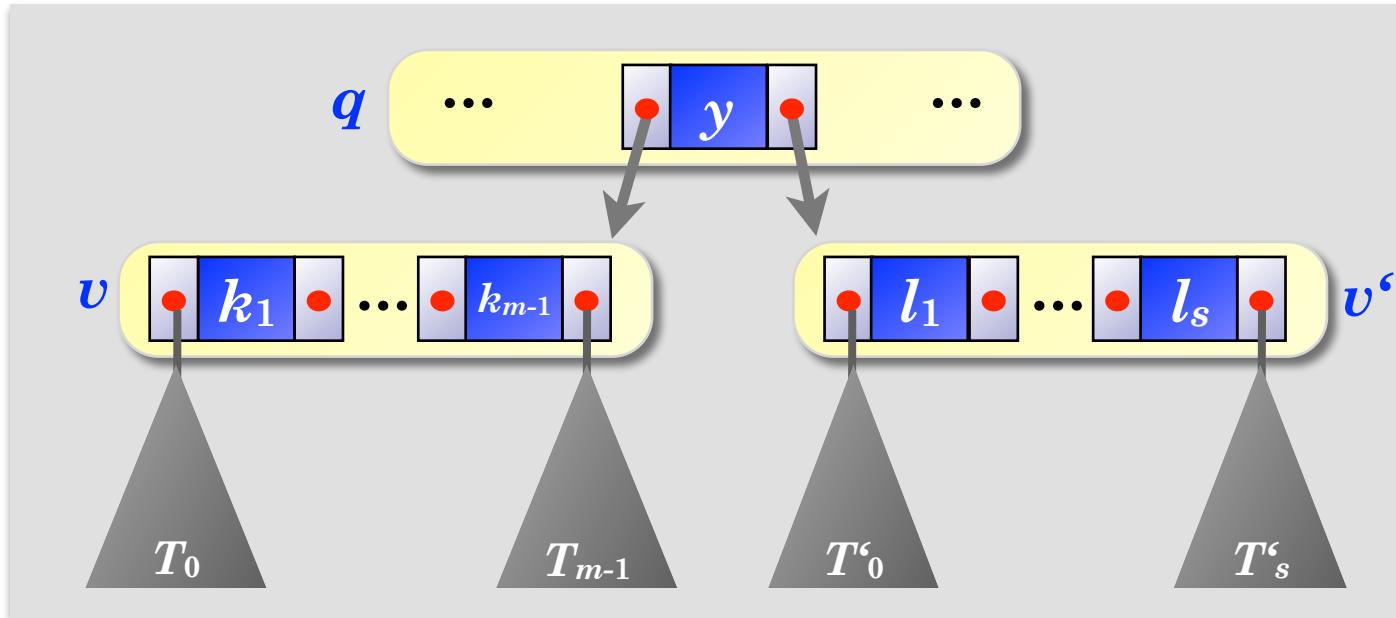
Zu löschen: 70



**Vorsicht!** Bei diesem Löschvorgang kann es zu einem Unterlauf kommen!

# B-Baum: Balancieren bei Unterlauf I

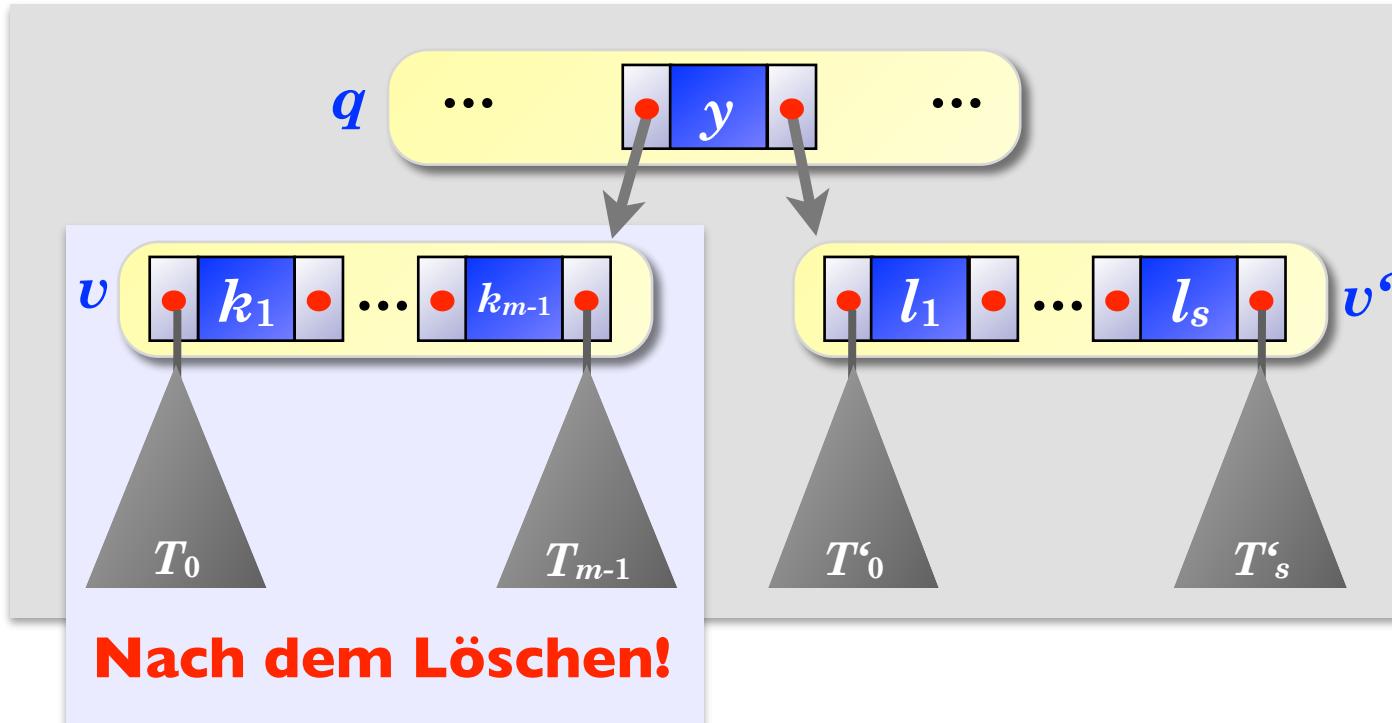
- Ausgangssituation:



alternativ:  
linker Bruder

# B-Baum: Balancieren bei Unterlauf I

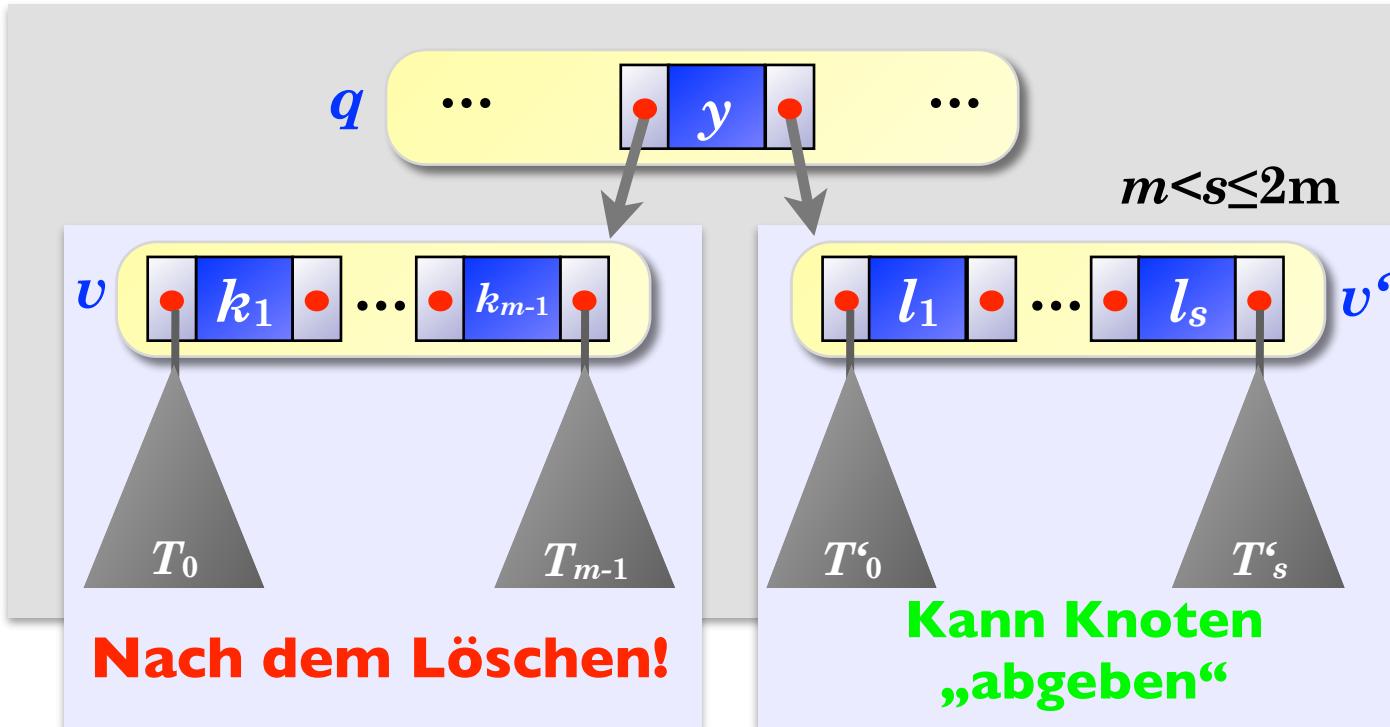
- Ausgangssituation:



alternativ:  
linker Bruder

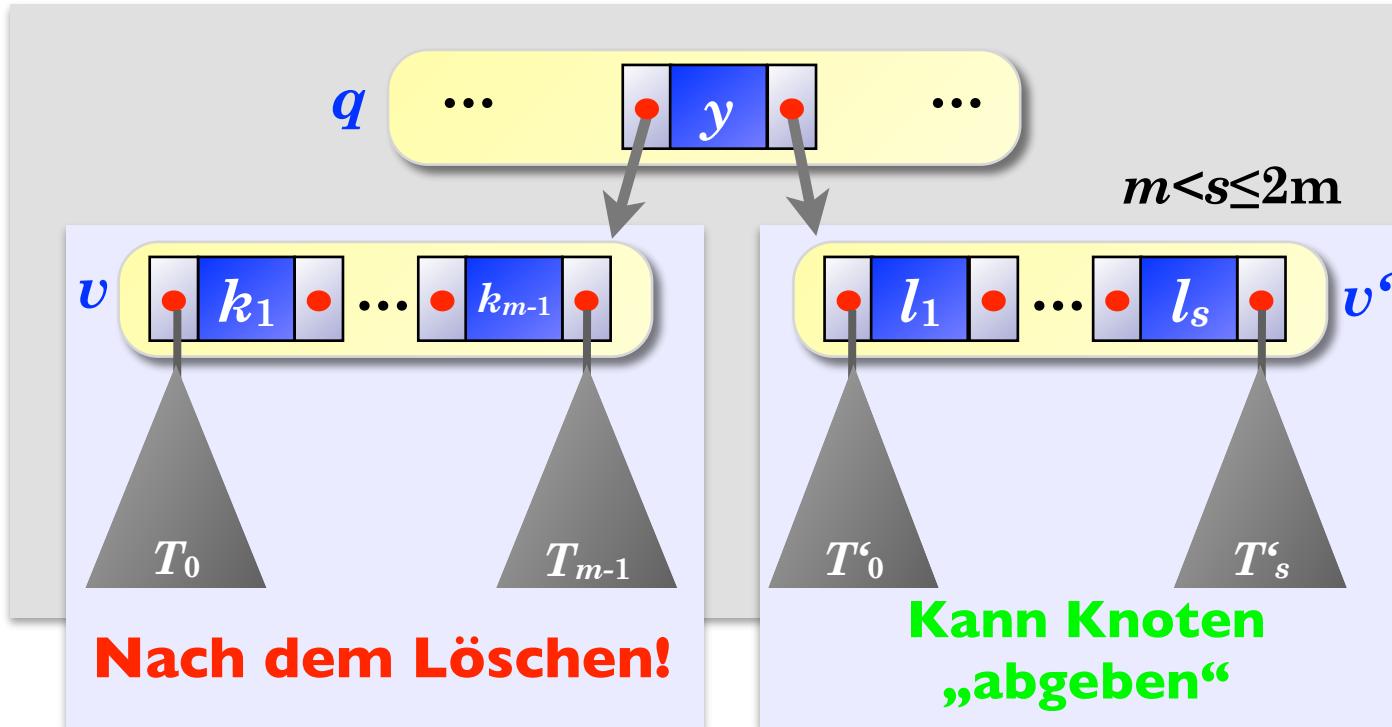
# B-Baum: Balancieren bei Unterlauf I

- Ausgangssituation:



# B-Baum: Balancieren bei Unterlauf I

- Ausgangssituation:



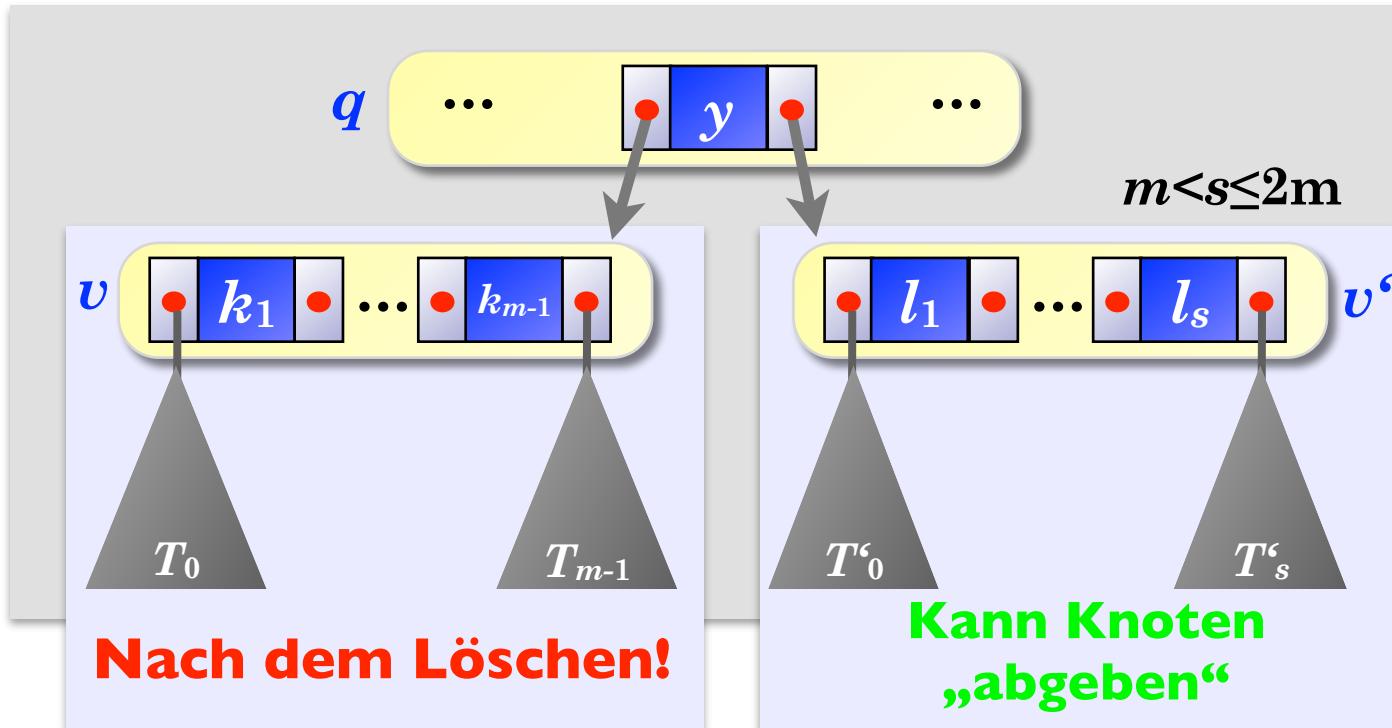
alternativ:  
linker Bruder

- Bilde „fiktiven Knoten“



# B-Baum: Balancieren bei Unterlauf I

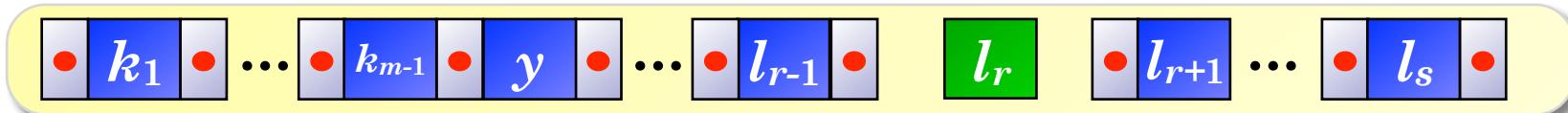
- Ausgangssituation:



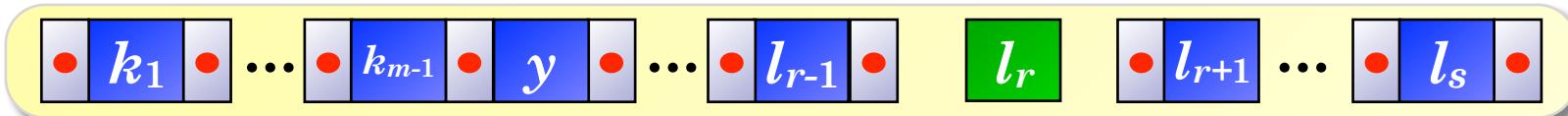
- Bilde „fiktiven Knoten“



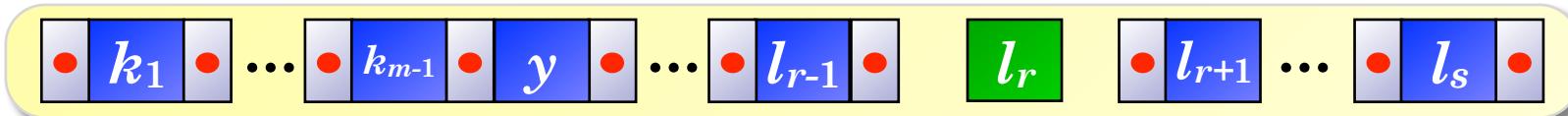
- Teile diesen am Schlüssel  $l_r$  mit  $r = \left\lceil \frac{(m-1)+1+s}{2} \right\rceil - ((m-1)+1)$



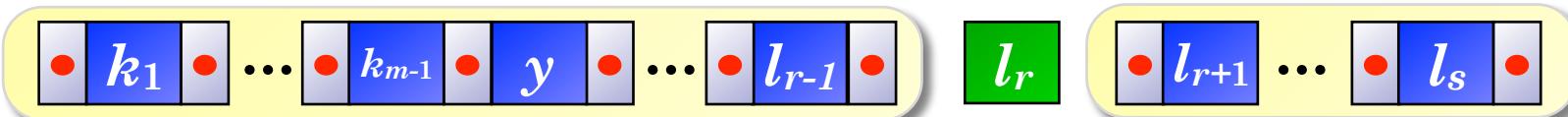
# B-Baum: Balancieren bei Unterlauf II



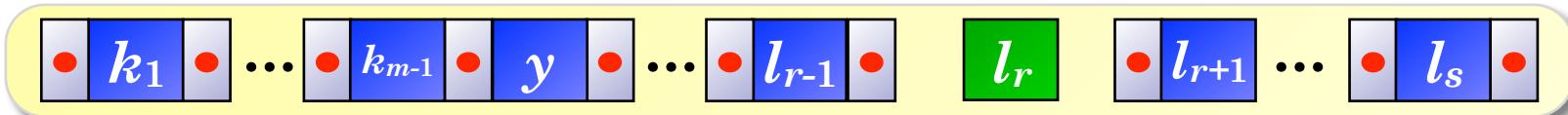
# B-Baum: Balancieren bei Unterlauf II



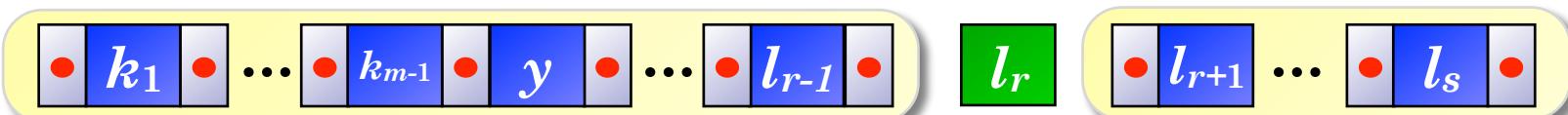
- **fiktiver Knoten wird gespalten:**



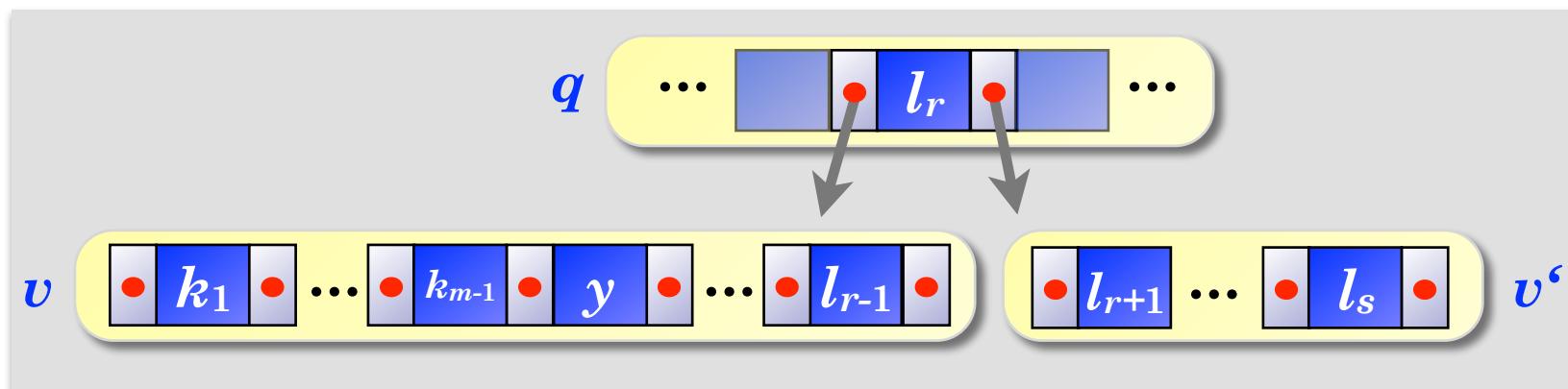
# B-Baum: Balancieren bei Unterlauf II



- **fiktiver Knoten wird gespalten:**

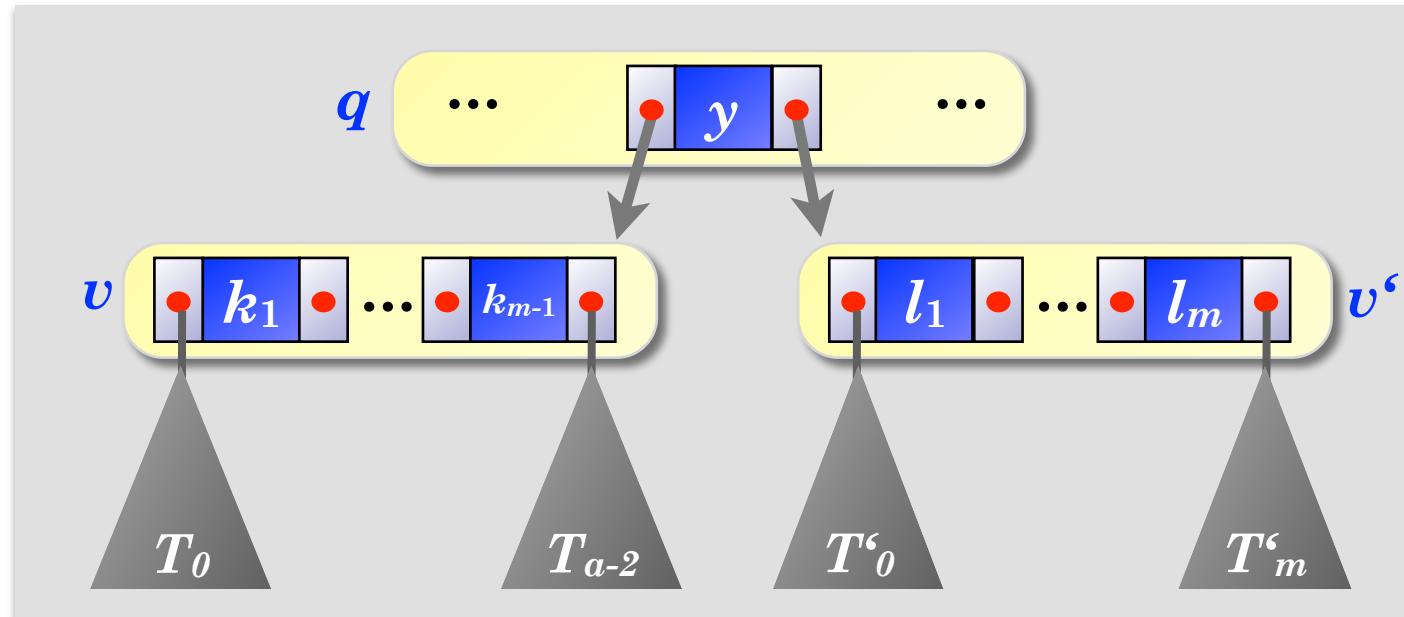


- **$l_r$  wandert in die Wurzel**  
an die stelle, wo vorher  $y$  war



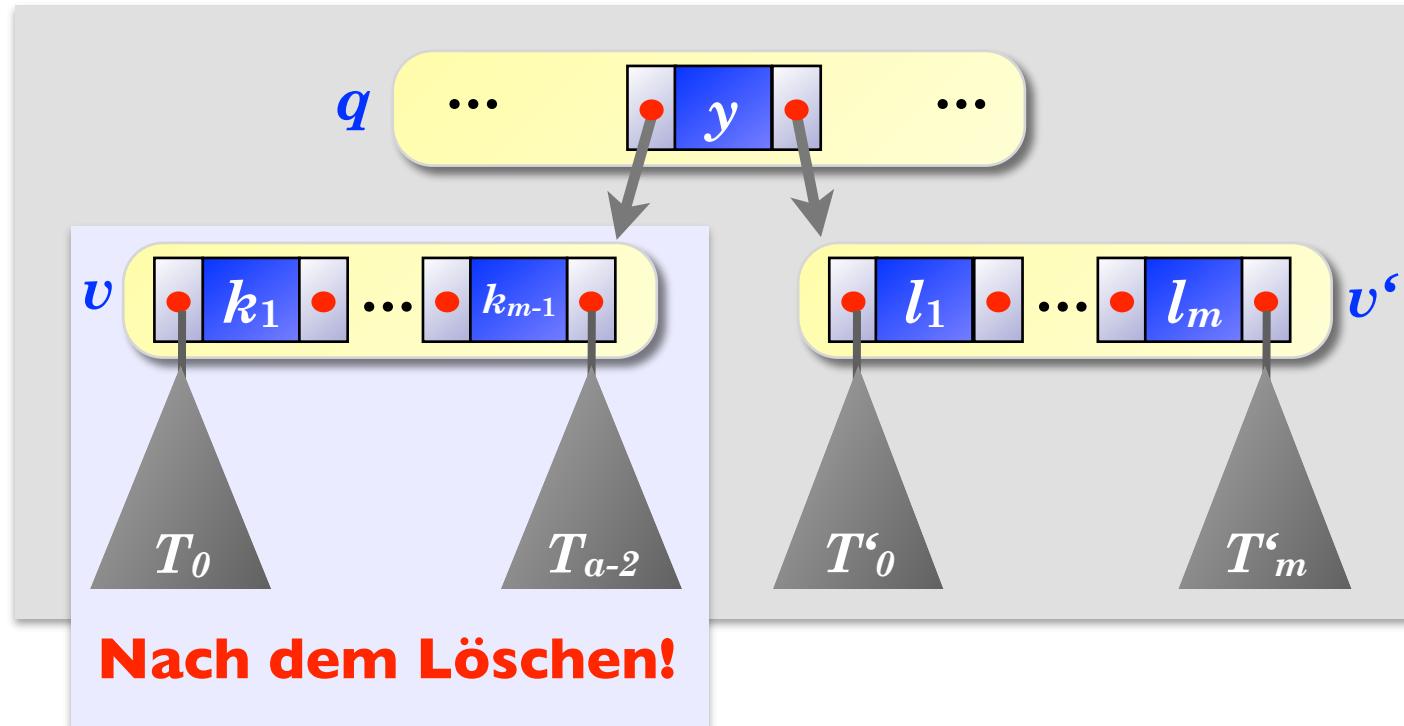
# B-Baum: Mischen bei Unterlauf I

- Balancieren mit keinem Bruderknoten möglich - neuen Knoten aus Bruder und Wurzel zusammenmischen
- **Ausgangssituation:**



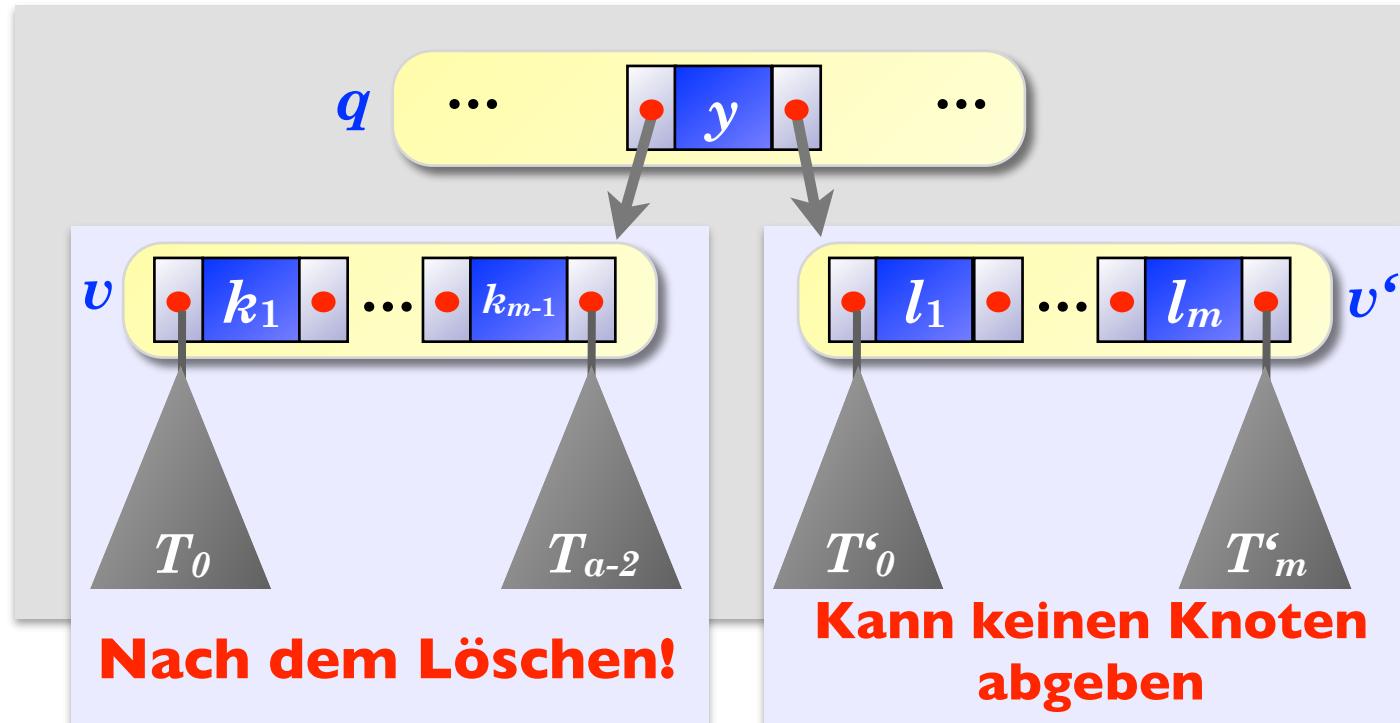
# B-Baum: Mischen bei Unterlauf I

- Balancieren mit keinem Bruderknoten möglich - neuen Knoten aus Bruder und Wurzel zusammenmischen
- **Ausgangssituation:**



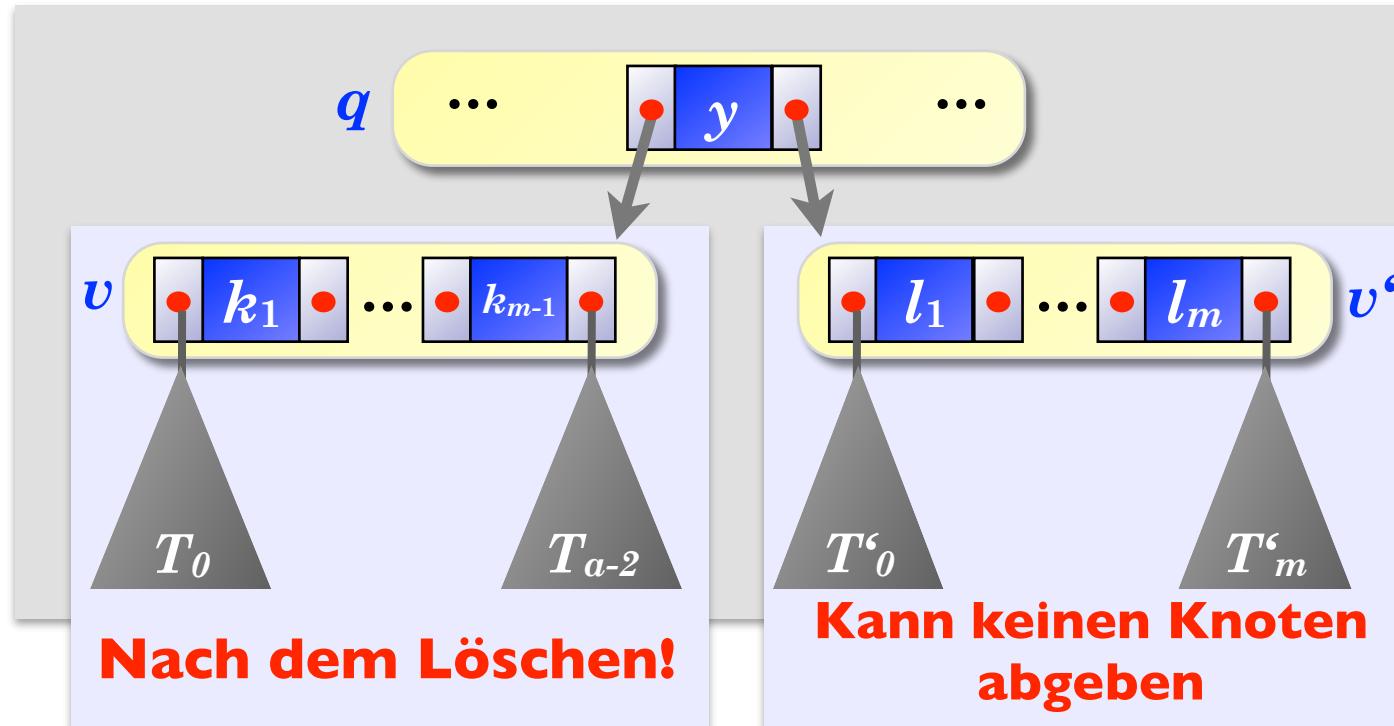
# B-Baum: Mischen bei Unterlauf I

- Balancieren mit keinem Bruderknoten möglich - neuen Knoten aus Bruder und Wurzel zusammenmischen
- **Ausgangssituation:**

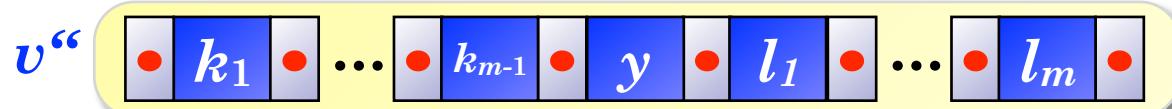


# B-Baum: Mischen bei Unterlauf I

- Balancieren mit keinem Bruderknoten möglich - neuen Knoten aus Bruder und Wurzel zusammenmischen
- **Ausgangssituation:**



- Bilde neuen Knoten  $v''$  aus  $v, v'$  und  $y$ :



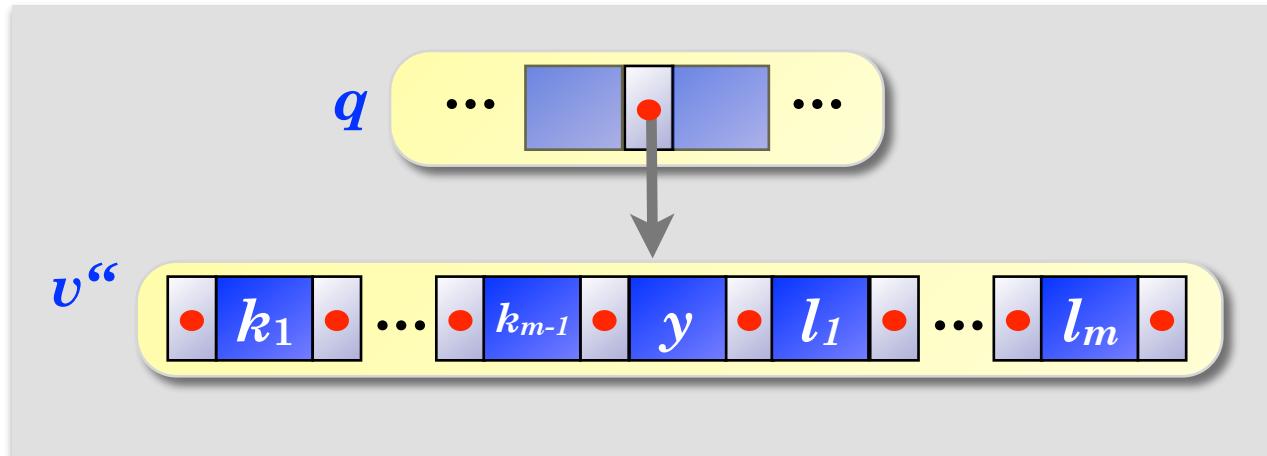
# B-Baum: Mischen bei Unterlauf II



# B-Baum: Mischen bei Unterlauf II



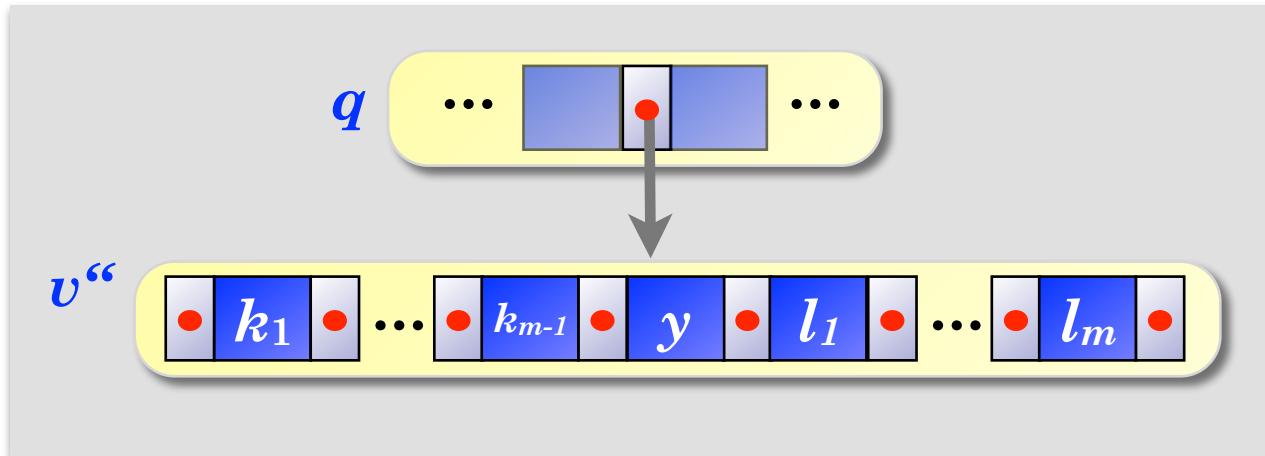
- Entferne  $y$  aus der Wurzel:



# B-Baum: Mischen bei Unterlauf II



- Entferne  $y$  aus der Wurzel:



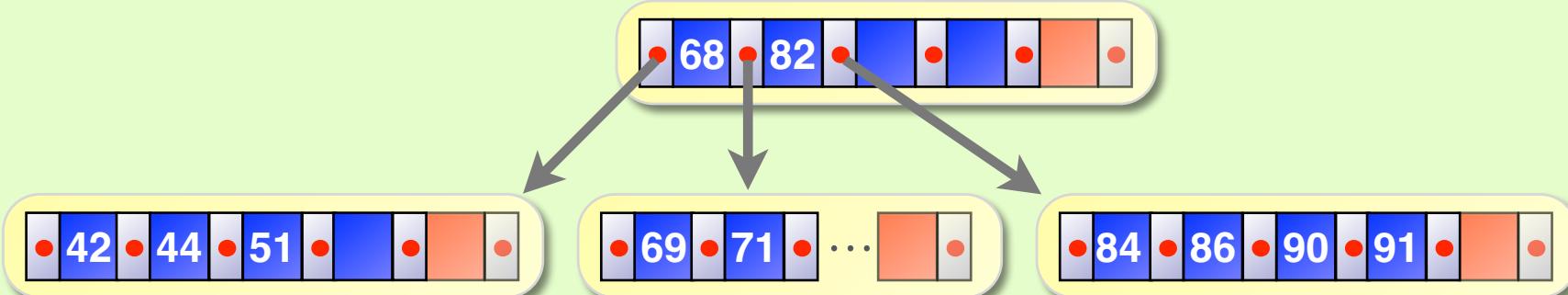
- Anmerkungen:

- u.U. kommt es am Knoten  $q$  erneut zu einem Unterlauf
- war  $q$  die Wurzel und  $y$  der letzte Schlüssel, wird  $v''$  zur neuen Wurzel

# Löschen - Einfachster Fall

## B B-Baum der Ordnung 2 - Löschen

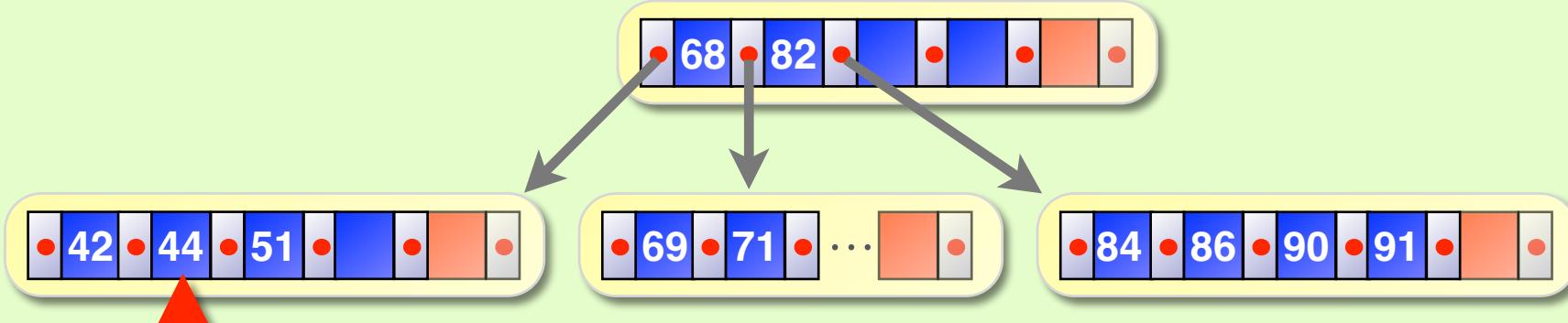
Zu löschen: 44



# Löschen - Einfachster Fall

## B B-Baum der Ordnung 2 - Löschen

Zu löschen: 44

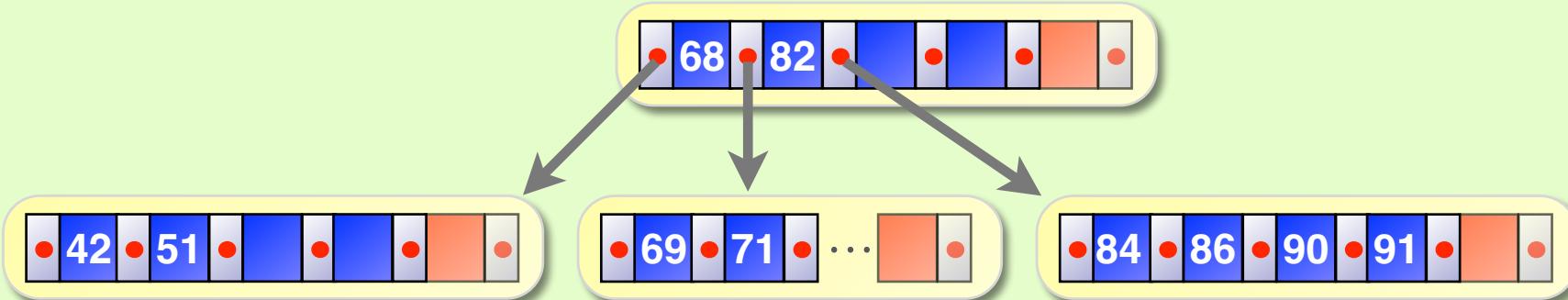


Kein Unterlauf nach  
Löschen der 44

Schlüssel entfernen  
und Knoten umsortieren

# Löschen - Balancieren 1

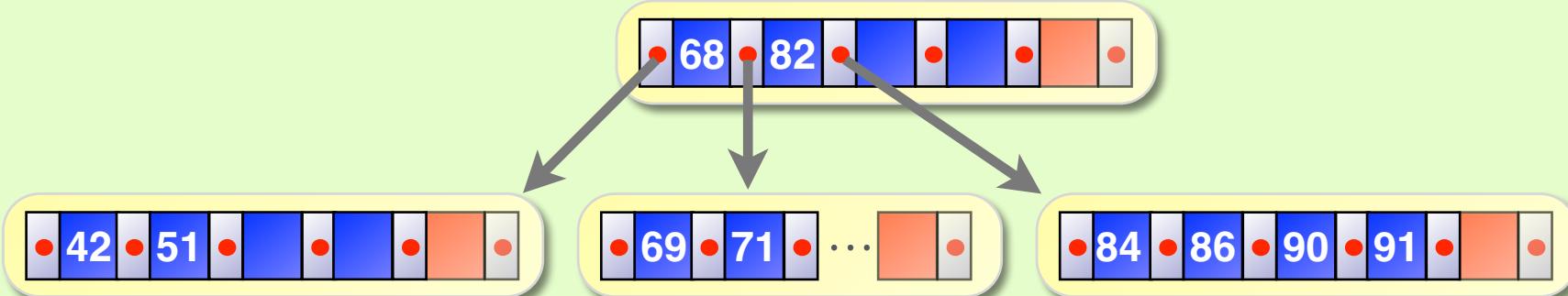
## B B-Baum der Ordnung 2 - Löschen



# Löschen - Balancieren 1

## B B-Baum der Ordnung 2 - Löschen

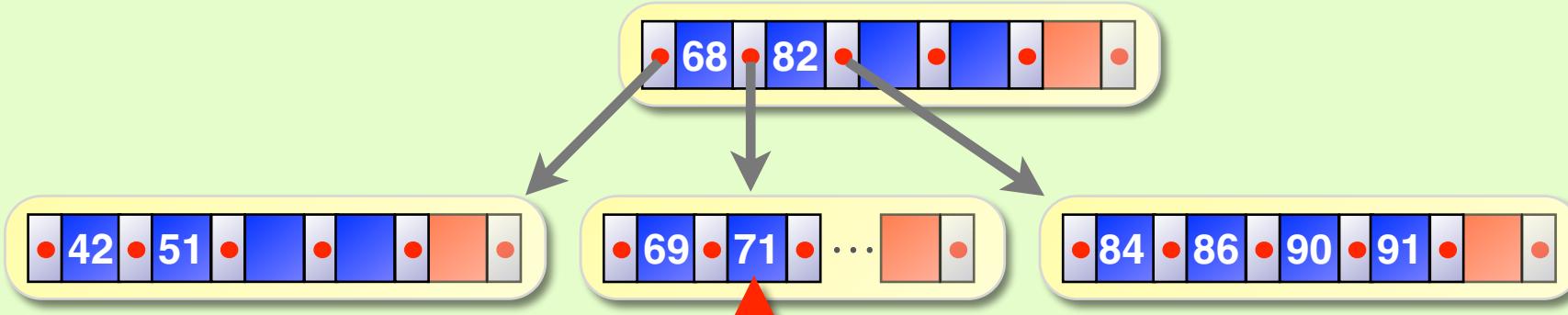
Zu löschen: 71



# Löschen - Balancieren 1

## B B-Baum der Ordnung 2 - Löschen

Zu löschen: 71



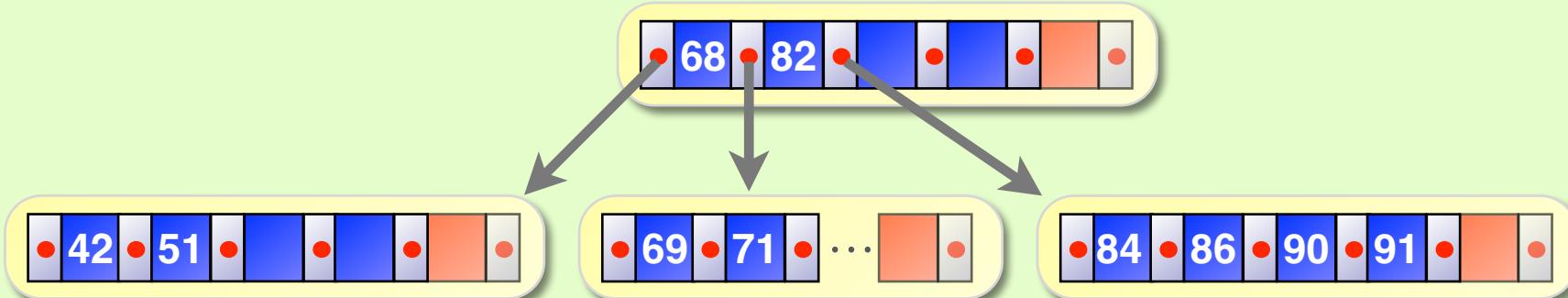
Unterlauf nach  
Löschen der 71

Rechter Geschwisterknoten  
kann (ggf. mehrere) Schlüssel abgeben  
⇒ **Balancieren**

# Löschen - Balancieren 2

## B B-Baum der Ordnung 2 - Löschen

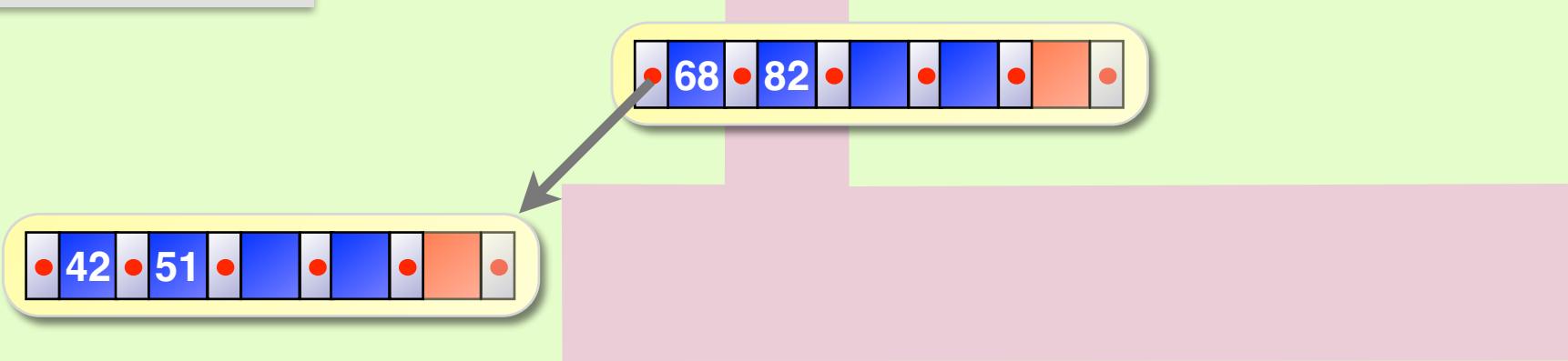
Zu löschen: 71



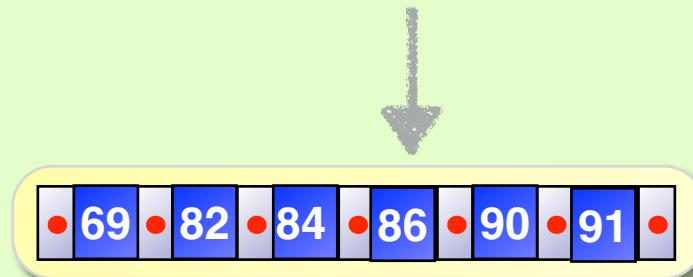
# Löschen - Balancieren 2

## B B-Baum der Ordnung 2 - Löschen

Zu löschen: 71



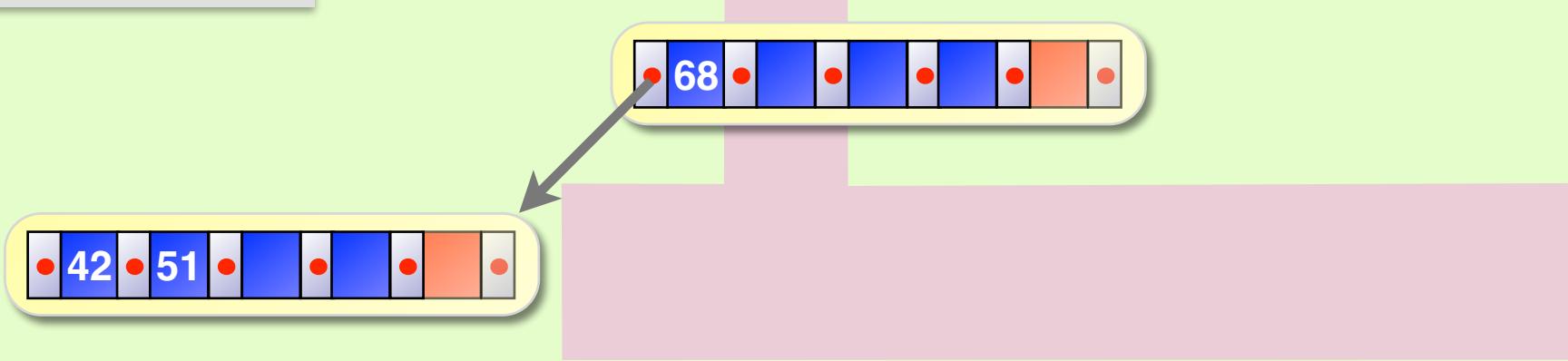
Beteiligte Schlüssel in fiktiven Knoten kopieren



# Löschen - Balancieren 3

## B B-Baum der Ordnung 2 - Löschen

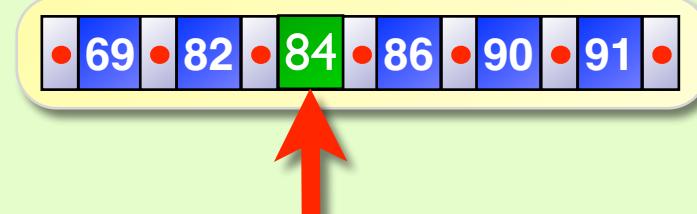
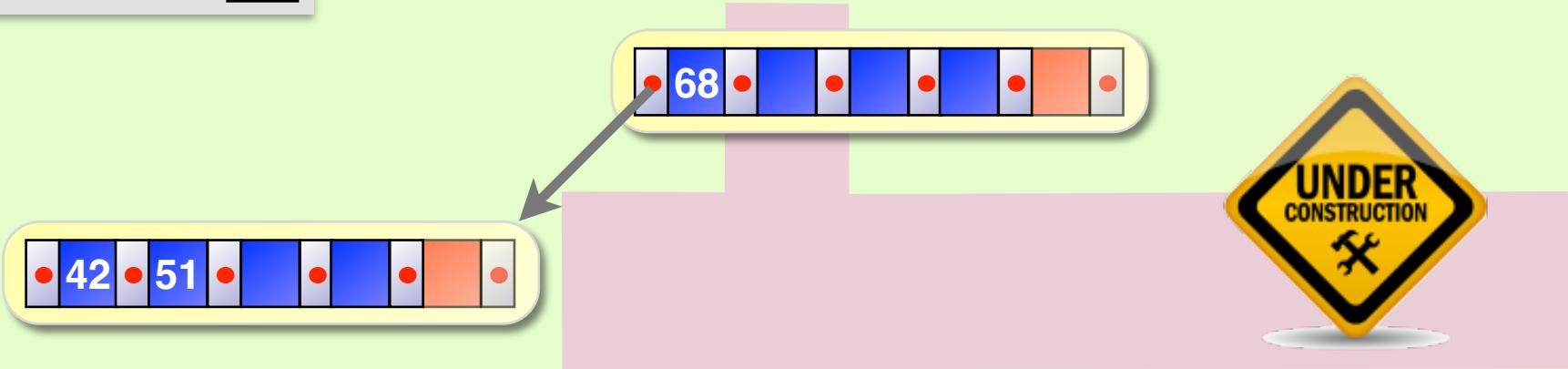
Zu löschen: 71



# Löschen - Balancieren 3

## B B-Baum der Ordnung 2 - Löschen

Zu löschen: 71

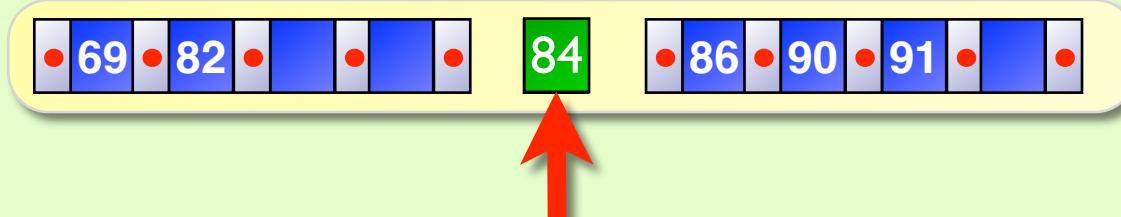
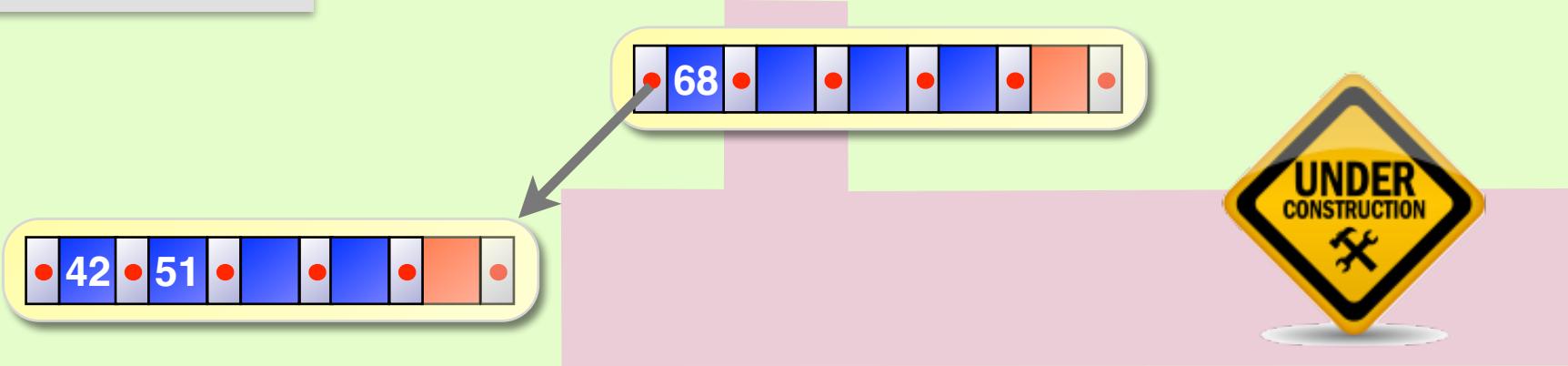


Split an Indexposition  $\left\lceil \frac{6}{2} \right\rceil = 3$

# Löschen - Balancieren 4

## B B-Baum der Ordnung 2 - Löschen

Zu löschen: 71

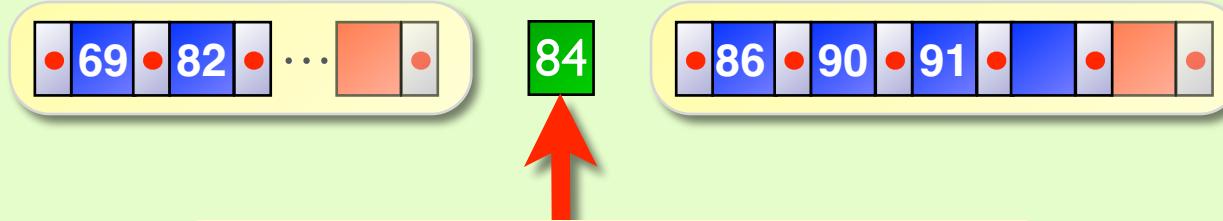
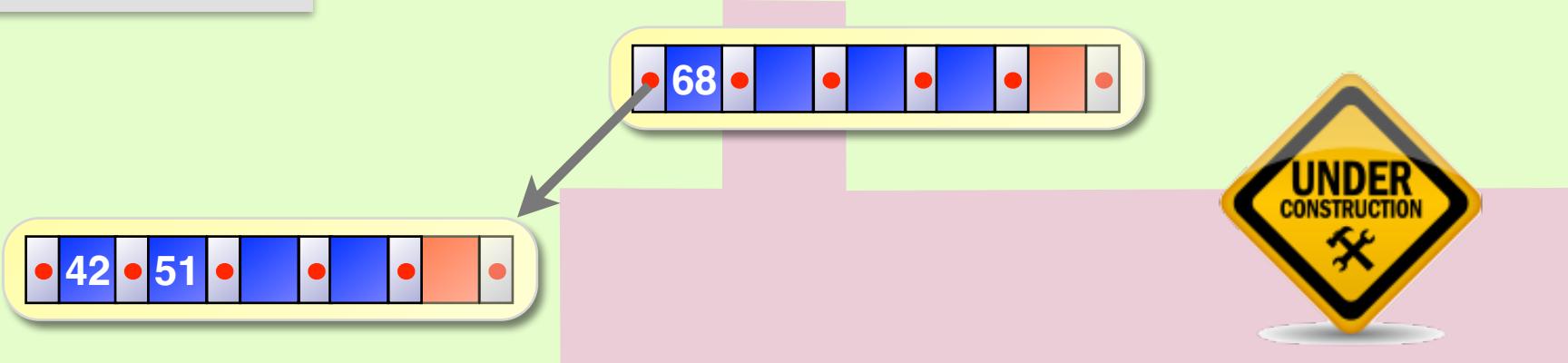


Split an Indexposition  $\left\lceil \frac{6}{2} \right\rceil = 3$

# Löschen - Balancieren 5

## B B-Baum der Ordnung 2 - Löschen

Zu löschen: 71

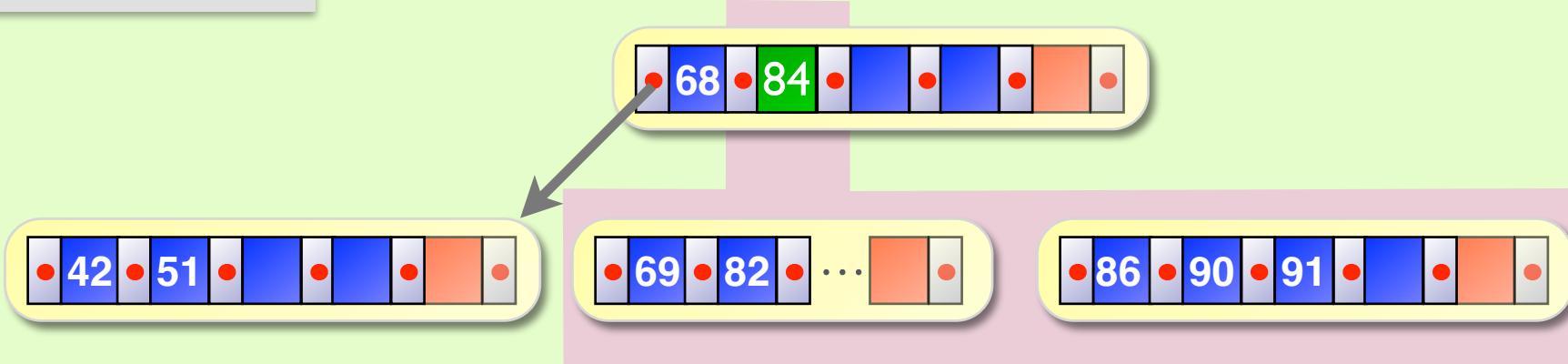


Split an Indexposition  $\left\lceil \frac{6}{2} \right\rceil = 3$

# Löschen - Balancieren 5

## B B-Baum der Ordnung 2 - Löschen

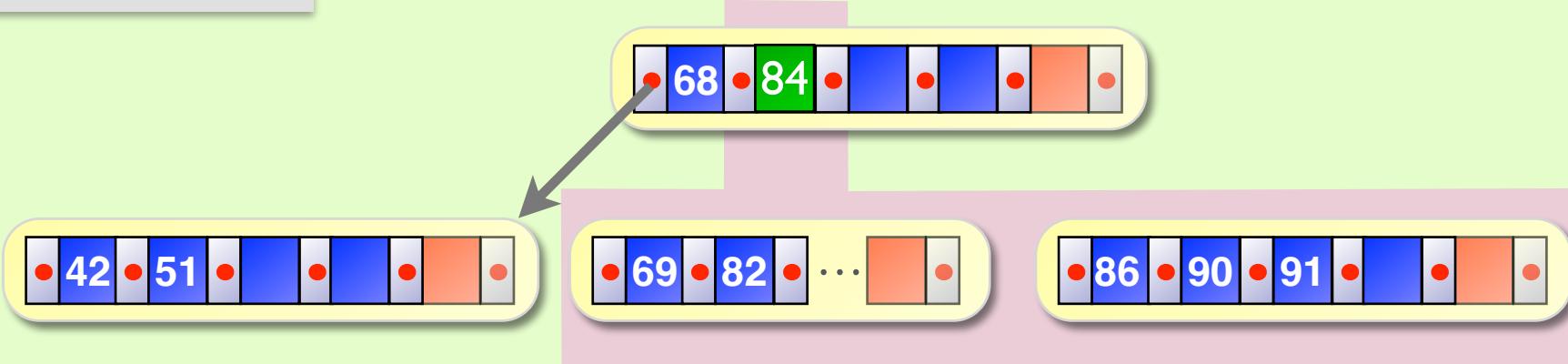
Zu löschen: 71



# Löschen - Balancieren 6

## B B-Baum der Ordnung 2 - Löschen

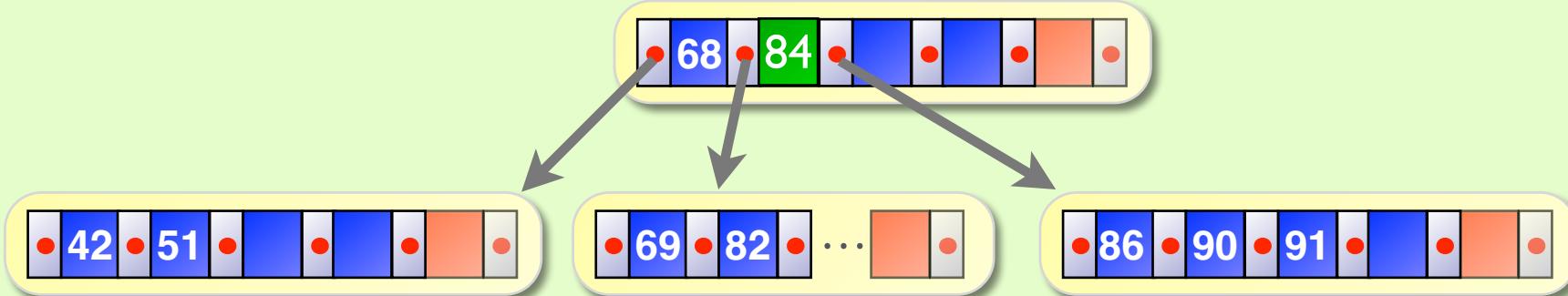
Zu löschen: 71



# Löschen - Balancieren 6

## B B-Baum der Ordnung 2 - Löschen

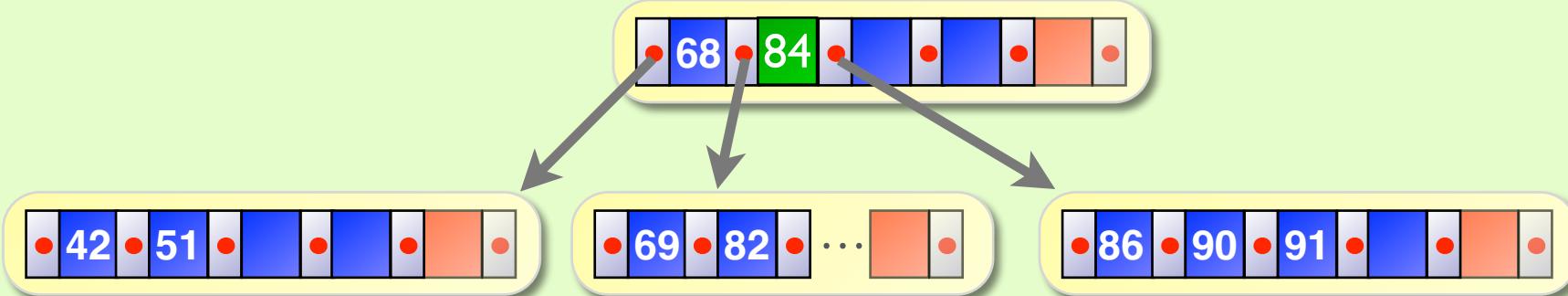
Zu löschen: 71



# Löschen - Balancieren 6

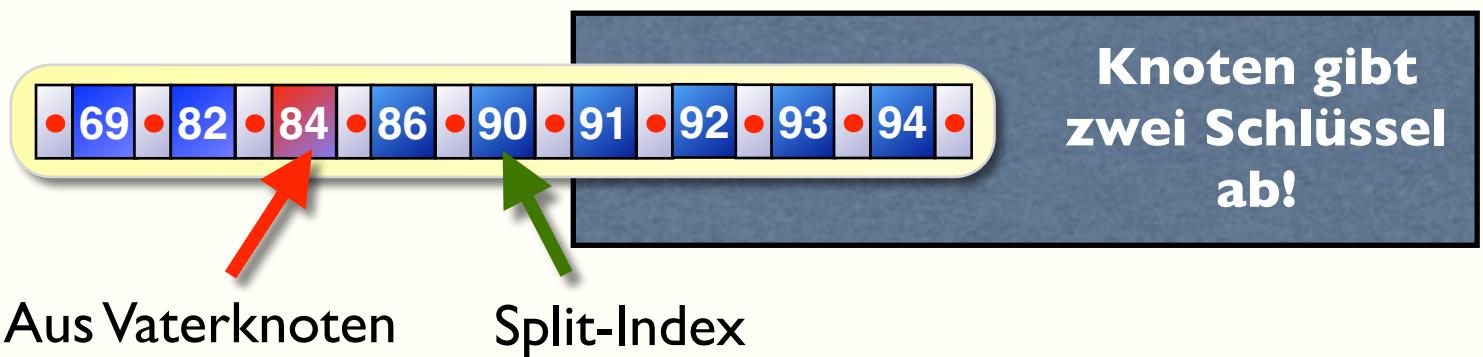
## B B-Baum der Ordnung 2 - Löschen

Zu löschen: 71



**Beachte: Keine Rotation!**

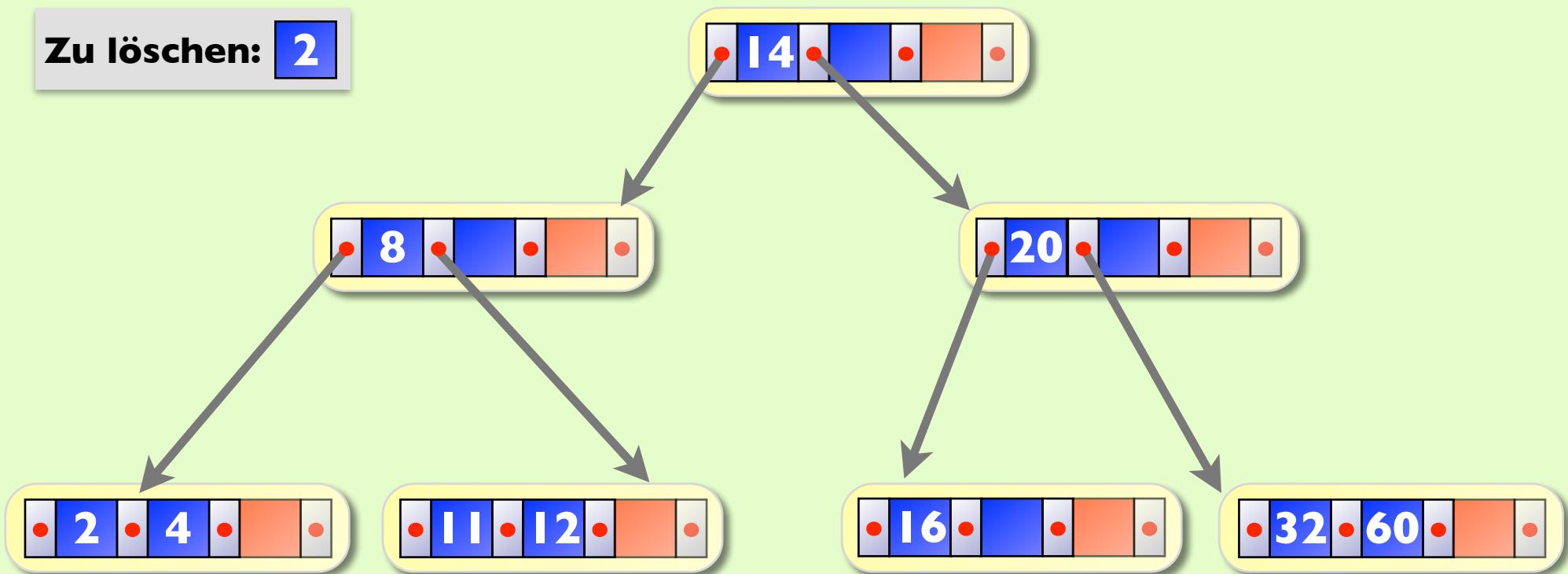
Geschwisterknoten kann (je nach Ordnung des B-Baumes / Füllgrad des Knotens) ggf. mehrere Schlüssel abgeben. Betrachte B-Baum der Ordnung 3 und fiktiven Knoten



# Beispiel: Löschen in einen B-Baum

## B B-Baum der Ordnung I - Löschen

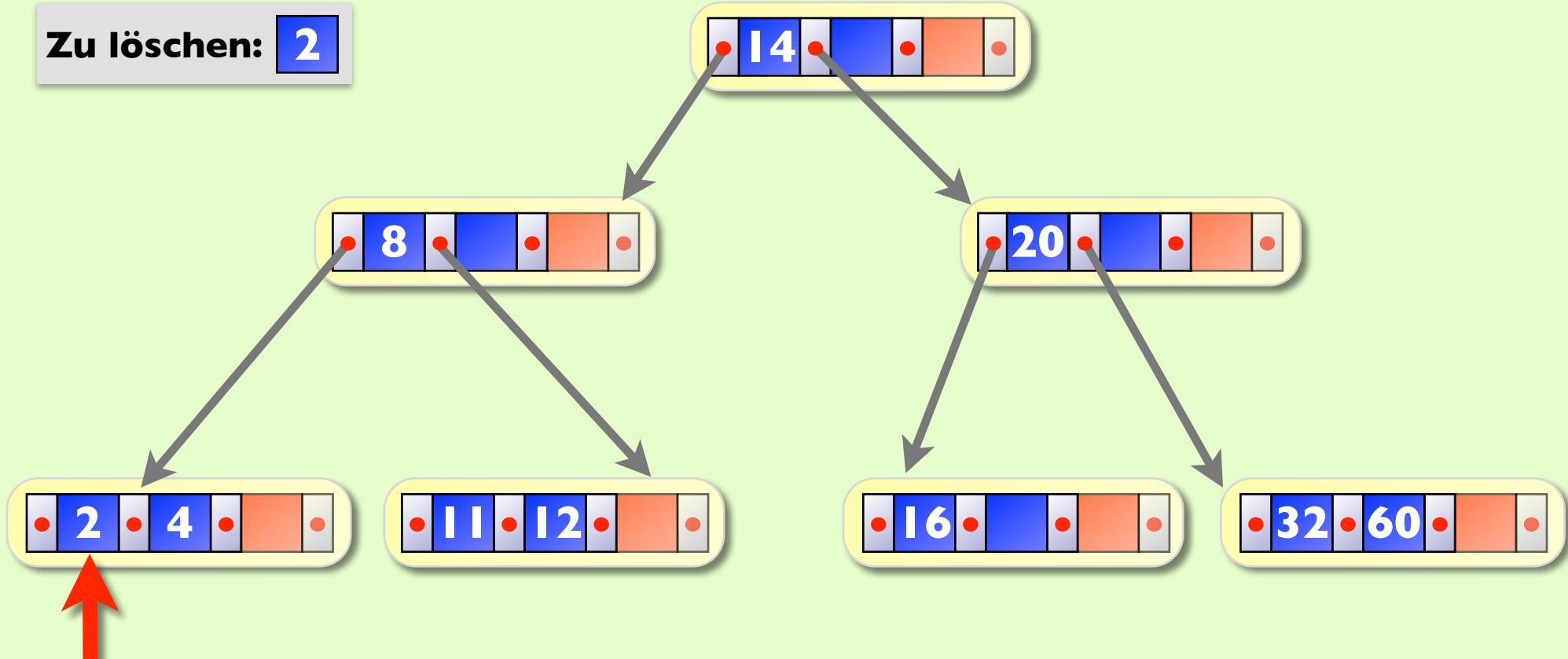
Zu löschen: 2



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

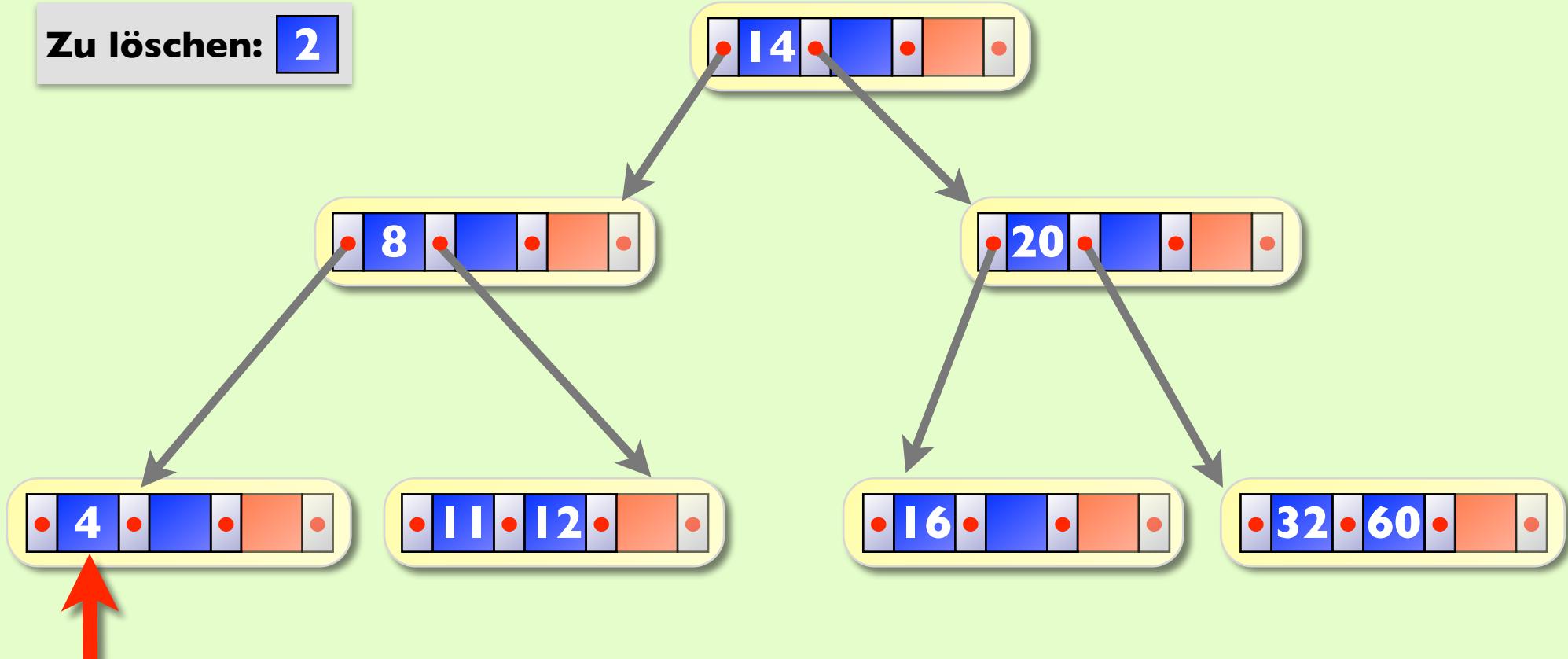
Zu löschen: 2



# Beispiel: Löschen in einem B-Baum

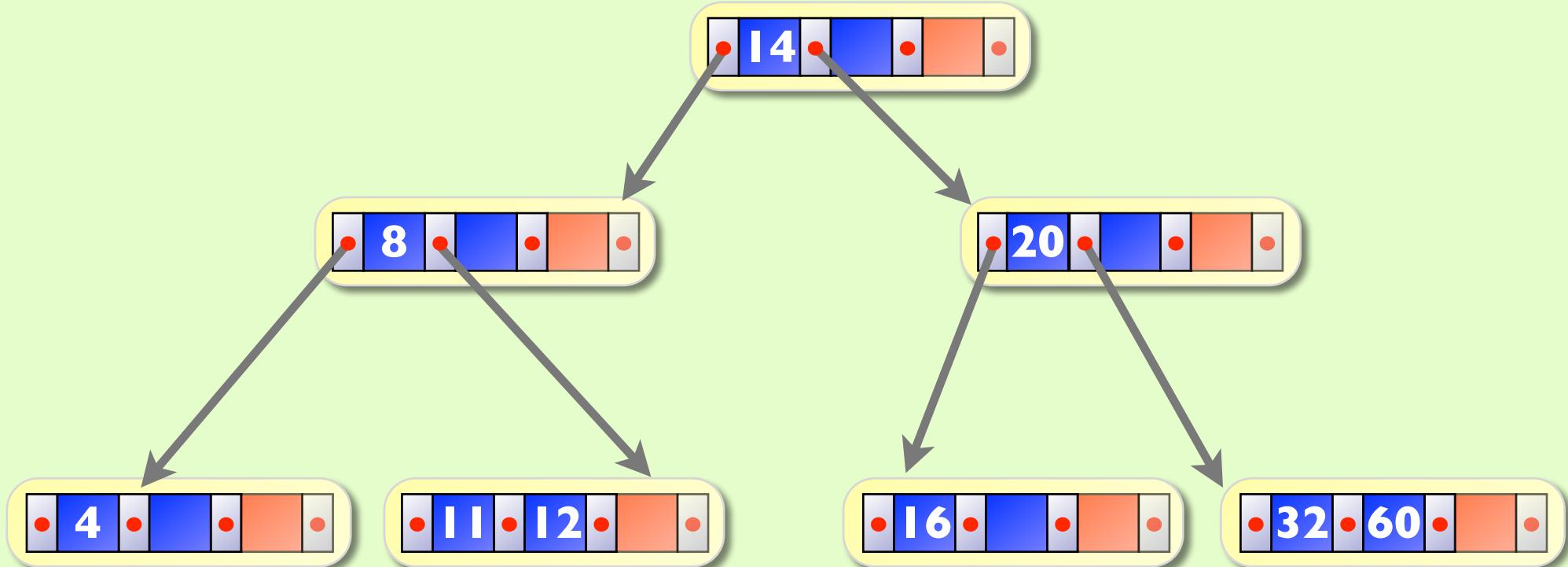
## B B-Baum der Ordnung I - Löschen

Zu löschen: 2



# Beispiel: Löschen in einem B-Baum

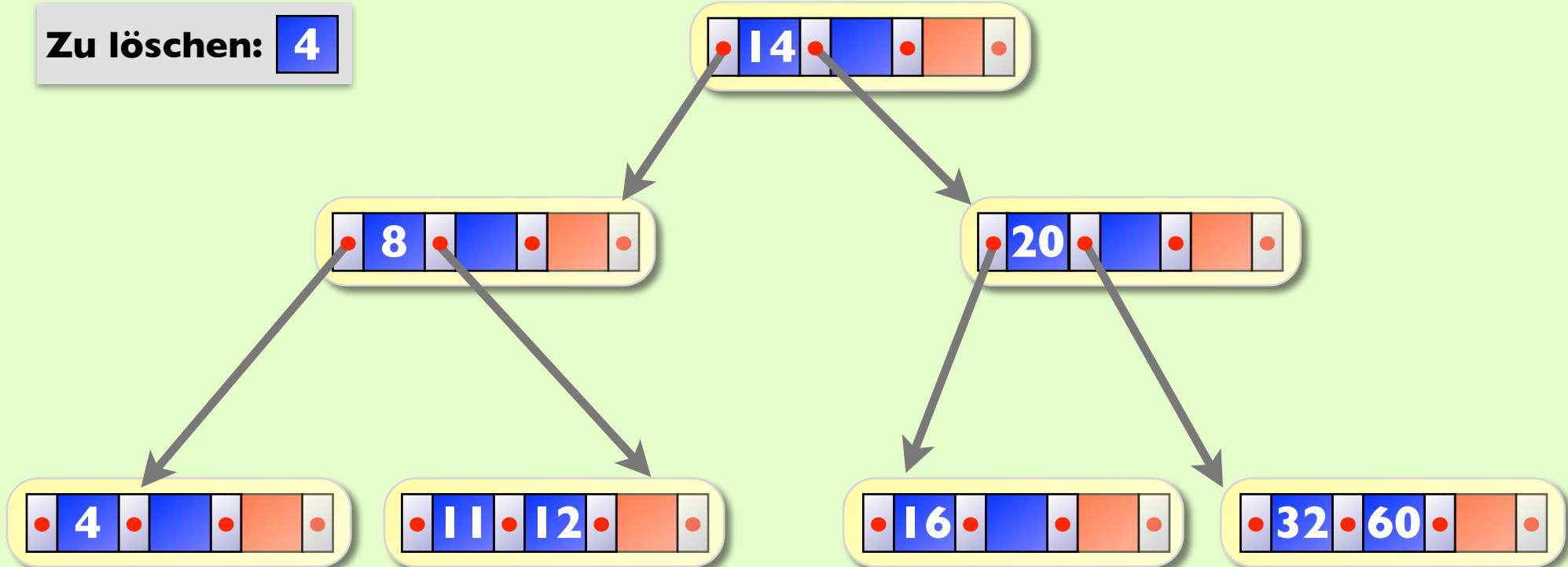
## B B-Baum der Ordnung I - Löschen



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

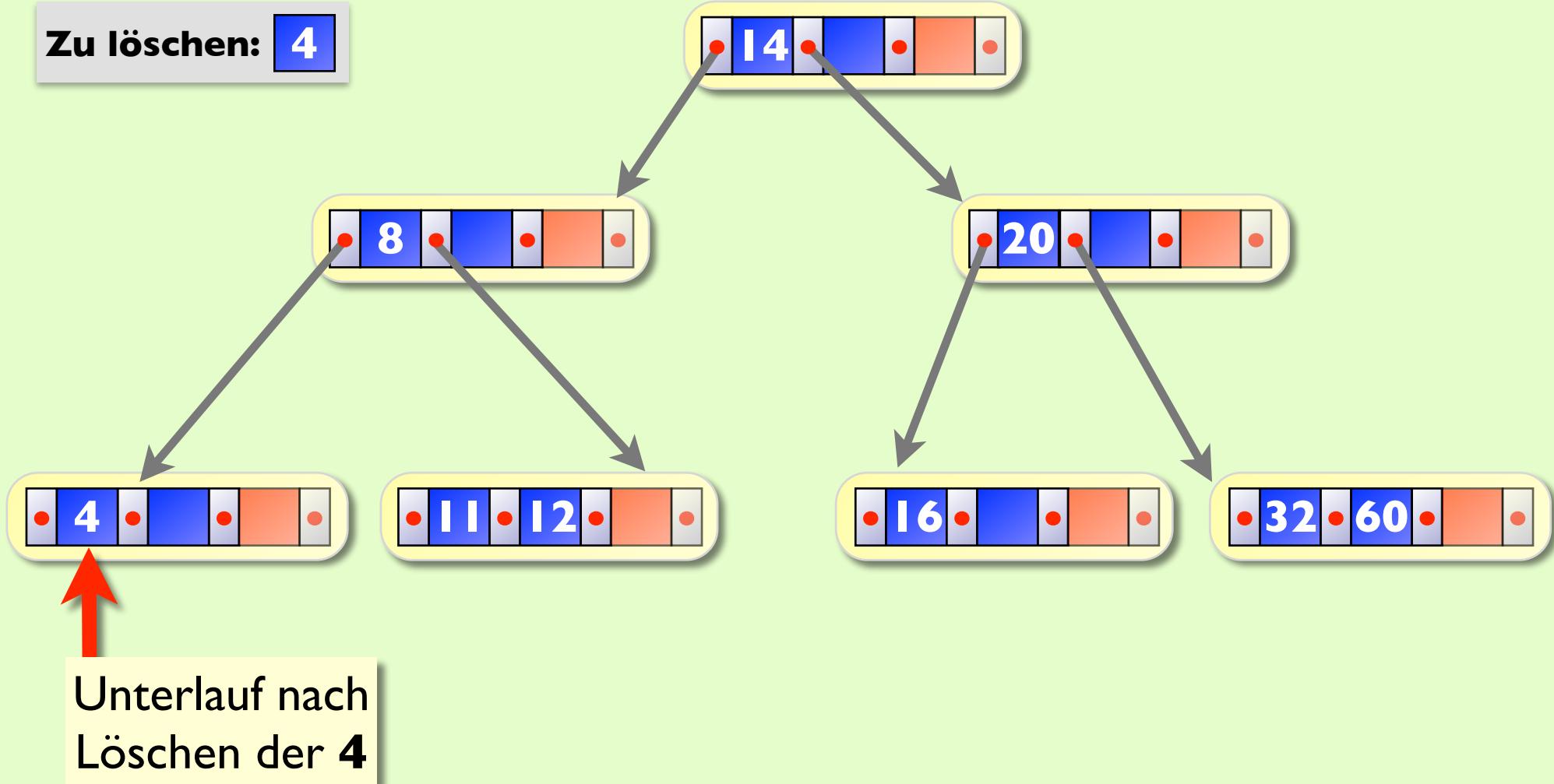
Zu löschen: 4



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

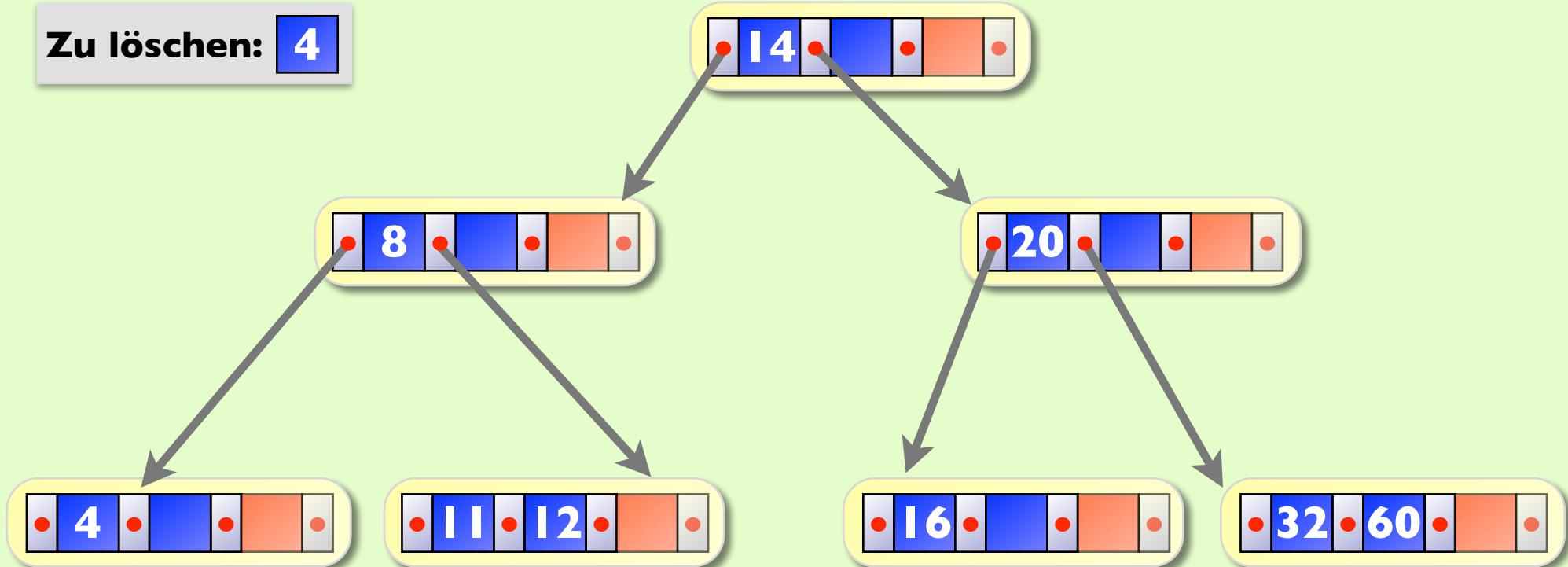
Zu löschen: 4



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

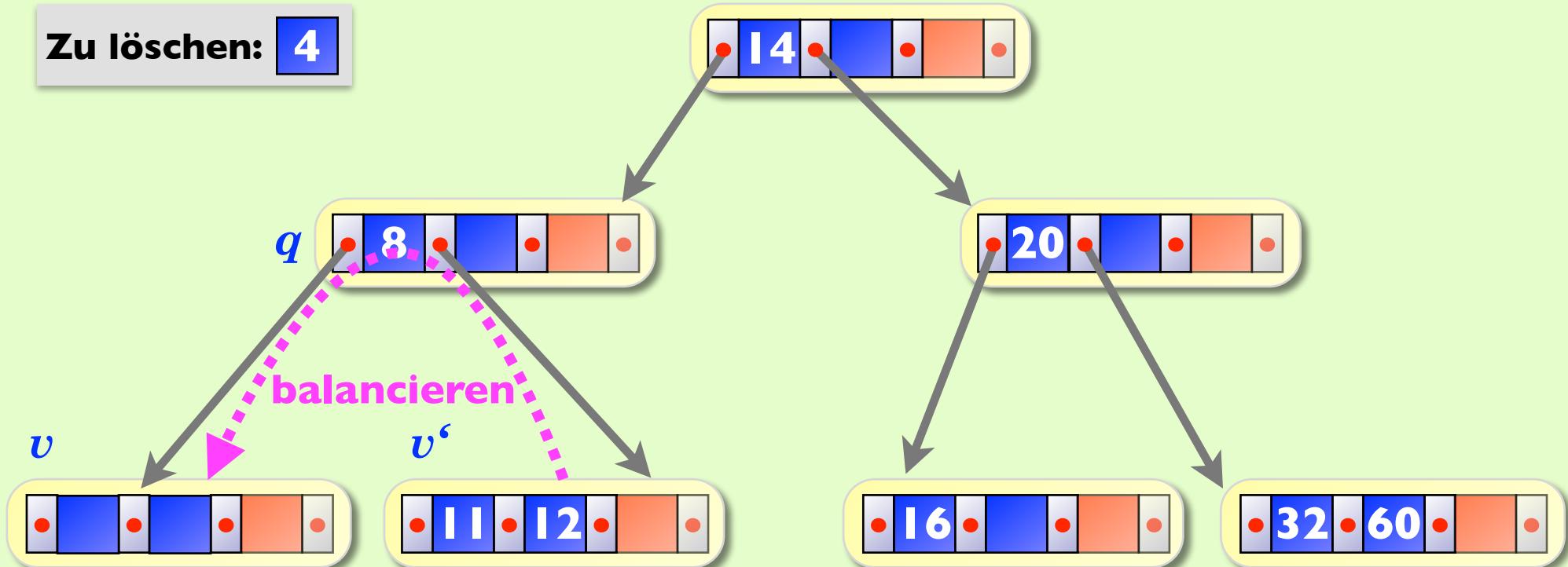
Zu löschen: 4



# Beispiel: Löschen in einem B-Baum

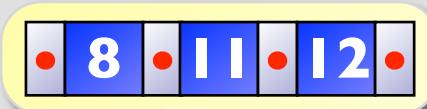
## B B-Baum der Ordnung I - Löschen

Zu löschen: 4



**Balancieren möglich!**

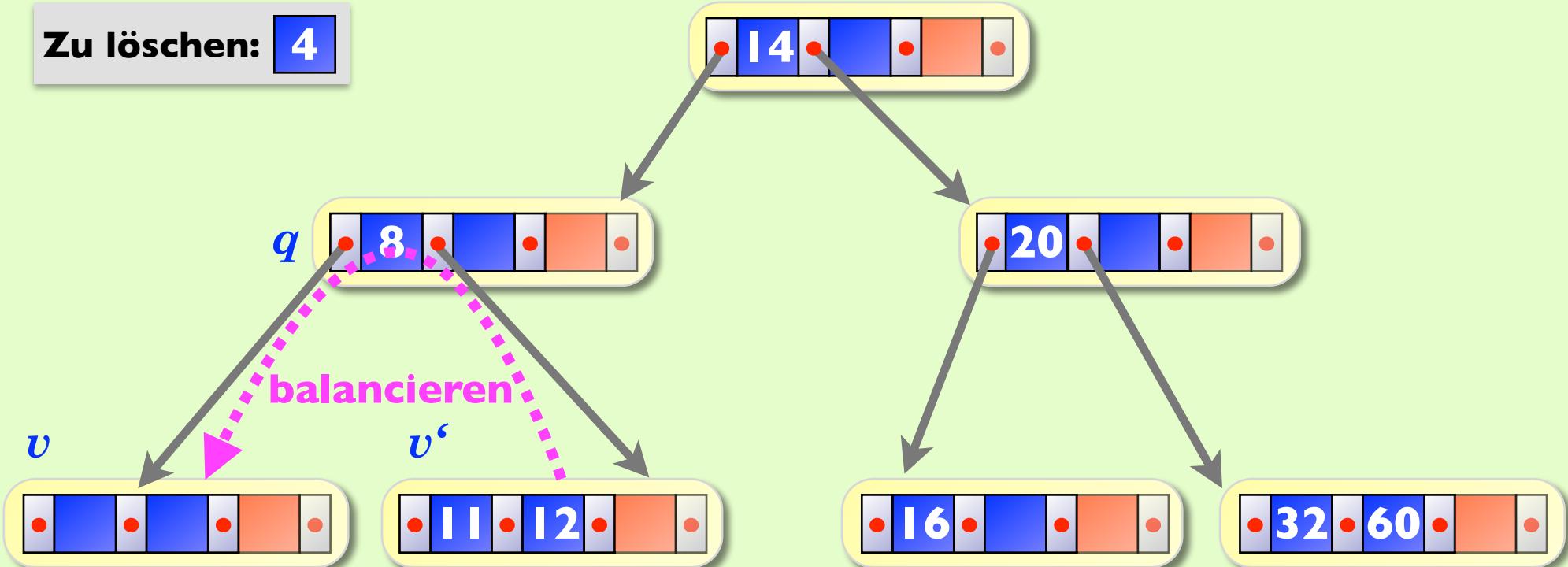
Fiktiver Knoten aus 8 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :



# Beispiel: Löschen in einem B-Baum

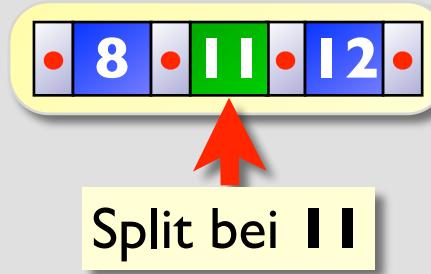
## B B-Baum der Ordnung I - Löschen

Zu löschen: 4



**Balancieren möglich!**

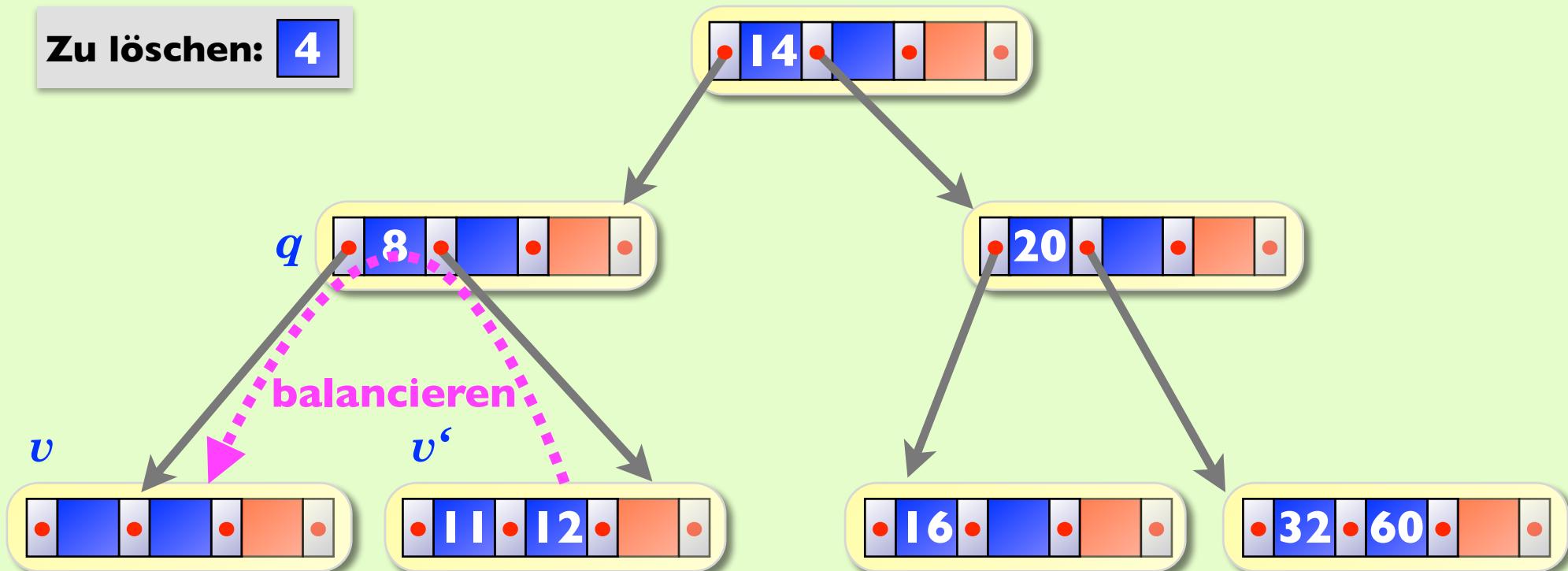
Split bei Median (11):



# Beispiel: Löschen in einem B-Baum

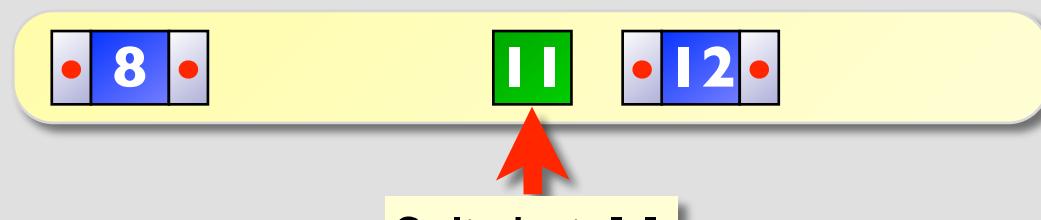
## B B-Baum der Ordnung I - Löschen

Zu löschen: 4



**Balancieren möglich!**

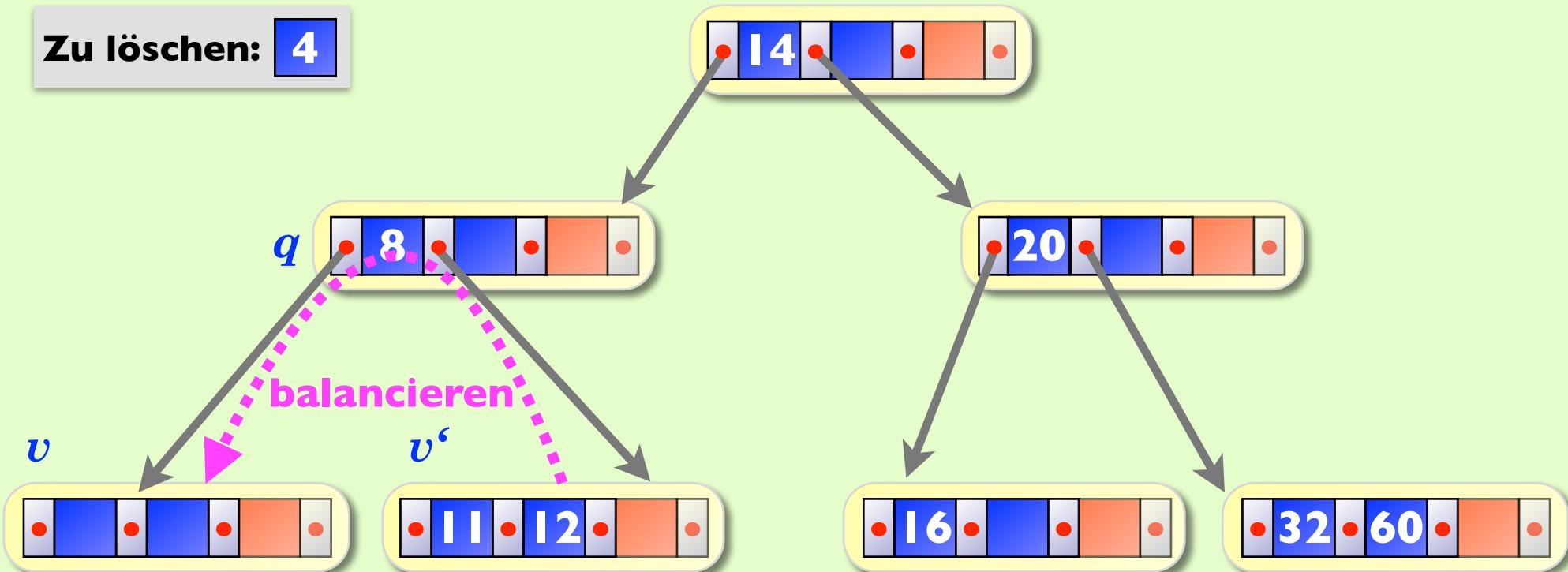
Split bei Median (11):



# Beispiel: Löschen in einem B-Baum

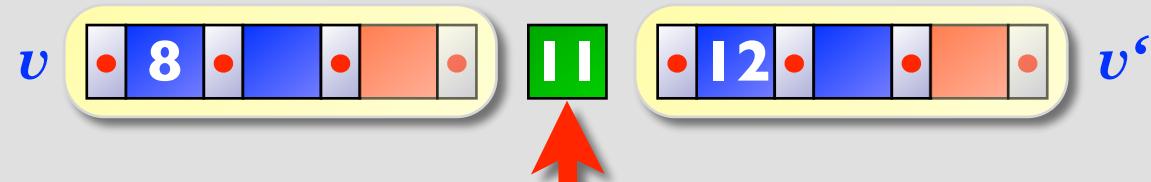
## B B-Baum der Ordnung I - Löschen

Zu löschen: 4



**Balancieren möglich!**

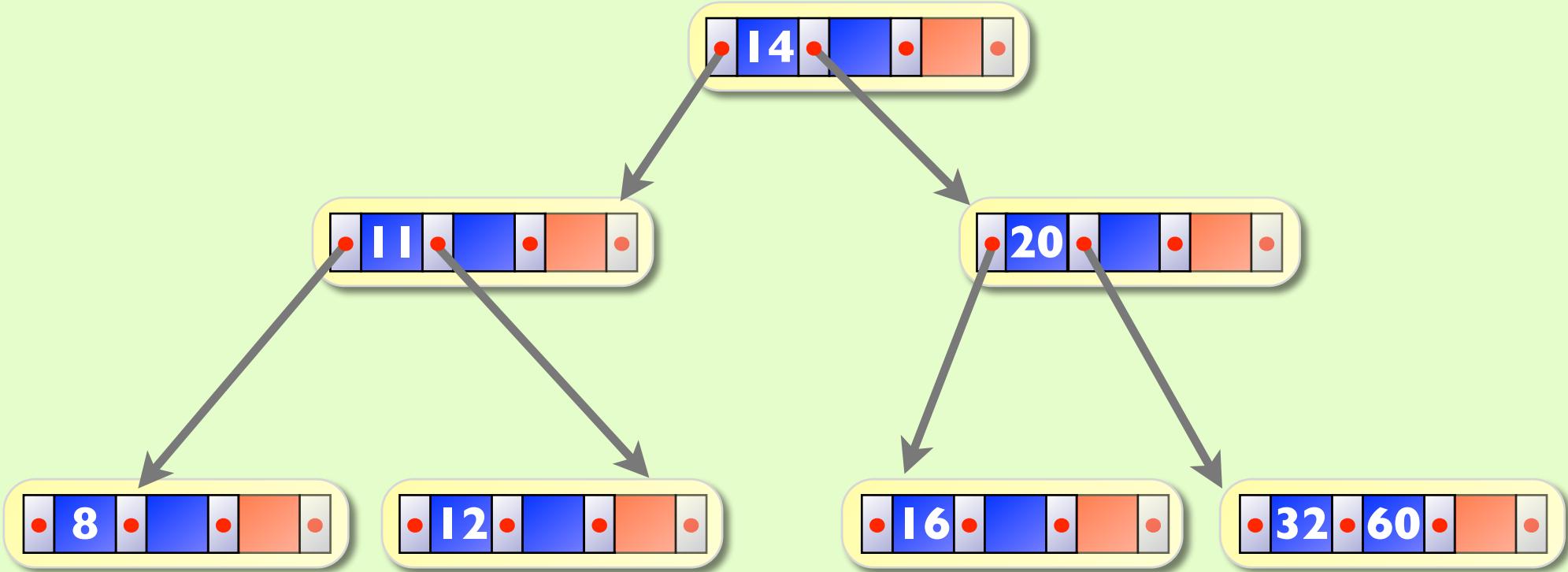
Split bei Median (11):



Wandert in  $q$  (anstelle der 8)

# Beispiel: Löschen in einem B-Baum

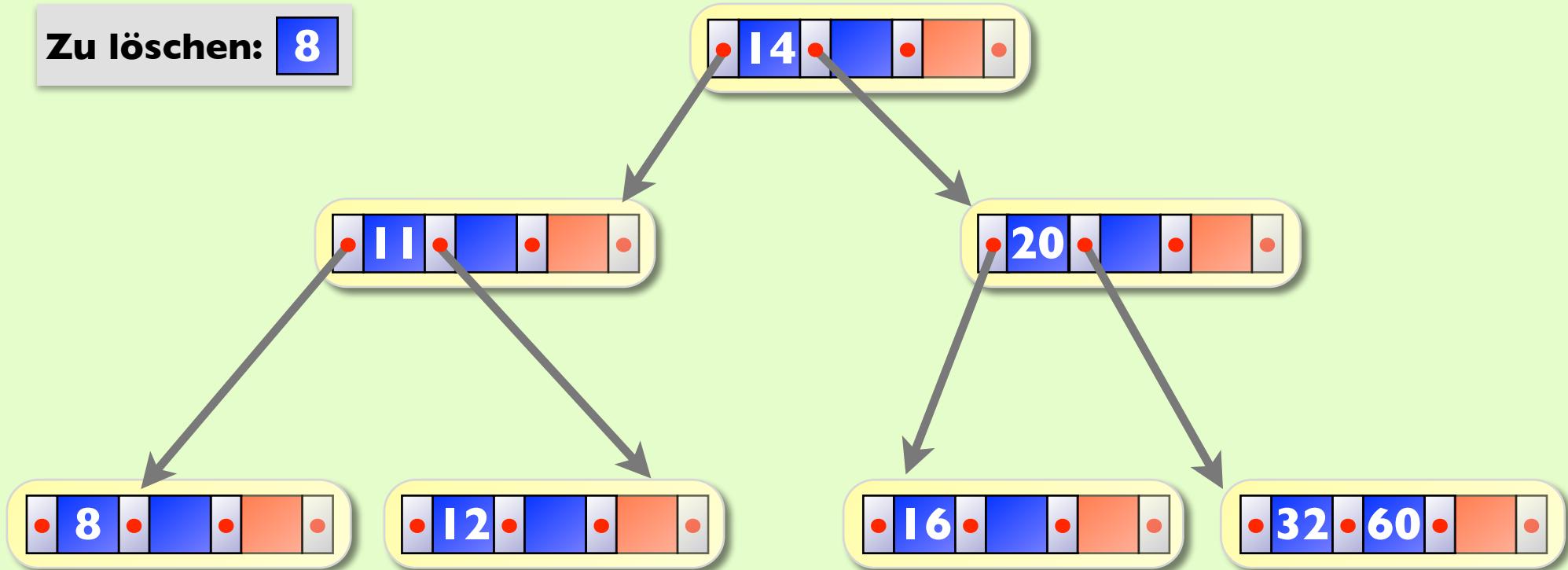
## B B-Baum der Ordnung I - Löschen



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

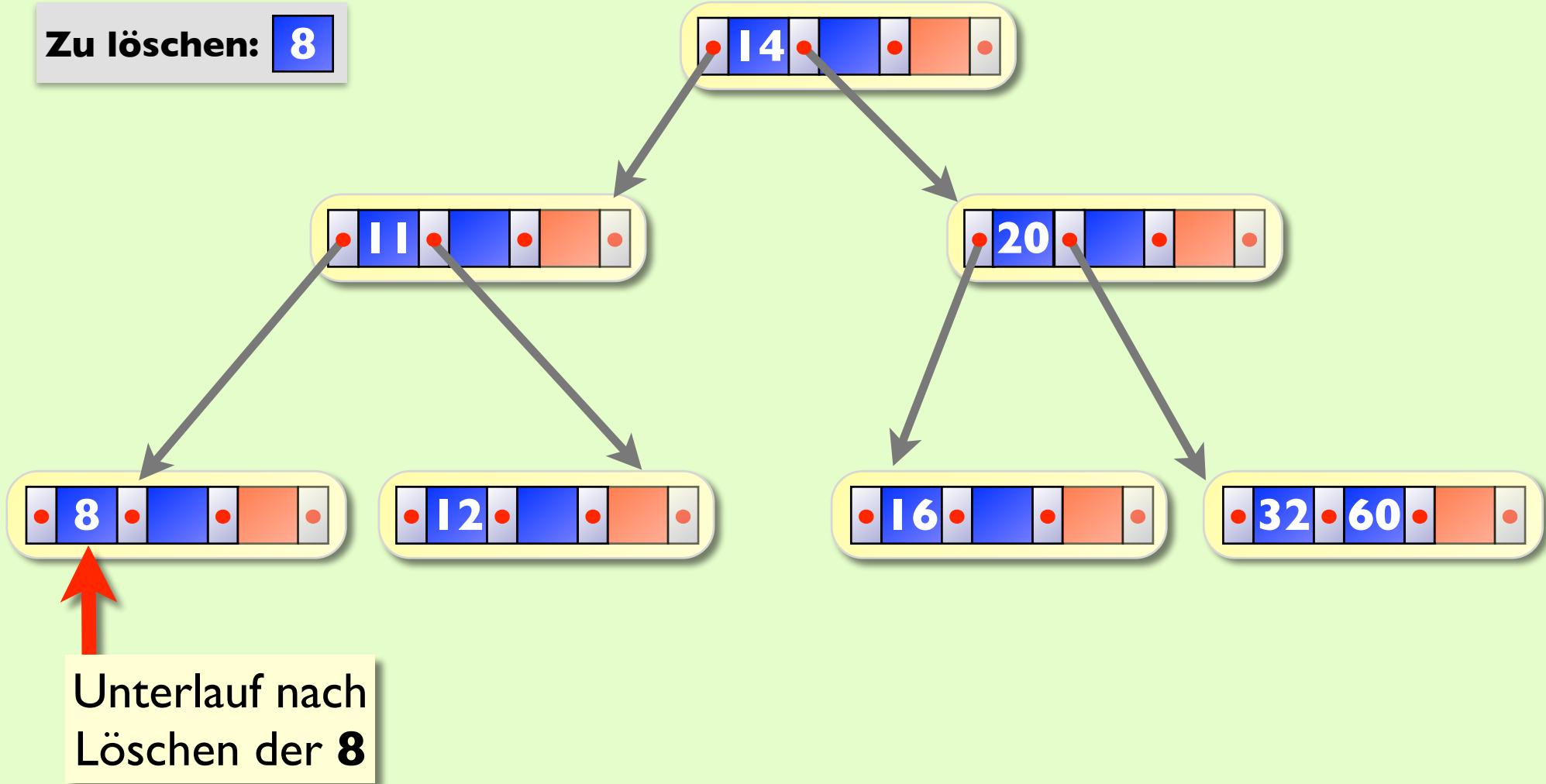
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

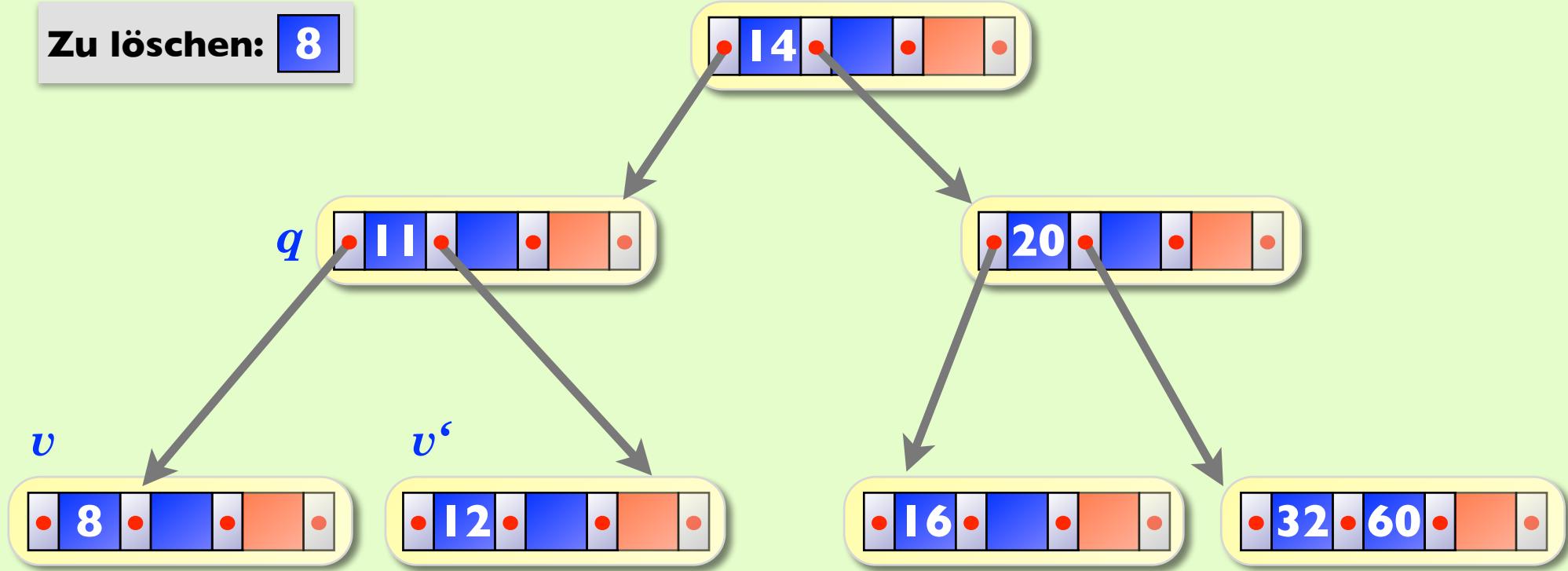
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

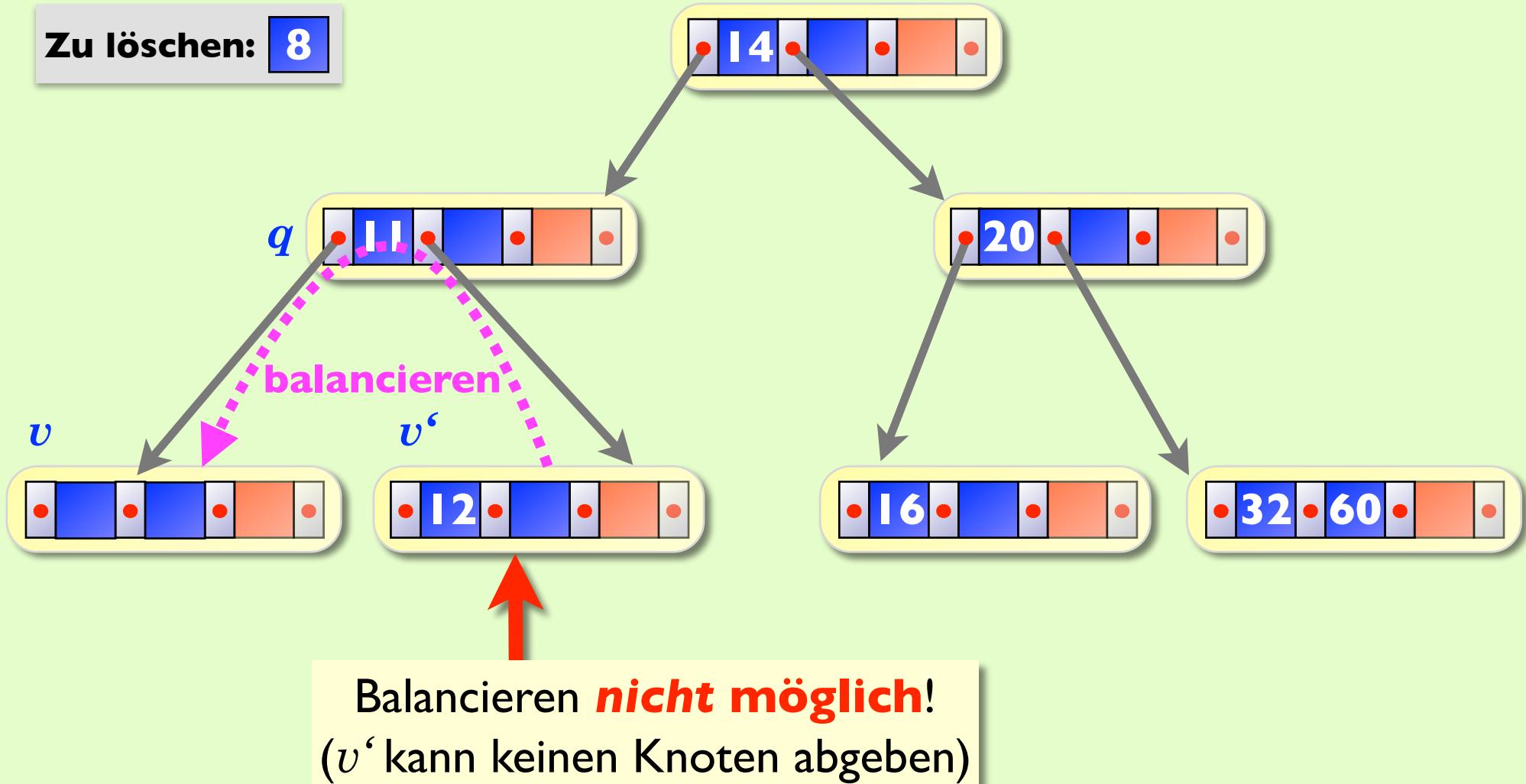
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

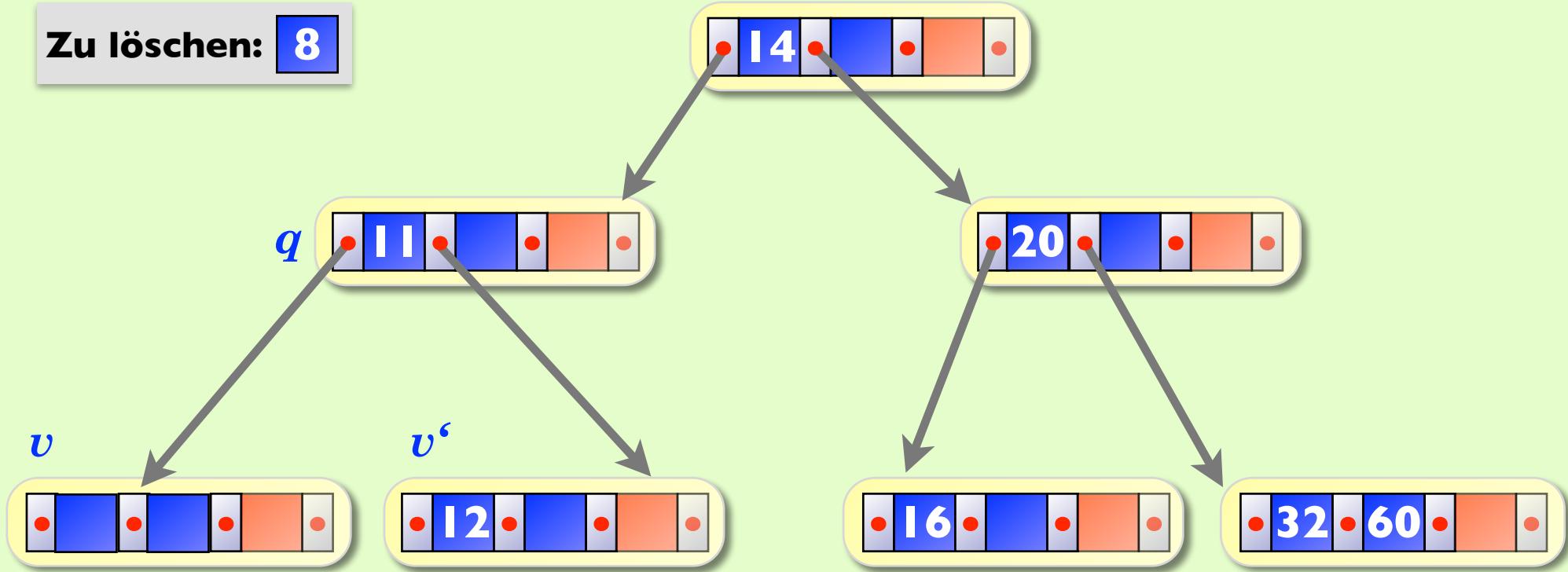
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

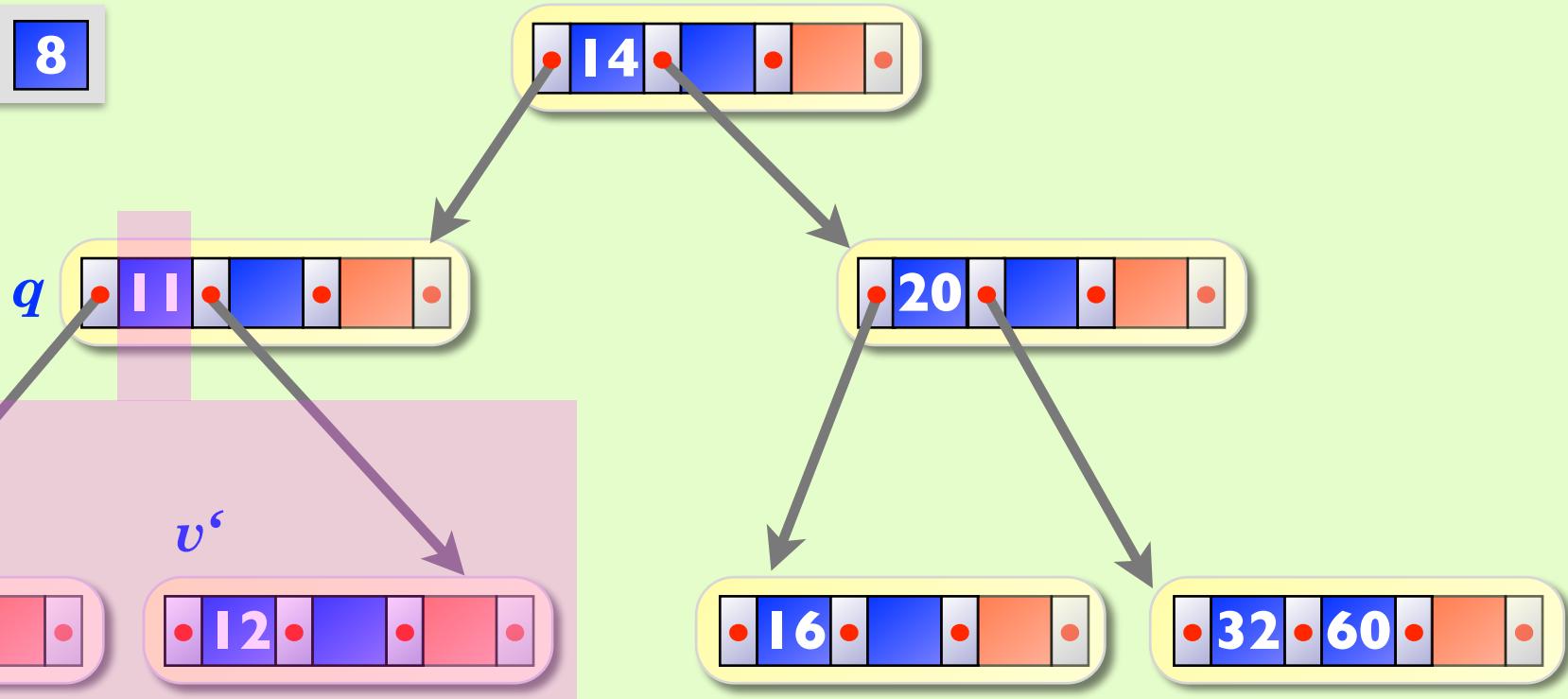
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

Zu löschen: 8



## Mischen

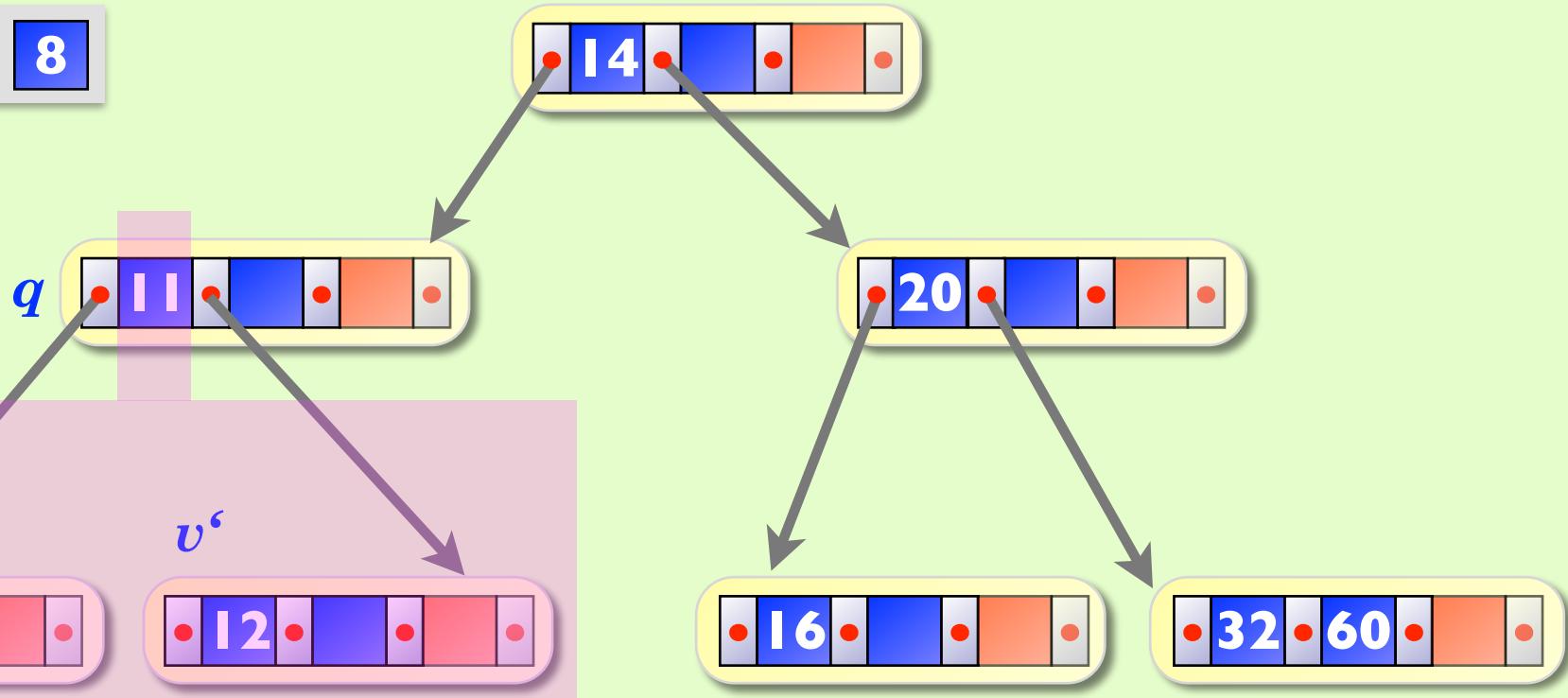
Bilde neuen Knoten aus 11 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

Zu löschen: 8



## Mischen

Bilde neuen Knoten aus 11 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :

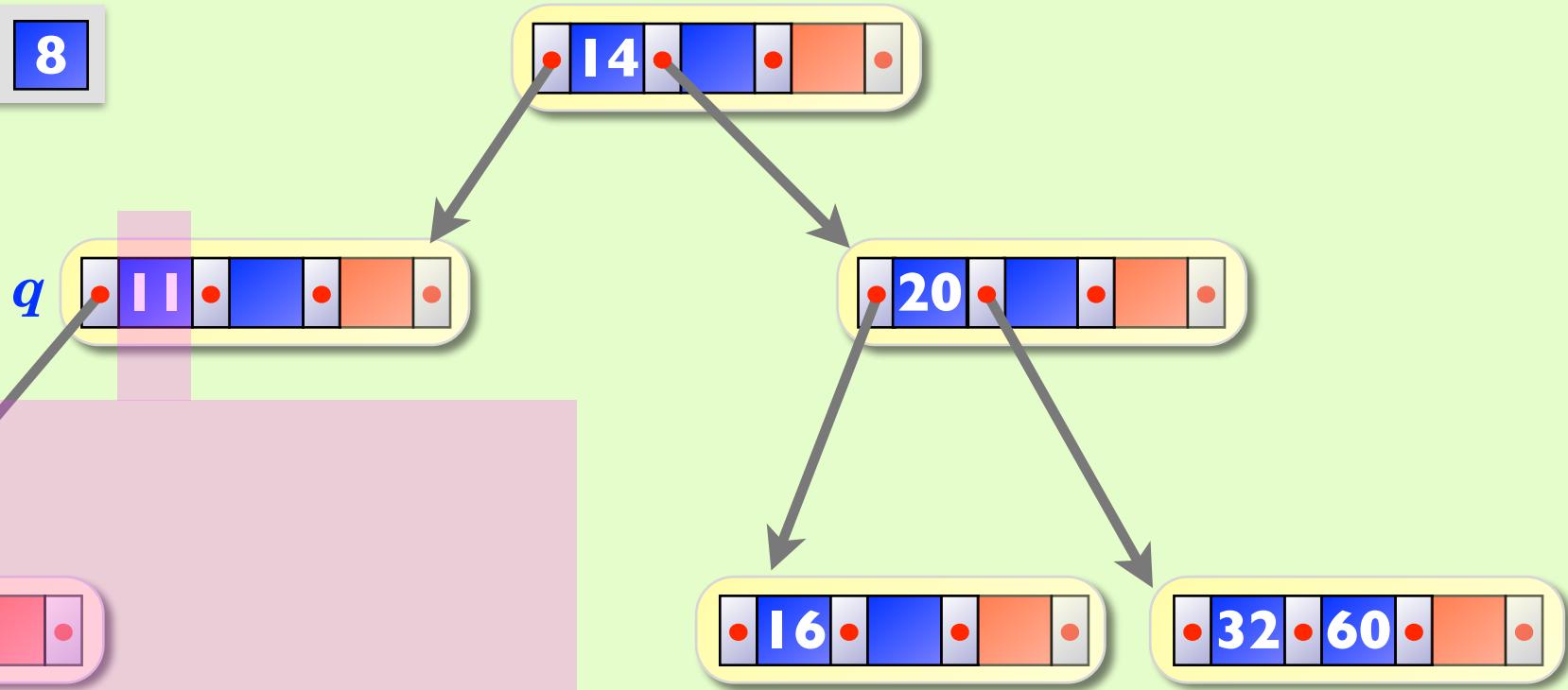


Der neue Knoten rückt an die Stelle von  $v$ ,  $v'$  wird entfernt ...  
... die 11 wird aus  $q$  gelöscht

# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

Zu löschen: 8



## Mischen

Bilde neuen Knoten aus 11 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :

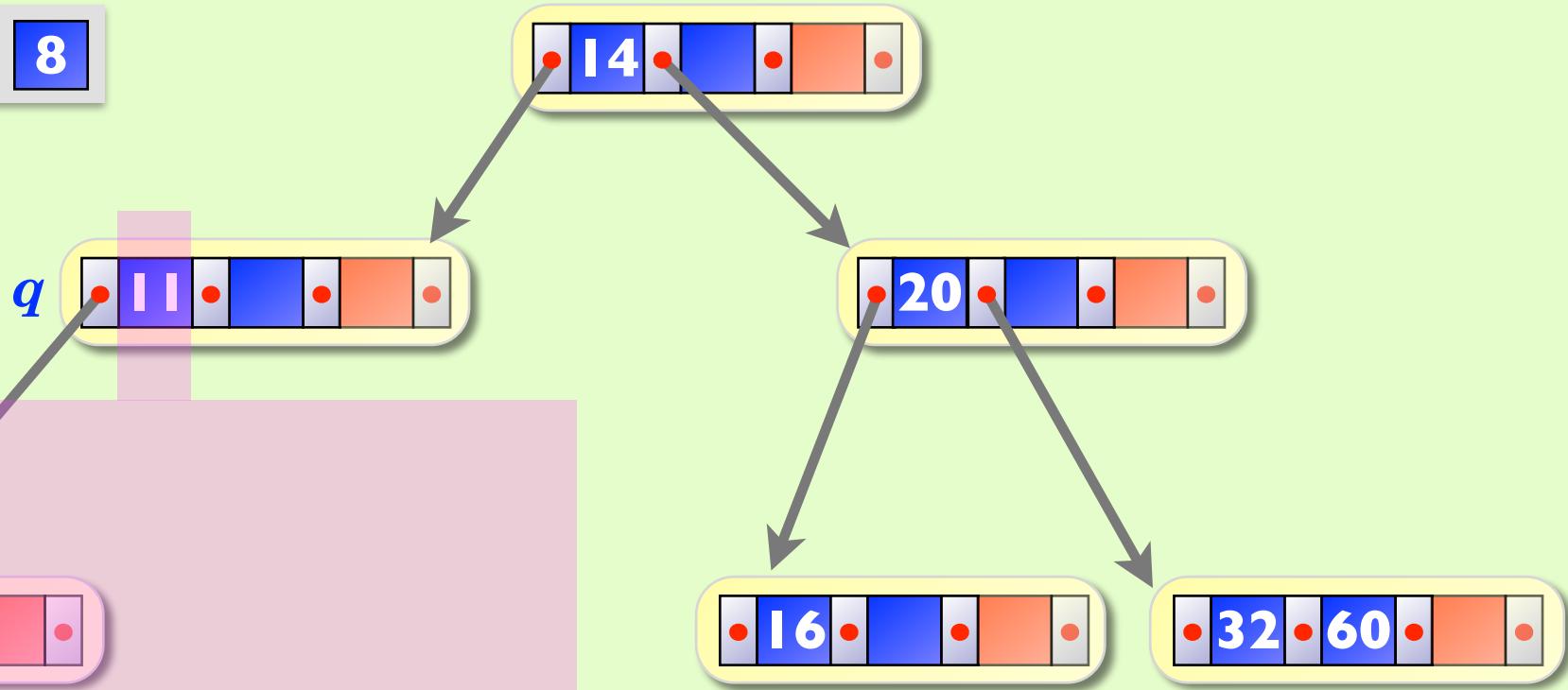


Der neue Knoten rückt an die Stelle von  $v$ ,  $v'$  wird entfernt ...  
... die 11 wird aus  $q$  gelöscht

# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

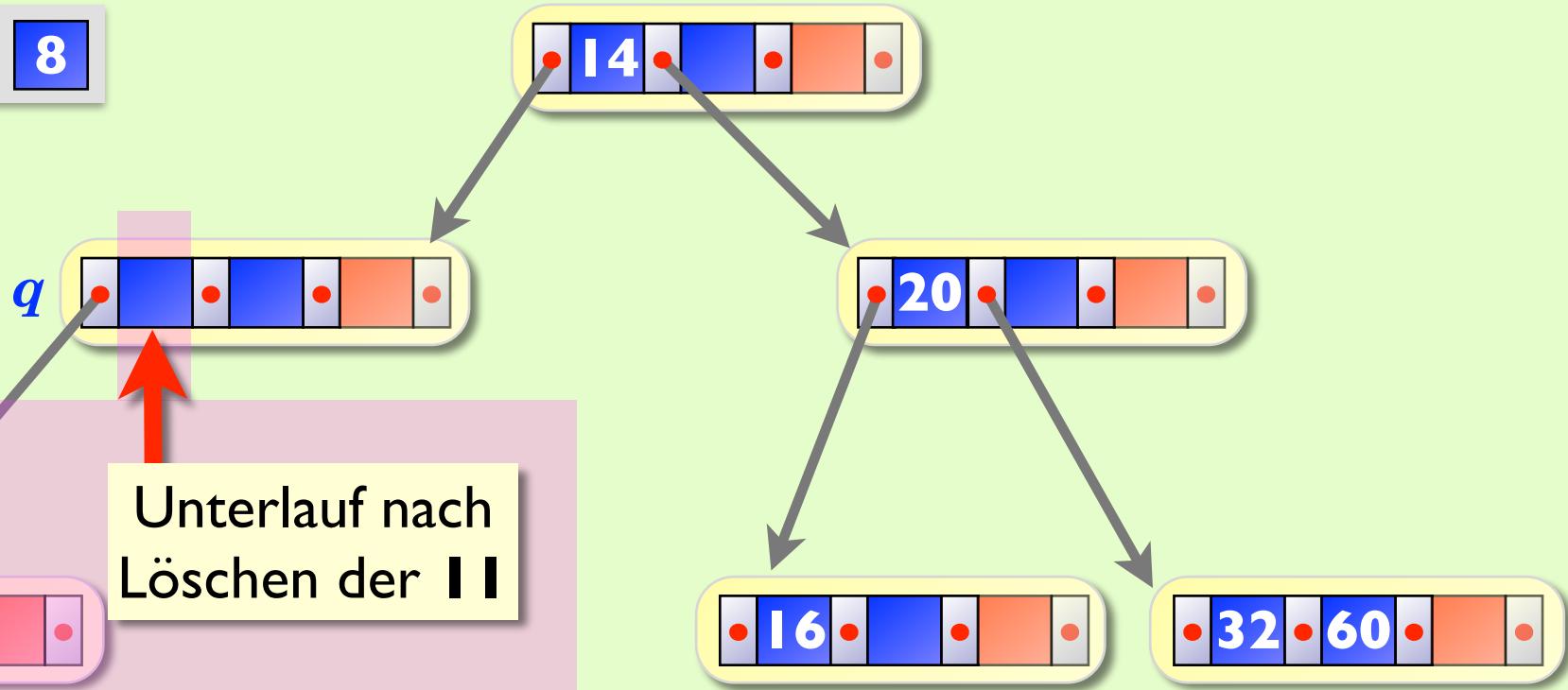
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

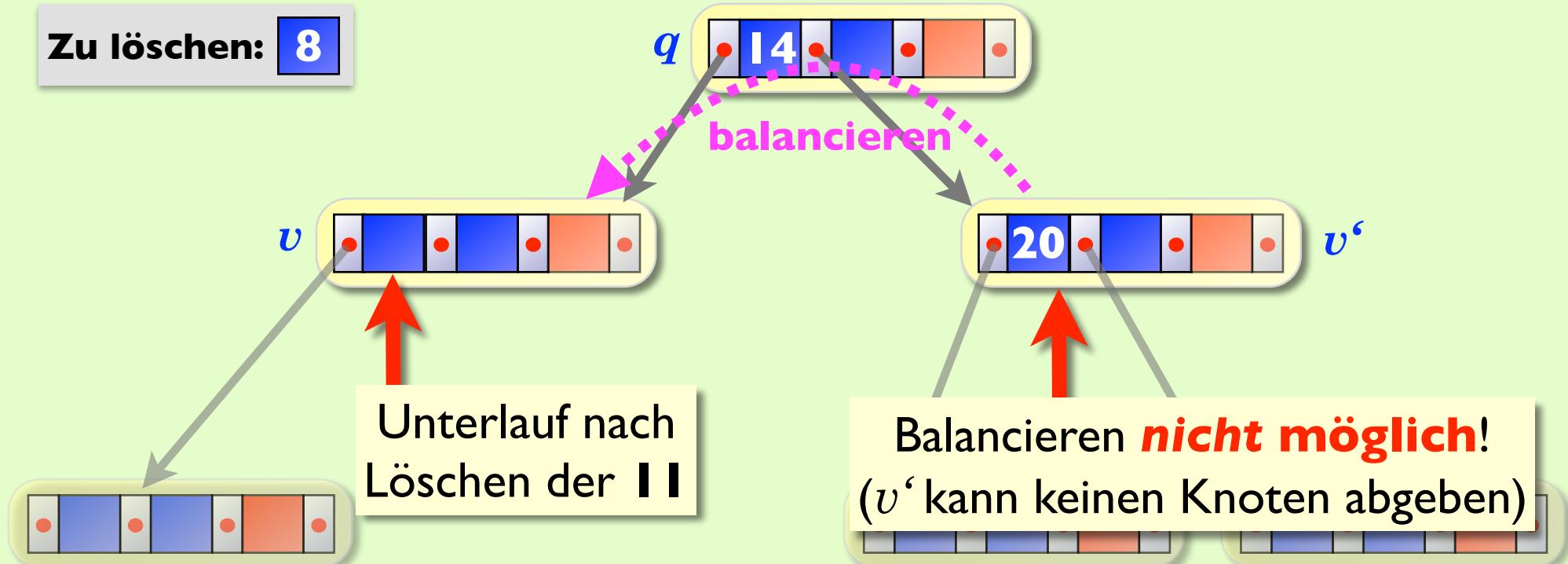
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

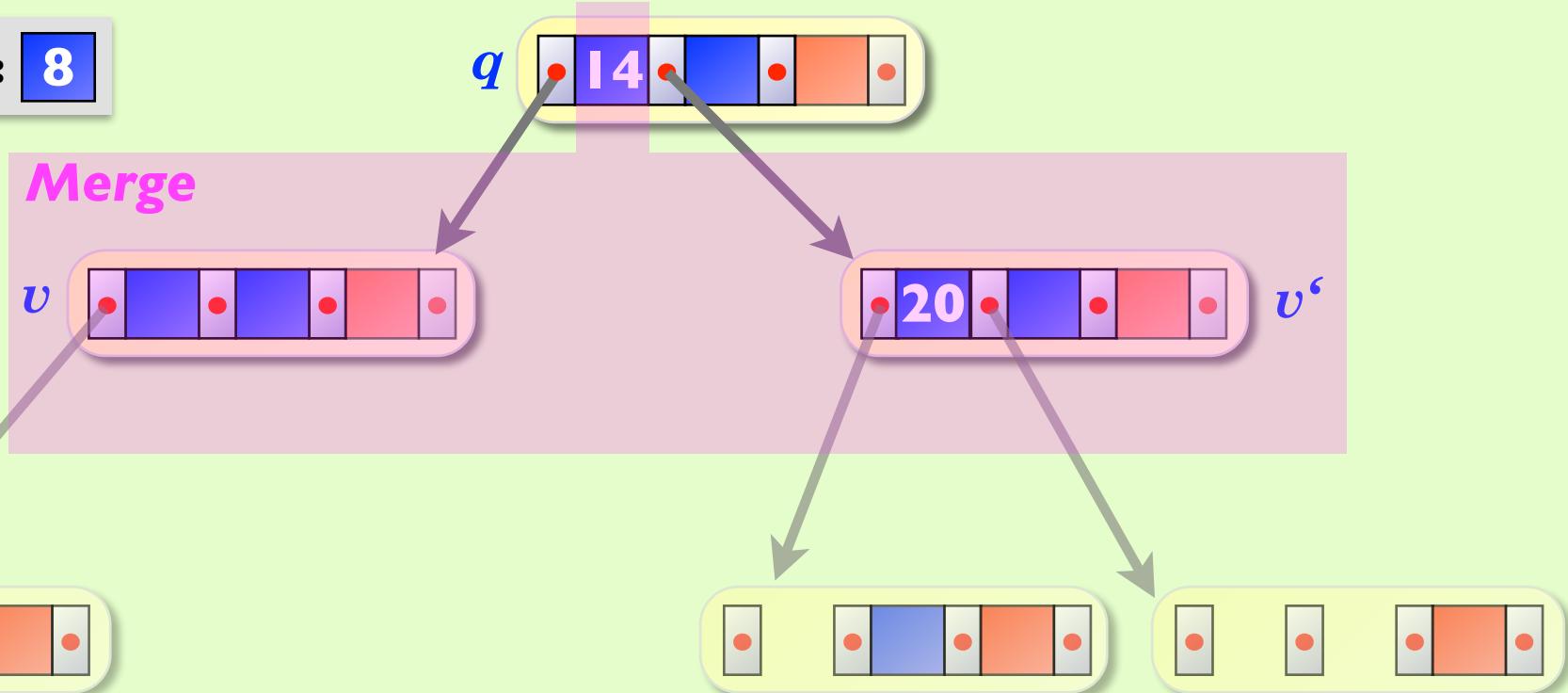
Zu löschen: 8



# Beispiel: Löschen in einem B-Baum

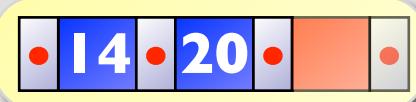
## B B-Baum der Ordnung I - Löschen

Zu löschen: 8



## Mischen

Bilde neuen Knoten aus 14 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :

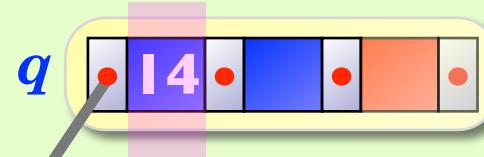


Der neue Knoten rückt an die Stelle von  $v$ ,  $v'$  wird entfernt ...  
... die 14 wird aus  $q$  gelöscht

# Beispiel: Löschen in einem B-Baum

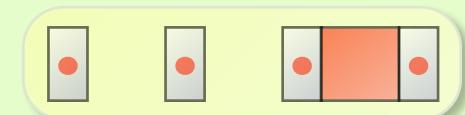
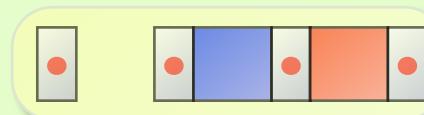
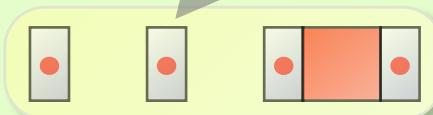
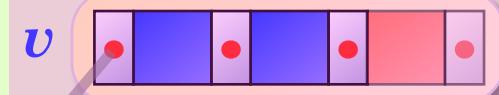
## B B-Baum der Ordnung I - Löschen

Zu löschen: 8



Merge

$v$



## Mischen

Bilde neuen Knoten aus 14 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :

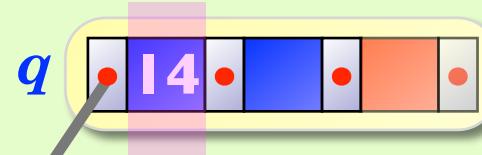


Der neue Knoten rückt an die Stelle von  $v$ ,  $v'$  wird entfernt ...  
... die **14** wird aus  $q$  gelöscht

# Beispiel: Löschen in einem B-Baum

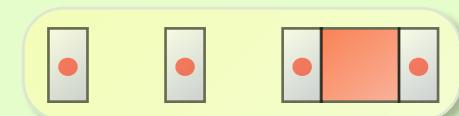
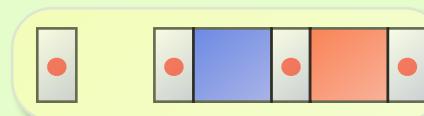
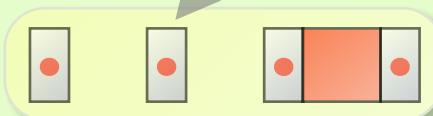
## B B-Baum der Ordnung I - Löschen

Zu löschen: 8



Merge

$v$



## Mischen

Bilde neuen Knoten aus 14 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :

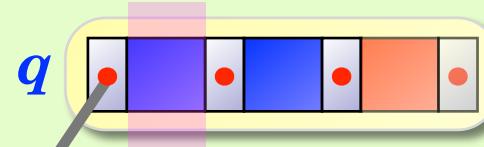


Der neue Knoten rückt an die Stelle von  $v$ ,  $v'$  wird entfernt ...  
... die 14 wird aus  $q$  gelöscht

# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen

Zu löschen: 8



Merge

$v$



## Mischen

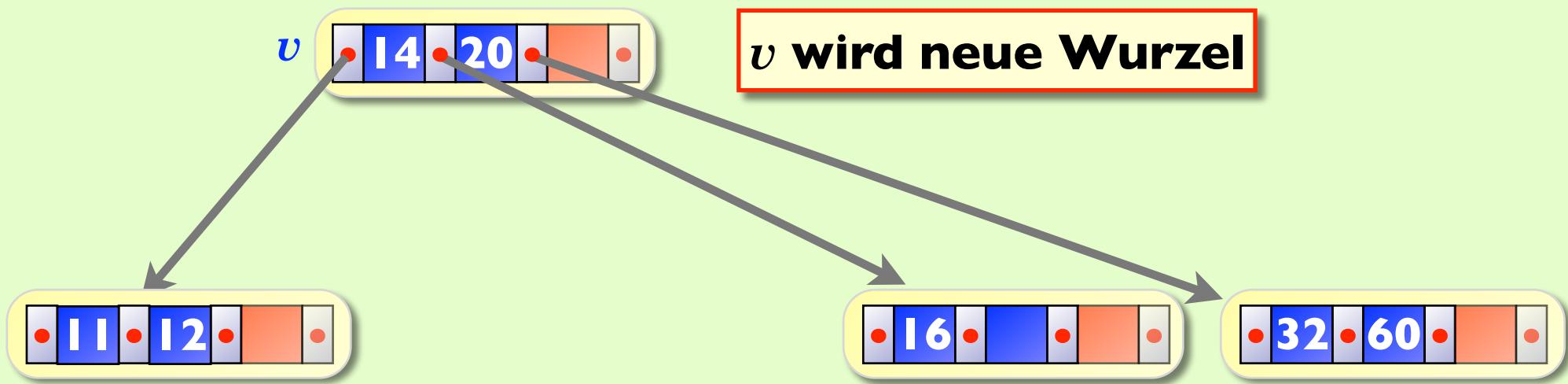
Bilde neuen Knoten aus 14 (dem relevanten Schlüssel aus  $q$ ),  $v$  und seinem Bruder  $v'$ :



Der neue Knoten rückt an die Stelle von  $v$ ,  $v'$  wird entfernt ...  
... die 14 wird aus  $q$  gelöscht

# Beispiel: Löschen in einem B-Baum

## B B-Baum der Ordnung I - Löschen



# V. Suchen in Mengen

## 5. Suchen in Mengen

- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume**
  - 5.5.1. Gewichtsbalanceierte Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume**
- 5.6. *Priority Queue* und *Heap*

# Motivation

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

# Motivation

- Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum mit folgenden Eigenschaften:
  1. Jeder Knoten ist **rot** oder **schwarz**

# Motivation

- Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum mit folgenden Eigenschaften:
  1. Jeder Knoten ist **rot** oder **schwarz**
  2. Die Wurzel ist **schwarz**

- Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum mit folgenden Eigenschaften:
  1. Jeder Knoten ist **rot** oder **schwarz**
  2. Die Wurzel ist **schwarz**
  3. Jedes **Bereichsblatt** ist **schwarz**

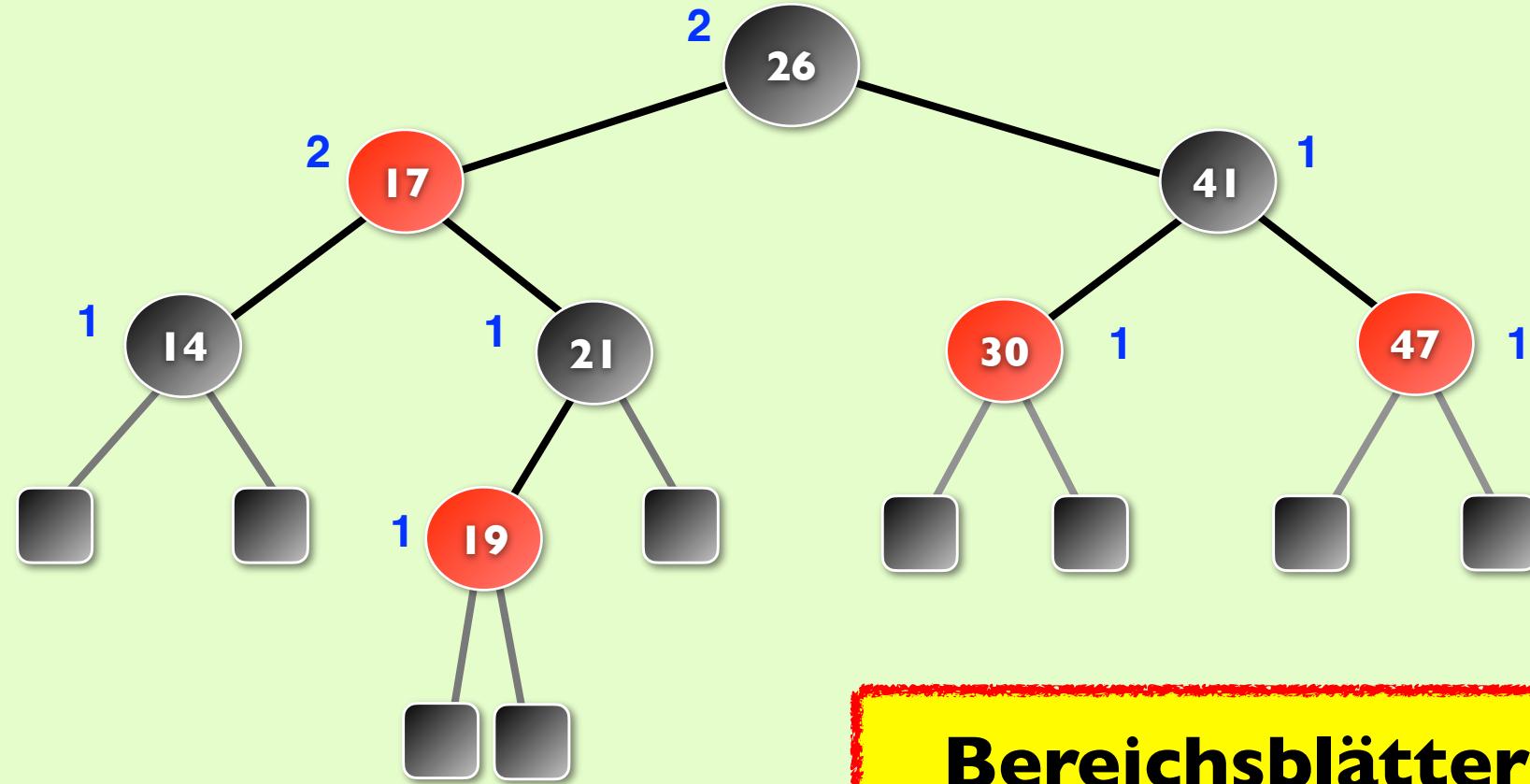
- Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum mit folgenden Eigenschaften:
  1. Jeder Knoten ist **rot** oder **schwarz**
  2. Die Wurzel ist **schwarz**
  3. Jedes **Bereichsblatt** ist **schwarz**
  4. Ist ein Knoten **rot**, so sind seine beiden Kinder **schwarz**  
**NRR - nicht-rot-rot**

- Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum mit folgenden Eigenschaften:
  1. Jeder Knoten ist **rot** oder **schwarz**
  2. Die Wurzel ist **schwarz**
  3. Jedes **Bereichsblatt** ist **schwarz**
  4. Ist ein Knoten **rot**, so sind seine beiden Kinder **schwarz**  
**NRR - nicht-rot-rot**
  5. Für jeden Knoten gilt: Jeder Pfad zu einem Blatt enthält dieselbe Anzahl an **schwarzen** Knoten  
**GAZ - gleiche-Anzahl-Schwarz**

- Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum mit folgenden Eigenschaften:
  1. Jeder Knoten ist **rot** oder **schwarz**
  2. Die Wurzel ist **schwarz**
  3. Jedes **Bereichsblatt** ist **schwarz**
  4. Ist ein Knoten **rot**, so sind seine beiden Kinder **schwarz**  
**NRR - nicht-rot-rot**
  5. Für jeden Knoten gilt: Jeder Pfad zu einem Blatt enthält dieselbe Anzahl an **schwarzen** Knoten  
**GAZ - gleiche-Anzahl-Schwarz**
- Vorteil gegenüber AVL-Baum
  - nur 1 Bit zusätzlicher Speicher erforderlich (AVL-Baum: 3 Bit)

# Beispiel: Rot-Schwarz-Baum

## B Rot-Schwarz-Baum

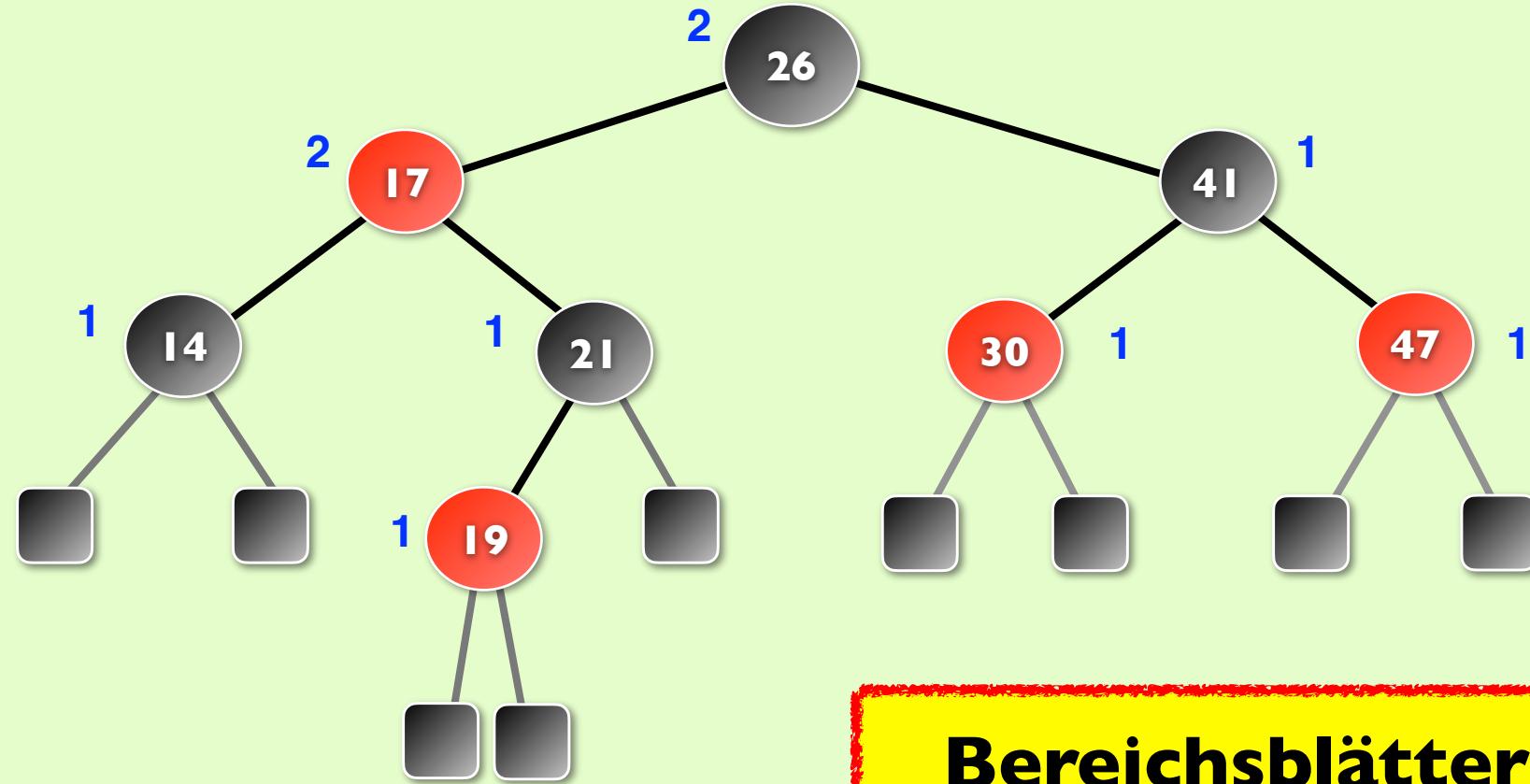


Anzahl schwarzer Knoten bis Blatt

**Bereichsblätter!**

# Beispiel: Rot-Schwarz-Baum

## B Rot-Schwarz-Baum

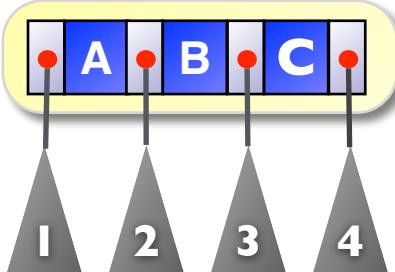


**Bereichsblätter!**

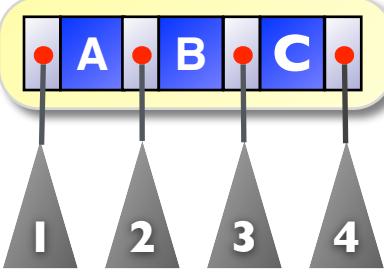
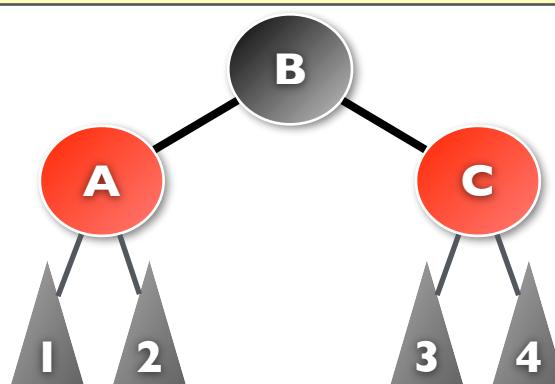
Anzahl schwarzer Knoten bis Blatt

**Rot-Schwarz-Baum ist  
isomorph zu (2,4)-Baum!**

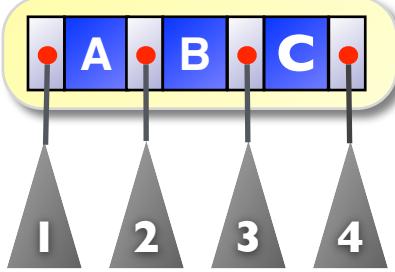
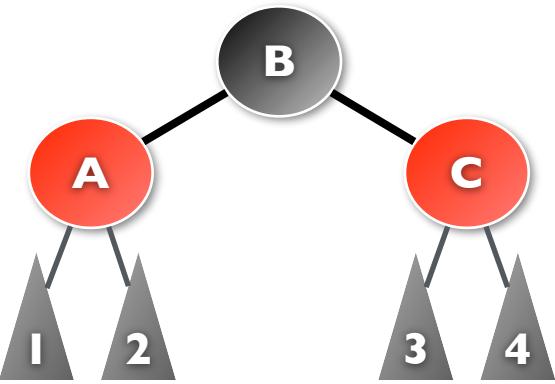
# Korrespondenz: Rot-Schwarz- und (2,4)-Baum

(2,4)-Baum	Rot.Schwarz-Baum
 A diagram of a (2,4)-tree node. It consists of a horizontal row of four blue rectangular boxes labeled A, B, C from left to right. Each box contains a red dot at its top center. Above the first box is a small grey rectangle. Below each box is a dark grey triangle pointing downwards, labeled 1, 2, 3, and 4 from left to right, representing the four children of the node.	

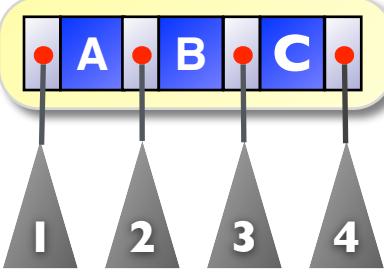
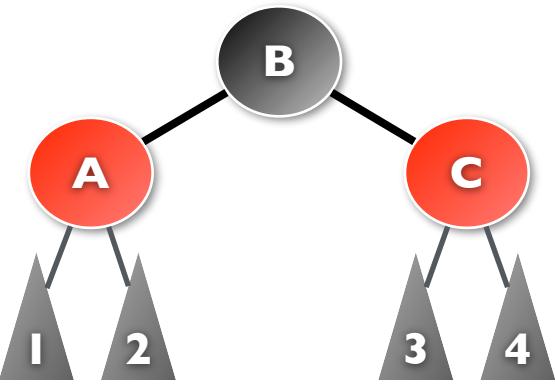
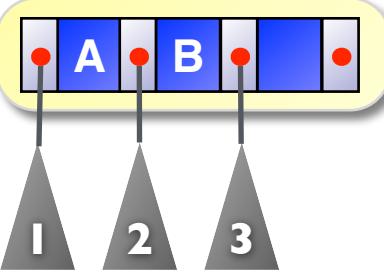
# Korrespondenz: Rot-Schwarz- und (2,4)-Baum

(2,4)-Baum	Rot.Schwarz-Baum
	

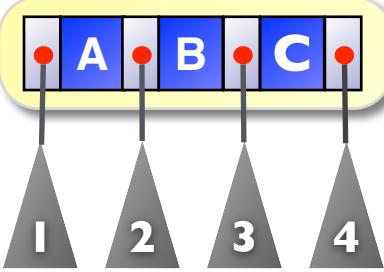
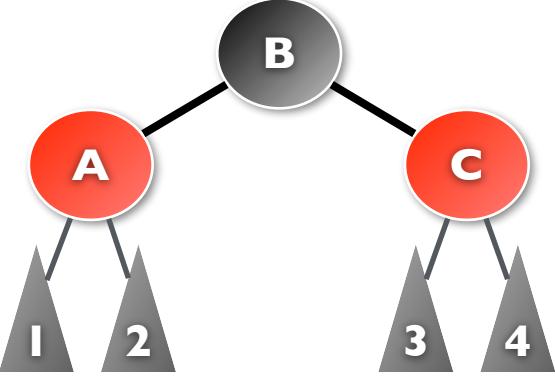
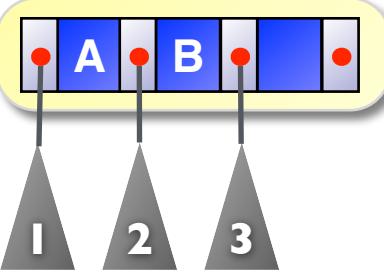
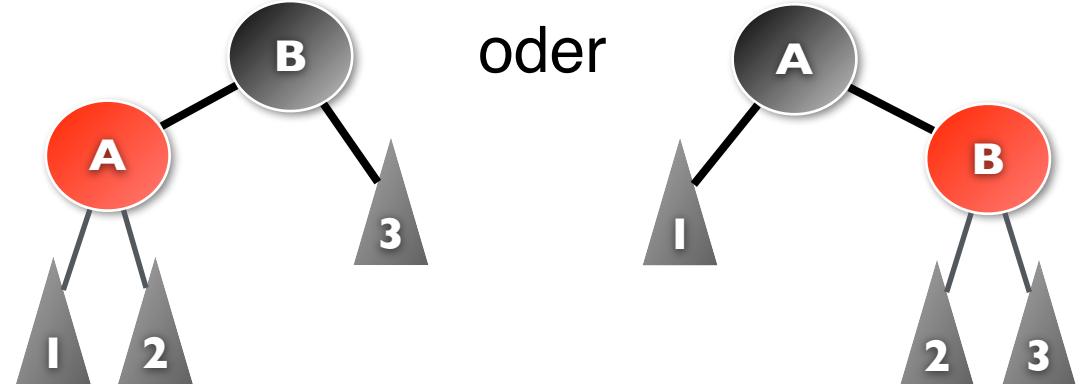
# Korrespondenz: Rot-Schwarz- und (2,4)-Baum

(2,4)-Baum	Rot.Schwarz-Baum
	 <p>Schwarzer Knoten wird mit seinen roten Kindern „verschmolzen“</p>

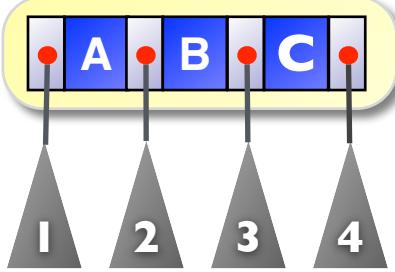
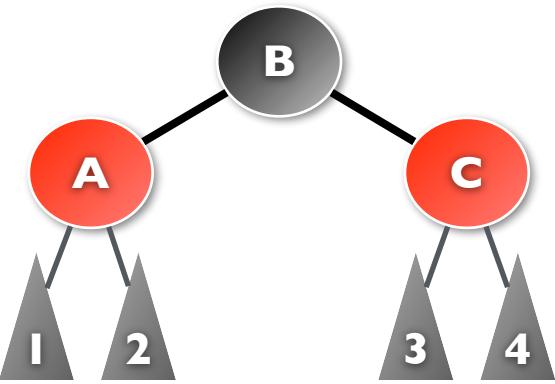
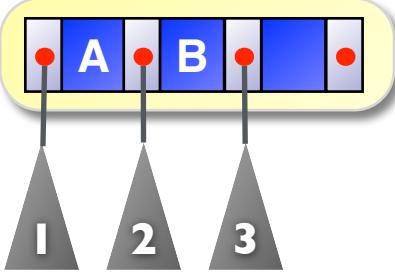
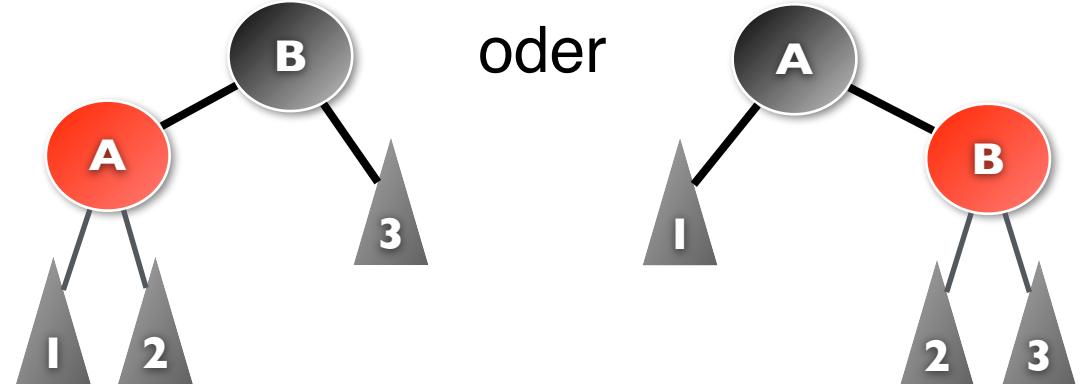
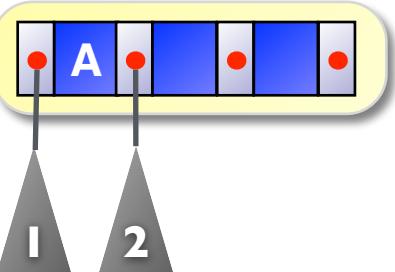
# Korrespondenz: Rot-Schwarz- und (2,4)-Baum

(2,4)-Baum	Rot.Schwarz-Baum
	 <p>Schwarzer Knoten wird mit seinen roten Kindern „verschmolzen“</p>
	

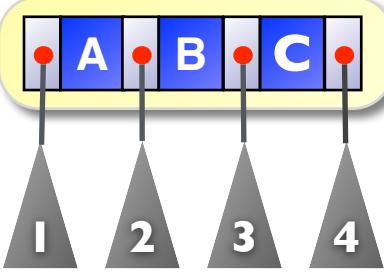
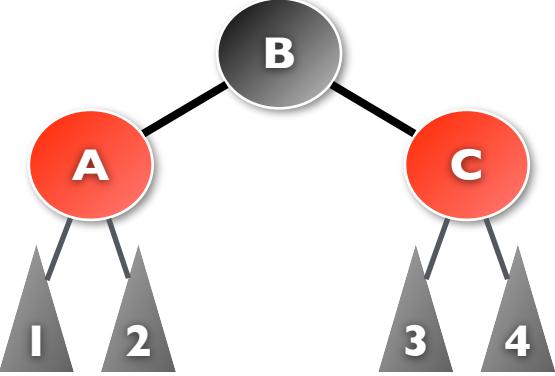
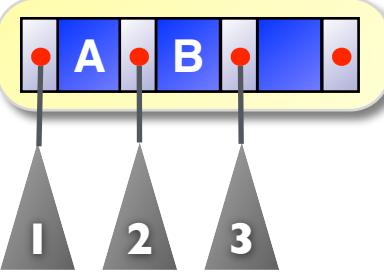
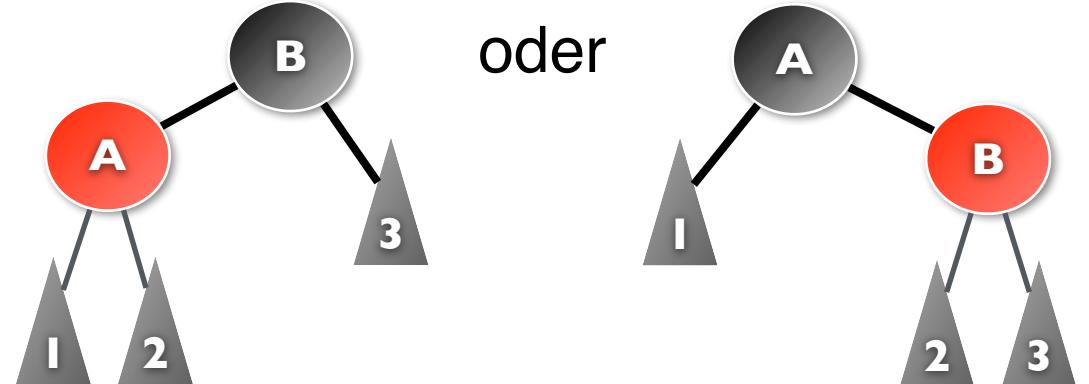
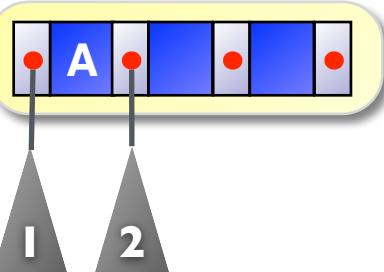
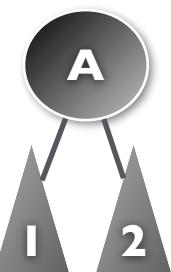
# Korrespondenz: Rot-Schwarz- und (2,4)-Baum

(2,4)-Baum	Rot.Schwarz-Baum
	 <p>Schwarzer Knoten wird mit seinen roten Kindern „verschmolzen“</p>
	<p>oder</p> 

# Korrespondenz: Rot-Schwarz- und (2,4)-Baum

(2,4)-Baum	Rot.Schwarz-Baum
	 <p>Schwarzer Knoten wird mit seinen roten Kindern „verschmolzen“</p>
	 <p>oder</p>
	

# Korrespondenz: Rot-Schwarz- und (2,4)-Baum

(2,4)-Baum	Rot.Schwarz-Baum
	 <p>Schwarzer Knoten wird mit seinen roten Kindern „verschmolzen“</p>
	 <p>oder</p>
	

# Maximale Höhe eines Rot-Schwarz-Baumes

- **Gewinn durch Invariante:**

## Rot-Schwarz-Bäume: Maximale Höhe

Sei  $T$  ein Rot-Schwarz-Baum mit  $n$  inneren Knoten, dann gilt:

$$\text{Höhe}(T) \leq 2 \cdot \text{ld}(n + 1)$$

# Maximale Höhe eines Rot-Schwarz-Baumes

- **Gewinn durch Invariante:**

## Rot-Schwarz-Bäume: Maximale Höhe

Sei  $T$  ein Rot-Schwarz-Baum mit  $n$  inneren Knoten, dann gilt:

$$\text{Höhe}(T) \leq 2 \cdot \text{ld}(n + 1)$$

**Etwas schlechter als AVL-Baum: dort**

$$\text{Höhe}(T) \leq 1.44 \cdot \text{ld}(n + 1)$$

# Einfügen in Rot-Schwarz-Baum

- **Zunächst:**
  - Einfügen wie in Binärbaum
  - Neuer Knoten ist **rot**; seine Bereichsblätter sind **schwarz**

# Einfügen in Rot-Schwarz-Baum

- **Zunächst:**
  - Einfügen wie in Binärbaum
  - Neuer Knoten ist **rot**; seine Bereichsblätter sind **schwarz**

# Einfügen in Rot-Schwarz-Baum

- **Zunächst:**
  - Einfügen wie in Binärbaum
  - Neuer Knoten ist **rot**; seine Bereichsblätter sind **schwarz**
- **Invariante kann gestört sein:**
  1. Jeder Knoten ist rot oder schwarz - bleibt erfüllt!
  2. Die Wurzel ist schwarz - **Störung möglich!**  
Grund: neue Wurzel wurde eingefügt - die ist rot!
  3. Jedes Bereichsblatt ist schwarz - bleibt erfüllt!
  4. Ist ein Knoten rot, so sind seine beiden Kinder schwarz - **Störung möglich!**  
Grund: Vaterknoten des neuen Knotens ist rot
  5. Für jeden Knoten gilt: Jeder Pfad zu einem Blatt enthält dieselbe Anzahl an schwarzen Knoten - bleibt erfüllt!  
Grund: Roter Knoten wurde eingefügt

# Einfügen in Rot-Schwarz-Baum

- **Zunächst:**
  - Einfügen wie in Binärbaum
  - Neuer Knoten ist **rot**; seine Bereichsblätter sind **schwarz**
- **Invariante kann gestört sein:**
  1. Jeder Knoten ist rot oder schwarz - bleibt erfüllt!
  2. Die Wurzel ist schwarz - **Störung möglich!**  
Grund: neue Wurzel wurde eingefügt - die ist rot!
  3. Jedes Bereichsblatt ist schwarz - bleibt erfüllt!
  4. Ist ein Knoten rot, so sind seine beiden Kinder schwarz - **Störung möglich!**  
Grund: Vaterknoten des neuen Knotens ist rot
  5. Für jeden Knoten gilt: Jeder Pfad zu einem Blatt enthält dieselbe Anzahl an schwarzen Knoten - bleibt erfüllt!  
Grund: Roter Knoten wurde eingefügt
- **Falls Störung:**
  - Rotationen zur Wiederherstellung (ähnlich zu AVL-Baum)
  - 6 Fälle sind zu unterscheiden (3 sind symmetrisch)

# Einfügen: Einfache Fälle

- **Erster Schlüssel Z wird eingefügt**

Z ist Wurzel



# Einfügen: Einfache Fälle

- **Erster Schlüssel Z wird eingefügt**

Z ist Wurzel



Umfärben



# Einfügen: Einfache Fälle

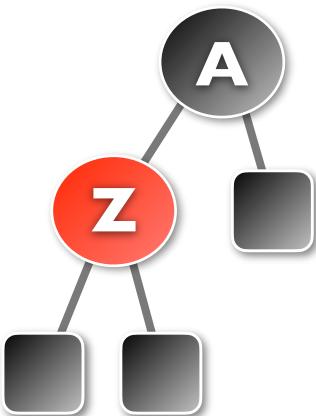
- **Erster Schlüssel Z wird eingefügt**

Z ist Wurzel

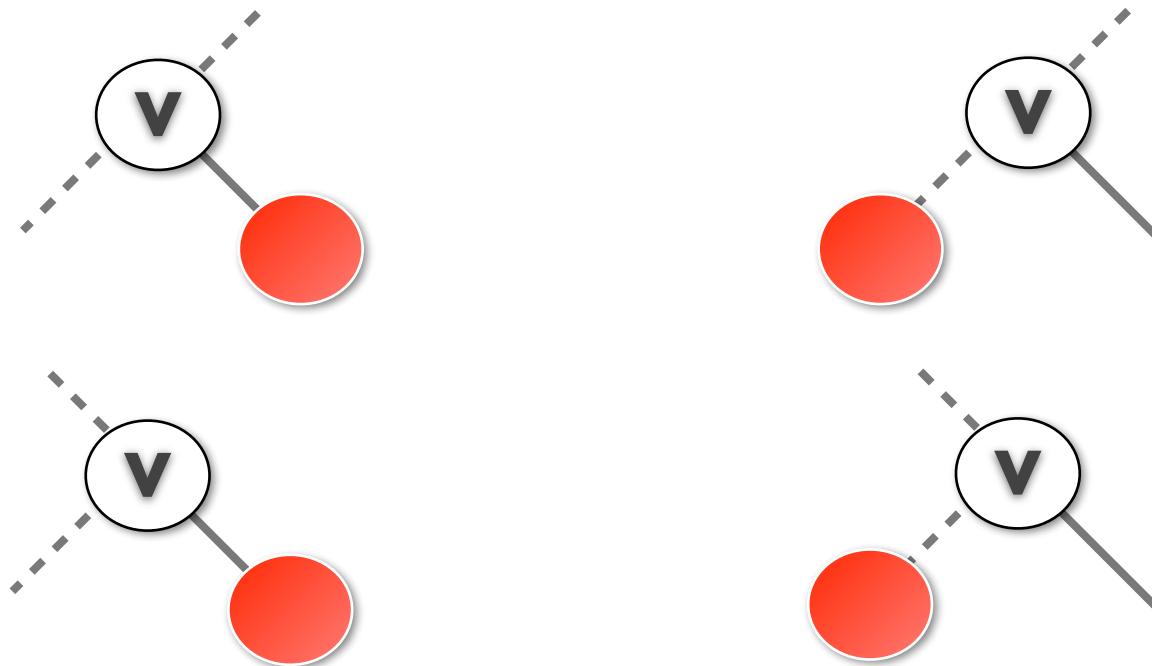


- **Zweiter Schlüssel Z wird eingefügt**

Z ist Blatt. Es ist nichts zu tun ... Z>A analog!



# Einfügen: mögliche Fälle



- **Eingefügter Knoten ist immer rot**
- **Ist Vaterknoten V schwarz, ist nichts zu tun**
- **Ist Vaterknoten rot, so liegt NRR-Konflikt vor**  
... den müssen wir behandeln
- **Wir betrachten nur die oberen Fälle**  
Die unteren Fälle lassen sich analog (symmetrisch) lösen

# 1. Fall: Umfärben

- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **rot**

d.h. Knoten im (2,4)-Baum wird  
mit **Z** übervoll

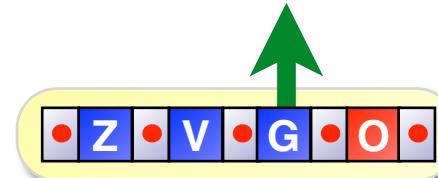
# 1. Fall: Umfärbchen

- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **rot**

d.h. Knoten im (2,4)-Baum wird  
mit **Z** üurvoll



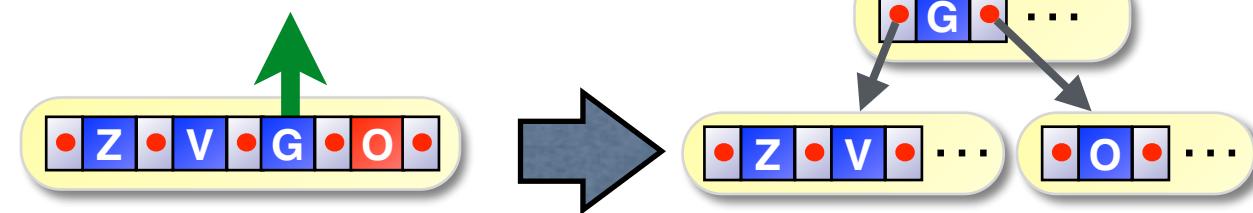
# 1. Fall: Umfärbung

- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **rot**

d.h. Knoten im (2,4)-Baum wird  
mit **Z** üurvoll

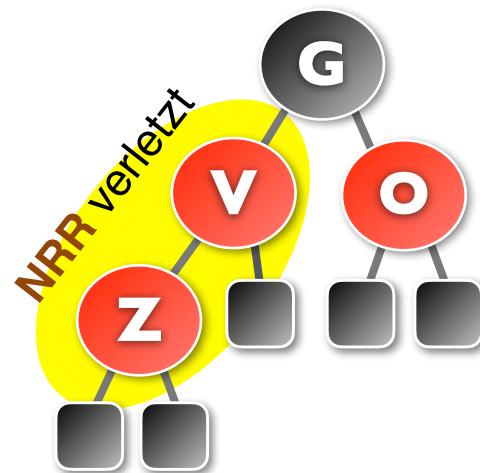
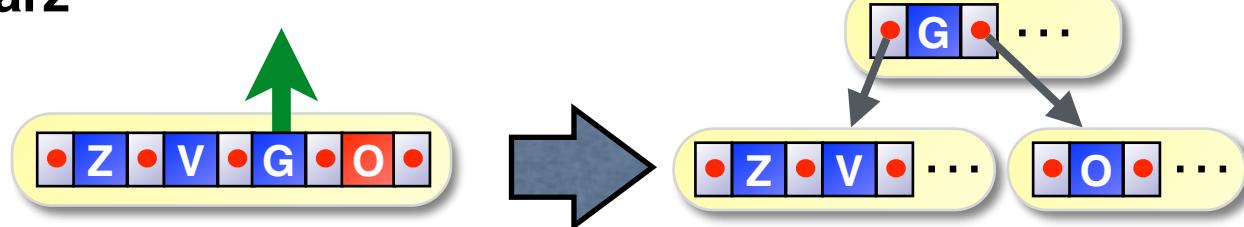


# 1. Fall: Umfärbchen

- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**  
**Onkel O** ist **rot**

d.h. Knoten im (2,4)-Baum wird  
mit **Z** üurvoll



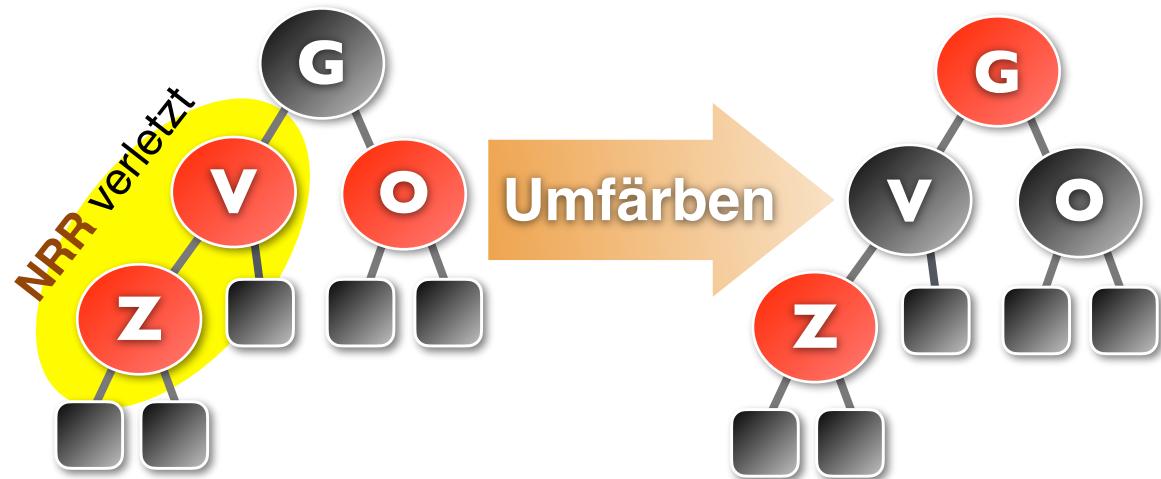
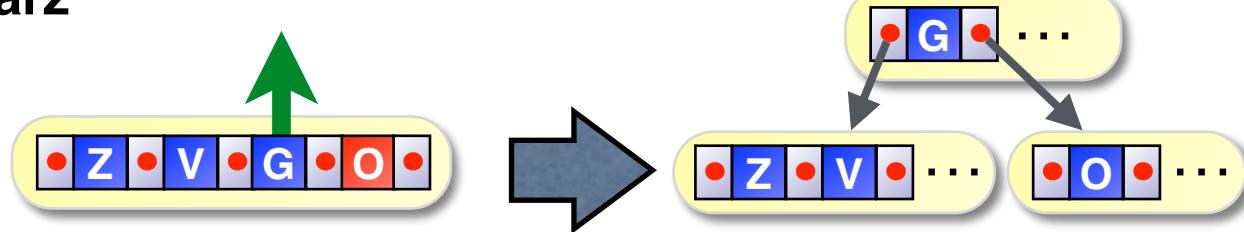
Nach Einfügen von **Z**

# 1. Fall: Umfärbung

- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**  
**Onkel O** ist **rot**

d.h. Knoten im (2,4)-Baum wird  
mit **Z** üurvoll



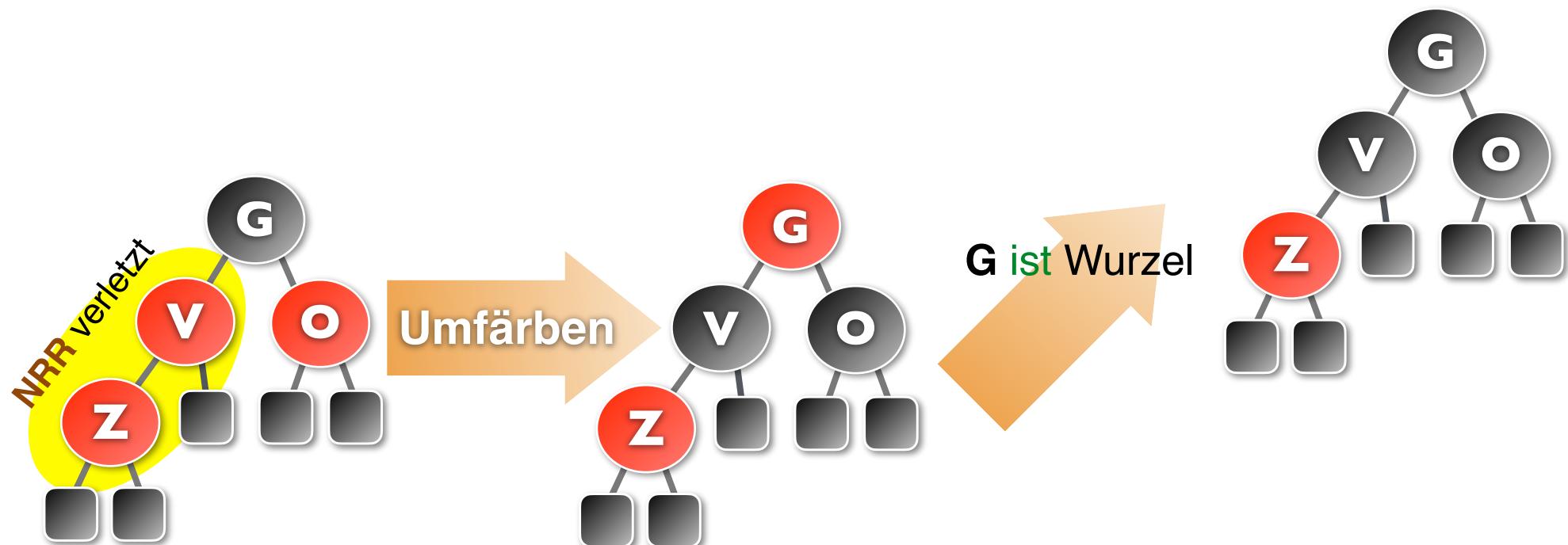
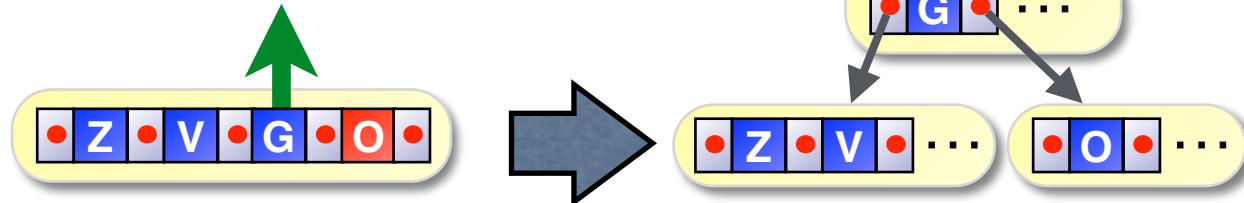
Nach Einfügen von **Z**

# 1. Fall: Umfärbung

- NRR wird verletzt

Z linker Sohn von V; G ist schwarz  
Onkel O ist rot

d.h. Knoten im (2,4)-Baum wird  
mit Z übergeladen



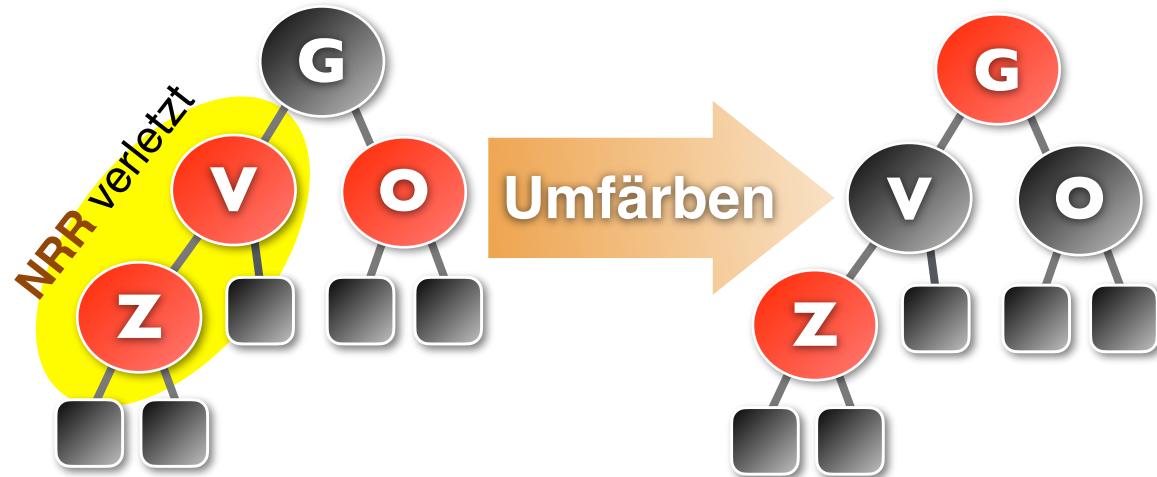
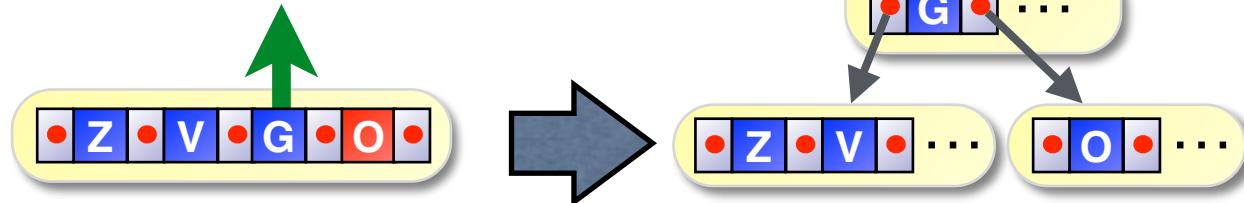
Nach Einfügen von Z

# 1. Fall: Umfärbung

- NRR wird verletzt

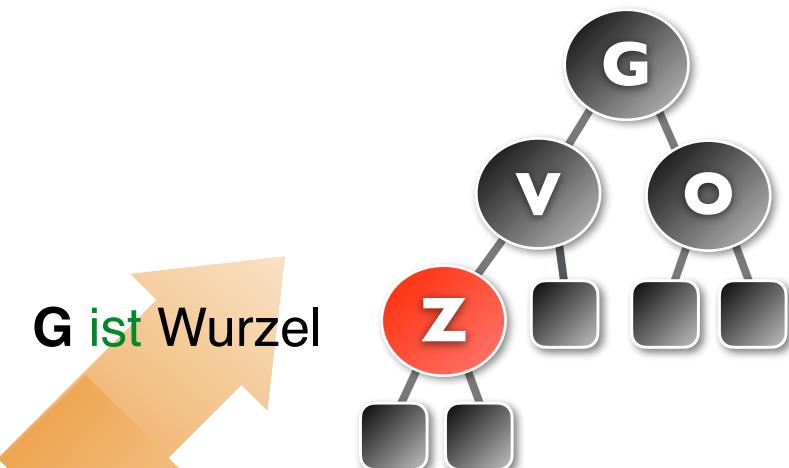
Z linker Sohn von V; G ist schwarz  
Onkel O ist rot

d.h. Knoten im (2,4)-Baum wird mit Z übergeladen



Nach Einfügen von Z

G ist Wurzel  
G nicht Wurzel  
weiterer NRR-Konflikt möglich



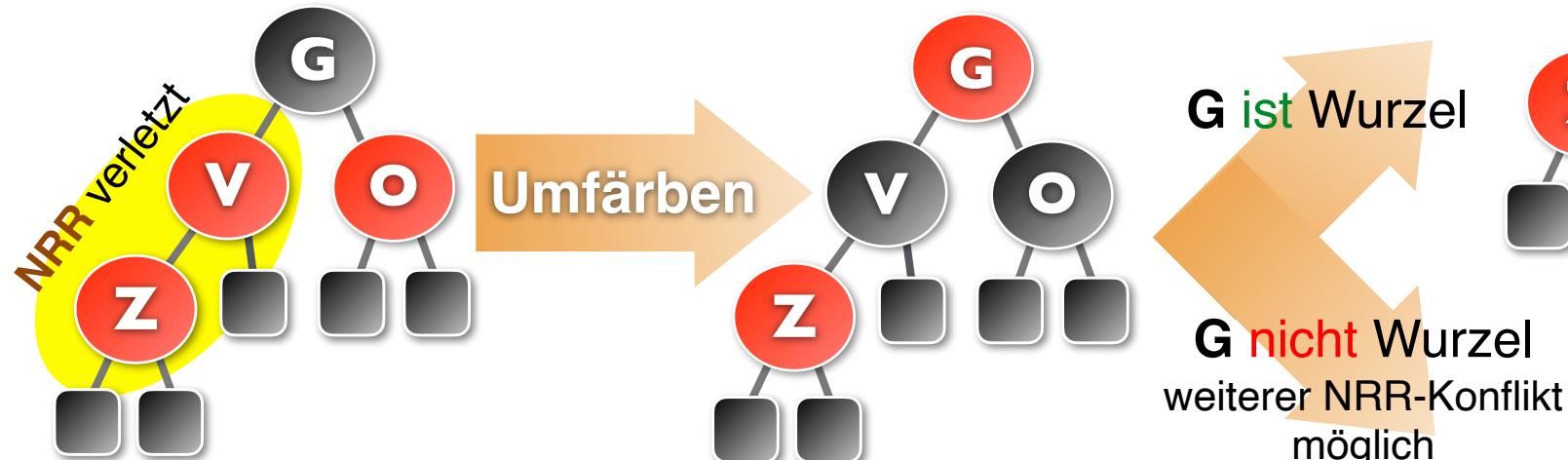
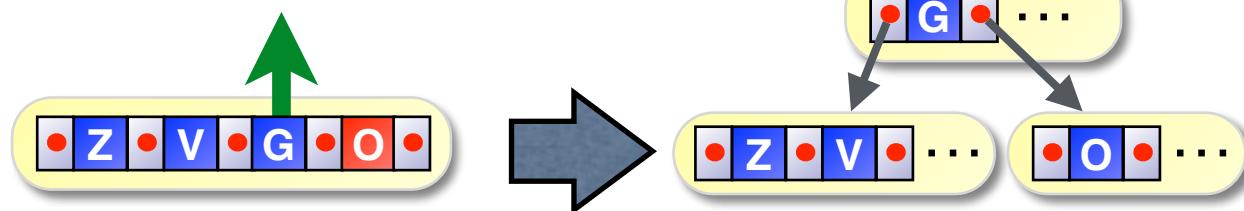
Weitere Umstrukturierungen

# 1. Fall: Umfärbung

- NRR wird verletzt

Z linker Sohn von V; G ist schwarz  
Onkel O ist rot

d.h. Knoten im (2,4)-Baum wird mit Z übergeladen



Nach Einfügen von Z

Analog, falls Z rechtes Kind von V

Weitere Umstrukturierungen

## 2. Fall: LL-Rotation und umfärben

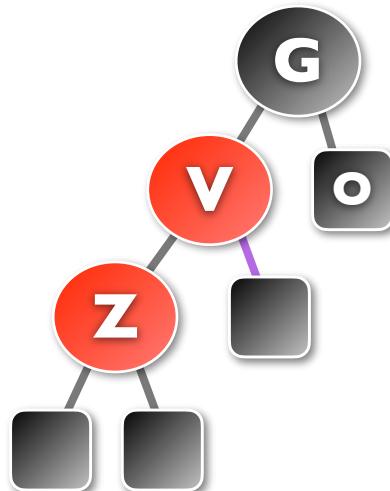
- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist kleinster Schlüssel

→ LL-Rotation und umfärben



Nach Einfügen von **Z**

## 2. Fall: LL-Rotation und umfärben

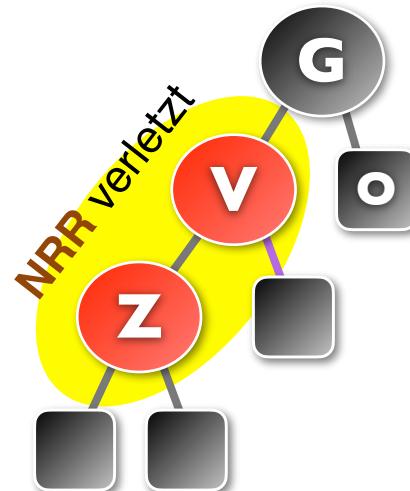
- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist kleinster Schlüssel

→ LL-Rotation und umfärben



Nach Einfügen von **Z**

## 2. Fall: LL-Rotation und umfärben

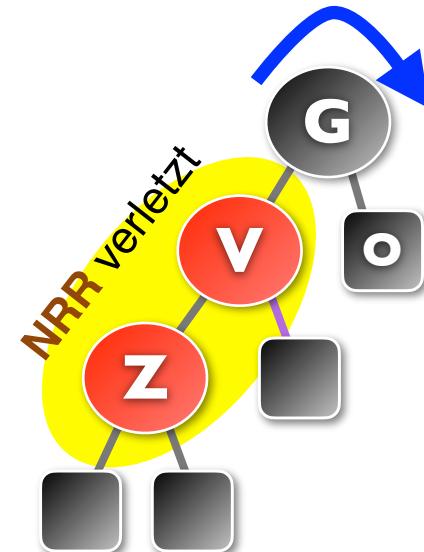
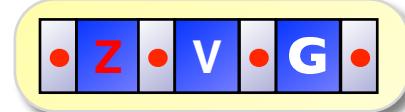
- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist kleinster Schlüssel

→ LL-Rotation und umfärben



Nach Einfügen von **Z**

## 2. Fall: LL-Rotation und umfärben

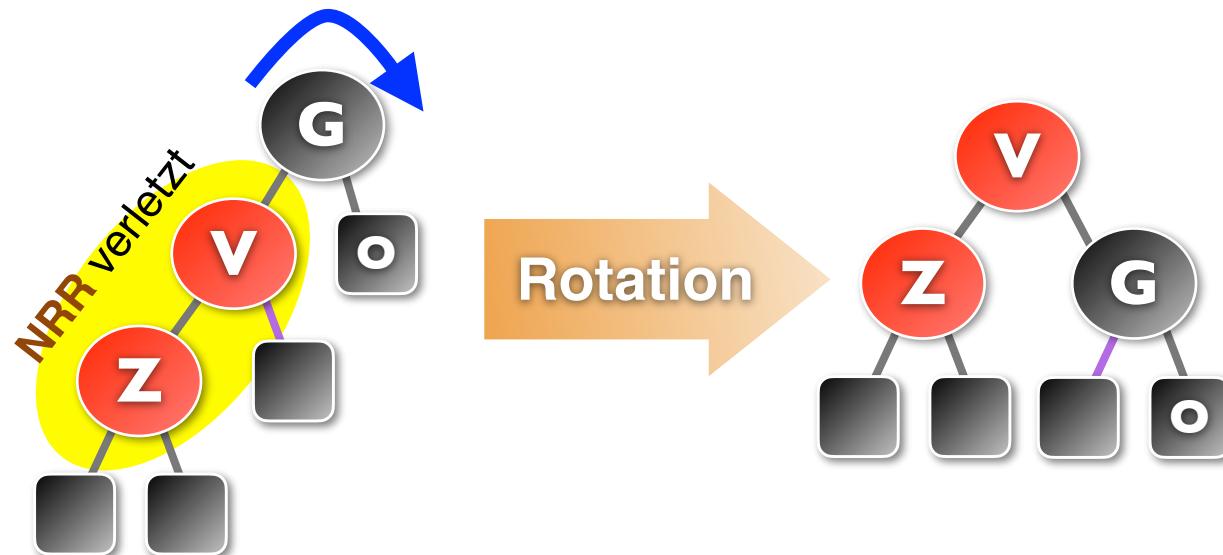
- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist kleinster Schlüssel

→ LL-Rotation und umfärben



Nach Einfügen von **Z**

## 2. Fall: LL-Rotation und umfärbeln

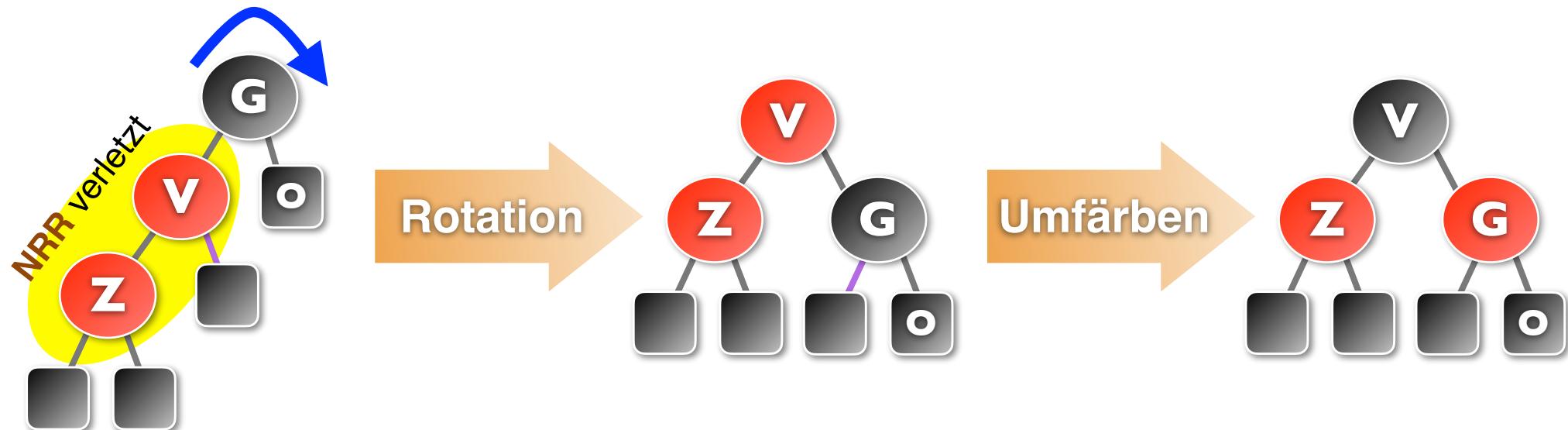
- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist kleinster Schlüssel

→ LL-Rotation und umfärbeln



Nach Einfügen von **Z**

## 2. Fall: LL-Rotation und umfärbeln

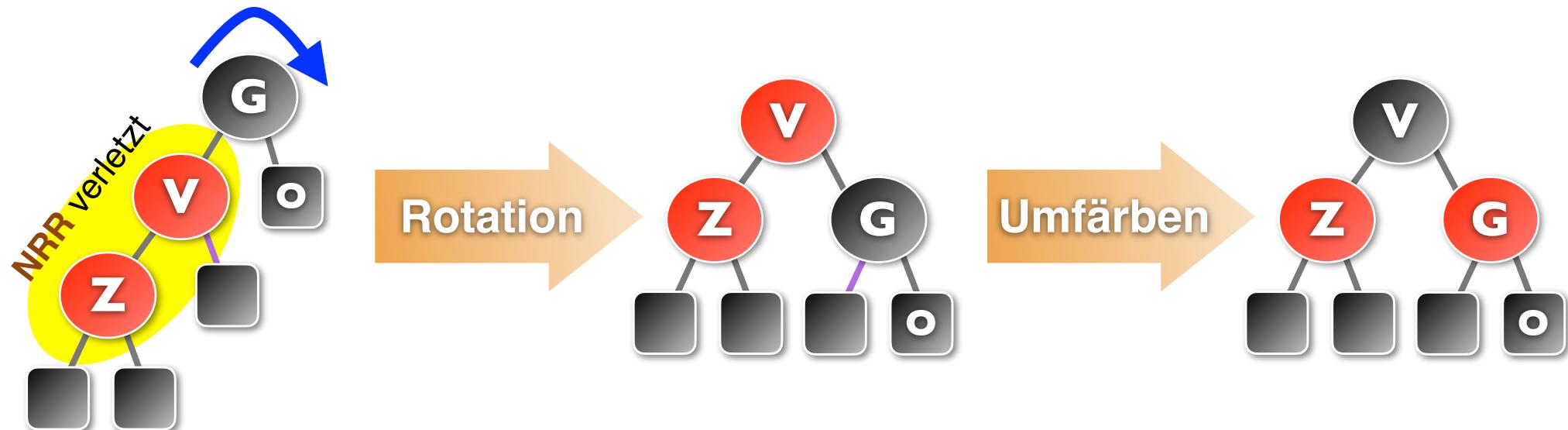
- **NRR wird verletzt**

**Z** linker Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

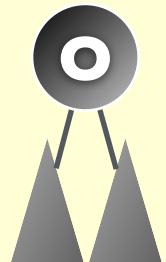
d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist kleinster Schlüssel

→ LL-Rotation und umfärbeln



Nach Einfügen von **Z**

Analog: **O** als  
schwarze Wurzel  
eines Teilbaumes



### 3. Fall: Mit RL-Rotation zu Fall 2

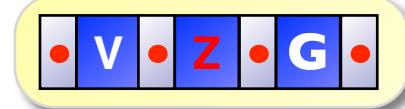
- **NRR wird verletzt**

**Z** rechter Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist mittlerer Schlüssel

→ RL-Rotation führt zu Fall 2



Nach Einfügen von **Z**

### 3. Fall: Mit RL-Rotation zu Fall 2

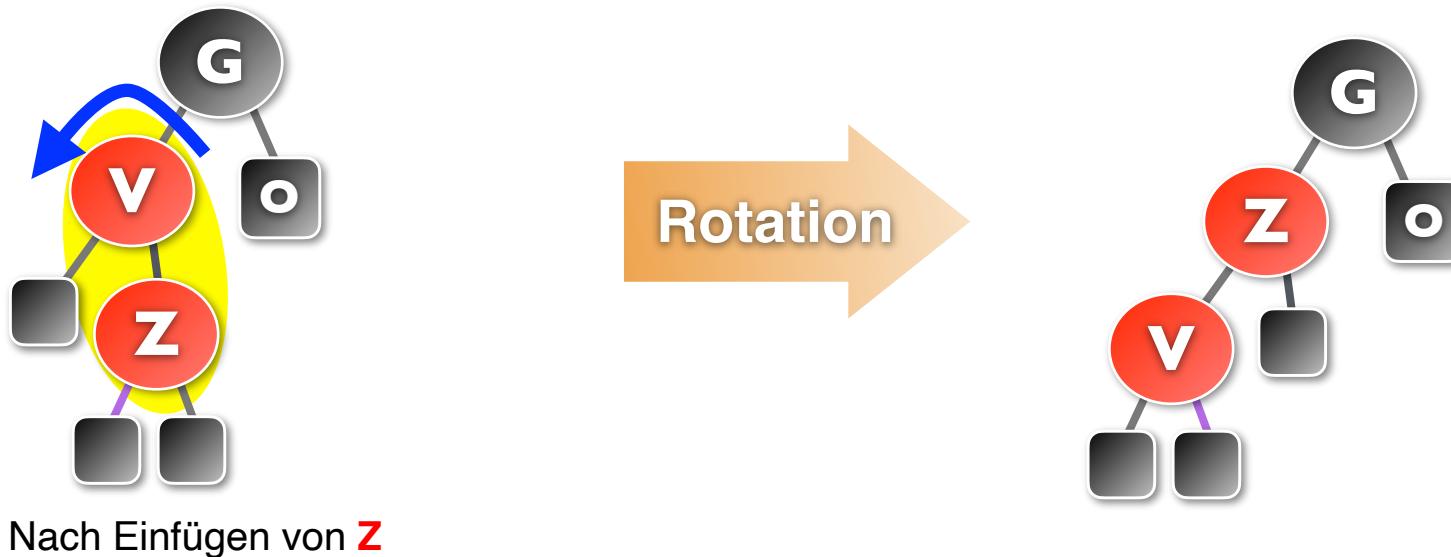
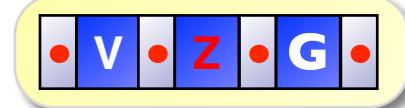
- **NRR wird verletzt**

**Z** rechter Sohn von **V**; **G** ist **schwarz**

**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist mittlerer Schlüssel

→ RL-Rotation führt zu Fall 2



### 3. Fall: Mit RL-Rotation zu Fall 2

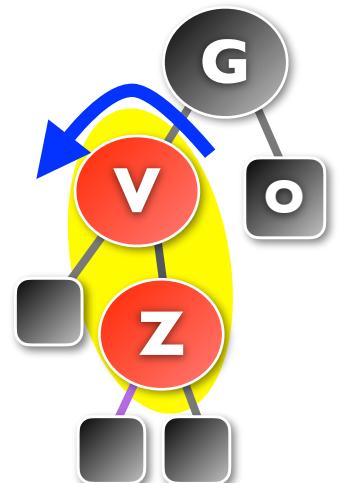
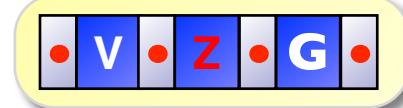
- **NRR wird verletzt**

**Z** rechter Sohn von **V**; **G** ist **schwarz**

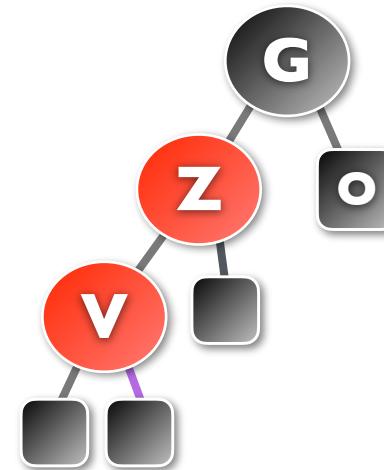
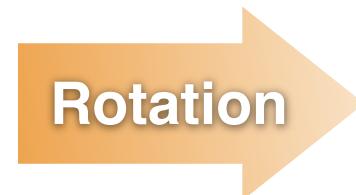
**Onkel O** ist **schwarz**

d.h. Knoten im (2,4)-Baum hat noch Platz und **Z** ist mittlerer Schlüssel

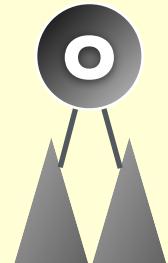
→ RL-Rotation führt zu Fall 2



Nach Einfügen von **Z**



Analog: **O** als  
schwarze Wurzel  
eines Teilbaumes



# V. Suchen in Mengen

## 5. Suchen in Mengen

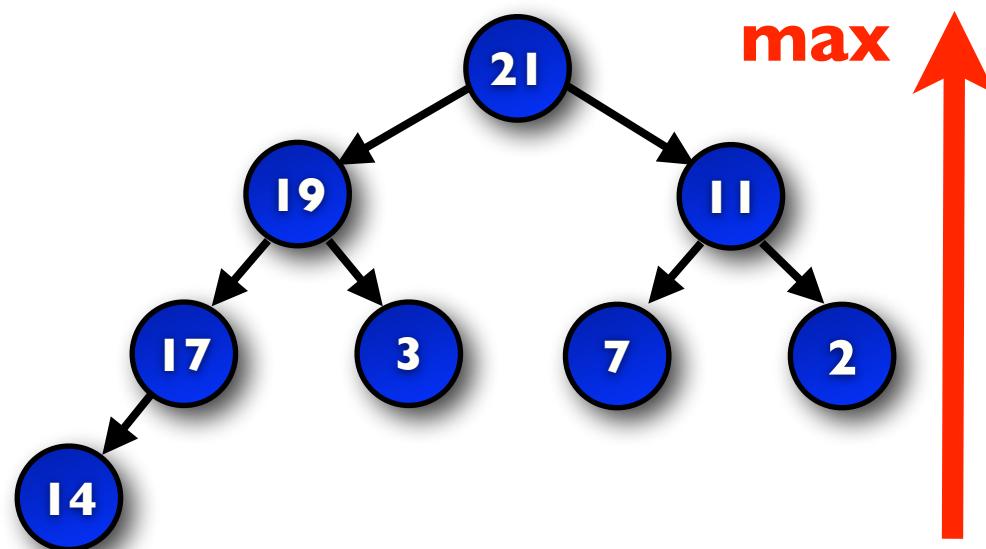
- 5.1. Einführung
- 5.2. Einfache Implementierungen
- 5.3. *Hashing*
- 5.4. Binäre Suchbäume
- 5.5. Balancierte Bäume
  - 5.5.1. Gewichtsbalanceierte Bäume
  - 5.5.2. AVL-Bäume
  - 5.5.3. (a,b)-Bäume
  - 5.5.4. rot/schwarz-Bäume
- 5.6. Priority Queue und Heap**

# Priority Queues

- Auch **Vorrangwarteschlange** oder **Prioritätswarteschlange** genannt
- Eine Warteschlange (Queue), deren Elemente einen Schlüssel (**eine Priorität**) besitzen
- **Wichtige Operationen**
  - $\text{Insert}(a, S)$  Element in Warteschlange einfügen
  - $\text{DeleteMax}(S)$  Element mit höchster Priorität aus Warteschlange entfernen  
(manchmal wird auch  $\text{DeleteMin}(S)$  betrachtet)
- **Anwendungen**
  - Ereignissimulationen (Schlüsselwerte sind Zeiten, zu denen ein Ereignis eingetreten ist)
  - Verteilung von Rechenzeit auf Prozesse

# Datenstrukturen für Priority Queues

- **AVL-Baum**
  - Knoten nach Priorität sortiert
- **Heap**
  - wie bei Heap-Sort - partiell geordneter, **links-vollständiger Baum**
  - in jedem Knoten steht das Maximum des jeweiligen Teilbaumes



**UpHeap:** verläuft analog zu  
**DownHeap:** solange Element  
größer als Vaterknoten, tausche  
mit diesem.

- **Insert( $a, S$ ):** Element am Ende des Feldes einfügen;  
dann **UpHeap** ausführen -  $O(\lg n)$
- **DeleteMax( $S$ ):** Wurzel entfernen; letztes Element an Wurzelposition;  
DownHeap ausführen -  $O(\lg n)$