



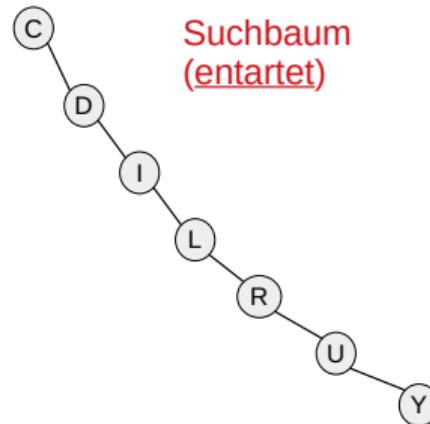
Algorithmen und Datenstrukturen

Kapitel 09: Balancierte Suchbäume

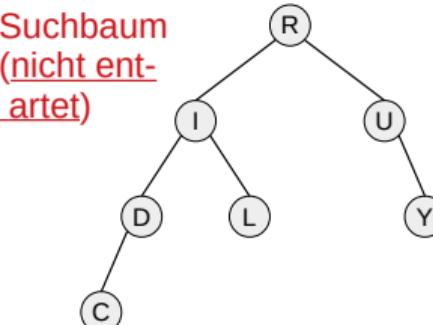
Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Effizienz von Suchbäumen



Suchbaum
(entartet)



Suchbaum
(nicht ent-
artet)

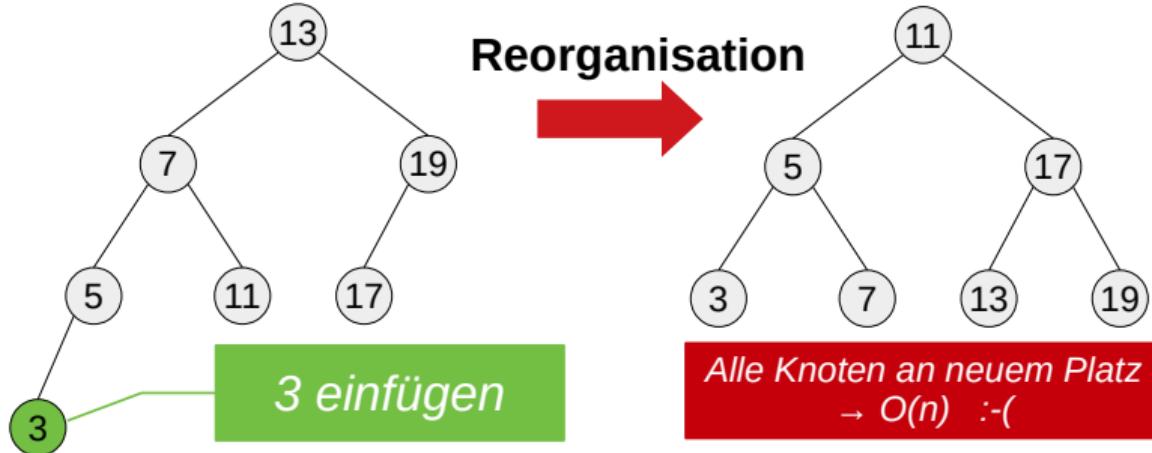
Problem: Entartung von Suchbäumen

- ▶ Bäume können zu **verketteten Listen** “entarten”.
- ▶ Suchen, Einfügen, Löschen kosten im Worst Case $\Theta(n)$ ☹
- ▶ **Ziel:** Baum sollte **Höhe** $O(\log n)$ garantieren.

Balancierte/ausgeglichene Suchbäume

- ▶ 234-Bäume, Red-Black-Trees, B-Bäume

Reorganisation von Suchbäumen



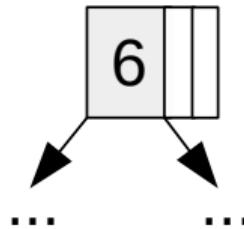
Vollständiges Ausgleichen

- ... müsste bei **jedem Einfügen** einen **(fast) vollständigen** Baum herstellen.
- ... Im schlimmsten Fall Änderung **aller Knoten** → $O(n)$ ☹
- Wir brauchen ein “partielles” **Balancieren**.
- **Ziel:** Suchen = Einfügen = Löschen = $O(\log n)$.

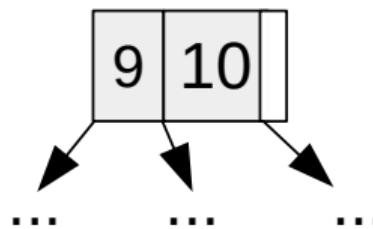
Outline

1. 2-3-4-Bäume
2. 2-3-4-Bäume: Effizienz
3. Red-Black-Trees
4. Red-Black-Trees: Einfügen

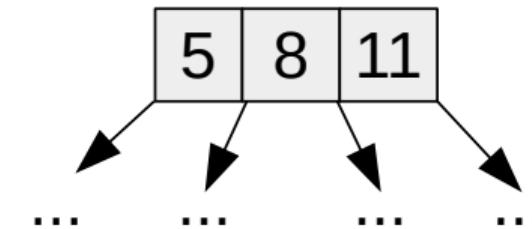
2-3-4-Bäume: Ansatz



2er-Knoten



3er-Knoten



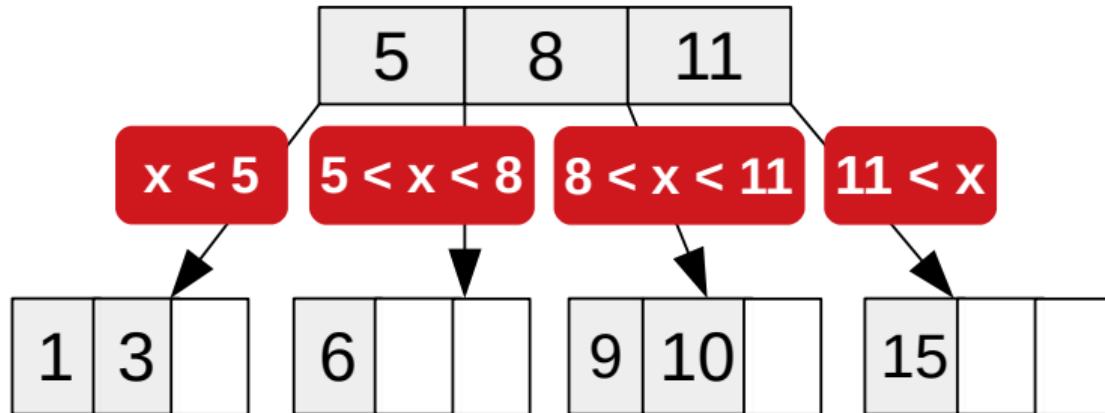
4er-Knoten

Idee: Ein Knoten enthält mehrere Schlüssel

Ein Knoten hat **1-3** (*aufsteigend sortierte*) Schlüssel und 2-4 Kinder:

- ▶ “**2er-Knoten**”: 2 Kinder, dazwischen 1 Schlüssel
- ▶ “**3er-Knoten**”: 3 Kinder, dazwischen 2 Schlüssel
- ▶ “**4er-Knoten**”: 4 Kinder, dazwischen 3 Schlüssel.

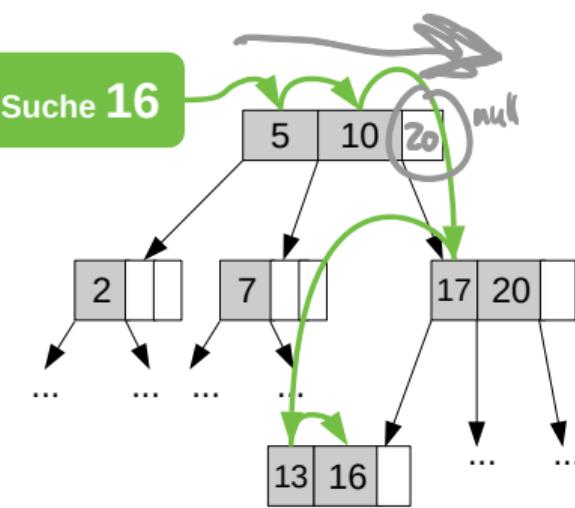
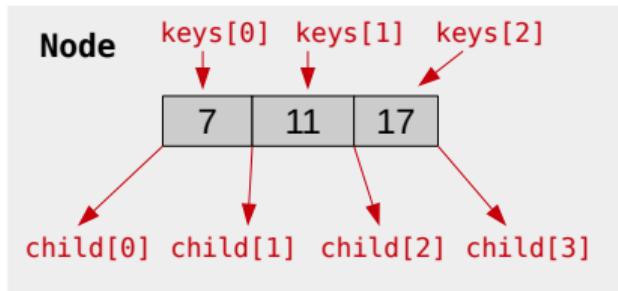
2-3-4-Bäume: Ansatz



Idee: Verallgemeinerte Suchbaum-Eigenschaft

- ▶ Schlüssel werden **von links aufsteigend** “aufgefüllt”.
- ▶ **Allgemeinerer Suchbaum:** Subbaum zwischen A und B enthält nur **Schlüssel zwischen** A und B .

2-3-4-Bäume: Suche (Code)



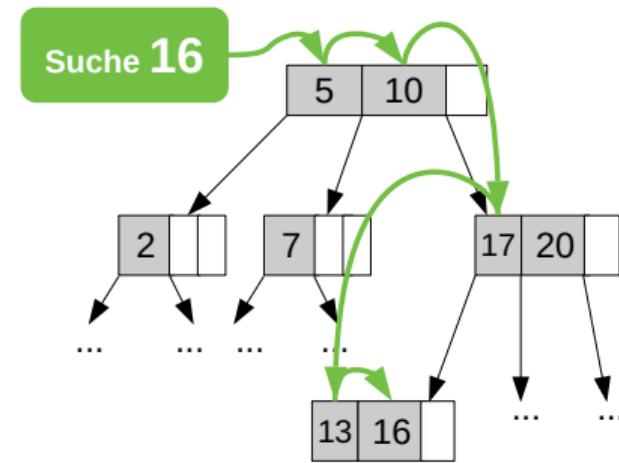
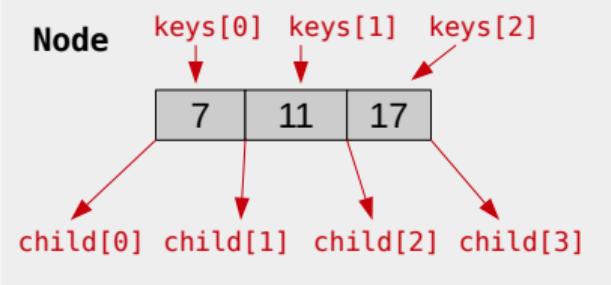
```

function find(Node n, int key):
    if n == null:
        return false
    for i = 0,1,2:
        if keys[i] == key:      // gefunden!
            return true
        if keys[i] == null ||   // nach „unten“
            keys[i] > key:       abbiegen
            return find(child[i], key)
    return find(child[3], key)

result = find(root, key)

```

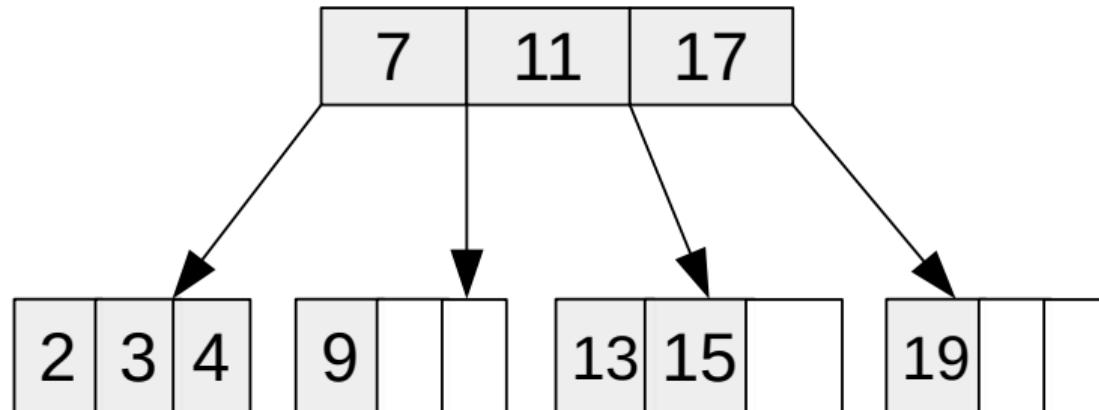
2-3-4-Bäume: Suche (Code)



2-3-4-Bäume: Einfügen

Was ist das Problem beim Einfügen in den 2-3-4-Baum?

- ▶ Füge 16 ein ☺
- ▶ Füge 6 ein ☹



2-3-4-Bäume: Einfügen

Einfüge-Prozedur

Ansatz: An den **Blättern** einfügen!

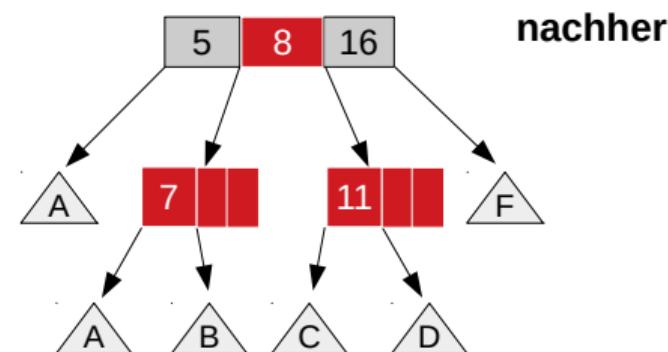
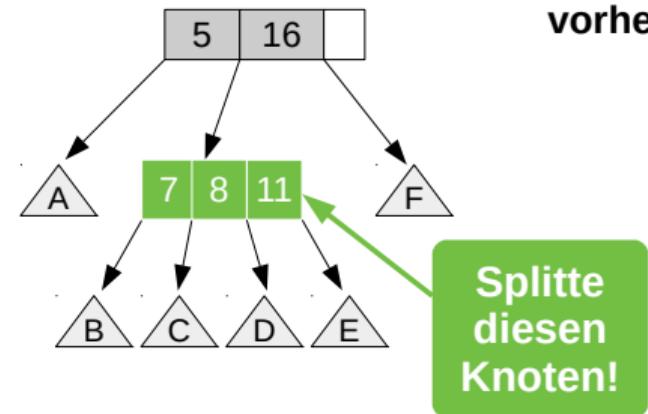
- ▶ Beginne an der **Wurzel**, wandere nach unten (*siehe "Suche"*).
- ▶ Schlüssel gefunden? → **Überschreiben**.
- ▶ Blatt erreicht? → Schlüssel **einfügen**.
- ▶ Entstehen hierbei **überschüssige Schlüssel**?
→ **nach oben** durchreichen.

Reorganisation

- ▶ Wir müssen sicher sein, dass obere Knoten potenzielle **Überläufe** aus dem Blatt aufnehmen können.
- ▶ Deshalb **splitten** wir jeden **vollen Knoten** den wir treffen.

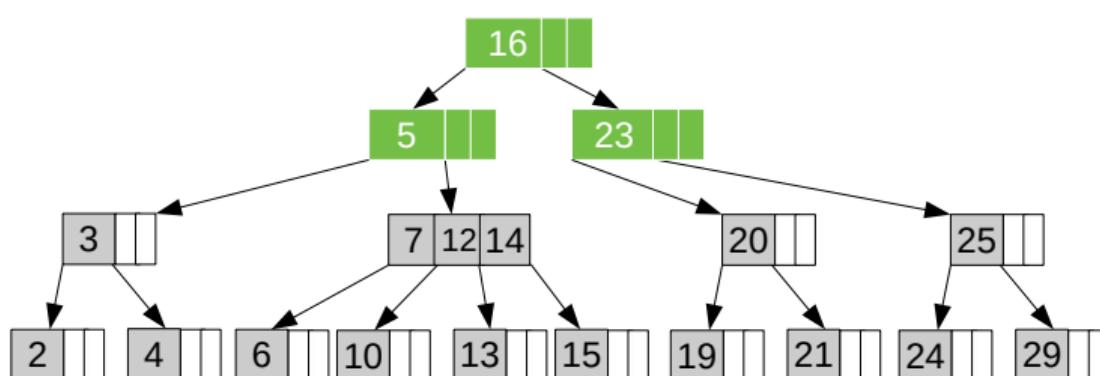
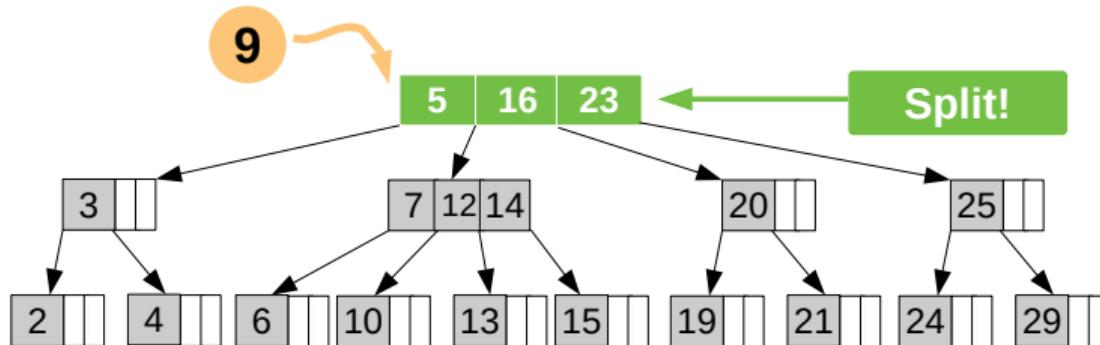
Definition "Split"

- ▶ Aus dem vollen Knoten werden **zwei neue**.
- ▶ Der mittlere Schlüssel wandert **nach oben** (*dort ist Platz, weil wir ja alle vollen Knoten oberhalb gesplittet haben!*).



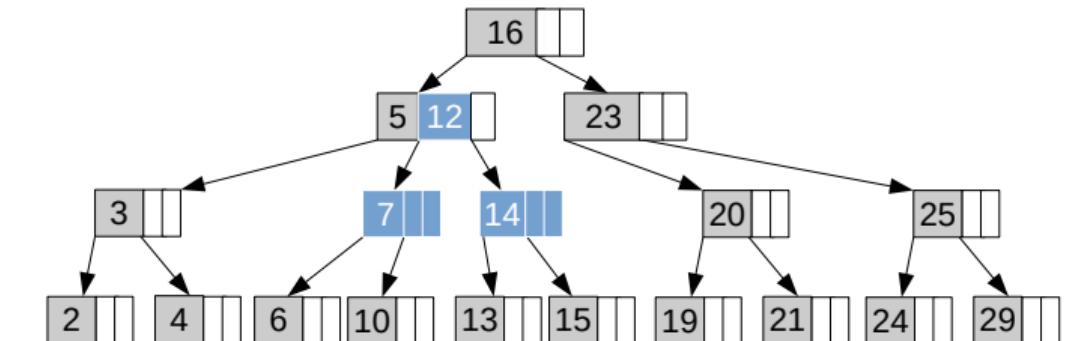
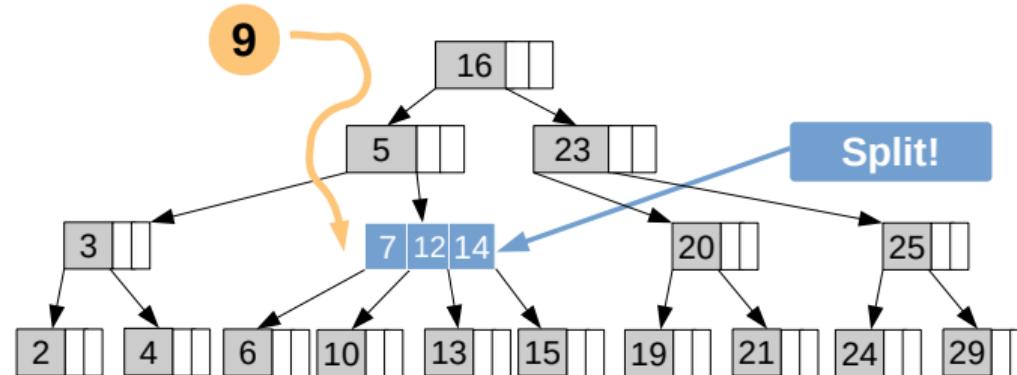
Beispiel: Füge 9 ein

1. Wurzelknoten voll → split.



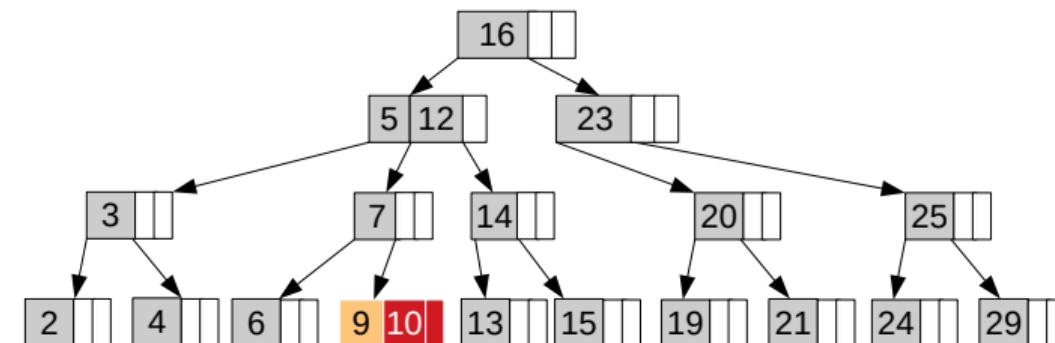
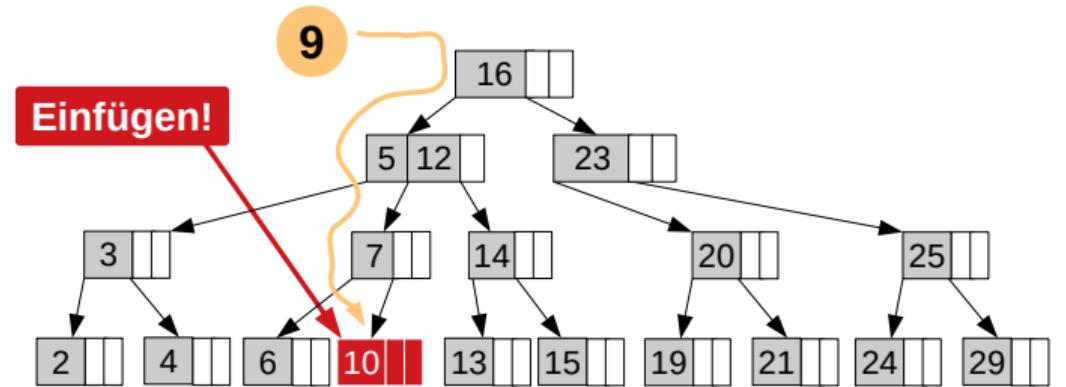
Beispiel: Füge 9 ein

2. Knoten (7-12-14) voll → split.



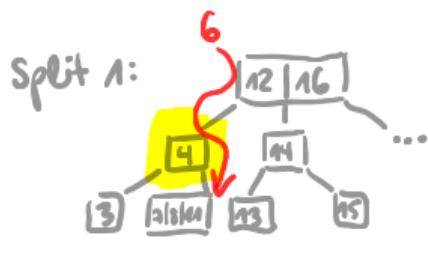
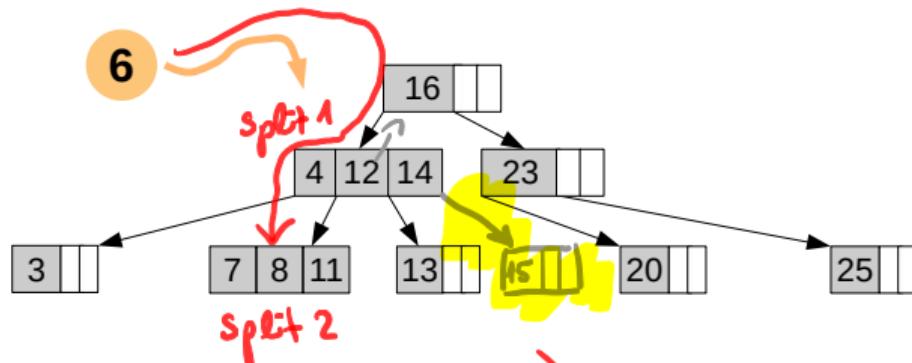
Beispiel: Füge 9 ein

3. Blatt erreicht: 9 einfügen.



Do-234-Yourself

- Was ist an diesem 2-3-4-Baum falsch?
- Fügen Sie die 6 ein.



Do-234-Yourself



Füge Sie nacheinander die Schlüssel 1,2,...,10 in einen (anfangs leeren)
2-3-4-Baum ein!

Do-234-Yourself

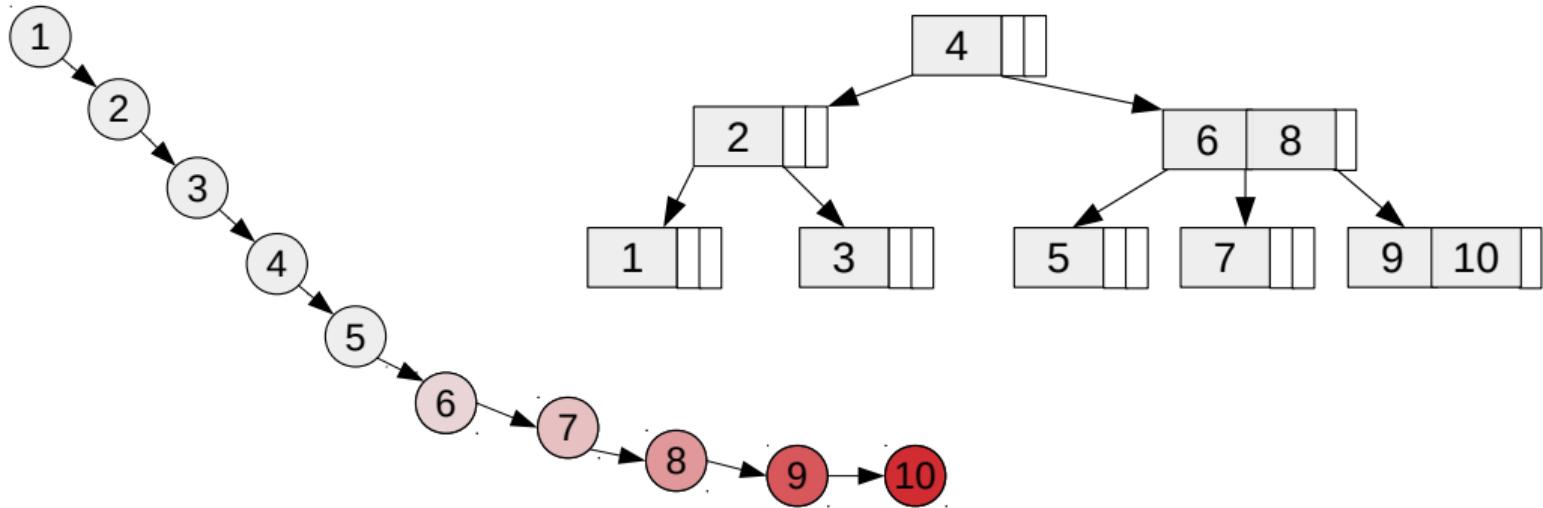




Outline

1. 2-3-4-Bäume
2. 2-3-4-Bäume: Effizienz
3. Red-Black-Trees
4. Red-Black-Trees: Einfügen

2-3-4-Bäume vs. “normale” Suchbäume



2-3-4-Bäume sind ausgeglichen!

- ▶ Die **Höhe** eines 2-3-4-Baums (*siehe Beispiel eben*) steigt nur wenn eine **neue Wurzel** angelegt wird.
- ▶ Hieraus folgt: Alle Blätter befinden sich auf **derselben Ebene**!
- ▶ **Schlechtester Fall:** Eine Seite nur **“leere”**, andere nur **“volle”** Knoten.



2-3-4-Bäume: Effizienz *

Was ist die **maximale Höhe** eines 2-3-4-Baums mit **n Elementen**?



$$n = (2^0 + 2^1 + \dots + 2^{h-1}) = 2^h - 1 \text{ Elemente}$$

$$\rightarrow \underbrace{2^h - 1}_{\leq n} \leq 4^h - 1$$

$$\rightarrow \text{Also: } 2^h \leq n+1$$

$$\Rightarrow h \leq \log_2(n+1)$$



$$n = 3 \cdot (4^0 + 4^1 + \dots + 4^{h-1}) = 3 \cdot \frac{4^h - 1}{4 - 1} = 4^h - 1$$

2-3-4 Bäume
haben logarithmische
Höhe ☺

2-3-4-Bäume: Effizienz *

Wieviele Knoten werden maximal beim **Suchen** in einem 2-3-4-Baum besucht?

Wieviele **Vergleiche** werden dabei maximal durchgeführt, und was ist die Komplexität?

$$\# \text{ Besuchte Knoten} \leq \log_2(n+1)$$

$$\text{Vergleiche je Knoten} \leq 3$$

$$\text{GesamtAufwand} \leq 3 \cdot \log_2(n+1)$$

$$\in O(\log n)$$



2-3-4-Bäume: Effizienz



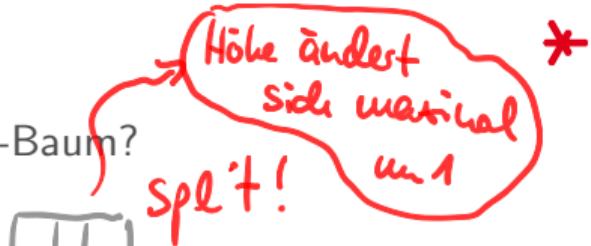
2-3-4-Bäume: Effizienz

Welche Komplexität besitzt das **Einfügen** in einen 2-3-4-Baum?



(Spliten)

Je Split: Kosten $O(1)$ ☺



split!



split!



split!

:



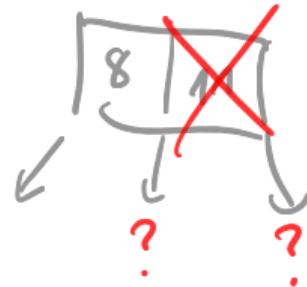
split!

⇒ Gesamtaufwand: $O(\log n)$

\downarrow

Höhe

2-3-4-Bäume: Effizienz



2-3-4-Bäume: Diskussion

2-3-4-Bäume sind **ausgeglichene Bäume**

- ▶ Effizientes Suchen + Einfügen
- ▶ Löschen: nicht besprochen.



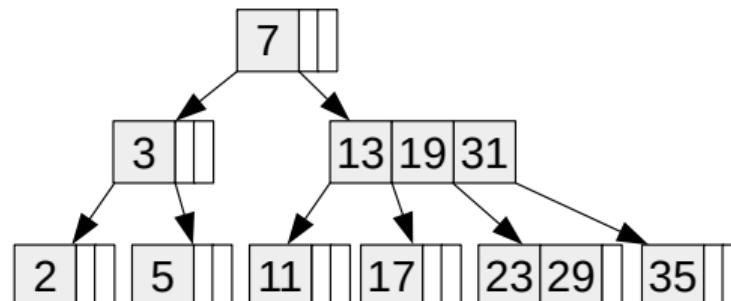
Outline

1. 2-3-4-Bäume
2. 2-3-4-Bäume: Effizienz
3. Red-Black-Trees
4. Red-Black-Trees: Einfügen

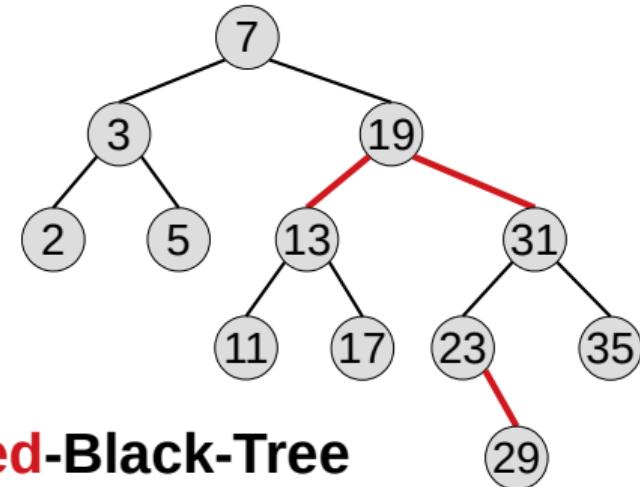
2-3-4-Bäume → Red-Black-Trees

Red-Black-Trees (Rot-Schwarz-Bäume) ...

- sind **binäre Versionen** von 2-3-4-Bäumen.
- werden im **JCF** verwendet (TreeMap).
- Einfügen / Suchen / Löschen: **Garantiert $O(\log(n))$** ☺

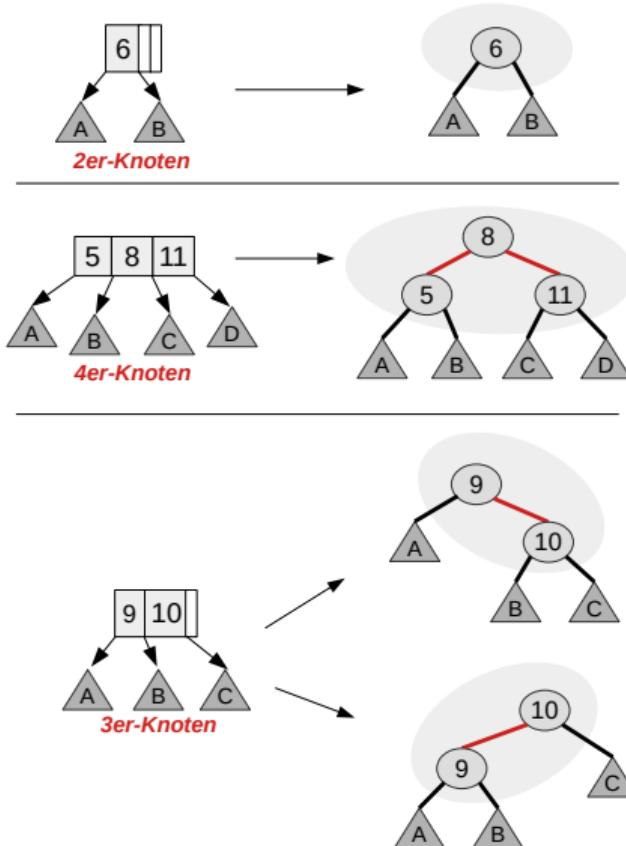


2-3-4-Baum



Red-Black-Tree

Red-Black-Trees: Struktur



Aus jedem Knoten des 2-3-4-Baums wird eine Gruppe von Knoten im Red-Black-Tree, verbunden durch spezielle **rote Kanten**:

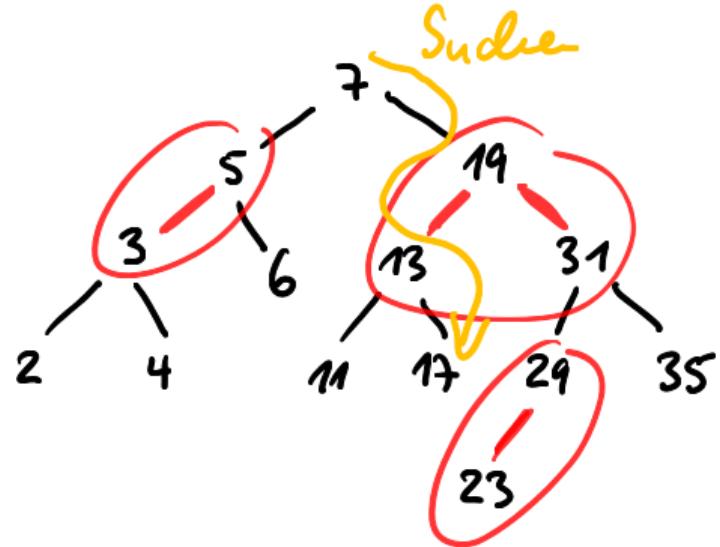
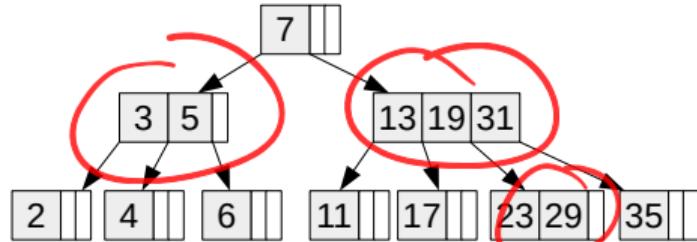
- ▶ Ein **2er-Knoten** wird zu **einem** normalen binären Knoten.
- ▶ Ein **4er-Knoten** wird zu **drei** binären Knoten, verbunden durch zwei rote Kanten.
- ▶ Ein **3er-Knoten** wird zu **zwei** binären Knoten, verbunden durch eine rote Kante (*zwei Varianten: beide Werte können "oben" liegen*)

Zusätzlich besitzt der Red-Black-Tree dieselben (**“schwarzen”**) Kanten wie der 2-3-4-Baum.

Red-Black-Trees: Beispiel

Stellen Sie diesen 2-3-4-Baum als Red-Black-Tree dar!

Suchbaum!



Red-Black-Trees: Beispiel



Red-Black-Trees: Implementierung

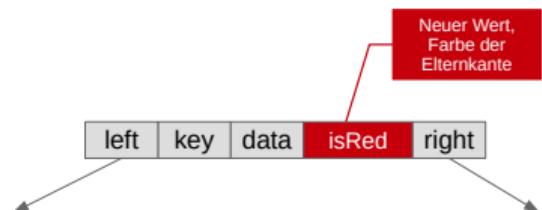
Red-Black-Trees sind Suchbäume!

- ▶ **Kindknoten, Schlüssel, Wert:**
Wie im normalen Suchbaum.
- ▶ **Suche:** Implementierung wie im
“normalen” Suchbaum
(Farbe der Kanten spielt keine Rolle!).

Red-Black-Trees erweitern Suchbäume

- ▶ **Subklassen** für Baum und Knoten.
- ▶ Wir speichern die Farbe jeder Kante
im **Kindknoten** der Kante.
- ▶ Hierzu: Zusätzliches **Attribut isRed** der Knotenklasse: ist genau dann true wenn
die Elternkante rot ist.
- ▶ Dieses Attribut wird bei Einfüge- und Löschoperationen
genutzt, um den Baum **balanciert** zu halten.

```
class RBNode extends Node {  
    private boolean isRed;  
  
    RBNode(T value) {  
        super(value);  
        isRed = false;  
    }  
  
}  
  
public class RBTree extends SearchTree {  
    // reimplement insert(), delete()  
}
```



Outline

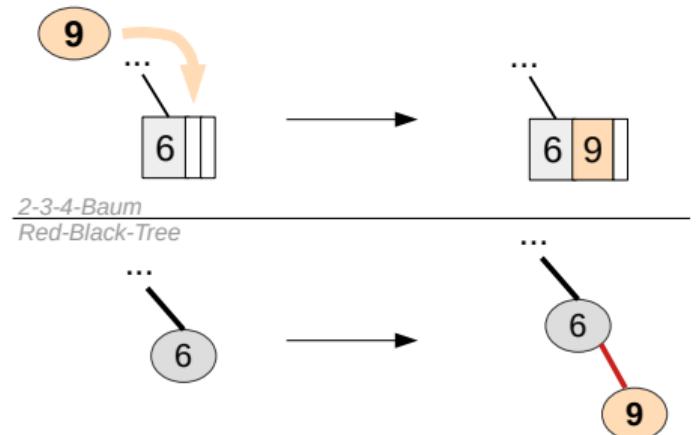
1. 2-3-4-Bäume
2. 2-3-4-Bäume: Effizienz
3. Red-Black-Trees
4. Red-Black-Trees: Einfügen

Red-Black-Trees: Einfügen



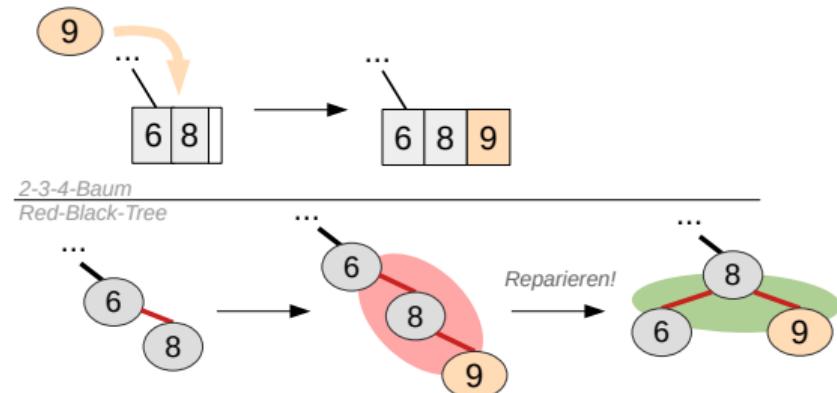
Grundprinzip: Wie im 2-3-4-Baum!

- Wandere nach unten zu einem **Blatt**.
- Falls 4er-Knoten angetroffen:
“Splitten” (gleich).
- Neuen Knoten mit einer **roten Kante** anhängen (*rechts*).



Was ist neu?

- Gegebenenfalls müssen wir das 2-3-4-Blatt **reparieren**, um **zwei aufeinanderfolgende rote Kanten** zu vermeiden.

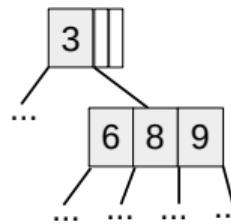


Red-Black-Trees: Einfügen

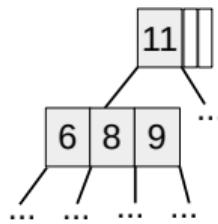
- Beim Durchlaufen des Baums bis zum Blatt **splitten** wir jeden **4er-Knoten**.
- Hierbei entstehen sogenannte **“Roll-Operationen”**, die die Balanciertheit des Baums sicherstellen.

Fall 1

(a)

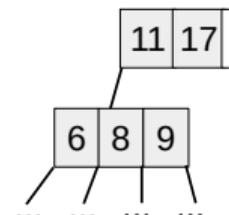


(b)

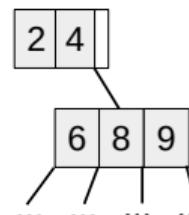


Fall 2

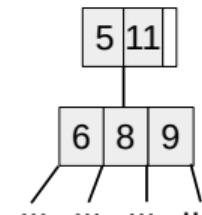
(a)



(b)



(c)



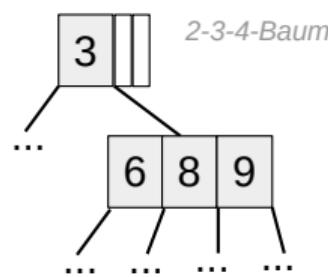
Wir unterscheiden verschiedene Fälle

- 2er-Knoten über dem 4er-Knoten (links(a) oder rechts(b))
- 3er-Knoten über dem 4er-Knoten (links(a) oder rechts(b) oder mittig(c)).

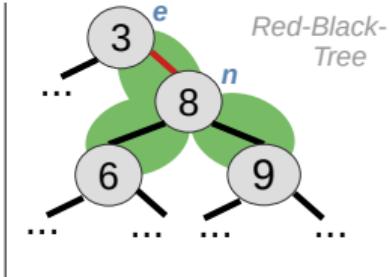
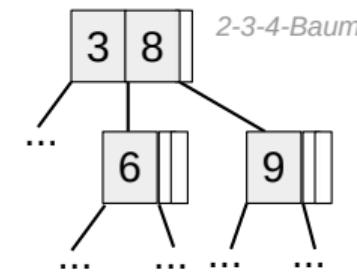
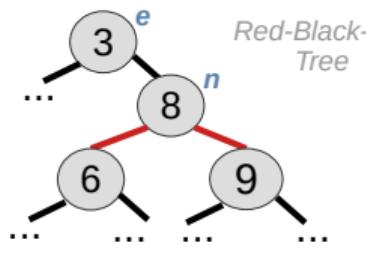
Fall 1a/b: 2er-Knoten über dem 4er-Knoten

- Wir betrachten **Fall 1a** (1b geht analog).
- n sei das oberste Element des **4er-Knotens** (erkannt durch zwei rote Kanten).
- e ist der Elternknoten von n .

Ausgangssituation



Zielsituation

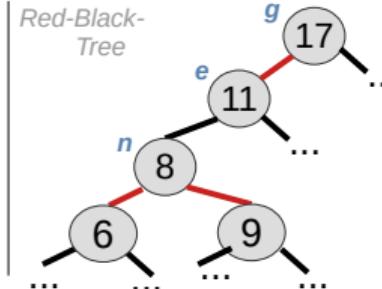
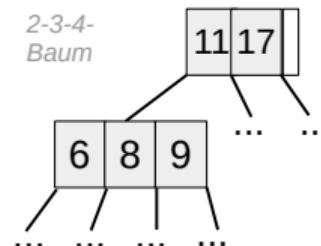


Hierfür müssen wir einfach nur **drei Kanten umfärben!**

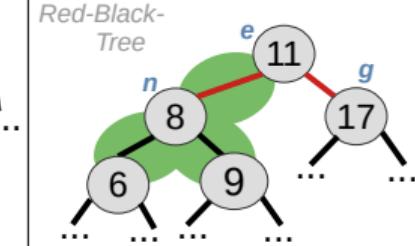
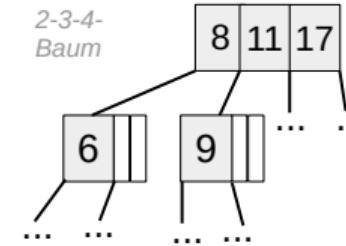
Fall 2a/b: 3er über 4er (links oder rechts)

- Wir betrachten **Fall 2a** (2b geht analog).
- n sei das oberste Element des **4er-Knotens**, e sein Elternknoten, g sein Großelternknoten.

Ausgangssituation

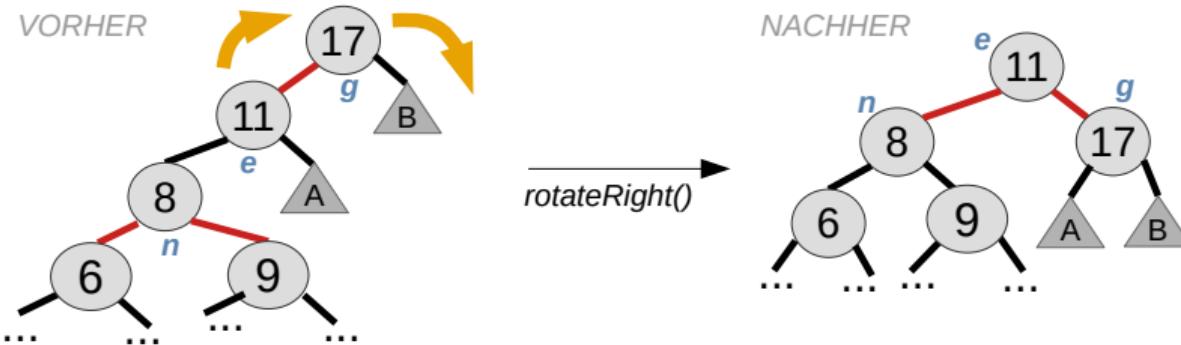


Zielsituation



Hierbei werden nicht nur **Kanten umgefärbt**, sondern auch die **Knotenkonstellation ändert sich**.

Fall 2a/b: Rotation



Es wurde eine sogenannte **Rechts-Rotation** durchgeführt

- ▶ Knoten e rotiert nach rechts oben
- ▶ Knoten g rotiert nach rechts unten.

Anmerkungen

- ▶ Die **Suchbaumeigenschaft** bleibt hierbei erhalten.
- ▶ Die Teilstruktur **verliert an Höhe** ☺.
- ▶ Fall 2b würde analog durch eine Linksrotation behandelt.

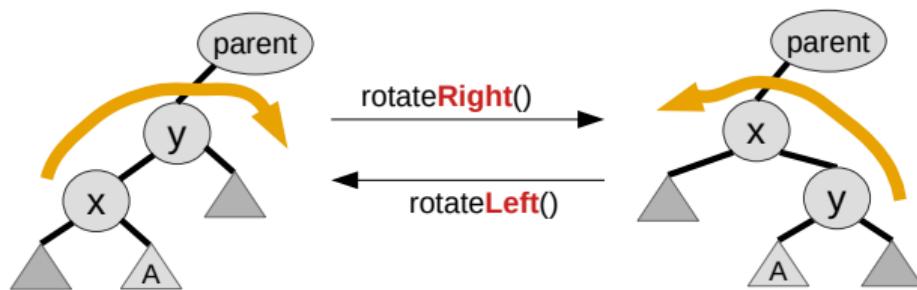
Rotation: Implementierung

```
private RBNode
    rotateRight(RBNode y) {
    RBNode x = y.left;
    y.left = x.right;
    x.right = y;
    return x;
}
```

```
private RBNode
    rotateLeft(RBNode x) {
    RBNode y = x.right;
    x.right = y.left;
    y.left = x;
    return y;
}
```

```
RBNode parent = ...;
parent.left =
    rotateRight(parent.left);
```

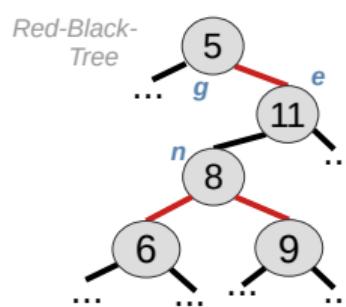
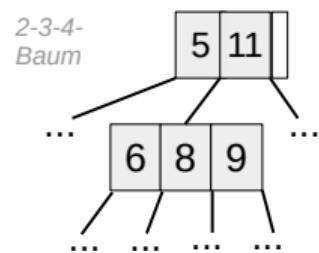
- ▶ Rotationen sind **einfach** zu implementieren!
- ▶ Wir ändern nur **zwei Referenzen** im Baum.
- ▶ Rechts- und Linksrotation sind sehr ähnlich.



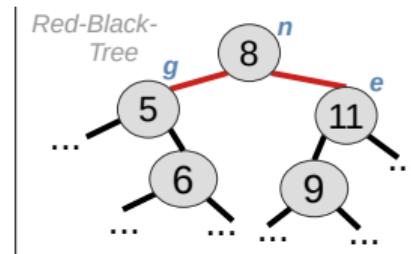
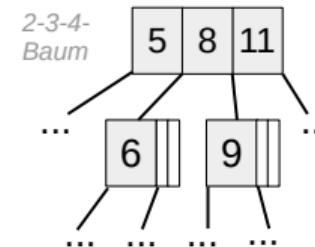
Fall 2c: 3er über 4er (mittig)

Letzter Fall: Der 4er-Knoten hängt **mittig** unter einem 3er-Knoten.

Ausgangssituation

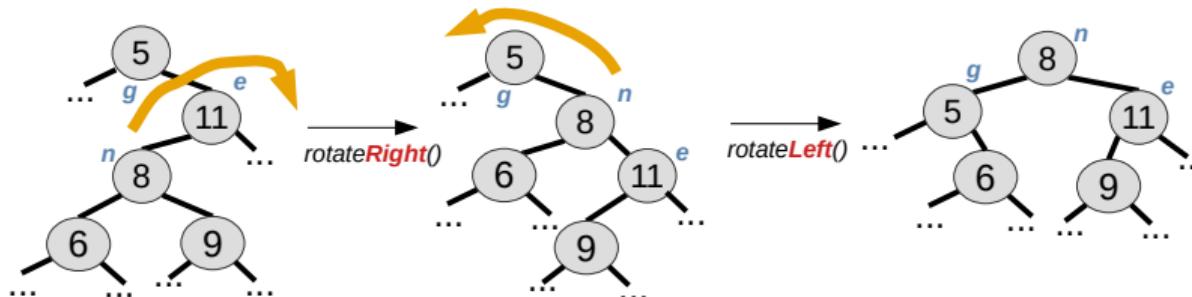


Zielsituation



- Hier ändert sich die **Knotenkonstellation drastisch** (n wandert ganz nach oben).
- Auch hier wird die Struktur **flacher** 😊.

Fall 2c: Doppelritation



Es wird eine sogenannte **Doppel-Rotation** durchgeführt

- ▶ **Schritt 1:** Knoten n rotiert nach **rechts oben** (auf die Position von e).
- ▶ **Schritt 2:** Knoten n rotiert nach **links oben** (auf die Position von g).

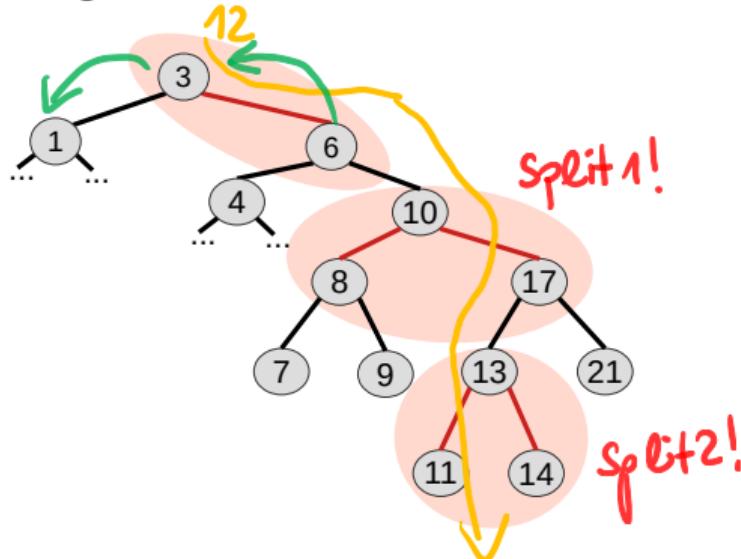
Anmerkungen

- ▶ Die **Suchbaumeigenschaft** bleibt hierbei erhalten.
- ▶ Die Teilstruktur **verliert an Höhe** 😊.

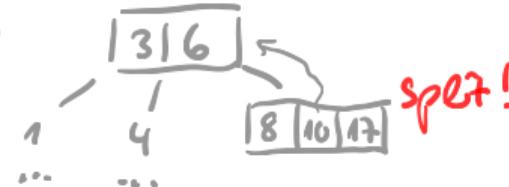
Do-Einfügen-in-Red-Black-Trees-Yourself



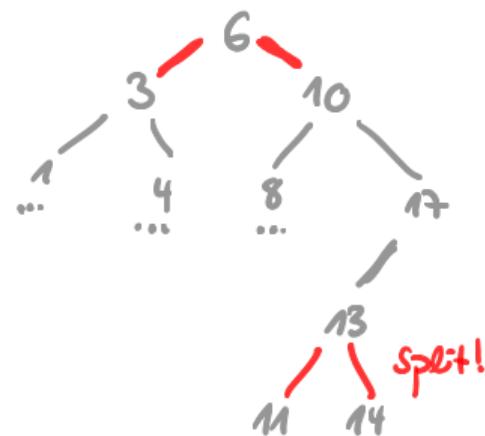
Fügen Sie die 12 ein!



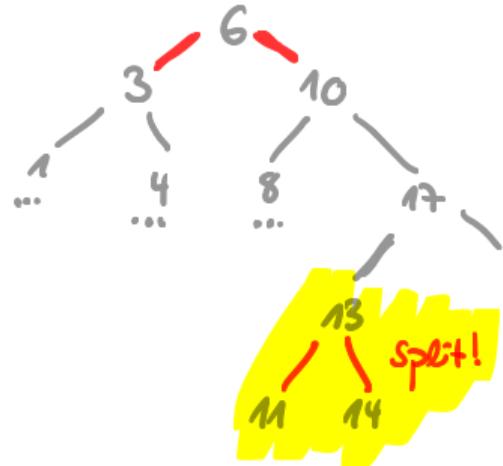
Split 1:



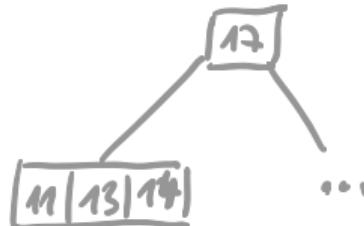
Fall 25 (Linkssrotation)



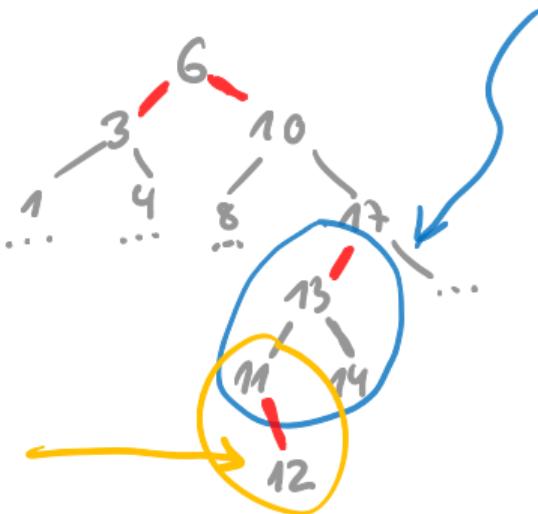
Do-Einfügen-in-Red-Black-Trees-Yourself



Split 2:



Fall 1b : Umläufen!



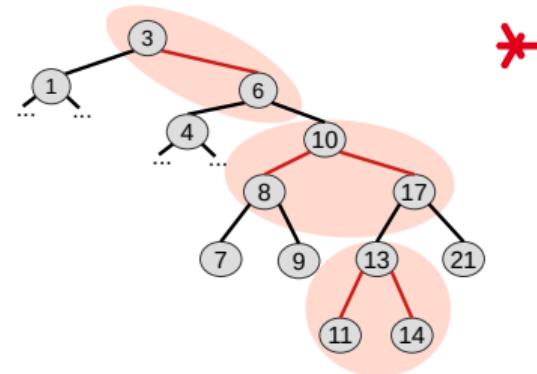
Als letztes:
Einfügen!

Do-Einfügen-in-Red-Black-Trees-Yourself



Red-Black-Trees: Eigenschaften

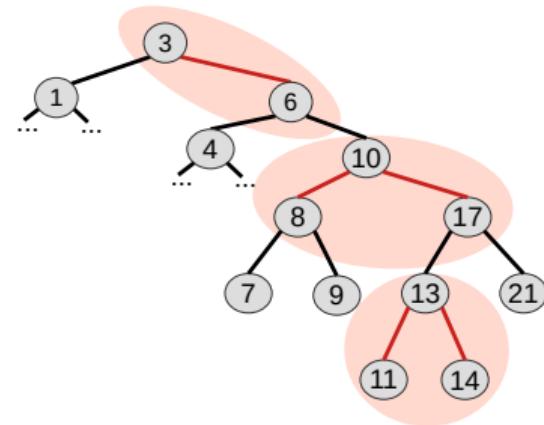
Stimmt diese Behauptung? “Ein Red-Black-Tree hat nie zwei aufeinanderfolgende rote Kanten.”



Red-Black-Trees: Eigenschaften

Höhe von Red-Black-Trees

- ▶ Vergleiche Red-Black-Tree und 2-3-4-Baum...
- ▶ Höhe 2-3-4-Baum: $O(\log n)$ schwarze Kanten.
- ▶ Höhe Red-Black-Tree: $O(\log n)$ schwarzen Kanten, plus max. $O(\log n)$ rote Kanten.
- ▶ Ein Red-Black-Tree hat also maximal die **doppelte Höhe** des entsprechenden 2-3-4-Baums, also ebenfalls $O(\log n)$.



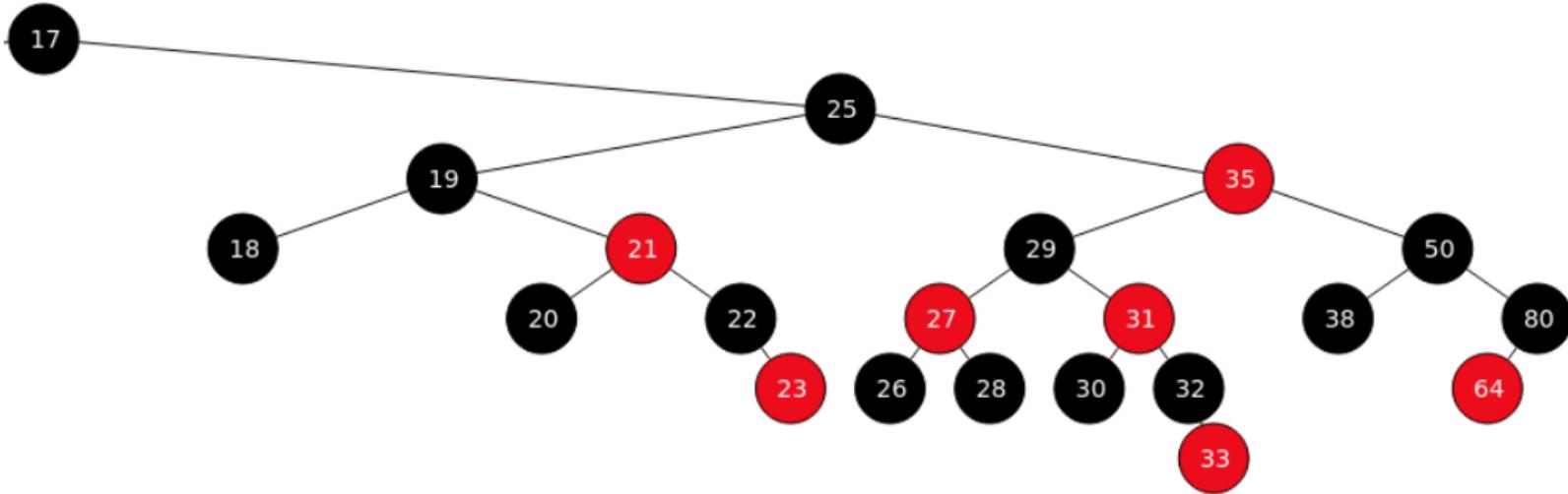
Komplexität der Suche?

- ▶ Besuche maximal $O(\log n)$ Knoten (Wurzel → Blatt).
- ▶ Je Knoten $O(1)$ Operationen → **insgesamt $O(\log n)$** ☺

Komplexität des Einfügens?

- ▶ Besuche maximal $O(\log n)$ Knoten (Wurzel → Blatt).
- ▶ Je Knoten $O(1)$ Operationen (*Rotieren ist günstig!*) → **insgesamt $O(\log n)$** ☺

Red-Black-Trees: Interaktive Visualisierung¹



¹<https://yongdanielliang.github.io/animation/web/RBTree.html>

Balancierte Bäume in der Praxis

```

import java.util.*;

// sets

Set<String> s = new TreeSet<String>();
s.add("Hallo");
s.add("Welt");
s.add("!");
s.add("!");

s.size(); // returns 3

// maps

Map<Integer, String> m = new
    TreeMap<Integer, String>();

m.put(17, "Hallo");
m.put(42, "Welt");
m.put(42, "!");

m.get(42); // returns "!"

for (Map.Entry e : m.entrySet()) {
    print( e.getKey(), e.getValue() );
}

```

Das Java Collection Framework (JCF) enthält balancierte Bäume

- ▶ **Zweck:** Umsetzung von Mengen (sets) und Wörterbüchern (maps/dictionaries)
- ▶ **Interne Datenstruktur:** Red-Black-Trees

Implemen- tierung	Schnittstelle			
	Queue	List	Set	Map
<i>HashTabelle</i>			HashSet	HashMap
<i>Feld</i>		ArrayList		
<i>Baum</i>	PriorityQueue		TreeSet	TreeMap
<i>Liste</i>	LinkedList	LinkedList		



Balancierte Bäume: Benchmarking

Einfügen von **1.000.000 Elementen**

(Mindesthöhe 20) in ...

- ▶ Suchbaum (nicht-optimiert)
- ▶ AVL-Baum (eine andere balancierte Baumstruktur)
- ▶ Red-Black-Tree (selbst implementiert)
- ▶ TreeMap des Java Collection Frameworks

Beobachtungen

- ▶ **Rot-Schwarz-Baum:** stabiles Verhalten, geringe Höhe
- ▶ **nicht-optimierter Suchbaum:** entartet bei sortierten Zahlen
- ▶ **JCF:** vergleichbare Performance zu Rot-Schwarz-Baum
(ist ein Rot-Schwarz-Baum...).

Datenstruktur	1.000.000 Elemente (zufällige Reihenfolge)		1.000.000 Elemente (sortiert)	
	Zeit (sek.)	Baumhöhe	Zeit (sek.)	Baumhöhe
Suchbaum	1,55	50	2236,67	1.000.000
Red-Black-Tree	1,11	25	0,20	27
AVLTree	1,35	24	0,17	20
TreeSet (JCF)	1,15	?	0,21	?

* java 1.7.0.21, OpenJDK 64-Bit Server VM, 23.7-b01, mixed mode, i7 2600, 3.4GHz, 8GB RAM

References I

