

11.1 Suchbäume

11.2 Menge als Suchbaum

11.3 AVL-Bäume

11.4 Rot/Schwarz Bäume

11.5 B-Bäume

11.6 Digitale Bäume



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

Skalierbarkeit und Persistenz

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

Autonome Dienste

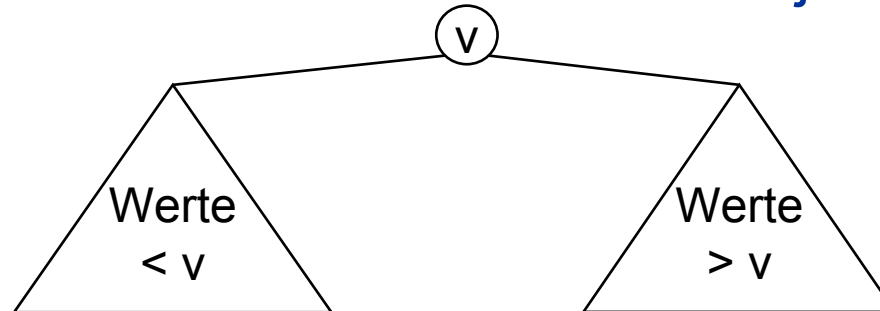
- Enge/lose Kopplung
- Protokolle / Automaten
- Verteilung



Effizientes Suchen

- Grundgedanke:
 - Kombiniere Offenheit der verketteten Liste mit Aufwand $O(\log n)$ des Teile-und-Herrsche-Prinzips.
- Bäume kann man sich als strukturelles Abbild des Teile-und-Herrsche-Prinzips vorstellen, wenn man beim Zugriff jeweils pro Knoten nur einen Sohn weiterverfolgt.
- Eine solche Datenstruktur in Form eines Baums wird **Suchbaum** genannt.

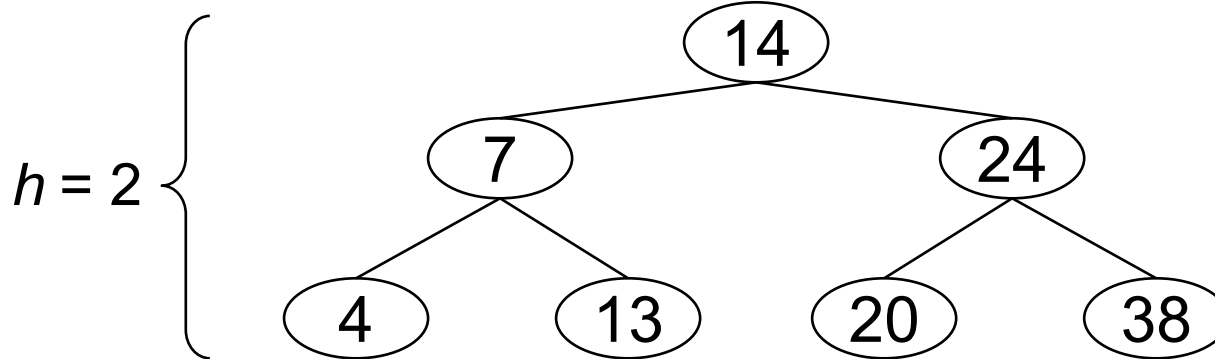
Ein binärer Suchbaum hat max. zwei Söhne je Knoten



Binäre Suche

4	7	13	14	20	24	38
---	---	----	----	----	----	----

Binärer Suchbaum



- Die Höhe h eines Knotens entspricht der Anzahl der Kanten des längsten Pfades zu einem von diesem Knoten aus erreichbaren Blatt.
- Die Höhe h eines Baums ist die Höhe der Wurzel.



Anzahl der Söhne pro Knoten

- **Binärbaum:** Anzahl $0 \leq m \leq 2$
- **Vielwegbaum:** Anzahl beliebig

Speicherung der Nutzdaten

- **Blattbaum/hohler Baum:** Nur in den Blattknoten
- **Natürlicher Baum:** Bei jedem Knoten

Ausgewogenheit/Ausgeglichenheit

- **ausgewogener/balancierter Baum:** Für jeden Knoten unterscheiden sich die Höhen der Unterbäume nur um den Faktor 1.
- **nicht ausgewogener Baum:** Es gibt keine derartige Einschränkung.

Implementierung von Mengen im Hauptspeicher

Implementierung von Mengen auf Hintergrundspeicher



Imperative Implementierung einer Menge im Hauptspeicher Binärer Suchbaum mit Knotenklasse **Entry**

- Speicherung der Mengenelemente in den Baumknoten

```
class Entry {  
    Entry left;        // linker Nachfolgerknoten im Baum  
    Entry right;       // rechter Nachfolgerknoten im Baum  
    Entry parent;      // Vaterknoten  
  
    Object value;      // Mengenelement  
  
    public Entry(Object o, Entry parent) {  
        left = right = null;  
        value = o;  
        this.parent = parent;  
    }  
}
```



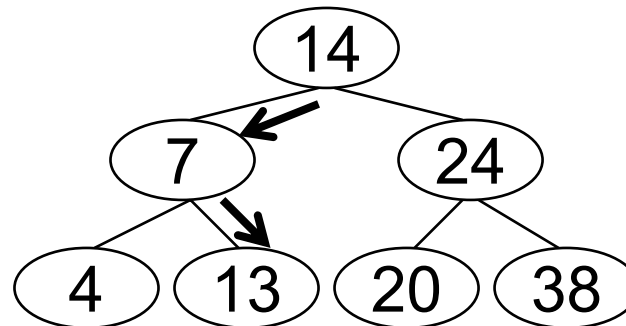
Die eigentliche Funktionalität zur Manipulation der Menge findet sich in der Klasse **TreeSet**, die als Datenstruktur einen Binärbaum verwendet.

```
public class TreeSet implements java.util.Set {  
    // Wurzel des Binärbaums  
    Entry root;  
  
    public TreeSet() {  
        root = null;  
    }  
  
    . . . .
```



Suche

13

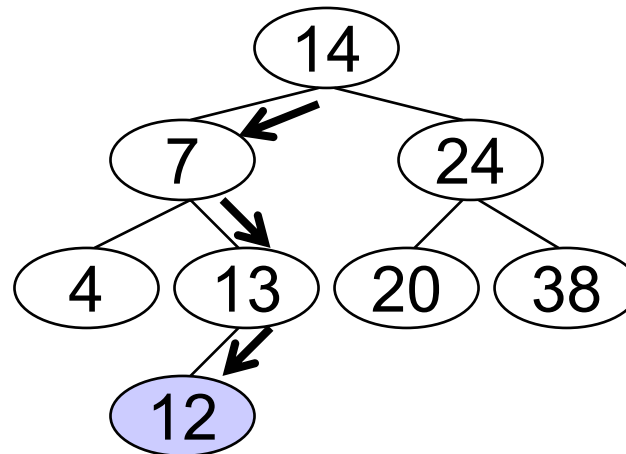


```
public boolean contains(Object o) {  
    return treeContains(o, root);  
}  
  
// Element suchen durch Abstieg im Baum  
private boolean treeContains(Object o, Entry entry) {  
    if (entry == null) return false;  
  
    int result = ((Comparable) entry.value).compareTo(o);  
  
    if (result == 0) {  
        return true;  
    } else if (result > 0) {  
        return treeContains(o, entry.left);  
    } else {  
        return treeContains(o, entry.right);  
    }  
}
```



Grundidee am Beispiel des Einfügens von

12



// Einfügen eines Elements

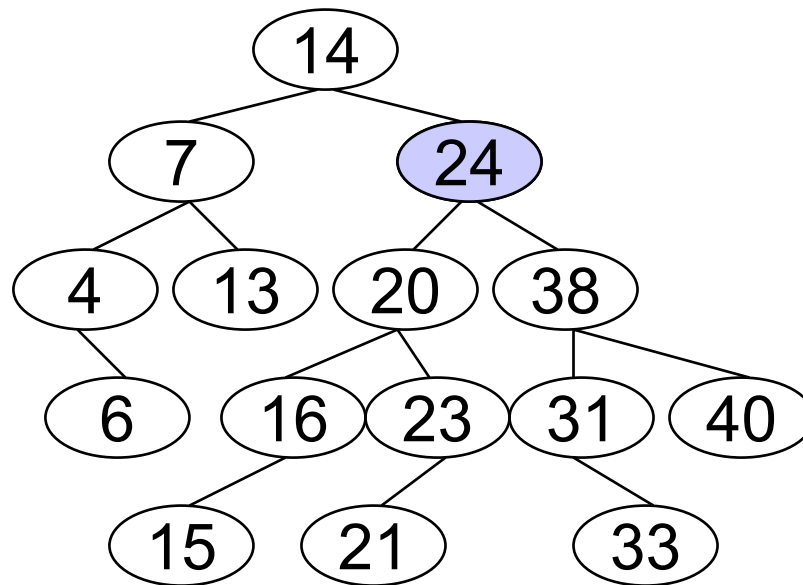
```
public boolean add(Object o) {  
    if (root == null) {  
        root = new Entry(o, null);  
        return true;  
    }  
    return addToTree(o, root);  
}
```



```
private boolean addToTree(Object o, Entry entry){
    int result = ((Comparable) entry.value).compareTo(o);
    if (result == 0) {
        // Element ist schon vorhanden
        return false;
    } else if (result > 0) {
        // Neues Element ist kleiner -> linker Unterbaum
        if (entry.left == null) {
            entry.left = new Entry(o, entry);
            return true;
        }
        return addToTree(o, entry.left); // Abstieg im Baum
    } else {
        // Neues Element ist größer -> rechter Unterbaum
        if (entry.right == null) {
            entry.right = new Entry(o, entry);
            return true;
        }
        return addToTree(o, entry.right); // Abstieg im Baum
    }
}
```

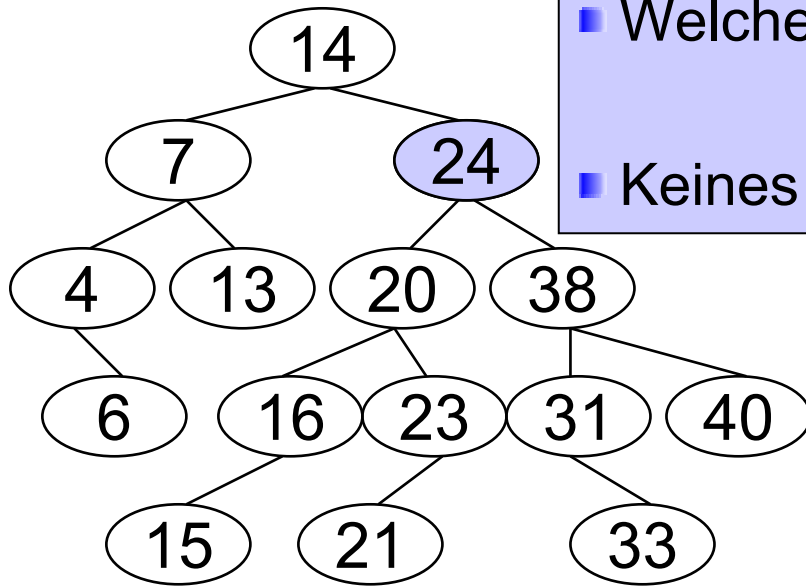


- Löschen von Blattknoten ist einfach.
- Löschen von Knoten mit nur einem Nachfolger auch.
- Aber wie funktioniert das Löschen innerer Knoten?



Beispiel: Lösche
Knoten





■ Welcher Knoten tritt an die Stelle von 24 ?

■ Keines der Blätter 6 15 33 13 21 40 passt!

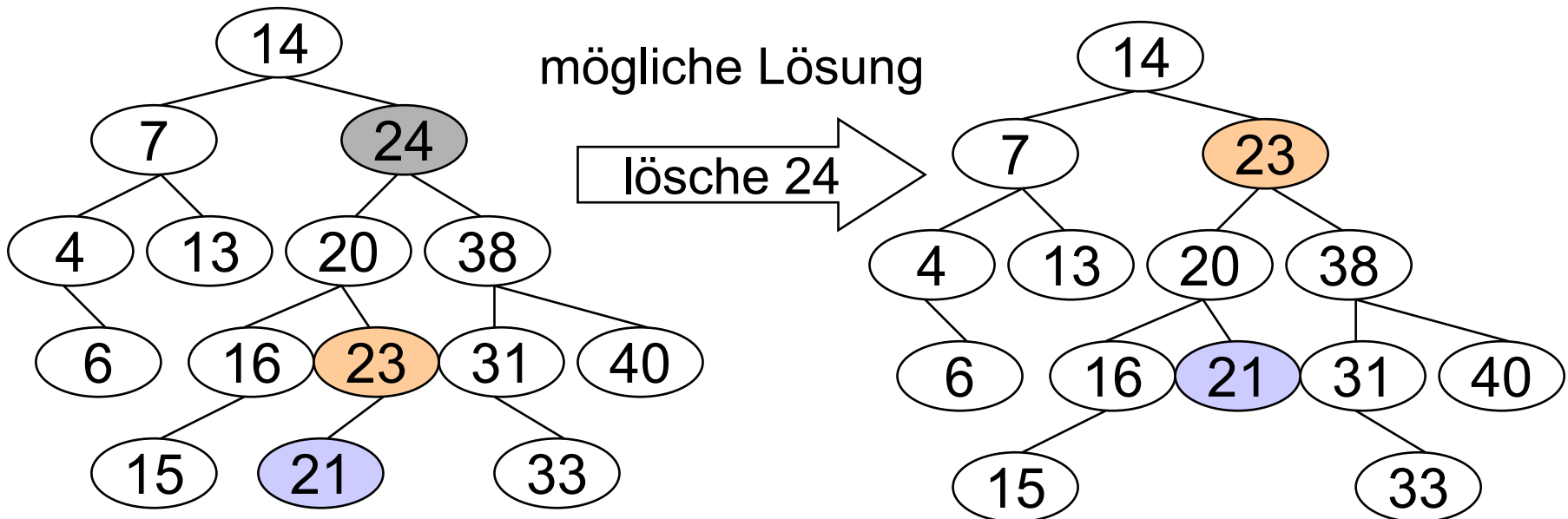
Bedingungen für Ersatz x von Knoten 24

- $x \geq 14$ (da rechter Nachfolger von 14)
- $20 \leq x \leq 38$ (da 20 linker und 38 rechter Nachfolger wird)
- $23 \leq x \leq 31$ (da 23 größter Nachfolger von 20 und 31 kleinster Nachfolger von 38 ist)



Bedingungen für Ersatz von Knoten 24

- $x \geq 14$ (da rechter Nachfolger von 14)
- $20 \leq x \leq 38$ (da 20 linker und 38 rechter Nachfolger wird)
- $23 \leq x \leq 31$ (da 23 größter Nachfolger von 20 und 31 kleinster Nachfolger von 38 ist)

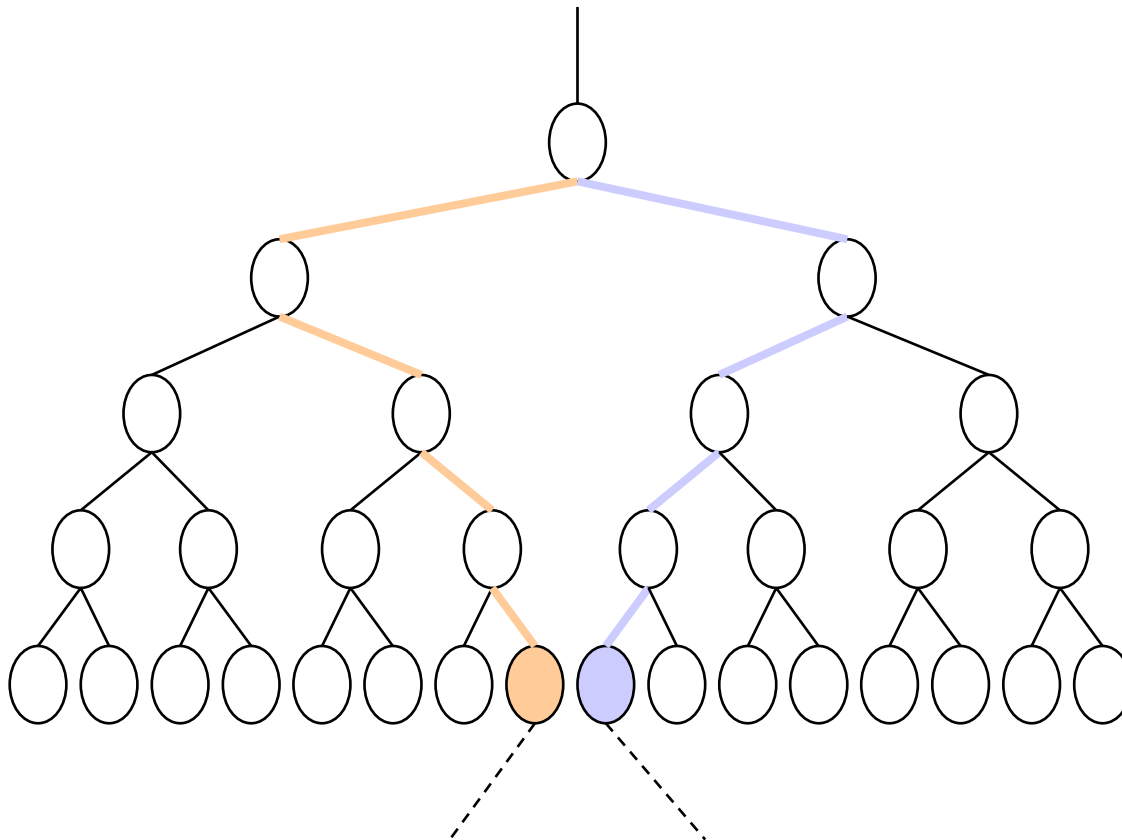


Auch Ersetzen durch die 31 und Aufrücken der 33 wäre eine Lösung.



Allgemein

- Ersetze zu löschenden Knoten durch größten Knoten im linken Teilbaum oder kleinsten Knoten im rechten Teilbaum (der hat max. 1 Nachfolger).



```
// Löschen eines Elements
```

```
public boolean remove(Object o) {
    return removeFromTree(o, root);
}
```

```
private boolean removeFromTree(Object o, Entry entry) {
    if (entry == null) return false;
```

```
    int result = ((Comparable) entry.value).compareTo(o);
```

```
    // Abstieg zum zu löschenden Eintrag:
```

```
    if (result > 0) return removeFromTree(o, entry.left);
    if (result < 0) return removeFromTree(o, entry.right);
```

```
    // Zu löschender Eintrag gefunden.
```

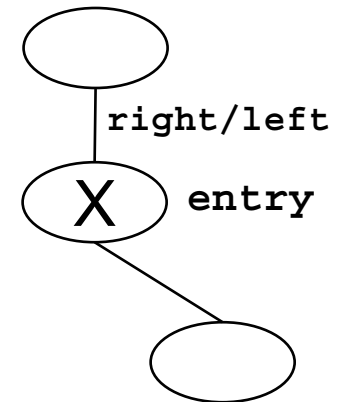
```
    // Falls kein linker Nachfolger vorhanden...
```

```
    if (entry.left == null) {
        // ... zu löschenden Eintrag durch rechten
        // Nachfolger ersetzen.
```

```
    if (entry.right != null) {
        entry.right.parent = entry.parent;
    }
```

```
    // Vater anpassen: right oder left verweist auf entry.
```

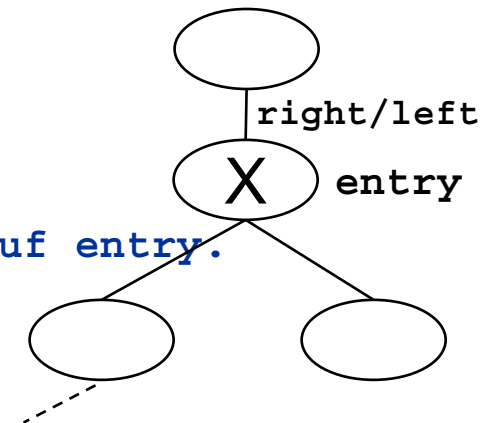
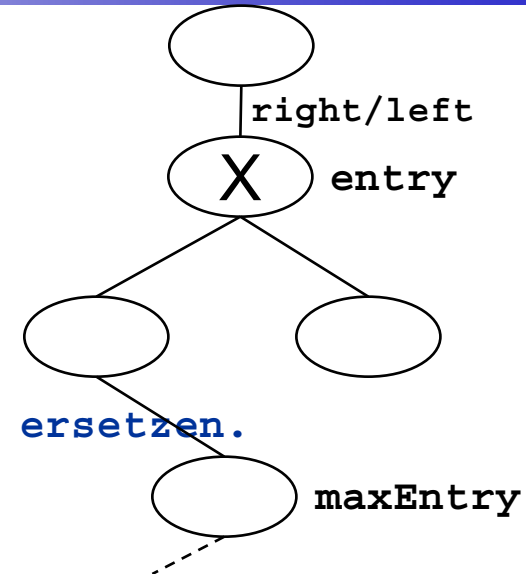
```
    adjustParentEntry(entry, entry.right);
}
```



```

else {
    // Linker Eintrag ist vorhanden.
    Entry maxEntry = entry.left;
    // Falls vorhanden, größten Eintrag im rechten
    // Unterbaum (des linken Eintrags) suchen...
    if (maxEntry.right != null) {
        while (maxEntry.right != null)
            maxEntry = maxEntry.right;
        // ... und zu löschenden Eintrag mit diesem ersetzen.
        maxEntry.parent.right = null;
        entry.value = maxEntry.value;
    } else {
        // Sonst: linker Eintrag ersetzt zu löschenden Eintrag.
        maxEntry.parent = entry.parent;
        maxEntry.right = entry.right;
        if (entry.right != null) {
            entry.right.parent = maxEntry;
        }
        // Vater anpassen: right oder left verweist auf entry.
        adjustParentEntry(entry, maxEntry);
    }
}
return true;
}

```

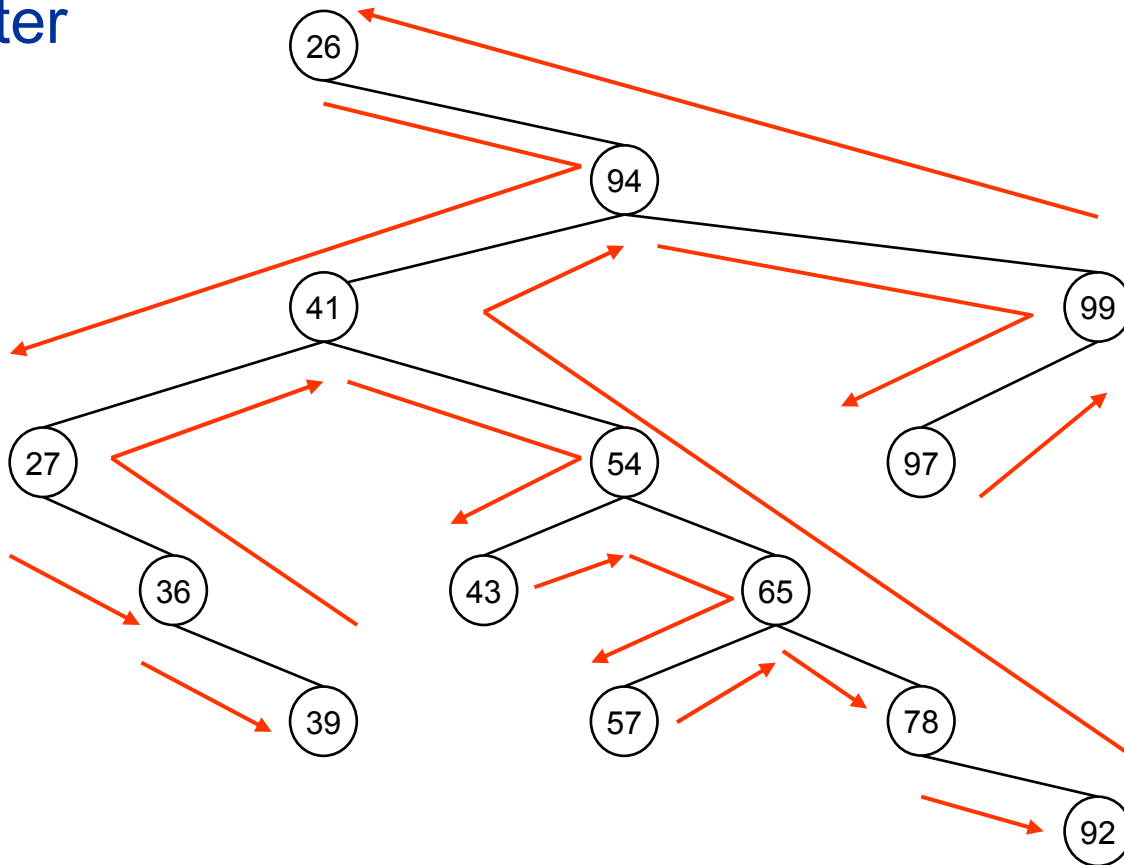


```
// Vatereintrag von zu löschendem Eintrag anpassen.  
private void adjustParentEntry (Entry entry,  
                                Entry replacingEntry){  
    if (entry == root) {  
        root = replacingEntry;  
    } else if (entry.parent.left == entry) {  
        entry.parent.left = replacingEntry;  
    } else {  
        entry.parent.right = replacingEntry;  
    }  
}  
  
// Die weiteren java.util.Set-Methoden wurden hier  
// (der Einfachheit halber) weggelassen...
```



Iterator zur Baumtraversierung in Sortierreihenfolge

Je Knoten: links absteigen - Wert ausgeben - rechts absteigen -
zum Vater



26 27 36 39 41 43 54 57 65 78 92 94 97 99



```
class TreeSetIterator implements java.util.Iterator {
    // Zu traversierender Baum
    private TreeSet treeSet;
    // Aktuelle Position für Baumdurchlauf
    private Entry position;

    public TreeSetIterator(TreeSet treeSet) {
        this.treeSet = (TreeSet)treeSet.clone();
        // Erste Position ist das am weitesten
        // links liegende Blatt.
        position = treeSet.root;
        if (position != null) {
            while (position.left != null) {
                position = position.left;
            }
        }
    }

    public boolean hasNext() {
        return position != null;
    }
}
```



```
public Object next() {  
    // Merke zu liefernden Wert von aktueller Position.  
    Object result = position.value;  
    // Berechne nächste Position: Abstieg nach rechts nicht möglich?  
    if (position.right == null) {  
        // Dann Aufstieg um 1 und weiter solange man von "rechts unten  
        // kommt"  
        // (also auf dem Rückweg der Traversierung ist...)  
        if (position == treeSet.root) position = null;  
        else {  
            Entry previousPosition = position;  
            position = position.parent; // Aufstieg um eins  
            while (position.right==previousPosition &&  
                    position!=treeSet.root) {  
                previousPosition = position;  
                position = position.parent;  
            }  
            if (position.right==previousPosition &&  
                    position==treeSet.root) {  
                position = null;  
            }  
        }  
    } else {  
        // Abstieg nach rechts und dort zum kleinsten Element -  
        // am weitesten links im Baum.  
        position = position.right;  
        while (position.left != null) position = position.left;  
    }  
    return result;  
}
```



Suchen: Bei Ausgewogenheit spiegelt der Suchbaum die Binärsuche strukturell wider (Materialisierung der Binärsuche).

- Suchen in $O(\text{Höhe}) = O(\log n)$ bei ausgewogenen Bäumen.

Einfügen: **add()** entspricht einem Einfügeschritt in einer Sortierprozedur mit $O(\log n)$. Wiederholtes **add()** ist daher effektiv eine Sortierprozedur mit einem Aufwand von $O(n \log n)$ bei n Elementen (Baumsortieren).

- Einfügen in $O(\text{Höhe}) = O(\log n)$ bei ausgewogenen Bäumen.

Löschen:

- Löschen in $O(\text{Höhe}) = O(\log n)$ bei ausgewogenen Bäumen.

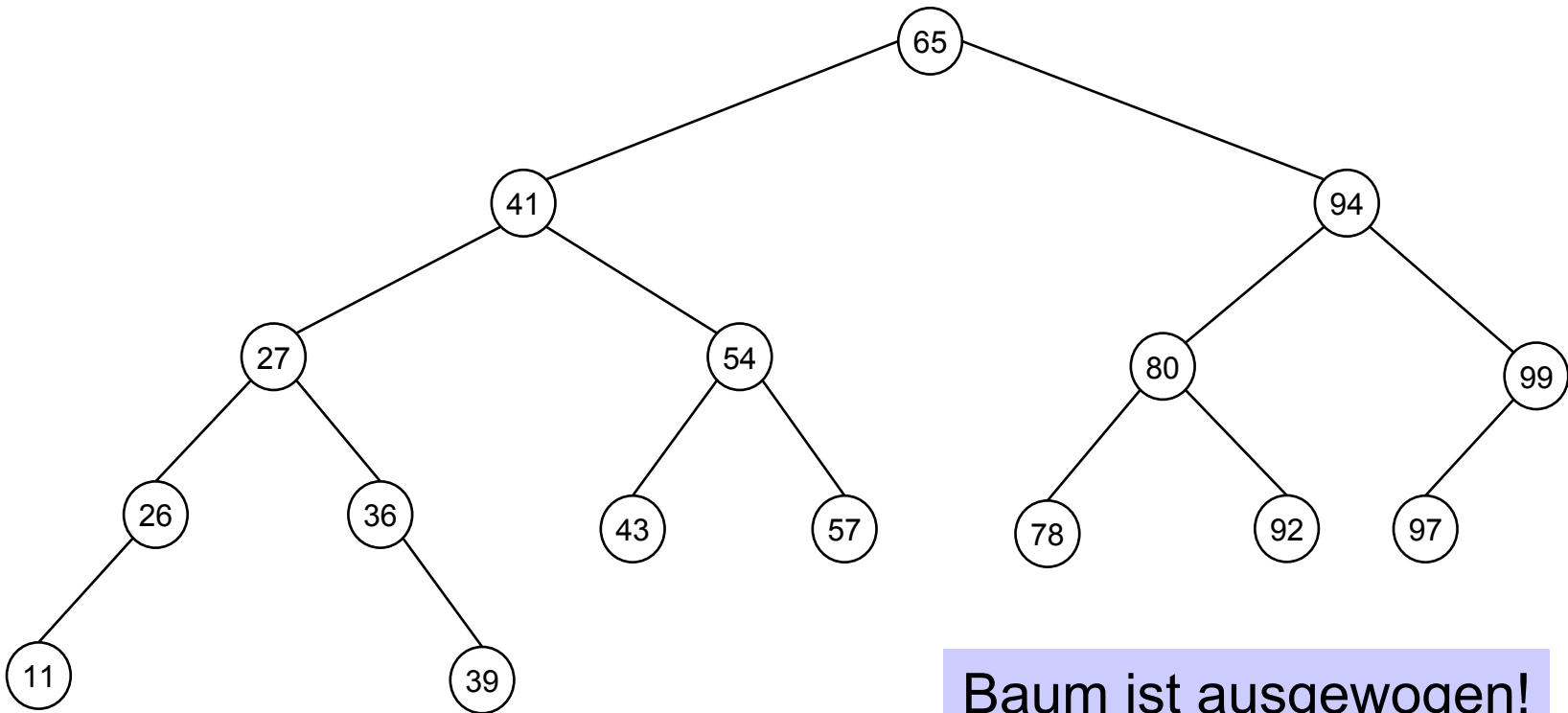
Lesen in Sortierreihenfolge: **next()** leistet dies bei der Traversierung.



Ist ein Baum immer ausgewogen?

Beispiel: Eingabefolge

65, 41, 54, 94, 57, 99, 43, 27, 97, 26, 80, 92, 11, 36, 39, 78

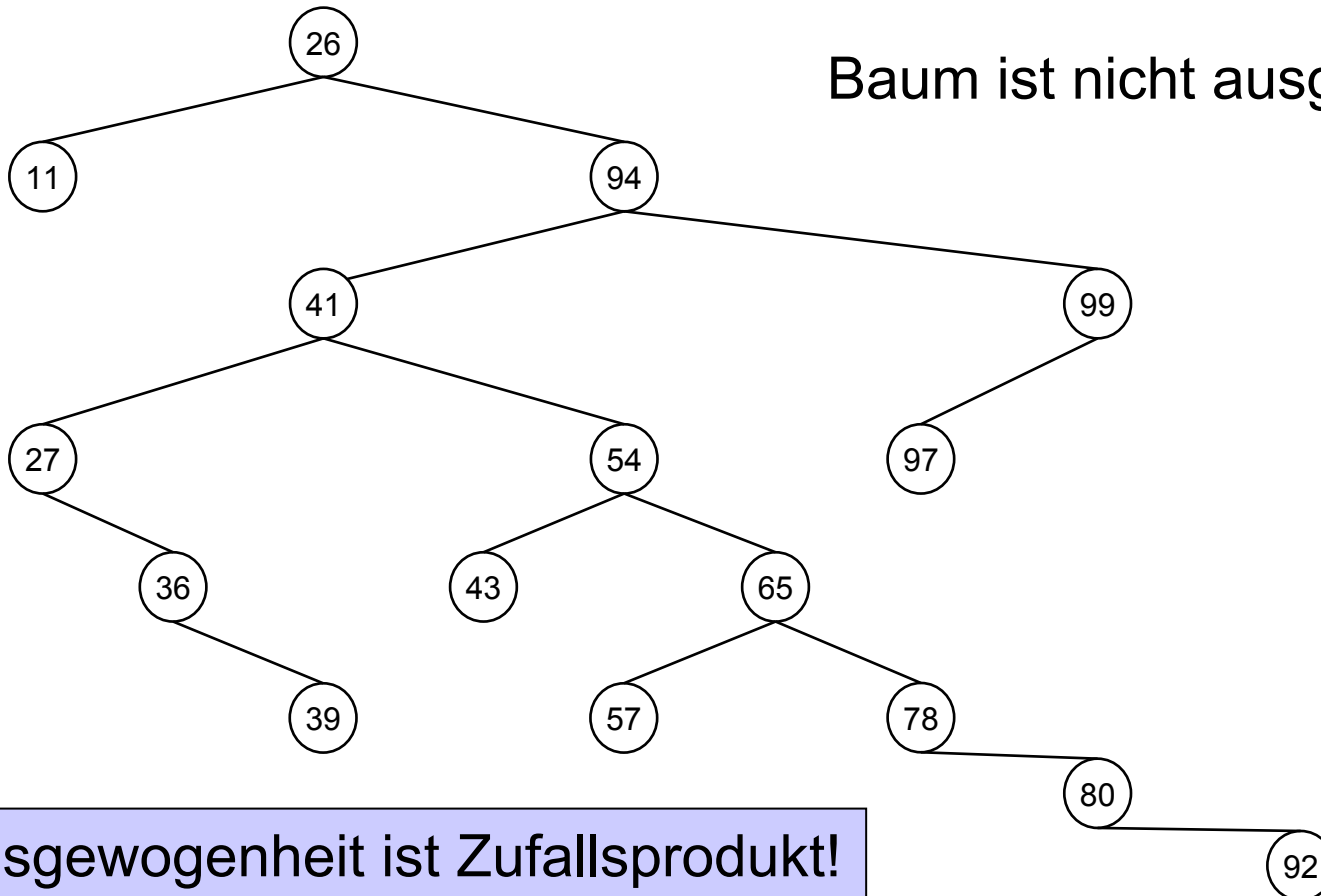


Baum ist ausgewogen!



Beispiel: Andere Reihenfolge der Eingabe

26, 94, 41, 54, 99, 65, 57, 78, 80, 27, 97, 36, 11, 43, 39, 92

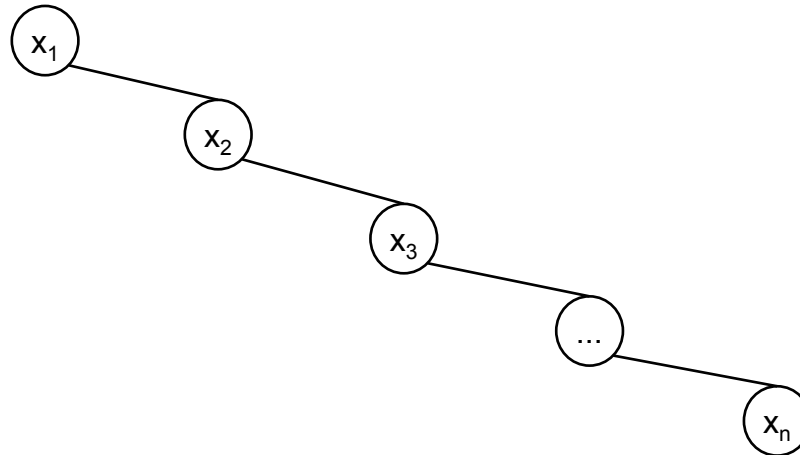


Merke: Ausgewogenheit ist Zufallsprodukt!



Bisher betrachtete binäre Suchbäume sind nicht balanciert:

- z.B. Einfügen einer sortierten Folge $x_1, x_2, x_3, \dots, x_n$ liefert:



Aufwand für die Änderungsoperationen steigt also von $O(\log n)$ auf $O(n)$ im schlechtesten Fall!



Ziel: Herstellen von Ausgewogenheit

- Reorganisation: Lies alle Elemente, ordne sie so, dass die neue Folge beim Einfügen Ausgewogenheit liefert, und baue den Baum neu auf.
⇒ aufwändig
- Besser: Ausgewogenheit schon bei jedem Einfügen eines Elements durch lokalen Umbau des Baums wiederherstellen.



AVL-Bäume

- Ausgewogenheit wird eine inhärente Eigenschaft der Datenstruktur.
- AVL-Bedingung:
Sei e beliebiger Knoten eines binären Suchbaumes, und $h(e)$ die Höhe des Unterbaums mit Wurzel e , dann gilt für die beide Söhne $e.links$ und $e.rechts$ von e :

$$| h(e.links) - h(e.rechts) | \leq 1$$

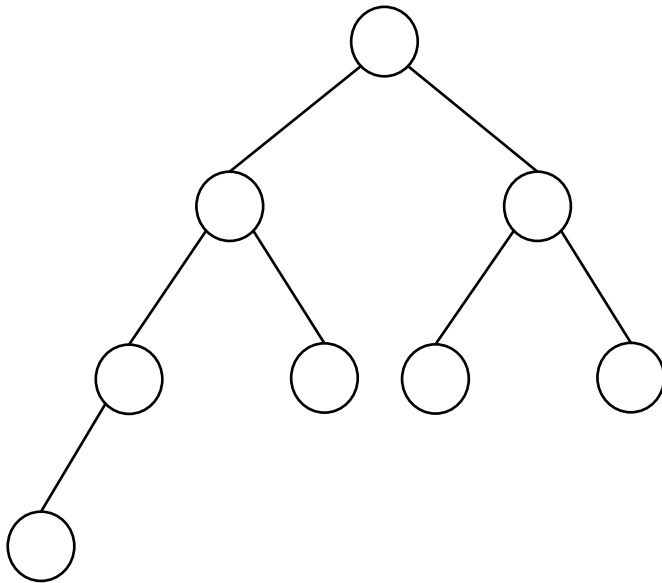
Ein binärer Suchbaum mit dieser Eigenschaft heisst AVL-Baum (nach den beiden russischen Mathematikern Adel'son-Vel'skii und Landis, 1962).

Anmerkung

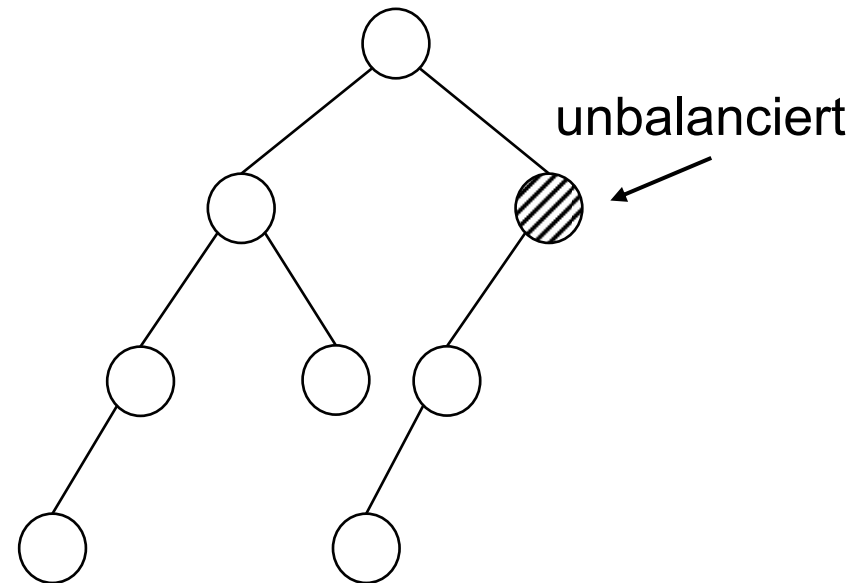
- Es genügt nicht, wenn die obige Bedingung nur für die Wurzel gilt.



Erfüllung der AVL-Bedingung:



Nichterfüllung der AVL-Bedingung:



Änderungsoperationen in AVL-Bäumen

- Suchen, Einfügen, Löschen von Elementen funktioniert im Prinzip wie bei unbalancierten Bäumen.
- Zusätzlich schließt sich an jedes Einfügen, Löschen ein Ausbalancieren an, falls die AVL-Bedingung jetzt verletzt wird.
- Das Überprüfen der Ausgewogenheit hat in einem Baum mit n Knoten den Aufwand $O(n)$.

Folgende Fälle können zu einer Verletzung der AVL-Eigenschaften führen

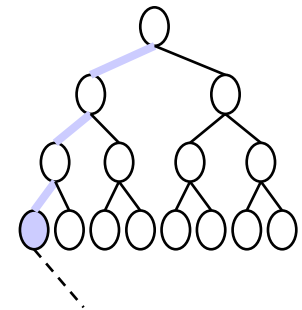
- Einfügen in linken Teilbaum des linken Kindes
 - Einfügen in rechten Teilbaum des linken Kindes
 - Einfügen in rechten Teilbaum des rechten Kindes
 - Einfügen in linken Teilbaum des rechten Kindes
-
- Die Fälle 1 und 3 sowie 2 und 4 sind symmetrisch



Optimierung

- Jeder Knoten merkt sich seinen Balancefaktor bf :

$$bf = h(\text{linker Unterbaum}) - h(\text{rechter Unterbaum})$$
- Bei jeder Änderung genügt es, die Balancefaktoren auf einem Baumast (in $O(\log n)$) zu aktualisieren.



Änderungsoperationen in AVL-Bäumen

- Balancefaktor **bf** als zusätzliches Knotenattribut der Klasse **Entry**

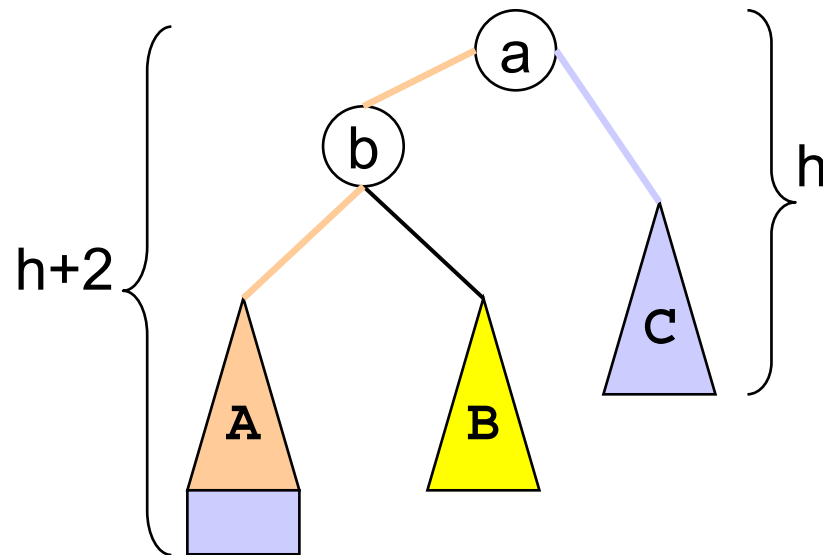
Anpassen der Methoden **add()** und **remove()** bei **TreeSet**

- Führe Methode wie zuvor aus; es entsteht ein neues Blatt, für das die AVL-Bedingung trivial erfüllt ist.
- Gehe anschließend vom eingefügten Knoten zur Wurzel, aktualisiere je Knoten **e** den Balancefaktor...
 - Neues Blatt kann, muss aber nicht die Höhe des Vorgängers um 1 erhöht haben.
 - Falls Erhöhung im linken Unterbaum:
$$e.bf = e.bf + 1$$
 - Falls Erhöhung im rechten Unterbaum:
$$e.bf = e.bf - 1$$
- ...und prüfe bei jedem Knoten die Einhaltung der AVL-Bedingung:
 - Gilt nun für den Balancefaktor $|e.bf| \geq 2$, so muss der Baum ausbalanciert werden.




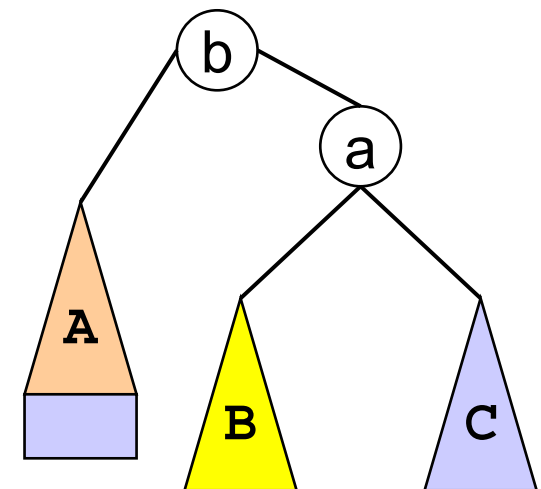
Ausbalancieren nach Einfügen geht durch „Rotation“:


 = Bäume identischer Höhe



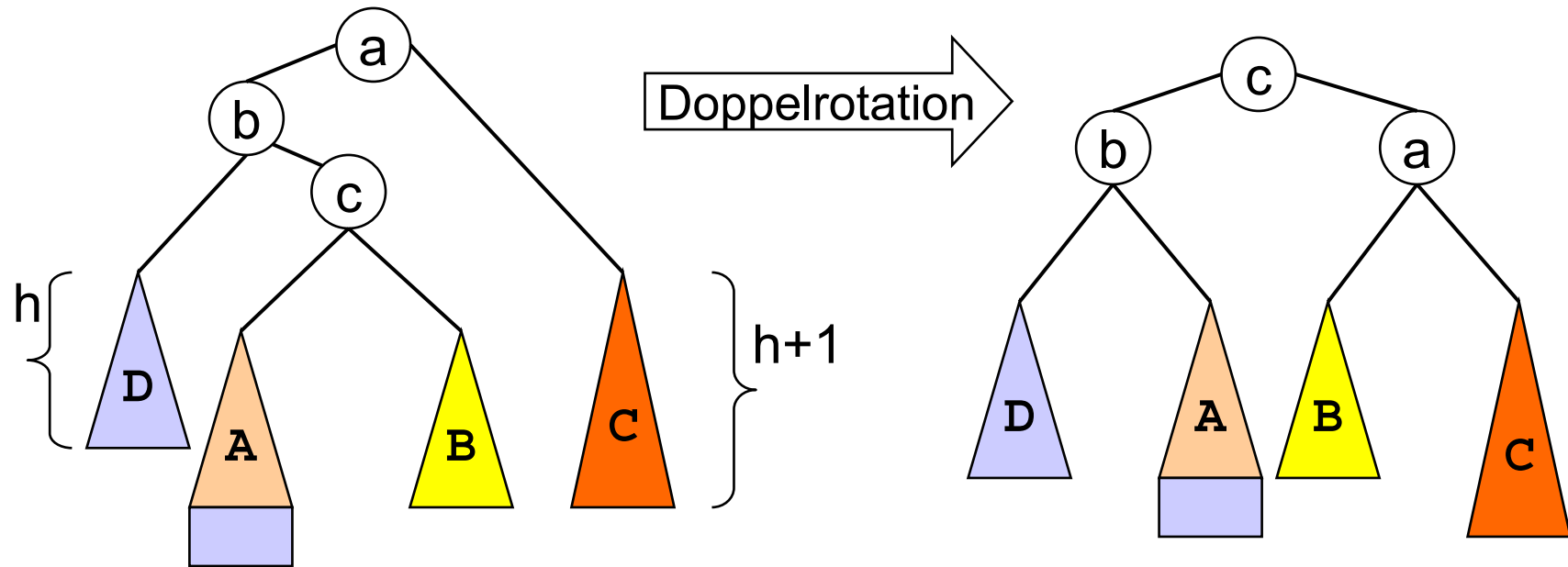
Durch Einfügen ist der **rote** Ast um 2 höher als der **blau**farbige Ast geworden

Rotation 



Umhängen der Teilbäume geht in $O(1)$
(der Haken von  wird an *a* gehängt und *b* zur neuen Wurzel)

Ausbalancieren benötigt manchmal auch „Doppelrotation“:



Einfügen im linken Teilbaum des linken Kindes

- Rotation mit linkem Kind

Einfügen in rechten Teilbaum des linken Kindes

- Doppelrotation mit linkem Kind

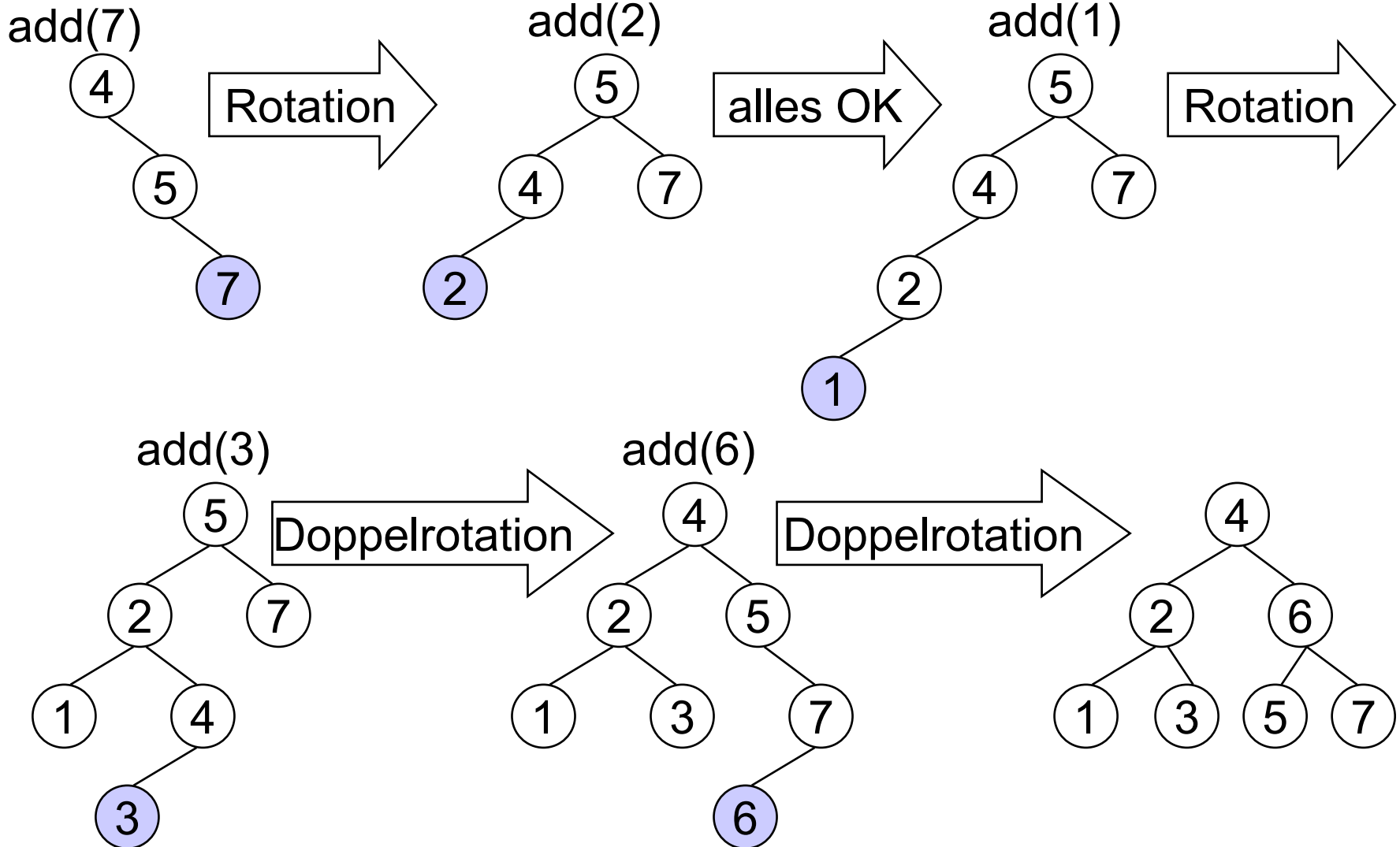
Einfügen in rechten Teilbaum des rechten Kindes

- Rotation mit rechtem Kind

Einfügen in linken Teilbaum des rechten Kindes

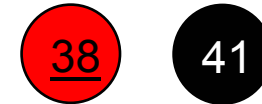
- Doppelrotation mit rechtem Kind



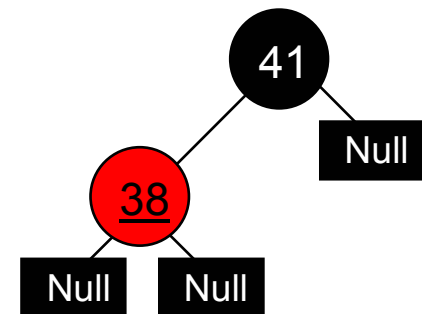


Allgemeines:

- Binärer Suchbaum
- Zusatzbit in jedem Knoten zur Speicherung seiner Farbe
 - 0=Schwarz
 - 1=Rot

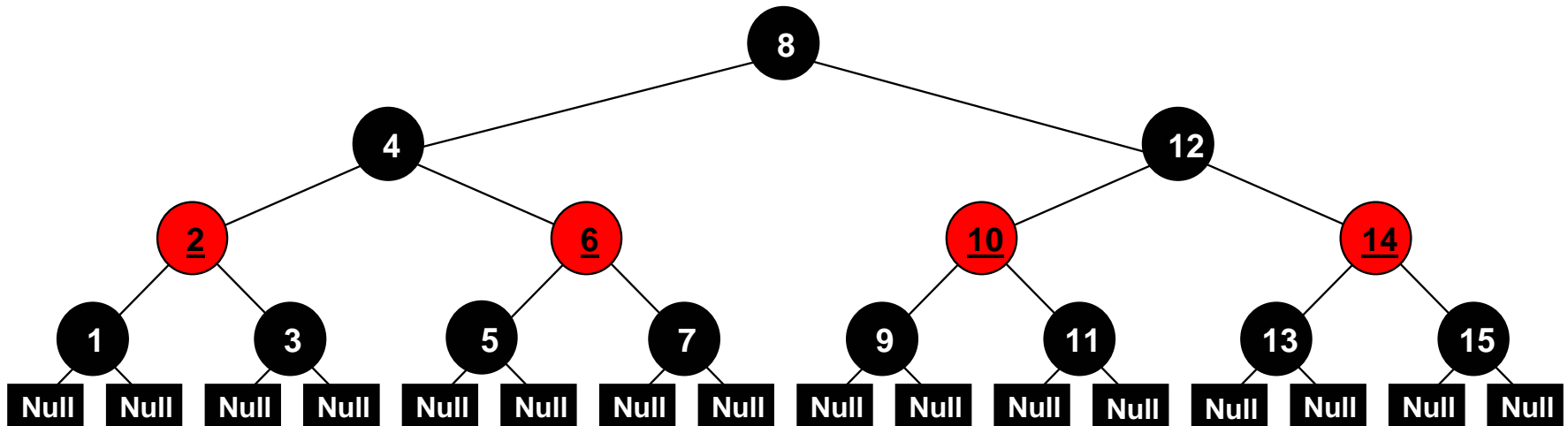


- Jeder Pfad von der Wurzel zu einem Blatt ist maximal doppelt so lang wie jeder andere
- Nicht vorhandener Kind-Knoten
 - speziellen Null-Knoten als Kind einfügen, dessen Wert das Null-Objekt repräsentiert
 - Kann wie normaler Knoten behandelt werden
 - bislang durch Referenz der Beziehung right/left auf Null repräsentiert



- (1) Jeder Knoten ist entweder rot oder schwarz.
- (2) Die Wurzel ist schwarz.
- (3) Jedes Blatt (Null) ist schwarz.
- (4) Wenn ein Knoten rot ist, so sind beide Kinder schwarz.
- (5) Für jeden Knoten gilt, dass alle Pfade vom Knoten zu einem Blatt die selbe Anzahl schwarzer Knoten beinhalten.

Beispiel:



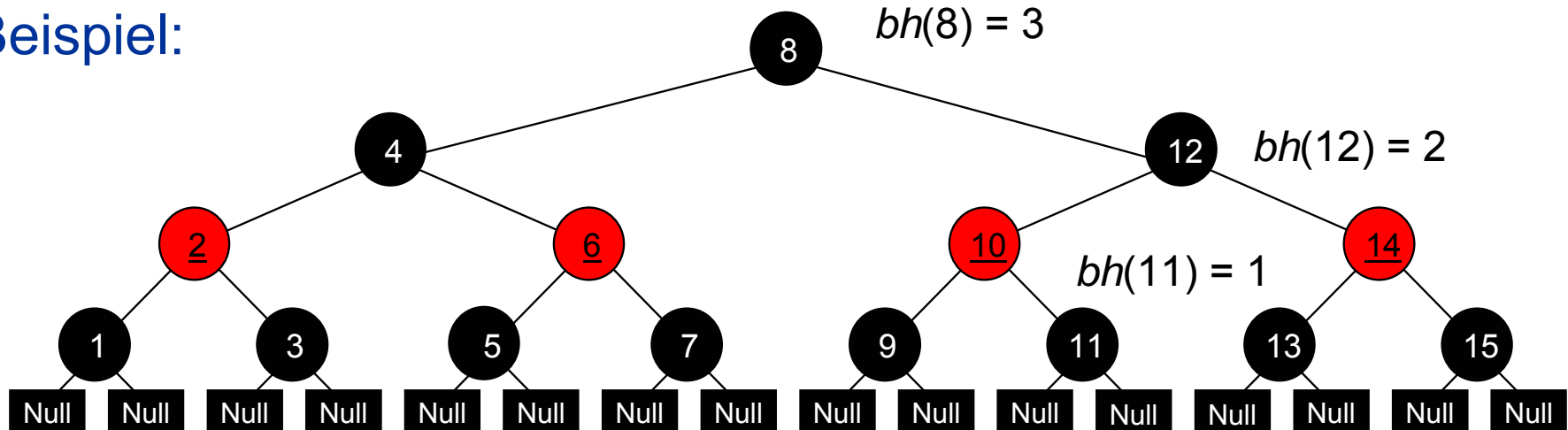
Definition:

- „Black-Height“ eines Knotens k : $bh(k)$
 - Anzahl der schwarzen Knoten auf beliebigen Pfaden von k zu einem Blatt. Hierbei wird k selbst nicht mitgezählt.
- Black-Height eines Baums
 - Black-Height der Wurzel w : $bh(w)$

Anmerkung:

- Die black-height-Eigenschaft ist aufgrund der Eigenschaft (5) wohl definiert.

Beispiel:



Satz:

- Die Höhe eines Rot/Schwarz-Baums mit n inneren Knoten beträgt maximal $2 \log(n+1)$.

Beweis:

- Jeder durch einen Knoten x aufgespannte Teilbaum hat wenigstens $2^{bh(x)} - 1$ innere Knoten.
- Vollständige Induktion über die Höhe h von x .
 - Induktionsanfang
 $h = 0 \Rightarrow x$ ist ein Blatt (=Null-Knoten) und der durch x aufgespannte Teilbaum enthält $2^{bh(x)} - 1 = 2^0 - 1 = 0$ innere Knoten.
 - Induktionsannahme
Ein durch x aufgespannter Teilbaum der Höhe h hat wenigstens $2^{bh(x)} - 1$ innere Knoten.



- Induktionsschritt

Ein durch x aufgespannter Teilbaum der Höhe $h+1$ hat wenigstens $2^{bh(x)} - 1$ innere Knoten.

- Knoten x sei innerer Knoten und habe die Höhe $h+1$ und zwei Kinder.

⇒ Jedes Kind hat die black-height $bh(x)$ oder $bh(x) - 1$ abhängig davon, ob seine Farbe Rot oder Schwarz ist.

⇒ (Induktionsannahme) Jedes Kind von x hat mindestens $2^{bh(x)-1} - 1$ innere Knoten.

⇒ Der durch x aufgespannte Teilbaum hat mindestens $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ innere Knoten. (q.e.d.)



Bisher bewiesen:

- Ein durch x aufgespannter Teilbaum der Höhe $h+1$ hat wenigstens $2^{bh(x)} - 1$ innere Knoten.

Sei nun h die Höhe des Baums. Entsprechend Eigenschaft (4) müssen mindestens die Hälfte aller Knoten auf allen Pfaden von der Wurzel zu einem Blatt (ausschließlich der Wurzel) schwarz gefärbt sein.

- Die Black-Height der Wurzel muss mindestens $h/2$ betragen.

$$\Rightarrow n \geq 2^{h/2} - 1$$

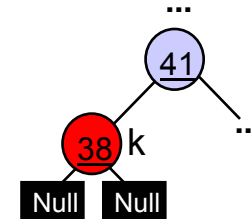
$$\Rightarrow n+1 \geq 2^{h/2}$$

$$\Rightarrow \log(n+1) \geq h/2 \quad \text{oder} \quad h \leq 2 \log(n+1)$$



Vorgehensweise

- Rot-färben des eingefügten Elements k
- Ergänzung zweier Null-Knoten als Kinder



Überprüfen der Eigenschaften

- (1) Jeder Knoten ist entweder rot oder schwarz. \Rightarrow Nicht verletzt.
- (2) Die Wurzel ist schwarz.
 - Verletzt nur, wenn k in leeren Baum eingefügt wird \Rightarrow k Schwarz färben.
- (3) Jedes Blatt (Null) ist schwarz. \Rightarrow Nicht verletzt.
- (4) Wenn ein Knoten rot ist, so sind beide Kinder schwarz.
 - Nicht durch k verletzt, da beide Kinder schwarze Null-Knoten.
 - Verletzt, falls k als Kind eines roten Vater-Knotens eingefügt wird.
- (5) Für jeden Knoten gilt, dass alle Pfade vom Knoten zu einem Blatt die selbe Anzahl schwarzer Knoten beinhalten.
 - Da nur ein roter Knoten hinzukommt, ist diese Eigenschaft ebenfalls nicht verletzt.



Maßnahmen, um Eigenschaft (4) wieder herzustellen

- Geeignete Links/Rechts/Doppel-Rotation zwecks Höhenausgleich.
 - Einfärbung der Knoten gibt Aufschluss über die notwendigen Rotationen.
- Korrektur der Einfärbung der falsch eingefärbten Knoten.

Einfärbung wird in Richtung der Wurzel korrigiert

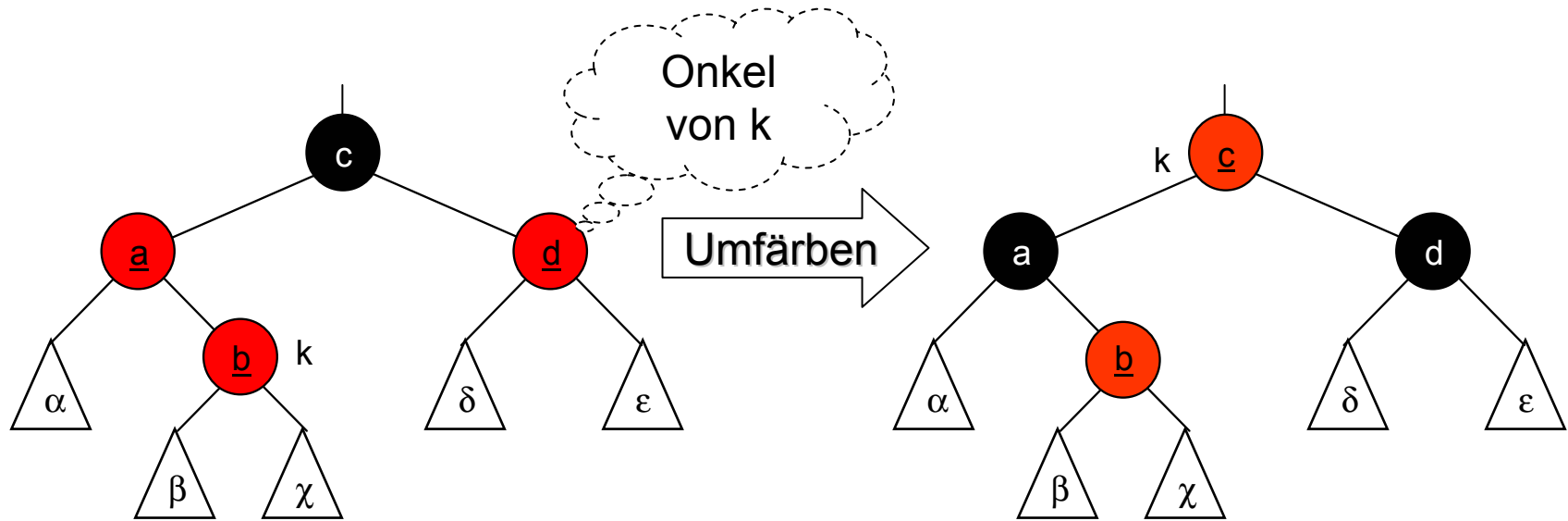
- Funktion zum Zugriff auf den Vaterknoten eines Knotens k nötig:
 $\text{parent}(k)$

Sechs Fälle sind zu unterscheiden

- $\text{parent}(k)$ ist linkes Kind von $\text{parent}(\text{parent}(k))$.
 - (E1) Der „Onkel“ von k ist rot.
 - (E2) Der „Onkel“ von k ist schwarz und k ist rechtes Kind.
 - (E3) Der „Onkel“ von k ist schwarz und k ist linkes Kind.
- $\text{parent}(k)$ ist rechtes Kind von $\text{parent}(\text{parent}(k))$.
3 analoge Fälle



Fall (E1): Der „Onkel“ von k ist rot.

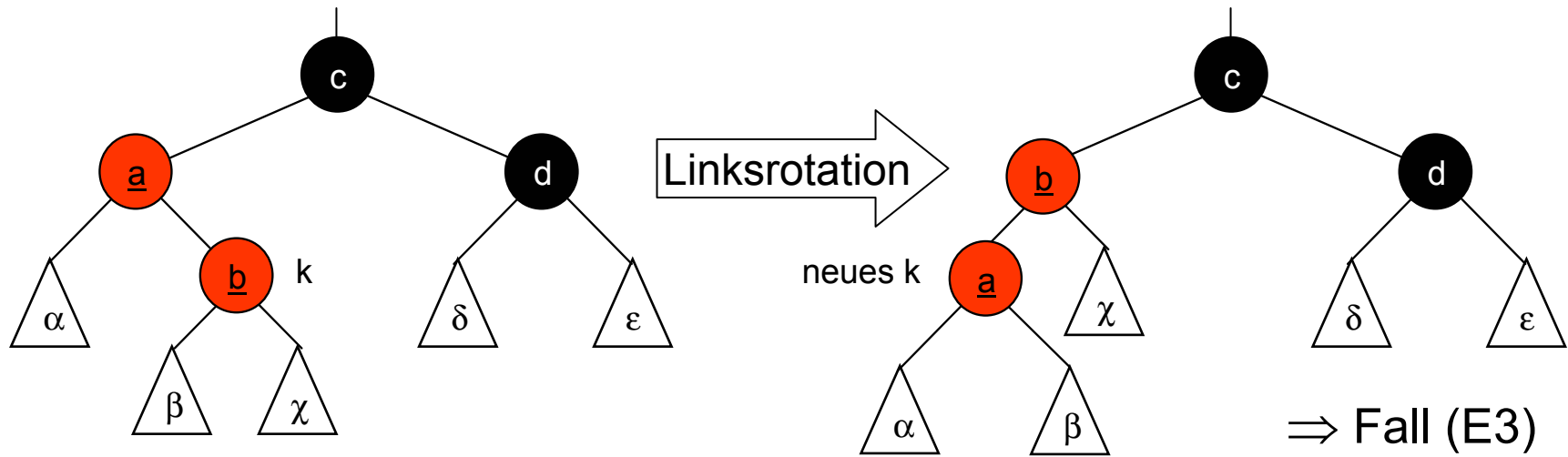


■ Maßnahmen

- Parent(k) und Onkel von k schwarz färben
 - Eigenschaft (4) erfüllt, Eigenschaft (5) beibehalten
- Parent(parent(k)) rot färben
 - Parent(parent(k)) kann ebenfalls wieder Kind eines roten Knoten sein (erneute Verletzung von Eigenschaft (4)).
 - Rekursive Wiederherstellung der Eigenschaft (4)
 - Aufsteigen um 2 Ebenen $k = \text{parent}(\text{parent}(k))$
 - Falls Wurzel: schwarz färben, fertig.



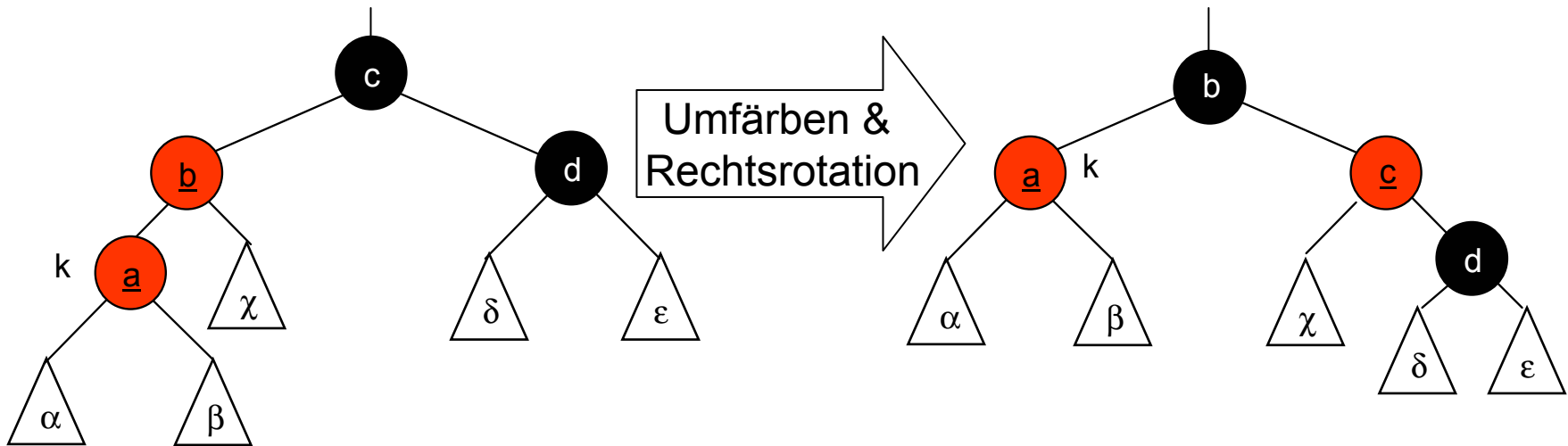
Fall (E2): Der „Onkel“ von k ist schwarz und k ist rechtes Kind.



- Durch Linksrotation der Knoten k und parent(k) entsteht Fall (E3) in Bezug auf den ehemaligen Vater-Knoten von k
 - Der „Onkel“ von k ist schwarz und k ist linkes Kind.
 - Keine Veränderungen hinsichtlich der Eigenschaften und der bh
- ⇒ Weiter wie bei Fall (E3) beschrieben.



Fall (E3): Der „Onkel“ von k ist schwarz und k ist linkes Kind.



■ Vorgehen

- Rechtsrotation und Rot-färbung des Knoten c
- Schwarz-färbung von Knoten b
- Knoten c hatte links und rechts gleiche Anzahl schwarzer Knoten
 - Da b vormals rot war, wird die Anzahl nach Übernahme des rechten Teilbaums von b als linken Teilbaum dies nicht ändern.
- Knoten b bekommt rechts schwarzen Knoten d hinzu
 - Da Anzahl der schwarzen Knoten der von b über a oder c laufenden Pfade aber gleich der von d (inklusive) ausgehenden Pfade war, bleibt Eigenschaft (5) gültig.



Analog zum Einfügen!

- Löschen eines Elements mit üblichem Algorithmus für Löschen im binären Suchbaum.

Analyse der Folgen für die Eigenschaften des Baums

- Entfernter Knoten hat die Farbe Rot
 - Keine Verletzung der Eigenschaften
- Entfernter Knoten k hat die Farbe Schwarz
 - Verletzung der Eigenschaft (5)
 - Außer der Baum bestand nur aus k
- Entfernen der Wurzel des Baums und Nachrücken eines roten Knotens
 - Verletzung der Eigenschaft (2)
- Vaterknoten von k hat die Farbe Rot und sein neuer Stellvertreter (Kind oder Vorgänger/Nachfolger aus dem linken/rechten Teilbaum) ebenfalls
 - Verletzung der Eigenschaft (4)



- Eigenschaften des Baums sind beginnend mit dem Kind x des entfernten Knotens k wieder herzustellen
 - durch Rotieren und
 - Umfärben der Knoten.

- Eigenschaften des Rot/Schwarz-Baums evtl. rekursiv/iterativ bis zur Wurzel wieder herstellen.



Zum Algorithmus:

- Sollte die neue Wurzel die Farbe rot haben, diese Schwarz färben
 - Eigenschaft (2) ist wieder hergestellt.
- Entfernen des schwarzen Knotens führt zu Unterzahl schwarzer Knoten auf dem Pfad von der Wurzel über x.
 - Beginnend von x wird ein erster auf dem Pfad hin zur Wurzel liegender roter Knoten schwarz gefärbt. (Eigenschaft (4))
 - Ggf. werden Rotationen/Umfärbungen hierbei nötig.



Ausgangspunkt

- Knoten x trage virtuell eine „doppelte“ Schwarz-Färbung, die nun in Richtung Wurzel auszugleichen ist.
- Verletzt Eigenschaft (1)

Zwei Fälle

- x ist linkes Kind von $\text{parent}(x)$
- x ist rechtes Kind von $\text{parent}(x)$
- Fälle in ihrer Behandlung symmetrisch. Im Folgenden linkes Kind.

Sei w der rechte Bruder von x .

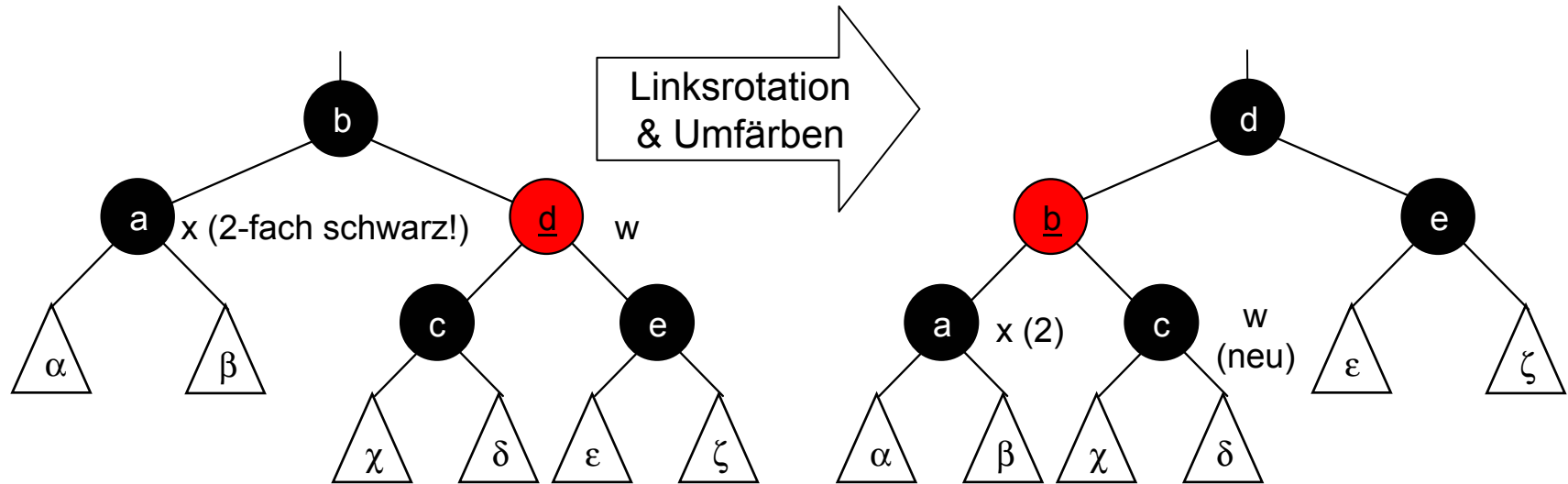
- Da x die ehemalige Überzahl von schwarz gefärbten Knoten trägt, kann w nicht ein Null-Knoten sein.

Vier Fälle lassen sich unterscheiden

- (L1) w ist rot
- (L2) w und beide Kinder von w sind schwarz
- (L3) w und dessen rechtes Kind sind schwarz, linkes Kind ist rot
- (L4) w ist schwarz und rechtes Kind ist rot.



Fall (L1): w ist rot



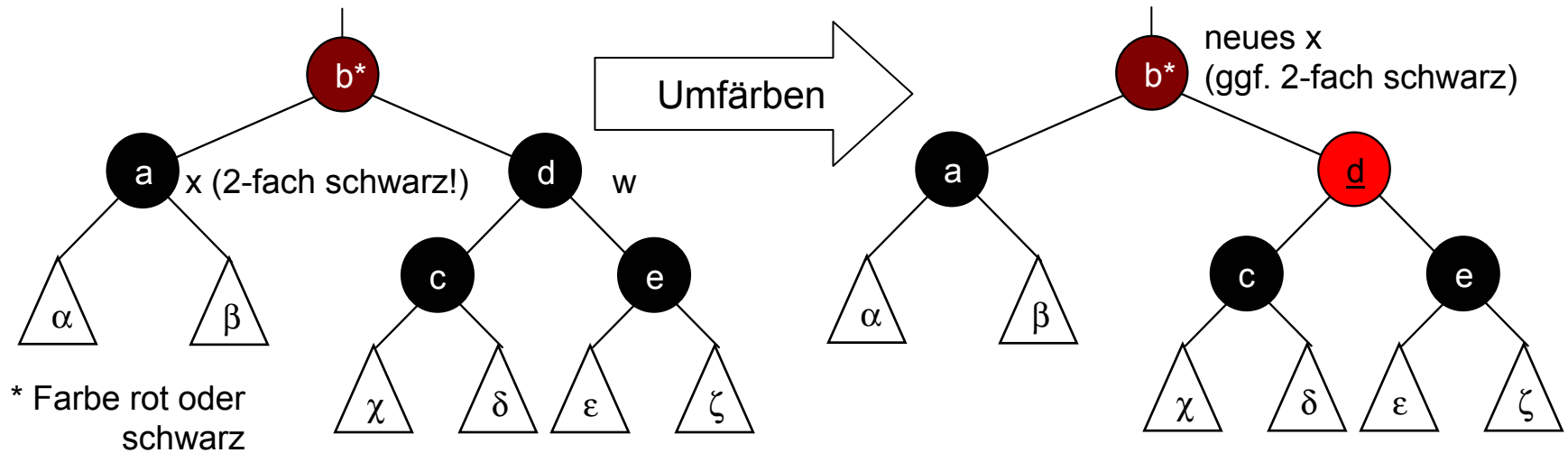
⇒ Fall (L2),(L3) oder (L4)

■ Vorgehen

- Umfärben w und parent(x)
- Linksrotation um parent(x)
- Durch Linksrotation und Umfärben wird einer der Fälle (L2), (L3) oder (L4) generiert.
- Neuer Bruder von x ist schwarz
- Unverändert bleibt die Anzahl schwarzer Knoten auf Pfaden von der Wurzel zu den Teilbäumen a, b (Anzahl = 3 inkl. dem doppelt gezählten Knoten x), c, d, e oder z (Anzahl = 2).



Fall (L2): w und beide Kinder von w sind schwarz

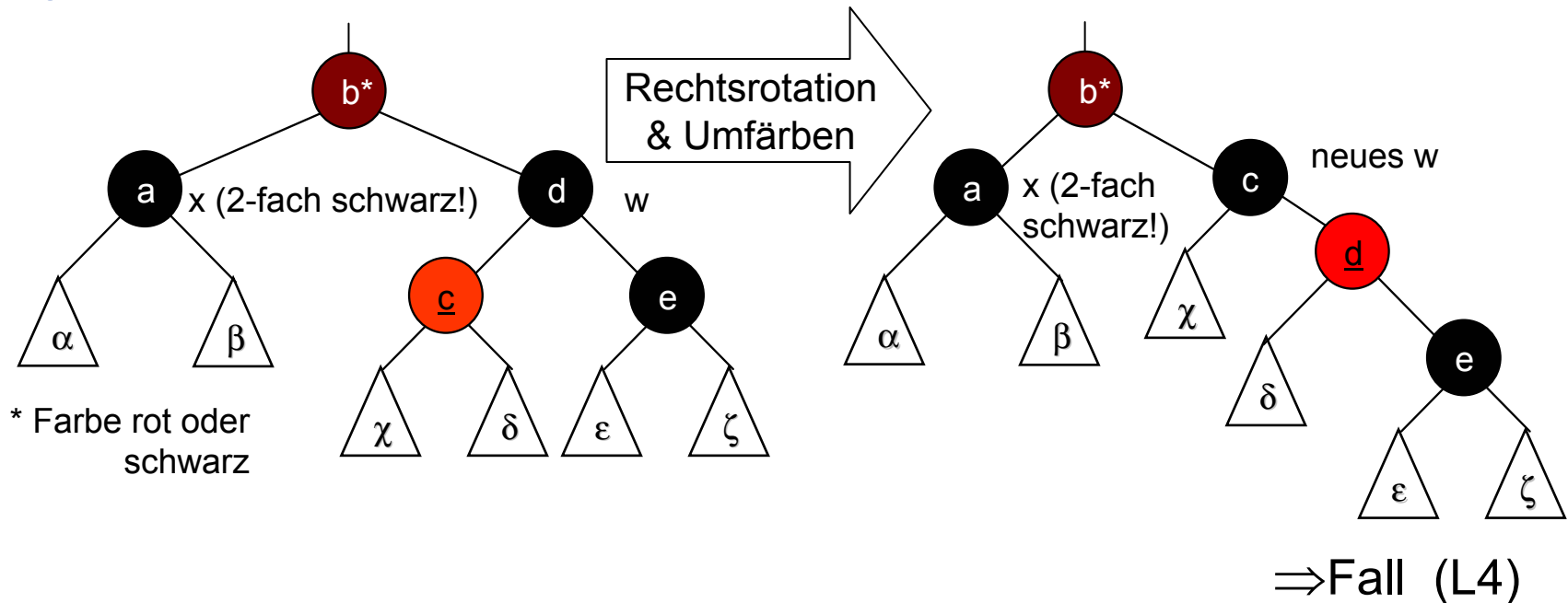


Vorgehen

- Umfärben von w und Reduzierung der Schwarz-Färbungen des Knotens x auf 1
 - $\text{parent}(x)$ um eine virtuelle Schwarz-Färbung ergänzen
 - Wiederholen mit $x = \text{parent}(x)$
-
- Eigenschaft (5) für den Teilbaum wieder hergestellt
 - Keine Änderung bei den anderen Eigenschaften



Fall (L3): w und dessen rechtes Kind sind schwarz, linkes Kind ist rot

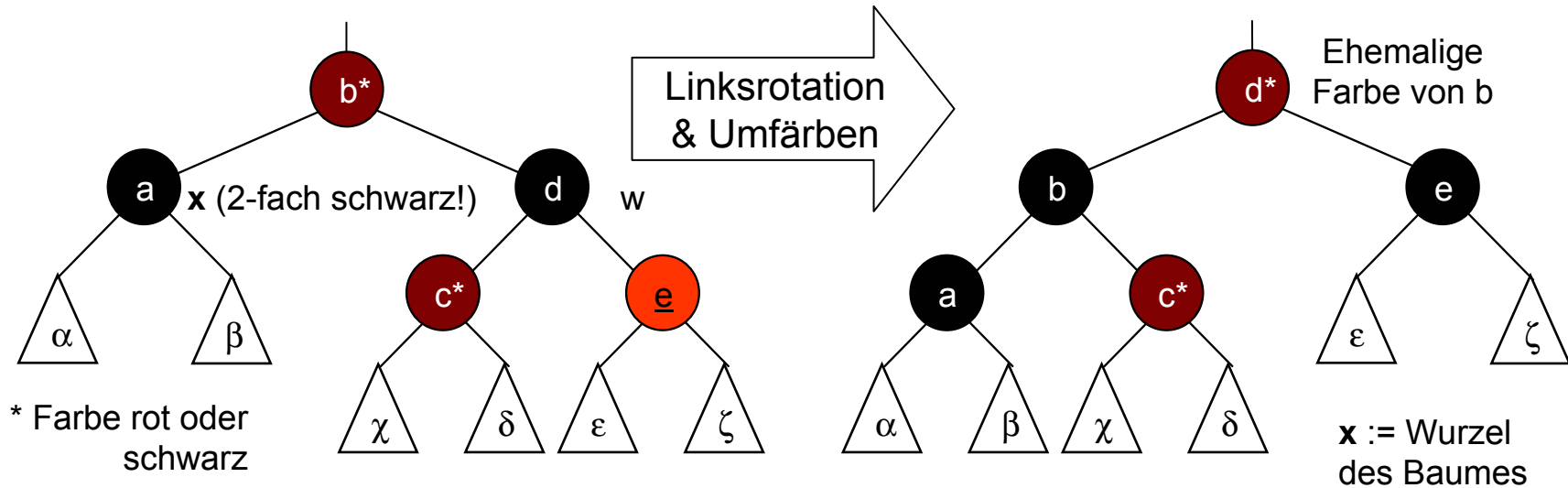


Vorgehen

- Umfärben von w und dessen linkem Kind und nachfolgende Rechtsrotation
- Transformation zu Fall (L4)
- Eigenschaften bleiben unverändert



Fall (L4): w ist schwarz und rechtes Kind ist rot



Vorgehen

- Schwarz färben von $\text{parent}(x)$, rot färben von Bruder
- Linksrotation um $\text{parent}(x)$
- Schwarz-Färbung des Knotens x auf 1 reduziert
- Wurzel des Baums wird als neuer Knoten x betrachtet
 - Evtl. Schwarz-färben der Wurzel
 - Beendet.



```
public class RBTree extends BinaryTree
{
```

```
//...
```

```
// Manipulatoren
```

```
private void rotateLeft
(RBTreeNode n){
```

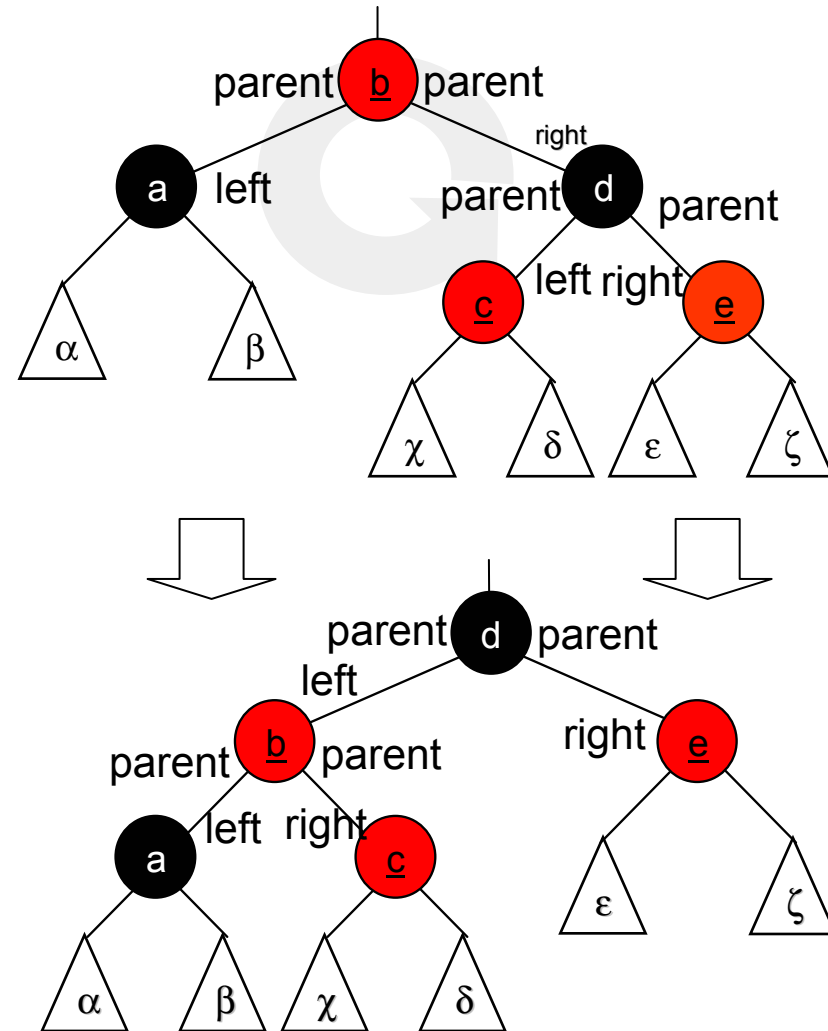
```
// Analog zu AVL-Bäumen !
```

```
// Allerdings ist die Parent-
```

```
// Beziehung zu berücksichtigen.
```

```
}
```

```
// Analog rotateRight(RBTreeNode n)
```

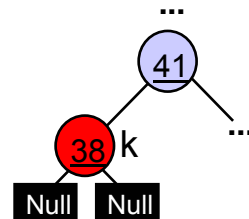


// ... Manipulatoren

```
public boolean insert(Object v, Object o) {
    if (findNode(v) == null) return false;
    rb_Insert(new RBTreeNode(v,o)); // Aufruf einer Hilfsmethode
    return true;                     // mit neu erzeugtem Knoten
}

private void rb_Insert(RBTreeNode k) {
    RBTreeNode o;
    // Zunächst Einfügen entspr. Binärem Suchbaum
    // Achtung: Eigener Dienst, da Parent-Beziehung zu
    // behandeln ist!
    bs_Insert(k);
    // Jeder (!) neu eingefügte Knoten wird zunächst Rot
    // gefärbt
    k.setColor(RED);
    // Hier nun die Iteration zur Ausbalancierung bis zur
    // Wurzel und nur so lange, bis der betrachtete Knoten
    // Schwarz gefärbt ist (Fall (E3))
    while (k != root && k.getParent().getColor() == RED)
        //...
```

Beispiel für
Situation
nach
Einfügen:



```
// ... rb_insert (Fortsetzung)
```

```
while (k != root && k.getParent().getColor() == RED)
```

```
    // Fallunterscheidung Einfügen:
```

```
    // parent(k) ist linkes Kind von parent(parent(k)).
```

```
    if (k.getParent() == k.getParent().getParent().getLeft()) {
```

```
        // Für Fallunterscheidung wichtig: der „Onkel“ von k (=:o)
```

```
        o = k.getParent().getParent().getRight();
```

```
    // Fallunterscheidung Fall (E1): Der „Onkel“ von k ist rot.
```

```
    if (o.getColor() == RED) {
```

```
        k.getParent().setColor(BLACK);
```

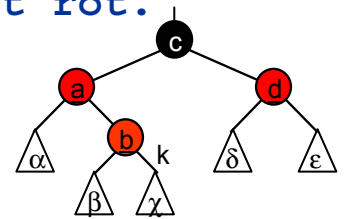
```
        o.setColor(BLACK);
```

```
        k.getParent().getParent().setColor(RED);
```

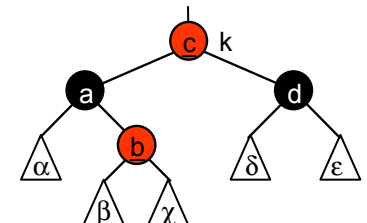
```
        k = k.getParent().getParent();
```

```
    } // Ende Fall (E1)
```

Fall (E1):

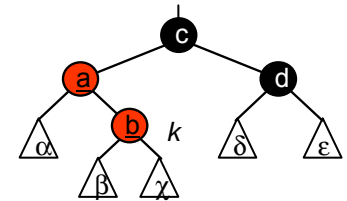


Umfärben



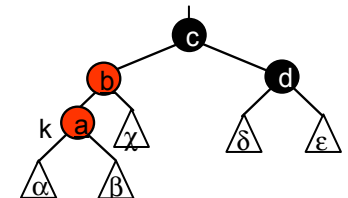
```
// ... rb_insert (Fortsetzung)
// Fall (E2/3): Der „Onkel“ von k ist schwarz und...
else { // Fall (E2): ... k ist rechtes Kind.
    if (k == k.getParent().getRight()) {
        k = k.getParent();
        rotateLeft(k);
    }
    // Fall (E3): ... k ist linkes Kind.
    k.getParent().setColor(BLACK);
    k.getParent().getParent().setColor(RED);
    rotateRight(k.getParent().getParent());
}
} // Ende Behandlung Einfügen ... ersten 3 Fälle
else { // weitere 3 Fälle: Behandlung analog
}
// Ende While
root.setColor(BLACK); // Wurzel wird Schwarz gefärbt
} // Ende rb_insert()
```

Fall (E2):

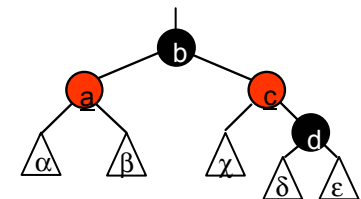


Links-rotation

Fall (E3):



Umfärben & Rechts-rotation

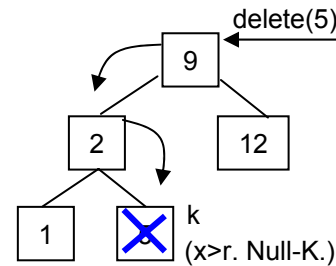


// ... Manipulatoren (Fortsetzung)

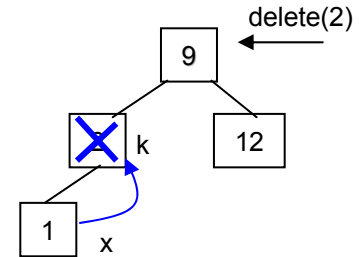
```
public boolean delete(Object v) {
    RBTreeNode k = findNode(v);
    if ( k == null ) return false;
    rb_Delete( k );
    // Aufruf einer Hilfsmethode
    return true;
}

private void rb_Delete(RBTreeNode z) {
    RBTreeNode x,k;
    // k: der tatsächlich entfernte Knoten
    // x: Kind von k
    if (z.getLeft().isNull() |
        z.getRight().isNull()) k = z;
    else k = bs_Successor(z);
    if (! k.getLeft().isNull()) x = k.getLeft();
    else x = k.getRight(); // ...
}
```

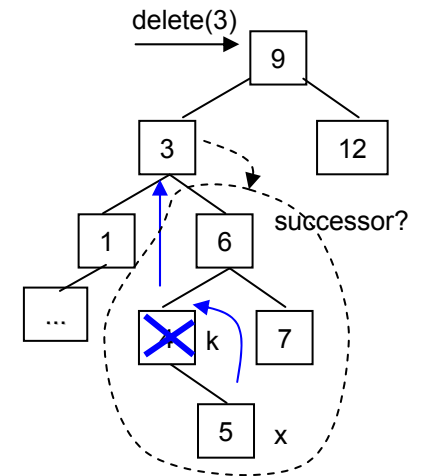
(1) Knoten ist Blatt



(2) Knoten hat ein Kind



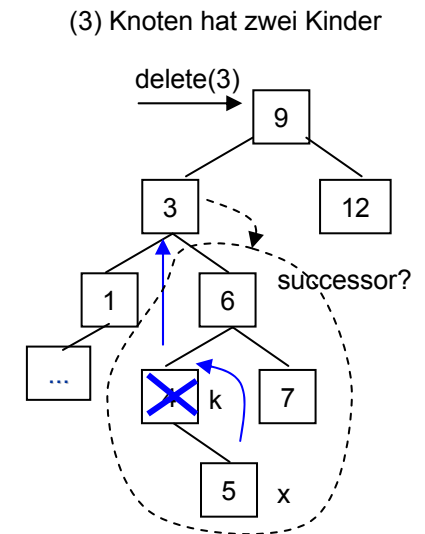
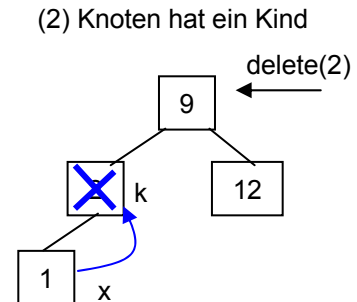
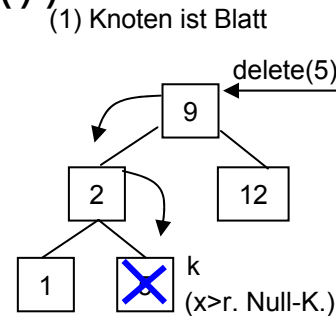
(3) Knoten hat zwei Kinder



```
// ... rb_Delete (Fortsetzung)
```

```
if (k.getParent() == null) root = x;
    else if (k == k.getParent().getLeft())
        k.getParent().setLeft(x);
    else k.getParent().setRight(x);
if (k != z) z.setKey(k.getKey());
// Nur, wenn der gelöschte Knoten
// Schwarz gefärbt war, muss
// ausgeglichen werden
if (k.getColor() == BLACK) rb_Delete_Fixup(x);
} // Ende rb_Delete
```

```
private void rb_Delete_Fixup(RBTreeNode x) {
    RBTreeNode w;
    // Ausgleich in Richtung Wurzel (Iteration):
    while (x != root & x.getColor() == BLACK)
        if (x == x.getParent().getLeft()) {
            // Fallgruppe a ...
```



```
// ... rb_Delete_Fixup (Fortsetzung)
```

```
w = x.getParent().getRight();
```

```
// w = „Bruder“ von x
```

```
// Fall (L1): w ist rot
```

```
if (w.getColor() == RED) {
```

```
    w.setColor(BLACK);
```

```
    x.getParent().setColor(RED);
```

```
    rotateLeft(x.getParent());
```

```
    w = x.getParent().getRight();
```

```
} // Fall (L2/3/4): w ist schwarz
```

```
// Fall (L2): ... und beide Kinder von w
```

```
// sind schwarz
```

```
if (w.getLeft().getColor() == BLACK &
```

```
    w.getRight().getColor() == BLACK) {
```

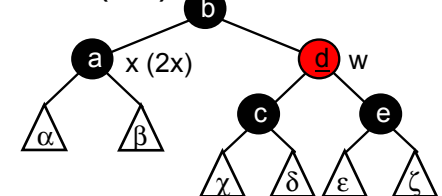
```
    w.setColor(RED);
```


```
    x = x.getParent();
```

```
} // else (d.h. nicht beide Kinder von w
```

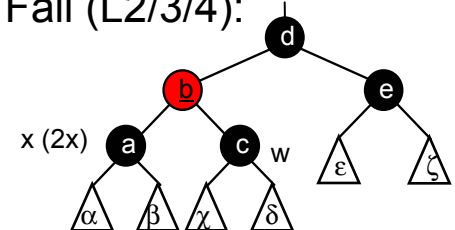
```
// sind schwarz)
```

Fall (L1):

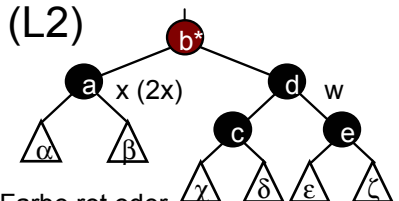


Linksrotation  & Umfärben

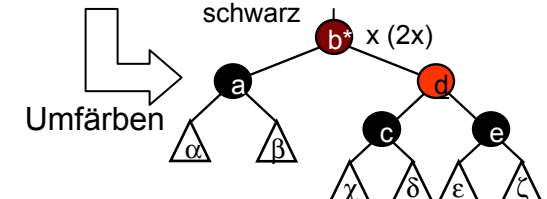
Fall (L2/3/4):



Fall (L2)



* Farbe rot oder schwarz

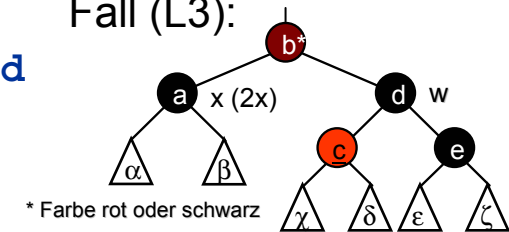



Umfärben

```
// ... rb_Delete_Fixup (Fortsetzung)
```

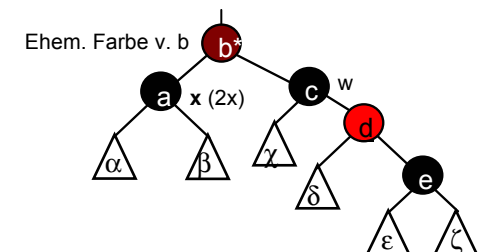
```
else {
    // Fall (L3): w und dessen rechtes Kind sind
    // schwarz, w's linkes Kind ist rot
    if (w.getRight().getColor() == BLACK) {
        w.getLeft().setColor(BLACK);
        w.setColor(RED);
        rotateRight(w);
        w = x.getParent().getRight();
    } // Fall (L4): w ist schwarz und
      // w's rechtes Kind rot:
    w.setColor(x.getParent().getColor());
    x.getParent().setColor(BLACK);
    w.getRight().setColor(BLACK);
    rotateLeft(x.getParent());
    x = root; // Fast fertig
} // Ende Fallgruppe a
```


Fall (L3):

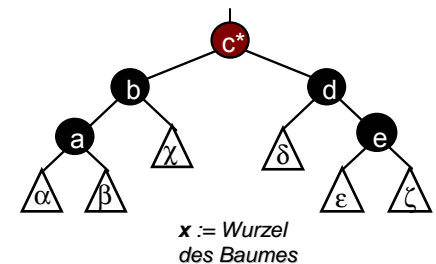


Rechtsrotation  & Umfärben

Fall (L4):



Linksrotation  & Umfärben



```
// ... rb_Delete_Fixup (Fortsetzung)
else {      // Fallgruppe b: symmetrische Fälle
            // Behandlung analog zu Fallgruppe a
}

// Der zuletzt behandelte Knoten x wird schwarz gefärbt.
// Dies kann insb. der Wurzelknoten sein
// > Eigenschaft 2 wird sichergestellt
x.setColor(BLACK);
} // Ende rb_Delete_Fixup

// Weitere Dienste

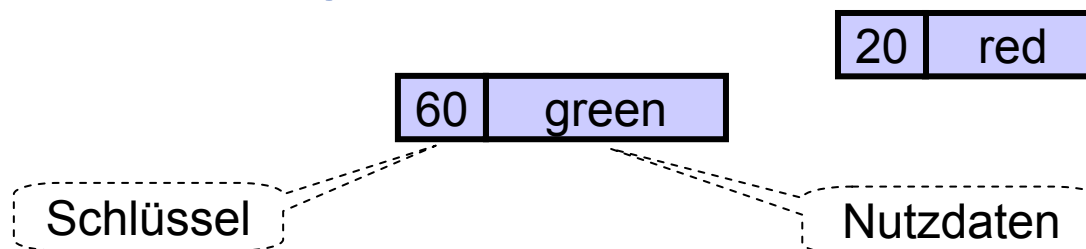
} // Ende RBTree
```



Implementierung von Mengen auf Hintergrundspeicher

- Sollen riesige Objektmengen verwaltet werden, erfolgt eine Speicherung auf einem Hintergrundspeicher (z.B. Festplatte).
- Für die Suche, das Einfügen und das Löschen von Objekten wird eine Datenstruktur benötigt, bei der die Anzahl der Zugriffe auf den Hintergrundspeicher so gering wie möglich ist:
 - Ein Hintergrundspeicher unterstützt meist das blockweise Lesen/Schreiben "größerer" Objektmengen bei einem Zugriff.
 - Idee: Objekte, die bei den Operationen zusammen verwendet werden, möglichst in einem Block speichern.
- Von R. Bayer und E. McCreight entwickelt

Die zu verwaltenden Objekte bestehen meist aus einem Schlüsselwert für den assoziativen Zugriff und Nutzdaten:



- Studium der Mathematik in München und Illinois
- 1966 Promotion in Illinois.
- Professor an der Purdue University und seit 1972 TU München
- Gastprofessor bei IBM, XEROX PARC sowie verschiedenen Universitäten in Japan, Australien, den USA und Singapur
- Vorsitzender der Forschungsgruppe für wissensbasierte Systeme am bayrischen Forschungszentrum FORWISS
- Arbeiten für Boeing, IBM, Siemens, Amdahl, DEC, Deutsche Telekom
- Mitbegründer der TransAction SW GmbH, im Besitz von zwei Patenten.
- Mitherausgeber von „ACM TODS“ und „Informatik Forschung und Entwicklung“
- Bundesverdienstkreuz 2001
- ACM SIGMOD Innovations Award 2001



Ein Baum ist ein B-Baum vom Typ $\tau(k, h)$, wenn er ein leerer Baum ist oder ein Baum mit folgenden Eigenschaften

- Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Zahl von Kanten, d.h. jeder Pfad definiert die Höhe h des Baums.
- Jeder Knoten j ist mit n_j sortiert angeordneten Schlüsseln belegt (sowie mit Nutzdaten).
- Jeder Knoten mit n_j Schlüsseln, der nicht Blatt ist, hat n_j+1 Söhne. Ist der Knoten Wurzel, so gilt $1 \leq n_j \leq 2k$, andernfalls $k \leq n_j \leq 2k$.
- Seien S_1, \dots, S_n die Schlüssel eines Knotens j mit n_j+1 Söhnen. Seien Z_0, Z_1, \dots, Z_n die Zeiger auf diese Söhne.
 - a) Z_0 weist auf Teilbaum mit Schlüsseln kleiner S_1
 - b) $Z_i (i=1, \dots, n-1)$ weist auf Teilbaum, dessen Schlüssel echt zwischen S_i und S_{i+1} liegen.
 - c) Z_n weist auf Teilbaum mit Schlüsseln größer S_n .
 - d) In den Blattknoten sind die Zeiger nicht definiert.



Höhe eines B-Baums mit N Knoten:

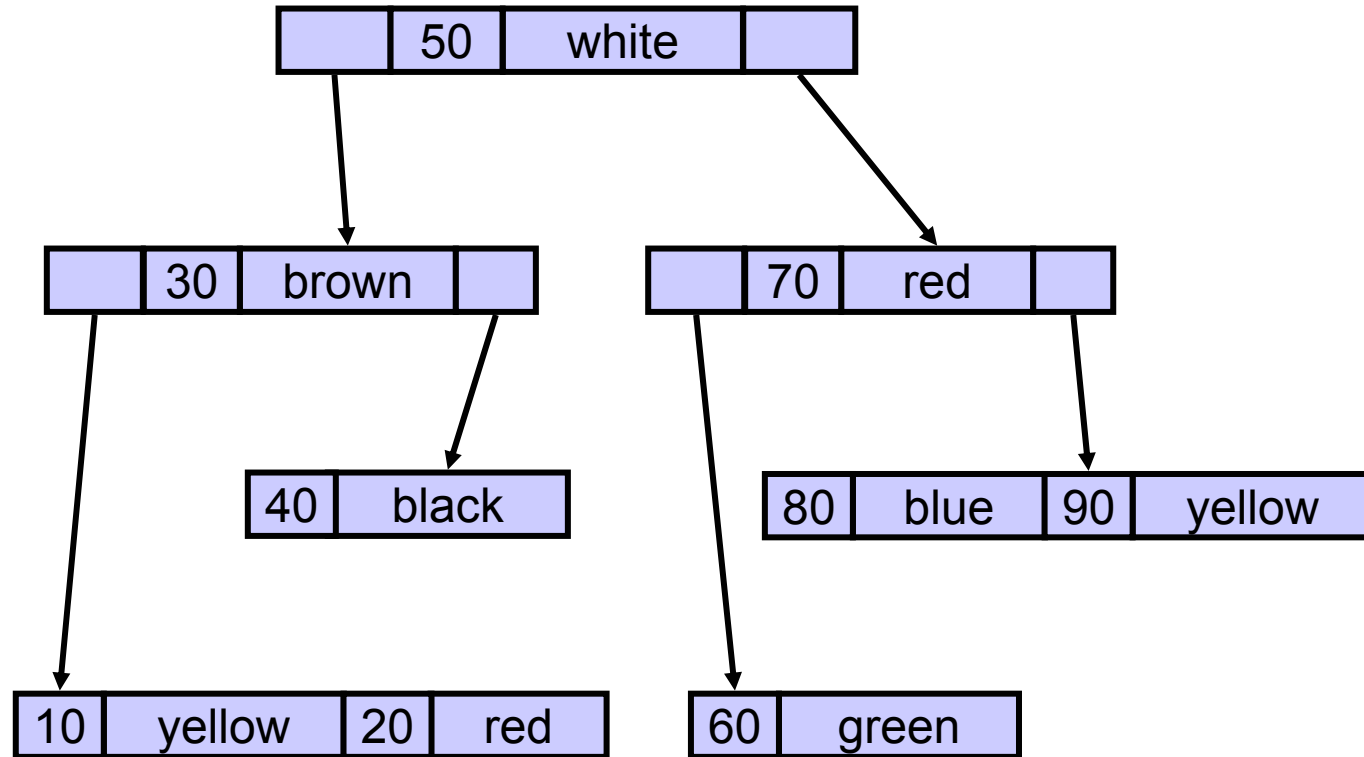
$$\log_{2k+1}(N+1)-1 \leq h \leq \log_{k+1}\left(\frac{N+1}{2}\right) \quad (N \geq 1)$$

Betrachte Objektmenge:

Annahme:
 $k=1$

Id	Farbe
10	yellow
20	red
30	brown
40	black
50	white
60	green
70	red
80	blue
90	yellow

Schlüssel



Jeder Knoten enthält n Schlüssel mit

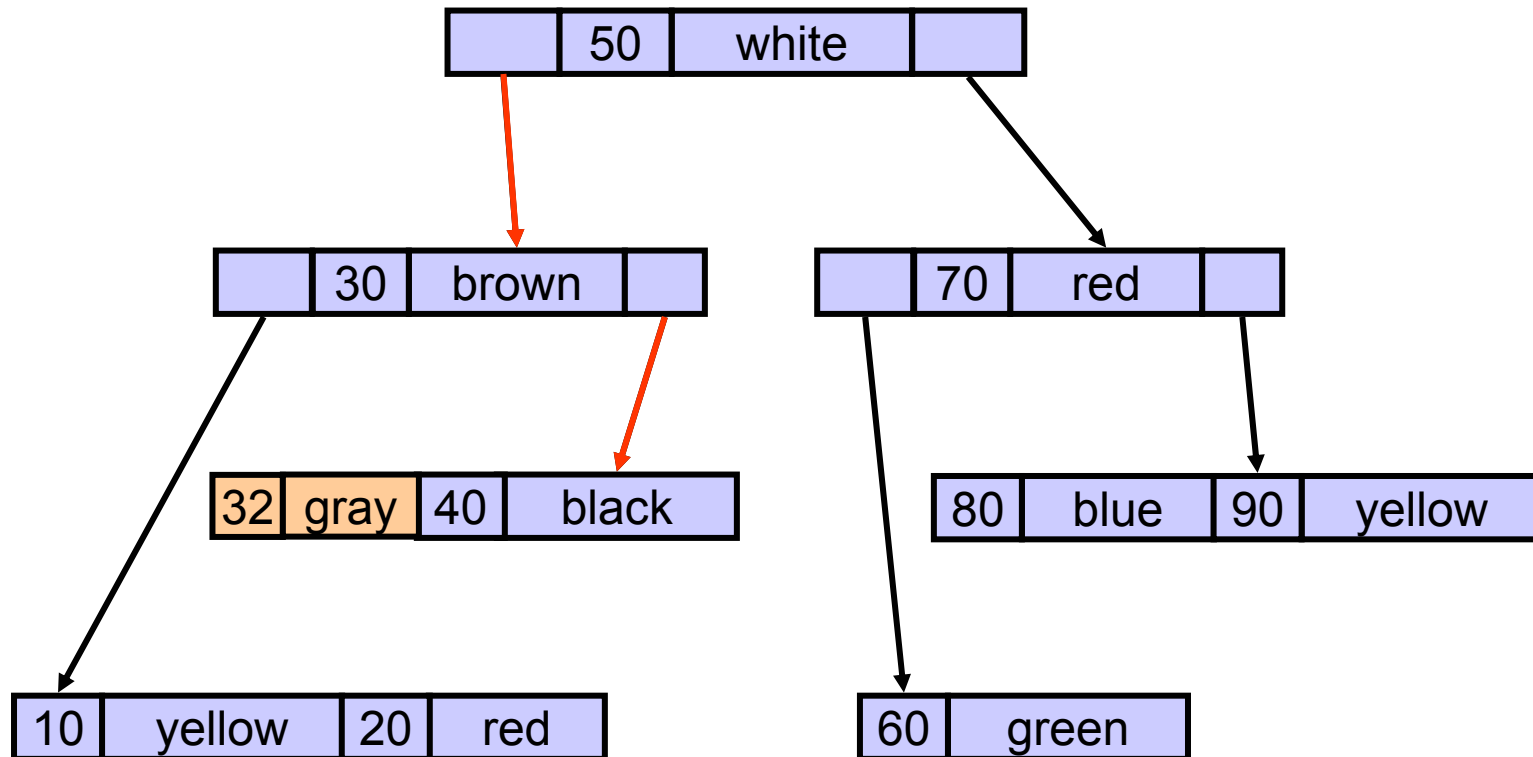
$$k \leq n \leq 2k$$

Einfügen erfordert 2 Phasen:

1. Suche nach dem Schlüssel des neuen Objekts (notfalls bis zum Blatt).
- 2a. Falls gefunden: Einfügen des neuen Objekts in den entsprechenden Knoten des B-Baums (bei nichteindeutigem Schlüssel).
- 2b. Falls nicht gefunden: Einfügen des neuen Objekts in das bei der Suche erreichte Blatt.

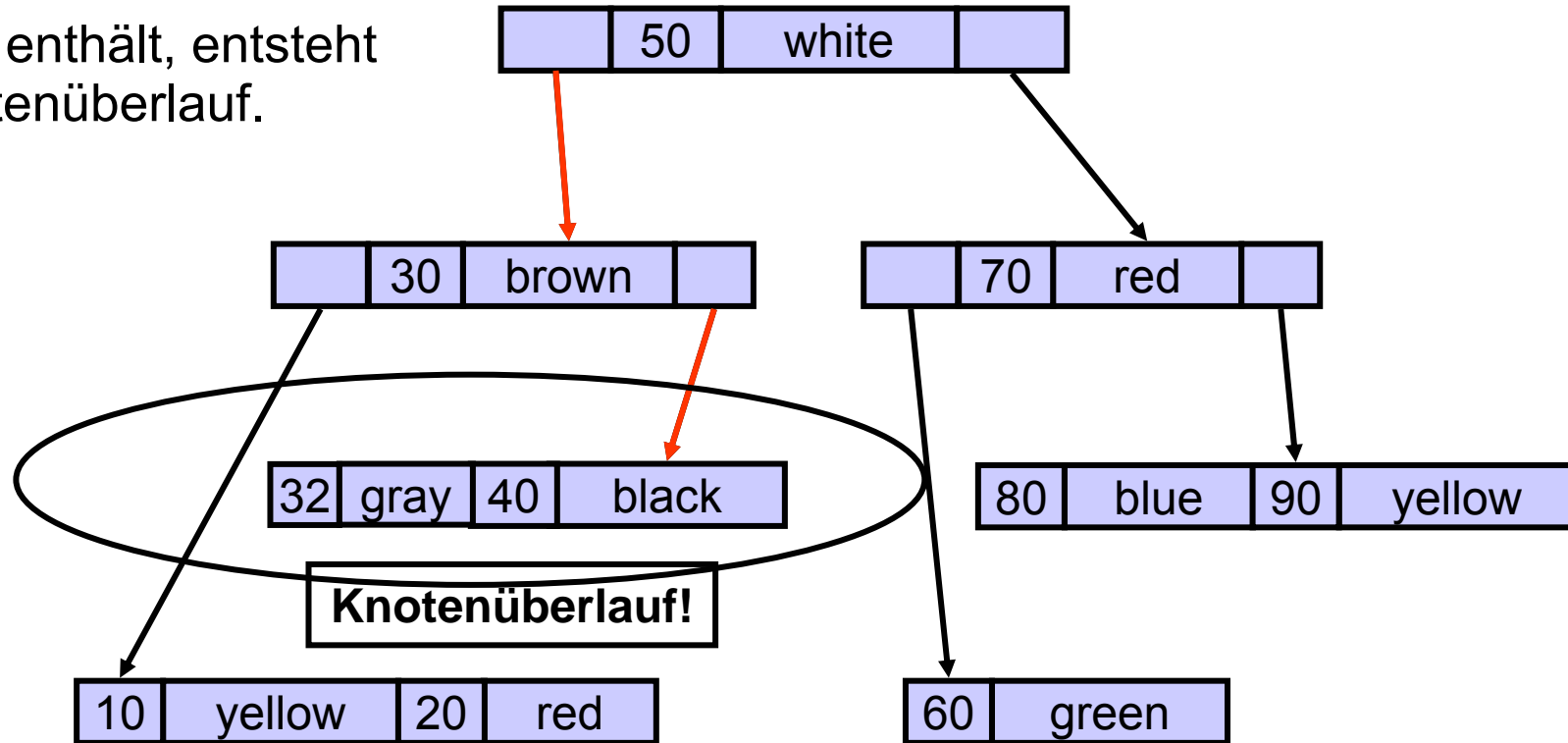


Einfügen von [32, gray]



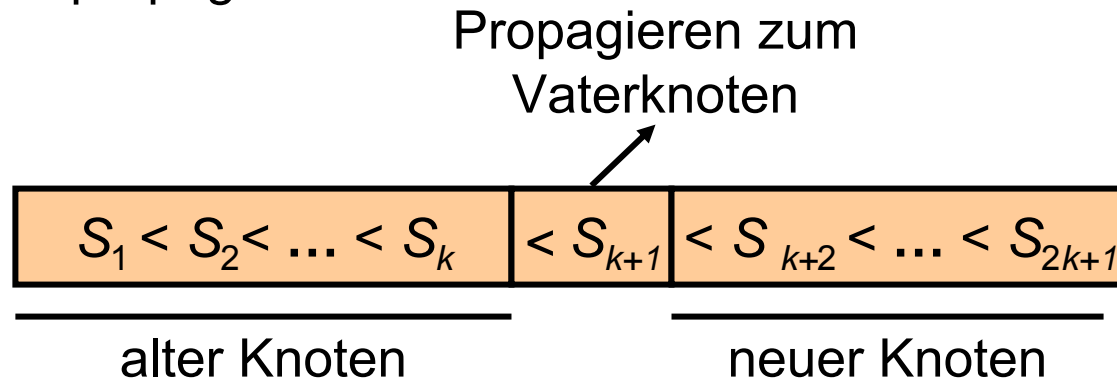
Einfügen von [42, rose]

Falls der Knoten
nunmehr mehr als $2k$
Objekte enthält, entsteht
ein Knotenüberlauf.



Fortsetzung

- Bei Überlauf eines Knotens wird der Knoten gespalten, d.h. die Einträge des Knotens werden auf zwei Knoten verteilt.
- Das Objekt, das in der Mitte des übergelaufenen Knotens liegt, wird zum Vaterknoten propagiert.



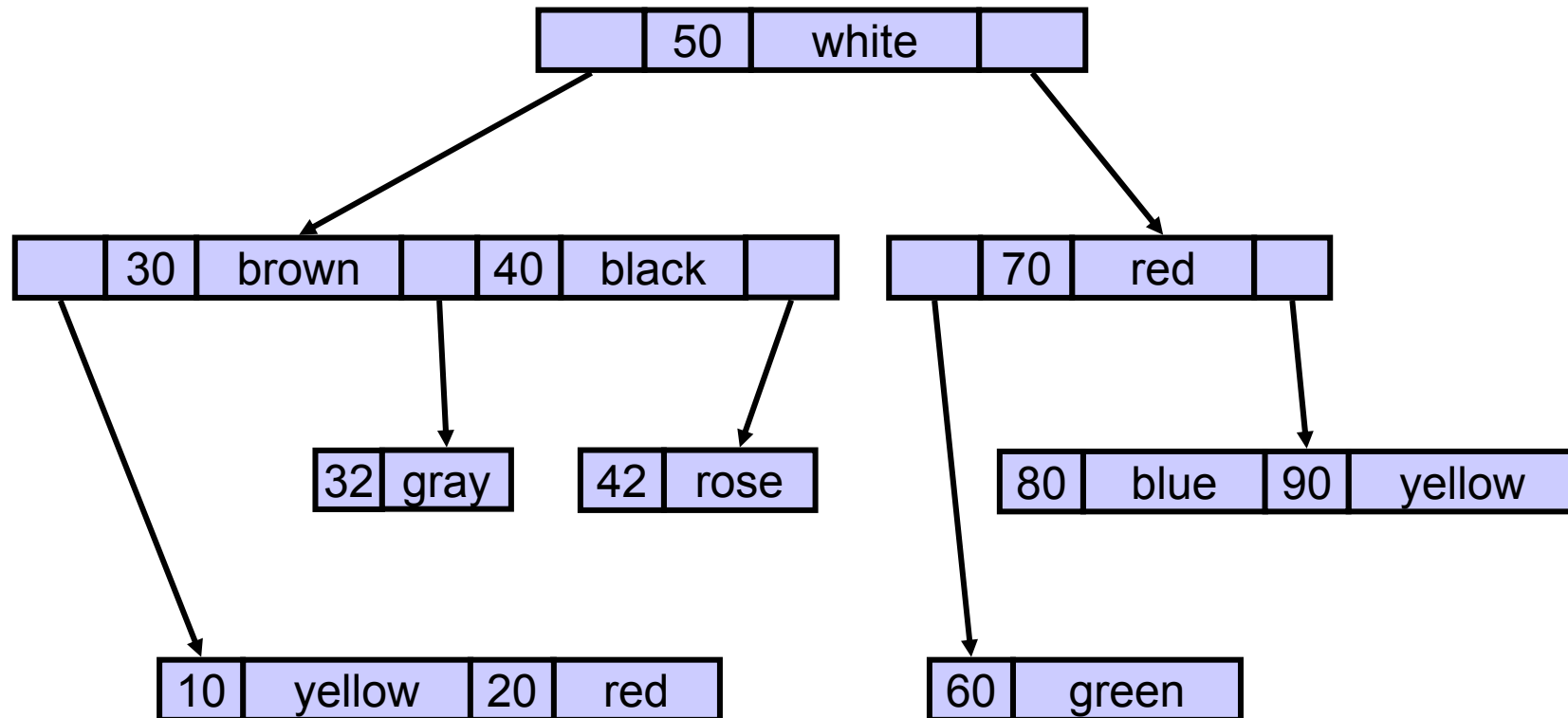
Hier:



- Schema
 - Die ersten m Werte verbleiben im alten Knoten
 - Die letzten m Werte werden verschoben in neuen Knoten
 - Das mittlere Element wandert in den Vaterknoten



Einfügen von [42, rose]



Spalten von Knoten kann sich im Vaterknoten rekursiv fortsetzen.

Schlimmster Fall

- Alle Knoten bis zur Wurzel laufen über und werden gespalten.
- Wenn sogar die Wurzel überläuft, wird auch sie gespalten.
- Folge: Der Baum wächst in der Höhe um 1.

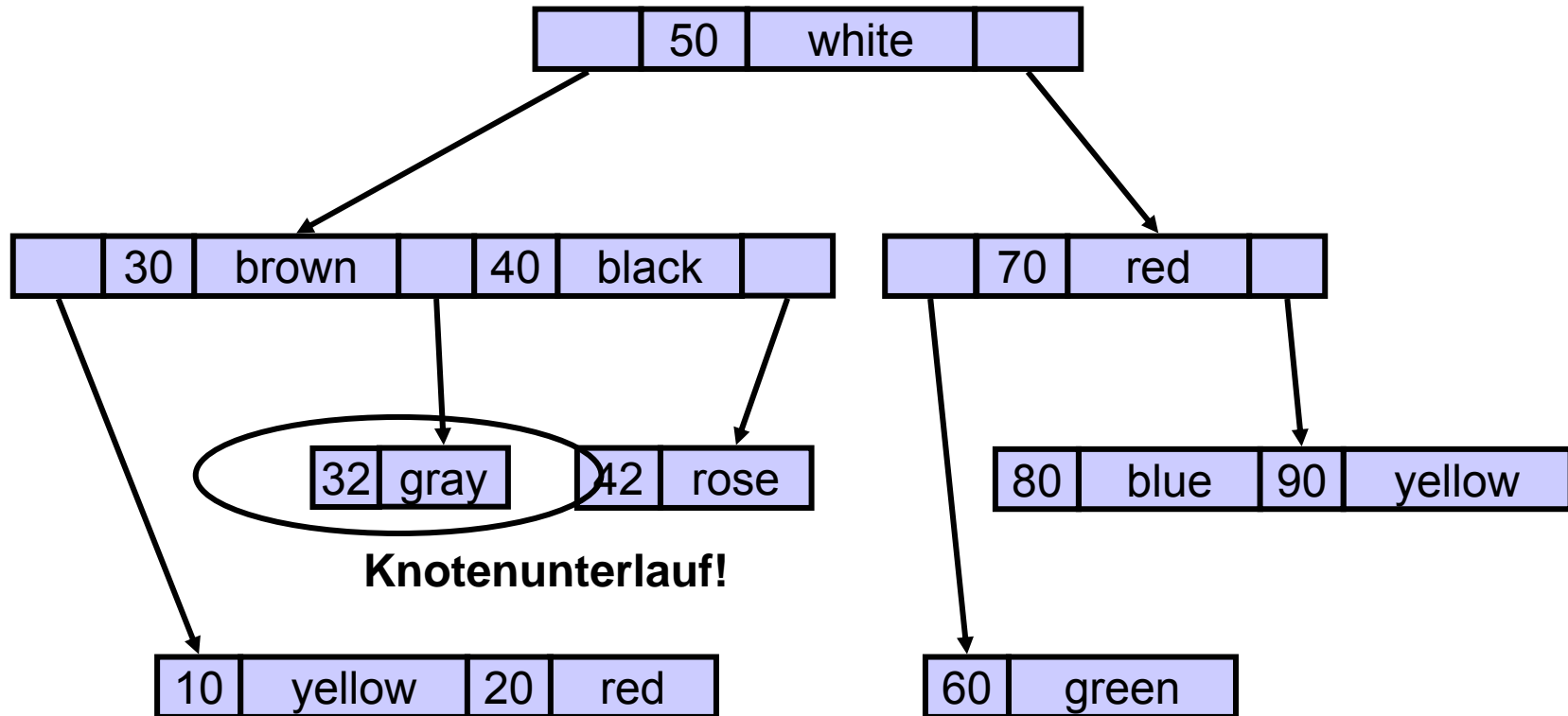


Entfernen erfordert 2 Phasen

- Suche nach dem zu entfernenden Objekt anhand seines Schlüssels.
- Entferne das Objekt. Zwei Fälle:
 - Objekt befindet sich in einem Blatt.
 - Objekt befindet sich in einem internen Knoten oder in der Wurzel.



Entfernen des Datensatzes [32, gray] (Blattknoten)



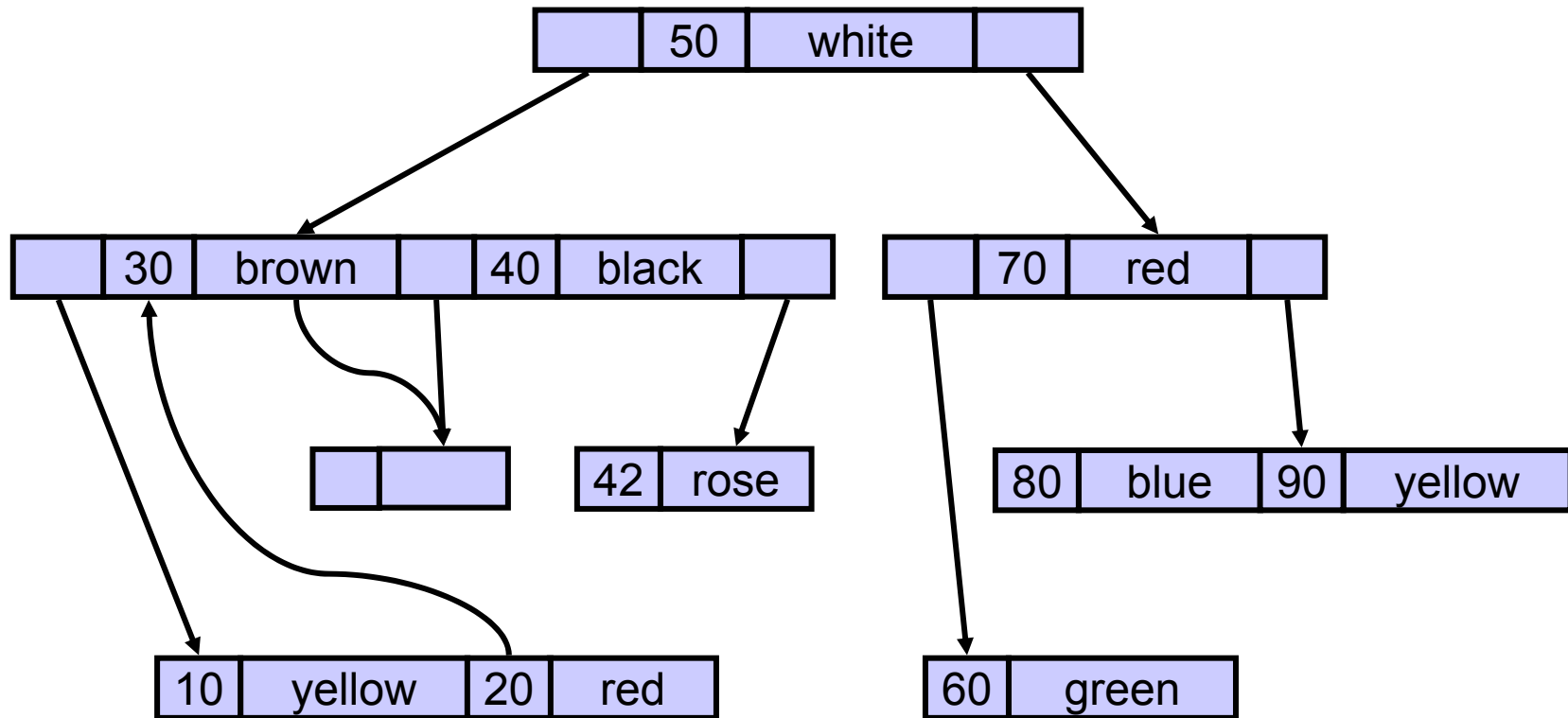
Falls der Blattknoten nunmehr weniger als k Objekte enthält, entsteht ein Knotenunterlauf.

Bei einem Knotenunterlauf wird versucht, Objekte aus den Nachbarknoten in den Knoten mit Unterlauf zu verschieben.

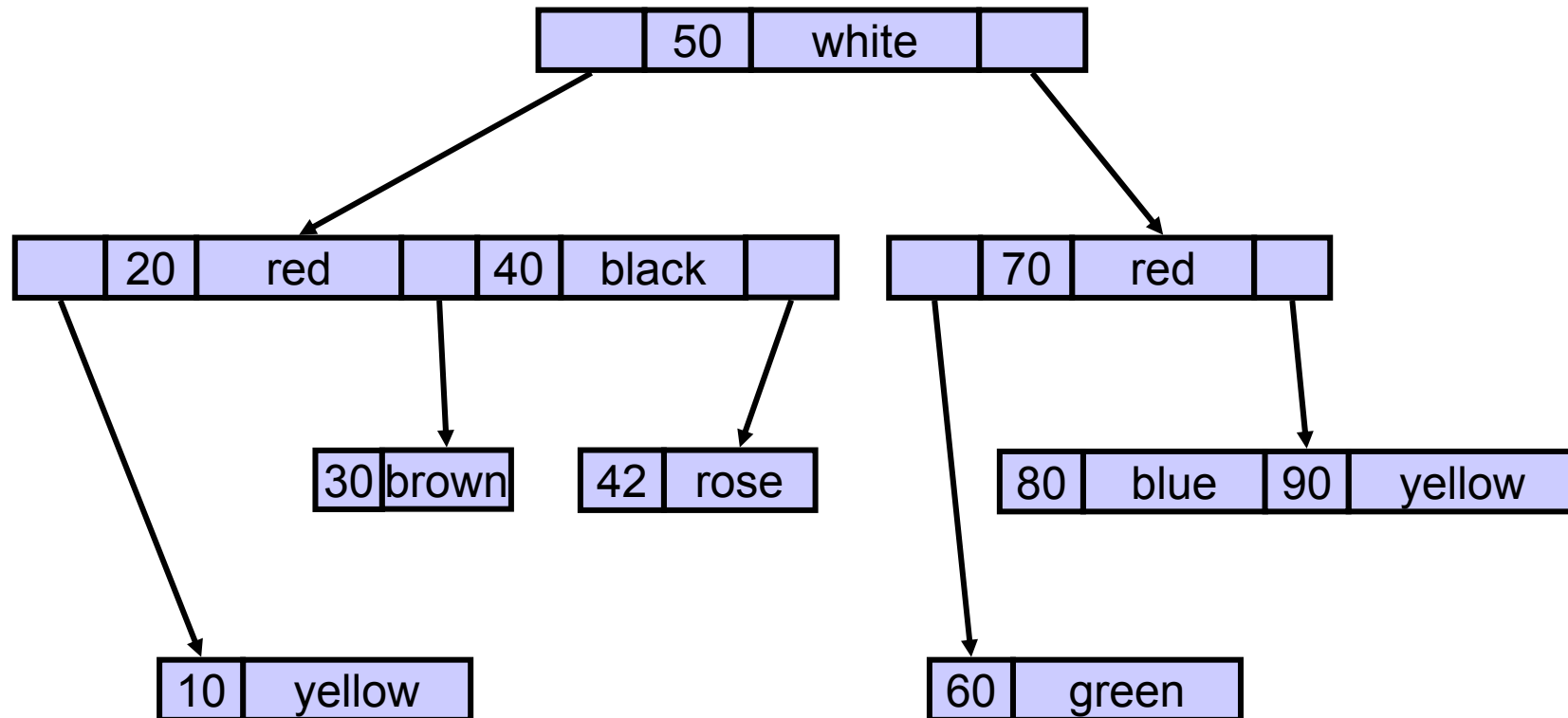
- Indirektion durch Bewegen über Vaterknoten!



Entfernen des Datensatzes [32, gray] (Blattknoten)



Entfernen des Datensatzes [32, gray] (Blattknoten)

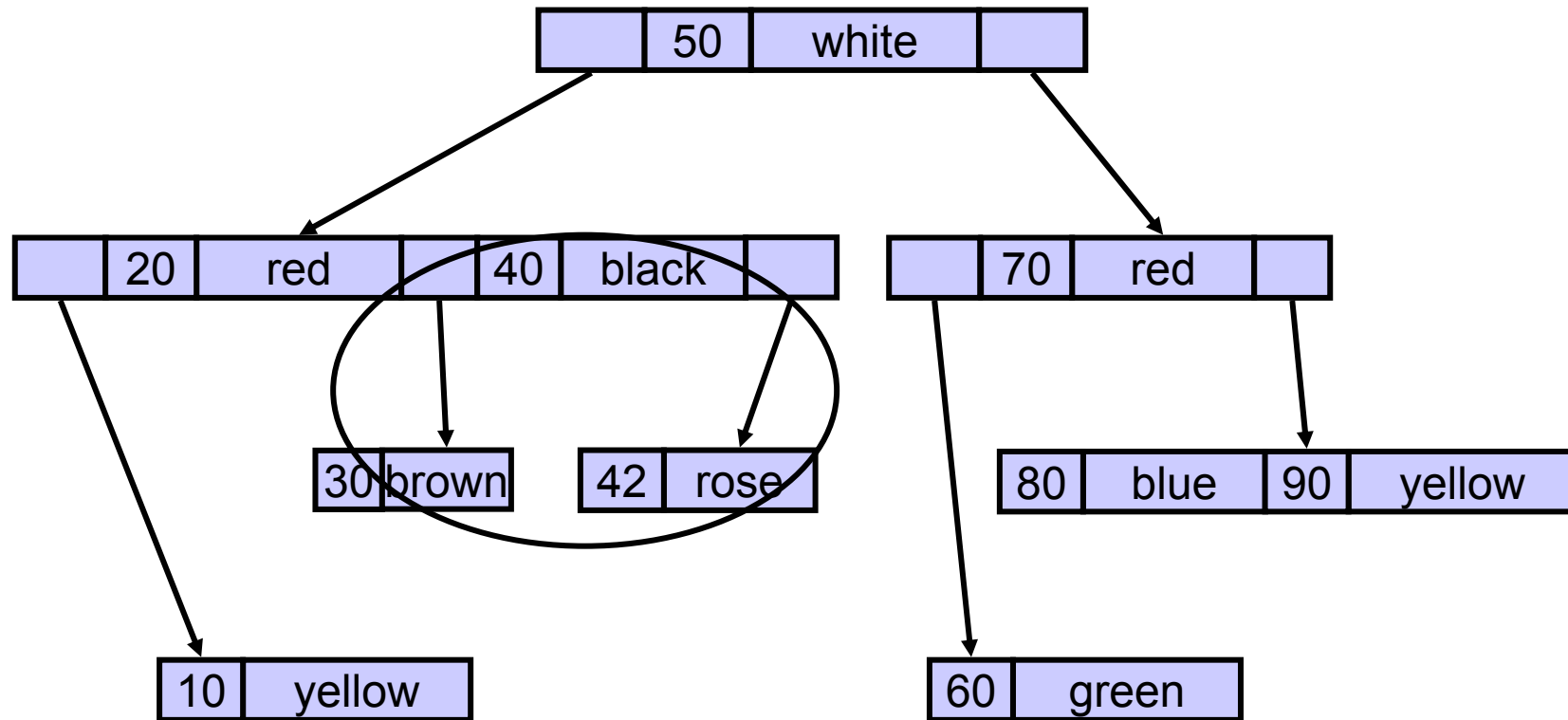


Fortsetzung

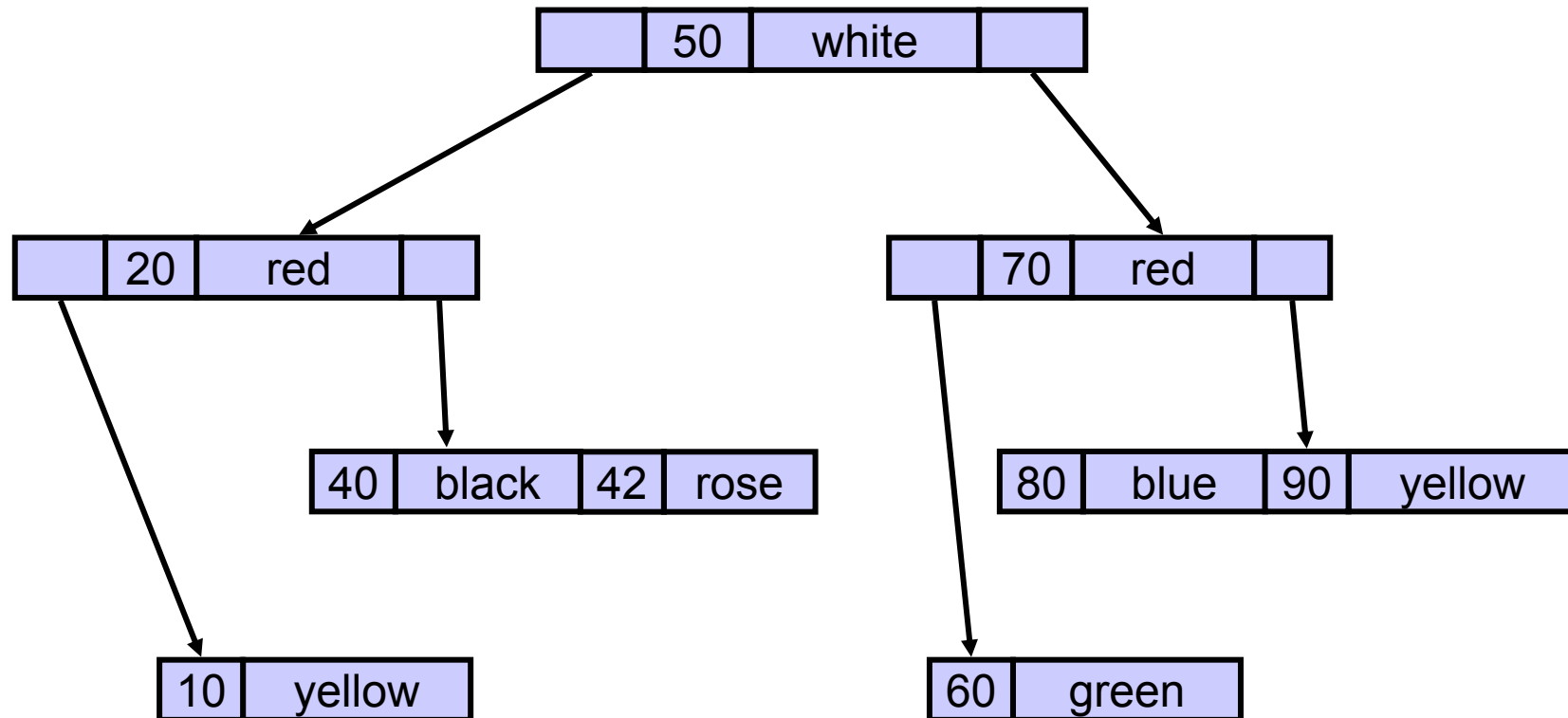
- Problem: Wenn alle Nachbarknoten nur minimal belegt sind, kann kein Ausgleich stattfinden (das Entfernen von Objekten aus den Nachbarknoten würde dort zu einem Unterlauf führen).
- In diesem Fall wird der Unterlaufknoten mit einem seiner Nachbarknoten verschmolzen.
 - Auch machbar, wenn 1 Nachbarknoten minimal belegt.
- Das Zusammenlegen zweier Knoten führt zu dem Verschieben des trennenden (*Separator*-) Objektes aus dem Vaterknoten in den neuen, durch das Verschmelzen entstehenden Knoten.
- Wie beim Splitten von Knoten kann sich das Verschmelzen von Knoten bis zur Wurzel fortsetzen.
- Folge: Der Baum schrumpft in der Höhe um 1.



Entfernen des Datensatzes [30, brown] (Blattknoten)



Entfernen des Datensatzes [30, brown] (Blattknoten)



Fortsetzung

- Zu löschendes Objekt befindet sich in einem inneren Knoten:
- Finde ein neues Separator-Objekt als Ersatz für das zu entfernende Objekt .

Zwei Alternativen für Auswahl des Separator-Objekts:

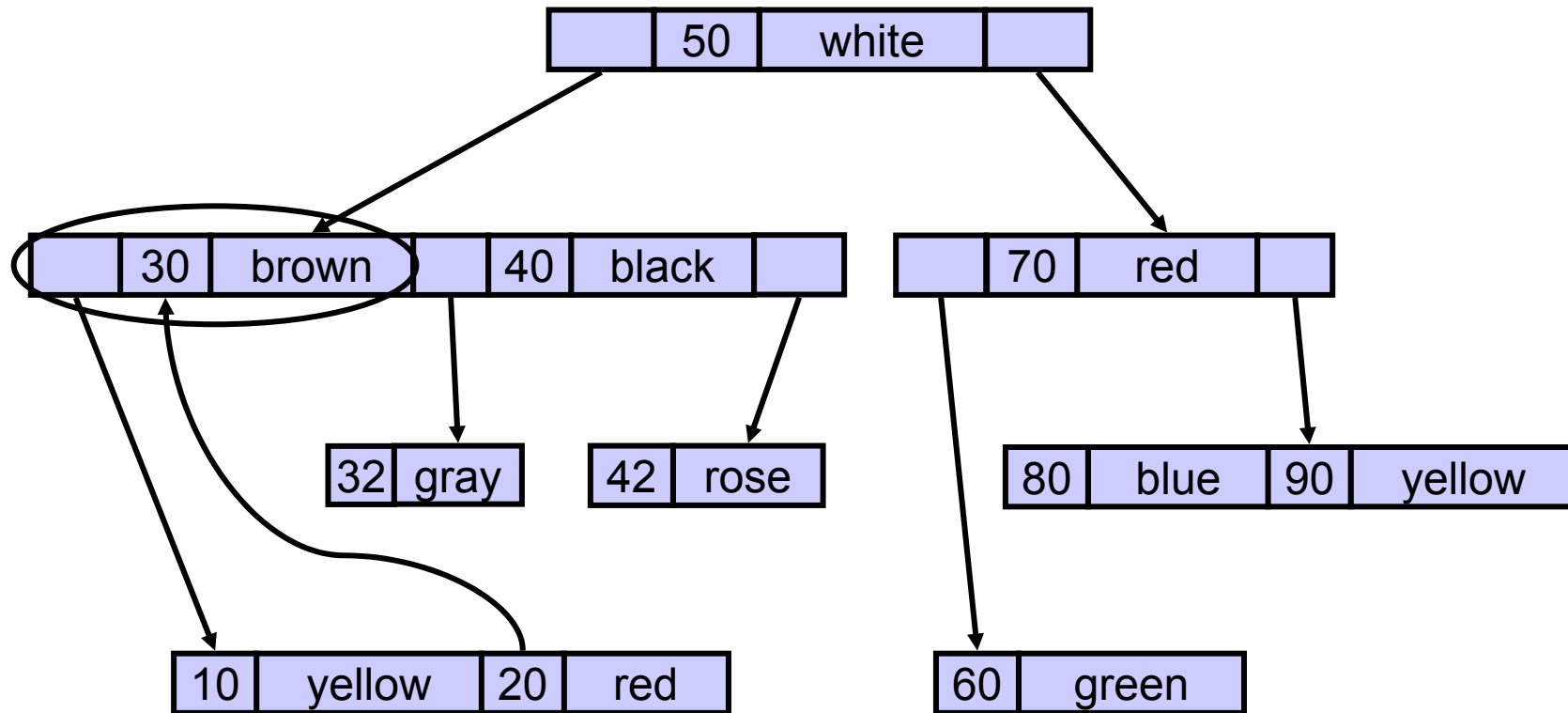
1. Wähle das Objekt mit dem kleinsten Schlüssel, der größer ist als der Schlüssel des zu entfernenden Objekts.
2. Wähle das Objekt mit dem größten Schlüssel, der kleiner ist als der Schlüssel des zu entfernenden Objekts.

Das ausgewählte Separator-Objekt

- wird aus seinem Knoten entfernt (Rekursiver Aufruf der Operation zum Entfernen von Objekten!) und
- ersetzt das zu entfernende Objekt in dessen Knoten.



Entfernen des Datensatzes [30, brown] (Interner Knoten)



Anwendungssituation

- Suchen nach Worten über einem Alphabet S z.B. in Texten oder HTML-Seiten durch WWW-Suchmaschinen, Content-Management-Systeme etc.

Bisher

- Jeder Knoten nimmt einen Wert bzw. Schlüssel vollständig auf

Probleme, wenn Worte als Werte von Knoten eines Suchbaumes gespeichert werden

- Worte müssen aufwändig Buchstabe für Buchstabe verglichen werden.
- Beispiel: „Organisationslehre“ < „Organisationsmuster“
- Kommen Wortanfänge mehrfach vor,
 - geht Speicherplatz verloren
 - müssen Vergleiche mehrfach bis zum ersten unterschiedlichen Buchstaben durchgeführt werden.



Suchbaum mit mehrwertigen Knoten, deren Werte

- den Buchstaben des Alphabets entsprechen und
- jeweils auf einen Kindknoten verweisen.

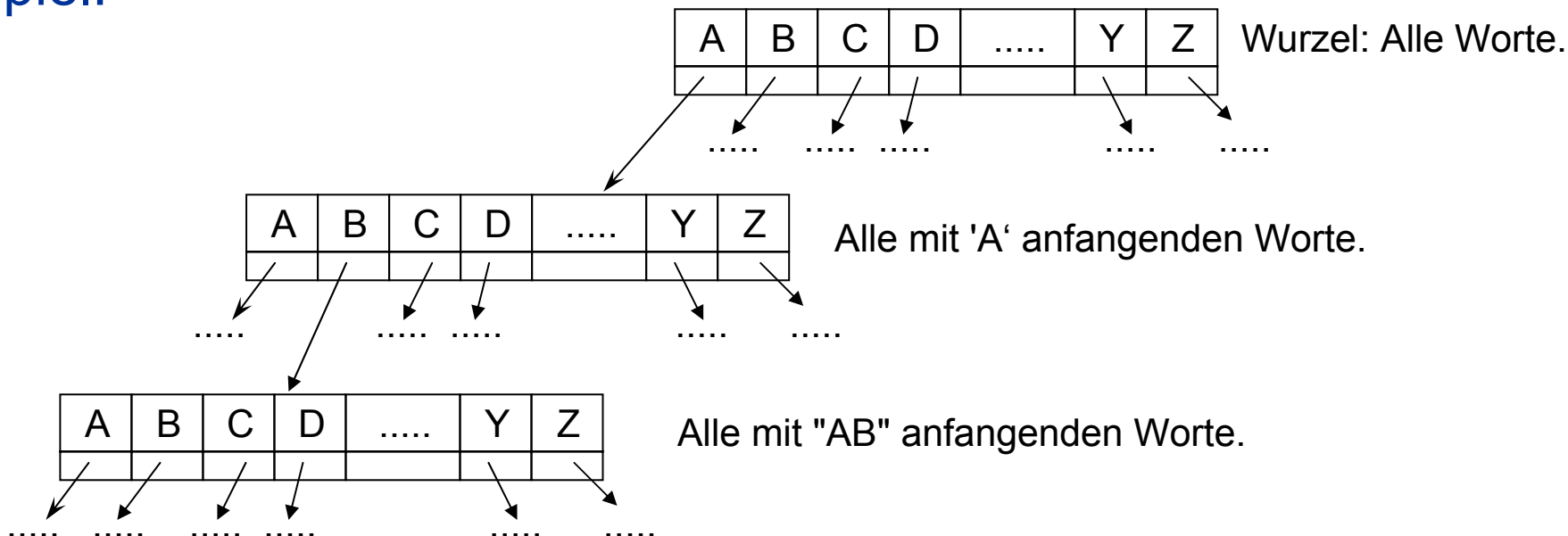
Das Verfolgen von Verweisen bis zu einem Blatt ergibt dann die gespeicherten Worte.

- Die Verweise sind dabei prinzipiell unabhängig von den gespeicherten Schlüsselwerten.
- Geeignet für Datentypen, bei denen eine derartige feste Verzweigung sinnvoll ist. Insbesondere: Zeichenketten über einem festen Alphabet mit Verzweigung nach dem jeweils ersten bzw. nächsten Buchstaben.
- Bezeichnung „digital“ entstammt der Interpretation von Zahlen eines Zahlensystems als Worte über dem Alphabet der Ziffern (insb. Binärzahlen über $\{0,1\}$) und deren Verwaltung in derartigen Bäumen.

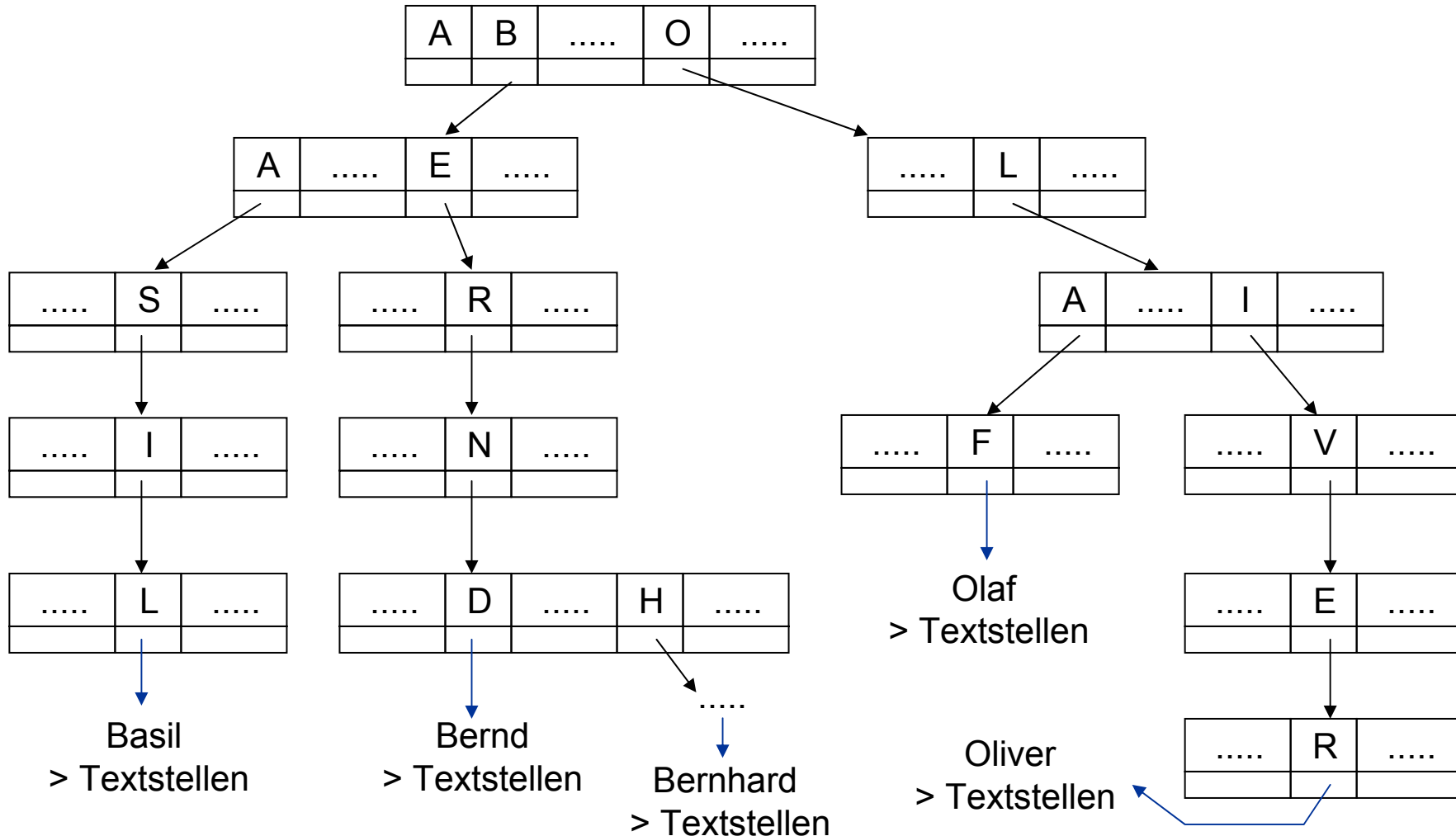


- Baumstruktur zur Speicherung von Zeichenketten, die das Suchen in Texten unterstützt.
- Das Wort Trie (gesprochen wie das engl. "try") entstammt der Hauptanwendung der Struktur, dem Text-Retrieval.
- Tries erlauben große Dokumentenbestände nach Zeichenketten zu durchsuchen: der Trie bildet einen so genannten Index zu diesen Dokumenten.

Beispiel:



Beispiel: Namens-Index (Basil, Bernd, Bernhard, Olaf, Oliver ...)



Ungleichmäßig verteilte Daten führen zu unausgeglichene z.T. entarteten Suchbäumen

- nicht vorkommende Buchstabenkombinationen ("QX" "QXY" etc.) führen zum Ende des Suchpfades bzw. zu Werten in Knoten ohne Verweise
- Längere Worte ohne verwandte Worte führen zu Knoten, die nur einen Nachfolger haben und entarten im weiteren Verlauf zu Listen.
 - Dies erzeugt ein ungünstiges Verhältnis von inneren Knoten zu Blättern.

Beispiel: „Xylophonspielerinnenvereinigung“

- Ab 'y' folgt eine Liste von 29 Knoten, von denen jeder nur ein Kind hat (sofern keine ähnlichen Worte enthalten sind).
- Letzteres führt dazu, dass ggf. unnötig lange Folgen von Knoten verfolgt werden müssen, bevor der gesuchte Verweis gefunden wird.



- Weiterentwicklung von Tries zur Vermeidung entarteter Strukturen.

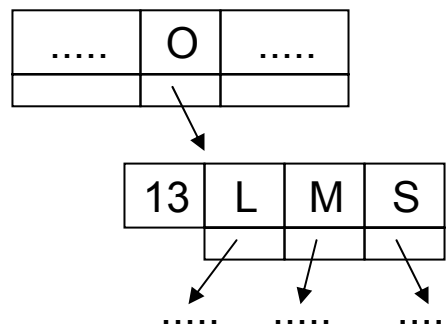
Patricia

- Practical Algorithm to Retrieve Information Coded in Alphanumeric.
- Ursprüngliche Form bezieht sich auf Binärzahlen.

Idee:

- Teile von Zeichenketten, die im weiteren Vergleich nicht zu Verzweigungen führen, werden übersprungen.
- Jeder Knoten enthält Anzahl der zu überspringenden Zeichen sowie nur die Buchstaben, die danach zu Vergleichen lohnt

Beispiel:



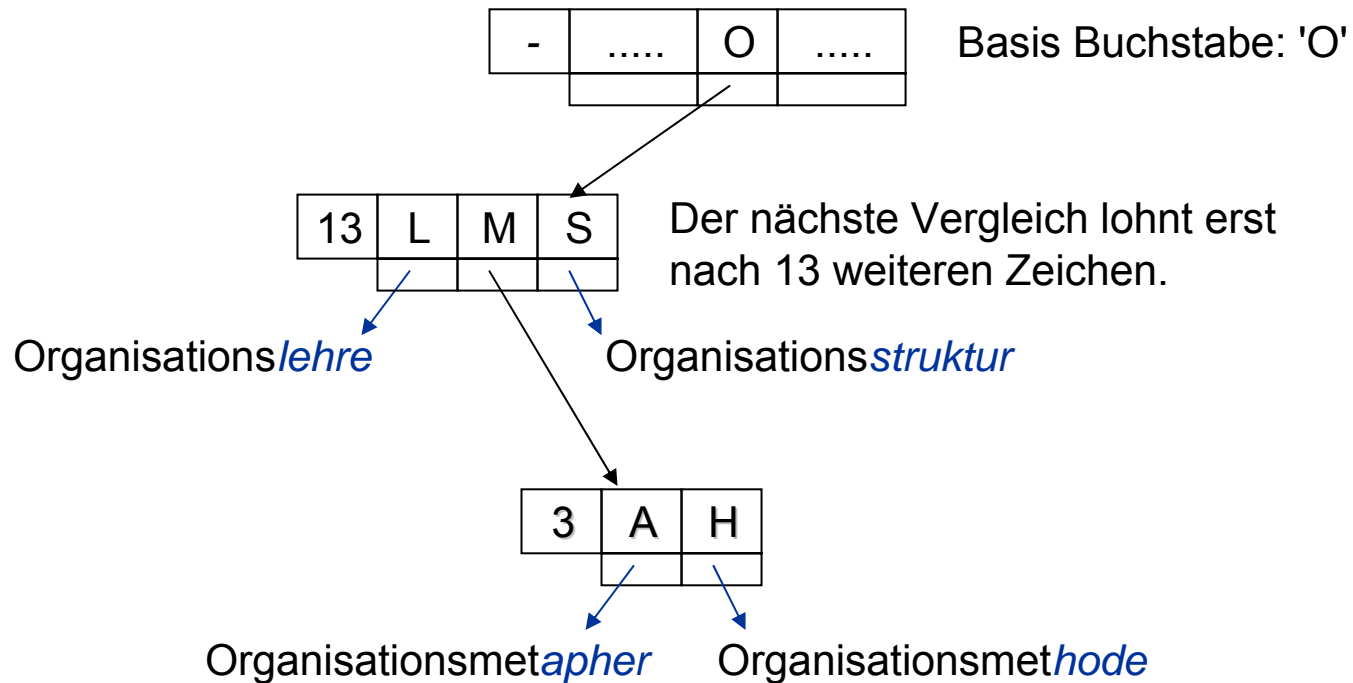
Basis ist Buchstabe 'O'

Der nächste Vergleich lohnt erst nach 13 weiteren Zeichen und nur mit den Zeichen 'L', 'M', und 'S', ..



Beispiel:

- Speicherung der Worte Organisationslehre, -struktur, -metapher und -methode.



Vorteile:

- Eine gegenüber Tries komprimierte Darstellung.
- Reduzierung der bei der Suche zu „durchwandernden“ Knoten insbesondere bei sehr langen „einsamen“ Worten.

Variante des Patricia-Baumes: der Präfix-Baum

- Zusätzlich zum Index des zu testenden Zeichens wird auch der Wert des übersprungenen Teilwortes im Knoten abgespeichert.

Beispiel (s.o.):

