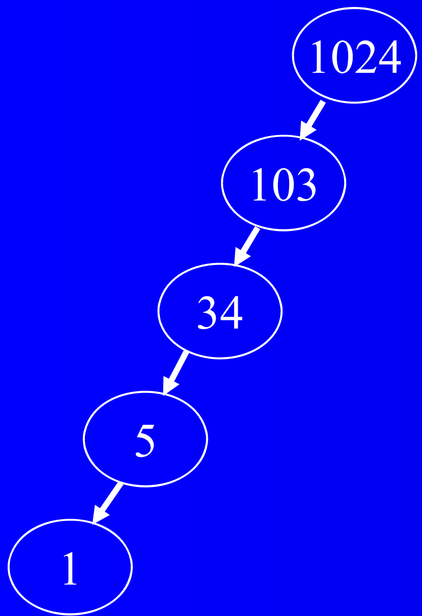
Vorlesung 7

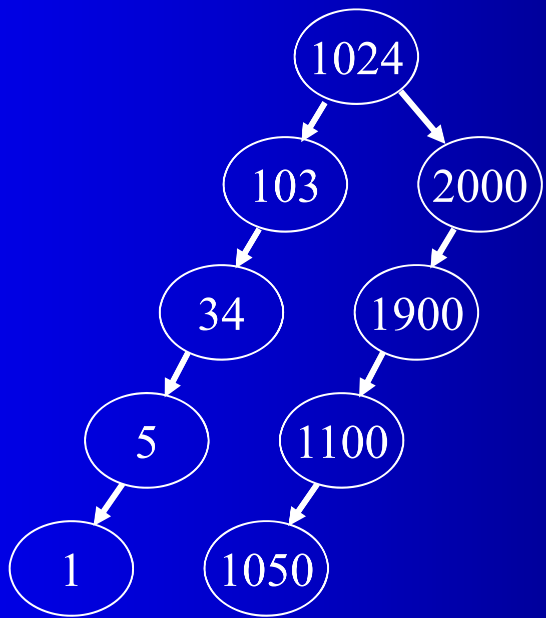
Nachteil von binären Bäumen

Die Entartung von binären Bäumen zu Listen kommt doch recht häufig vor.

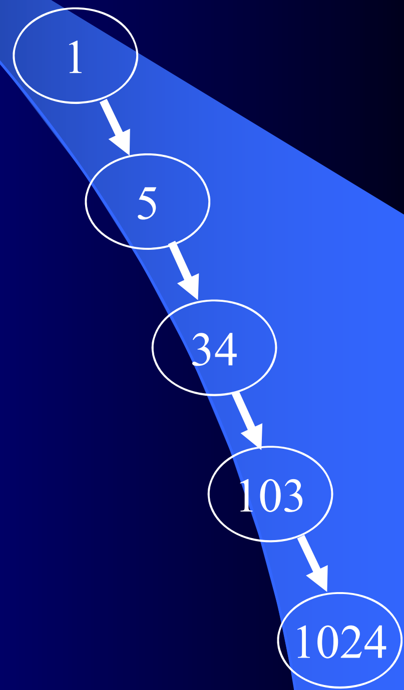
1024 103 34 5 1



1024 103 2000 34 1900 5 1100 1 1050



1 5 34 103 1024



Verbesserung von binären Bäumen

Problem der entarteten Bäume:

- ihre Tiefe ist nicht mehr logarithmisch sondern linear, da
- die Knoten (fast) immer nur einen und nicht zwei Nachfolger haben

Idee:

- Bäume ausbalanzieren

Top-Down 2-3-4-Bäume

Idee:

- statt Knoten mit 2 Nachfolgern auch welche mit 3 und 4 Nachfolgern erlauben
- dazu haben die Knoten 1, 2 bzw. 3 Schlüssel

2-Knoten

237

Zahlen < 237 Zahlen > 237

3-Knoten

237 406

Zahlen < 237 237 < Zahlen < 406 Zahlen > 406

4-Knoten

237 406 1024

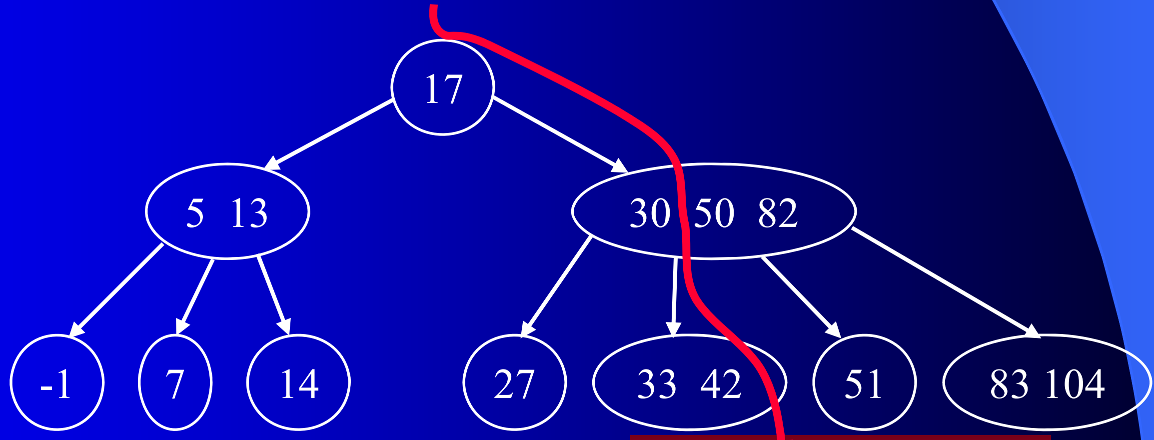
Zahlen < 237 237 < Zahlen < 406 406 < Zahlen < 1024 Zahlen > 1024

Top-Down 2-3-4-Bäume: Suchen

Idee:

- analog zu den Binärbäumen
- an jedem Knoten wird überprüft, ob der gesuchte Schlüssel der oder die (2 oder 3) abgespeicherten Schlüssel sind
- wenn nicht, wird in den entsprechenden Ast abgestiegen
- unten an einem Blatt kann dann entschieden werden, ob das Gesuchte vorhanden ist

suchen nach 47

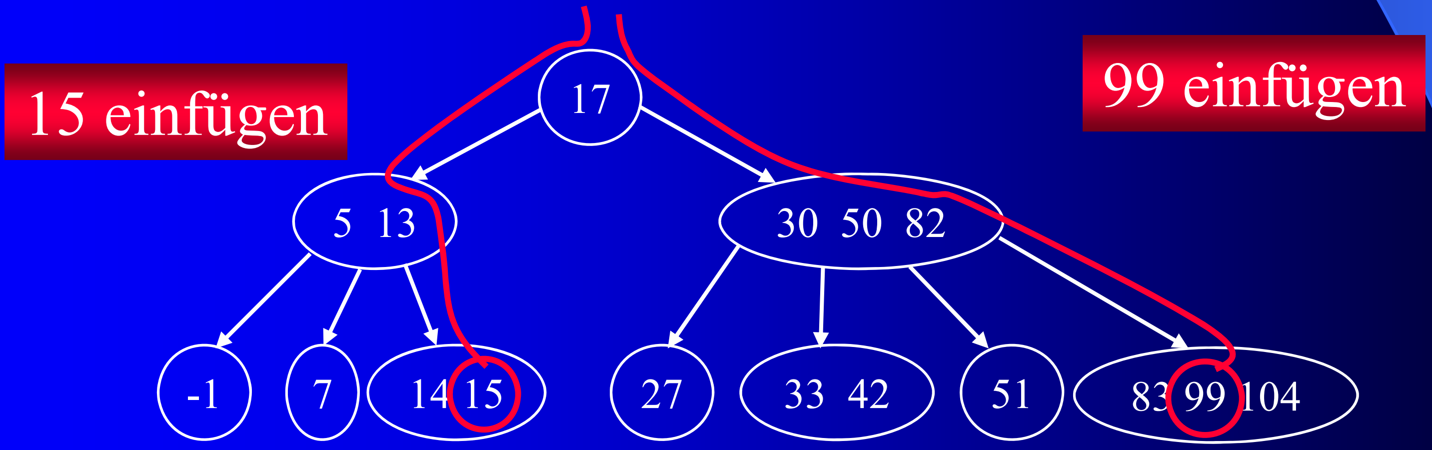


nicht gefunden

Top-Down 2-3-4-Bäume: Einfügen

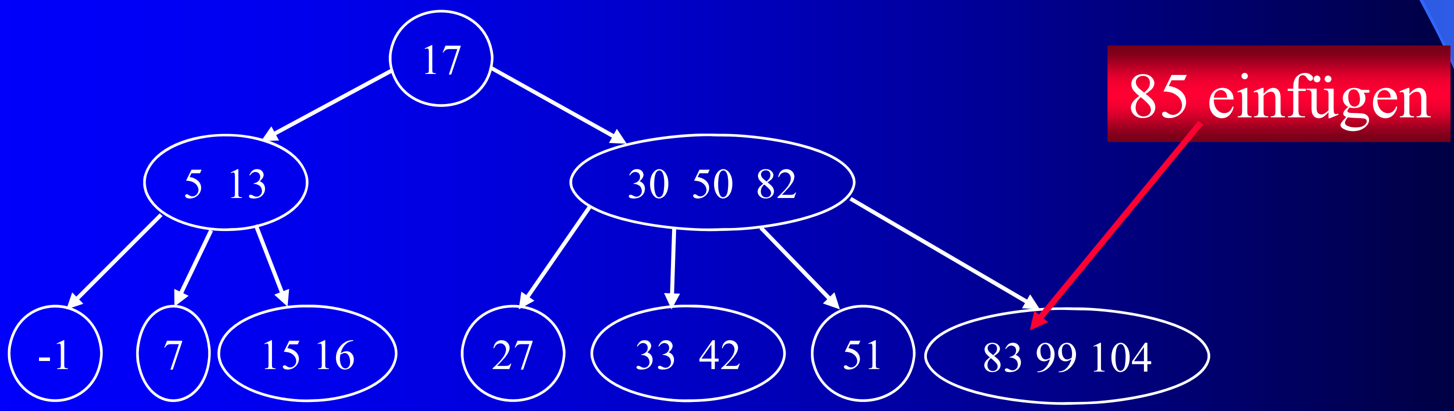
Idee:

- analog zu den Binärbäumen
- es wird bis zu einem Blatt abgestiegen
- wenn es sich um ein 2-Knoten oder 3-Knoten Blatt handelt, kann direkt der neue Schlüssel eingefügt werden
- aus dem 2-Knoten Blatt wird ein 3-Knoten Blatt
- aus dem 3-Knoten Blatt wird ein 4-Knoten Blatt

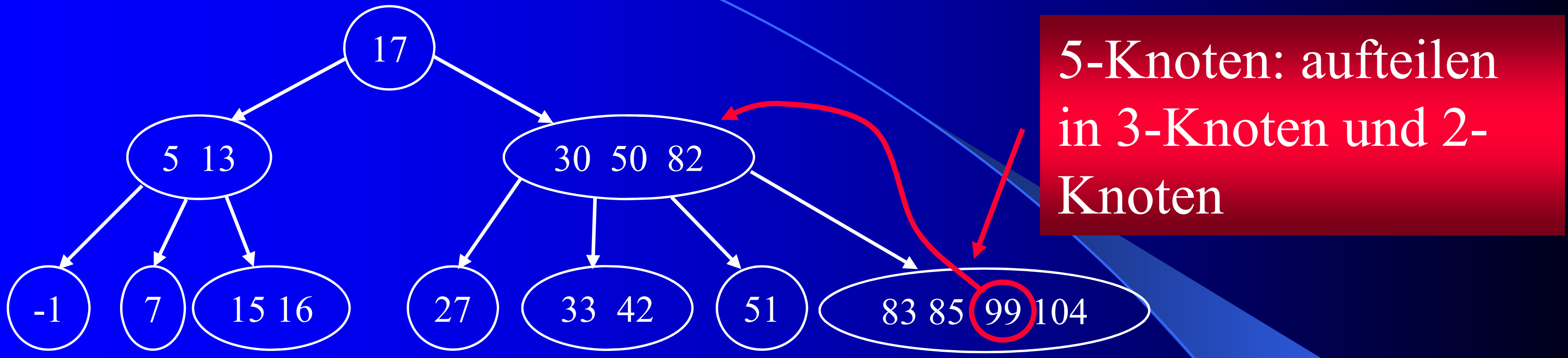


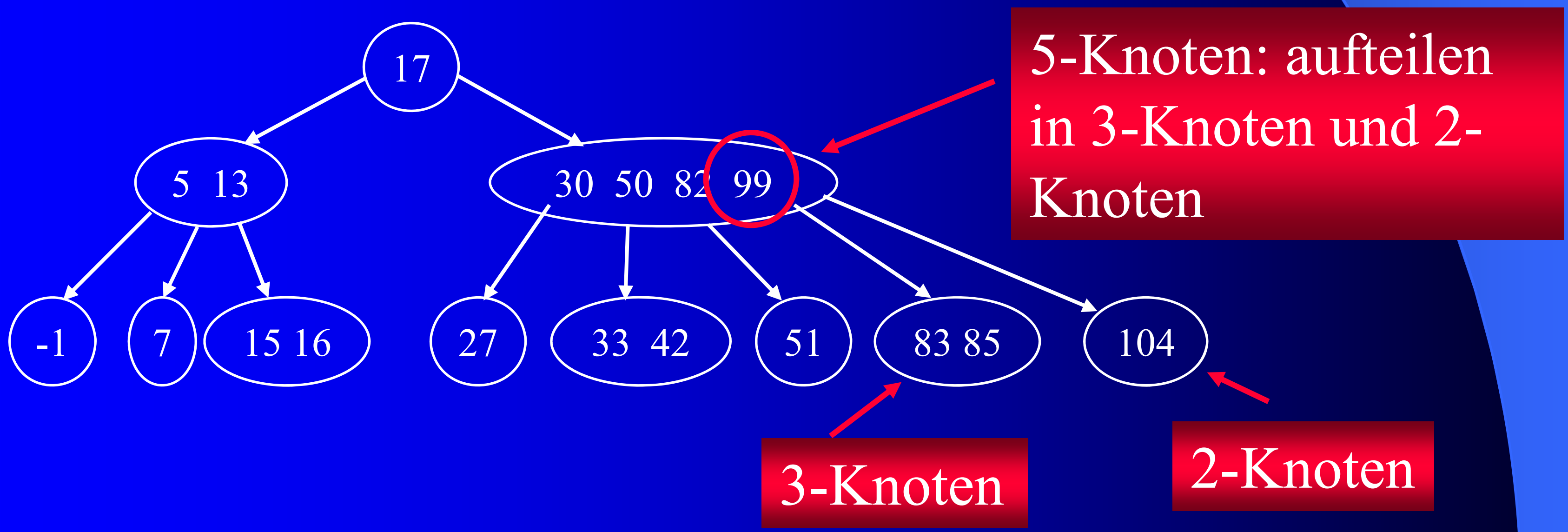
Top-Down 2-3-4-Bäume: Einfügen (Fort.)

- muss in einem 4-Knoten Blatt eingefügt werden (es müsste ein 5-Knoten entstehen), so wird er in ein 3-Knoten und ein 2-Knoten aufgeteilt
- dadurch bekommt der Vater einen Knoten mehr
- dadurch muss der Vater (und rekursiv dessen Vater usw.) u.U. ebenfalls neu aufgeteilt werden

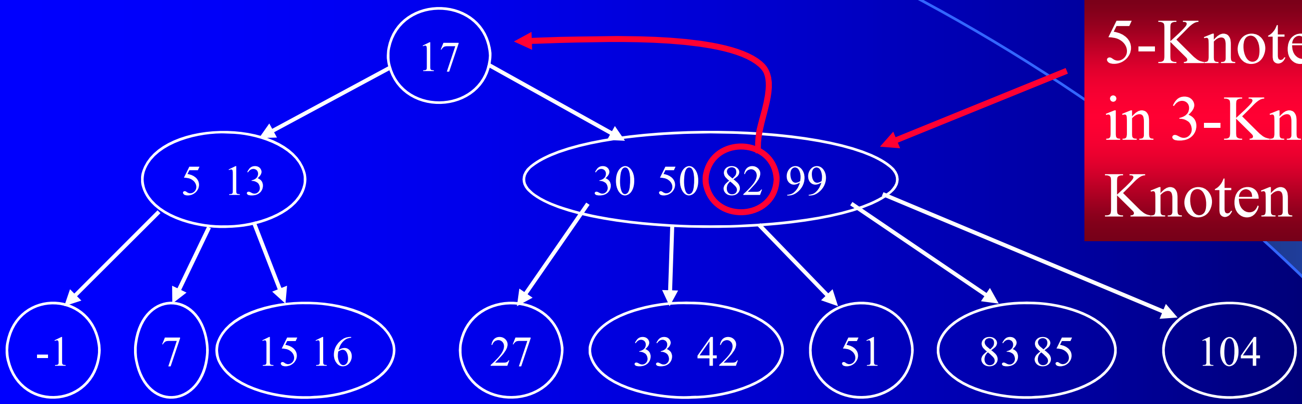


Top-Down 2-3-4-Bäume: Einfügen (Fort.)

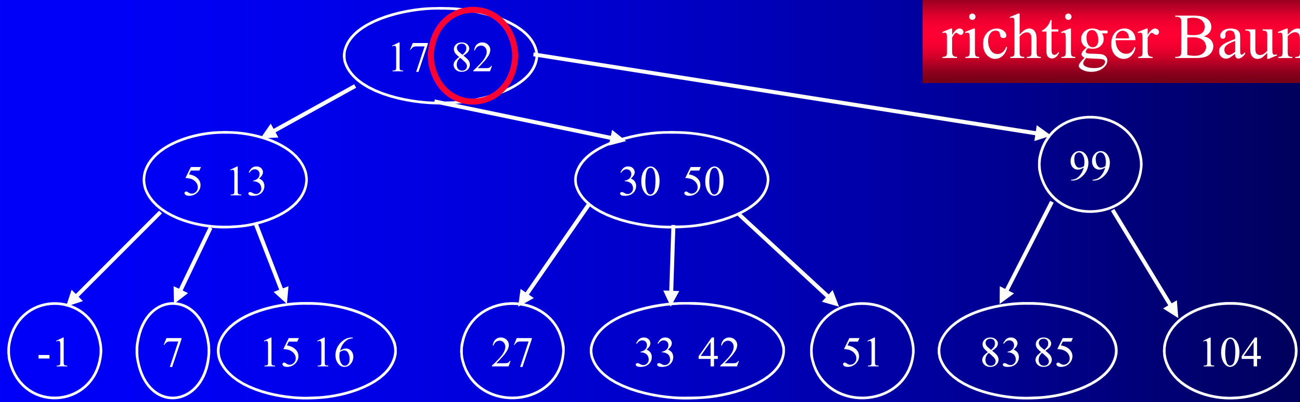




Top-Down 2-3-4-Bäume: Einfügen (Fort.)



5-Knoten: aufteilen in 3-Knoten und 2-Knoten

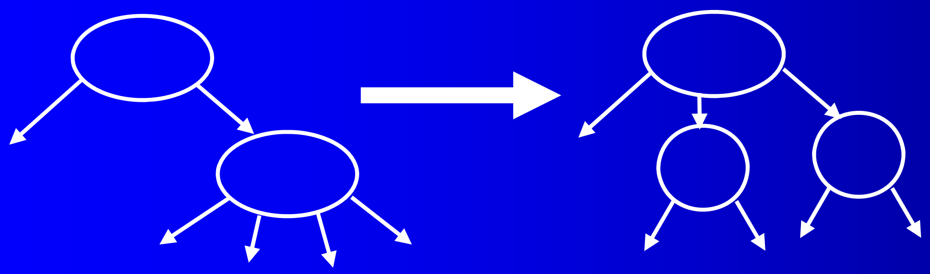


richtiger Baum

Top-Down 2-3-4-Bäume: Einfügen (Fort.)

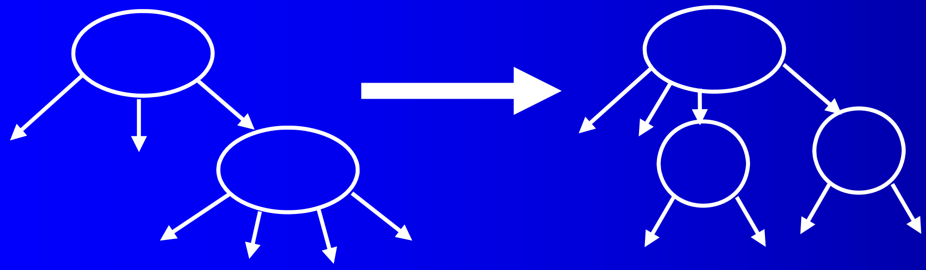
Optimierung:

- nicht erst beim Einfügen nach oben laufen und alle 4-Knoten aufspalten, sondern
- beim Abstieg alle 4-Knoten aufspalten, somit
- hat kein Knoten ein 4-Knoten Vorgänger und
- kann sofort aufgespaltet werden
- dazu folgende Regeln beim Abstieg:



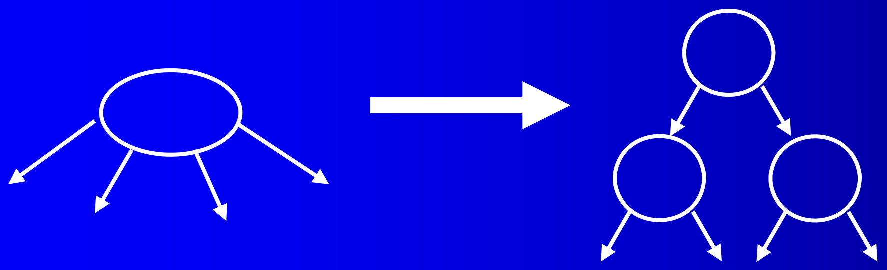
als einem 2-Knoten mit 4-Knoten Nachfolger wird ein 3-Knoten mit 2 2-Knoten Nachfolgern

Top-Down 2-3-4-Bäume: Einfügen (Fort.)



als einem 3-Knoten mit 4-Knoten Nachfolger wird ein 4-Knoten mit 2 2-Knoten Nachfolgern

Spezialfall: 4-Knoten Wurzel



4-Knoten Wurzel in 3 2-Knoten aufteilen; dadurch gewinnt der Baum an Höhe

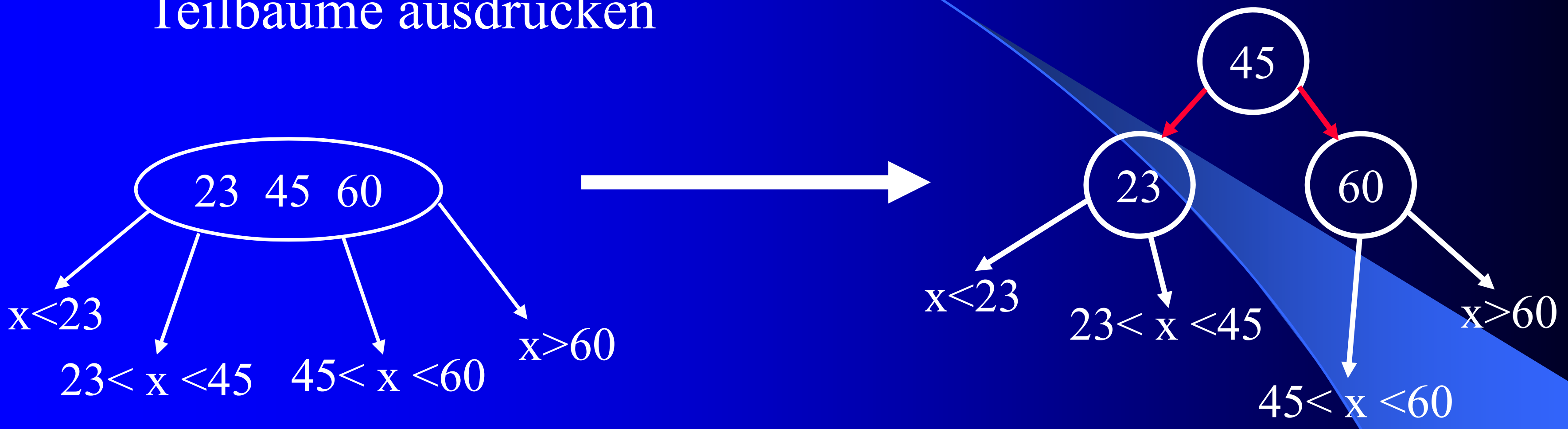
Top-Down 2-3-4-Bäume: Eigenschaften

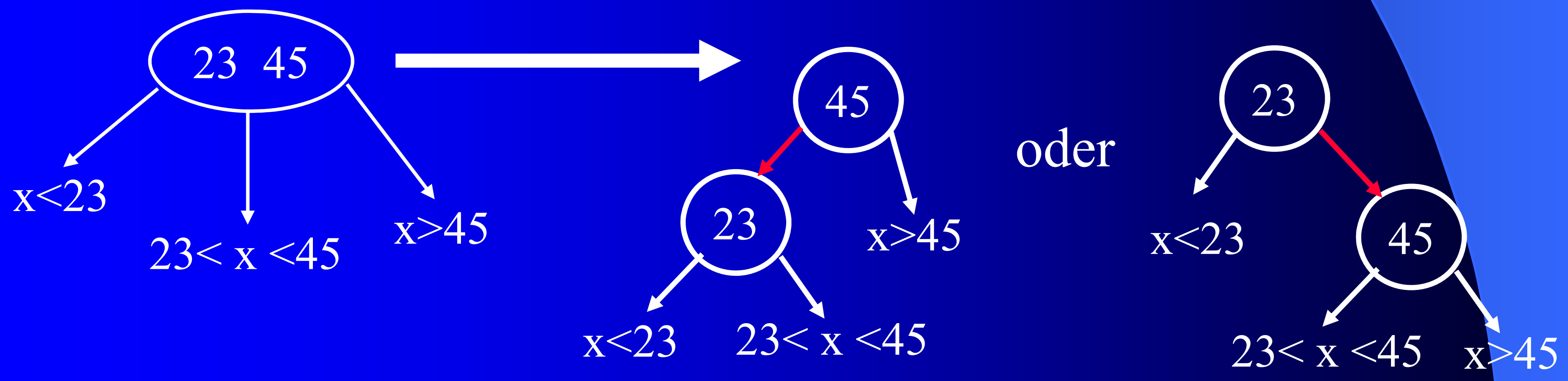
- da der Baum nur an der Wurzel wachsen kann, ist er immer ausgeglichen
- dadurch liegt das Suchen in O(log N)
- das Einfügen liegt garantiert in O(log N)

- gemäß Sedgewick ist es nicht ganz trivial, diesen Algorithmus zu implementieren, daher ...

Rot-Schwarz Bäume

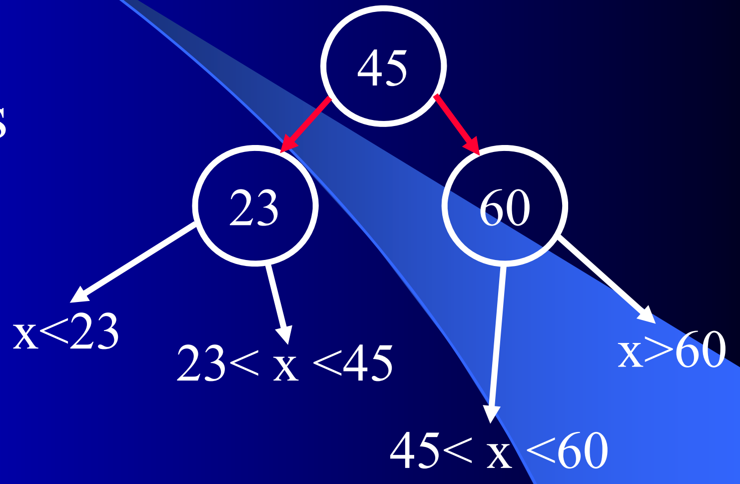
- 3-Knoten und 4-Knoten lassen sich auch durch binäre Teilbäume ausdrücken





Rot-Schwarz Bäume (Fort.)

- jeder 3-Knoten bzw. 4-Knoten lässt sich durch einen binären Teilbaum darstellen
- die Tiefe eines solchen Baums ist maximal 2-mal so groß wie die eines Top-down 2-3-4 Baums
- die roten Kanten dienen nur der Darstellung von 3- bzw. 4-Knoten
- die anderen Kanten dienen der Verkettung
- daher heißen diese Bäume rot-schwarz Bäume
- nach einer roten Kante folgt immer eine schwarze Kante !!!!



Rot-Schwarz Bäume: Implementierung

- jeder Knoten bekommt zusätzlich ein boolesches Flag
- ist dieses Flag true, so ist die Kante rot, die zu diesem Knoten führt
- ansonsten ist die Kante schwarz

```
public class BlackRedTree<K extends Comparable<K>,D> {
    class Node {
        public Node(K key,D data) {
            m_Key = key;
            m_Data = data;
        }
        K m_Key;
        D m_Data;
        Node m_Left = null;
        Node m_Right = null;
        boolean m_bIsRed = true;
    }
...
    private Node m_Root = null;
...
```

Flag, das die Kantenfarbe anzeigt

Rot-Schwarz Bäume: Implementierung (Fort.)

- das Suchen in einem Rot-Schwarz Baum schaut sich niemals die Kantenfarbe an
- daher kann die search Methode von BinTree unverändert übernommen werden

```
public Node search(K key) {
    Node tmp = m_Root;
    while (tmp != null) {
        final int RES = key.compareTo(tmp.m_Key);
        if (RES == 0)
            return tmp;
        tmp = RES < 0  ? tmp.m_Left : tmp.m_Right;
    }
    return null;
}
```

wenn der Schlüssel gefunden ist, gibt den Datensatz zurück

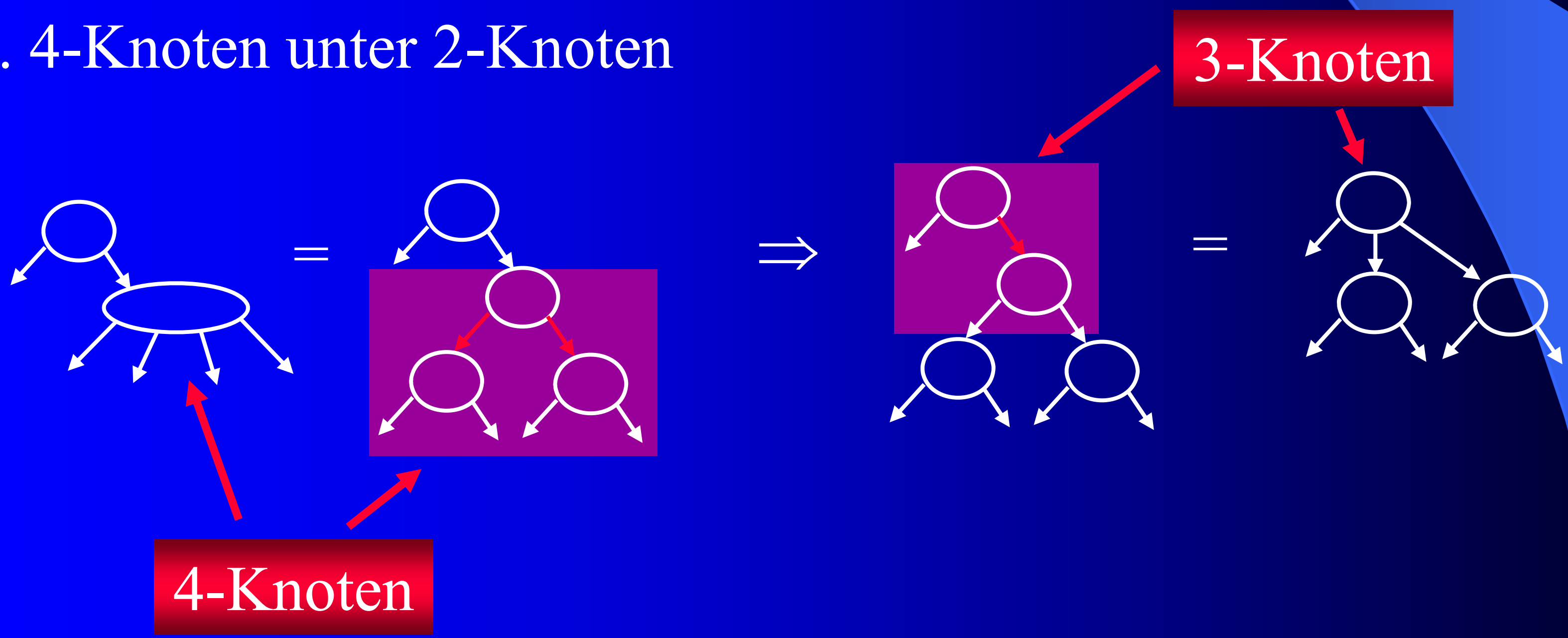
steige links bzw. rechts ab

Schlüssel ist nicht gefunden worden

Rot-Schwarz Bäume: Implementierung (Fort.)

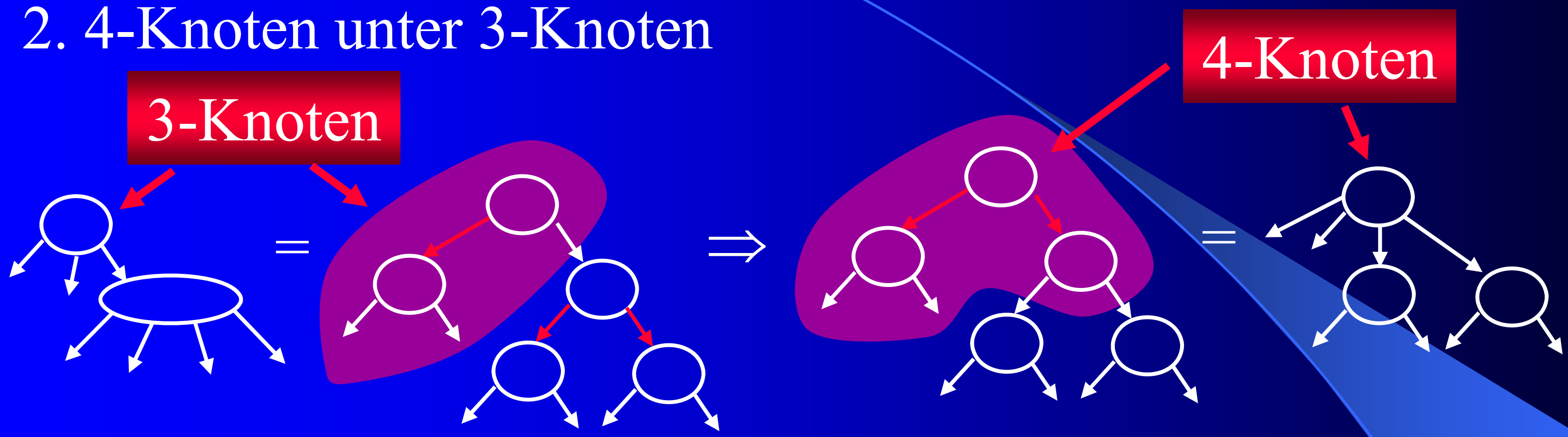
- beim Einfügen werden alle 4-Knoten aufgeteilt
- ein 4-Knoten erkennt man daran, dass beide Nachfolger-knoten das gesetzte Flag haben
- nicht sehr teuer, da es kaum 4-Knoten gibt
- es gibt 7 Fälle zu untersuchen

1. 4-Knoten unter 2-Knoten

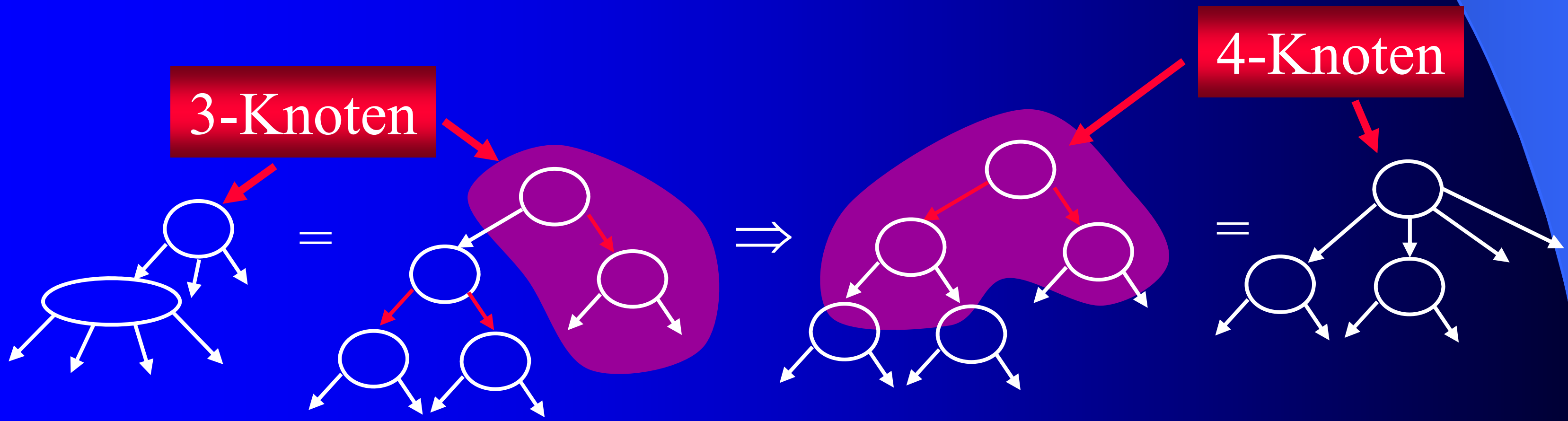


Rot-Schwarz Bäume: Implementierung (Fort.)

2. 4-Knoten unter 3-Knoten

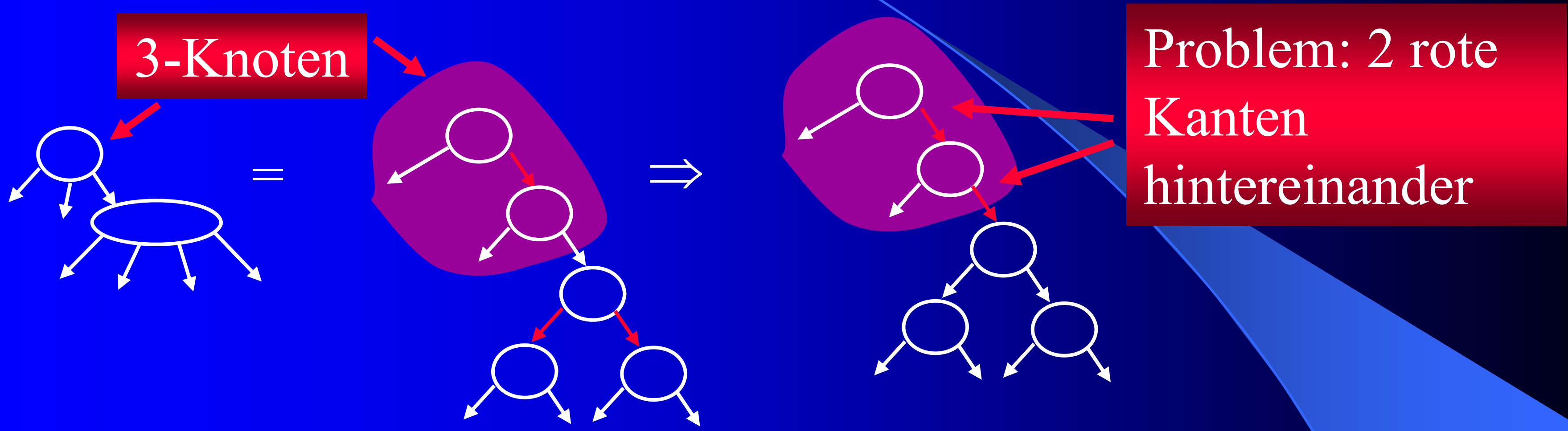


3. 4-Knoten unter 3-Knoten

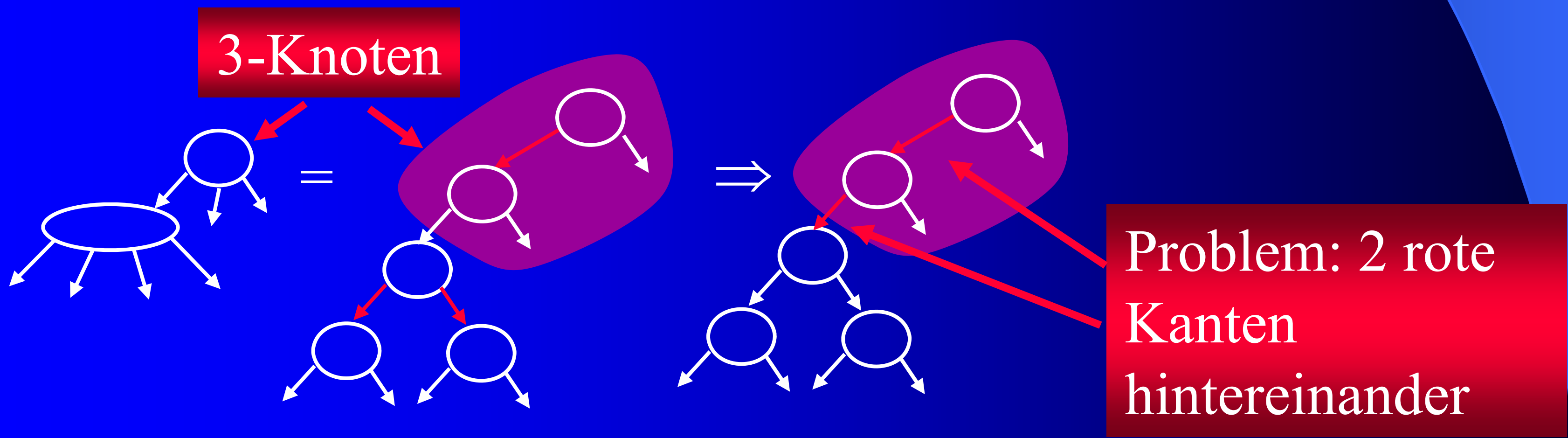


Rot-Schwarz Bäume: Implementierung (Fort.)

4. 4-Knoten unter 3-Knoten

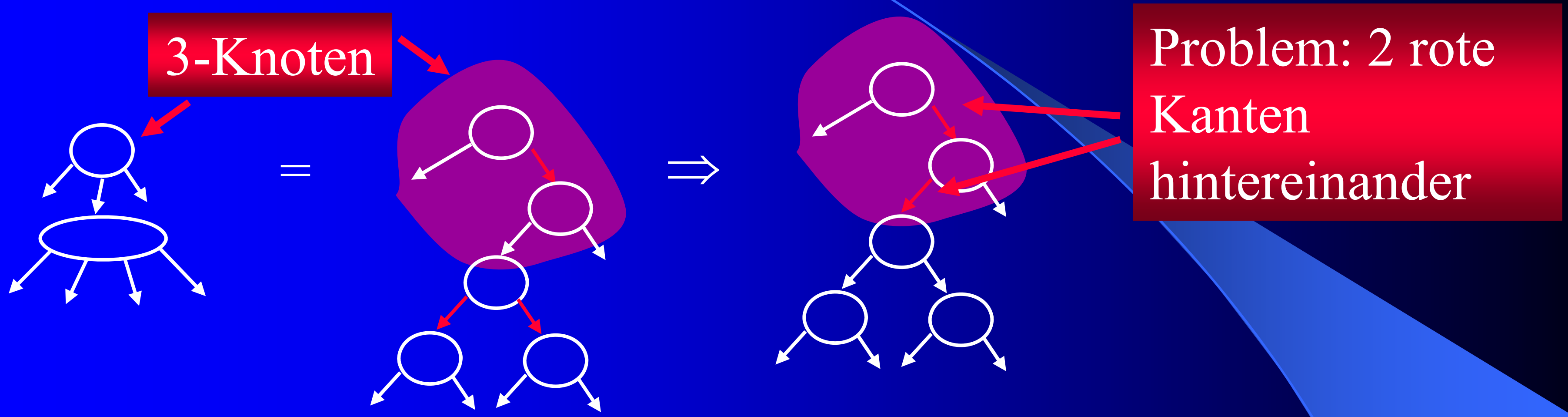


5. 4-Knoten unter 3-Knoten

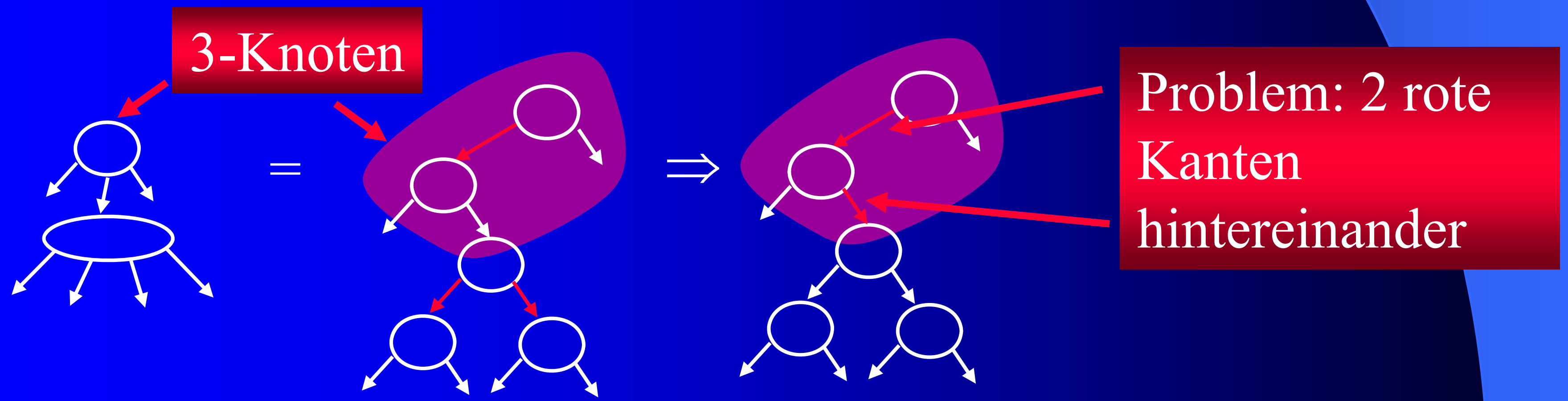


Rot-Schwarz Bäume: Implementierung (Fort.)

6. 4-Knoten unter 3-Knoten



7. 4-Knoten unter 3-Knoten

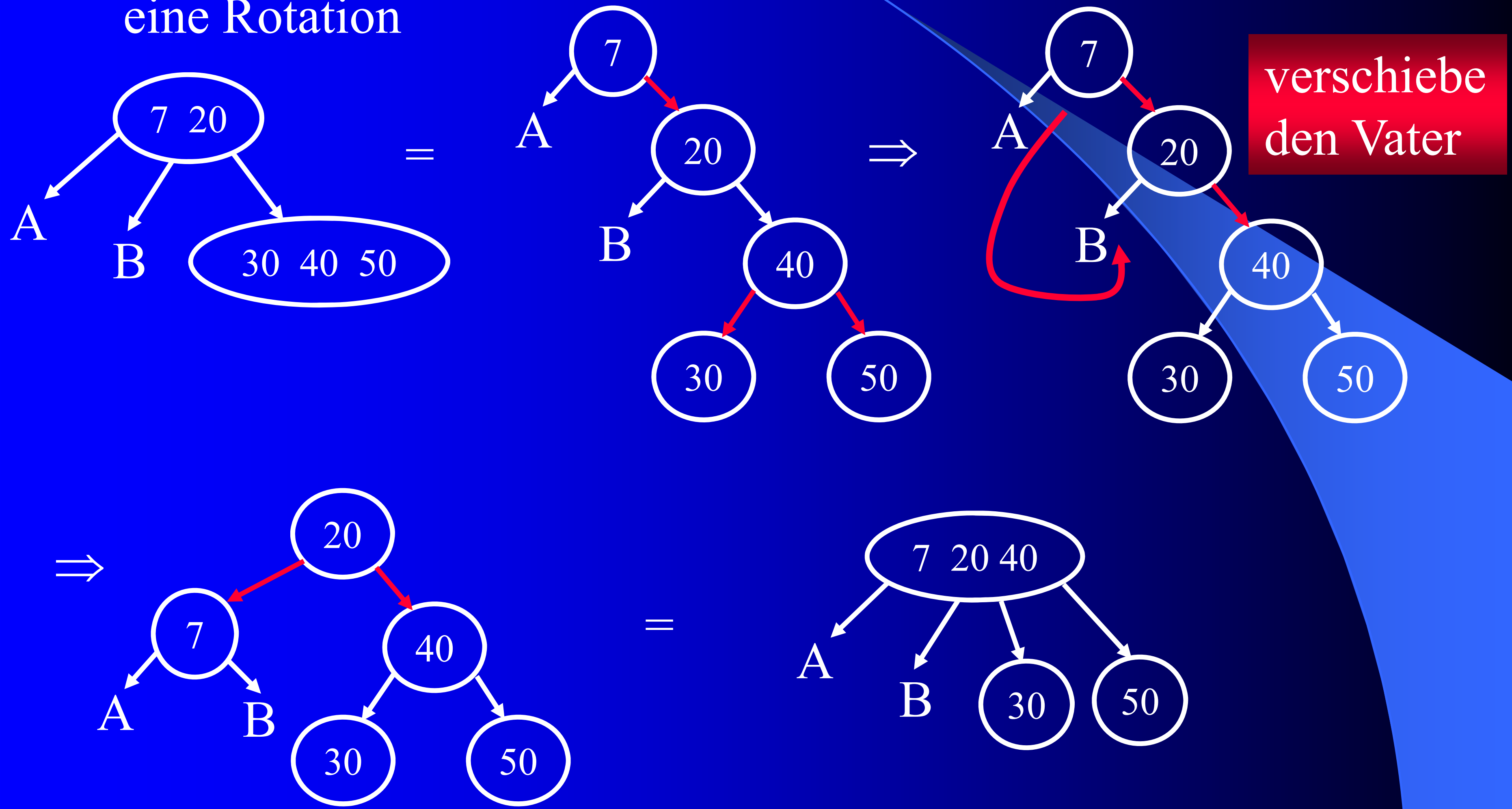


Rot-Schwarz Bäume: Implementierung (Fort.)

- Problem in Fall 4 und 5: die Ausrichtung der 3-Knoten war nicht richtig
- mit der richtigen Ausrichtung sind es dann die Fälle 2 bzw. 3
- Problem in Fall 6 und 7: hier kann eine andere Ausrichtung nichts bewirken
- andere Lösung ist gefragt

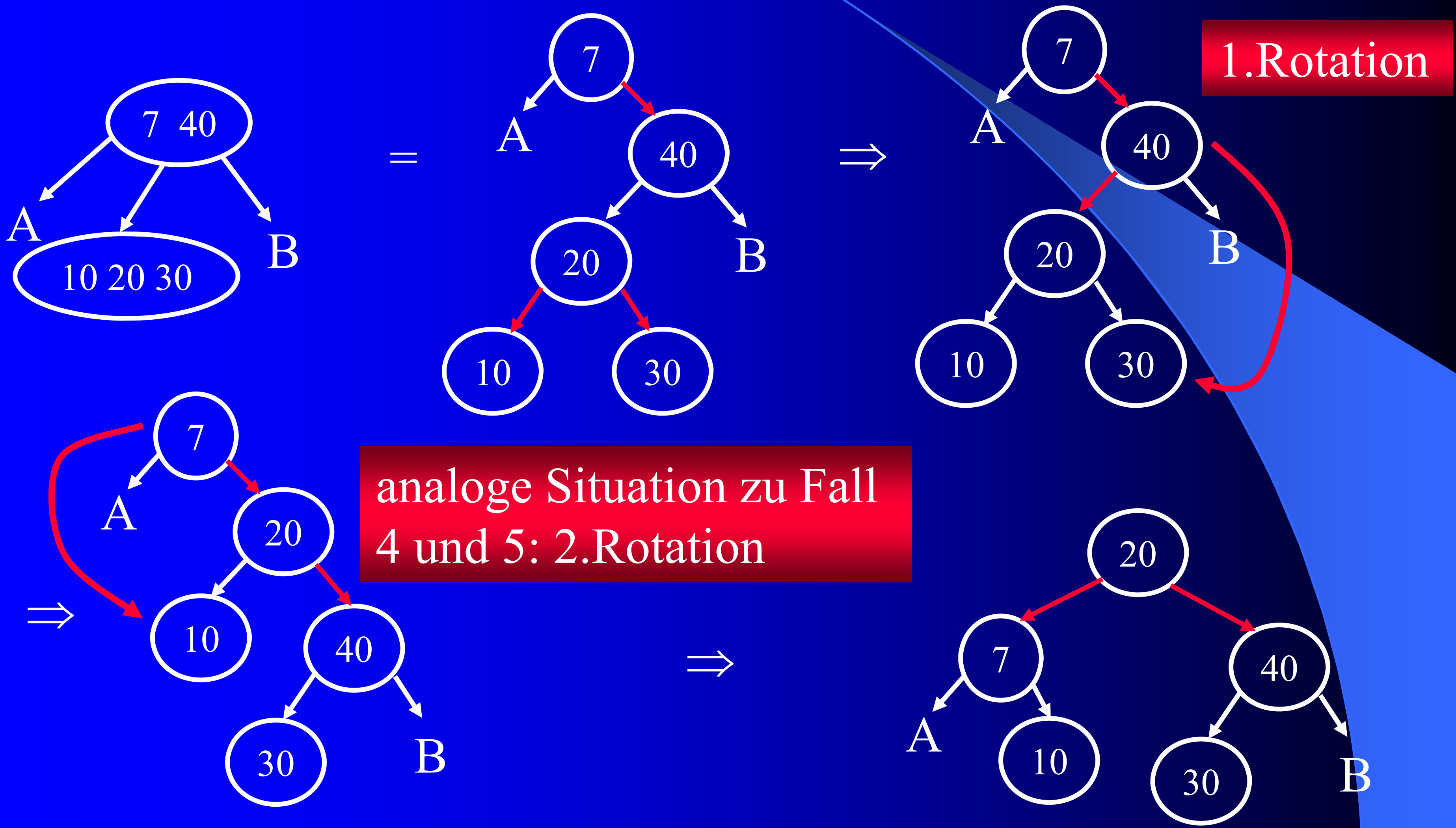
Rot-Schwarz Bäume: Implementierung (Fort.)

- Lösung für falsche Ausrichtung (Fall 4 und analog Fall 5): eine Rotation



Rot-Schwarz Bäume: Implementierung (Fort.)

- Lösung für Fall 6 und 7: zwei Rotationen



Vorlesung 8

Rot-Schwarz Bäume: Implementierung

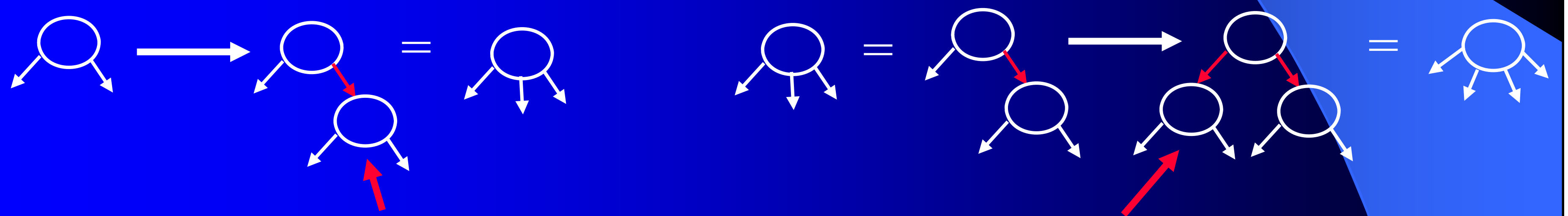
- die Knoten sind analog zu den binären Bäumen
- sie erhalten zusätzlich ein boolesches Flag, dass anzeigt, ob die ***hinführende Kante rot*** ist

```
class Node {
    public Node(K key,D data) {
        m_Key = key;
        m_Data = data;
    }

    K m_Key;
    D m_Data;
    Node m_Left = null;
    Node m_Right = null;
    boolean m_bIsRed = true;
}
```

ist die hinführende Kante rot?

Rot-Schwarz Bäume: Implementierung (Fort.)

- Situation: ein neuer Knoten wird in den Baum unten an das Ende angefügt
- 2 Fälle:
  - mache aus einem 2-Knoten einen 3-Knoten
  - mache aus einem 3-Knoten einen 4-Knoten

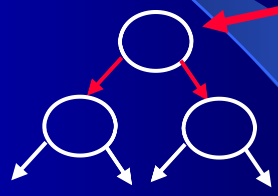


neuer Knoten: hinführende Kante ist rot

Rot-Schwarz Bäume: Implementierung (Fort.)

- ein Knoten kann selber erkennen, wann er ein 4-Knoten ist
- er hat dann 2 rote Nachfolger

ein 4-Knoten

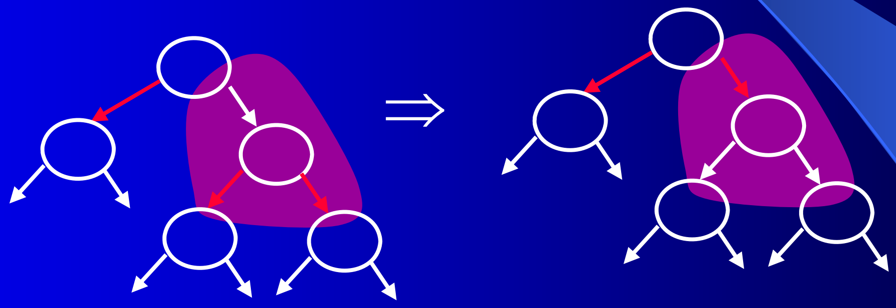


```
class Node {
...

    public boolean is4Node() {
        return m_Left != null && m_Left.m_bIsRed
            && m_Right != null && m_Right.m_bIsRed;
    }

...
```

ein 4-Knoten hat einen roten linken und einen roten rechten Nachfolger

Rot-Schwarz Bäume: Implementierung (Fort.)

- ein 4-Knoten wird konvertiert, indem die roten Kanten entfernt werden und die hinführende Kante rot eingefärbt wird



```
class Node {
...
    void convert4Node() {
        m_Left.m_bIsRed = false;
        m_Right.m_bIsRed = false;
        m_bIsRed = true;
    }
...
```

färbe die Nachfolger-kanten schwarz

die eigene Kante wird rot

Rot-Schwarz Bäume: Implementierung (Fort.)

- gesucht wird in einem Rot-Schwarz Baum wie in einem Binärbaum
- die Kantenfarbe wird einfach ignoriert

```
public class RedBlackTree<K extends Comparable<K>,D> {
...

    public Node search(K key) {
        Node tmp = m_Root;
        while (tmp != null) {
            final int RES = key.compareTo(tmp.m_Key);
            if (RES == 0)
                return tmp;
            tmp = RES < 0 ? tmp.m_Left : tmp.m_Right;
        }
        return null;
    }
...
```

Schlüssel gefunden?

iterativer Abstieg nach links bzw. rechts

Rot-Schwarz Bäume: Implementierung (Fort.)

- das Einfügen wird aus der insert-Methode der Binärbäume gewonnen

NodeHandler merkt sich aktuellen und Vorgängerknoten

```
boolean insert(K key,D data) {
    NodeHandler h = new NodeHandler(m_Root);
    while (!h.isNull()) {
        final int RES = key.compareTo(h.node().m_Key);
        if (RES == 0)
            return false;
        h.down(RES < 0);
    }
    h.set(new Node(key,data));
    m_Root.m_bIsRed = false;
    return true;
}
```

Schlüssel ist bereits eingetragen

die Wurzel soll nie ein 4-Knoten sein

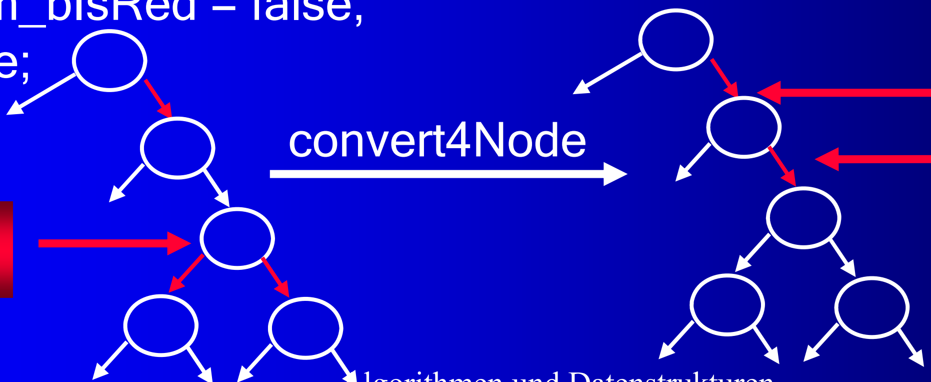
Rot-Schwarz Bäume: Implementierung (Fort.)

- beim Abstieg sollen alle 4-Knoten aufgeteilt werden

```
boolean insert(K key,D data) {
    NodeHandler h = new NodeHandler(m_Root);
    while (!h.isNull()) {
        if (h.node().is4Node()) {
            h.node().convert4Node();
        }
        final int RES = key.compareTo(h.node().m_Key);
        if (RES == 0)
            return false;
        h.down(RES < 0);
    }
    h.set(new Node(key,data));
    m_Root.m_bIsRed = false;
    return true;
}
```

Ist es ein 4-Knoten? Wenn ja, verschiebe die Kantenfarbe

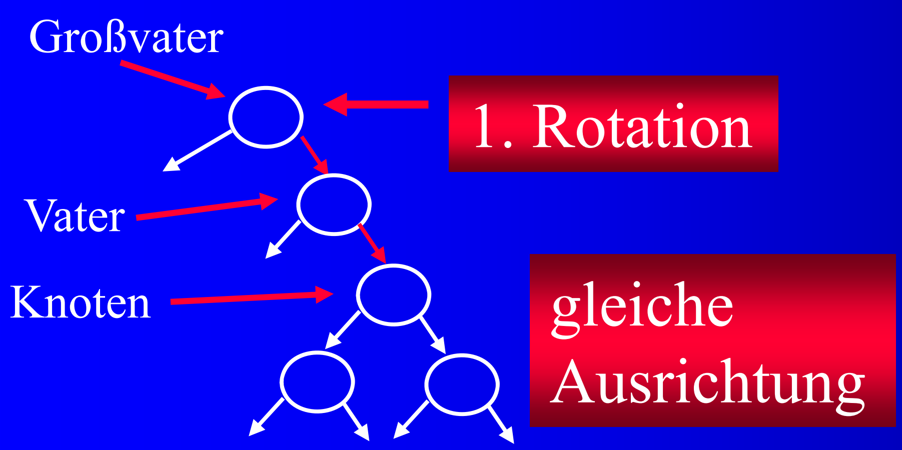
4-Knoten

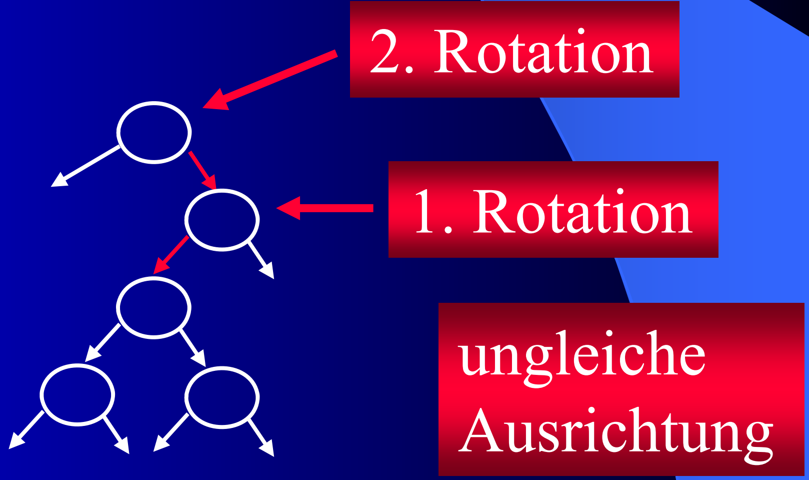


dabei entstehen Probleme: 2 rote Kanten nacheinander!

Rot-Schwarz Bäume: Implementierung (Fort.)

- Aufgaben bei 2 roten Kanten hintereinander:
- Situation erkennen, d.h. führt zum Vater eine rote Kante
- erkennen, ob beide Kanten gleiche Ausrichtung haben
- bei gleicher Ausrichtung: eine Rotation
- bei ungleicher Ausrichtung: zwei Rotationen





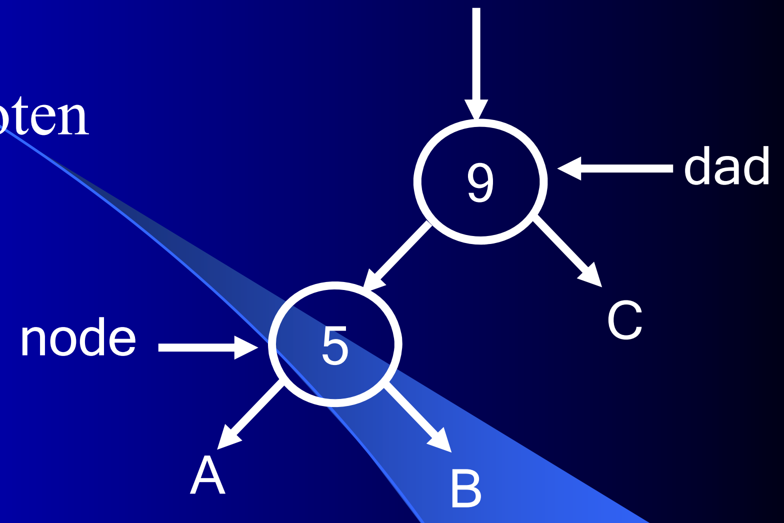
Rot-Schwarz Bäume: Implementierung (Fort.)

- Rotation: Vater und Sohn vertauschen ihre Plätze
- 2 Situationen: Links- und Rechtsdrehung

- für eine Drehung benötigt man die beiden Knoten ***und*** die Stelle, an der der Vater gespeichert ist
- danach haben der Vater und der Sohn die Farben getauscht

Rot-Schwarz Bäume: Implementierung (Fort.)

- **<u>unvollständige</u>** Rotation zweier Knoten



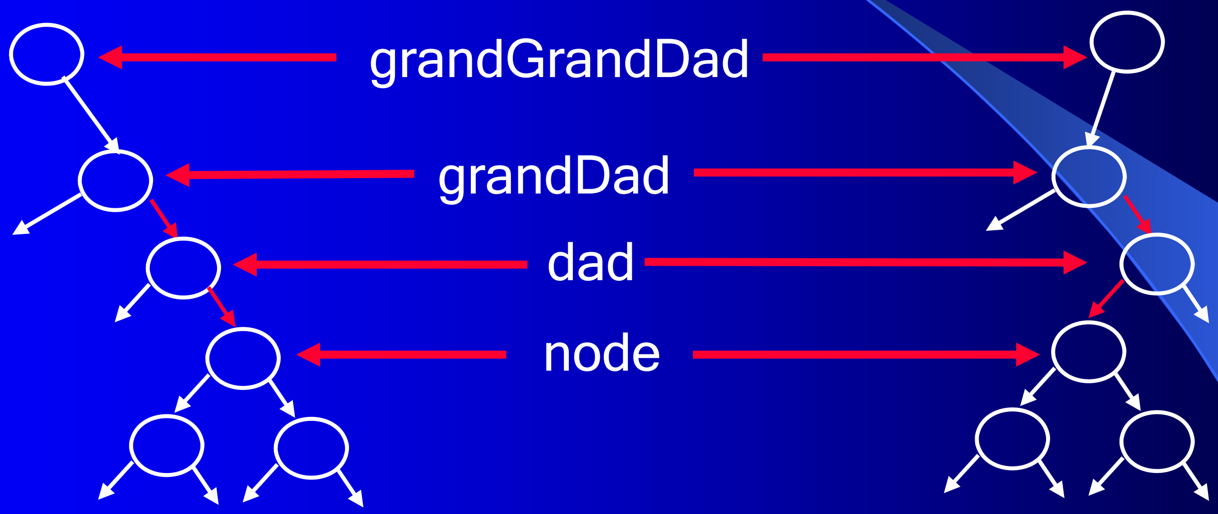

```
void rotate(Node dad,Node node) {
      boolean nodeColour = node.m_bIsRed;
      node.m_bIsRed = dad.m_bIsRed;
      dad.m_bIsRed = nodeColour;
      if (dad.m_Left == node) {
            // clockwise rotation
            dad.m_Left = node.m_Right;
            node.m_Right = dad;
      } else {
            // counter-clockwise rotation
            dad.m_Right = node.m_Left;
            node.m_Left = dad;
      }
      // ???? wer merkt sich den neuen Vater????
}
```

Vater und Sohn vertauschen die Farben

hier fehlt etwas: der Großvater müsste sich den Sohn merken
⇒NodeHandler für dad müsste übergeben werden

Rot-Schwarz Bäume: Implementierung (Fort.)

- Situation nach dem Konvertieren eines 4-Knoten



- neben dem eigentlichen Knoten (node) muss der Vaterverweis (dad) und der Großvaterverweis (grandDad) und der Urgroßvater (grandGrandDad) gemerkt werden, da
- die oberste Rotation den Urgroßvater betrifft (merkt sich einen neuen Großvater)

Rot-Schwarz Bäume: Der NodeHandler

- NodeHandler muss sich auch die weiteren Vorgänger merken

```
class NodeHandler {
      public final int NODE = 0;
      public final int DAD = 1;
      public final int G_DAD = 2;
      public final int GG_DAD = 3;

      private Object[] m_Nodes = new Object[4];

      NodeHandler(Node n) {
            m_Nodes[NODE] = n;
      }

      void down(boolean left) {
            for(int i = m_Nodes.length-1;i >0;--i)
                        m_Nodes[i] = m_Nodes[i-1];
            m_Nodes[NODE] = left ? node(DAD).m_Left : node(DAD).m_Right;
      }
```

Konstanten für die Indizes

Array für 4 Knoten: node, dad, grandDad, grandGrandDad

es fängt immer mit node an

beim Abstieg werden alle um eine Position verschoben

Rot-Schwarz Bäume: Der NodeHandler (Fort.)

```
boolean isNull() {
    return m_Nodes[NODE] == null;
}

Node node(int kind) {
    return (Node)m_Nodes[kind];
}

void set(Node n,int kind) {
    if (node(kind+1) == null)
            m_Root = n;
    else if (node(kind) != null ?
            node(kind+1).m_Left == node(kind) :
            n.m_Key.compareTo(node(kind+1).m_Key) < 0)
            node(kind+1).m_Left = n;
    else
            node(kind+1).m_Right = n;
    m_Nodes[kind] = n;
}
```

existiert noch der unterste Knoten?

Zugriff auf einen beliebigen Knoten mittels Index

setzen der Wurzel, wenn Baum leer ist

Wird für remove benötigt, da n gleich null werden kann

Setzen unter dem linken oder rechten Vater

Rot-Schwarz Bäume: Der NodeHandler (Fort.)

kind ist der Index des Vaters, um den rotiert werden soll

```
void rotate(int kind) {
    Node dad = node(kind);
    Node son = node(kind-1);
    boolean sonColour = son.m_bIsRed;
    son.m_bIsRed = dad.m_bIsRed;
    dad.m_bIsRed = sonColour;
    // rotate
    if (dad.m_Left == son) {
        // clockwise rotation
        dad.m_Left = son.m_Right;
        son.m_Right = dad;
    } else {
        // counter-clockwise rotation
        dad.m_Right = son.m_Left;
        son.m_Left = dad;
    }
    set(son,kind);
}
```

Vater und Sohn vertauschen die Farben

Vater und Sohn vertauschen die Plätze

Sohn nimmt den Platz des Vaters im NodeHandler ein

Rot-Schwarz Bäume: Implementierung (Fort.)

- insert Methode mit dem neuen NodeHandler

```
boolean insert(K key,D data) {
    NodeHandler h = new NodeHandler(m_Root);
    while (!h.isNull()) {
        if (h.node(h.NODE).is4Node()) {
            h.node(h.NODE).convert4Node();
            h.split();
        }
        final int RES = key.compareTo(h.node(h.NODE).m_Key);
        if (RES == 0)
            return false;
        h.down(RES < 0);
    }
    h.set(new Node(key,data),h.NODE);
    h.split();
    m_Root.m_bIsRed = false;
    return true;
}
```

beim Zugriff auf NodeHandler muss der Index mit angegeben werden

nach der Konvertierung muss der Teilbaum u.U. rotiert werden

auch beim Einfügen kann der Baum durcheinanderkommen

Rot-Schwarz Bäume: Implementierung (Fort.)

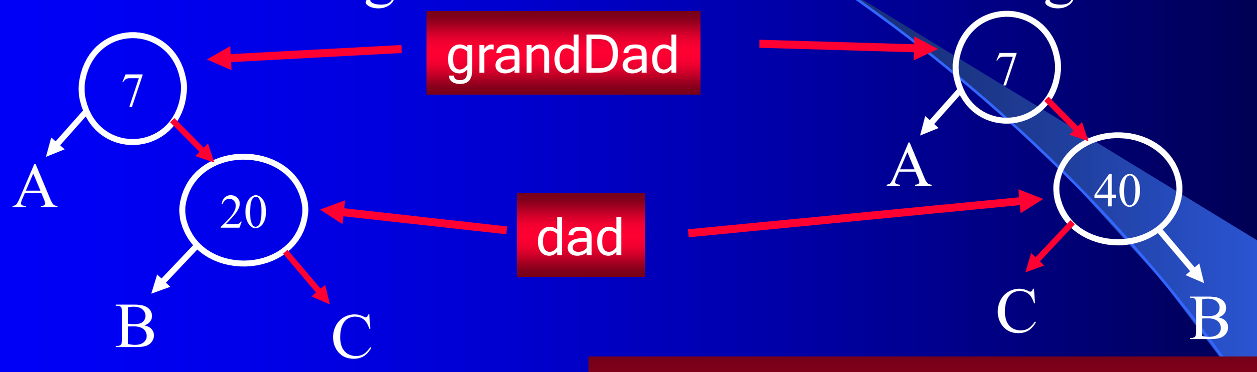
- die split Methode ist eine Methode des NodeHandlers
- sie wird nur von Knoten mit roten Kanten aufgerufen
- wenn der Vater existiert und auch rot ist, muss rotiert werden

```
private void split() {
    Node dad = node(DAD);
    if (dad != null && dad.m_bIsRed) {
        ...
    }
}
```

gibt es einen Vater und ist der rot?

Rot-Schwarz Bäume: Implementierung (Fort.)

- diese beiden Fälle müssen unterschieden werden
- ist die Ausrichtung der beiden roten Kanten gleich ?



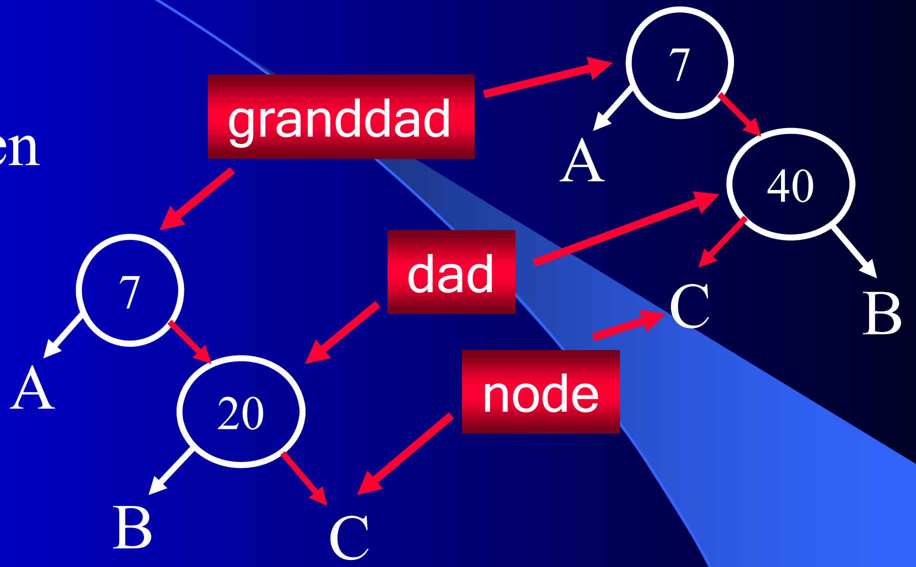
wenn es einen roten Vater gibt, muss es einen Großvater geben (weil, die Wurzel ist niemals rot)

```
private void split() {
    Node dad = node(DAD);
    if (dad != null && dad.m_bIsRed) {
        if ( node(G_DAD).m_Key.compareTo(dad.m_Key) < 0 !=
             dad.m_Key.compareTo(node(NODE).m_Key) < 0)
        ...
    }
}
```

ist das Schlüsselverhältnis Großvater ↔ Vater anders als Vater ↔ Sohn

Rot-Schwarz Bäume: Implementierung (Fort.)

- wenn die Ausrichtung unterschiedlich ist, muss zunächst der Knoten um den Vater rotiert werden
- in jedem Fall muss um den Großvater rotiert werden




```
private void split() {
    Node dad = node(DAD);
    if (dad != null && dad.m_bIsRed) {
        if (  node(G_DAD).m_Key.compareTo(dad.m_Key) < 0 !=
              dad.m_Key.compareTo(node(NODE).m_Key) < 0)
              rotate(DAD);
        rotate(G_DAD);
    }
}
```

1 oder 2 Rotationen

vordefinierte Baumimplementierungen

- in Java gibt es die Klasse TreeMap<K,D>, die auf Rot-Schwarz-Bäumen basiert

- in C++ gibt es std::map<K,D>, deren Implementierung nicht vorgeschrieben ist

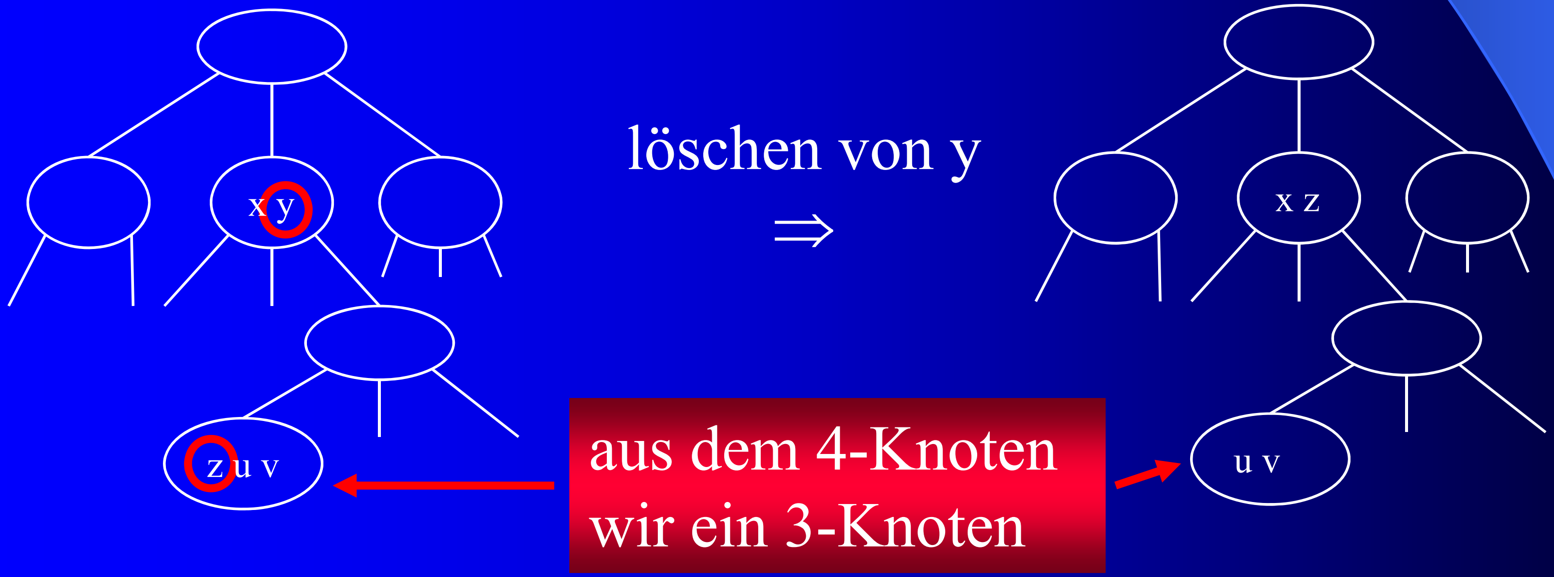
Vorlesung 9

Löschen aus Rot-Schwarz Bäume

- Analog zu dem Einfügen wird beim Löschen durch Rotationen die Baumtiefe ausgeglichen
- Löschen aus Rot-Schwarz Bäumen ist deutlich komplexer als das Einfügen, weil es
  - deutlich mehr Fälle gibt
  - u.U. dreimal rotiert werden muss (statt zweimal wie beim Einfügen)
- erste Überlegung: wie kann in einem Top-Down 2-3-4 Baum gelöscht werden
- folgende Arbeit basiert auf Arbeiten von Prof. Dr. Jonathan Shewchuk (http://www.cs.berkeley.edu/~jrs/61b/)
- Paper: http://www.cs.berkeley.edu/~jrs/61b/lec/27

Löschen aus Top-Down 2-3-4 Bäumen

- Analog zu Löschen aus Binärbaumen
- zu löschendes Element wird durch das nächstgrößere Element ersetzt
- dieses (das nächstgrößere Element) liegt garantiert in einem Blatt



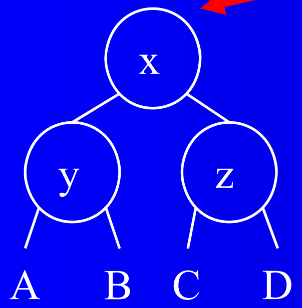
Löschen aus Top-Down 2-3-4 Bäumen (Forts.)

- funktioniert problemlos, wenn das Blatt ein 3-Knoten oder ein 4-Knoten ist
- Problem, wenn Blatt ein 2-Knoten ist
- Lösung: analog zum Einfügen
  - beim <u>Abstieg</u> werden Schlüssel nach <u>unten</u> gezogen (Knoten werden aufgebläht)
  - (beim <u>Einfügen</u> wurden Schlüssel nach <u>oben</u> geschoben)
- es gibt drei Situationen
  - 2-Wurzel mit zwei 2-Söhnen
  - aufzublähender Knoten hat (mindestens) einen 3- oder 4-Knoten Bruder
  - aufzublähender Knoten hat nur 2-Brüder

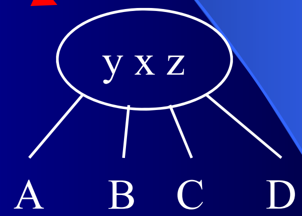
Fall 1

- 2-Wurzel mit zwei 2-Söhnen

Wurzel



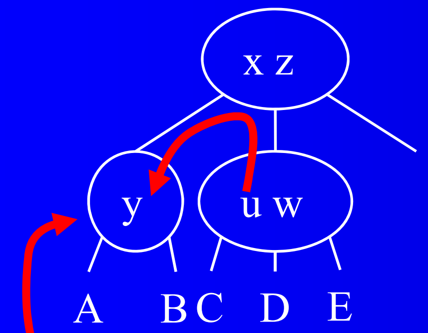
Wurzel und Söhne
werden zu neuer
⇒
Wurzel
zusammengefasst



- die einzige Situation, in der die Tiefe des Baums geringer wird

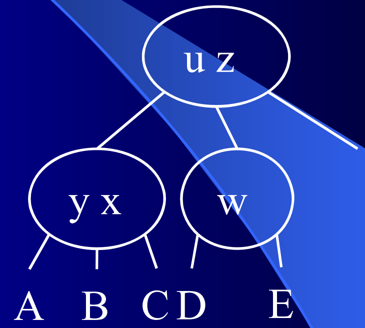
Fall 2

- aufzublähender Knoten hat (mindestens) einen 3- oder 4-Knoten Bruder



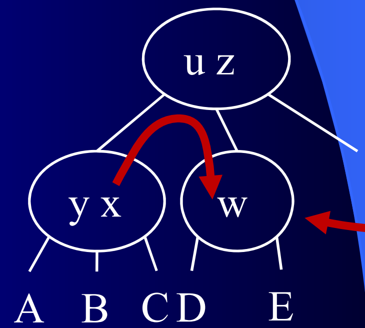
Linksrotation

⇒



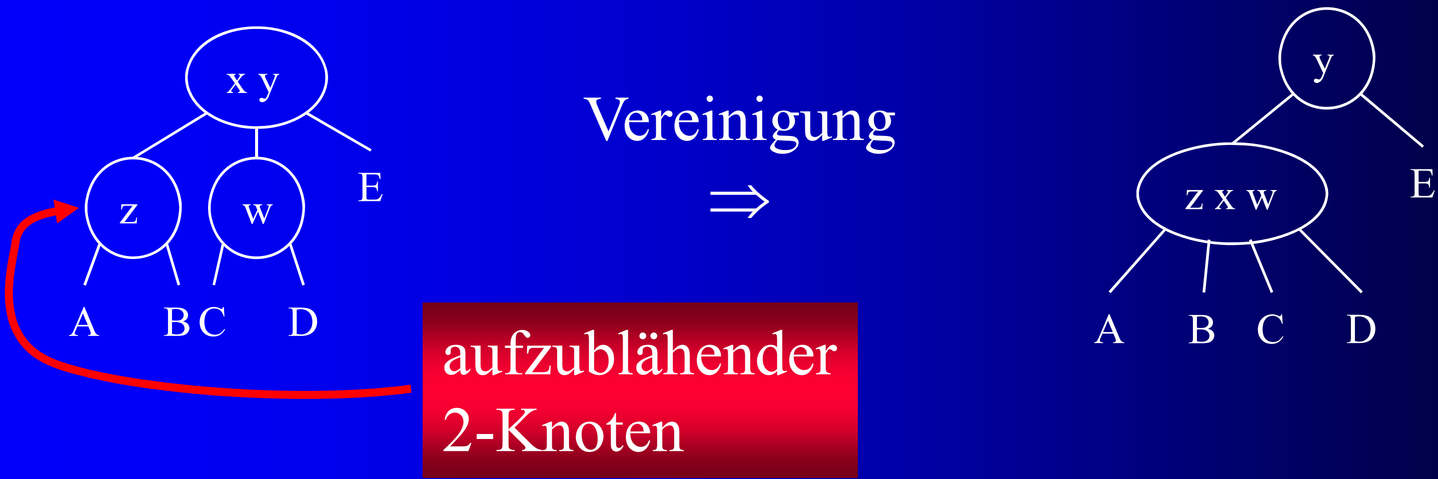
aufzublähender 2-Knoten

- gibt es natürlich auch als Rechtsrotation



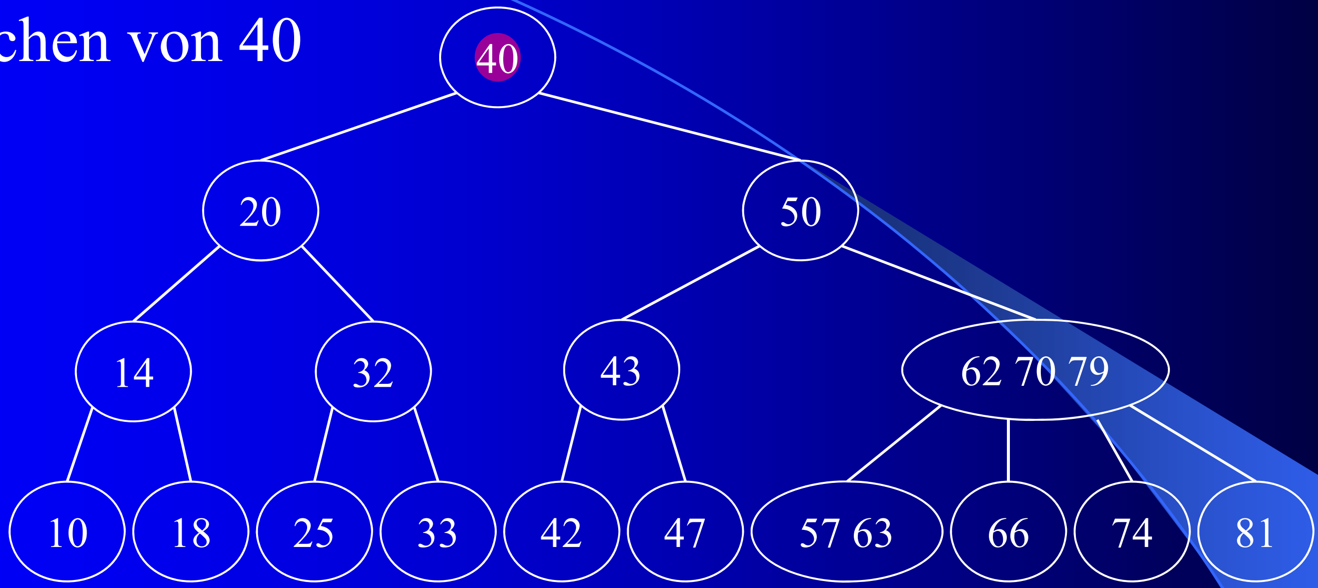
Fall 3

- aufzublähender Knoten hat nur 2-Brüder
- Folge: Vater ist 3- oder 4-Knoten, weil
  - er im vorherigen Schritt schon so groß war, oder
  - er im vorherigen Schritt aufgebläht wurde
  - (ist der Vater 2-Knoten Wurzel und beide Söhne sind 2-Knoten gilt Fall 1)

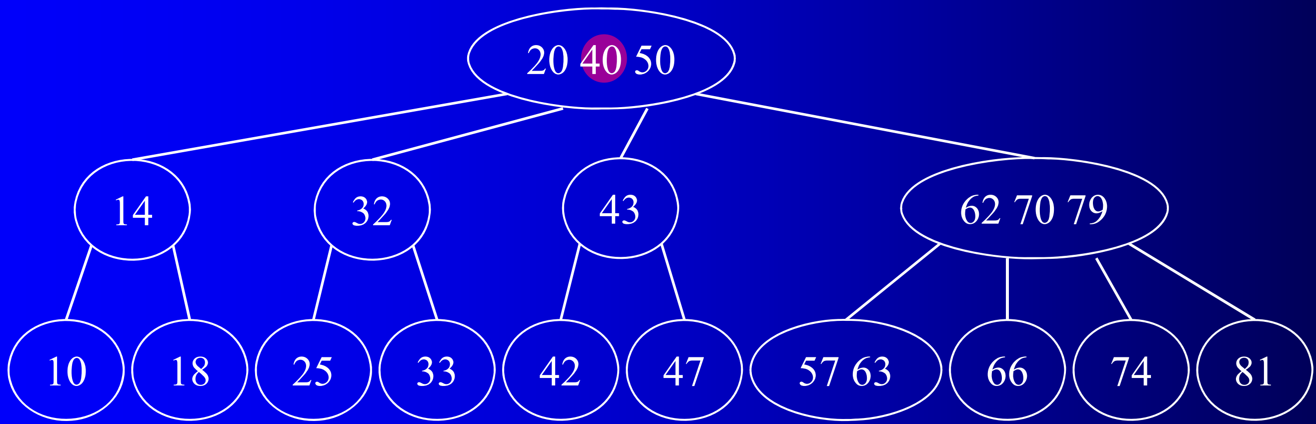


Beispiel

- Löschen von 40

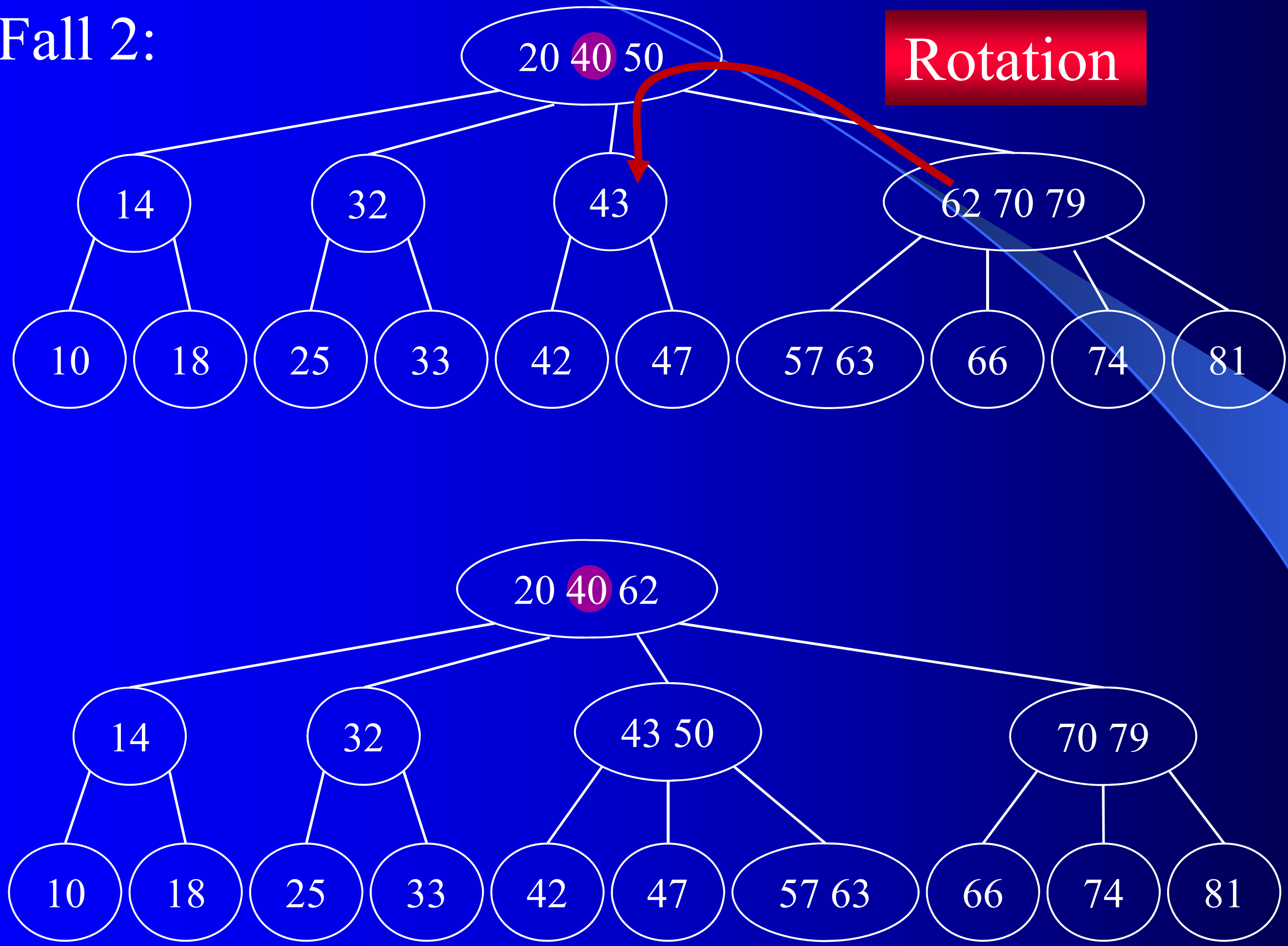


- Fall 1: Wurzel und beide Söhne zusammenfassen



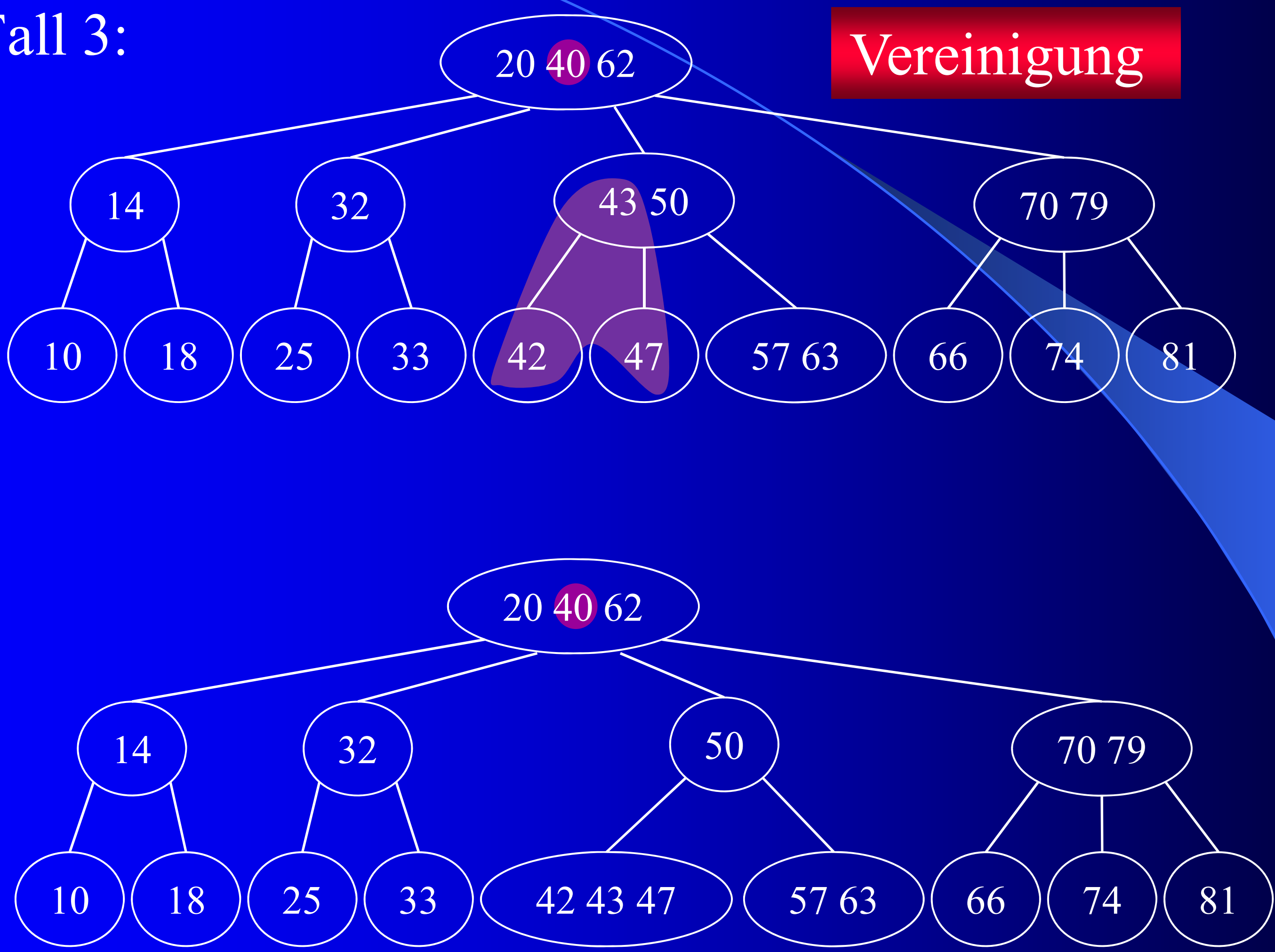
Beispiel (Forts.)

- Fall 2:



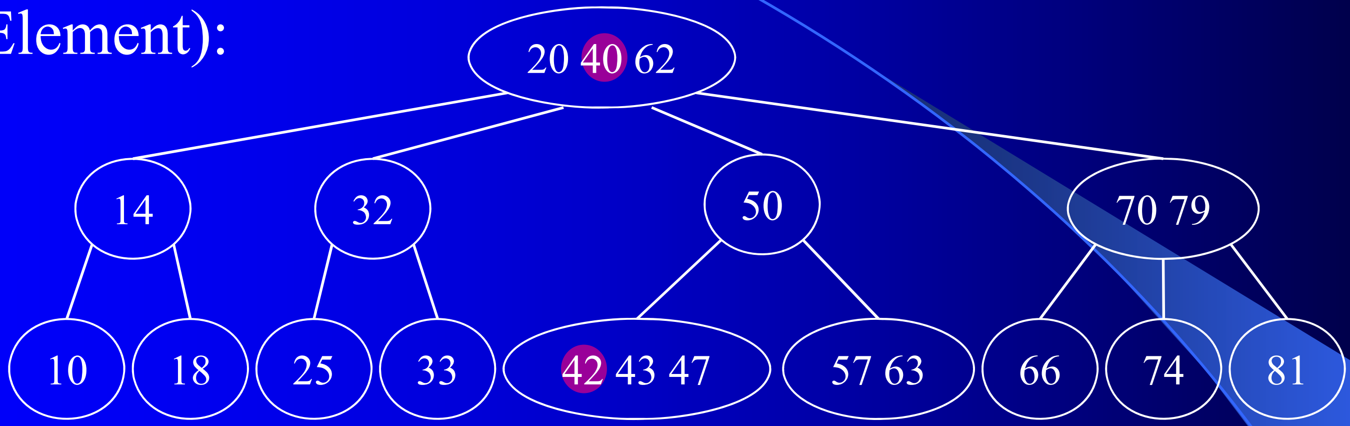
Beispiel (Forts.)

- Fall 3:

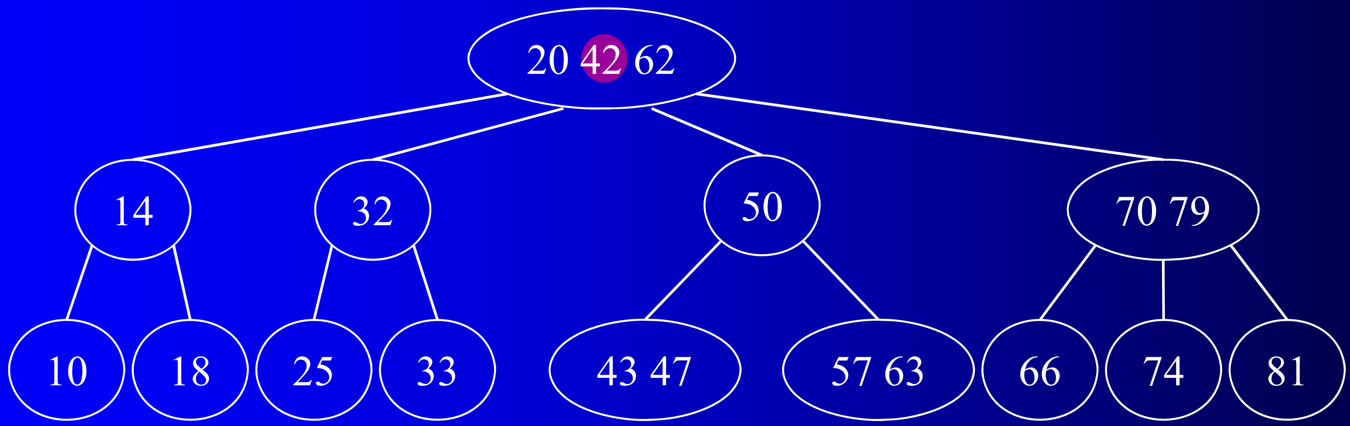


Beispiel (Forts.)

- Löschen von 40 durch Verschiebung der 42 (nächstgrößeres Element):



- Ergebnis:



Fallunterscheidung

• Fall 1: 2-Wurzel und 2-Söhne	
• Fall 2: 2-Wurzel mit 2-Sohn und	3-Bruder (2x)
	4-Bruder (2x)
• Fall 3: 3-Knoten mit 2-Sohn und	2-Bruder (3x)
	3-Bruder (3x)
	4-Bruder (3x)
• Fall 4: 4-Knoten mit 2-Sohn und	2-Bruder (4x)
	3-Bruder (4x)
	4-Bruder (4x)

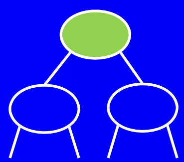
⇒ 26 (!!!) Fälle auf Ebene der Top-Down 2-3-4 Bäume

⇒ 46 (!!!) Fälle auf Ebene der Rot-Schwarz Bäume (sehr viele symmetrische Fälle)

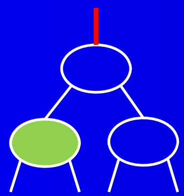
Fallunterscheidung (Forts.)

- anderer Ansatz: welche Fälle gibt es bei einem Rot-Schwarz Baum?

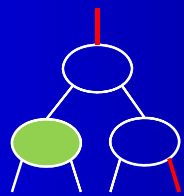
1. Wurzelfall



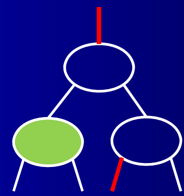
2. 2er unter 3er oder 4er mit 2er Bruder



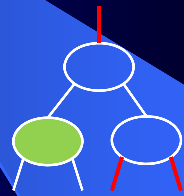
3. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder



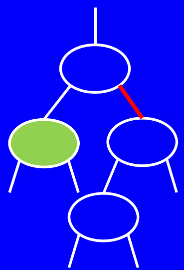
4. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder



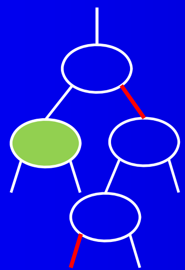
5. 2er unter 3er oder 4er oder Wurzel (!!!) mit 4er Bruder



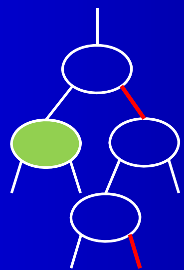
6. 2er unter 3er mit 2er Bruder



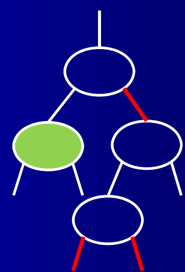
7. 2er unter 3er mit 3er Bruder



8. 2er unter 3er mit 3er Bruder

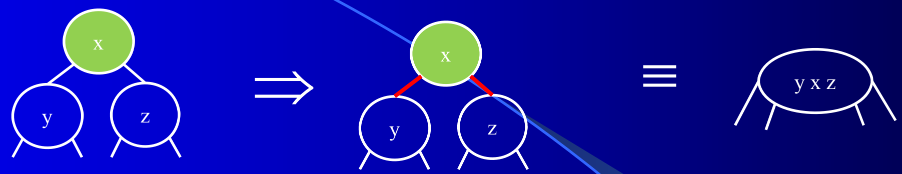


9. 2er unter 3er mit 4er Bruder

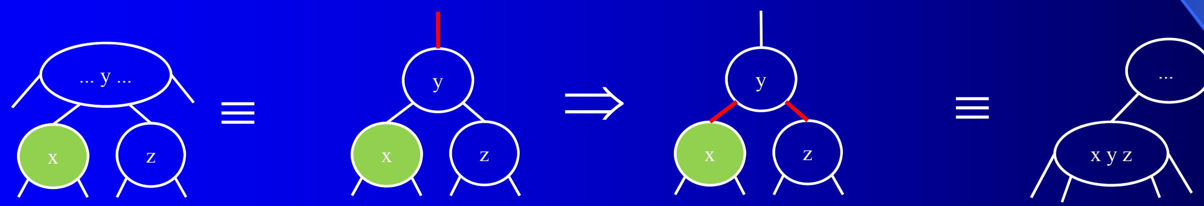


Fallunterscheidung (Forts.)

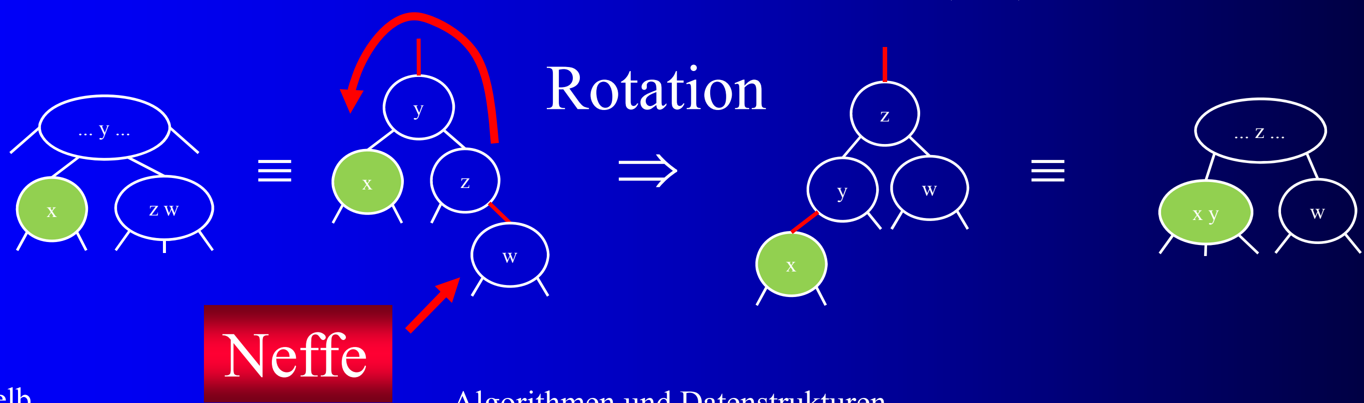
- 1. Wurzelfall



- 2. 2er unter 3er oder 4er mit 2er Bruder

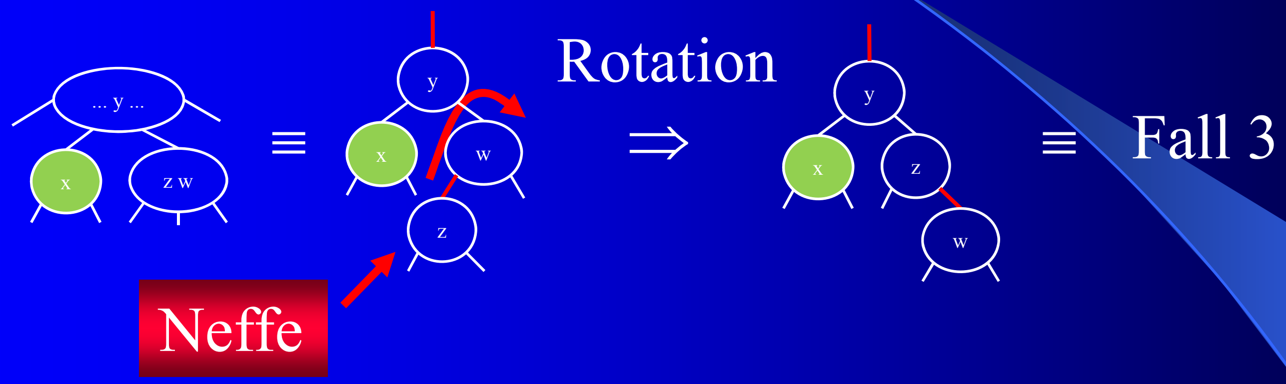


- 3. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder

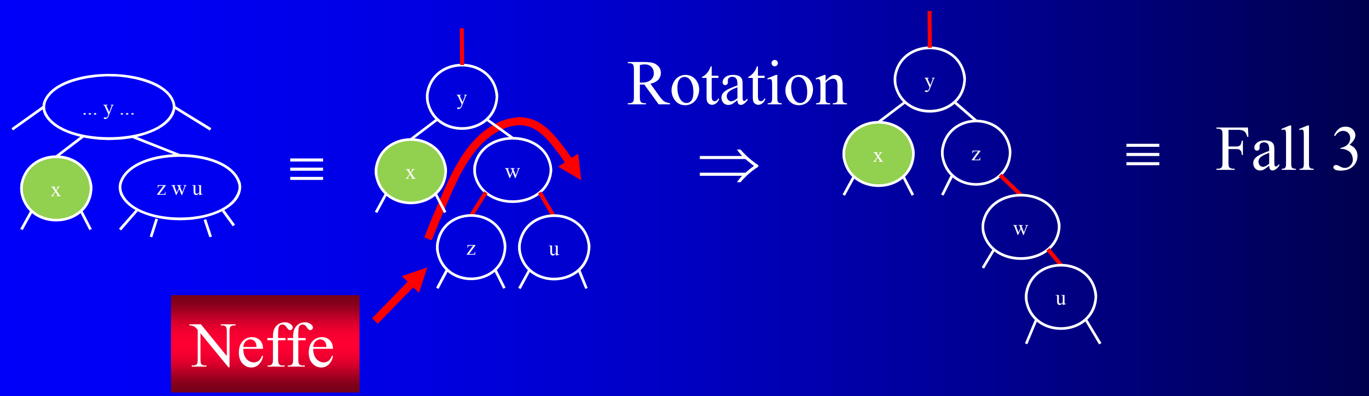


Fallunterscheidung (Forts.)

4. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder

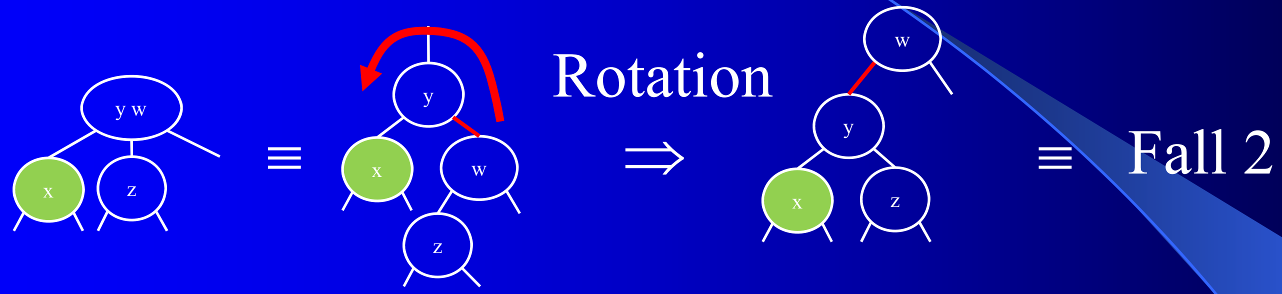


5. 2er unter 3er oder 4er oder Wurzel (!!!) mit 4er Bruder

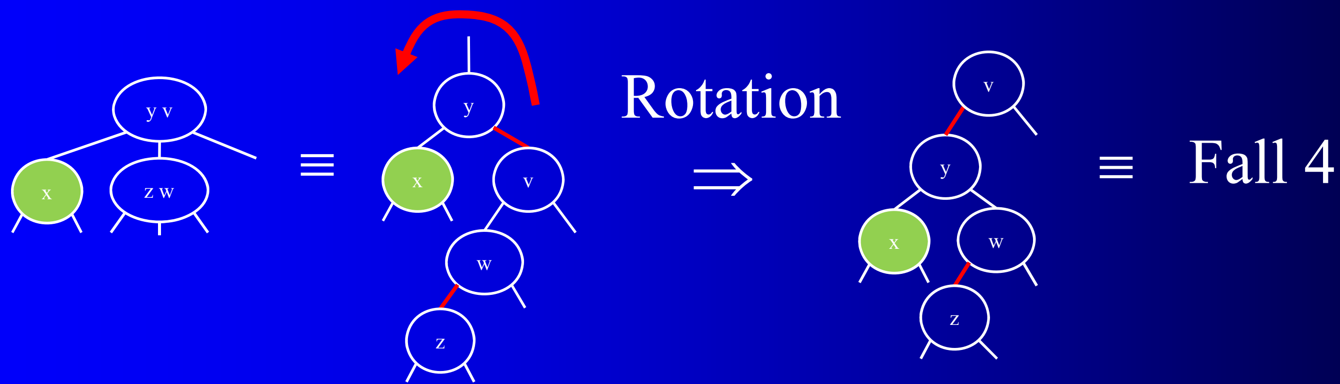


Fallunterscheidung (Forts.)

6. 2er unter 3er mit 2er Bruder

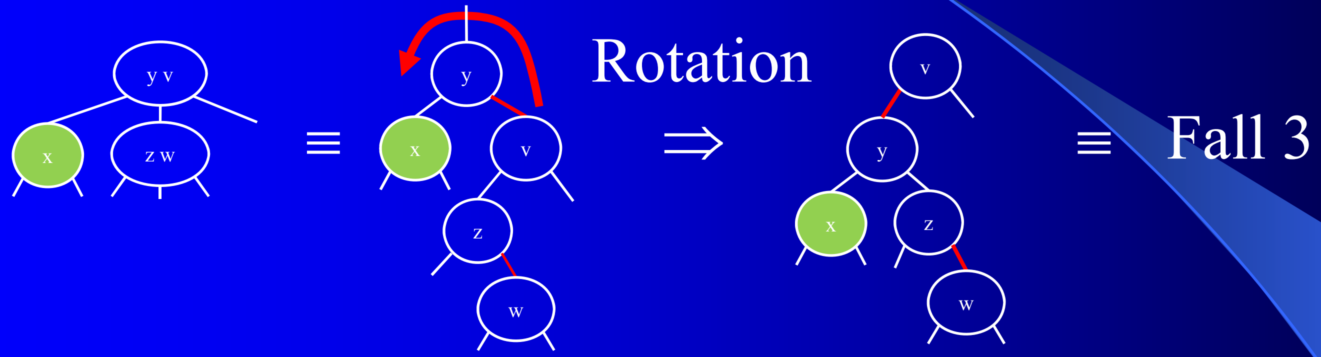


7. 2er unter 3er mit 3er Bruder

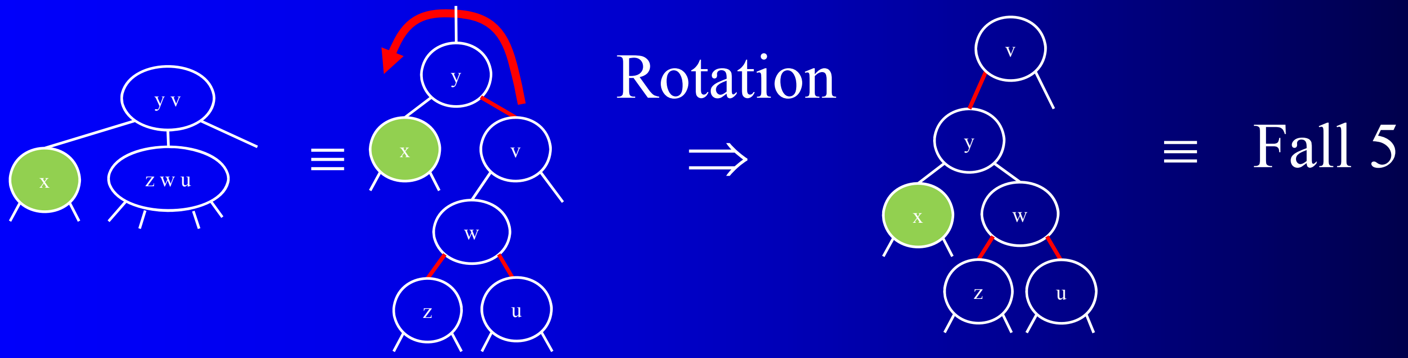


Fallunterscheidung (Forts.)

8. 2er unter 3er mit 3er Bruder



9. 2er unter 3er mit 4er Bruder



Implementierung des Löschens

```
boolean remove(K key) {
    NodeHandler h = new NodeHandler(m_Root);
    while (!h.isNull()) {
        h.join();
        final int RES = key.compareTo(h.node(h.NODE).m_Key);
        if (RES == 0) {
            if (h.node(h.NODE).m_Right == null) {
                h.set(h.node(h.NODE).m_Left,h.NODE,true);
            } else {
                NodeHandler h2 = new NodeHandler(h);
                h2.down(false); // go right
                h2.join();
                while (h2.node(h2.NODE).m_Left != null) {
                    h2.down(true);
                    h2.join();
                }
                h.node(h.NODE).m_Key = h2.node(h2.NODE).m_Key;
                h.node(h.NODE).m_Data = h2.node(h2.NODE).m_Data;
                h2.set(h2.node(h2.NODE).m_Right,h2.NODE,true);
            }
            if (m_Root != null)
                m_Root.m_bIsRed = false;
            return true;
        }
        h.down(RES < 0);
    }
    return false;
}
```

das Löschen ist identisch zu dem Löschen in Binärbäumen ...

... mit Ausnahme des Aufblähens (bzw. Vereinigung) der 2-Knoten

... und des Bewahrens der Kantenfarbe

Kopie des NoteHandlers

Die Wurzel ist nie rot.

Implementierung des Löschens (Forts.)

- die Node Klasse muss 2-Knoten identifizieren können

```
public boolean is2Node() {
    return !m_bIsRed
        && (m_Left == null || !m_Left.m_bIsRed)
        && (m_Right == null || !m_Right.m_bIsRed);
}
```

- beim Einfügen der Knoten muss die Kantenfarbe bewahrt werden

```
void set(Node n,int kind,boolean copyColours) {
    if (node(kind+1) == null)
        m_Root = n;
    else if   node(kind) != null ?
              node(kind+1).m_Left == node(kind) :
              n.m_Key.compareTo(node(kind+1).m_Key) < 0)
        node(kind+1).m_Left = n;
    else
        node(kind+1).m_Right = n;
    if (copyColours && node(kind) != null && n != null)
        n.m_bIsRed = node(kind).m_bIsRed;
    m_Nodes[kind] = n;
}
```

ursprüngliche Kanten-farbe auf den neuen Knoten übertragen

Implementierung des Löschens (Forts.)

- der **NodeHandler** bekommt die **join** Methode …

```
private void join() {
    if (node(NODE).is2Node()) {
        if (  node(DAD) == null &&
              node(NODE).m_Left != null &&
              node(NODE).m_Left.is2Node() &&
              node(NODE).m_Right != null &&
              node(NODE).m_Right.is2Node()) {
                  node(NODE).m_Left.m_bIsRed = true;
                  node(NODE).m_Right.m_bIsRed = true;
        } ...
    }
}
```

nur für 2-Knoten muss etwas getan werden

der Wurzelfall

Kanten werden nur umgefärbt

- … und die Kopiermethode

```
NodeHandler(NodeHandler h) {
    m_Nodes[NODE] = h.m_Nodes[NODE];
    m_Nodes[DAD] = h.m_Nodes[DAD];
    m_Nodes[G_DAD] = h.m_Nodes[G_DAD];
    m_Nodes[GG_DAD] = h.m_Nodes[GG_DAD];
}
```

Implementierung des Löschens (Forts.)

```
private void join() {
     if (node(NODE).is2Node()) {
...
          } else if (node(DAD) != null) {
               NodeHandler nephew = getNephew();
               if (nephew.node(DAD).m_bIsRed) {
                    nephew.rotate(G_DAD);
                    m_Nodes[GG_DAD] = m_Nodes[G_DAD];
                    m_Nodes[G_DAD] = nephew.m_Nodes[G_DAD];
                    nephew = getNephew();
               }
               if (nephew.node(DAD).is2Node()) {
                    node(NODE).m_bIsRed = true;
                    nephew.node(DAD).m_bIsRed = true;
                    node(DAD).m_bIsRed = false;
               } else {
                    if (!nephew.isNull() && nephew.node(NODE).m_bIsRed)
                         nephew.rotate(DAD);
                    nephew.rotate(G_DAD);
               }
          }
     }
}
```

ist es nicht der Wurzelfall und gibt es einen Vorgänger?

NodeHandler des Neffens

Vater des Neffens (=mein Bruder) rot? ⇒ Fall 6 - 9

Groß- und Urgroßvater sind jetzt vertauscht ⇒ richten im NodeHandler

neue Neffenhistory

Fall 2: Bruder ist 2-Knoten ⇒ Kanten umfärben

Fall 4 - 5: rotiere Neffen um Vater (= mein Bruder)

Fall 3: rotiere Bruder um Vater

Implementierung des Löschens (Forts.)

- die NodeHandler Klasse muss die Neffenhistory erzeugen können

```
NodeHandler getNephew() {
    Node node = node(NODE);
    Node dad = node(DAD);
    Node gDad = node(G_DAD);

    Node brother = node == dad.m_Left ? dad.m_Right : dad.m_Left;
    Node nephew = node == dad.m_Left ? brother.m_Left : brother.m_Right;
    NodeHandler res = new NodeHandler(nephew);

    res.m_Nodes[DAD] = brother;
    res.m_Nodes[G_DAD] = dad;
    res.m_Nodes[GG_DAD] = gDad;
    return res;
}
```

bin ich der linke Sohn, ist mein Bruder der rechte Sohn

bin ich der linke Sohn, will ich den linken Neffen

mein Bruder ist der Vater des Neffens

mein Vater ist der Großvater des Neffens

mein Großvater ist der Urgroßvater des Neffens

Implementierung des Löschens (Forts.)

- die **rotate** Methode muss noch angepasst werden

```
void rotate(int kind) {
      Node dad = node(kind);
      Node son = node(kind-1);
      boolean sonColour = son.m_bIsRed;
      if (!sonColour) {
            if (son.m_Left != null)
                  son.m_Left.m_bIsRed = false;
            if (son.m_Right != null)
                  son.m_Right.m_bIsRed = false;
            dad.m_bIsRed = false;
            dad.m_Left.m_bIsRed = true;
            dad.m_Right.m_bIsRed = true;
      } else {
            son.m_bIsRed = dad.m_bIsRed;
            dad.m_bIsRed = sonColour;
      }
... // rotate wie gehabt
      set(son,kind,false);
}
```

wenn der Sohn nicht rot ist (ist bei **insert** immer rot), ist es der Fall 3 der **remove** Methode

Enkel (wenn vorhanden) schwarz färben

Vater ist schwarz, beide Söhne (vor der Rotation) werden rot

beim Einfügen nicht die Farbe kopieren