Vorlesung 1

Programmierung von Algorithmen und Datenstrukturen

Dozent: Prof. Dr. Peter Kelb

Raumnummer: Z4030

e-mail: peter.kelb@gmx.de

Sprechzeiten: nach Vereinbarung

Sourcen: http://elearning.hs-bremerhaven.de/start.php
(im geschlossenen Bereich)

Ziel der Vorlesung:

- schnelle Graphikprogrammierung in Java
- komplexe Sortierverfahren
- komplexe Algorithmen
- Einführung in Graphalgorithmen

Voraussetzung:

Vorlesung: Programmierung II

Organisatorisches (Fort.)

- Vorlesung (5 CPs)
- Übungen (aufeinander aufbauend)

Übungen

- zu jeder Vorlesung gibt es Übungen (http://elearning.hs-bremerhaven.de/start.php), die in den Übungsstunden ***und*** zu Hause bearbeitet werden müssen
- in den Übungsstunden können Probleme mit den Tutoren besprochen werden

Bücher

- Algorithms, Robert Sedgewick, Addison-Wesley Longman, ISBN-13: 978-0321573513

Bücher (Fort.)

sehr umfangreich

- Datenstrukturen
- Sortieralgorithmen
- Suchalgorithmen
- Verarbeitung von Zeichenfolge (Pattern Matching, Parsing, Datenkomprimierung, Kryptologie)
- Geometrische Algorithmen
- Algorithmen mit Graphen
- Mathematische Algorithmen
- ...

Bücher (Fort.)

- sehr umfangreich ...
  - Ausblick auf: parallele Algorithmen, schnelle Fourier-Transformation, dynamische Programmierung, lineare Programmierung, erschöpfendes Durchsuchen, NP-vollständige Probleme
- recht kompliziert, aber:

Anschaffung für das ganze Informatikerleben

Schnelle Animation: Motivation

Bisher für komplexe Animation:

1. Beschaffung eines Hintergrundbildes
2. Malen in dieses Hintergrundbild mittels der draw-Methoden
3. Malen des Hintergrundbildes in den Frame

Schritt 1 und 3 sind schnell, Schritt 2 ist langsam. Die draw-Methoden sind oft umfangreicher als benötigt, und daher zu komplex. Beispiel: Setzen eines Punktes in einer spezifischen Farbe:

```
g.setColor(new Color(213,17,142));
g.drawLine(200,100,200,100);
```

Wunsch: Punkt in der gewünschten Farbe direkt setzen.

Schnelle Animation (Fort.)

Bisher wurde ein Image mittels der createImage erschaffen.

```
class Component extends Object {
    public Image createImage(int width, int height);
}
```

Es gibt aber noch eine weitere Möglichkeit:

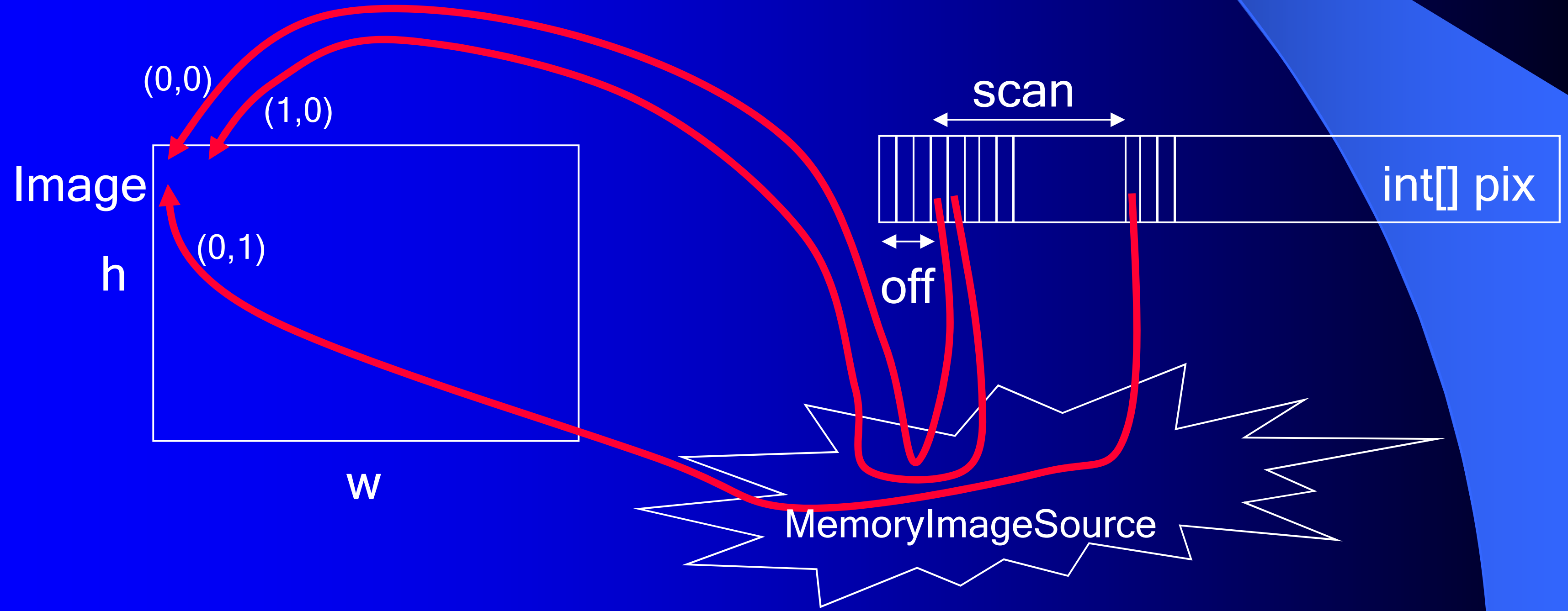
```
class Component extends Object {
    public Image createImage(ImageProducer prod);
}
```

Hier wird ein „Bilderschaffer“ übergeben, der das Bild erzeugt.

ImageProducer: MemoryImageSource

Die Klasse MemoryImageSource erzeugt Bilder auf der Basis von Integerfeldern. Dabei repräsentiert jeder Integerwert ein Pixel.

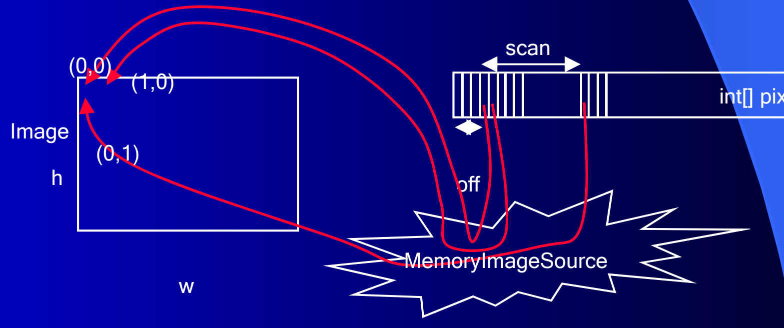
```
class MemoryImageSource extends Object implements ImageProducer {
    public MemoryImageSource(int w, int h, int[] pix, int off, int scan);
    ...
}
```



ImageProducer: MemoryImageSource

Zusammenspiel von Image und MemoryImageSource:

- wird in das Feld pix ein neuer Wert eingetragen, so wird ***nicht*** das ***zugehörige Pixel*** im Image ***verändert***
- das Bild bekommt die Werte beim ersten Mal
- wenn das Bild mittels der flush Methode der Klasse Image geleert wird, holt es sich neue Daten vom ImageProducer, d.h. vom Feld pix.



Beispiel

```
import java.util.*;
import java.awt.*;
import java.awt.image.*;

class RndColor extends Frame {
    final int W = 300;
    final int H = 200;
    Image m_Img;
    int[] m_Pix = new int[W*H];
    MemoryImageSource m_ImgSrc;

    public RndColor() {
        super("Random Color");
        setSize(300,200);
        m_ImgSrc = new MemoryImageSource(W,H,m_Pix,0,W);
        m_Img = createImage(m_ImgSrc);
        setVisible(true);
    }

    public void update(Graphics g) {}
    public void paint(Graphics g) {}

...
```

Ein ImageProducer wird erzeugt

Diese beiden Werte sind i.d.R. gleich

Ein neues Bild

update und paint ausschalten

Go!

Beispiel (Fort.)

Alte Technik

```
...
    public void rnd() {
        Random rnd = new Random();
        while (true) {
            for(int i = 0;i < W*H;++i) {
                m_Pix[i] = rnd.nextInt();
            }
            m_Img.flush();
            getGraphics().drawImage(m_Img,0,0,this);
            try {
                Thread.sleep(20);
            } catch (InterruptedException e) {}
        }
    }

    public static void main(String[] args) throws Exception {
        RndColor win = new RndColor();
        win.rnd();
    }
}
```

Alle Pixel werden zufällig gesetzt

Das Bild wird entleert

... und neu im Frame gezeichnet

Farben und Integer

Frage: Wie werden die Integer Werte als Farben interpretiert?

31			0
alpha	red	green	blue
7 0	7 0	7 0	7 0

alpha: Wert der Transparenz: 0 = durchsichtig, 255 = komplett undurchsichtig

red: Rotanteil der Farbe

green: Grünanteil der Farbe

blue: Blauanteil der Farbe

Beispiel: undurchsichtiges, volles Gelb

255 << 24 | 255 << 16 | 255 << 8

Beispiel: undurchsichtiger, mittlerer Grauton

255 << 24 | 127 << 16 | 127 << 8 | 127

Farben und Integer (Fort.)

Aufgabe: Gegeben sind 2 Farben als Integerwerte i1 und i2.
Erzeuge eine neue Farbe i (als Integerwert), die zu 40% aus i1 und 60% aus i2 besteht.

Idee: Mischen der einzelnen Farbanteile

```
int singleShuffle(int i1_part, int i2_part, int p) {
    return i1_part + (i2_part - i1_part) * p / 100;
}

int colorShuffle(int i1, int i2, int p) {
  int red    = singleShuffle((i1 >> 16) & 255,(i2 >> 16)  & 255,p);
  int green  = singleShuffle((i1 >> 8)  & 255,(i2 >> 8)   & 255,p);
  int blue   = singleShuffle((i1)       & 255,(i2)        & 255,p);
  return (255 << 24) | (red << 16) | (green << 8) | blue;
}
```

Beispiel

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class LabelScrollBar extends Panel {
    TextField m_Lab = new TextField(6);
    Scrollbar m_Bar = new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,256);
    String m_Prefix;

    public LabelScrollBar(String strPrefix) {
        m_Prefix = strPrefix;
        m_Lab.setText(m_Prefix);
        m_Lab.setEnabled(false);
        setLayout(new BorderLayout());
        add(BorderLayout.EAST,m_Lab);
        add(BorderLayout.CENTER,m_Bar);
        m_Bar.addAdjustmentListener(new AdjustmentListener() {
            public void adjustmentValueChanged(AdjustmentEvent e) {
                m_Lab.setText(m_Prefix + m_Bar.getValue());
            }
        });
    }
}
...
```

LabelScrollBar ist ein Panel bestehend aus einem Scrollbar und einem Label mit dem eingestellten Wert

sobald der Scrollbar verändert wird, wird das Label neu eingestellt

Beispiel (Fort.)

```
...
class ControlledColor extends Panel implements AdjustmentListener {
    LabelScrollBar red = new LabelScrollBar("red ");
    LabelScrollBar green = new LabelScrollBar("green ");
    LabelScrollBar blue = new LabelScrollBar("blue ");
    int[] cols;
    Shade shad;
    public ControlledColor(Shade shad, int[] cols) {
        this.shad = shad;   this.cols = cols;
        setLayout(new GridLayout(3,1));
        add(red);    add(green);    add(blue);
        red.m_Bar.addAdjustmentListener(this);
        green.m_Bar.addAdjustmentListener(this);
        blue.m_Bar.addAdjustmentListener(this);
    }
    public void adjustmentValueChanged(AdjustmentEvent e) {
        cols[0] = red.m_Bar.getValue();
        cols[1] = green.m_Bar.getValue();
        cols[2] = blue.m_Bar.getValue();
        shad.reRun();
    }
}
...
```

ControlledColor baut 3 Scroll-bars zusammen und merkt sich die eingestellten Werte in cols und ruft die Berechnungsroutine von shad auf

Beispiel (Fort.)

```
...
class Shade extends Frame {
    final int W = 500;    final int H = 300;     Image m_Img;
    int[] m_Pix = new int[W*H];   MemoryImageSource m_ImgSrc;
    int[] col1 = new int[3]; int[] col2 = new int[3];
    public Shade() {
        super("Shade ...");
        m_ImgSrc = new MemoryImageSource(W,H,m_Pix,0,W);
        m_Img = createImage(m_ImgSrc);
        setSize(W,H); setVisible(true);
    }
    private int compColor(int x1,int x2,int p) {     return x1+(x2-x1)*p/100; }
    public void reRun() {
        for(int i = 0;i < W;++i) {
            final int P = 100*i/W;
            final int COL = 0xff000000
                        | compColor(col1[0],col2[0],P) << 16
                        | compColor(col1[1],col2[1],P) << 8
                        | compColor(col1[2],col2[2],P);
            for(int j = 0;j < H;++j) {m_Pix[i+W*j] = COL;}
        }
        m_Img.flush();
        if (getGraphics() != null) getGraphics().drawImage(m_Img,0,0,
                                                   getWidth(),getHeight(),null);
    }
}
```

Das Hauptfenster für den Farbverlauf

Die Ziel- und Ausgangsfarbe

Berechnet alle Farben neu und malt am Ende das Bild

Go!

Beispiel (Fort.)

```
...
class ColorFade extends Frame {
    public ColorFade() {
        super("Fade it ...");
        Shade shad = new Shade();
        ControlledColor srcCol = new ControlledColor(shad,shad.col1);
        ControlledColor trgCol = new ControlledColor(shad,shad.col2);
        setLayout(new GridLayout(2,1));
        add(srcCol);
        add(trgCol);
        pack();
        setVisible(true);
    }

    public static void main(String [] args) {
        new ColorFade();
    }
}
```

Erzeugt das Fenster für den Farbübergang

Legt für sich selber 2 Control-Panels für die Start- und Zielfarbe an

Vorlesung 2

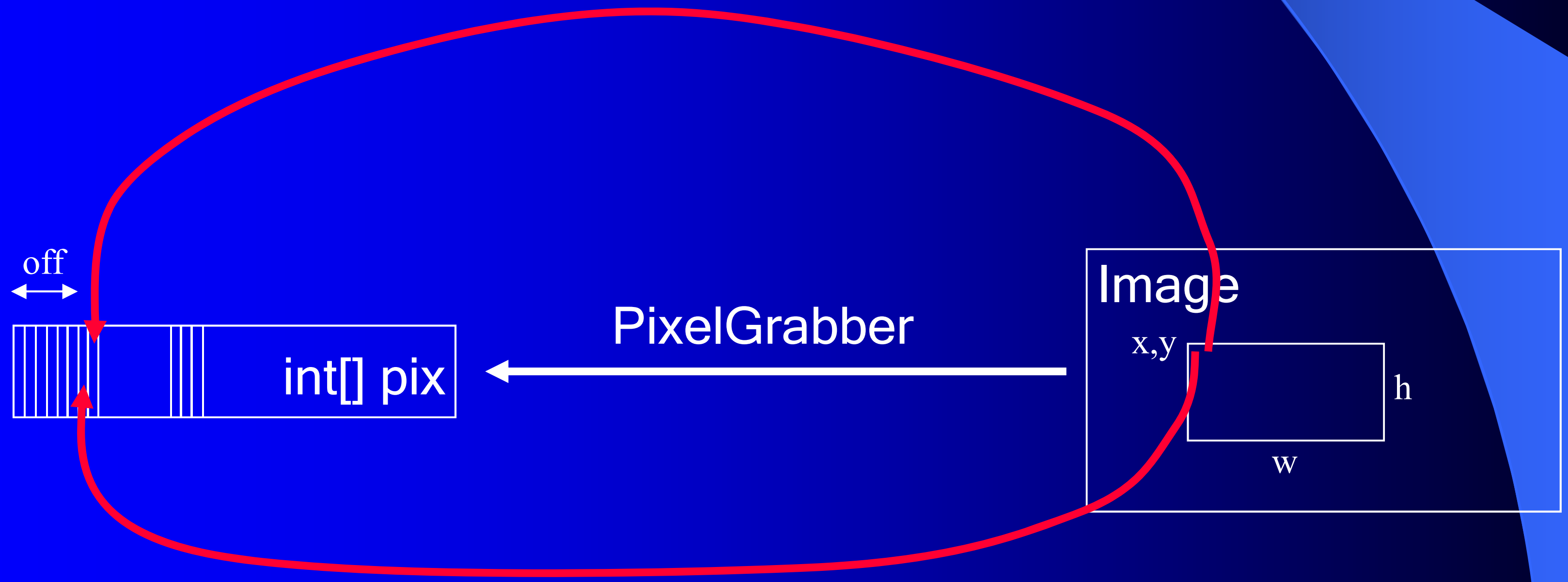
Bilder auslesen

Erzeugen von Bildern aus einem Integerfeld:

Auslesen von Bildern in ein Integerfeld:

Bilder auslesen (Fort.)

```
class PixelGrabber extends Object {
    public PixelGrabber(Image img,
                        int x, int y, int w, int h,
                        int[] pix, int off, int scansize);
}
```



Bilder auslesen (Fort.)

Der PixelGrabber startet das Auslesen aber noch nicht im Konstruktor. Dies muss explizit durch die Methode grabPixel() gestartet werden.

```
class PixelGrabber extends Object {
    ...
    boolean grabPixels() throws InterruptedException;
}
```

Die Methode liefert true zurück, wenn das Auslesen der Pixel erfolgreich war, ansonsten false.

Beispiel

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
class Shuffle extends Component {
    final int W = 500; final int H = 300;
    Image m_Img1,m_Img2,m_Img;
    int[] m_Img1Pix = new int[W*H];     int[] m_Img2Pix = new int[W*H];
    int[] m_Pix = new int[W*H];   MemoryImageSource m_ImgSrc;
    public Shuffle(Frame father) {
        try {
            FileDialog diag = new FileDialog(father); diag.setVisible(true);
            m_Img1 = getToolkit().getImage(diag.getDirectory()+diag.getFile()).
                                        getScaledInstance(W,H, Image.SCALE_SMOOTH);
            diag.setFile(""); diag.setVisible(true);
            m_Img2 = getToolkit().getImage(diag.getDirectory()+diag.getFile()).
                                        getScaledInstance(W,H, Image.SCALE_SMOOTH);
            MediaTracker mt = new MediaTracker(this);
            mt.addImage(m_Img1,0);mt.addImage(m_Img2,0);mt.waitForAll();
            PixelGrabber grab1 = new PixelGrabber(m_Img1,0,0,W,H,m_Img1Pix,0,W);
            PixelGrabber grab2 = new PixelGrabber(m_Img2,0,0,W,H,m_Img2Pix,0,W);
            grab1.grabPixels();grab2.grabPixels();
            m_ImgSrc = new MemoryImageSource(W,H,m_Pix,0,W);
            m_Img = createImage(m_ImgSrc);
        } catch (InterruptedException e) {}
    }
...
```

Objektvariablen

Lädt 2 Bilder ein und skaliert sie auf H×W

überträgt die Pixel in die Felder m_Img1Pix und m_Img2Pix

Beispiel (Fort.)

```
...
   public void paint(Graphics g) {g.drawImage(m_Img,0,0,this);}

   public Dimension getPreferredSize() {return getMinimumSize();}
   public Dimension getMinimumSize() {return new Dimension(W,H);}

   private int compColor(int x1,int x2,int p) {return x1+(x2-x1)*p/100; }
   private int compPix(int pix1,int pix2,int p) {
      final int RED = compColor((pix1 >> 16) & 0xff,(pix2 >> 16) & 0xff,p);
      final int GREEN = compColor((pix1 >> 8) & 0xff,(pix2 >> 8) & 0xff,p);
      final int BLUE = compColor(pix1 & 0xff,pix2 & 0xff,p);
      return 0xff000000 | (RED << 16) | (GREEN << 8) | BLUE;
   }

   public void shuffle(int p) {
      for(int i = 0;i < W*H;++i) {
         m_Pix[i] = compPix(m_Img1Pix[i],m_Img2Pix[i],p);
      }
      m_Img.flush();
      repaint();
   }
}
...
```

Die normalen Zeichen-routinen ändern

Mischt die beiden Farben pix1 und pix2 gemäß des Prozent-satzes p

Mischt die beiden Bilder gemäß des Prozent-satzes p und malt das neue, gemischte Bild

Go!

Beispiel (Fort.)


```
...
class Pic extends Frame {
    public Pic() {
        super("Hey, pictures ...");
        setLayout(new BorderLayout());
        final Shuffle SHUF = new Shuffle(this);
        final Scrollbar BAR = new Scrollbar(Scrollbar.HORIZONTAL,100,1,0,101);
        final Label LAB = new Label("100 %"); Panel pan = new Panel();
        pan.setLayout(new BorderLayout());
        add(BorderLayout .CENTER,SHUF);  add(BorderLayout.SOUTH,pan);
        pan.add(BorderLayout.CENTER,BAR);   pan.add(BorderLayout.EAST,LAB);
        BAR.addAdjustmentListener(new AdjustmentListener() {
                public void adjustmentValueChanged(AdjustmentEvent e) {
                    SHUF.shuffle(BAR.getValue()); LAB.setText(BAR.getValue() + "%");
                }
            });
        pack();
        setVisible(true);SHUF.shuffle(100);
    }
    public void update(Graphics g) {paint(g);}
    public static void main(String[] args) throws Exception {
        new Pic();
    }
}
```

Auch im Hauptfenster die update-Routinen ändern

Go!

Beispiel

```
import java.awt.*;
class RunShuffle extends Frame implements Runnable {
    Shuffle s = new Shuffle(this);
    public RunShuffle() {
        super("Cool, shuffling ...");
        add(s);
        pack();
        setVisible(true);
        Thread t = new Thread(this);
        t.start();
    }
    public void run() {
        while (true) {
            for(int i = 0;i <= 100;i+=2) {   s.shuffle(i);   }
            for(int i = 100;i >= 0;i-=2) {   s.shuffle(i);   }
        }
    }
    public void update(Graphics g) {
        paint();
    }
    public static void main(String[] args) { new RunShuffle(); }
}
```

Mischt die beiden Bilder von einem zum anderen und zurück

Weitere Beispiele

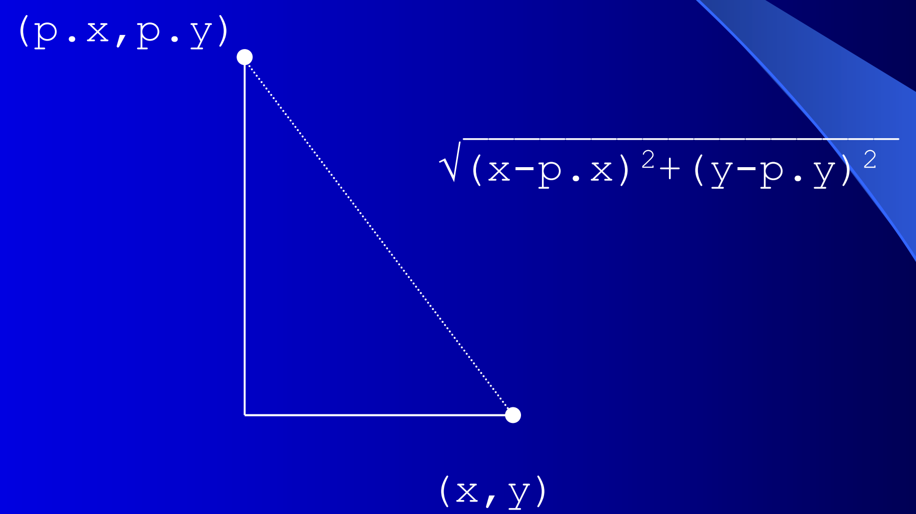
- das vorherige Beispiel hat gezeigt, wie zwei Bilder gemischt werden können
- diese Mischung dient zum Überblenden eines Bildes in ein anderes
- die Mischung ist dabei für das ***gesamte*** Bild erfolgt
- dies ist nicht unbedingt notwendig, die Überblendung kann auch nur für einen Teil erfolgen und auch für verschiedenen Bildbereiche unterschiedlich stark sein

Weitere Beispiele (Fort.)

- Ziel ist es, 2 Bilder einzuladen
- ein Bild wird angezeigt und in einem Umkreis um den Mauszeiger wird das andere Bild angezeigt
- dabei nimmt die Intensität des anderen Bildes mit dem Abstand zum Mauszeiger ab

Weitere Beispiele (Fort.)

- hierzu muss ausgehend von einem Punkt (der Mauszeiger) zunächst berechnet werden, wie weit ein anderer Punkt im Bild entfernt ist



- da der absolute Abstand nicht von Interesse ist, kann die Wurzel auch weggelassen werden

Beispiel

```
class Lens extends Shuffle {
    public Lens(Frame father) {
        super(father);
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseMoved(MouseEvent e) {
                lens(e.getPoint());
            }
        });
    }
    public void lens(Point p) {
        for(int x = 0;x < W;++x) {
            for(int y = 0;y < H;++y) {
                final int IDX = y * W + x;
                final int X_DIFF = p.x - x;
                final int Y_DIFF = p.y - y;
                final int VAL = (X_DIFF * X_DIFF + Y_DIFF * Y_DIFF) / 100;
                final int MAX_VAL = VAL > 100 ? 100 : VAL;
                m_Pix[IDX] = compPix(m_Img1Pix[IDX],m_Img2Pix[IDX],MAX_VAL);
            }
        }
        m_Img.flush();
        repaint();
    }
}
```

Lens erbt von Shuffle

bei Mausbewegung wird die Mauskoordinate an die eigene `lens` Methode übergeben

Abstandsberechnung: ist er zu groß (>100) wird er auf 100 (Prozent) begrenzt

Go!

Beispiel (Fort.)

```
class MainFrame extends Frame {
    MainFrame() {
        add(new Lens(this));
        pack();
        setVisible(true);
    }

    public void update(Graphics g) {
        paint(g);
    }

    public static void main(String[] args) throws Exception {
        new MainFrame();
    }
}
```

Einbindung der eigenen Komponente in einem Fenster, bei dem ebenfalls das Standardverhalten verändert ist

Kantendetektion

- wird ein Bildpunkt mit seiner unmittelbaren Nachbarschaft verglichen, so kann über die Unterschiede ermittelt werden, wo Kanten im Bild lang laufen
- hierzu wird im Wesentlichen die 1. Ableitung gebildet

(x-1, y-1)	(x, y-1)	(x+1, y-1)
(x-1, y)	(x,y)	(x+1, y)
(x-1, y+1)	(x, y+1)	(x+1, y+1)

- es werden die Differenzen zwischen dem Mittelpunkt und seinen 8 Nachbarn berechnet
- diese Differenzen werden zu einem Mittelwert zusammengefaßt

Beispiel

```
class Edge extends JComponent {
    final int W = 500;   final int H = 300;
    Image m_TrgImg,m_SrcImg;
    public Edge(Frame father) {
        try {
            FileDialog diag = new FileDialog(father);
            diag.setVisible(true);
            m_SrcImg = getToolkit().getImage(diag.getDirectory() + diag.getFile()).
                            getScaledInstance(W,H,Image.SCALE_SMOOTH);
            MediaTracker mt = new MediaTracker(this);
            mt.addImage(m_SrcImg,0);
            mt.waitForAll();
            int[] srcPix = new int[W*H];
            int[] trgPix = new int[W*H];
            PixelGrabber grab = new PixelGrabber(m_SrcImg,0,0,W,H,srcPix,0,W);
            grab.grabPixels();
            MemoryImageSource imgProd = new MemoryImageSource(W,H,trgPix,0,W);
            m_TrgImg = createImage(imgProd);
            detectEdges(srcPix,trgPix);
            m_TrgImg.flush();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
...
```

Die Komponente, die später in das Fenster eingebunden wird

Einladen und Skalieren des Bildes

Berechnung der 1. Ableitung

Beispiel (Fort.)

```
...
    public void paintComponent(Graphics g) {
        g.drawImage(m_SrcImg,0,0,this);
        g.drawImage(m_TrgImg,0,H,this);
    }

    public Dimension getPreferredSize() {    return getMinimumSize();    }

    public Dimension getMinimumSize() {     return new Dimension(W,H*2); }

    private void detectEdges(int[] srcPix,int[] trgPix) {
        for(int x = 0;x < W;++x) {
            for(int y = 0;y < H;++y) {
                trgPix[y * W + x] = compColor(srcPix,x,y);
            }
        }
    }

    private int getRed(int col) {       return (col >> 16) & 255; }
    private int getGreen(int col) {     return (col >> 8) & 255;  }
    private int getBlue(int col) {      return col & 255;         }
...
```

zeichnet das Original und das berechnete Bild

für jeden Punkt wird das Verhältnis zur Umgebung berechnet

Beispiel (Fort.)

```
...
    private int compColor(int[] srcPix,int x,int y)  {
        int red = 0;
        int green = 0;
        int blue = 0;
        int cnt = 0;
        final int IDX = y * W + x;
        final int RED = getRed(srcPix[IDX]);
        final int GREEN = getGreen(srcPix[IDX]);
        final int BLUE = getBlue(srcPix[IDX]);
        for(int dx = -1;dx <= 1;++dx) {
            for(int dy = -1;dy <= 1;++dy) {
                if (dx != 0 || dy != 0) {
                    final int X = x+dx;   final int Y = y+dy;final int LOCAL_IDX = Y * W + X;
                    if (0 <= X && X < W && 0 <= Y && Y < H) {
                        ++cnt;
                        red += Math.abs(RED - getRed(srcPix[LOCAL_IDX]));
                        green += Math.abs(GREEN - getGreen(srcPix[LOCAL_IDX]));
                        blue += Math.abs(BLUE - getBlue(srcPix[LOCAL_IDX]));
                    }
                }
            }
        }
        return 0xff000000 | (255 - (red / cnt) << 16) | (255 - (green / cnt) << 8) | (255 - (blue / cnt));
    }
```

die Farbanteile des Mittelpunkts

Der "Farbabstand" nach den 3 Basisfarben unterteilt

Go!

Beispiel (Fort.)

```
...
    public static void main(String[] args) throws Exception {
        JFrame f = new JFrame();
        f.getContentPane().add(new Edge(f));
        f.pack();
        f.setVisible(true);
    }
}
```

Die Einbindung: diesmal in einem JFrame

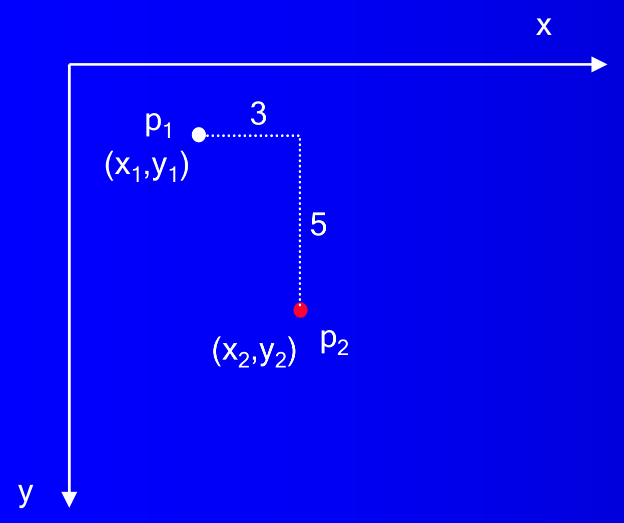
Vorlesung 3

Zweidimensionale Geometrische Transformationen

- Transformationen können dazu verwendet werden, um
  - ein Bild zu konstruieren
  - ein Bild zu verändern
- dabei wird jeder Bildpunkt als ein Vektor im zwei-dimensionalen Koordinatensystem betrachtet
- mit den Transformationen sollen Bildpunkte:
  - verschoben, rotiert, skaliert und verzerrt werden

Verschiebe-Transformation

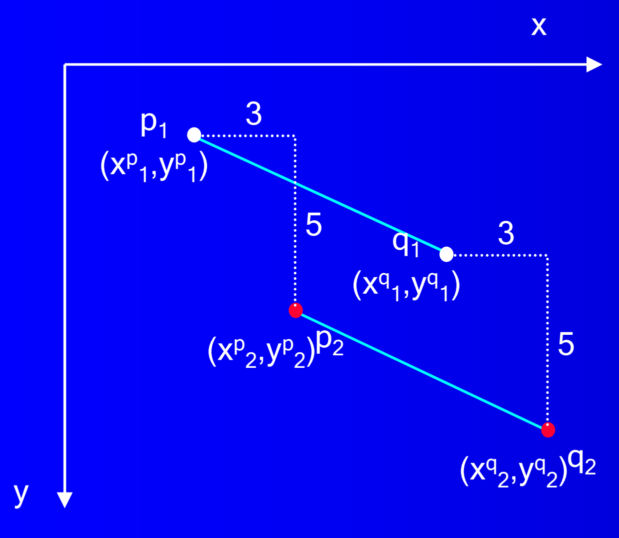
- die Verschiebe-Transformation wird auch ***Translation*** genannt
- sie ist die einfachste Transformation
- hierbei werden zu den Koordinaten eines Punktes lediglich feste Konstanten addiert



- Translation des Punktes p_1 mit den Koordinaten (x_1,y_1) um die Werte 3 und 5
- Ergebnis ist der Punkt p_2 mit den Koordinaten (x_2,y_2)
- Es gilt (in diesem Beispiel):
  - $x_2 = x_1 + 3$
  - $y_2 = y_1 + 5$

Verschiebe-Transformation (Fort.)

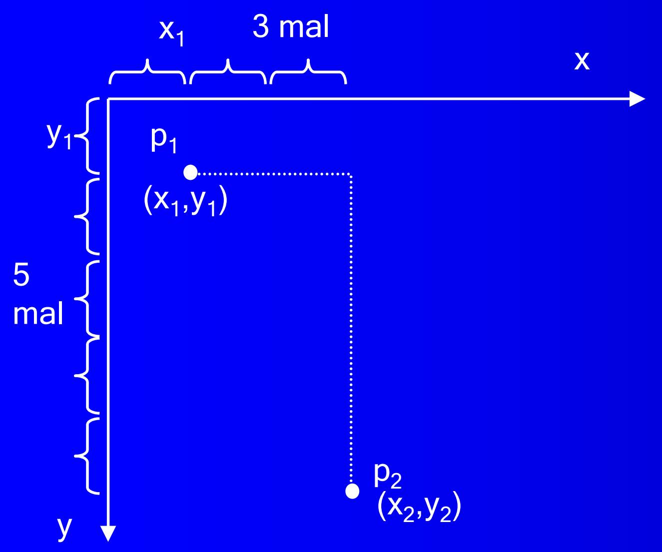
- durch die Addition verändern sich alle Punkte konstant zum Ursprung, d.h. alle entfernen sich um den gleichen Wert vom Ursprung
- das Bild selber verändert sich nicht



- Die Punkte p_1 und q_1 werden bei dieser Translation auf die Punkte p_2 und q_2 abgebildet
- Die zugehörigen Linien haben ihre ***Form nicht geändert***
- Sie haben lediglich ihre ***Lage im Raum*** (2-Dim.) ***geändert***

Skalierung

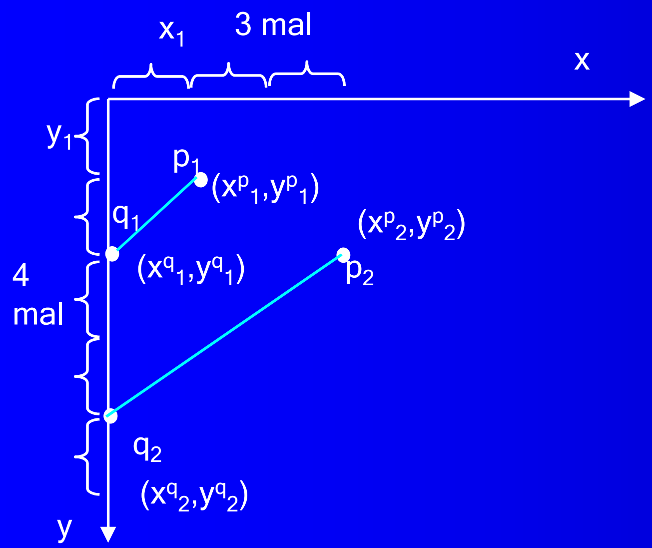
- bei der Skalierung werden die Koordinaten mit einem Wert multipliziert
- dieser Wert kann für den x-Anteil anders als für den y-Anteil sein



- Skalierung des Punktes p_1 mit den Koordinaten (x_1,y_1) um die Werte 3 und 5
- Ergebnis ist der Punkt p_2 mit den Koordinaten (x_2,y_2)
- Es gilt (in diesem Beispiel):
  - $x_2 = x_1 * 3$
  - $y_2 = y_1 * 5$

Skalierung (Fort.)

- anders als bei der Translation verändern sich die Punkte unterschiedlich in Abhängigkeit von ihrem Abstand zum Ursprung
- somit kann eine Vergrößerung bzw. Verkleinerung des Bildes erzielt werden



- Skalierung der Punktes p_1 und q_1 um die Werte 3 und 2
- das Ergebnis ist wieder eine Linie, die aber länger ist als die ursprüngliche Linie und eine andere Steigung hat
- durch Skalierungswerte < 1 kann eine Verkleinerung durchgeführt werden

Scherung

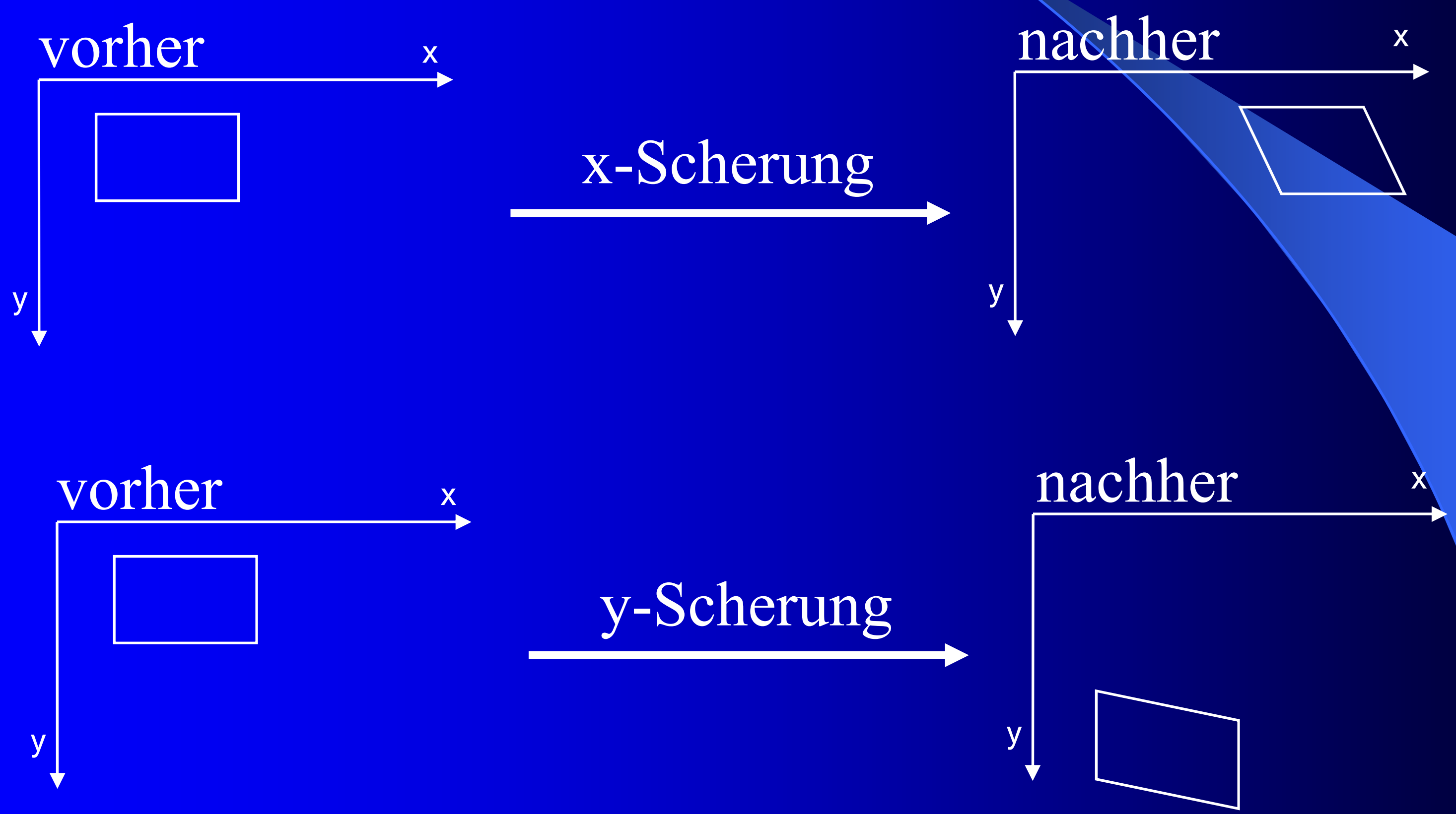
- bei der Scherung handelt es sich um eine Verzehrung einer Achse
- man unterscheidet zwischen einer x- und einer y-Scherung
- bei der y-Scherung wird der y-Wert der Koordinate verändert, während der x-Wert konstant bleibt
- bei der x-Scherung ist es umgekehrt, der y-Wert bleibt konstant, während der x-Wert sich ändert
- bei der Scherung wird zu dem jeweiligen Ursprungswert ein konstantes Vielfache des anderen Koordinatenanteil aufaddiert
- ...

Scherung (Fort.)

- ...
- x-Scherung: der Punkt p_1 mit (x_1,y_1) wird auf den Punkt p_2 abgebildet mit (x_2,y_2):
  - $x_2 = x_1 + y_1 * Sh$
  - $y_2 = y_1$
- y-Scherung: der Punkt p_1 mit (x_1,y_1) wird auf den Punkt p_2 abgebildet mit (x_2,y_2):
  - $x_2 = x_1$
  - $y_2 = y_1 + x_1 * Sh$
- Hierbei gibt Sh den Scherungsfaktor an

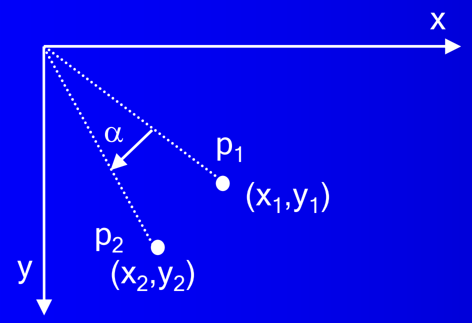
Scherung (Fort.)

- eine Scherung bewirkt eine Verzerrung in x- bzw. y-Richtung
- Beispiel:



Rotation

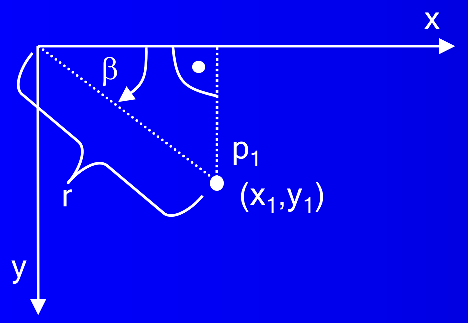
- bei der Rotation soll ein Punkt um den Ursprung um einen gegebenen Winkel rotiert werden
- Beispiel:



- der Punkt p_1 mit den Koordinaten (x_1,y_1) wird um den Winkel α um den Koordinatenursprung auf den neuen Punkt p_2 mit den Koordinaten (x_2,y_2) abgebildet

Rotation (Fort.)

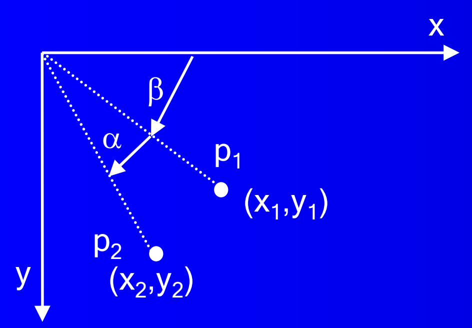
- für die Berechnung der Rotation muss man sich die Koordinaten des Punkts als eine Gleichung aus dem
  - Abstand zum Koordinatenursprung und
  - dem Winkel zwischen der Verbindung des Punktes und dem Koordinatenursprung und der y-Koordinate vorstellen



- es gilt:
  - $\sin(\beta) = y_1 / r$
  - $\cos(\beta) = x_1 / r$
- $\Rightarrow$
  - $x_1 = r \times \cos(\beta)$
  - $y_1 = r \times \sin(\beta)$

Rotation (Fort.)

- soll nun der Punkt mit den Koordinaten (x_1,y_1) um den Winkel α gedreht werden, so kann die Zielkoordinate (x_2,y_2) wiederum durch den Radius und den Trigonometrischen Funktionen dargestellt werden



- es gilt:
  - $x_2 = r \times \cos(\beta+\alpha)$
  - $y_2 = r \times \sin(\beta+\alpha)$
- hieraus folgt unmittelbar:
  - $x_2 = r\times\cos(\beta)\times\cos(\alpha)-r\times\sin(\beta)\times\sin(\alpha)$
  - $y_2 = r\times\sin(\beta)\times\cos(\alpha)+r\times\cos(\beta)\times\sin(\alpha)$

Rotation (Fort.)

mit der Definition von (x_1,y_1)

$x_1 = r \times \cos(\beta)$

$y_1 = r \times \sin(\beta)$

kann

$$x_2 = \overbrace{r\times\cos(\beta)}^{x_1}\times\cos(\alpha)-\overbrace{r\times\sin(\beta)}^{y_1}\times\sin(\alpha)$$

$$y_2 = \underbrace{r\times\sin(\beta)}_{y_1}\times\cos(\alpha)+\underbrace{r\times\cos(\beta)}_{x_1}\times\sin(\alpha)$$

wie folgt vereinfacht werden:

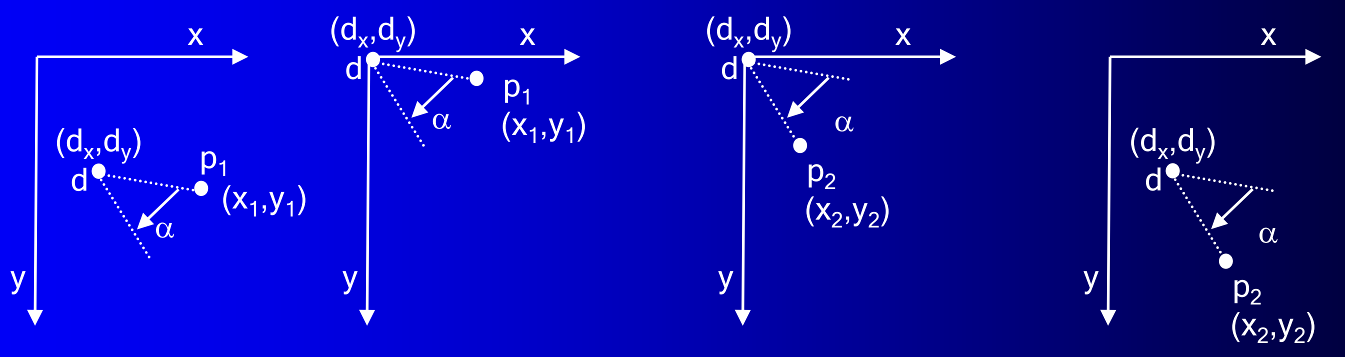
$$x_2 = x_1\times\cos(\alpha)-y_1\times\sin(\alpha)$$
$$y_2 = y_1\times\cos(\alpha)+x_1\times\sin(\alpha)$$

Gleichung für die Rotation des Punktes (x_1,y_1) um den Winkel α

Rotation (Fort.)

- die Rotation wird immer um den Koordinatenursprung durchgeführt
- Problem: was muss gemacht werden, wenn der Punkt p_1 mit (x_1,y_2) nicht um $(0,0)$ sondern um (d_x,d_y) gedreht werden soll?
- Antwort:
  1. verschiebe Punkt p_1 um $(-d_x,-d_y)$ (Translation)
  2. rotiere Punkt um den Ursprung
  3. verschiebe neuen Punkt um (d_x,d_y) (Translation)

Ausgangslage: 1. 2. 3.



Matrizen

- bei der Transformation eines Bildes möchte man oft mehrer einzelne Transformationen nacheinander ausführen (Beispiel: Rotation um einen beliebigen Punkt)
- es ist erstrebenswert, einen einheitlichen Mechanismus zu finden, mit denen man alle Transformationen einheitlich beschreiben kann
- da die behandelten Transformationen lineare Abbildung im 2-dimensionalen Vektorraum sind, können die Transformationen als Matrixmultiplikationen durchgeführt werden

Matrizen: Beispiel

- für die x-Scherung gilt:
  - $x_2 = x_1 + y_1 * ShX$
  - $y_2 = y_1$
- wird der Punkt (x_1,y_2) als Spaltenvektor interpretiert, so kann die x-Scherung als folgende 2×2 Matrix verstanden werden

$$\begin{vmatrix} 1 & ShX \\ 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} = \begin{vmatrix} x_1 + ShX \times y_1 \\ y_1 \end{vmatrix}$$

- für die y-Scherung gibt entsprechend:

$$\begin{vmatrix} 1 & 0 \\ ShY & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} = \begin{vmatrix} x_1 \\ ShY \times x_1 + y_1 \end{vmatrix}$$

Matrizen: Beispiel (Fort.)

- soll nun erst eine x-Scherung und dann eine y-Scherung durchgeführt werden, gilt folgendes:

$$\underbrace{\begin{vmatrix} 1 & 0 \\ ShY & 1 \end{vmatrix}}_{\text{y-Scherung}} \times \underbrace{\begin{vmatrix} 1 & ShX \\ 0 & 1 \end{vmatrix}}_{\text{x-Scherung}} \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix}$$

WICHTIG: man beachte die Leseweise von rechts nach links

- da für Matrizen das Assoziativgesetz gilt, können auch erst die beiden Matrizen multipliziert werden:

$$\begin{vmatrix} 1 & 0 \\ ShY & 1 \end{vmatrix} \times \begin{vmatrix} 1 & ShX \\ 0 & 1 \end{vmatrix} = \begin{vmatrix} 1 & ShX \\ ShY & ShY*ShX+1 \end{vmatrix}$$

- dies hat den Vorteil, dass mehrere Punkte immer ***nur mit einer statt*** mit ***zwei*** Matrizen multipliziert werden müssen

Matrizen: Problem

- es ist leicht zu zeigen, dass die x- und y-Scherung, die Rotation und die Skalierung mit 2×2 Matrizen dargestellt werden können
- leider kann die Translation nicht mit einer 2×2 Matrix realisiert werden
- die Translation erfordert eine 3×3 Matrix
- um ein ***einheitliches Schema*** zu bekommen und ***mehrer Transformationen*** mittels ***Matrixmultiplikation*** zu einer Transformation ***zusammenfassen*** zu können, werden alle Transformation durch 3×3 Matrizen realisiert
- dazu müssen die Vektoren von 2 auf 3 Komponenten erweitert werden

Transformations-Matrizen

- Translation:

$$\begin{vmatrix} 1\,0\,d_x \\ 0\,1\,d_y \\ 0\,0\,1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \\ 1 \end{vmatrix} = \begin{vmatrix} x_1+d_x \\ y_1+d_y \\ 1 \end{vmatrix}$$

erweiteter Koordinatenvektor

- Rotation:

$$\begin{vmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \\ 1 \end{vmatrix} = \begin{vmatrix} x_1 \times \cos(\alpha) - y_1 \times \sin(\alpha) \\ x_1 \times \sin(\alpha) + y_1 \times \cos(\alpha) \\ 1 \end{vmatrix}$$

Transformations-Matrizen (Fort.)

- Skalierung:

$$\begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \\ 1 \end{vmatrix} = \begin{vmatrix} x_1 \times S_x \\ y_1 \times S_y \\ 1 \end{vmatrix}$$

- x-Scherung:

$$\begin{vmatrix} 1 & Sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \\ 1 \end{vmatrix} = \begin{vmatrix} x_1 + y_1 \times Sh_x \\ y_1 \\ 1 \end{vmatrix}$$

- y-Scherung:

$$\begin{vmatrix} 1 & 0 & 0 \\ Sh_y & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \\ 1 \end{vmatrix} = \begin{vmatrix} x_1 \\ x_1 \times Sh_y + y_1 \\ 1 \end{vmatrix}$$

Letztes Problem

- bei den Operationen können „Löcher“ in dem Zielbild entstehen
- Beispiel: die beiden Punkte (x_1,x_2) und (x_1+1,x_2+1) werden durch eine Skalierung mit $S_x=3$ und $S_y=3$ auf die folgenden Koordinaten abgebildet:
  - $(3\times x_1,\ 3\times x_2)$
  - $(3+3\times x_1,\ 3+3\times x_2)$
- Frage: welche Punkte werden auf
  - $(1+3\times x_1,\ 1+3\times x_2)$ und
  - $(2+3\times x_1,\ 2+3\times x_2)$ abgebildet?

Löcher !!!

Letztes Problem: Lösung

- Lösung: nicht von der Ursprungskoordinate losrechnet, sondern von Zielkoordinate fragen, welche Ursprungskoordinate auf diese abgebildet wird
- mathematisch ist das leicht geschrieben: statt

$$\begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ y_1 \\ 1 \end{vmatrix} = \begin{vmatrix} x_1 \times S_x \\ y_1 \times S_y \\ 1 \end{vmatrix}$$

- rechnen wir:

$$\begin{vmatrix} x_u \\ y_u \\ 1 \end{vmatrix} = \frac{\begin{vmatrix} x_z \\ y_z \\ 1 \end{vmatrix}}{\begin{vmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{vmatrix}}$$

die Zielkoordinate wird durch die Transformationsmatrix geteilt

????? durch Matrix teilen ????

Letztes Problem: Lösung (Fort.)

- teilen bedeutet: mit dem multiplikativen Inversen multiplizieren, d.h. zu einer Matrix m wird die Matrix m^{-1} gesucht, so dass gilt:

$$m \times m^{-1} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

- im Normalfall existieren diese multiplikativen Inversen von Matrizen nicht, jedoch in diesem Fall sind sie sehr einfach:
- Translation: nicht H und V sondern -H und -V
- Rotation: nicht α sondern - α
- Skalierung: nicht S_x und S_y sondern $1/S_x$ und $1/S_y$
- usw.

Go!

Anwendung

- mit den inversen Matrizen kann jetzt eine Applikation erstellt werden
- soll ein Startbild S durch eine Transformationsmatrix m in ein Zielbild Z transformiert werden, wird folgendes gemacht:
  1. für alle Bildpunkte p_z in Z berechne $m^{-1} \times p_z$
  2. das Ergebnis beschreibt eine Koordinate p_s in S
  3. übertrage den Bildwert von p_s aus S in Z an den Punkt p_z

Vorlesung 4

Zur Erinnerung (letztes Semester): Insertion Sort

```
static  <K extends Comparable<K>> void insertion_sort(K[]field) {
      for(int i1 = 1;i1 < field.length;++i1) {
            final K IVAL = field[i1];
            int i2 = i1;

            while (i2 >= 1 && field[i2 - 1].compareTo(IVAL) > 0) {
                  field[i2] = field[i2 - 1];
                  i2 = i2 - 1;
            }
            field[i2] = IVAL;
      }
}
```

gehe alle bis auf das 1. Element durch

füge sie vorne sortiert ein

Nachteil:

- ein Element kann immer nur 1 Schritt aufrücken
- dadurch dauert es sehr lange, bis kleine Elemente von hinten nach vorne kommen
- Ziel: das muss schneller gehen

Shellsort

Idee:

- basierend auf Insertion Sort
- vergleiche nicht unmittelbar benachbarte Elemente, sondern nehme welche mit großem Abstand und vergleiche diese
- wiederhole das Vorgehen mit kleinerem Abstand
- wenn der Abstand einmal 1 ist, ist es der normale Insertion Sort und damit ist die Folge ***danach*** sortiert

Beispiel

712	452	307	145	102	50	12	7	3	1

schlimmerster Fall für Insertion Sort:
- invertiert sortierte Liste

Idee bei Shellsort:
- betrachte Teillisten, bei der die Nachbarn nur jedes 4. Element sind und sortiere sie nach Insertion Sort, d.h.

712	452	307	145	102	50	12	7	3	1
712	452	307	145	102	50	12	7	3	1
712	452	307	145	102	50	12	7	3	1
712	452	307	145	102	50	12	7	3	1

Beispiel (Fort.)

712	452	307	145	102	50	12	7	3	1
712	452	307	145	102	50	12	7	3	1
712	452	307	145	102	50	12	7	3	1
712	452	307	145	102	50	12	7	3	1

⇒

3	452	307	145	102	50	12	7	712	1
3	1	307	145	102	50	12	7	712	452
3	1	12	145	102	50	307	7	712	452
3	1	12	7	102	50	307	145	712	452

Beispiel (Fort.)

Das Ergebnis der 1. Durchgangs mit 4er Abstand:

3	1	12	7	102	50	307	145	712	452

Beobachtung:
- diese Liste ist wesentlich sortierter, als die Anfangsliste
- die kleinen Elemente sind von rechts nach links gewandert
- die großen Elemente sind von links nach rechts gewandert
- es fanden nur wenige Austausche statt

Nächster Schritt:
- mit Abstand 1 wiederholen, d.h. normalen Insertion Sort

Beispiel (Fort.)

Normaler Insertion Sort:

3	1	12	7	102	50	307	145	712	452

Ergebnis nach einem Durchlauf:

1	3	7	12	50	102	145	307	452	712

- die Liste ist fertig sortiert
- es musste nur jeweils 1 Element verschoben werden, d.h. hier direkter Tausch war möglich

Shell Sort: Abstände

- In diesem Beispiel wurden 2 Abstände gewählt: 4 und 1
- Welche Abstände sollte man im allgemeinen wählen?
  - Bsp.: ..., 1093, 364, 121, 40, 13, 4, 1

    ..., 64, 32, 16, 8, 4, 2,1
- Welche dieser Folgen ist besser?

Ziel:

- eine gute Durchmischung der Vergleiche
- in verschiedenen Durchläufen sollen verschiedene Elemente verglichen werden

Shell Sort: Abstände (Fort.)

- die Wahl der richtigen Abstände ist ganz entscheidend für das Laufzeitverhalten
- es gibt ***keine eindeutig richtige*** Wahl für die Abstände
- es gibt ***aber eindeutig falsche*** Wahlen für Abstände, z.B. 64, 32, 16, 8, 4, 2, 1 da hier immer die gleichen Elemente miteinander verglichen werden
- also: die Folge sollte möglichst ungleichmäßig sein
- somit werden verschiedene Elemente in den verschiedenen Durchläufen miteinander verglichen

Shell Sort: Implementierung

- für den Abstand wird die Variable iDist für Distanz eingeführt
- statt des unmittelbaren Nachbarn wird der Nachbar genommen, der iDist entfernt liegt
- es wird nicht mit dem 1. Element angefangen, sondern mit dem iDist

```
static  <K extends Comparable<K>> void shell_sort(K[] field) {
	for(int i1 = 1;i1 < field.length;++i1) {
		final K IVAL = field[i1];
		int i2 = i1;
		while (i2 >= 1 && field[i2 - 1].compareTo(IVAL) > 0) {
			field[i2] = field[i2 - 1];
			i2 = i2 - 1;
		}
		field[i2] = IVAL;
	}
}
```

diese Stellen müssen geändert werden

Shell Sort: Implementierung (Fort.)

- bisher läuft der Algorithmus einmal über das Feld mit dem Abstand iDist
- iDist muss jetzt noch verringert werden, der Algorithmus muss erneut laufen

```
    static  <K extends Comparable<K>> void shell_sort(K[] field) {
...
        for( ; iDist > 0; iDist /= 3) {
            for(int i1 = iDist; i1 < field.length; ++i1) {
                final K IVAL = field[i1];
                int i2 = i1;
                while (i2 >= iDist && field[i2 - iDist].compareTo(IVAL) > 0) {
                    field[i2] = field[i2 - iDist];
                    i2 = i2 - iDist;
                }
                field[i2] = IVAL;
            }
        }
    }
```

der Abstand wird nach jedem Durchlauf auf ein Drittel reduziert

Shell Sort: Implementierung (Fort.)

- Frage: mit welchem Abstand wird angefangen?

```
static  <K extends Comparable<K>> void shell_sort(K[] field) {
        int iDist = 1;
        for( ; iDist <= field.length / 9; iDist = 3 * iDist + 1) {
        }
        for( ; iDist > 0; iDist /= 3) {
                for(int i1 = iDist; i1 < field.length; ++i1) {
                        final K IVAL = field[i1];
                        int i2 = i1;
                        while (i2 >= iDist && field[i2 - iDist].compareTo(IVAL) > 0) {
                                field[i2] = field[i2 - iDist];
                                i2 = i2 - iDist;
                        }
                        field[i2] = IVAL;
                }
        }
}
```

Vorsicht:
leere Schleife

im 1. Durchlauf sollen maximal 9 Elemente miteinander verglichen werden

Shell Sort: Analyse

- bisher ist unklar, wie schnell Shell Sort arbeitet
- die Geschwindigkeit hängt sehr stark von der Folge der Abstände ab
- die Güte der Abstände hängt aber wiederum von der Vorsortierung ab
- in der Praxis läuft dieser Algorithmus ***sehr gut***
- er ist sehr einfach zu implementieren

Fragen:

1. Ist Shell Sort stabil?
2. Ist Shell Sort für externes Sortieren geeignet?

Distribution Counting

besondere Situation:

- es sollen n natürliche oder ganze Zahlen sortiert werden
- die Zahlen liegen in einem Bereich zwischen 0 und m
- n kann eine sehr große Zahl sein
- m ist eine relativ kleine Zahl

Idee:

- lege ein Feld mit m Einträgen an
- merke in der Stelle i, wieviele Zahlen i in den n Zahlen vorkommen

Distribution Counting: Beispiel

2	3	1	7	5	6	2	7	3	1

besondere Situation:

- es sollen 10 Zahlen sortiert werden
- die Zahlen liegen in einem Bereich zwischen 0 und 8, also [0,8)

0	1	2	3	4	5	6	7
0	2	2	2	0	1	1	2

Ergebnis:

- ein Feld, das sich merkt, wie oft Index als Zahl in der zu sortierenden Folge vorkommt

Distribution Counting: Beispiel (Fort.)

0	1	2	3	4	5	6	7
0	2	2	2	0	1	1	2

nach der "Sortierung": Ausgabe

- 2-mal die 1
- 2-mal die 2
- 2-mal die 3
- 1-mal die 5
- 1-mal die 6
- 2-mal die 7

1	1	2	2	3	3	5	6	7	7

Distribution Counting: Implementierung

```
static void distribution_counting(int[] field, int m) {
    int[] count = new int[m];
    for(int i = 0; i < m; ++i) {
        count[i] = 0;
    }
    for(int i = 0; i < field.length; ++i) {
        ++count[field[i]];
    }
    for(int i1 = 0, i2 = 0; i1 < m; ++i1) {
        for(int i3 = 0; i3 < count[i1]; ++i3) {
            field[i2++] = i1;
        }
    }
}
```

lege zusätzliches Feld an und initialisiere es

zähle die Einträge

speichere die Einträge aus count gezielt wieder in field ab

Distribution Counting: Analyse

```
static void distribution_counting(int[] field, int m) {
    int[] count = new int[m];
    for(int i = 0; i < m; ++i) {
        count[i] = 0;
    }
    for(int i = 0; i < field.length; ++i) {
        ++count[field[i]];
    }
    for(int i1 = 0, i2 = 0; i1 < m; ++i1) {
        for(int i3 = 0; i3 < count[i1]; ++i3) {
            field[i2++] = i1;
        }
    }
}
```

O(m) mit m deutlich kleiner als n (nach Voraussetzung)

O(n)

???

Distribution Counting: Analyse (Fort.)

```
...
        for(int i1 = 0, i2 = 0; i1 < m; ++i1) {
                for(int i3 = 0; i3 < count[i1]; ++i3) {
                        field[i2++] = i1;
                }
        }
...
```

Überlegung:

- die äußere Schleife wird m-mal durchlaufen
- die innere Schleife wird sooft durchlaufen, soviele count[i1]-Zahlen es in der zu sortierenden Folge gibt
- alle count[i1]-Zahlen für alle Einträge können aber nicht mehr als die zu sortierenden Zahlen sein, d.h. $\sum_{i1=0}^{m} \text{count}[i1] = n$
- d.h., die Komplexität und damit Gesamtkomplexität ist O(n)
- Frage: ist das Verfahren stabil?

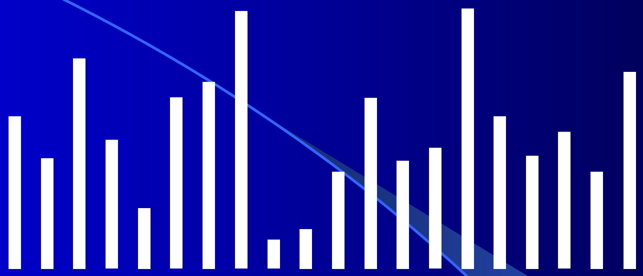
Quicksort

Idee:

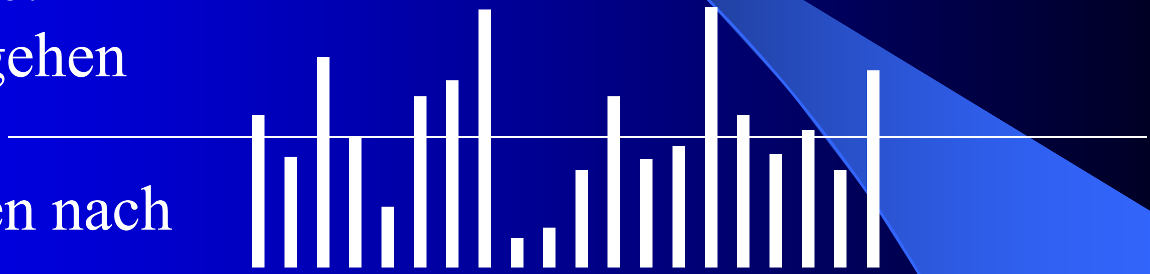
- eine Folge mit nur einem Element ist immer (trivialer weise) sortiert
- hat man mehr Elemente, die zu sortieren sind, teilt man das Problem auf
  - in eine Gruppe kommen alle großen Elemente
  - in eine Gruppe kommen alle kleinen Elemente
  - sortiere die beiden Gruppen jede für sich
  - danach ist die gesamte Folge sortiert

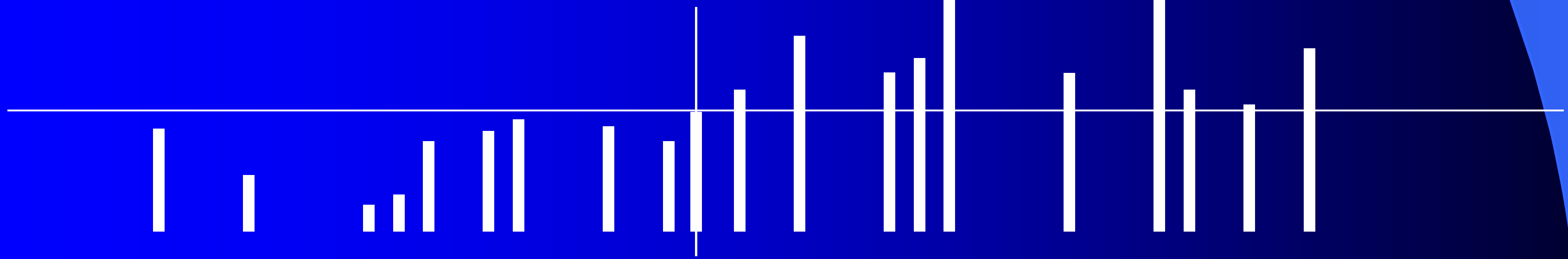
Quicksort: Illustration

Ausgangssituation:

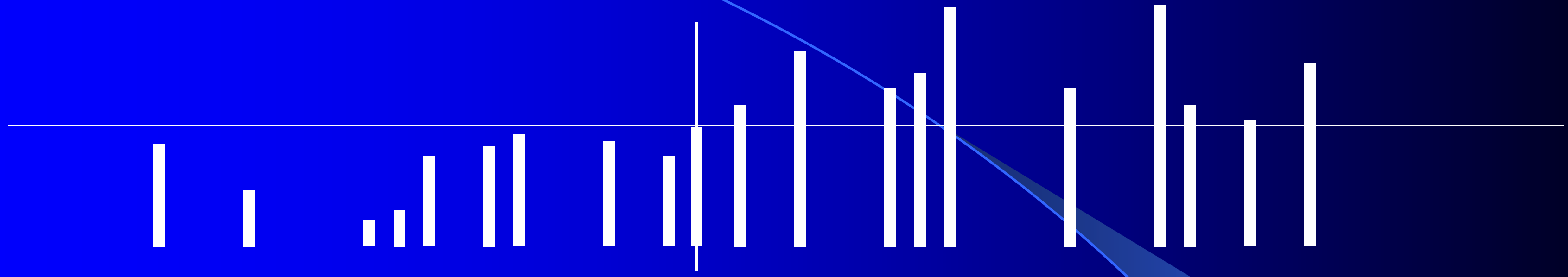


Zerlege gemäß der Line:
- alle, die größer sind gehen nach rechts,
- alle kleineren kommen nach links
- in der Mitte bleibe die, die genauso groß sind

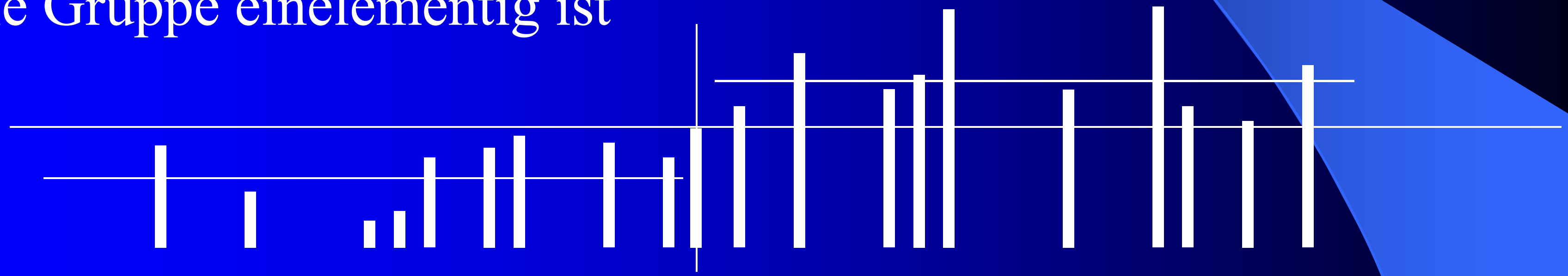




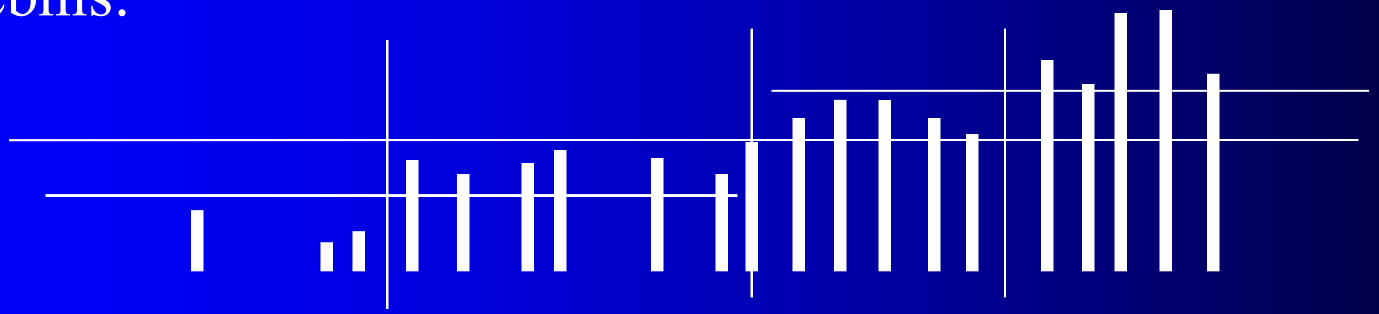
Quicksort: Illustration (Fort.)



Mache jetzt mit der linken und rechten Gruppe weiter, bis die Gruppe einelementig ist



Ergebnis:



Quicksort: Implementierung

```
static  <K extends Comparable<K>> void quick_sort(K[] field) {
        quick_sort_help(field,0,field.length-1);
}
static  <K extends Comparable<K>> void quick_sort_help(K[] field, int iLeft, int iRight) {
        final K MID = field[(iLeft + iRight) / 2];
        int l = iLeft;
        int r = iRight;

        while(l < r) {
                while(field[l].compareTo(MID) < 0) { ++l; }
                while(field[r].compareTo(MID) > 0) { --r; }
                if(l <= r)
                        swap(field, l++, r--);
        }
        if (iLeft  < r)
                quick_sort_help(field, iLeft, r );
        if (iRight > l)
                quick_sort_help(field, l, iRight);
}
```

ruft Hilfs-funktion mit maximalen Grenzen auf

nach MID müssen sich alle richten

suche Elemente, die noch vertauscht werden müssen

sortiere rekursiv die beiden restlichen Teile, wenn notwendig

Vorlesung 5

Quicksort: Analyse

Optimaler Fall:

- field[(iLeft+iRight)/2] liegt in der Mitte, d.h. es gibt genauso viele kleinere wie größere Elemente
- d.h. nach einem Durchlauf wird das Problem der Größe N auf 2 Probleme jeweils der Größe N/2 reduziert
- d.h. $N + 2 * O(N/2) = N*\log(N)$

```
static  <K extends Comparable<K>>
    void quick_sort_help(K[]field,
                         int iLeft,
                         int iRight) {
    final K MID = field[(iLeft + iRight) / 2];
    int l = iLeft;
    int r = iRight;

    while(l < r) {
        while(field[l].compareTo(MID) < 0) { ++l; }
        while(field[r].compareTo(MID) > 0) { --r; }
        if(l <= r)
            swap(field, l++, r--);
    }
    if (iLeft  < r)
        quick_sort_help(field, iLeft, r );
    if (iRight > l)
        quick_sort_help(field, l, iRight);
}
```

Quicksort hat im Durchschnitt eine Komplexität von O(N log N)

Quicksort: Analyse (Fort.)

Schlechter Fall:

- field[(iLeft+iRight)/2] ist das kleinste oder größte Element
- d.h. nach einem Durchlauf wird das Problem der Größe N auf 2 Probleme der Größe N-1 und 1 reduziert
- d.h. $N + O(N-1) + O(1) = N^2$

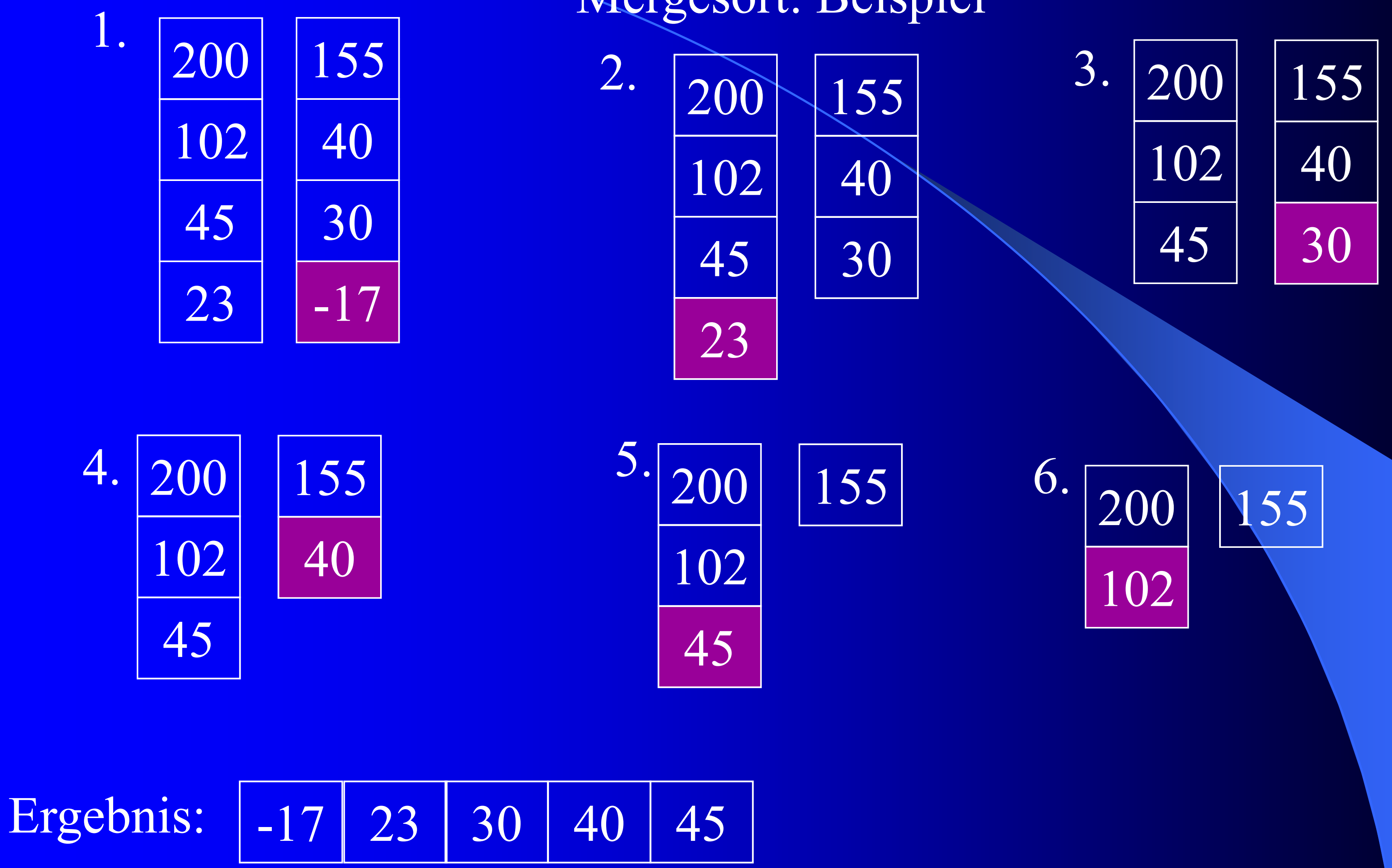
```
static  <K extends Comparable<K>>
        void quick_sort_help(K[] field,
                             int iLeft,
                             int iRight) {
        final K MID = field[(iLeft + iRight) / 2];
        int l = iLeft;
        int r = iRight;

        while(l < r) {
                while(field[l].compareTo(MID) < 0) { ++l; }
                while(field[r].compareTo(MID) > 0) { --r; }
                if(l <= r)
                        swap(field, l++, r--);
        }
        if (iLeft  < r)
                quick_sort_help(field, iLeft, r );
        if (iRight > l)
                quick_sort_help(field, l, iRight);
}
```

Quicksort hat im schlimmsten Fall eine Komplexität von $O(N^2)$

Mergesort

- Idee:
  - wenn man zwei sortierte Listen hätte, dann könnte man eine neue sortierte Liste erzeugen, indem
  - man das kleinste Element der beiden Köpfe nimmt,
  - dieses entfernt
  - und mit dem Rest weitermacht

Mergesort: Beispiel



Mergesort: Implementierung

```
static  <K extends Comparable<K>> void merge_sort_help(K[] field,int iLeft,int iRight) {
    if (iLeft < iRight) {
        final int MIDDLE = (iLeft + iRight) / 2;
        merge_sort_help(field, iLeft, MIDDLE);
        merge_sort_help(field, MIDDLE + 1, iRight);

        K[] tmp = (K[]) new Comparable[iRight - iLeft + 1];

        for(int i = iLeft; i <= MIDDLE; ++i)
            tmp[i - iLeft] = field[i];
        for(int i = MIDDLE+1; i <= iRight; ++i)
            tmp[tmp.length-i+MIDDLE] = field[i];

        int iL = 0;
        int iR = tmp.length-1;
        for(int i = iLeft; i <= iRight; ++i) {
            field[i] = tmp[iL].compareTo(tmp[iR]) < 0 ? tmp[iL++] : tmp[iR--];
        }
    }
}
```

die Mitte

sortiere links und rechts der Mitte

lege eine Kopie an, drehe dabei die 2. Hälfte um

mische aus der Kopie in das Originalfeld

Mergesort: Analyse

- im Gegensatz zu Quicksort wird bei Mergesort das Feld immer genau in 2 gleichgroße Teile zerlegt
- beim Mischen wird O(N) Zeit verbraucht
- somit ergibt sich eine Gesamtkomplexität von O(N log N)
- der zusätzliche Speicheraufwand beträgt O(N)
- da beim Mischen immer nur auf den Kopf von 2 Läufern zugegriffen wird, eignet sich dieses Verfahren zum externen Sortieren
- im Durchschnitt ist das Verfahren langsamer als Quicksort

Der Mergesort hat garantiert ein O(N log N) Verhalten

Heapsort

Sei A eine Datenstruktur mit folgenden Eigenschaften:

- bei der Initialisierung sagt man, wieviele Elemente gespeichert werden sollen
- es gibt eine Methode insert(), der man das zu sortierende Element mitgibt
- es gibt eine Methode remove(), die das größte Element ***zurückliefert und*** dieses auch noch ***entfernt***

Dann könnte man wie folgt sortieren:

Heapsort (Fort.)

pField enthält N Elemente, die zu sortieren sind

```
static  <K extends Comparable<K>>
void sort(K[] pField) {
        A a(pField.length);
        for(int i = 0; i < pField.length; ++i)
                a.insert(pField[i]);
        for(int i = 0; i < pField.length; ++i)
                pField[pField.length - i - 1] =
                        a.remove();
}
```

füge alle Elemente ein

lese alle Elemente sortiert aus (mit dem größten beginnend)

Gesucht ist eine solche Datenstruktur A

Heapsort (Fort.)

Möglichkeiten für eine solche Datenstruktur A:

- eine unsortierte Liste
  - insert erfolgt am Ende: Komplexität O(1)
  - remove durchläuft die Liste und sucht das Maximum: Komplexität O(N)
  - dies würde dem ***Selection Sort*** entsprechen: Komplexität $O(N^2)$

- eine sortierte Liste
  - insert erfolgt sortiert in der Liste: Komplexität O(N)
  - remove entfernt das letzte Element: Komplexität O(1)
  - dies würde dem ***Insertion Sort*** entsprechen: Komplexität $O(N^2)$

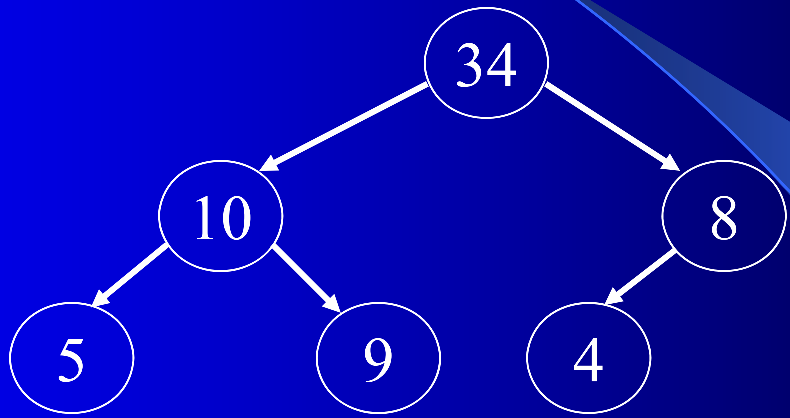
Heapsort (Fort.)

andere Möglichkeiten für eine solche Datenstruktur A:

- ein binärer Baum mit der folgenden Eigenschaft
- jeder Knoten enthält einen zu sortierenden Schlüssel
- der Schlüssel eines jeden Knoten ist größer oder gleich der Schlüssel seiner Söhne
- der Baum ist ausgeglichen, d.h. der Unterschied zwischen dem längsten und dem kürzestem Pfad von der Wurzel zu den Blättern beträgt maximal 1
- eine solche Datenstruktur nennt man ***Heap***

Heapsort (Fort.)

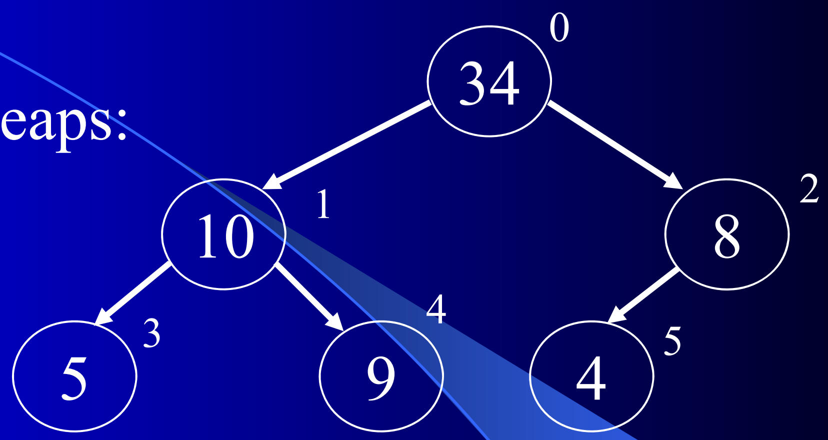
Beispiel für einen solchen Baum / Heap:



- jeder Knoten enthält einen Schlüssel, der größer als die seiner Söhne sind
- die Länge der Pfade zu den Blättern unterscheiden sich maximal um 1

Heapsort (Fort.)

Darstellung solcher Bäume / Heaps:



- wenn bekannt ist, wieviele Knoten maximal abgespeichert werden, können der Baum in einem Array abgespeichert werden

34	10	8	5	9	4
0	1	2	3	4	5

- von einem Knoten mit Index k wird auf die Söhne mittels 2*k+1 und 2*k+2 zugegriffen
- von einem Knoten mit Index k wird auf den Vater mittels (k-1)/2 zugegriffen

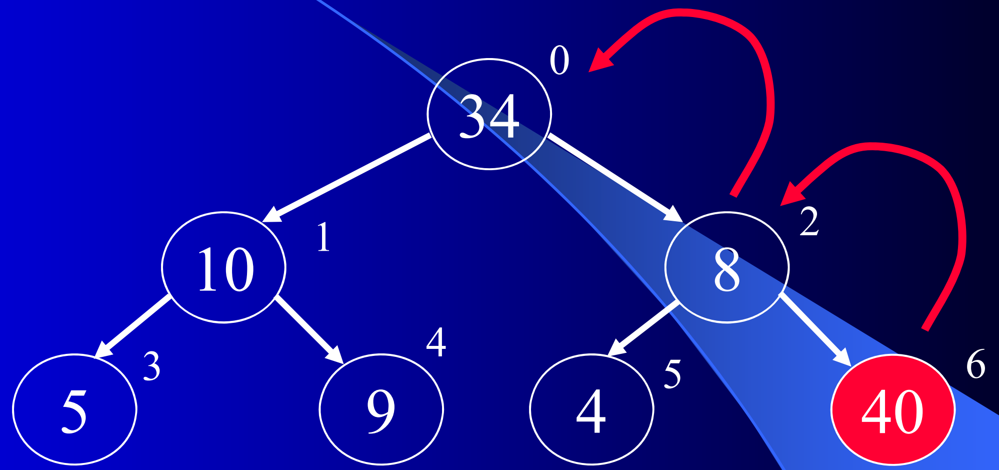
Heapsort (Fort.)

Implementierung eines Heaps für Comparable-Werte:

```
class Heap<K extends Comparable<K>> {

    public Heap(int iSize) {
        m_iNext = 0;
        m_Keys = (K[])new Comparable[iSize];
    }

    private int     m_iNext;        // der nächste freie Index
    private K[]     m_Keys;         // die einzelnen Schlüssel
}
```

Heapsort (Fort.)

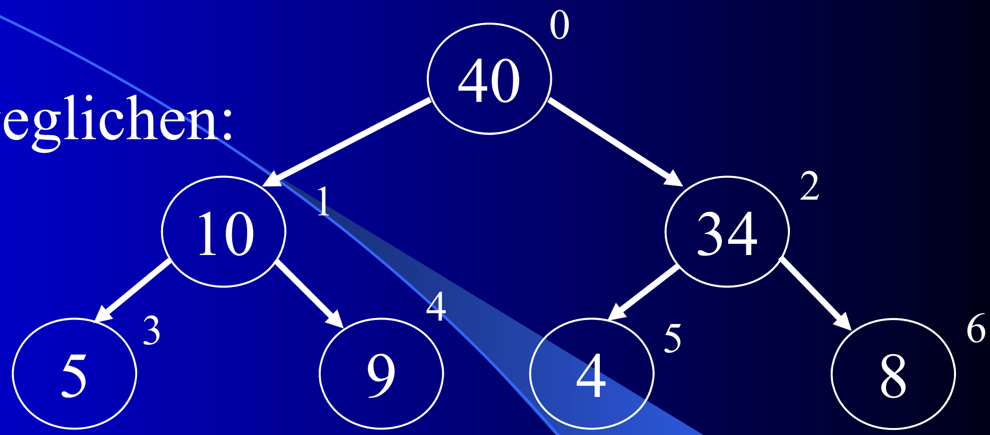
Einfügen eines Elements in einen solchen Baum:



- das neue Element wird am Ende des Arrays, sprich unten im Baum eingefügt
- dadurch verliert der Baum u.U. seine Eigenschaft, dass alle Knoten größere Schlüssel als ihre Söhne haben
- solche Schlüssel müssen dann nach oben wandern

Heapsort (Fort.)

dieser Baum ist wieder ausgeglichen:



- das nach-oben-wandern wird von der folgenden Methode upheap erledigt

```
private void upheap(int iIndex) {
    K k = m_Keys[iIndex];
    while (iIndex != 0 && m_Keys[(iIndex-1) / 2].compareTo(k) < 0) {
        m_Keys[iIndex] = m_Keys[(iIndex-1) / 2];
        iIndex = (iIndex - 1) / 2;
    }
    m_Keys[iIndex] = k;
}
```

Heapsort (Fort.)

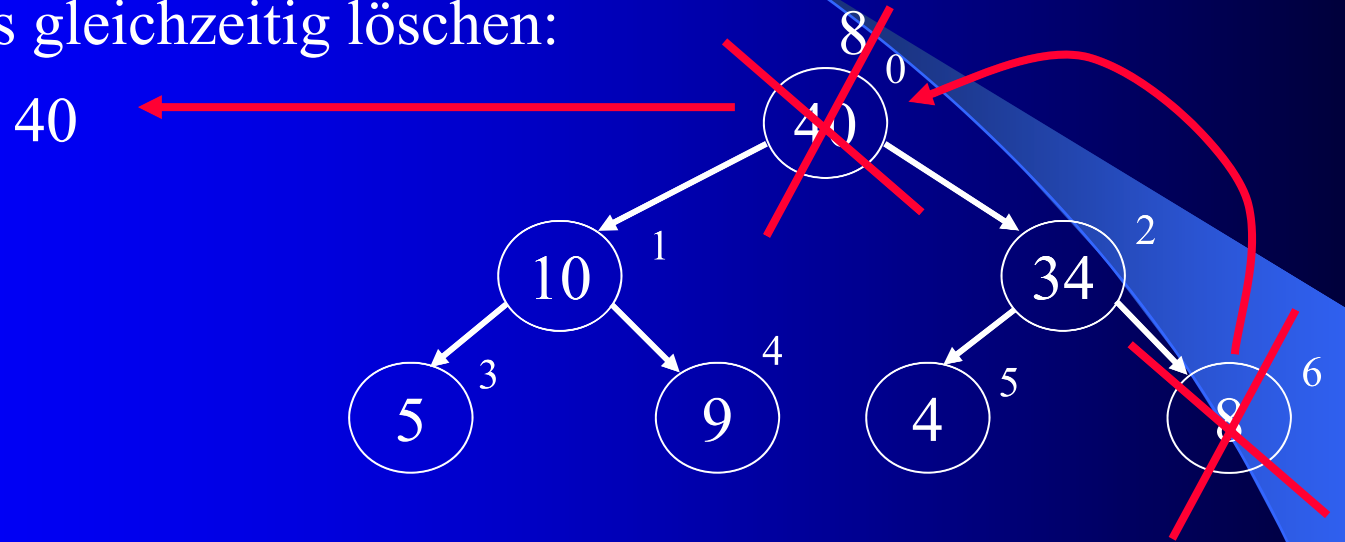
- basierend auf der upheap Methode kann die Insert Methode wie folgt implementiert werden:

```
public void insert(K key) {
        m_Keys[m_iNext] = key;
        upheap(m_iNext);
        ++m_iNext;
}
```

- zunächst wird das neue Element am Ende eingefügt
- dann wird die damit verbundene Unordnung wieder hergestellt
- am Ende wird der nächste freie Index um 1 erhöht

Heapsort (Fort.)

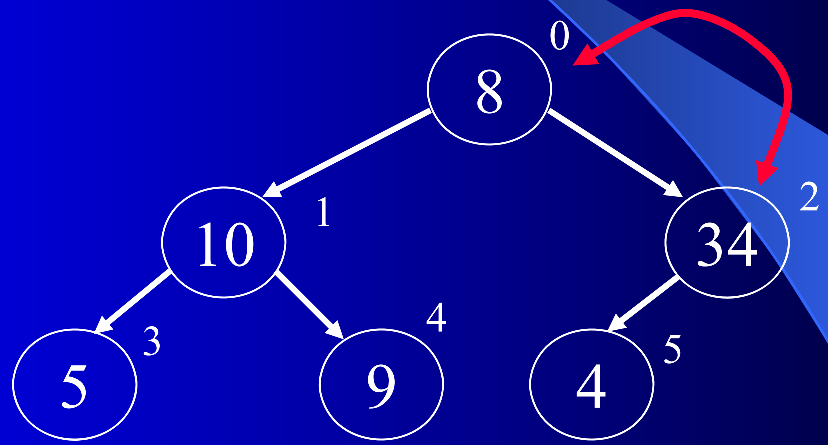
- die remove Methode soll das größte Element zurückliefern
- und es gleichzeitig löschen:



- das größte Element ist an der Spitze
- um es zu löschen, wird das letzte Element an dessen Stelle gesetzt

Heapsort (Fort.)

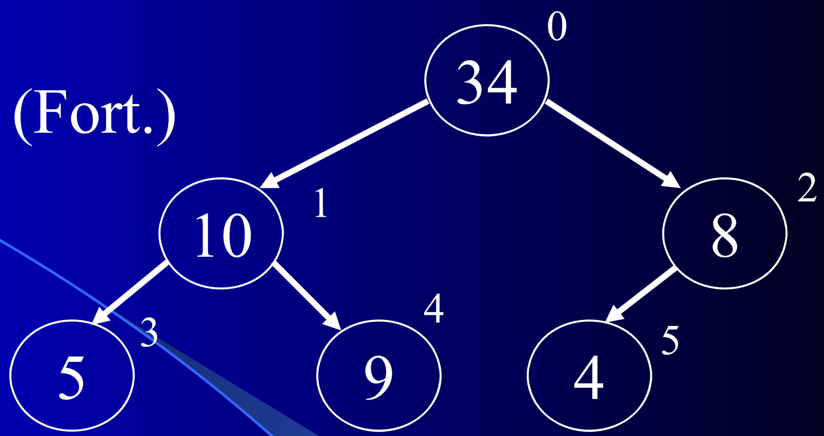
- der resultierende Baum ist nicht mehr korrekt
- die Spitze wird jetzt i.d.R. nicht mehr größer sein als die beiden Söhne



- solche Elemente müssen jetzt nach unten wandern
- ein Knoten wird dazu mit dem maximalen Sohn ausgetauscht

Heapsort (Fort.)

dieser Baum ist wieder ausgeglichen:



- das nach-unten-wandern wird von downheap erledigt

```
private void downheap(int iIndex) {
    K k = m_Keys[iIndex];
    while (iIndex < m_iNext / 2) {
        int iSon = 2 * iIndex + 1;
        if (iSon < m_iNext-1 &&
            m_Keys[iSon].compareTo(m_Keys[iSon + 1]) < 0)
            ++iSon;
        if (k.compareTo(m_Keys[iSon]) >= 0) break;
        m_Keys[iIndex] = m_Keys[iSon];
        iIndex = iSon;
    }
    m_Keys[iIndex] = k;
}
```

es gibt noch einen Sohn

1. Sohn

größter Sohn

Heapsort (Fort.)

- basierend auf der downheap Methode kann die Remove Methode wie folgt implementiert werden:

```
public K remove() {
        K res = m_Keys[0];
        m_Keys[0] = m_Keys[--m_iNext];
        downheap(0);
        return res;
}
```

- zunächst wird das erste (größte) Element zwischengespeichert
- dann wird das letzte Element an die vorderste Front gestellt
- der inkonsistente Zustand wird durch das Runterwandern des 1. Elements wieder korrigiert

Heapsort (Fort.)

- mit den beiden Methoden insert und remove ist jetzt ein Sortierverfahren implementiert
- jede der beiden Operationen benötigt O(log N) Schritten, da der Binärbaum ausgeglichen ist
- somit ist die Gesamtlaufzeit O(N log N)
- leider wird ein zusätzlicher Platz von O(N) benötigt

Heapsort braucht garantiert nur O(N log N) Zeit, ist im Durchschnitt aber ein bisschen langsamer als Quicksort

```
static <K extends Comparable<K>>
void heap_sort(K[] field)
    Heap<K> a = new Heap<K>(field.length);
    for(int i = 0;i < field.length;++i)
        a.insert(field[i]);
    for(int i = 0;i < field.length;++i)
        field[field.length - i - 1] = a.remove();
}
```

Heapsort (Fort.)

- Heapsort kann derart modifiziert werden, dass ohne zusätzlichen Speicherplatz sortiert werden kann
- Idee:
  - betrachte jeden Teilbaum von unten aufsteigend und mache ihn zum Heap, d.h. jeder Knoten muss einen größeren Schlüssel als seine Söhne haben (ist für die Blätter trivialerweise erfüllt)
  - jetzt steht das maximale Element am Anfang
  - vertausche das maximale Element mit dem letzten Element und stelle die Heapeigenschaft für das um 1 verkleinerte Array wieder her

Heapsort (Fort.)

- die Methode heapsort stützt sich auf eine Funktion downheap ab
- die Argumente
  - das Array, das den Heap enthält
  - die Anzahl der Elemente in dem Heap
  - den Index des Elements, dass jetzt in dem Heap runterwandern soll

```
static <K extends Comparable<K>>
void heap_sort(K field) {
    for(int i = (field.length-1) / 2; i >= 0; --i)
        Heap.downheap(field, field.length, i);
    for(int i = field.length-1; i > 0; --i) {
        swap(field, 0, i);
        Heap.downheap(field, i, 0);
    }
}
```

vordefinierte Sortierverfahren in Java und C++

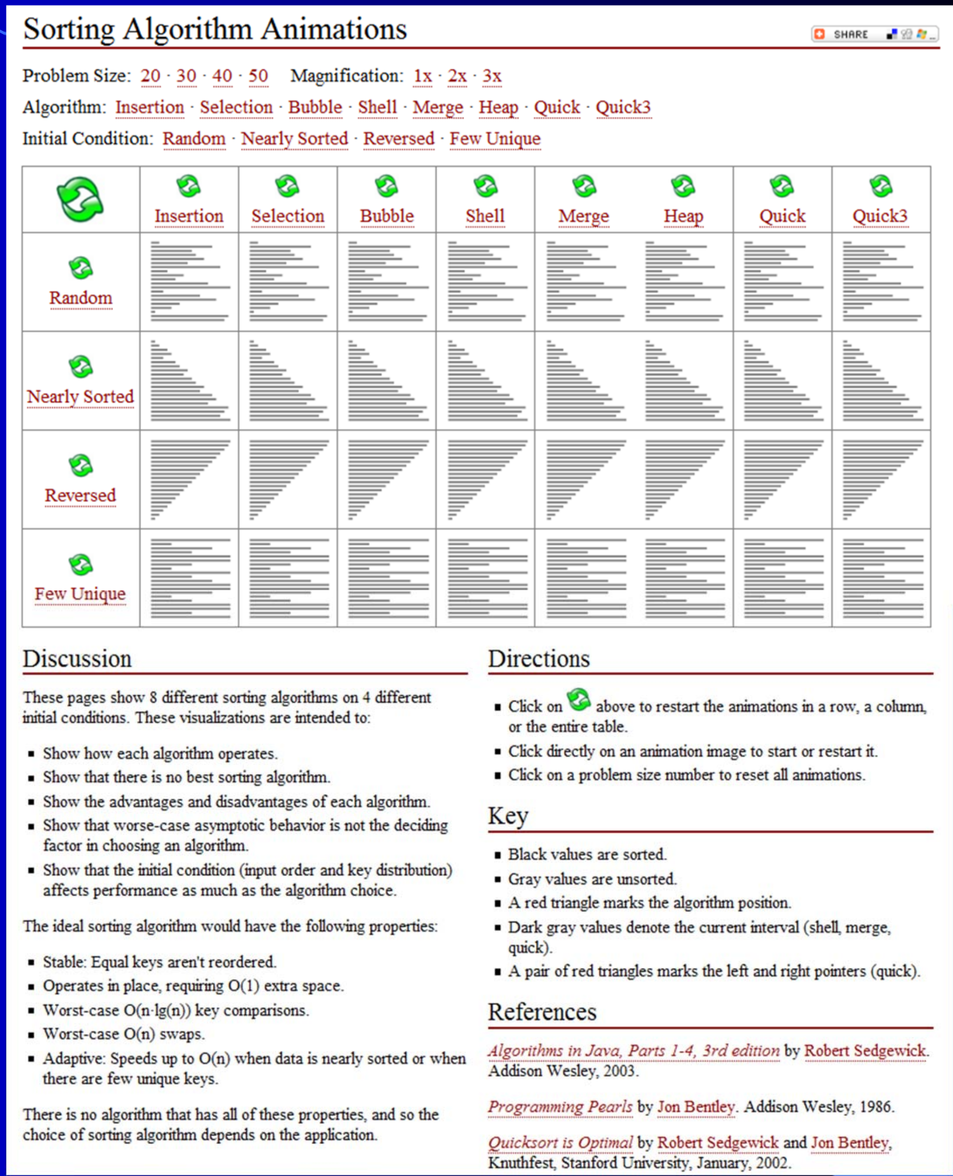
- in Java:

```
class Collection {
    static void sort(List list);
    static void sort(List list, Comparator c);
}
```

- in C++: sort, stable_sort, partial_sort, partial_sort_copy

Sortierverfahren: ein Vergleich

- Ein animierter Vergleich der vorgestellten Sortierverfahren findet man hier:
- http://www.sorting-algorithms.com/



Sorting Algorithm Animations

SHARE

Problem Size: 20 · 30 · 40 · 50 Magnification: 1x · 2x · 3x
Algorithm: Insertion · Selection · Bubble · Shell · Merge · Heap · Quick · Quick3
Initial Condition: Random · Nearly Sorted · Reversed · Few Unique

	Insertion	Selection	Bubble	Shell	Merge	Heap	Quick	Quick3
Random								
Nearly Sorted								
Reversed								
Few Unique								

Discussion

These pages show 8 different sorting algorithms on 4 different initial conditions. These visualizations are intended to:

- Show how each algorithm operates.
- Show that there is no best sorting algorithm.
- Show the advantages and disadvantages of each algorithm.
- Show that worse-case asymptotic behavior is not the deciding factor in choosing an algorithm.
- Show that the initial condition (input order and key distribution) affects performance as much as the algorithm choice.

The ideal sorting algorithm would have the following properties:

- Stable: Equal keys aren't reordered.
- Operates in place, requiring O(1) extra space.
- Worst-case O(n·lg(n)) key comparisons.
- Worst-case O(n) swaps.
- Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

Directions

- Click on above to restart the animations in a row, a column, or the entire table.
- Click directly on an animation image to start or restart it.
- Click on a problem size number to reset all animations.

Key

- Black values are sorted.
- Gray values are unsorted.
- A red triangle marks the algorithm position.
- Dark gray values denote the current interval (shell, merge, quick).
- A pair of red triangles marks the left and right pointers (quick).

References

Algorithms in Java, Parts 1-4, 3rd edition by Robert Sedgewick. Addison Wesley, 2003.

Programming Pearls by Jon Bentley. Addison Wesley, 1986.

Quicksort is Optimal by Robert Sedgewick and Jon Bentley, Knuthfest, Stanford University, January, 2002.

Vorlesung 6

Suchen

Aufgabe:

- zu einer Information ***K*** soll überprüft werden, ob eine assoziierte Information ***D*** existiert
- falls ja, so soll ***D*** zurückgeliefert werden
- ***K*** nennt man ***Schlüssel***
- ***D*** den assoziierten ***Datensatz***
- zu ***einem Schlüssel*** kann es ***mehrere Datensätze*** geben

Weitere Aufgaben:

- einen neuen Datensatz mit Schlüssel einfügen
- alle Datensätze mit gegebenen Schlüssel löschen
- eine leere Datenstruktur anlegen

Sequentielles Suchen

- einfachstes Suchverfahren
- Idee: lege alle Elemente hintereinander ab
- suche dann sequentiell vom Anfang bis zum Ende oder bis der gegebene Schlüssel gefunden ist

Am Ende werden neue Datensätze mit Schüsseln eingefügt

Schlüssel	34	17	-5	40	34	3	-15	13
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre

Suchen beginnt am Anfang (z.B. nach –5)

Sequentielles Suchen: Implementierung

```
class Node<K extends Comparable<K>,D> {

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

public SeqSearch(int iNrOfEntries) {
    m_iNextFree = 0;
    m_pData = new Node[iNrOfEntries];
}
....
```

Subklasse, die sich ein Schlüssel/Daten Paar merkt

Zu Beginn muss bereits feststehen, wieviele Datensätze ***maximal*** verwaltet werden sollen

Sequentielles Suchen: Implementierung (Fort.)

```
...
    public void insert(K key,D data) {
        m_pData[m_iNextFree++] = new Node<K,D>(key,data);
    }

    public Node<K,D> search(K key) {
        for(int i = 0;i < m_iNextFree;++i)
            if (key.compareTo(m_pData[i].m_Key) == 0)
                return m_pData[i];
        return null;
    }

    private int             m_iNextFree;
    private Node<K,D>[]     m_pData;
}
```

Das Einfügen erfolgt am Ende

Durchsucht wird vom Anfang alle bisher ein-gefügten Datensätze

Der 1. Datensatz mit Schlüssel key wird zurückgeliefert

Ist der Schlüssel nicht vorhanden, wird null zurückgeliefert

Sequentielles Suchen: Komplexität

- Das Einfügen ist konstant (weil am Ende), erfolgt also in O(1)
- Das Suchen
  - wenn der Schlüssel ***nicht vorhanden*** ist, müssen alle Einträge überprüft werden, also O(N)
  - wenn der Schlüssel vorhandeln ist, so findet man ihn im Durchschnitt nach N/2 Vergleichen, also auch O(N)

Nachteil

- bei mehreren Datensätzen gleichen Schlüssels wird nur der erste gefunden

Vorteil

- dieses Verfahren eignet sich auch für Listen

Binäres Suchen

Voraussetzung:

- die Daten sind nach ihren Schlüsseln sortiert

Idee (Divide and Conquer):

- zerlege den Suchraum in zwei Teile
- bestimme den Teil, in dem der Schlüssel ***enthalten sein kann***
- fahre mit diesem Teil fort

Binäres Suchen (Fort.)

hier mit sortierter Folge von Schlüsseln:

- vergleiche Schlüssel mit dem des mittleren Datensatzes
- ist er kleiner, suche in der 1. Hälfte, ansonsten in der 2. Hälfte

Nach 3 Vergleichen ist die 17 gefunden

hier muss die 17 sein

2. Vergleich

	0	1	2	3	4	5	6	7
Schlüssel	-15	-5	3	13	17	34	34	40
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre

Schlüssel sind sortiert

gesucht wird nach 17

1. Vergleich

hier muss die 17 sein

Binäres Suchen: Implementierung

```
public class BinSearch<K extends Comparable<K>,D> {

    class Node<K,D> {...}
    public BinSearch(int iNrOfEntries) {...}

    public Node<K,D> search(K key) {
        int iL = 0;
        int iR = m_iNextFree-1;
        while (iL <= iR) {
            final int MIDDLE = (iL + iR) / 2;
            final int RES = m_pData[MIDDLE].m_Key.compareTo(key);
            if (RES == 0)
                return m_pData[MIDDLE];
            else if (RES < 0)
                iL = MIDDLE+1;
            else
                iR = MIDDLE-1;
        }
        return null;
    }
...
}
```

Alles wie bei SeqSearch

iL und iR sind der linke und rechte Rand

Datensatz ist gefunden

mache rechts weiter

mache links weiter

Datensatz ist nicht gefunden

Binäres Suchen: Komplexität

- Das Einfügen
  - ist kompliziert, da immer sortiert eingefügt werden muss
  - erfolgt also in O(N) (siehe Insertion Sort)
  - Einfügen von N Elementen ist also $O(N^2)$

- Das Suchen
  - in jeden Schritt wird der Suchraum halbiert
  - somit ist man im schlimmsten Fall nach O(log N) Schritten fertig

Binäres Suchen: Diskussion

Nachteil

- das Verfahren eignet sich nicht für Listen
- das Einfügen ist kompliziert

Vorteil

- auch in sehr großen Datenmengen kann noch schnell gesucht werden
- gut geeignet, wenn erst alle Elemente eingefügt werden bevor das erste Element gesucht wird

Warum?

Interpolationssuche

Idee:

- das binäre Suchen verbessern
- in einem Telefonbuch schlägt man auch nicht die Mitte auf, wenn nach "Buchholz" gesucht wird
- anhand des Schlüssels schauen, in welchem Bereich die höhere Change besteht, dass der gesuchte Schlüssel dort liegt

Wichtig:

- für den Schlüssel muss eine Metrik existieren, d.h. die Differenz zwischen zwei Schlüsseln muss existieren

Interpolationssuche: Implementierung

Zerlegung beim binären Suchen:

```
final int MIDDLE = (iL + iR) / 2;
```

ist identisch zu:

```
final int MIDDLE = iL + 1 * (iR - iL) / 2;
```

Vorsicht: spezielle Behandlung, wenn der Schlüssel außerhalb des Bereichs liegt

bei der Interpolationssuche:

Schlüssel sind hier int

```
int keyL = m_pData[iL].m_Key;
int keyR = m_pData[iR].m_Key;
final int MIDDLE =
        iL + (key - keyL) * (iR - iL) / (keyR - keyL);
```

Der gesuchte Schlüssel **key** wird zu den linken und rechten Schlüssel in Beziehung gesetzt

Interpolationssuche: Beispiel

```
int keyL = m_pData[iL].m_Key;
int keyR = m_pData[iR].m_KeyM;
final int MIDDLE = iL + (key - keyL) * (iR - iL) / (keyR - keyL);
```

gesucht wird nach -5

iL = 0
iR = 7
keyL = -15
keyR = 40
MIDDLE = 0 + (-5--15)*(7-0)/(40--15) = 1

Schlüssel	-15	-5	3	13	17	34	34	40
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre
	0	1	2	3	4	5	6	7

1. Vergleich

Nach 1 Vergleich ist die -5 gefunden

Interpolationssuche: Komplexität

- Das Einfügen
  - ist so kompliziert wie bei der Binären Suche, da die Schlüssel sortiert sein müssen

- Das Suchen
  - die erfolglose Suche dauert im Durchschnitt O(log log N)
  - das ist sehr wenig, z.B. gilt bei einer Milliarde Datensätzen: log log N < 5

Interpolationssuche: Diskussion

Nachteil (wie bei Binärer Suche)

- das Verfahren eignet sich nicht für Listen
- das Einfügen ist kompliziert
- für die Schlüssel muss es eine Metrik geben, d.h. der Abstand zweier Schlüssel muss sich einfach bestimmen lassen

Wie sieht es bei Strings aus?

Vorteil

- auch in sehr großen Datenmengen kann sehr schnell (fast konstant) gesucht werden
- ist nicht wesentlich komplizierter als das Binäre Suchen

Suchen in binären Bäumen

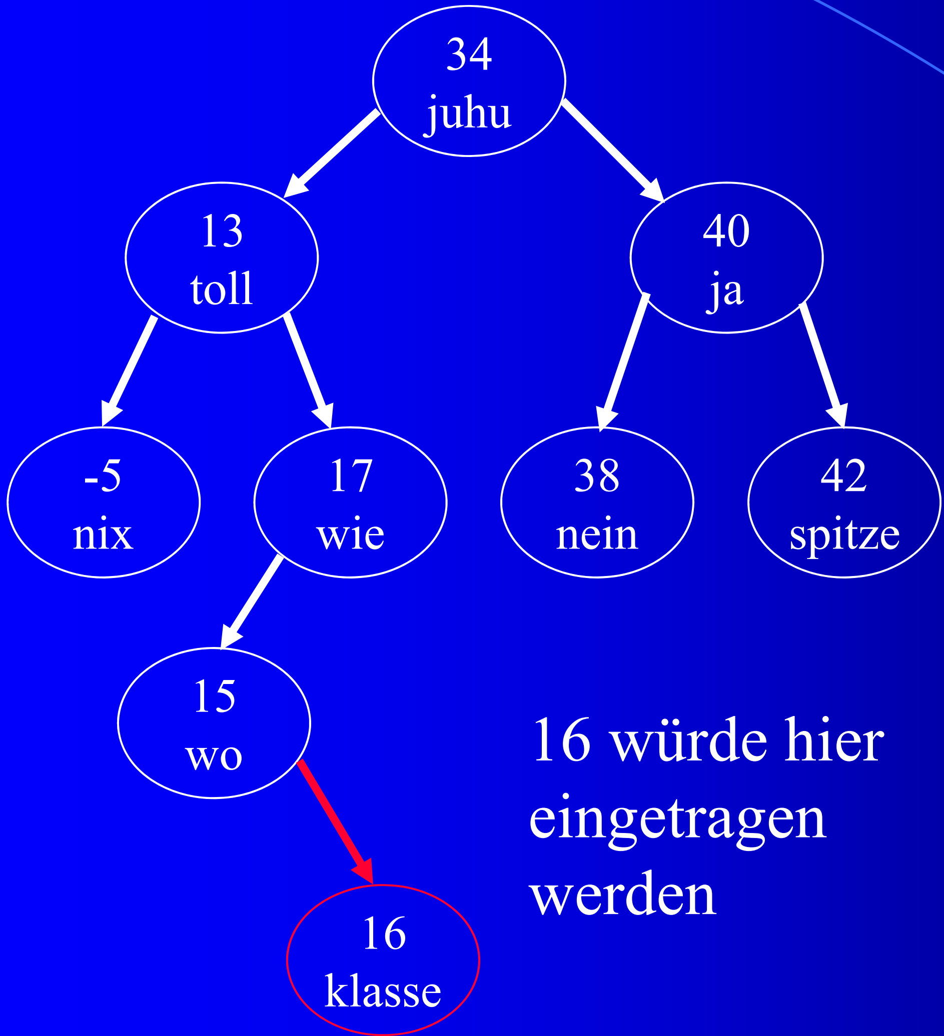
gesucht wird eine Datenstruktur:

- in der man schnell (O(log N)) suchen und schnell einfügen (O(log N)) kann

binärer Baum:

- jeder Knoten besitzt einen Schlüssel und den zugehörigen Datensatz
- jeder Knoten hat maximal 2 Nachfolger (links und rechts)
- in den ***linken Teilbaum*** gibt es nur Knoten mit ***kleineren Schlüsseln***
- in dem ***rechten Teilbaum*** gibt es nur Knoten mit ***größeren Schlüsseln***

Suchen in binären Bäumen (Fort.)



16 würde hier eingetragen werden

- binärer Baum mit 8 Einträgen
- eingefügt wird absteigend von der Spitze
- gesucht wird ebenfalls absteigend von der Spitze
- ist der gesuchte Schlüssel kleiner, gehe in den linken Teilbaum
- ist der gesuchte Schlüssel größer, gehe in den rechten Teilbaum

Suchen in binären Bäumen: Implementierung

```
class BinTree<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;
    }

    ...

    private Node m_Root = null;
}
```

Ein Knoten im binären Baum merkt sich

- den Schlüssel,
- den Datensatz,
- den linken und rechten Nachfolger

Der Baum merkt sich nur die Wurzel und ist zunächst leer

Suchen in binären Bäumen: Implementierung (Fort.)

```
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, kompletten Knoten zurückgeben

steige links bzw. rechts ab

Schlüssel ist nicht gefunden worden

Einfügen in binären Bäumen: Implementierung

```
public void insert(K key,D data) {
    Node tmp = m_Root;
    Node father = null;
    while (tmp != null) {
        father = tmp;
        tmp = (key.compareTo(tmp.m_Key) < 0)
                ? tmp.m_Left
                : tmp.m_Right;
    }
    tmp = new Node(key,data);
    if (father == null)
        m_Root = tmp;
    else if (key.compareTo(father.m_Key) < 0)
        father.m_Left = tmp;
    else
        father.m_Right = tmp;
}
```

father merkt sich den Vorgänger von tmp

steige links bzw. rechts ab

tmp ist jetzt garantiert null

der Baum war leer

erzeuge neuen Knoten und speichere ihn unter father ab

Einfügen in binären Bäumen: Implementierung (Forts.)

- Das Merken des Vorgängers ist symmtomatisch für alle Implementierungen von Bäumen für unterschiedliche Einfüge- und Löschoperationen
- Daher sollte dies nur einmal implementiert werden
- Hier könnte eine NodeHandler Klasse hilfreich sein, dessen Aufgabe ist
  - Vorgänger merken
  - selber feststellen, ob ein neuer Knoten rechts oder links unter den Vorgänger eingefügt werden muss

NodeHandler Klasse

```
class NodeHandler {
    Node m_Dad = null;
    Node m_Node = null;

    NodeHandler(Node n) {
        m_Node = n;
    }

    void down(boolean left) {
        m_Dad = m_Node;
        m_Node = left ? m_Node.m_Left : m_Node.m_Right;
    }

    boolean isNull() {
        return m_Node == null;
    }

    K key() {
        return m_Node.m_Key;
    }
```

aktueller Knoten und Vorgänger

Initialisierug durch aktuellen Knoten; Vorgänger ist null

Abstieg: links oder rechts

gibt es noch einen aktuellen Knoten

Schlüssel des aktuellen Knotens

NodeHandler Klasse (Forts.)

```
Node node() {
    return m_Node;
}

void set(Node n) {
    assert n != null || m_Node != null;
    if (m_Dad == null)
        m_Root = n;
    else if (m_Node != null ?
            m_Node == m_Dad.m_Left :
            n.m_Key.compareTo(m_Dad.m_Key) < 0)
        m_Dad.m_Left = n;
    else
        m_Dad.m_Right = n;
    m_Node = n;
}
}
```

der aktuelle Knoten

Einfügen eines neuen Knotens ...

... wenn die Wurzel leer war, an der Wurzel

... sonst rechts oder links unterhalb des Vaters

Einfügen in binären Bäumen mit NodeHandler

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

h merkt sich aktuellen Knoten und Vorgänger

Schlüssel bereits vorhanden

Abstieg

NodeHandler weiß selber, wo der neue Knoten einzufügen ist

Suchen in binären Bäumen: Komplexität

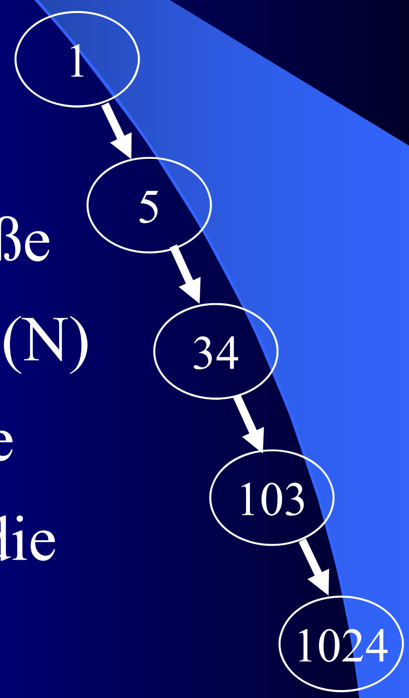
- Das Einfügen
  - dauert maximal bis zu der Tiefe des Baums
- Das Suchen
  - dauert maximal bis zu der Tiefe des Baums
- Die Tiefe des Baums ist minimal Logarithmus der Knotenanzahl, d.h. im Durchschnitt ist das Suchen und Einfügen in O(log N)

Suchen in binären Bäumen: Komplexität (Fort.)

- Vorsicht vor entarteten binären Bäumen
- Situation: die Schlüssel

  1 5 34 103 1024

  werden eingegeben
- Baum:
  - dieser Baum hat eine Tiefe linear zur Größe
  - damit liegt das Suchen und Einfügen in O(N)
  - dies gilt auch für die inverse Eingabefolge
  - binäre Bäume funktionieren nicht, wenn die Eingaben nicht möglichst gleichverteilt ankommen



Löschen in binären Bäumen

Alternative 1:

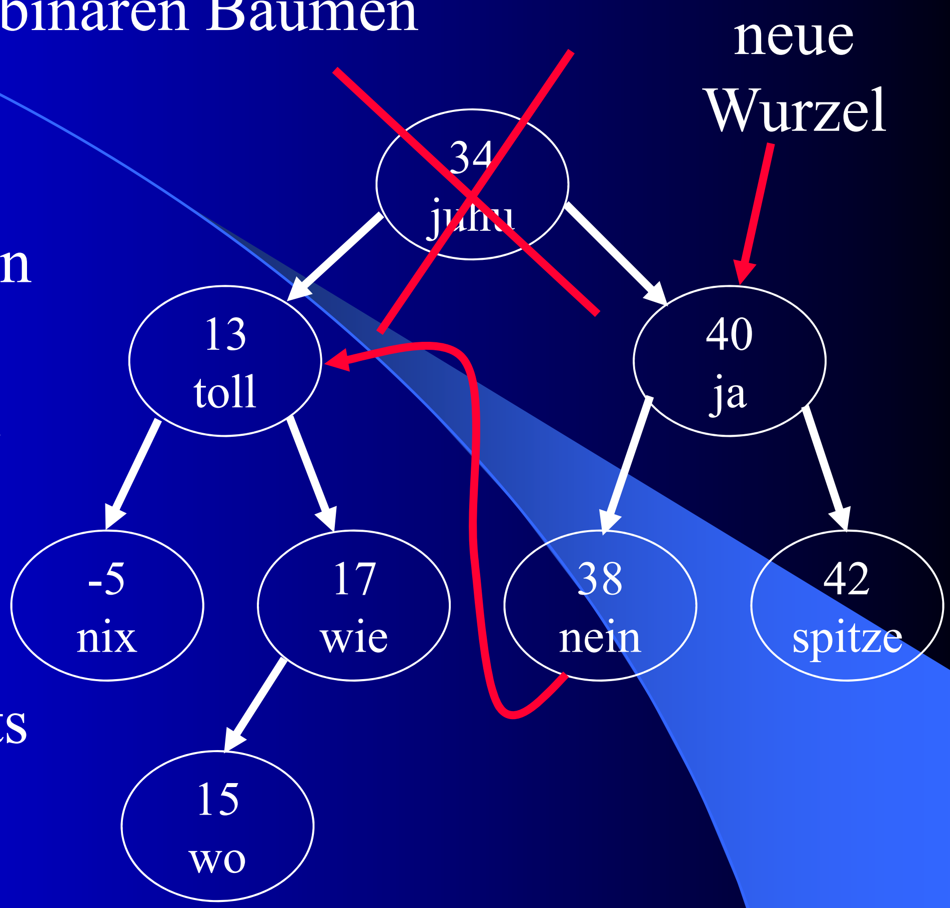
- ersetze den zu löschenden Knoten durch den rechten Nachfolger
- hänge linken Teilbaum unter den kleinsten Knoten des rechten Teilbaums
- um diesen Knoten zu finden:
  - gehe einen Schritt nach rechts
  - und dann immer links halten
- Bsp.:

  34 durch 40 ersetzen

  13 durch 17 ersetzen

  40 durch 42 ersetzen

  38 direkt löschen



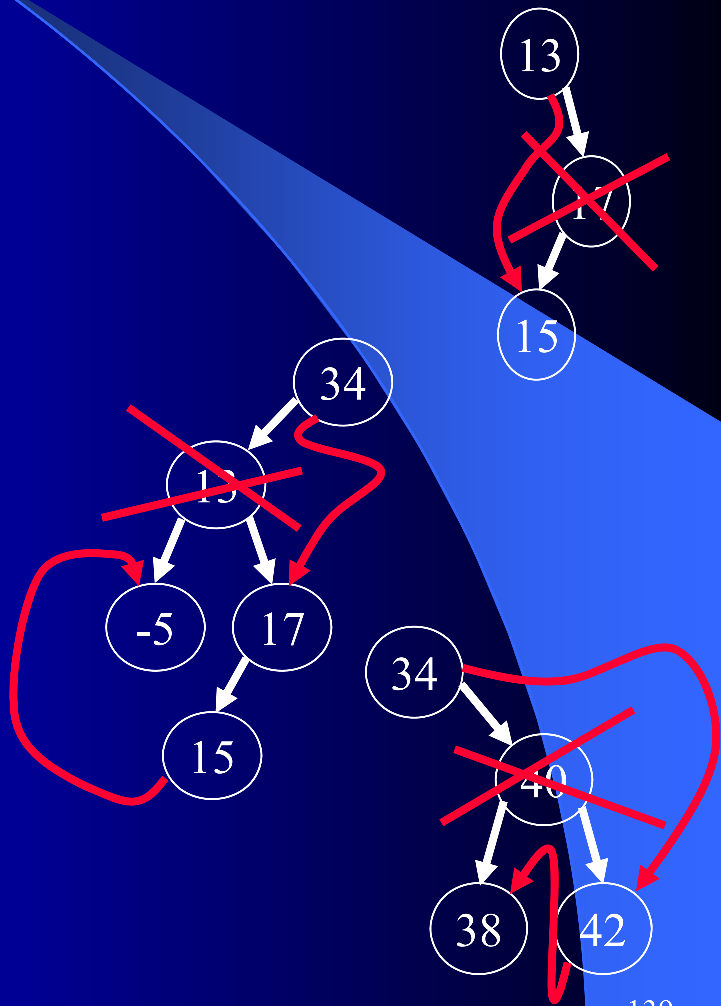
Löschen in binären Bäumen (Fort.)

Es gibt 3 Situationen:

1. der zu löschende Schlüssel ist nicht vorhanden

2. der zu löschende Knoten hat keinen rechten Nachfolger (dann ersetze ihn durch den linken Nachfolger)

3. der zu löschende Knoten hat einen rechten Nachfolger; dann gehe solange nach links, bis ein Knoten keinen linken Nachfolger mehr hat; dies kann auch schon der rechte Knoten sein



Löschen in binären Bäumen: Implementierung (Alternative 1)

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

gefunden ...

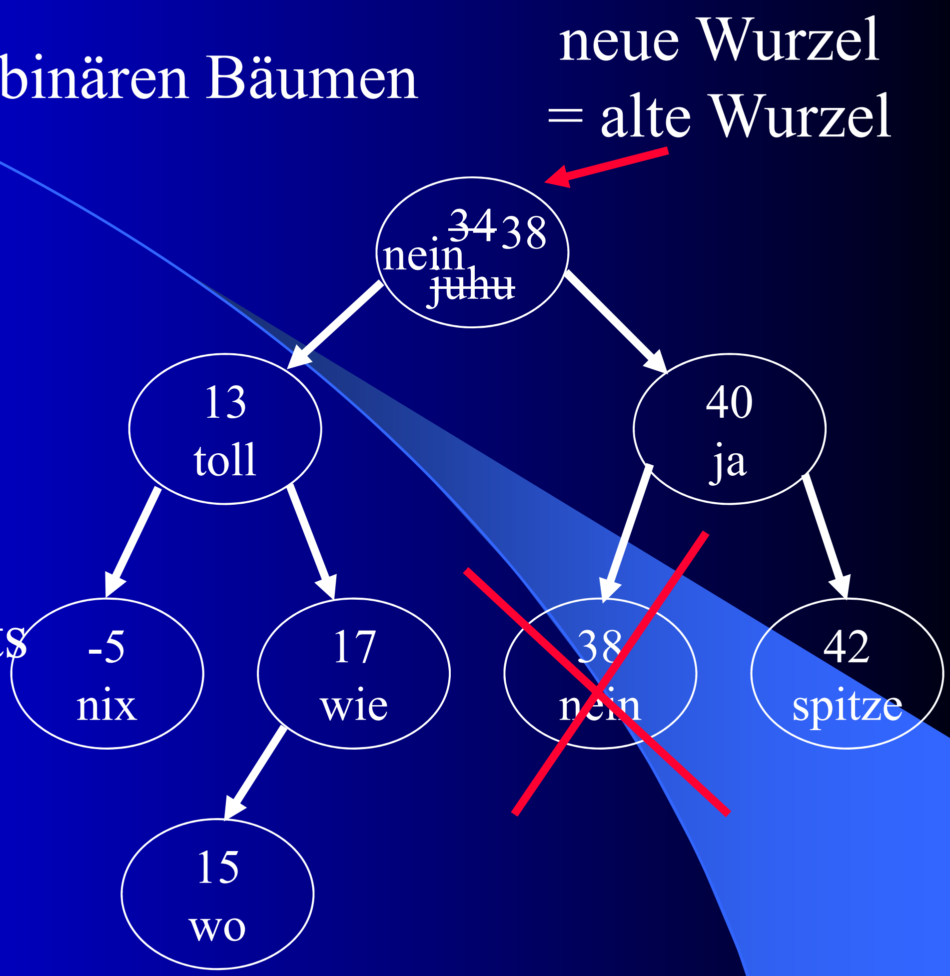
... gibt kein rechten Nachfolger

es gibt einen rechte Nachfolger; suche das kleinste Element in dem rechten Teilbaum

Abstieg

nicht gefunden

Löschen in binären Bäumen



Alternative 2:

- ersetze den Inhalt des zu löschenden Knoten durch den nächstgrößeren Inhalt
- um diesen Inhalt zu finden:
    - gehe einen Schritt nach rechts
    - und dann immer links halten
- Bsp.:

  34 durch 38 ersetzen

  13 durch 15 ersetzen

  40 durch 42 ersetzen

  38 direkt löschen

Löschen in binären Bäumen: Implementierung (Alternative 2)

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

gefunden ...

... gibt kein rechten Nachfolger

finde nächstgrößeres Element

überschreibe zu löschendes Element mit nächstgrößerem Element

Abstieg

nicht gefunden

Vorlesung 7

Hashing

- sehr gutes Verfahren für Suchen und Finden
- ist ein Kompromiss zwischen Speicherplatzverbrauch und Laufzeit
- relativ einfach zu implementieren
- wird sehr häufig verwendet
- Idee:
  - berechne zu dem zu suchenden Schlüssel einen Index
  - speichere unter diesem Index den Schlüssel mit Datensatz ab
  - dadurch erreicht man einen Zugriff in konstanter Zeit

Hashing (Fort.)

- die grundlegende Datenstruktur ist somit ein Array, deren einzelne Zellen über einen Index angesprochen werden können
- gesucht ist eine Funktion, die einem Schlüssel einen Index zuordnet
- Bsp.:
  - wenn der Schlüssel eine ganze Zahl ist, muss diese Zahl nur auf den Grenzbereich des Arrays abgebildet werden
  - Lösung: die Modulo Operation

Hashing: Illustration

- Suchen des Schlüssels 18

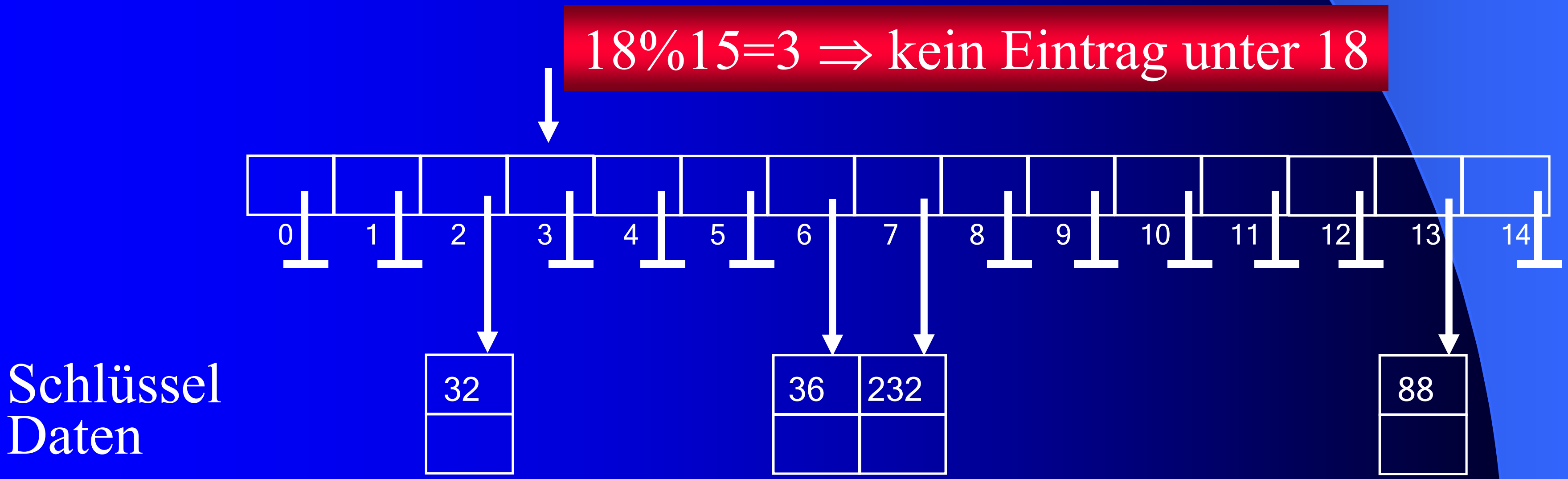
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Schlüssel			32				36	232						88	
Daten															

hier wird gesucht

- 18 % 15 = 3
- Zugriff unter Position 3

Hashing: Illustration (Fort.)

- Problem: wie unterscheidet man zwischen leeren und nicht-leeren Einträgen
- Lösung:
  - ein Eintrag ist ein Pointer zu einem Datensatz
  - ein Null-Pointer zeigt einen leeren Eintrag an



Hashing: 1. Implementierung

```
public class Hashing<D> {

    class Node<D> {
        public Node(int key,D data) {
            m_Key = key;
            m_Data = data;
        }

        private int m_Key;
        private D m_Data;
    }

    public Hashing() {
        m_Entries = new Node[1023];
        m_iNrOfEntries = 0;
    }

    private Node<D>[] m_Entries;
    private int m_iNrOfEntries;

...
}
```

Schlüssel/Daten Paar

zunächst sind alle Einträge 0

Array von Schlüssel/Daten Paaren

Hashing: Suchen

```
public class Hashing<D> {
...

    public D search(int key) {
        final int INDEX = key % m_Entries.length;
        if (m_Entries[INDEX] != null && m_Entries[INDEX].key == key)
            return m_Entries[INDEX].m_Data;
        else
            return null;
    }

...
}
```

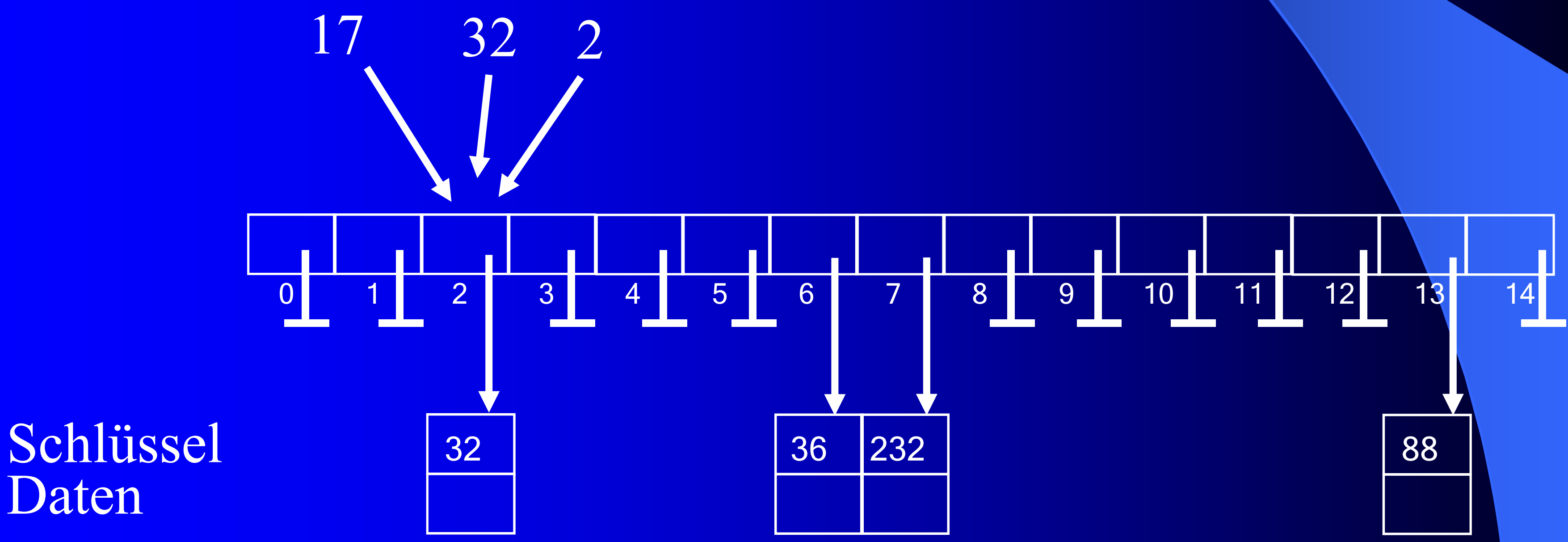
setzt voraus, dass key einen Modulo-Operator hat

dies gilt für int-Werte, aber ... (siehe Ende)

wenn es einen Eintrag gibt, dann wird der Datensatz zurückgeliefert

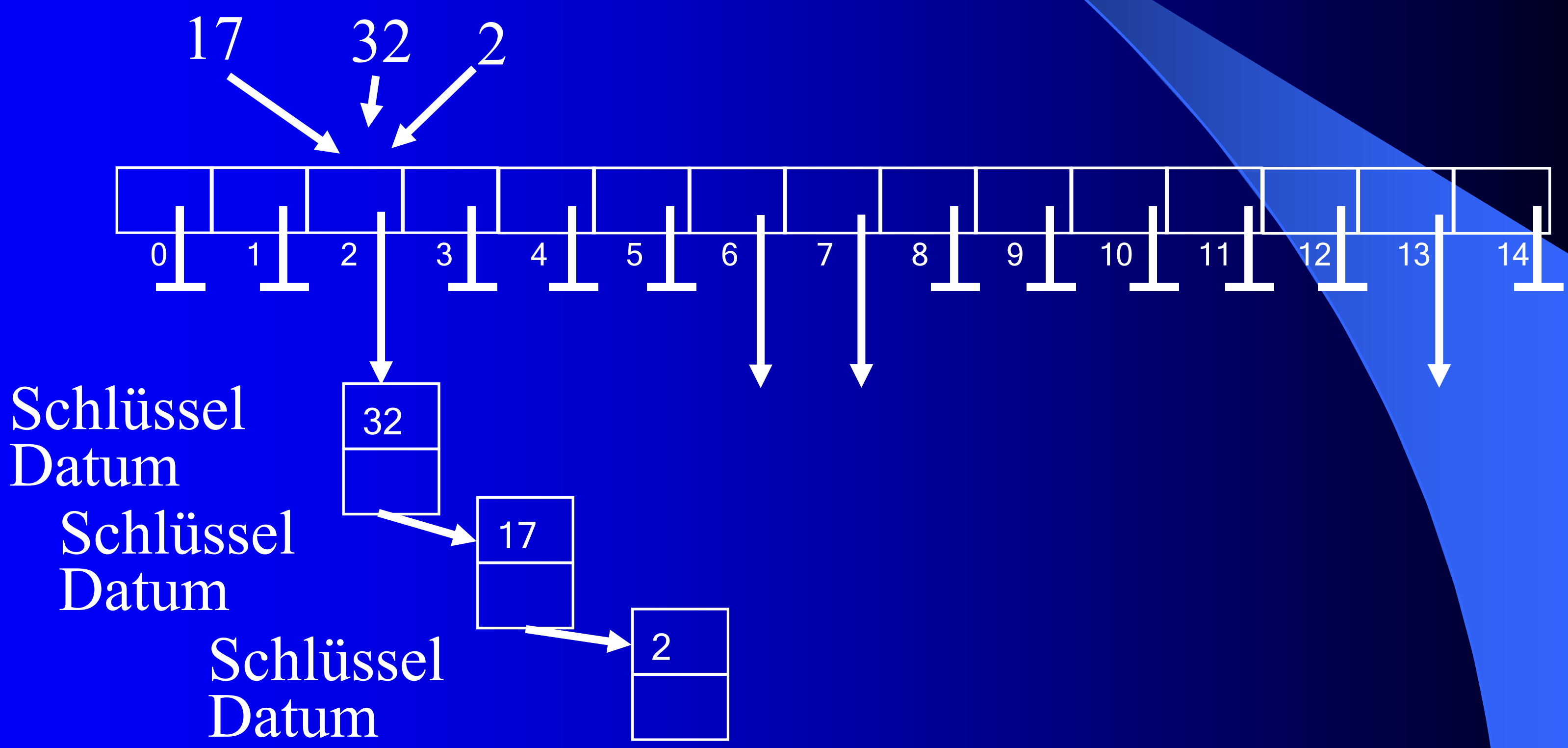
Hashing: Problem

- durch die Modulo Operation werden unterschiedliche Schlüssel auf den gleichen Index abgebildet
- Bsp.: 17, 32 und 2 werden alle auf die 2 abgebildet
- in einem solchen Fall spricht man von einer Kollision



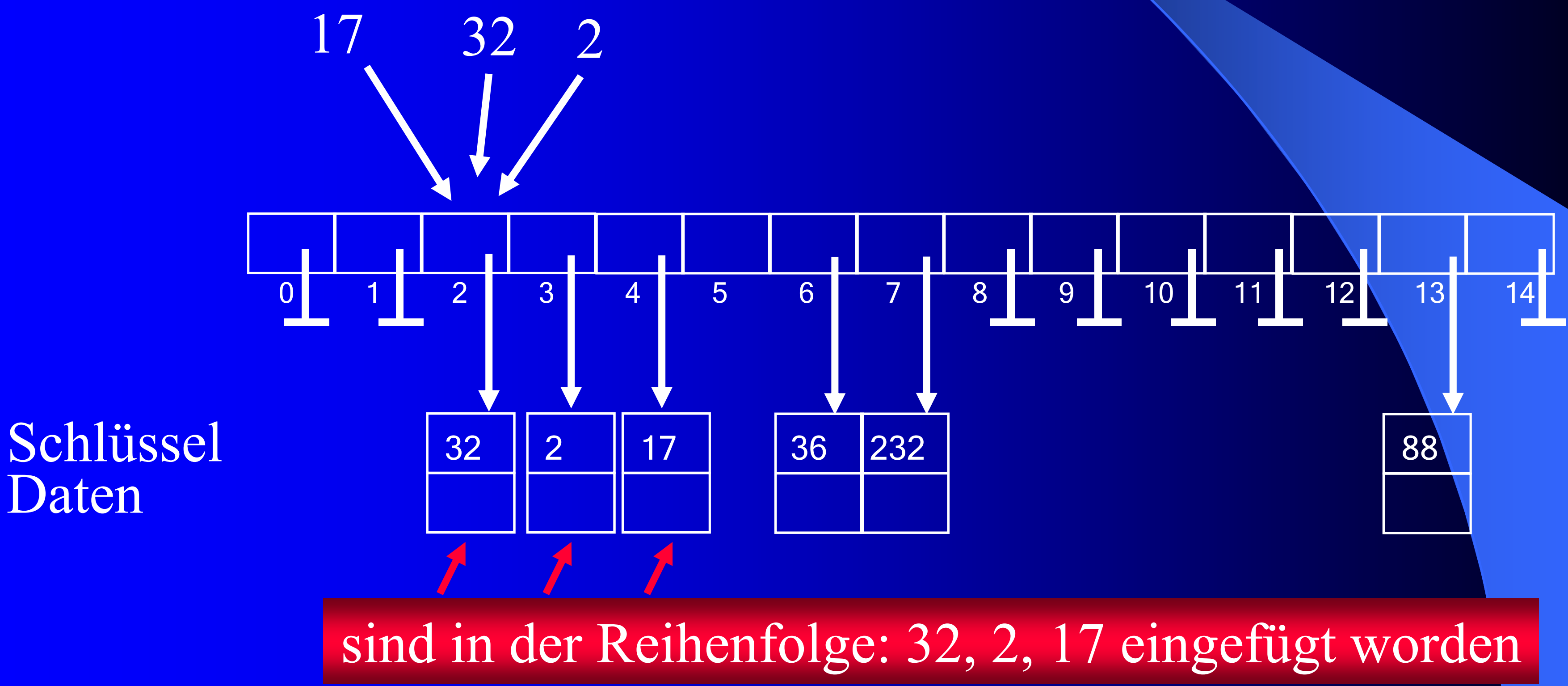
Hashing: Kollisionsbehebung

- Kollisionen können auf unterschiedliche Arten behoben werden
- zum einen können unter einem Index eine Liste von Einträgen verwaltet werden



Hashing: Kollisionsbehebung (Fort.)

- zum anderen kann ab dem berechnetem Index eine sequentielle Suche stattfinden
- am Ende muss wieder am Anfang begonnen werden



Hashing: Suchen (2. Versuch)

```
public class Hashing<D> {

...

    public D search(int key) {
        int iIndex = key % m_Entries.length;
        for(int i = 0; m_Entries[iIndex] != null && i < m_Entries.length; ++i) {
            if (m_Entries[iIndex].m_Key==key)
                return m_Entries[iIndex].m_Data;
            iIndex = (iIndex + 1) % m_Entries.length;
        }
        return null;
    }

...
}
```

iIndex ist nur der Start-index, ab dem gesucht wird

durchsuche maximal das gesamte Array

gibt es einen Eintrag und hat der den richtigen Schlüssel?

wenn nicht, such an der nächsten Stelle weiter; springe am Ende zum Anfang

Hashing: Einfügen

```
public class Hashing<D> {
...

    public void insert(int key, D data) {
        int iIndex = key % m_Entries.length;
        for(int i = 0; i < m_Entries.length; ++i) {
            if (m_Entries[iIndex] == null) {
                m_Entries[iIndex] = new Node<D>(key,data);
                ++m_iNrOfEntries;
                return;
            }
            iIndex = (iIndex + 1) % m_Entries.length;
        }
    }

...
}
```

iIndex ist nur der Start-index, ab dem gesucht wird

durchsuche maximal das gesamte Array

ist der Eintrag frei, ... ?

... dann füge einen neuen ein

wenn nicht, such an der nächsten Stelle weiter; springe am Ende zum Anfang

Hashing: Einfügen (Fort.)

- das Einfügen funktioniert nur, wenn es noch mindestens einen freien Platz gibt
- je weniger freie Plätze es noch gibt, desto größer ist die Wahrscheinlichkeit, dass das gesamte Array durchsucht werden muss
- also muss zur richtigen Zeit das Array vergrößert werden
- guter Wert ist, wenn das Array zu 80% voll ist

Frage: was sind gute Arraygrößen?

Hashing: Einfügen (Verfeinert)

```
public void insert(int key, D data) {
    int iIndex = key % m_Entries.length;
    for(int i = 0;i < m_Entries.length;++i) {
        if (m_Entries[iIndex] == null) {
            m_Entries[iIndex] = new Node<D>(key,data);
            ++m_iNrOfEntries;
            if (m_iNrOfEntries > m_Entries.length *8/10)
                resize();
            return;
        }
        iIndex = (iIndex + 1) % m_Entries.length;
    }
}
```

führe eine Vergrößerung durch, wenn 80% gefüllt sind

Hashing: Resize

das alte Array wird verdoppelt

```
private void resize() {
    final int OLDCAPACITY = m_Entries.length;
    Node<D>[] oldEntries = m_Entries;
    final int iNewCap = (m_Entries.length + 1) * 2 - 1;
    m_Entries = new Node[iNewCap];
    m_iNrOfEntries = 0;
    for(int i = 0;i < OLDCAPACITY;++i) {
        if (oldEntries[i] != null) {
            insert(oldEntries[i].m_Key, oldEntries[i].m_Data);
        }
    }
}
```

neues Array anlegen

ein alter Eintrag wird mittels der insert Methode eingefügt

Der Algorithmus kann optimiert werden, indem die Knoten direkt umgehängt werden

Hashing: Schlüssel

- Nachteil der bisherigen Implementierung ist, dass davon ausgegangen werden muss, dass der Schlüssel sich durch einen int teilen lassen muss
- dies ist für int und unsigned int ok
- für char* ist dies katastrophal, da zwei gleiche Strings, die an unterschiedlichen Stellen im Speicher stehen, unterschiedliche Pointer haben
- dadurch hätten diese beiden gleichen Strings unterschiedliche Startindizies $\Rightarrow$ man würde einen zuvor eingetragenen String nicht finden

C++

Hashing: Schlüssel (Fort.)

- Trennung der Berechnung des Index aus dem Schlüssel

```
public D search(Object key) {
    int iIndex = hashKey(key, m_Entries.length);
    ...
};

public void insert(Object key, D data) {
    int iIndex = hashKey(key, m_Entries.length);
    ...
}
```

Java

Hashing: Schlüssel (Fort.)

- Hashkeys für Character und Integer

```
private int hashKey(Object key,int iLength) {
    if (key instanceof Integer) {
        Integer i = (Integer)key;
        if (i.intValue() < 0)
            return -i.intValue() % iLength;
        else
            return i.intValue() % iLength;
    } else if (key instanceof Character) {
        Character c = (Character)key;
        return c.charValue() % iLength;
    } else
        return 0;
}
```

hashkey funktioniert nur für Integer und Character

int werden in positive Zahlen verwandelt

Character werden als Zahlenwert interpretiert und direkt verwendet

C++

Hashing: Schlüssel (Fort.)

- verschiedene Hashkeys

unsigned int können direkt verwendet werden

```
unsigned hashKey(unsigned ui , unsigned uiLength) {
    return ui % uiLength;
}
```

int werden in positive Zahlen verwandelt

```
unsigned hashKey(int i , unsigned uiLength) {
    return (unsigned)i % uiLength;
}
```

```
template<class K>
unsigned hashKey(K* p , unsigned uiLength) {
    return (unsigned)(p >> 2) % uiLength;
}
```

bei allgemeinen Pointern (z.B. Adressen von Objekten) werden die beiden unteren Bits weggeschnitten, da sie in einer 32-Bit Architektur immer 0 sind

Was ist in einer 64-Bit Architektur zu tun?

Hashing: Schlüssel (Fort.)

- für Strings möchte man einen Schlüssel aus der Buchstabenfolge berechnen
- wichtig: ähnliche Worte sollen an ganz unterschiedlichen Stellen in der Hashtabelle gespeichert werden, um lokale Häufungen zu vermeiden

C++

```
unsigned hashKey(const char* cpStr,
                               unsigned uiLength) {
      unsigned res;
      for(res = 0; *cpStr != '\0'; ++cpStr)
            res = ((res << 6) + *cpStr) % uiLength;
      return res;
}
```

Java

```
private int hashKey(Object key,int iLength) {
      ...
      } else if (key instanceof String){
            String str = (String)key;
            int res = 0;
            for(int i = 0; i < str.length(); ++i)
                  res = ((res << 6) + str.charAt(i)) % iLength;
            return res;
      } else
            return 0;
}
```

vordefinierte Hashimplementierungen

- in Java gibt es die Klasse HashMap<K,D>
- in C++ gibt es std::hash_map<K,D>

HashMap<K,D> in Java

- die Hashmap in Java verwendet die hashCode Methode der Object Klasse des Schlüssels K
- hierbei sind folgende Regeln zu beachten:
  1. während eines Programmablaufs muss hashCode für ein gegebenes Objekt immer den gleichen Wert zurückliefern
  2. sind zwei Objekte gemäß der equals Methode identisch, so muss hashCode für diese beiden Objekte den gleichen Wert zurückliefern
  3. es ist nicht notwendig, dass zwei Objekte, die nicht gleich gemäß equals sind, unterschiedliche hashCode Ergebnisse liefern

HashMap<K,D> in Java (Forts.)

- für equals gelten folgende Regeln:
  1. reflexiv: x.equals(x) = true
  2. symmetrisch: x.equals(y) == y.equals(x)
  3. transitiv: x.equals(y) $\wedge$ y.equals(z) $\Rightarrow$ x.equals(z)
  4. konsistent: x.equals(y) ist immer true oder immer false
  5. x.equals(null) == false

Vorlesung 8

Approximation: 1-dimensional

- Die binäre Suche kann auch zur Approximation dienen
- Aufgabe:
  - gegeben eine Menge M von Zahlen
  - gegeben eine Zahl x
  - finde $y \in M$ mit $\forall z \in M: |x-y| \leq |x-z| \vee y = z$
- Beispiel:

  M = {1, 4, 17, 23, -34, -2003, 1024, 6, 7}

  x = 9

  dann ist y = 7 die Zahl, die zu allen anderen Zahlen den kleinsten Abstand hat

Approximation: 1-dimensional (Forts.)

- Lösung für dieses Problem:
  - alle Zahlen aus M werden zunächst in einem Vektor v sortiert
  - mit der binären Suche sucht man die Zahl x
  - ist x in der Menge vorhanden → fertig
  - ist x nicht in der Menge vorhanden, dann
    1. hört die Suche bei einem Index i auf, an dem x gestanden hätte, wenn es in M vorhanden gewesen wäre
    2. wenn x < v[i] ist, dann vergleiche x mit v[i] und v[i-1]
    3. wenn x > v[i] ist, dann vergleiche x mit v[i] und v[i+1]

Approximation: 1-dimensional (Beispiel)

M = {1, 4, 17, 23, -34, -2003, 1024, 6, 7}

x = 9

v =

-2003	-34	1	4	6	7	17	23	1024
0	1	2	3	4	5	6	7	8

- Ergebnis der binären Suche: i = 5
- da v[5] = 7 < x = 9 ist, wird x zusätzlich mit v[6] verglichen
- Ergebnis des Vergleichs: v[5] = 7 hat den kleinsten Abstand zu 9 von allen Zahlen aus M

Approximation: 2-dimensional

- Frage:

  kann dieses Verfahren auch auf eine mehrdimensionale Approximation angewendet werden

- Beispiel:

  gegeben ist eine Menge von Punkten M in einem Koordinatensystem; gesucht ist der Punkt aus M, der von einem gegebenen Punkt x den kleinsten Abstand hat

- Antwort:

  leider nein, da die Elemente nicht mehr linear angeordnet werden können, d.h. für Zahlen gilt:

$$x \nless y \wedge x \ngtr y \Rightarrow x = y$$

dies gilt **<u>nicht</u>** für Punkte

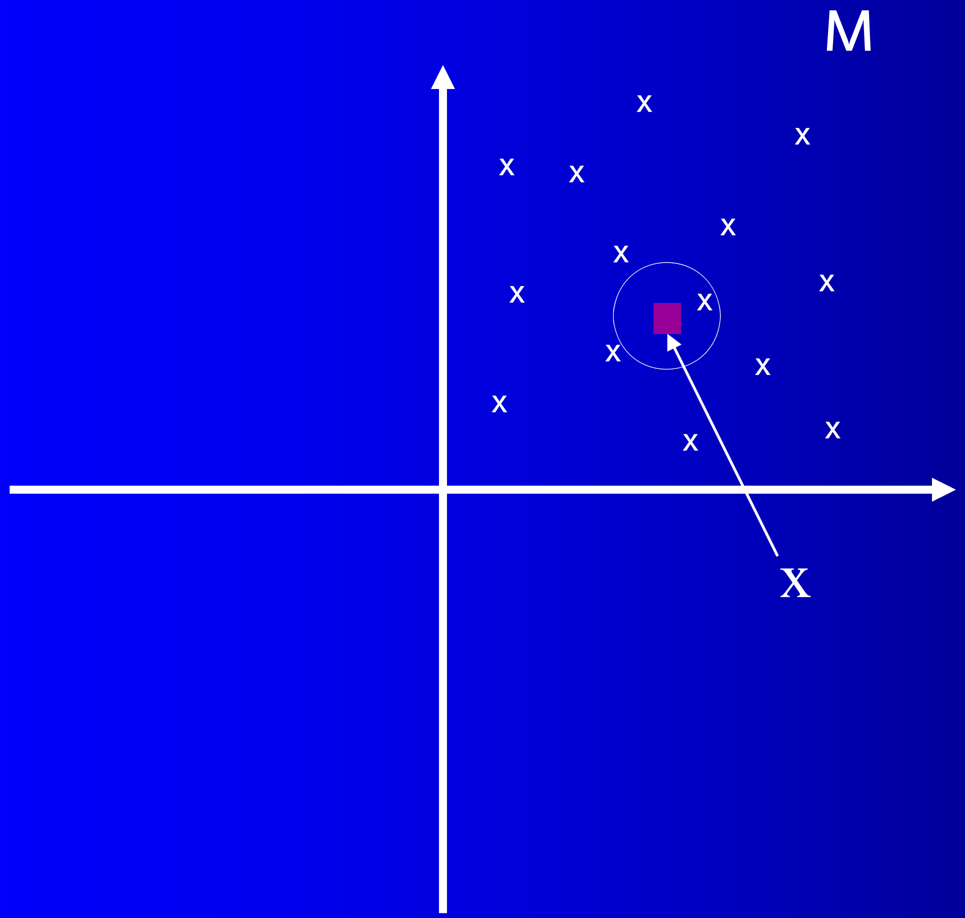
$$(1,2) \nless (2,1) \wedge (1,2) \ngtr (2,1) \not\Rightarrow (1,2) = (2,1)$$

Approximation: 2-dimensional (Forts.)

- Brute Force Ansatz:
  - Berechne die Distanz von x zu allen Elementen aus M
  - selektiere das Element mit dem kleinsten Abstand
  - Komplexität: $O(n)$
  - Zum Vergleich binärer Suche: $O(\log n)$

Approximation: 2-dimensional (Forts.)

- Visualisierung des Problems:

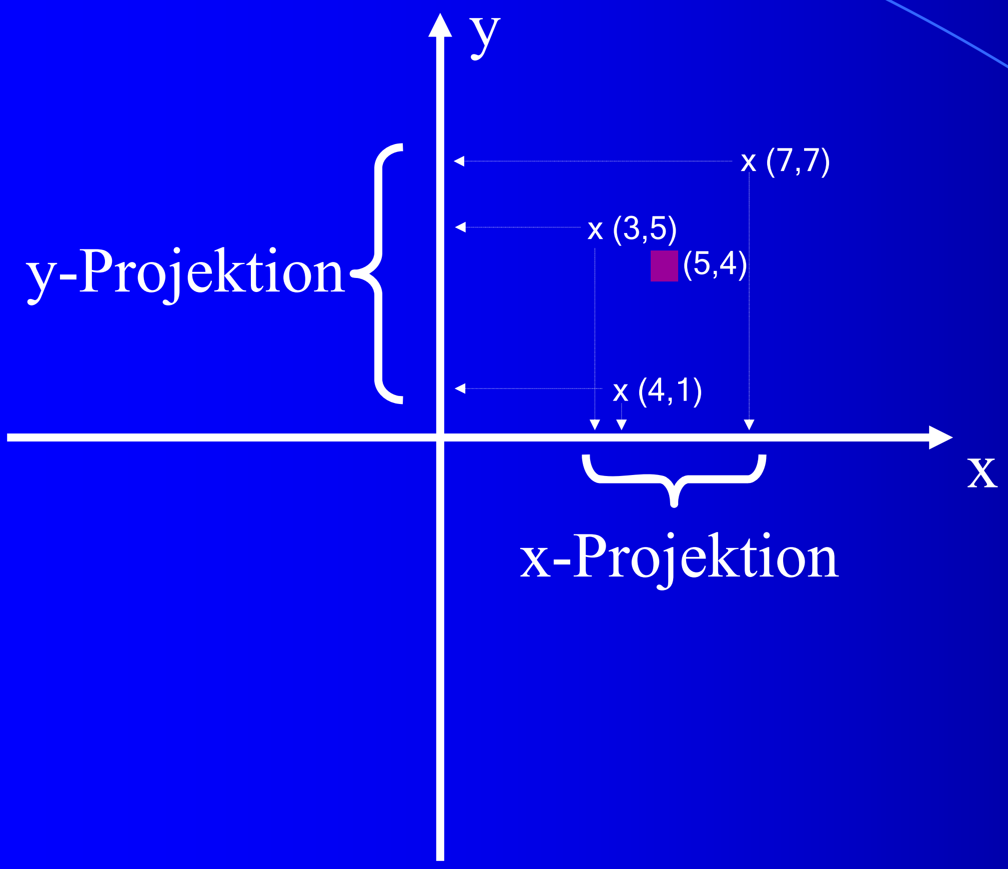


- Idee: um **x** konzentrische Kreise ziehen
- prüfen, ob ein Punkt aus **M** in diesem Kreis liegt
- wenn nicht, wird der Kreis vergrößert

Approximation: 2-dimensional (Forts.)

- Probleme:
  - Wie sollen konzentrische Kreise gezogen werden?
  - Wie wird effizient geprüft, ob ein Punkt im Kreis liegt?
- Idee:
  - sortiere die Punkte einmal nach der x-Koordinate
  - sortiere die Punkte einmal nach der y-Koordinate
  - suche mittels der binären Suche in beiden Vektoren
  - starte von den gefundenen Punkten die Suche und verkleinere sukzessiv den Radius des Kreises

Approximation: 2-dimensional: Beispiel



M = {(4,1) , (7,7), (3,5)}

x = (5,4)

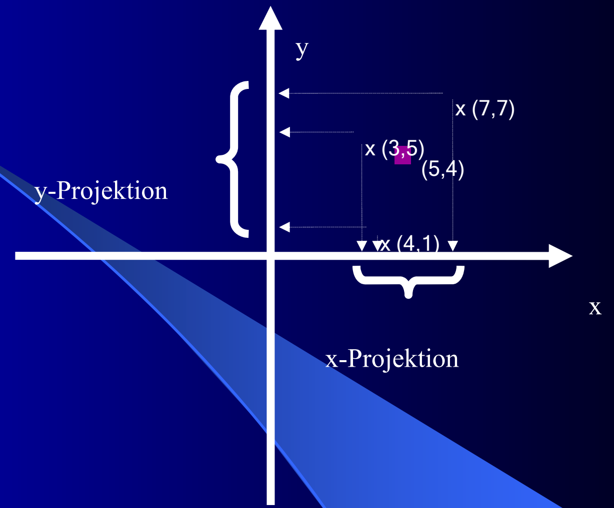
v_x =

(3,5)	(4,1)	(7,7)
0	1	2

v_y =

(4,1)	(3,5)	(7,7)
0	1	2

Approximation: 2-dimensional: Beispiel (Forts.)



- Suchen von (5,4)

Binärsuche bzgl. 5 (x-Wert) endet hier

v_x =

(3,5)	(4,1)	(7,7)
0	1	2

v_y =

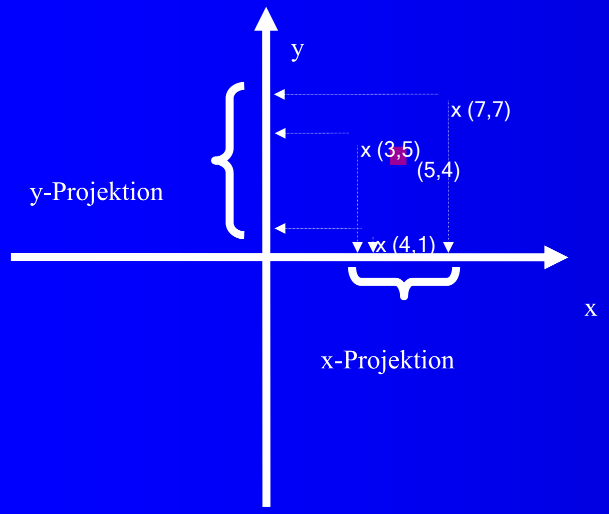
(4,1)	(3,5)	(7,7)
0	1	2

Binärsuche bzgl. 4 (y-Wert) endet hier

es sind 4 Vergleiche notwendig:

- 2 bzgl. der x-Projektion (5,4) mit (4,1) und (7,7)
- 2 bzgl. der y-Projektion (5,4) mit (4,1) und (3,5)

Approximation: 2-dimensional: Beispiel (Forts.)



bei den 4 Vergleichen werden die Abstände der Punkte zueinander berechnet (Satz des Pythagoras):

- $|(5,4) - (4,1)| = \sqrt{(1+9)} \approx 3{,}16$
- $|(5,4) - (7,7)| = \sqrt{(4+9)} \approx 3{,}60$
- $|(5,4) - (3,5)| = \sqrt{(4+1)} \approx 2{,}23$

die erste Vergleichsrunde hat ergeben, dass maximal in einem Abstand von 2,23 gesucht werden muss

$\Rightarrow$ es müssen maximal bzgl. der **x-Projektion** die Werte zwischen 5-2,23 = 2,77 und 5+2,23 = 7,23 betrachtet werden

$\Rightarrow$ es müssen maximal bzgl. der **y-Projektion** die Werte zwischen 4-2,23 = 1,77 und 4+2,23 = 6,23 betrachtet werden

Approximation: 2-dimensional (Forts.)

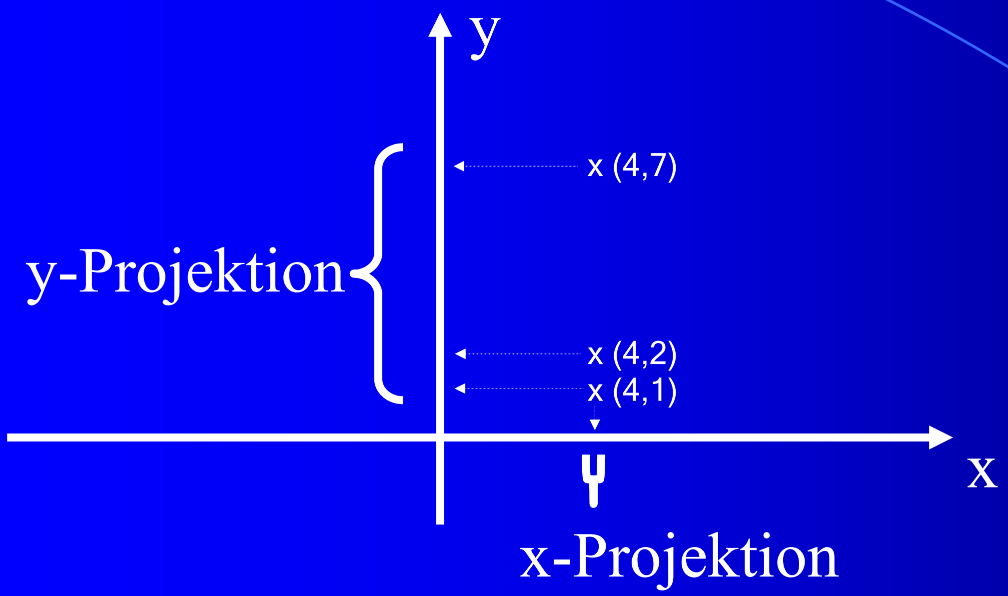
- bei den weiter zu untersuchenden Punkten werden die neuen Abstände mit dem alten Abstand verglichen
- ist der neue Abstand kleiner, so wird der Suchraum weiter eingeschränkt
- i.d.R. sollte das Verfahren schnell beendet werden

- offene Fragen:
  - Wie kann das Verfahren für mehr als 2 Dimensionen erweitert werden?
  - Wie sollen die Daten in verschiedenen Projektionen bei gleichen Werten sortiert (Bsp. (4,3), (4, 50), (4,16) bzgl. der x-Projektion)?
  - Was sind ungünstige Daten?

Approximation: mehr-dimensional

- das Verfahren kann kanonisch auf mehr als 2 Dimensionen erweitert werden
- neben der x- und y-Projektion müsste dann eine z-Projektion durchgeführt werden, wenn es sich um 3 Dimensionen handelt
- das Suchen würde dann nicht 4 sondern 6 Elemente ergeben, von denen dann die Abstände zu berechnen wären
- die Abstände würden wieder mittels des Satz des Pythagoras ermittelt werden
- in jedem Iterationsschritt würden 6 neue Elemente untersucht werden

Approximation: Verfeinerung der Projektion

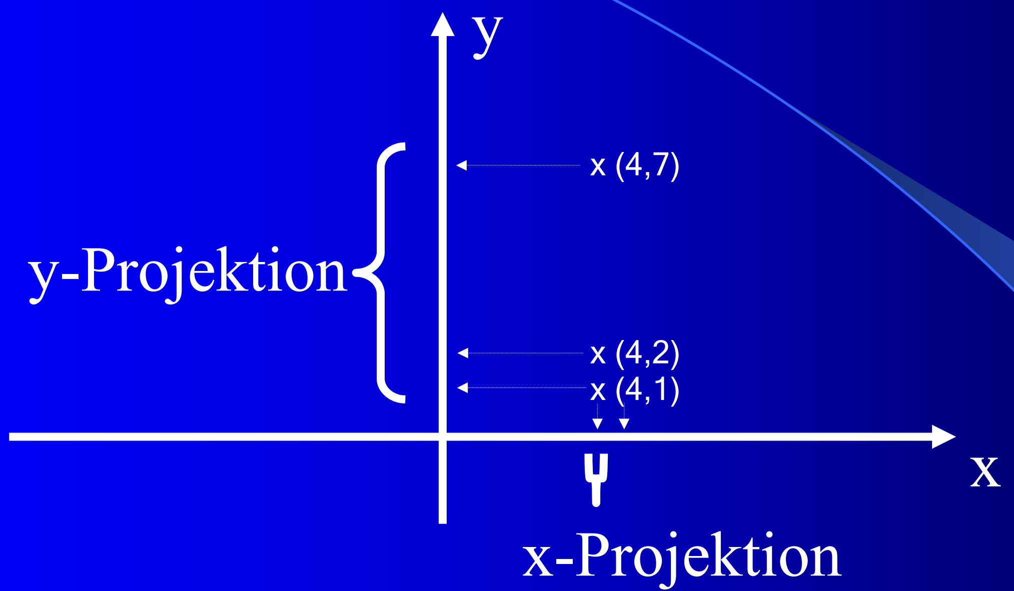


Problem:

- die y-Projektion verteilt die Punkte gut
- die x-Projektion bildet alle Punkte auf den selben Punkt ab

- dies führt dazu, dass ein binäres Suchen bzgl. der x-Projektion keinen Sinn mehr macht
- Optimierung: Elemente mit gleicher x-Projektion werden dann bzgl. ihres y-Wertes sortiert (bei gleichem y-Wert, bzgl. des z-Wertes usw.)

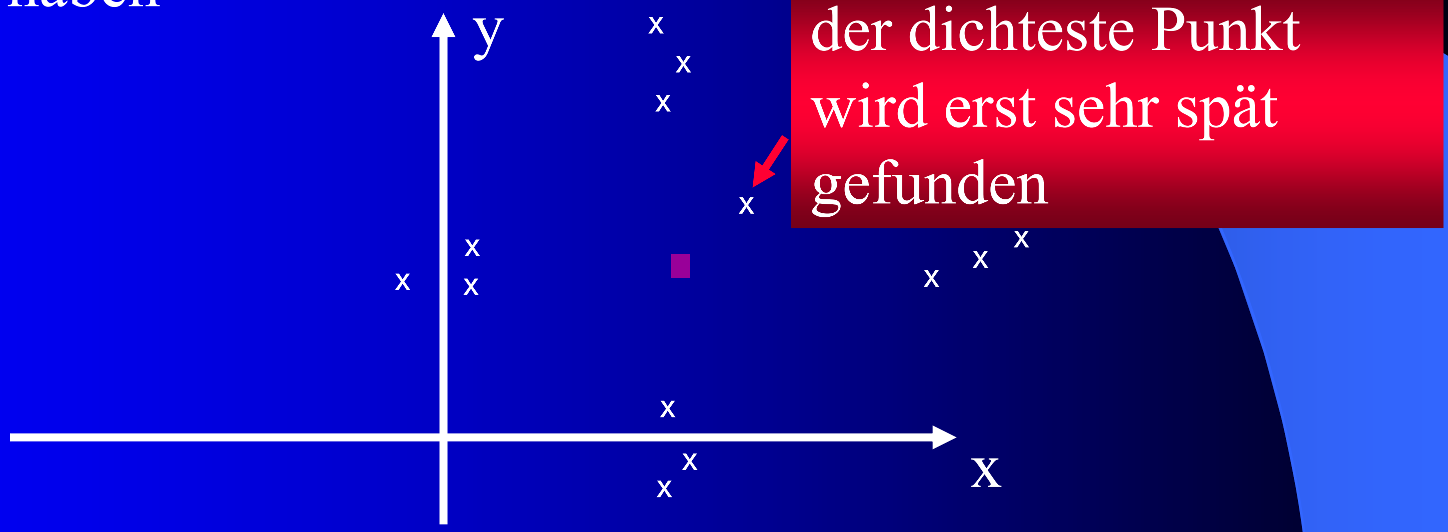
Approximation: Verfeinerung der Projektion (Forts.)



- x-Projektion ergibt dann **eindeutig** die Reihenfolge:
  (4,1) (4,2) (4,7)
- ein Suchen von (4,3) bzgl. der x-Projektion endet dann zwischen den Elementen (4,2) und (4,7)
- analog wird mit den anderen Projektionen verfahren

Approximation: ungünstige Daten

- Das Verfahren funktioniert gut,
  - wenn die Nähe der Punkte zueinander auch durch die Nähe der einzelnen Koordinatenanteile ausgedrückt wird
- Das Verfahren funktioniert schlecht,
  - wenn viele Punkte fast identische x-Werte (bzw. y-Werte) aber sehr weit auseinanderliegende y-Werte (bzw. x-Werte) haben



Anwendung der Approximation: Farbsubstitution

- Für die Übung 2 (Substitution von seltenen Farben in einem Bild durch häufig vorkommende Farbe) wird benötigt:
  - Sortierung der Farben (wichtig für das Zählen, welche Farben wie oft vorkommen)
  - Approximation der Farben (3-dimensionale Approximation)
- Frage: sind die Farbdaten geeignet?
- Hierzu soll die Farbverteilung (welche Farben kommen vor?) beispielhaft untersucht werden

Go!

Anwendung der Approximation: Farbsubstitution (Forts)

- Hierzu werden alle vorkommenden Farben nach ihrem Rot-, Grün- und Blauanteil in eine Datei geschrieben
- Diese Datei kann mittels des Programms `gnuplot` dargestellt werden
- Beispiel:

```
82 103 120
80 97 117
81 92 120
82 96 125
75 92 120
81 99 123
82 93 115
87 89 110
83 90 109
80 88 101
85 96 102
85 95 104
87 99 115
82 95 114
80 93 112
79 92 108
92 97 103
...
```

Go!

Anwendung der Approximation: Farbsubstitution (Forts)

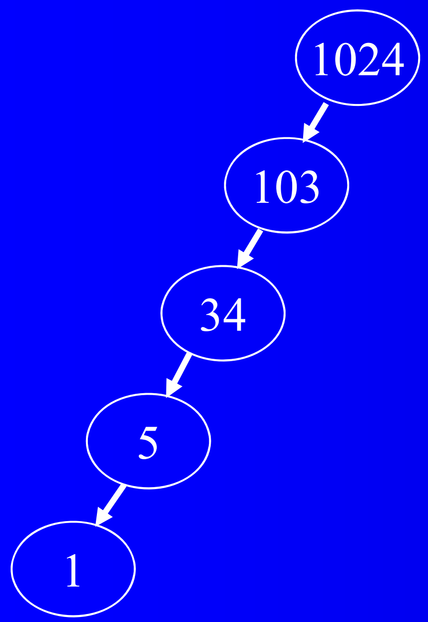
- Die Daten zeigen, dass das Verfahren geeignet ist, um die die Farben effizient zu approximieren.
- Somit kann dieses Verfahren für die Farbsubstitution eingesetzt werden.

Vorlesung 9

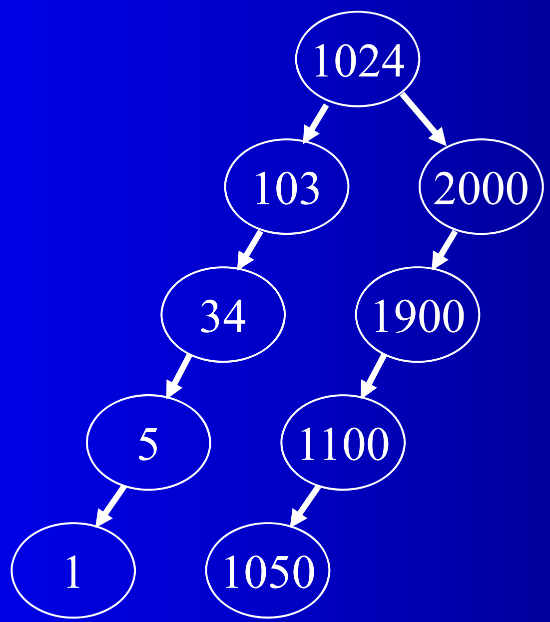
Nachteil von binären Bäumen

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

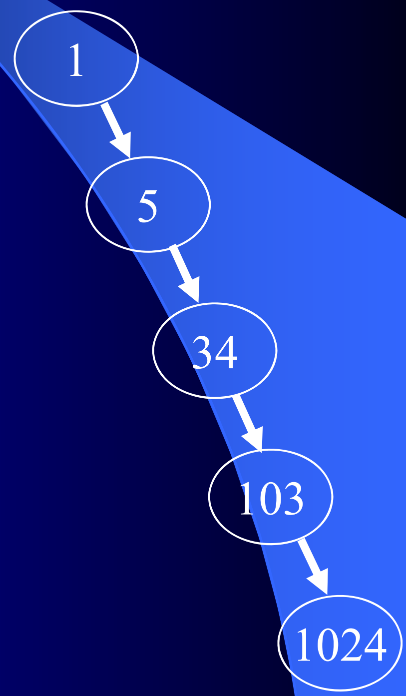
1024 103 34 5 1



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

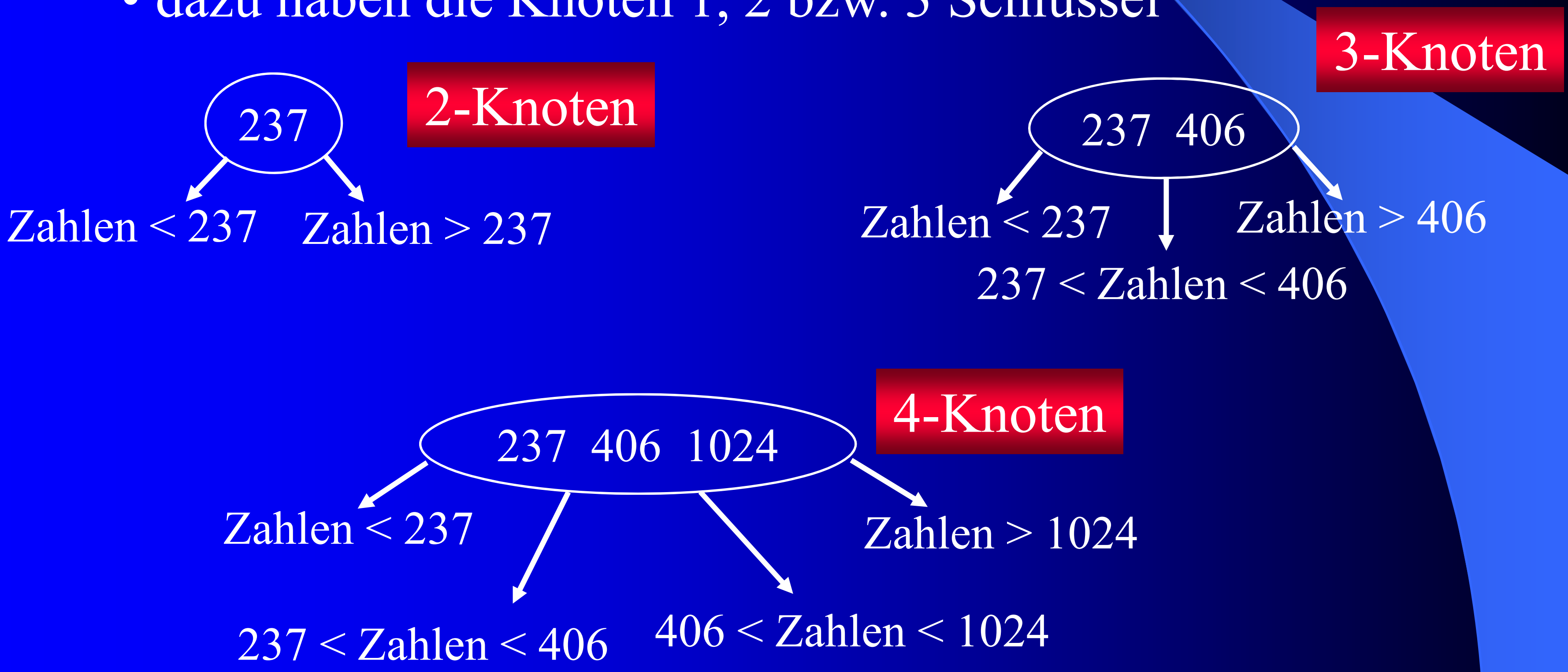
Idee:

- 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

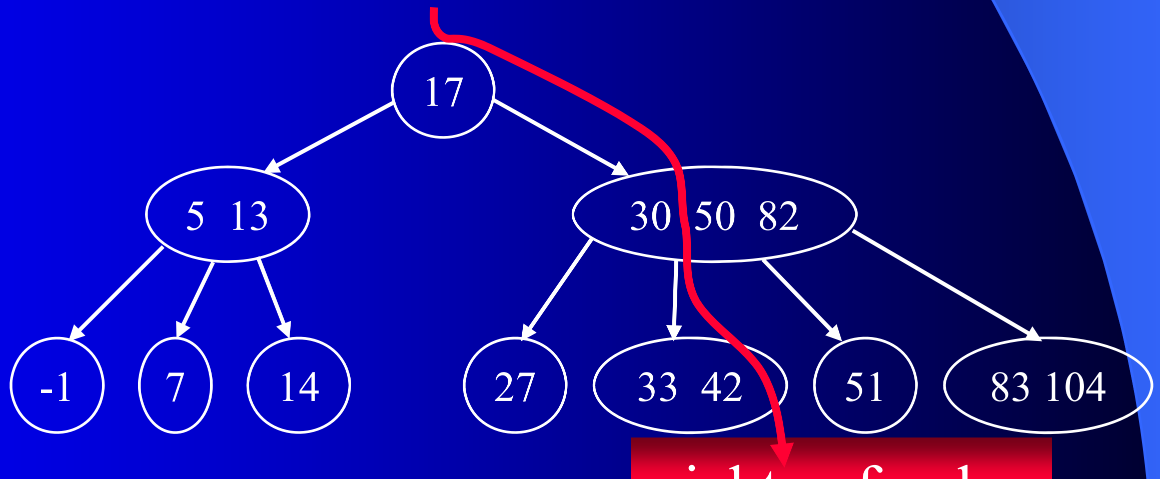


Top-Down 2-3-4-Bäume: Suchen

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

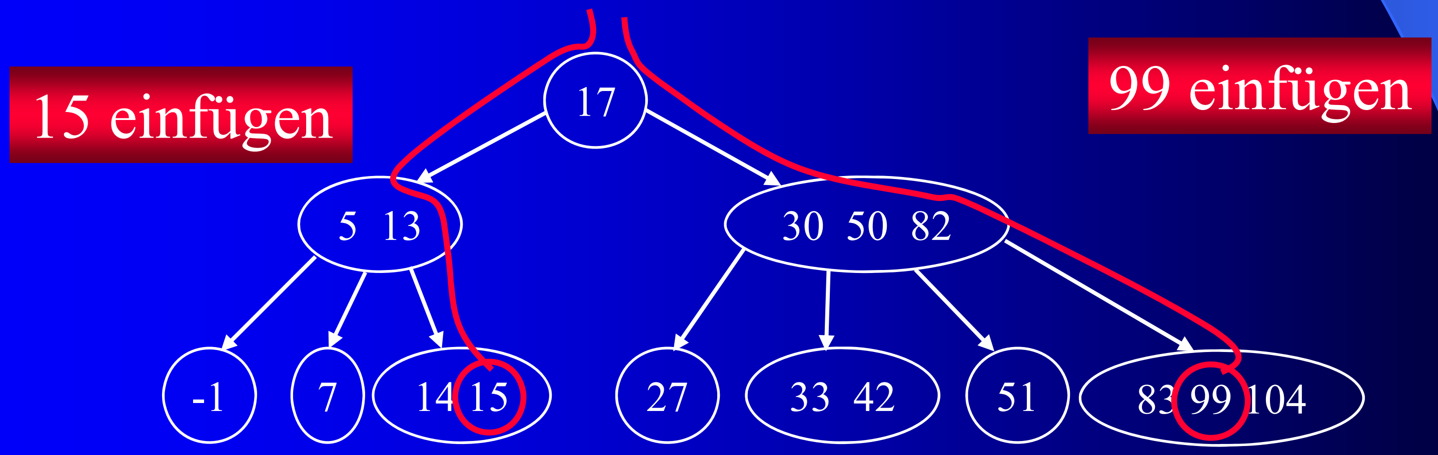


nicht gefunden

Top-Down 2-3-4-Bäume: Einfügen

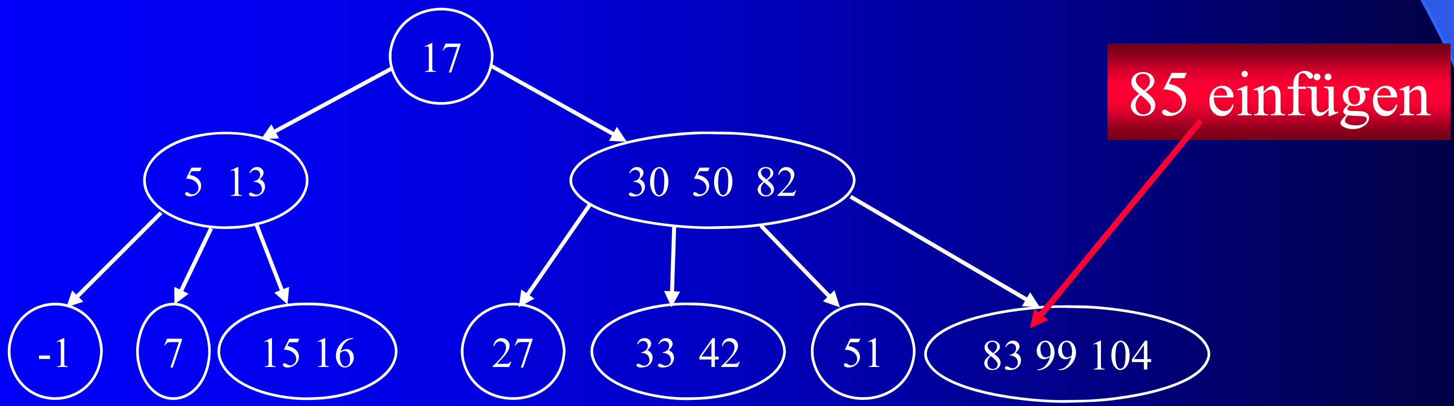
Idee:

- 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

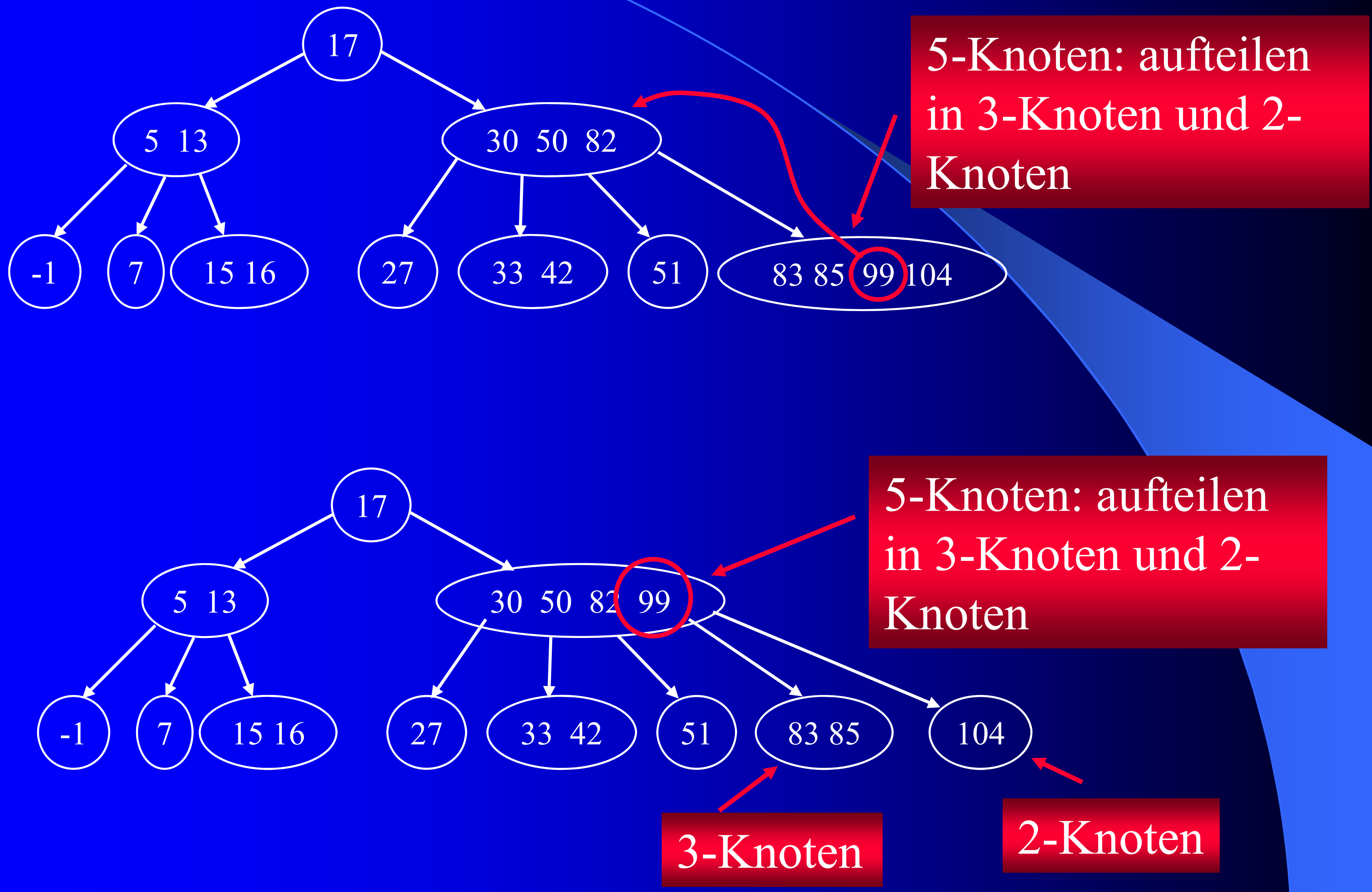


Top-Down 2-3-4-Bäume: Einfügen (Fort.)

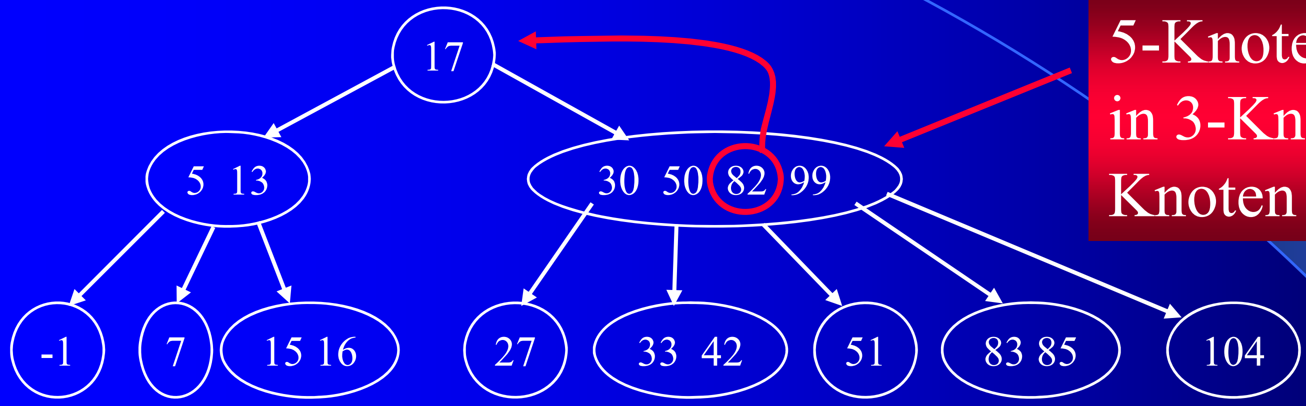
- muss in einem 4-Knoten Blatt eingefügt werden (es müsste ein 5-Knoten entstehen), so wird er in ein 3-Knoten und ein 2-Knoten aufgeteilt
- dadurch bekommt der Vater einen Knoten mehr
- dadurch muss der Vater (und rekursiv dessen Vater usw.) u.U. ebenfalls neu aufgeteilt werden



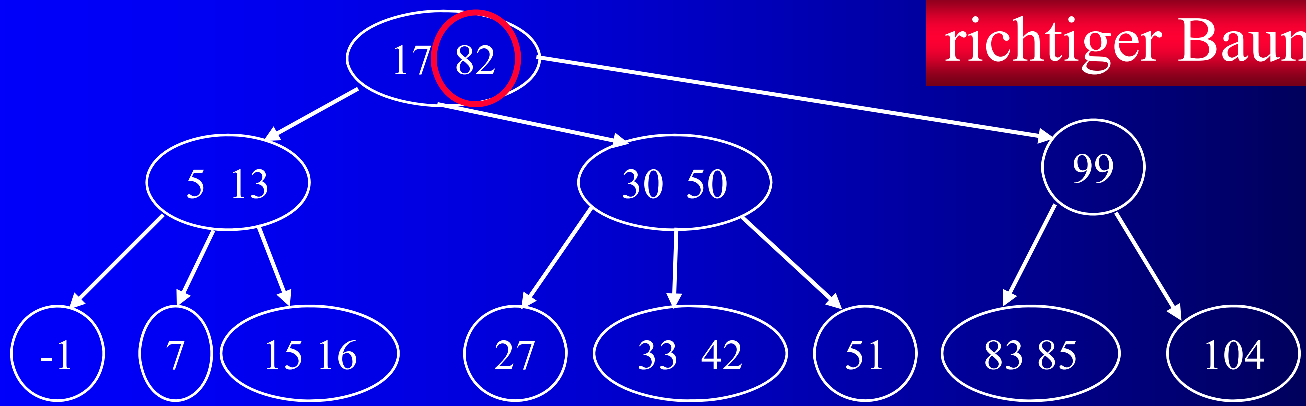
Top-Down 2-3-4-Bäume: Einfügen (Fort.)



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



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

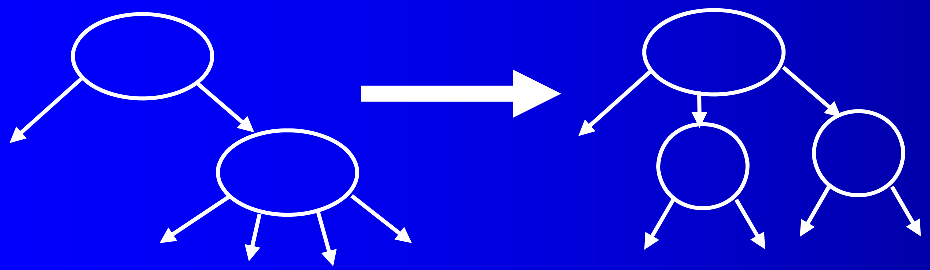


richtiger Baum

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

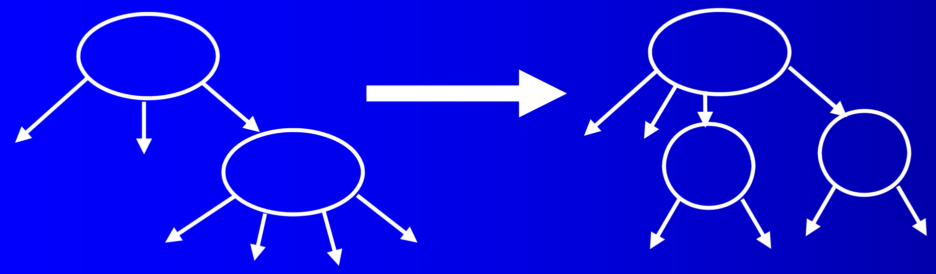
Optimierung:

- 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:



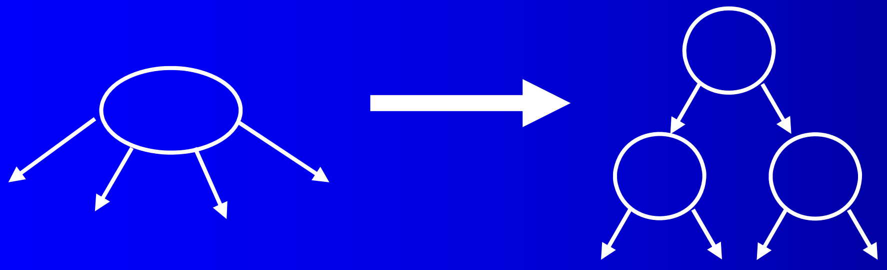
als einem 2-Knoten mit 4-Knoten Nachfolger wird ein 3-Knoten mit 2 2-Knoten Nachfolgern

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

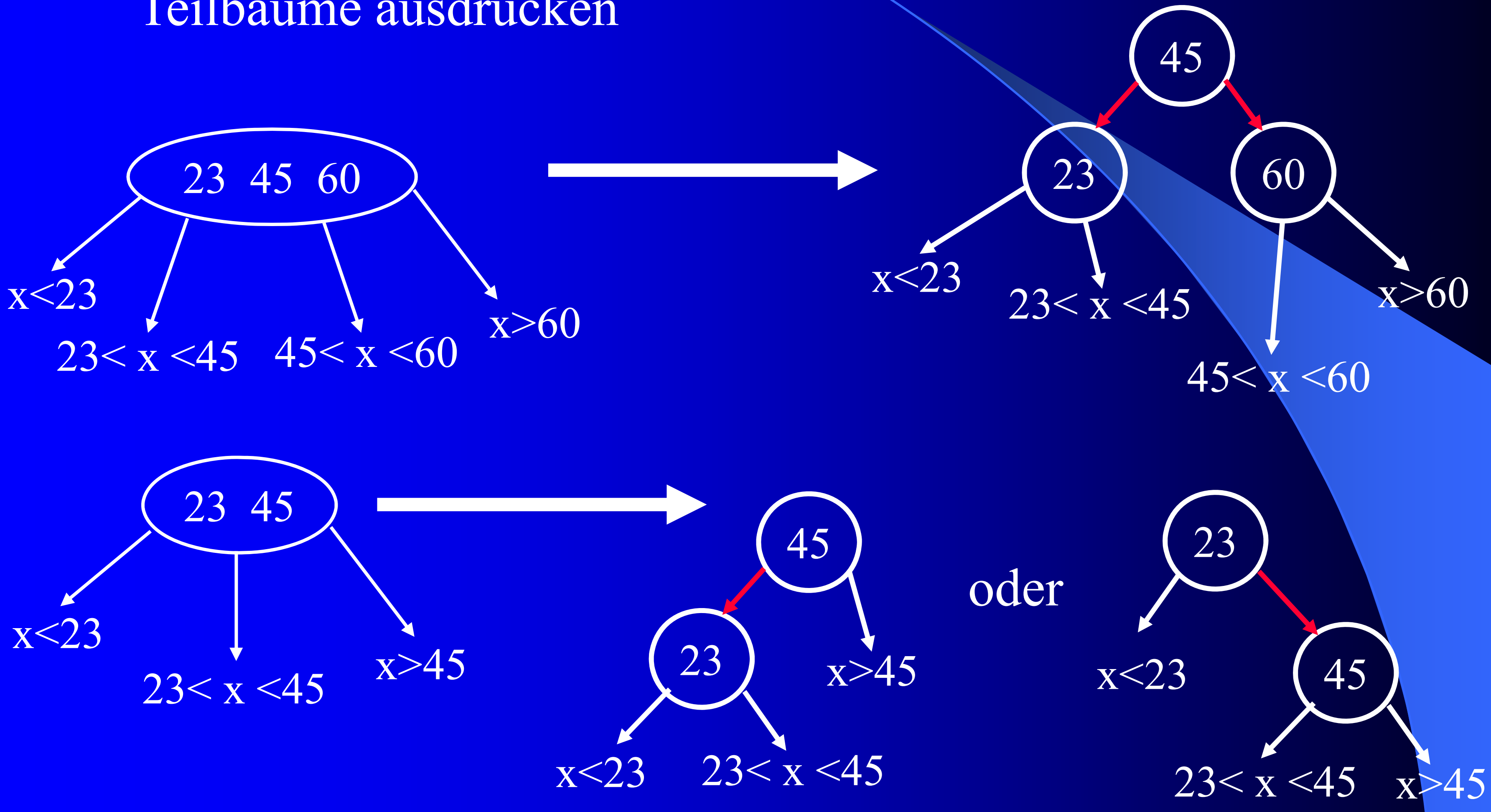
Top-Down 2-3-4-Bäume: Eigenschaften

- da der Baum nur an der Wurzel wachsen kann, ist er immer ausgeglichen
- dadurch liegt das Suchen in O(log N)
- das Einfügen liegt im ungünstigsten Fall in O(log N)
- der ungünstigste Fall tritt sehr selten ein

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

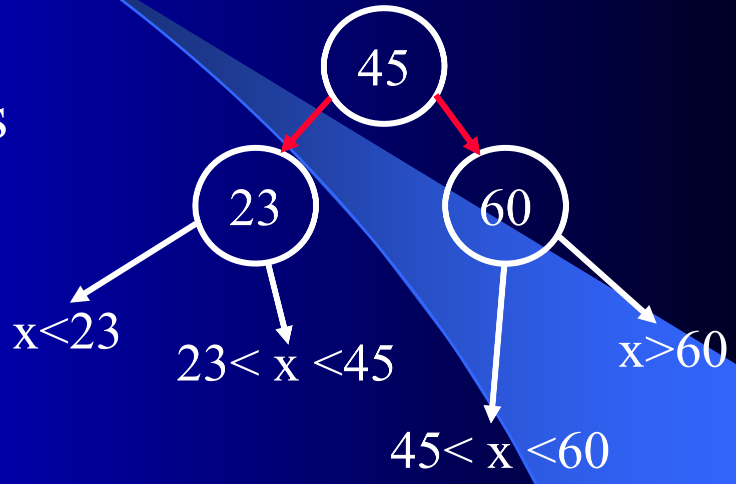
Rot-Schwarz Bäume

- 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;
        NodeRef m_Left = new NodeRef();
        NodeRef m_Right = new NodeRef();
        boolean m_bIsRed = true;
    }
...
    private NodeRef m_Root = new NodeRef();
...
```

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

wenn der Schlüssel gefunden ist, gibt den Datensatz zurück

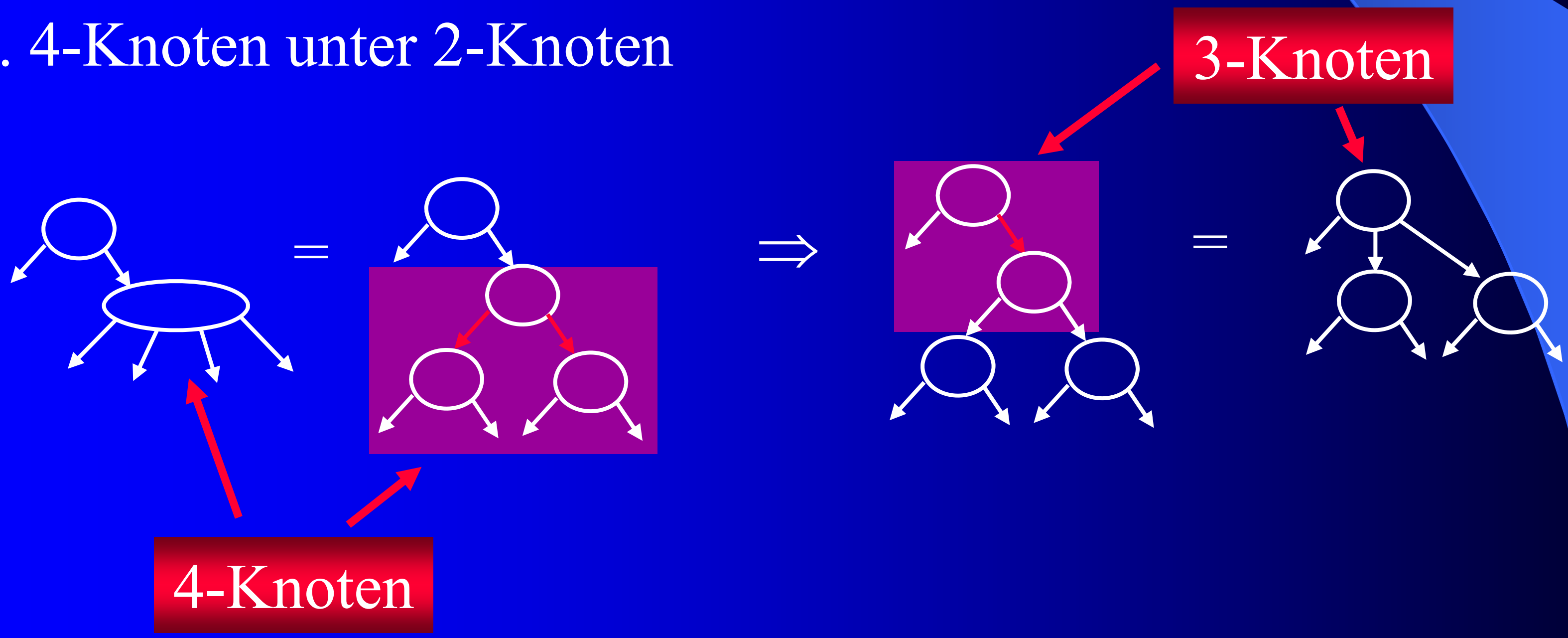
steige links bzw. rechts ab

Schlüssel ist nicht gefunden worden

Rot-Schwarz Bäume: Implementierung (Fort.)

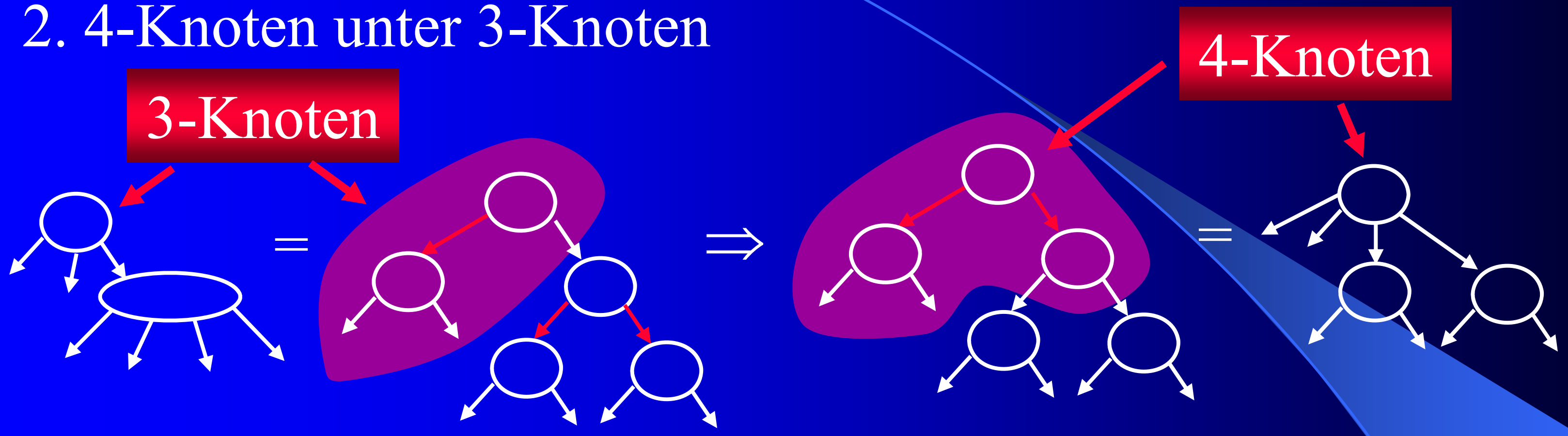
- beim Einfügen werden alle 4-Knoten aufgeteilt
- ein 4-Knoten erkennt man daran, dass beide Nachfolger-knoten das gesetzte Flag haben
- nicht sehr teuer, da es kaum 4-Knoten gibt
- es gibt 7 Fälle zu untersuchen

1. 4-Knoten unter 2-Knoten

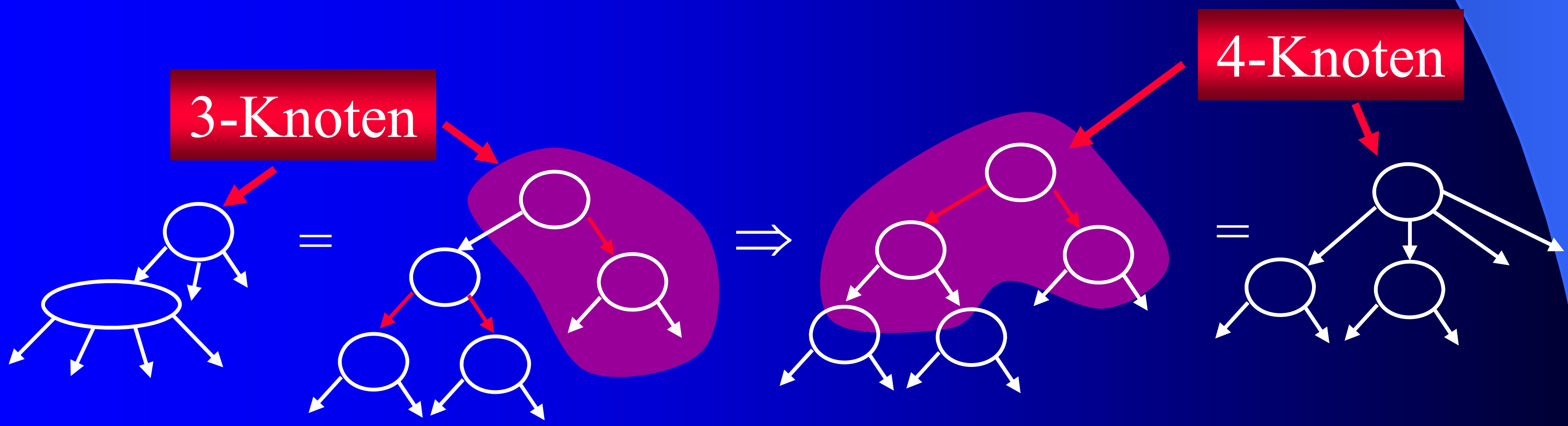


Rot-Schwarz Bäume: Implementierung (Fort.)

2. 4-Knoten unter 3-Knoten

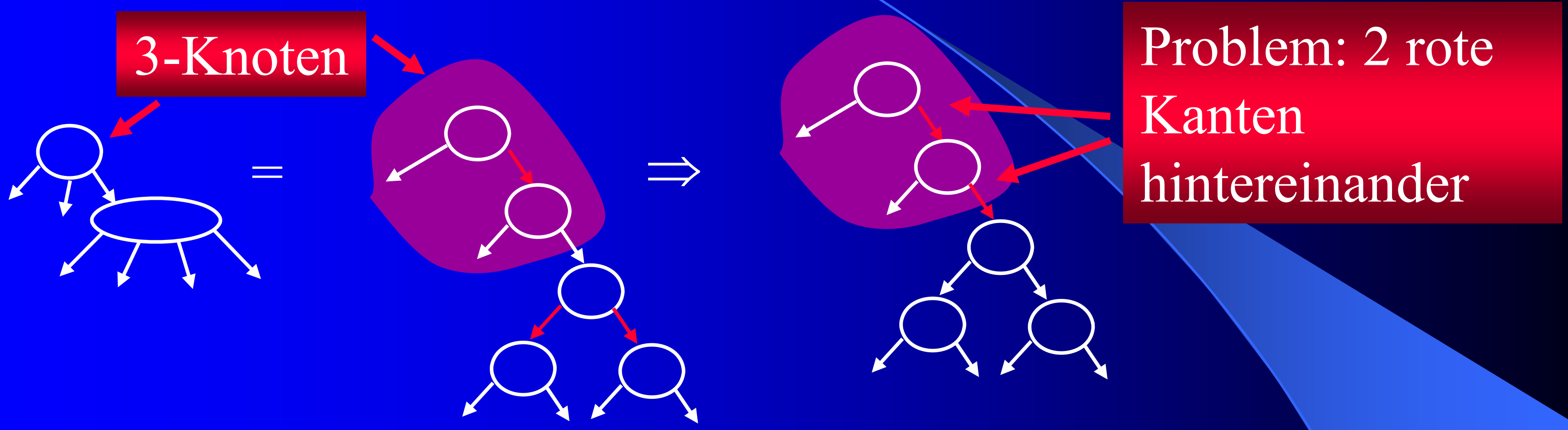


3. 4-Knoten unter 3-Knoten

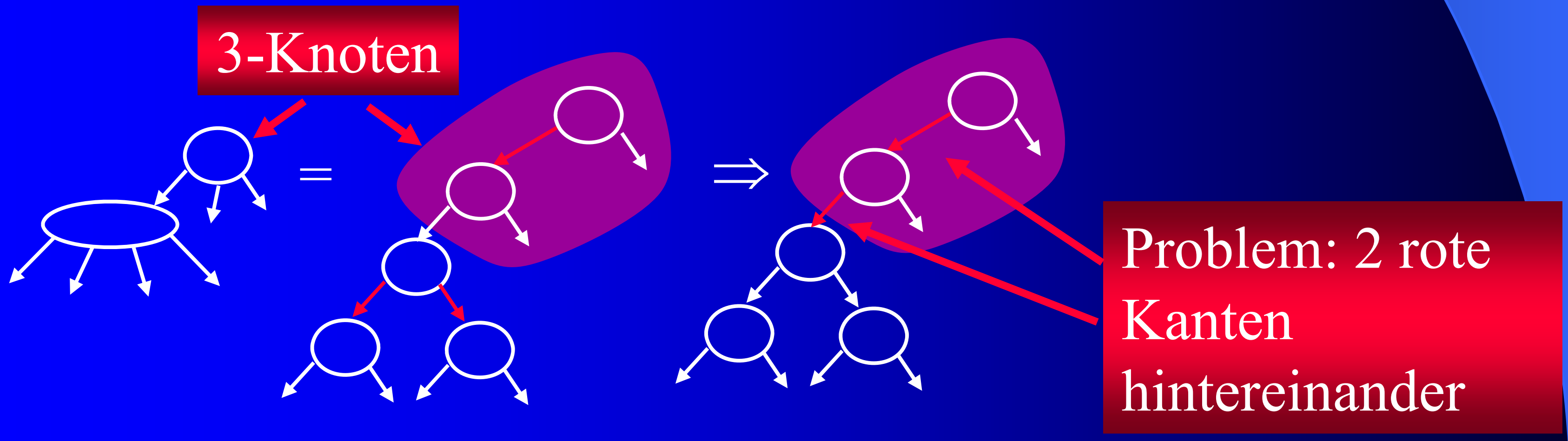


Rot-Schwarz Bäume: Implementierung (Fort.)

4. 4-Knoten unter 3-Knoten

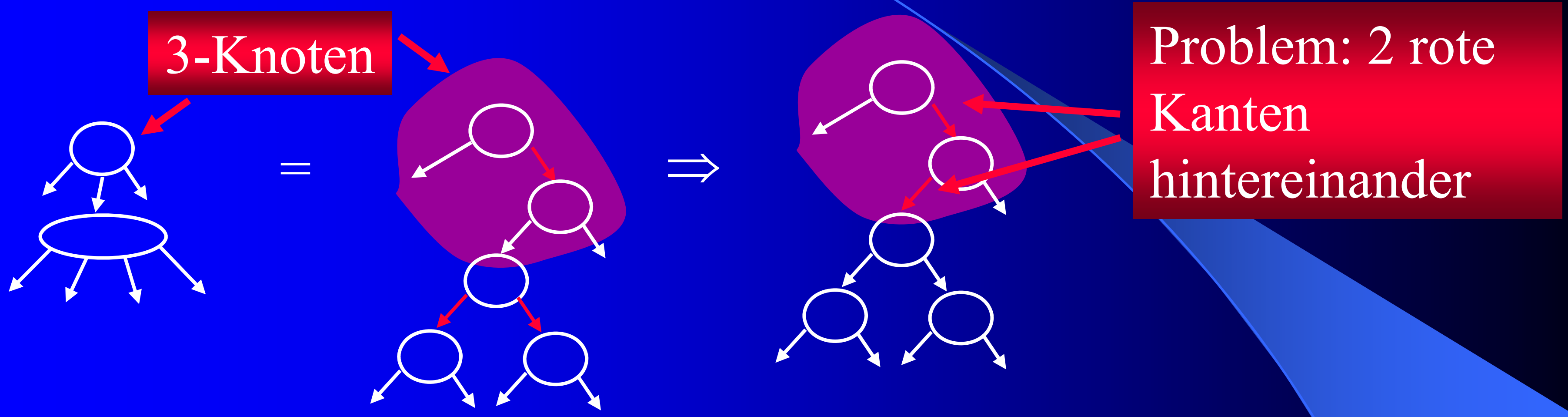


5. 4-Knoten unter 3-Knoten

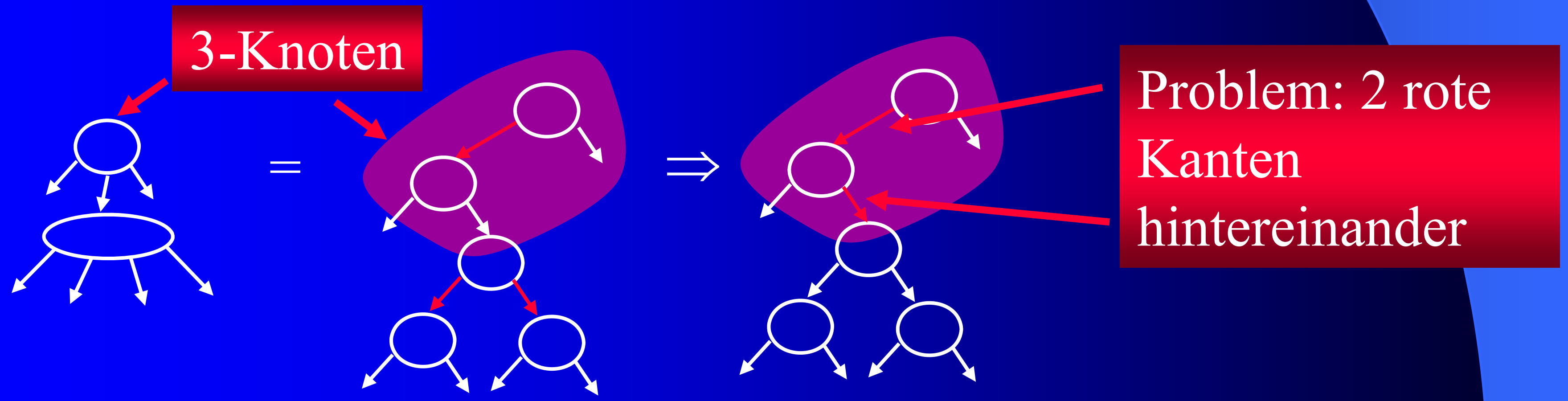


Rot-Schwarz Bäume: Implementierung (Fort.)

6. 4-Knoten unter 3-Knoten



7. 4-Knoten unter 3-Knoten

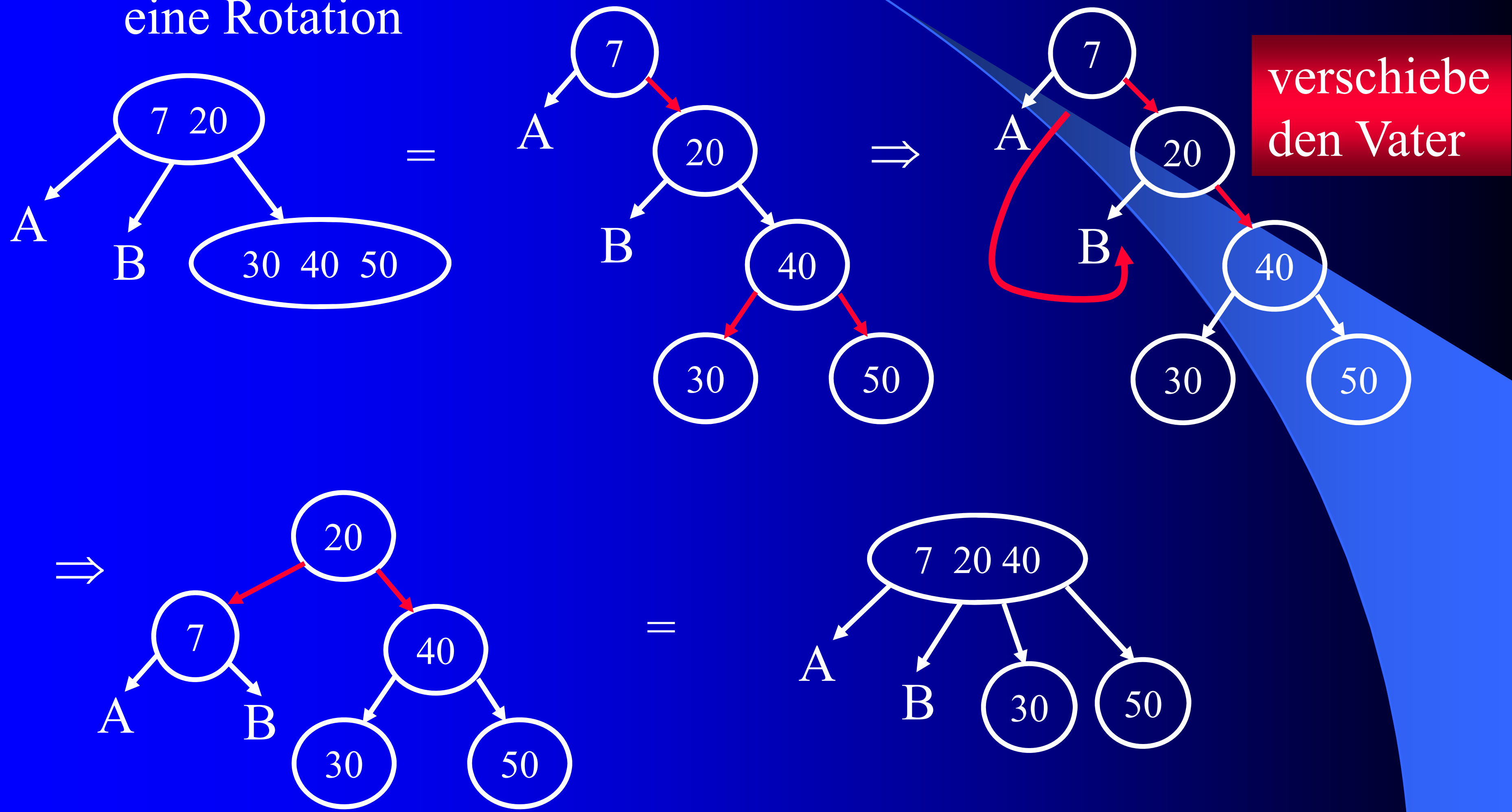


Rot-Schwarz Bäume: Implementierung (Fort.)

- 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

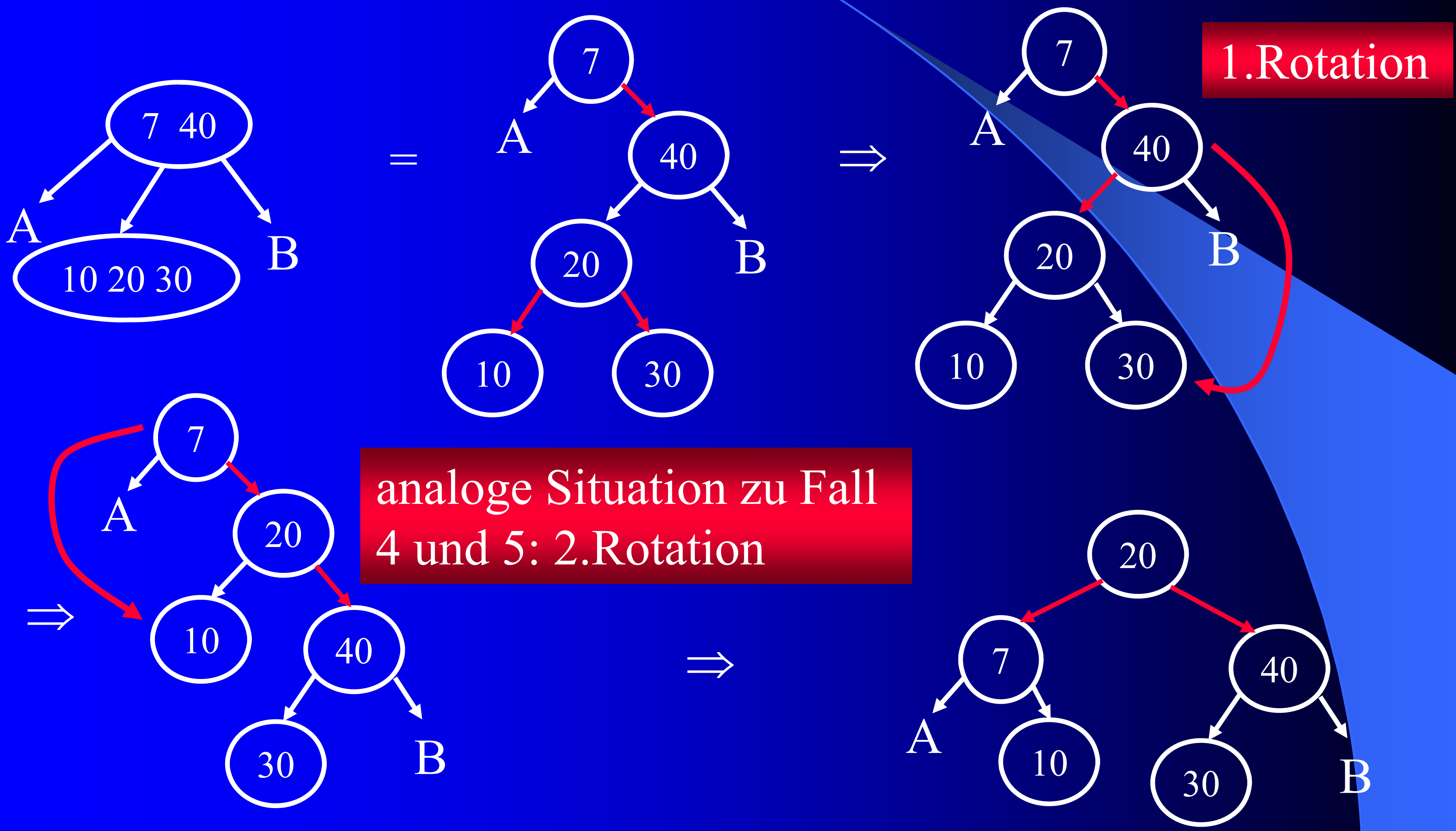
Rot-Schwarz Bäume: Implementierung (Fort.)

- 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 10

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?

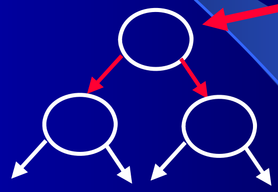
Rot-Schwarz Bäume: Implementierung (Fort.)

- 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

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

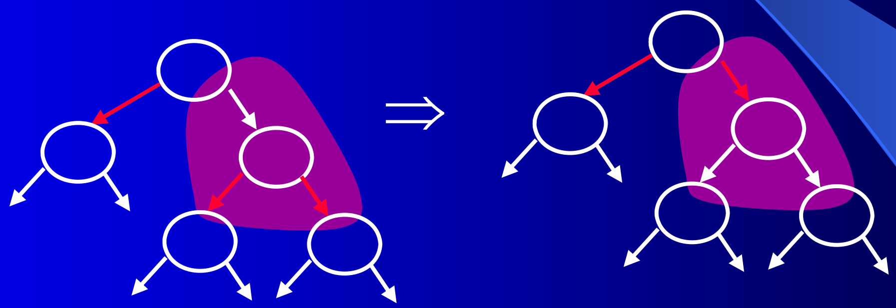


```
class Node {
...

        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

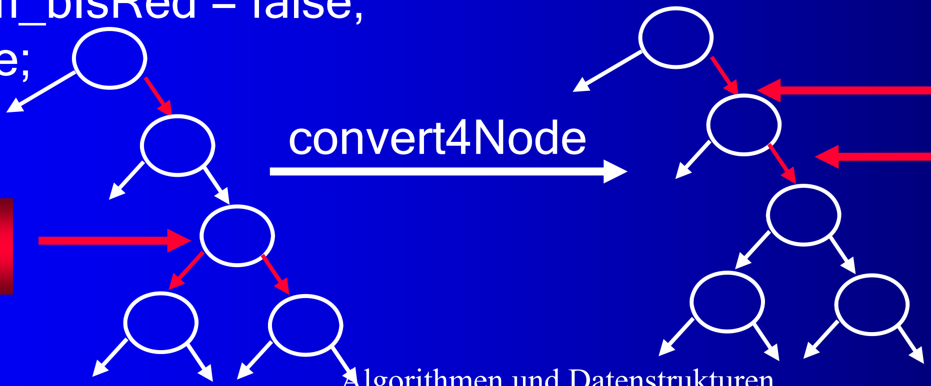
Rot-Schwarz Bäume: Implementierung (Fort.)

- 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

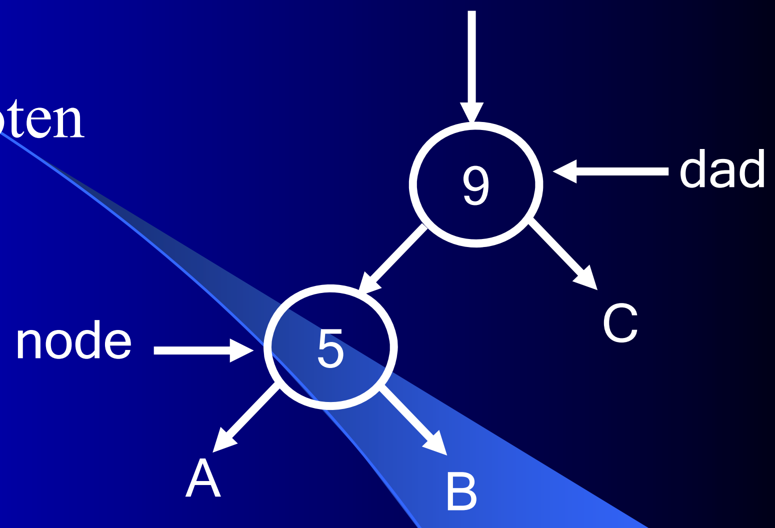
Rot-Schwarz Bäume: Implementierung (Fort.)

- 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.)

- **unvollständige** Rotation zweier Knoten



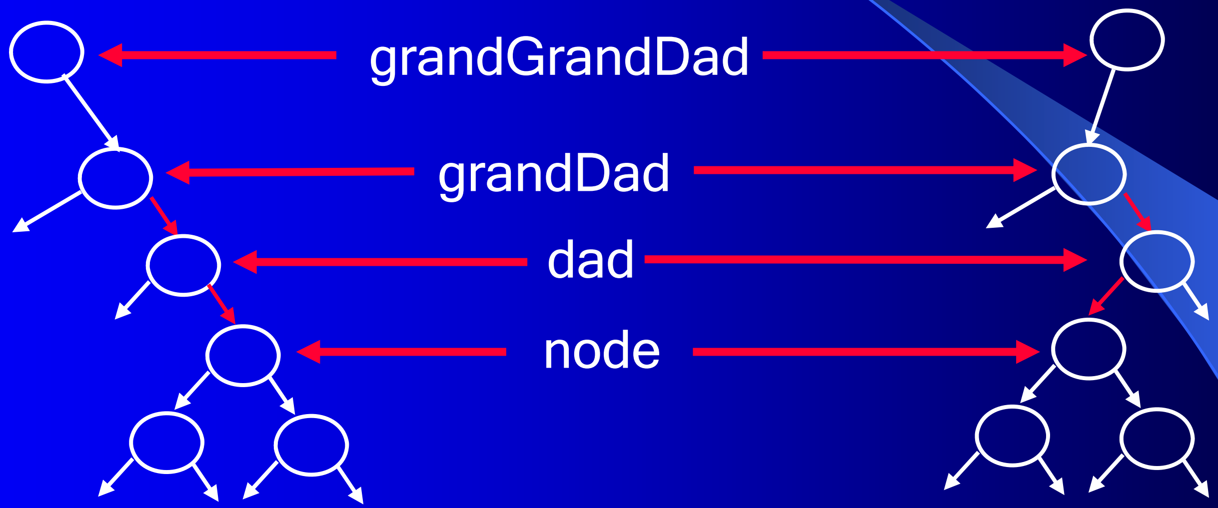

```
void rotate(Node dad,Node node) {
    boolean nodeColour = node.m_bIsRed;
    node.m_bIsRed = dad.m_bIsRed;
    dad.m_bIsRed = nodeColour;
    if (dad.m_Left == node) {
        // clockwise rotation
        dad.m_Left = node.m_Right;
        node.m_Right = dad;
    } else {
        // counter-clockwise 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

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.)

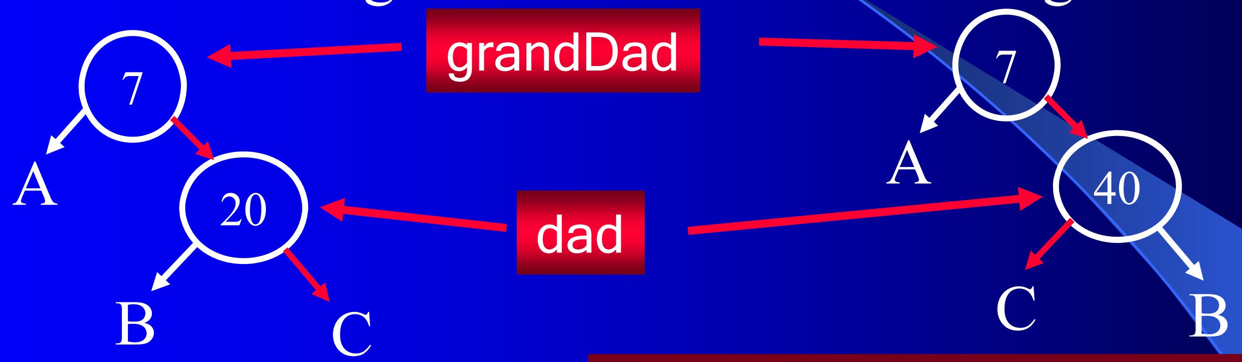
- die split Methode ist eine Methode des NodeHandlers
- sie wird nur von Knoten mit roten Kanten aufgerufen
- wenn der Vater existiert und auch rot ist, muss rotiert werden

```
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 ?



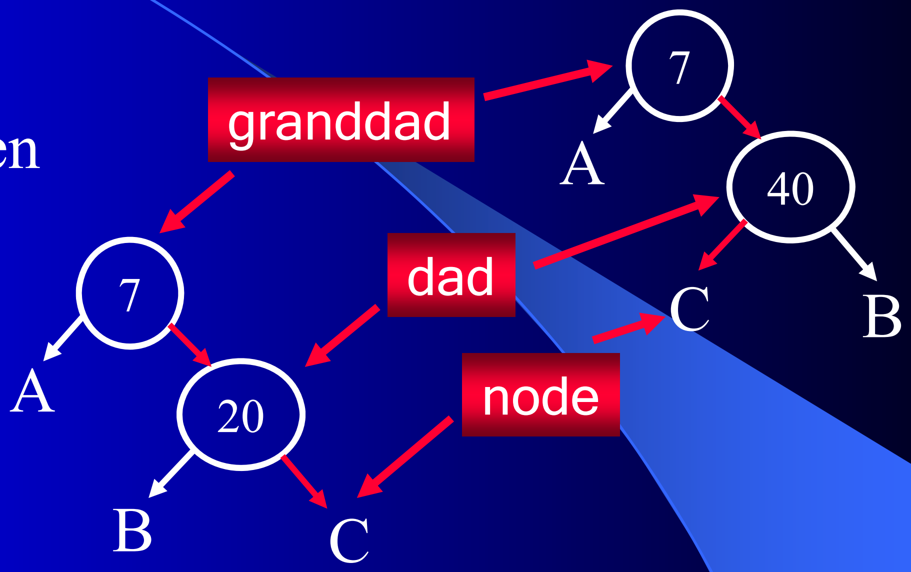
wenn es einen roten Vater gibt, muss es einen Großvater geben (weil, die Wurzel ist niemals rot)

```
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

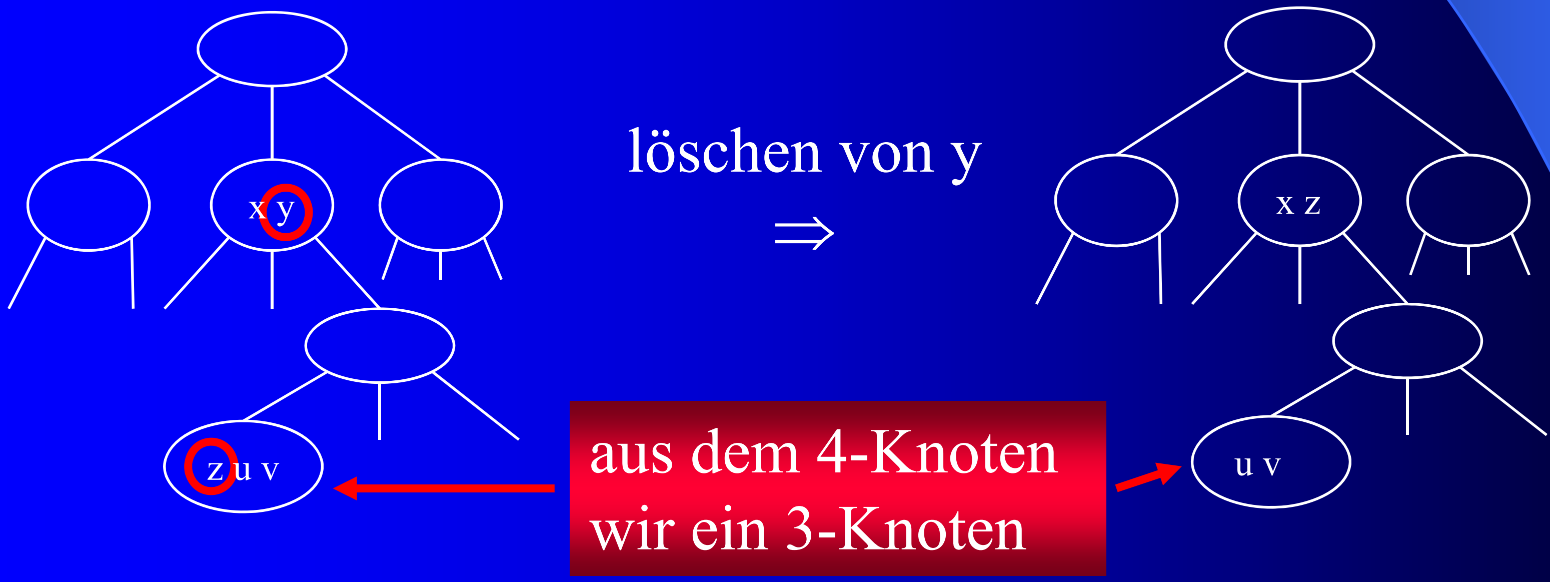
Vorlesung 11

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



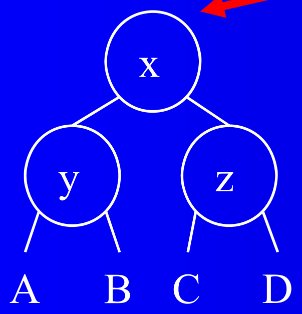
Löschen aus Top-Down 2-3-4 Bäumen (Forts.)

- 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

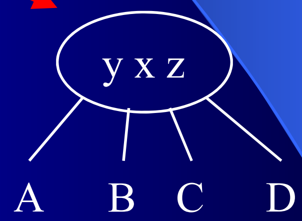
Wurzel



Wurzel und Söhne werden zu neuer

$\Rightarrow$

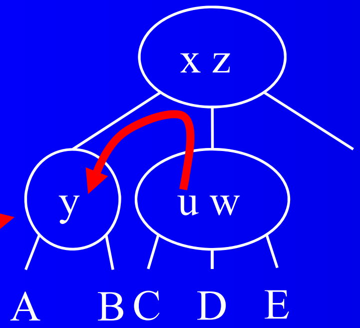
Wurzel zusammengefasst



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

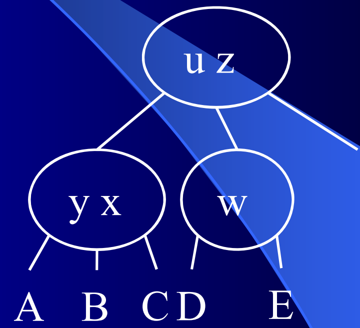
Fall 2

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



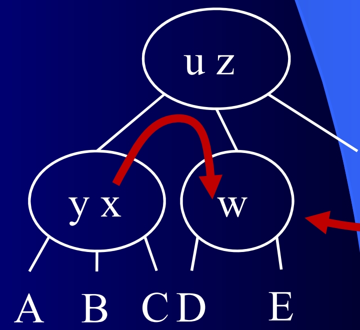
Linksrotation

$\Rightarrow$



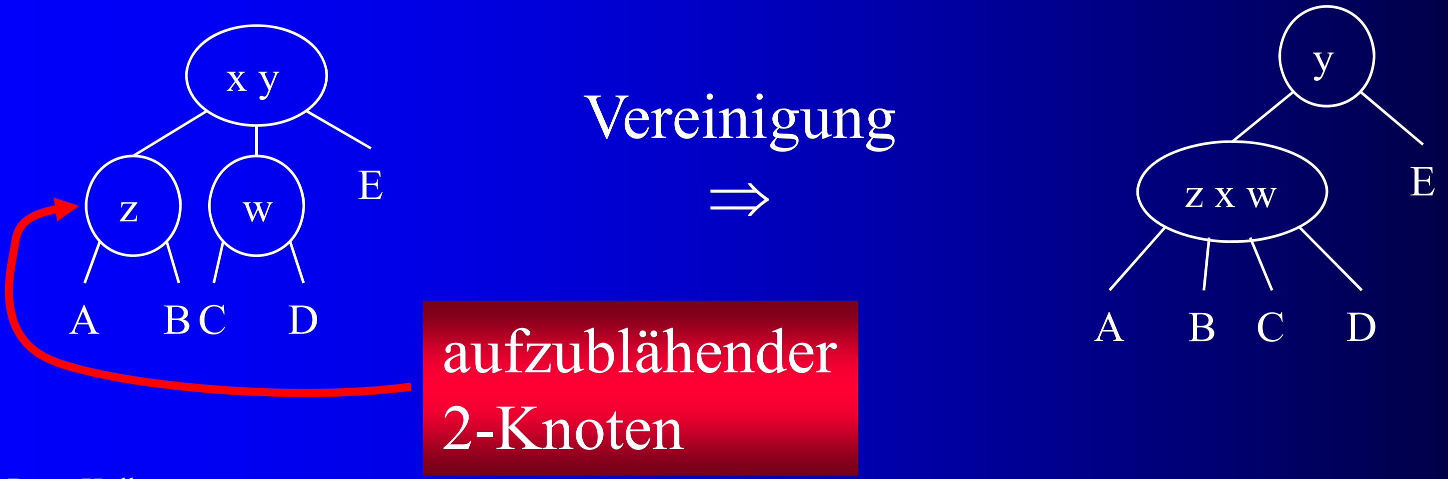
aufzublähender
2-Knoten

- gibt es natürlich auch als Rechtsrotation



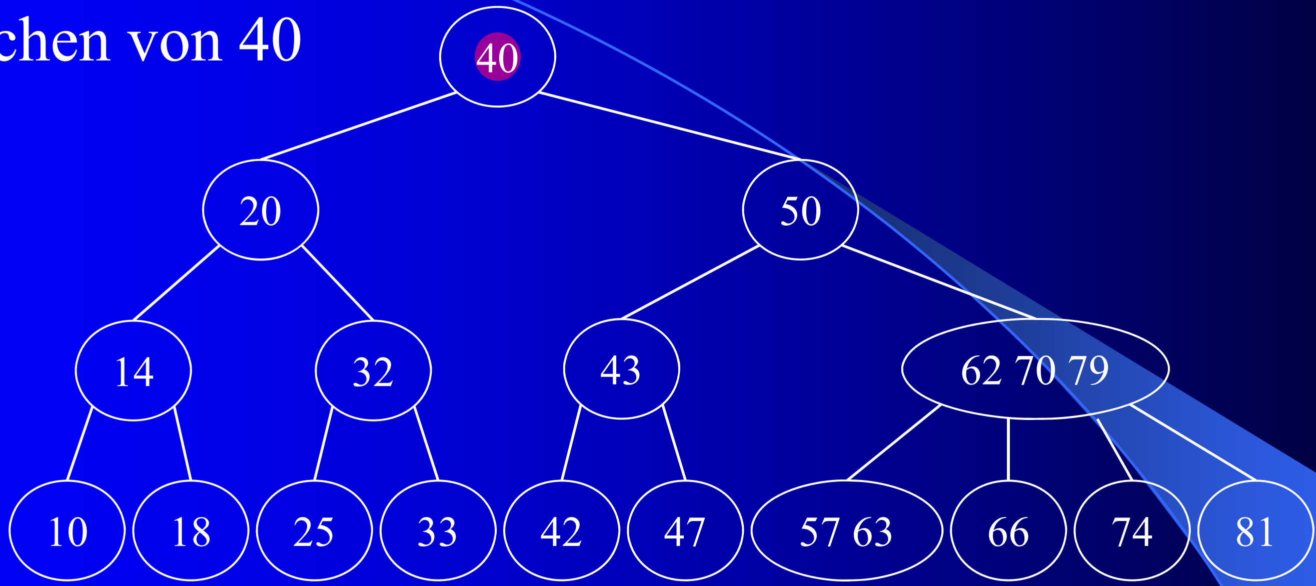
Fall 3

- 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)

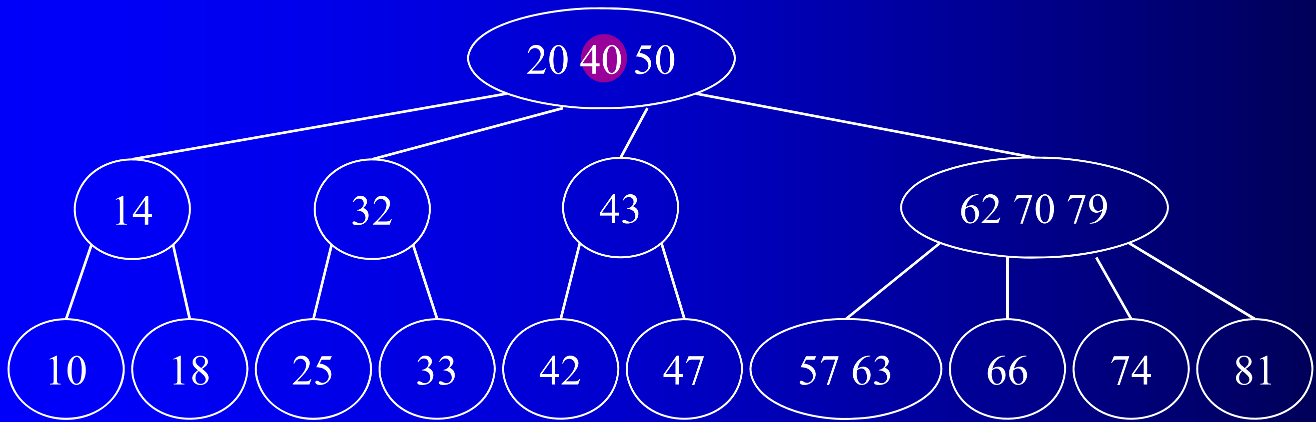


Beispiel

- Löschen von 40

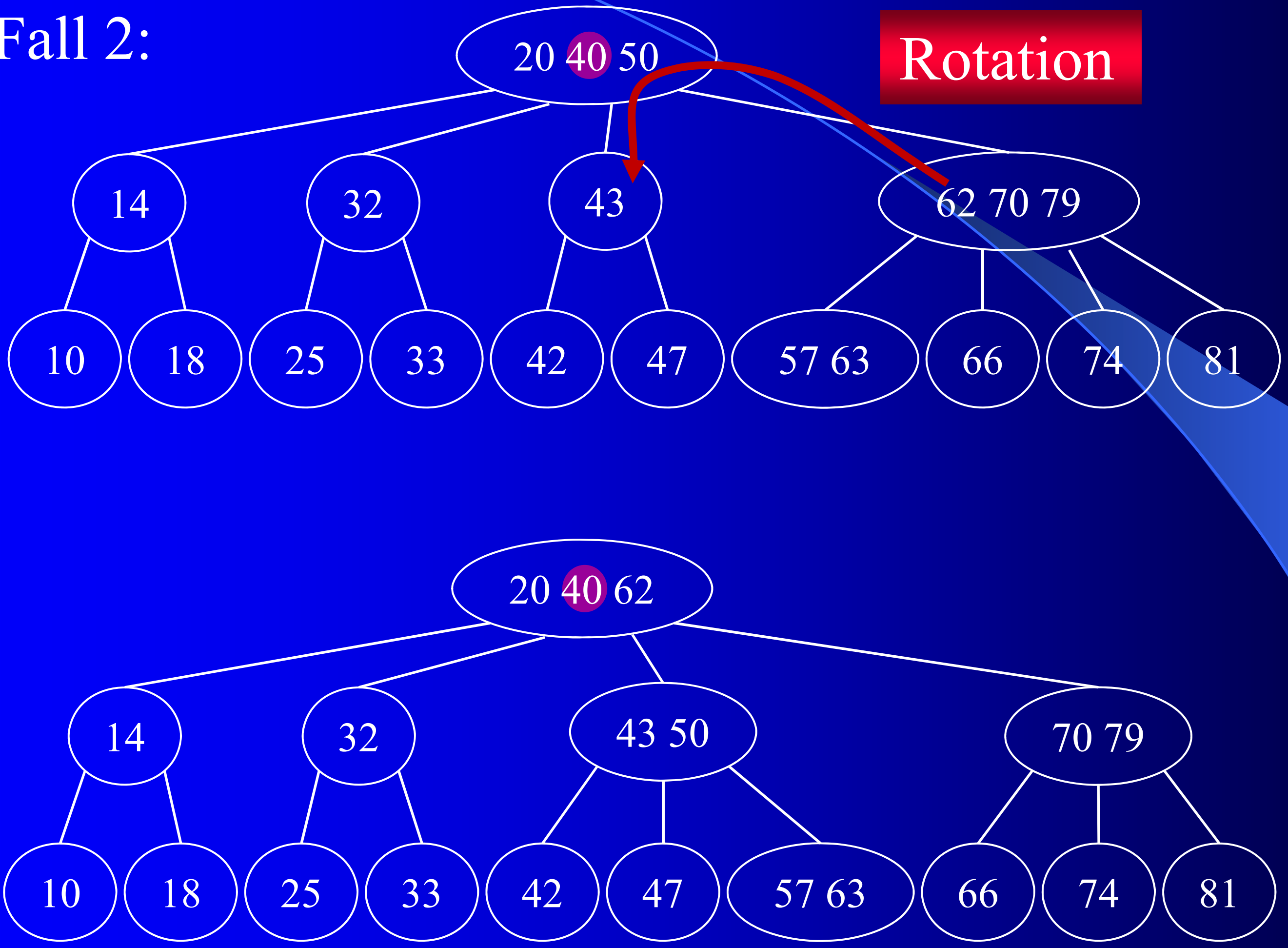


- Fall 1: Wurzel und beide Söhne zusammenfassen



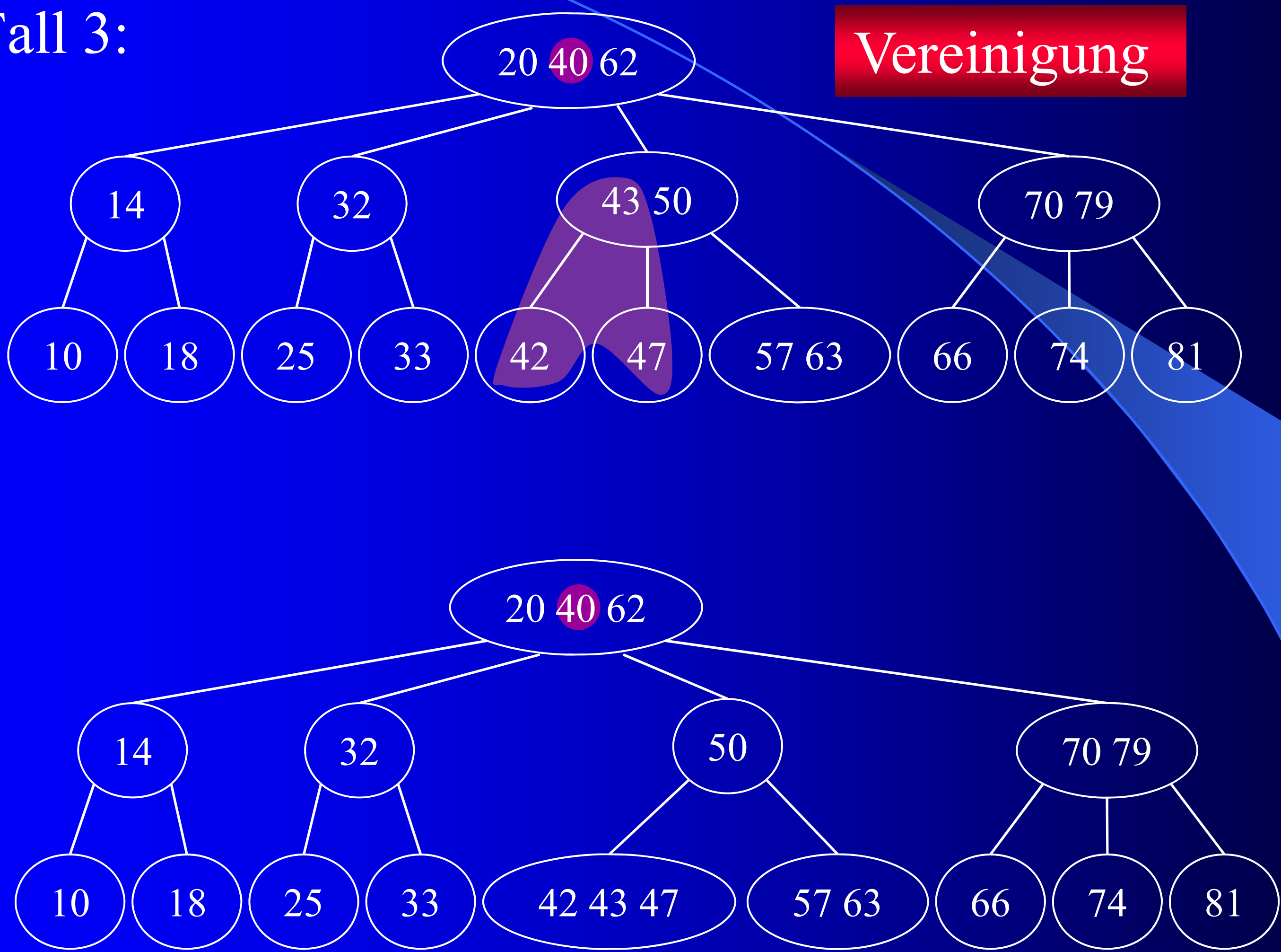
Beispiel (Forts.)

- Fall 2:



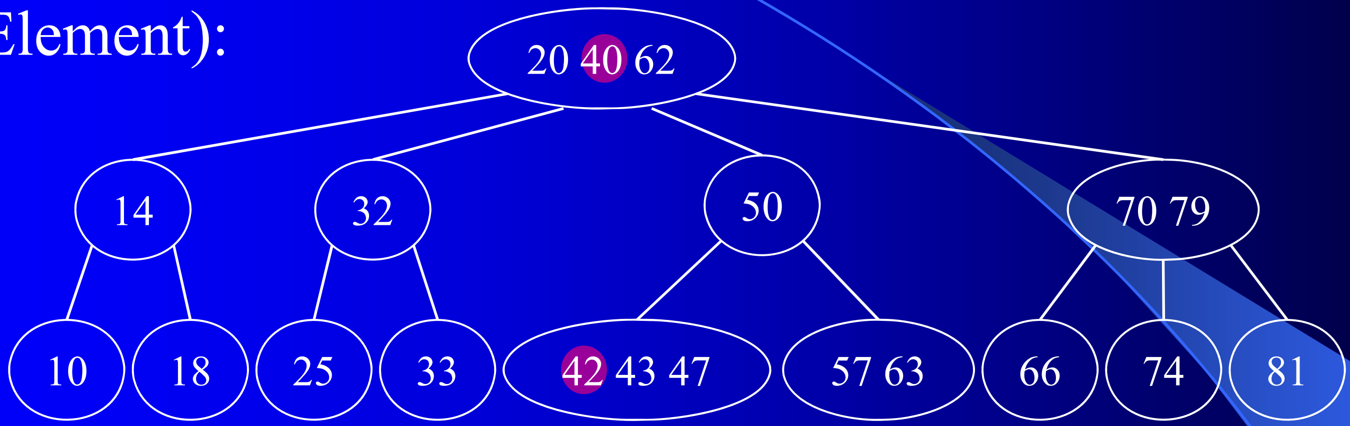
Beispiel (Forts.)

- Fall 3:

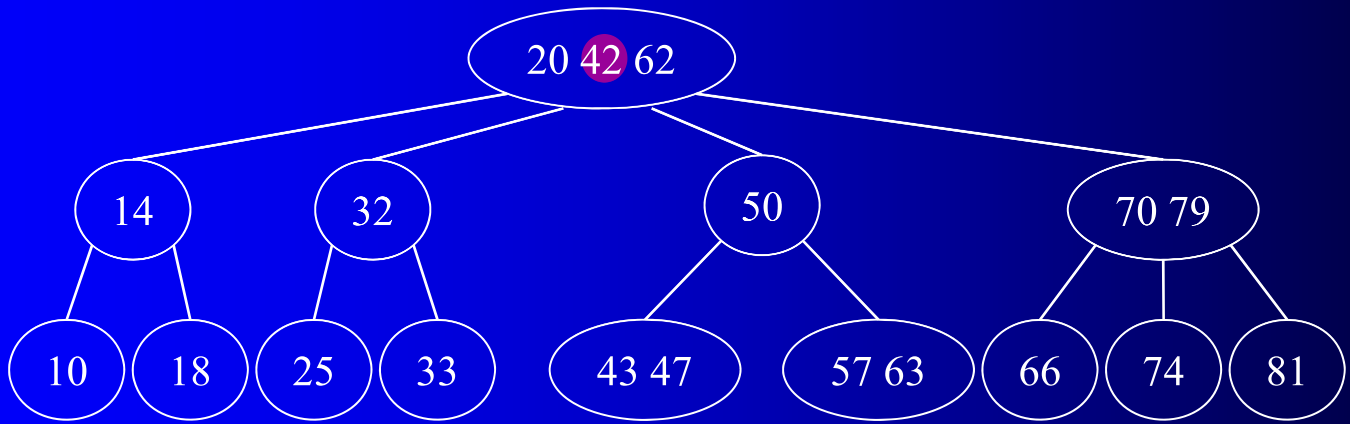


Beispiel (Forts.)

- 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)

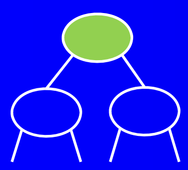
$\Rightarrow$ 26 (!!!) Fälle auf Ebene der Top-Down 2-3-4 Bäume

$\Rightarrow$ 46 (!!!) Fälle auf Ebene der Rot-Schwarz Bäume (sehr viele symmetrische Fälle)

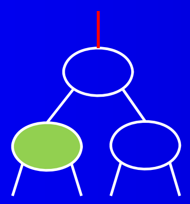
Fallunterscheidung (Forts.)

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

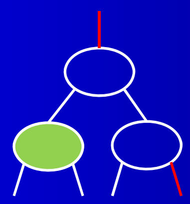
1. Wurzelfall



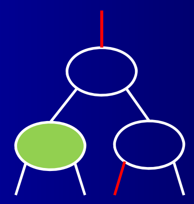
2. 2er unter 3er oder 4er mit 2er Bruder



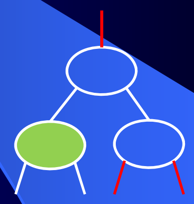
3. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder



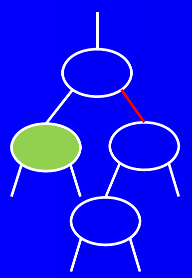
4. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder



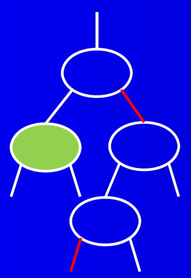
5. 2er unter 3er oder 4er oder Wurzel (!!!) mit 4er Bruder



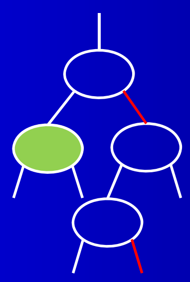
6. 2er unter 3er mit 2er Bruder



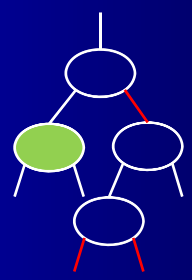
7. 2er unter 3er mit 3er Bruder



8. 2er unter 3er mit 3er Bruder

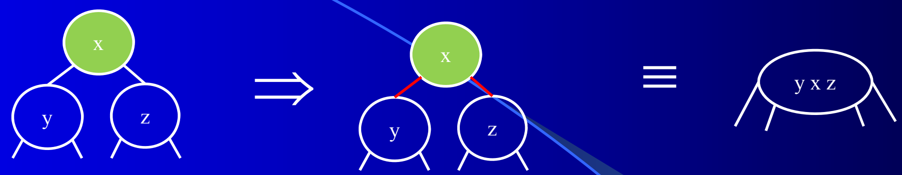


9. 2er unter 3er mit 4er Bruder

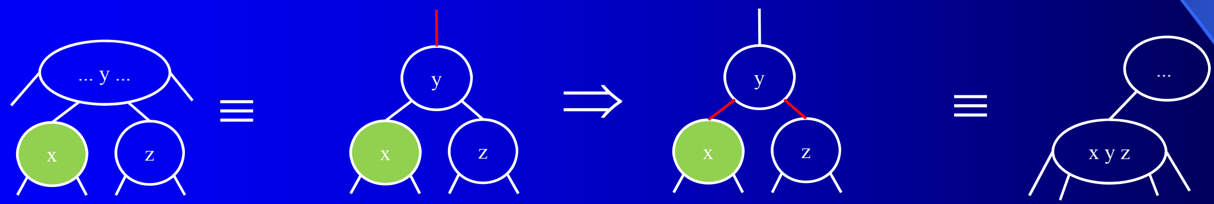


Fallunterscheidung (Forts.)

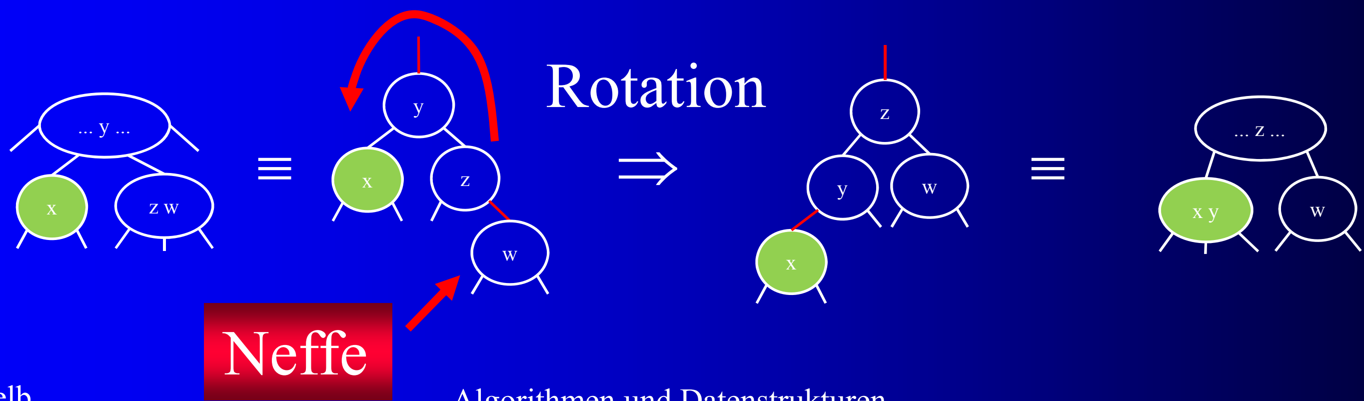
- 1. Wurzelfall



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

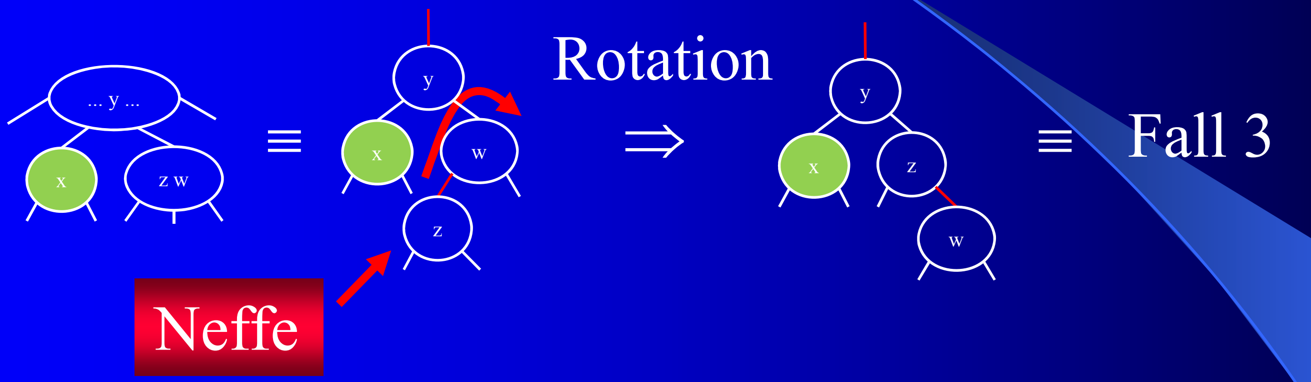


- 3. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder

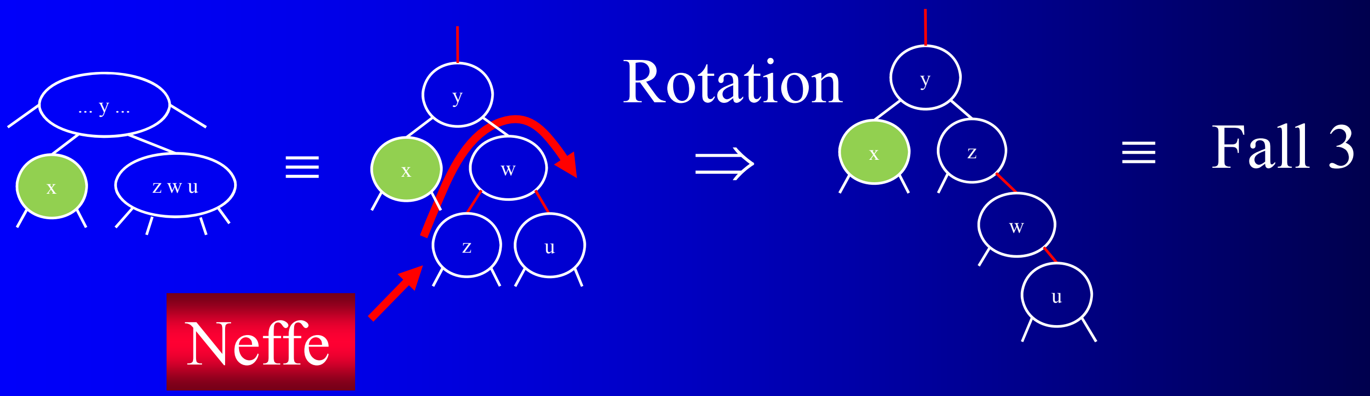


Fallunterscheidung (Forts.)

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

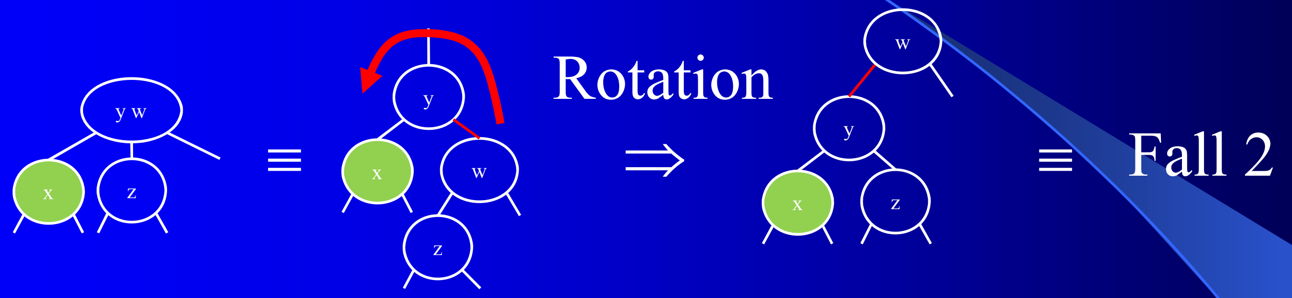


5. 2er unter 3er oder 4er oder Wurzel (!!!) mit 4er Bruder

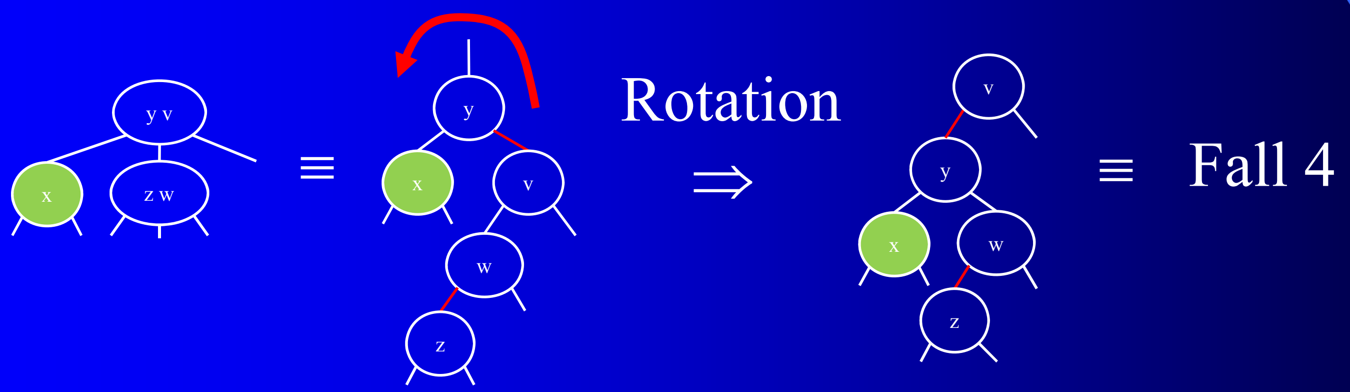


Fallunterscheidung (Forts.)

6. 2er unter 3er mit 2er Bruder

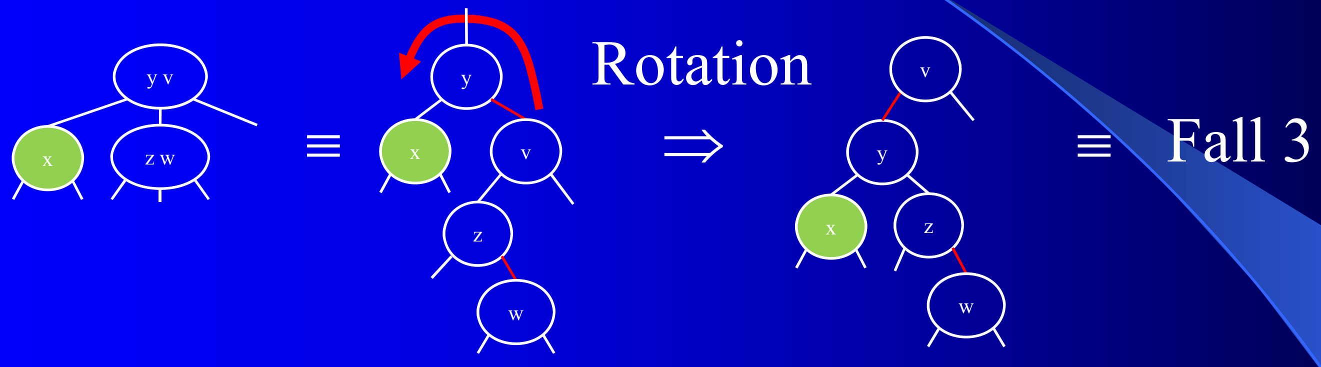


7. 2er unter 3er mit 3er Bruder

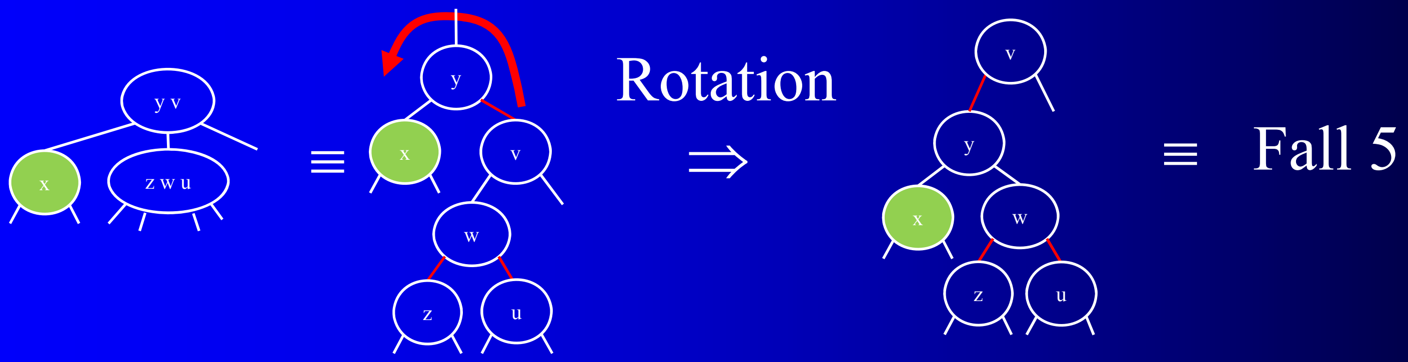


Fallunterscheidung (Forts.)

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);
            }
            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

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 Kantenfarbe 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

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] = node(G_DAD);
    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

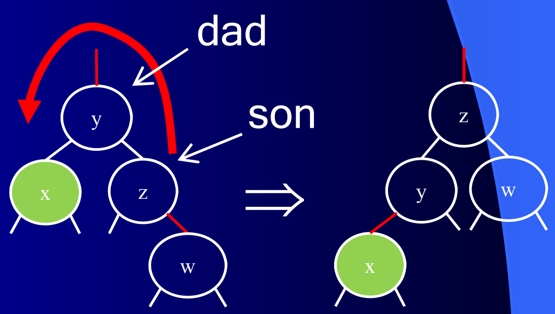
```
void rotate(int kind) {
      Node dad = node(kind);
      Node son = node(kind-1);
      boolean sonColour = son.m_bIsRed;
      if (!sonColour) {
            if (son.m_Left != null)
                  son.m_Left.m_bIsRed = false;
            if (son.m_Right != null)
                  son.m_Right.m_bIsRed = 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



Vorlesung 12

Digitales Suchen

Nachteile des Hashings:

- der gesamte Schlüssel wird immer mit den eingetragenen Schlüsseln verglichen
- die Berechnung eines Indexes aus einem Schlüssel kann u.U. relativ aufwendig sein (siehe Hashing für Strings)

Idee:

- baue Binärbaum auf, der jedoch für jedes Bit des Schlüssels eine Links-/Rechts-Verzweigung vornimmt
- nach Abarbeitung jeden Bits eines Schlüssels hat man den gesuchten Schlüssel gefunden oder er ist nicht vorhanden

Digitales Suchen: Motivation

Vorteile des digitalen Suchens:

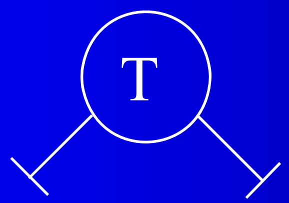
- nicht so kompliziert wie ausgeglichene Bäume (Rot-Schwarz-Bäume)
- trotzdem annehmbare Tiefen (damit Laufzeit) für ungünstige Anwendungen

Digitales Suchen: 1. Beispiel

Schlüssel sind Buchstaben:

- von jedem Buchstaben seine Binärcodierung betrachten
- hier: betrachte nur die Bits, in denen ein Unterschied besteht: Bit 0 bis 5
- Bit 6 und Bit 7 sind konstant 1 bzw. 0

T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210

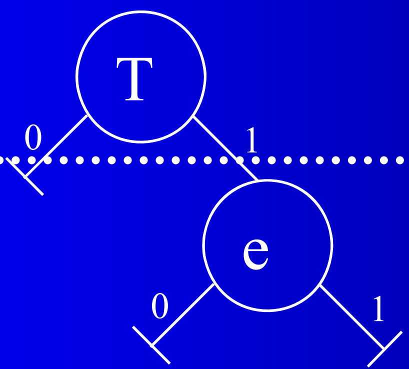


initiale Suchbaum, nur der Buchstabe 'T' ist eingetragen

Digitales Suchen: 1. Beispiel (Forts.)

- Buchstabe 'e' soll in den Baum eingetragen werden
- dazu werden solange die Bits 0 bis 5 entlanggegangen, bis ein Blatt erreicht ist
- bei 0 wird nach links gegangen
- bei 1 wird nach rechts gegangen

T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210



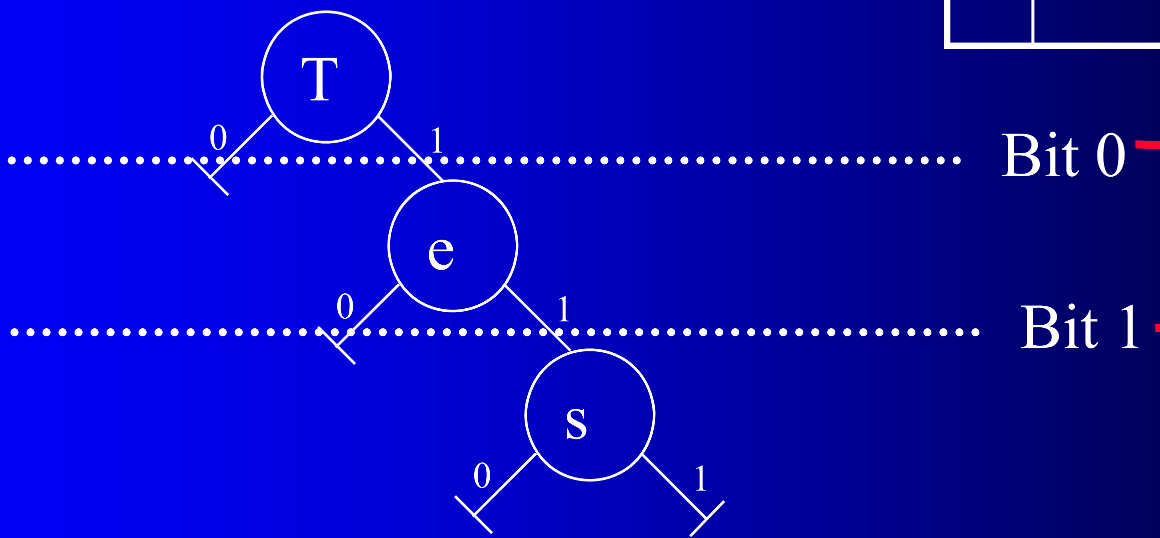
Bit 0

Suchbaum, mit 'T' und 'e'

Digitales Suchen: 1. Beispiel (Forts.)

- Buchstabe 's' soll in den Baum eingetragen werden
- dazu werden solange die Bits 0 bis 5 entlanggegangen, bis ein Blatt erreicht ist
- bei 0 wird nach links gegangen
- bei 1 wird nach rechts gegangen

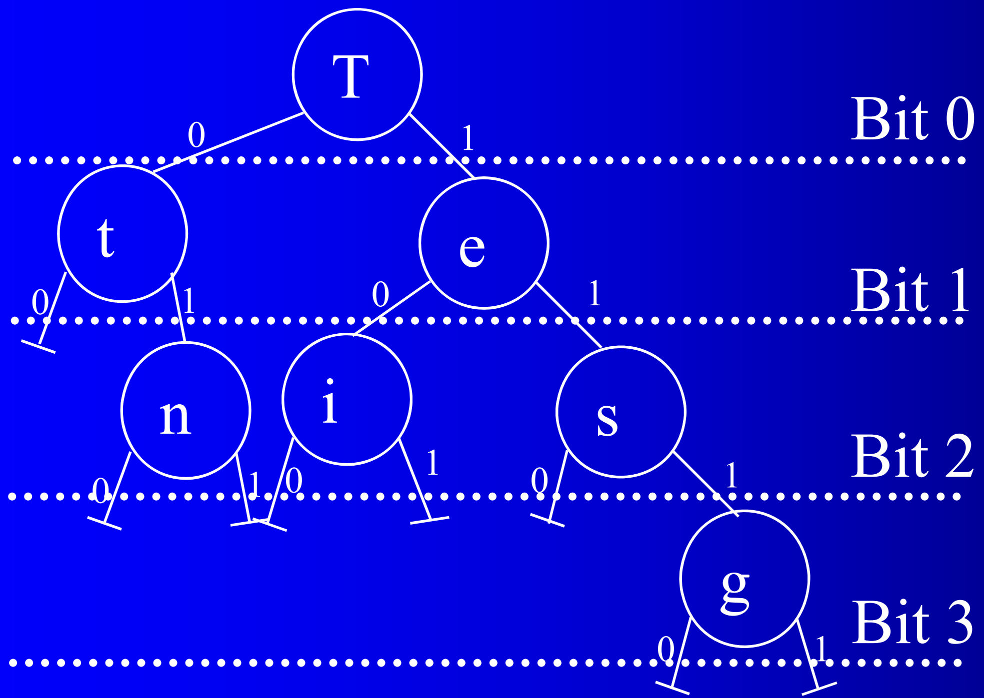
T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210



Suchbaum, mit 'T', 'e' und 's'

Digitales Suchen: 1. Beispiel (Forts.)

- nachdem alle Buchstaben eingefügt sind, sieht der digitale Suchbaum wie folgt aus:



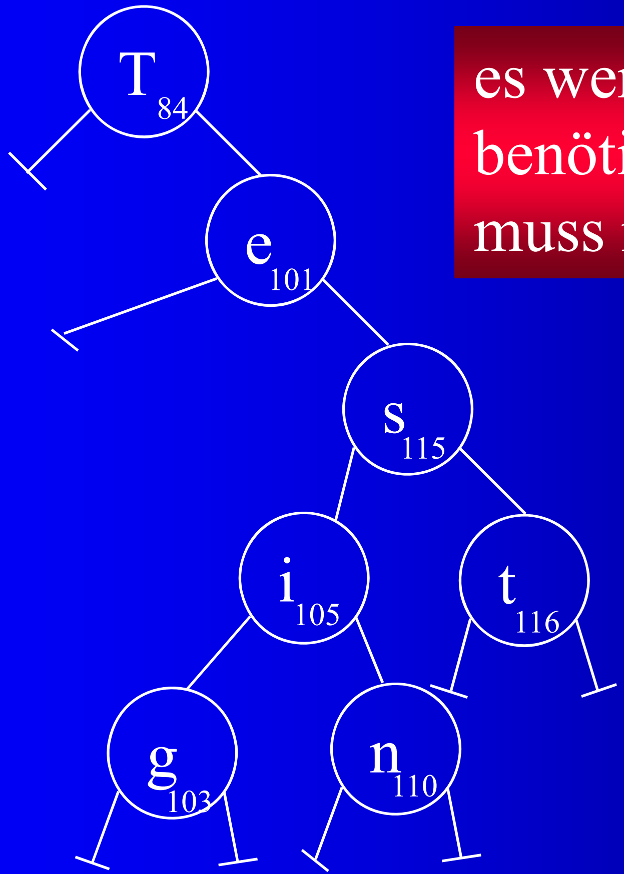
T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210

es werden nur 4 von den maximalen 6 Bits angeschaut

Digitales Suchen: 1. Beispiel (Forts.)

- der zugehörige Binärbaum hätte folgende Form:

T	84
e	101
s	115
t	116
i	105
n	110
g	103



es werden 5 Ebenen benötigt, aber dass muss nicht sein

ohne ‚g' wären hier auch 5 Ebenen notwendig, beim digitalen Suchbaum nur 3

Digitales Suchen: Implementierung

```
class DigiTree {
    class Node {
        public Node(char key) {
            m_Key = key;
        }
        public char m_Key;
        public Node m_Left = null;
        public Node m_Right = null;
    }

    public boolean search(char c) {...}

    public void insert(char c) {...}

    private Node m_Root = null;
}
```

normalerweise sollte in einem Knoten neben dem Schlüssel auch das assoziierte Datum gespeichert werden

wie gehabt in BinTree oder RedBlackTree

Digitales Suchen: Implementierung (Forts.)

solange noch Knoten vorhanden sind ...

... teste Bit 0, Bit 1, Bit2 usw. durch

```
class DigiTree {
    ...
    public boolean search(char c) {
        Node tmp = m_Root;
        for(int i = 0; tmp != null; ++i) {
            if (tmp.m_Key == c)
                return true;
            tmp = (c & (1 << i)) != 0 ? tmp.m_Right : tmp.m_Left;
        }
        return false;
    }
    ...
}
```

Ist das i-te Bit im Schlüssel gesetzt ...

... dann nimm den rechten Nachfolger ...

... sonst den Linken !

Digitales Suchen: Implementierung (Forts.)

```
class DigiTree {
    ...
    public void insert(char c) {
        NodeHandler h = new NodeHandler(m_Root);
        int i = 0;
        for(i = 0;!h.isNull();++i)
            h.down((c & (1 << i)) == 0);
        h.set(new Node(c),(c & (1 << (i-1))) == 0);
    }
    ...
}
```

solange noch Knoten vorhanden sind ...

... teste Bit 0, Bit 1, Bit2 usw. durch

Abstieg nach Links und Rechts

Der NodeHandler muss wissen, ob der neue Knoten links oder rechts unter den Vater eingefügt werden soll

Digitales Suchen: Diskussion

- Vorteile gegenüber dem Hashing: nachdem ***maximal*** alle Bits ***angeschaut*** worden sind, kann entschieden werden, ob der gesuchte Schlüssel vorhanden ist
- Für ***jede Bitposition*** ist ein ***Vergleich*** mit dem ***aktuellen Schlüssel*** und dem ***gesuchten Schlüssel*** notwendig
- dies kann ein erheblicher Aufwand bei langen Schlüsseln sein (Beispiel: Strings mit ca. 20 Zeichen, 6 Bits pro Zeichen: maximal 120 Stringvergleiche)
- bei langen Schlüsseln (Schlüssel mit vielen Bits) dominiert der Schlüsselvergleich den Baumdurchlauf

Digitale Such-Tries

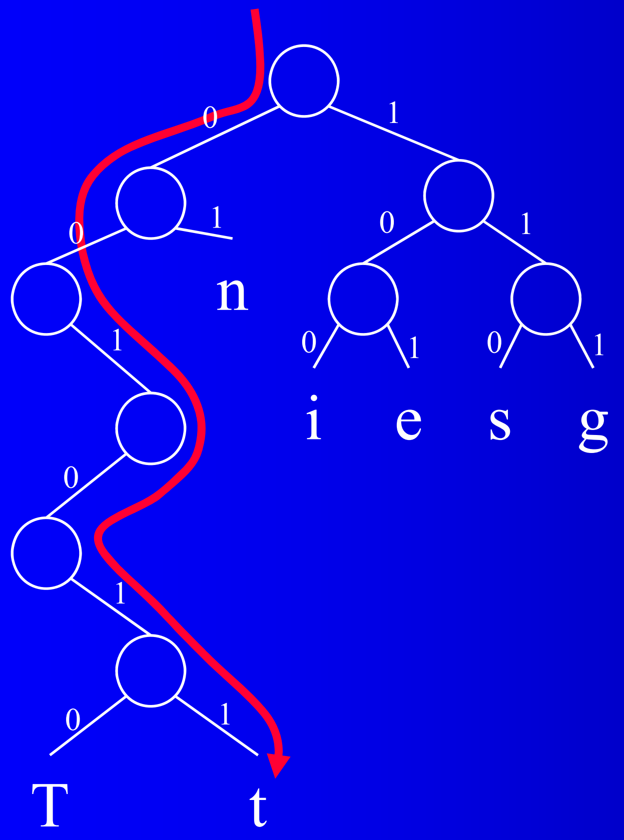
Ähnlich wie digitale Suchbäume, jedoch

- werden in den ***Knoten keine Schlüssel*** gespeichert
- ***nur*** in den ***Blättern*** werden die ***Schlüssel gespeichert***

Suchen und Einfügen erfolgen durch

- Abstieg analog zu den digitalen Suchbäumen, jedoch
- ***kein Schlüsselvergleich*** in den ***Knoten,*** sondern
- ***Schlüsselvergleich*** am Blatt, dadurch
- in dem Fall nur ***exakt ein Schlüsselvergleich***

Digitale Such-Tries: Beispiel



T	84	01010100
e	101	01100101
s	115	01110011
t	116	01110100
i	105	01101001
n	110	01101110
g	103	01100111
		76543210

Suchen von t

Digitale Such-Tries: Diskussion

Vorteil:

- nur am Ende muss maximal ein Schlüssel verglichen werden
- der Aufbau des Baums ist ***unabhängig*** von der Reihenfolge, in der die Schlüssel eingetragen werden

Nachteil:

- sehr viele innere Knoten, die nur zur Verzweigung dienen
- es gibt zwei unterschiedliche Knotentypen: aufwendig zu implementieren

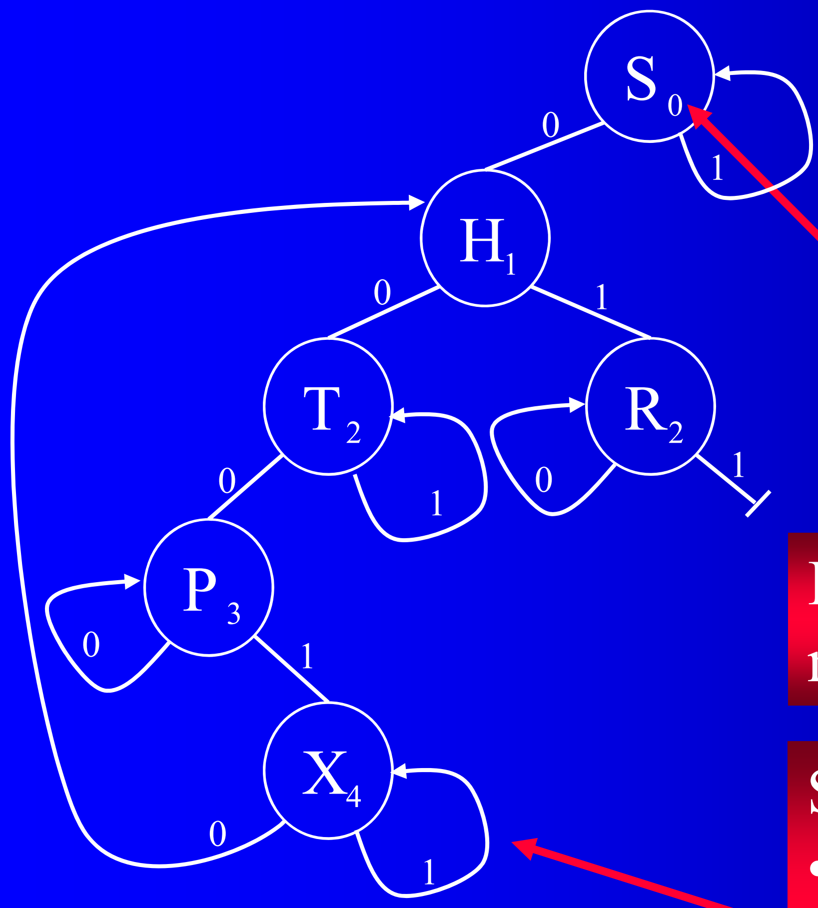
Patricia-Trees

Patricia: ***P***ractical ***A***lgorithm ***T***o ***R***etrieve ***I***nformation
Coded ***I***n ***A***lphanumeric

Idee:

- verwende normalen Digitalbaum, bei dem auch in den inneren Knoten Schlüssel abgespeichert sind
- vergleiche die Schlüssel dennoch erst am Ende
- um an den Blättern auf Schlüssel weiter oben im Baum zu verweisen zu können, führe Rückwärtskanten ein

Patricia-Trees: Beispiel



S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210

Index gibt die Bitposition an, nach der entschieden wird

Suche:
- steige solange ab, bis wieder aufgestiegen wird
- vergleiche dann den Schlüssel

Patricia-Trees: Implementierung

```
class PatriciaTree {
    static boolean left(char key,int bitPos) {
        return (key & (1 << bitPos)) == 0;
    }
    class Node {
        public Node(char key,int bitPos,Node succ) {
            m_Key = key;
            m_BitPos = bitPos;
            boolean bIsLeft = left(key,bitPos);
            m_Left = bIsleft ? this : succ;
            m_Right = bIsLeft ? succ : this;
        }
        public Node(char key,int bitPos) {this(key,bitPos,null);}
        public char m_Key;
        public int m_BitPos;
        public Node m_Left;
        public Node m_Right;
    }
    ...
    private Node m_Root;
}
```

setzt Rück-verkettung

Standardkonstruktor ohne Nachfolger

Knoten merkt sich zusätzlich die Bitposition

Patricia-Trees: Implementierung (Fort.)

- das Suchen erfolgt im wesentlichen im NodeHandler

```
public boolean search(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    return !h.isNull() && h.node(h.NODE).m_Key == c;
}
```

NodeHandler steigt ab, bis Rückwärts- oder Nullverweis gefunden wurde

ist der gefundene Knoten der gesuchte Knoten?

Patricia-Trees: Der NodeHandler

```
class NodeHandler {
      public final int NODE = 0;
      public final int DAD = 1;

      private Object[] m_Nodes = new Object[3];

      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;
      }

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

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

Analog zu RotSchwarz Bäumen: Knoten, Vater und Großvater (siehe später beim Löschen) müssen gemerkt werden

Abstieg

Zugriff auf die Knoten mittels der Konstanten NODE und DAD

Patricia-Trees: Der NodeHandler (Fort.)

```
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) :
                left(n.m_Key,node(kind+1).m_BitPos))
        node(kind+1).m_Left = n;
    else
        node(kind+1).m_Right = n;
    m_Nodes[kind] = n;
}
void search(char c,int maxPos) {
    int lastBitPos = -1;
    while (   !isNull() &&
                lastBitPos < node(NODE).m_BitPos &&
                maxPos > node(NODE).m_BitPos) {
        lastBitPos = node(NODE).m_BitPos;
        down(left(c,lastBitPos));
    }
}
void search(char c) {
    search(c,Integer.MAX_VALUE);
}
```

Analog zu RotSchwarz Bäumen: setzen der Wurzel, wenn es keinen Vater gibt ...

... oder linke bzw. rechts unterhalb des Vaters

Abstieg bis zur maximalen Position (siehe Einfügen) maxPos

Abstieg bis zum Ende

Patricia-Trees: Einfügen

Idee:

- analog zu binären Bäumen: Absteigen und am Ende einfügen
- steige in dem Patricia Tree analog zu der Search Methode ab
- füge den neuen Knoten am Ende ein

3 Fälle sind zu unterscheiden:

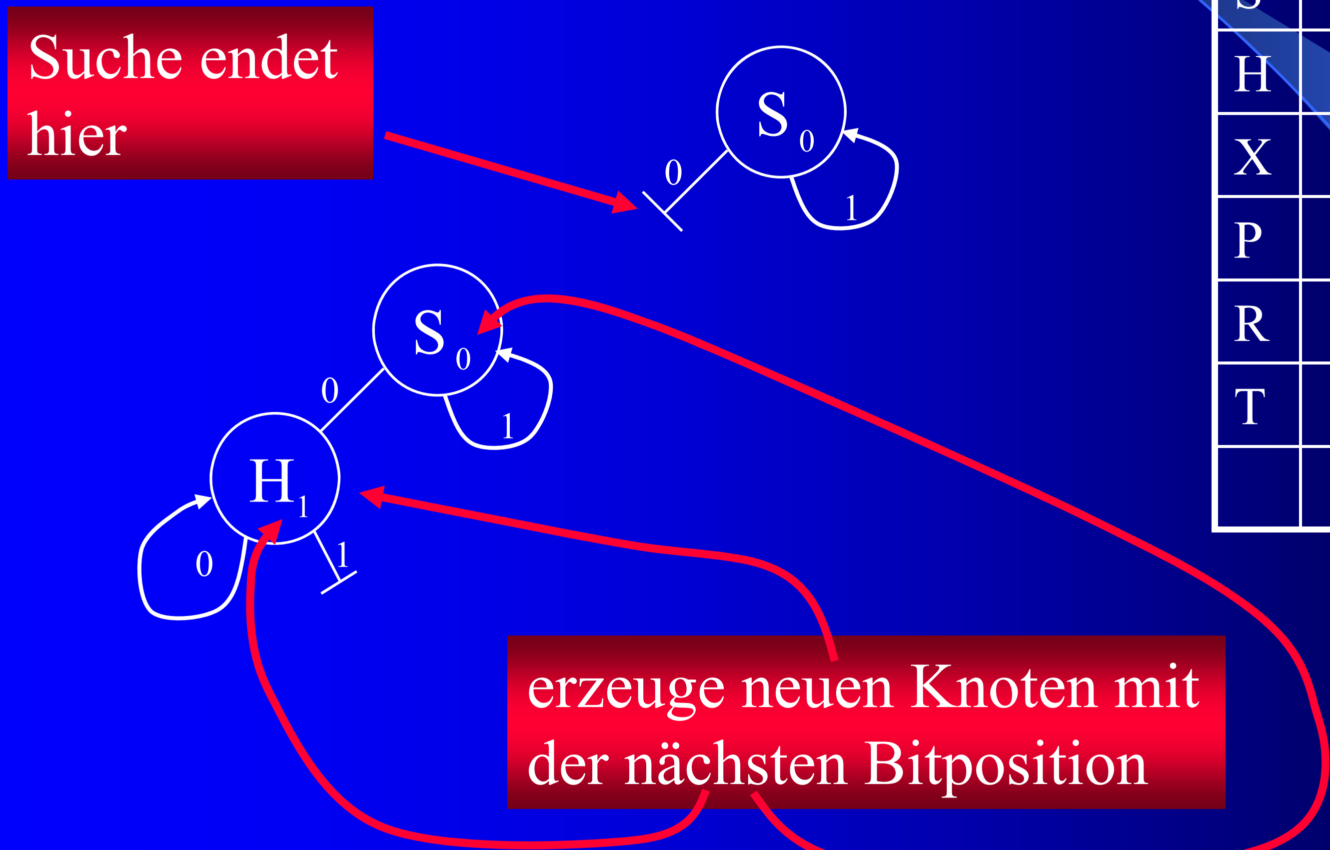
- einzufügender Schlüssel existiert schon: fertig — Fall 1
- Suche endet in einem null-Verweis: neuen Knoten erzeugen — Fall 2
- Suche Ende in einem Knoten mit einem Verweis nach oben in den Baum — Fall 3

Fall 2

Patricia-Trees: Einfügen (Fort.)

- Suche endet in einem null-Verweis: neuen Knoten erzeugen
- H soll eingefügt werden

S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210



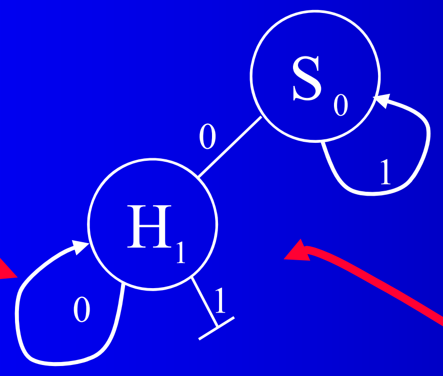
Fall 3 (leicht)

Patricia-Trees: Einfügen (Fort.)

- Suche endet in einem Knoten mit einem Verweis nach oben
- X soll eingefügt werden

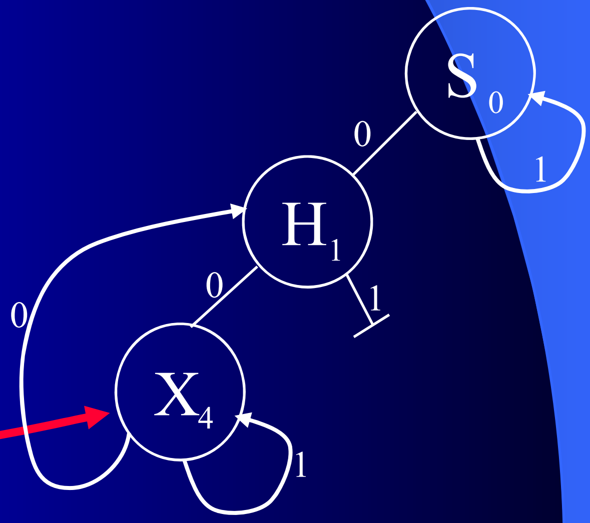
S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210

Suche endet hier



Suche kleinste Bitposition, in der sich H und X unterscheidet: 4

hänge neuen Knoten unterhalb von H mit Bitposition 4 auf

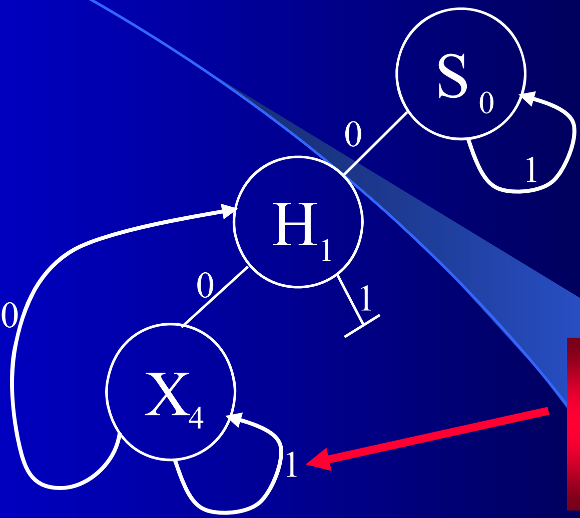


Fall 3 (schwierig)

Patricia-Trees: Einfügen (Fort.)

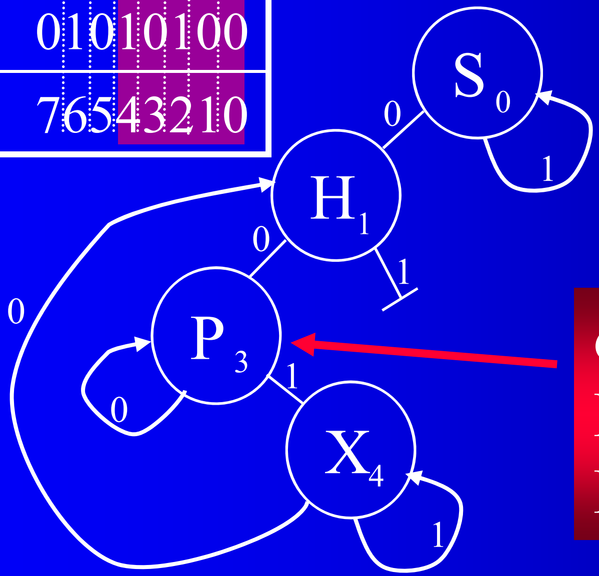
- Suche endet in einem Knoten mit einem Verweis nach oben
- P soll eingefügt werden

S	83	01010011
H	71	01001000
X	88	01011000
P	80	01010000
R	82	01010010
T	84	01010100
		76543210



Suche endet hier

Suche kleinste Bitposition, in der sich X und P unterscheidet: 3

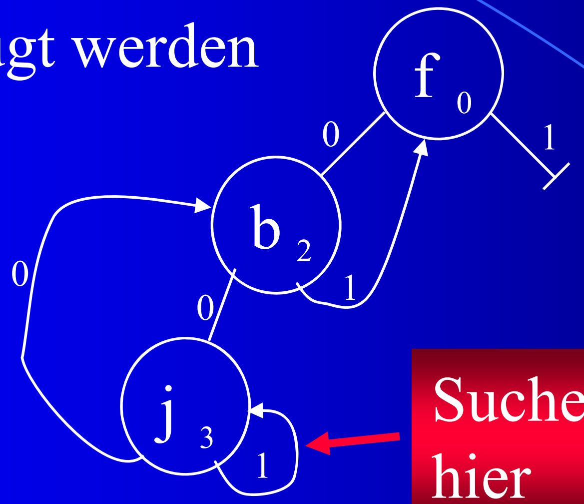


da 3<4 (von X), hänge neuen Knoten oberhalb von X mit Bitposition 3 auf

Fall 3 (noch schwieriger)

Patricia-Trees: Einfügen (Fort.)

- Suche endet in einem Knoten mit einem Verweis nach oben
- h soll eingefügt werden

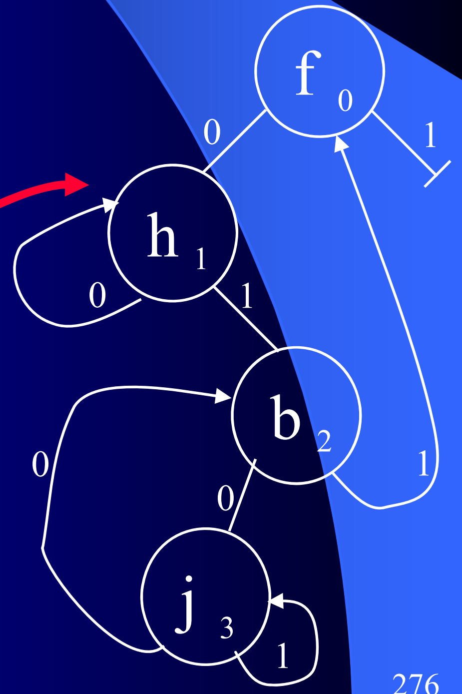


f	102	01100110
h	104	01101000
b	98	01100010
j	106	01101010

Suche endet hier

Suche kleinste Bitposition, in der sich j und h unterscheidet: 1

daher muss h zwischen f und b eingefügt werden. Dazu muss nochmals von oben der Baum durchlaufen werden



Patricia-Trees: Implementierung (Fort.)

```
public boolean insert(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    int index = 0;
    if (h.isNull()) {
        if (h.node(h.DAD) != null) {
            index = h.node(h.DAD).m_BitPos + 1;
        }
    } else if (h.node(h.NODE).m_Key != c) {
        while (left(c,index) == left(h.node(h.NODE).m_Key,index))
            ++index;
    } else {
        // already inserted
        return false;
    }
    h = new NodeHandler(m_Root);
    h.search(c,index);
    h.set(new Node(c,index,h.node(h.NODE)),h.NODE);
    return true;
}
```

1. Abstieg: Suche nach Schlüssel

Fall 2

Fall 3

Kleinste unterschiedliche Bitposition

Fall 1

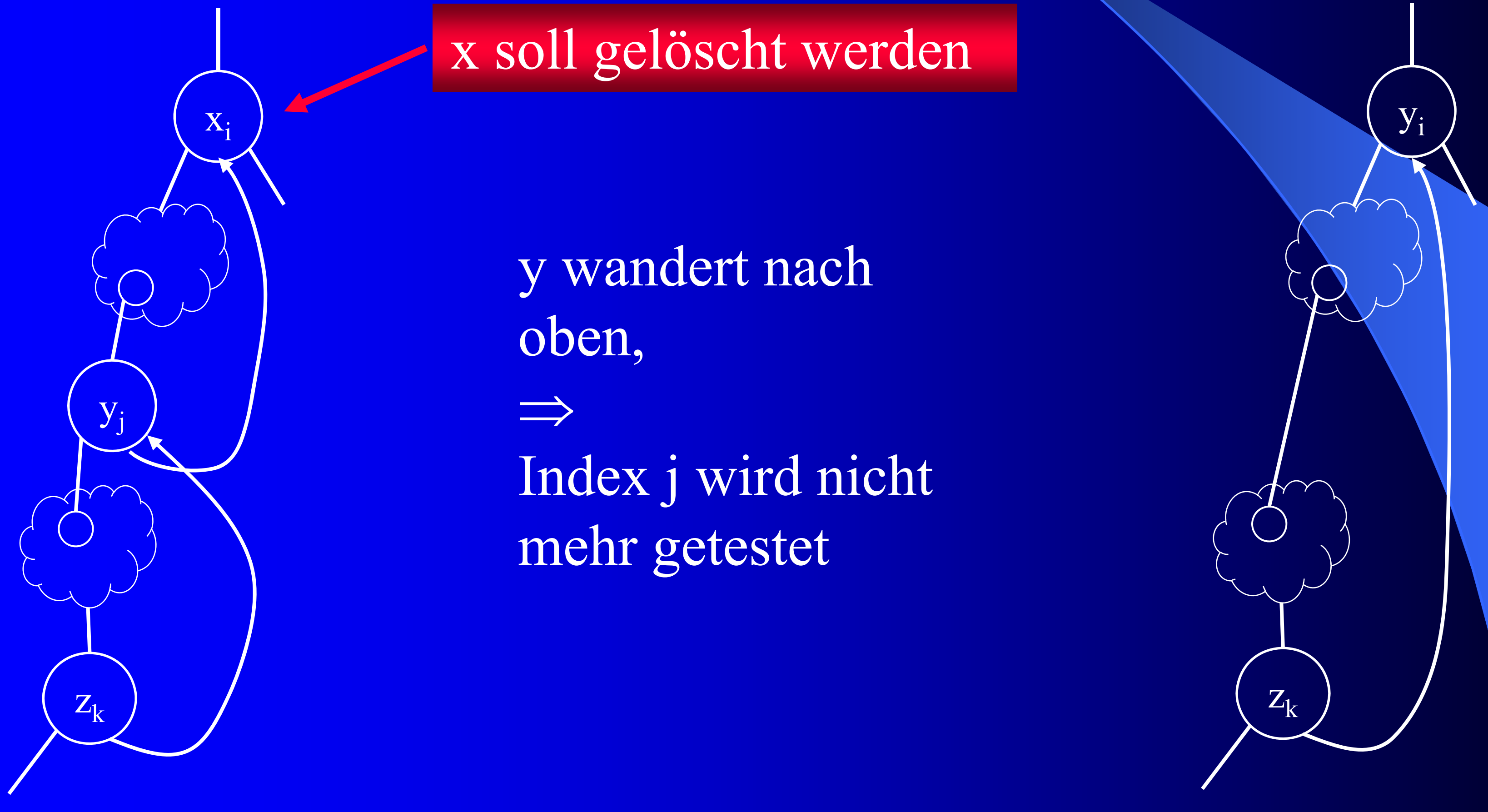
2. Abstieg: Suche nach Einfügeposition ...

... und einfügen

Patricia-Trees: Löschen

- das Löschen erfolgt analog zu Binärbaumen
- das zu löschende Element wird durch das Element ersetzt, das auf das zu löschende Element zeigt

x soll gelöscht werden



y wandert nach oben,

$\Rightarrow$

Index j wird nicht mehr getestet

Patricia-Trees: Implementierung (Fort.)

```
boolean remove(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    if (h.isNull()) {
        return false;
    } else {
        NodeHandler h2 = new NodeHandler(h.node(h.DAD));
        h2.search(h.node(h.DAD).m_Key);
        h.node(h.NODE).m_Key = h.node(h.DAD).m_Key;
        h2.set(h.node(h.NODE),h2.NODE);
        h.set(h.brother(h.NODE),h.DAD);
    }
    return true;
}

class NodeHandler
    ...
    Node brother(int kind) {
        Node dad = node(kind+1);
        Node node = node(kind);
        return dad.m_Left == node ? dad.m_Right : dad.m_Left;
    }
...
```

1. Abstieg: Suche nach Schlüssel

existiert nicht: fertig

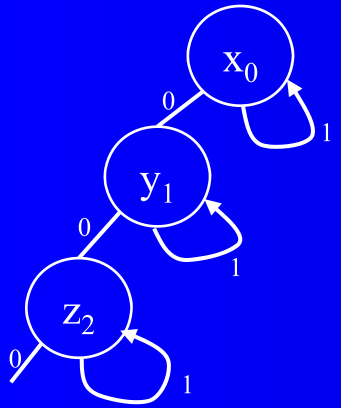
2. Abstieg: Suche nach dem Vater

kopieren des Schlüssels

Löschen des mittleren Knotens

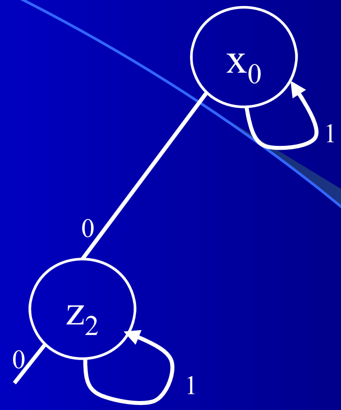
Umhängen des unteren Verweises

Patricia-Trees: Problem nach dem Löschen



löschen von y

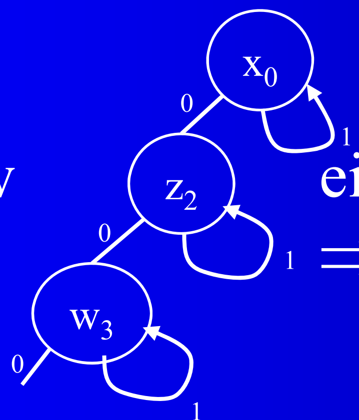
$\Rightarrow$



	3	2	1	0
x	_	_	_	1
y	_	_	1	0
z	_	1	0	0
w	1	0	1	0
u	1	1	1	0

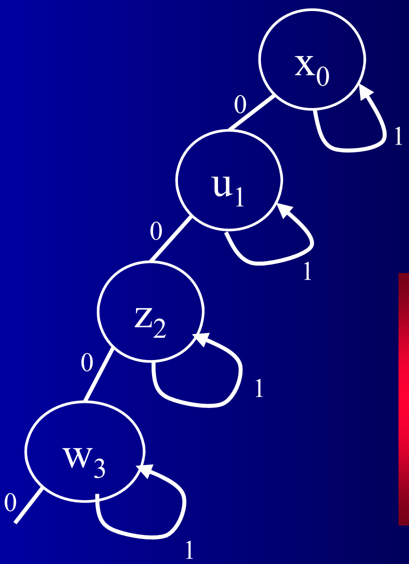
einfügen von w

$\Rightarrow$



einfügen von u

$\Rightarrow$



Fehler: w hätte mit Index 1 eingetragen werden müssen

w ist nicht mehr auffindbar

Patricia-Trees: Lösung für das Löschenproblem

- statt einfach nächsten Index beim "Null" Einfügen ...

```
public boolean insert(char c) {
    NodeHandler h = new NodeHandler(m_Root);
    h.search(c);
    int index = 0;
    if (h.isNull()) {
        if (h.node(h.DAD) != null)
            index = h.node(h.DAD).m_BitPos + 1;
    } else ...
```

- ... mit Vater vergleichen

```
public boolean insert(char c) {
...
    if (h.isNull()) {
        if (h.node(h.DAD) != null) {
            while (   left(c,index) == left(h.node(h.DAD).m_Key,index) &&
                      index < h.node(h.DAD).m_BitPos)
                ++index;
            if (index == h.node(h.DAD).m_BitPos)
                ++index;
        }
    } else ...
```

kleinster Index, in dem sich Vaterschlüssel und c unterscheiden

sonst nächster

Vorlesung 13

Darstellung boolescher Funktionen

- die folgenden Seiten basieren auf:
  - Efficient implementation of a BDD package; Brace, Rudell, Bryant; Proceeding, DAC '90 Proceedings of the 27th ACM/IEEE Design Automation Conference
- Ziel ist die effiziente Darstellung und Bearbeitung von booleschen Funktionen über viele boolesche Variablen (mehrere hundert bis über tausend Variablen)
- dies wird häufig im Bereich der Hardwareentwicklung (Testen und Verifikation) verwendet

Darstellung boolescher Funktionen (Fort.)

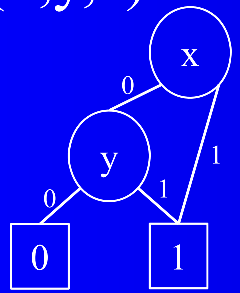
- die allgemeine Frage lautet immer:
  - ist eine boolesche Funktion f erfüllbar (SAT Problem)
  - Beispiel: $f(x,y,z) = (x \vee y) \wedge (\bar{z} \vee \bar{y})$
  - Frage: gibt es eine boolesche Belegung für x,y,z so dass f wahr wird?
- das SAT Problem ist ein NP-vollständiges Problem
- i.a. wird somit dieses Problem nicht effizient lösbar sein, solange
- P=NP Problem nicht gelöst ist
- (und nie lösbar sein, wenn $P \neq NP$ sein sollte)

Boolesche Entscheidungsdiagramme

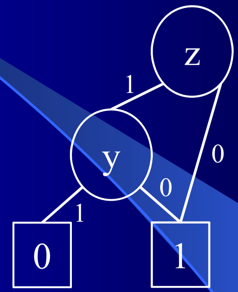
- das Problem mit der Darstellung boolescher Funktionen ist:
  1. werden sie effizient dargestellt, ist das SAT Problem schwer zu entscheiden
  2. ist das SAT Problem leicht zu entscheiden, ist die Darstellung i.d.R. exponentiell (z.B. disjunktive Normalform)
- boolesche Entscheidungsdiagramm (binary decision diagrams = BDDs) sind wie binäre Baume mit Sharing (gerichtete azyklische binäre Bäume)
- in den Knoten stehen die booleschen Variablen
- es gibt zwei Blätter: true und false

Boolesche Entscheidungsdiagramme (BDDs): Beispiel

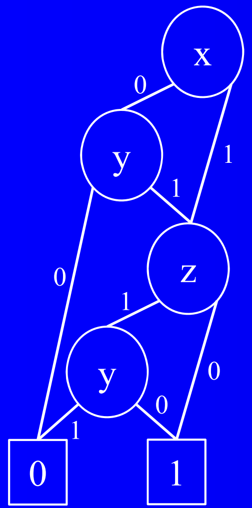
$g(x,y,z) = x \vee y$



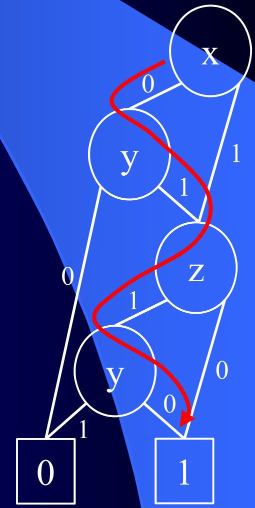
$h(x,y,z) = \bar{z} \vee \bar{y}$



$f(x,y,z) = g(x,y,z) \wedge h(x,y,z)$



- Entscheidungsdiagramm ist recht kompakte Darstellung
- die Op. $\vee$ und $\wedge$ und $\bar{\ }$ lassen sich effizient berechne (darstellen)
- Problem: Erfüllbarkeit ist nicht direkt sichtbar, da Pfade widersprüchlich sein können

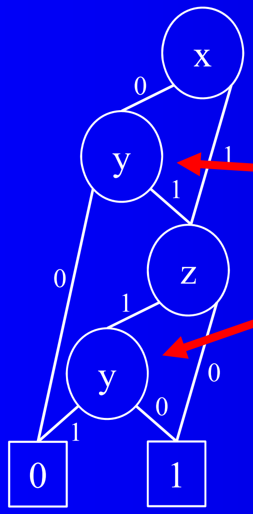


y soll wahr und falsch sein

Reduced Ordered BDDs (RoBDDs)

- Erweiterung der BDDs:

1. Variablen werden gemäß einer beliebigen aber fixen Ordnung getestet (ordered)
    - Folge:
        - keine Variable wird zweimal getestet
        - es kann keine Widersprüche geben



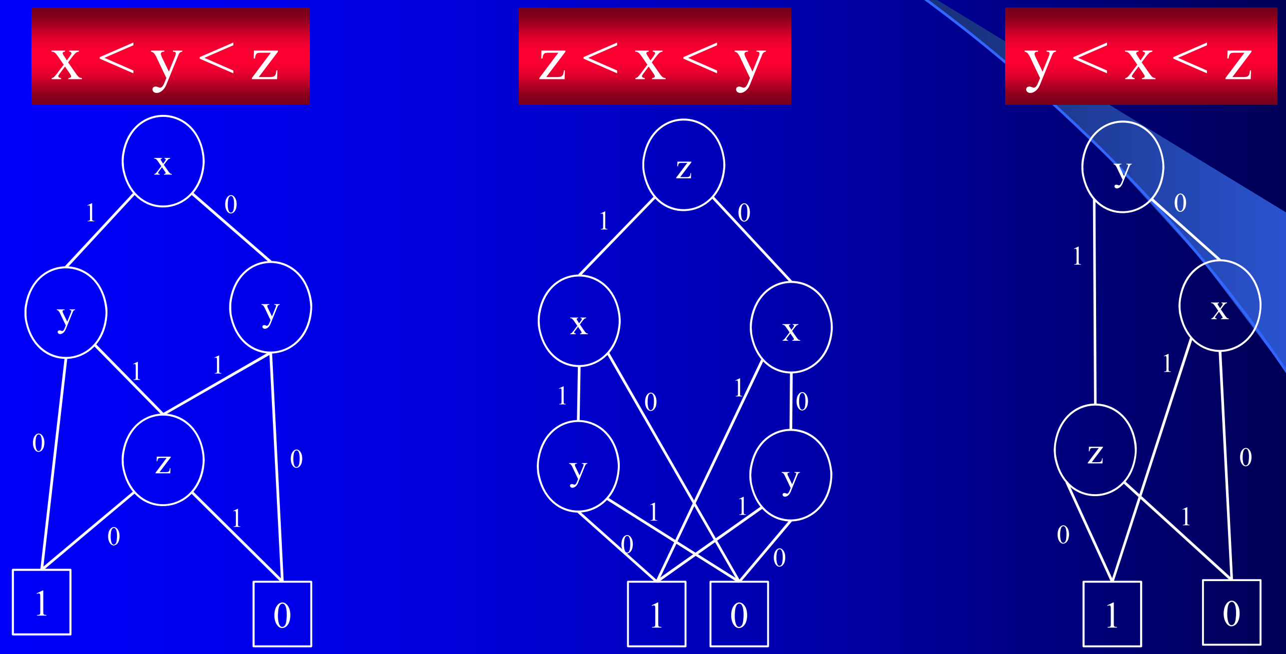
verboten in RoBDDs:
y darf nicht zweimal getestet werde

Reduced Ordered BDDs (RoBDDs) (Fort.)

2. jede Funktion wird nur einmal dargestellt, mehrfache Verwendung wird durch Sharing im Diagramm/Graphen realisiert (reduced)
   - Folge:
     - Funktionen haben eine eindeutige (=kanonische) Darstellung
     - Test auf Identität (und damit SAT) ist in konstanter Zeit machbar (!!!)
3. Folge davon:
   - die Op. $\vee$ und $\wedge$ und $\bar{\ }$ lassen sich nicht mehr trivial berechnen
   - die Darstellung muss i.A. exponentiell sein (ansonsten wäre P=NP)

Beispiele für RoBDDs

- korrekter RoBDDs für $(x \vee y) \wedge (\bar{z} \vee \bar{y})$ mit unterschiedlichen Variablenordnungen



- Wichtig: Variablenordnungen haben massiven Einfluss auf die Darstellungsgröße

Reduced Ordered BDDs (RoBDDs) (Fort.)

- Beobachtung: jede zweistellige boolesche Funktion kann durch if-then-else (ite-Operator) dargestellt werden
- Folge: es reicht, den es eine effiziente Implementierung für den ite-Operator gibt
- Seien f und g boolesche Funktionen, dann gilt:

$$f \wedge g = ite(f,g,0) = f \wedge g \vee \overline{f} \wedge 0$$

$$f \vee g = ite(f,1,g) = f \wedge 1 \vee \overline{f} \wedge g$$

$$\overline{f} = ite(f,0,1) = f \wedge 0 \vee \overline{f} \wedge 1$$

$$f \Rightarrow g = ite(f,g,1) = f \wedge g \vee \overline{f} \wedge 1$$

...

Co-Faktoren

- sei $f(x_1,\ldots,x_n)$ eine boolesche Funktion über n boolesche Variablen x_1 bis x_n
- mit $f_{x_i}(x_1,\ldots,x_{i-1,}x_{i-1,\ldots,}x_n) = f(x_1,\ldots,x_{i-1,}1,x_{i-1,\ldots,}x_n)$ wird der <u>positive</u> Co-Faktor von f bzgl. x_i gezeichnet
- mit $f_{\bar{x}_i}(x_1,\ldots,x_{i-1,}x_{i-1,\ldots,}x_n) = f(x_1,\ldots,x_{i-1,}0,x_{i-1,\ldots,}x_n)$ wird der <u>negative</u> Co-Faktor von f bzgl. x_i gezeichnet
- eine Funktion f lässt sich darstellen als

$$f = x_i \wedge f_{x_i} \vee \overline{x_i} \wedge f_{\bar{x}_i} = ite(x_i, f_{x_i}, f_{\bar{x}_i})$$

Definition des ite-Operators

- sei x die kleinste Variable (gemäß der Variablenordnung) in den Funktionen f, g und h
- dann gilt:

$$
\begin{aligned}
ite(f,g,h) &= f \wedge g \vee \bar{f} \wedge h \\
&= (x \wedge (f \wedge g \vee \bar{f} \wedge h)_x) \vee (\bar{x} \wedge (f \wedge g \vee \bar{f} \wedge h)_{\bar{x}}) \\
&= (x \wedge (f_x \wedge g_x \vee \bar{f}_x \wedge h_x)) \vee (\bar{x} \wedge (f_{\bar{x}} \wedge g_{\bar{x}} \vee \bar{f}_{\bar{x}} \wedge h_{\bar{x}})) \\
&= (x \wedge ite(f_x,g_x,h_x)) \vee (\bar{x} \wedge ite(f_{\bar{x}},g_{\bar{x}},h_{\bar{x}})) \\
&= ite(x, ite(f_x,g_x,h_x), ite(f_{\bar{x}},g_{\bar{x}},h_{\bar{x}}))
\end{aligned}
$$

- damit ist der ite-Operator rekursiv über die Co-Faktoren definiert

Definition des ite-Operators (Forts.)

- neben der rekursiven Definition des ite-Operators fehlt noch die Rekursionsverankerung
- hier gibt es drei Fälle des Rekursionsabbruchs

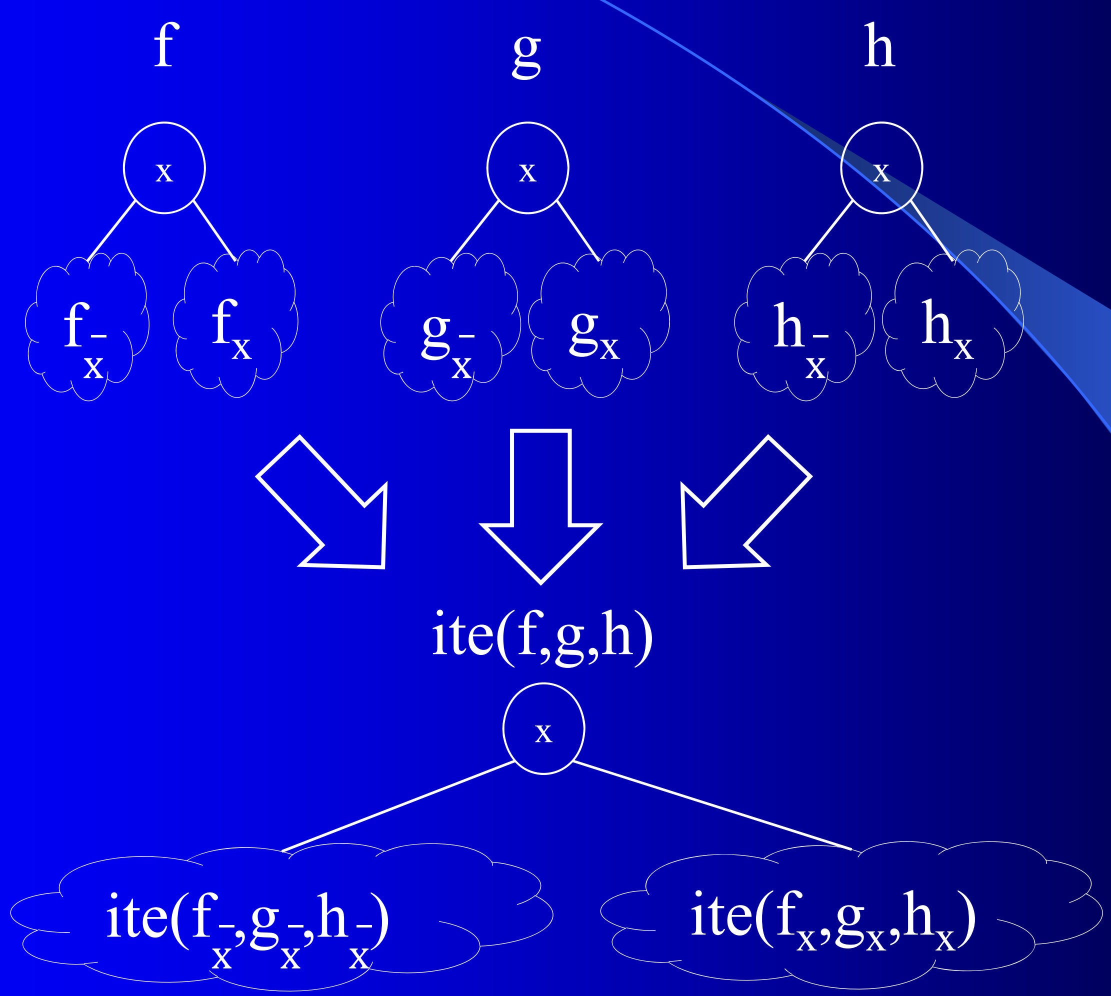
$$ite(1,g,h) = g$$

$$ite(0,g,h) = h$$

$$ite(f,1,0) = f$$

$$ite(f,g,g) = g$$

Definition des ite-Operators: Visualisierung



RoBDDs: Implementierung

- es gibt zwei Knotenarten
  1. die beiden Terminalknoten true (1) und false (0)
  2. interne Knoten mit einer Variablen und zwei Nachfolgern
- Variablen werden durch Zahlen beginnend von 0 dargestellt
- die beiden Konstanten true und false werden durch die maximale Integerzahl (true) bzw. maximale-1 Integerzahl (false) dargestellt

```
class Func {
    private static final int TRUE = 0x7fffffff;
    private static final int FALSE = TRUE-1;

    private final int m_ciVar;
    private final Func m_cThen,m_cElse;
...
```

die beiden Konstanten true und false als Klassenkonstanten

die drei Objektvariablen für die Variable und die beiden Cofaktoren

RoBDDs: Implementierung (Fort.)

```
class Func {
...
    Func(boolean b) {
        m_ciVar = b ? TRUE : FALSE;
        m_cThen = m_cElse = null;
    }

    Func(int iVar,Func t,Func e) {
        m_ciVar = iVar;
        m_cThen = t;
        m_cElse = e;
    }

    Func getThen(int iVar) {    return iVar == m_ciVar ? m_cThen : this;}
    Func getElse(int iVar) {    return iVar == m_ciVar ? m_cElse : this; }

    int getVar() {              return m_ciVar;                  }

    boolean isTrue() {          return m_ciVar == TRUE;          }
    boolean isFalse() {         return m_ciVar == FALSE;         }
    boolean isConstant() {      return isTrue() || isFalse();    }
```

Konstruktor für die beiden Konstanten

Konstruktor für eine Funktion mit der Variablen i und den beiden Cofaktoren t (then) und e (else) bzgl. i

Zugriff auf die Cofaktoren

Abfrage auf Konstanz

RoBDDs: Implementierung (Fort.)

```
public class ROBDD {
      private static final class Func ...;

      private final Func m_cTrue = new Func(true);
      private final Func m_cFalse = new Func(false);

      Func genTrue() {      return m_cTrue;          }
      Func genFalse() {     return m_cFalse;         }

      Func ite(Func i,Func t,Func e) {
            if (i.isTrue())
                  return t;
            else if (i.isFalse())
                  return e;
            else if (t.isTrue() && e.isFalse())
                  return i;
            else {
                  final int ciVar = Math.min(Math.min(i.getVar(),t.getVar()),e.getVar());
                  final Func T = ite(i.getThen(ciVar),t.getThen(ciVar),e.getThen(ciVar));
                  final Func E = ite(i.getElse(ciVar),t.getElse(ciVar),e.getElse(ciVar));
                  return new Func(ciVar,T,E);
            }
      }
```

die beiden Konstanten werden einmal zum Anfang erzeugt

Abfrage der ersten drei Rekursionsverankerungen

Rekursionsschritt

Implementierung: Diskussion

- das Problem mit der bisherigen Implementierung
  - sie ist exponentiell, da gemeinsame Teilgraphen immer wieder neu berechnet werden
  - Graphen werden noch nicht geteilt, da berechnete Ergebnisse nicht getestet werden, ob sie bereits berechnet wurden
- Lösung: Hashmaps einführen, die alle bereits berechneten booleschen Funktionen speichern
- diese bildet Triple von `int x Func x Func` auf `Func` ab
- dazu muss diese Triple Klasse implementiert werden

RoBDDs: Implementierung (Fort.)

```
private static final class Triple {
    private final int m_ciVar;
    private final Func m_cThen;
    private final Func m_cElse;

    Triple(int iVar,Func fThen,Func fElse) {
        m_ciVar = iVar;
        m_cThen = fThen;
        m_cElse = fElse;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Triple) {
            Triple arg = (Triple)obj;
            return arg.m_ciVar == m_ciVar
                && arg.m_cThen == m_cThen
                && arg.m_cElse == m_cElse;
        }
        return false;
    }

    public int hashCode() {
        return m_ciVar ^ m_cThen.hashCode() ^ m_cElse.hashCode();
    }
}
```

Konstruktor

zwei Triple sind gleich, wenn die beiden Variablen und die jeweiligen Cofaktoren gleich sind

zum Hashing die Hashfunktion

RoBDDs: Implementierung (Fort.)

```
public class ROBDD {
    private static final class Func ...;
    private static final class Triple ...;
    private final Func m_cTrue,m_cFalse;
    private Hashtable<Triple,Func> m_Unique;

    ROBDD() {
        m_cTrue = new Func(true);
        m_cFalse = new Func(false);
        m_iNextVar = 0;
        m_Unique = new Hashtable<Triple,Func>();
    }

    Func genVar(int i) {
        Triple entry = new Triple(i,genTrue(),genFalse());
        Func res = m_Unique.get(entry);
        if (res == null) {
            res = new Func(i,genTrue(),genFalse());
            m_Unique.put(entry,res);
        }
        return res;
    }
...
```

nochmal ROBBD: Func und Triple sind Subklassen

Initialisierungen

Generierung der Funktion f(x) = x, wenn sie noch nicht vorhanden ist, ansonsten Wiederverwendung der bereits alten Funktion

RoBDDs: Implementierung (Fort.)

```
public class ROBDD {
...
    Func ite(Func i,Func t,Func e) {
        if (i.isTrue())
            return t;
        else if (i.isFalse())
            return e;
        else if (t.isTrue() && e.isFalse())
            return i;
        else {
            final int ciVar = Math.min(Math.min(i.getVar(),t.getVar()),e.getVar());
            final Func T = ite(i.getThen(ciVar),t.getThen(ciVar),e.getThen(ciVar));
            final Func E = ite(i.getElse(ciVar),t.getElse(ciVar),e.getElse(ciVar));
            if (T.equals(E))
                return T;
            final Triple entry = new Triple(ciVar,T,E);
            Func res = m_Unique.get(entry);
            if (res == null) {
                res = new Func(ciVar,T,E);
                m_Unique.put(entry,res);
            }
            return res;
        }
    }
```

Abfrage der ersten drei Rekursionsverankerungen

Funktionsgleichheit ist Objektidentität wegen Kanonizität

Abfrage der letzten Rekursionsverankerung

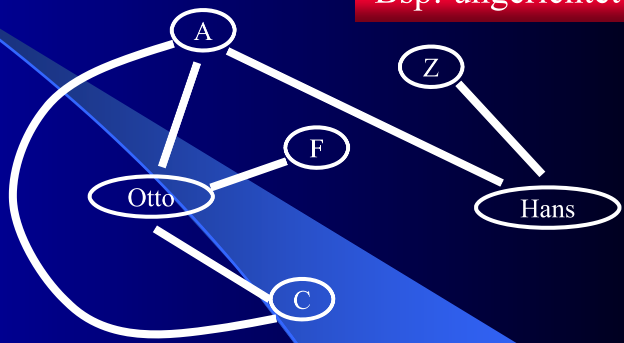
vor Funktionserzeugung wird geschaut, ob diese Funktion bereits existiert

Vorlesung 14

Graphen: Definitionen

- Ein Graph ist ein Paar, bestehend aus einer Menge von Knoten und einer binären Relation über den Knoten

Bsp: ungerichtet



  $G = (V, E)$ mit

  V ist die Menge der Knoten

  $E \subseteq V \times V$ ist die Menge der Kanten

- einen ***Weg*** ist eine Sequenz von Knoten (v1,v2,...,vn) mit:

  $(v_i,v_{i+1}) \in E \ \forall \ i \in \{1,\ldots,n-1\}$

- ein Weg ist ein ***einfacher Weg***, wenn kein Knoten doppelt vorkommt, d.h $\forall\ v_i, v_j : i \neq j \Rightarrow v_i \neq v_j$
- ein Weg $(v_1,v_2,\ldots,v_n)$ ist ein ***Zyklus***, wenn $(v_1,v_2,\ldots,v_{n-1})$ ein einfacher Weg ist und $v_1 = v_n$
- …

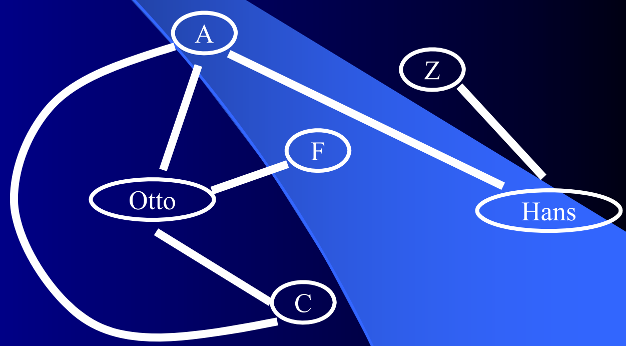
Graphen: Definitionen (Forts.)

- …
- ein Graph heißt zusammenhängend, wenn für 2 beliebige ***unterschiedliche*** Knoten v_i und v_j gilt, dass es einen Weg von v_i zu v_j gibt
- ein zusammenhängender Graph ohne Zyklen heißt ***Baum***
- ein ***Spannbaum*** ist ein Teil des Graphen
  - er enthält alle Knoten
  - er enthält nur $|V|-1$ Kanten, so dass er einen Baum bildet
- Graphen können unterteilt werden in,
  - ***gerichtete*** (Normalfall) und ***ungerichtete*** ($(v_i,v_j) \in E \Rightarrow ((v_j,v_i) \in E)$) Graphen
  - ***gewichtete*** (Kanten mit Informationen: $E \subseteq V \times V \times \mathbb{R}$) und ***ungewichtete*** Graphen

Implementierung

- Implementierung mittels ***Adjazenzmatrizen*** oder ***Adjazenzlisten***
- Adjazenzlisten verwendet man bei ***lichten*** Graphen (Anzahl der Kanten relativ klein)
- Adjazenzmatrizen verwendet man bei ***dichten*** Graphen (Anzahl der Kanten relativ hoch)

	0	1	2	3	4	5
0		x		x	x	
1	x		x	x		
2		x				
3	x	x				
4	x					x
5					x	



- Adjazenzmatrizen (ungewichteter Graph):
  - Nummerierung der Knoten von 0 bis |V|-1
  - 2-dimensionales boolesches Feld ***m*** mit m[i,j]=true $\Leftrightarrow$ $(v_i,v_j) \in E$
- Adjazenzmatrizen (gewichteter Graph):
  - Nummerierung der Knoten von 0 bis |V|-1
  - 2-dimensionales Float (o.ä.) Feld ***m*** mit m[i,j]=f $\Leftrightarrow$ $(v_i,v_j,f) \in E$

Adjazenzmatrizen: Implementierung

```
public class GraphMatrix {

    public GraphMatrix(int iNrOfNodes,boolean bIsDirected) {
        IS_DIRECTED = bIsDirected;
        m_Matrix = new boolean[iNrOfNodes][iNrOfNodes];
        for(int i = 0;i < iNrOfNodes;++i)
            for(int j = 0;j < iNrOfNodes;++j)
                m_Matrix[i][j] = false;
    }

    public void addEdge(int i1,int i2) {
        m_Matrix[i1][i2] = true;
        if (!IS_DIRECTED)
            m_Matrix[i2][i1] = true;
    }
...
    private boolean[][] m_Matrix;
    private final boolean IS_DIRECTED;
...
}
```

merkt sich bei der Instanziierung, ob gerichtet oder ungerichtet

legt ein iNrOfNodes mal iNrOfNodes großes Feld an

fügt eine Kante hinzu; ist es ein ungerichteter Graph, wird eine Rückverkettung eingeführt

Tiefen- und Breitensuche

- bei einer ***Tiefensuche*** wird von ***einem Knoten ausgegangen*** und ***alle Knoten*** besucht, die von diesem Startknoten aus ***erreichbar*** sind
- dabei muss man ***Knoten erkennen***, die man bereits vorher besucht hat (***Zyklen***)
- wird ein neuer Knoten besucht, wird erst sein ***1. Nachfolger komplett*** abgearbeitet, bevor es an den 2. Nachfolger geht usw.
- daher wird ***erst in die Tiefe*** und ***dann in die Breite*** gegangen
  $\Rightarrow$ Tiefensuche
- bei der Breitensuche werden erst alle Söhne bearbeitet, bevor dann alle Enkel und danach alle Urenkel besucht werden
- es wird erst in die Breite, dann in die Tiefe gegangen

  $\Rightarrow$ Breitensuche

Tiefen- und Breitensuche (Implementierung)

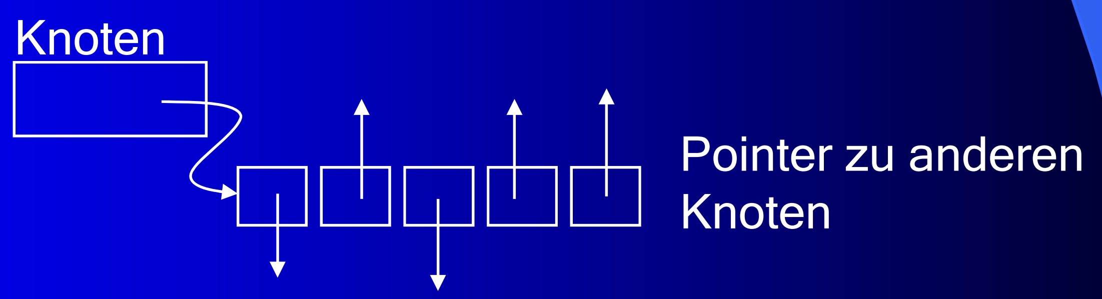
```
public void search(int iNode, boolean bDepthFirst) {
    boolean[] visited = new boolean[m_Matrix.length];
    for(int i = 0;i < m_Matrix.length;++i)
        visited[i] = false;

    DoubleList nodes2visit = new DoubleList();
    nodes2visit.push_back(iNode);
    visited[iNode] = true;

    while (!nodes2visit.isEmpty()) {
        final int CURRNODE = bDepthFirst
            ? nodes2visit.remove_back() : nodes2visit.remove_front();
        System.out.println(CURRNODE);
        for(int i = 0;i < m_Matrix.length;++i)
            if (m_Matrix[CURRNODE][i] && !visited[i]) {
                visited[i] = true;
                nodes2visit.push_back(i);
            }
    }
}
```

entscheiden, ob erst in die Tiefe gesucht wird (true) oder erst in die Breite(false)

nehme den letzten bei der Tiefensuche bzw. den ersten bei der Breitensuche

Adjazenzlisten

- bei ***dichten*** Graphen ist eine ***Adjazenzmatrix relativ gut***, da die Matrix hoch ausgelastet ist
- bei einem ***lichten*** Graphen ist eine ***Adjazenzmatrix schlecht***, da die Matrix nicht ausgelastet ist; viele Einträge sind leer
- daher wird hier viel Platz verschwendet
- für solche lichten Graphen ist die Darstellung mittels ***Adjazenzlisten*** deutlich besser
- bei Adjazenzlisten merkt sich jeder Knoten in einer Liste selber, welches seine Nachfolger sind



Adjazenzlisten: Implementierung

```
public class GraphList {
    class Node {
        public List m_Succ = new List();
    }
    public GraphList(boolean bIsBiDirected) {
        IS_BI_DIRECTED = bIsBiDirected;
    }
    public Node newNode() {...}

    public void addEdge(Node from,Node to) {...}

    private List m_Roots = new List();
    private final boolean IS_BI_DIRECTED;
}
```

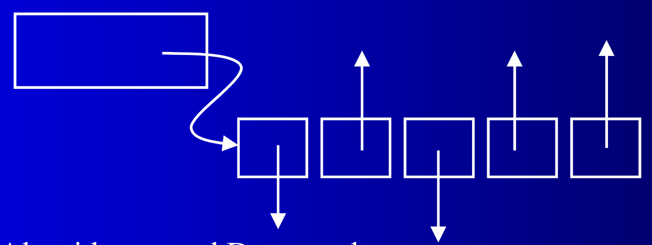
Ein Knoten ist ein Objekt, das eine Liste von Nachfolgerknoten enthält

erzeugt neuen Knoten in m_Roots

speichert to in m_Succ von from

Ein Graph ist nur eine Liste aller seiner Knoten

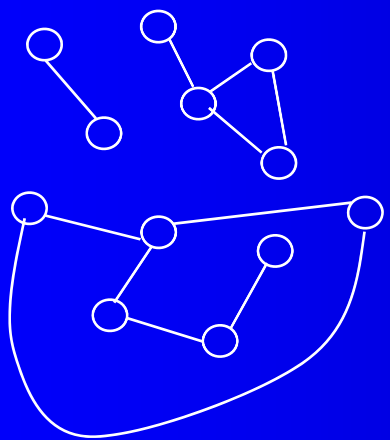
gerichtet oder ungerichtet?



Pointer zu anderen Knoten

Zusammenhang

- ***2 Knoten*** v_i und v_j sind ***zusammenhängend***, wenn es einen ***Weg*** von v_i zu v_j gibt
- eine ***Menge von Knoten*** $V_1 \subseteq V$ heißt ***Zusammenhangs-komponente*** $\Leftrightarrow \forall v_i, v_j \in V_1$: v_i, v_j sind zusammenhängend
- eine ***Zusammenhangskomponente*** $V_1 \subseteq V$ heißt ***maximal*** $\Leftrightarrow \forall\, V' \subseteq V: V_1 \subset V' \Rightarrow V'$ ist nicht zusammenhängend



- dieser Graph besteht aus 3 maximalen Zusammenhangskomponenten
- um maximale Zusammenhangs-komponenten zu finden, kann man den normalen Tiefen- oder Breitendurchlauf verwenden

Zusammenhang: Implementierung

```
public void maxConnectedComponent() {
    boolean[] visited = new boolean[m_Matrix.length];
    for(int i = 0;i < visited.length;++i)
        visited[i] = false;

    int iComp = 0;
    for(int i = 0;i < visited.length;++i) {
        if (!visited[i]) {
            System.out.println("Komponente " + (++iComp));



            search(i,true,visited);
        }
    }
}
...
```

merkt sich alle bereits besuchten Knoten

für alle Knoten ...

... ist der Knoten noch nicht besucht worden, so fängt eine neue Komponente an

WICHTIG! Die Liste der bereits besuchten Knoten muss über einen einzelnen Durchlauf bestehen bleiben

Zusammenhang: Implementierung (Fort.)

```
...
public void search(int iNode, boolean bDepthFirst, boolean[] visited) {
    DoubleList nodes2visit = new DoubleList();
    nodes2visit.push_back(iNode);
    visited[iNode] = true;

    while (!nodes2visit.isEmpty()) {
        final int CURRNODE = bDepthFirst
            ? nodes2visit.remove_back() : nodes2visit.remove_front();
        System.out.println(CURRNODE);
        for(int i = 0;i < m_Matrix.length;++i)
            if (m_Matrix[CURRNODE][i] && !visited[i]) {
                visited[i] = true;
                nodes2visit.push_back(i);
            }
    }
}
```

enthält die bereits besuchten Knoten

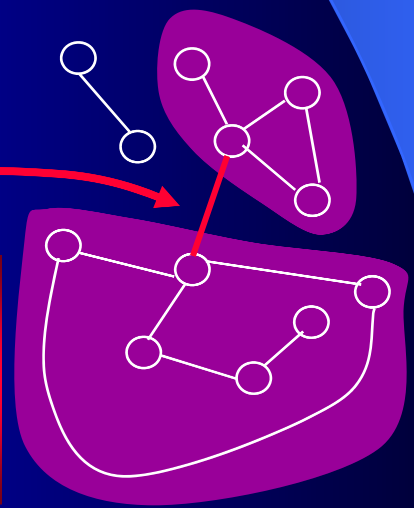
drucke die Nachfolgerknoten immer aus

normaler Tiefen- oder Breitendurchlauf

Anwendung von Zusammenhangskomponenten

- ***Zusammenhangskomponenten*** können dazu verwendet werden, um ***Mengen darzustellen***
- alle ***Knoten***, die zur ***gleichen Zusammenhangskomponente*** gehören, sind ***Elemente der gleichen Menge***
- wird eine ***Kante zwischen 2 Knoten unterschiedlicher Zusammenhangskomponenten*** (sprich Mengen) gezogen, so bilden die beiden Zusammenhangskomponenten jetzt eine Zusammenhangskomponente
- dadurch hat man die ***Mengenvereinigung*** implementiert
- ...

Vereinigung zweier Zusammenhangs-komponenten/Mengen



Anwendung von Zusammenhangskomponenten (Fort.)

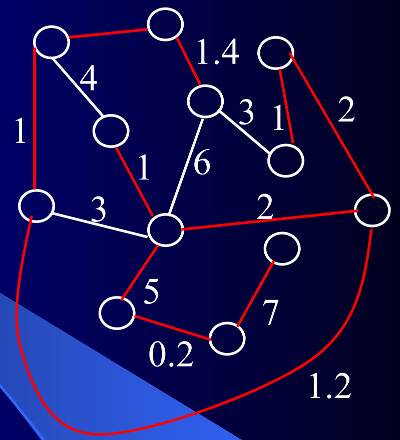
- ...
- versteht man Zusammenhangskomponenten als Mengen, ist eine Standardanwendung, ob ***2 Knoten zur gleichen Menge gehören***
- algorithmisch gesehen ist es die Frage nach einem Weg von dem einen zum anderen Knoten
- diese beiden Fragen werden i.d.R. abwechseln gestellt, d.h. es werden immer wieder Mengen vereinigt und zwischendurch wird abgefragt, ob 2 Knoten zur gleichen Menge gehören
- hierbei dient die Graphstruktur (sprich die Kanten) nur zur Information, welche Elemente zu einer Menge gehören

$A,B \in \mathfrak{R}$

- $A,B \in \mathfrak{R}$?
- $\mathfrak{R} = \mathfrak{R} \cup \mathfrak{I}$
- $A,B \in \mathfrak{R}$?

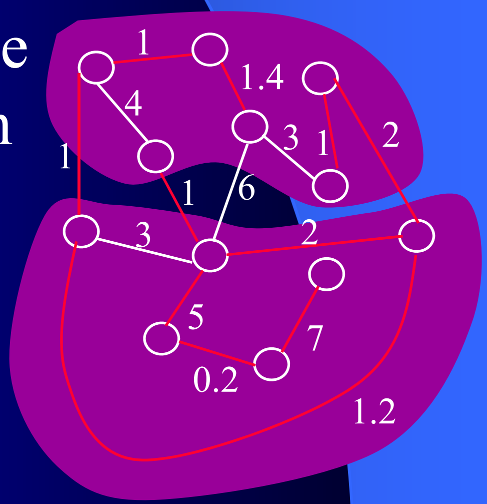
Minimaler Spannbaum

- ein minimaler Spannbaum ist eine Teilmenge der Kanten, so dass
  1. alle Knoten miteinander verbunden sind
  2. die Summe der Kantengewichte wenigstens so klein ist, wie jeder andere Spannbaum



- Eigenschaften minimaler Spannbäume

  Für ***jede***$_1$ gegebene ***Zerlegung*** eines Graphen in *2 Mengen* enthält der minimale Spannbaum die ***kürzeste der Kanten***, die Knoten aus der einen Menge mit der anderen verbindet.



Minimaler Spannbaum: Algorithmus

- basierend auf der Eigenschaft minimaler Spannbäume kann der folgende Algorithmus entwickelt werden
  1. starte bei einem beliebigen Knoten
  2. nehme den Knoten hinzu, der diesem am nächsten liegt
  3. nehme den Knoten, der einem der beiden am nächsten liegt
  4. usw. bis alle Knoten besucht worden sind
- falls bei der Auswahl einmal mehrer Kanten mit geringsten Gewichten gibt, wähle eine zufällig aus

Minimaler Spannbaum: Implementierung

- der Algorithmus ist im ***wesentlichen*** der des ***Tiefen- bzw. Breitendurchlaufs***
- jedoch darf ***nicht der erste oder letzte Knoten***, der noch zu behandelnden Knoten genommen werden, sondern der, der ***den geringsten Abstand*** zu den bereits aufgenommen hat
- folglich benötigt die minimale Spannbaumberechnung eine Prioritätensuche in der Liste
- die Prioritäten sind die Abstände der noch zu untersuchenden Knoten von den bereits untersuchten
- ...

Minimaler Spannbaum: Implementierung (Fort.)

- ...
- folglich benötigt man eine ***Datenstruktur***, in die man ein ***Element mit einem Schlüssel einfügen*** kann und
- die einem ***schnell das Element mit dem kleinsten Schlüssel zurückgeben*** (und entfernen) kann
- den Schlüssel zu einem ***bereits eingefügten Element*** eventuell auf einen ***kleineren Schlüssel ändern*** kann
- Lösung: Heap (siehe Vorlesung 5) oder Prioritätenliste
- hier:
  - Implementierung für Adjazenzmatrix
  - die Abstände sind float Werte

Prioritätenliste

```
public class PrioList {

    public PrioList(int iNrOfElem) {
        m_iNrOfEntries = 0;
        m_Keys = new Float[iNrOfElem];
    }

    public void insert(int iNode, float key) {...}

    public int remove(float[] key) {...}

    public boolean isEmpty() {...}

    int         m_iNrOfEntries;
    Float[]     m_Keys;
}
```

1.5		1.0	2.0			3.31	
0	1	2	3	4	5	6	7

aktuelle Einträge

Schlüssel

Prioritätenliste (Fort.)

gibt es für iNode noch keinen Eintrag ...

... oder hat der alte Eintrag eine höhere Priorität?

```
public void insert(int iNode, float key) {
    if (m_Keys[iNode]==null || key < m_Keys[iNode]) {
        if (m_Keys[iNode] == null)
            ++m_iNrOfEntries;
        m_Keys[iNode] = key;
    }
}
```

Schlüssel eintragen

```
public boolean isEmpty() {
    return m_iNrOfEntries == 0;
}
```

sind überhaupt Schlüssel eingetragen

Prioritätenliste (Fort.)

```
public int remove(float[] key) {
    for(int i = 0;i < m_Keys.length;++i) {
        if (m_Keys[i] != null) {
            int iMin = i;
            for(int i2 = i+1;i2 < m_Keys.length;++i2) {
                if (m_Keys[i2] != null && m_Keys[i2] < m_Keys[iMin])
                    iMin = i2;
            }
            --m_iNrOfEntries;
            key[0] = m_Keys[iMin];
            m_Keys[iMin] = null;
            return iMin;
        }
    }
    return m_Keys.length;
}
```

Ausgabe, nicht Eingabe

sucht den 1. Eintrag

sucht den minimalen Schlüssel in den restlichen Einträgen

Fehlerfall: remove auf leere Liste

Minimaler Spannbaum: Implementierung (Fort.)

```
class GraphMatrix {

    public GraphMatrix(int iNrOfNodes) {
        m_Matrix = new Float[iNrOfNodes][iNrOfNodes];
    }

    public void addEdge(int i1, int i2, float fWeight) {
        m_Matrix[i1][i2] = new Float(fWeight);
        m_Matrix[i2][i1] = new Float(fWeight);
    }

    private Float[][] m_Matrix;
}
```

ungerichteter, gewichteter Graph

Gewicht der Kante

jede Kante hat eine Rückwärtskante

Adjazenzmatrix

Minimaler Spannbaum: Implementierung (Fort.)

```
public void minimalerSpannBaum() {
    PrioList list = new PrioList(m_Matrix.length);
    boolean[] visited = new boolean[m_Matrix.length];
    for(int i = 0; i < m_Matrix.length; ++i)
        visited[i] = false;
    list.insert(0, 0.0f);
    while (!list.isEmpty()) {
        float[] fDistance = new float[1];
        int iNextNode = list.remove(fDistance);
        visited[iNextNode] = true;
        System.out.println("node" + iNextNode + " with distance " + fDistance[0]);
        for(int i = 0; i < m_Matrix.length; ++i) {
            final Float NEW_DISTANCE = m_Matrix[iNextNode][i];
            if (NEW_DISTANCE != null && !visited[i]) {
                list.insert(i,NEW_DISTANCE);
            }
        }
    }
}
```

Prioritätenliste

merkt sich besuchte Knoten

Initialisierung

dichtester Knoten

trage Knoten mit Abstand neu ein oder führe update durch

Minimaler Spannbaum: Implementierung: Diskussion

- der ***PrioList ersetzt*** die ***Liste*** in der Tiefen- bzw. Breitensuche
- damit wird ***nicht mehr das erste oder letzte Listenelement*** für den nächsten Schritt ausgewählt, sondern
- das ***Element***, das den ***geringsten Abstand*** zu einen der bereits besuchten Knoten hat

Minimaler Spannbaum: Komplexität

- die remove-Methode der Prioritätslist ist linear zu der Anzahl der Einträge (Minimumsuche in unsortierter Liste)
- maximal können alle Knoten in der Prioritätsliste eingetragen sein, also $O(V)$
- für alle Knoten muss diese Operation durchgeführt werden
- für jede Kante muss u.U. die Priorität verändert werden
- somit ist die Komplexität in $O(E + V^2)$

Minimaler Spannbaum: Komplexität (Fort.)

- die Prioritätsliste kann optimiert werden, indem ein Heap eingesetzt wird
- in einem Heap kann in logarithmischer Zeit ein Element
  - eingefügt
  - entfernt und
  - seine Priorität geändert werden
- daraus ergibt sich eine Komplexität von $O((E + V) \log V)$

Die Prioritätssuche bei lichten Graphen ermöglicht die Berechnung des minimalen Spannbaums in $O((E + V) \log V)$ Schritten

Prioritätenheap: Idee

- normaler Heap, der sich zusätzlich zum Schlüssel merkt
  - zu welchem Knoten gehört der Schlüssel
  - zu jedem Knoten, ob und wenn ja, wo er sich im Heap befindet

	0	1	2	3	4	5	6	7
Schlüssel: m_Keys	1.0	3.31	1.5	4.0				
Knoten: m_Entries	4	7	0	2				
Ort im Heap: m_PlaceInHeap	2	8	3	8	0	8	8	1

Prioritätenheap: Implementierung

```
class PrioHeap {

    public PrioHeap(int iNrOfNodes) {
        m_uiNextFree = 0;
        m_Keys = new float[iNrOfNodes];
        m_Entries = new int[iNrOfNodes];
        m_PlaceInHeap = new int[iNrOfNodes];
        for(int i = 0;i < iNrOfNodes;++i) {
                m_PlaceInHeap[i] = iNrOfNodes;
        }
    }

    int      m_iNextFree;
    float[]  m_Keys;
    int[]    m_Entries;
    int[]    m_PlaceInHeap;
}
```

zunächst gibt es keine Einträge im Heap

Prioritätenheap: Implementierung (Fort.)

```
public void insert(int iNode, float key) {
    final int PLACE_IN_HEAP = m_PlaceInHeap[iNode];
    if (PLACE_IN_HEAP != m_Keys.length) {
        // update
        if (m_Keys[PLACE_IN_HEAP] > key) {
            // new, lower priority
            m_Keys[PLACE_IN_HEAP] = key;
            upHeap(PLACE_IN_HEAP);
        }
    } else {
        // new entry
        m_Keys[m_iNextFree] = key;
        m_Entries[m_iNextFree] = iNode;
        m_PlaceInHeap[iNode] = m_iNextFree;
        upHeap(m_iNextFree);
        ++m_iNextFree;
    }
}
```

der Knoten ist bereits im Heap enthalten

soll mit einem kleineren Schlüssel eingetragen werden: eventuell nach oben wandern lassen

neuer Eintrag: am Ende einfügen und nach oben wandern lassen

Prioritätenheap: Implementierung (Fort.)

Ausgabe, nicht Eingabe

```
public int remove(float[] key) {
    key[0] = m_Keys[0];
    final int NODE = m_Entries[0];
    m_PlaceInHeap[NODE] = m_Keys.length;
    m_Keys[0] = m_Keys[--m_iNextFree];
    m_Entries[0] = m_Entries[m_iNextFree];
    m_PlaceInHeap[m_Entries[0]] = 0;
    downHeap(0);
    return NODE;
}

public boolean isEmpty() {
    return m_iNextFree == 0;
}
```

das kleinste Element liegt vorne

das letzte Element an Anfang stellen und nach unten wandern lassen

gibt es überhaupt Einträge?

Prioritätenheap: Implementierung (Fort.)

```
private void upHeap(int iIndex) {
    final float VAL = m_Keys[iIndex];
    final int NODE = m_Entries[iIndex];
    int iFather = (iIndex-1) / 2;
    while (iIndex != 0 && m_Keys[iFather] > VAL) {
        m_Keys[iIndex] = m_Keys[iFather];
        m_Entries[iIndex] = m_Entries[iFather];
        m_PlaceInHeap[m_Entries[iIndex]] = iIndex;
        iIndex = iFather;
        iFather = (iIndex - 1) / 2;
    }
    m_Keys[iIndex] = VAL;
    m_Entries[iIndex] = NODE;
    m_PlaceInHeap[NODE] = iIndex;
}
```

Standard Heapoperation

merken, wo die Einträge hinwandern

Prioritätenheap: Implementierung (Fort.)

```
void downHeap(int iIndex) {
    final float KEY = m_Keys[iIndex];
    final int NODE = m_Entries[iIndex];
    while (iIndex < m_iNextFree / 2) {
        int iSon = 2 * iIndex + 1;
        if (iSon < m_iNextFree-1 && m_Keys[iSon] > m_Keys[iSon+1])
            ++iSon;
        if (KEY <= m_Keys[iSon])
            break;
        m_Keys[iIndex] = m_Keys[iSon];
        m_Entries[iIndex] = m_Entries[iSon];
        m_PlaceInHeap[m_Entries[iIndex]] = iIndex;
        iIndex = iSon;
    }
    m_Keys[iIndex] = KEY;
    m_Entries[iIndex] = NODE;
    m_PlaceInHeap[NODE] = iIndex;
}
```

Standard Heapoperation

merken, wo die Einträge hinwandern

Modifikation von minimaler Spannbaum

- der Algorithmus zur Berechnung des minimalen Spannbaums kann leicht modifiziert werden, um
  - zu einem gegebenen Knoten iNode
  - und einer gegebenen Zahl iNr
  - die iNr dichtesten Knoten von iNode

  auszugeben

Modifikation von minimaler Spannbaum : Implementierung

```
public void getNext(int iNode,int iNr) {
      PrioHeap list = new PrioHeap(m_Matrix.length);
      boolean[] visited = new boolean[m_Matrix.length];
      for(int i = 0; i < m_Matrix.length; ++i)
            visited[i] = false;
      list.insert(iNode, 0.0f);
      int iNrOfFound = 0;
      while (!list.isEmpty() && iNrOfFound <= iNr) {
            float[] fDistance = new float[1];
            int iNextNode = list.remove(fDistance);
            ++iNrOfFound;
            visited[iNextNode] = true;
            System.out.println("node " + iNextNode + " with distance " + fDistance[0]);
            for(int i = 0; i < m_Matrix.length; ++i) {
                  final Float NEW_DISTANCE = m_Matrix[iNextNode][i];
                  if (NEW_DISTANCE != null && !visited[i]) {
                        list.insert(i,NEW_DISTANCE + fDistance[0]);
                  }
            }
      }
}
```

starte bei iNode und zähle die gefundenen Knoten

es zählt der Abstand von iNode

Vorlesung 15

Datenkomprimierung

- Im Gegensatz zu den meisten Algorithmen geht es bei der Datenkomprimierung **nicht** um **Zeitersparnis**, **sondern** um **Platzersparnis**
- Der Zugang zur Datenkomprimierung besteht in der Beobachtung, dass viele Daten sehr viele Redundanzen besitzen
- Ziel: eine möglichst hohe Komprimierung der Daten, die in einer möglichst kurzen Zeit berechnet werden kann
- Beispiel: Texte

Beispiel

- „A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS“
- 01000001 00100000 01010011 01001001 01001101 01010000 01001100 01000101 00100000 01010011 01010100 01010010 01001001 01001110 01000111 00100000 01010100 01001111 00100000 01000010 01000101 00100000 01000101 01001110 01000011 01001111 01000100 01000101 01000100 00100000 01010101 01010011 01001001 01001110 01000111 00100000 01000001 00100000 01001101 01001001 01001110 01001001 01001101 01000001 01001100 00100000 01001110 01010101 01001101 01000010 01000101 01010010 00100000 01001111 01000110 00100000 01000010 01001001 01010100 01010011

‚A‘

‚ ‘

60 Buchstaben

60*8 = 480 Bits

Lauflängenkodierung

- Beobachtung: es kommen viele 0 und 1 hintereinander vor
- Idee: nicht 5 mal 0 schreiben, sondern nur die 5
- Beispiel:

1 mal die ‚0‘

1 mal die ‚1‘

5 mal die ‚0‘

keine Einsparung, da 274 mal 3Bit (zur Codierung der Zahlen 1 bis 6) = 822 Bits sind

- 1 1 5 1 2 1 6 1 1 1 2 2 1 1 2 1 2 1 1 1 2 2 1 1 1 1 1 1 5 1 2 2 3 1 3 1 1 1 2 1 6 1 1 1 2 2 1 1 1
  1 1 1 3 1 1 1 2 1 2 1 2 1 2 1 1 1 2 3 2 1 3 3 2 1 6 1 1 1 1 1 3 1 2 4 2 1 6 1 4 1 2 1 3 1 1 1 2 1
  6 1 3 1 1 1 1 1 2 3 2 1 4 2 1 1 2 4 1 1 3 1 3 1 3 1 1 1 1 1 3 1 4 1 6 1 1 1 1 1 1 1 1 1 1 1 2 2 1
  1 2 1 2 1 1 1 2 3 2 1 3 3 2 1 6 1 5 1 2 1 6 1 2 2 1 1 1 1 2 1 2 1 1 1 2 3 2 1 2 1 2 1 1 1 2 2 1 1
  1 1 5 1 1 1 2 2 4 1 6 1 2 3 2 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 4 1 2 1 3 1 1 1 1 1 1 1 2 1 3 1 6 1 2
  4 1 1 3 2 3 1 6 1 4 1 2 1 2 1 2 1 1 1 1 1 1 1 3 1 1 1 2 2

274 Zahlen

Lauflängenkodierung: Problem

- die Lauflängenkodierung funktioniert nur dann gut, wenn möglichst viele lange Ketten existieren
- dies ist im Allgemeinen nicht gegeben
- daher ist die Lauflängenkodierung nur für Spezialfälle geeignet

Variable Lauflängenkodierung

- bei der normalen Lauflängenkodierung wird jeder Buchstabe mit gleich vielen Bits (7 oder 8) kodiert
- d.h. das ‚y' genauso viel Platz zum speichern braucht wie das ‚e'
- da in der natürlichen (hier: deutschen) Sprache aber das ‚e' viel häufiger als das ‚y' vorkommt, würde es Sinn machen, diese Buchstaben mit unterschiedlich vielen Bits zu codieren
- Beispiel: „Dies ist ein Test" benötigt mit einer 8-Bit Kodierung 17*8 = 136 Bits

Variable Lauflängenkodierung: Beispiel

- würden die Buchstaben aber wie folgt codiert:

‚ ‘ ↔ 110
‚D‘ ↔ 1000
‚T‘ ↔ 1001
‚e‘ ↔ 111
‚i‘ ↔ 00
‚n‘ ↔ 1010
‚s‘ ↔ 01
‚t‘ ↔ 1011

- ergäbe sich folgende Kodierung:

10000011101110000110111101110010101101001111011011

D i e s ‚ ‚ i s t ‚ ‚ e

- hier bräuchte man nur 50 Bits
- eine Einsparung von 63%

Variable Lauflängenkodierung: Eigenschaften

- nicht jede Codierung funktioniert
- Beispiel: 01011 mit der folgenden Kodierung
- Aufgabe: der ursprüngliche Text soll wieder hergestellt werden
- Problem: mehrer Möglichkeiten existieren

‚A‘ $\leftrightarrow$ 0
‚B‘ $\leftrightarrow$ 1
‚C‘ $\leftrightarrow$ 11
‚D‘ $\leftrightarrow$ 01
‚E‘ $\leftrightarrow$ 101

$$\underbrace{0\ 1}_{D}\ \underbrace{0\ 1}_{D}\ \underbrace{1}_{B} \qquad \underbrace{0}_{A}\ \underbrace{1\ 0\ 1}_{E}\ \underbrace{1}_{B}$$

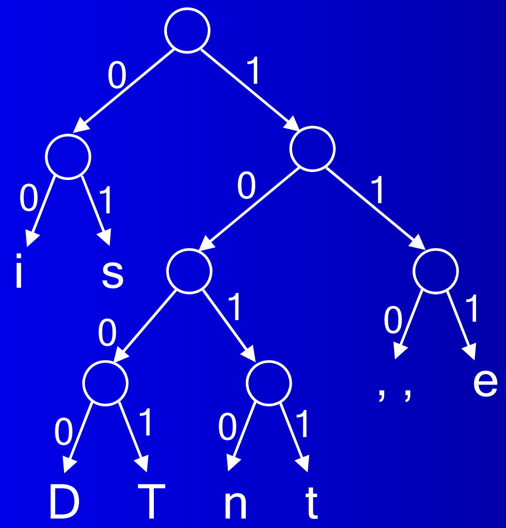
Variable Lauflängenkodierung: Eigenschaften (Fort.)

- das Problem mit dieser Kodierung ist, dass sie nicht ***präfixeindeutig*** ist,
- d.h. die Kodierung mancher Buchstaben sind echte Präfixe von Kodierungen anderer Buchstaben (A ist ein Präfix von D, B ist ein Präfix von C und von E)
- dies hat zur Folge, dass bei der Dekomprimierung nicht entschieden werden kann, ob die Kodierung eines Buchstabens bereits erreicht ist oder ob weitere Bits eingelesen werden müssen

‚A' ↔ 0
‚B' ↔ 1
‚C' ↔ 11
‚D' ↔ 01
‚E' ↔ 101

Variable Lauflängenkodierung: Darstellung

- Frage: wie kann eine solche Kodierung effizient dargestellt werden?
- Antwort: mittels eines Tries (siehe Vorlesung über Patrica Trees)

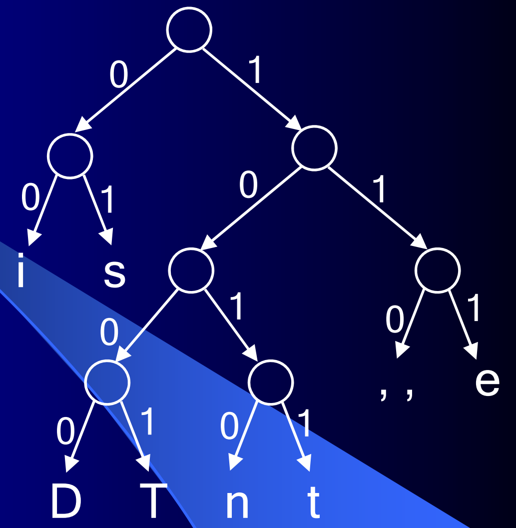


, ‘ ↔ 110
,D‘ ↔ 1000
,T‘ ↔ 1001
,e‘ ↔ 111
,i‘ ↔ 00
,n‘ ↔ 1010
,s‘ ↔ 01
,t‘ ↔ 1011

- Diese Art der Kodierung nennt man Huffman Code nach D. Huffman (1952 entwickelte er diesen Algorithmus)

Variable Lauflängenkodierung: Verwendung

- Dieser Trie kann dazu verwendet werden, die Nachricht zu dekomprimieren
- Dazu steigt man in dem Trie beginnend an der Wurzel entsprechend der 01 Folge hinab
- Wird ein Blatt mit einem Buchstaben erreicht, so wird dieser ausgegeben



Beispiel:

1000001110111000011011110111001010110100111011011

Variable Lauflängenkodierung: Aufbau des Tries

- Beim Aufbau des Tries soll darauf geachtet werden, dass die Buchstaben, die besonders häufig vorkommen, weit oben im Trie stehen, damit ihre Kodierung kurz ist
- Somit muss zunächst die Häufigkeit der Buchstaben im Text ermittelt werden

Ergebnis:

3	1	1	3	3	1	3	2
' '	D	T	i	e	n	s	t

```
public class Huffman {
   private final int MAX = 512;
   private int[] m_Count;

   public void compress(String arg) {
        m_Count = new int[MAX];
        for(int i = 0;i < MAX;++i) {
            m_Count[i] = 0;
        }
        for(int i = 0;i < arg.length();++i) {
            ++m_Count[arg.charAt(i)];
        }
        ...
```

maximal 256 Buchstaben, also 256 Blätter, also maximal 255 innere Knoten: 255+256 = 511

in m_Count steht an der Position i, wie oft der Buchstabe i im Text arg vorkommt

Variable Lauflängenkodierung: Aufbau des Tries (Fort.)

- Der Trie wird von unten nach oben aufgebaut
- Dazu werden jeweils zwei Elemente zu einem neuen Teil-Trie zusammengefasst

3	1	1	3	3	1	3	2
, ,	D	T	i	e	n	s	t

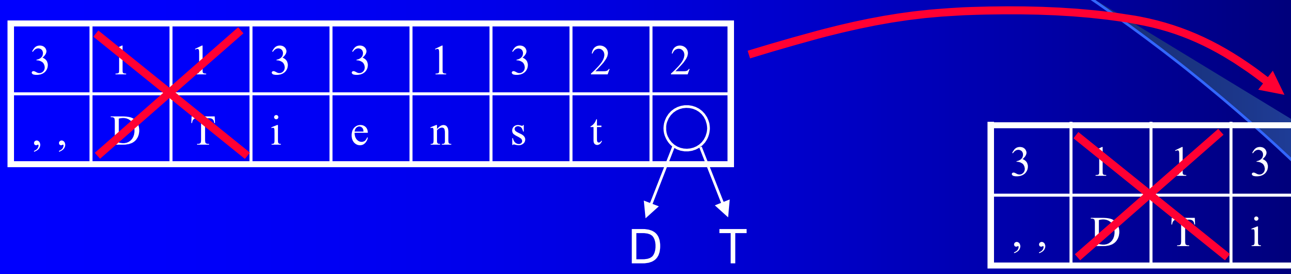
- Es wird bei den Blättern angefangen, die die ***geringsten*** Häufigkeiten haben
- gibt es mehrer, so wird zufällig ausgewählt
- die Häufigkeiten werden addiert und der neue Knoten wird damit annotiert

3	~~1~~	~~1~~	3	3	1	3	2	2
, ,	~~D~~	~~T~~	i	e	n	s	t	

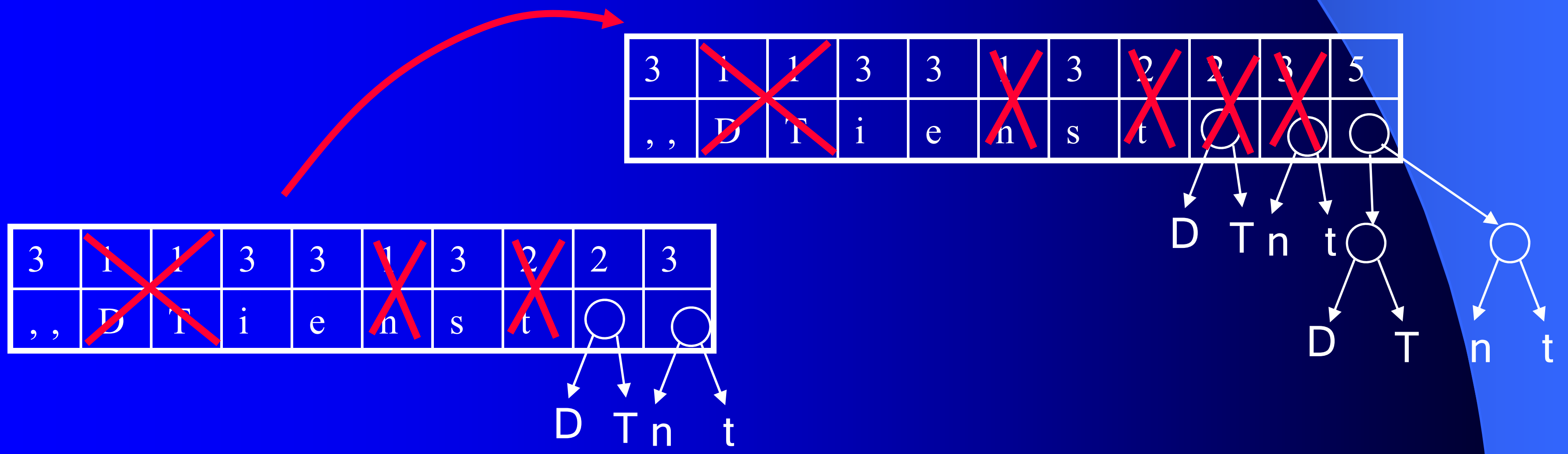
D T

Variable Lauflängenkodierung: Aufbau des Tries (Fort.)

- Dieses Verfahren setzt sich fort



- auch die internen Knoten werden weiter verknüpft



Variable Lauflängenkodierung: Aufbau des Tries (Fort.)

- zum Aufbau dieser Teilbäume braucht man immer die beiden Knoten (Blätter als auch interne Knoten), die die kleinsten Annotierungen haben
- Frage: wie kann man diese schnell finden
- Antwort: Prioritätenheap (siehe letzte Vorlesung)

```
public class Huffman {
  private int[] m_Dad;
  public void compress(String arg) {
      ...
      PrioHeap ph = new PrioHeap(MAX/2);
      for(int i = 0;i < MAX/2;++i) {
        if (m_Count[i] > 0) ph.insert(i,m_Count[i]);
      }
      for(int i = MAX/2;!ph.empty();++i) {
        int s1 = ph.remove();int s2 = ph.remove();
        m_Dad[i] = 0;
        m_Dad[s1] = i;
        m_Dad[s2] = -i;
        m_Count[i] = m_Count[s1] + m_Count[s2];
        if (!ph.empty())
          ph.insert(i,m_Count[i]);
      }
```

WICHTIG: kein PlaceInHeap weil kein update erfolgt

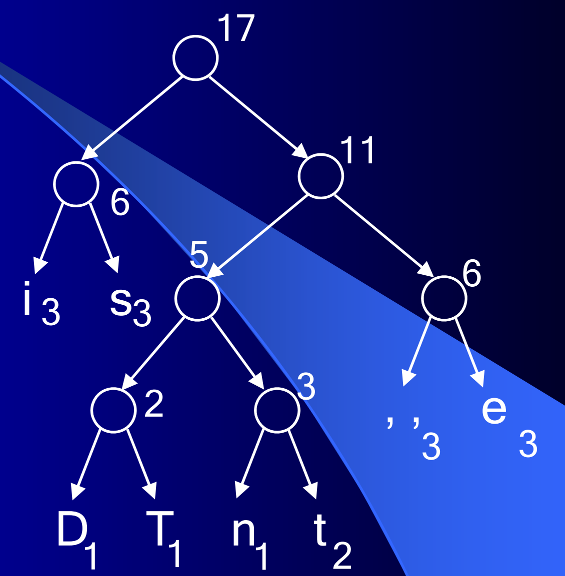
in dem Prioritätenheap stehen zunächst alle Buchstaben-häufigkeiten (=MAX/2)

trage alle Buchstaben ein, die mindestens einmal vorkommen

rechte Söhne speichern den Vater negativ ab

Variable Lauflängenkodierung: Aufbau des Tries (Fort.)

- das Ergebnis ist ein Trie, an dessen Blätter die vorkommenden Buchstaben mit ihrer Häufigkeit stehen
- Beispiel: „Dies ist ein Test“
- Beobachtung: das Gewicht der Wurzel ist die Länge des zu komprimierenden Strings

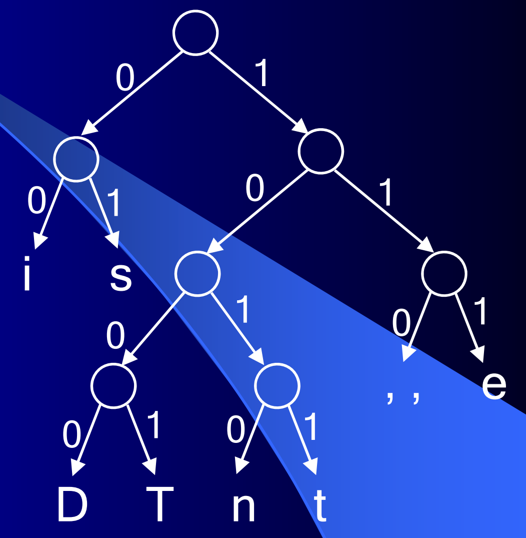


- Ergebnis:

	' '	D	T	e	i	n	s	t							
Index/Ascii	32	68	84	101	105	110	115	116	256	257	258	259	260	261	262
m_Count	3	1	1	3	3	1	3	2	2	3	5	6	6	11	17
m_Dad	260	256	-256	-260	259	257	-259	-257	258	-258	261	262	-261	-262	0

Variable Lauflängenkodierung: Kodierung der Buchstaben

- die vorkommenden Buchstaben können jetzt mit Hilfe des Tries kodiert werden
- jeder Buchstabe bekommt als Codierung die Zahl, die man erhalten würde, wenn man den Trie absteigen würde
- dabei verwendet der Buchstabe nur soviele Bits, wie tief er im Baum steht
- Beispiel: zum T kommt man über den Pfad 1 0 0 1



- 1001 bedeutet 9
- somit bekommt T die Kodierung 9 und als Länge die Tiefe 4
- Beispiel: e hat die Kodierung 111 (also 7) mit einer Länge von 3

Kodierung der Buchstaben: Implementierung

- um die Buchstaben zu Kodieren werden zwei zusätzlich Felder benötigt
- m_Code enthält an der Stelle i den Code des Buchstaben i
- m_Len enthält an der Stelle i, wieviele Bits von der Codierung der Buchstabe i verwenden muss

```
public class Huffman {
   m_Code = new int[MAX/2];
   m_Len = new int[MAX/2];
   public void compress(String arg) {
       ...
       for(int i = 0;i<MAX/2;++i) {
          int len = 0,code = 0;
          if (m_Count[i] > 0) {
             for(int t = m_Dad[i];t != 0;t = m_Dad[t],++len)
                if (t < 0) {
                   code += 1 << len;
                   t = -t;
                }
          }
          m_Code[i] = code;
          m_Len[i] = len;
       }
```

nur für die Buchstaben

kommt der Buchstabe überhaupt vor?

laufe bis zur Wurzel, berechne die Länge und den Codierwert

Kodierung der Buchstaben: Implementierung (Fort.)

- Mit der Kodierung und der Länge können die Buchstaben in einfacher Art und Weise in entsprechende Bitmuster unterschiedlicher Länge ausgegeben werden

für alle Buchstaben ...

```
public void printString(String arg) {
    for(int i = 0;i<arg.length();++i) {
        char c = arg.charAt(i);
        char code = (char)m_Code[c];
        int len = m_Len[c];
        for(int j = 0;j < len;++j) {
            System.out.print((code >> (len-j-1)) & 1);
        }
    }
}
```

... drucke soviele Bits, wie zuvor berechnet

... die Bits richten sich nach dem vorher berechneten Code

- Ergebnis:

1000001110111000011011110111001010110100111011011

Dekomprimierung

- für den Abstieg müssen zusätzlich zu m_Dad die Söhne in m_Left und m_Right Feldern gemerkt werden
- von der Wurzel muss gemäß der einkommenden 0 und 1 in dem Baum nach links bzw. rechts verzweigt werden
- wird ein Blatt erreicht, wird der Buchstabe ausgegeben
- das Verfahren beginnt mit dem nächsten Buchstaben wieder bei der Wurzel

```
public void decode(String arg) {
    for(int i = 0;i < arg.length();) {
        int node = m_Root;
        while (m_Left[node] != -1) {
            node = arg.charAt(i++) == '0' ? m_Left[node] : m_Right[node];
        }
        System.out.print((char)node);
    }
}
```

muss zunächst gemerkt werden: in diesem Beispiel die Nummer 262

Abstieg, bis kein Sohn mehr vorhanden ist

Zusammenfassung

- die Huffman Codierung bietet ein schnelles Verfahren zur guten Komprimierung von Texten
- das vorgestellte Verfahren müsste zu den generierten Text auch noch den Trie selber abspeichern
- dies kann vermieden werden, wenn man von einer Standardverteilung der Buchstaben in der zu verarbeitenden Sprache ausgeht
- dann würde man für z.B. der deutschen Sprache einen Trie aufbauen und danach kodieren
- diese Kodierung müsste sich dann nicht den Trie merken, würde jedoch für Texte der englischen Sprache eventuell nicht optimal

Vorlesung 16

Algorithmen
und
Datenstrukturen

Zusammenfassung

Graphikprogrammierung unter Java

- Images verwalten, die über ImageProducer generiert werden, die Bilder erzeugen
- MemoryImageSource ist ein ImageProducer, der ein Integerfeld mit einem Bild assoziiert
- Zusammenhang von Bildpunkten und Integerwerten:
    - 32 Bit Integerwert kodiert jeweils 8 Bit Farbwerte für Rot, Grün und Blau
- Bilder können ausgelesen werden durch einen PixelGrabber
- hierdurch können Bilder in Integerfelder konvertiert werden
- diese Integerfelder können modifiziert werden und wieder mittels eines MemoryImageSource in Bilder verwandelt werden

Graphikprogrammierung unter Java (Fort.)

- mittels der einfachen Transformation

  Translation, Skalierung, Rotation, x- und y-Scherung

  können komplexe Bildveränderungen durchgeführt werden
- jeder dieser Transformationen kann als Matrixoperation mittels einer 3×3 Matrix und einem erweiterten Punkt Spaltenvektor verstanden werden
- mehrer Transformationen hintereinander können zu einer Operation zusammengefasst werden, indem die Matrizen multipliziert werden

Sortierverfahren (Fort.)

- Shell Sort
- basiert auf dem Insertion Sort
- geeignet für Vectoren
- nicht geeignet für Listen (siehe Übung)
- Laufzeitkomplexität ist nicht klar; hängt stark von der Wahl der Abstände ab
- funktioniert in der Praxis sehr gut und ist leicht zu implementieren

712	452	307	145	102	50	12	7	3	1

712	452	307	145	102	50	12	7	3	1

712	452	307	145	102	50	12	7	3	1

712	452	307	145	102	50	12	7	3	1

Sortierverfahren (Fort.)

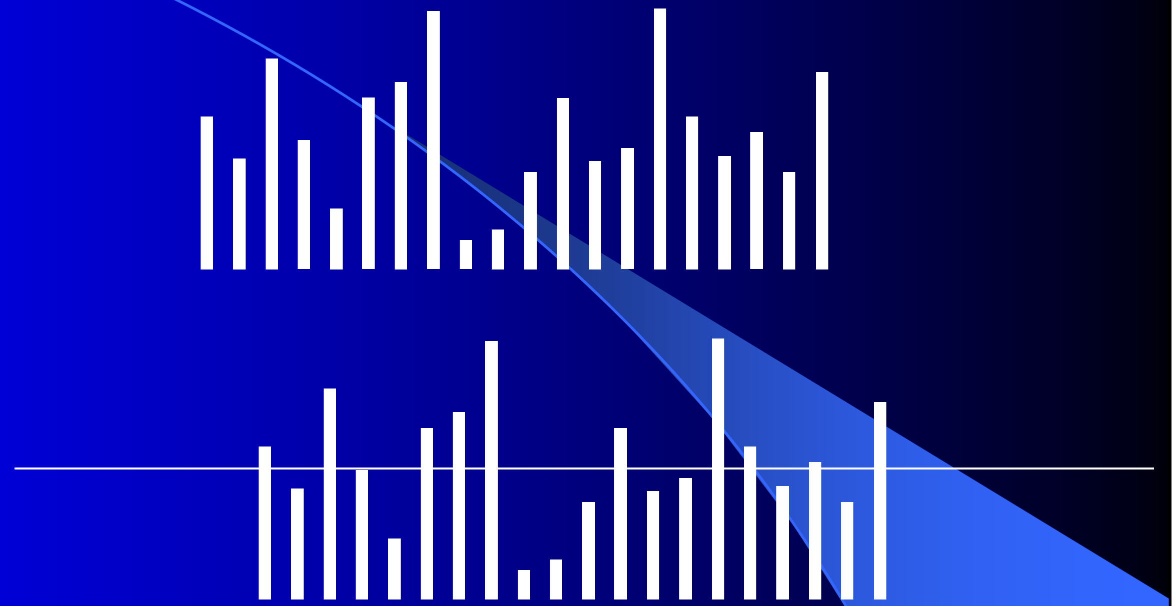
- Distribution Counting
- sortieren von viele Zahlen aus einem kleinen Wertebereich
- geeignet für Vectoren und Listen
- Laufzeitkomplexität ist O(n)
- sollte man immer verwenden, wenn die obige Bedingung erfüllt ist

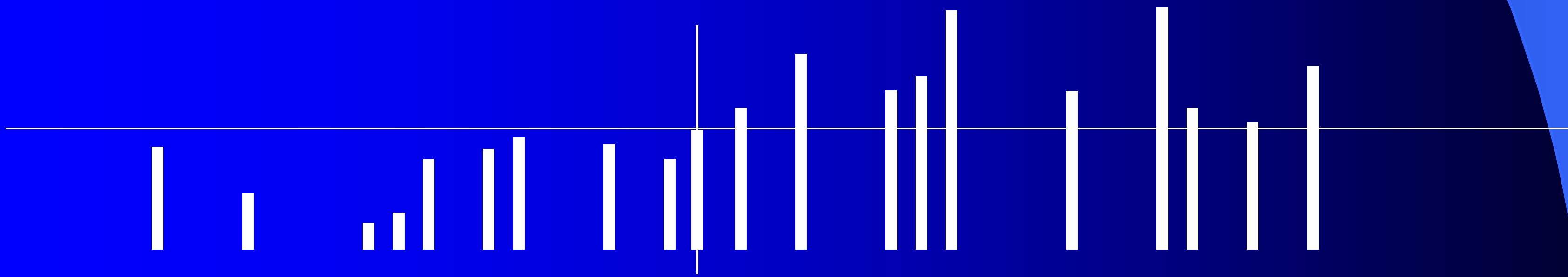
2	3	1	7	5	6	2	7	3	1

0	2	2	2	0	1	1	2
0	1	2	3	4	5	6	7

Sortierverfahren (Fort.)

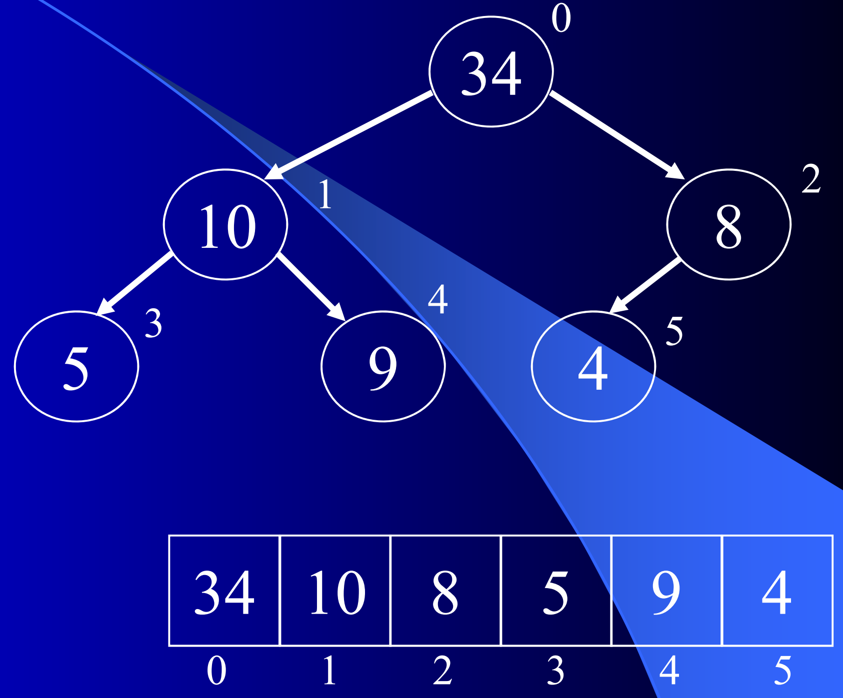
- Quick Sort
- geeignet für Vectoren
- nicht geeignet für Listen
- Laufzeitkomplexität ist im Durchschnitt O(n log n)
- Laufzeitkomplexität ist im Worst Case $O(n^2)$





Sortierverfahren (Fort.)

- Heap Sort
- geeignet für Vectoren
- nicht geeignet für Listen
- Laufzeitkomplexität ist garantiert O(n log n)
- wichtige Datenstruktur auch für Prioritätenauswahl (siehe Minimaler Spannbaum, Datenkodierung „Huffman“)



Sortierverfahren (Fort.)

- Merge Sort
- geeignet für externes Sortieren
- Laufzeitkomplexität ist garantiert O(n log n)
- zusätzlicher Speicherplatzverbrauch von O(n)

1.

200	155
102	40
45	30
23	-17

2.

200	155
102	40
45	30
23	

3.

200	155
102	40
45	30

4.

200	155
102	40
45	

5.

200	155
102	
45	

6.

200	155
102	

Ergebnis:

-17	23	30	40	45

Suchen

- sequentielles Suchen
- geeignet für Listen und Vectoren
- Laufzeitkomplexität ist im Worst Case O(n)
- neue Elemente kann man in O(1) einfügen (am Ende)
- Voraussetzungen: keine

Schlüssel	34	17	-5	40	34	3	-15	13
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre

Suchen (Fort.)

- binäres Suchen
- geeignet für Vectoren
- nicht geeignet für Listen
- Laufzeitkomplexität ist O(log n)
- neue Elemente kann man in O(n) einfügen
- Voraussetzungen: Elemente müssen nach ihrem Schlüssel sortiert sein

2. Vergleich

	0	1	2	3	4	5	6	7
Schlüssel	-15	-5	3	13	17	34	34	40
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre

1. Vergleich

gesucht wird nach 17

Suchen (Fort.)

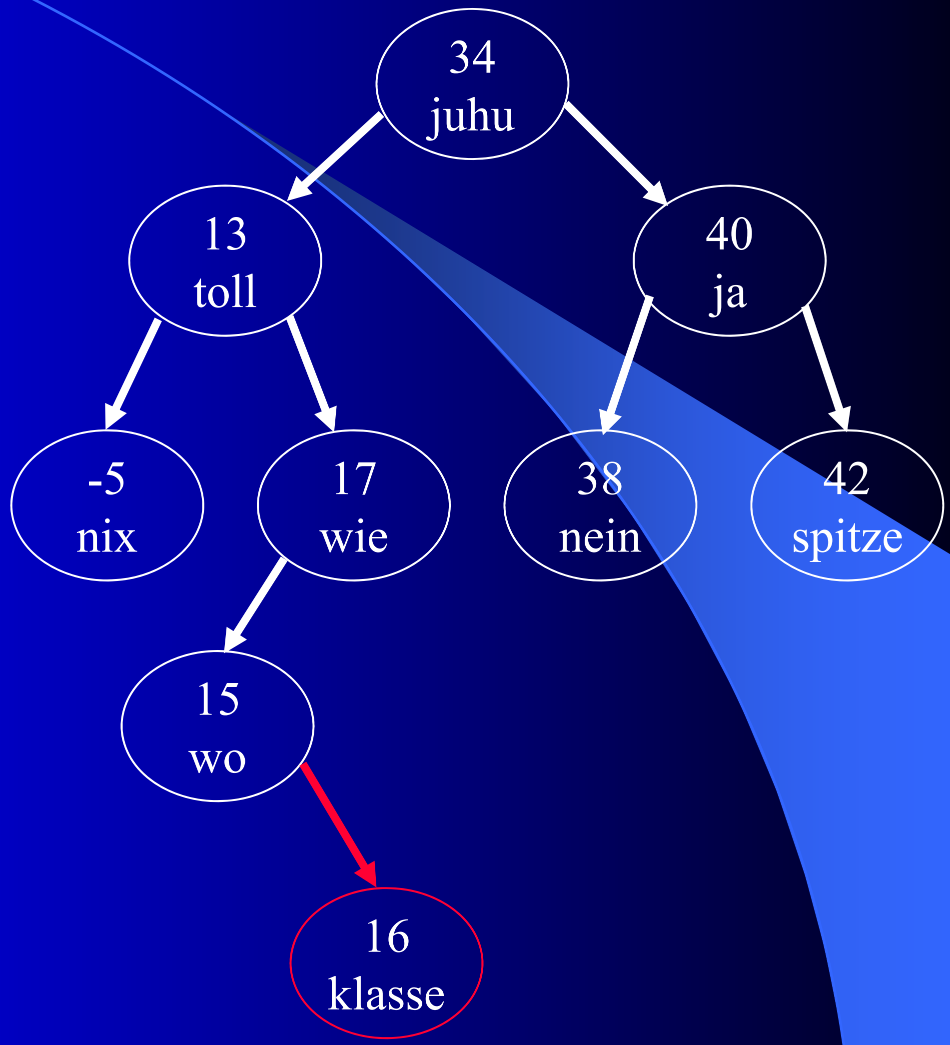
- Interpolationssuchen
- geeignet für Vectoren
- basiert auf der binären Suche
- Laufzeitkomplexität ist im Durchschnitt O(log log n)
- neue Elemente kann man in O(n) einfügen
- Voraussetzungen: Schlüssel sind einigermaßen gleichverteilt

uiL = 0
uiR = 7
keyL = -15
keyR = 40
cuiMiddle = 0 + (-5--15)*(7-0)/(40--15) = 1

Schlüssel	-15	-5	3	13	17	34	34	40
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre
	0	1	2	3	4	5	6	7

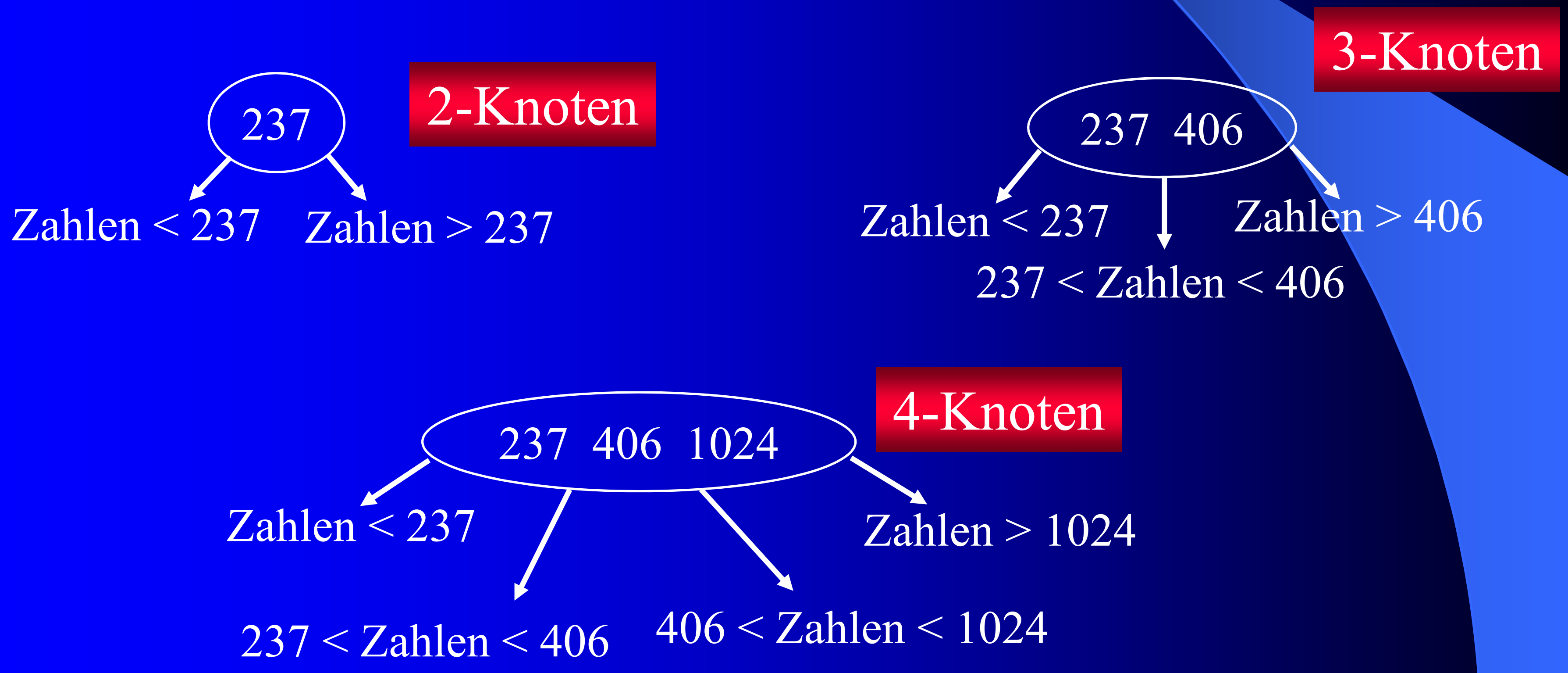
Suchen (Fort.)

- Suchen im binären Baum
- Laufzeitkomplexität ist im Durchschnitt O(log n)
- im Worst Case O(n)
- neue Elemente kann man in O(log n) einfügen
- im Worst Case O(n)
- Vorsicht vor entarteten Bäumen (= Listen)



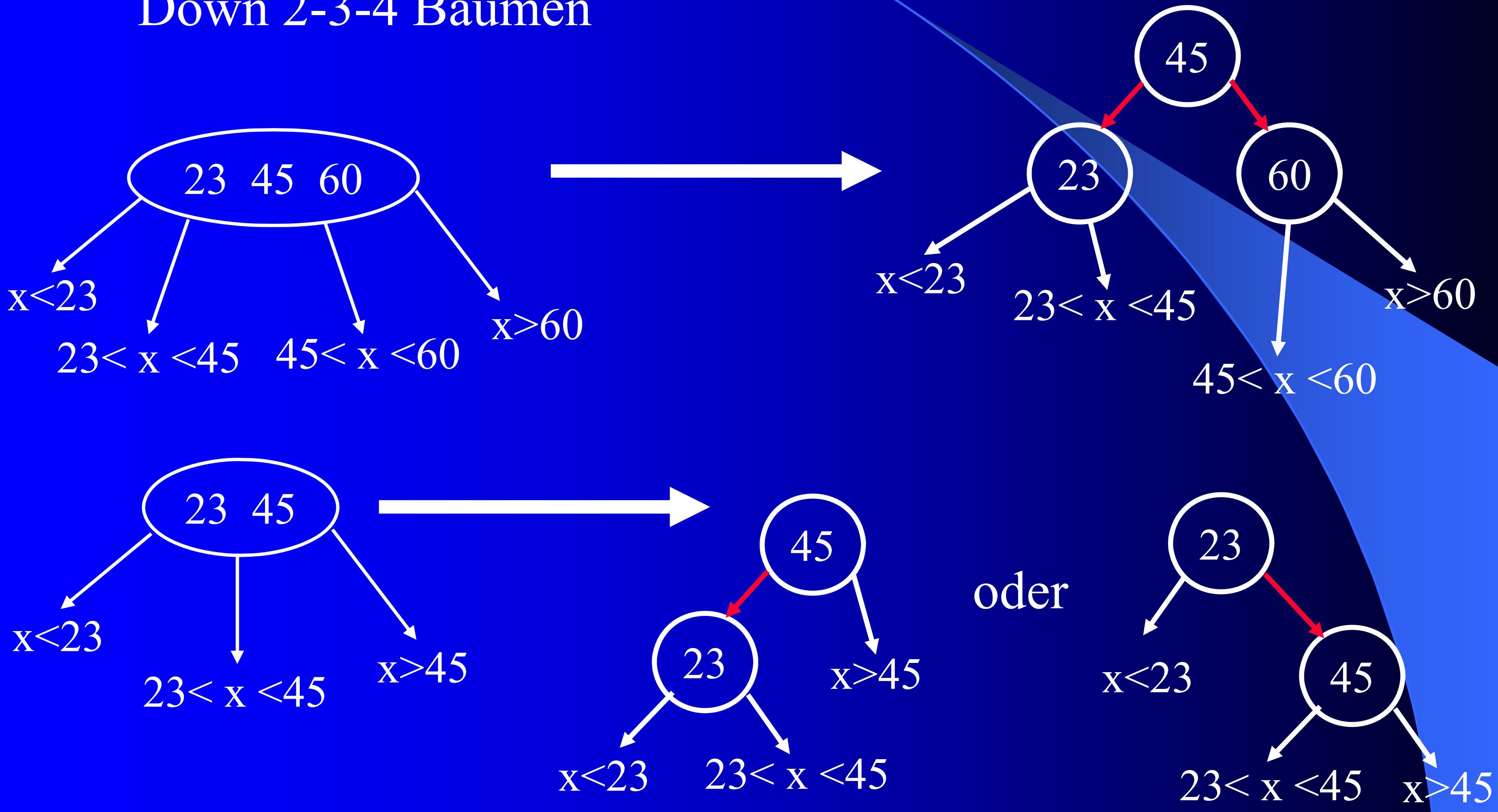
Suchen (Fort.)

- Suchen im Top-Down 2-3-4 Bäumen
- sind immer ausgeglichen
- Laufzeitkomplexität ist immer O(log n) (Suchen & Einfügen)
- theoretische Überlegung: Vorbereitung zu Rot-Schwarz Bäumen



Suchen (Fort.)

- Rot-Schwarz Bäume als binäre Implementierung von Top-Down 2-3-4 Bäumen



Suchen (Fort.)

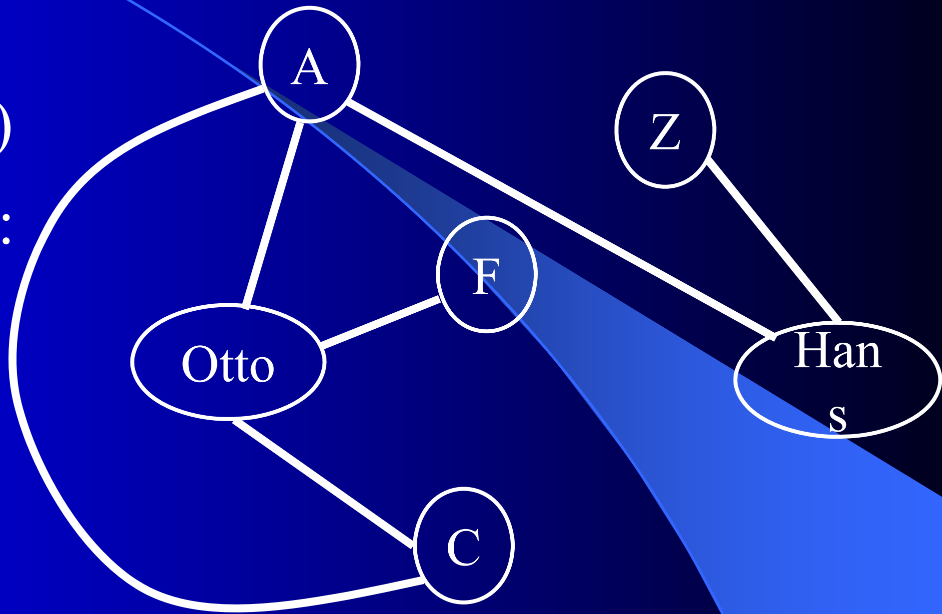
- Hashing
- sehr schnelles Suchverfahren
- Laufzeitkomplexität ist oft O(1) (Suchen & Einfügen)
- Voraussetzung: aus den Schlüsseln lässt sich ein Hashindex leicht und schnell berechnen

Suchen (Fort.)

- Digitales Suchen
- Basisstruktur ist ein binärer Baum
- in den innern Knoten wird nicht nach dem gesamten Schlüssel sondern nach den einzelnen Bits des Schlüssels verzweigt
- Tiefe des Baums ist nicht durch die Anzahl der Schlüssel, sondern durch die Größe der Schlüssel (Anzahl der Bits) bestimmt
- einfache Implementierung
- Patrica-Trees: Optimierung von Digitalen Suchbäumen, um nur einen Schlüsselvergleich am Ende einer Suche durchzuführen
- deutlich komplexere Implementierung
- sehr effizient für endlich große Schlüssel (Strings???)

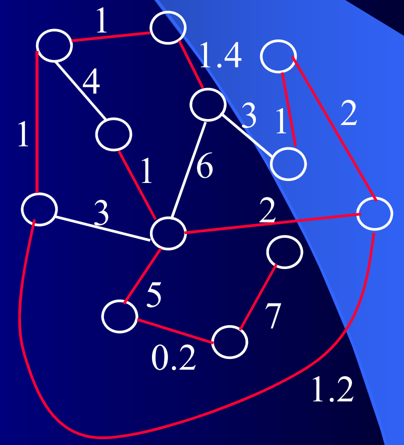
Graphen

- Einheiten (Knoten) werden miteinander assoziiert (Kanten)
- verschiedene Implementierung:
  - Adjazenzmatrix: gut für dichte Graphen
  - Adjazenzlisten: gut für lichte Graphen
- Algorithmen:
  - Tiefensuche: last-in-first-out Liste
  - Breitensuche: first-in-first-out Liste



Minimaler Spannbaum

- ein minimaler Spannbaum ist eine Teilmenge der Kanten, so dass
  1. alle Knoten miteinander verbunden sind
  2. die Summe der Kantengewichte wenigstens so klein ist, wie jeder andere Spannbaum
- Implementierung:
  - Tiefen-/Breitensuche mit
    - Prioritätenliste
    - Prioritätenheap (schön & schwer)



Modifikation von minimaler Spannbaum: Implementierung

- der Algorithmus zur Berechnung des minimalen Spannbaums kann leicht modifiziert werden, um
  - zu einem gegebenen Knoten k
  - und einer gegebenen Zahl m
  - die m dichtesten Knoten von k

  auszugeben

Datenkomprimierung

- einfaches Verfahren: Lauflängenkodierung
  - mehrfaches hintereinander Vorkommen von Daten wird ausgenutzt
  - funktioniert für Texte i.d.R. schlecht
- komplexes Verfahren: variable Lauflängenkodierung (Huffman Codierung)
  - der Text wird untersucht nach Häufigkeiten der Buchstaben
  - häufige Buchstaben erhalten eine kurze, seltene Buchstaben eine lange Kodierung
  - Kodierung muss präfixeindeutig sein
  - elementare Datenstruktur: Trie