Vorlesung 1

Programmierung II

Einführung in Algorithmen und Datenstrukturen

Dozent:	Prof. Dr. Peter Kelb
Raumnummer:	Z4030
e-mail:	pkelb@hs-bremerhaven.de
Sprechzeiten:	nach Vereinbarung
Sourcen:	Elli

Ziel der Vorlesung:

- Bewertung von Algorithmen (Groß-O Notation)
- Sortierverfahren
- dynamische Datenstrukturen (Vektoren und Listen)
- Suchverfahren (Bäume und Hashing)

Voraussetzung:

- Vorlesung: Programmierung I (inhaltlich, nicht formal)

Bücher:

- weiter das Java Handbuch aus Programmieren I

- die Onlinedokumentation zur aktuellen JDK Version

- Algorithmen: Algorithmen und Datenstrukturen
  (Pearson Studium - IT),
  Robert Sedgewick, Kevin Wayne,
  Pearson Studium,
  ISBN-13: 978-3868941845

Organisation:

- 2 SWS Vorlesung Swing für WInf
- 2 SWS Vorlesung Algorithmen für WInf und Inf

- 1 x 2 SWS Übung

Prüfung:

- siehe Veranstaltung Prog II für WInf bzw. Inf
- nähere Informationen in den Übungsgruppen

Worum geht es bei Algorithmen?

- viele Verfahren handeln davon, Daten zu finden
- oft werden dazu die Daten dazu sortiert, weil in sortierten Daten die gesuchten Informationen schneller gefunden werden
- leider dauert das Sortieren auch Zeit
- meistens ist man an den Verfahren interessiert, die am Schnellsten laufen
- jedoch brauchen i.A. die schnellen Verfahren mehr Speicherplatz als die langsamen Verfahren
- in selten Fällen ist man aber nicht primär an Geschwindigkeit sondern an Platzersparnis interessiert (siehe Komprimierungsverfahren für Text, Ton, Bild und Film)

Minium / Maximum Suche

Aufgabe

Eine Methode minMax soll aus einem übergebenen int-Array die minimale und die maximale Zahl suchen und zurück gegeben.

Problem

Wie kann eine Methode zwei int-Werte zurück geben?

Lösung

1. Eine Klasse MinMaxResult deklarieren, die zwei int-Werte als Objektvariablen besitzt
2. Die Methode minMax liefert ein Objekt der Klasse MinMaxResult zurück

MinMaxResult

```
import java.util.Random;
class MinMaxResult {
        MinMaxResult(int min,int max) {
                m_Min = min;
                m_Max = max;
        }
        final int m_Min;
        final int m_Max;
}

public class MinMaxUtils {
        static int[] genArray(int length) {
                int[] res = new int[length];
                Random rnd = new Random();
                for(int i = 0;i < res.length;++i)
                        res[i] = rnd.nextInt() % 1000;
                return res;
        }
        static void print(int[] field) {
                for(int i : field)
                        System.out.print(i + " ");
                System.out.println();
        }
}
```

Die MinMaxResult Klasse

Die Objektvariablen sind final, da sie nach der Initialisierung nicht mehr verändert werden sollen

2 Hilfs-methoden:

erzeugt ein int-Array der Länge length mit zufälligen int-Werten

druckt das übergebene int-Array auf der Konsole aus

Minium / Maximum Suche (Forts.)

- Mit der MinMaxResult Klasse kann die Bestimmung des Miniums/Maximums leicht implementiert werden
- hierzu gibt es unterschiedliche Varianten

1. die Clevere (???) Variante:
   - wäre das Array bereits sortiert, wäre das Minimum ganz am Anfang und das Maximum ganz am Ende des Arrays gespeichert
   - im 1. Semester wurden doch Verfahren zum Sortieren vorgestellt, dann könnte man doch die verwenden

Go!

MinMax 1

```
public class MinMax1 {

    static void selection_sort(int[] field) {
        ... // siehe Prog I 1. Semester
    }

    static MinMaxResult minMax(int[] field) {
        selection_sort(field);
        return new MinMaxResult(field[0],field[field.length-1]);
    }

    public static void main(String[] args) {
        int[] field = MinMaxUtils.genArray(10);
        System.out.println("Start Array");
        MinMaxUtils.print(field);
        MinMaxResult res = minMax(field);
        System.out.println("nach Minimum/Maximum Suche");
        MinMaxUtils.print(field);
        System.out.println("Ergebnis");
        System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
    }
}
```

Der Selection Sort wurde im 1. Semester besprochen

nach dem Sortieren ist das Minium an der Stelle 0 und das Maximum an der Stelle field.length-1

generiere zufälliges int-Array der Länge 10 und drucke es aus

bestimme MinMax

MinMax1: Diskussion

- prinzipiell funktioniert die MinMax1 Version insofern, dass sowohl das Minimum als auch das Maximum gefunden wird
- leider wird das Eingabearray durch die Sortierung verändert (es wird sortiert)
- dies soll nicht passieren, da die Methode nur die Information (Minium/Maximum) berechnen soll, nicht aber das Array modifizieren darf
- daher muss das Array, dass sortiert wird, zunächst kopiert werden

Go!

https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

MinMax 2

```
import java.util.Arrays;

public class MinMax2 {

    static void selection_sort(int[] field) { ... }

    static MinMaxResult minMax(int[] field) {
        int[] copy = Arrays.copyOf(field,field.length);
        selection_sort(copy);
        return new MinMaxResult(copy[0],copy[copy.length-1]);
    }

    public static void main(String[] args) {
        int[] field = MinMaxUtils.genArray(10);
        System.out.println("Start Array");
        MinMaxUtils.print(field);
        MinMaxResult res = minMax(field);
        System.out.println("nach Minimum/Maximum Suche");
        MinMaxUtils.print(field);
        System.out.println("Ergebnis");
        System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
    }
}
```

um die Arrays Klasse zu verwenden

vor dem Sortieren eine Kopie erzeugen

Rest wie gehabt

MinMax2: Diskussion

- die MinMax2 Version funktioniert jetzt, ohne dass das originale Eingabe Array verändert wird
- diese Lösung kann man aber noch verbessern
- in der Arrays Klasse befindet sich neben der copy-Methode auch eine sort-Methode zum Sortieren
- diese kann man statt der selbstgeschrieben Selection Sort Methode verwenden

Go!

MinMax 3

```
import java.util.Arrays;

public class MinMax3 {

    static MinMaxResult minMax(int[] field) {
        int[] copy = Arrays.copyOf(field,field.length);
        Arrays.sort(copy);
        return new MinMaxResult(copy[0],copy[copy.length-1]);
    }

    public static void main(String[] args) {
        int[] field = MinMaxUtils.genArray(10);
        System.out.println("Start Array");
        MinMaxUtils.print(field);
        MinMaxResult res = minMax(field);
        System.out.println("nach Minimum/Maximum Suche");
        MinMaxUtils.print(field);
        System.out.println("Ergebnis");
        System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
    }
}
```

Nach wie vor: vor dem Sortieren eine Kopie erzeugen

vordefinierte Sortierfunktion

Rest wie gehabt

MinMax3: Diskussion

- die MinMax3 Version funktioniert bei diesem einfachen Test genauso wie die MinMax2 Version mit selbstgeschriebenen Sortierverfahren
- insgesamt sind aber beide Verfahren nicht sehr intelligent
- für das Kopieren des Arrays muss schon das gesamte Array durchlaufen werden
- bei einem Arraydurchlauf hätte man schon das Minium suchen können
- bessere Idee: das Array zweimal durchlaufen, um beim
  1. Durchlauf das Minimum und beim
  2. Durchlauf das Maximum zu suchen

Go!

MinMax 4

```
public class MinMax4 {
    static MinMaxResult minMax(int[] field) {
        int min = field[0];
        for(int i : field)
            if (i < min)
                min = i;
        int max = field[0];
        for(int i : field)
            if (i > max)
                max = i;
        return new MinMaxResult(min,max);
    }

    public static void main(String[] args) {
        int[] field = MinMaxUtils.genArray(10);
        System.out.println("Start Array");
        MinMaxUtils.print(field);
        MinMaxResult res = minMax(field);
        System.out.println("nach Minimum/Maximum Suche");
        MinMaxUtils.print(field);
        System.out.println("Ergebnis");
        System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
    }
}
```

in min wird das bisherige Minimum gespeichert

durchlaufe gesamtes Array: ist das neue Element kleiner als das bisherige Minimum, ist es das neue Minimum

das Gleiche nochmal für das Maximum

Rest wie gehabt

MinMax4: Diskussion

- auch die MinMax4 Version scheint zu funktionieren, ohne dass das Array kopiert und sortiert wird
- die beiden Durchläufe könnten noch zusammengefasst werden
- statt zweimal das Array zu durchlaufen, wird bei dem einzigen Durchlauf bei jedem Arrayelement getestet, ob es sich um das neue Minimum oder um das neue Maximum handelt

Go!

MinMax 5

```
public class MinMax5 {
    static MinMaxResult minMax(int[] field) {
        int min = field[0];
        int max = field[0];
        for(int i : field) {
            if (i < min)
                min = i;
            if (i > max)
                max = i;
        }
        return new MinMaxResult(min,max);
    }

    public static void main(String[] args) {
        int[] field = MinMaxUtils.genArray(10);
        System.out.println("Start Array");
        MinMaxUtils.print(field);
        MinMaxResult res = minMax(field);
        System.out.println("nach Minimum/Maximum Suche");
        MinMaxUtils.print(field);
        System.out.println("Ergebnis");
        System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
    }
}
```

zum Anfang ist das 1. Element sowohl das Minimum als auch das Maximum

nur noch ein Durchlauf: teste jedes Element, ob es das neue Minimum oder das neue Maximum ist

Rest wie gehabt

MinMax5: Diskussion

- auch die MinMax5 Version scheint zu funktionieren
- es besteht keine Notwendigkeit für zwei Durchläufe
- ob die Version schneller als die vorherigen Versionen ist, kann noch nicht festgestellt werden
- eine Optimierung ist noch möglich:

```
for(int i : field) {
        if (i < min)
            min = i;
        if (i > max)
            max = i;
}
```

wenn i das neue Minimum ist, kann es nicht gleichzeitig das neue Maximum sein: daher die 2. if-Bedingung nur berechnen, wenn die 1. if-Bedingung falsch war

```
for(int i : field) {
        if (i < min)
            min = i;
        else if (i > max)
            max = i;
}
```

einzige Änderung

Go!

MinMax 6

```
public class MinMax6 {
    static MinMaxResult minMax(int[] field) {
        int min = field[0];
        int max = field[0];
        for(int i : field) {
            if (i < min)
                min = i;
            else if (i > max)
                max = i;
        }
        return new MinMaxResult(min,max);
    }

    public static void main(String[] args) {
        int[] field = MinMaxUtils.genArray(10);
        System.out.println("Start Array");
        MinMaxUtils.print(field);
        MinMaxResult res = minMax(field);
        System.out.println("nach Minimum/Maximum Suche");
        MinMaxUtils.print(field);
        System.out.println("Ergebnis");
        System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
    }
}
```

einzige Änderung

Rest wie gehabt

MinMax6: Diskussion

- auch die MinMax6 Version scheint zu funktionieren
- ob die Lösung nun schneller ist, lässt sich bei diesem kleinen Test nicht sagen
- insgesamt lässt sich beobachten, dass alle Versionen das gleiche Antwortverhalten haben, nämlich das sofortige Liefern des Ergebnisses
- für echte Messungen müssen die Arrays deutlich größer sein
- hierzu wird die Zeit gemessen, die die Verfahren 2 – 6 (1 wird nicht betrachtet, da das Array modifiziert wird) für unterschiedlich große Arrays brauchen
- die Arraygröße startet bei 100 und wird in jedem Schritt verdoppelt bis zu einer Größe von 400.000.000 (400 Millionen)

Go!

MinMaxTest

```
public class MinMaxTest {

    static void minMax(int which,int[] field) {
        if (which == 2 && field.length > 400000)
            System.out.println("\tZeit in mSec.: zu lang für MinMax" + which);
        else {
            long lStart = System.currentTimeMillis();
            switch(which) {
                case 2: MinMax2.minMax(field);break;
                case 3: MinMax3.minMax(field);break;
                case 4: MinMax4.minMax(field);break;
                case 5: MinMax5.minMax(field);break;
                case 6: MinMax6.minMax(field);break;
            }
            long lEnd = System.currentTimeMillis();
            System.out.println("\tZeit in mSec.: " + (lEnd - lStart) + " für MinMax" + which);
        }
    }

    public static void main(String[] args) {
        for(int size = 100;size < 400000000;size *= 2) {
            int[] field = MinMaxUtils.genArray(size);
            System.out.println("Arraylänge: " + field.length);
            for(int which = 2;which < 7;++which)
                minMax(which,field);
        }
    }
}
```

Selection Sort nur für Arrays < 400.000 verwendbar

die aktuelle Zeit in Millisekunden

die aktuelle Zeit nach der Berechnung

Differenz der beiden Zeitpunkte ergibt die Zeitdauer

Abschließende Diskussion

- Version 2 (Selection Sort) braucht bei doppelt so großem Array viermal soviel Zeit
- Version 3 (Arrays.sort) braucht bei doppelt so großem Array etwa doppelt soviel Zeit (in Wirklichkeit ist es ein bisschen mehr, aber die Zeitmessung ist zu ungenau)
- Version 4-6 (ohne Sortierung) brauchen nicht messbar mehr Zeit und es gibt auch keine Unterschiede zwischen ihnen

WICHTIG: die Zeitmessung ist recht ungenau. Alle Angaben unter 1000 ms (= 1 Sekunde) sind „Schmutz“

- Frage: wie kann man solche Algorithmen bzgl. ihrer Geschwindigkeit bewerten, ohne sie laufen zu lassen?
- Kann man die Laufzeit ausrechnen?

Vorlesung 2

Bewertung von Algorithmen

- die MinMax Beispiele haben gezeigt, dass unterschiedliche Verfahren für der Berechnung ein- und derselben Ergebnisse sehr unterschiedliche Laufzeiten benötigen
- bei der Untersuchung der Laufzeit von Algorithmen ist
  1. ***NICHT*** die konkrete Zeit interessiert, sondern
  2. die Veränderung der Laufzeit, wenn die Eingabe sich verändert
- Warum?
- Laufzeit hängt von vielen Faktoren ab: Speichergröße, Betriebssystem, SSD oder HDD, Anzahl der Prozessoren und deren Kerne, was lief noch auf dem System ...

Eine konkrete Laufzeit wäre mit der nächsten Rechnergeneration wieder hinfällig

Bewertung von Algorithmen (Forts.)

- da die konkrete Laufzeit nicht interessant ist, geht man davon aus, dass jede Anweisung die gleiche Zeit braucht
- Beispiel:

```
public static void doit(int n,int m) {
        int j = 0;                    // 1. Zeitschritt
        System.out.println(n);        // 2. Zeitschritt
        if (n < m)                    // 3. Zeitschritt
                j = m;                // 4. Zeitschritt
        else                          // oder
                j = n * 2 / m;        // 4. Zeitschritt
        System.out.println(j);        // 5. Zeitschritt
}
```

- für die Komplexitätsbetrachtung würde es reichen, nur die einzelnen Schritte zu zählen:

```
static int cnt = 0;
public static void doit(int n,int m) {
        ++cnt;                  // 1. Zeitschritt
        ++cnt;                  // 2. Zeitschritt
        ++cnt; if (n < m)       // 3. Zeitschritt
                ++cnt;          // 4. Zeitschritt
        else                    // oder
                ++cnt;          // 4. Zeitschritt
        ++cnt;                  // 5. Zeitschritt
}
```

d.h. doit(int n,int m) braucht immer 5 Schritte, unabhängig von n oder m

Go!

Bewertung von Algorithmen: Beispiel

- betrachten wir einmal den Selection Sort

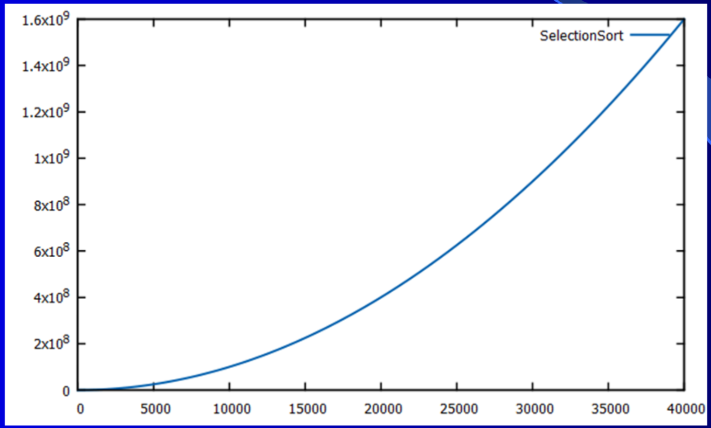
```
static void selection_sort(int[] field) {
    for(int i1 = 0;i1 < field.length - 1;++i1) {      // wiederhole alles „field.length“ mal
        int min = i1;                                 // 1 Schritt
        for(int i2 = i1 + 1;i2 < field.length;++i2) { // wiederhole alles „field.length-i1“ mal
            if (field[i2] < field[min])               // 1 Schritt
                min = i2;                             // 1 Schritt
        }
        swap(field, min, i1);                         // 3 Schritte (siehe unten)
    }
}
static void swap(int[] field,int iPos1,int iPos2) {
    int tmp = field[iPos1];          // 1. Schritt
    field[iPos1] = field[iPos2];     // 2. Schritt
    field[iPos2] = tmp;              // 3. Schritt
}
```

Für die Komplexität reicht es, die einzelnen Schritte zu zählen

```
static int selection_sort_analysis(int[] field) {
    int cnt = 0;
    for(int i1 = 0;i1 < field.length - 1;++i1) {
        ++cnt; // int min = i1;
        for(int i2 = i1 + 1;i2 < field.length;++i2) {
            ++cnt; // if (field[i2] < field[min])
            ++cnt; //    min = i2;
        }
        cnt += 3; // swap(field, min, i1);
    }
    return cnt;
}
```

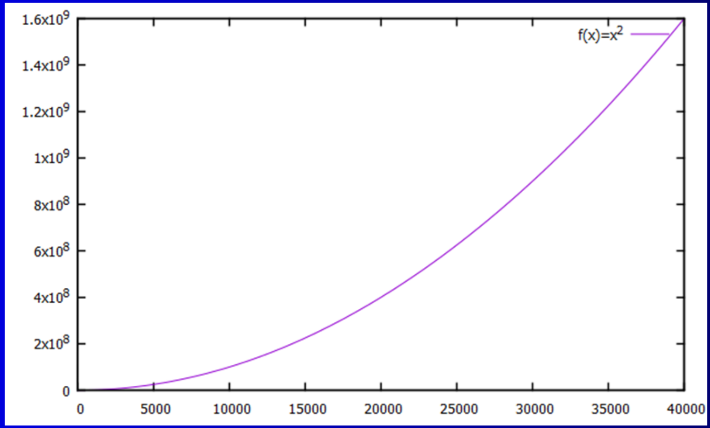
Bewertung von Algorithmen: Beispiel (Forts.)

- trägt man die ausgeführten Schritte (Y-Achse) über die Länge des Eingabearrays (X-Achse) ein, ergibt sich folgendes Bild:



- der Verlauf ist identisch zu der quadratischen Funktion $f(x) = x^2$

Der Selection Sort hat ein quadratisches Laufzeitverhalten !



Bewertung von Algorithmen: 2. Beispiel

- in Prog. I gab es die folgenden rekursiven Funktionen:

```
static void rec_double(int n) {
	if (n > 0) {
		rec_double(n-1);
		System.out.println(n);
		rec_double(n-1);
	}
}
```

Funktion ruft sich zweimal rekursiv auf

```
static void rec_first(int n) {
	if (n > 0) {
		rec_first(n-1);
		System.out.println(n);
	}
}
```

Funktion ruft sich einmal zum Anfang rekursiv auf

```
static void rec_last(int n) {
	if (n > 0) {
		System.out.println(n);
		rec_last(n-1);
	}
}
```

Funktion ruft sich einmal zum Ende rekursiv auf

Go!

Bewertung von Algorithmen: 2. Beispiel (Forts.)

- die entsprechenden Funktionen, die die einzelnen Schritte zählen, sähen wie folgt aus:

```
static void rec_double(int n) {

    if (n > 0) {
        rec_double(n-1);
        System.out.println(n);
        rec_double(n-1);
    }

}
```

```
static long rec_double_analysis(int n) {
    long cnt = 0;
    if (n > 0) {
        cnt += rec_double_analysis(n-1);
        ++cnt; // System.out.println(n);
        cnt += rec_double_analysis(n-1);
    }
    return cnt;
}
```

n	cnt
1	1
2	3
4	15
8	255
16	65535
32	4294967295

exponen
-tieller
Verlauf

```
static void rec_first(int n) {

    if (n > 0) {
        rec_first(n-1);
        System.out.println(n);
    }

}
```

```
static long rec_first_analysis(int n) {
    long cnt = 0;
    if (n > 0) {
        cnt += rec_first_analysis(n-1);
        ++cnt; // System.out.println(n);
    }
    return cnt;
}
```

n	cnt
1	1
2	2
4	4
8	8
16	16
32	32

```
static void rec_last(int n) {

    if (n > 0) {
        System.out.println(n);
        rec_last(n-1);
    }

}
```

```
static long rec_last_analysis(int n) {
    long cnt = 0;
    if (n > 0) {
        ++cnt; // System.out.println(n);
        cnt += rec_last_analysis(n-1);
    }
    return cnt;
}
```

n	cnt
1	1
2	2
4	4
8	8
16	16
32	32

linearer
Verlauf

Bewertung von Algorithmen: 2. Beispiel (Forts.)

- d.h. die rec_double Methode mit ihren 2 rekursiven Aufrufen hat einen exponentiellen Verlauf, während die beiden rec_first und rec_last einen linearen Verlauf haben
- der SelectionSort hatte ein quadratisches Laufzeitverhalten

Eingabe n	rec_first	rec_last	rec_double	SelectionSort
	cnt			
1	1	1	1	0
2	2	2	3	6
4	4	4	15	24
8	8	8	255	84
16	16	16	65535	300
32	32	32	4294967295	1116
	linear		exponentiell	quadratisch

Go!

Bewertung von Algorithmen: n×log(n)

- neben dem linearen, quadratischen und exponentiellen Laufzeitverhalten findet man oft noch das folgende Verhalten

zwei rekursive Aufrufe ...

```
static void rec_double(int n) {
      if (n > 0) {
            rec_double(n/2);
            for(int i = 0;i < n;++i)
                  System.out.print(i + " ");
            System.out.println();
            rec_double(n/2);
      }
}
```

... in der Mitte ein lineares Verhalten, ABER (!!!) ...

... die rekursiven Aufrufe halbieren das Argument

Bewertung von Algorithmen: n×log(n) (Forts.)

- betrachtet man das Laufzeitverhalten, so stellt man ***KEINEN*** exponentiellen Verlauf fest

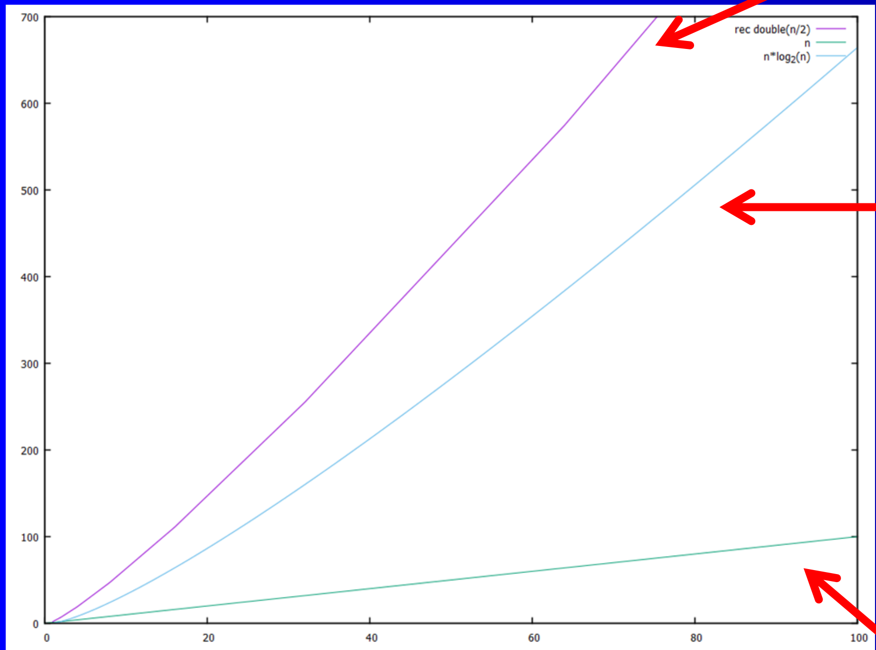
```
static long rec_double_analysis(int n) {
        long cnt = 0;
        if (n > 0) {
                cnt += rec_double_analysis(n/2);
                for(int i = 0;i < n;++i)
                        ++cnt; // System.out.print(i + " ");
                ++cnt; // System.out.println();
                cnt += rec_double_analysis(n/2);
        }
        return cnt;
}
```

n	cnt
1	2
2	7
4	19
8	47
16	111
32	255
64	575
128	1279
256	2815
512	6143
1024	13311
2048	28671
4096	61439
8192	131071
16384	278527
32768	589823
65536	1245183

- wenn die Eingabe n verdoppelt wird, erhöhen sich die Schritte ***ein wenig mehr*** als doppelt so viel

Bewertung von Algorithmen: n×log(n) (Forts.)

- die folgende Graphik zeigt, dass der Verlauf der Funktion $f(n) = n \times log_2(n)$ folgt
- zum Vergleich die lineare Funktion $f(n) = n$



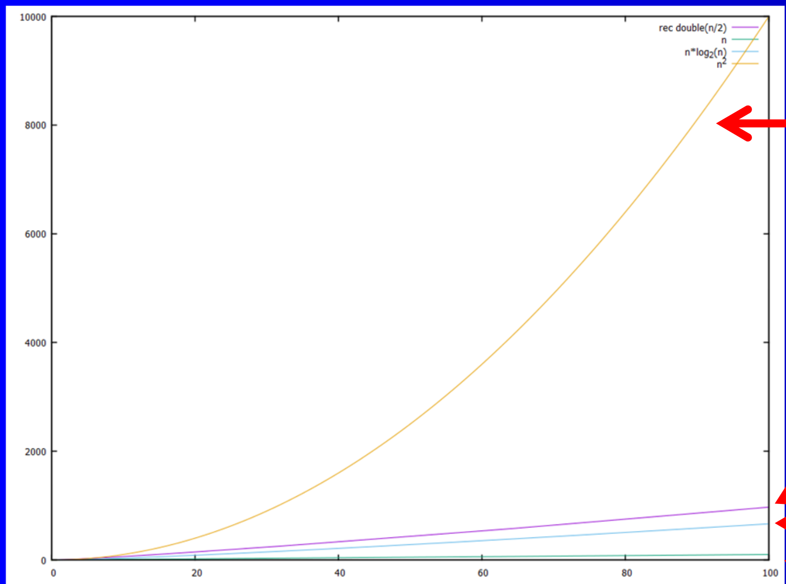
rec_double mit $\frac{n}{2}$

$f(n) = n \times log_2(n)$

$f(n) = n$

Bewertung von Algorithmen: n×log(n) (Forts.)

- man beachte, dass die y-Achse von 0 bis 700 geht, die x-Achse aber nur von 0 bis 100
- zur Verdeutlichung der Vergleich mit der quadratischen Funktion $f(n) = n^2$



$f(n) = n^2$

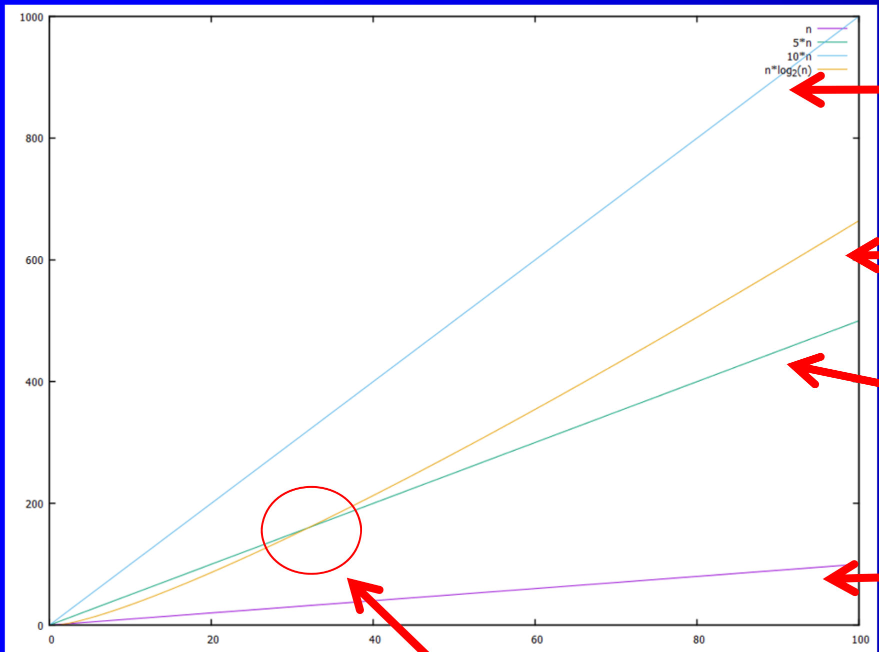
rec_double mit $\frac{n}{2}$

$f(n) = n \times log_2(n)$

$f(n) = n$

Bewertung von Algorithmen: n×log(n) (Forts.)

- die vorherige Graphik könnte suggerieren, dass $n \times log_2(n)$ viel stärker ansteigt, als eine lineare Funktion
- dies ist nicht der Fall, wie der Vergleich mit unterschiedlichen linearen Funktionen zeigt



$10 \times n$

$f(n) = n \times log_2(n)$

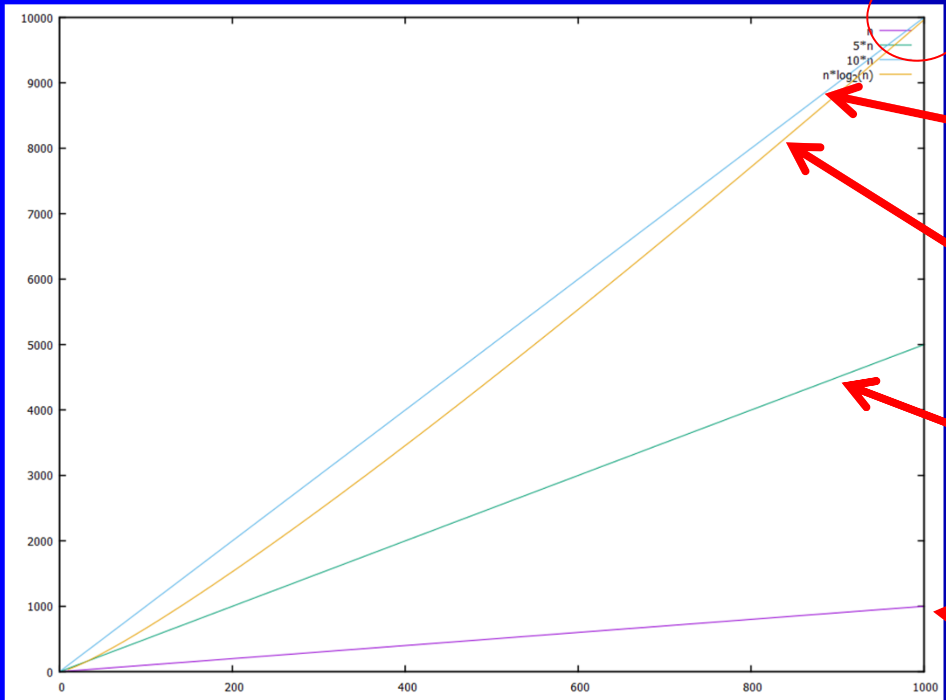
$5 \times n$

n

Schnitt mit $5 \times n$

Bewertung von Algorithmen: n×log(n) (Forts.)

- bis $n \times log_2(n)$ stärker ansteigt als die lineare Funktion $10 \times n$ dauert es lange
- selbst bei n=1000 ist es noch nicht ganz erreicht



noch kein Schnitt mit $10 \times n$

$10 \times n$

$f(n) = n \times log_2(n)$

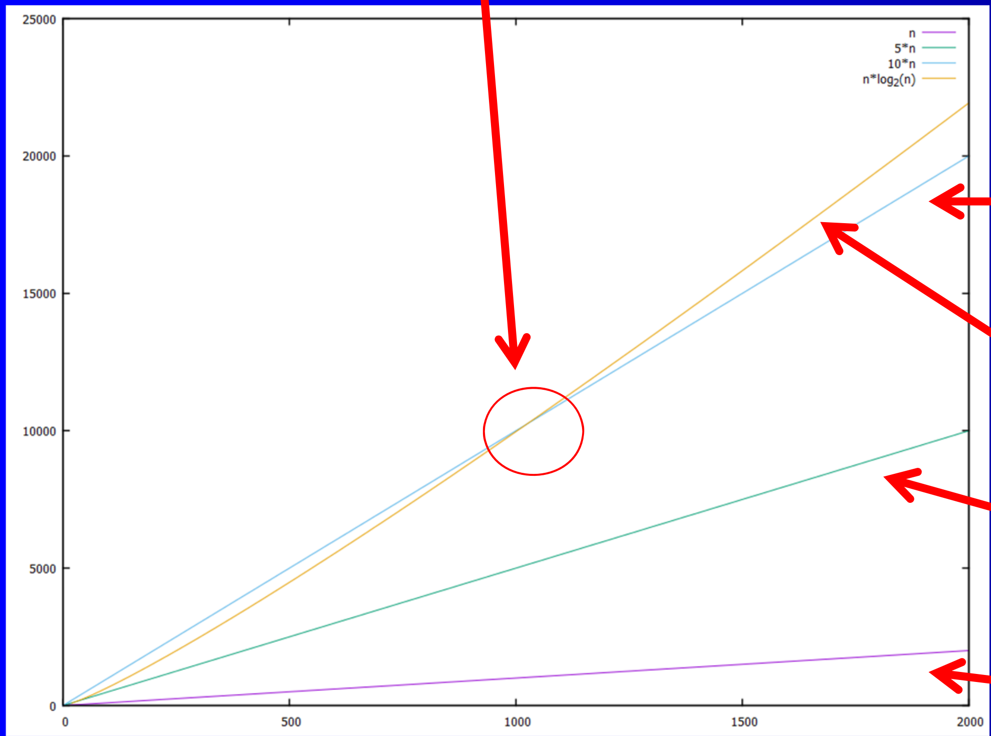
$5 \times n$

n

Bewertung von Algorithmen: n×log(n) (Forts.)

- der Schnitt liegt kurz hinter n=1000

Schnitt mit $10 \times n$



$10 \times n$

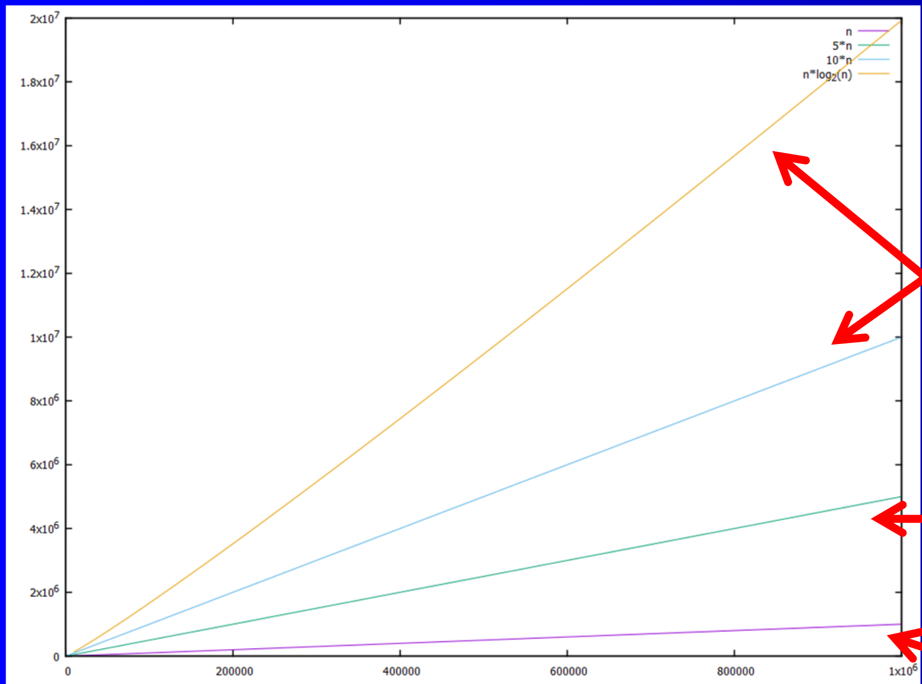
$f(n) = n \times log_2(n)$

$5 \times n$

n

Bewertung von Algorithmen: n×log(n) (Forts.)

- aber auch für deutlich größere n läuft die Funktion $n \times log_2(n)$ nicht der linearen Funktion $10 \times n$ fort



$10 \times n$

$f(n) = n \times log_2(n)$

$5 \times n$

n

Vergleich von Funktionen

- die MinMax1 Version hat das Array zu Beginn ***nicht*** kopiert
- die MinMax2 Version schon

- Frage:
  braucht die MinMax2 Version mehr Zeit als MinMax1?
- Antwort:
  Natürlich braucht MinMax2 mehr Zeit als MinMax1, aber ...

... am Ende (für große Eingaben) spielt dieser Mehraufwand keine Rolle !!!

Go!

Vergleich von Funktionen: Beispiel

- betrachten wir die drei folgenden Funktionen

```
static int simple(int n) {
        int cnt = 0;
        for(int i = 0;i < n;++i)
                for(int j = 0;j < n;++j)
                        ++cnt;
        return cnt;
}
```

n-mal

n-mal

1-mal

insgesamt:
$\underbrace{n + n + \cdots + n}_{\text{n-mal}} = n^2$

```
static int faster(int n) {
        int cnt = 0;
        for(int i = 0;i < n;++i)
                for(int j = 0;j < i;++j)
                        ++cnt;
        return cnt;
}
```

n-mal

i-mal (!!!!)

1-mal

insgesamt:
$0 + 1 + \cdots + n - 1 = \frac{n^2 - n}{2}$

```
static int complex(int n) {
        int cnt = 0;
        for(int i = 0;i < n;++i)
                ++cnt;
        cnt += simple(n);
        for(int i = 0;i < n;++i)
                ++cnt;
        return cnt;
}
```

n-mal

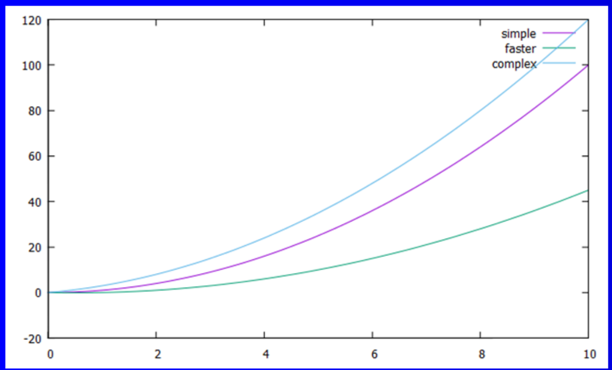
n^2

n-mal

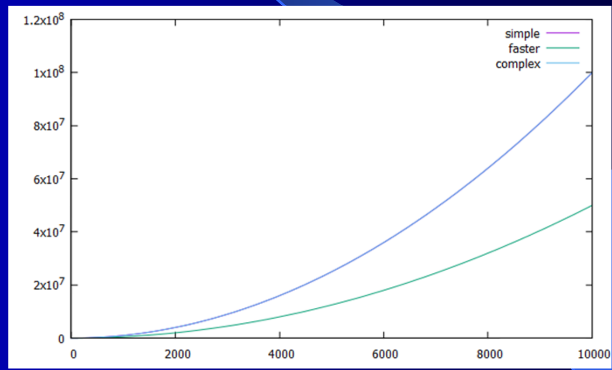
insgesamt:
$n + n^2 + n = 2n + n^2$

Vergleich von Funktionen: Beispiel (Forts.)

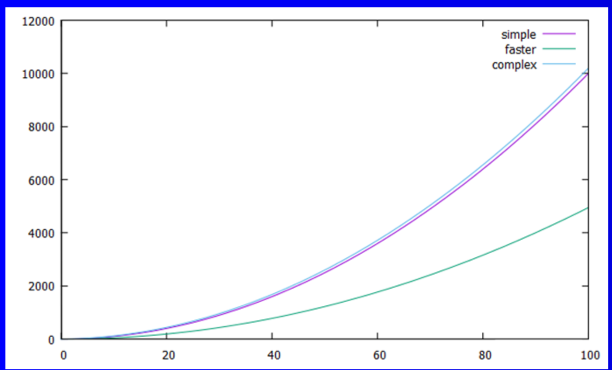
- Funktionsverläufe der Funktionen $simple(n) = n^2$, $faster(n) = \frac{n^2-n}{2}$ und $complex(n) = 2 \times n + n^2$



$n \leq 10$



$n \leq 10000$

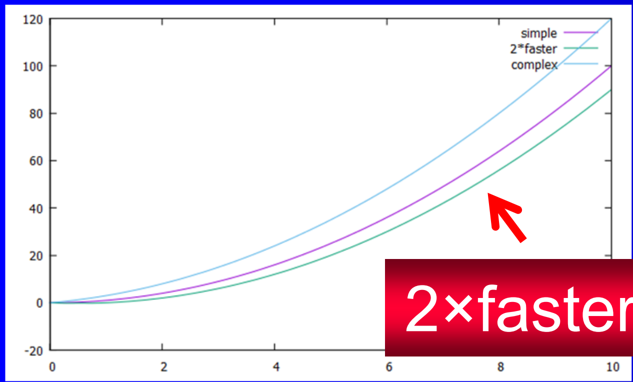


$n \leq 100$

simple und complex sind für große n nahezu identisch, nur faster scheint schneller zu sein, aber ...

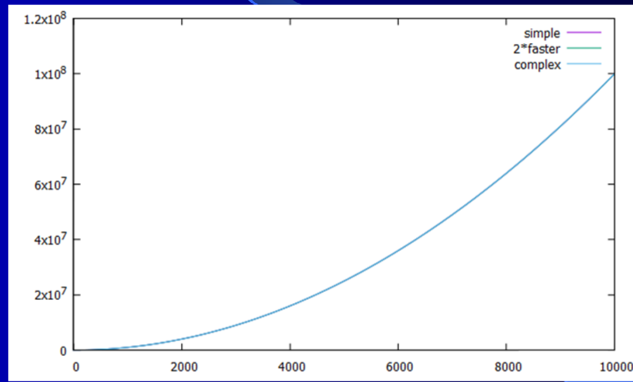
Vergleich von Funktionen: Beispiel (Forts.)

- betrachtet man statt der Funktion **faster(n)** die Funktion **2×faster(n)** ergibt sich ein anderes Bild

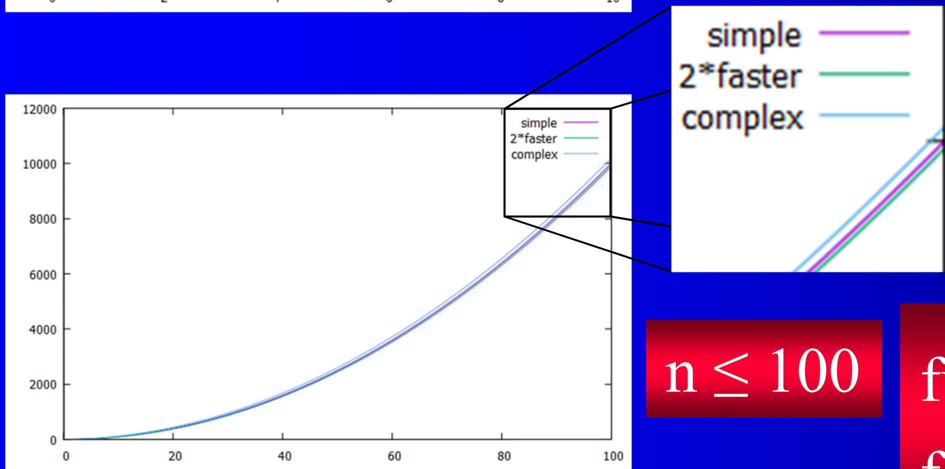


$n \leq 10$

2×faster



$n \leq 10000$



$n \leq 100$

für große n ist $2\frac{n^2-n}{2} = n^2 - n$ fast gleich zu n^2 und $2n + n^2$

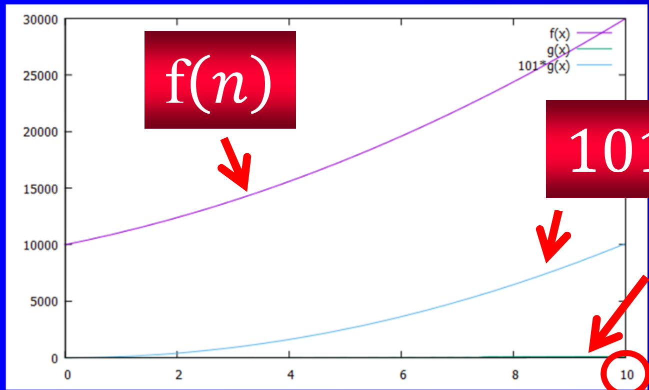
O-Notation

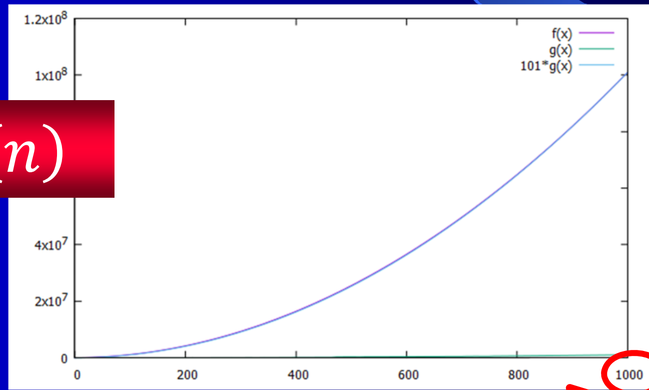
- wir sehen, dass bei großen n es so gut wie keine Rolle spielt, ob zu einer quadratischen Funktion noch eine beliebige lineare Funktion hinzuaddiert oder abgezogen wird
- für die Laufzeit (und auch Speicherplatz) Komplexitätsbetrachtung verwendet man daher die sogenannte O-Notation (Sprechweise: „groß O Notation")
- betrachten wir die Menge $F = \{f: \mathbb{N} \to \mathbb{N}\}$ aller Funktionen die natürliche Werte auf natürliche Werte abbildet
- dann bezeichnet $O(f)$ mit f$\in F$ die Menge aller Funktionen, für die folgendes gilt:

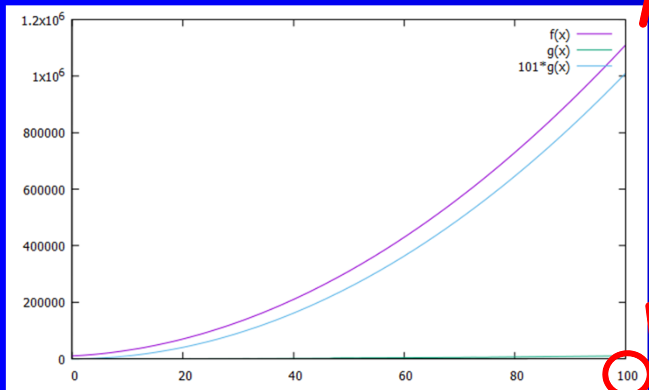
$$\forall f \in F: O(f) = \{g \in F \mid \exists\, c \in \mathbb{N}\ \exists\, n_0 \in \mathbb{N}\ \forall n \geq n_0: g(n) \leq c \times f(n)$$

O-Notation: Beispiel

- geben sei die Funktion:
$$f(n) = 10000 + 1000 \times n + 100 \times n^2$$
- Frage: liegt f in $O(n^2)$?
- zeichnen wir dazu die beiden Funktionen f und $g(n) = (n^2)$





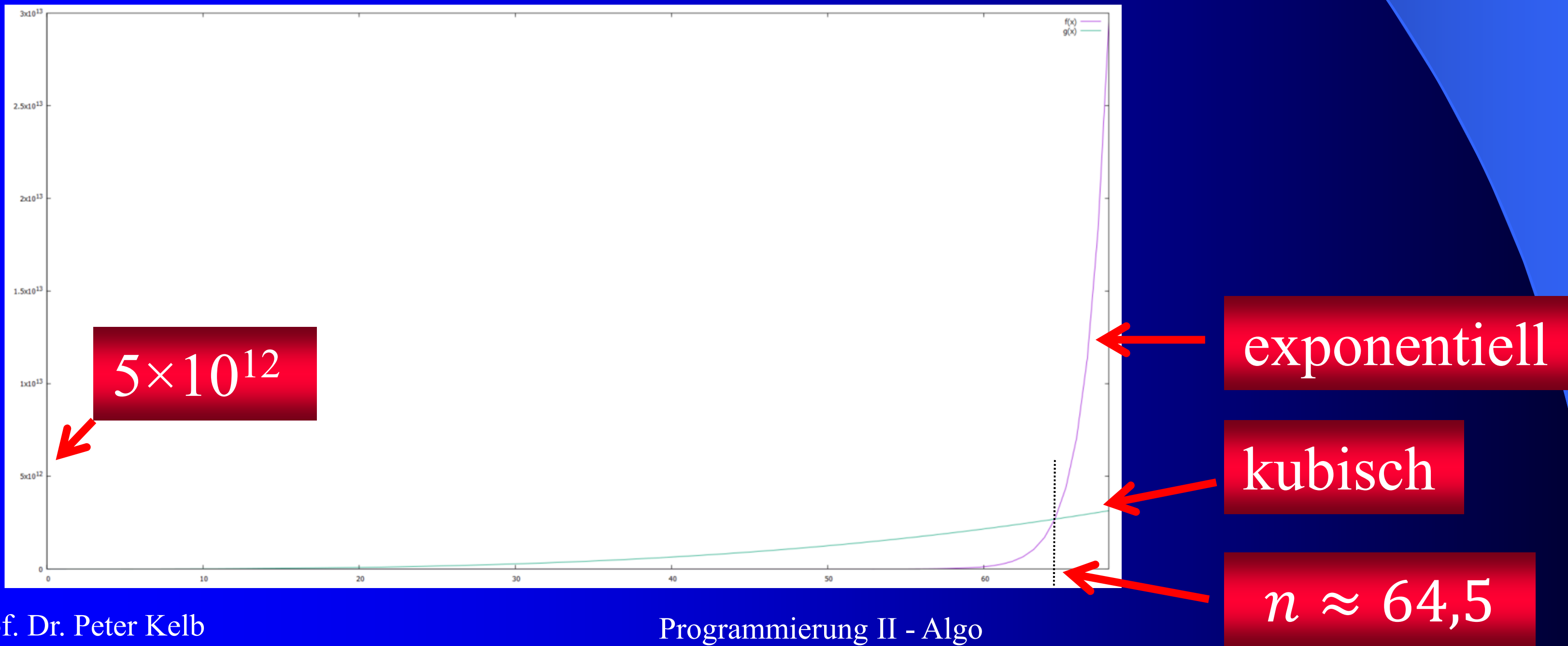


!!!

Antwort: ja, mit c=101 und $n_0 \approx 130$

O-Notation: weitere Überlegungen

- $f(n) = 2^n$ wächst sicherlich deutlich schneller als $g(n) = n^2$, aber wie sieht es aus, wenn die Exponentialfunktion stark gedämpft wird und statt einer quadratischen Funktion eine kubische mit großem Faktor gewählt wird
- also $f(n) = \frac{2^n}{10000000}$ und $g(n) = 10000000 \times n^3$



O-Notation: weitere Überlegungen

- zusammenfassend lässt sich sagen, dass die Komplexitätsklasse einer Funktion sich nach der am stärksten wachsenden Teilfunktion orientiert
- Beispiel:

$f(n) = 1000 \times n^3 + 500000 \times n^2 + 7600 \times n$ liegt in $O(n^3)$

$f(n) = 1000 \times n^3 + \frac{2^n}{500000} + 7600 \times n$ liegt in $O(2^n)$

$f(n) = 1000 \times n + 1243546456$ liegt in $O(n)$

Übliche Komplexitätsklasse

- im wesentlichen haben wir es mit den folgenden Komplexitätsklassen zu tun:

- $O(1)$ Zugriff auf ein beliebiges Arrayelement
- $O(log_2 log_2(n))$ Interpolationssuche
- $O(log_2(n))$ Binäre Suche
- $O(n)$ Zugriff auf ein beliebiges Listenelement
- $O(n \times log_2(n))$ gute Sortierverfahren
- $O(n^2)$ schlechte Sortierverfahren
- $O(2^n)$ das SAT Problem, eigentlich fast alles, was interessant ist

Vorlesung 3

Bewertung von einfachen Sortierverfahren

- in Prog. 1 wurden verschiedene einfache Sortierverfahren vorgestellt
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
  - Distribution Counting
- diese sollen jetzt untersucht und ihre Komplexität abgeschätzt werden

Selection Sort

```
static void sort(int[] field) {
	for(int i1 = 0;i1 < field.length - 1;++i1) {
		int min = i1;
		for(int i2 = i1 + 1;i2 < field.length;++i2) {
			if (field[i2] < field[min])
				min = i2;
		}
		swap(field, min, i1);
	}
}
```

min merkt sich immer die Position des kleinsten Elements

tausche die Elemente aus

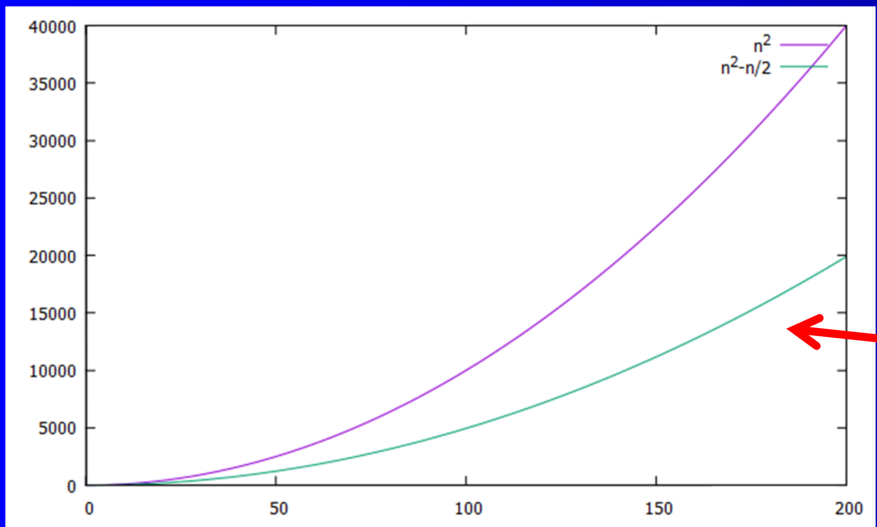
```
static void swap(int[] field,int iPos1,int iPos2) {
	int tmp = field[iPos1];
	field[iPos1] = field[iPos2];
	field[iPos2] = tmp;
}
```

vertauscht die beiden Elemente an den Positionen iPos1 und iPos2

Selection Sort: Analyse

Laufzeit: 1. Durchlauf: n-1 Schritte
2. Durchlauf: n-2 Schritte
3. Durchlauf: n-3 Schritte
...
insgesamt: $\frac{n^2-n}{2}$

$O(n^2)$ (zwei ineinander geschachtelte for-Schleifen)



Selection Sort Laufzeit

Insertion Sort

```
static void insertion_sort(int[] field) {
    for(int i1 = 1;i1 < field.length;++i1) {
        int val = field[i1];
        int i2 = i1;
        while (i2 > 0 && field[i2 - 1] > val) {
            field[i2] = field[i2 - 1];
            --i2;
        }
        field[i2] = val;
    }
}
```

IVAL ist das Element, das eingefügt werden soll

verschiebe die bereits sortierten Elemente, bis IVAL richtig platziert ist

speichere IVAL an dem neu geschafften Platz ab

Insertion Sort: Analyse

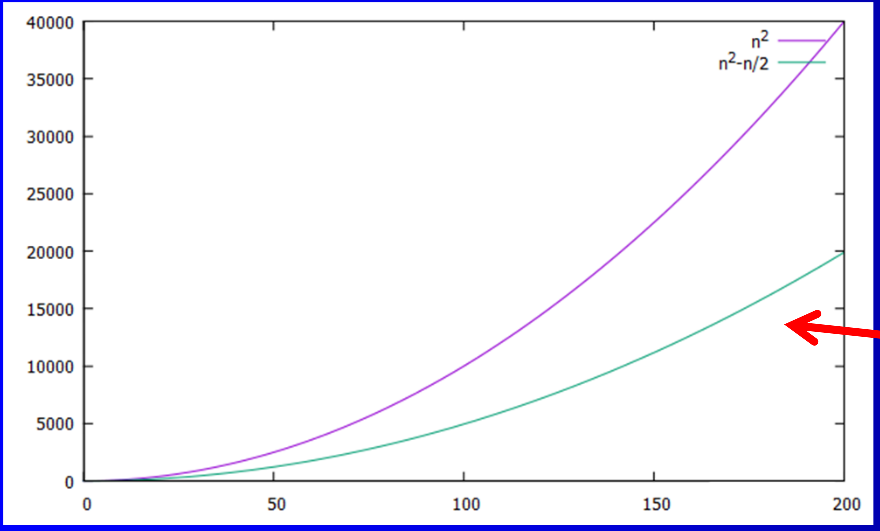
Laufzeit:

1. Durchlauf: maximal 1 Schritt
2. Durchlauf: maximal 2 Schritte
3. Durchlauf: maximal 3 Schritte

...

insgesamt: $\frac{n^2-n}{2}$

$O(n^2)$ (zwei ineinander geschachtelte Schleifen (for und while)

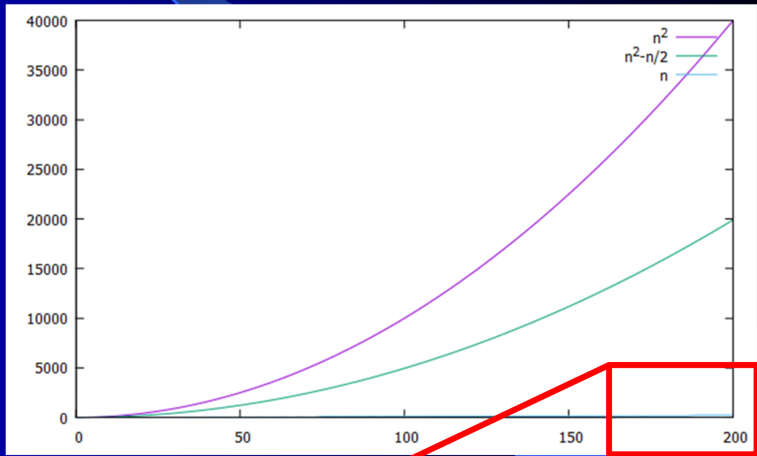


Insertion Sort Laufzeit (identisch zu Selection Sort)

Insertion Sort: Analyse (Fort.)

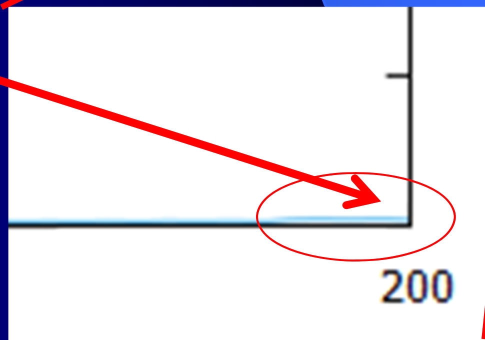
Was passiert bei Insertion Sort für dieses Array?

0	1	2	3	4
5	9	34	42	102



Laufzeit: 1. Durchlauf: 1 Schritt
2. Durchlauf: 1 Schritt
3. Durchlauf: 1 Schritt
...
insgesamt:

das bedeutet O(n)



O(n) (nur äußere for-Schleife)

d.h.: Laufzeit hängt stark von der Vorsortierung ab!

Bubble Sort

```
static void bubble_sort(int[] field) {
      for(int i1 = 1;i1 < field.length;++i1) {
            for(int i2 = 0; i2 < field.length-i1; ++i2) {
                  if (field[i2] > field[i2 + 1])
                        swap(field, i2, i2+1);
            }
      }
}
```

i2 läuft immer über das Aray, lässt dabei immer ein Element mehr aus

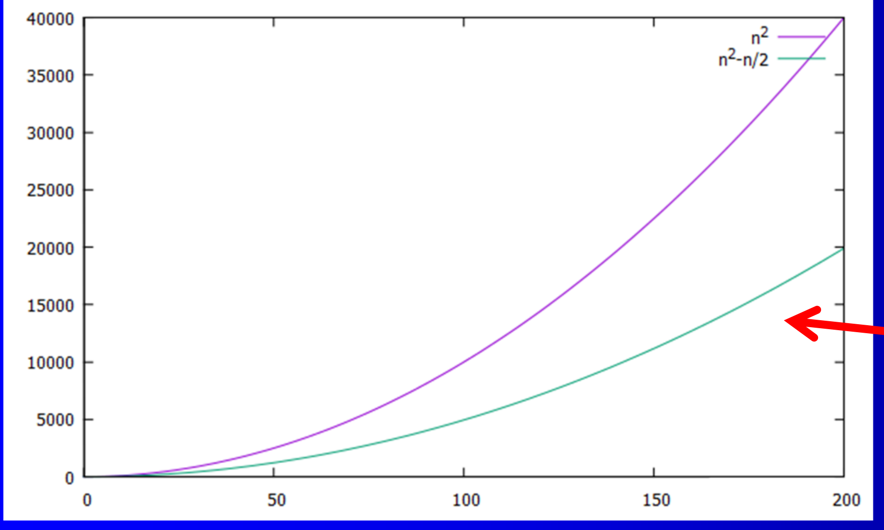
sind 2 aufeinanderfolgende Elemente nicht sortiert, werden sie vertauscht

geeignet für externes Sortieren, da nur sequentiell auf die Elemente zugegriffen wird (nach i2 kommt i2+1)

Bubble Sort: Analyse

Laufzeit: 1. Durchlauf: n-1 Schritte
2. Durchlauf: n-2 Schritte
3. Durchlauf: n-3 Schritte
...
insgesamt: $\frac{n^2-n}{2}$

$O(n^2)$ (zwei ineinander geschachtelte for-Schleifen)



Bubble Sort Laufzeit (identisch zu Selection und Insertion Sort)

Bubble Sort: Optimierung

```
static void bubble_sort_opt(int[] field) {
    for(int i1 = 1; i1 < field.length; ++i1) {
        boolean bAtLeastOneSwap = false;
        for(int i2 = 0; i2 < field.length-i1;++i2) {
            if (field[i2] > field[i2 + 1]) {
                swap(field, i2, i2+1);
                bAtLeastOneSwap = true;
            }
        }
        if (!bAtLeastOneSwap)
            return;
    }
}
```

merkt sich, ob wenigstens ein swap ausgeführt wurde

ja, es ist ein swap ausgeführt worden, das Array ist noch nicht sortiert

wenn im letzten Durchlauf kein swap ausgeführt wurde, sind wir fertig

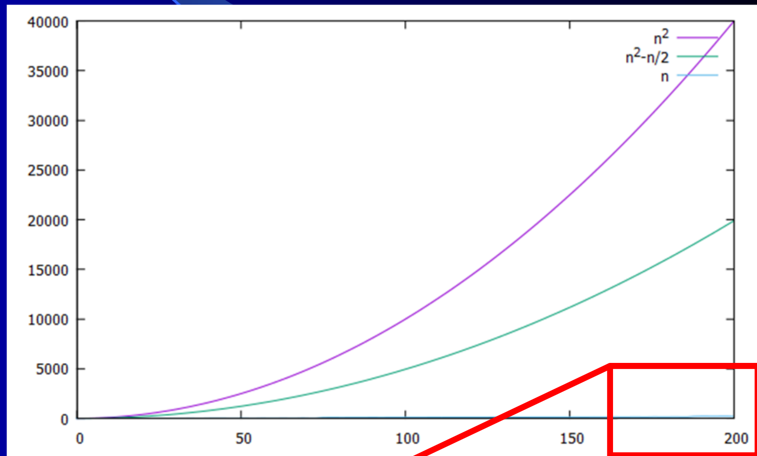
Bubble Sort: Analyse (Fort.)

Was passiert beim ***optimierten*** Bubble Sort für dieses Array?

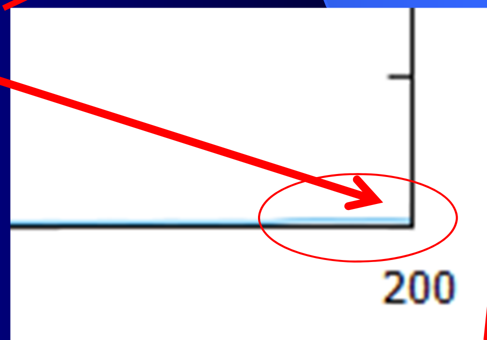
0	1	2	3	4
5	9	34	42	102

1. Durchlauf: n Schritte, kein swap, Abbruch

also: O(n)



das bedeutet O(n)



Distribution Counting

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

lege zusätzliches Feld an

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 < field.length; ++i) {
        ++count[field[i]];                      O(n)
    }
    for(int i1 = 0, i2 = 0; i1 < count.length; ++i1) {
        for(int i3 = 0; i3 < count[i1]; ++i3) {
            field[i2++] = i1;                   ???
        }
    }
}
```

Distribution Counting: Analyse (Fort.)

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

Überlegung:

- die äußere Schleife wird count.length-mal durchlaufen (also m)
- die innere Schleife wird sooft durchlaufen, so viele 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. $\sum_{i1=0}^{m} count[i1] = n$
- d.h., die Komplexität und damit Gesamtkomplexität ist O(n)

Zusammenfassung

	Durchschnitt	**vorsortierte Eingabe (Best Case)**
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$
Bubble Sort	$O(n^2)$	$O(n^2)$
optimierte Bubble Sort	$O(n^2)$	$O(n)$
Distrbution Counting	$O(n)$	$O(n)$

Bewertung: dies sind alles schlechte Sortierverfahren, die man nicht (!!!) verwenden sollte.
Ausnahme: Distribution Counting, das ist aber nur sehr sehr selten anwendbar

Go!

Beispiel 18

```
class Pair extends Object {
    int i;
    char c;
    Pair(int i,char c) {
        this.i = i;
        this.c = c;
    }
    public String toString() {
        return i + " " + c;
    }
}
class Beispiel18 {
    int value;
    public Beispiel18() {
        value = 42;
    }
    public Pair getValue(char ch) {
        return new Pair(value,ch);
    }
    public static void main(String[] args) {
        Beispiel18 b = new Beispiel18();
        System.out.println(b.getValue('?'));
    }
}
```

Frage: Was macht das Programm?

Frage: Muss die Klasse Pair an dieser Stelle deklariert sein?

Innere Klassen

```
class Pair extends Object {
    int i;
    char c;
    Pair(int i,char c) {
        this.i = i;
        this.c = c;
    }
    public String toString() {
        return i + " " + c;
    }
}
class Beispiel18 {
    int value;
    public Beispiel18() {
        value = 42;
    }
    public Pair getValue(char ch) {
        return new Pair(value,ch);
    }
    public static void main(String[] args) {
        Beispiel18 b = new Beispiel18();
        System.out.println(b.getValue('?'));
    }
}
```

Frage: Muss die Klasse Pair an dieser Stelle deklariert sein?

Antwort: Nein, sie kann auch innerhalb von der Klasse Beispiel18 deklariert sein.

Go!

```
class Beispiel18_1 {
    int value;
    class Pair extends Object {
        int i;
        char c;
        Pair(int i,char c) {
            this.i = i;
            this.c = c;
        }
        public String toString() {
            return i + " " + c;
        }
    }
    public Beispiel18_1() {
        value = 42;
    }
    public Pair getValue(char ch) {
        return new Pair(value,ch);
    }
    public static void main(String[] args) {
        Beispiel18_1 b = new Beispiel18_1();
        System.out.println(b.getValue('?'));
    }
}
```

Innere Klassen (Fort.)

Innere Klassen sind Klassen, die **innerhalb** einer anderen Klasse oder eines Blocks deklariert werden.

Innere Klassen (Fort.)

Es werden vier verschiedene Anwendungen von inneren Klassen unterschieden:

1. geschachtelte Top-Level Klassen

2. Elementklassen

3. Lokale Klassen

4. Anonyme Klassen

Geschachtelte Top-Level Klassen

```
class A {
        static class B {
                ...
        }
        ...
}

class Beispiel20 {
        ...
        public static void main(String [] args) {
                A.B b = new A.B();
        }
}
```

Die Klasse B ist in A geschachtelt, aber dennoch auf der Topebene zu verwenden, da sie static in A deklariert ist.

Elementklassen

```
class A {
        class B {
            ...
        }
        ...
}

class Beispiel20 {
        ...
        public static void main(String [] args) {
                A.B b = new A.B();
        }
}
```

Die Klasse B ist in A geschachtelt und kann auf der Topebene **nicht** verwendet werden, da sie nicht static deklariert ist.

Elementklassen (Fort.)

```
class Beispiel19 {
    int value;
    class Dummy extends Object {

        public String toString() {
            return Integer.toString(value);
        }
    }
    public Beispiel19(int i) {
        value = i;
    }
    public Dummy getValue() {
        return new Dummy();
    }
    public static void main(String[] args) {
        Beispiel19 b1 = new Beispiel19(23);
        Beispiel19 b2 = new Beispiel19(42);
        System.out.println(b1.getValue());
        System.out.println(b2.getValue());
    }
}
```

Objekte von Elementklassen können auf die Objektvariablen ihrer umschließenden Klassen zugreifen.

Objekte von Elementklassen merken sich die Objekte ihrer Oberklassen, aus denen sie erzeugt werden.

Instanziierung der Elementklasse erfolgt aus der umschließenden Klasse heraus.

Go!

Elementklassen (Fort.)

```
class Beispiel19 {
    int value;
    class Dummy extends Object {

        public String toString() {
            return Integer.toString(value);
        }
    }
    public Beispiel19(int i) {
        value = i;
    }
    public Dummy getValue() {
        return new Dummy();
    }
    public static void main(String[] args) {
        Beispiel19 b1 = new Beispiel19(23);
        Beispiel19 b2 = new Beispiel19(42);
        System.out.println(b1.getValue());
        System.out.println(b2.getValue());
    }
}
```

Was macht das Programm?

Lokale Klassen

```
class A {
        void doit() {
                class B {
                        ...
                }

                B b = new B();
                ...
        }

        public void test(String [] args) {
                B b = new B();
        }
...
}
```

Die Klasse B ist in der Methode doit geschachtelt und kann nur innerhalb der Methode verwendet werden.

Anonyme Klassen

```
class Beispiel19 {
    int value;
    class Dummy extends Object {

        public String toString() {
            return Integer.toString(value);
        }
    }
    public Beispiel19(int i) {
        value = i;
    }
    public Dummy getValue() {
        return new Dummy();
    }
    public static void main(String[] args) {
        Beispiel19 b1 = new Beispiel19(23);
        Beispiel19 b2 = new Beispiel19(42);
        System.out.println(b1.getValue());
        System.out.println(b2.getValue());
    }
}
```

Frage:

- Braucht man die Klasse Dummy eigentlich?
- Braucht man den Namen "Dummy"?
- Wo braucht man den Namen "Dummy"?

Anonyme Klassen (Fort.)

```
class Beispiel19 {
    int value;
    class Dummy extends Object {

        public String toString() {
            return Integer.toString(value);
        }
    }
    public Beispiel19(int i) {
        value = i;
    }
    public Dummy getValue() {
        return new Dummy();
    }
    public static void main(String[] args) {
        Beispiel19 b1 = new Beispiel19(23);
        Beispiel19 b2 = new Beispiel19(42);
        System.out.println(b1.getValue());
        System.out.println(b2.getValue());
    }
}
```

Idee:

Deklariere die Klasse dort, wo sie einmal gebraucht wird. Dann muss die Klasse keinen Namen haben

Go!

Anonyme Klassen (Fort.)

```
class Beispiel19_1 {
    int value;
    public Beispiel19_1(int i) {
        value = i;
    }
    public Object getValue() {
        return new Object() {
                public String toString() {
                    return Integer.toString(value);
                }
            };
    }
    public static void main(String[] args) {
        Beispiel19_1 b1 = new Beispiel19_1(23);
        Beispiel19_1 b2 = new Beispiel19_1(42);
        System.out.println(b1.getValue());
        System.out.println(b2.getValue());
    }
}
```

Anonyme Klassen können nur an genau einer Stelle instanziiert werden.

Anonyme Klassen und Interfaces

normale Klasse A	Anonyme Klasse

class A extends B {
    void doit() {...}
}
...
print(new A());
	print(new B() {
	    void doit() {...}
	  }
	);

class A implements C {
    void doit() {...}
}
...
print(new A());
	print(new C() {
	    void doit() {...}
	  }
	);

Innere Klassen: Zusammenfassung

- geschachtelte Top-Level Klassen

Klassen, die in einer anderen Klasse (mit `static`) deklariert werden, können auch woanders verwendet werden

- Elementklassen

Klassen, die in einer anderen Klasse (ohne `static`) deklariert werden, können nicht woanders verwendet werden, merken sich das umschließende Objekt

  - Lokale Klassen (ist eine Elementklasse)

  Klassen, die innerhalb eines Blocks definiert werden, können nur innerhalb des Blocks verwendet werden

    - Anonyme Klassen (ist eine lokale Klasse, ist eine El.)

    Lokale Klassen ohne Namen, können nur an der Stelle verwendet werden, an der sie deklariert werden

Vorlesung 4

nochmal Sortieren

konzeptionelle Nachteile bisher vorgestellter Sortierverfahren:

- Insertion-, Selection- und Bubblesort: langsam, sprich $O(n^2)$
- Distribution Couting: nur für eine spezielle Anwendung

Lösung: andere Algorithmen

implementationstechnische Nachteile der bisherigen Implementierungen:

- funktionieren nur für int-Arrays
- für Arrays anderen Typs müssen sie neu implementiert werden

Lösung: Generics und Interfaces

Go!

Generics: kurze Einführung

Generics: Parametrisierung von Klassen und Methoden in Typen

Aufgabe: Implementierung einer Klasse, die sich zwei beliebige Werte von beliebigen Typ (= Object) merkt

```
public class Pair1 {
    private Object m_o1,m_o2;
    public Pair1(Object o1,Object o2) {
        m_o1 = o1;m_o2 = o2;
    }
    Object get1() {return m_o1;}
    Object get2() {return m_o2;}

    public static void main(String[] args) {
        Pair1 p = new Pair1(2,'?');
        System.out.println(p.get1());
        char c      = (Character) p.get2();
        double d    = (Double)    p.get1();
        System.out.println(c + " " + d);
    }
}
```

Autoboxing: int → Integer und char → Character

expliziter Typcast mit Absturz

Go!

Generics: kurze Einführung (Forts.)

- ab Version 1.5 verfügbar
- sehr schwache Imitation von Templates (C++)

Klasse Pair2 ist parametrisiert in den Typen T1 und T2

```
public class Pair2<T1,T2> {
    private T1 m_o1;
    private T2 m_o2;
    public Pair2(T1 o1,T2 o2) {
        m_o1 = o1;m_o2 = o2;
    }
    T1 get1() {return m_o1;}
    T2 get2() {return m_o2;}

    public static void main(String[] args) {
        Pair2<Integer,Character> p = new Pair2<Integer,Character>(2,'?');
        System.out.println(p.get1());
        char c = p.get2();
        double d = p.get1();
        System.out.println(c + " " + d);
```

Members sind vom Typ T1 bzw. T2

Konstruktor erwartet die beiden Typen T1 und T2

Bei Instanziierung: Festlegung der Typparameter

Autounboxing: Character → char, Integer → int → double

Generics: kurze Einführung (Beispiel)

- eine Methode, die das Minimum zweier Werte beliebigen Typs ermittelt

zwei Werte a1 und a2 eines beliebigen Typs K

```
static <K> K min(K a1,K a2) {
      if (a1 < a2)
            return a1;
      else
            return a2;
}
```

Rückgabe ist natürlich auch vom Typ K

Problem: <-Operator ist nicht auf einen beliebigen Typen definiert

Generics: kurze Einführung (Beispiel)

- Lösung: zusätzlich ein Interface mitgeben, das beschreibt, wie zwei Werte vom Typ K verglichen werden

Interface mit einer Methode isLess, die besagt, ob a1 vom beliebigen Typ T kleiner als a2 ist

```
interface Compare<T> {
    boolean isLess(T a1,T a2);
}

static <K> K min(K a1,K a2,Compare<K> c) {
    if (c.isLess(a1,a2))
        return a1;
    else
        return a2;
}
```

die Methode min ist von beliebigen Typ K. Das Compare Interface soll genau diesen Typ verwenden.

isLess Methode des Interfaces Compare ersetzt den <-Operator

Go!

Generics: kurze Einführung (Beispiel)

- Für den Aufruf von min muss zunächst eine Klasse, die das Interface Compare implementiert, definiert werden

```
interface Compare<K> {
    boolean isLess(K a1,K a2);
}
class MyCompare implements Compare<Integer> {
    public boolean isLess(Integer a1,Integer a2) {
        return a1<a2;
    }
}
public class Sorted2 {
    static <K> K min(K a1,K a2,Compare<K> c) {
        if (c.isLess(a1,a2))
            return a1;
        else
            return a2;
    }
    public static void main(String[] args) {
        System.out.println(min(3,4,new MyCompare()));
    }
}
```

MyCompare implementiert den Vergleich für Integer Objekte (und nur für solche)

Beim Aufruf der min Methode muss ein MyCompare Objekt übergeben werden

Go!

https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html

Comparable Interface

- anstatt des eigenen Compare Interface sollte man das vordefinierte Comparator Interface von Java verwenden

```
import java.util.Comparator;

class MyCompare implements Comparator<Integer> {
	public int compare(Integer a1,Integer a2) {
		if (a1 < a2)
			return -1;
		else if (a2 < a1)
			return 1;
		else
			return 0;
	}
}

public class Sorted6 {

	static <K> K min(K a1,K a2,Comparator<K> c) {
		if (c.compare(a1,a2) < 0)
			return a1;
		else
			return a2;
	}

	public static void main(String[] args) {
		System.out.println(min(3,4,new MyCompare()));
	}
}
```

```
public interface Comparator<T> {
	int compare(T o1,T o2);
}
```

ersetzt das eigene Compare Interface

Ergebnis:
<0: wenn o1 < o2
>0: wenn o1 > o2
=0: wenn o1 == o2

Go!

Generics: kurze Einführung (Diskussion)

- Der Aufruf von min ist ok
- Die Definition der Klasse MyCompare wirkt fehl am Platz
- hier könnte man eine anonyme Klasse verwenden

```
public class Sorted5 {
    static <K> K min(K a1,K a2,Comparator<K> c) {
        if (c.compare(a1,a2) < 0)
            return a1;
        else
            return a2;
    }
    public static void main(String[] args) {
        System.out.println(min(3,4,
                          new Comparator<Integer>() {
                              public int compare(Integer a1,Integer a2) {
                                  return a1-a2;
                              }
                          })
                      );
    }
}
```

anonyme Klasse, die das Comparator<Integer> Interface implementiert

Generics: kurze Einführung (Diskussion)

- Viel Schreibaufwand, anonyme Klasse könnte auch vom Compiler generiert werden
- nur `a1<a2` ist neu und kann nicht vom Compiler generiert werden
- Lösung hierzu: Lambda Ausdrücke

könnte vom Compiler erzeugt werden

```
public static void main(String[] args) {
    System.out.println(min  (3,4,
                    new Comparator<Integer>() {
                        public int compare(Integer a1,Integer a2) {
                            return a1-a2;
                        }
                    }
                )
            );
}
```

muss explizit programmiert werden

Lambda Ausdrücke

Problem mit der Klasse MyCompare und anonymen Klasse:

- sehr viel Schreibarbeit
- die komplette Deklaration der Klasse könnte der Compiler selber schreiben
- die einzige Information, die der Compiler nicht kennt, ist die Anwendung des <-Operators, also der Methodenrumpf
- daher ab Java 1.8: Lambda Ausdrücke (Begriff aus der funktionalen Programmierung)
- nur noch der Inhalt der Funktion muss implementiert werden
- nicht mehr implementiert werden muss:
  - Deklaration der Klasse
  - Deklaration der überlagerten Methode

Lambda Ausdrücke (Forts.)

- Lambda Ausdrücke funktionieren nur bei Interfaces, die genau eine Methode enthalten
- sie funktionieren ***nicht*** bei
  - Interfaces mit mehreren Methoden
  - abstrakten Klassen
  - normale Klassen
- solche Interfaces heißen „funktionale Interfaces“ (sie spezifizieren im Wesentlichen eine Funktion)

Go!

Lambda Ausdrücke (Forts.)

- Lambda Ausdrücke: ein näherer Blick

```
interface Juhu {
    public void doit();
}

public class Lambda1 {

    public static void main(String[] args) {
        Juhu j1 = new Juhu() {
            public void doit() {
                System.out.println("dies ist der alte Weg");
            }
        };
        Juhu j2 = () -> System.out.println("mit Lambda Ausdruck");

        j1.doit();
        j2.doit();
    }
}
```

ohne Parameter müssen leere Klammern gesetzt werden

Go!

Lambda Ausdrücke (Forts.)

- die Methoden können auch mehrere Parameter enthalten

```
interface Juhu {
    public void doit(int i,float f);
}

public class Lambda2 {

    public static void main(String[] args) {
        Juhu j1 = new Juhu() {
            public void doit(int i,float f) {
                System.out.println("old school: i = " + i + " f = " + f);
            }
        };
        Juhu j2 = (i,f) -> System.out.println("Lambda: i = " + i + " f = " + f);

        j1.doit(12,34.6f);
        j2.doit(5,7.8f);
    }
}
```

mehrere Parameter müssen auch in Klammern gesetzt werden

Go!

Lambda Ausdrücke (Forts.)

- mehrere Statements müssen in einem Block zusammengefasst werden

```
interface Juhu {
    public void doit(int i,int j);
}

public class Lambda4 {

    public static void main(String[] args) {
        Juhu j = (i1,i2) -> {
                System.out.println("jetzt kommt " + i1 + " mal die " + i2);
                for(int i = 0;i < i1;++i)
                    System.out.println(i2);
            };

        j.doit(10,13);
    }
}
```

Block von hier bis hier

Go!

Lambda Ausdrücke (Forts.)

- die Methoden können auch einen Rückgabewert haben

```
interface Juhu {
    public int doit(int i,int j);
}

public class Lambda3 {

    public static void main(String[] args) {
        Juhu j1 = (i1,i2) -> {return i1*i2;};
        Juhu j2 = (x,y) -> x / y;

        int a = j1.doit(12,10);
        int b = j2.doit(12,5);
    }
}
```

obwohl nur ein Statement (**return**) muss es dennoch im Block stehen

Spezialfall „Lambda Ausdrücke“: das **return** kann weggelassen werden, dann auch ohne Block

Go!

Lambda Ausdrücke (Forts.)

- das min-Beispiel mit Lambda Ausdrücken:

```
import java.util.Comparator;

public class Sorted3 {

    static <K> K min(K a1,K a2,Comparator<K> c) {
        if (c.compare(a1,a2) < 0)
            return a1;
        else
            return a2;
    }

    public static void main(String[] args) {
        System.out.println(min(3,4,(a1,a2) -> a1-a2));
    }
}
```

Die Klasse MyCompare fehlt komplett

Der Lambda Ausdruck, der die MyCompare Definition und die Objekterzeugung ersetzt

Go!

Lambda Ausdrücke (Forts.)

- großer Vorteil: soll die min-Methode nicht mehr das Minimum sondern das Maximum berechnen, muss nur der <-Operator durch den >-Operator ersetzt werden mit Lambda Ausdrücken:

```
import java.util.Comparator;
public class Sorted4 {

    static <K> K min(K a1,K a2,Comparator<K> c) {
        if (c.compare(a1,a2) < 0)
            return a1;
        else
            return a2;
    }

    public static void main(String[] args) {
        System.out.println(min(3,4,(a1,a2) -> a1-a2));
        System.out.println(min(3,4,(a1,a2) -> a2-a1));
    }
}
```

Berechnet Minimum

Berechnet Maximum

MinMax Suche mit Generics

- mit Hilfe der Generics kann die MinMax Suche aus der 1. Vorlesung verallgemeinert werden, d.h.
- sie funktioniert nicht nur für int-Arrays, sondern für beliebige Arrays
- hierzu muss zunächst die MinMaxResult Klasse auf Generics umgestellt werden

```
import java.util.Comparator;

class MinMaxResult<T> {

    MinMaxResult(T min,T max) {
        m_Min = min;
        m_Max = max;
    }
    final T m_Min;
    final T m_Max;
}
...
```

MinMaxResult ist jetzt im Typ parametrisiert, den MinMaxResult in den Objektvariablen m_Min und m_Max speichern soll

Go!

MinMax Suche mit Generics (Forts.)

Wie werden die Elemente T verglichen?

```
...

public class MinMaxGeneric {

    static<T> MinMaxResult<T> minMax(T[] field,Comparator<T> c) {
        T min = field[0];
        T max = field[0];
        for(T i : field) {
            if (c.compare(i,min) < 0)
                min = i;
            else if (c.compare(i,max) > 0)
                max = i;
        }
        return new MinMaxResult<T>(min,max);
    }

    public static void main(String[] args) {
        Integer[] field = {23,-12,2,0,79,-56};
        MinMaxResult<Integer> res = minMax(field,(x,y) -> x - y);
        System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
    }
}
```

Lambda Ausdruck für das Interface Comparator

Go!

MinMax Suche mit Generics (Forts.)

- minMax funktioniert jetzt auch mit String Arrays
- der Vergleich zweier Strings erfolgt hierbei mittels der compareTo Methode der String Klasse

Ob der String x kleienr als der String y ist, entscheidet die compareTo Methode

```
public static void main(String[] args) {
    String[] field = {"juhu","otto","anna","horst","zoe"};
    MinMaxResult<String> res = minMax(field,(x,y) -> x.compareTo(y));
    System.out.println("min: " + res.m_Min + "; max: " + res.m_Max);
}
```

Vorlesung 5

Sortieren (die Rückkehr)
Zur Erinnerung: Insertion Sort

```
static  void insertion_sort(int[] field) {
    for(int i1 = 1;i1 < field.length;++i1) {
        final int IVAL = field[i1];
        int i2 = i1;
        while (i2 >= 1 && field[i2 - 1] > IVAL) {
            field[i2] = field[i2 - 1];
            i2 = i2 - 1;
        }
        field[i2] = IVAL;
    }
}
```

der Datentyp int muss an diesen beiden Stellen geändert werden

der Vergleichsoperator muss an dieser Stelle verändert werden

Insertion Sort mit Generics und Lambda

```
static <K> void insertion_sort(K[] field,Comparator<K> c) {
	for(int i1 = 1;i1 < field.length;++i1) {
		final K IVAL = field[i1];
		int i2 = i1;
		while (i2 >= 1 && c.compare(IVAL,field[i2 - 1]) < 0) {
			field[i2] = field[i2 - 1];
			i2 = i2 - 1;
		}
		field[i2] = IVAL;
	}
}
...
Integer[] f = {5,3,4,-2,0,-17};
insertion_sort(f,(x,y) -> x-y);
insertion_sort(f,(x,y) -> y-x);
```

aufsteigend sortieren

absteigend sortieren

Insertion Sort: eine genauere Beobachtung

```
static <K> void insertion_sort(K[] field,Comparator<K> c) {
    for(int i1 = 1;i1 < field.length;++i1) {
        final K IVAL = field[i1];
        int i2 = i1;
        while (i2 >= 1 && c.compare(IVAL,field[i2 - 1]) < 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

schlimmster 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

$\Rightarrow$

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> void insertion_sort(K[] field,Comparator<K> c) {
    for(int i1 = 1;i1 < field.length;++i1) {
        final K IVAL = field[i1];
        int i2 = i1;
        while (i2 >= 1 && c.compare (IVAL,field[i2 - 1]) < 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> void shell_sort(K[] field,Comparator<K> c) {
...
        for( ; iDist > 0; iDist /= 3) {
            for(int i1 = iDist; i1 < field.length; ++i1) {
                final K IVAL = field[i1];
                int i2 = i1;
                while (i2 >= iDist && c.compare(IVAL,field[i2 - iDist]) < 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> void shell_sort(K[] field,Comparator<K> c) {
	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 && c.compare(IVAL,field[i2 - iDist]) < 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?

Quicksort

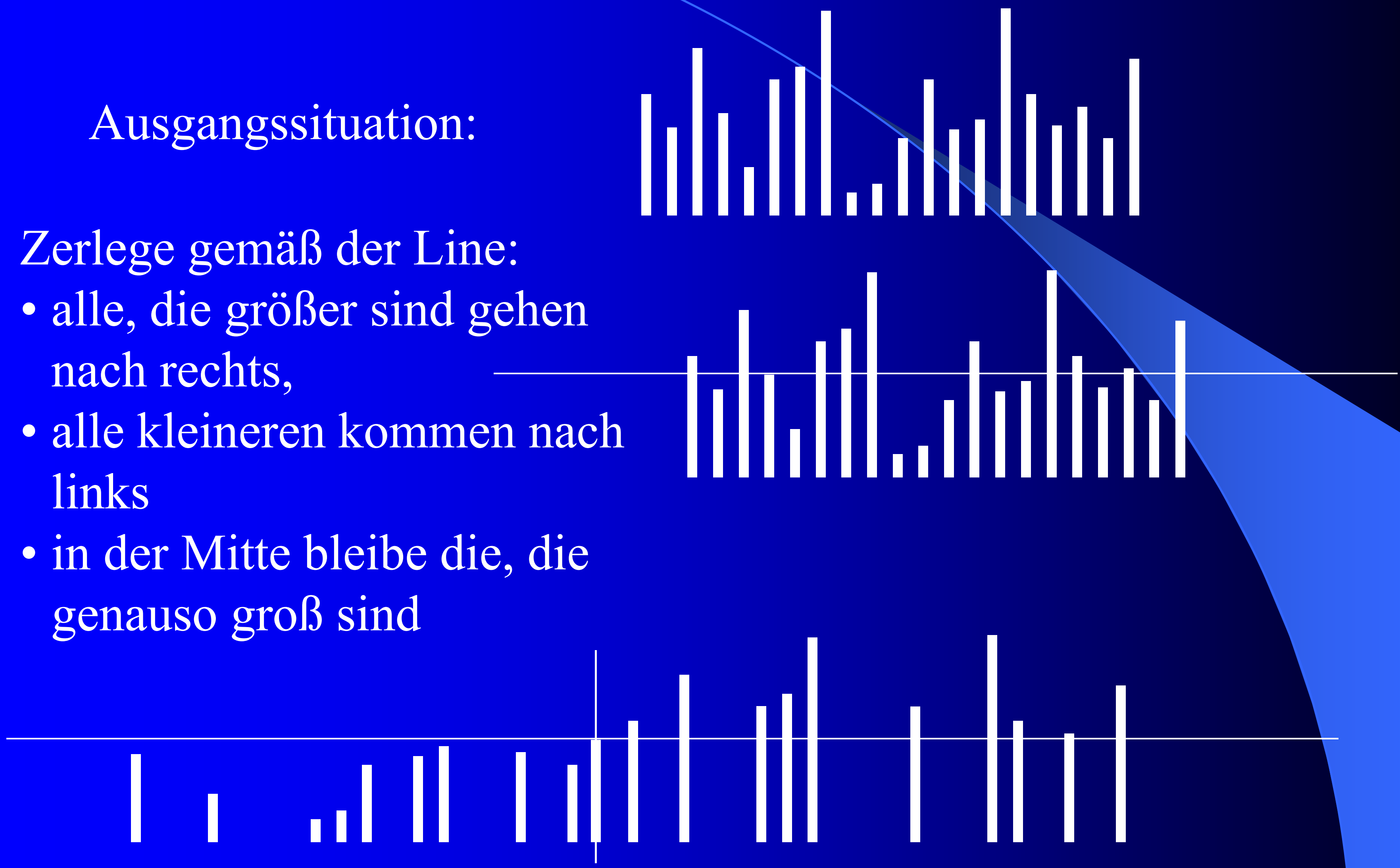
Idee:

- 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

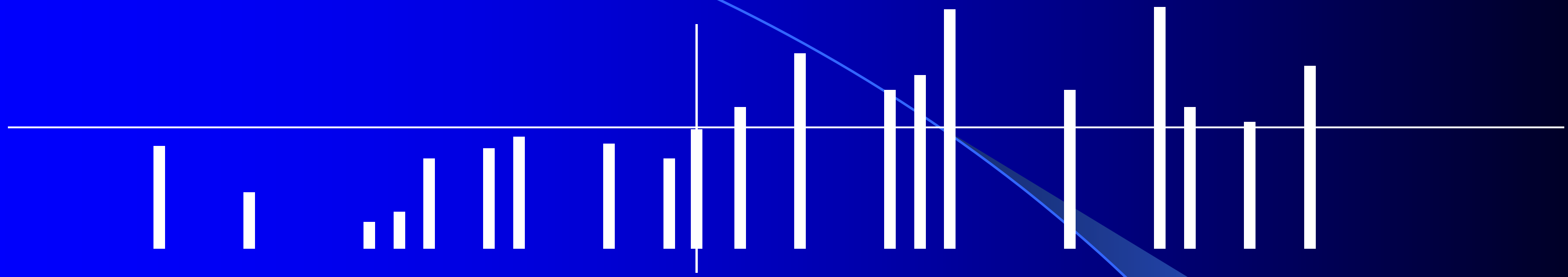
Quicksort: Illustration

Ausgangssituation:

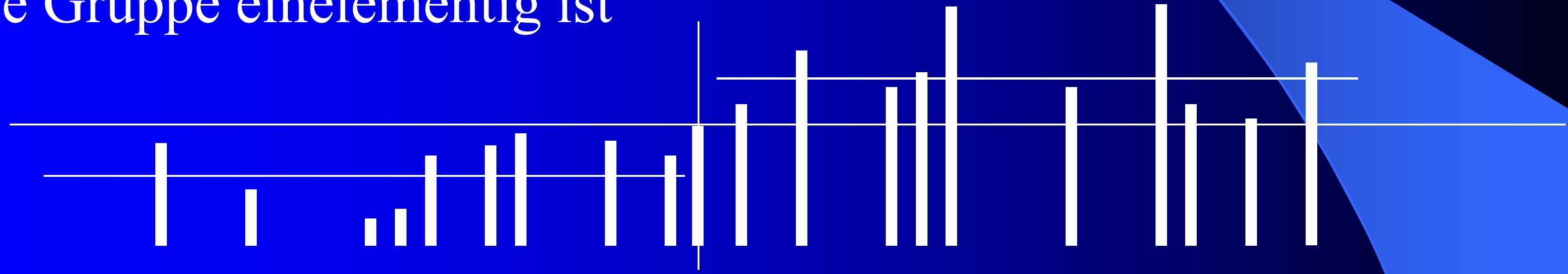
Zerlege gemäß der Line:
- alle, die größer sind gehen nach rechts,
- alle kleineren kommen nach links
- in der Mitte bleibe die, die genauso groß sind



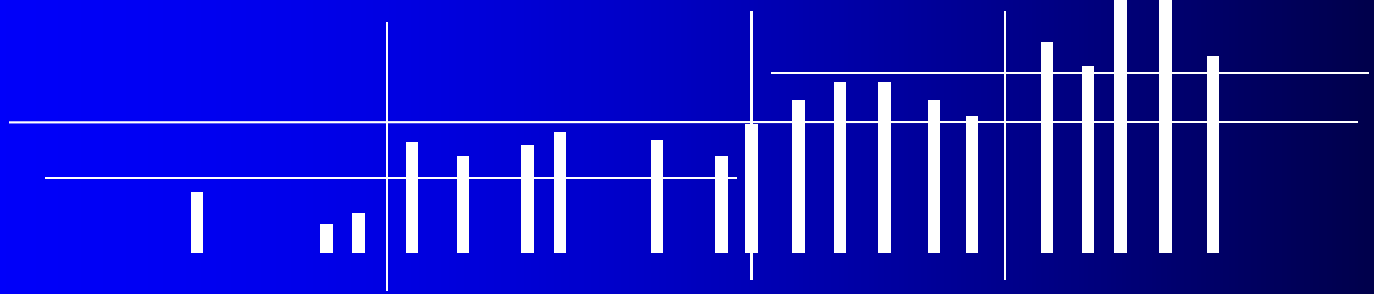
Quicksort: Illustration (Fort.)



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



Ergebnis:



Quicksort: Implementierung

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

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

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

    while(l < r) {
        while(c.compare(field[l],MID) < 0) { ++l; }
        while(c.compare(MID,field[r]) < 0) { --r; }
        if(l <= r)
            swap(field, l++, r--);
    }
    if (iLeft  < r)
        quick_sort_help(field,c, iLeft, r );
    if (iRight > l)
        quick_sort_help(field,c, 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> void quick_sort_help(K[] field,
                                Comparator<K> c,
                                int iLeft,
                                int iRight) {
    final K MID = field[(iLeft + iRight) / 2];
    int l = iLeft;
    int r = iRight;

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

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

Vorlesung 6

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

Ergebnis:

-17	23	30	40	45

Mergesort: Implementierung

```
static <K> void merge_sort2(K[] field,Comparator<K> c) {
    merge_sort_help2(field,c,0,field.length-1);
}

static <K> void merge_sort_help2(K[] field,Comparator<K> c,int iLeft,int iRight) {
    if (iLeft < iRight) {
        final int MIDDLE = (iLeft + iRight) / 2;
        merge_sort_help2(field,c, iLeft, MIDDLE);
        merge_sort_help2(field,c, MIDDLE + 1, iRight);

        K[] tmp = (K[]) new Object[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] = c.compare(tmp[iL],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.)

field enthält N Elemente, die zu sortieren sind

```
static <K> void heap_sort_1(K[] field,Comparator<K> c) {
        A<K> a = new A<K>(field.length);
        for(int i = 0;i < field.length;++i)
                a.insert(field[i],c);
        for(int i = 0;i < field.length;++i)
                field[field.length - i - 1] = a.remove(c);
}
```

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:

- ein unsortiertes Array
  - 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 $O(N^2)$

- ein sortiertes Array
  - insert erfolgt sortiert in das Array: Komplexität O(N)
  - remove entfernt das letzte Element: Komplexität O(1)
  - dies würde dem ***Insertion Sort*** entsprechen: Komplexität $O(N^2)$

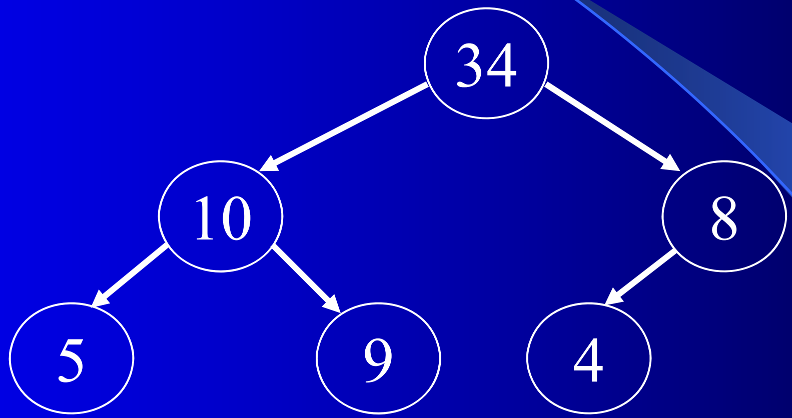
Heapsort (Fort.)

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*** (siehe Graphentheorie)

Heapsort (Fort.)

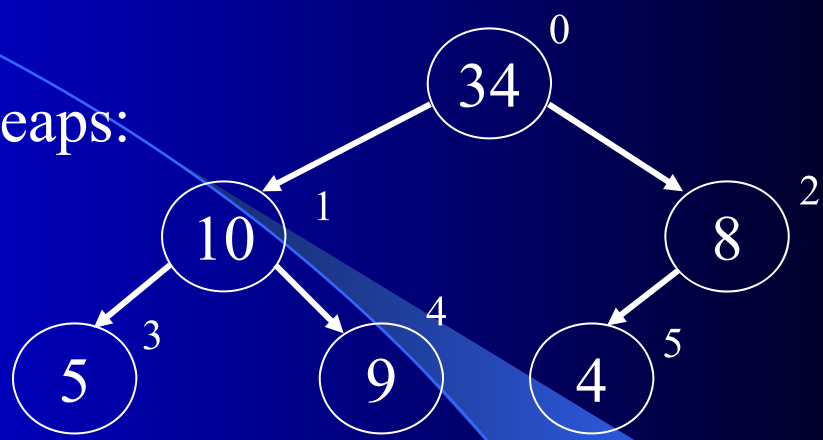
Beispiel für einen solchen Baum / Heap:



- 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:

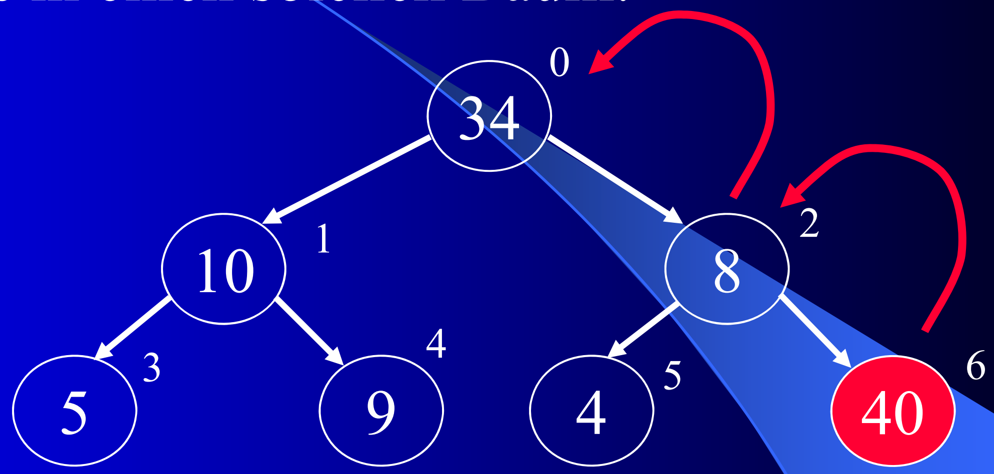
```
class Heap<K> {

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

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

Heapsort (Fort.)

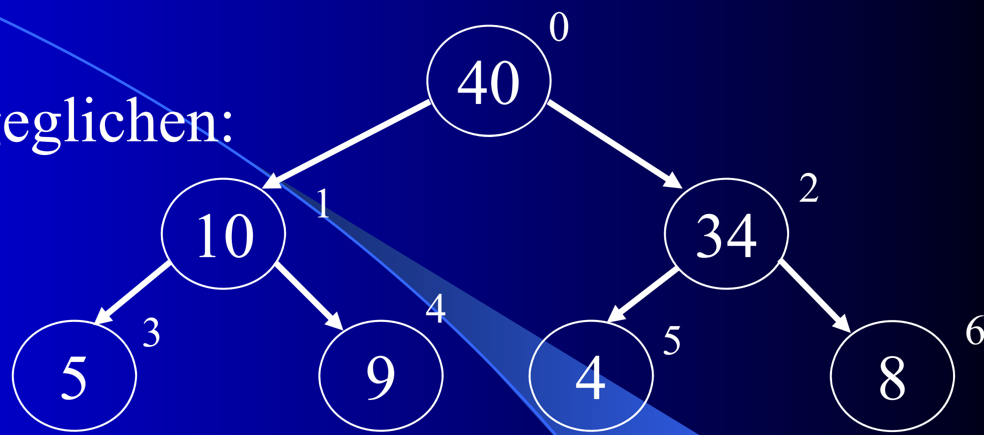
Einfügen eines Elements in einen solchen Baum:



- 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,Comparator<K> c) {
    K k = m_Keys[iIndex];
    while (iIndex != 0 && c.compare(m_Keys[(iIndex-1) / 2],k) < 0) {
        m_Keys[iIndex] = m_Keys[(iIndex-1) / 2];
        iIndex = (iIndex - 1) / 2;
    }
    m_Keys[iIndex] = k;
}
```

Heapsort (Fort.)

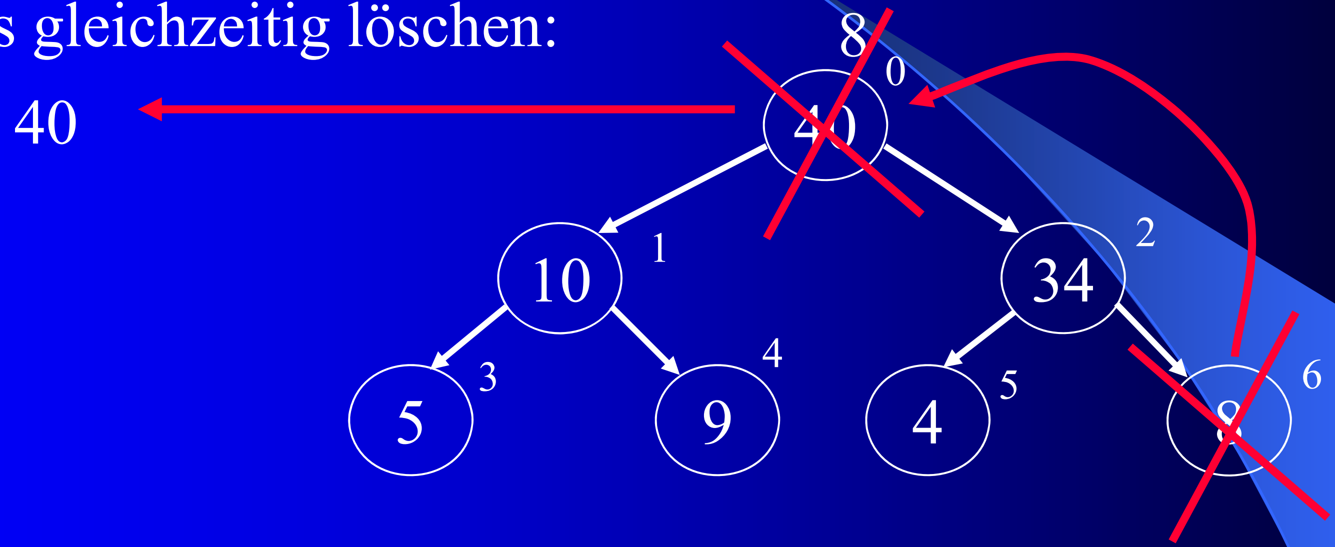
- basierend auf der upheap Methode kann die Insert Methode wie folgt implementiert werden:

```
public void insert(K key,Comparator<K> c) {
        m_Keys[m_iNext] = key;
        upheap(m_iNext,c);
        ++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.)

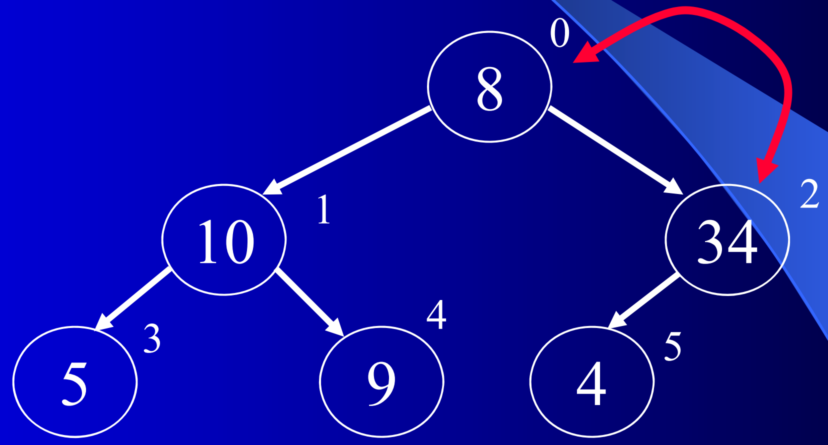
- die remove Methode soll das größte Element zurückliefern
- und es gleichzeitig löschen:



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

Heapsort (Fort.)

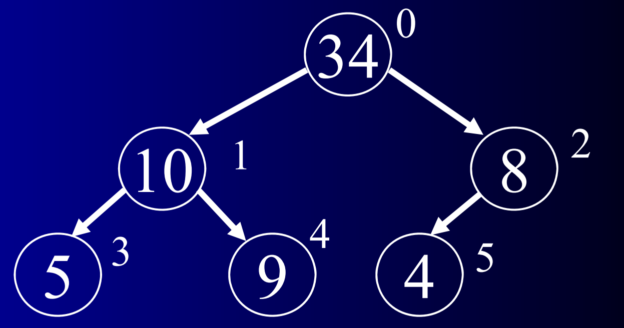
- der resultierende Baum ist nicht mehr korrekt
- die Spitze wird jetzt i.d.R. nicht mehr größer sein als die beiden Söhne



- 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(K iIndex,Comparator c) {
    K k = m_Keys[iIndex];
    while (iIndex < m_iNext / 2) {
        int iSon = 2 * iIndex + 1;
        if (iSon < m_iNext-1 &&
            c.compare(m_Keys[iSon] ,m_Keys[iSon + 1]) < 0)
            ++iSon;
        if (!(c.compare (k ,m_Keys[iSon]) < 0))
            break;
        m_Keys[iIndex] = m_Keys[iSon];
        iIndex = iSon;
    }
    m_Keys[iIndex] = k;
}
```

es gibt noch einen Sohn

1. Sohn

größerer der beiden Söhne

Heapsort (Fort.)

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

```
public K remove(Comparator<K> c) {
        K res = m_Keys[0];
        m_Keys[0] = m_Keys[--m_iNext];
        downheap(0,c);
        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 Hinunterwandern 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> void heap_sort(K[] field,Comparator<K> c) {
      Heap<K> a = new Heap<K>(field.length);
      for(int i = 0;i < field.length;++i)
            a.insert(field[i],c);
      for(int i = 0;i < field.length;++i)
            field[field.length - i - 1] = a.remove(c);
}
```

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> void heap_sort(K[] field,Comparator<K> c) {
    for(int i = ((field.length-1)-1) / 2; i >= 0; --i)
        Heap.downheap(field,c, field.length, i);
    for(int i = field.length-1; i > 0; --i) {
        swap(field, 0, i);
        Heap.downheap(field,c, i, 0);
    }
}
```

Heapsort (Fort.)

- die Klassenmethode downheap der Klasse Heap für das Aufbauen des Heaps ohne neuen Speicher

```
public static <K> void downheap(K[] keys,
                                Comparator<K> c,
                                int iEnd,
                                int iIndex) {
    K k = keys[iIndex];
    while (iIndex < iEnd / 2) {
        int iSon = 2 * iIndex + 1;
        if (iSon < iEnd-1 && c.compare(keys[iSon],keys[iSon + 1]) < 0)
            ++iSon;
        if (!(c.compare(k,keys[iSon]) < 0))
            break;
        keys[iIndex] = keys[iIndex];
        iIndex = iSon];
    }
    keys[iIndex] = k;
}
```

Go!

Heapsort (Fort.)

- Vergleich der neuen Klassen- zur alten Objektmethode

```
public static <K> void downheap(K[] keys,
                                Comparator<K> c,
                                int iEnd,
                                int iIndex) {
    K k = keys[iIndex];
    while (iIndex < iEnd / 2) {
        int iSon = 2 * iIndex + 1;
        if (iSon < iEnd-1 &&
            c.compare(  keys[iSon],
                        keys[iSon + 1]) < 0)
            ++iSon;
        if (!(c.compare(k,keys[iSon]) < 0))
            break;
        keys[iIndex] = keys[iIndex];
        iIndex = iSon];
    }
    keys[iIndex] = k;
}
```

```
private void downheap( int iIndex,
                       Comparator c) {

    int k = m_Keys[iIndex];
    while (iIndex < m_iNext / 2) {
        int iSon = 2 * iIndex + 1;
        if (iSon < m_iNext-1 &&
            c.compare(  m_Keys[iSon],
                        m_Keys[iSon + 1]) < 0)
            ++iSon;
        if (!(c.compare(k ,m_Keys[iSon]) < 0))
            break;
        m_Keys[iIndex] = m_Keys[iSon];
        iIndex = iSon;
    }
    m_Keys[iIndex] = k;
}
```

https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

vordefinierte Sortierverfahren in Java und C++

- in Java:

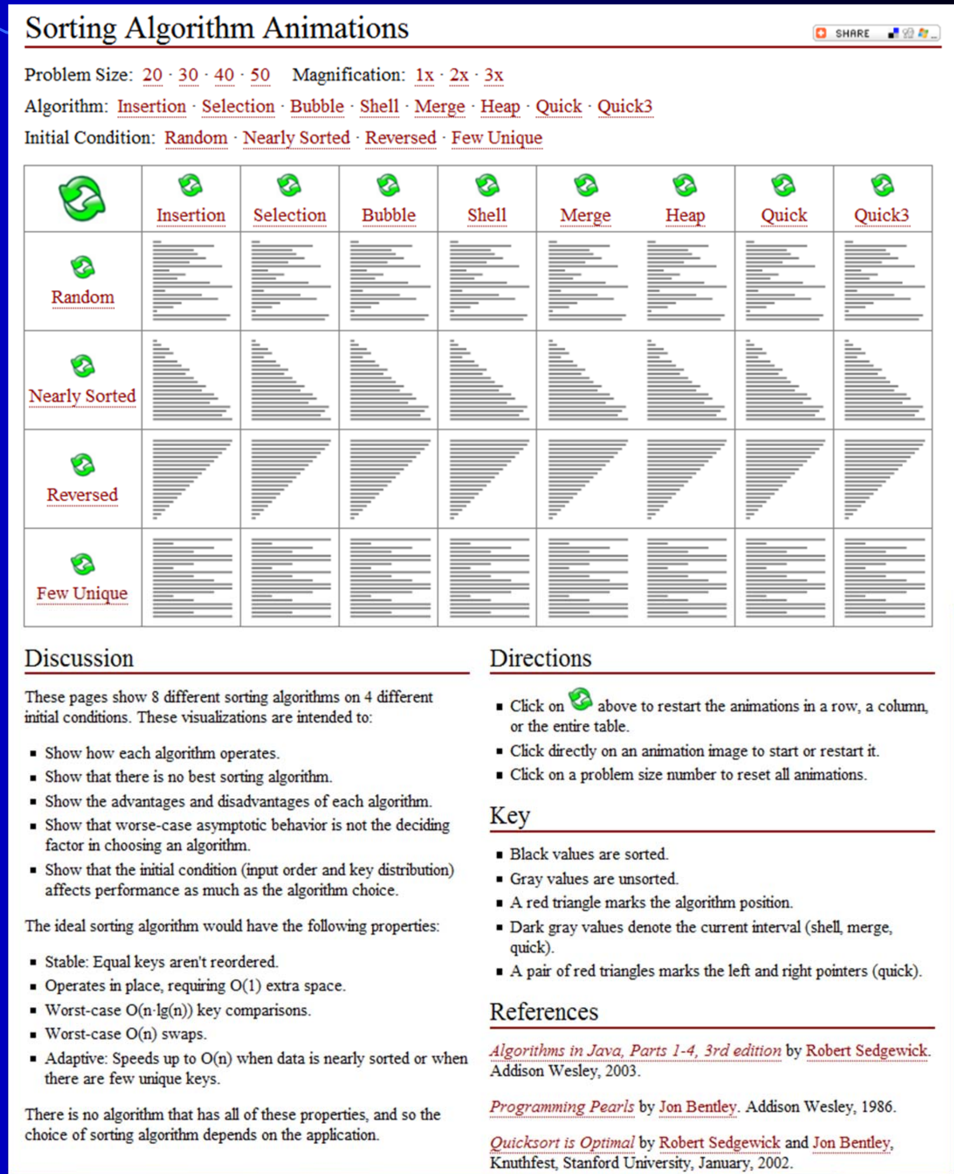
```
class Collection {
    static void sort(List list);
    static void sort(List list, Comparator c);
}

class Array {
    static void sort(byte[] a);
    static void sort(char[] a);
    static void sort(int[] a);
    ...
    static<T> void sort(T[] a,Comparator<? super T> 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

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 7

Vektoren

- in Java wie in C/C++ gilt: Arrays haben eine statische Größe, die zum Instanziierungszeitpunkt angegeben werden muss
- diese Größen lassen sich nicht mehr verändern
- Vektoren haben eine dynamische Größe; sie können während des Programms wachsen und werden durch die Klassen Vector und ArrayList in Java implementiert

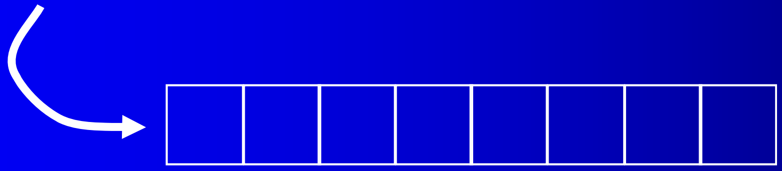
Vektoren (Fort.)

```
public class MyVector {
        public MyVector(int initialCapacity, int capacityIncrement);
        public MyVector(int initialCapacity);
        public MyVector();
        public void push_back(Object obj);
        ...
}
```

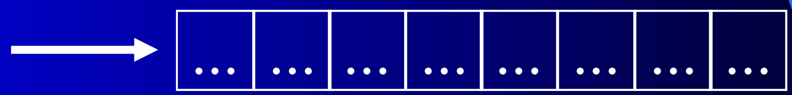
- initialCapacity bestimmt die initiale Größe
- capacityIncrement bestimmt, um wie viele Einheiten der Vektor vergrößert werden soll, wenn er voll ist und ein weiteres Element eingefügt werden soll
- wird capacityIncrement nicht angegeben oder auf 0 gesetzt, wird der Vektor immer verdoppelt

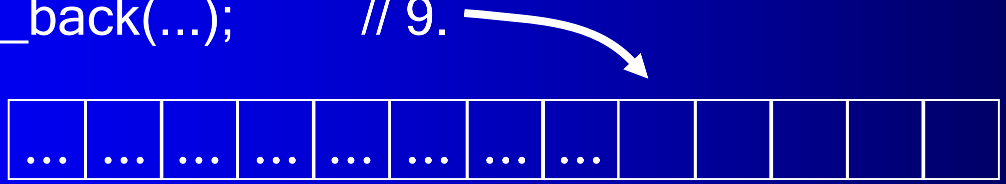
Vektor (Fort.)

```
MyVector vec = new MyVector(8,5);
```



```
vec.push_back(...);     // 1.
vec.push_back(...);     // 2.
vec.push_back(...);     // 3.
vec.push_back(...);     // 4.
vec.push_back(...);     // 5.
vec.push_back(...);     // 6.
vec.push_back(...);     // 7.
vec.push_back(...);     // 8.
vec.push_back(...);     // 9.
```





Vektor (Fort.)

- eigene Klasse entwickeln, um das Konzept zu erkennen

```
public class MyVector {
        public MyVector(int initialCapacity, int capacityIncrement);
        public MyVector(int initialCapacity);
        public MyVector();
        public void push_back(Object obj);
        ...
}
```

- was brauchen wir für Informationen, um
  1. die Objekte zu speichern,
  2. die initiale Kapazität zu speichern,
  3. die Inkrementweite zu speichern,
  4. ein weiteres Element am Ende einzufügen ?

Vektor (Fort.)

1. die Objekte zu speichern
   ein Feld von Objekt: Object[]

2. die initiale Kapazität zu speichern
   wird sich nicht gemerkt, weil man sie nach dem Konstruktor-
   aufruf nie wieder braucht

3. die Inkrementweite zu speichern
   einen einfachen Integerwert, der sich auch nicht mehr
   verändert: int mIncWidth

4. ein weiteres Element am Ende einzufügen
   einen einfachen Integerwert, der immer den Index des
   nächsten freien Elements enthält: int mNextFree

Vektor (Fort.)

```
class MyVector {

    public MyVector(int initialCapacity,int capacityIncrement) {
        mIncWidth = capacityIncrement;
        mNextFree = 0;
        mObjects = new Object[initialCapacity];
    }
    public MyVector(int initialCapacity) {
        this(initialCapacity,0);
    }
    public MyVector() {
        this(1,0);
    }
    public void push_back(Object obj) {...}

    private Object[] mObjects;          // das Feld (Array) der Objekte
    private final int mIncWidth;        // die Weite, um die erweitert werden soll
    private int mNextFree;              // Index des nächsten freien Eintrags
}
```

legt ein Feld mit initialCapacity Elementen an

standardmäßig wird der Vektor immer verdoppelt

Vektor (Fort.)

```
class MyVector {

    public void push_back(Object obj) {
        if (mNextFree >= mObjects.length) {
            resize();
        }
        mObjects[mNextFree++] = obj;
    }
    private void resize() {
        final int newSize = mIncWidth==0
                            ? mObjects.length * 2
                            : mObjects.length + mIncWidth;
        Object[] newObjects = new Object[newSize];
        for(int i = 0;i < mObjects.length;++i) {
            newObjects[i] = mObjects[i];
        }
        mObjects = newObjects;
    }
}
```

???

Ist noch Platz für ein weiteres Element?

Wie ist die neue Größe?

Kopiere die alten Objekte

Was passiert hier?

Go!

Vektor (Fort.)

aktuelle Zeit in Millisekunden

```
import java.lang.*;
public class VectorLongTime {
        public static void makeALotInsertion(MyVector vec) {
                long lStart = System.currentTimeMillis();
                for(int i = 0;i < 20000;++i)
                        vec.push_back(i);
                long lEnd = System.currentTimeMillis();
                System.out.println("Zeit in mSec.: " + (lEnd - lStart));
        }
        public static void main(String[] args) {
                MyVector vec1 = new MyVector(1000,1);
                MyVector vec2 = new MyVector(1000,0);
                makeALotInsertion(vec1);
                makeALotInsertion(vec2);
        }
}
```

2 gleichgroße Vektoren mit unterschiedlichen Allokationsstrategien

Vektor (Fort.)

Beobachtung:

- bei 1000 Einträge, beide Vektoren gleichschnell
- bei 5000 Einträge, Vektor mit 1er Inkrement langsamer als Vektor mit Verdopplung
- bei 10000 Einträgen, Vektor mit 1er Inkrement deutlich langsamer als Vektor mit Verdoppelung
- bei 50000 Einträgen, Vektor mit 1er Inkrement nicht mehr im Sekundenbereich (16 Sekunden)

Analyse von MyVector

Untersuchung: 50000 Elemente sollen eingefügt werden

Einfügeoperationen:

- Vektor mit Verdoppelung: 50000 Einfügeoperationen
- Vektor mit 1er Inkrement: 50000 Einfügeoperationen

Fazit: kein Unterschied bei den Einfügeoperationen

Frage: woher kommt der Unterschied?

Analyse von MyVector (Fort.)

Untersuchung: 50000 Elemente sollen eingefügt werden

Kopieroperationen:

- die ersten 1000 Elemente können eingefügt werden, ohne dass die resize Methode aufgerufen wird

Einfügung des 1001 Elementes:

- mit Verdoppelung:
  1000 Kopieroperationen, neue Größe 2000

- Vektor mit 1er Inkrement:
  1000 Kopieroperationen, neue Größe 1001

Analyse von MyVector (Fort.)

Einfügung des 1002 Elementes:
- mit Verdoppelung:
  keine Kopieroperation, da die Größe (2000) ausreicht

- Vektor mit 1er Inkrement:
  1001 Kopieroperation, neue Größe 1002

Einfügung des 1003 Elementes:
- mit Verdoppelung:
  keine Kopieroperation, da die Größe (2000) ausreicht

- Vektor mit 1er Inkrement:
  1002 Kopieroperation, neue Größe 1003

Analyse von MyVector (Fort.)

	Verdoppelung	1er Inkrement
<= 1000	0	0
1001	1000	1000
1002	0	1001
1003	0	1002
...	...	...
2001	2000	2000
2002	0	2001
...	...	...

Analyse von MyVector (Fort.)

Kopieroperationen:

- 1er Inkrement: 1000+1001+1002+...+49999 = 1249475500
- Verdoppelung: 1000+2000+4000+8000+16000+32000 = 63000

Was bedeutet das in der Praxis?

- Annahme: 100000 Kopieroperationen in 1 Millisekunde
  - 1er Inkrement: 12494,75500 mSec = 12,49475500 Sec
  - Verdoppelung: 0,63 mSec = 0,00063 Sec

Analyse von MyVector (Fort.)

Allgemeine Analyse:

- Annahme: die initiale Größe beträgt zum Anfang 1
- Frage: wieviele Einfüge- und Kopieroperationen hat man bei n Einträgen?

- Einfügeoperationen:
  - 1er Inkrement: n-viele
  - Verdoppelung: n-viele

- für beide gilt also eine Komplexität von O(n)

Analyse von MyVector (Fort.)

- Kopieroperationen:
  - 1er Inkrement: $1+2+3+...+n = (n^2+n)/2$
  - Verdoppelung: $1+2+4+8+16+32+...+m+2m = 4m-1$

    mit: $m < n <= 2m$

    damit gilt: $4m-1 < 4n-1$, also maximal $(4n-1)$-viele Kopieroperationen

- Komplexität des Kopierens:
  - 1er Inkrement: $O(n^2)$
  - Verdoppelung: $O(n)$

Analyse von MyVector (Fort.)

- gesamte Komplexität:
  - 1er Inkrement: $O(n^2) + O(n) = O(n^2)$
  - Verdoppelung: $O(n) + O(n) = O(n)$

- für die Praxis bedeutet dies bei 1.000.000 Elementen (Annahme: 100.000 Kopieroperationen pro 1 Millisekunde)

  - 1er Inkrement: 10.000.000 mSec ≈ 166 min
    ≈ 2,7 Stunden

  - Verdoppelung: 10 mSec.

Diskussion

- Eigenschaften:
  - brauchen wenig Speicherplatz, da nur die Elemente gespeichert werden müssen, und ansonsten 2 zusätzliche Integerwerte
    - aktuelle Anzahl der Elemente
    - Inkrementstrategie
  - auf Elemente kann mittels eines Index direkt zugegriffen werden

Nachteile:

- man muss die Inkrementstrategie zum Beginn festlegen
- wählt man die falsche Strategie, verschwendet man Speicher (viele kleine Vektoren bei Verdoppelung) oder der Vektor ist nicht mehr effizient (große Vektoren mit konstantem Inkrement)
- Löschen eines Elements ist aufwendig

Implementierung (Forts.)

```
class OutOfBoundsException extends Exception {}

public class MyVector {
...
    public Object get(int index) throws OutOfBoundsException {
        if (index < size() && 0<= index)
            return mObjects[index];
        else
            throw new OutOfBoundsException();
    }

    public void set(int iIndex,Object obj) throws OutOfBoundsException {
        if (iIndex < size() && 0 <= iIndex)
            mObjects[iIndex] = obj;
        else
            throw new OutOfBoundsException();
    }

    public int size() {
        return mNextFree;
    }
```

eine eigene Fehlerklasse

beim Lesen kann ein Fehler auftreten: Exception

liegt der Index außerhalb des Bereichs, wird eine Exception geworfen

wie viele Elemente sind schon abgespeichert?

Go!

Implementierung (Fort.)

```
...

    public static void main(String[] args) {
        MyVector vec = new MyVector();
        for(int i = -100;i < 100;++i)
            vec.push_back(i);
        try {
            for(int i = 0;i < vec.size();++i)
                System.out.println(i + ": " + vec.get(i));
        } catch (OutOfBoundsException e) {
            System.out.println("Zu weit");
        }
    }
}
```

ein einfacher Test des Vektors

Was wird ausgegeben? Wird eine Exception geworfen?

Go!

Implementierung (Fort.)

Setzen eines Wertes:

```
public static void main(String[] args) {
      MyVector vec = new MyVector();
      for(int i = -100;i < 100;++i)
            vec.add(i);
      try {
            vec.set(100,34653465);
            for(int i = 0;i < vec.size();++i)
                  System.out.println(i + ": " + vec.get(i));
      } catch (OutOfBoundsException e) {
            System.out.println("Zu weit");
      }
}
```

ändert den Vektor **vec** an der Position 100 ab und speichert dort die **34653465**

Go!

Negative Eigenschaften dieser Implementierung

- um allgemein zu bleiben, werden Objekte vom Typ Object gemerkt
- dadurch geht jede Typsicherheit verloren, Negativbeispiel:

```
public class Vector4 {
    public static void main(String[] args) {
        MyVector vec = new MyVector();
        vec.add(2);
        vec.add(4.2);
        try {
            System.out.println(vec.get(0));
            System.out.println(vec.get(1));
            Integer i1 = (Integer)vec.get(0);
            Integer i2 = (Integer)vec.get(1);
        } catch (OutOfBoundsException e) {}
    }
}
```

Autoboxing: int → Integer

Autoboxing: double → Double

expliziter Typcast

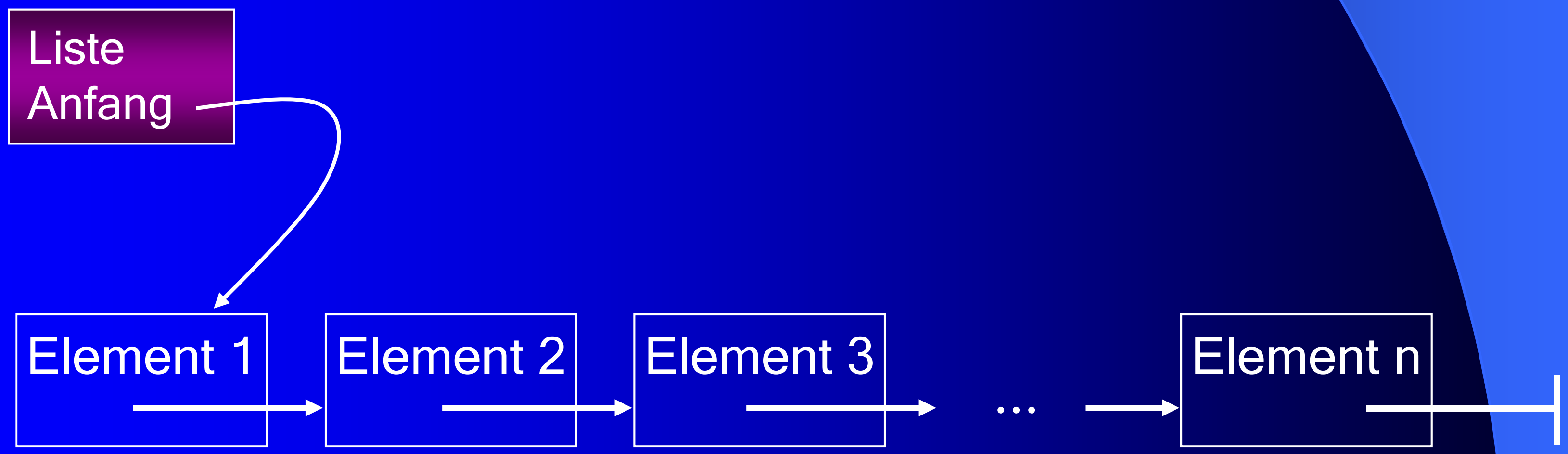
Vorlesung 8

Listen (mit Generics)

braucht man nicht den direkten Zugriff mittels Index, können Listen verwendet werden

Idee:

- einzelne Elemente werden einzeln gespeichert
- jedes Element enthält einen Verweis auf das nächste Element



Listen (Fort.)

Vorteile:

- die Inkrementstrategie muss nicht festgelegt werden
- jede Einfügeoperation ist gleichschnell

Nachteile:

- braucht mehr Speicher, da alle Verweise auch gespeichert werden müssen
- kein direkter Zugriff auf die einzelnen Elemente mehr möglich
- Fragmentierung des Speichers

Listen Implementierung

```
class SingleList<T> {

    class ListElem {

        public ListElem(T obj,ListElem next) {
            m_Next = next;
            m_Elem = obj;
        }

        public T getElement() {
            return m_Elem;
        }

        public ListElem getNext() {
            return m_Next;
        }

        private ListElem    m_Next;
        private T           m_Elem;
    }
...
```

ListElem implementiert ein Listenelement

Konstruktor erwartet das zu speichernde Element und den Verweis auf das nächste Listenelement

ein Listenelement merkt sich das Element und den Verweis auf das nächste Listenelement

Listen Implementierung (Fort.)

```
...
    public SingleList() {
        m_Head = null;
    }

    public void print() {
        for(ListElem elem = m_Head; elem != null;elem = elem.getNext())
            System.out.print(elem.getElement() + "\t");
    }

    private ListElem m_Head; // Liste Anfang

...
}
```

der Listenkonstruktor erwartet kein Argument

zum Drucken müssen alle Listenelemente durchlaufen werden

eine Liste merkt sich nur das erste Listenelement

Listen Implementierung (Fort.)

ein Element wird eingefügt

```
...

    public void push_front(T obj) {
        m_Head = new ListElem(obj, m_Head);
    }

...
```

was passiert hier?

Go!

Listen Implementierung (Fort.)

Die Anwendung der Liste:

```
...

public class List1 {
    public static void main(String[] args) {
        SingleList<Integer> sl = new SingleList<Integer>();
        for(int i = -100;i < 100;++i) {
            sl.push_front(i);
        }
        sl.print();
    }
}
```

legt eine Liste sl an

fügt 200 Elemente in die Liste ein

druckt die List aus

Listen Implementierung (Fort.)

Beobachtungen:

- Implementierung ist (deutlich) einfacher als die des Vektors
- Einfügung erfolgt am Anfang einer Liste, daher werden die Elemente in umgekehrter Reihenfolge ausgelesen

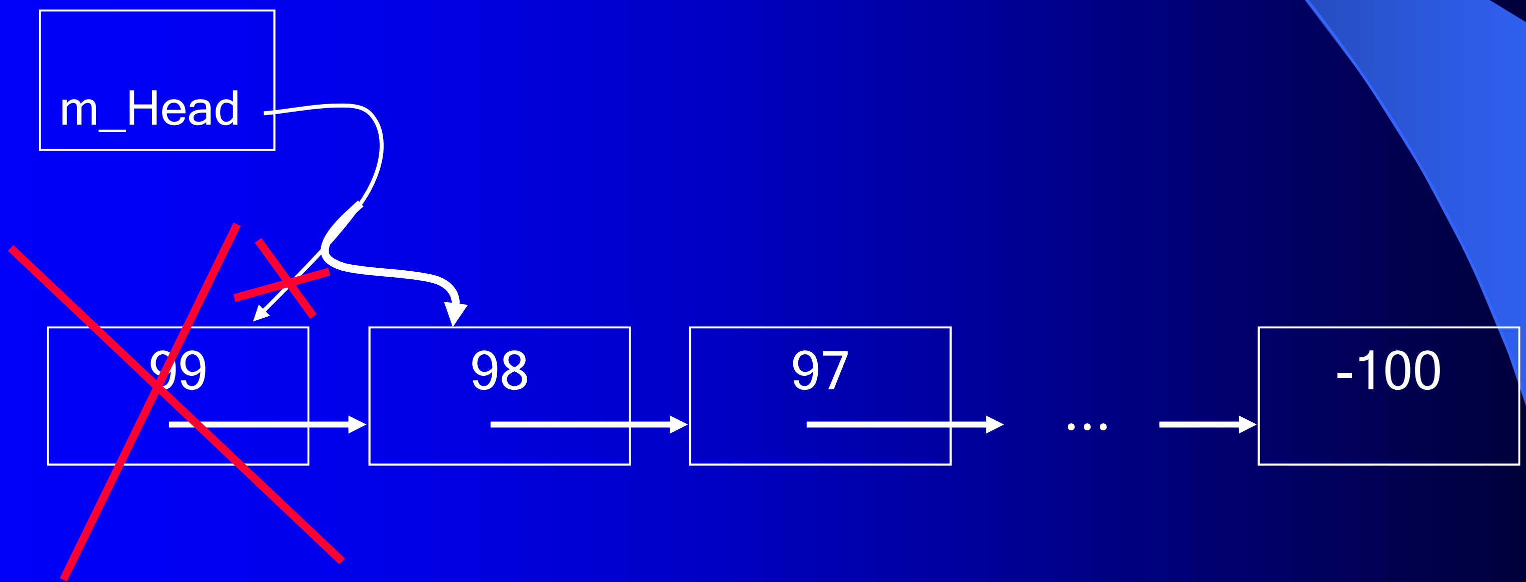
Löschen eines Elements

- die Elemente werden mittels des Objekts/Pointers ListElem angesprochen, identifiziert
- eine Methode zum Löschen eines Elements benötigt somit das/den zu löschenden ListElem Objekt/Pointer
- 3 Situationen sind zu unterscheiden:

1. das zu löschende Element ist nicht in der Liste
2. das zu löschende Element ist das erste Element in der Liste (m_Head zeigt auf dieses Element)
3. das zu löschende Element ist irgendwo in der Liste vorhanden

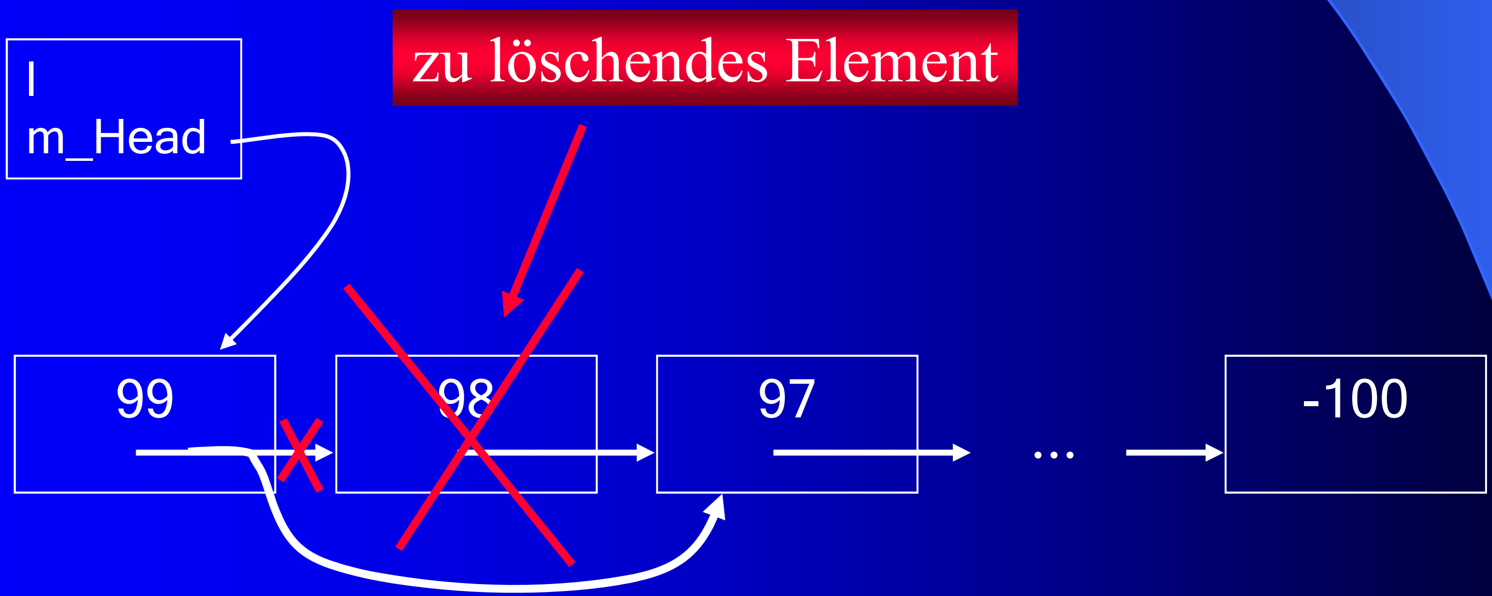
Löschen eines Elements (Fort.)

- ad 1: es ist nichts zu tun
- ad 2: m_Head ist auf den nächsten Eintrag zu setzen und der alte m_Head ist zu löschen



Löschen eines Elements (Fort.)

- ad 3: in der Liste solange suchen, bis der zu löschende Eintrag gefunden ist, sich den Vorgänger merken, beim Vorgänger den Verweis auf den nächsten Eintrag verändern, das zu löschende Element löschen



Löschen eines Elements (Fort.)

- zweiter Schritt: Löschen eines einzelnen Elements
- dazu die Methode void delete(ListElem) in der Klasse List einführen

```
void delete(ListElem pElem2Delete) {
    if (pElem2Delete != null) {
        if (m_Head == pElem2Delete) {
            m_Head = pElem2Delete.getNext();
        } else {
...
        }
    }
}
```

soll überhaupt ein Element gelöscht werden?

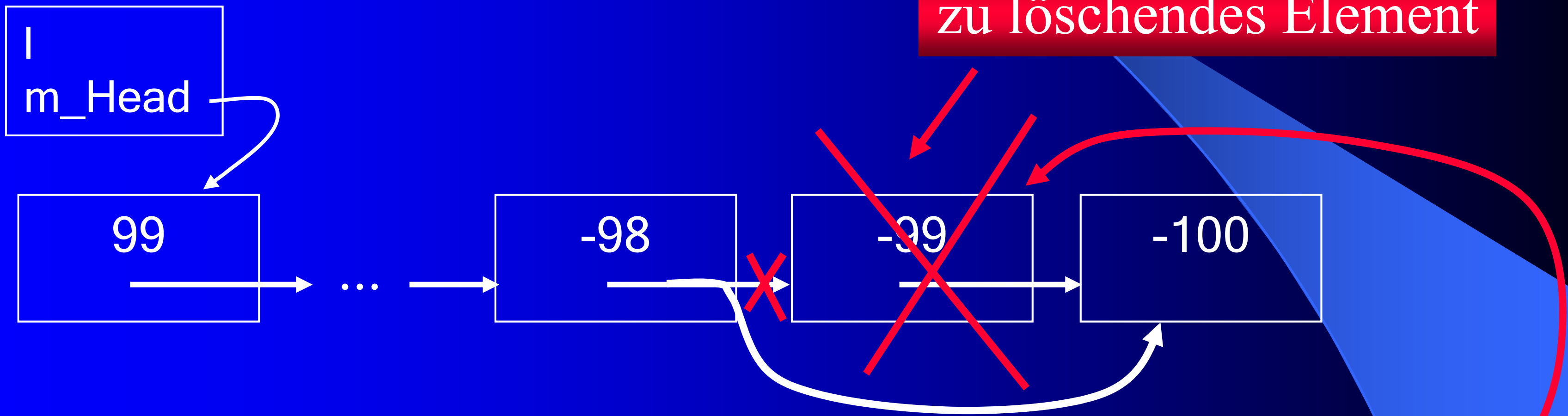
1. Situation: das Startelement soll gelöscht werden

verschiebe Start auf 2. Element

Löschen eines Elements (Fort.)

- Löschen in der Mitte der Liste:
- erster Ansatz: suchen des zu löschenden Elements

zu löschendes Element



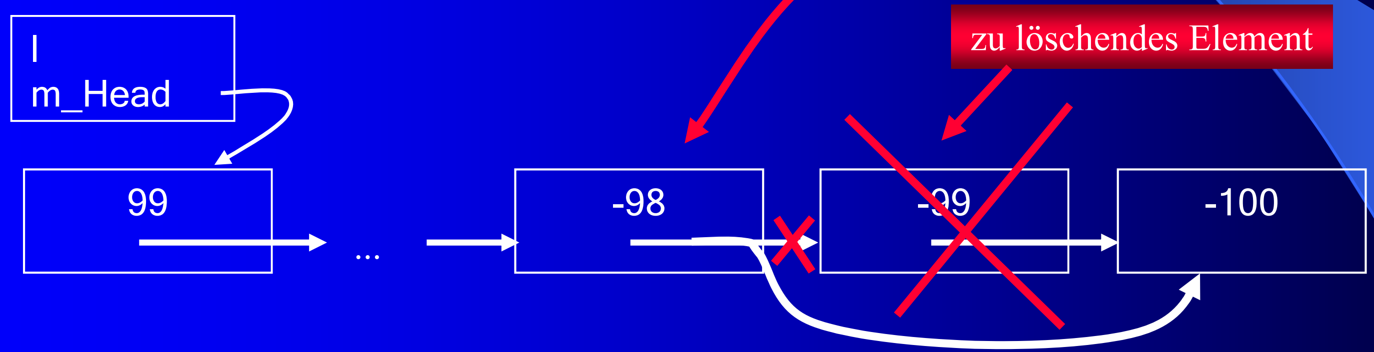

```
for(ListElem tmp = m_Head;  tmp  != null;  tmp = tmp.getNext()) {
            if (tmp == pElem2Delete) {
...
            }
}
```

jetzt zeigt tmp auf das zu löschende Element; es gibt keinen Zugriff auf den Vorgänger

Löschen eines Elements (Fort.)

Lösung:

- nicht überprüfen, ob das aktuelle Element zu löschen ist
- überprüfen, ob das nächste Element zu löschen ist
- wenn ja, dann ist das aktuelle Element der Vorgänger vom zu löschenden Element




```
for(ListElem tmp = m_Head;tmp != null;tmp = tmp.getNext()) {
            if (tmp.getNext() == pElem2Delete) {
...
            }
}
```

prüfen, ob der Nachfolger von tmp zu löschen ist

Go!

Löschen eines Elements (Fort.)

```
void delete(ListElem pElem2Delete) {
    if (pElem2Delete != null) {
        if (m_Head == pElem2Delete) {
            m_Head = pElem2Delete.getNext();
        } else {
            for(ListElem tmp = m_Head;tmp != null;tmp = tmp.getNext()) {
                if (tmp.getNext() == pElem2Delete) {
                    tmp.setNext(pElem2Delete.getNext());
                    return;
                }
            }
        }
    }
}
```

Ist die Liste leer ?

Umsetzen des Zeigers

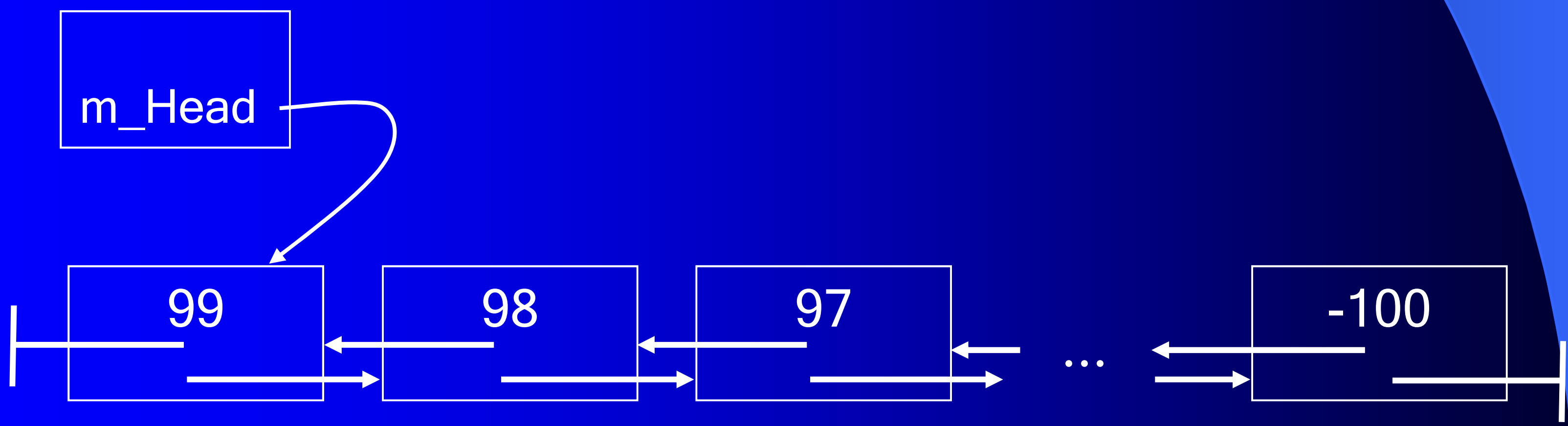
Verlassen der Methode

Analyse der einfach verketteten Liste

- braucht (relativ) wenig zusätzlichen Speicher für die Verwaltung (1 Zeiger pro Element)
- Einfügen am Anfang recht einfach
- Einfügen am Ende auch recht einfach; braucht einen zusätzlichen Verweis pro Liste (siehe Aufgabe zur Stack/Queue)
- Löschen sehr aufwendig, weil:
  - der Vorgänger verändert werden muss
  - kein direkter Zugriff auf den Vorgänger besteht
  - daher die Liste immer durchsucht werden muss
- mögliche Lösung: auch Vorgänger merken

Doppelt verkettete Listen

- eine doppelt verkette Liste merkt sich
  - nächstes Element
  - vorheriges Element
- damit ist der Verwaltungsaufwand größer; 2 Pointer pro Element
- doppelt so großer Verwaltungsaufwand (in Form von Speicher) wie bei einfach verketten Listen



Doppelt verkettete Listen (Fort.)

```
class DoubleList<T> {
    class ListElem {
        public ListElem(T obj,ListElem next,ListElem prev) {
            m_Next = next;
            m_Prev = prev;
            m_Elem = obj;
            if (m_Next != null)
                m_Next.setPrev(this);
            if (m_Prev != null)
                m_Prev.setNext(this);
        }
        public T getElement() {...}
        public ListElem getNext() {...}
        public ListElem getPrev() {...}
        public void setNext(ListElem next) {...}
        public void setPrev(ListElem prev) {...}

        private ListElem     m_Next;
        private ListElem     m_Prev;
        private T            m_Elem;
    }
```

wenn es das nächste Element gibt, dann bin ich der Vorgänger von meinem nächsten

wenn es das vorherige Element gibt, dann bin ich der Nachfolger von meinem vorherigen

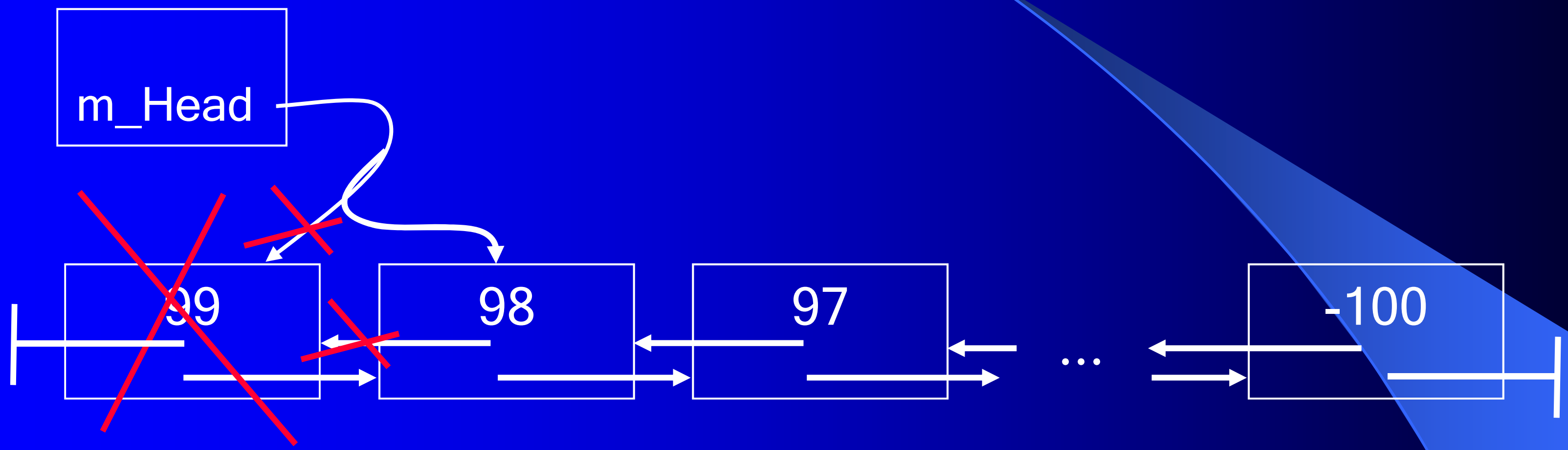
m_Next ist Nachfolger

m_Prev ist Vorgänger

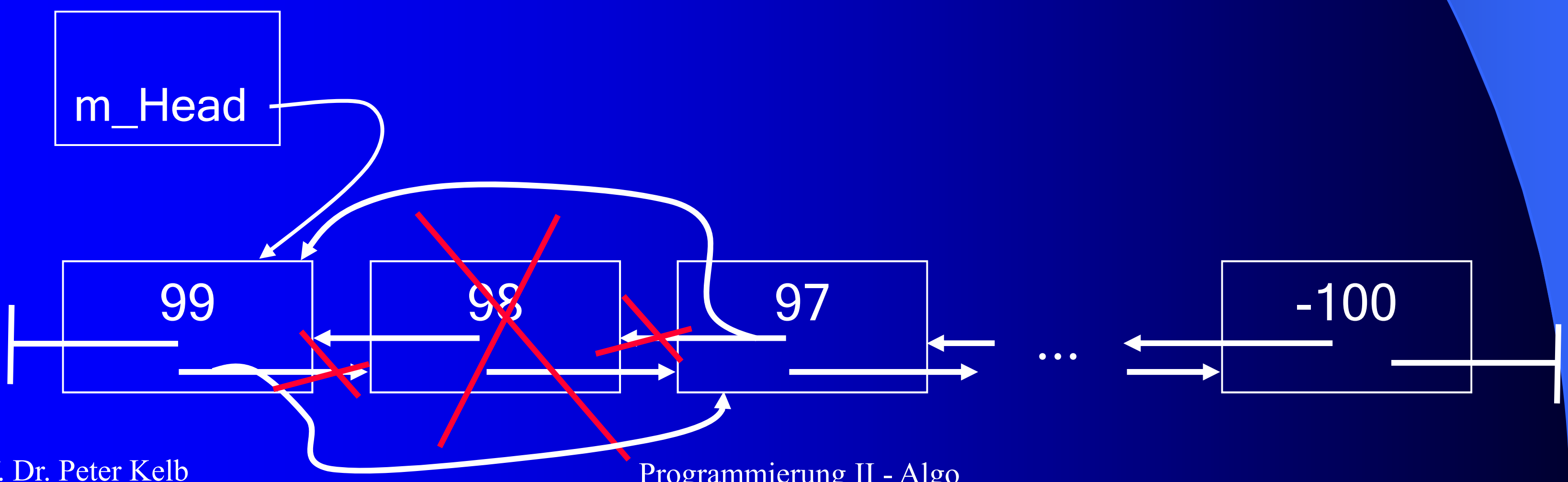
Löschen in einer doppelt verketteten Liste

2 Situationen:

1. der Kopf wird gelöscht



2. ein Element in der Liste gelöscht



Go!

Doppelt verkettete Listen (Fort.)

soll der Start gelöscht werden?

```
void delete(ListElem elem2Delete) {
    if (elem2Delete != null) {
        if (m_Head == elem2Delete)
            m_Head = elem2Delete.getNext();

        if (elem2Delete.getPrev() != null)
            elem2Delete.getPrev().setNext(elem2Delete.getNext());

        if (elem2Delete.getNext() != null)
            elem2Delete.getNext().setPrev(elem2Delete.getPrev());
    }
}
```

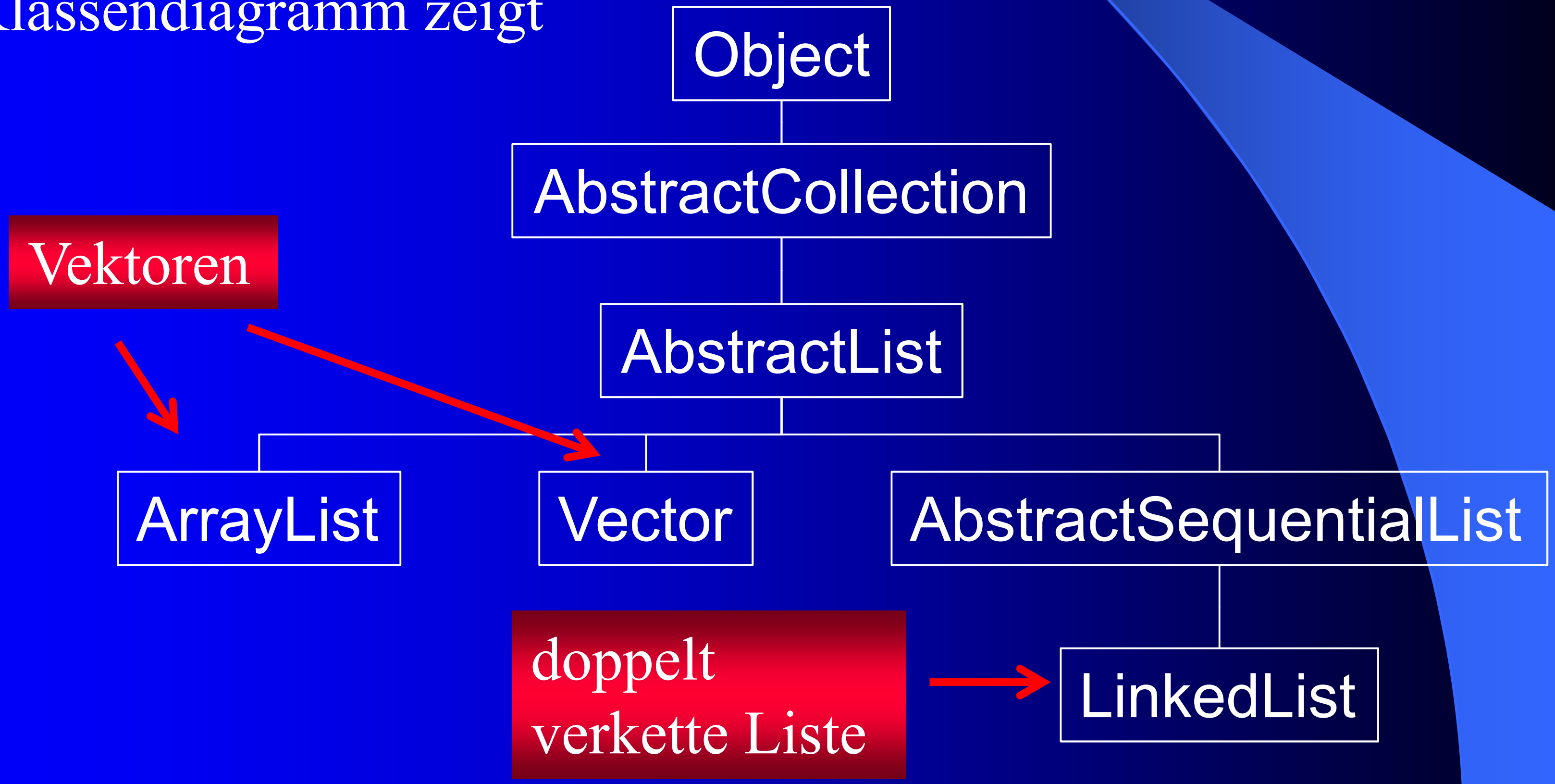
wenn es einen Vorgänger gibt, hänge dessen Nachfolger um

wenn es einen Nachfolger gibt, hänge dessen Vorgänger um

Vorlesung 9

Listen und Vectoren in Java

- selbstverständlich müssen Vektoren und Listen nicht selber implementiert werden
- in Java gibt es bereits Klassen dafür, wie das folgende Klassendiagramm zeigt



ArrayList (Java‘s Name für Vektoren)

- die Klasse Vektor ist älter als die Klasse ArrayList
- sie wurde in der Version 1.2 geändert, um in die Ableitungshierarchie eingepasst zu werden
- sie ist synchronisiert, sprich threadsicher, dadurch ist sie aber langsamer als die ArrayList Implementierung
- ist Multithreading nicht notwendig, sollte daher die ArrayList Implementierung verwendet werden
- statt push_back gibt es die add Methoden

fügt obj am Ende ein

```
public boolean add(T obj);
public void add(int index, T obj);
```

fügt obj vor Position index ein

ArrayList: Beispiel

Go!

```
import java.util.ArrayList;

public class ArrayList_Beispiel1 {

    static <T> void print(ArrayList<T> vec) {
        for(int i = 0;i < vec.size();++i)
            System.out.print(vec.get(i) + "\t");
        System.out.println();
        for(int i = 0;i < vec.size();++i)
            System.out.print(i + "\t");
        System.out.println();
        System.out.println();
    }

    public static void main(String[] args) {
        ArrayList<Integer> vec = new ArrayList<>();
        for(int i = -5;i < 5;++i)
            vec.add(i);
        print(vec);
        vec.set(0,42);
        print(vec);
        vec.add(0,-345678);
        print(vec);
    }
}
```

druckt eine beliebige ArrayList aus

fügt 10 Elemente in die ArrayList ein

verändert das 1. Element

fügt ein neues Element ganz am Anfang hinzu

ArrayList: add Methode

- aus den eigenen Überlegungen wissen wir, dass add(T obj) eine Laufzeitkomplexität von O(1) hat
- um an einer beliebigen Stelle i in einem Vektor etwas einzufügen, müssen zunächst alle Elemente >i um eine Stelle nach hinten verschoben werden
- das ist ein Aufwand von O(n)
- hierbei ist n die aktuelle Größe des Vektors

Komplexität:
O(1)

```
public boolean add(T obj);
```

```
public void add(int index, T obj);
```

Komplexität:
O(size() – index + 1)

Go!

ArrayList: Beispiel

```
import java.util.ArrayList;

public class ArrayList_Beispiel2 {

    static void test(int count,boolean addFront) {
        ArrayList<Integer> vec = new ArrayList<>();
        long lStart = System.currentTimeMillis();
        for(int i = 0;i < count;++i) {
            vec.add(addFront ? 0 : vec.size(),i);
        }
        long lEnd = System.currentTimeMillis();
        System.out.println(    "\t" + (addFront ? "vorne" : "hinten") +
                               ": Zeit in mSec.: " + (lEnd - lStart));
    }

    public static void main(String[] args) {
        for(int i = 50000;i < 1000000;i += 50000) {
            System.out.println(i + " viele Elemente einfügen");
            test(i,false);
            test(i,true);
        }
    }
}
```

Einfügen am Anfang ...

... oder vor dem Ende

in 50.000 Schritten die Anzahl der Einfüge-operationen vergrößern

LinkedList

- die LinkedList ist eine doppelt verkettete Liste
- damit kann an jedes Element in konstanter Zeit „O(1)“ gelöscht werden
- auch die LinkedList besitzt die beiden add Methoden, die auch die ArrayList besitzt

fügt obj am Ende ein

```
public boolean add(T obj);
public void add(int index, T obj);
```

fügt obj vor Position index ein

Go!

LinkedList: Beispiel

```
import java.util.LinkedList;

public class LinkedList_Beispiel1 {

    static <T> void print(LinkedList<T> list) {
        for(int i = 0;i < list.size();++i)
            System.out.print(list.get(i) + "\t");
        System.out.println();
        for(int i = 0;i < list.size();++i)
            System.out.print(i + "\t");
        System.out.println();
        System.out.println();
    }

    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        for(int i = -5;i < 5;++i)
            list.add(i);
        print(list);
        list.set(0,42);
        print(list);
        list.add(0,-345678);
        print(list);
    }
}
```

ArrayList ist durch LinkedList ausgetauscht worden

fügt 10 Elemente in die LinkedList ein

verändert das 1. Element

fügt ein neues Element ganz am Anfang hinzu

LinkedList: add Methode

- genau wie bei der ArrayList (Vektor) kann bei der LinkedList am Ende mittels add(obj) in konstanter Zeit „O(1)“ eingefügt werden
- jedoch kann anders als bei Vektoren auch am Anfang mittels add(0,obj) in O(1) eingefügt werden
- liegt der Index in der Mitte „l.add(l.size() / 2,obj)“, ist die Komplexität ebenfalls O(n)

```
public boolean add(T obj);
```

Komplexität: O(1)

```
public void add(int index, T obj);
```

Komplexität: O(index)

Go!

LinkedList: Beispiel

vorne

hinten

in der Mitte

```
import java.util.LinkedList;

public class LinkedList_Beispiel2 {

	static void test(int count,int where) {
		LinkedList<Integer> list = new LinkedList<>();
		long lStart = System.currentTimeMillis();
		for(int i = 0;i < count;++i) {
			switch (where) {
			case 0: list.add(0,i);break;
			case 1: list.add(list.size(),i);break;
			case 2: list.add(list.size()/2,i);break;
			}
		}
		long lEnd = System.currentTimeMillis();
		System.out.println( "\t" + (where == 0 ? "vorne" : where == 1 ? "hinten" : "mitte") +
					": Zeit in mSec.: " + (lEnd - lStart));
	}

	public static void main(String[] args) {
		for(int i = 50000;i < 1000000;i += 50000) {
			System.out.println(i + " viele Elemente einfügen");
			for(int j = 0;j < 3;++j)
				test(i,j);
		}
	}
}
```

https://docs.oracle.com/javase/7/docs/api/java/util/List.html

Das List Interface

- sowohl die ArrayList (Vektor) als auch die LinkedList implementieren das List Interface
- dieses List Interface beinhaltet neben den add Methoden auch get und set Methoden
- damit kann man die print Methode aus den beiden Beispielen vereinheitlichen
- sie erwartet nicht mehr ein Objekt der Klassen ArrayList<T> bzw. LinkedList<T>, sondern ein Objekt einer Klasse, die das List<T> Interface implementiert

Go!

Das List Interface: Beispiel

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ListInterface_Beispiel1 {

    static <T> void print(List<T> l) {
        for(int i = 0;i < l();++i)
            System.out.print(l.get(i) + "\t");
        System.out.println();
        for(int i = 0;i < l.size();++i)
            System.out.print(i + "\t");
        System.out.println();
        System.out.println();
    }

    static void fillNprint(List<Integer> l) {
        for(int i = -5;i < 5;++i)
            l.add(i);
        print(l);
        l.set(0,42);
        print(l);
        l.add(0,-345678);
        print(l);
    }

    public static void main(String[] args) {
        ArrayList<Integer> vec = new ArrayList<>();
        LinkedList<Integer> list = new LinkedList<>();
        fillNprint(vec);
        fillNprint(list);
    }
}
```

erwartet irgendetwas, das das List Interface implementiert

erwartet irgendetwas, das das List<Integer> Interface implementiert, also eingeschränkter als die print Methode

funktioniert für LinkedList und ArrayList

Das List Interface (Forts.)

- das große Problem mit dem List Interface ist, dass es nicht effizient sowohl von LinkedList als auch von ArrayList implementiert werden kann
- die get Methode ist in ArrayList effizient, da in O(1), in LinkedList in O(n)
- hingegen ist der add(0,obj) Methodenaufruf (Einfügen am Anfang) für die ArrayList in O(n) während es für die LinkedList in O(1) ist
- daraus folgt, dass bei dem Einsatz des List Interfaces im Grunde schon gewusst werden muss, mit welcher konkreten Klasse das List Interface abgefüllt wird

Dies wiederspricht dem Konzept der Datenabstraktion !!!

Go!

Das List Interface: Beispiel

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ListInterface_Beispiel2 {

    static void fill(List<Integer> l,int count) {
        long lStart = System.currentTimeMillis();
        for(int i = 0;i < count;++i)
            l.add(i);
        long lEnd = System.currentTimeMillis();
        System.out.println("\t" + "Zeit für Füllen in mSec.: " + (lEnd - lStart));
    }

    static void sum(List<Integer> l) {
        int dummy = 0;
        long lStart = System.currentTimeMillis();
        for(int i = 0;i < l.size();++i)
            dummy += l.get(i);
        long lEnd = System.currentTimeMillis();
        System.out.println("\t" + "Zeit für Aufsummieren in mSec.: " + (lEnd - lStart));
    }

    public static void main(String[] args) {
        for(int i = 50000;i < 10000000;i += 50000) {
            ArrayList<Integer> vec = new ArrayList<>();
            LinkedList<Integer> list = new LinkedList<>();
            System.out.println("Größe " + i);
            fill(vec,i);
            fill(list,i);
            sum(vec);
            sum(list);
        }
    }
}
```

füllen mit count-vielen Elementen;
Komplexität: $n \times O(1) = O(n)$

Zugriff über get
Komplexität:
- $n \times O(1) = O(n)$ für ArrayList
- $n \times O(n) = O(n^2)$ fürLinkedList

https://docs.oracle.com/javase/7/docs/api/java/util/List.html

Das Iterator Interface

- die Lösung für das vorangegangene Problem sind sogenannte Iteratoren
- Iteratoren sind Objekte, die einen abstrakten Zugriff auf die Elemente eines Containers (Listen oder Vektoren oder ...) darstellen
- die Iteratoren wissen dann selber, wie dann effizient auf das nächste Element zugegriffen werden kann
- das Vorgehen sieht wie folgt aus:

1. der Container gibt den Startiterator (mit dem Verweis auf das erste Element)

Dies wiederspricht dem Konzept der Datenabstraktion !!!

Go!

Das Iterator Interface: Beispiel

```
import java.util.*;

public class Iterator_Beispiel {
	static void fill(List<Integer> l,int count) {...}

	static void sum(List<Integer> l) {
		int dummy = 0;
		long lStart = System.currentTimeMillis();
		Iterator<Integer> i = l.iterator();
		while(i.hasNext())
			dummy += i.next();
		long lEnd = System.currentTimeMillis();
		System.out.println("\t" + "Zeit für Aufsummieren in mSec.: " + (lEnd - lStart));
	}
	public static void main(String[] args) {
		for(int i = 50000;i < 1000000;i += 50000) {
			ArrayList<Integer> vec = new ArrayList<>();
			LinkedList<Integer> list = new LinkedList<>();
			System.out.println("Größe " + i);
			fill(vec,i);
			fill(list,i);
			sum(vec);
			sum(list);
		}
	}
}
```

Zugriff über Iteratoren

- l.iterator() liefert den Iterator mit Verweis auf 1. Element
- i.hasNext() fragt, ob der Iterator auf ein gültiges Element verweist
- i.next() liefert den Inhalt des Iterators und schaltet zum nächsten Element weiter

Komplexitätsübersicht

- die Tabelle fasst die Beobachtungen und Überlegungen zusammen

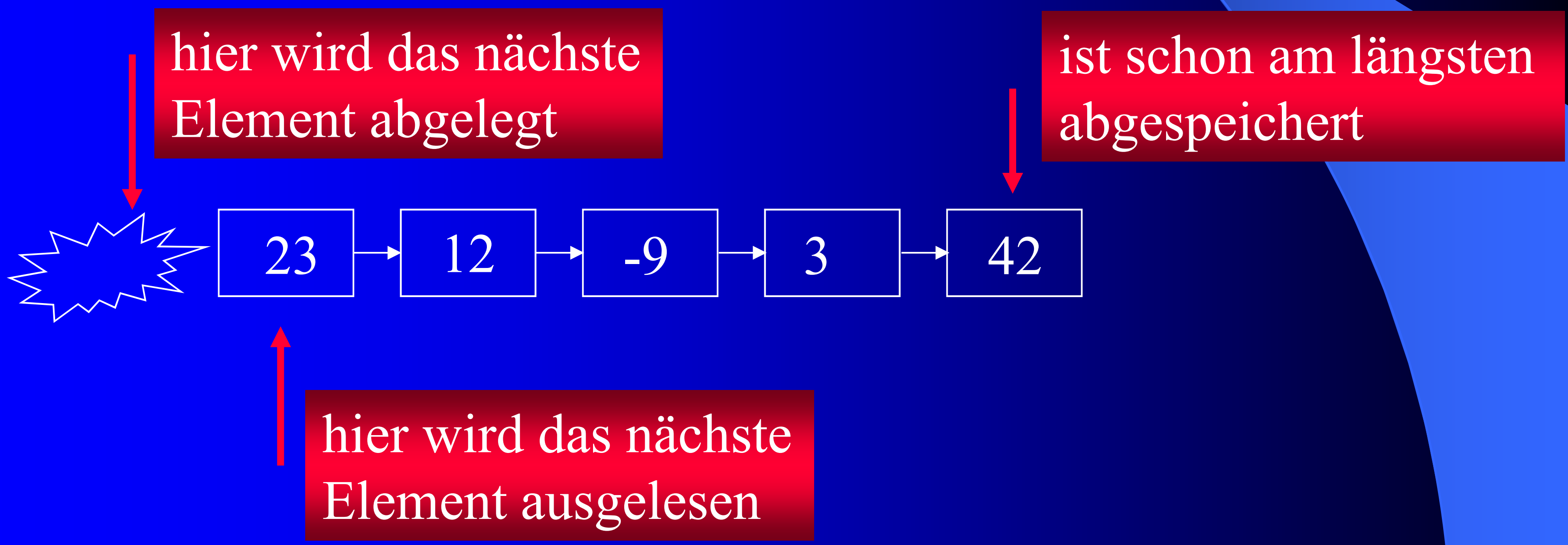
		ArrayList	**LinkedList**
Einfügen	Anfang	O(n)	O(1)
	Mitte	O(n)	O(n)
	Ende	O(1)	O(1)
Löschen	Anfang	O(n)	O(1)
	Mittel	O(n)	O(n)
	Ende	O(1)	O(1)
get/set		O(1)	O(n)
iterieren	Iterator	O(n)	O(n)
	get	O(n)	$O(n^2)$

Stack und Queue

- 2 wichtige Datenstrukturen, die häufig in der Informatik verwendet werden
- Elemente werden sequentiell abgespeichert
- der Zugriff auf die Elemente erfolgt **nicht** beliebig, sondern in bestimmter Form

Stack und Queue (Fort.)

- Stack:
  - last-in-first-out Prinzip (LIFO)
  - ein Element kann abgelegt werden, wird auf den Stack lesend zugegriffen, wird das zuletzt gespeicherte Element zurückgeliefert

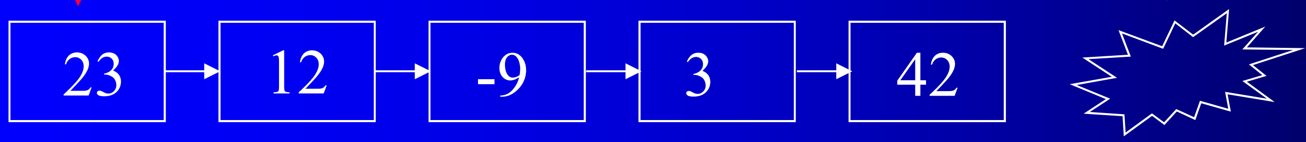


Stack und Queue (Fort.)

- Queue
  - first-in-first-out Prinzip (FIFO)
  - ein Element wird abgelegt, ein lesender Zugriff liefert das Element, dass schon am längsten in der Warteschlange sich befindet

ist schon am längsten abgespeichert

hier wird das nächste Element abgelegt



hier wird das nächste Element ausgelesen

Vordefinierte Implementierung von Stacks und Queues

- in Java: die Klasse (Interface) Stack<T>, Queue<T>
- in C++:

  die Templates std::stack<class T>
  std::queue<class T>

Stack und Queue (Fort.)

- beide Datenstrukturen können das gleiche Interface aufzeigen

Auch Interfaces können Generics enthalten

```
interface StackOrQueue<T> {
        boolean isEmpty();      // ist noch ein Element vorhanden?
        T top();                // liefert das aktuelle Element
        void push(T elm);       // legt ein neues Element ab
        void pop();             // entfernt das aktuelle Element
}
```

- beide Datenstrukturen können mittels einer einfach verketteten Liste implementiert werden
- bei der Queue braucht man zusätzlich zum Head auch ein Tail, um einen schnellen Zugriff auf das letzte Element zu gewährleisten

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

Neue Datensätze (Schlüssel-Daten-Paar) werden am Ende 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 SeqSearch {

    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 eingefügten 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

Schlüssel	-15	-5	3	13	17	34	38	40
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre
	0	1	2	3	4	5	6	7

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 und Löschen ist laufzeitaufwendig O(n)

Vorteil

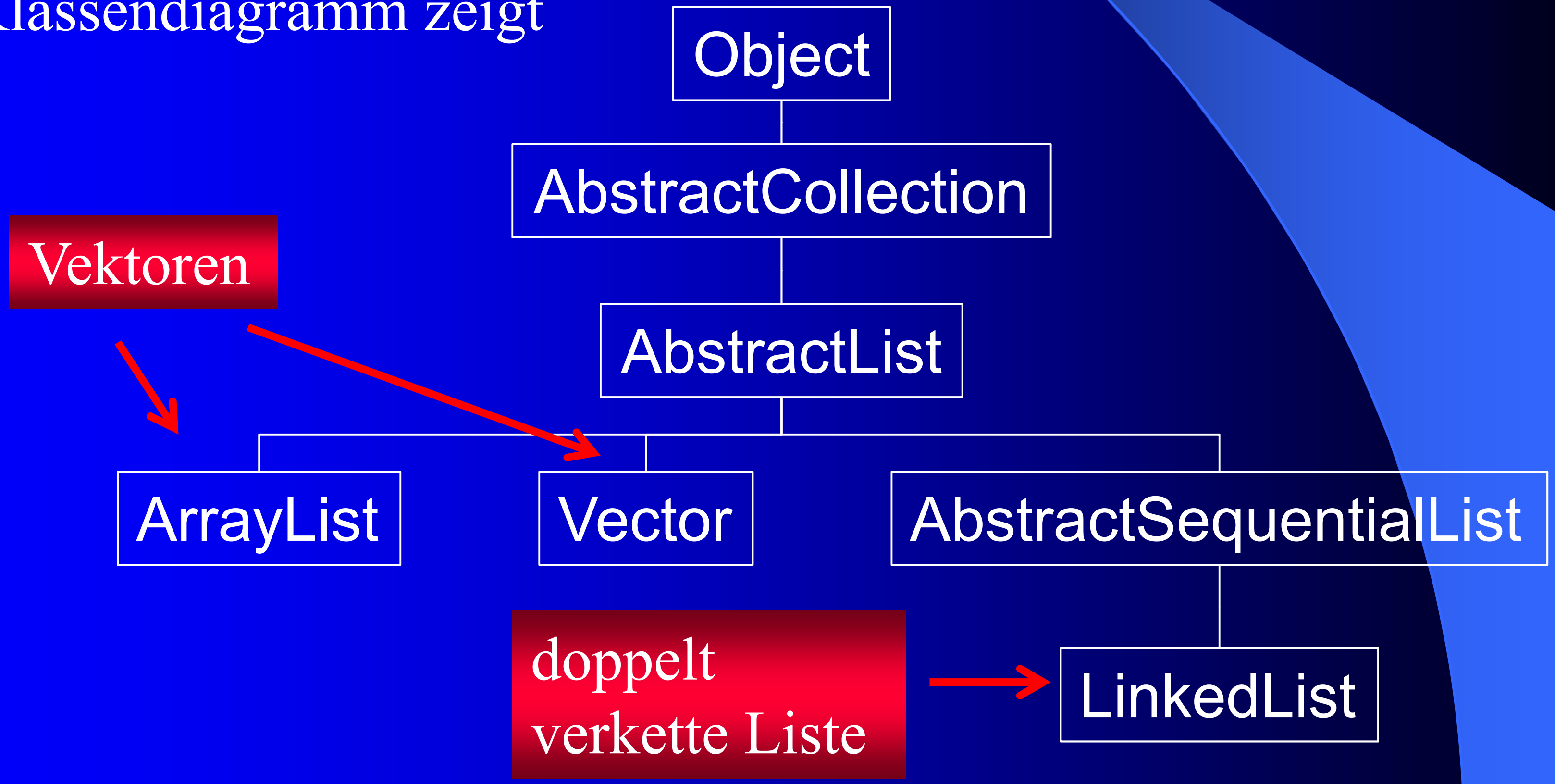
- auch in sehr großen Datenmengen kann noch schnell gesucht werden
- gut geeignet, wenn erst alle Elemente eingefügt werden bevor das erste Element gesucht wird

Warum?

Vorlesung 9

Listen und Vectoren in Java

- selbstverständlich müssen Vektoren und Listen nicht selber implementiert werden
- in Java gibt es bereits Klassen dafür, wie das folgende Klassendiagramm zeigt



ArrayList (Java's Name für Vektoren)

- die Klasse Vektor ist älter als die Klasse ArrayList
- sie wurde in der Version 1.2 geändert, um in die Ableitungshierarchie eingepasst zu werden
- sie ist synchronisiert, sprich threadsicher, dadurch ist sie aber langsamer als die ArrayList Implementierung
- ist Multithreading nicht notwendig, sollte daher die ArrayList Implementierung verwendet werden
- statt push_back gibt es die add Methoden

fügt obj am Ende ein

```
public boolean add(T obj);
public void add(int index, T obj);
```

fügt obj vor Position index ein

Go!

ArrayList: Beispiel

```
import java.util.ArrayList;

public class ArrayList_Beispiel1 {

    static <T> void print(ArrayList<T> vec) {
        for(int i = 0;i < vec.size();++i)
            System.out.print(vec.get(i) + "\t");
        System.out.println();
        for(int i = 0;i < vec.size();++i)
            System.out.print(i + "\t");
        System.out.println();
        System.out.println();
    }

    public static void main(String[] args) {
        ArrayList<Integer> vec = new ArrayList<>();
        for(int i = -5;i < 5;++i)
            vec.add(i);
        print(vec);
        vec.set(0,42);
        print(vec);
        vec.add(0,-345678);
        print(vec);
    }
}
```

druckt eine beliebige ArrayList aus

fügt 10 Elemente in die ArrayList ein

verändert das 1. Element

fügt ein neues Element ganz am Anfang hinzu

ArrayList: add Methode

- aus den eigenen Überlegungen wissen wir, dass add(T obj) eine Laufzeitkomplexität von O(1) hat
- um an einer beliebigen Stelle i in einem Vektor etwas einzufügen, müssen zunächst alle Elemente >i um eine Stelle nach hinten verschoben werden
- das ist ein Aufwand von O(n)
- hierbei ist n die aktuelle Größe des Vektors

Komplexität: O(1)

```
public boolean add(T obj);
```

```
public void add(int index, T obj);
```

Komplexität: O(size() – index + 1)

Go!

ArrayList: Beispiel

```
import java.util.ArrayList;

public class ArrayList_Beispiel2 {

	static void test(int count,boolean addFront) {
		ArrayList<Integer> vec = new ArrayList<>();
		long lStart = System.currentTimeMillis();
		for(int i = 0;i < count;++i) {
			vec.add(addFront ? 0 : vec.size(),i);
		}
		long lEnd = System.currentTimeMillis();
		System.out.println(	"\t" + (addFront ? "vorne" : "hinten") +
					": Zeit in mSec.: " + (lEnd - lStart));
	}

	public static void main(String[] args) {
		for(int i = 50000;i < 1000000;i += 50000) {
			System.out.println(i + " viele Elemente einfügen");
			test(i,false);
			test(i,true);
		}
	}
}
```

Einfügen am Anfang ...

... oder vor dem Ende

in 50.000 Schritten die Anzahl der Einfüge-operationen vergrößern

LinkedList

- die LinkedList ist eine doppelt verkettete Liste
- damit kann an jedes Element in konstanter Zeit „O(1)“ gelöscht werden
- auch die LinkedList besitzt die beiden add Methoden, die auch die ArrayList besitzt

fügt obj am Ende ein

```
public boolean add(T obj);
public void add(int index, T obj);
```

fügt obj vor Position index ein

Go!

LinkedList: Beispiel

```
import java.util.LinkedList;

public class LinkedList_Beispiel1 {

    static <T> void print(LinkedList<T> list) {
        for(int i = 0;i < list.size();++i)
            System.out.print(list.get(i) + "\t");
        System.out.println();
        for(int i = 0;i < list.size();++i)
            System.out.print(i + "\t");
        System.out.println();
        System.out.println();
    }

    public static void main(String[] args) {
        LinkedList<Integer> list = new LinkedList<>();
        for(int i = -5;i < 5;++i)
            list.add(i);
        print(list);
        list.set(0,42);
        print(list);
        list.add(0,-345678);
        print(list);
    }
}
```

ArrayList ist durch LinkedList ausgetauscht worden

fügt 10 Elemente in die LinkedList ein

verändert das 1. Element

fügt ein neues Element ganz am Anfang hinzu

LinkedList: add Methode

- genau wie bei der ArrayList (Vektor) kann bei der LinkedList am Ende mittels add(obj) in konstanter Zeit „O(1)“ eingefügt werden
- jedoch kann anders als bei Vektoren auch am Anfang mittels add(0,obj) in O(1) eingefügt werden
- liegt der Index in der Mitte „l.add(l.size() / 2,obj)“, ist die Komplexität ebenfalls O(n)

Komplexität: O(1)

```
public boolean add(T obj);
```

```
public void add(int index, T obj);
```

Komplexität: O(index)

Go!

LinkedList: Beispiel

```
import java.util.LinkedList;

public class LinkedList_Beispiel2 {

    static void test(int count,int where) {
        LinkedList<Integer> list = new LinkedList<>();
        long lStart = System.currentTimeMillis();
        for(int i = 0;i < count;++i) {
            switch (where) {
            case 0: list.add(0,i);break;
            case 1: list.add(list.size(),i);break;
            case 2: list.add(list.size()/2,i);break;
            }
        }
        long lEnd = System.currentTimeMillis();
        System.out.println( "\t" + (where == 0 ? "vorne" : where == 1 ? "hinten" : "mitte") +
                                    ": Zeit in mSec.: " + (lEnd - lStart));
    }

    public static void main(String[] args) {
        for(int i = 50000;i < 1000000;i += 50000) {
            System.out.println(i + " viele Elemente einfügen");
            for(int j = 0;j < 3;++j)
                test(i,j);
        }
    }
}
```

vorne

hinten

in der Mitte

https://docs.oracle.com/javase/7/docs/api/java/util/List.html

Das List Interface

- sowohl die ArrayList (Vektor) als auch die LinkedList implementieren das List Interface
- dieses List Interface beinhaltet neben den add Methoden auch get und set Methoden
- damit kann man die print Methode aus den beiden Beispielen vereinheitlichen
- sie erwartet nicht mehr ein Objekt der Klassen ArrayList<T> bzw. LinkedList<T>, sondern ein Objekt einer Klasse, die das List<T> Interface implementiert

Go!

Das List Interface: Beispiel

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ListInterface_Beispiel1 {

        static <T> void print(List<T> l) {
                for(int i = 0;i < l();++i)
                        System.out.print(l.get(i) + "\t");
                System.out.println();
                for(int i = 0;i < l.size();++i)
                        System.out.print(i + "\t");
                System.out.println();
                System.out.println();
        }

        static void fillNprint(List<Integer> l) {
                for(int i = -5;i < 5;++i)
                        l.add(i);
                print(l);
                l.set(0,42);
                print(l);
                l.add(0,-345678);
                print(l);
        }

        public static void main(String[] args) {
                ArrayList<Integer> vec = new ArrayList<>();
                LinkedList<Integer> list = new LinkedList<>();
                fillNprint(vec);
                fillNprint(list);
        }
}
```

erwartet irgendetwas, das das List Interface implementiert

erwartet irgendetwas, das das List<Integer> Interface implementiert, also eingeschränkter als die print Methode

funktioniert für LinkedList und ArrayList

Das List Interface (Forts.)

- das große Problem mit dem List Interface ist, dass es nicht effizient sowohl von LinkedList als auch von ArrayList implementiert werden kann
- die get Methode ist in ArrayList effizient, da in O(1), in LinkedList in O(n)
- hingegen ist der add(0,obj) Methodenaufruf (Einfügen am Anfang) für die ArrayList in O(n) während es für die LinkedList in O(1) ist
- daraus folgt, dass bei dem Einsatz des List Interfaces im Grunde schon gewusst werden muss, mit welcher konkreten Klasse das List Interface abgefüllt wird

Dies wiederspricht dem Konzept der Datenabstraktion !!!

Go!

Das List Interface: Beispiel

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ListInterface_Beispiel2 {

	static void fill(List<Integer> l,int count) {
		long lStart = System.currentTimeMillis();
		for(int i = 0;i < count;++i)
			l.add(i);
		long lEnd = System.currentTimeMillis();
		System.out.println("\t" + "Zeit für Füllen in mSec.: " + (lEnd - lStart));
	}

	static void sum(List<Integer> l) {
		int dummy = 0;
		long lStart = System.currentTimeMillis();
		for(int i = 0;i < l.size();++i)
			dummy += l.get(i);
		long lEnd = System.currentTimeMillis();
		System.out.println("\t" + "Zeit für Aufsummieren in mSec.: " + (lEnd - lStart));
	}

	public static void main(String[] args) {
		for(int i = 50000;i < 10000000;i += 50000) {
			ArrayList<Integer> vec = new ArrayList<>();
			LinkedList<Integer> list = new LinkedList<>();
			System.out.println("Größe " + i);
			fill(vec,i);
			fill(list,i);
			sum(vec);
			sum(list);
		}
	}
}
```

füllen mit count-vielen Elementen;
Komplexität: $n \times O(1) = O(n)$

Zugriff über get
Komplexität:
- $n \times O(1) = O(n)$ für ArrayList
- $n \times O(n) = O(n^2)$ fürLinkedList

https://docs.oracle.com/javase/7/docs/api/java/util/List.html

Das Iterator Interface

- die Lösung für das vorangegangene Problem sind sogenannte Iteratoren
- Iteratoren sind Objekte, die einen abstrakten Zugriff auf die Elemente eines Containers (Listen oder Vektoren oder ...) darstellen
- die Iteratoren wissen dann selber, wie dann effizient auf das nächste Element zugegriffen werden kann
- das Vorgehen sieht wie folgt aus:

1. der Container gibt den Startiterator (mit dem Verweis auf das erste Element)

Dies wiederspricht dem Konzept der Datenabstraktion !!!

Go!

Das Iterator Interface: Beispiel

```
import java.util.*;

public class Iterator_Beispiel {
      static void fill(List<Integer> l,int count) {...}

      static void sum(List<Integer> l) {
            int dummy = 0;
            long lStart = System.currentTimeMillis();
            Iterator<Integer> i = l.iterator();
            while(i.hasNext())
                  dummy += i.next();
            long lEnd = System.currentTimeMillis();
            System.out.println("\t" + "Zeit für Aufsummieren in mSec.: " + (lEnd - lStart));
      }
      public static void main(String[] args) {
            for(int i = 50000;i < 1000000;i += 50000) {
                  ArrayList<Integer> vec = new ArrayList<>();
                  LinkedList<Integer> list = new LinkedList<>();
                  System.out.println("Größe " + i);
                  fill(vec,i);
                  fill(list,i);
                  sum(vec);
                  sum(list);
            }
      }
}
```

Zugriff über Iteratoren

- l.iterator() liefert den Iterator mit Verweis auf 1. Element
- i.hasNext() fragt, ob der Iterator auf ein gültiges Element verweist
- i.next() liefert den Inhalt des Iterators und schaltet zum nächsten Element weiter

Komplexitätsübersicht

- die Tabelle fasst die Beobachtungen und Überlegungen zusammen

		ArrayList	**LinkedList**
Einfügen	Anfang	O(n)	O(1)
	Mitte	O(n)	O(n)
	Ende	O(1)	O(1)
Löschen	Anfang	O(n)	O(1)
	Mittel	O(n)	O(n)
	Ende	O(1)	O(1)
get/set		O(1)	O(n)
iterieren	Iterator	O(n)	O(n)
	get	O(n)	$O(n^2)$

Stack und Queue

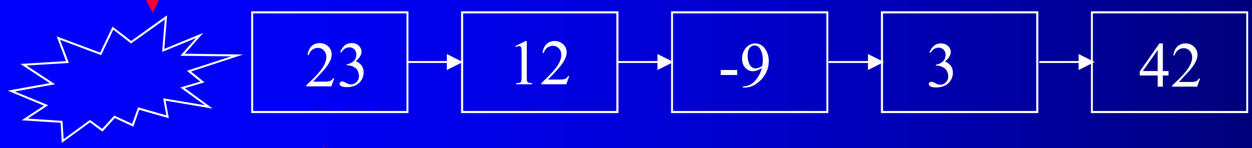
- 2 wichtige Datenstrukturen, die häufig in der Informatik verwendet werden
- Elemente werden sequentiell abgespeichert
- der Zugriff auf die Elemente erfolgt **nicht** beliebig, sondern in bestimmter Form

Stack und Queue (Fort.)

- Stack:
  - last-in-first-out Prinzip (LIFO)
  - ein Element kann abgelegt werden, wird auf den Stack lesend zugegriffen, wird das zuletzt gespeicherte Element zurückgeliefert

hier wird das nächste Element abgelegt

ist schon am längsten abgespeichert



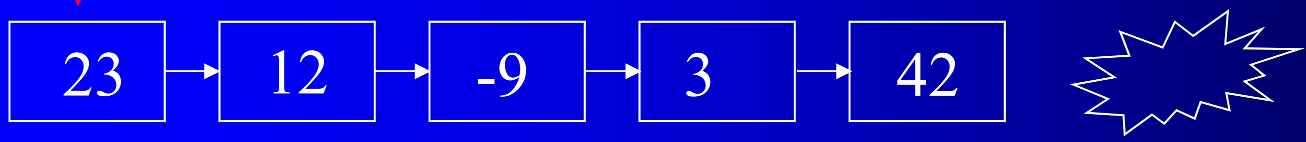
hier wird das nächste Element ausgelesen

Stack und Queue (Fort.)

- Queue
  - first-in-first-out Prinzip (FIFO)
  - ein Element wird abgelegt, ein lesender Zugriff liefert das Element, dass schon am längsten in der Warteschlange sich befindet

ist schon am längsten abgespeichert

hier wird das nächste Element abgelegt



hier wird das nächste Element ausgelesen

Vordefinierte Implementierung von Stacks und Queues

- in Java: die Klasse (Interface) Stack<T>, Queue<T>
- in C++:

  die Templates std::stack<class T>
  std::queue<class T>

Stack und Queue (Fort.)

- beide Datenstrukturen können das gleiche Interface aufzeigen

Auch Interfaces können Generics enthalten

```
interface StackOrQueue<T> {
        boolean isEmpty();      // ist noch ein Element vorhanden?
        T top();                // liefert das aktuelle Element
        void push(T elm);       // legt ein neues Element ab
        void pop();             // entfernt das aktuelle Element
}
```

- beide Datenstrukturen können mittels einer einfach verketteten Liste implementiert werden
- bei der Queue braucht man zusätzlich zum Head auch ein Tail, um einen schnellen Zugriff auf das letzte Element zu gewährleisten

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

Neue Datensätze (Schlüssel-Daten-Paar) werden am Ende 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 SeqSearch {

    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 eingefügten 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

Schlüssel	-15	-5	3	13	17	34	38	40
Datensätze	juhu	toll	super	nie	nein	ja	klar	irre
	0	1	2	3	4	5	6	7

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 und Löschen ist laufzeitaufwendig O(n)

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?

Vorlesung 10

Suchen in binären Suchbäumen

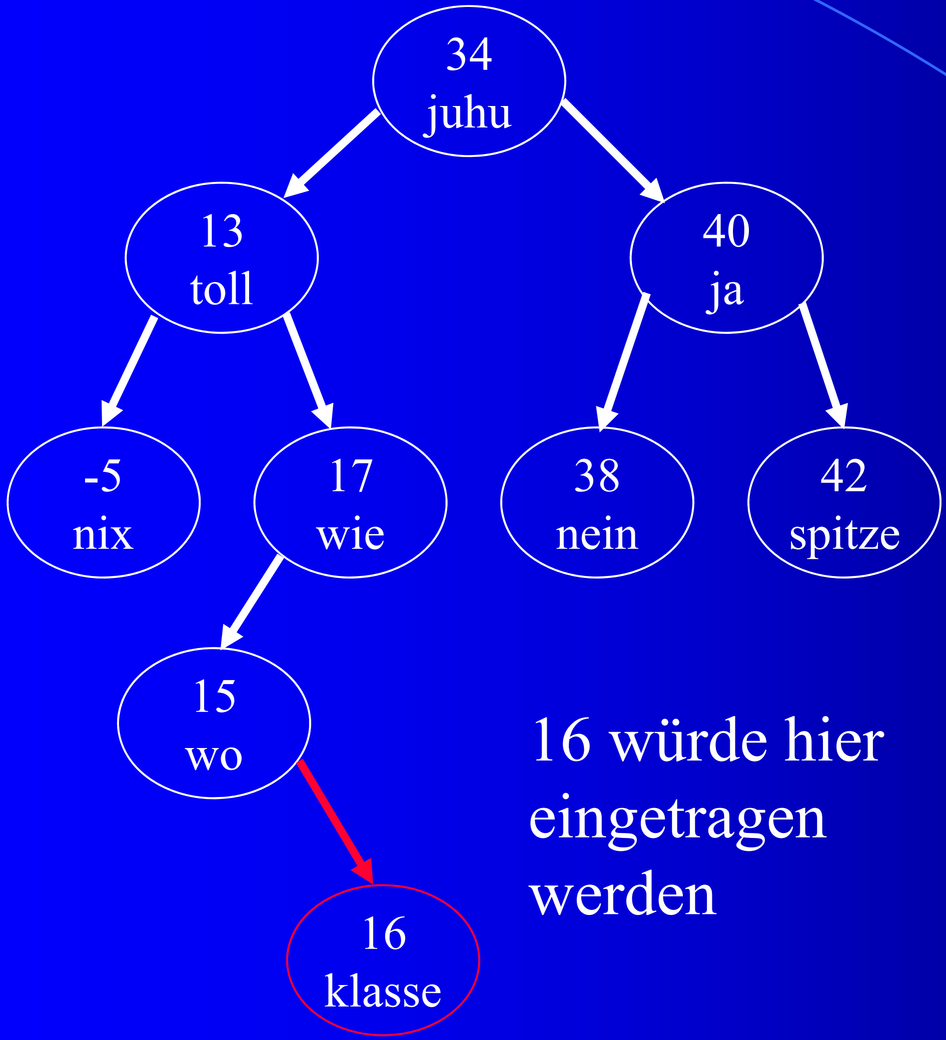
gesucht wird eine Datenstruktur:

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

binärer Suchbaum (vergleiche Vorlesung Graphentheorie):

- 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 Suchbäumen (Fort.)



16 würde hier eingetragen werden

- binärer Suchbaum 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 Suchbäumen: 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 Suchbaum 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 Suchbäumen: 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 Suchbäumen

- Das Einfügen kann auf zwei Arten erfolgen:
  - iterativ
  - rekursiv
- Die rekursive Lösung ist kompakter im Quellcode
- Die iterative Lösung ist schneller in der Ausführung

Einfügen in binären Suchbäumen: rekursiv

- Idee: rekursive Methode erzeugt neuen (Teil-)Baum, der in den bestehenden Baum eingehängt wird
- Rekursionsverankerung:
  - einzufügender Schlüssel ist identisch zu aktuellem Schlüssel → liefere aktuellen Knoten zurück
  - Abstieg ist auf Null-Verweis gelaufen → liefere neuen Knoten mit einzufügenden Schlüssel zurück
- Rekursionsschritt:
  - ersetze linken (resp. rechten) Teilbaum durch das Ergebnis des rekursiven Aufrufs, wenn der einzufügende Schlüssel kleiner (resp. größer) als aktueller Schlüssel ist.

Einfügen in binären Suchbäumen: rekursiv

```
public void insertRec(K key,D data) {
    m_Root = insertRec(m_Root,key,data);
}

Node insertRec(Node n,K key,D data) {
    if (n == null)
        return new Node(key,data);
    else {
        final int RES = key.compareTo(n.m_Key);
        if (RES < 0)
            n.m_Left = insertRec(n.m_Left,key,data);
        else if (RES > 0)
            n.m_Right = insertRec(n.m_Right,key,data);
        return n;
    }
}
```

2. Rekursionsverankerung

Rekursionsschritt

1. Rekursionsverankerung:
kein abschließendes `else`

Einfügen in binären Suchbäumen: iterativ

```
public boolean insert(K key,D data) {
    Node tmp = m_Root;
    Node father = null;
    while (tmp != null) {
        father = tmp;
        final int RES = key.compareTo(tmp.m_Key);
        if (RES == 0)
            return false;
        tmp = RES < 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;
    return true;
}
```

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 Suchbäumen: iterativ (Forts.)

- Das Merken des Vorgängers ist symptomatisch für alle Implementierungen von Bäumen für unterschiedliche Einfüge- und Löschoperationen (siehe Rot-Schwarz Bäume, Patrica Trees, …)
- Daher sollte dies nur einmal implementiert werden
- Hier könnte eine NodeHandler Klasse hilfreich sein, dessen Aufgabe ist es,
  - sich den Vorgänger zu merken
  - selber festzustellen, 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

wird für remove benötigt

... sonst rechts oder links unterhalb des Vaters

Einfügen in binären Suchbäumen 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 Suchbäumen: Komplexität

- Das Einfügen
  - dauert maximal bis zu der Tiefe des Baums

- Das Suchen
  - dauert maximal bis zu der Tiefe des Baums

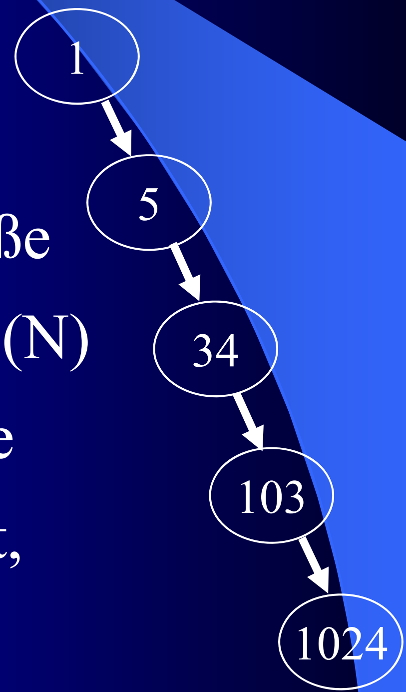
- Die Tiefe des Baums ist minimal Logarithmus der Knotenanzahl, d.h. im Durchschnitt ist das Suchen und Einfügen in O(log N)

Suchen in binären Suchbäumen: Komplexität (Fort.)

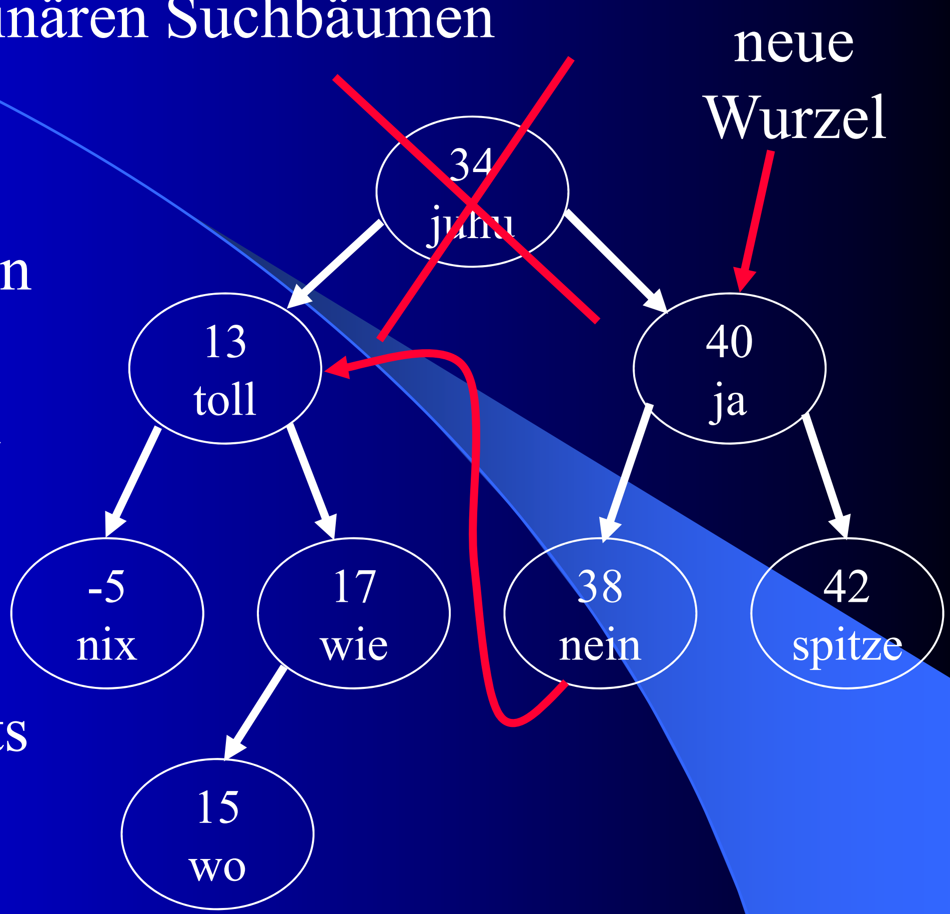
- Vorsicht vor entarteten binären Suchbäumen
- Situation: die Schlüssel

  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 Suchbäume funktionieren nicht gut, wenn die Eingaben nicht möglichst gleichverteilt ankommen



Löschen aus binären Suchbäumen



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

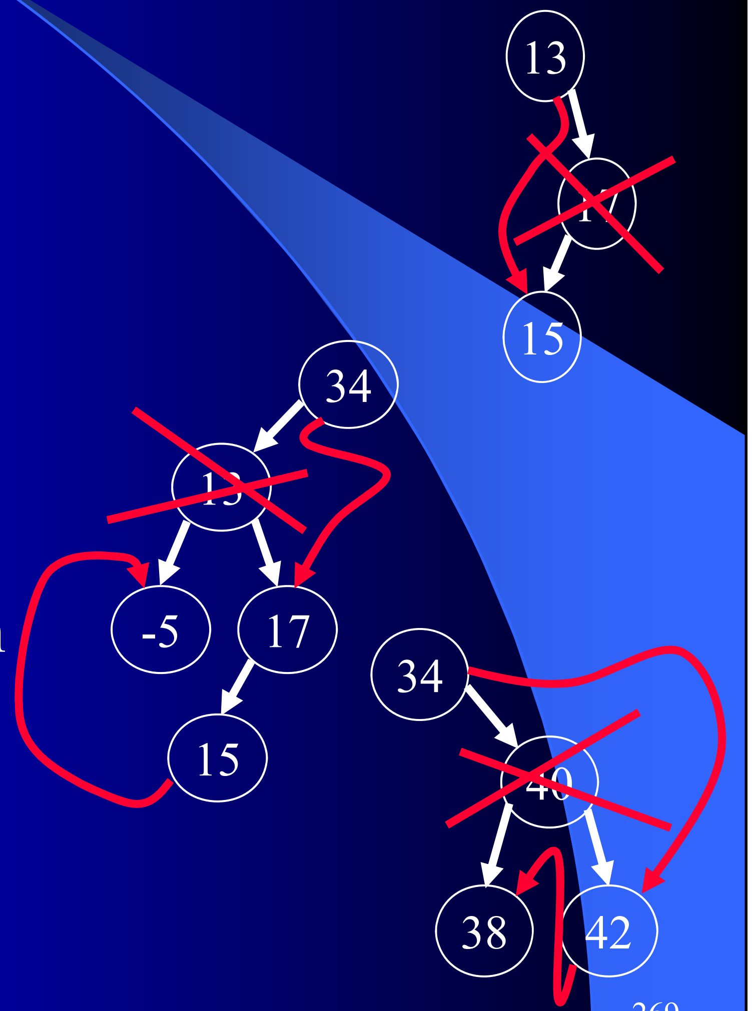
Löschen aus binären Suchbäumen (Fort.)

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 aus binären Suchbäumen: 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

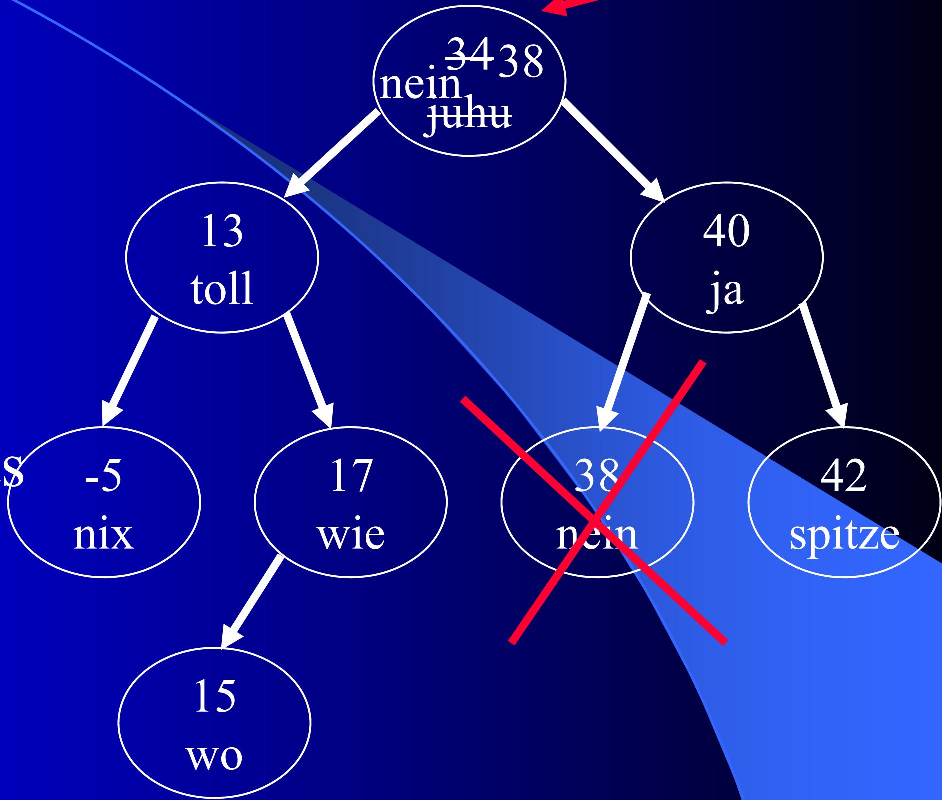
es gibt einen rechte Nachfolger; suche das kleinste Element in dem rechten Teilbaum

Abstieg

nicht gefunden

Löschen aus binären Suchbäumen

neue Wurzel
= alte Wurzel



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 aus binären Suchbäumen: 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 11

Hashing

- sehr gutes Verfahren für Suchen und Finden
- ist ein Kompromiss zwischen Speicherplatzverbrauch und Laufzeit
- relativ einfach zu implementieren
- 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

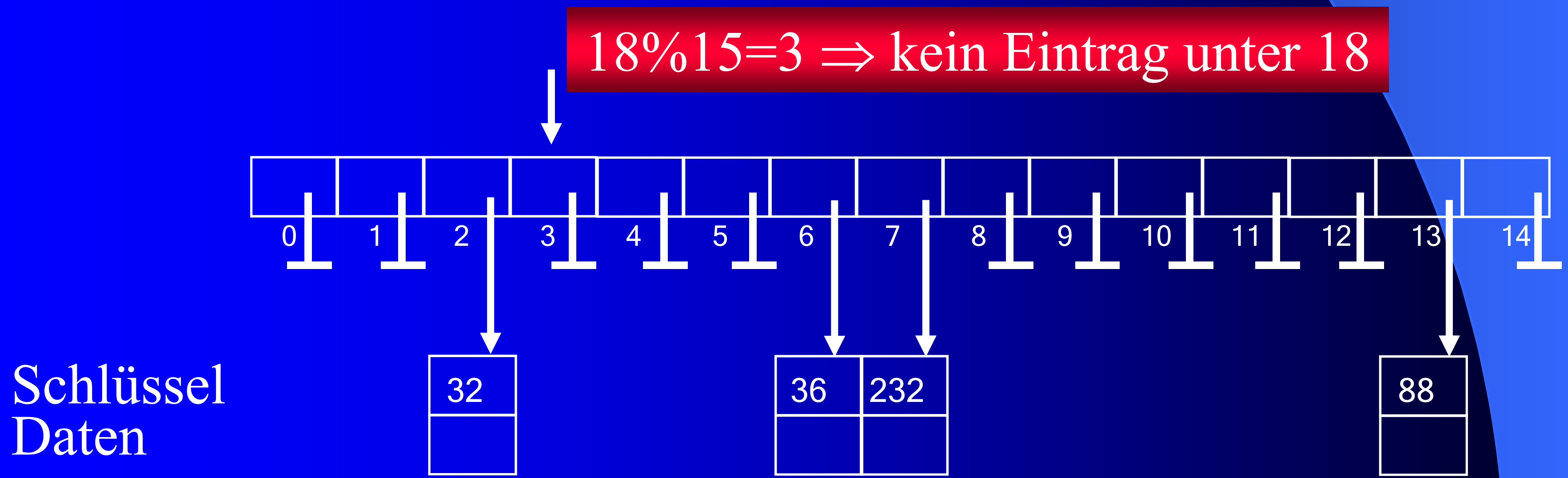
Schlüssel			32				36	232						88	
Daten															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

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;
    }

...
}
```

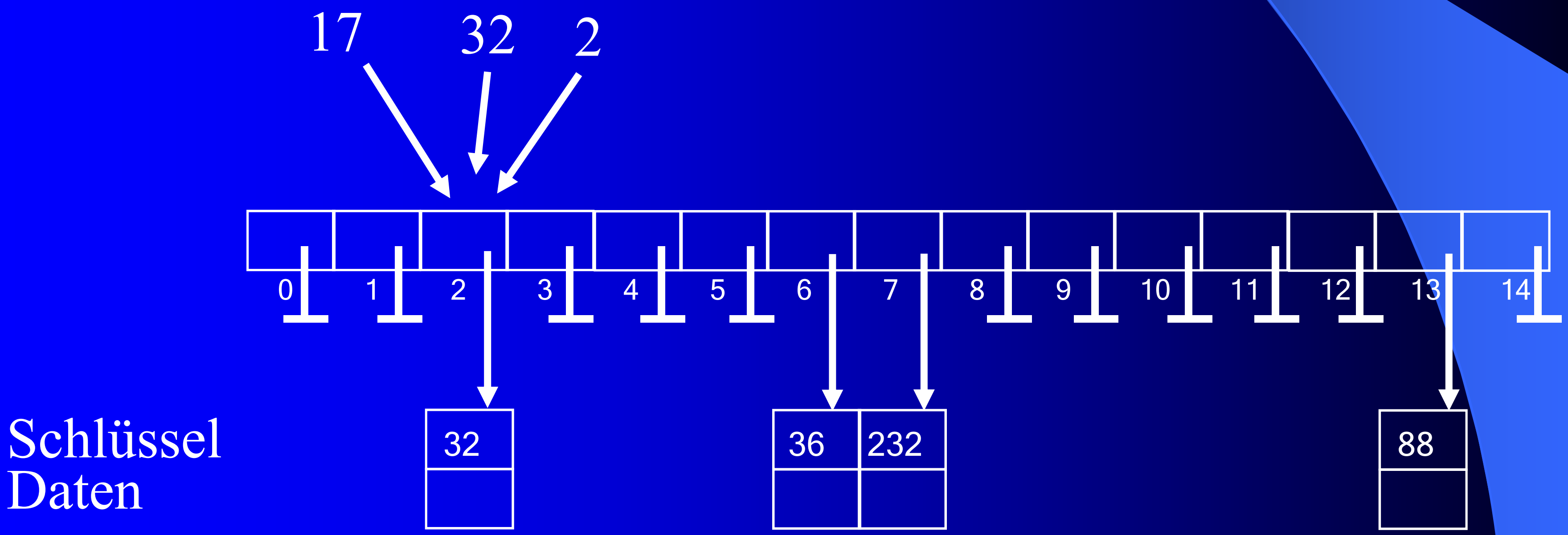
setzt voraus, dass key einen Modulo-Operator hat

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

wenn es einen Eintrag gibt, dann wird der Datensatz zurückgeliefert

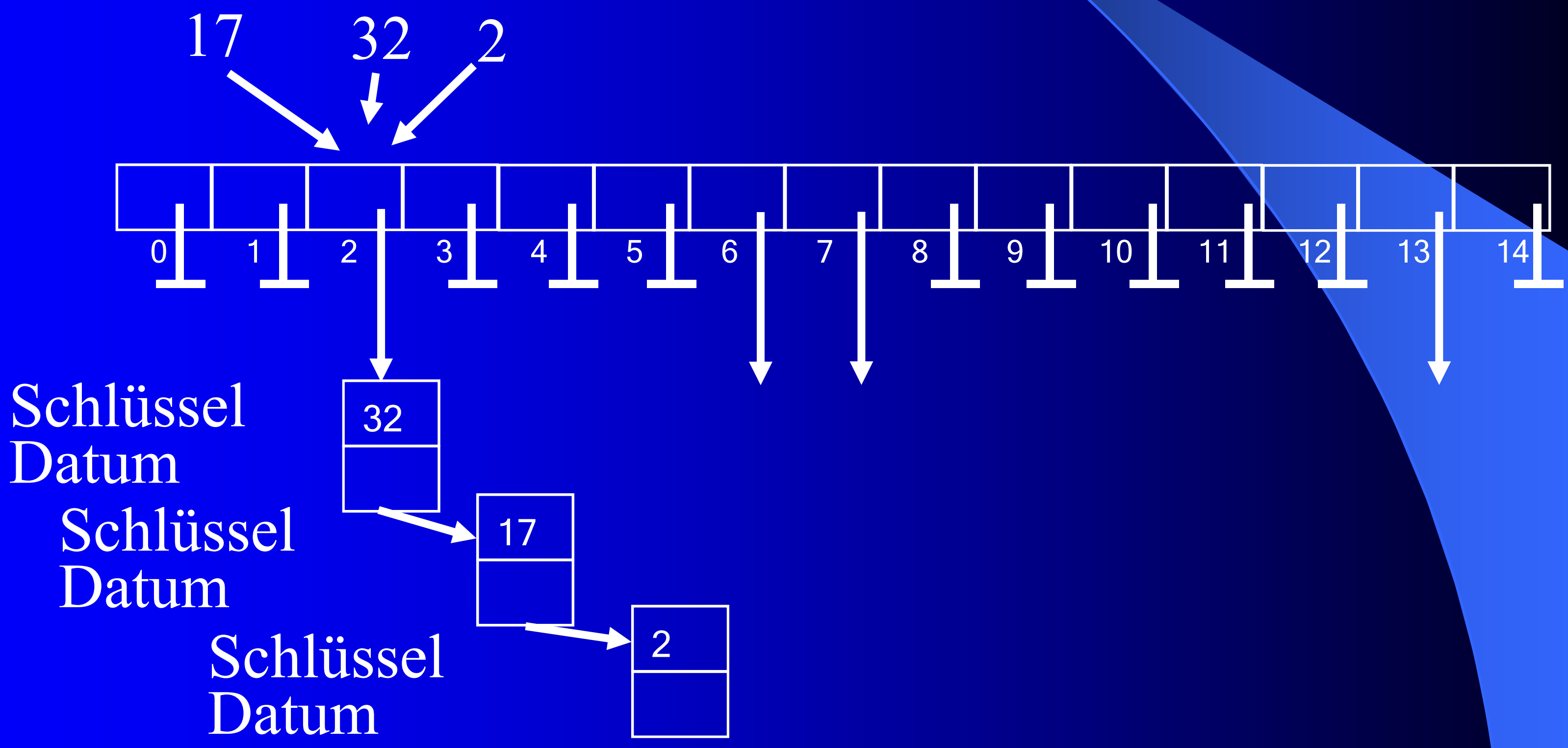
Hashing: Problem

- 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



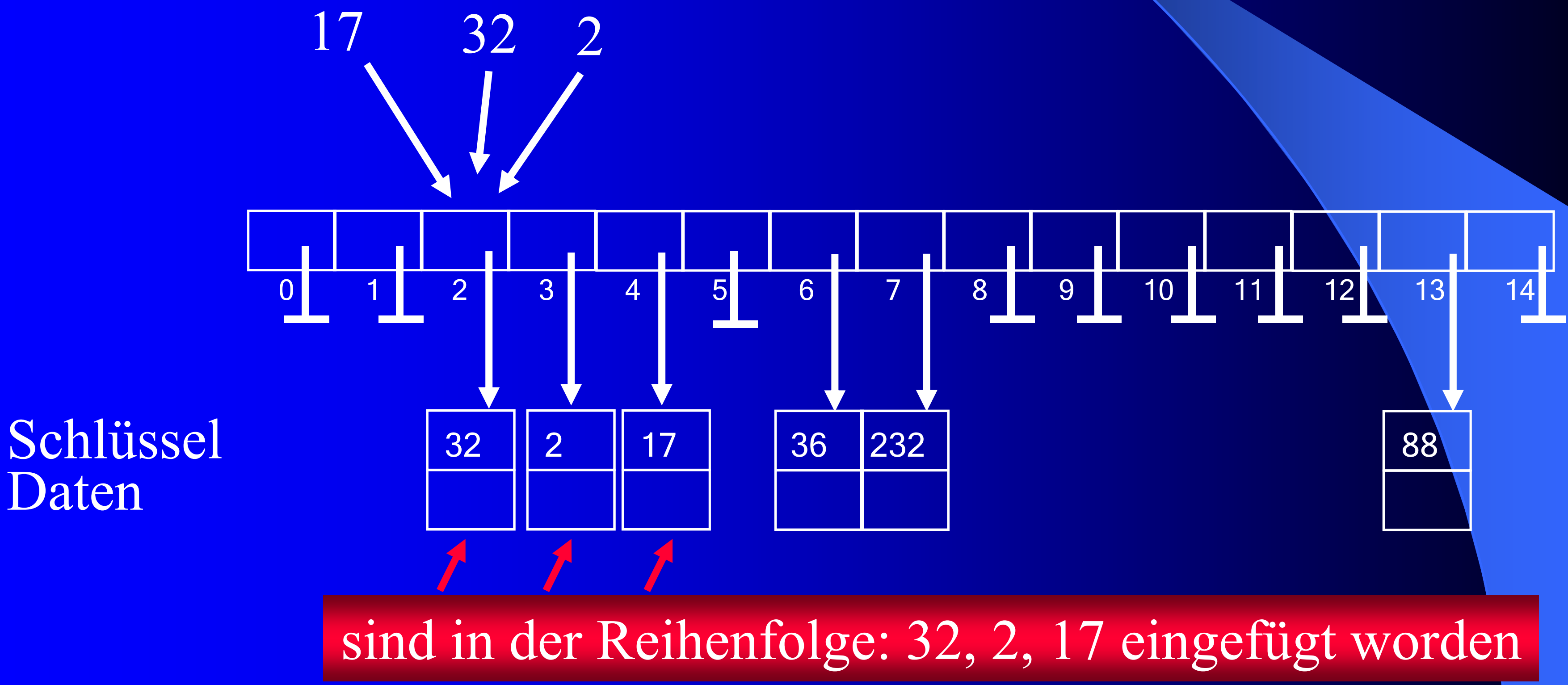
Hashing: Kollisionsbehebung

- 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

```
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);
        }
    }
}
```

das alte Array wird verdoppelt

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;
}
```

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

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

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.)

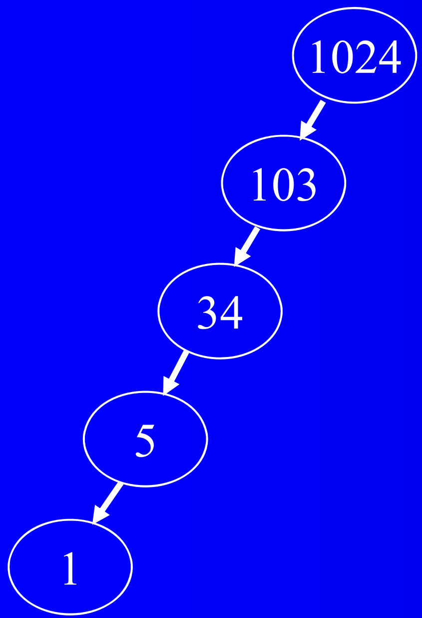
- für equals gelten folgende Regeln:
  1. reflexiv: x.equals(x) = true
  2. symmetrisch: x.equals(y) == y.equals(x)
  3. transitiv: x.equals(y) $\wedge$ y.equals(z) $\Rightarrow$ x.equals(z)
  4. konsistent: x.equals(y) ist immer true oder immer false
  5. x.equals(null) == false

Vorlesung 12

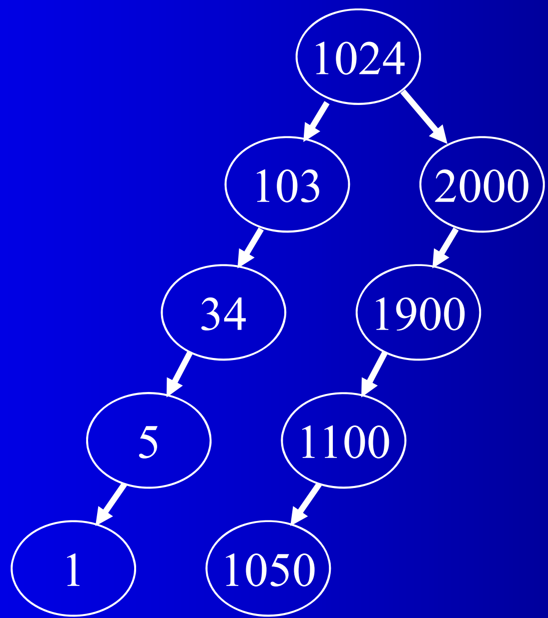
Nachteil von binären Bäumen

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

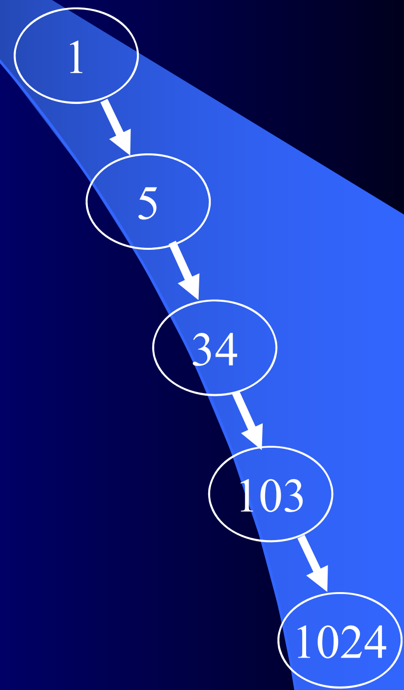
1024 103 34 5 1



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

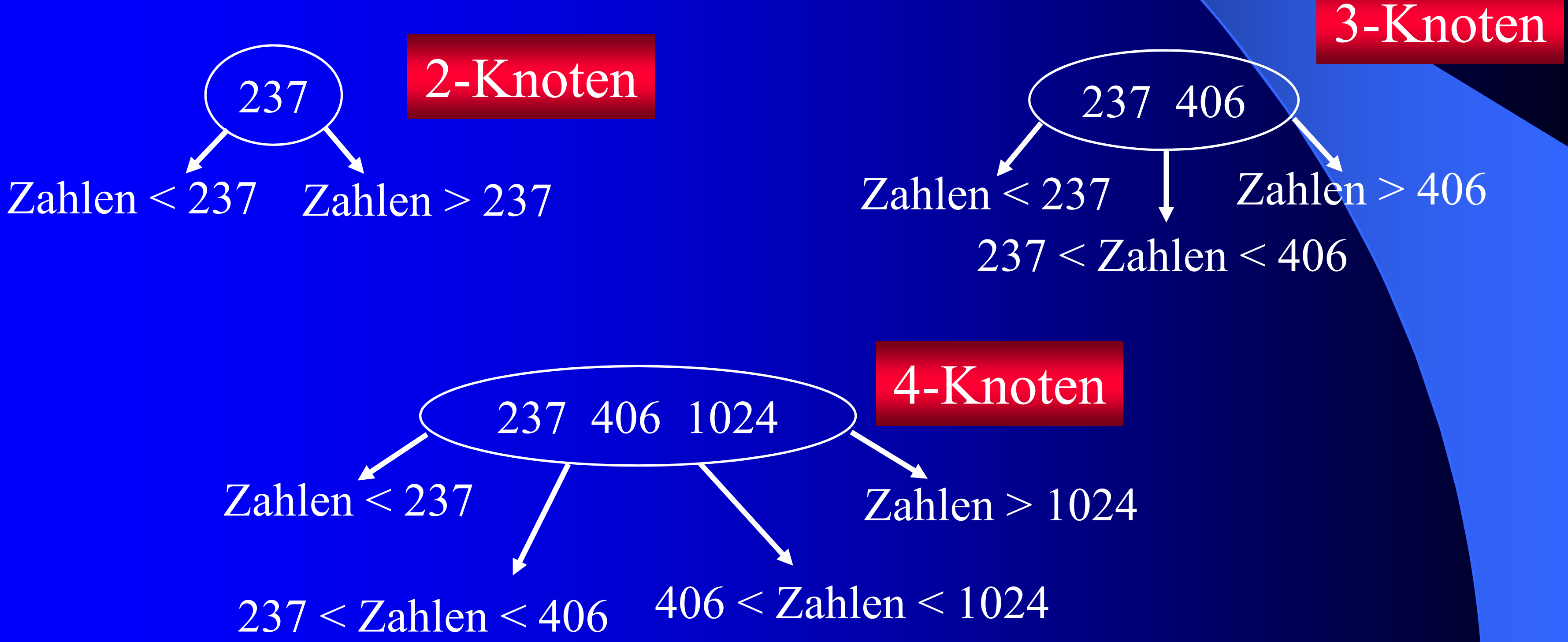
Idee:

- 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

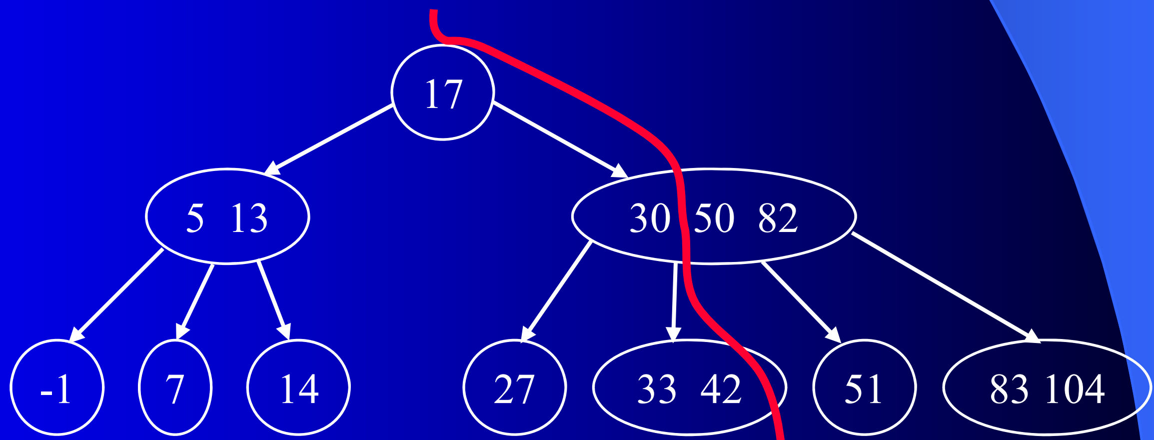


Top-Down 2-3-4-Bäume: Suchen

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

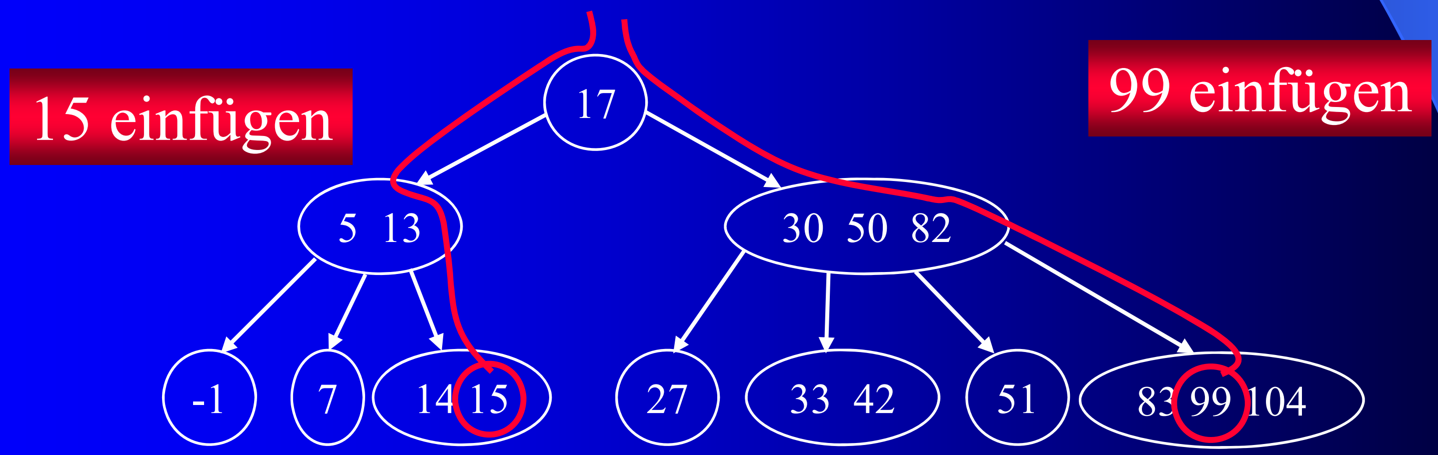


nicht gefunden

Top-Down 2-3-4-Bäume: Einfügen

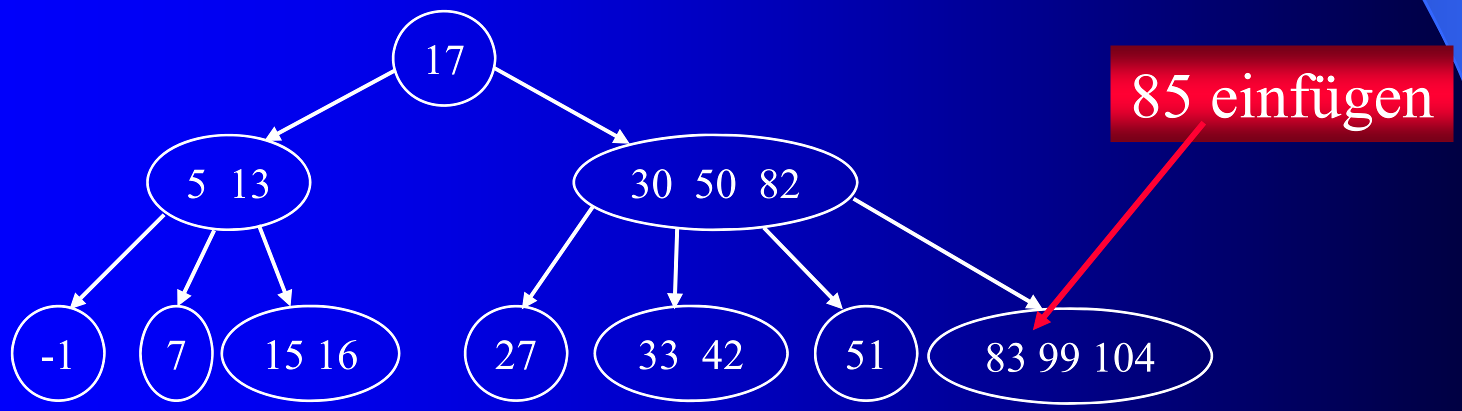
Idee:

- 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

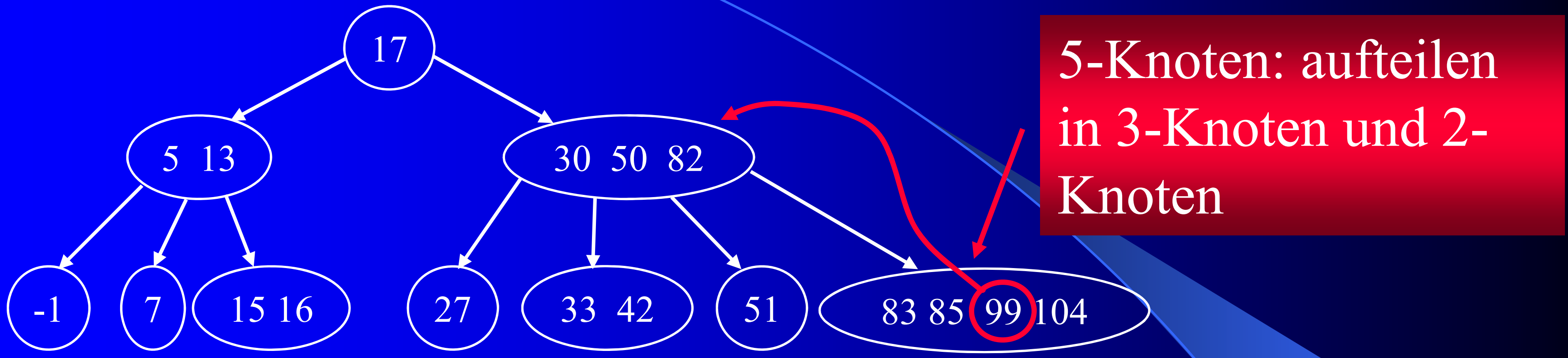


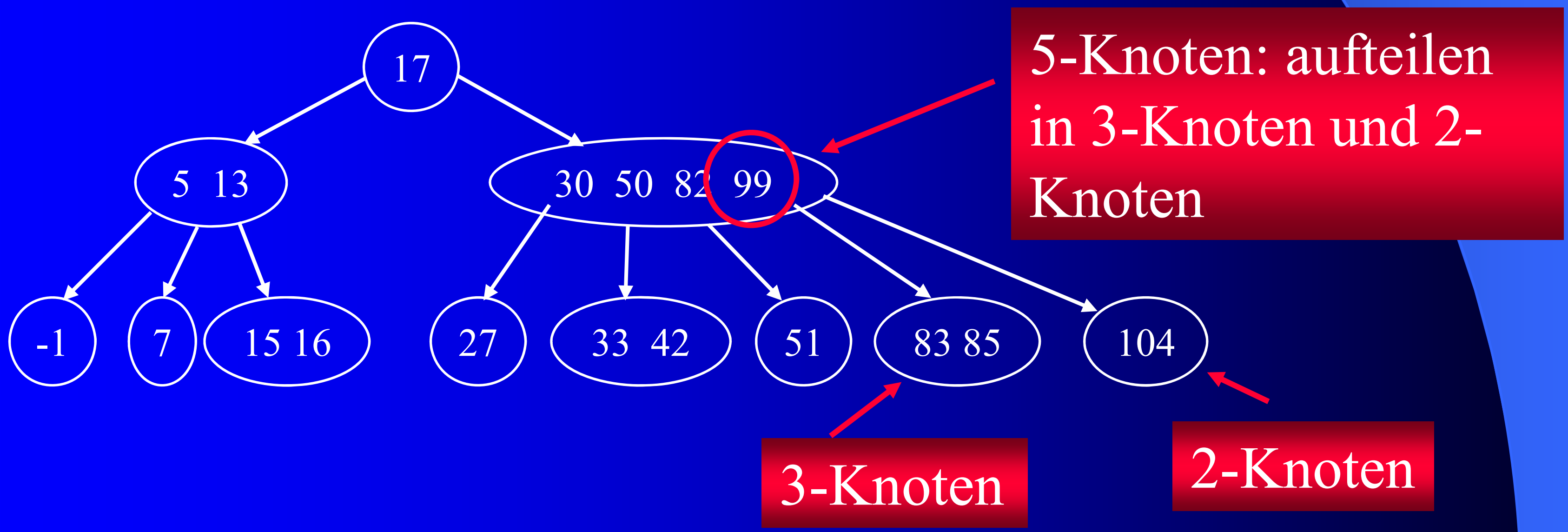
Top-Down 2-3-4-Bäume: Einfügen (Fort.)

- 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 Schlüssel mehr
- dadurch muss der Vater (und rekursiv dessen Vater usw.) u.U. ebenfalls neu aufgeteilt werden

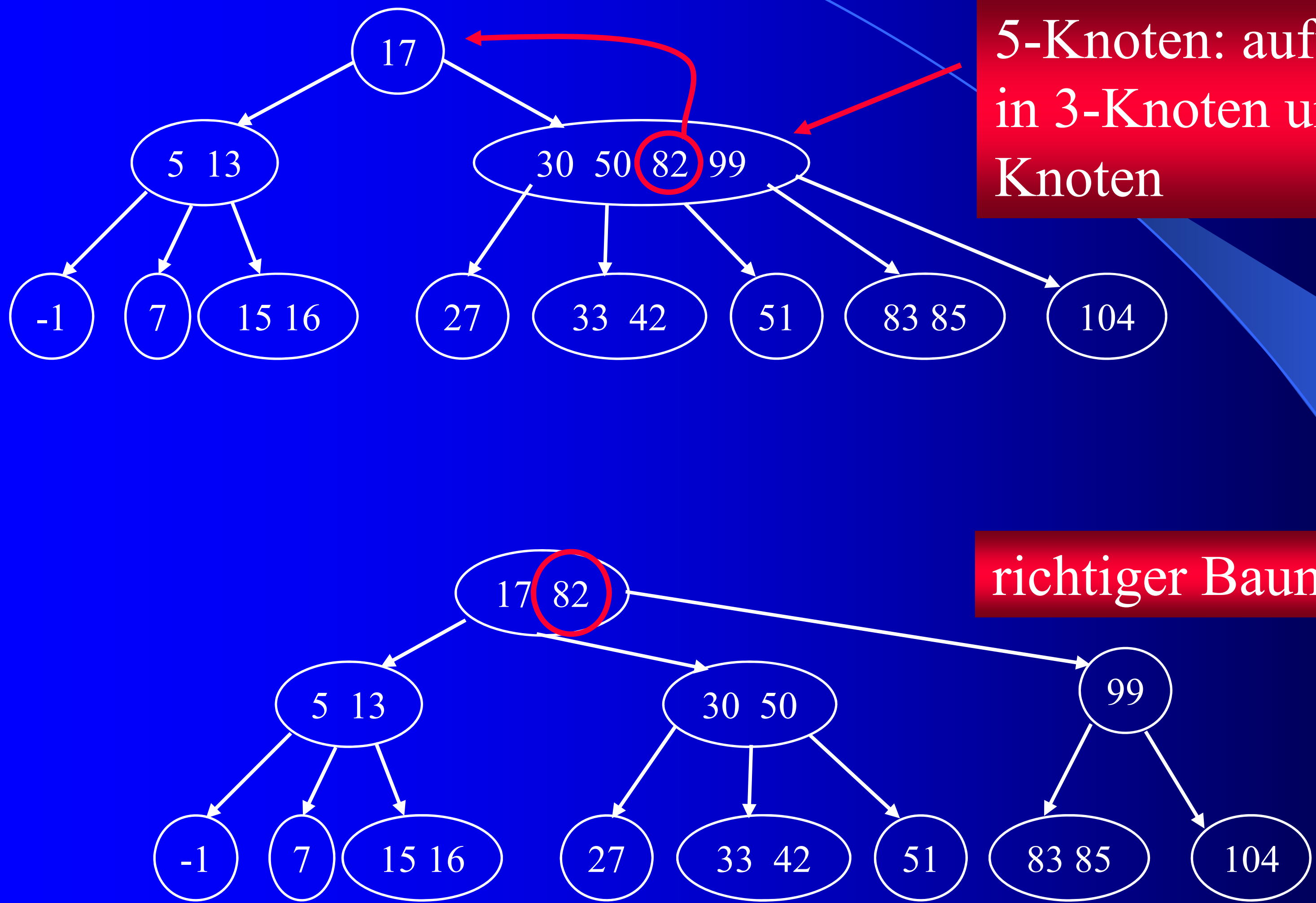


Top-Down 2-3-4-Bäume: Einfügen (Fort.)





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



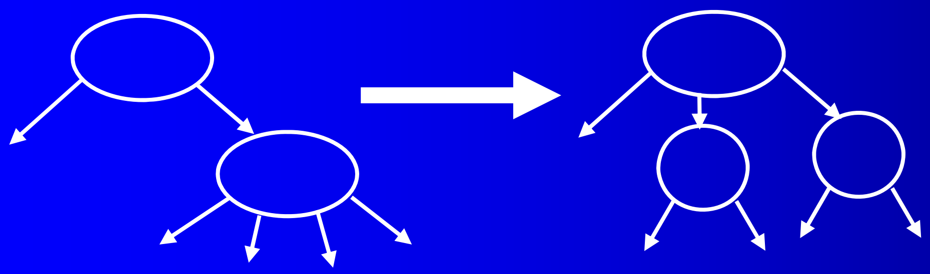
5-Knoten: aufteilen in 3-Knoten und 2-Knoten

richtiger Baum

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

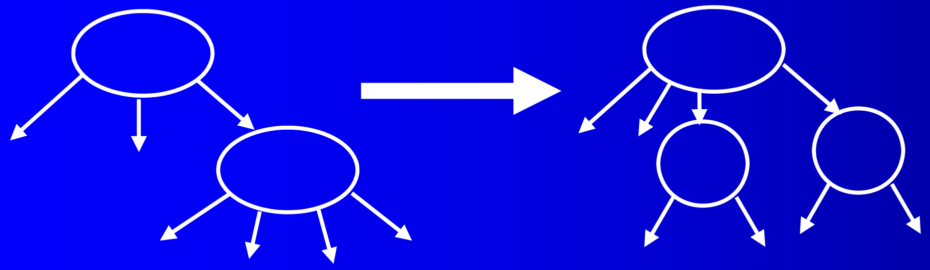
Optimierung:

- 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:



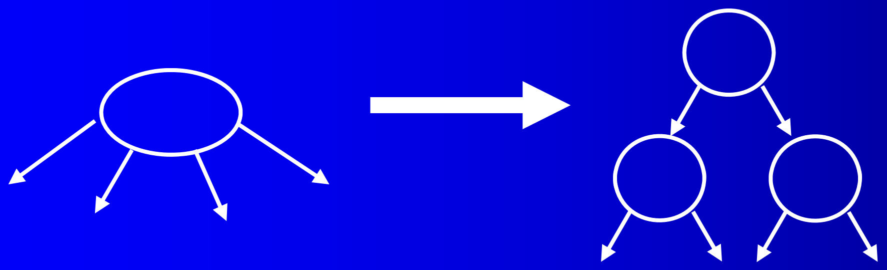
aus einem 2-Knoten mit 4-Knoten Nachfolger wird ein 3-Knoten mit 2 2-Knoten Nachfolgern

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



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

Spezialfall: 4-Knoten Wurzel



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

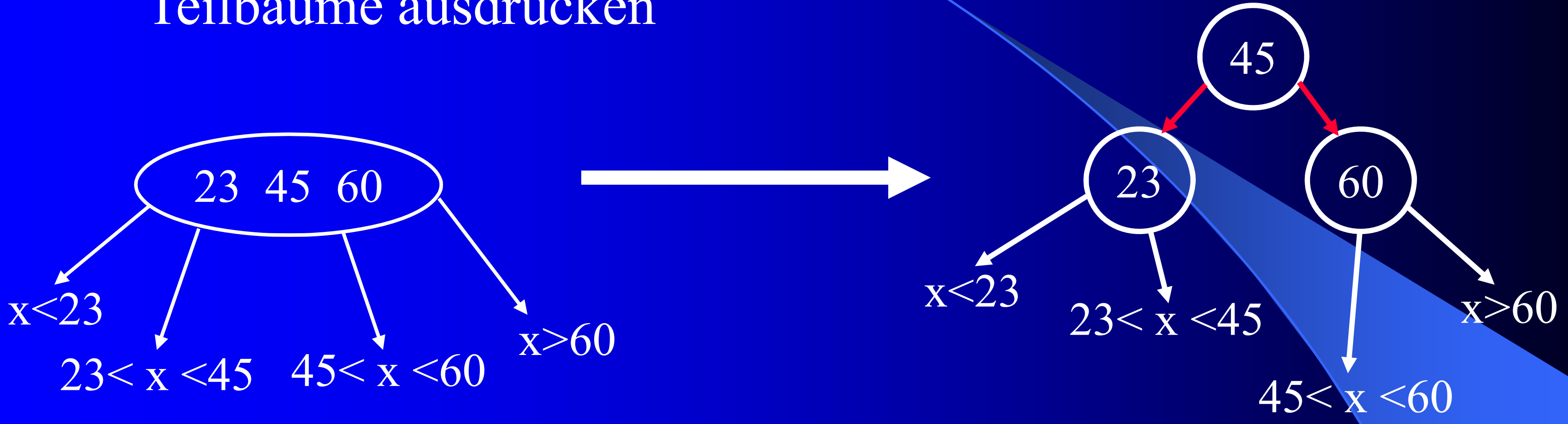
Top-Down 2-3-4-Bäume: Eigenschaften

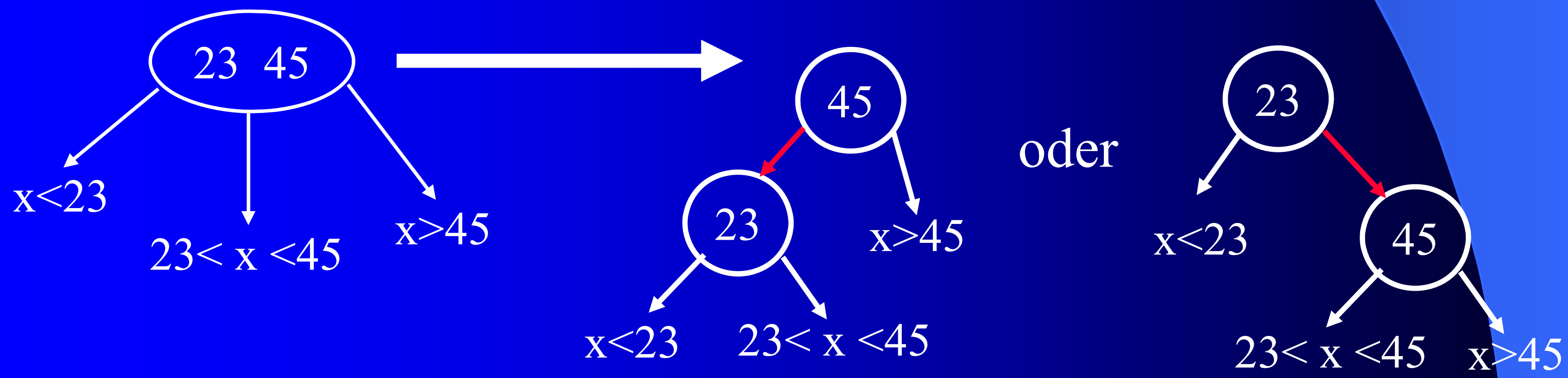
- da der Baum nur an der Wurzel wachsen kann, ist er immer ausgeglichen
- dadurch liegt das Suchen in O(log N)
- das Einfügen liegt garantiert in O(log N)

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

Rot-Schwarz Bäume

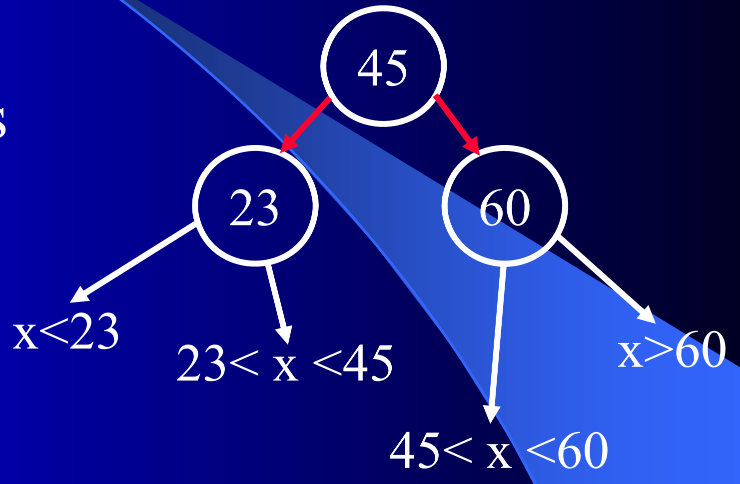
- 3-Knoten und 4-Knoten lassen sich auch durch binäre Teilbäume ausdrücken



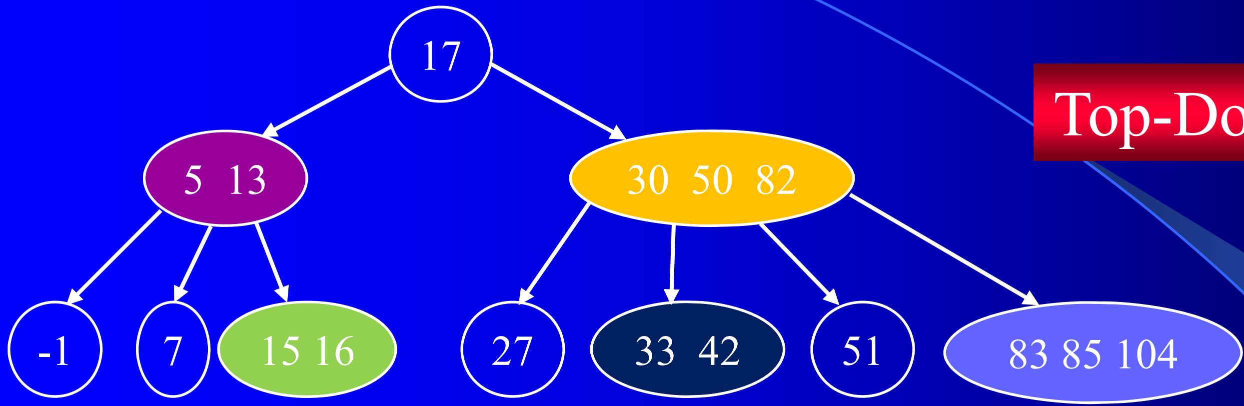


Rot-Schwarz Bäume (Fort.)

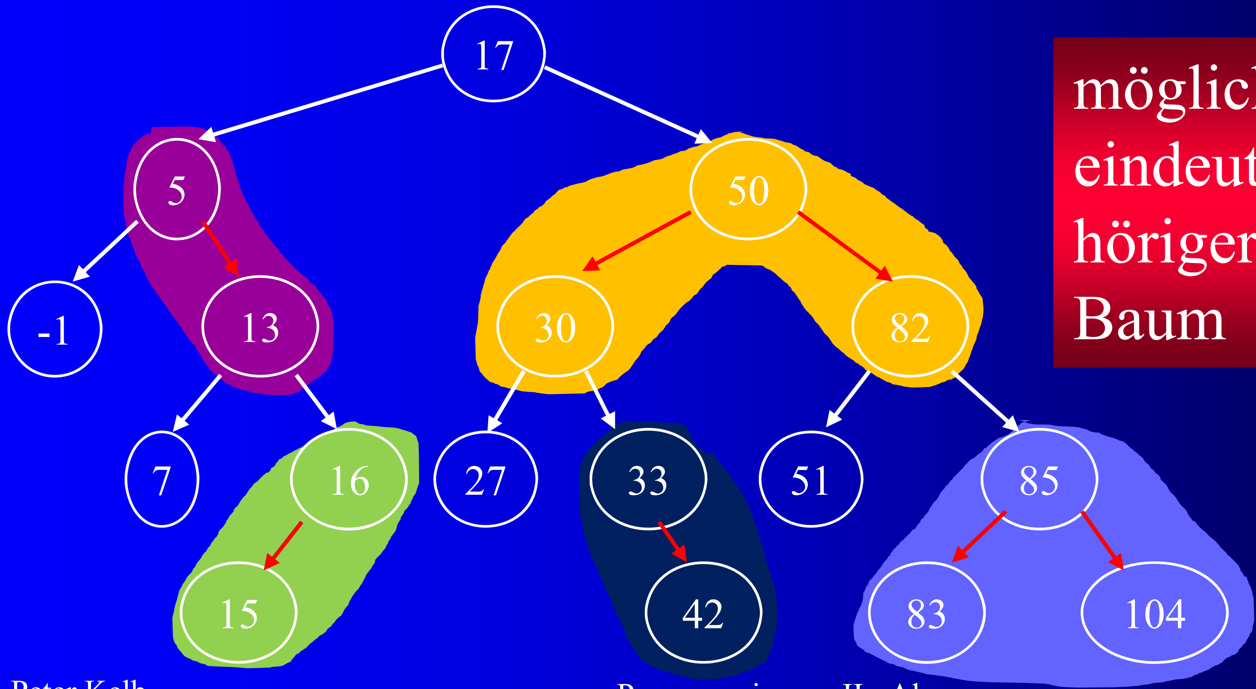
- jeder 3-Knoten bzw. 4-Knoten lässt sich durch einen binären Teilbaum darstellen
- die Tiefe eines solchen Baums ist maximal 2-mal so groß wie die eines Top-down 2-3-4 Baums
- die roten Kanten dienen nur der Darstellung von 3- bzw. 4-Knoten
- die anderen Kanten dienen der Verkettung
- daher heißen diese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



Rot-Schwarz Bäume (Beispiel)



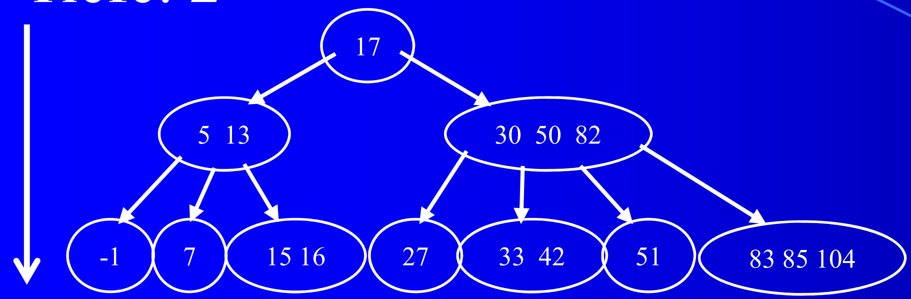
Top-Down-Baum



möglicher (selten eindeutiger) zugehöriger Rot-Schwarz-Baum

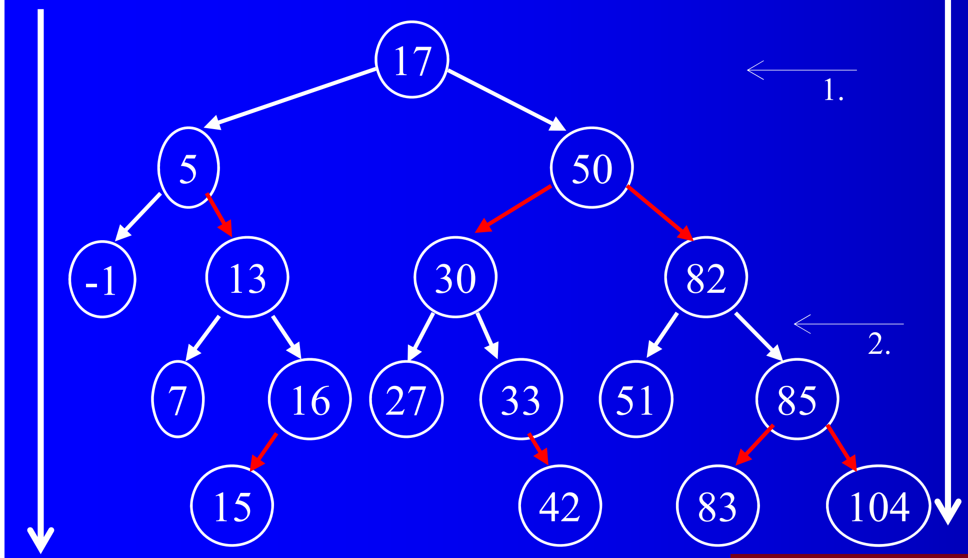
Rot-Schwarz Bäume: Tiefen

Tiefe: 2



Tiefe: 4

Schwarztiefe: 2



- die Tiefe eines Binärbaums ist die maximale Anzahl der Kanten von der Wurzel zu einem Blatt
- bei einem Top Down 2-3-4 Baum sind die Pfade alle gleich lang
- bei Rot-Schwarzen Bäumen wird neben der normalen Tiefe (maximale Anzahl der Kanten von der Wurzel zu einem Blatt) die sogenannte Schwarztiefe (maximale Anzahl der schwarzen Kanten von der Wurzel zu einem Blatt) betrachtet

Schwarztiefe = Tiefe des TD 234 Baums

Rot-Schwarz Bäume: Implementierung

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

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

Flag, das die Kantenfarbe anzeigt

Rot-Schwarz Bäume: Implementierung (Fort.)

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

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

wenn der Schlüssel gefunden ist, gibt den Datensatz zurück

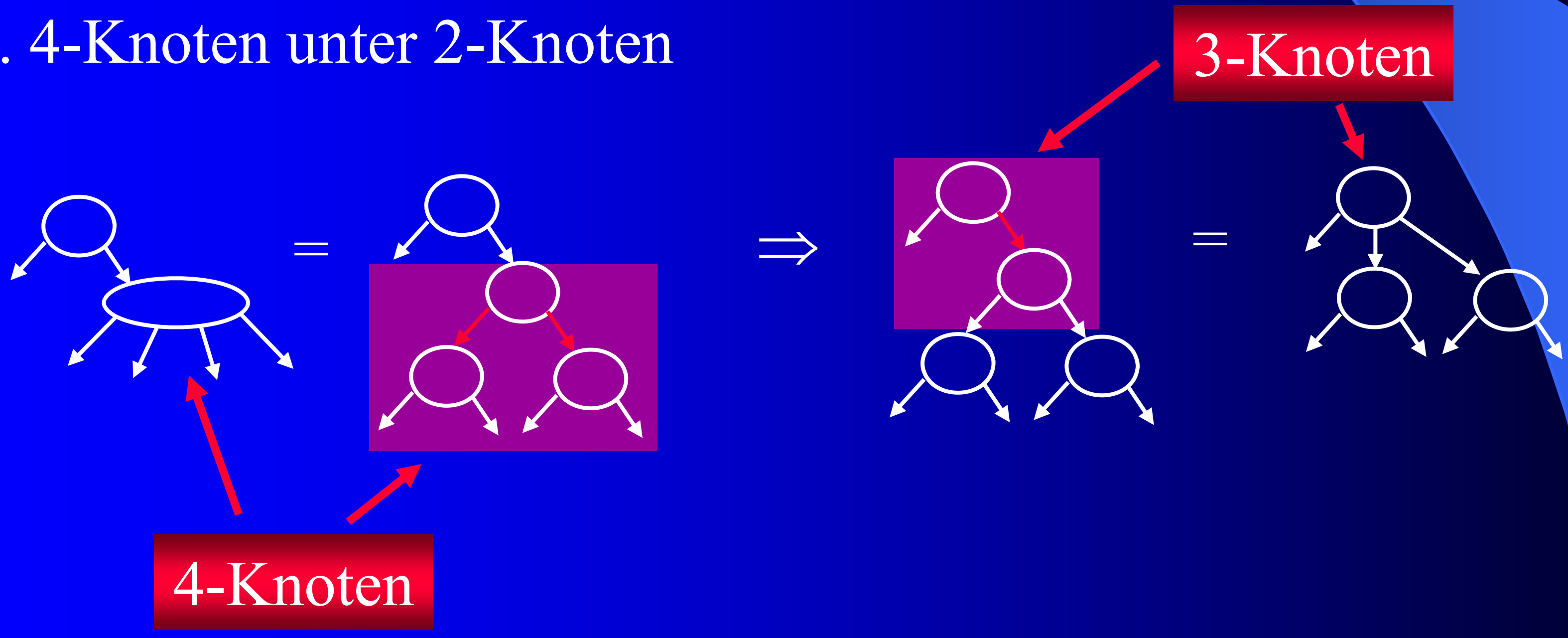
steige links bzw. rechts ab

Schlüssel ist nicht gefunden worden

Rot-Schwarz Bäume: Implementierung (Fort.)

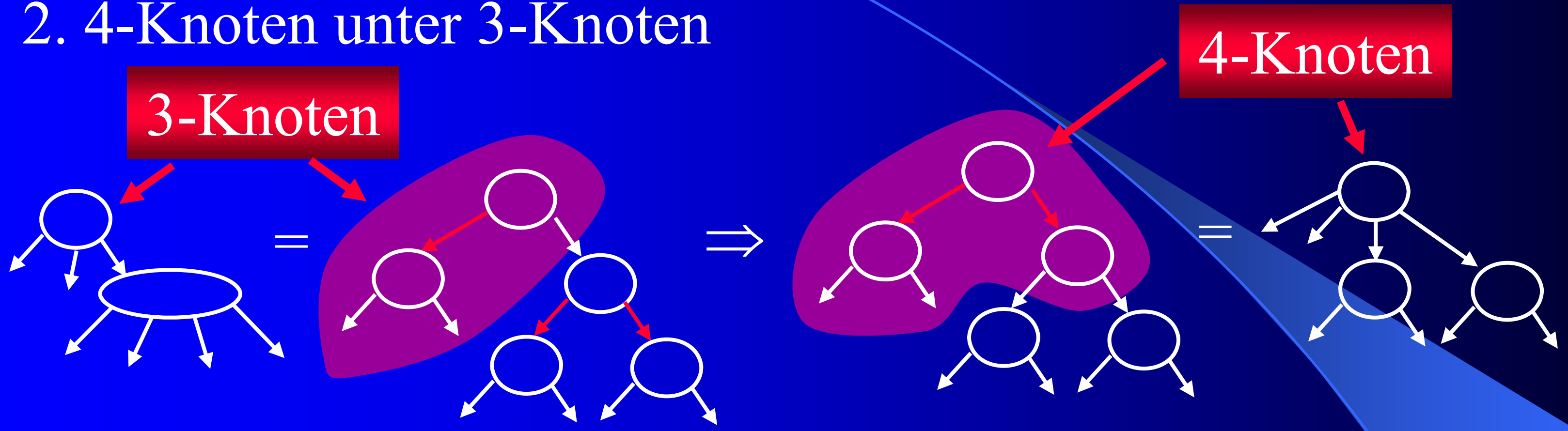
- beim Einfügen werden alle 4-Knoten aufgeteilt
- ein 4-Knoten erkennt man daran, dass beide Nachfolger-knoten das gesetzte Flag haben
- nicht sehr teuer, da es kaum 4-Knoten gibt
- es gibt 7 Fälle zu untersuchen

1. 4-Knoten unter 2-Knoten

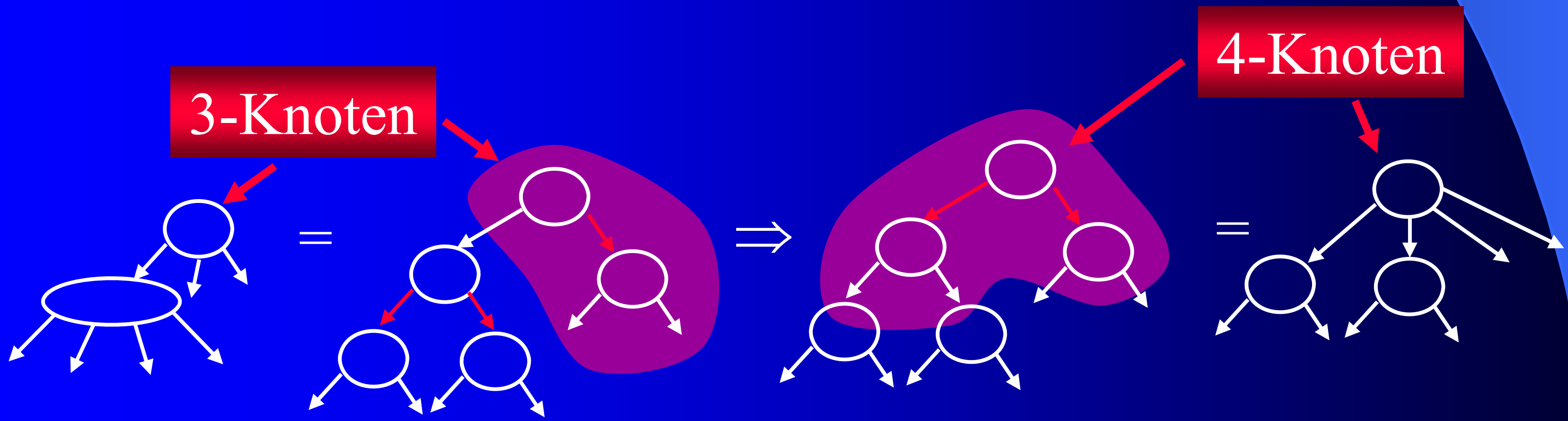


Rot-Schwarz Bäume: Implementierung (Fort.)

2. 4-Knoten unter 3-Knoten

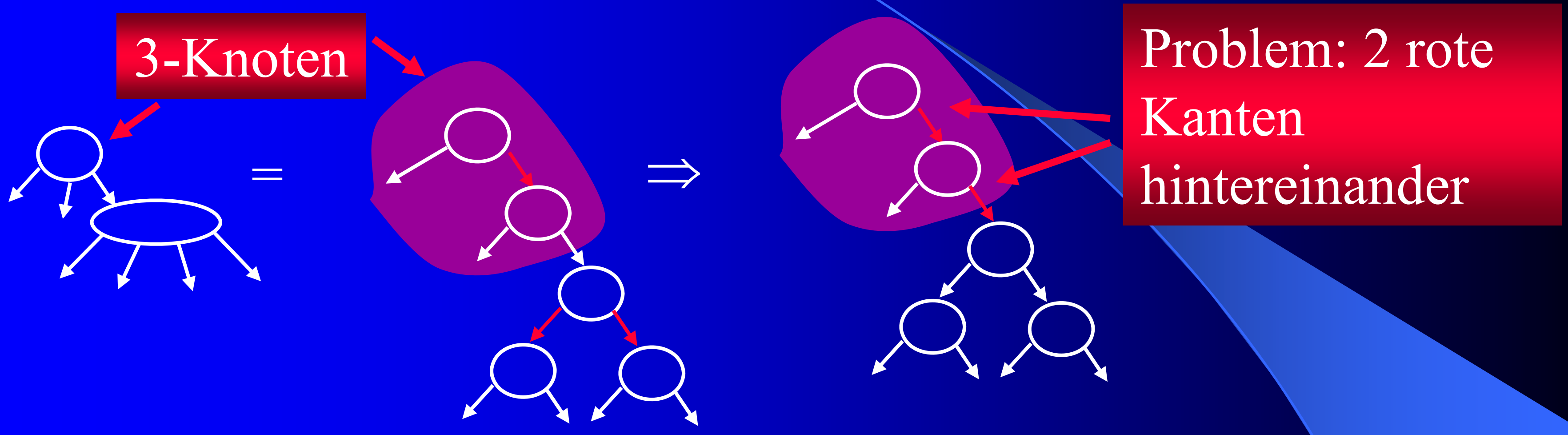


3. 4-Knoten unter 3-Knoten

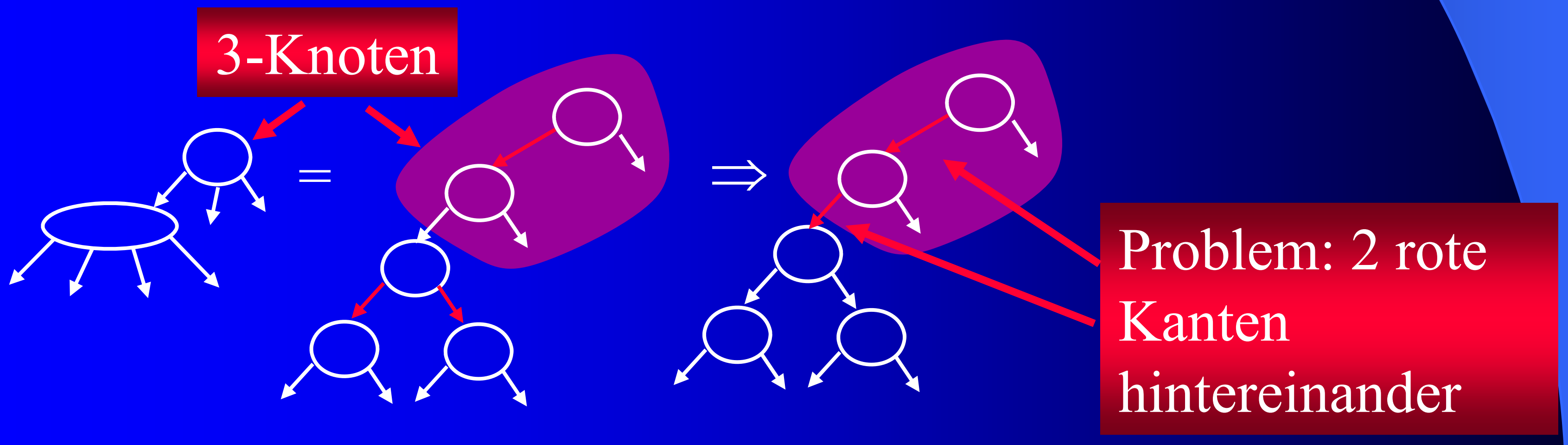


Rot-Schwarz Bäume: Implementierung (Fort.)

4. 4-Knoten unter 3-Knoten

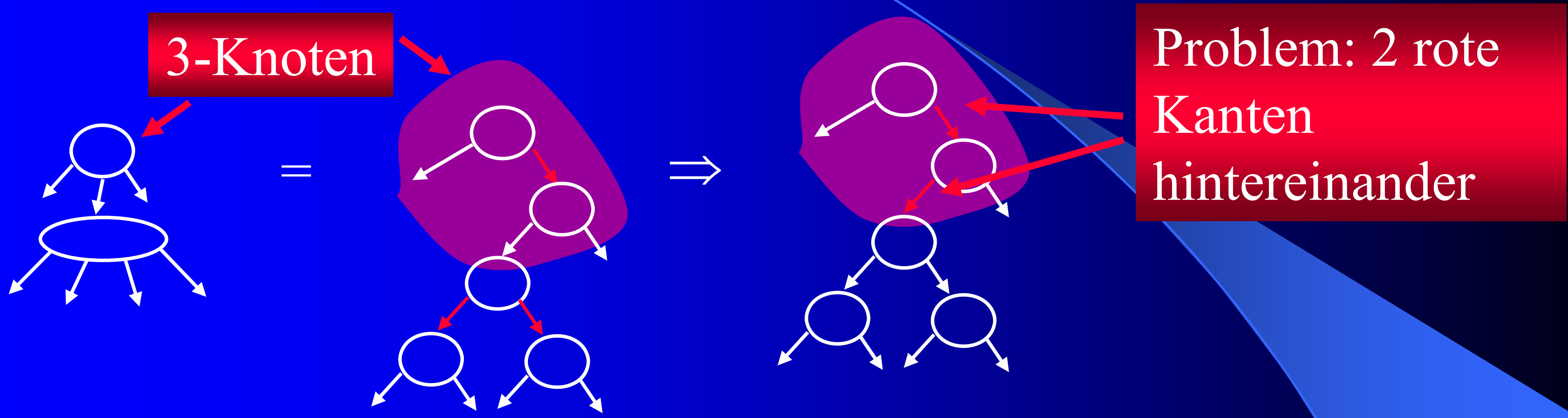


5. 4-Knoten unter 3-Knoten

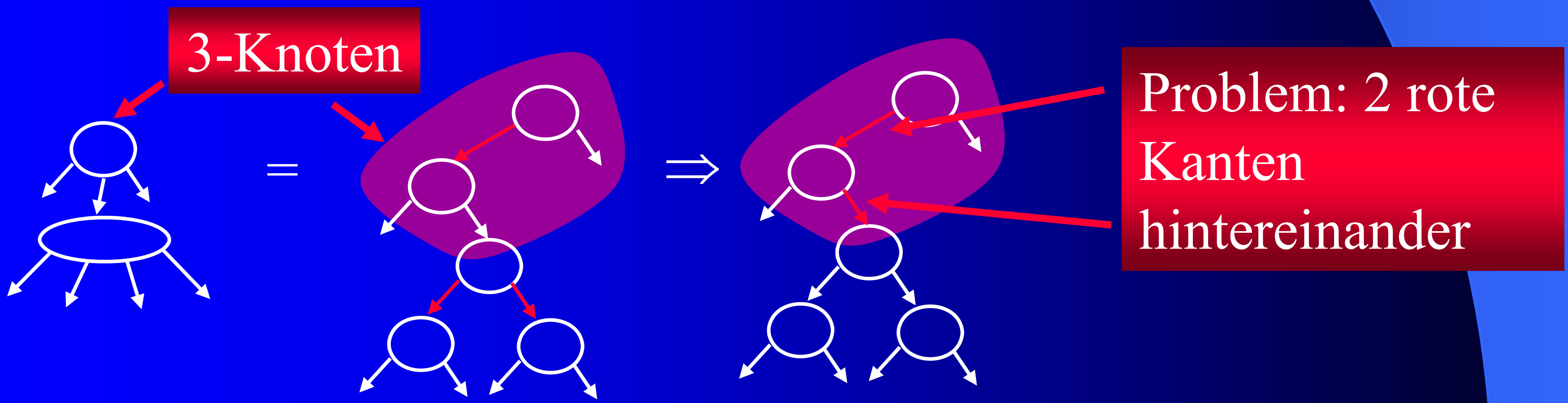


Rot-Schwarz Bäume: Implementierung (Fort.)

6. 4-Knoten unter 3-Knoten



7. 4-Knoten unter 3-Knoten

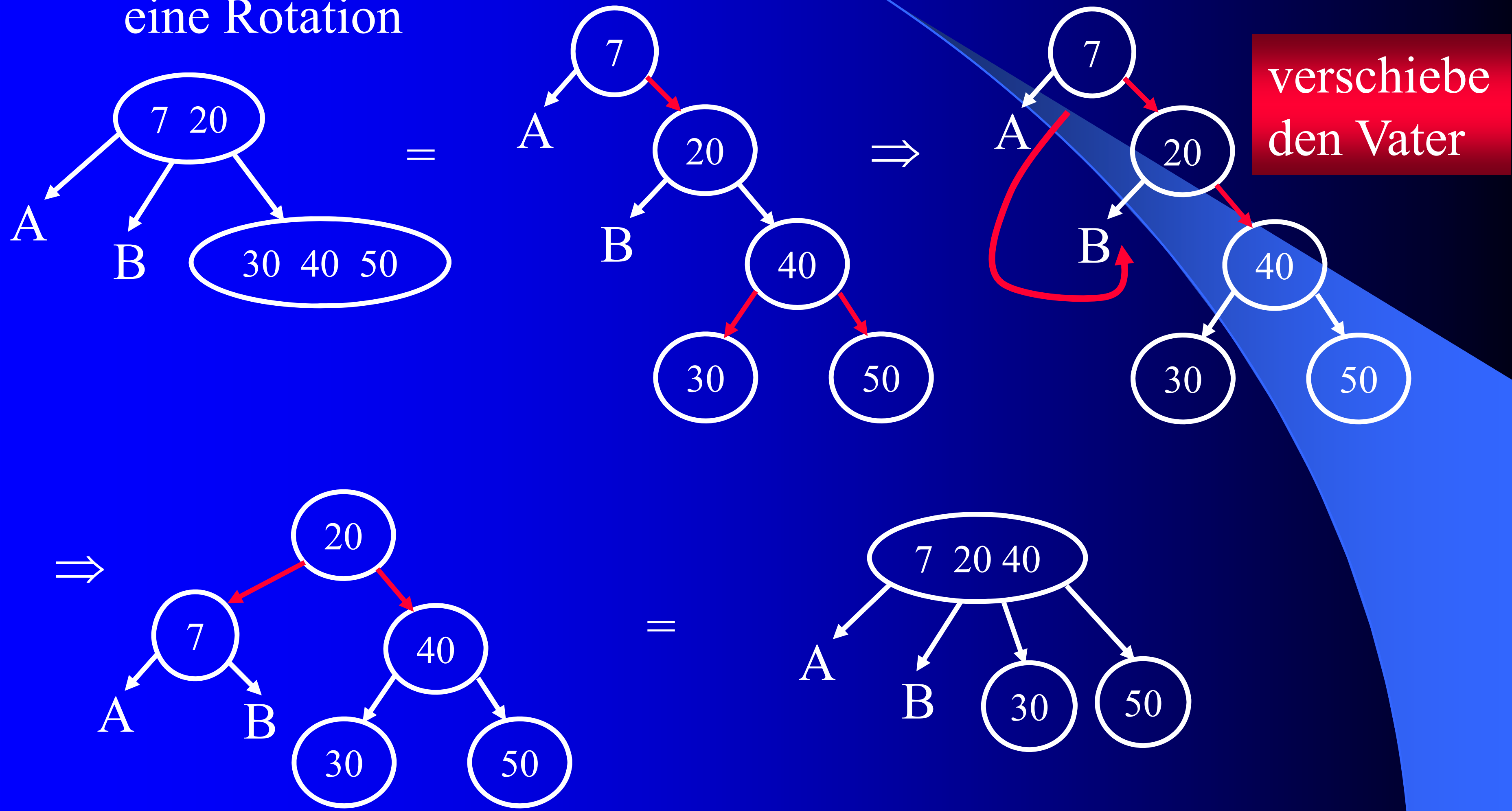


Rot-Schwarz Bäume: Implementierung (Fort.)

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

Rot-Schwarz Bäume: Implementierung (Fort.)

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



Rot-Schwarz Bäume: Implementierung (Fort.)

- Lösung für Fall 6 und 7: zwei Rotationen

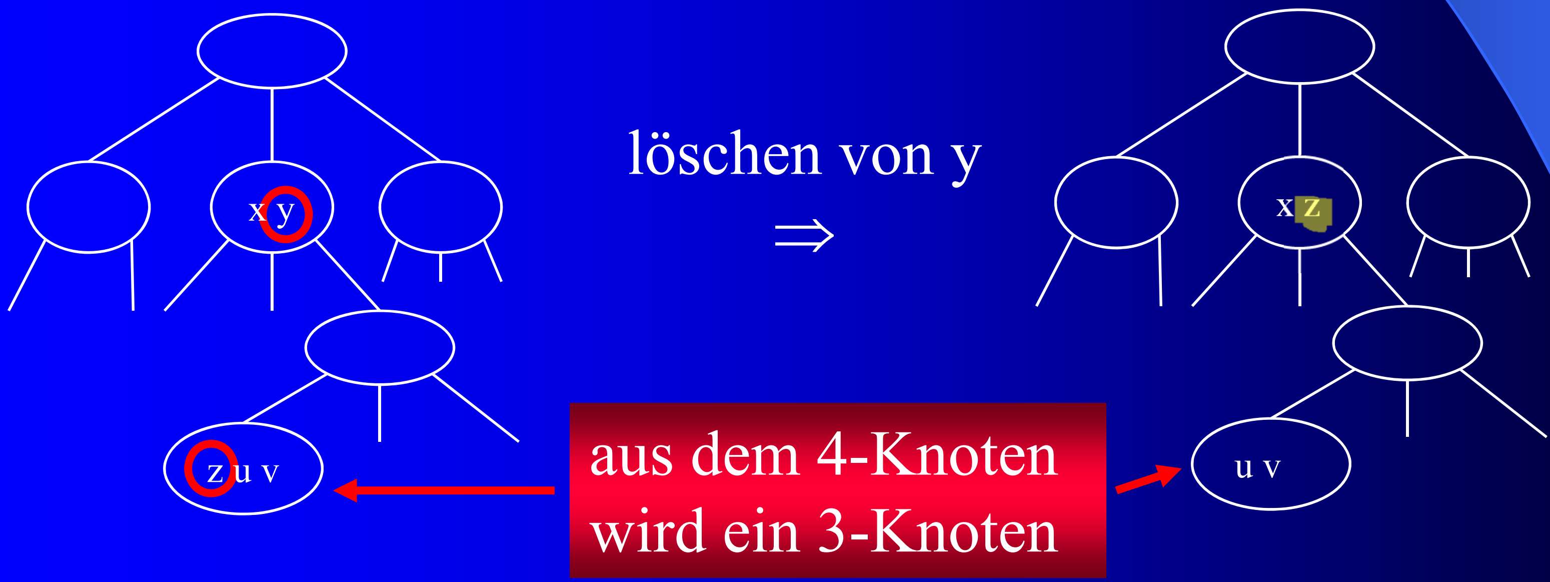
Vorlesung 13

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



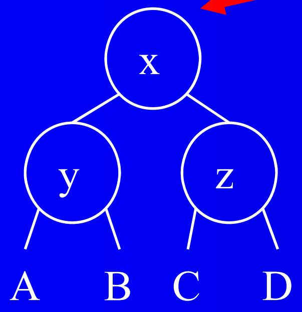
Löschen aus Top-Down 2-3-4 Bäumen (Forts.)

- 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

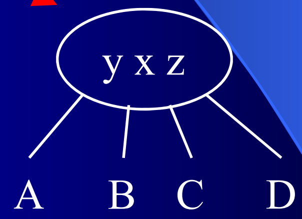
Fall 1

- 2-Wurzel mit zwei 2-Söhnen

Wurzel



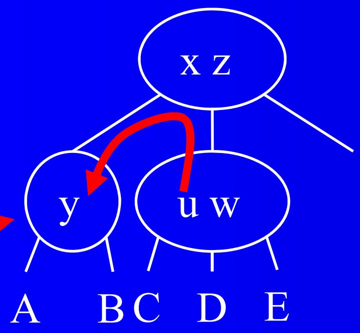
Wurzel und Söhne werden zu neuer $\Rightarrow$ Wurzel zusammengefasst



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

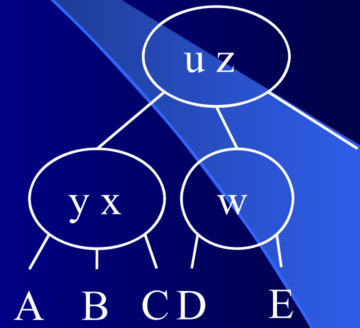
Fall 2

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

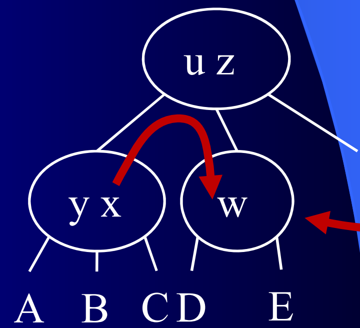


Linksrotation

$\Rightarrow$



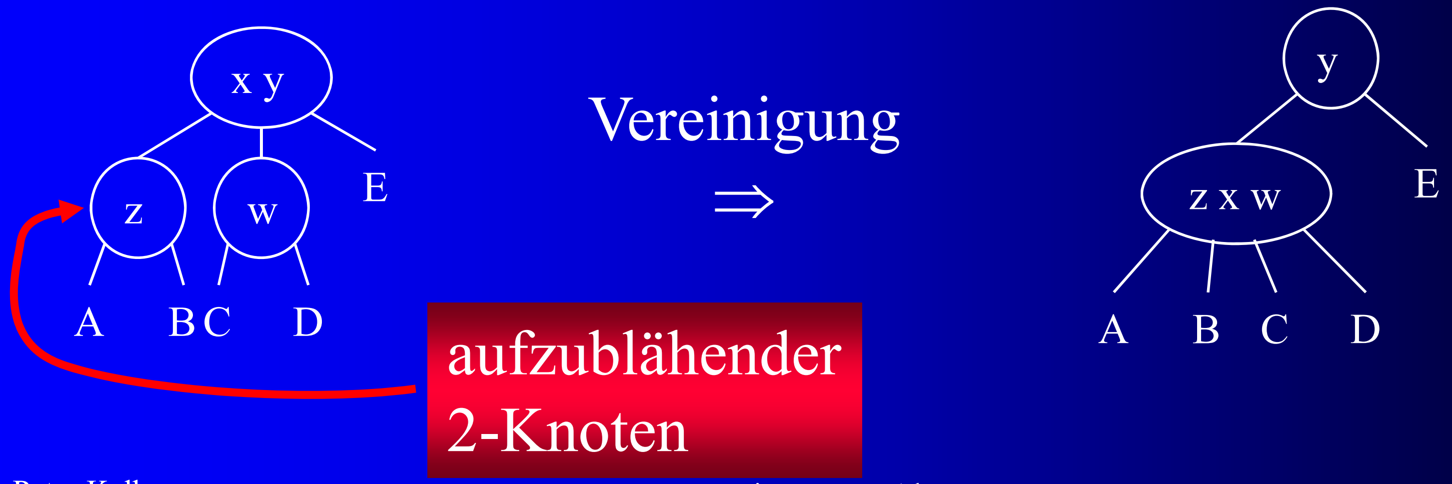
aufzublähender 2-Knoten



- gibt es natürlich auch als Rechtsrotation

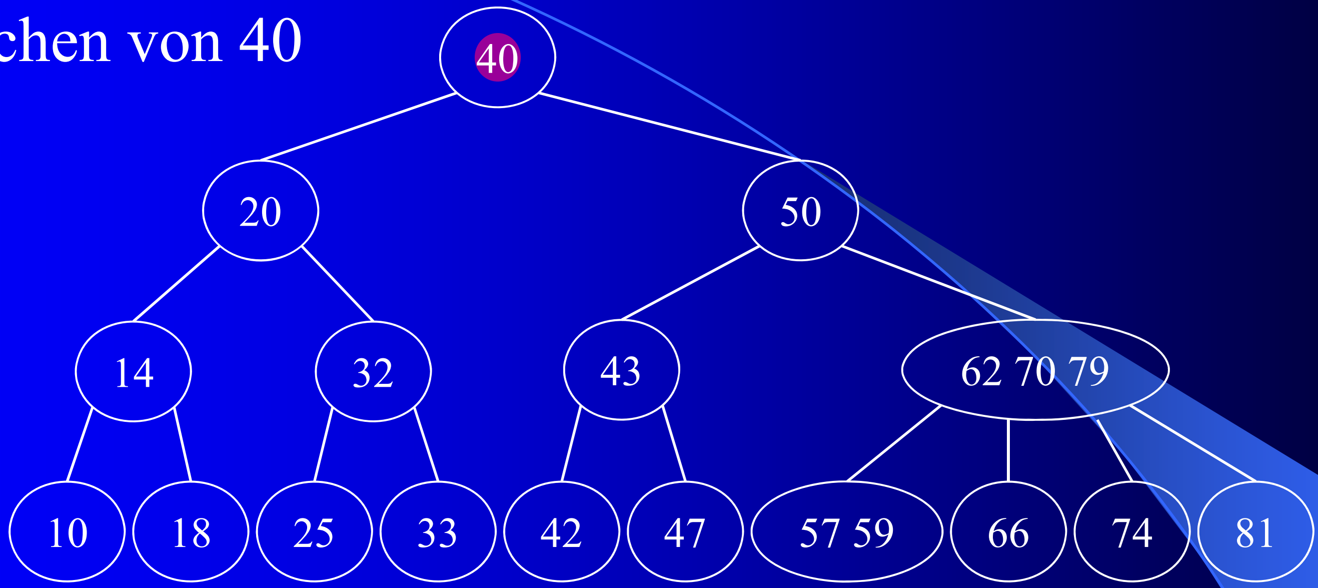
Fall 3

- 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)

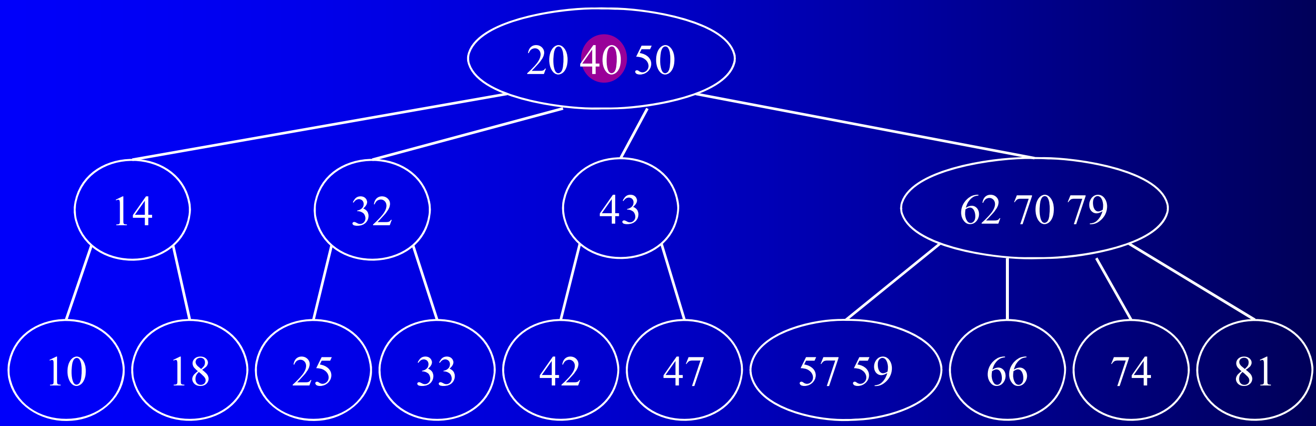


Beispiel

- Löschen von 40

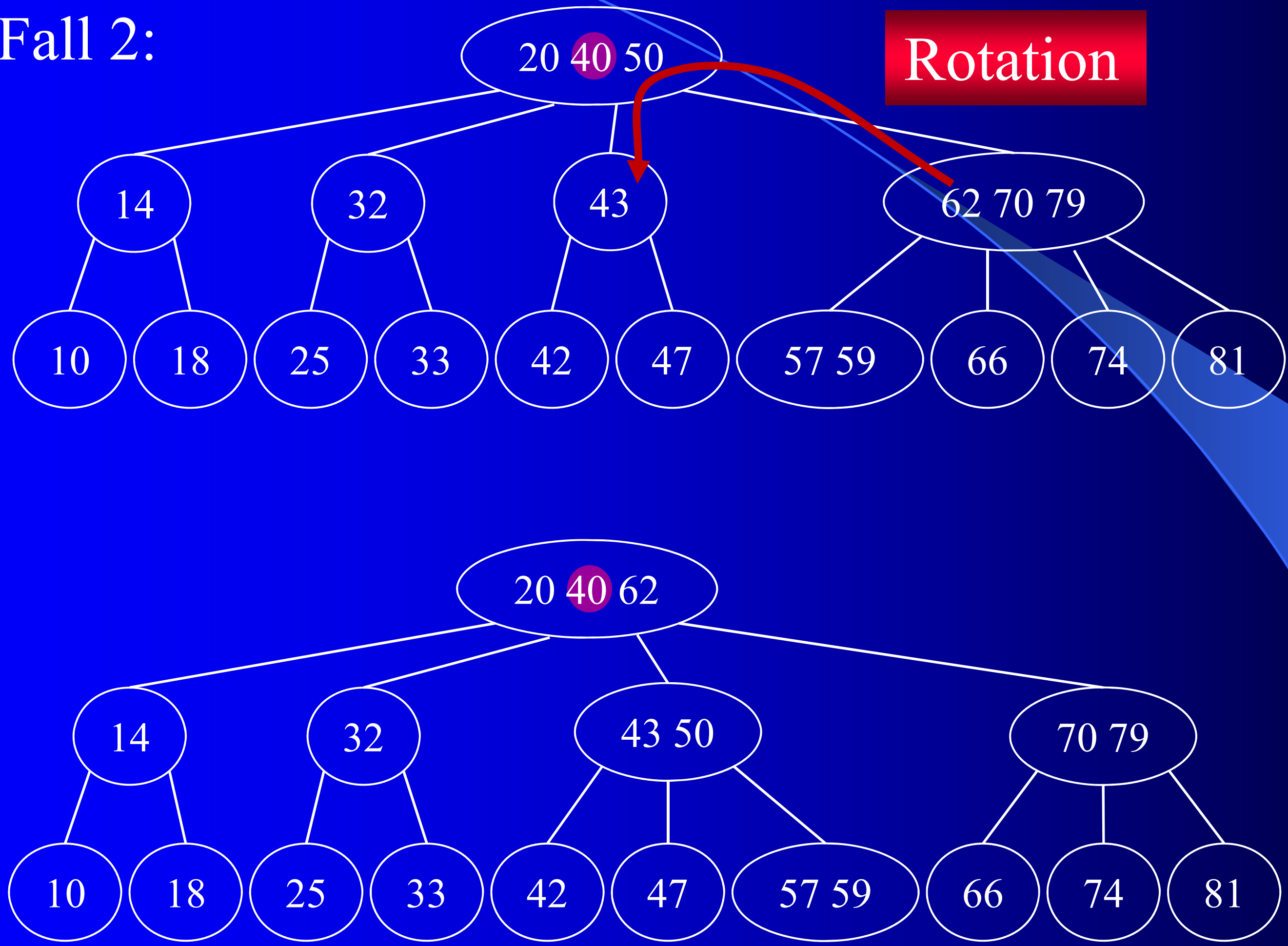


- Fall 1: Wurzel und beide Söhne zusammenfassen



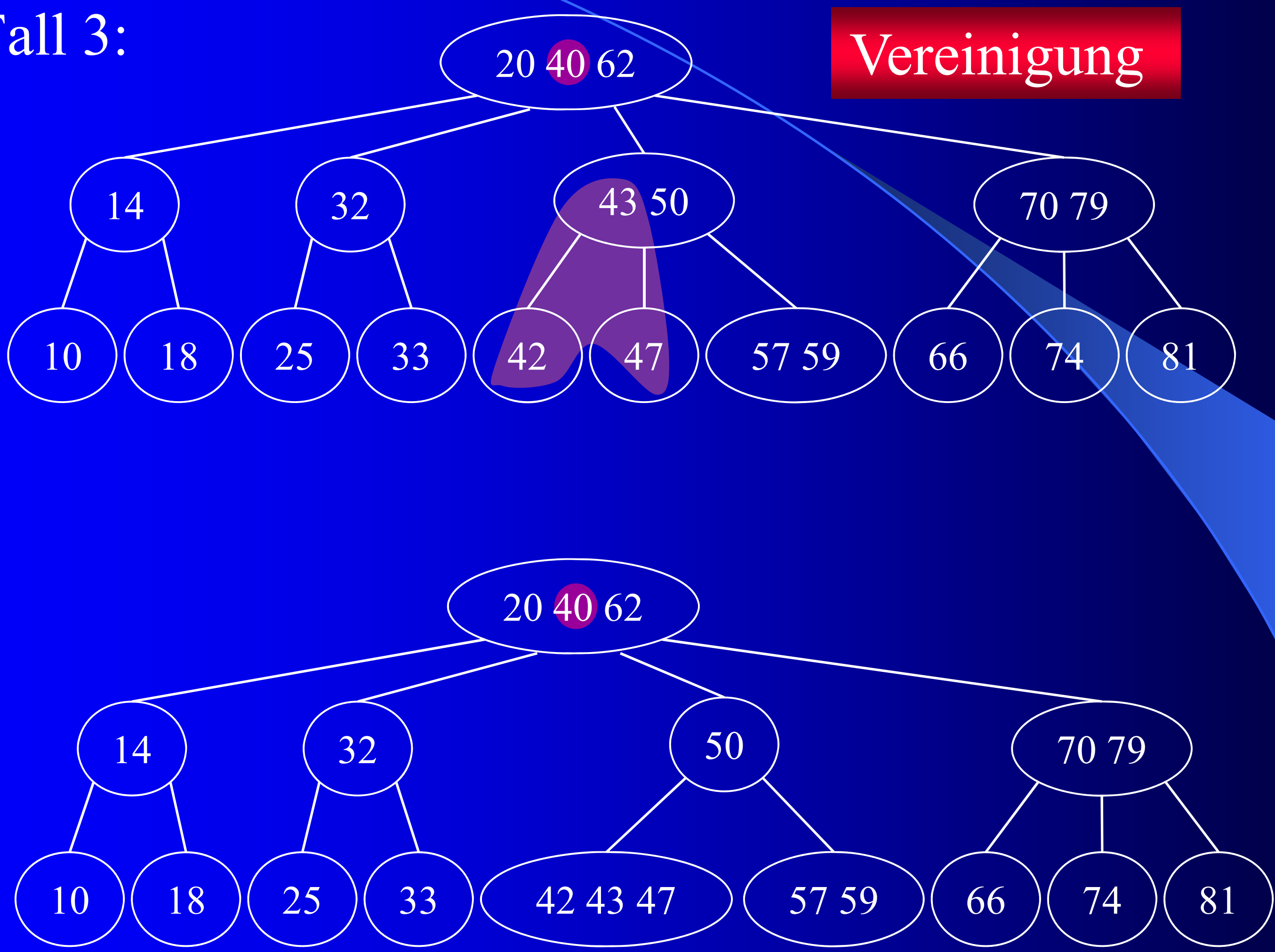
Beispiel (Forts.)

- Fall 2:



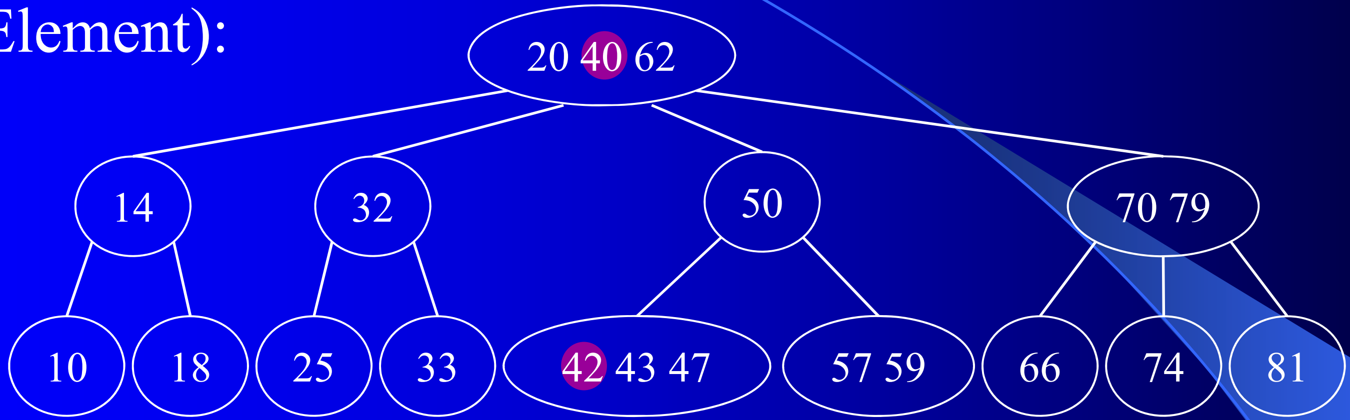
Beispiel (Forts.)

- Fall 3:

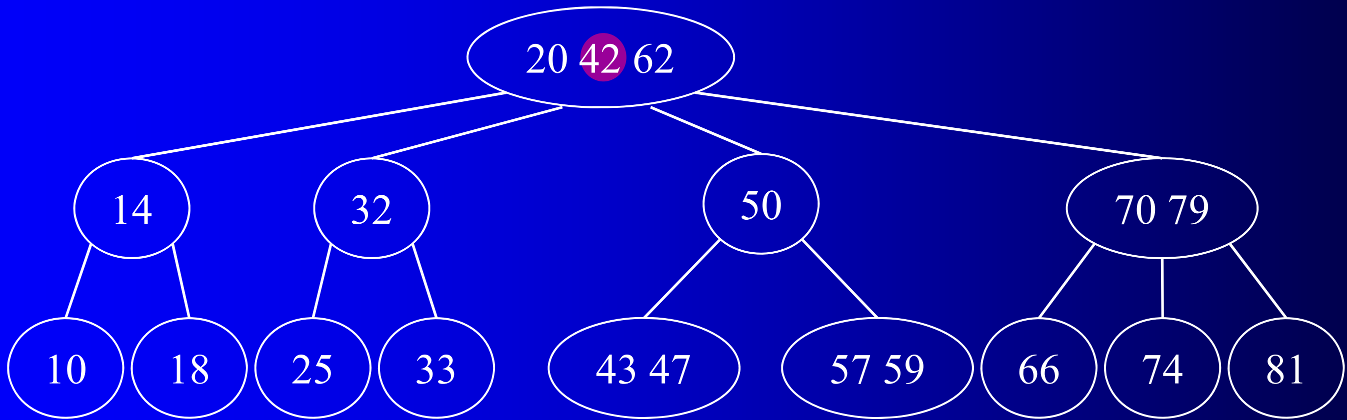


Beispiel (Forts.)

- 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)

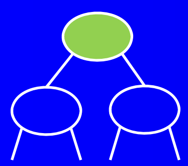
⇒ 26 (!!!) Fälle auf Ebene der Top-Down 2-3-4 Bäume

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

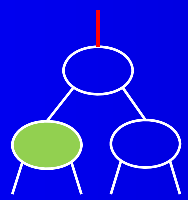
Fallunterscheidung (Forts.)

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

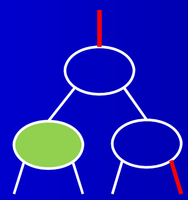
1. Wurzelfall



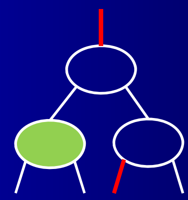
2. 2er unter 3er oder 4er mit 2er Bruder



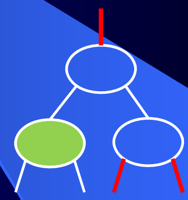
3. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder



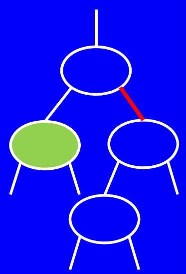
4. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder



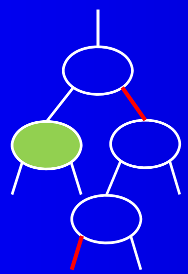
5. 2er unter 3er oder 4er oder Wurzel (!!!) mit 4er Bruder



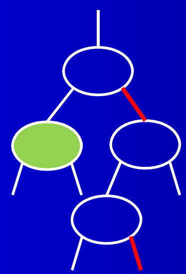
6. 2er unter 3er mit 2er Bruder



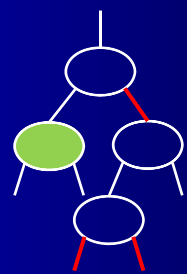
7. 2er unter 3er mit 3er Bruder



8. 2er unter 3er mit 3er Bruder

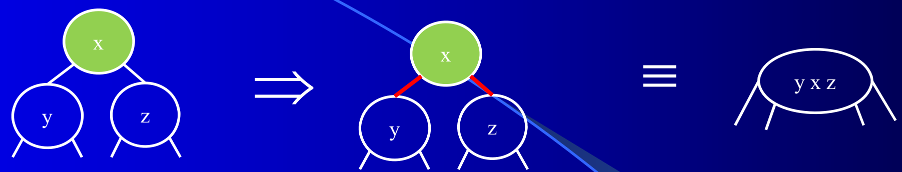


9. 2er unter 3er mit 4er Bruder

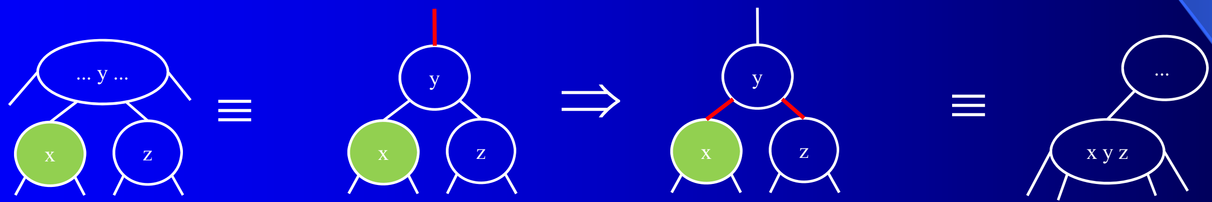


Fallunterscheidung (Forts.)

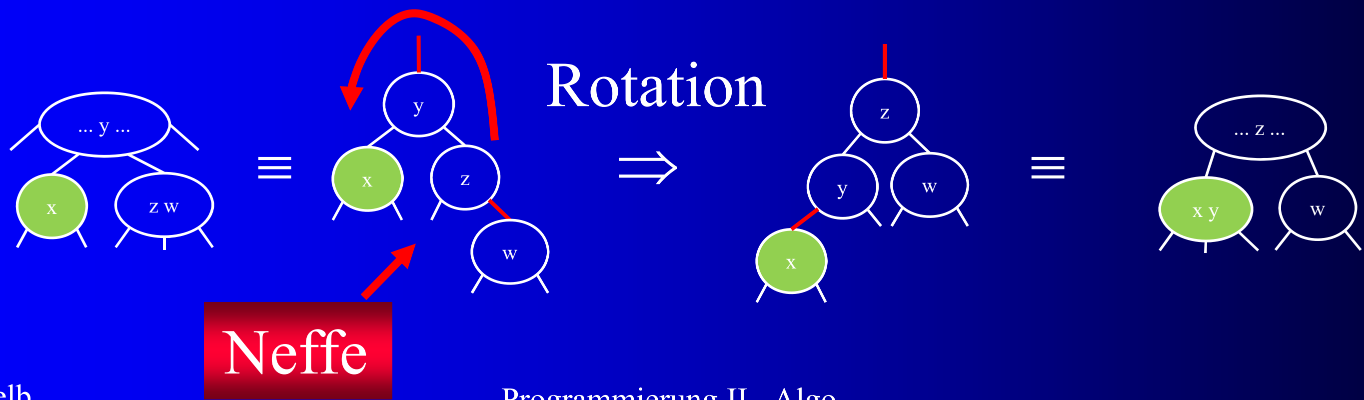
- 1. Wurzelfall



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

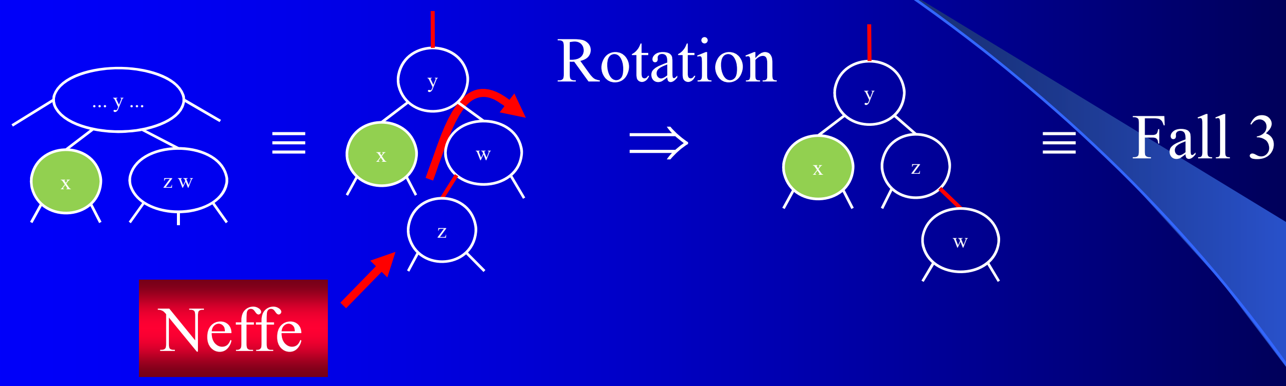


- 3. 2er unter 3er oder 4er oder Wurzel (!!!) mit 3er Bruder

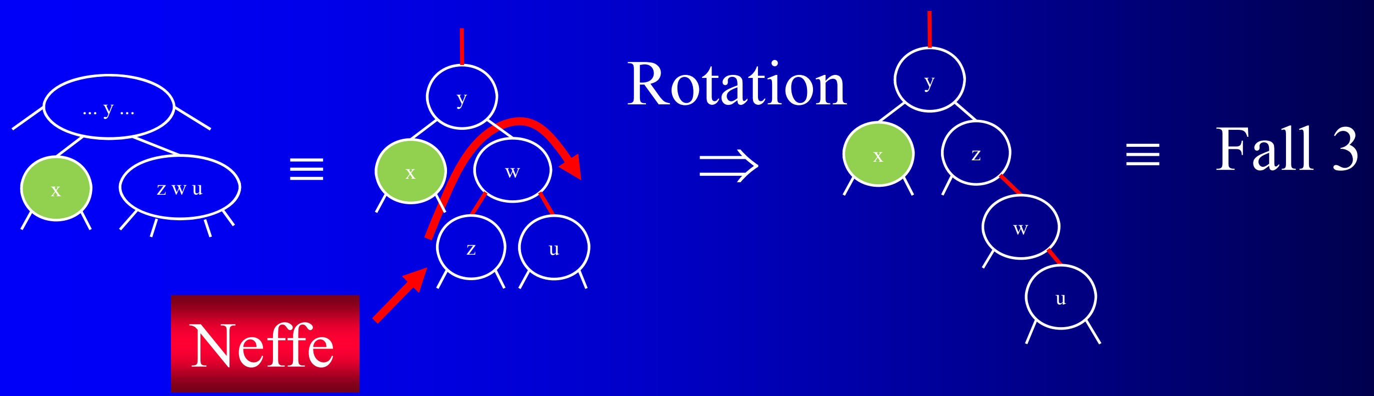


Fallunterscheidung (Forts.)

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

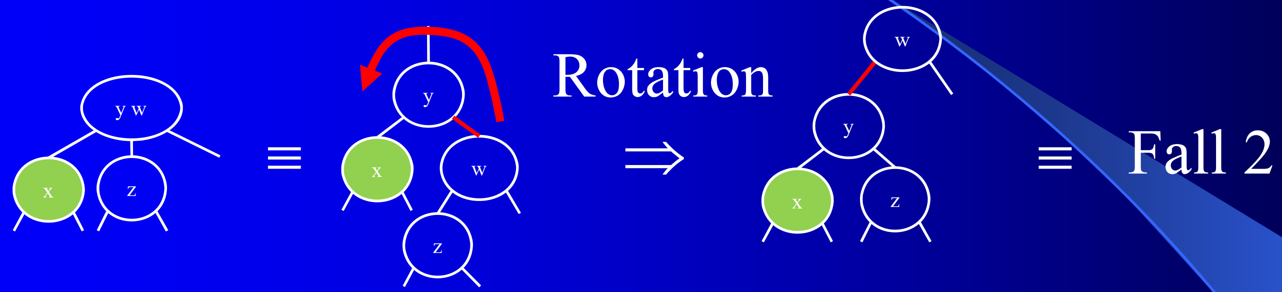


5. 2er unter 3er oder 4er oder Wurzel (!!!) mit 4er Bruder

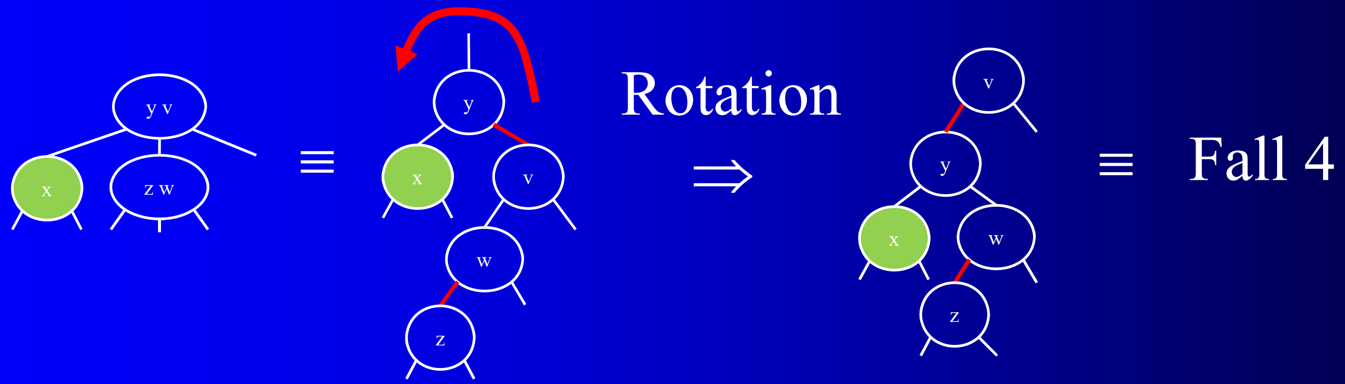


Fallunterscheidung (Forts.)

6. 2er unter 3er mit 2er Bruder

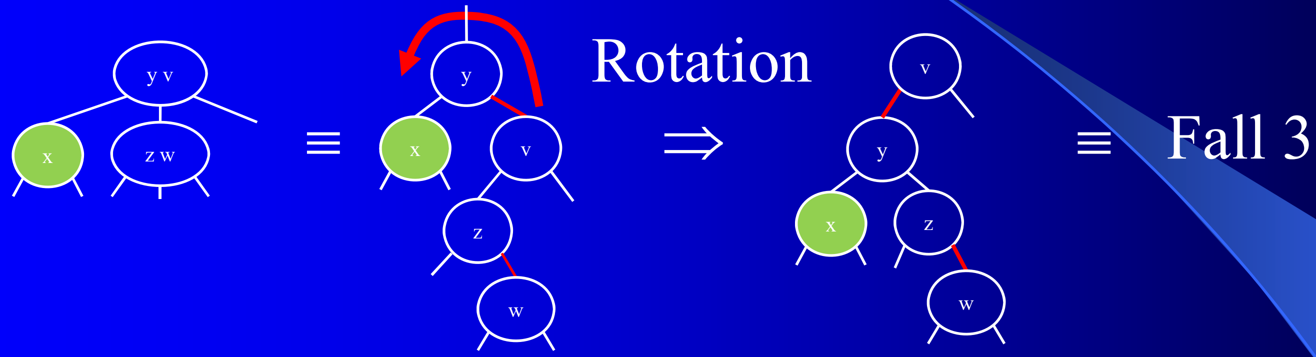


7. 2er unter 3er mit 3er Bruder



Fallunterscheidung (Forts.)

8. 2er unter 3er mit 3er Bruder



9. 2er unter 3er mit 4er Bruder