# 12. Such- und Sortierverfahren mit Arrays

Java-Beispiele:

suche.java
Quicksort.java
merge.java
Hash.java



## Schwerpunkte

- Zur Aufgabe
- Überblick, Klassifikation
- Suche: linear, binär
- Elementares Sortieren:
  Selectionsort, Bubblesort
- Schnelles Sortieren: Quicksort
- Schnelle Suche: Hash-Technik
- Externes Sortieren: Mergesort
- Komplexitätsbetrachtungen



## Aufgabe: Suchen und Sortieren

- Grundaufgabe vieler Softwaresysteme:
  Suche abgespeicherte Daten

  - Datenbanken
  - Compiler, Betriebssysteme
  - Verwaltung (Einwohner)
  - Banken ... (Kunden, Konten)

Sortieren ist Kein Selbstzweck

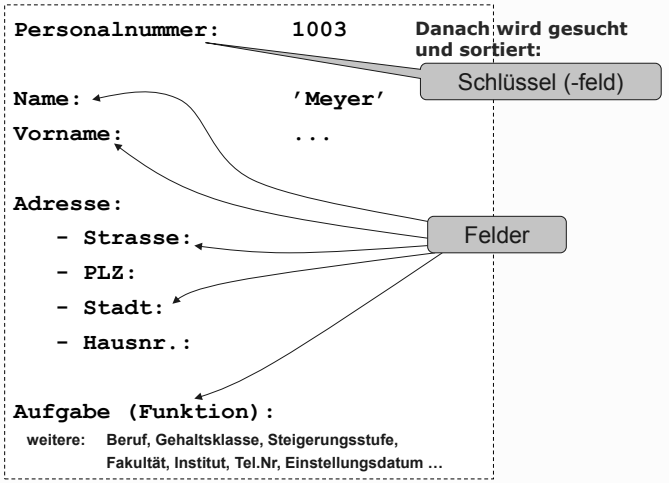
- Sortieren : Vereinfachung einer nachfolgenden Suche
- Besonderer Schwerpunkt : Effizienz
  (Komplexität von Algorithmen)

  - Datenbestände komplex
    (Einwohner von Berlin: 3,5 Mio)
  - Einzelne Daten(sätze) komplex
    (Bewegung / Austausch aufwendig)

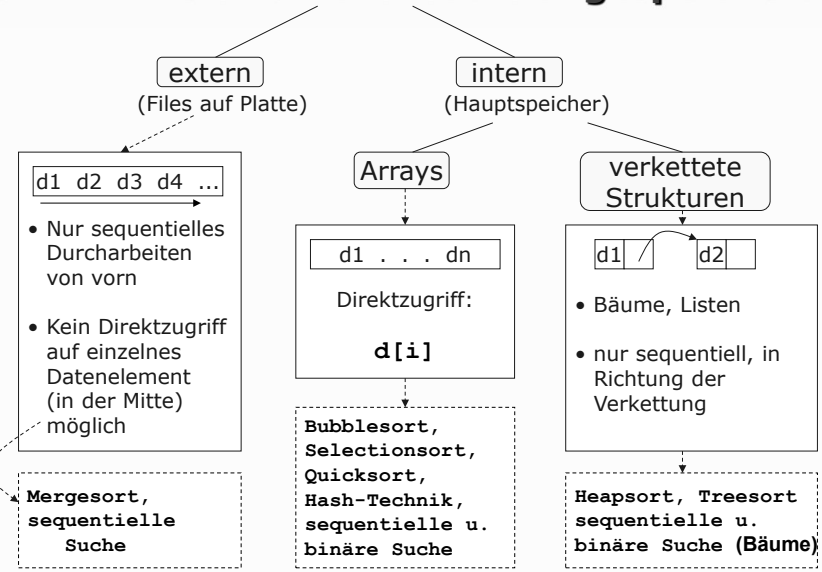


## Datensätze: komplexer Aufbau

z. B. Mitarbeiter (Einwohner)

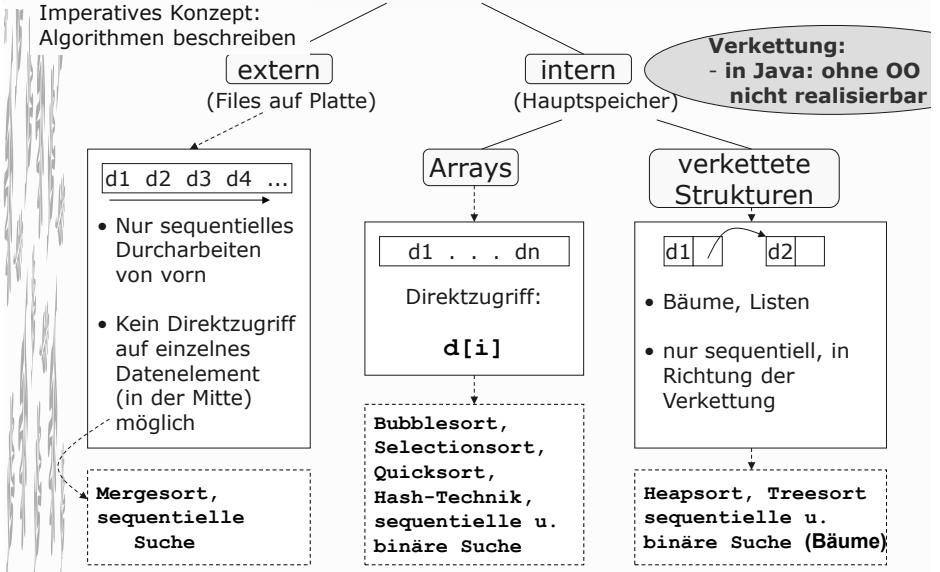


# Auswahl der Verfahren: Wo bzw. wie sind Datensätze abgespeichert ?






# Sortieren und Suchen sind imperative Konzepte – aber: Kapitel II.12: nur Arrays – Teil III: Verkettete Strukturen






# Komplexitätsklassen von Algorithmen (erweitert)

| | |
|---|---|
| logarithmisch: **(power1, binäre Suche)** | $O(n) = k * \log_2 n$ |
| linear: **(power, lineare Suche)** | $O(n) = k * n$ |
| n log 2 n: **(Quicksort, Mergesort, Heapsort)** | $O(n) = k * n \log_2 n$ |
| quadratisch: **( Selectionsort, Bubblesort)** | $O(n) = k * n^2$ |
| polynomiell: | $O(n) = k * n^m$ (m>1) |
| exponentiell: **(Hanoi)** | $O(n) = k * 2^n$ |

**Zusammenhang zwischen Größe der Eingabe n → Berechnungsaufwand O(n)**

Kurznotation: O(f(n)) für O(n) = k * f (n)

z. B. power1 ist $O(\log_2 n)$, Selectionsort ist $O(n^2)$



# Suchverfahren für Arrays

Sequentielle (lineare) Suche

Binäre Suche

# Lineare Suche in Arrays: Grundprinzip

Unsortiertes Feld:

| 100 | 6 | 33 | 77 | 39 | 20 | 20 | 206 | 200 |
|---|---|---|---|---|---|---|---|---|

**In Suchrichtung: alle Elemente nach gesuchtem Element (z. B. 39 ) durchsuchen**

Mittlerer Suchaufwand: O (n) = ½ n

d. h. sequentielle Suche hat lineare Komplexität

aber: 3,5 Millionen Einwohner . . .



# Lineare Suche in Arrays (mit Java)

```
public static void lineareSuche(int[] a, int x) {
   int i;

   for (i = 0; (i < a.length) && (x != a[i]); i++);

   if (i == a.length)
     System.out.println("Nicht gefunden");
   else
     System.out.println("Gefunden an Position " + i);
}
```

suche.java

Welche Anweisung wiederholt?

| 33 | 73 | 69 | 0 | 22 | 15 | 983 | 201 | 1 | 29 |
|---|---|---|---|---|---|---|---|---|---|

suche 1000: - - - - - -> i=10

suche 201: - - - -> i=7

Quelle: Java-Lehrbuch; vgl. suche.java in Java-Beispielsammlung



# Problem: for-Anweisung adäquat?

```
for (i = 0; (i < a.length) && (x != a[i]); i++);
```

- Grundgedanke von 'for':
  - feststehende Anzahl von Wiederholungen
  - Abbruchbedingung gibt obere Schranke für Laufvariable
  - Anweisung wiederholt für Bereich der Laufvariablen: ;

→ **for-Anweisung hier nicht gerechtfertigt!**

Natürliche Lösung:

Solange noch Elemente vorhanden

Aktuelles Element ist nicht das gesuchte

```
i = 0;
while ((i < a.length) && (x != a[i]))
      i++;
```



# Problem: Konjunktion kommutativ?

Solange noch Elemente vorhanden

Aktuelles Element ist nicht das gesuchte

```
for (i = 0; (i < a.length) && (x != a[i]); i++);
```

dasselbe ?

```
for (i = 0; (x != a[i]) && (i < a.length); i++);
```

Laufzeitfehler: falls Element nicht vorhanden

suche 1000:

| 33 | 73 | 69 | 0 | 22 | 15 | 983 | 201 | 1 | 29 |
|---|---|---|---|---|---|---|---|---|---|

i=10

Verkürzte Auswertung durch && entscheidend

# Binäre Suche: Halbierungsverfahren

**Voraussetzung:** Sortiertes Feld a:

| 2 | 5 | 7 | 10 | 20 | 55 | 77 | 78 | 80 | 100 | 101 |
|---|---|---|---|---|---|---|---|---|---|---|

**Algorithmus**: gesucht Element x (z. B. 80)

- Vergleiche x mit mittlerem Element a[m] des Feldes (z. B. m=5)
  - 1. Fall: x = a [m] → fertig: Element gefunden
  - 2. Fall: x > a [m] → suche im rechten Teilfeld
  - 3. Fall: x < a [m] → suche im linken Teilfeld
- Schluss: Teilfeld leer
  → fertig: Element kommt nicht vor



# Komplexität: binäre Suche

- Anzahl der Halbierungen:
  - schlechtester Fall:
    teile das Array, bis nur noch ein Element vorhanden
    → max. $\log_2 n$ Schritte
- Vergleich: lineare und binäre Suche

| Anzahl | 100 | 1024 | 1 Mio |
|---|---|---|---|
| Linear (im Durchschnitt) | 50 | 512 | 500.000 |
| Binär (maximal) | 7 | 10 | 20 |

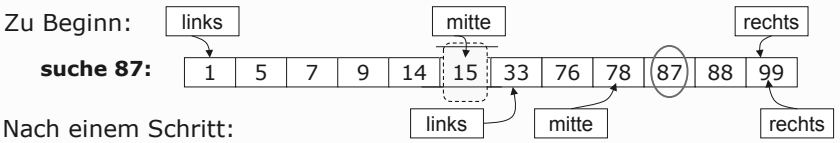


# Binäre Suche: Java-Programm

suche.java

```
public static void binaereSuche (int [] a, int x) {
   int links, rechts, mitte;

   links = 0;
   rechts = a.length-1;
   while (links <= rechts) {
         mitte = (links + rechts) / 2;
         if (a[mitte] == x) {
            System.out.println("Gefunden an Position " + mitte);
            return;
         }
         if (a[mitte] < x)
            links = mitte + 1; //suche rechts
         else
            rechts = mitte - 1; //suche links
   }
   System.out.println("Nicht gefunden");
}
```



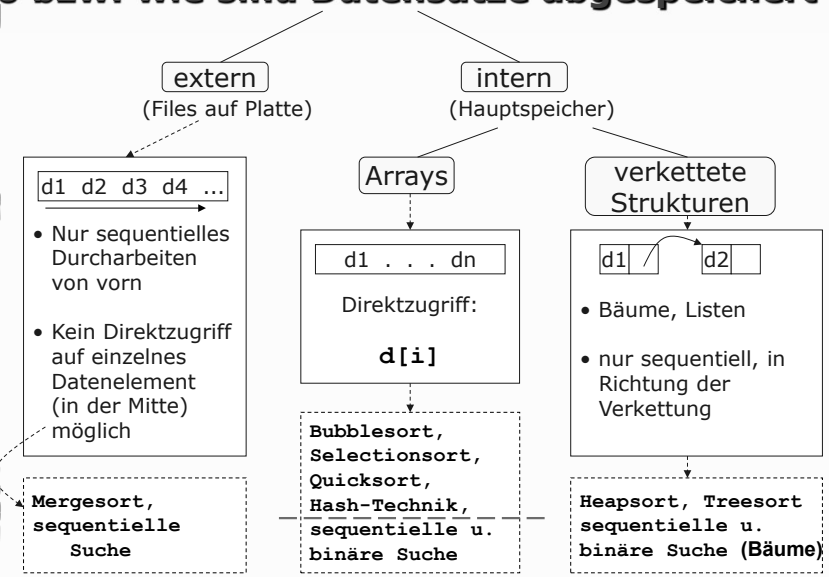



# Einfache Sortierverfahren:

Selectionsort

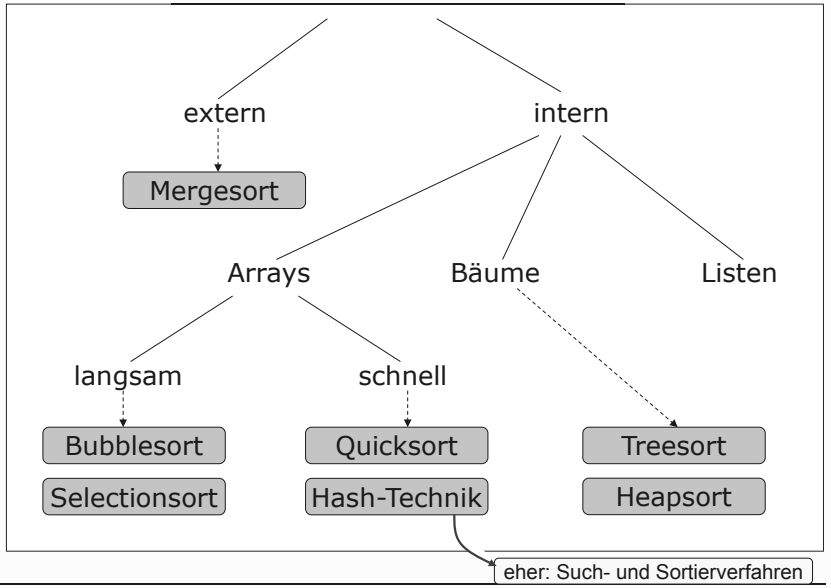
Bubblesort

# Auswahl der Verfahren: Wo bzw. wie sind Datensätze abgespeichert ?






# Sortierverfahren






# Selectionsort:

## Fast so wie ein Mensch sortieren würde ...

| 250 | 88 | 201 | 7 | 330 | 1021 | 56 |
|---|---|---|---|---|---|---|

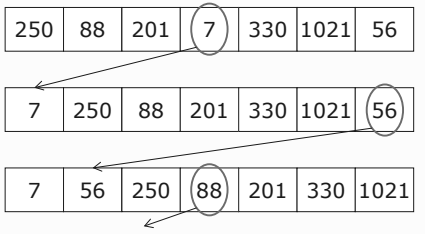


# Wie der Mensch sortieren würde ...

**Algorithmus – einfach, aber ineffizient:**

1. Man suche das kleinste Element und „schiebe" es vor die anderen.
2. Man suche das zweitkleinste Element und „schiebe" es an die zweite Position.

. . . u s w . . . .




**Problem?**

z.B. 3,5 Mill. Daten „schieben"

Weniger Daten bewegen?

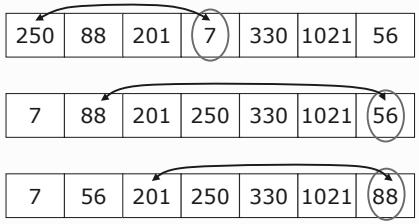
# Selectionsort: Sortieren durch Auswahl

**Algorithmus**:

1. Man suche das kleinste Element und vertausche es mit dem ersten Element

2. Man suche das zweitkleinste Element und vertausche es mit dem zweiten Element

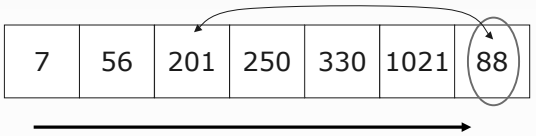
. . . u s w . . . .




Statt alle Elemente zu bewegen: nur 2 Elemente bewegen



# Komplexität: Selectionsort




- **Vertauschungen: O (n) linear**
  (genau: n - 1)

- **Vergleiche: O ($n^2$) quadratisch**
  (genau:
  (n-1) + (n-2) + … + 1
  = n * (n - 1) / 2
  = ($n^2$ – n) / 2 )
  = ½ $n^2$ – ½ n

**Komplexität**

Zusammenhang O:
Größe der Eingabe
→ Rechenaufwand



# Komplexitätsklasse durch höchste Dimension bestimmt

**Komplexitätsklasse bei Selectionsort: O ($n^2$)**

Obwohl genauer Wert für Anzahl der Vergleiche:
n * (n - 1) / 2 = ($n^2$ – n) / 2

Komplexitätsklasse: O($n^2$) oder O($n^2$ – n)?

Komplexitätsklasse: O($n^2$)
- da geringere Klasse bei großem n ohne Einfluss
- höhere Klasse bestimmt die Größenordnung
→ Geringere Klasse wird vernachlässigt.
→ O($n^2$ – n) existiert nicht (wird nicht betrachtet)



# Komplexitätsklassen: geringere Dimension kann vernachlässigt werden

| | 2 | 8 | 10 | 100 | 1000 |
|---|---|---|---|---|---|
| konstant | 1 | 1 | 1 | 1 | 1 |
| logarithmisch (power1) | 1 | 3 | 4 | 7 | 10 |
| linear (power) | 2 | 8 | 10 | 100 | 1000 |
| quadratisch | 4 | 64 | 100 | 10.000 | 1.000.000 |
| exponentiell (Hanoi) | 4 | 256 | 1024 | ~10 Mrd. | ~$10^{100}$ |

$n^2 - n$

Anzahl der Vergleiche: ½ ($n^2$ – n)

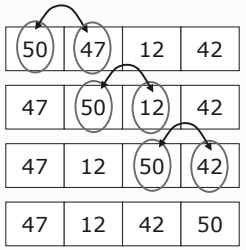
# Bubblesort: Austausch von Nachbarn

**Algorithmus**:

1. Durchsuche das Array und vertausche dabei benachbarte Elemente, die nicht in der richtigen Reihenfolge stehen.
   → Ergebnis: größtes Element rechts.

2. Wie unter 1. - nur bis zum vorletzten Element.

. . . u s w. . . .

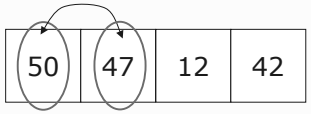



1. Schritt

bubble:
kochen,
sieden,
übersprudeln



# Komplexität: Bubblesort




- **Vertauschungen: $O(n^2)$**
  - minimal: 0
  - maximal: $(n-1) + (n-2) + \ldots + 1$
    $= \frac{1}{2}(n^2-n)$
  - durchschnittlich: $\frac{1}{4}(n^2-n)$

- **Vergleiche: $O(n^2)$**
  genau: **$\frac{1}{2}$** $(n^2 - n)$
  (wie maximale Vertauschungen)

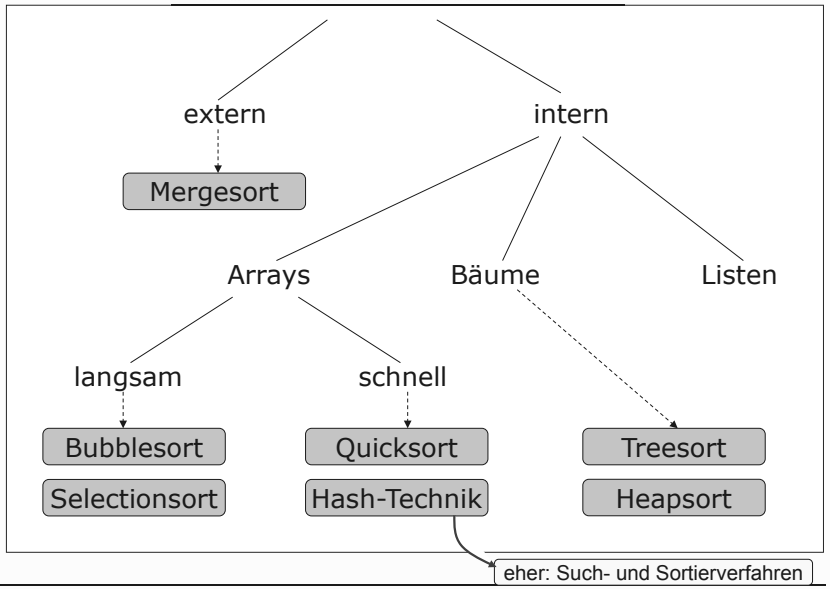


# Quicksort:

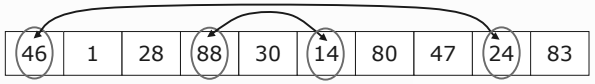
## Schnelles Sortieren durch rekursives Teilen



# Sortierverfahren



# Quicksort: Grundidee




**Rekursiver Algorithmus (Idee)**:

- Zerlege das gesamte Array durch Austausch von Elementen in 2 Teile:
  Alle Elemente des linken Teils sind
  <= alle Elemente des rechten Teils
- Sortiere anschließend beide Teile ***unabhängig*** voneinander (rekursiv).

| 24 | 1 | 28 | 14 | 30 | 88 | 80 | 47 | 46 | 83 |
|---|---|---|---|---|---|---|---|---|---|

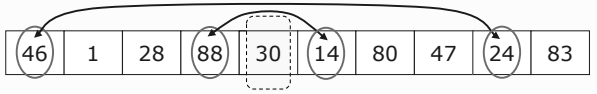


# Quicksort: Zerlegung in zwei Teile

Zerlege das gesamte Array durch Austausch von Elementen in 2 Teile:
alle Elemente des linken Teils <= alle Elemente des rechten Teils

**Technik:**

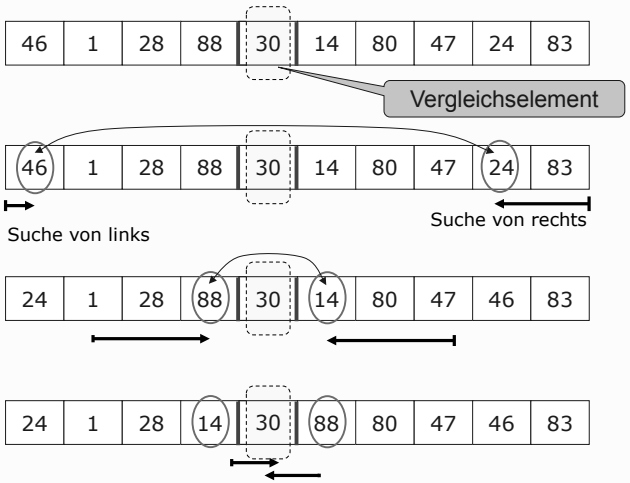
- Wähle beliebiges Element aus dem Array
  (meist: mittleres Element, Pivotelement, Vergleichselement)
- Suche Array von links bis:
  Element >= Vergleichselement gefunden
- Suche Array von rechts bis:
  Element <= Vergleichselement gefunden
- Tausche beide Elemente aus






# Beispiel: Quicksort (1)

1. Teilungsschritt






# Beispiel: Quicksort (2)

nach 1. Teilungsschritt:

| 24 | 1 | 28 | 14 | 30 | 88 | 80 | 47 | 46 | 83 |
|---|---|---|---|---|---|---|---|---|---|

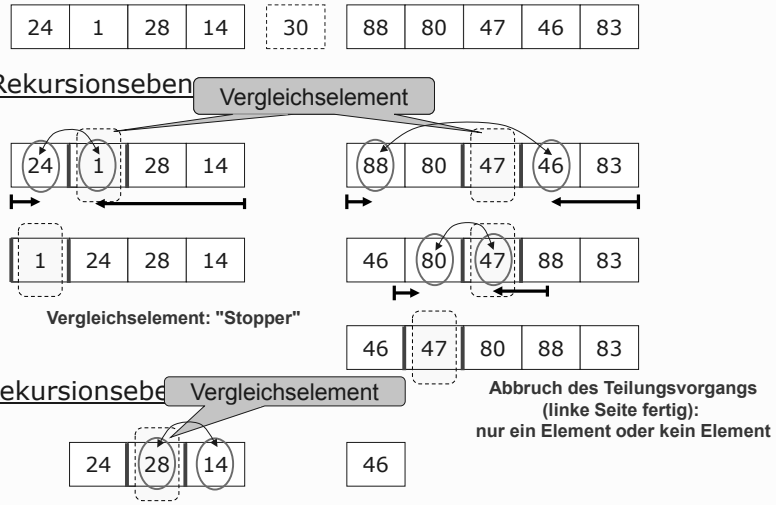
1. Rekursionsebene:


Rekursiv: linke Seite – rechte Seite
nach demselben Algorithmus sortieren (Quicksort)

Besonderheiten:
- Vergleichselement u.U. ausgetauscht
- Vergleichselement dient als „Stop"-Element
- Bereiche mit nur einem Element entstehen: fertig

# Beispiel: Quicksort (3)

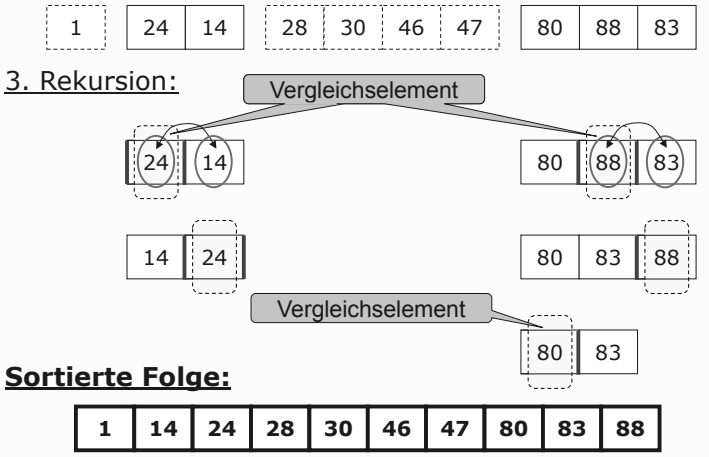
nach 1. Teilungsschritt:






# Beispiel: Quicksort (4)

nach 4 Teilungsschritten:




**Sortierte Folge:**

| 1 | 14 | 24 | 28 | 30 | 46 | 47 | 80 | 83 | 88 |
|---|---|---|---|---|---|---|---|---|---|



# Zerlegung in zwei Teile: Java-Programm

```
public static void quicksort (int[] a, int links, int rechts) {
  int help ;
  int i = links;
  int j = rechts;
    // Vergleichselement:
  int x = a[(links+rechts) / 2];
  do {
     while (a[i] < x) i++;
     while (a[j] > x) j--;
     if ( i<=j ) {
        help = a[i];
        a[i] = a[j];
        a[j] = help;
        i++; j--;
     }
  } while (i <= j);
   // Jetzt gilt: Elemente im linken Teil sind kleiner
   // als Elemente im rechten Teil
   // -> danach: sortiere linken und rechten Teil einzeln:
  . . .
}
```

Quicksort.java

überlese: linke kleine Elemente

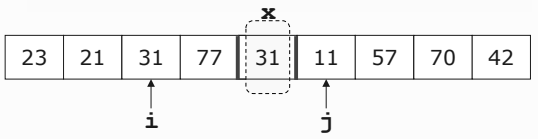
überlese: rechte größere Elemente

Tausche aus, falls Elemente auf der falschen Seite stehen

Solange, bis Teilung vollzogen

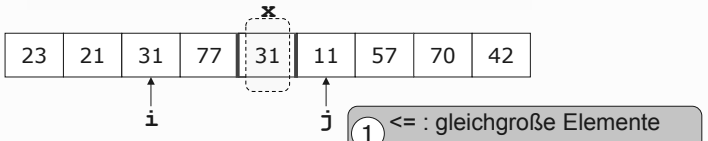


# Details: Vergleiche




```
do {
   while (a[i] < x) i++;
   while (a[j] > x) j--;
   if ( i <= j ) {
       help = a[i];
       a[i] = a[j];
       a[j] = help;
       i++; j--;
   }
} while (i <= j);
```

# Details: Vergleiche




```
do {
   while (a[i] < x) i++;
   while (a[j] > x) j--;
   if ( i <= j ) {
      help = a[i];
      a[i] = a[j];
      a[j] = help;
      i++; j--;
   }
} while (i <= j);
```

1 <= : gleichgroße Elemente stehen lassen?

Algorithmus terminiert u. U. nicht: „Stopp-Element" fehlt (z.B. alle Elemente gleich)

3 i < j : nicht tauschen bei i = j ?

Bei i < j klappt Abbruchbedingung von do-while nicht mehr

2 i < j : halten bei i = j ?

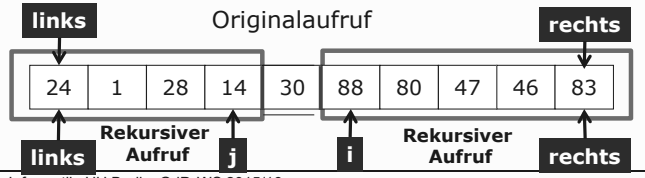
Mittleres Element würde rekursiv mitsortiert (Effizienz)



# Separate Sortierung: linker und rechter Teil

```
public static void quicksort (int[] a, int links, int rechts) {
   ...
   Austausch: linker (kleine) und rechter Teil (große Elemente)
   // danach: sortiere linken und rechten Teil einzeln,
   // falls mehr als ein Element vorhanden

   if (links < j)
      quicksort(a, links, j);
   if (i < rechts )
      quicksort(a, i, rechts );
}
```






# Quicksort auch iterativ möglich ?

```
public static void quicksort
      (int[] a, int links, int rechts)  {

   //zerlege Array a zwischen
   //links und rechts in zwei Teile
   do {...}
   while (...);

   if (links < j)
      quicksort(a, links, j);
   if (i < rechts)
      quicksort(a, i, rechts);
}
```

vgl. „Türme von Hanoi"

→ Problemstack

Was ist ein Problem?



# Quicksort: 'Problem-Stack'

Abzuspeicherndes Problem: der zu sortierende Indexbereich

Aufruf in Quicksort.java:

z.B. 100

```
quicksort(a, 0, n-1);
```

Stackentwicklung:

| | |
|---|---|
| Anfang: | 0 100 |
| 1. Zyklusschritt: | 0 68<br>70 100 |
| 2. Zyklusschritt: | 0 21<br>23 68<br>70 100 |

Problem**stack** (Stapel) notwendig ?

Kommt es auf die Reihenfolge der zu lösenden Probleme an?

Falls aktuelles Problem nur 1 Element enthält: streiche es vom Stack

# 12. Such- und Sortierverfahren mit Arrays

## Teil 2

Java-Beispiele:

suche.java
Quicksort.java
merge.java
Hash.java



# Nachtrag

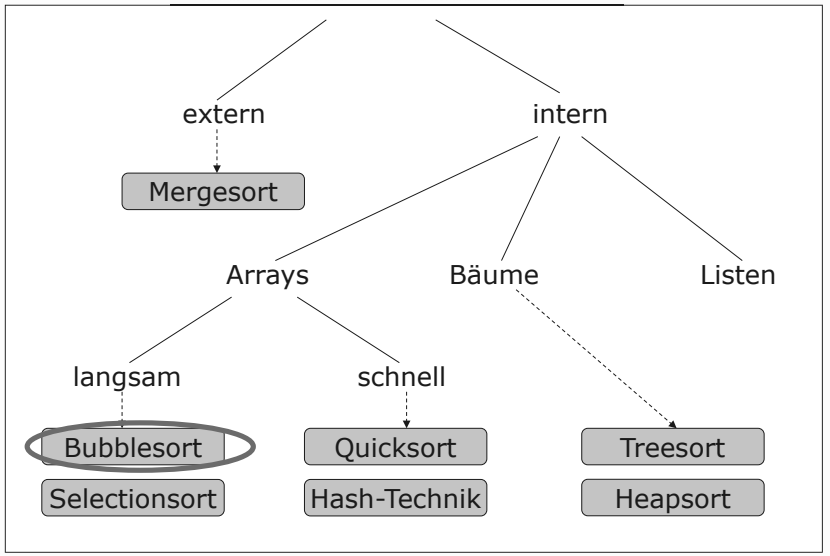
**(Folien schon veröffentlicht)**

Bubblesort

Komplexität von Quicksort



# Sortierverfahren





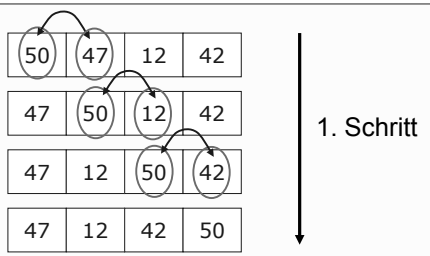

# Bubblesort: Austausch von Nachbarn

**Algorithmus**:

1. Durchsuche das Array und vertausche dabei benachbarte Elemente, die nicht in der richtigen Reihenfolge stehen.
   → Ergebnis: größtes Element rechts.

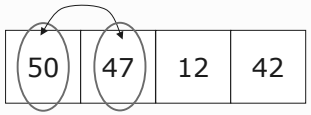
2. Wie unter 1. - nur bis zum vorletzten Element.

. . . u s w. . . .




bubble:
kochen,
sieden,
übersprudeln

# Komplexität: Bubblesort

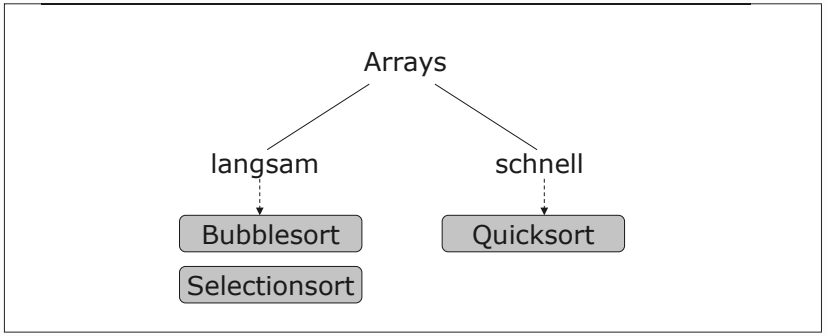



- **Vertauschungen: $O(n^2)$**
  - minimal: 0
  - maximal: (n-1) + (n-2) + … + 1
    = ½($n^2$-n)
  - durchschnittlich: ¼ ($n^2$-n)

- **Vergleiche: $O(n^2)$**
  genau: **½ (**$n^2$ – n)
  (wie maximale Vertauschungen)



# Sortierverfahren mit Arrays




**Lohnt es sich, anstelle der einfachen und übersichtlichen Sortierverfahren das kompliziertere Quicksort zu verwenden?**



# Komplexität: Quicksort

- **Günstigster Fall :** Ohne Beweis
  - das Array wird immer in zwei gleichgroße Teile geteilt.
  - → Vergleiche: $O(n) = n \log_2 n$
  - → Austauschoperationen: $O(n) = n \log_2 n / 6$

- **Mittlere Komplexität:**
  - nur um Faktor 2 * ln 2 = 1.39... schlechter

- **Schlechtester Fall:** $O(n) = n^2$
  - immer nur ein Element abgespalten (wie Selectionsort)



# Komplexität: Quicksort

- **Günstigster Fall :** Ohne Beweis
  - das Array wird immer in zwei gleichgroße Teile geteilt.
  - → Vergleiche: $O(n) = n \log_2 n$
  - → Austauschoperationen: $O(n) = n \log_2 n / 6$

- **Mittlere Komplexität:**
  - nur um Faktor 2 * ln 2 = 1.39... schlechter

- **Schlechtester Fall:** $O(n) = n^2$
  - immer nur ein Element abgespalten (wie Selectionsort)

**Beispiel mittlere Komplexität:** (vgl. 3,5 Mill. Einwohner)

| | | |
|---|---|---|
| n = 1.000.000 | Selectionsort | 1.000.000.000.000 |
| → Vergleiche | Quicksort | 20.000.000 |

50.000 mal mehr Vergleiche (1 Minute Qsort -> 1 Monat Ssort)

# Komplexitätsklassen von Algorithmen (erweitert)

| | |
|---|---|
| logarithmisch: **(power1, binäre Suche)** | $O(n) = k * \log_2 n$ |
| linear: **(power, lineare Suche)** | $O(n) = k * n$ |
| n log 2 n: **(Quicksort, Mergesort, Heapsort)** | $O(n) = k * n \log_2 n$ |
| quadratisch: **( Selectionsort, Bubblesort)** | $O(n) = k * n^2$ |
| polynomiell: | $O(n) = k * n^m$ (m>1) |
| exponentiell: **(Hanoi)** | $O(n) = k * 2^n$ |

**Zusammenhang zwischen Größe der Eingabe n → Berechnungsaufwand O(n)**

Kurznotation: O(f(n)) für O(n) = k * f (n)

z. B. power1 ist $O(\log_2 n)$, Selectionsort ist $O(n^2)$



# Hash-Technik: Suche mit einem Zugriff



# Komplexitätsklassen von Algorithmen (erweitert)

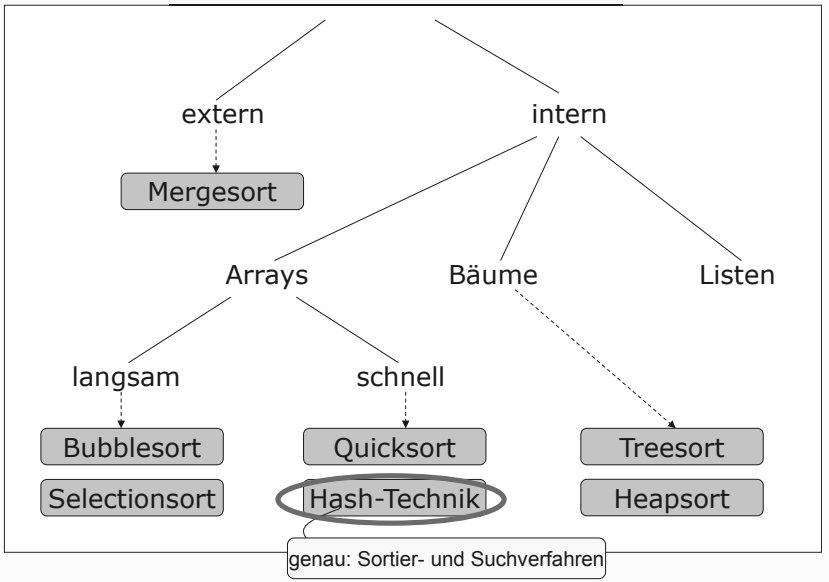
| | |
|---|---|
| konstant **Hash-Technik** | |
| logarithmisch: **(power1, binäre Suche)** | $O(n) = k * \log_2 n$ |
| linear: **(power, lineare Suche)** | $O(n) = k * n$ |
| n log 2 n: **(Quicksort, Mergesort, Heapsort)** | $O(n) = k * n \log_2 n$ |
| quadratisch: **( Selectionsort, Bubblesort)** | $O(n) = k * n^2$ |
| polynomiell: | $O(n) = k * n^m$ (m>1) |
| exponentiell: **(Hanoi)** | $O(n) = k * 2^n$ |

**Zusammenhang zwischen Größe der Eingabe n → Berechnungsaufwand O(n)**

Kurznotation: O(f(n)) für O(n) = k * f (n)



# Sortierverfahren



# Hash–Technik: Eigenschaften

- Schnellstes Suchverfahren
- Ziel : Suche mit e i n e m Zugriff:
  d. h. konstante Komplexität O (n) = k
  → Vielleicht sogar mit k = 1 …
- Laufzeit auf Kosten von Speicherplatz



# Hash-Technik: Grundidee

Hash-sortiertes Feld :

| - | - | 3 | 50 | - | 40 | - | 30 | - | 20 | 2 | 10 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Array der Länge 13 :
- z. T. unbesetzt (Streuspeicherung)
- äußerlich unsortiert

Idee: Element nicht suchen,
sondern Position im Feld berechnen

Hash-Funktion H :

H : Schlüssel → Adressen

**hash = klein hacken**



# Beispiel: Hash-Funktion

| - | - | 3 | 50 | - | 40 | - | 30 | - | 20 | 2 | 10 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Ziel:
H : Integer → [ 0 ... 12 ]

Spezielle Hash-Funktion:
H (n) = 5 * n mod 13

H (30) = 150 % 13 = 7
H (3) = 15 % 13 = 2
H (10) = 50 % 13 = 11



# Beispiel: Hash-Funktion

| - | - | 3 | 50 | - | 40 | - | 30 | - | 20 | 2 | 10 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Ziel:
H : Integer → [ 0 ... 12 ]

Spezielle Hash-Funktion:
H (n) = 5 * n mod 13

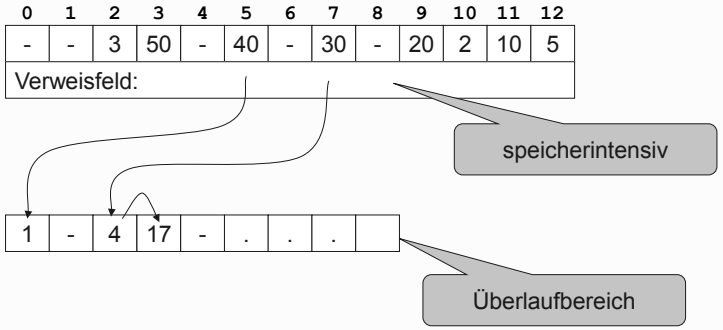
**Kollision:**

H (30) = 150 % 13 = 7
H (4) = 20 % 13 = 7

Kollision: H (n1) = H (n2)

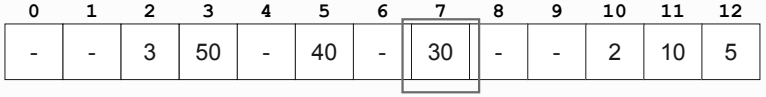
Was tun?

# Kollisionsbehandlung: Überlaufbereich





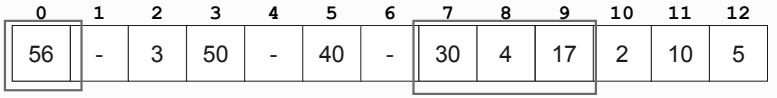

# Kollisionsbehandlung: offene Verkettung




Trage neu ein : 4, 17, 56

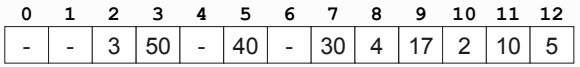
- H ( 4 ) = H ( 17 ) = H ( 56 ) = 7

→ Falls Platz belegt :
trage in nächsten freien Platz ein
(am Ende: wieder von vorn beginnen)






# Suchen bei offener Verkettung




Algorithmus: Wie findet man **s**?

**Suche ein Element s :**

H (s) = i (an Stelle i müsste s stehen)

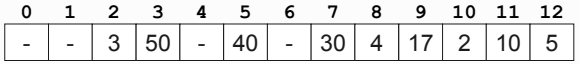
**1. Fall** : a [ i ] = s
→ Element gefunden

**2. Fall** : a [ i ] = - (Lücke)
→ Element nicht vorhanden

**3. Fall** : a [ i ] < > s (keine Lücke)
→ suche beginnend mit Position i + 1 das Array nach Element s
(solange bis eine Lücke '-' auftritt oder bis Element gefunden )



# Beispiele: Suchen bei offener Verkettung




H (n)
= 5 * n mod 13

1. Nicht vorhanden: s = 1 , H ( 1 ) = 5
   → a [ 5 ] < > 1
   → suche bei a [ 6 ] weiter : '-' fertig

2. Vorhanden: s = 17 , H ( 17 ) = 7
   → a [ 7 ] < > 17
   → suche von der Position i = 8 weiter:
   a [ 9 ] = 17 , d. h. fertig : gefunden

3. Nicht vorhanden: s = 56, H ( 56 ) = 7
   → a [ 7 ] < > 17
   → suche von der Position i = 8 weiter:
   i = 8 , 9 , 10 , 11 , 12 , 0 : '-' fertig

# Implementation der Hash-Technik: Was ist zu tun?

1. Größe der Hash-Tabelle festlegen
   (Array mit festem Indexbereich)
   ```
   int [] table = new int [tableSize];
   ```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | - | 3 | 50 | - | 40 | - | 30 | 4 | 17 | 2 | 10 | 5 |

2. Hash-Funktion festlegen (und implementieren)
   H : Integer ---> [ 0 . . tablesize-1]
   ```
   int hash (int key, int tableSize)
   ```

3. Implementation der Operationen:
   ```
   eintragen (int key)
   suchen (int key)
   ```

→ in Hash.java: eintragen und suchen fehlen noch
→ Aufgabe: selbst implementieren

Was ist invariant bzgl. der Aufgabe - d. h. was kann wiederverwendet werden bei neuer Hashtabelle?



# Hash-Funktionen: Definitionsbereiche

- Numerische Schlüsselfelder
  (Ausweisnummern, Personalnummern)

hash: int → Adressbereich

(arithmetische Operationen)

- Namen (Mitarbeiter), Bezeichner (Compiler) ...

hash: String → Adressbereich

(Transformation von Zeichen nach Unicode
+ arithmetische Operationen)



# Hash-Funktion für Strings

Hash.java

```
static int hash (String key, int tableSize) {

   int hashVal = 0;

   for (int i = 0; i < key.length(); i++)
       hashVal = 37 * hashVal + key.charAt(i);

   hashVal %= tableSize;
   if (hashVal < 0)
       hashVal += tableSize;

   return hashVal;
}
```

**Typ „String": Zeichenketten**

String:
API-Klasse
*(kein Array)*
Typ mit Operationen

`key.length() - 1`

| i: | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| key: | p | e | r | s | N | r |

nicht: `key[4]`

`key.charAt(4)`

hashVal: 750

z.B „Müller", „persNr", „aa"



# Hash-Funktion für Strings: Details

z. B.: `"aa"` 1000

```
static int hash (String key, int tableSize) {

   int hashVal = 0;

   for (int i = 0; i < key.length(); i++)
       hashVal = 37 * hashVal + key.charAt(i);

   hashVal %= tableSize;
   if (hashVal < 0)
       hashVal += tableSize;

   return hashVal;
}
```

int + char ?

?

37 * 97 + 'a'

für i = 1

37 * 97 + 97

Korrektur negativer Hashwerte?

automatische Konvertierung Unicode / ASCII

hashVal > maxInt = $2^{31}$-1 möglich

# Anwendung der Hash-Funktion

Hash.java

```
public static void main (String[] args) {
    String str;
    int length;

    System.out.print("Enter table length: ");
    length = Keyboard.readInt();

    while (true) {
        System.out.print("Enter a string: ");
        str = Keyboard.readString();
        System.out.println("String: " + str
                + "   Hash value: "
                + hash(str, length));
        if (str.equals("0")) return;
    }
}
```

String-Vergleich

```
% java Hash
Enter table length: 1000
Enter a string: abcdef
String: abcdef   Hash value: 401
```



# Effizienzbetrachtungen

**L** : Ladefaktor einer Hash-Tabelle
(Anteil der gefüllten Plätze der Tabelle)

**L** = 0.5

- Einfügen: durchschnittlich 2.5 betrachtete Plätze
- Suchen: durchschnittlich 1.5 betrachtete Plätze (erfolgreiche Suche)

**L** = 0.9

- Einfügen: durchschnittlich 50 betrachtete Plätze
- Suchen: durchschnittlich 5.5 betrachtete Plätze (erfolgreiche Suche)

**Unabh. von Anzahl n der abgespeicherten Elemente**

→ Komplexitätsklasse: konstant



# Komplexitätsklassen von Algorithmen (erweitert)

konstant
**Hash-Technik**
logarithmisch: $O(n) = k * \log_2 n$
**(power1, binäre Suche)**
linear: $O(n) = k * n$
**(power, lineare Suche)**
n log 2 n: $O(n) = k * n \log_2 n$
**(Quicksort, Mergesort, Heapsort)**
quadratisch: $O(n) = k * n^2$
**( Selectionsort, Bubblesort)**
polynomiell: $O(n) = k * n^m$ (m>1)

exponentiell: $O(n) = k * 2^n$
**(Hanoi)**

**Zusammenhang zwischen Größe der Eingabe n → Berechnungsaufwand O(n)**

Kurznotation: O(f(n)) für O(n) = k * f (n)



# Java: API-Klasse 'Hashtable'

- **Selbst implementieren:**
  - Methode 'hash' für die Berechnung der Hash-Funktion
  - equals() – bestimmt die Gleichheit von abgespeicherten Elementen / Objekten
- **Parameter:**
  - capacity: Größe der Tabelle
  - loadFactor: Ladefaktor L
    - → Ist die Tabelle zu L Prozent gefüllt : automatische Vergrößerung der Tabelle (Laufzeitaufwand)
    - → zu volle Tabellen: viele Kollisionen, d. h. lange Suchzeiten

# Webseite: API-Klasse Hashtable

java.util
**Class Hashtable<K,V>**

java.lang.Object
  └ java.util.Dictionary<K,V>
    └ **java.util.Hashtable<K,V>**

**All Implemented Interfaces:**
Serializable, Cloneable, Map<K,V>

**Direct Known Subclasses:**
Properties, UIDefaults

---

public class **Hashtable<K,V>**
extends Dictionary<K,V>
implements Map<K,V>, Cloneable, Serializable

This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. The initial capacity and load factor parameters are merely hints to the implementation. The exact details as to when and whether the rehash method is invoked are implementation-dependent.

If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

This example creates a hashtable of numbers. It uses the names of the numbers as keys:

```
Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

To retrieve a number, use the following code:

```
Integer n = (Integer)numbers.get("two");
if (n != null) {
    System.out.println("two = " + n);
}
```

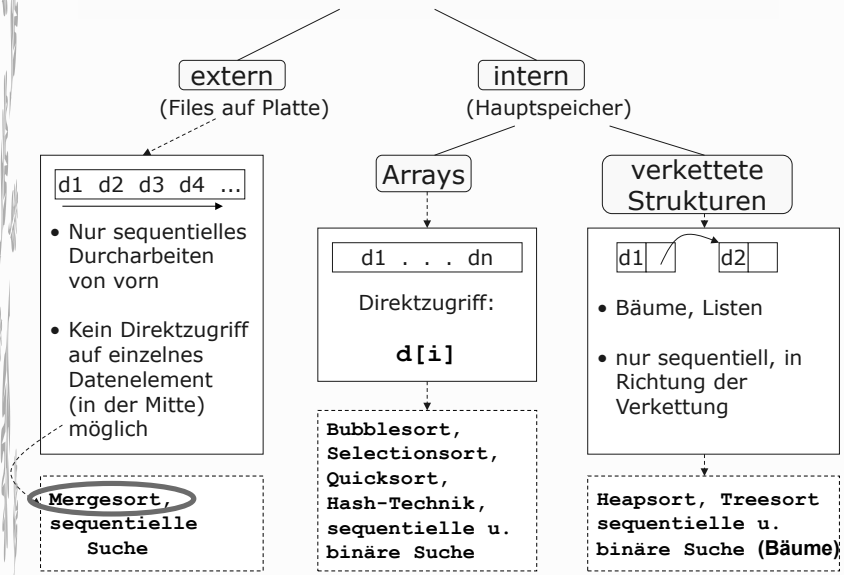


# Mergesort: Sortieren externer Files

**(demonstriert anhand von Arrays)**



# Sortier- und Suchverfahren

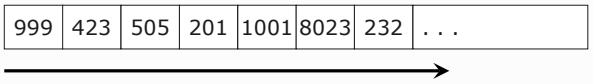





# Mergesort: Sortieren externer Files durch Mischen

Riesige Datenmengen: passen nicht als Ganzes in Hauptspeicher

- → sequentielle Folgen (Files: externe Speicher)
- → immer nur ein Element sequentiell zugreifbar: nur Lesen von links nach rechts
- → Kein Austausch von Elementen des Files, kein Direktzugriff

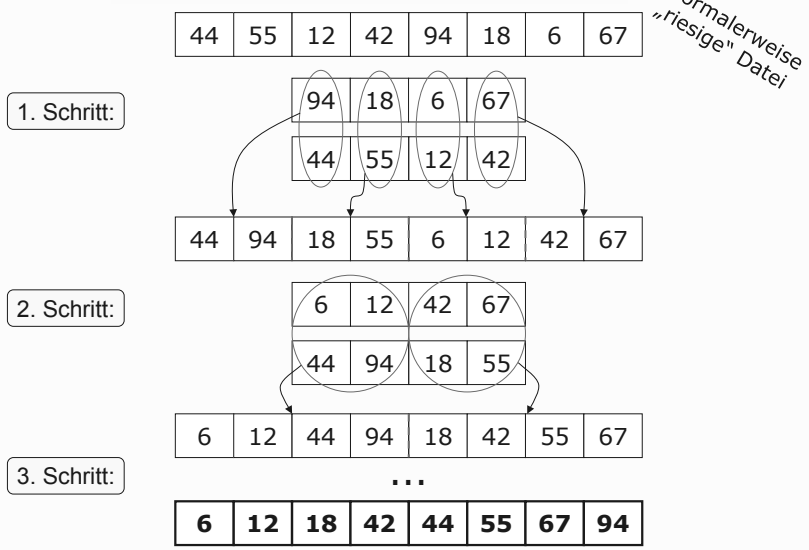



**Algorithmus:**

1. Zerlege die Folge in 2 Teilfolgen a und b
2. Mische a und b : Folge c sortierter Paare
3. Zerlege Folge c in 2 Teilfolgen a1 und b1
4. Mische a1 und b1: Folge c1 sortierter 4-Tupel

usw.

# Beispiel: Mergesort






# Mischen: sortierte Folgen*)

merge.java

```
public static int[] merge (int[] a, int[] b)   {
   int i = 0, j = 0, k = 0;
   int[] c = new int[a.length + b.length];

   // mischen, bis ein Array leer
   while ((i < a.length) && (j < b.length))   {
        if (a[i] < b[j])
           c[k++] = a[i++];
        else
           c[k++] = b[j++];
   }

   // Rest der nicht-leeren Folge:
   if (i == a.length)
        while (j < b.length)   c[k++] = b[j++];
   else
        while (i < a.length)   c[k++] = a[i++];
   return c;
}
```

Resultattyp: Array

Seiteneffekte !

*) • Folgen als Arrays betrachtet (intern)
→ Files (extern)
• aber: Arrays wie Files verarbeitet (nur sequentiell)



# Ende Teil II:

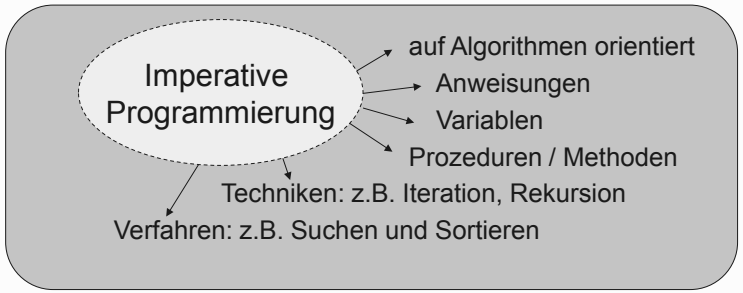
# Konzepte imperativer Sprachen



# Imperative Sprachen

Schwerpunkt:
Entwicklung von Algorithmen

# Konzepte imperativer Sprachen






# Struktur der Vorlesung GdP

Teil I (Grundlagen) : 6 VL
Teil II (Imperative Sprachen) : 8 VL
Teil III (Objektorientierung) : 16 VL



# GdP: angesprochene Bereiche

Teil I
Teil II
Teil III

**Grundlagen**
Grammatiken, Datendarstellung,
von-Neumann-Rechner, ...
Programmiersprachen: Klassifikation ...

**Datenstrukturen und Algorithmen**
Listen, Bäume, Sortieren und Suchen
Komplexität O(n) v. Alg.

**Konzepte von (imperativen) Programmiersprachen**
Variablen: Sichtbarkeit, Lebensdauer
Datentypen (einfach, strukturiert)
Ausdrücke: Prioritäten, cast...
Anweisungen (einfach, strukturiert)
Methoden als Abstraktion
Parameter: value –ref
Rekursion – Iteration
Sprachvergleiche + Kritik

**Softwareentwicklung**
SW-Qualitätsmerkmale
Phasen und Dokumente
Abstraktion + Dekomposition
SW-Architektur: UML
Komponentenarten
Fallbeispiel(e): Labyr,. Eyes

**Konzepte der OO**
ADT - Objekte - Klassen,
Vererbung, Sichtbarkeit,
Klassenvariablen, ~methoden
abstrakte Klassen, Überladung,
Polymorphie, dynamisches Binden,
Ausnahmebehandlung
Ereignisse,
API (ausgewählte Klassen)
Applets, Parallelität (Threads)

**Programmierfertigkeiten**
Sammlung von typischen Programmbeispielen
( Muster + bekannte Beispiele:
Hanoi, Stack, Quicksort, ...)
→ Praktikum, Übungen



# GdP: angesprochene Bereiche

**Grundlagen** Teil I
Grammatiken, Datendarstellung,
von-Neumann-Rechner, ...
Programmiersprachen: Klassifikation ...

Teil II
Teil III

**Datenstrukturen und Algorithmen**
Listen, Bäume, Sortieren und Suchen
Komplexität O(n) v. Alg.

**Konzepte von (imperativen) Programmiersprachen** Teil II
Variablen: Sichtbarke
Datentypen (einfach,
Ausdrücke: Prioritäten, cast...
Anweisungen (einfach, strukturiert)
Methoden als Abstraktion
Parameter: value –ref
Rekursion – Iteration
Sprachvergleiche + Kritik

Teil III
Teil II

**Softwareentwicklung**
Qualitätsmerkmale
n und Dokumente
ktion + Dekomposition
SW-Architektur: UML
Komponentenarten
Fallbeispiel(e): Labyr,. Eyes

**Konzepte der OO** Teil III
ADT - Objekte - Klassen,
Vererbung, Sichtbarkeit,
Klassenvariablen, ~methoden
abstrakte Klassen, Überladung,
Polymorphie, dynamisches Binden,
Ausnahmebehandlung
Ereignisse,
API (ausgewählte Klassen)
Applets, Parallelität (Threads)

Teil III
Teil II

**Programmierfertigkeiten**
Sammlung von typischen Programmbeispielen
+ bekannte Beispiele:
Hanoi, Stack, Quicksort, ...)
→ Praktikum, Übungen

# Fragebögen zu Vorkenntnissen