Universität Bremen

Praktische Informatik 2
Suchen

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

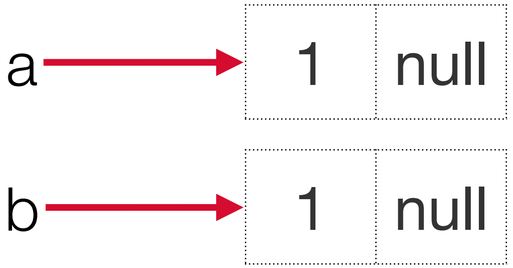
Bild: ©Michael Ihle

Gleichheit

- Identität: Es handelt sich um dasselbe Objekt
  - In Java: **==** für Referenzen

a == b

- Flache Gleichheit: Alle Elemente zweier Objekte sind „flach“ gleich
  - In Java: **==** für alle Attribute der Objekte



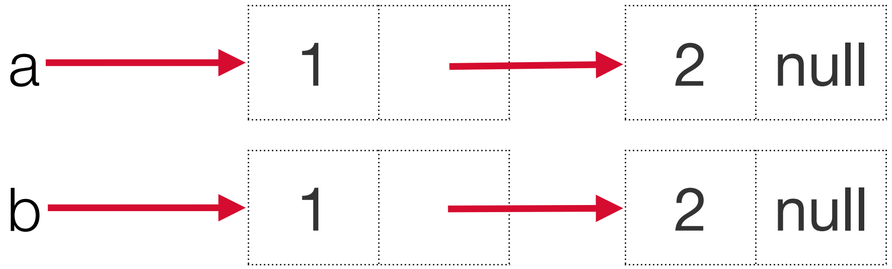
a.shallowEquals(b)

```
class List
{
    int value;
    List next;

    boolean shallowEquals(
            final List other)
    {
        return other != null
                && value == other.value
                && next == other.next;
    }
}
```

Gleichheit: Tiefe Gleichheit

- Alle Elemente zweier Objekte sind „tief“ (rekursiv) gleich
- In Java: **equals()** auf Objekten, wenn **equals()** für alle beteiligten Klassen geeignet definiert ist
- **equals()** in Klasse **Object** entspricht nur **==**, muss also überschrieben werden
- **equals** muss symmetrisch sein: **a.equals(b) == b.equals(a)**
- **java.util.Objects.equals** vergleicht zwei Objekte und kann auch mit **null** umgehen



a.equals(b)

```
class List
{
    int value;
    List next;

    public boolean equals(final Object other)
    {
        return other instanceof List otherList
                && value == otherList.value)
                && (next == otherList.next
                    || next != null
                        && next.equals(otherList.next));
    }
}
```

Für garantierte Kommutativität:
`&& other.getClass() == getClass()`

`&& Objects.equals(next, otherList.next);`

Universität Bremen

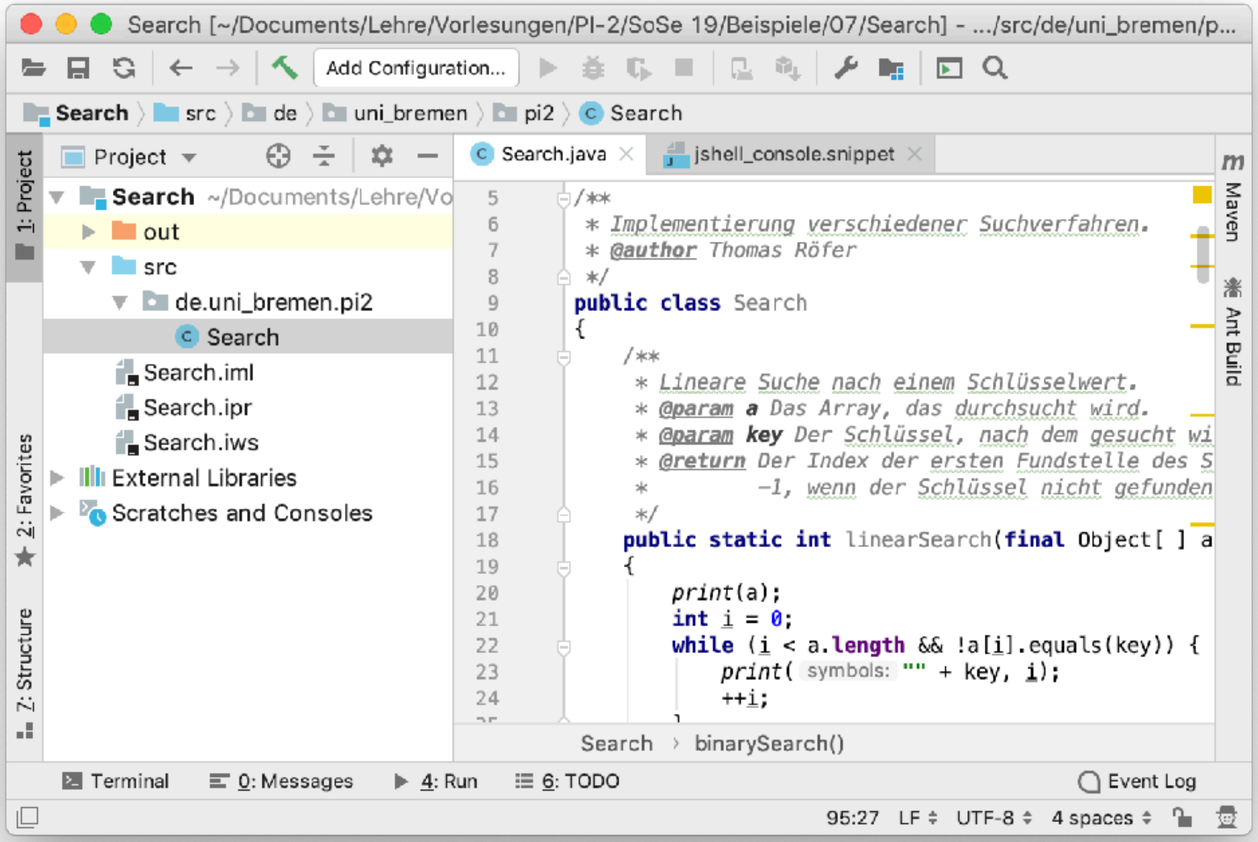
Tiefe Kopie

- Erstellen einer vom Original unabhängigen Kopie
- **clone()** in **Object**: Erzeugt eine **CloneNotSupported-Exception**, es sei denn, eine Klasse implementiert die Schnittstelle **Cloneable**
- Implementierung: **Cloneable** implementieren und **clone()** überschreiben
  - Neues Objekt erstellen
  - Alle Basistypen-Attribute kopieren
  - Für alle Objekt-Attribute wiederum **clone()** aufrufen

```
class List implements Cloneable
{
    int value;
    List next;

    public Object clone()
    {
        final List copy = new List();
        copy.value = value;
        if (next != null) {
            copy.next = (List) next.clone();
        }
        return copy;
    }
}
```

Lineare Suche: Demo



Universität Bremen

Lineare Suche

```
int linearSearch(final Object[ ] a, final Object key)
{
    int i = 0;
    while (i < a.length && !a[i].equals(key)) {
        ++i;
    }
    return i == a.length ? -1 : i;
}
```

- Suche in Array oder Liste
- Gleichheit der Elemente muss definiert sein
- Suchwert (**Schlüssel**) der Reihe nach mit allen Elementen vergleichen
- Komplexität
  - Bester Fall: Erstes Element ist das gesuchte (**O(1)**)
  - Schlechtester Fall: Das gesuchte Element ist nicht vorhanden (**O(N)**)
  - Im Mittel wird halbes Array / halbe Liste durchsucht (**O(½N) = O(N)**)

Universität Bremen

Ordnungsrelationen

- Definieren eine Ordnung zwischen Instanzen eines Typs
- Typische Ordnungsrelationen sind **<** und **>**
- Ordnungsrelationen sind transitiv: **$a > b \land b > c \rightarrow a > c$**
  - Dadurch können Werte auch implizit miteinander verglichen werden
- Eine Ordnung kann über allen Typen definiert werden, ist aber nicht immer intuitiv (Äpfel > Birnen?)

Ordnungsrelationen in Java

- **<**, **>**, **<=**, **>=**: Ordnungen auf primitiven Datentypen außer **boolean**
- **Comparable<T>**: Natürliche Ordnung von Klassen (**compareTo**)
- **Comparator<T>**: Weitere Ordnungen auf Objekten einer Klasse (**compare**)
  - Objekte, die die Ordnung für Objekte anderer Klassen festlegen

```
<T extends Comparable<T>>
        String test(final T a, final T b)
{
   final int c = a.compareTo(b);
   return c < 0 ? "a < b“
         : c == 0 ? "a == b“
         : "a > b";
}
```

```
class ByName implements java.util.Comparator<Student>
{
    public int compare(final Student a, final Student b)
    {
        return a.getName().compareTo(b.getName());
    }
}
```

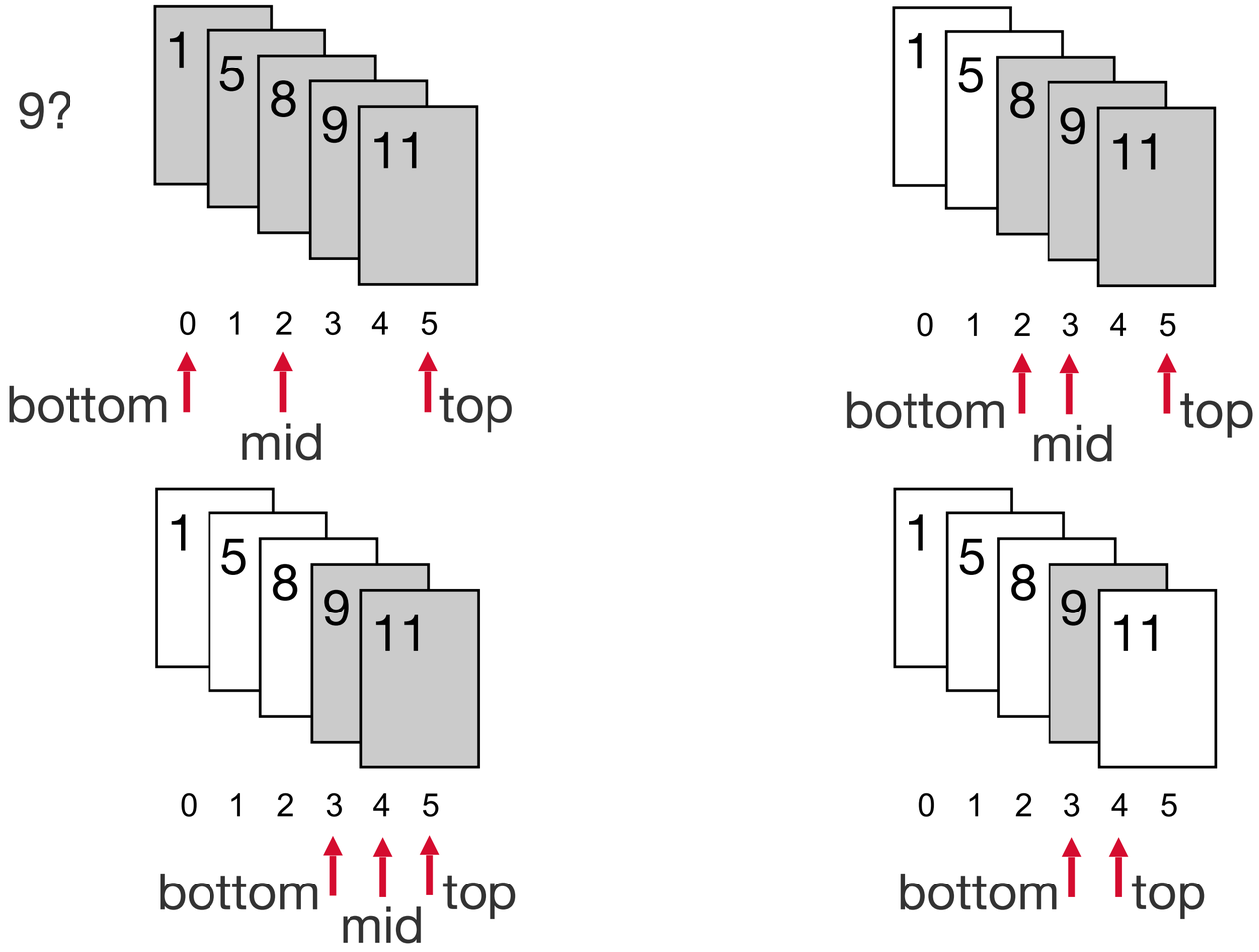
```
(a, b) -> a.getName().compareTo(b.getName());
```

Universität Bremen

Binäre Suche

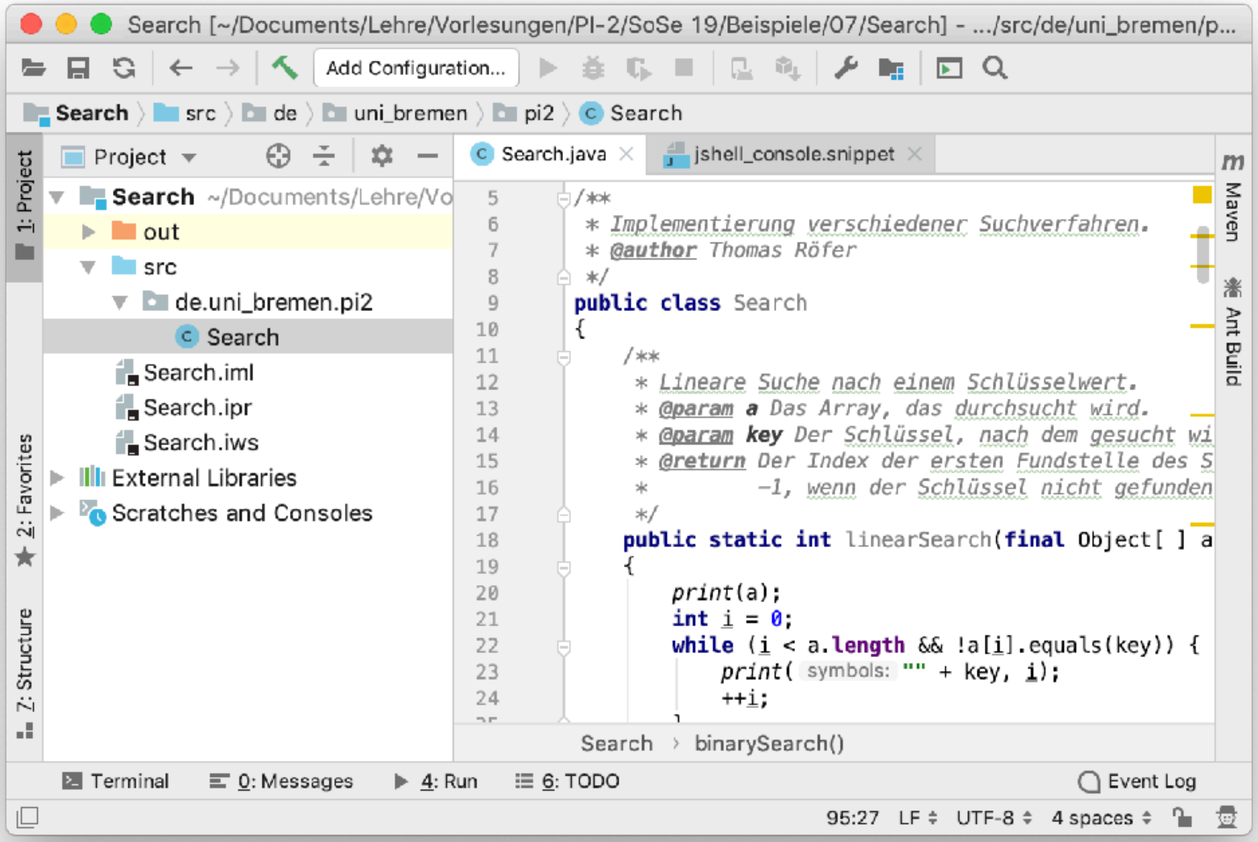
- Suche in Array
- Ordnungsrelation muss für Elemente definiert sein
- Elemente müssen entsprechend der Ordnungsrelation **sortiert** sein
- **Divide and Conquer**: Vergleiche mit mittlerem Element und suche in der richtigen Hälfte weiter
- Durch Sortierung muss nicht mit allen Elementen verglichen werden

Beispiel: Binäre Suche



Universität Bremen

Binäre Suche: Demo



Universität Bremen

Binäre Suche für natürliche Ordnung: Beispiel

(top + bottom) / 2
könnte überlaufen

```
<T extends Comparable<T>>
int binarySearch(final T[ ] a, final T key)
{
    if (a.length == 0) {
        return -1;
    }
    else {
        int bottom = 0;
        int top = a.length;
        while (bottom + 1 != top) {
            final int mid = bottom + (top - bottom) / 2;
            if (a[mid].compareTo(key) > 0) {
                top = mid;
            }
            else {
                bottom = mid;
            }
        }
        return a[bottom].compareTo(key) == 0
                ? bottom : -1;
    }
}
```

Binäre Suche mit Comparator: Beispiel

```
<T> int binarySearch(final T[ ] a,
        final T key, final Comparator<T> c)
{
    if (a.length == 0) {
        return -1;
    }
    else {
        int bottom = 0;
        int top = a.length;
        while (bottom + 1 != top) {
            final int mid = bottom + (top - bottom) / 2;
            if (c.compare(a[mid], key) > 0) {
                top = mid;
            }
            else {
                bottom = mid;
            }
        }
        return c.compare(a[bottom], key) == 0
                ? bottom : -1;
    }
}
```

Universität Bremen

Binäre Suche: Komplexität

- Wie oft kann ein Bereich von **N** Elementen halbiert werden, bis nur noch ein Element übrig bleibt?
  - **$\log_2 N$** mal
- Gesamtaufwand **O(log N)**
  - Eigentlich **O(log N · K)**, wobei **K** der Aufwand für den Schlüsselvergleich ist
- Schlechtester, bester und mittlerer Fall haben alle denselben Aufwand
  - Damit ist die lineare Suche in ihrem besten Fall besser

Universität Bremen

Binäre Suche mit frühem Abbruch

- Bester Fall
  - **O(1)**
- Schlechtester Fall
  - **O(log N)**
- Durchschnittlicher Fall
  - **O(log N)**

```
while (bottom != top) {
    final int mid = bottom + (top - bottom) / 2;
    final int comp = a[mid].compareTo(key);
    if (comp == 0) {
        return mid;
    }
    else if (comp > 0) {
        top = mid;
    }
    else {
        bottom = mid + 1;
    }
}
return -1;
```

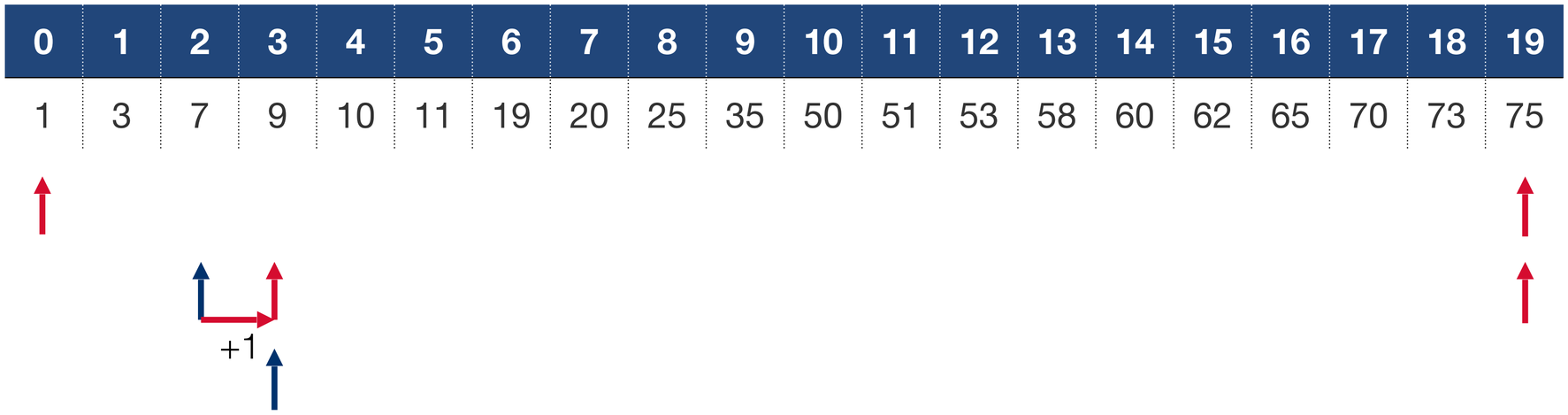
Universität Bremen

Interpolationssuche

- Bei der binären Suche hängt das nächste zu inspizierende Element nur von der Länge des Suchbereichs, aber nicht von den Werten der Schlüssel selbst ab
  - $mid = bottom + \left\lfloor \frac{1}{2}(top - bottom) \right\rfloor$
- Kann ein Abstand zwischen Schlüsseln definiert werden, kann die Position des Suchwerts innerhalb des Arrays abgeschätzt werden
  - $mid = bottom + \left\lfloor \frac{key - a[bottom]}{a[top] - a[bottom]}(top - bottom) \right\rfloor$
- **top** ist inklusive, neue **top** bzw. **bottom** werden auf **mid - 1** bzw. **mid + 1** gesetzt, um Verkleinerung des Suchbereichs zu garantieren

Interpolationssuche: Beispiel

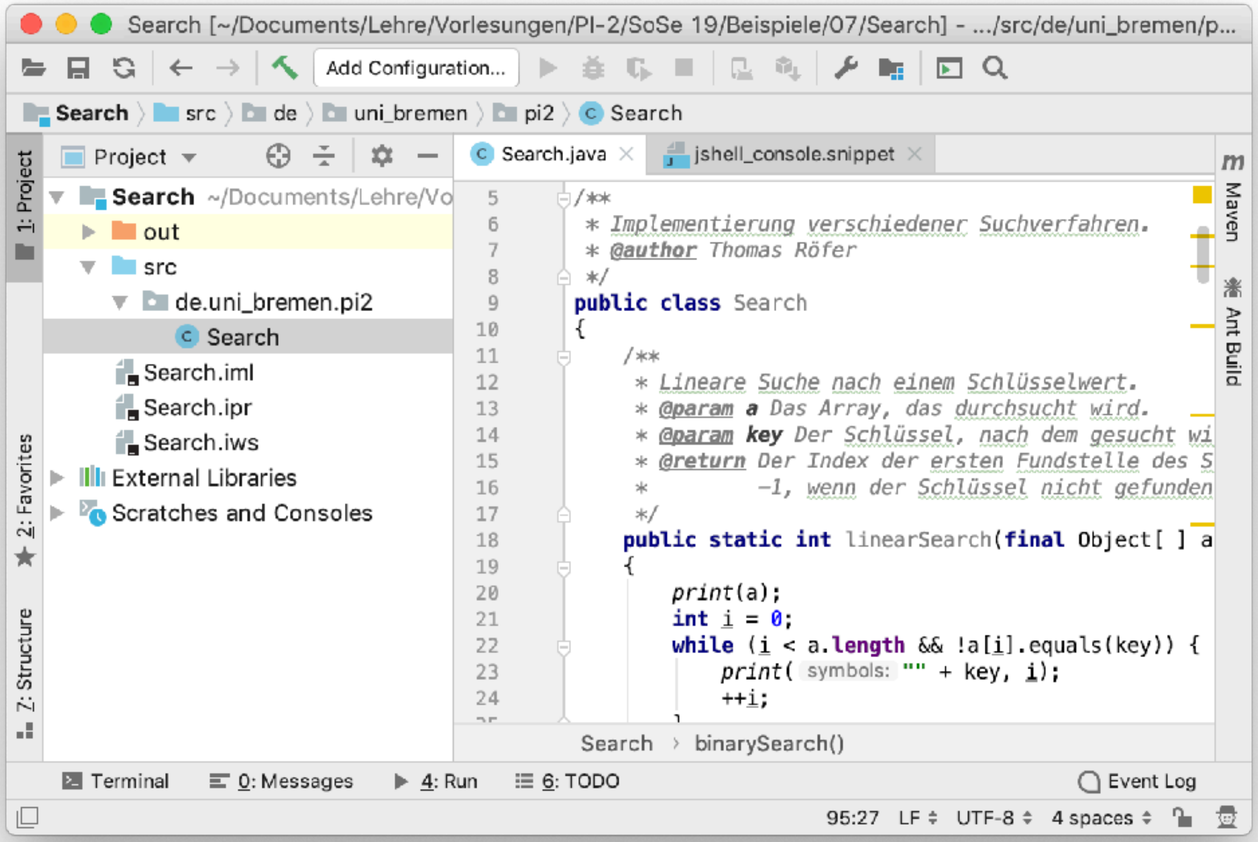
9?



$$mid = bottom + \left\lfloor \frac{key - a[bottom]}{a[top] - a[bottom]} (top - bottom) \right\rfloor$$

Universität Bremen

Interpolationssuche: Demo



Universität Bremen

Ähnlichkeitssuche / Ähnlichkeitsmaße

- Für Zeichenketten kann beim Suchen statt Gleichheit auch Ähnlichkeit getestet werden

- **k-Mismatch**: Weicht Zeichenkette an höchstens **k** Stellen vom gesuchten Wort ab?

- **Editierdistanz**: Wie viele Ersetzen-, Einfüge- und Löschoperationen sind notwendig, um aus gesuchtem Wort die verglichene Zeichenkette zu machen?

- **Phonetische Ähnlichkeit**: „Klingt“ das gesuchte Wort so wie das Vergleichswort? (Sprachabhängig)

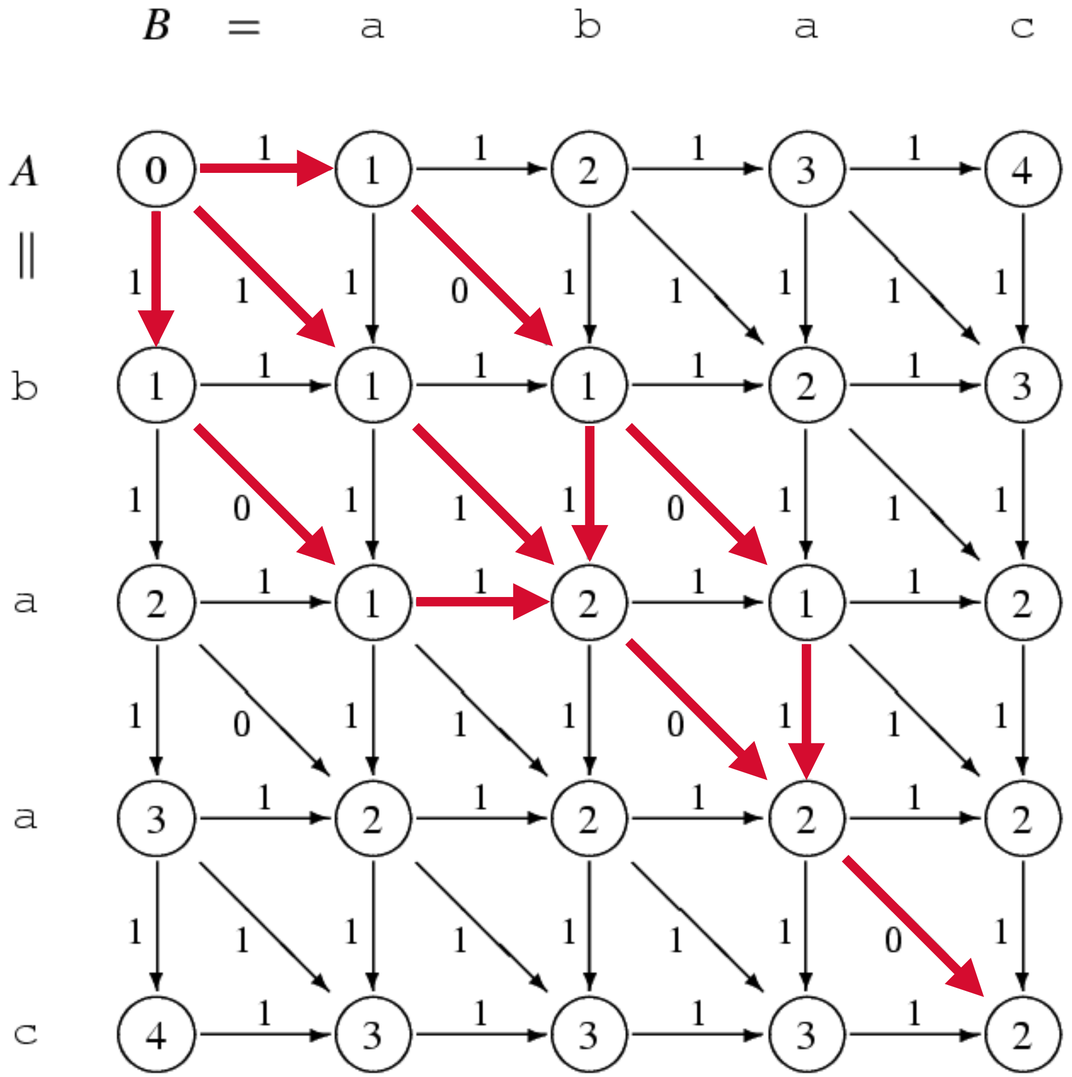
- **Synonymsuche**: Gleicht ein Synonym (Lexikon!) des Suchworts dem Vergleichswort?

Editierdistanz (Levenshtein-Distanz)

- Kleinste Anzahl der folgenden Operationen, die ein Wort in ein anderes wandeln
  - Einfügen eines Zeichens
  - Löschen eines Zeichens
  - Ersetzen eines Zeichens
- Beispiele
  - b a a c
    a a a c 0 ersetzt
    a b a c 1 ersetzt
  - b a a c
    a a c 0 gelöscht
    a b a c 1 eingefügt

Editierdistanz: Spurgraph

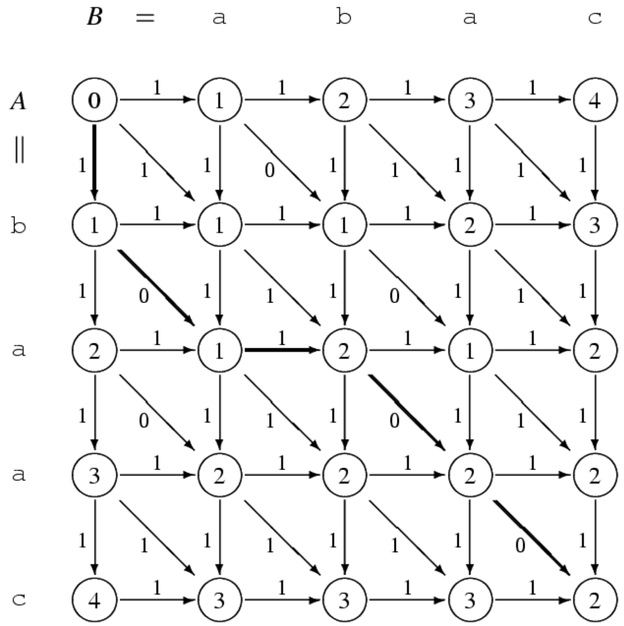
- Startmuster steht links
- Zielmuster steht oben
- Zeichen einfügen: Waagerecht, Kosten 1
- Zeichen löschen: Senkrecht, Kosten 1
- Zeichens ersetzen: Diagonal, 1 wenn Änderung, sonst 0
- Finde Spur mit den niedrigsten Kosten von oben links nach unten rechts



Universität Bremen

Editierdistanz: Beispiel

```
int editDistance(final String src, final String dest)
{
    final int[ ][ ] table = new int[src.length() + 1][dest.length() + 1];
    for (int i = 0; i <= src.length(); ++i) {
        for (int j = 0; j <= dest.length(); ++j) {
            table[i][j] = i == 0 ? j
                        : j == 0 ? i
                        : src.charAt(i - 1) == dest.charAt(j - 1) ? table[i - 1][j - 1]
                        : 1 + Math.min(table[i - 1][j - 1],
                                       Math.min(table[i][j - 1],
                                                table[i - 1][j]));
        }
    }
    return table[src.length()][dest.length()];
}
```



- 1. Zeile: **j**-mal einfügen
- 1. Spalte **i**-mal löschen
- Zeichen beibehalten
- Zeichen ersetzen, löschen oder einfügen (alle Kosten **1**), abhängig davon, welcher Vorgänger die geringsten Kosten hat

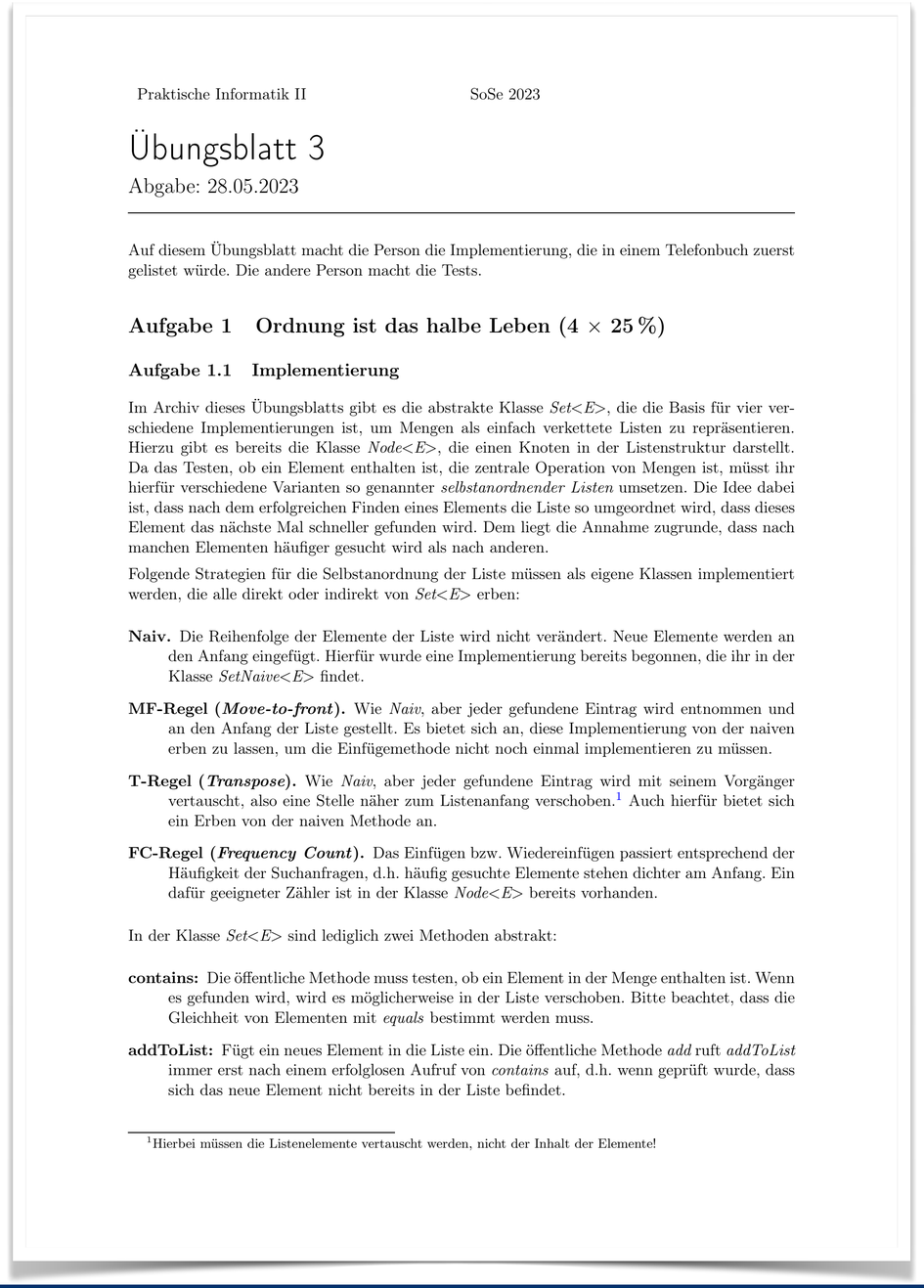
Universität Bremen

Zusammenfassung der Konzepte

- **Identität**, **flache Gleichheit** und **tiefe Gleichheit**
- **Lineare Suche**, **binäre Suche** und **Interpolationssuche**
- **Ordnungsrelationen**
- **Comparable** und **Comparator**
- **Editierdistanz**

Übungsblatt 3

- Selbstanordnende Listen zur Implementierung von Mengen
- Naiv, MF-Regel, T-Regel, FC-Regel
  - Zugriff auf welche(s) Listenelement(e) werden gebraucht, um die Umordnungen durchzuführen?
- Aufgabe 1.1: Implementierung
  - Klassenhierarchie
- Aufgabe 1.2: Tests
  - Analoge Klassenhierarchie



Praktische Informatik II SoSe 2023

Übungsblatt 3

Abgabe: 28.05.2023

Auf diesem Übungsblatt macht die Person die Implementierung, die in einem Telefonbuch zuerst gelistet würde. Die andere Person macht die Tests.

Aufgabe 1 Ordnung ist das halbe Leben (4 × 25 %)

Aufgabe 1.1 Implementierung

Im Archiv dieses Übungsblatts gibt es die abstrakte Klasse *Set<E>*, die die Basis für vier verschiedene Implementierungen ist, um Mengen als einfach verkettete Listen zu repräsentieren. Hierzu gibt es bereits die Klasse *Node<E>*, die einen Knoten in der Listenstruktur darstellt. Da das Testen, ob ein Element enthalten ist, die zentrale Operation von Mengen ist, müsst ihr hierfür verschiedene Varianten so genannter *selbstanordnender Listen* umsetzen. Die Idee dabei ist, dass nach dem erfolgreichen Finden eines Elements die Liste so umgeordnet wird, dass dieses Element das nächste Mal schneller gefunden wird. Dem liegt die Annahme zugrunde, dass nach manchen Elementen häufiger gesucht wird als nach anderen.

Folgende Strategien für die Selbstanordnung der Liste müssen als eigene Klassen implementiert werden, die alle direkt oder indirekt von *Set<E>* erben:

Naiv. Die Reihenfolge der Elemente der Liste wird nicht verändert. Neue Elemente werden an den Anfang eingefügt. Hierfür wurde eine Implementierung bereits begonnen, die ihr in der Klasse *SetNaive<E>* findet.

MF-Regel (*Move-to-front*). Wie *Naiv*, aber jeder gefundene Eintrag wird entnommen und an den Anfang der Liste gestellt. Es bietet sich an, diese Implementierung von der naiven erben zu lassen, um die Einfügemethode nicht noch einmal implementieren zu müssen.

T-Regel (*Transpose*). Wie *Naiv*, aber jeder gefundene Eintrag wird mit seinem Vorgänger vertauscht, also eine Stelle näher zum Listenanfang verschoben.[1] Auch hierfür bietet sich ein Erben von der naiven Methode an.

FC-Regel (*Frequency Count*). Das Einfügen bzw. Wiedereinfügen passiert entsprechend der Häufigkeit der Suchanfragen, d.h. häufig gesuchte Elemente stehen dichter am Anfang. Ein dafür geeigneter Zähler ist in der Klasse *Node<E>* bereits vorhanden.

In der Klasse *Set<E>* sind lediglich zwei Methoden abstrakt:

contains: Die öffentliche Methode muss testen, ob ein Element in der Menge enthalten ist. Wenn es gefunden wird, wird es möglicherweise in der Liste verschoben. Bitte beachtet, dass die Gleichheit von Elementen mit *equals* bestimmt werden muss.

addToList: Fügt ein neues Element in die Liste ein. Die öffentliche Methode *add* ruft *addToList* immer erst nach einem erfolglosen Aufruf von *contains* auf, d.h. wenn geprüft wurde, dass sich das neue Element nicht bereits in der Liste befindet.

[1] Hierbei müssen die Listenelemente vertauscht werden, nicht der Inhalt der Elemente!