1. Suchen - Einführung

- Dictionaries
- Elementare Suchverfahren
  - Sequentielle Suche
  - Binäre Suche
  - Umsetzung in Java

Dictionaries

Dictionary

Datenstruktur zur effizienten Verwaltung einer (dynamischen) Menge von Datensätzen.

Datensätze (records)

bestehen aus

- Suchschlüssel (search key) und
- Nutzdaten (value).

Beispiel Telefonbuch:

Maier	Hauptstr. 1	14234
Baier	Bahnhofstr. 5	24829
Müller	Uferweg 5	53289
Anton	Mozartstr. 2	36478

Datensatz

Suchschlüssel

Nutzdaten: Adresse + Tel.Nr

Beispiel Wörterbuch:

sehen	see; look
sprechen	speak; talk
gehen	go; walk
hören	hear; listen

Operationen für Dictionaries

- search(key)
  suche nach Datensatz mit Schlüssel key.
- insert(key, value)
  füge neuen Datensatz mit Schlüssel key und Daten value ein.
- remove(key)
  Lösche Datensatz mit Schlüssel key.
- Traversieren (Iterator):
  Gehe über alle Datensätze in bestimmter Reihenfolge:
  - über Schlüssel sortiert
  - nach Einfügezeitpunkt
  - irgendeine Reihenfolge (von der Implementierung bestimmt)

Datenstrukturen mit diesen Operationen werden in der Literatur und Programmiersprachen auch als Dictionaries oder Maps bezeichnet.

Varianten

Eindeutigkeit der Schlüssel:

- Es können auch mehrere Datensätze zu einem Schlüssel existieren.
- Die Forderung der Eindeutigkeit der Schlüssel hängt von der Anwendung ab.
  Beispiele:
  - Telefonbuch: Schlüssel (= Familiennamen) sind nicht eindeutig
  - Wörterbuch: Schlüssel sind eindeutig

Schlüsseltypen:

- int
- String
- zusammengesetzte Daten; z.B. Uhrzeit, Kalenderdatum, ...

Dictionaries ohne Nutzdaten:

- entsprechen Mengen von Schlüssel und werden auch Sets genannt.

In dieser Vorlesung:

- Schlüssel sind eindeutig.
- Im wesentlichen nur int-Werte als Schlüssel.
  Nutzdaten in den Darstellungen meist weggelassen.

Dictionaries in Programmbibliotheken

Üblicherweise als generische Klasse:

- Schlüsseltyp und Typ für Nutzdaten werden parameterisiert
- Z.B. K für Key und V für Value.

Java:

- Interface Set<K> mit den Implementierungen
  - TreeSet<K> (ausgeglichener binärer Suchbaum)
  - und HashSet<K> (Hashverfahren)
- Interface Map<K, V> mit den Implementierungen TreeMap<K, V> und HashMap<K, V>
- Bei TreeSet<K> und TreeMap<K, V> muss der Schlüsseltyp K den Vergleichsoperator compareTo (siehe Interface Comparable) anbieten.
- Dictionary<K, V> ist veraltet; stattdessen Map<K, V> verwenden.

STL in C++:

- Template-Klassen set<K> und map<K, V>
- Template-Klassen multiset<K> und multimap<K, V>, falls Schlüssel mehrfach vorkommen dürfen.

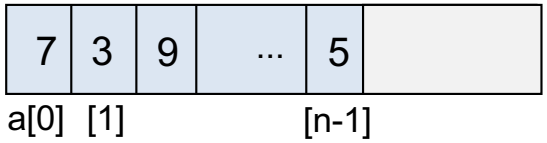
1. Suchen - Einführung

- Dictionaries
- Elementare Suchverfahren
  - Sequentielle Suche
  - Binäre Suche
  - Umsetzung in Java

Sequentielle Suche (1)

Datenstruktur

- Halte n Datensätzen in einem Feld lückenlos und unsortiert.
- Alternative: linear verkettete Liste.



Algorithmen (in Pseudocode)

```
search(int k) {
        for (int i = 0; i < n; i++)
                if (a[i].key == k)
                        return "k gefunden";
        return "k nicht gefunden";
}
```

```
remove(int k) {
        if (k kommt in a vor) {
                lösche k und schließe Lücke;
                n--;
        }
}
```

```
insert(int k) {
        if (k kommt nicht in a vor) {
                if (a vollständig gefüllt)
                        vergrößere Feld a;
                a[n++] = k;
        }
}
```

Sequentielle Suche (2)

Laufzeiten (Worst Case):

Operation bei einer Menge mit n Elementen	**T(n)**
search	O(n)
insert	O(n) *)
remove	O(n)
Aufbau einer Menge mit n Elementen (d.h. n insert-Aufrufe)	$O(n^2)$
n search-Aufrufe	$O(n^2)$

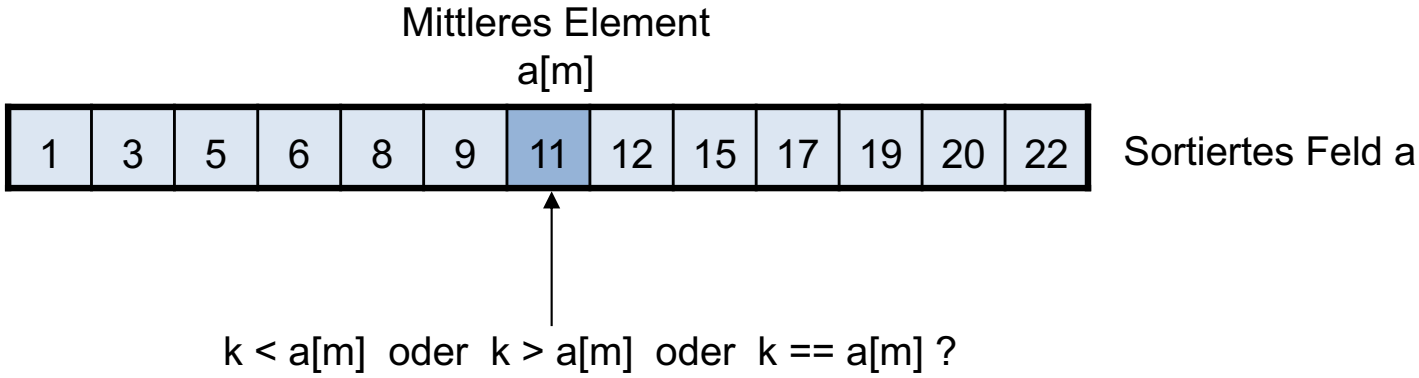
*) Es muss geprüft werden, ob das Element bereits vorkommt (search-Aufruf)

Binäre Suche (1)

Datenstruktur

- Halte Datensätze lückenlos in einem sortierten Feld.

Idee für Such-Algorithmus



Falls k == a[m], dann gefunden.

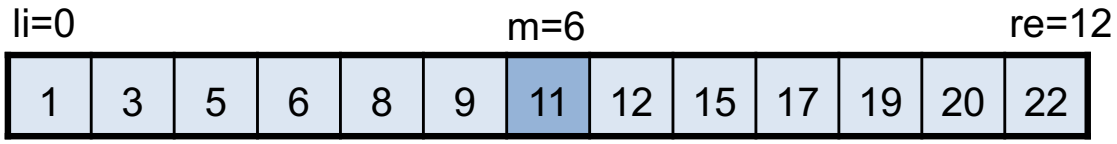
Falls k < a[m], dann suche in linker Hälfte weiter.

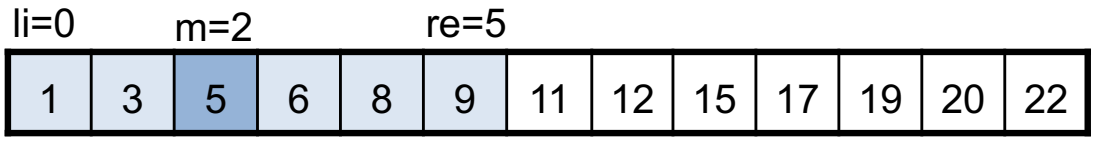
Falls k > a[m], dann suche in rechter Hälfte weiter

Binäre Suche (2)

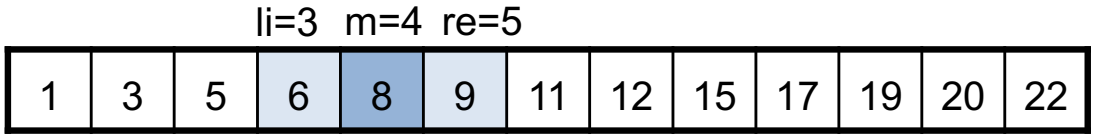
Beispiel:

Suche nach k = 8:





Suche in linker Hälfte



Suche in rechter Hälfte;
k wird gefunden!

Zu durchsuchender Bereich geht von a[li] bis a[re]

Mittleres Element m = (li + re)/2 (ganzzahlige Division)

Binäre Suche (3)

Algorithmen:

```
search(int k) {
        int li = 0;
        int re = n-1;

        while (re >= li) {
                int m = (li + re)/2;
                if (k == a[m].key)
                        return "k gefunden";
                else if (k < a[m].key)
                        re = m – 1;
                else
                        li = m + 1;
        }

        return "k nicht gefunden";
}
```

Suche weiter in linker Hälfte

Suche weiter in rechter Hälfte

Laufzeit von search:

$T(n) = O(\log n)$

In jedem Schleifendurchlauf wird das zu durchsuchende Teilfeld in etwa halbiert.

```
insert(int k)
// füge in sortiertes Feld ein
```

```
remove(int k)
// wie zuvor
```

Binäre Suche (4)

Laufzeiten (Worst Case):

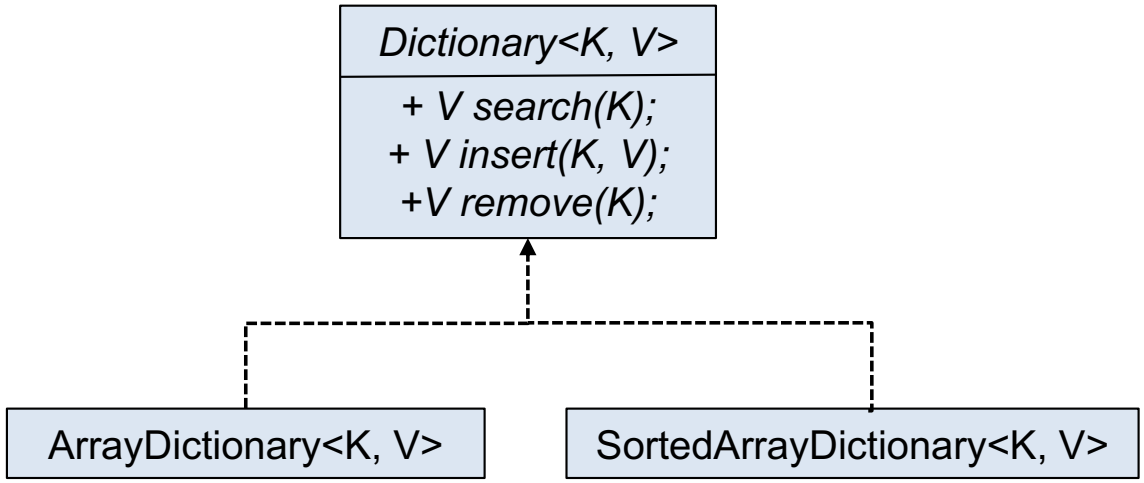
Operation bei einer Menge mit n Elementen	**T(n)**
search	O(log n)
insert	O(n)
remove	O(n)
Aufbau einer Menge mit n Elementen (d.h. n insert-Aufrufe)	$O(n^2)$ *)
n search-Aufrufe	O(n log n)

*) Alternativ lassen sich auch n Elemente unsortiert einfügen und dann mit einem schnellen Sortierverfahren sortieren. Damit würde man T(n) = O(n log n) erreichen

Umsetzung in Java

Konzept:

- Definiere Dictionary<K, V> als Interface
- Implementiere Klassen:
  - ArrayDictionary<K, V> mit sequentieller Suche
  - SortedArrayDictionaryK, V> mit binärer Suche



Interface Dictionary

```
public interface Dictionary<K,V> {

    V insert(K key, V value);
    // Falls ein Datensatz mit Schlüssel key noch nicht existiert, wird das
    // Schlüssel-Wert-Paar key, value eingefügt und null zurückgeliefert.
    // Ansonsten werden beim Datensatz mit Schlüssel key die alten
    // Nutzdaten durch value ausgetauscht und zurückgeliefert.

    V search(K key);
    // Liefert die Daten zu dem Schlüssel key zurück oder
    // null falls ein Datensatz mit Schlüssel key nicht vorhanden ist.

    V remove(K key);
    // Löscht den Datensatz mit Schlüssel key (falls vorhanden) und liefert die alten
    // Daten zurück. Falls der Datensatz nicht existiert, wird null zurückgeliefert.
}
```

ArrayDictionary (1)

```
public class ArrayDictionary<K, V> implements Dictionary<K, V> {

   static private class Entry<K,V> {
      K key;
      V value;
      Entry(K k, V v) {key = k; value = v;}
   };
   private static final int DEF_CAPACITY = 16;
   private int size;
   private Entry<K,V>[] data;

   @SuppressWarnings("unchecked")
   public ArrayDictionary() {
      size = 0;
      data = new Entry[DEF_CAPACITY];
   }
```

Feld data mit Schlüssel-Wert-Paaren

Raw type!

ArrayDictionary (2)

```
private int searchKey(K key) {
    for (int i = 0; i < size; i++) {
        if (data[i].key.equals(key)) {
            return i;
        }
    }
    return -1;  // nicht gefunden
}

public V search(K key) {
    int i = searchKey(key);
    if (i >= 0)
        return data[i].value;
    else
        return null;
}
```

data[i].key.equals(key) statt data[i].key == key, denn sonst würde Vergleich nur auf Referenzen stattfinden.

Jedoch muss equals für den Schlüsseltyp geeignet definiert sein.

ArrayDictionary (3)

```
public V insert(K key, V value) {
   int i = searchKey(key);

   // Vorhandener Eintrag wird überschrieben:
   if (i >= 0) {
      V r = data[i].value;
      data[i].value = value;
      return r;
   }

   // Neueintrag:
   if (data.length == size) {
      data = Arrays.copyOf(data, 2*size);
   }
   data[size] = new Entry<K,V>(key,value);
   size++;
   return null;
}
```

Feld data vergrößern.

ArrayDictionary (4)

```
    public V remove(K key) {
        int i = searchKey(key);
        if (i == -1)
            return null;

        // Datensatz loeschen und Lücke schließen
        V r = data[i].value;
        for (int j = i; j < size-1; j++)
            data[j] = data[j+1];
        data[--size] = null;
        return r;
    }
}
```

SortedArrayDictionary mit Typbeschränkung (1)

```
public class SortedArrayDictionary<K extends Comparable<? super K>, V>
implements Dictionary<K, V> {

   // ...

}
```

- Typbeschränkung:
  Schlüssel von Typ K müssen vergleichbar sein.
- Dazu genügt es zu fordern, dass die Elemente von irgendeinem Obertyp von K vergleichbar sind.

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

x.compareTo(y) liefert eine negative Zahl, 0, bzw. eine positive Zahl zurück, falls x kleiner, gleich bzw. größer als y ist.

SortedArrayDictionary mit Typbeschränkung (2)

```
public class SortedArrayDictionary<K extends Comparable<? super K>, V>
implements Dictionary<K, V> {


   // ...


   private int searchKey(K key) {
        int li = 0;
        int re = size - 1;

        while (re >= li) {
           int m = (li + re)/2;
           if (key.compareTo(data[m].key) < 0)
              re = m - 1;
           else if (key.compareTo(data[m].key) > 0)
              li= m + 1;
           else
              return m; // key gefunden
        }
        return -1; // key nicht gefunden
   }
```

Daten, Konstruktor, ensureCapacity, search und remove wie bei ArrayDictionary

Binäre Suche

SortedArrayDictionary mit Typbeschränkung (3)

```
public V insert(K key, V value) {
     int i = searchKey(key);

     // Vorhandener Eintrag wird überschrieben:
     if  (i != -1) {
         V r = data[i].value;
         data[i].value = value;
         return r;
     }

     // Neueintrag:
     if (data.length == size) {
          data = Arrays.copyOf(data, 2*size);
     }
     int j = size-1;
     while (j >= 0 && key.compareTo(data[j].key) < 0) {
         data[j+1] = data[j];
         j--;
     }
     data[j+1] = new Entry<K,V>(key,value);
     size++;
     return null;
 }
```

Einfuegen in sortiertes Feld.

Anwendung

```
public class WoerterbuchAnwendung {

    public static void main(String[] args) {
        Dictionary<String, String> wb;

        Scanner in = new Scanner(System.in);
        int i;
        i = in.nextInt();

        if (i == 1)
            wb = new ArrayDictionary<String,String>();
        else
            wb = new SortedArrayDictionary<String, String>();

        wb.insert("lesen", "read");
        wb.insert("schreiben", "write");

        System.out.println(wb.search("lesen"));
    }
```

- Beachte, dass für den Schlüsseltyp String sowohl equals als auch compareTo geeignet definiert ist.

Beispiel: Datum als Schlüssel (1)

- Wie muss die Klasse Datum erweitert werden, so dass Datum als Schlüsseltyp sowohl für ArrayDictionary als auch SortedArrayDictionary eingesetzt werden kann?

```
public class Datum {
    private int tag = 1;
    private int mon = 1;
    private int jahr = 1970;

    public Datum() {}
    public Datum(int t, int m, int j) {
        tag = t; mon = m; jahr = j;
    }
}
```

Beispiel: Datum als Schlüssel (2)

```
public class Datum implements Comparable<Datum> {
    private int tag = 1;
    private int mon = 1;
    private int jahr = 1970;

    // ...

    public boolean equals(Object o) {
        if (this == o) return true;
        if (! (o instanceof Datum))) return false;
        Datum d = (Datum) o;
        return (jahr == d.jahr && mon == d.mon && tag == d.tag);
    }

    public int compareTo(Datum d) {
        if (jahr < d.jahr) return -1;
        else if (jahr > d.jahr) return 1;
        else if (mon < d.mon) return -1;
        else if (mon > d.mon) return 1;
        else if (tag < d.tag) return -1;
        else if (tag > d.tag) return 1;
        else return 0;
    }
}
```

d1.compareTo(d2) liefert
- -1, falls d1 chronologisch vor d2
- 0, falls d1 gleich d2
- +1, falls d1 chronologisch nach d2

SortedArrayDictionary mit Comparable-Downcast

- SortedArrayDictionary kann auch ohne Typbeschränkung definiert werden.
  Eine compareTo-Methode wird dann durch ein Comparable-Downcast "erzwungen".
- Laufzeitfehler (ClassCastException), falls Schlüsseltyp K nicht Comparable ist.
- Die Java-Klassen TreeSet und TreeMap setzen diese Technik ein.

```
public class SortedArrayDictionary<K, V> implements Dictionary<K, V> {

   // ...

   private int searchKey(K key) {
       Comparable<? super K> comparableKey
            = (Comparable<? super K>) key;

       ...

       if (comparableKey.compareTo(data[m].key) < 0)
       ...
   }

   // ...
}
```

- Es genügt zu fordern, dass die Elemente von irgendeinem Obertyp von K vergleichbar sind.
- ClassCastException, falls Cast nicht möglich.

SortedArrayDictionary mit Comparator-Parameter (1)

```
public class SortedArrayDictionary<K, V> implements Dictionary<K, V> {

   private Comparator<? super K> cmp;
   // ...

   public SortedArrayDictionary(Comparator<? super K> c) {
     if (c == null)
         cmp = ...; // Natural Order als Default-Wert (später)
     else
         cmp = c;
     // ...
   }

   private int searchKey(K key) {
      ...
      if (cmp.compare(key,data[m].key)) < 0)
      ...
   }

   // ...
}
```

Für den übergebenen Comparator-Parameter c genügt es zu fordern, dass c die Elemente von irgendeinem Obertyp von K vergleichen kann.

```
public interface Comparator<T> {
    int compare(T x, T y);
}
```

compare(x, y) liefert eine negative Zahl, 0, bzw. eine positive Zahl zurück, falls x kleiner, gleich bzw. größer als y ist.

SortedArrayDictionary mit Comparator-Parameter (2)

- Comparator ist ein funktionales Interface. Daher kann einfachheitshalber ein Lambda-Ausdruck übergeben werden.

- Kalenderdaten chronologisch aufsteigend sortiert:

```
Dictionary<Datum,String> kalender =
        new SortedArrayDictionary<>( (d1, d2) -> d1.compareTo(d2) );
```

Lambda-Ausdruck.

- Kalenderdaten chronologisch absteigend sortiert:

```
Dictionary<Datum,String> kalender =
        new SortedArrayDictionary<>( (d1, d2) -> d2.compareTo(d1) );
```

SortedArrayDictionary mit Comparator-Parameter mit Default-Wert

```
public class SortedArrayDictionary<K, V> implements Dictionary<K, V> {

   private Comparator<? super K> cmp;
   // ...


   public SortedArrayDictionary(Comparator<? super K> c) {
      if (c == null)
          // Natural Order als Default-Wert:
          cmp = (x,y) -> ((Comparable<? super K>) x).compareTo(y);
      else
          cmp = c;
      // ...
   }


   public SortedArrayDictionary() {
      // Natural Order als Default-Wert:
      cmp = (x,y) -> ((Comparable<? super K>) x).compareTo(y);
   }


   // ...
}
```

Konstruktor mit Comparator-Parameter

Default-Konstruktor

Vorsicht!!!

- Bei der Definition von Comparable wird eine sogenannte natürliche Ordnung (natural order) $\leq$ definiert:
  - $x \leq y$ gdw. x.compareTo(y) <= 0
- Es muss darauf geachtet werden, dass die natürliche Ordnung eine lineare Ordnung ist:
  - $x \leq x$ (Reflexivität)
  - $x \leq y \wedge y \leq x \Rightarrow x = y$ (Antisymmetrie)
  - $x \leq y \wedge y \leq z \Rightarrow x \leq z$ (Transitivität)
  - $x \leq y \vee y \leq x$ (Totalität)
- Analog muss darauf geachtet dass bei der Implementierung von Comparator, die compare-Methode ebenfalls eine totale Ordnung ergibt.
- Java-API zu Comparable und Comparator beachten!

Interface Comparable<T> in der Java API

interface Comparable<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

It is strongly recommended (though not required) that natural orderings be consistent with equals.

int compareTo(T o)

(1) Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

(2) The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y.

(3) The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

(4) Finally, the implementor must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

(5) It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Aus der Definiton von compareTo ergibt sich die natürliche Ordnung:

$x \leq y$ gdw. x.compareTo(y) <= 0

(1) – (4) erzwingen, dass die natürliche Ordnung auch eine totale Ordnung ist.

Interface Comparator<T> in der Java API

interface Comparator<T>

A comparison function, which imposes a total ordering on some collection of objects. Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering.

The ordering imposed by a comparator c on a set of elements S is said to be consistent with equals if and only if c.compare(e1, e2)==0 has the same boolean value as e1.equals(e2) for every e1 and e2 in S.

Eine Relation ≤ ist eine totale Ordnung (lineare Ordnung), falls ≤ reflexiv, transitiv, antisymmetrisch und total ist.

int compare(T o1, T o2)

(1) Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

(2) The implementor must ensure that sgn(compare(x, y)) == -sgn(compare(y, x)) for all x and y.

(3) The implementor must also ensure that the relation is transitive: ((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

(4) Finally, the implementor must ensure that compare(x, y)==0 implies that sgn(compare(x, z))==sgn(compare(y, z)) for all z.

(5) It is generally the case, but not strictly required that (compare(x, y)==0) == (x.equals(y)). Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Aus der Definiton von compare ergibt sich eine ≤-Relation:

x ≤ y gdw. compare(x,y) <= 0

(1) – (4) erzwingen, dass die ≤-Relation auch eine totale Ordnung ist.

Vorsicht: Klasse Point verletzt Vertragsbedingung!

```
public class Point implements Comparable<Point> {
    private double x;
    private double y;

    public Point(double x, double y) {...}
    public boolean equals(Object o) {...}

    public int compareTo(Point p) {
        if (this.x < p.x && this.y < p.y)
            return -1;
        else if (this.x == p.x && this.y == p.y)
            return 0;
        else
            return 1;
    }
}
```

- Wodurch wird der Vertrag von Comparable<Point> verletzt?
- Wieso kann sich bei binärer Suche ein Problem ergeben?
- Wie könnte eine vertragsgemäße Definition von compareTo aussehen?

Wichtiger Merksatz

„Es gibt nichts Praktischeres als eine gute Theorie“

Kurt Lewin, 1890 – 1947, deutscher Sozialpsychologe