Spiegeln einer Liste mit Keller

```
    public DVList<T> spiegeln() {
    Stapel<T> stack = new Stapel<T>();
    int max = this.size();
    for(int i=1; i<=max; i++) {
       stack.push(this.proj(i));
    }

    DVList<T> erg = new DVList<T>();
    for(int i=1; i<=max; i++) {
       erg.postfix(stack.top());
       stack.pop();
    }
    return erg;
}
```

Lokale Klassen

- Eigentlich muss man die Klasse `Entry<T>` gar nicht unbedingt kennen, wenn man einen Stack oder eine Queue (oder auch eine Liste) benutzt.
- In Java gibt es die Möglichkeit, innerhalb einer Klassendefinition weitere Klassen zu definieren, sog. *lokale* oder *eingebettete* Klassen.
- Die lokalen Klassen haben Zugriff auf alle Komponenten (auch der **`private`** Komponenten) der Klasse, in die sie eingebettet sind.

Keller mit lokaler Entry-Klasse

```
public class Keller<T> {

private Entry first;

// Eingebettete Klasse:

private class Entry {
   Entry next;
   T element;

   Entry(T element) {
      this.next = null;
      this.element = element;
   }
}

// Die eigentlichen Methoden von Keller (naechste Seite):
...
```

Keller mit lokaler Entry-Klasse

```
    ...
    public Keller() {
        this.first = null;
    }

    public boolean isEmpty() {
        return (this.first == null);
    }

    public void push(T element) {
        Entry newEntry = new Entry(element);
        newEntry.next = this.first;
        this.first = newEntry;
    }

    public T pop() {
        T erg = this.first.element;
        this.first = this.first.next;
        return erg;
    }
}
```

Lokale Klassen

- Die Schnittstelle der Klasse `Keller` besteht nur aus folgenden Methoden (zzgl. des Konstruktors):

`boolean isEmpty()`	Ist der Keller leer?
`void push(T element)`	Legt ein Element auf dem Keller ab.
`T pop()`	Entfernt das oberste Objekt vom Keller und gibt es zurück.

- Die Klasse `Entry` ist nach außen hin nicht sichtbar.

Schlange mit lokaler Entry-Klasse

- Und dasselbe auch mit der Schlange:

```
public class Schlange<T> {

private Entry first;

private Entry last;

// Eingebettete Klasse:

private class Entry {
   Entry next;
   T element;
}

// Die eigentlichen Methoden von Schlange (naechste Seite):
...
```

- Aufgemerkt: Der Konstruktor von `Entry` fehlt hier.

Schlange mit lokaler Entry-Klasse

```
    ...
    // Auch hier sparen wir uns den Konstruktor.

    public boolean isEmpty() {
        return (this.first == null);
    }

    public void put(T item) {
        Entry newEntry = new Entry();
        newEntry.element = item;
        if(this.first == null) {
            this.first = newEntry;
            this.last = newEntry;
        } else {
        this.last.next = newEntry;
        this.last = newEntry;
    }

    public T get() {
        T erg = this.first.element;
        this.first = this.first.next;
        if(this.isEmpty()) {
            this.last = null;
        }
        return erg;
    }
}
```

Diskussion

- Vorteil der lokalen Klassen ist, dass sie von außen nicht sichtbar sind.
- Das macht Sinn: der Benutzer der Datenstrukturen muss deren Aufbau nicht verstehen, die Schnittstellen der Listen sind klar definiert und verwenden keine Entry-Objekte.
- Nachteil: keine Wiederverwendbarkeit der Klasse `Entry<T>`.
- Die beiden Entry-Klassen sind zwar unterschiedlich, aber z.B. könnte die Entry-Klasse für Schlangen von der Entry-Klasse des Kellers abgeleitet werden, etc.

Noch ein paar Tests der beiden Klassen

```
public class Test {

   public static void main(String[] args) {
                Schlange<Integer> queue = new Schlange<Integer>();
                Keller<Integer> stack = new Keller<Integer>();

                // Test von Schlange und Keller
                queue.put(1);
                stack.push(1);
                queue.put(2);
                stack.push(2);
                queue.put(3);
                stack.push(3);

                ausgabe(queue, stack); // Dannach sind beide leer!!!

                // Test von spiegeln
                queue.put(1);
                queue.put(2);
                queue.put(3);

                spiegeln(queue);

                System.out.println("Inhalt Queue:");
                while(!queue.isEmpty()) {
                        System.out.print(queue.get().toString());
                }
                System.out.println();
```

Noch ein paar Tests der beiden Klassen

```
...

public static void ausgabe(Schlange<Integer> queue, Keller<Integer> stack) {
    System.out.println("Inhalt Queue:");
    while(!queue.isEmpty()) {
        System.out.print(queue.get().toString());
    }
    System.out.println();

    System.out.println("Inhalt Stack:");
    while(!stack.isEmpty()) {
        System.out.print(stack.pop().toString());
    }
    System.out.println();
}

...
```

Noch ein paar Tests der beiden Klassen

Ausgabe nach Aufruf von `ausgabe(queue, stack)` in `main`:

```
Inhalt Queue:
123
Inhalt Stack:
321
```

Danach sind beide Listen leer!!!

Noch ein paar Tests der beiden Klassen

```
    ...

    public static void spiegeln(Schlange<Integer> q) {
        Keller<Integer> s = new Keller<Integer>();
        while(!q.isEmpty()) {
        Integer top = q.get();
        System.out.println("Lege "+top.toString()+" auf den Stack!");
        s.push(top);
    }

    while(!s.isEmpty()) {
        Integer top = s.pop();
        System.out.println("Lege "+top.toString()+" zurueck!");
        q.put(top);
    }
}
```

Noch ein paar Tests der beiden Klassen

Ausgabe nach Aufruf von `spiegeln(queue)` in `main`:

```
Lege 1 auf den Stack!
Lege 2 auf den Stack!
Lege 3 auf den Stack!
Lege 3 zurueck!
Lege 2 zurueck!
Lege 1 zurueck!
Inhalt Queue:
321
```

Überblick

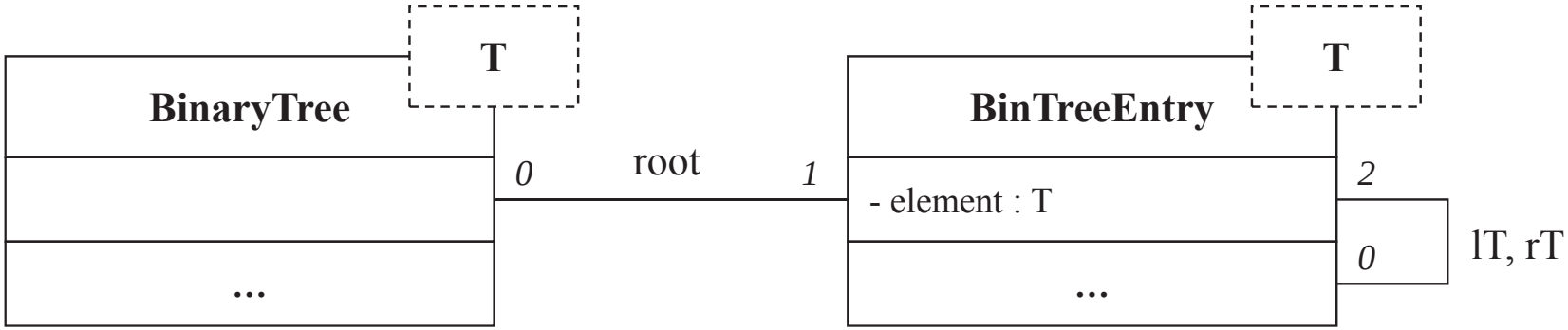
10. Datenstrukturen
10.1 Einleitung
10.2 Ein Datenmodell für Listen
10.3 Doppelt-verkettete Listen
10.4 Bäume
10.5 Mengen
10.6 Exkurs: Das Collections-Framework in Java

Aufbau

- Bäume organisieren Einträge (im folgenden *Knoten*) nicht mehr linear, sondern *hierarchisch*.
- Induktive Definition eines *binären Baums* über eine Knotenmenge K (siehe Kap. 3):
  - der leere Baum ε ist ein binärer Baum
  - sind lT und rT binäre Bäume und $k \in K$ ein Knoten (Eintrag), so ist (k, lT, rT) ebenfalls ein binärer Baum.
- Verallgemeinerung: m-äre Bäume: statt 2 Teilbäumen hat jeder Knoten m Teilbäume.
- Beispiel für einen Binärbaum:

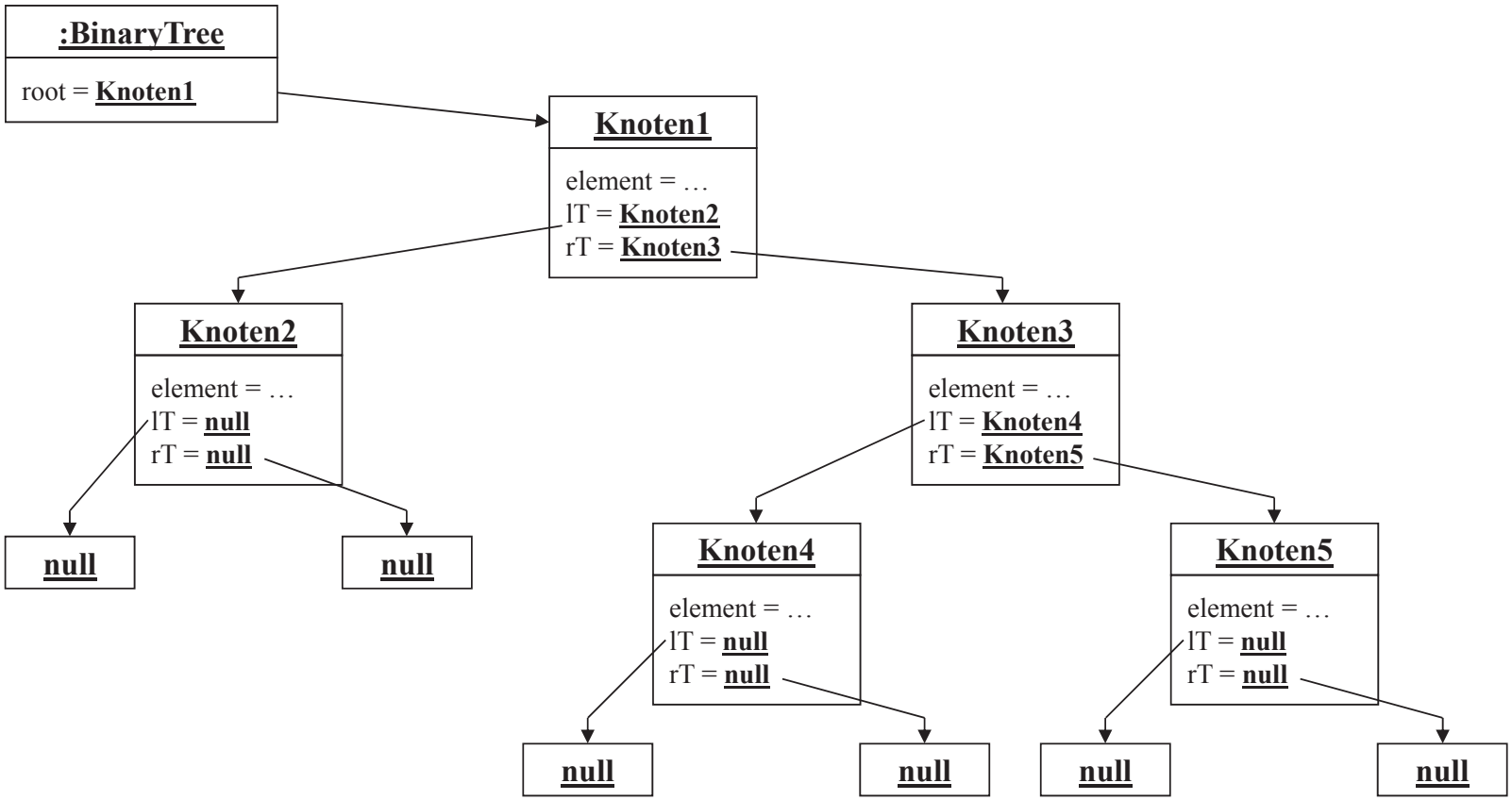
Realisierung

- lT und rT werden auch linker bzw. rechter *Teilbaum* genannt.
- Jeder Knoten ist der *Vaterknoten* (*Wurzel*) seiner Teilbäume.
- Knoten, deren linker und rechter Teilbaum leer sind, heißen *Blätter*.
- Der Vaterknoten des gesamten Baumes ist die Wurzel des Baumes.
- Bäume werden ähnlich wie Listen verkettet gespeichert.

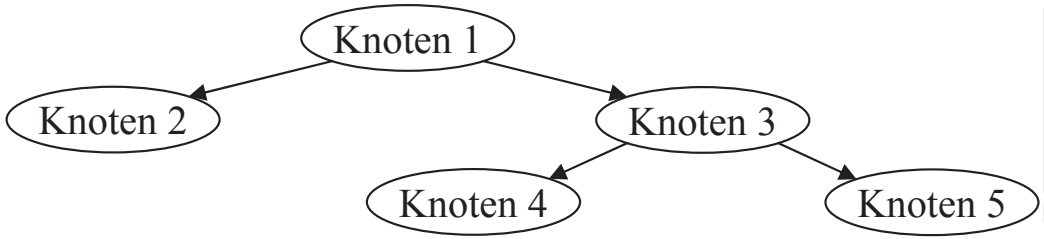


Beispiel

- Eine Beispiel-Inkarnation:



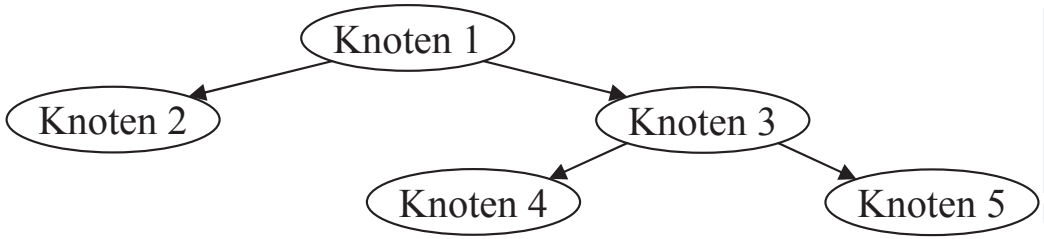
Anzahl der Knoten eines Baumes



- Unter der Annahme, entsprechende Getter-Methoden für die Wurzel und die Teilbäume definiert zu haben:

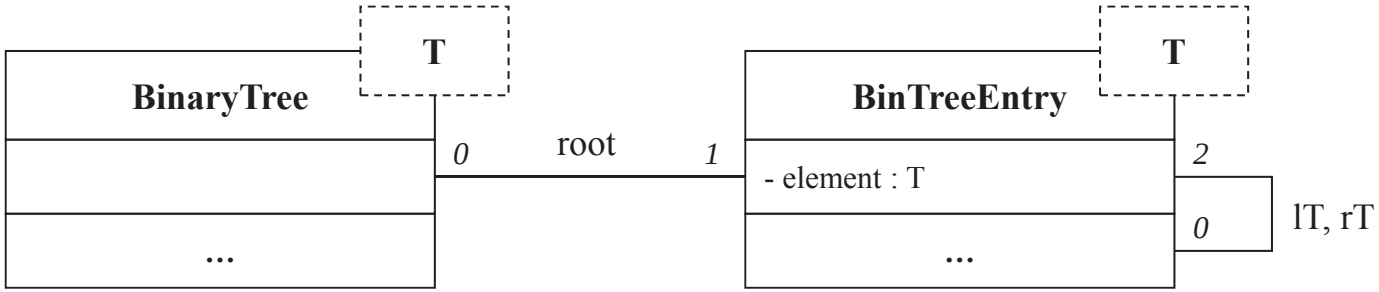
```
public int numberOfNodes() {
   if(this.getRoot() == null) {
      return 0;
   } else {
      return 1 + this.getlT().numberOfNodes() + this.getrT().numberOfNodes();
   }
}
```

Baumdurchlauf



- Ein Baum kann in verschiedenen Reihenfolgen durchlaufen werden (z.B. um einen bestimmten Eintrag zu suchen).
- *Präorder*-Reihenfolge: Wurzel – linker Teilbaum – rechter Teilbaum
  Im Beispiel: Knoten 1, Knoten 2, Knoten 3, Knoten 4, Knoten 5
- *Inorder*-Reihenfolge: linker Teilbaum – Wurzel – rechter Teilbaum
  Im Beispiel: Knoten 2, Knoten 1, Knoten 4, Knoten 3, Knoten 5
- *Postorder*-Reihenfolge: linker Teilbaum – rechter Teilbaum – Wurzel
  Im Beispiel: Knoten 2, Knoten 4, Knoten 5, Knoten 3, Knoten 1

Beispiel Präorder-Durchlauf




```
private static void preOrder(BinTree tree) {
   if(tree.getRoot() != null) {
      System.out.println(root.getElement().toString());
      preOrder(root.getlT());
      preOrder(root.getrT());
   }
```

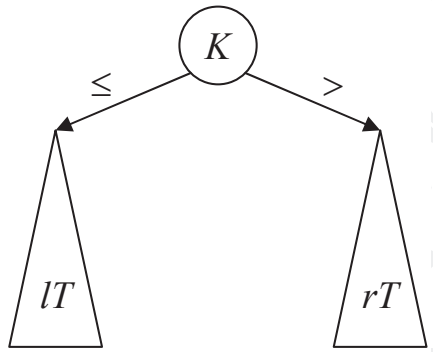
Einfügen, Löschen, Suchen

- Bisher unterscheidet sich der Aufbau eines Baumes nicht wirklich von einer Liste:
- Zum Einfügen, Löschen oder Suchen eines bestimmten Elements muss im schlechtesten Fall der gesamte Baum in einer bestimmten Reihenfolge durchlaufen werden.
- Wie kann man die hierarchische Struktur des Baumes nutzen, um die Effizienz dieser Operationen zu verbessern?
- Idee: Sortiere die Einträge im Baum so, dass eine Art binäre Suche ermöglicht wird, d.h. in jedem Knoten muss eindeutig entscheidbar sein, in welchem Teilbaum die Suche fortgesetzt werden muss.

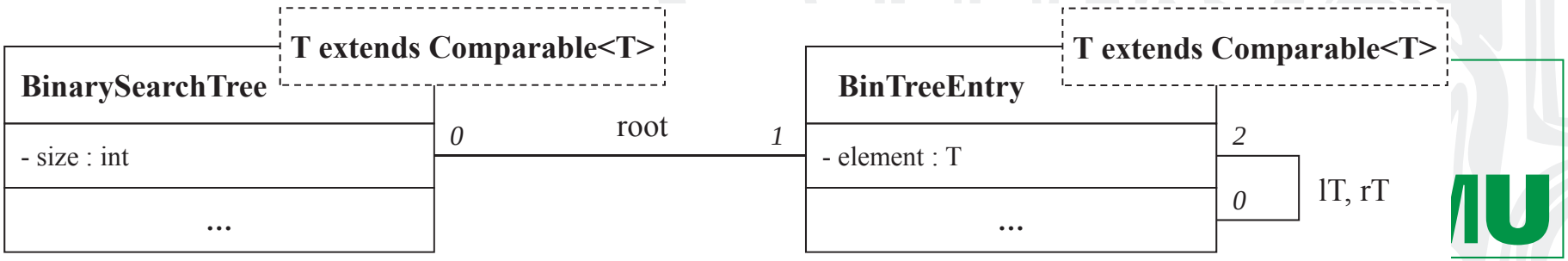
Suchbäume

- Ein (binärer) *Suchbaum* ist ein (Binär-)Baum für dessen Knoten gilt:

  Für jeden Knoten K gilt: alle Elemente in lT sind kleiner oder gleich dem Element in K und alle Elemente in rT sind größer als das Element in K (die Elemente heißen in diesem Zusammenhang auch *Schlüssel*).



- An der Datenstruktur ändert sich nichts, außer dass der Typ `T` nun das Interface `Comparable<T>` implementieren muss.



Das Interface `Comparable<T>`

- Das Interface `Comparable<T>` verlangt die Methode `compareTo`.
- Dabei wird angenommen, dass diese Methode eine totale Ordnung (auch die *natürliche Ordnung*) der Objekte des Typs `T` implementiert.
- Zur Erinnerung:
  Eine *(partielle) Ordnung* ist eine Relation, die reflexiv, transitiv und antisymmetrisch ist.
  Eine *totale Ordnung* ist eine partielle Ordnung, die außerdem total ist.
- Die Methode **`int`** `compareTo(T o)` sollte wie folgt implementiert sein:

$$R \subseteq T \times T \quad \text{mit} \quad (x, y) \in R \iff \texttt{x.compareTo(y) <= 0}$$

Natürliche Ordnung auf Objekten

Für zwei Objekte `x` und `y` vom Typ `T implements Comparable<T>` gilt:

- Der Ausdruck `x.compareTo(y)` ist vom Typ **`int`**.
- Der Wert des Ausdrucks `x.compareTo(y)` soll kleiner 0 sein, wenn `x` “kleiner” ist als `y`.
- Der Wert des Ausdrucks `x.compareTo(y)` soll größer 0 sein, wenn `x` “größer” ist als `y`.
- Der Wert des Ausdrucks `x.compareTo(y)` soll gleich 0 sein, wenn `x` “gleich” ist zu `y`.

Natürliche Ordnung auf Objekten

- Diese “Gleichheit” (`x.compareTo(y)==0`) soll normalerweise die gleiche Gleichheit sein, wie die von `x.equals(y)` angegebene, dies ist aber nicht notwendigerweise so.
- **Achtung:** Falls eine Klasse `Comparable` so implementiert wird, dass `x.compareTo(y)==0` $\not\Leftrightarrow$ `x.equals(y)`,
  sollte in der Dokumentation deutlich darauf hingewiesen werden. Die empfohlene Bemerkung ist: *Note: this class has a natural ordering that is inconsistent with equals.*

Suchbäume: Suchen

- Suchen eines Schlüssels k in einem binären Suchbaum:
  - Wenn die Wurzel R leer ist: Suche erfolglos beendet.
  - Vergleiche k mit dem Schlüssel der Wurzel R.
  - Bei Gleichheit: Suche erfolgreich beendet.
  - Wenn $k < R$.Schlüssel: suche rekursiv in lT nach k.
  - Wenn $k > R$.Schlüssel: suche rekursiv in rT nach k.

Suchbäume: Suchen

```
public BinTreeEntry suche(T k) {
 return suche(this.root, k);
}

private BinTreeEntry suche(BinarySearchTree tree, T key) {
 if(tree.root == null) {
   return null;
 }

 if(key.compareTo(tree.root.getElement()) == 0) {
   return tree.root;
 }

 if(key.compareTo(tree.root.getElement()) < 0) {
   return suche(tree.root.getlT(), key);
 }

 else {
   return suche(tree.root.getrT(), key);
 }
}
```

Überblick

10. Datenstrukturen
10.1 Einleitung
10.2 Ein Datenmodell für Listen
10.3 Doppelt-verkettete Listen
10.4 Bäume
10.5 Mengen
10.6 Exkurs: Das Collections-Framework in Java

Mengen und Multimengen

- Eine *Menge* ist eine (ungeordnete) Zusammenfassung verschiedener Objekte. Insbesondere enthält eine Menge keine Duplikate.
- Eine *Multimenge* ist eine Menge, die Duplikate enthalten kann.
- Mengen und Multimengen sind wichtige Konzepte, u.a. um Assoziationen zwischen Objekten verschiedener Klassen umzusetzen.
- Im folgenden wiederholen wir grundlegende Funktionalitäten von Mengen. Eine Implementierung sollte entsprechende Methoden für diese Funktionalitäten anbieten. Diese gelten natürlich genauso für Multimengen.

Grundlegende Operationen auf Mengen

- Leere Menge:

  $\emptyset :\rightarrow Set(T)$
- Elementbeziehung:

  $\in: T \times Set(T) \rightarrow \mathbb{B}$
- Teilmengenbeziehung:

  $\subseteq: Set(T) \times Set(T) \rightarrow \mathbb{B}$
- Vereinigung:

  $\cup : Set(T) \times Set(T) \rightarrow Set(T)$
- Durchschnitt:

  $\cap : Set(T) \times Set(T) \rightarrow Set(T)$
- Differenz:

  $\setminus : Set(T) \times Set(T) \rightarrow Set(T)$
- Kardinalität:

  $|.| : Set(T) \rightarrow \mathbb{N}$

Realisierung von Mengen

- Zur effizienten Realisierung von Mengen spielen folgende Gedanken eine Rolle:
  - Die grundlegenden Operationen sollten effizient sein.
  - Das Einfügen eines neuen Elementes sollte effizient sein. Insbesondere ist darauf zu achten, dass keine Duplikate eingefügt werden.
  - Das Löschen eines Elementes sollte effizient sein.
  - Die Suche nach einem Element sollte effizient sein.
- Für die Realisierung von Multimengen spielen natürlich die selben Gedanken eine Rolle. Allerdings muss nun auf Duplikate keine Rücksicht genommen werden.

Realisierung von Mengen: Beispiel

- Als Beispiel schauen wir uns eine Implementierung einer Menge auf Basis einer typisierten (einfach) verketteten Liste an (das war die Klasse `List<T>` ab Folie 847).

Realisierung von Mengen: Beispiel

```
public class Menge<T> {
 private List<T> elemente;

 public int cardinalitaet() {
   return this.elemente.size();
 }

 public T get(int index) {
   return this.elemente.proj(index);
 }

 public void einfuegen(T element) {
   // Methode enthaelt auf der naechsten Folie
   if(!this.enthaelt(element)) {
     this.elemente.prefix(new Entry<T>(element));
   }
 }
 ...
```

Realisierung von Mengen: Beispiel ($\in$, $\cup$)

```
  ...
  public boolean enthaelt(T t) {
    return this.elemente.contains(t);
  }

  public void vereinigung(Menge<T> andereMenge) {
    for(int i=1; i<=andereMenge.size(); i++) {
      this.prefix(newEntry<T>(andereMenge.get(i)));
    }
  }
}
```

Realisierung von Mengen: Beispiel ($\cap$)

Die Realisierung der Schnittmenge sei hier nur angedeutet:

- Man muss die Elemente einer Menge einzeln durchlaufen und prüfen, ob das entsprechende Element in der anderen Menge enthalten ist.
- Wenn ja, kommt dieses Element in die Ergebnismenge, ansonsten nicht.
- Klingt aufwändig? Ist es leider auch ...
- Alternative wäre, statt einer Liste einen sog. „assoziativen Speicher“ zu benutzen (siehe nächste Folien).

Überblick

10. Datenstrukturen
10.1 Einleitung
10.2 Ein Datenmodell für Listen
10.3 Doppelt-verkettete Listen
10.4 Bäume
10.5 Mengen
10.6 Exkurs: Das Collections-Framework in Java

Weitere Datenstrukturen

- Neben Listen und Bäumen gibt es noch weitere Klassen von Datenstrukturen, z.B. die der *assoziativen* Speicher (auch *Hashverfahren*), zu denen auch Arrays gehören.
- Ein assoziativer Speicher ist eine materialisierte Abbildung, die einen Schlüssel eines beliebigen Typs auf einen Wert eines beliebigen (meist anderen) Typs abbildet.
- Als einfachen assoziativen Speicher haben wir die Arrays kennengelernt: Ein Schlüssel vom Typ **`int`**(der Index) wird auf einen Wert (der an der Stelle `index` im Array gespeichert ist) abgebildet.

Allgemeine assoziative Speicher

- Allgemein ist ein assoziativer Speicher denkbar als Abbildung (*Hash-Funktion*) aus einer beliebigen Domäne in eine andere (oder auch die gleiche).
- Statt einer Indexmenge ist der Definitionsbereich der Abbildung also irgendeine Domäne, aus der die Schlüssel stammen.
- „Domäne“ kann in Java dabei auch eine Klasse sein, die z.B. das Kreuzprodukt von mehreren verschiedenen Domänen kapseln kann.
- Einen assoziativen Speicher nennt man auch *Map*, *Dictionary* oder *Symboltabelle*.

Allgemeine assoziative Speicher

- Beispiel:
  Wörterbuch deutsch – englisch
  - `String` $\rightarrow$ `String`
  - „hallo“ $\mapsto$ „hello“

  Platznummer für Passagier auf einem bestimmten Flug
  - `String` $\rightarrow$ **`int`** $\times$ **`char`**
  - „Peer Kroeger“ $\mapsto (1, B)$

  Hier wird man **`int`** $\times$ **`char`** in eine entsprechende Klasse kapseln.

Computerbasierte assoziative Speicher

- Wörterbücher, Telefonbücher, Lexika und ähnliche gedruckte Nachschlagewerke (also statische assoziative Speicher) unterstützen durch alphabetische Sortierung der Schlüssel (Stichwörter, Namen, ...) effiziente Suche nach einem Eintrag zu einem bestimmten Schlüssel.
- Computerbasierte assoziative Speicher sind dagegen auch dynamisch. Sie können wachsen und schrumpfen (ähnlich wie Listen gegenüber Arrays).

Collections in Java

- In Java gibt es eine große Menge an vordefinierten Klassen für mengenartige Datenstrukturen.
- Im folgenden geben wir einen kurzen Überblick über das Collections-Framework von Java.
- Zur Vertiefung empfehlen wir das intensive Studium der Dokumentationen der entsprechenden Klassen.

Das Collections-Framework

- Alle Klassen mengenartiger, assoziativer Datenstrukturen implementieren das Interface `java.util.Map`.
  Das Interface stellt Basis-Funktionalitäten wie Einfügen (`put`), Löschen (`remove`) und verschiedene Formen der Suche (`get`, `containsKey`, `containsValue`) zur Verfügung.
- Alle Klassen mengenartiger, nicht-assoziativer Datenstrukturen implementieren das Interface `java.util.Collection`.
  Das Interface verpflichtet ebenfalls zu Basis-Funktionalitäten wie Einfügen (`add`), Löschen (`remove`) und Suchen (`contains`).

Das Collections-Framework

- Vom Interface `java.util.Collection` abgeleitete Interfaces sind u.a.:
  - `java.util.Set`: Klassen, die Mengen ohne Duplikate modellieren, implementieren dieses Interface.
  - `java.util.List`: Klassen, die Multimengen modellieren, implementieren dieses Interface.

Hierarchie

Hierarchie einiger Interfaces und Klassen aus dem Collections-Framework von Java:

Collection		*Map*
List	*Set*	
LinkedList	HashSet	HashMap
ArrayList	LinkedHashSet	Hashtable
Stack	TreeSet	TreeMap
Vector		

Denken Sie daran: jede Datenstruktur hat in unterschiedlichen Aspekten jeweils gewisse Vor- bzw. Nachteile!

Die Wahl der richtigen Datenstruktur für ein Problem ist oft nicht trivial, hat aber u.U. erheblichen Einfluss auf die Eigenschaften von Algorithmen!

Wiederholung: Iteration über Arrays

- Zur Erinnerung: Iteration über ein Array

```
public static void accessFor(Integer[] intArray) {
    Integer currentInteger;
    for(int i = 0; i < intArray.length; i++) {
      currentInteger = intArray[i];
    }
 }
```

Iteration über Collections

- Um iterative Aktionen auf allen Elementen einer Collection (oder einer bestimmten Teilmenge davon) durchzuführen, sind nicht alle Schleifenkonstrukte gleichwertig.
- Beispiel:

```
public static void accessFor(DVListe<Integer> list) {
    Integer currentInteger;
    for(int i = 1; i <= list.size(); i++) {
        currentInteger = list.proj(i);
    }
}
```

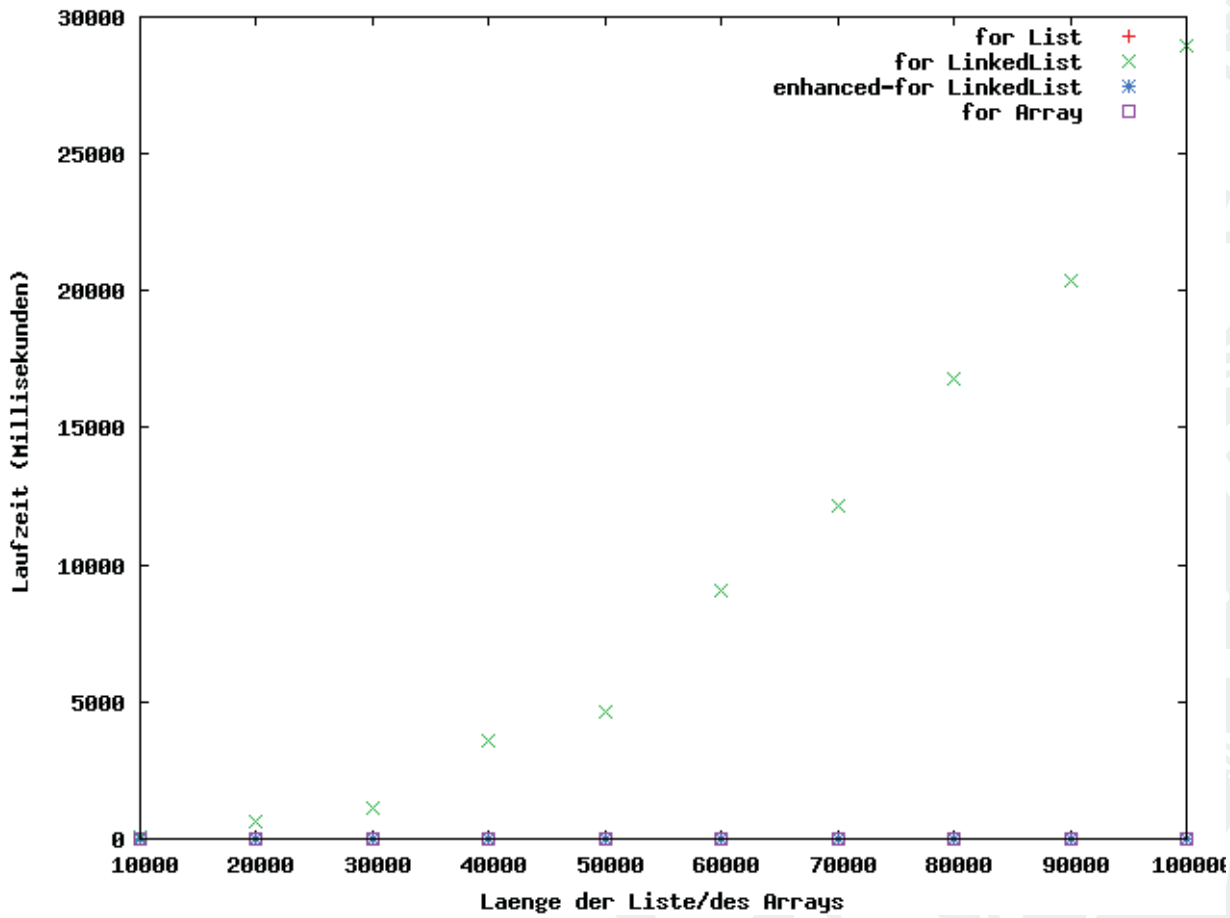
Vergleichen Sie das mal mit der Iteration über ein Array . . . Was fällt Ihnen auf?

Die `Iterator`- und `Iterable`-Interfaces und *enhanced for*

- Eine `Collection` implementiert das Interface `Iterable`, das nur vorschreibt, dass ein `Iterator` zurückgegeben werden kann, der es ermöglicht, die Collection zu durchwandern.
- Dieser `Iterator` wird auch implizit im sogenannten *enhanced for* verwendet:

```
public static void accessIterable(Iterable<Integer> intIterabl
    Integer currentInteger;
    for(Integer currentInt : intIterable) {
      currentInteger = currentInt;
    }
}
```

Test für die verschiedenen Iterationen auf verschiedenen Listen und einem Array



Lessons learned

- Wie bereits erwähnt: jede Datenstruktur hat in unterschiedlichen Aspekten jeweils gewisse Vor- bzw. Nachteile:

  Vergleich der durchschnittlichen Laufzeiten

	einfaches Array	einfache Liste	doppelt verankerte Liste	doppelt verkettete Liste	perfekte Hashverfahren (assoz. Array)	binärer Suchbaum
dynamisch	nein	ja	ja	ja	nein	ja
Einfügen	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$ bzw. $O(n)$	$O(\log n)$
Löschen	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$
Suchen	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(\log n)$

- Sie sehen: die Wahl der richtigen Datenstruktur für ein Problem ist oft wirklich nicht trivial . . .