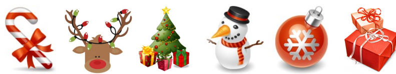
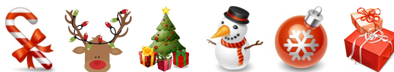
TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN



Software Entwicklung 1

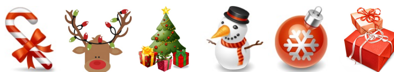
Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern



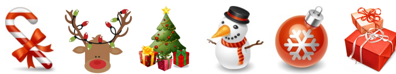
Was gefällt Ihnen?

- Sehr ausführliches Skript
- Fragen / Aufgaben in der Vorlesung
- Praktische Beispiele
- Tolle Aufgabenblätter und gutes Abgabesystem (1x genannt)
- Tutorien
- Fragen werden beantwortet



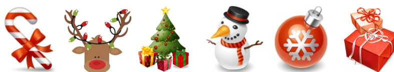
Verbesserungsvorschläge

- Folien zum Download
- Weniger Fehler im Skript
- Mehr Erklärungen im Skript
- Mehr praktische/große/schwierigere Anwendungsbeispiel in der Vorlesung
- Vorlesung um 10:00



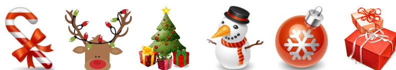
Kontroverse Punkte

- Für Anfänger (zu) schwierig vs. alles schon bekannt
- Zuviel Stoff
- Anwesenheit in den Übungen
- Wiederholungen: Gut oder schlecht?!
- "Sie beantwortet zwar Fragen jedoch stellt man Fragen nur zum Thema der Vorlesung der Rest hat in der Vorlesung recht wenig verloren."



Was uns gefällt!

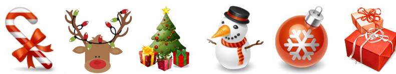
- Dass Sie Fragen stellen und mitdenken!
- Schnelle Rückmeldung bei Fehlern und Ungenauigkeiten auf den Aufgabenblättern
- Dass Sie nicht auf den letzten Moment warten...



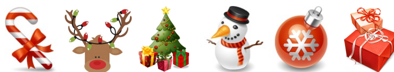
Übesicht: Die wichtigsten Collections

Interface	Hashtable	Resizable Array	Balanced Tree	LinkedList	Hashtable + LinkedList
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Quelle: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html



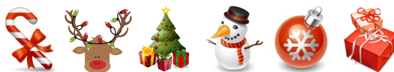
Abstrakte Datentypen: Maps



Abstrakte Datentypen: Maps

Indexstrukturen (engl. *maps* oder *dictionary*) erlauben eine effiziente Verwaltung von Daten (hier: Objekten).
Ein Datensatz besteht aus einem eindeutigen Schlüssel und dem ihm zugeordneten Wert.
Die Verwaltung von Datensätzen basiert auf den folgenden drei Grundoperationen:

- Einfügen eines Datensatzes in eine Menge von Datensätzen
- Suchen eines Datensatzes mit Schlüssel k
- Löschen eines Datensatzes mit Schlüssel k

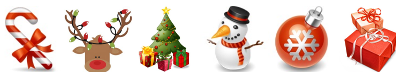


Implementierungen für Maps

In vereinfachter Anlehnung an `java.util.Map` legen wir für die hier diskutierten Implementierungen folgende Schnittstelle zugrunde:

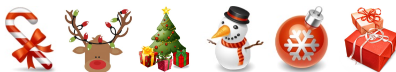
```
interface Map {
   String get(int key);
   void   put(int key, String value);
   void   remove(int key);
}
```

- Nicht-parametrisierte Variante mit `int`-Schlüsseln und `String`-Daten.
- Für jeden Schlüssel max. einen Datensatz



Beispiel: Abbildung Zahl auf Monatsname

```
Map monate = new ... // Implementierung folgt spaeter
monate.put(1, "Januar");
monate.put(2, "Februar");
...
monate.put(12, "Dezember");
System.out.println("Monat: " + monate.get(4));
// Monat: April
```



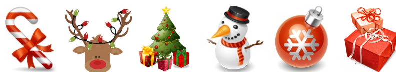
Implementierungen für Maps

Hier: Drei Varianten aufbauend auf verschiedenen Datenstrukturen

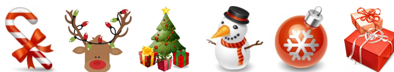
- A. Liste
- B. Suchbäume
- C. Hashtabellen

Dabei betracheten wir jeweils

- die Datenstruktur
- die drei grundlegenden Operationen
- eine einfache Komplexitätsabschätzung



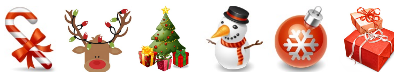
A. Maps basierend auf Listen



Maps basierend auf Listen

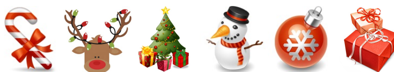
- Hier: Vereinfachend ohne Getter und Setter
- Zum Suchen etc. wird der Schlüssel der Datensätze verwendet Ergebnis ist der String `value`

```
// Repraesentierung eines Datensatzes
class DataSet {
  int key;
  String value;
  DataSet (int k, String v) {
    this.key = k;
    this.value = v;
  }
}
```



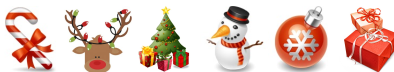
Implementierung I

```
import java.util.*;

class ListMap implements Map {
  private List<DataSet> elems;
  public ListMap() {
    elems = new ArrayList<DataSet>();
  }

  public String get (int key) {
    for (DataSet d : elems) {
      if (d.key == key) {
        return d.data;
      }
    }
    return null;
  }
```



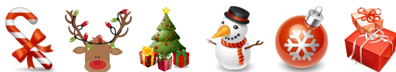
Implementierung II

```
public void put(int key, Strings value) {
  for (DataSet d : elems) {
    if (d.key == key) {
      d.data = value;
      return;
    }
  }
  elems.add(new DataSet(key,value));
}
}
```



Implementierung III

```
// Elemente koennen nur ueber Iterator
// waehrend des Iterierens entfernt werden!

public void remove(int key) {
 Iterator<DataSet> iter = elems.iterator();
 while (iter.hasNext()) {
  DataSet d = iter.next();
  if (d.key == key) {
    iter.remove();
    return;
  }
 }
}
```

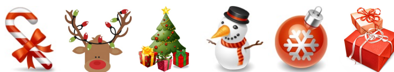


Diskussion

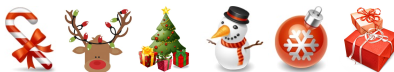
- Vorteile
  - → Einfach zu implementieren
- Nachteile
  - → Suchen, Einfügen und Löschen sind langsam
  - → Linearer Aufwand: Man muss über bis zu N Elemente iterieren, um in der List-Map mit N Einträgen ein Element zu finden.

Frage

In welchem Fall benötigt man N Schritte, um ein Element zu finden?



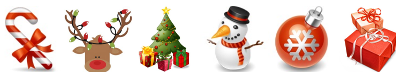
B. Maps basierend auf Suchbäumen



Maps basierend auf Suchbäumen

- Suche von Elementen auf Suchbäumen (aka binären markierten Binärbäumen) ist effizienter als die Suche in einer Liste ($\Rightarrow$ Vorlesung zu Suchen und Sortieren)
- Verwende als Knoten-Markierung den Schlüssel
- Weiteres Attribut `value` für Daten-Objekt
- Methoden wie bei `SortedBinTree`

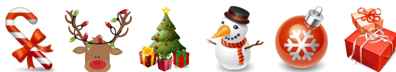
```
// Knoten fuer Binaerbaeume
class TreeNode {
    int key;
    String value;
    TreeNode left, right;

    TreeNode(int k, String v) {
        this.key  = k;
        this.value = v;
    }
}
```



C. Hashing/Streuspeicherung

Idee: *Berechne* aus dem Schlüssel die Positionsinformation des Datensatzes (z.B. den Arrayindex)!

Für viele praktisch relevante Szenarien erreicht man dadurch Datenzugriff mit *konstantem* Aufwand.



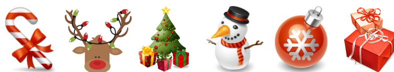
Begriffsklärung: Hashfunktion, -tabelle I

Seien

- S die Menge der möglichen Schlüsselwerte (*Schlüsselraum*) und
- A die Menge von Adressen in einer Hashtabelle (im Folgenden ist A immer die Indexmenge $0 \ldots m-1$ eines Arrays).

Eine **Hashfunktion** $h : S \rightarrow A$ ordnet jedem Schlüssel eine Adresse in der Hashtabelle zu.

Als **Hashtabelle (HT)** der Größe m bezeichnen wir einen Speicherbereich, auf den über die Adressen aus A mit konstantem Aufwand (also unabhängig von m) zugegriffen werden kann.



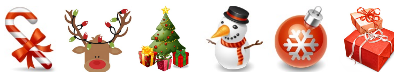
Begriffsklärung: Hashfunktion, -tabelle II

Enthält S weniger Elemente als A, kann h injektiv sein:

$$\text{Für alle } s, t \text{ in } S: \quad s \neq t \Rightarrow h(s) \neq h(t)$$

D.h. die Hashfunktion ordnet jedem Schlüssel eine eineindeutige Adresse zu.

Andernfalls können Kollisionen auftreten.



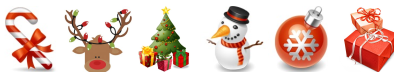
Begriffsklärung: Kollision, Synonym

Zwei Schlüssel s, t **kollidieren** bezüglich einer Hashfunktion h, wenn $h(s) = h(t)$.

Die Schlüssel s und t nennt man dann **Synonyme**.

Die Menge der Synonyme bezüglich einer Adresse a aus A heißt die **Kollisionsklasse** von a.

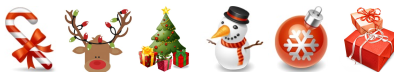
Ist schon ein Datensatz mit Schlüssel s in der Hashtabelle gespeichert, nennt man einen Datensatz mit einem Synonym von s einen **Überläufer**.



Anforderungen an Hashfunktionen

Eine Hashfunktion soll

- sich einfach und effizient berechnen lassen
- zu einer möglichst gleichmäßigen Belegung der Hashtabelle führen
- möglichst wenige Kollisionen verursachen

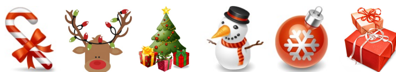


Klassifikation von Hashverfahren

Hashverfahren unterscheiden sich

- durch die Hashfunktion
- durch die Kollisionsauflösung:
  - **Verkettung**: Überläufer werden in einer Liste an der Position der Hashtabelle eingefügt
  - **offen**: Überläufer werden an noch offenen Positionen der Hashtabelle gespeichert
- durch die Wahl der Größe der Hashtabelle:
  - **statisch**: Die Größe wird bei der Erzeugung festgelegt und bleibt unverändert.
  - **dynamisch**: Die Größe kann angepasst werden.

Hier: Statische Hashtabelle mit Kollisionsauflösung durch Verkettung



Skizze

key3 | v3 | •

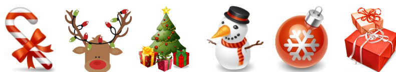
key2 | v2 | •

key1 | v1 | •

h(key) : A

key : S

h(.)



Hashfunktion

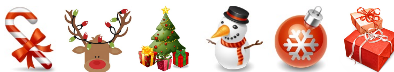
Entscheidend ist, dass die Hashfunktion die Schlüssel gut streut.

Verbreitetes Verfahren:

- Wähle eine Primzahl als Hashtabellen-Größe.
- Wähle den ganzzahligen Divisionsrest als Hashwert:

```
private  int hash( int key ) {
   return Math.abs(key % hashtable.length);
 }
```

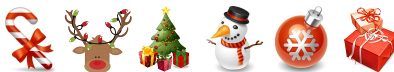
TECHNISCHE UNIVERSITÄT KAISERSLAUTERN



Datenstruktur I

Ein Eintrag in der Hashtabelle wird durch ein Objekt der folgenden Klasse repräsentiert:

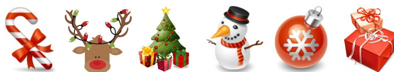
```
class HashEntry {
  int key;
  String value;
  HashEntry next;
  HashEntry(int k, String v) {
    this.key = k;
    this.value = v;
  }
}
```



Datenstruktur II

Wir realisieren eine Hashtabelle als Implementierung der Schnittstelle Map:

```
class HashMap implements Map {
  private HashEntry[] table;

  public HashMap( int tabsize ) {
    /* tabsize sollte eine Primzahl sein */
    this.table = new HashEntry[tabsize];
  }
  private  int hash( int key ) { ... }
  public String get( int key ) { ... }
  public void put (int key,String v ){ ... }
  public void remove( int key ) { ... }
}
```



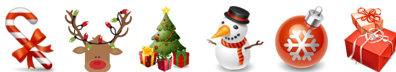
Datenstruktur III

Invarianten:

- Die Hashtabelle enthält den Datensatz zu s, wenn
  - → `table[ h(s) ]  != null` und
  - → für ein Element `entry` der verlinkten Liste startend bei `table[ h(s) ]` gilt: `entry.key == s`

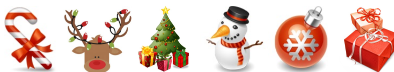
  Die Daten liefert dann `entry.value` .

- Alle Elemente der Kollisionsklasse zu $h(s)$ befinden sich in der Liste an Position $h(s)$ der Hashtabelle.



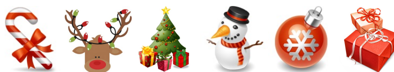
Einfügen

```
public void put ( int key, String value ) {
  if(value != null) {
    int hix = hash(key);
    HashEntry currentEntry = table[hix];
    HashEntry newEntry = new HashEntry(key, value);
    newEntry.next = currentEntry;
    table[hix] = newEntry;
  }
}
```



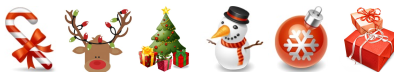
Suchen

```
public String get( int key ) {
  int hix = hash(key);
  HashEntry entry = table[hix];

  while(entry != null) {
    if (entry.key == key) {
      return entry.value;
    }
    entry = entry.next;
  }
  return null;
}
```



Löschen

```
public void remove( int key ) {
    int hix = hash(key);
    HashEntry entry = table[hix];

    // entfernen alle HashEntries mit dem gegebenen Schluessel
    HashEntry dummy = new HashEntry(0, null);
    dummy.next = entry;
    HashEntry current = dummy;
    while(current.next != null) {
      if(current.next.key == key) { // entferne den Eintrag
        current.next = current.next.next;
      } else { // gehe zum naechsten Eintrag
        current = current.next;
      }
    }
    table[hix] = dummy.next;
  }
```

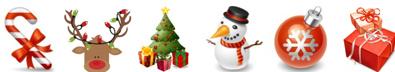


Diskussion

Die Komplexität der Operationen einer Hashtabelle ist abhängig von

- der Hashfunktion und dem Füllungsgrad der Tabelle
- dem Verfahren zur Kollisionsauflösung

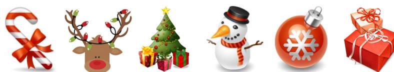
Bei guter Hashfunktion und kleinem Füllungsgrad kommt man im Mittel mit konstant vielen Operationen aus.



Bemerkung

Die gezeigte Implementierung ist nicht optimal.

Im schlechtesten Fall haben die Operationen einen Aufwand, der linear mit der Anzahl der Einträge in der Hashtabelle zunimmt.
Wann tritt dieser ungünstigste Fall ein?



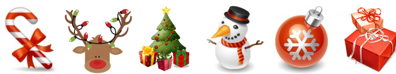
Bemerkung

Die gezeigte Implementierung ist nicht optimal.

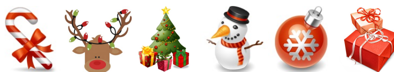
Im schlechtesten Fall haben die Operationen einen Aufwand, der linear mit der Anzahl der Einträge in der Hashtabelle zunimmt.
Wann tritt dieser ungünstigste Fall ein?

Mögliche Optimierungen:

- Sortierung der `HashEntry`-Liste
- Implementierung der Kollisionsauflösung mit Hilfe eines Suchbaums

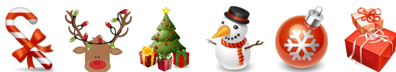


Generische Maps



Maps im Java Collections Framework

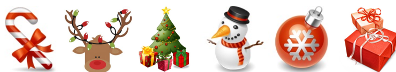
```
interface Map<K,V> {
  // Liefert das Objekt unter Schluessel key;
  // falls nicht vorhanden, wird null zurueckgegeben
  V get(Object key);

  // Fuegt den Wert value unter Schluessel key ein
  // liefert das Objekt, der zuvor unter key eingetragen war;
  // falls nicht vorhanden, wird null zurueckgegeben
  V put(K key, V value);

  // Entfernt den Eintrag unter Schluessel key;
  // Liefert das bisher eingetragene Objekt
  // falls nicht vorhanden, wird null zurueckgegeben
  V remove(Object key);

  // Liefert ein Set mit allen eingetragenen Schluesseln
  Set<K>  keySet();

  // Liefert ein Set mit allen Eintraegen
  Set<Map.Entry<K,V>> entrySet();
  ...
}
```



Hashen von Objekten

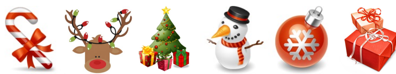
Bis jetzt haben wir als Schlüssel nur ganzzahlige Werte betrachtet. Bei der generischen Implementierung sind Schlüssel von einem Objekttyp.

Ein als Objekttyp vorliegender Schlüssel muss in einen numerischen Wert umgewandelt werden (Hashadresse). In Java implementiert und verwendet man dazu die Methode `int hashCode()` der Klasse `Object`.

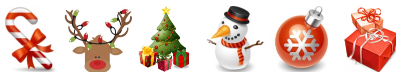


Hashen von Objekten (Code)

```
class HashMap<K, V> implements Map<K, V> { ...
  private int hash( K key ) {
    return Math.abs(key.hashCode() % table.length);
  }
  public V get( K key ) {
    int hix = hash(key);
    HashEntry<K, V> entry = table[hix];

    while(entry != null) {
      if (entry.key.equals(key)) {
        return entry.value;
      }
      entry = entry.next;
    }
    return null;
  }
}
```

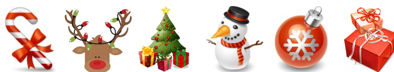


Vergleichen von Objekten



Vergleich von Objekten

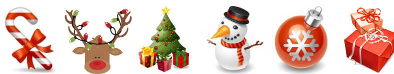
- Um Objekte zu sortieren ($\Rightarrow$ TreeMap), muss eine *Ordnung* auf ihnen definiert werden.
- Java: `Comparator` Interface



Das `Comparator` Interface

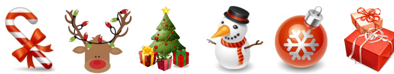
```
interface Comparator<T> {
  public int compare(T o1, T o2);
}
```

- `compare (x,y)` liefert einen `int`-Wert
  - < 0, falls `x` kleiner als `y`
  - $= 0$, falls `x` gleich `y`
  - > 0, falls `x` größer als `y`
- Merkhilfe:
  `compare(x,y) < 0`, genau dann wenn $x < y$
- Forderung: Konsistenz mit `equals` bei Verwendung mit SortedSet or SortedMap
  - `compare (x,y) == 0` genau dann, wenn `x.equals(y)` den Wert `true` liefert



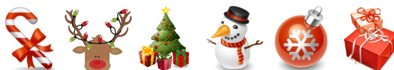
Beispiel: Date-Objekte vergleichen

```
import java.util.Comparator;
public class DateComparator implements Comparator<Date> {
  public int compare(Date d1, Date d2) {
    int result = d1.getYear() - d2.getYear();
    if (result == 0) {
      result = d1.getMonth() - d2.getMonth();
      if (result == 0) {
        result = d1.getDay() - d2.getDay();
      }
    }
    return result;
  }
}
```



Erwünschte Eigenschaften von `equals()`

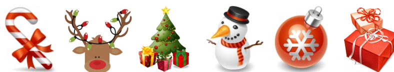
- *reflexiv*: Für jede Referenz `x` mit x != null,sollte `x.equals(x)` den Wert `true` liefern.
- *symmetrisch*: Für jede Referenz `x` und `y` mit x,y != null, sollte `x.equals(y)` den Wert `true` liefern, genau dann wenn `y.equals(x)` `true` liefert.
- *transitive*: Für jede Referenz `x,y,z` mit x,y,z != null, sollte `x.equals(z)` den Wert `true` liefern, wenn `x.equals(y)` und `y.equals(z)` `true` liefert.
- *konsistent*: Mehrfache Aufrufe von `x.equals(y)` sollten `true` bzw. `false` liefern, solange die referenzierten Objekte zwischenzeitlich nicht verändert wurden.
- Falls `x != null`, sollte `x.equals(null)` immer `false` liefern.



Anpassung von `equals` in Klasse `Date`

Die `equals()` Methode wird von der Klasse `Object` geerbt und kann bzw. muss bisweilen überschrieben und angepasst werden.

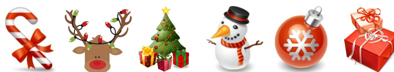
```
public class Date {
    ...
    public boolean equals(Object obj) {
      if (this == obj) {
         return true;
      }
      if (obj == null) {
         return false;
      }
      if (!(obj instanceof Date )) {
         return false; // andere Klasse
      }
      Date other = (Date) obj;
      return this.year == other.year
        && this.month == other.month
        && this.day == other.day;
    }
}
```



equals() und hashCode()

- Gleiche Objekte müssen gleichen Hashcode haben (⇒ Suchen von Einträgen in Hashtable)

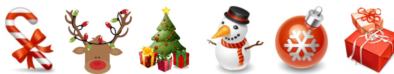
```
Date x = new Date(24,12,2016);
Date y = new Date(24,12,2016);
assertEquals(x,y);      // mit angepasster equals() Methode
assertEquals(x.hashCode(),y.hasCode()); // wuenschenswert
```



Ein schlechter Ansatz: Summe der Attribute

```
public class Date {
    ...
    public int hashCode() {
        return this.day + this.month + this.year;
    }
}
```

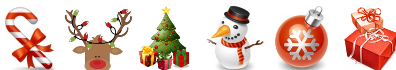
- Korrekt: Gleiche Objekte haben gleichen Hashcode
- Hohe Kollisionswahrscheinlichkeit: `new Date(24,12,2016)`, `new Date(23,11,2016)`, `new Date(23,12,2017)}`, ...



Besser: `Objects.hash()`

- Die Implementierung einer `hashCode()` Methode, die die Ergebnisse optimal streut, ist nicht einfach.
- Gutes Schema: Multipliziere die einzelnen Komponenten, die bei der Berechnung des Hashwertes einfließen, mit einer Primzahl multipliziert und addiere dann
- `hash` nimmt eine beliebig Anzahl an Parametern

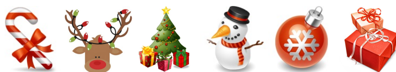
```
public class Date {
  ...
  public int hashCode() {
    return Objects.hash(this.day, this.month, this.year);
  }
}
```



Zusammenfassung: Die wichtigsten Collections

Interface	Hashtable	Resizable Array	Balanced Tree	LinkedList	Hashtable + LinkedList
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Quelle: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html



Typische Verwendung

- Hinzufügen und Entfernen von Elementen
- Ausgabe aller Elemente
- Zusammenführen / Vereinigen von Collections
- Filtern von Elementen nach bestimmten Kriterien
- Transformieren von einer Liste in eine andere Liste
- Aggregieren von Information