Maps

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Wir haben in Kapitel 13 bereits einen Überblick über die wichtigsten Collections. In diesem Kapitel betrachten wir ergänzend Implementierungen für Maps.

1 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

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

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

Um die Implementierung zu vereinfachen, betrachten wir zunächst eine nicht-parametrisierte Variante mit `int`-Schlüsseln und `Object`-Daten.

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));
// Erwartete Ausgabe:   Monat: April
```

Ziel ist es, *Datenstrukturen* für die Implementierung auszuwählen, auf denen sich die obige Operationen Suchen, Einfügen und Entfernen effizient ausführen lassen.
Hier behandeln wir drei Implementierungsvarianten:

- Listen
- Suchbäume
- Hashing

Dabei betracheten wir jeweils

- die Datenstruktur
- die drei grundlegenden Operationen
- eine einfache Aufwandsabschätzung für die Operationen

1.1 Maps basierend auf Listen

Zum Einstieg betrachten wir eine Map-Implementierung, der eine Liste als Datenstruktur zugrunde liegt. Die Einträge in der Liste sind Datensätze, d.h. Kombinationen aus Schlüssel und Datenobjekt. Ein Datensatz wird dabei durch folgende Klasse repräsentiert:

```
class DataSet {
  int key;
  Object data;
  DataSet (int k, Object d) {
    key = k;
    data = d;
  }
}
```

Für bessere Übersichtlichkeit auf den Folien und im Skript verzichten wir hier auf Getter/Setter-Methoden.

Datenstruktur Zur Verwaltung der Datensätze verwenden wir hier eine Array-Liste, in die wir die Datensätze einfügen können.

```
import java.util.*;

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

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

```
    }
    return null;
  }

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

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

Entfernen von Elementen aus einer Collection Der Rumpf von `foreach` darf die unterliegende Collection **nicht** ändern. Der folgende Code liefert bei der Ausführung eine `ConcurrentModificationException`, sobald aus der Collection `tasks` ein Task während der Iteration über die Elemente entfernt wird:

```
for (DataSet d : elems) {
  if (d.key == key) {
    elems.remove(d); // ConcurrentModificationException !!!
  }
}
```

Auch das Auflösen des `foreach` hilft **nicht**:

```
Iterator<DataSet> iter = elems.iterator();
while (iter.hasNext()) {
  DataSet d = iter.next();
  if (d.key == key) {
    elems.remove(d); // ConcurrentModificationException !!!
  }
}
```

Während ein Iterator aktiv ist, dürfen Veränderungen an der Collection nur über den Iterator selbst vorgenommen werden.

```
Iterator<DataSet> iter = elems.iterator();
```

```java
while (iter.hasNext()) {
  DataSet d = iter.next();
  if (d.key == key) {
    iter.remove();
  }
}
```

Dazu muss die `remove`-Methode aus dem `Iterator`-Interface verwendet werden.

Vor- und Nachteile dieser List-Maps List-Maps erlauben eine vergleichsweise einfache und speichersparende Implementierung von Maps. Das Suchen, Einfügen und Löschen aber ist vergleichsweise langsam. Man benötigt in der gezeigten Variante z.B. linearen Aufwand um ein Element zu finden: Man muss über bis zu N Elemente iterieren, um in der List-Map mit N Einträgen ein Element zu finden.

Frage 1: In welchem Fall benötigt man N Schritte, um ein Element zu finden?

Hinweis: In sortierten Arrays kann ein Datensatz mit Hilfe von binärer Suche wesentlich effizienter gefunden werden. Diese Variante der Array-Map betrachten wir in einem späteren Kapitel.

1.2 Suchbäume

In Kapitel 11 haben wir natürliche binäre Suchbäume (sortierte markierte Binärbäume) betrachtet. Auch diese Datenstruktur wird Implementierungen von Maps zugrunde gelegt (z.B. `TreeMap` im Java-Collections Framework).
Wir zeigen hier eine solche Implementierung. Um in einem Knoten einen Datensatz abzuspeichern, verwenden wir den Schlüssel als Markierung; das Datenobjekt kommt als weiteres Attribut hinzu.

```java
// Knoten fuer Binaerbaeume
class TreeNode {
    int key;
    Object data;
    TreeNode left, right;

    public TreeNode(int k, Object d) {
        key  = k;
        data = d;
    }
}
```

Damit ergibt sich folgendes Klassengerüst:

```java
//Repraesentiert einen sortierten markierten Binaerbaum
class TreeMap implements Map {
    private TreeNode root;
    public Object get( int key )       { ... }
    public void put (int key,Object v ) { ... }
```

```
    public void remove( int key )       { ... }
}
```

Analog zur Implementierung in Kapitel 11 wird die Methoden `get` rekursiv implementiert.

```
    public Object get(int key) {
        return get(root, key);
    }

    private Object get(TreeNode node, int key) {
        if (node == null) {
            return null;
        }
        if (key < node.key) {
            // kleinere Elemente links suchen
            return get(node.left, key);
        } else if (key > node.key) {
            // groessere Elemente rechts suchen
            return get(node.right, key);
        } else {
            // gefunden!
            return node.data;
        }
    }
```

Die Implementierung von Methode `put` soll wie bei der `ListMap`-Implementierung sicherstellen, dass jeder Schlüssel max. einmal vorhanden ist. Gibt es bereits einen Knoten mit dem `key`, wird lediglich das zugehörige Datenobjekt aktualisiert. Andernfalls wird ein weiterer Knoten eingefügt. Das Einfügen (wie auch das Entfernen) verändert die Struktur des Baums; d.h. Referenzen auf linke und rechte Kinder müssen (potentiell) verändert werden. Die hier gezeigte rekursive Implementierung verwendet einen "Trick": Der Baum wird beim Aufruf von `put` re-konstruiert.

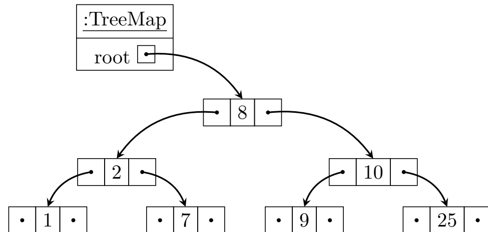
```
/** fuegt ein Element dem Baum hinzu, falls kein Eintrag
    fuer den key vorhanden, sonst wird der Eintrag geupdated */
public void put(int key, Object data) {
    root = put(root, key, data);
}
private TreeNode put(TreeNode node, int key, Object data){
    // Eintrag noch nicht vorhanden
    if (node == null) {
        return new TreeNode(key, data);
    } // Update von Eintrag
    if (key == node.key) {
        node.data = data;
    }
    else if (key < node.key) {
        node.left = put(node.left, key, data);
    } else {
        node.right = put(node.right, key, data);
    }
    return node;
}
```

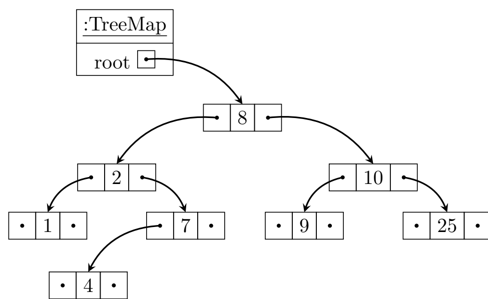
Alternativ können beim rekursiven Aufruf die Referenz auf den Elternknoten mitgegeben werden, sowie die Information, ob es sich um den linken oder rechten Kindknoten handelt.

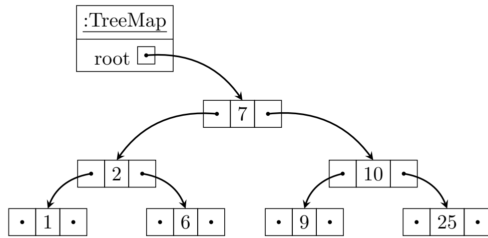
Löschen Das Entfernen von Knoten aus einem Suchbaum haben wir bisher noch nicht betrachtet. Wir unterscheiden drei Fälle:

- Knoten, die keine Kinder haben (d.h. Blattknoten), können direkt gelöscht werden, indem die entsprechende Referenz des Elternknotens auf `null` gesetzt wird.
- Bei Knoten, die nur ein Kind haben, können wir einfach das eine Kind des zu löschenden Knoten an die Stelle des zu löschenden Knoten setzen. Dadurch bleibt die Suchbaum-Eigenschaft des Baumes erhalten.
- Hat ein Knoten zwei Kinder, können wir ihn durch einen direkten Nachbarn ersetzen, der dem Teilbaum entstammt, der durch den zu löschenden Knoten aufgespannt wird. Mit direktem Nachbarn ist der Knoten mit der nächst größeren oder nächst kleineren Markierung aus diesem Teilbaum gemeint. Diesen nennt man auch ***in-order*** Nachbar.
  Hat der in-order Nachbar des zu löschenden Knoten selbst Kinder, so kann er höchstens ein Kind haben. Wenn wir den in-order Nachbarn verschieben, löschen wir damit im Prinzip also wieder entweder ein Blatt oder einen Knoten mit nur einem Kind. Für diese Löschoperation können wir uns auf die ersten beiden Fälle beziehen.

> **Frage 2:** Wie sieht der Suchbaum jeweils aus, wenn wir das Element mit Markierung bzw. Schlüssel 7 löschen?








```java
/** entfernt ein Element aus dem Baum */
public void remove(int key) {
    root = remove(root, key);
}

/** entfernt ein Element aus einem Teilbaum */
private TreeNode remove(TreeNode node, int key) {
    if (node == null) {
        // aus leerem Teilbaum kann kein Element entfernt werden
        return null;
    }
    if (key < node.key) {
        // suche im linken Teilbaum
        TreeNode newLeft = remove(node.left, key);
        node.left = newLeft;
        return node;
    } else if (key > node.key) {
        // suche im rechten Teilbaum
        TreeNode newRight = remove(node.right, key);
        node.right = newRight;
        return node;
    } else {
        // zu loeschendes Element gefunden
        if (node.left == null) {
            // wenn der linke Teilbaum leer ist,
            // dann nur den rechten nehmen
            return node.right;
        } else if (node.right == null) {
            // analog zum Fall node.left == null
            return node.left;
        } else {
            // wenn der zu löschende Knoten zwei Kinder hat,
            // den Knoten mit dem groessten Element aus dem
            //  linken Teilbaum suchen:
            TreeNode maxNodeLeft = maxNode(node.left);
            // Die Markierung und Daten von diesem Knoten nehmen
            // wir fuer den aktuellen:
            node.key = maxNodeLeft.key;
            node.data = maxNodeLeft.data;
            // Und dann loeschen wir das Element aus dem
            // linken Teilbaum:
            node.left = remove(node.left, node.key);
            return node;
        }
    }
}
```

```
    /** liefert den Knoten mit der groessten Markierung im Teilbaum */
    private TreeNode maxNode(TreeNode node) {
        if (node.right == null) {
            return node;
        }
        return maxNode(node.right);
    }
}
```

1.2.1 Balancierte Suchbäume

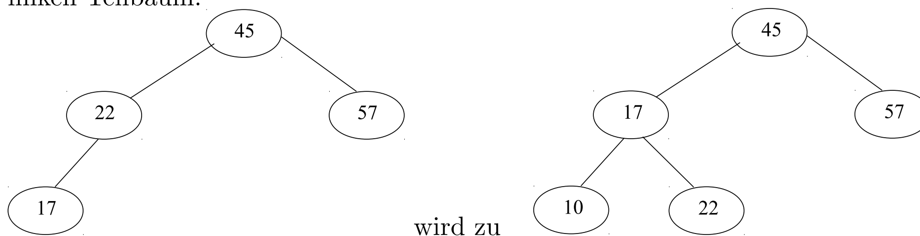
Bei degenerierten binäre Suchbäume erfordern alle drei Grundoperationen Suchen, Einfügen und Löschen im ungünstigsten Fall, dass alle Knoten besucht werden müssen. Der Aufwand für diese Operationen hängt wesentlich von der maximalen Länge des Pfades von der Wurzel des Baums bis zu den Knoten ab. Ziel ist es daher, bei den modifizierenden Operationen (Einfügen und Entfernen) den Baum wenn nötig wieder ***auszubalancieren***.
Durch Anforderungen bzgl. einer Verteilung der Blätter und Höhen in Unterbäumen kann man ein Degenerieren verhindern.

- Vorteil: geringer Aufwand für Grundoperationen kann zugesichert werden
- Nachteil: Strukturinvariante der Balancierung muss erhalten werden

Beispiel: AVL-Baum Adelson-Velskij und Landis schlugen folgende Balancierungseigenschaft vor:
Ein binärer Suchbaum heißt höhenbalanciert, wenn für jeden Knoten K gilt:

> Die Höhe des linken Unterbaums von K unterscheidet sich von der Höhe des rechten Unterbaums höchstens um eins.

Das Einfügen von Schlüssel 10 erfordert ein Vertauschen (Rotieren) der Knoten im linken Teilbaum:



Bei balancierten Suchbäumen verursachen die Such-, Einfüge- und Löschoperation (inkl. Rebalancierungsoperationen) jeweils logarithmischen Aufwand. Die Literatur listet eine Reihe verschiedener Balancierungsstrategien (RedBlack-Trees, AVL-Trees, B-Trees, Splay-Trees, etc.). Diese werden in der Vorlesung EAA bzw. “Algorithmen und Datenstrukturen” detailliert behandelt und analysiert.

1.3 Hashing/Streuspeicherung

Anstatt durch schrittweises Vergleichen von Schlüsseln auf einen Datensatz zuzugreifen, versucht man bei Hash- oder Streuspeicherverfahren aus dem Schlüssel die Positionsinformation des Datensatzes (z.B. den Arrayindex) zu *berechnen*. Für viele praktisch relevante Szenarien erreicht man dadurch Datenzugriff mit *konstantem* Aufwand.
Seien S die Menge der möglichen Schlüsselwerte (***Schlüsselraum***) und A die Menge von Adressen in einer Hashtabelle. (Im Folgenden ist A immer die Indexmenge $0 \dots m-1$ eines Feldes).
Eine ***Hashfunktion*** $h : S \to A$ 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.
Enthält S weniger Elemente als A, kann h injektiv sein:

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

D.h. die Hashfunktion ordnet jedem Schlüssel eine eineindeutige Adresse zu. Dann ist perfektes Hashing möglich. Andernfalls können Kollisionen auftreten.
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

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.

1.3.1 Kollisionsauflösung durch Verkettung

Wir betrachten im Folgenden eine Realisierung einer statischen Hashtabelle mit Kollisionsauflösung durch Verkettung. Entscheidend für die Effizienz des Hashverfahrens ist, dass die Hashfunktion die Schlüssel gut streut. Ein verbreitetes Verfahren ist folgendes:

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

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

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

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

Durch das Attribut `next` können Einträge miteinander verkettet werden.
Wir realisieren eine Hashtabelle nun als Implementierung des `Map`-Interfaces:

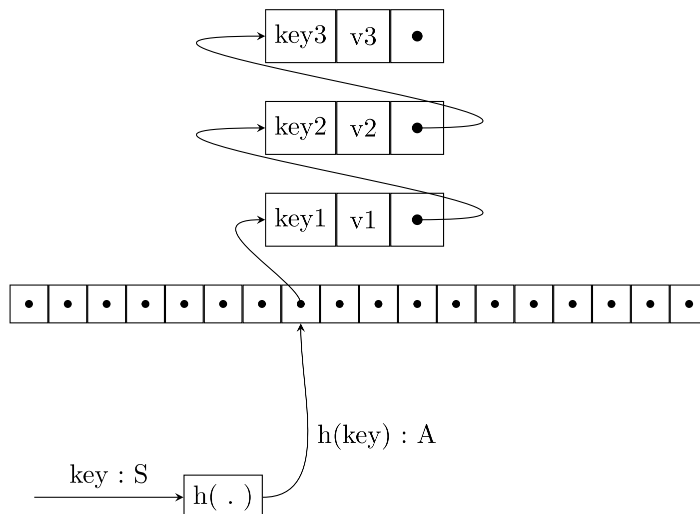
```
class HashMap implements Map {
  private HashEntry[] table;

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

Sei s ein Schlüssel, $h(s)$ sein Hashwert. Der Datenstruktur HashMap liegen folgende Invarianten zugrunde:

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

Die Daten liefert dann `entry.value` .

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



Beim Setzen von Einträgen in der Hashtabelle muss man sich entscheiden, ob man bestehende Einträge mit gleichem Schlüssel ersetzt (höhere Laufzeit), oder den Eintrag zunächst hinzufügt und alte Einträge erst bei Gelegenheit entfernt (höherer Speicherbedarf). In der hier gezeigten Implementierung werden alte Einträge nicht entfernt.

Einfügen

```
public void put ( int key, Object 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 Object 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;
  }
```

Laufzeitabschätzung Die Komplexität der Operationen einer Hashtabelle ist abhängig von

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

Bei guter Hashfunktion und kleinem Füllungsgrad kommt man im Mittel mit einer konstanten Anzahl von 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.

Frage 3: Wann tritt dieser ungünstigste Fall ein?

Mögliche Optimierungen:

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

2 Generische Maps

Im Allgemeinen möchte man sich nicht auf Schlüssel des Typs `int` und Werte des Typs `Object` beschränken. Man möchte eine generische Variante des Interfaces `Map` verwenden.
Das Java Collection Framework beinhaltet eine solche generische Variante des `Map`-Interfaces: `java.util.Map<K, V>`. Hier die wichtigsten Methoden:

```
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();
  ...
}
```

- Das `Map`-Interface ist parametrisiert über `K` (Typ der Schlüssel) und `V` (Typ der Daten).
- Wichtige implementierende Klassen: `TreeMap`, `HashMap`
- Die Einträge der `Map` sind vom Typ `Map.Entry<K,V>`.
  - Zugriff auf Schlüssel eines Eintrags: `K getKey()`
  - Zugriff auf Daten eines Eintrags: `V getValue()`

Hinweis: Beim Typ `Map.Entry<K, V>` handelt es sich um eine innere Klasse. Diese besprechen wir im Anschluss.

2.1 Hashen von Objekten

Bis jetzt haben wir als Schlüssel nur ganzzahlige Werte betrachtet. Bei der generischen Implementierung sind die 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`.[1]

Für die Implementierung ergibt sich damit folgende Herangehensweise:

```
class HashMap<K, V> implements Map<K, V> { ...

  // Berechnet Hashwert fuer Schluessel-Objekt key
  private int hash( K key ) {
    return Math.abs(key.hashCode() % table.length);
```

[1] Da jedes Objekt vom Typ `Object` in Java ist, steht diese Methode für all Objekte zur Verfügung.

```
  }
  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;
  }
}
```

Soll eine Klasse als Typ eines Schlüssels in einer HashMap verwendet werden, so muss die `equals()` konsistent mit der `hashCode()` Methode sein: Gleiche Objekte müssen den gleichen Hashcode haben! Andernfalls könnten gleiche Objekte an verschiedenen Indizes in der Hashtabelle abgelegt werden und dann nicht mehr über ihren Hashcode gefunden werden. Die Implementierungen von `equals()` und `hashCode()` müssen also zueinander passen. Dies ist bei der Standardimplementierung aus der Klasse `Object` sichergestellt. Näheres dazu in Abschnitt 3.2.

3 Vergleichen von Objekten

Beim Einfügen von Elementen in Suchbäume werden die jeweiligen Schlüssel der Datensätze miteinander verglichen. Bei ganzen Zahlen kann man dazu z.B. ihrer natürlichen Ordnung verwenden. Um beliebige Schlüssel-Objekte miteinander zu vergleichen, muss eine *Ordnung* auf ihnen definiert werden. Java stellt für Vergleiche von Objekten das `Comparator` Interface bereit.

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

Die Methode `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 $x < y$

(Man verschiebt quasi $<$, analog für die anderen Vergleichsoperatoren.)

Beispiel Eine Anwendung benötigt manchmal eine andere als die “natürliche” Ordnung. So kann man Strings nicht nur lexikographisch ordnen, sondern auch bezüglich ihrer Länge miteinander vergleichen. Um Strings zuerst der Länge nach zu vergleichen und nur bei gleicher Länge die lexikographische Ordnung zu verwenden, kann man folgenden `Comparator` für Strings implementieren:

```
class SizeOrder implements Comparator<String> {
  public SizeOrder () {}
  public int compare (String x, String y) {
    if (x.length() < y.length())
      return -1;
    if (x.length() > y.length())
      return 1;
    //lexikalische Ordnung der String-Klasse
    return x.compareTo(y);
  }
}
```

Tests:[2]

```
assertTrue(new SizeOrder().compare("two","three") < 0);
assertTrue("two".compareTo("three") > 0);
```

Beispiel Um `Date`-Objekte chronologisch zu ordnen, kann man folgenden `DateComparator` verwenden:

```
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;
  }
}
```

3.1 `equals()` und `compare()`

Bei der Implementierung von `Comparator` ist typischerweise gefordert, dass der Vergleich konsistent mit `equals` ist, da andernfalls bei Verwendung mit Collections wie `SortedSet` or `SortedMap` Fehler enstehen.

`compare (x,y) == 0` genau dann, wenn `x.equals(y)` den Wert `true` liefert

[2] `assertTrue` testet, ob ein boolscher Ausdruck zu `true` auswertet.

Für `Date`-Objekte ist diese Konsistenz noch nicht gegeben:

```
Date x = new Date(24,12,2016);
Date y = new Date(24,12,2016);
DateComparator dc = new DateComparator();
System.out.println(dc.compare(x,y)); // true
System.out.println(x.equals(y));     // false
```

Wie kann man die gewünschte Konsistenz erreichen? Die `equals()` Methode wird von der Klasse `Object` geerbt und kann bzw. muss bisweilen überschrieben und angepasst werden.
Bei der Implementierung von `equals()` sollten folgende Eigenschaften sichergestellt werden:

- *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.

Wir können ein Anpassung von `equals` in Klasse `Date` in folgender Form vornehmen:

```
1  public class Date {
2      ...
3      public boolean equals(Object obj) {
4        if (this == obj) {
5           return true;
6        }
7        if (obj == null) {
8           return false;
9        }
10       if (!(obj instanceof Date )) {
11          return false; // andere Klasse
12       }
13       Date other = (Date) obj;
14       return this.year == other.year
15         && this.month == other.month
16         && this.day == other.day;
17     }
18 }
19
```

Java bietet den Operator `instanceof` an, um den dynamischen Typen eines Objekts zu testen. Dieser Typtest wird insbesondere benötigt, wenn man eine Typkonvertierung zwischen Referenztypen durchführen will (siehe Zeile 13).

3.2 `equals()` und `hashCode()`

Auf Grund des Anpassens der `equals()` Methode in der Date-Klasse muss nun auch die `hashCode()` Methode angepasst werden, da gleiche Objekte einen gleichen Hashcode haben müssen.

```
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
```

Eine Möglichkeit wäre es als Hashcode die Summe der Attribute zu wählen:

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

Mit dieser Variante ist sichergestellt, dass gleiche Date-Objekte auch den gleichen Hashcode haben. Allerdings führt diese Methode dazu, dass viele Date-Objekte auf den gleichen Hashcode abgebildet werden; z.B. `new Date(24,12,2016)`, `new Date(23,11,2016)`, `new Date(23,12,2017)`}. Um eine gute Streuung zu erhalten, werden daher normalerweise die einzelnen Komponenten, die bei der Berechnung des Hashwertes einfließen, mit einer Primzahl multipliziert und erst dann aufaddiert. Die Implementierung einer `hashCode()` Methode, die die Ergebnisse optimal streut, ist nicht einfach. Näheres dazu erfahren Sie in der Vorlesung EAA bzw. "Algorithmen und Datenstrukturen".
Java stellt in der `Objects`-Klasse eine Methode `hash` bereit, die dies mit einer geeigneten Primzahlen durchführt[3].
Die Methode `hash` nimmt eine beliebig Anzahl an Parametern. Damit können wir die Date-Klasse wie folgt ergänzen:

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

[3]Im Detail:

```
public int hash(...) {
  ...
  for (Object element : a) // a enthaelt die Attributwerte
    result = 31 * result + (element == null ? 0 : element.hashCode());
  }
```

4 Zusammenfassende Übersicht: Java 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 Verwendungen von Collections basieren auf folgenden Operationen:

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

Diese wollen wir anhand der folgenden Fallstudie beispielhaft betrachten:

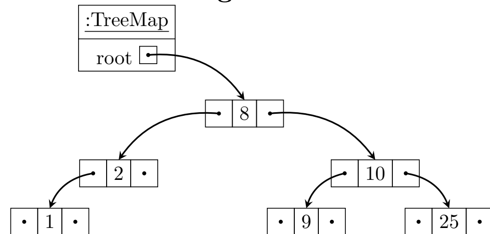
> Um nie wieder den Geburtstag von Verwandten und Freunden zu vergessen, wurde in den Tagen vor Facebook & Co, in vielen Familien Geburtstagslisten gepflegt.
>
> Wir wollen eine Geburtstagsliste implementieren, die jedem Namen den Geburtstag zuordnet. Folgende Operationen sollen unterstützt werden:
>
> - Die Methode `getBirthdate` soll für einen Namen das entsprechende Geburtsdatum liefern.
> - Die Methode `getByMonth` liefert die Namen aller Geburtstagskinder pro Monat.
> - Die Methode `getOldest` liefert den Namen der ältesten Person.

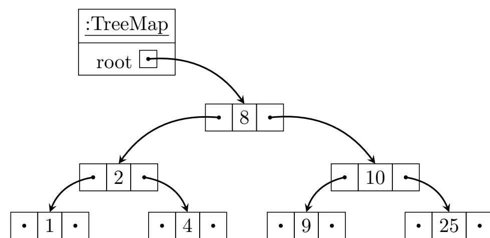
Eine kommentierte Implementierung finden Sie auf der Homepage der Vorlesung zum Download.

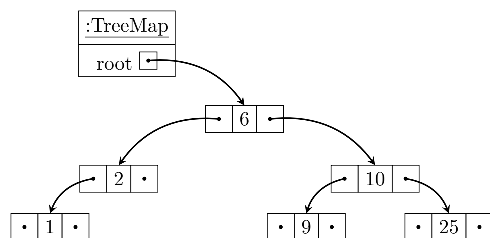
Hinweise zu den Fragen

Hinweise zu Frage 1: Wenn ein Element gesucht wird, das der Iterator erst als letztes Element bei der Iteration liefert. Die Reihenfolge, in der der Iterator die Element besucht, hängt von der Implementierung von ArrayList und des Iterators ab.

Hinweise zu Frage 2: Nach dem beschriebenen Schema:







Hinweise zu Frage 3: Wenn alle Schlüssel auf den gleichen Hashwert abgebildet werden, enthält die Hashtabelle eine Liste mit allen Einträgen. Das Suchen in dieser Liste benötigt bis zu n Schlüsselvergleiche, wenn die Liste n Einträge hat.