

# DAP2 Praktikum – Blatt 8

Abgabe: ab 30. Mai

## Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- Ansonsten kann die Testierung **nur in der zugeteilten Gruppe** garantiert werden.
- Bitte bereiten Sie den Tester sowie in den Aufgaben angegebene Beispieltests vor, bevor Sie sich testieren lassen!

## Wichtige Information (im Moodle verfügbar)

- Beachten Sie die Erklärung des **Ablaufs (Blatt A)**.
- Beachten Sie die **Regeln und Hinweise (Blatt R)** in der aktuellsten Version!
- Beachten Sie die **Hilfestellungen (Blatt H)** in der aktuellsten Version!

## Languaufgabe 8.1: Hashing

(8 Punkte)

In dieser Aufgabe sollen Sie eine simple Hashtabelle implementieren. Zunächst können Sie davon ausgehen, dass die Schlüssel vom Typ `Integer` sind. In Aufgabe 8.2 verallgemeinern Sie die Implementierung für beliebige Datentypen, und in Aufgabe 8.3 für beliebige Hashfunktionen.

Legen Sie eine Datei `SimpleHT.java` **ohne main-Methode** an. Zum Testen Ihrer Implementierung können Sie sich einen *eigenen* Tester in Form einer Testklasse schreiben.

- Ein Object vom Typ `SimpleHT` verwaltet eine Menge von Schlüssel-Wert-Paaren, wobei Schlüssel und Werte vom Typ `Integer` sind. Legen Sie in `SimpleHT` eine public innere Klasse `Pair` an, sodass ein `Pair`-Objekt genau ein Schlüssel-Wert-Paar beinhaltet.
- Die Hashtabelle ist als Array (oder `ArrayList`) mit  $m$  Einträgen organisiert, wobei jeder Eintrag eine verkettete Liste von `Pair`-Objekten enthält. (Die verketteten Listen dienen der Kollisionsauflösung, und die Hashtabelle soll so funktionieren, wie in der Vorlesung beschrieben.) Der Kontruktor von `SimpleHT` erwartet  $m$  als einzigen Parameter.
- Erstmal können Sie einfach die naive Hashfunktion ( $\text{key} \bmod m$ ) verwenden. Diese Funktion ist sehr schlecht: Stammen die Schlüssel etwa aus dem Universum  $\{m \cdot k \mid k \in \mathbb{N}\}$ , dann werden alle Paare in der gleichen verketteten List abgelegt. In Aufgabe 8.3 werden Sie die Implementierung für beliebige Hashfunktionen verallgemeinern. Deswegen sollten Sie bereits jetzt eine public Hilfsfunktion anlegen, welche aus dem Schlüssel die Adresse der verketteten Liste bestimmt:

```
public int addressOf(Integer key)           // welche Liste ist zuständig?
```

**Hinweis:** Der Operator `%` ist in Java streng genommen nicht der Modulo-Operator und kann ein negatives Ergebnis zurückgeben.

- Die Grundoperationen der Hashtabelle sind Einfügen, Auslesen, und Entfernen. Wenn beim Einfügen bereits ein Paar mit gleichem Schlüssel existiert, wird der Wert einfach überschrieben. Wenn beim Auslesen kein Paar mit dem gewünschten Schlüssel existiert, wird `null` zurückgegeben. Wenn beim Löschen kein Paar mit dem gewünschten Schlüssel existiert, wird `false` zurückgegeben (und ansonsten `true`).

```
public void insert(Integer key, Integer value) // füge Paar ein
```

```
public Integer get(Integer key)                // gebe Wert zurück
```

```
public boolean remove(Integer key)            // entferne Paar
```

Bestimmen Sie die Adresse der zuständigen Liste immer mit `addressOf`. Abgesehen vom Konstruktor und den Grundoperationen hat die Klasse **keine öffentlichen Methoden oder Attribute!**

### Geforderte Klassen und Methoden

```
public class SimpleHT{
    public SimpleHT(int capacity) {...}
    public class Pair{
        Integer key
        Integer value
    }
    public int size
    public ArrayList<ArrayList<Pair>> data;
    public int addressOf(Integer key) {...}
```

```
public void insert(Integer key, Integer value) {...}
public Integer get(Integer key) {...}
public boolean remove(Integer key) {...}
}
```

**Hinweis:** Sie dürfen die Bibliotheken `ArrayList`, `LinkedList`, `List`, und `ListIterator` aus `java.util`, sowie die Methode `java.lang.Math.floorMod` verwenden.

**Hinweis:** Sie sollen als Datentyp der Schlüssel und Werte wirklich die Klasse `Integer` verwenden, und nicht den primitiven Typen `int`. Mit der Wrapper-Klasse haben Sie zahlreiche Vorteile, z.B. dass Sie hilfreiche Methoden auf `Integer`-Objekten aufrufen können (beispielsweise `toString()`), dass Sie Referenzen auf `Integer`-Objekte an Methoden übergeben können, und insbesondere dass Sie die Wrapper-Klasse mit generische Klassen und Methoden verwenden können. Dies wird wichtig für Aufgabe 8.2 sein. Weitere Informationen zu den Wrapper-Klassen der primitiven Typen finden sie z.B. hier:

[https://www.w3schools.com/java/java\\_wrapper\\_classes.asp](https://www.w3schools.com/java/java_wrapper_classes.asp)

## Languaufgabe 8.2: Generische Klassen

(6 Punkte)

In dieser Aufgabe verallgemeinern Sie ihre Implementierung aus Aufgabe 8.1, sodass Sie nicht nur Schlüssel und Werte vom Typ `Integer` verwenden. Dazu machen Sie die Klasse `SimpleHT` zur generischen Klasse `GenHT<K, V>`, in welcher Schlüssel vom Typ `K` und Werte vom Typ `V` sind (hier sind die Namen von `K` und `V` eine Namenskonvention für Schlüssel-Wert-Paare, englisch *Key-Value-Pairs*). Die generischen Typen `K` und `V` sind Parameter, die bei der Instanziierung der Klasse durch beliebige Typen ersetzt werden können.

Natürlich können Sie jetzt nicht mehr einfach  $(\text{key} \bmod m)$  als Hashfunktion verwenden, denn der Modulo-Operator ist nicht für alle möglichen Schlüsseltypen `K` definiert. Verwenden Sie stattdessen erstmal  $(\text{key.hashCode()} \bmod m)$ . In Java erbt jede Klasse von der Klasse `Object`, und die Klasse `Object` stellt die Methode `hashCode` bereit. Die Funktionalität ist wie folgt:

- Der Aufruf von `hashCode()` gibt einen `int`-Wert zurück. Wird die Methode wiederholt für das gleiche Objekt aufgerufen, wird jedes mal der gleiche Wert zurück gegeben.
- Zwei Objekte `a` und `b` mit `a.equals(b)==true` müssen den gleichen `hashCode` haben.
- Zwei Objekte `a` und `b` mit `a.equals(b)==false` dürfen den gleichen `hashCode` haben.

Viele Klassen überschreiben die `hashCode`-Funktion. Informieren Sie sich im Internet darüber, wie `hashCode` für die Wrapper-Typen der primitiven Klassen funktioniert. Handelt es sich dabei um "gute" Hashfunktionen?

Zum Testen Ihrer Implementierung können Sie sich wieder eine *eigene* Testklasse schreiben.

**Hinweis:** Java Generics kennen Sie hoffentlich bereits aus DAP1. Ansonsten finden Sie zahlreiche gute Tutorials und Beispiele im Internet, z.B.:

<https://www.geeksforgeeks.org/generics-in-java>  
<https://www.w3schools.blog/generics-class-java>

## Geforderte Klassen und Methoden

```
interface SimpleHashFunction<K>{
    public int getHash(K key, int m);
}
public class GenHT<K, V>{
    public GenHT(int capacity) {...}
    public class Pair{
        K key;
        V value;
    }
    public int size;
    public ArrayList<ArrayList<Pair>> data;
    public SimpleHashFunction<K> hashFunction;
    public int addressOf(K key) {...}
    public void insert(K key, V value) {...}
    public V get(K key) {...}
    public boolean remove(key) {...}
}
```

### Languaufgabe 8.3: Hashfunktion als Parameter

(2 Punkte)

Ihre Hashtabelle aus Aufgabe 8.2 funktioniert für beliebige Klassen, verwendet jedoch die oftmals schlechte Hashfunktion (`key.hashCode() mod m`). Besser wäre es, wenn Sie dem Konstruktor der Hashtabelle eine beliebige Hashfunktion  $h : K, \text{int} \rightarrow \text{int}$  als Parameter übergeben könnten, und dann  $h(\text{key}, m)$  zum Adressieren der Listen verwenden. Dazu gehen Sie wie folgt vor:

- Das Interface `SimpleHashFunction<K>` definiert `getHash` als einzige Methode.
- Der Konstruktor von `ParamHT<K, V>` bekommt die Hashfunktion  $h$  in einem zweiten, optionalen Parameter vom Typ `SimpleHashFunction<K>` übergeben. Der ursprüngliche Konstruktor mit einem Parameter verwendet weiterhin `hashCode` als Hashfunktion. Implementieren Sie den Konstruktor mit einem Parameter mithilfe einer Konstruktor-Delegation. (Sie rufen also den Konstruktor mit zwei Parametern aus dem Konstruktor mit einem Parameter auf.)
- Beachten Sie, dass die übergebene Hashfunktion nun Werte außerhalb der Kapazität zurückgeben kann und fangen Sie diese Fehler ab! Hierzu soll die Funktion `addressOf` bei hash eine Exception werfen. Diese soll jedoch von den Funktionen, die `addressOf` aufrufen, abgefangen werden.
- Der restliche Code kann aus Aufgabe 2 übernommen werden.

Unten sehen Sie beispielhaft, wie das Interface verwendet werden soll.

---

```
// Hashfunktion 1: Gibt 1 zurueck, wenn Key 0 ist, und sonst 0.
class WorstHashEver<K> implements SimpleHashFunction<K> {
    public int getHash(K key, int m) {
        return (key.equals(0)) ? 1 : 0;
    }
}

var h = new ParamHT<Double, Float>(1337, new WorstHashEver<Double>());

// Hashfunktion 2: Wandelt String in Integer um, oder gibt 0 zurueck.
class WorstStringHashEver implements SimpleHashFunction<String> {
    public int getHash(String key, int m) {
        try { return Integer.parseInt(key); }
        catch (Exception e) { return 0; }
    }
}

var g = new ParamHT<String, Object>(1337, new WorstStringHashEver());
```

---

### Geforderte Klassen und Methoden

```
interface SimpleHashFunction<K>{
    public int getHash(K key, int m);
}

public class ParamHT<K, V>{
    public ParamHT(int capacity) {...}
    public ParamHT(int capacity, SimpleHashFunction<K> hashFunction) {...}
    public class Pair{
        K key;
        V value;
    }
    public int size;
    public ArrayList<ArrayList<Pair>> data;
    public SimpleHashFunction<K> hashFunction;
    public int addressOf(K key) throws IndexOutOfBoundsException {...}
    public void insert(K key, V value) {...}
    public V get(K key) {...}
    public boolean remove(key) {...}
}
```

---