# Lösungen Arbeitsblatt: Hash Tables

### Aufgabe 1 TreeMap vy HashMap

Studieren Sie die Java-Dokumentation von TreeMap und HashMap und beantworten Sie die folgenden Fragen. Worin unterscheiden sie sich hauptsächlich? Warum ist eine HashMap für gewisse Elemente einfacher zu nutzen als eine TreeMap? Was ist der asymptotische Aufwand der Operation put() im best und im worst case?

Ein wichtiger Unterschied liegt darin, dass TreeMaps eine Ordnung der Elemente in ihrer natürlicher Reihenfolge garantiert, eine Hashmap hingegen keine Ordnung besitzt, was bedeutet, dass die Schlüssel bei Iteration in beliebiger Reihenfolge ausgegeben werden.

Komplexität von put() in HashMap: O(1) im Best/Average Case, O(n) bei Kollision im worst case in TreeMap O(log n) sowohl im Best Case wie auch Worst Case.

 

**Aufgabe 2 MyInteger**

Gegeben sei die folgende, einfache Klasse `MyInteger`:

```
public class MyInteger {
    private int i;

        public MyInteger(int i) {
        this.i = i;
    }
}
```

Bei eigenen Datentypen müssen die Methoden `hashCode()` und `equals(Object o)` überschrieben werden. Beschreiben Sie, welche Fehler im Zusammenhang mit Hashing auftreten können, wenn eine der beiden Methoden nicht überschrieben oder falsch implementiert wird:

`equals(Object o)` nicht (korrekt) überschrieben:

Wird `equals()` nicht überschrieben, so führt Folgendes zur Ausgabe `false`, obwohl die von den Objekten repräsentierten Werte logisch gleich sind:

```
MyInteger a = new MyInteger(7), b = new MyInteger(7);
System.out.println(a.equals(b));
```

Dies passiert, weil die equals Funktion, wenn nicht implementiert, defaultmässig Referenzen vergleicht und nicht Werte. Fügt man den Wert der Variablen a in eine Datenstruktur ein und sucht dann nach dem (eigentlich gleichen) Wert b (z.B mit der `contains(Object)`-Methode), wird man deshalb nicht fündig.

**Kurz:** Der HashWert zeigt zwar aufs richtige Feld, jedoch ist die Suche nach dem Objekt erfolglos oder liefert das falsche Objekt zurück.

`hashCode()` nicht (korrekt) überschrieben:

Für die Default-Implementierung der Hash-Wert-Berechnung lässt die Spezifikation etwas Freiheit. Eine Möglichkeit ist die Adresse des Objektes zu verwenden (jedenfalls solange das Objekt nicht im Speicher verschoben wird, z.B. durch einen kompaktierenden Garbage Collector). Damit wird erreicht, dass verschiedene Objekte unterschiedliche Hash-Werte haben.

Wird zwar `equals()` aber nicht `hashCode()` überschrieben, erreicht man mit hoher Wahrscheinlichkeit, dass zwei an sich (logisch) gleiche Objekte verschiedene Hash-Werte haben und deswegen in einer Hash-Tabelle nicht am gleichen Platz gesucht würden. Deswegen muss man zusammen mit der Methode `equals()` immer auch `hashCode()` selbst implementieren.

**Kurz:** Zwei logisch gleiche Objekte haben unterschiedliche HashWerte und werden darum in unterschiedlichen Feldern gesucht oder abgelegt.

Machen Sie einen Vorschlag zur Implementation der beiden Methoden für die Klasse `MyInteger`:

```
@Override
public boolean equals(Object o) {
  return o != null &&
        getClass() == o.getClass() &&
        ((MyInteger)o).i == i;
}
```

```
@Override
public int hashCode() {
   return i;
}
```

**Aufgabe 3 Class Person I**

Die FHNW möchte von Eventoweb auf ein neues System umsteigen. Dazu müssen für alle Studierende die Gesamtzahl ECTS exportiert werden. Da sich für Zuordnungen Map-Strukturen gut eigen, schlagen Sie die Verwendung einer `HashMap` vor. (Auf einen Export ECTS / Note pro Modul verzichten wir einfachheitshalber einmal.)

Definieren Sie in Java eine Klasse `Person`, so dass diese korrekt funktionierend in einer `HashMap` als `Key` verwendet werden kann. Die Klasse `Person` soll die folgenden Variablen beinhalten, welche als Kombination eindeutig sind:

```
String firstname, lastname;
int birthyear, birthmonth, birthday, age;
```

Teil des (unveränderlichen) Schlüssels sollten alle Attribute ausser **age** sein, das sich jährlich ändert.

```
public class Person {
    public final String firstname, lastname;
    public final int birthyear, birthmonth, birthday;
    public int age;

    public Person(String first, String last, int year, int month, int day) {
        this.firstname = first;
        this.lastname = last;
        this.birthyear = year;
        this.birthmonth = month;
        this.birthday = day;
        age = (int)ChronoUnit.YEARS.between(LocalDate.of(year, month, day), Local-
Date.now());
    }

    @Override
    public int hashCode() {
        int result = birthday;
        result = 31 * result + birthmonth;
        result = 31 * result + birthyear;
        result = 31 * result + firstname.hashCode();
        result = 31 * result + lastname.hashCode();
        return result;
    }

    @Override
    public boolean equals(Object other) {
        if (other != null && getClass() == other.getClass()) {
            Person o = (Person)other;
            boolean result = firstname.equals(o.firstname);
            result = result && lastname.equals(o.lastname);
            result = result && birthyear == o.birthyear;
            result = result && birthmonth == o.birthmonth;
            result = result && birthday == o.birthday;
            return result;
        } else return false;
    }
}
```

Die meisten IDEs enthalten auch Generatoren für solche Methoden. Vergleichen Sie die Ergebnisse!

**Aufgabe 4 Class Person II**

Sie haben in der Klasse String gesehen, dass der Hashwert zwischengespeichert wird, so dass dieser nur einmal berechnet werden muss. Passen Sie Ihre Klasse `Person` an, so dass auch dort der Hashwert nur einmalig berechnet werden muss. Sehen Sie mögliche Schwierigkeiten, die bei zwischengespeicherten Hashwerten auftreten können? Falls ja, machen Sie einen Lösungsvorschlag.

Mögliches Problem: Wenn sich Werte ändern, stimmt der gespeicherte Hash nicht mehr. Das ist auch sonst ein Problem, da dann die Objekte nicht mehr gefunden werden. Deshalb ist empfehlenswert, die Hashabhängigen Variablen als final zu deklarieren.

```
public final String firstname, lastname;
public final int birthyear, birthmonth, birthday;
public int age;
private int hash;
...

@Override
public int hashCode() {
   int h = hash;
   if (h == 0) {
        h = birthday;
        h = 31 * h + birthmonth;
        h = 31 * h + birthyear;
        h = 31 * h + firstname.hashCode();
        h = 31 * h + lastname.hashCode();
        hash = h;
      }
      return h;
}
```

**Aufgabe 5 polygenelubricants**

Sie sehen folgende `HashMap`-Implementation vor sich. Die Methode `indexOf(K key)` wird verwendet, um die Position innerhalb des Arrays zu berechnen:

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

    Entry<K,V>[] table;

    private int indexOf(K key){
        return key.hashCode() % table.length;
    }

    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        Entry<K,V> next;
        ...
    }
}
```

Für negative Werte von `a` ist `a % b` möglicherweise ebenfalls negativ[1]. Negative Werte sind aber keine gültigen Array-Indizes. Die Methode `indexOf()` muss deshalb für beliebige `int`-Werte ein Ergebnis `r` im Bereich $0 \le r < \texttt{table.length}$ berechnen.

Machen Sie einen Korrekturvorschlag und testen sie diesen mit dem String „*polygenelubricants*“.

Es gibt verschiedene Möglichkeiten, das zu erreichen. Dabei sollte die Rechenzeit als Kriterium berücksichtigt werden.

Im Folgenden zwei mögliche Varianten:

a) Korrektur des Vorzeichens ***nach*** der Bereichs-Reduktion mit %

```
private int indexOf(K key) {
     int h = key.hashCode() % table.length;
     return h >= 0 ? h : -h;
}
```

Vorzeichenkorrektur des originalen Hash-Wertes *vor* der %-Operation wäre keine korrekte Lösung, weil das für `Integer.MIN_VALUE` nicht funktioniert. `Integer.MIN_VALUE` als HashWert kann aber durchaus vorkommen, wie der String «*polygenelubricants*» zeigt.

b) Ausmaskieren des Vorzeichens ***vor*** der Bereichs-Reduktion mit %

```
private int indexOf(K key) {
  return (key.hashCode() & 0x7FFFFFFF) % table.length;
}
```

Negative `int`-Werte kann man durch ausmaskieren des Vorzeichen-Bits (mittels `& 0x7FFFFFFF` bzw. `& Integer.Max_VALUE`) in einen nicht-negativen Wert verwandeln. Dabei ändert sich aber evtl. auch der Betrag, weswegen man diese Operation vor der Reduktion auf den Index-Bereich machen muss.

In Java HashMap hat die Map immer eine Grösse einer 2er Potenz. Dann wird de facto der hashcode & (length -1) gerechnet.

[1] Es gilt für alle *int a, b*: $0 \le a \Rightarrow 0 \le a \,\%\, b < |b|$ und $0 \ge a \Rightarrow 0 \ge a \,\%\, b > -|b|$, wobei $|b|$ für den Betrag von b steht. Grund ist, dass dann immer gilt: $a = (a / b) * b + a \,\%\, b$ und $|(a / b) * b| \le |a|$ – in Worten: Die Ganzzahldivision rundet gegen 0 und der Modulo-Operator berechnet den entsprechenden Divisionsrest.