Tips und Tricks zum Hashing

1. Hashfunktion

Die Hashfunktion sollte folgende Eigenschaften haben:

- *effizient berechenbar*
  z.B. Anwendung der Horner-Regel eliminiert eine Multiplikation in der for-Schleife
  z.B. bei langen Strings nur jeder n-te Buchstabe in Berechnung verwenden.
- *möglichst zufällig verteilte Schlüssel*
  Achtung: Viele Daten machen einen zufälligen Eindruck, sind es aber nicht!
  z.B. AHV-Nummer als int interpretiert: xxx.yy.qdd.zzz
  xxx: Familiencode ist schon eine Art Hashcode, denn es gibt mehr als 1000 verschiede Namen
  yy: Jahrgang nicht gleichverteilt (Alterspyramide, „Pillenknick“)
  q: Quartal 1-4 männlich, 5-8 weiblich, 9 kommt nicht vor
  dd: Tag 1. Monat: $dd \leq 31$, 2. Monat: $31 < dd \leq 62$, 3. Monat: $62 < dd \leq 93$
  zzz: Zusatznummer Fortlaufende Nummer (Nationalität) + Prüfziffer
- *alle Daten werden verwendet:*
  Bsp.: Hashcode für eine Personaldatei. Man erhält bessere Werte, wenn alle Daten berücksichtigt werden (Name, Strasse, PLZ, Geburtsdatum usw.), als wenn nur der Name verrechnet wird.
- *Nur „echte“ Daten berücksichtigen.*
  Bsp.: Ort und Postleitzahl sind redundant. Der Hashcode wird nicht besser, wenn beide Daten verrechnet werden, seine Berechnung verlangsamt sich aber.

2. Vergleich der Strategien zur Kollisionsbehebung

	separate chaining	**linear probing**	**quadratic probing**	**double hashing**
Bedingung für Tabellengrösse	keine (verteilt besser falls prim)	keine (verteilt besser falls prim)	prim, so gross, dass load factor < 0.5	prim
Duplikate	einfach	aufwendig	aufwendig	aufwendig
load factor max.	> 1	1	< 0.5	1
load factor opt.	< 1	< 0.75	< 0.5	< 0.9

3. Hashing in Java

Die Hashfunktion:

In der Klasse java.lang.Object ist eine Methode hashCode definiert.

```
public int hashCode();
```

Damit kann man sich von jedem Objekt einen Hashcode geben lassen. Falls diese Methode nicht überschrieben wird, liefert sie eine int-Darstellung der Speicheradresse des Objektes zurück. Diese Hashfunktion kann verwendet werden, ist aber nicht für alle Daten optimal. z.B. werden Objekte häufig im Speicher auf eine „gerade“ Adresse (Mehrfaches von 2, 4 oder 8, je nach Wortgrösse des Prozessors) aligniert. Wenn man also weiss, dass ein Objekt in einer Hashtabelle abgelegt wird, dann sollte hashCode() überschrieben werden.
Bei einigen Klassen der JFC ist die Methode hashCode() schon überschrieben (siehe API- Doku). z.B. bei String und allen Wrapperklassen (Integer, Long, Double, Float...).

Wichtig für die Implementation von hashCode( ):

- Schreiben Sie eine Funktion, die möglichst gleichmässig über den int-Bereich verteilt.
- Auf keinen Fall sollte das Ergebnis schon in einen bestimmten Bereich transformiert werden. Die Hashtabellengrösse ist ja nicht bekannt!
- Wird hashCode() mehrmals hintereinander auf einem unveränderten Objekt aufgerufen, so *muss* das Ergebnis immer dasselbe sein.
- equals(Object) und hashCode() sollten immer beide zusammen überschrieben werden.
- Sind zwei Objekte gleich (gemäss equals), dann müssen auch die Resultate ihrer hashCodes gleich sein.
- Liefern zwei Objekte den gleichen hashCode, dann sind sie aber nicht notwendigerweise gleich bezüglich equals.

Wird bei einer Klasse die Methode hashCode überschrieben, so werden üblicherweise die Hashcodes der einzelnen Attribute zu einem neuen Hashcode verarbeitet. Falls die Klasse eine andere Klasse erweitert, welche bereits die Methode hashCode überschrieben hat, so ist diese mit einem Supercall aufzurufen.

```
public int hashCode( ) {

    //int code = super.hashCode();     // falls hashCode in Basisklasse bereits überschrieben ist
    int code = 0;                      // (entweder oder)

    code = code * 37 + x;                          // für ein Attribut x vom Typ byte/char/int
    code = code * 37 + (b?1:0);                    // für ein Attribut b vom Typ boolean
    code = code * 37 + obj.hashCode();             // für ein Attribut obj != null vom Typ Object
    for(int i=0; i<a.length; i++)                  // für ein Feld a von Objekten
        code = code * 37 + a[i].hashCode();
}
```

Bsp:

```
class NameKey {
    String name, prename;
    public boolean equals(Object x){
        return name.equals(((NameKey)x).name) && prename.equals(((NameKey)x).prename);
    }
    public int hashCode(){
        return name.hashCode() * 1000003 + prename.hashCode();
    }
}
```

Implementationen von Hashtabellen

Eine Implementation von Hashtabellen steht Ihnen mit den Klassen java.util.HashSet und java.util.HashMap zur Verfügung (siehe Java API-Doku). Beide Klassen verwenden *separate chaining* zur Kollisionsbehebung.
Bsp.: java.util.HashMap, verwaltet (key, value)-Tupel.

```
public Object put(Object key, Object value); // a[key] = value;
```

Lässt sich den Hashcode von key berechnen und speichert dann value an die berechnete Stelle. Am einfachsten ist, man stellt sich eine java.util.HashMap als Array vor, der ein Object als Index haben kann.

4. Einige Begriffe:

im Englischen:

- ***Open Addressing,*** *gilt als* ***„closed implementation“***
  darunter versteht man die Kollisionsbehebungsstrategien *linear probing*, *quadratic probing* und *double hashing*
- ***Separate Chaining*** *gilt als* ***„open implementation“***
  Ein Eintrag in der Hashtabelle (der Kopf einer Liste) wird häufig auch als ***Bucket*** bezeichnet.
- ***Perfect Hashfunction***
  Eine Hashfunktion heisst *perfekt*, wenn Sie für jedes Objekt einen eindeutigen Wert liefert. Anders ausgedrückt, sind zwei Objekte verschieden, dann sind auch ihre Hashcodes unterschiedlich.

deutsche Übersetzungen:

hashing	Streuspeicherverfahren, Schlüsseltransformation
hash function	Transformationsfunktion
open addressing	offene Adressierung
linear/quadratic probing	lineares/quadratisches Sondieren
primary/secondary clustering	primäre/sekundäre Ballung
separate chaining	direkte Verkettung