Arbeitsblatt: Hash Tables – Lösungen

1. Insgesamt gibt es m^n Möglichkeiten, *n* Elemente auf *m* Plätze zu verteilen. Davon gibt es aber nur in $m \cdot (m-1) \cdot \cdots \cdot \big(m-(n-1)\big)$ Fällen keine Kollision.

   Also: Wahrscheinlichkeit, dass man in ein Array der Länge *m* insgesamt *n* Elemente ohne Kollision einfügen kann:

   $$\frac{\mathrm{m} \cdot (\mathrm{m}-1) \cdot \cdots \cdot \big(\mathrm{m}-(\mathrm{n}-1)\big)}{\mathrm{m}^{\mathrm{n}}} = \frac{\mathrm{m}!}{\mathrm{m}^{\mathrm{n}} \cdot (\mathrm{m}-\mathrm{n})!}$$

   Konkret für *m* = 13 und *n* = 7 bedeutet das:

   $$\frac{13!}{13^7 \cdot (13-7)!} \approx 0.138$$

   Die Wahrscheinlichkeit, dass mindestens eine Kollision auftritt, ist also ca. $1 - 0.138 = 0.862 = 86\%$.

   Analog lässt sich die Frage in der Fussnote (Geburtstagsparadoxon) beantworten. Es sollen 23 Geburtstage auf 365 Kalendertage verteilt werden. Das geht kollisionsfrei mit Wahrscheinlichkeit:

   $$\frac{365!}{365^{23} \cdot (365-23)!} \approx 0.493$$

   Die Wahrscheinlichkeit, dass von 23 Personen mindestens zwei am selben Tag Geburtstag haben ist also gerade etwas grösser als ½.

   Bei dem Experiment mit 7 selbstgewählten „Zufallszahlen" ist zu erwarten, dass es zu wenigen Kollisionen kommt (d.h. einzelne Plätze werden mit 2 Zahlen belegt). Belegungen mit 3 Zahlen können selten vorkommen.

2. Die angegebenen Worte „AUS", „USA" und „SAU" bestehen aus denselben Buchstaben, so dass die Hash-Werte alle gleich sein müssen (233). Aber auch die anderen drei Worte in der Tabelle haben denselben Hash-Wert.

3. Es wird ein Polynom in 31 verwendet, mit den Zeichen-Codes als Koeffizienten. Z.B. für „AUS" ergibt sich: $31^2 \cdot 65 + 31 \cdot 85 + 83 = 65183$. Die anderen Hash-Werte sind 84323 für „USA" und 81863 für „SAU". Obwohl die Wörter alle aus den gleichen Buchstaben aufgebaut sind, ergeben sich verschiedene Hash-Werte, da jede Position im Wort einem anderen Koeffizienten im Polynom entspricht.

4. Analyse von `String.hashCode()`:

   a. Der Hash-Wert wird nicht jedes Mal neu berechnet, sondern in einer Instanzvariablen *hash* gespeichert. Da String-Werte unveränderlich sind (was Schlüssel in Datenstrukturen ohnehin sein müssen, weil sie sonst nicht mehr gefunden werden können), darf sich auch der Hash-Wert über die Zeit nicht ändern. Deswegen ist diese Optimierung sinnvoll.

   b. Der Hash-Wert wird als Polynom über 31 (eine Primzahl!) berechnet, wobei die einzelnen Zeichen-Codes als Koeffizienten verwendet werden. Dadurch ist der Hash-Wert des Strings nicht nur von den Zeichen-Codes sondern auch den Positionen der Zeichen abhängig.

   c. Das Polynom wird effizient mittels des *Horner-Schemas* berechnet:[1]

   $$a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + a_i x^{n-i} + \cdots + a_n \;=\; \Big(\big((a_0 \cdot x + a_1) \cdot x + a_2\big) \cdot x + \cdots\Big) \cdot x + a_n$$

   d. Die Verwendung einer Primzahl im Polynom führt zu einer guten Verteilung der Werte.

[1] Das *Horner-Schema* erlaubt die schnelle Berechnung von Polynomen, denn es reduziert die Anzahl der (relativ teuren) Multiplikationen um ca. die Hälfte, weil nicht parallel zum Wert des Polynoms auch noch die entsprechenden Potenzen von *x* berechnet werden. – Wenn Sie die angegebene Identität überprüfen möchten, können sie einfach die rechts vom Gleichheitszeichen stehende Formel ausmultiplizieren und sollten das links stehende Polynom erhalten.

5. Für die Methoden `equals()` und `hashCode()` gibt es Default-Implementierungen in der Klasse `Object`. Der Gleichheitstest benutzt einfach den Referenzvergleich: `a.equals(b)` ⇔ `a == b`.

   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));
   ```

   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 nicht fündig. Deswegen muss `equals()` implementiert werden:

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

   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 also mit sehr 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 (s. auch Bedingung Kap. 6.4 Ende).

   Deswegen muss man zusammen mit der Methode `equals()` immer auch `hashCode()` selbst implementieren. Für das Beispiel `MyInteger` am einfachsten so:

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

6. 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 firstname, String lastname, int birthyear,
        int birthmonth, int birthday) {
     this.firstname = firstname; this.lastname = lastname;
     this.birthyear = birthyear; this.birthmonth = birthmonth;
     this.birthday = birthday;
     age = (int)ChronoUnit.YEARS.between(
      LocalDate.of(birthyear, birthmonth, birthday), LocalDate.now());
    }

    @Override
    public int hashCode() {
     return (((birthyear * 12 + birthmonth) * 31 + birthday) * 31
       + firstname.hashCode()) * 31 + lastname.hashCode();
    }

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

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

7.
```
private int hash;
...

@Override
public int hashCode() {
   int h = hash;
   if (h == 0) {
    h = (((birthyear * 12 + birthmonth) * 31 + birthday) * 31
       + firstname.hashCode()) * 31 + lastname.hashCode();
    hash = h;
   }
   return h;
}
```

8. Für negative Werte von `a` ist `a % b` möglicherweise ebenfalls negativ[2]. Negative Werte sind aber keine gültigen Array-Indices.

   Die Methode `indexOf()` muss also für beliebige `int`-Werte ein Ergebnis `r` im Bereich $0 \le r < \texttt{table.length}$ berechnen.

   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 %

      Vorzeichenkorrektur des originalen Hash-Wertes vor der %-Operation wäre keine korrekte Lösung, weil das für `Integer.MIN_VALUE` nicht funktioniert.

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

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

      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.

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

9. Die Array-Grösse ist die erste 2-er-Potenz, die mindestens so gross ist, wie die geforderte Kapazität.

   Es wird also höchstens knapp doppelt so viel Speicher reserviert, wie verlangt wurde und die Länge des Arrays ist immer eine Zweierpotenz. Zweierpotenzen lassen sich deutlich schneller berechnen als Primzahlen.

   Im Binärsystem werden Zweierpotenzen immer mit einer einzigen 1 dargestellt (z.B. 64: 1000000). Zieht man von einer solchen Zahl 1 ab, erhält man ein Bitmuster aus genau so vielen 1, wie es zunächst 0 nach der 1 gab (z.B. 64 – 1 = 63: 111111).

   Verwendet man – wie in `indexFor` – ein solches Bitmuster als Maske für beliebige Zahlen – egal ob positiv oder negativ – erhält man grundsätzlich ein nicht-negatives Ergebnis kleiner als die Zweierpotenz zu der das Bitmuster gehört. Eine einzige bitweise Und-Operation genügt also, um sicherzustellen, dass beliebige Werte des Typ `int` auf einen gültigen Index reduziert werden (z.B. -35789138 & 63: 11111101110111011110011010101110 & 111111 = 101110 entspricht 46).

---

[2] 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.