Hash-Tabellen

Binäres Suchen (in einer sortierten Reihung oder in einem sortierten binären Baum) ist sehr schnell (O(log(n))). Es gibt aber (erstaunlicherweise) Sammlungen, in denen man noch schneller suchen kann, sogenannte *Hash-Tabellen*. In einem bestimmten Sinn spielt *Zufall* (oder: Chaos) bei solchen Sammlungen eine wichtige Rolle. Hash-Tabellen gibt es in verschiedenen Varianten. Hier soll hauptsächlich ihr Grundprinzip anhand einer besonders einfachen Varianten dargestellt werden.

Def.: Eine *Hash-Tabelle* ist eine Reihung von Listen (engl.: A hash table is an array of lists).

Hash-Tabellen kombinieren also "Beton-Reihungen" mit "Gummi-Listen", und sind damit *schnell* (wie Reihungen) und *dehnbar* (wie Listen).

In einer Hash-Tabelle speichert man Objekte ab, die nur aus einem *Schlüssel* (oder aus einem *Schlüssel und irgendwelchen dazugehörigen Daten*) bestehen.

Ein ganz konkretes Beispiel

Angenommen, wir haben eine Hash-Tabelle (d.h. eine Reihung) namens `ht`, die 10 Listen von `String`-Objekten `ht[0]`, `ht[1]`, ..., `ht[9]` enthält. In die Hash-Tabelle `ht` sollen die folgenden 14 Schlüssel (vom Typ `String`) eingefügt werden:

```
"Ali",  "Babsy", "Alfred", "Arno", "Alice",  "Benno", "Kurt",
"Alex", "Angy",  "Bine",   "Max",  "Franz",  "Susi",  "Alf"
```

Wie funktioniert eine Hash-Tabelle allgemein?

Zu jeder Hash-Tabelle gehört eine sogenannte *Hash-Funktion*. Deren Aufgabe ist es, jeden möglichen *Schlüssel* auf einen *Index* der Hash-Tabelle (die ja eine *Reihung* ist) abzubilden.

Für unser konkretes Beispiel muss die Vereinbarung einer Hash-Funktion `hash` folgende Form haben:

```
int hash(String s) {...}
```

und das Ergebnis dieser Funktion muss immer zwischen 0 und 9 liegen.

Benutzt wird die Hash-Funktion `hash` wie folgt:

Soll ein Schlüssel `s` in die Hash-Tabelle `ht` *eingefügt* werden, so wird er in die Liste `ht[hash(s)]` eingefügt.

Soll `s` in der Hash-Tabelle `ht` *gesucht* werden ("Ist `s` in `ht` drin?"), so wird er in der Liste `ht[hash(s)]` gesucht.

Soll `s` aus der Hash-Tabelle `ht` gelöscht werden, so wird er aus der Liste `ht[hash(s)]` gelöscht.

Verschiedene Hash-Funktionen für unser konkretes Beispiel

Das "Geheimnis einer Hash-Tabelle" steckt im Wesentlichen in der verwendeten *Hash-Funktion*.

Es gibt sehr viele verschiedene Hash-Funktionen. Um ein Gefühl dafür zu bekommen, was eine Hash-Funktion gut (oder schlecht) macht, betrachten wir im Folgenden verschiedene Hash-Funktionen `hash01`, `hash02`, ... für unser konkretes Beispiel mit den 14 Schlüsseln ("Ali", "Babsy", ...).

Wir beginnen mit einer der garantiert *schlechtesten* Hash-Funktionen die es gibt:

```
1 int hash01(String s) {
2    return 3;
3 }
```

Diese Funktion bildet alle Schlüssel `s` auf den Index 3 ab.

Wenn man diese Funktion verwendet und alle 14 Beispiel-Schlüssel in die Hash-Tabelle `ht` (mit `ht.length` gleich 10) einfügt, sieht `ht` etwa so aus:

```
Index|Elemente der Listen (eingefügt mit hash01)
   0 |
   1 |
   2 |
   3 | Ali Babsy Alfred Arno Alice Benno Kurt Alex Angy Bine Max Franz Susi Alf
   4 |
   5 |
   6 |
   7 |
   8 |
   9 |
Suchschritte insgesamt mit hash01: 105
```

Die letzte Zeile ("Suchschritte ... **105**") bedeutet: Wenn man jeden Schlüssel, der eingefügt wurde, *einmal* sucht, braucht man dazu insgesamt (und "im Wesentlichen") 105 Listen-Suchschritte. Den Schlüssel `"Ali"` findet man nach *einem* Schritt, für `"Babsy"` braucht man *2 Schritte*, für `"Alfred"` *3 Schritte*, ... und für `"Alf"` *14 Schritte*, macht insgesamt 105 Schritte. Diese Zahl ist ein Maß für die Güte (oder Schlechtigkeit) der verwendeten Hash-Funktion (kleinere Werte sind besser als größere).

Wenn man die Funktion `hash01` verwendet, dauert das Einfügen, das Suchen und das Löschen eines Schlüssels etwa so lange wie bei einer *Liste* (hinzu kommt noch je 1 Aufruf der Hash-Funktion).

Aufgabe-01: Beschreiben Sie 9 weitere Hash-Funktionen, die garantiert genauso schlecht sind wie `hash01`.

Die folgende Funktion `hash02` ist schon deutlich besser als `hash01` ("der zweite Zwerg ist deutlich größer als der erste Zwerg, aber keiner der beiden ist ein Riese" :-):

```
1 int hash02(String s) {
2    if (s.charAt(0) % 2 == 0) {
3       return 3;
4    } else {
5       return 4;
6    }
7 }
```

Diese Funktion bildet einige Schlüssel auf den Index 3 und andere auf den Index 4 ab. Die Listen `ht[0]` bis `ht[2]` und `ht[5]` bis `ht[9]` bleiben garantiert leer. Nach dem Einfügen aller 14 Beispiel-Schlüssel sieht die Hash-Tabelle `ht` wie folgt aus:

```
Index|Elemente der Listen (eingefügt mit hash02)
   0 |
   1 |
   2 |
   3 | Babsy Benno Bine Franz
   4 | Ali Alfred Arno Alice Kurt Alex Angy Max Susi Alf
   5 |
   6 |
   7 |
   8 |
   9 |
Suchschritte insgesamt mit hash02: 65
```

An der letzten Zeile ("Suchschritte ... **65**") kann man erkennen, dass die Funktion `hash02` deutlich besser ist als `hash01`.

Die folgende Hash-Funktion ist noch besser als `has02`:

```
1 int hash03(String s) {
2    return s.charAt(0) % LAENGE_DER_HASHTAB;
3 }
```

Hier wird der Index aus dem ersten Zeichen des Schlüssels berechnet. Die Operation `% LAENGE_DER_HASHTAB` stellt sicher, dass wir immer einen gültigen Index der Hash-Tabelle bekommen. Nach dem Einfügen aller 14 Beispiel-Schlüssel sieht die Hash-Tabelle `ht` wie folgt aus:

```
Index|Elemente der Listen (eingefügt mit hash03)
   0 | Franz
   1 |
   2 |
   3 | Susi
   4 |
   5 | Ali Alfred Arno Alice Kurt Alex Angy Alf
   6 | Babsy Benno Bine
   7 | Max
   8 |
   9 |
Suchschritte insgesamt mit hash03: 45
```

Man sieht: Die Funktion `hash03` bewirkt, dass alle Schlüssel mit gleichem Anfangsbuchstaben in dieselbe Liste kommen. Allerdings können in einer Liste auch Schlüssel mit verschiedenen Anfangsbuchstaben stehen (weil z.B. `'A' % 10` gleich `'K' % 10` gleich `5` ist).

Aufgabe-02: Die Funktion `hash03` bewirkt, dass alle mit `'A'` und alle mit `'K'` beginnenden Schlüssel in die Liste `5` kommen. Nennen Sie einen weiteren Anfangsbuchstaben, der von `hash03` auf den Index 5 abgebildet wird.

Hash-Funktionen und Schlüssel

Für ein genaueres Verständnis von Hash-Funktionen besonders wichtig ist die folgende Tatsache: Ob die Funktion `hash03` besonders gut oder ziemlich schlecht oder mittelmäßig ist, kann man nicht allein anhand der *Funktion* selbst entscheiden. Vielmehr muss man auch die *Schlüssel* berücksichtigen, auf die man sie anwendet. Für die 14 Beispiel-Schlüssel ist `hash03` nicht besonders gut, denn sie lässt 5 der 10 Listen leer und bewirkt, dass 8 der 14 Schlüssel in dieselbe Liste (`ht[5]`) eingefügt werden. Für unsere 14 Schlüssel deutlich besser als `hash03` ist die ganz ähnlich aussehende Funktion `hash032`:

```
1 int hash032(String s) {
2    return s.charAt(2) % LAENGE_DER_HASHTAB;
1 }
```

Nach dem Einfügen aller 14 Beispiel-Schlüssel sieht die Hash-Tabelle `ht` wie folgt aus:

```
Index|Elemente der Listen (eingefügt mit hash032)
   0 | Arno Benno Bine Max
   1 | Alex
   2 | Alfred Alf
   3 | Angy
   4 | Kurt
   5 | Ali Alice Susi
   6 |
   7 | Franz
   8 | Babsy
   9 |
Suchschritte insgesamt mit hash032: 24
```

Qualitätskriterium: Eine Hash-Funktion ist gut, wenn sie die Schlüssel (auf die man sie anwendet) möglichst gleichmäßig auf alle Listen verteilt.

Bei unserer Hash-Tabelle der Länge 10 und den 14 Beispiel-Schlüsseln bedeutet das: Nach dem Einfügen aller Schlüssel sollte jede Liste *einen* oder *zwei* Schlüssel enthalten.

Aufgabe-03: Geben Sie 14 Schlüssel an (möglichst bekannte Vornamen), die von der Hash-Funktion `hash03` möglichst gleichmäßig auf die 10 Listen der Hash-Tabelle verteilt werden.

Die folgende Funktion `hash04` ist nicht nur für die 14 Beispiel-Schlüssel, sondern auch für viele andere Schlüssel, besser als `hash03`:

```
 1 int hash04(String s) {
 2    // Der Index wird aus 3 Zeichen von s berechnet (dem ersten und dem
 3    // letzten Zeichen und einem Zeichen aus der Mitte):
 4    int vorn   = s.charAt[0]          % 3; // 2 Bits vom erste char-Wert
 5    int mitte  = s.charAt[s.size()/2] % 7; // 3 Bits aus der Mitte
 6    int hinten = s.charAt[s.size()-1] % 7; // 3 Bits vom letzten char-Wert
 7    return (vorn + mitte + hinten)    % LAENGE_DER_HASHTAB;
 8 }
```

Nach dem Einfügen aller 14 Beispiel-Schlüssel sieht die Hash-Tabelle `ht` wie folgt aus:

```
Index|Elemente der Listen (erzeugt mit hash04)
   0 | Franz
   1 | Benno
   2 | Babsy
   3 | Arno
   4 |
   5 | Ali Alice Susi
   6 | Alfred Kurt Alex
   7 |
   8 | Bine
   9 | Angy Max Alf
Suchschritte insgesamt mit hash04: 23
```

Man sieht: hier sind nur noch 2 der 10 Listen leer und die längsten Listen enthalten 3 Schlüssel. Die folgende Funktion `hash05` ist für die 14 Beispiel-Schlüssel noch etwas besser (allerdings kostet eine Ausführung von `hash05` auch ein bisschen mehr, als eine Ausführung von `hash04`):

```
 1 int hash05(String s) {
 2    // Der Ergebnis-Index wird im wesentlichen aus der Summe aller Zeichen
 3    // des Schluessels s berechnet:
 4    int i = 0;
 5    for (int j=0; j<s.size(); j++) {
 6       i += s.charAt(j) % 32 + j;
 7    }
 8    return i % LAENGE_DER_HASHTAB;
 9 }
```

Der Ausdruck `s.charAt(j) % 32` bezeichnet die rechten 6 Bits des Zeichens `s.charAt(j)`. Beim Unicode der lateinischen Buchstaben (A-Z und a-z) sind alle anderen Bits gleich 0 und spielen somit keine Rolle. Die Funktion `hash05` verteilt die 14 Beispiel-Schlüssel wie folgt auf die 10 Listen unserer Hash-Tabelle:

```
Index|Elemente der Listen (eingefügt mit hash05)
   0 | Alice Benno
   1 | Alfred Max
   2 | Alf
   3 | Angy
   4 | Arno Susi
   5 | Ali Franz
   6 | Kurt Bine
   7 |
   8 | Alex
   9 | Babsy
Suchschritte insgesamt mit hash05: 19
```

Das ist schon nahe am Optimum: Nur noch *eine* Liste ist leer geblieben und keine Liste enthält mehr als 2 Schlüssel.

Positive Eigenschaft 2: Für Hash-Tabellen mit guten Hash-Funktionen gilt:
Vergrößert man die Hash-Tabelle (d.h. nimmt man eine längere Reihung), so werden die einzelnen Listen im allgemeinen *kürzer* und das Einfügen und Suchen wird *schneller*.

Positive Eigenschaft 1: Die **wichtigste** Eigenschaft einer guten Hash-Tabelle:
Solange die einzelnen Listen sehr kurz bleiben, ist die Zeit für den Zugriff auf ein Objekt (sowohl beim Einfügen als auch beim Suchen) praktisch *unabhängig von der Problemgröße n* (d.h. von der Anzahl der eingefügten Objekte). Die Methoden zum Einfügen, Suchen und Löschen haben also alle eine Zeitkomplexität **O(1)**.

Zum Vergleich: Beim binären Suchen in einer sortierten Reihung oder beim Suchen in einem Baum wächst die Zeit für das Suchen zwar langsam (Zeitkomplexität **O(log(n))**, aber sie *wächst*.

Hash-Funktionen sollten in der Praxis von Spezialisten mit fundierten Kenntnissen in Statistik entwickelt werden. Diese Spezialisten versuchen, möglichst viel über die statistischen Eigenschaften der Schlüssel herauszufinden und die Hash-Funktion auf diese Eigenschaften abzustimmen. Je mehr Eigenschaften der Schlüssel bekannt sind, desto besser kann man die Hash-Funktion darauf abstimmen. Wenn man z.B. weiß, dass das erste Zeichen eines Schlüssels immer ein Buchstabe ist, dann wird man wahrscheinlich von diesem ersten Zeichen nur die Bits verwenden, durch die sich Buchstaben voneinander unterscheiden. Und sollte man sogar wissen, dass als erstes Zeichen z.B. nur 'A' und 'B' in Frage kommen, dann nimmt man nur das Bit, durch das die beiden Zeichen sich unterscheiden etc. Bei sehr großen Schlüsseln besteht die Kunst darin, "die wichtigen Bits" herauszufinden und alle anderen Bits unberücksichtigt zu lassen, damit die Hash-Funktion schnell ist.

Die Java-Methode hashCode

In Java besitzt *jedes* Objekt `ob1` eine parameterlose Methode namens `hashCode`, die einen `int`-Wert liefert. Mit Hilfe der Operation `Math.abs(ob1.hashCode()) % LAENGE_DER_HASHTABELLE` kann man diese "universelle" Hash-Funktion an die Länge einer bestimmten Hash-Tabelle anpassen. Der Aufruf von `Math.abs` ist notwendig, weil die Funktion `hashCode` auch negative Ergebnisse liefert.

Die Methode `hashCode()` wurde von Spezialisten entwickelt, berücksichtigt aber natürliche nicht die besonderen Eigenschaften einer bestimmten Population von Schlüsseln. Deshalb muss man auch in Java für bestimmte Anwendungen maßgeschneiderte Hash-Funktionen entwickeln (lassen). Verwenden wir in unserem konkreten Beispiel die Funktion `hashCode`, so sieht das Ergebnis wie folgt aus:

```
Index|Elemente der Listen (eingefügt mit hashCode())
   0 | Alfred Arno Susi
   1 |
   2 | Kurt Bine
   3 | Franz
   4 | Alex Max
   5 | Alf
   6 |
   7 | Babsy
   8 | Ali Alice Benno
   9 | Angy
Suchschritte insgesamt mit hashCode(): 22
```

Für die 14 Beispiel-Schlüssel ist `hashCode` also nicht ganz so gut wie die "maßgeschneiderte" Funktion `hash05` (siehe oben).

Hash-Tabellen im Vergleich zu binären Bäumen

Eine Sammlung, die als (sortierter) Baum implementiert ist, kann man ganz einfach in aufsteigender oder absteigender Reihenfolge der Schlüssel traversieren ("durchklettern") und dabei die Knoten in dieser Reihenfolge bearbeiten (z.B. ausgeben). Bei einer Hash-Tabelle ist das nicht möglich, denn es gehört "zu ihrem Wesen", dass die Komponenten "möglichst zufällig über die einzelnen Listen verteilt sind".

Hash-Tabellen, die keine Reihungen von Listen sind

Angenommen, wir wollen `String`-Objekte sammeln. Es gibt Hash-Tabellen, die in diesem Fall keine *Reihungen von Listen von Strings* sind, sondern einfach *Reihungen von Strings*.

In solchen Hash-Tabellen kann es vorkommen, dass ein String `s1` eingefügt werden soll, sein Platz aber schon von einem anderen String `s2` besetzt ist (weil die Hash-Funktion die Strings `s1` und `s2` auf denselben Reihungsindex abgebildet hat).

Solche Situationen bezeichnet man als *Kollisionen*. Es gibt verschiedene Möglichkeiten, solche Kollisionen zu behandeln (z.B. Lineares Sondieren, Quadratisches Sondieren, Doppel-Hashing, Brent-Hashing etc.). Die Java-Klasse `Hashtable` (seit Java 1.0) enthält eine solche Kollisionsbehandlung. Die Klasse `HashSet` (seit Java 1.2) arbeitet stattdessen mit Verkettung, ähnlich wie in diesem Papier skizziert.

Was liefert die Java-Methode hashCode()? Ein paar Beispiele

```
HashCodes: Jetzt geht es los!
-------------------------------------
A new HashCodes()         .hashCode(): 21598637
A new HashCodes()         .hashCode(): 14236464
-------------------------------------
B new StringBuilder("ABC").hashCode(): 23934342
B new StringBuilder("ABC").hashCode(): 22307196
-------------------------------------
C                   "ABC" .hashCode(): 64578
C                   "ABC" .hashCode(): 64578
-------------------------------------
D        new String("ABC").hashCode(): 64578
D        new String("ABC").hashCode(): 64578
-------------------------------------
E new String("ABCDEFGHI") .hashCode(): -1113192379
E new String("ABCDEFGHI") .hashCode(): -1113192379
-------------------------------------
F new BigInteger("123")   .hashCode(): 123
F new BigInteger("123")   .hashCode(): 123
-------------------------------------
G new BigDecimal("1.5")   .hashCode(): 466
G new BigDecimal("1.5")   .hashCode(): 466
-------------------------------------
H new Boolean(true)       .hashCode(): 1231
H new Boolean(true)       .hashCode(): 1231
-------------------------------------
I new Double (123456789D) .hashCode(): 362639156
I new Double (123456789D) .hashCode(): 362639156
-------------------------------------
J new Integer(123456789)  .hashCode(): 123456789
J new Integer(123456789)  .hashCode(): 123456789
-------------------------------------
K new Long   (123456789L) .hashCode(): 123456789
K new Long   (123456789L) .hashCode(): 123456789
K new Long  (-123456789L) .hashCode(): 123456788
L new Long(123456789012L) .hashCode(): -1097262584
-------------------------------------
HashCodes: Das war's erstmal!
```

In Java enthält jedes Objekt eine parameterlose Funktion `hashCode` mit dem Rückgabetyp `int`.

Regel: Seien `ob1` und `ob2` zwei Objekte. Normalerweise haben die Ausdrücke
`ob1 != ob2` und `ob1.hashCode() != ob2.hashCode()`
gleiche Werte (beide sind `true` oder beide sind `false`), siehe Zeilen A und B.

Zur Erinnerung: Der Befehl `new` liefert bei jedem Aufruf einen *neuen* Referenz-Wert (der sich von allen vorher gelieferten Referenz-Werten unterscheidet). Der Ausdruck `ob1 != ob2` ist also immer gleich `true`, wenn `ob1` und `ob2` mit zwei Aufrufen von `new` initialisiert wurden.

Ausnahmen von obiger **Regel** sind Objekte der Klassen `String`, `BigInteger`, `BigDecimal` und die Objekte der Hüllklassen (`Byte`, `Boolean`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`). Bei denen hängt das Ergebnis von `hashCode()` nur vom "Inhalt der Objekte" ab, siehe Zeilen C bis K.

Achtung: Bei manchen Objekten `ob` ist `ob.hashCode()` *negativ*, siehe Zeilen E und L.

Lösung-01: Hash-Funktionen, die einen der folgenden Befehle als Rumpf haben, sind für unser Beispiel (mit einer Hash-Tabelle der Länge 10) so schlecht wie nur möglich:

```
int hash0(String s) {return 0;}
int hash1(String s) {return 1;}
...
int hash9(String s) {return 9;}
```

Lösung-02: A, K, U

Lösung-03: Für diese Aufgabe gibt es sehr viele verschiedene Lösungen, z.B. die folgende:

`Franz, Gabi, Hans, Ilse, Jürgen, Karin, Ludwig, Mira, Norbert, Ottilie, Fanny, Gert, Hanna, Ingo`

Erläuterung:

Die Großbuchstaben F - O haben im Unicode die Codenummern 70 bis 79. Daraus folgt:

`Franz` und `Fanny` kommen in die Liste 0, `Gabi` und `Gert` in die Liste 1, `Hans` und `Hanna` in die Liste 2, `Ilse` und `Ingo` in die Liste 3, `Jürgen` in die Liste 4, ..., `Ottilie` in die Liste 9.