

Hash-Tabellen

Erstaunlicherweise gibt es ein Suchverfahren, welches noch *schneller* ist als das *binäre Suchen* in einer *sortierten Reihung* oder das Suchen in einem *binären Baum*. Um es anzuwenden, muss man die Objekte in einer sogenannten Hash-Tabelle abspeichern. Solche Hash-Tabellen gibt es in zahlreichen Varianten. Hier soll nur ihr *Grundprinzip* anhand einer besonders einfachen Varianten dargestellt werden. Dieses Grundprinzip wird in der folgenden Definition zusammengefasst:

Def.: Eine Hash-Tabelle ist eine **Reihung** von verketteten Listen.

Die *Reihung* hat eine feste, unveränderbare Länge (z.B. die Länge 10 oder die Länge 10.000 oder ...). Die einzelnen *Listen* in der Reihung sind anfänglich leer, aber flexibel (d.h. man kann jederzeit noch eine weitere Komponente anhängen). Wenn eine Hash-Tabelle gut funktioniert, sind die meisten ihrer Listen *sehr kurz* und enthalten nur 1, 2 oder 3 Komponenten. Darum läßt man diese Listen in aller Regel *unsortiert*.

Eine Komponente κ in eine Hash-Tabelle ht einzufügen bedeutet, κ in eine der Listen $ht[i]$ einzufügen. In welche Liste $ht[i]$ man κ einfügt (d.h. welchen Wert man für i nimmt) entscheidet man mit Hilfe einer *Hash-Funktion*. Bevor wir uns konkrete Beispiele ansehen, soll hier beschrieben werden, was alle Hash-Funktionen *gemeinsam* haben und wie sie *grundsätzlich funktionieren*.

Eine *allgemeine Hash-Funktion* berechnet ("irgendwie") aus einem beliebigen Objekt einen `int`-Wert:

```
1 int allgHash(Object ob) { ... };
```

Eine *spezielle Hash-Funktion* für eine Hash-Tabelle ht berechnet aus einer *Komponenten* (die man in die Tabelle einfügen oder darin suchen will) einen *Index* für die Reihung ht . Angenommen, wir wollen `String`-Objekte in eine Hash-Tabelle ht einfügen und haben eine allgemeine Hash-Funktion `allgHash`, dann können wir damit leicht eine spezielle Hash-Funktion für ht definieren, etwa so:

```
2 int spezHash(String str) {
3     return Math.abs(allgHash(str)) % ht.length;
4 }
```

Die Betragsfunktion `abs` und die Operation `% ht.length` garantieren zusammen, dass das Ergebnis von `spezHash` immer zwischen 0 und `ht.length-1` liegt und somit ein gültiger Index für ht ist.

Mit der Funktion `spezHash` werden Komponenten wie folgt in ht eingefügt bzw. gesucht:

Einfügen einer Komponenten K:

Schritt E1: Aus K wird mit der Hash-Funktion `spezHash` ein Index `ind` für die Reihung ht berechnet.

Schritt E2: Dann wird K in die Liste $ht[ind]$ eingefügt.

Suchen einer Komponenten K:

Schritt S1: Aus K wird mit der Hash-Funktion `spezHash` ein Index `ind` für die Reihung ht berechnet.

Schritt S2: Dann wird K in der Liste $ht[ind]$ gesucht.

Wie man in eine verkettete Liste einfügt und wie man darin sucht wird hier als bekannt vorausgesetzt (siehe dazu z.B. die Klasse `MeineListe`).

Als konkretes Beispiel betrachten wir folgende Aufgabe: In eine Hash-Tabelle ht der Länge 10 sollen die folgenden 14 `String`-Komponenten eingefügt werden:

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

Das Geheimnis ("die Magie") einer Hash-Tabelle steckt im Wesentlichen in der verwendeten *Hash-Funktion*. Wir wollen jetzt mehrere konkrete (spezielle) *Hash-Funktionen* betrachten und beginnen mit der Funktion `hash01`, die garantiert zu den *schlechtesten* gehört, die es (für unsere Hash-Tabelle `ht` der Länge 10) gibt:

```
5 int hash01(String s) {
6     return 3;
7 }
```

Wenn man diese (pessimale) Hash-Funktion verwendet, werden alle Objekte in die Liste `ht[3]` der Hash-Tabelle eingefügt und die anderen Listen (`ht[0]` bis `ht[2]` und `ht[4]` bis `ht[9]`) bleiben *leer*. Hier eine Darstellung der Hash-Tabelle, nachdem alle 14 Beispiel-Komponenten (siehe oben) mit Hilfe der Funktion `hash01` eingefügt wurden:

| Index | Komponenten in den Listen |
|-------|--|
| 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 bedeutet: Wenn man jede Komponente, die in der Hash-Tabelle vorkommt, *einmal* sucht, braucht man dazu insgesamt (und "im wesentlichen") *105 Listen-Suchschritte*. Die Komponente "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: für den Mangel an Güte) der verwendeten Hash-Funktion.

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

Aufgabe: Beschreiben Sie neun weitere Hash-Funktionen, von denen Sie garantieren können, dass sie genau so schlecht sind wie `hash01`.

Die folgende Funktion `hash02` ist schon deutlich besser als `hash01`:

```
8 int hash02(String s) {
9     if (s.charAt(0) % 2 == 0) {
10        return 3;
11    } else {
12        return 4;
13    }
14 }
```

Je nachdem ob das erste Zeichen von `s` (`s.charAt(0)`) durch eine *gerade* oder durch eine *ungerade* Zahl codiert wird, liefert diese Hash-Funktion den Index 3 oder den Index 4. Alle Komponenten werden also in die Liste `ht[3]` oder in die Liste `ht[4]` eingefügt, die Listen `ht[0]` bis `ht[2]` und `ht[5]` bis `ht[9]` bleiben garantiert leer. Hier die Tabelle, nachdem alle 14 Beispiel-Komponenten mit Hilfe von `hash02` eingefügt wurden:

| Index | Komponenten in den Listen |
|-------|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | Babsy Benno Bine Franz |
| 4 | Ali Alfred Arno Alice Kurt Alex Angy Max Susi Alf |
| 5 | |
| ... | |
| 9 | |

Suchschritte insgesamt mit hash02: 65

An der letzten Zeile kann man erkennen, dass die Funktion `hash02` deutlich *besser* ist als `hash01`. Hier eine noch bessere Hash-Funktion:

```
15 int hash03(String s) {
16     return s.charAt(0) % ht.length;
17 }
```

Diese Hash-Funktion berechnet den Index aus dem *ersten Zeichen* von `s`. Die Operation `% ht.length` stellt sicher, dass wir immer einen gültigen Index für die Hash-Tabelle `ht` bekommen. Hier die mit Hilfe von `hash03` gefüllte Tabelle:

| Index | Komponenten in den Listen |
|-------|--|
| 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 Komponenten mit gleichem Anfangsbuchstaben in dieselbe Liste kommen. Allerdings können in einer Liste auch Komponenten mit *verschiedenen* Anfangsbuchstaben stehen (weil z.B. `'A' % 10` gleich `'K' % 10` gleich 5 ist).

Aufgabe: Die Funktion `hash03` bewirkt, dass alle mit `'A'` und alle mit `'K'` beginnenden Komponenten in die Liste `ht[5]` kommen. Nennen Sie einen weiteren Anfangsbuchstaben, der von `hash03` der Liste `ht[5]` zugeordnet wird.

Statistische Verteilung: 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 Komponenten berücksichtigen, auf die man sie anwendet. Für die 14 Beispiel-Komponenten ist `hash03` nicht besonders gut, denn sie läßt 5 der 10 Listen unserer Hash-Tabelle leer und bewirkt, dass 8 der 14 Komponenten in dieselbe Liste (Liste `ht[5]`) eingefügt werden. Am besten ist es, wenn eine Hash-Funktion alle Komponenten *möglichst gleichmäßig* auf alle Listen der Hash-Tabelle verteilt.

Aufgabe: Geben Sie 14 Komponenten (möglichst bekannte weibliche und männliche Vornamen) an, die von der Hash-Funktion `hash03` möglichst gleichmäßig auf die 10 Listen der Hash-Tabelle verteilt werden (dabei heißt "möglichst gleichmäßig": pro Liste 1 oder 2 Komponenten).

Die folgende Funktion `hash04` ist *für die 14 Beispiel-Komponenten* besser als `hash03`:

```
18 int hash04(String s) {
19     // Der Index wird aus 3 Zeichen von s berechnet (dem ersten und dem
20     // letzten Zeichen und einem Zeichen aus der Mitte):
21     int vorn    = s.charAt(0) % 3;
22     int mitte   = s.charAt(s.size()/2) % 5;
23     int hinten  = s.charAt(s.size()-1) % 7;
24     return (vorn + mitte + hinten) % ht.length;
25 }
```

Hier die mit Hilfe der Funktion `hash04` gefüllte Hash-Tab:

| Index | Komponenten in den Listen |
|-------|---------------------------|
| 0 | |
| 1 | |
| 2 | Susi |
| 3 | Bine |
| 4 | Alex |
| 5 | Ali Babsy Alice Max |
| 6 | Benno Franz |
| 7 | Angy |
| 8 | Alfred Arno Kurt |
| 9 | Alf |

Suchschritte insgesamt mit `hash04`: 24

Man sieht: hier sind nur noch **2** der 10 Listen leer und die längsten Listen enthalten **4** Komponenten. Die folgende Hash-Funktion ist für die 14 Beispiel-Komponenten noch etwas besser:

```

26 int hash05(String s) {
27     // Der Ergebnis-Index wird im wesentlichen aus der Summe aller Zeichen
28     // von s berechnet:
29     int erg = 0;
30     for (int i=0; i<s.size(); i++) {
31         erg += s.charAt(i) % 32 + i;
32     }
33     return erg % ht.length;
34 }

```

Der Ausdruck `s.charAt(i) % 32` bezeichnet die rechten 6 Bits des *i*-ten Zeichens von *s*. Bei den lateinischen Buchstaben (A-Z und a-z) spielen alle anderen Bits keine Rolle und sind gleich 0. Die Funktion `hash05` verteilt die 14 Beispiel-Komponenten wie folgt auf die 10 Listen unserer Hash-Tabelle `ht`:

| Index | Komponenten in den Listen |
|-------|---------------------------|
| 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 `hashFunk05`: 19

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

Hash-Tabellen mit guten Hash-Funktionen haben folgende angenehme Eigenschaft: Wenn man die Hash-Tabelle (d.h. die Reihung) vergrößert, werden die einzelnen Listen im allgemeinen *kürzer* und das Einfügen und Suchen wird *schneller*. Allerdings vergrößert man gleichzeitig die Wahrscheinlichkeit, dass einige oder viele der Listen *leer* bleiben und die entsprechende Null-Referenz nur Speicherplatz vergeudet. Manche Programme prüfen deshalb vor der Erzeugung einer Hash-Tabelle, wie viel freier Speicher vorhanden ist und berechnen daraus eine "gute Größe" für die Hash-Tabelle. Diese Tabelle ist umso schneller, je mehr freier Speicher vorhanden war. Bei wenig freiem Speicher funktioniert die Tabelle auch noch, ist aber entsprechend langsamer.

Eine weitere typische und *angenehme* 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 Größe der Tabelle* (d.h. von der Anzahl der eingefügten

Objekte). Zum Vergleich: Beim binären Suchen in einer sortierten Reihung *wächst* die Zeit für das Suchen mit dem Logarithmus der *Länge der Reihung* (damit wächst sie zwar *langsam*, aber sie wächst).

Hash-Funktionen werden in der Praxis typischerweise von Spezialisten mit fundierten Kenntnissen in Statistik entwickelt. Diese Spezialisten versuchen, möglichst viel über die statistischen Eigenschaften der Komponenten herauszufinden und die Hash-Funktion auf diese Eigenschaften abzustimmen. Je mehr Eigenschaften der Komponenten bekannt sind, desto besser kann man die Hash-Funktion darauf abstimmen. Wenn man z.B. weiss, dass das erste Zeichen eines Komponentens 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 Komponentenn besteht die Kunst darin, "die wichtigen Bits" herauszufinden und alle anderen Bits unberücksichtigt zu lassen.

In Java enthält *jedes Objekt* ob eine allgemeine Hash-Funktion namens `hashCode`, (siehe die Unterscheidung von allgemeinen und speziellen Hash-Funktionen auf S. 1). Die Methode `hashCode` wurde von Spezialisten entwickelt, berücksichtigt aber natürliche nicht die besonderen Eigenschaften einer bestimmten Population von Komponentenn. Deshalb muss man auch in Java für bestimmte Anwendungen maßgeschneiderte Hash-Funktionen entwickeln lassen.

Hier unsere Beispiel-Tabelle, nachdem sie mit Hilfe der Standard-Methode `hashCode` gefüllt wurde:

| Index | Komponenten in den Listen |
|-------|---------------------------|
| 0 | Alfred Arno Susi |
| 1 | |
| 2 | Kurt Bine |
| 3 | Franz |
| 4 | Alex Max |
| 5 | Alf |
| 6 | |
| 7 | Babsy |
| 8 | Ali Alice Benno |
| 9 | Angy |

Mit `hashCode()`-als-Hash-Funktion Anzahl, Suchschritte insgesamt: 22

Dieses Ergebnis ist recht gut, bleibt aber ein bisschen hinter dem vorigen Ergebnis zurück, welches mit der "maßgeschneiderten Hash-Funktion" `hash05` erzeugt wurde.

Zum Abschluß folgt hier eine tabellarische Zusammenfassung der oben erwähnten Hash-Funktionen und der Suchschritte, die sie erfordern (je weniger Suchschritte desto besser):

| Hash-Funktion | Suchschritte insgesamt |
|-----------------------|------------------------|
| <code>hash01</code> | 105 |
| <code>hash02</code> | 65 |
| <code>hash03</code> | 45 |
| <code>hash04</code> | 24 |
| <code>hash05</code> | 19 |
| <code>hashCode</code> | 22 |