

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*, nämlich das Suchen in einer sogenannten Hash-Tabelle. Das Grundprinzip solcher Tabellen fassen folgenden beiden Definitionen zusammen:

Def.: Eine Hash-Tabelle ht ist eine **Reihung von verketteten Listen** (von Komponenten).

Def.: Eine Hash-Funktion hf für ht bildet jede mögliche Komponente auf einen **Index** für ht ab, d.h. auf einen `int`-Wert zwischen 0 (einschließlich) und $ht.length$ (ausschließlich).

Eine Komponente K wird in eine Hash-Tabelle ht *eingefügt*, indem man K in die Liste $ht[hf(K)]$ einfügt. K wird in ht *gesucht*, indem man K in der Liste $ht[hf(K)]$ sucht.

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

Um "die Magie von Hash-Funktionen" genauer zu verstehen, betrachten wir als konkretes Beispiel eine Hash-Tabelle ht der Länge 10, in der wir die folgenden 14 String-Komponenten sammeln wollen:

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

Jetzt sollen einige konkrete Hash-Funktionen vorgeführt und bewertet werden. Als erste betrachten wir eine Funktion `hash01`, die garantiert zu den *schlechtesten* gehört, die es gibt:

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

Sie bewirkt, dass alle Komponenten in dieselbe Liste $ht[3]$ eingefügt werden und die übrigen Listen *leer* bleiben. Hier eine Darstellung der Hash-Tabelle ht , 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

In der letzten Zeile wird angegeben, dass man insgesamt 105 Listen-Suchschritte benötigt, um jede Komponente einmal zu suchen (für "Ali" braucht man *einen* Suchschritt, für "Babsy" 2, für "Alfred" 3, ... und für "Alf" 14 Suchschritte, und $1+2+\dots+14$ ist gleich 105).

Diese Zahl ist ein *Maß für die Güte* (oder 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-01: 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`:

```

4 int hash02(String s) {
5     if (s.charAt(0) % 2 == 0) {
6         return 3;
7     } else {
8         return 4;
9     }
10 }

```

Sie bewirkt, dass jede Komponente in eine der beiden Listen `ht[3]` oder `ht[4]` eingefügt wird. Nach dem Einfügen aller 14 Komponenten mit Hilfe der Funktion `hash02` sieht die Sammlung `ht` so aus:

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 Funktion `hash03`:

```

11 int hash03(String s) {
12     return s.charAt(0) % ht.length;
13 }

```

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 Hash-Tabelle `ht`:

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

Die Funktion `hash03` bewirkt offenbar, dass alle Komponenten mit gleichem Anfangsbuchstaben in dieselbe Liste kommen. Komponenten mit *verschiedenen* Anfangsbuchstaben kommen aber nicht unbedingt in verschiedene Listen (weil z.B. `'A' % 10` gleich `'K' % 10` gleich 5 ist).

Aufgabe-02: Nennen Sie einen weiteren Anfangsbuchstaben, der (wie die Buchstaben `'A'` und `'K'`) von `hash03` der Liste `ht[5]` zugeordnet wird.

Statistische Verteilung: Wie gut oder schlecht eine Hash-Funktion ist, hängt auch von den *Komponenten* ab, auf die man sie anwendet. Für unsere Beispiel-Komponenten ist `hash03` nicht besonders gut, denn sie läßt 5 der 10 Listen leer und bewirkt, dass 8 der 14 Komponenten in dieselbe 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-03: Geben Sie 14 Komponenten an (möglichst bekannte weibliche und männliche Vornamen), 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`:

```

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

```

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:

```

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

```

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 von `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) verlängert, 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 aber doch Speicherplatz belegen. Manche Program-

me 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.

Die wichtigste Eigenschaft einer Hash-Tabelle: Solange die einzelnen Listen sehr *kurz* bleiben, ist die Zeit für den Zugriff auf eine Komponente (sowohl beim Einfügen als auch beim Suchen) praktisch *unabhängig von der Größe der Sammlung* (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 Hash-Funktion namens `hashCode`. Die Methode `hashCode` wurde von Spezialisten entwickelt, berücksichtigt aber natürliche nicht die besonderen Eigenschaften einer bestimmten Population von Komponenten. 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