Hashing*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Hashfunktionen, Hashtabellen, Prüfsummen

Empfohlene Vorkenntnisse: Arrays.

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Motivation

Zugriffe auf Arrayelemente sind sehr effizient. Möchte man z.B. auf das 5. Element eines Arrays zugreifen, dann muss man nur zur Anfangsadresse des Arrays 5 mal die Speicherzellenbreite hinzuaddieren, und hat damit die Adresse des 5. Elements. Leider hat man oft Mengen von Objekten, denen man nicht direkt eine Zahl zuordnen kann, um sie an dieser Stelle im Array abspeichern zu können. Für ein Telefonbuch, z.B. muss man für jeden Namen die Telefonnummer speichern, d.h. man hat die Zuordnung: Name (String) $\rightarrow$ Telefonnummer (Integer). Ein String ist aber keine Zahl, die man als Arrayindex verwenden könnte, um dort die Telefonnummer abzuspeichern. Die einfachste Lösung wäre, die Telefonliste als *Liste* von Tupeln (Name, Telefonnummer) abzuspeichern. Um dann aber die Nummer eines Herrn Krause zu bekommen, muss man die ganze Liste durchsuchen, bis man auf den Namen Krause stößt. Das kann sehr aufwendig sein, insbesondere wenn die Liste sehr lang ist. Viel besser wäre es, wenn man es irgendwie schaffen könnte, einen Namen, d.h. einen String, auf eine Zahl abzubilden. Dann könnte man diese Zahl als Arrayindex in einer sog. *Hashtabelle* benutzen und dort die Telefonnummer abspeichern. Der Zugriff würde dann insbesondere nicht von der Länge der Liste (des Arrays) abhängen, und wäre daher viel schneller.

Eine Abbildung h: Objekt $\rightarrow$ Integer, die je nach Anwendung noch bestimmte Bedingungen erfüllt, bezeichnet man als *Hashfunktion*.

2 Hashfunktionen

Um eine erste Idee für eine Hashfunktion zu bekommen, bilden wir Strings auf Zahlen ab, indem wir einfach die ASCII-Nummer der Buchstaben addieren. Z.B. für den Namen Krause erhält man dann:

	K	r	a	u	s	e	Summe
ASCII	75	114	97	117	115	101	619

Diese Zahl kann sehr groß werden. Will man sie als Arrayindex für ein Array der Länge z.B. 100 benutzen, dann nimmt man die Zahl modulo der Arraylänge, $619 \; mod \; 100 = 19$. An Position 19 würde man jetzt die Telefonnummer von Herrn Krause abspeichern. Sucht man jetzt die Telefonnummer von Herrn Krause, berechnet man wieder den *Hashwert* 19 und greift damit auf das Array zu.

Allerdings wird jetzt schon ein Problem deutlich: Krause ist höchstwahrscheinlich nicht der einzige Name mit dem Hashwert 19. Gibt es noch einen zweiten Namen, der auch auf 19 abgebildet wird, dann gibt es eine *Kollision*, die man auf unterschiedliche Weise behandeln kann und auch muss (siehe Abschnitt 3).

2.1 Eigenschaften von Hashfunktionen

Hashfunktionen h bilden also Objekte, z.B. Strings, auf Zahlen ab, und zwar i.A. Zahlen in einem vorgegebenen Bereich I, z.B. von 0 bis zu einer gegebenen Arraylänge. Die abgebildeten Objekte nennt man manchmal auch den *Schlüssel*. Es werden also Schlüssel auf Zahlen abgebildet.

Eine Eigenschaft von Hashfunktionen, die man gerne hätte, die sich aber fast nie erreichen lässt, ist die *Injektivität*, d.h. *keine zwei* Objekte werden auf dieselbe Zahl abgebildet. Insbesondere wenn man eine große Zahl von Objekten (z.B. ein ganzes Telefonbuch) in ein relativ kleines Array quetschen will, müssen mehrere Objekte auf denselben Index abgebildet werden. Injektivität ist dann nicht möglich.

Andere Eigenschaften lassen sich aber meist schon erreichen:

Surjektivität: Für jede Zahl im Zielbereich I gibt es Objekte, die darauf abgebildet werden. In unserem Telefonlistenbeispiel bedeutet dass, dass im Array keine Lücken entstehen, in denen prinzipiell keine Telefonnummern eingetragen werden können.

Gleichverteilung: Wenn mehrere Objekte auf die gleiche Zahl abgebildet werden, dann sollte es möglichst keine Zahlen geben, bei denen sich die Objekte häufen, während es für andere Zahlen wenig Objekte gibt, die darauf abgebildet werden.

Effizienz: Der Hashwert sollte effizient berechenbar sein.

wenig Speicherbedarf: Der Hashwert sollte deutlich weniger Platz benötigen, als das Objekt selbst. Jedes im Speicher abgelegte Objekt besteht aus einer Bitfolge, und jede Bitfolge lässt sich als Binärdarstellung einer Zahl interpretieren. Diese Zahl könnte auch der Hashwert sein, genügt aber nicht dieser Bedingung.

Für bestimmte Anwendungen, insbesondere wenn es um sichere Datenübertragung geht, fordert man noch weitere Eigenschaften:

Einwegfunktion: Die Hashfunktion sollte so sein, dass man aus dem Hashwert nicht oder nur mit extrem großen Aufwand das Ausgangsobjekt rekonstruieren kann.
Die Funktion, die für einen String die Summe der ASCII-Nummern berechnet, hat diese Eigenschaft trivialerweise, weil es viele verschiedenen Strings mit der gleichen Summe gibt.

Chaotisch: Wenn man auch nur ein Bit des Ausgangsobjekts verändert, sollte sich der Hashwert *total* verändern.
Die Summe der ASCII-Nummern eines Strings hat diese Eigenschaft dann nicht, wenn man Buchstaben austauscht. Ersetzt man z.B. ein a durch ein b, verändert sich die Summe nur um 1.

Hashfunktionen, die diese Eigenschaften haben, nennt man *kryptologische Hashfunktion*.

3 Hashtabellen

Die prominenteste Anwendung von Hashfunktionen ist die in dem Motivationskapitel erwähnte Möglichkeit, Objekte mittels deren Hashwerten Arrayzellen zuzuordnen. Da die Hashfunktionen aber so gut wie nie injektiv sind, muss man das Problem lösen, dass zwei oder mehr Objekte auf denselben Hashwert abgebildet werden, und daher dieselbe Arrayzelle belegen wollen. Eine solche Situation bezeichnet man als eine *Kollision*.

Es gibt mehrere Möglichkeiten, mit Kollisionen umzugehen. Ganz grob unterscheidet man

Hashing mit Verkettung: Dabei werden die kollidierenden Objekte z.B. in einer Liste gespeichert. Falls im Telefonbuch z.B. Mayer und Müller auf denselben Hashwert, sagen wir 25, abgebildet werden, dann wird an der Position 25 eine Liste mit Tupeln ((Mayer, Telefonnummer von Mayer), (Müller, Telefonnummer von Müller)) abgespeichert. Beim Zugriff auf die Telefonnummer von Müller muss dann die Liste an der Stelle 25 nach Müller durchsucht werden. Anstelle einer Liste kann man aber auch eine andere Datenstruktur wählen, in der die Suche effizienter ist.

Hashing mit offener Adressierung: Hierbei wird im Kollisionsfall an der Position n in der Nähe von n nach einer freien Arrayzelle gesucht, um den Wert dort unterzubringen. Beim Zugriff muss man dann wenn nötig die Nachbarzellen durchsuchen (*sondieren*).

Es gibt dafür mehrere Möglichkeiten. Einige davon sind:

Lineares Sondieren: Falls die gewünschte Zelle belegt ist, wird rechts und links davon eine freie Zelle gesucht.

Quadratisches Sondieren: Falls die gewünschte Zelle belegt ist, wird rechts und links davon ebenfalls eine freie Zelle gesucht, allerdings verdoppelt man den Abstand der zu testenden Nachbarzelle mit jedem erfolglosen Versuch.

Doppel-Hashing: Man benutzt zwei Hashfunktionen h_1 und h_2. Die eigentliche Hashfunktion ist $h(s) = (h_1(s) + j \cdot h_2(s)) \bmod m$, wobei m die Arraylänge ist und j die Anzahl von Fehlversuchen. Man probiert also zunächst $h(s) = h_1(s) + 0 \cdot h_2(s) = h_1(s)$. Falls die Position belegt ist, probiert man $h(s) = h_1(s) + 1 \cdot h_2(s) = h_1(s) + h_2(s)$. Falls das ebenfalls belegt ist, probiert man mit $j = 2$ usw.

Dies sind aber nicht die einzigen Möglichkeiten.

Eine Hashtabelle mit Verkettung wird erst dann voll, wenn der Speicher voll ist. Mit offener Adressierung kann die Hashtabelle allerdings auch überlaufen. Will man dann das Programm nicht mit einer Fehlermeldung abbrechen, muss die Hashtabelle dynamisch vergrößert werden. Dazu muss aber auch die Hashfunktion verändert werden. Das erzwingt wiederum, dass alle Einträge neu eingetragen werden müssen.

Komplexität: Falls es keine Kollisionen gibt, hängt die Komplexität der Einfüge- und Zugriffsoperation nur von der Komplexität der Hashfunktion ab. Das Einfügen und Zugreifen selbst geht in konstanter Zeit. Die Kollisionsbehandlung kann man i.A. mit logarithmischem Aufwand bewältigen.

Nachteile: Hashtabellen sind eine sehr effiziente Speichertechnik, haben aber den Nachteil, dass es nicht möglich ist, eine Liste von Objekten in einer festgelegten Reihenfolge zu speichern. Für unser Telefonbuch Beispiel bedeutet das, dass man die in der Hashtabelle gespeicherten Einträge nicht alphabetisch geordnet ausgeben kann. Möchte man Mengen von Objekten in einer sortierten Reihenfolge abspeichern, dann braucht man andere Verfahren.

4 Prüfsummen

Eine ganz andere Anwendung von Hashfunktionen ist die Berechnung und Übertragung von Dokumenten zusammen mit einer *Prüfsumme*. Die Idee hierbei ist, ein Dokument auf einen Hashwert abzubilden, und beides zu übertragen. Der Empfänger kann dann ebenfalls das Dokument auf den Hashwert abbilden. Der übertragene und der neu berechnete Hashwert sollten übereinstimmen. Falls nicht, ist bei der Übertragung ein Fehler passiert. Hierfür eignen sich insbesondere kryptologische Hashfunktionen, da bei ihnen die Kollisionswahrscheinlichkeit am geringsten ist. Chaotische Hashfunktionen bewirken insbesondere, dass auch kleine Änderungen am Dokument den Hashwert radikal verändern. Daher können auch kleine Fehler und Manipulationen am Dokument bei der Überprüfung des Hashwertes leicht bemerkt werden.

Ein ganz konkrete Anwendung von Hashing steckt in den IBAN-Kontonummern. Die beiden ersten Ziffern hinter dem Länderkürzel sind eine Prüfsumme, die aus der Kontonummer berechnet wird. Hat man sich bei der Kontonummer vertippt, dann stimmt die Prüfsumme nicht. Das lässt sich leicht schon bei der Eingabe in einen Browser testen.

5 Hashing in Java

Die Klasse `Object` in Java enthält eine Methode `public int hashCode()`, die als Hashwert die momentane Adresse des Objekts im Speicher liefert. Der Hashwert wird z.B. zur Implementierung von `HashMaps` benutzt. Wenn die Adresse des Objekts als Hashwert benutzt wird, hängt es von Verteilung der Objekte im Speicher ab, ob Surjektivität und Gleichverteilung gewährleistet sind. Darauf hat man aber als Programmierer keinen Einfluss.

Daher sollte die `hashCode` Methode in Unterklassen von `Object` neu definiert werden. Implementiert man z.B. eine Klasse `Student` mit Instanzvariable `int matrikelnummer`, dann bietet es sich an, die (eindeutige) Matrikelnummer als Hashwert zu benutzen. Parallel zur Implementierung der Methode hashCode sollte man auch die Methode equals überschrieben, weil sie genutzt wird um zu prüfen, ob derselbe Wert an dem gehashten Index steht. In Joshua Bloch, Effective Java gibt es Vorschläge, wie man eine gute Hashfunktion für Objekte einer Klasse schreibt.

Stichwortverzeichnis