Hashing

Dana Müller
University of Applied Sciences Zittau/Görlitz
02826 Görlitz, Germany
s3damuel@hszg.de

ABSTRACT

Der folgende Beleg beschäftigt sich mit den Thema Hashing. Da dies ein sehr mathematischen Thema ist, werden die verschiedenen Verfahren in der ganzen Arbeit mit Gleichungen belegt. Eingegangen wird dabei auf die Eigenschaften, welche eine Hashfunktion mit sich bringen sollte, ebenso wie auf verschiedene Hashfunktionen, darunter Überläuferverkettung, offene Hashverfahren und einige aktuelle Hashverfahren. Mit dem Verlauf der Arbeit werden dabei von Kapitel zu Kapitel die verschiedenen Probleme durch neue Algorithmen versucht zu lösen, so dass sich eine gut nachvollziehbare Struktur ergibt.

Keywords

Hashwert, Algorithmen, Effizienzlösungen, Abspeichern von Datensätzen

1. EINLEITUNG

Im Jahre 2006 gab es das sogenannte Informatikjahr, in welchem jede Woche ein Algorithmus der Woche vorgestellt wurde. In Woche 34 war dieser Algorithmus das Hashing (nachzulesen unter [5]), welcher damals mit Hilfe eines Hasens vorgestellt wurde. Das möchte ich an dieser Stelle nicht tun, auch wenn sich meine Arbeit (in einem größeren Umfang) ebenfalls mit dem Thema Hashing beschäftigt.
Da Hashverfahren ein sehr mathematisches Thema sind, habe ich zu Beginn eine Vorstellung der von mir genutzten Variablen vorangestellt, damit ich später, wenn ich die verschiedenen Hashverfahren anhand von mathematischen Gleichungen vorstelle, diese Einführung nicht nochmals vornehmen muss und es für den Leser leichter ist, auch zwischendurch, wenn eine Variable unbekannt ist, an einer Stelle nachzuschlagen, für welche Werte diese steht. Doch mit was beschäftigt sich das Hashing überhaupt? Hashverfahren werden genutzt um einzelne Daten einer großen Datenmenge in einer Tabelle abzuspeichern, zu suchen oder zu entfernen. Für diese drei Tätigkeiten, werden jeweils verschiedene, z.T. voneinander abhängige Algorithmen verwendet. So ist leicht zu verstehen, dass sowohl für das Einfügen als auch das Entfernen eine vorherige Suche sinnvoll ist, denn je nach Ergebnis dieser, kann festgestellt werden, ob das Element überhaupt eingefügt werden muss bzw. gelöscht werden kann. Im Verlauf der Arbeit möchte ich auf die größeren Hauptgruppen von Verfahren (Verkettung der Überläufer und offene Hashverfahren) eingehen und verschiedene Vertreter dieser Gruppe vorstellen, wobei der Aufbau der Arbeit mit Absicht so gewählt ist, da so im Verlauf des Beleges die Nachteile der Verfahren aufgezeigt werden und wie diese in später entwickelten Verfahren verarbeitet wurden. Zum Abschluss möchte ich dann noch auf drei aktuelle bzw. heutzutage noch oft verwendete Hashverfahren eingehen um zu zeigen, dass dieses Thema sich stetig weiter entwickelt.

2. GENUTZTE VARIABLEN

Da Hashing ein sehr mathematisches Thema ist, möchte ich an dieser Stelle zunächst die Bedeutung von vielen Variablen erklären, welche ich dann im Fortlauf meiner Arbeit nutzen werde.
Die Grundlage zur Entstehung jeder Hashfunktion ist zunächst eine Menge, in welcher alle möglichen Schlüssel gespeichert werden, diese Menge wird mit $\mathcal{K}$ bezeichnet. Aus dieser Menge $\mathcal{K}$ kann jeweils immer nur eine bestimmte Teilmenge K betrachtet werden. Wählt man sich einen bestimmten Schlüssel aus der Menge $\mathcal{K}$ aus, oder betrachtet diesen, so wird er mit k bezeichnet. Ein Synonym von k, das heißt, ein Schlüssel, welcher die gleiche Hashadresse wie k besitzt so dass $h(k) = h(k')$ gilt, wird mit k' bezeichnet. Eine bestimmte Hashtabelle t wird als Feld von 0 bis $m-1$ dargestellt, damit hat sie auch immer eine feste Länge, welche mit m bezeichnet wird. Die Abbildung der Hashfunktion h ordnet dann jedem Schlüssel k einen Index mit $h(k)$ zu, welcher seine Hashadresse bezeichnet. Die Menge aller möglichen Hashfunktionen h wird mit $\mathcal{H}$ bezeichnet, vergleichend zu der Menge aller Schlüssel $\mathcal{K}$. Die Größe der Hashtabelle wurde nun durch m festgelegt, jedoch sagt m nichts über die Belegung der Hashtabelle aus. Dies ist erst durch n möglich, denn n beschreibt die Anzahl aller gespeicherten Schlüssel. Mit Hilfe von n und m kann man dann den Belegungsfaktor α berechnen, welcher dem Quotient $\frac{n}{m}$ entspricht. Weiterhin kann man zu jeder Hashfunktion h zwei Erwartungswerte berechnen, einerseits C_n und andererseits C'_n. C_n beschreibt den Erwartungswert für die Anzahl der betrachteten Einträge für eine erfolglose Suche und C'_n beschreibt den selben Wert, jedoch für die erfolgreiche Suche.

3. AUSWAHL EINER HASHFUNKTION

> „Eine gute Hashfunktion sollte möglichst leicht und schnell berechenbar sein und die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen um Adresskollisionen zu vermeiden.“ [4]

Dazu zunächst die Ergänzung, worum es sich bei einer Adresskollision handelt: diese tritt immer dann auf, wenn sich zwei Synonyme k und k' in der selben Menge K befinden, denn dann existiert ein Index $h(k) = h(k')$ welcher theoretisch doppelt belegt sein müsste, da dies nicht geht, kommt es zur Kollision.
Der Adressbereich der Hashtabelle von 0 bis $m-1$ soll also möglichst gut ausgewogen gefüllt sein und das unabhängig davon, wie k verteilt ist. Das Adresskollisionen jedoch auch bei der besten Hashfunktion nicht vermeidbar sind, zeigt das *Geburtstagsparadoxon*. Es sagt aus, das man bei einem vorgegebenen Geburtstagsdatum 183 Personen in einem Raum braucht, damit die Wahrscheinlichkeit dafür, das eine Person im Raum an eben diesem Tag Geburtstag hat 0,5 beträgt. Damit die Wahrscheinlichkeit dafür, dass zwei Personen am gleichen Tag Geburtstag haben 0,5 beträgt, reicht jedoch schon ein Raum mit 23 Personen [1]. Ottmann und Widmayer haben diese Gleichungen noch verallgemeinert: „Wenn eine Hashfunktion $\sqrt{\pi m/2}$ Schlüssel auf eine Hashtabelle der Größe m abbildet, dann gibt es wahrscheinlich eine Adresskollision“[4]. Für 365 Tage, ergibt sich dann die Anzahl von 23 Personen.
In den folgenden Betrachtungen von Hashfunktionen geht man von positiven ganzzahligen Schlüsseln aus, so dass $\mathcal{K} \subseteq \mathbb{N}_0$

3.1 Divisions-Rest-Methode (Modulo)

Bei der Division-Rest-Methode wird die Hashadresse $h(k)$ eines Schlüssels k mit Hilfe der Modulorechnung ermittelt, so dass die allgemeine Gleichung

$$h(k) = k \mod m$$

Dabei sollte man in der ebengenannten Gleichung besonders auf die Auswahl des m achten. Folgende m sind nicht zur Auswahl zu empfehlen bzw. sollte man auf deren Folgen achten:

- m als gerade Zahl, denn ist k gerade, so ist auch $h(k)$ gerade
- m als ungerade Zahl, denn ist k ungerade, so ist auch $h(k)$ ungerade
- m als Potenz der Basis des Zahlensystems in dem der Schlüssel dargestellt ist [4]
- für den Fall, dass r die Zahlensystembasis und i und j kleine, positive, ganze Zahlen sind, gilt: m sollte keine der Zahlen $r^i \pm j$ teilen

Daraus ergibt sich, das m im besten Falle eine Primzahl beschreibt.

3.2 Multiplikative Methode

Um die Hashfunktion zu erstellen, multipliziert man zunächst den ausgewählten Schlüssel mit einer irrationalen Zahl, dadurch ergibt sich eine weitere irrationale Zahl, nennen wir diese zur besseren Übersicht z_k. Vom Wert von z_k wird nun der ganzzahlige Anteil verworfen, so dass man eine reelle Zahl zwischen 0 und 1 erhält. Damit erhält man nun für verschiedene k Werte zwischen 0 und 1. Das diese Werte für Schlüssel von 1 bis n relativ gleichverteilt sind, zeigt der *Drei-Abstands-Satz* von Vera Turán Sós.

Satz 1 (*Drei-Abstands-Satz*)

Sei Θ eine irrationale Zahl. Platziert man die Punkte $\Theta - \lfloor\Theta\rfloor, 2\Theta - \lfloor 2\Theta\rfloor, 3\Theta - \lfloor 3\Theta\rfloor, \ldots, n\Theta - \lfloor n\Theta\rfloor$ in das Intervall [0,1], dann haben die $n+1$ Intervallteile höchstens drei verschiedene Längen. Außerdem fällt der nächste Punkt, $(n+1)\Theta - \lfloor(n+1)\Theta\rfloor$, in einen der größten Intervallteile.

Weiterhin ist bekannt, dass der goldene Schnitt

$$\Theta = \frac{\sqrt{5}-1}{2} \approx 0.6180339887$$

die gleichmäßigste Verteilung liefert. Aus diesen beiden Fakten ergibt sich dann folgende Hashfunktion:

$$h(k) = \lceil m(k\Theta^{-1} - \lfloor k\Theta^{-1}\rfloor)\rceil$$

3.3 Perfektes Hashing

In seltenen Fällen gilt, dass die Anzahl der zu speichernden Schlüssel kleiner ist, als die Anzahl der freien Speicherplätze, also gilt $n < m$ für alle möglichen Schlüssel. Außerdem ist bekannt welchen Wert K hat und das sich dieser nicht verändert. In diesem Fall kann man sagen, dass eine kollisionsfreie Speicherung möglich ist, da man eine Abbildung $h : K \to 0, 1, \ldots, m-1$ bilden kann, z.B. wenn man die Schlüssel lexikographisch ordnet und ihnen dann in dieser Reihenfolge ihre Ordnungsnummern (Indices der Hashtabelle) zuordnet. Man spricht dann von einer perfekten Hashfunktion, da keinerlei Kollisionen auftreten können. Da dies aber nicht die Regel ist, und K meist unbekannt ist und man selbst wenn $n < m$ gilt nicht sicher sein kann, dass es zu keinen Kollisionen kommt, entwickelte sich das Universelle Hashing, welches im nächsten Kapitel vorgestellt wird.

3.4 Universelles Hashing

Bei einer fest gewählten Hashfunktion wird man immer zwei oder mehr Schlüssel finden, welche auf die gleiche Hashadresse abgebildet werden müssten, so dass es zu einer Adresskollision kommen würde. Dieser Tatsache kann man nur aus dem Weg gehen, in dem man eine Menge $\mathcal{H}$ von Hashfunktionen nutzt, aus welcher dann eine Funktion h für jeden Schlüssel neu ausgewählt wird. Diese zufällige Auswahl garantiert dann, dass selbst eine ungünstige Menge $\mathcal{K}$ nicht zu vielen Kollisionen führt.
$\mathcal{H}$ kann also als *universell* bezeichnet werden, wenn es eine endliche Menge von Hashadressen ist und jede Funktion h aus dieser Menge die Schlüssel aus $\mathcal{K}$ auf eine Hashadresse $0, \ldots, m-1$ abgebildet werden kann und dann weiterhin gilt, dass jedes Paar von zwei verschiedenen Schlüsseln in höchstens $\frac{1}{m}$ Fällen eine Adresskollision zur Folge haben würde.
Eine universelle Hashfunktion schafft das Ergebnis, das selbst eine einseitige Menge $\mathcal{K}$ gleichmäßig auf die verfügbaren Adressen verteilt wird. Möchte man nun eine gute Hashfunktion auswählen, so muss man folgende Parameter beachten: die Anzahl der Schlüssel im Bereich m ist bekannt, man wählt eine Primzahl p welche größer oder gleich der Anzahl

dieser Schlüssel ist($p \geq n$), und zwei Zahlen a ($1 \leq a < p$) und b ($0 \leq b < p$) und so ergibt sich die Hashfunktion $h_{a,b}$.

4. HASHVERFAHREN MIT ÜBERLÄUFER-VERKETTUNG

Soll in einer Hashtabelle t, ein Synonym k' zu k eingetragen werden, so kommt es zu einer Adresskollision, denn $h(k) = h(k')$ ist bereits belegt, also muss k' (genannt Überläufer) anderswo gespeichert werden, z.B. außerhalb der Hashtabelle. Eine einfache Struktur wäre dabei die lineare Liste, welche an jeden Hashtabelleneintrag angehangen wird, wenn sich dieser Eintrag durch Anwendung der Hashfunktion ergibt.

4.1 Seperate Verkettung der Überläufer

Grundannahme: Jedes Element der Hashtabelle kann als Anfangselement einer Überläuferkette angesehen werden.

- Suchen nach k:
  - Man beginnt bei $t[h(k)]$ und folgt dann den Verweisen der Überläuferkette weiter. Wird k gefunden so ergibt sich eine erfolgreiche Suche, erreicht man das Ende der Überläuferkette so ergibt sich eine erfolglose Suche.
- Einfügen von k:
  - Die Suche von k war erfolglos. Endete diese bei $t[h(k)]$ so trage k ein, endete dieses am Ende der Überläuferkette, so hänge k an die Kette an.
- Entfernen eines k:
  - Die Suche von k war erfolgreich. Steht k in einer Überläuferkette, so lösche k aus dieser Kette, steht k in der Hashtabelle, dann streiche k dort, falls eine Überläuferkette bei $t[h(k)]$ beginnt, so schreibe das erste Element dieser Kette nach $t[h(k()]$.

4.2 Direkte Verkettung der Überläufer

Grundannahme: Jedes Element der Hashtabelle beinhaltet einen Zeiger auf die Überläuferkette.

- Suchen nach k:
  - Man beginnt beim Zeiger der Adresse der Hashtabelle ($t[h(k)] \uparrow$) und durchsucht die Überläuferkette bis man entweder k findet (erfolgreiche Suche) oder das Ende der Kette erreicht wurde (erfolglose Suche).
- Einfügen von k:
  - Die Suche verlief erfolglos.Der Schlüssel k wird als Listenelement an die Überläuferkette von $h(k)$ angefügt.
- Entfernen eines k:
  - Eine erfolgreiche Suche ist die Vorraussetzung zum Entfernen von k. Sie endete bei einem Listenelement der Überläuferkette, dieses wird nun entfernt.

4.3 Fallunterscheidung zur Nutzung

Um zu entscheiden, welche der beiden Verfahren mit Überläuferverkettung man nutzt, sollte man folgendes beachten: Sollen relativ kleine Datenmengen verarbeitet werden, welche gleichmäßig über die Hashtabelle t verbreitet sind, so bietet sich die Nutzung von Seperater Verkettung (Kapitel 4.1) an, da diese nicht zusätzlich den Speicher der Zeiger braucht. Sind die Datensätze allerdings relativ groß und ungleichmäßig verteilt, so ist die Direkte Verkettung besser, da man dabei nur wenig Speicherplatz durch unbesetzte Hashadressen ungebraucht lässt.

5. OFFENE HASHVERFAHREN

In Kapitel 4 wurden Verfahren behandelt, welche Überläufer außerhalb der Hashtabelle gespeichert haben. Im Weiteren soll nun auf Verfahren eingegangen werden, wie man diese Überläufer innerhalb der Hashtabelle abspeichern kann. Das bedeutet also, dass man, wenn die Position h(k) besetzt ist, eine feste Regel festlegt, mit welcher ein offener Platz gesucht wird, an welchem man dann k eintragen kann. Da man nun aber zum Moment der Abspeicherung nicht wissen kann, welche Plätze schon belegt sind, wird eine bestimmte Reihenfolge festgelegt, in welcher die Plätze überprüft werden, so lange bis ein freier Platz gefunden wird. Diese Folge von Überprüfungen nennt man *Sondierungsfolge* zum Schlüssel k. Methoden, die diesem Schema folgen, wurden von William Wesley Peterson *offene Hashverfahren* genannt, da immer nach einem offenen Platz gesucht wird. Das Entfernen eines Schlüssels ist bei diesen Verfahren allerdings immer problematisch, da man die Sondierungsfolge nicht zurückverfolgen kann. Aus diesem Grund wird k auch nicht wirklich entfernt, sondern nur als entfernt makiert. Das bedeutet, dass wenn ein neuer Schlüssel an der Adresse $h(k)$ eingefügt werden soll, dieser an der Stelle eingefügt wird und wird ein Schlüssel gesucht wird die Adresse als belegt angesehen.

Ein grundlegendes Schema für die meisten offenen Hashverfahren ergibt sich aus folgenden Grundaussagen:

- $s(j,k)$ sei eine Funktion, so dass
$$(h(k) - s(j,k)) \mod m$$
für $j = 0, 1, \ldots, m-1$ eine Sondierungsfuntion bildet (Permutation aller Hashadressen)
- Suchen nach k: Man beginnt bei $i = h(k)$. Wenn k nicht an dieser Stelle ($t[i]$) gespeichert und $t[i]$ nicht frei ist, fährt man bei
$$i = (h(k) - s(j,k)) \mod m$$
(mit steigendem j) fort. Wenn $t[i]$ belegt ist, war die Suche erfolgreich, ansonsten erfolglos.
- Einfügen eines k: Durch Suche wurde festgestellt, dass k nicht in t vorkommt. Begonnen wird nun bei Hashadresse $i = h(k)$. Solange aber diese Stelle der Hashtabelle belegt ist, wird fortgefahren mit
$$i = (h(k) - s(j,k)) \mod m$$
für steigende j. Wenn $t[i]$ frei ist, so trage k an dieser Stelle ein.
- Entfernen eines k: Die Suche verlief erfolgreich. Die Adresse an der k gefungen wurde sei i. Damit wird $t[i]$ als entfernt markiert.

In den folgenden Abschnitten (5.1 bis 5.7) wird nun auf verschiedene offene Hashverfahren eingegangen.

5.1 Lineares Sondieren

Beim linearen Sondieren zählt die Sondierungsfolge:

$$h(k), h(k)+1, h(k)+2, \ldots, 0, m-1, \ldots, h(k)+1$$

und somit die Sondierungsfunktion: $s(j,k) = j$. Das lineare Sondieren ist ein sehr einfaches Verfahren, hat aber auch einige Nachteile. So ist die Wahrscheinlichkeit für neu einzufügende Schlüssel an bestimmten Hashadressen gespeichert zu werden sehr unterschiedlich. Da lang belegte Teilstücke eine höhere Tendenz zu wachsen haben als kleinere und dazu kommt noch, dass lang belegte Bereiche durch zusammenwachsen noch länger werden. Aus dieser *Primären Häufung* heraus verschlechtert sich die Effizenz sehr stark, wenn sich α (Belegungsfaktor) dem Wert 1 annähert.

5.2 Quadratisches Sondieren

Um der Primären Häufung zu entgehen, wird beim quadratischen Sondieren in quadratisch wachsendem Abstand um k nach einem freien Platz gesucht. Damit ergibt sich die Sondierungsfolge

$$h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \ldots$$

und die Sondierungsfunktion $s(j,k) = ([j/2])^2(-1)^j$. Wenn m im Weiteren eine Primzahl der Form $4i+3$ ist, so ist belegt, dass die Sondierungsfolge eine Permutation von $t[0\backslash m-1]$ bildet. Durch diese Formeln wurde zwar die primäre Häufung beseitigt, doch es kommt zu einem neuen Phänomen, der *sekundären Häufung*. Zwei Synonyme k und k' durchlaufen die gleiche Sondierungsfolge, behindern sich also gegenseitig auf ihren Plätzen. Beim linearen Sondieren, welches zuvor in Kapitel 5.1 behandelt wurde, war dies ebenfalls schon der Fall.

5.3 Uniformes und Zufälliges Sondieren

Die Probleme der primären bzw. sekundären Häufung liegen in der Unabhängigkeit der Sondierungsfunktion von k. Zwei Synonyme durchlaufen immer die selbe Folge und behindern sich so gegenseitig. Im Idealfall würden die Schlüssel mit gleicher Platzwahrscheinlichkeit auf einem Platz in der Hashtabelle landen. Diesen Fall ermöglicht das *Uniforme Sondieren*. Die Sondierungsfunktion $s(j,k)$ ist eine nur von k abhängige Permutation der Hashadressen, mit einer gleichen Wahrscheinlichkeit für alle $m!$ möglichen Permutationen. Die Kollisionen werden durch diese Methode beim Einfügen minimiert, die Bedingungen für das Einfügen sind also optimal. Der Nachteil des uniformen Sondierens ist die schwierige praktische Umsetzung. Daher wird eher das *zufällige Sondieren* genutzt, welches nahe die gleiche Effizienz hat. Es wird eine zufällige Hashadresse für $s(j,k)$ gewählt, welche jedoch abhängig von k ist. So kann es zwar in Einzelfällen dazu kommen, dass ein belegter Platz nochmal betrachtet wird, da es keine Permutation aller Hashadressen darstellt, allerdings ist dies nicht der Regelfall.

5.4 Double Hashing

Die Effizienz des im letzten Kapitel beschriebenen uniformen Hashing, kann in einer ähnlichen Form auch schon erreicht werden, wenn keine Permutation als Sondierungsfolge, sondern eine zweite Hashfunktion genutzt wird Die gewählte Sondierungsfolge kann dann mit

$$h(k), h(k)-h'(k), h(k)-2h'(k), \ldots, h(k)-(m-1)h'(k)$$

jeweils mit $\mod m$ beschrieben werden, wenn man für $h'(k)$ eine zweite Hashfunktion einsetzt. Daraus ergibt sich dann die Sondierungsfunktion: $s(j,k) = j \cdot h'(k)$. Bei der Wahl von $h'(k)$ muss beachtet werden, dass die Sondierungsfolge eine Permutation der Hashadressen bildet, und das für alle Schlüssel k. Die Hashfunktion muss demnach ungleich der Zahl 0 sein ($h'(k) \neq 0$) und darf m ebenfalls nicht teilen. Demnach wäre es das Beste, wenn m als Primzahl gewählt wird, dann gelten diese beiden Bedingungen und die Auswahl wäre ebenfalls gut für die Anwendung der Division-Rest-Methode (vgl. Kapitel 3.1). Wäre nun $h(k)$ abhängig von $h'(k)$, so würde sich die Sondierungsfolge für einige Elemente wiederholen, dies kann man vermeiden, indem man $h'(k)$ unabhängig auswählt. Man nutzt dann für zwei verschiedene Schlüssel k und k' folgende Gleichungen:

$$p[h(k) = h(k')$$

und

$$h'(k) = h'(k')] = p[h(k) = h(k')] \cdot p[h'(k) = h'(k')]$$

wobei p die Wahrscheinlichkeit für das Gelten der Bedingung angibt. In Worten ausgedrückt bedeutet das: SSind zwei Schlüssel Synonyme bezüglich h, so sind sie mit Wahrscheinlichkeit $1/m^2$ Synonyme bezüglich h' (m' ist die Anzahl der Werte, die die Funktion annehmen kann), also sind zwei Schlüssel mit Wahrscheinlichkeit $1/(m \cdot m')$ Synonyme bezüglich h und h' gleichzeitig."[4] Wählt man m als Primzahl, so ist leicht feststellbar, dass die Gleichungen $h(k) = \mod m$ und $h'(k) = 1 + k \mod (m-2)$ die gerade genannten Anforderungen erfüllen. Im Folgendem soll nun noch auf ein Verfahren eingegangen werden, welches als Beispiel für double Hashing angesehen werden kann.

Binärbaum-Sondieren Ein Schlüssel k kann nicht an der Adresse $h(k)$ in die Hashtabelle eingetragen werden, weil sich dort schon der Schlüssel k befindet. Der Weg der nun weiter gegangen wird, teilt sich in zwei Möglichkeiten.

1. k' bleibt an seinem Platz und für k wird ein neuer Platz gesucht (gemäß der Sondierungsfolge)
2. k' wird durch k am Ausgangsplatz ersetzt und für k' wird ein neuer Platz gesucht (gemäß der Sondierungsfolge)

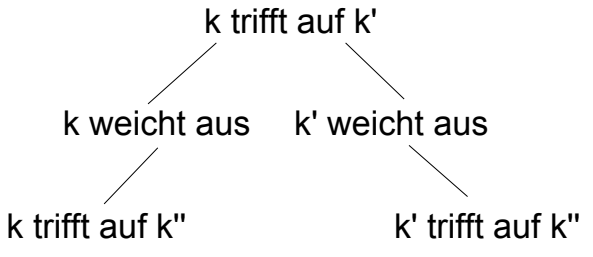

Figure 1: Anschaung einer Binärbaumsondierung mit zwei Überprüfungen

Wenn sich nun auf eine der beiden Möglichkeiten ein neuer Platz finden lässt, so wird der zugehörige Schlüssel an dieser

Stelle eingetragen und das Einfügen vom Schlüssel k ist beendet. Sollte dies nicht der Fall sein, so werden beide Wege analog verfolgt und es ergibt sich für jede neue Überprüfung eine neue Sondierungsfolge (in Form eines Binärbaumes).

5.5 Ordered Hashing

Wie der Name ordered (dt. geordnet) schon verrät, sollen bei diesem Verfahren die Schlüssel in einer bestimmten Reihenfolge in die Hashtabelle eingetragen werden. Die Synonyme sollen als so eingefügt werden, dass sie in sortierter Reihenfolge der Sondierungsfolge von k eintreffen. Dabei kann selbst festgelegt werden, ob dies auf- oder absteigend geschehen soll. Die Sondierungsfolge soll folgendermaßen beschrieben werden: $(h(k) - s(j,k)) \mod m$, mit geltendem $j = 0, 1, 2, \ldots, m-1$. Ebenso wird davon ausgegangen, das mindestens ein Platz in der Hashtabelle immer frei ist.

- Suchen nach k:
  - Man beginnt bei $i = h(k)$. Wenn k nicht in $t[i]$ gepeichert und $t[i]$ frei ist und weiterhin gilt, dass der Schlüssel der in $t[i]$ einen kleineren Wert hat als k, so wird in $i = (h(k) - s(j,k)) \mod m$ mit steigendem j weitergesucht. Ist $t[i]$ allerdings belegt und der Schlüssel in $t[i]$ gleich k, so wurde k gefunden und die Suche war erfolgreich.
- Einfügen von k:
  - Natürlich wird angenommen, dass k nicht in t gespeichert ist, denn sonst müsse k ja nicht eingefügt werden. Und wie bei der Suche wird mit $i = h(k)$ begonnen. Ist die Hashadresse bei $t[i]$ belegt, und k kleiner als der Schlüssel, der in $t[i]$ gespeichert ist, so vertausche die beiden Schlüssel. Fortgefahren wird mit dem nächsten i welches nun zu dem getauschten Schlüssel k gehört.Der Schlüssel k wird dann in der nächsten so findbaren freien Adresse eingetragen.
- Entfernen eines k:
  - Wird weiterhin so fortgeführt wie bisher.

5.6 Robin-Hood-Hashing

Das nun beschriebene Robin-Hood-Hashing ordnet ebenfalls wie das *Ordered Hashing* die Schlüssel beim Einfügen in die Hashtabelle um. Das Ziel dabei ist es, die längste Sondierungsfolge zu verringern. Der Fortgang beim Einfügen von k hat folgende Abfolge: man beginnt wie eben schon zuvor an der Stelle $i = h(k)$. Ist die Adresse $t[i]$ belegt, so wird die relative Position j von i in der Sondierungsfolge von k, mit der relativen Position j' der Adresse i in der Sondierungsfolge von k' verglichen. Ist die relative Adresse j' > j, so wird mit $i = (i - h'(k')) \mod m$ fortgefahren. In Folge dessen ist $t[i]$ frei und so kann k an dieser Stelle eingetragen werden. Bei diesem Verfahren werden jedoch nicht die durchschnittlichen Sondierungsfolgenlängen geändert, sondern die Längen der Folgen lediglich aneinander angepasst.

5.7 Coalesced Hashing

Die Prinzipien von offenem Hashing und Hashing mit Verkettung der Überläufer werden beim nun folgenden Verfahren verbunden. Die Überläufer befinden sich in einer Überläuferkette, welche in der Hashtabelle abgespeichert ist. Jeder Hashtabelleneintrag besteht aus einem Schlüssel mit zugehörigem Datensatz und einem Zeiger (in Form einer Hashadresse) welche auf den nächsten Überläuferketteneintrag zeigt, ebenso besitzt der Eintrag die üblichen Markierungen eines offenen Hashverfahrens. Ein neu einzufügender Schlüssel wird am Ende der Überläuferkette angefügt.

- Suchen nach k:
  - Man beginnt bei $t[h(k)]$ und folgt den Zeigern auf die Überläuferkette. Die Suche ist erfolgreich, wenn k gefunden wurde und erfolglos, wenn das Ende der Überläuferkette erreicht wurde.
- Einfügen von k:
  - Eine erfolglose Suche ging voraus, sie endete bei $t[h(k)]$. Dort endet sie entweder auf einem nicht belegten Feld (Fall a) oder am Ende der Überläuferkette (Fall b). In Fall b wird nun das frei Hashtabellenelement mit größter Hashtabelle ausgewählt und dort an die Überläuferkette angehangen. In Fall a wird k in das unbelegte Feld eingetragen, an welchem die Suche endete.
- Entfernen eines k:
  - Nach einer erfolgreichen Suche wird k als gelöscht markiert.

Wie aus diesen Verfahren erkennbar ist, ist dies eine Vermischung der am Anfang dieses Abschnitts benannten Prinzipien, wenn man das Auswählen eines freien Eintrages außer Acht lässt.

6. AKTUELLE HASHVERFAHREN

In den nächsten drei Abschnitten möchte ich nun auf aktuellere Hashverfahren eingehen. Dabei sind die Verfahren ab Abschnitt 6.2 Vertreter von krytographischen Hashverfahren, welche dazu genutzt werden um Prüfsummen zu bilden, mit denen man testen kann, ob an einer Datei Veränderungen vorgenommen wurden.

6.1 Kuckucks-Hashing

Im Jahre 2001 wurde von Rasmus Pagh und Flemming Friche Rodler ein neues Hashverfahren vorgestellt, es wurde Kuckucks-Hashing (bzw. im Englischen Cuckoo Hashing) genannt. Gebraucht werden für dieses Verfahren zwei Hashverfahren und zwei Tabellen. Soll nun ein Schlüssel k abgespeichert werden, so wird zunächst überprüft, ob der Platz der Tabelle 1 t_1 an der Adresse von $h(k)$ frei ist. Ist dies der Fall, so kann k an dieser Stelle eingefügt werden und es sind keine weiteren Operationen nötig. Ist das Einfügen an dieser Stelle jedoch nicht möglich, so wird geprüft ob die Adresse von $h'(k)$ in Tabelle 2 frei ist. Ist diese belegt, so wird der Schlüssel k in die erste Tabelle eingefügt und das dort befindige Element k' wird in die zweite Tabelle verschoben. Ist diese Adresse nun erneut belegt, so wird das Element wieder in die erste Tabelle verschoben. Dieser Vorgang findet solange statt, bis ein freier Platz gefunden wird und das Einfügen so beendet werden kann. Bei diesem Verfahren kann es allerdings passieren das dieser Zyklus in einer Endlosschleife endet, dann müssen die Tabellen mit zwei neuen Hashfunktionen neu aufgebaut werden ([3] und [6]).

6.2 SHA-3

Im Jahre 2007 rief das US National Institute of Standards and Technology (NIST) dazu auf, dass Kandidaten ihre Vorschläge für den neuen SHA-3 Algorithmus einsenden sollten, da befürchtet wurde, dass SHA-2 bald mit den neuen kryptoanalystischen Verfahren geknackt werden könne. Am Ende des Wettbewerbs wurde "Keccak"von Guido Bertoni, Joan Daemon, Gilles Van Assche und Michaël Peeters als Gewinner und damit als SHA-3 ausgewählt [2]. Dieses neue Verfahren funktioniert auf folgende Art und Weise: Genutzt wird ein mit 0 initialsierter Zustandvektor, welcher aus 25 Wörtern besteht mit jeweils w Bit. Der Parameter w des Verfahrens kann dabei jede Zweierpotenz zwischen eins und 64 annehmen. Ein zweiter Paramter n beschreibt die Bitlänge des gewünschten Hash-wertes (also der Prüfsumme). In der Wettbewerbsvariante besaß w den Wert 64 und n konnte zwischen 224 und 512 Bit lang sein. Die Nachricht, zu welcher die Prüfsumme gebildet werden soll, wird in r-bitlange Abschnitte geteilt, wobei $r = 25w - 2n$ gilt. Diese Teilung ist natürlich erst möglich, wenn sie auf ein Vielfaches von r erweitert wurde, mit Hilfe der Anfügung der Bitfolge $100\ldots01$, wobei die Anzahl der Nullen natürlich abhängig von der natürlichen Länge der Nachricht ist und zwischen 0 und $r - 1$ liegt.

6.3 MD5

MD5 ist ein von Ronald Linn Rivest im Jahre 1991 entwickeltes Hashverfahren, welches heutzutage weit verbreitet ist. Aus diesem Grund möchte ich es auch in diese Betrachtung mit einbeziehen, obwohl es mitlerweile nicht mehr als sicher gilt. MD5 erzeugt aus einer beliebig langen Nachricht eine Prüfsumme der Länge 128 Bit. Dabei wird folgendes Vorgehen angewandt: An die Ausgangsnachricht wird eine Eins angehangen, nun wird die Nachricht mit Nullen so aufgefüllt, dass die durch 512 teilbar wäre, wenn man 64 Bit hinzufügen würde. An diese Nullen wird dann eine 64-Bit-Zahl angehangen, die für die Codierung der Ausgangsnachrichtenlänge zuständig ist, und so ist die Nachrichtenlänge nach der Bearbeitung durch 512 teilbar. Der Hauptalgorithmus besteht dann aus einem 128-Bit-Puffer, welcher wiederum in vier 32-Bit-Worte unterteilt ist, diese seien A, B, C und D. Diese Wörter werden dann weiter mit bestimmten Konstanten initialisiert. Der Puffer wird im nächsten Schritt mit einer Komprimierungsfunktion aufgerufen. Der Nachrichtenblock wird so in 4 Runden behandelt, wobei jede aus 16 Operationen, basierend auf vier unterschiedlichen nichtlinearen Funktionen F, besteht. In jeder Runde wird dann eine der folgenden Funktionen angewandt:

$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$$
$$H(X,Y,Z) = X \oplus Y \oplus Z$$
$$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$$

Dabei gelten folgende Operationen: $\wedge$ ist UND, $\vee$ ist ODER, $\neg$ ist NOT und $\oplus$ steht für XOR.

7. ZUSAMMENFASSUNG

Schaut man sich die verschiedenen Verfahren des Hashings an, so wird deutlich, dass sie einer stetigen Entwicklung unterliegen. Es werden immer neue Nachteile entdeckt, welche aber ebenso durch neue Forschung wieder versucht werden durch bessere und vorallem effizientere Verfahren vermieden zu werden.

Von den beiden Hauptgruppen welche vorgestellt wurden (Verkettung der Überläufer und offene Hashverfahren) kann nicht gesagt werden, dass eines der beiden besser wäre, da beide Vor- und Nachteile haben, so dass man unterschiedlich von Anwendungsgebiet zu Anwendungsgebiet unterscheiden sollte, welches Verfahren für die eigene Nutzung komfortabler und effizienter ist. Was man sich zum Thema Hashverfahren merken sollte ist die unterschiedliche Speicherung der Schlüssel, also ob die Schlüssel innerhalb der Hashtabelle mit Hilfe von Sondierungsfolgen gespeichert werden oder ob sie außerhalb der Hashtabelle in Überläuferketten in Form von linearen Listen gespeichert werden. Die erfolgreiche Speicherung von Synonymen ist sicher eine der Hauptaufgaben des Hashing, da es das Ziel ist das die Verfahren so optimiert werden, dass möglichst wenige bis im besten Falle keine Kollisionen auftreten

Ebenso wie die Informatik wird sich das Gebiet des Hashing immer weiter entwickeln, da immer mehr Datenmengen elektronisch gespeichert werden und somit immer neue Wege ihrer Verwaltung gefunden werden müssen. Die Entwicklung des Hashing ist vor allem bei krytographischen Verfahren nötig, da es durch immer schnellere und bessere Rechner möglich ist, in kürzerer Zeit die verschiedensten Möglichkeiten durchzutesten und damit das Verfahren zu knacken.

Danksagung

8. REFERENCES

[1] A. Beutelspacher, H. B. Neumann, and T. Schwarzpaul. *Kryptographie in Theorie und Praxis*, volume 1. Vieweg, 2005.

[2] J. Ihlenfeld. Keccak Hash-Algorithmus für SHA-3 festgelegt. 10 2012.

[3] E. W. Mayr. Effiziente Algorithmen und Datenstrukturen. online, Wintersemester 2007/2008. Fakultät für Informatik TU München.

[4] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*, volume 5. Spektrum Akademischer Verlag, 2012.

[5] C. Schindelhauer. 34 algorithmus der woche hashing. Online, 2006.

[6] C. Weidling. Platzeffiziente Hashverfahren mit garantierter konstanter Zugriffszeit. Master's thesis, Technische Universität Ilmenau, 2004.

Selbstständigkeitserklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde. Keine anderen als die angegebenen Quellen und Hilfsmittel wurden benutzt. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Görlitz, 10. Februar 2013 Dana Müller