Prof. Dr. Sebastian Iwanowski

Informatik-Seminar Algorithmen

Zeit vs. Platz II

Hashing

Claas Casper (mi2144)

Inhaltsverzeichnis

1. Einführung

Hashing kommt vom englischen „to hash“, was so viel wie „zerhacken“ bedeutet. Dieser Name ist in sofern etwas irreführend, da es beim Hashing nicht darum geht, Daten wild zu zerhacken, sondern sie gezielt zu komprimieren.
Das Ziel beim Hashing ist es, eine beliebig große, beziehungsweise unendliche Quellmenge auf eine Zielmenge bestimmter, endlicher Größe abzubilden.
Dies geschieht mit Hilfe einer Hash-Funktion, auch Streuwertfunktion genannt. Dabei handelt es sich um eine arithmetische Funktion, die ein Element der Quellmenge auf einen skalaren Wert in der Zielmenge abbildet. Dieser wird Hash-Wert genannt.

Ein einfaches Beispiel für eine Hash-Funktion ist die einstellige Quersumme. Hierbei besteht die unendliche Quellmenge aus allen natürlichen Zahlen, die Zielmenge sind folglich die Zahlen von 0 bis 9 (wobei die 0 nur einmal als Quersumme vorkommt).

Welche Hash-Funktion benutzt wird, ist von der Anwendung abhängig. Eine gute Hash-Funktion sollte jedoch folgende Kriterien erfüllen:

- *Datenreduktion*: Der Speicherbedarf des berechneten Hash-Wertes sollte deutlich geringer sein als der des Original-Wertes.

- *Chaotisch*: Die Funktion sollte so konzipiert sein, dass ähnliche Eingabewerte zu völlig verschiedenen Hash-Werten führen. Im oben genannten Beispiel der Quersumme ist dies nicht der Fall. Hier verändert das Vertauschen der Positionen der Ziffern einer Zahl nicht den Hash-Wert. So wird zum Beispiel 15 genau wie 51 auf 6 abgebildet. In einer praktischen Anwendung sollte jedoch Idealerweise das Kippen eines Bits im Quellwert dazu führen, dass sich die Hash-Werte in durchschnittlich der Hälfte ihrer Bits unterscheiden.

- *Surjektiv*: Kein Wert der Zielmenge sollte von vornherein ausgeschlossen sein. Zwar kann es sein, dass einige Hash-Werte nicht vorkommen, da keine entsprechenden Elemente der Quellmenge genutzt werden, die auf sie abgebildet werden. Es sollte jedoch theoretisch die Möglichkeit bestehen, dass durch andere Quellelemente die Hash-Werte erreicht werden.

- *Effizienz*: Die Funktion sollte schnell berechenbar sein, einen möglichst geringen Speicherverbrauch haben und auf die Quelldaten möglichst nur einmal lesend zugreifen.

Leider kann man auch bei Einhaltung der oben genannten Kriterien ein Problem, das Hash-Funktionen grundsätzlich mit sich mitbringen, nicht völlig ausschließen. So sind Hash-Funktionen von Natur aus nicht injektiv. Eine Funktion ist dann injektiv, wenn jedes Element der Zielmenge höchstens einmal als Funktionswert vorkommt. Dies ist zum Beispiel bei $y(x) = x^2$ der Fall. Aus der Definition des Hashing, dass eine große Datenmenge auf eine kleinere abgebildet wird, ergibt sich allerdings logisch, dass mehreren Elementen der Quellmenge derselbe Hash-Wert zugeordnet werden kann. Diese Fälle nennt man beim Hashing Kollisionen.

2. Hash-Tabellen

Hash-Tabellen sind ein der Hauptanwendungsgebiete für Hashing. Es handelt sich um eine Möglichkeit Daten so in Felder zu speichern, dass man theoretisch eine Zugriffszeit von O(1) hat.
Dabei wird die Position im Array durch den Hash-Wert der Daten bestimmt. Üblicherweise handelt es sich bei den Daten um Records, die einen sie eindeutig identifizierenden Schlüssel besitzen. Von diesem wird der Hash-Wert gebildet, der die Position repräsentiert. So kann dieses Element theoretisch mit einem Zugriff, ohne sequentielle Suche, gefunden werden.
Um beispielsweise ein Wörterbuch zu realisieren, kann der Suchbegriff als Schlüssel verwendet werden. Ein Beispiel, um einen Hash-Wert für einen String zu berechnen wäre, die Position der einzelnen Buchstaben im Alphabet als Wert zu nehmen und zu summieren:

HALLO = 8+1+12+12+15 = 48
WELT = 23+5+12+20 = 60

Da man so allerdings immer noch auf eine unendliche Zielmenge kommt, braucht man noch eine Berechnung, die die Zielmenge endlich werden lässt. In der Praxis wird hierfür meist der Divisionsrest modulo benutzt.
So ergibt sich allgemein die Position eines Elements im Array als h(k) mod m, wobei h(k) die Funktion ist, die aus dem Schlüssel einen skalaren Wert berechnet, und m die Größe des Arrays.
Wählt man also ausgehend vom obigen Beispiel als Arraygröße 10, ergibt sich für den Schlüssel HALLO die Position 48 mod 10 = 8 und für WELT 60 mod 10 = 0.

Diese Form der Berechnung für den Hash-Wert eines Strings erfüllt jedoch das Kriterium, dass eine gute Hash-Funktion chaotisch sein sollte, nicht unbedingt. Denn das einfache Vertauschen von Buchstabenpositionen verändert in diesem Fall nicht den Hash-Wert. So ergibt sich für WELT der gleiche Wert wie für TLEW. Eine Alternative ist die Position eines Buchstabens mit einzubeziehen und die Berechnung nach dem Horner-Schema vorzunehmen, wobei von einem Zahlensystem zur Basis 26 ausgegangen werden kann (26 Buchstaben im Alphabet). So wäre der Wert für WELT $((23 * 26^3) + (5*26^2) + (12 * 26) + (20 * 1)) \bmod 10$, bzw. nach dem Horner-Schema $(((23*26+5)26+12)26+20) \bmod 10 = 0$ und für TLEW $(((20*26+12)26+5)26+23) \bmod 10 = 5$.
In Java gibt es für jede Klasse die Funktion hashCode(), die einen Skalarwert nach einer für die Klasse beschriebenen Hash-Funktion berechnet. Für Strings wird dort eben das Horner-Schema benutzt, allerdings auf Basis der Ascii-Werte.

Wie bereits erwähnt, bringen Hash-Funktionen das Problem der Kollisionen mit sich. Würde zum Beispiel oben für die Größe des Arrays nicht der Wert 10 gewählt, sondern 12, ergäbe sich (nach der einfachen Summenmethode) sowohl für den Schlüssel HALLO als auch für WELT die Position 0. Andererseits würde bei der Arraygröße 10 ein weiterer Eintrag mit dem Schlüssel ICH ebenfalls die Position 0 erhalten (9+3+8 = 20; 20 mod 10 = 0).
Zwar kann man mit einer guten Hash-Funktion und einer entsprechend gut gewählten Größe des Arrays die Wahrscheinlichkeit für Kollisionen minimiert werden, ausschließen kann man sie jedoch nie. Deswegen gibt es einige Konzepte für Kollisionslösungen, wovon die zwei meistgenutzten im Folgenden vorgestellt werden.

3. Kollisionslösungen
3.1 Separate Chaining

Die erste Möglichkeit mit Kollisionen in Hash-Tabellen umzugehen, ist das so genannte Separate Chaining, also getrennte oder verteilte Verkettung. Hierbei handelt es sich um eine Kollisionsbehandlung außerhalb der Hash-Tabelle, da das Problem praktisch aus dem Array verschoben wird.
So werden hierbei in die einzelnen Zellen des Arrays nicht direkt die Elemente gespeichert, sondern lediglich Zeiger auf Listen. In diese Listen werden schließlich alle Elemente mit demselben Hash-Wert gespeichert.
Im Einzelnen funktionieren die Methoden wie folgt:

Einfügen: Der Hash-Wert des Schlüssels wird berechnet. Ist die Liste leer, auf die in der berechneten Zelle gezeigt wird, wird der Schlüssel neues erstes Element der Liste und zeigt auf das Leerelement. Ist die Liste nicht leer, wird der Schlüssel als Element ans Ende der Liste gehängt.

Suchen: Es wird ebenfalls der Hash-Wert des Schlüssels berechnet. Enthält die entsprechende Zelle des Arrays lediglich die leere Liste, befindet sich der Schlüssel nicht in der Hash-Tabelle. Ist die Liste nicht leer, muss diese sequentiell durchgegangen und jeweils der Schlüssel direkt mit den gespeicherten Werten verglichen werden bis er gefunden oder das Ende der Liste erreicht ist.
Um die sequentielle Suche zu verbessern, können die Schlüssel auch geordnet eingefügt werden, was allerdings wiederum das Einfügen verlangsamt.

Löschen: Verläuft analog zum Suchen. Ist das Element gefunden, wird durch Neuverlinken des Zeigers des vorhergehenden Elements auf das nachfolgende das gesuchte Element aus der Liste entfernt.

Um die Effizienz beim Suchen zu beschreiben, wird zuerst der Belegungsfaktor der Hash-Tabelle berechnet. Dieser wird mit α bezeichnet und ergibt sich aus dem Quotienten aus der Anzahl der gespeicherten Elemente n und der Größe des Arrays m:

$$\alpha = n/m$$

Im Falle des Separate Chaining zeigt der Belegungsfaktor die durchschnittliche Länge einer Liste an. Sind also 39 Elemente in einem Array der Größe 13 gespeichert, so enthält jede Liste durchschnittlich 3 Elemente.
Für eine erfolgreiche Suche ergibt sich die Anzahl S der durchschnittlich verglichenen Elemente aus

$$S \approx 1 + (\alpha/2)$$

Für eine erfolglose Suche ist die Anzahl U durchschnittlich verglichener Elemente

$$U \approx \alpha$$

da alle Elemente einer Liste durchlaufen werden müssen.

Wir sehen, dass entgegen dem theoretischen Ansatz der Hash-Tabellen, doch sequentielle Suchen durchgeführt werden müssen. Im Vergleich zu einer einzigen Liste ist die Geschwindigkeit jedoch um den Faktor m, also der Größe des Arrays, höher, da wir statt einer Liste der Länge n, m Listen der Länge α = n/m haben. Andererseits muss hier mehr Speicher alloziert werden, nämlich für jede Liste auf jeden Fall ein Zeiger.
Ideal wäre deshalb ein Belegungsfaktor von α ≈ 1. Ist der Belegungsfaktor zu klein, haben wir viele leere Zellen/Listen, was eine Speicherverschwendung darstellen würde. Ist der Belegungsfaktor zu hoch, werden die einzelnen Listen zu lang und damit die durchschnittlichen Suchzeiten länger.
Da die Hash-Tabelle beim Separate Chaining nicht vollaufen kann, bietet sich die Verwendung vor allem an, wenn man die Anzahl der zu speichernden Daten im Vorfeld nicht genau abschätzen kann.

3.2 Open Addressing

Die zweite weit verbreitete Kollisionslösung für Hash-Tabellen ist das Open Addressing. Hierbei wird die Kollisionsbehandlung in der Hash-Tabelle selbst vorgenommen. Es gibt also keine Listen und die Daten sind direkt in den Zellen des Arrays gespeichert. Daraus lässt sich schließen, dass die Größe m des Arrays mindestens genauso groß wie die Anzahl der Werte n sein muss. Wir werden aber sehen, dass m in der praktischen Anwendung noch größer sein sollte.
Dadurch ergibt sich auch schon, dass diese Methode eher genutzt werden kann, wenn die Anzahl der Daten im Vornherein abgeschätzt werden kann und man eine Arraygröße von m > n vertreten kann. Eine weitere Einschränkung ist allerdings, dass sich das Benutzen von Open Addressing nur lohnt, wenn die Größe eines Datums, dass gespeichert werden soll, nicht größer ist als die eines Pointers. Denn in diesem Fall kann auch die „billigere“ Version Separate Chaining verwendet werden.

Im Falle von Kollisionen wird beim Open Addressing einfach ein anderes freies Feld gesucht. Aus diesem Umstand, nämlich dass zwei Elementen mit gleichem Hash-Wert zwei verschiedene Adressen zugeordnet werden, ergibt sich die Bezeichnung des Open Addressing, also der offenen Adressierung.

Eine andere Bezeichnung für das Verfahren ist Closed Hashing, welche auch bei Levitin benutzt wird. Dies bezieht sich auf die begrenzte Anzahl der Elemente, die die Tabelle in diesem Fall aufnehmen kann. Andererseits wird in anderen Quellen vom Open Hashing gesprochen, bezogen auf eben die offene Adressierung. Beim Separate Chaining wird entsprechend jeweils je nach Quelle die andere Bezeichnung verwendet. Um also Verwirrung zu vermeiden, sollten eher die Begriffe Separate Chaining und Open Addressing verwendet werden.

Das Finden eines anderen Feldes im Falle einer Kollision wird realisiert durch das Weiterspringen um eine bestimmte Schrittweite bei jeder Kollision. So ergibt sich die allgemeine Formel

$$h_{i(k)} = (h_{(k)} + s*i) \bmod m$$

wobei s die Schrittweite und i die Tiefe der Kollision ist (also beim ersten belegten Feld 1; ist das nach der Schrittweite erreichte Feld ebenfalls belegt, ist die Kollisionstiefe 2 usw.).

Die einzelnen Methoden lassen sich beim Open Addressing folgendermaßen implementieren:

Einfügen: Ist die Zelle an der berechneten Position leer, kann das Element direkt gespeichert werden. Ist sie belegt wird rekursiv um die Schrittweite weitergegangen, bis eine leere Zelle gefunden wurde. Dort wird das Element gespeichert.
Um zu prüfen, ob eine Zelle belegt ist, kann man sich vorstellen den Belegungsstatus einfach mit Hilfe eines boolean-Wertes zu speichern.

Suchen: Es wird ebenfalls zuerst der Hash-Wert berechnet und die Zelle an der entsprechenden Position untersucht. Ist diese leer, ist das Element nicht enthalten. Ist die Zelle belegt, müssen die Schlüssel verglichen werden. Sind diese unterschiedlich wird jetzt analog zum Einfügen um die Schrittweite weiter gegangen bis entweder der gesuchte Schlüssel oder eine leere Zelle gefunden wird.
Die Suche kann nicht abgebrochen werden, wenn ein Element mit anderem Hash-Wert gefunden wird, da dahinter trotzdem noch Elemente mit dem gleichen Hash-Wert wie dem gesuchten gespeichert werden soll.
Beispiel: Ein Element mit Hash-Wert 0 wird auf Position 0 gespeichert. Anschließend ein Element mit Hash-Wert 1 auf Position 1. Wird nun ein weiteres Element mit Hash-Wert 0 gespeichert, wird dies hinter dem Element mit dem Wert 1, nämlich auf Position 2 gespeichert.

Löschen: Spontan könnte man überlegen, den Belegungsstatus einer Zelle einfach auf unbelegt zu setzen um das darin befindliche Element zu löschen. Daraus ergibt sich jedoch ein großes Problem. Eine Suche nach einem anderen Element mit demselben Hash-Wert würde an dieser Stelle abbrechen!
Beispiel von oben: Es befinden sich auf der Position 0 ein Element mit dem Hash-Wert 0, auf Position 1 eines mit dem Wert 1 und auf Position 2 ein weiteres mit 0. Nun wird das erste Element nach oben genannter Methode entfernt. Eine Suche nach dem Element an Position 2 würde bei Position 0 beginnen, da der Hash-Wert des Schlüssels 0 ist. Da die Zelle 0 aber leer ist, wird, wie beim Suchen beschrieben, hier bereits abgebrochen.
Hierzu gibt es nun zwei Lösungen. Die umständliche Art des Löschens wäre, alle nachfolgenden Elemente zu überprüfen, ob sie den Hash-Wert gleich der Position des entfernten haben. Wird das erste gefunden, wird dieses auf die Position des gelöschten Elements verschoben. Nun muss von der alten Position des verschobenen Elements aus wieder geguckt werden, ob ein folgendes einen passenden Hash-Wert hat. Dies muss wiederholt werden, bis eine bereits vor dem löschen leer gewesene Zelle gefunden wird.
Eine sehr viel einfacher zu implementierende Methode ist die so genannte „lazy deletion“. Dabei wird der Belegungsstatus um einen dritten Zustand erweitert. Dieser Gelöscht-Zustand wird je nach Methode verschieden behandelt. Beim Suchen wird die Zelle als belegt angesehen, allerdings muss ein Vergleich der Schlüssel immer negativ ausfallen, falls der gelöschte Schlüssel gesucht wird. Beim Einfügen kann die Zelle jedoch als leer angesehen werden, sodass neue Elemente an dieser Stelle eingefügt werden können.

Für die Wahl der Schrittweite beim Open Addressing gibt es verschiedene Möglichkeiten.

3.2.1 Lineares Sondieren

Die erste Möglichkeit ist, eine konstante Schrittweite zu wählen. Dies wird lineares Sondieren bzw. linear probing genannt. In der Praxis wird meistens die Schrittweite 1 gewählt. Dies führt beim Einfügen von s = 1 in die allgemeine Funktion zu der Formel

$$h_{i(k)} = (h_{(k)} + i) \bmod m$$

So wird also einfach bei jeder Kollision die Kollisionstiefe zum errechneten skalaren Wert addiert und somit jeweils in das nachfolgende Feld gesprungen.

Die wiederum mit Hilfe des Belegungsfaktors angegebene Formel für die Anzahl der Vergleiche bei einer erfolgreichen Suche ist

$$S \approx \tfrac{1}{2} * (1 + 1/(1-\alpha))$$

Für eine erfolglose Suche ergibt sich

$$U \approx \tfrac{1}{2} * (1 + 1/(1-\alpha)^2)$$

So kommt man auf folgende Suchzeiten bei bestimmten Belegungsfaktoren:

α	S	U
50%	1,5	2,5
75%	2,5	8,5
90%	5,5	50,5

Man erkennt, dass es einen Effizientverlust gibt, je mehr die Tabelle gefüllt ist. Dies kommt dadurch, dass sich durch die konstante Schrittweite von 1 Cluster bilden. Ein Cluster ist eine Folge von belegten Zellen. Je größer so ein Cluster ist, desto größer ist auch die Wahrscheinlichkeit, dass ein neues Element beim Einfügen an den Cluster gehängt wird und diesen noch weiter vergrößert.
Beispiel: In einem Array von der Größe 10 sind die ersten 7 Elemente belegt. Die Wahrscheinlichkeit dass ein neu eingefügtes Element nun einen dieser 7 Hash-Werte erreicht liegt also bei 70%. Ist dies der Fall, wird es auf das erste freie Feld gespeichert, also das erste nach dem Cluster, was diesen größer werden lässt.

3.2.2 Quadratisches Sondieren

Eine Verbesserung im Vergleich zum linearen Sondieren stellt das quadratische Sondieren da. Dabei wird keine konstante, sondern eine quadratisch anwachsende Schrittweite gewählt. Zusätzlich wird noch ein Richtungswechsel realisiert. Die Formel dafür ist

$$h_{i(k)} = (h_{(k)} ((-1)^i+1) + (\text{ceil}(i/2))^2) \bmod m$$

Dabei wird der Richtungswechsel durch den Term ((-1)^i+1) realisiert, der dafür sorgt, dass abwechselnd mit -1 oder +1 multipliziert wird. Das aufrunden von i/2 sorgt dafür, dass zwei aufeinander folgende i zum selben Wert für $(i/2)^2$ führen.
So kommt es mit steigender Tiefe der Kollision zu den Schrittweiten +1, -1, +4, -4, +9, -9, +16, -16 usw.

Bei Erstkollisionen kommt es so zu keiner Verbesserung, da weiterhin nur um ein Feld weiter gegangen wird. Bei Folgekollisionen kommt es so allerdings zu weniger Clusterbildung.

3.2.3 Double Hashing

Eine weitere Verbesserung um Clustering zu vermeiden ist das doppelte Hashing. Dabei wird die Schrittweite abhängig vom Schlüssel des zu speichernden Elements gebildet. Man benutzt also eine zweite Hash-Funktion $h'_{(k)}$:

$$h_{i(k)} = (h_{(k)} + h'_{(k)} * i) \bmod m$$

Dabei sollte die zweite Hash-Funktion $h'_{(k)}$ so gewählt werden, dass die entstehende Schrittweite einerseits natürlich nicht 0 wird, und sie andererseits zur Größe m des Arrays prim ist, dass also ihr größter gemeinsamer Teiler 1 ist. Andernfalls könnte man schnell wieder bei der Position der ersten Kollision ankommen, beispielsweise bei m = 12 und einer Hash-Funktion, die Werte zwischen 1 und 6 liefert. In diesem Fall wäre beim Wert 2 nach 6, bei 3 nach 4, bei 4 nach 3 und bei 6 schon nach 2 Kollisionen wieder die Ausgangsposition erreicht.
Um zu garantieren, dass die Werte zueinander prim sind, muss lediglich für die Größe m eine Primzahl gewählt werden und die möglichen Schrittweiten kleiner sein als m.
Zwei in der Praxis oft verwendete Funktionen sind
$h'_{(k)} = 8 - (k \bmod 8)$ für kleinere Tabellen (führt zu Werten zwischen 1 und 8)
und
$h'_{(k)} = k \bmod 97 + 1$ für größere Tabellen (führt zu Werten zwischen 1 und 97).

Die Formeln zur Effizienzberechnung bei doppeltem Hashing sind

$$S \approx (-\ln(1-\alpha)) / \alpha$$

für eine erfolgreiche Suche, und

$$U \approx 1 / (1-\alpha)$$

für erfolgloses Suchen. Daraus ergibt sich folgende Tabelle für Beispielwerte:

α	S	U
50%	1,4	2
75%	1,8	4
90%	2,5	10
99%	4,7	100

Man sieht, dass man beim doppelten Hashing im Durchschnitt weniger Test hat als beim linearen Sondieren, da es hierbei nicht zu Clusterbildung kommt.
So benötigt man kleinere Tabellen um auf dieselben Suchzeiten wie beim linearen Sondieren zu kommen, und kann so Speicher sparen.

3.2.4 Rehashing

Da auch beim doppelten Hashing die Effizienz abnimmt, wenn die Tabelle voll läuft, gibt es die Möglichkeit des Rehashing, auch dynamisches Hashing genannt. Dabei wird die Hash-Tabelle vergrößert, wenn der Belegungsfaktor α gegen 1 läuft. Dadurch muss auch der Wertebereich der Hash-Funktion erweitert werden, um weiterhin alle Zellen des Arrays zu erreichen. Das Problem, dass sich dabei ergibt ist, dass sich theoretisch die Hash-Werte aller bereits gespeicherten Schlüssel ändern und neu berechnet werden müssen. Allerdings gibt es dafür eine Klasse von besonderen, so genannten konsistenten Hash-Funktionen. Diese sind so konzipiert, dass sich beim Erweitern des Wertebereiches nur einige wenige Hash-Werte ändern.

4. Fazit/Anwendung

Zusammenfassend kann man sagen, dass Hash-Tabellen ein guter Kompromiss zwischen Platz und Zeit sind. Hätte man beliebig Platz zur Verfügung, könnte man den Schlüssel direkt – ohne Hash-Wert – seine Position bestimmen lassen. Hätte man hingegen beliebig viel Zeit, könnte man auch eine einzelne Liste mit sequentieller Suche wählen.

Zwar hat das Suchen in Hash-Tabellen im Worst Case eine Komplexität von O(n), im Best Case allerdings O(1). Dies und die gesehenen Probleme beim Einfügen immer neuer Daten, führen zu dem Schluss, dass die Anwendung von Hash-Tabellen dann besonders sinnvoll ist, wenn überwiegend gesucht wird. Also bei einer konstanten Anzahl von Daten in der Tabelle.
In der Praxis werden Hash-Tabellen zum Beispiel bei Symboltabellen in Compilern oder in Datenbanken zum Finden der einzelnen Tabellen angewandt.

Andere Anwendungen für Hashing allgemein sind zum Beispiel Prüfsummen, um Dateien zu vergleichen. Dabei müssen nicht komplette Dateien verglichen werden, sondern lediglich ihre Hash-Werte. Sind diese gleich, sind die Dateien sehr wahrscheinlich (→ Kollisionen) gleich, sind sie verschieden, sind die Dateien auf jeden Fall verschieden.
Eine weitere Anwendung findet sich in der Kryptologie, bei der Verschlüsselung beim Signieren von Dateien.

Beispiele

In dem in Java erstellten Beispielprogramm bestehen zur Vereinfachung die eingefügten Elemente nur aus den Schlüsseln. Dies sind in diesem Fall Strings, die allerdings in einer eigenen Klasse eingepackt sind, um eine eigene Methode hashCode() zu implementieren. Diese liefert hierbei die Summe der Ascii-Werte des Strings zurück. Es werden jeweils die Strings SO LONG AND THANKS FOR ALL THE FISH hinzugefügt. In Klammern hinter dem Schlüssel steht zum Vergleich mit der Position der berechnete Hash-Wert.

1) Array-Größe 13. Jedes Element hat einen anderen Hash-Wert:

```
0: -
1: -
2: THANKS (2)
3: AND (3)
4: THE (4)
5: LONG (5)
6: SO (6)
7: -
8: -
9: ALL (9)
10: FOR (10)
11: -
12: FISH (12)
```

2) Array-Größe 11. Es entstehen Kollisionen, die hier mit Separate Chaining gelöst werden:

```
0: FOR (0)
1: FISH (1)
2: AND (2)
3: -
4: -
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8), ALL (8)
9: -
10: -
```

Es werden zusätzlich die Strings DONT und PANIC eingefügt:

```
0: FOR (0), PANIC (0)
1: FISH (1), DONT (1)
2: AND (2)
3: -
4: -
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8), ALL (8)
9: -
10: -
```

3) Selbes Beispiel wie 2). Diesmal gelöst mit Linearem Sondieren. Beim Einfügen von ALL ist Position 8 bereits belegt, es wird auf 9 die nächste freie Stelle gefunden:

```
0: FOR (0)
1: FISH (1)
2: AND (2)
3: -
4: -
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: ALL (8)
10: -
```

Da 1 ebenfalls belegt ist, wird DON'T auf die freie Position 3 gespeichert. Für PANIC wird anschließend erst auf Position 4 eine leere Zelle gefunden:

```
0: FOR (0)
1: FISH (1)
2: AND (2)
3: DONT (1)
4: PANIC (0)
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: ALL (8)
10: -
```

4) Ausgangssituation aus 2). Das Element FOR wird gelöscht:

```
0: -
1: FISH (1)
2: AND (2)
3: DONT (1)
4: PANIC (0)
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: ALL (8)
10: -
```

PANIC, das den selben Hash-Wert wie das entfernte FOR hat, wird gesucht und dank der „lazy deletion“ gefunden:

```
PANIC: 4
```

Nun wird auch PANIC gelöscht:

```
0: -
1: FISH (1)
2: AND (2)
3: DONT (1)
4: -
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: ALL (8)
10: -
```

PANIC wird neu eingefügt. Dank der „lazy deletion“ wird es auf der richtigen Position gespeichert:

```
0: PANIC (0)
1: FISH (1)
2: AND (2)
3: DONT (1)
4: -
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: ALL (8)
10: -
```

5) Vergleich zwischen selbst implementierter Hash-Funktion (Summe der Ascii-Werte) und in Java vorgegebener für Strings (nach Horner-Schema):

```
0: FOR (0)
1: FISH (1)
2: AND (2)
3: -
4: -
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: ALL (8)
10: -
```

String aus Java:

```
0: THANKS (0)
1: SO (1)
2: FISH (1)
3: -
4: -
5: -
6: FOR (6)
7: AND (7)
8: LONG (8)
9: ALL (8)
10: THE (8)
```

Wird OS eingefügt, erhält man bei normalen Strings einen anderen Wert als für SO:

```
0: FOR (0)
1: FISH (1)
2: AND (2)
3: -
4: -
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: ALL (8)
10: OS (8)
```

```
0: THANKS (0)
1: SO (1)
2: FISH (1)
3: OS (2)
4: -
5: -
6: FOR (6)
7: AND (7)
8: LONG (8)
9: ALL (8)
10: THE (8)
```

6) Selbes Beispiel wie 2). Kollisionen gelöst durch Double Hashing mit der zweiten Hash-Funktion
$h'(k) = 8 - (k \bmod 8)$
ALL landet so nicht auf der Position 9, sondern wird sieben Zellen „weiter“ auf 4 gespeichert:

```
0: FOR (0)
1: FISH (1)
2: AND (2)
3: -
4: ALL (8)
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: -
10: -
```

DONT und PANIC landen ebenfalls auf anderen Positionen:

```
0: FOR (0)
1: FISH (1)
2: AND (2)
3: -
4: ALL (8)
5: THE (5)
6: THANKS (6)
7: LONG (7)
8: SO (8)
9: PANIC (0)
10: DONT (1)
```

Literaturverzeichnis

Bücher:

Anany Levitin: *Introduction to the Design and Analysis of Algorithms*, Addison-Wesley 2006

Robert Sedgewick: *Algorithmen*, 2. Auflage, Pearson Studium 2002

PDF:

Hashverfahren
http://www.ibr.cs.tu-bs.de/courses/ws0708/aud/skript/hash.np.pdf

Sanders / van Stee: Algorithmentechnik – 4. Hashing (Streuspeicherung)
http://algo2.iti.uni-karlsruhe.de/documents/Algorithmentechnik/hash.pdf

HTML:

Wikipedia: Hash-Funktion
http://de.wikipedia.org/wiki/Hash-Funktion

Wikipedia: Hashtabelle
http://de.wikipedia.org/wiki/Hashtabelle

Hashing
http://www.palfrader.org/hashing/