Kapitel 8

Hashing

Hashing ist ein anderes Vorgehen, das auch ohne Baumstrukturen ein effizientes Suchen ermöglicht. Wie bei Bucketsort ist auch hier eine der grundsätzlichen Eigenschaften, dass Hashing nicht auf paarweisen Vergleichen beruht. Beim Hashing werden die Datensätze in einem Array mit Indizes $0, \ldots, m-1$, der sogenannten *Hash-Tabelle* gespeichert.

Wenn wir nun wüssten, dass als Schlüssel nur die Zahlen $0, \ldots, N$, $N \in \mathbb{N}$ vorkommen, könnten wir die Schlüssel direkt als Array-Indizes verwenden. Das ist aber im Allgemeinen nicht erfüllt. Viel öfter tritt der Fall auf, dass die Anzahl der tatsächlichen Schlüssel viel kleiner ist als die Anzahl der möglichen Schlüssel. Daher berechnet man sich beim Hashing einen Array-Index aus dem Schlüssel:

Bezeichne U das Universum aller möglichen Schlüssel und K die Menge der Schlüssel, die auch tatsächlich vorkommen. Dann verwendet man eine *Hash-Funktion* h, $h : U \to \{0, \ldots, m-1\}$, die jedem Schlüssel k einen Index $h(k)$ zuordnet, seine *Hash-Adresse*, und speicher es dort ab. Wenn man das Element dann in der Hash-Tabelle wiederfinden möchte, so berechnet man nach Vorschrift der Hash-Funktion die Hash-Adresse und findet es dann (idealerweise) in $O(1)$ wieder.

Leider ist es im Allgemeinen nicht ganz so einfach: Wenn das Universum der möglichen Schlüssel mehr Elemente enthält als die Hash-Tabelle, kann die Funktion nach dem Schubfachprinzip nicht injektiv sein. Dann gibt es also zwei verschiedene Schlüssel k_1 und k_2, so dass $h(k_1) = h(k_2)$. Das bezeichnet man beim Hashing als *Kollision*.

Der zweite Teil beim Hashing besteht also in der *Kollisionsbehandlung*, die entweder darin bestehen kann, es zu erlauben, auf einer Hash-Adresse mehrere Elemente zu speichern (*Chaining*) oder sich bei einer Kollision eine andere Hash-Adresse zu berechnen (*Offene Addressierung*).

Hashing ist ein gutes Beispiel für einen Kompromiss zwischen den beiden Gütefaktoren eines Algorithmus (Speicherplatzbedarf und Ausführungszeit). Wenn wir beliebig viel Zeit zur Verfügung hätten, könnten wir einfach alle Elemente sequentiell durchsuchen; und hätten wir beliebig viel Speicherplatz, könnten wir einfach den Schlüsselwert als Array-Index nehmen (beziehungsweise das Ergebnis einer injektiven Hash-Funktion).

Wir wenden uns nun dem Problem zu, eine gute Hash-Funktion zu finden.

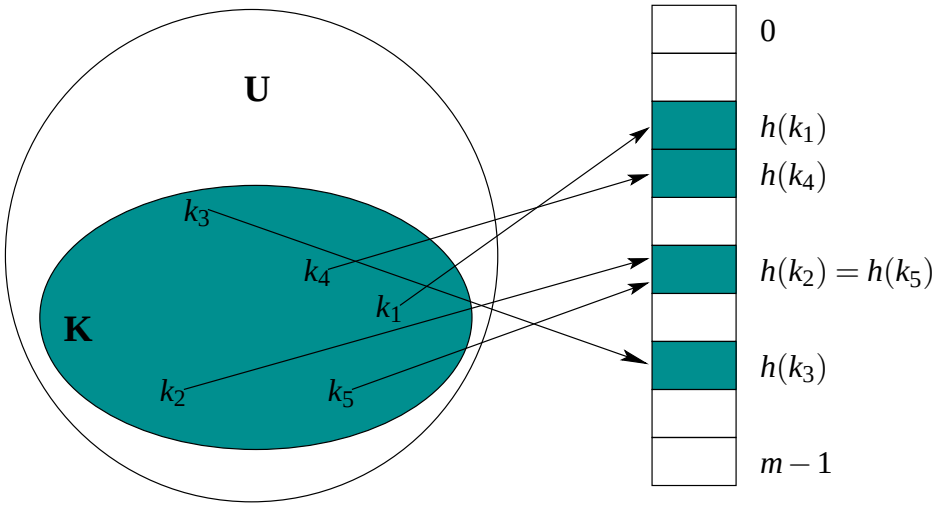

Abbildung 8.1: Hier werden die Elemente mit den Schlüsseln k_2 und k_5 auf den gleichen Eintrag in der Hash-Tabelle abgebildet.

8.1 Hash-Funktionen

Was zeichnet also eine gute Hash-Funktion aus? Sie sollte

- leicht und schnell berechenbar sein
- die Datensätze möglichst gleichmäßig verteilt auf die Hash-Tabelle abbilden
- deterministisch sein (sonst findet man seine Schlüssel ja nicht wieder)
- Kollisionen vermeiden

Es kann ziemlich schwierig sein, eine Hash-Funktion zu finden, die injektiv ist, selbst wenn die Zielmenge kleiner ist als die Ausgangsmenge.

Beispiel 8.1 Es gibt $41^{31} \approx 10^{50}$ verschiedene Funktionen, die von einer 31-elementigen Menge in eine 41-elementige Menge abbilden; injektiv sind davon aber nur $41 \cdot 40 \cdot 39 \cdot \ldots \cdot 11 = 41!/10! \approx 10^{43}$, also nur ungefähr jede 10millionste!

Beispiel 8.2 [Geburtstagsparadoxon] Die Frage beim Geburtstagsparadoxon ist: Wieviele Leute müssen in einem Raum sein, damit die Wahrscheinlichkeit, dass zwei von ihnen am gleichen Tag Geburtstag haben, höher als $\frac{1}{2}$ ist? (Das Jahr hat dabei 365 Tage, wir berücksichtigen keine Schaltjahre.)

Das lässt sich auf folgende Art berechnen:

Sei m die Anzahl der möglichen Tage und k die Anzahl der Personen. Die Wahrscheinlichkeit q, dass es keine Kollision gibt (also, dass alle Personen an verschiedenen Tagen Geburtstag haben), ist dann

$$q(m,k) = \frac{m \cdot (m-1) \cdot (m-2) \cdot (m-3) \cdot \cdots \cdot (m-k+1)}{m \cdot m \cdot m \cdot m \cdot \cdots \cdot m} = \frac{m!}{(m-k)! \cdot m^k}$$

Dann gilt

$$q(365,23) \approx 0,49270276567601459277458277 < \frac{1}{2}$$

Es genügen also schon 23 Leute, damit die Wahrscheinlichkeit, dass zwei von ihnen am gleichen Tag Geburtstag haben, größer als $\frac{1}{2}$ ist.

Beispiel 8.2 übertragen auf Hash-Funktionen bedeutet folgendes: Haben wir ein Universum von 23 Schlüsseln, eine Hash-Tabelle mit 365 Einträgen und eine zufällige Hash-Funktion, so ist die Wahrscheinlichkeit, dass zwei Schlüssel auf den gleichen Eintrag in der Hash-Tabelle abgebildet werden, größer als $\frac{1}{2}$. Und dabei haben wir nur eine Auslastung der Hash-Tabelle von $\frac{23}{365} \approx 6,301\%$.

Wir nehmen im Folgenden an, dass die Schlüsselwerte natürliche Zahlen sind; andernfalls kann man eine bijektive Funktion finden, die von den Schlüsselwerten in eine Teilmenge der natürlichen Zahlen abbildet. (Wir nehmen also an, dass U höchstens abzählbar viele Elemente enthält.)

8.1.1 Divisionsmethode

Bei der Divisionsmethode wird für festes $m \in \mathbb{N}$ folgende Hash-Funktion verwendet:

$$h : U \to \{0,\ldots,m-1\}, \qquad h(k) := k \bmod m$$

In diesem Fall sind einige Werte von m aber besser als andere. Wenn zum Beispiel m gerade ist, so wird $h(k)$ genau dann gerade sein, wenn k schon gerade war. Außerdem sollte man berücksichtigen, dass wir im Binärsystem rechnen. Daher ist es ungünstig, wenn m eine Potenz von 2 ist ($m = 2^p$), weil dann bei der Berechnung des Hash-Werts von k nur die letzten p Bits berücksichtigt werden (natürlich kann man diese Hash-Funktion verwenden, wenn man weiß, dass alle 1-0-Verteilungen in den letzten p Bits gleich wahrscheinlich sind).

Es wird empfohlen, für m eine Primzahl zu verwenden, die keine der Zahlen $r^i \pm j$ teilt, wobei $i, j \in \mathbb{N}$ kleine Zahlen und r die Basis des Zahlensystems ist. Das ist *im Allgemeinen* eine gute Wahl.

8.1.2 Multiplikationsmethode

Sei $0 < A < 1$ beliebig, aber fest. Dann benutzt man bei der Multiplikationsmethode folgende Hash-Funktion:

$$h : U \to \{0,\ldots,m-1\}, \qquad h(k) := \lfloor m(kA \bmod 1)\rfloor = \lfloor m(kA - \lfloor kA\rfloor)\rfloor$$

Es wird also k mit einer Konstante A zwischen 0 und 1 multipliziert, die Vorkommastellen werden abgeschnitten, das Ergebnis wird mit m multipliziert (der dabei entstehende Wert q ist $\in \mathbb{R}$ und es gilt $0 \le q \le m$). Dann wird abgerundet und wir erhalten einen ganzzahligen Wert q' mit $0 \le q' \le m-1$.

Dabei ist die Wahl von m nicht so entscheidend wie bei der Divisionsmethode.

Nach [Knu98] führt dabei eine Wahl von

$$A \approx (\sqrt{5}-1)/2 = 0.6180339887\ldots$$

zur gleichmäßigsten Verteilung von allen Zahlen zwischen 0 und 1.

8.2 Kollisionsbehandlung

8.2.1 Chaining

Beim Chaining steht in der Hash-Tabelle an jeder Stelle eine Liste. Tritt nun beim Einfügen eine Kollision auf, so werden beide Elemente in die Liste an dieser Stelle gehängt. Beim Suchen muss dann — nachdem die Hash-Funktion die Stelle berechnet hat, an der das gesuchte Element zu finden ist — die dortige Liste noch durchsucht werden.

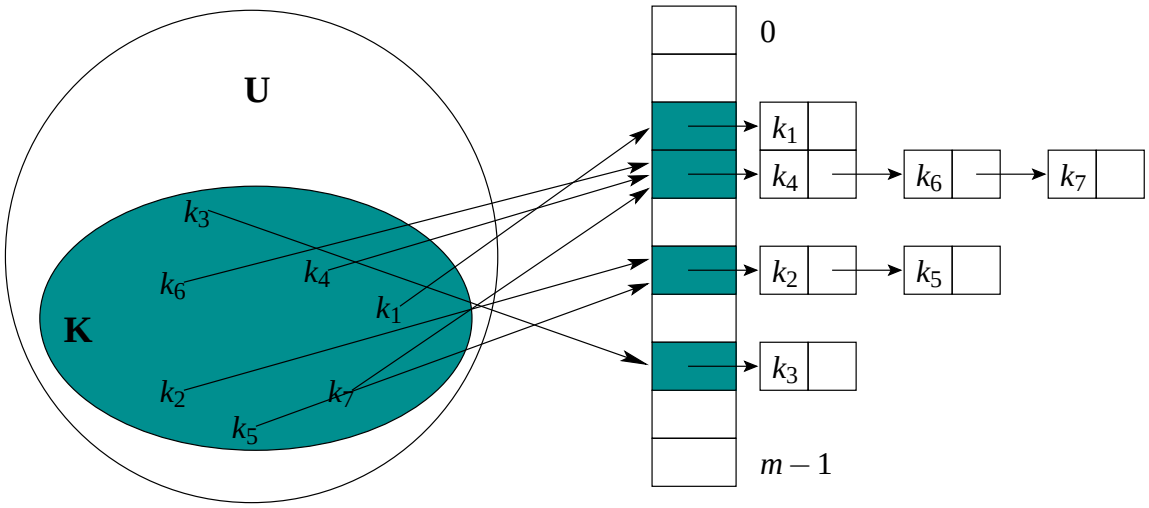

Abbildung 8.2: Beim Chaining enthält jeder Eintrag der Hash-Tabelle eine Liste der Elemente, die auf diese Stelle gehasht wurden.

Sei h die benutzte Hash-Funktion. Wir betrachten die drei Operationen, die wir durchführen wollen:

Einfügen Beim Einfügen können wir annehmen, dass das Element nicht bereits in der Hash-Tabelle T vorhanden ist; falls es nötig sein sollte, das zu überprüfen, können wir vorher das Element suchen. Die Einfügen-Methode hat also die Form

```
chainedHashInsert(T,x)
 füge x am Kopf der Liste T[h(x.key)] ein
```

Die Worst-Case-Laufzeit für das Einfügen ist also $O(1)$.

Suchen `chainedHashSearch(T,k)`

```
 suche in der Liste T[h(k)] nach einem Element x mit x.key==k
```

Die Worst-Case-Laufzeit vom Suchen ist also von der Länge der Liste abhängig.

Löschen `chainedHashDelete(T,x)`

```
lösche x aus der Liste T[h(x.key)]
```

Wenn wir das Element bereits gesucht haben und eine doppelt verkettete Liste verwenden, so können wir es nach der Suche in $O(1)$ löschen.

Wir betrachten jetzt den Aufwand für das Suchen etwas genauer. Dazu geben wir erst einmal eine Definition:

Sei T eine Hash-Tabelle mit $m \in \mathbb{N}$ Plätzen, die gerade $\mathbb{N} \ni n > 0$ Elemente speichert. Dann definieren wir den *Auslastungsfaktor* α der Tabelle als

$$\alpha = \frac{n}{m}$$

Weil beim Chaining auch mehr Elemente gespeichert werden können als die Hash-Tabelle Einträge hat, kann hier α auch größer als 1 sein. Das Worst-Case-Verhalten von Hashing mit Chaining tritt dann auf, wenn alle Elemente auf den gleichen Eintrag der Hash-Tabelle abgebildet werden. Dann hat das Suchen die gleiche Laufzeit wie bei einer einfach verketteten Liste, also $\Theta(n)$. Wir wollen nun das Durchschnittsverhalten von Hashing analysieren. Dazu treffen wir folgende Annahme:

Gleichverteilungsannahme: Die Wahrscheinlichkeit, dass ein Schlüssel k auf die Adresse i in der Hash-Tabelle abgebildet wird, ist unabhängig von i stets gleich $\frac{1}{m}$ und hängt nicht davon ab, welche Elemente bereits in der Tabelle gespeichert sind.

Wir nehmen an, dass der Wert der Hash-Funktion in $O(1)$ berechnet werden kann und die Suchdauer daher von der Länge der Liste dominiert wird. Der Erwartungswert für die Länge der Liste ist unter der Gleichverteilungsannahme offenbar gleich α.

Satz 8.1 *Beim Hashing mit Chaining ist unter der Gleichverteilungsannahme der Aufwand für das Suchen im Mittel $O(1+\alpha)$.*

Beweis: Wegen der Gleichverteilungsannahme wird Schlüssel k mit gleicher Wahrscheinlichkeit auf jeden der Einträge gehasht. Wir können den Wert der Hash-Funktion in $O(1)$ berechnen. Wir speichern die Elemente in der Hash-Tabelle in Listen. Diese haben eine mittlere Länge von α. Daher ist der Aufwand für eine Suche im Mittel α und damit der Aufwand für das Suchen $O(1+\alpha)$. □

Korollar 8.1 *Falls m proportional zu n ist, also $n = O(m)$ gilt, ist der Aufwand für das Suchen im Mittel konstant. Da das Einfügen und Löschen (nach dem Suchen) in $O(1)$ gehen, können daher alle drei Operationen im Mittel in konstanter Zeit ausgeführt werden.*

8.2.2 Offene Adressierung

Bei der offenen Adressierung werden alle Elemente in der Hash-Tabelle selbst gespeichert, ohne dass Listen verwendet werden. Wenn die Hash-Tabelle m Plätze hat, können dann natürlich auch nur maximal m Elemente gespeichert werden.

Die Kollisionsbehandlung erfolgt so, dass für jeden Schlüssel k auf eine bestimme Weise eine Sondierungssequenz zur Suche nach einer Ersatzadresse angegeben wird, die nacheinander abgearbeitet wird, wenn das Element mit Schlüssel k eingefügt beziehungsweise gesucht werden soll.

Falls wir ein Element suchen, das nicht in der Tabelle enthalten ist, suchen wir dabei entweder die ganze Tabelle durch oder können abbrechen, sobald wir in der Sondierungssequenz auf einen Eintrag in der Hash-Tabelle stoßen, der leer ist (wäre das Element schon gespeichert, so wäre es ja an dieser Stelle, weil die Sondierungssequenz für festes k ja jedesmal die gleiche ist).

Wir erweitern also die Hash-Funktion zu einer Funktion

$$h : U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}$$

wobei das zweite Argument die Anzahl der schon erfolgten Sondierungen (beginnend bei 0) sein soll.

Dabei muss gelten:

Permutationsbedingung

$$\boxed{(h(k,0),\, h(k,1),\ldots,h(k,m-1)) \text{ ist eine Permutation von } (0,\, 1,\ldots,m-1).}$$

Folgende Methode fügt ein Element in eine Hash-Tabelle `T` ein. Dabei sei `int h(int, int)` die Hash-Funktion und `T` die Hash-Tabelle, realsiert als Array von `Integer` Objekten.

```
boolean hashInsert(Integer[] T, int k) {
// returns true, if element is successfully inserted, else false
        int i = 0;
        do {
                int j = h(k,i);
                if (T[j] == NULL) {
                        T[j] = new Integer(k);
                        return true;
                }
        i++;
        } while (i!=T.length);
        return false;
}
```

Folgender Algorithmus sucht dann den Datensatz mit Schlüssel k:

```
int hashSearch(Integer[] T, int k) {
// returns index of hash table entry, if element is found, else -1
        int i = 0;
        do {
```

```
                int j = h(k,i);
                if (T[j].equals(k))
                        return j;
                i++;
        }
        while (T[j]!=NULL && i!=m);
        return -1;
}
```

Aus einer Hash-Tabelle zu löschen, die offene Adressierung verwendet, ist im Allgemeinen schwierig. Wir können ein Element nicht einfach löschen, weil dann an dieser Stelle beim Suchen von anderen Elementen die Sondierungssequenz abbrechen könnte und daher andere Elemente nicht mehr gefunden werden, obwohl sie noch in der Tabelle enthalten sind.

Eine mögliche Lösung besteht darin, einen Spezialwert GELÖSCHT zu definieren, den wir statt NULL in die Hash-Tabelle schreiben, damit die Sondierungssequenz nicht abbricht. Man müsste dann Hash-Insert ein bisschen modifizieren, damit dieser Wert dann so behandelt wird, als wäre die Tabelle an dieser Stelle leer.

Das Problem ist, dass beim Verwenden von GELÖSCHT die Suchzeit länger wird und nicht mehr nur vom Auslastungsfaktor α abhängt. Daher wird, falls gelöscht werden muss, meistens Chaining als Kollisionsbehandlung gewählt.

Wir betrachten jetzt drei verschiedene Hash Verfahren, die auf offener Adressierung basieren.

Lineare Sondierung / Linear Probing

Sei $h' : U \to \{0,1,\ldots,m-1\}$ eine gewöhnliche Hash-Funktion. Dann benutzt man beim Linear Probing folgende Hash-Funktion:

$$h(k,i) := \big(h'(k)+i\big) \mod m$$

Die davon generierte Sondierungssequenz hat folgende Form:

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

Ein Problem beim Linear Probing besteht darin, dass sich leicht Ketten von schon belegten Feldern bilden, weil die Wahrscheinlichkeit, dass ein Element an Stelle k eingefügt wird, wobei vor Stelle k schon i belegte Felder sind, gleich $\frac{i+1}{m}$ ist. Das bezeichnet man als *Primäres Clustering*. Dadurch steigt die Durchschnittszeit für das Einfügen und Suchen stark an, wenn sich der Auslastungsfaktor α der 1 nähert.

Das Problem wird besonders drastisch, wenn als Hash-Funktion h' dabei die Divisionsmethode verwendet wird und direkt aufeinanderfolgende Schlüssel $\{k, k+1, k+2, \ldots\}$ eingefügt werden, weil diese dann auch auf Felder gehasht werden, die direkt aufeinanderfolgen.

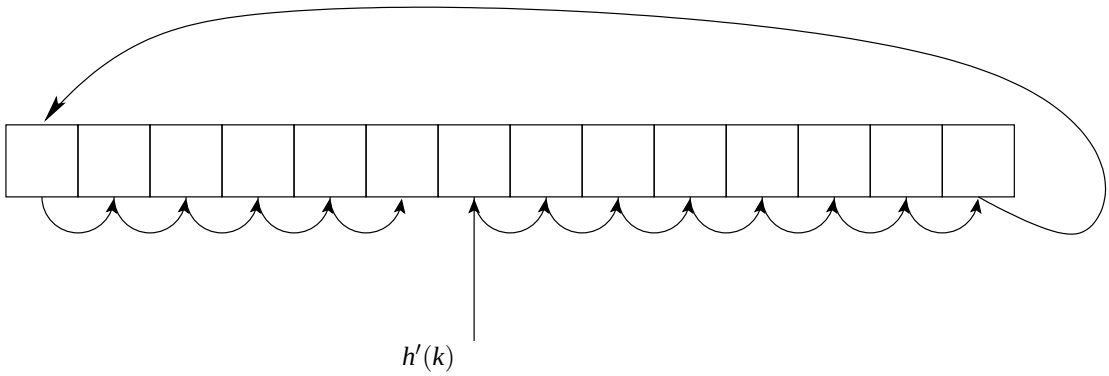

Abbildung 8.3: Lineare Sondierung

Beim Linear Probing gibt es noch eine Möglichkeit, das Löschen so unterzubringen, dass der Aufwand nicht anwächst, wenn man das Löschen geringfügig abändert (siehe [Knu98]). Beim folgenden Verfahren ist das aber nicht mehr praktikabel:

Quadratische Sondierung / Quadratic Probing

Beim Quadratischen Sondieren wird eine Hash-Funktion der Form

$$h(k,i) := \left(h'(k) + c_1 i + c_2 i^2\right) \mod m$$

verwendet.

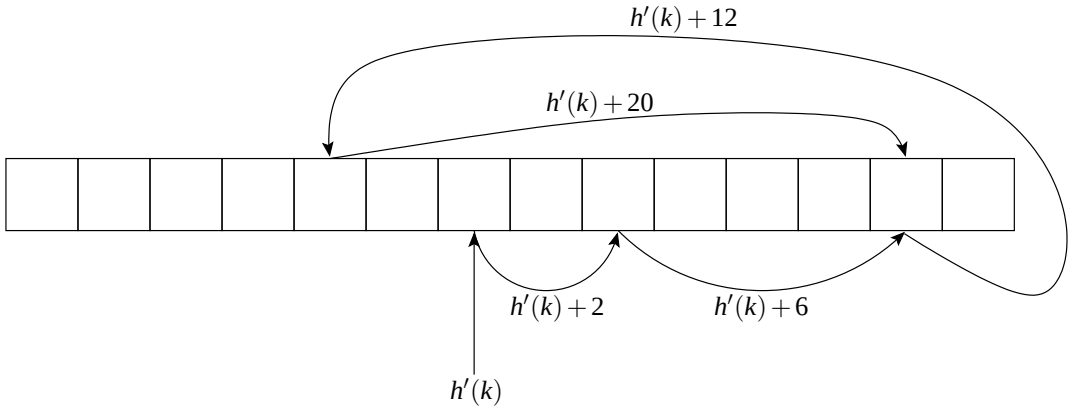

Abbildung 8.4: Quadratische Sondierung mit $c_1 = c_2 = 1$

Dabei ist die Permutationsbedingung kompliziert zu erfüllen, denn sie hängt von c_1, c_2 und m ab. Dieses Hashing-Verfahren vermeidet primäres Clustering, führt aber zu sogenanntem *sekundärem*

Clustering, denn wenn Schlüssel k_1, k_2 den gleichen Hash-Wert haben ($h(k_1,0) = h(k_2,0)$), so haben sie auch die gleiche Sondierungssequenz.

Doppelhash / Double Hashing

Beim Double-Hashing werden zwei Hash-Funktionen verwendet. Dies ist eine der besten Arten, Kollisionen durch offene Addressierung zu behandeln, weil die vom Double Hashing erzeugten Permutationen viele Charakteristika von zufälligen Permutationen besitzen. Die Hash-Funktion hat also die Form

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m$$

wobei h_1 und h_2 einfache Hash-Funktionen sind. Die Verwendung der zweiten Hash-Funktion behebt dabei das Problem des sekundären Clustering.

Satz 8.2 (Doppelhashing) *Die Permutationsbedingung ist beim Doppelhashing genau dann erfüllt, wenn die Länge m der Hash-Tabelle und $h_2(k)$ für alle k relativ prim sind, also gilt:*

$$\forall k: \quad ggT\,(m, h_2(k)) = 1$$

Beispiel 8.3 Wir machen erst einmal plausibel, dass die Permutationsbedingung nicht erfüllt ist wenn m und $h_2(k)$ nicht relativ prim sind. Dazu geben wir ein Gegenbeispiel explizit an.

Seien also $h_2(k) = 8$, $m = 12$. Dann ist $ggT(h_2(k), m) = 4 =: d$. Sei $h_1(k) = 1$. Die Folge der Adressen lautet dann $1, 9, 5, 1$. Es werden dann also nur $\frac{m}{d} = \frac{12}{4} = 3$ Adressen in der Hash-Tabelle besucht.

Wir betrachten diesen Sachverhalt nun genauer.

Beweis: (von Satz 8.2)

Definiere $h_2(k) =: w$ und sei $ggT = (m,w) =: d \in \mathbb{N}$. Dann gibt es $p, q \in \mathbb{N}$ mit

$$p \cdot d = w \tag{8.1}$$

und

$$q \cdot d = m \tag{8.2}$$

Sei o.B.d.A. $h_1(k) = 0$. Die Folge der durchlaufenen Adressen hat dann die Form

$$0, w, 2w, 3w, \ldots, \mathbf{q} \cdot \mathbf{w} \overset{8.1}{=} q \cdot \underbrace{p \cdot d}_{w} = \underbrace{q \cdot d}_{m} \cdot p \overset{8.2}{=} \mathbf{m} \cdot \mathbf{p} \tag{8.3}$$

Ein Vielfaches von m modulo m ist aber wieder 0. Für die Zykellänge ℓ, also die Anzahl der Schritte, bis man zum ersten Mal wieder am Ausgangspunkt angelangt ist, gilt also:

$$\ell \leq q$$

Dann sind zwei Fälle zu unterscheiden:

(1) $\ell = q$

Es werden also q Adressen besucht. Da nach Voraussetzung $m = q \cdot d$ gilt, folgt

$$d = 1 \Leftrightarrow q = m,$$

also die Aussage des Satzes.

(2) $\ell < q$

Nach ℓ Sprüngen wird das erste Mal der Ausgangspunkt wieder besucht. Aber nach q Sprüngen wird auch der Ausgangspunkt wieder besucht, also muss der Zykel der Länge ℓ mehrfach durchlaufen worden sein und damit ist q ein Vielfaches von ℓ. Es gilt also:

$$q = s \cdot \ell, \qquad s > 1 \tag{8.4}$$

Weil nach ℓ Sprüngen der Weite w wieder der Ausgangspunkt erreicht wird, ist also $\ell \cdot w$ ein Vielfaches von m. Es gilt also:

$$\ell \cdot w = r \cdot m, \qquad r \geq 1 \tag{8.5}$$

Also gilt

$$p \cdot m \overset{8.3}{=} q \cdot w \overset{8.4}{=} s \cdot \ell \cdot w \overset{8.5}{=} s \cdot r \cdot m$$
$$\Rightarrow p = s \cdot r \quad \text{und} \quad q = s \cdot \ell$$

Es folgen

$$m \overset{8.2}{=} d \cdot q = d \cdot s \cdot \ell$$

und

$$w \overset{8.1}{=} d \cdot p = d \cdot s \cdot r$$

Aber weil $s > 1$, ist $d \cdot s > d$ und damit teilt nicht nur d, sondern auch ds schon m und w und ist ein größerer Teiler als d. Aber nach Voraussetzung war d der größte gemeinsame Teiler von m und w. Widerspruch! Also tritt (2) nicht auf. □

Analyse unter Gleichverteilungsannahme

Die *Gleichverteilungsannahme* bedeutet hier, dass die nächste Sondierung gleichverteilt unter den m Adressen eine auswählt, also jede mit Wahrscheinlichkeit $\frac{1}{m}$.

Satz 8.3 *Bei Auslastungsfaktor $\alpha = \frac{n}{m} < 1$ ist die erwartete Anzahl der Sondierungen beim Einfügen gleich $\frac{1}{1-\alpha}$.*

Beispiel 8.4 $\alpha = \frac{1}{2} \Rightarrow \frac{1}{1-\alpha} = 2$ Sondierungen (im Mittel), $\alpha = 0{,}8 \Rightarrow \frac{1}{1-\alpha} = 5$ Sondierungen (im Mittel).

Der Beweis beruht auf einer allgemeinen wahrscheinlichkeitstheoretischen Aussage.

Satz 8.4 (Urnenmodell) *Gegeben seien m Kugeln in einer Urne, davon sind w weiß und s schwarz, $w+s=m$. Es werde aus der Urne gleichverteilt mit Zurücklegen gezogen. Dann ist die erwartete Anzahl von Ziehungen bis zur Ziehung einer weißen Kugel $\frac{m}{w}$, also*

$$\mathbb{E}(\#\textit{ Ziehungen bis zur ersten weißen Kugel}) = \frac{m}{w}.$$

Beispiel 8.5 Ein Spezialfall des Urnenmodells ist die mittlere Zahl von Würfeln eines Würfels bis zur ersten 6. Dabei entspricht die 6 einer weißen Kugel und die Zahlen 1 bis 5 schwarzen Kugeln. Also ist $w=1$ und $s=5$. Jede Zahl (Kugel) wird mit derselben Wahrscheinlichkeit von $\frac{1}{6}$ gezogen und das Urnenmodell ergibt

$$\mathbb{E}(\#\text{ Würfe, bis das erste mal eine 6 geworfen wird}) = \frac{6}{1} = 6.$$

Entsprechend ergibt sich

$$\mathbb{E}(\#\text{ Würfe, bis das erste mal eine 1 oder eine 2 geworfen wird}) = \frac{6}{2} = 3.$$

Bevor wir Satz 8.4 beweisen, rekapituliern wir kurz die Definition des Erwartungswertes für abzählbar diskrete Zufallsvariable.

X bezeichne eine diskrete Zufallsgröße, die die Werte $x_1, x_2, \ldots, x_n$ annehmen kann, wobei diese Werte mit den Wahrscheinlichkeiten $p_1, p_2, \ldots, p_n$ auftreten. Der *Erwartungswert* (*Mittelwert*) von X ist definiert als

$$\mathbb{E}(X) := \sum_{i=1}^{n} x_i \cdot p_i$$

Im abzählbar unendlichen Fall ist der *Erwartungswert* definiert als $\mathbb{E}(X) = \sum_{i=1}^{\infty} x_i \cdot p_i$, falls diese Reihe konvergiert.

Im Urnenmodell entspricht z_w dem Ziehen einer weißen, z_s dem Ziehen einer schwarzen Kugel und die Ereignisse treten mit den Wahrscheinlichkeiten p, q ein, wobei

$$p+q=1. \tag{8.6}$$

Sei X die Anzahl der Ziehungen, bis z_w eintritt. X ist dann eine abzählbar unendlich diskrete Zufallsvariable mit den Werten $x_1=1$, $x_2=2, \ldots$, $x_i=i, \ldots$. Diese Werte treten mit folgenden Wahrscheinlichkeiten auf:

$$\begin{array}{lcll}
x_1=1 & \hat{=} & \text{Folge} \quad z_w & \text{mit Wahrscheinlichkeit: } p \\
x_2=2 & \hat{=} & \text{Folge} \quad z_s z_w & \text{mit Wahrscheinlichkeit: } q\cdot p \\
 & \vdots & & \\
x_i=i & \hat{=} & \text{Folge} \quad \underbrace{z_s z_s \ldots z_s}_{(i-1)-\text{mal}} z_w & \text{mit Wahrscheinlichkeit: } q^{i-1}\cdot p =: p_i
\end{array}$$

Zur Kontrolle addieren wir die Wahrscheinlichkeiten noch einmal auf:

$$\sum_{i=1}^{\infty} p_i = \sum_{i=1}^{\infty} q^{i-1} \cdot p = p \cdot \sum_{i=0}^{\infty} q^i = p \cdot \frac{1}{1-q} \overset{8.6}{=} p \cdot \frac{1}{p} = 1$$

Wir stellen nun die Verbindung zwischen dem Sondieren beim Doppelhash und dem Urnenmodell her.

Urnenmodell	Hashing
Ziehen einer weißen Kugel	Hash-Funktion wählt eine leere Stelle in der Hash-Tabelle
Ziehen einer schwarzen Kugel	Hash-Funktion wählt eine besetzte Stelle in der Hash-Tabelle
Ziehen mit Zurücklegen	blindes Wählen eines neuen Platzes

Dann folgt mit Satz 8.4:

$$\mathbb{E}(\#\text{ Sondierungen}) \leq \mathbb{E}(\#\text{ Sondierungen bei Blindwahl}) = \frac{m}{m-n} = \frac{1}{1-\frac{n}{m}} = \frac{1}{1-\alpha},$$

also der Beweis von Satz 8.3.

Wir müssen daher noch die wahrscheinlichkeitstheoretische Aussage in Satz 8.4 zeigen.

Beweis: (von Satz 8.4)
Dazu betrachten wir Folgen von Ziehungen, bis das erste Mal eine weiße Kugel gezogen wird:

1. Mal:	(w)	Wahrscheinlichkeit : p mit $p = \frac{w}{m}$
2. Mal:	(s,w)	Wahrscheinlichkeit : $q \cdot p$ mit $q = \frac{s}{m}$
3. Mal:	(s,s,w)	Wahrscheinlichkeit : $q^2 \cdot p$
$\vdots$	$\vdots$	$\vdots$
i-tes Mal:	$(\underbrace{s,s,\ldots,s}_{(i-1)-\text{mal}},w)$	Wahrscheinlichkeit : $q^{i-1} \cdot p$
$\vdots$	$\vdots$	$\vdots$

Der Erwartungswert für die Anzahl der Ziehungen ergibt sich dann aus der Definition, indem man jeden Wert $x_i = i$ der Zufallsvariablen

$$X := \#\text{ Ziehungen bis zur ersten weißen Kugel}$$

mit der zugehörigen Wahrscheinlichkeit p_i multipliziert und diese Produkte alle aufsummiert. Er ist also

$$\begin{aligned}
\mathbb{E}(X) &= 1 \cdot p + 2 \cdot qp + 3 \cdot q^2 p + \cdots + i \cdot q^{i-1} p + \ldots \\
&= \sum_{i=0}^{\infty} i \cdot q^{i-1} \cdot p \\
&= p \cdot \underbrace{\sum_{i=1}^{\infty} i \cdot q^{i-1}}_{:=S}
\end{aligned}$$

Wir dürfen die Potenzreihe S umordnen, weil sie absolut konvergent ist. Also ist

$$\begin{aligned}
S &= 1\cdot q^0 + 2q^1 + 3q^2 + 4q^3 + \dots \\
&= q^0 + q^1 + q^2 + q^3 + \dots \quad (\to \quad \sum_{i=0}^{\infty} q^i) \\
&\quad + q^1 + q^2 + q^3 \dots \qquad (\to \quad q\cdot\sum_{i=0}^{\infty} q^i) \\
&\quad + q^2 + q^3 + \dots \qquad (\to \quad q^2\cdot\sum_{i=0}^{\infty} q^i) \\
&\quad + q^3 + \dots \qquad (\to \quad q^3\cdot\sum_{i=0}^{\infty} q^i) \\
&= \sum_{i=0}^{\infty} q^i + q\cdot\sum_{i=0}^{\infty} q^i + q^2\cdot\sum_{i=0}^{\infty} q^i + \dots \\
&= (1 + q + q^2 + q^3 + \dots)\sum_{i=0}^{\infty} q^i \\
&= \left(\sum_{i=0}^{\infty} q^i\right)\left(\sum_{i=0}^{\infty} q^i\right) \\
&= \frac{1}{1-q}\cdot\frac{1}{1-q} \\
&= \frac{1}{p}\cdot\frac{1}{p} \\
&= \frac{1}{p^2}
\end{aligned}$$

Also gilt für die erwartete Anzahl $\mathbb{E}(X)$ der Ziehungen bis zur ersten weißen Kugel

$$\begin{aligned}
\mathbb{E}(X) &= p\cdot S \\
&= \frac{1}{p} \\
&\overset{p=\frac{w}{m}}{=} \frac{m}{w}.
\end{aligned}$$

□

Korollar 8.2 *Die Anzahl der Sondierungen ist im Mittel also gleich $\frac{1}{1-\alpha}$ und damit konstant. Also gehen Einfügen und Suchen in Hash-Tabellen im Mittel in $O(1)$.*

Man könnte sich nun die Frage stellen, warum wir nicht Bäume statt Hashtabellen verwenden und auch dort den Aufwand im Mittel betrachten, da sich doch das Löschen bei Bäumen sehr viel einfacher gestaltet, und wir den Vorteil der Sortierung gemäß Inorder-Durchlauf haben. Es gilt jedoch:

Satz 8.5 *Der mittlere Aufwand für das Einfügen und Suchen ist bei Bäumen $O(\log n)$.*

Beweis: Zum Beweis dieses Satzes betrachten wir das Suchen in einem Suchbaum mit n Datensätzen. Der bestmögliche (weil höhenminimale) Baum für diese Suche ist ein voller Baum, also ein Baum, bei dem alle Schichten bis auf die letzte voll sind (falls $n = 2^p - 1$ mit $p \in \mathbb{N}$, so ist auch die letzte voll). Betrachte dabei einen Baum T, der auf der untersten Schicht nur ein Element enthält, so dass

$$n = \sum_{i=0}^{n-1} 2^i + 1 = 2^h - 1 + 1 = 2^h$$

gilt. Wir analysieren das Suchen im Baum T unter der Gleichverteilungsannahme, in diesem Fall also der Annahme, dass jeder Schlüssel mit der gleichen Wahrscheinlichkeit $\frac{1}{n}$ gesucht wird.

Dann gilt:

$$\begin{aligned}
\mathbb{E}(\#\text{ Anzahl Vergleiche zum Suchen}) &= \frac{\text{Summe der Vergleiche zum Suchen aller Knoten}}{n} \\
&\geq \frac{h \cdot \#\text{Knoten auf Schicht (h-1)}}{n} \\
&= \frac{1}{n} \cdot h \cdot 2^{h-1} \\
&= \frac{1}{n} \cdot \log n \cdot \frac{n}{2} \\
&= \frac{1}{2} \log n \\
&\in \Omega(\log n)
\end{aligned}$$

□

Hash-Tabellen haben also den Vorteil eines *konstanten mittleren Aufwandes* für die Basisoperationen gegenüber Suchbäumen. Suchbäume haben dagegen einen Worst Case Aufwand von $O(\log n)$ für die Basisoperationen und beinhalten per Inorder-Durchlauf eine Sortierung der Datensätze. Eshängt also sehr von der Anwendung ab, ob Hash-Tabellen oder Suchbäume geeigneter sind.

8.3 Literaturhinweise

Hashtabellen werde in nahezu allen Büchern über Datenstrukturen und Algorithmen behandelt. Die hier gewählte Darstellung lehnt sich an [CLRS01] an.